Chapter 1

Processes

1.1. Process structure

1.1.1. General information

Processes correspond to the execution of tasks: user programs, input/output, etc. by the system. An operating system generally has to handle several tasks at the same time. Since most of the time it only has one processor, it solves this problem by using pseudo-parallelism. It processes one task at a time, pauses and moves on to the next. Because switching tasks is so fast, the computer gives the illusion of simultaneous processing.

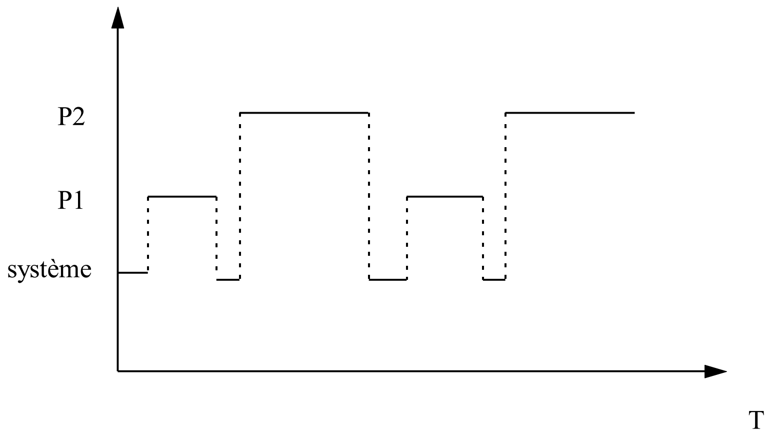


Figure Erreur ! Argument de commutateur inconnu. **Multi-tasking**

User processes are launched by a command interpreter. They can themselves launch other processes. The parent process is referred to as the "father", and the processes created are referred to as the "children". Processes can therefore be structured in the form of a tree. When the system is launched, there is only one process, which is the ancestor of all the others.

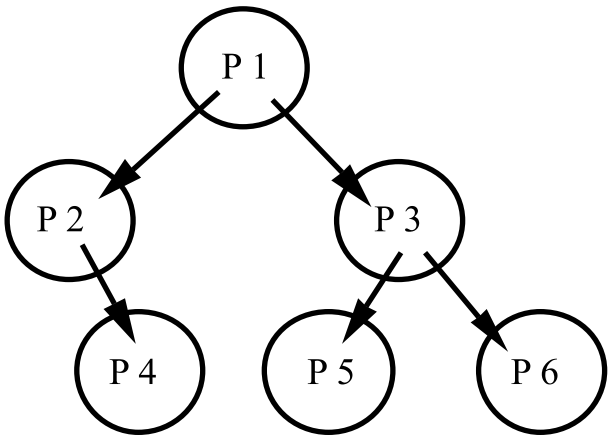
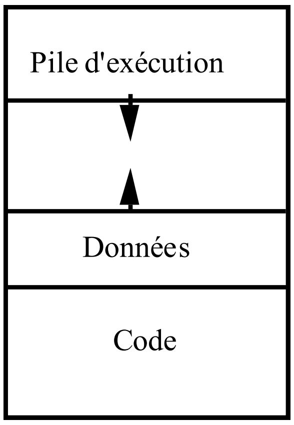


Figure Erreur ! Argument de commutateur inconnu. **The process hierarchy.**

Processes consist of a memory workspace made up of 3 segments: the stack, the data and the code, and a context.



FigureErreur ! Argument de commutateur inconnu. **Process segments.**

The code corresponds to the instructions, in assembly language, of the program to be executed. The data area contains the program's global or static variables and dynamic memory allocations. Finally, function calls, with their parameters and local variables, are stacked on the stack. The stack and data zones have moving boundaries that grow in opposite directions as the program is executed. Sometimes, the data zone is divided into the data itself and the heap. The heap is then reserved for dynamic data.
The context is made up of the data needed to manage the processes. A table contains a list of all the processes, and each entry retains its context[1] . The elements of the Minix process table, for example, have the simplified form of the table below.

Process	Memory	Files
Registers	Code pointer	umask mask
Ordinal counter	Data pointer	Root directory
Programme status	Pointer on battery	Work directory
Battery pointer	Completion status	File descriptors
Date of creation	Proc. signal no. Killed	effective uid
CP time used	PID	effective gid
Wire CP time	Parent process	
Date of next alarm	Process group	
Message pointers	real uid	
Waiting signal bits	effective uid	
PID	real gid	
	effective gid	
	Signal bits	

TableErreur ! Argument de commutateur inconnu. **The Minix context.**

The number of locations in the process table is limited for each system and each user.
The switching of tasks from one to another is carried out by a scheduler at the lowest level of the system. This scheduler is activated by interrupts:

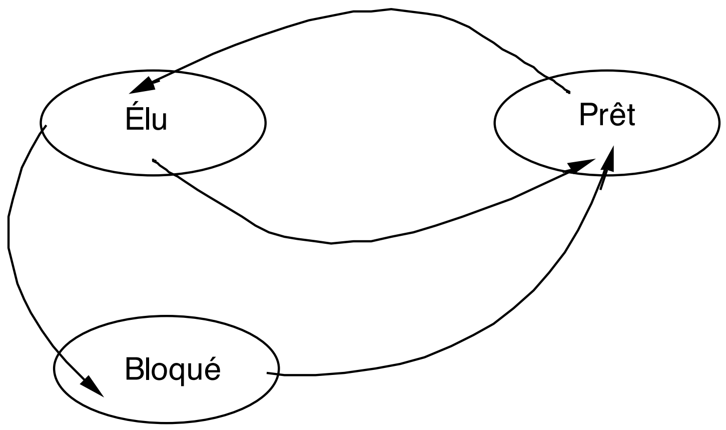
- clock,
- disc,
- terminals.

Each interrupt has a corresponding interrupt vector, a memory location containing an address. The arrival of an interrupt causes a branch to be made to this address. An interrupt generally leads to the following operations:

- stack of the ordinal counter, the status register and possibly other registers. The interrupt software procedure saves the elements of this stack and the other registers in the context of the current process;
- introduction of a new stack for handling interruptions;
- call the scheduler;
- election of a new programme;

[1] Bach, op. cit. p. 158 and Tanenbaum, op. cit. p. 60.

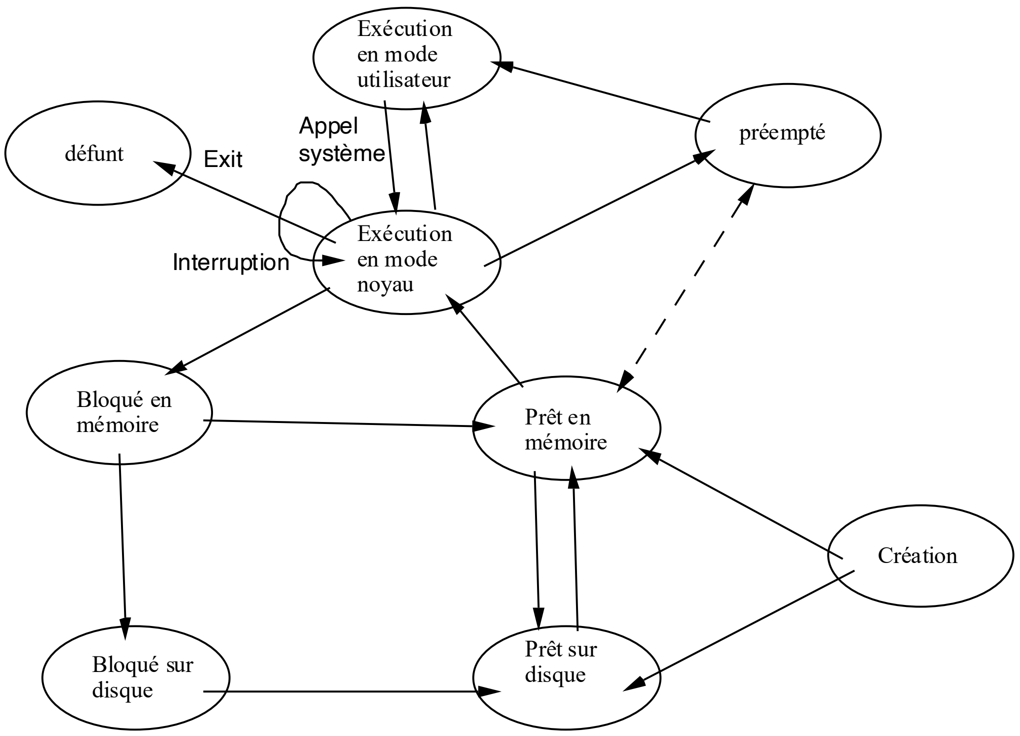
Explanation of the clock interrupt, then the disc.
We have now seen that a process can be active in main memory (Elected) or suspended, waiting to be executed (Ready). It can also be Blocked, waiting for a resource, for example during a disk read. The simplified diagram of the states of a process is therefore :



FigureErreur ! Argument de commutateur inconnu. **Process states.**

The process passes from the elected state to the ready state and vice versa during an intervention by the scheduler. The scheduler is triggered, for example, by a clock interrupt. It can then suspend the current process, if it has exceeded its time quantum, to elect one of the ready processes. The transition from the elected state to the blocked state occurs, for example, when a disk is read. This process will change from the blocked state to the ready state when the disk responds. This transition generally corresponds to the release of a resource.
In Unix, a process is executed in two modes: user mode and kernel mode. Kernel mode corresponds to calls to kernel code: write, read, etc. User mode corresponds to other instructions.
A process in kernel mode cannot be suspended by the scheduler. It switches to the preempted state at the end of its execution in this mode (unless it is blocked or destroyed). This state is virtually the same as ready in memory.
In addition, when memory cannot contain all the ready processes, a certain number of them are moved to disk. A process can then be ready in memory or ready on disk. Similarly, processes may be blocked in memory or blocked on disk.
Finally, terminated processes are not immediately removed from the process table (they are only removed by an explicit instruction from their parent). This corresponds to the dead state.
The states of a Unix process[2] are :

[2]Bach, op. cit., p. 156.



FigureErreur ! Argument de commutateur inconnu. **Unix process states.**

1.1.2. Unix processes

The fundamental functions

A process can be duplicated (and therefore add a new process) using the :

pid_t fork(void)

which returns -1 on failure, 0 in the child process, and the child process number (PID) in the parent. This function transmits part of the context from the father to the son: descriptors of standard files and other files, redirections, etc. Both programs are sensitive to the same interrupts. After a fork(), the two processes run simultaneously.
A new process can only be created by "overwriting" the code of an existing process using the :

int execl(char *ref, char *arg0, char *argn, 0)

ref is a character string giving the address of the new program to be substituted and executed. arg0, arg1, ..., argn are the arguments to the program. The first argument, arg0, is the program name.
The execle() and execlp() functions have similar arguments and effects. int execv(char *ref, char *argv[]) and execve() and execvp() are similar.
These functions return -1 in the event of failure.
For example:

```
void main()
{
        execute("/bin/ls", "ls", "-l", (char *) 0);
/* with execlp, the first argument may not be
only ls and not /bin/ls */
        printf("no print");
}
```

Creating a new process - occupying a new location in the process table - therefore involves duplicating an existing process. The latter will be the father. The code in the child process is then replaced by the code you want to run. Initially, the initial process launches all the processes useful to the system.
A process terminates when it has no more instructions or when it executes the :

void exit(int status)

A completed process can only be removed from the table by its parent, using the :

int wait(int * output_code)

With this instruction, the father is blocked waiting for the end of a son. It returns the number (PID) of the first dead child found. The value of the exit code is linked to the exit parameter of this child. The wait instruction can therefore be used to find out the eventual return value, supplied by exit(), of a process. This mode of use is analogous to that of a function. wait() returns -1 in the event of an error.
If the child terminates without its father waiting for it, the child passes to the defunct state in the table. If, on the other hand, the father terminates before the son, the latter is attached to the initial process. The initial process spends most of its time waiting for orphan processes.
Using the 3 instructions, fork(), exec(), and wait(), we can write a simplified command interpreter. It takes the following form:

```
while(1) {
                        /* awaits order */
          read_command(command, parameters);
          if (fork() != 0) {
                    wait(&status);                    /* proc. father */
          } else {
                    execv(command, parameters);       /* child proc */
          }
}
```

Signals

Instruction :

int kill(int pid, int signal)

allows a process to send a signal to another process. pid is the number of the process to be destroyed and signal, the number of the signal used. The kill function corresponds to software interrupts.
By default, a signal causes the destruction of the receiving process, provided, of course, that the sending process has the right to destroy.
The list of signal values includes :

Signal name	Signal no.	Comments
SIGUP	1	signal emitted on disconnection
SIGINT	2	^C
SIGQUIT	3	^\
SIGKILL	9	radical" interruption signal
SIGALRM	? The values are defined by macros in signal.h	signal emitted by alarm(int sec) after sec seconds
SIGCLD	?	signal sent by a son to his father at the end of his life

TableErreur ! Argument de commutateur inconnu. **Some Unix signals.**

kill() returns 0 on success and -1 on failure.
A process can modify ("mask") its behaviour to the signals received when the function is called:

void (*signal(int signal, void (* function)(int)))

with a function that can take the values:

Name	Action
SIG_IGN	the process will ignore the corresponding interrupt
SIG_DFL	the process will revert to its default behaviour when the interrupt arrives (termination)
void function(int n)	the process will execute a user-defined function when interrupt n occurs. It will then resume from the point at which it was interrupted

TableErreur ! Argument de commutateur inconnu. **Unix interrupt routines.**

Calling the signal function sets the state of the signal bits in the process table.

In addition, the signal(SIGCLD, SIG_IGN) function allows a parent to ignore the return of its children without the latter cluttering up the table of processes in the defunct state.

Some Unix identification functions

The Unix ps -ef command gives a list of the processes running on the system. Each process has an identifier and a group. These are obtained using the functions :

int getpid(void)

and

int getpgrp(void)

Sons inherit the father's process group. This group can be changed using the :

int setpgrp(void)

Each process also has a real and effective user. These are obtained using the functions :

int getuid(void)

and

int geteuid(void)

The real user is the user who started the process. The effective user number will be used to check certain access rights (files and sending signals). It normally corresponds to the real user number. However, you can set the effective user of a process to the owner of the executable file. This file will then run with the rights of the owner and not with the rights of the person who launched it. The :

int setuid(int euid)

is used to switch these user numbers from one to the other: real to effective and vice versa. It returns 0 if successful.

1.2. Execution threads

Graphical user interfaces, multi-processor systems and distributed systems have led to a revision of the usual conception of processes. This revision is based on the notion of threads *of control*. A traditional process is made up of an address space and a single thread of control. The idea is to associate several "threads of control" with an address space and a process. Execution threads are therefore degraded processes (without their own address space) within a process or application. They are also known as lightweight processes.

1.2.1. Why execution threads

Some applications duplicate themselves entirely during processing. This is particularly the case for client-server systems, where the server executes a fork to process each of its clients. This duplication is often very costly. With threads, the same result can be achieved without wasting space, by creating only one control thread for a new client, and keeping the same address, code and data space. Threads are also very well suited to parallelism. They can run simultaneously on multi-processor machines.

1.2.2. *What are execution threads made of?*

Components of a control wire	Elements of an ordinary process
Programme counter	Address space
Battery	Global variables
Registers	Table of open files
Threads	Child processes
Status	Counters
	Semaphores

TableErreur ! Argument de commutateur inconnu. **The comparative context of a control thread and a process.**

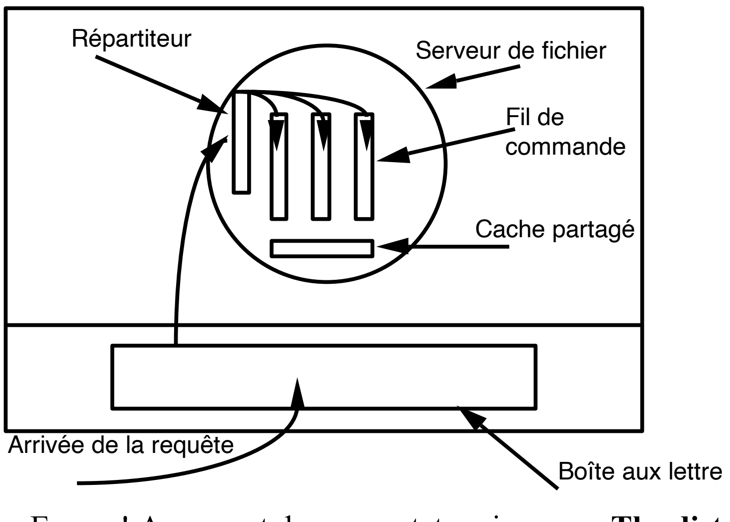
The elements of a process are common to all threads in that process.

1.2.3. *Using execution threads*

The essential advantage of threads is that they allow parallel processing at the same time as blocking system calls. This requires special co-ordination of the execution threads of a process. In optimal use, each thread has one processor. A. Tanenbaum, in *Modern Operating Systems*, distinguishes three possible organizational models: the *dispatcher/worker* model, the team and the pipeline.

The dispatcher/worker model

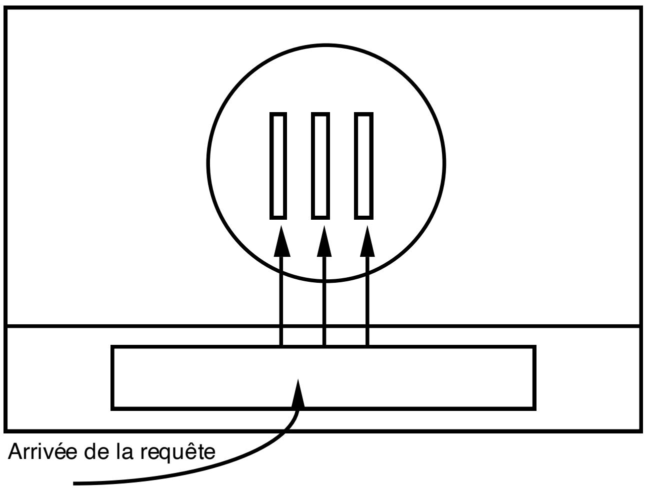
In this model, requests to a server arrive in a mailbox. The dispatcher control thread reads these requests and wakes up a worker control thread. The worker thread manages the request and shares the server's cache memory with the other threads. This architecture offers significant savings because it prevents the server from blocking during each input/output.



FigureErreur ! Argument de commutateur inconnu. **The distributor**

The team

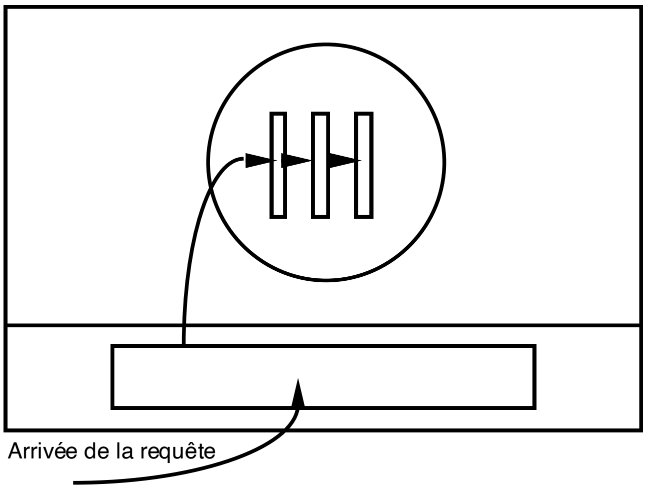
In the team model, each control thread handles one request. There is no dispatcher. Instead, a job queue is implemented.



FigureErreur ! Argument de commutateur inconnu. **The team**

The pipeline

In the pipeline model, each control thread handles part of the request. Several hardware architectures implement this type of organization. However, this model is not appropriate for all applications.



FigureErreur ! Argument de commutateur inconnu. **The *pipeline***

1.2.4 Implementing execution threads with Java

The Java language can be used to build applications and applets. Applets are small programs that can be loaded into a network browser. Java, thanks to its base classes, provides the ability to create threads at the programming language level.

Create an order thread

A thread can be created by creating a subclass of the Thread class and then creating an object belonging to this subclass. The object will inherit the methods (member functions) of the class and these will enable it to be managed.
The methods of the Thread class include :

- start(), which starts the command line;
- stop() which stops it permanently;
- run() the body of the thread which is executed after start() ;
- sleep() which puts it to sleep for a configurable time;
- suspend() which suspends it - puts it on hold - until further notice ;

- resume(), which resumes execution of the command line;
- yield() which lets its turn to execute pass.

Java's child execution model can be represented by a state machine.

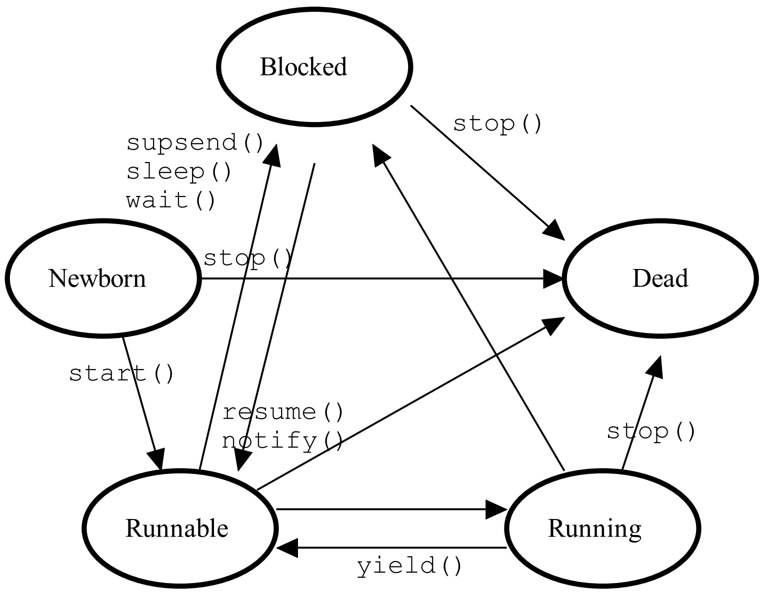


Figure Erreur ! Argument de commutateur inconnu. **The states of Java's execution threads.**

An example

The following example creates and launches two threads. Each command thread prints its name as it executes and its number of passes. The programme must be run using the java interpreter.

```
class ExThread {
        public static void main (String args[]) {
                new ThreadAffiche("Me").start();
                new ThreadAffiche("Other").start();
        }
}


class ThreadAffiche extends Thread {
        public ThreadAffiche(String str) {
                super(str);
        }
        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println(getName()+ " " + i + " passage(s)");
                        try {
                                sleep((int)(Math.random()*1000));
                        } catch (InterruptedException e) {}
                }
                System.out.println("Done" + getName());
                this.stop();
        }
}
```

1.2.5. *Implementing threads in Windows NT*

Threads can be implemented in user space or in the kernel. At present, few operating systems include threads in their kernel. Several external libraries exist, for example Sun's lightweight processes. This is not the case with Windows NT, which was designed from the outset with threads.
Windows NT can be programmed at two levels: in C or in C++. Programming in C is done using the APIs in the *Software Development Kit*. Programming in C++ is done using *Microsoft Foundation Classes*, which encapsulate the system entities.
A process in Windows is an instance of an application that has an address space, memory, files and open windows. A new process is created using the CreateProcess() function. This process will include a command line. You can add others to it using the :

HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD cbStack, LPTHREAD_START_ROUTINES, lpSStartAddr, LPVOID lpThreadParm, DWORD fdwCreate, LPDWORD lpIDThread).

A control thread can ensure the exclusivity of a resource by using the WaitForSingleObject() function, which corresponds to a P in a mutual exclusion semaphore. The semaphore is released using ReleaseMutex(). The InterLockedIncrement() and InterLockedDecrement() functions are applicable to generalised semaphores.

1.3. Scheduling

Scheduling regulates the transitions from one state to another of the various processes. The objectives of scheduling are to:
1. to maximize processor utilization;
2. to be fair between the different processes;
3. an acceptable response time;
4. to perform well;
5. to ensure certain priorities;

CPU scheduling is the method by which the operating system decides which process or thread to execute from the ready queue. It assigns CPU time to the selected process based on various scheduling algorithms. The objective is to optimize system performance by ensuring efficient CPU utilization, improving responsiveness, and maintaining fairness across processes.
What are some common CPU scheduling algorithms?
First-Come, First-Served (FCFS): The process or thread that has been in the ready queue the longest is selected for CPU allocation.
Shortest Job First (SJF): The process with the shortest expected execution time is given CPU time next.
Priority Scheduling: Processes are assigned priorities, and the one with the highest priority is allocated CPU resources.

1.3.1. The Round Robin Algorithm (RR)

This algorithm is one of the most widely used and reliable. Each ready process has a quantum of time in which to run. When it has used up this time or gets stuck, for example on an I/O, the next process in the queue is elected and replaces it. The suspended process is queued.
The only important parameter to set for the RR is the duration of the quantum. This must minimize the system management time and still be acceptable to users. The proportion of system management corresponds to the ratio of switching time to quantum duration. The longer the quantum, the lower this proportion, but the longer users wait for their turn. Depending on the machine, a compromise might be between 100 and 200 ms.

1.3.2. Priorities

In the turnstile algorithm, equal quanta make the different processes equal. It is sometimes necessary to give some processes priority over others. The priority algorithm chooses the process with the highest priority.
These priorities can be static or dynamic. System processes will have strong static (non-modifiable) priorities. User processes will have their priorities modified during execution by the scheduler. For example, a process that has just been executed will have its priority lowered. For an example of execution with priorities, see Bach .[3]

1.3.3. The Round Robin with priorities

A combination of the above two techniques is generally used. Each priority level has a corresponding turnstile. The scheduler chooses the highest priority non-empty turnstile and executes it.
To ensure that all processes can run, it is necessary to periodically adjust the different priorities.

1.3.4 FCFS

CFS (First Come, First Serve) is a non-preemptive CPU scheduling algorithm where processes are executed in the order they arrive in the ready queue. The first process in the queue is allocated CPU time, followed by the second, and so on. Once a process starts, it runs to completion or until it performs I/O, blocking other processes. This can lead to longer waiting times for subsequent processes, especially if a long process is scheduled first.

[3] op. cit. pp. 267-270.

1.3.5 Scheduling threads

Threads are subject to scheduling. In Windows NT, threads have 32 priority levels, which are either fixed or dynamic. The highest priority is always the one that executes. In the case of dynamic priority, the values of this priority vary between two limits. It increases, for example, during an I/O wait. The priority of threads can be changed in Windows NT using the SetThreadPriority() function.
With Java, priority levels vary between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY. The normal priority is Thread.NORM_PRIORITY. As with NT, the highest priority is always the one that is executed. Some interpreters use a tourniquet that manages threads of the same priority with no time limit for the thread that is executing. Other interpreters limit the execution of a thread to a quantum of time, then move on to another thread of the same priority. The execution model can be quite different depending on the machine, for example between the IBM 580 and the Sun E3000.

1.4 What is Context Switching?

Context switching is a fundamental process in multitasking operating systems. It occurs when the CPU switches from executing one process to another. During this switch, the operating system saves the current process's state—such as the program counter, registers, and other relevant data—into a Process Control Block (PCB). The PCB stores all necessary information about the process, including its state and memory allocation. After saving the state, the system loads the next process's information and resumes its execution. This enables efficient multitasking by allowing multiple processes to share the CPU.

1.5 What are the goals of CPU scheduling?

The goals of CPU scheduling are to optimize system performance based on various factors. Maximizing CPU utilization ensures that the CPU is fully used, minimizing idle time. Fair allocation guarantees that no process monopolizes the CPU, and each process receives its fair share of time. Maximizing throughput aims to complete the most processes in a given time period. Minimizing turnaround time reduces the total time a process takes to complete; while minimizing waiting time decreases the time a process spends in the ready queue. Lastly, minimizing response time is crucial for interactive systems, ensuring quick feedback to users. Different scheduling algorithms prioritize these goals based on system needs.