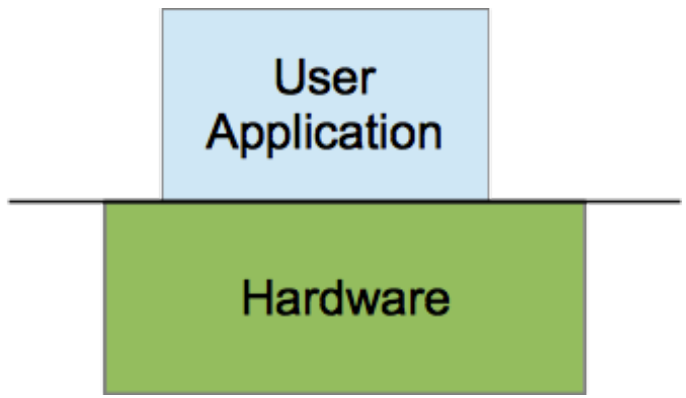
OS and Architecture Overview

Chapter 1
Chapter 12, 12.1 – 12.5

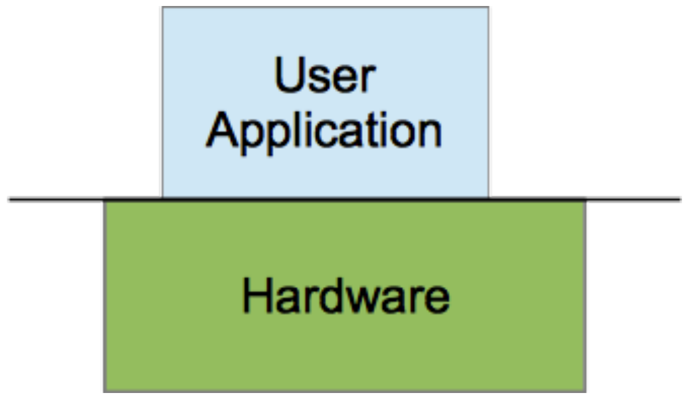
Running Programs

- We could run our computer system like this



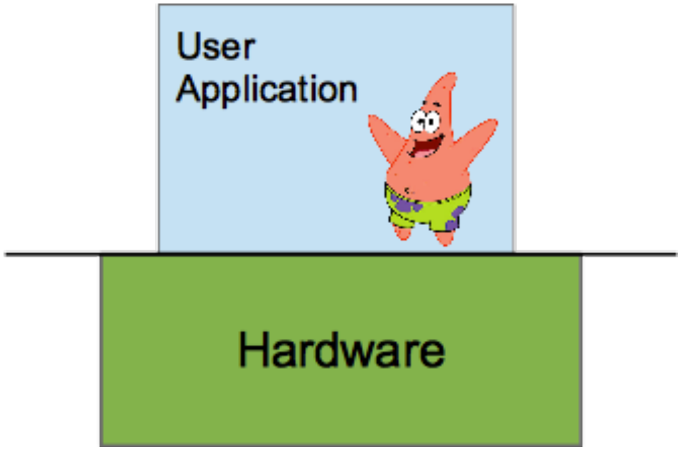
- But there would be some disadvantages …

Running Programs



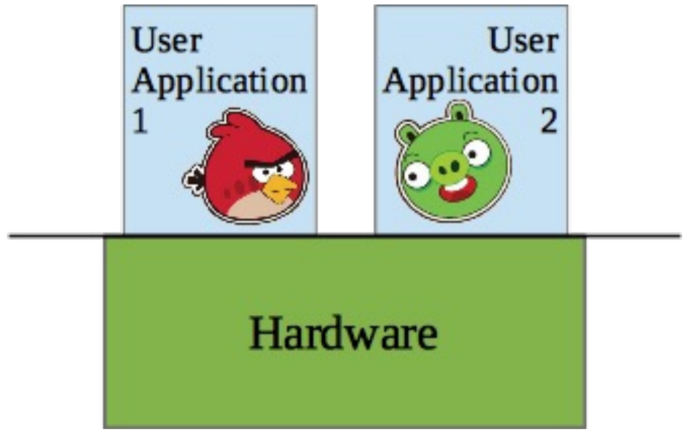
- Applications would be less portable.
  - Harder to move them to different types of hardware
- And they would be more complex.
  - Need to know how to talk to specific hardware

Running Programs



- Some applications might not know how to get the most out of the hardware.

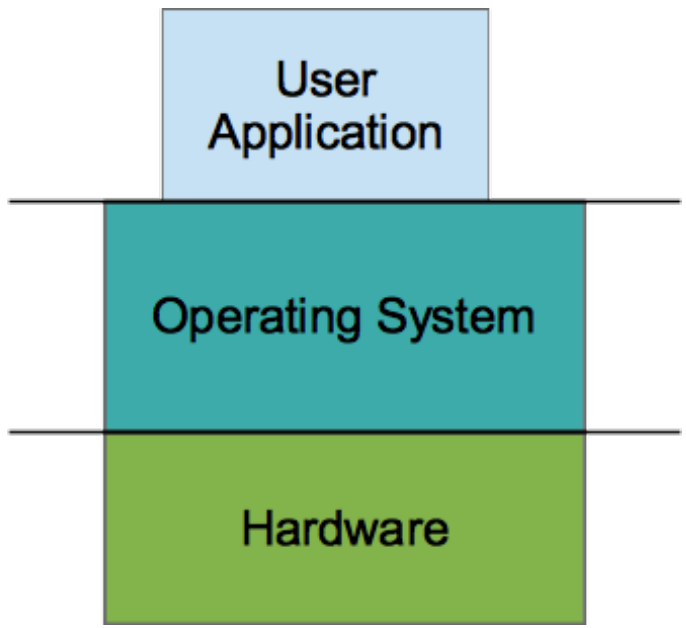
Running Programs



- Applications might not cooperate to share the hardware.
  - A malicious or faulty application could ruin things for everyone.
  - This would be a significant security vulnerability.

Role of the OS

- A Modern OS can help.
  - Serves as a layer separating applications from the hardware.



What is an operating system?

- A software layer
  - between hardware and user applications
- Takes the difficult-to-use, easy-to-break interface offered by various hardware components and turns it into something *virtualized*
  - easy to use (hides complexity)
  - safe (prevents and handles errors)
- Acts as *resource manager*
  - allows programs/users to share hardware resources
  - in a protected way: fair and efficient

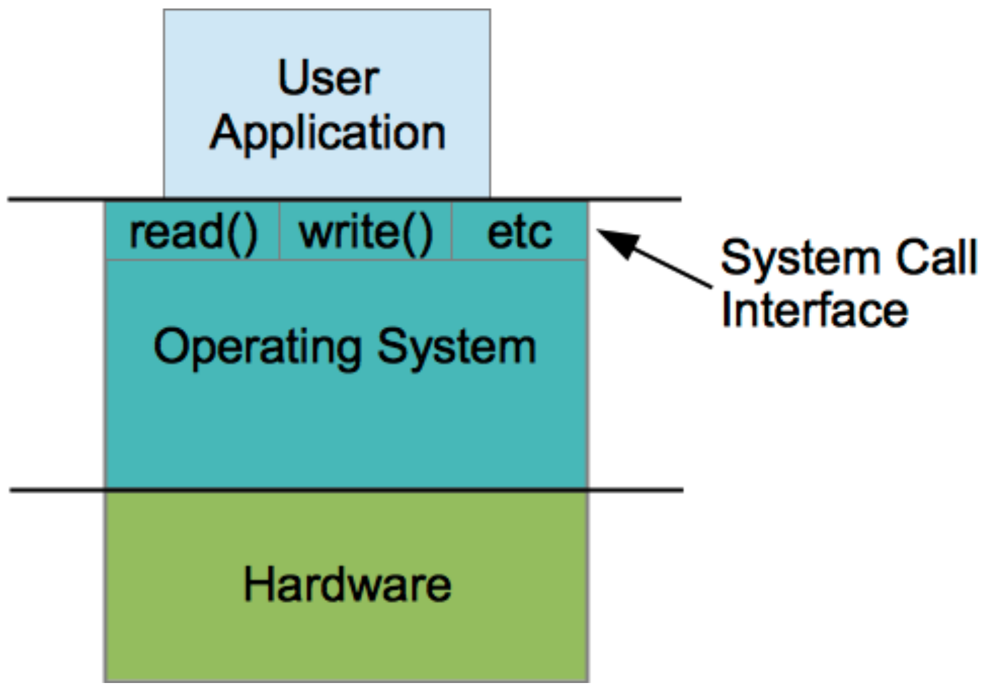
Windows

Linux

MAC OS

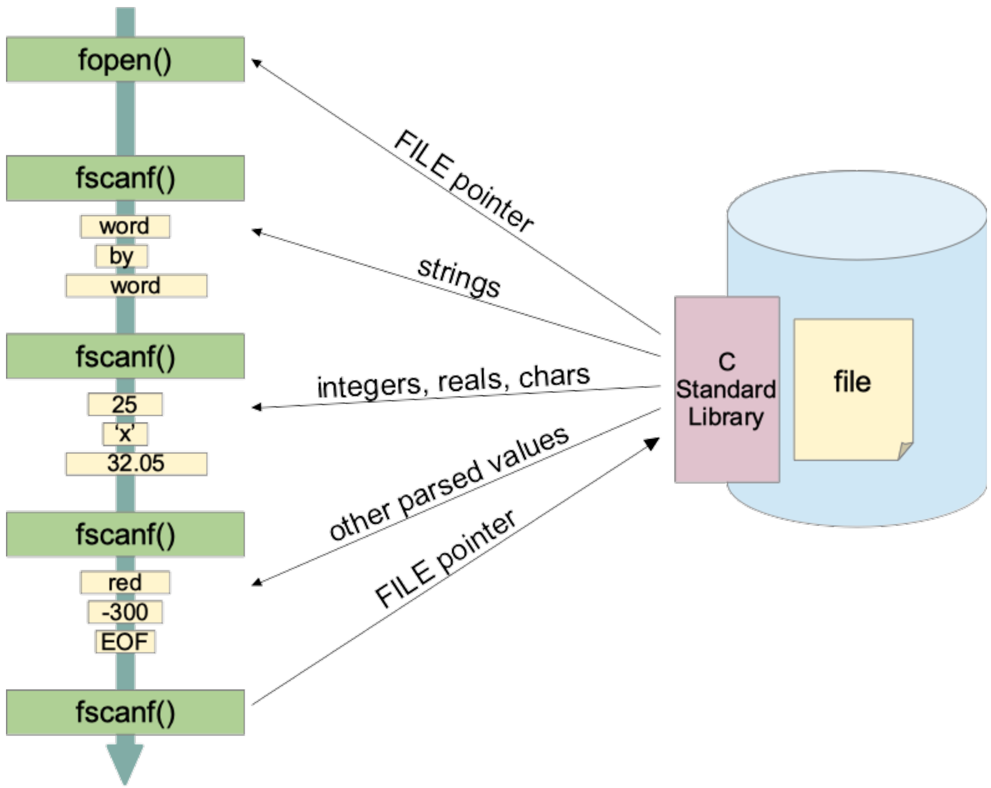
Using the OS

- What does the OS interface look like?
  - It's a collection of *system calls*.



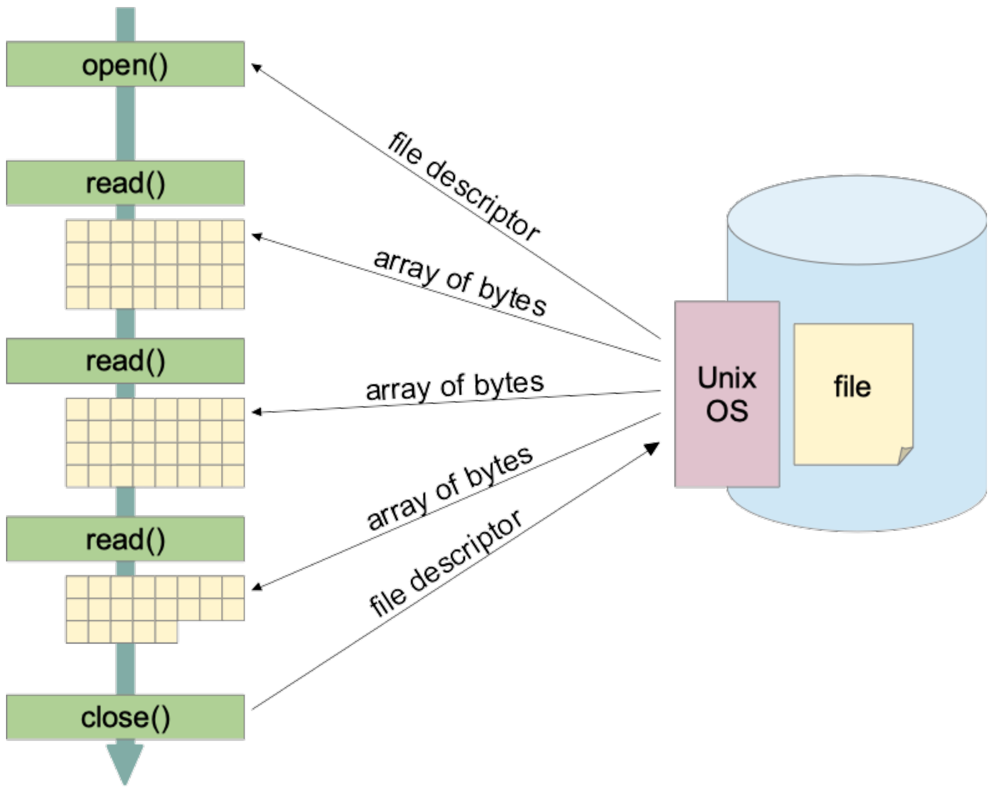
Using the Standard Library

- Normally, a C program might perform I/O through the C standard library



Talking to the OS

- System calls are a lower-level interface
  - For talking directly to the OS.



Unix / POSIX Example

- Let's use the system-call interface to read a file.
  - A text file, printing it out to the terminal as we read.
  - This is **readFile.c**
- We'll need three system calls.

```
int open( char *filename, int flags, mode_t flags );
```

You get back a ***file descriptor***, a small integer that's your ***handle*** for the file.

Or, -1 if it fails

Here's a common trick, a bitwise or of multiple flags.

This is optional, mode bits (permissions) if you're creating the file.

Unix / POSIX Example

- At this level, reading a file is byte-oriented.

```
int read( int fd, void *buffer, int count );
```

How many bytes you got

File you want to read from.

Where you want the bytes to go.

Number of bytes you'd like to try to read.

- When you're done, release the resource.

```
int close( int fd );
```

Unix / POSIX Example

- readFile.c uses system calls and C library calls.
- You can compile it like any other C program.

```
gcc -Wall -std=c99 readFile.c -o readFile
```

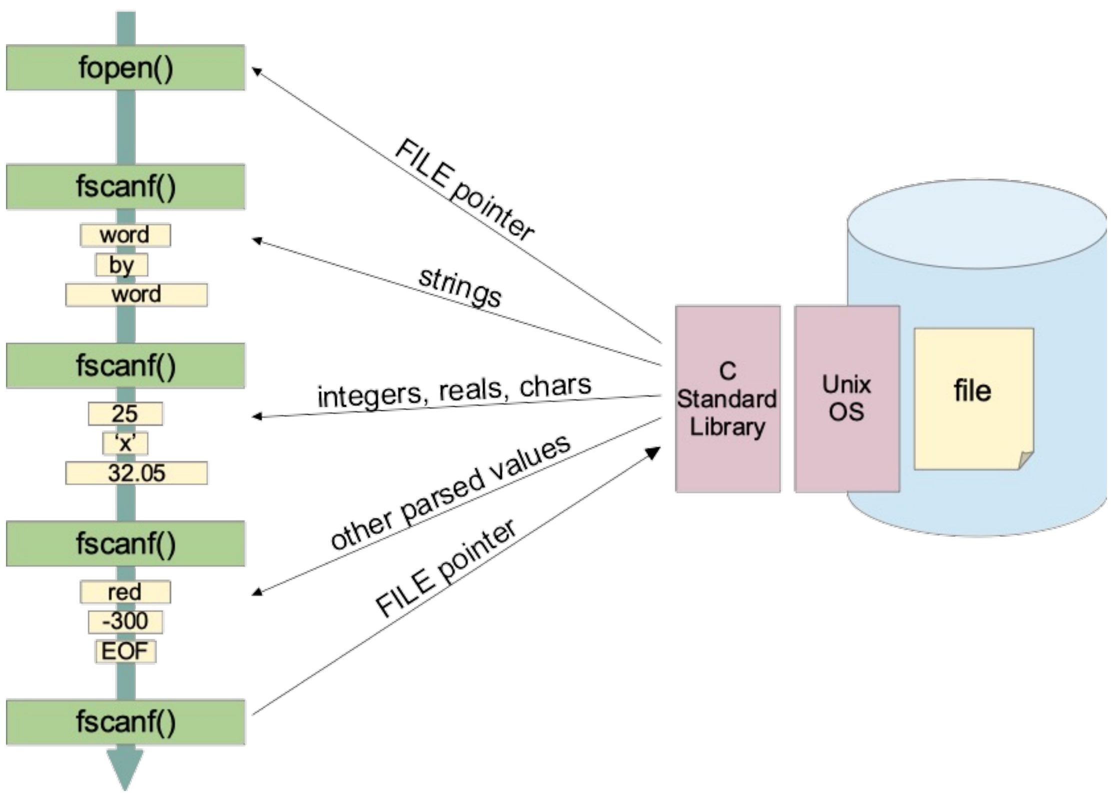
```
./readFile
```

- The shell command, strace, can show all the system calls:

```
strace ./readFile
```

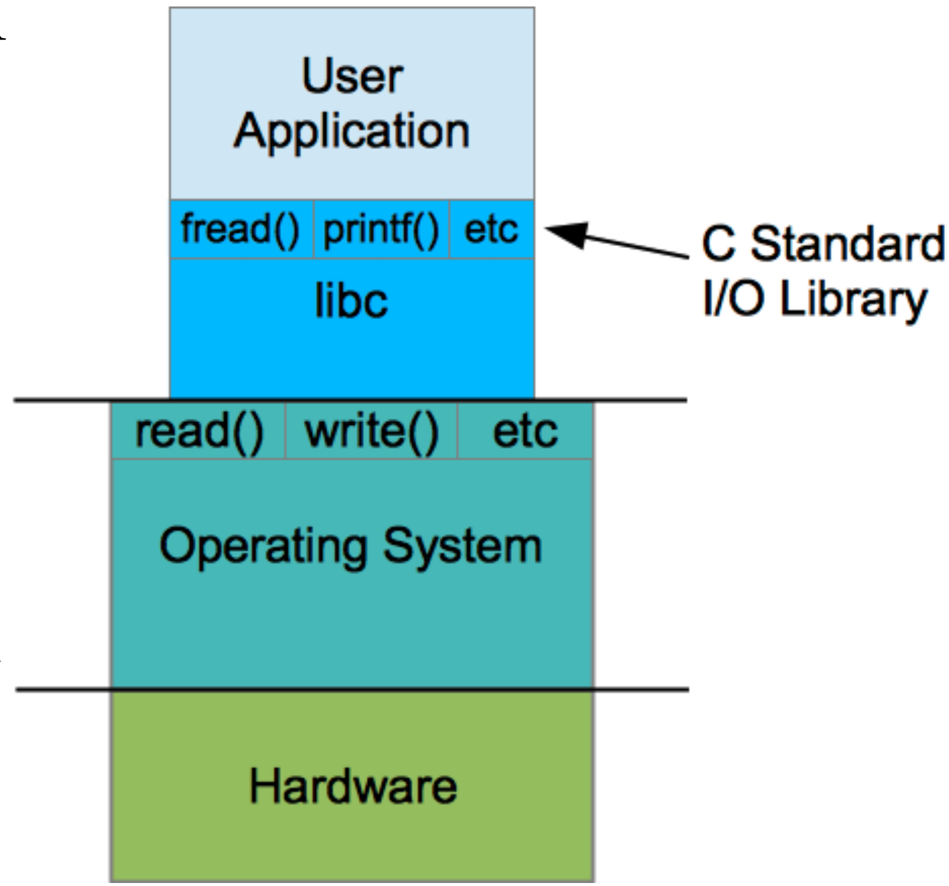
Library Functions and System Calls

- Internally, the C library uses the same system calls to implement I/O.



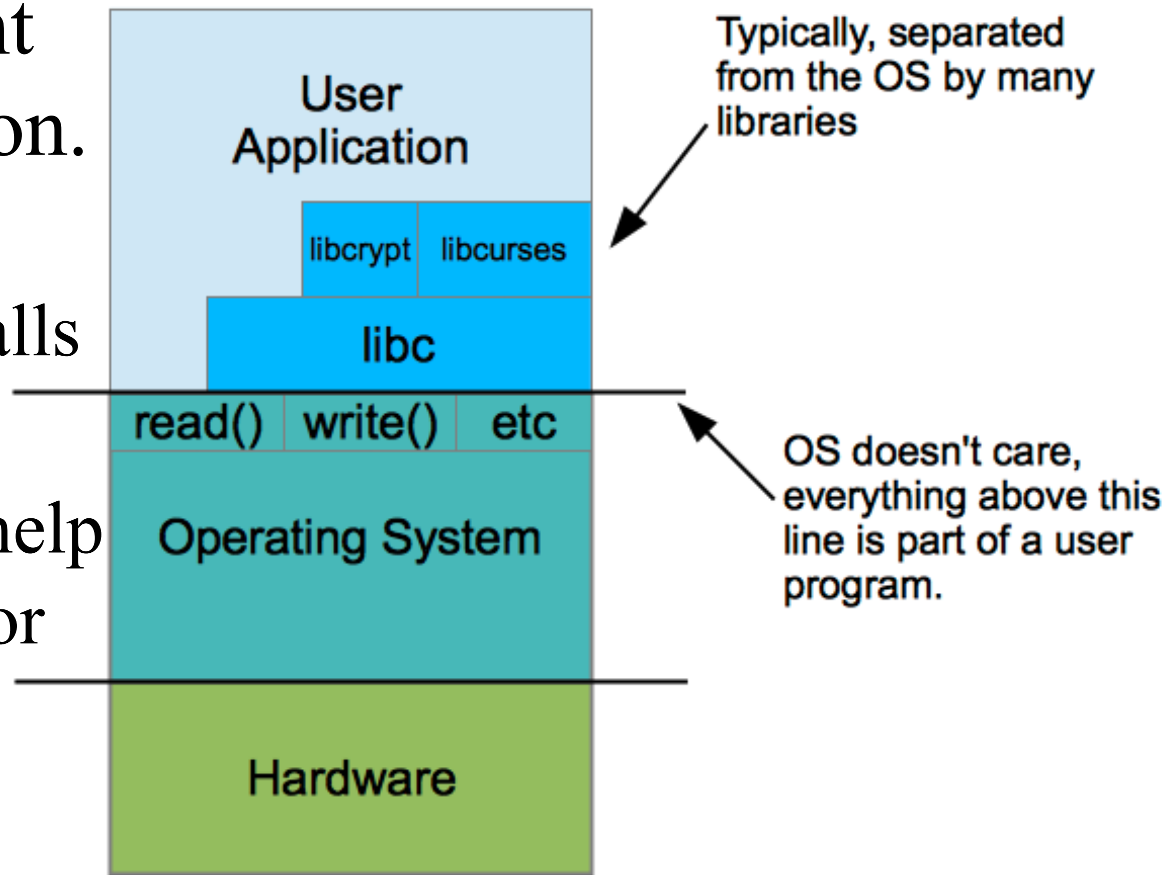
Using the OS

- Often, a program will access the OS indirectly
  - Through one or more libraries
  - The program calls a library function … and that library might make system calls to the OS.



Using the OS

- A program might use a combination.
  - Making some direct system calls to the OS.
  - … and getting help from libraries for other things.



System Calls for Running Programs

- The OS provides system calls for lots of different things.
  - Reading and writing files
  - Starting other programs
  - Allocating and configuring memory
  - Communication between programs
  - Monitoring the state of the system
  - …

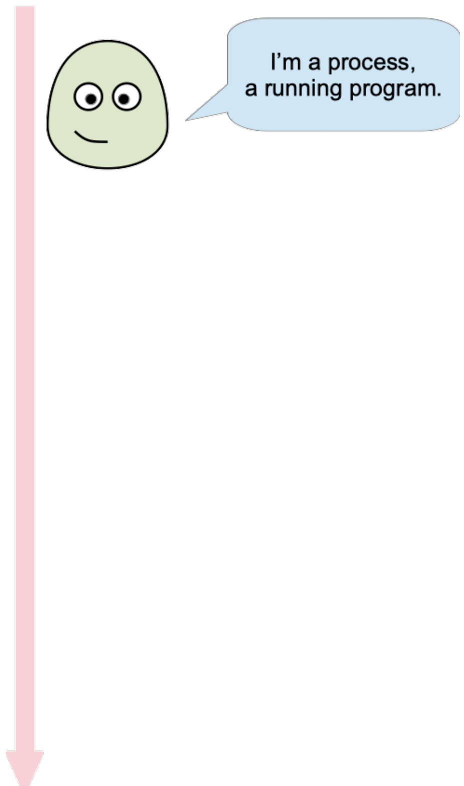
System Calls for Running Programs

- The OS provides system calls for lots of different things.
  - Reading and writing files
  - Starting other programs
  - Allocating and configuring memory
  - Communication between programs
  - Monitoring the state of the system
  - …

Let’s look at this one.

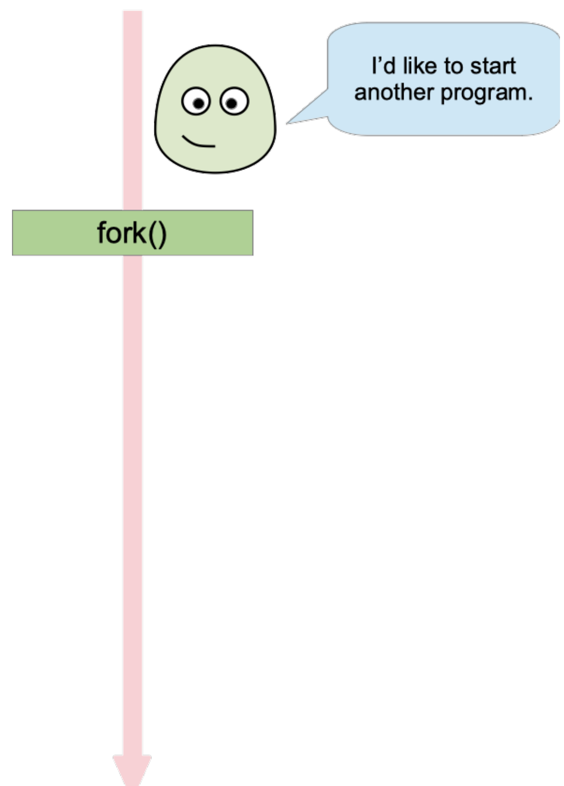
A Running Program

- The OS creates an environment for running programs (*processes*).



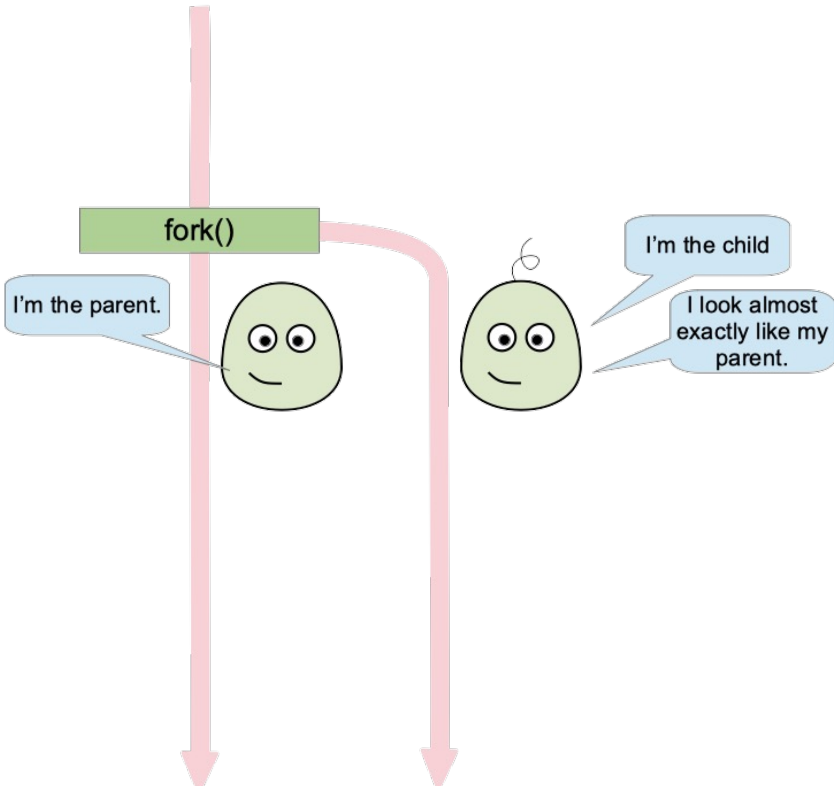
Starting a Program

- A process can create other processes.
  - That’s what fork() does.



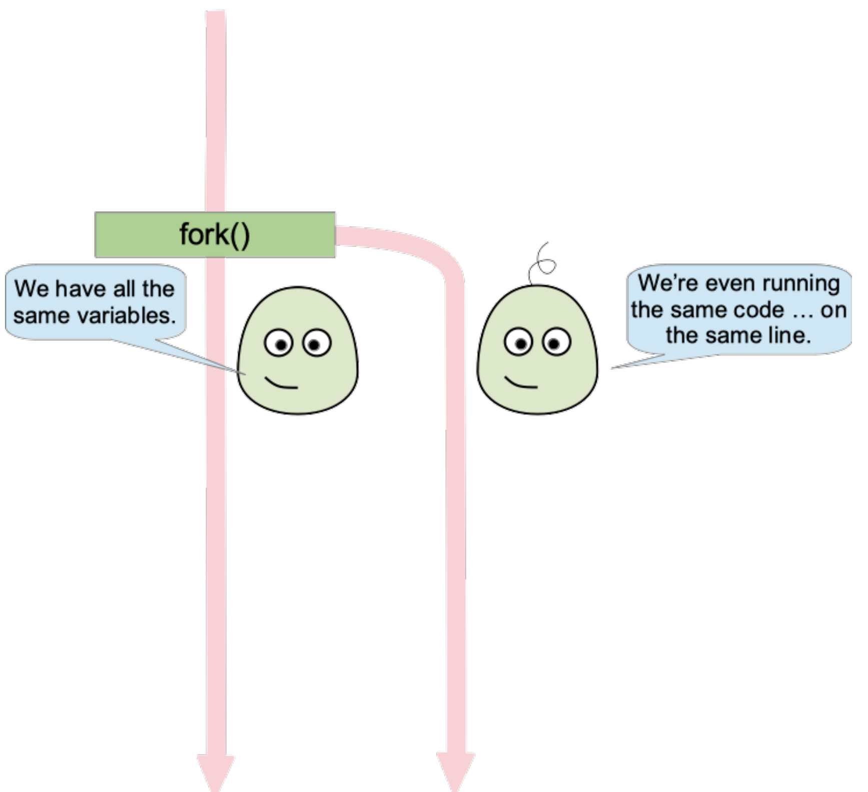
Starting a Program

- fork() makes a *child* process.
  - One that looks (mostly) like a copy of the *parent*.



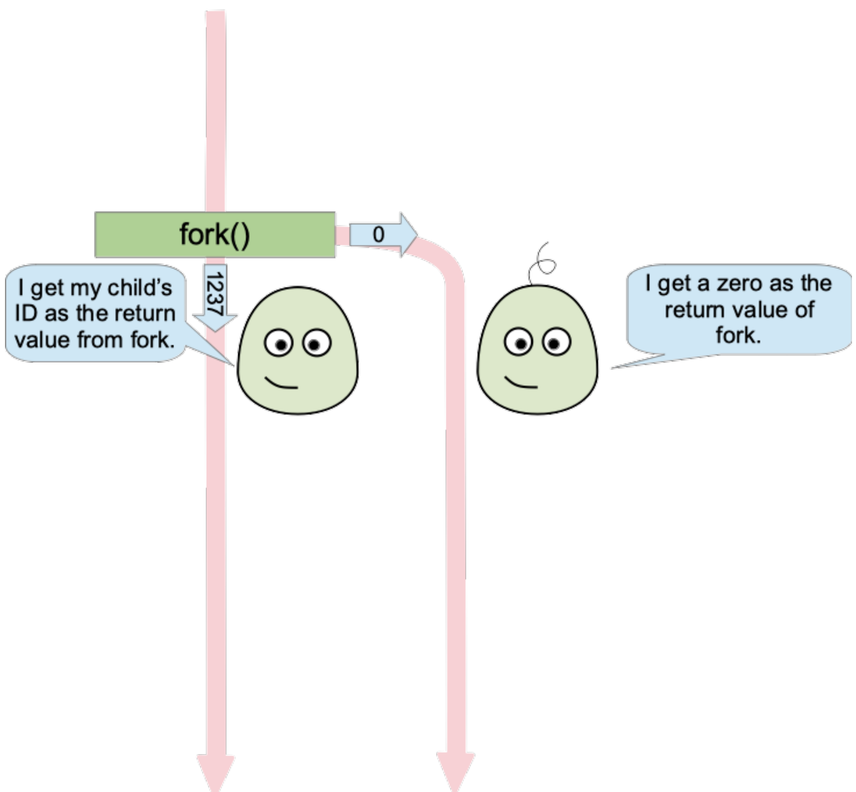
Starting a Program

- The child has a copy of the parent's memory.
  - Same variable values, running on on the same line of code.



Who's Who

- The parent and child get different return values from fork()

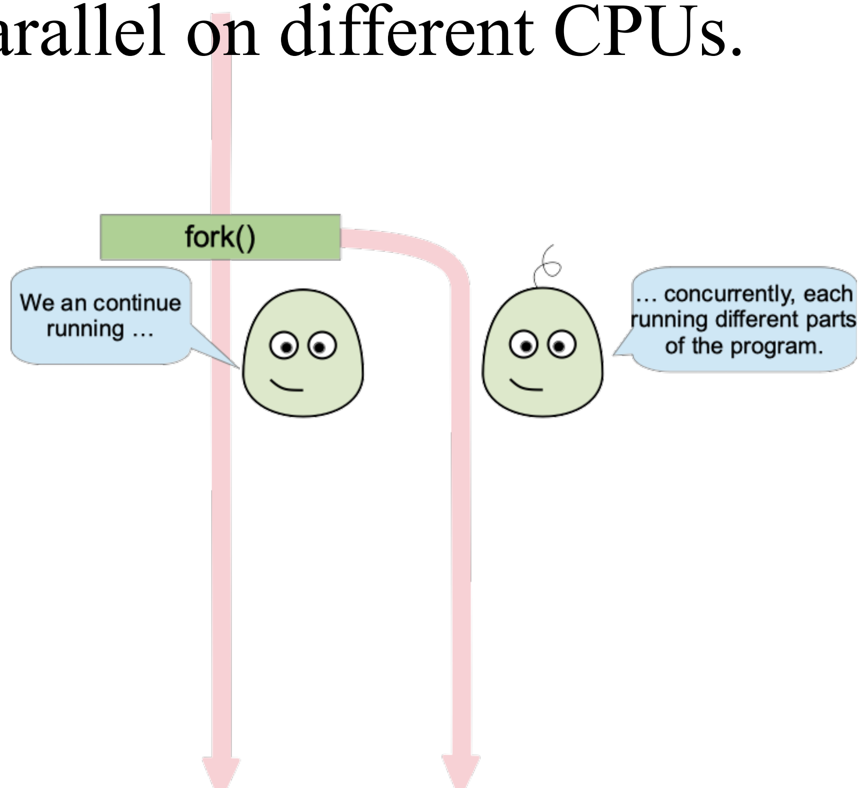


Unix / POSIX Example

- Typically, we use three system calls to run a separate program.
  - `int fork( );`
    This makes a copy of the running program (a child)
  - exec*()
    This gets a process (e.g., the child) to run a different program.
  - wait()
    This makes a parent wait for one of its children to terminate.

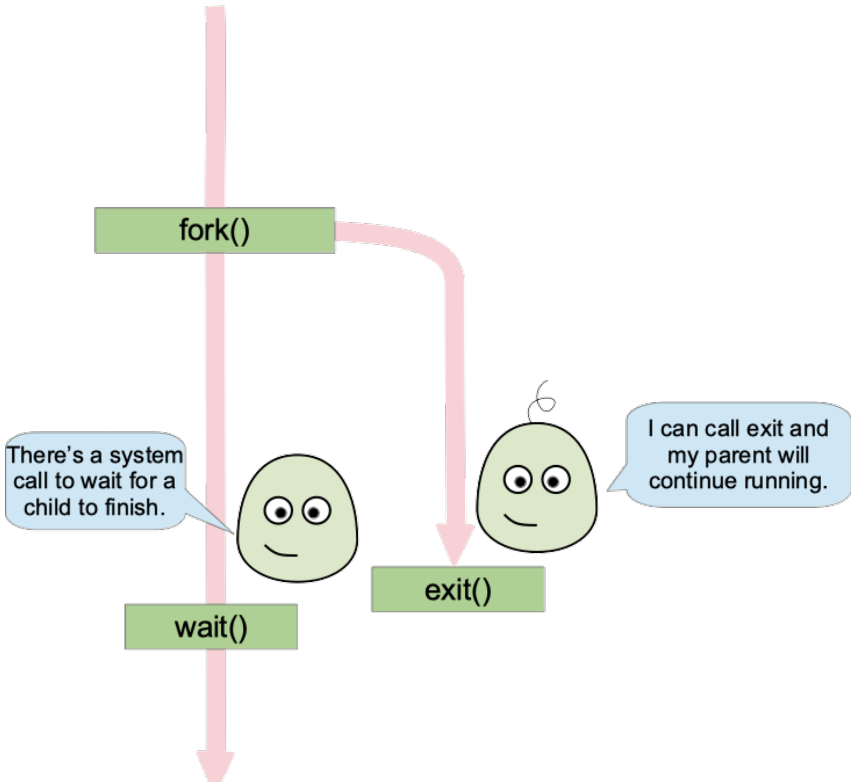
Concurrent Execution

- The parent and child can run concurrently.
  - Taking turns on a single CPU.
  - Or, in parallel on different CPUs.



Waiting for your Child

- A parent might wait for a child to finish.
  - There's a system call for that.



Unix / POSIX Example

- There's a call to terminate the current program.

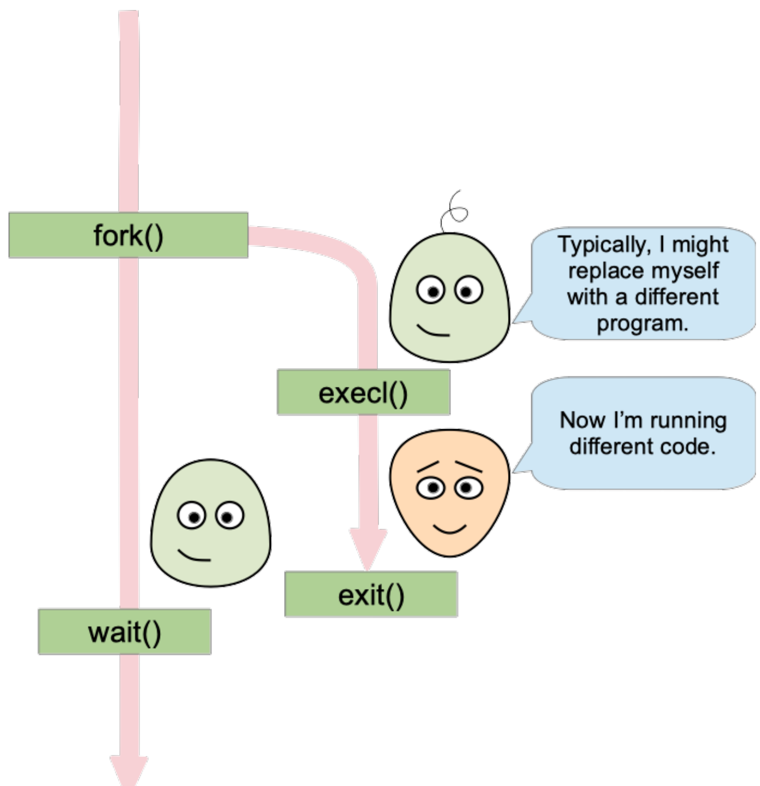
```
void _exit( int status );
```

- There's a call to wait for any one of your children to terminate.

```
int wait( int *status );
```

Running another Program

- The child can continue running the same code
- … or, typically, it might run a different program.



POSIX exec* Calls

- exec*() replaces the running program with a different one
  - It's usually called after a fork()
    to get the child to do something different.
  - Lots of different forms, for different ways to start

Path to the program you want to run.

Command-line arguments.

End marker, for the list of arguments.

```
int execl(char *path, char *arg0, char *arg1, …, null );
```

Returns -1 on failure.
Otherwise, never returns.

Starting a New Program

```
int main( int argc, char *argv[] ) {
  pid_t id = fork();

  if ( id == -1 )
    fail( "Can't create child" );

  if ( id == 0 ) {
    execl( "/bin/ls", "ls", "-l", NULL );
    // If successful, execl() never returns.
  } else {
    wait( NULL );
    printf( "Done\n" );
  }

  return EXIT_SUCCESS;
}
```

I'm the child.

child's pid

I'm the parent.

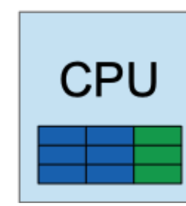
等 child

Architecture Refresher

- How does the OS do this?
- Let’s look at what computer hardware is like, logically.

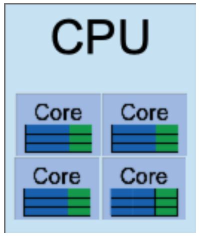
Thinking about Hardware

- You have a CPU (Central Processing Unit)
- It executes low-level machine instruction really, really fast.
- It has a small-ish collection of registers to keep up with what it's doing.
  - Like what instruction it's running
  - … where important things are in memory
  - … temporary space for computations



Thinking about Hardware

- Really, most modern systems have multiple CPUs or at least multiple cores.
  - Each with its own set of registers.
  - Running its own instructions.
  - So, a typical system can run multiple program at the same time.

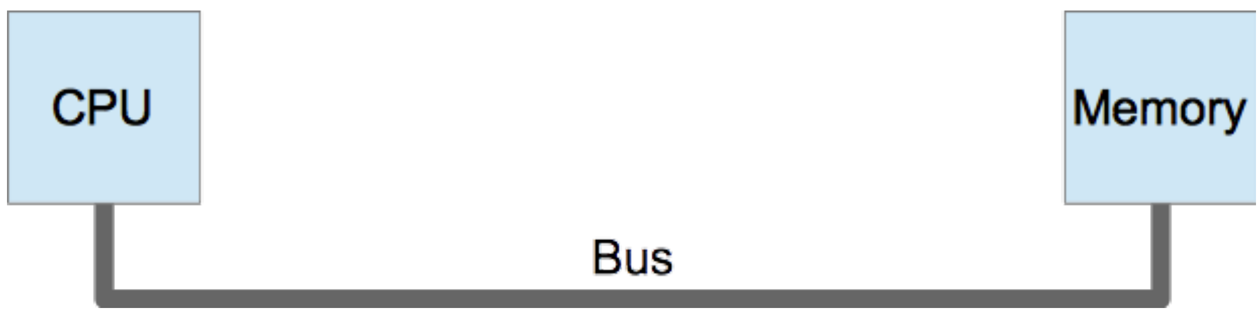


Thinking about Hardware

- You have main memory
  - For storing running programs, their data and the OS itself.

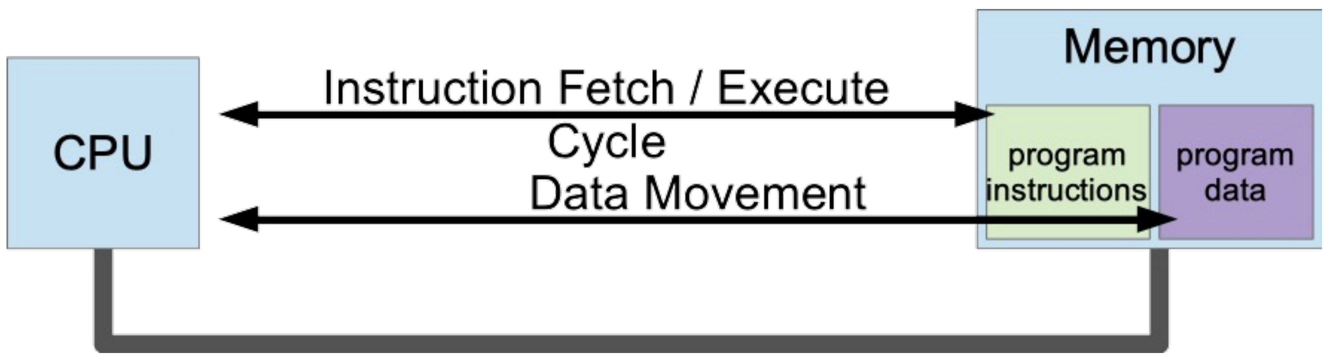
Thinking about Hardware

- You have a bus, to move data between the CPU and memory.



Thinking about Hardware

- This is your typical *Von Neumann architecture*
  - Code you're running lives in main memory
  - … along with the data it's working with.



Thinking about Hardware

- A typical CPU is much faster than main memory
- We can use a cache, to reduce delay of going to main memory all the time.

General Idea: Caching

- Caching
  - Use a faster, smaller type of storage
    - This is called the *cache*
  - To temporarily hold the subset of what's in a larger, slower storage area
- On access, first check the cache
  - If information is there, a *cache hit*, you're done ☺
  - If not, a *cache miss*, go to the backing store 😐
  - Maybe copy to cache to speed the next access

General Idea: Caching

- Why does cache work?
  - *Temporal Locality* : a program is likely to access data it has accessed recently
  - *Spatial Locality* : a program is likely to access data nearby to what it has accessed recently
- Remember, cache is too small to hold everything
  - Need an efficient way to find what's in the cache
  - May have limitations on what values it can contain at once (associativity)
  - Need a *cache management policy*
  - ... including a policy for *replacement* when the cache is full.

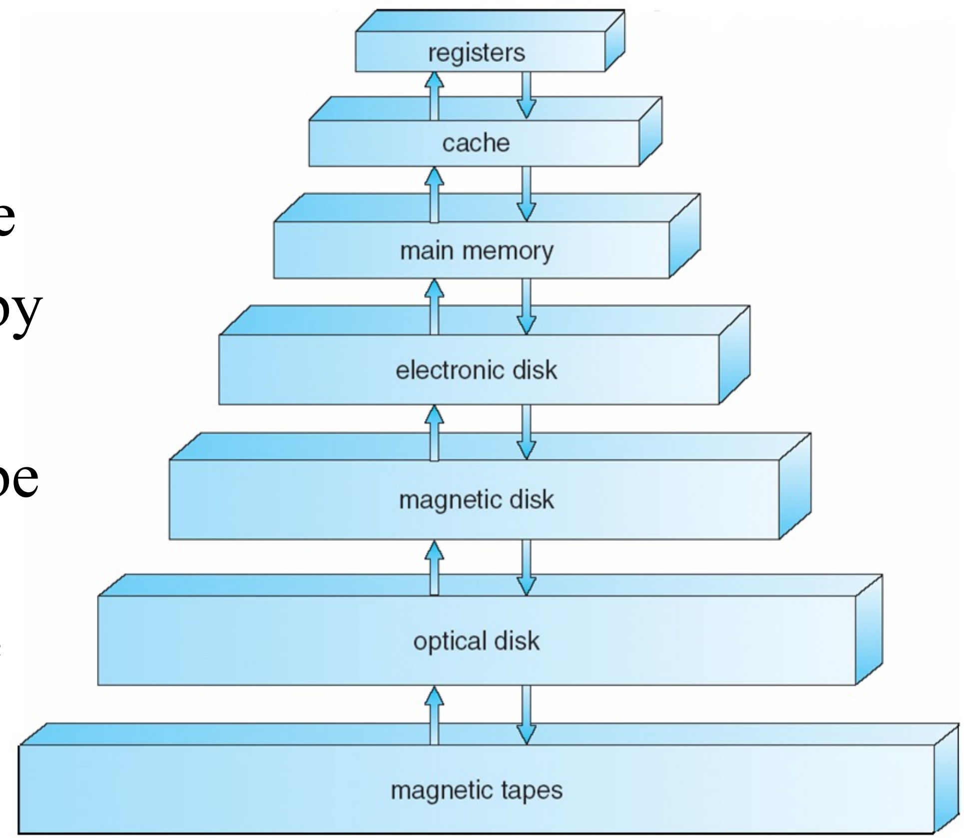
General Idea: Caching

- As cache contents are updated, contents of backing store may get out-of-date.
- Need a *write policy* to make sure backing store gets updated (eventually)
  - *Write-through*
  - *Write-back*
- Where is Caching Used?
  - By the hardware, to speed access to main memory
  - We can use this same technique elsewhere
  - … whenever we have a big, slow storage area but we just need some of its contents at a time.

Devices Responsible for Storage

A *Storage Device Hierarchy*

- From fast, small, managed statically by the compiler
- To slower, larger, managed dynamically by the hardware
- To slower, larger, managed by the OS
- To even slower, larger, maybe managed by the user
- Some are *volatile*, others are *non-volatile*

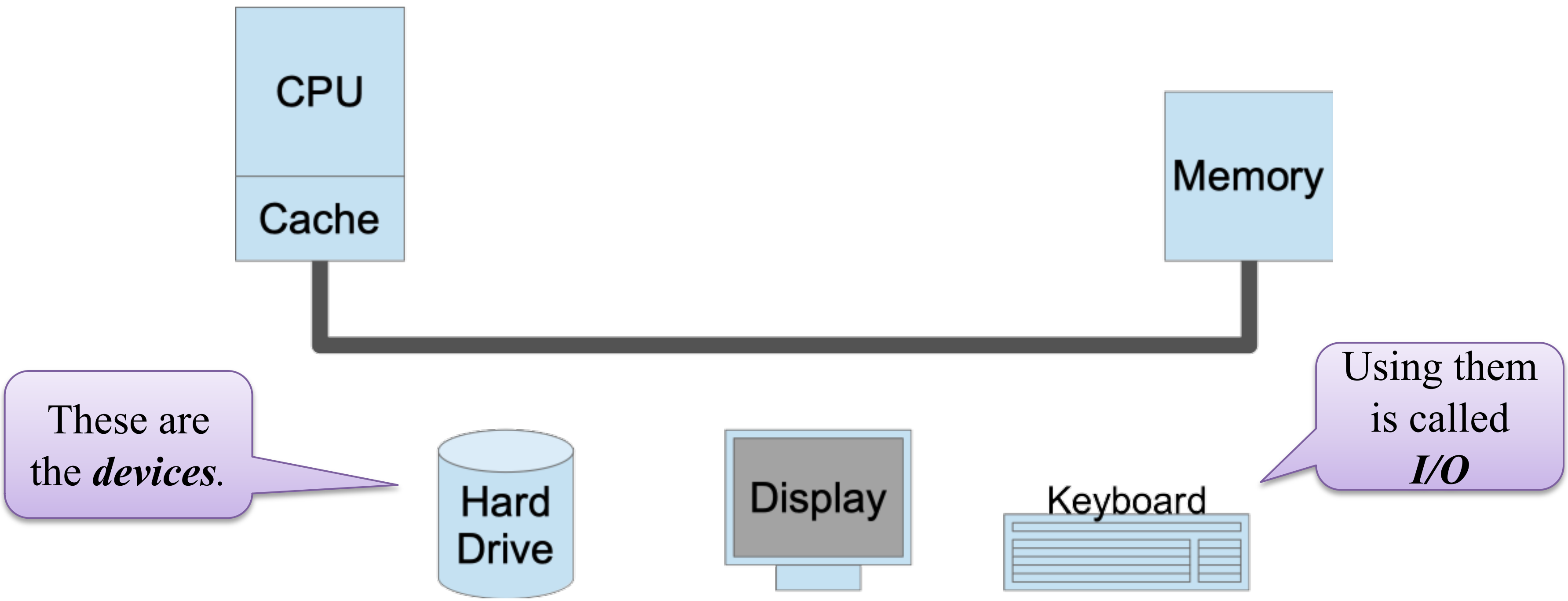


Typical Performance

Level	1	2	3	4
Name	Registers	Cache	Main memory	Disk storage
Typical size	< 1KB	> 16MB	> 16 GB	> 100GB
Access time (ns)	0.25-0.5	0.5-25	80-250	5000
Bandwidth (MB/sec)	20,000-100,000	5000-10,000	1000-5000	20-150
Managed by	compiler	hardware	Operating system	Operating system

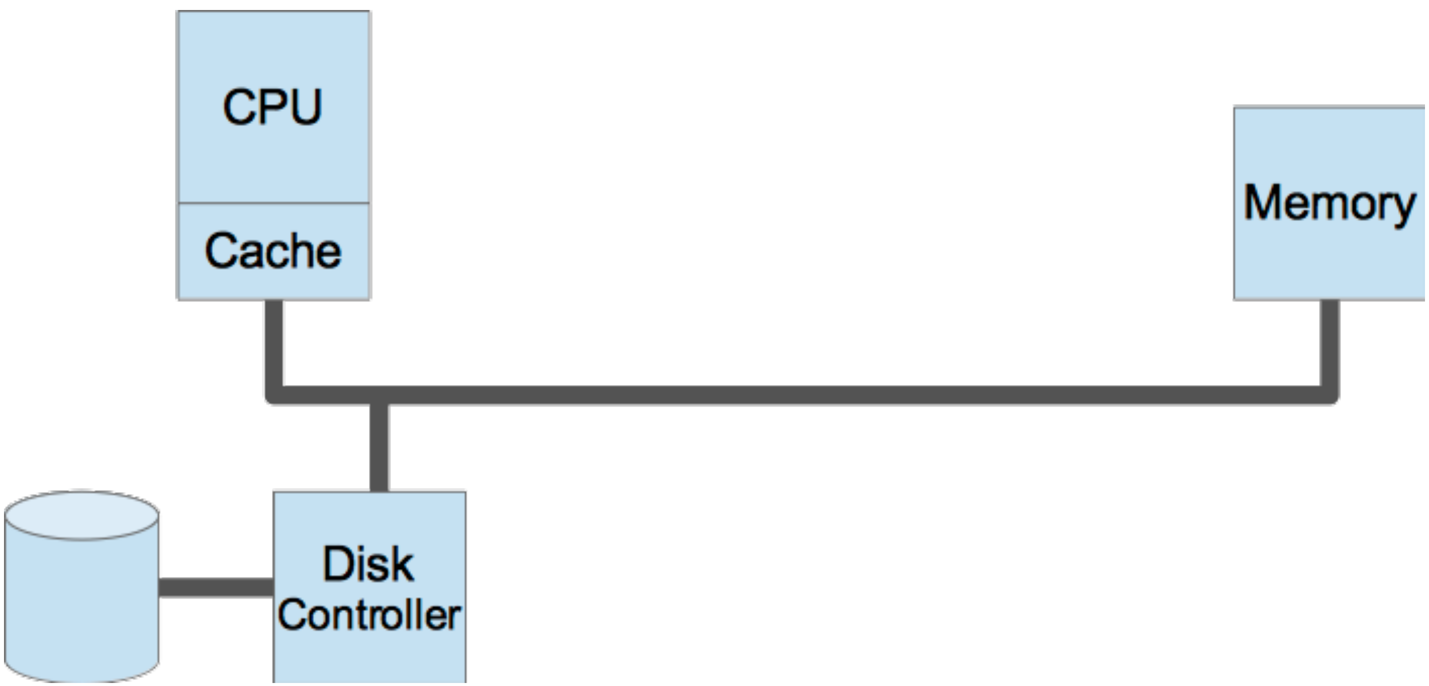
Thinking about Hardware

- A computer typically contains devices
  - To provide *non-volatile, secondary* storage
  - Or maybe interact with the outside world



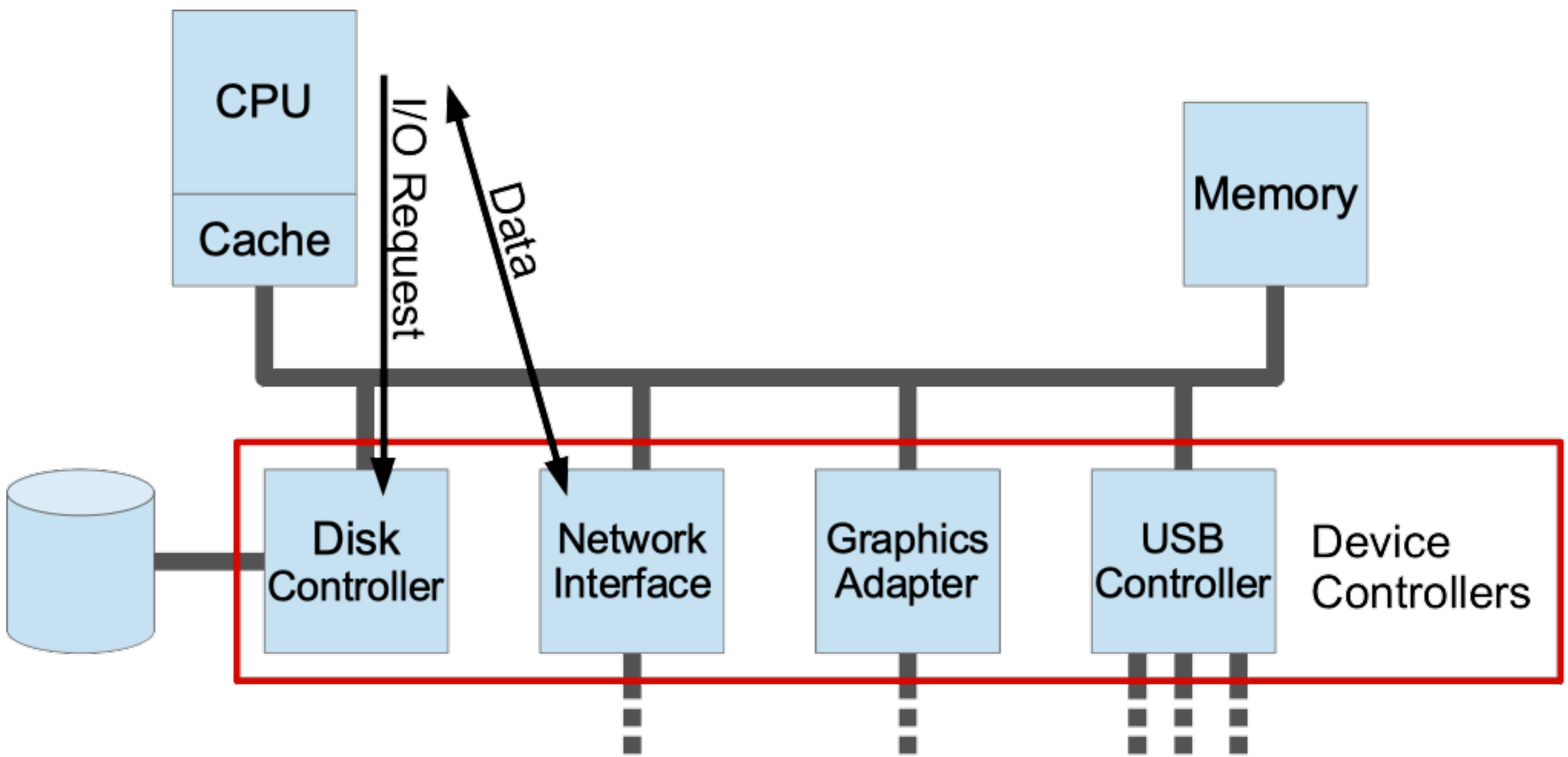
Thinking about Hardware

- The CPU doesn't interact with these devices directly.
  - There's normally a *device controller* responsible for communicating with the device.



Thinking about Hardware

- The CPU typically communicates with a device controller via the bus (or one of the buses)

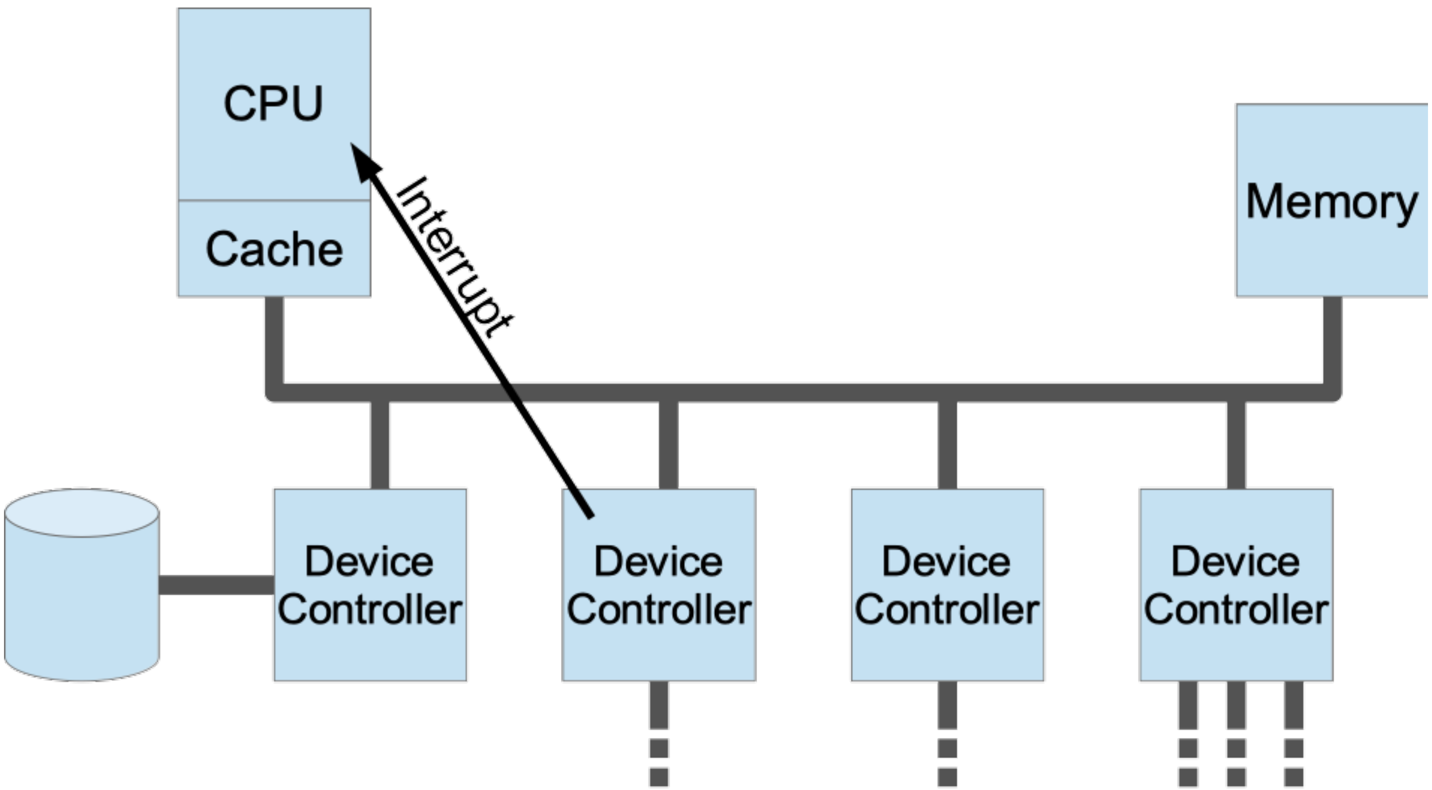


Performing I/O

- Device controllers are smart.
- You can tell them what to do … and they'll complete the task automatically.
  - You may even be able to queue up multiple requests.
- How can the CPU know when they're done.
  - It could check a status register on the device
  - … over and over.
  - This is called *polling*. Could be inefficient.
- Instead, how about …

Interrupts

- Hardware provides a mechanism for this.
  - A way for a device controller to get the CPUs attention.

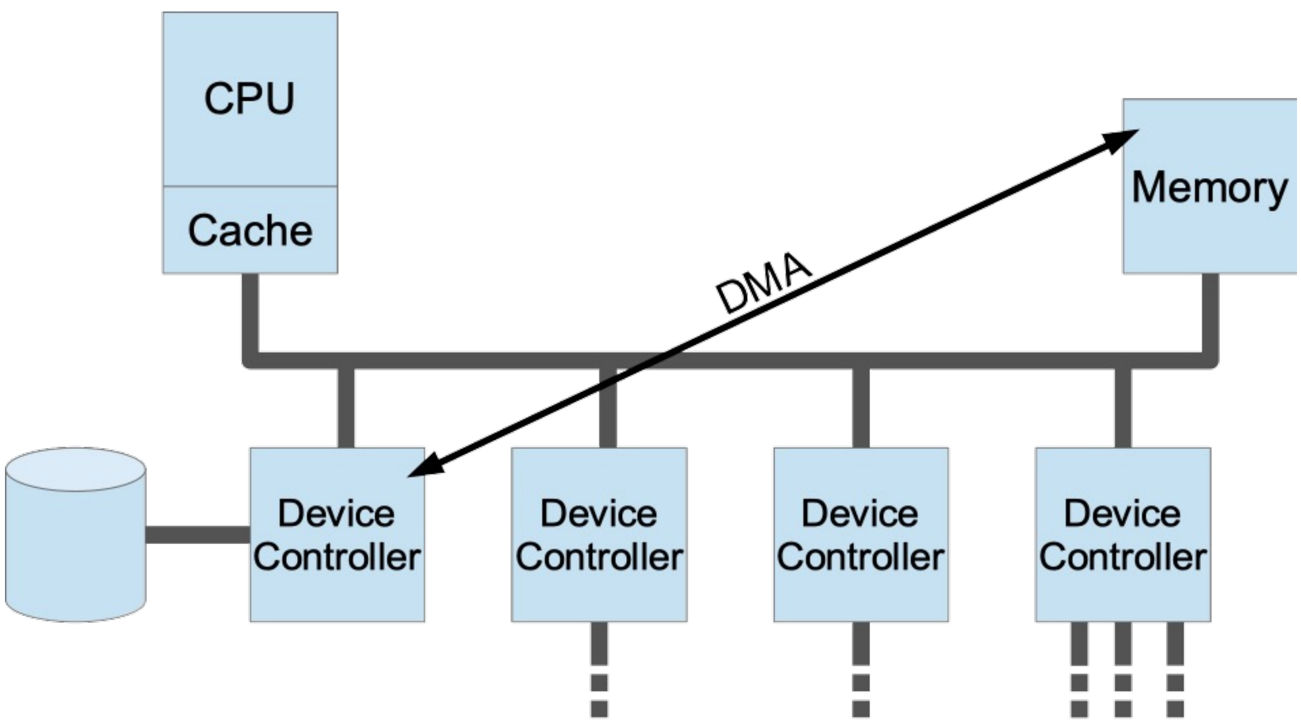


Efficient I/O

- For a bulk I/O device, how does it get the data it needs?
  - Like, a hard disk that needs to write a whole block of data?
- The CPU could write the data one byte at a time into a data-in register.
  - Could be inefficient … and a waste of CPU time.

Efficient I/O

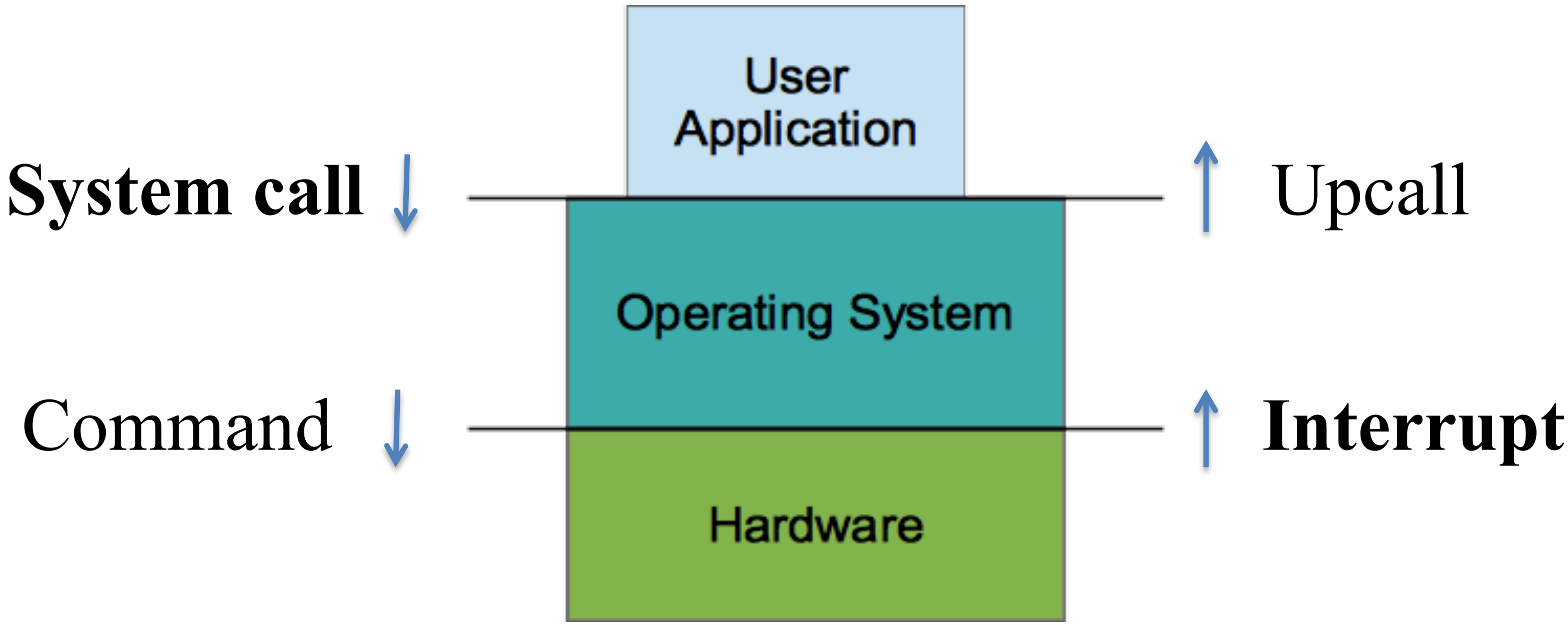
- Direct Memory Access (DMA)
  - Bulk data transfer between Device Controller and memory
  - So, a device controller can use the bus



Efficient I/O

- Why would we want this?
  - Program the device controller to start an I/O operation
  - CPU is notified (via interrupt) when I/O is done

How does an OS work?



- *System call* : OS receives requests from application
- *Command* : OS makes requests to hardware
- *Interrupts* : OS is notified when hardware needs service
- *Upcall* : OS can notify application of events of interest

Hardware Protection

- We need user programs that are willing to play nice
  - Share resources
  - Not touch each other's stuff
- You and I can write some really bad programs
- OS needs to prevent them from doing harm
- We need various types of *hardware protection*
  - Dual-Mode Operation
  - Memory Protection
  - I/O Protection
  - CPU Protection

Dual-Mode Operation

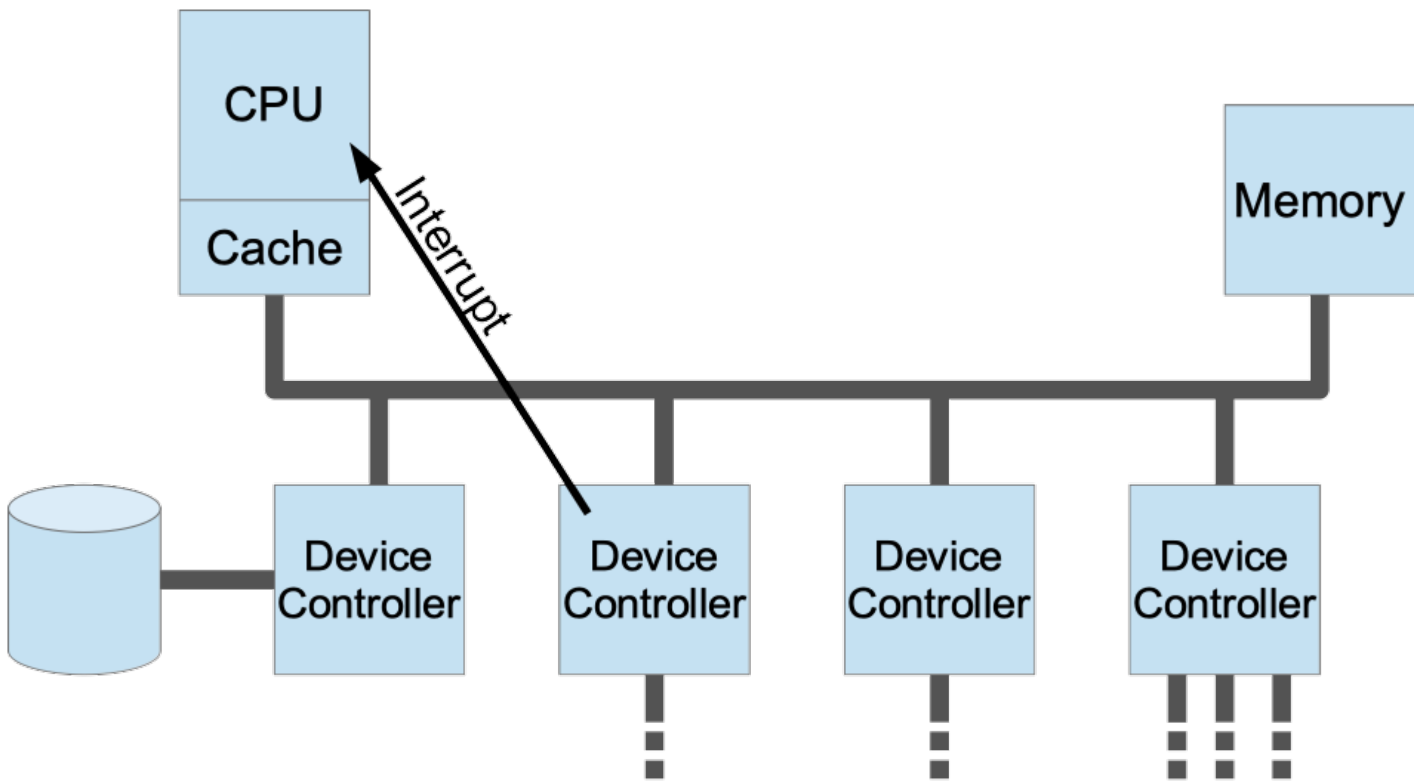
- There's just one CPU (well, pretend)
- OS needs to be able to do things user programs shouldn't
  - Solution: ***Dual-Mode Operation***
  - ***Kernel/Monitor mode***
    Access to all CPU instructions and registers
  - ***User mode***
    Access to a restricted set of CPU features
    Can't run ***privileged instructions***

Dual-Mode Operation

- CPU must provide a bit to keep up with mode
  - Maybe 0 for kernel mode, 1 for user mode
- What if a user program tries to execute a privileged instruction?
- How do we change this bit?
  - Via a CPU instruction?
  - Is that a privileged instruction?

Switching to Kernel Mode

- Interrupt driven (hardware and software)
  - Hardware interrupt by one of the devices



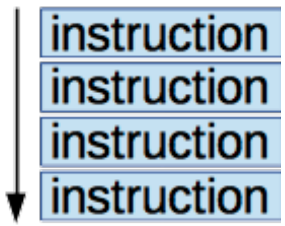
Switching to Kernel Mode

- Interrupt driven (hardware and software)
  - Software interrupt (exception or trap):
    - Software error (e.g., division by zero)
    - Request for operating system service

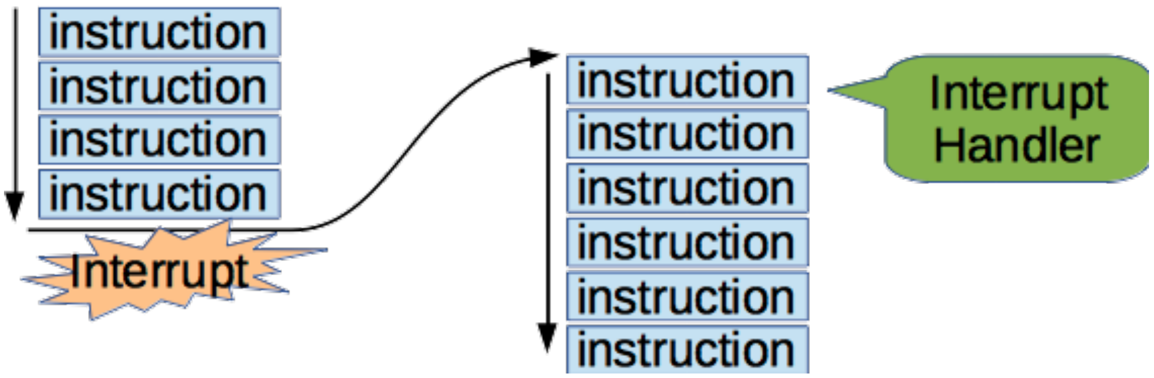
Switching to Kernel Mode

- When an interrupt occurs, the CPU will automatically:
  - Suspend execution of the current instruction sequence
  - Jump to an *interrupt handler* (routine)
  - Switch to kernel mode

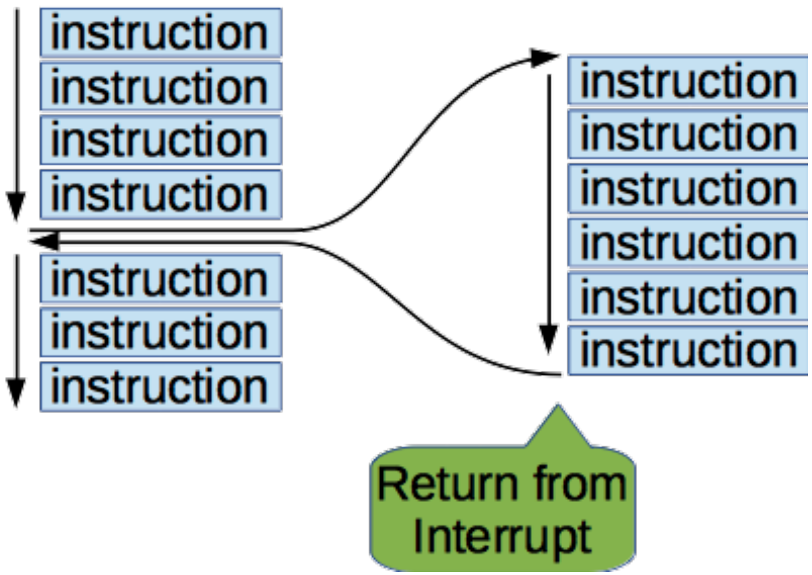
Thinking about Interrupts



Thinking about Interrupts



Thinking about Interrupts

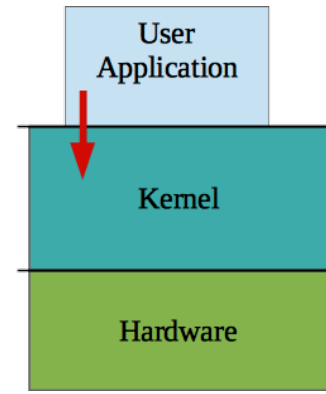


Switching to Kernel Mode

- The *kernel*, the part of the OS that:
  - Separates user programs from the hardware
  - Runs in kernel mode
  - Entry points through interrupt handlers

System Calls

- How to perform restricted operations from a user process: System calls
  - expose certain pieces of functionality to user programs
  - most OSes provide a few hundred calls
  - We have a mechanism for this: a *trap*.
- So, same mechanism for catching a bad program and for handling a system call
  - The *interrupt vector* helps to dispatch control to the right part of the kernel



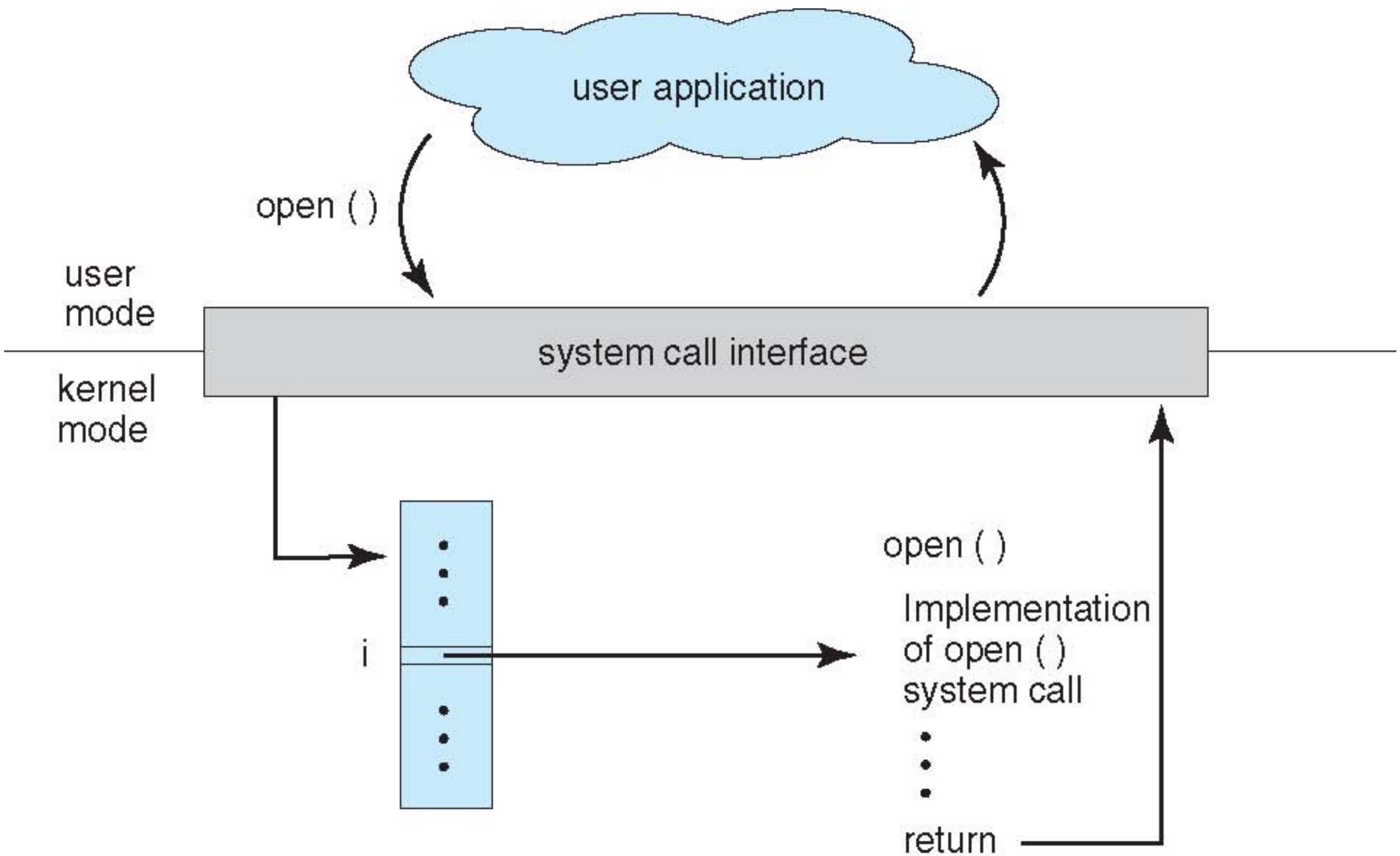
On the trap instruction

- trap instruction simultaneously
  - jumps into the kernel
  - raises the privilege level to kernel mode
- return-from-trap instruction simultaneously
  - returns into the calling user program
  - reduces the privilege level back to user mode
- Need the kernel stack for reliability and security
- The same for interrupt and exception

System Call Implementation

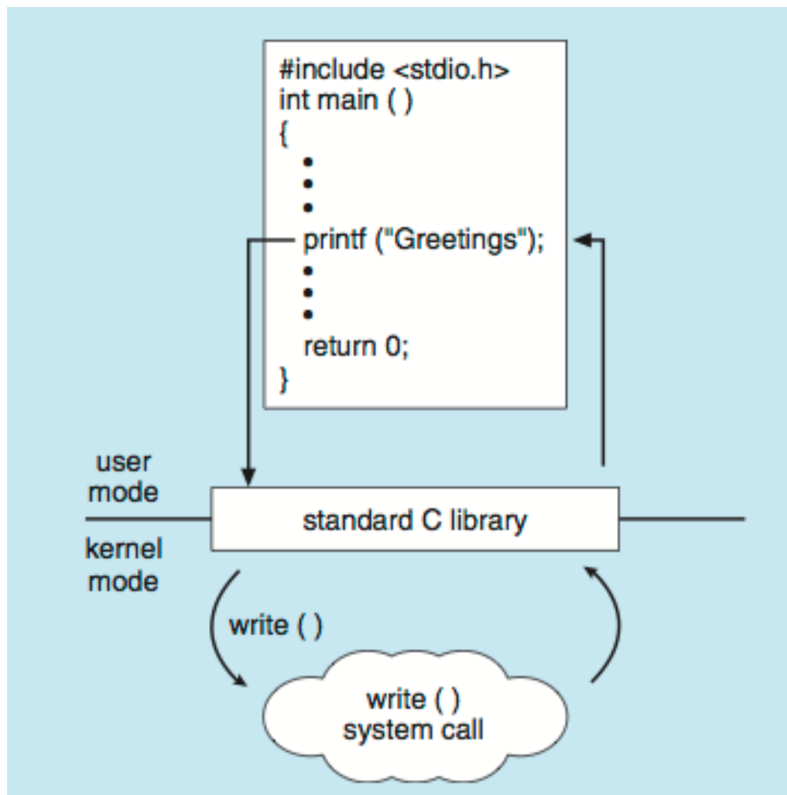
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented or what it does during execution
  - only needs to obey the API and understand what the OS will do as a result of the execution of that system call
  - most of the details of the OS interface are hidden from the programmers

API – System Call – OS Relationship



Standard C Library Example

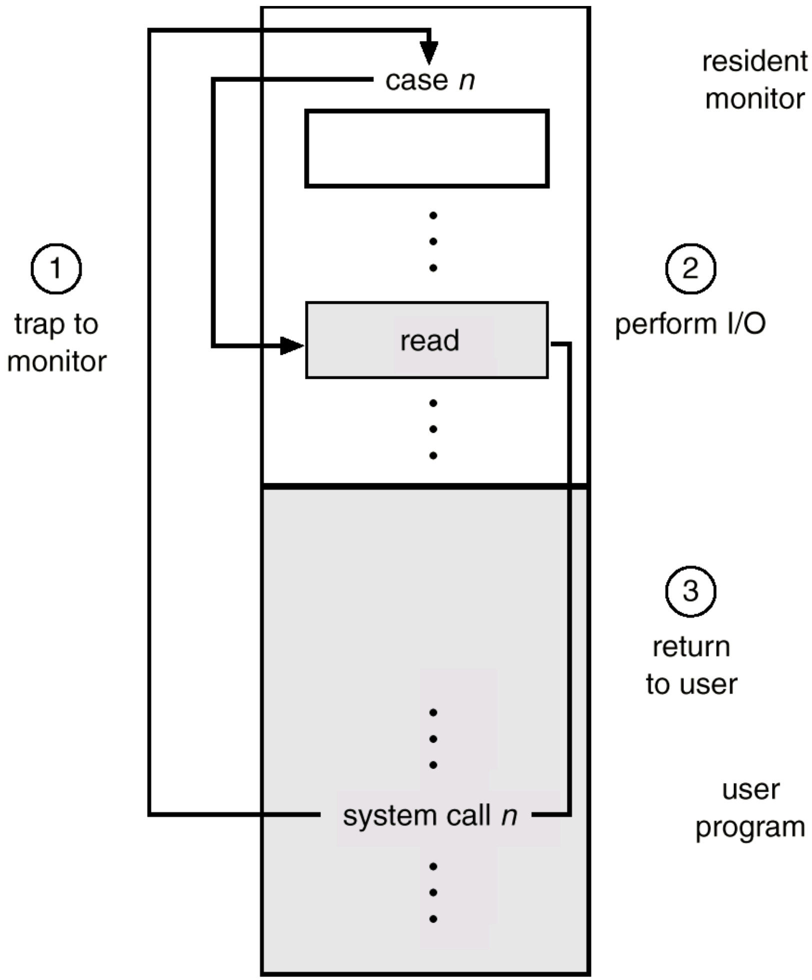
- C program invoking printf() library call, which calls write() system call



I/O Protection

- All I/O instructions are privileged instructions
- OS must force user programs to request I/O this way (via a system call)
- That's *I/O Protection*

Performing I/O Via a System Call



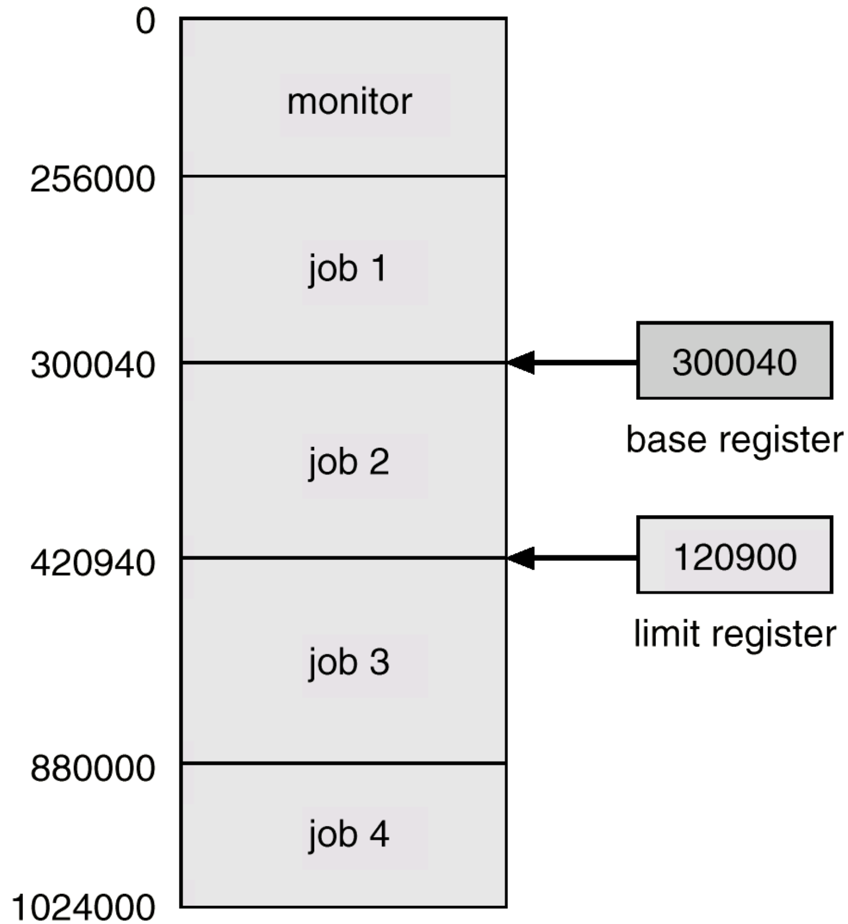
Memory Protection

- We need to control what memory a program can use
- So, we need a hardware mechanism for *memory protection*

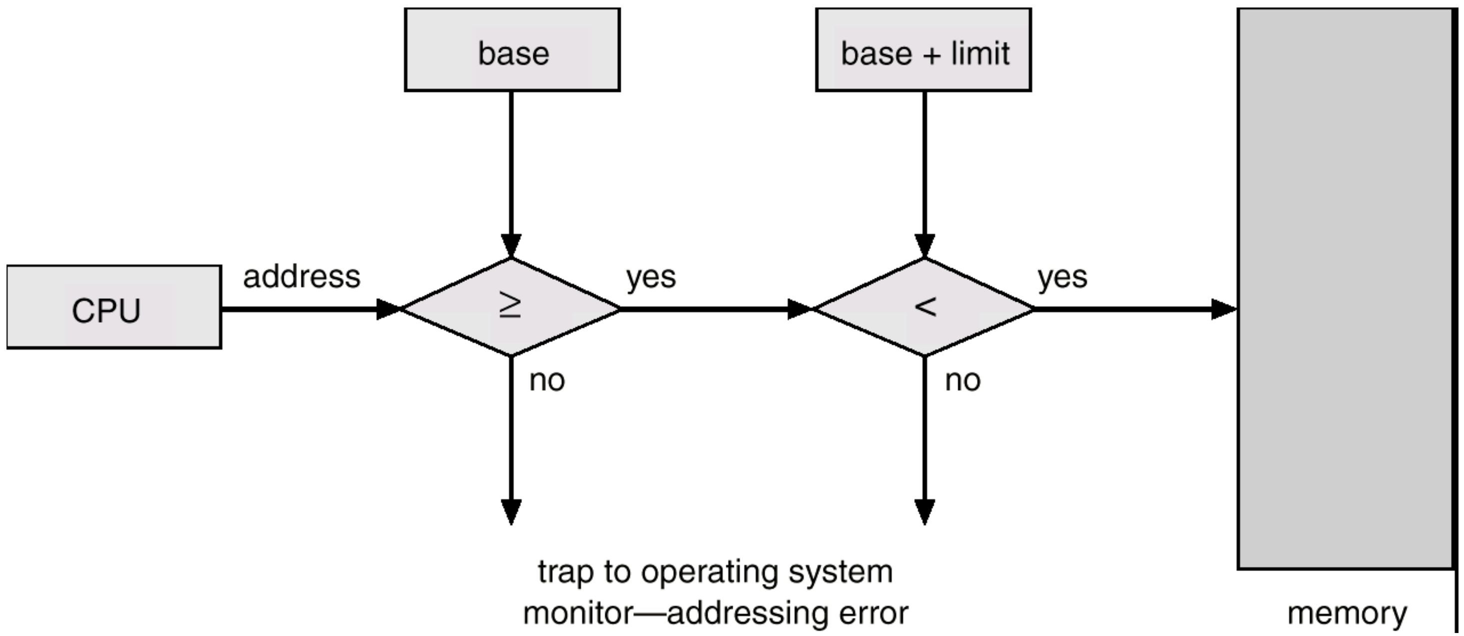
Memory Protection

- Let's imagine a really simple system for this (for now)
  - Program restricted to contiguous region of memory
  - An extra pair of CPU registers
  - *Base register*: address of first byte the CPU can access while in user mode
  - *Limit register*: number of bytes the program can access (starting from base)
- Hardware (CPU) automatically checks these on every memory access
- What if a program tries to cheat?

Operation of Base and Limit Registers



Hardware for Memory Protection



Memory Protection

- In Kernel Mode, base and limit registers have no effect
- Changing base and limit will need to be a privileged instruction
- So, when the kernel runs a user program
  - It sets up the base and limit registers
  - Then switches to user mode
  - Then jumps into the user program

- OK, so now user programs can’t abuse the hardware, right?
- What if the user program just keeps running forever ☹

CPU Protection

- Kernel needs to eventually get to run again
  - Maybe we’ll get lucky, maybe we’ll get an interrupt
  - What if we’re not lucky?
  - Let’s have a hardware interrupt that’s guaranteed to eventually fire
  - A *timer interrupt*
- *CPU Protection*: need to eventually get the CPU back from a process

Timer Interrupt

- Imagine a new CPU register that works as a counter
  - Timer counts down on every (user-mode) CPU instruction
  - Fires an interrupt when timer hits zero
  - Of course, setting the timer needs to be privileged
- So, kernel just sets this timer before giving user program a chance to run
- Essential for interactive systems

Summary

- Role of the Operating System
- System Calls
- Computer Organization
  - Memory hierarchy, caching
  - Interrupts, DMA, I/O, etc.
- Hardware Protection