University of California Berkeley

Computer Science 162

Operating Systems and Systems Programming

Course Reader for Spring 2008

Professor: Anthony D. Joseph

Contents:

CS162
Operating Systems and Systems Programming
Lecture 1

What is an Operating System?

January 23, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Who am I?

- Professor Anthony D. Joseph
  - 465 Soda Hall (RAD Lab)
  - adj AT cs.berkeley.edu
  - Office hours M 1pm/Tu 2pm in 413 Soda
- Background:
  - MIT undergrad and grad student
- Research areas:
  - Current: Network security, OS security, building a large security testbed, attacks against machine learning algorithms
  - Other: Mobile computing, wireless networking, cellular telephony

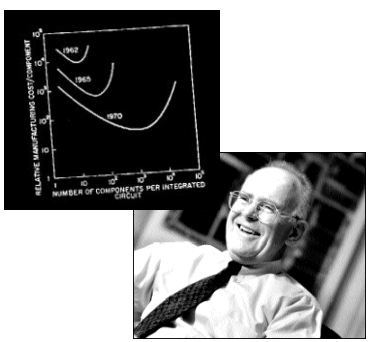
Goals for Today

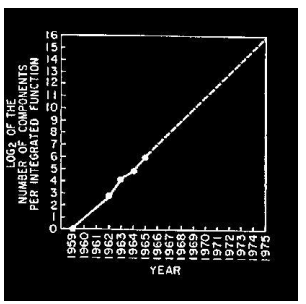
- What is an Operating System?
  - And – what is it not?
- Examples of Operating Systems design
- Why study Operating Systems?
- Oh, and "How does this class operate?"

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.

Rapid Underlying Technology Change



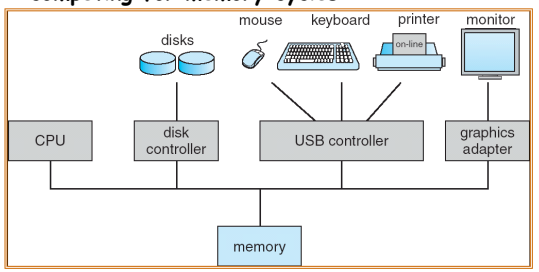


- "Cramming More Components onto Integrated Circuits"
  - Gordon Moore, Electronics, 1965

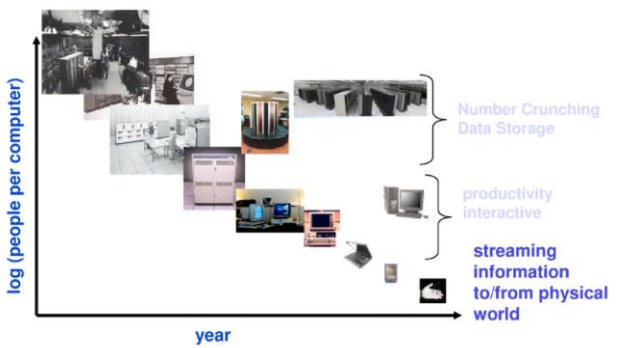
Computing Devices Everywhere

Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles

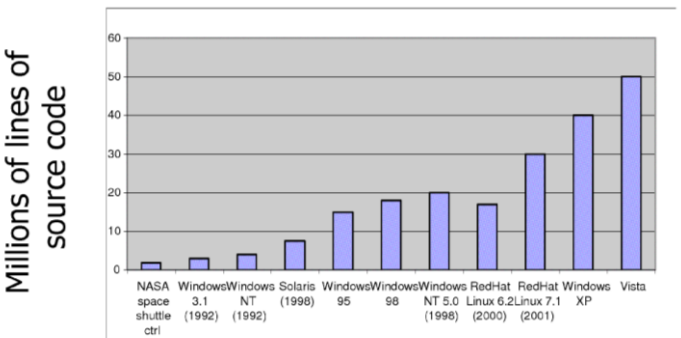


People-to-Computer Ratio Over Time



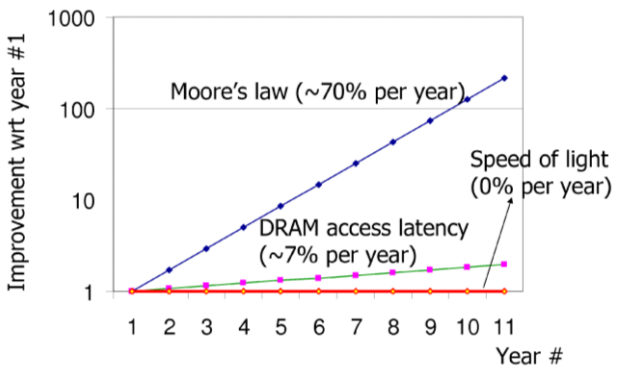
From David Culler

Increasing Software Complexity



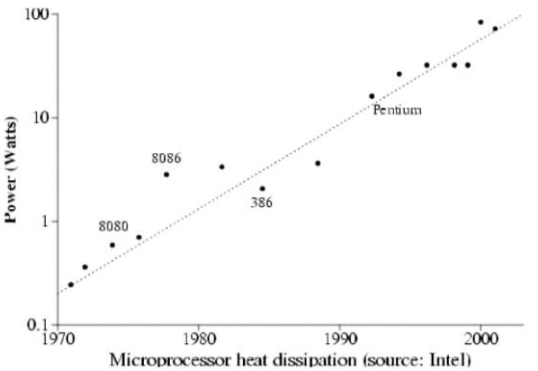
From MIT's 6.033 course

But, Latency Improves Slowly…



From MIT's 6.033 course

Heat is a Major Problem!



From MIT's 6.033 course

Complexity

- How to manage complexity at all levels?
- Many issues and many tradeoffs
- Need a global view of systems
  - Decompose into components
- Need a global understanding of systems
  - Applications, networks, databases, operating systems, security, software engineering…

Example: Some Mars Rover Requirements

- Serious hardware limitations/complexity:
  - 20Mhz powerPC processor, 128MB of RAM
  - cameras, scientific instruments, batteries, solar panels, and locomotion equipment
  - Many independent processes work together
- Can't hit reset button very easily!
  - Must reboot itself if necessary
  - Always able to receive commands from Earth
- Individual Programs must not interfere
  - Suppose the MUT (Martian Universal Translator Module) buggy
  - Better not crash antenna positioning software!
- Further, all software may crash occasionally
  - Automatic restart with diagnostics sent to Earth
  - Periodic checkpoint of results saved?
- Certain functions time critical:
  - Need to stop before hitting something
  - Must track orbit of Earth for communication

How do we tame complexity?

- Every piece of computer hardware different
  - Different CPU
    - » Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memory, disk, …
  - Different types of devices
    - » Mice, Keyboards, Sensors, Cameras, Fingerprint readers
  - Different networking environment
    - » Cable, DSL, Wireless, Firewalls,…
- Questions:
  - Does the programmer need to write a single program that performs many independent activities?
  - Does every program have to be altered for every piece of hardware?
  - Does a faulty program crash everything?
  - Does every program have access to all hardware?

OS Tool: Virtual Machine Abstraction

Application

Virtual Machine Interface

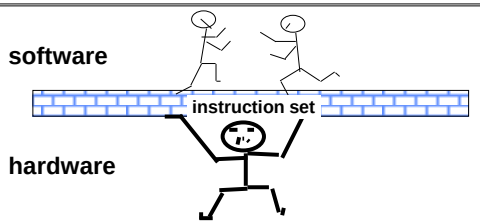
Operating System

Physical Machine Interface

Hardware

- Software Engineering Problem:
  - Turn hardware/software quirks ⇒ what programmers want/need
  - Optimize for convenience, utilization, security, reliability, etc…
- For Any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What's the hardware interface? (physical reality)
  - What's the application interface? (nicer abstraction)

Interfaces Provide Important Boundaries



- Why do interfaces look the way that they do?
  - History, Functionality, Stupidity, Bugs, Management
  - CS152 ⇒ Machine interface
  - CS160 ⇒ Human interface
  - CS169 ⇒ Software engineering/management
- Should responsibilities be pushed across boundaries?
  - RISC architectures, Graphical Pipeline Architectures

Course Administration

- Instructor: Anthony D. Joseph (adj@cs)
  465 Soda Hall (RAD Lab)
  Office Hours (TBA): 413 Soda Hall
- TAs: Barret Rhoden (cs162-tj@cory)
  Manu Srivastava (cs162-tk@cory)
  Man-Kit Leung (cs162-tl@cory)
- Labs: Second floor of Soda Hall (poll)
- Website: http://inst.eecs.berkeley.edu/~cs162
- Webcast/Podcast (3-day delay): http://webcast.berkeley.edu/courses/index.php
- Newsgroup: ucb.class.cs162 (use authnews.berkeley.edu)
- Course Email: cs162@cory
- Reader: Available from TBA
- Are you on the waitlist? See Michael-David in 379 Soda

Class Schedule

- Class Time: M/W 4 – 5:30pm, 277 Cory
  - Please come to class. Lecture notes do not have everything in them. The best part of class is the interaction!
- Sections:
  - Important information is in the sections
  - The sections assigned to you by Telebears are temporary!
  - Every member of a project group must be in same section

Section	Time	Location	TA
101	Th 10:00-11:00A	45 Evans	Barret
102	Th 11:00-12:00P	85 Evans	Barret
103	Th 4:00-5:00P	3102 Etcheverry	Man-Kit
104	F 2:00-3:00P	310 Soda	Manu
105	*F 3:00-4:00p*	*405 Soda*	*Manu*

Textbook

- Text: Operating Systems Concepts, 7th Edition Silbershatz, Galvin, Gagne

- Online supplements
  - See "Information" link on course website
  - Includes Appendices, sample problems, etc
- Question: need 7th edition?
  - No, but has new material that we may cover
  - Completely reorganized
  - Will try to give readings from both the 6th and 7th editions on the lecture page

Topic Coverage

Textbook: Silberschatz, Galvin, and Gagne, *Operating Systems Concepts*, 7th Ed., 2005

- 1 week: Fundamentals (Operating Systems Structures)
- 1.5 weeks: Process Control and Threads
- 2.5 weeks: Synchronization and scheduling
- 2 week: Protection, Address translation, Caching
- 1 week: Demand Paging
- 1 week: File Systems
- 2.5 weeks: Networking and Distributed Systems
- 1 week: Protection and Security
- 1 week: Software Engineering
- ??: Advanced topics

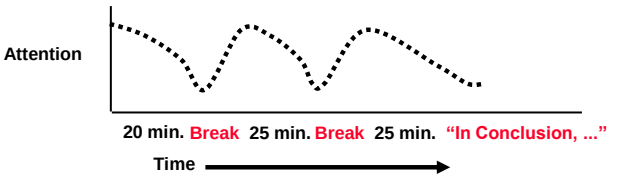
Grading

- Rough Grade Breakdown
  - Two Midterms: 15% each
    One Final: 15%
    Four Projects: 50% (i.e. 12.5% each)
    Participation: 5%
- Four Projects:
  - Phase I: Build a thread system
  - Phase II: Implement Multithreading
  - Phase III: Caching and Virtual Memory
  - Phase IV: Parallel and Distributed Systems
- Late Policy:
  - Each group has 5 "slip" days.
  - For Projects, slip days deducted from *all* partners
  - 10% off per day after slip days exhausted

Group Project Simulates Industrial Environment

- Project teams have 4 or 5 members in same discussion section
  - Must work in groups in "the real world"
- Communicate with colleagues (team members)
  - Communication problems are natural
  - What have you done?
  - What answers you need from others?
  - You must document your work!!!
  - Everyone must keep an on-line notebook
- Communicate with supervisor (TAs)
  - How is the team's plan?
  - Short progress reports are required:
    - » What is the team's game plan?
    - » What is each member's responsibility?

Typical Lecture Format



- 1-Minute Review
- 20-Minute Lecture
- 5- Minute Administrative Matters
- 25-Minute Lecture
- 5-Minute Break (water, stretch)
- 25-Minute Lecture
- Instructor will come to class early & stay after to answer questions

Lecture Goal

Interactive!!!

Computing Facilities

- Every student who is enrolled should get an account form at end of lecture
  - Gives you an account of form cs162-xx@cory
  - This account is required
    - » Most of your debugging can be done on other EECS accounts, however…
    - » All of the final runs must be done on your cs162-xx account and must run on the x86 Solaris machines
- Make sure to log into your new account this week and fill out the questions
- Project Information:
  - See the "Projects and Nachos" link off the course home page
- Newsgroup (ucb.class.cs162):
  - Read this regularly!

Academic Dishonesty Policy

- Copying all or part of another person's work, or using reference material not specifically allowed, are forms of cheating and will not be tolerated. A student involved in an incident of cheating will be notified by the instructor and the following policy will apply:

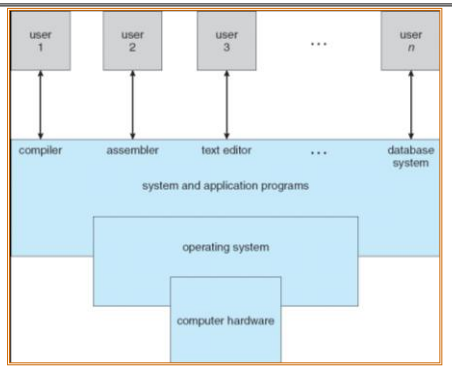
  http://www.eecs.berkeley.edu/Policies/acad.dis.shtml

- The instructor may take actions such as:
  - require repetition of the subject work,
  - assign an F grade or a 'zero' grade to the subject work,
  - for serious offenses, assign an F grade for the course.
- The instructor must inform the student and the Department Chair in writing of the incident, the action taken, if any, and the student's right to appeal to the Chair of the Department Grievance Committee or to the Director of the Office of Student Conduct.
- The Office of Student Conduct may choose to conduct a formal hearing on the incident and to assess a penalty for misconduct.
- The Department will recommend that students involved in a second incident of cheating be dismissed from the University.

Virtual Machines

- Software emulation of an abstract machine
  - Make it look like hardware has features you want
  - Programs from one hardware & OS on another one
- Programming simplicity
  - Each process thinks it has all memory/CPU time
  - Each process thinks it owns all devices
  - Different Devices appear to have same interface
  - Device Interfaces more powerful than raw hardware
    - » Bitmapped display ⇒ windowing system
    - » Ethernet card ⇒ reliable, ordered, networking (TCP/IP)
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine
- Protection and Portability
  - Java interface safe and stable across many platforms

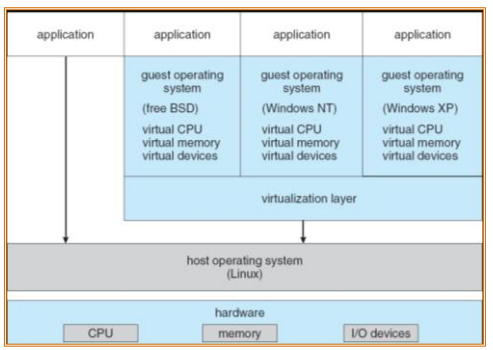
Four Components of a Computer System



Definition: *An operating system implements a virtual machine that is (hopefully) easier and safer to program and use than the raw hardware.*

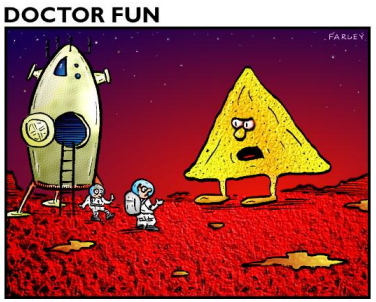
Virtual Machines (con't): Layers of OSs

- Useful for OS development
  - When OS crashes, restricted to one VM
  - Can aid testing programs on other OSs



Nachos: Virtual OS Environment

- You will be working with Nachos
  - Simulation environment
  - Hardware, interrupts, I/O
  - Execution of User Programs running on this platform



"This is the planet where nachos rule."

What does an Operating System do?

- Silerschatz and Gavin:
  "An OS is Similar to a government"
  - Begs the question: does a government do anything useful by itself?
- Coordinator and Traffic Cop:
  - Manages all resources
  - Settles conflicting requests for resources
  - Prevent errors and improper use of the computer
- Facilitator:
  - Provides facilities that everyone needs
  - Standard Libraries, Windowing systems
  - Make application programming easier, faster, less error-prone
- Some features reflect both tasks:
  - E.g. File system is needed by everyone (Facilitator)
  - But File system must be Protected (Traffic Cop)

What is an Operating System,… Really?

- Most Likely:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications? (Does Email belong in OS?)
  - Multitasking/multiprogramming?
- What about?
  - File System?
  - Multimedia Support?
  - User Interface?
  - Internet Browser? ☺
- Is this only interesting to Academics??

Operating System Definition (Cont.)

- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is good approximation
  - But varies wildly
- "The one program running at all times on the computer" is the kernel.
  - Everything else is either a system program (ships with the operating system) or an application program

What if we didn't have an Operating System?

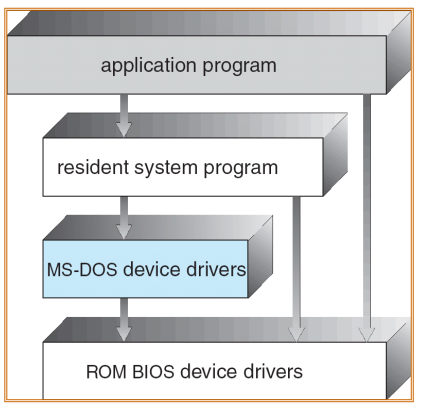
- Source Code⇒Compiler⇒Object Code⇒Hardware
- How do you get object code onto the hardware?
- How do you print out the answer?
- Once upon a time, had to Toggle in program in binary and read out answer from LED's!

Altair 8080

Simple OS: What if only one application?

- Examples:
  - Very early computers
  - Early PCs
  - Embedded controllers (elevators, cars, etc)
- OS becomes just a library of standard services
  - Standard device drivers
  - Interrupt handlers
  - Math libraries

MS-DOS Layer Structure



More thoughts on Simple OS

- What about Cell-phones, Xboxes, etc?
  - Is this organization enough?
- Can OS be encoded in ROM/Flash ROM?
- Does OS have to be software?
  - Can it be Hardware?
  - Custom Chip with predefined behavior
  - Are these even OSs?

More complex OS: Multiple Apps

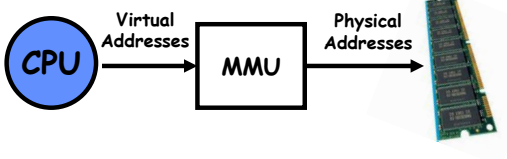
- Full Coordination and Protection
  - Manage interactions between different users
  - Multiple programs running simultaneously
  - Multiplex and protect Hardware Resources
    - » CPU, Memory, I/O devices like disks, printers, etc
- Facilitator
  - Still provides Standard libraries, facilities
- Would this complexity make sense if there were only one application that you cared about?

Example: Protecting Processes from Each Other

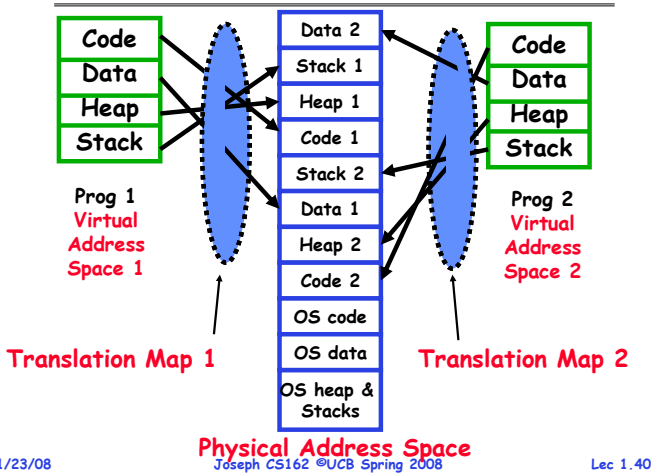
- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)

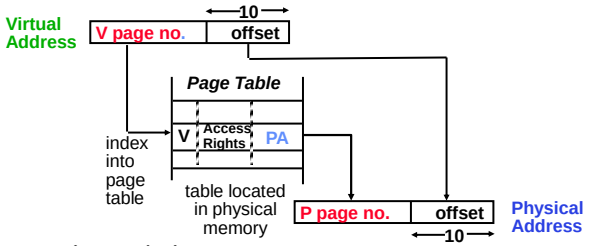


Example of Address Translation



Address Translation Details

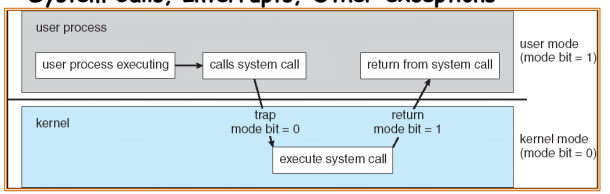
- For now, assume translation happens with table (called a Page Table):



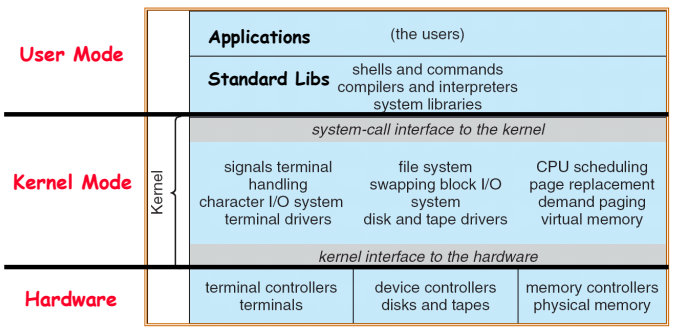
- Translation helps protection:
  - Control translations, control access
  - Should Users be able to change Page Table???

Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - » Attempt to modify ⇒ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



UNIX System Structure



OS Systems Principles

- OS as illusionist:
  - Make hardware limitations go away
  - Provide illusion of dedicated machine with infinite memory and infinite processors
- OS as government:
  - Protect users from each other
  - Allocate resources efficiently and fairly
- OS as complex system:
  - Constant tension between simplicity and functionality or performance
- OS as history teacher
  - Learn from past
  - Adapt as hardware tradeoffs change

Why Study Operating Systems?

- Learn how to build complex systems:
  - How can you manage complexity for future projects?
- Engineering issues:
  - Why is the web so slow sometimes? Can you fix it?
  - What features should be in the next mars Rover?
  - How do large distributed systems work? (Kazaa, etc)
- Buying and using a personal computer:
  - Why different PCs with same CPU behave differently
  - How to choose a processor (Opteron, Itanium, Celeron, Pentium, Hexium)? [Ok, made last one up]
  - Should you get Windows XP, Vista, Linux, Mac OS …?
  - Why does Microsoft have such a bad name?
- Business issues:
  - Should your division buy thin-clients vs PC?
- Security, viruses, and worms
  - What exposure do you have to worry about?

"In conclusion…"

- Operating systems provide a virtual machine abstraction to handle diverse hardware
- Operating systems coordinate resources and protect users from each other
- Operating systems simplify application development by providing standard services
- Operating systems can provide an array of fault containment, fault tolerance, and fault recovery
- CS162 combines things from many other areas of computer science –
  - Languages, data bases, data structures, hardware, networking, security, distributed systems, and algorithms

CS162
Operating Systems and Systems Programming
Lecture 2

Concurrency: Processes, Threads, and Address Spaces

January 28, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Virtual Machine Abstraction

Application

Virtual Machine Interface

Operating System

Physical Machine Interface

Hardware

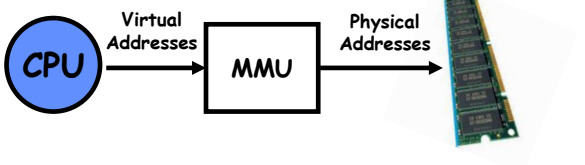
- Software Engineering Problem:
  - Turn hardware/software quirks ⇒ what programmers want/need
  - Optimize for convenience, utilization, security, reliability, etc…
- For Any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What's the hardware interface? (physical reality)
  - What's the application interface? (nicer abstraction)

Example: Protecting Processes from Each Other

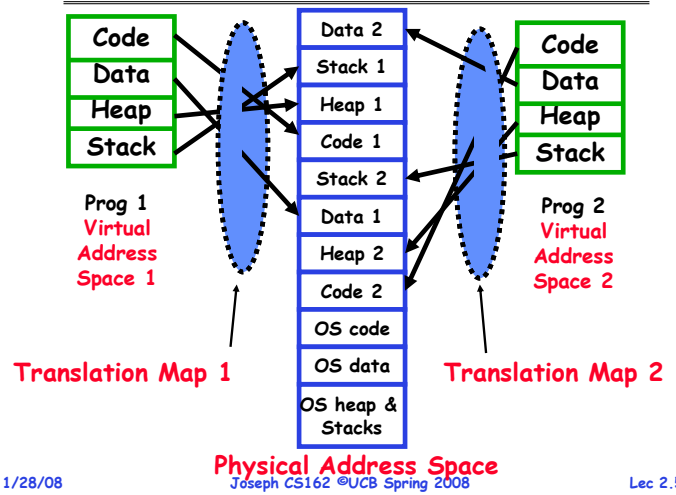
- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Example: Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)

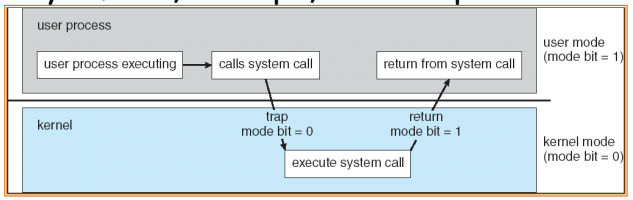


Example: Example of Address Translation



Example: Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - » Attempt to modify ⇒ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



Goals for Today

- How do we provide multiprogramming?
- What are Processes?
- How are they related to Threads and Address Spaces?

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

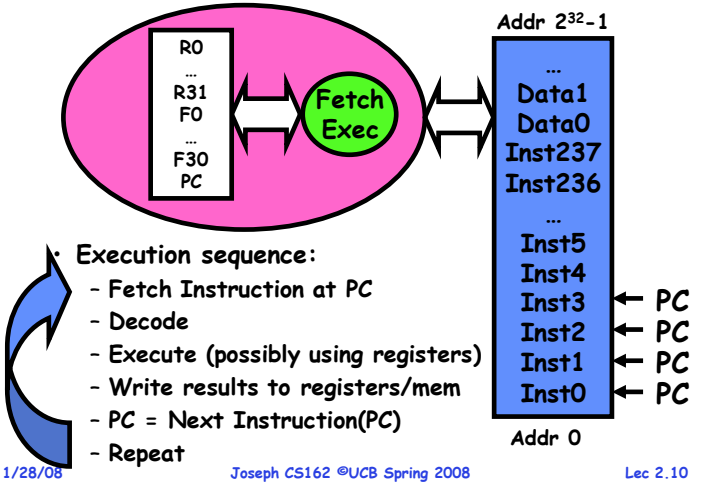
Concurrency

- "Thread" of execution
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- Uniprogramming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by defining it away
  - Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  - Often called "multitasking", but multitasking has other meanings (talk about this later)

The Basic Problem of Concurrency

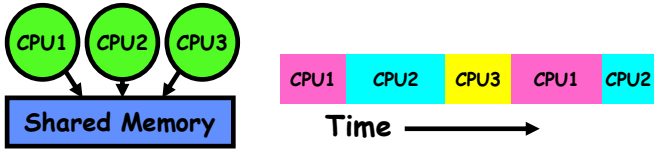
- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to machine
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Decompose hard problem into simpler ones
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines
- Dijkstra did this for the "THE system"
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

Recall (61C): What happens during execution?



- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

How can we give the illusion of multiple processors?



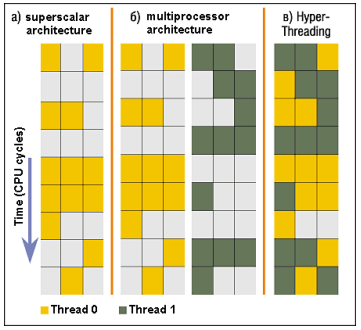
- How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)
  - Windows 95—ME? (switch with both yield and timer)

Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first
- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5



Administrivia

- cs162-xx accounts:
  - Make sure you got an account form
    - » We have more forms for those of you who didn't get one
  - If you haven't logged in and registered yet, you need to do so now
- Nachos readers:
  - TBA: Will be down at Copy Central on Hearst
  - Will include lectures and printouts of all of the code
- Video/Audio archives available off lectures page
  - Just click on the title of a lecture for webcast
  - Three-day delay on Webcasts and Podcasts
- No slip days on first design document for each phase
  - Need to get design reviews in on time
- Don't know Java well?
  - Talk CS 9G self-paced Java course

Administriva: Almost Time for Project Signup

- Project Signup: Use "Group/Section Assignment Link"
  - 4-5 members to a group
    - » Everyone in group must be able to *actually* attend same section
    - » The sections assigned to you by Telebears are temporary!
  - Only submit once per group!
    - » Everyone in group must have logged into their cs162-xx accounts once before you register the group
    - » Make sure that you select at least 2 potential sections
    - » Due date: Thursday (1/31) by 11:59pm
- Sections:
  - Go to desired section this week (Thurs/Fri)

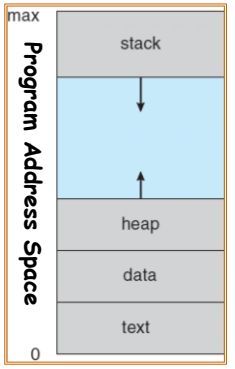
Section	Time	Location	TA
101	Th 10:00-11:00A	45 Evans	Barret
102	Th 11:00-12:00P	85 Evans	Barret
103	Th 4:00-5:00P	3102 Etcheverry	Man-Kit
104	F 2:00-3:00P	310 Soda	Manu
105	*F 3:00-4:00p*	*405 Soda*	*Manu*

How to protect threads from one another?

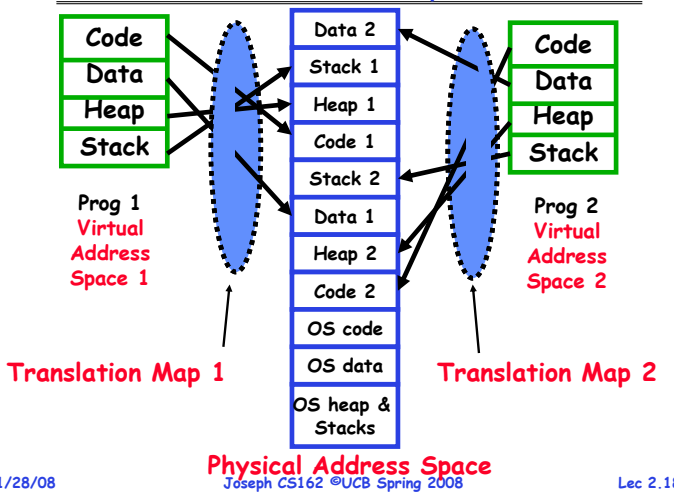
- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Preemptive switching from task to task
     - » Use of timer
     - » Must not be possible to disable timer from usercode

Recall: Program's Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)



Providing Illusion of Separate Address Space: Load new Translation Map on Switch

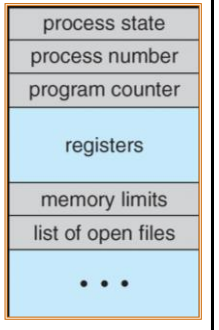


Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Often called a "HeavyWeight Process"
  - Formally: a single, sequential stream of execution in its *own* address space
- Two parts:
  - Sequential Program Execution Stream
    - » Code executed as a *single, sequential* stream of execution
    - » Includes State of CPU registers
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- Important: There is no concurrency in a heavyweight process

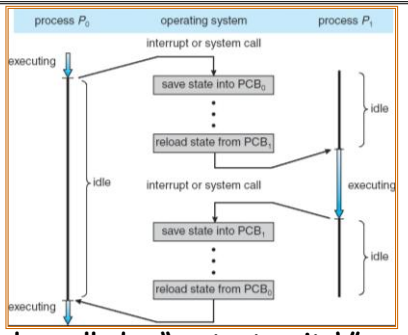
How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



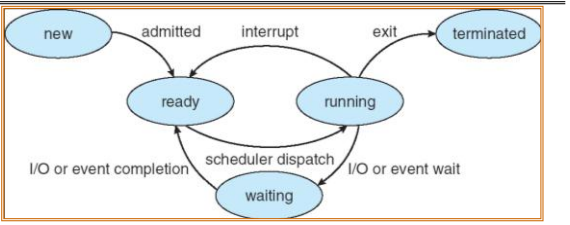
Process Control Block

CPU Switch From Process to Process



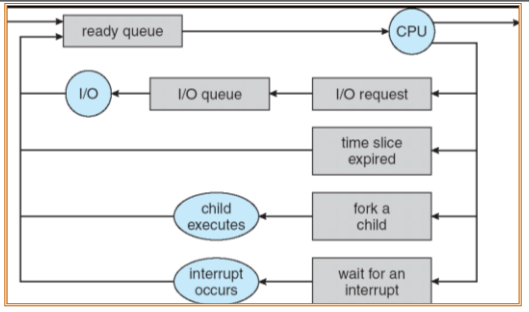
- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Diagram of Process State



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Scheduling

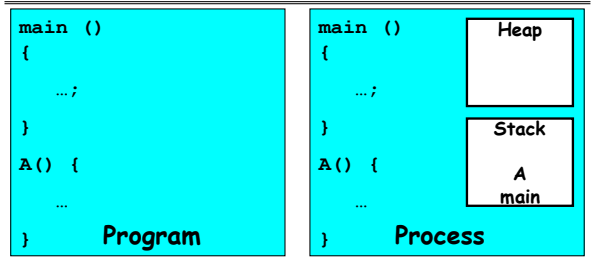


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

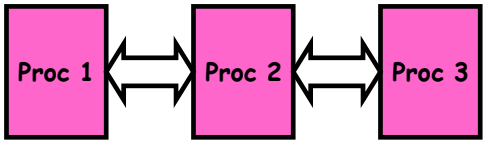
- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive
  - Much less expensive with "copy on write"
- Copy I/O state (file handles, etc)
  - Medium expense

Process =? Program



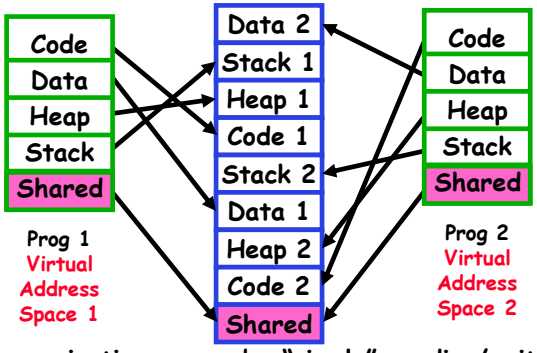
- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

Multiple Processes Collaborate on a Task



- High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - » Accomplished by mapping addresses to common DRAM
    - » Read and Write through memory
  - Message Passing
    - » `send()` and `receive()` messages
    - » Works across network

Shared Memory Communication



- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

BREAK

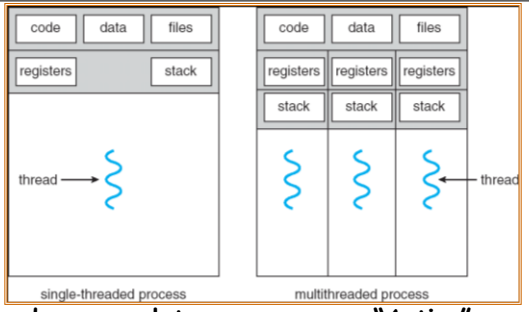
Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus, systcall/trap)
  - logical (e.g., logical properties)

Modern "Lightweight" Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process ≡ Process with one thread

Single and Multithreaded Processes



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Examples of multithreaded programs (con't)

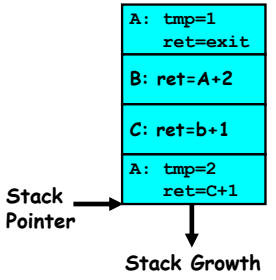
- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, Temporary variables
  - return PCs are kept while called procedures are executing

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Classification

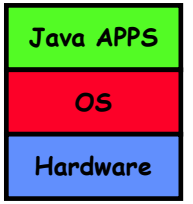
# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

Example: Implementation Java OS

- Many threads, one Address Space
- Why another OS?
  - Recommended Minimum memory sizes:
    - » UNIX + X Windows: 32MB
    - » Windows 98: 16-32MB
    - » Windows NT: 32-64MB
    - » Windows 2000/XP: 64-128MB
  - What if we want a cheap network point-of-sale computer?
    - » Say need 1000 terminals
    - » Want < 8MB
- What language to write this OS in?
  - C/C++/ASM? Not terribly high-level. Hard to debug.
  - Java/Lisp? Not quite sufficient – need direct access to HW/memory management

Java OS Structure



Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process

CS162
Operating Systems and Systems Programming
Lecture 3

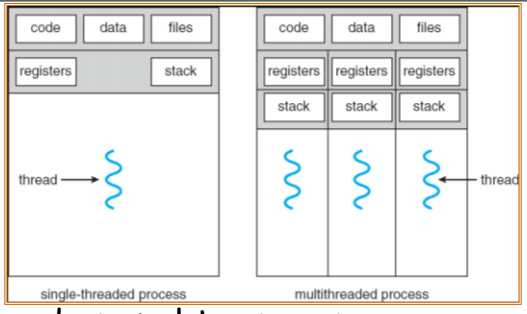
Thread Dispatching

January 30, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Recall: Modern Process with Multiple Threads

- Process: *Operating system abstraction to represent what is needed to run a single, multithreaded program*
- Two parts:
  - Multiple Threads
    - » Each thread is a *single, sequential stream of execution*
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process ≡ Process with one thread

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

Recall: Classification

# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux, Win 95?, Mac OS X, Win NT to XP, Solaris, HP-UX

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

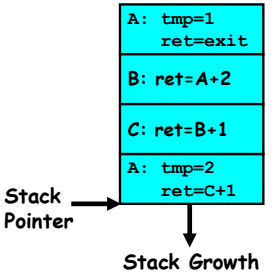
Goals for Today

- Further Understanding Threads
- Thread Dispatching
- Beginnings of Thread Scheduling

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Recall: Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
...		(callee can clobber)
15	t7	
16	s0	callee saves
...		(callee must save)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

- Before calling procedure:
  - Save caller-saves regs
  - Save v0, v1
  - Save ra
- After return, assume
  - Callee-saves reg OK
  - gp,sp,fp OK (restored!)
  - Other things trashed

Single-Threaded Example

- Imagine the following C program:

```
main() {
   ComputePI("pi.txt");
   PrintClassList("clist.text");
}
```

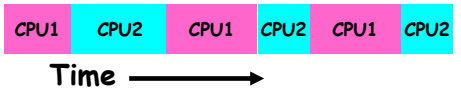
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads:

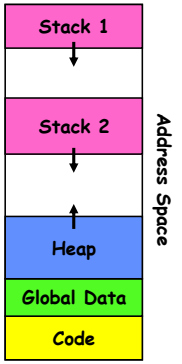
```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- What does "CreateThread" do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Memory Footprint of Two-Thread Example

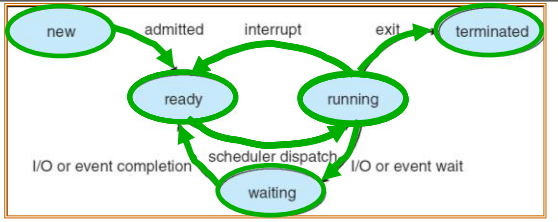
- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- In Nachos: "Thread" is a class that includes the TCB
- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

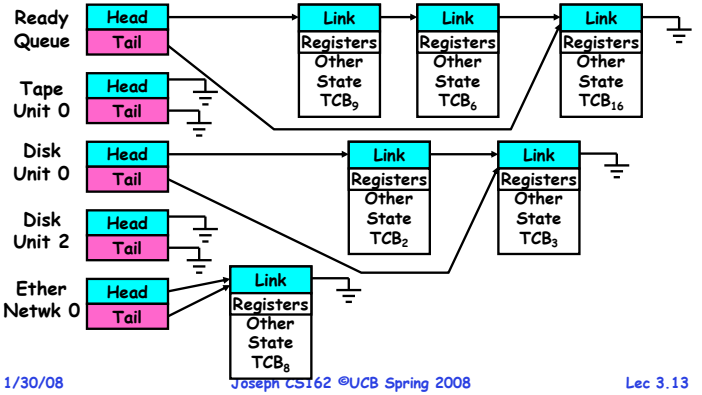
Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Administriva: Time for Project Signup

- Project Signup: Watch "Group/Section Assignment Link"
  - 4-5 members to a group
    - » Everyone in group must be able to *actually* attend same section
    - » The sections assigned to you by Telebears are temporary!
  - Only submit once per group!
    - » Everyone in group must have logged into their cs162-xx accounts once before you register the group
    - » Make sure that you select at least 2 potential sections
    - » Due date: Thursday (1/31) by 11:59pm
- Sections:
  - Go to desired section this week (Thurs/Fri)

Section	Time	Location	TA
101	Th 10:00-11:00A	45 Evans	Barret
102	Th 11:00-12:00P	85 Evans	Barret
103	Th 4:00-5:00P	3102 Etcheverry	Man-Kit
104	F 2:00-3:00P	310 Soda	Manu
105	*F 3:00-4:00p*	*405 Soda*	*Manu*

Administrivia (2)

- Cs162-xx accounts:
  - Make sure you got an account form
  - If you haven't logged in yet, you need to do so
- Email addresses
  - We need an email address from you
  - If you haven't given us one already, you should get prompted when you log in again (or type "register")
- Nachos reader: Required!
  - Available at Copy Central at corner of Hearst&Euclid
  - Includes lectures and printouts of all of the code
- Next Week: Start Project 1
  - Go to Nachos page and start reading up
  - Note that all the Nachos code is printed in your reader

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

Running a thread

Consider first portion: `RunThread()`

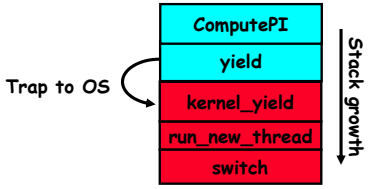
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```


Stack for Yielding Thread



- How do we run a new thread?

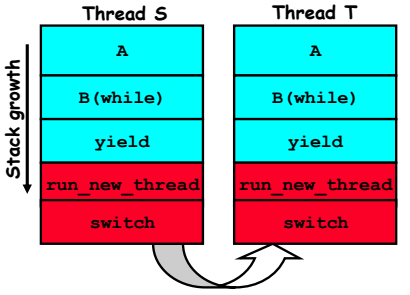
```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* next Lecture */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

What do the stacks look like?

- Consider the following code blocks:

```
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```



- Suppose we have 2 threads:
  - Threads S and T

Saving/Restoring state (often called "Context Switch)

```
Switch(tCur,tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
            …
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
            …
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
```

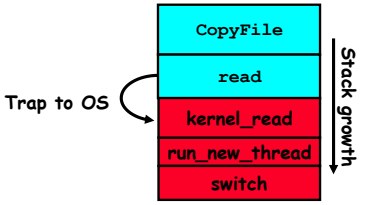

Switch Details

- How many registers need to be saved/restored?
  - MIPS 4k: 32 Int(32b), 32 Float(32b)
  - Pentium: 14 Int(32b), 8 Float(80b), 8 SSE(128b),…
  - Sparc(v7): 8 Regs(32b), 16 Int regs (32b) * 8 windows = 136 (32b)+32 Float (32b)
  - Itanium: 128 Int (64b), 128 Float (82b), 19 Other(64b)
- `retpc` is where the return should jump to.
  - In reality, this is implemented as a jump
- There is a real implementation of switch in Nachos.
  - See switch.s
    - » Normally, switch is implemented as assembly!
  - Of course, it's magical!
  - But you should be able to follow it!

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings
- Cautionary tail:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented!
    - » Only works As long as kernel size < 1MB
  - What happened?
    - » Time passed, People forgot
    - » Later, they added features to kernel (no one removes features!)
    - » Very weird behavior started happening
  - Moral of story: Design for simplicity

What happens when thread blocks on I/O?

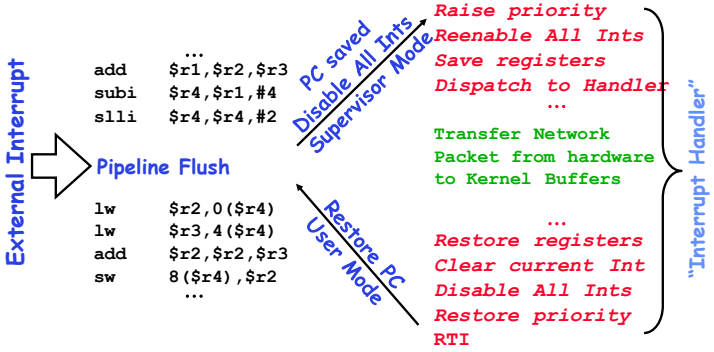


- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!
- Answer: Utilize External Events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some many milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

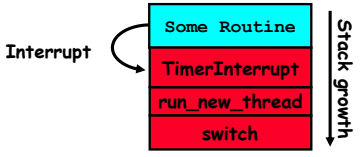
Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- I/O interrupt: same as timer interrupt except that `DoHousekeeping()` replaced by `ServiceIO()`.

Choosing a Thread to Run

- How does Dispatcher decide what to run?
  - Zero ready threads – dispatcher loops
    - » Alternative is to create an "idle thread"
    - » Can put machine into low-power mode
  - Exactly one ready thread – easy
  - More than one ready thread: use scheduling priorities
- Possible priorities:
  - LIFO (last in, first out):
    - » put ready threads on front of list, remove from front
  - Pick one at random
  - FIFO (first in, first out):
    - » Put ready threads on back of list, pull them from front
    - » This is fair and is what Nachos does
  - Priority queue:
    - » keep ready list sorted by TCB priority field

Summary

- The state of a thread is contained in the TCB
  - Registers, PC, stack pointer
  - States: New, Ready, Running, Waiting, or Terminated
- Multithreading provides simple illusion of multiple CPUs
  - Switch registers and stack to dispatch new thread
  - Provide mechanism to ensure dispatcher regains control
- Switch routine
  - Can be very expensive if many registers
  - Must be very carefully constructed!
- Many scheduling options
  - Decision of which thread to run complex enough for complete lecture

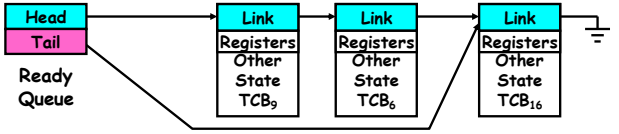
CS162
Operating Systems and Systems Programming
Lecture 4

Cooperating Threads

February 4, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
  - In Arrays, or Linked Lists, or …



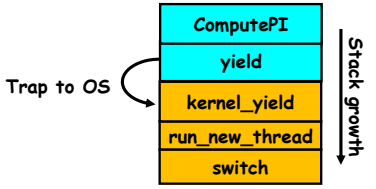
Review: Yielding through Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

  - Note that `yield()` must be called by programmer frequently enough!

Review: Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Later in lecture */
}
```

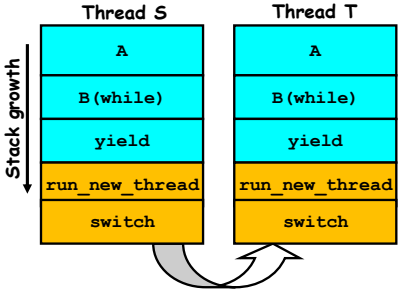
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

Review: Two Thread Yield Example

- Consider the following code blocks:

```
proc A() {
   B();

}
proc B() {
   while(TRUE) {
      yield();
   }
}
```



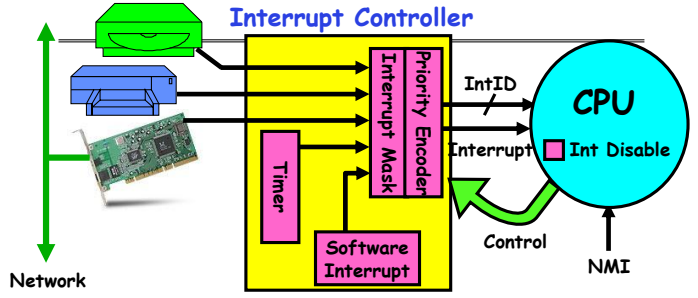
- Suppose we have 2 threads:
  - Threads S and T

Goals for Today

- More on Interrupts
- Thread Creation/Destruction
- Cooperating Threads

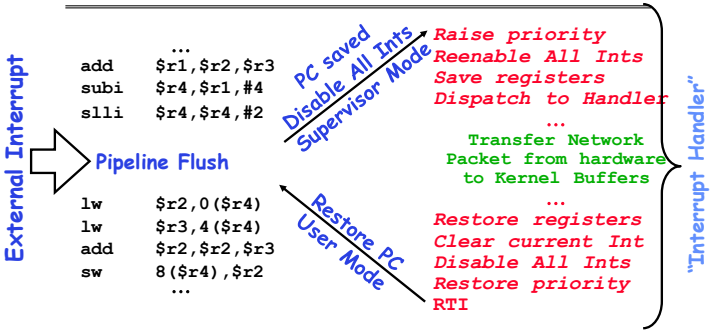
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

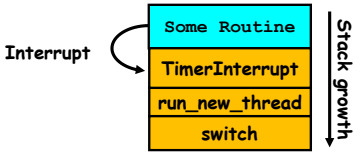
Example: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions

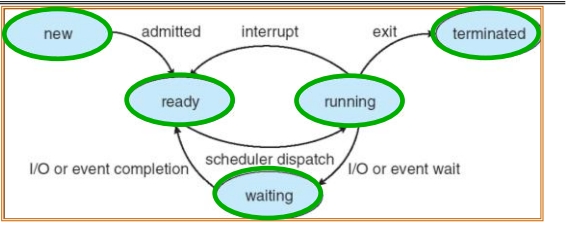


- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- This is often called preemptive multithreading, since threads are prempted for better scheduling
  - Solves problem of user who doesn't insert yield();

Review: Lifecycle of a Thread (or Process)



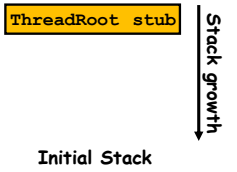
- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

ThreadFork(): Create a New Thread

- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue
  - We called this CreateThread() earlier
- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable).

How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine ThreadRoot()
  - Two arg registers initialized to fcnPtr and fcnArgPtr
- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of ThreadRoot() really gets started

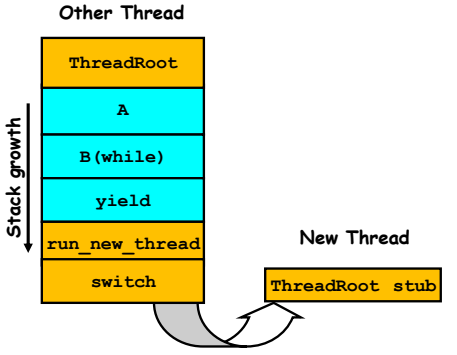


Initial Stack

Administrivia

- Information about Subversion on Handouts page
  - Make sure to take a look
- Other things on Handouts page
  - Synchronization examples/Interesting papers
  - Previous finals/solutions
- Sections in this class are mandatory
  - Make sure that you go to the section that you have been assigned!
  - Sections will be up off the home page
    - » Make sure to respond to Barret if he contacts you
    - » He is attempting to fix section assignments
- Reader still TBA
- Should be reading Nachos code by now!
  - Start working on the first project
  - Set up regular meeting times with your group
  - Try figure out group interaction problems early on

How does Thread get started?



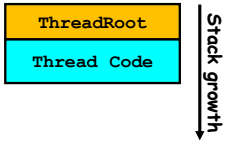
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

What does `ThreadRoot()` look like?

- ThreadRoot() is the root for the thread routine:

```
ThreadRoot() {
   DoStartupHousekeeping();
   UserModeSwitch(); /* enter user mode */
   Call fcnPtr(fcnArgPtr);
   ThreadFinish();
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() will start at user-level



What does `ThreadFinish()` do?

- Needs to re-enter kernel mode (system call)
- "Wake up" (place on ready queue) threads waiting for this thread
  - Threads (like the parent) may be on a wait queue waiting for this thread to finish
- Can't deallocate thread yet
  - We are still running on its stack!
  - Instead, record thread as "waitingToBeDestroyed"
- Call `run_new_thread()` to run another thread:

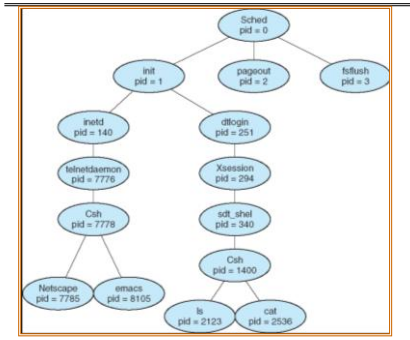
```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, newThread);
   ThreadHouseKeeping();
}
```

  - `ThreadHouseKeeping()` notices waitingToBeDestroyed and deallocates the finished thread's TCB and stack

Additional Detail

- Thread Fork is not the same thing as UNIX fork
  - UNIX fork creates a new *process* so it has to create a new address space
  - For now, don't worry about how to create and switch between address spaces
- Thread fork is very much like an asynchronous procedure call
  - Runs procedure in separate thread
  - Calling thread doesn't wait for finish
- What if thread wants to exit early?
  - `ThreadFinish()` and `exit()` are essentially the same procedure entered at user level

Parent-Child relationship

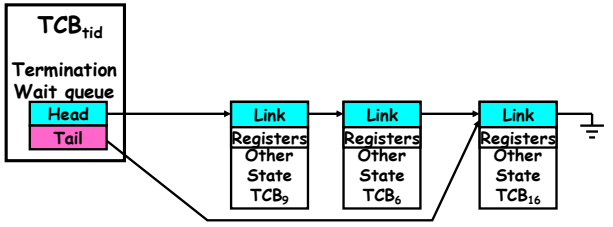


Typical process tree for Solaris system

- Every thread (and/or Process) has a parentage
  - A "parent" is a thread that creates another thread
  - A child of a parent was created by that parent

ThreadJoin() system call

- One thread can wait for another to finish with the `ThreadJoin(tid)` call
  - Calling thread will be taken off run queue and placed on waiting queue for thread `tid`
- Where is a logical place to store this wait queue?
  - On queue inside the TCB



- Similar to `wait()` system call in UNIX
  - Lets parents wait for child processes

Use of Join for Traditional Procedure Call

- A traditional procedure call is logically equivalent to doing a ThreadFork followed by ThreadJoin
- Consider the following normal procedure call of B() by A():

```
A() { B(); }
B() { Do interesting, complex stuff }
```

- The procedure A() is equivalent to A'():

```
A'() {
    tid = ThreadFork(B,null);
    ThreadJoin(tid);
}
```

- Why not do this for every procedure?
  - Context Switch Overhead
  - Memory Overhead for Stacks

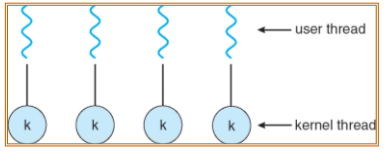
BREAK

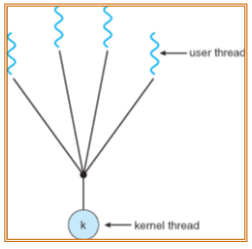
Kernel versus User-Mode threads

- We have been talking about Kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads

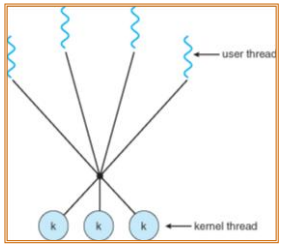
Threading models mentioned by book

Simple One-to-One Threading Model





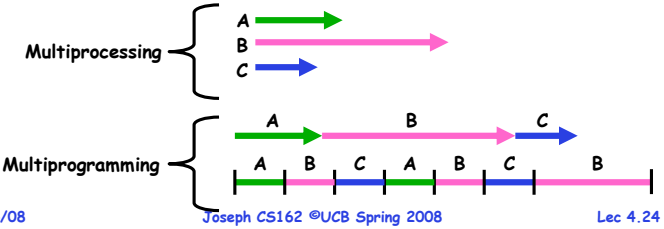
Many-to-One



Many-to-Many

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs or cores
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

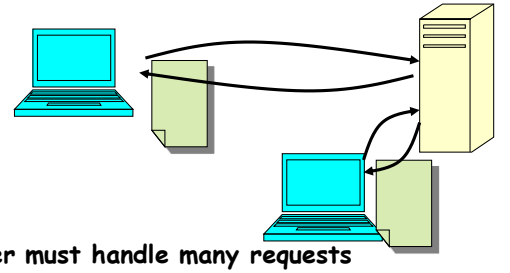
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multi-proc/-core – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

- What are some disadvantages of this technique?

Threaded Web Server

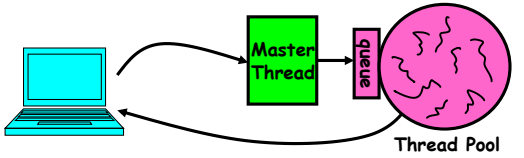
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
}
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block…
- What about Denial of Service attacks or digg / Slash-dot effects?

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
    allocThreads(worker,queue);
    while(TRUE) {
        con=AcceptCon();
        Enqueue(queue,con);
        wakeUp(queue);
    }
}
```

```
worker(queue) {
    while(TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```


Summary

- Interrupts: hardware mechanism for returning control to operating system
  - Used for important/high-priority events
  - Can force dispatcher to schedule a different thread (premptive multithreading)
- New Threads Created with `ThreadFork()`
  - Create initial TCB and stack to point at `ThreadRoot()`
  - `ThreadRoot()` calls thread code, then `ThreadFinish()`
  - `ThreadFinish()` wakes up waiting threads then prepares TCB/stack for distruction
- Threads can wait for other threads using `ThreadJoin()`
- Threads may be at user-level or kernel level
- Cooperating threads have many potential advantages
  - But: introduces non-reproducibility and non-determinism
  - Need to have Atomic operations

CS162
Operating Systems and Systems Programming
Lecture 5

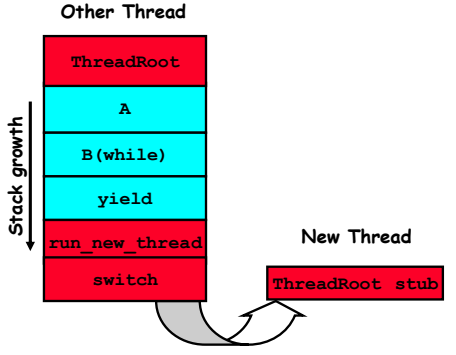
Synchronization

February 6, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: `ThreadFork()`: Create a New Thread

- `ThreadFork()` is a user-level procedure that creates a new thread and places it on ready queue
- Arguments to `ThreadFork()`
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable).

Review: How does Thread get started?



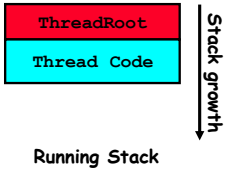
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

Review: What does `ThreadRoot()` look like?

- ThreadRoot() is the root for the thread routine:

```
ThreadRoot() {
  DoStartupHousekeeping();
  UserModeSwitch(); /* enter user mode */
  Call fcnPtr(fcnArgPtr);
  ThreadFinish();
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads



Review: Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Goals for Today

- Concurrency examples
- Need for synchronization
- Examples of valid synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

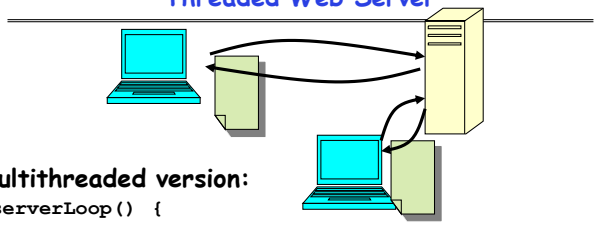
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

Threaded Web Server



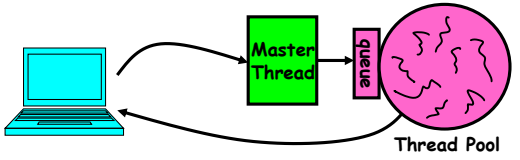
- Multithreaded version:

```
serverLoop() {
      connection = AcceptCon();
      ThreadFork(ServiceWebPage(),connection);
  }
```

- Advantages of threaded version:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- What if too many requests come in at once?

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of threads, representing the maximum level of multiprogramming



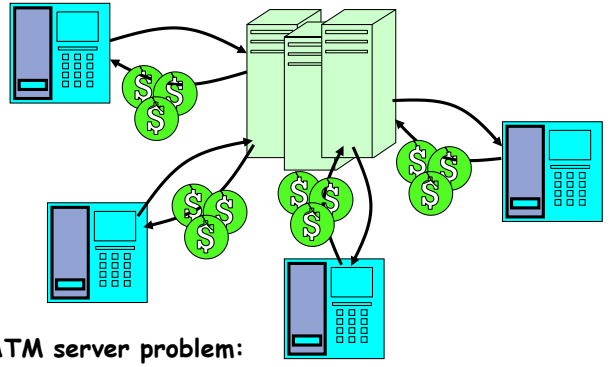

```
master() {
   allocThreads(slave,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
slave(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```


Administrivia

- Sections in this class are mandatory
  - Go to the section that you have been assigned
  - Some topics will only appear in section!
- Should be working on first project
  - Make sure to be reading Nachos code
  - First design document due next Thursday! (Feb 14th)
  - Set up regular meeting times with your group
  - Let's get group interaction problems solved early
- Notice problems with the webcast?
  - Support email: webcast@media.berkeley.edu
- If you need to know more about synchronization primitives before I get to them use book!
  - Chapter 6 (in 7th edition) and Chapter 7 (in 6th edition) are all about synchronization

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
  - This technique is used for graphical programming

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without "deconstructing" code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

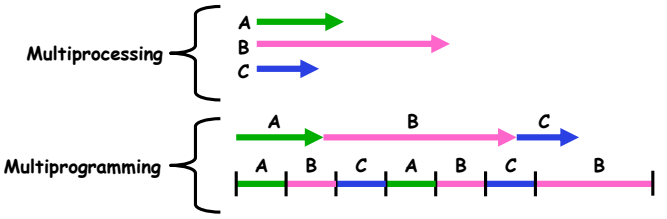
```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);       /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
load r1, acct->balance	
	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

Review: Multiprocessing vs Multiprogramming

- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



- Also recall: Hyperthreading
  - Possible to interleave threads on a per-instruction basis
  - Keep this in mind for our examples (like multiprocessing)

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

- However, What about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

  - What are the possible values of x?
- Or, what are the possible values of x below?

Thread A	Thread B
x = 1;	x = 2;

  - X could be 1 or 2 (non-deterministic!)
  - Could even be 3 for serial processors:
    - » Thread A writes 0001, B writes 0010.
    - » Scheduling order ABABABBA yields 3!

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - » Software control of electron accelerator and electron beam/ Xray production
    - » Software control of dosage
  - Software errors caused the death of several patients
    - » A series of race conditions on shared variables and poor software design
    - » "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

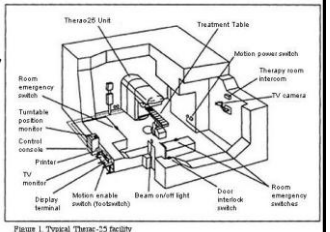
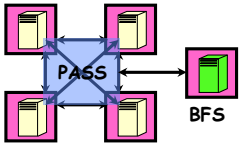


Figure 1. Typical Therac-25 facility

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - » Asynchronous and real-time
    - » Runs all of the control systems
    - » Results synchronized and compared every 3 to 4 ms
  - The Fifth computer is the "Backup Flight System" (BFS)
    - » stays synchronized in case it is needed
    - » Written by completely different team than PASS
- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS
    - » A delayed init request placed into timer queue
    - » As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation



Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A	Thread B
i = 0;	i = 0;
while (i < 10)	while (i > -10)
i = i + 1;	i = i - 1;
printf("A wins!");	printf("B wins!");

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What it both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

BREAK

Motivation: "Too much milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ

  - Of Course – We don't know how to make a lock yet

Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem???
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```

- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
   }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

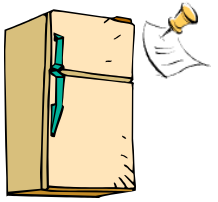
```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.

Where are we going with synchronization?

Programs	Shared Programs			
Higher-level API	Locks	Semaphores	Monitors	Send/Receive
Hardware	Load/Store	Disable Ints	Test&Set	Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!

CS162
Operating Systems and Systems Programming
Lecture 6

Mutual Exclusion, Semaphores, Monitors, and Condition Variables

February 11, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Synchronization problem with Threads

- One thread per transaction, each running:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);       /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
load r1, acct->balance	
	load r1, acct->balance
	add r1, amount2
	store r1, acct->balance
add r1, amount1	
store r1, acct->balance	

- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle

Review: Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A	Thread B
leave note A;	leave note B;
while (note B) {\\X	if (noNote A) {\\Y
do nothing;	if (noMilk) {
}	buy milk;
if (noMilk) {	}
buy milk;	}
}	remove note B;
remove note A;	

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Review: Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Goals for Today

- Hardware Support for Synchronization
- Higher-level Synchronization Abstractions
  - Semaphores, monitors, and condition variables
- Programming paradigms for concurrent programs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider "too much milk" example
  - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- Today, we'll implement higher-level operations on top of atomic operations provided by hardware
  - Develop a "synchronization toolbox"
  - Explore some common programming paradigms

Review: Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.

How to implement Locks?

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
    - » Should *sleep* if waiting for a long time
- Atomic Load/Store: get solution like Milk #3
  - Looked at this last lecture
  - Pretty complex and error prone
- Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
    - » Done in the Intel 432
    - » Each feature makes hardware more complex and slow
  - What about putting a task to sleep?
    - » How do you handle the interface between the hardware and scheduler?

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - » Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - » "Reactor about to meltdown. Help?"

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if (anyone on wait queue) {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = FREE;
  }
  enable interrupts;
}
```


New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

Critical Section

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Interrupt re-enable in going to sleep

- What about re-enabling ints when going to sleep?

Enable Position
Enable Position
Enable Position

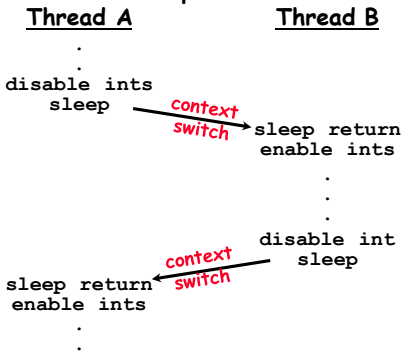
```
Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

Administrivia

- First Design Document due Thursday 1/14
  - Subsequently need to schedule design review with TA (through web form)
  - Note that most of the design document grade comes from first version (some from final version)
- Design doc contents:
  - Architecture, correctness constraints, algorithms, pseudocode, testing strategy, and test case types
- SVN group accounts are setup
  - Password-based access: SVN over WebDAV (SSL)
  - Check out the SVN Quick Start Guide for instructions on how to get your SVN repository working with Eclipse

How to Re-enable After Sleep()?

- In Nachos, since ints are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



Interrupt disable and enable across context switches

- An important point about structuring code:
  - In Nachos code you will see lots of comments about assumptions made concerning when interrupts disabled
  - This is an example of where modifications to and assumptions about program state can't be localized within a small body of code
  - In these cases it is possible for your program to eventually "acquire" bugs as people modify code
- Other cases where this will be a concern?
  - What about exceptions that occur after lock is acquired? Who releases the lock?

```
mylock.acquire();
a = b / 0;
mylock.release()
```


Atomic Read-Modify-Write instructions

- Problems with previous solution:
  - Can't give lock implementation to users
  - Doesn't work well on multiprocessor or multi-core CPU
    - » Disabling interrupts on all processors/cores requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (not too hard)
    - » and multiprocessors/multi-core (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on uniprocessors, multiprocessors, and multi-core CPUs

Examples of Read-Modify-Write

```
• test&set (&address) {      /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
• swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
         M[address] = reg2;
         return success;
      } else {
         return failure;
      }
  }
• load-linked&store conditional(&address) {
      /* R4000, alpha */
      loop:
         ll r1, M[address];
         movi r2, 1;           /* Can do arbitrary comp */
         sc r2, M[address];
         beqz r2, loop;
  }
```


Implementing Locks with test&set

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
  while (test&set(value)); // while busy
}
Release() {
  value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Busy-Waiting: thread consumes cycles while waiting

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor/multi-core
- Negatives
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should not have busy-waiting!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
  // Short busy-wait time
  while (test&set(guard));
  if (value == BUSY) {
     put thread on wait queue;
     go to sleep() & guard = 0;
  } else {
     value = BUSY;
     guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
     take thread off wait queue
     Place on ready queue;
  } else {
     value = FREE;
  }
  guard = 0;
}
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a couple of ways of structuring the sharing

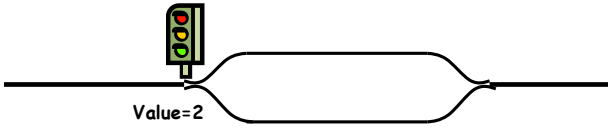
BREAK

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called "Binary Semaphore".
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```


Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example 1: GCC compiler
  - cpp | cc1 | cc2 | as | ld
- Example 2: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - Semaphore fullBuffers; // consumer's constraint
  - Semaphore emptyBuffers;// producer's constraint
  - Semaphore mutex; // mutual exclusion

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```

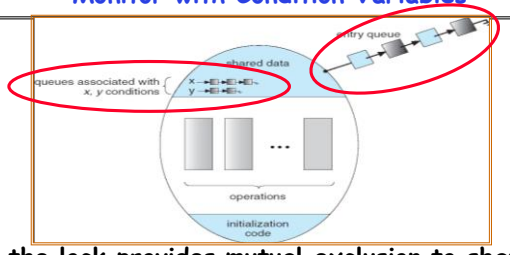
Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
  - Problem is that semaphores are dual purpose:
    - » They are used for both mutex and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitor with Condition Variables



- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Simple Monitor Example

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();           // Get Lock
   queue.enqueue(item);      // Add item
   dataready.signal();       // Signal any waiters
   lock.Release();           // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();           // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();           // Release Lock
   return(item);
}
```

Summary

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
    - » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Talked about Semaphores, Monitors, and Condition Variables
  - Higher level constructs that are harder to "screw up"

CS162
Operating Systems and Systems Programming
Lecture 7

Readers-Writers
Language Support for Synchronization

February 13, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - Only time can set integer directly is at initialization time
- P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
  - Think of this as the wait() operation
- V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
  - This of this as the signal() operation

Review: Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```


Review: Discussion about Solution

- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except for scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Nothing changes!
- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints

Review: Monitors and Condition Variables

- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- The lock provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

Goals for Today

- Continue with Synchronization Abstractions
  - Monitors and condition variables
- Readers-Writers problem and solution
- Language Support for Synchronization
- An Overview of ACID properties in a DBMS

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

AddToQueue(item) {
   lock.Acquire();         // Lock shared data
   queue.enqueue(item);    // Add item
   lock.Release();         // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();         // Lock shared data
   item = queue.dequeue();// Get next item or null
   lock.Release();         // Release Lock
   return(item);           // Might return null
}
```


Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();           // Get Lock
   queue.enqueue(item);      // Add item
   dataready.signal();       // Signal any waiters
   lock.Release();           // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();           // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();           // Release Lock
   return(item);
}
```


Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

  - Why didn't we do this?

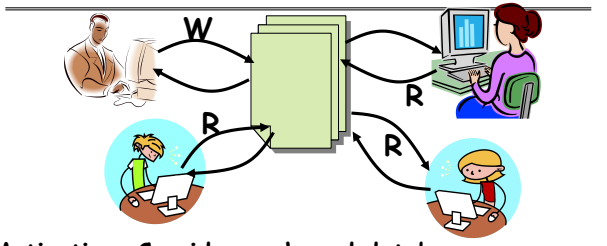
```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling
  - Hoare-style (most textbooks):
    - » Signaler gives lock, CPU to waiter; waiter runs immediately
    - » Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - Mesa-style (Nachos, most real operating systems):
    - » Signaler keeps lock and processor
    - » Waiter placed on ready queue with no special priority
    - » Practically, need to check condition again after wait

Administrivia

- First design document due *tomorrow* by 11:59pm
  - Good luck!
  - Use the newsgroup for questions (search Google groups)
- Design reviews:
  - Everyone must attend! (no exceptions)
  - 2 points off for one missing person
  - 1 additional point off for each additional missing person
  - Penalty for arriving late (plan on arriving 5—10 mins early)
  - Sign up link will be posted on announcements and projects pages
- What we expect in document/review:
  - Architecture, correctness constraints, algorithms, pseudocode, NO CODE!
  - Important: testing strategy, and test case types

Readers/Writers Problem



- Motivation: Consider a shared database
  - Two classes of users:
    - » Readers – never modify database
    - » Writers – read and modify database
  - Is using a single lock on the whole database sufficient?
    - » Like to have many readers at the same time
    - » Only one writer at a time

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()
    ```
    Wait until no writers
    Access data base
    Check out - wake up a waiting writer
    ```
  - Writer()
    ```
    Wait until no active readers or writers
    Access database
    Check out - wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {  // Is it safe to read?
    WR++;                  // No. Writers exist
    okToRead.wait(&lock);  // Sleep on cond var
    WR--;                  // No longer waiting
  }
  AR++;                    // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                    // No longer active
  if (AR == 0 && WW > 0)   // No other active readers
    okToWrite.signal();    // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) {  // Is it safe to write?
    WW++;                  // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                  // No longer waiting
  }
  AW++;                    // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                    // No longer active
  if (WW > 0){             // Give priority to writers
    okToWrite.signal();    // Wake up one writer
  } else if (WR > 0) {     // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```


Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) {  // Is it safe to read?
  WR++;                  // No. Writers exist
  okToRead.wait(&lock);  // Sleep on cond var
  WR--;                  // No longer waiting
}
AR++;                    // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- Next, W1 comes along:

```
while ((AW + AR) > 0) { // Is it safe to write?
  WW++;                 // No. Active users exist
  okToWrite.wait(&lock); // Sleep on cond var
  WW--;                 // No longer waiting
}
AW++;
```

- Can't start because of readers, so go to sleep:
  AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

```
if (AR == 0 && WW > 0)   // No other active readers
  okToWrite.signal();    // Wake up one writer
```


Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){            // Give priority to writers
  okToWrite.signal();   // Wake up one writer
} else if (WR > 0) {    // Otherwise, wake reader
  okToRead.broadcast(); // Wake all readers
}
```

  - Writer wakes up reader, so get:
    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  okToRead.wait(&lock); // Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

- What if we erase the condition check in Reader exit?

```
AR--;                   // No longer active
if (AR == 0 && WW > 0)  // No other active readers
  okToWrite.signal();   // Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                   // No longer active
okToWrite.broadcast();  // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```

Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() {
   if semaphore queue is not empty
      semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables Wait if necessary

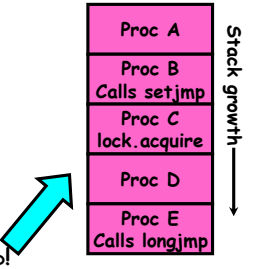
Check and/or update state variables

BREAK

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
   lock.acquire();
   …
   if (exception) {
      lock.release();
      return errReturnCode;
   }
   …
   lock.release();
   return OK;
}
```



  - Watch out for `setjmp/longjmp`!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
void Rtn() {
  lock.acquire();
  …
  DoFoo();
  …
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Notice that an exception in DoFoo() will exit without releasing the lock

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {      // catch exception
    lock.release(); // release lock
    throw;          // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Even Better: auto_ptr<T> facility. See C++ Spec.
    - » Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
  …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method or block:
    - » `void wait(long timeout); // Wait for timeout`
    - » `void wait(long timeout, int nanoseconds); //variant`
    - » `void wait();`
  - How to signal in a synchronized method or block:
    - » `void notify(); // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
  wait (CHECKPERIOD);
  t2 = time.new();
  if (t2 - t1 > LONG_TIME) checkMachine();
}
```

  - Not all Java VMs equivalent!
    - » Different scheduling policies, not necessarily preemptive!

ACID

- How does a database handle concurrency?
  - It provides A.C.I.D. properties – Atomicity, Consistency, Isolation, Durability
- Key concept: Transaction
  - An atomic sequence of database actions (reads/writes)
    - » Actions all happen or none at all
  - Takes DB from one consistent state to another

DBMS Consistency Example

- Here, *consistency* is based on our knowledge of banking "semantics"
- In general, up to writer of transaction to ensure transaction preserves consistency
- DBMS provides (limited) automatic enforcement, via integrity constraints
  - e.g., balances must be >= 0

Challenge: Concurrent transactions

- Goal: execute xacts {T1, T2, … Tn}, and ensure a consistent outcome
  - Isolate xact's intermediate state from other xacts
- *One option:* "serial" schedule (one after another)
- *Better:* allow interleaving of xact actions, as long as outcome is equivalent to some serial schedule
- Two possible enforcement methods
  - Optimistic: permit arbitrary interleaving, then check equivalence to serial schedule
  - Pessimistic: xacts set *locks* on data objects, such that illegal interleaving is impossible
    - » More on locking in another lecture…

Ensuring Transaction Properties

- DBMS ensures:
  - *Atomicity* even if xact aborted (due to deadlock, system crash, …)
  - *Durability* (persistence) of committed xacts, even if system crashes
- Idea: Keep a *log* of all actions carried out by the DBMS:
  - Record all DB modifications in log, *before* they are executed
  - To abort a xact, undo logged actions in reverse order
  - If system crashes, must:
    1) undo partially executed xacts (ensures atomicity)
    2) redo committed xacts (ensures durability)
  - *Much trickier than it sounds!*

ACID Summary

- Atomicity: guarantee that either all of the tasks of a transaction are performed, or none of them are
- Consistency: database is in a legal state when the transaction begins and when it ends – a transaction cannot break the rules, or *integrity constraints*
- *Isolation:* operations inside a transaction appear isolated from all other operations – no operation outside transaction can see data in an intermediate state
- Durability: guarantee that once the user has been notified of success, the transaction will persist (survive system failure)

Summary

- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Language support for synchronization:
  - Java provides synchronized keyword and one condition-variable per object (with `wait()` and `notify()`)

CS162
Operating Systems and Systems Programming
Lecture 8

Tips for Working in a Project Team/ Cooperating Processes and Deadlock

February 20, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Tips for Programming in a Project Team
- Discussion of Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Tips for Programming in a Project Team

"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    - » Empire state building finished in one year: staging iron production thousands of miles away
    - » Or the Hoover dam: built towns to hold workers
  - Is it OK to miss deadlines?
    - » We make it free (slip days)
    - » Reality: they're very expensive as time-to-market is one of the most important things!

Big Projects

- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)!
    - » This is why we bug you about starting the project early
    - » Had a grad student who used to say he just needed "10 minutes" to fix something. Two hours later…
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    - » Time reaches a minimum bound
    - » With complex interactions, time increases!
  - Mythical person-month problem:
    - » You estimate how long a project will take
    - » Starts to fall behind, so you add more people
    - » Project takes even more time!

Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
  - Problem: Lots of communication across APIs
    - » If B changes the API, A may need to make changes
    - » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most CS162 project teams are functional, but people have had success with task-based divisions

Defensive Programming

- Like defensive driving, but for code:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Communication

- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - » You would never be able to schedule meetings otherwise
  - Why do we require 4 people minimum?
    - » You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: phase 4, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document

  - Why Document?
    - » Expose decisions and communicate to others
    - » Easier to spot mistakes early
    - » Easier to estimate progress
  - What to document?
    - » Everything (but don't overwhelm people or no one will read)
  - Standardize!
    - » One programming format: variable naming conventions, tab indents,etc.
    - » Comments (Requires, effects, modifies)—javadoc?

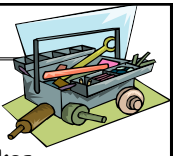
Suggested Documents for You to Maintain

- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Taming Complexity with Abstractions

- Break large, complex system into *independent* components
  - Goal: independently design, implement, and test each component
  - Added benefit: better security through isolation
  - But, components must work together in the final system
- We need interfaces (specs) between the components
  - The boundaries between components (and people)
  - To isolate them from one another
  - To ensure the final system actually works
- The interfaces must not change (much)!
  - Otherwise, development is not parallel

Use Software Tools



- Source revision control software
  - (CVS, Subversion, others…)
  - Easy to go back and see history/undo mistakes
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use CVS's features)
- Use automated testing tools
  - Write scripts for non-interactive software
  - Use "expect" for interactive software
  - JUnit: automate unit testing
  - Microsoft rebuilds the Vista kernel every night with the day's changes. Everyone is running/testing the latest software
- Use E-mail and instant messaging consistently to leave a history trail

Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    - » Let's people test continuously, but more work
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
- Testing types:
  - Unit tests: check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?

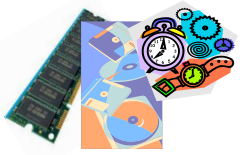
Administrivia

- Midterm I next week:
  - Wednesday, 2/27, 10 Evans 6-7:30pm
  - Closed book, no notes, no calculators/PDAs
  - Topics: Everything including today (lectures, book, readings, projects)
  - Email cs162 with conflicts (academic only)
- *No class on day of Midterm*
- I will post extra office hours for people who have questions about the material (or life, whatever)
- Midterm I review session Monday 2/25 after class
  - 120 Latimer, 6-7:30pm

Resource Contention and Deadlock

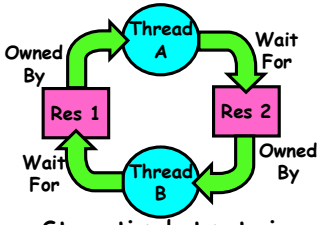
Resources

- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - » CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    - » Disk space, plotter, chunk of virtual address space
    - » Mutual exclusion – the right to enter a critical section
- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources



Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - » Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock ⇒ Starvation but not vice versa
    - » Starvation can end (but doesn't have to)
    - » Deadlock can't end without external intervention

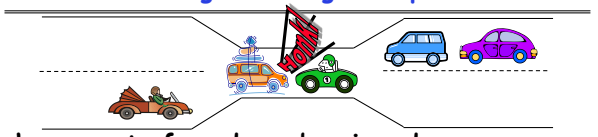
Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

Thread A	Thread B
x.P();	y.P();
y.P();	x.P();
y.V();	x.V();
x.V();	y.V();

  - Deadlock won't always happen with this code
    - » Have to have exactly the right timing ("wrong" timing?)
    - » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

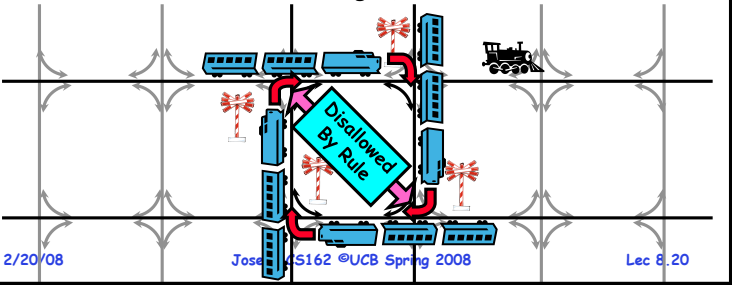
Bridge Crossing Example



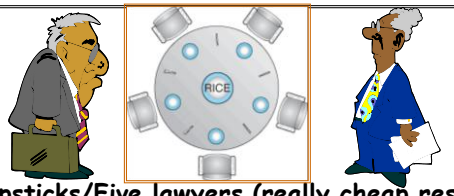
- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Dining Lawyers Problem



- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

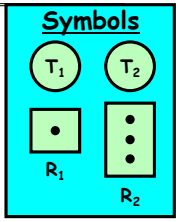
Safe and Unsafe States

- *Safe state* – system will not enter a deadlock condition
- *Unsafe state* – system *may* enter a deadlock condition
- Being in an unsafe state does not guarantee that the system will deadlock
  - Thread requests A resulting in an unsafe state
  - Then it releases B which would prevent circular wait
  - The system is in an unsafe state, but not in deadlock

BREAK

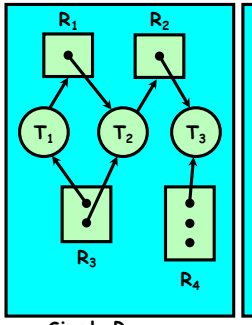
Resource-Allocation Graph

- System Model
  - A set of Threads T_1, T_2, . . . , T_n
  - Resource types R_1, R_2, . . . , R_m
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

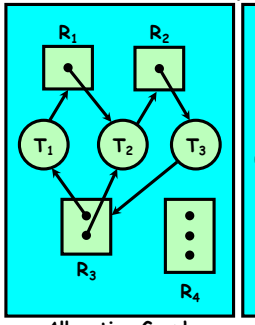


Resource Allocation Graph Examples

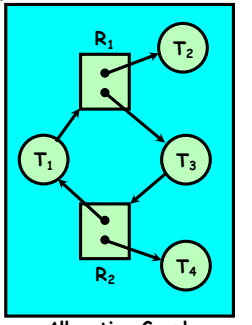
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

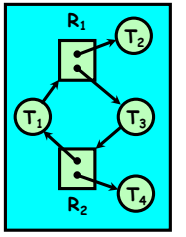
Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):
    - `[FreeResources]:` Current free resources each type
    - `[Request`$_X$`]:` Current requests from thread X
    - `[Alloc`$_X$`]:` Current resources held by thread X
  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

Techniques for Preventing Deadlock (con't)

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max remaining that might be needed by any thread

- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \leq [Avail]$) for ($[Request_{node}] \leq [Avail]$) Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker's Algorithm Example

- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Summary

- Suggestions for dealing with Project Partners
  - Start Early, Meet Often
  - Develop Good Organizational Plan, Document Everything, Use the right tools, Develop Comprehensive Testing Plan
  - (Oh, and add 2 years to every deadline!)
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress
- Next Time: Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this

CS162
Operating Systems and Systems Programming
Lecture 9

History of the World Parts 1—5
Operating Systems Structures

February 25, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- History of Operating Systems
  - Really a history of resource-driven choices
- Operating Systems Structures
- Operating Systems Organizations
- Abstractions and layering

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Moore's Law Change Drives OS Change

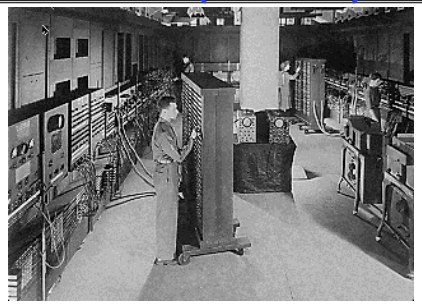
	1981	2006	Factor
CPU MHz, Cycles/inst	10 3—10	3200x4 0.25—0.5	1,280 6—40
DRAM capacity	128KB	4GB	32,768
Disk capacity	10MB	1TB	100,000
Net bandwidth	9600 b/s	1 Gb/s	110,000
# addr bits	16	32	2
#users/machine	10s	≤ 1	≤ 0.1
Price	$25,000	$4,000	0.2

Typical academic computer 1981 vs 2006

Moore's law effects

- Nothing like this in any other area of business
- Transportation in over 200 years:
  - 2 orders of magnitude from horseback @10mph to Concorde @1000mph
  - Computers do this every decade (at least until 2002)!
- What does this mean for us?
  - Techniques have to vary over time to adapt to changing tradeoffs
- I place a lot more emphasis on principles
  - The key concepts underlying computer systems
  - Less emphasis on facts that are likely to change over the next few years…
- Let's examine the way changes in $/MIP has radically changed how OS's work

Dawn of time
ENIAC: (1945—1955)

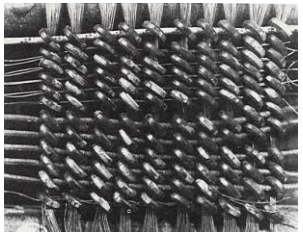


- "The machine designed by Drs. Eckert and Mauchly was a monstrosity. When it was finished, the ENIAC filled an entire room, weighed thirty tons, and consumed two hundred kilowatts of power."
- http://ei.cs.vt.edu/~history/ENIAC.Richey.HTML

History Phase 1 (1948—1970)
Hardware Expensive, Humans Cheap

- When computers cost millions of $'s, optimize for more efficient use of the hardware!
  - Lack of interaction between user and computer
- User at console: one user at a time
- Batch monitor: load program, run, print
- Optimize to better use hardware
  - When user thinking at console, computer idle⇒BAD!
  - Feed computer batches and make users wait
  - Autograder for this course is similar
- *No protection:* what if batch program has bug?

Core Memories (1950s & 60s)



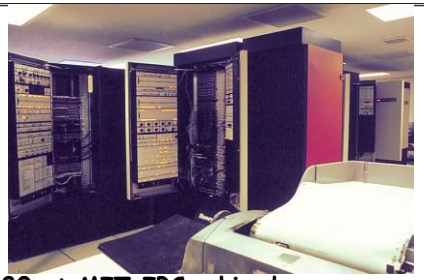
The first magnetic core memory, from the IBM 405 Alphabetical Accounting Machine.

- Core Memory stored data as magnetization in iron rings
  - Iron "cores" woven into a 2-dimensional mesh of wires
  - Origin of the term "Dump Core"
  - Rumor that IBM consulted Life Saver company
- See: http://www.columbia.edu/acis/history/core.html

History Phase 1½ (late 60s/early 70s)

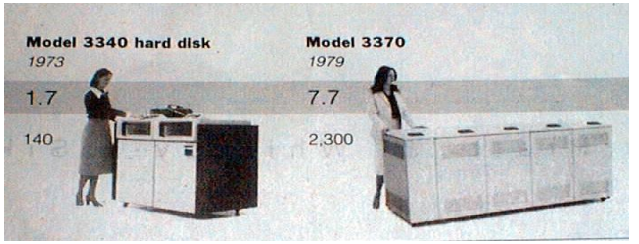
- Data channels, Interrupts: overlap I/O and compute
  - DMA – Direct Memory Access for I/O devices
  - I/O can be completed asynchronously
- Multiprogramming: several programs run simultaneously
  - Small jobs not delayed by large jobs
  - More overlap between I/O and CPU
  - Need memory protection between programs and/or OS
- Complexity gets out of hand:
  - Multics: announced in 1963, ran in 1969
    - » 1777 people "contributed to Multics" (30-40 core dev)
    - » Turing award lecture from Fernando Corbató (key researcher): "On building systems that will fail"
  - OS 360: released with 1000 known bugs (APARs)
    - » "Anomalous Program Activity Report"
- OS finally becomes an important science:
  - How to deal with complexity???
  - UNIX based on Multics, but vastly simplified

A Multics System (Circa 1976)



- The 6180 at MIT IPC, skin doors open, circa 1976:
  - "We usually ran the machine with doors open so the operators could see the AQ register display, which gave you an idea of the machine load, and for convenient access to the EXECUTE button, which the operator would push to enter BOS if the machine crashed."
- http://www.multicians.org/multics-stories.html

Early Disk History



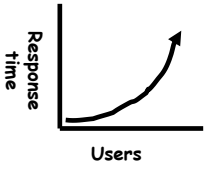
1973:
1. 7 Mbit/sq. in
140 MBytes

1979:
7. 7 Mbit/sq. in
2,300 MBytes

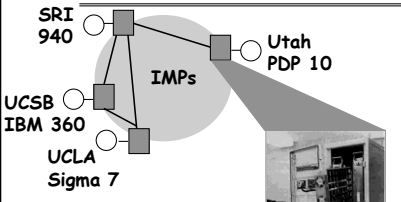
Contrast: Seagate 1TB, 164 GB/SQ in, 3½ in disk, 4 platters

History Phase 2 (1970 – 1985)
Hardware Cheaper, Humans Expensive

- Computers available for tens of thousands of dollars instead of millions
- OS Technology maturing/stabilizing
- *Interactive* timesharing:
  - Use cheap terminals (~$1000) to let multiple users interact with the system at the same time
  - Sacrifice CPU time to get better response time
  - Users do debugging, editing, and email online
- Problem: Thrashing
  - Performance very non-linear response with load
  - Thrashing caused by many factors including
    - » Swapping, queueing



The ARPANet (1968-1970's)

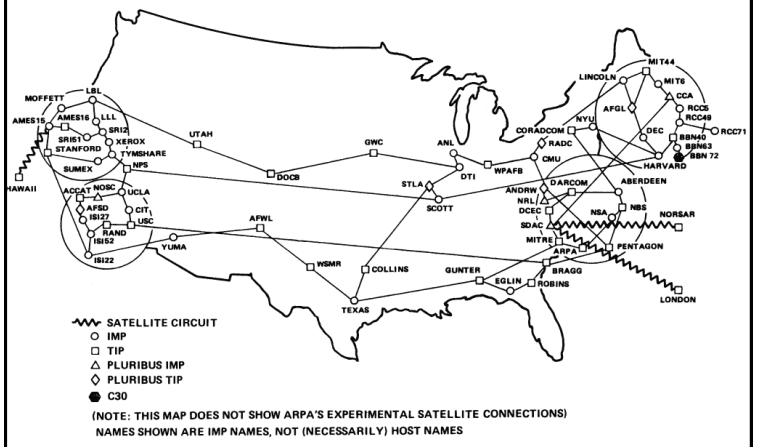


BBN team that implemented the interface message processor

- Paul Baran
  - RAND Corp, early 1960s
  - Communications networks that would survive a major enemy attack
- ARPANet: Research vehicle for "Resource Sharing Computer Networks"
  - 2 September 1969: UCLA first node on the ARPANet
  - December 1969: 4 nodes connected by 56 kbps phone lines
  - 1970's: <100 computers

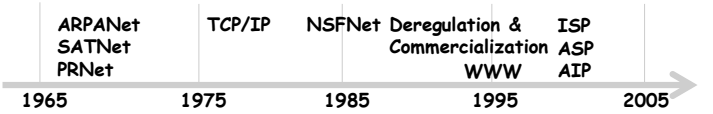
http://www.cnn.com/2004/TECH/internet/08/29/internet.birthday.ap/index.html

ARPANET GEOGRAPHIC MAP, OCTOBER 1980



ARPANet Evolves into Internet

- First E-mail SPAM message: 1 May 1978 12:33 EDT
- 80-83: TCP/IP, DNS; ARPANET and MILNET split
- 85-86: NSF builds NSFNET as backbone, links 6 Supercomputer centers, 1.5 Mbps, 10,000 computers
- 87-90: link regional networks, NSI (NASA), ESNet (DOE), DARTnet, TWBNet (DARPA), 100,000 computers



SATNet: Satelite network
PRNet: Radio Network

Administrivia

- Midterm I: Wednesday, 2/27, 10 Evans 6-7:30pm
  - Closed book, no notes, no calculators/PDAs
  - Topics: Everything up to 2/20 (lectures, book, readings, projects)
  - Email cs162 with conflicts (academic only)
- *No class on day of Midterm*
- I will hold extra office hours for people who have questions about the material (or life, whatever)
  - Monday 2-3:30, Tuesday 12:30-2
- Midterm I review session today after class
  - 120 Latimer, 6-7:30pm

What is a Communication Network? (End-system Centric View)

- Network offers one basic service: move information
  - Bird, fire, messenger, truck, telegraph, telephone, Internet …
  - Another example, transportation service: move objects
    - » Horse, train, truck, airplane ...
- What distinguish different types of networks?
  - The services they provide
- What distinguish the services?
  - Latency
  - Bandwidth
  - Loss rate
  - Number of end systems
  - Service interface (how to invoke the service?)
  - Others
    - » Reliability, unicast vs. multicast, real-time...

What is a Communication Network? (Infrastructure Centric View)

- Communication medium: electron, photon
- Network components:
  - Links – carry bits from one place to another (or maybe multiple places): fiber, copper, satellite, …
  - Interfaces – attach devices to links
  - Switches/routers – interconnect links: electronic/optic, crossbar/Banyan
  - Hosts – communication endpoints: workstations, PDAs, cell phones, toasters
- Protocols – rules governing communication between nodes
  - TCP/IP, ATM, MPLS, SONET, Ethernet, X.25
- Applications: Web browser, X Windows, FTP, ...

Network Components (Examples)

Links

Fibers

Coaxial Cable

Interfaces

Ethernet card

Wireless card

Switches/routers

Large router

Telephone switch

Types of Networks

- Geographical distance
  - Local Area Networks (LAN): Ethernet, Token ring, FDDI
  - Metropolitan Area Networks (MAN): DQDB, SMDS
  - Wide Area Networks (WAN): X.25, ATM, frame relay
  - Caveat: LAN, MAN, WAN may mean different things
    - » Service, network technology, networks
- Information type
  - Data networks vs. telecommunication networks
- Application type
  - Special purpose networks: airline reservation network, banking network, credit card network, telephony
  - General purpose network: Internet

Types of Networks

- Right to use
  - Private: enterprise networks
  - Public: telephony network, Internet
- Ownership of protocols
  - Proprietary: IBM System Network Architecture (SNA)
  - Open: Internet Protocol (IP)
- Technologies
  - Terrestrial vs. satellite
  - Wired vs. wireless
- Protocols
  - IP, AppleTalk, SNA

History Phase 3 (1981—)
Hardware Very Cheap, Humans Very Expensive

- Computer costs $1K, Programmer costs $100K/year
  - If you can make someone 1% more efficient by giving them a computer, it's worth it!
  - Use computers to make people more efficient
- Personal computing:
  - Computers cheap, so give everyone a PC
- Limited Hardware Resources Initially:
  - OS becomes a subroutine library
  - One application at a time (MSDOS, CP/M, …)
- Eventually PCs become powerful:
  - OS regains all the complexity of a "big" OS
  - multiprogramming, memory protection, etc (NT,OS/2)
- Question: As hardware gets cheaper does need for OS go away?

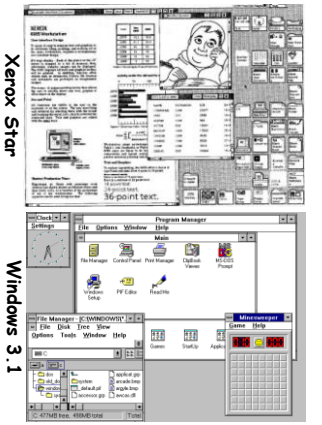
History Phase 3 (con't)
Graphical User Interfaces

- CS160 ⇒ All about GUIs
- Xerox Star: 1981
  - Originally a research project (Alto)
  - First "mice", "windows"
- Apple Lisa/Machintosh: 1984
  - "Look and Feel" suit 1988
- Microsoft Windows:
  - Win 1.0 (1985)
  - Win 3.1 (1990)
  - Win 95 (1995)
  - Win NT (1993)
  - Win 2000 (2000)
  - Win XP (2001)
  - Win Vista (2007)

Single Level

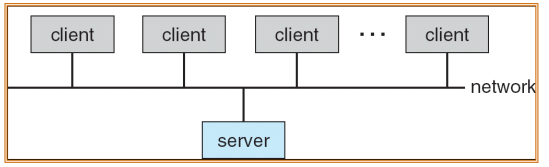
HAL/Protection

No HAL/ Full Prot



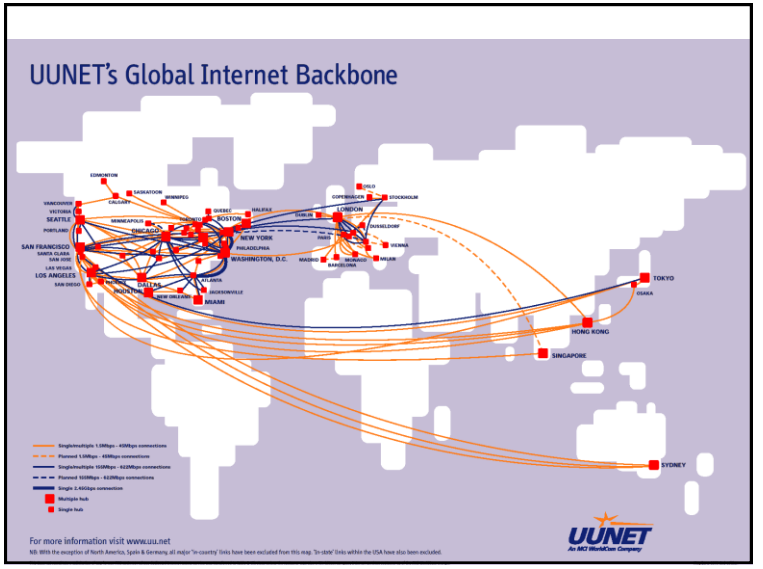
History Phase 4 (1989—): Distributed Systems

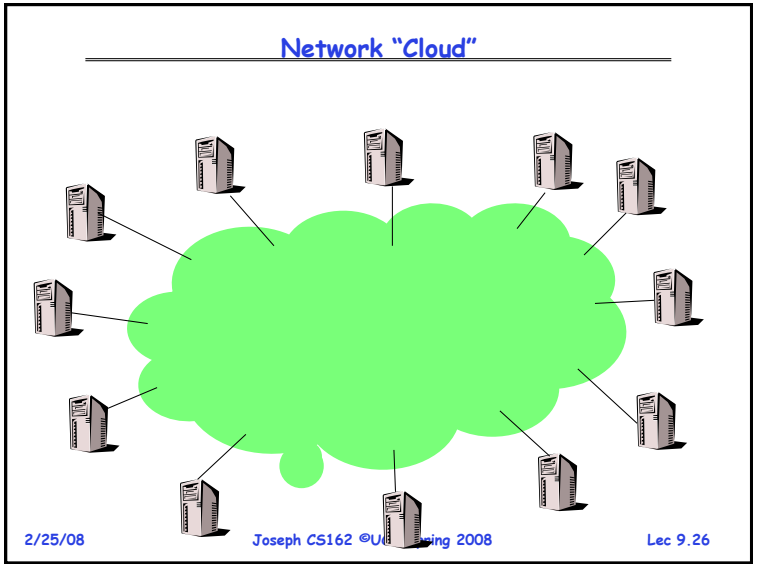
- Networking (Local Area Networking)
  - Different machines share resources
  - Printers, File Servers, Web Servers
  - Client – Server Model
- Services
  - Computing
  - File Storage

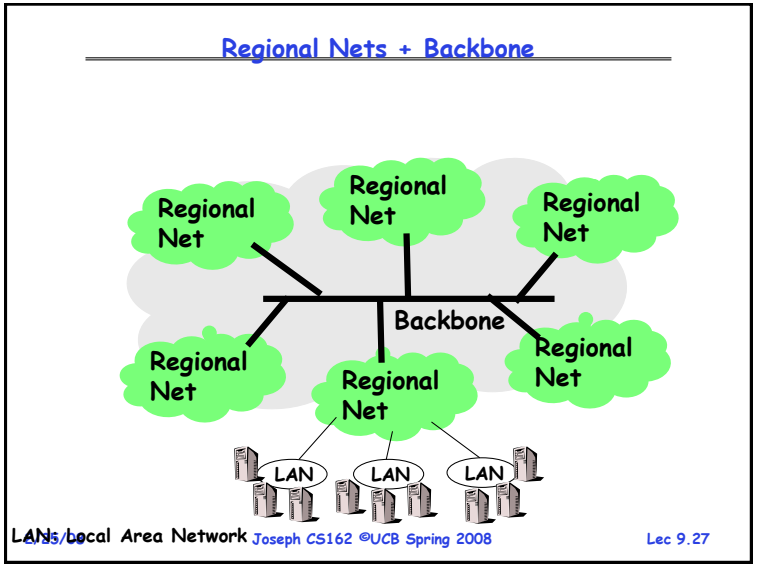


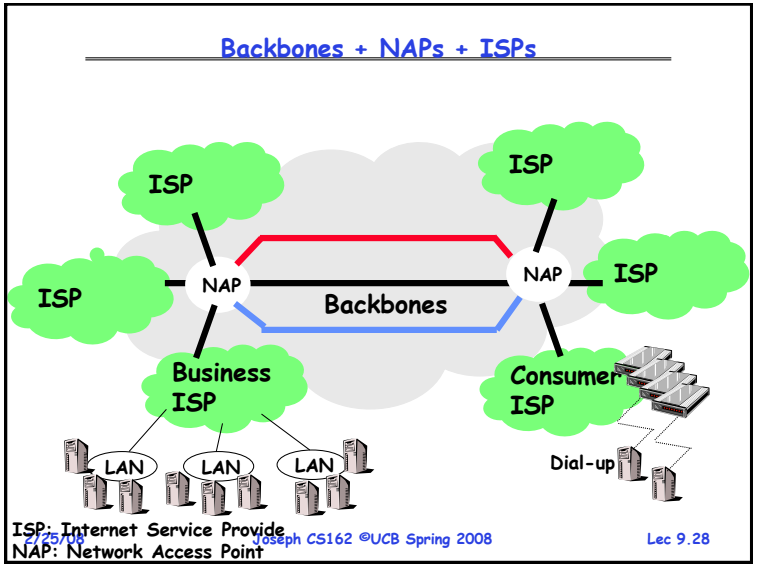
History Phase 4 (1989—): Internet

- Developed by the research community
  - Based on open standard: Internet Protocol
  - Internet Engineering Task Force (IETF)
- Technical basis for many other types of networks
  - Intranet: enterprise IP network
- Services Provided by the Internet
  - Shared access to computing resources: telnet (1970's)
  - Shared access to data/files: FTP, NFS, AFS (1980's)
  - Communication medium over which people interact
    - » email (1980's), on-line chat rooms, instant messaging (1990's)
    - » audio, video (1990's, early 00's)
  - Medium for information dissemination
    - » USENET (1980's)
    - » WWW (1990's)
    - » Audio, video (late 90's, early 00's) – replacing radio, TV?
    - » File sharing (late 90's, early 00's)

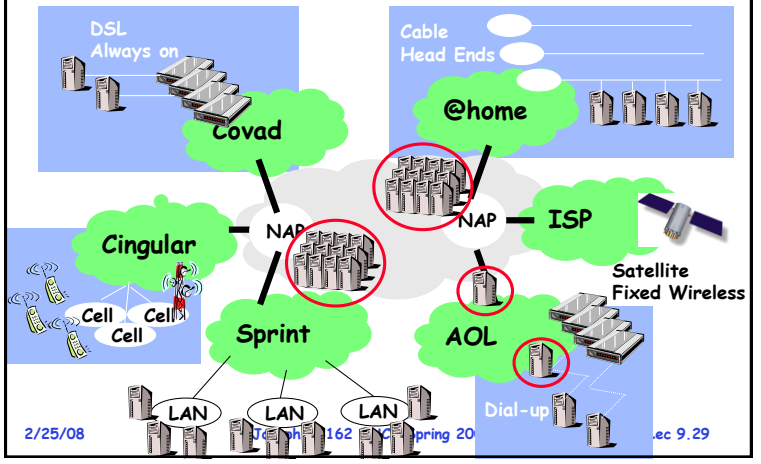

UUNET's Global Internet Backbone
SEATTLE
SAN FRANCISCO
LOS ANGELES
CHICAGO
NEW YORK
WASHINGTON, D.C.
BOSTON
LONDON
TOKYO
HONG KONG
SINGAPORE
SYDNEY
MIAMI
For more information visit www.uu.net
UUNET


Network "Cloud"
2/25/08
Joseph CS162 ©UCB Spring 2008
Lec 9.26


Regional Nets + Backbone
Regional Net
Regional Net
Regional Net
Backbone
Regional Net
Regional Net
Regional Net
LAN
LAN
LAN
LAN: Local Area Network
2/25/08
Joseph CS162 ©UCB Spring 2008
Lec 9.27


Backbones + NAPs + ISPs
ISP
ISP
ISP
ISP
NAP
NAP
Backbones
Business ISP
Consumer ISP
LAN
LAN
LAN
Dial-up
ISP: Internet Service Provide
NAP: Network Access Point
2/25/08
Joseph CS162 ©UCB Spring 2008
Lec 9.28

Computers Inside the Core



History Phase 5 (1995—): Mobile Systems

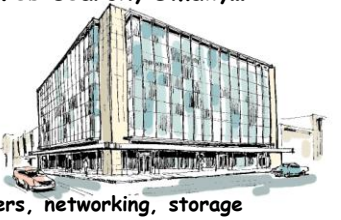
- Ubiquitous Mobile Devices
  - Laptops, PDAs, phones
  - Small, portable, and inexpensive
    - » Recently twice as many smart phones as PDAs
    - » Many computers/person!
  - Limited capabilities (memory, CPU, power, etc…)
- Wireless/Wide Area Networking
  - Leveraging the infrastructure
  - Huge distributed pool of resources extend devices
  - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Peer-to-peer systems
  - Many devices with equal responsibilities work together
  - Components of "Operating System" spread across globe

CITRIS's Model: A Societal Scale Information System

- Center for Information Technology Research in the Interest of Society
- The Network is the OS
  - Functionality spread throughout network

Datacenter is the Computer

- *(From Luiz Barroso's talk at RAD Lab 12/11)*
- Google *program* == Web search, Gmail,…
- Google *computer* ==

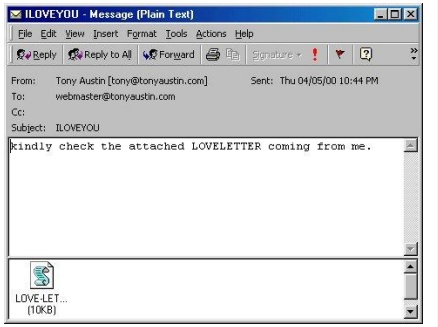


  - Thousands of computers, networking, storage
- Warehouse-sized facilities and workloads may be unusual today but are likely to be more common in the next few years

BREAK

LoveLetter Virus (May 2000)

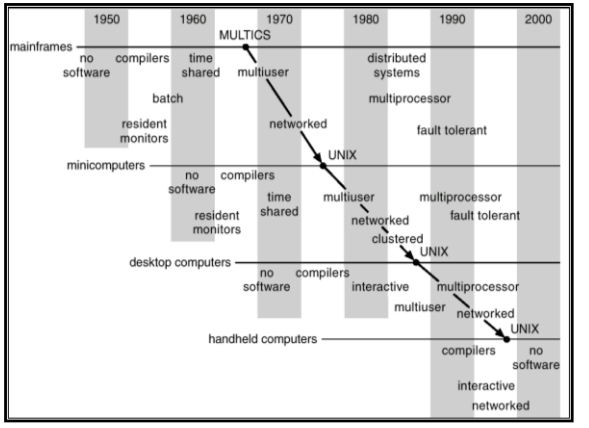
- E-mail message with VBScript (simplified Visual Basic)
- Relies on Windows Scripting Host
  - Enabled by default in Win98/2000
- User clicks on attachment➔ infected!
  - E-mails itself to everyone in Outlook address book
  - Replaces some files with a copy of itself
  - Searches all drives
  - Downloads password cracking program
- 60-80% of US companies infected and 100K European servers



Moore's Law Reprise: Modern Laptop

	1981	2005	2008 Ultralight Tablet Laptop
CPU MHz, Cycles/inst	10 3—10	3200x4 0.25—0.5	1600 (Core 2 Duo) 0.25—0.5
DRAM capacity	128KB	4GB	4GB
Disk capacity	10MB	1TB	200GB
Net bandwidth	9600 b/s	1 Gb/s	1 Gb/s (wired) 248 Mb/s (wireless) 2 Mb/s (wide-area)
# addr bits	16	32	64
#users/machine	10s	≤ 1	≤ ¼
Price	$25,000	$4,000	$2,000

Migration of Operating-System Concepts and Features



History of OS: Summary

- Change is continuous and OSs should adapt
  - Not: look how stupid batch processing was
  - But: Made sense at the time
- Situation today is much like the late 60s [poll]
  - Small OS: 100K lines
  - Large OS: 10M lines (5M for the browser!)
    - » 100-1000 people-years
- Complexity still reigns
  - NT developed (early to late 90's): Never worked well
  - Windows 2000/XP: Very successful
  - Windows Vista (aka "Longhorn") delayed many times
    - » Finally released in January 2007
    - » Promised by removing some of the intended technology
    - » Slow adoption rate, even in 2008
- CS162: understand OSs to simplify them

Now for a quick tour of OS Structures

Operating Systems Components (What are the pieces of the OS)

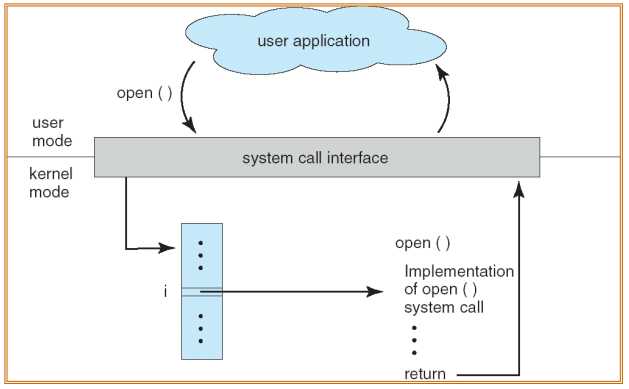
- Process Management
- Main-Memory Management
- I/O System management
- File Management
- Networking
- User Interfaces

Operating System Services (What things does the OS do?)

- Services that (more-or-less) map onto components
  - Program execution
    - » How do you execute concurrent sequences of instructions?
  - I/O operations
    - » Standardized interfaces to extremely diverse devices
  - File system manipulation
    - » How do you read/write/preserve files?
    - » Looming concern: How do you even find files???
  - Communications
    - » Networking protocols/Interface with CyberSpace?
- Cross-cutting capabilities
  - Error detection & recovery
  - Resource allocation
  - Accounting
  - Protection

System Calls (What is the API)

- See Chapter 2 of 7th edition or Chapter 3 of 6th

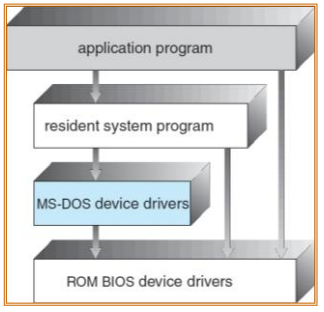


Operating Systems Structure (What is the organizational Principle?)

- Simple
  - Only one or two levels of code
- Layered
  - Lower levels independent of upper levels
- Microkernel
  - OS built from many user-level processes
- Modular
  - Core kernel with Dynamically loadable modules

Simple Structure

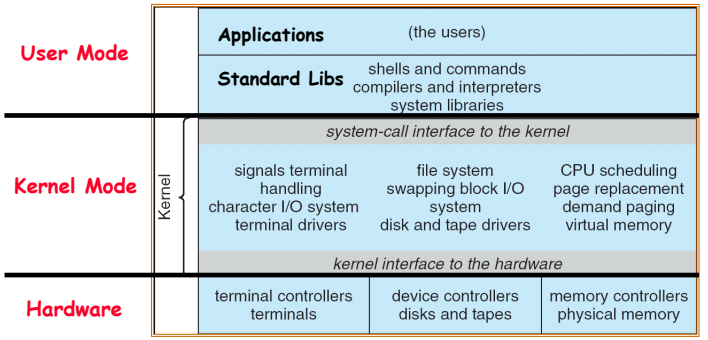
- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Interfaces and levels of functionality not well separated



UNIX: Also "Simple" Structure

- UNIX – limited by hardware functionality
- Original UNIX operating system consists of two separable parts:
  - Systems programs
  - The kernel
    - » Consists of everything below the system-call interface and above the physical hardware
    - » Provides the file system, CPU scheduling, memory management, and other operating-system functions;
    - » Many interacting functions for one level

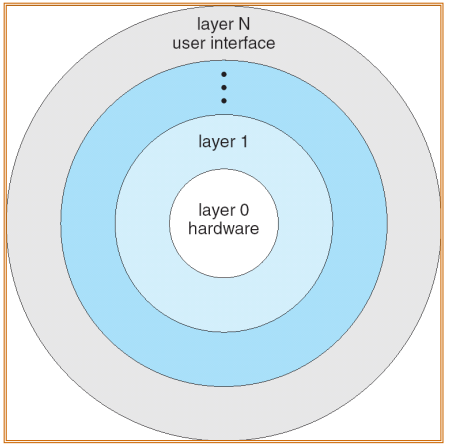
UNIX System Structure



Layered Structure

- Operating system is divided many layers (levels)
  - Each built on top of lower layers
  - Bottom layer (layer 0) is hardware
  - Highest layer (layer N) is the user interface
- Each layer uses functions (operations) and services of only lower-level layers
  - Advantage: modularity ⇒ Easier debugging/Maintenance
  - Not always possible: Does process scheduler lie above or below virtual memory layer?
    - » Need to reschedule processor while waiting for paging
    - » May need to page in information about tasks
- Important: Machine-dependent vs independent layers
  - Easier migration between platforms
  - Easier evolution of hardware platform
  - Good idea for you as well!

Layered Operating System

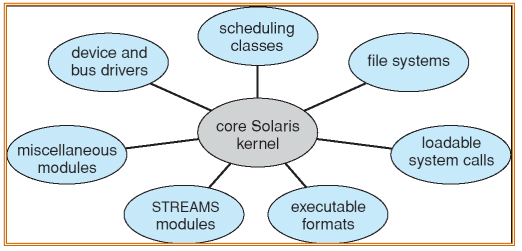


Microkernel Structure

- Moves as much from the kernel into "*user*" space
  - Small core OS running at kernel level
  - OS Services built from many independent user-level processes
- Communication between modules with message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port OS to new architectures
  - More reliable (less code is running in kernel mode)
  - Fault Isolation (parts of kernel protected from other parts)
  - More secure
- Detriments:
  - Performance overhead severe for naïve implementation

Modules-based Structure

- Most modern operating systems implement modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible



Implementation Issues (How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since both change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc…
- Event models used
  - threads vs event loops
- Backward compatability issues
  - Very important for Windows 2000/XP
- System generation/configuration
  - How to make generic OS fit on specific hardware

Conclusion

- Rapid Change in Hardware Leads to changing OS
  - Batch ⇒ Multiprogramming ⇒ Timeshare ⇒ Graphical UI ⇒ Ubiquitous Devices ⇒ Cyberspace/Metaverse/??
- OS features migrated from mainframes ⇒ PCs
- Standard Components and Services
  - Process Control
  - Main Memory
  - I/O
  - File System
  - UI
- Policy vs Mechanism
  - Crucial division: not always properly separated!
- Complexity is always out of control
  - However, "Resistance is NOT Useless!"

CS162
Operating Systems and Systems Programming
Lecture 10

Thread Scheduling

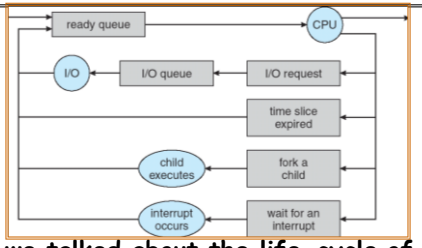
March 3, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Scheduling Policy goals
- Policy Options
- Implementation Considerations

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

CPU Scheduling

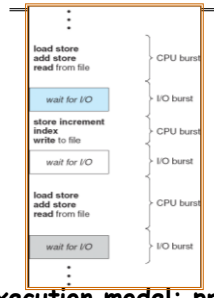


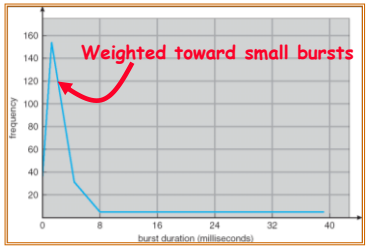
- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Assumption: CPU Bursts





- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

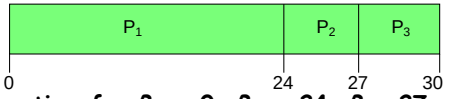
- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

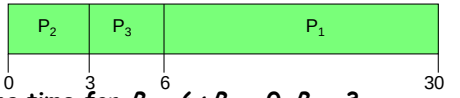
  - Suppose processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1 Now, the Gantt chart for the schedule is:



  - Waiting time for P_1 = 6; P_2 = 0, P_3 = 3
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - » Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens!

Administrivia

- Midterm #1– Mean 73.2, Std dev 12.8

•	30.0 - 35.0:	1	*
•	35.0 - 40.0:	1	*
•	40.0 - 45.0:	1	*
•	45.0 - 50.0:	0	
•	50.0 - 55.0:	5	*****
•	55.0 - 60.0:	11	**********
•	60.0 - 65.0:	9	*********
•	65.0 - 70.0:	7	*******
•	70.0 - 75.0:	13	*************
•	75.0 - 80.0:	19	*******************
•	80.0 - 85.0:	22	**********************
•	85.0 - 90.0:	11	**********
•	90.0 - 95.0:	7	*******

Administrivia

- Course resources
  - Staff office hours
  - Peer tutoring (contact hkn@eecs)
- Project 1 code due tonight
  - Conserve your slip days!!!
  - It's not worth it yet
- Group Participation: Required!
  - Group evaluations (with TA oversight) used in computing grades
  - Zero-sum game!

Round Robin (RR)

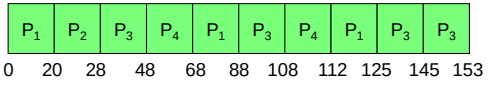
- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - *n* processes in ready queue and time quantum is *q* ⇒
    - » Each process gets 1/*n* of the CPU time
    - » In chunks of at most *q* time units
    - » No process waits more than (*n*-1)*q* time units
- Performance
  - *q* large ⇒ FCFS
  - *q* small ⇒ Interleaved (really small ⇒ hyperthreading?)
  - *q* must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

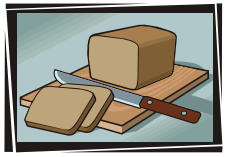
  - The Gantt chart is:



  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
  - What if too big?
    - » Response time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!



- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - » Typical time slice today is between 10ms – 100ms
    - » Typical context-switching overhead is 0.1ms – 1ms
    - » Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	31¼
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¼
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

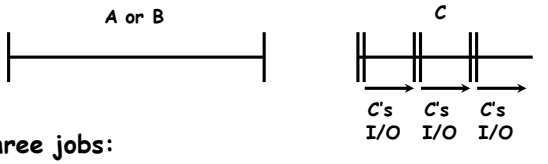
What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

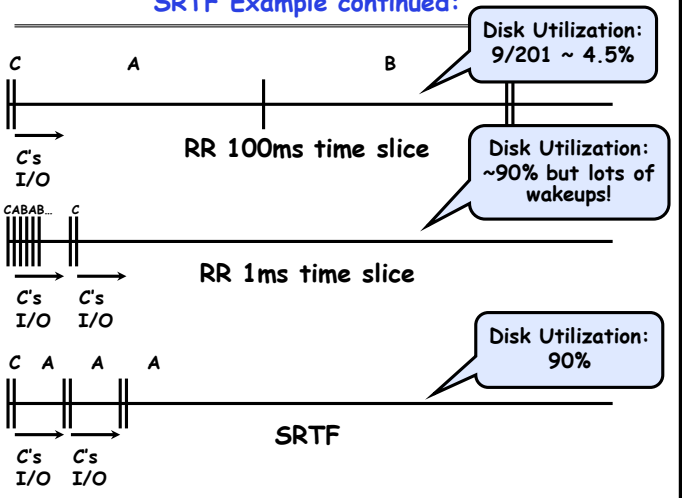
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - » SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:



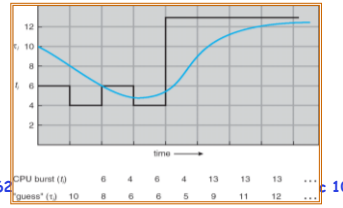
SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - » When you submit a job, have to say how long it will take
    - » To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

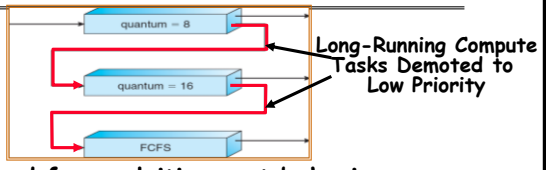
BREAK

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts: Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0 < \alpha \leq 1)$



Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - » serve all from highest priority, then next priority, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - » Put in printf's, ran much faster!

What about Fairness?

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority ⇒ Interactive jobs suffer

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

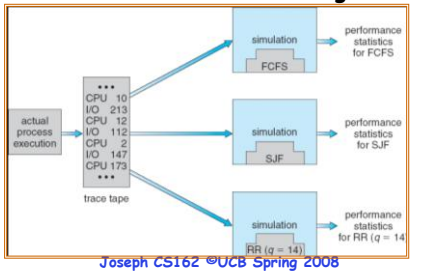
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable response time?
    - » In UNIX, if load average is 100, hard to make progress
    - » One approach: log some user out

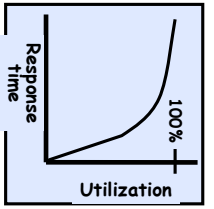
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queuing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.



A Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

Summary (Deadlock)

- Four conditions required for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress
- Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this

Summary (Scheduling)

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length

Summary (Scheduling 2)

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

CS162
Operating Systems and Systems Programming
Lecture 11

Protection: Address Spaces

March 5, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)

Review

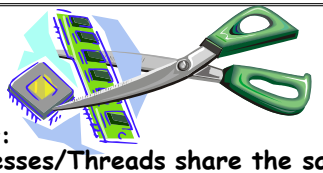
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness
- Evaluation of mechanisms:
  - Analytical, Queuing Theory, Simulation

Goals for Today

- Kernel vs User Mode
- What is an Address Space?
- How is it Implemented?

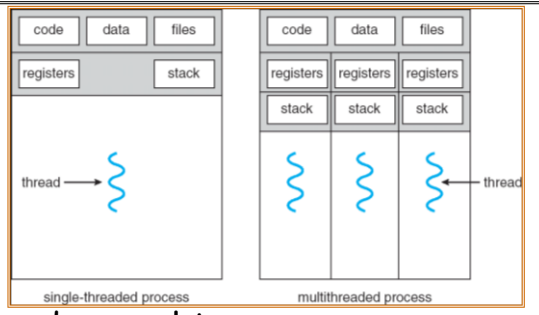
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Virtualizing Resources



- Physical Reality: Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory (protection)

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

Important Aspects of Memory Multiplexing

- Controlled overlap:
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- Protection:
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - » Kernel data protected from User programs
    - » Programs protected from themselves

Binding of Instructions and Data to Memory

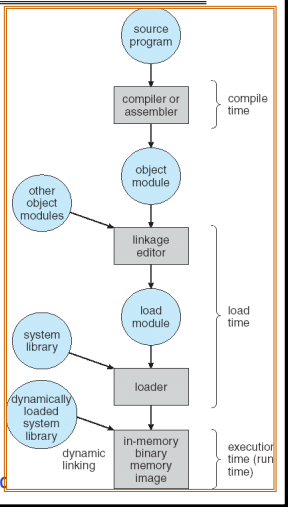
- Binding of instructions and data to addresses:
  - Choose addresses for instructions and data from the standpoint of the processor

```
data1: dw   32                    0x300  00000020
       …                             …
start: lw   r1,0(data1)           0x900  8C2000C0
       jal  checkit               0x904  0C000340
loop:  addi r1, r1, -1            0x908  2021FFFF
       bnz  r1, r0, loop          0x90C  1420FFFF
       …                             …
checkit: …                        0xD00  …
```

  - Could we place `data1`, `start`, and/or `checkit` at different addresses?
    - » Yes
    - » When? Compile time/Load time/Execution time
  - Related: which physical memory locations hold particular instructions or data?

Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
  - Compile time (i.e. "gcc")
  - Link/Load time (unix "ld" does link)
  - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine

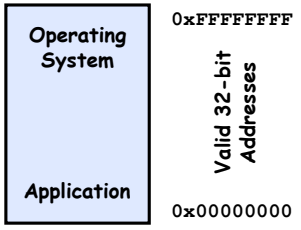


Administrivia

- Midterm #1
  - Solution with grading guidelines posted
  - Midterms will be returned Thursday and Friday
  - Regrade request deadline is next Friday (3/14)
- Project 2 started yesterday
  - Design doc due next Monday (3/10)
  - Code due Thursday 3/20

Recall: Uniprogramming

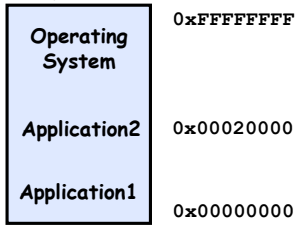
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

Multiprogramming (First Version)

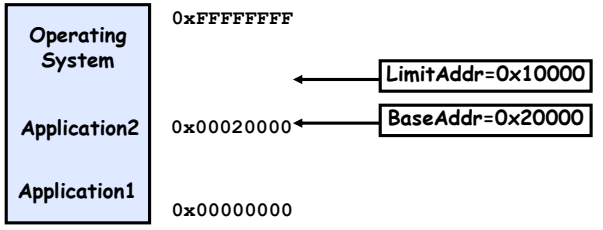
- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads



  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location of program
    - » Translation done by a linker-loader
    - » Was pretty common in early days
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

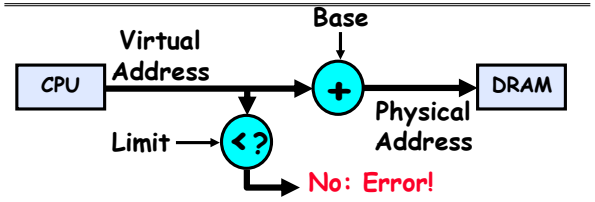
Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



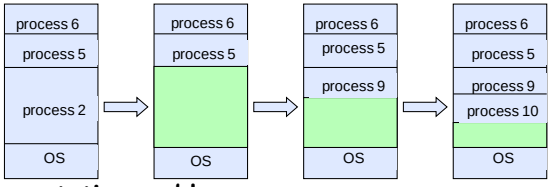
  - Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
    - » If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from TCB
    - » User not allowed to change base/limit registers

Segmentation with Base and Limit registers



- Could use base/limit for dynamic address translation (often called "segmentation"):
  - Alter address of every load/store by adding "base"
  - User allowed to read/write within segment
    - » Accesses are relative to segment so don't have to be relocated when program moved to different segment
  - User may have multiple segments available (e.g x86)
    - » Loads and stores include segment ID in opcode:
      x86 Example: `mov [es:bx],ax.`
    - » Operating system moves around segment base pointers as necessary

Issues with simple segmentation method

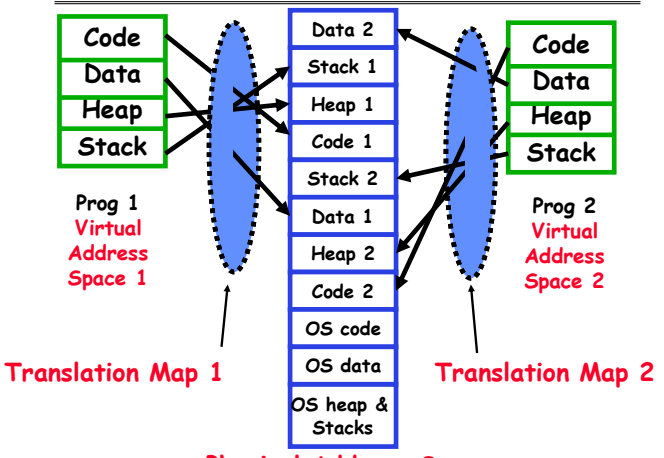


- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by by providing multiple segments per process
- Need enough physical memory for every process

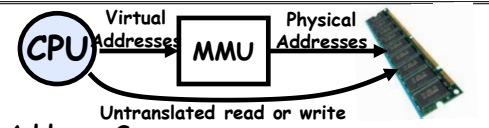
Multiprogramming (Translation and Protection version 2)

- Problem: Run multiple applications in such a way that they are protected from one another
- Goals:
  - Isolate processes and kernel from one another
  - Allow flexible translation that:
    - » Doesn't lead to fragmentation
    - » Allows easy sharing between processes
    - » Allows only part of process to be resident in physical memory
- (Some of the required) Hardware Mechanisms:
  - General Address Translation
    - » Flexible: Can fit physical chunks of memory into arbitrary places in users' address spaces
    - » Not limited to small number of segments
    - » Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
  - Dual Mode Operation
    - » Protection base involving kernel/user distinction

Example of General Address Translation



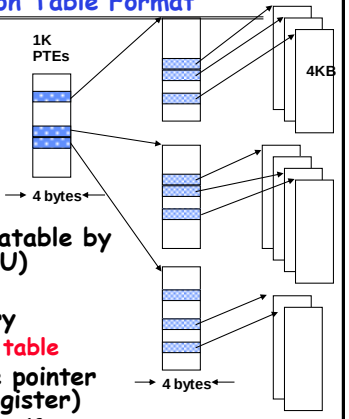
Two Views of Memory



- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently: two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View fom memory (physical memory)
  - Translation box converts between the two views
- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space
  - Overlap avoided through translation, not relocation

Example of Translation Table Format

Two-level Page Tables
32-bit address:



- Page: a unit of memory translatable by memory management unit (MMU)
  - Typically 1K – 8K
- Page table structure in memory
  - Each user has different page table
- Address Space switch: change pointer to base of table (hardware register)
  - Hardware traverses page table (for many architectures)
  - MIPS uses software to traverse table

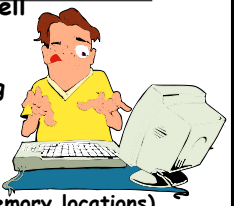
BREAK

Dual-Mode Operation

- Can Application Modify its own translation tables?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Priviledge Level) from 0 – 3
    - » PL0 has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PL0 ("kernel") and PL3 ("user")

For Protection, Lock User-Programs in Asylum

- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - User cannot modify page table mapping
  - Limited access to memory: cannot adversely effect other processes
    - » Side-effect: Limited access to memory-mapped I/O operations (I/O that occurs by reading/writing memory locations)
  - Limited access to interrupt controller
  - What else needs to be protected?



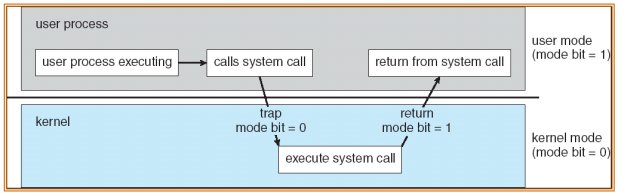
- A couple of issues
  - How to share CPU between kernel and user programs?
    - » Kinda like both the inmates and the warden in asylum are the same person. How do you manage this???
  - How do programs interact?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): getting into cell
    - » User→ OS (user → kernel mode): getting out of cell

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

System Call Continued

- What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
- Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
  - » On entry to kernel, sets system to kernel mode
  - » Handler address fetched from table/Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    - » User addresses must be translated!w
    - » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)

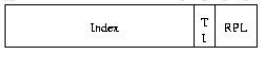
- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "Coprocessor 0"
  - Use mfc0 read contents of these registers:
    - » BadVAddr (register 8): contains memory address at which memory reference error occurred
    - » Status (register 12): interrupt mask and enable bits
    - » Cause (register 13): the cause of the exception
    - » EPC (register 14): address of the affected instruction

- Status Register fields:
  - Mask: Interrupt enable
    - » 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0⇒kernel mode
  - e = interrupt enable: 0⇒interrupts disabled
  - Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - » run in kernel mode with interrupts disabled

Intel x86 Special Registers

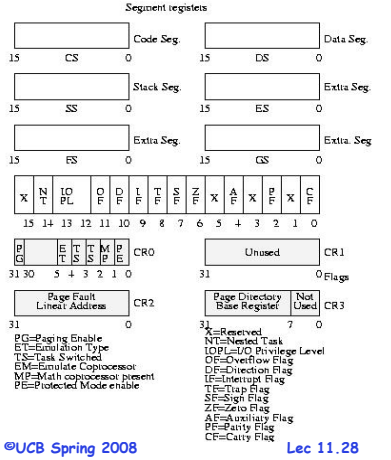


RPL = Requestor Privilege Level
TI = Table Indicator
(0 = GDT, 1 = LDT)
Index = Index into table

Protected Mode segment selector

Typical Segment Register Current Priority is RPL Of Code Segment (CS)

80386 Special Registers



Communication

- Now that we have isolated processes, how can they communicate?
  - Shared memory: common mapping to physical page
    - » As long as place objects in shared memory address range, threads from each process can communicate
    - » Note that processes A and B can talk to shared memory through different addresses
    - » In some sense, this violates the whole notion of protection that we have been developing
  - If address spaces don't share memory, all inter-address space communication must go through kernel (via system calls)
    - » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting `stdin/stdout`
    - » Message passing (send/receive): Will explain later how you can use this to build remote procedure call (RPC) abstraction so that you can have one program make procedure calls to another
    - » File System (read/write): File system is shared state!

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

Summary

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Simple Protection through Segmentation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Full translation of addresses through Memory Management Unit (MMU)
  - Every Access translated through page table
  - Changing of page tables only available to kernel
- Dual-Mode
  - Kernel/User distinction: User restricted
  - User→Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)

CS162
Operating Systems and Systems Programming
Lecture 12

Address Translation

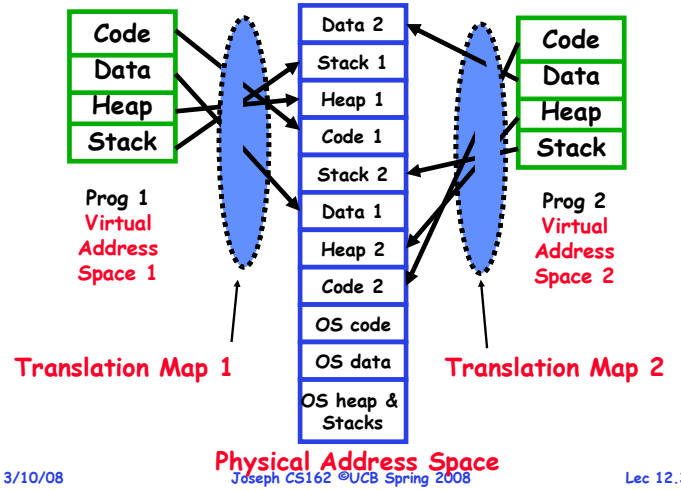
March 10, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Important Aspects of Memory Multiplexing

- **Controlled overlap:**
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- **Translation:**
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- **Protection:**
  - Prevent access to private memory of other processes
    - » Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - » Kernel data protected from User programs
    - » Programs protected from themselves

Review: General Address Translation

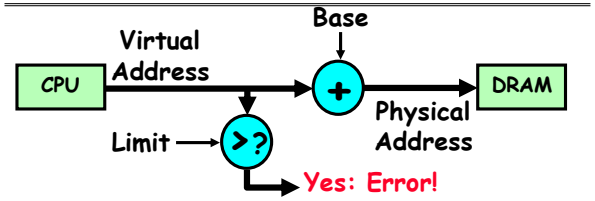


Goals for Today

- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables
- Comparison among options

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Review: Simple Segmentation: Base and Bounds (CRAY-1)



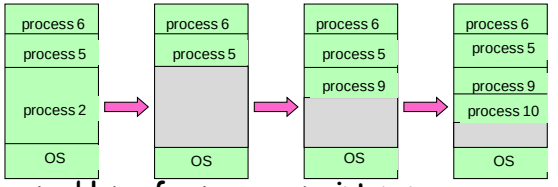
- Can use base & bounds/limit for dynamic address translation (Simple form of "segmentation"):
  - Alter every address by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Base and Limit segmentation discussion

- Provides level of indirection
  - OS can move bits around behind program's back
  - Can be used to correct if program needs to grow beyond its bounds or coalesce fragments of memory
- Only OS gets to change the base and limit!
  - Would defeat protection
- What gets saved/restored on a context switch?
  - Everything from before + base/limit values
  - Or: How about complete contents of memory (out to disk)?
    - » Called "Swapping"
- Hardware cost
  - 2 registers/Adder/Comparator
  - Slows down hardware because need to take time to do add/compare on every access
- Base and Limit Pros: Simple, relatively fast

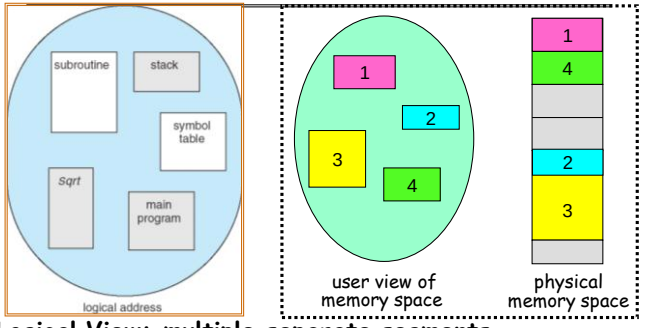
Cons for Simple Segmentation Method

- Fragmentation problem (complex memory allocation)
  - Not every process is the same size
  - Over time, memory space becomes fragmented
  - Really bad if want space to grow dynamically (e.g. heap)



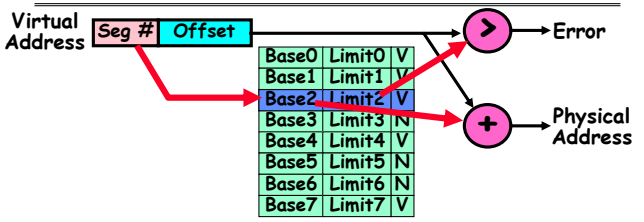
- Other problems for process maintenance
  - Doesn't allow heap and stack to grow independently
  - Want to put these as far apart in virtual memory space as possible so that they can grow as needed
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes

More Flexible Segmentation



- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

Implementation of Multi-Segment Model

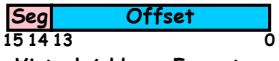


- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: `mov [es:bx],ax.`
- What is "V/N"?
  - Can mark segments as invalid; requires check as well

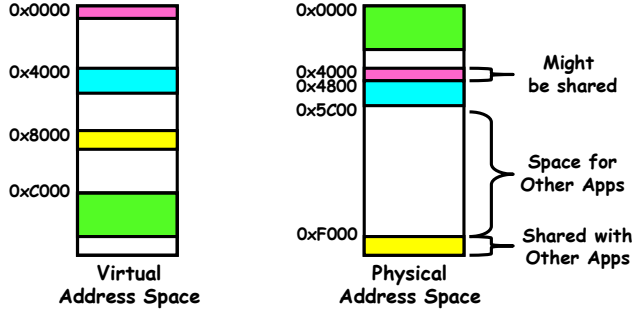
Administrivia

- Project 2
  - Initial Design Document due today (3/10) at 11:59pm
  - Look at the lecture schedule to keep up with due dates!
- Midterm #1 re-grade requests due by Friday at 5pm
  - *Entire exam will be re-graded*

Example: Four Segments (16 bit addresses)



Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example of segment translation

```
0x240   main:   la $a0, varx
0x244           jal strlen
  …               …
0x360   strlen: li   $v0, 0  ;count
0x364   loop:   lb   $t0, ($a0)
0x368           beq  $r0,$t1, done
  …               …
0x4050  varx    dw   0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

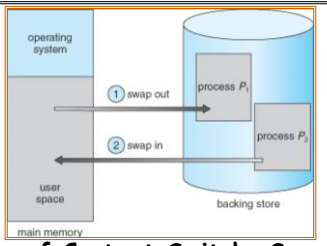
Let's simulate a bit of this code to see what happens (PC=0x240):

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
   Physical address? Base=0x4000, so physical addr=0x4240
   Fetch instruction at 0x4240. Get "la $a0, varx"
   Move 0x4050 → $a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    - » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Schematic View of Swapping

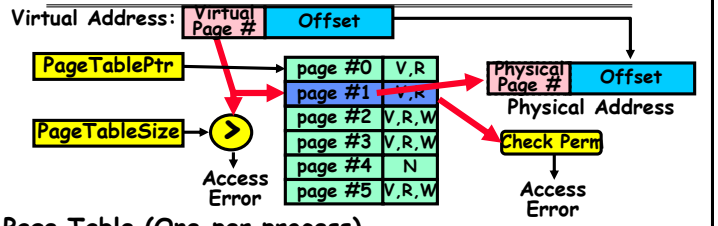


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Paging: Physical Memory in Fixed Size Chunks

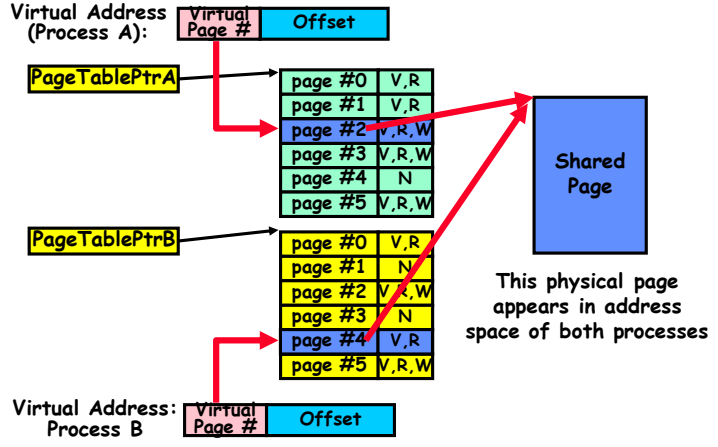
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation: 00110001110001101 … 110010
    - » Each bit represents page of physical memory 1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?

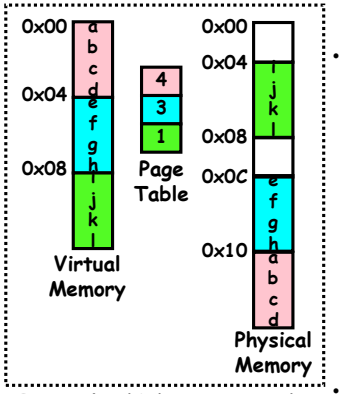


- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



Simple Page Table Discussion



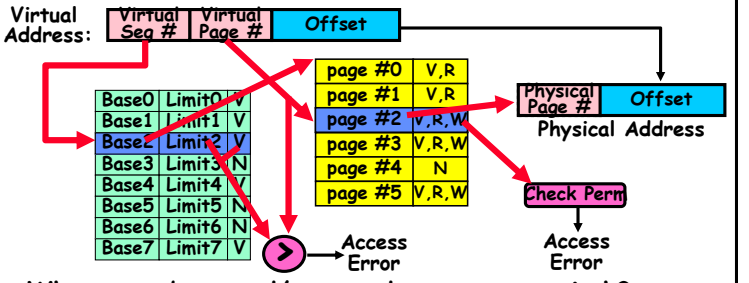
Example (4 byte pages)

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at $(2^{31}-1)$.
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

BREAK

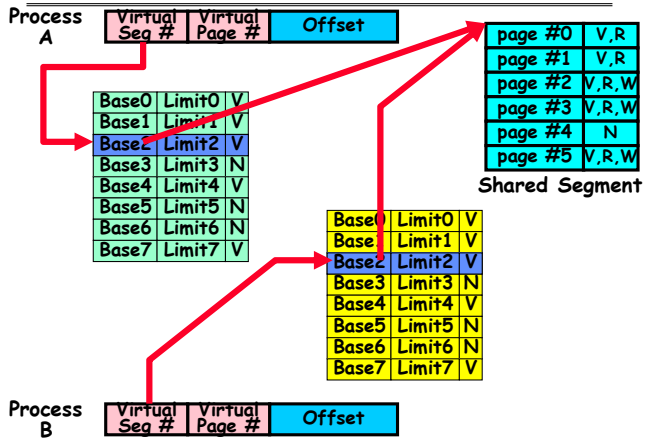
Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

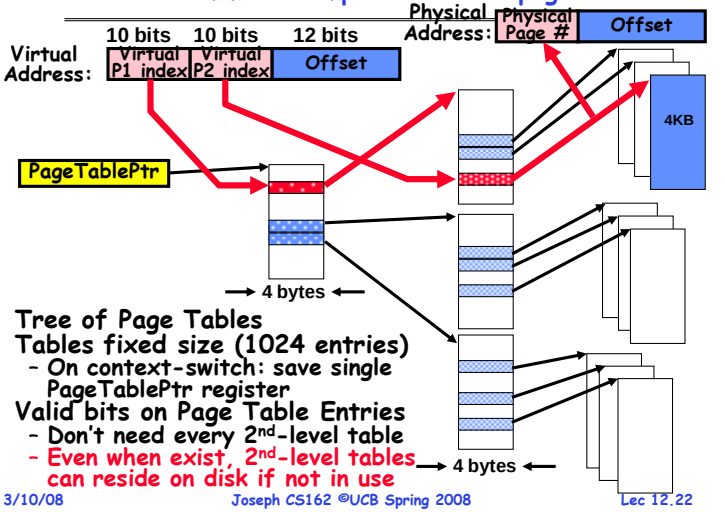


- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

What about Sharing (Complete Segment)?



Another common example: two-level page table



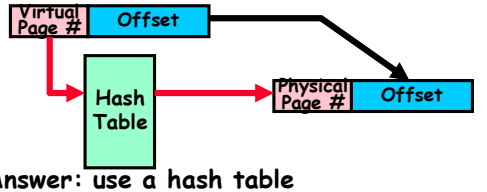
- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other words, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

Inverted Page Table

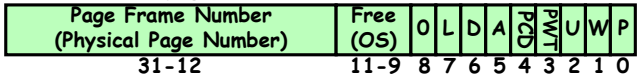
- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

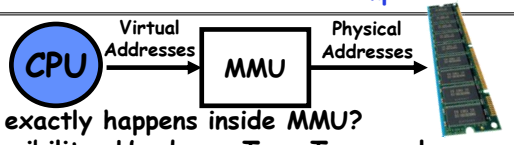


P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

How is the translation accomplished?



- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - » Fault handler will decide what to do
    - » More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
- In fact, need way to cache translations for either case!

Summary (1/2)

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Segment Mapping
  - Segment registers within processor
  - Segment ID associated with each access
    - » Often comes from portion of virtual address
    - » Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    - » Offset (rest of address) adjusted by adding base

Summary (2/2)

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table
  - Size of page table related to physical memory size

CS162
Operating Systems and Systems Programming
Lecture 13

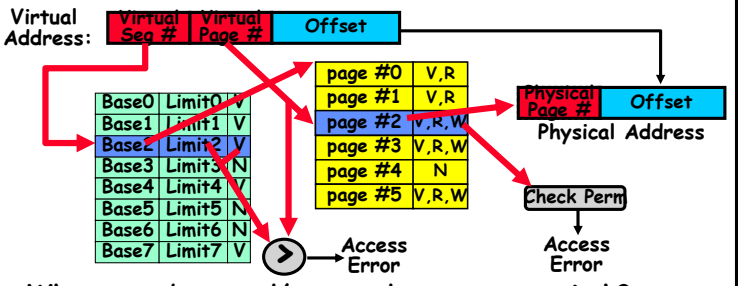
Caches and TLBs

March 12, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

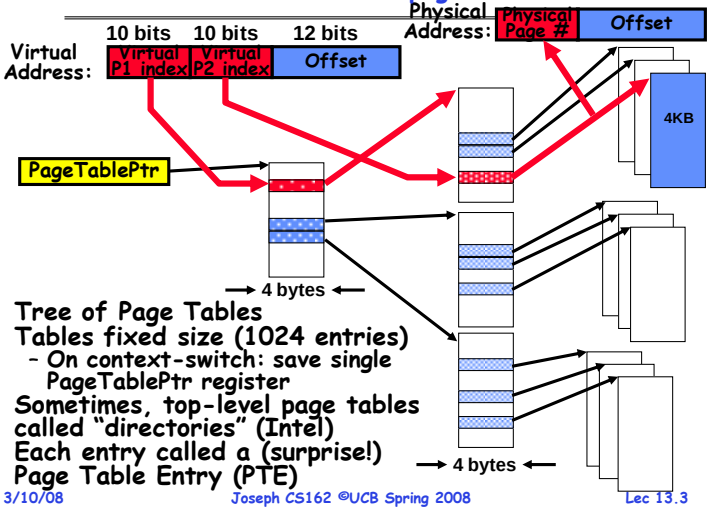
Review: Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):



- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

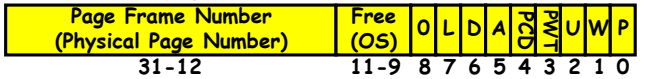
Review: Two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Sometimes, top-level page tables called "directories" (Intel)
- Each entry called a (surprise!) Page Table Entry (PTE)

Review: What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"



P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Goals for Today

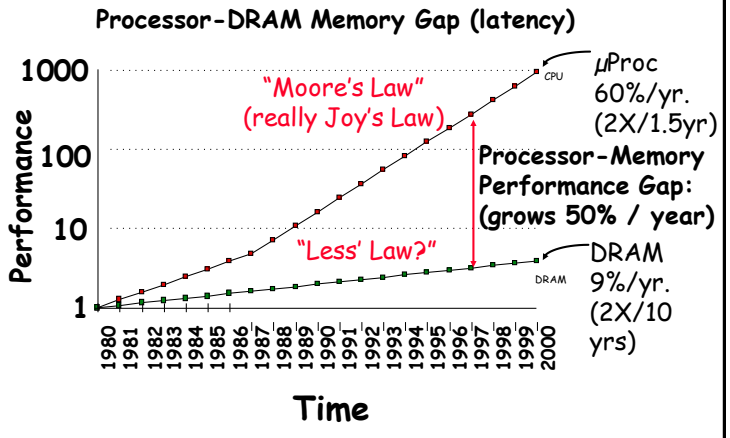
- Caching
- Translation Look-aside Buffers

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

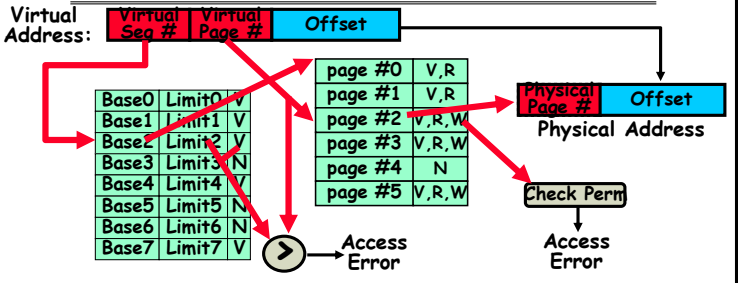
Caching Concept

- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many of the techniques that are used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time =
  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Why Bother with Caching?



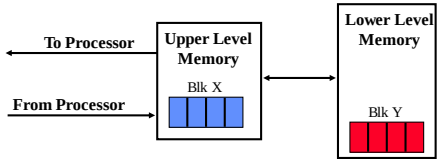
Another Major Reason to Deal with Caching



- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

Why Does Caching Help? Locality!

- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels

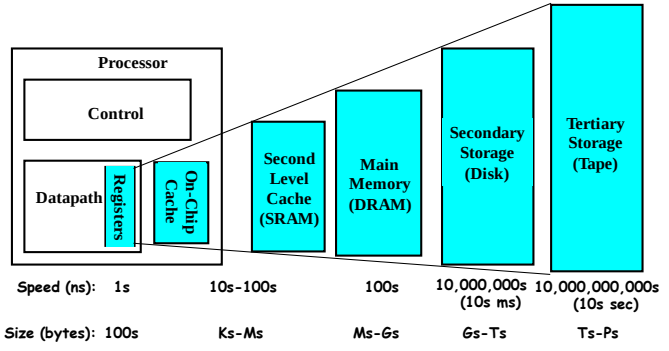


Administrivia

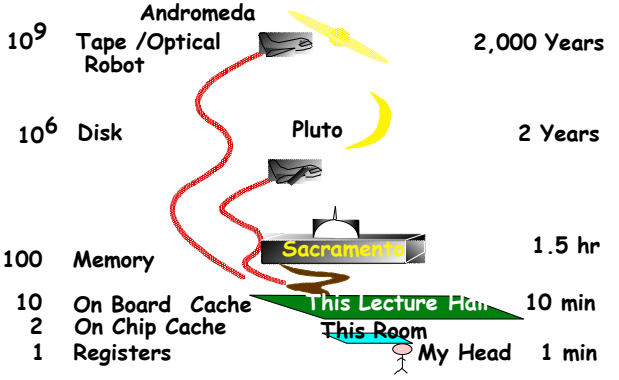
- Project #2 code deadline is next Thu (3/20)
  - Having Eclipse startup problems?
    - » The fix is to delete your ~/.eclipse folder:
      rm -rf ~/.eclipse
      Then restart eclipse to recreate your config, you don't have to delete your workspace
- Please use the CS162 newsgroup for faster response
  - EECS email is significantly delayed this week
- Midterm #2 re-grade requests due by Fri 3/14 5pm
  - Talk with us if your grade is 1-2 std devs below mean
- Attend a CSUA Unix session to better understand Unix
  - CSUA holds them towards the beginning of each semester

Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



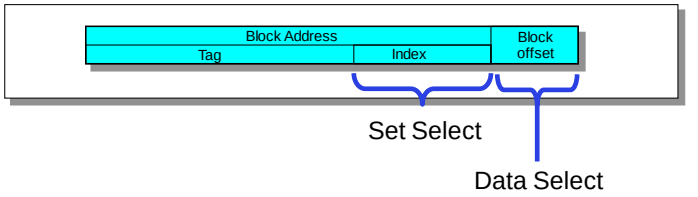
Jim Gray's Storage Latency Analogy: How Far Away is the Data?



A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- **Coherence** (Invalidation): other process (e.g., I/O) updates memory

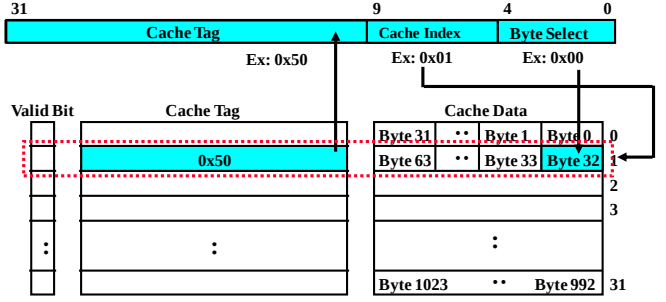
How is a Block found in a Cache?



- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field

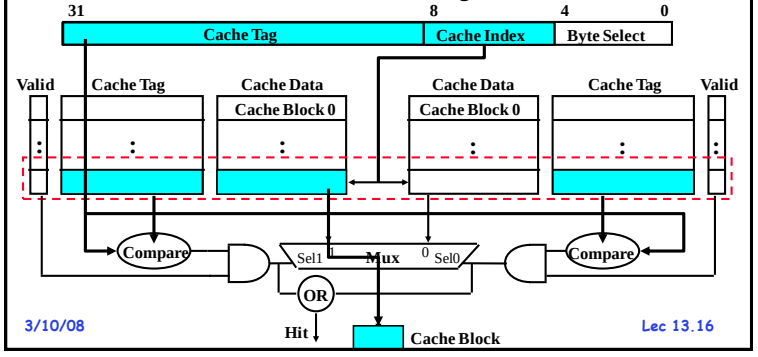
Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The lowest L bits are the Byte Select (Block Size = 2^L)
  - The middle M bits are the Cache Index (Cache Lines = 2^M)
  - The uppermost bits are the Cache Tag (32 – (M + L))
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block, Tag checked to verify block, Byte select chooses byte within block



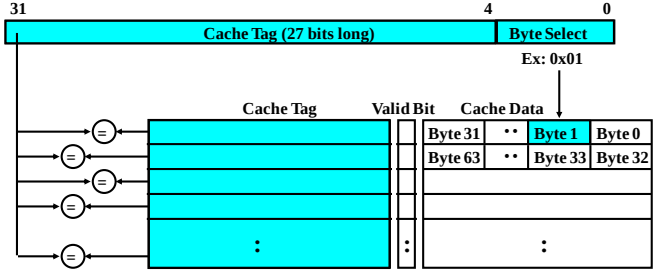
Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



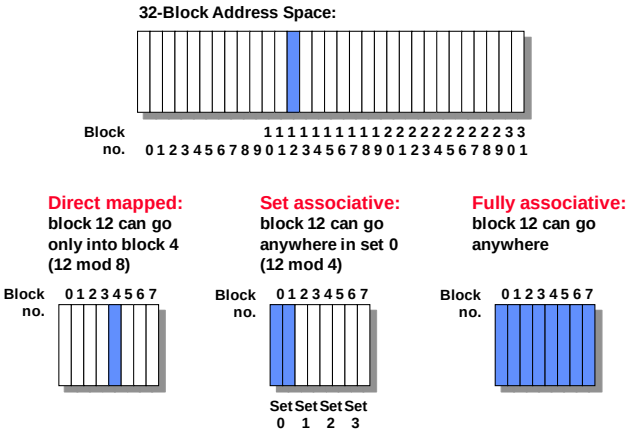
Review: Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



Review: Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache



Review: Which block should be replaced on a miss?

- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

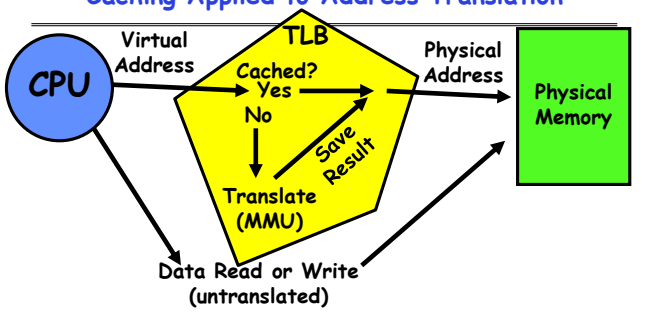
	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Review: What happens on a write?

- Write through: The information is written to both the block in the cache and to the block in the lower-level memory
- Write back: The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: Processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM processor not held up on writes
    - » CON: More complex Read miss may require writeback of dirty data

BREAK

Caching Applied to Address Translation



- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

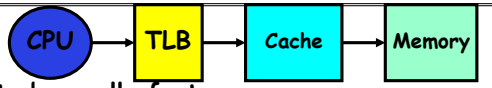
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - » shared segments
    - » user-level portions of an operating system

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

What TLB organization makes sense?



- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

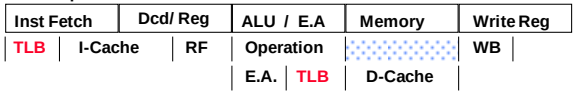
TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

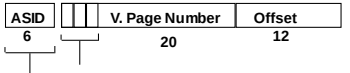
Example: R3000 pipeline includes TLB "stages"

MIPS R3000 Pipeline



TLB
64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

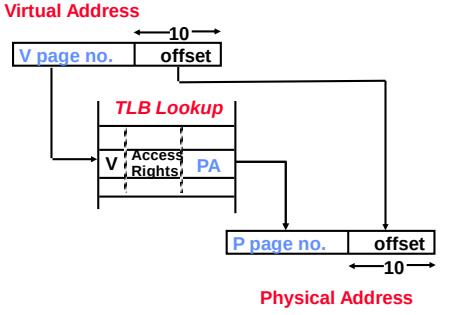


0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Reducing translation time further

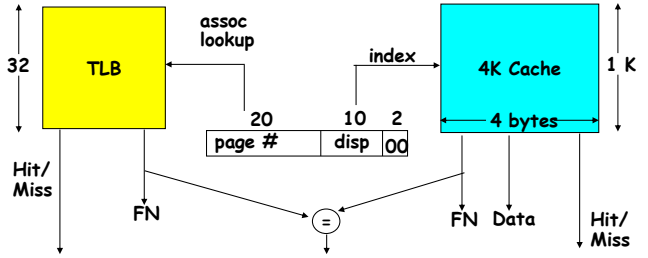
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:



- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

Summary #1/2

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

Summary #2/2: Translation Caching (TLB)

- PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)
- A cache of translations called a "Translation Lookaside Buffer" (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID
- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated somehow
- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast

CS162
Operating Systems and Systems Programming
Lecture 14

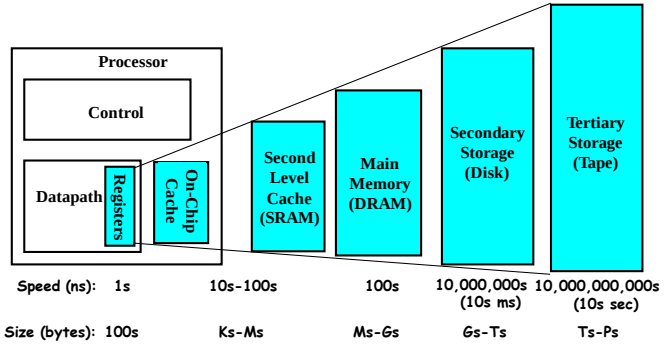
Caching and Demand Paging

March 17, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

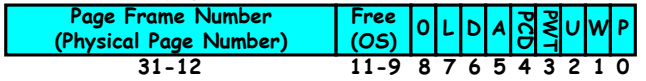
Review: Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Review: What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"



P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Review: Other Caching Questions

- What line gets replaced on cache miss?
  - Easy for Direct Mapped: Only one possibility
  - Set Associative or Fully Associative:
    - » Random
    - » LRU (Least Recently Used)
- What happens on a write?
  - Write through: The information is written to both the cache and to the block in the lower-level memory
  - Write back: The information is written only to the block in the cache
    - » Modified cache block is written to main memory only when it is replaced
    - » Question is block clean or dirty?

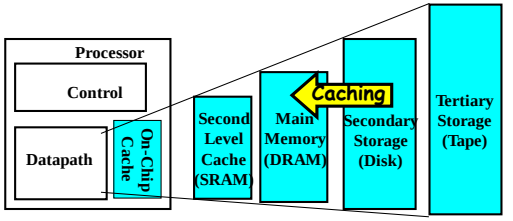
Goals for Today

- Concept of Paging to Disk
- Page Faults and TLB Faults
- Precise Interrupts
- Page Replacement Policies

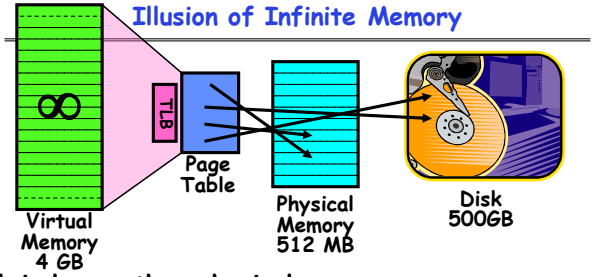
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as cache for disk



Illusion of Infinite Memory



- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging is Caching

- Since Demand Paging is Caching, must ask:
  - What is block size?
    - » 1 page
  - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    - » Fully associative: arbitrary virtual→physical mapping
  - How do we find a page in the cache when look for it?
    - » First check TLB, then page-table traversal
  - What is page replacement policy? (i.e. LRU, Random…)
    - » This requires more explanation… (kinda LRU)
  - What happens on a miss?
    - » Go to lower level to fill miss (i.e. disk)
  - What happens on a write? (write-through, write back)
    - » Definitely write-back. Need dirty bit!

Administrative

- Project 2 code due Thursday 3/20 at 11:59pm
  - Project 2 autograder is up and running every 15 minutes
- Make sure you attend sections!
  - There will be a lot of information about the projects that I cannot cover in class
  - Also supplemental information and detail that we don't have time for in class
- We have an anonymous feedback link on the course homepage
  - Please use to give feedback on course
  - Wednesday: We will have a survey to fill out

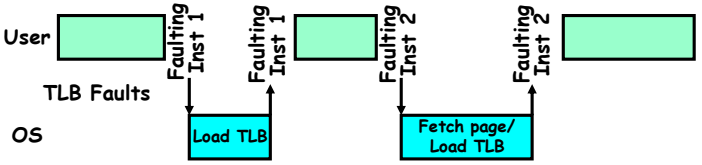
Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Software-Loaded TLB

- MIPS/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without access to page table?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a "TLB Fault"
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Transparent Exceptions



- How to transparently restart faulting instructions?
  - Could we just skip it?
    - » No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    - » Need to know which instruction caused fault
    - » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    - » Save/restore registers, stack, etc
- What if an instruction has side-effects?

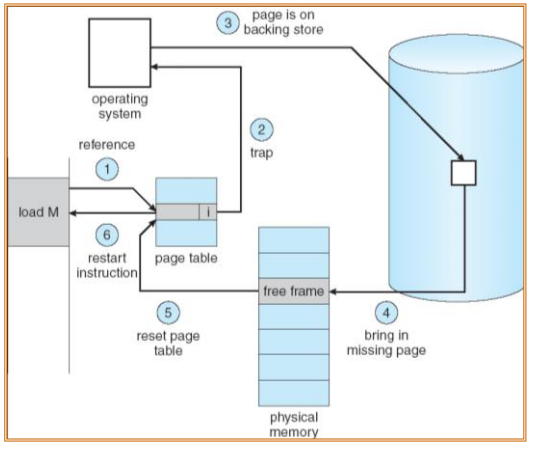
Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    - » Unwind side-effects (easy to restart)
    - » Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`
    - » What if page fault occurs when write to stack pointer?
    - » Did `sp` get incremented before or after the page fault?
  - Example 2: `strcpy (r1), (r2)`
    - » Source and destination overlap: can't unwind in principle!
    - » IBM S/370 and VAX solution: execute twice – once read-only
- What about "RISC" processors?
  - For instance delayed branches?
    - » Example:
      ```
      bne somewhere
      ld r1,(sp)
      ```
    - » Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    - » Example:
      ```
      div r1, r2, r3
      ld r1, (sp)
      ```
    - » What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Steps in Handling a Page Fault



Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

    EAT = (1 – p) x 200ns + p x 8 ms
    = (1 – p) x 200ns + p x 8,000,000ns
    = 200ns + p x 7,999,800ns
- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - 200ns x 1.1 < EAT ⇒ p < 2.5 x 10^{-6}
  - This is about 1 page fault in 400000!

Review: What Factors Lead to Misses?

- Compulsory Misses:
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - » Prefetching: loading them into memory before needed
    - » Need to predict future somehow! More later…
- Capacity Misses:
  - Not enough memory – must somehow increase size
    - » One option: Increase amount of DRAM (not quick fix!)
    - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses (collision):
  - Doesn't exist in virtual memory ("fully-associative" cache)
- Policy Misses:
  - Replacement policy kicks pages out of memory early
  - How to fix? Better replacement policy
- Coherence Misses (invalidation):
  - Not a problem for virtual memory, since all processes/cores share same memory

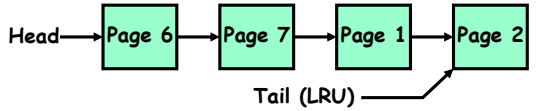
BREAK

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

When will LRU perform badly?

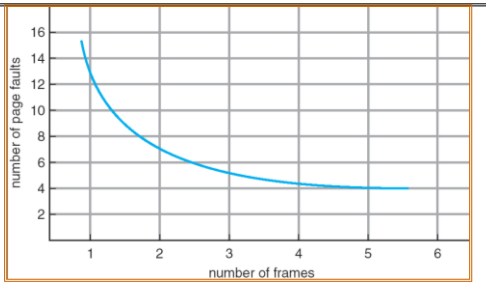
- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- MIN Does much better:

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: BeLady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page

- Details:
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO

Summary

- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss ⇒ get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Software-loaded TLB
  - Fast Path: handled in hardware (TLB hit with valid=1)
  - Slow Path: Trap to software to scan page table
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: replace page that will be used farthest in future
  - LRU: Replace page that hasn't be used for the longest time
  - Clock Algorithm: Approximation to LRU

CS162
Operating Systems and Systems Programming
Lecture 15

Page Allocation and Replacement

March 19, 2008

Prof. Anthony D. Josep
http://inst.eecs.berkeley.edu/~cs162

Review: Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Review: Software-Loaded TLB

- MIPS/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without hardware TLB fill?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a TLB Fault
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Review: Implementing LRU

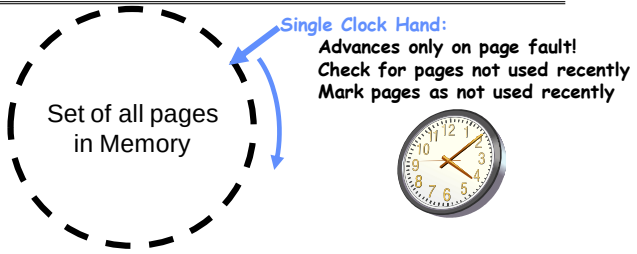
- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page
- Details:
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO

Goals for Today

- Page Replacement Policies
  - Clock Algorithm, Nth chance algorithm, 2nd-Chance-List Algorithm
- Page Allocation Policies
- Working Set/Thrashing
- Distributed Problems
  - Brief History
  - Parallel vs. Distributed Computing
  - Parallelization and Synchronization
  - Prelude to MapReduce

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne; and from slides licensed under the Creative Commons Attribution 2.5 License by the University of Washington (2007). Many slides generated from my lecture notes by Kubiatowicz.

Clock Algorithm: Not Recently Used



- What if hand moving slowly?
  - Good sign or bad sign?
    - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

Nth Chance version of Clock Algorithm

- Nth chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

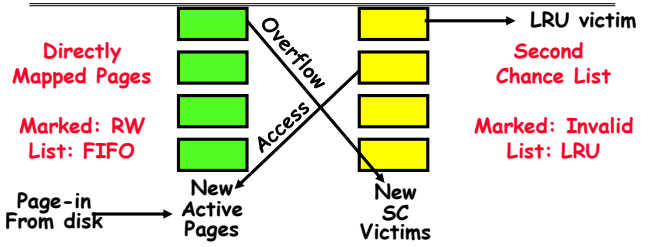
Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - » For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - » Initially, mark all pages as read-only, even data pages
    - » On write, trap to OS. OS sets software "modified" bit, and marks page as read-write.
    - » Whenever page comes back in from disk, mark read-only

Clock Algorithms Details (continued)

- Do we really need a hardware-supported "use" bit?
  - No. Can emulate it similar to above:
    - » Mark all pages as invalid, even if in memory
    - » On read to invalid page, trap to OS
    - » OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous:
    - » On write, trap to OS (either invalid or read-only)
    - » Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)



- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ LRU, but page fault on every page reference
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn't VAX include "use" bit?
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did OK anyway

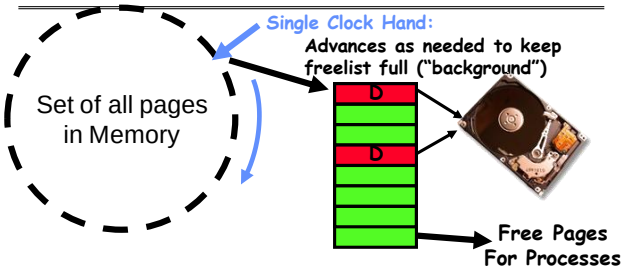
Administrivia

- All slides except last lecture are (finally) online!
  - Reader should be available Friday
- Project 2 code due Thursday 3/20 at 11:59pm

Aside: Powers of 10 and 2

- Strict powers of 10:
  - yotta: 10^{24}
  - exa: 10^{18}
  - peta: 10^{15}
  - tera: 10^{12}
  - giga: 10^{9}
  - mega: 10^{6}
  - kilo: 10^{3}
  - milli(m): 10^{-3}
  - micro (μ): 10^{-6}
  - nano(n): 10^{-9}
  - pico: 10^{-12}
  - femto: 10^{-15}
  - atto: 10^{-18}
  - yocto: 10^{-24}
- Strict powers of 2:
  - yotta: $2^{80} \cong 10^{24}$
  - exa: $2^{60} \cong 10^{18}$
  - peta: $2^{50} \cong 10^{15}$
  - tera: $2^{40} \cong 10^{12}$
  - giga: $2^{30} = 1{,}073{,}741{,}824 \cong 10^{9}$
  - mega: $2^{20} = 1{,}048{,}576 \cong 10^{6}$
  - kilo: $2^{10} = 1024 \cong 10^{3}$
- When to use one or the other?
  - Powers of 2
    - » Memory sizes
  - Powers of 10
    - » Time
    - » Bandwidth

Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit? Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - » Can't push page out to disk without invalidating all PTEs

Allocation of Page Frames (Memory Pages)

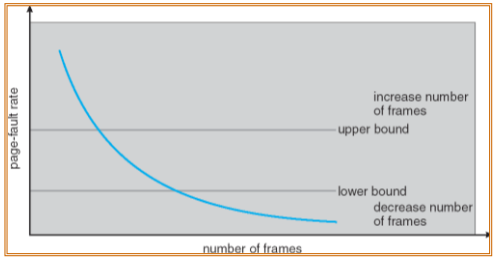
- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes⇒process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:
    s_i = size of process p_i and $S = \Sigma s_i$
    m = total number of frames
    a_i = allocation for $p_i = \frac{s_i}{S} \times m$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

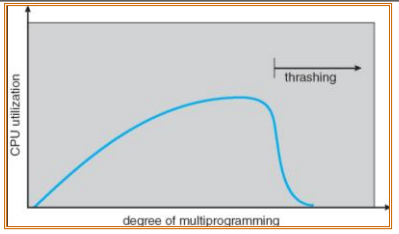
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

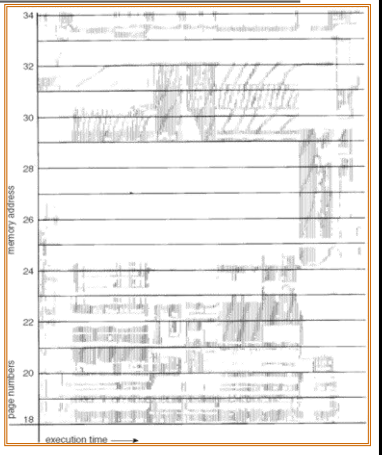
Thrashing



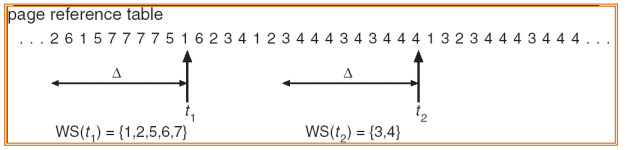
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model



- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma |WS_i|$ ≡ total demand frames
- if $D > m$ ⇒ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

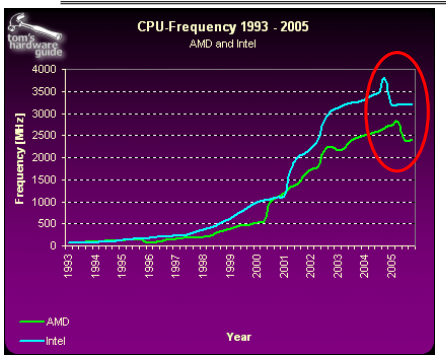
- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages "around" the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

BREAK

Computer Speedup



Moore's Law: "*The density of transistors on a chip doubles every 18 months, for the same cost*" (1965)

- What can you do with 1 computer?
- What can you do with 100 computers?
- What can you do with an entire data center?

Image: Tom's Hardware

Some Distributed Problems

- Rendering multiple frames of high-quality animation

- Simulating several hundred or thousand characters

Happy Feet © Kingdom Feature Productions; *Lord of the Rings* © New Line Cinema; *Shrek* © DreamWorks Animation

More Distributed problems

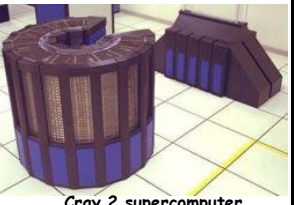
- Indexing the web (Google)
- Simulating an Internet-sized network for networking experiments (PlanetLab, DETER)
- Speeding up content delivery (Akamai)

- *What is the key attribute that all these examples have in common?*

- Distributed computing:
  - Multiple CPUs/cores across many computers (MIMD)
- Parallel computing:
  - Vector processing of data (SIMD)
  - Multiple CPUs/cores in a single computer (MIMD)

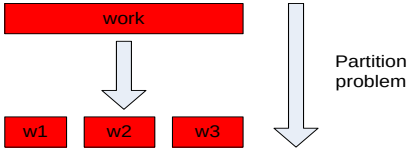
A Brief History… 1975-85

- 1975-85:
  - Primarily vector-based parallel computing in early years
  - Gradually more thread-based parallelism introduced
- 1985-95:
  - "Massively parallel architectures" rise in prominence
  - Message Passing Interface (MPI) and other libs developed
  - Bandwidth was a big problem
- 1995-today:
  - Berkeley Network of Workstations
    - » COTS tech instead of special node machines
  - Cluster/grid architecture increasing
  - Web-wide cluster software
    - » Microsoft, Google, Amazon take this to the extreme (thousands of nodes/cluster)


Cray 2 supercomputer (Wikipedia)

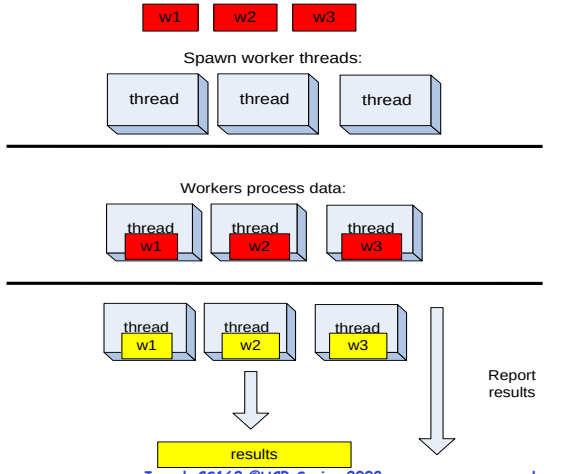
Parallelization Idea

- Parallelization is "easy" if processing can be cleanly split into n units:



- In a parallel computation, we would like to have as many threads as we have processors
  - Ex. 4-CPU computer would be able to run four threads at the same time

Parallelization Idea (2)



Parallelization Pitfalls

- This model is too simple!
  - How do we assign work units to worker threads?
  - What if we have more work units than threads?
  - How do we aggregate the results at the end?
  - How do we know all the workers have finished?
  - What if the work cannot be divided into completely separate tasks?
- Multiple threads must communicate with one another, or access a shared resource
  - We need a *synchronization system!*

Prelude to MapReduce

- Explicit parallelism/synchronization is really hard!!
  - Must consider all possible shared state, keep locks organized, and use them consistently and correctly
- Knowing there are bugs may be tricky; fixing them can be even worse!
- Solution: *Minimize shared state to reduce total system complexity*
- But, synchronization doesn't address distributed computing questions (e.g., moving data around)
- Fortunately, MapReduce handles this for us
  - Google-designed paradigm for making large subset of distributed problems easier to code
  - Automates data distribution & result aggregation
  - Restricts the ways data can interact to eliminate locks (no shared state = no locks!)

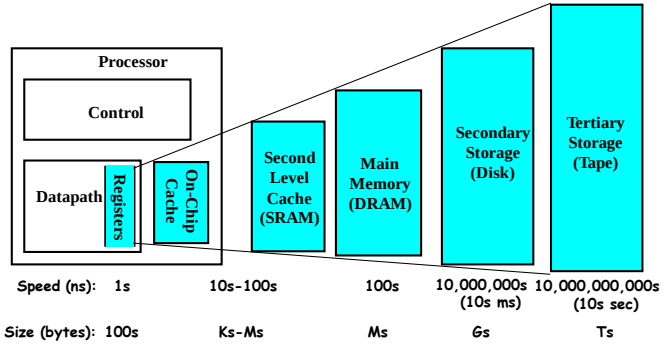
CS162
Operating Systems and Systems Programming
Lecture 16

I/O Systems

March 31, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Goals for Today

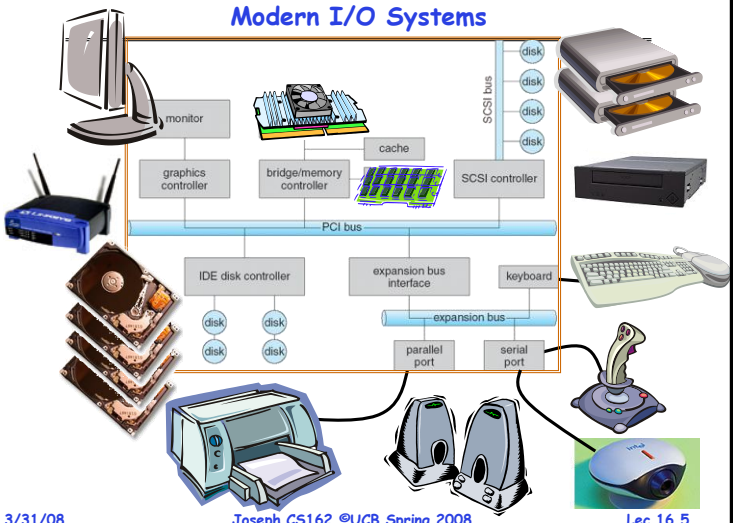
- I/O Systems
  - Hardware Access
  - Device Drivers
- Queuing Theory

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

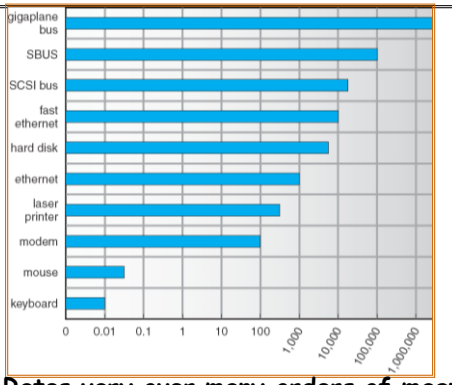
The Requirements of I/O

- So far in this course:
  - We have learned how to manage CPU, memory
- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But… thousands of devices, each slightly different
    - » How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    - » How can we make them reliable???
  - Devices unpredictable and/or slow
    - » How can we manage them if we don't know what they will do or how they will perform?
- Some operational parameters:
  - Byte/Block
    - » Some devices provide single byte at a time (*e.g.* keyboard)
    - » Others provide whole blocks (*e.g.* disks, networks, etc)
  - Sequential/Random
    - » Some devices must be accessed sequentially (*e.g.* tape)
    - » Others can be accessed randomly (*e.g.* disk, cd, etc.)
  - Polling/Interrupts
    - » Some devices require continual monitoring
    - » Others generate interrupts when they need service

Modern I/O Systems



Example Device-Transfer Rates (Sun Enterprise 6000)



- Device Rates vary over many orders of magnitude
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices
  - This code works on many different devices:

```
FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd,"Count %d\n",i);
}
close(fd);
```

  - Why? Because code that controls devices ("device driver") implements standard interface.
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

Administrivia

- Would you like an extra 5% for your course grade?
  - Attend lectures and sections! 5% of grade is participation
  - Midterm 1 was only 15%
- Project #3 design doc due next Monday (4/7) at 11:59pm
- Midterm #2 is in two weeks (Wed 4/16) 6-7:30pm in 10 Evans

Want Standard Interfaces to Devices

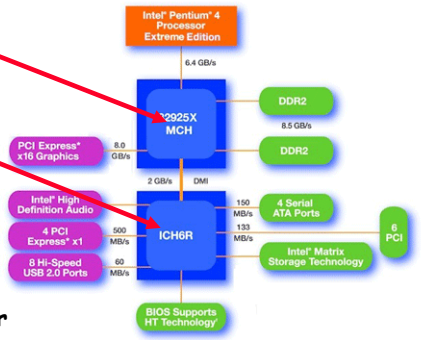
- Block Devices: *e.g.* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- Network Devices: *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

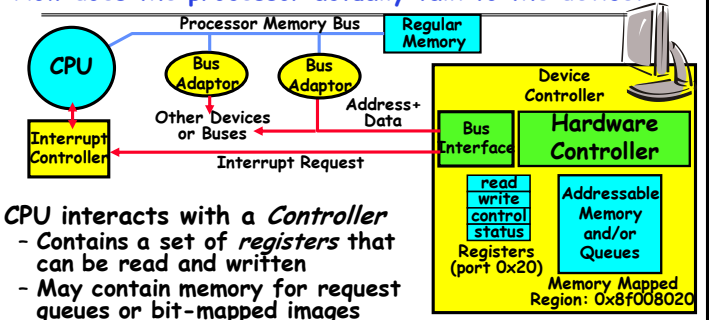
- Blocking Interface: "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Main components of Intel Chipset: Pentium 4

- Northbridge:
  - Handles memory
  - Graphics
- Southbridge: I/O
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers



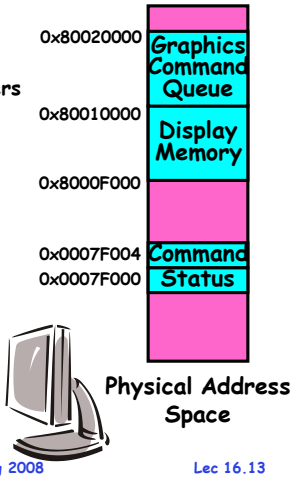
How does the processor actually talk to the device?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

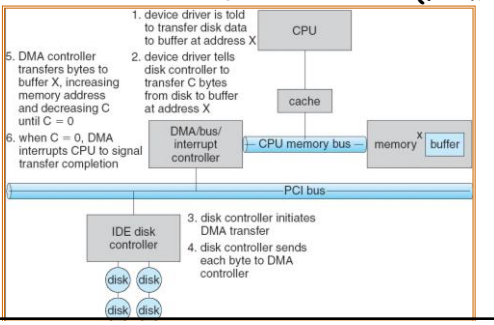
Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- Can protect with page tables

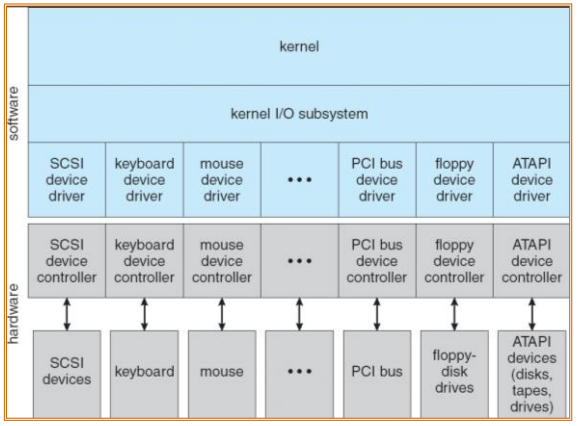


Transfering Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- Sample interaction with DMA controller (from book):



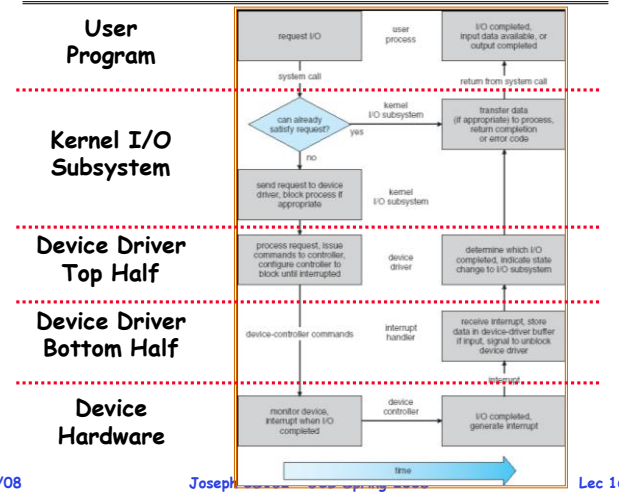
A Kernel I/O Structure



Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » Implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

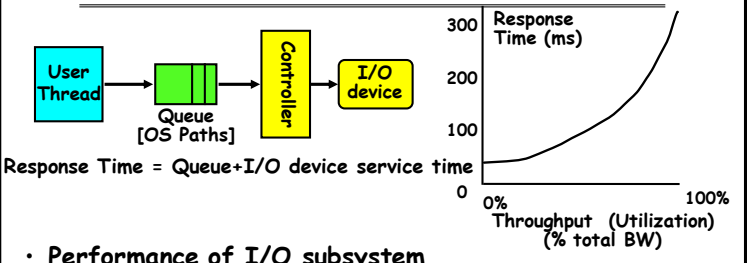


I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Handled in bottom half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

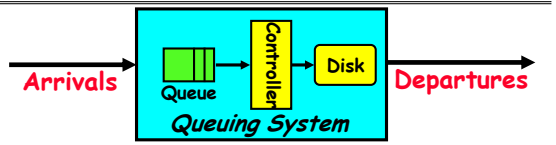
BREAK

I/O Performance



- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization approaches 100%

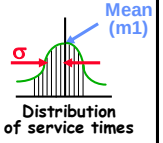
Introduction to Queuing Theory



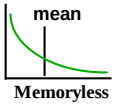
- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little's Law:
  Mean # tasks in system = arrival rate x mean response time
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks
  - Typical queuing theory doesn't deal with transient behavior, only steady-state behavior

Background: Use of random distributions

- Server spends variable time with customers
  - Mean (Average) m1 = $\Sigma p(T) \times T$
  - Variance $\sigma^2 = \Sigma p(T) \times (T-m1)^2 = \Sigma p(T) \times T^2 - m1$
  - Squared coefficient of variance: $C = \sigma^2/m1^2$
    Aggregate description of the distribution.



- Important values of C:
  - No variance or deterministic ⇒ C=0
  - "memoryless" or exponential ⇒ C=1
    - » Past tells nothing about future
    - » Many complex systems (or aggregates) well described as memoryless
  - Disk response times C ≈ 1.5 (wider variance ⇒ long tail)



A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless

- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution (C = 1):
    - » Called M/M/1 queue: $T_q = T_{ser} \times u/(1-u)$
  - General service distribution (no restrictions), 1 server:
    - » Called M/G/1 queue: $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1-u))$

A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (C=1.0)
  - Avg. service = 20 ms (From controller+seek+rot+trans)
- Questions:
  - How utilized is the disk?
    - » Ans: server utilization, $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    - » Ans: T_q
  - What is the number of requests in the queue?
    - » Ans: $L_q = \lambda T_q$ (Little's law)
  - What is the avg response time for disk request?
    - » Ans: $T_{sys} = T_q + T_{ser}$
- Computation:
  λ (avg # arriving customers/s) = 10/s
  T_{ser} (avg time to service customer) = 20 ms (0.02s)
  u (server utilization) = $\lambda \times T_{ser}$ = 10/s x .02s = 0.2
  T_q (avg time/customer in queue) = $T_{ser} \times u/(1-u)$
  = 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)
  L_q (avg length of queue) = $\lambda \times T_q$ =10/s x .005s = 0.05
  T_{sys} (avg time/customer in system) = $T_q + T_{ser}$ = 25 ms

Summary

- Working Set: Set of pgs touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process
- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns: block, char, net devices
  - Different Access Timing: Non-/Blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - CPU accesses thru I/O insts, ld/st to special phy memory
  - Report results thru interrupts or a status register polling
- Device Driver: Device-specific code in kernel
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency $\rightarrow \infty$

$$T_q = T_{ser} \times \tfrac{1}{2}(1+C) \times u/(1-u))$$

CS162
Operating Systems and Systems Programming
Lecture 17

Disk Management and File Systems

April 2, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Want Standard Interfaces to Devices

- Block Devices: *e.g.* disk drives, tape drives, Cdrom
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- Network Devices: *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

Review: How Does User Deal with Timing?

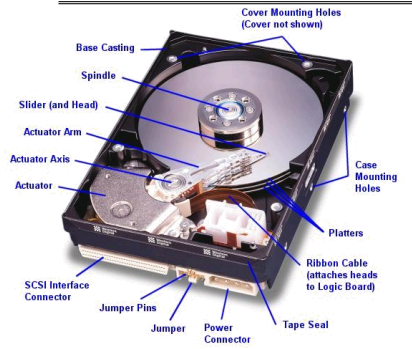
- Blocking Interface: "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Goals for Today

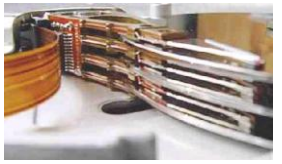
- Disk Performance
  - Hardware performance parameters
- File Systems
  - Structure, Naming, Directories

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Hard Disk Drives



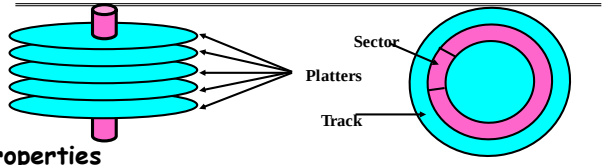
Western Digital Drive
http://www.storagereview.com/guide/



Read/Write Head
Side View

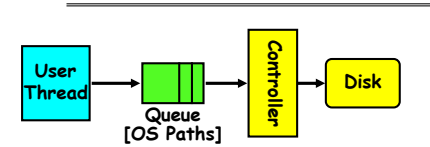
IBM/Hitachi Microdrive

Properties of a Hard Magnetic Disk

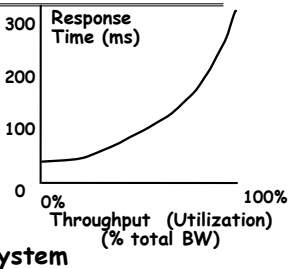


- Properties
  - Independently addressable element: sector
    - » OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
    - » A sector is the smallest unit that can be read or written
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

Disk I/O Performance

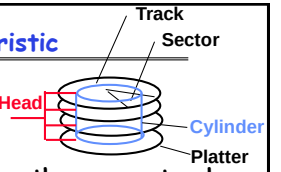


Response Time = Queue+Disk Service Time

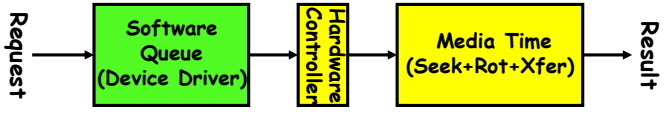


- Performance of disk drive/file system
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » Physical disk media
- Queuing behavior:
  - Can lead to big increases of latency as utilization approaches 100%

Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

Typical Numbers of a Magnetic Disk

- Average seek time as reported by the industry:
  - Typically in the range of 8 ms to 12 ms
  - Due to locality of disk reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - *Most* disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 512B – 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: 2 to 50 MB per second
- Controller time depends on controller hardware
- Cost drops by factor of two per year (since 1991)

Disk Performance Examples

- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms,
  - 7500RPM ⇒ Time for one rotation: 8ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Read sector from random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 10ms to fetch/put data: 100 KByte/sec
- Read sector from random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Approx 5ms to fetch/put data: 200 KByte/sec
- Read next sector on same track:
  - Transfer (0.25ms): 4 MByte/sec
- Key to using disk effectively (esp. for filesystems) is to minimize seek and rotational delays

Administrivia

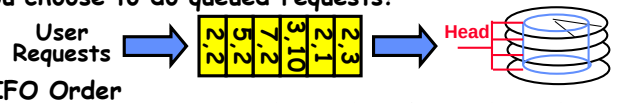
- Group problems? Don't wait.
  - Talk to TA/talk to me – Let's get things fixed!
- Project #3 design doc due next Monday (4/7) at 11:59pm
- Midterm #2 is in two weeks (Wed 4/16)
  - 6-7:30pm in 10 Evans
  - All material from projects 1-3, lectures #9 (2/25) to #19 (4/9)
    - » OS History, Services, and Structure; CPU Scheduling; Kernel and Address Spaces; Address Translation, Caching and TLBs; Demand Paging; I/O Systems; Filesystems, Disk Management, Naming, and Directories; Distributed Systems

Disk Tradeoffs

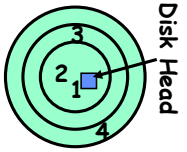
- How do manufacturers choose disk sector sizes?
  - Need 100-1000 bits between each sector to allow system to measure how fast disk is spinning and to tolerate small (thermal) changes in track length
- What if sector was 1 byte?
  - Space efficiency – only 1% of disk has useful space
  - Time efficiency – each seek takes 10 ms, transfer rate of 50 – 100 Bytes/sec
- What if sector was 1 KByte?
  - Space efficiency – only 90% of disk has useful space
  - Time efficiency – transfer rate of 100 KByte/sec
- What if sector was 1 MByte?
  - Space efficiency – almost all of disk has useful space
  - Time efficiency – transfer rate of 4 MByte/sec

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation

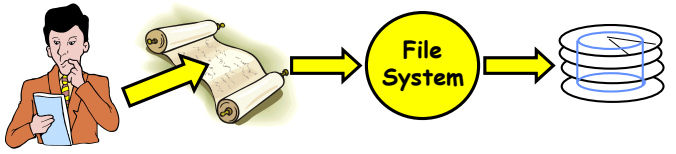


- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle

Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files (next lecture)
- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - » First case: OS/BIOS must deal with bad sectors
    - » Second case: hardware shields OS from structure of disk
- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

Designing the File System: Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with JOSEPH")
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- Digression, danger of predicting future:
  - In 1950's, marketing study by IBM said total worldwide need for computers was 7!
  - Company (that you haven't heard of) called "GenRad" invented oscilloscope; thought there was no market, so sold patent to Tektronix (bet you have heard of them!)

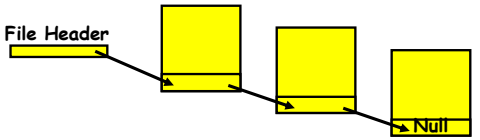
BREAK

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - » Analogous to base+bounds in virtual memory
    - » User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - » What if not enough contiguous space for new file?
  - File Header Contains:
    - » First block/LBA in file
    - » File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - » Free holes get smaller and smaller
    - » Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

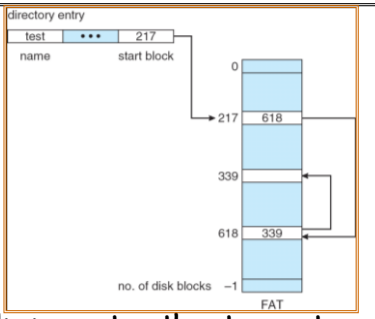
Linked List Allocation

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



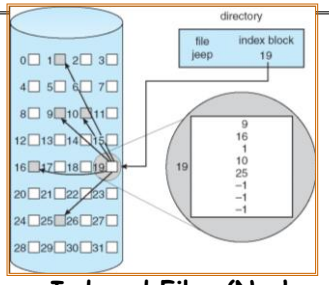
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks

Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

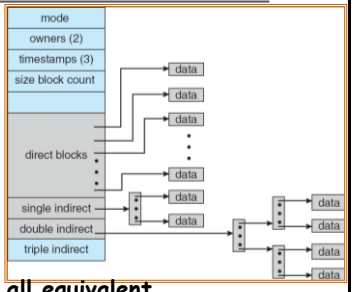
Indexed Allocation



- Third Technique: Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index Random access is fast
  - Cons: Clumsy to grow file bigger than table size Still lots of seeks: blocks may be spread over disk

Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files



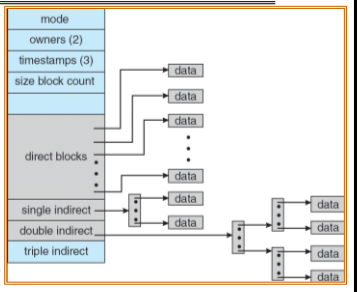
- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Multilevel Indexed Files (UNIX 4.1): Discussion

- Basic technique places an upper limit on file size that is approximately 16Gbytes
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Fallacy: today, EOS producing 2TB of data per day
- Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/Os per block!)

Summary

- Disk Performance:
  - Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: spec of disk depends on rotation speed and bit storage density
- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File (and directory) defined by header
  - Called "inode" with index called "inumber"
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..

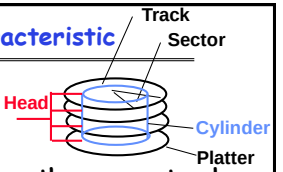
CS162
Operating Systems and Systems Programming
Lecture 18

File Systems, Naming, and Directories

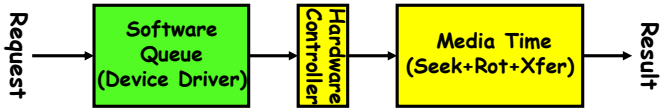
April 7, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Magnetic Disk Characteristic



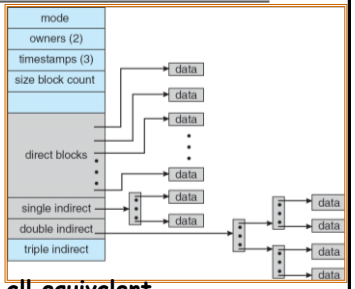
- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - transfer large group of blocks sequentially from one track

Multilevel Indexed Files (UNIX 4.1)

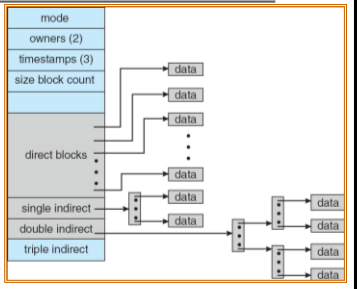
- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files



- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/Os per block!)

Multilevel Indexed Files (UNIX 4.1): Discussion

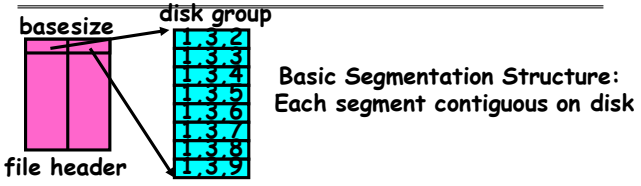
- Basic technique places an upper limit on file size that is approximately 16Gbytes
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Fallacy: today, EOS producing 2TB of data per day
- Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

Goals for Today

- File Systems
  - Structure, Naming, Directories
- Caching in File Systems

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

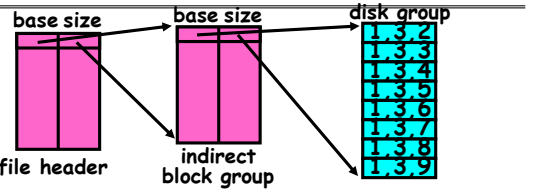
File Allocation for Cray-1 DEMOS



Basic Segmentation Structure: Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group ⇒ 80GB max file
    - » Assuming 8KB blocks, 8byte entries⇒ (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply Easy to find free block groups (when disk not full)
  - Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access
  - Full disk ⇒ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

How to keep DEMOS performing well?

- In many systems, disks are always full
  - CS department growth: 300 GB to 1TB in a year
    - » That's 2GB/day! (Now at 3—4 TB!)
  - How to fix? Announce that disk space is getting low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

Administrivia

- Plan Ahead: this month will be difficult!!
  - Project deadlines every week
- Project #3 design doc due today at 11:59pm
- Midterm #2 is next Wednesday (April 16th)
  - 6-7:30pm in 10 Evans
  - All material from projects 1-3, lectures #9 (2/25) to #19 (4/9)
    - » OS History, Services, and Structure; CPU Scheduling; Kernel and Address Spaces; Address Translation, Caching and TLBs; Demand Paging; I/O Systems; Filesystems, Disk Management, Naming, and Directories; Distributed Systems

UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other
- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!

  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

How do we actually access files?

- All information about a file contained in its file header
  - UNIX calls this an "inode"
    - » Inodes are global resources identified by index ("inumber")
  - Once you load the header structure, all the other blocks of the file are locatable
- Question: how does the user ask for a particular file?
  - One option: user specifies an inode by a number (index).
    - » Imagine: open("14553344")
  - Better option: specify by textual name
    - » Have to map name→inumber
  - Another option: Icon
    - » This is how Apple made its money. Graphical user interfaces. Point to a file and click.
- Naming: The process by which a system translates from user-visible names to system resources
  - In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes
  - For global file systems, data may be spread over globe⇒need to translate from strings or icons to some combination of physical server location and inumber

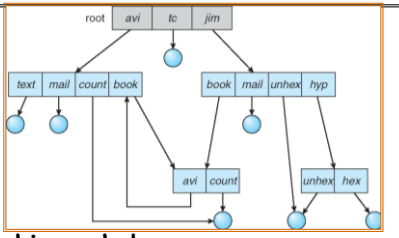
Directories

- Directory: a relation used for naming
  - Just a table of (file name, inumber) pairs
- How are directories constructed?
  - Directories often stored in files
    - » Reuse of existing mechanism
    - » Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - » Options: Simple list or Hashtable
    - » Can be cached into memory in easier form to search
- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
  - Ties to file creation/destruction
    - » On creating a file by name, new inode grabbed and associated with new file in particular directory

Directory Organization

- Directories organized into a hierarchical structure
  - Seems standard, but in early 70's it wasn't
  - Permits much easier organization of data structures
- Entries in directory can be either files or directories
- Files named by ordered set (e.g., /programs/p/list)

Directory Structure



- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - Soft Links: "shortcut" pointers to other files
    - » Implemented by storing the logical name of actual file
- Name Resolution: The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

Directory Structure (Con't)

- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data bock for root
    - » Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"
- Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

Where are inodes stored?

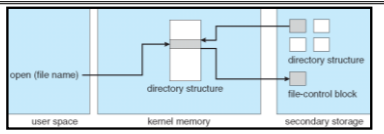
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored anywhere near the data blocks. To read a small file, seek to get header, see back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

Where are inodes stored?

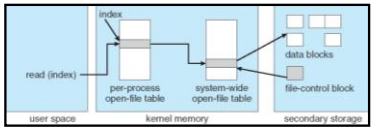
- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - » UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder⇒no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  - Part of the Fast File System (FFS)
    - » General optimization to avoid seeks

BREAK

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

Summary

- Cray DEMOS: optimization for sequential access
  - Inode holds set of disk ranges, similar to segmentation
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc
  - Optimizations for sequential access: start new files in open ranges of free blocks
  - Rotational Optimization
- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
- Buffer cache used to increase performance
  - Read Ahead Prefetching and Delayed Writes

CS162
Operating Systems and Systems Programming
Lecture 19

File Systems continued
Distributed Systems

April 9, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Data Durability
- Beginning of Distributed Systems Discussion
  - Lisp/ML map/fold review
  - MapReduce overview

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- RAID: Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

Log Structured and Journaled File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*.
    - » A transaction either happens *completely* or *not at all*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - Log Structured Filesystem (LFS): data stays in log form
  - Journaled Filesystem: Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), NTFS (Windows)

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter
  - fun foo(L: int list) = sum(L) + mul(L) + length(L)
  - Order of sum(), mul(), length() does not matter – they do not modify *L*

Functional Updates Do Not Modify Structures

- fun append(x, lst) =
  let lst' = reverse lst in
  reverse (x :: lst')
- The append() function above reverses a list, adds a new element to the front, and returns all of that, reversed, which appends an item
- But, it *never modifies lst*!

Functions Can Be Used As Arguments

- fun DoDouble(f, x) = f (f x)
- It does not matter what f does to its argument; DoDouble() will do it twice
- *What is the type of this function?*

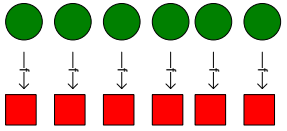
Administrivia

- Project zero-sum game:
  - In the end, we decide how to distribute pts to partners
    - » Normally, we are pretty even about this
    - » But, under extreme circumstances, may take points from non-working members and give to working members
  - This is a zero-sum game!
- Make sure to do your project evaluations
  - This is supposed to be an individual evaluation, not done together as a group
  - This is part of the information that we use to decide how to distributed points
  - We will give 0 (ZERO) to people who don't fill out evals
- Midterm #2 is next Wednesday (April 16th)
  - 6-7:30pm in 10 Evans
  - Covers projects 1-3, lectures #9 (2/25) to #19 (4/9)
    - » OS History, Services, and Structure; CPU Scheduling; Kernel and Address Spaces; Address Translation, Caching and TLBs; Demand Paging; I/O Systems; Filesystems, Disk Management, Naming, and Directories; Distributed Systems

Map

map f lst: ('a→'b) → ('a list) → ('b list)

Creates a new list by applying f to each element of the input list; returns output in order

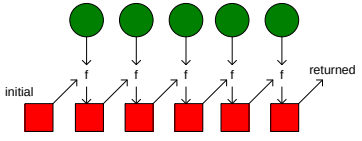


Fold

fold f x_0 lst: ('a*'b→'b)→'b→('a list)→'b

Moves across a list, applying *f* to each element plus an *accumulator*

f returns the next accumulator value, which is combined with the next element of the list



fold left vs. fold right

- Order of list elements can be significant
  - Fold left moves left-to-right across the list
  - Fold right moves from right-to-left

SML Implementation:

```
        fun foldl f a []       = a
| foldl f a (x::xs) = foldl f (f(x, a)) xs

        fun foldr f a []       = a
| foldr f a (x::xs) = f(x, (foldr f a xs))
```

Example

- fun foo(l: int list) = sum(l) + mul(l) + length(l)
- How can we implement this?
- fun sum(lst) = foldl (fn (x,a)=>x+a) 0 lst
- fun mul(lst) = foldl (fn (x,a)=>x*a) 1 lst
- fun length(lst) = foldl (fn (x,a)=>1+a) 0 lst

More Complicated Problems

- More complicated fold problem
  - Given a list of numbers, how can we generate a list of partial sums?
    - » e.g.: [1, 4, 8, 3, 7, 9] → [0, 1, 5, 13, 16, 23, 32]
- More complicated map problem
  - Given a list of words, can we: reverse the letters in each word, and reverse the whole list, so it all comes out backwards?
    - » e.g.: ["my", "happy", "cat"] → ["tac", "yppah", "ym"]

map Implementation

```
fun map f []       = []
| map f (x::xs) = (f x) :: (map f xs)
```

- This implementation moves left-to-right across the list, mapping elements one at a time
- … But does it need to?

Implicit Parallelism In map

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of f to elements in list is commutative, we can reorder or parallelize execution
- *This is the "secret" that MapReduce exploits!*

BREAK

MapReduce

- Motivation: Large Scale Data Processing
  - Want to process lots of data (> 1 TB)
  - Want to parallelize across hundreds/thousands of CPUs
  - Want to make this easy…
- Features:
  - Automatic parallelization & distribution
  - Fault-tolerant
  - Provides status and monitoring tools
  - Clean abstraction for programmers
- Hadoop:
  - Open-source version of Google's MapReduce

Programming Model

- Borrows from functional programming
- Users implement interface of two functions:

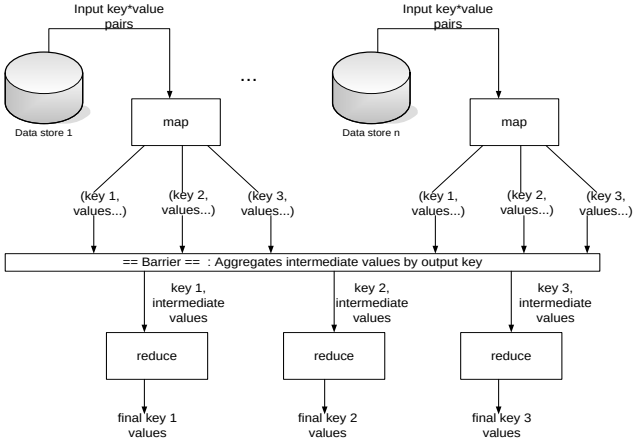
```
- map   (in_key, in_value) ->
     (out_key, intermediate_value) list

- reduce (out_key, intermediate_value list) ->
     out_value list
```


Functions

- Map:
  - Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs
    - » e.g., (filename, line)
  - map() produces one or more intermediate values along with an output key from the input
- Reduce:
  - After the map phase is over, all the intermediate values for a given output key are combined together into a list
  - reduce() combines those intermediate values into one or more *final values* for that same output key
    - » (in practice, usually only one final value per key)

MapReduce



Parallelism

- map() functions:
  - Run in parallel, creating different intermediate values from different input data sets
- reduce() functions:
  - Also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished

Example: Count word occurrences

```
map(String input_key, String input_value):
      // input_key: document name
      // input_value: document contents
   for each word w in input_value:
     EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
   // output_key: a word
   // output_values: a list of counts
   int result = 0;
   for each v in intermediate_values:
     result += ParseInt(v);
   Emit(AsString(result));
```


Example vs. Actual Source Code

- Example is written in pseudo-code
  - Actual Google implementation is a C++ library and Python/Java interfaces
  - Hadoop implementation is Java
- True code is somewhat more involved
  - Defines how input key/values are divided up, accessed, …
- Locality:
  - Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

MapReduce Techniques

- Locality
  - Master pgm places map() tasks based on data location
  - map() task inputs are divided into 64 MB blocks
- Fault-Tolerance
  - Master detects worker failures
    - » Re-executes completed & in-progress map() tasks
    - » Re-executes in-progress reduce() tasks
  - Master notices particular input key/values cause crashes in map(), and skips those values on re-exec
- Optimization – speculative execution
  - No reduce can start until map is complete:
    - » A single slow machine rate-limits the whole process
  - Master redundantly executes "slow " map tasks
    - » Uses results of first copy to finish – *why is this safe?*

Summary

- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- MapReduce has proven to be a useful abstraction
  - Greatly simplifies large-scale computations at Google, Yahoo!, and Facebook
  - Functional programming paradigm can be applied to large-scale applications
  - Fun to use: focus on problem, let library deal with messy details

CS162
Operating Systems and Systems Programming
Lecture 20

Distributed Systems, Networking

April 14, 2008

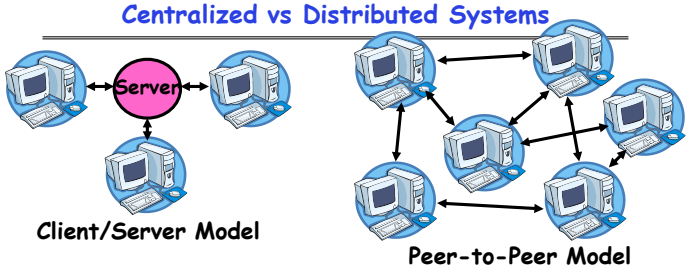
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Distributed Systems
- Networking
  - Broadcast
  - Point-to-Point Networking
  - Routing
  - Internet Protocol (IP)

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Centralized vs Distributed Systems



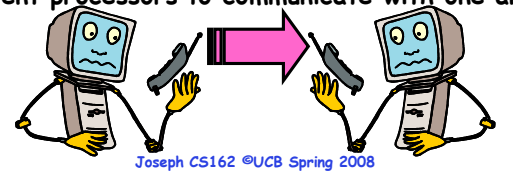
- **Centralized System:** System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System:** physically separate computers working together on some task
  - Early model: multiple servers working together
    - » Probably in the same room or building
    - » Often called a "cluster"
  - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues

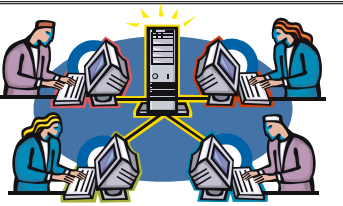
- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: Much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
  - Higher availability: one machine goes down, use another
  - Better durability: store data in multiple locations
  - More security: each piece easier to make secure
- Reality has been disappointing
  - Worse availability: depend on every machine being up
    - » Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

Distributed Systems: Goals/Requirements

- Transparency: the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - Location: Can't tell where resources are located
  - Migration: Resources may move without the user knowing
  - Replication: Can't tell how many copies of resource exist
  - Concurrency: Can't tell how many users there are
  - Parallelism: System may speed up large jobs by spliting them into smaller pieces
  - Fault Tolerance: System may hide varoius things that go wrong in the system
- Transparency and collaboration require some way for different processors to communicate with one another



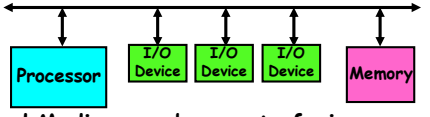
Networking Definitions



- Network: physical connection that allows two computers to communicate
- Packet: unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- Protocol: agreement between two parties as to how information is to be transmitted

Broadcast Networks

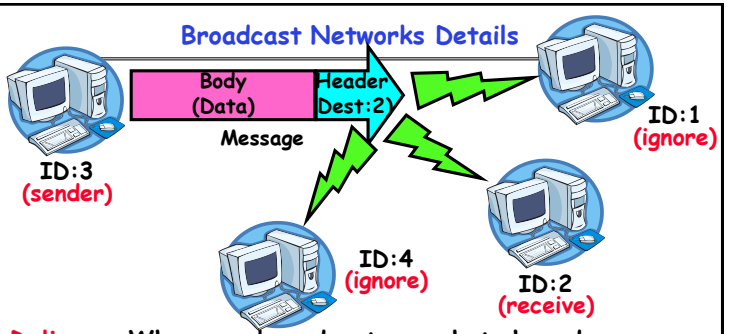
- Broadcast Network: Shared Communication Medium



  - Shared Medium can be a set of wires
    - » Inside a computer, this is called a bus
    - » All devices simultaneously connected to devices

  - Originally, Ethernet was a broadcast network
    - » All computers on local subnet connected to one another
  - More examples (wireless: medium is air): cellular phones, GSM GPRS, EDGE, CDMA 1xRTT, and 1EvDO

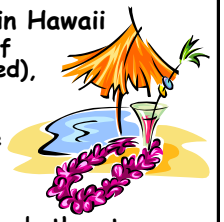
Broadcast Networks Details



- Delivery: When you broadcast a packet, how does a receiver know who it is for? (packet goes to everyone!)
  - Put header on front of packet: [Destination | Packet]
  - Everyone gets packet, discards if not the target
  - In Ethernet, this check is done in hardware
    - » No OS interrupt if not for particular destination
  - This is layering: we're going to build complex network protocols by layering on top of the packet

Broadcast Network Arbitration

- Arbitration: Act of negotiating use of shared medium
  - What if two senders try to broadcast at same time?
  - Concurrent activity but can't use shared memory to coordinate!
- Aloha network (70's): packet radio within Hawaii
  - Blind broadcast, with checksum at end of packet. If received correctly (not garbled), send back an acknowledgement. If not received correctly, discard.
    - » Need checksum anyway – in case airplane flies overhead
  - Sender waits for a while, and if doesn't get an acknowledgement, re-transmits.
  - If two senders try to send at same time, both get garbled, both simply re-send later.
  - Problem: Stability: what if load increases?
    - » More collisions ⇒ less gets through ⇒more resent ⇒ more load… ⇒ More collisions…
    - » Unfortunately: some sender may have started in clear, get scrambled without finishing



Carrier Sense, Multiple Access/Collision Detection

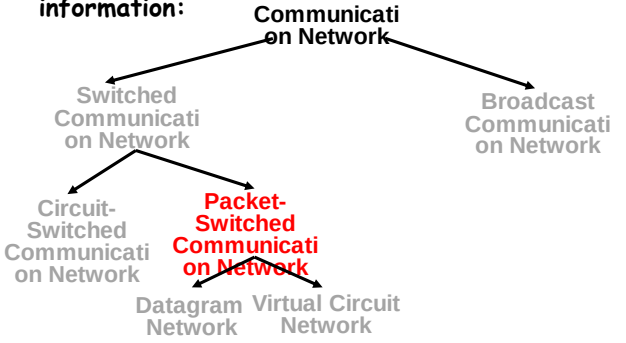
- Ethernet (early 80's): first practical local area network
  - It is the most common LAN for UNIX, PC, and Mac
  - Use wire instead of radio, but still broadcast medium
- Key advance was in arbitration called CSMA/CD: Carrier sense, multiple access/collision detection
  - Carrier Sense: don't send unless idle
    - » Don't mess up communications already in process
  - Collision Detect: sender checks if packet trampled.
    - » If so, abort, wait, and retry.
  - Backoff Scheme: Choose wait time before trying again
- How long to wait after trying to send and failing?
  - What if everyone waits the same length of time? Then, they all collide again at some time!
  - Must find way to break up shared behavior with nothing more than shared communication channel
- Adaptive randomized waiting strategy:
  - Adaptive and Random: First time, pick random wait time with some initial mean. If collide again, pick random value from bigger mean wait time. Etc.
  - Randomness is important to decouple colliding senders
  - Scheme figures out how many people are trying to send!

Administrivia

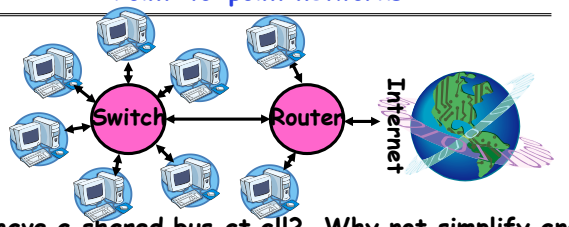
- Midterm #2 is Wednesday April 16th
  - 6-7:30pm in 10 Evans
  - All material from projects 1-3, lectures #9 (2/25) to #19 (4/9)
    - » OS History, Services, and Structure; CPU Scheduling; Kernel and Address Spaces; Address Translation, Caching and TLBs; Demand Paging; I/O Systems; Filesystems, Disk Management, Naming, and Directories; Distributed Systems
- Project #3 code deadline is Tuesday 4/22 at 11:59pm
- Final Exam
  - May 21st, 12:30-3:30pm

Taxonomy of Communication Networks

- Communication networks can be classified based on the way in which the nodes exchange information:



Point-to-point networks

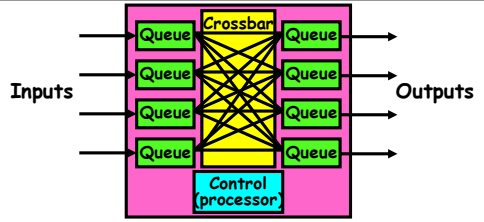


- Why have a shared bus at all? Why not simplify and only have point-to-point links + routers/switches?
  - Didn't used to be cost-effective
  - Now, easy to make high-speed switches and routers that can forward packets from a sender to a receiver.
- **Point-to-point network:** a network in which every physical wire is connected to only two computers
- **Switch:** a bridge that transforms a shared-bus configuration into a point-to-point network.
- **Router:** a device that acts as a junction between two networks to transfer data packets among them.

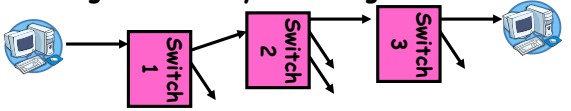
Point-to-Point Networks Discussion

- Advantages:
  - Higher link performance
    - » Can drive point-to-point link faster than broadcast link since less capacitance/less echoes (from impedance mismatches)
  - Greater aggregate bandwidth than broadcast link
    - » Can have multiple senders at once
  - Can add capacity incrementally
    - » Add more links/switches to get more capacity
  - Better fault tolerance (as in the Internet)
  - Lower Latency
    - » No arbitration to send, although need buffer in the switch
- Disadvantages:
  - More expensive than having everyone share broadcast link
  - However, technology costs now much cheaper
- Examples
  - ATM (asynchronous transfer mode)
    - » The first commercial point-to-point LAN
    - » Inspiration taken from telephone network
  - Switched Ethernet
    - » Same packet format and signaling as broadcast Ethernet, but only two machines on each ethernet.

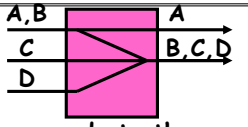
Point-to-Point Network design



- Switches look like computers: inputs, memory, outputs
  - In fact probably contains a processor
- Function of switch is to forward packet to output that gets it closer to destination
- Can build big crossbar by combining smaller switches



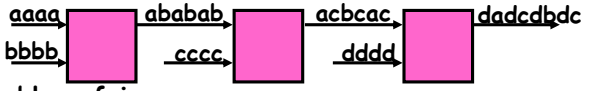
Flow control options



- What if everyone sends to the same output?
  - Congestion—packets don't flow at full rate
- In general, what if buffers fill up?
  - Need flow control policy
- Option 1: no flow control. Packets get dropped if they arrive and there's no space
  - If someone sends a lot, they are given buffers and packets from other senders are dropped
  - Internet actually works this way
- Option 2: Flow control between switches
  - When buffer fills, stop inflow of packets
  - Problem: what if path from source to destination is completely unused, but goes through some switch that has buffers filled up with unrelated traffic?

Flow Control (con't)

- Option 3: Per-flow flow control.
  - Allocate a separate set of buffers to each end-to-end stream and use separate "don't send me more" control on each end-to-end stream



- Problem: fairness
  - Throughput of each stream is entirely dependent on topology, and relationship to bottleneck
- Automobile Analogy
  - At traffic jam, one strategy is merge closest to the bottleneck
    - » Why people get off at one exit, drive 500 feet, merge back into flow
    - » Ends up slowing everybody else a huge amount
  - Also why have control lights at on-ramps
    - » Try to keep from injecting more cars than capacity of road (and thus avoid congestion)

BREAK

The Internet Protocol: "IP"

- The Internet is a large network of computers spread across the globe
  - According to the Internet Systems Consortium, there were over 353 million computers as of July 2005
  - In principle, every host can speak with every other one under the right circumstances
- IP Packet: a network packet on the internet
- IP Address: a 32-bit integer used as the destination of an IP packet
  - Often written as four dot-separated integers, with each integer from 0—255 (thus representing 8x4=32 bits)
  - Example CS file server is: 169.229.60.83 ≡ 0xA9E53C53
- Internet Host: a computer connected to the Internet
  - Host has one or more IP addresses used for routing
    - » Some of these may be private and unavailable for routing
  - Not every computer has a unique IP address
    - » Groups of machines may share a single IP address
    - » In this case, machines have private addresses behind a "Network Address Translation" (NAT) gateway

Address Subnets

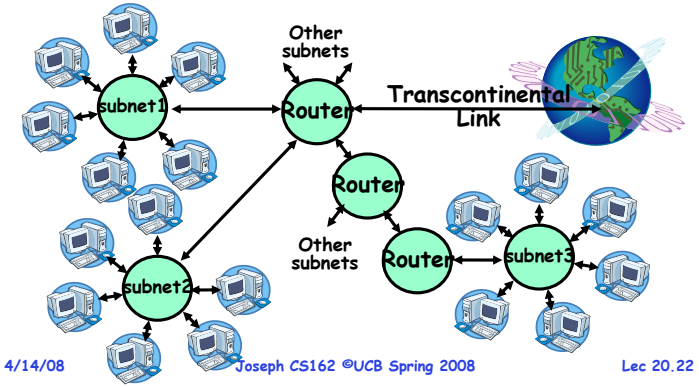
- Subnet: A network connecting a set of hosts with related destination addresses
- With IP, all the addresses in subnet are related by a prefix of bits
  - Mask: The number of matching prefix bits
    - » Expressed as a single value (e.g., 24) or a set of ones in a 32-bit value (e.g., 255.255.255.0)
- A subnet is identified by 32-bit value, with the bits which differ set to zero, followed by a slash and a mask
  - Example: 128.32.131.0/24 designates a subnet in which all the addresses look like 128.32.131.XX
  - Same subnet: 128.32.131.0/255.255.255.0
- Difference between subnet and complete network range
  - Subnet is always a subset of address range
  - Once, subnet meant single physical broadcast wire; now, less clear exactly what it means (virtualized by switches)

Address Ranges in IP

- IP address space divided into prefix-delimited ranges:
  - Class A: NN.0.0.0/8
    - » NN is 1–126 (126 of these networks)
    - » 16,777,214 IP addresses per network
    - » 10.xx.yy.zz is private
    - » 127.xx.yy.zz is loopback
  - Class B: NN.MM.0.0/16
    - » NN is 128–191, MM is 0-255 (16,384 of these networks)
    - » 65,534 IP addresses per network
    - » 172.[16-31].xx.yy are private
  - Class C: NN.MM.LL.0/24
    - » NN is 192–223, MM and LL 0-255 (2,097,151 of these networks)
    - » 254 IP addresses per networks
    - » 192.168.xx.yy are private
- Address ranges are often owned by organizations
  - Can be further divided into subnets

Hierarchical Networking: The Internet

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together
    - » Routing is often by prefix: e.g. first router matches first 8 bits of address, next router matches more, etc.



Simple Network Terminology

- Local-Area Network (LAN) – designed to cover small geographical area
  - Multi-access bus, ring, or star network
  - Speed ≈ 10 – 10,000 Megabits/second (100Gb/s soon!)
  - Broadcast is fast and cheap
  - In small organization, a LAN could consist of a single subnet. In large organizations (like UC Berkeley), a LAN contains many subnets
- Wide-Area Network (WAN) – links geographically separated sites
  - Point-to-point connections over long-haul lines (often leased from a phone company)
  - Speed ≈ 1.544 – 10,000 Megabits/second
  - Broadcast usually requires multiple messages

Routing

- Routing: the process of forwarding packets hop-by-hop through routers to reach their destination
  - Need more than just a destination address!
    - » Need a path
  - Post Office Analogy:
    - » Destination address on each letter is not sufficient to get it to the destination
    - » To get a letter from here to Florida, must route to local post office, sorted and sent on plane to somewhere in Florida, be routed to post office, sorted and sent with carrier who knows where street and house is…

- Internet routing mechanism: routing tables
  - Each router does table lookup to decide which link to use to get packet closer to destination
  - Don't need 4 billion entries in table: routing is by subnet
  - Could packets be sent in a loop? Yes, if tables incorrect
- Routing table contains:
  - Destination address range → output link closer to destination
  - Default entry (for subnets without explicit entries)

Setting up Routing Tables

- How do you set up routing tables?
  - Internet has no centralized state!
    - » No single machine knows entire topology
    - » Topology constantly changing (faults, reconfiguration, etc)
  - Need dynamic algorithm that acquires routing tables
    - » Ideally, have one entry per subnet or portion of address
    - » Could have "default" routes that send packets for unknown subnets to a different router that has more information
- Possible algorithm for acquiring routing table
  - Routing table has "cost" for each entry
    - » Includes number of hops to destination, congestion, etc.
    - » Entries for unknown subnets have infinite cost
  - Neighbors periodically exchange routing tables
    - » If neighbor knows cheaper route to a subnet, replace your entry with neighbors entry (+1 for hop to neighbor)
- In reality:
  - Internet has networks of many different scales
  - Different algorithms run at different scales

Conclusion

- Network: physical connection that allows two computers to communicate
  - Packet: sequence of bits carried over the network
- Broadcast Network: Shared Communication Medium
  - Transmitted packets sent to all receivers
  - Arbitration: act of negotiating use of shared medium
    - » Ethernet: Carrier Sense, Multiple Access, Collision Detect
- Point-to-point network: a network in which every physical wire is connected to only two computers
  - Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- Protocol: Agreement between two parties as to how information is to be transmitted
- Internet Protocol (IP): Layering used to abstract details
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports

CS162
Operating Systems and Systems Programming
Lecture 21

Networking

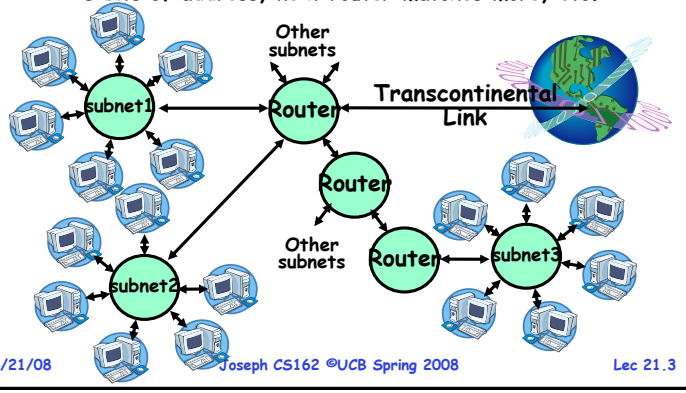
April 21, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: The Internet Protocol: "IP"

- The Internet is a large network of computers spread across the globe
  - According to the Internet Systems Consortium, there were over 353 million computers as of July 2005
  - In principle, every host can speak with every other one under the right circumstances
- IP Packet: a network packet on the internet
- IP Address: a 32-bit integer used as the destination of an IP packet
  - Often written as four dot-separated integers, with each integer from 0—255 (thus representing 8x4=32 bits)
  - Example CS file server is: 169.229.60.83 ≡ 0xA9E53C53
- Internet Host: a computer connected to the Internet
  - Host has one or more IP addresses used for routing
    - » Some of these may be private and unavailable for routing
  - Not every computer has a unique IP address
    - » Groups of machines may share a single IP address
    - » In this case, machines have private addresses behind a "Network Address Translation" (NAT) gateway

Review: Hierarchical Networking: The Internet

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together
    - » Routing is often by prefix: e.g. first router matches first 8 bits of address, next router matches more, etc.

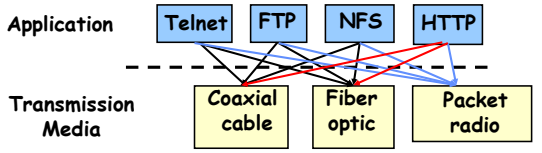


Goals for Today

- Networking
  - Broadcast
  - Point-to-Point Networking
  - Routing
  - Internet Protocol (IP)

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

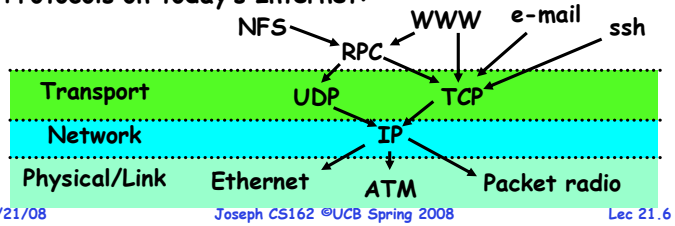
The Problem



- Re-implement every application for every technology?
- No! But how does the Internet architecture avoid this?

Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Example: system calls are the protocol between the operating system and application
  - Networking examples: many levels
    - » Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
    - » Link level: packet formats/error control (for instance, the CSMA/CD protocol)
    - » Network level: network routing, addressing
    - » Transport Level: reliable message delivery
- Protocols on today's Internet:

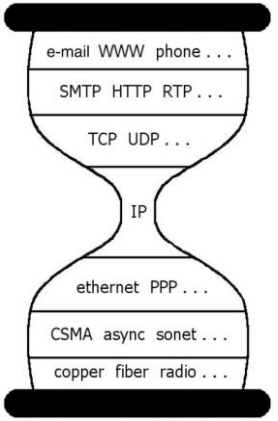


Network Layering

- Layering: building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

Physical Reality: Packets	Abstraction: Messages
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Hourglass – Single Internet Layer



- Allows networks to interoperate
  - Any network technology that supports IP can exchange packets
- Allows applications to function on all networks
  - Applications that can run on IP can use any network
- Simultaneous developments above and below IP

Back to Reality

- Layering is a convenient way to think about networks
- But layering is often violated
  - Firewalls
  - Transparent caches
  - NAT boxes
  - .....

Administrivia

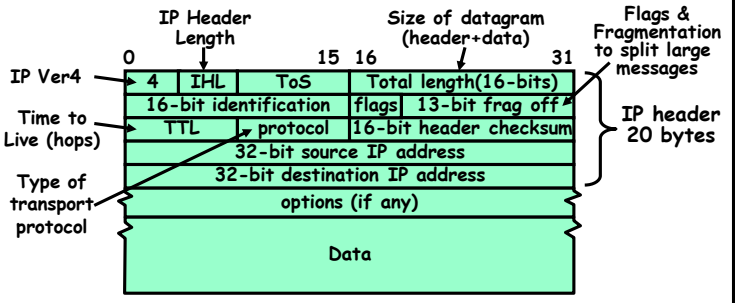
- Project #3 code deadline is tomorrow (Tue 4/22) at 11:59pm
- Final Exam
  - May 21st, 12:30-3:30pm

Building a messaging service

- Handling Arbitrary Sized Messages:
  - Must deal with limited physical packet size
  - Split big message into smaller ones (called fragments)
    - » Must be reassembled at destination
  - Checksum computed on each fragment or whole message
- Internet Protocol (IP): Must find way to send packets to arbitrary destination in network
  - Deliver messages unreliably ("best effort") from one machine in Internet to another
  - Since intermediate links may have limited size, must be able to fragment/reassemble packets on demand
  - Includes 256 different "sub-protocols" build on top of IP
    - » Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

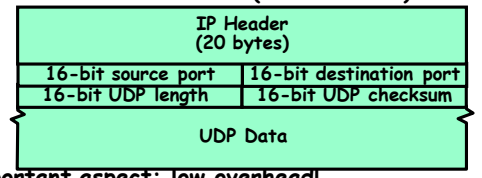
IP Packet Format

- IP Packet Format:



Building a messaging service

- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - » Add "ports", which are 16-bit identifiers
    - » A communication channel (connection) defined by 5 items: [source addr, source port, dest addr, dest port, protocol]
- UDP: The Unreliable Datagram Protocol
  - Layered on top of basic IP (IP Protocol 17)
    - » Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)



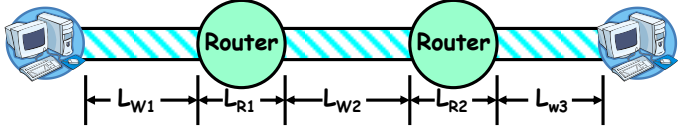
  - Important aspect: low overhead!
    - » Often used for high-bandwidth video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP

Building a messaging service (con't)

- UDP: The Unreliable Datagram Protocol
  - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)
  - Important aspect: low overhead!
    - » Often used for high-bandwidth video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP
- But we need ordered messages
  - Create ordered messages on top of unordered ones
    - » IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
  - How to fix this? Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection

Performance Considerations

- Before continue, need some performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
    - » Depends on "wire speed", but also limited by slowest router (routing delay) or by congestion at routers
  - Latency: time until first bit of packet arrives at receiver
    - » Raw transfer time + overhead at each routing hop



- Contributions to Latency
  - Wire latency: depends on speed of light on wire
    - » about 1–1.5 ns/foot
  - Router latency: depends on internals of router
    - » Could be < 1 ms (for a good router)
    - » Question: can router handle full wire throughput?

Sample Computations

- E.g.: Ethernet within Soda
  - Latency: speed of light in wire is 1.5ns/foot, which implies latency in building < 1 µs (if no routers in path)
  - Throughput: 10-1000Mb/s
  - Throughput delay: packet doesn't arrive until all bits
    - » So: 4KB/100Mb/s = 0.3 milliseconds (same order as disk!)
- E.g.: ATM within Soda
  - Latency (same as above, assuming no routing)
  - Throughput: 155Mb/s
  - Throughput delay: 4KB/155Mb/s = 200µ
- E.g.: ATM cross-country
  - Latency (assuming no routing):
    - » 3000miles * 5000ft/mile ⇒ 15 milliseconds
  - How many bits could be in transit at same time?
    - » 15ms * 155Mb/s = 290KB
  - In fact, Berkeley→MIT Latency ~ 45ms
    - » 872KB in flight if routers have wire-speed throughput
- Requirements for good performance:
  - Local area: minimize overhead/improve bandwidth
  - Wide area: keep pipeline full!

BREAK

Sequence Numbers

- Ordered Messages
  - Several network services are best constructed by ordered messaging
    - » Ask remote machine to first do x, then do y, etc.
  - Unfortunately, underlying network is packet based:
    - » Packets are routed one at a time through the network
    - » Can take different paths or be delayed individually
  - IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
- Solution requires queuing at destination
  - Need to hold onto packets to undo misordering
  - Total degree of reordering impacts queue size
- Ordered messages on top of unordered ones:
  - Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection
    - » Reordering among connections normally doesn't matter
  - If restart connection, need to make sure use different range of sequence numbers than previously…

Domain Name Service (DNS)

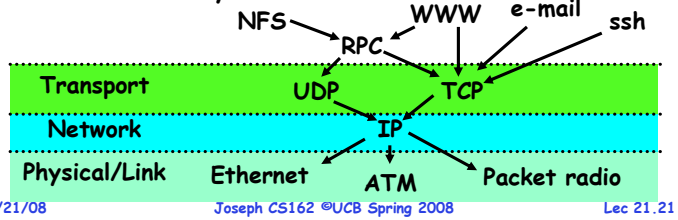
- Humans/applications use machine names
  - e.g., www.cs.berkeley.edu
- Network (IP) uses IP addresses
  - e.g., 67.114.112.23
- DNS translates between the two
  - An overlay service in its own right
  - Global distribution of name-to-IP address mappings—a kind of content distribution system as well
  - Unsung hero of the Internet

Domain Name Service (DNS)

- Humans/applications use machine names
  - e.g., www.cs.berkeley.edu
- Network (IP) uses IP addresses
  - e.g., 67.114.112.23
- DNS translates between the two
  - An overlay service in its own right
  - Global distribution of name-to-IP address mappings—a kind of content distribution system as well
  - Unsung hero of the Internet

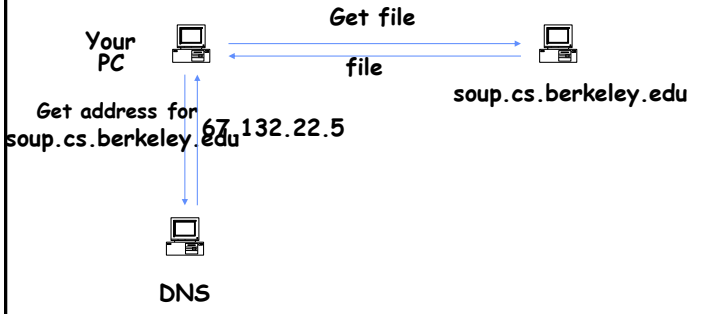
Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Example: system calls are the protocol between the operating system and application
  - Networking examples: many levels
    - » Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
    - » Link level: packet formats/error control (for instance, the CSMA/CD protocol)
    - » Network level: network routing, addressing
    - » Transport Level: reliable message delivery
- Protocols on today's Internet:



File Transfer (FTP, SCP, etc.)

Get file from soup.cs.berkeley.edu



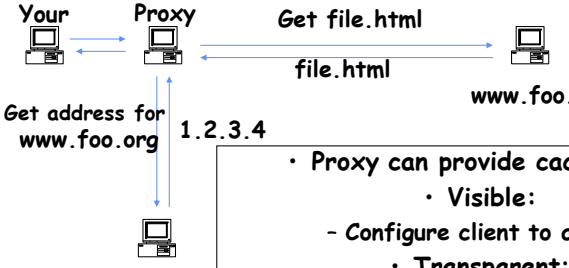
Email

Email message exchange is similar to previous example, except

- Exchange is between mail servers
- DNS gives name of mail server for domain
  - E.g., smtp.berkeley.edu

Web

Get www.foo.org/file.html



- Proxy can provide caching:
  - Visible:
    - Configure client to ask cache
  - Transparent:
    - Cache intercepts packet on its way to web server
    - An "application-aware middlebox"
    - Violates architectural purity, but are prevalent...

Content Distribution Network (CDN)

How to get closest copy of replicated content?

- CDNs have mirror servers distributed globally
- CDN customers allow CDN to run their DNS
- "Smart" DNS server returns results based on requester's IP address
- "Anycast IP address" routes to nearest server
  - Used for DNS top-level servers

Conclusion

- Internet Protocol (IP): Layering used to abstract details
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports
- Layering: building complex services from simpler ones
- Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
  - Latency: time until first bit of packet arrives at receiver
- Arbitrary Sized messages:
  - Fragment into multiple packets; reassemble at destination
- Ordered messages:
  - Use sequence numbers and reorder at destination

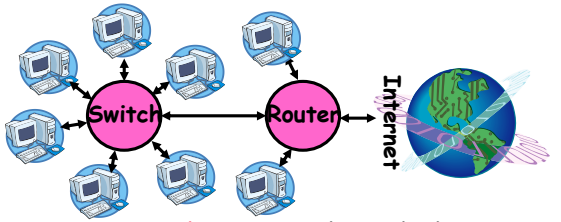
CS162
Operating Systems and Systems Programming
Lecture 22

Networking II

April 23, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Point-to-point networks



- Point-to-point network: a network in which every physical wire is connected to only two computers
- Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- Hub: a multiport device that acts like a repeater broadcasting from each input to every output
- Router: a device that acts as a junction between two networks to transfer data packets among them.

Sequence Numbers

- Ordered Messages
  - Several network services are best constructed by ordered messaging
    - » Ask remote machine to first do x, then do y, etc.
  - Unfortunately, underlying network is packet based:
    - » Packets are routed one at a time through the network
    - » Can take different paths or be delayed individually
  - IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
- Solution requires queuing at destination
  - Need to hold onto packets to undo misordering
  - Total degree of reordering impacts queue size
- Ordered messages on top of unordered ones:
  - Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection
    - » Reordering among connections normally doesn't matter
  - If restart connection, need to make sure use different range of sequence numbers than previously…

Goals for Today

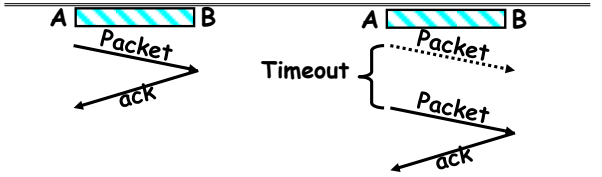
- Networking
  - Protocols
  - Reliable Messaging
    - » TCP windowing and congestion avoidance
  - Two-phase commit

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery
  - Reliable messages on top of unreliable packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received only once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

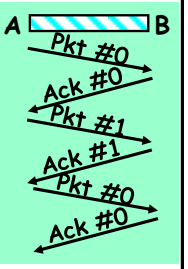
Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ack") when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn't get an ack, does that mean the receiver didn't get the original message?
    - » No
  - What it ack gets dropped? Or if message gets delayed?
    - » Sender doesn't get ack, retransmits. Receiver gets message twice, acks each.

How to deal with message duplication

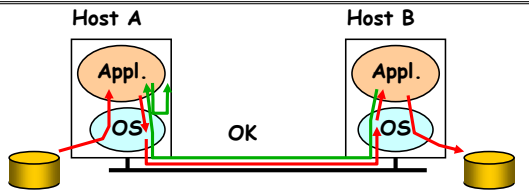
- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate #'s; Discard if detected
- Requirements:
  - Sender keeps copy of unack'ed messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don't send next message until ack received
  - Sender keeps last message; receiver tracks sequence # of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily



"End-to-End Arguments in System Design" (Saltzer, Reed, and Clark)

- Most influential paper about placing functionality
  - "Sacred Text" of the Internet
    - » Endless disputes about what it means
    - » Everyone cites it as supporting their position
- Some applications have end-to-end performance requirements:
  - Reliability, security, …
- Implementing these in the network is very hard:
  - Every step along the way must be fail-proof
- Hosts:
  - Can satisfy the requirement without the network
  - Can't depend on the network

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
  - Solution 1 not complete (e.g., misbehaving net element)
    - » What happens if any network element misbehaves?
    - » Receiver has to do the check anyway!
- Solution 2: end-to-end check and retry [complete!]
  - Solution 2 is complete
    - » Full functionality can be entirely implemented at application layer with no need for reliability from lower layers

Summary

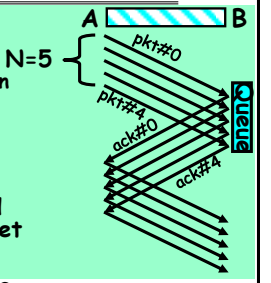
- Implementing this functionality in the network:
  - Doesn't reduce host implementation complexity
  - Does increase network complexity
  - Probably imposes delay and overhead on all applications, even if they don't need functionality
- However, implementing in network can enhance performance in some cases
  - Such as a very lossy link
- Layering is a good way to organize networks
  - Unified Internet layer decouples apps from networks
  - E2E argument encourages us to keep IP simple
  - Commercial realities may undo all of this...

Administrivia

- Project #4 design deadline is Thu 5/1 at 11:59pm
  - Code deadline is Wed 5/14
- Final Exam
  - May 21st, 12:30-3:30pm
- Final Topics: Any suggestions?
  - Please send them to me…

Better messaging: Window-based acknowledgements

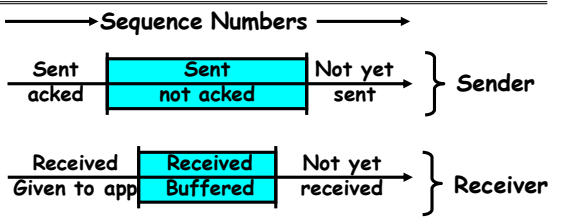
- Window based protocol (TCP):
  - Send up to N packets without ack
    - » Allows pipelining of packets
    - » Window size (N) < queue at destination
  - Each packet has sequence number
    - » Receiver acknowledges each packet
    - » Ack says "received all packets up to sequence number X"/send more
- Acks serve dual purpose:
  - Reliability: Confirming packet received
  - Flow Control: Receiver ready for packet
    - » Remaining space in queue at receiver can be returned with ACK
- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ack packet
    - » Resend missing packets⇒ Receiver gets packets out of order!
  - Should receiver discard packets that arrive out of order?
    - » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    - » Reduces # of retransmits, but more complex
- What if ack gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets



Transmission Control Protocol (TCP)

- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - » "Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space
    - » Also, window should reflect speed/capacity of network – sender shouldn't overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - » A "good citizen"

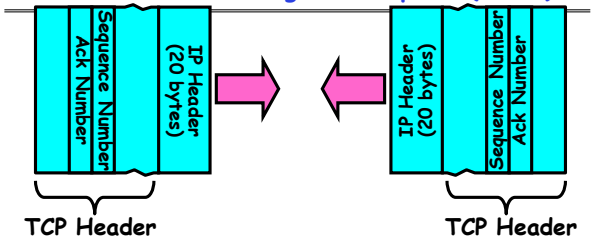
TCP Windows and Sequence Numbers



- Sender has three regions:
  - Sequence regions
    - » sent and ack'ed
    - » Sent and not ack'ed
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ack'ed (given to application)
    - » received and buffered
    - » not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)

Selective Acknowledgement Option (SACK)



- Vanilla TCP Acknowledgement
  - Every message encodes Sequence number and Ack
  - Can include data for forward stream and/or ack for reverse stream
- Selective Acknowledgement
  - Acknowledgement information includes not just one number, but rather ranges of received packets
  - Must be specially negotiated at beginning of TCP setup
    - » Not widely in use (although in Windows since Windows 98)

Congestion Avoidance

- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - » Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - » Goal: fill network between sender and receiver
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ack received
  - Timeout ⇒ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

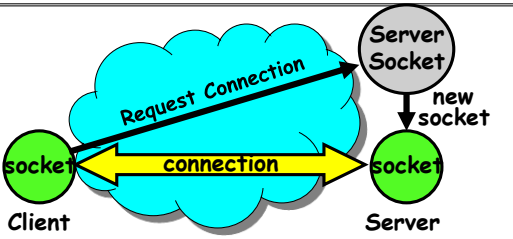
Sequence-Number Initialization

- How do you choose an initial sequence number?
  - When machine boots, ok to start with sequence #0?
    - » No: could send two messages with same sequence #!
    - » Receiver might end up discarding valid packets, or duplicate ack from original transmission might hide lost packet
  - Also, if it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
  - Time to live: each packet has a deadline.
    - » If not delivered in X seconds, then is dropped
    - » Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
  - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
    - » Epoch # stored on disk, Put in every message
    - » Epoch # incremented on crash and/or when run out of sequence #
  - Pseudo-random increment to previous sequence number
    - » Used by several protocol implementations

Use of TCP: Sockets

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Setup (Con't)



- Things to remember:
  - Connection requires 5 values:
    [Src Addr, Src Port, Dst Addr, Dst Port, Protocol]
  - Often, Src Port "randomly" assigned
    - » Done by OS during client socket setup
  - Dst Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

BREAK

Socket Example (Java)

```
server:
      //Makes socket, binds addr/port, calls listen()
      ServerSocket sock = new ServerSocket(6013);
      while(true) {
        Socket client = sock.accept();
        PrintWriter pout = new
          PrintWriter(client.getOutputStream(),true);

        pout.println("Here is data sent to client!");
          …
        client.close();
      }

client:
      // Makes socket, binds addr/port, calls connect()
      Socket sock = new Socket("169.229.60.38",6013);
      BufferedReader bin =
        new BufferedReader(
          new InputStreamReader(sock.getInputStream));
      String line;
      while ((line = bin.readLine())!=null)
        System.out.println(line);
      sock.close();
```


Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set

  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - » Wait until `mbox` has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that the receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1,mbox);
   }

Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer,mbox);
      process message;
   }
```

Send Message

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Server: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

Get Response

Receive Request

Send Response

Conclusion

- Layering: building complex services from simpler ones
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput
- TCP: Reliable byte stream between two processes on different machines over Internet (read, write, flush)

CS162
Operating Systems and Systems Programming
Lecture 23

Network Communication Abstractions / Remote Procedure Call

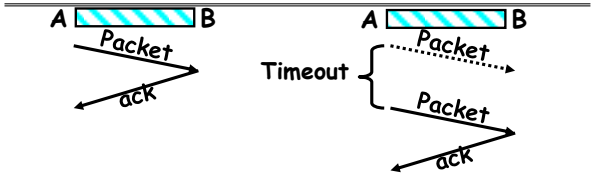
April 28, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Reliable Networking

- Layering: building complex services from simpler ones
- Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
  - Latency: time until first bit of packet arrives at receiver
- Arbitrary Sized messages:
  - Fragment into multiple packets; reassemble at destination
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput

Review: Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ack") when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn't get an ack, does that mean the receiver didn't get the original message?
    - » No
  - What it ack gets dropped? Or if message gets delayed?
    - » Sender doesn't get ack, retransmits. Receiver gets message twice, acks each.

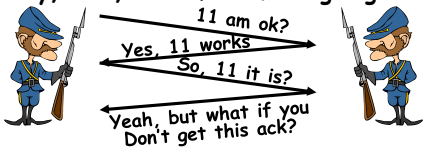
Goals for Today

- Distributed Decision Making
  - Two-phase commit/Byzantine Commit
- Remote Procedure Call
- Examples of Distributed File Systems

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

General's Paradox

- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    - » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - » The global coordinator requests that all participants will promise to commit or rollback the transaction
    - » Participants record promise in log, then acknowledge
    - » If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
  - Commit Phase:
    - » After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
    - » Then asks all nodes to commit; they respond with ack
    - » After receive acks, coordinator writes "Got Commit" to log
  - Log can be used to complete this process such that all machines either commit or don't commit

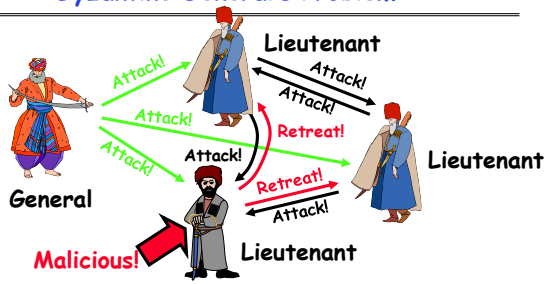
Two phase commit example

- Simple Example: A≡WellsFargo Bank, B≡Bank of America
  - Phase 1: Prepare Phase
    - » A writes "Begin transaction" to log
      A→B: OK to transfer funds to me?
    - » Not enough funds:
      B→A: transaction aborted; A writes "Abort" to log
    - » Enough funds:
      B: Write new account balance & promise to commit to log
      B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - » A: write new account balance to log
    - » Write "Commit" to log
    - » Send message to B that commit occurred; wait for ack
    - » Write "Got Commit" to log
- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout, abort and retry
- What if A crashes at beginning of phase 2?
  - Wakes up, sees that there is a transaction in progress; sends "Abort" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A sends it "Commit" message, it will say, "oh, ok, commit"

Distributed Decision Making Discussion

- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - » Simple failure mode called "failstop" (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - » Site B writes "prepared to commit" record to its log, sends a "yes" vote to the coordinator (site A) and crashes
    - » Site A crashes
    - » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. If sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    - » B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

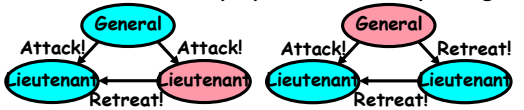
Byzantine General's Problem



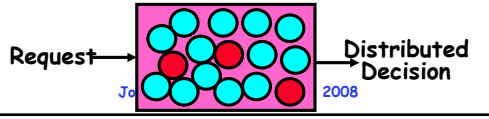
- Byzantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

- Impossibility Results:
  - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things



  - With f faults, need n > 3f to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity $O(n^2)$
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious



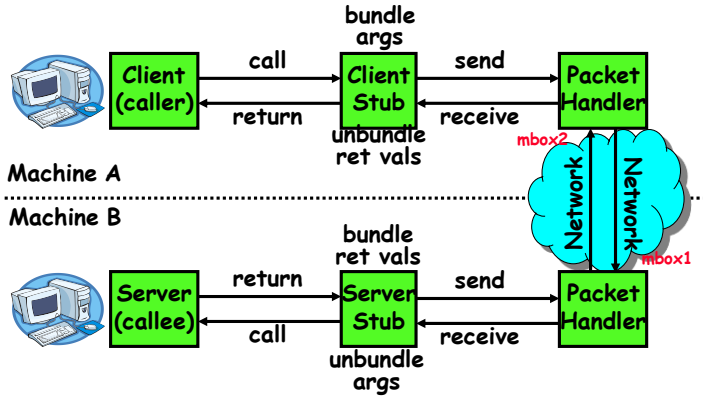
Administrivia

- Project #4 design deadline is Thu 5/1 at 11:59pm
  - Code deadline is Wed 5/14
- Final Exam
  - May 21st, 12:30-3:30pm
- Final Topics: Any suggestions?
  - Please send them to me…

Remote Procedure Call

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("rutabaga");`
  - Translated automatically into call on server:
    `fileSys→Read("rutabaga");`
- Implementation:
  - Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
    - » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    - » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynmaic translation of service→mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

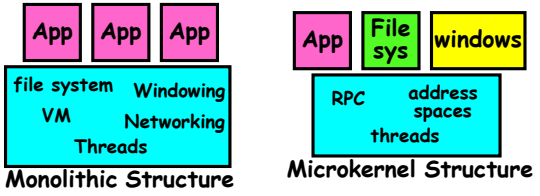
- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - » User-level bug causes address space to crash
    - » Machine failure, kernel bug causes all processes on same machine to fail
    - » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - » Did my cached data get written back or not?
    - » Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - » Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - "Remote" procedure call (2-way communication)
- RPC's can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine



- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

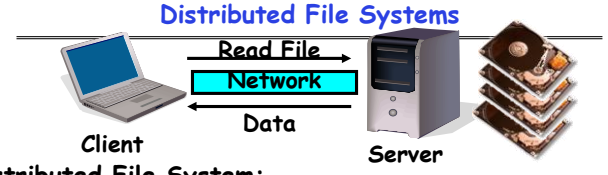
Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine

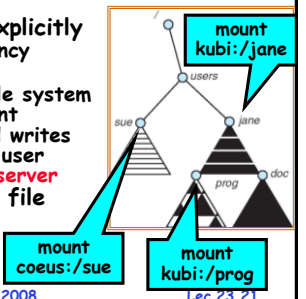
- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

BREAK

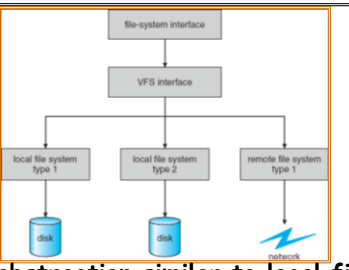
Distributed File Systems



- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - *Hostname*:localname: Name files explicitly
    - » No location or migration transparency
  - *Mounting* of remote file systems
    - » System manager mounts remote file system by giving name and local mount point
    - » Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - *A single, global name space:* every file in the world has unique name
    - » Location Transparency: servers can change and files can move without involving user

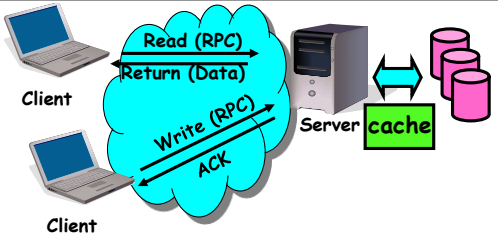


Virtual File System (VFS)



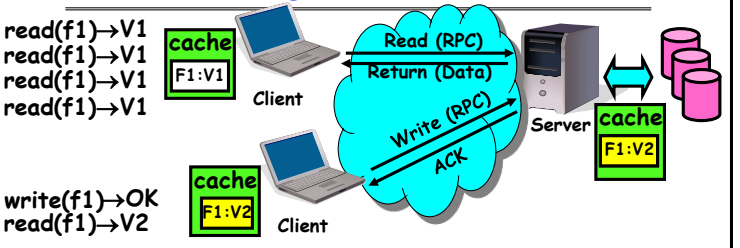
- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Simple Distributed File System



- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Use of caching to reduce network load



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Failures

- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "rm foo", but before acknowledgment?
    - » Message system will retry: send it again
    - » How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- **Stateless protocol:** A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

World Wide Web

- Key idea: graphical front-end to RPC protocol
- What happens when a web server fails?
  - System breaks!
  - Solution: Transport or network-layer redirection
    - » Invisible to applications
    - » Can also help with scalability (load balancers)
    - » Must handle "sessions" (e.g., banking/e-commerce)
- Initial version: no caching
  - Didn't scale well – easy to overload servers

WWW Caching

- Use client-side caching to reduce number of interactions between clients and servers and/or reduce the size of the interactions:
  - Time-to-Live (TTL) fields – HTTP "Expires" header from server
  - Client polling – HTTP "If-Modified-Since" request headers from clients
  - Server refresh – HTML "META Refresh tag" causes periodic client poll
- What is the polling frequency for clients and servers?
  - Could be adaptive based upon a page's age and its rate of change
- Server load is still significant!

WWW Proxy Caches

- Place caches in the network to reduce server load
  - But, increases latency in lightly loaded case
  - Caches near servers called "reverse proxy caches"
    - » Offloads busy server machines
  - Caches at the "edges" of the network called "content distribution networks"
    - » Offloads servers and reduce client latency
- Challenges:
  - Caching static traffic easy, but only ~40% of traffic
  - Dynamic and multimedia is harder
    - » Multimedia is a big win: Megabytes versus Kilobytes
  - Same cache consistency problems as before
- Caching is changing the Internet architecture
  - Places functionality at higher levels of comm. protocols

Conclusion

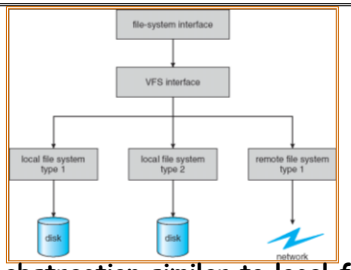
- **Two-phase commit:** distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- **Byzantine General's Problem:** distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$
- **Remote Procedure Call (RPC):** Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
- **VFS:** Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems

CS162
Operating Systems and Systems Programming
Lecture 24

Distributed File Systems

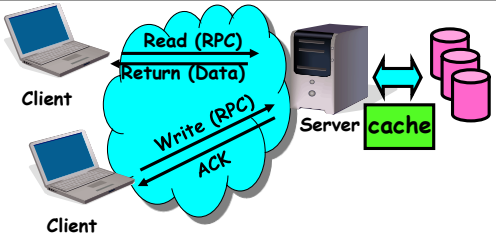
April 30, 2008
Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Review: Virtual File System (VFS)



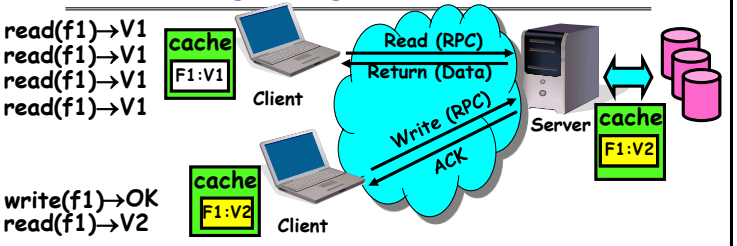
- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Review: Simple Distributed File System



- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Review: Using caching to reduce network load



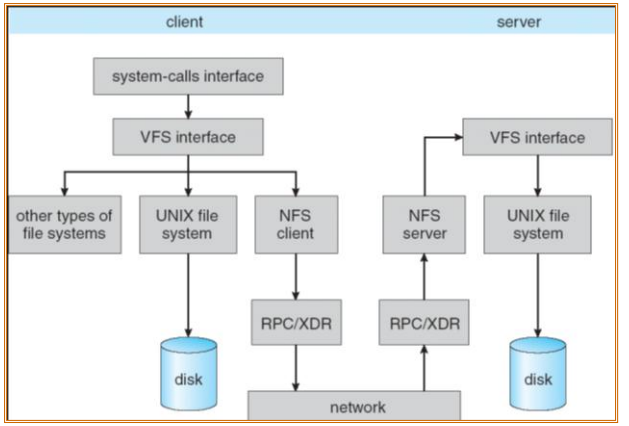
- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Goals for Today

- Examples of Distributed File Systems
- Cache Coherence Protocols
- Worms and Viruses

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Schematic View of NFS Architecture



Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

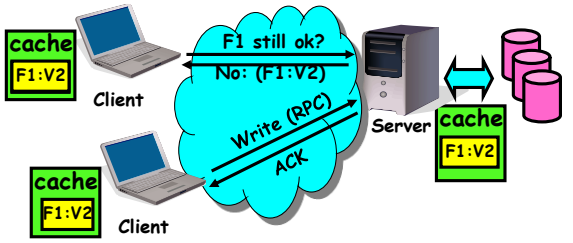
- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

Administrivia

- Project #4 design deadline is Thu 5/1 at 11:59pm
  - Code deadline is Wed 5/14
- Final Exam
  - May 21st, 12:30-3:30pm
- Final Topics: Any suggestions?
  - Please send them to me…

NFS Cache consistency

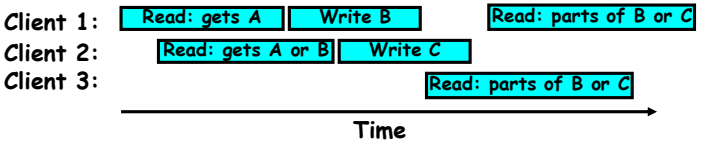
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large # clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

Summary

- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Distributed File System:
  - Transparent access to files stored on a remote disk
    - » NFS: Network File System
    - » AFS: Andrew File System
  - Caching for performance
- Cache Consistency: Keeping contents of client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks so can be notified by server of changes

Internet Worms

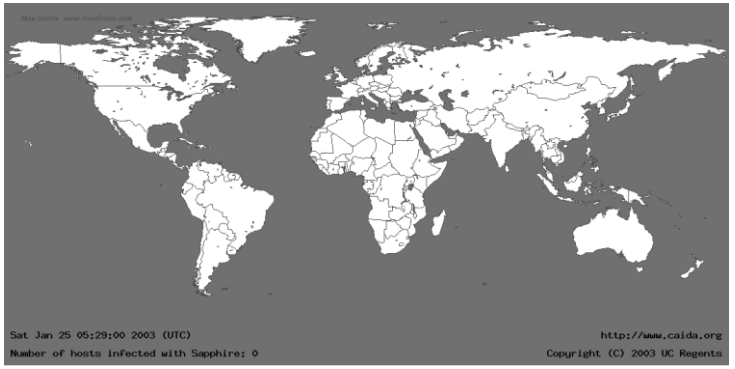
- Self-replicating, self-propagating code and data
- Use network to find potential victims
- Typically exploit vulnerabilities in an application running on a machine or the machine's operating system to gain a foothold
- Then search the network for new victims

Sapphire (AKA Slammer) Worm

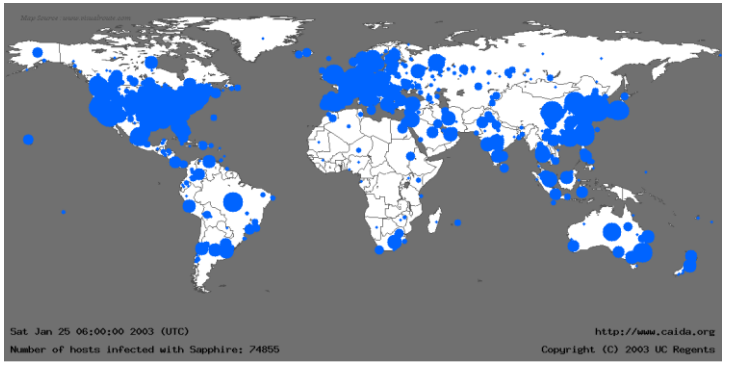
- January 25, 2003
- Fastest computer worm in history
  - Used MS SQL Server buffer overflow vulnerability
  - Doubled in size every 8.5 seconds, 55M scans/sec
  - Infected >90% of vulnerable hosts within 10 mins
  - Infected at least 75,000 hosts
  - Caused network outages, canceled airline flights, elections problems, interrupted E911 service, and caused ATM failures

BREAK

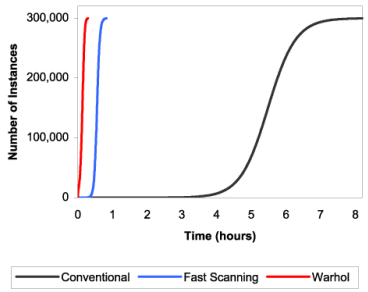
Before Sapphire



After Sapphire



Worm Propagation Behavior



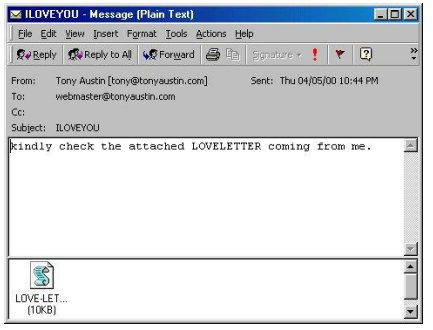
- More efficient scanning finds victims faster (< 1hr)
- Even faster propagation is possible if you cheat
  - Wasted effort scanning non-existent or non-vulnerable hosts
  - Warhol: seed worm with a "hit list" of vulnerable hosts (15 mins)

Internet Viruses

- Self-replicating code and data
- Typically requires human interaction before exploiting an application vulnerability
  - Running an e-mail attachment
  - Clicking on a link in an e-mail
  - Inserting/connecting "infected" media to a PC
- Then search for files to infect or sends out e-mail with an infected file

LoveLetter Virus (May 2000)

- E-mail message with VBScript (simplified Visual Basic)
- Relies on Windows Scripting Host
  - Enabled by default in Win98/2000
- User clicks on attachment ➔ infected!



LoveLetter's Impact

- Approx 60 – 80% of US companies infected by the "ILOVEYOU" virus
- Several US gov. agencies and the Senate were hit
- > 100,000 servers in Europe
- Substantial lost data from replacement of files with virus code
  - Backups anyone?
- Could have been worse – not all viruses require opening of attachments…

Worm/Virus Summary

- Worms are a critical threat
  - More than 100 companies, including Financial Times, ABCNews and CNN, were hit by the Zotob Windows 2000 worm in August 2005
- Viruses are a critical threat
  - FBI survey of 269 companies in 2004 found that viruses caused ~$55 million in damages
  - DIY toolkits proliferate on Internet
- How can we protect against worms and viruses?
  - Scanners
  - Firewalls
  - …

CS162
Operating Systems and Systems Programming
Lecture 25

Protection and Security in Distributed Systems

May 5, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

Goals for Today

- Security Mechanisms
  - Authentication
  - Authorization
  - Enforcement
- Cryptographic Mechanisms

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Protection vs Security

- Protection: one or more mechanisms for controlling the access of programs, processes, or users to resources
  - Page Table Mechanism
  - File Access Mechanism
- Security: use of protection mechanisms to prevent misuse of resources
  - Misuse defined with respect to policy
    - » E.g.: prevent exposure of certain sensitive information
    - » E.g.: prevent unauthorized modification/deletion of data
  - Requires consideration of the external environment within which the system operates
    - » Most well-constructed system cannot protect information if user accidentally reveals password
- What we hope to gain today and next time
  - Conceptual understanding of how to make systems secure
  - Some examples, to illustrate why providing security is really hard in practice

Preventing Misuse

- Types of Misuse:
  - Accidental:
    - » If I delete shell, can't log in to fix it!
    - » Could make it more difficult by asking: "do you really want to delete the shell?"
  - Intentional:
    - » Some high school brat who can't get a date, so instead he transfers $3 billion from B to A.
    - » Doesn't help to ask if they want to do it (of course!)
- Three Pieces to Security
  - Authentication: who the user actually is
  - Authorization: who is allowed to do what
  - Enforcement: make sure people do only what they are supposed to do
- Loopholes in any carefully constructed system:
  - Log in as superuser and you've circumvented authentication
  - Log in as self and can do anything with your resources; for instance: run program that erases all of your files
  - Can you trust software to correctly enforce Authentication and Authorization?????

Authentication: Identifying Users

- How to identify users to the system?
  - Passwords
    - » Shared secret between two parties
    - » Since only user knows password, someone types correct password ⇒ must be user typing it
    - » Very common technique
  - Smart Cards
    - » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries
    - » May have display to allow reading of password
    - » Or can be plugged in directly; several credit cards now in this category

  - Biometrics
    - » Use of one or more intrinsic physical or behavioral traits to identify someone
    - » Examples: fingerprint reader, palm reader, retinal scan
    - » Becoming quite a bit more common

Passwords: Secrecy

- System must keep copy of secret to check against passwords
  - What if malicious user gains access to list of passwords?
    - » Need to obscure information somehow
  - Mechanism: utilize a transformation that is difficult to reverse without the right key (e.g. encryption)
- Example: UNIX /etc/passwd file
  - passwd→one way transform(hash)→encrypted passwd
  - System stores only encrypted version, so OK even if someone reads the file!
  - When you type in your password, system compares encrypted version
- Problem: Can you trust encryption algorithm?
  - Example: one algorithm thought safe had back door
    - » Governments want back door so they can snoop
  - Also, security through obscurity doesn't work
    - » GSM encryption algorithm was secret; accidentally released; Berkeley grad students cracked in a few hours

Passwords: How easy to guess?

- Ways of Compromising Passwords
  - Password Guessing:
    - » Often people use obvious information like birthday, favorite color, girlfriend's name, etc…
  - Dictionary Attack:
    - » Work way through dictionary and compare encrypted version of dictionary words with entries in /etc/passwd
  - Dumpster Diving:
    - » Find pieces of paper with passwords written on them
    - » (Also used to get social-security numbers, etc)
- Paradox:
  - Short passwords are easy to crack
  - Long ones, people write down!
- Technology means we have to use longer passwords
  - UNIX initially required lowercase, 5-letter passwords: total of 26^5=10million passwords
    - » In 1975, 10ms to check a password→1 day to crack
    - » In 2005, .01μs to check a password→0.1 seconds to crack
  - Takes less time to check for all words in the dictionary!

Passwords: Making harder to crack

- How can we make passwords harder to crack?
  - Can't make it impossible, but can help
- Technique 1: Extend everyone's password with a unique number (stored in password file)
  - Called "salt". UNIX uses 12-bit "salt", making dictionary attacks 4096 times harder
  - Without salt, would be possible to pre-compute all the words in the dictionary hashed with the UNIX algorithm: would make comparing with /etc/passwd easy!
  - Also, way that salt is combined with password designed to frustrate use of off-the-shelf DES hardware
- Technique 2: Require more complex passwords
  - Make people use at least 8-character passwords with upper-case, lower-case, and numbers
    - » 70^8=6x10^{14}=6million seconds=69 days@0.01μs/check
  - Unfortunately, people still pick common patterns
    - » e.g. Capitalize first letter of common word, add one digit

Passwords: Making harder to crack (con't)

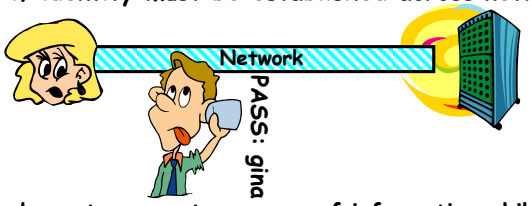
- Technique 3: Delay checking of passwords
  - If attacker doesn't have access to /etc/passwd, delay every remote login attempt by 1 second
  - Makes it infeasible for rapid-fire dictionary attack
- Technique 4: Assign very long passwords
  - Long passwords or pass-phrases can have more entropy (randomness→harder to crack)
  - Give everyone a smart card (or ATM card) to carry around to remember password
    - » Requires physical theft to steal password
    - » Can require PIN from user before authenticates self
  - Better: have smartcard generate pseudorandom number
    - » Client and server share initial seed
    - » Each second/login attempt advances to next random number
- Technique 5: "Zero-Knowledge Proof"
  - Require a series of challenge-response questions
    - » Distribute secret algorithm to user
    - » Server presents a number, say "5"; user computes something from the number and returns answer to server
    - » Server never asks same "question" twice
  - Often performed by smartcard plugged into system

Administrivia

- Project #4 code deadline is Wed 5/14 at 11:59pm
- Final Exam
  - May 21st, 12:30-3:30pm
- Final Topics: Any suggestions?
  - Please send them to me…

Authentication in Distributed Systems

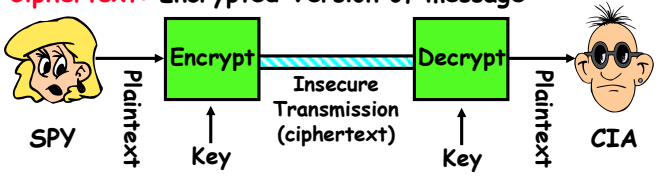
- What if identity must be established across network?



  - Need way to prevent exposure of information while still proving identity to remote system
  - Many of the original UNIX tools sent passwords over the wire "in clear text"
    - » E.g.: telnet, ftp, yp (yellow pages, for distributed login)
    - » Result: Snooping programs widespread
- What do we need? Cannot rely on physical security!
  - Encryption: Privacy, restrict receivers
  - Authentication: Remote Authenticity, restrict senders

Private Key Cryptography

- Private Key (Symmetric) Encryption:
  - Single key used for both encryption and decryption
- Plaintext: Unencrypted Version of message
- Ciphertext: Encrypted Version of message

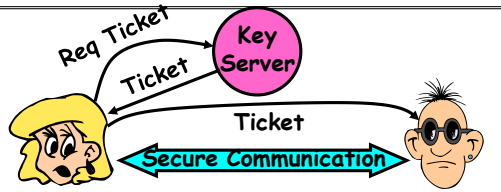


- Important properties
  - Can't derive plain text from ciphertext (decode) without access to key
  - Can't derive key from plain text and ciphertext
  - As long as password stays secret, get both secrecy and authentication
- Symmetric Key Algorithms: DES, Triple-DES, AES

Key Distribution

- How do you get shared secret to both places?
  - For instance: how do you send authenticated, secret mail to someone who you have never met?
  - Must negotiate key over private channel
    - » Exchange code book
    - » Key cards/memory stick/others
- Third Party: Authentication Server (like Kerberos)
  - Notation:
    - » K_{xy} is key for talking between x and y
    - » $(\ldots)^K$ means encrypt message (…) with the key K
    - » Clients: A and B, Authentication server S
  - A asks server for key:
    - » A→S: [Hi! I'd like a key for talking between A and B]
    - » Not encrypted. Others can find out if A and B are talking
  - Server returns *session* key encrypted using B's key
    - » S→A: Message [Use K_{ab} (This is A! Use K_{ab})^{Ksb}] ^{Ksa}
    - » This allows A to know, "S said use this key"
  - Whenever A wants to talk with B
    - » A→B: Ticket [This is A! Use K_{ab}]^{Ksb}
    - » Now, B knows that K_{ab} is sanctioned by S

Authentication Server Continued



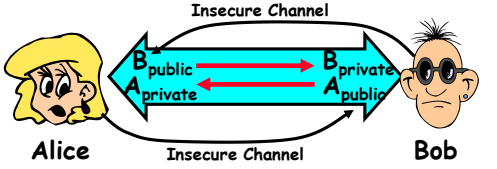
- Details
  - Both A and B use passwords (shared with key server) to decrypt return from key servers
  - Add in timestamps to limit how long tickets will be used to prevent attacker from replaying messages later
  - Also have to include encrypted checksums (hashed version of message) to prevent malicious user from inserting things into messages/changing messages
  - Want to minimize # times A types in password
    - » A→S (Give me temporary secret)
    - » S→A (Use $K_{temp\text{-}sa}$ for next 8 hours)^{Ksa}
    - » Can now use $K_{temp\text{-}sa}$ in place of K_{sa} in prototcol

Public Key Encryption

- Can we perform key distribution without an authentication server?
  - Yes. Use a Public-Key Cryptosystem.
- Public Key Details
  - Don't have one key, have two: K_{public}, $K_{private}$
    - » Two keys are mathematically related to one another
    - » Really hard to derive K_{public} from $K_{private}$ and vice versa
  - Forward encryption:
    - » Encrypt: (cleartext)^{Kpublic}= $ciphertext_1$
    - » Decrypt: ($ciphertext_1$)^{Kprivate} = cleartext
  - Reverse encryption:
    - » Encrypt: (cleartext)^{Kprivate} = $ciphertext_2$
    - » Decrypt: ($ciphertext_2$)^{Kpublic} = cleartext
  - Note that $ciphertext_1 \neq ciphertext_2$
    - » Can't derive one from the other!
- Public Key Examples:
  - RSA: Rivest, Shamir, and Adleman
    - » K_{public} of form (k_{public}, N), $K_{private}$ of form ($k_{private}$, N)
    - » N = pq. Can break code if know p and q
  - ECC: Elliptic Curve Cryptography

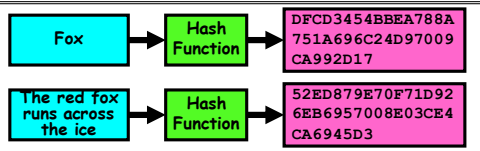
Public Key Encryption Details

- Idea: K_{public} can be made public, keep $K_{private}$ private



- Gives message privacy (restricted receiver):
  - Public keys (secure destination points) can be acquired by anyone/used by anyone
  - Only person with private key can decrypt message
- What about authentication?
  - Use combination of private and public key
  - Alice→Bob: [(I'm Alice)^{Aprivate} Rest of message]^{Bpublic}
  - Provides restricted sender and receiver
- But: how does Alice know that it was Bob who sent her B_{public}? And vice versa…

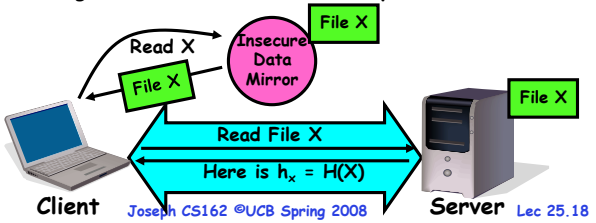
Secure Hash Function



- Hash Function: Short summary of data (message)
  - For instance, $h_1=H(M_1)$ is the hash of message M_1
    - » h_1 fixed length, despite size of message M_1.
    - » Often, h_1 is called the "digest" of M_1.
- Hash function H is considered secure if
  - It is infeasible to find M_2 with $h_1=H(M_2)$; ie. can't easily find other message with same digest as given message.
  - It is infeasible to locate two messages, m_1 and m_2, which "collide", i.e. for which $H(m_1) = H(m_2)$
  - A small change in a message changes many bits of digest/can't tell anything about message given its hash

Use of Hash Functions

- Several Standard Hash Functions:
  - MD5: 128-bit output
  - SHA-1: 160-bit output
- Can we use hashing to securely reduce load on server?
  - Yes. Use a series of insecure mirror servers (caches)
  - First, ask server for digest of desired file
    - » Use secure channel with server
  - Then ask mirror server for file
    - » Can be insecure channel
    - » Check digest of result and catch faulty or malicious mirrors



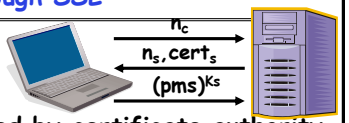
BREAK

Signatures/Certificate Authorities

- Can use X_{public} for person X to define their identity
  - Presumably they are the only ones who know $X_{private}$.
  - Often, we think of X_{public} as a "principle" (user)
- Suppose we want X to sign message M?
  - Use private key to encrypt the digest, i.e. $H(M)^{Xprivate}$
  - Send both M and its signature:
    - » Signed message = $[M,H(M)^{Xprivate}]$
  - Now, anyone can verify that M was signed by X
    - » Simply decrypt the digest with X_{public}
    - » Verify that result matches H(M)
- Now: How do we know that the version of X_{public} that we have is really from X???
  - Answer: Certificate Authority
    - » Examples: Verisign, Entrust, Etc.
  - X goes to organization, presents identifying papers
    - » Organization signs X's key: $[X_{public}, H(X_{public})^{CAprivate}]$
    - » Called a "Certificate"
  - Before we use X_{public}, ask X for certificate verifying key
    - » Check that signature over X_{public} produced by trusted authority
- How do we get keys of certificate authority?
  - Compiled into your browser, for instance!

Security through SSL

- SSL Web Protocol
  - Port 443: secure http
  - Use public-key encryption for key-distribution



- Server has a certificate signed by certificate authority
  - Contains server info (organization, IP address, etc)
  - Also contains server's public key and expiration date
- Establishment of Shared, 48-byte "master secret"
  - Client sends 28-byte random value n_c to server
  - Server returns its own 28-byte random value n_s, plus its certificate $cert_s$
  - Client verifies certificate by checking with public key of certificate authority compiled into browser
    - » Also check expiration date
  - Client picks 46-byte "premaster" secret (pms), encrypts it with public key of server, and sends to server
  - Now, both server and client have n_c, n_s, and pms
    - » Each can compute 48-byte master secret using one-way and collision-resistant function on three values
    - » Random "nonces" n_c and n_s make sure master secret fresh

SSL Pitfalls

- Netscape claimed to provide secure comm. (SSL)
  - So you could send a credit card # over the Internet
- Three problems (reported in NYT):
  - Algorithm for picking session keys was predictable (used time of day) – brute force key in a few hours
  - Made new version of Netscape to fix #1, available to users over Internet (unencrypted!)
    - » Four byte patch to Netscape executable makes it always use a specific session key
    - » Could insert backdoor by mangling packets containing executable as they fly by on the Internet.
    - » Many mirror sites (including Berkeley) to redistribute new version – anyone with root access to any machine on LAN at mirror site could insert the backdoor
  - Buggy helper applications – can exploit *any* bug in either Netscape, or its helper applications

Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D3 can read F2 or execute F3
  - In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Authorization: Two Implementation Choices

- Access Control Lists: store permissions with object
  - Still might be lots of users!
  - UNIX limits each file to: r,w,x for owner, group, world
  - More recent systems allow definition of groups of users and permissions for each group
  - ACLs allow easy changing of an object's permissions
    - » Example: add Users C, D, and F with rw permissions
- Capability List: each process tracks which objects has permission to touch
  - Popular in the past, idea out of favor today
  - Consider page table: Each process has list of pages it has access to, not each page has list of processes …
  - Capability lists allow easy changing of a domain's permissions
    - » Example: you are promoted to system administrator and should be given access to all system files

Authorization: Combination Approach

- Users have capabilities, called "groups" or "roles"
  - Everyone with particular group access is "equivalent" when accessing group resource
  - Like passport (which gives access to country of origin)
- Objects have ACLs
  - ACLs can refer to users or groups
  - Change object permissions object by modifying ACL
  - Change broad user permissions via changes in group membership
  - Possessors of proper credentials get access

Authorization: How to Revoke?

- How does one revoke someone's access rights to a particular object?
  - Easy with ACLs: just remove entry from the list
  - Takes effect immediately since the ACL is checked on each object access
- Harder to do with capabilities since they aren't stored with the object being controlled:
  - Not so bad in a single machine: could keep all capability lists in a well-known place (e.g., the OS capability table).
  - Very hard in distributed system, where remote hosts may have crashed or may not cooperate (more in a future lecture)

Revoking Capabilities

- Various approaches to revoking capabilities:
  - Put expiration dates on capabilities and force reacquisition
  - Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt)
  - Maintain back pointers to all capabilities that have been handed out (Tough if capabilities can be copied)
  - Maintain a revocation list that gets checked on every access attempt

Summary

- User Identification
  - Passwords/Smart Cards/Biometrics
- Passwords
  - Encrypt them to help hid them
  - Force them to be longer/not amenable to dictionary attack
  - Use zero-knowledge request-response techniques
- Distributed identity
  - Use cryptography
- Private Key Encryption (also Symmetric Key)
  - Single Key used to encode and decode
  - Pros: Very Fast
    - » can encrypt at network speed (even without hardware)
  - Cons: Need to distribute *secret* key to both parties

Summary (cont'd)

- Secure Hash Function
  - Fixed length summary of data
  - Hard to find another block of data with same hash
- Public Key Encryption (also Asymmetric Key)
  - Two keys: a public key and a private key
    - » Not derivable from one another
  - Pros: Can distribute keys in public
    - » Need certificate authority (Public Key Infrastructure)
  - Cons: Very Slow
    - » 100—1000 times slower than private key encryption
- Session Key
  - Randomly generated private key used for single session
  - Often distributed via public key encryption

CS162
Operating Systems and Systems Programming
Lecture 26

Protection and Security in Distributed Systems II

May 7, 2008

Prof. Anthony D. Joseph
http://inst.eecs.berkeley.edu/~cs162

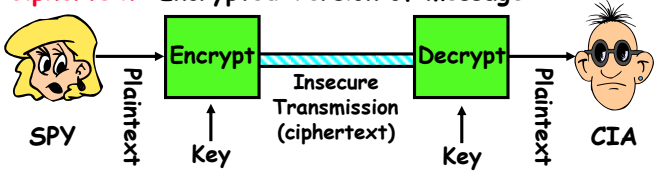
Review: Authentication: Identifying Users

- How to identify users to the system?
  - Passwords
    - » Shared secret between two parties
    - » Since only user knows password, someone types correct password ⇒ must be user typing it
    - » Very common technique
  - Smart Cards
    - » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries
    - » May have display to allow reading of password
    - » Or can be plugged in directly; several credit cards now in this category

  - Biometrics
    - » Use of one or more intrinsic physical or behavioral traits to identify someone
    - » Examples: fingerprint reader, palm reader, retinal scan
    - » Becoming quite a bit more common

Review: Private Key Cryptography

- Private Key (Symmetric) Encryption:
  - Single key used for both encryption and decryption
- Plaintext: Unencrypted Version of message
- Ciphertext: Encrypted Version of message



- Important properties
  - Can't derive plain text from ciphertext (decode) without access to key
  - Can't derive key from plain text and ciphertext
  - As long as password stays secret, get both secrecy and authentication
- Symmetric Key Algorithms: DES, Triple-DES, AES

Goals for Today

- Public Encryption
- Use of Cryptographic Mechanisms
- Authorization Mechanisms
- Worms and Viruses

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Recall: Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D_3 can read F_2 or execute F_3
  - In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- Two approaches to implementation
  - Access Control Lists: store permissions with each object
    - » Still might be lots of users!
    - » UNIX limits each file to: r,w,x for owner, group, world
    - » More recent systems allow definition of groups of users and permissions for each group
  - Capability List: each process tracks objects has permission to touch
    - » Popular in the past, idea out of favor today
    - » Consider page table: Each process has list of pages it has access to, not each page has list of processes …

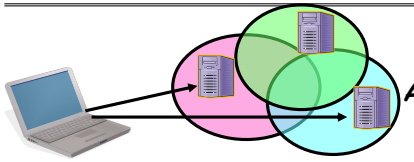
How fine-grained should access control be?

- Example of the problem:
  - Suppose you buy a copy of a new game from "Joe's Game World" and then run it.
  - It's running with your userid
    - » It removes all the files you own, including the project due the next day…
- How can you prevent this?
  - Have to run the program under *some* userid.
    - » Could create a second *games* userid for the user, which has no write privileges.
    - » Like the "nobody" userid in UNIX – can't do much
  - But what if the game needs to write out a file recording scores?
    - » Would need to give write privileges to one particular file (or directory) to your *games* userid.
  - But what about non-game programs you want to use, such as Quicken?
    - » Now you need to create your own private *quicken* userid, if you want to make sure tha the copy of Quicken you bought can't corrupt non-quicken-related files
  - But – how to get this right??? Pretty complex…

Authorization Continued

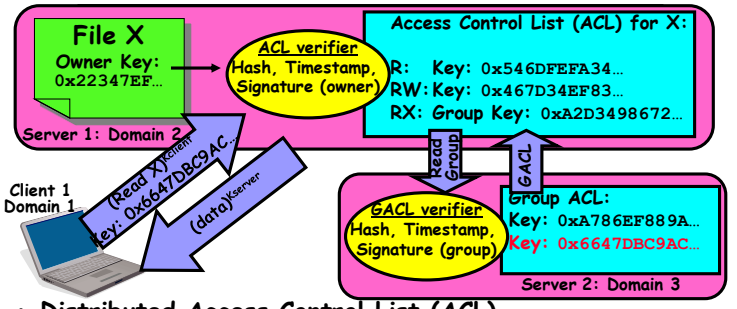
- Principle of least privilege: programs, users, and systems should get only enough privileges to perform their tasks
  - Very hard to do in practice
    - » How do you figure out what the minimum set of privileges is needed to run your programs?
  - People often run at higher privilege then necessary
    - » Such as the "administrator" privilege under windows
- One solution: Signed Software
  - Only use software from sources that you trust, thereby dealing with the problem by means of authentication
  - Fine for big, established firms such as Microsoft, since they can make their signing keys well known and people trust them
    - » Actually, not always fine: recently, one of Microsoft's signing keys was compromised, leading to malicious software that looked valid
  - What about new startups?
    - » Who "validates" them?
    - » How easy is it to fool them?

How to perform Authorization for Distributed Systems?



- Issues: Are all user names in world unique?
  - No! They only have small number of characters
    - » kubi@mit.edu → kubitron@lcs.mit.edu → kubitron@cs.berkeley.edu
    - » However, someone thought their friend was kubi@mit.edu and I got very private email intended for someone else…
  - Need something better, more unique to identify person
- Suppose want to connect with any server at any time?
  - Need an account on every machine! (possibly with different user name for each account)
  - OR: Need to use something more universal as identity
    - » Public Keys! (Called "Principles")
    - » People *are* their public keys

Distributed Access Control



- Distributed Access Control List (ACL)
  - Contains list of attributes (Read, Write, Execute, etc) with attached identities (Here, we show public keys)
    - » ACLs signed by owner of file, only changeable by owner
    - » Group lists signed by group key
  - ACLs can be on different servers than data
    - » Signatures allow us to validate them
    - » ACLs could even be stored separately from verifiers

Analysis of Previous Scheme

- Positive Points:
  - Identities checked via signatures and public keys
    - » Client can't generate request for data unless they have private key to go with their public identity
    - » Server won't use ACLs not properly signed by owner of file
  - No problems with multiple domains, since identities designed to be cross-domain (public keys domain neutral)
- Revocation:
  - What if someone steals your private key?
    - » Need to walk through all ACLs with your key and change…!
    - » This is very expensive
  - Better to have unique string identifying you that people place into ACLs
    - » Then, ask Certificate Authority to give you a certificate matching unique string to your current public key
    - » Client Request: (request + unique ID)^{Cprivate}; give server certificate if they ask for it.
    - » Key compromise⇒must distribute "certificate revocation", since can't wait for previous certificate to expire.
  - What if you remove someone from ACL of a given file?
    - » If server caches old ACL, then person retains access!
    - » Here, cache inconsistency leads to security violations!

Analysis Continued

- Who signs the data?
  - Or: How does the client know they are getting valid data?
  - Signed by server?
    - » What if server compromised? Should client trust server?
  - Signed by owner of file?
    - » Better, but now only owner can update file!
    - » Pretty inconvenient!
  - Signed by group of servers that accepted latest update?
    - » If must have signatures from all servers ⇒ Safe, but one bad server can prevent update from happening
    - » Instead: ask for a threshold number of signatures
    - » Byzantine agreement can help here
- How do you know that data is up-to-date?
  - Valid signature only means data is valid older version
  - Freshness attack:
    - » Malicious server returns old data instead of recent data
    - » Problem with both ACLs and data
    - » E.g.: you just got a raise, but enemy breaks into a server and prevents payroll from seeing latest version of update
  - Hard problem
    - » Needs to be fixed by invalidating old copies or having a trusted group of servers (Byzantine Agrement?)

Administrivia

- Project #4 code deadline is Wed 5/14 at 11:59pm
- Final Exam
  - May 21st, 12:30-3:30pm
- Final Topics: Any suggestions?
  - Please send them to me…

Involuntary Installation

- What about software loaded without your consent?
  - Macros attached to documents (such as Microsoft Word)
  - Active X controls (programs on web sites with potential access to whole machine)
  - Spyware included with normal products
- Active X controls can have access to the local machine
  - Install software/Launch programs
- Sony Spyware [Sony XCP] (October 2005)
  - About 50 recent CDs from Sony automatically install software when you played them on Windows machines
    - » Called XCP (Extended Copy Protection)
    - » Modify operating system to prevent more than 3 copies and to prevent peer-to-peer sharing
  - Side Effects:
    - » Reporting of private information to Sony
    - » Hiding of generic file names of form $sys_xxx; easy for other virus writers to exploit
    - » Hard to remove (crashes machine if not done carefully)
  - Vendors of virus protection software declare it spyware
    - » Computer Associates, Symantec, even Microsoft

Enforcement

- Enforcer checks passwords, ACLs, etc
  - Makes sure the only authorized actions take place
  - Bugs in enforcer⇒things for malicious users to exploit
- In UNIX, superuser can do anything
  - Because of coarse-grained access control, lots of stuff has to run as superuser in order to work
  - If there is a bug in any one of these programs, you lose!
- Paradox
  - Bullet-proof enforcer
    - » Only known way is to make enforcer as small as possible
    - » Easier to make correct, but simple-minded protection model
  - Fancy protection
    - » Tries to adhere to principle of least privilege
    - » Really hard to get right
- Same argument for Java or C++: What do you make private vs public?
  - Hard to make sure that code is usable but only necessary modules are public
  - Pick something in middle? Get bugs and weak protection!

State of the World

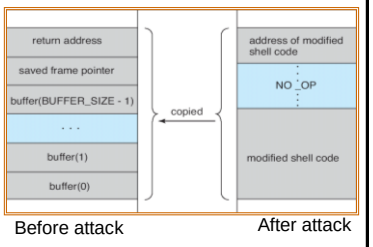
- State of the World in Security
  - Authentication: Encryption
    - » But almost no one encrypts or has public key identity
  - Authorization: Access Control
    - » But many systems only provide very coarse-grained access
    - » In UNIX, need to turn off protection to enable sharing
  - Enforcement: Kernel mode
    - » Hard to write a million line program without bugs
    - » Any bug is a potential security loophole!
- Some types of security problems
  - Abuse of privilege
    - » If the superuser is evil, we're all in trouble/can't do anything
    - » What if sysop in charge of instructional resources went crazy and deleted everybody's files (and backups)???
  - Imposter: Pretend to be someone else
    - » Example: in unix, can set up an .rhosts file to allow logins from one machine to another without retyping password
    - » Allows "rsh" command to do an operation on a remote node
    - » Result: send rsh request, pretending to be from trusted user→install .rhosts file granting you access

Other Security Problems

- Virus:
  - A piece of code that attaches itself to a program or file so it can spread from one computer to another, leaving infections as it travels
  - Most attached to executable files, so don't get activated until the file is actually executed
  - Once caught, can hide in boot tracks, other files, OS
- Worm:
  - Similar to a virus, but capable of traveling on its own
  - Takes advantage of file or information transport features
  - Because it can replicate itself, your computer might send out hundreds or thousands of copies of itself
- Trojan Horse:
  - Named after huge wooden horse in Greek mythology given as gift to enemy; contained army inside
  - At first glance appears to be useful software but does damage once installed or run on your computer

Security Problems: Buffer-overflow Condition

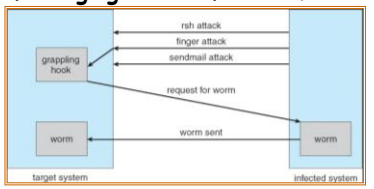
```
#define BUFFER SIZE 256
int process(int argc,
            char *argv[])
{
  char buffer[BUFFER SIZE];
  if (argc < 2)
       return -1;
  else {
       strcpy(buffer,argv[1]);
       return 0;
  }
}
```



- Technique exploited by many network attacks
  - Anytime input comes from network request and is not checked for size
  - Allows execution of code with same privileges as running program – but happens without any action from user!
- How to prevent?
  - Don't code this way! (ok, wishful thinking)
  - New mode bits in Intel, Amd, and Sun processors
    - » Put in page table; says "don't execute code in this page"

The Morris Internet Worm

- Internet worm (Self-reproducing)
  - Author Robert Morris, a first-year Cornell grad student
  - Launched close of Workday on November 2, 1988
  - Within a few hours of release, it consumed resources to the point of bringing down infected machines



- Techniques
  - Exploited UNIX networking features (remote access)
  - Bugs in *finger* (buffer overflow) and *sendmail* programs (debug mode allowed remote login)
  - Dictionary lookup-based password cracking
  - Grappling hook program uploaded main worm program

BREAK

Some other Attacks

- Trojan Horse Example: Fake Login
  - Construct a program that looks like normal login program
  - Gives "login:" and "password:" prompts
    - » You type information, it sends password to someone, then either logs you in or says "Permission Denied" and exits
  - In Windows, the "ctrl-alt-delete" sequence is supposed to be really hard to change, so you "know" that you are getting official login program
- Salami attack: Slicing things a little at a time
  - Steal or corrupt something a little bit at a time
  - E.g.: What happens to partial pennies from bank interest?
    - » Bank keeps them! Hacker re-programmed system so that partial pennies would go into his account.
    - » Doesn't seem like much, but if you are large bank can be millions of dollars
- Eavesdropping attack
  - Tap into network and see everything typed
  - Catch passwords, etc
  - Lesson: never use unencrypted communication!

Tenex Password Checking

- Tenex – early 70's, BBN
  - Most popular system at universities before UNIX
  - Thought to be very secure, gave "red team" all the source code and documentation (want code to be publicly available, as in UNIX)
  - In 48 hours, they figured out how to get every password in the system
- Here's the code for the password check:

```
for (i = 0; i < 8; i++)
  if (userPasswd[i] != realPasswd[i])
    go to error
```

- How many combinations of passwords?
  - 256^8?
  - Wrong!

Defeating Password Checking

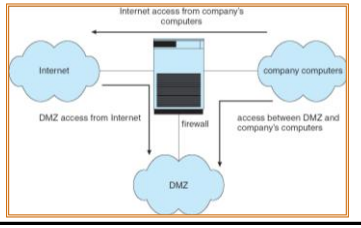
- Tenex used VM, and it interacts badly with the above code
  - Key idea: force page faults at inopportune times to break passwords quickly
- Arrange 1st char in string to be last char in pg, rest on next pg
  - Then arrange for pg with 1st char to be in memory, and rest to be on disk (e.g., ref lots of other pgs, then ref 1st page)

```
        a|aaaaaa
         |
page in memory| page on disk
```

- Time password check to determine if first character is correct!
  - If fast, 1st char is wrong
  - If slow, 1st char is right, pg fault, one of the others wrong
  - So try all first characters, until one is slow
  - Repeat with first two characters in memory, rest on disk
- Only 256 * 8 attempts to crack passwords
  - Fix is easy, don't stop until you look at all the characters

Defense in Depth: Layered Network Security

- How do I minimize the damage when security fails?
  - For instance: I make a mistake in the specification
  - Or: A bug lets something run that shouldn't?
- Firewall: Examines every packet to/from public internet
  - Can disable all traffic to/from certain ports
  - Can route certain traffic to DMZ (De-Militarized Zone)
    - » Semi-secure area separate from critical systems
  - Can do network address translation
    - » Inside network, computers have private IP addresses
    - » Connection from inside→outside is translated
    - » E.g. [10.0.0.2,port 2390] → [169.229.60.38,port 80]
      [12.4.35.2,port 5592] → [169.229.60.38,port 80]



Shrink Wrap Software Woes

- Can I trust software installed by the computer manufacturer?
  - Not really, most major computer manufacturers have shipped computers with viruses
  - How?
    - » Forgot to update virus scanner on "gold" master machine
- Software companies, PR firms, and others routinely release software that contains viruses
- Linux hackers say "Start with the source"
  - Does that work?

Ken Thompson's self-replicating program

- Bury Trojan horse in binaries, so no evidence in source
  - Replicates itself to every UNIX system in the world and even to new UNIX's on new platforms. No visible sign.
  - Gave Ken Thompson ability to log into any UNIX system
- Two steps: Make it possible (easy); Hide it (tricky)
- Step 1: Modify login.c

```
A: if (name == "ken")
       don't check password
       log in as root
```

  - Easy to do but pretty blatant! Anyone looking will see.
- Step 2: Modify C compiler
  - Instead of putting code in login.c, put in compiler:

```
B: if see trigger1
     insert A into input stream
```

  - Whenever compiler sees trigger1 (say /*gobbledygook*/), puts A into input stream of compiler
  - Now, don't need A in login.c, just need trigger1

Self Replicating Program Continued

- Step 3: Modify compiler source code:

```
C: if see trigger2
     insert B+C into input stream
```

  - Now compile this new C compiler to produce binary
- Step 4: Self-replicating code!
  - Simply remove statement C in compiler source code and place "trigger2" into source instead
    - » As long as existing C compiler is used to recompile the C compiler, the code will stay into the C compiler and will compile back door into login.c
    - » But no one can see this from source code!
- When porting to new machine/architecture, use existing C compiler to generate cross-compiler
  - Code will migrate to new architecture!
- Lesson: never underestimate the cleverness of computer hackers for hiding things!

Conclusion

- Authorization
  - Abstract table of users (or domains) vs permissions
  - Implemented either as access-control list or capability list
- Issues with distributed storage example
  - Revocation: How to remove permissions from someone?
  - Integrity: How to know whether data is valid
  - Freshness: How to know whether data is recent
- Buffer-Overrun Attack: exploit bug to execute code

Nachos for Java README

Welcome to Nachos for Java. We believe that working in Java rather than C++ will greatly simplify the development process by preventing bugs arising from memory management errors, and improving debugging support.

Getting Nachos:

Download nachos-java.tar.gz from the Projects section of the class homepage at:

```
http://www-inst.EECS.Berkeley.EDU/~cs162/
```

Unpack it with these commands:

```
gunzip -c nachos-java.tar.gz | tar xf -
```

Additional software:

Nachos requires the Java Devlopment Kit, version 1.5 or later. This is installed on all instructional machines in:

```
/usr/sww/lang/jdk-1.5.0_05
```

To use this version of the JDK, be sure that

```
/usr/sww/lang/jdk-1.5.0_05/bin
```

is on your PATH. (This should be the case for all class accounts already.)

If you are working at home, you will need to download the JDK. It is available from:

```
http://java.sun.com/j2se/1.5/
```

Please DO NOT DOWNLOAD the JDK into your class account! Use the preinstalled version instead.

The build process for Nachos relies on GNU make. If you are running on one of the instructional machines, be sure you run 'gmake', as 'make' does not support all the features used. If you are running Linux, the two are equivalent. If you are running Windows, you will need to download and install a port. The most popular is the Cygnus toolkit, available at:

```
http://sources.redhat.com/cygwin/mirrors.html
```

The Cygnus package includes ports of most common GNU utilities to Windows.

For project 2, you will need a MIPS cross compiler, which is a specially compiled GCC which will run on one architecture (e.g. Sparc) and produce files for the MIPS processor. These compilers are already installed on the instructional machines, and are available in the directory specified by the $ARCHDIR environment variable.

If you are working at home, you will need to get a cross-compiler for yourself. Cross-compilers for Linux and Win32 will be available from the CS162 Projects web page. Download the cross compiler distribution and unpack it with the following command:

```
gunzip -c mips-x86-linux-xgcc.tar.gz | tar xf -
```

(Substitute the appropriate file name for mips-x86.linux-xgcc in the above command.) You need to add the mips-x86.linux-xgcc directory to your PATH, and set an environment variable ARCHDIR to point to this directory. (Again, this has already been done for you on the instructional machines.)

Compiling Nachos:

You should now have a directory called nachos, containing a Makefile, this README, and a number of subdirectories.

First, put the 'nachos/bin' directory on your PATH. This directory contains the script 'nachos', which simply runs the Nachos code.

To compile Nachos, go to the subdirectory for the project you wish to compile (I will assume 'proj1/' for Project 1 in my examples), and run:

```
gmake
```

This will compile those portions of Nachos which are relevant to the project, and place the compiled .class files in the proj1/nachos directory.

You can now test Nachos from the proj1/ directory with:

```
nachos
```

You should see output resembling the following:

```
  nachos 5.0j initializing... config interrupt timer elevators user-check grader
  *** thread 0 looped 0 times
  *** thread 1 looped 0 times
  *** thread 0 looped 1 times
  *** thread 1 looped 1 times
  *** thread 0 looped 2 times
  *** thread 1 looped 2 times
  *** thread 0 looped 3 times
  *** thread 1 looped 3 times
  *** thread 0 looped 4 times
  *** thread 1 looped 4 times
  Machine halting!

  Ticks: total 24750, kernel 24750, user 0
  Disk I/O: reads 0, writes 0
  Console I/O: reads 0, writes 0
  Paging: page faults 0, TLB misses 0
  Network I/O: received 0, sent 0
```

This is the correct output for the "bare bones" Nachos, without any of the features you will add during the projects.

If you are working on a project which runs user programs (projects 2-4), you will also need to compile the MIPS test programs with:

```
gmake test
```

Command Line Arguments:

For a summary of the command line arguments, run:

```
nachos -h
```

The commands are:

```
-d <debug flags>
        Enable some debug flags, e.g. -d ti

-h
        Print this help message.
```

```
        -s <seed>
                Specify the seed for the random number generator

        -x <program>
                Specify a program that UserKernel.run() should execute,
                instead of the value of the configuration variable
                Kernel.shellProgram

        -z
                print the copyright message

        -- <grader class>
                Specify an autograder class to use, instead of
                nachos.ag.AutoGrader

        -# <grader arguments>
                Specify the argument string to pass to the autograder.

        -[] <config file>
                Specifiy a config file to use, instead of nachos.conf


Nachos offers the following debug flags:

    c: COFF loader info
    i: HW interrupt controller info
    p: processor info
    m: disassembly
    M: more disassembly
    t: thread info
    a: process info (formerly "address space", hence a)

To use multiple debug flags, clump them all together. For example, to
monitor coff info and process info, run:

        nachos -d ac

nachos.conf:

When Nachos starts, it reads in nachos.conf from the current
directory.  It contains a bunch of keys and values, in the simple
format "key = value" with one key/value pair per line. To change the
default scheduler, default shell program, to change the amount of
memory the simulator provides, or to reduce network reliability, modify
this file.

Machine.stubFileSystem:
    Specifies whether the machine should provide a stub file system. A
    stub file system just provides direct access to the test directory.
    Since we're not doing the file system project, this should always
    be true.

Machine.processor:
    Specifies whether the machine should provide a MIPS processor. In
    the first project, we only run kernel code, so this is false. In
    the other projects it should be true.

Machine.console:
    Specifies whether the machine should provide a console. Again, the
    first project doesn't need it, but the rest of them do.

Machine.disk:
    Specifies whether the machine should provide a simulated disk. No
    file system project, so this should always be false.

ElevatorBank.allowElevatorGUI:
    Normally true. When we grade, this will be false, to prevent
    malicious students from running a GUI during grading.

NachosSecurityManager.fullySecure:
    Normally false. When we grade, this will be true, to enable
    additional security checks.

Kernel.kernel:
    Specifies what kernel class to dynmically load.  For proj1, this is
    nachos.threads.ThreadedKernel. For proj2, this should be
    nachos.userprog.UserKernel. For proj3, nachos.vm.VMKernel. For
    proj4, nachos.network.NetKernel.

Processor.usingTLB:
    Specifies whether the MIPS processor provides a page table
    interface or a TLB interface. In page table mode (proj2), the
    processor accesses an arbitrarily large kernel data structure to do
    address translation. In TLB mode (proj3 and proj4), the processor
    maintains a small TLB (4 entries).

Processor.numPhysPages:
    The number of pages of physical memory.  Each page is 1K. This is
    normally 64, but we can lower it in proj3 to see whether projects
    thrash or crash.

Documentation:

The JDK provides a command to create a set of HTML pages showing all
classes and methods in program. We will make these pages available on
the webpage, but you can create your own for your home machine by doing
the following (from the nachos/ directory):

        mkdir ../doc
        gmake doc

Troubleshooting:

If you receive an error about "class not found exception", it may be
because you have not set the CLASSPATH environment variable. Add the
following to your .cshrc:

        setenv CLASSPATH .

Credits:

Nachos was originally written by Wayne A. Christopher, Steven J.
Procter, and Thomas E. Anderson. It incorporates the SPIM simulator
written by John Ousterhout. Nachos was rewritten in Java by Daniel
Hettena.

Copyright:

Copyright (c) 1992-2001 The Regents of the University of California.
All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose, without fee, and without written
agreement is hereby granted, provided that the above copyright notice
and the following two paragraphs appear in all copies of this
software.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY
FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF
```

THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE
PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE UNIVERSITY OF
CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

```
JAVADOCPARAMS = -doctitle "Nachos 5.0 Java" -protected \
                -link http://java.sun.com/j2se/1.5.0/docs/api/

machine =       Lib Config Stats Machine TCB \
                Interrupt Timer \
                Processor TranslationEntry \
                SerialConsole StandardConsole \
                OpenFile OpenFileWithPosition ArrayFile FileSystem StubFileSystem \
                ElevatorBank ElevatorTest ElevatorGui \
                ElevatorControls ElevatorEvent ElevatorControllerInterface \
                RiderControls RiderEvent RiderInterface \
                Kernel Coff CoffSection \
                NetworkLink Packet MalformedPacketException

security =      Privilege NachosSecurityManager

ag =            AutoGrader BoatGrader

threads =       ThreadedKernel KThread Alarm \
                Scheduler ThreadQueue RoundRobinScheduler \
                Semaphore Lock Condition SynchList \
                Condition2 Communicator Rider ElevatorController \
                PriorityScheduler LotteryScheduler Boat

userprog =      UserKernel UThread UserProcess SynchConsole

vm =            VMKernel VMProcess

network =       NetKernel NetProcess PostOffice MailMessage

ALLDIRS = machine security ag threads userprog vm network

PACKAGES := $(patsubst %,nachos.%,$(ALLDIRS))

CLASSFILES := $(foreach dir,$(DIRS),$(patsubst %,nachos/$(dir)/%.class,$($(dir))))

.PHONY: all rmtemp clean doc hwdoc swdoc

all: $(CLASSFILES)

nachos/%.class: ../%.java
        javac -classpath . -d . -sourcepath ../.. -g $<

clean:
        rm -f */*/*.class

doc:
        mkdir -p ../doc
        javadoc $(JAVADOCPARAMS) -d ../doc -sourcepath .. $(PACKAGES)

test:
        cd ../test ; gmake

ag:     $(patsubst ../ag/%.java,nachos/ag/%.class,$(wildcard ../ag/*.java))
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.ag;

import nachos.machine.*;
import nachos.security.*;
import nachos.threads.*;

import java.util.Hashtable;
import java.util.StringTokenizer;

/**
 * The default autograder. Loads the kernel, and then tests it using
 * <tt>Kernel.selfTest()</tt>.
 */
public class AutoGrader {
    /**
     * Allocate a new autograder.
     */
    public AutoGrader() {
    }

    /**
     * Start this autograder. Extract the <tt>-#</tt> arguments, call
     * <tt>init()</tt>, load and initialize the kernel, and call
     * <tt>run()</tt>.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public void start(Privilege privilege) {
        Lib.assertTrue(this.privilege == null,
                   "start() called multiple times");
        this.privilege = privilege;

        String[] args = Machine.getCommandLineArguments();

        extractArguments(args);

        System.out.print(" grader");

        init();

        System.out.print("\n");

        kernel =
            (Kernel) Lib.constructObject(Config.getString("Kernel.kernel"));
        kernel.initialize(args);

        run();
    }

    private void extractArguments(String[] args) {
        String testArgsString = Config.getString("AutoGrader.testArgs");
        if (testArgsString == null) {
                testArgsString = "";
        }

        for (int i=0; i<args.length; ) {
            String arg = args[i++];
            if (arg.length() > 0 && arg.charAt(0) == '-') {
                if (arg.equals("-#")) {
                    Lib.assertTrue(i < args.length,
                               "-# switch missing argument");
                    testArgsString = args[i++];
                }
            }
        }

        StringTokenizer st = new StringTokenizer(testArgsString, ",\n\t\f\r");

        while (st.hasMoreTokens()) {
            StringTokenizer pair = new StringTokenizer(st.nextToken(), "=");

            Lib.assertTrue(pair.hasMoreTokens(),
                       "test argument missing key");
            String key = pair.nextToken();

            Lib.assertTrue(pair.hasMoreTokens(),
                       "test argument missing value");
            String value = pair.nextToken();

            testArgs.put(key, value);
        }
    }

    String getStringArgument(String key) {
        String value = (String) testArgs.get(key);
        Lib.assertTrue(value != null,
                   "getStringArgument(" + key + ") failed to find key");
        return value;
    }

    int getIntegerArgument(String key) {
        try {
            return Integer.parseInt(getStringArgument(key));
        }
        catch (NumberFormatException e) {
            Lib.assertNotReached("getIntegerArgument(" + key + ") failed: " +
                                 "value is not an integer");
            return 0;
        }
    }

    boolean getBooleanArgument(String key) {
        String value = getStringArgument(key);

        if (value.equals("1") || value.toLowerCase().equals("true")) {
            return true;
        }
        else if (value.equals("0") || value.toLowerCase().equals("false")) {
            return false;
        }
        else {
            Lib.assertNotReached("getBooleanArgument(" + key + ") failed: " +
                                 "value is not a boolean");
            return false;
        }
    }

    long getTime() {
        return privilege.stats.totalTicks;
    }

    void targetLevel(int targetLevel) {
        this.targetLevel = targetLevel;
    }

    void level(int level) {
        this.level++;
```

```
    Lib.assertTrue(level == this.level,
               "level() advanced more than one step: test jumped ahead");

    if (level == targetLevel)
        done();
}

private int level = 0, targetLevel = 0;

void done() {
    System.out.print("\nsuccess\n");
    privilege.exit(162);
}

private Hashtable<String, String> testArgs =
    new Hashtable<String, String>();

void init() {
}

void run() {
    kernel.selfTest();
    kernel.run();
    kernel.terminate();
}

Privilege privilege = null;
Kernel kernel;

/**
 * Notify the autograder that the specified thread is the idle thread.
 * <tt>KThread.createIdleThread()</tt> <i>must</i> call this method before
 * forking the idle thread.
 *
 * @param   idleThread      the idle thread.
 */
public void setIdleThread(KThread idleThread) {
}

/**
 * Notify the autograder that the specified thread has moved to the ready
 * state. <tt>KThread.ready()</tt> <i>must</i> call this method before
 * returning.
 *
 * @param   thread  the thread that has been added to the ready set.
 */
public void readyThread(KThread thread) {
}

/**
 * Notify the autograder that the specified thread is now running.
 * <tt>KThread.restoreState()</tt> <i>must</i> call this method before
 * returning.
 *
 * @param   thread  the thread that is now running.
 */
public void runningThread(KThread thread) {
    privilege.tcb.associateThread(thread);
    currentThread = thread;
}

/**
 * Notify the autograder that the current thread has finished.
 * <tt>KThread.finish()</tt> <i>must</i> call this method before putting
 * the thread to sleep and scheduling its TCB to be destroyed.
 */
public void finishingCurrentThread() {
    privilege.tcb.authorizeDestroy(currentThread);
}

/**
 * Notify the autograder that a timer interrupt occurred and was handled by
 * software if a timer interrupt handler was installed. Called by the
 * hardware timer.
 *
 * @param   privilege       proves the authenticity of this call.
 * @param   time    the actual time at which the timer interrupt was
 *                  issued.
 */
public void timerInterrupt(Privilege privilege, long time) {
    Lib.assertTrue(privilege == this.privilege,
               "security violation");
}

/**
 * Notify the autograder that a user program executed a syscall
 * instruction.
 *
 * @param   privilege       proves the authenticity of this call.
 * @return  <tt>true</tt> if the kernel exception handler should be called.
 */
public boolean exceptionHandler(Privilege privilege) {
    Lib.assertTrue(privilege == this.privilege,
               "security violation");
    return true;
}

/**
 * Notify the autograder that <tt>Processor.run()</tt> was invoked. This
 * can be used to simulate user programs.
 *
 * @param   privilege       proves the authenticity of this call.
 */
public void runProcessor(Privilege privilege) {
    Lib.assertTrue(privilege == this.privilege,
               "security violation");
}

/**
 * Notify the autograder that a COFF loader is being constructed for the
 * specified file. The autograder can use this to provide its own COFF
 * loader, or return <tt>null</tt> to use the default loader.
 *
 * @param   file    the executable file being loaded.
 * @return  a loader to use in loading the file, or <tt>null</tt> to use
 *          the default.
 */
public Coff createLoader(OpenFile file) {
    return null;
}

/**
 * Request permission to send a packet. The autograder can use this to drop
 * packets very selectively.
 *
 * @param   privilege       proves the authenticity of this call.
 * @return  <tt>true</tt> if the packet should be sent.
 */
public boolean canSendPacket(Privilege privilege) {
    Lib.assertTrue(privilege == this.privilege,
```

```
                "security violation");
        return true;
    }

    /**
     * Request permission to receive a packet. The autograder can use this to
     * drop packets very selectively.
     *
     * @param   privilege       proves the authenticity of this call.
     * @return  <tt>true</tt> if the packet should be delivered to the kernel.
     */
    public boolean canReceivePacket(Privilege privilege) {
        Lib.assertTrue(privilege == this.privilege,
                "security violation");
        return true;
    }

    private KThread currentThread;
}
```

```
package nachos.ag;

public class BoatGrader {

    /**
     * BoatGrader consists of functions to be called to show that
     * your solution is properly synchronized. This version simply
     * prints messages to standard out, so that you can watch it.
     * You cannot submit this file, as we will be using our own
     * version of it during grading.

     * Note that this file includes all possible variants of how
     * someone can get from one island to another. Inclusion in
     * this class does not imply that any of the indicated actions
     * are a good idea or even allowed.
     */

    /* ChildRowToMolokai should be called when a child pilots the boat
       from Oahu to Molokai */
    public void ChildRowToMolokai() {
        System.out.println("**Child rowing to Molokai.");
    }

    /* ChildRowToOahu should be called when a child pilots the boat
       from Molokai to Oahu*/
    public void ChildRowToOahu() {
        System.out.println("**Child rowing to Oahu.");
    }

    /* ChildRideToMolokai should be called when a child not piloting
       the boat disembarks on Molokai */
    public void ChildRideToMolokai() {
        System.out.println("**Child arrived on Molokai as a passenger.");
    }

    /* ChildRideToOahu should be called when a child not piloting
       the boat disembarks on Oahu */
    public void ChildRideToOahu() {
        System.out.println("**Child arrived on Oahu as a passenger.");
    }

    /* AdultRowToMolokai should be called when a adult pilots the boat
       from Oahu to Molokai */
    public void AdultRowToMolokai() {
        System.out.println("**Adult rowing to Molokai.");
    }

    /* AdultRowToOahu should be called when a adult pilots the boat
       from Molokai to Oahu */
    public void AdultRowToOahu() {
        System.out.println("**Adult rowing to Oahu.");
    }

    /* AdultRideToMolokai should be called when an adult not piloting
       the boat disembarks on Molokai */
    public void AdultRideToMolokai() {
        System.out.println("**Adult arrived on Molokai as a passenger.");
    }

    /* AdultRideToOahu should be called when an adult not piloting
       the boat disembarks on Oahu */
    public void AdultRideToOahu() {
        System.out.println("**Adult arrived on Oahu as a passenger.");
    }
}
```

```
#!/bin/sh

# Shell-script front-end to run Nachos.
# Simply sets terminal to a minimum of one byte to complete a read and
# disables character echo. Restores original terminal state on exit.

onexit () {
  stty $OLDSTTYSTATE
}

OLDSTTYSTATE=‘stty -g‘
trap onexit 0
stty -icanon min 1 -echo
java nachos.machine.Machine $*
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A read-only <tt>OpenFile</tt> backed by a byte array.
 */
public class ArrayFile extends OpenFileWithPosition {
    /**
     * Allocate a new <tt>ArrayFile</tt>.
     *
     * @param   array   the array backing this file.
     */
    public ArrayFile(byte[] array) {
        this.array = array;
    }

    public int length() {
        return array.length;
    }

    public void close() {
        array = null;
    }

    public int read(int position, byte[] buf, int offset, int length) {
        Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <= buf.length);

        if (position < 0 || position >= array.length)
            return 0;

        length = Math.min(length, array.length-position);
        System.arraycopy(array, position, buf, offset, length);

        return length;
    }

    public int write(int position, byte[] buf, int offset, int length) {
        return 0;
    }

    private byte[] array;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import java.io.EOFException;

/**
 * A COFF (common object file format) loader.
 */
public class Coff {
    /**
     * Allocate a new Coff object.
     */
    protected Coff() {
        file = null;
        entryPoint = 0;
        sections = null;
    }

    /**
     * Load the COFF executable in the specified file.
     *
     * <p>
     * Notes:
     * <ol>
     * <li>If the constructor returns successfully, the file becomes the
     * property of this loader, and should not be accessed any further.
     * <li>The autograder expects this loader class to be used. Do not load
     * sections through any other mechanism.
     * <li>This loader will verify that the file is backed by a file system,
     * by asserting that read() operations take non-zero simulated time to
     * complete. Do not supply a file backed by a simulated cache (the primary
     * purpose of this restriction is to prevent sections from being loaded
     * instantaneously while handling page faults).
     * </ol>
     *
     * @param   file    the file containing the executable.
     * @exception       EOFException    if the executable is corrupt.
     */
    public Coff(OpenFile file) throws EOFException {
        this.file = file;

        Coff coff = Machine.autoGrader().createLoader(file);

        if (coff != null) {
            this.entryPoint = coff.entryPoint;
            this.sections = coff.sections;
        }
        else {
            byte[] headers = new byte[headerLength+aoutHeaderLength];

            if (file.length() < headers.length) {
                Lib.debug(dbgCoff, "\tfile is not executable");
                throw new EOFException();
            }

            Lib.strictReadFile(file, 0, headers, 0, headers.length);

            int magic = Lib.bytesToUnsignedShort(headers, 0);
            int numSections = Lib.bytesToUnsignedShort(headers, 2);
            int optionalHeaderLength = Lib.bytesToUnsignedShort(headers, 16);
            int flags = Lib.bytesToUnsignedShort(headers, 18);
            entryPoint = Lib.bytesToInt(headers, headerLength+16);

            if (magic != 0x0162) {
                Lib.debug(dbgCoff, "\tincorrect magic number");
                throw new EOFException();
            }
            if (numSections < 2 || numSections > 10) {
                Lib.debug(dbgCoff, "\tbad section count");
                throw new EOFException();
            }
            if ((flags & 0x0003) != 0x0003) {
                Lib.debug(dbgCoff, "\tbad header flags");
                throw new EOFException();
            }

            int offset = headerLength + optionalHeaderLength;

            sections = new CoffSection[numSections];
            for (int s=0; s<numSections; s++) {
                int sectionEntryOffset = offset + s*CoffSection.headerLength;
                try {
                    sections[s] =
                        new CoffSection(file, this, sectionEntryOffset);
                }
                catch (EOFException e) {
                    Lib.debug(dbgCoff, "\terror loading section " + s);
                    throw e;
                }
            }
        }
    }

    /**
     * Return the number of sections in the executable.
     *
     * @return  the number of sections in the executable.
     */
    public int getNumSections() {
        return sections.length;
    }

    /**
     * Return an object that can be used to access the specified section. Valid
     * section numbers include <tt>0</tt> through <tt>getNumSections() -
     * 1</tt>.
     *
     * @param   sectionNumber   the section to select.
     * @return  an object that can be used to access the specified section.
     */
    public CoffSection getSection(int sectionNumber) {
        Lib.assertTrue(sectionNumber >= 0 && sectionNumber < sections.length);

        return sections[sectionNumber];
    }

    /**
     * Return the program entry point. This is the value that to which the PC
     * register should be initialized to before running the program.
     *
     * @return  the program entry point.
     */
    public int getEntryPoint() {
        Lib.assertTrue(file != null);

        return entryPoint;
    }

    /**
```

```
     * Close the executable file and release any resources allocated by this
     * loader.
     */
    public void close() {
        file.close();

        sections = null;
    }

    private OpenFile file;

    /** The virtual address of the first instruction of the program. */
    protected int entryPoint;
    /** The sections in this COFF executable. */
    protected CoffSection sections[];

    private static final int headerLength = 20;
    private static final int aoutHeaderLength = 28;

    private static final char dbgCoff = 'c';
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

import java.io.EOFException;
import java.util.Arrays;

/**
 * A <tt>CoffSection</tt> manages a single section within a COFF executable.
 */
public class CoffSection {
    /**
     * Allocate a new COFF section with the specified parameters.
     *
     * @param   coff            the COFF object to which this section belongs.
     * @param   name            the COFF name of this section.
     * @param   executable      <tt>true</tt> if this section contains code.
     * @param   readOnly        <tt>true</tt> if this section is read-only.
     * @param   numPages        the number of virtual pages in this section.
     * @param   firstVPN        the first virtual page number used by this.
     */
    protected CoffSection(Coff coff, String name, boolean executable,
                          boolean readOnly, int numPages, int firstVPN) {
        this.coff = coff;
        this.name = name;
        this.executable = executable;
        this.readOnly = readOnly;
        this.numPages = numPages;
        this.firstVPN = firstVPN;

        file = null;
        size = 0;
        contentOffset = 0;
        initialized = true;
    }

    /**
     * Load a COFF section from an executable.
     *
     * @param   file            the file containing the executable.
     * @param   headerOffset    the offset of the section header in the
     *                          executable.
     *
     * @exception       EOFException    if an error occurs.
     */
    public CoffSection(OpenFile file, Coff coff,
                       int headerOffset) throws EOFException {
        this.file = file;
        this.coff = coff;

        Lib.assertTrue(headerOffset >= 0);
        if (headerOffset+headerLength > file.length()) {
            Lib.debug(dbgCoffSection, "\tsection header truncated");
            throw new EOFException();
        }

        byte[] buf = new byte[headerLength];
        Lib.strictReadFile(file, headerOffset, buf, 0, headerLength);

        name = Lib.bytesToString(buf, 0, 8);
        int vaddr = Lib.bytesToInt(buf, 12);
        size = Lib.bytesToInt(buf, 16);
        contentOffset = Lib.bytesToInt(buf, 20);
        int numRelocations = Lib.bytesToUnsignedShort(buf, 32);
        int flags = Lib.bytesToInt(buf, 36);

        if (numRelocations != 0) {
            Lib.debug(dbgCoffSection, "\tsection needs relocation");
            throw new EOFException();
        }

        switch (flags & 0x0FFF) {
        case 0x0020:
            executable = true;
            readOnly = true;
            initialized = true;
            break;
        case 0x0040:
            executable = false;
            readOnly = false;
            initialized = true;
            break;
        case 0x0080:
            executable = false;
            readOnly = false;
            initialized = false;
            break;
        case 0x0100:
            executable = false;
            readOnly = true;
            initialized = true;
            break;
        default:
            Lib.debug(dbgCoffSection, "\tinvalid section flags: " + flags);
            throw new EOFException();
        }

        if (vaddr%Processor.pageSize != 0 || size < 0 ||
            initialized && (contentOffset < 0 ||
                            contentOffset+size > file.length())) {
            Lib.debug(dbgCoffSection, "\tinvalid section addresses: " +
                      "vaddr=" + vaddr + " size=" + size +
                      " contentOffset=" + contentOffset);
            throw new EOFException();
        }

        numPages = Lib.divRoundUp(size, Processor.pageSize);
        firstVPN = vaddr / Processor.pageSize;
    }

    /**
     * Return the COFF object used to load this executable instance.
     *
     * @return  the COFF object corresponding to this section.
     */
    public Coff getCoff() {
        return coff;
    }

    /**
     * Return the name of this section.
     *
     * @return  the name of this section.
     */
    public String getName() {
        return name;
    }
```

```
    /**
     * Test whether this section is read-only.
     *
     * @return  <tt>true</tt> if this section should never be written.
     */
    public boolean isReadOnly() {
        return readOnly;
    }

    /**
     * Test whether this section is initialized. Loading a page from an
     * initialized section requires a disk access, while loading a page from an
     * uninitialized section requires only zero-filling the page.
     *
     * @return  <tt>true</tt> if this section contains initialized data in the
     *          executable.
     */
    public boolean isInitialzed() {
        return initialized;
    }

    /**
     * Return the length of this section in pages.
     *
     * @return  the number of pages in this section.
     */
    public int getLength() {
        return numPages;
    }

    /**
     * Return the first virtual page number used by this section.
     *
     * @return  the first virtual page number used by this section.
     */
    public int getFirstVPN() {
        return firstVPN;
    }

    /**
     * Load a page from this segment into physical memory.
     *
     * @param   spn     the page number within this segment.
     * @param   ppn     the physical page to load into.
     */
    public void loadPage(int spn, int ppn) {
        Lib.assertTrue(file != null);

        Lib.assertTrue(spn>=0 && spn<numPages);
        Lib.assertTrue(ppn>=0 && ppn<Machine.processor().getNumPhysPages());

        int pageSize = Processor.pageSize;
        byte[] memory = Machine.processor().getMemory();
        int paddr = ppn*pageSize;
        int faddr = contentOffset + spn*pageSize;
        int initlen;

        if (!initialized)
            initlen = 0;
        else if (spn == numPages-1)
            initlen = size % pageSize;
        else
            initlen = pageSize;

        if (initlen > 0)
            Lib.strictReadFile(file, faddr, memory, paddr, initlen);

        Arrays.fill(memory, paddr+initlen, paddr+pageSize, (byte) 0);
    }

    /** The COFF object to which this section belongs. */
    protected Coff coff;
    /** The COFF name of this section. */
    protected String name;
    /** True if this section contains code. */
    protected boolean executable;
    /** True if this section is read-only. */
    protected boolean readOnly;
    /** True if this section contains initialized data. */
    protected boolean initialized;

    /** The number of virtual pages in this section. */
    protected int numPages;
    /** The first virtual page number used by this section. */
    protected int firstVPN;

    private OpenFile file;
    private int contentOffset, size;

    /** The length of a COFF section header. */
    public static final int headerLength = 40;

    private static final char dbgCoffSection = 'c';
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import java.util.HashMap;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
import java.io.StreamTokenizer;

/**
 * Provides routines to access the Nachos configuration.
 */
public final class Config {
    /**
     * Load configuration information from the specified file. Must be called
     * before the Nachos security manager is installed.
     *
     * @param   fileName        the name of the file containing the
     *                          configuration to use.
     */
    public static void load(String fileName) {
        System.out.print(" config");

        Lib.assertTrue(!loaded);
        loaded = true;

        configFile = fileName;

        try {
            config = new HashMap<String, String>();

            File file = new File(configFile);
            Reader reader = new FileReader(file);
            StreamTokenizer s = new StreamTokenizer(reader);

            s.resetSyntax();
            s.whitespaceChars(0x00, 0x20);
            s.wordChars(0x21, 0xFF);
            s.eolIsSignificant(true);
            s.commentChar('#');
            s.quoteChar('"');

            int line = 1;

            s.nextToken();

            while (true) {
                if (s.ttype == StreamTokenizer.TT_EOF)
                    break;

                if (s.ttype == StreamTokenizer.TT_EOL) {
                    line++;
                    s.nextToken();
                    continue;
                }

                if (s.ttype != StreamTokenizer.TT_WORD)
                    loadError(line);

                String key = s.sval;

                if (s.nextToken() != StreamTokenizer.TT_WORD ||
                    !s.sval.equals("="))
                    loadError(line);

                if (s.nextToken() != StreamTokenizer.TT_WORD && s.ttype != '"')
                    loadError(line);

                String value = s.sval;

                // ignore everything after first string
                while (s.nextToken() != StreamTokenizer.TT_EOL &&
                       s.ttype != StreamTokenizer.TT_EOF);

                if (config.get(key) != null)
                    loadError(line);

                config.put(key, value);
                line++;
            }
        }
        catch (Throwable e) {
            System.err.println("Error loading " + configFile);
            System.exit(1);
        }
    }

    private static void loadError(int line) {
        System.err.println("Error in " + configFile + " line " + line);
        System.exit(1);
    }

    private static void configError(String message) {
        System.err.println("");
        System.err.println("Error in " + configFile + ": " + message);
        System.exit(1);
    }

    /**
     * Get the value of a key in <tt>nachos.conf</tt>.
     *
     * @param   key     the key to look up.
     * @return  the value of the specified key, or <tt>null</tt> if it is not
     *          present.
     */
    public static String getString(String key) {
        return (String) config.get(key);
    }

    /**
     * Get the value of a key in <tt>nachos.conf</tt>, returning the specified
     * default if the key does not exist.
     *
     * @param   key             the key to look up.
     * @param   defaultValue    the value to return if the key does not exist.
     * @return  the value of the specified key, or <tt>defaultValue</tt> if it
     *          is not present.
     */
    public static String getString(String key, String defaultValue) {
        String result = getString(key);

        if (result == null)
            return defaultValue;

        return result;
    }

    private static Integer requestInteger(String key) {
        try {
```

```
        String value = getString(key);
        if (value == null)
            return null;

        return new Integer(value);
    }
    catch (NumberFormatException e) {
        configError(key + " should be an integer");

        Lib.assertNotReached();
        return null;
    }
}

/**
 * Get the value of an integer key in <tt>nachos.conf</tt>.
 *
 * @param   key     the key to look up.
 * @return  the value of the specified key.
 */
public static int getInteger(String key) {
    Integer result = requestInteger(key);

    if (result == null)
        configError("missing int " + key);

    return result.intValue();
}

/**
 * Get the value of an integer key in <tt>nachos.conf</tt>, returning the
 * specified default if the key does not exist.
 *
 * @param   key             the key to look up.
 * @param   defaultValue    the value to return if the key does not exist.
 * @return  the value of the specified key, or <tt>defaultValue</tt> if the
 *          key does not exist.
 */
public static int getInteger(String key, int defaultValue) {
    Integer result = requestInteger(key);

    if (result == null)
        return defaultValue;

    return result.intValue();
}

private static Double requestDouble(String key) {
    try {
        String value = getString(key);
        if (value == null)
            return null;

        return new Double(value);
    }
    catch (NumberFormatException e) {
        configError(key + " should be a double");

        Lib.assertNotReached();
        return null;
    }
}

/**
 * Get the value of a double key in <tt>nachos.conf</tt>.
 *
 * @param   key     the key to look up.
 * @return  the value of the specified key.
 */
public static double getDouble(String key) {
    Double result = requestDouble(key);

    if (result == null)
        configError("missing double " + key);

    return result.doubleValue();
}

/**
 * Get the value of a double key in <tt>nachos.conf</tt>, returning the
 * specified default if the key does not exist.
 *
 * @param   key             the key to look up.
 * @param   defaultValue    the value to return if the key does not exist.
 * @return  the value of the specified key, or <tt>defaultValue</tt> if the
 *          key does not exist.
 */
public static double getDouble(String key, double defaultValue) {
    Double result = requestDouble(key);

    if (result == null)
        return defaultValue;

    return result.doubleValue();
}

private static Boolean requestBoolean(String key) {
    String value = getString(key);

    if (value == null)
        return null;

    if (value.equals("1") || value.toLowerCase().equals("true")) {
        return Boolean.TRUE;
    }
    else if (value.equals("0") || value.toLowerCase().equals("false")) {
        return Boolean.FALSE;
    }
    else {
        configError(key + " should be a boolean");

        Lib.assertNotReached();
        return null;
    }
}

/**
 * Get the value of a boolean key in <tt>nachos.conf</tt>.
 *
 * @param   key     the key to look up.
 * @return  the value of the specified key.
 */
public static boolean getBoolean(String key) {
    Boolean result = requestBoolean(key);

    if (result == null)
        configError("missing boolean " + key);

    return result.booleanValue();
}
```

```
    /**
     * Get the value of a boolean key in <tt>nachos.conf</tt>, returning the
     * specified default if the key does not exist.
     *
     * @param   key             the key to look up.
     * @param   defaultValue    the value to return if the key does not exist.
     * @return  the value of the specified key, or <tt>defaultValue</tt> if the
     *          key does not exist.
     */
    public static boolean getBoolean(String key, boolean defaultValue) {
        Boolean result = requestBoolean(key);

        if (result == null)
            return defaultValue;

        return result.booleanValue();
    }

    private static boolean loaded = false;
    private static String configFile;
    private static HashMap<String, String> config;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;
import nachos.threads.KThread;
import nachos.threads.Semaphore;

import java.util.Vector;
import java.util.LinkedList;
import java.util.Iterator;

/**
 * A bank of elevators.
 */
public final class ElevatorBank implements Runnable {
    /** Indicates an elevator intends to move down. */
    public static final int dirDown = -1;
    /** Indicates an elevator intends not to move. */
    public static final int dirNeither = 0;
    /** Indicates an elevator intends to move up. */
    public static final int dirUp = 1;

    /**
     * Allocate a new elevator bank.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public ElevatorBank(Privilege privilege) {
        System.out.print(" elevators");

        this.privilege = privilege;

        simulationStarted = false;
    }

    /**
     * Initialize this elevator bank with the specified number of elevators and
     * the specified number of floors. The software elevator controller must
     * also be specified. This elevator must not already be running a
     * simulation.
     *
     * @param   numElevators    the number of elevators in the bank.
     * @param   numFloors       the number of floors in the bank.
     * @param   controller      the elevator controller.
     */
    public void init(int numElevators, int numFloors,
                     ElevatorControllerInterface controller) {
        Lib.assertTrue(!simulationStarted);

        this.numElevators = numElevators;
        this.numFloors = numFloors;

        manager = new ElevatorManager(controller);

        elevators = new ElevatorState[numElevators];
        for (int i=0; i<numElevators; i++)
            elevators[i] = new ElevatorState(0);

        numRiders = 0;
        ridersVector = new Vector<RiderControls>();

        enableGui = false;
        gui = null;
    }

    /**
     * Add a rider to the simulation. This method must not be called after
     * <tt>run()</tt> is called.
     *
     * @param   rider   the rider to add.
     * @param   floor   the floor the rider will start on.
     * @param   stops   the array to pass to the rider's <tt>initialize()</tt>
     *                  method.
     * @return  the controls that will be given to the rider.
     */
    public RiderControls addRider(RiderInterface rider,
                                  int floor, int[] stops) {
        Lib.assertTrue(!simulationStarted);

        RiderControls controls = new RiderState(rider, floor, stops);
        ridersVector.addElement(controls);
        numRiders++;
        return controls;
    }

    /**
     * Create a GUI for this elevator bank.
     */
    public void enableGui() {
        Lib.assertTrue(!simulationStarted);
        Lib.assertTrue(Config.getBoolean("ElevatorBank.allowElevatorGUI"));

        enableGui = true;
    }

    /**
     * Run a simulation. Initialize all elevators and riders, and then
     * fork threads to each of their <tt>run()</tt> methods. Return when the
     * simulation is finished.
     */
    public void run() {
        Lib.assertTrue(!simulationStarted);
        simulationStarted = true;

        riders = new RiderState[numRiders];
        ridersVector.toArray(riders);

        if (enableGui) {
            privilege.doPrivileged(new Runnable() {
                public void run() { initGui(); }
            });
        }

        for (int i=0; i<numRiders; i++)
            riders[i].initialize();
        manager.initialize();

        for (int i=0; i<numRiders; i++)
            riders[i].run();
        manager.run();

        for (int i=0; i<numRiders; i++)
            riders[i].join();
        manager.join();

        simulationStarted = false;
    }
```

```
    private void initGui() {
        int[] numRidersPerFloor = new int[numFloors];
        for (int floor=0; floor<numFloors; floor++)
            numRidersPerFloor[floor] = 0;

        for (int rider=0; rider<numRiders; rider++)
            numRidersPerFloor[riders[rider].floor]++;

        gui = new ElevatorGui(numFloors, numElevators, numRidersPerFloor);
    }

    /**
     * Tests whether this module is working.
     */
    public static void selfTest() {
        new ElevatorTest().run();
    }

    void postRiderEvent(int event, int floor, int elevator) {
        int direction = dirNeither;
        if (elevator != -1) {
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);
            direction = elevators[elevator].direction;
        }

        RiderEvent e = new RiderEvent(event, floor, elevator, direction);
        for (int i=0; i<numRiders; i++) {
            RiderState rider = riders[i];
            if ((rider.inElevator && rider.elevator == e.elevator) ||
                (!rider.inElevator && rider.floor == e.floor)) {
                rider.events.add(e);
                rider.schedule(1);
            }
        }
    }

    private class ElevatorManager implements ElevatorControls {
        ElevatorManager(ElevatorControllerInterface controller) {
            this.controller = controller;

            interrupt = new Runnable() { public void run() { interrupt(); }};
        }

        public int getNumFloors() {
            return numFloors;
        }

        public int getNumElevators() {
            return numElevators;
        }

        public void setInterruptHandler(Runnable handler) {
            this.handler = handler;
        }

        public void openDoors(int elevator) {
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);
            postRiderEvent(RiderEvent.eventDoorsOpened,
                           elevators[elevator].openDoors(), elevator);

            if (gui != null) {
                if (elevators[elevator].direction == dirUp)
                    gui.clearUpButton(elevators[elevator].floor);
                else if (elevators[elevator].direction == dirDown)
                    gui.clearDownButton(elevators[elevator].floor);

                gui.openDoors(elevator);
            }
        }

        public void closeDoors(int elevator) {
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);
            postRiderEvent(RiderEvent.eventDoorsClosed,
                           elevators[elevator].closeDoors(), elevator);

            if (gui != null)
                gui.closeDoors(elevator);
        }

        public boolean moveTo(int floor, int elevator) {
            Lib.assertTrue(floor >= 0 && floor < numFloors);
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);

            if (!elevators[elevator].moveTo(floor))
                return false;

            schedule(Stats.ElevatorTicks);
            return true;
        }

        public int getFloor(int elevator) {
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);
            return elevators[elevator].floor;
        }

        public void setDirectionDisplay(int elevator, int direction) {
            Lib.assertTrue(elevator >= 0 && elevator < numElevators);
            elevators[elevator].direction = direction;

            if (elevators[elevator].doorsOpen) {
                postRiderEvent(RiderEvent.eventDirectionChanged,
                               elevators[elevator].floor, elevator);
            }

            if (gui != null) {
                if (elevators[elevator].doorsOpen) {
                    if (direction == dirUp)
                        gui.clearUpButton(elevators[elevator].floor);
                    else if (direction == dirDown)
                        gui.clearDownButton(elevators[elevator].floor);
                }

                gui.setDirectionDisplay(elevator, direction);
            }
        }

        public void finish() {
            finished = true;

            Lib.assertTrue(KThread.currentThread() == thread);

            done.V();
            KThread.finish();
        }

        public ElevatorEvent getNextEvent() {
            if (events.isEmpty())
                return null;
            else
                return (ElevatorEvent) events.removeFirst();
```

```
    }

    void schedule(int when) {
        privilege.interrupt.schedule(when, "elevator", interrupt);
    }

    void postEvent(int event, int floor, int elevator, boolean schedule) {
        events.add(new ElevatorEvent(event, floor, elevator));

        if (schedule)
            schedule(1);
    }

    void interrupt() {
        for (int i=0; i<numElevators; i++) {
            if (elevators[i].atNextFloor()) {
                if (gui != null)
                    gui.elevatorMoved(elevators[i].floor, i);

                if (elevators[i].atDestination()) {
                    postEvent(ElevatorEvent.eventElevatorArrived,
                              elevators[i].destination, i, false);
                }
                else {
                    elevators[i].nextETA += Stats.ElevatorTicks;
                    privilege.interrupt.schedule(Stats.ElevatorTicks,
                                                 "elevator",
                                                 interrupt);
                }
            }
        }

        if (!finished && !events.isEmpty() && handler != null)
            handler.run();
    }

    void initialize() {
        controller.initialize(this);
    }

    void run() {
        thread = new KThread(controller);
        thread.setName("elevator controller");
        thread.fork();
    }

    void join() {
        postEvent(ElevatorEvent.eventRidersDone, -1, -1, true);
        done.P();
    }

    ElevatorControllerInterface controller;
    Runnable interrupt;
    KThread thread;

    Runnable handler = null;
    LinkedList<ElevatorEvent> events = new LinkedList<ElevatorEvent>();
    Semaphore done = new Semaphore(0);
    boolean finished = false;
}

private class ElevatorState {
    ElevatorState(int floor) {
        this.floor = floor;
        destination = floor;
    }

    int openDoors() {
        Lib.assertTrue(!doorsOpen && !moving);
        doorsOpen = true;
        return floor;
    }

    int closeDoors() {
        Lib.assertTrue(doorsOpen);
        doorsOpen = false;
        return floor;
    }

    boolean moveTo(int newDestination) {
        Lib.assertTrue(!doorsOpen);

        if (!moving) {
            // can't move to current floor
            if (floor == newDestination)
                return false;

            destination = newDestination;
            nextETA = Machine.timer().getTime() + Stats.ElevatorTicks;

            moving = true;
            return true;
        }
        else {
            // moving, shouldn't be at destination
            Lib.assertTrue(floor != destination);

            // make sure it's ok to stop
            if ((destination > floor && newDestination <= floor) ||
                (destination < floor && newDestination >= floor))
                return false;

            destination = newDestination;
            return true;
        }
    }

    boolean enter(RiderState rider, int onFloor) {
        Lib.assertTrue(!riders.contains(rider));

        if (!doorsOpen || moving || onFloor != floor ||
            riders.size() == maxRiders)
            return false;

        riders.addElement(rider);
        return true;
    }

    boolean exit(RiderState rider, int onFloor) {
        Lib.assertTrue(riders.contains(rider));

        if (!doorsOpen || moving || onFloor != floor)
            return false;

        riders.removeElement(rider);
        return true;
    }

    boolean atNextFloor() {
        if (!moving || Machine.timer().getTime() < nextETA)
```

```
            return false;

        Lib.assertTrue(destination != floor);
        if (destination > floor)
            floor++;
        else
            floor--;

        for (Iterator i=riders.iterator(); i.hasNext(); ) {
            RiderState rider = (RiderState) i.next();

            rider.floor = floor;
        }

        return true;
    }

    boolean atDestination() {
        if (!moving || destination != floor)
            return false;

        moving = false;
        return true;
    }

    static final int maxRiders = 4;

    int floor, destination;
    long nextETA;

    boolean doorsOpen = false, moving = false;
    int direction = dirNeither;
    public Vector<RiderState> riders = new Vector<RiderState>();
}

private class RiderState implements RiderControls {
    RiderState(RiderInterface rider, int floor, int[] stops) {
        this.rider = rider;
        this.floor = floor;
        this.stops = stops;

        interrupt = new Runnable() { public void run() { interrupt(); }};
    }

    public int getNumFloors() {
        return numFloors;
    }

    public int getNumElevators() {
        return numElevators;
    }

    public void setInterruptHandler(Runnable handler) {
        this.handler = handler;
    }

    public int getFloor() {
        return floor;
    }

    public int[] getFloors() {
        int[] array = new int[floors.size()];
        for (int i=0; i<array.length; i++)
            array[i] = ((Integer) floors.elementAt(i)).intValue();
        return array;
    }

    public int getDirectionDisplay(int elevator) {
        Lib.assertTrue(elevator >= 0 && elevator < numElevators);
        return elevators[elevator].direction;
    }

    public RiderEvent getNextEvent() {
        if (events.isEmpty())
            return null;
        else
            return (RiderEvent) events.removeFirst();
    }

    public boolean pressDirectionButton(boolean up) {
        if (up)
            return pressUpButton();
        else
            return pressDownButton();
    }

    public boolean pressUpButton() {
        Lib.assertTrue(!inElevator && floor < numFloors-1);

        for (int elevator=0; elevator<numElevators; elevator++) {
            if (elevators[elevator].doorsOpen &&
                elevators[elevator].direction == ElevatorBank.dirUp &&
                elevators[elevator].floor == floor)
                return false;
        }

        manager.postEvent(ElevatorEvent.eventUpButtonPressed,
                          floor, -1, true);

        if (gui != null)
            gui.pressUpButton(floor);

        return true;
    }

    public boolean pressDownButton() {
        Lib.assertTrue(!inElevator && floor > 0);

        for (int elevator=0; elevator<numElevators; elevator++) {
            if (elevators[elevator].doorsOpen &&
                elevators[elevator].direction == ElevatorBank.dirDown &&
                elevators[elevator].floor == floor)
                return false;
        }

        manager.postEvent(ElevatorEvent.eventDownButtonPressed,
                          floor, -1, true);

        if (gui != null)
            gui.pressDownButton(floor);

        return true;
    }

    public boolean enterElevator(int elevator) {
        Lib.assertTrue(!inElevator &&
                       elevator >= 0 && elevator < numElevators);
        if (!elevators[elevator].enter(this, floor))
            return false;
```

```
        if (gui != null)
            gui.enterElevator(floor, elevator);

        inElevator = true;
        this.elevator = elevator;
        return true;
    }

    public boolean pressFloorButton(int floor) {
        Lib.assertTrue(inElevator && floor >= 0 && floor < numFloors);

        if (elevators[elevator].doorsOpen &&
            elevators[elevator].floor == floor)
            return false;

        manager.postEvent(ElevatorEvent.eventFloorButtonPressed,
                          floor, elevator, true);

        if (gui != null)
            gui.pressFloorButton(floor, elevator);

        return true;
    }

    public boolean exitElevator(int floor) {
        Lib.assertTrue(inElevator && floor >= 0 && floor < numFloors);

        if (!elevators[elevator].exit(this, floor))
            return false;

        inElevator = false;
        floors.add(new Integer(floor));

        if (gui != null)
            gui.exitElevator(floor, elevator);

        return true;
    }

    public void finish() {
        finished = true;

        int[] floors = getFloors();
        Lib.assertTrue(floors.length == stops.length);
        for (int i=0; i<floors.length; i++)
            Lib.assertTrue(floors[i] == stops[i]);

        Lib.assertTrue(KThread.currentThread() == thread);

        done.V();
        KThread.finish();
    }

    void schedule(int when) {
        privilege.interrupt.schedule(when, "rider", interrupt);
    }

    void interrupt() {
        if (!finished && !events.isEmpty() && handler != null)
            handler.run();
    }

    void initialize() {
        rider.initialize(this, stops);
        }

        void run() {
            thread = new KThread(rider);
            thread.setName("rider");
            thread.fork();
        }

        void join() {
            done.P();
        }

        RiderInterface rider;
        boolean inElevator = false, finished = false;
        int floor, elevator;
        int[] stops;
        Runnable interrupt, handler = null;
        LinkedList<RiderEvent> events = new LinkedList<RiderEvent>();
        Vector<Integer> floors = new Vector<Integer>();
        Semaphore done = new Semaphore(0);
        KThread thread;
    }

    private int numFloors, numElevators;
    private ElevatorManager manager;
    private ElevatorState[] elevators;

    private int numRiders;
    private Vector<RiderControls> ridersVector;
    private RiderState[] riders;

    private boolean simulationStarted, enableGui;
    private Privilege privilege;
    private ElevatorGui gui;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A controller for all the elevators in an elevator bank. The controller
 * accesses the elevator bank through an instance of <tt>ElevatorControls</tt>.
 */
public interface ElevatorControllerInterface extends Runnable {
    /**
     * Initialize this elevator controller. The controller will access the
     * elevator bank through <i>controls</i>. This constructor should return
     * immediately after this controller is initialized, but not until the
     * interupt handler is set. The controller will start receiving events
     * after this method returns, but potentially before <tt>run()</tt> is
     * called.
     *
     * @param   controls        the controller's interface to the elevator
     *                          bank. The controller must not attempt to access
     *                          the elevator bank in <i>any</i> other way.
     */
    public void initialize(ElevatorControls controls);

    /**
     * Cause the controller to use the provided controls to receive and process
     * requests from riders. This method should not return, but instead should
     * call <tt>controls.finish()</tt> when the controller is finished.
     */
    public void run();

    /** The number of ticks doors should be held open before closing them. */
    public static final int timeDoorsOpen = 500;

    /** Indicates an elevator intends to move down. */
    public static final int dirDown = -1;
    /** Indicates an elevator intends not to move. */
    public static final int dirNeither = 0;
    /** Indicates an elevator intends to move up. */
    public static final int dirUp = 1;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A set of controls that can be used by an elevator controller.
 */
public interface ElevatorControls {
    /**
     * Return the number of floors in the elevator bank. If <i>n</i> is the
     * number of floors in the bank, then the floors are numbered <i>0</i>
     * (the ground floor) through <i>n - 1</i> (the top floor).
     *
     * @return  the number of floors in the bank.
     */
    public int getNumFloors();

    /**
     * Return the number of elevators in the elevator bank. If <i>n</i> is the
     * number of elevators in the bank, then the elevators are numbered
     * <i>0</i> through <i>n - 1</i>.
     *
     * @return  the numbe rof elevators in the bank.
     */
    public int getNumElevators();

    /**
     * Set the elevator interrupt handler. This handler will be called when an
     * elevator event occurs, and when all the riders have reaced their
     * destinations.
     *
     * @param   handler the elevator interrupt handler.
     */
    public void setInterruptHandler(Runnable handler);

    /**
     * Open an elevator's doors.
     *
     * @param   elevator        which elevator's doors to open.
     */
    public void openDoors(int elevator);

    /**
     * Close an elevator's doors.
     *
     * @param   elevator        which elevator's doors to close.
     */
    public void closeDoors(int elevator);

    /**
     * Move an elevator to another floor. The elevator's doors must be closed.
     * If the elevator is already moving and cannot safely stop at the
     * specified floor because it has already passed or is about to pass the
     * floor, fails and returns <tt>false</tt>. If the elevator is already
     * stopped at the specified floor, returns <tt>false</tt>.
     *
     * @param   floor           the floor to move to.
     * @param   elevator        the elevator to move.
     * @return  <tt>true</tt> if the elevator's destination was changed.
     */
    public boolean moveTo(int floor, int elevator);

    /**
     * Return the current location of the elevator. If the elevator is in
     * motion, the returned value will be within one of the exact location.
     *
     * @param   elevator        the elevator to locate.
     * @return  the floor the elevator is on.
     */
    public int getFloor(int elevator);

    /**
     * Set which direction the elevator bank will show for this elevator's
     * display. The <i>direction</i> argument should be one of the <i>dir*</i>
     * constants in the <tt>ElevatorBank</tt> class.
     *
     * @param   elevator        the elevator whose direction display to set.
     * @param   direction       the direction to show (up, down, or neither).
     */
    public void setDirectionDisplay(int elevator, int direction);

    /**
     * Call when the elevator controller is finished.
     */
    public void finish();

    /**
     * Return the next event in the event queue. Note that there may be
     * multiple events pending when an elevator interrupt occurs, so this
     * method should be called repeatedly until it returns <tt>null</tt>.
     *
     * @return  the next event, or <tt>null</tt> if no further events are
     * currently pending.
     */
    public ElevatorEvent getNextEvent();
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * An event that affects elevator software.
 */
public final class ElevatorEvent {
    public ElevatorEvent(int event, int floor, int elevator) {
        this.event = event;
        this.floor = floor;
        this.elevator = elevator;
    }

    /** The event identifier. Refer to the <i>event*</i> constants. */
    public final int event;
    /** The floor pertaining to the event, or -1 if not applicable. */
    public final int floor;
    /** The elevator pertaining to the event, or -1 if not applicable. */
    public final int elevator;

    /** An up button was pressed. */
    public static final int eventUpButtonPressed = 0;
    /** A down button was pressed. */
    public static final int eventDownButtonPressed = 1;
    /** A floor button was pressed inside an elevator. */
    public static final int eventFloorButtonPressed = 2;
    /** An elevator has arrived and stopped at its destination floor. */
    public static final int eventElevatorArrived = 3;
    /** All riders have finished; the elevator controller should terminate. */
    public static final int eventRidersDone = 4;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import java.awt.Frame;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.Panel;
import java.awt.ScrollPane;
import java.awt.Canvas;

import java.awt.GridLayout;
import java.awt.FlowLayout;
import java.awt.Insets;

import java.awt.Rectangle;
import java.awt.Graphics;
import java.awt.Color;

/**
 * A graphical visualization for the <tt>ElevatorBank</tt> class.
 */
public final class ElevatorGui extends Frame {
    private final static int w=90, h=75;

    private int numFloors, numElevators;

    private ElevatorShaft[] elevators;
    private Floor[] floors;

    private int totalWidth, totalHeight;

    ElevatorGui(int numFloors, int numElevators, int[] numRidersPerFloor) {
        this.numFloors = numFloors;
        this.numElevators = numElevators;

        totalWidth = w*(numElevators+1);
        totalHeight = h*numFloors;

        setTitle("Elevator Bank");

        Panel floorPanel = new Panel(new GridLayout(numFloors, 1, 0, 0));

        floors = new Floor[numFloors];
        for (int i=numFloors-1; i>=0; i--) {
            floors[i] = new Floor(i, numRidersPerFloor[i]);
            floorPanel.add(floors[i]);
        }

        Panel panel = new Panel(new GridLayout(1, numElevators+1, 0, 0));

        panel.add(floorPanel);

        elevators = new ElevatorShaft[numElevators];
        for (int i=0; i<numElevators; i++) {
            elevators[i] = new ElevatorShaft(i);
            panel.add(elevators[i]);
        }

        add(panel);
        pack();

        setVisible(true);

        repaint();
    }

    void openDoors(int elevator) {
        elevators[elevator].openDoors();
    }

    void closeDoors(int elevator) {
        elevators[elevator].closeDoors();
    }

    void setDirectionDisplay(int elevator, int direction) {
        elevators[elevator].setDirectionDisplay(direction);
    }

    void pressUpButton(int floor) {
        floors[floor].pressUpButton();
    }

    void clearUpButton(int floor) {
        floors[floor].clearUpButton();
    }

    void pressDownButton(int floor) {
        floors[floor].pressDownButton();
    }

    void clearDownButton(int floor) {
        floors[floor].clearDownButton();
    }

    void enterElevator(int floor, int elevator) {
        floors[floor].removeRider();
        elevators[elevator].addRider();
    }

    void pressFloorButton(int floor, int elevator) {
        elevators[elevator].pressFloorButton(floor);
    }

    void exitElevator(int floor, int elevator) {
        elevators[elevator].removeRider();
        floors[floor].addRider();
    }

    void elevatorMoved(int floor, int elevator) {
        elevators[elevator].elevatorMoved(floor);
    }

    private void paintRider(Graphics g, int x, int y, int r) {
        g.setColor(Color.yellow);

        g.fillOval(x-r, y-r, 2*r, 2*r);

        g.setColor(Color.black);

        g.fillOval(x-r/2, y-r/2, r/3, r/3);
        g.fillOval(x+r/4, y-r/2, r/3, r/3);

        g.drawArc(x-r/2, y-r/2, r, r, 210, 120);
    }

    private void paintRiders(Graphics g, int x, int y, int w, int h, int n) {
        int r = 8, t = 20;

        int xn = w/t;
```

```
    int yn = h/t;

    int x0 = x + (w-xn*t)/2 + t/2;
    int y0 = y + h - t/2;

    for (int j=0; j<yn; j++) {
        for (int i=0; i<xn; i++) {
            if (n-- > 0)
                paintRider(g, x0 + i*t, y0 - j*t, r);
        }
    }
}

private class Floor extends Canvas {
    int floor, numRiders;

    boolean upSet = false;
    boolean downSet = false;

    Floor(int floor, int numRiders) {
        this.floor = floor;
        this.numRiders = numRiders;

        setBackground(Color.black);
    }

    public Dimension getPreferredSize() {
        return new Dimension(w, h);
    }

    public Dimension getMinimumSize() {
        return getPreferredSize();
    }

    public Dimension getMaximumSize() {
        return getPreferredSize();
    }

    public void repaint() {
        super.repaint();

        if (TCB.isNachosThread()) {
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
            }
        }
    }

    void pressUpButton() {
        if (!upSet) {
            upSet = true;
            repaint();
        }
    }

    void pressDownButton() {
        if (!downSet) {
            downSet = true;
            repaint();
        }
    }

    void clearUpButton() {
        if (upSet) {
            upSet = false;
            repaint();
        }
    }

    void clearDownButton() {
        if (downSet) {
            downSet = false;
            repaint();
        }
    }

    void addRider() {
        numRiders++;

        repaint();
    }

    void removeRider() {
        numRiders--;

        repaint();
    }

    public void paint(Graphics g) {
        g.setColor(Color.lightGray);
        g.drawLine(0, 0, w, 0);

        paintRiders(g, 0, 5, 3*w/4, h-10, numRiders);

        paintButtons(g);
    }

    private void paintButtons(Graphics g) {
        int s = 3*w/4;

        int x1 = s+w/32;
        int x2 = w-w/32;
        int y1 = h/8;
        int y2 = h-h/8;

        g.setColor(Color.darkGray);
        g.drawRect(x1, y1, x2-x1, y2-y1);
        g.setColor(Color.lightGray);
        g.fillRect(x1+1, y1+1, x2-x1-2, y2-y1-2);

        int r = Math.min((x2-x1)/3, (y2-y1)/6);
        int xc = (x1+x2)/2;
        int yc1 = (y1+y2)/2 - (3*r/2);
        int yc2 = (y1+y2)/2 + (3*r/2);

        g.setColor(Color.red);

        if (floor < numFloors-1) {
            if (upSet)
                g.fillOval(xc-r, yc1-r, 2*r, 2*r);
            else
                g.drawOval(xc-r, yc1-r, 2*r, 2*r);
        }

        if (floor > 0) {
            if (downSet)
                g.fillOval(xc-r, yc2-r, 2*r, 2*r);
            else
```

```
                g.drawOval(xc-r, yc2-r, 2*r, 2*r);
            }
        }
    }

    private class ElevatorShaft extends Canvas {
        ElevatorShaft(int elevator) {
            this.elevator = elevator;

            floorsSet = new boolean[numFloors];
            for (int i=0; i<numFloors; i++)
                floorsSet[i] = false;

            setBackground(Color.black);
        }

        public Dimension getPreferredSize() {
            return new Dimension(w, h*numFloors);
        }

        public Dimension getMinimumSize() {
            return getPreferredSize();
        }

        public Dimension getMaximumSize() {
            return getPreferredSize();
        }

        private void repaintElevator() {
            repaint(s, h*(numFloors-1-Math.max(floor, prevFloor)), w-2*s, h*2);

            try {
                Thread.sleep(100);
            }
            catch (InterruptedException e) {
            }
        }

        void openDoors() {
            doorsOpen = true;

            repaintElevator();
        }

        void closeDoors() {
            doorsOpen = false;

            repaintElevator();
        }

        void setDirectionDisplay(int direction) {
            this.direction = direction;

            repaintElevator();
        }

        void pressFloorButton(int floor) {
            if (!floorsSet[floor]) {
                floorsSet[floor] = true;

                repaintElevator();
            }
        }

        void elevatorMoved(int floor) {
            prevFloor = this.floor;
            this.floor = floor;

            floorsSet[floor] = false;

            repaintElevator();
        }

        void addRider() {
            numRiders++;

            repaintElevator();
        }

        void removeRider() {
            numRiders--;

            repaintElevator();
        }

        public void paint(Graphics g) {
            g.setColor(Color.lightGray);

            if (g.hitClip(0, 0, s, h*numFloors)) {
                g.drawLine(0, 0, 0, h*numFloors);
                g.drawLine(s-1, 0, s-1, h*numFloors);
                for (int y=0; y<h*numFloors-s; y+=s)
                    g.drawLine(0, y, s-1, y+s-1);
            }

            if (g.hitClip(w-s, 0, s, h*numFloors)) {
                g.drawLine(w-s, 0, w-s, h*numFloors);
                g.drawLine(w-1, 0, w-1, h*numFloors);
                for (int y=0; y<h*numFloors-s; y+=s)
                    g.drawLine(w-s, y, w-1, y+s-1);
            }

            // rectangle containing direction display area
            Rectangle d = new Rectangle(s*3/2, h*(numFloors-1-floor),
                                        w-3*s, w/3-s);

            // unit of measurement in direction rect (12ux4u)
            int u = d.width/12;

            // draw elevator, fill riders
            Rectangle e = new Rectangle(d.x, d.y+d.height,
                                        d.width, h-d.height-u);
            g.drawRect(e.x, e.y, e.width, e.height);
            paintRiders(g, e.x, e.y, e.width, e.height, numRiders);

            g.setColor(Color.lightGray);

            // draw doors...
            if (doorsOpen) {
                g.drawLine(e.x+2*s, e.y, e.x+2*s, e.y+e.height);
                for (int y=0; y<e.height-2*s; y+=2*s)
                    g.drawLine(e.x, e.y+y, e.x+2*s, e.y+y+2*s);

                g.drawLine(e.x+e.width-2*s, e.y,
                           e.x+e.width-2*s, e.y+e.height);
                for (int y=0; y<e.height-2*s; y+=2*s)
                    g.drawLine(e.x+e.width-2*s, e.y+y, e.x+e.width, e.y+y+2*s);
            }
            else {
                for (int x=0; x<e.width; x+=2*s)
```

```
                g.drawLine(e.x+x, e.y, e.x+x, e.y+e.height);
            }

            g.setColor(Color.yellow);

            int[] xUp = { d.x + u*6, d.x + u*8, d.x + u*7 };
            int[] yUp = { d.y + u*3, d.y + u*3, d.y + u*1 };

            int[] xDown = { d.x + u*4, d.x + u*6, d.x + u*5 };
            int[] yDown = { d.y + u*1, d.y + u*1, d.y + u*3 };

            // draw arrows
            if (direction == ElevatorBank.dirUp)
                g.fillPolygon(xUp, yUp, 3);
            else
                g.drawPolygon(xUp, yUp, 3);

            if (direction == ElevatorBank.dirDown)
                g.fillPolygon(xDown, yDown, 3);
            else
                g.drawPolygon(xDown, yDown, 3);
        }

        private static final int s = 5;

        private boolean doorsOpen = false;
        private int floor = 0, prevFloor = 0, numRiders = 0;
        private int direction = ElevatorBank.dirNeither;

        private int elevator;

        private boolean floorsSet[];
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;
import nachos.threads.KThread;
import nachos.threads.Semaphore;

/**
 * Tests the <tt>ElevatorBank</tt> module, using a single elevator and a single
 * rider.
 */
public final class ElevatorTest {
    /**
     * Allocate a new <tt>ElevatorTest</tt> object.
     */
    public ElevatorTest() {
    }

    /**
     * Run a test on <tt>Machine.bank()</tt>.
     */
    public void run() {
        Machine.bank().init(1, 2, new ElevatorController());

        int[] stops = { 1 };

        Machine.bank().addRider(new Rider(), 0, stops);

        Machine.bank().run();
    }

    private class ElevatorController implements ElevatorControllerInterface {
        public void initialize(ElevatorControls controls) {
            this.controls = controls;

            eventWait = new Semaphore(0);

            controls.setInterruptHandler(new Runnable() {
                    public void run() { interrupt(); }
                });
        }

        public void run() {
            ElevatorEvent e;

            Lib.assertTrue(controls.getFloor(0) == 0);

            e = getNextEvent();
            Lib.assertTrue(e.event == ElevatorEvent.eventUpButtonPressed &&
                       e.floor == 0);

            controls.setDirectionDisplay(0, dirUp);
            controls.openDoors(0);

            e = getNextEvent();
            Lib.assertTrue(e.event == ElevatorEvent.eventFloorButtonPressed &&
                       e.floor == 1);

            controls.closeDoors(0);
            controls.moveTo(1, 0);

            e = getNextEvent();
            Lib.assertTrue(e.event == ElevatorEvent.eventElevatorArrived &&
                       e.floor == 1 &&
                       e.elevator == 0);

            controls.openDoors(0);

            e = getNextEvent();
            Lib.assertTrue(e.event == ElevatorEvent.eventRidersDone);

            controls.finish();
            Lib.assertNotReached();
        }

        private void interrupt() {
            eventWait.V();
        }

        private ElevatorEvent getNextEvent() {
            ElevatorEvent event;
            while (true) {
                if ((event = controls.getNextEvent()) != null)
                    break;

                eventWait.P();
            }
            return event;
        }

        private ElevatorControls controls;
        private Semaphore eventWait;
    }

    private class Rider implements RiderInterface {
        public void initialize(RiderControls controls, int[] stops) {
            this.controls = controls;
            Lib.assertTrue(stops.length == 1 && stops[0] == 1);

            eventWait = new Semaphore(0);

            controls.setInterruptHandler(new Runnable() {
                    public void run() { interrupt(); }
                });
        }

        public void run() {
            RiderEvent e;

            Lib.assertTrue(controls.getFloor() == 0);

            controls.pressUpButton();

            e = getNextEvent();
            Lib.assertTrue(e.event == RiderEvent.eventDoorsOpened &&
                       e.floor == 0 &&
                       e.elevator == 0);
            Lib.assertTrue(controls.getDirectionDisplay(0) == dirUp);

            Lib.assertTrue(controls.enterElevator(0));
            controls.pressFloorButton(1);

            e = getNextEvent();
            Lib.assertTrue(e.event == RiderEvent.eventDoorsClosed &&
                       e.floor == 0 &&
                       e.elevator == 0);

            e = getNextEvent();
            Lib.assertTrue(e.event == RiderEvent.eventDoorsOpened &&
```

```
                    e.floor == 1 &&
                    e.elevator == 0);

            Lib.assertTrue(controls.exitElevator(1));

            controls.finish();
            Lib.assertNotReached();
        }

        private void interrupt() {
            eventWait.V();
        }

        private RiderEvent getNextEvent() {
            RiderEvent event;
            while (true) {
                if ((event = controls.getNextEvent()) != null)
                    break;

                eventWait.P();
            }
            return event;
        }

        private RiderControls controls;
        private Semaphore eventWait;
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A file system that allows the user to create, open, and delete files.
 */
public interface FileSystem {
    /**
     * Atomically open a file, optionally creating it if it does not
     * already exist. If the file does not
     * already exist and <tt>create</tt> is <tt>false</tt>, returns
     * <tt>null</tt>. If the file does not already exist and <tt>create</tt>
     * is <tt>true</tt>, creates the file with zero length. If the file already
     * exists, opens the file without changing it in any way.
     *
     * @param   name            the name of the file to open.
     * @param   create          <tt>true</tt> to create the file if it does not
     *                          already exist.
     * @return  an <tt>OpenFile</tt> representing a new instance of the opened
     *          file, or <tt>null</tt> if the file could not be opened.
     */
    public OpenFile open(String name, boolean create);

    /**
     * Atomically remove an existing file. After a file is removed, it cannot
     * be opened until it is created again with <tt>open</tt>. If the file is
     * already open, it is up to the implementation to decide whether the file
     * can still be accessed or if it is deleted immediately.
     *
     * @param   name    the name of the file to remove.
     * @return  <tt>true</tt> if the file was successfully removed.
     */
    public boolean remove(String name);
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

import java.util.TreeSet;
import java.util.Iterator;
import java.util.SortedSet;

/**
 * The <tt>Interrupt</tt> class emulates low-level interrupt hardware. The
 * hardware provides a method (<tt>setStatus()</tt>) to enable or disable
 * interrupts.
 *
 * <p>
 * In order to emulate the hardware, we need to keep track of all pending
 * interrupts the hardware devices would cause, and when they are supposed to
 * occur.
 *
 * <p>
 * This module also keeps track of simulated time. Time advances only when the
 * following occur:
 * <ul>
 * <li>interrupts are enabled, when they were previously disabled
 * <li>a MIPS instruction is executed
 * </ul>
 *
 * <p>
 * As a result, unlike real hardware, interrupts (including time-slice context
 * switches) cannot occur just anywhere in the code where interrupts are
 * enabled, but rather only at those places in the code where simulated time
 * advances (so that it becomes time for the hardware simulation to invoke an
 * interrupt handler).
 *
 * <p>
 * This means that incorrectly synchronized code may work fine on this hardware
 * simulation (even with randomized time slices), but it wouldn't work on real
 * hardware. But even though Nachos can't always detect when your program
 * would fail in real life, you should still write properly synchronized code.
 */
public final class Interrupt {
    /**
     * Allocate a new interrupt controller.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public Interrupt(Privilege privilege) {
        System.out.print(" interrupt");

        this.privilege = privilege;
        privilege.interrupt = new InterruptPrivilege();

        enabled = false;
        pending = new TreeSet<PendingInterrupt>();
    }

    /**
     * Enable interrupts. This method has the same effect as
     * <tt>setStatus(true)</tt>.
     */
    public void enable() {
        setStatus(true);
    }

    /**
     * Disable interrupts and return the old interrupt state. This method has
     * the same effect as <tt>setStatus(false)</tt>.
     *
     * @return  <tt>true</tt> if interrupts were enabled.
     */
    public boolean disable() {
        return setStatus(false);
    }

    /**
     * Restore interrupts to the specified status. This method has the same
     * effect as <tt>setStatus(<i>status</i>)</tt>.
     *
     * @param   status  <tt>true</tt> to enable interrupts.
     */
    public void restore(boolean status) {
        setStatus(status);
    }

    /**
     * Set the interrupt status to be enabled (<tt>true</tt>) or disabled
     * (<tt>false</tt>) and return the previous status. If the interrupt
     * status changes from disabled to enabled, the simulated time is advanced.
     *
     * @param   status          <tt>true</tt> to enable interrupts.
     * @return                  <tt>true</tt> if interrupts were enabled.
     */
    public boolean setStatus(boolean status) {
        boolean oldStatus = enabled;
        enabled = status;

        if (oldStatus == false && status == true)
            tick(true);

        return oldStatus;
    }

    /**
     * Tests whether interrupts are enabled.
     *
     * @return  <tt>true</tt> if interrupts are enabled.
     */
    public boolean enabled() {
        return enabled;
    }

    /**
     * Tests whether interrupts are disabled.
     *
     * @return <tt>true</tt> if interrupts are disabled.
     */
    public boolean disabled() {
        return !enabled;
    }

    private void schedule(long when, String type, Runnable handler) {
        Lib.assertTrue(when>0);

        long time = privilege.stats.totalTicks + when;
        PendingInterrupt toOccur = new PendingInterrupt(time, type, handler);

        Lib.debug(dbgInt,
                  "Scheduling the " + type +
```

```
                " interrupt handler at time = " + time);

        pending.add(toOccur);
    }

    private void tick(boolean inKernelMode) {
        Stats stats = privilege.stats;

        if (inKernelMode) {
            stats.kernelTicks += Stats.KernelTick;
            stats.totalTicks += Stats.KernelTick;
        }
        else {
            stats.userTicks += Stats.UserTick;
            stats.totalTicks += Stats.UserTick;
        }

        if (Lib.test(dbgInt))
            System.out.println("== Tick " + stats.totalTicks + " ==");

        enabled = false;
        checkIfDue();
        enabled = true;
    }

    private void checkIfDue() {
        long time = privilege.stats.totalTicks;

        Lib.assertTrue(disabled());

        if (Lib.test(dbgInt))
            print();

        if (pending.isEmpty())
            return;

        if (((PendingInterrupt) pending.first()).time > time)
            return;

        Lib.debug(dbgInt, "Invoking interrupt handlers at time = " + time);

        while (!pending.isEmpty() &&
               ((PendingInterrupt) pending.first()).time <= time) {
            PendingInterrupt next = (PendingInterrupt) pending.first();
            pending.remove(next);

            Lib.assertTrue(next.time <= time);

            if (privilege.processor != null)
                privilege.processor.flushPipe();

            Lib.debug(dbgInt, "  " + next.type);

            next.handler.run();
        }

        Lib.debug(dbgInt, "  (end of list)");
    }

    private void print() {
        System.out.println("Time: " + privilege.stats.totalTicks
                           + ", interrupts " + (enabled ? "on" : "off"));
        System.out.println("Pending interrupts:");

        for (Iterator i=pending.iterator(); i.hasNext(); ) {
            PendingInterrupt toOccur = (PendingInterrupt) i.next();
            System.out.println("  " + toOccur.type +
                               ", scheduled at " + toOccur.time);
        }

        System.out.println("  (end of list)");
    }

    private class PendingInterrupt implements Comparable {
        PendingInterrupt(long time, String type, Runnable handler) {
            this.time = time;
            this.type = type;
            this.handler = handler;
            this.id = numPendingInterruptsCreated++;
        }

        public int compareTo(Object o) {
            PendingInterrupt toOccur = (PendingInterrupt) o;

            // can't return 0 for unequal objects, so check all fields
            if (time < toOccur.time)
                return -1;
            else if (time > toOccur.time)
                return 1;
            else if (id < toOccur.id)
                return -1;
            else if (id > toOccur.id)
                return 1;
            else
                return 0;
        }

        long time;
        String type;
        Runnable handler;

        private long id;
    }

    private long numPendingInterruptsCreated = 0;

    private Privilege privilege;

    private boolean enabled;
    private TreeSet<PendingInterrupt> pending;

    private static final char dbgInt = 'i';

    private class InterruptPrivilege implements Privilege.InterruptPrivilege {
        public void schedule(long when, String type, Runnable handler) {
            Interrupt.this.schedule(when, type, handler);
        }

        public void tick(boolean inKernelMode) {
            Interrupt.this.tick(inKernelMode);
        }
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * An OS kernel.
 */
public abstract class Kernel {
    /** Globally accessible reference to the kernel. */
    public static Kernel kernel = null;

    /**
     * Allocate a new kernel.
     */
    public Kernel() {
        // make sure only one kernel is created
        Lib.assertTrue(kernel == null);
        kernel = this;
    }

    /**
     * Initialize this kernel.
     */
    public abstract void initialize(String[] args);

    /**
     * Test that this module works.
     *
     * <b>Warning:</b> this method will not be invoked by the autograder when
     * we grade your projects. You should perform all initialization in
     * <tt>initialize()</tt>.
     */
    public abstract void selfTest();

    /**
     * Begin executing user programs, if applicable.
     */
    public abstract void run();

    /**
     * Terminate this kernel. Never returns.
     */
    public abstract void terminate();
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.security.PrivilegedAction;
import java.util.Random;

/**
 * Thrown when an assertion fails.
 */
class AssertionFailureError extends Error {
    AssertionFailureError() {
        super();
    }

    AssertionFailureError(String message) {
        super(message);
    }
}

/**
 * Provides miscellaneous library routines.
 */
public final class Lib {
    /**
     * Prevent instantiation.
     */
    private Lib() {
    }

    private static Random random = null;

    /**
     * Seed the random number generater. May only be called once.
     *
     * @param   randomSeed      the seed for the random number generator.
     */
    public static void seedRandom(long randomSeed) {
        assertTrue(random == null);
        random = new Random(randomSeed);
    }

    /**
     * Return a random integer between 0 and <i>range - 1</i>. Must not be
     * called before <tt>seedRandom()</tt> seeds the random number generator.
     *
     * @param   range   a positive value specifying the number of possible
     *                  return values.
     * @return  a random integer in the specified range.
     */
    public static int random(int range) {
        assertTrue(range > 0);
        return random.nextInt(range);
    }

    /**
     * Return a random double between 0.0 (inclusive) and 1.0 (exclusive).
     *
     * @return  a random double between 0.0 and 1.0.
     */
    public static double random() {
        return random.nextDouble();
    }

    /**
     * Asserts that <i>expression</i> is <tt>true</tt>. If not, then Nachos
     * exits with an error message.
     *
     * @param   expression      the expression to assert.
     */
    public static void assertTrue(boolean expression) {
        if (!expression)
            throw new AssertionFailureError();
    }

    /**
     * Asserts that <i>expression</i> is <tt>true</tt>. If not, then Nachos
     * exits with the specified error message.
     *
     * @param   expression      the expression to assert.
     * @param   message         the error message.
     */
    public static void assertTrue(boolean expression, String message) {
        if (!expression)
            throw new AssertionFailureError(message);
    }

    /**
     * Asserts that this call is never made. Same as <tt>assertTrue(false)</tt>.
     */
    public static void assertNotReached() {
        assertTrue(false);
    }

    /**
     * Asserts that this call is never made, with the specified error messsage.
     * Same as <tt>assertTrue(false, message)</tt>.
     *
     * @param   message the error message.
     */
    public static void assertNotReached(String message) {
        assertTrue(false, message);
    }

    /**
     * Print <i>message</i> if <i>flag</i> was enabled on the command line. To
     * specify which flags to enable, use the -d command line option. For
     * example, to enable flags a, c, and e, do the following:
     *
     * <p>
     * <pre>nachos -d ace</pre>
     *
     * <p>
     * Nachos uses several debugging flags already, but you are encouraged to
     * add your own.
     *
     * @param   flag    the debug flag that must be set to print this message.
     * @param   message the debug message.
     */
    public static void debug(char flag, String message) {
        if (test(flag))
            System.out.println(message);
    }

    /**
     * Tests if <i>flag</i> was enabled on the command line.
```

```
     *
     * @param   flag    the debug flag to test.
     *
     * @return  <tt>true</tt> if this flag was enabled on the command line.
     */
    public static boolean test(char flag) {
        if (debugFlags == null)
            return false;
        else if (debugFlags[(int) '+'])
            return true;
        else if (flag >= 0 && flag < 0x80 && debugFlags[(int) flag])
            return true;
        else
            return false;
    }

    /**
     * Enable all the debug flags in <i>flagsString</i>.
     *
     * @param   flagsString     the flags to enable.
     */
    public static void enableDebugFlags(String flagsString) {
        if (debugFlags == null)
            debugFlags = new boolean[0x80];

        char[] newFlags = flagsString.toCharArray();
        for (int i=0; i<newFlags.length; i++) {
            char c = newFlags[i];
            if (c >= 0 && c < 0x80)
                debugFlags[(int) c] = true;
        }
    }

    /** Debug flags specified on the command line. */
    private static boolean debugFlags[];

    /**
     * Read a file, verifying that the requested number of bytes is read, and
     * verifying that the read operation took a non-zero amount of time.
     *
     * @param   file            the file to read.
     * @param   position        the file offset at which to start reading.
     * @param   buf             the buffer in which to store the data.
     * @param   offset          the buffer offset at which storing begins.
     * @param   length          the number of bytes to read.
     */
    public static void strictReadFile(OpenFile file, int position,
                                      byte[] buf, int offset, int length) {
        long startTime = Machine.timer().getTime();
        assertTrue(file.read(position, buf, offset, length) == length);
        long finishTime = Machine.timer().getTime();
        assertTrue(finishTime>startTime);
    }

    /**
     * Load an entire file into memory.
     *
     * @param   file    the file to load.
     * @return  an array containing the contents of the entire file, or
     *          <tt>null</tt> if an error occurred.
     */
    public static byte[] loadFile(OpenFile file) {
        int startOffset = file.tell();

        int length = file.length();
        if (length < 0)
            return null;

        byte[] data = new byte[length];

        file.seek(0);
        int amount = file.read(data, 0, length);
        file.seek(startOffset);

        if (amount == length)
            return data;
        else
            return null;
    }

    /**
     * Take a read-only snapshot of a file.
     *
     * @param   file    the file to take a snapshot of.
     * @return  a read-only snapshot of the file.
     */
    public static OpenFile cloneFile(OpenFile file) {
        OpenFile clone = new ArrayFile(loadFile(file));

        clone.seek(file.tell());

        return clone;
    }

    /**
     * Convert a short into its little-endian byte string representation.
     *
     * @param   array   the array in which to store the byte string.
     * @param   offset  the offset in the array where the string will start.
     * @param   value   the value to convert.
     */
    public static void bytesFromShort(byte[] array, int offset, short value) {
        array[offset+0] = (byte) ((value>>0)&0xFF);
        array[offset+1] = (byte) ((value>>8)&0xFF);
    }

    /**
     * Convert an int into its little-endian byte string representation.
     *
     * @param   array   the array in which to store the byte string.
     * @param   offset  the offset in the array where the string will start.
     * @param   value   the value to convert.
     */
    public static void bytesFromInt(byte[] array, int offset, int value) {
        array[offset+0] = (byte) ((value>>0) &0xFF);
        array[offset+1] = (byte) ((value>>8) &0xFF);
        array[offset+2] = (byte) ((value>>16)&0xFF);
        array[offset+3] = (byte) ((value>>24)&0xFF);
    }

    /**
     * Convert an int into its little-endian byte string representation, and
     * return an array containing it.
     *
     * @param   value   the value to convert.
     * @return  an array containing the byte string.
     */
    public static byte[] bytesFromInt(int value) {
        byte[] array = new byte[4];
        bytesFromInt(array, 0, value);
```

```
    return array;
}

/**
 * Convert an int into a little-endian byte string representation of the
 * specified length.
 *
 * @param   array   the array in which to store the byte string.
 * @param   offset  the offset in the array where the string will start.
 * @param   length  the number of bytes to store (must be 1, 2, or 4).
 * @param   value   the value to convert.
 */
public static void bytesFromInt(byte[] array, int offset,
                                int length, int value) {
    assertTrue(length==1 || length==2 || length==4);

    switch (length) {
    case 1:
        array[offset] = (byte) value;
        break;
    case 2:
        bytesFromShort(array, offset, (short) value);
        break;
    case 4:
        bytesFromInt(array, offset, value);
        break;
    }
}

/**
 * Convert to a short from its little-endian byte string representation.
 *
 * @param   array   the array containing the byte string.
 * @param   offset  the offset of the byte string in the array.
 * @return  the corresponding short value.
 */
public static short bytesToShort(byte[] array, int offset) {
    return (short) ((((short) array[offset+0] & 0xFF) << 0) |
                    (((short) array[offset+1] & 0xFF) << 8));
}

/**
 * Convert to an unsigned short from its little-endian byte string
 * representation.
 *
 * @param   array   the array containing the byte string.
 * @param   offset  the offset of the byte string in the array.
 * @return  the corresponding short value.
 */
public static int bytesToUnsignedShort(byte[] array, int offset) {
    return (((int) bytesToShort(array, offset)) & 0xFFFF);
}

/**
 * Convert to an int from its little-endian byte string representation.
 *
 * @param   array   the array containing the byte string.
 * @param   offset  the offset of the byte string in the array.
 * @return  the corresponding int value.
 */
public static int bytesToInt(byte[] array, int offset) {
    return (int) ((((int) array[offset+0] & 0xFF) << 0)  |
                  (((int) array[offset+1] & 0xFF) << 8)  |
                  (((int) array[offset+2] & 0xFF) << 16) |
                  (((int) array[offset+3] & 0xFF) << 24));
}

/**
 * Convert to an int from a little-endian byte string representation of the
 * specified length.
 *
 * @param   array   the array containing the byte string.
 * @param   offset  the offset of the byte string in the array.
 * @param   length  the length of the byte string.
 * @return  the corresponding value.
 */
public static int bytesToInt(byte[] array, int offset, int length) {
    assertTrue(length==1 || length==2 || length==4);

    switch (length) {
    case 1:
        return array[offset];
    case 2:
        return bytesToShort(array, offset);
    case 4:
        return bytesToInt(array, offset);
    default:
        return -1;
    }
}

/**
 * Convert to a string from a possibly null-terminated array of bytes.
 *
 * @param   array   the array containing the byte string.
 * @param   offset  the offset of the byte string in the array.
 * @param   length  the maximum length of the byte string.
 * @return  a string containing the specified bytes, up to and not
 *          including the null-terminator (if present).
 */
public static String bytesToString(byte[] array, int offset, int length) {
    int i;
    for (i=0; i<length; i++) {
        if (array[offset+i] == 0)
            break;
    }

    return new String(array, offset, i);
}

/** Mask out and shift a bit substring.
 *
 * @param   bits    the bit string.
 * @param   lowest  the first bit of the substring within the string.
 * @param   size    the number of bits in the substring.
 * @return  the substring.
 */
public static int extract(int bits, int lowest, int size) {
    if (size == 32)
        return (bits >> lowest);
    else
        return ((bits >> lowest) & ((1<<size)-1));
}

/** Mask out and shift a bit substring.
 *
 * @param   bits    the bit string.
 * @param   lowest  the first bit of the substring within the string.
 * @param   size    the number of bits in the substring.
 * @return  the substring.
```

```
     */
    public static long extract(long bits, int lowest, int size) {
        if (size == 64)
            return (bits >> lowest);
        else
            return ((bits >> lowest) & ((1L<<size)-1));
    }

    /** Mask out and shift a bit substring; then sign extend the substring.
     *
     * @param   bits    the bit string.
     * @param   lowest  the first bit of the substring within the string.
     * @param   size    the number of bits in the substring.
     * @return  the substring, sign-extended.
     */
    public static int extend(int bits, int lowest, int size) {
        int extra = 32 - (lowest+size);
        return ((extract(bits, lowest, size) << extra) >> extra);
    }

    /** Test if a bit is set in a bit string.
     *
     * @param   flag    the flag to test.
     * @param   bits    the bit string.
     * @return  <tt>true</tt> if <tt>(bits & flag)</tt> is non-zero.
     */
    public static boolean test(long flag, long bits) {
        return ((bits & flag) != 0);
    }

    /**
     * Creates a padded upper-case string representation of the integer
     * argument in base 16.
     *
     * @param   i       an integer.
     * @return  a padded upper-case string representation in base 16.
     */
    public static String toHexString(int i) {
        return toHexString(i, 8);
    }

    /**
     * Creates a padded upper-case string representation of the integer
     * argument in base 16, padding to at most the specified number of digits.
     *
     * @param   i       an integer.
     * @param   pad     the minimum number of hex digits to pad to.
     * @return  a padded upper-case string representation in base 16.
     */
    public static String toHexString(int i, int pad) {
        String result = Integer.toHexString(i).toUpperCase();
        while (result.length() < pad)
            result = "0" + result;
        return result;
    }

    /**
     * Divide two non-negative integers, round the quotient up to the nearest
     * integer, and return it.
     *
     * @param   a       the numerator.
     * @param   b       the denominator.
     * @return  <tt>ceiling(a / b)</tt>.
     */
    public static int divRoundUp(int a, int b) {
        Lib.assertTrue(a >= 0 && b > 0);

        return ((a + (b-1)) / b);
    }

    /**
     * Load and return the named class, or return <tt>null</tt> if the class
     * could not be loaded.
     *
     * @param   className       the name of the class to load.
     * @return  the loaded class, or <tt>null</tt> if an error occurred.
     */
    public static Class tryLoadClass(String className) {
        try {
            return ClassLoader.getSystemClassLoader().loadClass(className);
        }
        catch (Throwable e) {
            return null;
        }
    }

    /**
     * Load and return the named class, terminating Nachos on any error.
     *
     * @param   className       the name of the class to load.
     * @return  the loaded class.
     */
    public static Class loadClass(String className) {
        try {
            return ClassLoader.getSystemClassLoader().loadClass(className);
        }
        catch (Throwable e) {
            Machine.terminate(e);
            return null;
        }
    }

    /**
     * Create and return a new instance of the named class, using the
     * constructor that takes no arguments.
     *
     * @param   className       the name of the class to instantiate.
     * @return  a new instance of the class.
     */
    public static Object constructObject(String className) {
        try {
            // kamil - workaround for Java 1.4
            // Thanks to Ka-Hing Cheung for the suggestion.
            // Fixed for Java 1.5 by geels
            Class[] param_types = new Class[0];
            Object[] params = new Object[0];
            return loadClass(className).getConstructor(param_types).newInstance(params
);
        }
        catch (Throwable e) {
            Machine.terminate(e);
            return null;
        }
    }

    /**
     * Verify that the specified class extends or implements the specified
     * superclass.
     *
     * @param   cls     the descendant class.
```

```
     * @param   superCls        the ancestor class.
     */
    public static void checkDerivation(Class<?> cls, Class<?> superCls) {
        Lib.assertTrue(superCls.isAssignableFrom(cls));
    }

    /**
     * Verifies that the specified class is public and not abstract, and that a
     * constructor with the specified signature exists and is public.
     *
     * @param   cls     the class containing the constructor.
     * @param   parameterTypes  the list of parameters.
     */
    public static void checkConstructor(Class cls, Class[] parameterTypes) {
        try {
            Lib.assertTrue(Modifier.isPublic(cls.getModifiers()) &&
                           !Modifier.isAbstract(cls.getModifiers()));
            Constructor constructor = cls.getConstructor(parameterTypes);
            Lib.assertTrue(Modifier.isPublic(constructor.getModifiers()));
        }
        catch (Exception e) {
            Lib.assertNotReached();
        }
    }

    /**
     * Verifies that the specified class is public, and that a non-static
     * method with the specified name and signature exists, is public, and
     * returns the specified type.
     *
     * @param   cls     the class containing the non-static method.
     * @param   methodName      the name of the non-static method.
     * @param   parameterTypes  the list of parameters.
     * @param   returnType      the required return type.
     */
    public static void checkMethod(Class cls, String methodName,
                                   Class[] parameterTypes, Class returnType) {
        try {
            Lib.assertTrue(Modifier.isPublic(cls.getModifiers()));
            Method method = cls.getMethod(methodName, parameterTypes);
            Lib.assertTrue(Modifier.isPublic(method.getModifiers()) &&
                           !Modifier.isStatic(method.getModifiers()));
            Lib.assertTrue(method.getReturnType() == returnType);
        }
        catch (Exception e) {
            Lib.assertNotReached();
        }
    }

    /**
     * Verifies that the specified class is public, and that a static method
     * with the specified name and signature exists, is public, and returns the
     * specified type.
     *
     * @param   cls     the class containing the static method.
     * @param   methodName      the name of the static method.
     * @param   parameterTypes  the list of parameters.
     * @param   returnType      the required return type.
     */
    public static void checkStaticMethod(Class cls, String methodName,
                                         Class[] parameterTypes,
                                         Class returnType) {
        try {
            Lib.assertTrue(Modifier.isPublic(cls.getModifiers()));
            Method method = cls.getMethod(methodName, parameterTypes);
            Lib.assertTrue(Modifier.isPublic(method.getModifiers()) &&
                           Modifier.isStatic(method.getModifiers()));
            Lib.assertTrue(method.getReturnType() == returnType);
        }
        catch (Exception e) {
            Lib.assertNotReached();
        }
    }

    /**
     * Verifies that the specified class is public, and that a non-static field
     * with the specified name and type exists, is public, and is not final.
     *
     * @param   cls     the class containing the field.
     * @param   fieldName       the name of the field.
     * @param   fieldType       the required type.
     */
    public static void checkField(Class cls, String fieldName,
                                  Class fieldType) {
        try {
            Lib.assertTrue(Modifier.isPublic(cls.getModifiers()));
            Field field = cls.getField(fieldName);
            Lib.assertTrue(field.getType() == fieldType);
            Lib.assertTrue(Modifier.isPublic(field.getModifiers()) &&
                           !Modifier.isStatic(field.getModifiers()) &&
                           !Modifier.isFinal(field.getModifiers()));
        }
        catch (Exception e) {
            Lib.assertNotReached();
        }
    }

    /**
     * Verifies that the specified class is public, and that a static field
     * with the specified name and type exists and is public.
     *
     * @param   cls     the class containing the static field.
     * @param   fieldName       the name of the static field.
     * @param   fieldType       the required type.
     */
    public static void checkStaticField(Class cls, String fieldName,
                                        Class fieldType) {
        try {
            Lib.assertTrue(Modifier.isPublic(cls.getModifiers()));
            Field field = cls.getField(fieldName);
            Lib.assertTrue(field.getType() == fieldType);
            Lib.assertTrue(Modifier.isPublic(field.getModifiers()) &&
                           Modifier.isStatic(field.getModifiers()));
        }
        catch (Exception e) {
            Lib.assertNotReached();
        }
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;
import nachos.ag.*;

import java.io.File;

/**
 * The master class of the simulated machine. Processes command line arguments,
 * constructs all simulated hardware devices, and starts the grader.
 */
public final class Machine {
    /**
     * Nachos main entry point.
     *
     * @param   args    the command line arguments.
     */
    public static void main(final String[] args) {
        System.out.print("nachos 5.0j initializing...");

        Lib.assertTrue(Machine.args == null);
        Machine.args = args;

        processArgs();

        Config.load(configFileName);

        // get the current directory (.)
        baseDirectory = new File(new File("").getAbsolutePath());
        // get the nachos directory (./nachos)
        nachosDirectory = new File(baseDirectory, "nachos");

        String testDirectoryName =
            Config.getString("FileSystem.testDirectory");

        // get the test directory
        if (testDirectoryName != null) {
            testDirectory = new File(testDirectoryName);
        }
        else {
            // use ../test
            testDirectory = new File(baseDirectory.getParentFile(), "test");
        }

        securityManager = new NachosSecurityManager(testDirectory);
        privilege = securityManager.getPrivilege();

        privilege.machine = new MachinePrivilege();

        TCB.givePrivilege(privilege);
        privilege.stats = stats;

        securityManager.enable();
        createDevices();
        checkUserClasses();

        autoGrader = (AutoGrader) Lib.constructObject(autoGraderClassName);

        new TCB().start(new Runnable() {
            public void run() { autoGrader.start(privilege); }
        });
    }

    /**
     * Yield to non-Nachos threads. Use in non-preemptive JVM's to give
     * non-Nachos threads a chance to run.
     */
    public static void yield() {
        Thread.yield();
    }

    /**
     * Terminate Nachos. Same as <tt>TCB.die()</tt>.
     */
    public static void terminate() {
        TCB.die();
    }

    /**
     * Terminate Nachos as the result of an unhandled exception or error.
     *
     * @param   e       the exception or error.
     */
    public static void terminate(Throwable e) {
        if (e instanceof ThreadDeath)
            throw (ThreadDeath) e;

        e.printStackTrace();
        terminate();
    }

    /**
     * Print stats, and terminate Nachos.
     */
    public static void halt() {
        System.out.print("Machine halting!\n\n");
        stats.print();
        terminate();
    }

    /**
     * Return an array containing all command line arguments.
     *
     * @return  the command line arguments passed to Nachos.
     */
    public static String[] getCommandLineArguments() {
        String[] result = new String[args.length];

        System.arraycopy(args, 0, result, 0, args.length);

        return result;
    }

    private static void processArgs() {
        for (int i=0; i<args.length; ) {
            String arg = args[i++];
            if (arg.length() > 0 && arg.charAt(0) == '-') {
                if (arg.equals("-d")) {
                    Lib.assertTrue(i < args.length, "switch without argument");
                    Lib.enableDebugFlags(args[i++]);
                }
                else if (arg.equals("-h")) {
                    System.out.print(help);
                    System.exit(1);
                }
                else if (arg.equals("-m")) {
                    Lib.assertTrue(i < args.length, "switch without argument");
                    try {
```

```
                numPhysPages = Integer.parseInt(args[i++]);
            }
            catch (NumberFormatException e) {
                Lib.assertNotReached("bad value for -m switch");
            }
        }
        else if (arg.equals("-s")) {
            Lib.assertTrue(i < args.length, "switch without argument");
            try {
                randomSeed = Long.parseLong(args[i++]);
            }
            catch (NumberFormatException e) {
                Lib.assertNotReached("bad value for -s switch");
            }
        }
        else if (arg.equals("-x")) {
            Lib.assertTrue(i < args.length, "switch without argument");
            shellProgramName = args[i++];
        }
        else if (arg.equals("-z")) {
            System.out.print(copyright);
            System.exit(1);
        }
        // these switches are reserved for the autograder
        else if (arg.equals("-[]")) {
            Lib.assertTrue(i < args.length, "switch without argument");
            configFileName = args[i++];
        }
        else if (arg.equals("--")) {
            Lib.assertTrue(i < args.length, "switch without argument");
            autoGraderClassName = args[i++];
        }
    }

    Lib.seedRandom(randomSeed);
}

private static void createDevices() {
    interrupt = new Interrupt(privilege);
    timer = new Timer(privilege);

    if (Config.getBoolean("Machine.bank"))
        bank = new ElevatorBank(privilege);

    if (Config.getBoolean("Machine.processor")) {
        if (numPhysPages == -1)
            numPhysPages = Config.getInteger("Processor.numPhysPages");
        processor = new Processor(privilege, numPhysPages);
    }

    if (Config.getBoolean("Machine.console"))
        console = new StandardConsole(privilege);

    if (Config.getBoolean("Machine.stubFileSystem"))
        stubFileSystem = new StubFileSystem(privilege, testDirectory);

    if (Config.getBoolean("Machine.networkLink"))
        networkLink = new NetworkLink(privilege);
}

private static void checkUserClasses() {
    System.out.print(" user-check");

    Class aclsInt = (new int[0]).getClass();

    Class clsObject = Lib.loadClass("java.lang.Object");
    Class clsRunnable = Lib.loadClass("java.lang.Runnable");
    Class clsString = Lib.loadClass("java.lang.String");

    Class clsKernel = Lib.loadClass("nachos.machine.Kernel");
    Class clsFileSystem = Lib.loadClass("nachos.machine.FileSystem");
    Class clsRiderControls = Lib.loadClass("nachos.machine.RiderControls");
    Class clsElevatorControls =
        Lib.loadClass("nachos.machine.ElevatorControls");
    Class clsRiderInterface =
        Lib.loadClass("nachos.machine.RiderInterface");
    Class clsElevatorControllerInterface =
        Lib.loadClass("nachos.machine.ElevatorControllerInterface");

    Class clsAlarm = Lib.loadClass("nachos.threads.Alarm");
    Class clsThreadedKernel =
        Lib.loadClass("nachos.threads.ThreadedKernel");
    Class clsKThread = Lib.loadClass("nachos.threads.KThread");
    Class clsCommunicator = Lib.loadClass("nachos.threads.Communicator");
    Class clsSemaphore = Lib.loadClass("nachos.threads.Semaphore");
    Class clsLock = Lib.loadClass("nachos.threads.Lock");
    Class clsCondition = Lib.loadClass("nachos.threads.Condition");
    Class clsCondition2 = Lib.loadClass("nachos.threads.Condition2");
    Class clsRider = Lib.loadClass("nachos.threads.Rider");
    Class clsElevatorController =
        Lib.loadClass("nachos.threads.ElevatorController");

    Lib.checkDerivation(clsThreadedKernel, clsKernel);

    Lib.checkStaticField(clsThreadedKernel, "alarm", clsAlarm);
    Lib.checkStaticField(clsThreadedKernel, "fileSystem", clsFileSystem);

    Lib.checkMethod(clsAlarm, "waitUntil", new Class[] { long.class },
                    void.class);

    Lib.checkConstructor(clsKThread, new Class[] { });
    Lib.checkConstructor(clsKThread, new Class[] { clsRunnable });

    Lib.checkStaticMethod(clsKThread, "currentThread", new Class[] {},
                          clsKThread);
    Lib.checkStaticMethod(clsKThread, "finish", new Class[] {},
                          void.class);
    Lib.checkStaticMethod(clsKThread, "yield", new Class[] {}, void.class);
    Lib.checkStaticMethod(clsKThread, "sleep", new Class[] {}, void.class);

    Lib.checkMethod(clsKThread, "setTarget", new Class[]{ clsRunnable },
                    clsKThread);
    Lib.checkMethod(clsKThread, "setName", new Class[] { clsString },
                    clsKThread);
    Lib.checkMethod(clsKThread, "getName", new Class[] { }, clsString);
    Lib.checkMethod(clsKThread, "fork", new Class[] { }, void.class);
    Lib.checkMethod(clsKThread, "ready", new Class[] { }, void.class);
    Lib.checkMethod(clsKThread, "join", new Class[] { }, void.class);

    Lib.checkField(clsKThread, "schedulingState", clsObject);

    Lib.checkConstructor(clsCommunicator, new Class[] {});
    Lib.checkMethod(clsCommunicator, "speak", new Class[] { int.class },
                    void.class);
    Lib.checkMethod(clsCommunicator, "listen", new Class[] { }, int.class);

    Lib.checkConstructor(clsSemaphore, new Class[] { int.class });
    Lib.checkMethod(clsSemaphore, "P", new Class[] { }, void.class);
    Lib.checkMethod(clsSemaphore, "V", new Class[] { }, void.class);
```

```
    Lib.checkConstructor(clsLock, new Class[] { });
    Lib.checkMethod(clsLock, "acquire", new Class[] { }, void.class);
    Lib.checkMethod(clsLock, "release", new Class[] { }, void.class);
    Lib.checkMethod(clsLock, "isHeldByCurrentThread", new Class[]{ },
                    boolean.class);

    Lib.checkConstructor(clsCondition, new Class[] { clsLock });
    Lib.checkConstructor(clsCondition2, new Class[] { clsLock });

    Lib.checkMethod(clsCondition, "sleep", new Class[] { }, void.class);
    Lib.checkMethod(clsCondition, "wake", new Class[] { }, void.class);
    Lib.checkMethod(clsCondition, "wakeAll", new Class[] { }, void.class);
    Lib.checkMethod(clsCondition2, "sleep", new Class[] { }, void.class);
    Lib.checkMethod(clsCondition2, "wake", new Class[] { }, void.class);
    Lib.checkMethod(clsCondition2, "wakeAll", new Class[] { }, void.class);

    Lib.checkDerivation(clsRider, clsRiderInterface);

    Lib.checkConstructor(clsRider, new Class[] { });
    Lib.checkMethod(clsRider, "initialize",
                    new Class[] { clsRiderControls, aclsInt }, void.class);

    Lib.checkDerivation(clsElevatorController,
                        clsElevatorControllerInterface);

    Lib.checkConstructor(clsElevatorController, new Class[] { });
    Lib.checkMethod(clsElevatorController, "initialize",
                    new Class[] { clsElevatorControls }, void.class);
}

/**
 * Prevent instantiation.
 */
private Machine() {
}

/**
 * Return the hardware interrupt manager.
 *
 * @return  the hardware interrupt manager.
 */
public static Interrupt interrupt() { return interrupt; }

/**
 * Return the hardware timer.
 *
 * @return  the hardware timer.
 */
public static Timer timer() { return timer; }

/**
 * Return the hardware elevator bank.
 *
 * @return  the hardware elevator bank, or <tt>null</tt> if it is not
 *          present.
 */
public static ElevatorBank bank() { return bank; }

/**
 * Return the MIPS processor.
 *
 * @return  the MIPS processor, or <tt>null</tt> if it is not present.
 */
public static Processor processor() { return processor; }

/**
 * Return the hardware console.
 *
 * @return  the hardware console, or <tt>null</tt> if it is not present.
 */
public static SerialConsole console() { return console; }

/**
 * Return the stub filesystem.
 *
 * @return  the stub file system, or <tt>null</tt> if it is not present.
 */
public static FileSystem stubFileSystem() { return stubFileSystem; }

/**
 * Return the network link.
 *
 * @return  the network link,  or <tt>null</tt> if it is not present.
 */
public static NetworkLink networkLink() { return networkLink; }

/**
 * Return the autograder.
 *
 * @return  the autograder.
 */
public static AutoGrader autoGrader() { return autoGrader; }

private static Interrupt interrupt = null;
private static Timer timer = null;
private static ElevatorBank bank = null;
private static Processor processor = null;
private static SerialConsole console = null;
private static FileSystem stubFileSystem = null;
private static NetworkLink networkLink = null;
private static AutoGrader autoGrader = null;

private static String autoGraderClassName = "nachos.ag.AutoGrader";

/**
 * Return the name of the shell program that a user-programming kernel
 * must run. Make sure <tt>UserKernel.run()</tt> <i>always</i> uses this
 * method to decide which program to run.
 *
 * @return  the name of the shell program to run.
 */
public static String getShellProgramName() {
    if (shellProgramName == null)
        shellProgramName = Config.getString("Kernel.shellProgram");

    Lib.assertTrue(shellProgramName != null);
    return shellProgramName;
}

private static String shellProgramName = null;

/**
 * Return the name of the process class that the kernel should use. In
 * the multi-programming project, returns
 * <tt>nachos.userprog.UserProcess</tt>. In the VM project, returns
 * <tt>nachos.vm.VMProcess</tt>. In the networking project, returns
 * <tt>nachos.network.NetProcess</tt>.
 *
 * @return  the name of the process class that the kernel should use.
 *
```

```
     * @see     nachos.userprog.UserKernel#run
     * @see     nachos.userprog.UserProcess
     * @see     nachos.vm.VMProcess
     * @see     nachos.network.NetProcess
     */
    public static String getProcessClassName() {
        if (processClassName == null)
            processClassName = Config.getString("Kernel.processClassName");

        Lib.assertTrue(processClassName != null);
        return processClassName;
    }

    private static String processClassName = null;

    private static NachosSecurityManager securityManager;
    private static Privilege privilege;

    private static String[] args = null;

    private static Stats stats = new Stats();

    private static int numPhysPages = -1;
    private static long randomSeed = 0;

    private static File baseDirectory, nachosDirectory, testDirectory;
    private static String configFileName = "nachos.conf";

    private static final String help =
        "\n" +
        "Options:\n" +
        "\n" +
        "\t-d <debug flags>\n" +
        "\t\tEnable some debug flags, e.g. -d ti\n" +
        "\n" +
        "\t-h\n" +
        "\t\tPrint this help message.\n" +
        "\n" +
        "\t-m <pages>\n" +
        "\t\tSpecify how many physical pages of memory to simulate.\n" +
        "\n" +
        "\t-s <seed>\n" +
        "\t\tSpecify the seed for the random number generator (seed is a\n" +
        "\t\tlong).\n" +
        "\n" +
        "\t-x <program>\n" +
        "\t\tSpecify a program that UserKernel.run() should execute,\n" +
        "\t\tinstead of the value of the configuration variable\n" +
        "\t\tKernel.shellProgram\n" +
        "\n" +
        "\t-z\n" +
        "\t\tprint the copyright message\n" +
        "\n" +
        "\t-- <grader class>\n" +
        "\t\tSpecify an autograder class to use, instead of\n" +
        "\t\tnachos.ag.AutoGrader\n" +
        "\n" +
        "\t-# <grader arguments>\n" +
        "\t\tSpecify the argument string to pass to the autograder.\n" +
        "\n" +
        "\t-[] <config file>\n" +
        "\t\tSpecifiy a config file to use, instead of nachos.conf\n" +
        ""
        ;

    private static final String copyright = "\n"
        + "Copyright 1992-2001 The Regents of the University of California.\n"
        + "All rights reserved.\n"
        + "\n"
        + "Permission to use, copy, modify, and distribute this software and\n"
        + "its documentation for any purpose, without fee, and without\n"
        + "written agreement is hereby granted, provided that the above\n"
        + "copyright notice and the following two paragraphs appear in all\n"
        + "copies of this software.\n"
        + "\n"
        + "IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY\n"
        + "PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL\n"
        + "DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS\n"
        + "DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN\n"
        + "ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.\n"
        + "\n"
        + "THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY\n"
        + "WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES\n"
        + "OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE\n"
        + "SOFTWARE PROVIDED HEREUNDER IS ON AN \"AS IS\" BASIS, AND THE\n"
        + "UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE\n"
        + "MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.\n"
        ;

    private static class MachinePrivilege
        implements Privilege.MachinePrivilege {
        public void setConsole(SerialConsole console) {
            Machine.console = console;
        }
    }

    // dummy variables to make javac smarter
    private static Coff dummy1 = null;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * Thrown when a malformed packet is processed.
 */
public class MalformedPacketException extends Exception {
    /**
     * Allocate a new <tt>MalformedPacketException</tt>.
     */
    public MalformedPacketException() {
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

import java.io.IOException;
import java.net.DatagramSocket;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.SocketException;

/**
 * A full-duplex network link. Provides ordered, unreliable delivery of
 * limited-size packets to other machines on the network. Packets are
 * guaranteed to be uncorrupted as well.
 *
 * <p>
 * Recall the general layering of network protocols:
 * <ul>
 * <li>Session/Transport
 * <li>Network
 * <li>Link
 * <li>Physical
 * </ul>
 *
 * <p>
 * The physical layer provides a bit stream interface to the link layer. This
 * layer is very hardware-dependent.
 *
 * <p>
 * The link layer uses the physical layer to provide a packet interface to the
 * network layer. The link layer generally provides unreliable delivery of
 * limited-size packets, but guarantees that packets will not arrive out of
 * order. Some links protect against packet corruption as well. The ethernet
 * protocol is an example of a link layer.
 *
 * <p>
 * The network layer exists to connect multiple networks together into an
 * internet. The network layer provides globally unique addresses. Routers
 * (a.k.a. gateways) move packets across networks at this layer. The network
 * layer provides unordered, unreliable delivery of limited-size uncorrupted
 * packets to any machine on the same internet. The most commonly used network
 * layer protocol is IP (Internet Protocol), which is used to connect the
 * Internet.
 *
 * <p>
 * The session/transport layer provides a byte-stream interface to the
 * application. This means that the transport layer must deliver uncorrupted
 * bytes to the application, in the same order they were sent. Byte-streams
 * must be connected and disconnected, and exist between ports, not machines.
 *
 * <p>
 * This class provides a link layer abstraction. Since we do not allow
 * different Nachos networks to communicate with one another, there is no need
 * for a network layer in Nachos. This should simplify your design for the
 * session/transport layer, since you can assume packets never arrive out of
 * order.
 */
public class NetworkLink {
    /**
     * Allocate a new network link.
     *
     * <p>
     * <tt>nachos.conf</tt> specifies the reliability of the network. The
     * reliability, between 0 and 1, is the probability that any particular
     * packet will not get dropped by the network.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public NetworkLink(Privilege privilege) {
        System.out.print(" network");

        this.privilege = privilege;

        try {
            localHost = InetAddress.getLocalHost();
        }
        catch (UnknownHostException e) {
            localHost = null;
        }

        Lib.assertTrue(localHost != null);

        reliability = Config.getDouble("NetworkLink.reliability");
        Lib.assertTrue(reliability > 0 && reliability <= 1.0);

        socket = null;

        for (linkAddress=0;linkAddress<Packet.linkAddressLimit;linkAddress++) {
            try {
                socket = new DatagramSocket(portBase + linkAddress, localHost);
                break;
            }
            catch (SocketException e) {
            }
        }

        if (socket == null) {
            System.out.println("");
            System.out.println("Unable to acquire a link address!");
            Lib.assertNotReached();
        }

        System.out.print("(" + linkAddress + ")");

        receiveInterrupt = new Runnable() {
                public void run() { receiveInterrupt(); }
            };

        sendInterrupt = new Runnable() {
                public void run() { sendInterrupt(); }
            };

        scheduleReceiveInterrupt();

        Thread receiveThread = new Thread(new Runnable() {
                public void run() { receiveLoop(); }
            });

        receiveThread.start();
    }

    /**
     * Returns the address of this network link.
     *
     * @return  the address of this network link.
     */
```

```
public int getLinkAddress() {
    return linkAddress;
}

/**
 * Set this link's receive and send interrupt handlers.
 *
 * <p>
 * The receive interrupt handler is called every time a packet arrives
 * and can be read using <tt>receive()</tt>.
 *
 * <p>
 * The send interrupt handler is called every time a packet sent with
 * <tt>send()</tt> is finished being sent. This means that another
 * packet can be sent.
 *
 * @param   receiveInterruptHandler the callback to call when a packet
 *                                  arrives.
 * @param   sendInterruptHandler    the callback to call when another
 *                                  packet can be sent.
 */
public void setInterruptHandlers(Runnable receiveInterruptHandler,
                                 Runnable sendInterruptHandler) {
    this.receiveInterruptHandler = receiveInterruptHandler;
    this.sendInterruptHandler = sendInterruptHandler;
}

private void scheduleReceiveInterrupt() {
    privilege.interrupt.schedule(Stats.NetworkTime, "network recv",
                                 receiveInterrupt);
}

private synchronized void receiveInterrupt() {
    Lib.assertTrue(incomingPacket == null);

    if (incomingBytes != null) {
        if (Machine.autoGrader().canReceivePacket(privilege)) {
            try {
                incomingPacket = new Packet(incomingBytes);

                privilege.stats.numPacketsReceived++;
            }
            catch (MalformedPacketException e) {
            }
        }

        incomingBytes = null;
        notify();

        if (incomingPacket == null)
            scheduleReceiveInterrupt();
        else if (receiveInterruptHandler != null)
            receiveInterruptHandler.run();
    }
    else {
        scheduleReceiveInterrupt();
    }
}

/**
 * Return the next packet received.
 *
 * @return  the next packet received, or <tt>null</tt> if no packet is
 *          available.
 */
public Packet receive() {
    Packet p = incomingPacket;

    if (incomingPacket != null) {
        incomingPacket = null;
        scheduleReceiveInterrupt();
    }

    return p;
}

private void receiveLoop() {
    while (true) {
        synchronized(this) {
            while (incomingBytes != null) {
                try {
                    wait();
                }
                catch (InterruptedException e) {
                }
            }
        }

        byte[] packetBytes;

        try {
            byte[] buffer = new byte[Packet.maxPacketLength];

            DatagramPacket dp = new DatagramPacket(buffer, buffer.length);

            socket.receive(dp);

            packetBytes = new byte[dp.getLength()];

            System.arraycopy(buffer,0, packetBytes,0, packetBytes.length);
        }
        catch (IOException e) {
            return;
        }

        synchronized(this) {
            incomingBytes = packetBytes;
        }
    }
}

private void scheduleSendInterrupt() {
    privilege.interrupt.schedule(Stats.NetworkTime, "network send",
                                 sendInterrupt);
}

private void sendInterrupt() {
    Lib.assertTrue(outgoingPacket != null);

    // randomly drop packets, according to its reliability
    if (Machine.autoGrader().canSendPacket(privilege) &&
        Lib.random() <= reliability) {
        // ok, no drop
        privilege.doPrivileged(new Runnable() {
                public void run() { sendPacket(); }
            });
    }
    else {
        outgoingPacket = null;
    }
```

```
    if (sendInterruptHandler != null)
        sendInterruptHandler.run();
}

private void sendPacket() {
    Packet p = outgoingPacket;
    outgoingPacket = null;

    try {
        socket.send(new DatagramPacket(p.packetBytes, p.packetBytes.length,
                                       localHost, portBase+p.dstLink));

        privilege.stats.numPacketsSent++;
    }
    catch (IOException e) {
    }
}

/**
 * Send another packet. If a packet is already being sent, the result is
 * not defined.
 *
 * @param   pkt     the packet to send.
 */
public void send(Packet pkt) {
    if (outgoingPacket == null)
        scheduleSendInterrupt();

    outgoingPacket = pkt;
}

private static final int hash;
private static final int portBase;

/**
 * The address of the network to which are attached all network links in
 * this JVM. This is a hash on the account name of the JVM running this
 * Nachos instance. It is used to help prevent packets from other users
 * from accidentally interfering with this network.
 */
public static final byte networkID;

static {
    hash = System.getProperty("user.name").hashCode();
    portBase = 0x4E41 + Math.abs(hash%0x4E41);
    networkID  = (byte) (hash/0x4E41);
}

private Privilege privilege;

private Runnable receiveInterrupt;
private Runnable sendInterrupt;

private Runnable receiveInterruptHandler = null;
private Runnable sendInterruptHandler = null;

private InetAddress localHost;
private DatagramSocket socket;

private byte linkAddress;
private double reliability;

private byte[] incomingBytes = null;
private Packet incomingPacket = null;
    private Packet outgoingPacket = null;

    private boolean sendBusy = false;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import java.io.EOFException;

/**
 * A file that supports reading, writing, and seeking.
 */
public class OpenFile {
    /**
     * Allocate a new <tt>OpenFile</tt> object with the specified name on the
     * specified file system.
     *
     * @param   fileSystem      the file system to which this file belongs.
     * @param   name            the name of the file, on that file system.
     */
    public OpenFile(FileSystem fileSystem, String name) {
        this.fileSystem = fileSystem;
        this.name = name;
    }

    /**
     * Allocate a new unnamed <tt>OpenFile</tt> that is not associated with any
     * file system.
     */
    public OpenFile() {
        this(null, "unnamed");
    }

    /**
     * Get the file system to which this file belongs.
     *
     * @return  the file system to which this file belongs.
     */
    public FileSystem getFileSystem() {
        return fileSystem;
    }

    /**
     * Get the name of this open file.
     *
     * @return  the name of this open file.
     */
    public String getName() {
        return name;
    }

    /**
     * Read this file starting at the specified position and return the number
     * of bytes successfully read. If no bytes were read because of a fatal
     * error, returns -1
     *
     * @param   pos     the offset in the file at which to start reading.
     * @param   buf     the buffer to store the bytes in.
     * @param   offset  the offset in the buffer to start storing bytes.
     * @param   length  the number of bytes to read.
     * @return  the actual number of bytes successfully read, or -1 on failure.
     */
    public int read(int pos, byte[] buf, int offset, int length) {
        return -1;
    }

    /**
     * Write this file starting at the specified position and return the number
     * of bytes successfully written. If no bytes were written because of a
     * fatal error, returns -1.
     *
     * @param   pos     the offset in the file at which to start writing.
     * @param   buf     the buffer to get the bytes from.
     * @param   offset  the offset in the buffer to start getting.
     * @param   length  the number of bytes to write.
     * @return  the actual number of bytes successfully written, or -1 on
     *          failure.
     */
    public int write(int pos, byte[] buf, int offset, int length) {
        return -1;
    }

    /**
     * Get the length of this file.
     *
     * @return  the length of this file, or -1 if this file has no length.
     */
    public int length() {
        return -1;
    }

    /**
     * Close this file and release any associated system resources.
     */
    public void close() {
    }

    /**
     * Set the value of the current file pointer.
     */
    public void seek(int pos) {
    }

    /**
     * Get the value of the current file pointer, or -1 if this file has no
     * pointer.
     */
    public int tell() {
        return -1;
    }

    /**
     * Read this file starting at the current file pointer and return the
     * number of bytes successfully read. Advances the file pointer by this
     * amount. If no bytes could be* read because of a fatal error, returns -1.
     *
     * @param   buf     the buffer to store the bytes in.
     * @param   offset  the offset in the buffer to start storing bytes.
     * @param   length  the number of bytes to read.
     * @return  the actual number of bytes successfully read, or -1 on failure.
     */
    public int read(byte[] buf, int offset, int length) {
        return -1;
    }

    /**
     * Write this file starting at the current file pointer and return the
     * number of bytes successfully written. Advances the file pointer by this
     * amount. If no bytes could be written because of a fatal error, returns
     * -1.
     *
     * @param   buf     the buffer to get the bytes from.
     * @param   offset  the offset in the buffer to start getting.
```

```
     * @param   length  the number of bytes to write.
     * @return  the actual number of bytes successfully written, or -1 on
     *          failure.
     */
    public int write(byte[] buf, int offset, int length) {
        return -1;
    }

    private FileSystem fileSystem;
    private String name;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * An <tt>OpenFile</tt> that maintains a current file position.
 */
public abstract class OpenFileWithPosition extends OpenFile {
    /**
     * Allocate a new <tt>OpenFileWithPosition</tt> with the specified name on
     * the specified file system.
     *
     * @param   fileSystem      the file system to which this file belongs.
     * @param   name            the name of the file, on that file system.
     */
    public OpenFileWithPosition(FileSystem fileSystem, String name) {
        super(fileSystem, name);
    }

    /**
     * Allocate a new unnamed <tt>OpenFileWithPosition</tt> that is not
     * associated with any file system.
     */
    public OpenFileWithPosition() {
        super();
    }

    public void seek(int position) {
        this.position = position;
    }

    public int tell() {
        return position;
    }

    public int read(byte[] buf, int offset, int length) {
        int amount = read(position, buf, offset, length);
        if (amount == -1)
            return -1;

        position += amount;
        return amount;
    }

    public int write(byte[] buf, int offset, int length) {
        int amount = write(position, buf, offset, length);
        if (amount == -1)
            return -1;

        position += amount;
        return amount;
    }

    /**
     * The current value of the file pointer.
     */
    protected int position = 0;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A link-layer packet.
 *
 * @see nachos.machine.NetworkLink
 */
public class Packet {
    /**
     * Allocate a new packet to be sent, using the specified parameters.
     *
     * @param   dstLink         the destination link address.
     * @param   srcLink         the source link address.
     * @param   contents        the contents of the packet.
     */
    public Packet(int dstLink, int srcLink, byte[] contents)
        throws MalformedPacketException {
        // make sure the paramters are valid
        if (dstLink < 0 || dstLink >= linkAddressLimit ||
            srcLink < 0 || srcLink >= linkAddressLimit ||
            contents.length > maxContentsLength)
            throw new MalformedPacketException();

        this.dstLink = dstLink;
        this.srcLink = srcLink;
        this.contents = contents;

        packetBytes = new byte[headerLength + contents.length];

        packetBytes[0] = NetworkLink.networkID;
        packetBytes[1] = (byte) dstLink;
        packetBytes[2] = (byte) srcLink;
        packetBytes[3] = (byte) contents.length;

        // if java had subarrays, i'd use them. but System.arraycopy is ok...
        System.arraycopy(contents, 0, packetBytes, headerLength,
                         contents.length);
    }

    /**
     * Allocate a new packet using the specified array of bytes received from
     * the network.
     *
     * @param   packetBytes     the bytes making up this packet.
     */
    public Packet(byte[] packetBytes) throws MalformedPacketException {
        this.packetBytes = packetBytes;

        // make sure we have a valid header
        if (packetBytes.length < headerLength ||
            packetBytes[0] != NetworkLink.networkID ||
            packetBytes[1] < 0 || packetBytes[1] >= linkAddressLimit ||
            packetBytes[2] < 0 || packetBytes[2] >= linkAddressLimit ||
            packetBytes[3] < 0 || packetBytes[3] > packetBytes.length-4)
            throw new MalformedPacketException();

        dstLink = packetBytes[1];
        srcLink = packetBytes[2];

        contents = new byte[packetBytes[3]];
        System.arraycopy(packetBytes, headerLength, contents, 0,
                         contents.length);
    }

    /** This packet, as an array of bytes that can be sent on a network. */
    public byte[] packetBytes;
    /** The address of the destination link of this packet. */
    public int dstLink;
    /** The address of the source link of this packet. */
    public int srcLink;
    /** The contents of this packet, excluding the link-layer header. */
    public byte[] contents;

    /**
     * The number of bytes in a link-layer packet header. The header is
     * formatted as follows:
     *
     * <table>
     * <tr><td>offset</td><td>size</td><td>value</td></tr>
     * <tr><td>0</td><td>1</td><td>network ID (collision detecting)</td></tr>
     * <tr><td>1</td><td>1</td><td>destination link address</td></tr>
     * <tr><td>2</td><td>1</td><td>source link address</td></tr>
     * <tr><td>3</td><td>1</td><td>length of contents</td></tr>
     * </table>
     */
    public static final int headerLength = 4;
    /**
     * The maximum length, in bytes, of a packet that can be sent or received
     * on the network.
     */
    public static final int maxPacketLength = 32;
    /**
     * The maximum number of content bytes (not including the header). Note
     * that this is just <tt>maxPacketLength - headerLength</tt>.
     */
    public static final int maxContentsLength = maxPacketLength - headerLength;

    /**
     * The upper limit on Nachos link addresses. All link addresses fall
     * between <tt>0</tt> and <tt>linkAddressLimit - 1</tt>.
     */
    public static final int linkAddressLimit = 128;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

/**
 * The <tt>Processor</tt> class simulates a MIPS processor that supports a
 * subset of the R3000 instruction set. Specifically, the processor lacks all
 * coprocessor support, and can only execute in user mode. Address translation
 * information is accessed via the API. The API also allows a kernel to set an
 * exception handler to be called on any user mode exception.
 *
 * <p>
 * The <tt>Processor</tt> API is re-entrant, so a single simulated processor
 * can be shared by multiple user threads.
 *
 * <p>
 * An instance of a <tt>Processor</tt> also includes pages of physical memory
 * accessible to user programs, the size of which is fixed by the constructor.
 */
public final class Processor {
    /**
     * Allocate a new MIPS processor, with the specified amount of memory.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     * @param   numPhysPages    the number of pages of physical memory to
     *                          attach.
     */
    public Processor(Privilege privilege, int numPhysPages) {
        System.out.print(" processor");

        this.privilege = privilege;
        privilege.processor = new ProcessorPrivilege();

        Class<?> clsKernel = Lib.loadClass(Config.getString("Kernel.kernel"));
        Class<?> clsVMKernel = Lib.tryLoadClass("nachos.vm.VMKernel");

        usingTLB =
            (clsVMKernel != null && clsVMKernel.isAssignableFrom(clsKernel));

        this.numPhysPages = numPhysPages;

        for (int i=0; i<numUserRegisters; i++)
            registers[i] = 0;

        mainMemory = new byte[pageSize * numPhysPages];

        if (usingTLB) {
            translations = new TranslationEntry[tlbSize];
            for (int i=0; i<tlbSize; i++)
                translations[i] = new TranslationEntry();
        }
        else {
            translations = null;
        }
    }

    /**
     * Set the exception handler, called whenever a user exception occurs.
     *
     * <p>
     * When the exception handler is called, interrupts will be enabled, and
     * the CPU cause register will specify the cause of the exception (see the
     * <tt>exception<i>*</i></tt> constants).
     *
     * @param   exceptionHandler        the kernel exception handler.
     */
    public void setExceptionHandler(Runnable exceptionHandler) {
        this.exceptionHandler = exceptionHandler;
    }

    /**
     * Get the exception handler, set by the last call to
     * <tt>setExceptionHandler()</tt>.
     *
     * @return  the exception handler.
     */
    public Runnable getExceptionHandler() {
        return exceptionHandler;
    }

    /**
     * Start executing instructions at the current PC. Never returns.
     */
    public void run() {
        Lib.debug(dbgProcessor, "starting program in current thread");

        registers[regNextPC] = registers[regPC] + 4;

        Machine.autoGrader().runProcessor(privilege);

        Instruction inst = new Instruction();

        while (true) {
            try {
                inst.run();
            }
            catch (MipsException e) {
                e.handle();
            }

            privilege.interrupt.tick(false);
        }
    }

    /**
     * Read and return the contents of the specified CPU register.
     *
     * @param   number  the register to read.
     * @return  the value of the register.
     */
    public int readRegister(int number) {
        Lib.assertTrue(number >= 0 && number < numUserRegisters);

        return registers[number];
    }

    /**
     * Write the specified value into the specified CPU register.
     *
     * @param   number  the register to write.
     * @param   value   the value to write.
     */
    public void writeRegister(int number, int value) {
        Lib.assertTrue(number >= 0 && number < numUserRegisters);

        if (number != 0)
            registers[number] = value;
```

```
}

/**
 * Test whether this processor uses a software-managed TLB, or single-level
 * paging.
 *
 * <p>
 * If <tt>false</tt>, this processor directly supports single-level paging;
 * use <tt>setPageTable()</tt>.
 *
 * <p>
 * If <tt>true</tt>, this processor has a software-managed TLB;
 * use <tt>getTLBSize()</tt>, <tt>readTLBEntry()</tt>, and
 * <tt>writeTLBEntry()</tt>.
 *
 * <p>
 * Using a method associated with the wrong address translation mechanism
 * will result in an assertion failure.
 *
 * @return  <tt>true</tt> if this processor has a software-managed TLB.
 */
public boolean hasTLB() {
    return usingTLB;
}

/**
 * Get the current page table, set by the last call to setPageTable().
 *
 * @return  the current page table.
 */
public TranslationEntry[] getPageTable() {
    Lib.assertTrue(!usingTLB);

    return translations;
}

/**
 * Set the page table pointer. All further address translations will use
 * the specified page table. The size of the current address space will be
 * determined from the length of the page table array.
 *
 * @param   pageTable       the page table to use.
 */
public void setPageTable(TranslationEntry[] pageTable) {
    Lib.assertTrue(!usingTLB);

    this.translations = pageTable;
}

/**
 * Return the number of entries in this processor's TLB.
 *
 * @return  the number of entries in this processor's TLB.
 */
public int getTLBSize() {
    Lib.assertTrue(usingTLB);

    return tlbSize;
}

/**
 * Returns the specified TLB entry.
 *
 * @param   number  the index into the TLB.
 * @return  the contents of the specified TLB entry.
 */
public TranslationEntry readTLBEntry(int number) {
    Lib.assertTrue(usingTLB);
    Lib.assertTrue(number >= 0 && number < tlbSize);

    return new TranslationEntry(translations[number]);
}

/**
 * Fill the specified TLB entry.
 *
 * <p>
 * The TLB is fully associative, so the location of an entry within the TLB
 * does not affect anything.
 *
 * @param   number  the index into the TLB.
 * @param   entry   the new contents of the TLB entry.
 */
public void writeTLBEntry(int number, TranslationEntry entry) {
    Lib.assertTrue(usingTLB);
    Lib.assertTrue(number >= 0 && number < tlbSize);

    translations[number] = new TranslationEntry(entry);
}

/**
 * Return the number of pages of physical memory attached to this simulated
 * processor.
 *
 * @return  the number of pages of physical memory.
 */
public int getNumPhysPages() {
    return numPhysPages;
}

/**
 * Return a reference to the physical memory array. The size of this array
 * is <tt>pageSize * getNumPhysPages()</tt>.
 *
 * @return  the main memory array.
 */
public byte[] getMemory() {
    return mainMemory;
}

/**
 * Concatenate a page number and an offset into an address.
 *
 * @param   page    the page number. Must be between <tt>0</tt> and
 *                  <tt>(2<sup>32</sup> / pageSize) - 1</tt>.
 * @param   offset  the offset within the page. Must be between <tt>0</tt>
 *                  and
 *                  <tt>pageSize - 1</tt>.
 * @return  a 32-bit address consisting of the specified page and offset.
 */
public static int makeAddress(int page, int offset) {
    Lib.assertTrue(page >= 0 && page < maxPages);
    Lib.assertTrue(offset >= 0 && offset < pageSize);

    return (page * pageSize) | offset;
}

/**
 * Extract the page number component from a 32-bit address.
 *
```

```
     * @param   address the 32-bit address.
     * @return  the page number component of the address.
     */
    public static int pageFromAddress(int address) {
        return (int) (((long) address & 0xFFFFFFFFL) / pageSize);
    }

    /**
     * Extract the offset component from an address.
     *
     * @param   address the 32-bit address.
     * @return  the offset component of the address.
     */
    public static int offsetFromAddress(int address) {
        return (int) (((long) address & 0xFFFFFFFFL) % pageSize);
    }

    private void finishLoad() {
        delayedLoad(0, 0, 0);
    }

    /**
     * Translate a virtual address into a physical address, using either a
     * page table or a TLB. Check for alignment, make sure the virtual page is
     * valid, make sure a read-only page is not being written, make sure the
     * resulting physical page is valid, and then return the resulting physical
     * address.
     *
     * @param   vaddr   the virtual address to translate.
     * @param   size    the size of the memory reference (must be 1, 2, or 4).
     * @param   writing <tt>true</tt> if the memory reference is a write.
     * @return          the physical address.
     * @exception       MipsException   if a translation error occurred.
     */
    private int translate(int vaddr, int size, boolean writing)
        throws MipsException {
        if (Lib.test(dbgProcessor))
            System.out.println("\ttranslate vaddr=0x" + Lib.toHexString(vaddr)
                               + (writing ? ", write" : ", read..."));

        // check alignment
        if ((vaddr & (size-1)) != 0) {
            Lib.debug(dbgProcessor, "\t\talignment error");
            throw new MipsException(exceptionAddressError, vaddr);
        }

        // calculate virtual page number and offset from the virtual address
        int vpn = pageFromAddress(vaddr);
        int offset = offsetFromAddress(vaddr);

        TranslationEntry entry = null;

        // if not using a TLB, then the vpn is an index into the table
        if (!usingTLB) {
            if (translations == null || vpn >= translations.length ||
                translations[vpn] == null ||
                !translations[vpn].valid) {
                privilege.stats.numPageFaults++;
                Lib.debug(dbgProcessor, "\t\tpage fault");
                throw new MipsException(exceptionPageFault, vaddr);
            }

            entry = translations[vpn];
        }
        // else, look through all TLB entries for matching vpn
        else {
            for (int i=0; i<tlbSize; i++) {
                if (translations[i].valid && translations[i].vpn == vpn) {
                    entry = translations[i];
                    break;
                }
            }
            if (entry == null) {
                privilege.stats.numTLBMisses++;
                Lib.debug(dbgProcessor, "\t\tTLB miss");
                throw new MipsException(exceptionTLBMiss, vaddr);
            }
        }

        // check if trying to write a read-only page
        if (entry.readOnly && writing) {
            Lib.debug(dbgProcessor, "\t\tread-only exception");
            throw new MipsException(exceptionReadOnly, vaddr);
        }

        // check if physical page number is out of range
        int ppn = entry.ppn;
        if (ppn < 0 || ppn >= numPhysPages) {
            Lib.debug(dbgProcessor, "\t\tbad ppn");
            throw new MipsException(exceptionBusError, vaddr);
        }

        // set used and dirty bits as appropriate
        entry.used = true;
        if (writing)
            entry.dirty = true;

        int paddr = (ppn*pageSize) + offset;

        if (Lib.test(dbgProcessor))
            System.out.println("\t\tpaddr=0x" + Lib.toHexString(paddr));
        return paddr;
    }

    /**
     * Read </i>size</i> (1, 2, or 4) bytes of virtual memory at <i>vaddr</i>,
     * and return the result.
     *
     * @param   vaddr   the virtual address to read from.
     * @param   size    the number of bytes to read (1, 2, or 4).
     * @return          the value read.
     * @exception       MipsException   if a translation error occurred.
     */
    private int readMem(int vaddr, int size) throws MipsException {
        if (Lib.test(dbgProcessor))
            System.out.println("\treadMem vaddr=0x" + Lib.toHexString(vaddr)
                               + ", size=" + size);

        Lib.assertTrue(size==1 || size==2 || size==4);

        int value = Lib.bytesToInt(mainMemory, translate(vaddr, size, false),
                                   size);

        if (Lib.test(dbgProcessor))
            System.out.println("\t\tvalue read=0x" +
                               Lib.toHexString(value, size*2));

        return value;
    }
```

```
    /**
     * Write <i>value</i> to </i>size</i> (1, 2, or 4) bytes of virtual memory
     * starting at <i>vaddr</i>.
     *
     * @param   vaddr   the virtual address to write to.
     * @param   size    the number of bytes to write (1, 2, or 4).
     * @param   value   the value to store.
     * @exception       MipsException   if a translation error occurred.
     */
    private void writeMem(int vaddr, int size, int value)
        throws MipsException {
        if (Lib.test(dbgProcessor))
            System.out.println("\twriteMem vaddr=0x" + Lib.toHexString(vaddr)
                               + ", size=" + size + ", value=0x"
                               + Lib.toHexString(value, size*2));

        Lib.assertTrue(size==1 || size==2 || size==4);

        Lib.bytesFromInt(mainMemory, translate(vaddr, size, true), size,
                         value);
    }

    /**
     * Complete the in progress delayed load and scheduled a new one.
     *
     * @param   nextLoadTarget  the target register of the new load.
     * @param   nextLoadValue   the value to be loaded into the new target.
     * @param   nextLoadMask    the mask specifying which bits in the new
     *                          target are to be overwritten. If a bit in
     *                          <tt>nextLoadMask</tt> is 0, then the
     *                          corresponding bit of register
     *                          <tt>nextLoadTarget</tt> will not be written.
     */
    private void delayedLoad(int nextLoadTarget, int nextLoadValue,
                             int nextLoadMask) {
        // complete previous delayed load, if not modifying r0
        if (loadTarget != 0) {
            int savedBits = registers[loadTarget] & ~loadMask;
            int newBits = loadValue & loadMask;
            registers[loadTarget] = savedBits | newBits;
        }

        // schedule next load
        loadTarget = nextLoadTarget;
        loadValue = nextLoadValue;
        loadMask = nextLoadMask;
    }

    /**
     * Advance the PC to the next instruction.
     *
     * <p>
     * Transfer the contents of the nextPC register into the PC register, and
     * then add 4 to the value in the nextPC register. Same as
     * <tt>advancePC(readRegister(regNextPC)+4)</tt>.
     *
     * <p>
     * Use after handling a syscall exception so that the processor will move
     * on to the next instruction.
     */
    public void advancePC() {
        advancePC(registers[regNextPC]+4);
    }

    /**
     * Transfer the contents of the nextPC register into the PC register, and
     * then write the nextPC register.
     *
     * @param   nextPC  the new value of the nextPC register.
     */
    private void advancePC(int nextPC) {
        registers[regPC] = registers[regNextPC];
        registers[regNextPC] = nextPC;
    }

    /** Caused by a syscall instruction. */
    public static final int exceptionSyscall = 0;
    /** Caused by an access to an invalid virtual page. */
    public static final int exceptionPageFault = 1;
    /** Caused by an access to a virtual page not mapped by any TLB entry. */
    public static final int exceptionTLBMiss = 2;
    /** Caused by a write access to a read-only virtual page. */
    public static final int exceptionReadOnly = 3;
    /** Caused by an access to an invalid physical page. */
    public static final int exceptionBusError = 4;
    /** Caused by an access to a misaligned virtual address. */
    public static final int exceptionAddressError = 5;
    /** Caused by an overflow by a signed operation. */
    public static final int exceptionOverflow = 6;
    /** Caused by an attempt to execute an illegal instruction. */
    public static final int exceptionIllegalInstruction = 7;

    /** The names of the CPU exceptions. */
    public static final String exceptionNames[] = {
        "syscall      ",
        "page fault   ",
        "TLB miss     ",
        "read-only    ",
        "bus error    ",
        "address error",
        "overflow     ",
        "illegal inst "
    };

    /** Index of return value register 0. */
    public static final int regV0 = 2;
    /** Index of return value register 1. */
    public static final int regV1 = 3;
    /** Index of argument register 0. */
    public static final int regA0 = 4;
    /** Index of argument register 1. */
    public static final int regA1 = 5;
    /** Index of argument register 2. */
    public static final int regA2 = 6;
    /** Index of argument register 3. */
    public static final int regA3 = 7;
    /** Index of the stack pointer register. */
    public static final int regSP = 29;
    /** Index of the return address register. */
    public static final int regRA = 31;
    /** Index of the low register, used for multiplication and division. */
    public static final int regLo = 32;
    /** Index of the high register, used for multiplication and division. */
    public static final int regHi = 33;
    /** Index of the program counter register. */
    public static final int regPC = 34;
    /** Index of the next program counter register. */
    public static final int regNextPC = 35;
    /** Index of the exception cause register. */
    public static final int regCause = 36;
```

```
    /** Index of the exception bad virtual address register. */
    public static final int regBadVAddr = 37;

    /** The total number of software-accessible CPU registers. */
    public static final int numUserRegisters = 38;

    /** Provides privilege to this processor. */
    private Privilege privilege;

    /** MIPS registers accessible to the kernel. */
    private int registers[] = new int[numUserRegisters];

    /** The registered target of the delayed load currently in progress. */
    private int loadTarget = 0;
    /** The bits to be modified by the delayed load currently in progress. */
    private int loadMask;
    /** The value to be loaded by the delayed load currently in progress. */
    private int loadValue;

    /** <tt>true</tt> if using a software-managed TLB. */
    private boolean usingTLB;
    /** Number of TLB entries. */
    private int tlbSize = 4;
    /**
     * Either an associative or direct-mapped set of translation entries,
     * depending on whether there is a TLB.
     */
    private TranslationEntry[] translations;

    /** Size of a page, in bytes. */
    public static final int pageSize = 0x400;
    /** Number of pages in a 32-bit address space. */
    public static final int maxPages = (int) (0x100000000L / pageSize);
    /** Number of physical pages in memory. */
    private int numPhysPages;
    /** Main memory for user programs. */
    private byte[] mainMemory;

    /** The kernel exception handler, called on every user exception. */
    private Runnable exceptionHandler = null;

    private static final char dbgProcessor = 'p';
    private static final char dbgDisassemble = 'm';
    private static final char dbgFullDisassemble = 'M';

    private class ProcessorPrivilege implements Privilege.ProcessorPrivilege {
        public void flushPipe() {
            finishLoad();
        }
    }

    private class MipsException extends Exception {
        public MipsException(int cause) {
            Lib.assertTrue(cause >= 0 && cause < exceptionNames.length);

            this.cause = cause;
        }

        public MipsException(int cause, int badVAddr) {
            this(cause);

            hasBadVAddr = true;
            this.badVAddr = badVAddr;
        }

        public void handle() {
            writeRegister(regCause, cause);

            if (hasBadVAddr)
                writeRegister(regBadVAddr, badVAddr);

            if (Lib.test(dbgDisassemble) || Lib.test(dbgFullDisassemble))
                System.out.println("exception: " + exceptionNames[cause]);

            finishLoad();

            Lib.assertTrue(exceptionHandler != null);

            // autograder might not want kernel to know about this exception
            if (!Machine.autoGrader().exceptionHandler(privilege))
                return;

            exceptionHandler.run();
        }

        private boolean hasBadVAddr = false;
        private int cause, badVAddr;
    }

    private class Instruction {
        public void run() throws MipsException {
            // hopefully this looks familiar to 152 students?
            fetch();
            decode();
            execute();
            writeBack();
        }

        private boolean test(int flag) {
            return Lib.test(flag, flags);
        }

        private void fetch() throws MipsException {
            if ((Lib.test(dbgDisassemble) && !Lib.test(dbgProcessor)) ||
                Lib.test(dbgFullDisassemble))
                System.out.print("PC=0x" + Lib.toHexString(registers[regPC])
                                 + "\t");

            value = readMem(registers[regPC], 4);
        }

        private void decode() {
            op = Lib.extract(value, 26, 6);
            rs = Lib.extract(value, 21, 5);
            rt = Lib.extract(value, 16, 5);
            rd = Lib.extract(value, 11, 5);
            sh = Lib.extract(value, 6, 5);
            func = Lib.extract(value, 0, 6);
            target = Lib.extract(value, 0, 26);
            imm = Lib.extend(value, 0, 16);

            Mips info;
            switch (op) {
            case 0:
                info = Mips.specialtable[func];
                break;
            case 1:
                info = Mips.regimmtable[rt];
                break;
            default:
```

```
            info = Mips.optable[op];
            break;
        }

        operation = info.operation;
        name = info.name;
        format = info.format;
        flags = info.flags;

        mask = 0xFFFFFFFF;
        branch = true;

        // get memory access size
        if (test(Mips.SIZEB))
            size = 1;
        else if (test(Mips.SIZEH))
            size = 2;
        else if (test(Mips.SIZEW))
            size = 4;
        else
            size = 0;

        // get nextPC
        nextPC = registers[regNextPC]+4;

        // get dstReg
        if (test(Mips.DSTRA))
            dstReg = regRA;
        else if (format == Mips.IFMT)
            dstReg = rt;
        else if (format == Mips.RFMT)
            dstReg = rd;
        else
            dstReg = -1;

        // get jtarget
        if (format == Mips.RFMT)
            jtarget = registers[rs];
        else if (format == Mips.IFMT)
            jtarget = registers[regNextPC] + (imm<<2);
        else if (format == Mips.JFMT)
            jtarget = (registers[regNextPC]&0xF0000000) | (target<<2);
        else
            jtarget = -1;

        // get imm
        if (test(Mips.UNSIGNED)) {
            imm &= 0xFFFF;
        }

        // get addr
        addr = registers[rs] + imm;

        // get src1
        if (test(Mips.SRC1SH))
            src1 = sh;
        else
            src1 = registers[rs];

        // get src2
        if (test(Mips.SRC2IMM))
            src2 = imm;
        else
            src2 = registers[rt];

        if (test(Mips.UNSIGNED)) {
            src1 &= 0xFFFFFFFFL;
            src2 &= 0xFFFFFFFFL;
        }

        if (Lib.test(dbgDisassemble) || Lib.test(dbgFullDisassemble))
            print();
    }

    private void print() {
        if (Lib.test(dbgDisassemble) && Lib.test(dbgProcessor) &&
            !Lib.test(dbgFullDisassemble))
            System.out.print("PC=0x" + Lib.toHexString(registers[regPC])
                             + "\t");

        if (operation == Mips.INVALID) {
            System.out.print("invalid: op=" + Lib.toHexString(op, 2) +
                             " rs=" + Lib.toHexString(rs, 2) +
                             " rt=" + Lib.toHexString(rt, 2) +
                             " rd=" + Lib.toHexString(rd, 2) +
                             " sh=" + Lib.toHexString(sh, 2) +
                             " func=" + Lib.toHexString(func, 2) +
                             "\n");
            return;
        }

        int spaceIndex = name.indexOf(' ');
        Lib.assertTrue(spaceIndex!=-1 && spaceIndex==name.lastIndexOf(' '));

        String instname = name.substring(0, spaceIndex);
        char[] args = name.substring(spaceIndex+1).toCharArray();

        System.out.print(instname + "\t");

        int minCharsPrinted = 0, maxCharsPrinted = 0;

        for (int i=0; i<args.length; i++) {
            switch (args[i]) {
            case Mips.RS:
                System.out.print("$" + rs);
                minCharsPrinted += 2;
                maxCharsPrinted += 3;

                if (Lib.test(dbgFullDisassemble)) {
                    System.out.print("#0x" +
                                     Lib.toHexString(registers[rs]));
                    minCharsPrinted += 11;
                    maxCharsPrinted += 11;
                }
                break;
            case Mips.RT:
                System.out.print("$" + rt);
                minCharsPrinted += 2;
                maxCharsPrinted += 3;

                if (Lib.test(dbgFullDisassemble) &&
                    (i!=0 || !test(Mips.DST)) &&
                    !test(Mips.DELAYEDLOAD)) {
                    System.out.print("#0x" +
                                     Lib.toHexString(registers[rt]));
                    minCharsPrinted += 11;
                    maxCharsPrinted += 11;
                }
                break;
            case Mips.RETURNADDRESS:
```

```
            if (rd == 31)
                continue;
        case Mips.RD:
            System.out.print("$" + rd);
            minCharsPrinted += 2;
            maxCharsPrinted += 3;
            break;
        case Mips.IMM:
            System.out.print(imm);
            minCharsPrinted += 1;
            maxCharsPrinted += 6;
            break;
        case Mips.SHIFTAMOUNT:
            System.out.print(sh);
            minCharsPrinted += 1;
            maxCharsPrinted += 2;
            break;
        case Mips.ADDR:
            System.out.print(imm + "($" + rs);
            minCharsPrinted += 4;
            maxCharsPrinted += 5;

            if (Lib.test(dbgFullDisassemble)) {
                System.out.print("#0x" +
                                 Lib.toHexString(registers[rs]));
                minCharsPrinted += 11;
                maxCharsPrinted += 11;
            }

            System.out.print(")");
            break;
        case Mips.TARGET:
            System.out.print("0x" + Lib.toHexString(jtarget));
            minCharsPrinted += 10;
            maxCharsPrinted += 10;
            break;
        default:
            Lib.assertTrue(false);
        }
        if (i+1 < args.length) {
            System.out.print(", ");
            minCharsPrinted += 2;
            maxCharsPrinted += 2;
        }
        else {
            // most separation possible is tsi, 5+1+1=7,
            // thankfully less than 8 (makes this possible)
            Lib.assertTrue(maxCharsPrinted-minCharsPrinted < 8);
            // longest string is stj, which is 40-42 chars w/ -d M;
            // go for 48
            while ((minCharsPrinted%8) != 0) {
                System.out.print(" ");
                minCharsPrinted++;
                maxCharsPrinted++;
            }
            while (minCharsPrinted < 48) {
                System.out.print("\t");
                minCharsPrinted += 8;
            }
        }
    }

    if (Lib.test(dbgDisassemble) && Lib.test(dbgProcessor) &&
        !Lib.test(dbgFullDisassemble))
        System.out.print("\n");
    }

    private void execute() throws MipsException {
        int value;
        int preserved;

        switch (operation) {
        case Mips.ADD:
            dst = src1 + src2;
            break;
        case Mips.SUB:
            dst = src1 - src2;
            break;
        case Mips.MULT:
            dst = src1 * src2;
            registers[regLo] = (int) Lib.extract(dst, 0, 32);
            registers[regHi] = (int) Lib.extract(dst, 32, 32);
            break;
        case Mips.DIV:
            try {
                registers[regLo] = (int) (src1 / src2);
                registers[regHi] = (int) (src1 % src2);
                if (registers[regLo]*src2 + registers[regHi] != src1)
                    throw new ArithmeticException();
            }
            catch (ArithmeticException e) {
                throw new MipsException(exceptionOverflow);
            }
            break;

        case Mips.SLL:
            dst = src2 << (src1&0x1F);
            break;
        case Mips.SRA:
            dst = src2 >> (src1&0x1F);
            break;
        case Mips.SRL:
            dst = src2 >>> (src1&0x1F);
            break;

        case Mips.SLT:
            dst = (src1<src2) ? 1 : 0;
            break;

        case Mips.AND:
            dst = src1 & src2;
            break;
        case Mips.OR:
            dst = src1 | src2;
            break;
        case Mips.NOR:
            dst = ~(src1 | src2);
            break;
        case Mips.XOR:
            dst = src1 ^ src2;
            break;
        case Mips.LUI:
            dst = imm << 16;
            break;

        case Mips.BEQ:
            branch = (src1 == src2);
            break;
        case Mips.BNE:
            branch = (src1 != src2);
```

```
    break;
case Mips.BGEZ:
    branch = (src1 >= 0);
    break;
case Mips.BGTZ:
    branch = (src1 > 0);
    break;
case Mips.BLEZ:
    branch = (src1 <= 0);
    break;
case Mips.BLTZ:
    branch = (src1 < 0);
    break;

case Mips.JUMP:
    break;

case Mips.MFLO:
    dst = registers[regLo];
    break;
case Mips.MFHI:
    dst = registers[regHi];
    break;
case Mips.MTLO:
    registers[regLo] = (int) src1;
    break;
case Mips.MTHI:
    registers[regHi] = (int) src1;
    break;

case Mips.SYSCALL:
    throw new MipsException(exceptionSyscall);

case Mips.LOAD:
    value = readMem(addr, size);

    if (!test(Mips.UNSIGNED))
        dst = Lib.extend(value, 0, size*8);
    else
        dst = value;

    break;

case Mips.LWL:
    value = readMem(addr&~0x3, 4);

    // LWL shifts the input left so the addressed byte is highest
    preserved = (3-(addr&0x3))*8;   // number of bits to preserve
    mask = -1 << preserved;         // preserved bits are 0 in mask
    dst = value << preserved;       // shift input to correct place
    addr &= ~0x3;

    break;

case Mips.LWR:
    value = readMem(addr&~0x3, 4);

    // LWR shifts the input right so the addressed byte is lowest
    preserved = (addr&0x3)*8;       // number of bits to preserve
    mask = -1 >>> preserved;        // preserved bits are 0 in mask
    dst = value >>> preserved;      // shift input to correct place
    addr &= ~0x3;

    break;

case Mips.STORE:
    writeMem(addr, size, (int) src2);
    break;

case Mips.SWL:
    value = readMem(addr&~0x3, 4);

    // SWL shifts highest order byte into the addressed position
    preserved = (3-(addr&0x3))*8;
    mask = -1 >>> preserved;
    dst = src2 >>> preserved;

    // merge values
    dst = (dst & mask) | (value & ~mask);

    writeMem(addr&~0x3, 4, (int) dst);
    break;

case Mips.SWR:
    value = readMem(addr&~0x3, 4);

    // SWR shifts the lowest order byte into the addressed position
    preserved = (addr&0x3)*8;
    mask = -1 << preserved;
    dst = src2 << preserved;

    // merge values
    dst = (dst & mask) | (value & ~mask);

    writeMem(addr&~0x3, 4, (int) dst);
    break;

case Mips.UNIMPL:
    System.err.println("Warning: encountered unimplemented inst");

case Mips.INVALID:
    throw new MipsException(exceptionIllegalInstruction);

default:
    Lib.assertNotReached();
}
}

private void writeBack() throws MipsException {
    // if instruction is signed, but carry bit !+ sign bit, throw
    if (test(Mips.OVERFLOW) && Lib.test(dst,31) != Lib.test(dst,32))
        throw new MipsException(exceptionOverflow);

    if (test(Mips.DELAYEDLOAD))
        delayedLoad(dstReg, (int) dst, mask);
    else
        finishLoad();

    if (test(Mips.LINK))
        dst = nextPC;

    if (test(Mips.DST) && dstReg != 0)
        registers[dstReg] = (int) dst;

    if ((test(Mips.DST) || test(Mips.DELAYEDLOAD)) && dstReg != 0) {
        if (Lib.test(dbgFullDisassemble)) {
            System.out.print("#0x" + Lib.toHexString((int) dst));
            if (test(Mips.DELAYEDLOAD))
                System.out.print(" (delayed load)");
        }
```

```
        }

        if (test(Mips.BRANCH) && branch) {
            nextPC = jtarget;
        }

        advancePC(nextPC);

        if ((Lib.test(dbgDisassemble) && !Lib.test(dbgProcessor)) ||
            Lib.test(dbgFullDisassemble))
            System.out.print("\n");
    }

    // state used to execute a single instruction
    int value, op, rs, rt, rd, sh, func, target, imm;
    int operation, format, flags;
    String name;

    int size;
    int addr, nextPC, jtarget, dstReg;
    long src1, src2, dst;
    int mask;
    boolean branch;
}

private static class Mips {
    Mips() {
    }

    Mips(int operation, String name) {
        this.operation = operation;
        this.name = name;
    }

    Mips(int operation, String name, int format, int flags) {
        this(operation, name);
        this.format = format;
        this.flags = flags;
    }

    int operation = INVALID;
    String name = "invalid ";
    int format;
    int flags;

    // operation types
    static final int
        INVALID     = 0,
        UNIMPL      = 1,
        ADD         = 2,
        SUB         = 3,
        MULT        = 4,
        DIV         = 5,
        SLL         = 6,
        SRA         = 7,
        SRL         = 8,
        SLT         = 9,
        AND         = 10,
        OR          = 11,
        NOR         = 12,
        XOR         = 13,
        LUI         = 14,
        MFLO        = 21,
        MFHI        = 22,
        MTLO        = 23,
        MTHI        = 24,
        JUMP        = 25,
        BEQ         = 26,
        BNE         = 27,
        BLEZ        = 28,
        BGTZ        = 29,
        BLTZ        = 30,
        BGEZ        = 31,
        SYSCALL     = 32,
        LOAD        = 33,
        LWL         = 36,
        LWR         = 37,
        STORE       = 38,
        SWL         = 39,
        SWR         = 40,
        MAX         = 40;

    static final int
        IFMT = 1,
        JFMT = 2,
        RFMT = 3;

    static final int
        DST         = 0x00000001,
        DSTRA       = 0x00000002,
        OVERFLOW    = 0x00000004,
        SRC1SH      = 0x00000008,
        SRC2IMM     = 0x00000010,
        UNSIGNED    = 0x00000020,
        LINK        = 0x00000040,
        DELAYEDLOAD = 0x00000080,
        SIZEB       = 0x00000100,
        SIZEH       = 0x00000200,
        SIZEW       = 0x00000400,
        BRANCH      = 0x00000800;

    static final char
        RS                  = 's',
        RT                  = 't',
        RD                  = 'd',
        IMM                 = 'i',
        SHIFTAMOUNT         = 'h',
        ADDR                = 'a',  // imm(rs)
        TARGET              = 'j',
        RETURNADDRESS       = 'r';  // rd, or none if rd=31; can't be last

    static final Mips[] optable = {
        new Mips(),                                             // special
        new Mips(),                                             // reg-imm
        new Mips(JUMP,      "j j",          JFMT, BRANCH),
        new Mips(JUMP,      "jal j",        JFMT, BRANCH|LINK|DST|DSTRA),
        new Mips(BEQ,       "beq stj",      IFMT, BRANCH),
        new Mips(BNE,       "bne stj",      IFMT, BRANCH),
        new Mips(BLEZ,      "blez sj",      IFMT, BRANCH),
        new Mips(BGTZ,      "bgtz sj",      IFMT, BRANCH),
        new Mips(ADD,       "addi tsi",     IFMT, DST|SRC2IMM|OVERFLOW),
        new Mips(ADD,       "addiu tsi",    IFMT, DST|SRC2IMM),
        new Mips(SLT,       "slti tsi",     IFMT, DST|SRC2IMM),
        new Mips(SLT,       "sltiu tsi",    IFMT, DST|SRC2IMM|UNSIGNED),
        new Mips(AND,       "andi tsi",     IFMT, DST|SRC2IMM|UNSIGNED),
        new Mips(OR,        "ori tsi",      IFMT, DST|SRC2IMM|UNSIGNED),
        new Mips(XOR,       "xori tsi",     IFMT, DST|SRC2IMM|UNSIGNED),
        new Mips(LUI,       "lui ti",       IFMT, DST|SRC2IMM|UNSIGNED),
        new Mips(),
        new Mips(),
```

```
        new Mips(),
        new Mips(),
        new Mips(BEQ,       "beql stj",     IFMT, BRANCH),
        new Mips(BNE,       "bnel stj",     IFMT, BRANCH),
        new Mips(BLEZ,      "blezl sj",     IFMT, BRANCH),
        new Mips(BGTZ,      "bgtzl sj",     IFMT, BRANCH),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(LOAD,      "lb ta",        IFMT, DELAYEDLOAD|SIZEB),
        new Mips(LOAD,      "lh ta",        IFMT, DELAYEDLOAD|SIZEH),
        new Mips(LWL,       "lwl ta",       IFMT, DELAYEDLOAD),
        new Mips(LOAD,      "lw ta",        IFMT, DELAYEDLOAD|SIZEW),
        new Mips(LOAD,      "lbu ta",    IFMT, DELAYEDLOAD|SIZEB|UNSIGNED),
        new Mips(LOAD,      "lhu ta",    IFMT, DELAYEDLOAD|SIZEH|UNSIGNED),
        new Mips(LWR,       "lwr ta",       IFMT, DELAYEDLOAD),
        new Mips(),
        new Mips(STORE,     "sb ta",        IFMT, SIZEB),
        new Mips(STORE,     "sh ta",        IFMT, SIZEH),
        new Mips(SWL,       "swl ta",       IFMT, 0),
        new Mips(STORE,     "sw ta",        IFMT, SIZEW),
        new Mips(),
        new Mips(),
        new Mips(SWR,       "swr ta",       IFMT, 0),
        new Mips(),
        new Mips(UNIMPL,    "ll "),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(UNIMPL,    "sc "),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
    };

    static final Mips[] specialtable = {
        new Mips(SLL,       "sll dth",      RFMT, DST|SRC1SH),
        new Mips(),
        new Mips(SRL,       "srl dth",      RFMT, DST|SRC1SH),
        new Mips(SRA,       "sra dth",      RFMT, DST|SRC1SH),
        new Mips(SLL,       "sllv dts",     RFMT, DST),
        new Mips(),
        new Mips(SRL,       "srlv dts",     RFMT, DST),
        new Mips(SRA,       "srav dts",     RFMT, DST),
        new Mips(JUMP,      "jr s",         RFMT, BRANCH),
        new Mips(JUMP,      "jalr rs",      RFMT, BRANCH|LINK|DST),
        new Mips(),
        new Mips(),
        new Mips(SYSCALL,   "syscall "),
        new Mips(UNIMPL,    "break "),
        new Mips(),
        new Mips(UNIMPL,    "sync "),
        new Mips(MFHI,      "mfhi d",       RFMT, DST),
        new Mips(MTHI,      "mthi s",       RFMT, 0),
        new Mips(MFLO,      "mflo d",       RFMT, DST),
        new Mips(MTLO,      "mtlo s",       RFMT, 0),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(MULT,      "mult st",      RFMT, 0),
        new Mips(MULT,      "multu st",     RFMT, UNSIGNED),
        new Mips(DIV,       "div st",       RFMT, 0),
        new Mips(DIV,       "divu st",      RFMT, UNSIGNED),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(ADD,       "add dst",      RFMT, DST|OVERFLOW),
        new Mips(ADD,       "addu dst",     RFMT, DST),
        new Mips(SUB,       "sub dst",      RFMT, DST|OVERFLOW),
        new Mips(SUB,       "subu dst",     RFMT, DST),
        new Mips(AND,       "and dst",      RFMT, DST),
        new Mips(OR,        "or dst",       RFMT, DST),
        new Mips(XOR,       "xor dst",      RFMT, DST),
        new Mips(NOR,       "nor dst",      RFMT, DST),
        new Mips(),
        new Mips(),
        new Mips(SLT,       "slt dst",      RFMT, DST),
        new Mips(SLT,       "sltu dst",     RFMT, DST|UNSIGNED),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
    };

    static final Mips[] regimmtable = {
        new Mips(BLTZ,      "bltz sj",      IFMT, BRANCH),
        new Mips(BGEZ,      "bgez sj",      IFMT, BRANCH),
        new Mips(BLTZ,      "bltzl sj",     IFMT, BRANCH),
        new Mips(BGEZ,      "bgezl sj",     IFMT, BRANCH),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
        new Mips(),
```

```
            new Mips(),
            new Mips(),
            new Mips(BLTZ,      "bltzal sj",    IFMT, BRANCH|LINK|DST|DSTRA),
            new Mips(BGEZ,      "bgezal sj",    IFMT, BRANCH|LINK|DST|DSTRA),
            new Mips(BLTZ,      "bltzlal sj",   IFMT, BRANCH|LINK|DST|DSTRA),
            new Mips(BGEZ,      "bgezlal sj",   IFMT, BRANCH|LINK|DST|DSTRA),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips(),
            new Mips()
        };
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A set of controls that can be used by a rider controller. Each rider uses a
 * distinct <tt>RiderControls</tt> object.
 */
public interface RiderControls {
    /**
     * Return the number of floors in the elevator bank. If <i>n</i> is the
     * number of floors in the bank, then the floors are numbered <i>0</i>
     * (the ground floor) through <i>n - 1</i> (the top floor).
     *
     * @return  the number of floors in the bank.
     */
    public int getNumFloors();

    /**
     * Return the number of elevators in the elevator bank. If <i>n</i> is the
     * number of elevators in the bank, then the elevators are numbered
     * <i>0</i> through <i>n - 1</i>.
     *
     * @return  the numbe rof elevators in the bank.
     */
    public int getNumElevators();

    /**
     * Set the rider's interrupt handler. This handler will be called when the
     * rider observes an event.
     *
     * @param   handler the rider's interrupt handler.
     */
    public void setInterruptHandler(Runnable handler);

    /**
     * Return the current location of the rider. If the rider is in motion,
     * the returned value will be within one of the exact location.
     *
     * @return  the floor the rider is on.
     */
    public int getFloor();

    /**
     * Return an array specifying the sequence of floors at which this rider
     * has successfully exited an elevator. This array naturally does not
     * count the floor the rider started on, nor does it count floors where
     * the rider is in the elevator and does not exit.
     *
     * @return  an array specifying the floors this rider has visited.
     */
    public int[] getFloors();

    /**
     * Return the indicated direction of the specified elevator, set by
     * <tt>ElevatorControls.setDirectionDisplay()</tt>.
     *
     * @param   elevator        the elevator to check the direction of.
     * @return  the displayed direction for the elevator.
     *
     * @see     nachos.machine.ElevatorControls#setDirectionDisplay
     */
    public int getDirectionDisplay(int elevator);

    /**
     * Press a direction button. If <tt>up</tt> is <tt>true</tt>, invoke
     * <tt>pressUpButton()</tt>; otherwise, invoke <tt>pressDownButton()</tt>.
     *
     * @param   up      <tt>true</tt> to press the up button, <tt>false</tt> to
     *                  press the down button.
     * @return  the return value of <tt>pressUpButton()</tt> or of
     *          <tt>pressDownButton()</tt>.
     */
    public boolean pressDirectionButton(boolean up);

    /**
     * Press the up button. The rider must not be on the top floor and must not
     * be inside an elevator. If an elevator is on the same floor as this
     * rider, has the doors open, and says it is going up, does nothing and
     * returns <tt>false</tt>.
     *
     * @return  <tt>true</tt> if the button event was sent to the elevator
     *          controller.
     */
    public boolean pressUpButton();

    /**
     * Press the down button. The rider must not be on the bottom floor and
     * must not be inside an elevator. If an elevator is on the same floor as
     * as this rider, has the doors open, and says it is going down, does
     * nothing and returns <tt>false</tt>.
     *
     * @return  <tt>true</tt> if the button event was sent to the elevator
     *          controller.
     */
    public boolean pressDownButton();

    /**
     * Enter an elevator. A rider cannot enter an elevator if its doors are not
     * open at the same floor, or if the elevator is full. The rider must not
     * already be in an elevator.
     *
     * @param   elevator        the elevator to enter.
     * @return  <tt>true</tt> if the rider successfully entered the elevator.
     */
    public boolean enterElevator(int elevator);

    /**
     * Press a floor button. The rider must be inside an elevator. If the
     * elevator already has its doors open on <tt>floor</tt>, does nothing and
     * returns <tt>false</tt>.
     *
     * @param   floor   the button to press.
     * @return  <tt>true</tt> if the button event was sent to the elevator
     *          controller.
     */
    public boolean pressFloorButton(int floor);

    /**
     * Exit the elevator. A rider cannot exit the elevator if its doors are not
     * open on the requested floor. The rider must already be in an elevator.
     *
     * @param   floor   the floor to exit on.
     * @return  <tt>true</tt> if the rider successfully got off the elevator.
     */
    public boolean exitElevator(int floor);

    /**
     * Call when the rider is finished.
     */
```

```
    public void finish();

    /**
     * Return the next event in the event queue. Note that there may be
     * multiple events pending when a rider interrupt occurs, so this
     * method should be called repeatedly until it returns <tt>null</tt>.
     *
     * @return  the next event, or <tt>null</tt> if no further events are
     * currently pending.
     */
    public RiderEvent getNextEvent();
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * An event that affects rider software. If a rider is outside the elevators,
 * it will only receive events on the same floor as the rider. If a rider is
 * inside an elevator, it will only receive events pertaining to that elevator.
 */
public final class RiderEvent {
    public RiderEvent(int event, int floor, int elevator, int direction) {
        this.event = event;
        this.floor = floor;
        this.elevator = elevator;
        this.direction = direction;
    }

    /** The event identifier. Refer to the <i>event*</i> constants. */
    public final int event;
    /** The floor pertaining to the event, or -1 if not applicable. */
    public final int floor;
    /** The elevator pertaining to the event, or -1 if not applicable. */
    public final int elevator;
    /** The direction display of the elevator (neither if not applicable). */
    public final int direction;

    /** An elevator's doors have opened. */
    public static final int eventDoorsOpened = 0;
    /** An elevator's doors were open and its direction display changed. */
    public static final int eventDirectionChanged = 1;
    /** An elevator's doors have closed. */
    public static final int eventDoorsClosed = 2;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

/**
 * A single rider. Each rider accesses the elevator bank through an
 * instance of <tt>RiderControls</tt>.
 */
public interface RiderInterface extends Runnable {
    /**
     * Initialize this rider. The rider will access the elevator bank through
     * <i>controls</i>, and the rider will make stops at different floors as
     * specified in <i>stops</i>. This method should return immediately after
     * this rider is initialized, but not until the interrupt handler is
     * set. The rider will start receiving events after this method returns,
     * potentially before <tt>run()</tt> is called.
     *
     * @param   controls        the rider's interface to the elevator bank. The
     *                          rider must not attempt to access the elevator
     *                          bank in <i>any</i> other way.
     * @param   stops           an array of stops the rider should make; see
     *                          below.
     */
    public void initialize(RiderControls controls, int[] stops);

    /**
     * Cause the rider to use the provided controls to make the stops specified
     * in the constructor. The rider should stop at each of the floors in
     * <i>stops</i>, an array of floor numbers. The rider should <i>only</i>
     * make the specified stops.
     *
     * <p>
     * For example, suppose the rider uses <i>controls</i> to determine that
     * it is initially on floor 1, and suppose the stops array contains two
     * elements: { 0, 2 }. Then the rider should get on an elevator, get off
     * on floor 0, get on an elevator, and get off on floor 2, pushing buttons
     * as necessary.
     *
     * <p>
     * This method should not return, but instead should call
     * <tt>controls.finish()</tt> when the rider is finished.
     */
    public void run();

    /** Indicates an elevator intends to move down. */
    public static final int dirDown = -1;
    /** Indicates an elevator intends not to move. */
    public static final int dirNeither = 0;
    /** Indicates an elevator intends to move up. */
    public static final int dirUp = 1;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

/**
 * A serial console can be used to send and receive characters. Only one
 * character may be sent at a time, and only one character may be received at a
 * time.
 */

public interface SerialConsole {
    /**
     * Set this console's receive and send interrupt handlers.
     *
     * <p>
     * The receive interrupt handler is called every time another byte arrives
     * and can be read using <tt>readByte()</tt>.
     *
     * <p>
     * The send interrupt handler is called every time a byte sent with
     * <tt>writeByte()</tt> is finished being sent. This means that another
     * byte can be sent.
     *
     * @param   receiveInterruptHandler the callback to call when a byte
     *                                  arrives.
     * @param   sendInterruptHandler    the callback to call when another byte
     *                                  can be sent.
     */
    public void setInterruptHandlers(Runnable receiveInterruptHandler,
                                     Runnable sendInterruptHandler);

    /**
     * Return the next unsigned byte received (in the range <tt>0</tt> through
     * <tt>255</tt>).
     *
     * @return  the next byte read, or -1 if no byte is available.
     */
    public int  readByte();

    /**
     * Send another byte. If a byte is already being sent, the result is not
     * defined.
     *
     * @param   value   the byte to be sent (the upper 24 bits are ignored).
     */
    public void writeByte(int value);
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

import java.io.IOException;

/**
 * A text-based console that uses System.in and System.out.
 */
public class StandardConsole implements SerialConsole {
    /**
     * Allocate a new standard console.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public StandardConsole(Privilege privilege) {
        System.out.print(" console");

        this.privilege = privilege;

        receiveInterrupt = new Runnable() {
                public void run() { receiveInterrupt(); }
            };

        sendInterrupt = new Runnable() {
                public void run() { sendInterrupt(); }
            };

        scheduleReceiveInterrupt();
    }

    public final void setInterruptHandlers(Runnable receiveInterruptHandler,
                                           Runnable sendInterruptHandler) {
        this.receiveInterruptHandler = receiveInterruptHandler;
        this.sendInterruptHandler = sendInterruptHandler;
    }

    private void scheduleReceiveInterrupt() {
        privilege.interrupt.schedule(Stats.ConsoleTime, "console read",
                                     receiveInterrupt);
    }

    /**
     * Attempt to read a byte from the object backing this console.
     *
     * @return  the byte read, or -1 of no data is available.
     */
    protected int in() {
        try {
            if (System.in.available() <= 0)
                return -1;

            return System.in.read();
        }
        catch (IOException e) {
            return -1;
        }
    }

    private int translateCharacter(int c) {
        // translate win32 0x0D 0x0A sequence to single newline
        if (c == 0x0A && prevCarriageReturn) {
            prevCarriageReturn = false;
            return -1;
        }
        prevCarriageReturn = (c == 0x0D);

        // invalid if non-ASCII
        if (c >= 0x80)
            return -1;
        // backspace characters
        else if (c == 0x04 || c == 0x08 || c == 0x19 || c == 0x1B || c == 0x7F)
            return '\b';
        // if normal ASCII range, nothing to do
        else if (c >= 0x20)
            return c;
        // newline characters
        else if (c == 0x0A || c == 0x0D)
            return '\n';
        // everything else is invalid
        else
            return -1;
    }


    private void receiveInterrupt() {
        Lib.assertTrue(incomingKey == -1);

        incomingKey = translateCharacter(in());
        if (incomingKey == -1) {
            scheduleReceiveInterrupt();
        }
        else {
            privilege.stats.numConsoleReads++;

            if (receiveInterruptHandler != null)
                receiveInterruptHandler.run();
        }
    }

    public final int readByte() {
        int key = incomingKey;

        if (incomingKey != -1) {
            incomingKey = -1;
            scheduleReceiveInterrupt();
        }

        return key;
    }

    private void scheduleSendInterrupt() {
        privilege.interrupt.schedule(Stats.ConsoleTime, "console write",
                                     sendInterrupt);
    }

    /**
     * Write a byte to the object backing this console.
     *
     * @param   value   the byte to write.
     */
    protected void out(int value) {
        System.out.write(value);
        System.out.flush();
    }

    private void sendInterrupt() {
```

```
        Lib.assertTrue(outgoingKey != -1);

        out(outgoingKey);
        outgoingKey = -1;

        privilege.stats.numConsoleWrites++;

        if (sendInterruptHandler != null)
            sendInterruptHandler.run();
    }

    public final void writeByte(int value) {
        if (outgoingKey == -1)
            scheduleSendInterrupt();

        outgoingKey = value&0xFF;
    }

    private Privilege privilege = null;

    private Runnable receiveInterrupt;
    private Runnable sendInterrupt;

    private Runnable receiveInterruptHandler = null;
    private Runnable sendInterruptHandler = null;

    private int incomingKey = -1;
    private int outgoingKey = -1;

    private boolean prevCarriageReturn = false;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.machine.*;

/**
 * An object that maintains Nachos runtime statistics.
 */
public final class Stats {
    /**
     * Allocate a new statistics object.
     */
    public Stats() {
    }

    /**
     * Print out the statistics in this object.
     */
    public void print() {
        System.out.println("Ticks: total " + totalTicks
                           + ", kernel " + kernelTicks
                           + ", user " + userTicks);
        System.out.println("Disk I/O: reads " + numDiskReads
                           + ", writes " + numDiskWrites);
        System.out.println("Console I/O: reads " + numConsoleReads
                           + ", writes " + numConsoleWrites);
        System.out.println("Paging: page faults " + numPageFaults
                           + ", TLB misses " + numTLBMisses);
        System.out.println("Network I/O: received " + numPacketsReceived
                           + ", sent " + numPacketsSent);
    }

    /**
     * The total amount of simulated time that has passed since Nachos
     * started.
     */
    public long totalTicks = 0;
    /**
     * The total amount of simulated time that Nachos has spent in kernel mode.
     */
    public long kernelTicks = 0;
    /**
     * The total amount of simulated time that Nachos has spent in user mode.
     */
    public long userTicks = 0;

    /** The total number of sectors Nachos has read from the simulated disk.*/
    public int numDiskReads = 0;
    /** The total number of sectors Nachos has written to the simulated disk.*/
    public int numDiskWrites = 0;
    /** The total number of characters Nachos has read from the console. */
    public int numConsoleReads = 0;
    /** The total number of characters Nachos has written to the console. */
    public int numConsoleWrites = 0;
    /** The total number of page faults that have occurred. */
    public int numPageFaults = 0;
    /** The total number of TLB misses that have occurred. */
    public int numTLBMisses = 0;
    /** The total number of packets Nachos has sent to the network. */
    public int numPacketsSent = 0;
    /** The total number of packets Nachos has received from the network. */
    public int numPacketsReceived = 0;

    /**
     * The amount to advance simulated time after each user instructions is
     * executed.
     */
    public static final int UserTick = 1;
    /**
     * The amount to advance simulated time after each interrupt enable.
     */
    public static final int KernelTick = 10;
    /**
     * The amount of simulated time required to rotate the disk 360 degrees.
     */
    public static final int RotationTime = 500;
    /**
     * The amount of simulated time required for the disk to seek.
     */
    public static final int SeekTime = 500;
    /**
     * The amount of simulated time required for the console to handle a
     * character.
     */
    public static final int ConsoleTime = 100;
    /**
     * The amount of simulated time required for the network to handle a
     * packet.
     */
    public static final int NetworkTime = 100;
    /**
     * The mean amount of simulated time between timer interrupts.
     */
    public static final int TimerTicks = 500;
    /**
     * The amount of simulated time required for an elevator to move a floor.
     */
    public static final int ElevatorTicks = 2000;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;
import nachos.threads.*;

import java.io.File;
import java.io.RandomAccessFile;
import java.io.IOException;

/**
 * This class implements a file system that redirects all requests to the host
 * operating system's file system.
 */
public class StubFileSystem implements FileSystem {
    /**
     * Allocate a new stub file system.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     * @param   directory       the root directory of the stub file system.
     */
    public StubFileSystem(Privilege privilege, File directory) {
        this.privilege = privilege;
        this.directory = directory;
    }

    public OpenFile open(String name, boolean truncate) {
        if (!checkName(name))
            return null;

        delay();

        try {
            return new StubOpenFile(name, truncate);
        }
        catch (IOException e) {
            return null;
        }
    }

    public boolean remove(String name) {
        if (!checkName(name))
            return false;

        delay();

        FileRemover fr = new FileRemover(new File(directory, name));
        privilege.doPrivileged(fr);
        return fr.successful;
    }

    private class FileRemover implements Runnable {
        public FileRemover(File f) {
            this.f = f;
        }

        public void run() {
            successful = f.delete();
        }

        public boolean successful = false;
        private File f;
    }

    private void delay() {
        long time = Machine.timer().getTime();
        int amount = 1000;
        ThreadedKernel.alarm.waitUntil(amount);
        Lib.assertTrue(Machine.timer().getTime() >= time+amount);
    }

    private class StubOpenFile extends OpenFileWithPosition {
        StubOpenFile(final String name, final boolean truncate)
            throws IOException {
            super(StubFileSystem.this, name);

            final File f = new File(directory, name);

            if (openCount == maxOpenFiles)
                throw new IOException();

            privilege.doPrivileged(new Runnable() {
                public void run() { getRandomAccessFile(f, truncate); }
            });

            if (file == null)
                throw new IOException();

            open = true;
            openCount++;
        }

        private void getRandomAccessFile(File f, boolean truncate) {
            try {
                if (!truncate && !f.exists())
                    return;

                file = new RandomAccessFile(f, "rw");

                if (truncate)
                    file.setLength(0);
            }
            catch (IOException e) {
            }
        }

        public int read(int pos, byte[] buf, int offset, int length) {
            if (!open)
                return -1;

            try {
                delay();

                file.seek(pos);
                return Math.max(0, file.read(buf, offset, length));
            }
            catch (IOException e) {
                return -1;
            }
        }

        public int write(int pos, byte[] buf, int offset, int length) {
            if (!open)
                return -1;

            try {
                delay();
```

```
            file.seek(pos);
            file.write(buf, offset, length);
            return length;
        }
        catch (IOException e) {
            return -1;
        }
    }

    public int length() {
        try {
            return (int) file.length();
        }
        catch (IOException e) {
            return -1;
        }
    }

    public void close() {
        if (open) {
            open = false;
            openCount--;
        }

        try {
            file.close();
        }
        catch (IOException e) {
        }
    }

    private RandomAccessFile file = null;
    private boolean open = false;
}

private int openCount = 0;
private static final int maxOpenFiles = 16;

private Privilege privilege;
private File directory;

private static boolean checkName(String name) {
    char[] chars = name.toCharArray();

    for (int i=0; i<chars.length; i++) {
        if (chars[i] < 0 || chars[i] >= allowedFileNameCharacters.length)
            return false;
        if (!allowedFileNameCharacters[(int) chars[i]])
            return false;
    }
    return true;
}

private static boolean[] allowedFileNameCharacters = new boolean[0x80];

private static void reject(char c) {
    allowedFileNameCharacters[c] = false;
}

private static void allow(char c) {
    allowedFileNameCharacters[c] = true;
}

private static void reject(char first, char last) {
    for (char c=first; c<=last; c++)
        allowedFileNameCharacters[c] = false;
    }

    private static void allow(char first, char last) {
        for (char c=first; c<=last; c++)
            allowedFileNameCharacters[c] = true;
    }

    static {
        reject((char) 0x00, (char) 0x7F);

        allow('A', 'Z');
        allow('a', 'z');
        allow('0', '9');

        allow('-');
        allow('_');
        allow('.');
        allow(',');
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;
import nachos.threads.KThread;

import java.util.Vector;
import java.security.PrivilegedAction;

/**
 * A TCB simulates the low-level details necessary to create, context-switch,
 * and destroy Nachos threads. Each TCB controls an underlying JVM Thread
 * object.
 *
 * <p>
 * Do not use any methods in <tt>java.lang.Thread</tt>, as they are not
 * compatible with the TCB API. Most <tt>Thread</tt> methods will either crash
 * Nachos or have no useful effect.
 *
 * <p>
 * Do not use the <i>synchronized</i> keyword <b>anywhere</b> in your code.
 * It's against the rules, <i>and</i> it can easily deadlock nachos.
 */
public final class TCB {
    /**
     * Allocate a new TCB.
     */
    public TCB() {
    }

    /**
     * Give the TCB class the necessary privilege to create threads. This is
     * necessary, because unlike other machine classes that need privilege, we
     * want the kernel to be able to create TCB objects on its own.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public static void givePrivilege(Privilege privilege) {
        TCB.privilege = privilege;
        privilege.tcb = new TCBPrivilege();
    }

    /**
     * Causes the thread represented by this TCB to begin execution. The
     * specified target is run in the thread.
     */
    public void start(Runnable target) {
        /* We will not use synchronization here, because we're assuming that
         * either this is the first call to start(), or we're being called in
         * the context of another TCB. Since we only allow one TCB to run at a
         * time, no synchronization is necessary.
         *
         * The only way this assumption could be broken is if one of our
         * non-Nachos threads used the TCB code.
         */

        /* Make sure this TCB has not already been started. If done is false,
         * then destroy() has not yet set javaThread back to null, so we can
         * use javaThread as a reliable indicator of whether or not start() has
         * already been invoked.
         */
        Lib.assertTrue(javaThread == null && !done);

        /* Make sure there aren't too many running TCBs already. This
         * limitation exists in an effort to prevent wild thread usage.
         */
        Lib.assertTrue(runningThreads.size() < maxThreads);

        isFirstTCB = (currentTCB == null);

        /* Probably unnecessary sanity check: if this is not the first TCB, we
         * make sure that the current thread is bound to the current TCB. This
         * check can only fail if non-Nachos threads invoke start().
         */
        if (!isFirstTCB)
            Lib.assertTrue(currentTCB.javaThread == Thread.currentThread());

        /* At this point all checks are complete, so we go ahead and start the
         * TCB. Whether or not this is the first TCB, it gets added to
         * runningThreads, and we save the target closure.
         */
        runningThreads.add(this);

        this.target = target;

        if (!isFirstTCB) {
            /* If this is not the first TCB, we have to make a new Java thread
             * to run it. Creating Java threads is a privileged operation.
             */
            tcbTarget = new Runnable() {
                    public void run() { threadroot(); }
                };

            privilege.doPrivileged(new Runnable() {
                    public void run() { javaThread = new Thread(tcbTarget); }
                });

            /* The Java thread hasn't yet started, but we need to get it
             * blocking in yield(). We do this by temporarily turning off the
             * current TCB, starting the new Java thread, and waiting for it
             * to wake us up from threadroot(). Once the new TCB wakes us up,
             * it's safe to context switch to the new TCB.
             */
            currentTCB.running = false;

            this.javaThread.start();
            currentTCB.waitForInterrupt();
        }
        else {
            /* This is the first TCB, so we don't need to make a new Java
             * thread to run it; we just steal the current Java thread.
             */
            javaThread = Thread.currentThread();

            /* All we have to do now is invoke threadroot() directly. */
            threadroot();
        }
    }

    /**
     * Return the TCB of the currently running thread.
     */
    public static TCB currentTCB() {
        return currentTCB;
    }

    /**
     * Context switch between the current TCB and this TCB. This TCB will
```

```
     * become the new current TCB. It is acceptable for this TCB to be the
     * current TCB.
     */
    public void contextSwitch() {
        /* Probably unnecessary sanity check: we make sure that the current
         * thread is bound to the current TCB. This check can only fail if
         * non-Nachos threads invoke start().
         */
        Lib.assertTrue(currentTCB.javaThread == Thread.currentThread());

        // make sure AutoGrader.runningThread() called associateThread()
        Lib.assertTrue(currentTCB.associated);
        currentTCB.associated = false;

        // can't switch from a TCB to itself
        if (this == currentTCB)
            return;

        /* There are some synchronization concerns here. As soon as we wake up
         * the next thread, we cannot assume anything about static variables,
         * or about any TCB's state. Therefore, before waking up the next
         * thread, we must latch the value of currentTCB, and set its running
         * flag to false (so that, in case we get interrupted before we call
         * yield(), the interrupt will set the running flag and yield() won't
         * block).
         */

        TCB previous = currentTCB;
        previous.running = false;

        this.interrupt();
        previous.yield();
    }

    /**
     * Destroy this TCB. This TCB must not be in use by the current thread.
     * This TCB must also have been authorized to be destroyed by the
     * autograder.
     */
    public void destroy() {
        // make sure the current TCB is correct
        Lib.assertTrue(currentTCB != null &&
                       currentTCB.javaThread == Thread.currentThread());
        // can't destroy current thread
        Lib.assertTrue(this != currentTCB);
        // thread must have started but not be destroyed yet
        Lib.assertTrue(javaThread != null && !done);

        // ensure AutoGrader.finishingCurrentThread() called authorizeDestroy()
        Lib.assertTrue(nachosThread == toBeDestroyed);
        toBeDestroyed = null;

        this.done = true;
        currentTCB.running = false;

        this.interrupt();
        currentTCB.waitForInterrupt();

        this.javaThread = null;
    }

    /**
     * Destroy all TCBs and exit Nachos. Same as <tt>Machine.terminate()</tt>.
     */
    public static void die() {
        privilege.exit(0);
    }

    /**
     * Test if the current JVM thread belongs to a Nachos TCB. The AWT event
     * dispatcher is an example of a non-Nachos thread.
     *
     * @return  <tt>true</tt> if the current JVM thread is a Nachos thread.
     */
    public static boolean isNachosThread() {
        return (currentTCB != null &&
                Thread.currentThread() == currentTCB.javaThread);
    }

    private void threadroot() {
        // this should be running the current thread
        Lib.assertTrue(javaThread == Thread.currentThread());

        if (!isFirstTCB) {
            /* start() is waiting for us to wake it up, signalling that it's OK
             * to context switch to us. We leave the running flag false so that
             * we'll still run if a context switch happens before we go to
             * sleep. All we have to do is wake up the current TCB and then
             * wait to get woken up by contextSwitch() or destroy().
             */

            currentTCB.interrupt();
            this.yield();
        }
        else {
            /* start() called us directly, so we just need to initialize
             * a couple things.
             */

            currentTCB = this;
            running = true;
        }

        try {
            target.run();

            // no way out of here without going throw one of the catch blocks
            Lib.assertNotReached();
        }
        catch (ThreadDeath e) {
            // make sure this TCB is being destroyed properly
            if (!done) {
                System.out.print("\nTCB terminated improperly!\n");
                privilege.exit(1);
            }

            runningThreads.removeElement(this);
            if (runningThreads.isEmpty())
                privilege.exit(0);
        }
        catch (Throwable e) {
            System.out.print("\n");
            e.printStackTrace();

            runningThreads.removeElement(this);
            if (runningThreads.isEmpty())
                privilege.exit(1);
            else
                die();
        }
```

```
}

/**
 * Invoked by threadroot() and by contextSwitch() when it is necessary to
 * wait for another TCB to context switch to this TCB. Since this TCB
 * might get destroyed instead, we check the <tt>done</tt> flag after
 * waking up. If it is set, the TCB that woke us up is waiting for an
 * acknowledgement in destroy(). Otherwise, we just set the current TCB to
 * this TCB and return.
 */
private void yield() {
    waitForInterrupt();

    if (done) {
        currentTCB.interrupt();
        throw new ThreadDeath();
    }

    currentTCB = this;
}

/**
 * Waits on the monitor bound to this TCB until its <tt>running</tt> flag
 * is set to <tt>true</tt>. <tt>waitForInterrupt()</tt> is used whenever a
 * TCB needs to go to wait for its turn to run. This includes the ping-pong
 * process of starting and destroying TCBs, as well as in context switching
 * from this TCB to another. We don't rely on <tt>currentTCB</tt>, since it
 * is updated by <tt>contextSwitch()</tt> before we get called.
 */
private synchronized void waitForInterrupt() {
    while (!running) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
}

/**
 * Wake up this TCB by setting its <tt>running</tt> flag to <tt>true</tt>
 * and signalling the monitor bound to it. Used in the ping-pong process of
 * starting and destroying TCBs, as well as in context switching to this
 * TCB.
 */
private synchronized void interrupt() {
    running = true;
    notify();
}

private void associateThread(KThread thread) {
    // make sure AutoGrader.runningThread() gets called only once per
    // context switch
    Lib.assertTrue(!associated);
    associated = true;

    Lib.assertTrue(thread != null);

    if (nachosThread != null)
        Lib.assertTrue(thread == nachosThread);
    else
        nachosThread = thread;
}

private static void authorizeDestroy(KThread thread) {
    // make sure AutoGrader.finishingThread() gets called only once per
    // destroy
    Lib.assertTrue(toBeDestroyed == null);
    toBeDestroyed = thread;
}

/**
 * The maximum number of started, non-destroyed TCB's that can be in
 * existence.
 */
public static final int maxThreads = 250;

/**
 * A reference to the currently running TCB. It is initialized to
 * <tt>null</tt> when the <tt>TCB</tt> class is loaded, and then the first
 * invocation of <tt>start(Runnable)</tt> assigns <tt>currentTCB</tt> a
 * reference to the first TCB. After that, only <tt>yield()</tt> can
 * change <tt>currentTCB</tt> to the current TCB, and only after
 * <tt>waitForInterrupt()</tt> returns.
 *
 * <p>
 * Note that <tt>currentTCB.javaThread</tt> will not be the current thread
 * if the current thread is not bound to a TCB (this includes the threads
 * created for the hardware simulation).
 */
private static TCB currentTCB = null;

/**
 * A vector containing all <i>running</i> TCB objects. It is initialized to
 * an empty vector when the <tt>TCB</tt> class is loaded. TCB objects are
 * added only in <tt>start(Runnable)</tt>, which can only be invoked once
 * on each TCB object. TCB objects are removed only in each of the
 * <tt>catch</tt> clauses of <tt>threadroot()</tt>, one of which is always
 * invoked on thread termination. The maximum number of threads in
 * <tt>runningThreads</tt> is limited to <tt>maxThreads</tt> by
 * <tt>start(Runnable)</tt>. If <tt>threadroot()</tt> drops the number of
 * TCB objects in <tt>runningThreads</tt> to zero, Nachos exits, so once
 * the first TCB is created, this vector is basically never empty.
 */
private static Vector<TCB> runningThreads = new Vector<TCB>();

private static Privilege privilege;
private static KThread toBeDestroyed = null;

/**
 * <tt>true</tt> if and only if this TCB is the first TCB to start, the one
 * started in <tt>Machine.main(String[])</tt>. Initialized by
 * <tt>start(Runnable)</tt>, on the basis of whether <tt>currentTCB</tt>
 * has been initialized.
 */
private boolean isFirstTCB;

/**
 * A reference to the Java thread bound to this TCB. It is initially
 * <tt>null</tt>, assigned to a Java thread in <tt>start(Runnable)</tt>,
 * and set to <tt>null</tt> again in <tt>destroy()</tt>.
 */
private Thread javaThread = null;

/**
 * <tt>true</tt> if and only if the Java thread bound to this TCB ought to
 * be running. This is an entirely different condition from membership in
 * <tt>runningThreads</tt>, which contains all TCB objects that have
 * started and have not terminated. <tt>running</tt> is only <tt>true</tt>
 * when the associated Java thread ought to run ASAP. When starting or
 * destroying a TCB, this is temporarily true for a thread other than that
 * of the current TCB.
 */
```

```
    private boolean running = false;

    /**
     * Set to <tt>true</tt> by <tt>destroy()</tt>, so that when
     * <tt>waitForInterrupt()</tt> returns in the doomed TCB, <tt>yield()</tt>
     * will know that the current TCB is doomed.
     */
    private boolean done = false;

    private KThread nachosThread = null;
    private boolean associated = false;
    private Runnable target;
    private Runnable tcbTarget;

    private static class TCBPrivilege implements Privilege.TCBPrivilege {
        public void associateThread(KThread thread) {
            Lib.assertTrue(currentTCB != null);
            currentTCB.associateThread(thread);
        }
        public void authorizeDestroy(KThread thread) {
            TCB.authorizeDestroy(thread);
        }
    }
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.security.*;

/**
 * A hardware timer generates a CPU timer interrupt approximately every 500
 * clock ticks. This means that it can be used for implementing time-slicing,
 * or for having a thread go to sleep for a specific period of time.
 *
 * The <tt>Timer</tt> class emulates a hardware timer by scheduling a timer
 * interrupt to occur every time approximately 500 clock ticks pass. There is
 * a small degree of randomness here, so interrupts do not occur exactly every
 * 500 ticks.
 */
public final class Timer {
    /**
     * Allocate a new timer.
     *
     * @param   privilege       encapsulates privileged access to the Nachos
     *                          machine.
     */
    public Timer(Privilege privilege) {
        System.out.print(" timer");

        this.privilege = privilege;

        timerInterrupt = new Runnable() {
                public void run() { timerInterrupt(); }
            };

        autoGraderInterrupt = new Runnable() {
                public void run() {
                    Machine.autoGrader().timerInterrupt(Timer.this.privilege,
                                                        lastTimerInterrupt);
                }
            };

        scheduleInterrupt();
    }

    /**
     * Set the callback to use as a timer interrupt handler. The timer
     * interrupt handler will be called approximately every 500 clock ticks.
     *
     * @param   handler         the timer interrupt handler.
     */
    public void setInterruptHandler(Runnable handler) {
        this.handler = handler;
    }

    /**
     * Get the current time.
     *
     * @return  the number of clock ticks since Nachos started.
     */
    public long getTime() {
        return privilege.stats.totalTicks;
    }

    private void timerInterrupt() {
        scheduleInterrupt();
        scheduleAutoGraderInterrupt();

        lastTimerInterrupt = getTime();

        if (handler != null)
            handler.run();
    }

    private void scheduleInterrupt() {
        int delay = Stats.TimerTicks;
        delay += Lib.random(delay/10) - (delay/20);

        privilege.interrupt.schedule(delay, "timer", timerInterrupt);
    }

    private void scheduleAutoGraderInterrupt() {
        privilege.interrupt.schedule(1, "timerAG", autoGraderInterrupt);
    }

    private long lastTimerInterrupt;
    private Runnable timerInterrupt;
    private Runnable autoGraderInterrupt;

    private Privilege privilege;
    private Runnable handler = null;
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.machine;

import nachos.machine.*;

/**
 * A single translation between a virtual page and a physical page.
 */
public final class TranslationEntry {
    /**
     * Allocate a new invalid translation entry.
     */
    public TranslationEntry() {
        valid = false;
    }

    /**
     * Allocate a new translation entry with the specified initial state.
     *
     * @param   vpn             the virtual page numben.
     * @param   ppn             the physical page number.
     * @param   valid           the valid bit.
     * @param   readOnly        the read-only bit.
     * @param   used            the used bit.
     * @param   dirty           the dirty bit.
     */
    public TranslationEntry(int vpn, int ppn, boolean valid, boolean readOnly,
                            boolean used, boolean dirty) {
        this.vpn = vpn;
        this.ppn = ppn;
        this.valid = valid;
        this.readOnly = readOnly;
        this.used = used;
        this.dirty = dirty;
    }

    /**
     * Allocate a new translation entry, copying the contents of an existing
     * one.
     *
     * @param   entry   the translation entry to copy.
     */
    public TranslationEntry(TranslationEntry entry) {
        vpn = entry.vpn;
        ppn = entry.ppn;
        valid = entry.valid;
        readOnly = entry.readOnly;
        used = entry.used;
        dirty = entry.dirty;
    }

    /** The virtual page number. */
    public int vpn;

    /** The physical page number. */
    public int ppn;

    /**
     * If this flag is <tt>false</tt>, this translation entry is ignored.
     */
    public boolean valid;

    /**
     * If this flag is <tt>true</tt>, the user pprogram is not allowed to
     * modify the contents of this virtual page.
     */
    public boolean readOnly;

    /**
     * This flag is set to <tt>true</tt> every time the page is read or written
     * by a user program.
     */
    public boolean used;

    /**
     * This flag is set to <tt>true</tt> every time the page is written by a
     * user program.
     */
    public boolean dirty;
}
```

```
package nachos.network;

import nachos.machine.*;

/**
 * A mail message. Includes a packet header, a mail header, and the actual
 * payload.
 *
 * @see nachos.machine.Packet
 */
public class MailMessage {
    /**
     * Allocate a new mail message to be sent, using the specified parameters.
     *
     * @param   dstLink         the destination link address.
     * @param   dstPort         the destination port.
     * @param   srcLink         the source link address.
     * @param   srcPort         the source port.
     * @param   contents        the contents of the packet.
     */
    public MailMessage(int dstLink, int dstPort, int srcLink, int srcPort,
                       byte[] contents) throws MalformedPacketException {
        // make sure the paramters are valid
        if (dstPort < 0 || dstPort >= portLimit ||
            srcPort < 0 || srcPort >= portLimit ||
            contents.length > maxContentsLength)
            throw new MalformedPacketException();

        this.dstPort = (byte) dstPort;
        this.srcPort = (byte) srcPort;
        this.contents = contents;

        byte[] packetContents = new byte[headerLength + contents.length];

        packetContents[0] = (byte) dstPort;
        packetContents[1] = (byte) srcPort;

        System.arraycopy(contents, 0, packetContents, headerLength,
                         contents.length);

        packet = new Packet(dstLink, srcLink, packetContents);
    }

    /**
     * Allocate a new mail message using the specified packet from the network.
     *
     * @param   packet  the packet containg the mail message.
     */
    public MailMessage(Packet packet) throws MalformedPacketException {
        this.packet = packet;

        // make sure we have a valid header
        if (packet.contents.length < headerLength ||
            packet.contents[0] < 0 || packet.contents[0] >= portLimit ||
            packet.contents[1] < 0 || packet.contents[1] >= portLimit)
            throw new MalformedPacketException();

        dstPort = packet.contents[0];
        srcPort = packet.contents[1];

        contents = new byte[packet.contents.length - headerLength];
        System.arraycopy(packet.contents, headerLength, contents, 0,
                         contents.length);
    }

    /**
     * Return a string representation of the message headers.
     */
    public String toString() {
        return "from (" + packet.srcLink + ":" + srcPort +
            ") to (" + packet.dstLink + ":" + dstPort +
            "), " + contents.length + " bytes";
    }

    /** This message, as a packet that can be sent through a network link. */
    public Packet packet;
    /** The port used by this message on the destination machine. */
    public int dstPort;
    /** The port used by this message on the source machine. */
    public int srcPort;
    /** The contents of this message, excluding the mail message header. */
    public byte[] contents;

    /**
     * The number of bytes in a mail header. The header is formatted as
     * follows:
     *
     * <table>
     * <tr><td>offset</td><td>size</td><td>value</td></tr>
     * <tr><td>0</td><td>1</td><td>destination port</td></tr>
     * <tr><td>1</td><td>1</td><td>source port</td></tr>
     * </table>
     */
    public static final int headerLength = 2;

    /** Maximum payload (real data) that can be included in a single mesage. */
    public static final int maxContentsLength =
        Packet.maxContentsLength - headerLength;

    /**
     * The upper limit on mail ports. All ports fall between <tt>0</tt> and
     * <tt>portLimit - 1</tt>.
     */
    public static final int portLimit = 128;
}
```

```
package nachos.network;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;
import nachos.vm.*;
import nachos.network.*;

/**
 * A kernel with network support.
 */
public class NetKernel extends VMKernel {
    /**
     * Allocate a new networking kernel.
     */
    public NetKernel() {
        super();
    }

    /**
     * Initialize this kernel.
     */
    public void initialize(String[] args) {
        super.initialize(args);

        postOffice = new PostOffice();
    }

    /**
     * Test the network. Create a server thread that listens for pings on port
     * 1 and sends replies. Then ping one or two hosts. Note that this test
     * assumes that the network is reliable (i.e. that the network's
     * reliability is 1.0).
     */
    public void selfTest() {
        super.selfTest();

        KThread serverThread = new KThread(new Runnable() {
                public void run() { pingServer(); }
            });

        serverThread.fork();

        System.out.println("Press any key to start the network test...");
        console.readByte(true);

        int local = Machine.networkLink().getLinkAddress();

        // ping this machine first
        ping(local);

        // if we're 0 or 1, ping the opposite
        if (local <= 1)
            ping(1-local);
    }

    private void ping(int dstLink) {
        int srcLink = Machine.networkLink().getLinkAddress();

        System.out.println("PING " + dstLink + " from " + srcLink);

        long startTime = Machine.timer().getTime();

        MailMessage ping;

        try {
            ping = new MailMessage(dstLink, 1,
                                   Machine.networkLink().getLinkAddress(), 0,
                                   new byte[0]);
        }
        catch (MalformedPacketException e) {
            Lib.assertNotReached();
            return;
        }

        postOffice.send(ping);

        MailMessage ack = postOffice.receive(0);

        long endTime = Machine.timer().getTime();

        System.out.println("time=" + (endTime-startTime) + " ticks");
    }

    private void pingServer() {
        while (true) {
            MailMessage ping = postOffice.receive(1);

            MailMessage ack;

            try {
                ack = new MailMessage(ping.packet.srcLink, ping.srcPort,
                                      ping.packet.dstLink, ping.dstPort,
                                      ping.contents);
            }
            catch (MalformedPacketException e) {
                // should never happen...
                continue;
            }

            postOffice.send(ack);
        }
    }

    /**
     * Start running user programs.
     */
    public void run() {
        super.run();
    }

    /**
     * Terminate this kernel. Never returns.
     */
    public void terminate() {
        super.terminate();
    }

    private PostOffice postOffice;

    // dummy variables to make javac smarter
    private static NetProcess dummy1 = null;
}
```

```
package nachos.network;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;
import nachos.vm.*;

/**
 * A <tt>VMProcess</tt> that supports networking syscalls.
 */
public class NetProcess extends VMProcess {
    /**
     * Allocate a new process.
     */
    public NetProcess() {
        super();
    }

    private static final int
        syscallConnect = 11,
        syscallAccept = 12;

    /**
     * Handle a syscall exception. Called by <tt>handleException()</tt>. The
     * <i>syscall</i> argument identifies which syscall the user executed:
     *
     * <table>
     * <tr><td>syscall#</td><td>syscall prototype</td></tr>
     * <tr><td>11</td><td><tt>int  connect(int host, int port);</tt></td></tr>
     * <tr><td>12</td><td><tt>int  accept(int port);</tt></td></tr>
     * </table>
     *
     * @param   syscall the syscall number.
     * @param   a0      the first syscall argument.
     * @param   a1      the second syscall argument.
     * @param   a2      the third syscall argument.
     * @param   a3      the fourth syscall argument.
     * @return  the value to be returned to the user.
     */
    public int handleSyscall(int syscall, int a0, int a1, int a2, int a3) {
        switch (syscall) {
        default:
            return super.handleSyscall(syscall, a0, a1, a2, a3);
        }
    }
}
```

```
package nachos.network;

import nachos.machine.*;
import nachos.threads.*;

/**
 * A collection of message queues, one for each local port. A
 * <tt>PostOffice</tt> interacts directly with the network hardware. Because
 * of the network hardware, we are guaranteed that messages will never be
 * corrupted, but they might get lost.
 *
 * <p>
 * The post office uses a "postal worker" thread to wait for messages to arrive
 * from the network and to place them in the appropriate queues. This cannot
 * be done in the receive interrupt handler because each queue (implemented
 * with a <tt>SynchList</tt>) is protected by a lock.
 */
public class PostOffice {
    /**
     * Allocate a new post office, using an array of <tt>SynchList</tt>s.
     * Register the interrupt handlers with the network hardware and start the
     * "postal worker" thread.
     */
    public PostOffice() {
        messageReceived = new Semaphore(0);
        messageSent = new Semaphore(0);
        sendLock = new Lock();

        queues = new SynchList[MailMessage.portLimit];
        for (int i=0; i<queues.length; i++)
            queues[i] = new SynchList();

        Runnable receiveHandler = new Runnable() {
            public void run() { receiveInterrupt(); }
        };
        Runnable sendHandler = new Runnable() {
            public void run() { sendInterrupt(); }
        };
        Machine.networkLink().setInterruptHandlers(receiveHandler,
                                                   sendHandler);

        KThread t = new KThread(new Runnable() {
                public void run() { postalDelivery(); }
            });

        t.fork();
    }

    /**
     * Retrieve a message on the specified port, waiting if necessary.
     *
     * @param   port    the port on which to wait for a message.
     *
     * @return  the message received.
     */
    public MailMessage receive(int port) {
        Lib.assertTrue(port >= 0 && port < queues.length);

        Lib.debug(dbgNet, "waiting for mail on port " + port);

        MailMessage mail = (MailMessage) queues[port].removeFirst();

        if (Lib.test(dbgNet))
            System.out.println("got mail on port " + port + ": " + mail);

        return mail;
    }

    /**
     * Wait for incoming messages, and then put them in the correct mailbox.
     */
    private void postalDelivery() {
        while (true) {
            messageReceived.P();

            Packet p = Machine.networkLink().receive();

            MailMessage mail;

            try {
                mail = new MailMessage(p);
            }
            catch (MalformedPacketException e) {
                continue;
            }

            if (Lib.test(dbgNet))
                System.out.println("delivering mail to port " + mail.dstPort
                                   + ": " + mail);

            // atomically add message to the mailbox and wake a waiting thread
            queues[mail.dstPort].add(mail);
        }
    }

    /**
     * Called when a packet has arrived and can be dequeued from the network
     * link.
     */
    private void receiveInterrupt() {
        messageReceived.V();
    }

    /**
     * Send a message to a mailbox on a remote machine.
     */
    public void send(MailMessage mail) {
        if (Lib.test(dbgNet))
            System.out.println("sending mail: " + mail);

        sendLock.acquire();

        Machine.networkLink().send(mail.packet);
        messageSent.P();

        sendLock.release();
    }

    /**
     * Called when a packet has been sent and another can be queued to the
     * network link. Note that this is called even if the previous packet was
     * dropped.
     */
    private void sendInterrupt() {
        messageSent.V();
    }

    private SynchList[] queues;
    private Semaphore messageReceived;  // V'd when a message can be dequeued
    private Semaphore messageSent;      // V'd when a message can be queued
```

```
    private Lock sendLock;

    private static final char dbgNet = 'n';
}
```

```
DIRS = threads machine security ag

include ../Makefile
```

```
Machine.stubFileSystem = false
Machine.processor = false
Machine.console = false
Machine.disk = false
Machine.bank = false
Machine.networkLink = false
ElevatorBank.allowElevatorGUI = true
NachosSecurityManager.fullySecure = false
ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler #nachos.threads.Priority
Scheduler
Kernel.kernel = nachos.threads.ThreadedKernel
```

```
DIRS = userprog threads machine security ag

include ../Makefile
```

```
Machine.stubFileSystem = true
Machine.processor = true
Machine.console = true
Machine.disk = false
Machine.bank = false
Machine.networkLink = false
Processor.usingTLB = false
Processor.numPhysPages = 64
ElevatorBank.allowElevatorGUI = false
NachosSecurityManager.fullySecure = false
ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler #nachos.threads.LotteryS
cheduler
Kernel.shellProgram = halt.coff #sh.coff
Kernel.processClassName = nachos.userprog.UserProcess
Kernel.kernel = nachos.userprog.UserKernel
```

```
DIRS = vm userprog threads machine security ag

include ../Makefile
```

```
Machine.stubFileSystem = true
Machine.processor = true
Machine.console = true
Machine.disk = false
Machine.bank = false
Machine.networkLink = false
Processor.usingTLB = true
Processor.numPhysPages = 16
ElevatorBank.allowElevatorGUI = false
NachosSecurityManager.fullySecure = false
ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler
Kernel.shellProgram = sh.coff
Kernel.processClassName = nachos.vm.VMProcess
Kernel.kernel = nachos.vm.VMKernel
```

```
DIRS = network vm userprog threads machine security ag

include ../Makefile
```

```
Machine.stubFileSystem = true
Machine.processor = true
Machine.console = true
Machine.disk = false
Machine.bank = false
Machine.networkLink = true
Processor.usingTLB = true
Processor.variableTLB = true
Processor.numPhysPages = 16
ElevatorBank.allowElevatorGUI = false
NetworkLink.reliability = 1.0                   # use 0.9 when you're ready
NachosSecurityManager.fullySecure = false
ThreadedKernel.scheduler = nachos.threads.RoundRobinScheduler
Kernel.shellProgram = sh.coff
Kernel.processClassName = nachos.network.NetProcess
Kernel.kernel = nachos.network.NetKernel
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.security;

import nachos.machine.*;

import java.io.File;
import java.security.Permission;
import java.io.FilePermission;
import java.util.PropertyPermission;
import java.net.NetPermission;
import java.awt.AWTPermission;
import java.security.PrivilegedAction;
import java.security.PrivilegedExceptionAction;
import java.security.PrivilegedActionException;

/**
 * Protects the environment from malicious Nachos code.
 */
public class NachosSecurityManager extends SecurityManager {
    /**
     * Allocate a new Nachos security manager.
     *
     * @param   testDirectory   the directory usable by the stub file system.
     */
    public NachosSecurityManager(File testDirectory) {
        this.testDirectory = testDirectory;

        fullySecure = Config.getBoolean("NachosSecurityManager.fullySecure");
    }

    /**
     * Return a privilege object for this security manager. This security
     * manager must not be the active security manager.
     *
     * @return  a privilege object for this security manager.
     */
    public Privilege getPrivilege() {
        Lib.assertTrue(this != System.getSecurityManager());

        return new PrivilegeProvider();
    }

    /**
     * Install this security manager.
     */
    public void enable() {
        Lib.assertTrue(this != System.getSecurityManager());

        doPrivileged(new Runnable() {
            public void run() {
                System.setSecurityManager(NachosSecurityManager.this);
            }
        });
    }

    private class PrivilegeProvider extends Privilege {
        public void doPrivileged(Runnable action) {
            NachosSecurityManager.this.doPrivileged(action);
        }

        public Object doPrivileged(PrivilegedAction action) {
            return NachosSecurityManager.this.doPrivileged(action);
        }

        public Object doPrivileged(PrivilegedExceptionAction action)
            throws PrivilegedActionException {
            return NachosSecurityManager.this.doPrivileged(action);
        }

        public void exit(int exitStatus) {
            invokeExitNotificationHandlers();
            NachosSecurityManager.this.exit(exitStatus);
        }
    }

    private void enablePrivilege() {
        if (privilegeCount == 0) {
            Lib.assertTrue(privileged == null);
            privileged = Thread.currentThread();
            privilegeCount++;
        }
        else {
            Lib.assertTrue(privileged == Thread.currentThread());
            privilegeCount++;
        }
    }

    private void rethrow(Throwable e) {
        disablePrivilege();

        if (e instanceof RuntimeException)
            throw (RuntimeException) e;
        else if (e instanceof Error)
            throw (Error) e;
        else
            Lib.assertNotReached();
    }

    private void disablePrivilege() {
        Lib.assertTrue(privileged != null && privilegeCount > 0);
        privilegeCount--;
        if (privilegeCount == 0)
            privileged = null;
    }

    private void forcePrivilege() {
        privileged = Thread.currentThread();
        privilegeCount = 1;
    }

    private void exit(int exitStatus) {
        forcePrivilege();
        System.exit(exitStatus);
    }

    private boolean isPrivileged() {
        // the autograder does not allow non-Nachos threads to be created, so..
        if (!TCB.isNachosThread())
            return true;

        return (privileged == Thread.currentThread());
    }

    private void doPrivileged(final Runnable action) {
        doPrivileged(new PrivilegedAction() {
            public Object run() { action.run(); return null; }
        });
    }
```

```
    private Object doPrivileged(PrivilegedAction action) {
        Object result = null;
        enablePrivilege();
        try {
            result = action.run();
        }
        catch (Throwable e) {
            rethrow(e);
        }
        disablePrivilege();
        return result;
    }

    private Object doPrivileged(PrivilegedExceptionAction action)
        throws PrivilegedActionException {
        Object result = null;
        enablePrivilege();
        try {
            result = action.run();
        }
        catch (Exception e) {
            throw new PrivilegedActionException(e);
        }
        catch (Throwable e) {
            rethrow(e);
        }
        disablePrivilege();
        return result;
    }

    private void no() {
        throw new SecurityException();
    }

    private void no(Permission perm) {
        System.err.println("\n\nLacked permission: " + perm);
        throw new SecurityException();
    }

    /**
     * Check the specified permission. Some operations are permissible while
     * not grading. These operations are regulated here.
     *
     * @param   perm    the permission to check.
     */
    public void checkPermission(Permission perm) {
        String name = perm.getName();

        // some permissions are strictly forbidden
        if (perm instanceof RuntimePermission) {
            // no creating class loaders
            if (name.equals("createClassLoader"))
                no(perm);
        }

        // allow the AWT mess when not grading
        if (!fullySecure) {
            if (perm instanceof NetPermission) {
                // might be needed to load awt stuff
                if (name.equals("specifyStreamHandler"))
                    return;
            }

            if (perm instanceof RuntimePermission) {
                // might need to load libawt
                if (name.startsWith("loadLibrary.")) {
                    String lib = name.substring("loadLibrary.".length());
                    if (lib.equals("awt")) {
                        Lib.debug(dbgSecurity, "\tdynamically linking " + lib);
                        return;
                    }
                }
            }

            if (perm instanceof AWTPermission) {
                // permit AWT stuff
                if (name.equals("accessEventQueue"))
                    return;
            }
        }

        // some are always allowed
        if (perm instanceof PropertyPermission) {
            // allowed to read properties
            if (perm.getActions().equals("read"))
                return;
        }

        // some require some more checking
        if (perm instanceof FilePermission) {
            if (perm.getActions().equals("read")) {
                // the test directory can only be read with privilege
                if (isPrivileged())
                    return;

                enablePrivilege();

                // not allowed to read test directory directly w/out privilege
                try {
                    File f = new File(name);
                    if (f.isFile()) {
                        File p = f.getParentFile();
                        if (p != null) {
                            if (p.equals(testDirectory))
                                no(perm);
                        }
                    }
                }
                catch (Throwable e) {
                    rethrow(e);
                }

                disablePrivilege();
                return;
            }
            else if (perm.getActions().equals("write") ||
                     perm.getActions().equals("delete")) {
                // only allowed to write test diretory, and only with privilege
                verifyPrivilege();

                try {
                    File f = new File(name);
                    if (f.isFile()) {
                        File p = f.getParentFile();
                        if (p != null && p.equals(testDirectory))
                            return;
                    }
                }
                catch (Throwable e) {
                    no(perm);
```

```
                }
            }
            else if (perm.getActions().equals("execute")) {
                // only allowed to execute with privilege, and if there's a net
                verifyPrivilege();

                if (Machine.networkLink() == null)
                    no(perm);
            }
            else {
                no(perm);
            }
        }

        // default to requiring privilege
        verifyPrivilege(perm);
    }

    /**
     * Called by the <tt>java.lang.Thread</tt> constructor to determine a
     * thread group for a child thread of the current thread. The caller must
     * be privileged in order to successfully create the thread.
     *
     * @return  a thread group for the new thread, or <tt>null</tt> to use the
     *          current thread's thread group.
     */
    public ThreadGroup getThreadGroup() {
        verifyPrivilege();
        return null;
    }

    /**
     * Verify that the caller is privileged.
     */
    public void verifyPrivilege() {
        if (!isPrivileged())
            no();
    }

    /**
     * Verify that the caller is privileged, so as to check the specified
     * permission.
     *
     * @param   perm    the permission being checked.
     */
    public void verifyPrivilege(Permission perm) {
        if (!isPrivileged())
            no(perm);
    }

    private File testDirectory;
    private boolean fullySecure;

    private Thread privileged = null;
    private int privilegeCount = 0;

    private static final char dbgSecurity = 'S';
}
```

```
// PART OF THE MACHINE SIMULATION. DO NOT CHANGE.

package nachos.security;

import nachos.machine.*;
import nachos.threads.KThread;

import java.util.LinkedList;
import java.util.Iterator;
import java.security.PrivilegedAction;
import java.security.PrivilegedExceptionAction;
import java.security.PrivilegedActionException;

/**
 * A capability that allows privileged access to the Nachos machine.
 *
 * <p>
 * Some privileged operations are guarded by the Nachos security manager:
 * <ol>
 * <li>creating threads
 * <li>writing/deleting files in the test directory
 * <li>exit with specific status code
 * </ol>
 * These operations can only be performed through <tt>doPrivileged()</tt>.
 *
 * <p>
 * Some privileged operations require a capability:
 * <ol>
 * <li>scheduling interrupts
 * <li>advancing the simulated time
 * <li>accessing machine statistics
 * <li>installing a console
 * <li>flushing the simulated processor's pipeline
 * <li>approving TCB operations
 * </ol>
 * These operations can be directly performed using a <tt>Privilege</tt>
 * object.
 *
 * <p>
 * The Nachos kernel should <i>never</i> be able to directly perform any of
 * these privileged operations. If you have discovered a loophole somewhere,
 * notify someone.
 */
public abstract class Privilege {
    /**
     * Allocate a new <tt>Privilege</tt> object. Note that this object in
     * itself does not encapsulate privileged access until the machine devices
     * fill it in.
     */
    public Privilege() {
    }

    /**
     * Perform the specified action with privilege.
     *
     * @param   action  the action to perform.
     */
    public abstract void doPrivileged(Runnable action);

    /**
     * Perform the specified <tt>PrivilegedAction</tt> with privilege.
     *
     * @param   action  the action to perform.
     * @return  the return value of the action.
     */
    public abstract Object doPrivileged(PrivilegedAction action);

    /**
     * Perform the specified <tt>PrivilegedExceptionAction</tt> with privilege.
     *
     * @param   action  the action to perform.
     * @return  the return value of the action.
     */
    public abstract Object doPrivileged(PrivilegedExceptionAction action)
        throws PrivilegedActionException;

    /**
     * Exit Nachos with the specified status.
     *
     * @param   exitStatus      the exit status of the Nachos process.
     */
    public abstract void exit(int exitStatus);

    /**
     * Add an <tt>exit()</tt> notification handler. The handler will be invoked
     * by exit().
     *
     * @param   handler the notification handler.
     */
    public void addExitNotificationHandler(Runnable handler) {
        exitNotificationHandlers.add(handler);
    }

    /**
     * Invoke each <tt>exit()</tt> notification handler added by
     * <tt>addExitNotificationHandler()</tt>. Called by <tt>exit()</tt>.
     */
    protected void invokeExitNotificationHandlers() {
        for (Iterator i=exitNotificationHandlers.iterator(); i.hasNext(); ) {
            try {
                ((Runnable) i.next()).run();
            }
            catch (Throwable e) {
                System.out.println("exit() notification handler failed");
            }
        }
    }

    private LinkedList<Runnable> exitNotificationHandlers =
        new LinkedList<Runnable>();

    /** Nachos runtime statistics. */
    public Stats stats = null;

    /** Provides access to some private <tt>Machine</tt> methods. */
    public MachinePrivilege machine = null;
    /** Provides access to some private <tt>Interrupt</tt> methods. */
    public InterruptPrivilege interrupt = null;
    /** Provides access to some private <tt>Processor</tt> methods. */
    public ProcessorPrivilege processor = null;
    /** Provides access to some private <tt>TCB</tt> methods. */
    public TCBPrivilege tcb = null;

    /**
     * An interface that provides access to some private <tt>Machine</tt>
     * methods.
     */
    public interface MachinePrivilege {
        /**
         * Install a hardware console.
```

```
         *
         * @param       console the new hardware console.
         */
        public void setConsole(SerialConsole console);
    }

    /**
     * An interface that provides access to some private <tt>Interrupt</tt>
     * methods.
     */
    public interface InterruptPrivilege {
        /**
         * Schedule an interrupt to occur at some time in the future.
         *
         * @param       when    the number of ticks until the interrupt should
         *                      occur.
         * @param       type    a name for the type of interrupt being
         *                      scheduled.
         * @param       handler the interrupt handler to call.
         */
        public void schedule(long when, String type, Runnable handler);

        /**
         * Advance the simulated time.
         *
         * @param inKernelMode  <tt>true</tt> if the current thread is running kernel
         *              code, <tt>false</tt> if the current thread is running
         *              MIPS user code.
         */
        public void tick(boolean inKernelMode);
    }

    /**
     * An interface that provides access to some private <tt>Processor</tt>
     * methods.
     */
    public interface ProcessorPrivilege {
        /**
         * Flush the processor pipeline in preparation for switching to kernel
         * mode.
         */
        public void flushPipe();
    }

    /**
     * An interface that provides access to some private <tt>TCB</tt> methods.
     */
    public interface TCBPrivilege {
        /**
         * Associate the current TCB with the specified <tt>KThread</tt>.
         * <tt>AutoGrader.runningThread()</tt> <i>must</i> call this method
         * before returning.
         *
         * @param       thread  the current thread.
         */
        public void associateThread(KThread thread);
        /**
         * Authorize the TCB associated with the specified thread to be
         * destroyed.
         *
         * @param       thread  the thread whose TCB is about to be destroyed.
         */
        public void authorizeDestroy(KThread thread);
    }
}
```

```
# GNU Makefile for building user programs to run on top of Nachos
#
# Things to be aware of:
#
#    The value of the ARCHDIR environment variable must be set before using
#       this makefile. If you are using an instructional machine, this should
#       be automatic. However, if you are not using an instructional machine,
#       you need to point ARCHDIR at the cross-compiler directory, e.g.
#               setenv ARCHDIR ../mips-x86.win32-xgcc

# you need to point to the right executables
GCCDIR = $(ARCHDIR)/mips-

ASFLAGS = -mips1
CPPFLAGS =
CFLAGS = -O2 -B$(GCCDIR) -G 0 -Wa,-mips1 -nostdlib -ffreestanding
LDFLAGS = -s -T script -N -warn-common -warn-constructors -warn-multiple-gp

CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld
CPP = $(GCCDIR)cpp
AR = $(GCCDIR)ar
RANLIB = $(GCCDIR)ranlib

STDLIB_H = stdio.h stdlib.h ag.h
STDLIB_C = stdio.c stdlib.c
STDLIB_O = start.o stdio.o stdlib.o

LIB = assert atoi printf readline stdio strncmp strcat strcmp strcpy strlen memcpy mem
set
NLIB = libnachos.a

TARGETS = halt sh matmult sort echo cat cp mv rm #chat chatserver

.SECONDARY: $(patsubst %.c,%.o,$(wildcard *.c))

all: $(patsubst %,%.coff,$(TARGETS))

ag: grade-file.coff grade-exec.coff grade-mini.coff grade-dumb.coff

clean:
        rm -f strt.s *.o *.coff $(NLIB)

agclean: clean
        rm -f f1-* f2-*

$(NLIB): $(patsubst %,$(NLIB)(%.o),$(LIB)) start.o
        $(RANLIB) $(NLIB)

start.o: start.s syscall.h
        $(CPP) $(CPPFLAGS) start.s > strt.s
        $(AS) $(ASFLAGS) -o start.o strt.s
        rm strt.s

%.o: %.c *.h
        $(CC) $(CFLAGS) -c $<

%.coff: %.o $(NLIB)
        $(LD) $(LDFLAGS) -o $@ $< start.o -lnachos
```

```
#include "stdio.h"
#include "stdlib.h"

void __assert(char* file, int line) {
  printf("\nAssertion failed: line %d file %s\n", line, file);
  exit(1);
}
```

```
#include "stdlib.h"

int atoi(const char *s) {
  int result=0, sign=1;

  if (*s == -1) {
    sign = -1;
    s++;
  }

  while (*s >= '0' && *s <= '9')
    result = result*10 + (*(s++)-'0');

  return result*sign;
}
```

```
#include "syscall.h"
#include "stdio.h"
#include "stdlib.h"

#define BUFSIZE 1024

char buf[BUFSIZE];

int main(int argc, char** argv)
{
  int fd, amount;

  if (argc!=2) {
    printf("Usage: cat <file>\n");
    return 1;
  }

  fd = open(argv[1]);
  if (fd==-1) {
    printf("Unable to open %s\n", argv[1]);
    return 1;
  }

  while ((amount = read(fd, buf, BUFSIZE))>0) {
    write(1, buf, amount);
  }

  close(fd);

  return 0;
}
```

```
#include "syscall.h"
#include "stdio.h"
#include "stdlib.h"

#define BUFSIZE 1024

char buf[BUFSIZE];

int main(int argc, char** argv)
{
  int src, dst, amount;

  if (argc!=3) {
    printf("Usage: cp <src> <dst>\n");
    return 1;
  }

  src = open(argv[1]);
  if (src==-1) {
    printf("Unable to open %s\n", argv[1]);
    return 1;
  }

  creat(argv[2]);
  dst = open(argv[2]);
  if (dst==-1) {
    printf("Unable to create %s\n", argv[2]);
    return 1;
  }

  while ((amount = read(src, buf, BUFSIZE))>0) {
    write(dst, buf, amount);
  }

  close(src);
  close(dst);

  return 0;
}
```

```
#include "stdio.h"
#include "stdlib.h"

int main(int argc, char** argv)
{
  int i;

  printf("%d arguments\n", argc);

  for (i=0; i<argc; i++)
    printf("arg %d: %s\n", i, argv[i]);

  return 0;
}
```

```
/* halt.c
 *      Simple program to test whether running a user program works.
 *
 *      Just do a "syscall" that shuts down the OS.
 *
 *      NOTE: for some reason, user programs with global data structures
 *      sometimes haven't worked in the Nachos environment.  So be careful
 *      out there!  One option is to allocate data structures as
 *      automatics within a procedure, but if you do this, you have to
 *      be careful to allocate a big enough stack to hold the automatics!
 */

#include "syscall.h"

int
main()
{
    halt();
    /* not reached */
}
```

```
/* matmult.c
 *    Test program to do matrix multiplication on large arrays.
 *
 *    Intended to stress virtual memory system. Should return 7220 if Dim==20
 */

#include "syscall.h"

#define Dim     20      /* sum total of the arrays doesn't fit in
                         * physical memory
                         */

int A[Dim][Dim];
int B[Dim][Dim];
int C[Dim][Dim];

int
main()
{
    int i, j, k;

    for (i = 0; i < Dim; i++)           /* first initialize the matrices */
        for (j = 0; j < Dim; j++) {
             A[i][j] = i;
             B[i][j] = j;
             C[i][j] = 0;
        }

    for (i = 0; i < Dim; i++)           /* then multiply them together */
        for (j = 0; j < Dim; j++)
            for (k = 0; k < Dim; k++)
                 C[i][j] += A[i][k] * B[k][j];

    printf("C[%d][%d] = %d\n", Dim-1, Dim-1, C[Dim-1][Dim-1]);
    return (C[Dim-1][Dim-1]);           /* and then we're done */
}
```

```
#include "stdlib.h"

void *memcpy(void *s1, const void *s2, unsigned n) {
  int i;

  for (i=0; i<n; i++)
    ((char*)s1)[i] = ((char*)s2)[i];

  return s1;
}
```

```
#include "stdlib.h"

void *memset(void *s, int c, unsigned int n) {
  int i;

  for (i=0; i<n; i++)
    ((char*)s)[i] = (char) c;

  return s;
}
```

```
#include "syscall.h"
#include "stdio.h"
#include "stdlib.h"

#define BUFSIZE 1024

char buf[BUFSIZE];

int main(int argc, char** argv)
{
  int src, dst, amount;

  if (argc!=3) {
    printf("Usage: cp <src> <dst>\n");
    return 1;
  }

  src = open(argv[1]);
  if (src==-1) {
    printf("Open to open %s\n", argv[1]);
    return 1;
  }

  creat(argv[2]);
  dst = open(argv[2]);
  if (dst==-1) {
    printf("Unable to create %s\n", argv[2]);
    return 1;
  }

  while ((amount = read(src, buf, BUFSIZE))>0) {
    write(dst, buf, amount);
  }

  close(src);
  close(dst);
  unlink(argv[1]);

  return 0;
}
```

```
#include "stdio.h"
#include "stdlib.h"

static char digittoascii(unsigned n, int uppercase) {
    assert(n<36);

    if (n<=9)
        return '0'+n;
    else if (uppercase)
        return 'A'+n-10;
    else
        return 'a'+n-10;
}

static int charprint(char **s, char c) {
    *((*s)++) = c;

    return 1;
}

static int mcharprint(char **s, char* chars, int length) {
    memcpy(*s, chars, length);
    *s += length;

    return length;
}

static int integerprint(char **s, int n, unsigned base, int min, int zpad, int upper)
{
    char buf[32];
    int i=32, digit, len=0;

    assert(base>=2 && base < 36);

    if (min>32)
        min=32;

    if (n==0) {
        for (i=1; i<min; i++)
            len += charprint(s, zpad ? '0' : ' ');

        len += charprint(s, '0');
        return len;
    }

    if (n<0) {
        len += charprint(s, '-');
        n *= -1;
    }

    while (n!=0) {
        digit = n%base;
        n /= base;

        if (digit<0)
            digit *= -1;

        buf[--i] = digittoascii(digit, upper);
    }

    while (i>32-min)
        buf[--i] = zpad ? '0' : ' ';

    len += mcharprint(s, &buf[i], 32-i);
    return len;
}

static int stringprint(char **s, char *string) {
    return mcharprint(s, string, strlen(string));
}

static int _vsprintf(char *s, char *format, va_list ap) {
    int min,zpad,len=0,regular=0;
    char *temp;

    /* process format string */
    while (*format != 0) {
        /* if switch, process */
        if (*format == '%') {
            if (regular > 0) {
                len += mcharprint(&s, format-regular, regular);
                regular = 0;
            }
            format++;
            /* bug: '-' here will potentially screw things up */
            assert(*format != '-');

            min=zpad=0;

            if (*format == '0')
                zpad=1;

            min = atoi(format);

            temp = format;
            while (*temp >= '0' && *temp <= '9')
                temp++;

            switch (*(temp++)) {

            case 'c':
                len += charprint(&s, va_arg(ap, int));
                break;

            case 'd':
                len += integerprint(&s, va_arg(ap, int), 10, min, zpad, 0);
                break;

            case 'x':
                len += integerprint(&s, va_arg(ap, int), 16, min, zpad, 0);
                break;

            case 'X':
                len += integerprint(&s, va_arg(ap, int), 16, min, zpad, 1);
                break;

            case 's':
                len += stringprint(&s, (char*) va_arg(ap, int));
                break;

            default:
                len += charprint(&s, '%');
                temp = format;
            }

            format = temp;
        }
        else {
            regular++;
            format++;
```

```
        }
    }

    if (regular > 0) {
        len += mcharprint(&s, format-regular, regular);
        regular = 0;
    }

    *s = 0;

    return len;
}

void vsprintf(char *s, char *format, va_list ap) {
    _vsprintf(s, format, ap);
}

static char vfprintfbuf[256];

void vfprintf(int fd, char *format, va_list ap) {
    int len = _vsprintf(vfprintfbuf, format, ap);
    assert(len < sizeof(vfprintfbuf));
    write(fd, vfprintfbuf, len);
}

void vprintf(char *format, va_list ap) {
    vfprintf(stdout, format, ap);
}

void sprintf(char *s, char *format, ...) {
    va_list ap;
    va_start(ap,format);

    vsprintf(s, format, ap);

    va_end(ap);
}

void fprintf(int fd, char *format, ...) {
    va_list ap;
    va_start(ap,format);

    vfprintf(fd, format, ap);

    va_end(ap);
}

void printf(char *format, ...) {
    va_list ap;
    va_start(ap,format);

    vprintf(format, ap);

    va_end(ap);
}
```

```
#include "stdio.h"
#include "stdlib.h"

void readline(char *s, int maxlength) {
  int i = 0;

  while (1) {
    char c = getch();
    /* if end of line, finish up */
    if (c == '\n') {
      putchar('\n');
      s[i] = 0;
      return;
    }
    /* else if backspace... */
    else if (c == '\b') {
      /* if nothing to delete, beep */
      if (i == 0) {
        beep();
      }
      /* else delete it */
      else {
        printf("\b \b");
        i--;
      }
    }
    /* else if bad character or no room for more, beep */
    else if (c < 0x20 || i+1 == maxlength) {
      beep();
    }
    /* else add the character */
    else {
      s[i++] = c;
      putchar(c);
    }
  }
}
```

```
#include "syscall.h"
#include "stdio.h"
#include "stdlib.h"

int main(int argc, char** argv)
{
  if (argc!=2) {
    printf("Usage: rm <file>\n");
    return 1;
  }

  if (unlink(argv[1]) != 0) {
      printf("Unable to remove %s\n", argv[1]);
      return 1;
  }

  return 0;
}
```

```
OUTPUT_FORMAT("ecoff-littlemips")
SEARCH_DIR(.)
ENTRY(__start)

SECTIONS {
  .text         0               : { *(.text) }
  .rdata        BLOCK(0x400)    : { *(.rdata) }
  .data         BLOCK(0x400)    : { *(.data) }
  .sbss         BLOCK(0x400)    : { *(.sbss) }
  .bss          BLOCK(0x400)    : { *(.bss) }
  .scommon      BLOCK(0x400)    : { *(.scommon) }
}
```

```
#include "stdio.h"
#include "stdlib.h"

#define BUFFERSIZE      64

#define MAXARGSIZE      16
#define MAXARGS         16

/**
 * tokenizeCommand
 *
 * Splits the specified command line into tokens, creating a token array with a maximu
m
 * of maxTokens entries, using storage to hold the tokens. The storage array should be
 as
 * long as the command line.
 *
 * Whitespace (spaces, tabs, newlines) separate tokens, unless
 * enclosed in double quotes. Any character can be quoted by preceeding
 * it with a backslash. Quotes must be terminated.
 *
 * Returns the number of tokens, or -1 on error.
 */
static int tokenizeCommand(char* command, int maxTokens, char *tokens[], char* storage
) {
    const int quotingCharacter = 0x00000001;
    const int quotingString = 0x00000002;
    const int startedArg = 0x00000004;

    int state = 0;
    int numTokens = 0;

    char c;

    assert(maxTokens > 0);

    while ((c = *(command++)) != '\0') {
        if (state & quotingCharacter) {
            switch (c) {
            case 't':
                c = '\t';
                break;
            case 'n':
                c = '\n';
                break;
            }
            *(storage++) = c;
            state &= ~quotingCharacter;
        }
        else if (state & quotingString) {
            switch (c) {
            case '\\':
                state |= quotingCharacter;
                break;
            case '"':
                state &= ~quotingString;
                break;
            default:
                *(storage++) = c;
                break;
            }
        }
        else {
            switch (c) {
            case ' ':
            case '\t':
            case '\n':
                if (state & startedArg) {
                    *(storage++) = '\0';
                    state &= ~startedArg;
                }
                break;
            default:
                if (!(state & startedArg)) {
                    if (numTokens == maxTokens) {
                        return -1;
                    }
                    tokens[numTokens++] = storage;
                    state |= startedArg;
                }

                switch (c) {
                case '\\':
                    state |= quotingCharacter;
                    break;
                case '"':
                    state |= quotingString;
                    break;
                default:
                    *(storage++) = c;
                    break;
                }
            }
        }
    }

    if (state & quotingCharacter) {
        printf("Unmatched \\.\n");
        return -1;
    }

    if (state & quotingString) {
        printf("Unmatched \".\n");
        return -1;
    }

    if (state & startedArg) {
        *(storage++) = '\0';
    }

    return numTokens;
}

void runline(char* line) {
    int pid, background, status;

    char args[BUFFERSIZE], prog[BUFFERSIZE];
    char *argv[MAXARGS];

    int argc = tokenizeCommand(line, MAXARGS, argv, args);
    if (argc <= 0)
        return;

    if (argc > 0 && strcmp(argv[argc-1], "&") == 0) {
        argc--;
        background = 1;
    }
    else {
        background = 0;
    }
```

```
    if (argc > 0) {
        if (strcmp(argv[0], "exit")==0) {
            if (argc == 1) {
                exit(0);
            }
            else if (argc == 2) {
                exit(atoi(argv[1]));
            }
            else {
                printf("exit: Expression Syntax.\n");
                return;
            }
        }
        else if (strcmp(argv[0], "halt")==0) {
            if (argc == 1) {
                halt();
                printf("Not the root process!\n");
            }
            else {
                printf("halt: Expression Syntax.\n");
            }
            return;
        }
        else if (strcmp(argv[0], "join")==0) {
            if (argc == 2) {
                pid = atoi(argv[1]);
            }
            else {
                printf("join: Expression Syntax.\n");
                return;
            }
        }
        else {
            strcpy(prog, argv[0]);
            strcat(prog, ".coff");

            pid = exec(prog, argc, argv);
            if (pid == -1) {
                printf("%s: exec failed.\n", argv[0]);
                return;
            }
        }

        if (!background) {
            switch (join(pid, &status)) {
            case -1:
                printf("join: Invalid process ID.\n");
                break;
            case 0:
                printf("\n[%d] Unhandled exception\n", pid);
                break;
            case 1:
                printf("\n[%d] Done (%d)\n", pid, status);
                break;
            }
        }
        else {
            printf("\n[%d]\n", pid);
        }
    }
}

int main(int argc, char *argv[]) {
    char prompt[] = "nachos% ";
    char buffer[BUFFERSIZE];

    while (1) {
        printf("%s", prompt);

        readline(buffer, BUFFERSIZE);

        runline(buffer);
    }
}
```

```
/* sort.c
 *    Test program to sort a large number of integers.
 *
 *    Intention is to stress virtual memory system. To increase the memory
 *    usage of this program, simply increase SORTSHIFT. The size of the array
 *    is (SORTSIZE)(2^(SORTSHIFT+2)).
 */

#include "syscall.h"

/* size of physical memory; with code, we'll run out of space! */
#define SORTSIZE        256
#define SORTSHIFT       0

int array[SORTSIZE<<SORTSHIFT];

#define A(i)    (array[(i)<<SORTSHIFT])

void swap(int* x, int* y)
{
  int temp = *x;
  *x = *y;
  *y = temp;
}

int
main()
{
  int i, j;

  /* first initialize the array, in reverse sorted order */
  for (i=0; i<SORTSIZE; i++)
    A(i) = (SORTSIZE-1)-i;

  /* then sort! */
  for (i=0; i<SORTSIZE-1; i++) {
    for (j=i; j<SORTSIZE; j++) {
      if (A(i) > A(j))
        swap(&A(i), &A(j));
    }
  }

  /* and last, verify */
  for (i=0; i<SORTSIZE; i++) {
    if (A(i) != i)
      return 1;
  }

  /* if successful, return 0 */
  return 0;
}
```

```
/* Start.s
 *      Assembly language assist for user programs running on top of Nachos.
 *
 *      Since we don't want to pull in the entire C library, we define
 *      what we need for a user program here, namely Start and the system
 *      calls.
 */

#define START_S
#include "syscall.h"

        .text
        .align  2

/* -------------------------------------------------------------
 * __start
 *      Initialize running a C program, by calling "main".
 * -------------------------------------------------------------
 */

        .globl  __start
        .ent    __start
__start:
        jal     main
        addu    $4,$2,$0
        jal     exit     /* if we return from main, exit(return value) */
        .end    __start

        .globl  __main
        .ent    __main
__main:
        jr      $31
        .end    __main

/* -------------------------------------------------------------
 * System call stubs:
 *      Assembly language assist to make system calls to the Nachos kernel.
 *      There is one stub per system call, that places the code for the
 *      system call into register r2, and leaves the arguments to the
 *      system call alone (in other words, arg1 is in r4, arg2 is
 *      in r5, arg3 is in r6, arg4 is in r7)
 *
 *      The return value is in r2. This follows the standard C calling
 *      convention on the MIPS.
 * -------------------------------------------------------------
 */

#define SYSCALLSTUB(name, number) \
        .globl  name            ; \
        .ent    name            ; \
name:                           ; \
        addiu   $2,$0,number    ; \
        syscall                 ; \
        j       $31             ; \
        .end    name

        SYSCALLSTUB(halt, syscallHalt)
        SYSCALLSTUB(exit, syscallExit)
        SYSCALLSTUB(exec, syscallExec)
        SYSCALLSTUB(join, syscallJoin)
        SYSCALLSTUB(creat, syscallCreate)
        SYSCALLSTUB(open, syscallOpen)
        SYSCALLSTUB(read, syscallRead)
        SYSCALLSTUB(write, syscallWrite)
        SYSCALLSTUB(close, syscallClose)
        SYSCALLSTUB(unlink, syscallUnlink)
        SYSCALLSTUB(mmap, syscallMmap)
        SYSCALLSTUB(connect, syscallConnect)
        SYSCALLSTUB(accept, syscallAccept)
```

```
/* stdarg.h for GNU.
   Note that the type used in va_arg is supposed to match the
   actual type **after default promotions**.
   Thus, va_arg (..., short) is not valid.  */

#ifndef _STDARG_H
#ifndef _ANSI_STDARG_H_
#ifndef __need___va_list
#define _STDARG_H
#define _ANSI_STDARG_H_
#endif /* not __need___va_list */
#undef __need___va_list

#ifdef __clipper__
#include "va-clipper.h"
#else
#ifdef __m88k__
#include "va-m88k.h"
#else
#ifdef __i860__
#include "va-i860.h"
#else
#ifdef __hppa__
#include "va-pa.h"
#else
#ifdef __mips__
#include "va-mips.h"
#else
#ifdef __sparc__
#include "va-sparc.h"
#else
#ifdef __i960__
#include "va-i960.h"
#else
#ifdef __alpha__
#include "va-alpha.h"
#else
#if defined (__H8300__) || defined (__H8300H__) || defined (__H8300S__)
#include "va-h8300.h"
#else
#if defined (__PPC__) && (defined (_CALL_SYSV) || defined (_WIN32))
#include "va-ppc.h"
#else
#ifdef __arc__
#include "va-arc.h"
#else
#ifdef __M32R__
#include "va-m32r.h"
#else
#ifdef __sh__
#include "va-sh.h"
#else
#ifdef __mn10300__
#include "va-mn10300.h"
#else
#ifdef __mn10200__
#include "va-mn10200.h"
#else
#ifdef __v850__
#include "va-v850.h"
#else

/* Define __gnuc_va_list.  */

#ifndef __GNUC_VA_LIST
#define __GNUC_VA_LIST
#if defined(__svr4__) || defined(_AIX) || defined(_M_UNIX) || defined(__NetBSD__)
typedef char *__gnuc_va_list;
#else
typedef void *__gnuc_va_list;
#endif
#endif

/* Define the standard macros for the user,
   if this invocation was from the user program.  */
#ifdef _STDARG_H

/* Amount of space required in an argument list for an arg of type TYPE.
   TYPE may alternatively be an expression whose type is used.  */

#if defined(sysV68)
#define __va_rounded_size(TYPE)  \
  (((sizeof (TYPE) + sizeof (short) - 1) / sizeof (short)) * sizeof (short))
#else
#define __va_rounded_size(TYPE)  \
  (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
#endif

#define va_start(AP, LASTARG)                                           \
 (AP = ((__gnuc_va_list) __builtin_next_arg (LASTARG)))

#undef va_end
void va_end (__gnuc_va_list);           /* Defined in libgcc.a */
#define va_end(AP)      ((void)0)

/* We cast to void * and then to TYPE * because this avoids
   a warning about increasing the alignment requirement.  */

#if (defined (__arm__) && ! defined (__ARMEB__)) || defined (__i386__) || defined (__i
860__) || defined (__ns32000__) || defined (__vax__)
/* This is for little-endian machines; small args are padded upward.  */
#define va_arg(AP, TYPE)                                                \
 (AP = (__gnuc_va_list) ((char *) (AP) + __va_rounded_size (TYPE)),     \
  *((TYPE *) (void *) ((char *) (AP) - __va_rounded_size (TYPE))))
#else /* big-endian */
/* This is for big-endian machines; small args are padded downward.  */
#define va_arg(AP, TYPE)                                                \
 (AP = (__gnuc_va_list) ((char *) (AP) + __va_rounded_size (TYPE)),     \
  *((TYPE *) (void *) ((char *) (AP)                                    \
                       - ((sizeof (TYPE) < __va_rounded_size (char)     \
                           ? sizeof (TYPE) : __va_rounded_size (TYPE))))))
#endif /* big-endian */

/* Copy __gnuc_va_list into another variable of this type.  */
#define __va_copy(dest, src) (dest) = (src)

#endif /* _STDARG_H */

#endif /* not v850 */
#endif /* not mn10200 */
#endif /* not mn10300 */
#endif /* not sh */
#endif /* not m32r */
#endif /* not arc */
#endif /* not powerpc with V.4 calling sequence */
#endif /* not h8300 */
#endif /* not alpha */
#endif /* not i960 */
#endif /* not sparc */
#endif /* not mips */
```

```
#endif /* not hppa */
#endif /* not i860 */
#endif /* not m88k */
#endif /* not clipper */

#ifdef _STDARG_H
/* Define va_list, if desired, from __gnuc_va_list. */
/* We deliberately do not define va_list when called from
   stdio.h, because ANSI C says that stdio.h is not supposed to define
   va_list.  stdio.h needs to have access to that data type,
   but must not use that name.  It should use the name __gnuc_va_list,
   which is safe because it is reserved for the implementation.  */

#ifdef _HIDDEN_VA_LIST  /* On OSF1, this means varargs.h is "half-loaded".  */
#undef _VA_LIST
#endif

#ifdef _BSD_VA_LIST
#undef _BSD_VA_LIST
#endif

#if defined(__svr4__) || (defined(_SCO_DS) && !defined(__VA_LIST))
/* SVR4.2 uses _VA_LIST for an internal alias for va_list,
   so we must avoid testing it and setting it here.
   SVR4 uses _VA_LIST as a flag in stdarg.h, but we should
   have no conflict with that.  */
#ifndef _VA_LIST_
#define _VA_LIST_
#ifdef __i860__
#ifndef _VA_LIST
#define _VA_LIST va_list
#endif
#endif /* __i860__ */
typedef __gnuc_va_list va_list;
#ifdef _SCO_DS
#define __VA_LIST
#endif
#endif /* _VA_LIST_ */
#else /* not __svr4__ || _SCO_DS */

/* The macro _VA_LIST_ is the same thing used by this file in Ultrix.
   But on BSD NET2 we must not test or define or undef it.
   (Note that the comments in NET 2's ansi.h
   are incorrect for _VA_LIST_--see stdio.h!)  */
#if !defined (_VA_LIST_) || defined (__BSD_NET2__) || defined (____386BSD____) || defi
ned (__bsdi__) || defined (__sequent__) || defined (__FreeBSD__) || defined(WINNT)
/* The macro _VA_LIST_DEFINED is used in Windows NT 3.5  */
#ifndef _VA_LIST_DEFINED
/* The macro _VA_LIST is used in SCO Unix 3.2.  */
#ifndef _VA_LIST
/* The macro _VA_LIST_T_H is used in the Bull dpx2  */
#ifndef _VA_LIST_T_H
typedef __gnuc_va_list va_list;
#endif /* not _VA_LIST_T_H */
#endif /* not _VA_LIST */
#endif /* not _VA_LIST_DEFINED */
#if !(defined (__BSD_NET2__) || defined (____386BSD____) || defined (__bsdi__) || defi
ned (__sequent__) || defined (__FreeBSD__))
#define _VA_LIST_
#endif
#ifndef _VA_LIST
#define _VA_LIST
#endif
#ifndef _VA_LIST_DEFINED
#define _VA_LIST_DEFINED
#endif
#ifndef _VA_LIST_T_H
#define _VA_LIST_T_H
#endif

#endif /* not _VA_LIST_, except on certain systems */

#endif /* not __svr4__ */

#endif /* _STDARG_H */

#endif /* not _ANSI_STDARG_H_ */
#endif /* not _STDARG_H */
```

```
#include "stdio.h"
#include "stdlib.h"

int fgetc(int fd) {
    unsigned char c;

    while (read(fd, &c, 1) != 1);

    return c;
}

void fputc(char c, int fd) {
    write(fd, &c, 1);
}

void fputs(const char *s, int fd) {
    write(fd, (char*) s, strlen(s));
}
```

```
/*----------------------------------------------------------------
 * stdio.h
 *
 * Header file for standard I/O routines.
 *----------------------------------------------------------------*/

#ifndef STDIO_H
#define STDIO_H

#include "syscall.h"
#include "stdarg.h"

typedef int             FILE;
#define stdin           fdStandardInput
#define stdout          fdStandardOutput

int  fgetc(FILE stream);
void readline(char *s, int maxlength);
int  tryreadline(char *s, char c, int maxlength);

#define getc(stream)    fgetc(stream)
#define getchar()       getc(stdin)
#define getch()         getchar()

void fputc(char c, FILE stream);
void fputs(const char *s, FILE stream);

#define puts(s)         fputs(s,stdout)
#define putc(c,stream)  fputc(c,stream)
#define putchar(c)      putc(c,stdout)
#define beep()          putchar(0x07)

void vsprintf(char *s, char *format, va_list ap);
void vfprintf(FILE f, char *format, va_list ap);
void vprintf(char *format, va_list ap);
void sprintf(char *s, char *format, ...);
void fprintf(FILE f, char *format, ...);
void printf(char *format, ...);

#endif // STDIO_H
```

```
#include "stdlib.h"
```

```
/*----------------------------------------------------------------
 * stdlib.h
 *
 * Header file for standard library functions.
 *----------------------------------------------------------------*/

#ifndef STDLIB_H
#define STDLIB_H

#include "syscall.h"

#define null    0L
#define true    1
#define false   0

#define min(a,b)  (((a) < (b)) ? (a) : (b))
#define max(a,b)  (((a) > (b)) ? (a) : (b))

#define divRoundDown(n,s)  ((n) / (s))
#define divRoundUp(n,s)    (((n) / (s)) + ((((n) % (s)) > 0) ? 1 : 0))

#define assert(_EX)     ((_EX) ? (void) 0 : __assert(__FILE__, __LINE__))
void __assert(char* file, int line);

#define assertNotReached()      assert(false)

void *memcpy(void *s1, const void *s2, unsigned int n);
void *memset(void *s, int c, unsigned int n);

unsigned int strlen(const char *str);
char *strcpy(char *dst, const char *src);
int strcmp(const char *a, const char *b);
int strncmp(const char *a, const char *b, int n);

int atoi(const char *s);

#endif // STDLIB_H
```

```
#include "stdlib.h"

/* concatenates s2 to the end of s1 and returns s1 */
char *strcat(char *s1, const char *s2) {
    char* result = s1;

    while (*s1 != 0)
        s1++;

    do {
        *(s1++) = *(s2);
    }
    while (*(s2++) != 0);

    return result;
}
```

```
#include "stdlib.h"

/* lexicographically compares a and b */
int strcmp(const char* a, const char* b) {
  do {
    if (*a < *b)
      return -1;
    if (*a > *b)
      return 1;
  }
  while (*(a++) != 0 && *(b++) != 0);

  return 0;
}
```

```
#include "stdlib.h"

/* copies src to dst, returning dst */
char *strcpy(char *dst, const char *src) {
  int n=0;
  char *result = dst;

  do {
    *(dst++) = *src;
    n++;
  }
  while (*(src++) != 0);

  return result;
}
```

```
#include "stdlib.h"

/* returns the length of the character string str, not including the null-terminator *
/
unsigned strlen(const char *str) {
  int result=0;

  while (*(str++) != 0)
    result++;

  return result;
}
```

```
#include "stdlib.h"

/* lexicographically compares a and b up to n chars */
int strncmp(const char* a, const char* b, int n)
{
  assert(n > 0);

  do {
    if (*a < *b)
      return -1;
    if (*a > *b)
      return 1;
    n--;
    a++;
    b++;
  }
  while (n > 0);

  return 0;
}
```

```
/**
 * The Nachos system call interface. These are Nachos kernel operations that
 * can be invoked from user programs using the syscall instruction.
 *
 * This interface is derived from the UNIX syscalls. This information is
 * largely copied from the UNIX man pages.
 */

#ifndef SYSCALL_H
#define SYSCALL_H

/**
 * System call codes, passed in $r0 to tell the kernel which system call to do.
 */
#define syscallHalt             0
#define syscallExit             1
#define syscallExec             2
#define syscallJoin             3
#define syscallCreate           4
#define syscallOpen             5
#define syscallRead             6
#define syscallWrite            7
#define syscallClose            8
#define syscallUnlink           9
#define syscallMmap             10
#define syscallConnect          11
#define syscallAccept           12

/* Don't want the assembler to see C code, but start.s includes syscall.h. */
#ifndef START_S

/* When a process is created, two streams are already open. File descriptor 0
 * refers to keyboard input (UNIX stdin), and file descriptor 1 refers to
 * display output (UNIX stdout). File descriptor 0 can be read, and file
 * descriptor 1 can be written, without previous calls to open().
 */
#define fdStandardInput         0
#define fdStandardOutput        1

/* The system call interface. These are the operations the Nachos kernel needs
 * to support, to be able to run user programs.
 *
 * Each of these is invoked by a user program by simply calling the procedure;
 * an assembly language stub stores the syscall code (see above) into $r0 and
 * executes a syscall instruction. The kernel exception handler is then
 * invoked.
 */

/* Halt the Nachos machine by calling Machine.halt(). Only the root process
 * (the first process, executed by UserKernel.run()) should be allowed to
 * execute this syscall. Any other process should ignore the syscall and return
 * immediately.
 */
void halt();

/* PROCESS MANAGEMENT SYSCALLS: exit(), exec(), join() */

/**
 * Terminate the current process immediately. Any open file descriptors
 * belonging to the process are closed. Any children of the process no longer
 * have a parent process.
 *
 * status is returned to the parent process as this process's exit status and
 * can be collected using the join syscall. A process exiting normally should
 * (but is not required to) set status to 0.
 *
 * exit() never returns.
 */
void exit(int status);

/**
 * Execute the program stored in the specified file, with the specified
 * arguments, in a new child process. The child process has a new unique
 * process ID, and starts with stdin opened as file descriptor 0, and stdout
 * opened as file descriptor 1.
 *
 * file is a null-terminated string that specifies the name of the file
 * containing the executable. Note that this string must include the ".coff"
 * extension.
 *
 * argc specifies the number of arguments to pass to the child process. This
 * number must be non-negative.
 *
 * argv is an array of pointers to null-terminated strings that represent the
 * arguments to pass to the child process. argv[0] points to the first
 * argument, and argv[argc-1] points to the last argument.
 *
 * exec() returns the child process's process ID, which can be passed to
 * join(). On error, returns -1.
 */
int exec(char *file, int argc, char *argv[]);

/**
 * Suspend execution of the current process until the child process specified
 * by the processID argument has exited. If the child has already exited by the
 * time of the call, returns immediately. When the current process resumes, it
 * disowns the child process, so that join() cannot be used on that process
 * again.
 *
 * processID is the process ID of the child process, returned by exec().
 *
 * status points to an integer where the exit status of the child process will
 * be stored. This is the value the child passed to exit(). If the child exited
 * because of an unhandled exception, the value stored is not defined.
 *
 * If the child exited normally, returns 1. If the child exited as a result of
 * an unhandled exception, returns 0. If processID does not refer to a child
 * process of the current process, returns -1.
 */
int join(int processID, int *status);

/* FILE MANAGEMENT SYSCALLS: creat, open, read, write, close, unlink
 *
 * A file descriptor is a small, non-negative integer that refers to a file on
 * disk or to a stream (such as console input, console output, and network
 * connections). A file descriptor can be passed to read() and write() to
 * read/write the corresponding file/stream. A file descriptor can also be
 * passed to close() to release the file descriptor and any associated
 * resources.
 */

/**
 * Attempt to open the named disk file, creating it if it does not exist,
 * and return a file descriptor that can be used to access the file.
 *
 * Note that creat() can only be used to create files on disk; creat() will
 * never return a file descriptor referring to a stream.
 *
 * Returns the new file descriptor, or -1 if an error occurred.
 */
```

```
int creat(char *name);

/**
 * Attempt to open the named file and return a file descriptor.
 *
 * Note that open() can only be used to open files on disk; open() will never
 * return a file descriptor referring to a stream.
 *
 * Returns the new file descriptor, or -1 if an error occurred.
 */
int open(char *name);

/**
 * Attempt to read up to count bytes into buffer from the file or stream
 * referred to by fileDescriptor.
 *
 * On success, the number of bytes read is returned. If the file descriptor
 * refers to a file on disk, the file position is advanced by this number.
 *
 * It is not necessarily an error if this number is smaller than the number of
 * bytes requested. If the file descriptor refers to a file on disk, this
 * indicates that the end of the file has been reached. If the file descriptor
 * refers to a stream, this indicates that the fewer bytes are actually
 * available right now than were requested, but more bytes may become available
 * in the future. Note that read() never waits for a stream to have more data;
 * it always returns as much as possible immediately.
 *
 * On error, -1 is returned, and the new file position is undefined. This can
 * happen if fileDescriptor is invalid, if part of the buffer is read-only or
 * invalid, or if a network stream has been terminated by the remote host and
 * no more data is available.
 */
int read(int fileDescriptor, void *buffer, int count);

/**
 * Attempt to write up to count bytes from buffer to the file or stream
 * referred to by fileDescriptor. write() can return before the bytes are
 * actually flushed to the file or stream. A write to a stream can block,
 * however, if kernel queues are temporarily full.
 *
 * On success, the number of bytes written is returned (zero indicates nothing
 * was written), and the file position is advanced by this number. It IS an
 * error if this number is smaller than the number of bytes requested. For
 * disk files, this indicates that the disk is full. For streams, this
 * indicates the stream was terminated by the remote host before all the data
 * was transferred.
 *
 * On error, -1 is returned, and the new file position is undefined. This can
 * happen if fileDescriptor is invalid, if part of the buffer is invalid, or
 * if a network stream has already been terminated by the remote host.
 */
int write(int fileDescriptor, void *buffer, int count);

/**
 * Close a file descriptor, so that it no longer refers to any file or stream
 * and may be reused.
 *
 * If the file descriptor refers to a file, all data written to it by write()
 * will be flushed to disk before close() returns.
 * If the file descriptor refers to a stream, all data written to it by write()
 * will eventually be flushed (unless the stream is terminated remotely), but
 * not necessarily before close() returns.
 *
 * The resources associated with the file descriptor are released. If the
 * descriptor is the last reference to a disk file which has been removed using
 * unlink, the file is deleted (this detail is handled by the file system
 * implementation).
 *
 * Returns 0 on success, or -1 if an error occurred.
 */
int close(int fileDescriptor);

/**
 * Delete a file from the file system. If no processes have the file open, the
 * file is deleted immediately and the space it was using is made available for
 * reuse.
 *
 * If any processes still have the file open, the file will remain in existence
 * until the last file descriptor referring to it is closed. However, creat()
 * and open() will not be able to return new file descriptors for the file
 * until it is deleted.
 *
 * Returns 0 on success, or -1 if an error occurred.
 */
int unlink(char *name);

/**
 * Map the file referenced by fileDescriptor into memory at address. The file
 * may be as large as 0x7FFFFFFF bytes.
 *
 * To maintain consistency, further calls to read() and write() on this file
 * descriptor will fail (returning -1) until the file descriptor is closed.
 *
 * When the file descriptor is closed, all remaining dirty pages of the map
 * will be flushed to disk and the map will be removed.
 *
 * Returns the length of the file on success, or -1 if an error occurred.
 */
int mmap(int fileDescriptor, char *address);

/**
 * Attempt to initiate a new connection to the specified port on the specified
 * remote host, and return a new file descriptor referring to the connection.
 * connect() does not give up if the remote host does not respond immediately.
 *
 * Returns the new file descriptor, or -1 if an error occurred.
 */
int connect(int host, int port);

/**
 * Attempt to accept a single connection on the specified local port and return
 * a file descriptor referring to the connection.
 *
 * If any connection requests are pending on the port, one request is dequeued
 * and an acknowledgement is sent to the remote host (so that its connect()
 * call can return). Since the remote host will never cancel a connection
 * request, there is no need for accept() to wait for the remote host to
 * confirm the connection (i.e. a 2-way handshake is sufficient; TCP's 3-way
 * handshake is unnecessary).
 *
 * If no connection requests are pending, returns -1 immediately.
 *
 * In either case, accept() returns without waiting for a remote host.
 *
 * Returns a new file descriptor referring to the connection, or -1 if an error
 * occurred.
 */
int accept(int port);

#endif /* START_S */
```

```
#endif /* SYSCALL_H */
```

```
/* ----------------------------------------- */
/*           VARARGS  for MIPS/GNU CC        */
/*                                           */
/*                                           */
/*                                           */
/*                                           */
/* ----------------------------------------- */


/* These macros implement varargs for GNU C--either traditional or ANSI.  */

/* Define __gnuc_va_list.  */

#ifndef __GNUC_VA_LIST
#define __GNUC_VA_LIST
#if defined (__mips_eabi) && ! defined (__mips_soft_float) && ! defined (__mips_single
_float)

typedef struct {
  /* Pointer to FP regs.  */
  char *__fp_regs;
  /* Number of FP regs remaining.  */
  int __fp_left;
  /* Pointer to GP regs followed by stack parameters.  */
  char *__gp_regs;
} __gnuc_va_list;

#else /* ! (defined (__mips_eabi) && ! defined (__mips_soft_float) && ! defined (__mip
s_single_float)) */

typedef char * __gnuc_va_list;

#endif /* ! (defined (__mips_eabi) && ! defined (__mips_soft_float) && ! defined (__mi
ps_single_float)) */
#endif /* not __GNUC_VA_LIST */

/* If this is for internal libc use, don't define anything but
   __gnuc_va_list.  */
#if defined (_STDARG_H) || defined (_VARARGS_H)

#ifndef _VA_MIPS_H_ENUM
#define _VA_MIPS_H_ENUM
enum {
  __no_type_class = -1,
  __void_type_class,
  __integer_type_class,
  __char_type_class,
  __enumeral_type_class,
  __boolean_type_class,
  __pointer_type_class,
  __reference_type_class,
  __offset_type_class,
  __real_type_class,
  __complex_type_class,
  __function_type_class,
  __method_type_class,
  __record_type_class,
  __union_type_class,
  __array_type_class,
  __string_type_class,
  __set_type_class,
  __file_type_class,
  __lang_type_class
};
#endif


/* In GCC version 2, we want an ellipsis at the end of the declaration
   of the argument list.  GCC version 1 can't parse it.  */

#if __GNUC__ > 1
#define __va_ellipsis ...
#else
#define __va_ellipsis
#endif

#ifdef __mips64
#define __va_rounded_size(__TYPE)  \
  (((sizeof (__TYPE) + 8 - 1) / 8) * 8)
#else
#define __va_rounded_size(__TYPE)  \
  (((sizeof (__TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
#endif

#ifdef __mips64
#define __va_reg_size 8
#else
#define __va_reg_size 4
#endif

/* Get definitions for _MIPS_SIM_ABI64 etc.  */
#ifdef _MIPS_SIM
#include <sgidefs.h>
#endif

#ifdef _STDARG_H
#if defined (__mips_eabi)
#if ! defined (__mips_soft_float) && ! defined (__mips_single_float)
#ifdef __mips64
#define va_start(__AP, __LASTARG)                                          \
  (__AP.__gp_regs = ((char *) __builtin_next_arg (__LASTARG)               \
                     - (__builtin_args_info (2) < 8                        \
                        ? (8 - __builtin_args_info (2)) * __va_reg_size    \
                        : 0)),                                             \
   __AP.__fp_left = 8 - __builtin_args_info (3),                           \
   __AP.__fp_regs = __AP.__gp_regs - __AP.__fp_left * __va_reg_size)
#else /* ! defined (__mips64) */
#define va_start(__AP, __LASTARG)                                          \
  (__AP.__gp_regs = ((char *) __builtin_next_arg (__LASTARG)               \
                     - (__builtin_args_info (2) < 8                        \
                        ? (8 - __builtin_args_info (2)) * __va_reg_size    \
                        : 0)),                                             \
   __AP.__fp_left = (8 - __builtin_args_info (3)) / 2,                     \
   __AP.__fp_regs = __AP.__gp_regs - __AP.__fp_left * 8,                   \
   __AP.__fp_regs = (char *) ((int) __AP.__fp_regs & -8))
#endif /* ! defined (__mips64) */
#else /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float) ) */
#define va_start(__AP, __LASTARG)                                          \
  (__AP = ((__gnuc_va_list) __builtin_next_arg (__LASTARG)                 \
           - (__builtin_args_info (2) >= 8 ? 0                             \
              : (8 - __builtin_args_info (2)) * __va_reg_size)))
#endif /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float) ) */
#else /* ! defined (__mips_eabi) */
#define va_start(__AP, __LASTARG) \
  (__AP = (__gnuc_va_list) __builtin_next_arg (__LASTARG))
#endif /* ! (defined (__mips_eabi) && ! defined (__mips_soft_float) && ! defined (__mi
ps_single_float)) */
#else /* ! _STDARG_H */
#define va_alist  __builtin_va_alist
#ifdef __mips64
/* This assumes that 'long long int' is always a 64 bit type.  */
```

```
#define va_dcl    long long int __builtin_va_alist; __va_ellipsis
#else
#define va_dcl    int __builtin_va_alist; __va_ellipsis
#endif
#if defined (__mips_eabi)
#if ! defined (__mips_soft_float) && ! defined (__mips_single_float)
#ifdef __mips64
#define va_start(__AP)                                          \
  (__AP.__gp_regs = ((char *) __builtin_next_arg ()             \
                     - (__builtin_args_info (2) < 8             \
                        ? (8 - __builtin_args_info (2)) * __va_reg_size \
                        : __va_reg_size)),                      \
   __AP.__fp_left = 8 - __builtin_args_info (3),                \
   __AP.__fp_regs = __AP.__gp_regs - __AP.__fp_left * __va_reg_size)
#else /* ! defined (__mips64) */
#define va_start(__AP)                                          \
  (__AP.__gp_regs = ((char *) __builtin_next_arg ()             \
                     - (__builtin_args_info (2) < 8             \
                        ? (8 - __builtin_args_info (2)) * __va_reg_size \
                        : __va_reg_size)),                      \
   __AP.__fp_left = (8 - __builtin_args_info (3)) / 2,          \
   __AP.__fp_regs = __AP.__gp_regs - __AP.__fp_left * 8,        \
   __AP.__fp_regs = (char *) ((int) __AP.__fp_regs & -8))
#endif /* ! defined (__mips64) */
#else /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float)) */
#define va_start(__AP)                                          \
  (__AP = ((__gnuc_va_list) __builtin_next_arg ()               \
           - (__builtin_args_info (2) >= 8 ? __va_reg_size      \
              : (8 - __builtin_args_info (2)) * __va_reg_size)))
#endif /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float)) */
/* Need alternate code for _MIPS_SIM_ABI64.  */
#elif defined(_MIPS_SIM) && (_MIPS_SIM == _MIPS_SIM_ABI64 || _MIPS_SIM == _MIPS_SIM_NA
BI32)
#define va_start(__AP)                                          \
  (__AP = (__gnuc_va_list) __builtin_next_arg ()                \
   + (__builtin_args_info (2) >= 8 ? -8 : 0))
#else
#define va_start(__AP)  __AP = (char *) &__builtin_va_alist
#endif
#endif /* ! _STDARG_H */

#ifndef va_end
void va_end (__gnuc_va_list);           /* Defined in libgcc.a */
#endif
#define va_end(__AP)    ((void)0)

#if defined (__mips_eabi)

#if ! defined (__mips_soft_float) && ! defined (__mips_single_float)
#ifdef __mips64
#define __va_next_addr(__AP, __type)                            \
  ((__builtin_classify_type (*(__type *) 0) == __real_type_class        \
    && __AP.__fp_left > 0)                                      \
   ? (--__AP.__fp_left, (__AP.__fp_regs += 8) - 8)              \
   : (__AP.__gp_regs += __va_reg_size) - __va_reg_size)
#else
#define __va_next_addr(__AP, __type)                            \
  ((__builtin_classify_type (*(__type *) 0) == __real_type_class        \
    && __AP.__fp_left > 0)                                      \
   ? (--__AP.__fp_left, (__AP.__fp_regs += 8) - 8)              \
   : (((__builtin_classify_type (* (__type *) 0) < __record_type_class  \
        && __alignof__ (__type) > 4)                            \
       ? __AP.__gp_regs = (char *) (((int) __AP.__gp_regs + 8 - 1) & -8) \
       : (char *) 0),                                           \
      (__builtin_classify_type (* (__type *) 0) >= __record_type_class  \
       ? (__AP.__gp_regs += __va_reg_size) - __va_reg_size      \
       : ((__AP.__gp_regs += __va_rounded_size (__type))        \
          - __va_rounded_size (__type)))))
#endif
#else /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float)) */
#ifdef __mips64
#define __va_next_addr(__AP, __type)                            \
  ((__AP += __va_reg_size) - __va_reg_size)
#else
#define __va_next_addr(__AP, __type)                            \
  (((__builtin_classify_type (* (__type *) 0) < __record_type_class     \
     && __alignof__ (__type) > 4)                               \
    ? __AP = (char *) (((int) __AP + 8 - 1) & -8)               \
    : (char *) 0),                                              \
   (__builtin_classify_type (* (__type *) 0) >= __record_type_class     \
    ? (__AP += __va_reg_size) - __va_reg_size                   \
    : ((__AP += __va_rounded_size (__type))                     \
       - __va_rounded_size (__type))))
#endif
#endif /* ! (! defined (__mips_soft_float) && ! defined (__mips_single_float)) */

#ifdef __MIPSEB__
#define va_arg(__AP, __type)                                    \
  ((__va_rounded_size (__type) <= __va_reg_size)                \
   ? *(__type *) (void *) (__va_next_addr (__AP, __type)        \
                           + __va_reg_size                      \
                           - sizeof (__type))                   \
   : (__builtin_classify_type (*(__type *) 0) >= __record_type_class    \
      ? **(__type **) (void *) (__va_next_addr (__AP, __type)   \
                                + __va_reg_size                 \
                                - sizeof (char *))              \
      : *(__type *) (void *) __va_next_addr (__AP, __type)))
#else
#define va_arg(__AP, __type)                                    \
  ((__va_rounded_size (__type) <= __va_reg_size)                \
   ? *(__type *) (void *) __va_next_addr (__AP, __type)         \
   : (__builtin_classify_type (* (__type *) 0) >= __record_type_class   \
      ? **(__type **) (void *) __va_next_addr (__AP, __type)    \
      : *(__type *) (void *) __va_next_addr (__AP, __type)))
#endif

#else /* ! defined (__mips_eabi) */

/* We cast to void * and then to TYPE * because this avoids
   a warning about increasing the alignment requirement.  */
/* The __mips64 cases are reversed from the 32 bit cases, because the standard
   32 bit calling convention left-aligns all parameters smaller than a word,
   whereas the __mips64 calling convention does not (and hence they are
   right aligned).  */
#ifdef __mips64
#ifdef __MIPSEB__
#define va_arg(__AP, __type)                                    \
  ((__type *) (void *) (__AP = (char *) ((((__PTRDIFF_TYPE__)__AP + 8 - 1) & -8) \
                                         + __va_rounded_size (__type))))[-1]
#else
#define va_arg(__AP, __type)                                    \
  ((__AP = (char *) ((((__PTRDIFF_TYPE__)__AP + 8 - 1) & -8))   \
                     + __va_rounded_size (__type))),            \
   *(__type *) (void *) (__AP - __va_rounded_size (__type)))
#endif

#else /* not __mips64 */

#ifdef __MIPSEB__
/* For big-endian machines.  */
```

```
#define va_arg(__AP, __type)                                    \
  ((__AP = (char *) ((__alignof__ (__type) > 4                  \
                      ? ((int)__AP + 8 - 1) & -8                \
                      : ((int)__AP + 4 - 1) & -4)               \
                     + __va_rounded_size (__type))),            \
   *(__type *) (void *) (__AP - __va_rounded_size (__type)))
#else
/* For little-endian machines.  */
#define va_arg(__AP, __type)                                            \
  ((__type *) (void *) (__AP = (char *) ((__alignof__(__type) > 4        \
                                          ? ((int)__AP + 8 - 1) & -8    \
                                          : ((int)__AP + 4 - 1) & -4)   \
                                         + __va_rounded_size(__type))))[-1]
#endif
#endif
#endif /* ! defined (__mips_eabi)  */

/* Copy __gnuc_va_list into another variable of this type.  */
#define __va_copy(dest, src) (dest) = (src)

#endif /* defined (_STDARG_H) || defined (_VARARGS_H) */
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * Uses the hardware timer to provide preemption, and to allow threads to sleep
 * until a certain time.
 */
public class Alarm {
    /**
     * Allocate a new Alarm. Set the machine's timer interrupt handler to this
     * alarm's callback.
     *
     * <p><b>Note</b>: Nachos will not function correctly with more than one
     * alarm.
     */
    public Alarm() {
        Machine.timer().setInterruptHandler(new Runnable() {
                public void run() { timerInterrupt(); }
            });
    }

    /**
     * The timer interrupt handler. This is called by the machine's timer
     * periodically (approximately every 500 clock ticks). Causes the current
     * thread to yield, forcing a context switch if there is another thread
     * that should be run.
     */
    public void timerInterrupt() {
        KThread.currentThread().yield();
    }

    /**
     * Put the current thread to sleep for at least <i>x</i> ticks,
     * waking it up in the timer interrupt handler. The thread must be
     * woken up (placed in the scheduler ready set) during the first timer
     * interrupt where
     *
     * <p><blockquote>
     * (current time) >= (WaitUntil called time)+(x)
     * </blockquote>
     *
     * @param   x       the minimum number of clock ticks to wait.
     *
     * @see     nachos.machine.Timer#getTime()
     */
    public void waitUntil(long x) {
        // for now, cheat just to get something working (busy waiting is bad)
        long wakeTime = Machine.timer().getTime() + x;
        while (wakeTime > Machine.timer().getTime())
            KThread.yield();
    }
}
```

```
package nachos.threads;
import nachos.ag.BoatGrader;

public class Boat
{
    static BoatGrader bg;

    public static void selfTest()
    {
        BoatGrader b = new BoatGrader();

        System.out.println("\n ***Testing Boats with only 2 children***");
        begin(0, 2, b);

//      System.out.println("\n ***Testing Boats with 2 children, 1 adult***");
//      begin(1, 2, b);

//      System.out.println("\n ***Testing Boats with 3 children, 3 adults***");
//      begin(3, 3, b);
    }

    public static void begin( int adults, int children, BoatGrader b )
    {
        // Store the externally generated autograder in a class
        // variable to be accessible by children.
        bg = b;

        // Instantiate global variables here

        // Create threads here. See section 3.4 of the Nachos for Java
        // Walkthrough linked from the projects page.

        Runnable r = new Runnable() {
            public void run() {
                SampleItinerary();
            }
        };
        KThread t = new KThread(r);
        t.setName("Sample Boat Thread");
        t.fork();

    }

    static void AdultItinerary()
    {
        /* This is where you should put your solutions. Make calls
           to the BoatGrader to show that it is synchronized. For
           example:
               bg.AdultRowToMolokai();
           indicates that an adult has rowed the boat across to Molokai
        */
    }

    static void ChildItinerary()
    {
    }

    static void SampleItinerary()
    {
        // Please note that this isn't a valid solution (you can't fit
        // all of them on the boat). Please also note that you may not
        // have a single thread calculate a solution and then just play
        // it back at the autograder -- you will be caught.
        System.out.println("\n ***Everyone piles on the boat and goes to Molokai***");
        bg.AdultRowToMolokai();
        bg.ChildRideToMolokai();
        bg.AdultRideToMolokai();
        bg.ChildRideToMolokai();
    }

}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A <i>communicator</i> allows threads to synchronously exchange 32-bit
 * messages. Multiple threads can be waiting to <i>speak</i>,
 * and multiple threads can be waiting to <i>listen</i>. But there should never
 * be a time when both a speaker and a listener are waiting, because the two
 * threads can be paired off at this point.
 */
public class Communicator {
    /**
     * Allocate a new communicator.
     */
    public Communicator() {
    }

    /**
     * Wait for a thread to listen through this communicator, and then transfer
     * <i>word</i> to the listener.
     *
     * <p>
     * Does not return until this thread is paired up with a listening thread.
     * Exactly one listener should receive <i>word</i>.
     *
     * @param   word    the integer to transfer.
     */
    public void speak(int word) {
    }

    /**
     * Wait for a thread to speak through this communicator, and then return
     * the <i>word</i> that thread passed to <tt>speak()</tt>.
     *
     * @return  the integer transferred.
     */
    public int listen() {
        return 0;
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

import java.util.LinkedList;

/**
 * An implementation of condition variables built upon semaphores.
 *
 * <p>
 * A condition variable is a synchronization primitive that does not have
 * a value (unlike a semaphore or a lock), but threads may still be queued.
 *
 * <p><ul>
 *
 * <li><tt>sleep()</tt>: atomically release the lock and relinkquish the CPU
 * until woken; then reacquire the lock.
 *
 * <li><tt>wake()</tt>: wake up a single thread sleeping in this condition
 * variable, if possible.
 *
 * <li><tt>wakeAll()</tt>: wake up all threads sleeping inn this condition
 * variable.
 *
 * </ul>
 *
 * <p>
 * Every condition variable is associated with some lock. Multiple condition
 * variables may be associated with the same lock. All three condition variable
 * operations can only be used while holding the associated lock.
 *
 * <p>
 * In Nachos, condition variables are summed to obey <i>Mesa-style</i>
 * semantics. When a <tt>wake()</tt> or <tt>wakeAll()</tt> wakes up another
 * thread, the woken thread is simply put on the ready list, and it is the
 * responsibility of the woken thread to reacquire the lock (this reacquire is
 * taken core of in <tt>sleep()</tt>).
 *
 * <p>
 * By contrast, some implementations of condition variables obey
 * <i>Hoare-style</i> semantics, where the thread that calls <tt>wake()</tt>
 * gives up the lock and the CPU to the woken thread, which runs immediately
 * and gives the lock and CPU back to the waker when the woken thread exits the
 * critical section.
 *
 * <p>
 * The consequence of using Mesa-style semantics is that some other thread
 * can acquire the lock and change data structures, before the woken thread
 * gets a chance to run. The advance to Mesa-style semantics is that it is a
 * lot easier to implement.
 */
public class Condition {
    /**
     * Allocate a new condition variable.
     *
     * @param   conditionLock   the lock associated with this condition
     *                          variable. The current thread must hold this
     *                          lock whenever it uses <tt>sleep()</tt>,
     *                          <tt>wake()</tt>, or <tt>wakeAll()</tt>.
     */
    public Condition(Lock conditionLock) {
        this.conditionLock = conditionLock;

        waitQueue = new LinkedList<Semaphore>();
    }

    /**
     * Atomically release the associated lock and go to sleep on this condition
     * variable until another thread wakes it using <tt>wake()</tt>. The
     * current thread must hold the associated lock. The thread will
     * automatically reacquire the lock before <tt>sleep()</tt> returns.
     *
     * <p>
     * This implementation uses semaphores to implement this, by allocating a
     * semaphore for each waiting thread. The waker will <tt>V()</tt> this
     * semaphore, so thre is no chance the sleeper will miss the wake-up, even
     * though the lock is released before caling <tt>P()</tt>.
     */
    public void sleep() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        Semaphore waiter = new Semaphore(0);
        waitQueue.add(waiter);

        conditionLock.release();
        waiter.P();
        conditionLock.acquire();
    }

    /**
     * Wake up at most one thread sleeping on this condition variable. The
     * current thread must hold the associated lock.
     */
    public void wake() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        if (!waitQueue.isEmpty())
            ((Semaphore) waitQueue.removeFirst()).V();
    }

    /**
     * Wake up all threads sleeping on this condition variable. The current
     * thread must hold the associated lock.
     */
    public void wakeAll() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        while (!waitQueue.isEmpty())
            wake();
    }

    private Lock conditionLock;
    private LinkedList<Semaphore> waitQueue;
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * An implementation of condition variables that disables interrupt()s for
 * synchronization.
 *
 * <p>
 * You must implement this.
 *
 * @see nachos.threads.Condition
 */
public class Condition2 {
    /**
     * Allocate a new condition variable.
     *
     * @param   conditionLock   the lock associated with this condition
     *                          variable. The current thread must hold this
     *                          lock whenever it uses <tt>sleep()</tt>,
     *                          <tt>wake()</tt>, or <tt>wakeAll()</tt>.
     */
    public Condition2(Lock conditionLock) {
        this.conditionLock = conditionLock;
    }

    /**
     * Atomically release the associated lock and go to sleep on this condition
     * variable until another thread wakes it using <tt>wake()</tt>. The
     * current thread must hold the associated lock. The thread will
     * automatically reacquire the lock before <tt>sleep()</tt> returns.
     */
    public void sleep() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());

        conditionLock.release();

        conditionLock.acquire();
    }

    /**
     * Wake up at most one thread sleeping on this condition variable. The
     * current thread must hold the associated lock.
     */
    public void wake() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());
    }

    /**
     * Wake up all threads sleeping on this condition variable. The current
     * thread must hold the associated lock.
     */
    public void wakeAll() {
        Lib.assertTrue(conditionLock.isHeldByCurrentThread());
    }

    private Lock conditionLock;
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A controller for all the elevators in an elevator bank. The controller
 * accesses the elevator bank through an instance of <tt>ElevatorControls</tt>.
 */
public class ElevatorController implements ElevatorControllerInterface {
    /**
     * Allocate a new elevator controller.
     */
    public ElevatorController() {
    }

    /**
     * Initialize this elevator controller. The controller will access the
     * elevator bank through <i>controls</i>. This constructor should return
     * immediately after this controller is initialized, but not until the
     * interupt handler is set. The controller will start receiving events
     * after this method returns, but potentially before <tt>run()</tt> is
     * called.
     *
     * @param   controls        the controller's interface to the elevator
     *                          bank. The controler must not attempt to access
     *                          the elevator bank in <i>any</i> other way.
     */
    public void initialize(ElevatorControls controls) {
    }

    /**
     * Cause the controller to use the provided controls to receive and process
     * requests from riders. This method should not return, but instead should
     * call <tt>controls.finish()</tt> when the controller is finished.
     */
    public void run() {
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A KThread is a thread that can be used to execute Nachos kernel code. Nachos
 * allows multiple threads to run concurrently.
 *
 * To create a new thread of execution, first declare a class that implements
 * the <tt>Runnable</tt> interface. That class then implements the <tt>run</tt>
 * method. An instance of the class can then be allocated, passed as an
 * argument when creating <tt>KThread</tt>, and forked. For example, a thread
 * that computes pi could be written as follows:
 *
 * <p><blockquote><pre>
 * class PiRun implements Runnable {
 *     public void run() {
 *         // compute pi
 *         ...
 *     }
 * }
 * </pre></blockquote>
 * <p>The following code would then create a thread and start it running:
 *
 * <p><blockquote><pre>
 * PiRun p = new PiRun();
 * new KThread(p).fork();
 * </pre></blockquote>
 */
public class KThread {
    /**
     * Get the current thread.
     *
     * @return  the current thread.
     */
    public static KThread currentThread() {
        Lib.assertTrue(currentThread != null);
        return currentThread;
    }

    /**
     * Allocate a new <tt>KThread</tt>. If this is the first <tt>KThread</tt>,
     * create an idle thread as well.
     */
    public KThread() {
        if (currentThread != null) {
            tcb = new TCB();
        }
        else {
            readyQueue = ThreadedKernel.scheduler.newThreadQueue(false);
            readyQueue.acquire(this);

            currentThread = this;
            tcb = TCB.currentTCB();
            name = "main";
            restoreState();

            createIdleThread();
        }
    }

    /**
     * Allocate a new KThread.
     *
     * @param   target  the object whose <tt>run</tt> method is called.
     */
    public KThread(Runnable target) {
        this();
        this.target = target;
    }

    /**
     * Set the target of this thread.
     *
     * @param   target  the object whose <tt>run</tt> method is called.
     * @return  this thread.
     */
    public KThread setTarget(Runnable target) {
        Lib.assertTrue(status == statusNew);

        this.target = target;
        return this;
    }

    /**
     * Set the name of this thread. This name is used for debugging purposes
     * only.
     *
     * @param   name    the name to give to this thread.
     * @return  this thread.
     */
    public KThread setName(String name) {
        this.name = name;
        return this;
    }

    /**
     * Get the name of this thread. This name is used for debugging purposes
     * only.
     *
     * @return  the name given to this thread.
     */
    public String getName() {
        return name;
    }

    /**
     * Get the full name of this thread. This includes its name along with its
     * numerical ID. This name is used for debugging purposes only.
     *
     * @return  the full name given to this thread.
     */
    public String toString() {
        return (name + " (#" + id + ")");
    }

    /**
     * Deterministically and consistently compare this thread to another
     * thread.
     */
    public int compareTo(Object o) {
        KThread thread = (KThread) o;

        if (id < thread.id)
            return -1;
        else if (id > thread.id)
            return 1;
        else
            return 0;
    }
```

```
/**
 * Causes this thread to begin execution. The result is that two threads
 * are running concurrently: the current thread (which returns from the
 * call to the <tt>fork</tt> method) and the other thread (which executes
 * its target's <tt>run</tt> method).
 */
public void fork() {
    Lib.assertTrue(status == statusNew);
    Lib.assertTrue(target != null);

    Lib.debug(dbgThread,
              "Forking thread: " + toString() + " Runnable: " + target);

    boolean intStatus = Machine.interrupt().disable();

    tcb.start(new Runnable() {
            public void run() {
                runThread();
            }
        });

    ready();

    Machine.interrupt().restore(intStatus);
}

private void runThread() {
    begin();
    target.run();
    finish();
}

private void begin() {
    Lib.debug(dbgThread, "Beginning thread: " + toString());

    Lib.assertTrue(this == currentThread);

    restoreState();

    Machine.interrupt().enable();
}

/**
 * Finish the current thread and schedule it to be destroyed when it is
 * safe to do so. This method is automatically called when a thread's
 * <tt>run</tt> method returns, but it may also be called directly.
 *
 * The current thread cannot be immediately destroyed because its stack and
 * other execution state are still in use. Instead, this thread will be
 * destroyed automatically by the next thread to run, when it is safe to
 * delete this thread.
 */
public static void finish() {
    Lib.debug(dbgThread, "Finishing thread: " + currentThread.toString());

    Machine.interrupt().disable();

    Machine.autoGrader().finishingCurrentThread();

    Lib.assertTrue(toBeDestroyed == null);
    toBeDestroyed = currentThread;


    currentThread.status = statusFinished;
    sleep();
}

/**
 * Relinquish the CPU if any other thread is ready to run. If so, put the
 * current thread on the ready queue, so that it will eventually be
 * rescheuled.
 *
 * <p>
 * Returns immediately if no other thread is ready to run. Otherwise
 * returns when the current thread is chosen to run again by
 * <tt>readyQueue.nextThread()</tt>.
 *
 * <p>
 * Interrupts are disabled, so that the current thread can atomically add
 * itself to the ready queue and switch to the next thread. On return,
 * restores interrupts to the previous state, in case <tt>yield()</tt> was
 * called with interrupts disabled.
 */
public static void yield() {
    Lib.debug(dbgThread, "Yielding thread: " + currentThread.toString());

    Lib.assertTrue(currentThread.status == statusRunning);

    boolean intStatus = Machine.interrupt().disable();

    currentThread.ready();

    runNextThread();

    Machine.interrupt().restore(intStatus);
}

/**
 * Relinquish the CPU, because the current thread has either finished or it
 * is blocked. This thread must be the current thread.
 *
 * <p>
 * If the current thread is blocked (on a synchronization primitive, i.e.
 * a <tt>Semaphore</tt>, <tt>Lock</tt>, or <tt>Condition</tt>), eventually
 * some thread will wake this thread up, putting it back on the ready queue
 * so that it can be rescheduled. Otherwise, <tt>finish()</tt> should have
 * scheduled this thread to be destroyed by the next thread to run.
 */
public static void sleep() {
    Lib.debug(dbgThread, "Sleeping thread: " + currentThread.toString());

    Lib.assertTrue(Machine.interrupt().disabled());

    if (currentThread.status != statusFinished)
        currentThread.status = statusBlocked;

    runNextThread();
}

/**
 * Moves this thread to the ready state and adds this to the scheduler's
 * ready queue.
 */
public void ready() {
    Lib.debug(dbgThread, "Ready thread: " + toString());

    Lib.assertTrue(Machine.interrupt().disabled());
    Lib.assertTrue(status != statusReady);
```

```
    status = statusReady;
    if (this != idleThread)
        readyQueue.waitForAccess(this);

    Machine.autoGrader().readyThread(this);
}

/**
 * Waits for this thread to finish. If this thread is already finished,
 * return immediately. This method must only be called once; the second
 * call is not guaranteed to return. This thread must not be the current
 * thread.
 */
public void join() {
    Lib.debug(dbgThread, "Joining to thread: " + toString());

    Lib.assertTrue(this != currentThread);

}

/**
 * Create the idle thread. Whenever there are no threads ready to be run,
 * and <tt>runNextThread()</tt> is called, it will run the idle thread. The
 * idle thread must never block, and it will only be allowed to run when
 * all other threads are blocked.
 *
 * <p>
 * Note that <tt>ready()</tt> never adds the idle thread to the ready set.
 */
private static void createIdleThread() {
    Lib.assertTrue(idleThread == null);

    idleThread = new KThread(new Runnable() {
        public void run() { while (true) yield(); }
    });
    idleThread.setName("idle");

    Machine.autoGrader().setIdleThread(idleThread);

    idleThread.fork();
}

/**
 * Determine the next thread to run, then dispatch the CPU to the thread
 * using <tt>run()</tt>.
 */
private static void runNextThread() {
    KThread nextThread = readyQueue.nextThread();
    if (nextThread == null)
        nextThread = idleThread;

    nextThread.run();
}

/**
 * Dispatch the CPU to this thread. Save the state of the current thread,
 * switch to the new thread by calling <tt>TCB.contextSwitch()</tt>, and
 * load the state of the new thread. The new thread becomes the current
 * thread.
 *
 * <p>
 * If the new thread and the old thread are the same, this method must
 * still call <tt>saveState()</tt>, <tt>contextSwitch()</tt>, and
 * <tt>restoreState()</tt>.
 *
 * <p>
 * The state of the previously running thread must already have been
 * changed from running to blocked or ready (depending on whether the
 * thread is sleeping or yielding).
 *
 * @param   finishing       <tt>true</tt> if the current thread is
 *                          finished, and should be destroyed by the new
 *                          thread.
 */
private void run() {
    Lib.assertTrue(Machine.interrupt().disabled());

    Machine.yield();

    currentThread.saveState();

    Lib.debug(dbgThread, "Switching from: " + currentThread.toString()
              + " to: " + toString());

    currentThread = this;

    tcb.contextSwitch();

    currentThread.restoreState();
}

/**
 * Prepare this thread to be run. Set <tt>status</tt> to
 * <tt>statusRunning</tt> and check <tt>toBeDestroyed</tt>.
 */
protected void restoreState() {
    Lib.debug(dbgThread, "Running thread: " + currentThread.toString());

    Lib.assertTrue(Machine.interrupt().disabled());
    Lib.assertTrue(this == currentThread);
    Lib.assertTrue(tcb == TCB.currentTCB());

    Machine.autoGrader().runningThread(this);

    status = statusRunning;

    if (toBeDestroyed != null) {
        toBeDestroyed.tcb.destroy();
        toBeDestroyed.tcb = null;
        toBeDestroyed = null;
    }
}

/**
 * Prepare this thread to give up the processor. Kernel threads do not
 * need to do anything here.
 */
protected void saveState() {
    Lib.assertTrue(Machine.interrupt().disabled());
    Lib.assertTrue(this == currentThread);
}

private static class PingTest implements Runnable {
    PingTest(int which) {
        this.which = which;
    }

    public void run() {
        for (int i=0; i<5; i++) {
```

```
                System.out.println("*** thread " + which + " looped "
                                   + i + " times");
                currentThread.yield();
            }
        }

        private int which;
    }

    /**
     * Tests whether this module is working.
     */
    public static void selfTest() {
        Lib.debug(dbgThread, "Enter KThread.selfTest");

        new KThread(new PingTest(1)).setName("forked thread").fork();
        new PingTest(0).run();
    }

    private static final char dbgThread = 't';

    /**
     * Additional state used by schedulers.
     *
     * @see     nachos.threads.PriorityScheduler.ThreadState
     */
    public Object schedulingState = null;

    private static final int statusNew = 0;
    private static final int statusReady = 1;
    private static final int statusRunning = 2;
    private static final int statusBlocked = 3;
    private static final int statusFinished = 4;

    /**
     * The status of this thread. A thread can either be new (not yet forked),
     * ready (on the ready queue but not running), running, or blocked (not
     * on the ready queue and not running).
     */
    private int status = statusNew;
    private String name = "(unnamed thread)";
    private Runnable target;
    private TCB tcb;

    /**
     * Unique identifer for this thread. Used to deterministically compare
     * threads.
     */
    private int id = numCreated++;
    /** Number of times the KThread constructor was called. */
    private static int numCreated = 0;

    private static ThreadQueue readyQueue = null;
    private static KThread currentThread = null;
    private static KThread toBeDestroyed = null;
    private static KThread idleThread = null;
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A <tt>Lock</tt> is a synchronization primitive that has two states,
 * <i>busy</i> and <i>free</i>. There are only two operations allowed on a
 * lock:
 *
 * <ul>
 * <li><tt>acquire()</tt>: atomically wait until the lock is <i>free</i> and
 * then set it to <i>busy</i>.
 * <li><tt>release()</tt>: set the lock to be <i>free</i>, waking up one
 * waiting thread if possible.
 * </ul>
 *
 * <p>
 * Also, only the thread that acquired a lock may release it. As with
 * semaphores, the API does not allow you to read the lock state (because the
 * value could change immediately after you read it).
 */
public class Lock {
    /**
     * Allocate a new lock. The lock will initially be <i>free</i>.
     */
    public Lock() {
    }

    /**
     * Atomically acquire this lock. The current thread must not already hold
     * this lock.
     */
    public void acquire() {
        Lib.assertTrue(!isHeldByCurrentThread());

        boolean intStatus = Machine.interrupt().disable();
        KThread thread = KThread.currentThread();

        if (lockHolder != null) {
            waitQueue.waitForAccess(thread);
            KThread.sleep();
        }
        else {
            waitQueue.acquire(thread);
            lockHolder = thread;
        }

        Lib.assertTrue(lockHolder == thread);

        Machine.interrupt().restore(intStatus);
    }

    /**
     * Atomically release this lock, allowing other threads to acquire it.
     */
    public void release() {
        Lib.assertTrue(isHeldByCurrentThread());

        boolean intStatus = Machine.interrupt().disable();

        if ((lockHolder = waitQueue.nextThread()) != null)
            lockHolder.ready();

        Machine.interrupt().restore(intStatus);
    }

    /**
     * Test if the current thread holds this lock.
     *
     * @return  true if the current thread holds this lock.
     */
    public boolean isHeldByCurrentThread() {
        return (lockHolder == KThread.currentThread());
    }

    private KThread lockHolder = null;
    private ThreadQueue waitQueue =
        ThreadedKernel.scheduler.newThreadQueue(true);
}
```

```
package nachos.threads;

import nachos.machine.*;

import java.util.TreeSet;
import java.util.HashSet;
import java.util.Iterator;

/**
 * A scheduler that chooses threads using a lottery.
 *
 * <p>
 * A lottery scheduler associates a number of tickets with each thread. When a
 * thread needs to be dequeued, a random lottery is held, among all the tickets
 * of all the threads waiting to be dequeued. The thread that holds the winning
 * ticket is chosen.
 *
 * <p>
 * Note that a lottery scheduler must be able to handle a lot of tickets
 * (sometimes billions), so it is not acceptable to maintain state for every
 * ticket.
 *
 * <p>
 * A lottery scheduler must partially solve the priority inversion problem; in
 * particular, tickets must be transferred through locks, and through joins.
 * Unlike a priority scheduler, these tickets add (as opposed to just taking
 * the maximum).
 */
public class LotteryScheduler extends PriorityScheduler {
    /**
     * Allocate a new lottery scheduler.
     */
    public LotteryScheduler() {
    }

    /**
     * Allocate a new lottery thread queue.
     *
     * @param   transferPriority        <tt>true</tt> if this queue should
     *                                  transfer tickets from waiting threads
     *                                  to the owning thread.
     * @return  a new lottery thread queue.
     */
    public ThreadQueue newThreadQueue(boolean transferPriority) {
        // implement me
        return null;
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

import java.util.TreeSet;
import java.util.HashSet;
import java.util.Iterator;

/**
 * A scheduler that chooses threads based on their priorities.
 *
 * <p>
 * A priority scheduler associates a priority with each thread. The next thread
 * to be dequeued is always a thread with priority no less than any other
 * waiting thread's priority. Like a round-robin scheduler, the thread that is
 * dequeued is, among all the threads of the same (highest) priority, the
 * thread that has been waiting longest.
 *
 * <p>
 * Essentially, a priority scheduler gives access in a round-robin fassion to
 * all the highest-priority threads, and ignores all other threads. This has
 * the potential to
 * starve a thread if there's always a thread waiting with higher priority.
 *
 * <p>
 * A priority scheduler must partially solve the priority inversion problem; in
 * particular, priority must be donated through locks, and through joins.
 */
public class PriorityScheduler extends Scheduler {
    /**
     * Allocate a new priority scheduler.
     */
    public PriorityScheduler() {
    }

    /**
     * Allocate a new priority thread queue.
     *
     * @param   transferPriority        <tt>true</tt> if this queue should
     *                                  transfer priority from waiting threads
     *                                  to the owning thread.
     * @return  a new priority thread queue.
     */
    public ThreadQueue newThreadQueue(boolean transferPriority) {
        return new PriorityQueue(transferPriority);
    }

    public int getPriority(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());

        return getThreadState(thread).getPriority();
    }

    public int getEffectivePriority(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());

        return getThreadState(thread).getEffectivePriority();
    }

    public void setPriority(KThread thread, int priority) {
        Lib.assertTrue(Machine.interrupt().disabled());

        Lib.assertTrue(priority >= priorityMinimum &&
                   priority <= priorityMaximum);

        getThreadState(thread).setPriority(priority);
    }

    public boolean increasePriority() {
        boolean intStatus = Machine.interrupt().disable();

        KThread thread = KThread.currentThread();

        int priority = getPriority(thread);
        if (priority == priorityMaximum)
            return false;

        setPriority(thread, priority+1);

        Machine.interrupt().restore(intStatus);
        return true;
    }

    public boolean decreasePriority() {
        boolean intStatus = Machine.interrupt().disable();

        KThread thread = KThread.currentThread();

        int priority = getPriority(thread);
        if (priority == priorityMinimum)
            return false;

        setPriority(thread, priority-1);

        Machine.interrupt().restore(intStatus);
        return true;
    }

    /**
     * The default priority for a new thread. Do not change this value.
     */
    public static final int priorityDefault = 1;
    /**
     * The minimum priority that a thread can have. Do not change this value.
     */
    public static final int priorityMinimum = 0;
    /**
     * The maximum priority that a thread can have. Do not change this value.
     */
    public static final int priorityMaximum = 7;

    /**
     * Return the scheduling state of the specified thread.
     *
     * @param   thread  the thread whose scheduling state to return.
     * @return  the scheduling state of the specified thread.
     */
    protected ThreadState getThreadState(KThread thread) {
        if (thread.schedulingState == null)
            thread.schedulingState = new ThreadState(thread);

        return (ThreadState) thread.schedulingState;
    }

    /**
     * A <tt>ThreadQueue</tt> that sorts threads by priority.
     */
    protected class PriorityQueue extends ThreadQueue {
        PriorityQueue(boolean transferPriority) {
            this.transferPriority = transferPriority;
```

```
    }

    public void waitForAccess(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());
        getThreadState(thread).waitForAccess(this);
    }

    public void acquire(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());
        getThreadState(thread).acquire(this);
    }

    public KThread nextThread() {
        Lib.assertTrue(Machine.interrupt().disabled());
        // implement me
        return null;
    }

    /**
     * Return the next thread that <tt>nextThread()</tt> would return,
     * without modifying the state of this queue.
     *
     * @return      the next thread that <tt>nextThread()</tt> would
     *              return.
     */
    protected ThreadState pickNextThread() {
        // implement me
        return null;
    }

    public void print() {
        Lib.assertTrue(Machine.interrupt().disabled());
        // implement me (if you want)
    }

    /**
     * <tt>true</tt> if this queue should transfer priority from waiting
     * threads to the owning thread.
     */
    public boolean transferPriority;
}

/**
 * The scheduling state of a thread. This should include the thread's
 * priority, its effective priority, any objects it owns, and the queue
 * it's waiting for, if any.
 *
 * @see     nachos.threads.KThread#schedulingState
 */
protected class ThreadState {
    /**
     * Allocate a new <tt>ThreadState</tt> object and associate it with the
     * specified thread.
     *
     * @param       thread  the thread this state belongs to.
     */
    public ThreadState(KThread thread) {
        this.thread = thread;

        setPriority(priorityDefault);
    }

    /**
     * Return the priority of the associated thread.
     *
     * @return      the priority of the associated thread.
     */
    public int getPriority() {
        return priority;
    }

    /**
     * Return the effective priority of the associated thread.
     *
     * @return      the effective priority of the associated thread.
     */
    public int getEffectivePriority() {
        // implement me
        return priority;
    }

    /**
     * Set the priority of the associated thread to the specified value.
     *
     * @param       priority        the new priority.
     */
    public void setPriority(int priority) {
        if (this.priority == priority)
            return;

        this.priority = priority;

        // implement me
    }

    /**
     * Called when <tt>waitForAccess(thread)</tt> (where <tt>thread</tt> is
     * the associated thread) is invoked on the specified priority queue.
     * The associated thread is therefore waiting for access to the
     * resource guarded by <tt>waitQueue</tt>. This method is only called
     * if the associated thread cannot immediately obtain access.
     *
     * @param       waitQueue       the queue that the associated thread is
     *                              now waiting on.
     *
     * @see nachos.threads.ThreadQueue#waitForAccess
     */
    public void waitForAccess(PriorityQueue waitQueue) {
        // implement me
    }

    /**
     * Called when the associated thread has acquired access to whatever is
     * guarded by <tt>waitQueue</tt>. This can occur either as a result of
     * <tt>acquire(thread)</tt> being invoked on <tt>waitQueue</tt> (where
     * <tt>thread</tt> is the associated thread), or as a result of
     * <tt>nextThread()</tt> being invoked on <tt>waitQueue</tt>.
     *
     * @see nachos.threads.ThreadQueue#acquire
     * @see nachos.threads.ThreadQueue#nextThread
     */
    public void acquire(PriorityQueue waitQueue) {
        // implement me
    }

    /** The thread with which this object is associated. */
    protected KThread thread;
    /** The priority of the associated thread. */
    protected int priority;
}
```

```
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A single rider. Each rider accesses the elevator bank through an
 * instance of <tt>RiderControls</tt>.
 */
public class Rider implements RiderInterface {
    /**
     * Allocate a new rider.
     */
    public Rider() {
    }

    /**
     * Initialize this rider. The rider will access the elevator bank through
     * <i>controls</i>, and the rider will make stops at different floors as
     * specified in <i>stops</i>. This method should return immediately after
     * this rider is initialized, but not until the interrupt handler is
     * set. The rider will start receiving events after this method returns,
     * potentially before <tt>run()</tt> is called.
     *
     * @param   controls        the rider's interface to the elevator bank. The
     *                          rider must not attempt to access the elevator
     *                          bank in <i>any</i> other way.
     * @param   stops           an array of stops the rider should make; see
     *                          below.
     */
    public void initialize(RiderControls controls, int[] stops) {
    }

    /**
     * Cause the rider to use the provided controls to make the stops specified
     * in the constructor. The rider should stop at each of the floors in
     * <i>stops</i>, an array of floor numbers. The rider should <i>only</i>
     * make the specified stops.
     *
     * <p>
     * For example, suppose the rider uses <i>controls</i> to determine that
     * it is initially on floor 1, and suppose the stops array contains two
     * elements: { 0, 2 }. Then the rider should get on an elevator, get off
     * on floor 0, get on an elevator, and get off on floor 2, pushing buttons
     * as necessary.
     *
     * <p>
     * This method should not return, but instead should call
     * <tt>controls.finish()</tt> when the rider is finished.
     */
    public void run() {
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

import java.util.LinkedList;
import java.util.Iterator;

/**
 * A round-robin scheduler tracks waiting threads in FIFO queues, implemented
 * with linked lists. When a thread begins waiting for access, it is appended
 * to the end of a list. The next thread to receive access is always the first
 * thread in the list. This causes access to be given on a first-come
 * first-serve basis.
 */
public class RoundRobinScheduler extends Scheduler {
    /**
     * Allocate a new round-robin scheduler.
     */
    public RoundRobinScheduler() {
    }

    /**
     * Allocate a new FIFO thread queue.
     *
     * @param   transferPriority        ignored. Round robin schedulers have
     *                                  no priority.
     * @return  a new FIFO thread queue.
     */
    public ThreadQueue newThreadQueue(boolean transferPriority) {
        return new FifoQueue();
    }

    private class FifoQueue extends ThreadQueue {
        /**
         * Add a thread to the end of the wait queue.
         *
         * @param       thread  the thread to append to the queue.
         */
        public void waitForAccess(KThread thread) {
            Lib.assertTrue(Machine.interrupt().disabled());

            waitQueue.add(thread);
        }

        /**
         * Remove a thread from the beginning of the queue.
         *
         * @return      the first thread on the queue, or <tt>null</tt> if the
         *              queue is
         *              empty.
         */
        public KThread nextThread() {
            Lib.assertTrue(Machine.interrupt().disabled());

            if (waitQueue.isEmpty())
                return null;

            return (KThread) waitQueue.removeFirst();
        }

        /**
         * The specified thread has received exclusive access, without using
         * <tt>waitForAccess()</tt> or <tt>nextThread()</tt>. Assert that no
         * threads are waiting for access.
         */
        public void acquire(KThread thread) {
            Lib.assertTrue(Machine.interrupt().disabled());

            Lib.assertTrue(waitQueue.isEmpty());
        }

        /**
         * Print out the contents of the queue.
         */
        public void print() {
            Lib.assertTrue(Machine.interrupt().disabled());

            for (Iterator i=waitQueue.iterator(); i.hasNext(); )
                System.out.print((KThread) i.next() + " ");
        }

        private LinkedList<KThread> waitQueue = new LinkedList<KThread>();
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * Coordinates a group of thread queues of the same kind.
 *
 * @see nachos.threads.ThreadQueue
 */
public abstract class Scheduler {
    /**
     * Allocate a new scheduler.
     */
    public Scheduler() {
    }

    /**
     * Allocate a new thread queue. If <i>transferPriority</i> is
     * <tt>true</tt>, then threads waiting on the new queue will transfer their
     * "priority" to the thread that has access to whatever is being guarded by
     * the queue. This is the mechanism used to partially solve priority
     * inversion.
     *
     * <p>
     * If there is no definite thread that can be said to have "access" (as in
     * the case of semaphores and condition variables), this parameter should
     * be <tt>false</tt>, indicating that no priority should be transferred.
     *
     * <p>
     * The processor is a special case. There is clearly no purpose to donating
     * priority to a thread that already has the processor. When the processor
     * wait queue is created, this parameter should be <tt>false</tt>.
     *
     * <p>
     * Otherwise, it is beneficial to donate priority. For example, a lock has
     * a definite owner (the thread that holds the lock), and a lock is always
     * released by the same thread that acquired it, so it is possible to help
     * a high priority thread waiting for a lock by donating its priority to
     * the thread holding the lock. Therefore, a queue for a lock should be
     * created with this parameter set to <tt>true</tt>.
     *
     * <p>
     * Similarly, when a thread is asleep in <tt>join()</tt> waiting for the
     * target thread to finish, the sleeping thread should donate its priority
     * to the target thread. Therefore, a join queue should be created with
     * this parameter set to <tt>true</tt>.
     *
     * @param   transferPriority        <tt>true</tt> if the thread that has
     *                                  access should receive priority from the
     *                                  threads that are waiting on this queue.
     * @return  a new thread queue.
     */
    public abstract ThreadQueue newThreadQueue(boolean transferPriority);

    /**
     * Get the priority of the specified thread. Must be called with
     * interrupts disabled.
     *
     * @param   thread  the thread to get the priority of.
     * @return  the thread's priority.
     */
    public int getPriority(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());
        return 0;
    }

    /**
     * Get the priority of the current thread. Equivalent to
     * <tt>getPriority(KThread.currentThread())</tt>.
     *
     * @return  the current thread's priority.
     */
    public int getPriority() {
        return getPriority(KThread.currentThread());
    }

    /**
     * Get the effective priority of the specified thread. Must be called with
     * interrupts disabled.
     *
     * <p>
     * The effective priority of a thread is the priority of a thread after
     * taking into account priority donations.
     *
     * <p>
     * For a priority scheduler, this is the maximum of the thread's priority
     * and the priorities of all other threads waiting for the thread through a
     * lock or a join.
     *
     * <p>
     * For a lottery scheduler, this is the sum of the thread's tickets and the
     * tickets of all other threads waiting for the thread through a lock or a
     * join.
     *
     * @param   thread  the thread to get the effective priority of.
     * @return  the thread's effective priority.
     */
    public int getEffectivePriority(KThread thread) {
        Lib.assertTrue(Machine.interrupt().disabled());
        return 0;
    }

    /**
     * Get the effective priority of the current thread. Equivalent to
     * <tt>getEffectivePriority(KThread.currentThread())</tt>.
     *
     * @return  the current thread's priority.
     */
    public int getEffectivePriority() {
        return getEffectivePriority(KThread.currentThread());
    }

    /**
     * Set the priority of the specified thread. Must be called with interrupts
     * disabled.
     *
     * @param   thread  the thread to set the priority of.
     * @param   priority        the new priority.
     */
    public void setPriority(KThread thread, int priority) {
        Lib.assertTrue(Machine.interrupt().disabled());
    }

    /**
     * Set the priority of the current thread. Equivalent to
     * <tt>setPriority(KThread.currentThread(), priority)</tt>.
     *
     * @param   priority        the new priority.
     */
    public void setPriority(int priority) {
```

```
        setPriority(KThread.currentThread(), priority);
    }

    /**
     * If possible, raise the priority of the current thread in some
     * scheduler-dependent way.
     *
     * @return  <tt>true</tt> if the scheduler was able to increase the current
     *          thread's
     *          priority.
     */
    public boolean increasePriority() {
        return false;
    }

    /**
     * If possible, lower the priority of the current thread user in some
     * scheduler-dependent way, preferably by the same amount as would a call
     * to <tt>increasePriority()</tt>.
     *
     * @return  <tt>true</tt> if the scheduler was able to decrease the current
     *          thread's priority.
     */
    public boolean decreasePriority() {
        return false;
    }
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A <tt>Semaphore</tt> is a synchronization primitive with an unsigned value.
 * A semaphore has only two operations:
 *
 * <ul>
 * <li><tt>P()</tt>: waits until the semaphore's value is greater than zero,
 * then decrements it.
 * <li><tt>V()</tt>: increments the semaphore's value, and wakes up one thread
 * waiting in <tt>P()</tt> if possible.
 * </ul>
 *
 * <p>
 * Note that this API does not allow a thread to read the value of the
 * semaphore directly. Even if you did read the value, the only thing you would
 * know is what the value used to be. You don't know what the value is now,
 * because by the time you get the value, a context switch might have occurred,
 * and some other thread might have called <tt>P()</tt> or <tt>V()</tt>, so the
 * true value might now be different.
 */
public class Semaphore {
    /**
     * Allocate a new semaphore.
     *
     * @param   initialValue    the initial value of this semaphore.
     */
    public Semaphore(int initialValue) {
        value = initialValue;
    }

    /**
     * Atomically wait for this semaphore to become non-zero and decrement it.
     */
    public void P() {
        boolean intStatus = Machine.interrupt().disable();

        if (value == 0) {
            waitQueue.waitForAccess(KThread.currentThread());
            KThread.sleep();
        }
        else {
            value--;
        }

        Machine.interrupt().restore(intStatus);
    }

    /**
     * Atomically increment this semaphore and wake up at most one other thread
     * sleeping on this semaphore.
     */
    public void V() {
        boolean intStatus = Machine.interrupt().disable();

        KThread thread = waitQueue.nextThread();
        if (thread != null) {
            thread.ready();
        }
        else {
            value++;
        }
        Machine.interrupt().restore(intStatus);
    }

    private static class PingTest implements Runnable {
        PingTest(Semaphore ping, Semaphore pong) {
            this.ping = ping;
            this.pong = pong;
        }

        public void run() {
            for (int i=0; i<10; i++) {
                ping.P();
                pong.V();
            }
        }

        private Semaphore ping;
        private Semaphore pong;
    }

    /**
     * Test if this module is working.
     */
    public static void selfTest() {
        Semaphore ping = new Semaphore(0);
        Semaphore pong = new Semaphore(0);

        new KThread(new PingTest(ping, pong)).setName("ping").fork();

        for (int i=0; i<10; i++) {
            ping.V();
            pong.P();
        }
    }

    private int value;
    private ThreadQueue waitQueue =
        ThreadedKernel.scheduler.newThreadQueue(false);
}
```

```
package nachos.threads;

import java.util.LinkedList;
import nachos.machine.*;
import nachos.threads.*;

/**
 * A synchronized queue.
 */
public class SynchList {
    /**
     * Allocate a new synchronized queue.
     */
    public SynchList() {
        list = new LinkedList<Object>();
        lock = new Lock();
        listEmpty = new Condition(lock);
    }

    /**
     * Add the specified object to the end of the queue. If another thread is
     * waiting in <tt>removeFirst()</tt>, it is woken up.
     *
     * @param   o       the object to add. Must not be <tt>null</tt>.
     */
    public void add(Object o) {
        Lib.assertTrue(o != null);

        lock.acquire();
        list.add(o);
        listEmpty.wake();
        lock.release();
    }

    /**
     * Remove an object from the front of the queue, blocking until the queue
     * is non-empty if necessary.
     *
     * @return  the element removed from the front of the queue.
     */
    public Object removeFirst() {
        Object o;

        lock.acquire();
        while (list.isEmpty())
            listEmpty.sleep();
        o = list.removeFirst();
        lock.release();

        return o;
    }

    private static class PingTest implements Runnable {
        PingTest(SynchList ping, SynchList pong) {
            this.ping = ping;
            this.pong = pong;
        }

        public void run() {
            for (int i=0; i<10; i++)
                pong.add(ping.removeFirst());
        }

        private SynchList ping;
        private SynchList pong;
    }

    /**
     * Test that this module is working.
     */
    public static void selfTest() {
        SynchList ping = new SynchList();
        SynchList pong = new SynchList();

        new KThread(new PingTest(ping, pong)).setName("ping").fork();

        for (int i=0; i<10; i++) {
            Integer o = new Integer(i);
            ping.add(o);
            Lib.assertTrue(pong.removeFirst() == o);
        }
    }

    private LinkedList<Object> list;
    private Lock lock;
    private Condition listEmpty;
}
```

```
package nachos.threads;

/**
 * Schedules access to some sort of resource with limited access constraints. A
 * thread queue can be used to share this limited access among multiple
 * threads.
 *
 * <p>
 * Examples of limited access in Nachos include:
 *
 * <ol>
 * <li>the right for a thread to use the processor. Only one thread may run on
 * the processor at a time.
 *
 * <li>the right for a thread to acquire a specific lock. A lock may be held by
 * only one thread at a time.
 *
 * <li>the right for a thread to return from <tt>Semaphore.P()</tt> when the
 * semaphore is 0. When another thread calls <tt>Semaphore.V()</tt>, only one
 * thread waiting in <tt>Semaphore.P()</tt> can be awakened.
 *
 * <li>the right for a thread to be woken while sleeping on a condition
 * variable. When another thread calls <tt>Condition.wake()</tt>, only one
 * thread sleeping on the condition variable can be awakened.
 *
 * <li>the right for a thread to return from <tt>KThread.join()</tt>. Threads
 * are not allowed to return from <tt>join()</tt> until the target thread has
 * finished.
 * </ol>
 *
 * All these cases involve limited access because, for each of them, it is not
 * necessarily possible (or correct) for all the threads to have simultaneous
 * access. Some of these cases involve concrete resources (e.g. the processor,
 * or a lock); others are more abstract (e.g. waiting on semaphores, condition
 * variables, or join).
 *
 * <p>
 * All thread queue methods must be invoked with <b>interrupts disabled</b>.
 */
public abstract class ThreadQueue {
    /**
     * Notify this thread queue that the specified thread is waiting for
     * access. This method should only be called if the thread cannot
     * immediately obtain access (e.g. if the thread wants to acquire a lock
     * but another thread already holds the lock).
     *
     * <p>
     * A thread must not simultaneously wait for access to multiple resources.
     * For example, a thread waiting for a lock must not also be waiting to run
     * on the processor; if a thread is waiting for a lock it should be
     * sleeping.
     *
     * <p>
     * However, depending on the specific objects, it may be acceptable for a
     * thread to wait for access to one object while having access to another.
     * For example, a thread may attempt to acquire a lock while holding
     * another lock. Note, though, that the processor cannot be held while
     * waiting for access to anything else.
     *
     * @param   thread  the thread waiting for access.
     */
    public abstract void waitForAccess(KThread thread);

    /**
     * Notify this thread queue that another thread can receive access. Choose
     * and return the next thread to receive access, or <tt>null</tt> if there
     * are no threads waiting.
     *
     * <p>
     * If the limited access object transfers priority, and if there are other
     * threads waiting for access, then they will donate priority to the
     * returned thread.
     *
     * @return  the next thread to receive access, or <tt>null</tt> if there
     *          are no threads waiting.
     */
    public abstract KThread nextThread();

    /**
     * Notify this thread queue that a thread has received access, without
     * going through <tt>request()</tt> and <tt>nextThread()</tt>. For example,
     * if a thread acquires a lock that no other threads are waiting for, it
     * should call this method.
     *
     * <p>
     * This method should not be called for a thread returned from
     * <tt>nextThread()</tt>.
     *
     * @param   thread  the thread that has received access, but was not
     *                  returned from <tt>nextThread()</tt>.
     */
    public abstract void acquire(KThread thread);

    /**
     * Print out all the threads waiting for access, in no particular order.
     */
    public abstract void print();
}
```

```
package nachos.threads;

import nachos.machine.*;

/**
 * A multi-threaded OS kernel.
 */
public class ThreadedKernel extends Kernel {
    /**
     * Allocate a new multi-threaded kernel.
     */
    public ThreadedKernel() {
        super();
    }

    /**
     * Initialize this kernel. Creates a scheduler, the first thread, and an
     * alarm, and enables interrupts. Creates a file system if necessary.
     */
    public void initialize(String[] args) {
        // set scheduler
        String schedulerName = Config.getString("ThreadedKernel.scheduler");
        scheduler = (Scheduler) Lib.constructObject(schedulerName);

        // set fileSystem
        String fileSystemName = Config.getString("ThreadedKernel.fileSystem");
        if (fileSystemName != null)
            fileSystem = (FileSystem) Lib.constructObject(fileSystemName);
        else if (Machine.stubFileSystem() != null)
            fileSystem = Machine.stubFileSystem();
        else
            fileSystem = null;

        // start threading
        new KThread(null);

        alarm  = new Alarm();

        Machine.interrupt().enable();
    }

    /**
     * Test this kernel. Test the <tt>KThread</tt>, <tt>Semaphore</tt>,
     * <tt>SynchList</tt>, and <tt>ElevatorBank</tt> classes. Note that the
     * autograder never calls this method, so it is safe to put additional
     * tests here.
     */
    public void selfTest() {
        KThread.selfTest();
        Semaphore.selfTest();
        SynchList.selfTest();
        if (Machine.bank() != null) {
            ElevatorBank.selfTest();
        }
    }

    /**
     * A threaded kernel does not run user programs, so this method does
     * nothing.
     */
    public void run() {
    }

    /**
     * Terminate this kernel. Never returns.
     */
    public void terminate() {
        Machine.halt();
    }

    /** Globally accessible reference to the scheduler. */
    public static Scheduler scheduler = null;
    /** Globally accessible reference to the alarm. */
    public static Alarm alarm = null;
    /** Globally accessible reference to the file system. */
    public static FileSystem fileSystem = null;

    // dummy variables to make javac smarter
    private static RoundRobinScheduler dummy1 = null;
    private static PriorityScheduler dummy2 = null;
    private static LotteryScheduler dummy3 = null;
    private static Condition2 dummy4 = null;
    private static Communicator dummy5 = null;
    private static Rider dummy6 = null;
    private static ElevatorController dummy7 = null;
}
```

```
package nachos.userprog;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;

/**
 * Provides a simple, synchronized interface to the machine's console. The
 * interface can also be accessed through <tt>OpenFile</tt> objects.
 */
public class SynchConsole {
    /**
     * Allocate a new <tt>SynchConsole</tt>.
     *
     * @param   console the underlying serial console to use.
     */
    public SynchConsole(SerialConsole console) {
        this.console = console;

        Runnable receiveHandler = new Runnable() {
            public void run() { receiveInterrupt(); }
        };
        Runnable sendHandler = new Runnable() {
            public void run() { sendInterrupt(); }
        };
        console.setInterruptHandlers(receiveHandler, sendHandler);
    }

    /**
     * Return the next unsigned byte received (in the range <tt>0</tt> through
     * <tt>255</tt>). If a byte has not arrived at, blocks until a byte
     * arrives, or returns immediately, depending on the value of <i>block</i>.
     *
     * @param   block   <tt>true</tt> if <tt>readByte()</tt> should wait for a
     *                  byte if none is available.
     * @return  the next byte read, or -1 if <tt>block</tt> was <tt>false</tt>
     *          and no byte was available.
     */
    public int readByte(boolean block) {
        int value;
        boolean intStatus = Machine.interrupt().disable();
        readLock.acquire();

        if (block || charAvailable) {
            charAvailable = false;
            readWait.P();

            value = console.readByte();
            Lib.assertTrue(value != -1);
        }
        else {
            value = -1;
        }

        readLock.release();
        Machine.interrupt().restore(intStatus);
        return value;
    }

    /**
     * Return an <tt>OpenFile</tt> that can be used to read this as a file.
     *
     * @return  a file that can read this console.
     */
    public OpenFile openForReading() {
        return new File(true, false);
    }

    private void receiveInterrupt() {
        charAvailable = true;
        readWait.V();
    }

    /**
     * Send a byte. Blocks until the send is complete.
     *
     * @param   value   the byte to be sent (the upper 24 bits are ignored).
     */
    public void writeByte(int value) {
        writeLock.acquire();
        console.writeByte(value);
        writeWait.P();
        writeLock.release();
    }

    /**
     * Return an <tt>OpenFile</tt> that can be used to write this as a file.
     *
     * @return  a file that can write this console.
     */
    public OpenFile openForWriting() {
        return new File(false, true);
    }

    private void sendInterrupt() {
        writeWait.V();
    }

    private boolean charAvailable = false;

    private SerialConsole console;
    private Lock readLock = new Lock();
    private Lock writeLock = new Lock();
    private Semaphore readWait = new Semaphore(0);
    private Semaphore writeWait = new Semaphore(0);

    private class File extends OpenFile {
        File(boolean canRead, boolean canWrite) {
            super(null, "SynchConsole");

            this.canRead = canRead;
            this.canWrite = canWrite;
        }

        public void close() {
            canRead = canWrite = false;
        }

        public int read(byte[] buf, int offset, int length) {
            if (!canRead)
                return 0;

            int i;
            for (i=0; i<length; i++) {
                int value = SynchConsole.this.readByte(false);
                if (value == -1)
                    break;

                buf[offset+i] = (byte) value;
            }
```

```
            return i;
        }

        public int write(byte[] buf, int offset, int length) {
            if (!canWrite)
                return 0;

            for (int i=0; i<length; i++)
                SynchConsole.this.writeByte(buf[offset+i]);

            return length;
        }

        private boolean canRead, canWrite;
    }
}
```

```
package nachos.userprog;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;

/**
 * A UThread is KThread that can execute user program code inside a user
 * process, in addition to Nachos kernel code.
 */
public class UThread extends KThread {
    /**
     * Allocate a new UThread.
     */
    public UThread(UserProcess process) {
        super();

        setTarget(new Runnable() {
                public void run() {
                    runProgram();
                }
            });

        this.process = process;
    }

    private void runProgram() {
        process.initRegisters();
        process.restoreState();

        Machine.processor().run();

        Lib.assertNotReached();
    }

    /**
     * Save state before giving up the processor to another thread.
     */
    protected void saveState() {
        process.saveState();

        for (int i=0; i<Processor.numUserRegisters; i++)
            userRegisters[i] = Machine.processor().readRegister(i);

        super.saveState();
    }

    /**
     * Restore state before receiving the processor again.
     */
    protected void restoreState() {
        super.restoreState();

        for (int i=0; i<Processor.numUserRegisters; i++)
            Machine.processor().writeRegister(i, userRegisters[i]);

        process.restoreState();
    }

    /**
     * Storage for the user register set.
     *
     * <p>
     * A thread capable of running user code actually has <i>two</i> sets of
     * CPU registers: one for its state while executing user code, and one for
     * its state while executing kernel code. While this thread is not running,
     * its user state is stored here.
     */
    public int userRegisters[] = new int[Processor.numUserRegisters];

    /**
     * The process to which this thread belongs.
     */
    public UserProcess process;
}
```

```
package nachos.userprog;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;

/**
 * A kernel that can support multiple user processes.
 */
public class UserKernel extends ThreadedKernel {
    /**
     * Allocate a new user kernel.
     */
    public UserKernel() {
        super();
    }

    /**
     * Initialize this kernel. Creates a synchronized console and sets the
     * processor's exception handler.
     */
    public void initialize(String[] args) {
        super.initialize(args);

        console = new SynchConsole(Machine.console());

        Machine.processor().setExceptionHandler(new Runnable() {
                public void run() { exceptionHandler(); }
            });
    }

    /**
     * Test the console device.
     */
    public void selfTest() {
        super.selfTest();

        System.out.println("Testing the console device. Typed characters");
        System.out.println("will be echoed until q is typed.");

        char c;

        do {
            c = (char) console.readByte(true);
            console.writeByte(c);
        }
        while (c != 'q');

        System.out.println("");
    }

    /**
     * Returns the current process.
     *
     * @return  the current process, or <tt>null</tt> if no process is current.
     */
    public static UserProcess currentProcess() {
        if (!(KThread.currentThread() instanceof UThread))
            return null;

        return ((UThread) KThread.currentThread()).process;
    }

    /**
     * The exception handler. This handler is called by the processor whenever
     * a user instruction causes a processor exception.
     *
     * <p>
     * When the exception handler is invoked, interrupts are enabled, and the
     * processor's cause register contains an integer identifying the cause of
     * the exception (see the <tt>exceptionZZZ</tt> constants in the
     * <tt>Processor</tt> class). If the exception involves a bad virtual
     * address (e.g. page fault, TLB miss, read-only, bus error, or address
     * error), the processor's BadVAddr register identifies the virtual address
     * that caused the exception.
     */
    public void exceptionHandler() {
        Lib.assertTrue(KThread.currentThread() instanceof UThread);

        UserProcess process = ((UThread) KThread.currentThread()).process;
        int cause = Machine.processor().readRegister(Processor.regCause);
        process.handleException(cause);
    }

    /**
     * Start running user programs, by creating a process and running a shell
     * program in it. The name of the shell program it must run is returned by
     * <tt>Machine.getShellProgramName()</tt>.
     *
     * @see     nachos.machine.Machine#getShellProgramName
     */
    public void run() {
        super.run();

        UserProcess process = UserProcess.newUserProcess();

        String shellProgram = Machine.getShellProgramName();
        Lib.assertTrue(process.execute(shellProgram, new String[] { }));

        KThread.currentThread().finish();
    }

    /**
     * Terminate this kernel. Never returns.
     */
    public void terminate() {
        super.terminate();
    }

    /** Globally accessible reference to the synchronized console. */
    public static SynchConsole console;

    // dummy variables to make javac smarter
    private static Coff dummy1 = null;
}
```

```
package nachos.userprog;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;

import java.io.EOFException;

/**
 * Encapsulates the state of a user process that is not contained in its
 * user thread (or threads). This includes its address translation state, a
 * file table, and information about the program being executed.
 *
 * <p>
 * This class is extended by other classes to support additional functionality
 * (such as additional syscalls).
 *
 * @see nachos.vm.VMProcess
 * @see nachos.network.NetProcess
 */
public class UserProcess {
    /**
     * Allocate a new process.
     */
    public UserProcess() {
        int numPhysPages = Machine.processor().getNumPhysPages();
        pageTable = new TranslationEntry[numPhysPages];
        for (int i=0; i<numPhysPages; i++)
            pageTable[i] = new TranslationEntry(i,i, true,false,false,false);
    }

    /**
     * Allocate and return a new process of the correct class. The class name
     * is specified by the <tt>nachos.conf</tt> key
     * <tt>Kernel.processClassName</tt>.
     *
     * @return  a new process of the correct class.
     */
    public static UserProcess newUserProcess() {
        return (UserProcess)Lib.constructObject(Machine.getProcessClassName());
    }

    /**
     * Execute the specified program with the specified arguments. Attempts to
     * load the program, and then forks a thread to run it.
     *
     * @param   name    the name of the file containing the executable.
     * @param   args    the arguments to pass to the executable.
     * @return  <tt>true</tt> if the program was successfully executed.
     */
    public boolean execute(String name, String[] args) {
        if (!load(name, args))
            return false;

        new UThread(this).setName(name).fork();

        return true;
    }

    /**
     * Save the state of this process in preparation for a context switch.
     * Called by <tt>UThread.saveState()</tt>.
     */
    public void saveState() {
    }

    /**
     * Restore the state of this process after a context switch. Called by
     * <tt>UThread.restoreState()</tt>.
     */
    public void restoreState() {
        Machine.processor().setPageTable(pageTable);
    }

    /**
     * Read a null-terminated string from this process's virtual memory. Read
     * at most <tt>maxLength + 1</tt> bytes from the specified address, search
     * for the null terminator, and convert it to a <tt>java.lang.String</tt>,
     * without including the null terminator. If no null terminator is found,
     * returns <tt>null</tt>.
     *
     * @param   vaddr   the starting virtual address of the null-terminated
     *                  string.
     * @param   maxLength       the maximum number of characters in the string,
     *                          not including the null terminator.
     * @return  the string read, or <tt>null</tt> if no null terminator was
     *          found.
     */
    public String readVirtualMemoryString(int vaddr, int maxLength) {
        Lib.assertTrue(maxLength >= 0);

        byte[] bytes = new byte[maxLength+1];

        int bytesRead = readVirtualMemory(vaddr, bytes);

        for (int length=0; length<bytesRead; length++) {
            if (bytes[length] == 0)
                return new String(bytes, 0, length);
        }

        return null;
    }

    /**
     * Transfer data from this process's virtual memory to all of the specified
     * array. Same as <tt>readVirtualMemory(vaddr, data, 0, data.length)</tt>.
     *
     * @param   vaddr   the first byte of virtual memory to read.
     * @param   data    the array where the data will be stored.
     * @return  the number of bytes successfully transferred.
     */
    public int readVirtualMemory(int vaddr, byte[] data) {
        return readVirtualMemory(vaddr, data, 0, data.length);
    }

    /**
     * Transfer data from this process's virtual memory to the specified array.
     * This method handles address translation details. This method must
     * <i>not</i> destroy the current process if an error occurs, but instead
     * should return the number of bytes successfully copied (or zero if no
     * data could be copied).
     *
     * @param   vaddr   the first byte of virtual memory to read.
     * @param   data    the array where the data will be stored.
     * @param   offset  the first byte to write in the array.
     * @param   length  the number of bytes to transfer from virtual memory to
     *                  the array.
     * @return  the number of bytes successfully transferred.
     */
    public int readVirtualMemory(int vaddr, byte[] data, int offset,
```

```
                              int length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <= data.length);

    byte[] memory = Machine.processor().getMemory();

    // for now, just assume that virtual addresses equal physical addresses
    if (vaddr < 0 || vaddr >= memory.length)
        return 0;

    int amount = Math.min(length, memory.length-vaddr);
    System.arraycopy(memory, vaddr, data, offset, amount);

    return amount;
}

/**
 * Transfer all data from the specified array to this process's virtual
 * memory.
 * Same as <tt>writeVirtualMemory(vaddr, data, 0, data.length)</tt>.
 *
 * @param   vaddr   the first byte of virtual memory to write.
 * @param   data    the array containing the data to transfer.
 * @return  the number of bytes successfully transferred.
 */
public int writeVirtualMemory(int vaddr, byte[] data) {
    return writeVirtualMemory(vaddr, data, 0, data.length);
}

/**
 * Transfer data from the specified array to this process's virtual memory.
 * This method handles address translation details. This method must
 * <i>not</i> destroy the current process if an error occurs, but instead
 * should return the number of bytes successfully copied (or zero if no
 * data could be copied).
 *
 * @param   vaddr   the first byte of virtual memory to write.
 * @param   data    the array containing the data to transfer.
 * @param   offset  the first byte to transfer from the array.
 * @param   length  the number of bytes to transfer from the array to
 *                  virtual memory.
 * @return  the number of bytes successfully transferred.
 */
public int writeVirtualMemory(int vaddr, byte[] data, int offset,
                              int length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <= data.length);

    byte[] memory = Machine.processor().getMemory();

    // for now, just assume that virtual addresses equal physical addresses
    if (vaddr < 0 || vaddr >= memory.length)
        return 0;

    int amount = Math.min(length, memory.length-vaddr);
    System.arraycopy(data, offset, memory, vaddr, amount);

    return amount;
}

/**
 * Load the executable with the specified name into this process, and
 * prepare to pass it the specified arguments. Opens the executable, reads
 * its header information, and copies sections and arguments into this
 * process's virtual memory.
 *
 * @param   name    the name of the file containing the executable.
 * @param   args    the arguments to pass to the executable.
 * @return  <tt>true</tt> if the executable was successfully loaded.
 */
private boolean load(String name, String[] args) {
    Lib.debug(dbgProcess, "UserProcess.load(\"" + name + "\")");

    OpenFile executable = ThreadedKernel.fileSystem.open(name, false);
    if (executable == null) {
        Lib.debug(dbgProcess, "\topen failed");
        return false;
    }

    try {
        coff = new Coff(executable);
    }
    catch (EOFException e) {
        executable.close();
        Lib.debug(dbgProcess, "\tcoff load failed");
        return false;
    }

    // make sure the sections are contiguous and start at page 0
    numPages = 0;
    for (int s=0; s<coff.getNumSections(); s++) {
        CoffSection section = coff.getSection(s);
        if (section.getFirstVPN() != numPages) {
            coff.close();
            Lib.debug(dbgProcess, "\tfragmented executable");
            return false;
        }
        numPages += section.getLength();
    }

    // make sure the argv array will fit in one page
    byte[][] argv = new byte[args.length][];
    int argsSize = 0;
    for (int i=0; i<args.length; i++) {
        argv[i] = args[i].getBytes();
        // 4 bytes for argv[] pointer; then string plus one for null byte
        argsSize += 4 + argv[i].length + 1;
    }
    if (argsSize > pageSize) {
        coff.close();
        Lib.debug(dbgProcess, "\targuments too long");
        return false;
    }

    // program counter initially points at the program entry point
    initialPC = coff.getEntryPoint();

    // next comes the stack; stack pointer initially points to top of it
    numPages += stackPages;
    initialSP = numPages*pageSize;

    // and finally reserve 1 page for arguments
    numPages++;

    if (!loadSections())
        return false;

    // store arguments in last page
    int entryOffset = (numPages-1)*pageSize;
    int stringOffset = entryOffset + args.length*4;

    this.argc = args.length;
```

```
    this.argv = entryOffset;

    for (int i=0; i<argv.length; i++) {
        byte[] stringOffsetBytes = Lib.bytesFromInt(stringOffset);
        Lib.assertTrue(writeVirtualMemory(entryOffset,stringOffsetBytes) == 4);
        entryOffset += 4;
        Lib.assertTrue(writeVirtualMemory(stringOffset, argv[i]) ==
                   argv[i].length);
        stringOffset += argv[i].length;
        Lib.assertTrue(writeVirtualMemory(stringOffset,new byte[] { 0 }) == 1);
        stringOffset += 1;
    }

    return true;
}

/**
 * Allocates memory for this process, and loads the COFF sections into
 * memory. If this returns successfully, the process will definitely be
 * run (this is the last step in process initialization that can fail).
 *
 * @return  <tt>true</tt> if the sections were successfully loaded.
 */
protected boolean loadSections() {
    if (numPages > Machine.processor().getNumPhysPages()) {
        coff.close();
        Lib.debug(dbgProcess, "\tinsufficient physical memory");
        return false;
    }

    // load sections
    for (int s=0; s<coff.getNumSections(); s++) {
        CoffSection section = coff.getSection(s);

        Lib.debug(dbgProcess, "\tinitializing " + section.getName()
                  + " section (" + section.getLength() + " pages)");

        for (int i=0; i<section.getLength(); i++) {
            int vpn = section.getFirstVPN()+i;

            // for now, just assume virtual addresses=physical addresses
            section.loadPage(i, vpn);
        }
    }

    return true;
}

/**
 * Release any resources allocated by <tt>loadSections()</tt>.
 */
protected void unloadSections() {
}

/**
 * Initialize the processor's registers in preparation for running the
 * program loaded into this process. Set the PC register to point at the
 * start function, set the stack pointer register to point at the top of
 * the stack, set the A0 and A1 registers to argc and argv, respectively,
 * and initialize all other registers to 0.
 */
public void initRegisters() {
    Processor processor = Machine.processor();

    // by default, everything's 0
    for (int i=0; i<processor.numUserRegisters; i++)
        processor.writeRegister(i, 0);

    // initialize PC and SP according
    processor.writeRegister(Processor.regPC, initialPC);
    processor.writeRegister(Processor.regSP, initialSP);

    // initialize the first two argument registers to argc and argv
    processor.writeRegister(Processor.regA0, argc);
    processor.writeRegister(Processor.regA1, argv);
}

/**
 * Handle the halt() system call.
 */
private int handleHalt() {

    Machine.halt();

    Lib.assertNotReached("Machine.halt() did not halt machine!");
    return 0;
}


private static final int
    syscallHalt = 0,
    syscallExit = 1,
    syscallExec = 2,
    syscallJoin = 3,
    syscallCreate = 4,
    syscallOpen = 5,
    syscallRead = 6,
    syscallWrite = 7,
    syscallClose = 8,
    syscallUnlink = 9;

/**
 * Handle a syscall exception. Called by <tt>handleException()</tt>. The
 * <i>syscall</i> argument identifies which syscall the user executed:
 *
 * <table>
 * <tr><td>syscall#</td><td>syscall prototype</td></tr>
 * <tr><td>0</td><td><tt>void halt();</tt></td></tr>
 * <tr><td>1</td><td><tt>void exit(int status);</tt></td></tr>
 * <tr><td>2</td><td><tt>int  exec(char *name, int argc, char **argv);
 *                                                              </tt></td></tr>
 * <tr><td>3</td><td><tt>int  join(int pid, int *status);</tt></td></tr>
 * <tr><td>4</td><td><tt>int  creat(char *name);</tt></td></tr>
 * <tr><td>5</td><td><tt>int  open(char *name);</tt></td></tr>
 * <tr><td>6</td><td><tt>int  read(int fd, char *buffer, int size);
 *                                                              </tt></td></tr>
 * <tr><td>7</td><td><tt>int  write(int fd, char *buffer, int size);
 *                                                              </tt></td></tr>
 * <tr><td>8</td><td><tt>int  close(int fd);</tt></td></tr>
 * <tr><td>9</td><td><tt>int  unlink(char *name);</tt></td></tr>
 * </table>
 *
 * @param   syscall the syscall number.
 * @param   a0      the first syscall argument.
 * @param   a1      the second syscall argument.
 * @param   a2      the third syscall argument.
 * @param   a3      the fourth syscall argument.
 * @return  the value to be returned to the user.
 */
public int handleSyscall(int syscall, int a0, int a1, int a2, int a3) {
```

```
        switch (syscall) {
        case syscallHalt:
            return handleHalt();


        default:
            Lib.debug(dbgProcess, "Unknown syscall " + syscall);
            Lib.assertNotReached("Unknown system call!");
        }
        return 0;
    }

    /**
     * Handle a user exception. Called by
     * <tt>UserKernel.exceptionHandler()</tt>. The
     * <i>cause</i> argument identifies which exception occurred; see the
     * <tt>Processor.exceptionZZZ</tt> constants.
     *
     * @param   cause   the user exception that occurred.
     */
    public void handleException(int cause) {
        Processor processor = Machine.processor();

        switch (cause) {
        case Processor.exceptionSyscall:
            int result = handleSyscall(processor.readRegister(Processor.regV0),
                                       processor.readRegister(Processor.regA0),
                                       processor.readRegister(Processor.regA1),
                                       processor.readRegister(Processor.regA2),
                                       processor.readRegister(Processor.regA3)
                                       );
            processor.writeRegister(Processor.regV0, result);
            processor.advancePC();
            break;

        default:
            Lib.debug(dbgProcess, "Unexpected exception: " +
                      Processor.exceptionNames[cause]);
            Lib.assertNotReached("Unexpected exception");
        }
    }

    /** The program being run by this process. */
    protected Coff coff;

    /** This process's page table. */
    protected TranslationEntry[] pageTable;
    /** The number of contiguous pages occupied by the program. */
    protected int numPages;

    /** The number of pages in the program's stack. */
    protected final int stackPages = 8;

    private int initialPC, initialSP;
    private int argc, argv;

    private static final int pageSize = Processor.pageSize;
    private static final char dbgProcess = 'a';
}
```

```
package nachos.vm;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;
import nachos.vm.*;

/**
 * A kernel that can support multiple demand-paging user processes.
 */
public class VMKernel extends UserKernel {
    /**
     * Allocate a new VM kernel.
     */
    public VMKernel() {
        super();
    }

    /**
     * Initialize this kernel.
     */
    public void initialize(String[] args) {
        super.initialize(args);
    }

    /**
     * Test this kernel.
     */
    public void selfTest() {
        super.selfTest();
    }

    /**
     * Start running user programs.
     */
    public void run() {
        super.run();
    }

    /**
     * Terminate this kernel. Never returns.
     */
    public void terminate() {
        super.terminate();
    }

    // dummy variables to make javac smarter
    private static VMProcess dummy1 = null;

    private static final char dbgVM = 'v';
}
```

```
package nachos.vm;

import nachos.machine.*;
import nachos.threads.*;
import nachos.userprog.*;
import nachos.vm.*;

/**
 * A <tt>UserProcess</tt> that supports demand-paging.
 */
public class VMProcess extends UserProcess {
    /**
     * Allocate a new process.
     */
    public VMProcess() {
        super();
    }

    /**
     * Save the state of this process in preparation for a context switch.
     * Called by <tt>UThread.saveState()</tt>.
     */
    public void saveState() {
        super.saveState();
    }

    /**
     * Restore the state of this process after a context switch. Called by
     * <tt>UThread.restoreState()</tt>.
     */
    public void restoreState() {
        super.restoreState();
    }

    /**
     * Initializes page tables for this process so that the executable can be
     * demand-paged.
     *
     * @return  <tt>true</tt> if successful.
     */
    protected boolean loadSections() {
        return super.loadSections();
    }

    /**
     * Release any resources allocated by <tt>loadSections()</tt>.
     */
    protected void unloadSections() {
        super.unloadSections();
    }

    /**
     * Handle a user exception. Called by
     * <tt>UserKernel.exceptionHandler()</tt>. The
     * <i>cause</i> argument identifies which exception occurred; see the
     * <tt>Processor.exceptionZZZ</tt> constants.
     *
     * @param   cause   the user exception that occurred.
     */
    public void handleException(int cause) {
        Processor processor = Machine.processor();

        switch (cause) {
        default:
            super.handleException(cause);
            break;
        }
    }

    private static final int pageSize = Processor.pageSize;
    private static final char dbgProcess = 'a';
    private static final char dbgVM = 'v';
}
```

A Guide to Nachos 5.0j

Dan Hettena

Rick Cox

rick@rescomp.berkeley.edu

We have ported the Nachos instructional operating system [1 (http://http.cs.berkeley.edu/~tea/nachos/nachos.ps)] to Java, and in the process of doing so, many details changed (hopefully for the better). [2 (http://www.cs.duke.edu/~narten/110/nachos/main/main.html)] remains an excellent resource for learning about the C++ versions of Nachos, but an update is necessary to account for the differences between the Java version and the C++ versions.

We attempt to describe Nachos 5.0j in the same way that [2 (http://www.cs.duke.edu/~narten/110/nachos/main/main.html)] described previous versions of Nachos, except that we defer some of the details to the Javadoc-generated documentation. We do not claim any originality in this documentation, and freely offer any deserved credit to Narten.

Table of Contents

1. Nachos and the Java Port

The Nachos instructional operating system, developed at Berkeley, was first tested on guinea pig students in 1992 [1 (http://http.cs.berkeley.edu/~tea/nachos/nachos.ps)]. The authors intended it to be a simple, yet realistic, project for undergraduate operating systems classes. Nachos is now in wide use.

The original Nachos, written in a subset of C++ (with a little assembly), ran as a regular UNIX process. It simulated the hardware devices of a simple computer: it had a timer, a console, a MIPS R3000 processor, a disk, and a network link. In order to achieve reasonable performance, the operating system kernel ran natively, while user processes ran on the simulated processor. Because it was simulated, multiple Nachos instances could run on the same physical computer.

1.1. Why Java?

Despite the success of Nachos, there are good reasons to believe that it would be more useful in Java:

- Java is much simpler than C++. It is not necessary to restrict Nachos to a subset of the language; students can understand the whole language.
- Java is type-safe. C++ is not type-safe; it is possible for a C++ program to perform a legal operation (e.g. writing off the end of an array) such that the operation of the program can no longer be described in terms of the C++ language. This turns out to be a major problem; some project groups are unable to debug their projects within the alotted time, primarily because of bugs not at all related to operating systems concepts.
- It is much more reasonable to machine-grade a Java project than a C++ project.
- Many undergraduate data structures classes, including the one at Berkeley, now use Java, not C++; students know Java well.
- Java is relatively portable. Nachos 4.0 uses unportable assembly to support multithreading. Adding a new target to Nachos 4.0 required writing a bit of additional code for the port.

1.2. Will it work?

One of the first concerns many people have about Java is its speed. It is an undebatable fact that Java programs run slower than their C++ equivalents. This statement can be misleading, though:

- Compiling is a significant part of the Nachos 4.0 debug cycle. Because javac compiles as much as it can everytime it is invoked, Nachos 5.0j actually compiles faster than Nachos 4.0 (running on a local disk partition with no optimizations enabled).
- Generating large files on network partitions further slows down the debug cycle. Nachos 5.0j's .class files are significantly smaller than Nachos 4.0's .o files, even when compiling with -Os. This is in part due to C++ templates, which, without a smart compiler or careful management, get very big.
- Type-safe languages are widely known to make debugging cycles more effective.

Another common concern is that writing an operating system in a type-safe language is unrealistic. In short, it *is* unrealistic, but not as unrealistic as you might think. Two aspects of real operating systems are lost by using Java, but neither are critical:

- Since the JVM provides threads for Nachos 5.0j, the context switch code is no longer exposed. In Nachos 4.0, students could read the assembly code used to switch between threads. But, as mentioned above, this posed a portability problem.
- The kernel can allocate kernel memory without releasing it; the garbage collector will release it. In Linux, this would be similar to removing all calls to `kfree`. This, however, is conceptually one of the simplest forms of resource allocation within the kernel (there's a lot more to Linux than `kmalloc` and `kfree`). The Nachos kernel must still directly manage the allocation of physical pages among processes, and must close files when processes exit, for example.

2. Nachos Machine

Nachos simulates a real CPU and harware devices, including interrupts and memory management. The Java package nachos.machine provides this simulation.

2.1. Configuring Nachos

The nachos simulation is configured for the various projects using the `nachos.conf` file (for the most part, this file is equivelant to the BIOS or OpenFirmware configuration of modern PCs or Macintoshes). It specifies which hardware devices to include in the simulation as well as which Nachos kernel to use. The project directories include appropriate configurations, and, where neccessary, the project handouts document any changes to this file required to complete the project.

2.2. Boot Process

The nachos boot process is similar to that of a real machine. An instance of the nachos.machine.Machine class is created to begin booting. The hardware (Machine object) first initializes the devices including the interrupt controller, timer, elevator controller, MIPS processor, console, and file system.

The Machine object then hands control to the particular AutoGrader in use, an action equivelant to loading the bootstrap code from the boot sector of the disk. It is the AutoGrader that creates a Nachos kernel, starting the operating system. Students need not worry about this step in the boot process - the interesting part begins with the kernel.

A Nachos kernel is just a subclass of nachos.machine.Kernel. For instance, the thread project uses nachos.threads.ThreadedKernel (and later projects inherit from ThreadedKernel).

2.3. Nachos Hardware Devices

The Nachos machine simulation includes several hardware devices. Some would be found in most modern computers (e.g. the network interface), while others (such as the elevator controller) are unique to Nachos. Most classes in the `machine` directory are part of the hardware simulation, while all classes outside that directory are part of the Nachos operating system.

2.3.1. Interrupt Management

The nachos.machine.Interrupt class simulates interrupts by maintaining an event queue together with a simulated clock. As the clock ticks, the event queue is examined to find events scheduled to take place now. The interrupt controller is returned by `Machine.interrupt()`.

The clock is maintained entirely in software and ticks only under the following conditions:

- Every time interrupts are re-enabled (i.e. only when interrupts are disabled and get enabled again), the clock advances 10 ticks. Nachos code frequently disables and restores interrupts for mutual exclusion purposes by making explicit calls to `disable()` and `restore()`.
- Whenever the MIPS simulator executes one instruction, the clock advances one tick.

> **Note:** Nachos C++ users: Nachos C++ allowed the simulated time to be advanced to that of the next interrupt whenever the ready list is empty. This provides a small performance gain, but it creates unnatural interaction between the kernel and the hardware, and it is unnecessary (a normal OS uses an idle thread, and this is exactly what Nachos does now).

Whenever the clock advances, the event queue is examined and any pending interrupt events are serviced by invoking the device event handler associated with the event. Note that this handler is *not* an interrupt handler (a.k.a. interrupt service routine). Interrupt handlers are part of software, while device event handlers are part of the hardware simulation. A device event handler will *invoke* the software interrupt handler for the device, as we will see later. For this reason, the Interrupt class disables interrupts before calling a device event handler.

Caution

Due to a bug in the current release of Nachos, *only the timer interrupt handler may cause a context switch* (the problem is that a few device event handlers are not reentrant; in order for an interrupt handler to be allowed to do a context switch, the device event handler that invoked it must be reentrant). All interrupt handlers besides the timer interrupt handler must not directly or indirectly cause a context switch before returning, or deadlock may occur. However, you probably won't even want to context switch in any other interrupt handler anyway, so this should not be a problem.

The Interrupt class accomplishes the above through three methods. These methods are only accessible to hardware simulation devices.

- `schedule()` takes a time and a device event handler as arguments, and schedules the specified handler to be called at the specified time.
- `tick()` advances the time by 1 tick or 10 ticks, depending on whether Nachos is in user mode or kernel mode. It is called by `setStatus()` whenever interrupts go from being disabled to being enabled, and also by `Processor.run()` after each user instruction is executed.
- `checkIfDue()` invokes event handlers for queued events until no more events are due to occur. It is invoked by `tick()`.

The Interrupt class also simulates the hardware interface to enable and disable interrupts (see the Javadoc for Interrupt).

The remainder of the hardware devices present in Nachos depend on the Interrupt device. No hardware devices in Nachos create threads, thus, the only time the code in the device classes execute is due to a function call by the running KThread or due to an interrupt handler executed by the Interrupt object.

2.3.2. Timer

Nachos provides an instance of a Timer to simulate a real-time clock, generating interrupts at regular intervals. It is implemented using the event driven interrupt mechanism described above. `Machine.timer()` returns a reference to this timer.

Timer supports only two operations:

- `getTime()` returns the number of ticks since Nachos started.
- `setInterruptHandler()` sets the timer interrupt handler, which is invoked by the simulated timer approximately every `Stats.TimerTicks` ticks.

The timer can be used to provide preemption. Note however that the timer interrupts do not always occur at exactly the same intervals. Do not rely on timer interrupts being equally spaced; instead, use `getTime()`.

2.3.3. Serial Console

Nachos provides three classes of I/O devices with read/write interfaces, of which the simplest is the serial console. The serial console, specified by the SerialConsole class, simulates the behavior of a serial port. It provides byte-wide read and write primitives that never block. The machine's serial console is returned by `Machine.console()`.

The read operation tests if a byte of data is ready to be returned. If so, it returns the byte immediately, and otherwise it returns -1. When another byte of data is received, a receive interrupt occurs. Only one byte can be queued at a time, so it is not possible for two receive interrupts to occur without an intervening read operation.

The write operation starts transmitting a byte of data and returns immediately. When the transmission is complete and another byte can be sent, a send interrupt occurs. If two writes occur without an intervening send interrupt, the actual data transmitted is undefined (so the kernel should always wait for a send interrupt first).

Note that the receive interrupt handler and send interrupt handler are provided by the kernel, by calling `setInterruptHandlers()`.

Implementation note: in a normal Nachos session, the serial console is implemented by class StandardConsole, which uses stdin and stdout. It schedules a read device event every `Stats.ConsoleTime` ticks to poll stdin for another byte of data. If a byte is present, it stores it and invokes the receive interrupt handler.

2.3.4. Disk

The file systems project has not yet been ported, so the disk has not been tested.

2.3.5. Network Link

Separate Nachos instances running on the same real-life machine can communicate with each other over a network, using the NetworkLink class. An instance of this class is returned by `Machine.networkLink()`.

The network link's interface is similar to the serial console's interface, except that instead of receiving and sending bytes at a time, the network link receives and sends packets at a time. Packets are instances of the Packet class.

Each network link has a *link address*, a number that uniquely identifies the link on the network. The link address is returned by `getLinkAddress()`.

A packet consists of a header and some data bytes. The header specifies the link address of the machine sending the packet (the source link address), the link address of the machine to which the packet is being sent (the destination link address), and the number of bytes of data contained in the packet. The data bytes are not analyzed by the network hardware, while the header is. When a link transmits a packet, it transmits it only to the link specified in the destination link address field of the header. Note that the source address can be forged.

The remainder of the interface to NetworkLink is equivalent to that of SerialConsole. The kernel can check for a packet by calling `receive()`, which returns `null` if no packet is available. Whenever a packet arrives, a receive interrupt is generated. The kernel can send a packet by calling `send()`, but it must wait for a send interrupt before attempting to send another packet.

3. Threads and Scheduling

Nachos provides a kernel threading package, allowing multiple tasks to run concurrently (see nachos.threads.ThreadedKernel and nachos.threads.KThread). Once the user-prcesses are implmented (phase 2), some threads may be running the MIPS processor simulation. As the scheduler and thread package are concerned, there is no difference between a thread running the MIPS simulation and one running just kernel Java code.

3.1. Thread Package

All Nachos threads are instances of nachos.threads.KThread (threads capable of running user-level MIPS code are a subclass of KThread, nachos.userprog.UThread). A nachos.machine.TCB object is contained by each KThread and provides low-level support for context switches, thread creation, thread destruction, and thread yield.

Every KThread has a `status` member that tracks the state of the thread. Certain KThread methods will fail (with a `Lib.assert()`) if called on threads in the wrong state; check the KThread Javadoc for details.

`statusNew`

A newly created, yet to be forked thread.

`statusReady`

A thread waiting for access to the CPU. `KThread.ready()` will add the thread to the ready queue and set the status to `statusReady`.

`statusRunning`

The thread currently using the CPU. `KThread.restoreState()` is responsible for setting `status` to `statusRunning`, and is called by `KThread.runNextThread()`.

`statusBlocked`

A thread which is asleep (as set by `KThread.sleep()`), waiting on some resource besides the CPU.

`statusFinished`

A thread scheduled for destruction. Use `KThread.finish()` to set this status.

Internally, Nachos implements threading using a Java thread for each TCB. The Java threads are synchronized by the TCBs such that exactly one is running at any given time. This provides the illusion of context switches saving state for the current thread and loading the saved state of the new thread. This detail, however, is only important for use with debuggers (which will show multiple Java threads), as the behavior is equivelant to a context switch on a real processor.

3.2. Scheduler

A sub-class (specified in the `nachos.conf`) of the abstract base class nachos.threads.Scheduler is responsible for scheduling threads for all limited resources, be it the CPU, a synchronization construct like a lock, or even a thread join operation. For each resource a nachos.threads.ThreadQueue is created by `Scheduler.newThreadQueue()`. The implementation of the resource (e.g. nachos.threads.Semaphore class) is responsible for adding KThreads to the ThreadQueue (`ThreadQueue.waitForAccess()`) and requesting the ThreadQueue return the next thread (`ThreadQueue.nextThread()`). Thus, all scheduling decisions (including those regarding the CPU's ready queue) reduce to the selection of the next thread by the ThreadQueue objects [1].

Various phases of the project will require modifications to the scheduler base class. The nachos.threads.RoundRobinScheduler is the default, and implements a fully functional (though naive) FIFO scheduler. Phase 1 of the projects requires the student to complete the nachos.threads.PriorityScheduler; for phase 2, students complete nachos.threads.LotteryScheduler.

3.3. Creating the First Thread

Upto the point where the Kernel is created, the boot process is fairly easy to follow - Nachos is just making Java objects, same as any other Java program. Also like any other single-threaded Java program, Nachos code is executing on the initial Java thread created automaticaly for it by Java. `ThreadedKernel.initialize()` has the task of starting threading:

```
public void initialize(String[] args) {
      ...
      // start threading
      new KThread(null);
      ...
}
```

The first clue that something special is happening should be that the new KThread object created is not stored in a variable inside `initialize()`. The constructor for KThread follows the following procedure the first time it is called:

1. Create the ready queue (`ThreadedKernel.scheduler.newThreadQueue()`).
2. Allocate the CPU to the new KThread object being created (`readyQueue.acquire(this)`).
3. Set `KThread.currentThread` to the new KThread being made.
4. Set the TCB object of the new KThread to `TCB.currentTCB()`. In doing so, the currently running Java thread is assigned to the new KThread object being created.
5. Change the `status` of the new KThread from the default (`statusNew`) to `statusRunning`. This bypasses the `statusReady` state.
6. Create an idle thread.
    a. Make another new KThread, with the target set to an infinite `yield()` loop.
    b. Fork the idle thread off from the main thread.

After this procedure, there are two KThread objects, each with a TCB object (one for the main thread, and one for the idle thread). The main thread is not special - the scheduler treats it exactly like any other KThread. The main thread can create other threads, it can die, it can block. The Nachos session will not end until all KThreads finish, regardless of whether the main thread is alive.

For the most part the idle thread is also a normal thread, which can be contexted switched like any other. The only difference is it will never be added to the ready queue (`KThread.ready()` has an explicit check for the idle thread). Instead, if `readyQueue.nextThread()` returns `null`, the thread system will switch to the idle thread.

Note: While the Nachos idle thread does nothing but `yield()` forever, some systems use the idle thread to do work. One common use is zeroing memory to prepare it for reallocation.

3.4. Creating More Threads

Creating subsequent threads is much simpler. As described in the KThread Javadoc, a new KThread is created, passing the constructor a Runnable object. Then, `fork()` is called:

```
KThread newThread = new KThread(myRunnable);
...
newThread.fork();
```

This sequence results in the new thread being placed on the ready queue. The currently running thread does not immediatly yield, however.

3.4.1. Java Anonymous Classes in Nachos

The Nachos source is relatively clean, using only basic Java, with the exception of the use of anonymous classes to replicate the functionality of function pointers in C++. The following code illustrates the use of an anonymous class to create a new KThread object which, when forked, will execute the `myFunction()` method of the encolosing object.

```
Runnable myRunnable = new Runnable() {
public void run() {
    myFunction();
}
    };
KThread newThread = new KThread(myRunnable);
```

This code creates a new object of type Runnable inside the context of the enclosing object. Since `myRunnable` has no method `myFunction()`, executing `myRunnable.run()` will cause Java to look in the enclosing class for a `myFunction()` method.

3.5. On Thread Death

All threads have some resources allocated to them which are neccessary for the thread to run (e.g. the TCB object). Since the thread itself cannot deallocate these resources while it is running, it leaves a virtual will asking the next thread which runs to deallocate its resources. This is implemented in

`KThread.finish()`, which sets `KThread.toBeDestroyed` to the currently running thread. It then sets current thread's `status` field to `statusFinished` and calls `sleep()`.

Since the thread is not waiting on a ThreadQueue object, its sleep will be permanent (that is, Nachos will never try to wake the thread). This scheme does, however, require that after every context switch, the newly running thread must check `toBeDestroyed`.

> **Note:** In the C++ version of Nachos, thread death was complicated by the explicit memory deallocation required, combined with dangling references that still pointing to the thread after death (for example, most thread `join()` implementations requires some reference to the thread). In Java, the garbage collector is responsible for noticing when these references are detached, significantly simplifying the thread finishing process.

4. The Nachos Simulated MIPS Machine

Nachos simulates a machine with a processor that roughly approximates the MIPS architecture. In addition, an event-driven simulated clock provides a mechanism to schedule events and execute them at a later time. This is a building block for classes that simulate various hardware devices: a timer, an elevator bank, a console, a disk, and a network link.

The simulated MIPS processor can execute arbitrary programs. One simply loads instructions into the processor's memory, initializes registers (including the program counter, regPC) and then tells the processor to start executing instructions. The processor then fetches the instruction that regPC points at, decodes it, and executes it. The process is repeated indefinitely, until either an instruction causes an exception or a hardware interrupt is generated. When an exception or interrupt takes place, execution of MIPS instructions is suspended, and a Nachos interrupt service routine is invoked to deal with the condition.

Conceptually, Nachos has two modes of execution, one of which is the MIPS simulator. Nachos executes user-level processes by loading them into the simulator's memory, initializing the simulator's registers and then running the simulator. User-programs can only access the memory associated with the simulated processor. The second mode corresponds to the Nachos "kernel". The kernel executes when Nachos first starts up, or when a user-program executes an instruction that causes an exception (e.g., illegal instruction, page fault, system call, etc.). In kernel mode, Nachos executes the way normal Java programs execute. That is, the statements corresponding to the Nachos source code are executed, and the memory accessed corresponds to the memory assigned to Nachos variables.

4.1. Processor Components

The Nachos/MIPS processor is implemented by the Processor class, an instance of which is created when Nachos first starts up. The Processor class exports a number of public methods and fields that the Nachos kernel accesses directly. In the following, we describe some of the important variables of the Processor class; describing their role helps explain what the simulated hardware does.

The processor provides registers and physical memory, and supports virtual memory. It provides operations to run the machine and to examine and modify its current state. When Nachos first starts up, it creates an instance of the Processor class and makes it available through `Machine.processor()`. The following aspects of the processor are accessible to the Nachos kernel:

Registers

> The processor's registers are accessible through `readRegister()` and `writeRegister()`. The registers include MIPS registers 0 through 31, the low and high registers used for multiplication and division, the program counter and next program counter registers (two are necessary because of branch delay slots), a register specifying the cause of the most recent exception, and a register specifying the virtual memory address associated with the most recent exception. Recall that the stack pointer register and return address registers are general MIPS registers (specifically, they are registers 29 and 31, respectively). Recall also that r0 is always 0 and cannot be modified.

Physical memory

> Memory is byte-addressable and organized into 1-kilobyte pages, the same size as disk sectors. A reference to the main memory array is returned by `getMemory()`. Memory corresponding to physical address `m` can be accessed in Nachos at `Machine.processor().getMemory()[m]`. The number of pages of physical memory is returned by `getNumPhysPages()`.

Virtual memory

> The processor supports VM through either a single linear page table or a software-managed TLB (but not both). The mode of address translation is actually used is determined by `nachos.conf`, and is returned by `hasTLB()`. If the processor does not have a TLB, the kernel can tell it what page table to use by calling `setPageTable()`. If the processor does have a TLB, the kernel can query the size of the TLB by calling `getTLBSize()`, and the kernel can read and write TLB entries by calling `readTLBEntry()` and `writeTLBEntry()`.

Exceptions

> When the processor attempts to execute an instruction and it results in an exception, the kernel exception handler is invoked. The kernel must tell the processor where this exception handler is by invoking `setExceptionHandler()`. If the exception resulted from a syscall instruction, it is the kernel's responsibility to advance the PC register, which it should do by calling `advancePC()`.

At this point, we know enough about the Processor class to explain how it executes arbitrary user programs. First, we load the program's instructions into the processor's physical memory (i.e. the array returned by `getMemory()`). Next, we initialize the processor's page table and registers. Finally, we invoke `run()`, which begins the fetch-execute cycle for the processor.

`run()` causes the processor to enter an infinite fetch-execute loop. This method should only be called after the registers and memory have been properly initialized. Each iteration of the loop does three things:

1. It attempts to run an instruction. This should be very familiar to students who have studied the generic 5-stage MIPS pipeline. Note that when an exception occurs, the pipline is aborted.
    a. The 32-bit instruction is fetched from memory, by reading the word of virtual memory pointed to by the PC register. Reading virtual memory can cause an exception.
    b. The instruction is decoded by looking at its 6-bit `op` field and looking up the meaning of the instruction in one of three tables.
    c. The instruction is executed, and data memory reads and writes occur. An exception can occur if an arithmetic error occurs, if the instruction is invalid, if the instruction was a syscall, or if a memory operand could not be accessed.
    d. The registers are modified to reflect the completion of the instruction.
2. If an exception occurred, handle it. The cause of the exception is written to the cause register, and if the exception involved a bad virtual address, this address is written to the bad virtual address register. If a delayed load is in progress, it is completed. Finally, the kernel's exception handler is invoked.
3. It advances the simulated clock (the clock, used to simulate interrupts, is discussed in the following section).

Note that from a user-level process's perspective, exceptions take place in the same way as if the program were executing on a bare machine; an exception handler is invoked to deal with the problem. However, from our perspective, the kernel's exception handler is actually called via a normal procedure call by the simulated processor.

The processor provides three methods we have not discussed yet: `makeAddress()`, `offsetFromAddress()`, and `pageFromAddress()`. These are utility procedures that help the kernel go between virtual addresses and virtual-page/offset pairs.

4.2. Address Translation

The simulated processor supports one of two address translation modes: linear page tables, or a software-managed TLB. While the former is simpler to program, the latter more closely corresponds to what current machines support.

In both cases, when translating an address, the processor breaks the 32-bit virtual address into a virtual page number (VPN) and a page offset. Since the processor's page size is 1KB, the offset is 10 bits wide and the VPN is 22 bits wide. The processor then translates the virtual page number into a translation entry.

Each translation entry (see the TranslationEntry class) contains six fields: a valid bit, a read-only bit, a used bit, a dirty bit, a 22-bit VPN, and a 22-bit physical page number (PPN). The valid bit and read-only bit are set by the kernel and read by the processor. The used and dirty bits are set by the processor, and read and cleared by the kernel.

4.2.1. Linear Page Tables

When in linear page table mode, the processor uses the VPN to index into an array of translation entries. This array is specified by calling `setPageTable()`. If, in translating a VPN, the VPN is greater than or equal to the length of the page table, or the VPN is within range but the corresponding translation entry's valid bit is clear, then a page fault occurs.

In general, each user process will have its own private page table. Thus, each process switch requires calling `setPageTable()`. On a real machine, the page table pointer would be stored in a special processor register.

4.2.2. Software-Managed TLB

When in TLB mode, the processor maintains a small array of translation entries that the kernel can read/write using `readTLBEntry()` and `writeTLBEntry()`. On each address translation, the processor searches the entire TLB for the first entry whose VPN matches.

5. User-Level Processes

Nachos runs each user program in its own private address space. Nachos can run any COFF MIPS binaries that meet a few restrictions. Most notably, the code must only make system calls that Nachos understands. Also, the code must not use any floating point instructions, because the Nachos MIPS simulator does not support coprocessors.

5.1. Loading COFF Binaries

COFF (Common Object File Format) binaries contain a lot of information, but very little of it is actually

relevent to Nachos programs. Further, Nachos provides a COFF loader class, nachos.machine.Coff, that abstracts away most of the details. But a few details are still important.

A COFF binary is broken into one or more *sections*. A section is a contiguous chunk of virtual memory, all the bytes of which have similar attributes (code vs. data, read-only vs. read-write, initialized vs. uninitialized). When Nachos loads a program, it creates a new processor, and then copies each section into the program's virtual memory, at some start address specified by the section. A COFF binary also specifies an initial value for the PC register. The kernel must initialize this register, as well as the stack pointer, and then instruct the processor to start executing the program.

The Coff constructor takes one argument, an OpenFile referring to the MIPS binary file. If there is any error parsing the headers of the specified binary, an EOFException is thrown. Note that if this constructor succeeds, the file belongs to the Coff object; it should not be closed or accessed anymore, except through Coff operations.

There are four Coff methods:

- `getNumSections()` returns the number of sections in this binary.
- `getSection()` takes a section number, between `0` and `getNumSections() - 1`, and returns a CoffSection object representing the section. This class is described below.
- `getEntryPoint()` returns the value with which to initialize the program counter.
- `close()` releases any resources allocated by the loader. This includes closing the file passed to the constructor.

The CoffSection class allows Nachos to access a single section within a COFF executable. Note that while the MIPS cross-compiler generates a variety of sections, the only important distinction to the Nachos kernel is that some sections are read-only (i.e. the program should never write to any byte in the section), while some sections are read-write (i.e. non-`const` data). There are four methods for accessing COFF sections:

- `getFirstVPN()` returns the first virtual page number occupied by the section.
- `getLength()` returns the number of pages occupied by the section. This section therefore occupies pages `getFirstVPN()` through `getFirstVPN() + getLength() - 1`. Sections should never overlap.
- `isReadOnly()` returns true if and only if the section is read-only (i.e. it only contains code or constant data).
- `loadPage()` reads a page of the section into main memory. It takes two arguments, the page within the section to load (in the range `0` through `getLength() - 1`) and the physical page of memory to write.

5.2. Starting a Process

The kernel starts a process in two steps. First, it calls `UserProcess.newUserProcess()` to instantiate a process of the appropriate class. This is necessary because the process class changes as more functionality is added to each process. Second, it calls `execute()` to load and execute the program, passing the name of the file containing the binary and an array of arguments.

`execute()` in turn takes two steps. It first loads the program into the process's address space by calling `load()`. It then forks a new thread, which initializes the processor's registers and address translation information and then calls `Machine.processor().run()` to start executing user code.

`load()` opens the executable's file, instantiates a COFF loader to process it, verifies that the sections are contiguously placed in virtual memory, verifies that the arguments will fit within a single page, calculates the size of the program in pages (including the stack and arguments), calls `loadSections()` to actually load the contents of each section, and finally writes the command line arguments to virtual memory.

`load()` lays out the program in virtual memory as follows: first, starting at virtual address 0, the sections of the executable occupy a contiguous region of virtual memory. Next comes the stack, the size of which is determined by the variable `stackPages`. Finally, one page is reserved for command line arguments (that `argv` array).

`loadSections()` allocates physical memory for the program and initializes its page table, and then loads sections to physical memory (though for the VM project, this loading is done lazily, delayed until pages are demanded). This is separated from the rest of `load()` because the loading mechanism depends on the details of the paging system.

In the code you are given, Nachos assumes that only a single process can exist at any given time. Therefore, `loadSections()` assumes that no one else is using physical memory, and it initializes its page table so as to map virtual memory addresses directly to physical memory addresses, without any translation (i.e. virtual address `n` maps to physical address `n`).

The method `initRegisters()` zeros out the processor's registers, and then initializes the program counter, the stack pointer, and the two argument registers (which hold `argc` and `argv`) with the values computed by `load()`. `initRegisters()` is called exactly once by the thread forked in `execute()`.

5.3. User Threads

User threads (that is, kernel threads that will be used to run user code) require additional state. Specifically, whenever a user thread starts running, it must restore the processor's registers, and possibly restore some address translation information as well. Right before a context switch, a user thread needs to save the processor's registers.

To accomplish this, there is a new thread class, UThread, that extends KThread. It is necessary to know which process, if any, the current thread belongs to. Therefore each UThread is bound to a single process.

UThread overrides `saveState()` and `restoreState()` from KThread so as to save/restore the additional information. These methods deal only with the user register set, and then direct the current process to deal with process-level state (i.e. address translation information). This separation makes it possible to allow multiple threads to run within a single process.

5.4. System Calls and Exception Handling

User programs invoke system calls by executing the MIPS `syscall` instruction, which causes the Nachos kernel exception handler to be invoked (with the cause register set to `Processor.exceptionSyscall`). The kernel must first tell the processor where the exception handler is by calling `Machine.processor().setExceptionHandler()`.

The default Kernel exception handler, `UserKernel.exceptionHandler()`, reads the value of the processor's cause register, determines the current process, and invokes `handleException` on the current process, passing the cause of the exception as an argument. Again, for a syscall, this value will be `Processor.exceptionSyscall`.

The `syscall` instruction indicates a system call is requested, but doesn't indicate which system call to perform. By convention, user programs place the value indicating the particular system call desried into MIPS register `r2` (the first return register, `v0`) before executing the `syscall` instruction. Arguments to the system call, when necessary, are passed in MIPS registers `r4` through `r7` (i.e. the argument registers, `a0 ... a3`), following the standard C procedure call convention. Function return values, including system call return values, are expected to be in register `r2` (`v0`) on return.

> **Note:** When accessing user memory from within the exception handler (or within Nachos in general), user-level addresses cannot be referenced directly. Recall that user-level processes execute in their own private address spaces, which the kernel cannot reference directly. Use `readVirtualMemory()`, `readVirtualMemoryString()`, and `writeVirtualMemory()` to make use of pointer arguments to syscalls.

Notes

1. The ThreadQueue object representing the ready queue is stored in the static variable `KThread.readyQueue`.