Last time

Things you should remember

History of operating systems

- serial processing
- batch processing
- multitasking batch processing
- time-sharing
- OS family tree

Serial Processing:
Executing tasks one after another in a sequential manner, where each task completes before the next one begins.

Batch Processing:
Processing a set of tasks or jobs simultaneously without manual intervention, often in a non-interactive, offline mode.

Multitasking:
Simultaneously executing multiple tasks or processes on a computer, allowing the appearance of concurrent operation.

Batch Processing, Time-Sharing:
Combining batch processing (non-interactive, offline) with time-sharing (interactive, online) to efficiently utilize computer resources for both scheduled and interactive tasks.

Module 1:

Processes, privilege and system calls

Process

A *running program*

What is a program?

How do programs run?

- OS allocates memory for the program (for what?)
- OS loads program code into memory
- Execution begins at `main()`... right?

Process:
A process is an instance of a computer program that is being executed. It includes the program code, associated data, and the execution context (such as registers, program counter, and environment variables).

Program:
A program is a set of instructions or code written in a programming language that performs a specific task when executed by a computer.

How Programs Run:
Execution Steps:
OS allocates memory: The operating system allocates memory space for the program to store both code and data during execution.

OS loads program code into memory: The operating system loads the program's executable code into the allocated memory space.

Execution begins at main(): The program's execution typically begins at the main() function (or an equivalent starting point), where the operating system transfers control to the program.

Processes

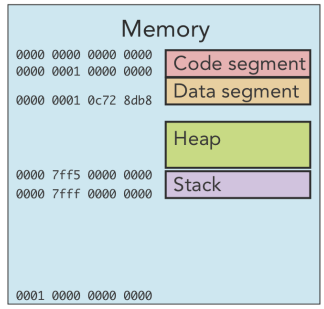
Reality is, of course, **slightly more complicated**!

Memory allocation

What do programs need memory for?

Process execution

Also, programs don't actually start running in `main`! See: crt1_c.c.



Memory allocation

As we saw way back in our first OO programming course, running programs need memory for code, global variables (`.bss`), the stack and the heap. Referring back to our "hello, world" example), identify how our program uses memory for code, global variables, the stack *and the heap* (yes, it's there!).

Processes

Processes have memory

What sort of addresses are your program's pointers?

Processes execute

OS must *schedule* processes: "what shall we run next?"

Processes have (indirect) access to resources (e.g., files)

Processes are *unprivileged*

The addresses used directly by your software are _____. This is why, when the same code is executed in ten processes in parallel, they can all refer to the _____ _____, but *without interference*.

“The addresses used directly by your software are virtual addresses. This is why, when the same code is executed in ten processes in parallel, they can all refer to the same virtual address space, but without interference.” This is because each process has its own virtual address space, and the operating system ensures that these spaces do not overlap, preventing interference between processes. The operating system and the hardware work together to map these virtual addresses to physical addresses in memory. This mapping is unique per process, ensuring isolation and preventing interference.

Privilege

Processors many have many modes, but we're mostly interested in two categories:

- Supervisor mode: access to privileged instructions
- User mode: no privileged instructions

The mode is indicated by bits in a *status register*, e.g., x86 `FLAGS`/`EFLAGS`/`RFLAGS` or Arm CPSR.

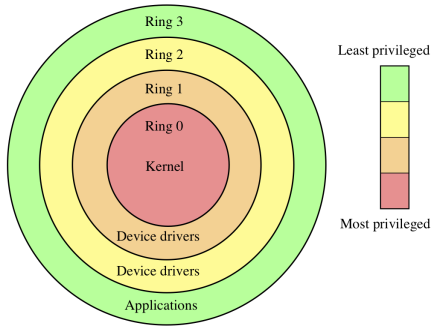
The processor mode changes on interrupt handling, system calls and other traps.

If we attempt to execute a privileged instruction from user mode, it will result in a kind of interrupt called a *trap*, which the OS will need to deal with.

Rings

- GE 645: software emulation
- Honeywell 6180 had 8 (!)
- x86 has four rings
- Use of intermediates is... inconsistent

x86 IOPL rings



"Priv rings" by Hertzsprung at English Wikipedia. CC BY-SA 3.0.

Although the x86 architecture has four privilege rings, in practice we only use two: ring 0 (kernel) and ring 3 (application). The intermediate rings were intended for use with partially-privileged device drivers, but providing code with access to some kinds of privileged instructions and not others is often a distinction without a difference: if you can interact with I/O, you can generally use that to modify physical memory. If you can modify physical memory, you can see (or modify) any other code on the system, including device drivers. Thus, it doesn't make a lot of coherent sense to give "some" privilege to code via Ring 1 or Ring 2.

Arm privilege

Processor modes:

Mode	Priv	Mode	Priv
User	PL0	Abort	PL1
FIQ	PL1	Hyp	PL2
IRQ	PL1	Undefined	PL1
Supervisor	PL1	System	PL1
Monitor	PL1		

Privilege levels

PL0: User

PL1: System

PL2: Hyp

Processor mode exposed via the CPSR register

Question

What would happen if a process could access *physical memory*?

Exercise

Identify the parts of hello-world.cpp that require memory regions for code, global variables, the stack *and the heap*.

If a process could directly access physical memory, it would bypass the protection and isolation mechanisms provided by virtual memory1. Here are some potential consequences:

Unprivileged processes

So how can an unprivileged process perform privileged operations?

Answer: *system calls*

When a user-level process wants to perform a task that requires operating system intervention or access to privileged resources, it makes the appropriate system call. The operating system then switches to kernel mode to execute the requested operation on behalf of the process. After completion, control is returned to the user-level process.

Examples of System Prompts:

File Operations:
open(), read(), write(), close()

Process Control:
fork(), exec(), exit(), wait()

Memory Management:
brk(), mmap()

Communication:
pipe(), socket()

Device Control:
ioctl()

System calls

- privileged system (kernel) code called by unprivileged (user) process
- not called like ordinary functions:
  1. Set arguments (including system call #) in registers or memory
  2. Invoke interrupt or syscall instruction

Subject of Lab 1!

Kernel is the basis for interactions between hardware and software and manages their resources as efficiently as possible1.