CS162
Operating Systems and Systems Programming
Lecture 3

Processes (con't), Fork, Introduction to I/O

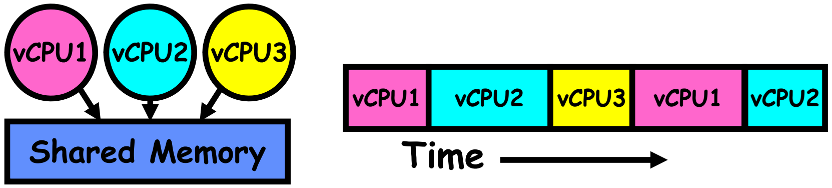
January 29th, 2019
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: Four fundamental OS concepts

- **Thread**
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- **Address Space** w/ translation
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- **Process**
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- **Dual Mode** operation/protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

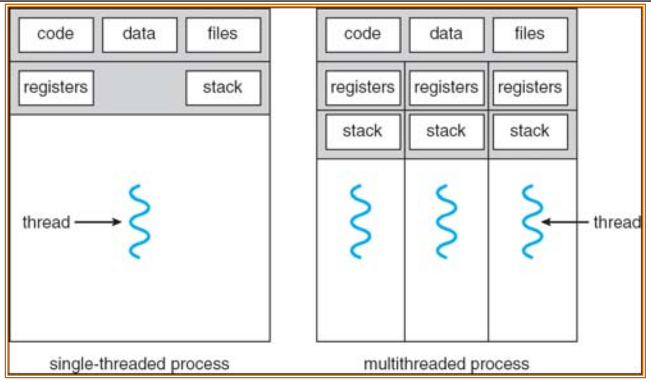
---

Recall: give the illusion of multiple processors?



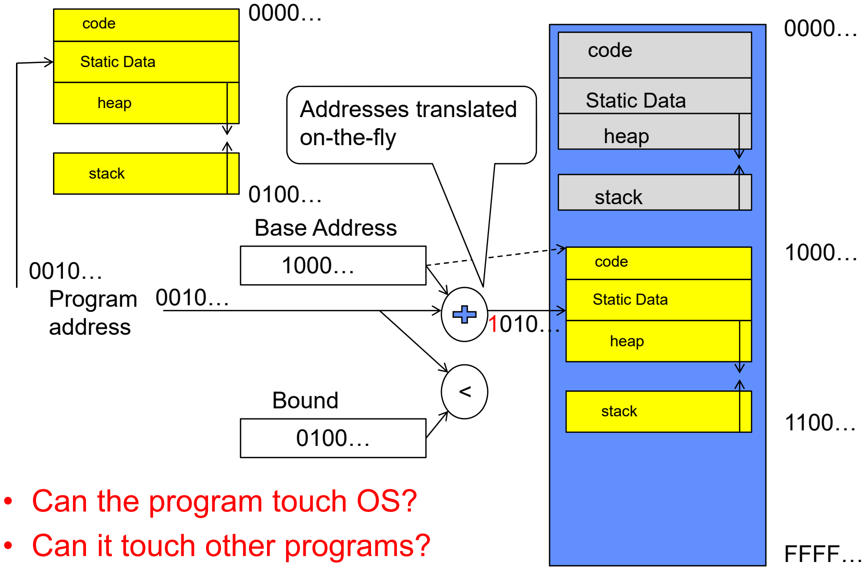
- Assume a single processor. How do we provide the *illusion* of multiple processors?
  - Multiplex in time!
  - Multiple "virtual CPUs"
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one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

Recall: Single and Multithreaded Processes



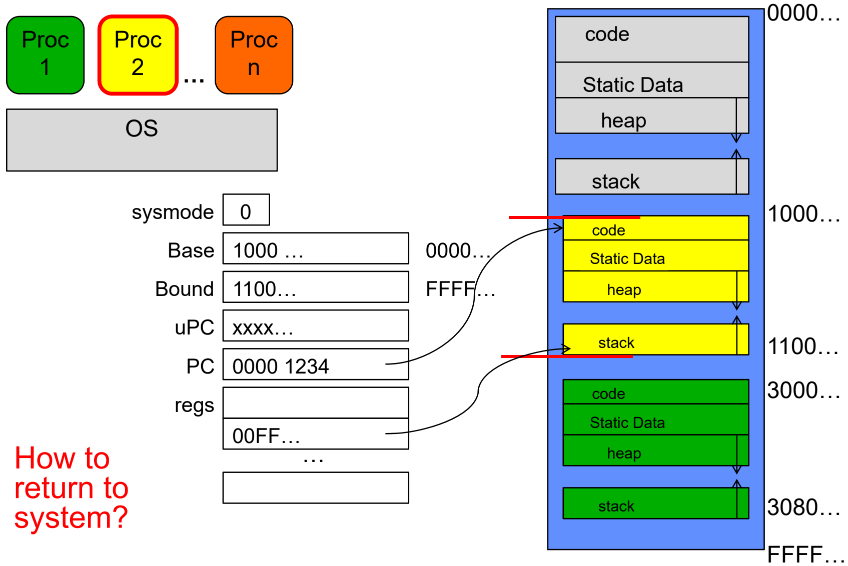
- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Recall: Simple address translation with Base and Bound



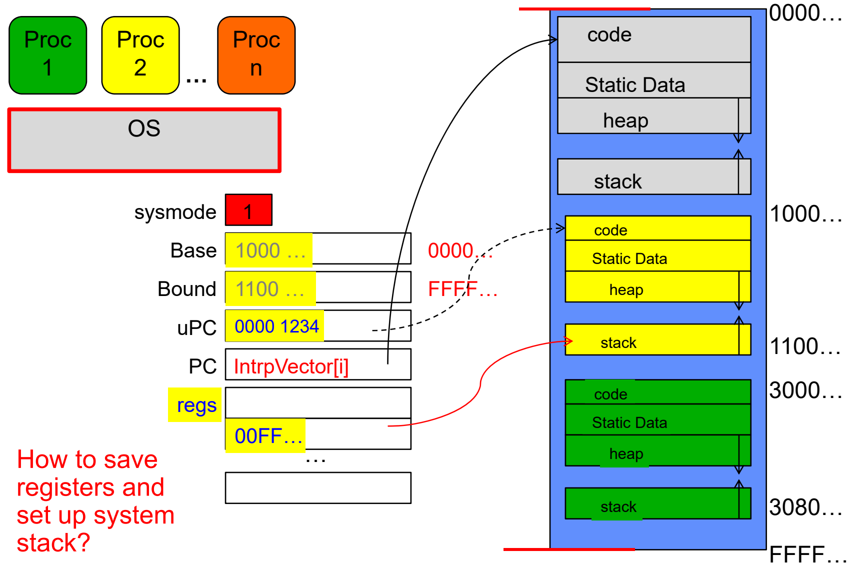
- Can the program touch OS?
- Can it touch other programs?

Simple B&B: User => Kernel



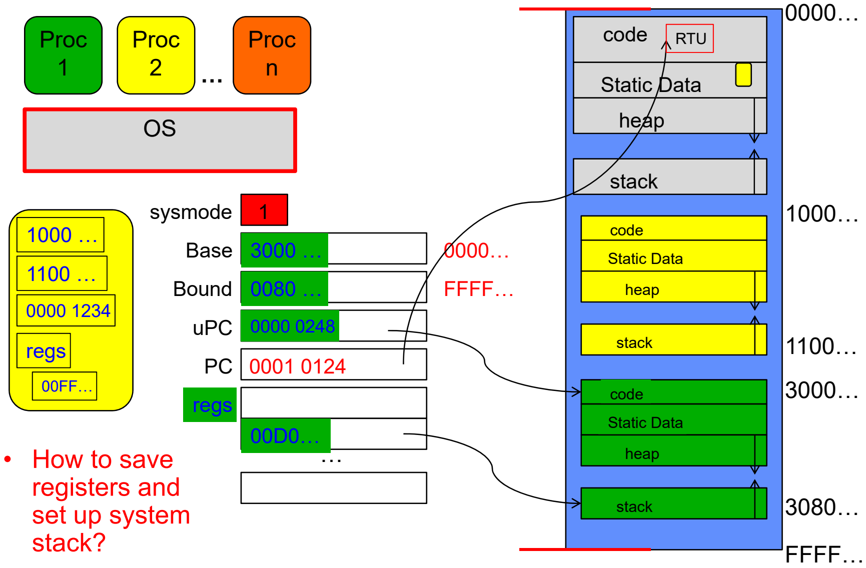
- How to return to system?

Simple B&B: Interrupt



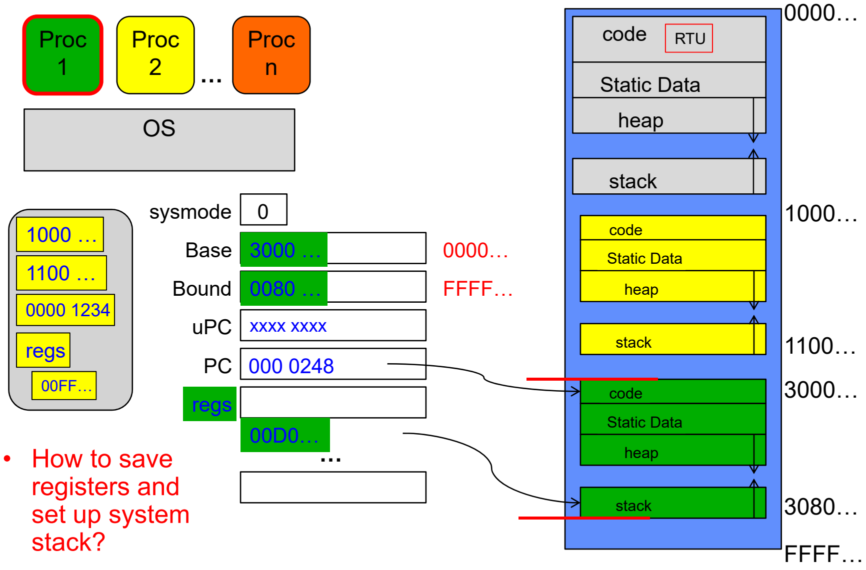
- How to save registers and set up system stack?

Simple B&B: Switch User Process



- How to save registers and set up system stack?

Simple B&B: “resume”



- How to save registers and set up system stack?

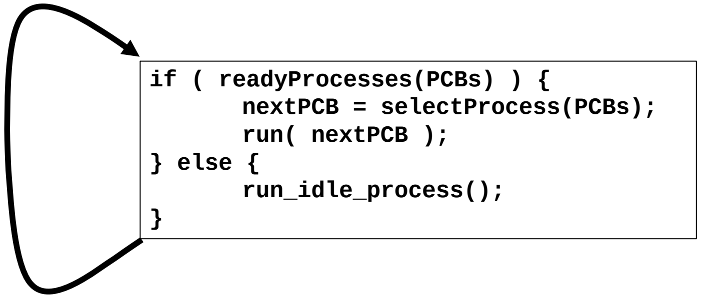
Running Many Programs

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other
- Questions ???
  - How do we represent user processes in the OS?
  - How do we decide which user process to run?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren’t we wasting are lot of memory?

Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

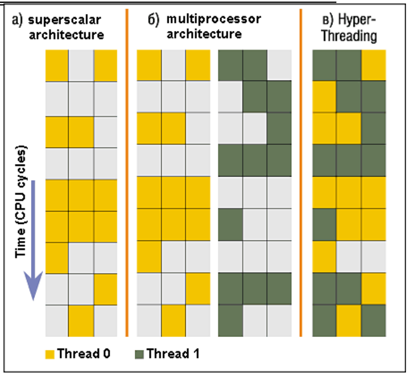
Scheduler



```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

Simultaneous MultiThreading/Hyperthreading

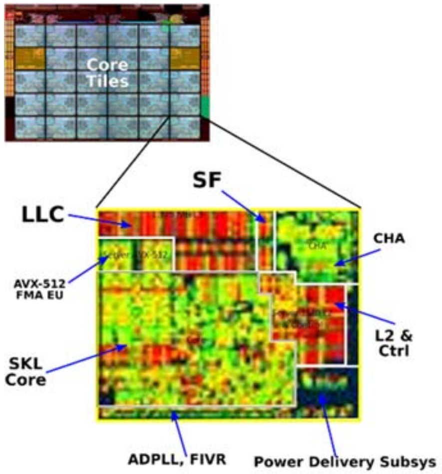


Colored blocks show instructions executed

- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original technique called "Simultaneous Multithreading"
  - http://www.cs.washington.edu/research/smt/index.html
  - SPARC, Pentium 4/Xeon ("Hyperthreading"), Power 5

Also Recall: The World Is Parallel

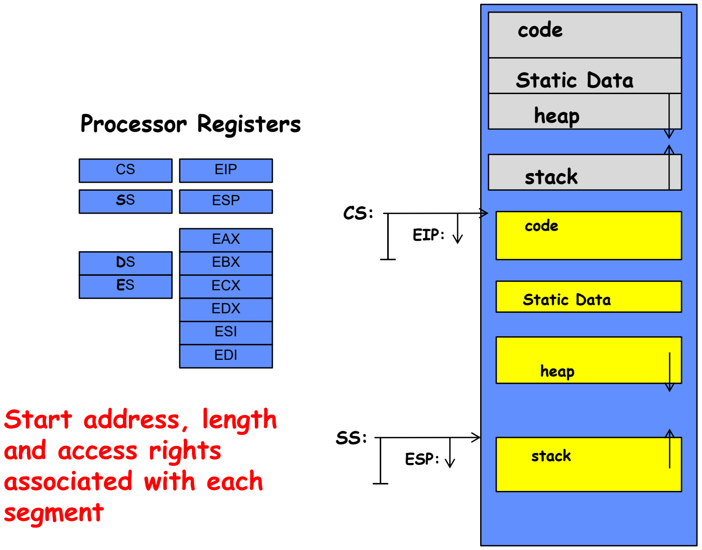
- Intel Skylake (2017)
  - 28 Cores
  - Each core has two hyperthreads!
  - So: 54 Program Counters(PCs)
- Scheduling here means:
  - Pick which core
  - Pick which thread
- Space of possible scheduling much more interesting
  - Can afford to dedicate certain cores to housekeeping tasks
  - Or, can devote cores to services (e.g. Filesystem)



Is Branch and Bound a Good-Enough Protection Mechanism?

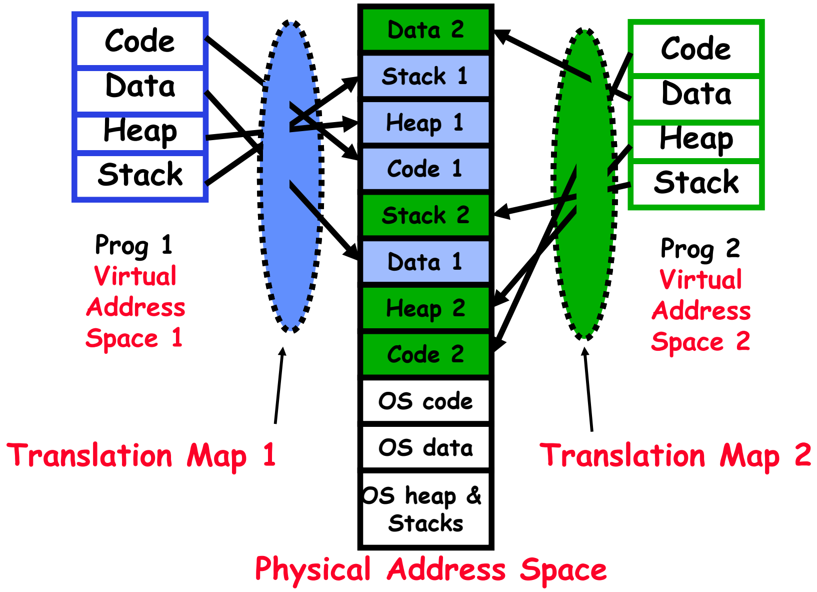
- NO: Too simplistic for real systems
- Inflexible/Wasteful:
  - Must dedicate physical memory for *potential* future use
  - (Think stack and heap!)
- Fragmentation:
  - Kernel has to somehow fit whole processes into contiguous block of memory
  - After a while, memory becomes fragmented!
- Sharing:
  - Very hard to share any data between Processes or between Process and Kernel
  - Need to communicate indirectly through the kernel…

Better: x86 – segments and stacks



Start address, length and access rights associated with each segment

Alternative: Address Mapping



Administrivia: Getting started

- Kubiatowicz Office Hours:
  - 1-2pm, Monday & Thursday
- Homework 0 Due Today!
  - Get familiar with the cs162 tools
  - configure your VM, submit via git
  - Practice finding out information:
    - » How to use GDB? How to understand output of unix tools?
    - » We don't assume that you already know everything!
    - » Learn to use "man" (command line), "help" (in gdb, etc), google
- Should be going to sections now – Important information there
  - Any section will do until groups assigned
- THIS Friday is Drop Deadline! HARD TO DROP LATER!
  - If you know you are going to drop, please do so to leave room for others on waitlist!

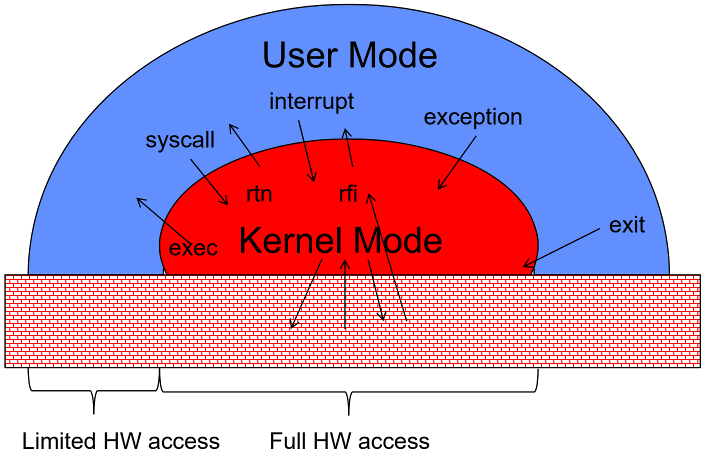
Administrivia (Con't)

- Group sign up via autograder form next week
  - Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA
  - Remember: Your TA needs to see you in section!
- Midterm 1 conflicts
  - We will handle these conflicts after have final class roster
  - I know about one problem with Midterm 1 scheduling, and it can be dealt with. Have I missed any others?
  - Watch for queries by HeadTA to collect information

Recall: 3 types of Kernel Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but "outside" the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - eg. Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

Recall: User/Kernel (Privileged) Mode



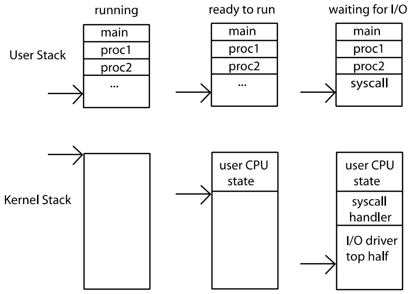
Implementing Safe Kernel Mode Transfers

- Important aspects:
  - Controlled transfer into kernel (e.g., syscall table)
  - Separate kernel stack
- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself

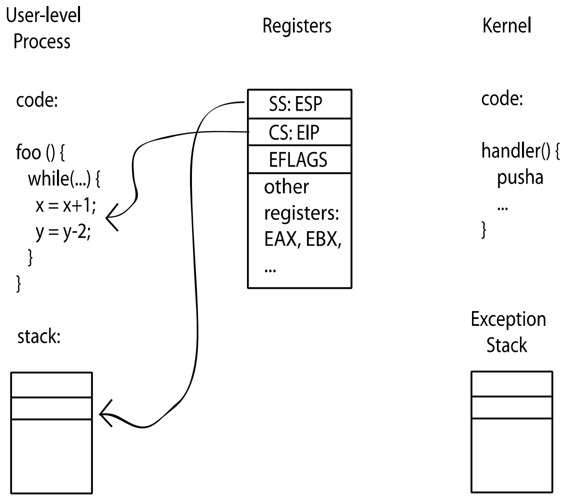
---

Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)



Before



User-level Process

code:

```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

stack:

Registers

SS: ESP

CS: EIP

EFLAGS

other registers: EAX, EBX, ...

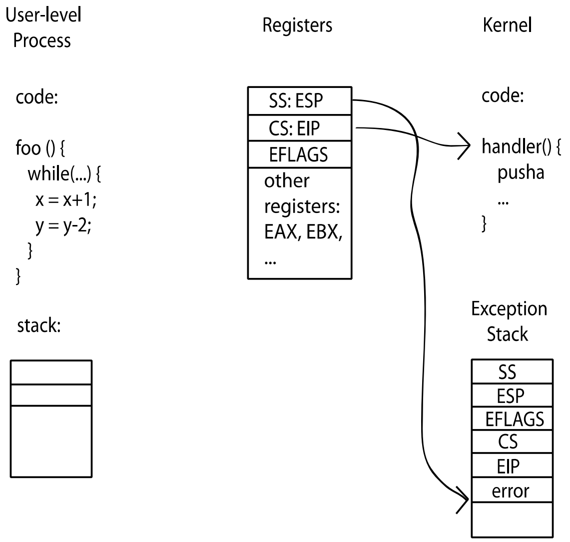
Kernel

code:

```
handler() {
  pusha
  ...
}
```

Exception Stack

During



```
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

```
handler() {
  pusha
  ...
}
```

Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?

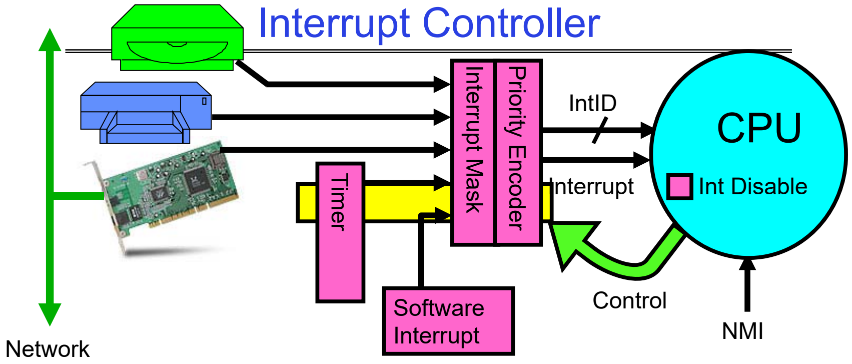
- Interrupt Handler invoked with interrupts ‘disabled’
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - » wake up an existing OS thread

Hardware support: Interrupt Control

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall

- HW may have multiple levels of interrupts
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    - » e.g., kernel segmentation fault
    - » Also: Power about to fail!

Interrupt Controller

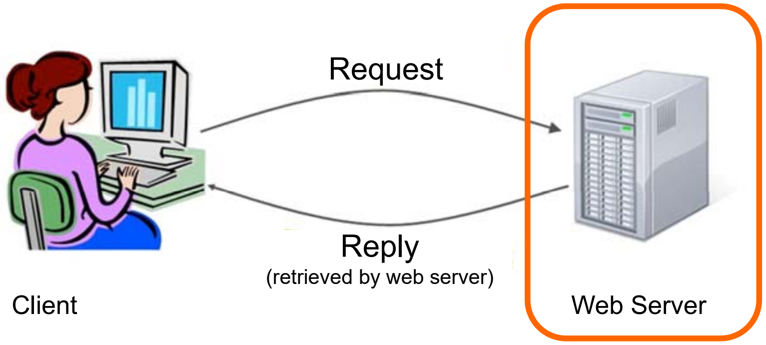


- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU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

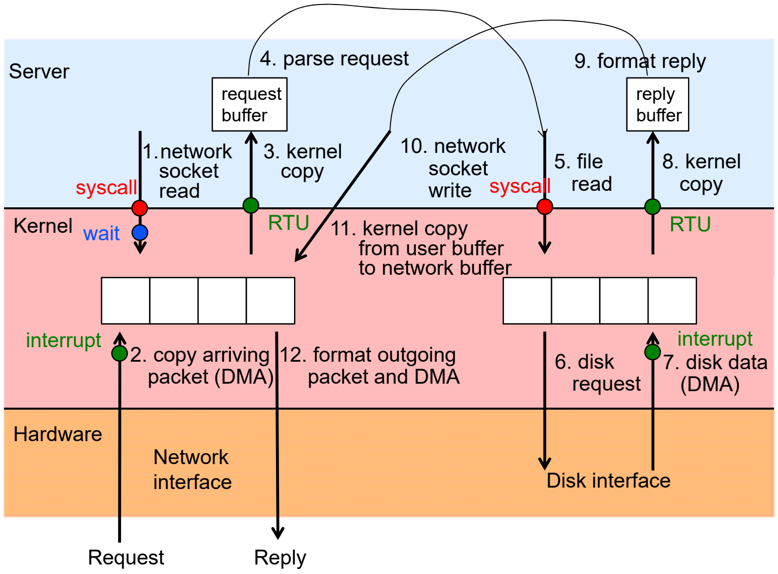
How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change:
    - » Program counter
    - » Stack pointer
    - » Memory protection
    - » Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Putting it together: web server



Putting it together: web server



Can a process create a process ?

- Yes! Unique identity of process is the “process ID” (or PID)
- **fork()** system call creates a *copy* of current process with a new PID
- Return value from **fork()**: integer
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running in new Child process
  - When < 0:
    - » Error! Must handle somehow
    - » Running in original process
- State of original process duplicated in *both* Parent and Child!
  - Memory, File Descriptors (next topic), etc…

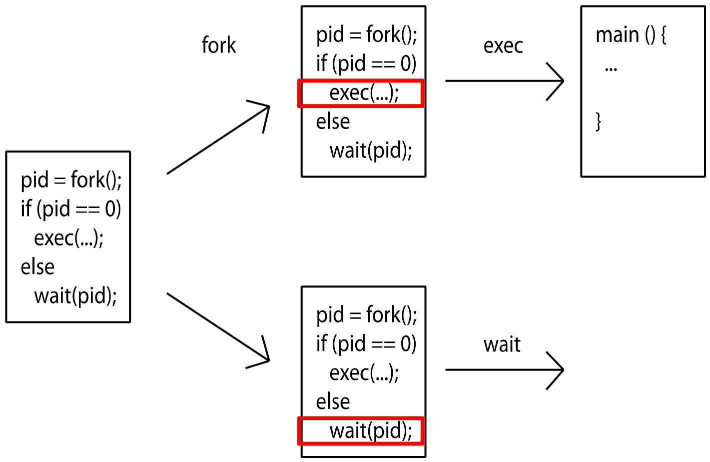
Create Process: fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
  pid_t cpid, mypid;

  pid_t pid = getpid();            /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {         /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

UNIX Process Management

- UNIX `fork` – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX `exec` – system call to *change the program* being run by the current process
- UNIX `wait` – system call to wait for a process to finish
- UNIX `signal` – system call to send a notification to another process
- UNIX man pages: `fork`(2), `exec`(3), `wait`(2), `signal`(3)

UNIX Process Management



```
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```

fork

```
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```

exec

```
main () {
  ...

}
```

```
pid = fork();
if (pid == 0)
  exec(...);
else
  wait(pid);
```

wait

Parent wait for child: fork2.c

```
int status;
pid_t = tcpid;
…
cpid = fork();
if (cpid > 0) {               /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d(%d)\n", mypid, tcpid, status);
}  else if (cpid == 0) {      /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
}
…
```

Process Races: fork3.c

```
int i;
cpid = fork();
if (cpid > 0) {
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    for (i=0; i<10; i++) {
      printf("[%d] parent: %d\n", mypid, i);
      // sleep(1);
    }
  }  else if (cpid == 0) {
    mypid = getpid();
    printf("[%d] child\n", mypid);
    for (i=0; i>-10; i--) {
      printf("[%d] child: %d\n", mypid, i);
      // sleep(1);
    }
  }
```

- Question: What does this program print?
- Does it change if you add in one of the sleep() statements?

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
- Example: to compile a C program

  cc –c sourcefile1.c

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o

  ./program

HW1

Signals – infloop.c

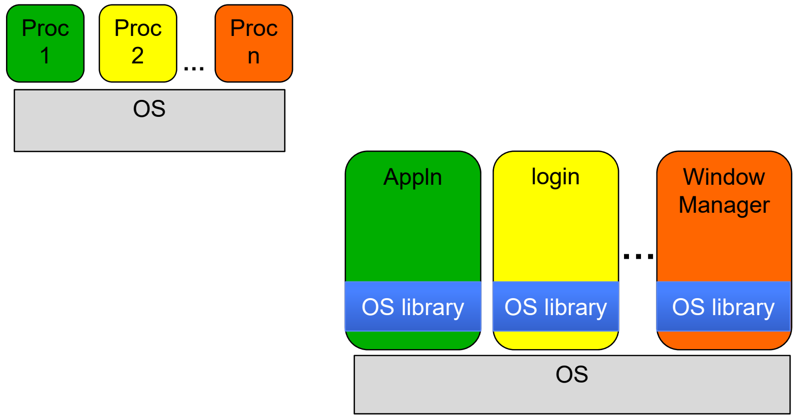
```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
  printf("Caught signal %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

How Does the Kernel Provide Services?

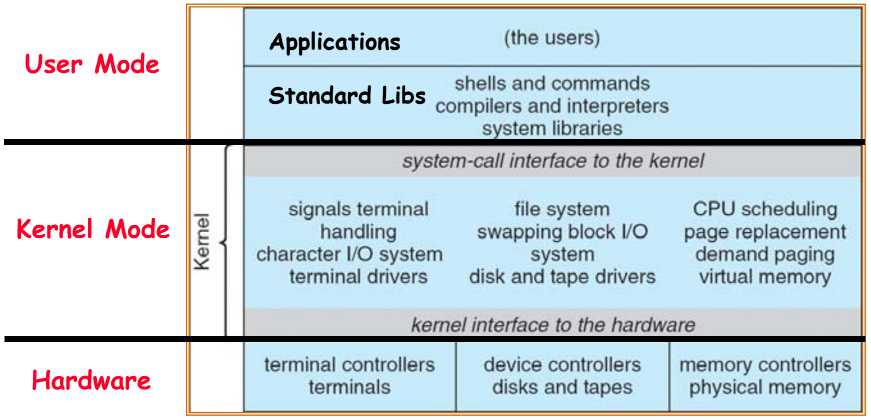
- You said that applications request services from the operating system via syscall, but …
- I've been writing all sort of useful applications and I never ever saw a "syscall" !!!

- That's right.
- It was buried in the programming language runtime library (e.g., libc.a)
- … Layering

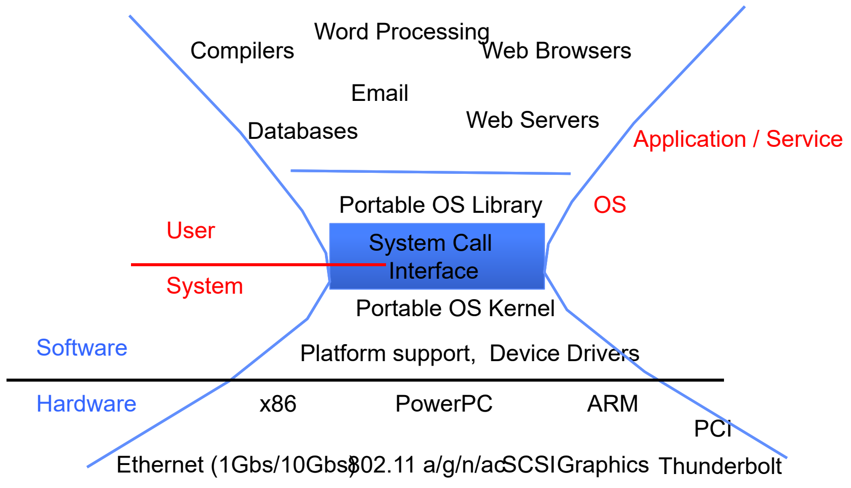
OS Run-Time Library



Recall: UNIX System Structure



A Kind of Narrow Waist



Summary

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Interrupts
  - Hardware mechanism for regaining control from user
  - Notification that events have occurred
  - User-level equivalent: Signals
- Native control of Process
  - Fork, Exec, Wait, Signal