CS 153
Design of Operating Systems

Winter 2016

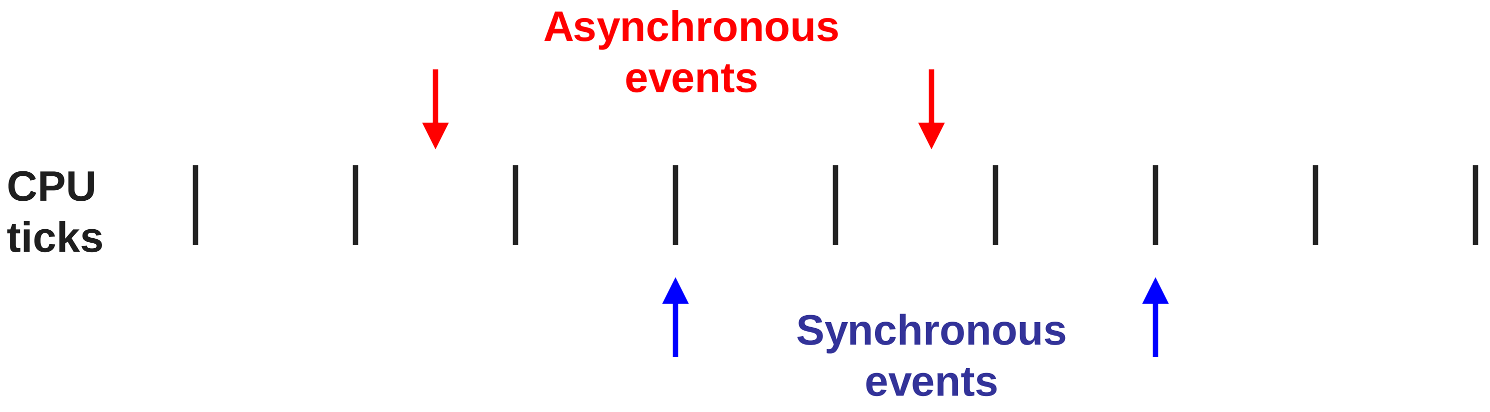
Lecture 3: Intro and Architectural Support for Operating Systems

Administrivia

- Project group signup linked in the lab webpage
- Office hours posted on my homepage as well
- Project 1 out!

Categorizing Events

- Two *kinds* of events: synchronous and asynchronous
- Sync events are caused by executing instructions
  - Example?
- Async events are caused by an external event
  - Example?



Categorizing Events

- Two *kinds* of events: synchronous and asynchronous
  - Sync events are caused by executing instructions
  - Async events are caused by an external event

- Two *reasons* for events: unexpected and deliberate
- Unexpected events are, well, unexpected
  - Example?
- Deliberate events are scheduled by OS or application
  - Why would this be useful?

Categorizing Events

- This gives us a convenient table:

	Unexpected	Deliberate
Synchronous	fault	software interrupt (syscall trap)
Asynchronous	interrupt	Asynchronous system trap (AST)

  - Terms may be used slightly differently by various OSes, CPU architectures…
    - » e.g., exceptions include fault and software interrupt
  - Will cover faults, system calls, and interrupts next

Faults

- Hardware detects and reports "exceptional" conditions
  - Page fault, unaligned access, divide by zero
- Upon exception, hardware "faults" (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted

Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault
- Some faults are handled by notifying the process
  - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals or NT user-mode Async Procedure Calls (APCs)
    - SIGFPE, SIGALRM, SIGHUP, SIGTERM, SIGSEGV, etc.

Handling Faults

- The kernel may handle unrecoverable faults by killing the user process (core dump)
  - Program fault with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)

- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows "Blue screen of death"
    - » Kernel is halted, state dumped to a core file, machine locked up
  - Improvement from Windows 95 to Windows 7
    - » Where does the improvement come from?

Categorizing Events

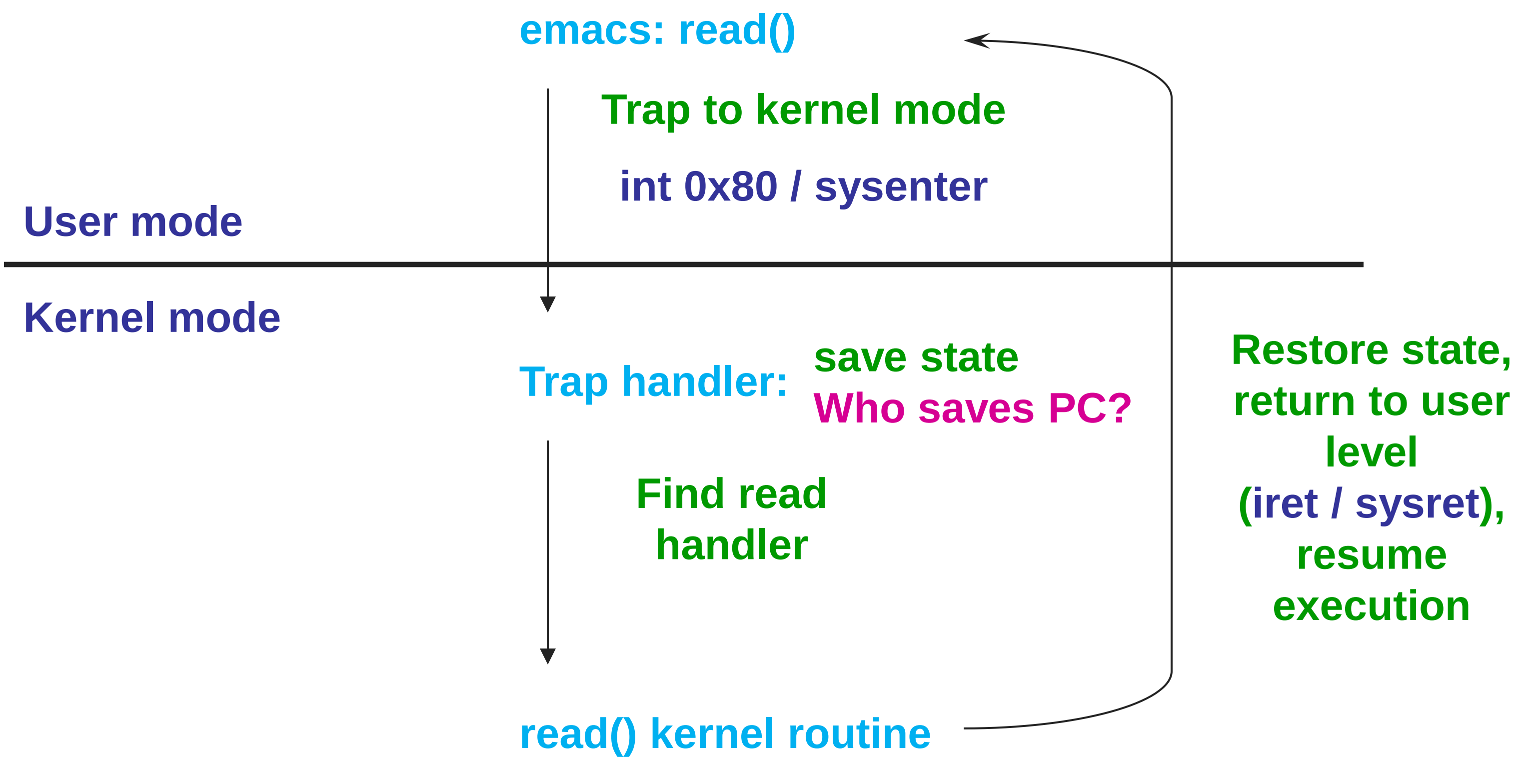
	Unexpected	Deliberate
Synchronous	fault	software interrupt (syscall trap)
Asynchronous	interrupt	Asynchronous system trap (AST)

System Calls

- For a user program to do something "privileged" (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call

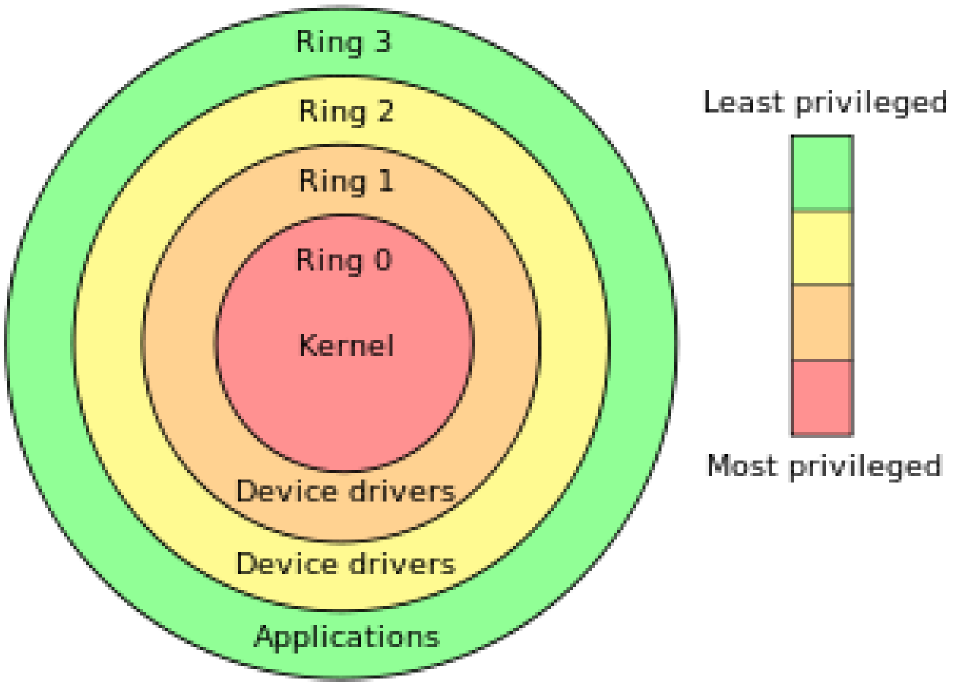
- Hardware provides a system call instruction that:
  - Causes an exception, which invokes a kernel handler
  - Passes a parameter determining the system routine to call
  - Saves caller state (PC, regs, mode) so it can be restored
    - » Why save mode?
  - Returning from system call restores this state

System Call

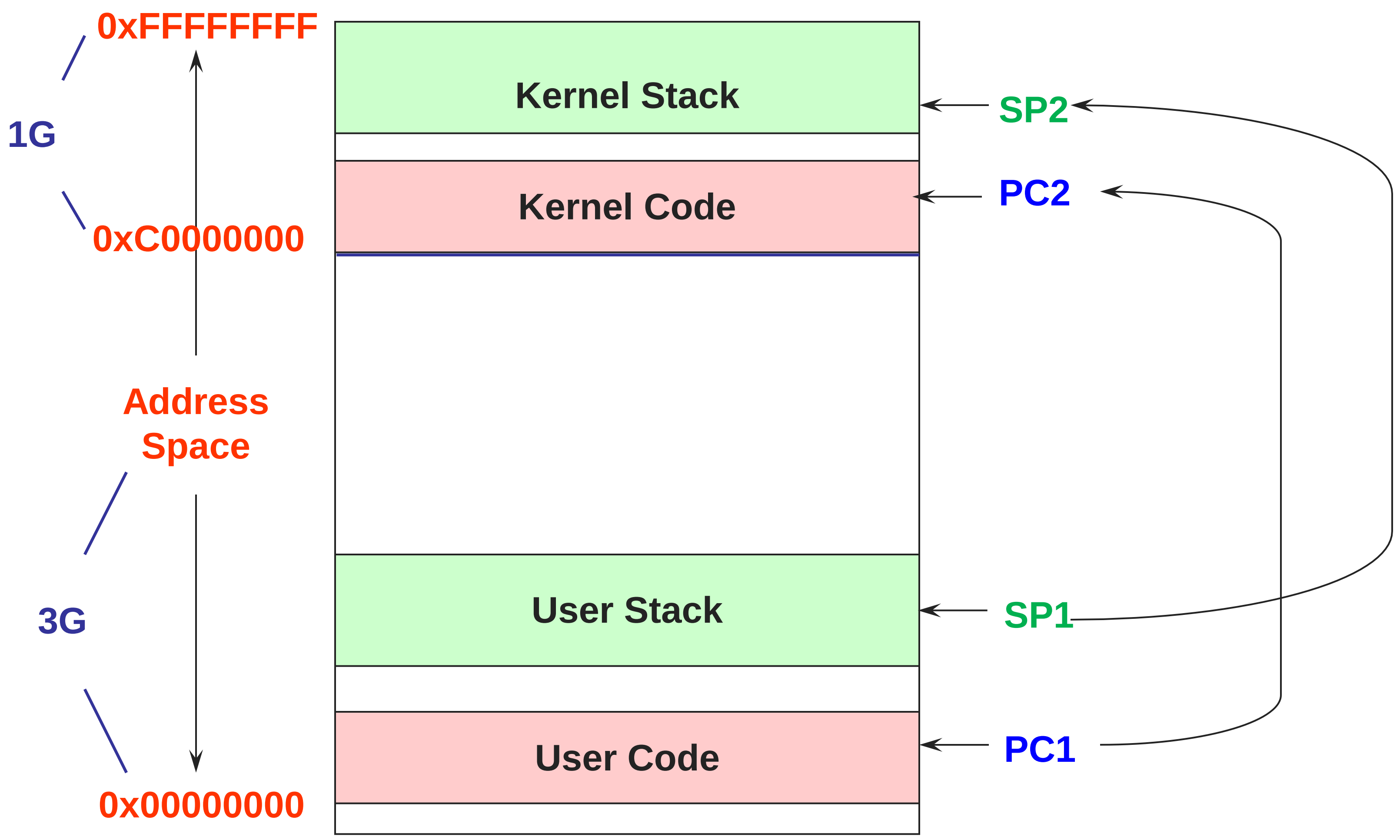

emacs: read()
Trap to kernel mode
int 0x80 / sysenter
User mode
Kernel mode
Trap handler:
save state
Who saves PC?
Find read handler
Restore state, return to user level (iret / sysret), resume execution
read() kernel routine

CPU Modes/Privileges

- System call
  - Ring 3 → Ring 0



Another view

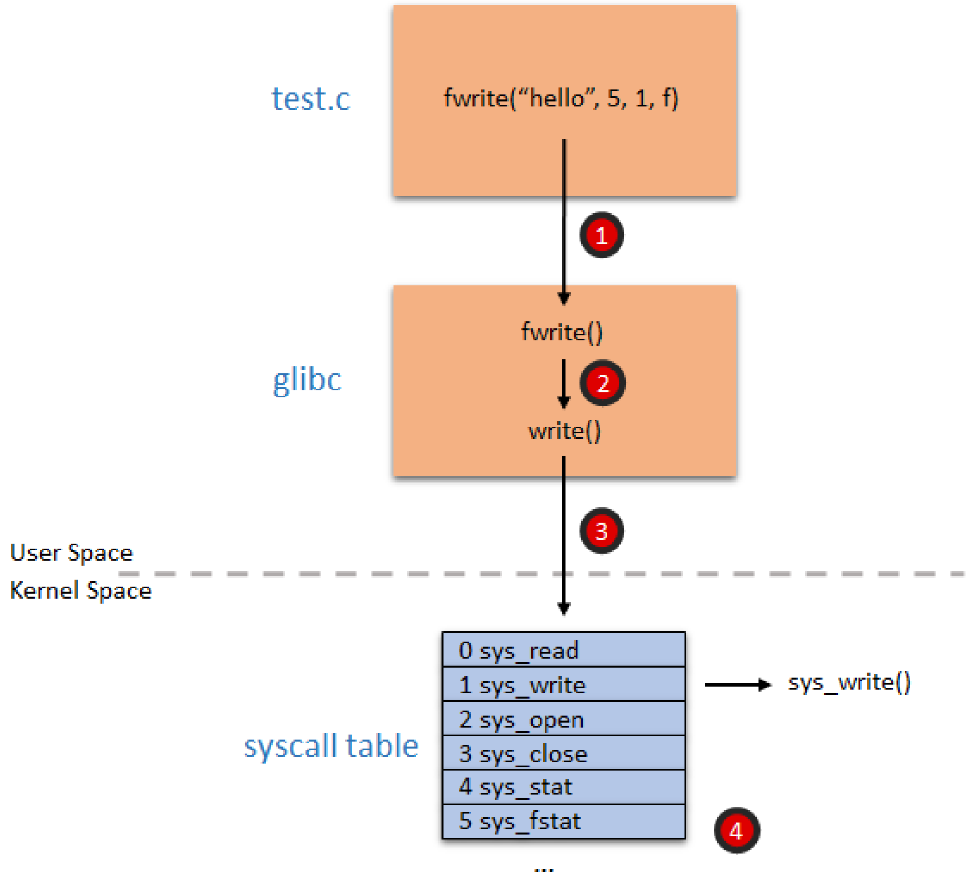


System Call Questions

- There are hundreds of syscalls. How do we let the kernel know which one we intend to invoke?
  - Before issuing **int $0x80** or **sysenter**, set **%eax** with the syscall number
- System calls are like function calls, but how to pass parameters?
  - Just like calling convention in syscalls, typically passed through **%ebx, %ecx, %edx, %esi, %edi, %ebp**
- How to reference kernel objects (e.g., files, sockets)?
  - Naming problem – an integer mapped to a unique object
    - » int fd = open("file"); read(fd, buffer);
  - Why can't we reference the kernel objects by memory address?

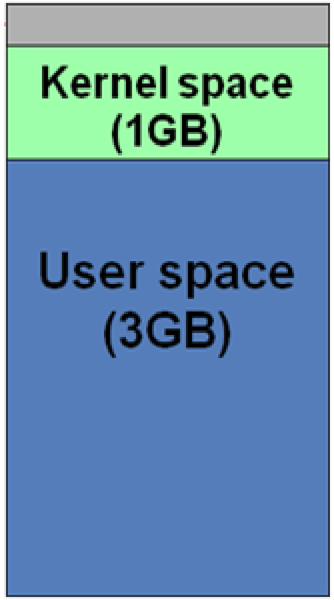
Interface to user programs (PintOS hint)

- For project 2 of pintos, there’s no real glibc. Instead, stubs such as write() and open() are provided in /lib/user/syscall.c
- Your job is to implement the kernel portion, the actual functionality



How to handle data from user space? (PintOS hint)

- System calls run in Ring 0 (highest privilege) and can read/write the entirety of the memory space, while a user-space program in Ring 3 can read/write only its user-space portion
- What can possibly go wrong in this syscall?
  - size_t read(int fd, void *buf, size_t nbytes)
- What if the user-mode program specifies an address in the kernel's address space?
  - Guards it by checking the address range



Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	software interrupt (syscall trap)
Asynchronous	interrupt	Asynchronous system trap (AST)

Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

Timer – special interrupt

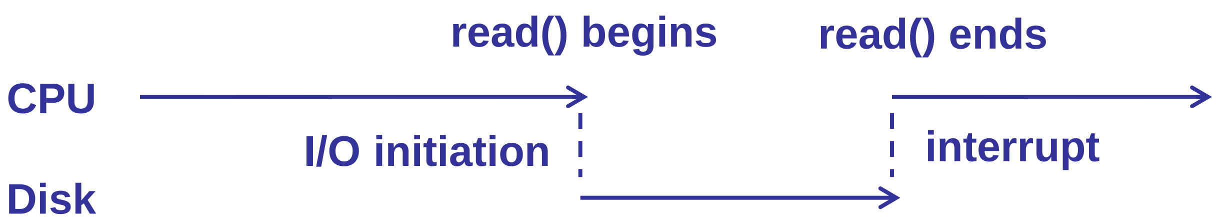
- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - » Setting timer is a privileged instruction
  - Basis for scheduling multiple tasks (*more later…*)
  - When timer expires, generates an interrupt
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., *sleep()*)

Timer (PintOS Hint)

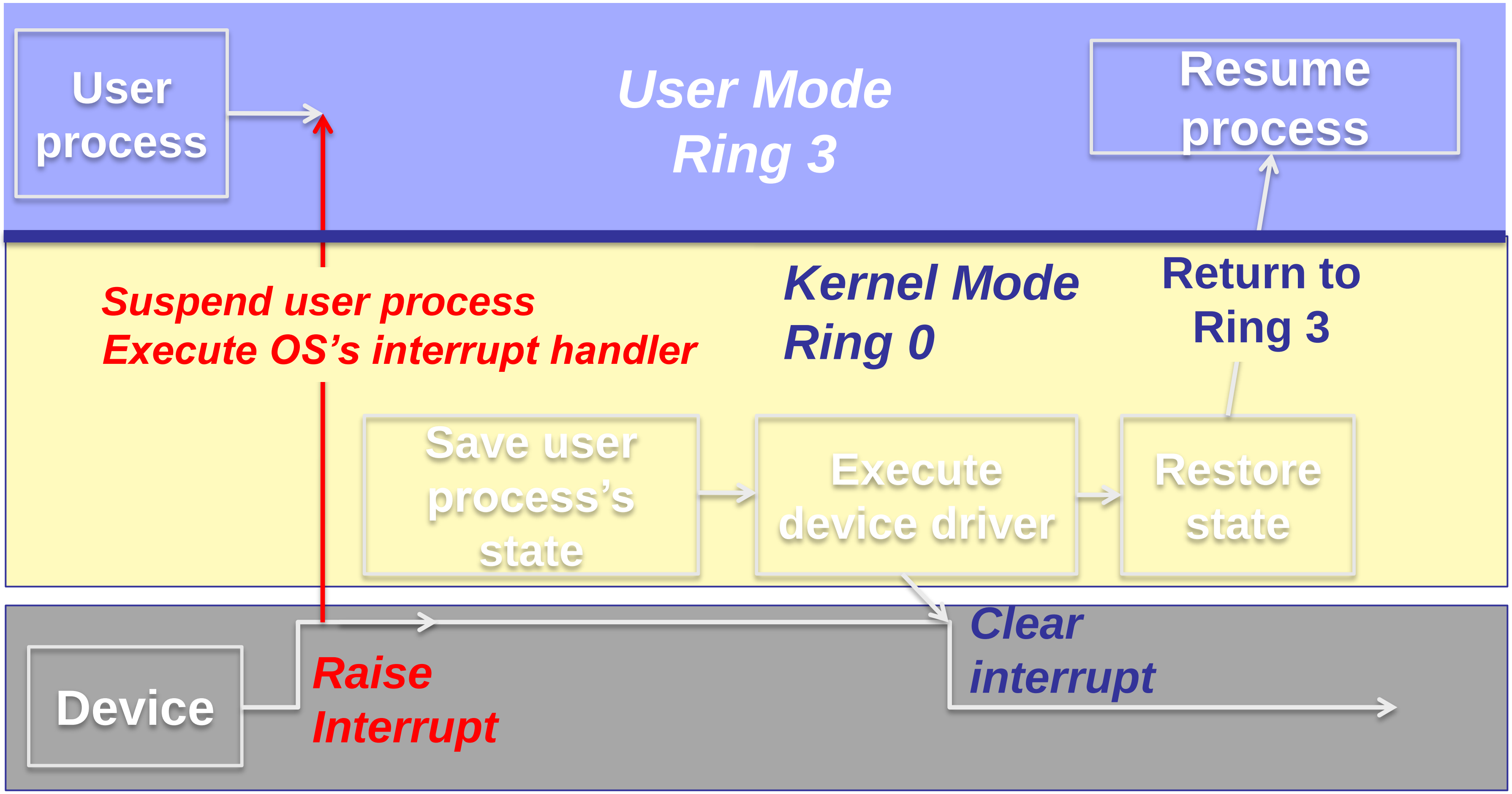
- How does it work?
  - Mechanical resonance of a vibrating crystal that is integrated into an electronic oscillator circuit
  - Programmable Interrupt Timer (PIT) configured in devices/timer.c:timer_init(), and devices/pit.c

I/O using Interrupts

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine



Interrupt Illustrated



I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the OS be interrupted? (Consider why there might be different interrupt levels)
  - Why not transfer control to user mode?
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?

How does it happen behind the scene (PintOS hint)

- Kernel initializes the interrupt descriptor table (IDT), a critical data structure gets called whenever an interrupt occurs (threads/interrupt.c)
- One of the entry in IDT gets invoked according to the interrupt number, control transferred to the pre-determined kernel function (threads/intr-stubs.S, threads/interrupt.c)
- In the case of syscall software interrupts (invoked by int $0x30), the control transfers to the syscall handler registered to handle interrupt 0x30 (as seen in userprog/syscall.c)
- After interrupt handling finishes, the control transfers back to where it was interrupted (via iret). In the case of syscall, it returns back to user space (Ring 3). See userprog/syscall.c

Synchronization

- Interrupts cause difficult problems
  - An interrupt can occur at any time
  - A handler can execute that interferes with code that was interrupted
- Need to guarantee that short instruction sequences execute atomically
  - Disable interrupts – turn off interrupts before sequence, execute sequence, turn interrupts back on
  - void driver_setup (void) {

```
    DISABLE_INTERRUPTS()
    global_variable += 100;
    global_variable += 100;
    assert( global_variable == 200);
    ENABLE_INTERRUPTS()
}
```

Synchronization

- OS must be able to synchronize concurrent execution
- Need to guarantee that short instruction sequences execute atomically
  - Special atomic instructions – read/modify/write a memory address, test and conditionally set a bit based upon previous value
    - » XCHG on x86

Summary

- Faults
  - Handled by the OS immediately
- System calls
  - Used by user-level processes to access OS functions
  - Access what is “in” the OS
- Exceptions
  - Unexpected event during execution (e.g., divide by zero)
- Interrupts
  - Timer, I/O

Next Time...

- Processes
  - Read Chapter 3