CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 4

OS Structure (Con't)
Modern Architecture

February 3th, 2014
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

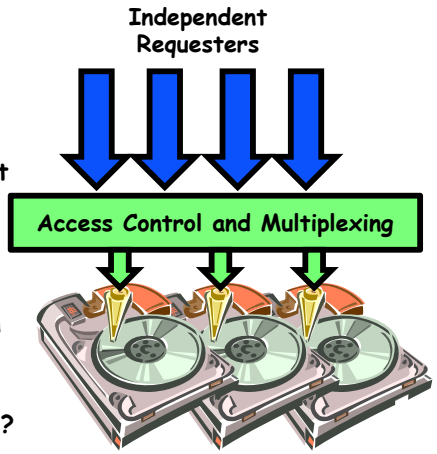
- OS Organizations (Con't): The Linux Kernel
- Processes, Threads, and Such

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: OS Resources – at the center of it all!

- What do modern OSs do?
  - Why all of these pieces running together?
  - Is this complexity necessary?
- Control of Resources
  - Access/No Access/ Partial Access
    - » Check every access to see if it is allowed
  - Resource Multiplexing
    - » When multiple valid requests occur at same time – how to multiplex access?
    - » What fraction of resource can requester get?
  - Performance Isolation
    - » Can requests from one entity prevent requests from another?
- What or Who is a requester???
  - Process? User? Public Key?
  - Think of this as a "Principle"



Recall: System Calls: Details

- Challenge: Interaction Despite Isolation
  - How to isolate processes and their resources…
    - » While permitting them to request help from the kernel
    - » Processes interact while maintaining policies such as security, QoS, etc
  - Letting processes interact with one another in a controlled way
    - » Through messages, shared memory, etc
- Enter the System Call interface
  - Layer between the hardware and user-space processes
  - Programming interface to the services provided by the OS
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than directly
  - Get at system calls by linking with libraries in glibc

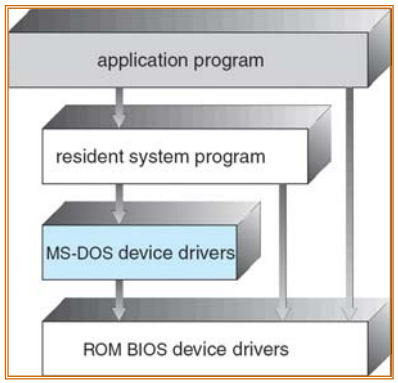
- Three most common APIs are:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Java API for the Java virtual machine (JVM)

Operating Systems Structure (What is the organizational Principle?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since both change
- Organizational Properties
  - 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
  - ExoKernel
    - » Separate protection from management of resources
  - Cell-based OS
    - » Two-level scheduling of resources

Simple Structure

- All aspects of the OS linked together in one binary
  - APIs not carefully designed (and/or lots of global variables)
  - Interfaces and levels of functionality not well separated
  - No address protection
- Example: MS-DOS
  - provide the most functionality in the least space
  - Made sense in early days of personal computers with limited processors (e.g. 6502)
- Advantages?
  - Low memory footprint
- Disadvantages?
  - Very fragile, no enforcement of structure/boundaries

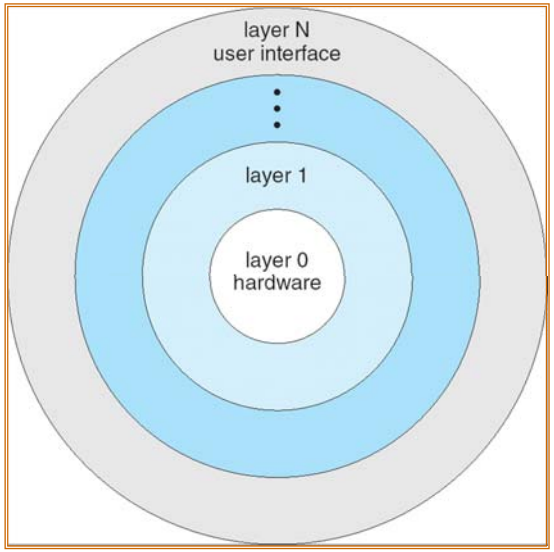


- What about Language enforcement? (Microsoft Singularity?)

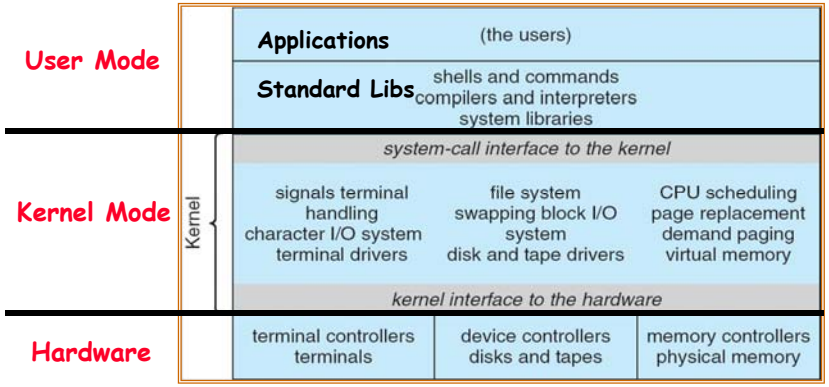
Layered Structure

- 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!
- Can utilize hardware enforcement
  - x86 processor: 4 "rings"
  - Call gates

Layered Operating System



Monolithic Structure: UNIX System Structure



- Two-Layered Structure: User vs Kernel
  - All code representing protection and management of resources placed in same address space
  - Compromise of one component can compromise whole OS
- Clear division of labor?
  - The producer of the OS and the User of the OS

Microkernel Structure

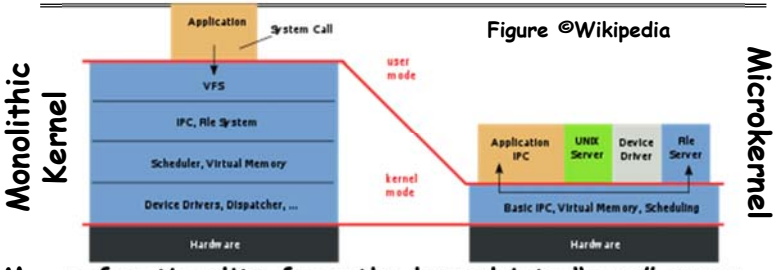
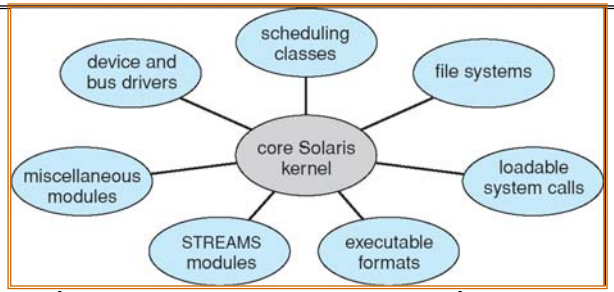


Figure ©Wikipedia

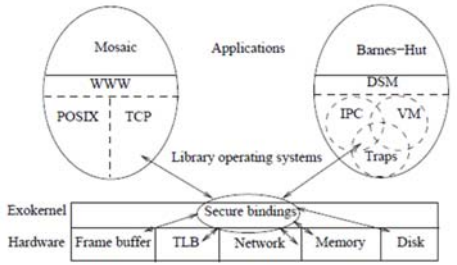
- Moves functionality from the kernel into "*user*" space
  - Small core OS running at kernel level
  - OS Services built from many independent user-level processes
  - Communication between modules with message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port OS to new architectures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

Modules-based Structure



- Most modern operating systems implement modules
  - Uses object-oriented approach
    - » careful API design/Few if any global variables
- 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
  - May or may not utilize hardware enforcement

ExoKernel: Separate Protection from Management

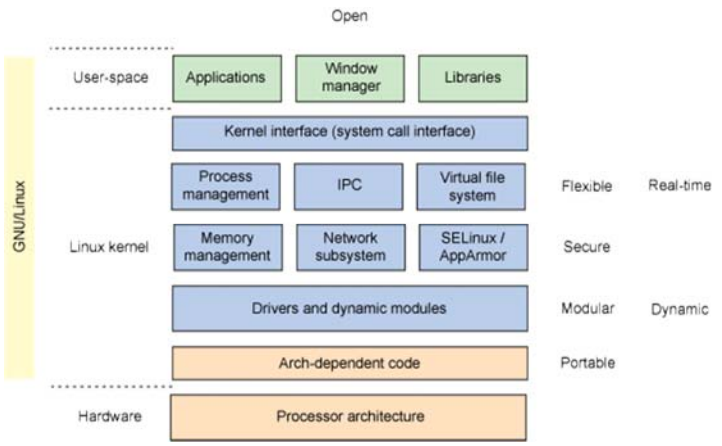


- Thin layer exports hardware resources directly to users
  - As little abstraction as possible
  - Secure Protection and Multiplexing of resources
- LibraryOS: traditional OS functionality at User-Level
  - Customize resource management for every application
  - Is this a practical approach?
- Very low-level abstraction layer
  - Need extremely specialized skills to develop LibraryOS

Administrivia

- Please sign up for groups by tonight!
  - You will need them starting on Wednesday, and we need time to get them set up
  - Remember 4 or 5 members to a group
- Schedule for Labs 1-4
  - Three Weeks for each lab ⇒ Three checkpoints
  - Checkpoints Due on Thursday @9pm
  - Meeting with TAs on Friday
- Get your final VMWare licenses!
  - Log into inst.eecs.berkeley.edu with class account
  - Should have email with sufficient information to get your license

Administrivia: Linux Structure



Administrivia: Layout of Linux Sources

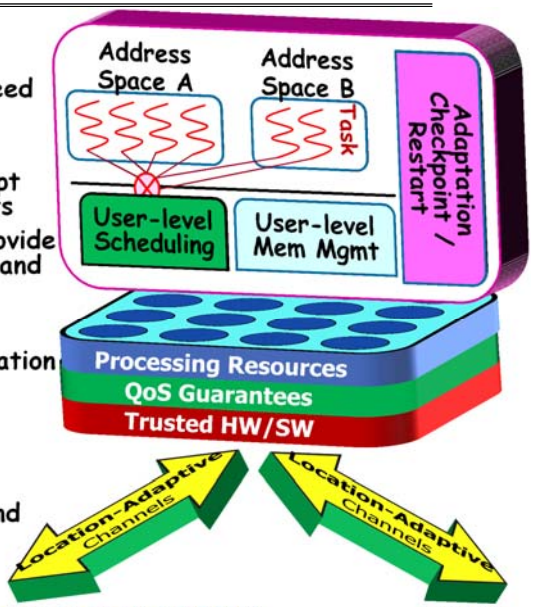
- Layout of basic linux sources:

```
kubitron@kubi(16)% ls
arch/          drivers/   Kbuild       modules.builtin  samples/    usr/
block/         firmware/  kernel/      modules.order    scripts/    virt/
COPYING        fs/        lib/         Module.symvers   security/   vmlinux*
CREDITS        include/   MAINTAINERS  net/             sound/      vmlinux.o
crypto/        init/      Makefile     README           System.map
Documentation/ ipc/       mm/          REPORTING-BUGS   tools/
kubitron@kubi(17)%
```

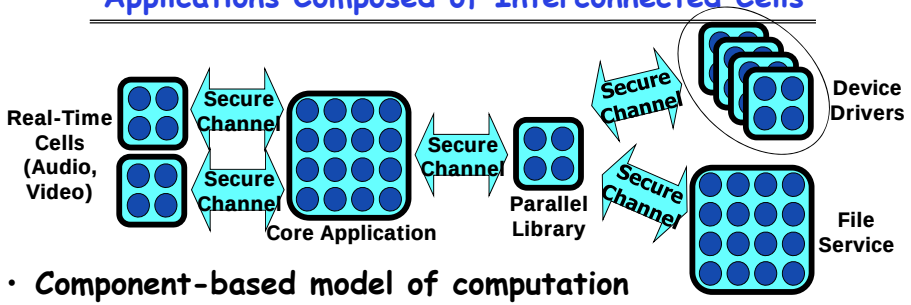
- Specific Directories:
  - arch: Architecture-specific source
  - block: Block I/O layer
  - crypto: Crypto API
  - Documentation: Kernel source Documentation
  - drivers: Device drivers
  - firmware: Device firmware needed to use certain drivers
  - fs: The VFS and individual filesystems
  - include: Kernel headers
  - init: Kernel boot and initialization
  - ipc: Interprocess Communication code
  - kernel: Core subsystems, such as the scheduler
  - lib: Helper routines
  - mm: Memory management subsystem and the vm
  - net: Networking subsystem
  - samples: Sample, demonstrative code
  - scripts: Scripts used to build the kernel
  - security: Linux Security Module
  - sound: Sound subsystem
  - usr: Early user-space code (called initramfs)
  - tools: Tools helpful for developing Linux
  - virt: Virtualization infrastructure

Resource-Oriented Model: the Cell

- Cell Properties:
  - A user-level software component, with guaranteed resources
  - Explicit security context
  - Knowledge of how to adapt itself to new environments
  - Checkpoint/restart to provide fault tolerance, mobility and adaptation
- Execution Environment:
  - Explicitly parallel computation
  - Resource Guarantees
  - Trusted computing base
  - Secure channels (intra/interchip) with ability to suspend and restart during migration



Applications Composed of Interconnected Cells

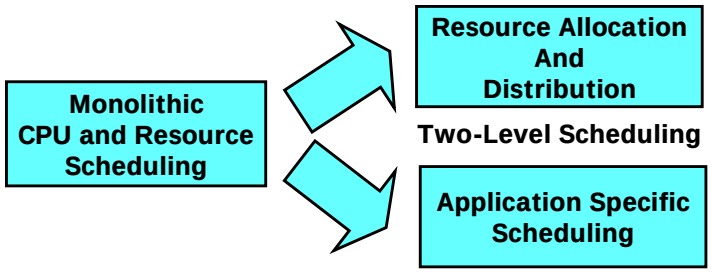


- Component-based model of computation
  - Applications consist of interacting components
  - Components may be local or remote
- Communication defines Security Model
  - Channels are points at which data may be compromised
- Naming process for initiating endpoints
  - Need to find consistent version of library code (within cell)
  - Need to find compatible remote services

Complete Focus on Resources

- What might we want to guarantee?
  - Examples:
    - » Guarantees of BW (say data committed to Cloud Storage)
    - » Guarantees of Requests/Unit time (DB service)
    - » Guarantees of Latency to Response (Deadline scheduling)
    - » Guarantees of maximum time to Durability in cloud
    - » Guarantees of total energy/battery power available to Cell
- What level of guarantee?
  - Firm Guarantee (Better than existing systems)
    - » With high confidence (specified), Maximum deviation, etc.
- What does it mean to have guaranteed resources?
  - A Service Level Agreement (SLA)?
  - Something else?
- "Impedance-mismatch" problem
  - The SLA guarantees properties that programmer/user wants
  - The *resources* required to satisfy SLA are not things that programmer/user really understands

Two Level Scheduling



- Split monolithic scheduling into two pieces:
  - Course-Grained Resource Allocation and Distribution to Cells
    - » Chunks of resources (CPUs, Memory Bandwidth, QoS to Services)
    - » Ultimately a hierarchical process negotiated with service providers
  - Fine-Grained (User-Level) Application-Specific Scheduling
    - » Applications allowed to utilize their resources in any way they see fit
    - » Performance Isolation: Other components of the system cannot interfere with Cells use of resources

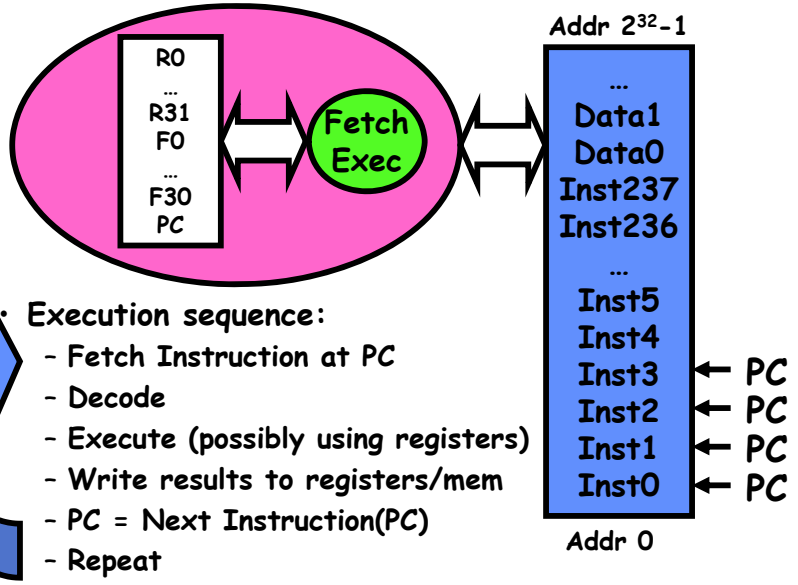
Concurrency

- "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)
- ManyCore ⇒ Multiprogramming, right?

The Basic Problem of Concurrency

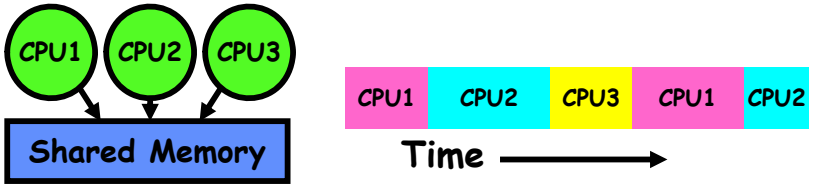
- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these 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?

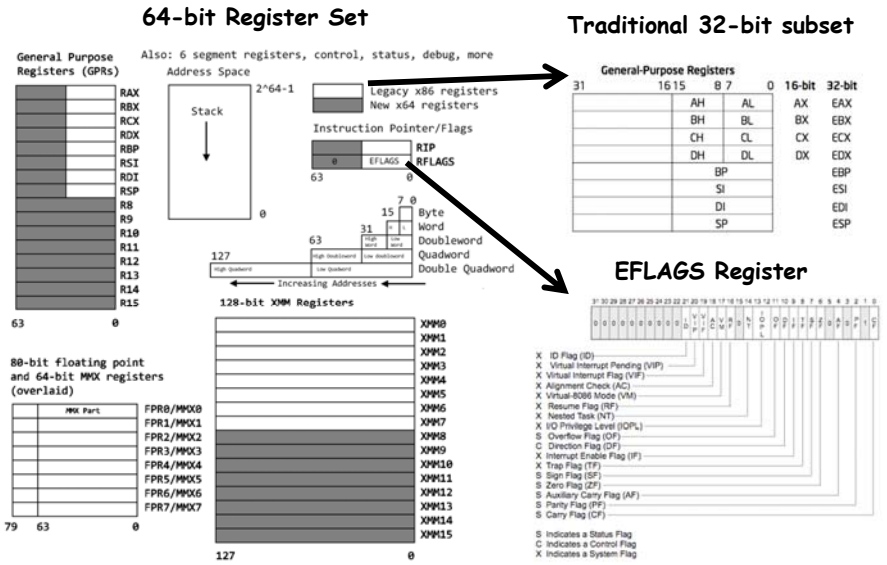


- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
  - Call result a "Thread" for now…
- How switch from 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

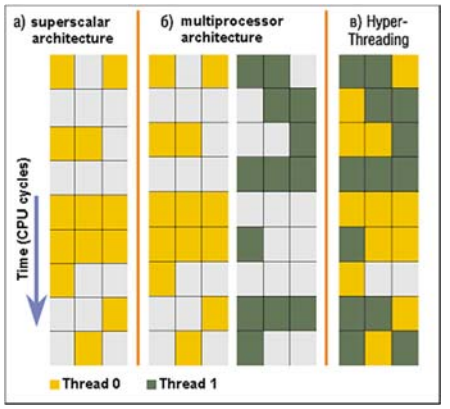
- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads 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)

What needs to be saved in Modern X86?



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



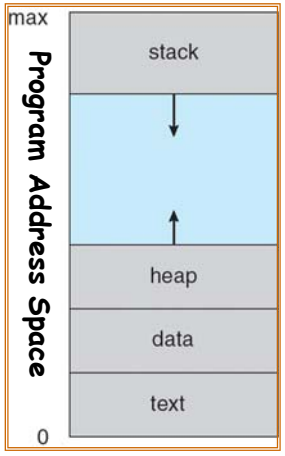
- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

How to protect threads from one another?

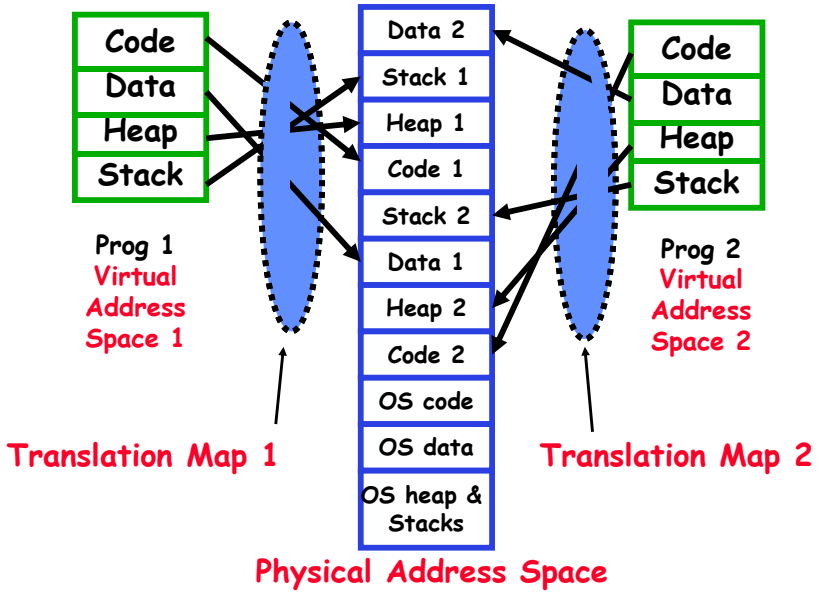
- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Protection of Access to Processor: Preemptive switching from task to task
     - » Use of timer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

Recall from CS162: Program's Address Space

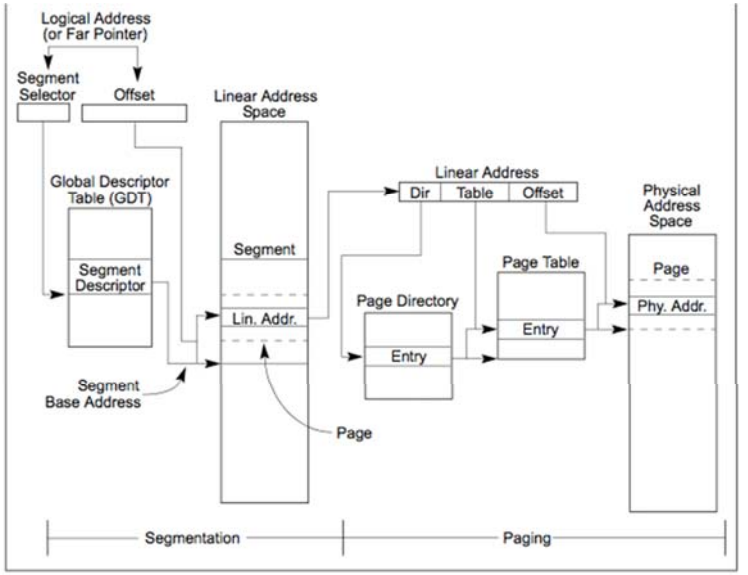
- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes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



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



X86 Memory model with segmentation



The Six x86 Segment Registers

- CS - Code Segment
- SS - Stack Segment
  - "Stack segments are data segments which must be read/write segments. Loading the SS register with a segment selector for a nonwritable data segment generates a general-protection exception (#GP)"
- DS - Data Segment
- ES/FS/GS - Extra (usually data) segment registers
  - FS and GS used for thread-local storage/by glibc
- The "hidden part" is like a cache so that segment descriptor info doesn't have to be looked up each time

Visible Part	Hidden Part	
Segment Selector	Base Address, Limit, Access Information	CS
		SS
		DS
		ES
		FS
		GS

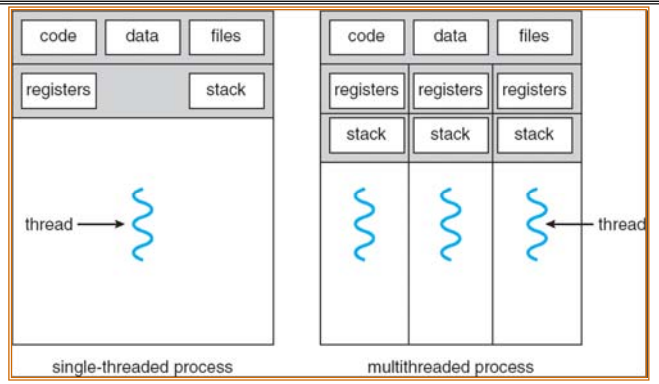
UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Originally: a single, sequential stream of execution in its *own* address space
  - Modern Process: multiple threads in same address space!
- Two parts:
  - Sequential Program Execution Streams
    - » Code executed as one or more *sequential* stream of execution (threads)
    - » Each thread includes its own state of CPU registers
    - » Threads either multiplexed in software (OS) or hardware (simultaneous multithrading/hyperthreading)
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- This is a virtual machine abstraction
  - Some might say that the only point of an OS is to support a clean Process abstraction

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
- Linux confuses this model a bit:
  - Processes and Threads are "the same"
  - Really means: Threads are managed separately and can share a variety of resources (such as address spaces)
  - Threads related to one another in fashion similar to Processes with Threads within

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?

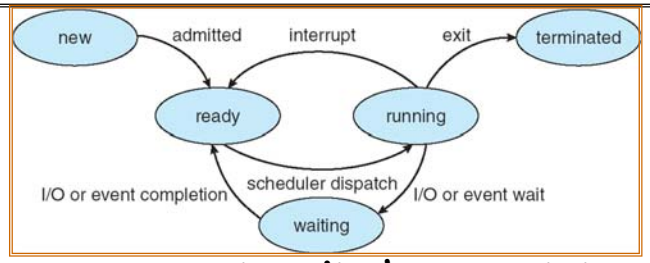
Process comprises

- What is in a process?
  - an address space – usually protected and virtual – mapped into memory
  - the code for the running program
  - the data for the running program
  - an execution stack and stack pointer (SP); also heap
  - the program counter (PC)
  - a set of processor registers – general purpose and status
  - a set of system resources
    - » files, network connections, pipes, …
    - » privileges, (human) user association, …
    - » Personalities (linux)
  - …

Process – starting and ending

- Processes are created …
  - When the system boots
  - By the actions of another process (more later)
  - By the actions of a user
  - By the actions of a batch manager
- Processes terminate …
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by the actions a user or a process

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
    Can be *Interruptible* or *Non-Interruptible*
  - terminated: The process has finished execution
    Stays as *Zombie* until relays result to parent

Preview: System-Level Control of x86

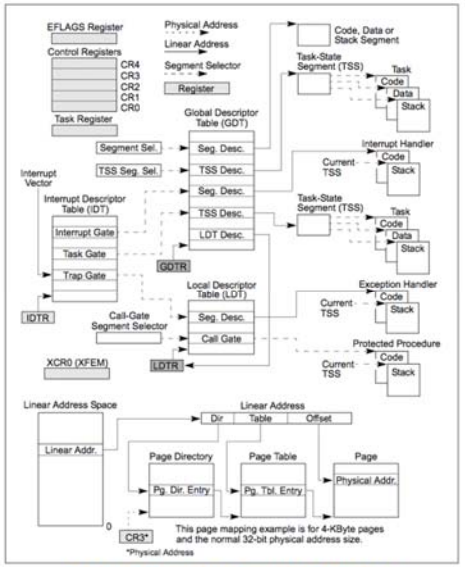
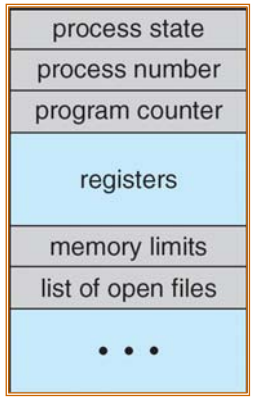


Figure 2-1. IA-32 System-Level Registers and Data Structures

- Full support for Process Abstraction involves a lot of system-level state
  - This is state that can only be accessed in kernel mode!
  - We will be talking about a number of these pieces as we go through the term…
- There is a tradeoff between amount of system state and cost of switching from thread to thread!

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

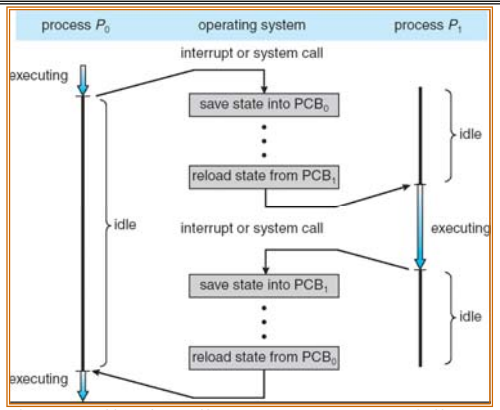


Process Control Block

Processes in the OS – PCB

- Typical PCB contains:
  - execution state
  - PC, SP & processor registers – stored when process is not in *running* state
  - memory management info
  - Privileges and owner info
  - scheduling priority
  - resource info
  - accounting info

CPU Switch From Process to Process

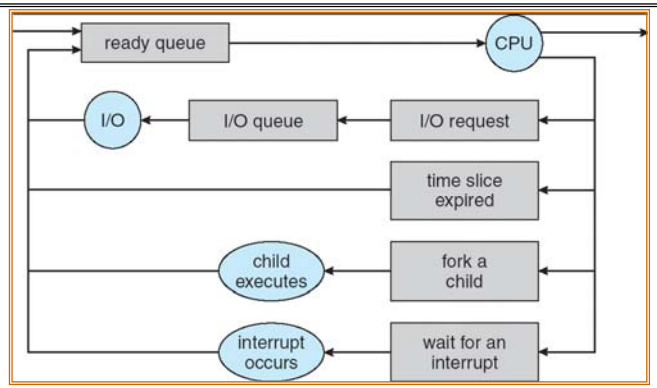


- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Processes – State Queues

- The OS maintains a collection of *process state* queues
  - typically one queue for each state – e.g., ready, waiting, …
  - each PCB is put onto a queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- Process state and the queues change in response to events – interrupts, traps

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)

Processes – Privileges

- Users are given privileges by the system administrator
- Privileges determine what *rights* a user has for an *object*.
  - Unix/Linux – Read|Write|eXecute by user, group and "other" (i.e., "world")
  - WinNT – *A*ccess *C*ontrol *L*ist
- Processes "inherit" privileges from user

Linux Kernel Implementation

- Kernel may execute in either *Process context* or *Interrupt context*
- In *Process context*, kernel has access to
  - Virtual memory, files, other process resources
  - May sleep, take page faults, etc., on behalf of process
- In *Interrupt context*, no assumption about what process was executing (if any)
  - No access to virtual memory, files, resources
  - May not sleep, take page faults, etc.

System Calls for Creating Processes

- Begins life with the fork() system call
  - Creates new process by duplicating an existing one
  - Duplicate: memory, file descriptors, etc
  - The process that calls fork() is the parent, wheres the new process is the child.
- After creation, the parent resumes execution and the child starts execution at same place:
  - Where the call to fork() returns
  - i.e. fork() returns from the kernel twice!
- exit() system call:
  - Terminates the process and frees all its resources
  - A parent process can inquire about status of a terminated child via one of the wait system calls
  - Exiting process placed into special zombie state until parent calls wait or waitpid()

Process Creation – Unix & Linux

- Create a new (child) process – `fork();`
  - Allocates new PCB
  - Clones the calling process (almost exactly)
    - » Copy of parent process address space
    - » Copies resources in kernel (e.g. files)
  - Places new PCB on *Ready queue*
  - Return from `fork()` call
    - » 0 for child
    - » child PID for parent
- Copy-on-Write optimization
  - Copy parent address space ⇒ copy page tables
  - All pages marked as read only
  - Actual pages copied only on write

Example of *fork( )*

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s sees PID of %d\n",
           name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. My child is %d\n",
              name, child_pid);
    return 0;
  }
}
```

```
% ./forktest
Child of forktest sees PID of 0
I am the parent forktest. My child is 486
```

Summary

- Many organizations for Operating Systems
  - All oriented around Resources
  - Common organizations: Monolithic, MicroKernel, ExoKernel
- 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