CS 475
Operating Systems

Department of Mathematics
and Computer Science

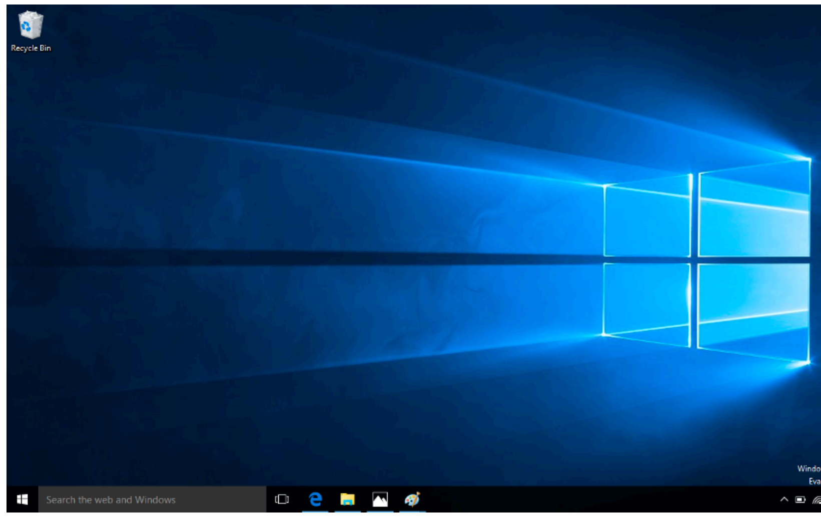
Lecture 2
Invoking the OS: Interrupts
and System Calls

Goals for Today

- Invoking the Kernel
  - Hardware Interrupts
  - Software Interrupts (Traps, system calls)
  - Protection: Dual Mode Operation
- Conclusion

After the OS Boots

- After the OS boots up.... it goes straight to sleep.
  - OS calls **HLT** to tell the CPU to go into low-power idle.
  - CPU idles until it detects an *event* ...which springs the OS into action once again.

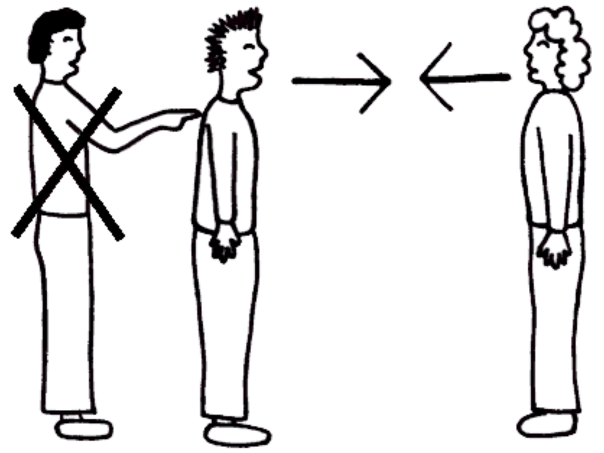


- Key questions:
  - What are these "events" that can wake up the CPU?
  - How does an event wake the CPU?
  - What does the CPU do when an event is detected?

Interrupting the CPU to Invoke the OS

- Interrupts can notify the CPU of an "event" that occurred
  - Two types of interrupts:
    - **Hardware Interrupts (or simply, *"Interrupts"*)**
      - Raised by hardware
    - **Software Interrupts (*"Faults, Traps, Errors"*)**
      - Raised by code execution
      - Faults and errors are forced by OS: divide by zero, bad memory access, etc.
      - Traps are user-initiated

- *First focus on hardware interrupts*

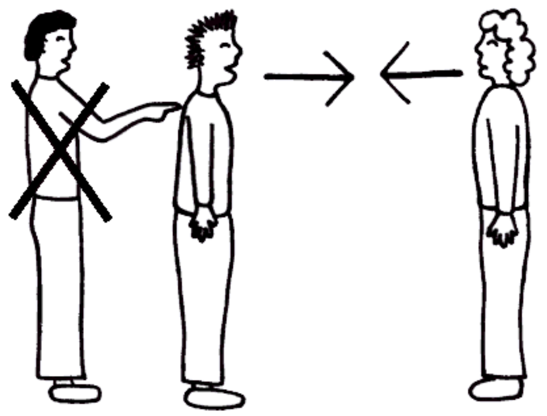


(Hardware) Interrupts

- *Interrupts* are hardware-raised signals that notify CPU of an "event" that has occurred, requiring OS to mediate.

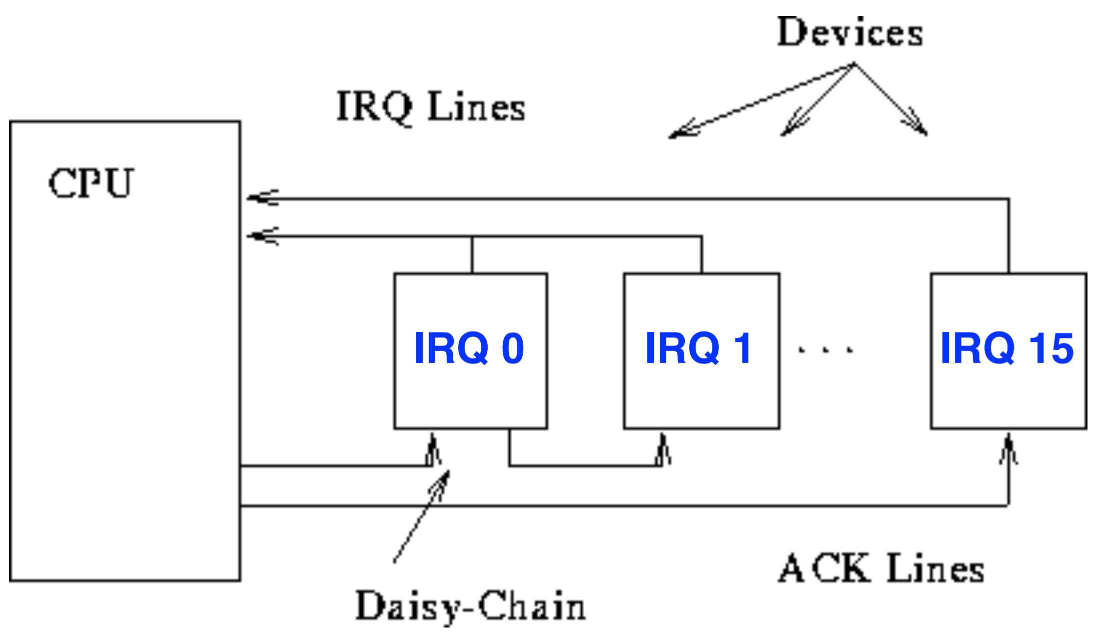
- Examples of hardware interrupts:
  - Mouse click, trackpad movement, keystroke, USB controller finishes reading, a disk finishes writing, network card read a packet, *etc*.

- Some questions regarding their mechanism:
  1. How does a device signal an interrupt to the CPU?
  2. How does the CPU detect that an interrupt has occurred? (It may be asleep)
  3. What does the CPU do afterwards to wake the OS?



Question #1 & #2: How to Interrupt the CPU?

- CPUs have one or more *Interrupt Request Lines (IRQs)*
  - Periodically senses for signals on IRQ lines
  - Priority matters (low IRQ number = checked first)



IRQ	Description
0	Timer Interrupt
1	Keyboard Interrupt
2	Cascade (used internally by the two PICs. never raised)
3	COM2 (if enabled)
4	COM1 (if enabled)
5	LPT2 (if enabled)
6	Floppy Disk
7	LPT1 / Unreliable "spurious" interrupt (usually)
8	CMOS real-time clock (if enabled)
9	Free for peripherals / legacy SCSI / NIC
10	Free for peripherals / SCSI / NIC
11	Free for peripherals / SCSI / NIC
12	PS2 Mouse
13	FPU / Coprocessor / Inter-processor
14	Primary ATA Hard Disk
15	Secondary ATA Hard Disk

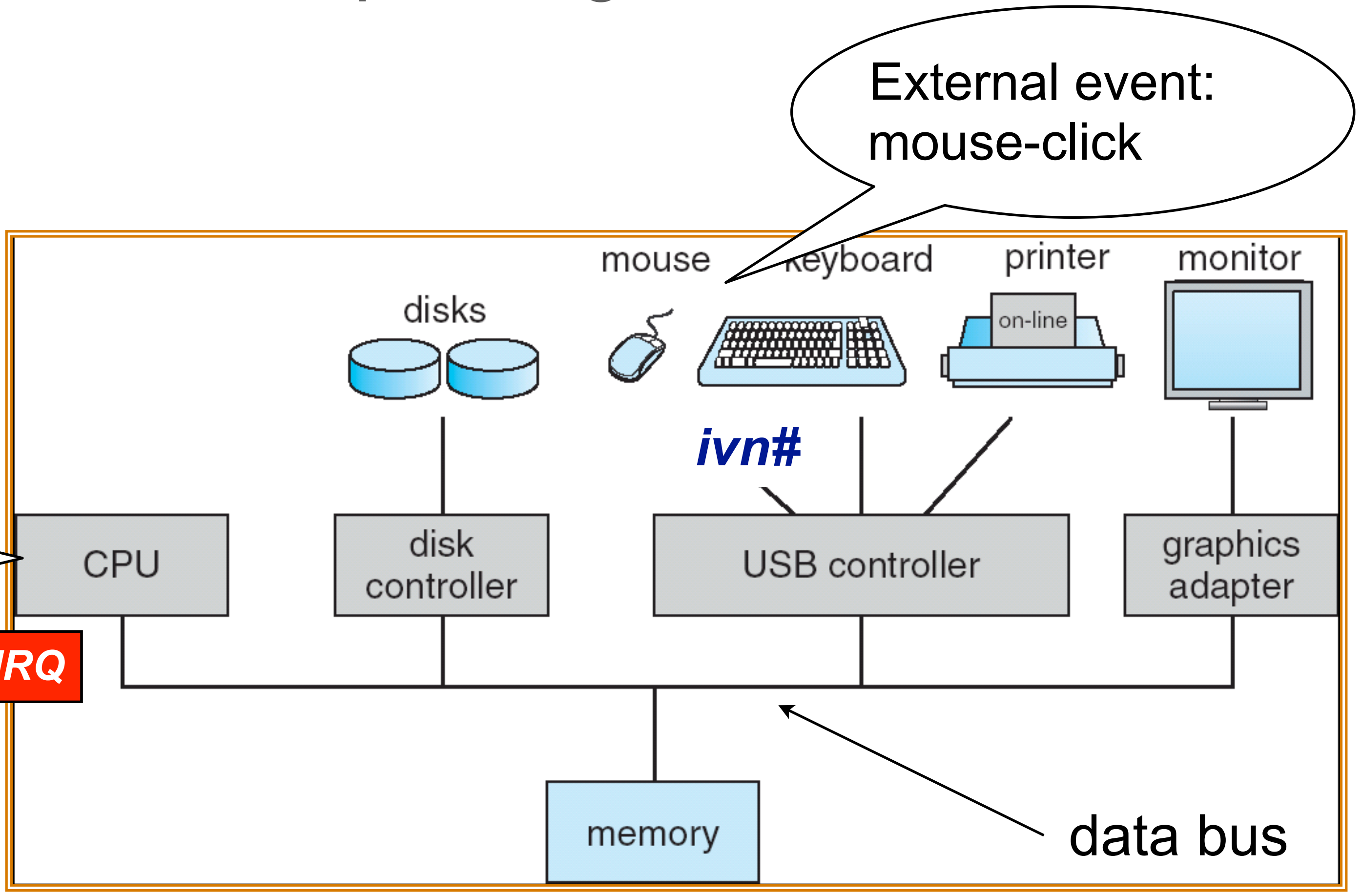
source: http://wiki.osdev.org/Interrupts

Question 1&2: How to Interrupt the CPU?

- An external device (e.g., mouse, or keyboard, or disk, ...)
  - Places an interrupt vector number (*ivn#*) on the data bus
    - (The ***ivn#*** tells the CPU of the specific interrupt that triggered!)
  - Next, the device *signals* its corresponding IRQ line

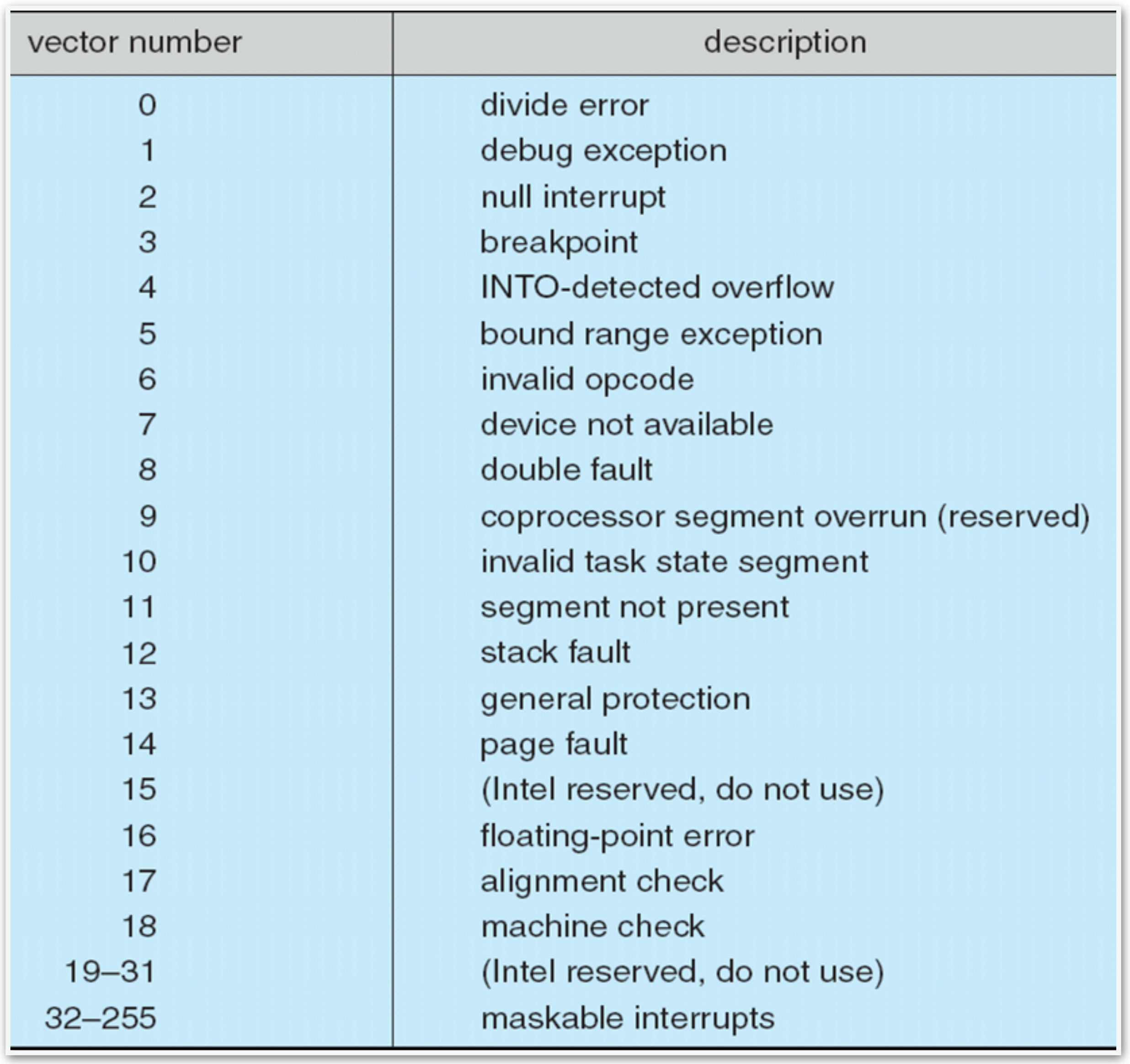
1) Wake up! IRQ line 12 (mouse) is raised.

2) Read ***ivn#*** off bus. (**"It's a click"**) ***... then what?***



Question 3: How Does the CPU Handle an Interrupt?

- The CPU architecture and the OS together define an *Interrupt Vector Table*
  - It's a hash table that maps an ***ivn#*** code to an OS system function.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

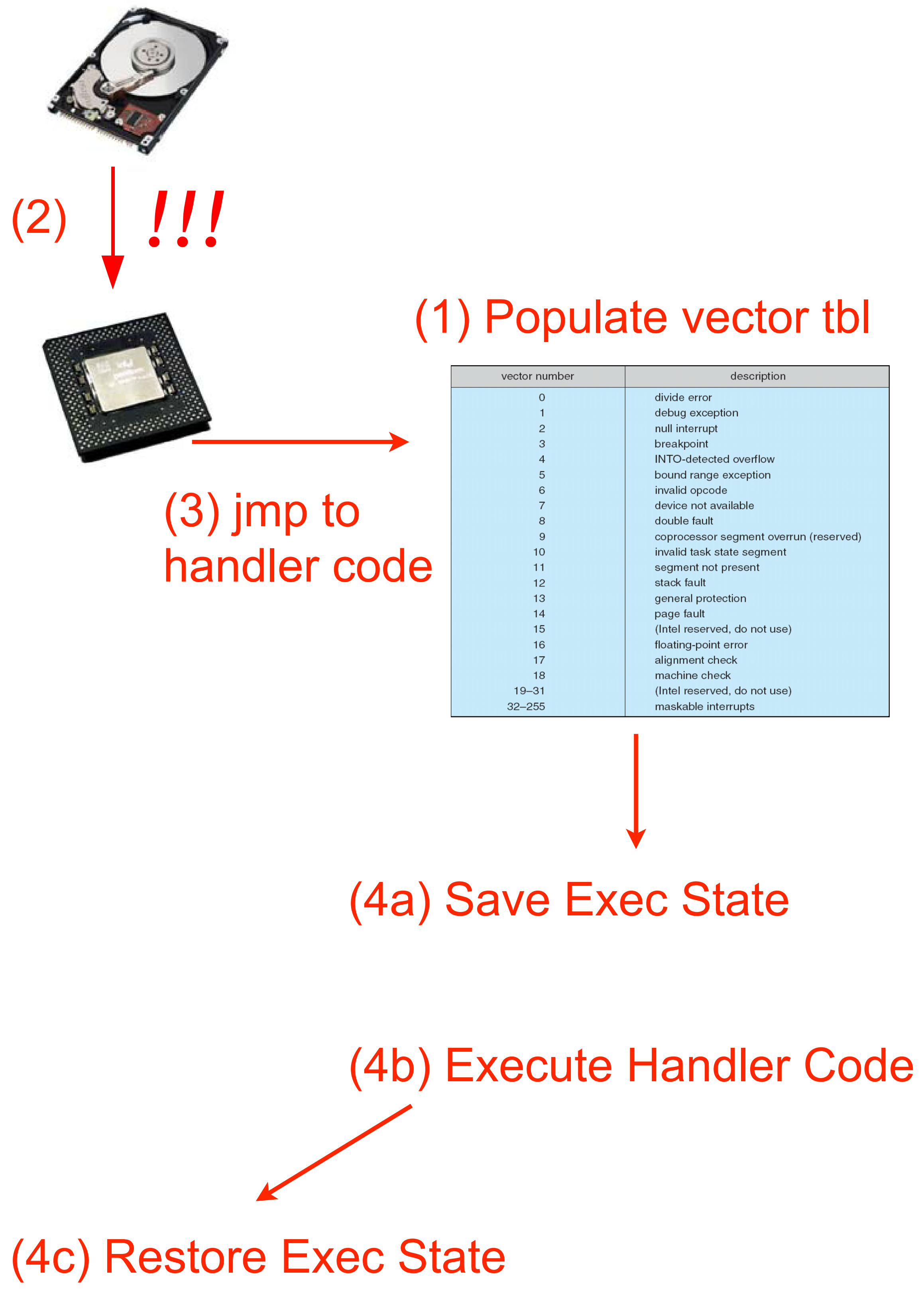
On Intel x86 architecture:

* 0-31 are defined by the CPU ("unmaskable" - can't be ignored - must handle now!)

* 32-255 are defined by the kernel ("maskable" - OS can finish what it's doing - these can wait)

Putting It All Together: Disk Example

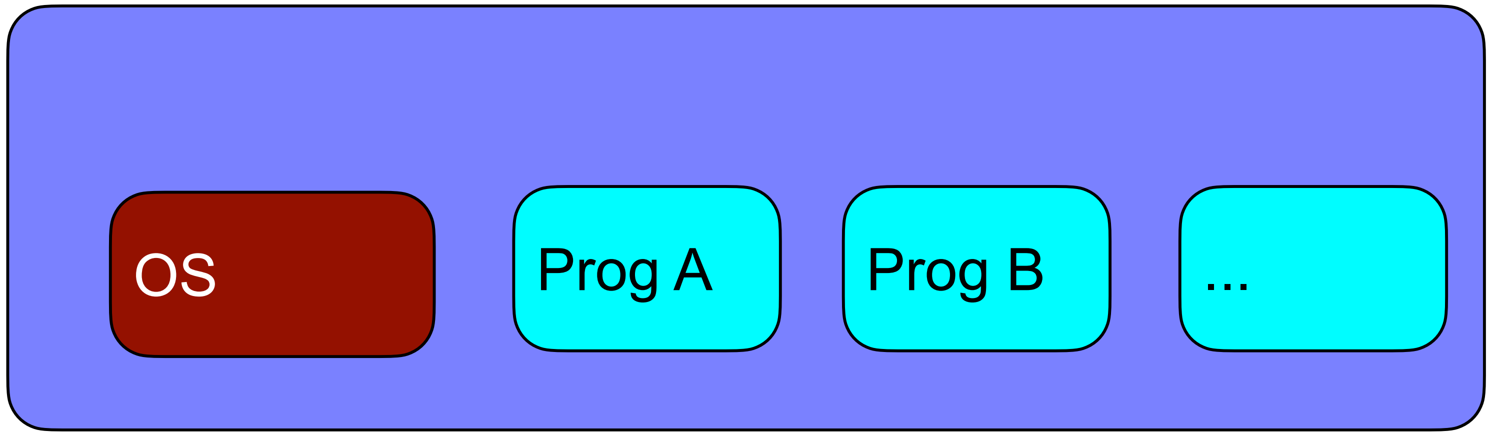
1. Interrupt Vector Tbl is populated boot time
2. Disk signals CPU that it is done writing
3. CPU senses signal on IRQ:
   a. Reads ***ivn#*** off data bus
   b. Looks up ***ivn#*** in the Vector Table
   c. Jumps to run appropriate interrupt-handler routine
4. Interrupt-Handler Routine (Now in OS!):
   - Save current execution state to memory
   - Run handler code (OS code)
   - CPU reloads saved state, returns to execution



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Another Interrupt Example

- Multiple programs can coexist and run concurrently.
  - ***But the CPU does not discern whose code it's running!***



Another Example: Evil Professor Problem

- David and OS students login to a server.
- David starts this program:

```
// Purpose: Students will get nothing done on the server
//          before tomorrow's deadline. LOL
// @date The day before homework deadline

int main() {
   while (1)
       ;   // Monopolize CPU so no one's code will ever get to run
   return 0;
}
```

- David's code gets scheduled to run on the CPU.
  - *(How does the OS ever regain control of the CPU to run someone else's code?)*

Solution: Build-in a Hardware Timer

```
// Purpose: Students will get nothing done on the server
//          before tomorrow's deadline. LOL
// @date The day before homework deadline

int main() {
   while (1)
       ;   // no one's code will ever get to run
   return 0;
}
```

Hardware Timer

- **Solution:** Add a hardware timer that will periodically interrupt the CPU.
  - CPU looks up and runs the interrupt-service-routine to handle the timer interrupt, which simply performs a context switch!
  - (And thus, the Timesharing Model is born)

Software Interrupts

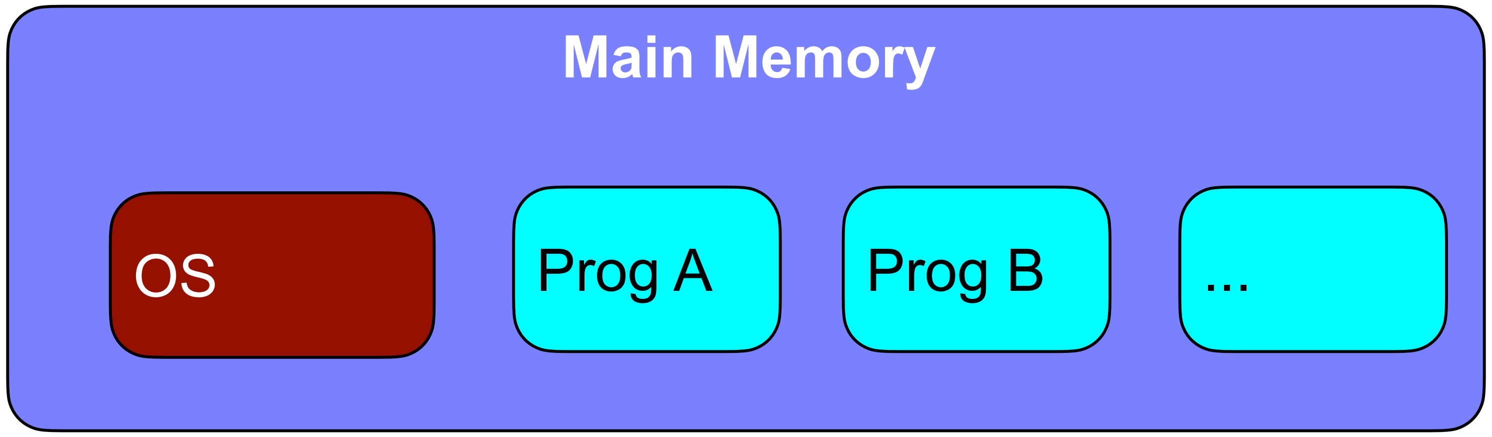
- *Software Interrupts* occur as a result from executing a program instruction
  - *Faults and Errors*
    - **Faults** are recoverable (*e.g.*, a "page fault")
      - Fault-raising code can continue to run *after* the "fault" has been handled
    - **Errors** are not recoverable (*e.g.*, div-by-zero, segmentation fault)
      - Error-raising code must be terminated *immediately* by OS
  - *Traps*
    - Purposefully raised by user programs using the **`"int"`** assembly instruction (it means "interrupt" in assembly language, not integer!)
    - *(Why and when would we want to intentionally interrupt the CPU?)*

Goals for Today

- Invoking the Kernel
  - Hardware Interrupts
  - Software Interrupts
  - Protection: Dual Mode Operation
- Conclusion

Protection of OS Code

- Recall: Problem of Multiprogramming and Timesharing
  - Programs are all loaded in memory.
  - Must keep user programs isolated from the OS



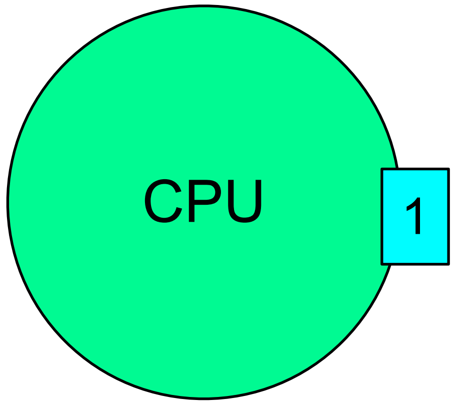
- Also recall what a CPU does:
  - Fetch Instruction, Decode It, Execute It, repeat.
  - ***The CPU does not discern whose code it's running!***

Dual-Mode Operation

- User programs must not be allowed to execute certain CPU instructions
  - e.g., instructions that could tamper with OS's memory

- Mechanism ("how to do it"): Dual-Mode Operation
  - Classify each CPU instruction to be either *privileged* or *unprivileged*
    - *(What are some examples of each?)*
  - Next, add a "Mode Bit" register to the CPU
    - When **mode == 1** the CPU is in User Mode
      - (CPU can only run unprivileged instructions)
    - When **mode == 0** the CPU is in Kernel (Supervisor) Mode
      - (CPU can now run privileged instructions also)

Protection: CPU Perspective



Mode Bit
(1 = user mode)
(0 = kernel mode)

User code

```
MOV #0x08,R5       ; unprivileged. OK
AND #0x00,R6       ; unprivileged. OK
ADD #0x03,R6       ; unprivileged. OK
HTL                ; privileged (but in user mode)! Terminate
                                                    program!
```

Crossing the Protection Boundary

- But certainly, a user program should be able to:
  - Create another process
  - Read a file
  - Print to screen
  - But these are all "privileged" instructions!

- User code can do these things, but they just need to be requested *through* the OS!

How to Cross the Protection Boundary?

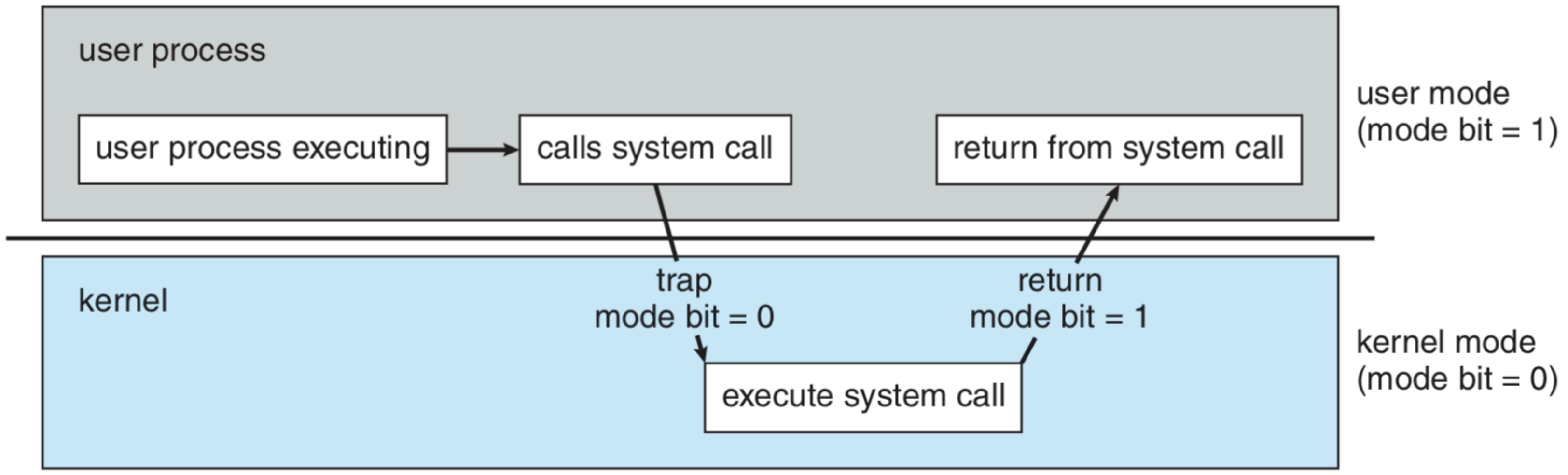
- How *do we* go from user-mode to kernel-mode?
- Let users ***Trap an Interrupt***
  - Allow users to intentionally interrupt the CPU in order to trigger OS
    - Use the unprivileged **int** (trap interrupt) instruction

- **Important:** As soon as the CPU detects *any* interrupt:
  - `CPU[mode] = 0  // CPU automatically enters kernel mode`
  - Look up interrupt vector
  - Execute the interrupt-handler function in kernel model
  - `CPU[mode] = 1  // CPU resumes user mode`

System Calls

- Sometimes users need to invoke privileged instructions
  - *e.g.*, create a process/thread, printing to screen, ...
  - Must get the OS to do privileged instructions on our behalf

- *System Calls* are made available by the users to invoke one of the OS's provided services.
  - Made available through the OS System API



Making a System Call

- From the user program:
  - I need to use an OS service, and that service is identified by some W
  - Put W in a CPU register
  - Put 128 on the data bus
  - Now trap the CPU by issue an **`int`** instruction
- On the CPU: 🤔 I've been interrupted by something!
  - *Enter kernel mode automatically on interrupt detection! CPU[Mode] = 0*
  - Read 128 off the data bus, lookup 128 in the Interrupt Vector Table
    - "Okay, 128 is listed as a system call, but *which one?"*
    - Lookup W, then run the corresponding OS system function.
    - *Leave kernel mode! CPU[Mode] = 1*

Common System Functions

	Unix-Based	Windows
Process Control	`fork()`	`CreateProcess()`
	`exit()`	`ExitProcess()`
	`wait()`	`WaitForSingleObject()`
File System	`open()`	`CreateFile()`
	`read()`	`ReadFile()`
	`write()`	`WaitFile()`
	`close()`	`CloseHandle()`
Protection	`chmod()`	`SetFileSecurity()`
	`chown()`	`SetSecurityDescriptorGroup()`
…	**…**	**…**

Example System Call in `printf()`

- C program in user-mode invoking a `printf()` call
  - But writing to the screen is privileged!
  - `printf()` actually traps the `write()` *syscall* in kernel mode

```c
int main(int argc, char *argv[]) {
   //start in user-mode: unprivileged instructions
   int x, y;
   x = 1;
   y = 3;

   //privileged instruction: syscall to enter kernel mode
   printf("Hello world! %d %d\n", x, y);

   //back to user mode
   return 0;
}
```

`hello.c`

The Cost of System Calls

- System calls are expensive 🤑🤑🤑 Consider this program:
  - Let's time a program with and without making 100,000 system calls

- Assuming system calls are as fast as regular CPU instructions. The slow-down between **A** vs. **B** should be about 2x, but...

```c
int main() {
    int i = 0;

    while (i < 100000) {
        printf("Hello!\n");
        i++;
    }
    return 0;
}
```

A

```c
int main() {
    int i = 0;

    while (i < 100000) {
        i++;
    }
    return 0;
}
```

B

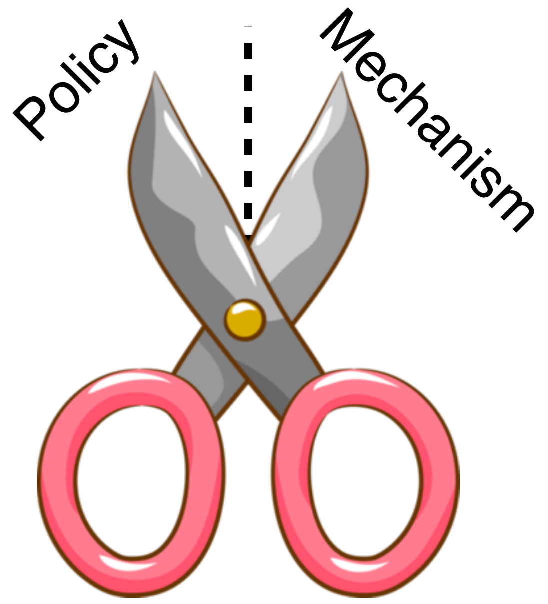
Goals for Today

- Invoking the Kernel
  - Hardware Interrupts
  - Software Interrupts (Traps, system calls)
  - Protection: Dual Mode Operation
- OS Design Principles
- Conclusion

OS Design Principles

- Just like in real life, there are pros/cons to big/small governments.
  - When designing an OS, we ask ourselves, how imposing should it be?
  - Should OS be big & feature rich?
  - Should OS be small & only offer basic services?

- To guide OS design, we must first understand the idea of *Separation of policy and mechanism*
  - Policy: What do we want to achieve?
  - Mechanism: How do we enforce that policy?



Separation of Policy and Mechanism

- Why separate Policy from Mechanism?
  - Adaptability: one can change without affecting the other
  - Mechanism may change, but doesn't change policy and vice versa

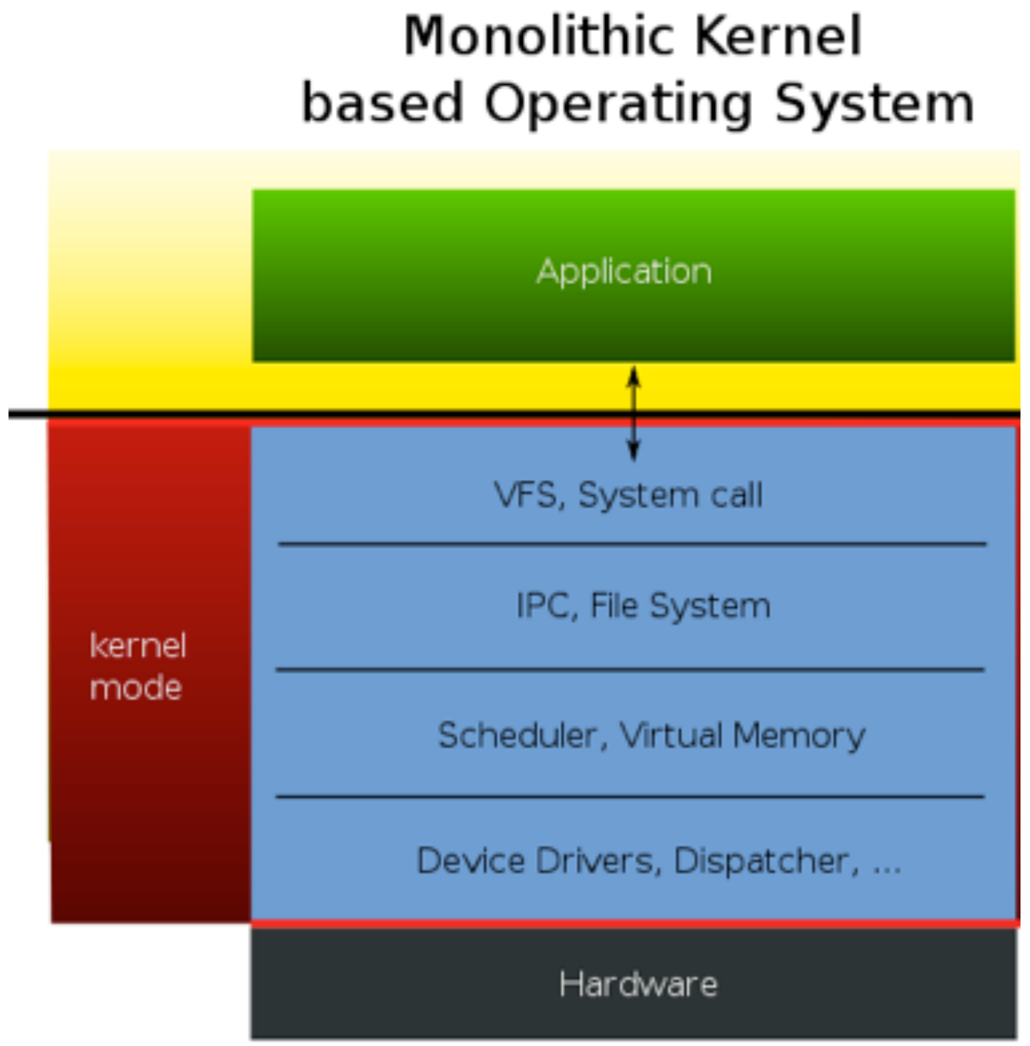
- Real Life Example
  - Policy: We want driver and passenger safety
    - Mechanism: Use seat belts
  - Another policy: Heavy loads (e.g., furniture) must be secured!
    - Mechanism: Use seat belts

Separation of Policy and Mechanism (2)

- OS Examples:
  - Policy: We want the OS to always re-gain control of the CPU
    - Mechanism: Interrupt handling system
  - Policy: We want to support multiprogramming
    - Mechanism: context switch
  - Policy: We want to protect OS code from users
    - Mechanism: dual-mode operation
  - Policy: We want FIFO (batch) job scheduling
    - Mechanism: Use a FIFO queue, pull next job from the head
  - Policy: We want Round Robin scheduling
    - Mechanism: Use a FIFO queue, pull next job from the head

Monolithic Kernel Design (Big Gov't)

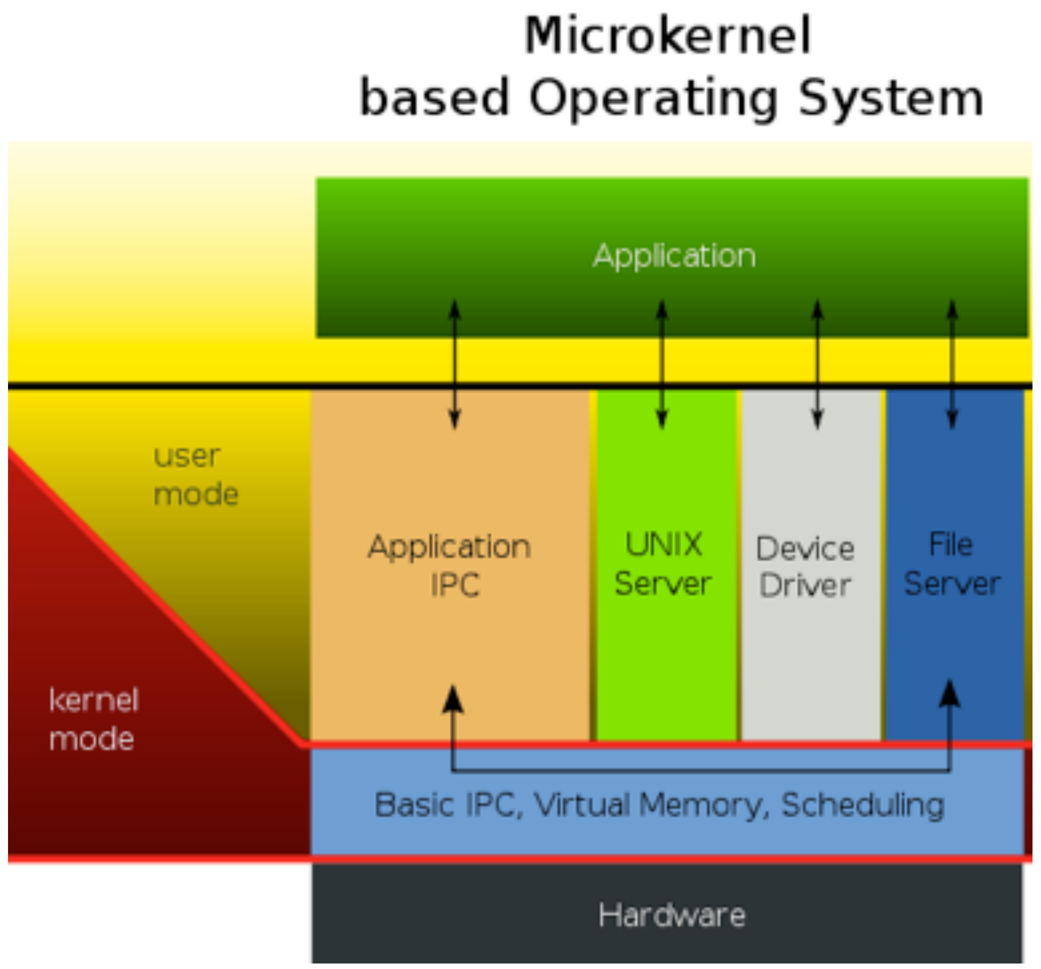
- Large Monolithic Kernel Design
  - All OS services are in kernel space
    - OS code is privileged and lives in own address space
  - All user code is unprivileged
  - All OS services are made through system calls.
- Multics, early Windows/DOS, UNIX


Monolithic Kernel based Operating System
Application
kernel mode
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware

Pros
- Low overhead *between* kernel services (no protection check)

Cons
- Large codebase for the kernel
- No isolation: bug anywhere in the kernel can crash the OS

Microkernel Design (Small Gov't)



Pros
- Much more secure OS code
- More flexibility (users control mechanism)

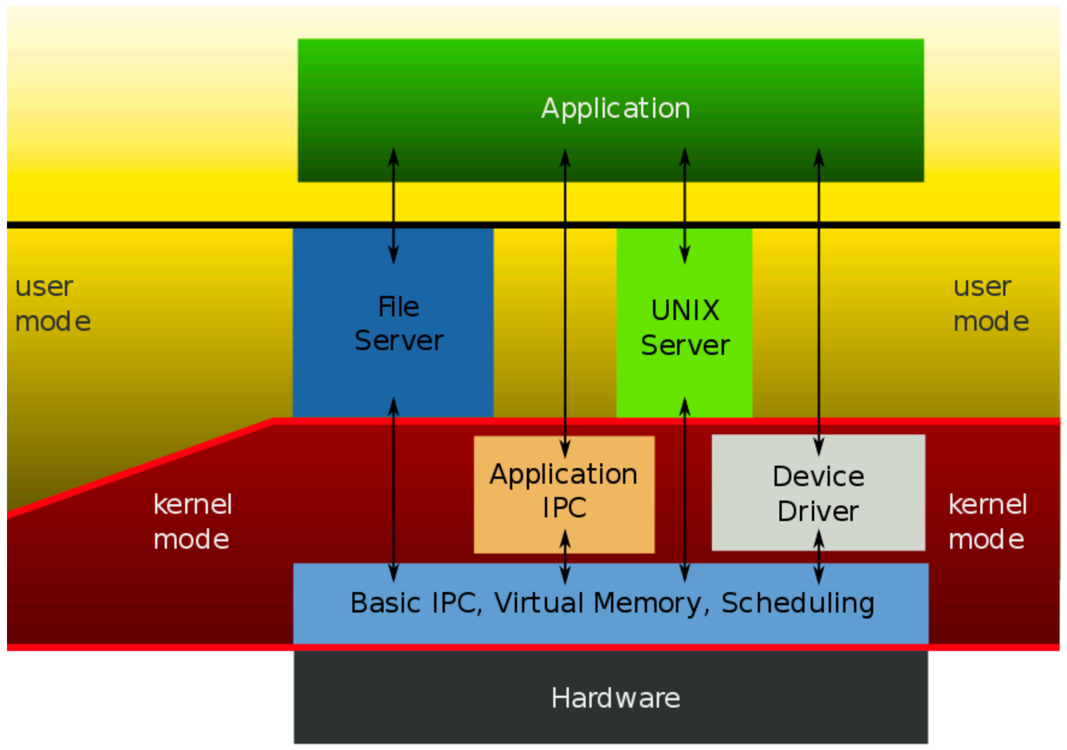
Cons
- Reduced performance
- Puts a lot of burden on programmers

- Extreme separation of policy and mechanism
  - Minimize what goes into the kernel
  - The kernel only know how to perform crucial tasks
- "Servers" are implemented in user space
  - User code talks to servers, servers access kernel when necessary
- Examples: CMU Mach, iOS, MINIX

Monolithic Kernel with Loadable Modules

- A compromise! Load kernel code as needed.
  - (Why load all device drivers? Why load code for all the shells?)
- Only load kernel modules that are essential for the system you're running
  - Service code can be loaded on-demand
  - Examples: Linux, MacOS, Windows.

"Hybrid kernel"
based Operating System



Pros
- Saves space like microkernel
- Still separates policy from mechanism
- Better performance than microkernels

Goals for Today

- Invoking the Kernel
  - Hardware Interrupts
  - Software Interrupts (Traps, system calls)
  - Protection: Dual Mode Operation
- Conclusion

In Conclusion...

- Know these:
  - Kernel is just another program (the one that's always running)
    - Needs started, but how? (bootstrap)
    - Needs to regain control of CPU every once in a while (timer interrupt)
    - Needs invoked when providing services (interrupts, traps)
  - Protection: Know kernel mode vs. user mode
    - And how to cross between the two modes
  - Policy vs. mechanism: why separate these?
  - Kernel design philosophies

Administrivia 1/31

- Reminders:
  - Read Chapter 2 (Dino book)
  - Hwk 3 due next Friday, 2/9
- Last time...
  - Batch processing, multiprogramming, timesharing
  - Success of UNIX
- Today:
  - Invoking the OS
  - Interrupts
  - Dual-mode operation and system calls

Administrivia 2/1

- Reminders:
  - Hwk 3 due in a week
  - Reading to Chapter 2.7 inclusive (Dino book)
- Last time...
  - Booting the OS
  - Waking up the CPU
  - Interrupts
- Today
  - Policy vs. Mechanism
  - Traps (intentional interrupts by users) and system calls.
  - printf() example

Administrivia 2/5

- Reminders:
  - Hwk 3 due Friday
  - Reading Chapters 2.8-2.9 & Start Chapter 3
- Last time...
  - Traps (intentional interrupts by users) and system calls.
  - A system call example in printf()
- Today
  - Policy vs. Mechanism and Kernel Design Principles
  - Start Chapter 3
    - Concurrency vs. Parallelism
    - Process Management