COS 122: Operating Systems
Open Lecture Notes

Mogale Lebethe

August 28, 2025

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.

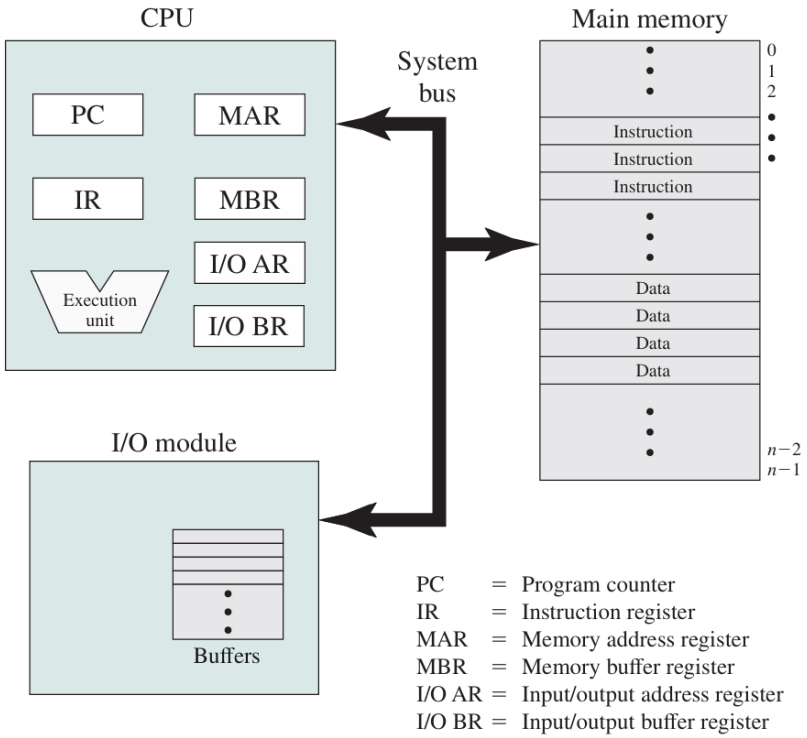

Figure 1: **Computer Components: Top-Level View**

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.

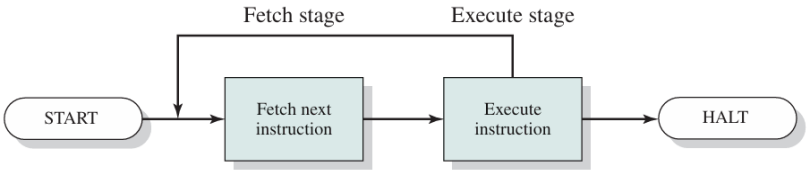

Figure 2: **Fetch-Execute Cycle**

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

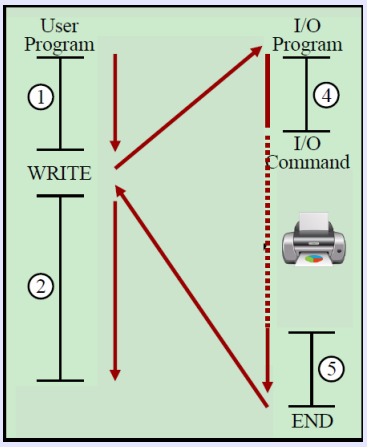

Figure 3: **Execution without interrupts**

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

Example

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.

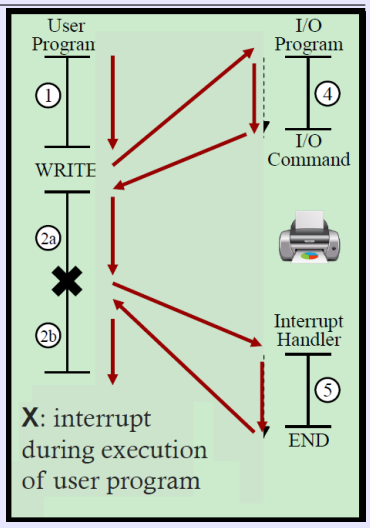

Figure 4: **Execution with interrupts**

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.

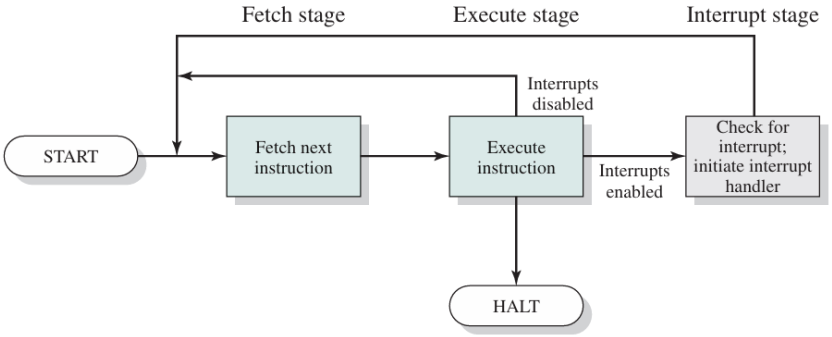

Figure 5: **Instruction Cycle with Interrupts**

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, 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.

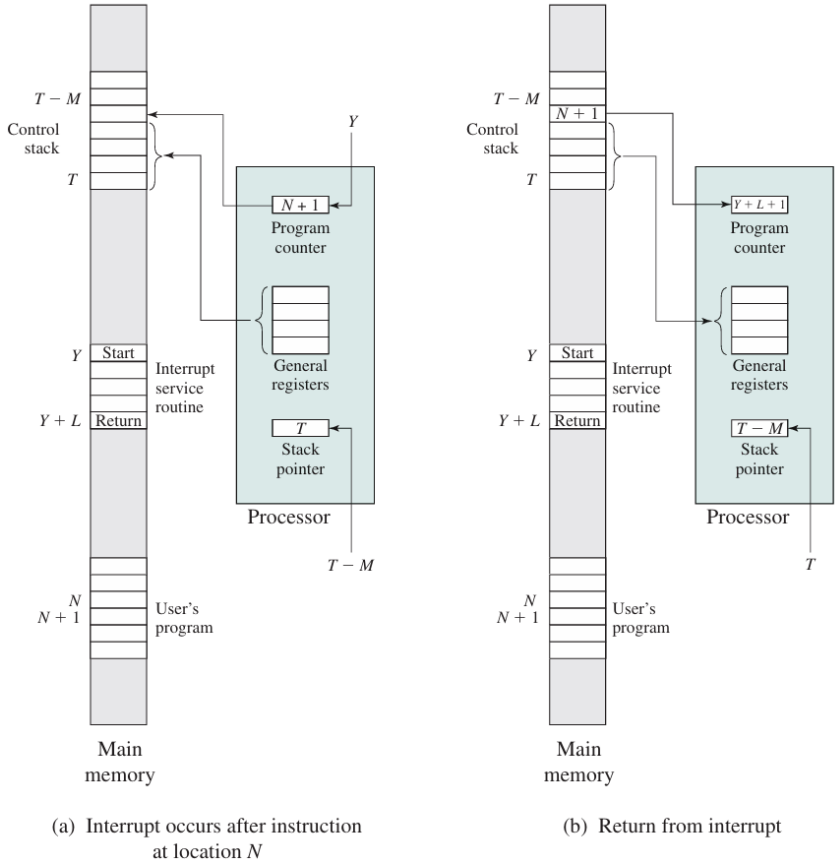

Figure 6: **Memory and Stack Changes During Interrupt Handling**

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:

- Faster access time ⇒ Higher cost per bit
- Larger capacity ⇒ Lower cost per bit
- Larger capacity ⇒ Slower 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.

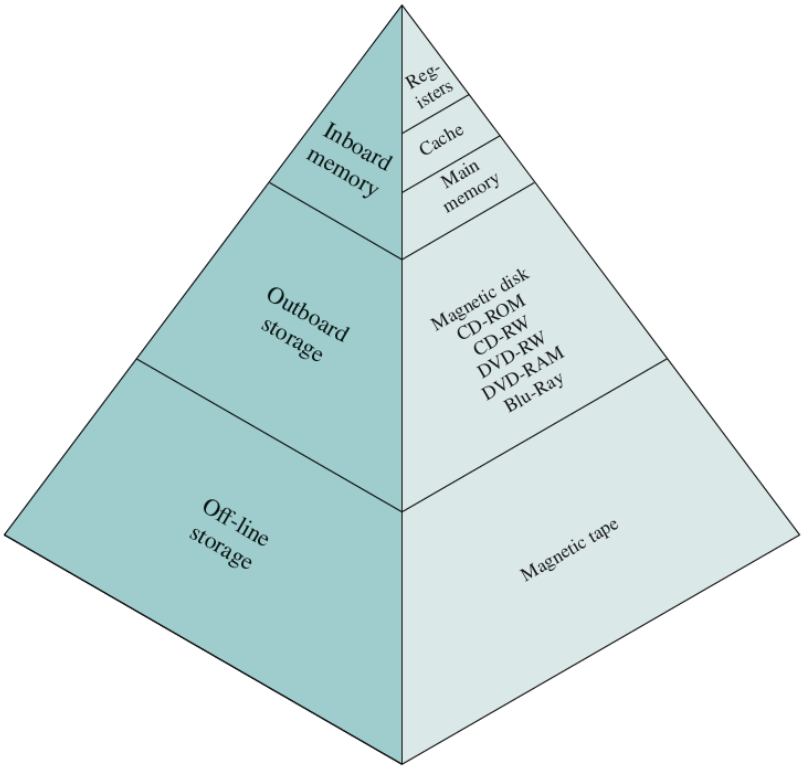

Figure 7: **The Memory Hierarchy**

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 so that data in such a way that the current cluster in use is in the fastest (the cache) memory.

Unit 1.6: Cache Memory

Although cache memory is invisible to the OS, it interacts with other memory management components. 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.