ALGERIAN DEMOCRATIC AND PEOPLE'S REPUBLIC

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY OF RELIZANE

Faculty of Science and Technology

Department of Computer Science

Course support

Specialty: Computer Science

Written By:

Dr. MEDJAHED SEYYID AHMED

Operating Systems 01

- 2025 - 2026 -

Foreword

This course is primarily designed for undergraduate students enrolled in the LMD computer science program second year, as well as for learners in professional training institutes and schools. It serves as a comprehensive resource for lectures, tutorials, and hands-on exercises, presenting the fundamentals of operating systems in a clear and accessible manner.

Developed in alignment with the updated curriculum, this course material reflects years of experience in the field. Its strength lies in its educational approach, which emphasizes clarity and simplicity through practical, easy-to-understand examples, ensuring effective knowledge transfer on the topic of operating systems.

The course content—including lessons, exercises, and practical work—is structured into five main sections. Each section is accompanied by a series of exercises.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of figures

List of figures

List of figures

List of tables

List of tables

Abbreviations

Abbreviations

OS	Operating system
IT	Interruption
OC	Ordinal Counter
CM	Central Memory
NMI	No Maskable Interrupt
SVC	Supervisor Call
FCFS	First Come First Served
SJF	Shortest Job First
SRTF	Shortest Remaining Time First
RR	Round Robin

Chapter I

Introduction to Operating Systems

Chapter I : Introduction to Operating Systems

1. Definition of an operating system

An operating system is software that facilitates and simplifies the use of a computer. It serves as the interface between the user and the hardware. In other words, the operating system is, above all, essential software for the functioning of a computer. It provides an interface that enables communication between humans and machines through various application software.

2. Roles and functions of an OS

The OS is essential to use the machine's resources, its main functions are:

- Processor management.
- Central memory management.
- Process management: calculation time between multiple programs running simultaneously.
- Input/Output management.
- File management (access, storage, etc.).
- Resource management: allocation of necessary resources and execution of programs without encroachment (creation, allocation, release, etc.).
- User management: no interference between them.
- Rights management.
- Information management.

3. Classification of operating systems

There are several families:

3.1. Single-task system (Monoprogramming, 1950)

One task, one person. In return, the user has access to all the resources and power of the machine.

Example

Dos

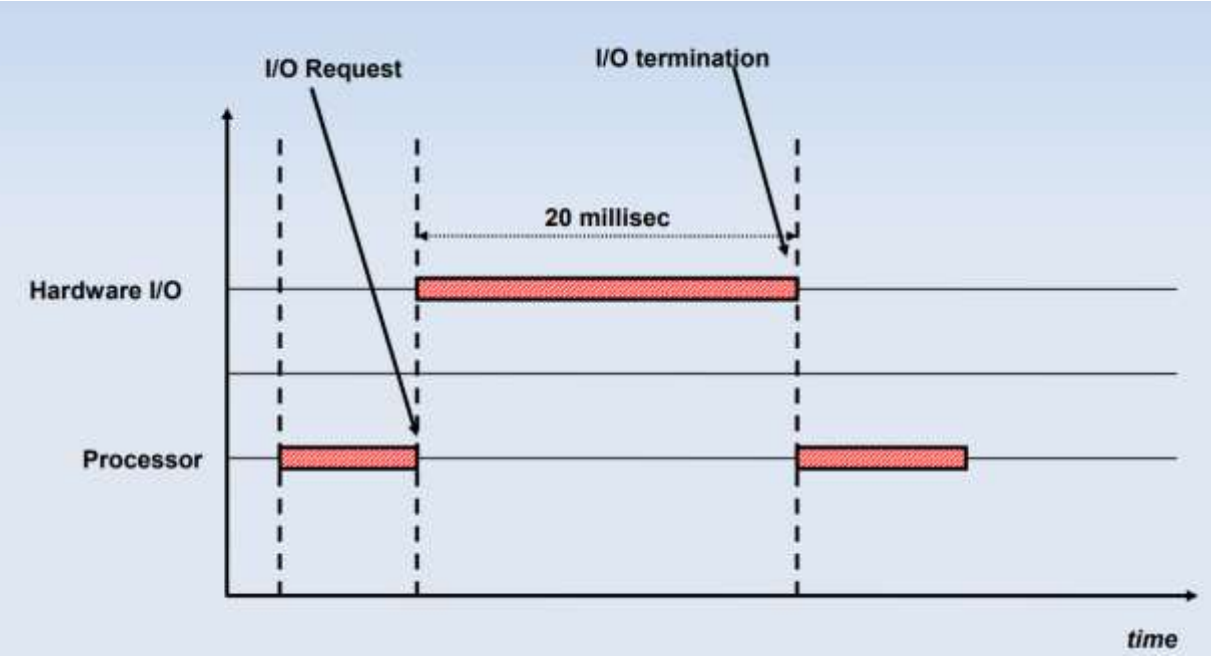


Figure 1. Monoprogramming.

3.2. Multitasking system (Multi-programming): 1960/1970

Multiple tasks run in "pseudo-parallel" (one user launches multiple applications). Tasks can run simultaneously. A task is also called a process.

Advantages

- Better CPU utilization

Disadvantage

- Still not very efficient
- Need Protection

Example

Windows, Mac OS, Linux, Solaris.

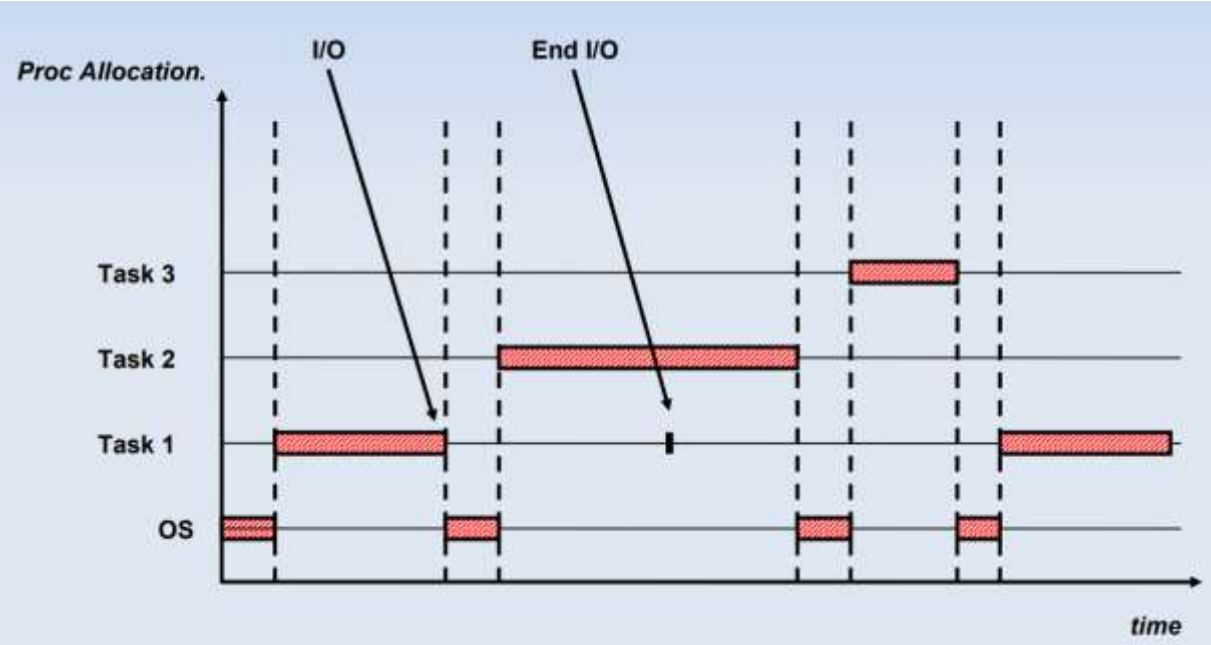


Figure 2. Multiprogramming.

In multitasking systems, we can have three systems:

- Preemptive system.
- Non-preemptive system.
- Cooperative system.

A system is said to be **preemptive** if the tasks it executes can be stopped (interrupted) at any time and then resumed in the state they were in previously. These systems have a scheduler (also called a planner).

Non-preemptive system does not stop the execution of the current task.

cooperative system is a system that operates in multitasking mode, meaning that several tasks share resources and the processor, and all must work together to execute these tasks. The operating system gives one task control of the processor. It is the task that decides when to free up the processor to allow another task to execute. Tasks can cause the system to freeze.

The difference between a **preemptive** and **cooperative system** can be summarized as follows:

"Cooperative multitasking allows multiple applications to run and occupy memory spaces, leaving it up to those applications to manage this occupation, at the risk of blocking the entire system. On the other hand, with "preemptive multitasking," the kernel always retains control (who does what, when, and how), and reserves the right to close applications that monopolize it."

3.3. Single user system

These are the systems used by a single person.

Example

Dos, Windows 3.X.

3.4. Multi-user systems

Multiple users use the same computer at the same time, while limiting each user's access rights to ensure the integrity of their data.

Example

Windows NT, Linux, Mac OS X.

3.5. Multiprocessor system

Multiprocessor systems are used in computers that have multiple processors. These systems must be able to run multiple processors in parallel.

Example

Linux (87,80%), Unix (4,60%), Windows (0,8%), BSD and Mac OS (0,2%).

3.6. Real-time operating system

RTOS stands for Real-Time Operating System is a specialized operating system designed to handle time-critical tasks. This type of system is used to control industrial processes, where meeting response time requirements is critical. These applications include embedded systems

(programmable thermostats, household appliance controllers, mobile phones), industrial robots, spacecraft, industrial control systems, and scientific research equipment.

It is used when there are fixed time constraints on a processor's operations or on data flow. It has strict and well-defined time constraints: the processing must be done within the time limit, or the system fails.

An example of real-time management is the airbag system: when a sensor detects significant deformation of the car's body, it sends a signal to the controller, which must inflate the airbags within 10 ms, or it will be too late, with potentially disastrous consequences for the car's occupants.

Example

RealTime Linux, OS-9, VxWork.

3.7. Embedded system

These are operating systems designed for use in small machines, such as: space probes, robots, vehicle on-board computers, etc.

Example

iOS (ex-iPhone OS), Windows phone, Android.

4. Comparison of some operating systems

Table I-1. Classification of OS

System	Coding	Single/Multi-user	Single/Multi-task
Back	16	Mono	Mono
Windows 3.1	16/32	Mono	Non-preemptive
Windows 95	32	Multi	Cooperative
Windows NT/2000	32	Multi	Preemptive

Windows XP	32/64	Multi	Preemptive
UNIX/LINUX	32/64	Multi	Preemptive
MAC OS X	32	Multi	preemptive

5. Composition of an operating system

An operating system is composed of:

5.1. A kernel

The kernel is the core component of an operating system (OS). It acts as a bridge between the hardware and software, managing system resources and enabling applications to interact with the hardware in a secure and efficient manner.

The kernel is the set of basic functions necessary for the machine's operation. It resides in main memory and represents the fundamental functions of the operating system, such as memory management, process management, file management, main input/output operations, and communication features.

5.1.1. Key Functions of the Kernel

Process Management, Memory Management, Device Management, File System Management, Security and Access Control, Interrupt Handling.

5.1.2. Types of Kernels

1. Monolithic Kernel

- All core services (e.g., device drivers, file system, memory management) run in a single address space.
- Examples: Linux, Unix.
- Advantages: High performance due to direct communication between components.
- Disadvantages: Bugs in one part of the kernel can crash the entire system.

2. Microkernel

- Only essential services (e.g., process and memory management) run in kernel space; other services run in user space.
- Examples: MINIX, QNX.
- Advantages: Higher security and stability due to isolation of services.
- Disadvantages: Slower performance because of frequent communication between user and kernel space.

3. Hybrid Kernel

- Combines features of monolithic and microkernels.
- Examples: Windows NT, macOS.
- Advantages: Balances performance and modularity.
- Disadvantages: More complex to design.

4. Exokernel

- Minimizes the kernel's role, providing direct hardware access to applications.
- Example: Aegis.
- Advantages: Maximized flexibility and performance.
- Disadvantages: Requires highly specialized application design.

5.1.3. How the Kernel Works

Startup

- When the computer boots, the kernel is loaded into memory.
- It initializes hardware, sets up essential processes, and prepares the system for user interactions.

System Calls

- Applications interact with the kernel using system calls (e.g., open a file, allocate memory).
- The kernel handles these requests and returns results to the application.

Resource Sharing

- The kernel ensures fair and secure sharing of system resources like CPU, memory, and I/O devices among multiple processes.

5.2. Shell

Shell = command interpreter. It allows communication with the operating system through a command language. It allows the user to control peripherals without any knowledge of the hardware it uses, physical address management, etc.

5.3. The file system

A file management system (FMS) is a way of storing and organizing information into files. It allows files to be stored in a tree structure.

5.4. Dynamic libraries (booksellers)

They group together frequently used operations, according to certain functionalities (I/O, files, etc.). These operations are available, they are called and executed by other programs.

5.5. Basic application programs

Applications and services often installed together with the OS, such as, calculator, text editor, web browser, etc.

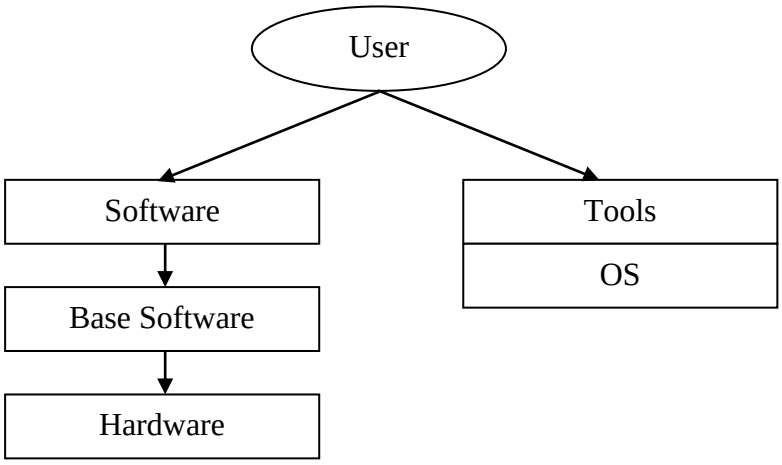


Figure I-3. Operating Systems.

6. The Virtual Machine

A Virtual Machine (VM) is a software-based simulation of a physical computer that runs an operating system (OS) and applications just like a real computer. However, instead of using physical hardware, VMs operate in a virtualized environment provided by a hypervisor or virtual machine monitor (VMM).

The goal of the virtual machine is to provide users with features tailored to their needs, it is designed to hide the physical characteristics of the hardware.

The solution it provides is designed as a layered structure, each offering increasingly advanced services towards the higher level.

Concept of virtual machine:

- A "guest" operating system no longer runs on the real machine but on a "virtual" machine.
- Virtualization software is therefore necessary.

Virtualization of resources allocated to a machine:

- Sharing processor time between virtual machines;
- A virtual disk unit: created by a “large” file;
- A basic graphics card: made by a window;

6.1. Virtual Machine – Key concepts

6.1.1. Virtualization

Virtualization is the process that enables the creation of virtual machines. It uses software to abstract and divide the physical hardware of a computer into multiple virtual machines, each acting as an independent computer system.

6.1.2. Hypervisor (Virtual Machine Monitor)

The hypervisor is a piece of software responsible for creating and managing virtual machines. It sits between the physical hardware and the virtual machines, allocating resources (such as CPU, memory, and storage) to each VM. There are two types of hypervisors:

- Type 1 (Bare-metal): Runs directly on the host's physical hardware, without an underlying operating system (e.g., VMware ESXi, Microsoft Hyper-V).
- Type 2 (Hosted): Runs on top of an existing operating system (e.g., VirtualBox, VMware Workstation).

6.1.3. Guest OS

The guest operating system is the OS installed inside the virtual machine. It behaves just like an OS running on a physical machine, allowing you to install software, manage files, and perform other tasks. The guest OS can be different from the host OS.

- Host OS: The host operating system is the OS that is running the hypervisor and providing the hardware resources to the virtual machine. It is the real, physical OS on which the hypervisor runs.
- Virtual Hardware: The virtual machine is allocated "virtual" hardware, which includes virtual CPU, RAM, hard disk, network interface, and other components. These virtual components are abstracted from the actual physical hardware.

Example

Oracle VM VirtualBox is a free virtualization software released by Oracle .

7. Layered organization of an operating system

Layers allow to clearly identify the **x levels** of interdependence between the functions provided by the OS.

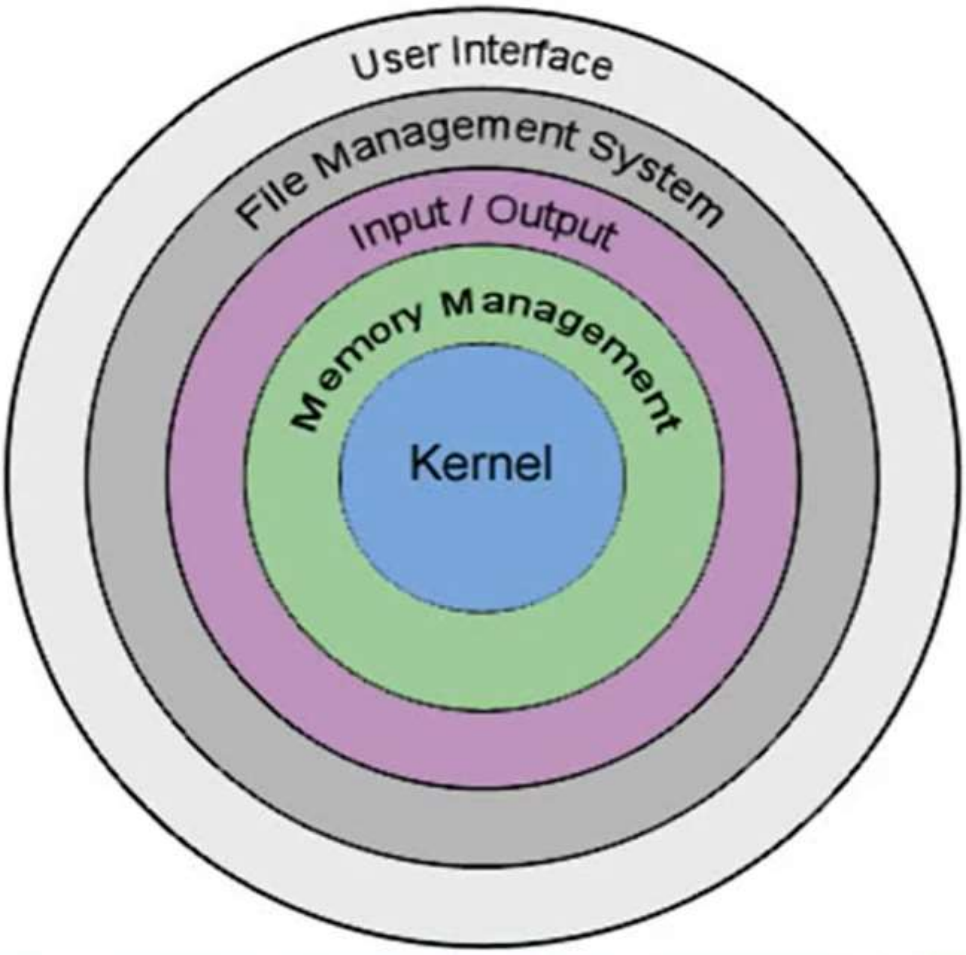


Figure 4. Layers OS.

8. Evolution of computer systems

8.1. Self-service operating systems (Open Shop)

We begin the story of operating systems in 1954 when computers had no operating systems but were operated manually by their users.

The open shop operation of IBM's first computer, the famous 701:

Each user was allocated a minimum 15-minute slot, of which time. He usually spent 10 minutes in setting up the equipment to do his computation ... By the time he got his calculation going, he may have had only 5 minutes or less of actual computation completed—wasting two thirds of his time slot.

The first generation of machines had no software. User programs were loaded into memory, executed, and debugged from a control panel.

To use the machine, the procedure was to allocate time slots directly to users, who would reserve all the machine's resources in turn for their time duration. Programs were written at that time on punched card media, and the I/O devices were the punched card reader and the printer, respectively. A control panel was used to manipulate the machine and its peripherals.

Each user, acting as an operator, had to launch a set of Operations:

- Place the program cards in the card reader.
- Initialize a card reading program.
- Start compiling the user program.
- Initialize execution of the compiled program.
- Detect errors at the desk and print the results.

8.1.1. Disadvantages

- Time wasted waiting to start running a program.
- Machine execution speed limited by the speed of the operator pressing buttons and powering peripherals.

Figure 5. Punched card.

8.2. Job Sequencer

A first improvement was to automate manual operations initiated by the operator. Thus, the sequence monitor was born, a program responsible for sequencing user work and ensuring the continuity of the sequence of read, compile, load, and execute operations. The operator's role was then reduced to loading the cards at one end of the machine and extracting the results on paper at the other.

In fact, the sequence monitor is a special program automatically activated at the end of execution of each user program, and whose purpose is to ensure the reading from memory and the launching of the next user program.

To perform this sequencing, the monitor must be able to interpret a job control language, which allows commands to be specified on special cards called control cards. A command on a control card indicates control information about the nature of the job being submitted to the machine.

8.3. Batch systems

Surely, the greatest leap of imagination in the history of operating systems was the idea that computers might be able to schedule their own workload by means of software.

The early operating systems took drastic measures to reduce idle computer time: the users were simply removed from the computer room! They were now asked to prepare their programs and data on punched cards and submit them to a computing center for execution.

The open shop had become a closed shop.

During this period, there was the evolution of computers with the appearance of transistor machines using magnetic tape units. As the concern was to reduce the time losses caused by the idleness of the processor between the execution of 2 jobs or programs, an inexpensive machine was used to read the jobs and store them on magnetic tape. The jobs stored on tape were then automatically sauntered using an enchantment monitor according to the order of the control instruction read from the cards.

The principle of job sequencing is as follows: when the monitor encounters a control card indicating the execution of a program, it loads the program and gives it control. Once completed, the program returns control to the sequencing monitor. The sequencing monitor continues with the next control card, and so on until all jobs are completed.

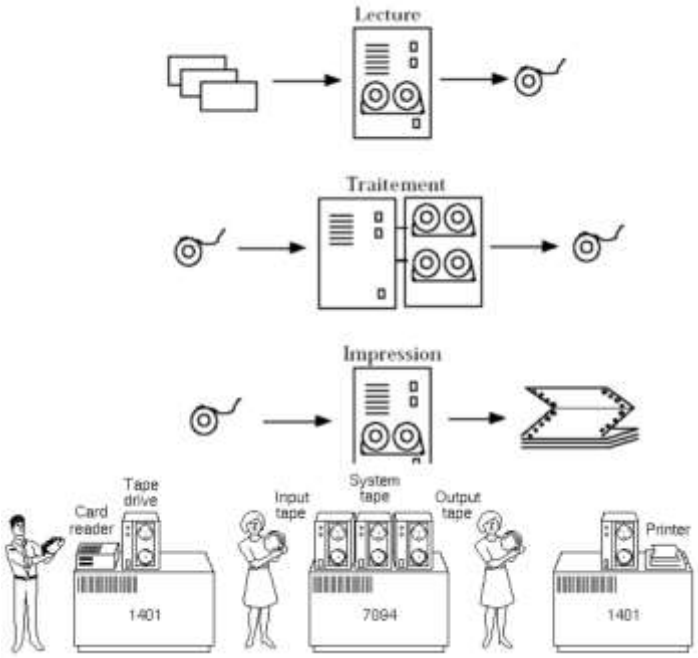


Figure 6. Batch Processing.

8.3.1. Disadvantages

wasted due to the processor being busy during I/O operations. Indeed. The processor remained idle because the speed of mechanical peripherals was slower than that of electronic devices.

In order to exploit the idle time of the processor, the idea of occupying the processor during the execution of an I/O operation by the execution of another user program (other than the one that requested the I/O) germinated. This was the beginning of parallelism (multiprogrammed systems).

Tutorial

Exercise 01

We consider a computer whose peripherals are: a card reader (1000 cards/minute) and a printer (1000 lines/minute). An average job is thus defined:

- read 300 cards,
- use the processor for one minute,
- print 500 lines.

It is assumed that all jobs submitted by users have characteristics identical to those of this average job. Two measures of system performance are defined:

- the average flow rate ***D*** of the work: number of jobs carried out in one hour.
- efficiency ***R*** : fraction of the total CPU time during which it is performing useful work (other than managing peripherals).

A - First, assume that the peripherals are managed by the central unit. Calculate ***R*** and ***D*** under the following operating assumptions:

A.1 - The system is operated with open doors; the duration of a session is limited to 15 minutes.

A.2 - The system is operated in an open-door environment; the duration of a session is limited to 15 minutes. It is assumed that a user needs 4 minutes to correct their program based on the results and make a new submission.

A.3 - The system is operated with a sequential work sequence monitor (door closed).

B - We now assume that the peripherals are managed by a separate computer, which creates an input magnetic tape from the cards and lists on a printer the contents of an output magnetic tape. The computer is powered by the input magnetic tape and produces the output tape; we neglect the time taken to read and write the tapes. The transfer time of the tapes from one computer to another is 5 minutes in each direction; we assume that a tape contains a batch of 50 jobs.

B.1 - Assume that the rate of job submission is sufficient to keep the central computer busy full time. Calculate the values of ***R*** and ***D.***

B.2 - Establish the construction schedule for the work trains and calculate the average waiting time for a user (time between submitting the work and receiving the results). We will assume that the work arrives at a regular rate, that the construction time for a batch (preparing the card train) is 10 minutes and that the time for distributing the results of a batch (cutting and sorting the listings) is also 10 minutes.

C - The peripherals are now managed by an input-output channel. The system is multi-programmed, and the chain monitor allows the central unit to execute the processing of one job in parallel with the reading of the next and the printing of the previous one. Calculate under these conditions ***R*** and ***D.*** Same question if the average job reads 1200 cards and prints 1500 lines for 1 minute of central unit time.

D - Input-output is now managed with disk buffers (read and print spools). The average job is the one defined in C (1200 cards, 1 minute and 1500 lines).

D.1 - Assume that a card and a print line occupy 80 and 100 bytes respectively. What is the minimum size of the disk read and print buffers required for the CPU to be used at its maximum efficiency? What is the job throughput?

D.2 - The job arrival rate is at saturation, the read buffer size is 576,0000 bytes, and the disk print buffer size is 2 megabytes. What is the CPU efficiency?

Exercise 02

A batch is made up of 50 works, to simplify, we assume that they all consist of 3 phases:

- card reading (20 seconds);
- calculation (15 seconds);
- print results (5 seconds).

The time taken to move from one job to another is negligible.

Calculate the total batch processing time and CPU utilization rate for the calculation in the following cases:

1- The central unit manages the input-output devices.

2- Devices are now managed by an input-output channel.

3- Devices are now managed by two channels (input and output).

Chapter II

Basic mechanisms for program execution

Chapter II : Basic mechanisms for program execution

1. Path of a program in a system

A source program is written in a high-level language. To support a source program by a machine, the following steps will be taken:

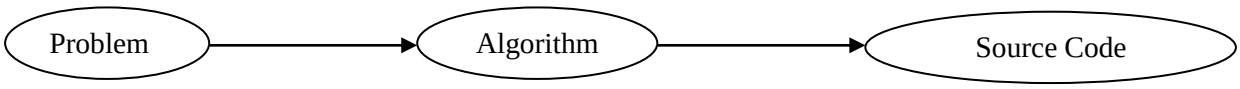


Figure 7. Prgoram.

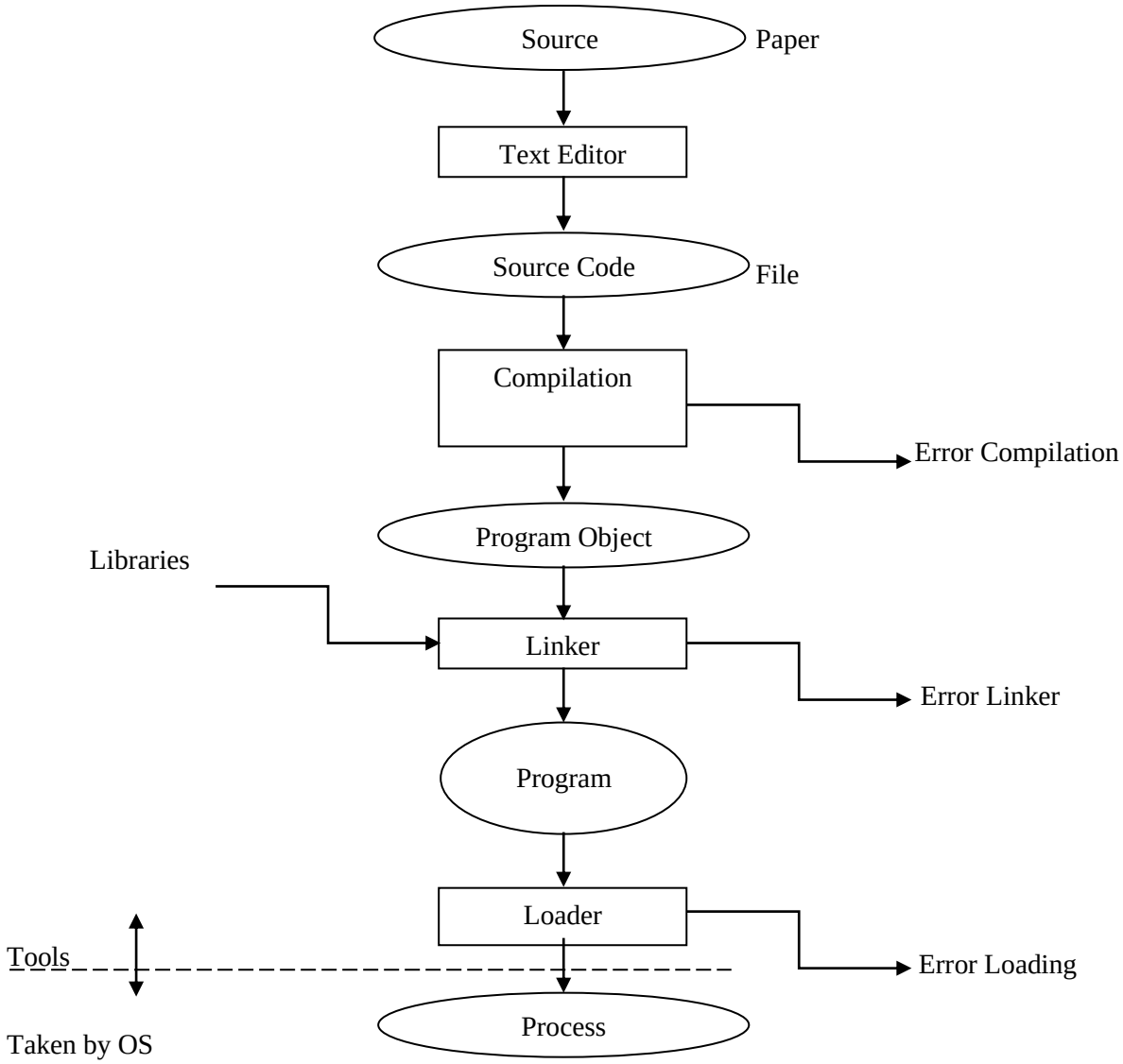


Figure 8. Program Execution.

1.1. Text editor

It is an interactive software that allows you to enter text from the keyboard and store it in a file.

1.2. Translator/Compiler

It is a program that allows you to convert a user program written in a high-level language (source program) into a program written in machine language (object program).

Translators are divided into two classes:

- Interpreters
- Compilers and assemblers

a. **Interpreter:** reads the instructions one by one and immediately converts them into machine language (the translation is redone each time the program is executed).
b. **Compiler:** translates the source program once and for all into an object program stored on disk.

1.3. Link editor

A source program can consist of:

- Instructions
- Locally defined data
- From external data
- From external procedures or subroutines

Before running the object program, it is necessary to bundle the non-local parts and external procedures with the program. A linker is therefore a software program that allows you to combine several object programs into a single one, usually called a loadable module.

Example

The complete translation of a source program requires 2 steps:

- Compilation or assembly of source procedures.
- Establishing connections between modules.

The duration of a compilation is related to the size of the source program. It is best to split it into separately compliable modules.

The linker allows you to link these modules together, link these modules and library functions to build the final executable program (loadable module).

Example

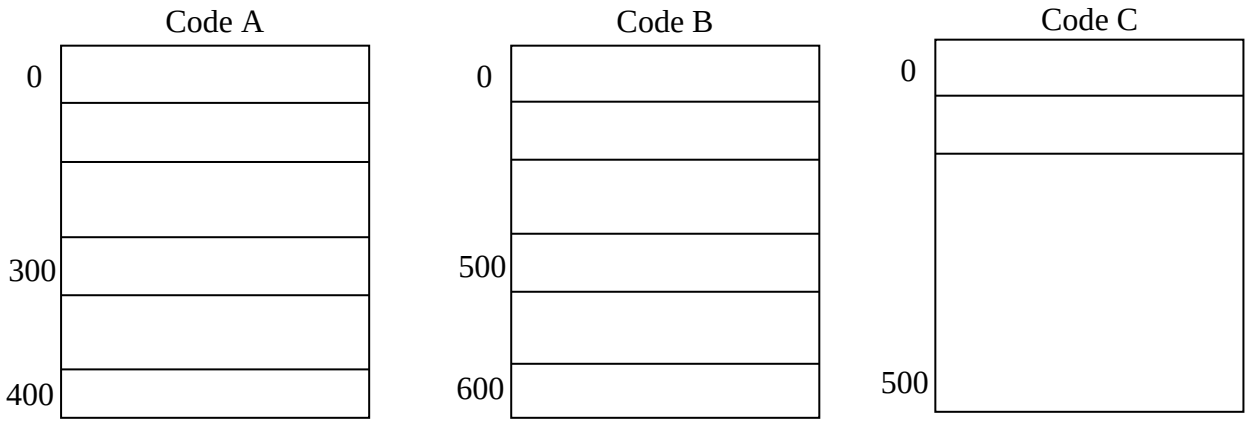


Figure 9. Linker.

After compilation the generated addresses are calculated from the value '0' each module has its own address space which starts with 0.

1- the linker must calculate the implementation addresses of these modules (the address of the 1st ^{byte} of each module).

- The location address of module A = 0
- The implementation address of module B = 400 (length of A = 400)

- The implementation address of module C = 1000 (length of B = 600)

2- the linker must establish links between the procedures (calling and called).

1.4. The charger

The loader is called when the user wants to run the program, it copies (loads) the executable program from disk to main memory (MM) from an address determined by the OS.

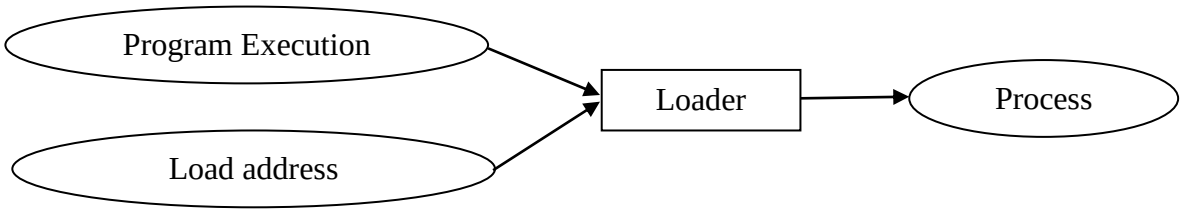


Figure 10. Charger (Loader).

The addresses of instructions and data in executable code are calculated starting from '0'. When the loader copies code from disk to the MC, it implements this code in a free space in the MC that starts with any address (other than 0) called the memory location address.

We have 2 types of charger:

a. **Static (absolute) loader:** the program address in MC is fixed.
b. **Relocatable loaders (dynamic):** A program can change memory areas while it is running. In this case, it is relocated to a new partition than the one in which it was initially loaded. The program must also be relocatable: the addresses of the program objects are logical, numbered relative to the logical address 0. Each instruction or data address is then calculated as follows:

Effective address = base address + referenced address

If we want to load a program from address 2000, the first instruction is set to 2000, and 2000 is added to all logical addresses in the program.

Program	**Logical address**
Instruction 1	Address 0
Instruction 2	Address 4
Instruction 3	Address 8

MC	**Physical address**
Instruction 1	2000
Instruction 2	2004
Instruction 3	2008

2. Process concepts and multiprogramming

2.1. The processes

A process is the unit of work of the operating system. It is represented as an action, a sequence of operations that takes place to perform a specific task. It is a running program, an active entity capable of generating events.

Computer operation is seen as a set of processes running in parallel. Concurrency is achieved by running multiple processes alternately on a single processor, which then switches between multiple processes.

A process is composed of:

- Executable code from the source program adding libraries introduced during linking.
- A context: all the resources it owns, an image of the registers, process status word, etc.

2.2. The different states of a process

From its creation until its destruction (end), a process will pass through a certain number of states.

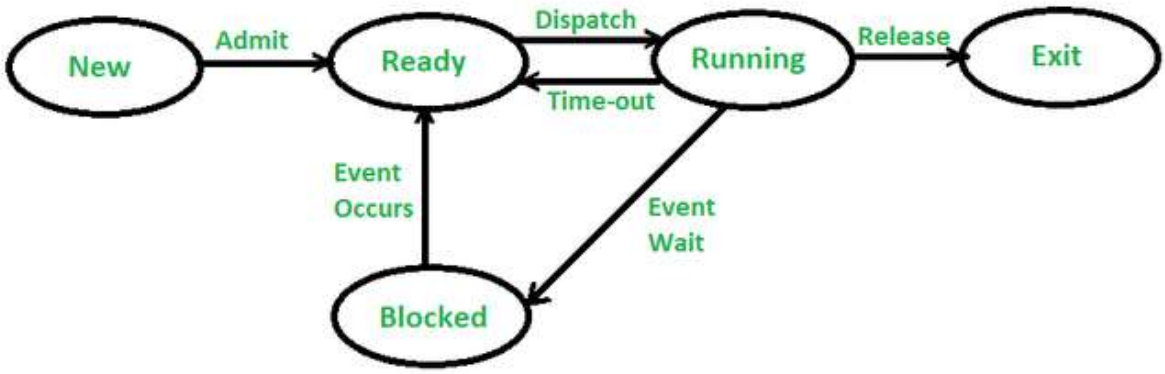


Figure 11. Process state.

A process transitions through several states during its lifecycle:

New:

The process is being created but is not yet ready for execution.

Ready:

The process is prepared to execute but is waiting for CPU scheduling.

Running:

The process is actively executing on the CPU.

Waiting (Blocked):

The process is waiting for an event, such as I/O completion or a resource becoming available.

Terminated:

The process has completed its execution or has been explicitly killed.A process can be created in the following situations:

- System startup
- System call
- Request for creation by another user

And it can be terminated in the following cases:

- Normal end
- Error
- Killed by another process

2.3. Context of a process

The operating system maintains a process table that contains an entry for each existing process. When a new process is created, the operating system:

1. Loads the code to be executed into memory (if this has not already been done for another process running the same program) and allocates a new memory location for the process's data and stack,

2. Creates the “descriptor” or “process context block” (PCB = Process Control Block) which contains all the information relating to the process,

3. Add a new entry in the process table that points to this PCB,

4. Inserts the new process into the list of its processes ready to be executed.

The operating system maintains a process table that contains an entry for each existing process. The information in this table is called a process handle or context block (PCB). Thus, for each process, the associated context block contains:

– the process memory image: these are pointers to the process’s code segments (the program), data segments (the heap) and stack

– the state of the process (ready, waiting, running, etc.)

– the identifying numbers which are used to locate it among the processes:

– the process number **(pid)**

– the number of the parent process that created the process **(ppid)**

– the process group number to which the process belongs **(pgid)**

A new process group is created when a sequence of commands linked by pipes, or a sequence of commands in parentheses, is run; the first process in a group is called the "leader process"; all processes in the group have the same gid as the pid of the leader process in the group.

– the number of the session to which the process belongs **(sid)**

A new session is created when the user logs in or opens a new terminal **(tty)** ; the first process in a session is called the “leader process”; all processes launched from a session have the same sid number as the pid number of the session's leader process.

– the numbers identifying the rights associated with the process:

– the number of the user who launched the process **(uid)** ,

By default, a process runs with the rights of the user who launched it.

– the group number of the user who launched the process **(gid)** ,

– the number of the effective user of the process **(euid)** ,

A process can temporarily run with the rights of a user (called the effective user) other than the one that launched it. To do this, the program must call the setuid(new-uid) system primitive (provided that the program owner is root or new-uid).

– the effective group number of the process user (egid), Same as euid but at the group level.

– the contents of the central unit registers, including the ordinal counter (which gives the address of the next instruction to be executed),

– the table of file descriptors associated with the process,

This table contains one entry for each file opened by the process. When the process is created, it contains three entries by default:

– file descriptor number 0 designates the standard input of the process, that is to say the flow of data communicated by the user to the process via the keyboard; this descriptor is associated with the file /dev/stdin,

– file descriptor number 1 designates the standard output of the process, that is to say the data stream communicated by the process to the user via the screen; this descriptor is associated with the file /dev/stdout,

– file descriptor number 2 designates the process error output, that is, the stream of error messages communicated by the process to the user via the screen; this descriptor is associated with the file /dev/stderr.

– information for the scheduler: process priority, recently consumed CPU time, etc.

– information for the file manager: mask for defining access rights for created files, current directory, current root, etc.

– signals sent to the process (by other processes), waiting to be processed.

3. Operations on processes

An OS must be able to perform certain operations on processes:

Create/destroy a process, block/unblock a process, suspend (transfer to MA)/reactivate (transfer to MP) a process, change the priority of a process and schedule processes.

3.1. Creating a process

A process can create one or more processes, which in turn can create others.

In some systems (MS-DOS), when a process creates a process, execution of the creating process is suspended until the created process terminates (sequential execution). In others (UNIX), the creating and created processes run concurrently.

Process creation in UNIX is performed by the **fork() system call** which clones the calling process.

CreateProcess function call (which takes 10 parameters) takes care of both creating the process and loading the appropriate program into the new process.

3.2. Destruction of a process (termination)

A process ends:

- By a voluntary shutdown request (**exit system call** under UNIX and **ExitProcess** under Windows)
- Or by a forced shutdown caused by another process (**kill system call** under UNIX and **TerminateProcess** under Windows)
- Or following an error.

If a process is killed by a **kill signal** , this signal is also sent to all its child processes.

3.3. Creating processes with fork()

In the case of Unix, the fork() system call is the only real way to create child processes:

```
#include <unistd.h>

int fork();
```

`fork()` system call creates an exact copy of the original process.

The return value of `fork()` , can be:

- 0 for the child process.
- Strictly positive for the parent process and which corresponds to the pid of the child process
- Negative if process creation failed, if there is not enough memory space, or if the maximum number of creations allowed has been reached.

After the `fork() system call` , the value of pid is set to 0 in the child process but is equal to the child process ID in the parent process.

Example

C code: p2.c

```
#include<unistd.h>

int main() {

printf("Hello %d\n",getpid());

fork();

fork();

printf("He %d:%d\n",getpid(), getppid()); //getpid gets the process's pid and getppid // gets the parent process's pid

}
```

Result

```
Machine:~/Me> ./p2

Hello 7011

He 7013:7012
```

```
He 7012:7011

He 7014:7011

He 7011:5668

Machine:~/Me>
```

Family tree (process hierarchy) generated by the program:

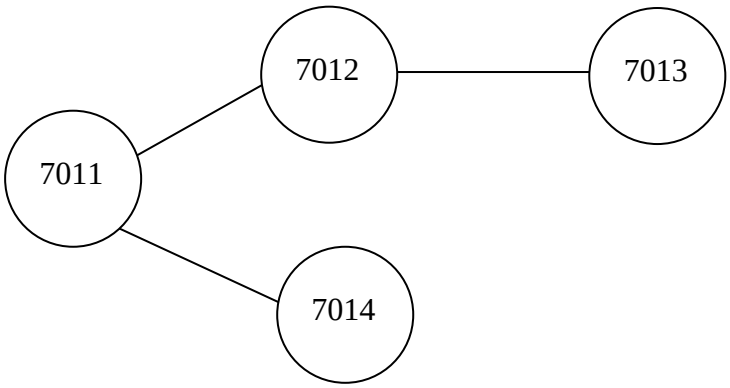


Figure 13. Family Tree.

Execution diagram

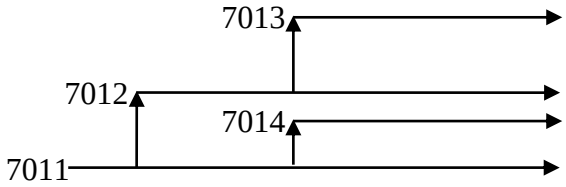


Figure 14. Schema Execution.

3.4. The exit() and wait() primitives

```
void exit(int n);
```

`exit(n)` terminates the current process with return code n (by convention, the return code after correct behavior is 0). It should be noted that a process can also be terminated by a forced termination caused by another process with the sending of a signal of the type kill().

The `wait() system call` allows the parent process to wait for one of its child processes to complete and retrieve its termination status. The syntax of the system call is:

```
#include <sys/wait.h>
pid = wait(&status);
```

pid: is the pid of the child process if the return is due to the termination of a child process; -1 in case of error.

Status: is its termination state.

A child process that terminates before the parent terminates becomes a "zombie" process until a wait is executed or the parent process terminates.

Zombie process: terminated, but parent did not call wait().

3.5. The sleep() primitive

Blocks the current process for the specified number of seconds.

3.6. The getpid() and getppid() primitives

`getpid()` system call returns the PID of the calling process. `getppid()` returns the PID of the process's parent.

4. Process hierarchy

In most systems, it is necessary to be able to dynamically create and destroy processes. When the system is initialized, all the processes that are useful and necessary are created. To achieve this, every system must provide mechanisms for creating and destroying processes. Also, any process must be able to use these mechanisms, i.e., create its own processes, which are called child processes, of which it becomes the parent, and be able to destroy them at any time.

When a process creates a child, the child can in turn create its own children. This leads to a process hierarchy (tree-like: a process has only one parent and 0 or more children).

When the UNIX system is initialized, a special process called *init* (present in the boot image) is loaded. It reads a file indicating how many terminals are present. It creates one process for each terminal. These processes wait for any connections. When a connection has been made, it launches the Shell. The Shell creates a process for each command, and so on.

So all processes are part of the same tree of which *init* is the root.

Windows does not support the concept of process hierarchy. A hierarchy can exist at process creation time: the parent process gets a special token (handle) to control the child process, and can pass it on to another process, thus invalidating the hierarchy.

In the case of figure 6, the *init process* creates the processes *user* $_1$, *user* $_2$, *user* $_3$, ... *user* $_n$. *The user* 1 process , by executing its shell, launches the vi command, creating a new process.

$_3$ process launches the vi& command, which creates a process to run the text editor asynchronously and hands over to the shell, which then launches another process to run the payroll program. This, in turn, launches the salary and accounting programs in parallel, each of which requires a process. The user $_{\text{n process}}$ launches vi and Ps.

5. Multiprogramming

In a multiprogrammed system, the processor ensures the execution of several processes in parallel (pseudo-parallel).

Switching execution from one process to another requires saving the context of the stopped process and loading that of the new process. This is called context switching.

6. Context switching mechanism

6.1. Execution mode (supervisor/user)

A program runs by default in user (slave) mode. The actions taken by the program are deliberately restricted in order to protect the machine (unprivileged mode).

The OS runs in a privileged mode (kernel mode, supervisor mode, master mode). No rights restrictions exist.

The coding of the slave world or master mode is done at the processor level (in the PSW status register: Program status word). Generally on 1 bit.

6.2. State change mechanisms

A context is the set of information accessible to the processor during processing carried out by it (execution) [set of registers].

Switching from one program to another is called context switching (it is expensive). It is done by calling (returning) procedures.

When a user program requests the execution of an OS routine (function) through a system call, this program leaves its current execution mode (user mode) to enter the system execution mode: supervisor mode

Switching from user mode to supervisor mode is a context switch. It is accompanied by a save operation of the user context (usually the CO registers and PSW status register). When the execution of the system function is completed, the program returns from the supervisor world to user mode. There is again a context switch operation with restoration of the user context (old CO, old PSW).

The context of a stopped process is generally saved in two possible types of locations:

- Fixed location: this is a location associated with a fixed memory area, specific or not to the old process, in which its context is saved. Example: memory words with addresses 0 and 2 are used for saving the context (0 for the CO and 2 for the PSW).
- Stack: The processor has a stack for saving the contexts of stopped processes.

The following figure illustrates context switching between processes. The current process is interrupted and a scheduler is possibly called.

The scheduler runs in kernel mode to be able to manipulate PCBs.

6.3. Causes causing the transition (user - supervisor)

a. Software interrupt or the user program calls a system function: this is an explicit request to switch to supervisor mode.
b. The execution by the program of an illegal operation (division by 0, memory violation, capacity overflow, etc.). This is called: a trapdoor, an exception, a dismissal.
c. Taking into account a hardware interrupt (from outside). The user program is stopped and the execution of the interrupt handling routine associated with the interrupt that occurred is executed in supervisor mode.

Example

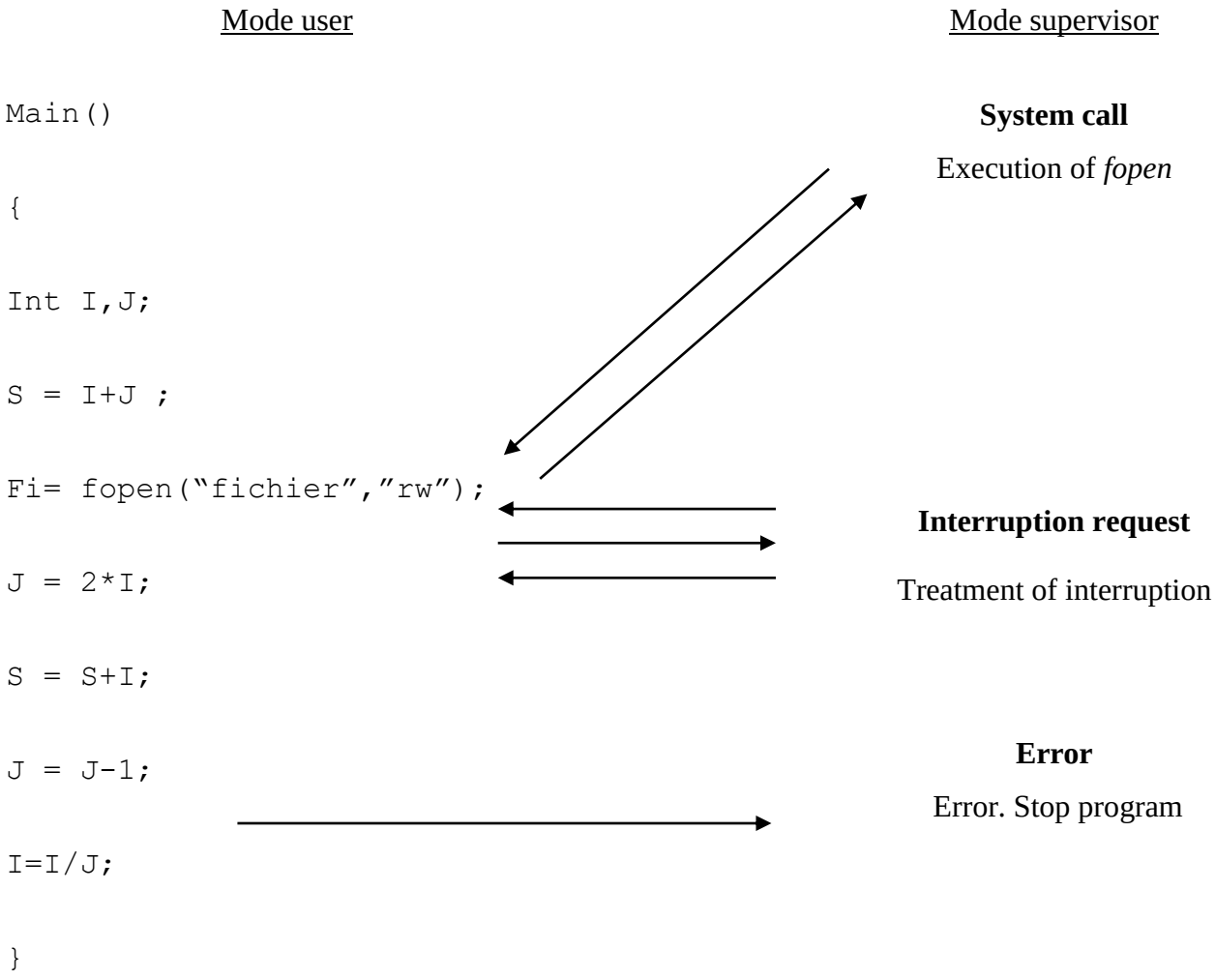


Figure 16. The 3 causes of the transition (user → supervisor).

4. Interrupt systems:

An interrupt is a temporary stopping of the normal execution of a computer program by the microprocessor in order to execute another program.

An interrupt is a response to an event that interrupts the execution of the current program at an observable (interruptible) point in the central processor.

Physically, an interrupt is a signal sent to the processor. It forces the processor to suspend execution of the current program and trigger the execution of a predefined program specific to the event, called an interrupt routine (ISR).

The interruption may occur due to:

A synchronous event (linked to program execution):

- Division by zero
- Execution of a non-existent or prohibited instruction
- Attempt to access a memory area
- A call to an OS function.

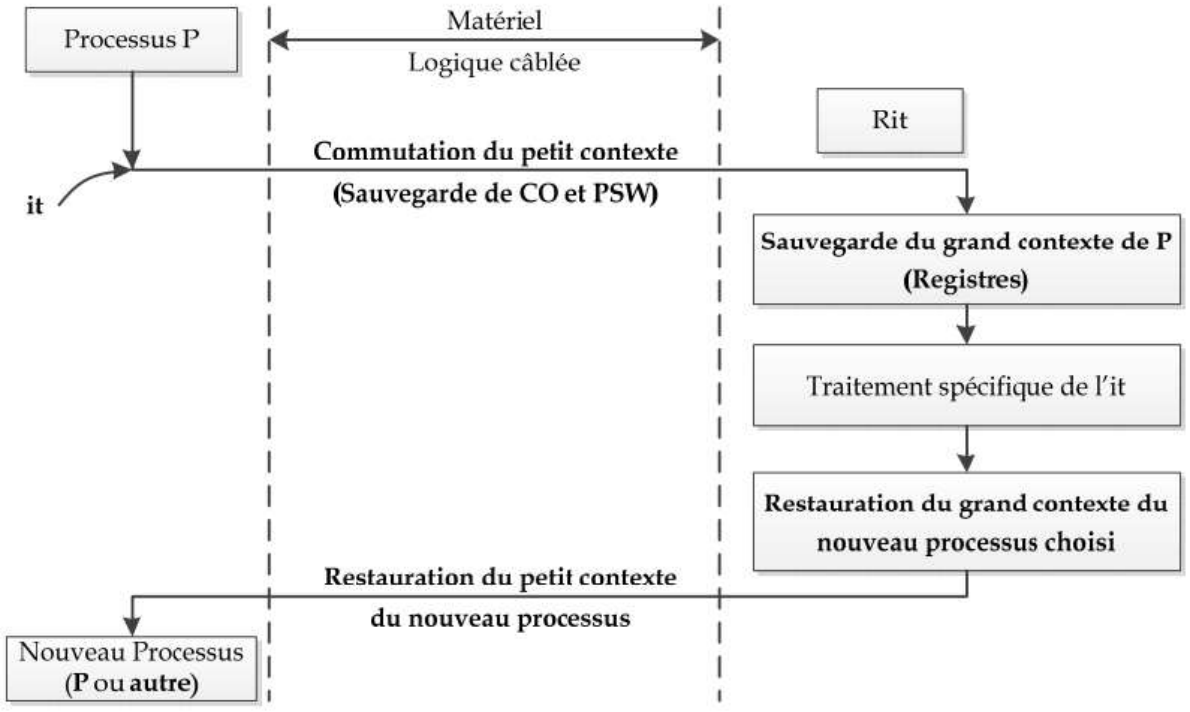
An asynchronous event (not related to program execution):

- I/O Operation
- Clock signal

5. General interrupt management mechanism:

5.1. General organization chart:

- The program is proceeding normally
- The event occurs
- The program completes the current instruction
- The program jumps to the interrupt handling address
- The program handles the interrupt
- The program jumps to the instruction following the last one executed in the main program.



5.2. Conditions for the arrival of an interruption:

An interrupt can only occur to the processor under the following conditions:

1. The interrupt system is active.

2. The CPU is at an observable (interruptible) point.

3. The interruption is armed.

4. The interruption is unmasked.

5. The interrupt has higher priority than the running program.

5.2.1. Active interrupt system:

In some cases, the processor needs to prohibit all possible interrupts. For this, it has a global interrupt enable/disable mechanism. In these conditions, no interrupt can interrupt the CPU, and any interrupt is delayed until the next activation of the interrupt system.

5.2.2. The interruption is armed:

A disarmed interrupt cannot interrupt the CPU. This occurs as if the cause of the interrupt were removed. Any interrupt requests made while it is disarmed are lost.

This method is used when you want an element to no longer interrupt.

5.2.3. The interruption is unmasked:

Sometimes it is useful to protect the execution of certain instructions (for example, the interrupt routines themselves) from being interrupted. A masked interrupt cannot then interrupt the CPU, but any interrupt requests made during the masking are delayed (stored) to be processed when the masking is lifted. Information about the masked state of interrupts is contained in the processor status word.

Masking is used to define priority rules between different interrupt causes. Thus, interrupts of the same or lower priority can be masked while a higher priority interrupt is being executed.

Noticed

Masking addresses one level or cause of interruption, unlike activation which addresses the entire interruption system.

Non -Masquable – NMI: (No maskable interrupt) An interrupt is said to be non -maskable , meaning that it must always be recognized by the microprocessor as soon as the electrical signal has been triggered. This signal is normally used to detect hardware errors (failing main memory for example).

5.3. Types of interruptions:

Interruptions can be of various origins, but they are generally classified into three main types:

5.3.1. Hardware interrupts (external): notifies the processor of external events (Clock, keyboard key pressed, etc.).

Interrupts allow the hardware to communicate with the processor. In some cases, we want the processor to react quickly to an external event: for example, the arrival of a data packet on a network connection, the typing of a character on the keyboard, the modification of the time (obviously the time changes constantly... we will see that the clock circuit, external to the processor, sends an interrupt signal at regular intervals of a few ms). Interrupts are therefore mainly used for managing the computer's peripherals.
An interrupt is signaled to the processor by an electrical signal on a special terminal. Upon receiving this signal, the processor "processes" the interrupt as soon as the instruction it was executing has finished.

5.3.2. Software (internal) interrupts: triggered following the execution of a system or user interrupt call instruction (In the 8086, software interrupts are triggered by the **INT assembler instruction. Example:** INT 21h for DOS interrupts and INT 14h for BIOS interrupts).

Supervisor Call (SVC) is an instruction allowing access to the OS service from a user program (dynamic memory allocation, file access, etc.).

5.3.3. Trap (trap or exception): following an internal processor error (overflow, division by zero, etc.).

"Generally, internal interrupts are divided into two subclasses: traps and system calls."

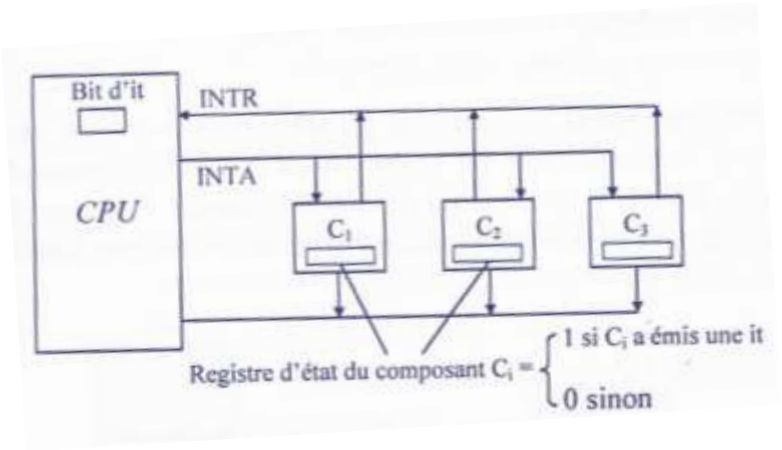
5.4. Request and Acknowledgment of Interruption

An interrupt physically corresponds to an electrical signal allowing a bit to be positioned at the processor level, called an interrupt indicator, indicating the arrival of an interrupt.

The electrical interrupt signal is emitted by one or more interrupting elements on one or more bus lines called interrupt request lines (Interrupt Request Line). If an interrupt has arrived and been accepted, the processor identifies the interrupting element and acknowledges it with another signal on a bus line called the interrupt acknowledgement line (Interrupt Acknowledge Line), then triggers the interrupt mechanism.

5.4.1. Identification by polling

In this technique, upon detection of the interrupt, a single interrupt routine is triggered. This routine will read status registers of the various interrupt causes to determine who issued the interrupt signal. Although polling is a simple way to access a peripheral, the data rate in a peripheral is sometimes much slower than the operating speed of a processor, and polling can therefore be very inefficient. The interrupt method is generally preferred.



Advantage :

- Easy to program.

Inconvenience :

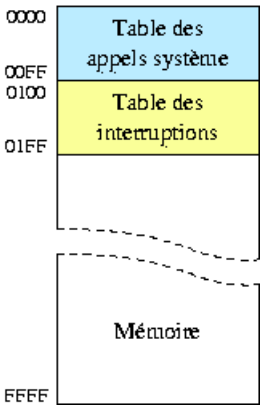
- Waste of time especially when you have a lot of devices.
- Loss of data when loading quickly.
- Waste of time when there is no new data.

5.4.2. Identification by interrupt vector:

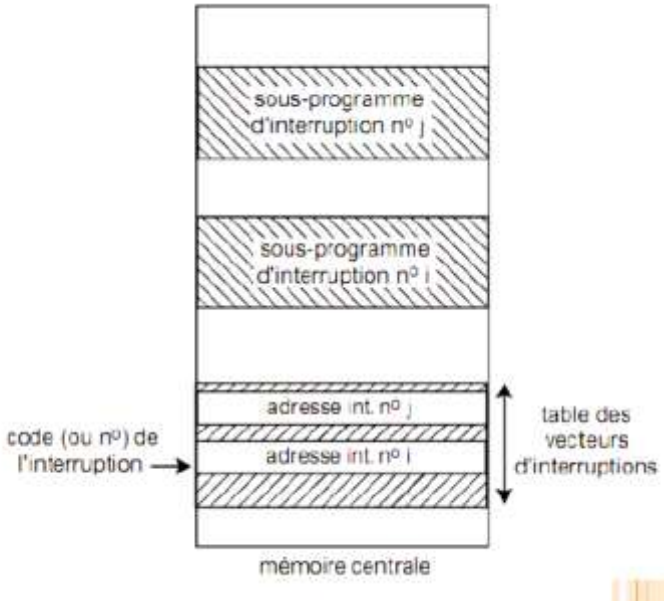
Instead of the processor scanning the components, the components identify themselves to it by sending their identity in the form of an interrupt vector.

This type of interrupt not only consists of a request signal, but also includes the identifier or vector that allows you to branch directly to the appropriate routine program.

Each interrupt is associated with the address of the corresponding interrupt routine in the interrupt vector.

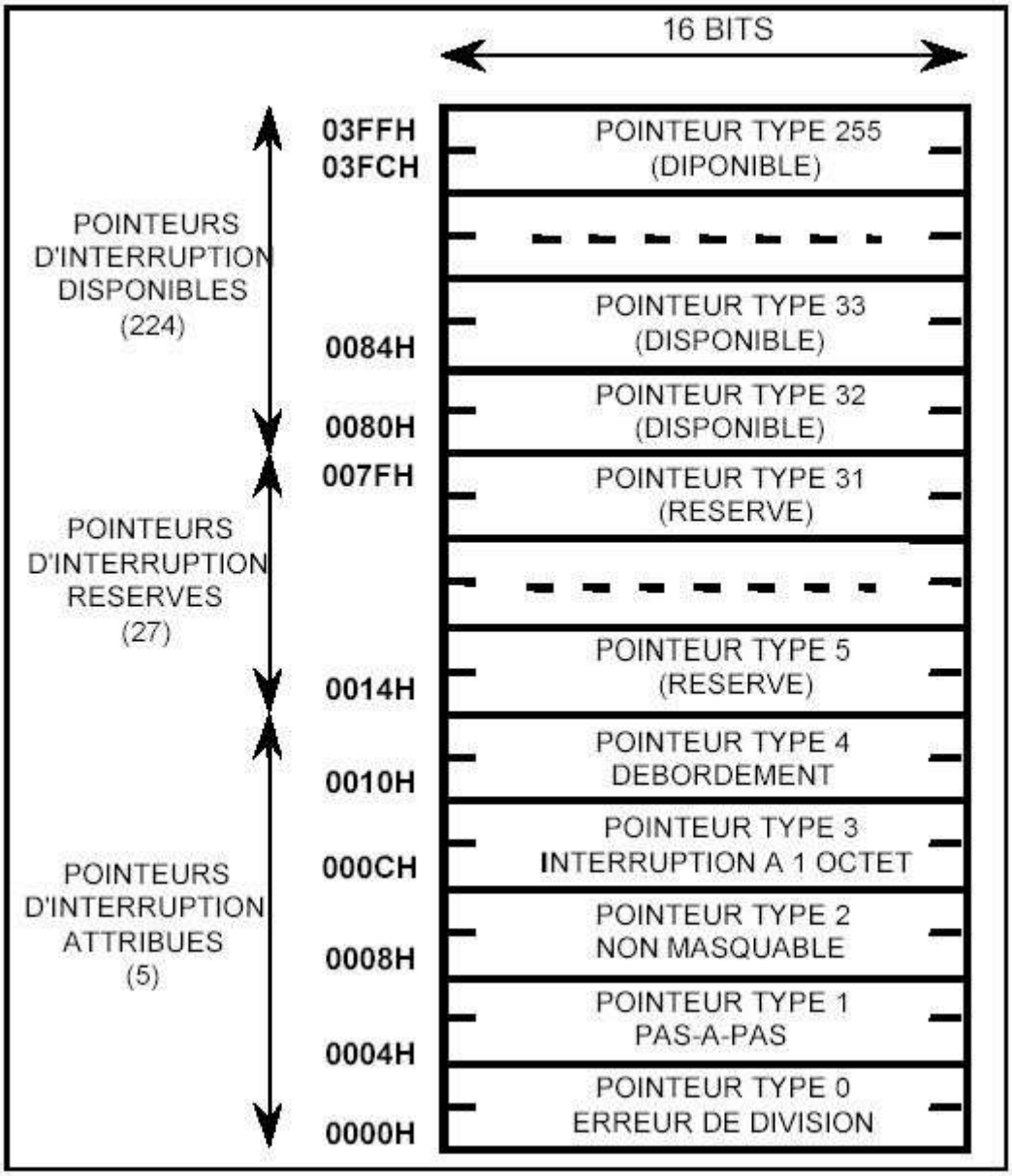


N° d'interruption	Adresse de la routine d'interruption
0	Adr0
1	Adr1
2	Adr2
3	Adr3
...	...
N	adrN



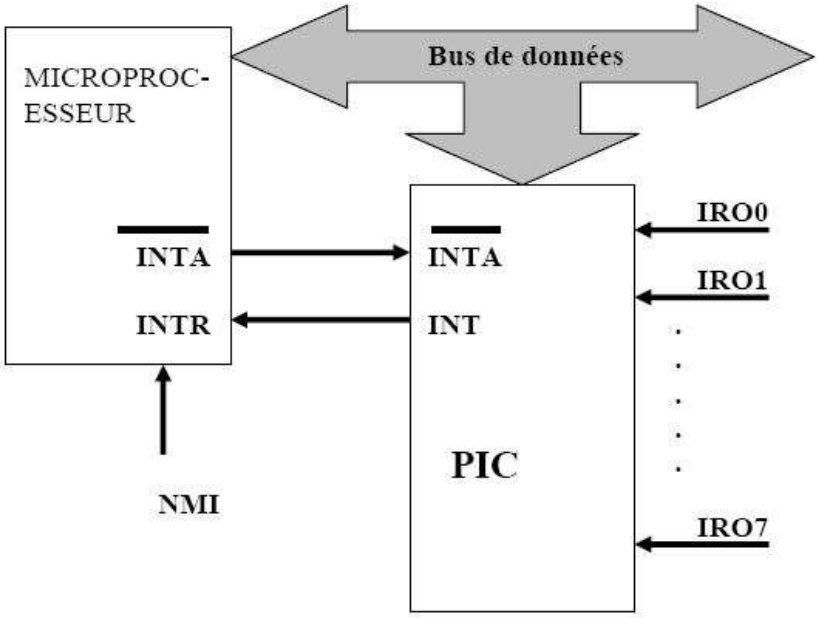
There are 16 hardware interrupts. They are called IRQ 0 to 15 (IRQ: Interrupt ReQuest).

IRQ	Interruption
IRQ0 (timer système)	08h
IRQ1 (clavier)	09h
IRQ2 (port LPT2)	0Ah
IRQ3 (port COM2)	0Bh
IRQ4 (port COM1)	0Ch
IRQ5 (disque dur)	0Dh
IRQ6 (lecteur de disquettes)	0Eh
IRQ7 (port LPT1)	0Fh
IRQ8 (horloge temps réel)	70h
IRQ9 (asservissement PIC1)	71h
IRQ10	72h
IRQ11	73h
IRQ12 (souris PS/2)	74h
IRQ13 (erreur du coprocesseur)	75h
IRQ14 (contrôleur de disques IDE 1)	76h
IRQ15 (contrôleur de disques IDE 2)	77h

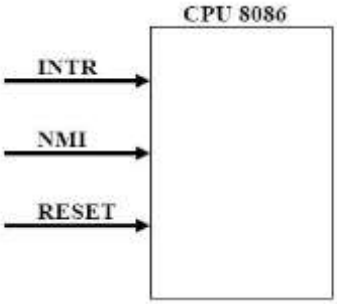


The PIC (Interrupt Controller) has a special input, usually called IRQ (Interrupt ReQuest), associated with a bit called the interrupt bit. Before moving on to the next instruction, the CPU tests the state of this bit. If it is at 1, the CPU is informed of an interrupt request. To be able to process it, it must determine its origin.

PIC example: Case of the 8086 processor.



The 8086 microprocessor has three main interrupt lines: INTR, NMI, and RESET



These three inputs allow the arrival of an external request (external interrupt).

NMI , **INTR** : Interrupt request inputs.

INTR: normal interrupt.

NMI (Non- Masquerable Interrupt): priority interrupt.

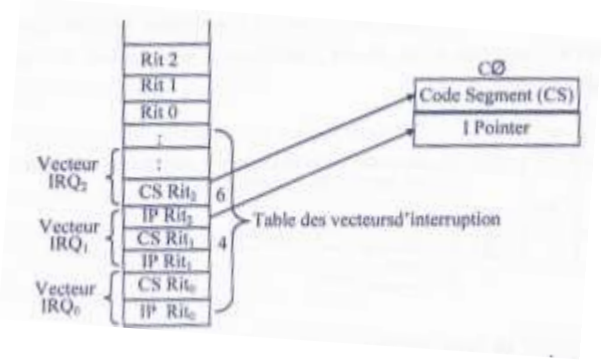
RESET : A reset signal is generated from an external RES signal that synchronizes with the CLK clock.

Example of operation:

Phase 1: The peripheral transmits an interrupt request to the interrupt controller, which forwards it to the microprocessor.

Phase 2: The processor authorizes the interrupt and the interrupt controller places the vector relating to the request on the IRQi line on the data bus .

IRQi interrupt routine .



Phase 4: The processor connects to the RTIi routine

"Each computer design has its own interrupt mechanism, but several functions are common."

6. Taking into account and processing an interruption:

6.1. Hardware interruptions:

To communicate with these peripherals, the microprocessor has three ways of communicating with them:

1. By continuously polling the device to verify that data can be read or written (Polling).
2. By interrupting it when a device is ready to read or write data (interrupt).
3. By establishing direct communication between two devices (DMA: Direct memory access).

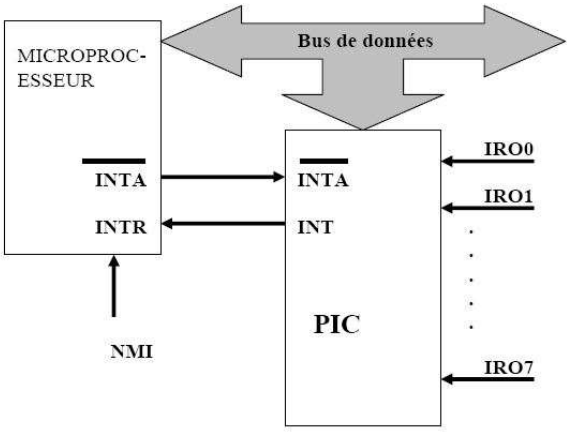
a. Device controller or any other hardware system generates an interrupt (interrupt signal **INT**). The interrupt processing consists of either:

- to ignore it and move on normally to the next instruction: this is only possible for certain interrupts, called maskable interrupts. It is indeed sometimes it is necessary to be able to ignore interrupts for a certain period of time, for example to perform very urgent processing. When the processing is finished, the processor unmasks the interrupts and then takes them into account.
- to execute an interrupt handler. An interrupt handler is a program that is automatically called when an interrupt occurs. The starting address of the handler is given by the interrupt vector table. When the interrupt handler has completed its work, it executes the special instruction IRET, which allows execution to be resumed from the point where it was interrupted.

b. Processor completes execution of the current instruction.

c. The interrupt controller receives this signal on one of its IRQi terminals. As soon as possible (depending on the other interrupts waiting to be processed), the controller sends a signal on its INT terminal.

The interrupt controller is a special circuit, external to the processor, whose role is to distribute and queue interrupt requests from various peripherals. The interrupt controller manages interrupt signals from peripherals. It can enable/disable them, change their priority, and more.



The controller is connected to the interfaces managing the peripherals by the IRQ terminals (InteRrupt reQuest). It manages interrupt requests sent by peripherals, so that they are sent one by one to the processor (via INTR). The controller can be programmed to assign different priorities to each peripheral. Before sending the next interrupt, the controller waits until it has received the INTA signal, indicating that the processor has successfully processed the current interrupt.

The PIC has a bit called **IMR** (Interrupt Masque Register) to mask interrupts.

d. The microprocessor considers the signal on its INTR terminal after completing the execution of the current instruction (which may take a few clock cycles). If the IF flag = 0, the signal is ignored, otherwise, the interrupt request is accepted.

e. If the request is accepted, the microprocessor sets its INTA output to level 0 for 2 clock cycles, to indicate to the controller that it is taking its request into account.

f. In response, the interrupt controller places the interrupt number associated with the IRQi terminal on the data bus.

g. Processor saves the resume information of the interrupted program (registers: CO, PSW). The processor sets the IF flag to 0 and searches the interrupt vector table for the address of the interrupt handler.

h. Processor loads the CO with the address of the interrupt handler (ISR).

i. Interruption handling.

j. Restoration of resume information from the interrupted program.

k. Restoration of CO, PSW backup registers.

l. The restitution of the CO and PSW values is carried out following the execution of the end of the RTI or IRET interrupt routine.

6.1.1. Simultaneous interruptions:

Most systems include peripheral devices that operate simultaneously. When an interrupt routine is running for one device, a new IT request arises from another device. There are two approaches to addressing this problem:

1. Disallow the full interrupt capture while an interrupt is being executed.
2. Set priorities.

6.2. Software interrupts:

Software interrupts are similar to hardware interrupts. The only difference is that software interrupts are issued by programs. These interrupts have well-defined functions, such as reading and writing to disk, writing data to the screen, etc. Software interrupts cannot be disabled or masked.

Software interrupts have a specific number that allow you to call functions of the operating system or the Bios.

A software interrupt is a stoppage of program execution to execute a DOS or BIOS interrupt routine.

For the 8086, software interrupts are caused by the INT instruction followed by the interrupt number.

- INT 9pm: for DOS interrupts
- INT 2 p.m.: for BIOS interrupts

How a software interrupt works:

1. The processor receives the INT instruction followed by the interrupt number

2. The processor uses the interrupt number to find the interrupt vector

3. The processor sets the IF=0 flag (IF is the interrupt flag), saves the status register, CS and IP registers to the stack, and loads the interrupt vector into CS:IP

4. The processor executes the interrupt routine which ends with the IRET instruction

5. The processor restores the status registers and the CS and IP registers, and resumes execution of the current program.

6.2.1. Examples of using software interrupts:

`INT 13h` : This function provides disk and floppy disk access functions.

`Int 1Ah` : This function allows you to read the time from the battery-powered real-time clock (BIOS Clock).

`Int 21h` : This function outputs characters to the standard output device or reads them from the standard input device.

`Int 33h` : Display the mouse cursor on the screen.

Chapter III

Central Processor Management

Chapter III : Central Processor Management

1. Introduction

The concept of a process is the most important one in an operating system. It is an abstraction of a running program. The design and implementation of a system is based on this concept.

Once a program has been transformed into executable code, it must be loaded and executed. It is said to be transformed into a process because operating systems do not recognize any executable entities other than processes. A process is an entity manipulated by any operating system.

2. Scheduling

In a system, multiple processes may be present in memory waiting to be executed. If multiple processes are ready, the operating system must manage the allocation of the processor to the different processes to be executed. This task is performed by the scheduler.

The scheduler is the component of the operating system kernel that determines the order in which processes are executed on a computer's processors. The scheduler is called a scheduler.

2.1. Types of scheduling

Three types of scheduling can be distinguished: long-term, medium-term and short-term. Their main functions are as follows:

- Long-term scheduling (admission scheduler-job scheduler): The scheduler selects programs to be admitted into the system for processing. Admitted programs become ready processes. Admission depends on the system capacity (degree of multiprogramming).
- Medium-term scheduling (memory scheduler): the scheduler selects processes already admitted to be unloaded or reloaded from memory (disk/memory transfer). It is based on the degree of multiprogramming. This type is often included in the long-term type.
- Short-term scheduling (CPT scheduler) : The scheduler selects the next process to be executed from among the ready processes, based on a certain policy. It also performs the context switch of the processes (dispatcher). The short-term scheduler's task is to manage the queue of ready processes.

In a system, processes (threads) are ordered into lists. Those waiting for memory are on a list in secondary memory. The others are divided into classes according to different states in the diagram in the following figure.

2.2. The dispatcher

It is a component involved in processor scheduling. It gives control of the CPU to the process elected by the schedule. Its function mainly includes context switching.

2.3. Scheduler Objectives

Some goals of a scheduler depend on the environment (batch, interactive, etc.), but others are desirable in all cases.
These include, among others:

- **Fairness:** Allocate each process fair processor time.
- **Response time:** minimize response time.
- **Processing capacity:** optimize performance (Throughput).
- **Efficiency:** Use the processor at 100%.
- **Be predictable:** avoid quantity degradation in multimedia systems

The best scheduling algorithm is the one that optimally meets these constraints.

3. Scheduling algorithms

There are two families of algorithms that involve or do not involve preemption (requisition) of the processor.

- Non-preemptive scheduling: The processor is freed for a new process only when the currently running process blocks or terminates. Non-preemptive algorithms are: FCFS (First-Come First Served) and SJF (Shortest Job First).
- Preemptive scheduling: Freeing up the processor for a new process can be done following an OS decision. The OS decides to suspend the currently running process for another (memo if it is not blocked or terminated) to make room for another. Preemptive algorithms are: SRTF (Shortest Remaining Time First), Round Robin, priority scheduling.

3.1. FCFS

FCFS (First Come First Served) uses a simple queue (FIFO).

In the following examples we will calculate:

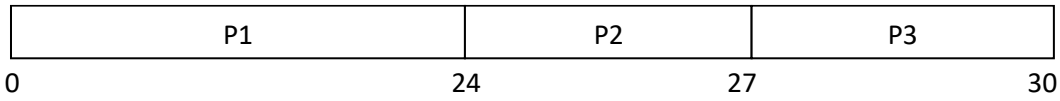
- The residence time (turnaround or transfer) (S) for each process is obtained by subtracting the process entry time from the termination time.
- The waiting (response) time (W) is obtained by subtracting the execution time from the sojourn time.

And to model the results we will use the Gantt chart.

Example

Process	CPU time
P1	24
P2	3
P3	3

If we consider that the order of arrival is P1, P2, P3, the Grantt diagram is:



	P1	P2	P3	Average
W	24-24=0	27-3=24	30-3=27	17
S	24	27	30	27

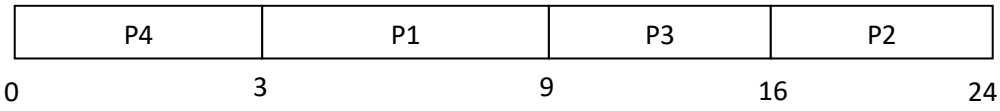
- FCFS is simple and easy to program (a single linked list to track Pret processes).
- But it puts short processes at a disadvantage if they are not the first.

3.2. SJF

SJF (Shortest Job First) uses a queue ordered by increasing execution time (shortest at the top of the list and slowest at the bottom). If two processes have the same estimated time, they are processed in FCFS. The estimation is done a priori.

Example

Process	CPU time
P1	6
P2	8
P3	7
P4	3



	P1	P2	P3	P4	Average
W	9-6=3	24-8=16	16-7=9	3-3=0	7
S	9	24	16	3	13

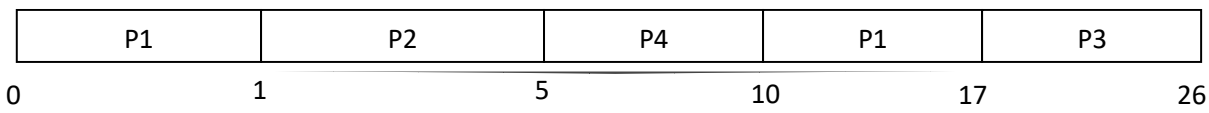
SW is optimal when all jobs are available simultaneously.

3.3. SRTF

SRTF (shortest remaining time first) is the preemptive version of the SJF algorithm: if a process shorter than the currently running process arrives in the queue, the running process is preempted by giving way to the newcomer.

Example

Process	CPU time	Arrival time
P1	8	0
P2	4	1
P3	9	2
P4	5	3



	P1	P2	P3	P4	Average
W	17-8=9	4-4=0	24-9=15	7-5=2	6.5
S	17-0=17	5-1=4	26-2=24	10-3=7	13

- The SRTF promotes short-term job services.

3.4. Round Robin (RR)

The round-robin algorithm is characterized by a time quantum and a circular FIFO queue.

The CPU is always allocated to the head process, for a quantum of time.

If the process crashes or terminates before its quantum expires, the CPU is immediately allocated to another process. If the process does not terminate at the end of its quantum, its execution is suspended and the CPU is allocated to another process.

The suspended process is inserted at the end of the queue.

Processes that arrive or transition from blocked to ready are inserted at the end of the queue.

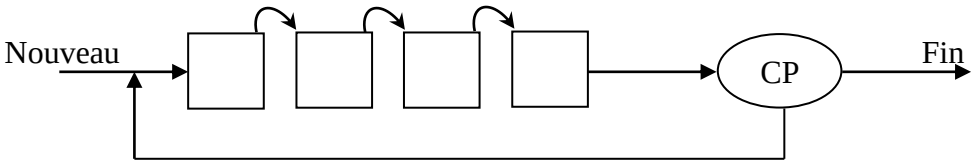


Figure 17. Circular Ordering.

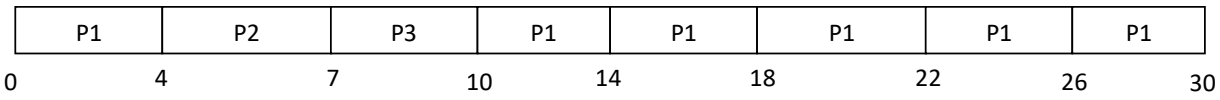
Choosing the value of a quantum

Too small a quantum causes too many process switches and lowers processor efficiency. Too high a quantum increases the response time of short commands in interactive mode. A quantum between 20 and 50 ms is often a reasonable compromise.

Example

Process	CPU time
P1	24
P2	3
P3	3

If we consider that the order of arrival is P1, P2, P3 at t=0 and that the quantum q=4, then the Gantt chart is:



	P1	P2	P3	Average
W	30-24=6	7-3=4	10-3=7	5.66
S	30	7	10	15.66

The round-robin algorithm allows for fair distribution of CPU.

However, it is not interesting if some processes are more important or urgent than others.

3.5. Scheduling with priorities

The priority scheduler assigns each process a priority. The CPU is allocated to the highest-priority process. New processes are queued according to their priority. Processes with the same priority are grouped into a FIFO-type queue.

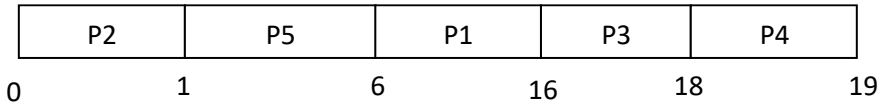
Note: The highest priority is defined by convention. 0 may be the highest or lowest depending on the system.

Priorities can be assigned to the process statically or dynamically.

Priority scheduling can be non-preemptive.

3.5.1. Non-preemptive priority

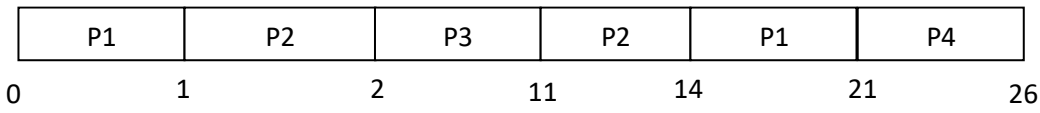
Process	CPU time	Priority
P1	10	3
P2	1	1 (highest)
P3	2	3
P4	1	4 (lowest)
P5	5	2



	P1	P2	P3	P4	P4	Average
W	16-10=6	1-1=0	18-2=16	19-1=18	6-5=1	8.2
S	16	1	18	19	6	12

3.5.2. Preemptive priority

Process	CPU time	Priority	Arrival time
P1	8	3	0
P2	4	2	1
P3	9	1 (the highest)	2
P4	5	3	3



- To prevent high priority processes from running indefinitely, there are methods of dynamically adjusting priority.

1. One way is to lower the priority of the running process after the quantum click expires, until there is a process with higher priority than it.
2. Another way is to prioritize I/O-intensive processes (typically interactive processes, which need to acquire the CPU as soon as they request it).

Its principle is as follows: when a process moves from the elected state to the blocked state, its priority is recalculated. Its new value is the ratio of the quantum/time actually used by the process. The processes with the highest ratio have the highest priority.

- If quantum = 100 ms and time used = 2 ms, the new priority is 50.
- If quantum. 100 ms and time used = 50 ms, the new priority is 2.

It is often convenient to have as many queues as there are priority levels. Typically, processes of the same priority are scheduled using a round-robin algorithm. With this policy, a category is only processed if the queues in higher (higher priority) categories are empty.

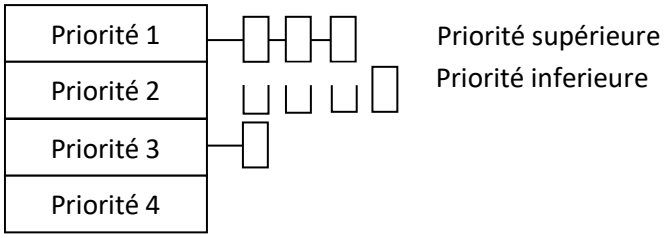


Figure III-2. Ordonnancement avec 4 catégories de priorités.

Lower categories risk starvation (the resource will never be allocated to the process). Therefore, it is necessary to periodically adjust the different priorities (dynamic assignment).

3.6. Multiple queues (quantum variable)

To avoid many switchings of time-consuming processes, it is better to allocate a larger quantum to these processes. When a process transitions from elected state for the first time, it is allocated the nth processor for one quantum. For the second time, the processor is allocated for 2 quanta. For the nth ^{time}, the processor is allocated for $2^{(n-1)}$ quanta.

Each process has a priority. Processes with the lowest number of quanta have the highest priority. Ready processes are distributed according to their priority into queues (FIFO). For example, when a process is suspended for the nth ^{time}, its priority is recalculated (2^{n}), and then it is inserted at the end of the appropriate queue.

Example

- Consider a process that has to perform calculations for 100 quanta. This process successively obtains 1, 2, 4, 8, 16, 32 and 64 quanta (it will use 37 of the last 64 quanta).
- The number of context switches goes from 100 (turnstile case) to 7.
- The processor spends less time switching, and therefore has better response time. Especially in cases where switching requires disk transfers.
- A process that moves lower and lower in the priority queues runs less and less frequently and thus favors short processes in interactive mode.
- To avoid disadvantaging a process that has been running for a long time before becoming interactive, it can be given the highest priority as soon as a carriage return is typed on the terminal associated with the process (early versions of Unix).

Tutorial

Exercise 01

The following figure represents the Gantt chart of a processor scheduling using the "Round Robin" algorithm and three processes: P1, P2 and P3.

P1	P2	P3	P1	P2	P1	Inactive	P3	P3

0 3 6 7 10 12 15 16 19 20

Question 1: What is the duration of the quantum?

Question 2: What is the waiting time of process P1?

Question 3: What is the waiting time for process P3? Justify.

Question 4: What happened between times t = 15 and t = 16? Justify.

Question 5 : What is the state of process P3 at time t = 9.

Question 6: Draw the Gantt chart of the same problem, but considering a quantum equal to 4.

Exercise 02

I-With the processes listed in the table below, draw a diagram illustrating their execution using:

1- The FCFS algorithm
2- The SJF algorithm
3- The SRTF algorithm
4- The round-robin algorithm (quantum = 2)
5- The round-robin algorithm (quantum = 1)

Process	Arrival date	Processing time
HAS	0.000	3
B	1.001	6
C	4.001	4
D	6,001	2

II- For the processes in the table above, what is the average rotation time if we use:

1- The FCFS algorithm
2- The SJF algorithm
3- The SRTF algorithm
4- The round-robin algorithm (quantum = 2)
5- The round-robin algorithm (quantum = 1)

III- For the processes in the table above, what is the waiting time of each process if we use:

1- The FCFS algorithm
2- The SJF algorithm
3- The SRTF algorithm
4- The round-robin algorithm (quantum = 2)
5- The round-robin algorithm (quantum = 1)

Chapter IV

Memory Management

Chapter IV : Memory Management

1. Definition

1.1. Random access memory

A type of memory that allows temporary storage of information. Its major advantage is its very fast reading capability compared to a hard drive, which ensures smooth computer usage (its purpose is to access data quickly and temporarily). When the system shuts down, the RAM is cleared.

Figure 19. RAM.

1.2. Read-Only memory

A type of memory whose content is accessible for reading but not for writing. It retains data even in the absence of electrical power. It stores the information needed to start a system (BIOS, embedded equipment, constant tables, etc.).

Figure 20. ROM.

1.3. Masse storage

Is a high-capacity memory that can be read and written by a system (magnetic tape, hard drive, optical disk (CD, DVD, Blu-ray), magneto-optical disk, and flash memory).

Figure 21. Masse storage.

1.4. Virtual Memory

It is an internal hard disk space in a computer that supplements the RAM. It is materialized by a swap file, which contains data that is not constantly accessed. Virtual memory, as its name suggests, is used to artificially increase the RAM.

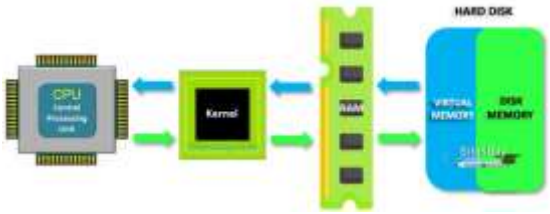


Figure 22. Virtual memory.

1.5. Cache

It is a faster memory located closer to the hardware (processor, hard drive). Its role is to store the most frequently used information by software and applications when they are active. This direct access determines a program's performance as it reduces the constant exchanges between the processor and the RAM.

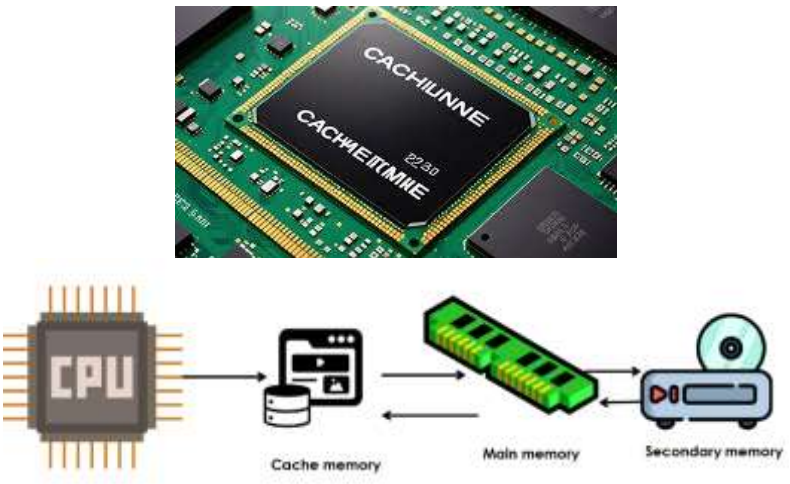


Figure 23. Cache.

2. Memory Manager

A memory manager is a software or hardware component responsible for allocating, managing, and freeing memory in a computer system. Its role is essential for ensuring the efficient and optimal use of available memory resources. Here are its main functions:

- Creation of a process.

- Activation/deactivation of a process.

- Deletion of a process.

- Sharing the available memory between processes (protection).

- Mapping the memory.

- Dynamically allocating/deallocating memory for the needs of a process.

- Optimizing memory usage.

3. Memory organization

An application can only run if its instructions and data are in memory. Therefore, if we want to run multiple programs simultaneously in a system, all these programs must be loaded into memory. The operating system will need to allocate a memory area for each program where it will be loaded.

The goal of good memory management is to increase the overall performance of the system.

3.1. Mono-programming

The simplest role of a memory manager is to run a single program at a time, sharing memory between the program and the operating system.

Only one process can run at a time. As soon as the user enters a command, the operating system copies the requested program from the disk to memory and executes it.

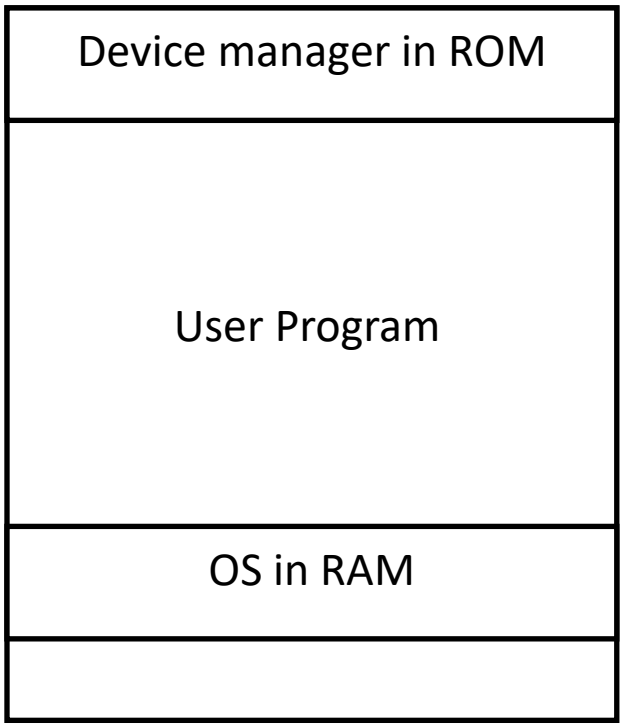


Figure 24. Monoprogramming.

3.2. Multiprogramming

To support multiprogramming and the existence of multiple processes in main memory, two main memory allocation strategies can be distinguished: contiguous partition allocation (Contiguous Memory Allocation) and non-contiguous partition allocation (Non-Contiguous Memory Allocation).

3.2.1. Multiprogramming and contiguous multiple partitions:

The memory space is divided into partitions. Each partition can be allocated to a program. The size of the partitions and thus their number can be either fixed or variable.

a. Fixed contiguous partitions

The memory is divided into *n* partitions of fixed sizes (if possible, unequal). This partitioning is done at system startup. When a process arrives, it can be placed in the queue.

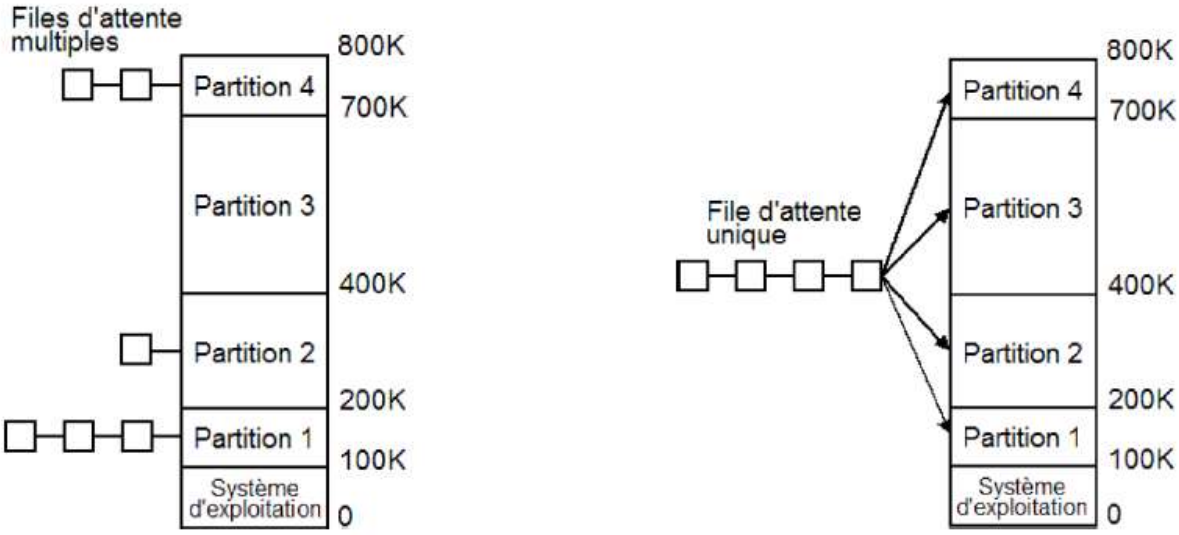


Figure 25. Fixed contiguous partitions.

b. Dynamic contiguous partitions

In this strategy, memory is partitioned dynamically according to demand. When a process terminates, its partition is reclaimed to be reused (completely or partially) by other processes. The memory manager must keep track of the occupied partitions and/or the free partitions.

The following placement strategies are distinguished:

- First Fit (FF): The first sufficiently large partition is allocated to the process.

- Best Fit (BF): The smallest partition whose size is at least equal to the size of the waiting process is allocated.

-Worst Fit (WF): The largest partition is allocated to the process.

Note: The BF and WF algorithms require sorting the free partitions in increasing address order.

c. Swap

Sometimes, the main memory is insufficient to keep all active processes in memory: in this case, additional processes must be stored on a disk and loaded to execute dynamically.

The swapping strategy involves considering each process in its entirety. The currently running program must be saved to disk before loading its successor into main memory for execution.

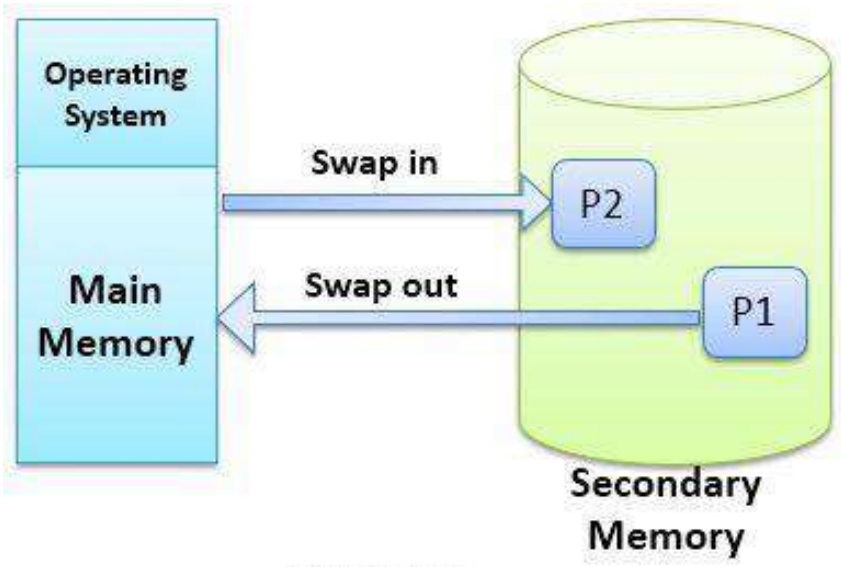


Figure 26. Swap.

Fragmentation problem

The previous contiguous memory allocation algorithms cause memory fragmentation. Indeed, due to the various process entries and exits, separate fragments (pieces) are formed in memory.

Solution → Defragmentation by compaction: the principle is to gather all the gaps into a single block

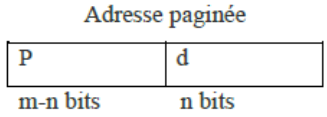
3.2.2. Multiprogramming and non-contiguous multiple partitions

We have seen that the allocation techniques associated with them lead to memory fragmentation (many small free areas that cannot be used). To avoid this, it is necessary to be able to place a program in several non-contiguous areas. Several techniques propose allocating non-contiguous memory spaces for processes: Paging and Segmentation.

A. Paging

Paging involves dividing the addressable space, or virtual space, into fixed-size areas called **pages**. The actual memory (RAM) is also divided into **frames**, each having the size of a page, so that each page can be placed in any frame of the physical memory.

- An address is divided into two parts:



- A page number **p**, and

- An offset (address) **d** within the page.

- The size of the page (and thus the frame) is a power of 2, typically ranging from 512 to 8192 bytes, depending on the architecture.

- If the size of the logical address space is 2^m and the size of a page is 2^n :

- The **m-n** most significant bits of a paged address represent the page number,

- The **n** least significant bits represent the offset within the page.

Page table: Address translation uses a page table, which is located in main memory or in registers. The real address (f, d) of a word from a virtual/logical address (p, d) is obtained by replacing the page number **p** with the frame number **f** found in the corresponding entry of the page table.

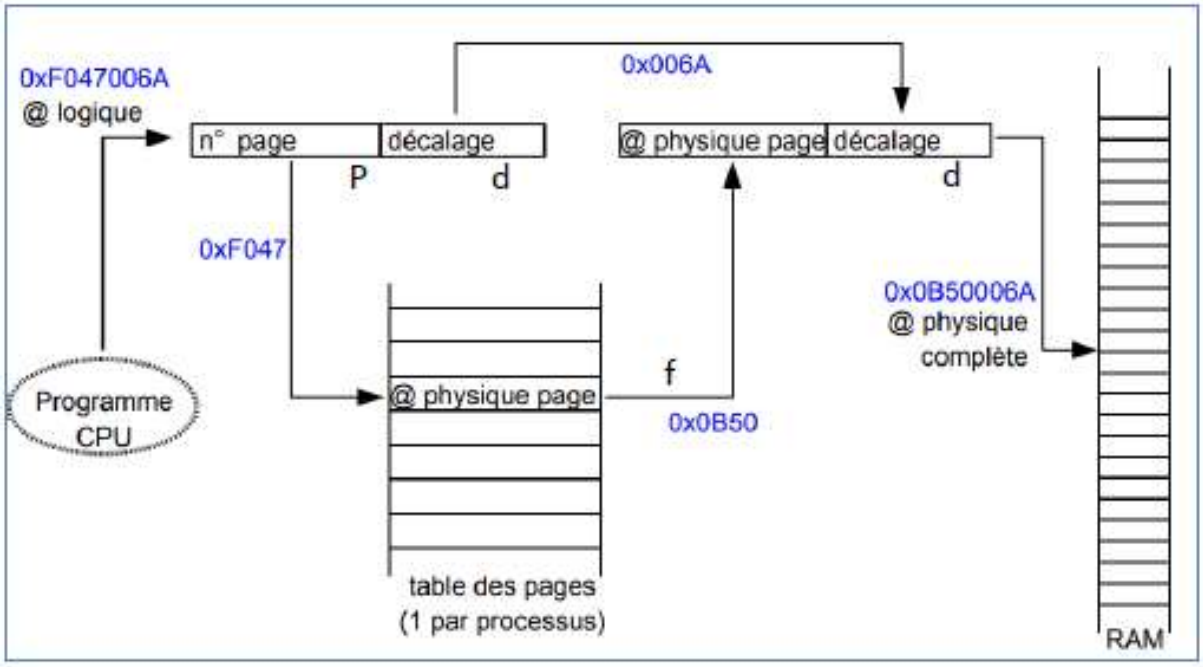


Figure 27. Segmentation.

b. Segmentation

Segmentation is similar to paging, except that the size of a segment is variable. The logical address space is divided into a set of segments. The advantage of segmentation over paging is that the segments can reflect a logical view of the program. For example, each segment may represent a program module generated during compilation.

Segment table: The segment table specifies two values for each segment:

- Base: the starting address of the segment in main memory,

- Limit: the size of the segment.

A logical (virtual) address is called segmented. It includes a segment number (s) and an offset within the segment (d). The segment number (s) is used as an index in the segment table. The segment table is usually implemented using fast registers. If d > limit, a boundary error occurs (famous Segmentation fault).

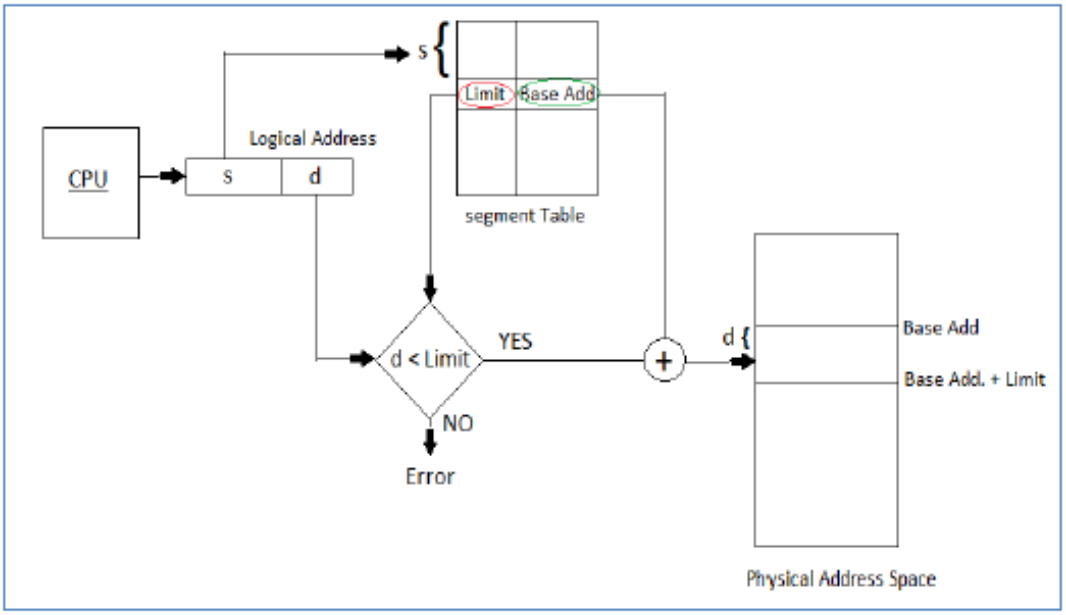


Figure 28. Pagination.

Laboratory Sessions

Chapter V : Laboratory Sessions

1. Laboratory Session 01 – Introduction to DOS Commands

Objective: Learn how to use DOS commands to navigate, manage files, and perform basic tasks in an operating system environment.

I. Exploring the DOS Environment

1. **Opening the Command Prompt**

   - Click **Start** → **Run** → Type `cmd` → Press **Enter**.
   - A black window opens, showing the command prompt (e.g., `C:\Users\YourName>`).

2. **Display Help**

   - Type `help` and press **Enter**.
   - Note down some basic commands from the displayed list.

II. Navigating the File System

1. **Using `dir`**

   - Type `dir` to display the contents of the current directory.
   - Use `dir /p` to display the contents page by page.
   - Type `dir /w` for a wide listing view.

2. **Changing Directories**

   - Type `cd directory_name` to enter a directory (e.g., `cd Documents`).
   - Type `cd ..` to move up to the parent directory.
   - Type `cd\` to return to the root directory.

3. **Creating a Directory**
   - Use `mkdir directory_name` to create a new directory.
4. **Deleting a Directory**
   - Type `rmdir directory_name` to remove an empty directory.

III. Managing Files

1. **Creating a File**

- Type `echo Hello > file.txt` to create a text file.
- Verify its creation by using `dir`.

2. **Viewing File Content**
   - Type `type file.txt` to display the file's content.
3. **Renaming a File**
   - Use `rename file.txt new_name.txt`.
4. **Copying a File**
   - Type `copy new_name.txt copy.txt`.
5. **Deleting a File**
   - Use `del copy.txt`.

2. Laboratory Session 02 – Process Management Commands Windows

Objective: Learn how to manage processes in Windows, including creating, listing, filtering, searching, and terminating processes using Command Prompt or PowerShell.

I. Introduction to Processes

A process is an instance of a program running on a system. Managing processes is essential for system performance, troubleshooting, and automation.

II. Common Commands for Process Management

1. Listing Processes

- **`tasklist`**: Displays a list of all running processes.
  - **Options**:
    - `tasklist /v`: Displays detailed information about each process.
    - `tasklist /fi "USERNAME eq <username>"`: Filters processes by user.
    - `tasklist /fi "STATUS eq running"`: Filters only running processes.

2. Searching for a Process

- **`findstr`**: Searches for a specific process in the tasklist output.
  - Example: `tasklist | findstr chrome`

3. Killing a Process

- **`taskkill`**: Terminates a process.
  - By process ID (PID):

    ```
    taskkill /PID <process_id>
    ```

  - By process name:

    ```
    taskkill /IM <process_name> /F
    ```

    - `/F`: Forces the process to terminate.

4. Creating a Process

- **`start`**: Launches a new process.
  - Example:

    ```
    start notepad.exe
    ```

    This opens Notepad.

III. Practical Exercises

1. **Listing and Filtering Processes**
   - Use `tasklist` to display all processes.
   - Filter the list to show processes running under your username.
2. **Searching for a Process**
   - Search for `explorer.exe` using `findstr`.
3. **Terminating Processes**
   - Open Notepad using `start notepad.exe`.
   - Identify its PID using `tasklist`.
   - Terminate it using `taskkill`.
4. **Detailed Process Information**
   - Display information about `notepad.exe` using `wmic`.
5. **Monitoring Performance**
   - Monitor CPU usage using `typeperf`.

3. Laboratory Session 03 – Thread in Java

Exercise 1: Basic Thread Creation

Objective: Understand how to create and run threads in Java.

1. Create a class that extends the `Thread` class.
2. In the `run()` method, print the name of the current thread 5 times with a delay of 500 milliseconds between each print.
3. In the `main()` method, create and start three threads that execute the above task concurrently.

Expected Output: Each thread should print its name 5 times with a delay between prints.

```
Thread-0
Thread-1
Thread-2
```

Exercise 2: Thread with Runnable Interface

Objective: Use the `Runnable` interface to define thread tasks.

1. Implement the `Runnable` interface to create a class `MyRunnable`.
2. In the `run()` method of `MyRunnable`, print numbers from 1 to 5 with a delay of 1000 milliseconds between each number.
3. In the `main()` method, create two instances of `MyRunnable`, and start them on two separate threads.

Expected Output

```
1
2
3
4
5
1
2
3
4
5
```

Exercise 3: Thread Synchronization

Objective: Learn how to synchronize threads to avoid race conditions.

1. Create a class `Counter` with a `count` variable and methods to increment and get the value of `count`.
2. In the `main()` method, create two threads that call the increment method 1000 times each.

3. The `increment` method should be synchronized to ensure that only one thread can access it at a time.
4. Print the final value of `count` after both threads finish.

Expected Output: The output should always be 2000, regardless of the order in which the threads execute.

```
count = 2000
```

4. Laboratory Session 04 – Linux and Process Management

By the end of this lab, students should be able to:

- Navigate and manipulate the Linux file system using basic shell commands.
- Understand the concept of **processes** in an operating system.
- Create and manage processes using **system calls** (`fork()`, `getpid()`, `exec()`, `wait()`).
- Observe process attributes using Linux utilities (`ps`, `top`, `kill`, `nice`).

Concepts Covered

- User vs kernel mode
- Process lifecycle
- Process IDs (PID, PPID)
- Parent-child relationship
- Zombie and orphan processes
- System calls related to process creation and control

Pre-Lab Preparation

Before starting, students should review:

1. The basic Linux command line structure.
2. The concept of a process and how it differs from a program.
3. The purpose of the following commands:

```
4. ps aux
5. top
6. kill -9 PID
7. nice -n 10 ./program
```

8. The syntax and behavior of:

```
9. fork(), getpid(), getppid(), exec(), wait()
```

Exercises

Exercise 1 — Exploring Processes with Linux Commands

1. Open a terminal and run:
2. `ps -ef | more`
    - Identify your current shell process and its parent process.
3. Run a background job:
4. `gedit &`
5. `ps -ef | grep gedit`
    - Note its **PID** and **PPID**.
    - Terminate it using:
    - `kill PID`
6. Run `top` and observe:
    - Which processes consume the most CPU?
    - What happens when you press `k` inside `top`?

Exercise 2 — Creating a Child Process

Create a program that spawns a child process.

Code (student template)

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork();

    if (pid < 0) {
        printf("Fork failed!\n");
    } else if (pid == 0) {
        // Child process
        printf("I am the child process. My PID: %d, Parent PID: %d\n",
getpid(), getppid());
    } else {
        // Parent process
        printf("I am the parent process. My PID: %d, Child PID: %d\n",
getpid(), pid);
    }

    return 0;
}
```

Compile and run

```
gcc fork_example.c -o fork_example
./fork_example
```

Expected Output

```
I am the parent process. My PID: 1234, Child PID: 1235
I am the child process. My PID: 1235, Parent PID: 1234
```

Exercise 3 — Using `exec()` and `wait()`

Modify the previous program so the child executes a new program (`ls -l`).

Code with solution

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process replaces itself with 'ls -l'
        printf("Child process executing 'ls -l'...\n");
        execlp("ls", "ls", "-l", NULL);
        printf("This line will not be executed if exec is successful.\n");
    } else if (pid > 0) {
        // Parent process waits for child to finish
        wait(NULL);
        printf("Parent process: Child completed.\n");
    } else {
        printf("Fork failed!\n");
    }

    return 0;
}
```

Output Example

```
Child process executing 'ls -l'...
(total file list appears here)
Parent process: Child completed.
```

Exercise 4 — Demonstrating a Zombie Process

Code with solution

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
```

```
        printf("Child process exiting...\n");
        exit(0);
    } else {
        printf("Parent sleeping for 10 seconds...\n");
        sleep(10);
        system("ps -l | grep Z"); // Check zombie process
    }

    return 0;
}
```

Explanation

- The child terminates before the parent calls `wait()`.
- The child becomes a **zombie** (defunct) until the parent collects its exit status.

Exercise 5 — Process Priority

Run the same program with different priorities:

```
nice -n 10 ./fork_example
nice -n -10 ./fork_example
```

Observe differences in CPU scheduling with the `top` command.

Solutions Summary

Exercise	Key Learning	Expected Outcome
1	Understanding process hierarchy	Students identify parent/child using PIDs
2	Process creation	Correct use of `fork()` and `getpid()`
3	Program replacement	Successful `exec()` and `wait()`
4	Process states	Identification of zombie process
5	Scheduling	Understanding `nice` and priority effects

5. Laboratory Session 05 – System call and Process control

1. **Objectives**

By the end of this lab, students will be able to:

- Understand and use key **process-related system calls** in Linux.
- Create and manage **multiple child processes**.
- Use **`fork()`**, **`exec()`**, and **`wait()`** to control process execution.
- Understand **process trees**, **orphan processes**, and **zombie processes** in depth.

2. **Concepts Covered**

- Process creation using `fork()`
- Replacing process images using `exec()` family
- Synchronization between parent and child using `wait()` and `waitpid()`
- Orphan and zombie process behavior
- Process hierarchy visualization with `pstree`

3. **Pre-Lab Preparation**

Before starting, students should:

1. Review `man` pages for:
2. `man fork`
3. `man exec`
4. `man wait`
5. `man waitpid`
6. Understand the return values and behavior of each function:
    - `fork()` returns **0** to the child, **child PID** to the parent.
    - `exec()` replaces the current process image.
    - `wait()` suspends the parent until the child terminates.
7. Install **`pstree`** if not available:
8. `sudo apt install psmisc`

4. Exercises

4.1. Exercise 1 — Creating Multiple Child Processes

4.1.1. Task

Write a C program that creates **three child processes**, each printing its own PID and parent PID.

Student Template

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    int i;

    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == 0) {
            printf("Child %d: PID=%d, PPID=%d\n", i+1, getpid(),
getppid());
            _exit(0);
        }
    }

    return 0;
}
```

Compile and Run

```
gcc multi_fork.c -o multi_fork
./multi_fork
```

Sample Output

```
Child 1: PID=1345, PPID=1344
Child 2: PID=1346, PPID=1344
Child 3: PID=1347, PPID=1344
```

Exercise 2 — Sequential vs Parallel Execution

Task

Modify the above program so that:

- Children run **sequentially** (the parent waits for each before creating the next).

Solution

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```
int main() {
    pid_t pid;
    int i;

    for (i = 0; i < 3; i++) {
        pid = fork();
        if (pid == 0) {
            printf("Child %d running (PID=%d)\n", i+1, getpid());
            sleep(2);
            _exit(0);
        } else {
            wait(NULL); // Parent waits for each child
        }
    }

    printf("All child processes completed.\n");
    return 0;
}
```

Concept

- Demonstrates **synchronous** process creation.
- Parent process waits before continuing.

4.3. Exercise 3 — Using `exec()` to Run Another Program

Task

Child process executes a new program (e.g., list directory contents).

Solution

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child process executing 'ls -l'...\n");
        execlp("ls", "ls", "-l", NULL);
        perror("exec failed"); // In case exec fails
    } else if (pid > 0) {
        wait(NULL);
        printf("Parent: child completed successfully.\n");
    } else {
        perror("fork failed");
    }

    return 0;
```

```
}
```

Concept

- `execlp()` replaces the current process with a new one.
- Parent remains unaffected and waits for child completion.

4.4. Exercise 4 — Orphan and Zombie Process Behavior

Part A: Orphan Process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        sleep(5); // Ensure parent exits first
        printf("Child (PID=%d): Parent PID after orphaning = %d\n",
getpid(), getppid());
    } else {
        printf("Parent (PID=%d) exiting now...\n", getpid());
        exit(0);
    }

    return 0;
}
```

Expected Output

```
Parent (PID=1234) exiting now...
Child (PID=1235): Parent PID after orphaning = 1
```

→ The **init/systemd process (PID=1)** adopts the orphan.

Part B: Zombie Process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child exiting...\n");
        exit(0);
    } else {
        printf("Parent sleeping before wait...\n");
        sleep(10);
```

```
        system("ps -l | grep Z"); // Check for zombie
        wait(NULL);
    }

    return 0;
}
```

→ Observe the "**Z**" status in process list.

4.5. Exercise 5 — Using `waitpid()`

Task

Parent waits for specific child process (out of many) using `waitpid()`.

Solution

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;

    pid1 = fork();

    if (pid1 == 0) {
        printf("Child 1 (PID=%d)\n", getpid());
        sleep(3);
        _exit(0);
    } else {
        pid2 = fork();
        if (pid2 == 0) {
            printf("Child 2 (PID=%d)\n", getpid());
            sleep(1);
            _exit(0);
        } else {
            // Parent waits for first child only
            waitpid(pid1, NULL, 0);
            printf("Parent waited for Child 1.\n");
        }
    }

    return 0;
}
```

Key Points

Concept	System Call(s)	Takeaway
Process creation	`fork()`	Creates a new process (child)
Execution replacement	`exec()`	Replaces current process image
Synchronization	`wait()`, `waitpid()`	Ensures ordered completion
Orphan process	`fork()` + `exit()`	Adopted by `init`

6. Laboratory Session 05 – Process communication

5.1. **Objectives**

By the end of this lab, students will be able to:

- Understand the concept of **Inter-Process Communication (IPC)**.
- Implement **anonymous pipes** for data transfer between parent and child processes.
- Implement **named pipes (FIFOs)** for communication between unrelated processes.
- Understand the **synchronization and data flow** between processes.

5.2. **Concepts Covered**

- Pipe mechanism in Linux
- Anonymous vs named pipes
- Blocking and non-blocking reads/writes
- FIFO creation and usage (`mkfifo`, `open`, `read`, `write`)
- Synchronization issues in IPC

5.3. **Pre-Lab Preparation**

Students should review:

1. System calls related to pipes:
2. `man pipe`
3. `man mkfifo`
4. `man read`
5. `man write`
6. `man open`
7. The difference between **anonymous pipes** (parent-child) and **named pipes/FIFOs** (any processes).
8. The behavior of **blocking reads/writes**.

5.4. **Exercises**

Exercise 1 — Using Anonymous Pipes

Task

Create a **pipe** to send a message from the parent process to the child process.

Student Template

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd[2];
    char write_msg[] = "Hello from parent!";
    char read_msg[100];

    if (pipe(fd) == -1) {
        perror("pipe failed");
        return 1;
    }

    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        close(fd[1]); // Close write end
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Child received: %s\n", read_msg);
        close(fd[0]);
    } else {
        // Parent process
        close(fd[0]); // Close read end
        write(fd[1], write_msg, strlen(write_msg)+1);
        close(fd[1]);
    }

    return 0;
}
```

Expected Output

```
Child received: Hello from parent!
```

Exercise 2 — Bi-Directional Communication with Anonymous Pipe

Task

Parent sends a message to child, child responds back through another pipe.

Solution

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int pipe1[2], pipe2[2];
    char parent_msg[] = "Message to child";
    char child_msg[] = "Reply from child";
    char buffer[100];

    pipe(pipe1); // Parent → Child
    pipe(pipe2); // Child → Parent

    pid_t pid = fork();

    if (pid == 0) {
        // Child
        close(pipe1[1]); // Close write end of pipe1
        close(pipe2[0]); // Close read end of pipe2

        read(pipe1[0], buffer, sizeof(buffer));
        printf("Child received: %s\n", buffer);

        write(pipe2[1], child_msg, strlen(child_msg)+1);
        close(pipe1[0]);
        close(pipe2[1]);
    } else {
        // Parent
        close(pipe1[0]);
        close(pipe2[1]);

        write(pipe1[1], parent_msg, strlen(parent_msg)+1);
        read(pipe2[0], buffer, sizeof(buffer));
        printf("Parent received: %s\n", buffer);

        close(pipe1[1]);
        close(pipe2[0]);
    }

    return 0;
}
```

Expected Output

```
Child received: Message to child
Parent received: Reply from child
```

Exercise 3 — Using Named Pipes (FIFOs)

Task

Use a **FIFO** to send data between two unrelated processes.

Steps

1. Create FIFO in terminal:

```
mkfifo myfifo
```

2. Create **writer.c**:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main() {
    int fd = open("myfifo", O_WRONLY);
    char msg[] = "Hello through FIFO!";
    write(fd, msg, strlen(msg)+1);
    close(fd);
    return 0;
}
```

3. Create **reader.c**:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("myfifo", O_RDONLY);
    char buffer[100];
    read(fd, buffer, sizeof(buffer));
    printf("Reader received: %s\n", buffer);
    close(fd);
    return 0;
}
```

4. Compile:

```
gcc writer.c -o writer
```

```
gcc reader.c -o reader
```

5. Run in **two terminals**:

```
./reader
./writer
```

Expected Output (Reader Terminal)

```
Reader received: Hello through FIFO!
```

Exercise 4 — Synchronization Considerations

Task

- Observe what happens if the reader is started **after** the writer finishes writing to FIFO.
- Hint: FIFO is **blocking by default**, so writer waits for reader.

5. Key Points

Concept	Key Learning
Anonymous pipes	Parent-child communication only
Bi-directional pipes	Need two separate pipes
FIFO (named pipe)	Can communicate between unrelated processes
Blocking behavior	Reading or writing blocks until other end is open
Synchronization	Order of execution matters for correct data transfer

7. Laboratory Session 06 – Shared Memory and Message Queues

Objectives

By the end of this lab, students will be able to:

- Understand and implement **shared memory** for IPC.
- Use **message queues** to exchange data between processes.
- Synchronize processes to avoid **data inconsistency**.
- Compare the advantages and limitations of different IPC methods.

Concepts Covered

- Shared memory (`shmget()`, `shmat()`, `shmdt()`, `shmctl()`)
- Message queues (`msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`)
- Inter-process synchronization
- Key differences between **pipes** and **shared memory**

Pre-Lab Preparation

Students should review:

1. `man shmget`, `man shmat`, `man msgget`
2. Structure of a message in System V message queues:

```
struct msgbuf {
    long mtype;
    char mtext[100];
};
```

3. Shared memory operations: creation, attachment, detachment, and deletion.
4. Ensure `#include <sys/ipc.h>`, `<sys/shm.h>`, `<sys/msg.h>` are available.

Exercises

Exercise 1 — Shared Memory

Task

Create a shared memory segment where **parent writes data** and **child reads it**.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>

int main() {
    key_t key = 1234;
```

```
    int shmid = shmget(key, 1024, 0666|IPC_CREAT);
    if (shmid == -1) {
        perror("shmget failed");
        return 1;
    }

    char *data = (char*) shmat(shmid, NULL, 0);
    if (data == (char*) -1) {
        perror("shmat failed");
        return 1;
    }

    pid_t pid = fork();

    if (pid == 0) {
        // Child process reads shared memory
        sleep(1); // Ensure parent writes first
        printf("Child read from shared memory: %s\n", data);
        shmdt(data);
    } else {
        // Parent process writes to shared memory
        strcpy(data, "Hello from parent via shared memory!");
        wait(NULL);
        shmdt(data);
        shmctl(shmid, IPC_RMID, NULL); // Delete shared memory
    }

    return 0;
}
```

Expected Output

```
Child read from shared memory: Hello from parent via shared memory!
```

Exercise 2 — Message Queue

Task

Parent sends a message to child using a **System V message queue**.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>

struct msgbuf {
    long mtype;
    char mtext[100];
};
```

```
int main() {
    key_t key = 5678;
    int msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget failed");
        return 1;
    }

    struct msgbuf message;
    pid_t pid = fork();

    if (pid == 0) {
        // Child receives message
        msgrcv(msgid, &message, sizeof(message.mtext), 1, 0);
        printf("Child received: %s\n", message.mtext);
    } else {
        // Parent sends message
        message.mtype = 1;
        strcpy(message.mtext, "Hello from parent via message queue!");
        msgsnd(msgid, &message, sizeof(message.mtext), 0);
        wait(NULL);
        msgctl(msgid, IPC_RMID, NULL); // Delete message queue
    }

    return 0;
}
```

Expected Output

```
Child received: Hello from parent via message queue!
```

Exercise 3 — Multiple Messages

Task

Send multiple messages from **parent to child**, each with a different type.

- Child receives only messages of a specific type.

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>

struct msgbuf {
    long mtype;
    char mtext[100];
};
```

```
int main() {
    key_t key = 6789;
    int msgid = msgget(key, 0666 | IPC_CREAT);

    struct msgbuf message;
    pid_t pid = fork();

    if (pid == 0) {
        // Child receives only type 2 messages
        msgrcv(msgid, &message, sizeof(message.mtext), 2, 0);
        printf("Child received: %s\n", message.mtext);
    } else {
        message.mtype = 1;
        strcpy(message.mtext, "Message type 1");
        msgsnd(msgid, &message, sizeof(message.mtext), 0);

        message.mtype = 2;
        strcpy(message.mtext, "Message type 2");
        msgsnd(msgid, &message, sizeof(message.mtext), 0);

        wait(NULL);
        msgctl(msgid, IPC_RMID, NULL);
    }

    return 0;
}
```

Expected Output

```
Child received: Message type 2
```

Instructor Notes / Key Points

Concept	System Call(s)	Key Takeaway
Shared memory	`shmget()`, `shmat()`, `shmdt()`, `shmctl()`	Fastest IPC, shared memory segment accessible by multiple processes
Message queue	`msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`	Allows structured messages, supports different types
Synchronization	sleep() / wait()	Ensure data is written before reading
Resource	`shmctl(..., IPC_RMID)` /	Avoids resource leaks

cleanup	`msgctl(..., IPC_RMID)`	

Bibliography

1. Y. Epelboin, “Operating System Course”, Pierre and Marie Curie University, France, www-int.impmc.upmc.fr , consulted on: 04/08/2017 at 08:00.
2. M. Loukam, “Operating System Course”, Hassiba Benbouali University of Chlef, Algeria, www.loukam.net , consulted on: 04/08/2017 at 08:00.
3. A. Silberschatz, P. Galvin, “Principles of Operating Systems”, Addison-Welsly, 1994.
4. G. Nutt, “Open Systems”, InterEdition 1995.
5. A. Tanenbaum, “Operating Systems: Centralized Systems, Distributed Systems,” Prentice-Hall 1994.
6. Website, http://www.samomoi.com/ordinateur/les-interfaces-d-entree-sortie.php , consulted on 09/22/2017 at 12:13.
7. M. Belaid, “Computer Architecture, Operation & Technology”, Blue Pages.
8. N. Salmi, “Principles of Operating Systems”, Blue Pages.
9. M. Halfeld Ferrari, “Gestion de la mémoire”, Université d’Orleans.