CSE 153
Design of Operating Systems

Winter 23

Lecture 3: OS model and Architectural Support

Last time/Today

- Historic evolution of Operating Systems (and computing!)
- Today—a little background:
  - Introduce some architecture support for Operating Systems
  - Understand how it is used to enable basic OS operation (the sleeping beauty model)

Lets start with a question

- What is the operating system?
  - Some special program?
  - If so, is it running all the time?
    - » But what if we have only one CPU?
  - How would it interact with the other programs? With the hardware?

Sleeping Beauty Model

- Answer: Sleeping beauty model
  - Technically known as *controlled direct execution*
  - OS runs in response to “events”; we support the switch in hardware
  - Only the OS can manipulate hardware or critical system state

- Most of the time the OS is sleeping
  - Good! Less overhead
  - Good! Applications are running directly on the hardware

Arch Support for this model

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Other stuff
  - Synchronization, memory protection, …

Protected Instructions

- OS must have exclusive access to hardware and critical data structures

- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
  - Halt instruction (why?)

Privilege mode

- Hardware restricts privileged instructions to OS
  - HW must support (at least) two execution modes: OS (kernel) mode and user mode
- Mode kept in a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == “kernel”)
  - CPU checks mode bit when protected instruction executes
  - Attempts to execute in user mode trap to OS

How do we make sure OS gets privileged mode but not programs?

Protocol for Secure Switching

- When the machine boots, OS is running
  - OS is mapped into part of memory of every process

- Going from higher privilege to lower privilege
  - Easy: can directly modify the mode register to drop privilege

- But how do we escalate privilege?
  - Special instructions to change mode and switch to the OS
    - System calls (`int 0x80, syscall, svc`)
    - Saves context and invokes designated handler
      - You jump to the privileged code; you cannot execute your own
    - OS checks your syscall request and honors it only if safe
  - Or, some kind of event happens in the system

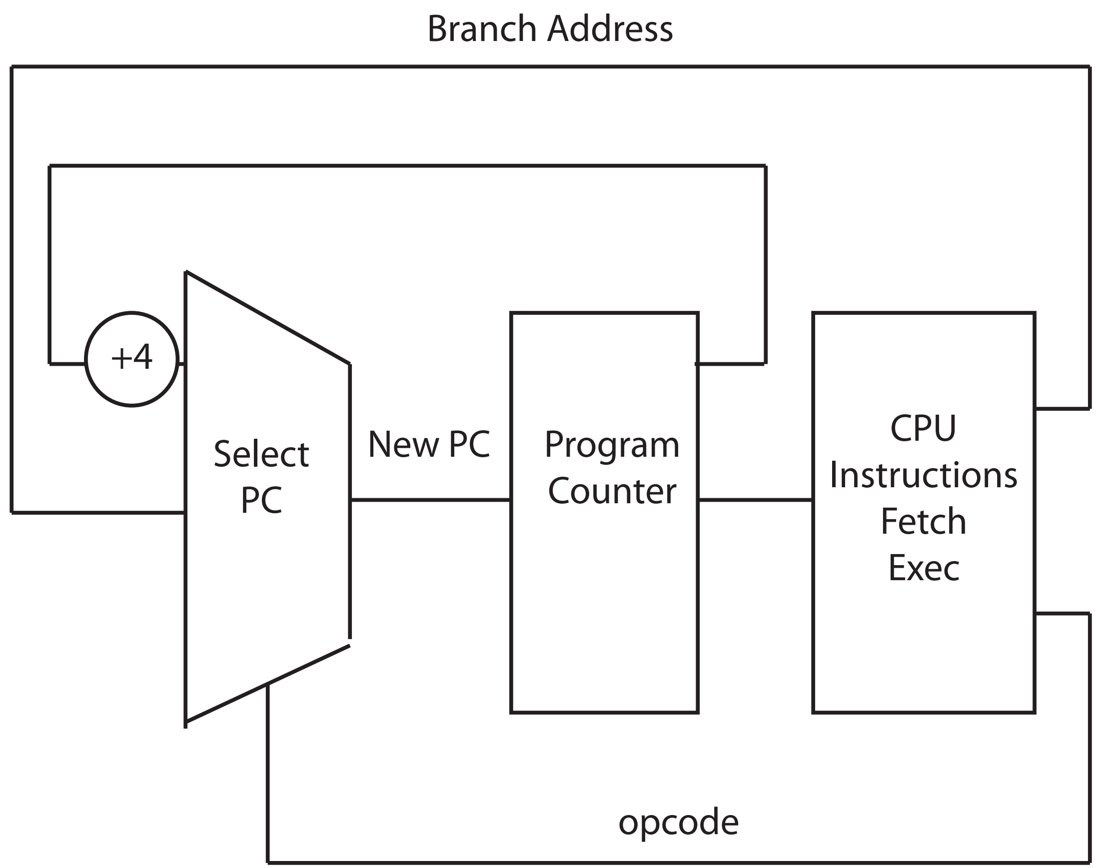
Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access

- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Other stuff
  - Synchronization, memory protection, …

Review: Computer Organization



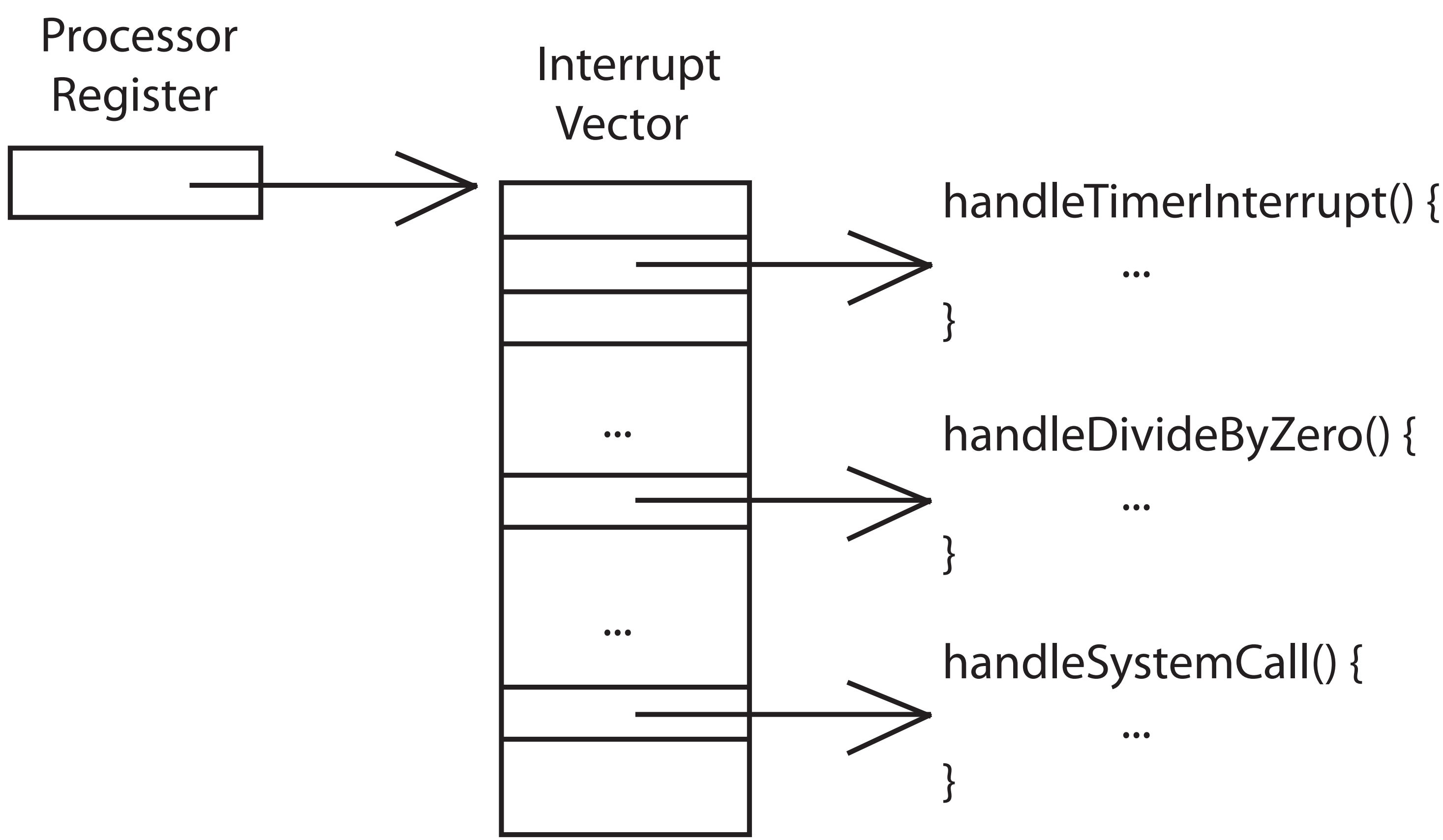
Events

- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both

- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine

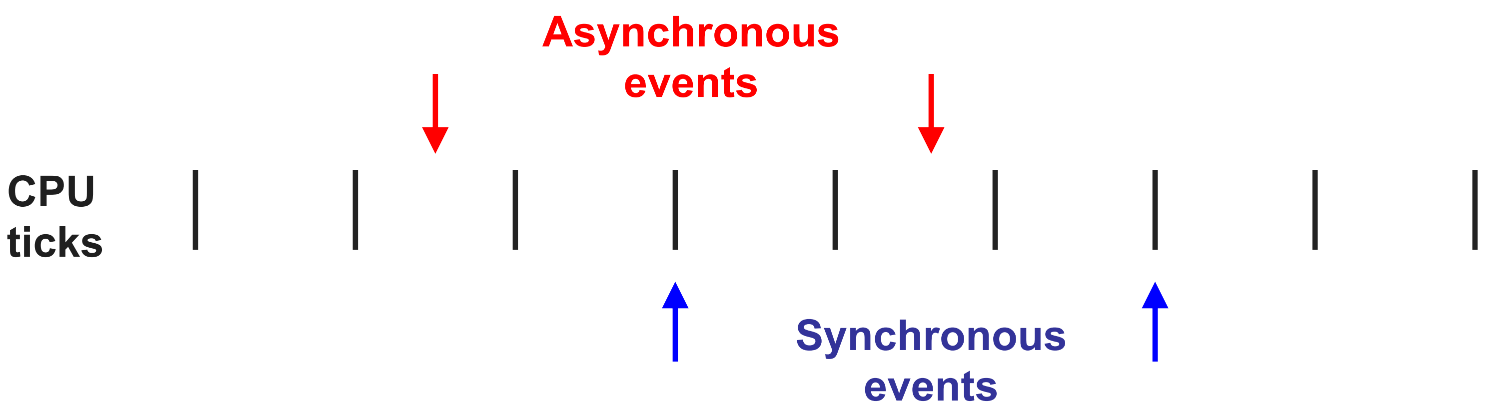
- Once the system is booted, OS is one big event handler
  - all entry to the kernel occurs as the result of an event

Handling events – Interrupt vector table



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	syscall trap
Asynchronous	interrupt	signal

- Terms may be slightly different by OS and architecture
  - E.g., POSIX signals, asynch system traps, async or deferred procedure calls

Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, memory access violation (unaligned, permission, not mapped, bounds…), illegal instruction, divide by zero
- Upon exception, hardware “traps” or “faults” (verb)
  - Must save state (PC, regs, mode, etc.) so that the faulting process can be restarted
  - Invokes registered handler

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

Handling Faults

- The kernel may handle unrecoverable faults by killing the user process
  - 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

Categorizing Events

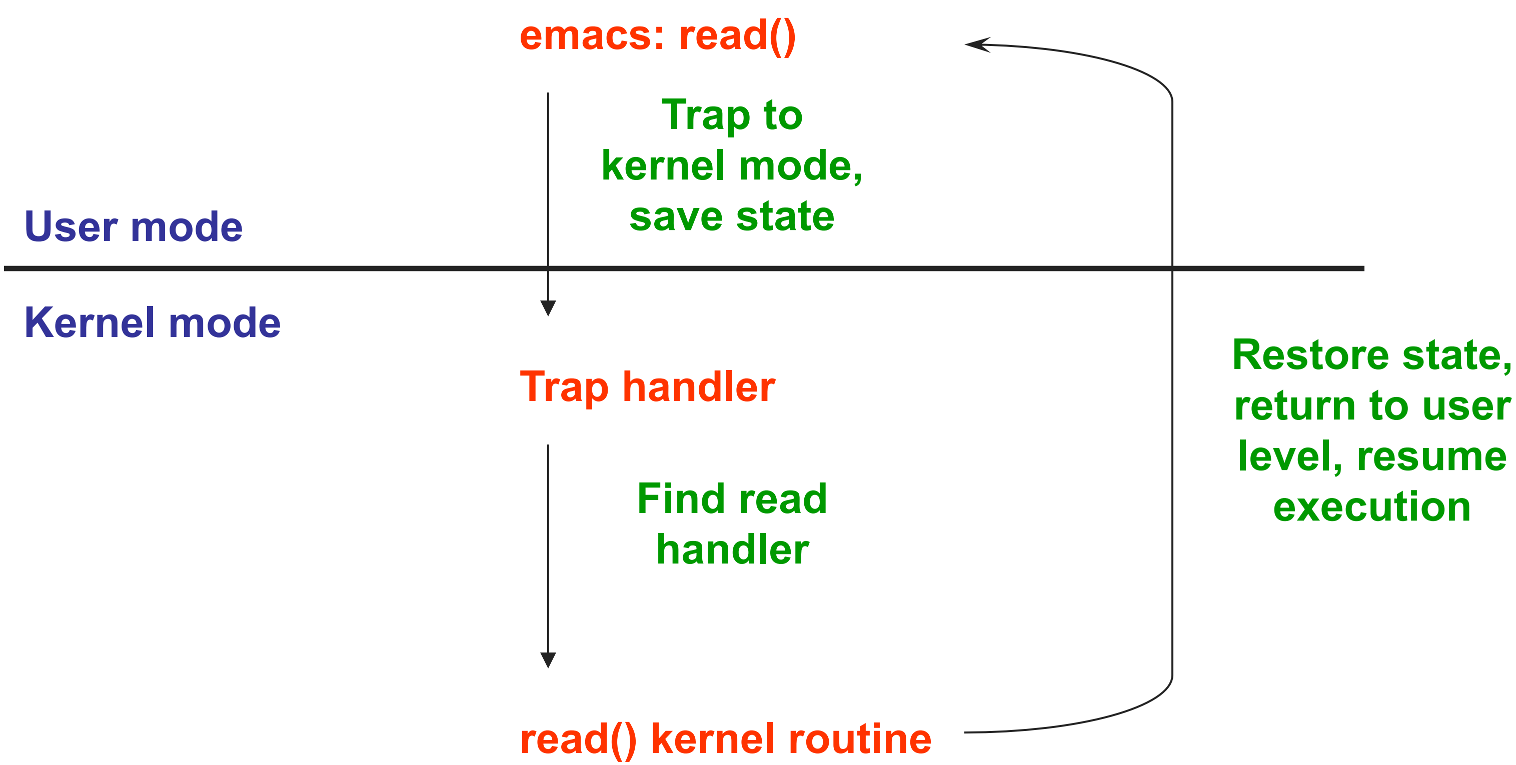
	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	signal

System Calls

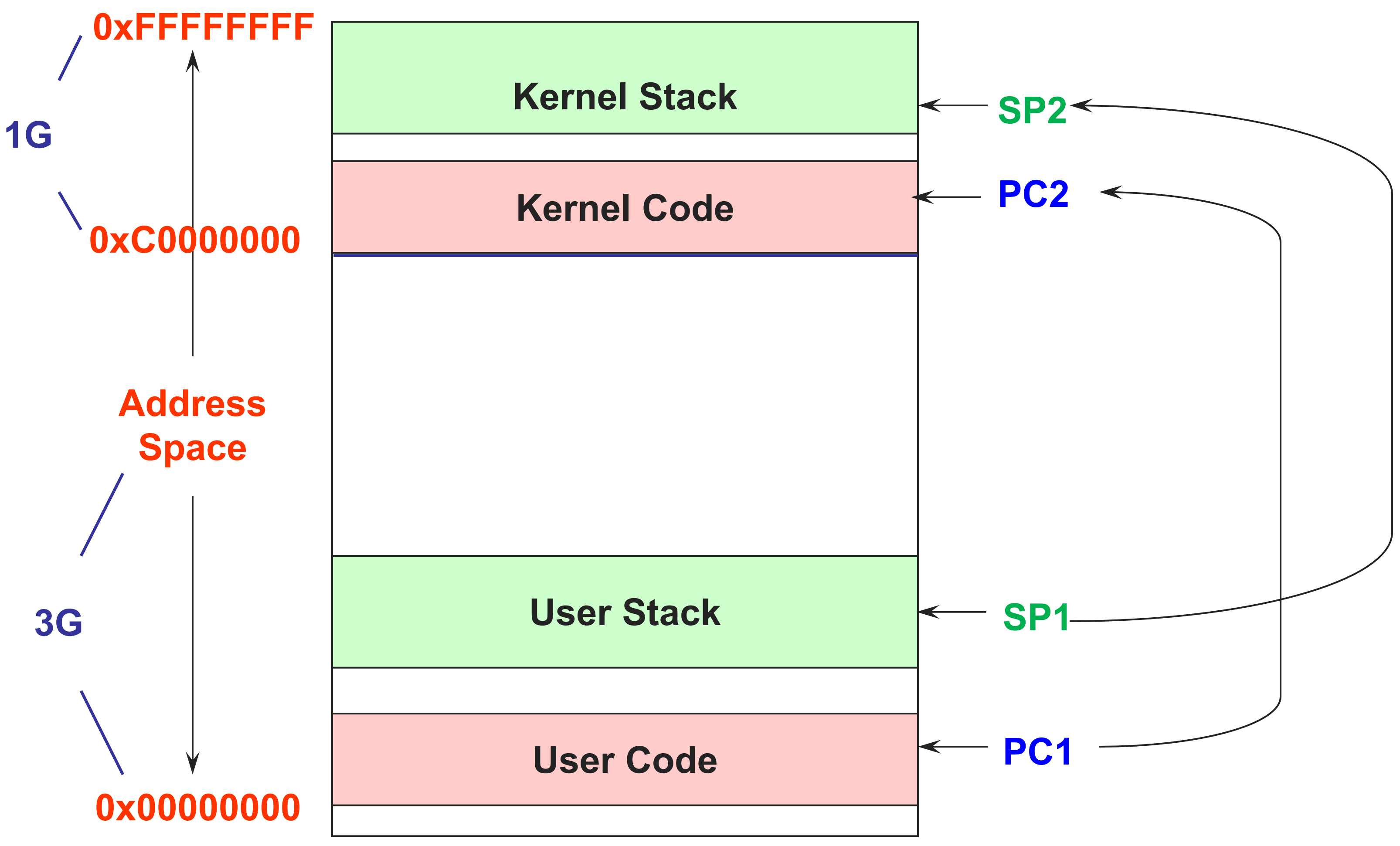
- For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
  - Known as crossing the protection boundary, or a protected procedure call

- 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,
save state
User mode
Kernel mode
Trap handler
Find read
handler
Restore state,
return to user
level, resume
execution
read() kernel routine

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/%rax** 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**

More questions

- 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?

System calls in xv6

- Look at trap.h and trap.c
  - Interrupt handlers are initialized in two arrays (idt and vectors)
    - Tvinit() function does the initialization
  - Syscalls have a single trap handler (T_SYSCALL, 64)
  - Trap() handles all exceptions, including system calls
    - If the exception is a system call, it calls syscall()

- Keep digging from there to understand how system calls are supported
  - You will be adding a new system call in Lab 1

Categorizing Events

	Unexpected	Deliberate
Synchronous	fault	syscall trap
Asynchronous	interrupt	software interrupt

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

Timer

- The key to a timesharing OS

- The fallback mechanism by which the OS reclaims control
  - Timer is set to generate an interrupt after a period of time
    - » Setting timer is a privileged instruction
    - » When timer expires, generates an interrupt
      - Handled by the OS, forcing a switch from the user program
    - » Basis for OS scheduler *(more later…)*

- Also used for time-based functions (e.g., *sleep()*)

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

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)
- 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?

Types of Arch Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.
  - Controlling access
- Generating and handling "events"
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap
- Other stuff
  - Synchronization, memory protection, …