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

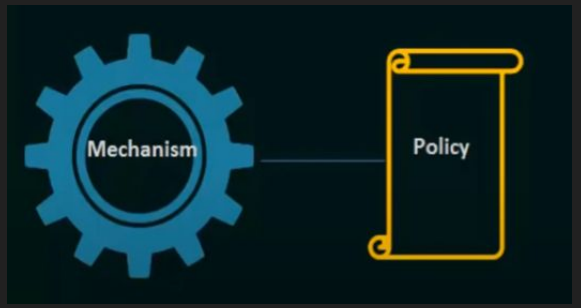
The Separation of Mechanisms and Policies

- 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

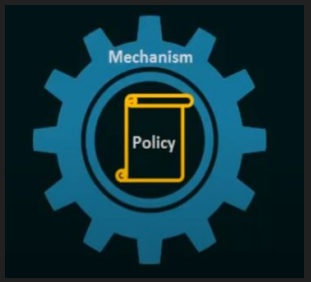
The Separation of Mechanisms and Policies (cont.)

- 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

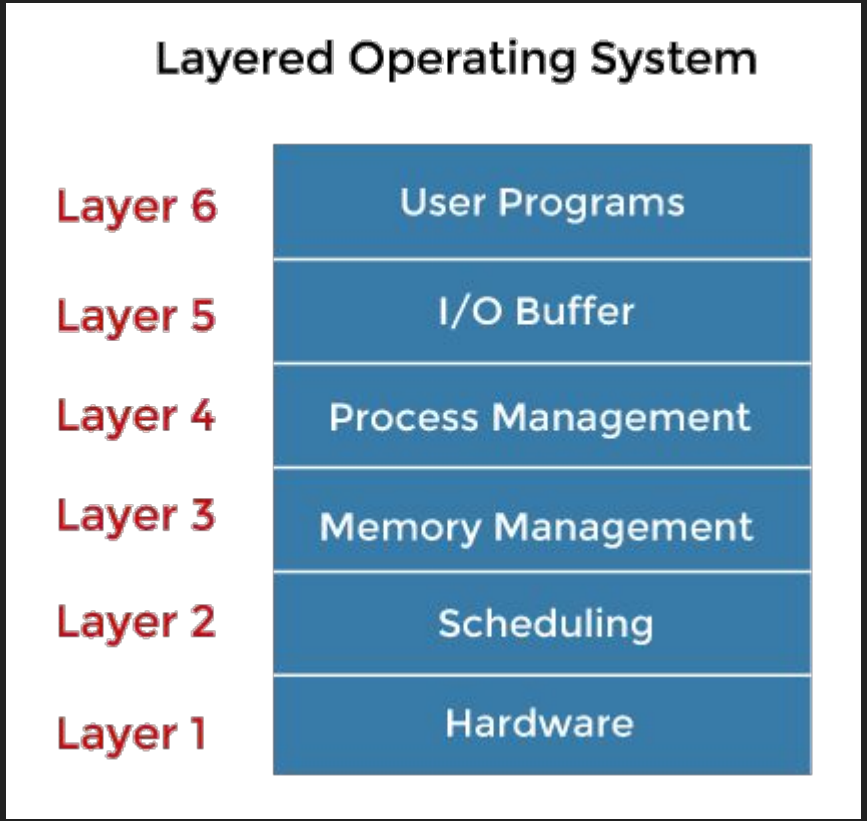
- The OS is a resource manager
  - Allocates time for programs to run on the processor
  - Allocates memory for the programs to use
  - Allocates usage of devices for programs to use, hard drive, printer, keyboard, etc.
- Do we need ensure tasks meet timing deadlines?
  - RTOS
- Do we need to ensure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

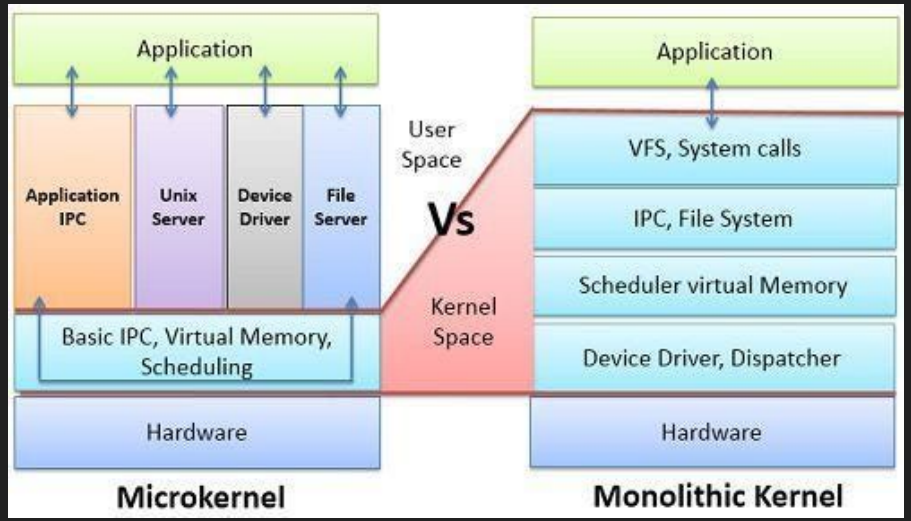
Structuring an OS

- 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.)

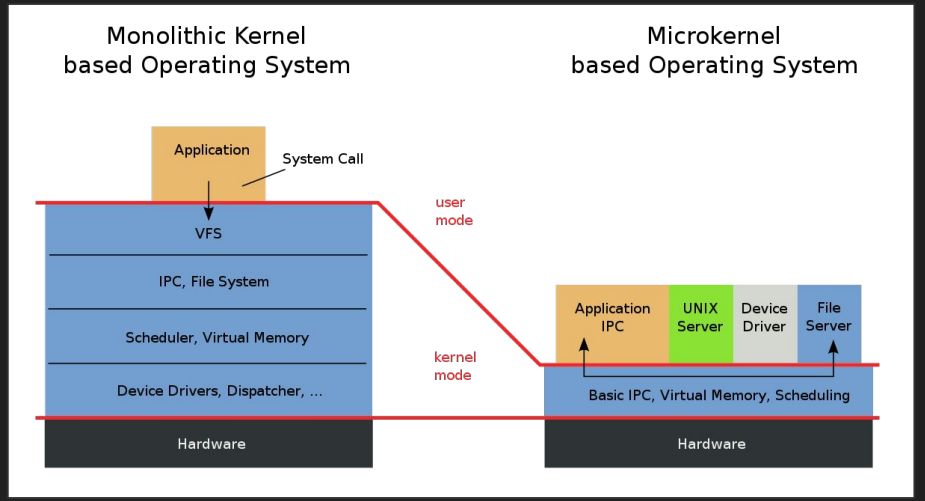
- Monolithic OS Structure
  - Entire OS is working in kernel space
  - Can be hard to write or update
  - Modules stacked on top of modules on top of modules



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

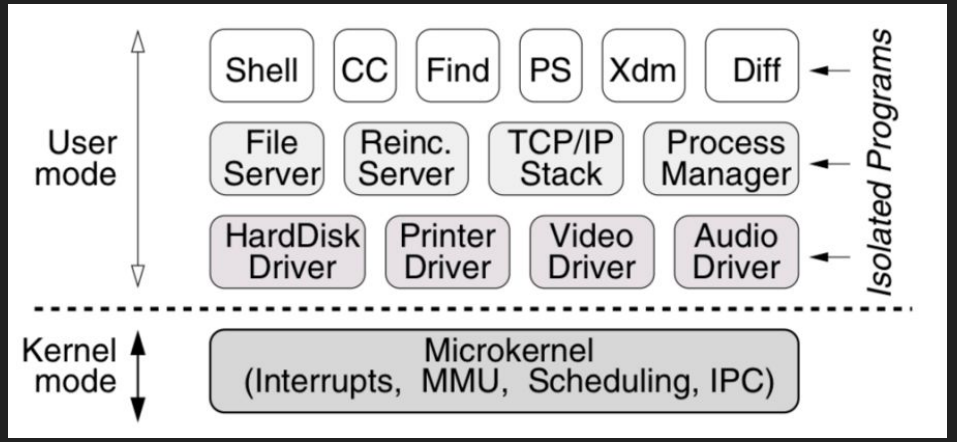
Structuring an OS (cont.)

- 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

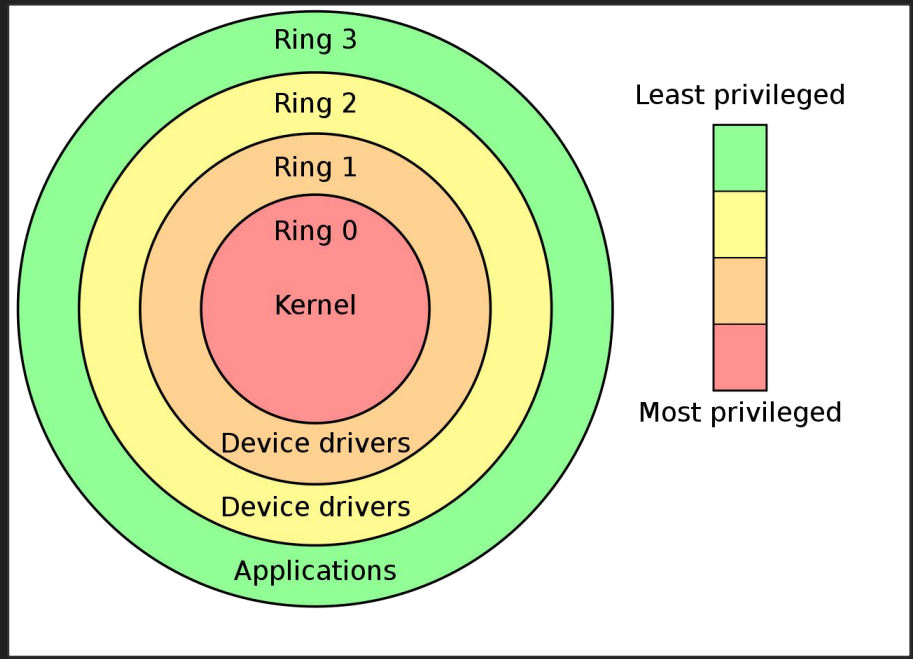


Protections

- 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

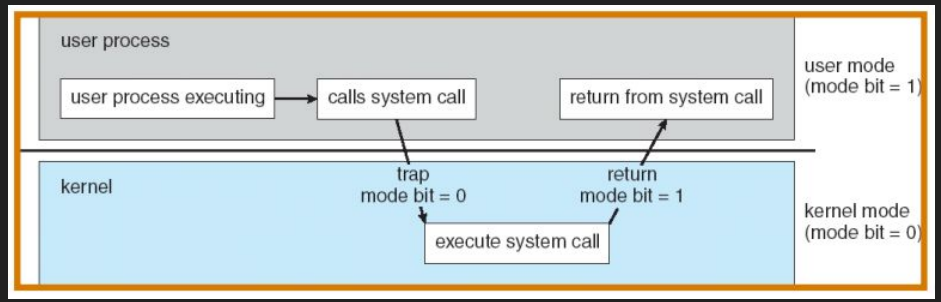
Kernel Mode vs User Mode

- 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



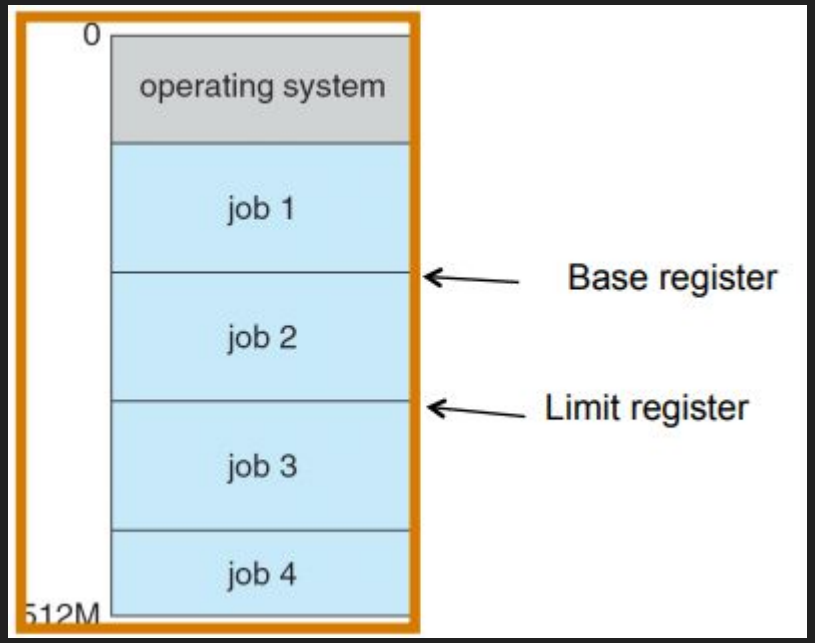
Kernel Mode vs User Mode (cont.)

- 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