02 - Design Principles

CEG 4350/5350 Operating Systems Internals and Design

Max Gilson

Defining Requirements

- Before we can implement an OS, we must identify why we want to and what requirements must be satisfied
  - Not every computer requires and operating system
    - Example: Arduino Uno can do many complex things without an OS

Defining Requirements (cont.)

- What is required of our OS?
  - What hardware should it run on?
    - There are many types of processors
    - Should we support them all or focus on one?
  - What type of system should it be?
    - Does our system need to be RTOS, multiprocessor, multitasking, etc?
  - What are the goals of our users?
    - What do our users want from our OS?
  - What are the goals of the system?
    - What do developers want from our OS?

User’s Goals

- Think about the average user, they probably want an OS to be:
  - Convenient to use
  - Easy to learn and use
  - Reliable
  - Safe
  - Fast

System Goals

- Think about YOU having to write code, you probably want to work with a system that is:
  - Easy to design, implement, maintain, and operate
  - Flexible
  - Reliable
  - Error free
  - Efficient
- Imagine writing a lot of really complex code without meeting any of these requirements, nobody would want to expand upon it

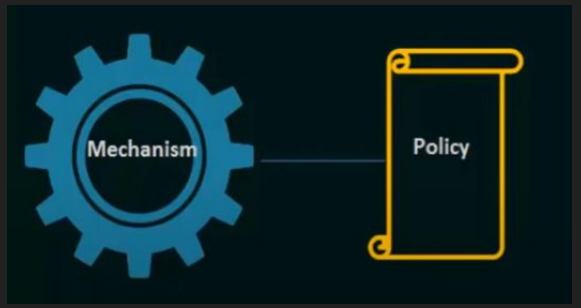
The Separation of Mechanisms and Policies

- Mechanisms determine how to do something
  - A car drives on a road by using a motor, controlled by a pedal, to spin the wheels, which moves the car
- Policies determine what will be done
  - A car driving on the road is forced to obey laws like speed limits, staying within road lines, and maintain distance from other cars

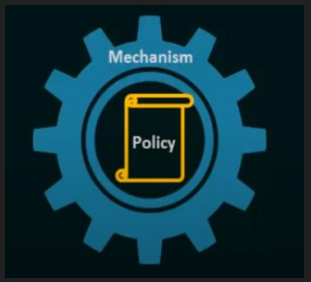
The Separation of Mechanisms and Policies (cont.)

- Mechanisms and policies must be separated
  - A car should not be allowed to change the speed limit that the other cars abide by
  - The speed limit should not be allowed to change how the car drives
- Keeping these separate is critical in working with a complex OS

Good



Bad



Separation of Mechanisms and Policies Example

- Scheduling CPU access
  - Mechanism: the CPU scheduler itself that switches processes in and out of the CPU
  - Policy: the scheduling algorithm which determines which process runs next
- The CPU scheduler (mechanism) should be able to operate with any scheduling algorithm (policy)
- The two should not be dependent or coupled with each other

Hardware Requirements

- In the early days of computing, OSs were written in assembly
  - Assembly language is unique to the CPU's architecture
  - If you write assembly language to run on x86 it will NEVER run on ARM
  - MS-DOS was written in 8088 assembly, and is only available on Intel family of CPUs
- Now most OSs are written in C or C++ with a little bit of assembly
  - Linux is mostly written in C and is available on x86, ARM, RISC-V, MIPS, etc.

Using a High Level Language

- Writing in a high level language grants you:
  - Writing code faster
  - Code is compact
  - Code is easier to understand and debug
  - Easier to port to numerous architectures

Requirements of the OS

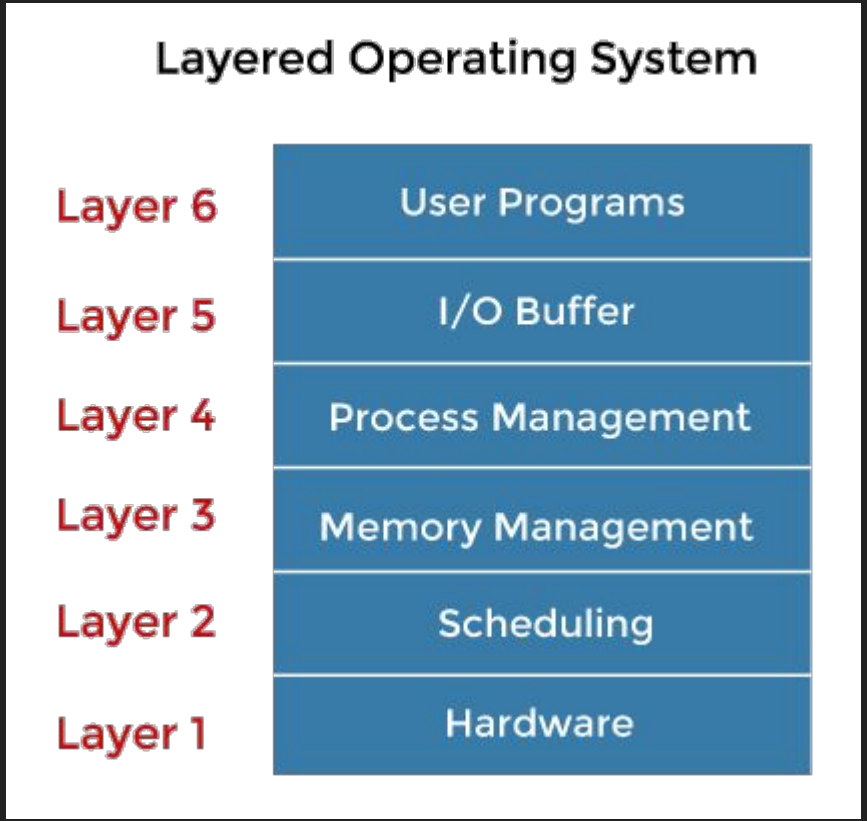
- The OS is a resource manager
  - Allocates time for programs to run on the processor
  - Allocates memory for the programs to use
  - Allocates usage of devices for programs to use, hard drive, printer, keyboard, etc.
- Do we need ensure tasks meet timing deadlines?
  - RTOS
- Do we need to ensure we can run multiple tasks?
  - Multitasking

Kernels

- The kernel is the protected part of the OS that runs in kernel mode
  - This is opposed to programs the user runs which run in user mode
- Protects the critical OS data structures and device registers from user programs

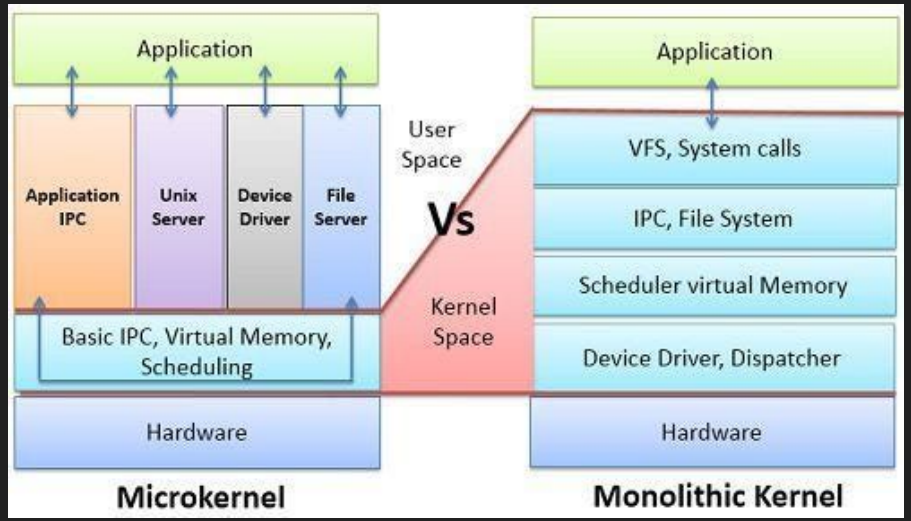
Structuring an OS

- Layered OS Structure
  - All layers exist separately to perform certain functionality
  - Modification in one layer does not affect other layers
  - Lower layers have higher privileges for accessing hardware compared to higher layers
  - Communication overhead between layers



Structuring an OS (cont.)

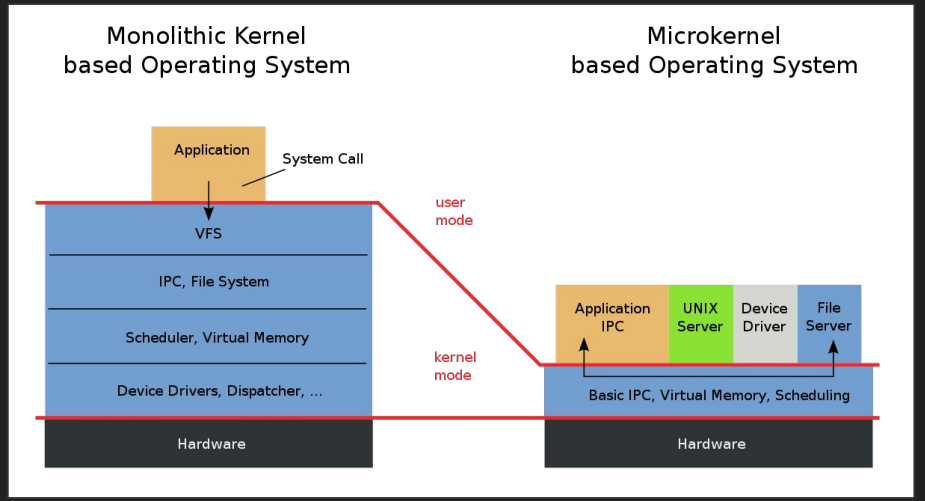
- Monolithic OS Structure
  - Entire OS is working in kernel space
  - Can be hard to write or update
  - Modules stacked on top of modules on top of modules



“Unix is structured like an onion. If you look closely at its insides, you will cry.”
- [Oscar Vivo on Twitter]

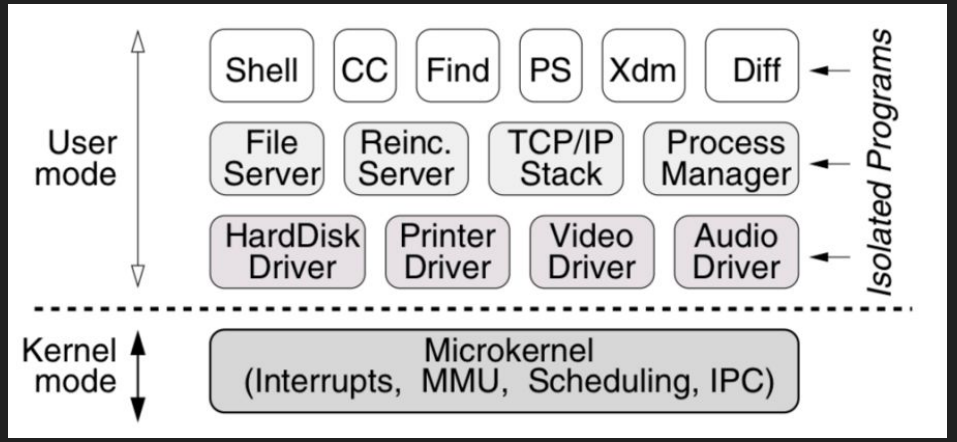
Structuring an OS (cont.)

- Microkernel OS Structure
  - Only basic functionality is provided in the core of the software system
  - Contains the required minimum amount of functions, data, and features to implement an operating system
  - Not as efficient as monolithic OS but is fast to implement



Microkernel

- A microkernel only implements the most basic tools required of the OS
  - Hardware interfaces
  - Task scheduling
- The file system, device drivers, and user programs are all ran in user mode

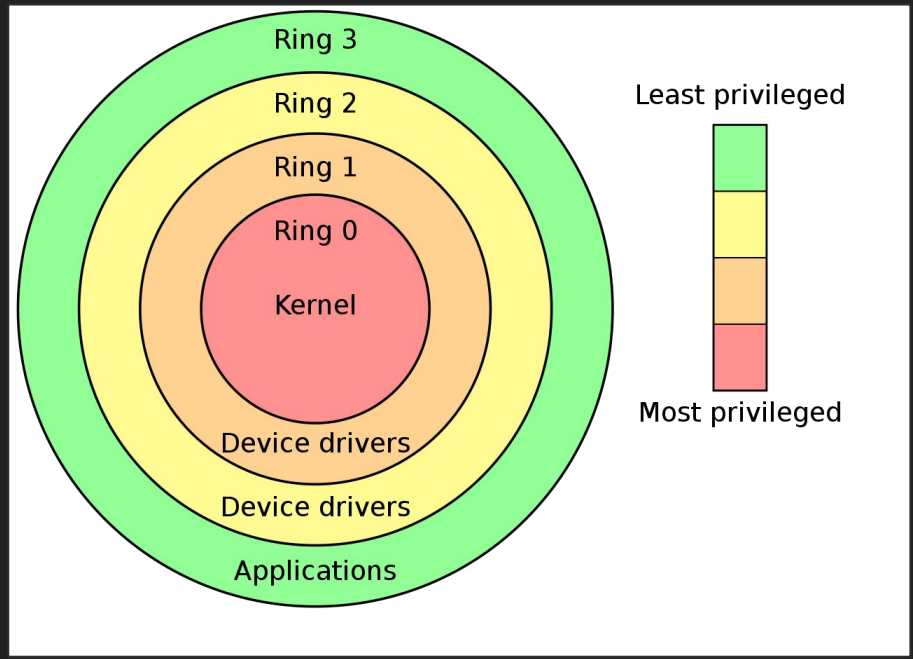


Protections

- An OS must offer certain protections
  - Protect memory
  - Protect I/O
  - Protect CPU
  - Protect user programs from each other
  - Protect the OS from user programs

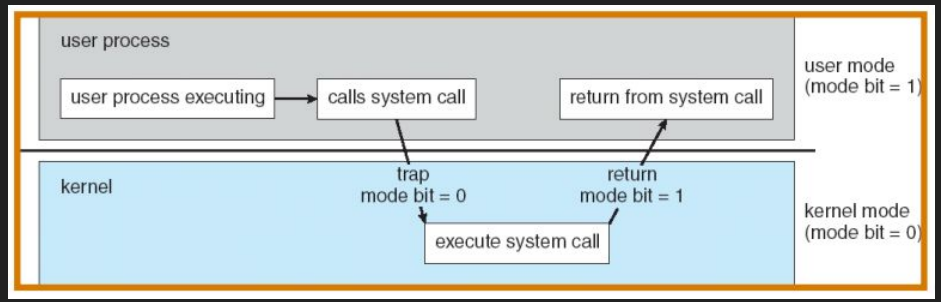
Kernel Mode vs User Mode

- Kernel Mode
  - Has most privileges for accessing memory, I/O, interrupts, special instructions, and halting the CPU
  - This gives the operating system complete control over the hardware
- User Mode
  - Has least privileges to prevent rogue programs from tampering with memory, I/O, interrupts, and halting
- Some OS implement multiple rings of protection for other purposes
- The hardware itself must have a way of enabling and disabling the protection layers usually through bits in a register



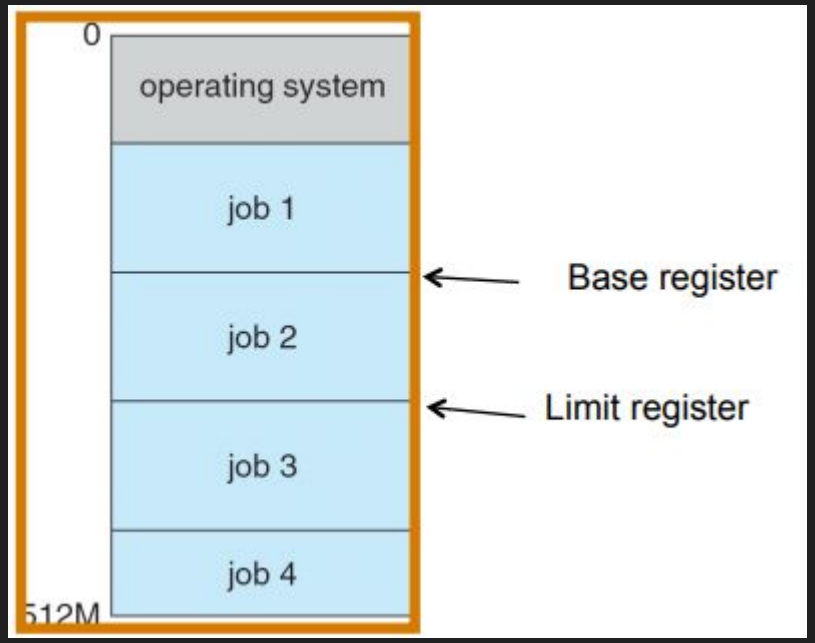
Kernel Mode vs User Mode (cont.)

- Switching between modes should be through system calls
- A system call is a mechanism used by programs to request services from the OS
  - System calls exist so the user programs can “ask” the OS if they can use sensitive resources
  - The OS determines how the system call is executed, so the OS is still in control



Memory Protection

- The OS must protect user programs from tampering with each other
- The OS must protect itself from user programs tampering with it
- The hardware must allow for checking if an instruction or data address is within a specific range (base and limit registers)
    - Example:
      base register points to:
      x0000 F000
      limit register points to:
      x0000 FFFF
      The hardware should not allow memory accesses outside of this range

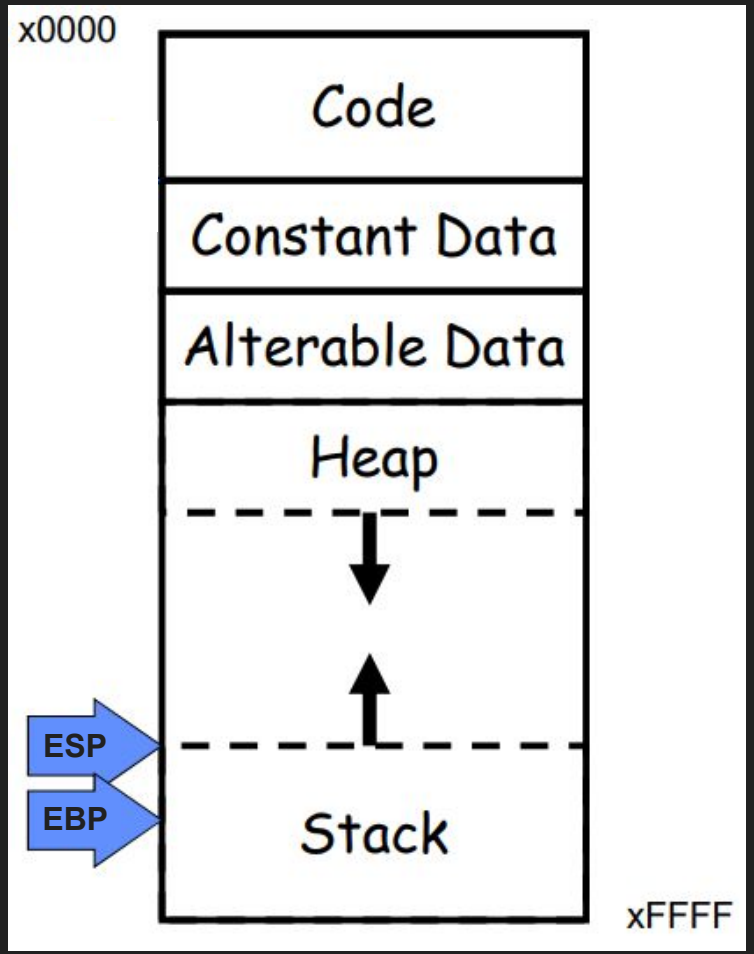


I/O Access

- Computers are not very useful without inputs and outputs (I/O)
- Interrupt based I/O
  - I/O sends an interrupt request when it needs CPU to handle it
  - I/O can be handled immediately
- Memory Mapped I/O
  - Uses a portion of the regular address space (memory addresses) for the I/O
  - I/O can be accessed by reading and writing to memory
- Port Mapped I/O
  - Uses a separate address space (not memory addresses) to identify the port address and special instructions to read from or write to the port
  - The distinction between the two addressing spaces is made in the control bus

Runtime Stack

- To manage data for multiple programs or even basic programs a runtime stack is required
- The runtime stack keeps track of all the variables and data in our program
- The stack pointer
  - (ESP or SP for 32 bit or 16 bit)
  - Used to keep track of the top of the the runtime stack
- The frame pointer
  - (EBP or BP for 32 bit or 16 bit)
  - Use to keep track of the start of variables in the current frame
  - The current frame is the function we're inside



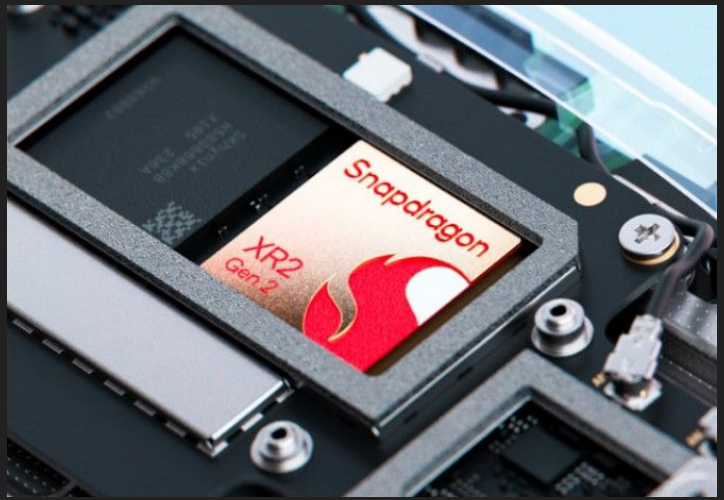
03 - Hardware and The OS

CEG 4350/5350 Operating Systems Internals and Design

Max Gilson

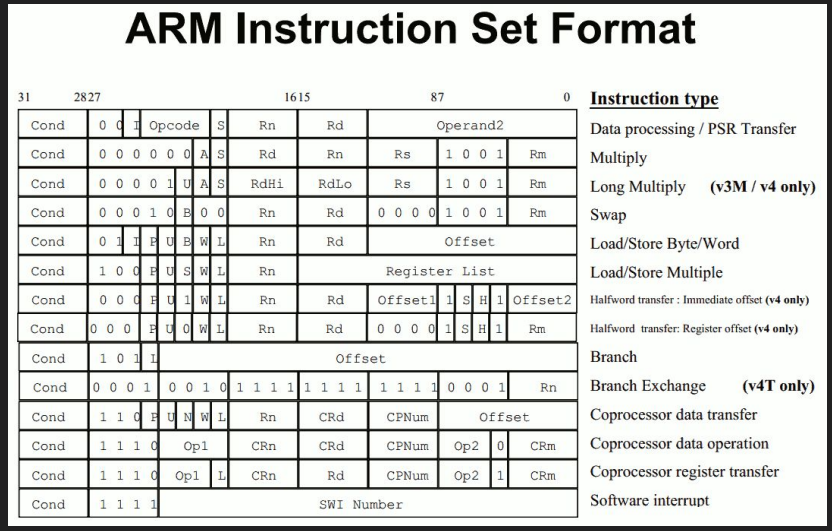
The CPU

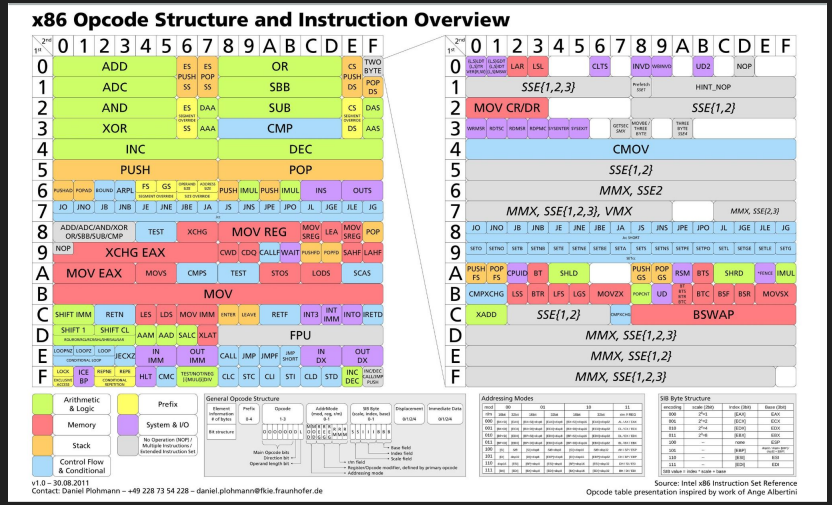
- The Central Processing Unit (CPU) executes the instructions we provide to it
- Without the CPU, the computer cannot run programs or do anything useful
- The CPU has a specific instruction set architecture (ISA) that defines what instructions it runs and how it runs them



The ISA

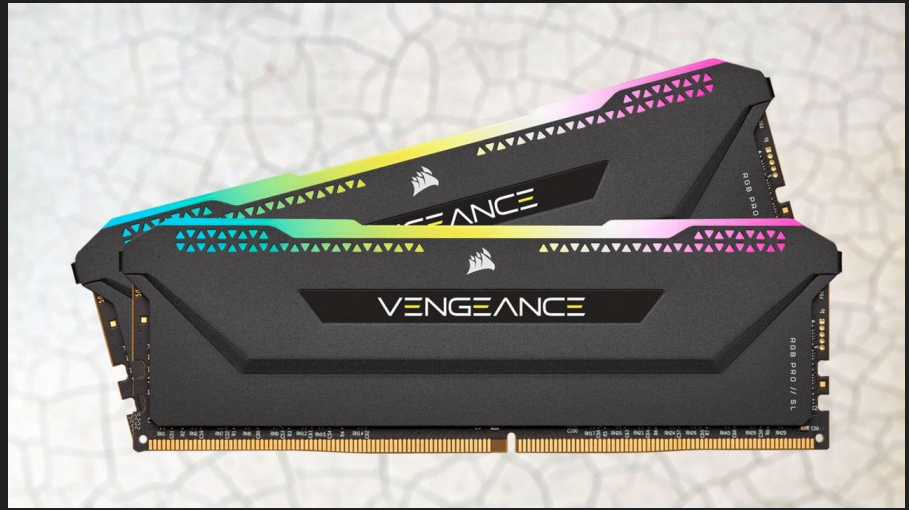
- The ISA is a “blueprint” for the tools the CPU has at its disposal
- The ISA defines what registers, instructions, and systems are available for the CPU to use
- x86, ARM, RISC-V, MIPS are examples
  - If you write code for one ISA it cannot run on another ISA without recompiling or rewriting code
  - Assembly code written for x86 will never run on ARM without a complete rewrite from scratch
  - A C program can be compiled for multiple ISAs





The RAM

- The Random Access Memory (RAM) stores our programs and data
- Without RAM, the CPU wouldn’t have instructions to execute
- Any program you want to run has to loaded into RAM



The Storage Device

- The storage device holds our files and programs
  - Hard Disk Drive (HDD)
  - Solid State Drive (SSD)
  - Floppy Disk
- Our storage device is an I/O device
- When you want to execute a program or read a file, it is copied from storage and put into RAM
- If you hear the word “disk” we are talking about these

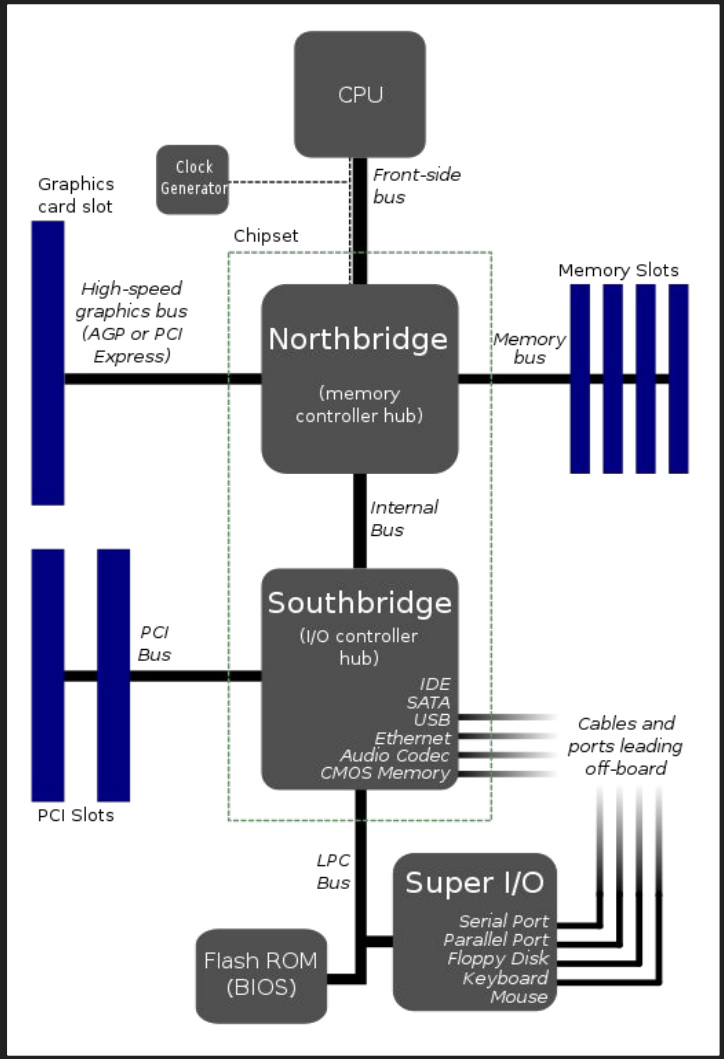
SSD

HDD

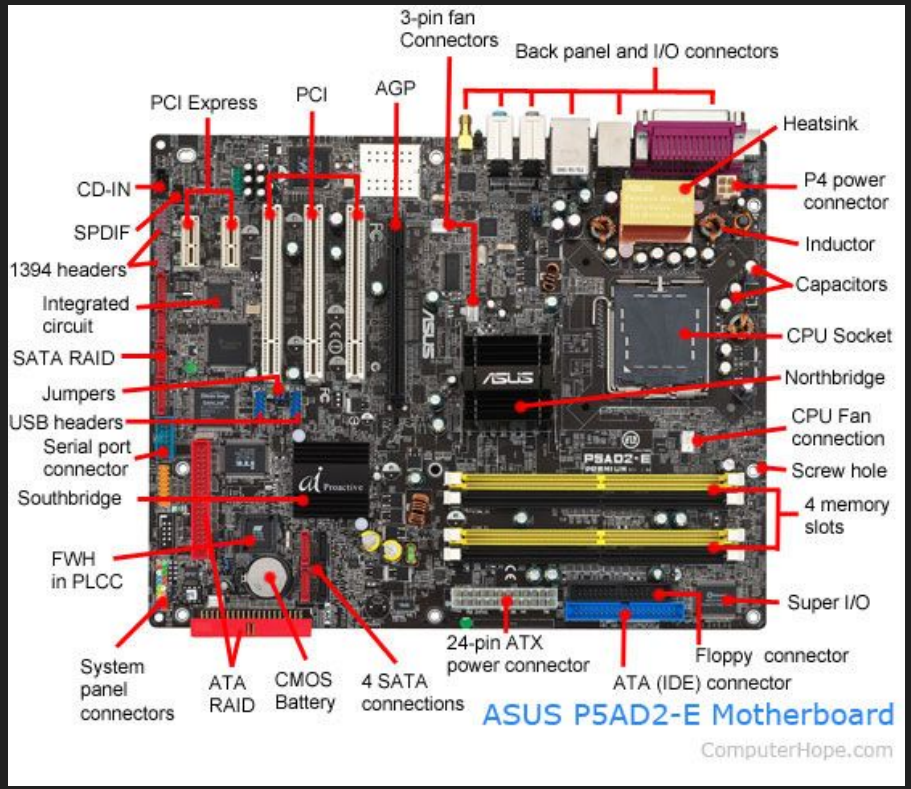
Floppy Disk

The Motherboard

- The motherboard is a physical component of a computer that connects CPU, RAM, and I/O devices together
- The CPU, RAM, and I/O all plug into the motherboard as individual modules
- There are other components on the motherboard that assist in the transfer of data between these components
  - Northbridge interconnects CPU, RAM, and Southbridge
    - Fast stuff
  - Southbridge interconnects I/O
    - Slower stuff
  - Busses are the paths for data to travel

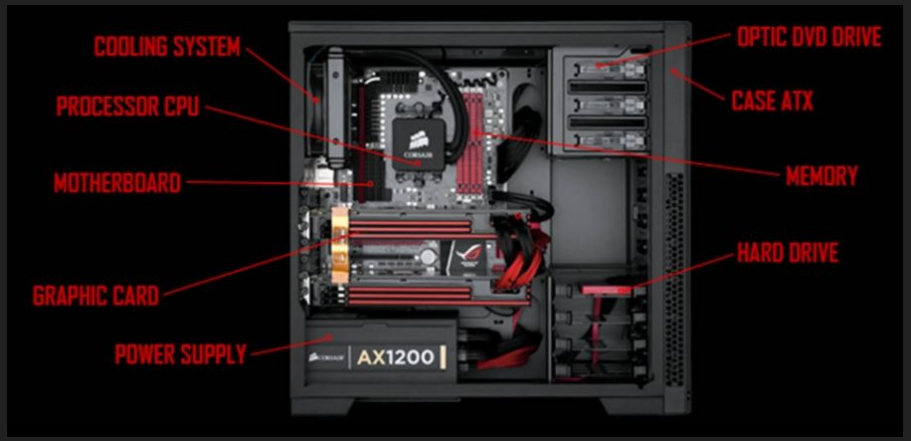


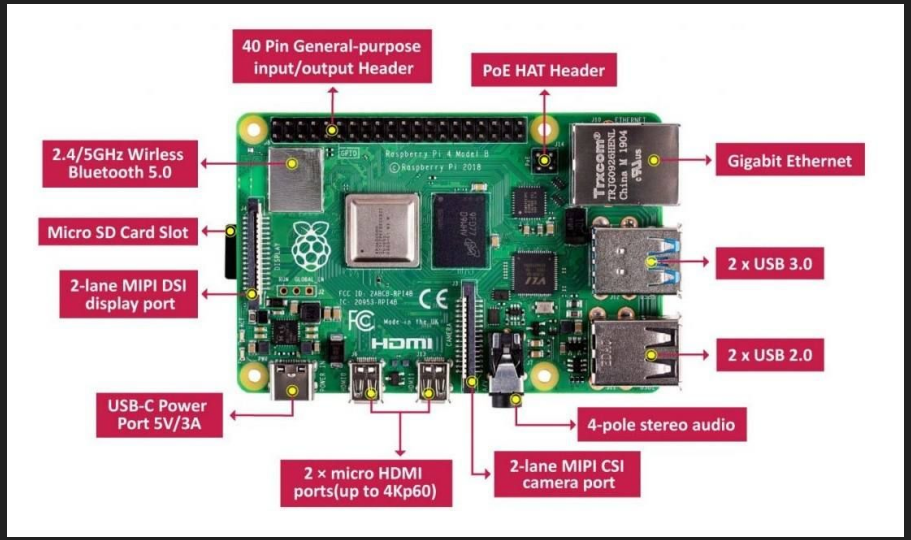
The Motherboard (cont.)



Some Extras

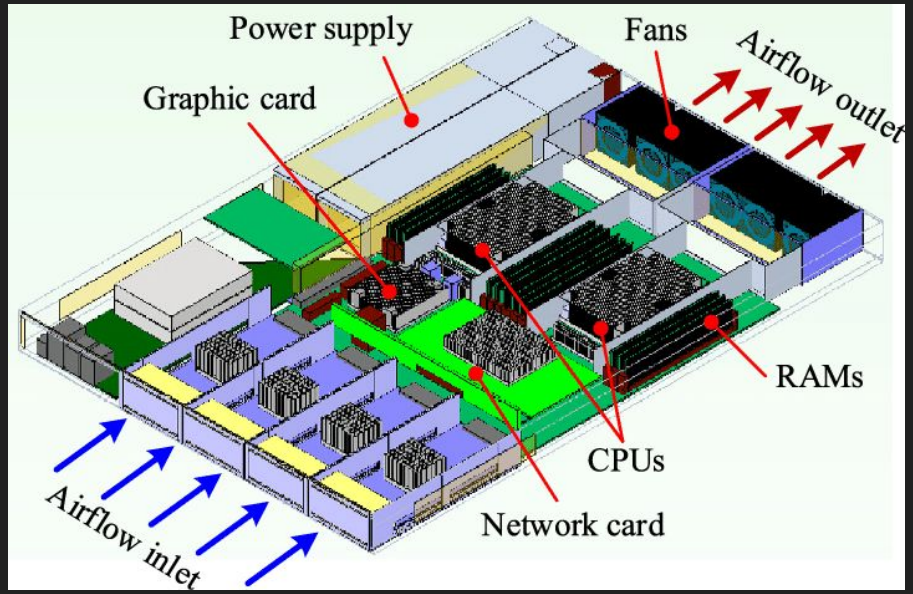
- The CPU, RAM, and Storage are our main components
- We need something to power our computer
  - Power Supply Unit (PSU)
  - Battery (embedded system)
- We *might* need some other things
  - Mouse, Keyboard, Graphics Card
  - Case
  - CPU cooler and fans
  - Network connectivity





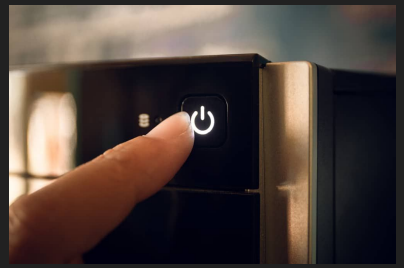
There are Many Ways To Design a Computer

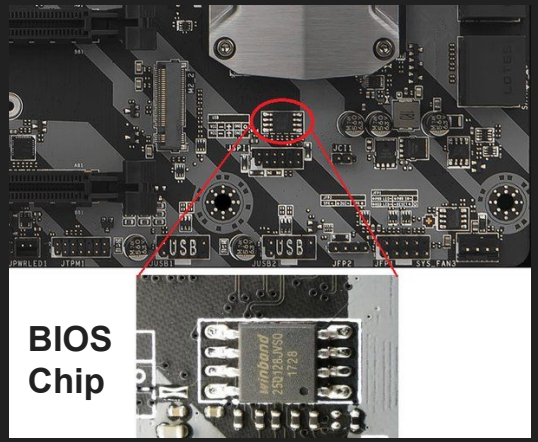
- Computers are ubiquitous and complex
- Applications for computers vary but general purpose computers can do a lot
  - Desktops/laptops
- Before trying to solve a problem, ensure you're working with the correct hardware
  - Locally running a LLM AI on a Raspberry Pi is technically possible but produces lackluster results



The Boot Process

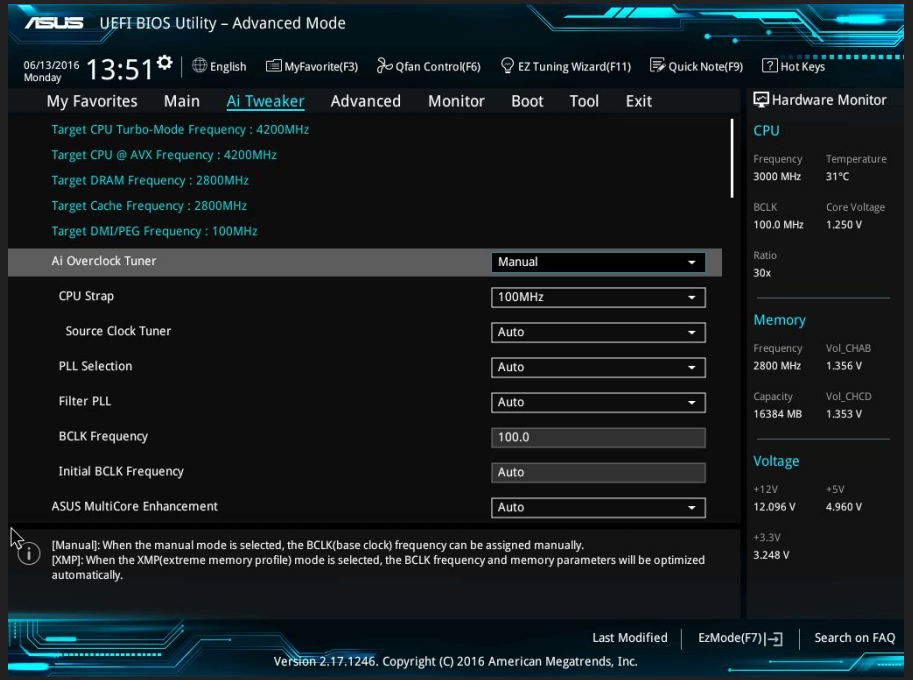
1. The power button is pressed
    a. The PSU begins sending power to the system
2. The Basic Input and Output System (BIOS) prepares the hardware and loads the bootloader from the Master Boot Record (MBR) into memory
    a. The BIOS is a program stored on a chip on the motherboard usually 16MB max size
    b. The MBR is the first 512 bytes on the storage device
    c. Power-On Self Test (POST) initializes RAM, search for storage, USB devices, performs quick tests (i.e. keyboard check), initializes video card
    d. If a bootable disk is found, start its bootloader





Unified Extensible Firmware Interface

- More modern machines use Unified Extensible Firmware Interface (UEFI) instead of BIOS
- UEFI does everything BIOS does and more
  - Provides nice GUI interface
  - Mouse support
  - Secure boot
  - Faster boot times
  - More options for configuring boot
    - Up to 128 physical partitions (BIOS has 4)



The Boot Process (cont.)

3. The bootloader stored in the MBR sets up hardware for the OS and loads the kernel into memory
   a. The bootloader is a very small (512 bytes maximum) program
   b. The bootloader must run in “Real Mode” and will switch to “Protected Mode” on x86 ISA to execute the kernel
   c. The MBR also specifies which partitions are available on the storage
   d. Sometimes, there is little bootloader code in the MBR and it simply jumps the computer’s execution to the active partition’s code
   e. The small bootloader in the MBR is the First-Stage Bootloader, the code in the main partition is the Second-Stage Bootloader
   f. Multiple partitions can be used to boot different OSs from the same storage drive

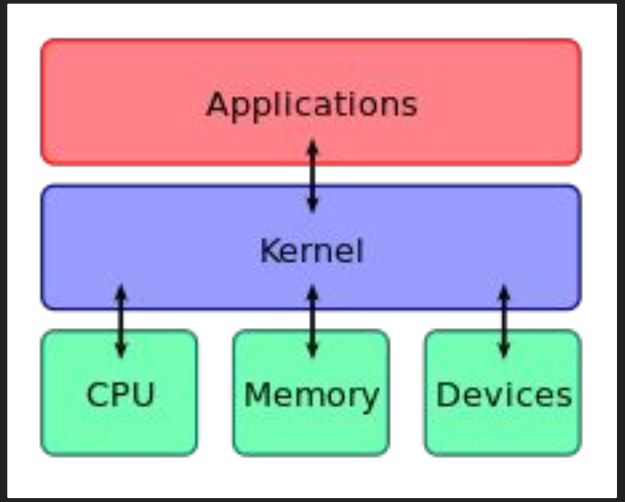
Structure of a classical generic MBR

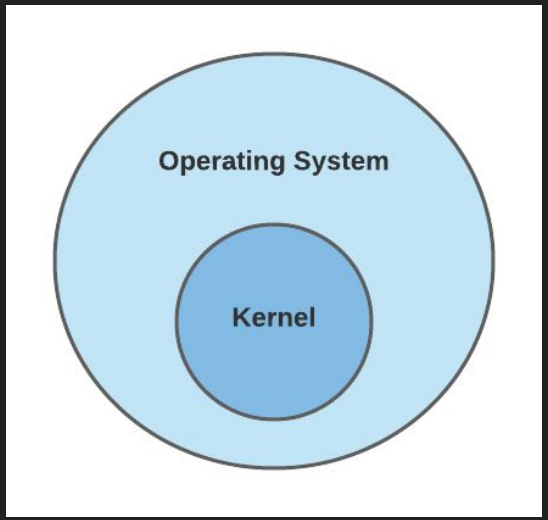
Address		Description		Size (bytes)
Hex	Dec			
000_{hex}	+0	Bootstrap code area		446
$1BE_{hex}$	+446	Partition entry №1	*Partition table* (for primary partitions)	16
$1CE_{hex}$	+462	Partition entry №2		16
$1DE_{hex}$	+478	Partition entry №3		16
$1EE_{hex}$	+494	Partition entry №4		16
$1FE_{hex}$	+510	55_{hex}	*Boot signature*[a]	2
$1FF_{hex}$	+511	AA_{hex}		
			Total size: 446 + 4×16 + 2	**512**

Element (offset)	Size	Description
0	byte	Boot indicator bit flag: 0 = no, 0x80 = bootable (or "active")
1	byte	Starting head
2	6 bits	Starting sector (Bits 6-7 are the upper two bits for the Starting Cylinder field.)
3	10 bits	Starting Cylinder
4	byte	System ID
5	byte	Ending Head
6	6 bits	Ending Sector (Bits 6-7 are the upper two bits for the ending cylinder field)
7	10 bits	Ending Cylinder
8	uint32_t	Relative Sector (to start of partition -- also equals the partition's starting LBA value)
12	uint32_t	Total Sectors in partition

The Boot Process (cont.)

4. The kernel is now running and begins to initialize and execute the core functions of the operating system
    a. The kernel connects and manages resources for the computer's operating system
    b. Task management, memory management, and I/O interfaces
    c. The kernel sits between the hardware and the user’s programs
5. The operating system is now running and takes inputs from the user and user’s programs to do complex and useful tasks
    a. Running Netflix, playing video games, crypto mining, etc.





The Shutdown Process

- When the computer is instructed to shutdown, the OS may do some things first
  - It is not a good idea for the OS to “pull the plug”
- If there is unsaved file data in RAM, it needs to be saved to storage (or it is lost)
- If there are processes that are running, the OS should request them to terminate
  - This takes a long time because the processes may be in virtual memory (waiting in slow storage) and have to finish up before being forced to terminate
- Once processes have stopped and the file systems are unloaded, the kernel sends a signal to the BIOS that turns the PSU off

The Kernel

- The kernel is the lowest level of the operating system
  - There are many components and often involves low level programming (assembly)
  - Low level programming might be necessary to interface the hardware directly
  - Kernels are usually a combination of both C and assembly
  - Even though C is a high level programming language, it gets us close enough to the hardware most of the time

Execution Modes

- On x86 based processors we have two main CPU modes:
  - Real Mode
    - 16 bit instructions, 16 bit registers, 16 bit addresses
    - CAN call BIOS interrupts
    - No safety nets or protections
    - CPU boots into this mode
    - Kept for legacy purposes
  - Protected Mode
    - 32 bit instructions, 32 bit registers, 32 bit addresses
    - CANNOT call BIOS interrupts
    - Extra protections, multitasking, virtual memory
    - The CPU must be forced into this mode from running in real mode
    - All major OSs run in protected mode
- Protected mode is enabled by setting bit 0 to “1” in register CR0
  - It’s not that simple though
  - Switching back to real mode is not easy
- There are more than these two modes but these are what we will focus on for now

Hardware Abstraction Layer

- The OS must implement a Hardware Abstraction Layer (HAL)
- The HAL is a software component that acts as an interface between the hardware and the operating system
- At its most basic, we must have a HAL that allows us to:
  - Write to display
  - Write to storage
  - Read from keyboard
  - Read from storage

VGA Text Mode

- Using VGA text mode, writing to the display is the simplest hardware interface to implement
  - VGA text mode makes things easy, otherwise we need more advanced techniques, i.e. writing drivers for graphics card 😨
  - VGA text mode is limited, we cannot display nice graphics
- To set VGA text mode the computer must be in “Real Mode”
  - Once you are still in real mode, set AH = x00 and AL = x##
  - AL should be set to your desired BIOS Video Mode found here: https://www.minuszerodegrees.net/video/bios_video_modes.htm
  - Once these values are set call the INT 0x10 interrupt
- More info on INT 0x10 interrupt (including how to disable the blinking cursor) can be found here:
  - https://en.wikipedia.org/wiki/INT_10H

Write to Display

- You can write to the display by writing to memory locations starting at 0xB8000
- Each character on the display is comprised of two bytes, first the ASCII character and second the color
  - For example, if you want to display the character “a” in monochrome green at the first character of the display, you must set memory to:
    - 0xB8000 <- 0x61
    - 0xB8001 <- 0x2A

Write to Display (cont.)

- In the previous example:
  - 0xB8000 <- 0x61
  - 0xB8001 <- 0x2A
- 0x61 is an ASCII character ‘a’
  - Available characters: https://www.asciitable.com/
- 0x2A is the color code for green text on a light green background
  - Available colors: https://wiki.osdev.org/Printing_To_Screen
    - Hint: look for “Color Number” in the “Color Table”
  - The most significant 3 bits specifies the background color
  - The least significant 4 bits specifies the text color
  - 0x2A = 0010 1010
- For light grey text on black background use 0x07
  - 0x07 = 0000 0111

Write to Display (cont.)

- In C it is very easy to write to video memory:
  - char *vidmem = (char *) 0xB8000;
    - Creates a pointer pointing to address 0xB8000
  - vidmem[0] = 0x61;
    - Sets the first location the pointer points to
    - 0xB8000 to 0x61
  - vidmem[1] = 0x2A;
    - Sets the second location the pointer points to
    - 0xB8001 to 0x2A

Reading From Keyboard

- Reading from the keyboard is not as easy as reading from a memory location
- To read from the keyboard port 0x60 and 0x64 must be read from
  - Port 0x60 is NOT a memory address
  - Port 0x60 is the Keyboard Data I/O port
  - Port 0x60 is the keyboard data port (provides a scancode of what was typed)
  - Port 0x64 is the keyboard status port (indicates if the keyboard is ready to send data)
    - The least significant bit (bit 0) will indicate keyboard readiness (1 for ready, 0 for waiting)
- There are many different types of keyboards
  - US English, UK (British) English, Chinese, Spanish, etc.
  - To ensure support for these keyboards, scancodes are used (not ASCII)
  - There is no ASCII equivalent for special keys like SHIFT or INSERT so scancodes must be used to register all the keys
- Keyboard scancodes can be found here: https://wiki.osdev.org/PS/2_Keyboard

Reading From Keyboard (cont.)

- Reading from the port must be done in assembly:

```
in al, 0x60             ; put byte from port 80 into al
```

  - The above code reads port 0x60 and puts the result into the AL register
  - If using inline assembly, the `inb` (input byte) and `inw` (input word) can be used for reading from ports
- The scancode must be converted to an ASCII character, which can be implemented with an array of ASCII characters mapped to scancode indices
- This assembly code can be called from a C program, or written in a C program using inline assembly, to create your own barebones scanf function
  - This is the preferred method to avoid having to swap between C and assembly in different .c and .asm files

Accessing Ports using Inline Assembly in C

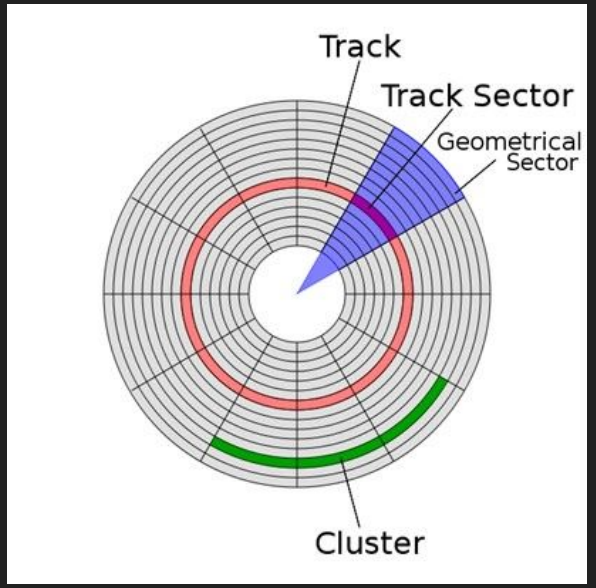
```
typedef unsigned char  uint8;
typedef unsigned short uint16;

void outb(uint16 port, uint8 value)
{
    asm volatile ("outb %1, %0" : : "dN" (port), "a" (value));
}

void outw(uint16 port, uint16 value)
{
    asm volatile ("outw %1, %0" : : "dN" (port), "a" (value));
}

uint8 inb(uint16 port)
{
   uint8 ret;
   asm volatile("inb %1, %0" : "=a" (ret) : "dN" (port));
   return ret;
}

uint16 inw(uint16 port)
{
   uint16 ret;
   asm volatile ("inw %1, %0" : "=a" (ret) : "dN" (port));
   return ret;
}
```

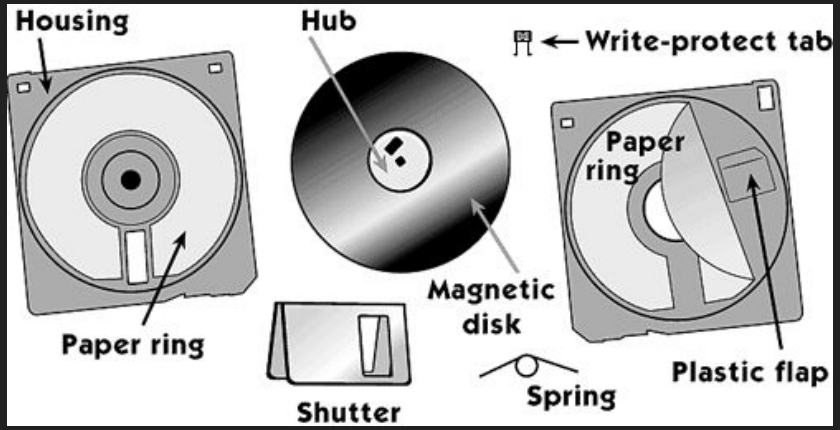
Reading from and Writing to Storage

- We will only focus on floppy disks for our storage
- Floppy disks are the easiest to interface with and are still supported to this day
  - Even though first introduced in 1967
- HDD or SSD drives are a more difficult storage to interface with
- Even though floppies are easier than HDD/SSD, it is still very complex
- For those interested in just how complicated this can get:
  - https://wiki.osdev.org/Floppy_Disk_Controller

Anatomy of a Floppy Disk

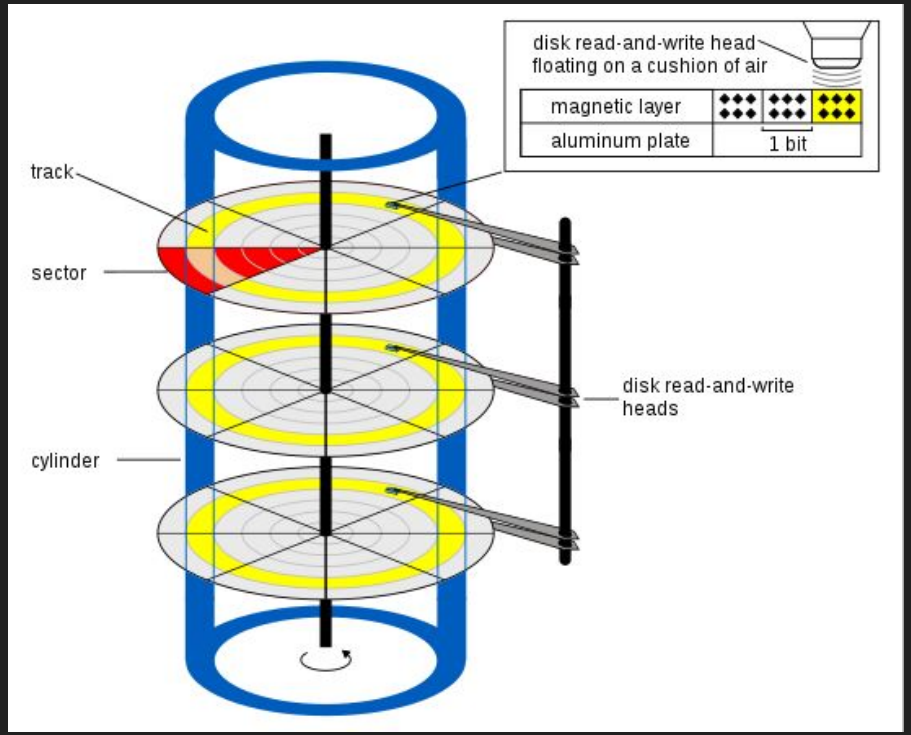
- 512 bytes per sector
- 18 sectors per track
- 80 tracks per side
- 2 heads
- Total 1,474,560 bytes per disk
- 3.5 inch 1.44 MB disks are the most common





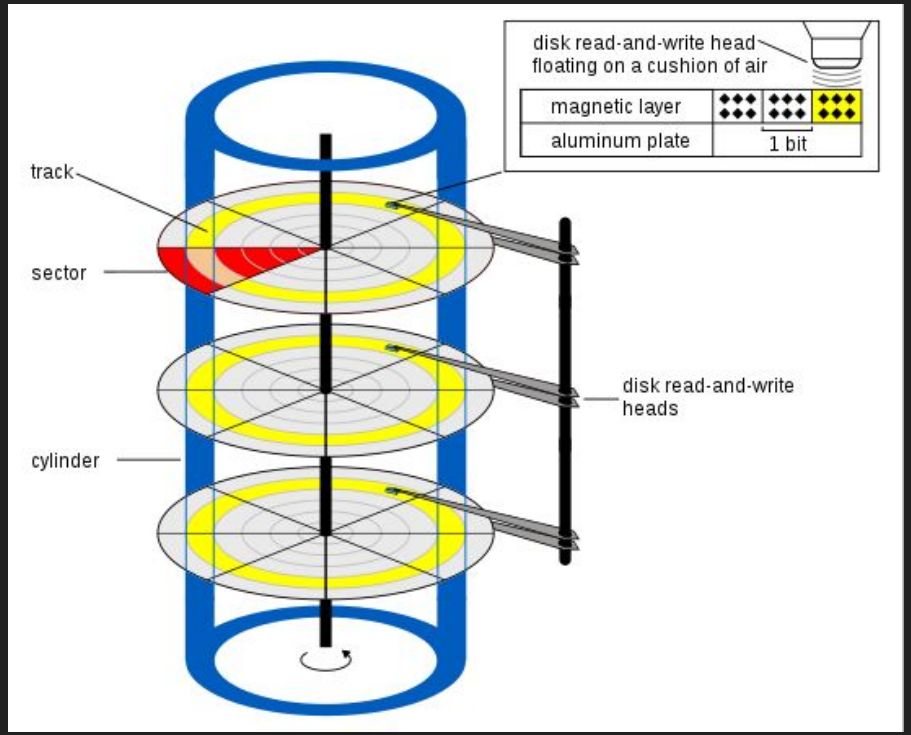
Cylinder-Head-Sector Addressing

- The cylinder-head-sector (CHS) addressing scheme is a way of accessing a specific memory location in storage
- This is an old school way of addressing and is required by INT 13,2 BIOS interrupt



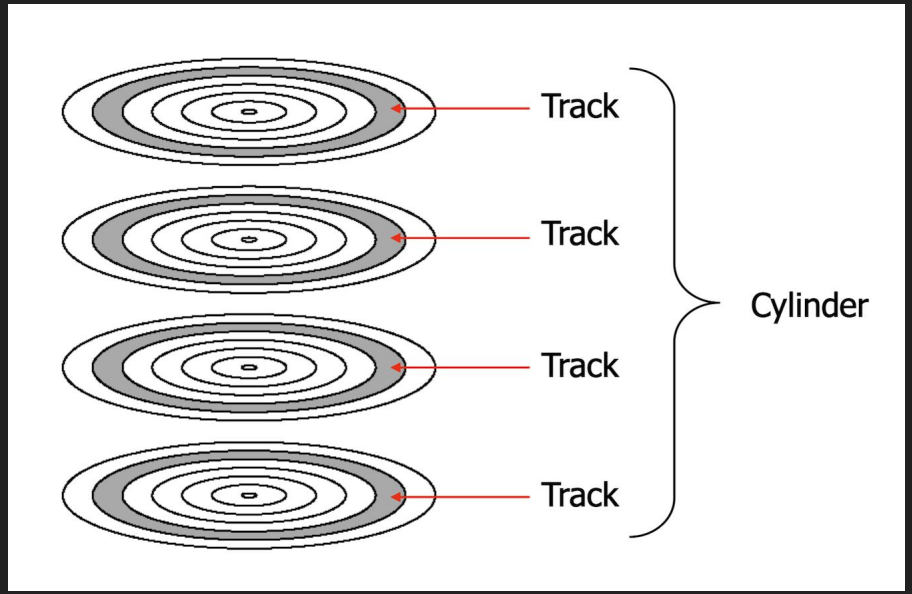
Cylinder-Head-Sector Addressing (cont.)

- To read from cylinder 0, head 0, sector 2:
  - CHS = 0,0,2
- Note: Sectors start at 1, there is NO sector 0!
  - Cylinder and head start from 0



Cylinder-Head-Sector Addressing (cont.)

- The cylinder is the aggregate of all tracks on all platters (disks)
- Cylinder count is not much of a concern today since most OSs use logical block addressing
  - Often the OS will report inaccurate numbers for the physical CHS that exists



Logical Block Addressing

- It is more intuitive to use logical block addressing (LBA)
  - We don’t care where our data is physically on the drive, we just want to access it specifically
- LBA provides a linear address space for dealing with storage
- Instead of specifying the cylinder, head, and sector, the next sector is +1 value
- To calculate the CHS value for a given LBA value use the following equations
  - Hint: We know a floppy drive has 18 sectors per track and 2 heads

$$C = (\mathrm{LBA} \,/\, \text{sectors per track}) \,/\, \text{number of heads}$$
$$H = (\mathrm{LBA} \,/\, \text{sectors per track}) \,\%\, \text{number of heads}$$
$$S = (\mathrm{LBA} \,\%\, \text{sectors per track}) + 1$$

LBA and CHS equivalence with 16 heads per cylinder

LBA value	CHS tuple
0	0, 0, 1
1	0, 0, 2
2	0, 0, 3
62	0, 0, 63
63	0, 1, 1
945	0, 15, 1
1007	0, 15, 63
1008	1, 0, 1
1070	1, 0, 63
1071	1, 1, 1
1133	1, 1, 63
1134	1, 2, 1
2015	1, 15, 63
2016	2, 0, 1
16,127	15, 15, 63
16,128	16, 0, 1
32,255	31, 15, 63
32,256	32, 0, 1
16,450,559	16319, 15, 63
16,514,063	16382, 15, 63

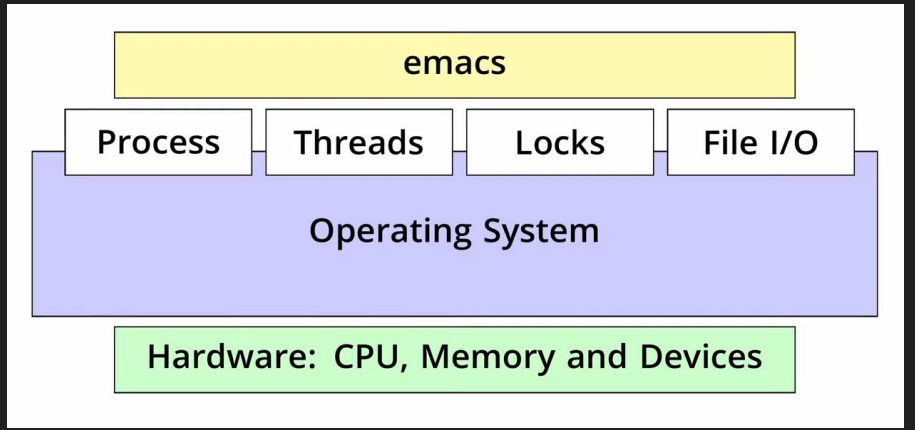
04 - Processes and Threads

CEG 4350/5350 Operating Systems Internals and Design

Max Gilson

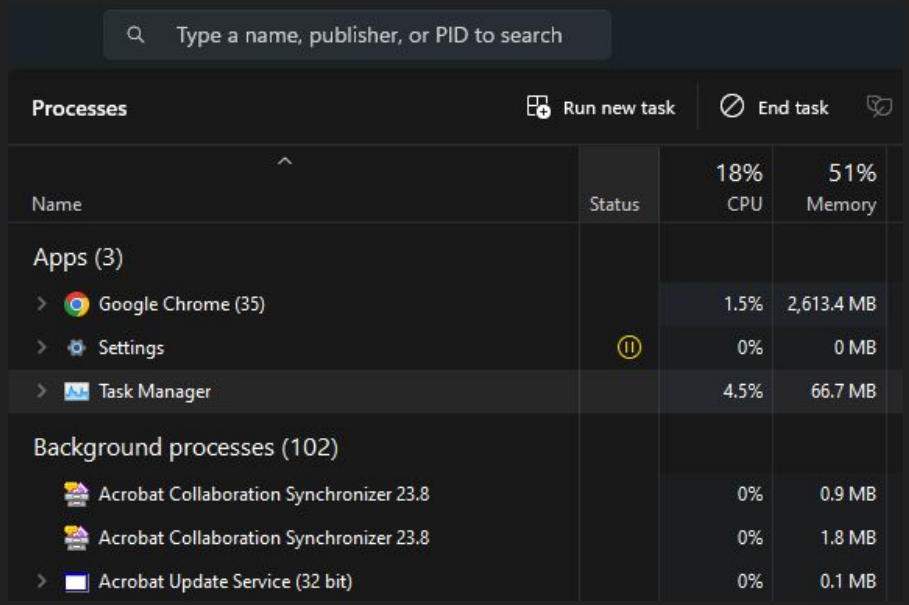
The Process Abstraction

- The OS provides abstractions for our programs
  - Abstraction is breaking down complex problems into smaller, manageable parts
  - An abstraction is basically a simplified interface
- The OS provides a process abstraction for our programs to use to get access to CPU time, memory, or other resources
- Emacs is a text editor available on Linux



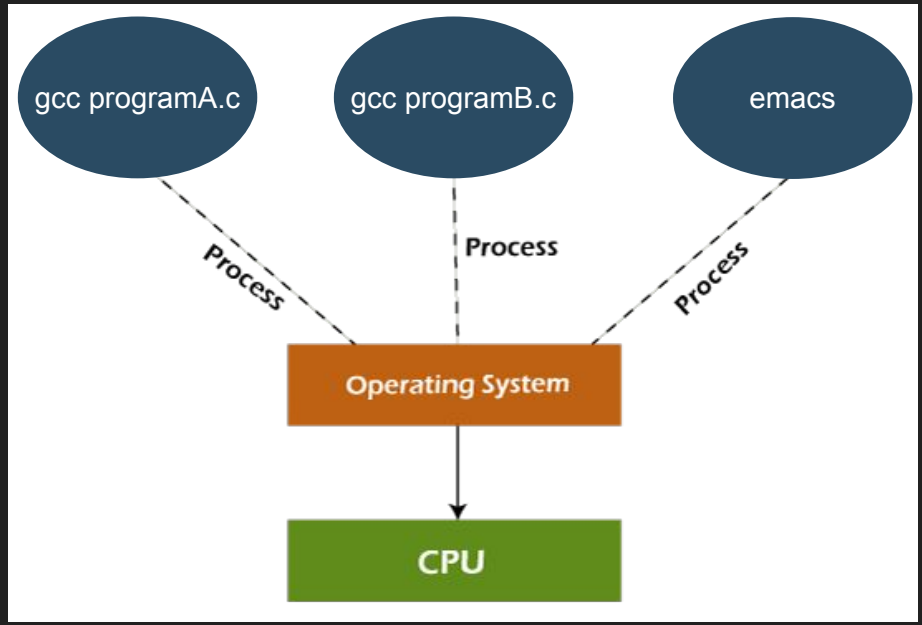
The Process Abstraction (cont.)

- A process (sometimes called a job or task) is an instance of a program running: Chrome, Fortnite, Notepad++, etc.
- Typically, multiple open windows of a program are still one process
  - Exception: Chrome creates a process for each site visited or tab opened
- Most OS can run multiple processes simultaneously
  - Notice the 35 Chrome processes ->



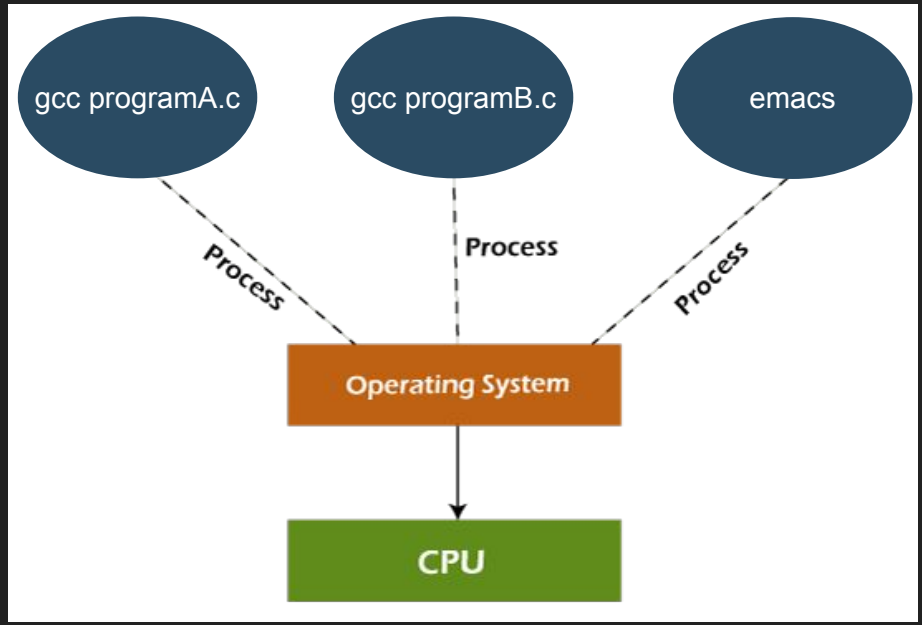
The Process Abstraction (cont.)

- The process abstraction ensures that data inside each process is unique to the process itself
  - Example: Starting an instance of gcc to compile programA.c and simultaneously starting another instance of gcc to compile programB.c
  - The two processes won't accidentally combine programA and programB together or get their data mixed up while compiling
  - The processes are kept separate



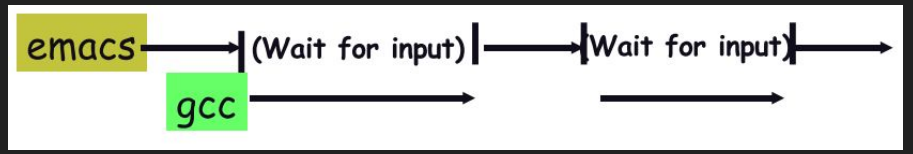
The Process Abstraction (cont.)

- This separation of processes states and data keeps our jobs as programmers “simple”
  - When gcc is compiling programA it doesn’t have to worry about any other instance of gcc that might be currently running
  - When new processes are ran the states and data are basically reset
- Running multiple processes allow us to better utilize our CPU’s computing power
  - Imagine only being able to run 1 process that spends 1 clock cycles running and 1,000,000 clock cycles waiting!



Multiple Processes Can Improve Performance

- Emacs (a text editor) spends most of its time waiting for you to type
  - So why not give up that CPU time spent waiting to a program that needs it (a compiler)

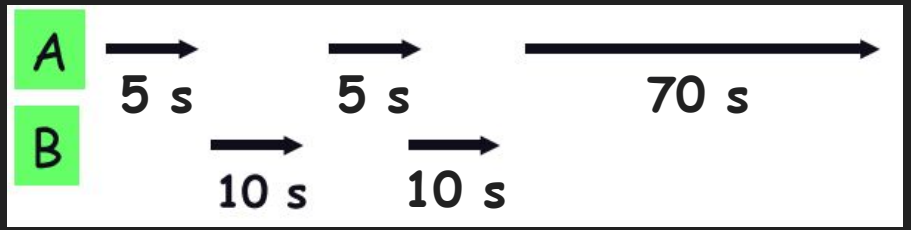


Multiple Processes Can Improve Performance (cont.)

- Old school computers could only run one process at a time
  - One right after the other
- Modern OSs allow us to switch between processes
  - From the user's perspective, the computer is running much faster!
- Note: Context switching (switching between processes) takes slightly more time than if ran one right after the other

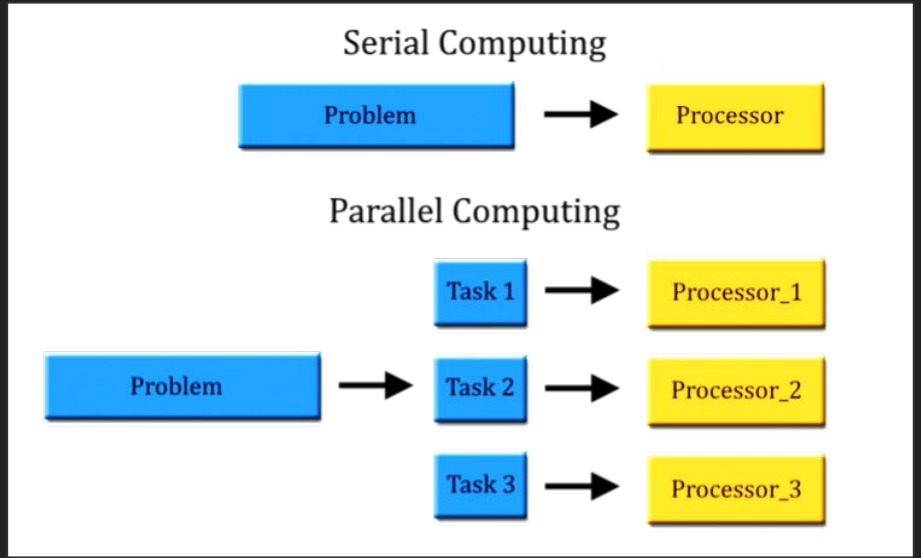
Old way:

Modern way:



Parallelism

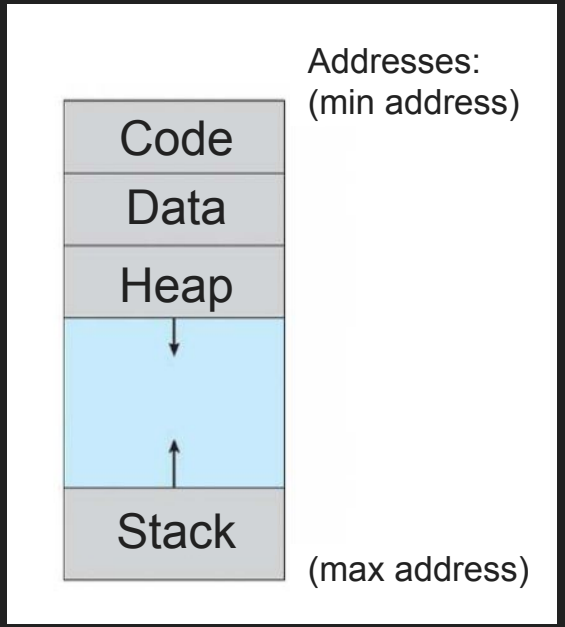
- Parallelism is the simultaneous execution of multiple tasks or processes in order to increase efficiency and speed
  - Imagine it takes a factory worker 1 day to make a product
  - If you want to make 100 products it will take this worker 100 days
  - Hire 100 workers it will take 1 day to make 100 products if they work perfectly in parallel
- Multi-core CPUs are how real parallelism is possible
- A 4 core computer can run 4 processes in parallel which yields 4x throughput



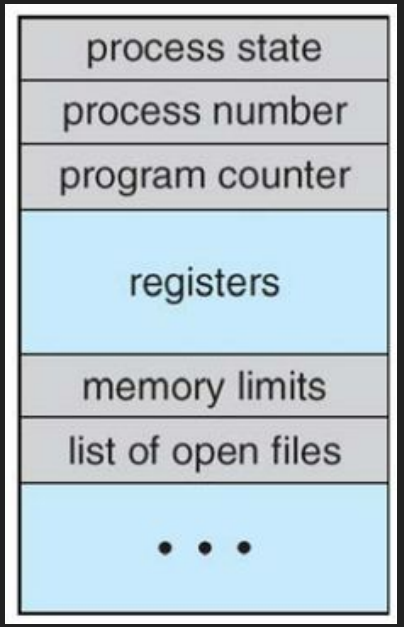
From the Process's Perspective

- Each process has its own unique view of the machine it's running on
  - Address space for code and data
  - Opened files (in memory)
  - "Virtual" CPU
    - The OS can take away CPU access whenever it wants even though the process does not know this
- One process of gcc is isolated from another process of gcc in memory
- To keep track of multiple processes information the OS uses a Process Control Block (PCB)
  - We'll take about this later

Process Memory (Process's View)

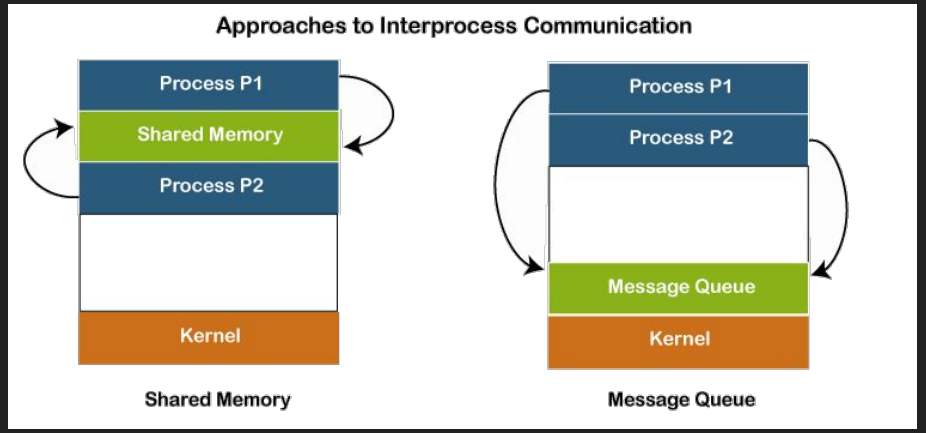


PCB (OS's View)



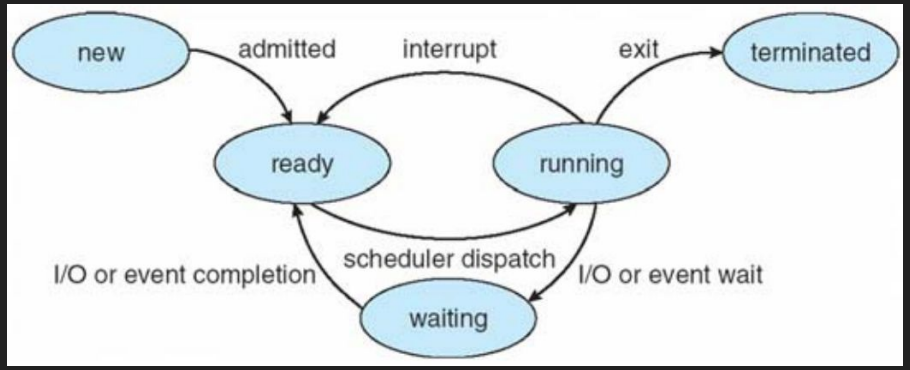
Processes Communicating

- Sometimes, we want processes to communicate with each other
- This is called Inter-Process Communication (IPC)
  - Shared files
    - i.e. edit a file with a emacs, save it, then compile that file with gcc
  - Shared memory
  - Message passing



Processes States

- New
  - A new process wants to run
  - The OS *admits* the process to the pool of other processes that want to run
- Ready
  - The process is now ready to run but must wait for the OS to dispatch it
- Running
  - The process is now executing on the CPU but it can be interrupted or forced to wait for I/O (i.e. keyboard or disk)
- Waiting
  - The process cannot run because it is waiting for something to happen
- Terminated
  - The process has finished or was forcefully terminated (i.e. closing out a window)



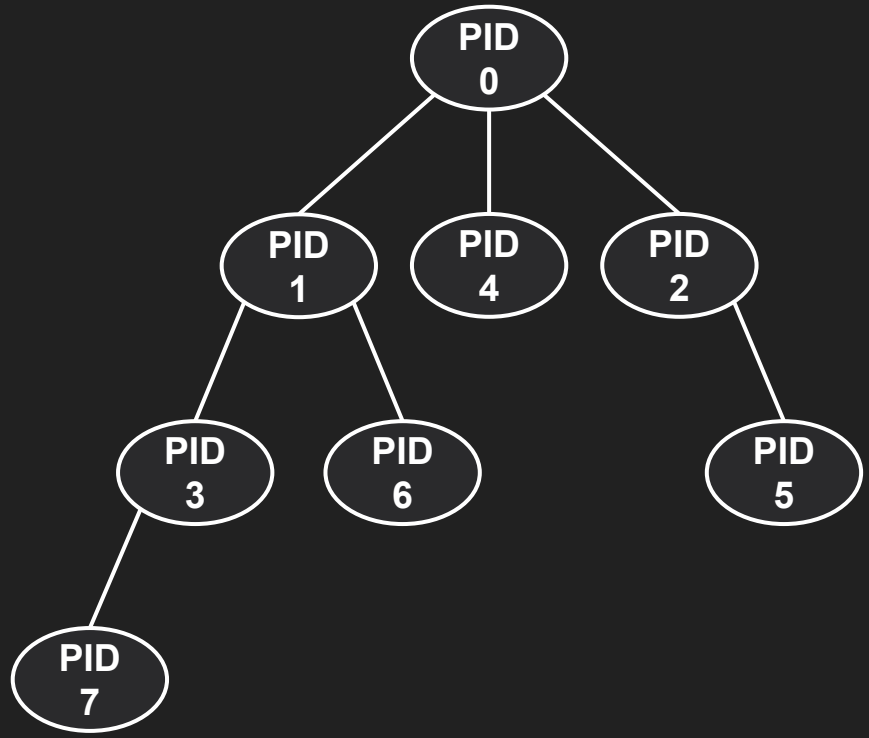
Creating a New Process in Linux

- Linux system calls make it easy to create new processes of the current program in C using the `fork()` function
- `int fork(void);`
  - Creates a process that is an exact copy of the current one that starts immediately after the fork() call
  - Returns process ID of new process to “parent”
  - Returns 0 to “child”
  - Returns negative if error

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello World!\n");
    return 0;
}

// "Hello World!" will get printed
8 times, why?
```

Creating a New Process in Linux (cont.)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("Hello World!\n");
    return 0;
}

// "Hello World!" will get
printed 8 times, why?
```



Creating a New Process in Linux (cont.)

- Your program can also wait for your child processes to finish before continuing using the `waitpid()` function
- `int waitpid(int pid, int *stat, int opt)`
  - `pid` - the process id to wait for, or -1 for any
  - `stat` - contains exit value of finished process
  - `opt` - custom options
  - Returns process ID or -1 on error

Creating a New Process in Linux (cont.)

- Sometimes you might want to execute a different program instead of creating a child process
- For this you can use `execve()`, `execvp()`, or `execlp()`
- `int execve(char *prog, char **argv, char **envp)`
  - `prog` - full path of program to run
  - `argv` - arguments the program should run with
  - `envp` - environment variables such as `PATH`, and `HOME`
  - Returns -1 if an error has occurred calling the program, i.e. program could not be found
  - The current process is now overtaken by the program we have specified
- `int execvp(char *prog, char **argv)`
  - Search `PATH` for `prog` using the current environment
- `int execlp(char *prog, char *argv, ...)`
  - List arguments one at a time, finishing with `NULL`

Terminating a Process in Linux

- You can forcefully terminate the current process (or another process) using `exit(status)` or `kill(pid, SIGTERM)`
- `void exit (int status)`
  - `status` - status code returned to `waitpid`
  - Terminates the current process
  - By convention, `status` of 0 is success, non-zero is error
- `int kill(int pid, int sig)`
  - `pid` - the process id you want to terminate/kill
  - `sig` - either `SIGTERM` (15) or `SIGKILL` (9)
  - `SIGTERM` - safely terminate the process but allow the process to do some clean up before it is forced to terminate
  - `SIGKILL` - immediately terminate the process and give no warning or time for process to clean up

Creating a New Process in Windows

- Creating a new process in Windows is not so straightforward and requires many arguments using the Windows API, `WINAPI`
- There are even multiple functions that can be called to create a process, which makes it more confusing:
  - `CreateProcess()`
  - `CreateProcessAsUser()`
  - `CreateProcessWithLogonW()`
  - `CreateProcessWithTokenW()`
  - `...`

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR lpApplicationName,
  _Inout_opt_  LPTSTR lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_         BOOL bInheritHandles,
  _In_         DWORD dwCreationFlags,
  _In_opt_     LPVOID lpEnvironment,
  _In_opt_     LPCTSTR lpCurrentDirectory,
  _In_         LPSTARTUPINFO lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION lpProcessInformation
);
```

From the OS's Perspective

- The OS maintains a data structure for each process called a Process Control Block (PCB)
  - Sometimes called Task Control Block (TCB)
- Tracks state of process
  - Running, ready, waiting, etc.
- Includes necessary information for running
  - Current registers being used, virtual memory mappings (what's in memory?), etc.
  - Open files
- Includes other details
  - User credentials, priority, etc.

PCB
(OS's View)

Scheduling Processes

- If more than 1 process needs to run, the OS must schedule them to run individually
  - If the machine has multiple cores and is capable of parallelism, multiple processes can run simultaneously
  - Without parallelism, your computer just appears to be doing many things at the same time because it's just switching between them very quickly
- The OS will look at its list of PCBs, find all that are "Ready", and decide which one gets to run
- How should the OS decide which one gets to run if there are multiple ready processes?
  - FIFO - First process that was ready is the first to run
  - Round Robin - Arrival time, burst time, and quantum
  - Priority - Highest priority runs first

Preemptive Multitasking

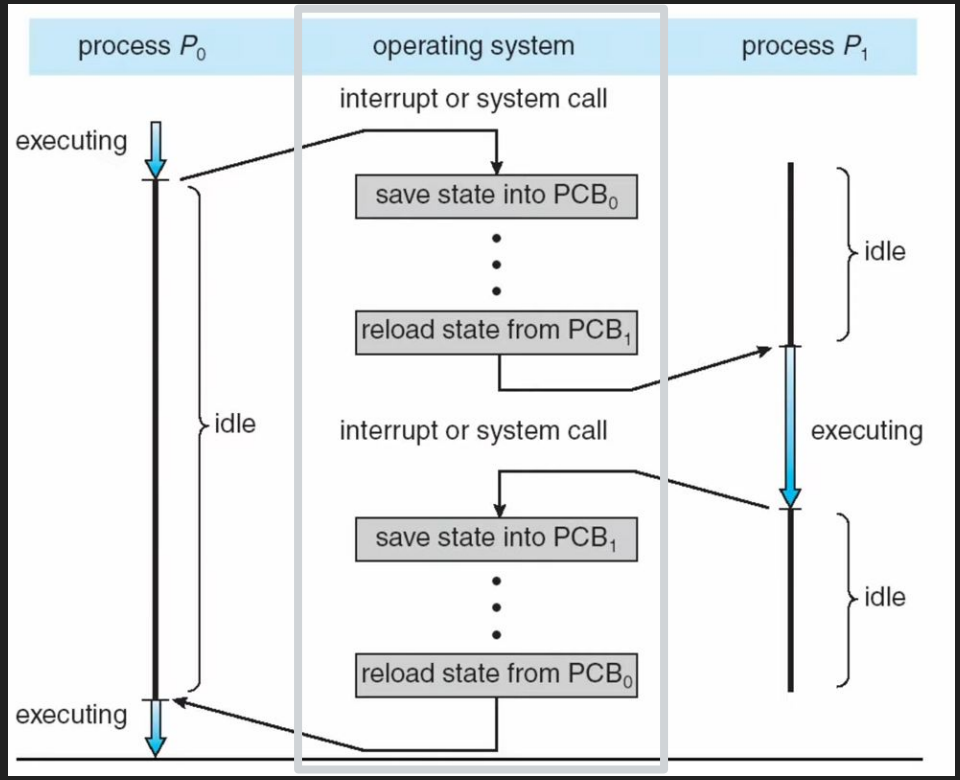
- A process can be preempted by the OS
  - Preempted means, pause for a short time, and allow something else to run, then resume later
  - Without preemption, 1 process could hog the CPU forever and your computer would appear frozen!
- The process may also preempt itself:
  - Makes a system call, waits to read disk, makes another process runnable (i.e. `fork()`), etc.
- Periodic timers interrupt the process
  - If the OS gives a fixed maximum amount of time the process can run, and the process exceeds this time, preempt the process so another process gets a chance to run
- Device interrupts
  - If process A is stuck waiting for our keyboard while process B is running, and we finally press a key, preempt process B and let the process A get a chance to capture the keyboard input
- The changeover between running processes is called *context switching*

Cooperative Multitasking

- A process must willingly yield or give up control for another task to run
  - Cooperative means every process must cooperate
  - At some point a process must give other tasks/processes a chance to run
  - The downside is if you have a process that does not play fair and hogs the CPU!
- This type of multitasking is much easier to implement because it does not involve any interrupts (timers)

Context Switch

- Switching between running processes is called context switching
- Process P_0 is currently executing but we want to run P_1:
  1. Save P_0's PCB data
  2. Reload P_1's PCB data
  3. Run P_1
- When the OS wants to switch back to P_0 the steps are the same
- If a process is not executing, it is not doing anything useful and is just waiting for the OS to give it a chance to run



Context Switch (cont.)

- Context switching is not free, it costs CPU time and memory (to store and access PCB data)
- Context switching is very hardware dependent
  - Which registers should get saved in the PCB and restored when we run the process?
    - What about floating point or special registers?
  - Are there flags that must be maintained?
  - Save/restore memory translations

Implementing Basic Multiprocessing in C

- To allow your OS to run multiple processes, we need to implement multiprocessing
  - Also known as multitasking
- Our OS will switch between multiple processes and execute each of them individually
- How does our system know when it needs to switch between processes?
  - Preemptive - hard to implement but better solution
  - Cooperative - easy to implement but worse solution

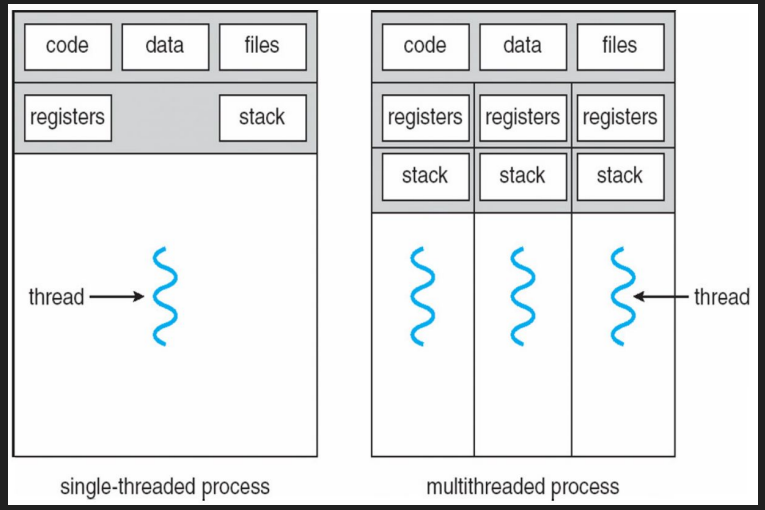
Implementing Basic Multiprocessing in C (cont.)

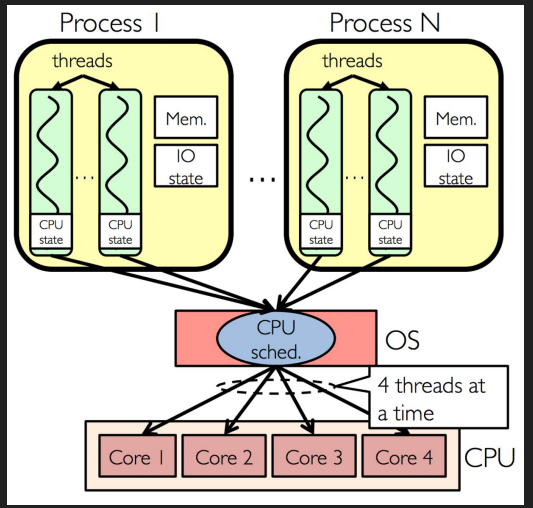
- To implement cooperative multitasking we must:
  1. Write a set of tasks (functions) that we want to execute
  2. Write a scheduler that can schedule tasks from a maintained list of PCBs
  3. Pass the tasks to a scheduler that will maintain PCBs for each task passed to it and construct a sequence of these tasks to execute them in order
  4. Write a `yield()` function that will save the state of the current task in the PCB in the scheduler and resume the state or start the next task in the scheduler
     a. Make sure to put the `yield()` function in each task (function) so it can give time to another task at some point
     b. Make sure to write an `exit()` function in each task so it can communicate to the scheduler when it should be removed from the task list
  5. The scheduler should loop through the list of PCBs until they have all exited

The remaining slides focus on ***threads*** not ***processes***

Threads

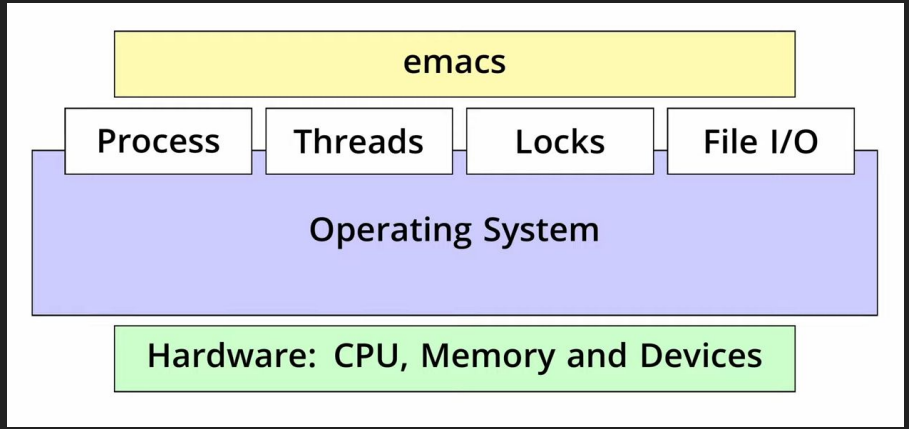
- A thread is a schedulable execution context
  - An execution context is all the information a CPU needs to execute a stream of instructions
- A process may have 1 or more threads
- Multithreaded processes share the same address space
  - All the threads share code, data, and any open files
  - Each thread has its own registers and stack
- Threads can be executed simultaneously by using CPU *cores*
  - A core is the CPU hardware
  - A thread is the instructions/data provided to the core





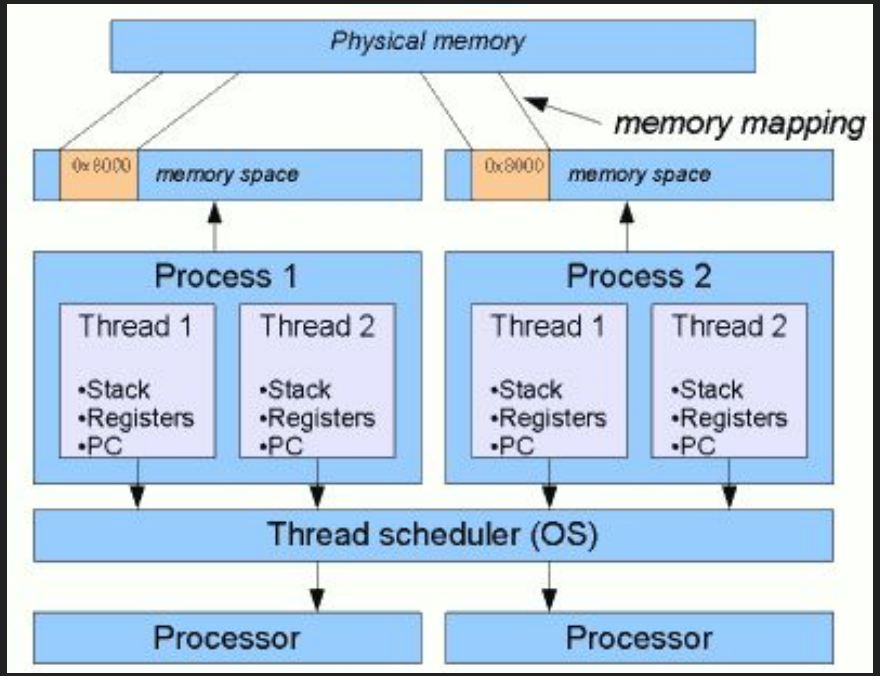
The Thread Abstraction

- Why separate the thread abstraction from the process abstraction?
  - What if you have 4 processes that want 4 threads?
  - What if you have 1 process that wants 4 threads?
- Keeping the threads as a separate system from processes allows for more flexibility
- The kernel usually has its own thread internally for every user mode thread or process
  - This internal thread keeps an eye on the user's processes/threads
  - Also has threads for every user currently logged into the system
- Just like processes, threads must be scheduled by the OS *or* by the process



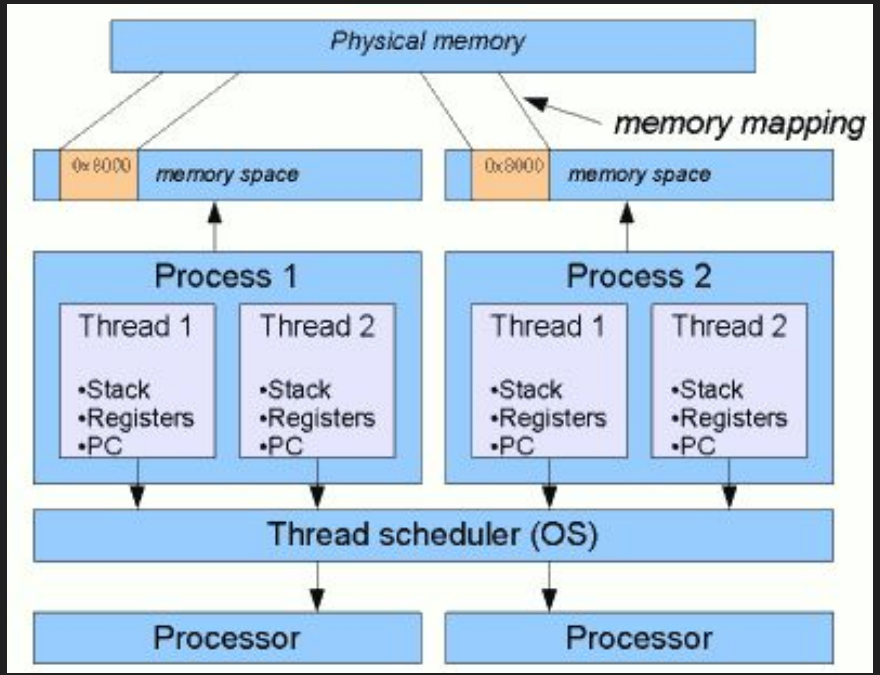
The Thread Abstraction (cont.)

- Threads are great for concurrency
  - Concurrency is executing multiple computations at the same time
  - This requires multiple physical CPU cores
- Threads are lighter-weight than processes
  - Threads share memory, files, etc.
  - Threads are easier to assign rather than spinning up a new process for each execution context
- Threads allow a process to do many things simultaneously
  - Execute code WHILE reading from a file, keyboard, etc.
  - I/O is really slow so this is very beneficial
  - Execute multiple pieces of code for faster results (physics simulations, games, etc.)
- The OS or user processes can create as much threads as they want and are not bound by how many physical cores exist



Multiprocess ≠ Multithreaded

- Threads and processes are two different things!
  - An OS can be multiprocess without being multithreaded and vice versa
- A multiprocess OS can run or switch between multiple processes
- A multithreaded OS can allow processes to use multiple threads
  - As long as there are CPU cores to use



Portable Operating System Interface

- The Portable Operating System Interface (POSIX) is a set of standards used to maintain compatibility between OSs
- POSIX standards allow for many OSs to use standardized methods for threads, I/O, file operations, and other OS functions
- The goal of POSIX is to define both the kernel and user-level application programming interfaces (APIs), along with command line shells and utility interfaces
    - This allows for software to be ported easier to other variants of Unix and other operating systems

Threads in POSIX

- `int pthread_create(pthread_t *thr, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`
  - Create a new thread `thr`, with attributes `attr`, to run a function `fn`, using arguments `arg`
  - If the syntax for `fn` looks confusing, just know it is a pointer to a function, that accepts a pointer as its arguments, and a pointer as its return value, each are void to allow the developer flexibility to choose any type they want to return or pass as arguments
- `void pthread_exit(void *return_value);`
  - Exit or terminate the current thread and return a value `return_value`
- `int pthread_join(pthread_t thread, void **return_value);`
  - Wait for thread `thread`, to exit, and capture its return value `return_value`
- `void pthread_yield();`
  - Tell the OS to allow other threads to execute and pause this thread
  - Helpful when this thread needs to wait for something (like I/O)
- There are more APIs that allow for thread synchronization and other benefits but this is all we care about for now

Implementing POSIX Compliant Threads

- The kernel can implement thread creation using a system call and maintains user threads with kernel threads
  - Every process that wants a thread must use this system call
  - The OS has the final say on whether or not a process gets a thread and controls access to threads
- Implement `pthread_create`, and other thread APIs as a system call
- Create the process abstraction in kernel
- Allow processes to utilize `pthread_create`
- When a process calls `pthread_create`, create a new thread that uses the same address space, file table, code, as the process
- Assigns one kernel thread to this user thread using one-to-one thread model

Kernel Threads vs User Threads

- There are two levels of threads that we use in an OS
- Kernel Threads
  - Managed and scheduled within the kernel
  - Has direct access to the CPU cores
- User Threads
  - Managed and scheduled within the user program
  - Allows multiple parts of the user program to execute simultaneously
- In order for a process to execute, there must exist a relationship between user threads and kernel threads

Contention Scope

- Contention scope is the level at which contention for resources occurs between threads
  - This can be in user space and/or kernel space
- There are two methods for scheduling threads:
  - Process-Contention Scope (PCS)
  - System-Contention Scope (SCS)

Process-Contention Scope

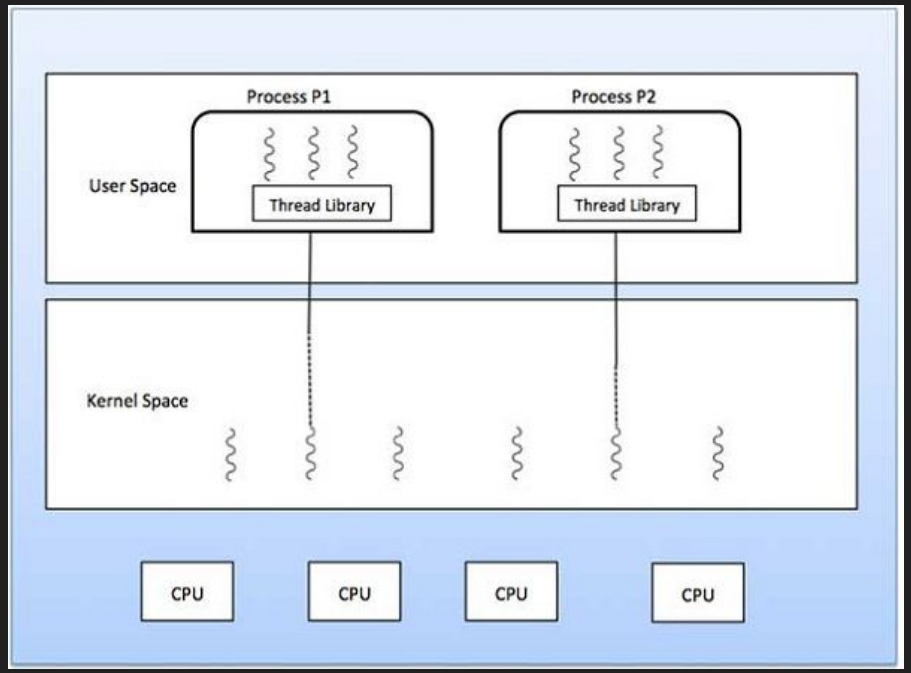
- In PCS, threads within the process compete with each other
- The scheduling mechanism for the thread is local to the process
  - The thread's library has full control over which thread will be scheduled
  - This is typically handled by the programmer assigning priorities to the threads when writing the multithreaded program

System-Contention Scope

- In SCS, threads within the kernel compete with each other
- The scheduling mechanism for the thread is within the OS
  - The OS determines, out of all kernel threads, which get to execute on the CPU(s)
- Many modern OSs only allow for SCS scheduling and the one-to-one thread model

Many-To-One Thread Model

- Each user-level thread is mapped to a single kernel-level thread per process
  - The operating system handles multiple threads as a single task
- This is implemented using a user level library rather than a system call
- If one user thread blocks, the entire kernel thread will become blocked
  - This prevents the other threads from getting a chance to execute
- Lacks true parallelism, the single kernel thread can only execute the user threads on 1 CPU



Implementing User Level Threads with Many-To-One

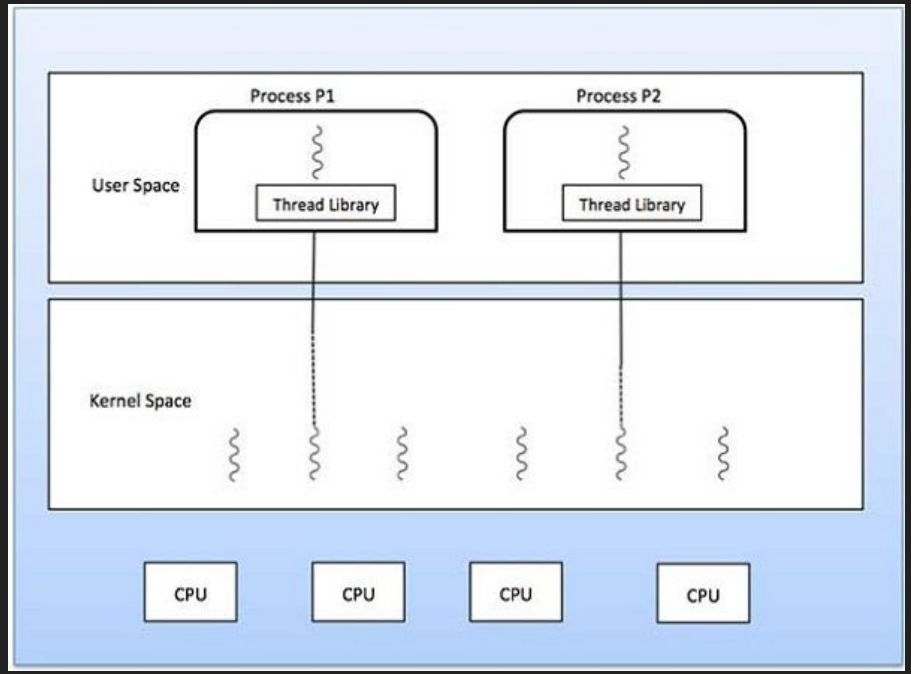
- In Many-To-One the user level threads are required to do a few things:
  - Allocate a new stack for each thread
  - Maintain a queue of *runnable* threads
    - Threads not waiting for I/O or other system calls
- A thread scheduler (running in user space) selects the next runnable thread from the queue and gives it to the kernel thread to execute
- When we want to execute a function using these threads we need to:
  - Write non-blocking versions of any blocking system calls so the thread can yield temporarily, a different thread can run, then resume this thread when done waiting (for I/O, network, storage, etc.)

Problems With User Level Threads in Many-to-One

- Our process only has access to 1 CPU core through the kernel thread
  - The only way to use multiple cores is to have multiple processes running
- Often, it is impossible to write a non-blocking disk read function (OSs don't often give you the tools to do so)
  - This blocks the thread, waiting for the disk
  - Once the thread is blocked it can't yield to other threads so the whole process is waiting for that 1 thread!
  - This gets even worse when working with virtual memory
- If one thread blocks another thread, the whole process may get stuck in deadlock

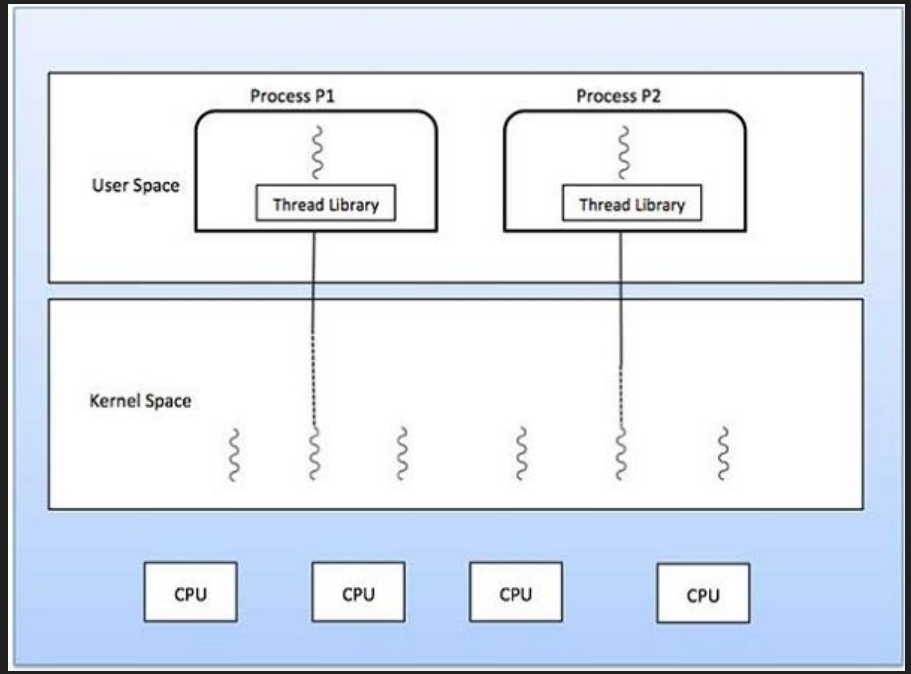
One-To-One Thread Model

- Each user-level thread corresponds to exactly one kernel-level thread
- Creates a direct association with a kernel-level thread managed by the operating system
- Threads can execute simultaneously on multiple processors
- When creating a user thread a kernel thread must also be created
    - This typically results in a lot of memory and processing overhead, creating kernel threads is very slow



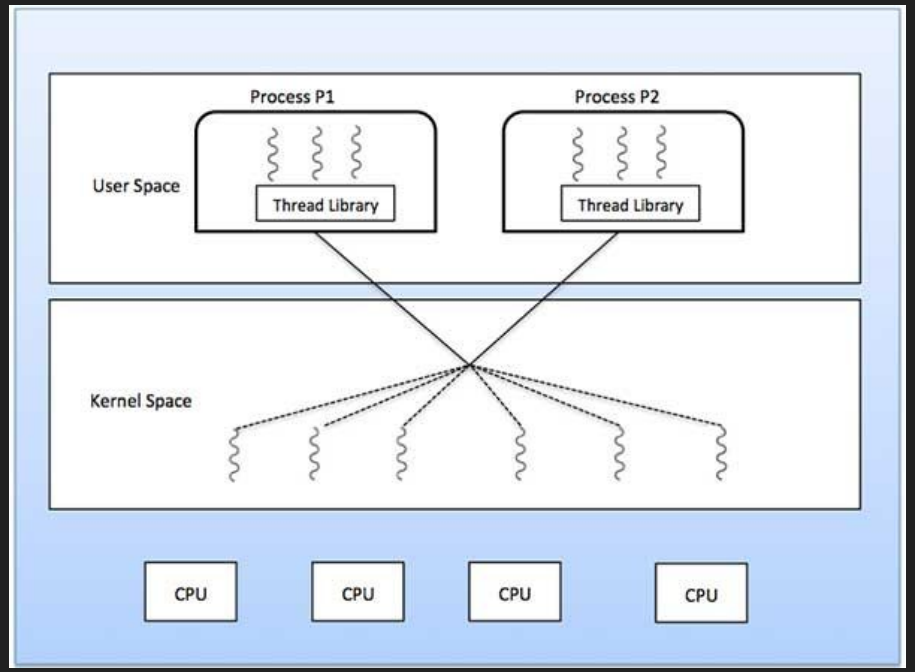
One-To-One Thread Model (cont.)

- What if a process wants 8 user threads on a 4 CPU system?
  - Only 4 can run in parallel
- What if 2 processes want 4 user threads each?
  - Only 1 process can execute its threads in parallel
- The application is required to limit its maximum user threads to the number of cores on the system



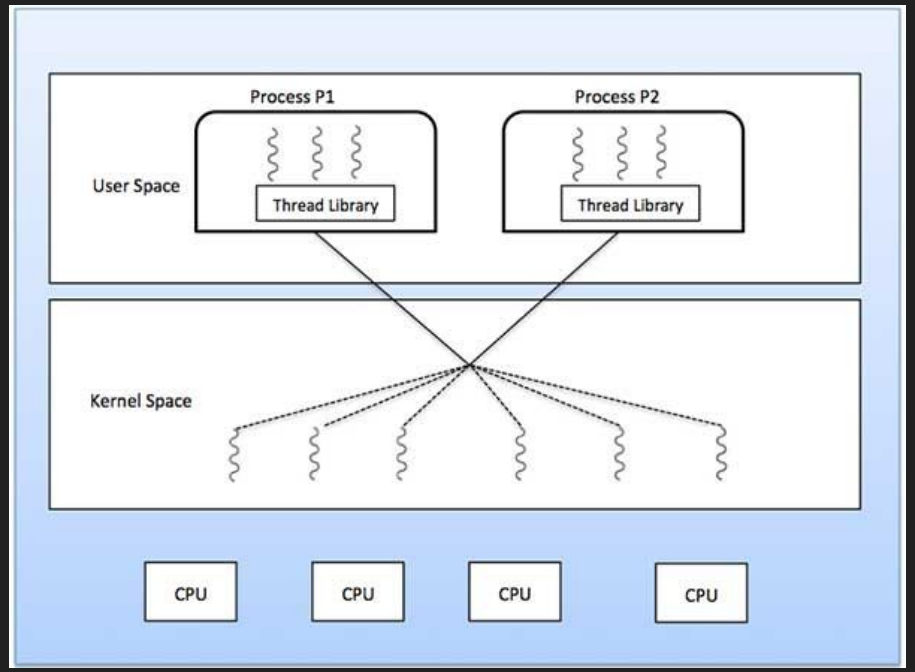
Many-To-Many Thread Model (aka n:m Threading)

- (n) number user-level threads are mapped to an equal or smaller number (m) of kernel-level threads
- Provides a balance between parallelism and efficiency, achieves better performance and flexibility
- User-level threads can run in parallel on multiple processors, and the operating system can manage and schedule a smaller number of kernel-level threads
    - Typically the kernel threads are between 1 thread to the number of cores on the CPU



Many-To-Many Thread Model (aka n:m Threading) (cont.)

- The OS determines how many kernel threads to assign
  - This may be specific to the application or to the hardware
- If a user thread blocks, the other user threads can continue to execute on separate kernel threads
- Many-to-many is not as commonly used due to its complexity



Problems With n:m Threading

- Our processes don’t really know what is happening with the actual CPU cores
  - Only the kernel knows how many CPU cores are available
  - The user space thread scheduler might constantly schedule a user thread that has a blocked *kernel* thread
- The kernel doesn’t know if one user thread is more important than another
  - If one user thread holds an important resource the kernel thread will eventually preempt and lock that thread for some time

Process Scheduling vs Thread Scheduling

- When using a multithreaded OS, threads are typically the main schedulable entity
- Each thread can, generally, be thought of as its own process
- On Linux and POSIX:
  - Threads of a multithreaded process are scheduled independently, rather than the whole process itself
  - Some threads may be allowed to be grouped, or be assigned priority, which may allow all the threads of one process to run simultaneously
  - For single threaded processes, only the single thread is scheduled amongst all the other threads in the system

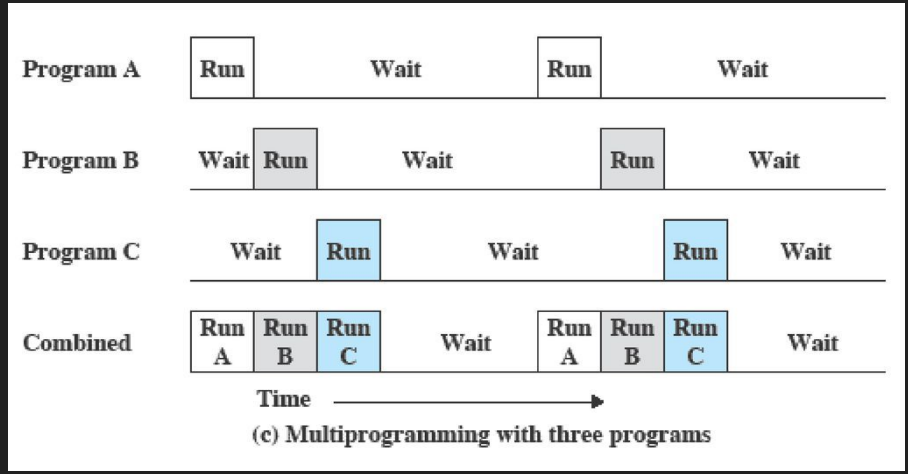
05 - CPU Scheduling

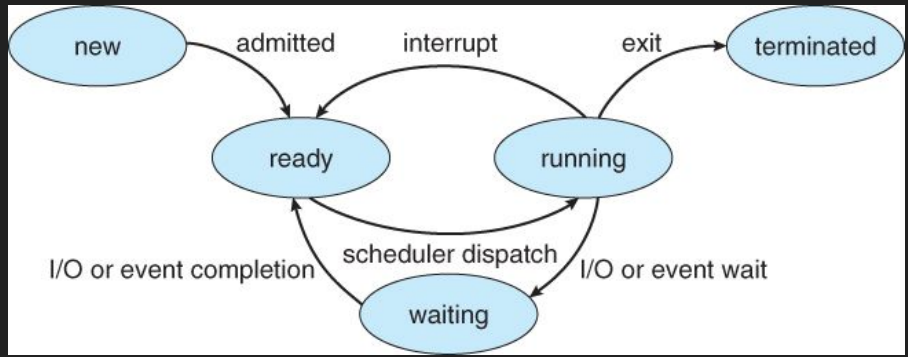
CEG 4350/5350 Operating Systems Internals and Design

Max Gilson

Scheduling Processes

- Multiprocessing allows us to better utilize the CPU and execute multiple processes “at once”
  - In reality, only one process is running on the CPU at a time
  - The OS switches between multiple processes very quickly
  - If our hardware has multiple CPU cores we can execute multiple processes simultaneously
- If a process needs to wait for I/O it can give another process a chance to run
- All of our processes reside in a state queue
  - i.e. all ready processes live inside a “ready” queue that the OS will pull from when choosing to execute a new process





Scheduling Processes (cont.)

- Long Term Scheduling
  - How many processes will the OS allow to exist
  - We are limited by memory so we can’t have an infinite amount
- Short Term Scheduling
  - How should the OS select a “ready” process from the ready queue

Short Term Scheduling

- The process scheduler resides in the kernel and can make a decision when:
  - A process switches from running to waiting
    - i.e. if the process wants to access I/O
  - An interrupt occurs
    - i.e. if a process wants the keyboard data and the keyboard sends and interrupt
  - A processes is created or terminated
- Non-preemptive scheduling method must wait for one of the above events to occur before switching processes
- Preemptive scheduling method allows the scheduler to interrupt a process

What Makes a Good Scheduling Algorithm?

- CPU Utilization
  - How much is the CPU being utilized to its fullest potential?
  - Ideal: as close to 100% as possible
- I/O Utilization
  - How much of our I/O or storage is being used to its fullest potential?
  - Ideal: as close to 100% as possible
- Throughput
  - How fast are the processes completing?
  - Ideal: very fast
- Turnaround Time
  - How much time are processes taking to complete from “new” to “terminated”?
  - Ideal: very short
- Waiting Time
  - How much time do processes stay in the “ready” queue?
  - Ideal: very short
- Response Time
  - How much time between a process being “ready” and its next I/O request?
  - Ideal: very short

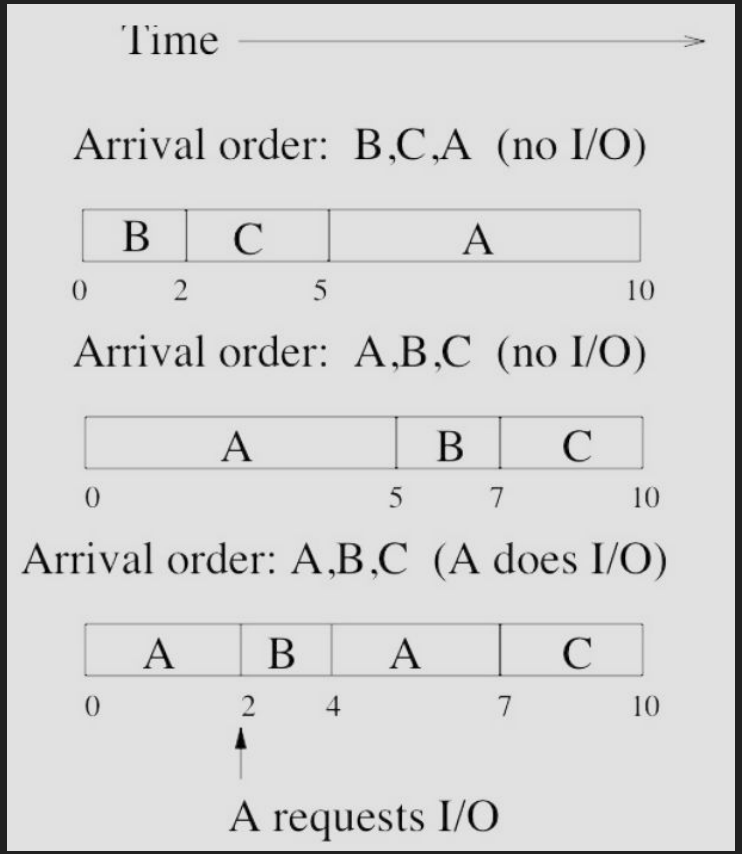
Scheduling Policies

- Ideally, you want a process scheduler to optimize all criteria but this is not realistic
- Instead, choose a scheduler that optimizes your most important metric(s):
  - Shortest response time
    - Good for when the user has to interact with the system (i.e. moving the mouse, typing on keyboard, loading screens, etc.)
  - Lowest response time *variance*
    - Having a consistent response time might make the system less frustrating to work with
  - Maximize throughput
    - Minimize OS overhead, context switching overhead, and efficiently use I/O and resources
  - Minimize waiting time
    - Give each process the same amount of CPU time but this may increase response time

First In First Out (FIFO)

- The scheduler will execute processes in the order in which they are created or arrive
  - Not necessarily the order that they are added to the ready queue
- Assume processes only release the CPU when they are waiting for I/O
- Notice, A still gets to finish before C
  - Even though A had to wait for I/O, it still arrived before C
- This method is cooperative (requires process to release the CPU)
- Advantage: simple to implement
- Disadvantage: process's time spent waiting is highly variable
- Disadvantage: I/O bound processes are not prioritized and forced to wait for CPU bound processes

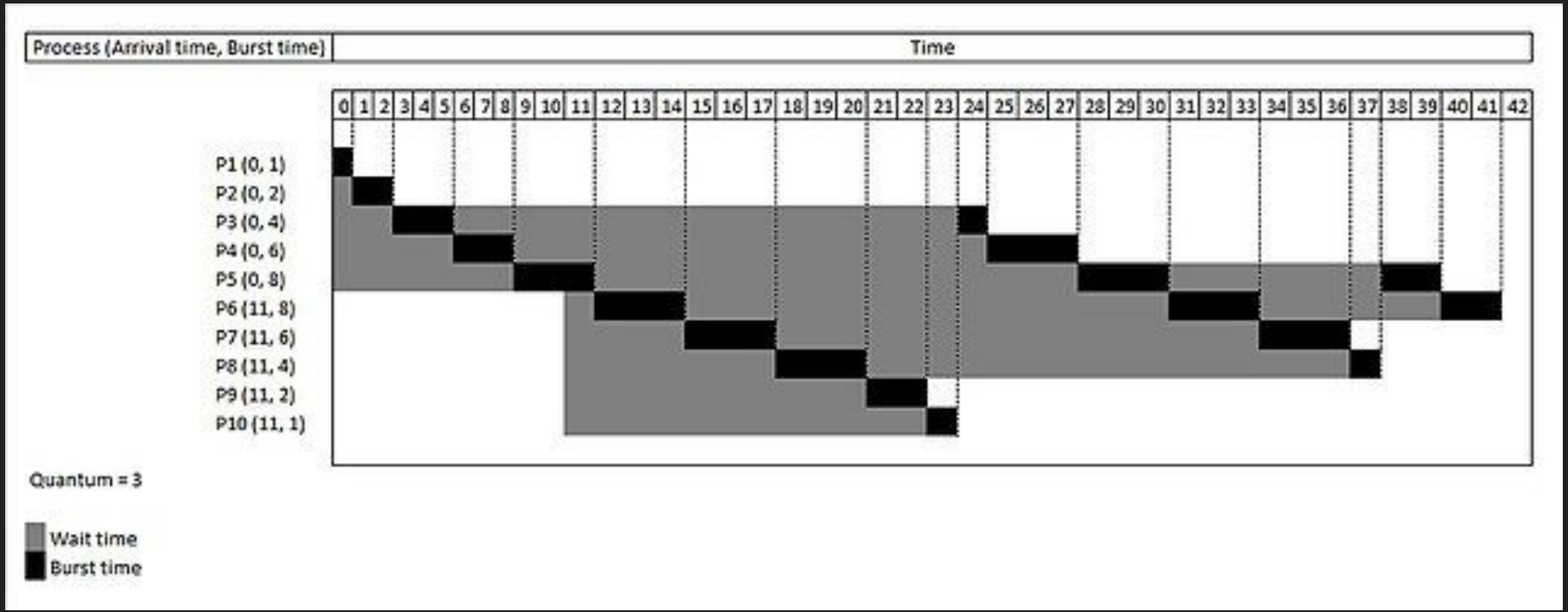
First In First Out (FIFO) (cont.)



Round Robin (RR)

- The scheduler schedules processes evenly or fairly giving each process a *quanta* of time to run on the CPU
  - The quantum is a predefined slice of time (e.g. 100 clock cycles)
  - The quanta is how many of these slices a process may use at a maximum (e.g. 3 quanta max. = 300 clock cycles max. CPU time)
- This method is preemptive (uses a clock to force processes to stop)
- Quanta too large - processes spend way too much time waiting
- Quanta too small - most of your time is spent context switching
  - Try to find a quanta where context switching is 1% of the total quanta
- Advantage: Consistent response time, all processes have equal time to access CPU
- Disadvantage: Process's average time spent waiting can be long (system may feel slow with 100's of processes running)

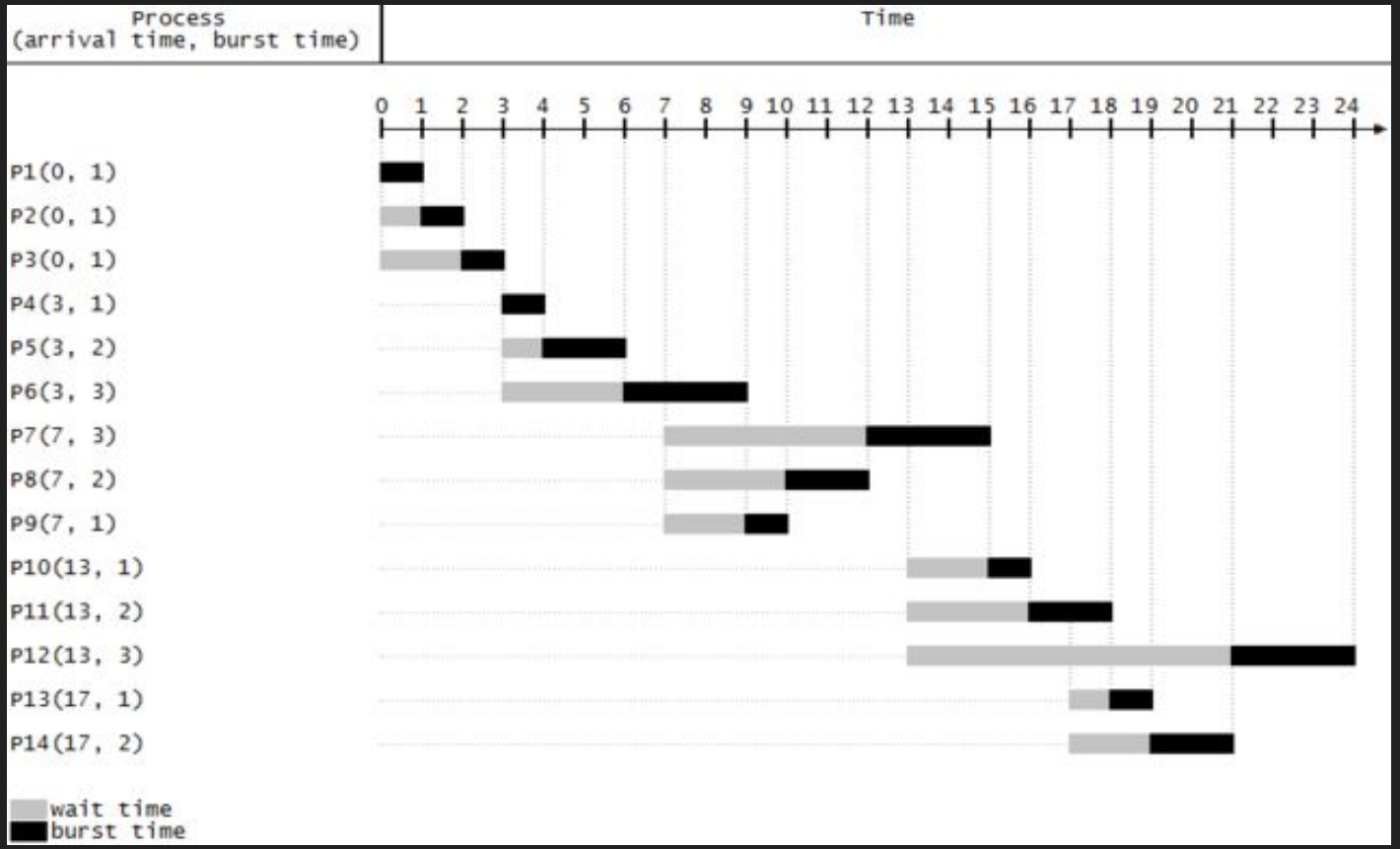
Round Robin (RR) (cont.)



Shortest Process Next (SPN)

- The scheduler schedules the process with the least (expected) amount of work (CPU time) to run until the process has an I/O request or terminates
- Can be preemptive or cooperative
  - Preemptive version uses shortest remaining time first
  - Preemptive version prioritizes I/O bound jobs over CPU bound jobs
- Advantage: Optimally minimizes (on average) process's waiting time
  - This is an optimal solution for this metric
- Disadvantage: impossible to predict how long a process needs to run
- Disadvantage: long processes may never get a chance to run

Shortest Process First (cont.)

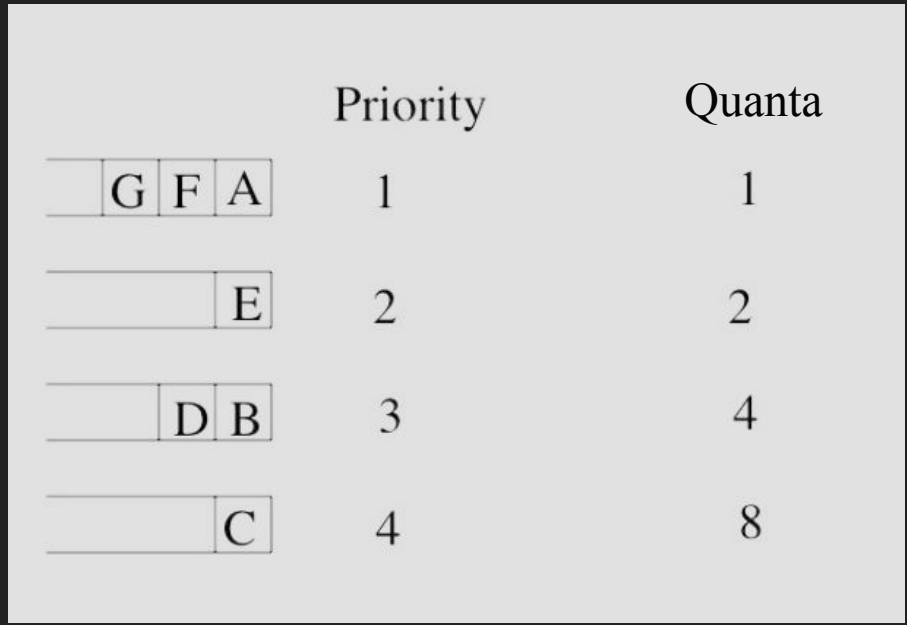


Multilevel Feedback Queue (MLFQ)

- The scheduler schedules based on past behavior of the process to attempt to predict the future and assign process priorities
- Used in most modern UNIX like systems
  - Overcomes the limitations of preemptive SPN
- If a process was I/O bound in the past, it is likely to be I/O bound in the future
- The scheduler can favor jobs that have used the least amount of CPU time (I/O bound processes), thus approximating SPN
- This policy is adaptive because it adapts to how the processes run and changes it scheduling behavior based on this past history
- The kernel keeps track of how often a process waits for I/O and prioritizes the processes that often access I/O

Multilevel Feedback Queue (MLFQ) (cont.)

- Multiple queues with priority based on predicted run time
- Use RR scheduling for each priority queue
  - Once finished, run the next priority level queue with RR
    - This can lead to starvation!
- Increase RR quanta exponentially for each priority level
  - This gives CPU bound processes more time to get stuff done



Multilevel Feedback Queue (MLFQ) (cont.)

- New processes start in the highest priority queue
- If the process wants to exceed its quanta, decrease the process's priority level by 1
  - The process is using more CPU time than expected
- If the process does not exceed its quanta, increase the priority level by 1 up to the highest priority level
  - The process is using less CPU time than expected
  - This can happen if the process begins waiting for I/O very quickly
- I/O bound jobs become higher priority
- CPU bound processes become lower priority

Improving Fairness

- Since SPN is optimal, but unfair and can starve long processes, increasing fairness must increase the waiting time
- Possible solutions:
  - Give each queue a fraction of CPU time
    - This is only fair if priority level queues have the same number of processes
  - Adjust the priority of processes if they are not getting ran
    - Unix originally took this approach
    - Avoids starvation but waiting time suffers when system is overloaded
      - This is because every process becomes high priority!

Lottery Scheduling

- Give each process a set of tickets
  - Assign more tickets to short running processes
  - Assign fewer tickets to long running processes
- Each quantum, randomly pick a winning ticket
- On average, CPU time is proportional to the number of tickets given to a process
- This approximates SPN while avoiding starvation since every process has a ticket that can be picked to run
- As the CPU load changes, adding or removing processes affects all other processes proportionately
  - Regardless of how many tickets each process has

Lottery Scheduling Performance

- In the example to the right, assume:
  - Short jobs get 10 tickets
  - Long jobs get 1 ticket
- How do we know how long the processes run?
  - Look at past history and estimate
- How do we determine how many tickets to give out?
  - Let the user decide (basically allows user to choose priority)
  - Let the OS determine based on the time

# of short processes / # of long processes	% of CPU each short process gets	% of CPU each long process gets
1/1	91% (10/11)	9% (1/11)
0/2	N/A	50% (1/2)
2/0	50% (10/20)	N/A
10/1	~10% (10/101)	< 1% (1/101)
1/10	50% (10/20)	5% (1/20)

Scheduling Algorithm Summary

- FIFO
  - Not fair, and average waiting time is poor
- Round Robin
  - Fair, response time variance minimized, but average waiting time is poor
- SPN
  - Not fair, but average waiting time is minimized assuming we can accurately predict the length of the next CPU burst
  - Starvation is possible
- Multilevel Queuing
  - An implementation (approximation) of SJF.
- Lottery Scheduling
  - Fairer with a low average waiting time, but less predictable.

06 - Main Memory

CEG 4350/5350 Operating Systems Internals and Design

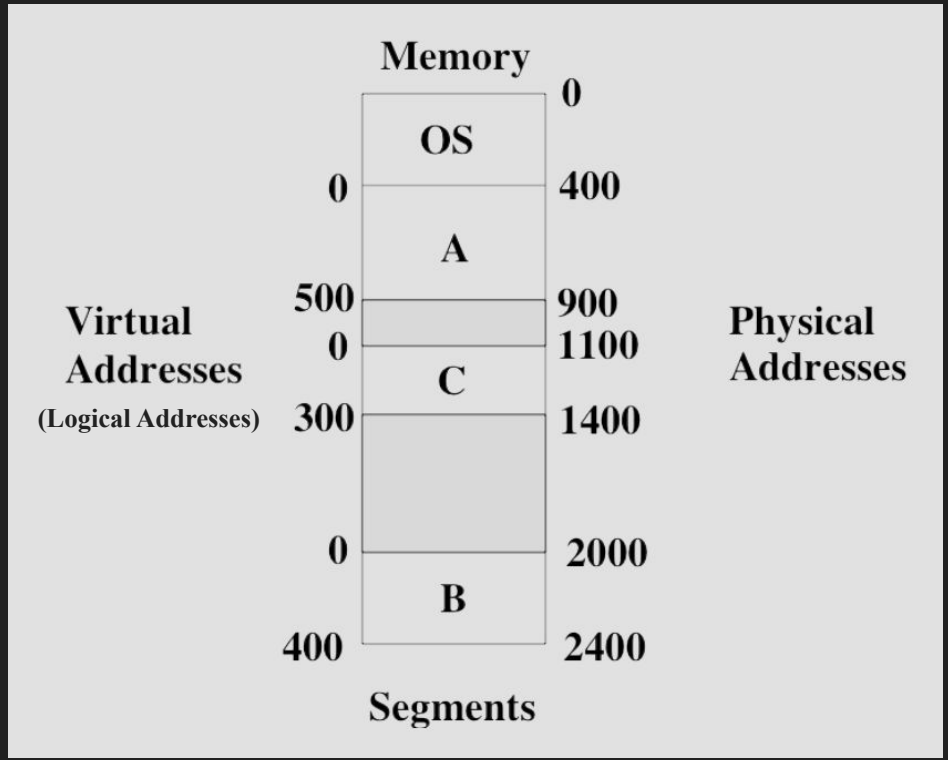
Max Gilson

Background

- All of our programs exist on our disk
- Before we can execute them they have to be brought into main memory (RAM) by the OS
- The CPU will fetch instructions from RAM to execute our program
- Your OS project does this already for your kernel
    - Before your kernel can be executed, it must be loaded off of the floppy disk

Memory Terminology

- Segment - a chunk of memory assigned to a process
- Physical address - a real address in memory that corresponds to an actual physical memory location
  - e.g. x000B8000 or xFFFF1234
- Virtual address - an address relative to the start of the process's address space
  - e.g. if a process exists within the range x1234 to x1FFF it will have virtual addresses from x000 to xDCB
  - This gives our programs more flexibility when accessing memory
  - Also called "logical address"
- Contiguous memory - memory that is contained within one region, one address after the other

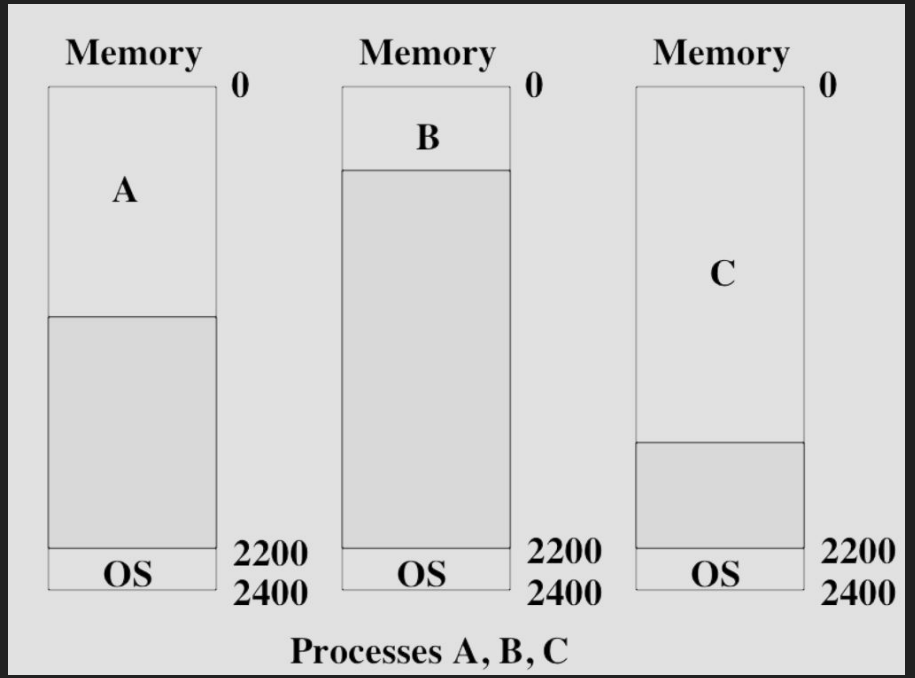


Where Do Addresses Come From?

- How do programs generate addresses for instructions and data?
- 3 different methods:
  - Compile time
    - The compiler decides where the program starts in memory at a fixed physical memory address
    - The OS does nothing
  - Load time
    - The compiler decides a starting address
    - The OS determines where this starting address gets placed in physical memory
    - Once loaded, the process does not move in memory
  - Execution time
    - The compiler decides a starting address
    - The OS determines where this starting address gets placed in physical memory
    - When the process is running, the OS will translate the virtual addresses to physical addresses

Uniprogramming

- Uniprogramming - running only 1 process + OS
- Assume the OS gets a fixed part of memory up to the highest address (DOS-like)
- Process is loaded at physical address x00000000
  - Process executes in a contiguous section of memory
- Maximum address = Memory Size - OS Size
- Simple but does not allow for overlap of I/O and CPU usage

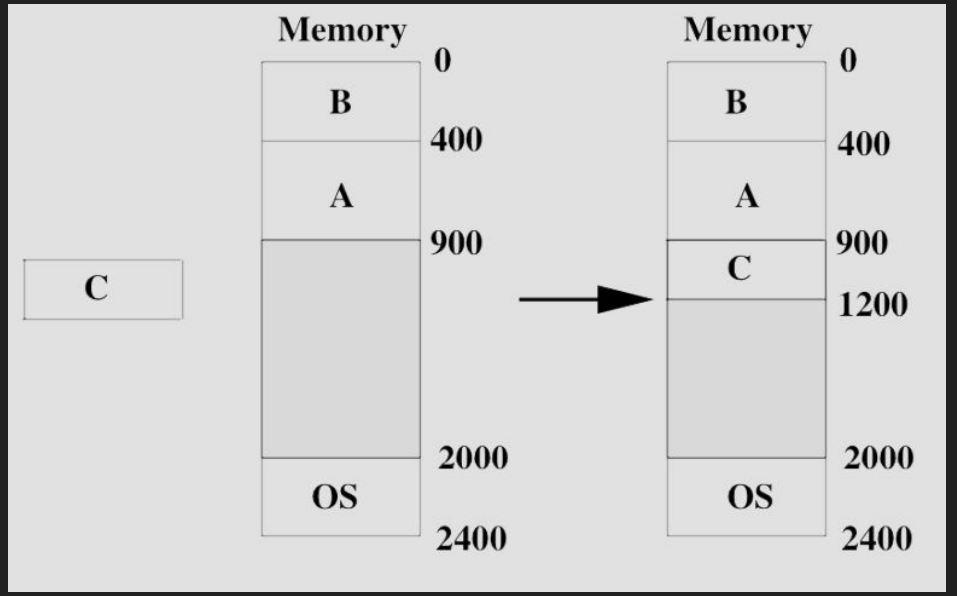


Requirements of Multiprogramming

- Transparency
  - We want multiple processes to coexist in physical memory
  - No process should be aware memory is shared
  - Processes should not care where they are in memory
- Safety
  - Processes must not corrupt each other or the OS
- Efficiency
  - Performance of CPU and memory should not be degraded

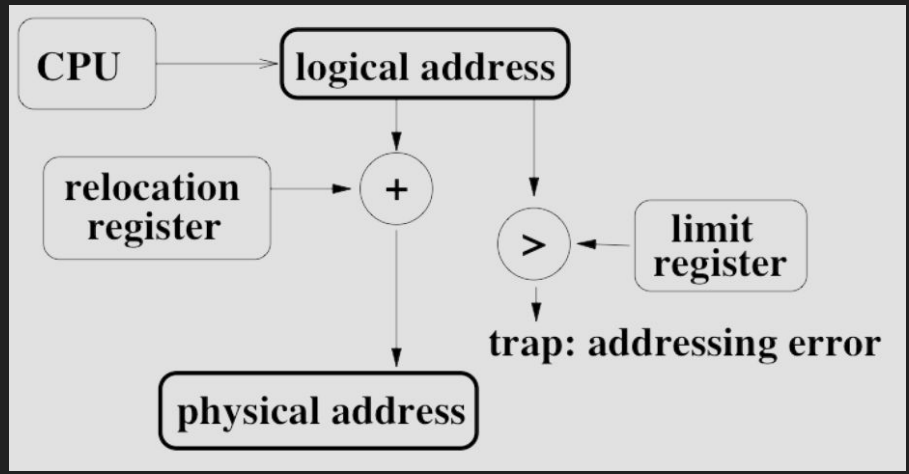
Relocation

- Relocation is moving a process to any location in memory
- Assume the OS gets a fixed part of memory up to the highest address (DOS-like)
- Assume at compile time the process starts at address 0 with: Maximum address = Memory Size - OS Size
- Base address
  - The first physical address of the process
- Limit address
  - The last physical address of the process



Types of Relocation

- Static Relocation
  - When the process is loaded, the OS offsets all addresses to reflect the process's new location in memory
  - Once the process is assigned to memory it cannot be moved
    - Moving would require re-offsetting all the addresses for every instruction in the program!
- Dynamic Relocation
  - Hardware has base register that gets added to virtual address, the result is the physical address
  - Hardware compares address with limit address
    - If the address exceeds the limit address, execute a trap service routine to handle addressing error and ignore physical address
  - Assume all logical addresses are positive



Benefits of Dynamic Relocation

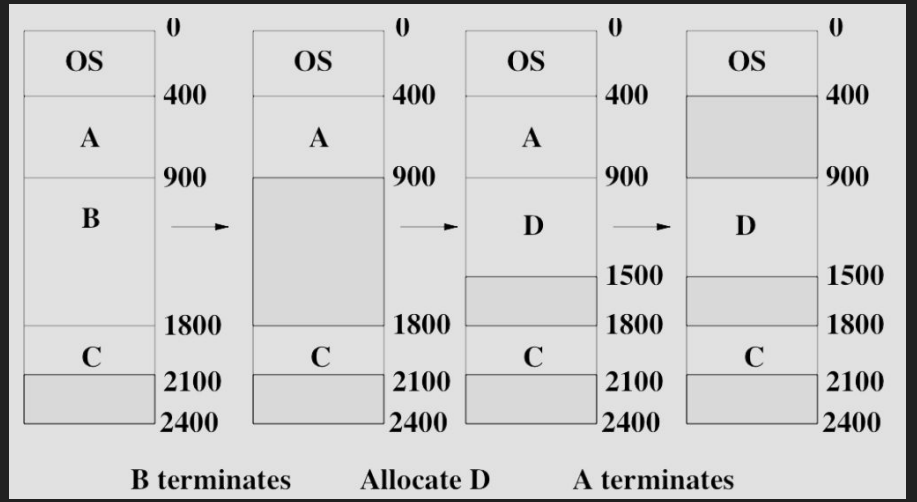
- Advantage:
  - OS can move a process in memory much easier
  - OS can allow a process to grow over time
  - Hardware requirements are simple
    - 2 extra registers, addition, and comparison
- Disadvantage:
  - Extra hardware may increase time to access memory
  - Sharing memory between processes is impossible
    - Each process is restricted to its segment
  - All running processes must coexist in physical memory
  - We are still using contiguous memory

Benefits of Dynamic Relocation (cont.)

- Transparency
  - Processes are unaware of other processes in memory
- Safety
  - Each memory access is checked by hardware
- Efficiency
  - Slightly slower but still very fast
  - Moving a process to a new location in memory is very slow
    - This may be necessary if a process grows

Memory Allocation

- As processes start, grow, and terminate, the OS tracks all used and unused memory
- When a new process starts, the OS must decide where the process goes into memory
  - A *hole* is a location in memory that is not filled
  - The OS tries to fill these holes with new processes
- What if we tried to allocate B again in the last diagram?
  - Hint: it won't fit

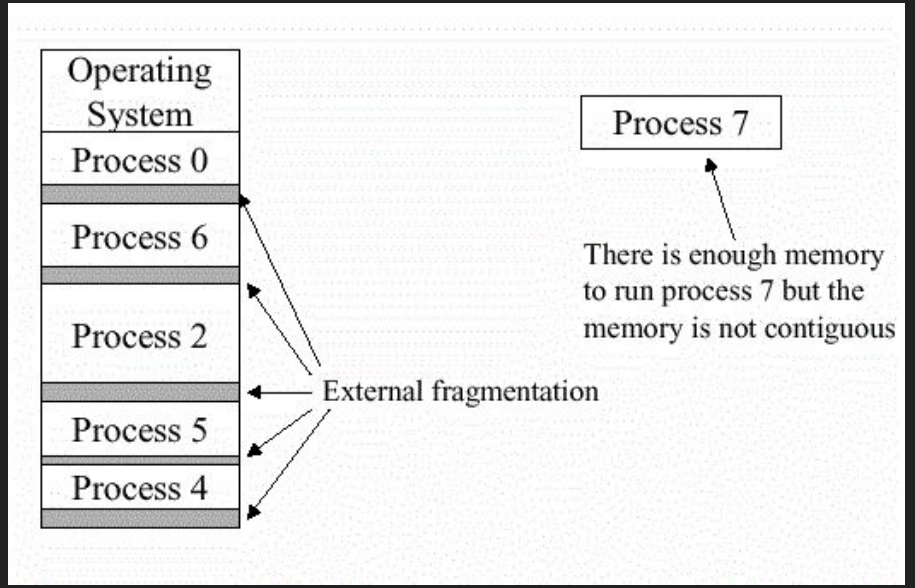


Memory Allocation Policies

- First Fit
  - Allocate the process to the first hole it fits in
  - Generally faster than Best Fit
- Best Fit
  - Allocate the process to the smallest hole that the process still fits in
  - Generally better storage utilization than Worst Fit
- Worst Fit
  - Allocate the process to the largest hole in memory
  - The remaining memory after the process might be large enough to fit another process or allow the process itself to grow
- For Best Fit and Worst Fit, the OS must search the entire list of holes to find the desired location to put the process
  - This can be slow if there are 1,000’s of holes!

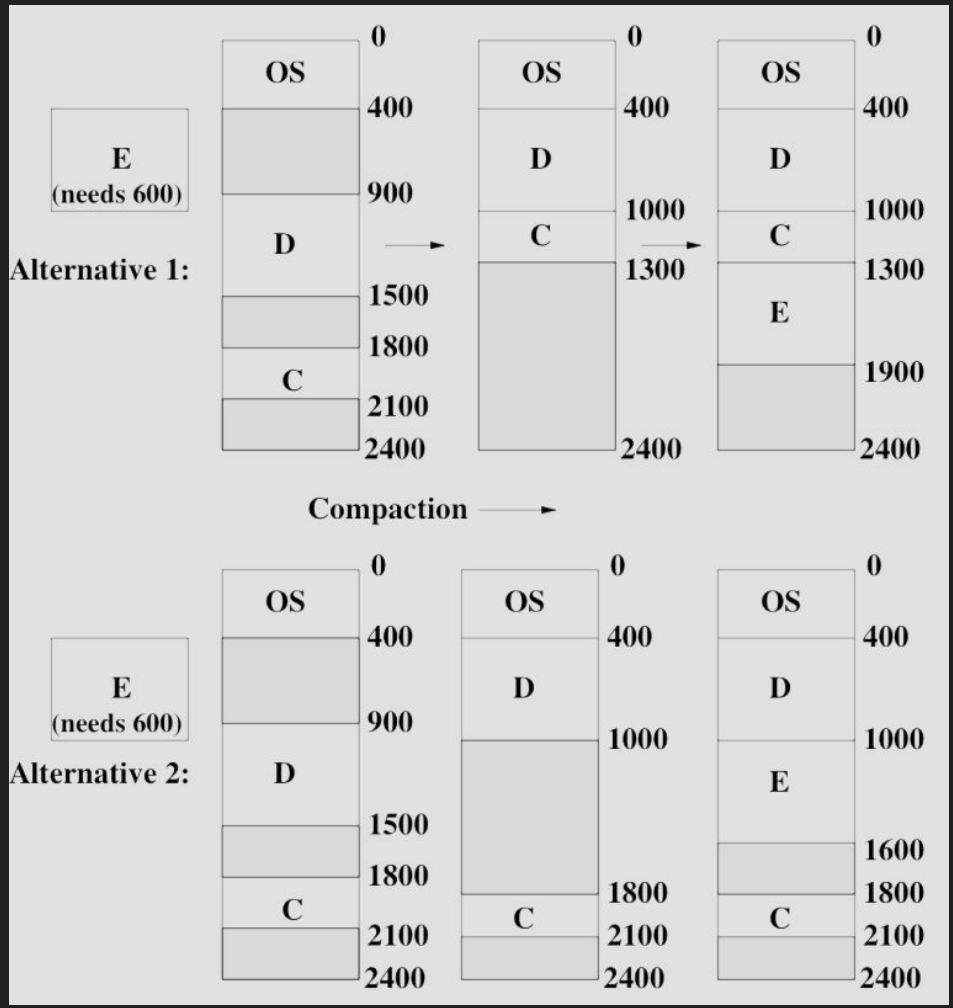
Fragmentation

- External Fragmentation
  - The memory between processes that is too small to be used
- Internal Fragmentation
  - Occurs if memory split into fixed sized chunks
  - If a process occupies one fixed sized chunk, but doesn't use it all



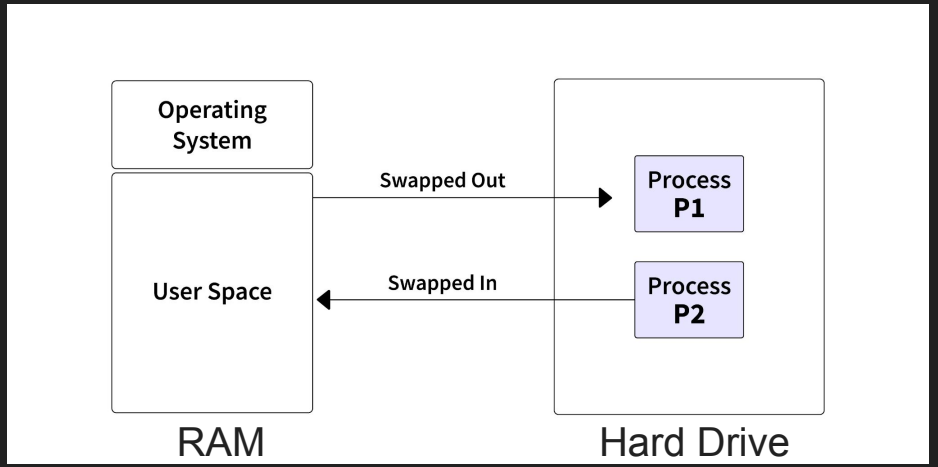
Compaction

- Compaction involves moving processes inside of memory to remove external fragmentation
- Advantage:
  - We can now load another process into memory
- Disadvantage:
  - Expensive operation, takes a lot of time
- Question:
  - We only need processes in memory if we're executing them. So, why do all 3 processes have to be in memory?



Swapping

- When we aren't executing a process, save it to the disk so we can resume it later
  - This frees up memory for other processes to use
- If the process becomes active again, reload it into memory
  - If using static relocation, process must go into same memory location
  - If using dynamic relocation, process can go anywhere in memory and OS must update relocation/limit registers
    - Compaction can be performed at this time
- Swapping processes takes a long time
  - If our scheduler wants to execute a process that is on disk, it should load that process while scheduling other processes first

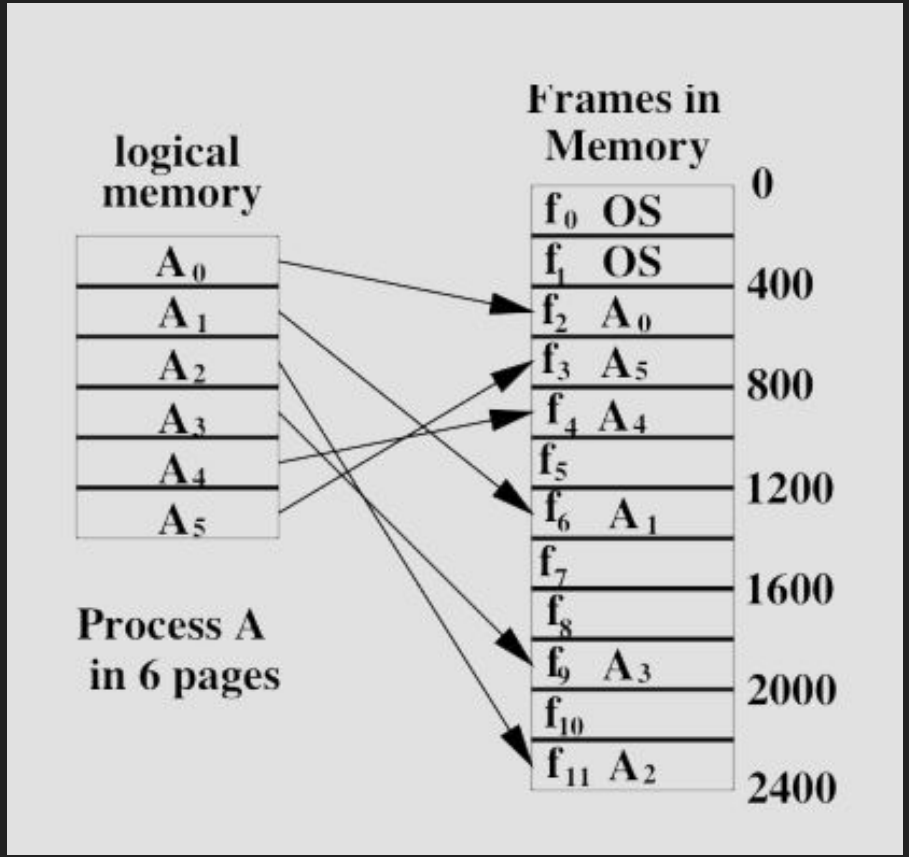


Paging

- Let’s assume that processes spend 90% of their time accessing 10% of their memory
  - Let’s only keep that 10% of their memory *in* memory unless the process needs more
- The logical memory of a process is contiguous
- The physical memory of a process is NOT contiguous
- Memory is divided into fixed size pages
  - This eliminates external fragmentation
  - Internal fragmentation still exists, about ½ a page is wasted per process
- Each process has its own table that translates pages to frames in memory
  - This table is managed by the OS

Paging (cont.)

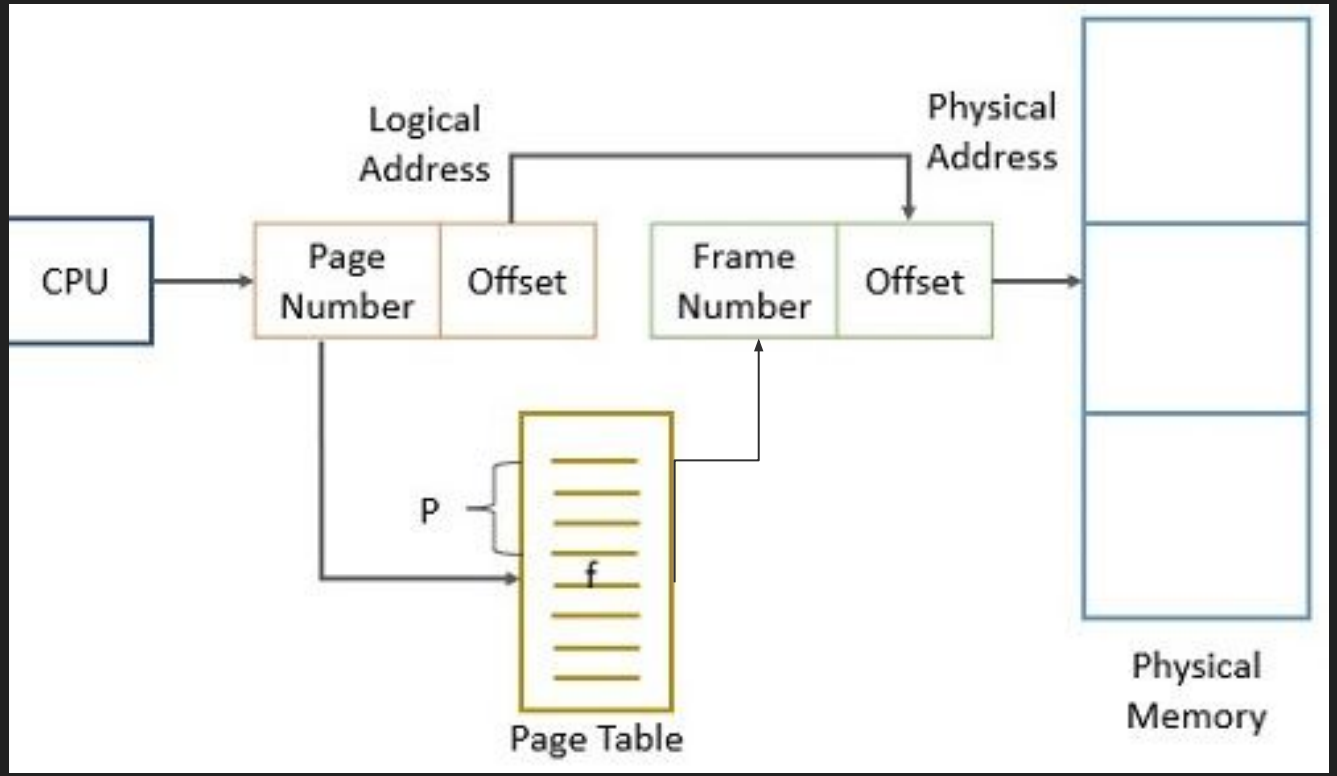
- The entirety of our process is divided up into fixed size pages
- These pages get mapped to frames in physical memory
- Pages can be moved, removed, added to frames in memory
- Our process should not care where its mapped to in physical memory
    - Our OS must do extra work keeping track of and maintaining processes page mapping
    - Our OS must translate logical addresses to physical addresses at runtime



Creating and Mapping Pages

- Processes refer to their memory locations with a virtual address
- Process generates contiguous virtual addresses from 0 to the size of the process
- The OS separates the process into pages and keeps track of their mapping to frames in the page table
- The paging hardware translates a virtual address (page number and page offset) into a physical address (frame number and frame offset)

Paging Hardware



Finding a Physical Address

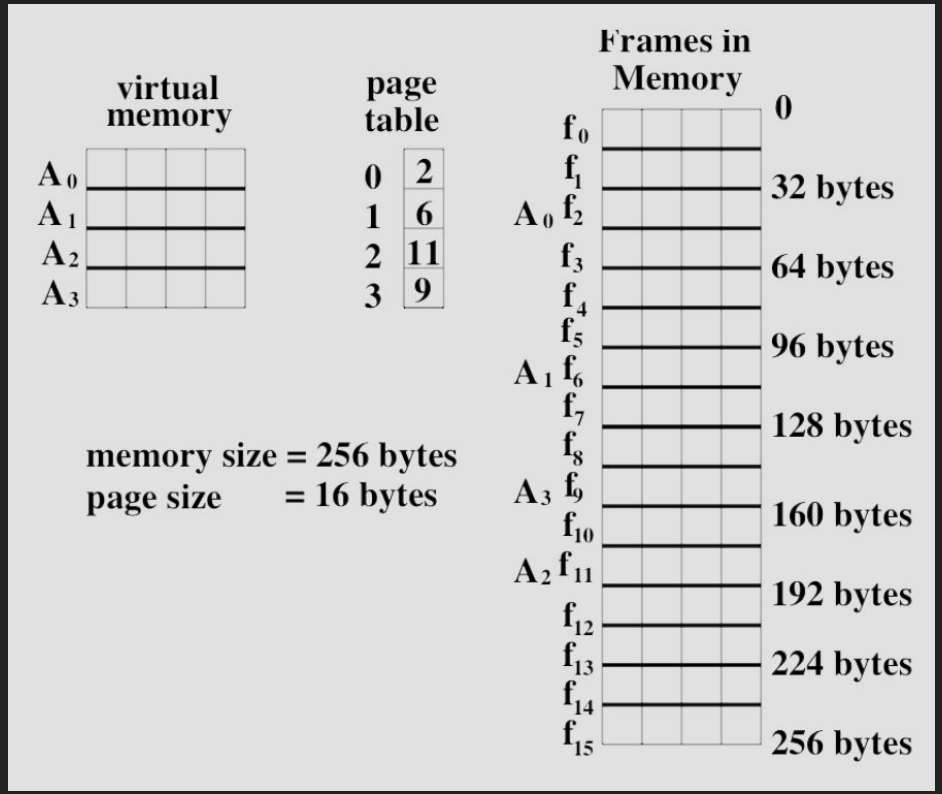
- Pages are typically 4096 bytes
- Assume 32 bit addresses with byte addressable memory
- Assume that the logical address requested by the CPU is:
  - x01A2C003
- Assume that there is an entry in the page table:
  - x01A2C : x20002
- What is the physical address that will be accessed?
  - x20002003

Paging Hardware

- Paging is a form of dynamic relocation, each virtual address is bound by the paging hardware to a physical address
- The page table acts as a set of relocation registers, one for each frame
- Mapping is invisible to the process, the OS does the mapping and the hardware does the translation
- Protections are provided with the same mechanisms used in dynamic relocation

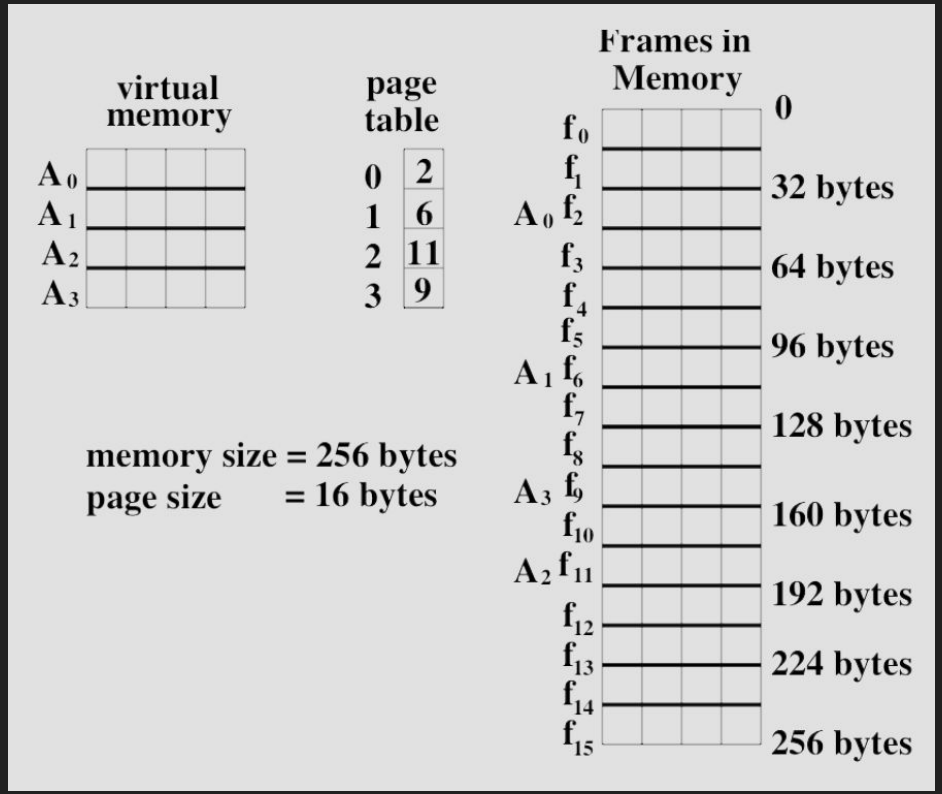
Address Translation Example 1

- Assume 256 bytes of total memory
- Assume byte addressable
- Assume page size of 16 bytes
- How big is the page table?
  - 16 entries = 256 total bytes / 16 bytes per page
- How many bits for a physical address?
  - 4 bits for frame (16 frames)
  - 4 bits for offset (16 bytes per frame)
  - 8 bits = 4 + 4 (256 addresses)
- How many bits for a virtual address?
  - 4 bits for 16 entries (page)
  - 4 bits for 16 bytes per page (offset)
  - 8 bits = 4 + 4
- Given a virtual address 0001 1000 in process A, what is the frame and offset of the physical address?
  - Page = 1, offset = 8
  - Frame = 6 (from page table), offset = 8
  - Physical address = 0110 1000



Address Translation Example 2

- Assume 256 bytes of total memory
- Assume word addressable with 4 byte word size
- Assume page size of 16 bytes
- How big is the page table?
  - 16 entries = 256 total bytes / 16 bytes per page
- How many bits for a physical address?
  - 4 bits for frame (16 frames)
  - 2 bits for offset (2 words per frame)
  - 6 bits = 4 + 2 (64 addresses) (64 = 256 / 4)
- How many bits for a virtual address?
  - 4 bits for 16 entries (page)
  - 2 bits for 4 words per page (offset)
  - 6 bits = 4 + 2
- Given a virtual address 0011 01 in process A, what is the frame and offset of the physical address?
  - Page = 3, offset = 1
  - Frame = 9 (from page table), offset = 1
  - Physical address = 1001 01



Where is The Page Table?

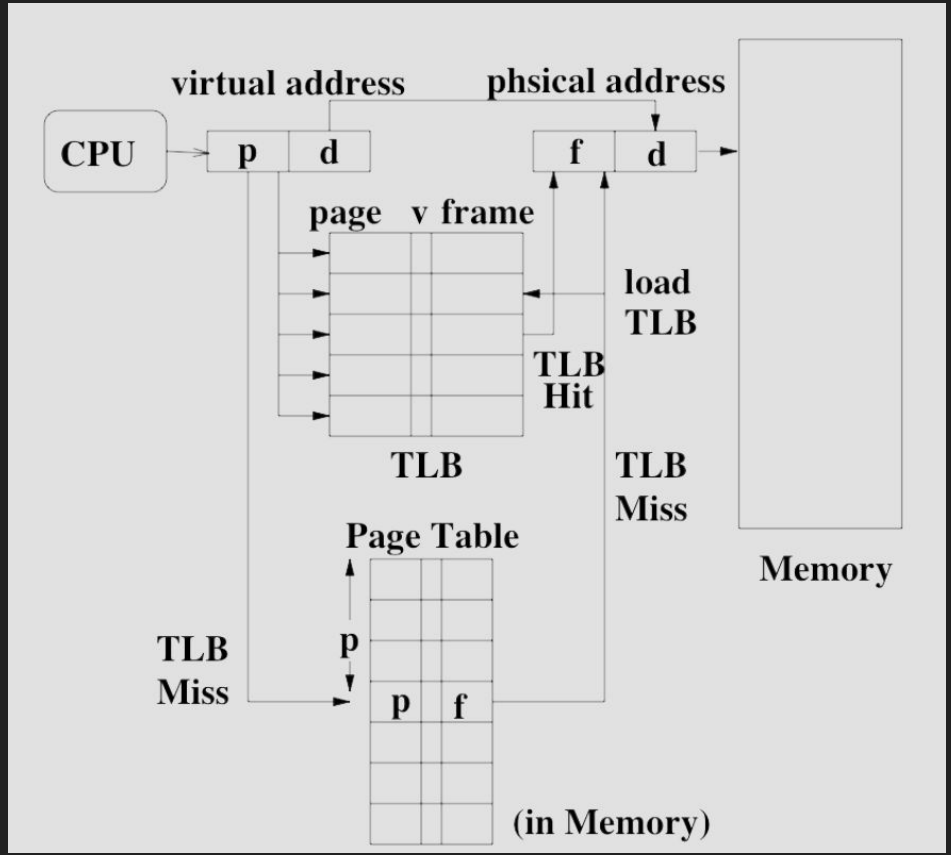
- Registers?
  - Very fast but we only have a few fixed number to work with
- Memory?
  - Slow but we probably have plenty of space
- Translation Look-Aside Buffer (TLB)
  - Works like a cache
  - Much larger than registers
  - Much faster than memory
  - Holds only a portion of our page table, the rest can exist in memory

Where is The Page Table?

- Registers?
  - Very fast but we only have a few fixed number to work with
- Memory?
  - Slow but we probably have plenty of space
- Translation Lookaside Buffer (TLB)
  - Works like a cache
  - Much larger than registers
  - Much faster than memory
  - Holds only a portion of our page table, the rest can exist in memory

TLB

- Have a small cache (TLB) that's going to store most of our page table
- TLB Hit:
  - If the page we requested exists in the TLB, translate the virtual address to a physical address
- TLB Miss:
  - If the page we requested does not exist in the TLB, go to memory to get it and put the missing page into the TLB
- Valid bit indicates if the TLB entry matches the page table entry
- A replacement algorithm must be implemented to replace entries in the TLB
  - Least recently used page can be replaced



Costs of Using the TLB

- How much time does it take to access memory if the page table is inside of memory?
  - T = total time, ma = memory access time
  - T = 2 * ma
  - We have to access the page table, then the physical memory, so 2 memory accesses are required
- How much additional time does the TLB add?
  - tlb = TLB access time, p = hit rate
  - T = (ma + tlb) * p + (2 * ma + tlb) * (1 - p)
- A larger TLB size decreases average memory access time

Initializing Memory With a New Process

1. Process arrives and needs k number of pages
2. If k frames are free, allocate the frames to pages, otherwise, free up frames that are no longer needed
3. OS puts each page inside each frame and adds the frame number to the page table
4. OS marks all TLB entries as invalid (flushes TLB)
5. OS starts executing process
6. OS loads TLB entries as each page is accessed, replacing existing entries if needed

The Process Control Block

- The PCB must be extended to contain
  - The entire process's page table
  - A copy of the TLB maybe
- During a context switch
  - Copy the page table base register value into the PCB for the process we are switching out
    - Page table base register (PTBR) points to the page table in memory for this process
  - Copy the TLB into the PCB (optional)
  - Flush the TLB
  - Restore the page table base register for process we're switching to
  - Restore the TLB if it was saved in the new process's PCB

Sharing Code Between Processes

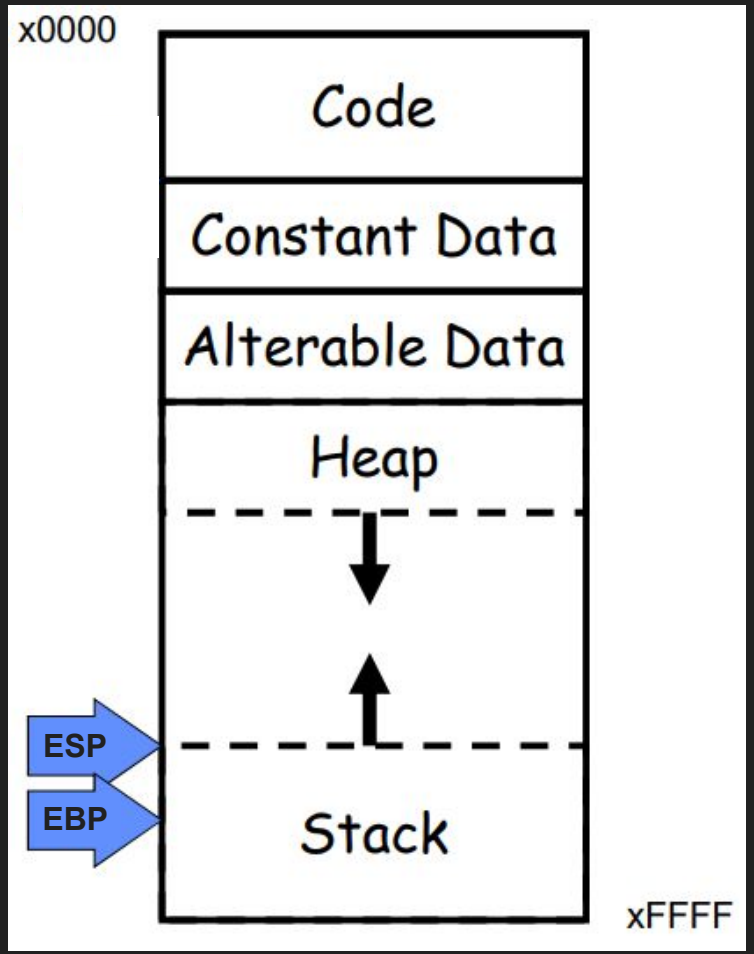
- Since a process's physical memory does not need to be contiguous, multiple processes could be allowed to access the same frame
- Any shared code must be reentrant, meaning processes should not be allowed to change it
  - Shared libraries (like standard C libraries) can easily be shared amongst processes and only one instance of the library is needed to be in memory

Paging Summary

- Advantages to paging:
  - Eliminates external fragmentation and compaction
  - Allows for sharing of code amongst processes
  - Processes can be partially loaded into memory
- Disadvantages to paging:
  - Translating a virtual address to a physical address takes more time
  - Requires hardware support with TLB
  - Requires more complex OS to maintain page table

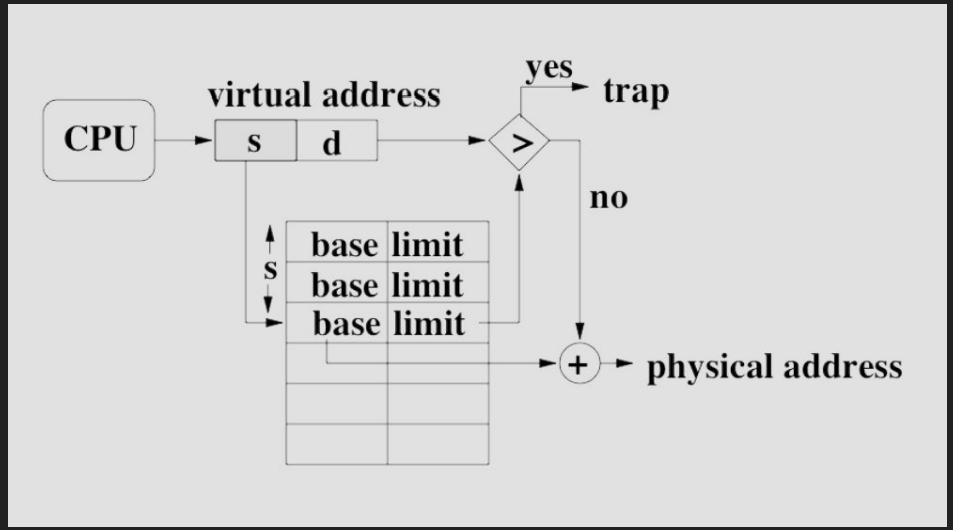
Segmentation

- Segmentation is another form of memory management
  - OSs may use segmentation instead of paging or they may use a combination of both
  - Windows and Linux use a combination of paging and segmentation
- Processes are made of different segments:
  - Code (instructions), variables (data), stack, heap
- Segmentation refers to these segments directly, instead of treating a process's memory like a linear array of bytes
  - The virtual address is a combination of a base register that identifies the segment and an offset inside that segment



Segmentation Hardware

- A segment table contains the base and limit registers of the segment
  - Additional information may be included to specify if a segment can be shared, read, written, etc.
- Typically, each process only has a few segments
  - x86 architecture only allows for 6 segments with the following registers: CS, SS, DS, ES, FS, GS
  - If a system uses many segments a TLB-like system may need to be implemented
- Segmentation does not eliminate external fragmentation
  - Segments must be contiguous and are of arbitrary size



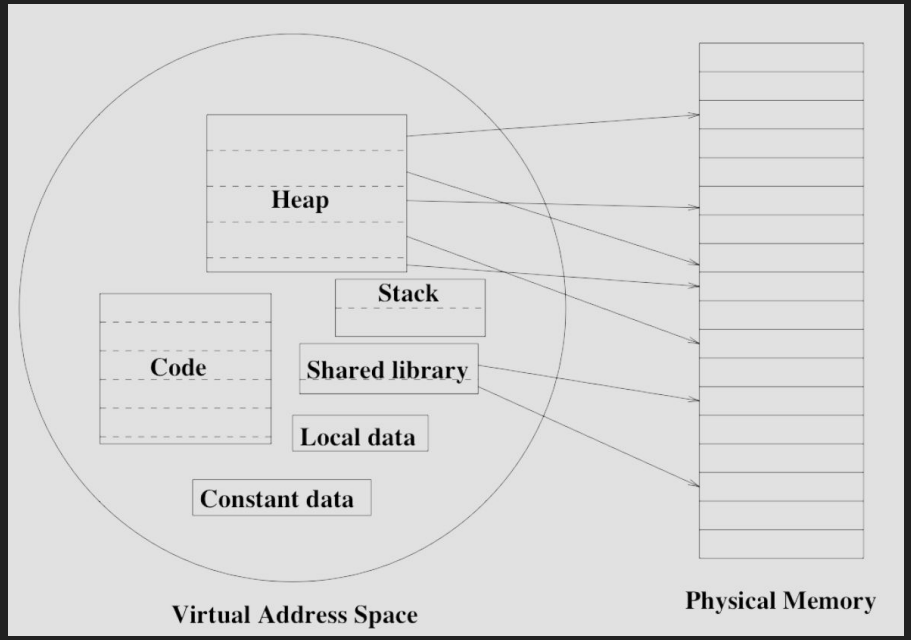
CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment
FS	General Purpose Segments
GS	

Segmentation Implementation

- At compile time, virtual addresses are generated where the most significant bit(s) are the segment number and the least significant bits are the offset into that segment
- Segmentation could be combined with dynamic or static relocation
  - Each segment is contiguous and could be assigned to any region of physical memory

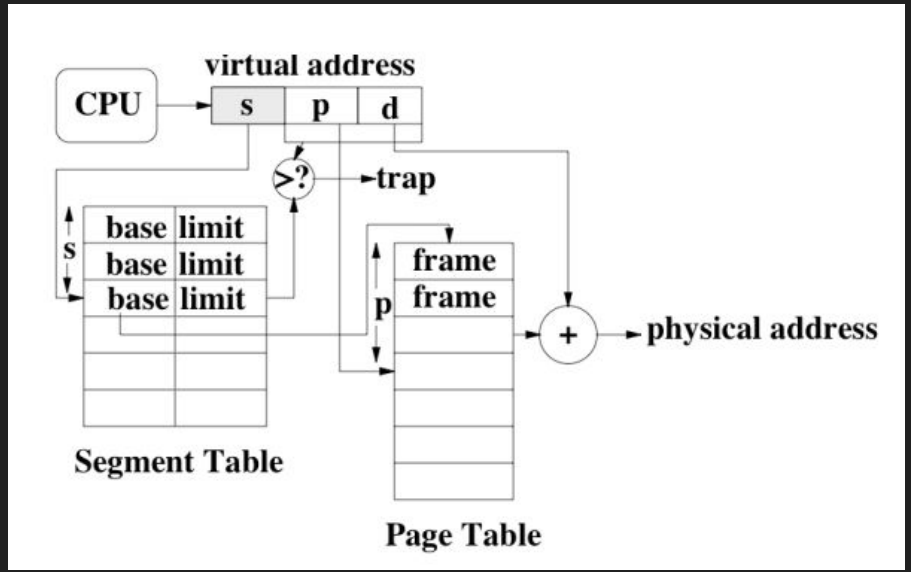
Segments and Pages

- Treat virtual address space as a collection of segments
- Treat physical memory as a sequence of fixed size frames
- Split the virtual segments into pages and map onto multiple frames



Translating Addresses in Segmented Paging

- The CPU uses a virtual address that contains a segment number, page number, and offset
- Both a segment table and page table are required
- Segment table can exist inside of registers or memory
    - Limits the maximum number of segments
- Page table can exist inside of memory and utilize a TLB
- Each base segment register points to a different page table that corresponds to the given segment



Segmented Paging Example 1

- Assume 256 bytes of total memory
- Assume byte addressable memory
- Assume page size of 32 bytes
- Assume maximum segments of 8
- How big is the page table?
  - 8 entries = 256 total bytes / 32 bytes per page
- How many bits for a physical address?
  - 3 bits for frame (8 frames)
  - 5 bits for offset (32 bytes per frame)
  - 8 bits (256 bytes)
- How many bits for a virtual address?
  - 3 bits (8 segments)
  - 3 bits (8 entries) (page)
  - 5 bits (32 bytes per page) (offset)
  - 11 bits = 3 + 3 + 5
- Given a virtual address 011 001 10101 in process A, what is the segment, frame, and offset of the physical address? Assume the segment points to the page table listed in the diagram.
  - Segment = 3
  - Page = 1, offset = 21
  - Frame = 2 (from page table), offset = 21
  - Physical address = 010 10101