Operating Systems Overview

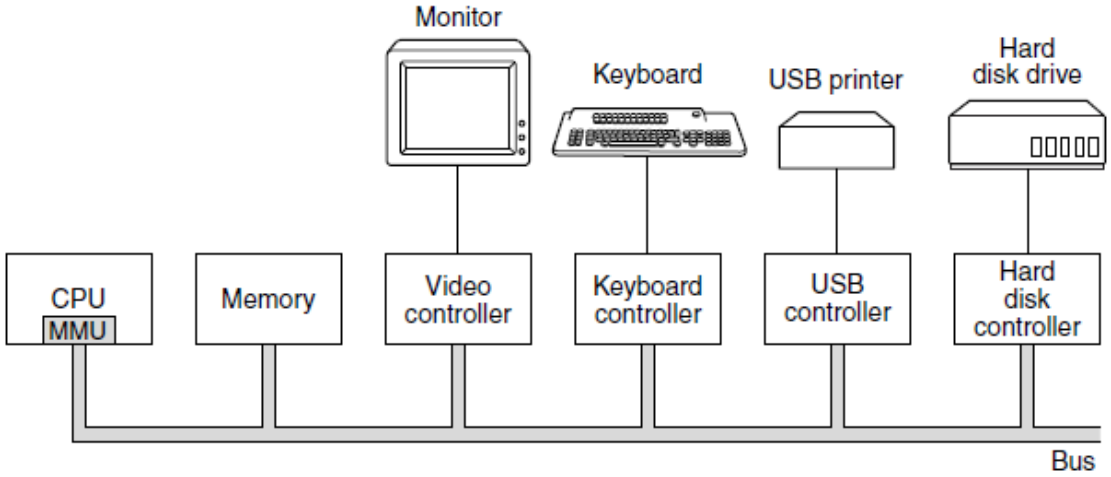
Module 02

Textbook

This module covers Chapters 1.1-1.5 in the textbook.

Anatomy of a (Personal) Computer

An operating system manages devices - often called resources. What sort of devices?



Categories of Devices (my terminology)

Core Devices: CPU, Memory, Storage (Disc)

- We'll spend the bulk of the semester here
- About 50-60% of the semester is just CPU and memory!

Peripherals: *Everything else*

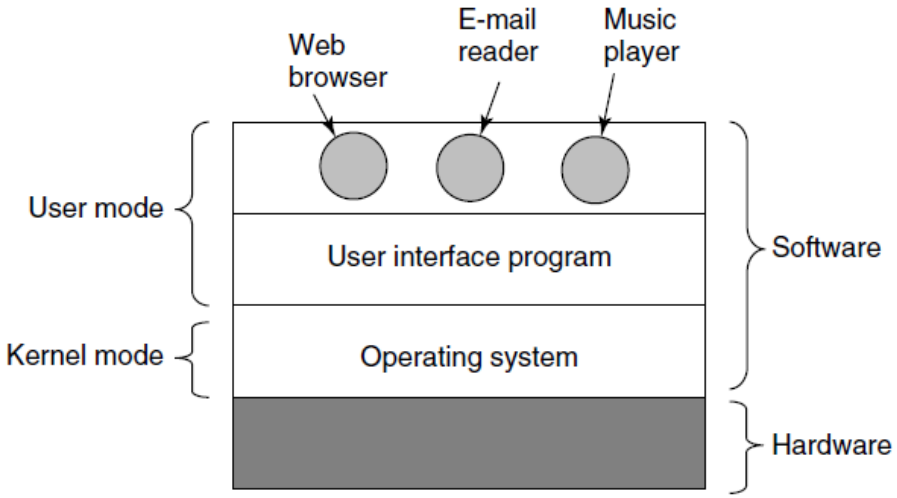
Where is the OS?

An operating system ***is a program***.

- Its written in a normal language (C and C++ with a little Assembly).
- It is stored on disc
- It is loaded into memory
- It’s instructions are executed by the CPU

Its not really all that special!

Where is the OS?



Conceptually, the OS is different

It sits between “User programs” and the hardware

Types of Programs

The operating system can be thought of as a ***collection*** of programs - commonly called the kernel

All other programs are called user programs

Some are used by real people, others are behind the scenes - but none have direct access to the hardware.

What's a user?

When we say user, we don't mean a person…

We mean a **program** that is not part of the kernel.

- This distinction is **critical**
- User programs might have a real user interface - but that's besides the point!

Why do we call a program a user?

- The operating system provides access to devices through function calls
  - An **Application Programming Interface (API)**
  - Programs **are** the “customer” - they call the OS’s API to do things like:
    - print to the screen
    - read data from disc
    - draw to the frame buffer

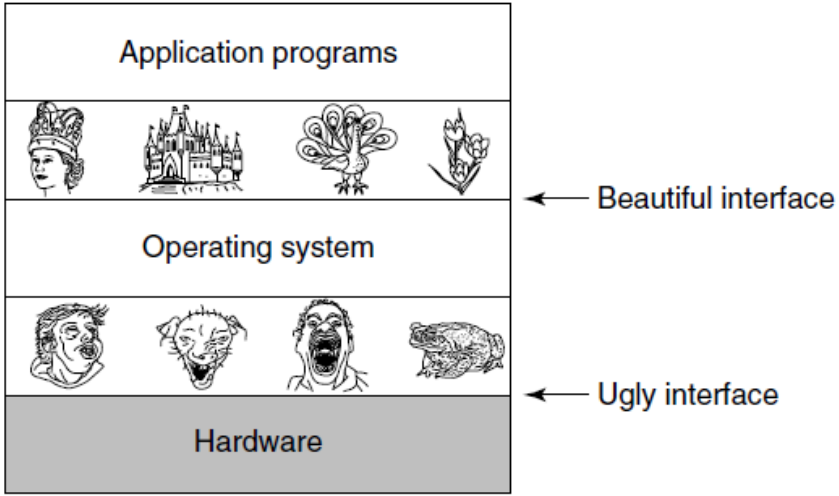
Why not?

Why don’t user programs have access to devices?

You probably don’t want access...

Sure, the OS needs to ensure everyone is sharing the resources…

However in addition, its the OS that shields programs from the **horrors of all the device API’s!**



Summarizing the OS's role

Bottom-up view: Ensures orderly, safe, and fair use and allocation of resources

Top-down view: Provides abstractions to simplify access to a diverse set of devices

The most important resource: CPU

Lets diagram a CPU - what are the important components?

- ALU
- Registers
- Control Unit
- Special Registers - PC, PSW (Program Status Word), etc.

This is overly simplistic - but is sufficient

The most important resource: CPU

Programs are stored in memory - linear arrays of *instructions*

- Instructions are **fetched** by examining the PC
  - Instruction decoded
  - For mathematical operations
    - Operands flow into ALU
    - Results flow out of ALU, stored in registers
  - For load/store operations
    - Memory addresses are read into registers
    - Or registers values are saved to memory
  - And then… there's I/O instructions

Instruction Fetch is endless… until I/O

- Mathematical, Jump, and Load/Store operations are done completely by the CPU
  - CPU just keeps... going…
    - It runs a single program!

I/O and the CPU

- I/O instructions require another device:
  - Write to frame buffer (screen)
  - Read from Disk
  - Wait for Key-press
- This is very bad…
  - If there are other programs that *could* run, it would be nice to let the CPU keep running - **those programs**.

Multiprogramming

- On an I/O call, the current state of the CPU is copied into memory
  - Registers, PC, etc.
- Another program is **selected** and its next instruction is **loaded** into the PC.
- And the CPU goes on its way...

The CPU will get notified when the I/O device is done, we'll deal with this in a few minutes...

Who does all this?

The Operating System!

- The OS runs when there is an I/O call, or when a program terminates (we'll see how shortly)
- The OS picks a new program, and loads the PC with the appropriate memory address for the instruction
- This is called a **context switch**.
- This portion of the OS is called the ***Scheduler***

This is called Multiprogramming

I/O Devices and Interrupts

Once an I/O device is started (read data) - it must have some way of **interrupting** the CPU - which will be furiously running the computations of another program

I/O Device when it's done

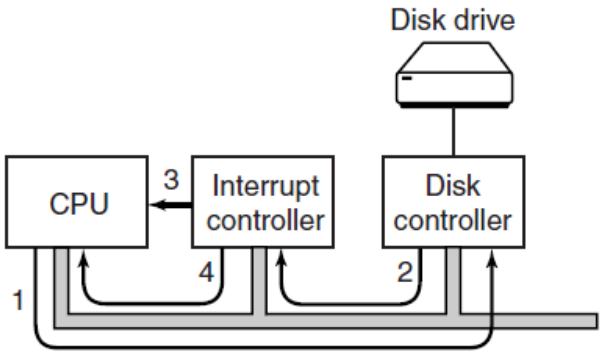
CPU

Interrupt

The CPU is simply a circuit

There are “wires” fed into it that trigger logic

One wire, if “**signaled**” will break the normal instruction fetch cycle



Interrupt Handlers

The CPU is configured to automatically set the PC to a predefined location in memory

- This location is called the interrupt vector
- It contains code that will
  - Examine the signal (stored in a register now) to determine which type of interrupt was fired
  - Handle/Process the interrupt
    - The interrupt vector is part of the OS

Timers as Interrupts

Let's say the CPU is running a user program:

```
while (true) {
    // I’m doing stuff...
}
```

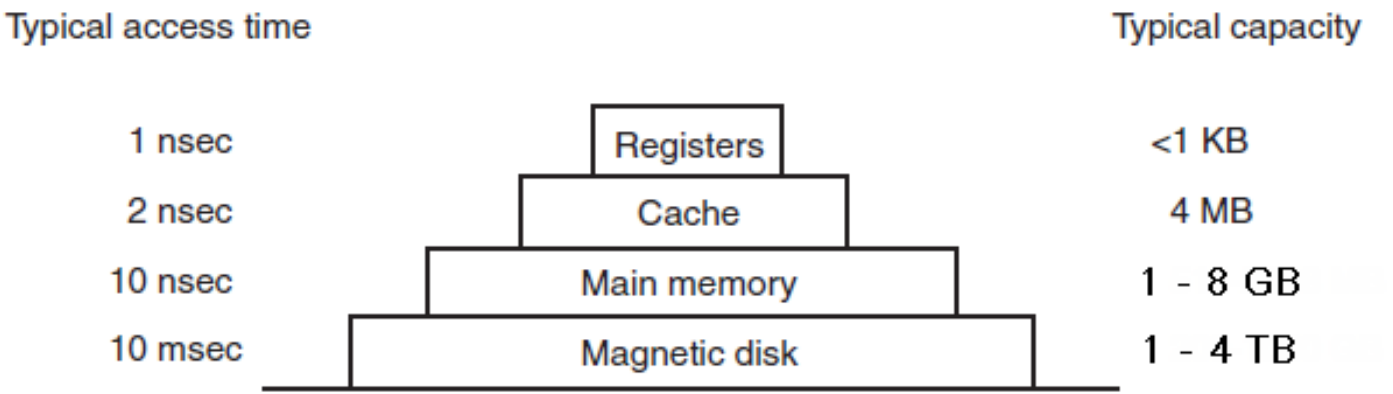
Unless the OS has a way of ***interrupting*** the user program - the OS might never be able to run again!

The Memory System

Memory ***should*** be as fast as the registers on the CPU, large enough to hold all of our programs, and cheap

No memory device is all three

A bit about caching



Part of the OS's responsibility is to work with the hardware to make this seamless to the user program.

The Memory System - Protection

Multiprogramming tells us *multiple* programs are in memory:

- Another job of the OS is to make sure programs can't access each other's memory addresses
- This is harder than it seems, since they all think they "own" address 100 (for example)
- Its called Memory Management and Virtual Memory - and we'll cover it in **October**

Fundamental OS Abstractions

Processes - a running program (plus its state). We'll be covering this first.

Address Space - the list of memory addresses available for variables, code, and the like. We'll be covering this in more detail in October

Files and Devices - common interface for user programs to read/write data. November & December

Next time

Read Chapters 1.5-1.6 on System Calls and OS Organization