Operating System (OS): Resource Manager

- The OS is responsible for resource management & allocation
- Resources like CPU time or memory space are limited
- The OS must decide how to allocate & keep track of the system resources
- In the event of conflicting requests, choose the winner
- The OS is responsible for abstracting away hardware details

OS: Multitasking

- Multiple programs means that resources are shared
  ↳ source of conflicts!
- OS creates + enforces the rules so all programs get along

Kernel: the "core" of the OS ⇒ the portion of the OS that is always present in main memory and the central part that makes it all work.

The Three Desirable Properties: Confidentiality, Integrity, Availability

Protection: internal threats

- Enforce policies about responsible usage
- Examples (of access controls): file permissions, walls between processes
- Rules take effort to enforce
  - Exceptions are allowed and administrators can override policies!

Security: external threats

- Possible security problems: breach of confidenticality, breach of integrity, breach of availability, theft of service, etc...

⇒ most likely the weak link is the people

Specific Attacks:

- Excessive Requests
- Back Door
- Trojan Horse
- Malformed Requests
- Intercepting Messages
- Etc...

To execute a program (minimally), we need:

1. Main Memory
2. System Bus
3. Processor

1. Main Memory

- Ideally, main memory would be:
  - fast enough that the processor never has to wait,
  - large enough to hold all the data,
  - inexpensive

⇒ the iron triangle: "fast, good, cheap; pick two"

- Good news: we can have different levels of memory
  - Example of possible memory levels:

Memory Level	Access Time	Total Capacity
Register	1 ns	< 1 KB
Cache	2 ns	16 MB
Main Memory (RAM)	10 ns	64 GB
Solid State Hard Disk	250 us	1000 GB
Backup Hard Disk Drive	10 ms	2 TB

- Memory Access Analogy: I'm the CPU and I want a book. If data is in cache, the book is in my office on a shelf. If it's on a hard disk, I'd have to walk 550 km to Ottawa's main library to get it.

⇒ CPU doesn't get data itself → it waits for it to arrive. So, what does it do while waiting? coming up...

2. System Buses

- Every sort of communication using the same bus (old systems) ⇒ contention for this resource is a limiting factor
- Modern systems have numerous buses

3. Central Processing Unit (CPU)

- The "brain" of the computer
- Fetch - Decode - Execute cycle repeated until program ends
- Different steps may be completed in parallel (pipeline)
- Processor's largest unit is the word
  - 32-bit computer ⇒ 32-bit word
  - 64-bit computer ⇒ 64-bit word
- CPUs have storage locations: registers

Registers

- Store data or instructions
- Management of registers is partly the role of the OS
- A few registers in a typical CPU: program counter, status register, instruction register, stack pointer, general purpose registers, etc

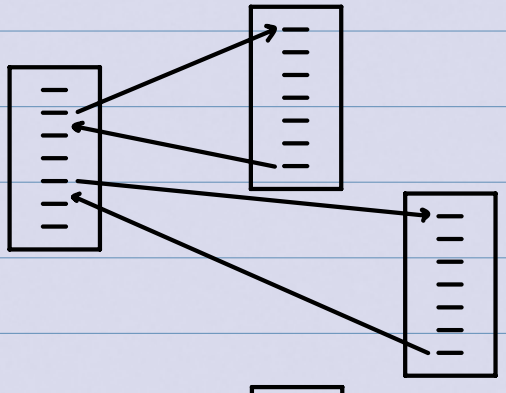
Program Execution: a program is a sequence of instructions. We can categorise them as:

1. Processor - Memory
2. Processor - I/O
3. Data Processing
4. Control

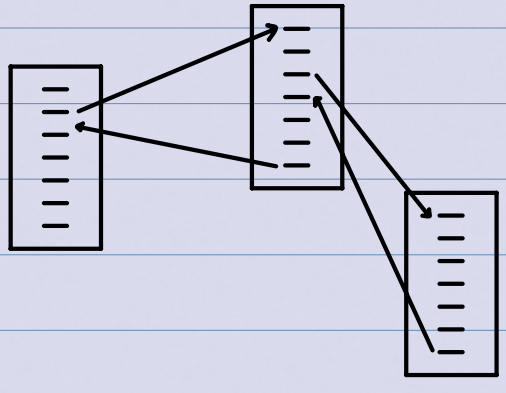
Interrupts

- If I order a book from Ottawa, it takes a while to arrive
- Polling: check periodically if the book has arrived
- Interrupts: get a notif when the book arrives
  ⇒ if someone knocks (ie, notified), I pause what I'm doing to go sign for the book
- We can put interrupts into **4 categories**, based on origin:
  1. Program
  2. Timer
  3. Input / Output
  4. Hardware Failure
- Interrupts are a way to **improve processor utilization**
  ↳ more commonly, we have to handle it somehow
- The OS: **stores the state, handle the interrupt, restore state**
- Sometimes, the CPU is in the middle of something uninterruptable, so interrupts may be (temporarily) disabled
- Interrupts can have **different priorities**

Sequential Interrupts:



Nested Interrupts:



Storing and Restoring the State

- The state must be stored
- What is state? Values of registers
- Where is it stored? Pushed onto the stack
- When the interrupt is finished, we restore the state by popping off of the stack and continue the execution

There are 3 major strategies for communication:
1. Programmed I/O 2. Interrupt Driven I/O 3. Direct Memory Access (DMA)

Direct Memory Address

- CPU does setup - data sent to DMA module
- Delegation requires setting up:
  1. the operation to perform (read or write)
  2. the source
  3. the destination
  4. how much data is to be transferred
- The I/O device will interact directly with memory

Traps: Software-generated interrupts

- Generated by an error (invalid instruction) or user program request
- If it's an error: the OS will decide what to do
  ⇒ usual strategy: give the error to the program
- The program can decide what to do if it can handle it
- If it can't handle it, the program just dies

User Mode vs Kernel (Supervisor) Mode

- Supervisor mode allows all instructions and operations

⇒ even simple things like reading from disk or writing to console require priveledged instructions

- Modern processors keep track of what mode they're in with the **mode bit**
- At boot up, the computer **starts in kernel mode** while the OS is started and loaded
- **User programs are always started in user mode**
- When a **trap or interrupt occurs** and the OS takes over, the **mode bit is set to kernel mode**

⇒ When finished, system goes **back to user mode** before the user program continues

- **Switching** from user to kernel mode **takes time**

⇒ The performance hit is worth it for the **security**

Example: Reading from Disk

```
SSize_t read(int file_descriptor, void *buffer, size_t count);
```

(file_descriptor: the file; buffer: where to read the data to; count: how many bytes to read)

- In preperation for a call to read, the parameters are pushed onto the stack
- read is called; the normal instruction to enter another function
- the read function will put it's identifier in a predefined location
- then it executes the trap instruction, activating the OS
- the OS takes over and control switches to kernel mode
- control transfers to a predefined memory location within the kernel
- the trap handler examines the request: it checks the identifier
- now it knows what system call request handler should execute: read
- that routine executes
- when it's finished, control will be returned to the read function

⇒ exit the kernel and return to user mode

- read finishes and returns, and control goes back to the user program

System Call Summary

1. The user program pushes arguments onto the stack
2. The user program invokes the system call
3. The system call puts its identifier in the designated location
4. The system call issues the trap instruction
5. The OS responds to the interrupt and examines the identifier in the designated location
6. The OS runs the system call handler that matches the identifier
7. When the handler is finished, control exits the kernel and goes back to the system call (in user mode)
8. The system call returns control to the user program