COS 122
Operating Systems

Open Course Notes

Based on
Operating Systems: Internal and Design Principles, 9th Edition by William Stallings

Available at `https://tinyurl.com/cos122-git`

Contents

PART I: BACKGROUND

Chapter 1: Computer System Overview

Unit 1.1: Basic Elements of a Computer System

A computer, at a top level, consists of processor, memory, and input/output (I/O) components, with one or more modules of each type. These components are interconnected in a certain way to allow the computer to function as a machine that can execute programs. There are four main elements:

- **Processor:** Controls the operation of the computer and performs its data processing functions. When a single processor is used, it is often referred to as the *central processing unit (CPU)*.
- **Main memory:** Stores data and programs. The memory is generally volatile, meaning that when the computer is turned off, the contents of memory are lost. It is also referred to as *primary memory* or *real memory*.
- **I/O modules:** Provide the means for the computer to communicate with the external environment, which may consist of secondary memory (e.g. hard disks), communication equipment (e.g. keyboard, printer), terminals.
- **System bus:** Provides a communication path for the various components of the computer system.

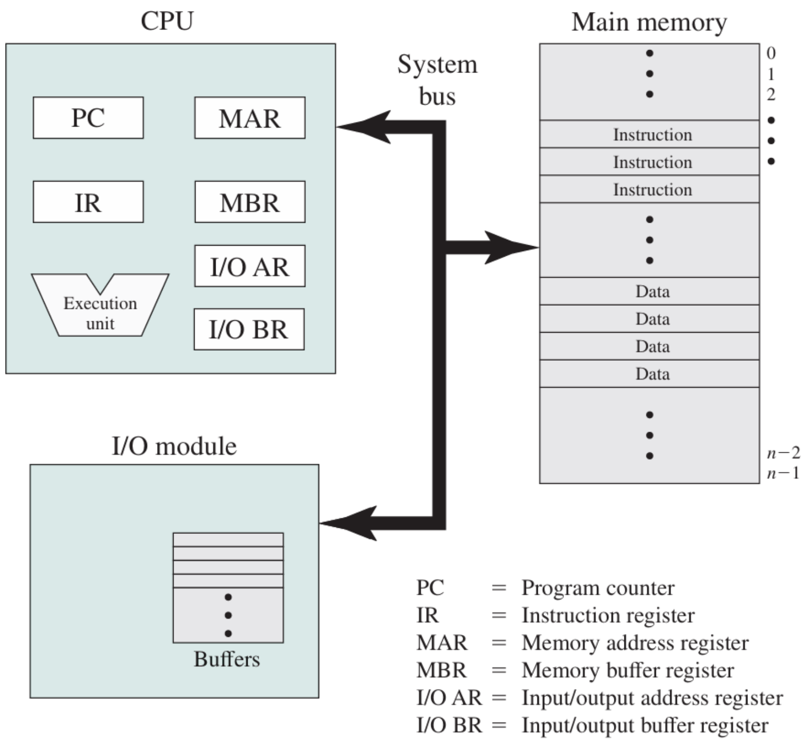
CPU
PC
MAR
IR
MBR
I/O AR
Execution unit
I/O BR
System bus
Main memory
0
1
2
Instruction
Instruction
Instruction
Data
Data
Data
Data
n−2
n−1
I/O module
Buffers
PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

Figure 1: **Computer Components: Top-Level View**

One of the most important functions of the processor is to exchange data with the memory. For this purpose, the processor contains two internal registers: the *memory address register (MAR)* and the *memory buffer register (MBR)*. The MAR contains the address of the memory location to be accessed. The MBR contains the data to be written into the memory or the data received from the memory.
In addition to the MAR and MBR, the processor contains a number of other registers:

- **Program counter (PC):** Contains the address of the next instruction to be executed.
- **Instruction register (IR):** Contains the instruction currently being executed.
- **I/O address register (I/O AR):** Contains the address of the I/O module to be accessed.

- **I/O buffer register (I/O BR):** Contains the data to be written into the I/O module or the data received from the I/O module.

Note

The buffer or the address register?
Be careful with the terminology. The term *buffer* is used for registers that hold data, while the term *address register* is used for registers that hold addresses.

Figure 1 shows the internal organisation of the processor and its connection to the memory and I/O modules. A memory module is a set of locations, defined by sequential addresses. Each location stores a fixed number of bits, called a *word*. A word can represent data or an instruction. The number of bits in a word is called the *word length* of the computer. An I/O module transfers data between the external device, processor, and memory. It contains internal buffers to temporarily hold data being transferred.

Unit 1.2: Evolution of the Microprocessor

The invention of the microprocessor—a processor on a single chip—sparked the desktop and handheld computing revolution. Modern microprocessors often have multiple cores and logical processors, while GPUs provide efficient parallel computation beyond graphics. CPUs now include powerful vector units, and specialised processors like DSPs handle audio, video, and other streaming signals. In handheld devices, the trend is toward System on a Chip (SoC) designs, integrating CPUs, caches, DSPs, GPUs, I/O devices, and memory on a single chip for compact and efficient computing.

Unit 1.3: Instruction Execution

A program is a sequence of instructions that specifies a computation. Each instruction executed by the processor is stored in memory as a binary number. The execution of a program consists of a sequence of fetch and execute cycles.

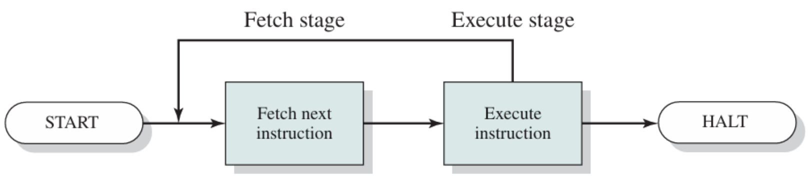

Figure 2: **Fetch-Execute Cycle**

At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched. Unless instructed otherwise, the processor always increments the PC after each instruction fetch so it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address). For example, consider a simplified computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained subsequently. The fetched instruction is loaded into the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action.

Most modern processors include instructions that contain more than one address. Thus, the execution stage for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation.

Method

Instruction Execution Using the Fetch-Decode-Execute Cycle

1. **Fetch the instruction:**
   - Read the instruction from the memory location pointed to by the PC into the IR.
   - Increment the PC to point to the next instruction.
2. **Decode the instruction:**
   - Examine the opcode in the IR to determine the operation to perform.
   - Identify any operands or memory addresses involved.
3. **Execute the instruction:** Perform the operation specified by the instruction:
   (a) For a load instruction, copy the contents from the specified memory address into the accumulator (AC).
   (b) For an arithmetic instruction, perform the calculation using AC and memory contents.
   (c) For a store instruction, write the contents of AC back to the specified memory address.
4. **Repeat:** Return to the fetch step for the next instruction and continue until the program completes.

Unit 1.4: Interrupts

An interrupt is a signal from a device or software indicating the need for attention or a request for a service. When an interrupt occurs, the processor temporarily halts its current activities, saves its state, and executes a function called an interrupt handler to deal with the event. After the interrupt has been serviced, the processor resumes normal activities from where it left off.
The following table lists some common types of interrupts:

Interrupt Type	**Description**
Program	Generated by an error or exception in the program, such as division by zero or invalid memory access.
Timer	Generated by a timer within the processor to allow the operating system to perform periodic tasks.
I/O	Generated by an I/O device to signal the completion of an operation or to request service.
Hardware Failure	Generated by hardware malfunctions, such as power failures or memory errors.

Table 1: Classes of Interrupts

Interrupts improve the efficiency and responsiveness of a computer system by allowing it to handle asynchronous events and prioritise tasks effectively.
Consider these motivations:

- I/O devices are much slower than the CPU and memory.
- Without interrupts, the CPU would have to wait for I/O operations to complete, wasting valuable processing time.
- With interrupts, the CPU can continue executing other instructions while waiting for I/O operations to finish.
- When an I/O operation completes, the device sends an interrupt signal to the CPU, which temporarily halts its current activities to service the interrupt.
- This allows the CPU to efficiently manage multiple tasks and respond to events in a timely manner.

Let us now compare program execution with and without interrupts.

Example

Program Execution Without Interrupts
Study the figure which illustrates program execution without interrupts and the accompanying description.

User Program

1. Execute code segment 1 that does not involve I/O
2. WRITE calls (pause user program and execute *I/O program* on processor)
3. Execute code segment 2 that does not involve I/O

I/O Program

1. Segment 4 copies data to the write buffer
2. I/O command prints the data on printer (processor is idle while waiting for I/O operation to complete)
3. Segment 5 notifies user program about success or failure of WRITE operation

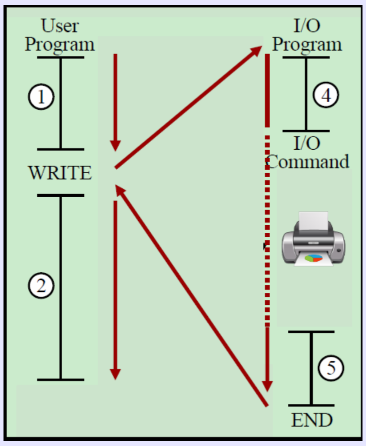

Figure 3: **Execution without interrupts**

Notice that without interrupts, the user program must wait until the I/O operation is complete before it can continue executing.

Example

Program Execution With Interrupts
Study the figure which illustrates program execution with interrupts and the accompanying description.

1. After **Segment 4**, the processor continues executing the **user program** (Segment 2) without waiting for the I/O to complete.
2. The **I/O module** executes the I/O command independently, running in parallel with the user program.
3. When the I/O operation finishes, the I/O module sends an **interrupt request** to the processor.
4. The processor **temporarily suspends** the user program and switches to the **interrupt handler routine**.
5. The interrupt handler (Segment 5) completes the I/O service, notifies the user program of the result, and the processor **resumes the user program** exactly where it was interrupted.

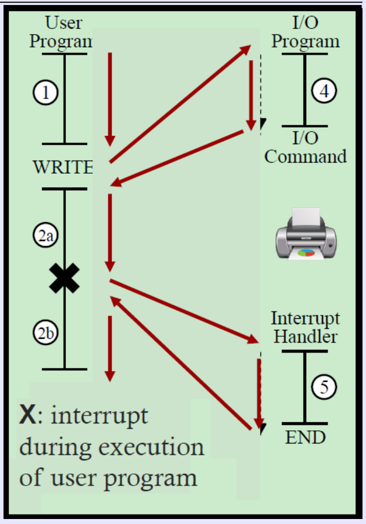

Figure 4: **Execution with interrupts**

The key advantage of using interrupts is that the processor can perform other tasks while waiting for I/O operations to complete, thereby improving overall system efficiency and responsiveness. Note that interrupts can occur at almost any time, so the state of the user program must be saved and restored correctly to ensure proper execution.

To accommodate interrupts, an interrupt stage is added to the instruction cycle, as shown in the figure below.

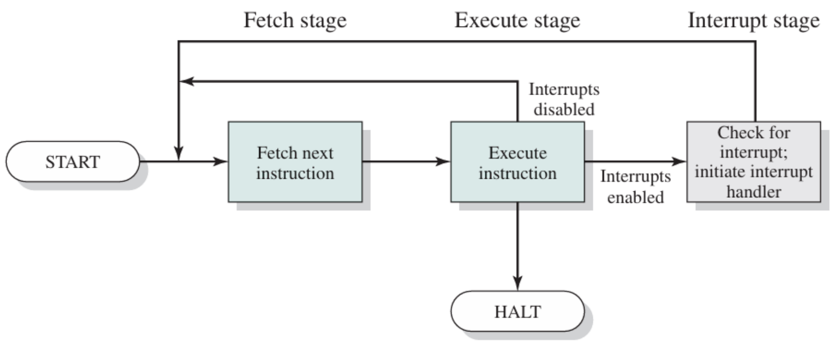

Figure 5: **Instruction Cycle with Interrupts**

To handle interrupts, the processor must perform several steps during the interrupt stage.

Method

Interrupt Handling by the Processor

1. **Save the status of the user program:** The processor stores the current state (registers, program counter, etc.) of the interrupted user program on the stack.
2. **Set the Program Counter (PC):** The PC is updated to point to the start of the Interrupt Service Routine (ISR).
3. **Execute the Interrupt Service Routine:** The processor executes the instructions in the ISR to handle the interrupt.
4. **Restore the user program state:** After completing the ISR, the processor loads the saved state of the user program from the stack.
5. **Resume user program execution:** The processor continues execution of the user program exactly where it was interrupted.

Naturally, this interrupt handling process introduces some overhead processes, as the processor must save and restore the program state. However, the benefits of improved responsiveness and efficient multitasking generally outweigh this overhead.

The following figure illustrates the changes in the memory and stack during interrupt handling.

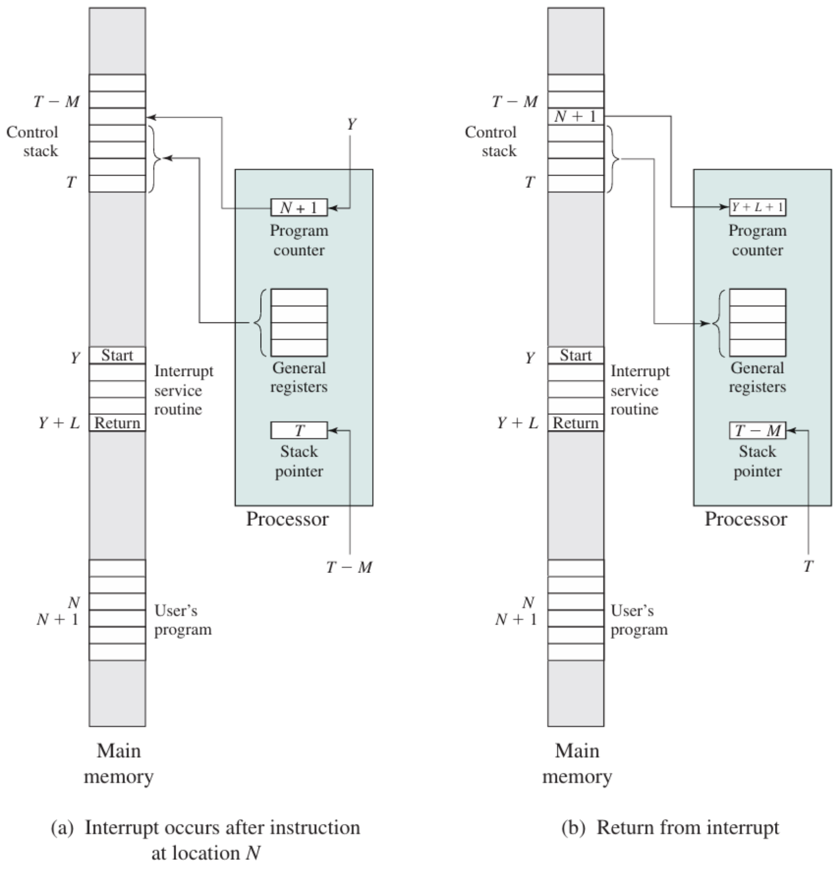

Figure 6: **Memory and Stack Changes During Interrupt Handling**

It is important to note that an interrupt may occur even while the processor is executing an interrupt handler. For example, the computer may be receiving data from a keyboard while it is printing a document.
There are two common strategies to handle such nested interrupts:

- **Sequential Interrupt Handling:** The processor completes the current interrupt handler before servicing any new interrupts. New interrupts are queued until the current handler finishes.
- **Priority-Based Interrupt Handling:** Each interrupt is assigned a priority level. If a higher-priority interrupt occurs while a lower-priority handler is executing, the processor temporarily suspends the lower-priority handler to service the higher-priority interrupt. After handling the higher-priority interrupt, the processor resumes the lower-priority handler.

Unit 1.5: The Memory Hierarchy

The design constraints on memory can be reduced to three main factors: capacity, speed, and cost. Considering capacity is a rather straight-forward issue. The more memory a computer has, the more programs and data it can store.
Speed, however, is a more complex issue. The speed of memory is usually measured in terms of access time, which is the time interval between the request for a word and the delivery of the word. Memory access time must be considered in relation to the speed of the processor–it must keep up with the processor. If the memory is too slow, the processor will have to wait for it to deliver data, resulting in wasted processing time.
Cost is the final factor. The cost of memory is usually expressed in terms of cost per bit. The lower the cost per bit, the more memory can be provided within a given budget. The cost of memory must also be reasonable in relation to the cost of the other hardware components.

As one might expect, there is a trade-off among these three factors:

- Shorter access time ⇒ Higher cost per bit
- Larger capacity ⇒ Lower cost per bit
- Larger capacity ⇒ Longer access time

The solution to this trade-off is to use a memory hierarchy, which is a structure that uses multiple types of memory, each with different speeds, costs, and capacities.

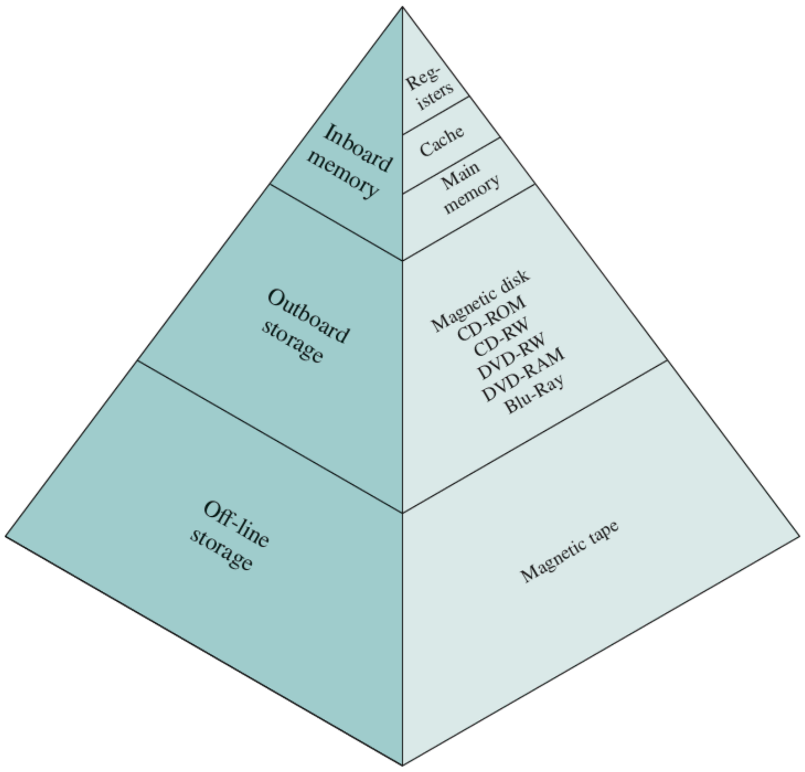

Figure 7: **The Memory Hierarchy**

Figure 7 illustrates a typical memory hierarchy. The trend, going from the top to the bottom of the hierarchy, is as follows:

- Decreasing cost per bit
- Increasing capacity
- Increasing access time (i.e., slower speed)
- Decreasing frequency of access to memory by the processor

Example

Two-Level Memory Hierarchy Example
A processor has access to two levels of memory:

- **Level 1** (L_1), with 1 000 bytes and an access time of 0.1 μs. (This is typically a cache memory.)
- **Level 2** (L_2), with 100 000 bytes and an access time of 1 μs. (This is typically the main memory.)

Assume that if data is found in L_1, then the processor accesess it directly. (This is called a *hit.*) If the data is found in L_2, then it is copied to L_1 and accessed from there. (This is called a *miss.*) The average access time of the memory hierarchy can be calculated using the formula:

$$\text{Average Access Time} = (h \times t_1) + \big((1-h) \times (t_1 + t_2)\big)$$

where:

- h is the hit ratio (the fraction of accesses found in L_1),
- t_1 is the access time of L_1,
- t_2 is the access time of L_2.

Suppose that 95% of the memory accesses are found in L_1 (i.e., $h = 0.95$). Then:

$$\text{Average Access Time} = (0.95 \times 0.1\mu s) + (0.05 \times \mu s + 1\mu s) = 0.095\mu s + 0.05\mu s = 0.15\mu s.$$

Now the big question is: How can we achieve a high hit ratio? Observations show that memory accesess by the processor tend to cluster (loops, arrays, etc.). This is called the *principle of locality.* The solution is to organise data across the memory hierarchy in such a way that the current cluster in use is in the fastest (the cache) memory.

Definition

Principle of Locality
The principle of locality is the tendency of a program to access a relatively small portion of its address space at any given time, meaning that recently accessed data and instructions are likely to be accessed again soon.

Unit 1.6: Cache Memory

On all instruction cycles, the processor accesess memory at least once to fetch the instruction and often more than once to fetch data. If the memory is slow, the processor will have to wait for it to deliver data, resulting in wasted processing time.
The solution to this problem is to exploit the principle of locality by providing a small, fast memory called *cache memory* that holds the most frequently used instructions and data.
Cache principles are as follows:

- The cache is smaller and faster than main memory.
- The cache is located between the processor and main memory.
- When the processor needs to read or write a location in main memory, it first checks whether that location is in the cache.
- If the location is in the cache (a *hit*), the processor reads or writes the location in the cache.
- If the location is not in the cache (a *miss*), the processor reads the location from main memory and also copies it to the cache.
- The cache is managed by hardware, which automatically handles the transfer of data between the cache and main memory.

Cache memory can be organised in several ways, but the most common is to divide the cache into blocks (clusters), where each block holds a fixed number of words. The cache consists of slots that can hold one block each. If a word is not in the cache, an entire block containing that word is copied from main memory to the cache. A binary tag is then added to identify each block in the cache.

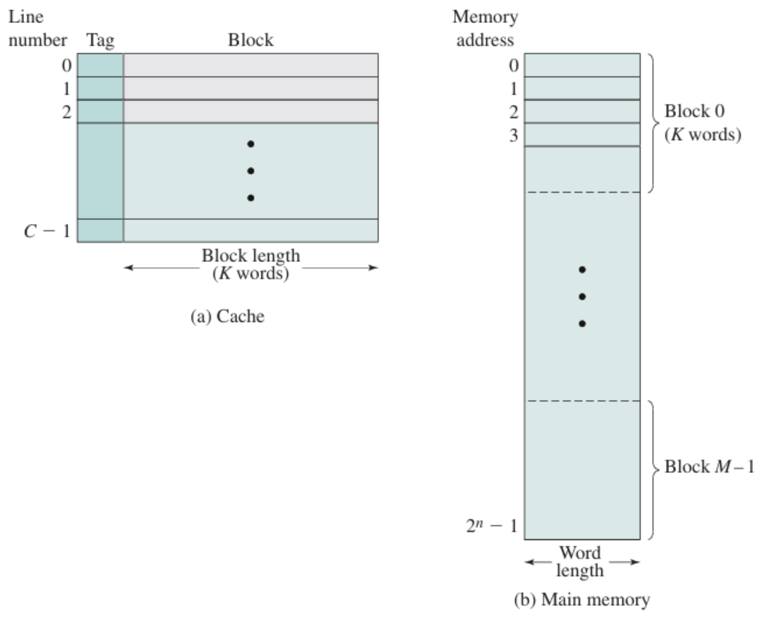

Figure 8: **Cache/Main Memory Organisation**

As a simple example, suppose we have a 6-bit address and a 2-bit tag. The tag 01 refers to the block of locations with the following addresses: [010000, 010001, 010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011, 011100, 011101, 011110, 011111].

The following figure illustrates the read operation.

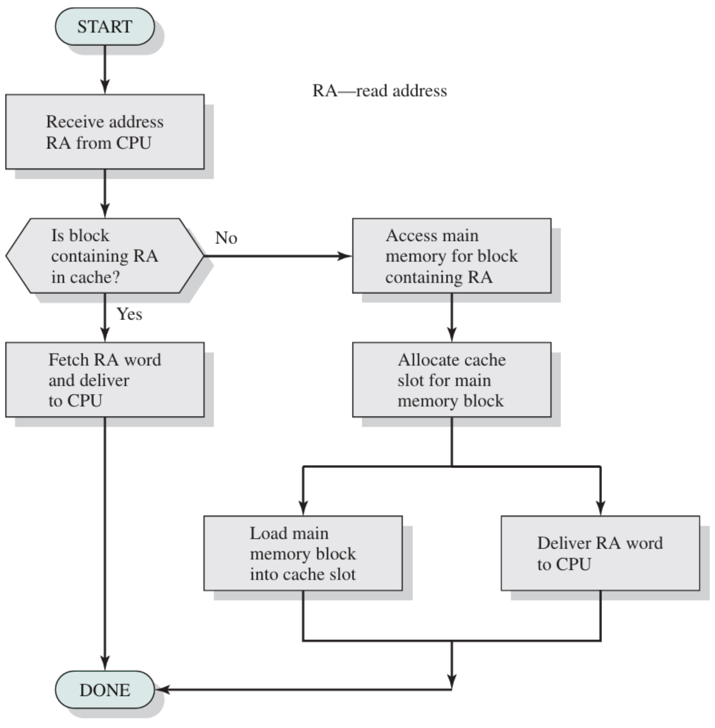
START
RA—read address
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache slot for main memory block
Load main memory block into cache slot
Deliver RA word to CPU
DONE

Figure 9: **Cache Read Operation**

The design of cache memory involves several considerations:

- **Cache Size:** The size of the cache affects the hit ratio. A larger cache can store more data, potentially increasing the hit ratio, but it also increases cost and access time.
- **Block Size:** The block size determines how much data is transferred between main memory and the cache. As the block size increases, the hit ratio may improve due to the principle of locality, but larger blocks also increase the time taken to transfer data. [Recall that larger capacity $\Rightarrow$ longer access time.]
- **Mapping Function:** The mapping function determines which cache slot a block will occupy. This is subject to two constraints:
  - When one block is read in, it might replace another block already in the cache.
  - The more flexible the mapping function, the more complex the search for a block in the cache.
- **Replacement Algorithm:** When a new block is loaded into the cache, a decision must be made about which existing block to replace if the cache is full. A common strategy is the Least Recently Used (LRU) algorithm, which replaces the block that has not been accessed for the longest time.
- **Write Policy:** The write policy dictates when an updated cache block is written back to main memory. There are two extreme policies:
  - **Write-Through:** Writing occures every time the block is updated.
  - **Write-Back:** Writing occurs only when the block is replaced.

  The latter policy minimizes memory write operations, but leaves main memory in an obsolete state. This can interfere with multiple-processor operation, and with direct memory access by I/O hardware modules.

Unit 1.7: Multiprocessors and Multicore Organisation

Traditionally, the computer has been viewed as a sequential machine. Most com puter programming languages require the programmer to specify algorithms as sequences of instructions. A processor executes programs by executing machine instructions in sequence and one at a time.
As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability.

Definition

Symmetric Multiprocessor (SMP)
An SMP is a stand-alone computer system with the following characteristics:

1. Two or more similar processors of similar capability.
2. The processors share a common main memory and I/O facilities, and are connected to the processors by a system bus or an interconnection network such that each processor has approximately equal memory access time.
3. All processors can perform the same functions (hence the term *symmetric*).
4. Communcation among processors is by means of shared memory.
5. The system is controlled by an operating system that is capable of supporting multiple processors.

An SMP organisation has several advantages over a single-processor organisation:

- **Performance:** With multiple processors, an SMP can execute multiple processes simultaneously (*in parallel*), improving overall system throughput and responsiveness.
- **Availabilty:** The failure of one processor does not necessarily bring down the entire system. Other processors can continue to operate, providing a degree of fault tolerance.
- **Incremental growth:** A user can enhance the performance of an SMP by adding more processors.
- **Scalability:** Vendors can offer a range of products with different price and performance characteristics.

The following figure illustrates a typical SMP organisation.

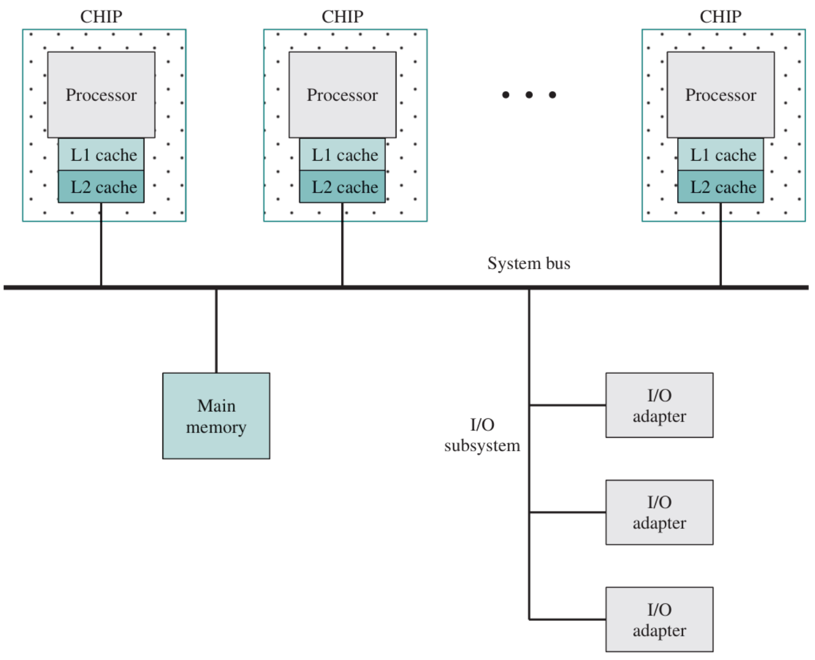

Figure 10: **Symmetric Multiprocessor Organisation**

Definition

Multicore Processor
A multicore processor is a single computing component with two or more independent processing units called *cores*. The cores are integrated onto a single chip or die

Each core in a multicore processor can have its own cache memory, and multiple cores can share a common cache. The cores are connected to each other and to the main memory via a high-speed interconnect, allowing for efficient communication and data sharing. External communication to other chips is handheld by controllers on the chip.

Note

Multicore or Multiprocessor?
A multicore processor is a single chip with multiple processing units (cores), while a multiprocessor system consists of multiple separate processors.

Chapter 2: Operating System Overview

Unit 2.1: Operating System Objectives and Functions

In the previous chapter, we discussed the basic elements of a computer system. From an abstract point of view, all these components can be viewed as 'resources'. The main task of an operating system (OS) is to manage these resources and allocate them to users and programs in an efficient and fair manner.

Definition

Operating System (OS)
An OS is a program that controls the execution of application programs, and acts as an interface between the user of a computer and the computer hardware.

An OS can be thought of as having three main objectives:

- **Convenience:** The OS makes the computer system convenient to use by providing a user-friendly interface.
- **Efficiency:** The OS manages the hardware resources to ensure that they are used efficiently, maximising system performance.
- **Ability to evolve:** The OS provides a stable and consistent environment that allows application programs to evolve over time without being affected by changes in the underlying hardware.

The OS an a User/Computer Interface

The hardware and software of a computer system are complex, as depicted in the figure below.

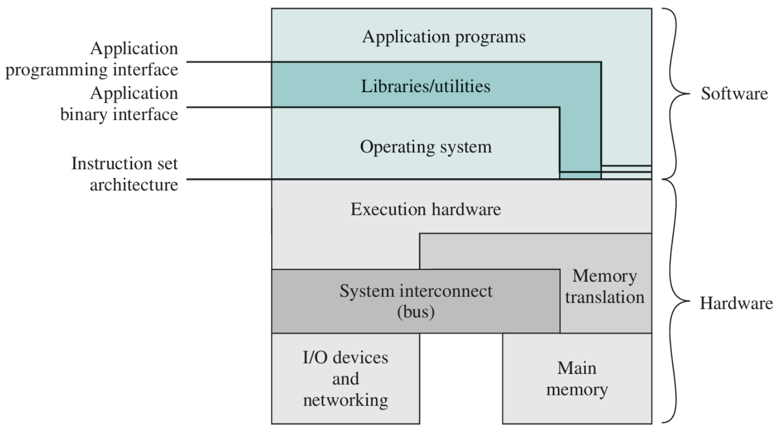
Application programming interface
Application binary interface
Instruction set architecture
Application programs
Libraries/utilities
Operating system
Execution hardware
System interconnect (bus)
Memory translation
I/O devices and networking
Main memory
Software
Hardware

Figure 1: **Computer System Components**

From this figure, we can see that the user is 'shielded' from the confusing details of the hardware. Instead of seeing the hardware directly, the user sees the OS. The OS provides well-defined functions that allow the user to interact with the hardware without needing to understand its complexities. The hardware's resources are similarly 'shielded' from inappropriate access or unwanted utilisation by incompetent or malicious users.
The OS provides support services for:

- Program development and execution
- Access to I/O devices
- Controlled access to files
- Concurrent access to the system by multiple users
- Error detection and recovery
- Accounting and performance monitoring

The OS as a Resource Manager
Since the OS itself is a program, it competes with other programs for the same hardware resources, and must therefore manage itself.
The OS has to frequently 'relinquish' control to the user program, and must rely on specialised hardware mechanisms to regain control. (Recall from *Unit 1.4: Interrupts.*)

Unit 2.2: Evolution of Operating Systems

A major OS will evolve over time for a number of reasons:

- Hardware upgrades
- New hardware types
- New services
- Fixes

The evolution of an OS can be illustrated by the following figure:

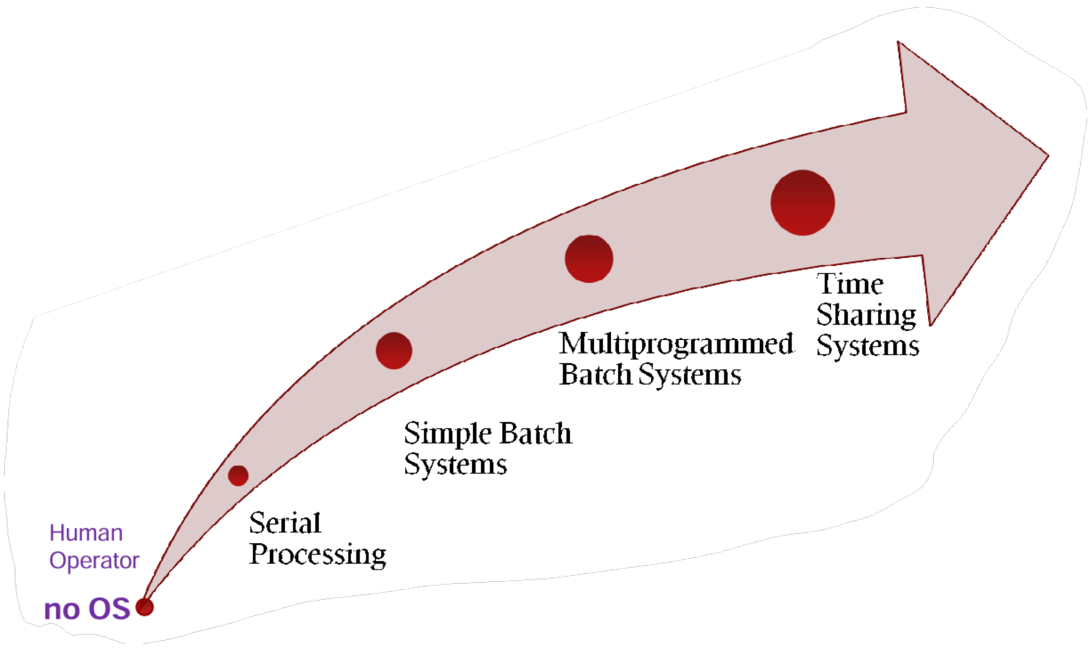

Figure 2: **Evolution of an Operating System**

Early OSs mainly mimicked the work-behaviour of the human operator.
The underlying principle was that a user-program U had to run entirely before another user-program V could run. This is called a *batch system.*
The *monitor* component of the OS had to recognise the completion of U, and then carry out the necessary administrative tasks (code loading, etc.) to prepare for the execution of V, and so on, until an entire 'batch' of jobs had been executed.

To understand how this scheme works, let us look at it from two points of view: that of the monitor, and that of the processor.

- **Monitor's Point of View:** The monitor controls the sequence of events, and must always be in main memory and available for execution. This is called the *resident monitor*. The monitor reads in a job and gives it control. The job then returns control to the monitor when it terminates.

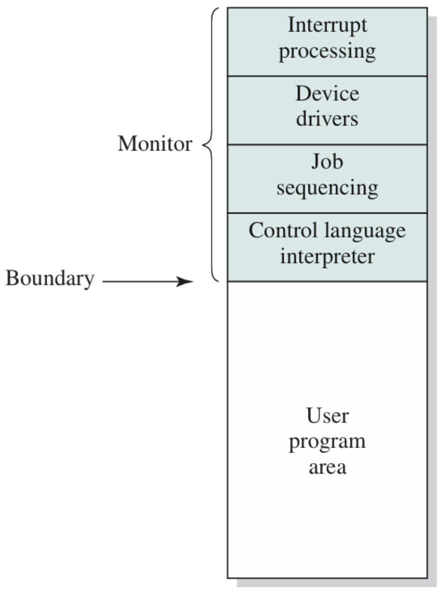

Figure 3: **Memory Layout for a Resident Monitor**

- **Processor's Point of View:** Note that the processor does not know whether it is executing a user program or the monitor. Notice the following implications:
  - CPU executes an instruction from the memory containing the monitor $\Rightarrow$ OS is running.
  - CPU executes an the instructions from the user program (until an error or termination) $\Rightarrow$ user program is running.

  When control is transferred from the monitor to the user program, the CPU is fetching and executing instructions from the user program. When control is transferred back to the monitor, the CPU is fetching and executing instructions from the monitor.

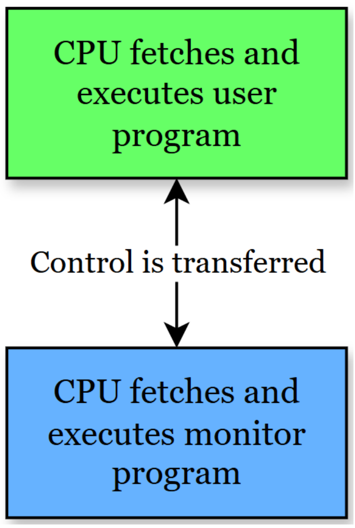
CPU fetches and executes user program
Control is transferred
CPU fetches and executes monitor program

Figure 4: **Transfer of Control Between Monitor and User Program**