01. Introduction

Ch. 1, 2

Course Structure

Part I	Structures [RMM]
01	Introduction
02	Protection
Part II	CPU [EK]
03	Processes
04	Scheduling
05	Scheduling Algorithms
Part III	Memory [RMM]
06	Memory Management

Part III	Memory (continued) [RMM]
07	Paging
08	Virtual Memory
Part IV	Input/Output and Storage [EK]
09	I/O Subsystem
10	File Management
Part V	Case Study [RMM]
11	Unix 1
12	Unix 2

Objectives

- To describe the basic organisation of computer systems
- To give an abstract view of the operating system
- To introduce some key concepts in (operating) systems
- To give a brief tour of the major functions of the operating system

- Recall Part 2 of Introduction to Microprocessors in IA Digital Electronics
  - Fetch-Decode-Execute cycle, Pipelining

Outline

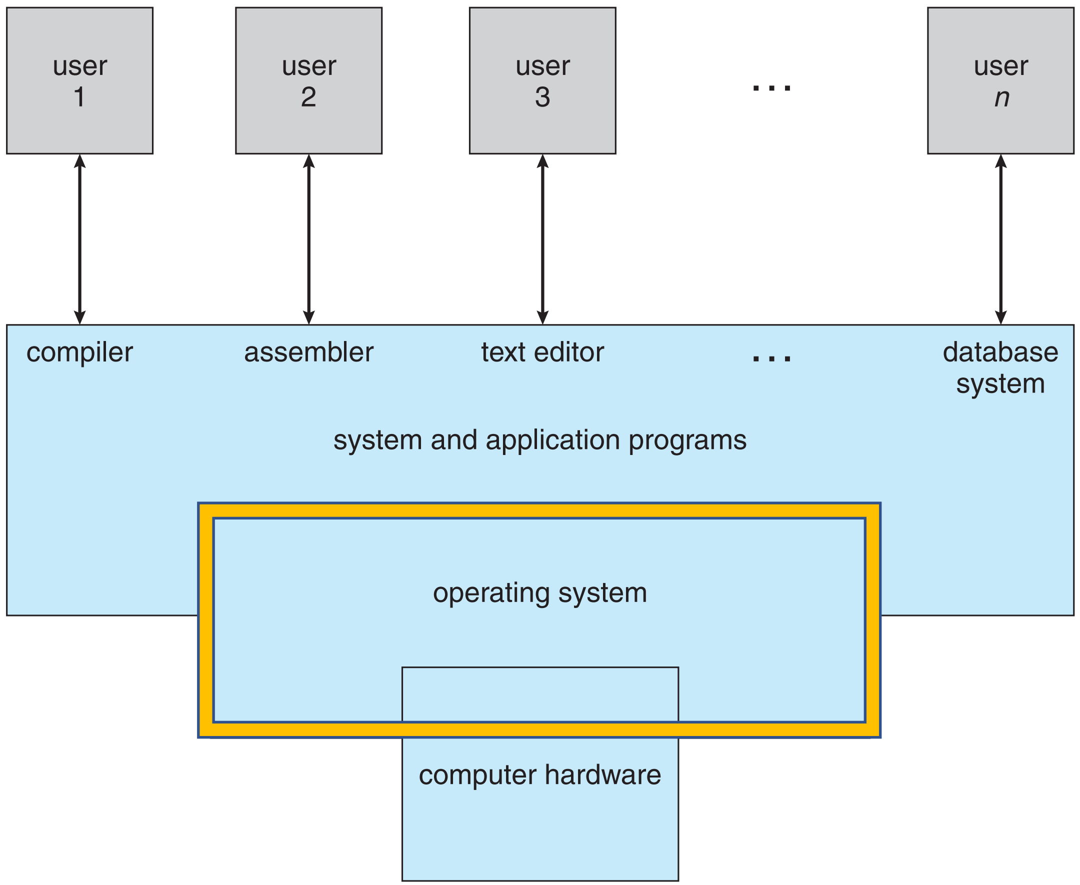
- System organisation
- System operation
- Concepts
- What is an Operating System?

Outline

- System organisation
    - Hardware resources
    - Fetch-execute cycle
    - Buses
- System operation
- Concepts
- What is an Operating System?

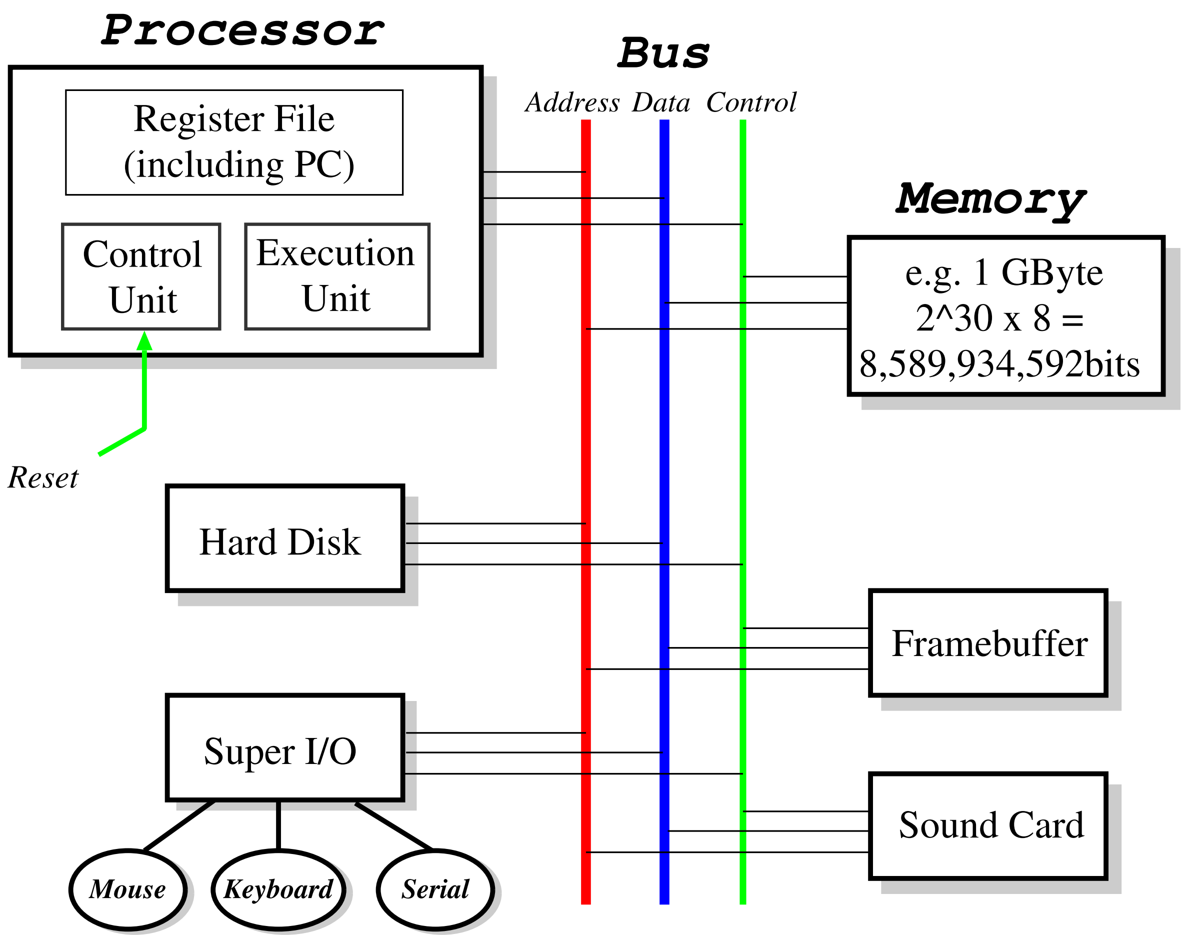
Computer system organisation

1. **Hardware** provides basic computing resources: CPU, memory, I/O devices
2. **Operating system** controls and coordinates use of those resources
3. **Application programs** define how those resources are used to solve the computing problems of the users
4. **Users** motivate the whole thing!



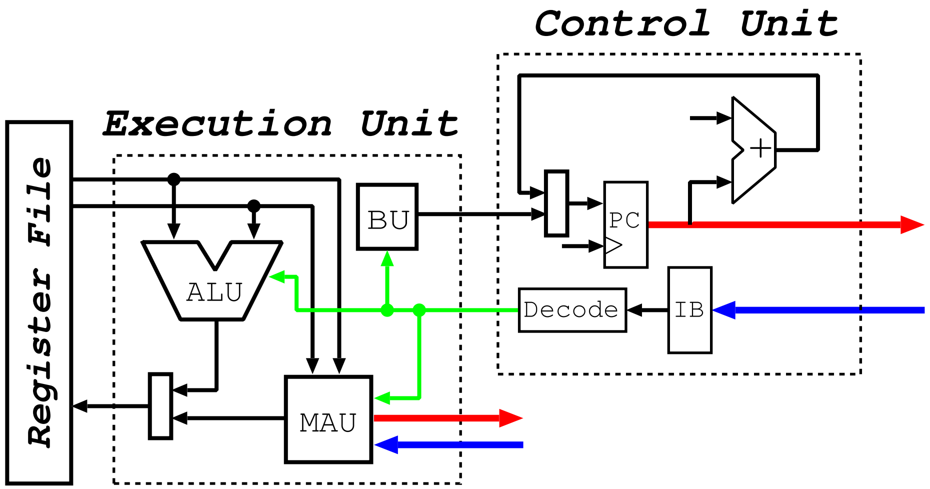
Hardware resources

- **Processor** (**CPU**) executes programs using
  - **Memory** to store both programs & data, effectively a large byte-addressed array,
  - **Devices** for input and output, and
  - **Bus** to transfer information between
- CPUs operate on data obtained from input devices and held in memory
  - CPUs and devices are concurrently active, competing for memory cycles and bus access
- Computer logically
  - Reads values from main memory into registers,
  - Performs operations, and
  - Stores results back

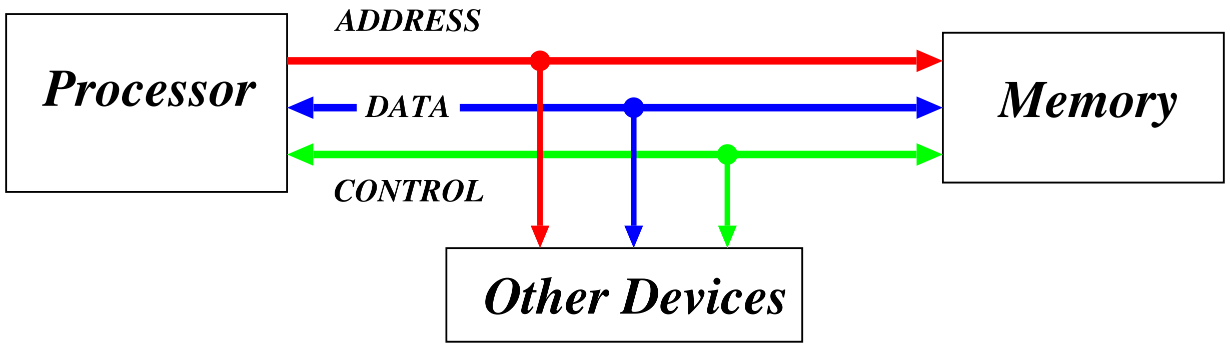


Fetch-Execute Cycle

- CPU repeatedly
  - Fetches & decodes next instruction,
  - Generating control signals and operand information
- Inside the **Execution Unit** (EU), control signals select the **Functional Unit** (FU) ("instruction class") and operation
  - If **Arithmetic Logic Unit** (ALU), read one/two registers, perform operation, (probably) write result back
  - If **Branch Unit** (BU), test condition and (maybe) add value to PC
  - If **Memory Access Unit** (MAU), generate address ("addressing mode") and use bus to read/write value



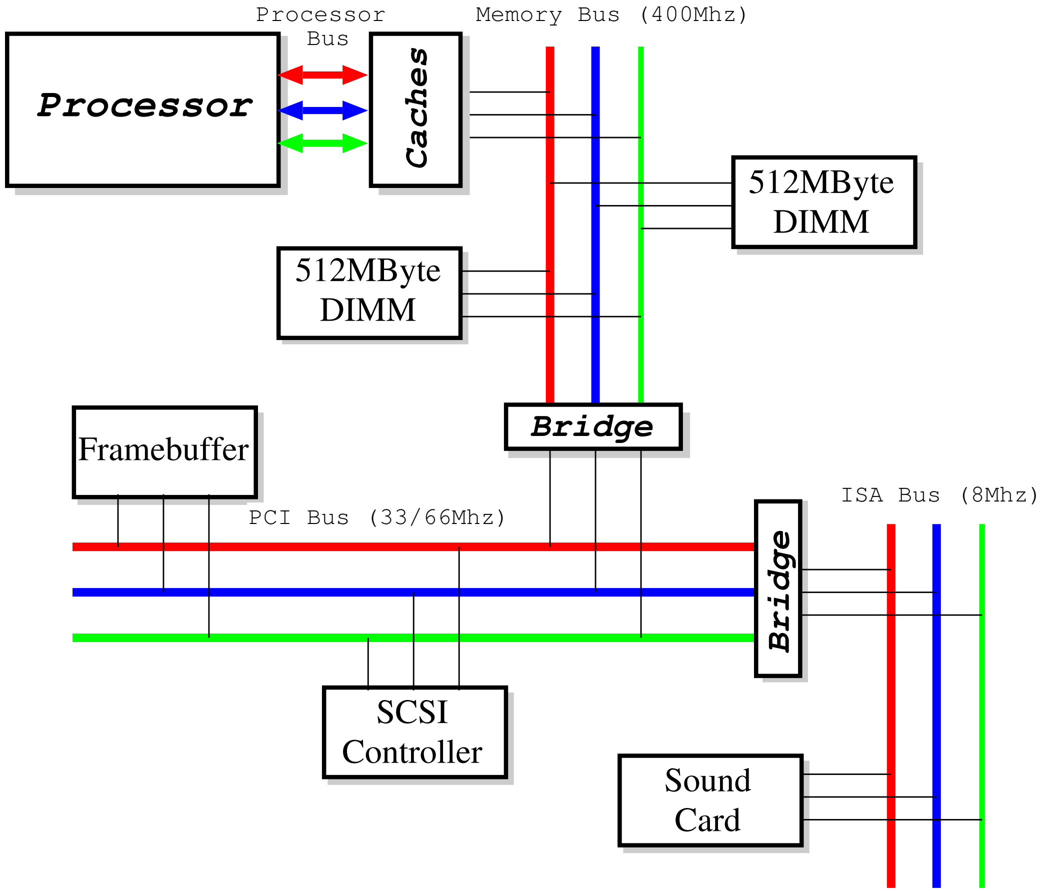
Buses



- Shared communication wires
  - Don’t need wires everywhere!
  - Low cost, versatile
  - Potential bottleneck
- Typically comprises:
  - **address lines** determine how many devices on bus,
  - **data lines** determine how many bits transferred at once, and
  - **control** lines indicate target devices and selected operations
- Operates in a initiator-responder manner, e.g.,
  - Initiator decides to read data
  - Initiator puts address onto bus and asserts read
  - Responder reads address from bus, retrieves data, and puts onto bus
  - Initiator reads data from bus

Bus hierarchy

- Different buses with different characteristics
  - E.g., data width, max number of devices, max length
  - Most are synchronous, i.e. share a clock signal
- **Processor bus** is the fastest and often the widest for CPU to talk to cache
- **Memory bus** to communicate with memory
- **PCI buses** to communicate with devices
  - Other legacy buses also seen: ISA, EISA etc
- **Bridges** forwards from one side to the other
  - E.g., to access a device on ISA bus, CPU generates magic [physical] address which is sent to memory bridge, then to PCI bridge, and then to ISA bridge, and finally to ISA device

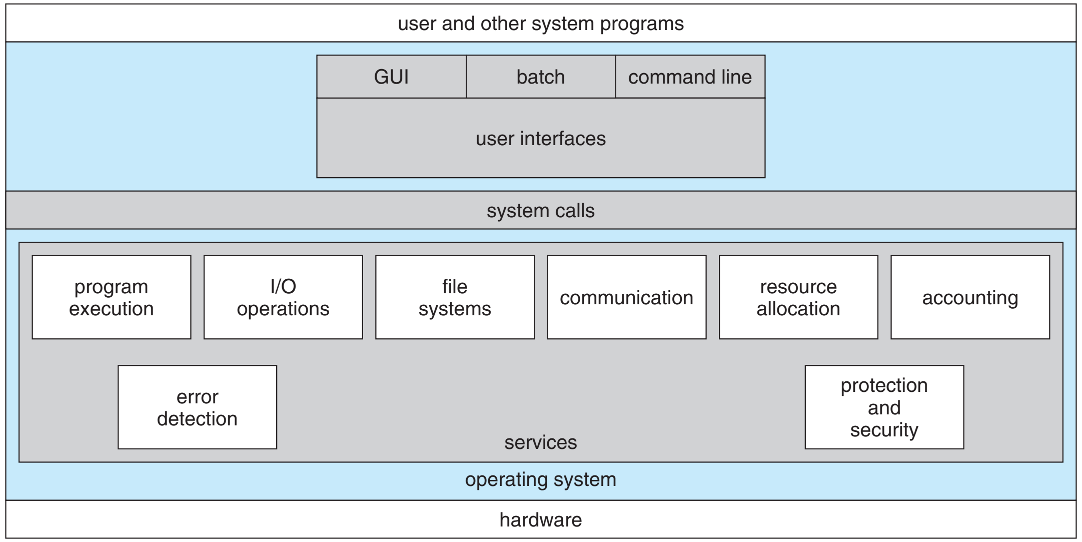


Outline

- System organisation
- System operation
  - Booting
  - Interrupts
  - Storage
- Concepts
- What is an Operating System?

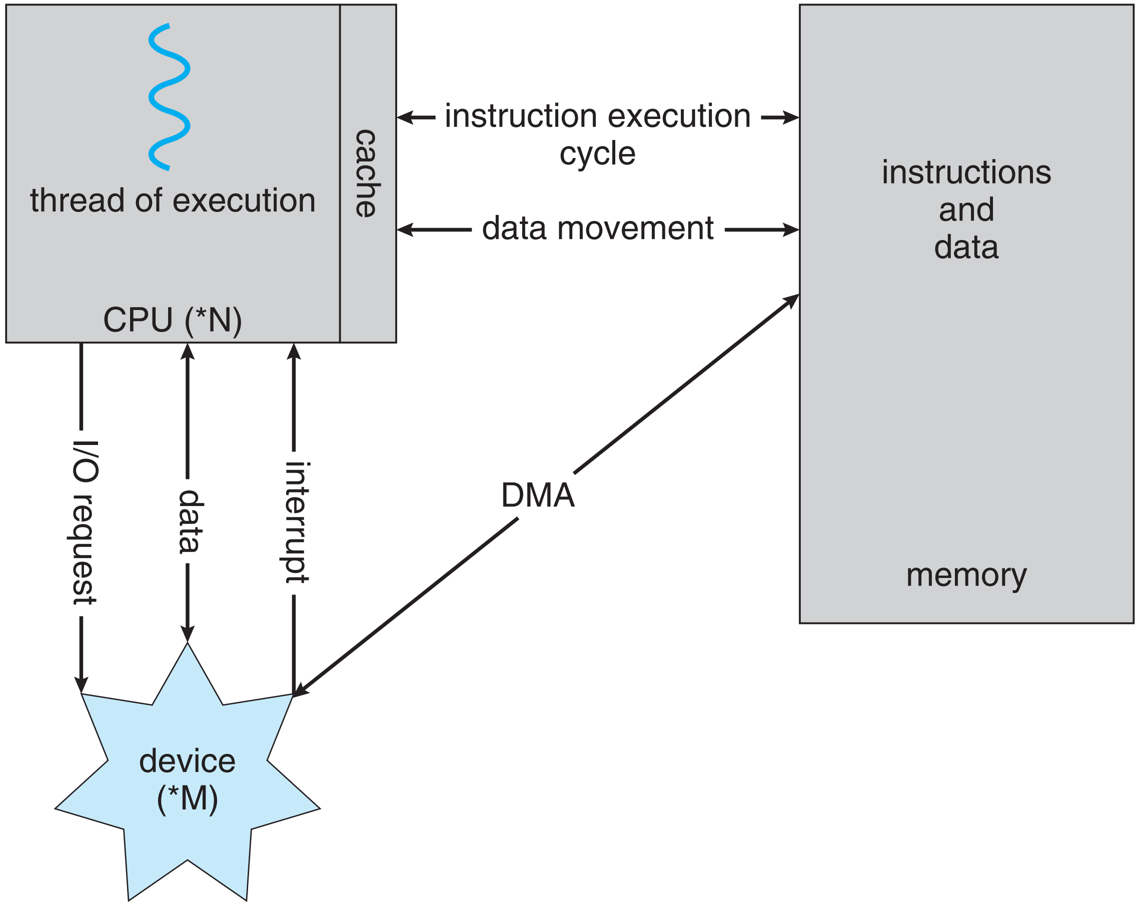
Booting the computer

- **Bootstrap program** (bootloader) executes when machine powered on
  - Traditionally ROM containing BIOS, now more complex UEFI
  - Initialises all parts of the system: memory, device controllers
  - Finds, loads, and executes the kernel, possibly in stages
- Operating system starts in stages
  - **Kernel** enables processes to be created, devices to be read/written, file system to be accessed
  - Then system processes start, beginning with *init* on Unix



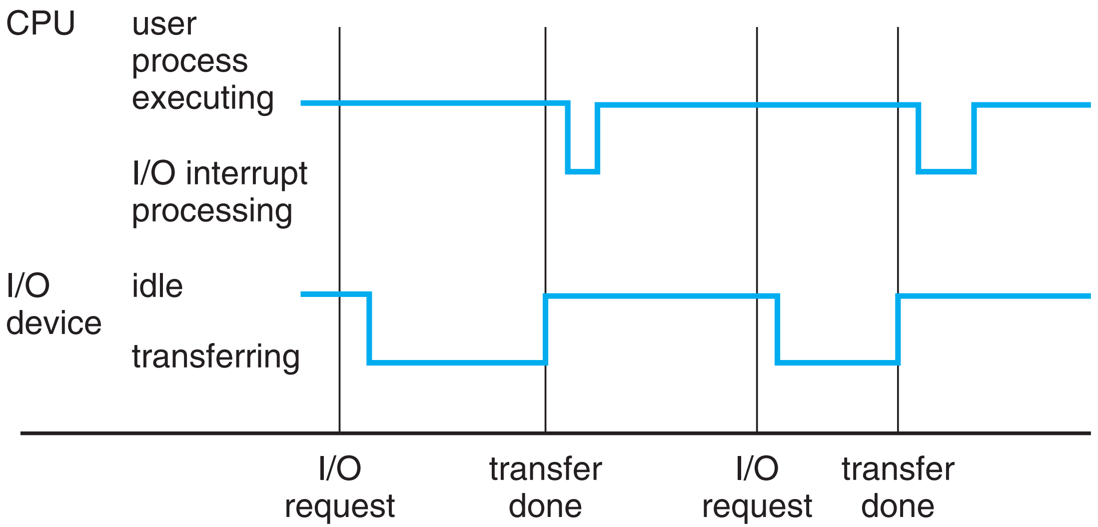
System operation

- I/O devices and CPU execute concurrently
- Each device controller
  - responsible for a particular device type
  - has a local buffer
- CPU moves data from/to main memory to/from local buffers
  - I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by raising an **interrupt**
  - OS is interrupt driven



Interrupts

- Device controllers communicate with CPU via **interrupts**
  - Controller controls interaction between device and local buffer
  - CPU moves data between main memory and device buffer



- Interrupts decouple CPU requests from device responses
  - Reading a block of data from a hard-disk might take 2ms, which could be 5×10^6 clock cycles!
- Controller informs CPU it is finished by **raising an interrupt**

Interrupt handling

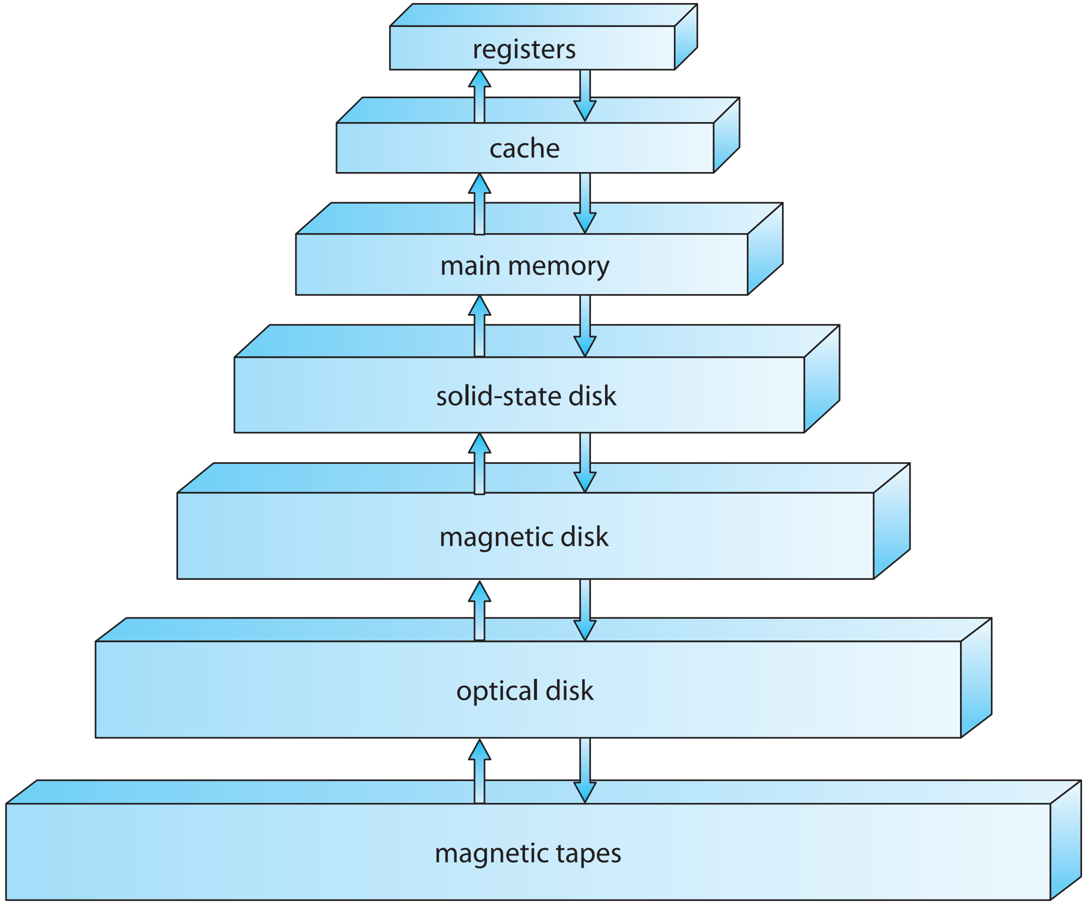
- A raised interrupt must be handled
  - Transfer control to the **interrupt service routine** (**ISR**) via
  - The **interrupt vector**, a table containing addresses of all the ISRs
  - Interrupt architecture saves the address of the interrupted instruction
  - After reading from device, CPU resumes using a special instruction, e.g., *rti*
- Interrupts can happen at any time
  - Typically deferred to an instruction boundary
  - ISRs must not trash registers, and must know where to resume
  - CPU thus typically saves values of all (or most) registers, restoring on return
- A **trap** or **exception** is a software-generated interrupt
  - Can be caused either by an error or a deliberate user request

Storage definitions

- Basic unit of computer storage is the **bit**, containing either 0 or 1
- A **byte** (or **octet**) is 8 bits, typically the smallest convenient chunk of storage
  - E.g., most computers can move a byte in memory but not a single bit
- A **word** is a given computer architecture's native unit of data, one or more bytes
  - E.g., a computer with 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words
- Storage generally measured and manipulated collections of bytes
  - A **kilobyte** (**KB**) is 1,024 bytes
  - A **megabyte** (**MB**) is $1{,}024^2$ bytes
  - A **gigabyte** (**GB**) is $1{,}024^3$ bytes
  - A **terabyte** (**TB**) is $1{,}024^4$ bytes
  - A **petabyte** (**PB**) is $1{,}024^5$ bytes
- Manufacturers often round so a megabyte is 1 million bytes and a gigabyte is 1 billion bytes

Storage hierarchy

- Storage systems organized in hierarchy
  - Speed, cost, volatility
- **Main memory** that the CPU can access directly
  - Large, random access, typically volatile
- **Secondary storage** extends main memory
  - Very large, non-volatile
  - **Hard disks** (**HDs**), rigid metal or glass platters covered with magnetic recording material divided logically into tracks, which are subdivided into sectors
  - **Solid-state disks** (**SSDs**), faster than hard disks, non-volatile
- **Device Driver** for each device controller to manage I/O provides a uniform interface between controller and kernel



Storage performance

Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Outline

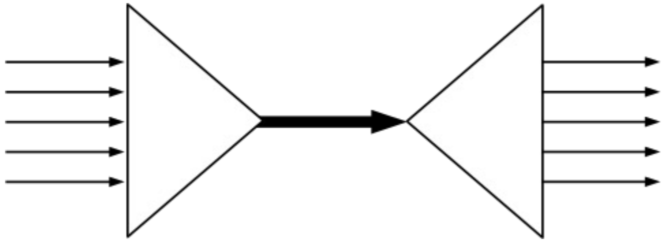
- System organisation
- System operation
- Concepts
    - Layering, multiplexing
    - Latency, bandwidth, jitter
    - Caching, buffering
    - Bottlenecks, tuning, 80/20 rule
    - Data structures
- What is an Operating System?

Layering, multiplexing

- **Layering** is a means to manage complexity by controlling interactions between components:
  - arrange components in a stack and restrict a component at layer X from
  - relying on any other component except the one at layer X-1 and
  - providing service to any component except the one at layer X+1

Application	Application
	Presentation
	Session
Transport	Transport
Internet	Network
Physical	Data Link
	Physical
Internet	**OSI**

- **Multiplexing** is where one resource is being consumed by multiple consumers simultaneously
  - Traditionally, the combination of multiple (analogue) signals into a single signal over a shared medium



Latency, bandwidth, jitter

- Different metrics of concern to systems designers
  - **Latency** is how long something takes

    E.g., “This read took 3ms”
  - **Bandwidth** is the rate at which something occurs
    - E.g., “This disk transfers data at 2Gb/s”
  - **Jitter** is the variation (statistical dispersal) in latency (frequency)
    - E.g., “Scheduling was periodic with jitter 50 μsec”
- Be aware
  - is it the absolute or relative value that matters, and
  - is the distribution of values also of interest

Caching, buffering

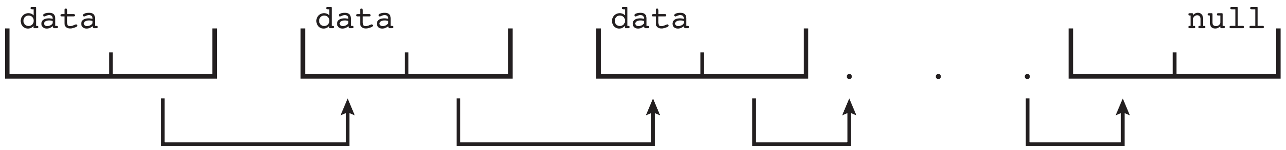
- Often need to handle two components operating at different speeds (latencies, bandwidths) – so-called **impedance mismatch**
- **Caching**, where a small amount of higher-performance storage is used to mask the performance impact of a larger lower-performance component. Relies on locality in time (finite resource) and space (non-zero cost)
  - E.g., CPU has registers, L1 cache, L2 cache, L3 cache, main memory
- **Buffering**, where memory of some kind is introduced between two components to soak up small, variable imbalances in bandwidth
  - E.g., A hard disk will have on-board memory into which the disk controller reads data, and from which the OS reads data out
  - No use if long-term average bandwidth of one component simply exceeds the other!

Bottlenecks, tuning, the 80/20 rule

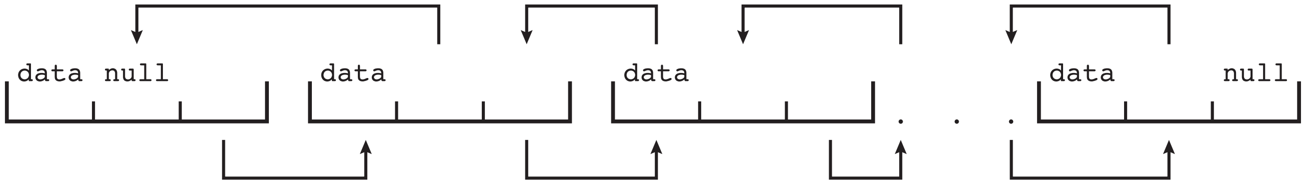
- The **bottleneck** is typically the most constrained resource in a system
- Performance optimisation and tuning focuses on determining and eliminating bottlenecks
  - Often introducing new ones in the process
- A perfectly balanced system has all resources simultaneously bottlenecked
  - Impossible to actually achieve
  - Often find that optimising the common case gets most of the benefit anyway
- Means that measurement is a prerequisite to performance tuning!
  - The 80/20 rule — 80% time spent in 20% code
  - No matter how much you optimise a very rare case, it will make no difference

Common data structures

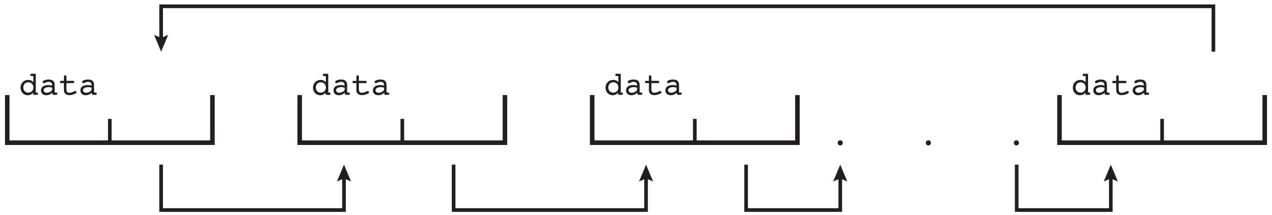
Linked list



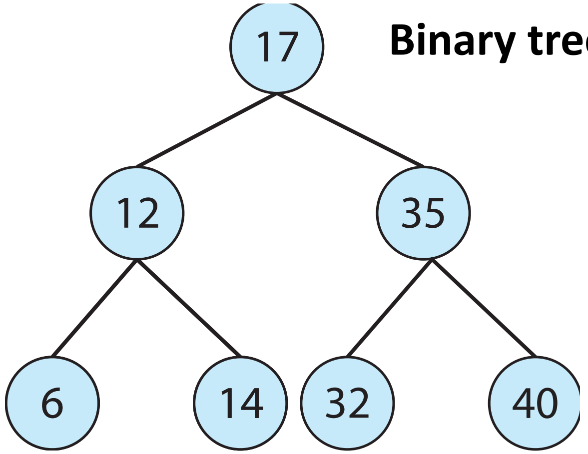
Doubly-linked list



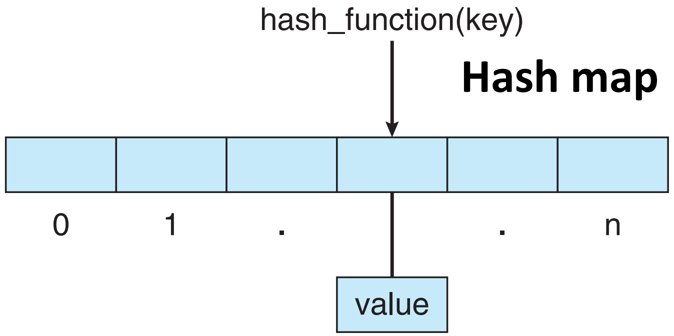
Circularly-linked list



Binary tree



Hash map



Outline

- System organisation
- System operation
- Concepts
- What is an Operating System?
    - Resource protection
    - CPU, memory, I/O

What is an Operating System?

- Just a program – a piece of software that (efficiently) provides
  - **Control**, over the execution of all other programs
  - **Multiplexing**, of resources between programs
  - **Abstraction**, over the complexity and low-level details
  - **Extensibility**, enabling evolution to meet changing demands and constraints
- Typically involves libraries and tools provided as part of the OS
  - Kernel – but also a *libc*, a language runtime, a web browser, …
  - Thus no-one really agrees precisely what the OS is
  - In this course we will focus on the **kernel**
- OS provides **mechanisms** that are used to implement **policies**
  - Policies may be deliberately designed, or accidents of implementation

Resource management

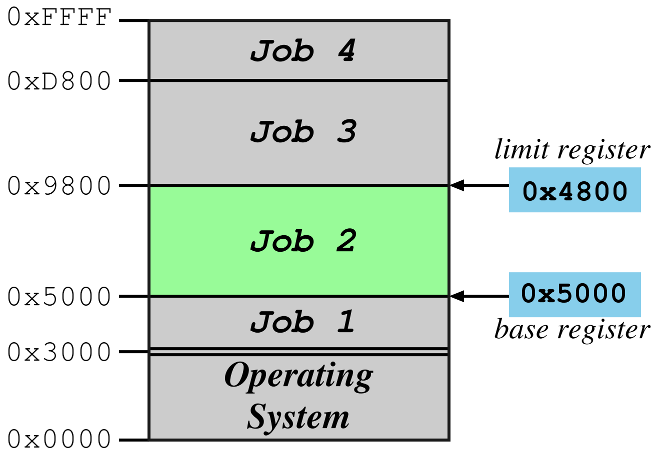
- Running program executes instructions sequentially to completion using resources
- **CPU**
  - OS multiplexes many running programs (threads) over the CPU(s)
  - Lifecycle management, synchronisation, communication
- **Memory**
  - Running programs require code and data in memory
  - Tracking memory ownership, managing de/allocation
- **Storage**
  - Abstracting different storage media and their characteristics
  - Creating, deleting, manipulating files, directories and free space
- **I/O Subsystem**
  - Abstracting peculiarities of different devices
  - Providing device drivers, managing I/O buffering, caching, spooling

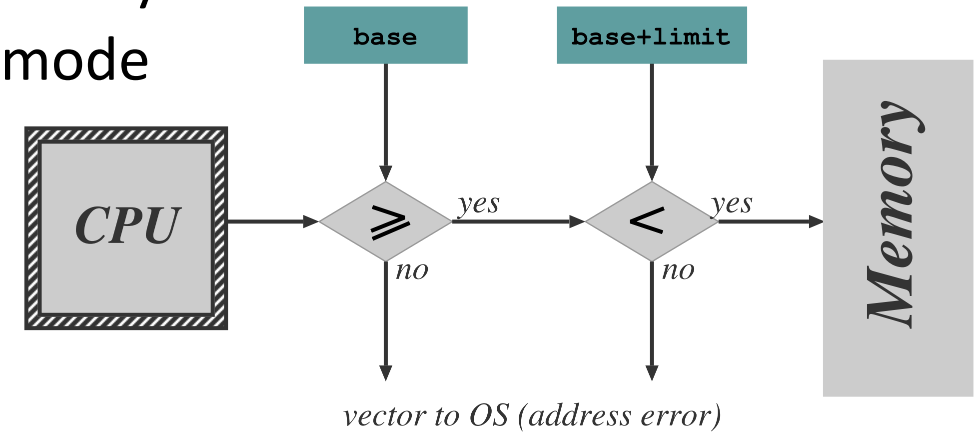
Protecting the CPU

- Need to ensure that the OS stays in control, able to prevent any application from “hogging” the CPU the whole time
- Means using a timer, usually a countdown timer, e.g.,
  - Set timer to initial value (e.g. 0xFFFF)
  - Every tick (nowadays programmable), timer decrements value
  - When value hits zero, interrupt
- Ensures the OS runs periodically provided
  - only OS can load timer, and
  - timer interrupt cannot be masked
- Also enables implementation of time-sharing

Protecting memory

- Define a base and a limit for each program, and protect access outside allowed range
- Have hardware check every memory reference:
  - Access out of range causes exception, vectored into OS
  - Only allow update of base and limit registers by OS
  - Can disable memory protection in kernel mode (but this is a bad idea)
- In reality, more complex protection hardware is used





Protecting I/O

- Initially, tried to make IO instructions privileged:
  - Applications can’t mask interrupts (that is, turn one or many off)
  - Applications can’t control IO devices
- Unfortunately, some devices are accessed via memory, not special instructions
  - Applications can rewrite interrupt vectors
- Hence protecting IO relies on memory protection mechanisms

Summary

- System organisation
  - Hardware resources
  - Fetch-execute cycle
  - Buses
- System operation
  - Booting
  - Interrupts
  - Storage
- Concepts
  - Layering, multiplexing
  - Latency, bandwidth, jitter
  - Caching, buffering
  - Bottlenecks, tuning, 80/20 rule
  - Data structures
- What is an Operating System?
  - Resource protection
  - CPU, memory, I/O