OS: An Overview

ICS332

Operating Systems

Why are we studying this?

- After all, you probably will not develop an OS
- Important to understand what you use:
  - Develop better (apps);
  - What can and cannot be done;
  - Performance;
  - Why some OS can be better than some other
- Ubiquitous abstractions
  - OS concepts are fundamental and re-usable when implementing apps that are not operating systems
- Complex software systems
  - Many of you will contribute to complex software systems
  - Lessons learned from OSes study can be applied in other contexts

Studying OS Today

- Thanks to the open-source movement we have access to a lot of OS code
- Before 1993 OSes were even more mysterious
  - Implementation details of old (commercial) Oses often reveal how they were pretty cool (or pretty scary)
- In fact, it’s become possible for anyone to create an OS after reading other OS code (or how to contribute to an existing OS)
- And thanks to **virtualization** technology, one can play with and run OSes easily
  - Without compromising one’s computer
  - We won’t do that because we’re not doing any C

This Set of Lecture Notes

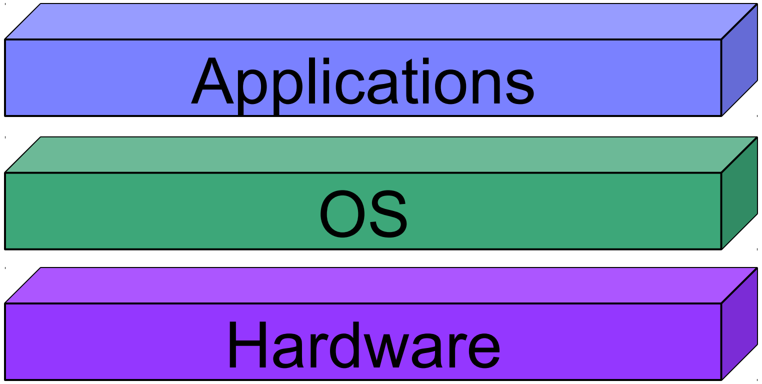
- This set of lecture notes is a 10,000 ft **overview** of the OS
- Many details will be explained throughout the semester
- Some terms are used, which you may not be familiar with, and that will all be explained later
- Some simplifying assumptions are made

  (If you know better, then bear with us until a further lecture)

What is an OS?

What is an OS?

- Typical answer: software layer between the applications and the hardware because the hardware would be too difficult for users to use



What is an OS?

Typical answer: it's "all the code you don't have to write" when you wrote your application

(Not quite right as there are tons of non-OS libraries that you didn't write as well)

What is an OS?

- Typical wrong answer: It's the one program that runs at all times

  It's a ***misleading view***

  No need to reserve one CPU for the OS!

What is an OS?

- The OS is a resource abstractor

  The OS defines a set of **logical** resources that **correspond to hardware** resources, and a set of well-defined operations on logical resources

  - Disks / Files;
  - RAM / Program data
- The OS is a resource allocator

  The OS decides **who** (which running program) gets **what** resource (share) and **when**

  - CPU / Processes;
  - Disks / Files
  - RAM / Program data

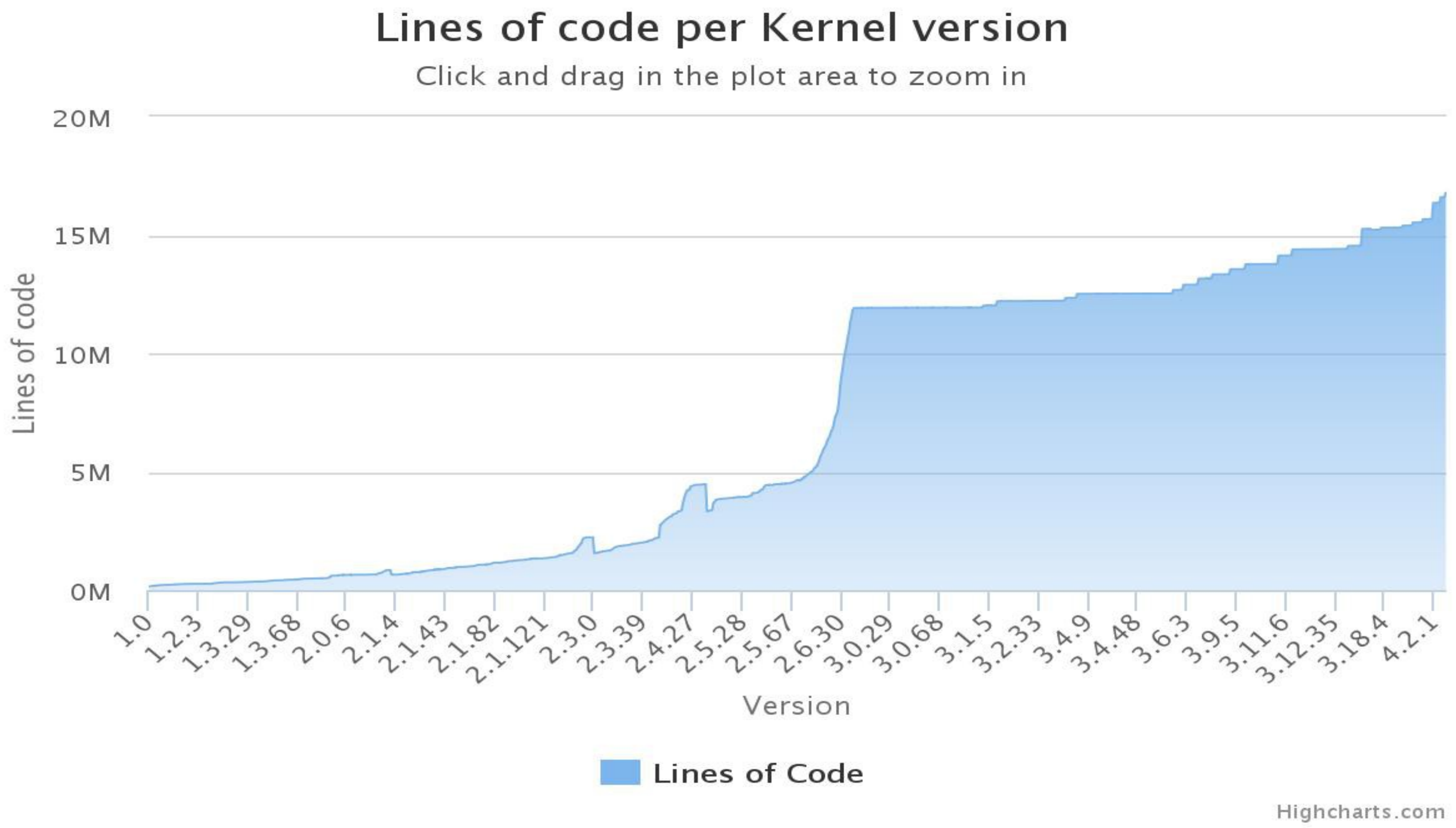
How big is an OS?

- The question “What is part of the OS and what isn’t?” is a difficult one
  - What about the windowing system? “system” programs?
  - The 1998 lawsuit against Microsoft putting “too much” in what they called the Operating System (see the book p. 6)
- But here are a few SLOC (Source Lines of Code) numbers
  - Windows NT (1993): 6 Million
  - Windows XP: ~50 Million
  - Windows Vista: ~XP + 10
  - Max OS X 10.4: ~86 Million
  - Linux kernel 2.6.29 (2009): 11 Million
  - Linux 4.3.2 (2015): 17 Million

Ubuntu distribution (not only OS): ~ 250 Million

- No matter: **OSes are BIG**

Linux Kernel Lines of Code



https://www.linuxcounter.net/statistics/kernel

How does one start an OS?

- When a computer boots, it needs to run a first program: the bootstrap program
  - Stored in Read Only Memory (ROM)
  - Called the "firmware"
- The bootstrap program initializes the computer
  - Register content, device controller contents, etc.
- It gets a piece of code on a device and execute it
- This code loads the OS kernel (code+data) into memory
- Within the kernel a piece of code is invoked to start the first process (called "init" on Linux, "launchd" on Mac OS X)
  - let's see it... "ps faux | less"
- And then, nothing happens until an event occurs
  - more on events in a few slides

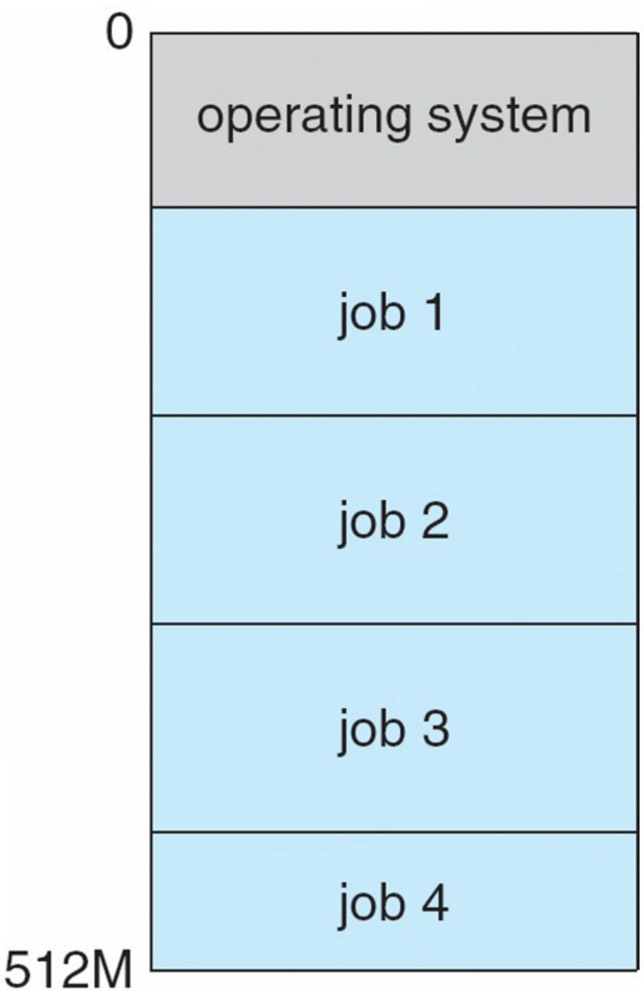
Multi-Programming

- Multi-Programming: Modern OSes allow multiple "jobs" (running programs) to reside in memory simultaneously
  - The OS picks and begins to execute one of the jobs in memory
  - When the job has to wait for "something", then the OS picks another job to run
  - This is called a context-switch, and improves productivity
- We are used to this now, but it wasn't always so
  - Single-user mode
    - Terrible productivity (while you "think", nobody else is using the machine)
  - Batch processing (jobs in a queue)
    - Low productivity (CPU idle during I/O operations)

Time-Sharing

- Time-Sharing: Multi-programming with rapid context-switching
- Jobs cannot run for “too long”
- Allows for interactivity
  - Response time very short
  - Each job has the illusion that it is alone on the system
- In modern OSes, jobs are called processes
  - A process is a running program
- There are many processes, some of which are (partly) in memory concurrently
  - Let's run the “ps aux” command on my laptop

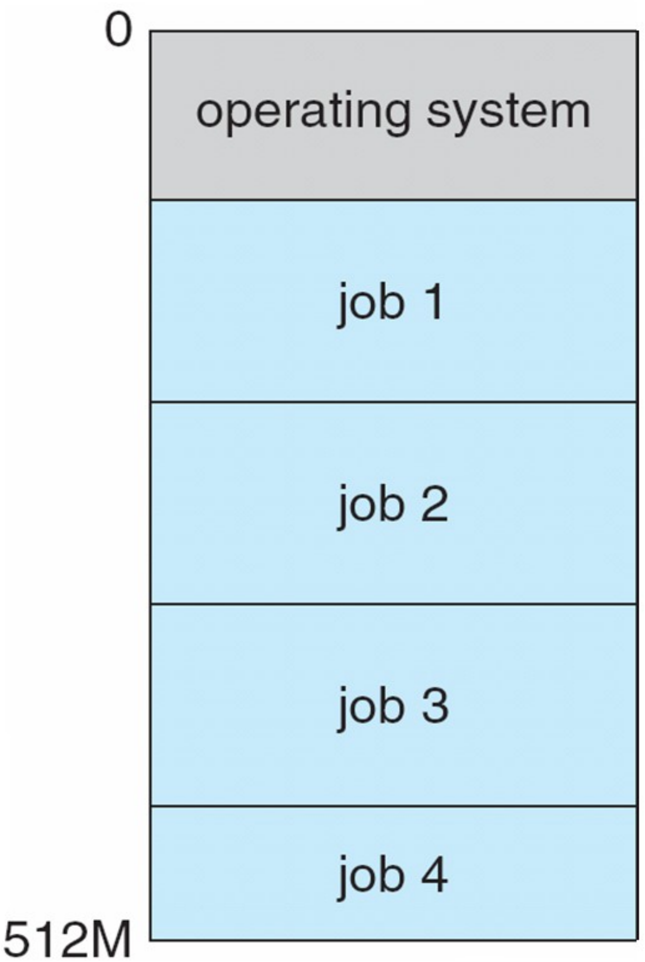
The Running OS



- The code of the operating system resides in memory at a specified address, as loaded by the bootstrap program
- At times, some of this code can be executed by a process
  - Branch to some OS code segment
  - Return to the program's code later
- Each process is loaded in a subset of the memory
  - Code + data
- Memory protection among processes is ensured by the OS
  - A process cannot step on another process' toes

Running the OS Code?

- The kernel is NOT a running job
- It's code (i.e., a data and a text segment) that resides in memory and is ready to be executed at any moment
  - When some event occurs
- It can be executed on behalf of a job whenever requested
- It can do special/dangerous things
  - having to do with hardware



A Note on Kernel Size

- In the previous figure you see that the kernel uses some space in the physical memory
- As a kernel designer you want to be careful to not use too much memory!
  - Hence the fight about whether new features are truly necessary in the kernel
  - Hence the need to write lean and mean code
- Furthermore, there is no memory protection within the kernel
  - The kernel's the one saying to a process "segmentation fault"
  - Nobody's watching over the kernel
- So one must be extremely careful when developing kernels
- Hence the reason why Kernel "hacking" is highly respected

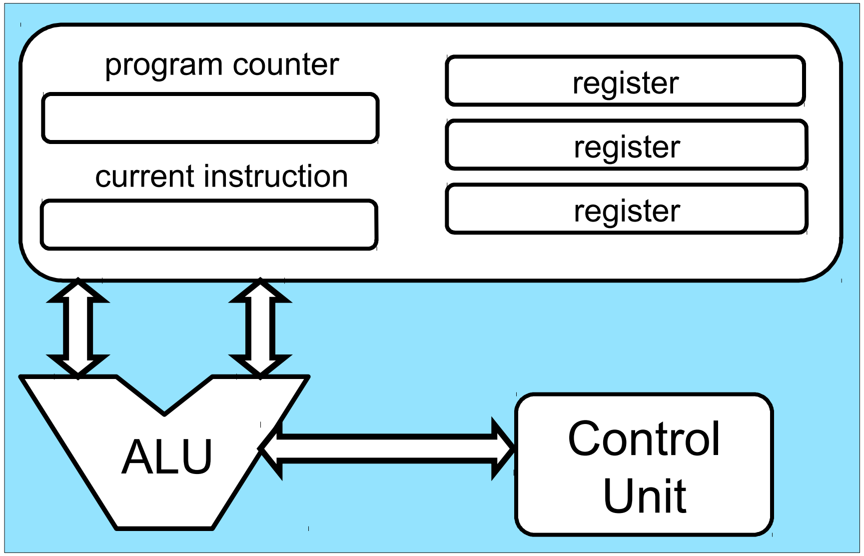
Protected Instructions

- A subset of instructions of every CPU is restricted in usage: only the OS can execute them
  - Known as protected (or privileged) instructions
- For instance, only the OS can:
  - Directly access I/O devices (printer, disk, etc.)
    - Fairness, security
  - Manipulate memory management state
    - Fairness, security
  - Manipulate protected control registers
    - Kernel mode, interrupt level (more on all this later)
  - Execute the halt instruction that shuts down the processor
- The CPU needs to know whether it can execute a protected instruction or not...

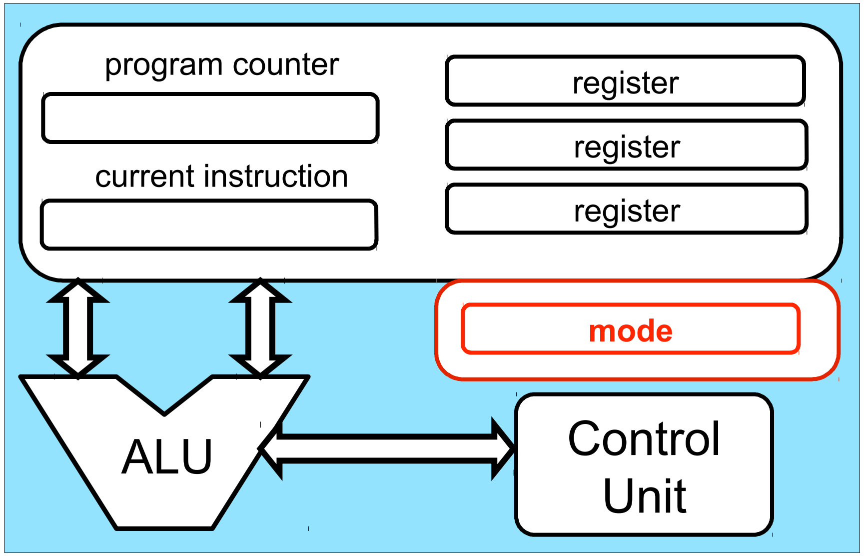
User vs. Kernel Mode

- All modern processors support (at least) two modes of execution:
  - User mode: In this mode protected instructions cannot be executed
  - Kernel mode: In this mode all instructions can be executed
- User code executes in user mode
- OS code executes in kernel mode
- The mode is indicated by a status bit in a protected control register
  - The CPU checks this bit before executing a protected instruction
- Setting the mode bit is, of course, a protected instruction

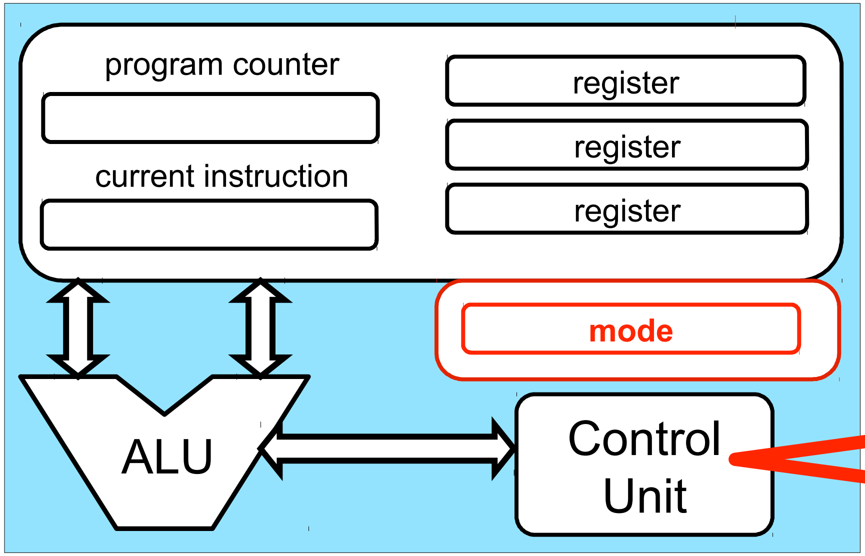
User vs. Kernel Mode



User vs. Kernel Mode



User vs. Kernel Mode



- Decode instruction
- Determine if instruction is privileged or not
  - Based on the instruction code (e.g., the binary code for all privileged instructions could start with ’00’)
- If instruction is privileged and mode == user, then abort!
  - Raise a “trap”

User vs. Kernel Mode

- There can be multiple modes
  - multiple levels in the kernel (Embedded systems)
- MS-DOS had only one mode, because it was written for the Intel 8088, which had no mode bit
  - A user program could wipe out the whole system due to a bug (or a malicious user)

OS Events

- An event is an "unusual" change in control flow
  - A usual change is some "branch" instruction within a user program for instance
- An event stops execution, changes mode, and changes context
  - i.e., it starts running kernel code
- The kernel defines a handler for each event type
  - i.e., a piece of code executed in kernel mode
- Once the system is booted, all entries to the kernel occur as the result of an event
  - The OS can be seen as a huge event handler

10K-foot View of Kernel Code

```
void processEvent(event) {
    switch (event.type) {
        case NETWORK_COMMUNICATION:
            NetworkManager.handleEvent(event);
            break;

        case SEGMENTATION_FAULT:
        case INVALID_MODE:
            ProcessManager.handleEvent(event);
            break;
        ...
    }
    return;
}
```

OS Events

- There are two kinds of events: interrupts (asynchronous) and traps (or exceptions or faults) (synchronous)
  - The two terms are often confused (even in the textbook)
  - The term fault often refers to unexpected events
- **Interrupts are asynchronous**
  - e.g., some device controller says “something happened”
  - e.g. INTR: “incoming data on keyboard” → RESP: “put the characters in memory for further processing”)
- **Traps are synchronous**
  - You thought you could do something but you can't
  - e.g., the CPU tried to execute a privileged instruction but it’s not in kernel mode
  - e.g., a division by zero

OS Events

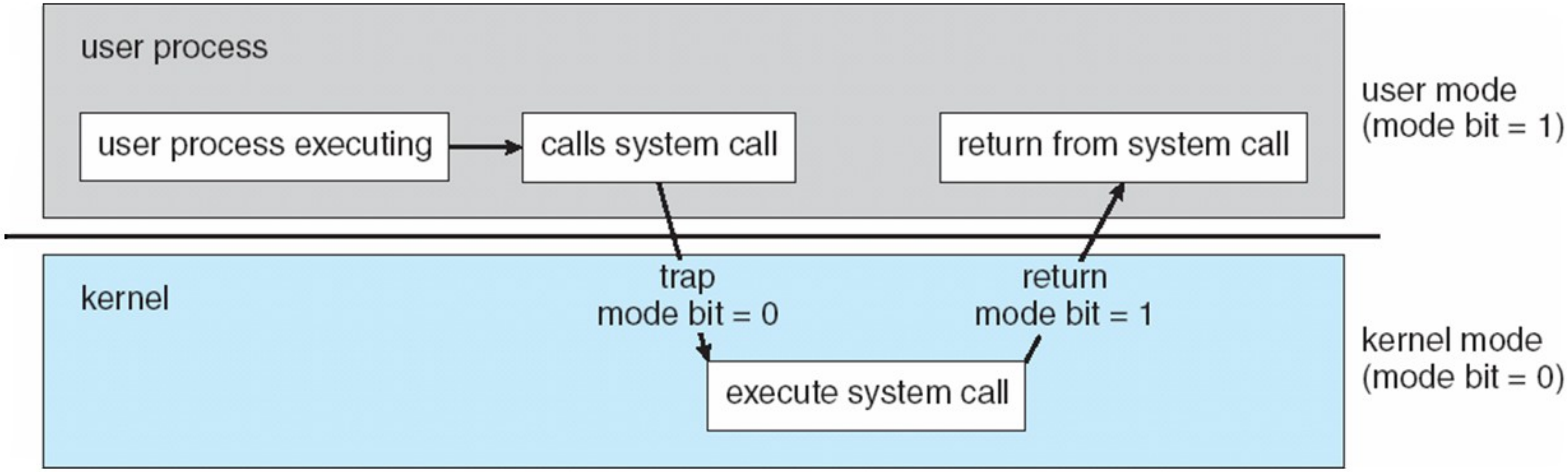
- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location in the kernel code
  - the “processEvent()” method in my mock-up kernel code a couple slides ago
- Could result in:
  - Some work being done by the kernel
  - A user process being terminated (e.g., segmentation fault)
  - Notifying a user process of the event
- What about “faults” in the kernel?
  - Say dereferencing of a NULL pointer, or a divide by zero
  - This is a fatal fault
  - UNIX Kernel Panic, Windows blue screen of death
    - Kernel is halted, state dumped to a core file, machine is locked up

System Calls

- When a user program needs to do something privileged, it places a system call
  - e.g., to create a process, write to disk, read from the network card
- A system call is a special kind of trap
- Every Instruction Set Architecture provides a system call instruction that
  - Causes a trap, which “vectors” to a kernel handler
  - Passes a parameter determining which system call to place (a number)
  - Saves caller state (PC, regs, mode) so it can be restored later
- But... On the x86 architecture the instruction is called INT :(

```
MOV    AH, 9
INT    21H           System call 21h = 33
```

System Calls



10K-foot View of Kernel Code

```
void processEvent(event) {
    switch (event.type) {
        case NETWORK_COMMUNICATION:
            NetworkManager.handleEvent(event);
            break;

        case SEGMENTATION_FAULT:
        case INVALID_MODE:
            ProcessManager.handleEvent(event);
            break;

        case SYSTEM_CALL:
            SystemCallManager.execute(event);
            break;
        ...
    }
    return;
}
```

Timers

- The OS must keep control of the CPU
  - Programs cannot get stuck in infinite loop and lock up the computer
  - Programs cannot gain an unfair share of the computer
- One way in which the OS (or kernel) retrieves control is when an interrupt occurs
- To make sure that an interrupt will occur reasonably soon, we can use a timer
- The timer interrupts the computer regularly
  - The OS always makes sure the timer is set before turning over control to user code
- Modifying the timer is done via privileged instructions

10K-foot View of Kernel Code

```
void processEvent(event) {
    Timer.set(1000);  // Will generate an event in 1000 time units
    switch (event.type) {
        case NETWORK_COMMUNICATION:
            NetworkManager.handleEvent(event);
            break;

        case SEGMENTATION_FAULT:
        case INVALID_MODE:
            ProcessManager.handleEvent(event);
            break;

        case SYSTEM_CALL:
            SystemCallManager.execute(event);
            break;

        case TIMER:
            Timer.handleEvent(event);
            break;
        ...
    }
    return;
}
```

Main OS Services

- Process Management
- Memory Management
- Storage Management
- I/O Management
- Protection and Security

Process Management

- A process is a program in execution
  - Program: *passive entity*
  - Process: *active entity*

- The OS is responsible for :
  - Creating and deleting processes;
  - Suspending and resuming processes;
  - Providing mechanisms for process synchronization;
  - Providing mechanisms for process communication;
  - Providing mechanisms for deadlock handling.

Memory Management

- Memory management determines what is in memory when
    - The kernel is ALWAYS in memory
- The OS is responsible for:
    - Keeping track of which parts of memory are currently being used and by which process
    - Deciding which processes (or parts thereof) and data to move into and out of memory
    - Allocating and deallocating memory space as needed
- The OS is not responsible for memory caching, cache coherency, etc.
    - These are managed by the hardware

Storage Management

- The OS provides a uniform, logical view of information storage
  - It abstracts physical properties to logical storage unit (e.g., as a “file”)
- The OS operates File-System management
  - Creating and deleting files and directories
  - Manipulating files and directories
  - Mapping files onto secondary storage
  - Backup files onto stable (non-volatile) storage media
  - Free-space management
  - Storage allocation
  - Disk scheduling

Note for Experts: We're talking about the OS not the kernel

I/O Management

- The OS hides peculiarities of hardware devices from the user
- The OS is responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), spooling (the overlapping of output of one job with input of other jobs), etc.
  - General device-driver interface

    So that multiple devices can be used with the same kernel as long as they offer some standard interface
  - Drivers for specific hardware devices

Protection and Security

- **Protection:** mechanisms for controlling access of processes to resources defined by the OS
- **Security:** defense of the system against internal and external attacks
  - including denial-of-service, worms, viruses, identity theft, theft of service
- The OS provides:
  - Memory protection
  - Device protection
  - User IDs associated to processes and files
  - Group IDs for sets of users
  - Definition of privilege levels

Privileged Instructions

- In class discussion: which of these instructions should be privileged, and why?
  - Set value of the system timer
  - Read the clock
  - Clear memory
  - Issue a system call instruction
  - Turn off interrupts
  - Modify entries in device-status table
  - Switch from user to kernel mode
  - Access I/O device

Privileged Instructions

- In class discussion: which of these instructions should be privileged, and why?
  - Set value of the system timer
  - Read the clock
  - Clear memory
  - Issue a system call instruction
  - Turn off interrupts
  - Modify entries in device-status table
  - Switch from user to kernel mode
  - Access I/O device

Sections 1.11 and 1.12

- The textbook has sections on “Computing environments” and “Open-Source Operating Systems”
  - Make sure you read them!
  - They talk about a number of topics that are part of the general culture that you should have, in case you don’t have it already

Conclusion

- This set of slides gave a grand tour of what an OS is and what it does
- We have purposely left many elements not fully explained... they will be elucidated throughout the semester

- Reading assignment: Chapter 1
  - It’s always a good idea to read and try to quickly do the “Practice Exercises” at the end of each chapter