UNIVERSITY MOULOUD MAMMERI OF TIZI OUZOU
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

# Operating systems 1

## Level: First year engineering cycle

**FILALI Idir**
**2023/2024**

# TABLE OF CONTENTS

**I. INTRODUCTION TO OPERATING SYSTEMS**………………………………… 1
I.1INTRODUCTION………………………………………………………………… 1
I.2 PRELIMINAIRY CONCEPTS……………………………………………………… 2
I.2.1 Computer System…………………………………………………………… 2
I.2.2 OPERATING SYSTEMS……………………………………………………… 3
I.2.2.1 Définition…………………………………………………………… 3
I.2.2.2 Fonctions of an operating system ……………………………………… 3
I.2.3 Evolution of operating systems ……………………………………………… 5
I.2.3.1 Bare machines: 1945-1955…………………………………………… 5
I.2.3.2 Batch processing: 1955-1965…………………………………………… 5
I. 2.3.3 Multiprogramming and batch processing: 1965-1980……………………… 6
I.2.3.4 The fourth generation: 1980-1990 (Personal computing-Network operating systems- Distributed systems)…………………………………… 9
I.2.3.5 The fifth generation: 1990 to the present day (Miniaturization, Embedded systems)……………………………………………………………… 10
I.3 USER/SYSTEM INTERACTION ……………………………………………… 10
I.4 SYSTEM calls………………………………………………………………… 11
I.4.1 Processes……………………………………………………………… 11
I.4.2 Files…………………………………………………………………… 14
I.5 STRUCTURE OF AN OPERATING SYSTEM ……………………………………… 14
I.5.1 Layered Structure ……………………………………………………… 14
I.5.2 Monolithic structure …………………………………………………… 15
I.5.3 Micro-kernel…………………………………………………………… 15

**II. INTRODUCTION TO THE SYSTEM LINUX/UNIX**. ……………………………… 17
II.1 INTRODUCTION………………………………………………………………… 17
II.2 BRIEF HISTORY………………………………………………………………… 17
II.3 FEATURES……………………………………………………………………… 19
II.3.1 Structure of a Unix/Linux system……………………………………………… 19
II.3.2 Start of a session…………………………………………………………… 20
II.3.3 Command interpreters……………………………………………………… 21
II.3.4 Useful Unix/Linux commands 22
II.4 FILE PROTECTION……………………………………………………………… 26
II.4.1 File access rights…………………………………………………………… 26
II.4.2 Modification of file permissions……………………………………………… 27
II.4.3 Permission at the file creation………………………………………………… 28
II.4.4 Access permissions of directories……………………………………………… 29
II.4.5 Permission to delete and the bit t……………………………………………… 30
II.4.6 Change owner and group…………………………………………………… 30
II.4.7 Multi-group membership…………………………………………………… 31

**III. THE BASIC MECHANISMS OF OPERATING SYSTEMS**……………………… 34
III.1 INTRODUCTION………………………………………………………………… 34
III.2 THE LIFE CYCLE OF A PROGRAM……………………………………………… 34
III.2.1 Translation (compilation or assembly)……………………………………… 34

III.2.1.1 The assembler.. 36
III.2.1.1.1 Operation Codes (Mnemonics) 37
III.2.1.1.2 Operands and labels 37
III.2.1.1.3 Literals 38
III.2.1.1.4 Directives 38
III.2.1.2 ASSEMBLY TECHNIQUES 39
III.2.1.2.1 Two-pass assembly 39
III.2.1.2.2 One pass assembly 41
III.2.1.3 Format of an object program (Unix type) 41
III.2.2 Link editing 42
III.2.3 Loading 44
III.2.4 Processes 45
III.3 SEPARATE COMPILATION 46
III.3.1 Global variables 46
III.3.2 Putting code in multiple files 47
III.3.3 Compile a multi-file project 50
III.3.3.1 Without makefile 50
III.3.3.2 With makefile 51

IV SHELL PROGRAMMING 53
IV.1 INTRODUCTION 53
IV.2 BASIC SHELL PROGRAMMING 54
IV.2.1 The first script 54
IV.2.2 Parameter passing 55
IV.2.3 Special variables 56
IV.2.4 Special characters 57
IV.2.5 Read and write instructions 57
IV.2.6 Control structures 58
IV.3 INTEGER ARITHMETIC ON VARIABLES 66
IV.4 ADVANCED BASH PROGRAMMING 67
IV.4.1 Bash predefined variables (not defined in Bourne-shell) 67
IV.4.2 Variable definition: the declare command 67
IV.4.3 The test command 68
IV.4.4 Integer Arithmetic 70
IV.5 SCRIPT WRITING 71
IV.5.1 Parentheses 71
IV.5.2 Substitution 71
IV.5.3 The command substitution $(command) 72
IV.5.4 Bash extensions in name generation 72
IV.5.5 Quoting (neutralization) 73

V PROCESS MANAGEMENT 75
V.1 INTRODUCTION 75
V.2 PROCESS FEATURES 75
V.2.1 States of a process 75
V.2.2 Process implementation 76
V.2.2 .1 Process Control Block 76
V.2.2.2 Process Context Switch 77
V.2.3 Daemons 78

V.2.4 Process creation and termination........................................................................ 78
V.3 SERVICES POSIX FOR PROCESS MANAGEMENT................................................ 79
V.3.1 Process creation........................................................................................... 79
V.3.1.1 Creating processes with system ( )........................................................ 79
V.3.1.2 Creating processes with fork ( )............................................................ 80
V.3.2 System calls wait ( ), waitpid ( ) and exit ( )...................................................... 88
V.3.3 Zombie process............................................................................................ 94
V.3.4 The family of system calls exec........................................................................ 96
V.3.5 Comment on system ( )................................................................................. 99

# List of Figures


FIG I.1 The layers of a computer system........................................................ 2
FIG I.2 Batch processing and memory organization ........................................... 5
FIG I.3 CPU and Peripheral Usage .......................................................... 6
FIG I.4 Memory partitions .............................................................. 7
FIG I.5 Processor switching............................................................... 8
FIG I.6 Time sharing: Model1.............................................................. 9
FIG I.7 Time sharing: Model2.............................................................. 9
FIG I.8 Alternation of execution modes of a process ......................................... 13
FIG I.9 Steps of a system call.............................................................. 14
FIG I.10 Layered structure..... ............................................................ 15
FIG I.11 Monolithic structure................................................................ 16
FIG I.12 Micro-kernel structure.............................................................. 17
FIG II.1 Evolution of Unix.................................................................. 19
FIG II.2 Standard Posix..................................................................... 19
FIG II.3 Architecture of a Unix system......................................................... 21
FIG II.4 Parallel chaining of cat and sort commands............................................ 26
FIG III.1 Life cycle of a program.............................................................. 36
FIG III.2 Link editor......................................................................... 44
FIG III.3 Image of a process in memory.......................................................... 47
FIG III.4 Example of separate compilation of two executable files................................ 51
FIG V.1 The main states of a process........................................................... 77
FIG V.2 Five-state model of a process.......................................................... 77
FIG V.3 The context switch.................................................................... 79
FIG V.4 Father process and its cloned child...................................................... 82
FIG V.5 Services Posix : fork ( )............................................................... 82
FIG V.6 Process tree structure.................................................................. 89
FGI V.7 Zombie process: the son dies and the father did not wait ( )................................ 95
FGI V.8 Orphan process: the father dies before the son. init adopt the son......................... 95
FIG V.9 Services Posix : exec.................................................................. 97

# List of tables


Table I.1 Some Posix system calls.......... 12
Table II.1 Unix BSD kernel architecture .......... 22
Table II.2 Architecture of Linux.......... 23
Table II.3 Commands Posix from Unix.......... 22
Table III.1 Table of symbols.......... 41
Table III.2 Table of operation codes.......... 41
Table III.3 Table of symbols (one-pass assembler).......... 43
Table III.4 table of assembled instructions.......... 43
Table III.5 Table of unassembled instructions.......... 43
Table V.1 Main entries of the Process table from Unix.......... 78

# Chapter I

## INTRODUCTION TO OPERATING SYSTEMS

### I.1 INTRODUCTION

A computer is a set of very complex hardware components that can run user programs. The best known architecture is that of Von Neumann, in which the computer is composed of a central memory, a central unit and a set of input/output units. Without programs, a computer is useless. With its various programs (software), the computer can record, manipulate and generate information; To see movies; listen to music; send e-mails; connect to the internet, etc.

To be executed, a program must reside in memory. The processor generally executes it sequentially (instruction by instruction). Program data and results pass through I/O units. Due to the complexity of the hardware, it becomes increasingly difficult to manipulate it directly by user programs. Access, for example, to an input/output unit requires several pages of complex programs (often in assembly language). In addition, each material is specific and requires an adapted program. In the past, users ran their programs on bare metal machines (without pre-installed software). They programmed in low-level languages (machine or assembler) and also took care of the programs necessary for the management of the hardware. Currently, this task of hardware management is bequeathed to a particular software called the operating system. Thanks to the operating system, saving a file to a disk drive becomes as easy as one or two mouse clicks.

## I.2 PRELIMINARY CONCEPTS

### I.2.1 Computer System

A computer system is the set of hardware and software in a computer. It can be seen as a set of layers that wrap around each other as summarized in FIG I.1. The following elements (or layers) can be distinguished:

a) Hardware (Hardware): Refers to the processor, main memory, disk units and other input/output units. These devices are called computer resources.

b) The Operating System: It wraps the hardware to make it easier to use at higher layers.

c) System programs: Such as compilers, linkers, command interpreters and graphical interfaces. Although these programs perform system tasks, they are not actually part of the operating system.

d) User programs: These are programs written (or purchased) by users to perform specific tasks (word processing, management, calculations, engineering, etc.).

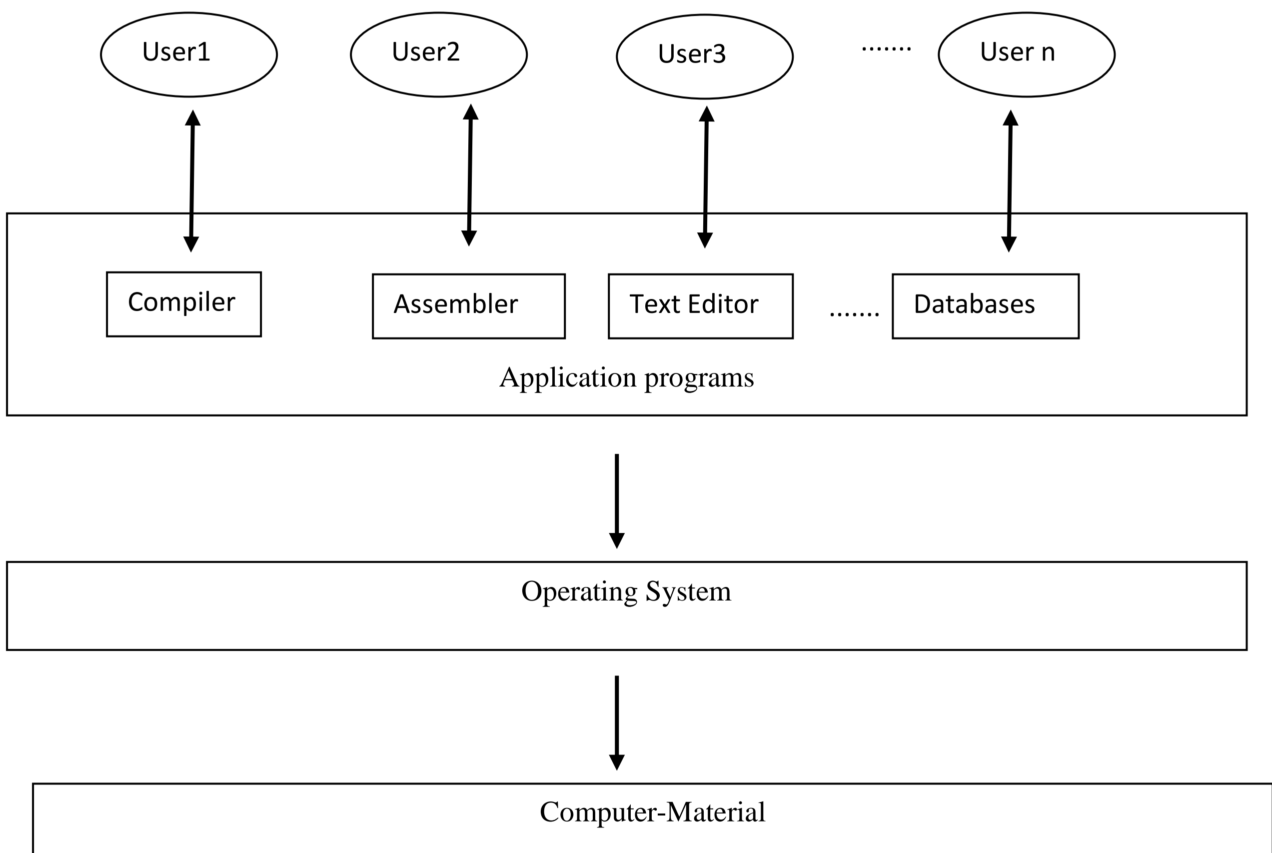



FIG I.1 The layers of a computer system.

### I.2.2 Operating system

### I.2.2.1 Definition

The operating system is a set of programs that act as an intermediary between the user and the computer. Therefore, the user has a simplified view of the computer allowing him to use it more easily. The operating system provides programmers with a virtual machine that is easier to program than the real machine. It provides them with services (manipulation of input/output devices, files, etc.) that programs can request via a simple interface called a system call. The operating system is also responsible for managing computer resources. It allows the equitable sharing of these resources between the different users. The operating system is responsible for running user programs. It controls their execution and avoids the damage that can be caused by execution errors.

### I.2.2.2 Functions of an operating system

A modern operating system must ensure the following functions:

*a) Processor and process management*

A process is a running program. For it to be run, a process must be housed in memory. It must have all the resources necessary for its execution (CPU, I/O devices, files, etc.). The operating system is responsible for:

• creation and destruction of user and system processes

• the allocation of the processor to the processes (scheduling or scheduling)

• the suspension and resumption of processes if necessary

• provide synchronization and communication mechanisms between processes

*b) Central memory management*

Main memory is the only memory directly accessible by the processor. It is also a shared zone between the processor and the majority of the input/output devices. For example, if the processor wants to save data to disk, it puts it in a special area of memory. The disk controller then takes care of accessing this area and physically recording this data on the disk. A modern operating system can load multiple programs into memory. Some programs require more memory than the capacity of the target computer. Memory must therefore be managed

and used in such a way as to satisfy all the needs of the programs. The operating system is responsible for:

• keep track of each allocated memory area and the associated user

• choose the process to which to allocate or remove a memory area.

• emulate a large memory using secondary memories (virtual memory).

*c) File management*

To save information, the computer uses several secondary memory devices (magnetic disk, optical disk, tapes, etc.). Each of these devices has its own characteristics and physical organization. The operating system must provide a single logical view of these media. This logical view is the file. The operating system is responsible for:

• Creating and deleting files

• Creating and deleting directories

• Provide programs with primitives (procedures) to manipulate files and directories

• Associate files with physical devices

*d) Entry/exit management*

One of the main purposes of operating systems is to be able to hide the complexity and peculiarities of input/output devices from users. This complexity is bequeathed to the input-output management subsystem which is composed of the following elements:

• A component responsible for communicating between the device and the central memory

• A generic interface common to all input/output devices

• A device manager (driver or driver) specific to each device

Only the driver is aware of the specifics of the device it manages.

*e) Protection and security*

If the operating system is multi-user and allows the concurrent execution of several processes, it must ensure the protection of some vis-à-vis the others. A process must not

access any space outside its authorized zone. No process should monopolize the processor for longer than allowed. Input/output devices must not be directly accessible by users to avoid damaging them. An operating system with protective devices is still fallible. A malicious process (virus, etc.) cannot damage the data of other users but can damage that of its owner. The operating system must have security devices that defend against external and internal attacks.

### I.2.3 Evolution of operating systems

Starting from the fact that an operating system is designed for a very specific machine, it is logical that operating systems evolve with generations of computers. Without advances in optics and chemistry there would be no integrated circuit or processor, essential to modern microcomputers. It took integrated circuits and the first real computer to have a real operating system. Originally we started by developing software to automate certain tasks, to compile programs and to perform input-output. It was only with multiprogramming that we really felt the need for an operating system capable of supporting the machine and relieving the programmer of it.

#### I.2.3.1 Bare machines: 1945-1955

Calculating machines were built using electron tubes. These machines were huge, expensive, very unreliable and much slower because the cycle time was measured in seconds. They had no operating systems. Programs were written directly in machine language: they were loaded into memory, executed and debugged from a control desk. The mode of operation was to reserve a time slice for each programmer (only one program in memory). In early 1950, the procedure improved with the introduction of punched cards. The obvious problems were due to the mode of operation: uneconomical, expensive hardware, long program input times, and tedious debugging.

#### I.2.3.2 Batch processing: 1955-1965

The machines were built using transistors and had magnetic tape drives. They were more reliable but still huge (locked in air-conditioned rooms). Programs were written in Fortran or assembler on punched cards. The mode of operation used, shown in FIG I.2, was batch processing which consists of:

- Transfer work to magnetic tape.

- Mount the tape on the tape drive.

- Load into memory a special program (the ancestor of operating systems) which reads then executes, one after the other, the programs on the tape. The results are retrieved to another tape after the entire batch has finished running.

- Print the results.

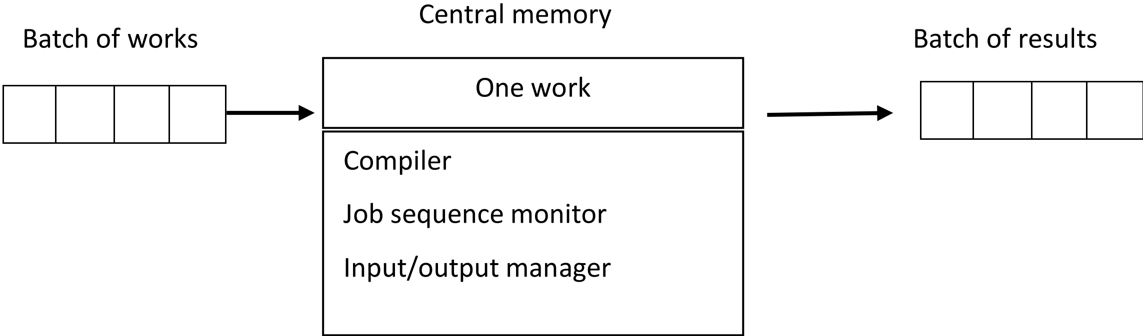



FIG I.2 Batch processing and memory organization.

This mode of operation required two machines, the most powerful of which was reserved for calculations and the other, less expensive, took care of slow peripherals. The problem is that the processor was sitting idle during input and output (I/O) operations, as seen in FIG I.3. The waiting time for the results was too long and in addition there was no interaction with the user.

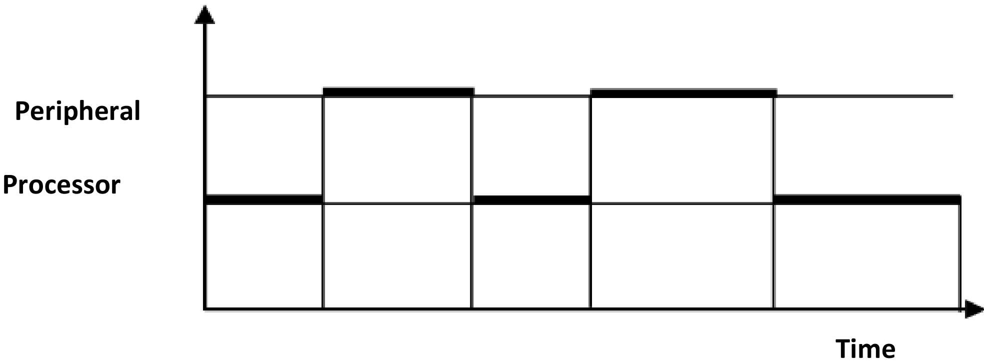



FIG I.3 CPU and Peripheral Usage.

### I.2.3.3 Multiprogramming and batch processing: 1965-1980

The introduction of integrated circuits in the construction of machines has made it possible to offer a better cost/performance ratio. The advent of disk drives, which offer random access and large storage capacities, has contributed to a series of system developments, including the ability to transfer jobs to disk as soon as they arrive in the room.

machine. This technique is called spooling (Simultaneous Peripheral Operation On Line) and is also used for outputs. This notion of spooling or batch processing remains in modern operating systems. All disk-resident jobs waiting to run are kept in the input job pool. The memory is organized in a set of partitions (FIG I.4). Each partition can contain at most one job. The operating system also resides in a partition. If there is a free partition and jobs in the input pool, the operating system chooses a job and then starts loading it into memory. It stores several jobs in memory and manages the sharing of the processor between the different jobs loaded in memory (multiprogramming). The processor is allocated to a job until it requests an I/O (first come, first served). When a job requests an I/O (waiting for an I/O to complete), the processor is allocated to another job in memory (the next one). At the end of an I/O, an interrupt occurs and the operating system regains control to process the interrupt and initiate or continue execution of a job. As soon as a job completes, the operating system can start loading a new job from disk into the newly freed partition.

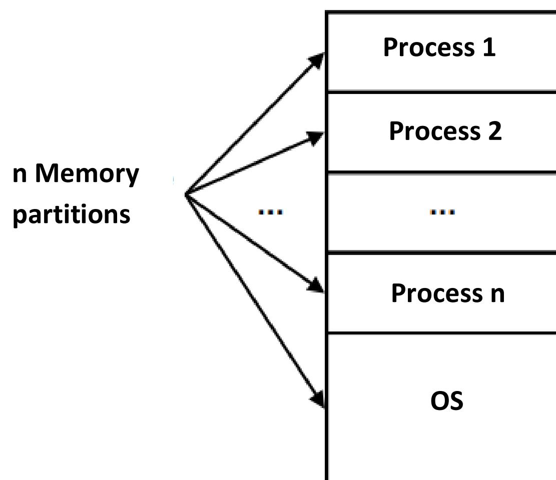



FIG I.4 Memory partitions.

Suppose three jobs A, B and C. In a multiprogrammed system, three activities can therefore be carried out in parallel: 1. Loading job C into memory, 2. Executing job B and 3. Outputting the results of the job A. Multiprogramming requires control circuits to protect each job from the intrusions and errors of others. Computers of that time had these circuits.

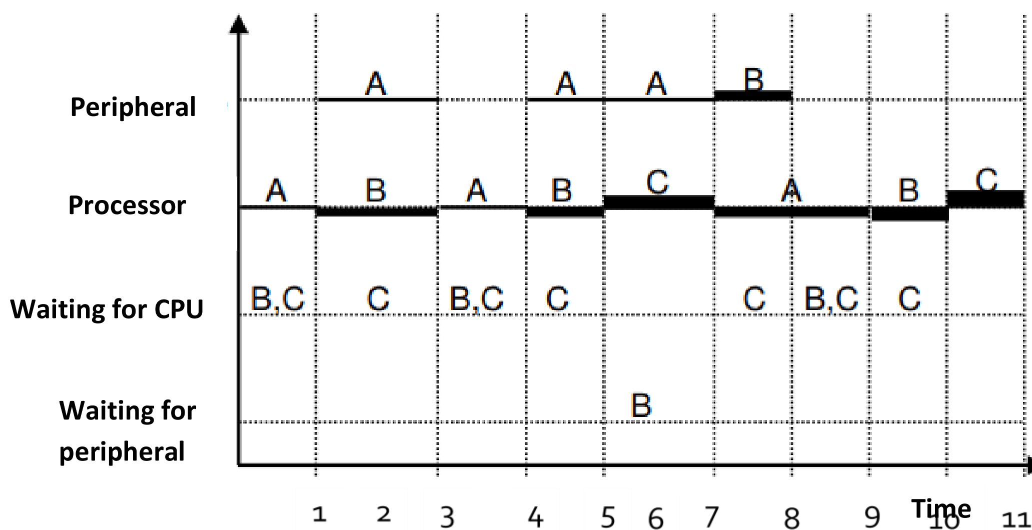



FIG I.5 Processor switching.

Consider the jobs A, B and C in memory, which are shown in FIG I.5. The execution of A is started first then when it requests an Input/Output (I/O), the processor switches to B. At the end of the I/O requested by A, the processor suspends the execution of B to switch to A. Assume that A has priority. After some computation time, A requests an I/O again; causing the processor to switch to B. While A's I/O is executing, job B requests an I/O. It therefore puts itself on hold because the device is busy. The processor then switches to job C. When the execution of the I/O requested by A has been completed, the processor switches to A and the processing of the I/O request from job B is started by the I/O device.

The desire for faster response time and interactive operation introduced the technique of time-sharing (time-sharing or multi-user systems): several users can connect to the machine through their terminals and work at the same time. The processor is allocated, in turn, for a certain time to each of the jobs waiting to be executed. At the end of this time, even if the current job has not ended, its execution is suspended. The processor is allocated to other work. If several users launch their programs from their terminals simultaneously, this mode of operation gives the impression that the programs are running in parallel. The case of three jobs A, B and C is shown in FIG I.6 where the three upper lines correspond to the jobs and the lower line shows the jobs that request the CPU and are inserted in waiting queue. The alternation through CPU allocation allows acceptable response times for each user. FIG I.7 shows another example using the pattern shown in FIG 1.5.

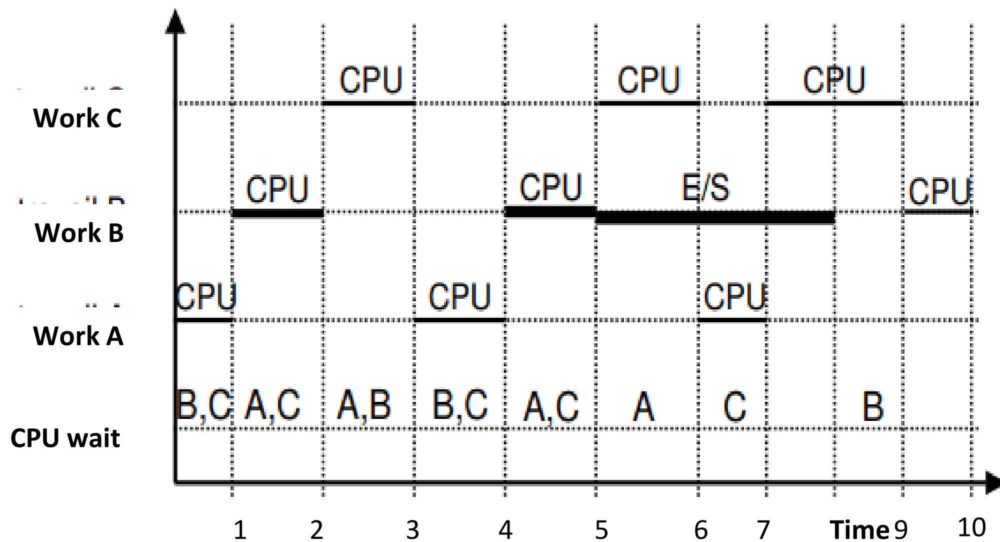



FIG I.6 Time sharing : Model 1

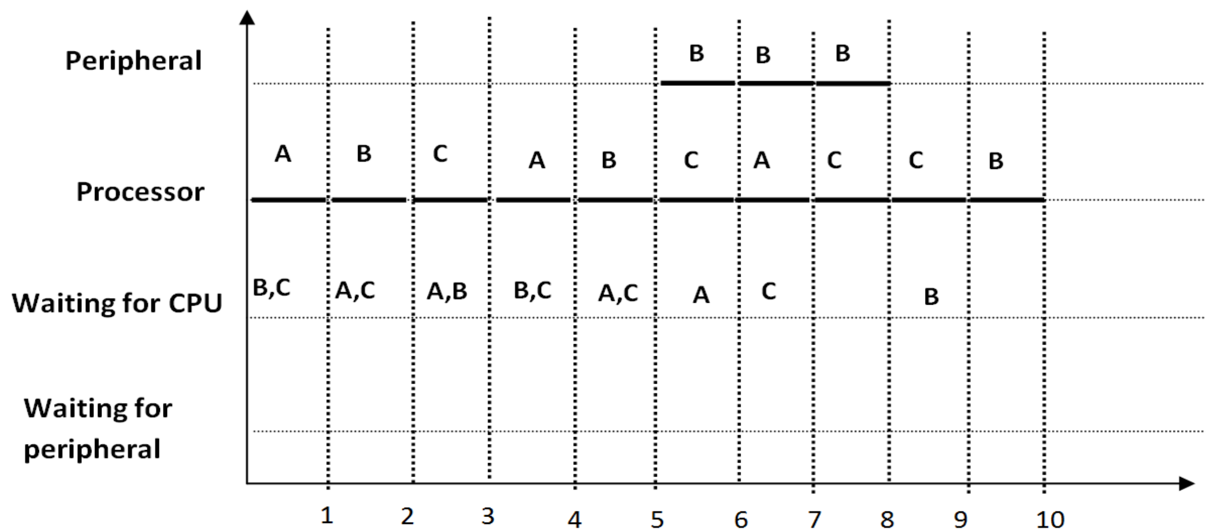



FIG I.7 Time sharing : Model 2

### I.2.3.4 The fourth generation: 1980-1990 (Personal computing - Network operating systems - Distributed systems)

These single-user operating systems did not emphasize CPU usage or protection. Their goal was to provide a user-friendly interface and quick response. Then, the functionalities of mainframes were gradually transferred to microcomputers.

*Networking*: Networks of personal computers that run under network operating systems make it possible to connect to a remote machine and transfer files from one machine to another. These operating systems require a network interface, low-level control software, and programs that allow remote login and access to files on different machines.

*Distributed Operation:* Networks of computers running distributed operating systems appear to users as a single-processor machine, even when this is not the case. The distributed operating system manages and controls all the components of all the connected computers (processors, memories, disks, etc.).

*Multiprocessor Systems:* These systems consist of multiple processors attached to the computer bus. They are characterized by their processing capacity and their reliability: the failure of a processor will not stop the system.

*Real-time Operating System:* These are systems specialized in the control of industrial equipment or in the control of processes where time plays a critical role (strict or flexible time constraints to be respected).

#### I.2.3.5 The fifth generation: 1990 to the present day (Miniaturization, Embedded systems)

Currently, a wide range of electronic equipment (PDAs, Cell Phones, Televisions, washing machines, etc.) have a computer architecture and have processors that run programs. These devices with special tasks are equipped with specific operating systems called embedded operating systems. The expansion of embedded systems continues to grow these days. The equipment is either autonomous or interconnected in networks (IR, Bluetooth, WLAN, LAN, etc.) to make our daily life easier. A home system can, for example, control the temperature, the lighting, the alarm and even the coffee machine. Soon, the refrigerator will be able to order milk online when it realizes that there is not enough left.

## I.3 USER/SYSTEM INTERACTION

For a user, the operating system appears as a set of procedures, too complex for him to write them himself. The system call libraries are then procedures made available to programmers. So a C/C++ program can use Unix/Linux system calls like open(), write() and read() to perform low-level I/O. The shell is a user/system interface. It is available in all systems. It is launched upon connection to the system and prompts the user to enter a command. The command interpreter fetches and then executes the command by a combination of system calls and tools (compilers, linkers, etc.). It displays results or errors, then waits for the next command. For example, the following Unix shell command displays the contents of a file named archive: cat archive on the screen. The introduction of graphics in user interfaces has revolutionized the world of computing. The GUI was made popular by

Apple's Macintosh. It is now available for most machines. For a user, the operating system appears as a set of procedures, too complex for him to write them himself. The system call libraries are then procedures made available to programmers. Thus a C/C++ program can use Unix/Linux system calls like open ( ), write() and read ( ) to perform low-level I/O.

The shell is a user/system interface. It is available in all systems. It is launched upon connection to the system and prompts the user to enter a command. The command interpreter fetches and then executes the command by a combination of system calls and tools (compilers, linkers, etc.). It displays results or errors, then waits for the next command. For example, the following Unix shell command displays the contents of a file named archive: cat archive on the screen. The introduction of graphics in user interfaces has revolutionized the world of computing. The GUI was made popular by Apple's Macintosh. It is now available for most machines.

## I.4 SYSTEM CALLS

In general, processors have two modes of operation: Supervisor mode (kernel, privileged or master): for the operating system, where all instructions are authorized and user mode (slave): for user programs and utilities, where some instructions are not allowed.

These operating modes ensure the protection of the operating system against intrusions and errors. This is not the case with single-user systems like MS-DOS or MacOS which have only one mode of operation: user mode. They are unprotected and therefore unreliable. A system call consists of a software interrupt (TRAP instruction) whose role is to activate the operating system. Its purpose is to: change execution mode to go from user mode to master mode, retrieve the parameters and check the validity of the call, launch the execution of the requested function, retrieve the value(s) back and return to calling program with return to user mode. Table I.1 lists some Unix system calls that conform to the Posix standard. System calls manipulate various objects managed by the operating system. Processes and files are the most important of these objects.

### I.4.1 Processes

A process is a running program. It is composed of an executable program (code), a ordinal counter (CO), a set of data, an execution stack and other registers and information necessary for execution. System calls are used in particular to create and stop processes. A process can create one or more child processes which, in turn, can create child processes in a

tree structure. Processes can synchronize and communicate with each other. Currently, the concept of lightweight processes is increasingly being used. Threads are a means of refining and dividing up the work normally associated with a process.

When an application is launched, the process that is created by the operating system executes the instructions of the program in user mode. At times, system calls are made that require the execution of instructions (and access to data) that belong to the operating system. To do this, the same process then switches to supervisor mode. It should be noted that it is indeed the same process, even if it executes operating system instructions. It can be considered that operating system routines are executed on behalf of the process in question. FIG I.8 illustrates this principle. FIG I.9 illustrates in more detail what typically happens during a system call. In this example, an application executes fork ( ). This is a function from a library that was compiled with the application. This function prepares the ground for executing the real system call, i.e. a TRAP assembly instruction which takes as a parameter a pointer whose value is here indicated by SYS_FORK.

Table I.1 Some Posix system calls.

| | |
|---|---|
| fork | Create process |
| waitpid ; wait | Wait for a process to terminate |
| execve | Run another program |
| exit | End execution |
| open | Create or open a file |
| close | Close a file |
| read | Reading data |
| write | Write data |
| Iseek | Pointer in a file |
| stat | Get status of attributes |
| mkdir | Create directory |
| rmdir | Delete a directory |
| link | Links to a file |
| unlink | Delete a file |

| | |
|---|---|
| mount | Mount a file system |
| umount | Unmount a file system |
| chdir | Change directory |
| chmod | Change access permissions |
| Kill | Signals |
| time | Get the time |

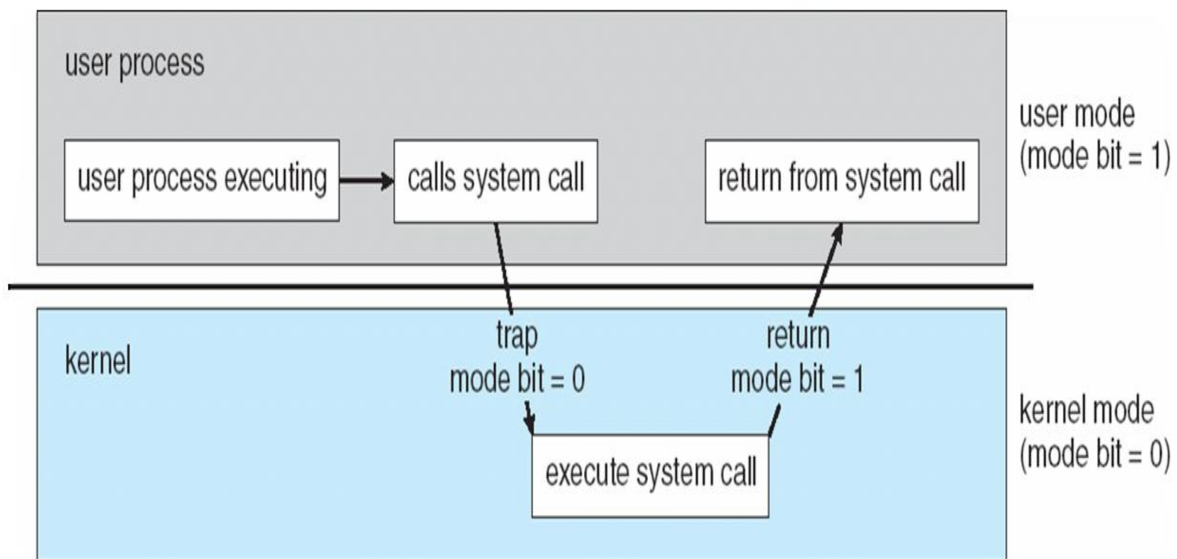



FIG I.8 Alternation of execution modes of a process.

Essentially, this instruction first puts the processor into supervisor mode, then searches in a table (which was created when the operating system was loaded), at the position indicated by SYS_FORK, the address of the routine that needs to be executed. Then the ordinal counter is positioned at this address and continues the execution of the process, this time with instructions that belong to the operating system. When the requested service is finished, we return to the application, while restoring the user mode.

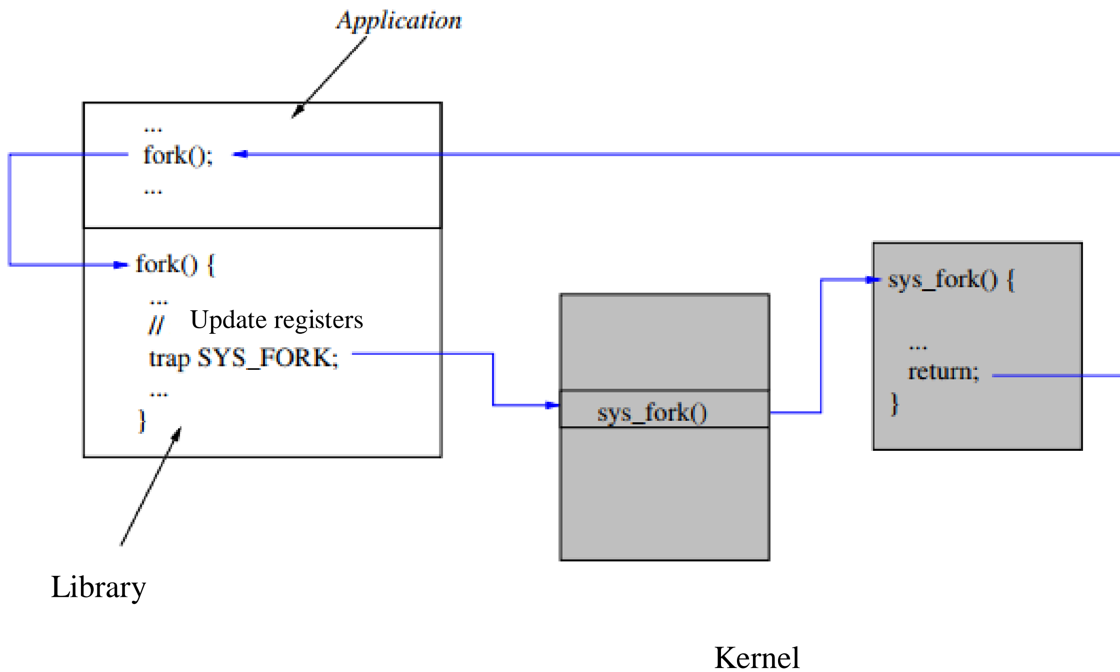



FIG I.9 Steps of a system call.

### I.4.2 Files

A file is a set of data recorded in order to be read and processed by computer. Files can be grouped into directories. A directory can contain either files or other directories in a tree structure. Access to files is done by specifying a path, i.e. the list of directories to cross to access the file. A path is absolute if the starting point is the root directory. A path is relative if the starting point is the current directory. System calls are used to create files and directories, as well as to delete, open, read and modify them.

## I.5 STRUCTURE OF AN OPERATING SYSTEM

### I.5.1 Layered structure

The operating system is structured in layers. Each layer uses the functions of the lower layers. The main difficulty is the definition of the different layers. For example, it can be organized into five layers, as shown in FIG I.10 :

- At the lowest level is the kernel, the interface between hardware and software. It is responsible, using the functions provided by the hardware, to manage the CPU, the interrupts

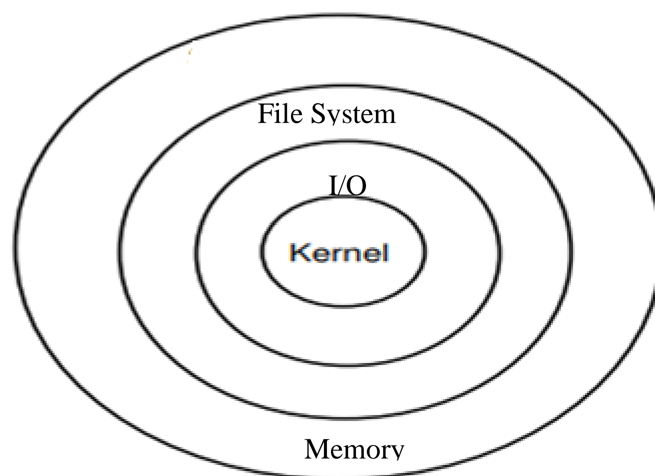
and the processes (the communication and the synchronization). It must reside entirely in memory.

- At the second level, we find the memory manager which is responsible for sharing memory between processes waiting to be executed.

- At the third level, we have the input/output management module which manages all the peripherals (keyboard, screen, disks, printers, etc.).

- At the fourth level, we find the file manager which takes care of disk space management, file manipulation while ensuring data integrity, file protection, etc.

- At the fifth level, we have the resource allocation module which is responsible for ensuring proper use of resources; to record and provide statistics on the exploitation of the main resources; to create new processes and assign them a level of priority: to allow each existing process in the system to obtain the necessary resources within reasonable time limits; to mutually exclude processes that request a non-shareable resource and to avoid deadlock situations.




Ressources allocation

FIG I.10 Layered structure.

### I.5.2 Monolithic structure

Operating systems under a monolithic structure (FIG I.11) are a set of -almost- same-level procedures: a main procedure that calls the required service procedure, service procedures that perform system calls, and a set utility procedures that support service procedures, such as retrieving user program data. Unix and Linux are examples of monolithic systems.

### I.5.3 Microkernel

A more modern architecture than the monolithic one is the micro-kernel architecture (see FIG I.11, FIG I.12) used in MACH5/HURD, Minix and NT. The main attribute that distinguishes micro-kernels from monolithic kernels is the implementation of their respective architectures in supervisor mode (kernel mode) and in user mode (user mode). The monolithic architecture implements all operating system services (device controllers, virtual memory, file system, networks, etc.) in the supervisor mode domain of the CPU. On the other hand, the micro-kernel architecture makes a division between the services of the Operating System, dividing them into 'high-level' implanted in the domain of the user and 'low-level' implanted in the space of the user. supervisor mode.

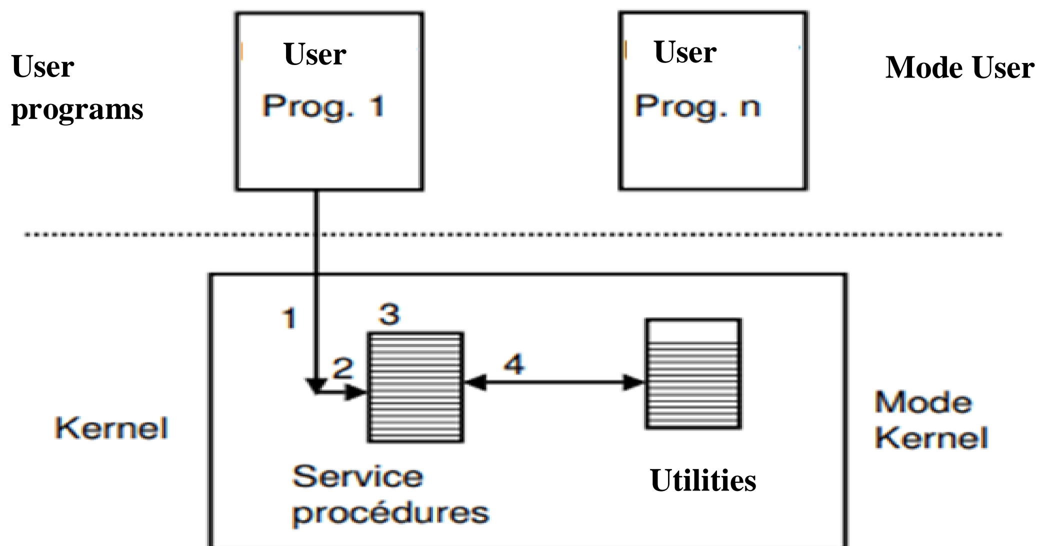



FIG I.11 Monolithic structure. (1) System call (user mode/kernel mode). (2) Verification of parameters. (3) Call of service procedure. (4) Service procedure call utilitaries, and then return to user mode.

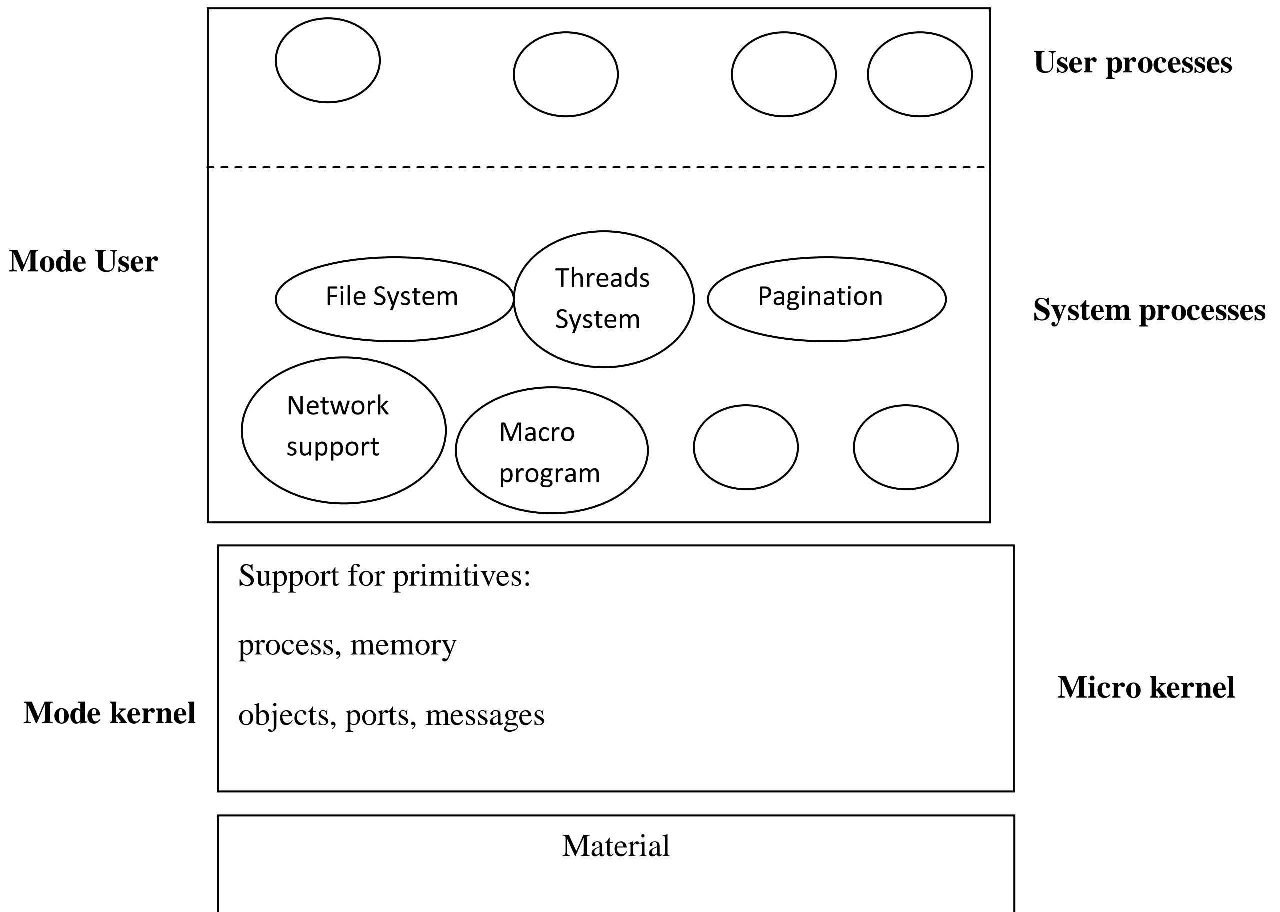



FIG I.12 Micro-kernel structure.

# Chapter II

## INTRODUCTION TO THE SYSTEM LINUX/UNIX

### II.1 INTRODUCTION

The Unics system (uniplexed Information and Computing Service) have been created at Bell's AT&T Laboratories in 1969 by Ken Thompson, and modified and baptized by Brian Kernighan. It was a scaled down version of Multics (Multiplexed Information and Computing Service). The name Unics is after changed to Unix, and from that moment the system Unix began a long road of technical development.

### II.2 BRIEF HISTORY

The Unix system had a real success, when it was rewritten in the language C in 1973 by Denis Ritchie and Thompson1. The University of California at Berkeley obtained a copy of version 6 of the Unix system. AT&T nd Berkeley separately made many changes and improvements to the Unix system (**System V** of AT&T and **4.4 BSD** of Bekerley). A simplified vision of the evolution undergone by Unix is shown on FIG II.1. The projectPosix (Portable Operating System UnIX) standardization of the Unix system has made it possible to develop a standard which defines a set of procedures. Any Unix system conforming toPosixprovides these standard procedures or system calls. These procedures constitute the standard Unix library (FIG II.2). Any software written using only the procedures of the standardPosix should work on all systems Unix conforming to this standard. A free version of Unix is called Linux (source code available). It was created by Linus Torvalds in 1991. Subsequently, a large number of programmers contributed to its accelerated development. Originally designed to run on machines with the 80x86 processor, Linux has migrated to

several other platforms. The Linux operating system can be downloaded from several sites, for example: http://www.linux.org.

Free BSD is another free version of Unix. It is an operating system for compatible x86, DEC Alpha, and PC-98 architectures. It is derived from BSD-Unix and developed by a large community of individuals (http:// www.freebsd.org).

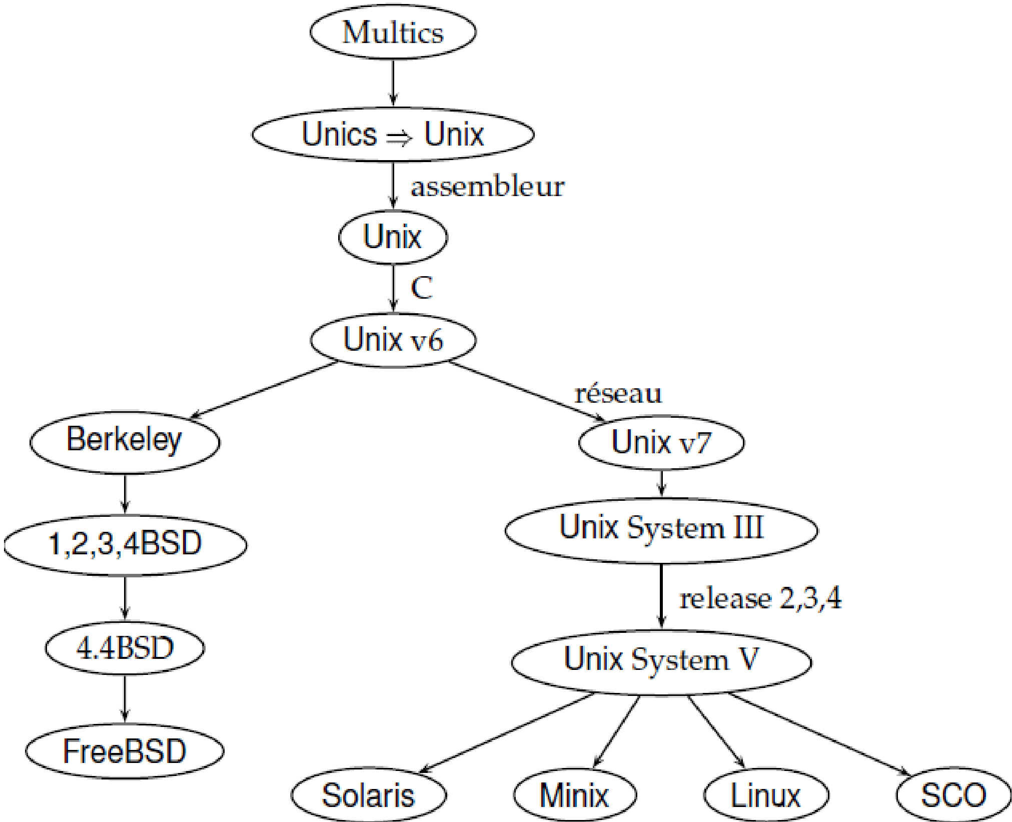



FIG II.1 Evolution of Unix.

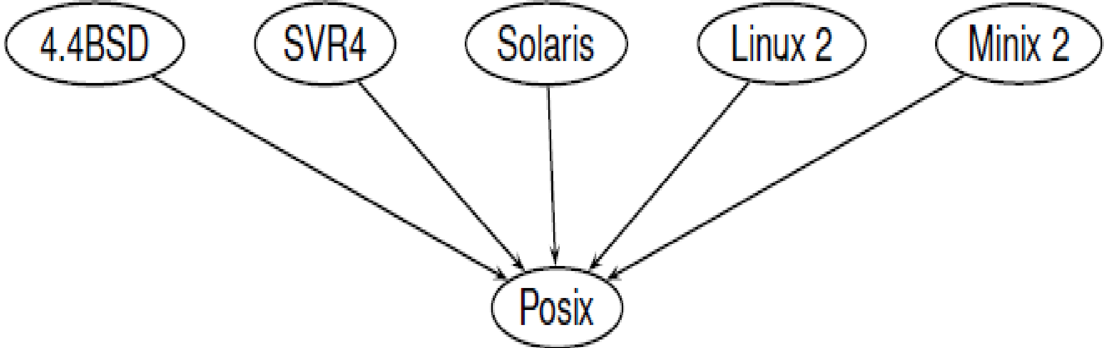



FIG II.2 Standard Posix.

## II.3 FEATURES

Unix is an interactive and multi-user (time-sharing multiprogrammed) system. Several users can have the computing power of the system at the same time. The Unix system is responsible for controlling and managing the use of resources by allocating them, in turn, to different users. It allows the creation, communication and synchronization of processes. Unix is an open operating system, portable and available on different platforms. Linux is also free and we have the permission to study and modify the source code.

### II.3.1 Structure of a Unix/Linux system

A computer system under Unix/Linux is made up of layers of software, as shown in FIG II.3. The operating system, called **core** or kernel, manages hardware and provides programs with a system call interface. The LINUX architecture and the kernel of BSD are shown in Table II.1 and Table II.2 respectively. System calls allow programs to create and manage processes and files. Each system call corresponds to a library procedure that the user can call (standard library). Each procedure is responsible for placing the parameters of the corresponding system call in a predefined place such as the registers of the processor, and for causing a **software interrupt** (instruction TRAP) to pass from **user mode** at **kernel mode** and thus activate the operating system. The purpose of the library procedure is to hide the details of the statement TRAP and make procedure calls appear as ordinary procedure calls. For example *read (f, b, 50)* will be used to read 50 characters in temporary storage or buffer b, from a file f currently open. When the operating system takes control due to TRAP, it checks the validity of the parameters and in this case performs the requested processing. At the end of the processing, it places a status code, indicating whether the processing succeeded or failed, in a register, then returns control to the library procedure with return to user mode. The library procedure, in turn, hands control back to the caller by passing it the status code. Users can develop programs and invoke system calls, thanks to the standard library and a set of utilities (shells, editors, compilers, etc.) provided with the Unix/Linux system. For example, to run the system call READ, a program C/C++ can call standard library procedure read() which will make the actual system call READ.

**Notice:** The real system calls run in **kernel mode**, This which ensures the protection of the operating system against intrusion attempts and errors.

### II.3.2 Start of a session

When a user asks to connect, the system prompts him to enter his user name (code) and password. If these data are correct, the system opens a work session and launches the **command interpreter** (shell process) which displays on the screen, immediately after its initialization, an invitation then waits for user orders. Depending on the shell used, this can be a simple symbol: $ or, the prompt may show the name of the machine. In our case one of the machines is called leibnitz:

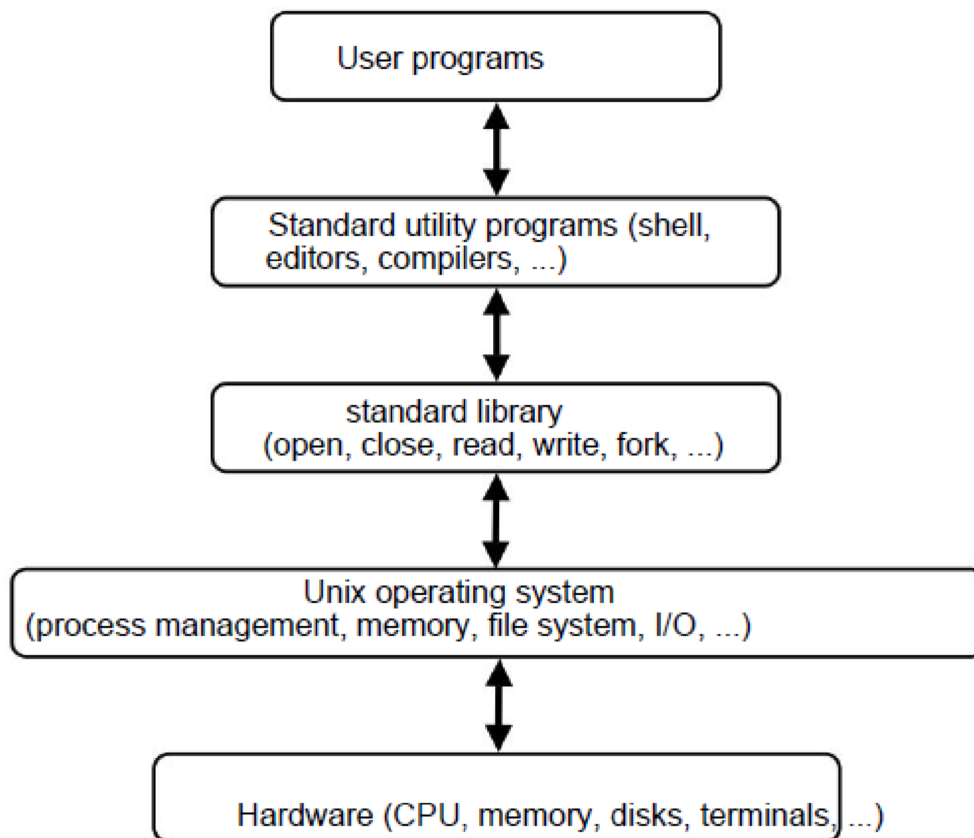



FIG II.3 Architecture of a Unix system.

*leibnitz>*

When the user enters a command, the shell checks if it is correct, then creates, if it is, a process which has the task of executing the command. For example the command:

*leibnitz> cp source destination*

will copy the file source in the file destination. The shell waits for the termination of the created process before prompting the user to enter a new command.

### II.3.3 Command interpreters

There are several popular command interpreters or shells for the Unix system. Among them are:

- The Bourne shell:sh
- The Korn shell:ksh
- The C-shell:csh

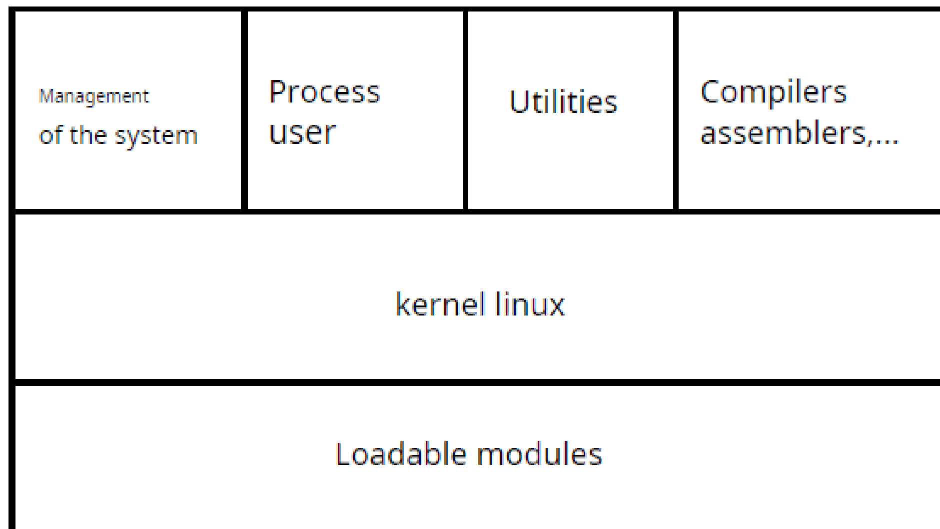
Korn's shell encompasses Bourne's. Many commands are common to all three shell's.

Table II.1 Unix BSD kernel architecture.

<table>
<tr><td colspan="5">System calls</td><td colspan="3">Interrupts and traps</td></tr>
<tr><td colspan="2">Terminals</td><td>Sockets</td><td>File naming</td><td>Map fault</td><td>Ping page</td><td rowspan="2">Signals</td><td rowspan="2">Creation and termination process</td></tr>
<tr><td rowspan="2">Raw tty</td><td>tty</td><td>Network protocols</td><td>file system</td><td colspan="2">Virtual memory</td></tr>
<tr><td>Disciplines line</td><td>Routers</td><td>buffer cache</td><td colspan="2">Page cache</td><td colspan="2">Process scheduling</td></tr>
<tr><td colspan="2">Devices to characters</td><td>Network device controllers</td><td colspan="3">Disk Device Controllers</td><td colspan="2">Dispatch process</td></tr>
<tr><td colspan="8">Material</td></tr>
</table>

The shell bash,written by Brian Fox, part of the project **GNU**, recursive acronym of GNU's **NOT** Unix, started by Richard Stallman in 1984 (http ://www.gnu.org). The GNU project wanted to create a complete, Unix-compatible but freely distributed system. GNU had developed several programs like the compiler gcc, the editor emacs, make, tar, the X-Windows graphical interface. In 1991 the Linux kernel joined the project and it was called **GNU/Linux**. The shell bash (acronym of Bourne-Again SHell) is compatible with the standard Posix, and also compatible with the shellsh. It incorporates many of the characteristics of the Korn shell ksh and the C shell csh.

Table II.2 Architecture of Linux.




### II.3.4 Useful Unix/Linux commands

In table II.3 we show some commands Posix widely used in Unix/Linux.

Table II.3 Commands Posix from Unix.

| Order | Description |
|---|---|
| who | Display the list of connected users |
| whoami | Display the user of the current session |
| date | Display the date |
| ps | Display the list of user processes Stop a |
| kill | process |
| passwd | Create or change password Display the |
| pwd | name of the working directory |
| mkdir | Create a directory |
| CD | Change working directory |
| cat | Merge a list of files and display the result |
| head | Displaythe beginning of a file |
| grep | Show lines of referenced files that contain a |
| | given character string Count the number of |
| | words, lines or characters. |
| Sleep | Sleep for a certaintime (seconds) |
| find | Find a file |
| ls | Display the contents of a directory |
| pc | Copya file to another |
| ln | Add another name to an existing file |
| mv | Rename a file |
| rm | Delete a file |
| rmdir | Delete a directory |
| chmod | Change the protections of a file |

**man**

*man [-s section] word*

The command man provides online help. It displays the information corresponding to the given title, appearing in the standard Linux manual. This manual is made up of 8 sections.

**Example 1.** The command :

*leibnitz> man chmod*

select section 1 of the commands, and: associated one input and two standard outputs:

*leibnitz> man -s 2 chmod*

- The standard input: the keyboard

selects section 2 of system calls. Obviously if you want to understand how man works, you have to type:

*leibnitz> man man*

**Sort**

The command sort performs the reading of the lines coming from the terminal (until the user types CRTL-D indicating the end of file), sorts the lines in alphabetical order and displays the result on the screen.

**Exemple 2.** Use of sort

leibnitz> sort

*Martin 16 HULL*

*Robert 17 Ottawa*

*Catherine 15 Montréal*

*^D*

*Catherine 15 Montréal*

*Martin 16 HULL*

*Robert 17 Ottawa*

*leibnitz>*

**I/O redirection**

Each process created (command or program in execution) is associated with an input and two standard outputs:

- The standard input: the keyboard

- The standard output: the terminal screen
- The standard error output: the terminal screen

The shell allows you to modify or make the **redirect** Inputs/Outputs of a process.

**Redirect standard input**

We redirect with the < operator:

*command < file*

**Example 3.** This command sorts the source file, then displays the result on the screen:

*leibnitz> sort < source*

**Redirect standard output**

You can redirect standard output with the > or » operator:

*command > filename*

*command >> file_name*

**Example 4.**The list of users connected to the system is retrieved from the file users with the following command. If the file already exists, there will be an error:

*leibnitz> who > users*

If the file already exists, the list is inserted at the end:

*leibnitz> who >> users*

**Example 5.**This command reads the lines from the terminal then copies them into the file destination (until the user types CRTL-D,to indicate end of file):

*leibnitz > cat > dest*

**Redirection of standard outputs (error and normal)**

(command) >& filename

(command) >> & filename

**Example 6.** Output redirection.

*leibnitz > (ls ggg) > & er leibnitz*

*> cat er*

*ggg: This file or directory does not exist leibnitz> (lhiumo )*

*>>& er*

*leibnitz > cat er*

*ggg: This file or directory does not exist lhiumo: Command not found*

*leibnitz>*

**Sequential sequence of commands**

The shell allows several commands (processes) to be launched sequentially with the operator; semicolon :

*command_1; command_2; ....; order*

**Example 7.** This command is used to display the working directory then its contents. Then it will create the directory (course) in the working directory. It displays the contents of the working directory, then it changes to the working directory (course),displays the new working directory (course),then it returns to the parent directory, and finally displays the new working directory:

*leibnitz> pwd; ls; mkdir course; ls; cd course;pwd; cd ..; pwd*

**Parallel sequence of commands**

Similarly, multiple commands can be run in parallel with the | and to direct the output of one towards the input of another. We call this **pipelines**:

*command_1 | command_2 | ... | order*

**Example 8.** The following command (see FIG II.4) creates two processes cat And sort. cat merge files src1 and src2 and send the result to sort ; sort sorts the received data before storing it in the file destination. Both processes run in parallel:

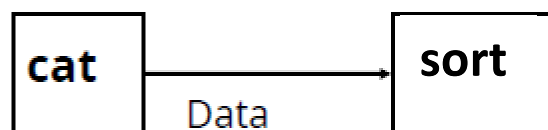



FIG II.4 Parallel chaining of cat and sort commands.

*leibnitz> cat src1 src2 | sort > dest*

**Example 9.** Chained use.

*leibnitz> grep ter *.t | sort | head -20 | tail -5 >file*

The command grep send to sort lines containing the string "ter" of all files ending with .t,Then sort sorts the lines received and forwards them to head, which in turn sends the first 20 totail. Finally, the command tail writes the last five to the file file

**Example 10.**Outputs (error or not) from compiling/linking the program are redirected to the process inputmore.The latter displays the results gradually on the screen:

*leibnitz> gcc prog.c | & more*

**background tasks**

The shell allows you to launch a command in the background using the & operator. In this case, it does not wait for the end of the execution of the order :

**Example 11.** This command will do the recursive search from the root directory for all files with name bin. Absolute paths of found files are stored in the file result:

*leibnitz> find / -name bin -print >result &*

## II.4 FILE PROTECTION

Like any multi-user system, Linux has mechanisms that allow the owner of a file to protect its contents. The owner is the user who created the file. To allow file sharing and facilitate teamwork, Linux defines the notion of user group, and any user belongs to at least one group. The access rights (permissions) to a file are defined by its owner.

### II.4.1 File protection

Each file is associated with a set of indicators specifying the access rights to the file. For each file there are three types of users:

- the owner of the file,
- the members of the group that owns the file,
- other system users.

For each file and by type of user there are three main modes:

- write permission (w)
- read permission (r)
- execute permission (x)

In addition to these nine bits (rwx rwx rwx), Unix/Linux defines three other bits of permission: SUID, SGID, t which we will present later. In Linux there are different types of files:

- ordinary file ( )
- directory file (d)
- special file: device accessed in character mode (c)

● special file: device accessed in block mode (b)

● named pipe (p)

● symbolic link (l)

● socket (s)

Thus, each Linux file is associated with 10 attributes (1 to designate the type, and 9 protection attributes, 3 for the owner, 3 for the group and 3 for the other users). All information about a file is obtained using the ls -l command.

**Example:**

```
xstra> ls -l /etc/passwd
 rw -  r    r    1 root bin 2055 Jul 28 18:03 /etc/passwd
```

Others permission
Group permission
Owner permission
File type

The string rw - r - - r - - represents the protections in the /etc/passwd file. The first character represents the file type (ordinary in this case). The next three characters indicate the access permissions of the owner (rw -), the next three are those of the group (r - -) and the last three (r - -) are the access permission of other users. The next field represents the number of links on this file (one in the example). The next two fields indicate the user (root) and group (bin) owners of the file. Then appears the size of the file in bytes, followed by the date and time of the last modification of the file. At the end appears the name of the file.

Here are two examples of file protection:

c rw - rw - r - - Special character file: read and write for the owner and for the group, and read for the others (for example terminals).

- rwx r-x r - - Ordinary file: read, write and execute allowed for owner, read and execute for group and only reading for others. It is therefore impossible for group members and other users to write in this file.

### II.4.2 Modification of file permissions

The protection of a file can only be changed by the owner using the chmod (CHange MODe) command. There are two ways to use this order. The first (the oldest) uses the description of the protections by an octal number.

**Example 1**

rwx rw - r - x is represented by the octal number 765. Indeed, one letter is equivalent to 1, a dash to 0. So we have rwx rw - r - x = 111 110 101 = 765

**Example 2**

This example will modify the protection of the toto file as follows: r - x rw - rwx allowing read and execute access to the owner, read and writing to the group and unrestricted access to others (very rare case).

*xstra> chmod 567 toto*

*xstra> ls -l toto*

*r - x rw - rwx 1 xstra staff 55 Jul 20 17:01 toto*

*xstra>*

The second mode of use of chmod, the symbolic mode, allows an absolute or relative description of permissions, as follows:

*chmod [who] op [permission] fichier*

where

who is a combination of letter u (user=owner), g (group), o (other=other) or a (all=all) for ugo, op + adds an access right, - deletes a permission of access and = to assign a right in an absolute manner (all other bits are reset), permission r (read=read), w (write=write), x (execute).

As examples :

*chmod u - w* file removes the write permission to the owner.

*chmod g +r* file adds read permission for the group.

*chmod ug =x* file execute-only access for owner and the group, no modification for the others.

### II.4.3 Permission at the file creation

The protection of a file, as well as the name of the owner (yours) and the name of the group to which you belong, are established when it is created and can only be modified by its owner. The *umask* command allows you to define a protection mask for files (and directories) when they are created. This command is usually found in the *.bashrc* file, but it can be run at any time. The mask is expressed in base 8.

**Examples**

*xstra>umask 022*

The value 022 is subtracted from the permanent permission (111 111 111):

*111 111 111 <= permanent permission*

*000 010 010 <= we remove the bits we don't want*

*111 101 101 => 755*

umask 022 makes it possible to create directory files whose protection is rwx r- x r - x. This is often the default option.
For ordinary files, umask 022 will give rw r- - r- - type protection, as executability is not allowed on ordinary files. Here is what happens after running umask 022:

*xstra > touch f1*

*xstra> ls -l f1*

*rw- r- - r- - 1 xstra staff 40 Jan 15 16:04 f1*

*xstra>*

The symbolic form of the umask command is much more pleasant to use than the octal form. Accepted by bash and Z-shell (but not by TCshell), it is fully consistent with the symbolic form of the command chmod, and should be preferred to the octal form, which is considered obsolete.
As an example:

*xstra> umask u=rwx, g=rx, o=rx*

*xstra> umask*

*0022*

*xstra> umask –S*

*u rwx,g rx,o rx*

**II.4.4 Access permissions of directories**

In the case of directories, the interpretation of rights is slightly different from that concerning the files. Information about a directory is obtained by the command ls -dl rept.
Example

*xstra > ls -dl bin*

*d rwx r - x r - x 3 xstra staff 1024 Jul 28 18:04 bin*

*xstra>*

The interpretation of protections for directories is as follows:
r allows reading the contents of the directory as in the case of files; allows you to see the list of files that are in the directory. x allows access to the directory (using the cd command). w

allows creation, deletion and renaming of a directory item. This permission is independent of access to files in the directory.

**Examples**

Consider a file f1 in the xstra1 directory.

1) The xstra1 directory belonging to the xstra1 user has the following protections:

*d rwx*

The f1 file of xstra1 has the following protections:

- *rwx*

Only the user xstra1 will be able to modify and delete his file f1.

2) The xstra1 directory belonging to the xstra1 user has the following protections:

d r - x

The f1 file of xstra1 has the following protections:

- rwx

Only the user xstra1 will be able to modify his file f1 but he will not be able to DELETE. Indeed the owner of the xstra1 directory (the xstra1 user) does not have w authorization (permission to create, delete, modify the name of an element of the directory).

### II.4.5 Permission to delete and the bit t

The possibility of deleting a file is therefore not fixed in the permissions of this file, but in the permissions of the directory which contains it. It is therefore not a fixed permission file by file. This is problematic, especially in the /tmp directory: any user must be able to create files in /tmp, which therefore has rwx permissions for other. Everyone can therefore read and write in /tmp, and therefore create files there. But anyone could also delete any file in /tmp, including files created by other users! Hence the usefulness of the bit t on a directory: if this bit is set, a user who can write to the directory can create files there, but can only delete a file if he owns it. The t bit appears instead of the x bit of other in the ls -l command:

*xstra> ls –ld /tmp*

*d rwx rwx rwt 11 root root 3072 May 11 15:09 /tmp*

### 2.4.6 Change owner and group

The chown command allows you to change the owner of a file. For security reasons, only the administrator can change the owner of a file or directory. This privileged user, called root, can access all files and directories without any restriction and can modify all their attributes (name, owner, owner group, permissions, dates). It still exists in all Unix/Linux systems. The chgrp command allows the change of group for the files or directories

mentioned, provided that the user is part of the new group and is the owner of these files or directories.

Example

*xstra > chgrp   info   f1*

*§ possible if I belong to the info group*

*xstra>chown    soline   f1*

*chown: f1: Operation not permitted*

*§ possible only for root*

The f1 file will be owned by soline and the info group.

An equivalent command (usable only by root) would be:

*root> chown   soline.info   f1*

### II.4.7 Multi-group membership

When logging on, a user belongs to his main home group: the one defined in the /etc/passwd file. However, this user can be a member of other groups, as defined in the /etc/group file. BSD and System V differ on the following points:

- In Unix versions of BSD strain:

Group permissions for a file (or directory) are applicable to any user who is a member of the group that owns the file. Any newly created file has the owner group of the group to which the directory in which it is created belongs. (A user can therefore create a file belonging to a group of which he is not a member).

- In System V strain Unix versions:

Group permissions for a file (or directory) are only applicable to a user if their effective group is the group that owns the file. A user can change the effective group at any time by issuing the newgrp new group command, provided that they are a member of the new group. Any newly created file has the effective group of its creator as its owner group.

Linux, like other versions of Unix, achieves a synthesis between these two behaviours :

● Group permissions for a file (or directory) are applicable to any user who is a member of the group that owns the file, as in BSD.

● Any newly created file has as owner group the effective group of the user who creates it if the directory which contains it is not SGID. This is System V behavior.

● Any newly created file has as owner group the group to which the directory in which it is created belongs if this directory is SGID. This is BSD behavior.

The choice between System V and BSD behavior can therefore be fixed directory by directory by setting the SGID bit on the directory. This is achieved by the commands:

*chmod g+s directory § set the SGID bit*

*chmod g - s directory § remove SGID bit*

If a directory is SGID, any subdirectories created later will inherit the SGID bit, and thus BSD behavior. This behavior is very suitable for group work:

**Example**

The xstra user is part of the staff group and also of the network group. His main group is staff. Let's see what happens in a non-SGID directory:

*xstra>umask 002*

*xstra > ls -ld project1*

*d rwx rwx r - x 2 xstra staff 4096 Jan 22 10:0 project1*

*§ the project1 directory is not SGID*

*xstra > cd project1*

*xstra > ls -l hello*

*rwx r 1 stone network 25 Feb 10 10:25 hello*

*xstra> cat hello § xstra is part of network group Hello*

*HELLO § so he can read the content of the*

*How are you? § of the file hello*

*xstra>touch prog2.c*

*xstra > ls –l prog2.c*

*- rwx rwx r - - 1 xstra staff 0 Feb 10 15:02 prog2.c*

*§ The prog2.c file belongs to the staff group*

*§ and not to the network group.*

Other members of the network group do not have permission to modify the prog2.c file: we are not in a group work logic for the network group. Now let's see what happens in an SGID directory:

*xstra> umask u rwx, g rwx, o rx*

*xstra > ls -ld project2*

*drwx rws r - x 3 xstra network 4096 Jan 22 10:0 project2*

*§ the project2 directory is SGID*

*xstra > cd project2*

*xstra>touch prog2.c*

*xstra > ls -l prog2.c*

*- rwx rwx r - - 1 xstra network 0 Feb 10 15:02 prog2.c*

*§ The prog2.c file belongs to the network group.*

In the latter case, all members of the network group will be able to modify any file created in this directory. Group work is very simplified. A much more detailed and very interesting discussion of this logic is available in the online help in HTML format, for example for the Red Hat distibution in:

Red Hat Linux Reference Guide

System Related Reference

System Administration

Users, Groups and User-Private Groups

# Chapter III

## THE BASIC MECHANISMS OF OPERATING SYSTEMS

### III.1 INTRODUCTION

Operating systems interact with both hardware and users (or their programs). Although each system has particularities, they have common characteristics at the level of the basic mechanisms. The purpose of this chapter is to present these mechanisms.

### III.2 THE LIFE CYCLE OF A PROGRAM

A program (written by the user in high-level language) cannot be executed directly on a target processor. It has to go through several steps for it to run. All of these steps are summarized in FIG III.1.

#### III.2.1 Translation (compilation or assembly)

Translation is the process of transforming the program written (by a user with a text editor or a development environment) in source language into a program in machine language (in binary code).

- If the source program is written in a high-level language (C, C++, Pascal, etc.), the translation requires a compiler (compilation). The compiler uses lexical and syntactical analysis techniques to generate the object code.

- If the source program is written in assembly language, the translation requires a so-called assembler program. Translation basically consists of determining the address (relative to the file) of each symbol (variable or label) and replacing each mnemonic instruction with its machine equivalent.

Whether compilation or assembly, the result obtained is said to be an object module which cannot be executed by the machine for at least two reasons:

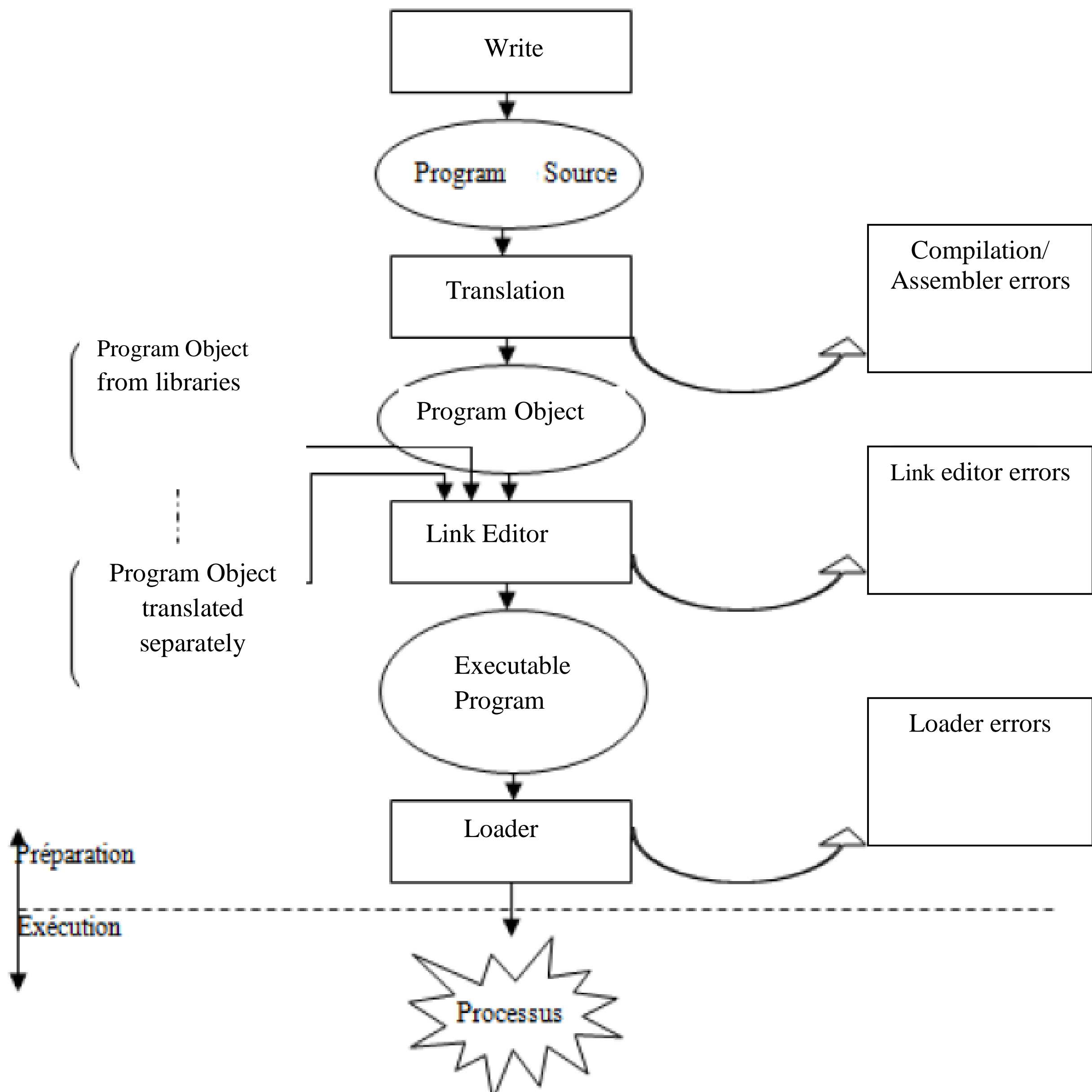



FIG III.1 Life cycle of a program.

- Instructions and data will be located at relative addresses which are not necessarily permitted addresses for execution. The source program can call on procedures that are in other source files or objects.

*Remarks:*

- Some languages (like PHP scripts, HTML, etc.) are not compiled but read and executed directly. They are said to be interpreted.

- Some compilers first translate the source code into assembly language so that you can make low-level changes and then reassemble it.

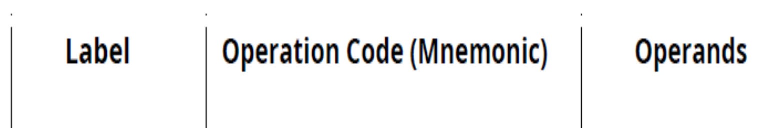
- The object program can be absolute (the addresses of the instructions and of the data have fixed addresses) as it can be reloadable (the instructions and the data can be relocated in any place in memory).

### III.2.1.1 The assembler

The first programs were written in binary. It was a difficult task and tedious subject to errors because the interpretation of bit sequences is not obvious. It was then necessary to consult a table describing all the operations in the form of successions of bits and their interpretation. To facilitate the work, the programs are written giving the names of the operations. We called them mnemonic codes because we could memorize them easily (ADD, SUB, DIV, MOV, etc.). Addresses of variables and instructions are given in symbolic form. The assembler is therefore only a symbolic variant of machine language: it therefore has the same set of instructions. Consequently, the assembler is specific to each type of machine.

It is a low-level language that represents machine language in a human-readable form. We note that the assembler signifies both the language using the mnemonics and the symbols and the translator who converts them into machine language. He is always used because it is the only one which makes it possible to exploit the resources of the machine to the maximum such as the optimization of the exploitation of the memory, diagnosis of errors which consists in examining millions of bits and building step by step the evolution of the program and its effects on registers and memory. For this, it is necessary to have a precise knowledge of the architecture of the machine and its operation at the level of the elementary operations. It therefore hides nothing from the programmer, which is an advantage in some cases but can become tedious in others.

An assembly instruction is divided into fields. Here is the typical structure of an assembly instruction.



| Label | Operation Code (Mnemonic) | Operands |
|---|---|---|

The different fields of an instruction are generally separated by one or more spaces. The number of operands of the third field varies from one machine to another from 0 to 3. After this field, it is desirable to add comments.

Example of instructions:

```
Data 1 :    DS      1
Adresse1 :  MOVE    A1, donnes1
            ADD     A1, A2
            JUMP    adresse2
```

The first instruction (a directive) makes it possible to define the variable given1 and to reserve 1 byte for it. The second instruction bearing the label address1, transfers the content of register A1 into the variable data1. The third instruction performs an addition between two registers and stores the result in the second register. The last instruction makes an unconditional jump to the address address2 which is somewhere in the program.

#### III.2.1.1.1 Operation Codes (Mnemonics)

It is obvious that it is easier to remember the terms ADD and SUB than to remember their corresponding binary codes (101101, 001001 for example), but the current machines allow several variants of the same operation: thus, one can have several types of addition depending on the nature of the operands. The operation code is therefore linked to the type of the operands which follow it. The instruction set can thus become complex. We are therefore often obliged to consult the table of mnemonic codes to write a program in assembler.

#### III.2.1.1.2 Operands and labels

Unlike machine language, assembly language allows you to give alphanumeric names to variables and labels (addresses of instructions), which greatly facilitates programming. For example, suppose you want to make a branch. In machine language, we must give in binary the exact memory position which contains the instruction to which you want to branch. In assembler, it suffices to precede the instruction (where you want to branch) with a symbolic label and to give this label as operand of the branching operation. In the same way, for the operands, we are no longer obliged to give the exact binary address. The operands have a name that allows them to be referenced.

Example of operands and labels:

| | | | |
|---|---|---|---|
| Tab | DS | 2 | Definition of a Tab variable of one word (Define Storage). |
| Dix | DC | 10 | Definition of a constant Ten which has the value 10 (Define Constant). |
| Boucle: | MOVE | Dix, A1 | Transfer of the value 10, stored at address Ten, into register A1. |
| | MOVE | A2, Tab | Transfer of the value of register A2 to the variable Tab. |
| | JUMP | Boucle | Unconditional jump to the address defined by the Loop label. |

### III.2.1.1.3 Literals

In machine language, any constant must be coded in binary. Assembly language allows you to define integer or real values in different bases (2, 8, 10 or 16) as well as character strings. It is always the assembler who takes care of their conversion. The indication of the base is carried out by placing a particular character at the beginning of each data. Thus, in the assembler of the MOTOROLA MC68000 microprocessor, binary data is preceded by a %, hexadecimal data by a $. If there is no particular character, it is a decimal data. The character strings are surrounded by the sign ' '.

Example of literals:

| | | | |
|---|---|---|---|
| ‘A’ | 65 | $41 | %0100 \| 0001 |
| ‘01’ | 12337 | $3031 | %0011 \| 0000 \| 0011 \| 0001 |

### III.2.1.1.4 Directives

The directives, Or pseudo-instructions, are non-executable instructions that have no equivalent machine code. These are directives given to the assembler which provide him with indications for translating the program. As with executable instructions, directives are used by referencing them with mnemonics. There are different kinds of directives. Take, for example, variables and constants. Symbol definition directives allow you to assign its value to a constant or to reserve the memory space of a variable.

Examples of directives:

TTL ‘Title of the program’

| | | | |
|---|---|---|---|
| Vecteur | DS | 50 | Definition of the Vector variable and reservation of 50 bytes. |
| Zéro | DC | 0 | Definition of the constant Zero which has the value 0. |
| | PLEN | 50 | 50 lines per page (PLEN- Page Length) |
| | END | | End of assembly |

### III.2.1.2 ASSEMBLY TECHNIQUES

To translate an assembly program into machine language, two techniques are used. One pass assembly and two pass assembly.

#### III.2.1.2.1 Two-pass assembly

In this type of assembly, the source program is read twice.

- ✓ **first pass (preparation):**

The first reading consists of performing the following tasks:

- Initializes the location counter to find the address (absolute or relative) of each instruction and data.
- Creates and populates a symbol table containing all symbols encountered and associated addresses.
- Reserves memory spaces for constants and variables (associating the appropriate number of bytes with each type).

The location counter allows the assembler to find the address of each instruction or data in the source program. If the assembler produces an absolute object module, the location counter is initialized to the start address of the instruction or data memory space. If the object module can be relocated, the slot counter is initialized to 0. The symbol table as its name suggests contains all the symbols (identifiers and labels) of the program. For each symbol, the assembler tries to associate its address with it. If the address is known (symbol definition), the assembler saves it with the symbol. If the address is unknown (forward reference), the assembler stores only the symbol. When the assembler arrives at the place where this symbol is defined, it will update the table with the value of this symbol. The operation code table is a table containing the operation code of each instruction and its length in bytes. This table makes it possible to know the size of the space occupied by each instruction and thus to know the address of the symbols encountered.

- ✓ **Second pass (proper translation)**

In this pass, the source code is read again and the object code is generated progressively by the assembler by replacing each mnemonic by its equivalent in machine language in the operand table and each label by its machine language equivalent in the symbol table built in the first pass.

Example: Consider the following assembly program where it is assumed that 001A is the address of the instruction labeled by Loop and which is located at the level of the tenth line of the program.

| Program in assembler | | | | Adress (on 2 bytes) | Line number |
|---|---|---|---|---|---|
| Tab | DS | 2 | | 0000 | 01 |
| Dix | DC | 10 | | 0002 | 02 |
| JUMP | Boucle | | | 0004 | 03 |
| | ⋮ | | | | ⋮ |
| | | | | | ⋮ |
| Loop | MOVE | Dix | A1 | 001A | 10 |
| | MOVE | A | Tab | 001F | 11 |
| A1 | DS | 2 | | 0024 | 12 |

**First pass:**

Table III.1 table of symbols.

| **Symbole** | **Value** | **Type** |
|---|---|---|
| Tab | 0000 | Word |
| Dix | 0002 | Word |
| Loop | 001A | Label |
| A1 | 0024 | Word |

Table III.2 table of operation codes.

| **Mnemonic** | **Code opération(Hex)** |
|---|---|
| JUMP | 05 |
| MOVE | 07 |

**Second pass:**

| Source line | | | |
|---|---|---|---|
| Tab DS 2 | | | |
| Dix DC 10 | | 00 | 0A |
| JUMP Boucle | 05 | 00 | 1A |

⋮

⋮

| | | | | | | |
|---|---|---|---|---|---|---|
| Loop | MOVE Dix A1 | 07 | 00 | 02 | 00 | 24 |
| | MOVE A1 Tab | 07 | 00 | 24 | 00 | 00 |
| A1 | DS 2 | | | | | |

**III.2.1.2.2 One pass assembly**

In this technique, only one read of the source program is performed. This pass is unique and is identical to the first pass of the two-pass assembler. Object code is generated in memory as instructions are read. When the assembler encounters an instruction that references an unresolved tag, it keeps a pointer to that instruction until that tag is encountered. Once the label is found, the assembler returns to the instructions that referenced it and updates them. At the end of the read, the assembler creates an object file and puts the assembled memory contents in this file. The assembler uses three tables in this case: an assembled instruction table which contains directly translatable instructions, the unassembled instruction table which contains instructions with a forward reference and a symbol table. The latter two are updated as forward reference addresses are resolved. A flag bit is set to 1 if the reference is unknown and is set to 0 otherwise.

This technique is faster than the previous one but consumes a lot of memory because all the object code must be built in memory before putting it in an object file. We take the example of the previous code and process it with a pass as follows:

**III.2.1.3 Format of an object program (Unix type)**

An assembler produces an object file. An object (Unix-like) file contains six sections:

-A header indicating the size and position of the other sections.

-A section of text containing machine language instructions.

-A data section containing the binary representation of the data.

-A section containing relocation information. It mainly identifies instructions that use absolute addresses (memory access, for example). The memory addresses to which these instructions access are not necessarily the real addresses at runtime. They will need to be changed during linking and loading.

Table III.3 Table of symbols (one-pass assembler).

| **Symbol** | **Value** | **Type** | **Bit indicator** | **Queue Waiting** |
|---|---|---|---|---|
| Tab | 0000 | Word | 0 | |
| Dix | 0002 | Word | 0 | |
| Loop | 001A | Label | ~~1~~ 0 | 0004 |
| A1 | 0024 | Word | ~~1~~ 0 | 001F, 001A |

Table III.4 table of assembled instructions.

| **Ligne** | **Instruction** | **Adresse** |
|---|---|---|
| 1 | ---- ---- | 0000 |
| 2 | 000A | 0002 |
| 12 | ---- ---- | 0024 |

Table III.5 Table of unassembled instructions.

| **Ligne** | **Instruction** | **Adresse** |
|---|---|---|
| 3 | 05 00 1A | 0004 |
| 10 | 07 0002 0024 | 001A |
| 11 | 07 0024 0000 | 001F |

-A symbol table containing global symbols (which are defined in the module but which may be referenced by other object modules) and external references (those which are referenced by the module but defined in other modules).

-Debugging information (optional) useful for debugging and correcting applications. They contain precise information on the way in which the program was compiled and the correspondence between each instruction of the source program with that of the object program so that a user can know which is the source instruction which produces a given result.

### III.2.2 Link editing

Applications are, in general, developed in a modular way. The program is broken down into parts (description of data, procedures, access to files, etc.) which can be found in their own source files. It can call on system procedures, program libraries, pre-existing modules that have been written in different languages: assembler, C, Pascal, Ada, etc. Each of these modules can be compiled separately so that a change to one module does not require recompiling the entire program. The production of an executable program requires the

combination of these different modules. This operation is called link editing (figure II.2). It consists of performing the following tasks:

-Search the program libraries for the routines used and insert them into the program.

-Determine the memory area that each module occupies and adjust the absolute references

-Resolve references between modules.

-Produce the executable program grouping all the modules.

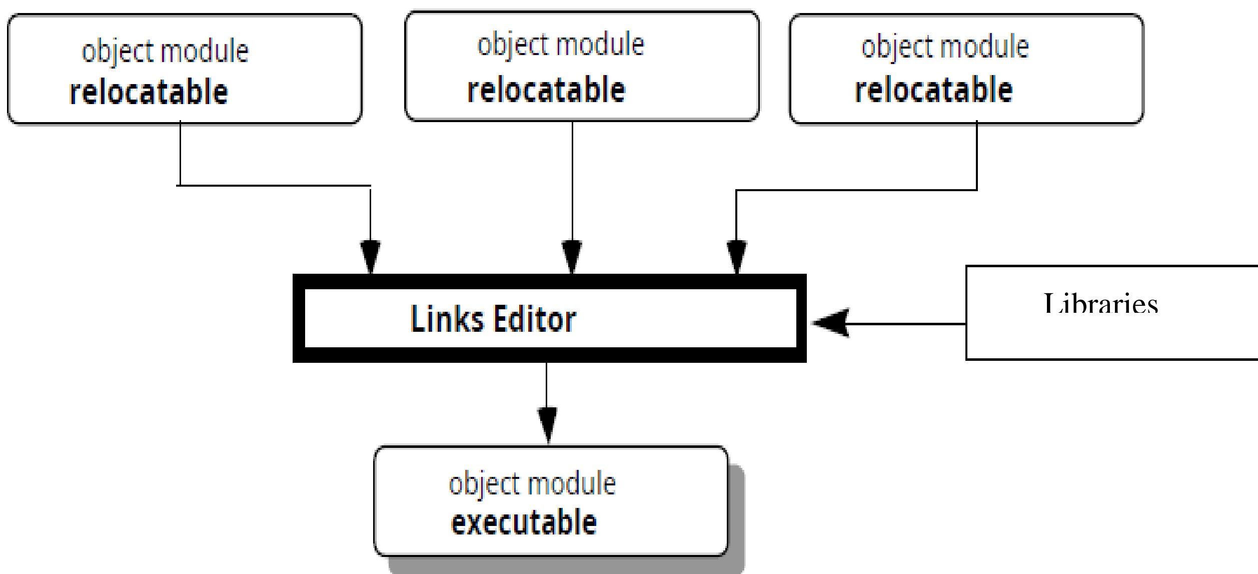



FIG III.2 Link editor.

Linking is done in two passes:

- ✓ **First pass**

This pass consists of:

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

- ✓ **Second pass**

This pass consists of:

- Read the modules one by one
- Transform the addresses of absolute references according to the installation address of the module.

- Set address of unresolved references (using symbol table)
- Generate the executable file.

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

### III.2.3 Loading

The object program, obtained after the linking, must still be loaded into main memory to be executed. The charger takes care of this task. It is usually coupled with the linker. In most modern multiprogrammed systems, it is decided at the last moment at which address to load the program. In the early days, computers had only one program in central memory and it was therefore sufficient to fix the addresses of a program in advance and then to load it at the prefixed location. This type of charger is called absolute loader. Now, the loaders also take care of relocating the program in central memory, they are loaders relocatable. The principle is simple: as the program begins at address 0 and all the instructions are numbered with respect to this origin, it suffices to add to all the addresses (the instructions and the data) the displacement with respect to l'address 0. For example, if we want to load the program from address 2000, the first instruction is loaded at address 2000 and we add 2000 to all the addresses found in the program. The loader must therefore add the value of the displacement to the load address of each instruction, which is easy, and it must modify each reference to an instruction or to a data, which is less easy. Suppose the following statement, represented in octal, is part of a program that the loader places in main memory from address 2000.

| *Instruction address* | *Instruction content* |
|---|---|
| *00043* | *171600000012* |

The loader can easily add the value 2000 to 00043, but the problem is whether the loader should modify the contents of the instruction. The content of the operand field lends itself to several possible interpretations. It can be an integer, a real, a character string, a reference to an operand stored in a data area or a branch address. It is necessary to modify the content of the instruction only in the last two cases by adding the value 2000 to the address of the operand. However, the loader cannot know the case handled. It is up to the assembler to indicate in the object program that it generates, if the content of the instruction must be relocated. Here is an example of a relocatable program instruction:

| *Instruction address* | *Instruction content* | *Relocation indicator* |
| --- | --- | --- |
| *00043* | *171600000012* | *1* |

Another way to perform relocation is to use a base registry. The relocation technique then consists of:

- assemble the program with respect to address 0 and do the link editing.
- choose a base register among those available to the computer.
- store the base address (absolute origin of the program).
- load the program into memory from the base address, without modifying the program addresses.

During execution, at each reference, the machine performs the following calculation:

*Effective address= base address + displacement*

Such a program can be relocated without modifying the addresses inside the program. To do this, simply modify the contents of the base register. Obviously, it is necessary to avoid any inopportune modification of this register because it would be fatal to the program. We realize that the loader is an essential program for loading programs into memory and executing them.

**III.2.4 Processes**

Once the program is loaded into memory, it becomes an active entity that uses and acts on computer resources. It becomes a living, working and sleeping process. It can spawn other child processes and dies after its task is completed. The activity of a computer at a given time is characterized by a set of user and system processes which run concurrently on one or more central units. In general, the central unit is multiplexed between the different processes so as to be allocated to each of them for a well determined period.

In addition to code and data (TEXT section and DATA section), a process is characterized by other essential information. The current activity is represented by the current value of the ordinal counter (or PC). A process is also characterized by its stack which contains temporary data (such as procedure parameters, local variables, etc.). A process can also contain a memory area that it uses dynamically (that it dynamically allocates according to its needs) called HEAP. Figure III.3 gives the image of a process in memory. Several processes can be associated with the same program. In this case they are considered as separate entities. For example, multiple users are running different copies of a Mail messaging client. A single user can launch multiple copies of a web browser. Each of these copies is a

separate process that has the same TEXT section but has different DATA, HEAP and STACK sections.

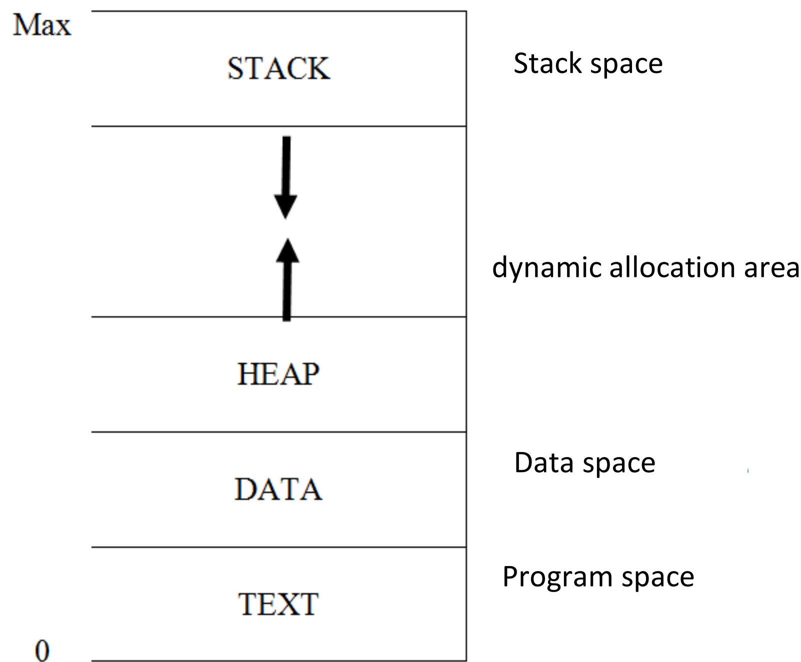



FIG III.3 Image of a process in memory.

## III.3 SEPARATE COMPILATION

### III.3.1 Global variables

A global variable is a variable that is defined outside of any function. A global variable declared at the beginning of a source file can be used in all functions of the file. The variable exists only once, and changing the global variable in one function changes the value of that variable in other functions.

```
#include <stdio.h>
inte;      /* declaration outside of any function */
\void  ModifyGlobalData(\void)    /* no parameter */
 {
x = x+1;
}
```

```
void   DisplayGlobalGive (void) /* no parameter */
{
printf ("%d\n", x);
}
int main (void)
{
x=1;
ModifyGlobalData ( );
DisplayGlobalGlobal ( );      /* displays 2 */
return 0;
}
```

In the case of a project with multi-file programming, you can use in a source file a global variable defined in another source file by declaring this variable with the key word external (which means that the global variable is defined elsewhere).

*external int x;      /* declaration of an external variable*

**III.3.2 Putting code in multiple files**

Suppose a type Item that aggregates data for a product in a store. The function main, in the file main.c, calls the functions EnterProduct and DisplaysProduct, which are defined in the file routines.c. Both files .c include the file producttype.h.

```
/************************************************** **
|******* HEADER FILE producttype.h ***************|
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * /
/* 1) Protection against multiple inclusions */
# ifndef   MY_HEADER_FILE  /* Avoid multiple definition */
# define  MY_HEADER_FILE


/* 2) Definition of exported global constants and variables */
#define STRING_LENGTH 100
 extern int error;
```

```
/* 3) Definition of structures and types */
typedef struct {
                int code;      /* item code */
                char denomination[STRING_LENGTH]; /* Product Name */
                 float price;   /* unit price of the product */
                  int stock; /*Available stock */
                 }ItemType;
/* 4) Prototypes of functions that are defined in one file but used in another */
void EnterProduct(ItemType *addr_prod);
void DisplayProduct(ItemType prod);
# endif /* end of protection */
```

Protection #ifndef prevents the code in the header file from being compiled multiple times, causing multiple definition errors, if multiple source files include the header.

```
/**************************************************\
|********** SOURCE FILE routines.c ****************|
\**************************************************/
#include <stdio.h>

# include "producttype.h"      /* watch out for quotes */
int max_stock=150;
/* The error global variable is defined here */
   int error;
/* The following function is static */
/* LitString is not accessible in main.c */

static GetString(const char *string)
 {
fgets(string, STRING_LENGTH, stdin);
}
```

```
/************************************************* **\
************* SOURCE FILE main.c *****************
\************************************************* **/
# include "producttype.h" /* watch out for quotes */

int main(void)

{
TypeProduct prod;
error=0; /* variable defined in another file */ EnterProduct(&prod); /*
function defined in another file */ if (error==0)
DisplayProduct(prod);
}
```

```
void EnterProduct(ItemType *adr_prod)
{
printf("Product code: ");
scanf("%d", &adr_prod->code);
 printf("Denomination: ");
GetString(adr_prod->denomination);
printf("Price: ");
scanf("%f", &adr_prod->price);
printf("Stock available: ");
scanf("%d", &adr_prod->stock);
if(addr_prod->stock > max_stock)
     {
     fprintf(stderr, "Error, stock too big\n");
    error=1;
      }
}
```

```
void DisplayProduct(ItemType prod) {
printf ("Code: %d\n", prod.code);
 printf ("Denomination: %s\n", prod.denomination);
printf ("Price: %f\n", prod.price);
printf ("Stock available: %d\n", prod.stock);
}
```

**III.3.3 Compile a multi-file project**

**III.3.3.1 Without makefile**

To compile the previous example without makefile below linux, i.e. to create an executable file, you must first create a file object for each source file.

*$ gcc -c routines.c $*

*gcc -c main.c*

This should generate two object files routines.o and main.o. We then create the executable (for example called product.exe) by performing the link editing (link) with the following instruction:

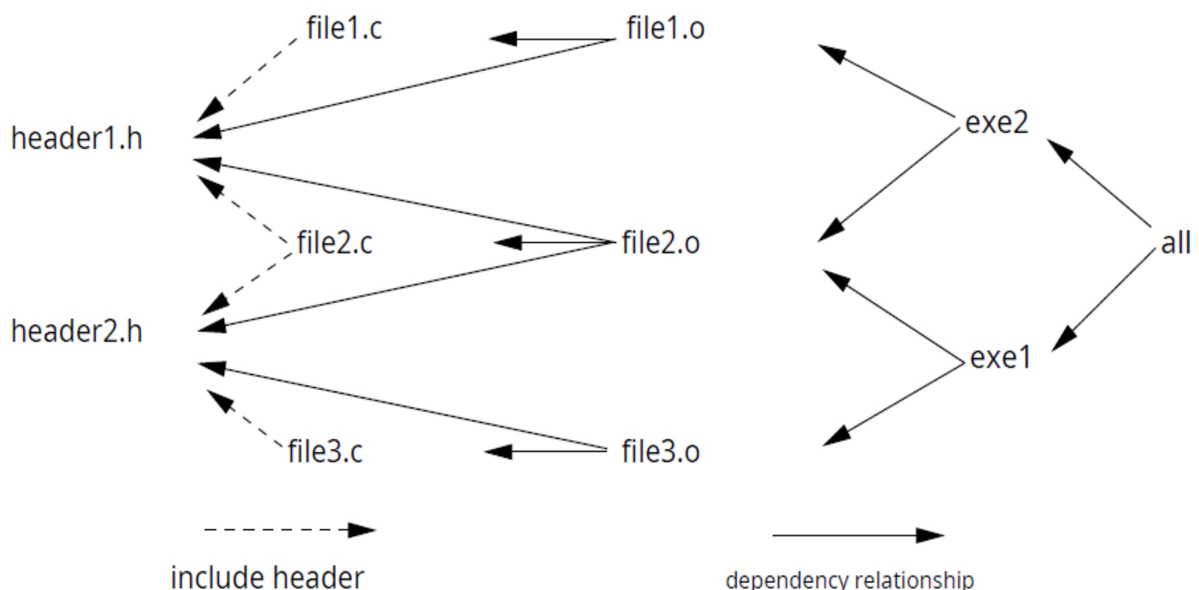



FIG III.4: Example of separate compilation of two executable files.

*$ gcc routines.o main.o  -o  product.exe*

It is during link editing that the links between functions calling from one file to another, and global variables external are made. We can schematize this process of separate compilation of source files by FIG III.4, in which two executable files exe1 and mttex2 respectively depend on certain source files, which themselves include headers. Dependency rules state that when a certain file is modified, certain other files must be recompiled.

**Example.** In FIG III.4, if the file header1.h is modified, it is necessary to rebuild file1.o and file2.o. This entails that exe1.o and exe2.o also need to be rebuilt. In case of modification of header1.h, so you have to run the following commands:

$ gcc -c file1.c
$ gcc -cfile2.c
$ gcc file1.o file2.o  -o  exe1
$ gcc file2.o file3.o  -o  exe2

**III.3.3.2 With makefile**

A makefile is a way to automate the compilation of a multi-file project. Thanks to makefile, updating the object files and the executable file following a modification of a source is done by simply using the command:
*$make*
To do this, you must specify to the system the dependencies between the different project files, by creating a file makefile.

**Example 1.** For example files main.c, routines.c and producttype.h above, we create a text file named makefile containing the following code:

```
product.exe : routines.o main.o
	gcc routines.o main.o -o product.exe
routines.o : routines.c producttype.h
	gcc -c routines.c
main.o: main.c producttype.h
	gcc -c main.c
```

This file consists of three parts. Each part expresses a dependency rule and a reconstruction rule. The reconstruction rules (lines2,4And6)must begin with a tab. The dependency rules are as follows:

• The executable file product.exe depends on all object files. If one of the object files is modified, the reconstruction rule must be used to edit the links.

• The object file routine.o depends on the file routine.c and file itemtype.h. If one of these two files is modified, use the rebuild rule to rebuild routine.o

• Likewise, main.o depends on main.c and itemtype.h.

**Example 2.** For the files in Figure II.4, a simple makefile would look like this:

```
CC=gcc # variable giving compiler
CFLAGS= -Wall  -pedantic  -g  # compilation options
all: exe1 exe2 # rule to rebuild all
file1.o : file1.c header1.h
	$(CC) $(CFLAGS) -c  file1.c
file2.o : file2.c  header1.h  header2.h
	$(CC) $(CFLAGS) -c file2.c
 file3.o : file3.c   header2.h
	$(CC) $(CFLAGS) -c file3.c
 exe1: file1.o file2.o
	$(CC) $(CFLAGS) file1.o   file2.o  -o  exe1
exe2: file2.o  file3.o
	$(CC) $(CFLAGS) file3.o file3.o  -o exe2
clean:
	rm  *.o  exe1  exe2
```

Use :

```
$ make exe1          # rebuild target exe1
$ make all           # rebuild all
$ make clean         # clean up
$ make               # rebuild my first target (in this case all)
```

# Chapter IV

## SHELL PROGRAMMING

### IV.1 INTRODUCTION

The shell is more than a command interpreter: it is also a powerful programming language. This is not unique to Linux; any operating system offers the possibility of saving sequences of commands in files which can be invoked later. But no operating system offers so much flexibility and power than the Linux shell in this type of programming. The other side of this coin is that the syntax of this language is quite strict and off-putting. Moreover, the existence of several shells leads to several different languages. In Linux, a file containing commands is called a script and we will only use this term in what follows. We will also use the term shell to designate both the command interpreter and the corresponding language (just as "assembler" designates both assembly language and compiler of this language).

Like any conventional programming language, the shell has statements and variables. Variable names are character strings; their contents are also character strings. The assignment (Bourne-shell, POSIX-shell and Bash) of a value to a variable is made by name; the reference to this variable is made by its name preceded by the character $, as in:

*my variable hello § assignment*

*echo $myvariable § reference*

The instruction set itself includes:

- all Linux commands,
- the invocation of executable programs (or scripts) with parameter passing,
- variable assignment instructions,
- conditional and iterative instructions,
- input-output instructions.

of course, the mechanisms of pipes and redirections can be used in a script.

**Remarks**

1) The shell is an interpreted language; consequently any change in the system will be taken into account by a script during its next use (there is no need to “recompile” the scripts).

2) It is entirely possible to write and invoke scripts in a certain shell while using another shell interactively. In particular, it is very common (but not mandatory) to use a TC-shell as “login shell” and the Bourne-shell or another shell for writing scripts. The simplest scripts (lists of commands) will be identical whatever the shell, but as soon as instructions for tests or iterations are required, the syntaxes of Bourne-shell, Bash and C-shell differ.

3) If a script begins with the line

*#!/bin/xxx* (shell path xxx which must interpret this script).

It is interpreted by the /bin/xxx shell. Our goal here is not an exhaustive study of shell programming, but an introduction to this technique illustrating the main concepts and instructions.

## IV.2 BASIC SHELL PROGRAMMING

In what follows, we will assume that the user's environment is Bash, and that he writes his scripts in the language of the Bash command interpreter. The programming basics presented in this paragraph can be considered as common to all command interpreters from the Bourne-shell family (Bourne-shell, POSIX-shell, Bash).

Note that it must always start a shell script with the line *#!/bin/bash*.

### IV.2.1 The first script

Creation with the vi editor of the listf file containing the line ls aCF. An ordinary file does not have the x right (it is not executable) when it is created, so :

*xstra> chmod a+x listf § add the right x*

*§ for everyone.*

It can therefore be executed as a command:

*xstra > listf*

*./ .kshrc .security/ bin/ florent/ xavier/*

*../ .profile .sh_history dpt/ jerome/*

*xstra>*

*Running the script in debug mode:*

*xstra > sh x listf § trace mode*

Or

*xstra > sh v listf § verbose mode*

make it possible to ask the shell which interprets the script to trace the progress of the script or to comment it (or both). It is also possible to include in the script the lines:

*set x (for trace mode)*

*set v (for verbose mode)*

Trace mode copies each line to standard output as it is interpreted. The verbose mode copies on the standard output each line before interpretation.

**IV.2.2 Parameter passing**

The listf script only applies to the current directory. It can be made more general by passing it the name of a directory as an argument during the invocation. To do this, the variables 1, 2, …, 9 are used to respectively designate the first, the second, …, the ninth parameter associated with the invocation of the script.

**a) First script with parameter passing**

With vi, modify the listf file as follows:

echo ”contents of directory $1”

ls aCF $1

Execution gives:

xstra > listf /tmp

contents of /tmp directory

./ ../ df_file

xstra>

**b) Generalization**

The number of parameters passed as arguments to a script is not limited to 9; however, only the nine variables 1, …, 9 allow these parameters to be designated in the script. The shift command works around this problem. After shift, the ith parameter is denoted by $i 1.

**Example 1**

The echopara script contains:

echo $1 $2 $3

P1 $1

shift

echo $1 $2 $3

echo $P1

The execution gives:

xstra> echopara one two three

one two three

two three

one

xstra>

This example shows the behavior of shift, assigning a value to the variable P1 (P1=$1) and the reference to this variable (echo $P1).

**Example 2**

Parameter shift echopara1 script contains:

*echo $1 $2 $3 $4 $5 $6 $7 $8 $9*

*shift*

*echo $1 $2 $3 $4 $5 $6 $7 $8 $9*

The execution gives:

*xstra> echopara1 1 2 3 4 5 6 7 8 9 10*

*1 2 3 4 5 6 7 8 9*

*2 3 4 5 6 7 8 9 10*

*xstra>*

**IV.2.3 Special variables**

In addition to the variables 1, 2, …, 9, the shell predefine variables facilitating programming.

0 contains the name under which the script is invoked,

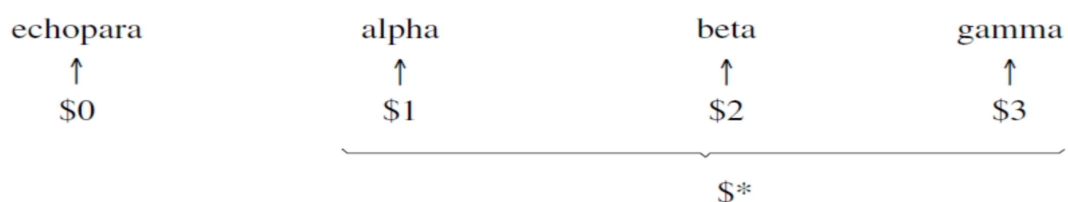
# contains the number of parameters passed as argument,

* contains the list of parameters passed as arguments,

? contains the return code of the last executed command,

$ contains the process number (PID) of the shell (decimal).

**Exemple 1**



**Example 2**

The echopara2 script contains:

*echo $0 was called with $# parameters*

*echo which are: $**

The execution gives:

*xstra> echopara2 a b c d*

*./echopara2 was called with 4 parameters*

*which are: a b c d*

*xstra>*

Be careful, the # variable is not a numeric variable (which does not exist) but a character string type variable (the same for $).

0 lets you know under what name this script was invoked. In the case where the script bears several names (by links), this makes it possible to take such and such a decision according to the name under which the script was invoked.

$ process number (PID) of the shell, is unique in the system: it is frequently used to generate temporary file names.

**Example**

*tempfil /tmp/infile_user_$$*

*§ use of temporary file*

*rm $tempfil*

#### IV.2.4 Special characters

These characters generate file names and are used in the passing parameters. According to a general mechanism, the shell can generate a list of filenames corresponding to a certain pattern (thanks to the characters * and ?). This generation takes place before invoking the command (and therefore the script) concerned. For example, if the current directory contains only the files file1 and file2, during the ls fi* command, the shell generates the list file1 file2 and passes it as an argument to the ls command. The command actually launched by the shell is therefore: ls file1 file2.

### IV.2.5 Read and write instructions

These instructions make it possible to create interactive command files by establishing a dialogue in the form of questions/answers. The question is asked by the echo command and the answer is obtained by the read command from the keyboard.

*read variable1 variable2… variable n*

read reads a line of text from the keyboard, splits the line into words, and assigns to the variables variable1 to variable n these different words. If there are more words as many variables, the last variable will be assigned the rest of the line.

Example

The poster script contains:

*echo -n "File name to display: "*

*read file*

*more $file*

### IV.2.6 Control structures

The shell has control structures such as are found in general purpose programming languages:

- conditional statements (if.. then.. else, test, case),
- bounded iterations,
- unbounded iterations.

#### a) Conditional statements

For the programming of conditional actions, we have three tools:

- the if statement,
- the test command that completes it,
- the case statement.

➤ The if statement

It presents three variants which correspond to the selective structures with one, two
or n alternatives.

α) The selection to an alternative: if… then… fi

*if command*

*then commands*

*fi*

Commands are executed if the command condition command returns a null return code *($?=0).*

**Example 1**

The echoif1 script contains:

*if grep i xstra1 /etc/passwd*

*then echo User xstra1 is known to the system*

*fi*

β) Selection with two alternatives: if… then… else… fi

*if command*

*then commands1*

*else commands2*

*fi*

The commands commands1 are executed if the command_condition *command* returns a null return code, otherwise it is the *commands2* which are executed.

**Example 2**

The echoif2 script contains:

*if grep qi "xstra1" /etc/passwd*

*then echo User xstra1 is known to the system*

*else echo User xstra1 is unknown to the system*

*fi*

γ) Selection with n alternatives: if… then…. elif…then…fi

*if command1*

*then commands1*

*elif command2*

*then commands2*

*elif command3*

*then commands3*

…

*else*

*commands0*

*fi*

**Exemple 3**

```
#!/bin/bash
# this script is called scriptf
# this script shows how to use
# replaceable parameters
# when invoking a script.
# examples: xstra>scriptf
# xstra> scriptf 1
# xstra> scriptf 1 2 3 4
# xstra> scriptf "1 2 3 4"
# xstra> scriptf  ok one two three
echo " " #echo of an empty line newline
echo " "
echo "Example of passing parameters to a script."
echo " "
# the next line shows how to use
# the result of a command:
echo "Current directory is: `pwd`"
# the next line shows the difference between
# single and double quotes:
echo 'Current directory is: `pwd`'
echo "This script has the name: $0"
if [ $# eq 0 ] # how to test the number of parameters?
  then
          echo "It was called without a parameter."
else
           if [ $# eq 1 ]
             then
                     echo "It was called with parameter $1"
          else
                      echo "It was called with $# parameters, which are: $*"
           fi
```

```
        if [ $1 'ok' ] # how to test the contents of a parameter
          then
             echo "Well done: the first parameter is $1"
        else
            echo " The first parameter is not ok but $1"
          fi
fi
echo "Goodbye $LOGNAME, script $0 is finished."
```

➢ The test command

It is the essential complement to the if statement. It allows very merely :

- to recognize the characteristics of files and directories,
- to compare character strings,
- compare numbers algebraically.

This command exists in two different syntaxes:

*test expression*

Or

*[ expression ]*

The test command responds to the query formulated in expression, by a zero return code in the event of a positive response and non-zero otherwise. The second form is more frequently encountered and gives rise to type programs:

*if [ expression ]*
*then commands*
*fi*

In [ expression ], do not forget the space character between [ and expression and between expression and ]. If then is on the same line, it must be separated from ] by a space and a character; the most used expressions are:

*-d name* — *true if directory name exists,*

*-f name* — *true if file name exists,*

*-s name* — *true if file name exists and is not empty,*

*-r name* — *true if file name exists and is readable,*

*-w name* — *true if file name exists and is writable,*

*-x name* — *true if file name exists and is executable,*

*-z string* *true if string string is empty,*

*-n string* *true if string string is non-empty,*

*c1 = c2* *true if the character strings c1 and c2 are identical,*

*c1 != c2* *true if the character strings c1 and c2 are different,*

*n1 -eq n2* *true if integers n1 and n2 are equal.*

(Other relational operators are ne, lt, le, -gt, and -ge.).

Notice that expressions can be negated by the negation logical operator! And combined by the operators or logical o and and logical a.

➤ The case expression

The case expression is a very powerful and very convenient instruction to perform a multiple choice in a file of commands.

*case chain in*

*motif1) commands 1 ;;*

*motif2) commands 2 ;;*

....

....

*motifn) commandes n ;;*

*esac*

The shell searches, among the different character strings motif1, motif2,…, motifn proposed, the first one that matches string and it executes the corresponding commands. A double semicolon (;;) terminates each choice. The string in a case can take various forms:

- a number,
- a letter or a word,
- special shell characters,
- a combination of these elements.

The string can be read, passed as a parameter or be the result of a command executed with the backquote ` ` or $( ) operator. In the different strings motif1 to n, special characters (*, ?,…) can be used. Moreover, to group several patterns in the same alternative, we use the character | (obtained on a standard keyboard by the combination <Alt Gr><6>).

**b) Iterations**

The presence of iterative instructions in the shell makes it a programming language complete and powerful. The shell has three iterative structures: for, while and until.

➢ Bounded Iteration: The for Loop

Three forms of syntax are possible:

1) Form 1

*for variable in string1 string2… stringe n*

*do*

*commands*

*done*

2) Form 2

*for variable*

*do*

*commands*

*done*

3) Form 3

*for variable in **

*do*

*commands*

*done*

For each of the three forms, the commands placed between do and done are executed for each value taken by the shell variable variable. What change is where variable gets its values. For form 1, the values of variable are the strings from *chain1* to *chainen*. For form 2, variable takes its values from the list of script parameters. For form 3, the list of files in the directory constitutes the values taken by variable.

**Example 1**

The echofor1 script contains:

*for i in one two three*

*do*

*echo $i*

*done*

Execution gives:

*xstra> echofor1*

*A*

*two*

*three*

*xstra>*

**Example 2**

The echofor2 script contains:

*for i*

*do*

*echo $i*

*done*

Execution gives:

*xstra> echofor2 the Linux system*

*THE*

*system*

*Linux*

*xstra>*

**Example 3**

The echofor3 script contains:

*for i in **

*do*

*echo $i*

*done*

Execution gives:

*xstra>echofor3*

*file1*

*file2*

*file3*

*xstra>*

**Example 4**

The lsd script contains:

echo ”list of directories under `pwd`”

*echo ” ”*

*for i in **

*do*

*if [ d$i ]*

*then*

*echo $i ”:directory”*

*fi*

*done*

*echo ” ”*

Execution gives:

*xstra > lsd*

*list of directories under /home/xstra/test*

*vein: repertoire*

*xstra>*

➤ Unbounded iterations: while and until

*while alphacommand*

*do betacommands*

*done*

*until alphacommand*

*do betacommands*

*done*

beta commands are executed while or until what (until) the command alpha command returns a null code (the condition is true).

**Example 1**

The following script lists the parameters passed to it as arguments until that it meets the fine parameter.

*#!/bin/bash*

*# this script is named while_*

*# this script shows how it works*

*# of the while construct*

*# example: xstra>while_ 1 2 3 end 4 5 6*

```
#
while [ $1 ! end ];do
echo $1
shift
done
```

*Execution gives:*

```
xstra> while_ 1 2 3 4 end 5 6 7
1
2
3
4
xstra>
```

**Example 2**

```
#!/bin/bash
# this script is called until_
# this script shows usage
# of the until construct
# example: xstra>until_ 1 2 3 end 4 5 6
#
until [ $1 end ] ;do
echo $1
shift
done
```

*Execution gives:*

```
xstra> until_ 1 2 3 end 4 5 6
1
2
3
xstra>
```

## IV.3 INTEGER ARITHMETIC ON VARIABLES

The Bourne-shell doesn't allow you to define numeric variables, yet the -eq operators don't lt le gt ge exist. It's not an inconsistency.

All variables defined in Bourne-shell are of character string type, but if the content of this string represents an integer, the preceding operators may apply.

Arithmetic operations on variables are very easy in bash thanks to two mechanisms not available in the traditional Bourne-shell: Arithmetic evaluation:

$((variable1 operator variable2))

The arithmetic test:

if (( variable1 operator variable2 ))

## IV.4 ADVANCED BASH PROGRAMMING

The exhaustive study of Bash as a command language would require an entire book. We will therefore simply present the most important points in relation to the basic concepts presented in the previous paragraph. Moreover there are differences between versions of Bash. In this paragraph we will consider being in version 2 (bash version). Some points developed below do not work in version 1.

### IV.4.1 Bash predefined variables (not defined in Bourne-shell)

The variables dynamically updated by Bash are:

*PPID* *number of the parent process (Parent Process IDentifier),*

*PWD* *working directory,*

*RANDOM* *a random number,*

*SECONDS* *time elapsed since the launch of this Korn-shell,*

*!* *number of the last process launched in the background,*

*_ (underscore)* *last word of the last executed command.*

### IV.4.2 Variable definition: the declare command

In addition to the *variable=value* form, the Bash has a general variable definition command:

*declare [+/ option] [ p] [variable[ value]]*

*option* assign attribute to variable

*+option* remove attribute from variable

The –p option, exclusive of the other options, allows you to display the variables and their value. The –f or –F option allows functions to be displayed with or without their definition. The other options are:

*-a variable is of type array,*

*-i variable is of integer numeric type,*

*-r read-only variable,*

*-x exported variable (variable placed in the environment)*

The typeset command is equivalent to declare.
In the case of creating a table, the simplest syntax is:

*declare a tab=(element1 element2 element3)*

The array *tab* will have 3 elements which will be: tab[0]=element1, tab[1]=element2, tab[2]=element3.
An element of the array tab is created using the simple syntax tab[element number]=value.
The elements of a variable array are numbered from 0 to N-1.

**Example**

*xstra> declare a trees=(fir oak acacia elm)*
*§ definition of the tree array*
*xstra > declare p trees*
*declare a trees '([0] "fir" [1] "oak" [2] "acacia" [3] "elm")'*
*xstra> echo $trees § designates the first fir  tree element*
*fir*
*xstra> echo ${trees[2]} § denotes element number 2 (0, 1, ..)*
*acacia*
*xstra> echo ${trees[*]} § denotes the sequence of all elements*
*fir oak acacia elm*
*xstra> echo ${#trees[1]} § denotes the size of element 1*
*5*
*xstra> echo ${#trees[*]} § denotes the number of elements*
*4*
*xstra>*

The +a option does not exist. To destroy a table, use the unset tab command.

### IV.4.3 The test command

The command test expression or [ expression ] is the same as in Bourne-shell, but the game possible for expression is richer there.

*a) The operator == can replace = for comparison*
c1 == c2 true if the character strings c1 and c2 are identical, (notation preferable to simple =)

*b) Test on the type of a file*

expression - The return code is true if:

*-a file file exists*

*-b file file is block special type*

*-c file file is special type character*

*-d file file is a directory*

*-f file file is a file*

*-L file file is a symbolic link*

*-p file file is a named pipe (pipe)*

*-s file file exists and is of nonzero size*

*c) Testing relationships between files*

expression - The return code is true if:

file1 -ef file2 file2 is a link on file1 (ln)
file1 -nt file2 file1 is newer than file2 (newer than)
file1 -ot file2 file1 is older than file2 (older than)

*d) Testing file access rights*

expression - The return code is true if:
-r file the current process has r permission on file
-w file current process has w permission on file
-x file current process has x permission on file
-g file file has the SGID bit set
-u file file has the SUID bit set

e) The test [[ … ]]

The double-bracket test differs from the single-bracket test for string comparison with the and !. Within single square brackets, these operators compare one string to another string; in double brackets, they compare a string to a pattern. A pattern can contain naming characters.
The following example illustrates this difference: how to test whether the variable TERM contains vt100 or vt220 or vt330 or vtxxx…

**Example**

```
if [ $TERM vt100 ] # 1 # correct, but insufficient
if [ $TERM vt* ] # 2 # FALSE
if [[ $TERM vt* ]] # 3 # correct: the correct solution
# between single brackets should be written:
if [ $TERM vt100 or $TERM vt220 or $TERM vt330 ]
```

In line 2 of this example, the shell replaces vt* with the list of all filenames that start with vt in the current directory before doing the test, which is probably not the intended purpose.

### IV.4.4 Integer Arithmetic

The use of integer variables is not planned in Bourne-shell, and arithmetic manipulations are very inconvenient there. In Bash, integer arithmetic is easy, thanks to an adapted notation: double parenthesizing ((…). Between double parentheses, bash interprets the characters < > () * according to their usual arithmetic meaning, and the character $ is not necessary in front of a name of variable. Parenthesizing is possible there, and without parenthesizing, the priority of the arithmetic operators is the usual priority. This notation provides a consistent framework for arithmetic assessment and arithmetic testing.

*a) The arithmetic evaluation: $((...))*

A few examples are better than a formal description.

```
declare i n1 n2 n3 x
n1 17
n2 3 # so far nothing new
n3 $((17/3)) # n3 17/3 5 (integer)
n3 $((17%3)) # %: the remainder of the division: n3 2
# the next line shows a more complex calculation
n1 $((n2*(n1+27) 5)) # n1 127
# the writing is very clear between (( and ))
# is the usual arithmetic notation.
x $((1<<4)) # << : shift left: x 16
```

*b) The arithmetic test: if ((...))*

The arithmetic test offers the same writing facilities. This is particularly practical for the characters < > which no longer designate redirections, but arithmetic relational operators.

**Examples**

if ((x>1000)); then # valid and very readable

# between [ ] should be written

if [ $x gt 1000 ]; then # less readable

# even more convincing:

if (( n2*(n1+27) 5) > n3 )); then # valid

*if ((record_size*nbr_record > bufferlength)); then*

Without the notation ((…)), the last two lines of the previous example would require completely illegible contortions.

Note that it is not necessary to declare integer variables with the *declare -i* command to use the evaluation and the arithmetic test. It is sufficient that the variables contain strings whose evaluation is an integer.

## IV.5 SCRIPT WRITING

The Bash is a very powerful language, its syntax is complex. We will briefly present the parenthesizing and substitution mechanisms, sources of syntax error.

### IV.5.1 Parentheses

A sequence of commands can be parenthesized by () or by { }:

(command1; command2; command3)

All three commands are executed in a subshell.

{ command1; command2; command3; }

All three commands are executed in the current shell.

Don't forget the space after { and the ; Before }.

### IV.5.2 Substitution

$variable or ${variable}

Designates the value of the variable. If variable is an array, $variable or ${variable} designates its first element. In a shell script, the notation ${variable} should be preferred to the notation $variable.

If variable is not defined, Bash substitutes the empty string.

${#variable} Designates the length of the variable (or of its first element if it is an array). Special case: ${*} where $* designates the number of position variables.

${#variable[*]} Designates the number of elements in the variable array.

${variable: string}

Designates the value of the variable if it is defined and not empty, otherwise designates the string string. Very useful to set a default value, for example:

*TERM ${TERM: VT220}*

*${variable:=string}*

Designates the value of the variable if it is defined and not empty. Otherwise, the string value is assigned to it, then the substitution takes place.

*${variable:?string}* Designates the value of the variable if it is defined and not empty. Otherwise, string is sent by the shell to standard output. If this shell is not interactive, it finishes.

*${variable:+string}* Designates the string string if the variable is defined and not empty. Otherwise, denotes the empty string.

**IV.5.3 *The command substitution $(command)***

Bash recognizes the POSIX $(command) notation, many clearer when used at several levels of nesting. The following example illustrates this possibility. How to copy in the /tmp/backup directory all the files whose name is *log* and which are in the directories listed by the PATH variable. The tr command, which allows you to replace the character: with a
space.

*xstra > cp $(find $(echo $PATH | tr ':' ' ') *

*–type f –name '*log*' –print) /tmp/backup*

*xstra>*

**IV.5.4 Bash extensions in name generation**

The Bash has an extensive filename generation mechanism for describing filenames that match certain patterns. Its syntax differs from regular expression syntax used by sed and awk. The syntax is summarized as follows:

**(pattern)   0 or 1 or more occurrences of pattern*
*+(pattern)  1 or more occurrences of pattern*
*?(pattern)    0 or 1 occurrence of pattern*
*@(pattern1|pattern2) pattern1 or pattern2*
*!(pattern)      anything but pattern*

This mechanism can be used in a test, which makes it very practical in programming. The following examples illustrate this possibility.

**Examples**

1) Delete all files except files with .c or .h extension and files whose names begin with Makefile or makefile or README:

*rm !(*.c|*.h|[Mm]akefile*|README*)*

2) In a script, make a decision on the filename. If the file name has a .c or .h extension, do this, otherwise do this:

```
#!/bin/bash
for filename in *; do
if [ $filename @(*.c|*.h|[Mm]akefile*|README*) ]
then
fi
done
```

Warning: Do not forget to include either in the script or in the file ~/.bashscript the *extglob* option using the command *shopt -s extglob*.

3) If the TERM variable contains vt100 or vt200 or vtxxx, or xterm, do

*if [ $TERM @(vt+([0 9])|xterm) ]; then*

### IV.5.5 Quoting (neutralization)

The special shell characters can be disabled in several different ways. This problem is encountered even more often in a shell script than in an interactive command line, and the question is: "What special characters are neutralized in a string placed between double apostrophes (double quotes)? »

Now that all the evaluation mechanisms have been presented, we can state a very simple rule to remember:

In a string between double quotes, all special characters are neutralized except the $ character in all its forms:

*Variable evaluation* *$TERM, ${TERM:-vt100}*

*Command evaluation* *$(cmd), and therefore also `cmd`*

*Arithmetic evaluation* *$((LINES*COLUMNS))*

We won't say more about programming in Bash. All of the programming description described above applies to writing scripts in Bash.

# Chapter V

## PROCESS MANAGEMENT

### V.1 INTRODUCTION

The concept of **process** is the most important in an operating system. All computer software is organized into a number of sequential processes. Simplifying, we can consider a process as a program which is executed and has an ordinal counter, to indicate to which instruction it is rendered, registers, variables and an execution stack (we will see later that a process can contain several threads, therefore several execution stacks). Its execution is, in general, an alternation of calculations carried out by the processor and Input/Output requests carried out by the peripherals.

### V.2 PROCESS FEATURES

#### V.2.1 States of a process

When a process runs, it changes state, as can be seen in the figure V.1. It can then be in one of three main states:

- Elected : running.
- Ready : waiting for processor.
- Blocked : waiting for an event.

Initially, a process is in the ready state. It enters the execution state when the processor begins its execution. A process transitions from the running state to the ready state or when it is temporarily suspended to allow another process to run. It changes from the execution state to the blocked state if it can no longer continue its execution (request of an I/O). It then waits for an event (end of I/O). When the event occurs, it becomes ready again.

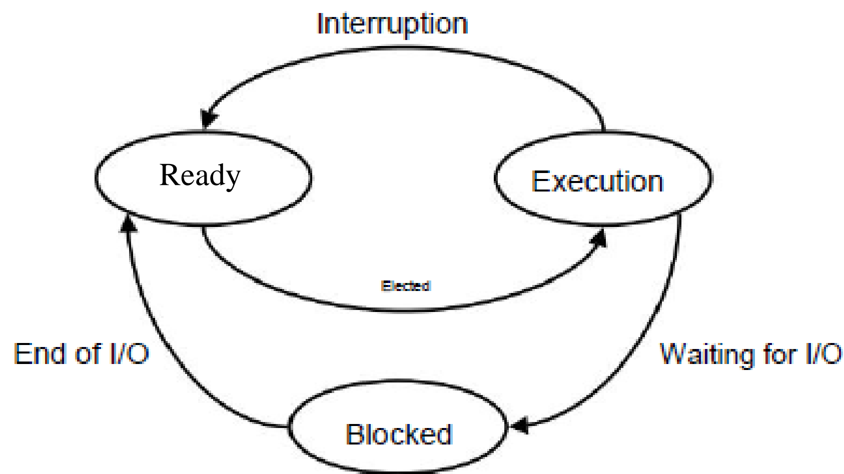



FIG V.1 The main states of a process.

The three-state model is complemented by two additional states: **New** And **END**, which respectively indicate the creation of a process in the system and its termination, as shown in FIG V.2.

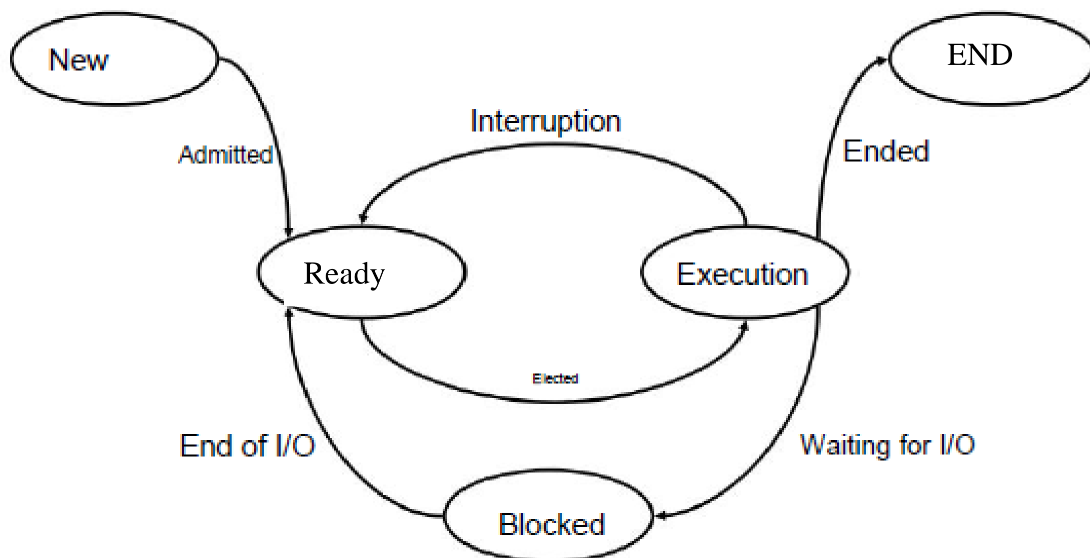



FIG V.2 Five-state model of a process.

### V.2.2 Process implementation

To manage the processes, the operating system saves several information in data structures. There is a table to hold information about all created processes. There is one entry per process in the table, called the Process Control Block (PCB).

#### V.2.2 .1 Process Control Block

Each table entry **PCB** includes information on:

- The pid of the process.
- The state of the process.
- Its ordinal counter (address of the next instruction to be executed by this process).
- Its memory allocation.
- Open files.
- The values contained in the registers of the processor.
- Anything that needs to be backed up when the execution of a process is suspended.

On the Table **V.1**, we show the main entries of the **PCB**.

Table V.1 Main entries of the Process table from Unix.

| Process management | Memory mangement | File management |
|---|---|---|
| Registers<br>Ordinal counter (OC)<br>Priority<br>Stack pointer (SP)<br>Process status<br>Launch time<br>Used CPU time<br>CPU time of sons<br>Signals<br>Pid<br>ppid | Segment pointer.code<br>Segment pointer.data<br>Segment pointer.pile | Root directory<br>Working directory<br>fd<br>uid<br>gid |

#### V.2.2.2 Process Context Switch

One of the main reasons for the existence of process control blocks is that in a multiprogrammed system we often need to give control of the CPU back to another process. It is therefore necessary to memorize all the necessary information in order to be able to restart the current process in the same state. For example, it is absolutely necessary to remember to which instruction it is rendered. FIG V.3 illustrates context switching between processes. The current process is interrupted and a scheduler is possibly called. The scheduler runs in mode kernel to be able to handle the **PCB**.

It is important to note that switching to kernel mode by a system call does not necessarily imply a context switch. We generally stay in the same process, except that we have access to data and instructions that are prohibited in user mode.

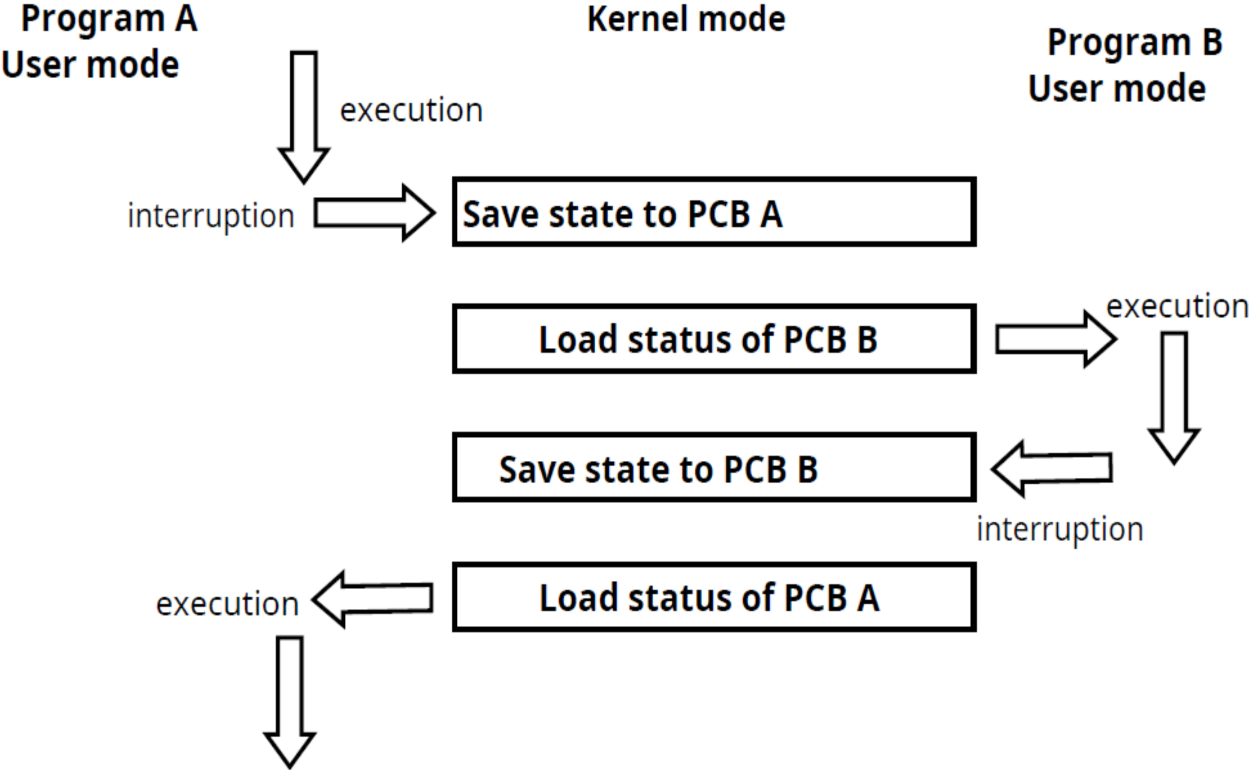



FIG V.3 The context switch.

### V.2.3 Daemons

The **daemons** are particular processes. A daemon always runs in the background. This implies that his father is not waiting for the end of his execution. Daemons are not associated with any terminal or process log in of a user. They are always listening and waiting for an event to occur. They perform tasks periodically. The daemons normally start at the start of the operating system load and do not die. The daemons do not work: rather, they launch other processes to perform the tasks. Examples of demons include the **page thief** virtual memory or the **terminal daemon** who launches getty, then login when prompted by users.

### V.2.4 Process creation and termination

The operating system provides a set of system calls that enable the creation, destruction, communication, and synchronization of processes. Processes are created and

destroyed dynamically. A process can create one or more child processes which, in turn, can create child processes in a form of tree structure. The creator process is called the parent process.

In some systems, such as MS-DOS, when a process creates a child, execution of the parent process is suspended until the child process terminates. It's called the **sequential execution**. On the other hand, in Unix-like systems, the parent continues to run in competition with its children. That's what we call **asynchronous execution**. A created child process can share certain resources such as memory or files with its parent process or have its own resources. The parent process can control the use of shared resources and can have some authority over its child processes. Also, it can suspend or destroy its child processes. It can also put itself on hold for the end of the execution of its children. The address space of the child process is obtained by duplicating that of the parent. He can run the same program as his father or load another program. A process ends with a request for voluntary termination (system call exit ( )) or by a forced stop caused by another process (system call kill ( )).When a process terminates, all system resources allocated to it are freed by the operating system.

## V.3 SERVICES POSIX FOR PROCESS MANAGEMENT

Standard Posix defines a relatively small number of system calls for process management:

- pid_t fork() : Creation of child processes.
- int execl(), int execlp(), int execvp(), int execle(), int execv(): Services exec ( ) allow a process to run a different program (code).
- pid_t wait() : Wait for a process to complete.
- void exit ( ) : End the execution of a process.
- pid_t  getpid ( ) : Returns the process ID.
- pid_t  getppid ( ) : Returns the id of the parent process.

In Linux, the type pid_t normally corresponds to a *long int*.

### V.3.1 Process creation

#### V.3.1.1 Creating processes with system ( )

There is a way to create a subprocess in Unix/Linux, using the command system ( ), from the standard library of C <stdlib.h>. As arguments it receives the name of the command (and perhaps a list of arguments) in quotes.

It creates a child process by running the command:

*ls -l /*

in a shell. It must be remembered that system ( ) is not a system call, but a C function, which makes using the function system ( ) less efficient than a process creation system call. This will be discussed at the end of the section.

system.c

```
# include < stdlib.h>

int main ( )
{
   int  return_value ;
/* returns 127 if the shell could not run
returns -1 in case of error
Otherwise, returns the code of the command */
return_value = system( " ls  -l   /" ) ;
 return return_value ;
}
```

#### V.3.1.2 Creating processes with fork ( )

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

*#include <unistd.h>*

*int fork ( );*

fork ( ) is the only way to create processes, by duplicating an existing process1. The system call fork ( ) creates an exact copy of the original process, as shown in FIG V.4. But now there is a problem because both parent and child processes are running the same code. How then to distinguish the parent process from the child process? To solve this problem, we look at the return value of fork ( ), who 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 the process creation failed, if there is not enough memory space or if the maximum number of authorized creations has been reached.

This connection is shown in FIG V.5. The father and the son each have their own private memory image. After system call fork ( ), the value of pid receives the value 0 in the child process but it is equal to the identifier of the child process in the parent process. Look closely at the following examples:

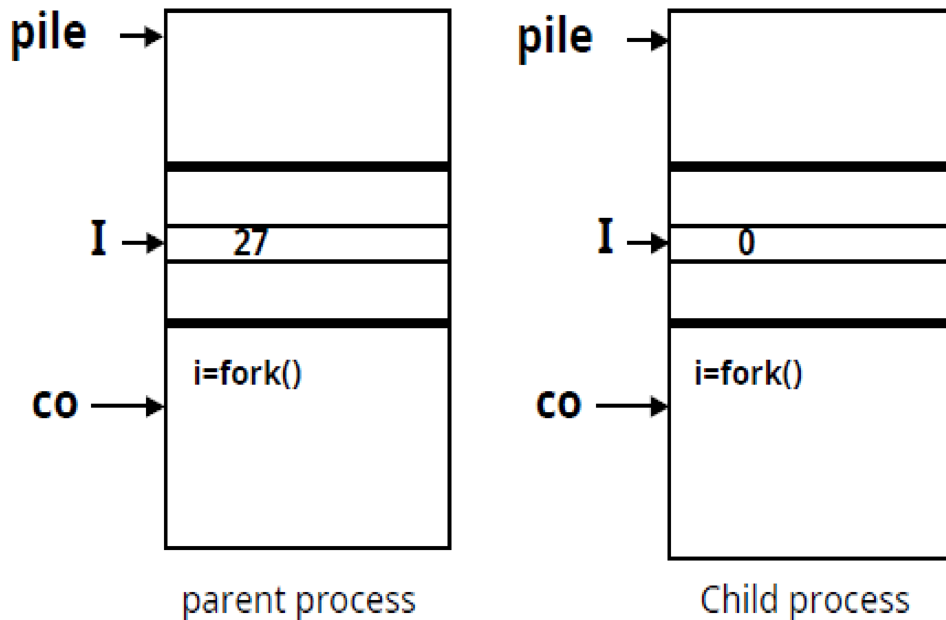



FIG V.4 Father process and its cloned child. i=27 after the call fork ( ) for the father and i=0 for the son. The memory image is the same for both, in particular the ordinal counter CO.

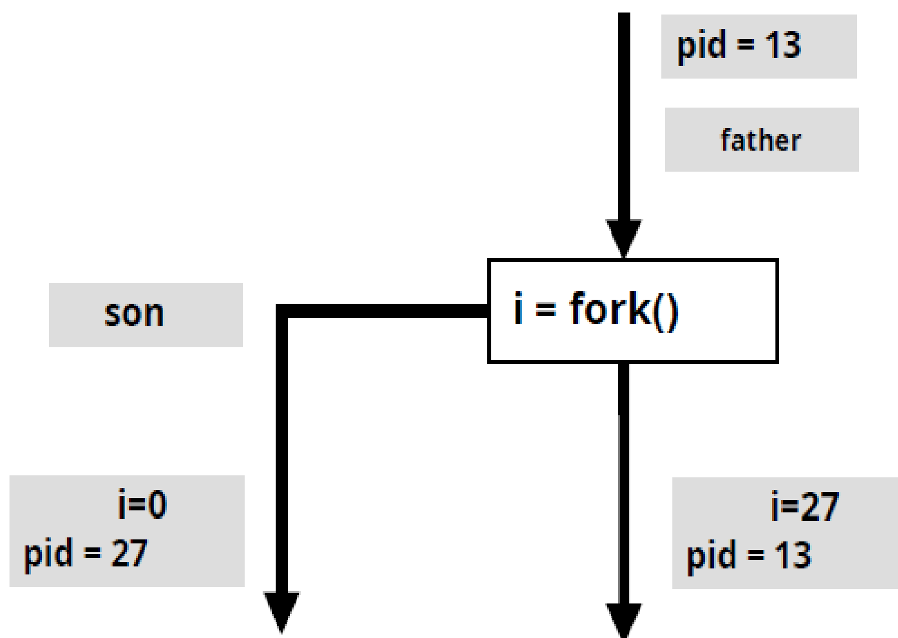



FIG V.5 Services Posix: fork ( ).

**Example 1.**After the execution of son.c father and son show their pid:

son.c

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

int   main ( )
{
pid_t  son _ pid;
son_pid=fork();
if (son_pid ==0)
        printf ("I am the son with pid %d\n", getpid( ));
else if (son_pid >0)
          printf ("I am the father with pid %d\n", getpid( ));
      else printf ("Error creating the son\n");
}
```

The execution of son.c show the pids of both father and son:

*leibnitz > gcc -o son son.c*

*leibnitz >  son*

*I am the father with pid 4130*

*leibnitz > I am the son with pid 4131*

Notice that the console redisplayed the prompt (leibnitz>) before the second message is displayed. This is because the parent process, which was launched from the console, terminated before the child process, and therefore returned control to the console before the child could display its message.

**Example 2.** Consider the following program in detail.

chaine.c

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

int  main (void) {
int i, j, k
       n=5;
 pid_t child_pid;

 for(i = 1; i<n; i++)
{
son_pid =fork ( );
if ( son _ pid > 0 ) // this is the father
   break;
printf  ("Process %d with parent %d\n", getpid( ), getppid ( ));
}
 // To waste the process' time
for (j = 1; j < 100000; d++)
for (k=1; k<1000; k++);
}
```

This program will create a chain of  n-1 process :

*leibnitz> gcc -o chaine chaine.c*

*leibnitz> chaine*

*Process 23762 with father 23761*

*leibnitz > Process 23763 with father 23762*

*Process 23764 with father 23763*

*Process 23765 with father 23764*

If we omit the final loops of the program, we get an unexpected result:

*leibnitz> gcc -o chaine chaine.c*

*leibnitz> chaine*

*leibnitz> Process 23778 with father  1*

*Process 23779 with father 1*

*Process 23780 with father 1*

*Process 23781 with father 1*

The problem here is that in all cases the parent process terminates before the ls process. As any process must have a parent, the orphaned process is therefore "adopted" by process 1 (the init process). Thus, we can notice that when the processes display the number of their father, they have already been adopted by the init process.

**Example 3.**The following program tfork.cpp will create two processes that will modify the variable a :

tfork.cpp

```
# include < sys/types.h>    // For the type fork
# include <stdio.h>         // For perror, printf

int main ( void )
{
pid_t  p ;
int   a = 20 ;
// creation of a son
switch ( p = fork ( ) )
{
case -1:
// the fork failed
per ror ( " The fork failed! " ) ;
break ;
case 0 :
// It's a son process
printf ( " Here is the son process, its PID %d.\n" , getpid ( ) ) ;
a + = 10 ;
break ;
defaul t :
// It's about the father process
printf ( " Here is the father process, its PID %d.\n" , getpid ( ) ) ;
```

```
a += 100;
}
//Both the two processes execute this instruction
printf ( "End of the process %d with a = %d.\n" , getpid ( ) , a ) ;
return 0 ;
}
```

Two executions of the program tfork.cpp show that the parent and child processes are concurrent:

*leibnitz> gcc -o tfork tfork.cpp*

*leibnitz> tfork*

*Here process father with pid 12339.*

*Here process son with pid 12340.*

*End of the  Process 12340 with a = 30.*

*End of the Process 12339 with a = 120.*

*leibnitz>*

*leibnitz> tfork*

*Here the process father with pid 15301.*

*End of the Process 15301 with a = 120.*

*Here process son with pid 15302.*

*End of the process 15302 with a = 30.*

*leibnitz>*

**Example 4.** The following examples are worth studying in detail. What is the difference between the programs fork1.c and fork2.c  next? How many sons will each be get?

fork1.c

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

int main ( )
{
int i , n=5;
```

```
int childpid ;
for ( i = 1 ; i<n ; i ++)
{
i f ( ( childpid=fork ( ) ) < = 0 ) break ;
printf ( " Process  %d with father %d , i=%d\n" , getpid ( ) , getppid ( ) , i ) ;
}return 0 ;
}
```

fork2.c

```
# include < sys/types.h>
# include < uni s td.h>
# include < stdio.h>

int main ( )
{
int i , n=5;
pid_t pid ;
for ( i = 1 ; i <n ; i ++)
{
i f ( ( pid=fork ( ) )==1)
break ;
i f ( pid == 0 )
pr i n t f ( " Process %d with father %d , i=%d\n" , getpid ( ) , getppid ( ) , i ) ;
}
return 0 ;
 }
```

The output of fork1.cis quite understandable. On each return from the fork( ) call, we exit the loop if we are in a child process (return value equal to 0), or if the call failed (negative return value). If the shell, which the father of the process creates when executing fork1, has a pid of 759, and the process itself a pid =904 , we will have the following output:

*leibnitz> gcc -o fork2 fork2.c*

*leibnitz>fork2*

*Process 904 with father 759, i=1*

*Process 904 with father 759, i=2*

*Process 904 with father 759, i=3*

*Process 904 with father 759, i=4*

On the other hand, the operation of fork2.c is more complex. If the call to fork() never fails, we know that the first process will create 4 new child processes. The first of these children continues to execute the loop from the current value of the counter. It will therefore itself create three processes. The second child will create two, and so on. And all of this repeats recursively with each of the threads. Assuming the shell still has a pid of 759, and that the process associated with fork2 has a pid of 874, we should get the tree shown in FIG V.6. Unfortunately, it won't be not the case, since again parent processes die before their children can display their messages:

*leibnitz> gcc -o fork2 fork2.c*

*leibnitz> fork2*

*Process 23981 with father 23980, i=1*

*Process 23982 with father 23980, i=2*

*Process 23983 with father 23980, i=3*

*leibnitz> Process 23984 with father 1, i=4*

*Process 23985 with father 23981, i=2*

*Process 23986 with father 23981, i=3*

*Process 23987 with father 1, i=4*

*Process 23988 with father 23982, i=3*

*Process 23989 with father 1, i=4*

*Process 23990 with father 1, i=4*

*Process 23991 with father 23985, i=3*

*Process 23992 with father 1, i=4*

*Process 23993 with father 1 , i=4*

*Process 23994 with father 1, i=4*

*Process 23995 with father 1, i=4*

We will see in the next section how to avoid this problem.

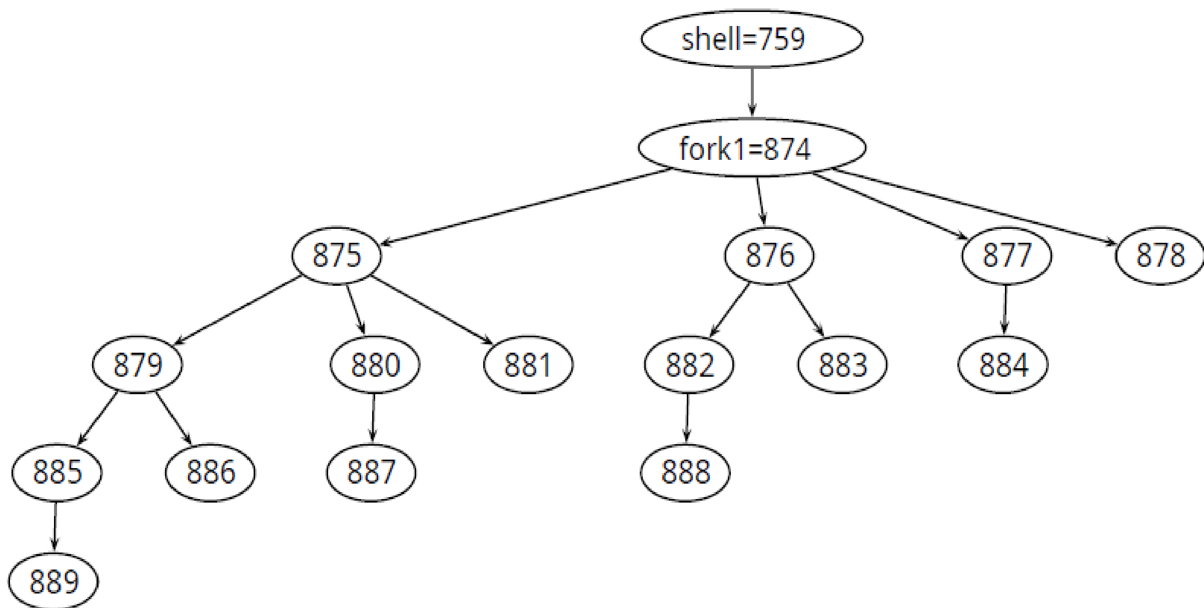



FIG V.6 Process tree structure that we would obtain with fork2.c if the parent processes would not die before the children could display their messages.

### V.3.2 System calls wait ( ), waitpid ( ) and exit ( )

These system calls allow the parent process to wait for the termination of one of its child processes and retrieve its termination status. Thus, a process can synchronize its execution with the end of its child process by executing the system call wait().The system call syntax is:

*pid = wait(status);*

where pid is the id of the child process and status is the address in user space of an integer that will contain the status of exit ( ) of the child process.

*#include <sys/wait.h>*

*int wait (int *status);*

*int waitpid (int pid, int *status, int options);*

*void exit(int return_code);*

- wait ( ) : Allows a father process to wait until a child process terminates. It returns the child process id and its termination status in &status.
- waitpid ( ) : Allows a father process to wait until the child process with the number pid ended. It returns the child process id and its termination status in &status.

- void exit ( ) : Used to voluntarily end the execution of a process and gives its termination status. It should be noted that a process can also be terminated by a forced stop caused by another process with the sending of a **signal** kill ( ).

The calling process is put on hold until one of its children terminates. When this happens, it returns from the function. Whether status is different from 0, then 16 bits of information are stored in the 16 least significant bits of the integer pointed to by status. This information shows how the process ended according to the following conventions:

- If the child is stopped, the 8 most significant bits contain the number of the signal which stopped the process and the 8 least significant bits have the octal value 0177.
- If the child ended with exit ( ), the least significant 8 bits of status are zero and the 8 most significant bits contain the 8 bits of least significant of the parameter used by the child process when calling exit ( ).
- If the child ended on the reception of a signal, the 8 most significant bits of status are zero and the 7 least significant bits contain the number of the signal which caused the end of the process.

In the header file <sys/wait.h> are defined different macros which allow to analyze the value of status in order to determine the cause of termination of the process. In particular, the parent process can obtain the value of the least significant 8 bits of the parameter which receives from exit ( ) from the son using the macro:

*WEXITSTATUS (status)*
*The definition of this macro is in <sys/wait.h>:*
*# define WEXITSTATUS(s) (((s)>>8)&0xFF)*

Of course, if the calling process wait ( ) has no living child processes, an error is generated.

**Example 5.** Remember that the program son.c as the default of returning to the shell before the child had a chance to display its message. Using the call wait, we can easily fix this problem:

wait.c

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

int main ( )
{
```

```
pid_t  son_ pid ;
son_ pid=fork ( ) ;
i f ( son _ pid ==0)
printf ( " I'm the son with pid %d\n" , getpid ( ) ) ;
else
      if ( son _ pid > 0 ) {
              printf ( " I'm the father with pid %d.\n" , getpid ( ) ) ;
              printf ( " I'm waiting the termination of my son \n" ) ;
              wait (NULL) ;
                    }
      else
             printf ( " Error in the son creation \n" ) ;
              exit ( 0 ) ;
}
```

**Example 6.** To avoid the problem of parent processes dying before their children in the program fork2.c, just add a loop on the call wait. As soon as there are no more children left, the call will immediately return a negative value. We will then know that we can go out without any problem. Here is the modified program:

fork2b.c

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

int main ( )
{
int i , n=5;
int pid ;
for ( i = 1 ; i <n ; i ++)
{ i f ( ( pid=fork ( ) ) == - 1)
            break ;
```

```
if ( pid == 0 )
   printf ( " Process %d with father %d , i=%d\n" , getpid ( ) , getppid ( ) , i ) ;
}
// Wait the termination if the sons
while ( wait (NULL) > = 0 ) ;
return 0 ;
}
```

**Example 7.** Creation of two threads (version 1), and use of the most significant 8 bits:

two_sons-1.c

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

void son ( int i ) ;
int main ( )
{
int status ;
if ( fork ( ) ) // creation of the first son
{
if ( fork ( ) = = 0 ) // creation of the second son
son ( 2 ) ;
}
else son ( 1 ) ;
if ( wait (&status )>0)
     printf ( "End of the son %d\n" , status >>8 ) ;
if ( wait (&status )>0)
    printf ( " End of the son %d\n" , status >>8 ) ;
return 0 ;
 }
```

```
void son ( int i )
{
sleep ( 2 ) ;
exit ( i ) ;
}
```

Two executions of the program two_sons-1.c:

*leibnitz > gcc -o two_sons-1 two_sons-1.c*

*leibnitz > two_sons-1*

*end of son 2*

*end of son 1*

*leibnitz > two_sons-1*

*end of son 1*

*end of son 2*

*leibnitz>*

**Example 8.** Creation of two threads (version 2), with the simulation of waiting for a non-existent third thread and the use of the macro WEXITSTATUS:

two_sons-2.c

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

void son ( int i ) ;
int main ( )
{
int status ;
pid_t  pid ;
// Creation of the first son
pid = fork ( ) ;
if ( pid == - 1 )
        perror ( " fork " ) ;
```

```
else
        if ( pid = = 0 )
            son ( 1 ) ;
 // Creation of the second son
    pid = fork ( ) ;
i f ( pid == -1 )
perror ( " fork " ) ;
else
        if ( pid = = 0 )
        son ( 2 ) ;
// Waiting of the termination of the two sons
if ( wait (&status )>0)
    printf ( "End of the son  %d\n" , WEXITSTATUS( status ) ) ;
else
      perror ( "wait " ) ;
    if ( wait (&status )>0)
         printf ( "End of the son %d\n" , WEXITSTATUS( status ) ) ;
     else
         perror ( "wait " ) ;
        // Here, we wait the termination of a third son that does not exist
       // An error will be returned
if ( wait (&status )>0)
   printf ( "End of the son %d\n" , WEXITSTATUS( s t a tus ) ) ;
else
    perror ( "wait " ) ;
return 0 ;
}
```

Execution of the program two_sons-2.c:

*leibnitz > gcc -o two_sons-2 two_sons-2.c*

*leibnitz > two_sons-2*

*end of son 2*

*end of son 1*

*wait: No child processes*

*leibnitz>*

### V.3.3 Zombie process

Asynchronous execution between parent and child processes has certain consequences. Often the child of a process ends, but its father does not expect it. The child process then becomes a **zombie process**, as shown in Fig. **V.7**. The child process still exists in the process table, but it no longer uses the resources of the kernel.

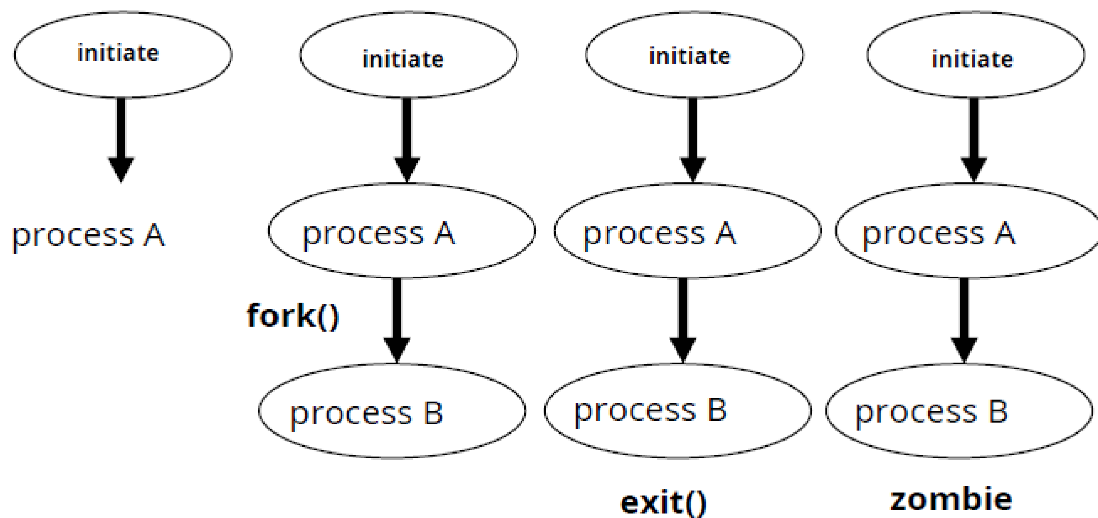



FIG V.7 Zombie process: the son dies and the father did not wait ( ).

The execution of the system call wait ( ) or waitpid ( ) by the parent process eliminates the child from the process table. There can also be premature terminations, where the father process terminates its child processes (FIG. **V.8**). In this situation, its child processes are adopted by the process initiate, of which the pid =1.

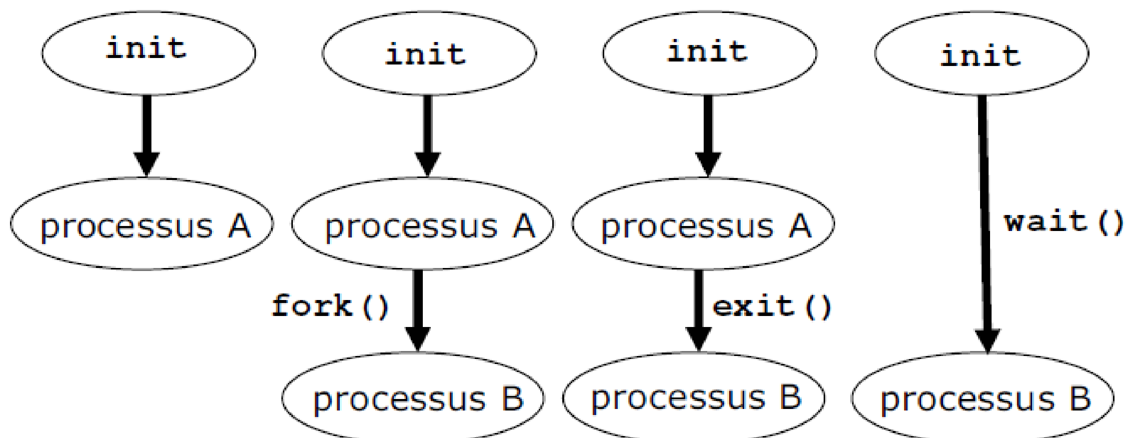



FIG V.8 Orphan process: the father dies before the son. init adopt the son.

**Example 9.** Generate zombies is relatively easy. The program zombie.c shows the creation of a zombie child process for 30 seconds:

zombie.c

```
# include <unistd.h> //For sleep
# include <stdlib.h>
# include <sys/types.h>

int main ( )
{
pid_t pid ;

// process son
pid = fork ( ) ;
i f ( pid > 0 )
{
// Father : sleep 30 secondes
sleep (30) ;
}
else
{
// Son : exit immediately
exit (0) ;
}
return 0 ;
}
```

Execution of zombie.c in the background: the command ps allows to see his zombie state before and after his final disappearance:

*leibnitz > gcc -o zombie zombie.c*

*leibnitz > zombie &*

*[1] 5324*

*leibnitz > ps -u jmtorres*

*PID TTY TIME CMD*

*2867 pts/0 00:00:00 tcsh*

*5324 pts/0 00:00:00 zombie*

*5325 pts/0 00:00:00 zombie <defunct>*

*5395 pts/0 00:00:00 ps*

*leibnitz>*

*[1] Done zombie*

*leibnitz>*

*leibnitz> ps -u jmtorres*

*PID TTY TIME CMD*

*2867 pts/0 00:00:00 tcsh*

*5396 pts/0 00:00:00 ps*

*leibnitz>*

**V.3.4 The family of system calls exec**

A son process created can replace its program code with another program. The Unix system offers a family of system calls executive which allow to change the image of a process (FIG. V.9). All system calls executive replace the current process with a new process built from an ordinary executable file. Text and data segments from the process are replaced with those from the executable file.

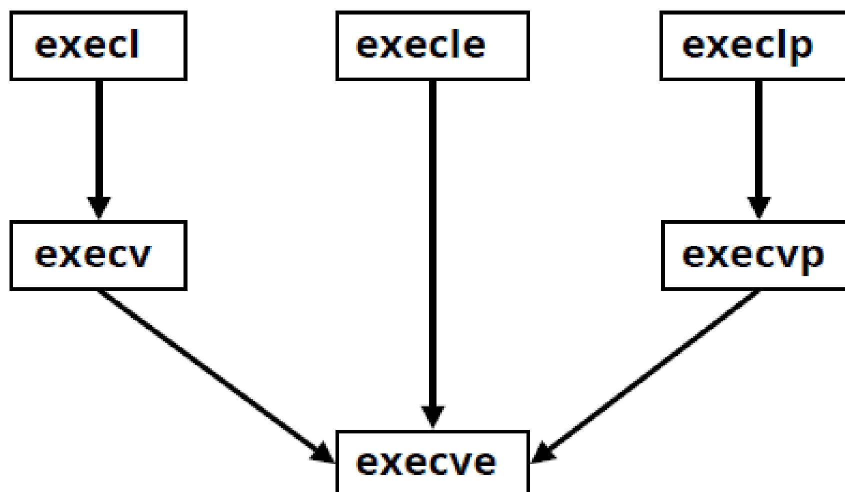



FIG V.9 Services Posix: exec.

*#include <unistd.h>*

*int execl (const char *path, const char *argv, ...);*

*int execv (const char *path, const char *argv[ ]);*

*int execle (const char *path, const char *argv, const char *envp[ ]);*

*int execlp (const char *file, const char *argv, ...);*

*int execvp (const char *file, const char *argv[ ]);*

- execl ( ):Allows you to pass a fixed number of parameters to the new program.
- execv ( ):allows you to pass a free number of parameters to the new program.
- execle ( ): same operation as execl ( ) with a supplementary argument envp which represents a pointer to an array of pointers to character strings defining the environment.
- exelp ( ):Same interface and action as execl ( ), but the difference is that if the filename is not a full name -relative to the root- the system will use the command search path -the paths given by the variable PATH- to find out which directory the program is in.
- exevp ( ):Same interface and action as execv ( ),but the difference is that if the file name is not a full name, the command uses the directories specified in PATH. The Unix convention is that each string has the form name=value. So the arguments for execv ( ) can be passed, for example:

```
char *arguments[4]
...
arguments[0]="/bin/ls";
arguments[1]="-l";
arguments[2]="/etc";
arguments[3]="NULL";
execv("/bin/ls", arguments);
...
```

After executing a family system call exec ( ), the memory image of a process is over written by the new memory image of an executable program, but the pid do not change. The content of the user context that existed before the call to exec ( ) is no longer accessible.

**Example 10.** The program fork-exec.c shows system call usage fork ( ) and execvp ( ) :

fork-exec.c

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

```
int main ( int argc , char * argv [ ] )
{
pid_t  son _ pid ;
//List of  arguments for the command  " ls "
char * args [ ] = {  " ls " , "-l " , argv [ 1 ] , NULL} ;
// Duplicate the process
son_ p i d = fork ( ) ;
if ( son_ pid ! = 0 ) {
  // It's the father
   waitpid ( son_ pid, NULL, NULL) ;
   printf ( "Principal Program terminated \n" ) ;
   return 0 ;
}
else
{
   // Execute the program with the arguments from PATH
        execvp ( "ls", args ) ;
        // Return in case of error
        perror ( " Error in execvp" ) ;
        exit (1) ;
}
return 0 ;
}
```

Exécution of fork-exec.cpp :

*leibnitz> gcc -o fork-exec fork-exec.cpp*

*leibnitz> fork-exec /*

*total 113*

*drwxr-xr-x 2 root bin 4096 Aug 22 12:00 bin*

*drwxr-xr-x 2 root root 4096 Jan 20 15:46 boot*

*drwxr-xr-x 2 root root 4096 Aug 22 16:43 cdrom*

*drwxr-xr-x 14 root root 40960 Jan 20 16:10 dev*

*drwxr-xr-x 33 root root 4096 Jan 22 15:55 etc*

*drwx------ 2 root root 16384 Aug 22 11:18 lost+found*

*drwxr-xr-x 5 root root 4096 Mar 16 2002 mnt*

...

*drwxrwxrwt 8 root root 4096 Jan 22 16:44 tmp*

*drwxr-xr-x 20 root root 4096 Jan 20 16:04 usr*

*drwxr-xr-x 17 root root 4096 Aug 22 16:43 var*

*Principal Program ends*

*leibnitz>*

**V.3.5 Comment on system ( )**

The execution of the library function system ( ) is essentially a fork ( ) + exec ( ) + wait ( ), and it may be even worse than that, because it invokes a shell to execute commands, which is very resource intensive, because the shell has to load scripts and initializations. Although system ( ) may be useful for small programs that run very occasionally, it is strongly discouraged for other applications. For example, the single call:

*system("ls -l /usr")*

is an incredible waste of resources. First we perform a fork ( ), then exec ( ) for the shell. Then the shell initializes and does another fork ( ) and one exec ( ) for launching " ls ". It is preferable to use system calls for reading directories which are less expensive than the function system ( ).

**Example 11.** The programs father.cpp and son.cpp show the creation of the processes with the combination fork ( ) and execl ( ):

father.cpp

```
# include < unistd.h> // For fork and execl
# include < sys/wait.h> // For wait
# include < s tdio . h> // For printf

int main ( void )
{
int p , child , status ;
p=fork ( ) ;
if ( p == -1)
   return  - 1;
```

```
if ( p>0 )     // It's the father as p > 0
{
printf (" Here the father [%d ], my child [ %d]\n" , getpid ( ) , p ) ;
if ( ( child=wait (&status ) )>0 )
     printf ( " Here the father [%d ] , end of the child [%d]\n" , getpid ( ) , child ) ;
   // Sleep one second
     sleep( 1 ) ;
      printf ( "The father [%d ] ends \n" , getpid ( ) ) ;
}
else
{ // It's the child
   if ( (status=e xecl("/home/.../inf3600 /logiciel/a.out", "a.out " , NULL)) == -1)
         printf ( "Sorry, the program does not exist : %d\n", status ) ;
   else
        printf ( "This instruction is never executed\n" ) ;
}
```

child.cpp

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

int main ( )
{
printf ("Here the program child [%d] \n", getpid ( ));
return 0 ;
}
```

The execution of the programs father.cpp and child.cpp; Let be the executable file of child.cpp named *a.out* :

*leibnitz> gcc child.cpp*

*leibnitz> gcc -o father.cpp*

*leibnitz> father*

*Here the father 19909, my child 19910*

*Here the program child 19910*

*Here the father 19909, end of the child 19910*

*The father 19909 ends*

*leibnitz>*

**Example 12.**Use of execvp ( ) :

texecvp.c

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

int main ( int argc , char * argv [ ] )
{
i f (fork ( ) == 0) {
     execvp (argv [ 1 ], & argv [ 1 ] ) ;
     fprintf  (stderr , "we can't execute %s \n " , argv [ 1 ] ) ;
      }
else
          if (wait(NULL) > 0)
             printf ("The father detects the end of the child \n" ) ;
return 0 ;
}
```

Exécution of the program texecvp.c :

*leibnitz> gcc -o texecvp texecvp.cpp*

*leibnitz> texecvp date*

*Wed Sep 4 15:42:06 EDT 2002*

*The father detects the end of the child*

leibnitz> texecvp ugg+kju

*We can't execute  ugg+kju*

*The father detects the end of the child*

leibnitz> gcc -o child child.c

leibnitz> child

*Here the program child [20498]*

leibnitz> texecvp child

*Here the program child [20498]*

*The father detects the end of the child*

leibnitz>

**Bibliography**

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Operating System Concepts, 10th Edition, Wiley, 2021.

- Greg Gagne, Peter B. Galvin, Peter Galvin, Avi Silberschatz, Applied Operating System Concepts, Wiley, 2003.

- Andrew Tanenbaum, Herbert Bos ,Modern Operating Systems, 4 th edition, Prentice Hall, New Jersey, 2014.

- Ethem Mining, Linux for Beginners: A Practical and Comprehensive Guide to Learn Linux Operating System and Master Linux Command Line. Contains Self-Evaluation Tests to Verify Your Learning Level Paperback, Everooks Ltd , 2020.

- Ida M. Flynn, Ann McIver-McHoes, Understanding Operating Systems, Third Edition, Cengage Learning, 2000.

- Mary S. Gorman; S. Todd Stubbs, Introduction To Operating Systems, A Survey Course, Course Technology, 2000.

- William S. Davis, Operating Systems: A Systematic View, 4th edition, Addison-Wesley, 1992.

- Andrew Tanenbaum, Albert Woodhull, Operating Systems Design and Implementation, Pearson, 2006.