CHAPTER 2

Basic Mechanisms of Operating System

II.1. Introduction

Human has used computers to solve their problems automatically. These problems are formulated into programs, then executed by the hardware machine. For a problem to be interpreted, executed and produce the results expected by the machine's processor, it must be formulated in a language that the machine can understand, and the operating system must control its execution. In order to do this, the programmer begins by writing his problem in a structured way, in the form of an algorithm.

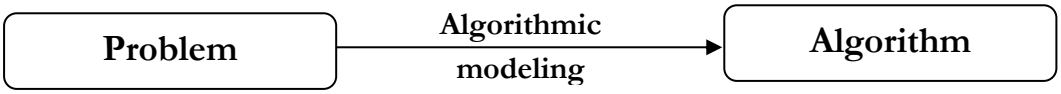


Figure 2.1 : Algorithmic Modeling

This algorithm is written in human language, and to make it executable by the computer, the programmer takes it through several stages using programming language and system tools.

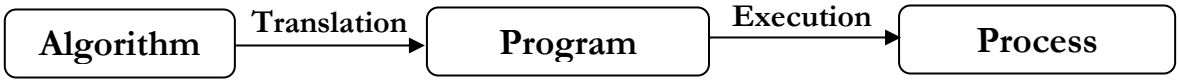


Figure 2.2 : Procedures from Algorithm to Execution

In order to understand the basic mechanisms used to execute a program, we recall a few necessary concepts of a machine's hardware architecture, then outline the various stages in a program's progress through a system, as well as the mechanisms used by interrupt systems.

II.2. Computer Architecture and Technology

The term architecture refers to the organization of system components and the relationships between them. There are two types of architecture: software and hardware. Software architecture is concerned with the organization of different programs in relation to each other. Hardware architecture concerns the organization of the various physical devices found in a computer. In terms of computer architecture and technology, we distinguish two types of HARVARD & VON NEUMANN architecture.

- In a **Von Neumann Architecture** , programs and data are stored in the same memory and managed by the same information-handling subsystem.

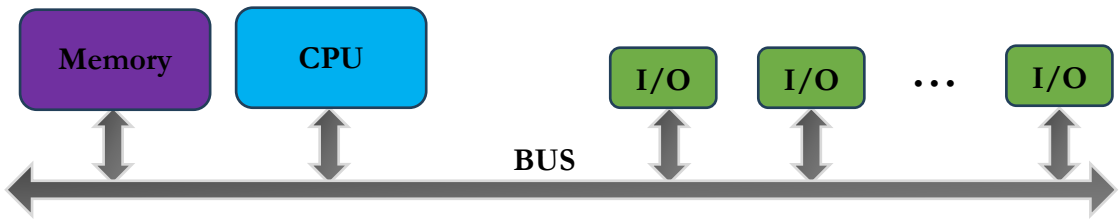


Figure 2.3 : Von Neumann Architecture

- In the **Harvard Architecture** , programs and data are stored and handled by different subsystems. This is the essential difference between these two architectures.

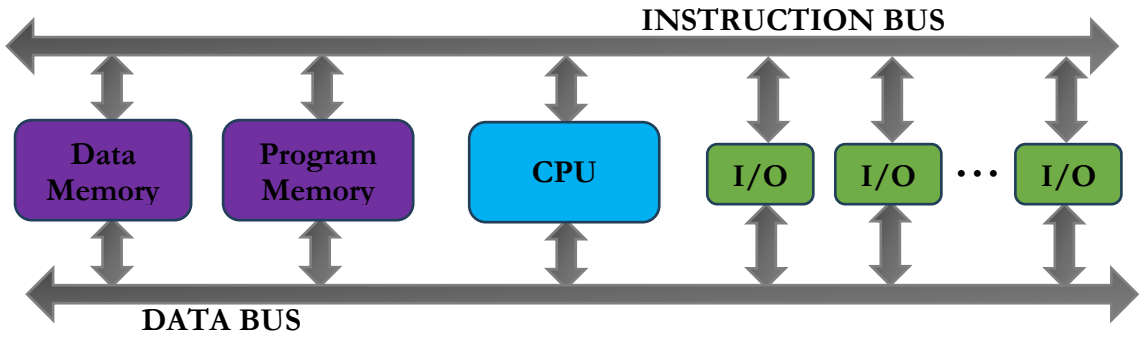


Figure 2.4 : Harvard Architecture

The best-known architecture is Von Neumann (the same architecture as our PC), while the Harvard architecture is used in devices more specific to the computing field (DSP: digital signal processor). Our studies focus on Von Neumann architecture.

II.3. VON-NEUMANN Machine

A computer is made up of a processor that carries out the processing, a central memory where the processor stores the data and results of this processing, and peripherals that enable the exchange of information with the outside world. All these components are linked together via a bus, which is the central artery and enables them to exchange data. Practically all today's computers have this architecture, from personal computers to corporate mainframes.

Definition 2.1

A Von-Neumann machine is a **memory-based electronic computer** whose components are :

- Main Memory (RAM).
- Processor or Central Processing Unit (CPU); to perform calculations and execute instructions.
- Peripheral or Input/Output (I/O) Units.

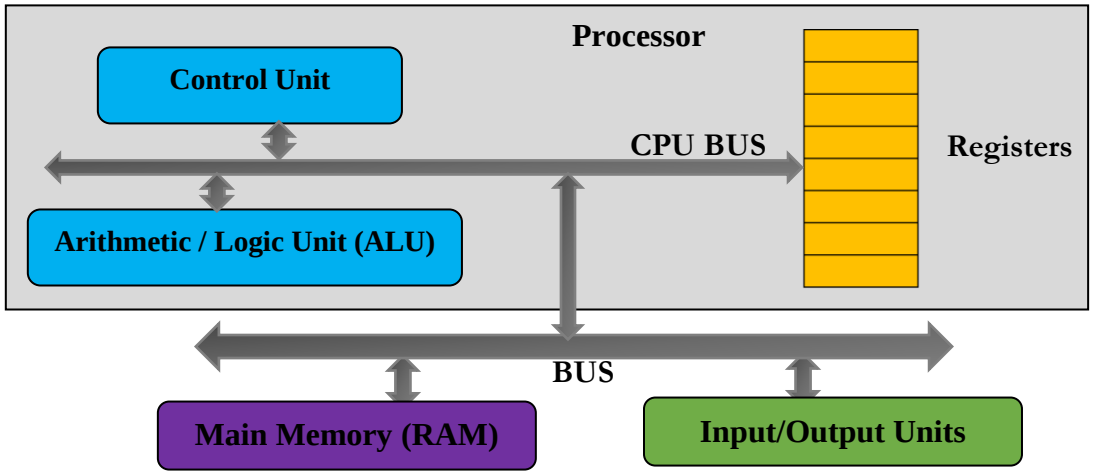


Figure 2.5 : Architecture of Computing System.

These devices implement the basic functions of a computer: data storage, data processing, data movement and control.

II.3.1. Main Memory

An addressable memory is a collection of information storage units of equal capacity known as Locations. Each Location within the memory is identified by a specific piece of information called an address, which allows it to be designated and distinguished from other locations. The interface between a processor and an addressable memory is illustrated in the figure 2.6.

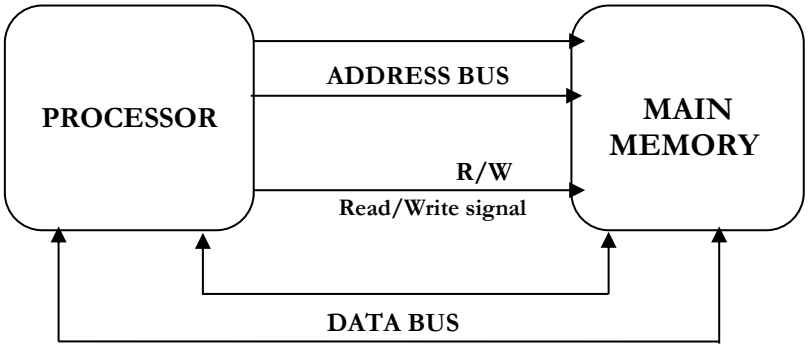


Figure 2.6 : Processor-Memory Links

Memory cycle: the minimum time between two successive memory accesses.

II.3.2. Peripheral or Input/Output (I/O) Units

The I/O unit is a device capable of transferring a quantity of information between a computer's memory (main memory) and an external data medium (peripheral). The basic principle of data exchange between two physical components is shown in the figure 2.7:

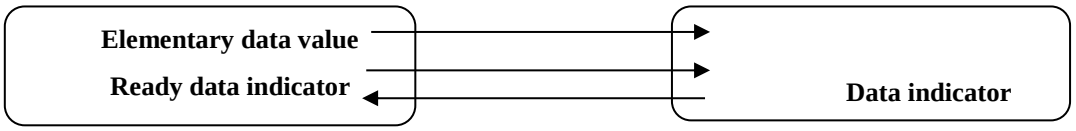


Figure 2.7 : Data Exchange Between Two Physical Components

Different input/output modes can be distinguished:

- Direct Input/Output (programmed)
- Direct Memory Access (DMA) I/O.
- I/O via specialized processor (I/O channel).

II.3.3. Central Process Unit (CPU)

Controls the operations of the computer and performs its data processing functions; often simply referred to as **Processor**. It consists of :

- **Arithmetic and Logic Unit (ALU)** : Performs the computer data processing functions.
- **Control Unit (CU):** Controls the operations of the CPU and hence the computer.
- **Registers** (Data, Addresses, Control, Instruction) : Provides storage internal to the CPU.
- **Clock :** A circuit that transmits electrical impulses at regular intervals.
- **CPU interconnection (CPU Bus):** Some mechanism that provides for communication among the control unit, ALU and registers.

The processor executes program instructions or actions. It is the computer's brain. An **action** takes the processor and its environment from an **initial state** to a **final state** in a finite amount of time. Actions are made up of **elementary instruction** sequences.

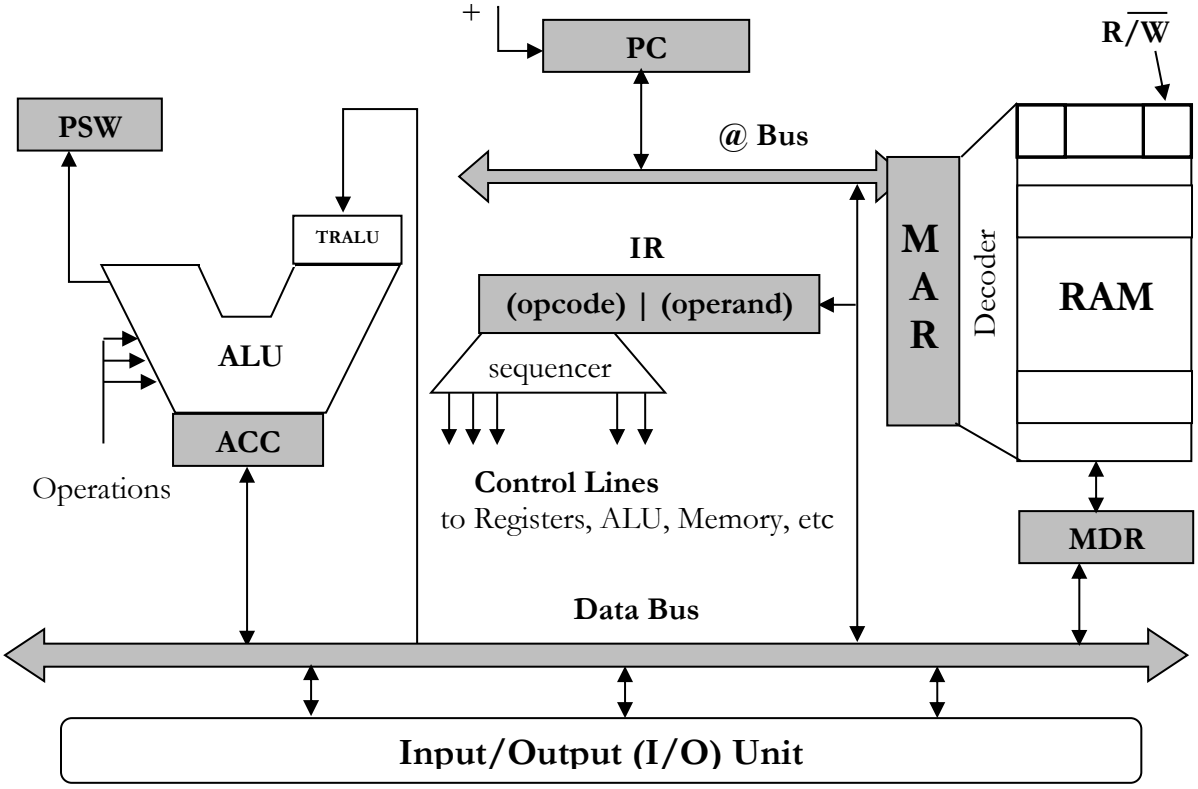


Figure 2.8 : Internal Microprocessor Architecture.

II.3.3.1. Arithmetic and Logical Unit (ALU)

- ALU Made up of circuits that performs the arithmetic and logical execution within the processor.

The results are stored in the accumulator in the CPU, it has no internal storage.

II.3.3.2. Control Unit (CU)

- The control unit fetches instructions from memory, decodes them and synchronizes the operations before sending signals to other parts of the computer.
- It is used to control the correct operation of processing and commands the flow of instructions.

II.3.3.3. Registers

- Registers are high speed temporary data storage area within the processor to support execution activities.
- Both instructions and data can be stored in registers for processing by the ALU.
- The number and type of registers implemented in a CPU are part of its architecture, and have a major influence on the machine's programming and performance.
- The exact number varies between different CPUs.

In most machines, the processor contains many registers such as:

Program Counter (PC)

- PC holds the address of the instruction to be fetched and executed.
- During the execution of an instruction, its contents are updated to point to the next instruction to be executed.
- It doesn't connect directly to the memory, but must go via the MAR.

Memory Address Register (MAR)

- MAR is used to store the address of the: next address -or- address of the currently executing instruction

Memory Data (or Buffer) Register (MDR or MBR)

- MDR stores the data that is to be sent to or fetched from memory.

Accumulator (ACC)

- ACC is used to store data that is being worked on by the ALU, and is the key register in the data section of the CPU.
- The memory can't access the ACC directly except by MDR (MDR is an intermediary).

Instruction Register (IR)

- IR retains the current instruction while it is being executed.
- When memory is read, the data first goes to the MDR. If the data is an instruction, it gets moved to the Instruction Register.
- The IR has two parts:
  1. **IR (opcode)**: The most significant bits of the instruction make up the opcode. This the genuine instruction part of the instruction, that tells the CPU what to do. The instruction in IR (opcode) gets decoded and executed by the control unit.
  2. **IR (operand)**: The least significant bits of the instruction are actually data. They get moved to IR (operand). As the name suggests they usually form all or part of an address (operand) for later use in the MAR. (However, in immediate addressing they are sent to the ACC.)

Stack Pointer (SP)

- SP contains the address of a section of memory known as **stack** which may be used for temporary storage of data or addresses.
- SP used for managing the program stack during function calls and returns.

Program Status Word (PSW)

- PSW is a register used in computer architecture to store the current status and control information related to the execution of a program.
- It typically contains various status **bits** or **flags** that reflect the outcome of arithmetic or logical operations, as well as control bits that affect the behavior of the processor during program execution.
- PSW can include information such as the current execution mode (user mode or supervisor mode), interrupt enable/disable bits, condition code flags (e.g., zero flag, carry flag), and other bits that control the execution flow of the program.
- The contents of the PSW are crucial for managing the execution of programs and handling various events that occur during the operation of a computer system.

II.3.3.4. Clock

- Pulses are sent out to the other components to coordinate their activities and ensure instructions are carried out and completed.

- The timing is controlled by a vibrating quartz crystal.
- One instruction can be carried out with each pulse of the clock, and therefore the higher the clock speed, the faster the CPU will be able to carry out the program instructions.
- The clock speed is measured in cycles per second. 1 cycle per second is a rate of 1 Hertz. Rates of 1 to 3 GHz are common in most personal computers.

II.4. Instruction Cycle (Fetch-Decode-Execute Cycle)

The "instruction cycle" refers to the sequence of steps that a CPU (Central Processing Unit) follows to process a single instruction. The instruction cycle is fundamental to the operation of a CPU and is crucial for executing programs and processing instructions in a sequential manner. The cycle ensures that instructions are fetched, decoded, and executed in the correct sequence, allowing the CPU to perform the tasks specified by the program being run.

II.4.1. Fetch

1. The program counter (PC) stores the address of the next instruction to be fetched. This value is copied to the MAR via the "address" bus.
2. The control unit then sends a signal over the control bus to read the memory. In other words, the MAR has triggered the read signal. So, the address data is transferred the contents of the selected memory cell to the MDR via the "data" bus. Simultaneously the PC increments by one (holding the address of next instruction to be executed).
3. The contents of the MDR is copied into the instruction register (IR) via the instruction bus. This consists of the instruction (opcode) and the data part (operand).

II.4.2. Decode

The next step for the CPU to interpret the data just fetched is to decode it. This instruction in the IR is split into the opcode and operand.

1. The decode analyses the instruction (opcode) in the IR and decodes it. The decode has an instruction set that defines the opcodes and so what commands to carry out. Every CPU will have its own instruction set defining legitimate commands. The opcode determines what part of the hardware is needed for execution.
2. A series of micro-signals are sent to different areas to prepare the CPU for readiness of the next step (by the Control unit).
3. The data to be operated on (operand) may be: passed onto the ALU OR the address of the data may be used with the operation which is then copied to the MAR OR the data to be operated on is copied to the MDR.

II.4.3. Execute

1. The appropriate instruction is carried out upon the data (operand) in the MDR or accumulator by the execute unit (containing an arithmetic logic unit that carries out calculations on the data). This is the executing of the instruction
2. The results of the arithmetic logic unit (ALU) are output into another register called the accumulator.

3. The control unit assigns new signals to reset the CPU to prepare for the next cycle

II.5. Boot Process

The main memory, cache memory, and internal registers of the CPU are supposed to be volatile. The information that is stored in these devices is lost when the computer's power is turned off.

From instruction cycle, we learned that a computer is a device that fetches the instruction from main memory that the PC register points to. The computer then executes the instruction. This cycle is repeated over and over again until the computer is turned off. However, if we have only volatile memories, there is no instruction in the main memory when the computer is turned on. To overcome this problem, a program is designed and stored in a special memory called the Read Only Memory (ROM) and it is installed into the computer's motherboard (sometimes called the main-board). This program is called the Basic Input Output System (BIOS). When a computer is turned on, an address is automatically loaded into the PC register. This is done by hardware circuitry. The address given is the location of the first executable instruction of the BIOS. The journey starts from there. BIOSes are produced by many factories, but perform the same basic functions.

II.5.1. Loading of Operating System

The process by which the OS gets loaded into the memory so that it can start executing is known as the System Bootup. During the boot up sequence, a series of instructions need to be executed so that at the end of this sequence the OS is running and can in turn start executing user programs. The boot sequence begins with the following:

1. The RESET pin of the CPU is set to logical high.
2. The code which is found at some specific starting address (0xfffffff0 in case of an Intel processor) is executed.
3. 0xfffffff0 maps to the persistent memory chip of the computer known as the Read Only Memory (ROM).
4. The series of instructions stored in ROM is called the Basic Input /Output System (BIOS).

II.5.2. BIOS Actions

Although the BIOS performs a number of functions e.g. making sure that all the different chips, hard drives, and the CPU function together as an entity, its most important function is **loading the Operating System**. When the computer is first powered on, the microprocessor attempts to execute the first instruction. For this purpose, as mentioned earlier the processor needs to fetch this instruction. The processor cannot fetch it from the Operating System because the OS has not been loaded yet into the memory; it is still residing on the disk. It is the BIOS which provides the processor with the first instructions to be executed in order to load the OS.

The other functions performed by the BIOS during the system bootup apart from loading the OS are summarized as:

- A power-on self-test (POST) for the different hardware components in the system to make sure everything are functioning properly.
- Activating other BIOS entities e.g. graphics cards.

- Providing a set of low-level routines to enable the OS to interface different hardware devices keyboard, screen, and the ports (serial as well parallel).

II.5.3. Fetch Execute Cycle During System Bootup

When the computer is powered on, the PC as usual points to some specific address. This pointer holds the address of the instruction to be executed. Thus, the Program Counter is initialized to some specific address like 0xfffffff0 in the case of Intel CPUs.

This address would be different for different Operating Systems. This is the address of Read Only Memory (ROM). ROM contains a special program called Boot Loader. Thus immediately, after we turn on the computer, the instructions comprising the Boot loader start executing because the PC was initialized with its address. The boot loader loads the OS from disk to the memory and then executes a JUMP instruction to that section of memory. The boot loader only loads a part of the OS and the instructions subsequent to the jump instruction read the remaining part of the OS into memory. The above process can be summarized as follows:

- The ROM contains the Basic Input Output System (BIOS) which makes a call to the boot loader program.
- The boot loader program loads the OS from disk to memory. It reads from the first sector of the disk which is termed as the Master Boot Record (MBR).
- The boot loader copies the code from the MBR to memory, starting at some specified location depending on the processor.
- The boot loader then performs a jump to that address, and subsequently the code for the OS starts executing.

The Operating system when completely loaded in the memory starts accepting and executing so that it can now in turn start executing the user instructions. It does this by continuously performing the Instruction Fetch Execute Cycle as mentioned before.

II.6. Program Execution Steps in an Operating System

Developing a program, from problem analysis to fine-tuning, requires a number of software tools that make up a programming environment. To function, these tools use the services of the operating system. The program production chain transforms a program written in a high-level language (Pascal, C, VB, Java, etc.) into a so-called executable program, written in machine language. This transformation is carried out in several steps (see Figure 2.9):

These translate the source code into machine code, while allowing the programmer to correct any errors that may have arisen at each step.

II.6.1. Text Editor (Program Editor)

This is the step in which we enter the algorithm established to solve the problem, and translate it into a program written in a High-level language (e.g. Java, Pascal, Delphi, C++, ...) or in Assembler.

The text editor is the tool that enables us to carry out this step. It's an interactive program, either associated with the system or with the programming language. It allows us to apply all the necessary

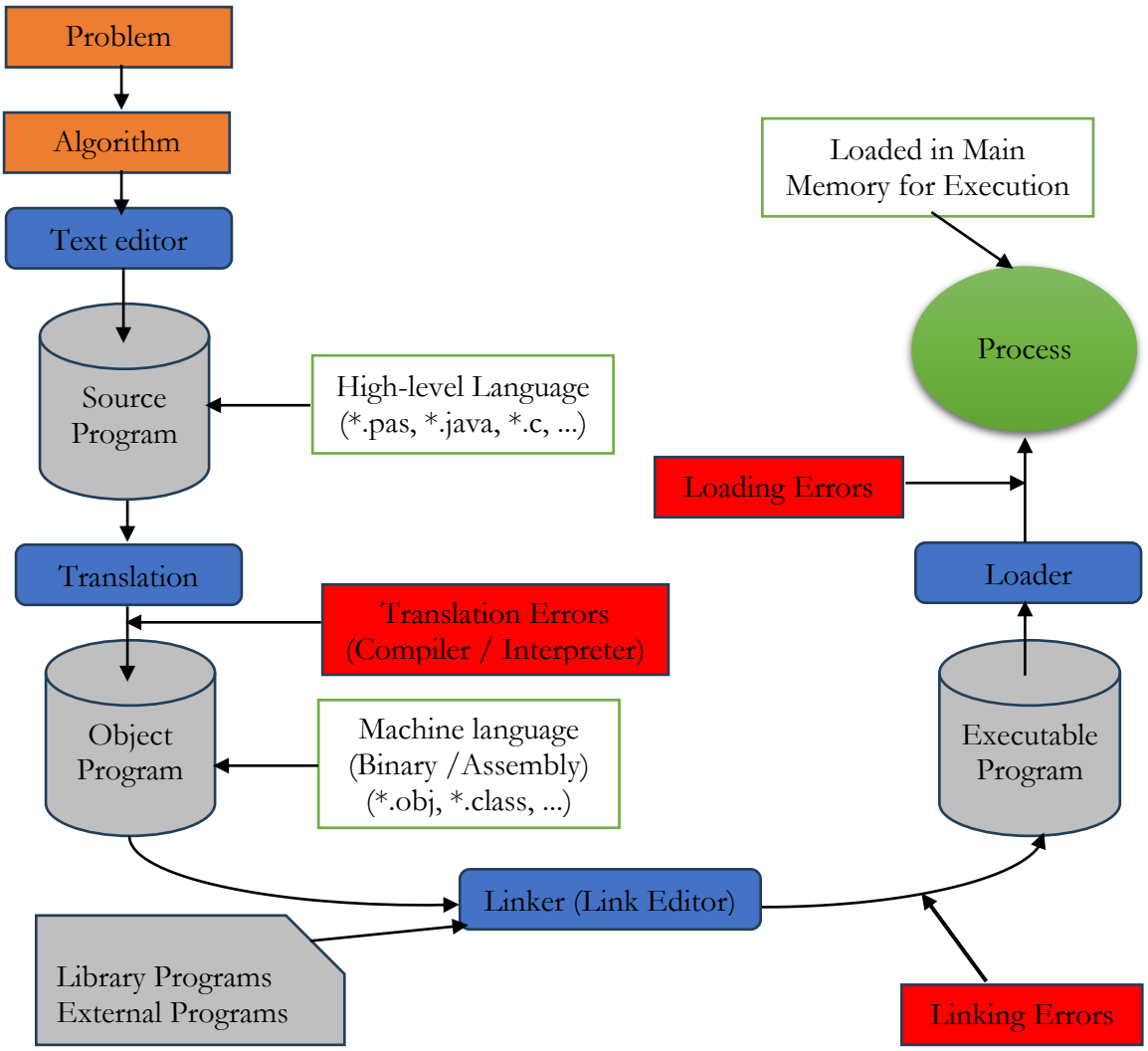


Figure 2.9 : Program Steps in an Operating System

operations to a file called "Source Code". This source code has a specific extension depending on the language used (e.g. *.pas, *.java, *.cpp, ...).

Example: Notepad, Wordpad, MS-DOS Edit, C compiler editor, JEDPlus for java, ...

II.6.2. Translator

Translator is a program that takes as input a program written in a programming language (the source language) and produces as output a program in machine language (the Object or Binary language). There are two types of translators:

- **Compiler** is a complex program, which translates a source program into an equivalent machine language (object program). The compilation process typically involves several stages such as lexical analysis, syntax analysis, semantic analysis, code generation, and code optimization. During these steps, the compiler signals errors to be corrected, which may appear in each step. The result of this phase is a file called object code, with a specific extension depending on the programming language (Example: *.obj Pascal, *.class Java, ...).
- **Interpreter** is a program that translates the source program into machine language and executes it directly, without, producing any object code (e.g. PHP, Python, ...).
  It does this by fetching the source program instructions one by one, analyzing them one by one, and then "executing" them one by one.
  - Smaller (advantage) and Slower (disadvantage)

II.6.3. Linker (Link Editor)

A linker is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file. Its aim is to complete the compiler's task by adding to the main object code all the external references. This involves carrying out the following tasks:

- Search for the origin of each external reference in all modules or libraries and insert them into the program.
- Determine the memory area that each module occupies and adjust the absolute references.
- Establish a global table of modules containing the following information: module name, size and implementation address.
- Resolve references between modules.
- Produce the executable program grouping all the modules.

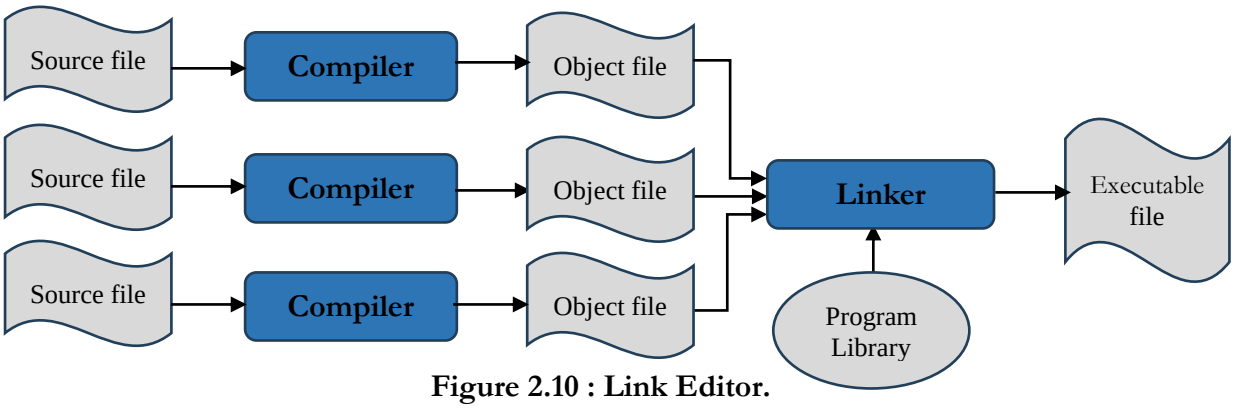


Figure 2.10 : Link Editor.

Linking can be done at compile time, i.e. when the source code is translated or, at load time, i.e. when the program is loaded into memory or, even at run time.

II.6.3.1. The Notion of Link

When translating a program into object code, the compiler associates a type with each instruction: Variable data, Constant data, Instructions, and Calls to procedures, modules or external variables.

In the last type, the programmer can reference external structures or modules in relation to the module in question (these are external references from the language library or personal references). The compiler then marks these references without bringing them back and adding them to the object code. So, it associates with each reference either :

- **EXTDEF (external definition)** - The **EXTDEF** statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.
- **EXTREF (external reference)** - The **EXTREF** statement names symbols used in this (present) control section and are defined elsewhere.

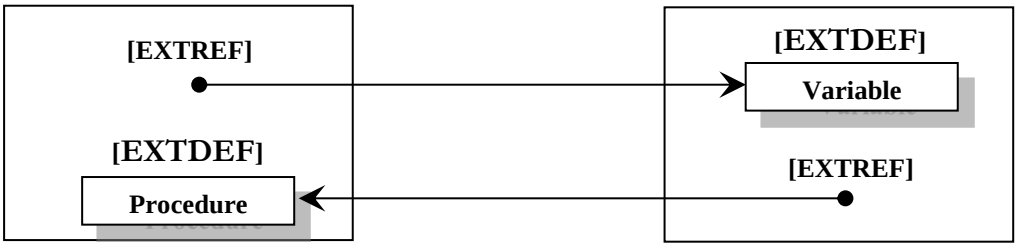


Figure 2.11 : The Notion of Links.

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

II.6.3.2. Linker Steps

Linking uses two-passes logic.

Pass 1 : The first pass consists of:

- Read all the modules and determine the location address of each of these read modules.
- Calculate the addresses of the external references of all the modules and save them in a global reference table. If a reference still remains unresolved, look for the object module that defines it in the module library.

Pass 2 : The second pass consists of:

- Read the modules one by one
- Transform absolute reference addresses according to module installation address.
- Set the address of unresolved references (using the symbol table)
- Generate the executable file.

The linker produces an executable file. Typically, this file has the same format as an object file except that it has no unresolved reference.

II.6.3.3. Linker Types

There are two types of link editors:

- **Static linker:** this is a linker that is set up once and for all, and generates a result file linking all external references with the main program, called the executable file. This type of linker does not need to be redone each time the program is run.
  Example: the Pascal language linker produces *.exe files.
- **Dynamic linked:** this is a link editor that is created each time the program is run, and does not generate an executable file. In other words, it is executed during program execution rather than during the compilation process.
  Example: The Java language linker

Example :

Consider the following two C-language programs: **Prog1.c** and **Prog2.c**

Prog1.c

```
extern int v1 ;
extern int proc1(int x) ;
float v2 ;
main()
{
      int a, x ;
      x = v1 + proc1(a) ;
}
```

Prog2.c

```
int v1 ;
extern float v2 ;
int proc1(int x) ;
main()
{
    float b ;
    b = v2 *2;
}
int proc1(int x)
{
     int y ;
     return (x * y /2) ;
}
```

In this example, we see that for the program **Prog1.c**, the **external references** are the **v1** variable and the **proc1** procedure, while the **external definition** is the variable **v2**. On the other hand, for **Prog2.c**, the **external reference** is the **v2** variable, while the **external definitions** are the **v1** variable and the **proc1** procedure.

Module	External References	External Definition
Prog1.c	Variable **v1** Procedure **proc1**	Variable **v2**
Prog2.c	Variable **v2**	Variable **v1** Procedure **proc1**

II.6.4. Loader

Once the linking operation has been completed, the program is ready to run. This phase is performed by a system tool called the loader. This step consists in reading the instructions in secondary memory, and transferring them to main memory, while reading the loading address provided by the linker at the beginning.

A loader is a program used by an operating system to load programs from a secondary to main memory so as to be executed. In this case, the program becomes a process.

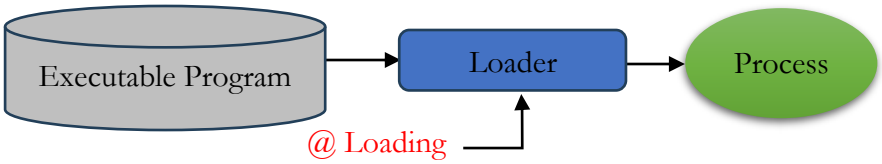


Figure 2.12 : Loading.

There are two types of loading:

- **Absolute loading:** the loaded code must not be placed in another data block other than the one indicated in the object code (in other words, the code must be copied).
- **Translatable loading:** loading can take place at any address in main memory. The way addresses are translated is based on information provided by the linker.

II.7. Process

In this part, we discuss one of the most fundamental abstractions that the OS provides to users: the process. The definition of a process, informally, is quite simple: it is a running program. The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions (and maybe some static data), waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.

Definition 2.2

A process is a running program executed by a processor. Several processes can be associated with a program. Each process has its own memory workspace, program counter and registers.

Definition 2.3

A process is a dynamic entity corresponding to the execution of a sequence of instructions: an executing program, together with its data, stack, program counter, stack pointer, resources and other register contents required for execution.

Definition 2.4

A process can be defined as a running program. In other words, a program by itself is not a process. A program is a passive entity, like the contents of a file stored on disk, whereas a process is an active entity, with an instruction counter specifying the next instruction to be executed and a set of associated resources.

II.7.1. What Happens After a Program Start-Up

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. When the user runs a program, and the machine uses the principle of virtual memory and multitasking, the program goes through the following stages:

- The system creates a **job** in virtual memory while loading the program.
- Afterwards, according to some criterion or other, this job is admitted into the system and loaded into main memory in its entirety or in part, creating what is known as a **process**.
- The process is then directed to execution in the processor according to specific conditions, where it can change its state several times during execution, until it terminates.

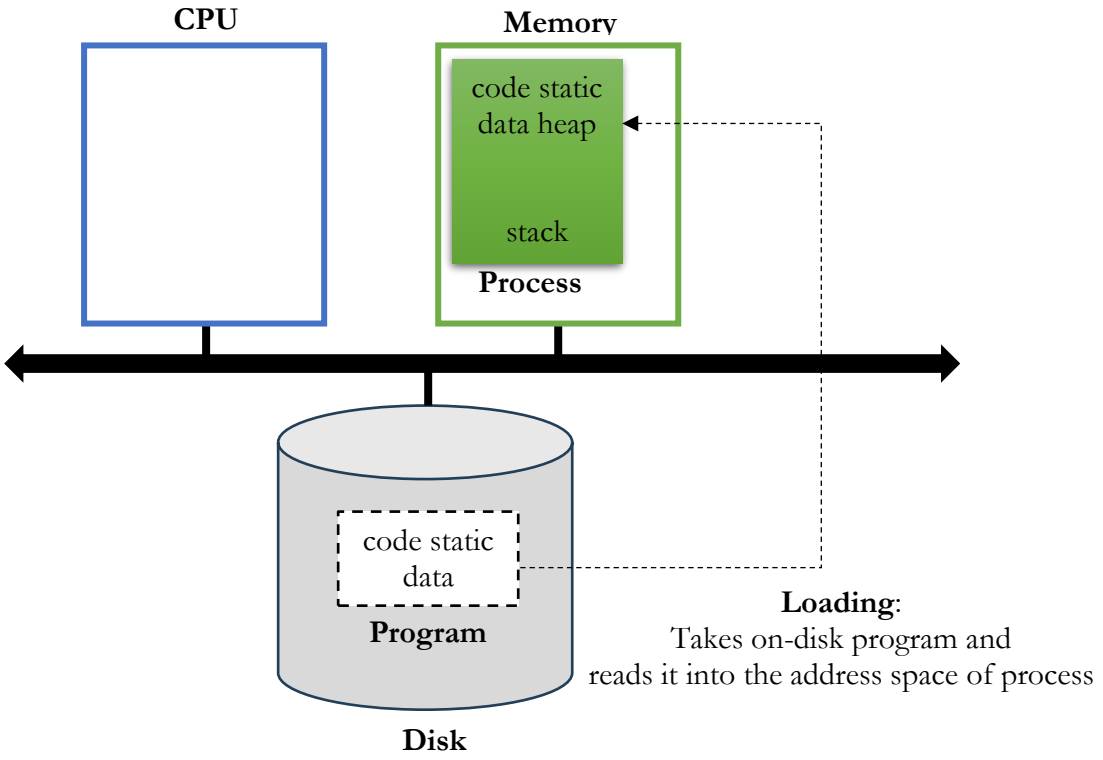


Figure 2.13 : Loading From Program To Process

II.7.2. Components of a Process

The memory space allocated to a process; known as the process memory map. It is divided into the following four sections:

1. **Stack:** Temporary data like method or function parameters, return address, and local variables are stored in the process stack.
2. **Text (Code):** This comprises the contents present in the processor's registers as well as the current activity reflected by the value of the program counter. Access to this area is read-only.

3. **Heap:** This is the memory that is dynamically allocated to a process during its execution.
4. **Data:** The global as well as static variables are included in this section.

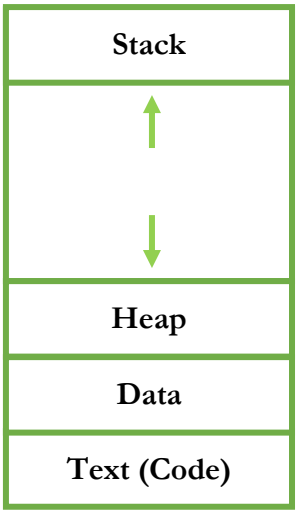


Figure 2.13 : Components of a Process

II.7.3. Characteristics of a Process

A process has the following attributes.

- **Process Id:** A unique identifier assigned by the operating system.
- **Process State:** Can be ready, running, etc.
- **CPU registers:** Like the Program Counter (CPU registers must be saved and restored when a process is swapped in and out of the CPU)
- **Accounts information:** Amount of CPU used for process execution, time limits, execution ID, etc.
- **I/O status information:** For example, devices allocated to the process, open files, etc.
- **CPU scheduling information:** For example, Priority (Different processes may have different priorities, for example, a shorter process assigned high priority in the shortest job first scheduling)

II.7.4. Process Control Block (PCB)

Every process has a process control block, which is a data structure managed by the operating system. An integer process ID (or PID) is used to identify the PCB. The PCB architecture is fully dependent on the operating system, and different operating systems may include different information. As shown in figure 2.14 below, PCB stores all of the information required to maintain track of a process.

1. **Process ID** : Each process in the OS has its own unique identifier. In other words, each process has two names for identification purposes:
   - **An external name** in the form of a character string supplied by the user (this is the name of the executable file).
   - **An internal name** in the form of an integer supplied by the system. Any reference to the process within the system is made using the internal name, for ease of handling.
2. **Process State:** The process's present state, such as whether it's ready, waiting, running, or whatever.

3. **Pointer:** It refers to a pointer that points to the parent process.
4. **Program Counter:** The program counter refers to a pointer that points to the address of the process's next instruction.
5. **CPU Registers:** Processes must be stored in various CPU registers for execution in the running state.
6. **CPU Scheduling Information:** Process priority and additional scheduling information are required for the process to be scheduled.
7. **Memory Management Information:** This includes information from the page table, memory limitations, and segment table, all of which are dependent on the amount of memory used by the OS.
8. **Accounting Information:** This comprises CPU use for process execution, time constraints, and execution ID, among other things.
9. **I/O Status Information:** This section includes a list of the process's I/O devices.

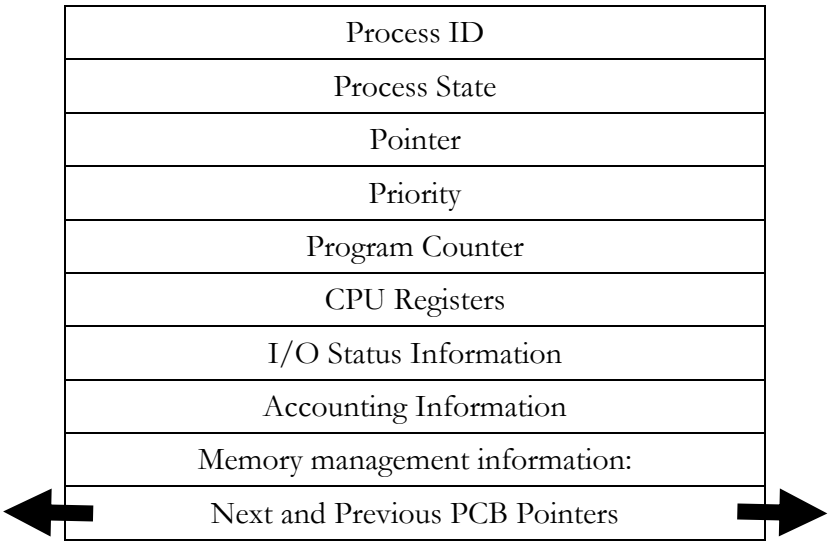


Figure 2.14 : Process Control Block (PCB).

II.7.5. Process States and State Transition Model

When a process executed, it changes the state, generally the state of process is determined by the current activity of the process. Each process may be in one of the following states:

- **New:** Newly Created Process (or) being-created process.
- **Ready:** After the creation process moves to the Ready state, i.e. the process is ready for execution.
- **Running:** Currently running process in CPU (only one process at a time can be under execution in a single processor)
- **Waiting (or Block):** The process is waiting to be assigned to a processor (When a process requests I/O access).
- **Terminated (or Complete):** The process the process has finished (completed) its execution.

Only one process can be running in any processor at any time, But many processes may be in ready and waiting states. The ready processes are loaded into a "ready queue".

The various transitions from one state to another are performed as illustrated in the diagram below.

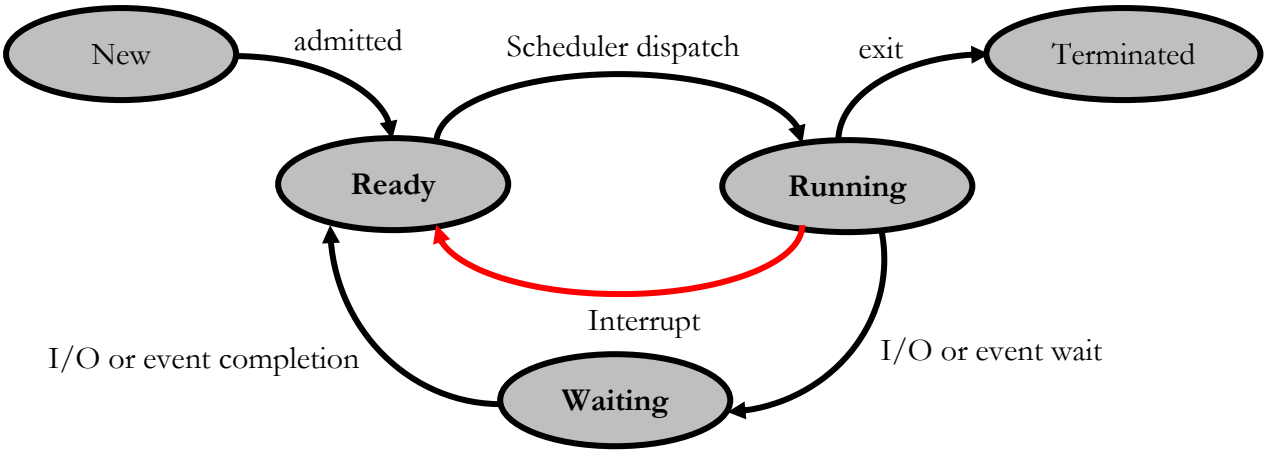


Figure 2.15 : Process States and Their Transitions

- **New → Ready :** OS creates process and prepares the process to be executed, then OS moved the process into ready queue.
- **Ready → Running :** OS selects one of the Jobs from ready Queue and move them from ready to Running.
- **Running → Terminated :** When the Execution of a process has Completed, OS terminates that process from running state. Sometimes, OS terminates the process for some other reasons including Time exceeded, memory unavailable, access violation, protection Error, I/O failure and soon.
- **Running → Waiting** : A process is put into the waiting state, if the process need an event occur (or) an I/O Device require.
- **Waiting → Ready** : A process in the waiting state is moved to ready state when the event for which it has been Completed.
- **Running → Ready :** When the time slot of the processor expired (or) If the processor received any interrupt signal, the OS shifted **Running → Ready State.**

II.7.6. Context Switching

In a multi-programmed system, the processor executes several processes in parallel (pseudo-parallelism). When the CPU is changed from one process to another, the context of the first process is saved and that of the second process is loaded into appropriate registers and other data structures. We call this context switching or process switching. Mind that process switching happens from one process to another in relation to a CPU allotment and is managed by the OS. But mode switching (user mode to kernel mode or vice versa) is essentially a processor mode activity - that happens within the context of a running process.

Who Causes The Context Switch and When?

Context switching only occurs under three events: interrupts, system calls and traps.

1. **Interrupts:** An interrupt is an asynchronous activity. That can come from :

- the timer when time slice allocated for the running process is over and another process scheduled to run next needs to get the CPU.
- I/O devices when some tasks assigned by some process to an I/O device is complete and the processor is notified. The process is to be scheduled for the CPU (either immediately or later)

as decided by the OS. If the notification comes from a device and the interrupt was not blocked, there will be a process switch (currently running process will be halted and a kernel process will start) to handle the interrupt.

2. **System calls:** It happens when the running process itself requires to execute a privileged instruction. Most system calls are for accessing hardware, like memory units or I/O devices. However, note that interrupts are caused by I/O devices to the processor, but system calls go from a running process to devices through the OS kernel. Context of the running process is saved, and a suitable kernel process is executed to meet the requirement.
3. **Trap / Exception:** When a running process encounters some errors, attempts illegal operation or to access restricted resources, traps are flagged and handled in kernel mode by kernel processes

How do Context Switches Happen?

Context switching can be described in slightly more detail as the kernel (i.e., the core of the operating system) performing the following activities with regard to processes (including threads) on the CPU:

1. **suspending** the **progression of one process** and **storing** the CPU's state (i.e., the **context**) for that process somewhere in memory,
2. **retrieving** the **context of the next process from memory** and **restoring** it in the CPU's registers and
3. returning to the location indicated by the program counter (i.e., **returning to the line of code at which the process was interrupted**) in order to resume the process.

A context switch thus takes some amount of time to complete these tasks. In comparison, processor mode switching (user to kernel or vice-versa) is less time-consuming.

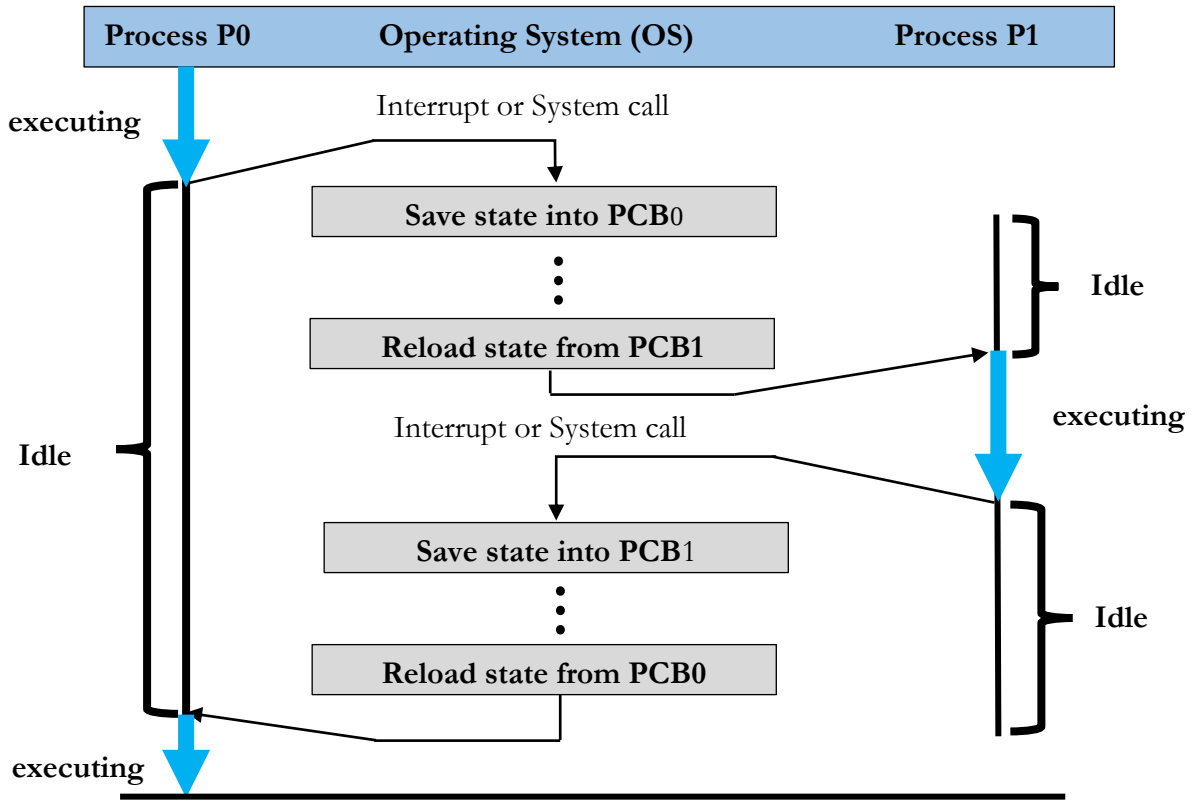


Figure 2.16 : Context Switching Mechanism

II.8. Interruption Systems

II.8.1. Issues

In a computer, two types of programs co-exist:

- **User programs**, which perform useful calculations.
- **OS programs**, which act as supervisors of all the events arriving at the machine.

These two types of programs share the machine's common resources, particularly the processor. When a program is running, several events can occur:

a) Synchronous events linked to the execution of the program in progress, such as :

1. Division by zero.
2. Execution of a non-existent or forbidden instruction.
3. Attempt to access a protected area.
4. Call to an OS function.

b) Asynchronous events not related to the execution of the current program:

1. End of I/O operation.
2. Clock signal.

These two types of events are supervised by continuous checks on their arrival.

Question: Consider a process that was performing an I/O operation and that the operation has just finished. How could the CPU know that it has finished its I/O operation, so as to schedule the remaining computational operations? How can the supervision time of the operating system be reduced?

Solution: Instead of the processor continuously monitoring the state of a resource, it's up to the resource to inform the CPU of its state at the relevant moment. This is the principle of program interrupts.

Definition 2.5

An **interrupt** is a signal triggered by a machine-internal or external event, causing a running program to stop at the end of the current operation, in favor of a higher-priority program called the interrupt program. The interrupted program then resumes execution at the point where it or another program was interrupted.

Definition 2.6

An interrupt is a mechanism by which modules (I/O, memory, processes) can interrupt normal processor processing.

An interrupt is a response to an event that interrupts the execution of programs in progress at an observable (interruptible) point on the CPU. It takes the form of a signal sent to the processor, forcing it to suspend execution of the program in progress and triggering the execution of a predefined program called an "**interrupt routine**".

Definition 2.7

Hardware point of view: An interrupt is a physical (electronic) signal sent by a peripheral to the CPU to signal the occurrence of an event.

Software point of view: An interrupt is an external event causing the temporary suspension (pre-emption) of the active process (removing the CPU from it) in favor of a special program called an interrupt program (**routine** or **handler**) in charge of the incoming event, or a call to an operating system service (system call).

II.8.2. Causes of Interrupt

There are several types of events that can cause interruptions, and they can be grouped into two main causes:

1. External or Hardware Interrupts

These are triggered by the peripherals connected to the machine following an I/O termination, an anomaly or to signal other events.

Examples include the connection or removal of a Flash disk to the USB port, the signal emitted by the keyboard following a keystroke, the signal emitted by the clock following a tick, or the signal generated by a temperature sensor in an industrial installation.

2. Internal or Software Interrupts

The main cause of this type of IT is the running process (the program being executed). There are two main causes:

a) **Trap / Exception:** this is an exception raised following an anomaly (an error) generated by the execution of the current instruction. It can be caused by :

- ✓ An arithmetic error: division by zero, overflow, etc.
- ✓ Reference to a non-existent memory address (limits of available memory).
- ✓ A memory protection violation.
- ✓ An illegal instruction
- ✓ Attempt to execute a privileged instruction in slave mode.
- ✓ Page faults, stack overflow, etc.

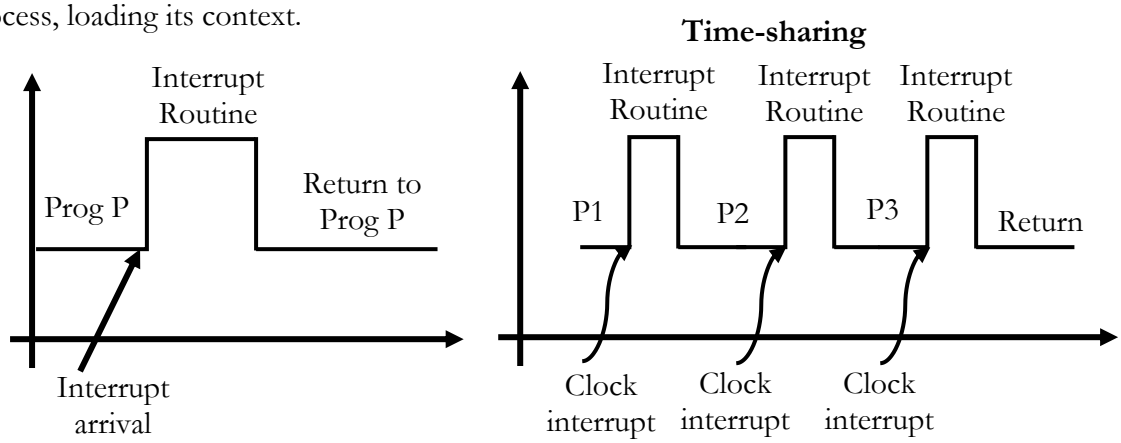
Following a trap (exception), the user will be notified of the error and the running process will be stopped immediately.

b) **Supervisor Calls (SVC):** These interrupts occur when the program issues an SVC to request a particular system service. An SVC interrupts the program being executed and passes control to the supervisor so that it can perform the service. Programs request these services through macros such as OPEN (open a file), GETMAIN (obtain storage), or WTO (write a message to the system operator).

Example:

Case 1 represents an End of I/O event, and Case 2 is a Clock Signal event, which is triggered in the case of a time-sharing system. Three processes P1, P2 and P3 are running in time-sharing mode.

Each time a clock interrupt occurs, the interrupt routine switches to the execution of another process, loading its context.



1) End of I/O **2) Clock Signal**

Figure 2.17 : Interrupt Mechanism

Remark 2.1

Some authors consider an SVC call to be a trap (exception). The main difference is that a trap expresses an **involuntary error**, whereas an SVC is an instruction **intended by the user**.
The distinction between interrupt, trap (exception) and supervisor call is based on function, not mechanism, which is the same.

II.8.3. Interrupt Handling Mechanisms

a) Interrupt Handling

When a user process **P** is running, in time **T**, and the processor receives any interrupt with higher priority, then :

1. Suspend the current **P** process and save its context in a stack:
   - ✓ address of the next instruction to be executed in the interrupted program,
   - ✓ contents of registers to be modified by the interrupt program,
   - ✓ contents of the status word (flag register) containing the flags (all this forms the saved context)
   - ✓ saving its context so that it can be executed again at a later date.
2. Identify interrupt cause: to detect the source of the IT signal.
3. Load the interrupt program context (active context) and switch to system (or supervisor) mode.
4. Execute the program to process this IT
5. Restore the context of the suspended **P** program by switching back to user mode or selecting another, and continue execution.

b) Conditions for the Arrival of an Interrupt

An interrupt can only reach the processor under the following conditions:

1. The interrupt system is active.
2. The CPU is at an observable (interruptible) point.
3. The interrupt is armed.
4. The interrupt is unmasked.
5. The interrupt has a higher priority than the current program.

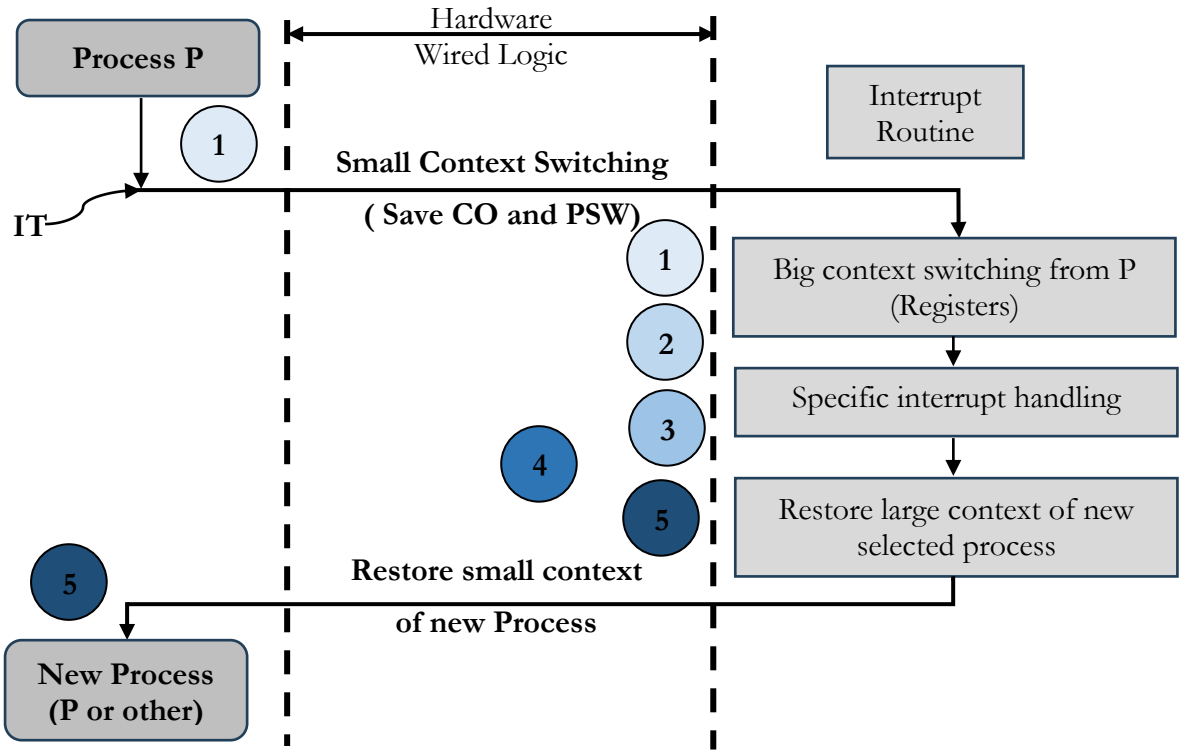


Figure 2.18 : Interrupt Handling

- **Active Interrupt System:** In some cases, the processor needs to prohibit all possible interrupts. To do this, it has a global interrupt enable/disable mechanism.
  Under these conditions, no interrupt can interrupt the CPU, and any interruption is delayed until the next time the interrupt system is activated.
- **Interrupt is Armed:** A disarmed interrupt cannot interrupt the CPU. This happens as if the cause of the interrupt had been removed. Any interrupt request made during disarming is lost. This procedure is used when you don't want an element to interrupt again.
- **Interrupt is Unmasked:** Sometimes it's useful to protect the execution of certain instructions (e.g. the interrupt programs themselves) from certain interrupts. In this case, a masked interrupt cannot interrupt the CPU, but any interrupt request made during masking is delayed (stored) for processing when the masking is lifted.

c) Interrupt Priorities

Once the signal has been detected at an **observable point**, the cause of the interruption must be determined. To do this, we use an indicator for the various causes, known as the **interruption vector**. Various methods are available for this purpose:

1. If the interrupt flag is unique, the interrupt code is stored somewhere in memory and must be read by the processing program.
2. If there are multiple flags, each is called an interrupt level. A different processing program is attached to each of these levels.
3. Both methods can be used simultaneously. Each interrupt level is accompanied by a code which is read by the processing program for that level.

Each interrupt is assigned a priority (hierarchical interrupt system), enabling interrupts to be grouped into classes. Each class is characterized by a degree of urgency in the execution of its interrupt program.

Rule: an interrupt of priority **j** has higher priority than an interrupt of level **i** if **j > i.**

The different interrupt levels can be represented in the following diagram. Priority decreases from top to bottom.

Hardware errors
Clock
Disk requests
Network requests
Terminals
Software interruptions

Figure 2.19 : The Different Levels of Interrupts

This system can be used to solve problems such as :

- ✓ arrival of several interrupt signals during the execution of an instruction,
- ✓ arrival of an interrupt signal during execution of the processing signal of a previous interrupt.

An interrupt controller can be used to group interrupt children, manage priorities between interrupts and provide address calculation elements to the processor.

d) Interrupt Hierarchy

Interrupts can be hierarchical, i.e. they can be ranked in order of priority. An interrupt handler can therefore itself be interrupted by an interrupt request with a higher priority level. It then enters the waiting state.

The figure 2.20 shows program activity over time for an 8-level hierarchical system, where level 0 has the highest priority, with level 7 corresponding to the background program.

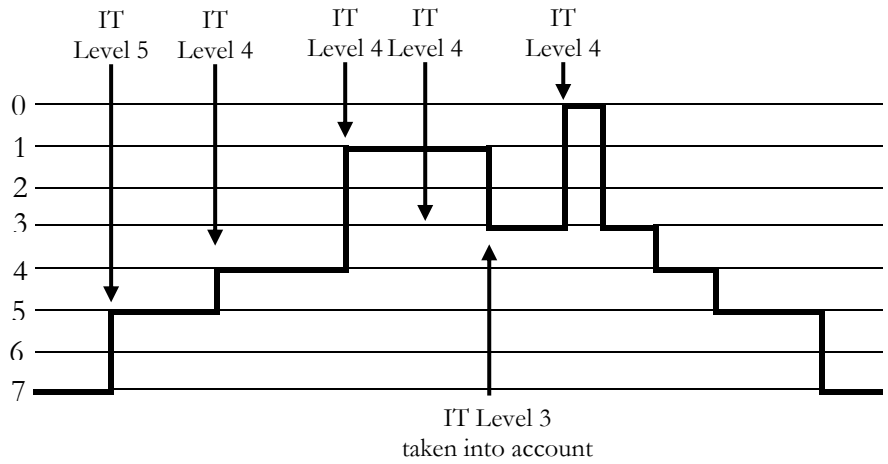


Figure 2.20 : Interrupt Hierarchy.

Also, certain operations must be atomic (they must not be interrupted), such as saving or restoring the context, which must always be completed (otherwise there is a risk of losing certain information).

e) Interrupt Operations

- ➢ **Interrupt Masking:** Some interrupts are so important that it should not be possible to interrupt their processing. In this case, other interrupts are masked to prevent them from being taken into account. Some interrupts cannot be masked: they must be taken into account. A masked interrupt is not ignored: it is taken into account as soon as it is unmasked.
- ➢ **Disarm an Interrupt:** it will be ignored. By default, all interrupts are armed.

Example

A supervisor process can mask an interrupt. Masking (inhibiting) an interrupt means delaying its effect temporarily. It can then be unmasked. At a given moment, interrupts are either masked or allowed, depending on the state of a special flag in the status register, **IF** (Interrupt Flag).

- If **IF = 1** (interrupts allowed), then the processor accepts maskable interrupt requests, i.e. processes them immediately;
- If **IF = 0** (masked interrupts), then the processor ignores these interrupts. The state of the **IF** flag can be changed using two instructions, **CLI** (CLear IF, to set IF to 0), and **STI** (SeT IF, to set IF to 1).

f) Interrupt Vector Table

The interrupt vector is a table containing the addresses of interrupt routines. Interrupts are numbered, and these numbers (ranging from 0 to 255 (FFh) on a PC) are used as an index to search the interrupt vector for the address of the routine to be executed.

The start address of an interrupt routine is called the **Interrupt Vector**. All the addresses (CS:IP logic) at the start of these vectors are stored in a table, making up the Interrupt Vector Table. This table is loaded into **RAM** from the lowest address (0000h). Each interrupt vector is stored on **4 bytes** (2 for CS and 2 for IP).

The following table shows how Intel's Pentium processor handles rerouting.

Table 2.1 : Intel Pentium Processor Interrupt Vector Table

Vector Number	Usage
0	Division by zero
1	Debugging exception
2	Non-Maskable Interrupts (NMI)
3	Breakpoint
4	Internal register overflow
5	Limit overflow
6	Invalid operation code
7	Device unavailable
8	Double precision number error
9	reserved: coprocessor management
10	invalid task status segment
11	segment absent
12	Stack error
13	Protection error
14	Page fault
15	reserved
16	Floating-point number error
17	Alignment check
18	Hardware check
19 - 31	reserved
32 - 255	Maskable interruptions