University of Tlemcen

Computer Science Department

1st Year Engineer

"Introduction to the operating systems 2"

Sessions 3 and 4: Process management

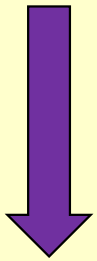
Academic year: 2023-2024

Outline

- Notion of process ?
- Process creation.
- Termination of a process.
- Relation between processes

Introduction

- The first O.S. allowed only one program to be run at a time.
  - *Such a program had full control of the system and access to all system resources.*
- Today's operating systems allow several programs to be loaded into memory and run at the same time.



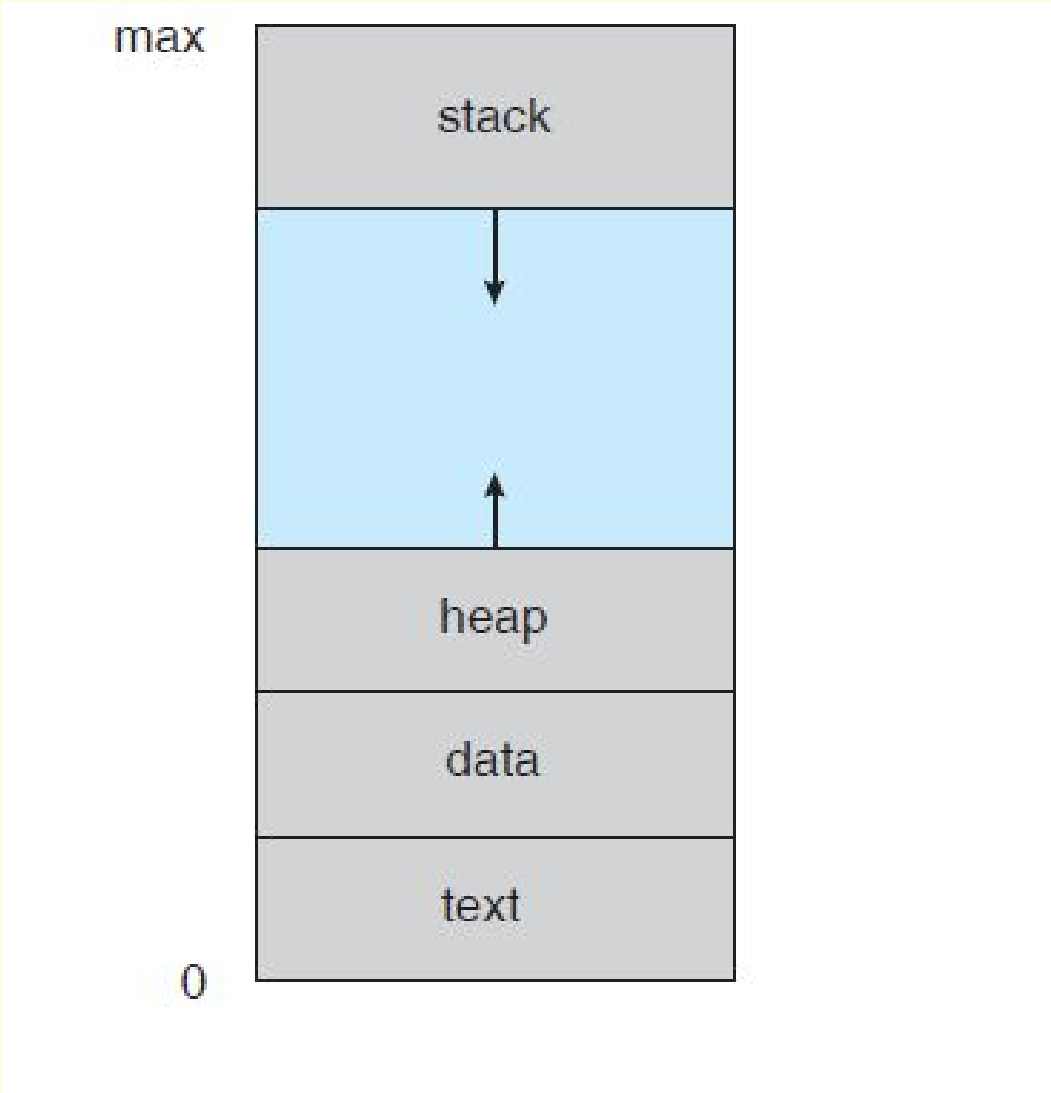
This development has necessitated stricter control and more rigorous partitioning of the different programs.

- This need led to the creation of the notion of process, which is a " program in execution ".
  - *A process is the fundamental unit in a modern time-sharing E.S..*
- A system is composed of a set of processes:
  - *O.S. processes that execute the O.S. code*
  - *User processes that execute the user's code.*
- All these processes can run at the same time.
  - *By switching the processor between processes, the operating system can make the computer more productive.*

Process Concept

- A process is often defined as a running program.
  - *A process is more than program code (text section)*
- A process includes a representation of the program's current activity:
  - *The contents of the processor registers.*
  - *The program counter (PC) (a register that contains the memory address of the next instruction).*
- A process also includes :
  - *an execution stack which contains temporary data (such as the function, parameters, return addresses, and local variables),*
  - *and a data section, which contains global variables.*
- A process may also include a heap, which is memory that is dynamically allocated during execution.

- Process representation in memory



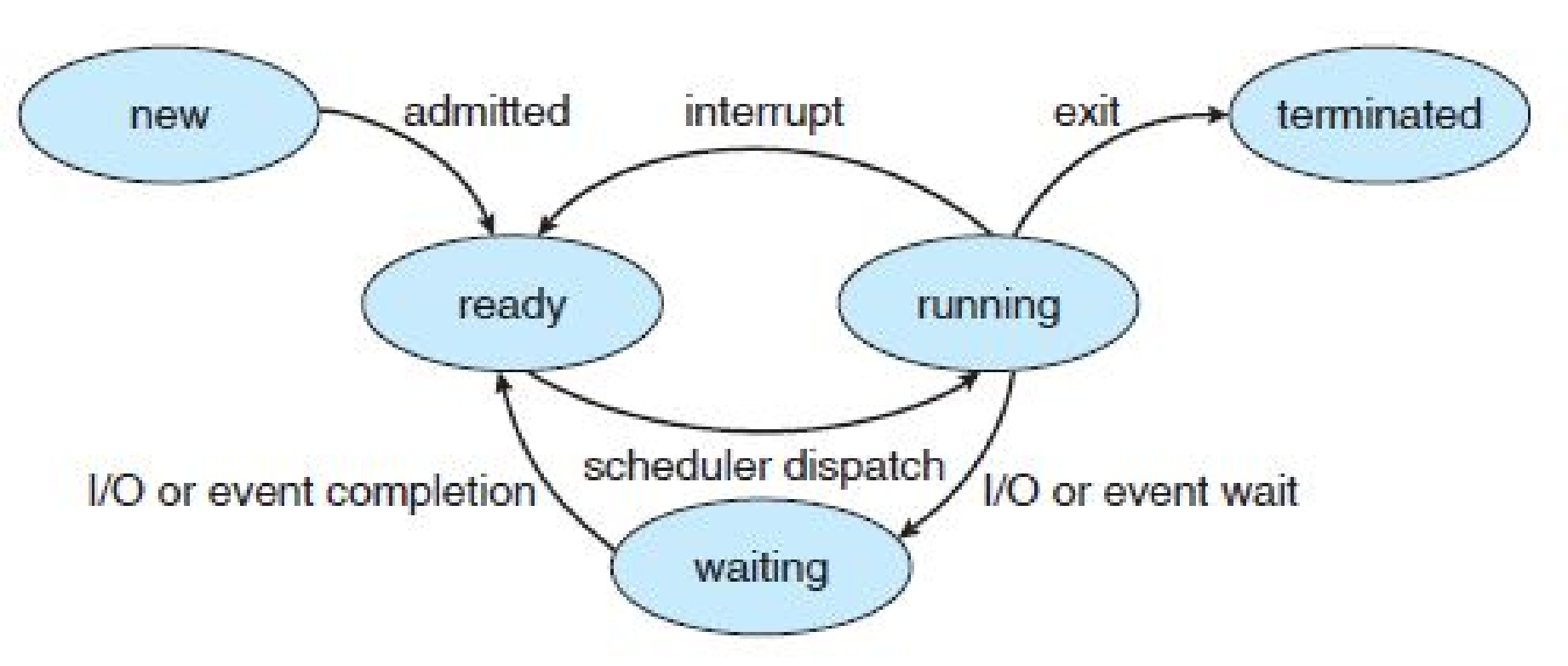
The concept of process: Program vs. Process

- The program itself is not a process,
  - *A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file),*
  - *whereas a process is an active entity, with a program count specifying the next instruction to be executed and a set of associated resources.*
  - *A program becomes a process when an executable file is loaded into memory.*

- Although two processes may be associated with the same program, they are nevertheless considered as two separate execution sequences.
  - *For example, several users may run different copies of email software.*
  - *Each of these is a separate process, and although the text sections are equivalent, the data section, stack and heap are different.*

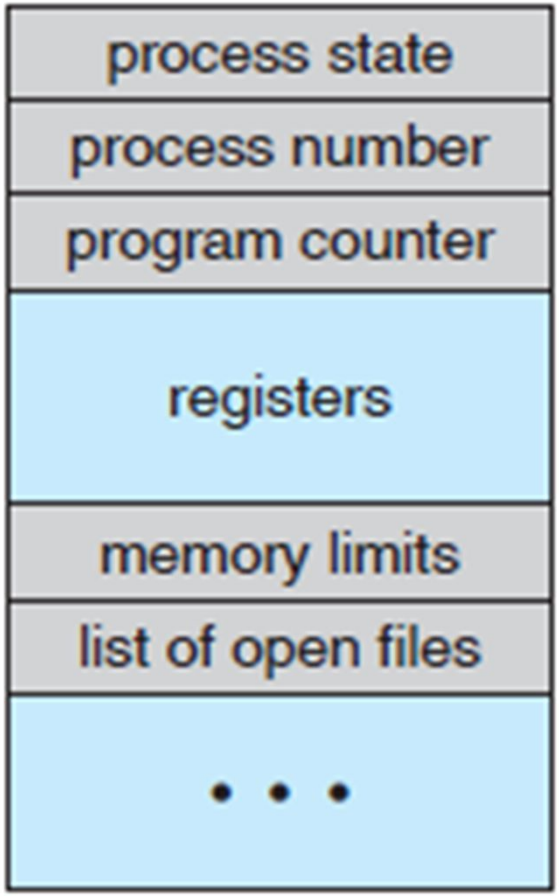
Process states

- During its execution, a process changes state.
- Each process can be in one of the following states:
  - ***New**: The process is being created*
  - ***Running**: Instructions are being executed.*
  - ***Waiting**: The process is waiting for an event to occur (for example an I/O, completion or reception of a signal).*
  - ***Ready**: The process is waiting to be allocated to a processor.*
  - ***Terminated**: The process has finished executing.*


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process Control Block (PCB)

- *Each process is represented in the operating system by a Process Control Block (PCB), also known as a task-control block.*
- *The operating system maintains in a table called the "process table".*
  - *Information on all the processes created (one entry per process: PCB).*



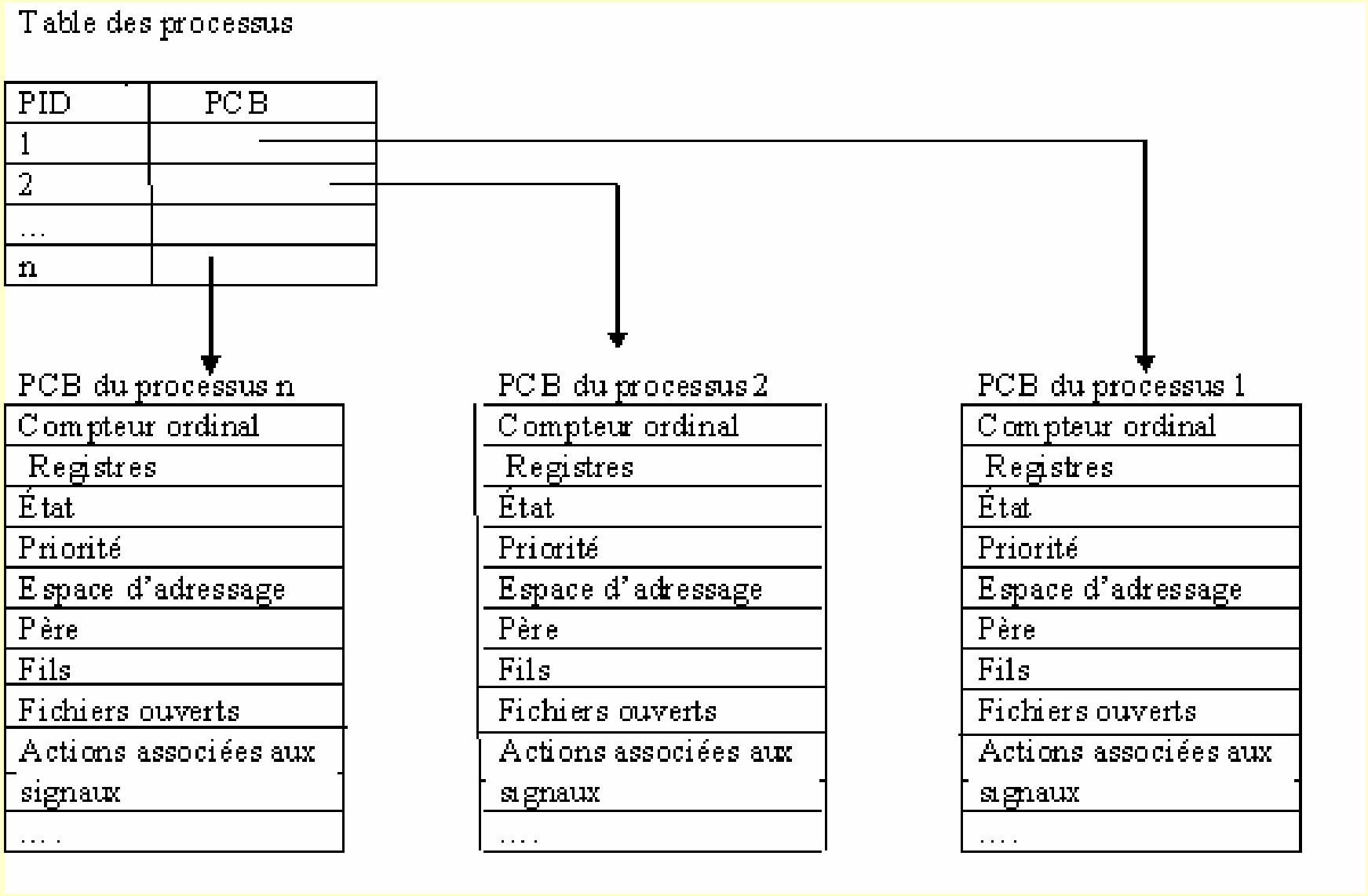


Table des processus
PID
PCB
1
2
...
n
PCB du processus n
Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....
PCB du processus 2
Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....
PCB du processus 1
Compteur ordinal
Registres
État
Priorité
Espace d'adressage
Père
Fils
Fichiers ouverts
Actions associées aux signaux
....

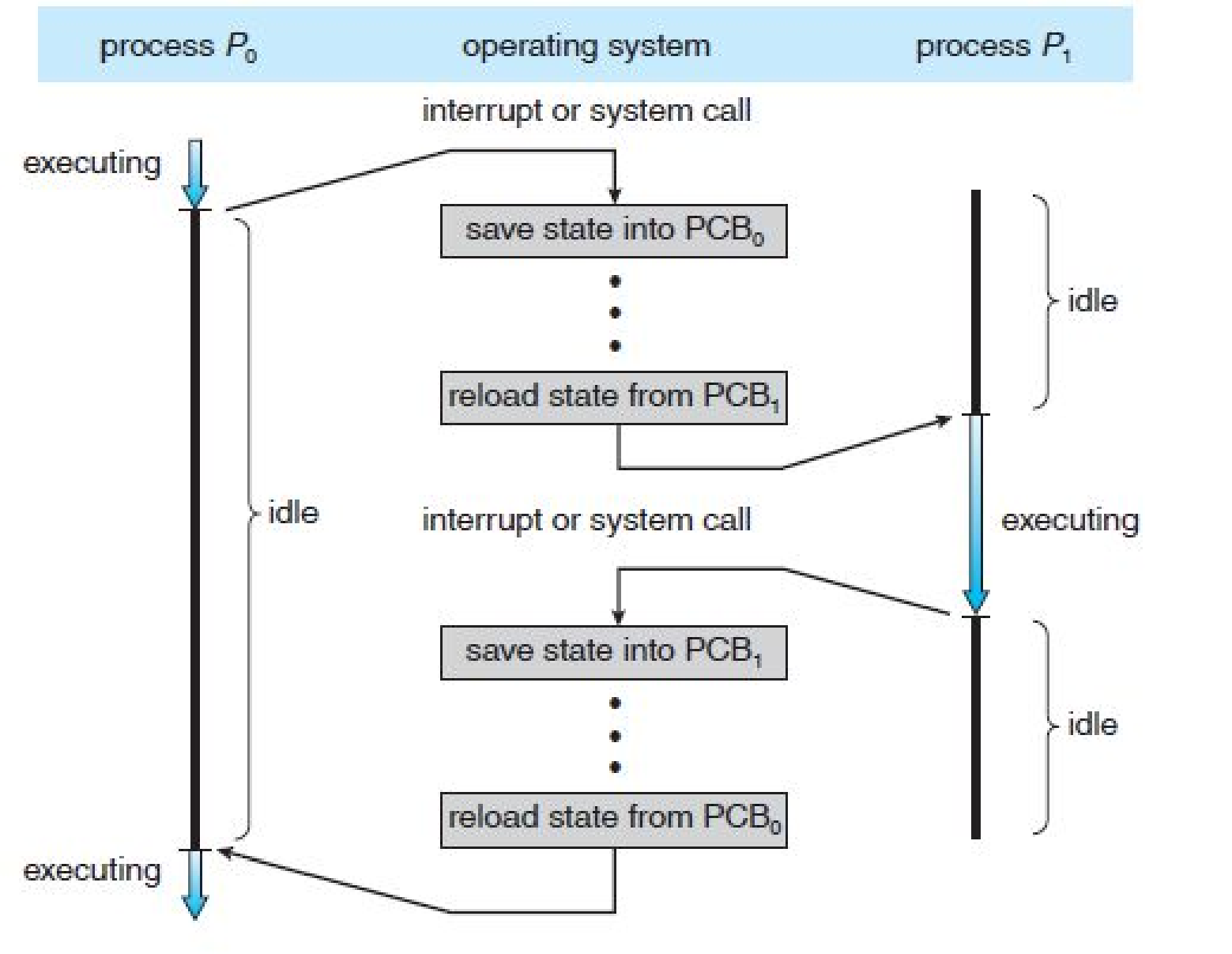
PCB : Cont'

- ***Process status**: The status can be New, Ready, Running, Waiting, etc.*
- ***Program counter**: The counter indicates the address of the next instruction to be executed for the process.*
- ***CPU registers**: Registers vary in number and type, depending on the computer architecture.*
- *They include accumulators, index registers, stack pointers and general purpose registers.*
- *These registers must be saved if an interrupt occurs, to allow the process to continue correctly afterwards.*

Cont'

- *CPU scheduling information: This includes process priority, pointers to scheduling queues, and other scheduling parameters.*
- *Memory management information: This can include page tables, segment tables, etc.*
- *I/O status information: Includes the list of I/O devices allocated to the process, the list of open files,…*

PCB


process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
reload state from PCB_1
idle
idle
interrupt or system call
executing
save state into PCB_1
reload state from PCB_0
idle
executing

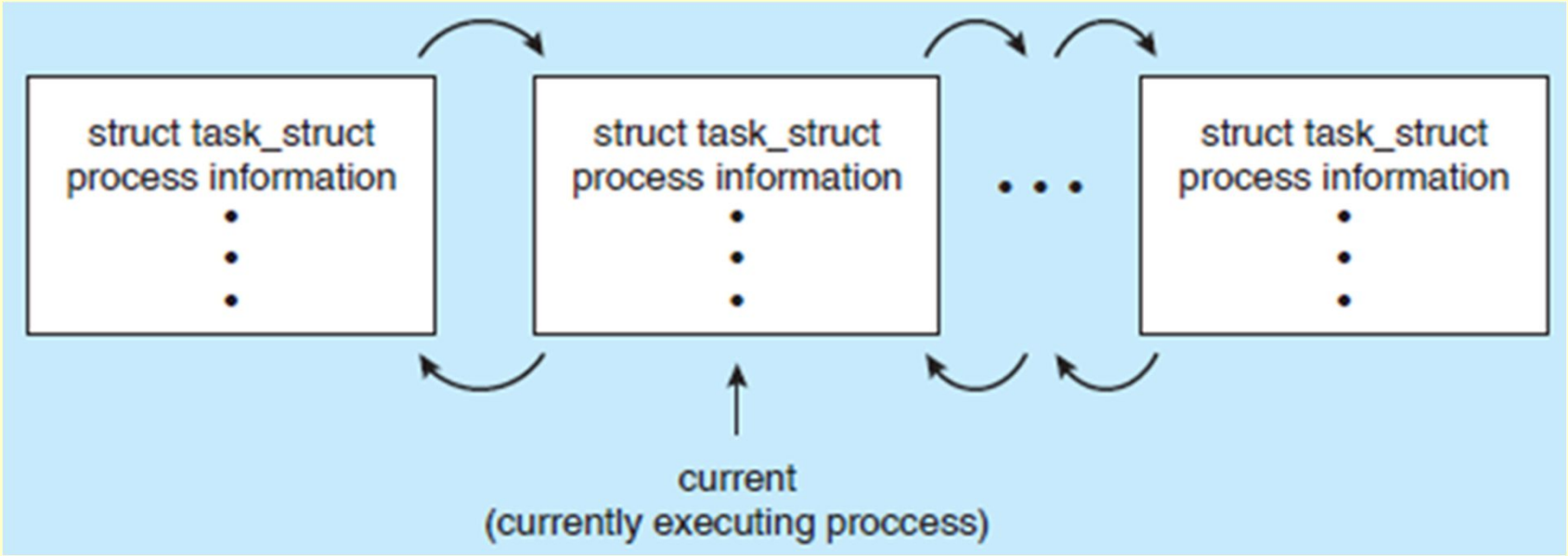
- In the Linux operating system, the PCB is represented by the C task_struct structure.
- This structure contains all the information needed to represent a process:
  - *the state of the process,*
  - *its scheduling,*
  - *memory management,*
  - *the list of open files,*
  - *and pointers to the parent process and one of its children.*
    - *The parent of a process is the process that created it,*
    - *its children are all the processes it creates.*

PCB

- *Some fields in the structure "task_struct" (/linux/sched.h):*
  - *pid_t pid; /* process identifier */*
  - *long state; /* state of the process */*
  - *unsigned int time slice /* scheduling information */*
  - *struct task struct *parent; /*this process's parent */*
  - *struct list head children; /*this process's children */*
  - *struct files struct *files; /*list of open files */*
  - *struct mm struct *mm; /*address space of this process*/*

PCB

- *In the Linux kernel, all active processes are represented using a double-chained list of task_struct.*
- *The kernel maintains a current pointer to the process currently running on the system.*



PCB

- *Let's suppose that the system wants to change the state of the process currently running to the new value new_state.*
- *With current pointing to the currently running process, the state is changed as follows:*

current->state = new_state;

Process Scheduling

- In a multi-user time-sharing system, several processes may be present in main memory waiting to be executed.
- If several processes are ready, the operating system must manage the allocation of the processor to the different processes to be executed.
- The scheduler performs this task.
- A scheduler faces two problems:
  - *choosing which process to run, and*
  - *the time taken to allocate the processor to the chosen process.*

Process Scheduling: Preemptive operating system

- A multitasking operating system is preemptive
  - when it can stop (requisition) any application at any time to hand over to the next one.
- In preemptive operating systems, you can run several applications at once and switch between them, or even run one application while another is working.
- The kernel always retains control
  - reserves the right to close applications that are monopolizing system resources.

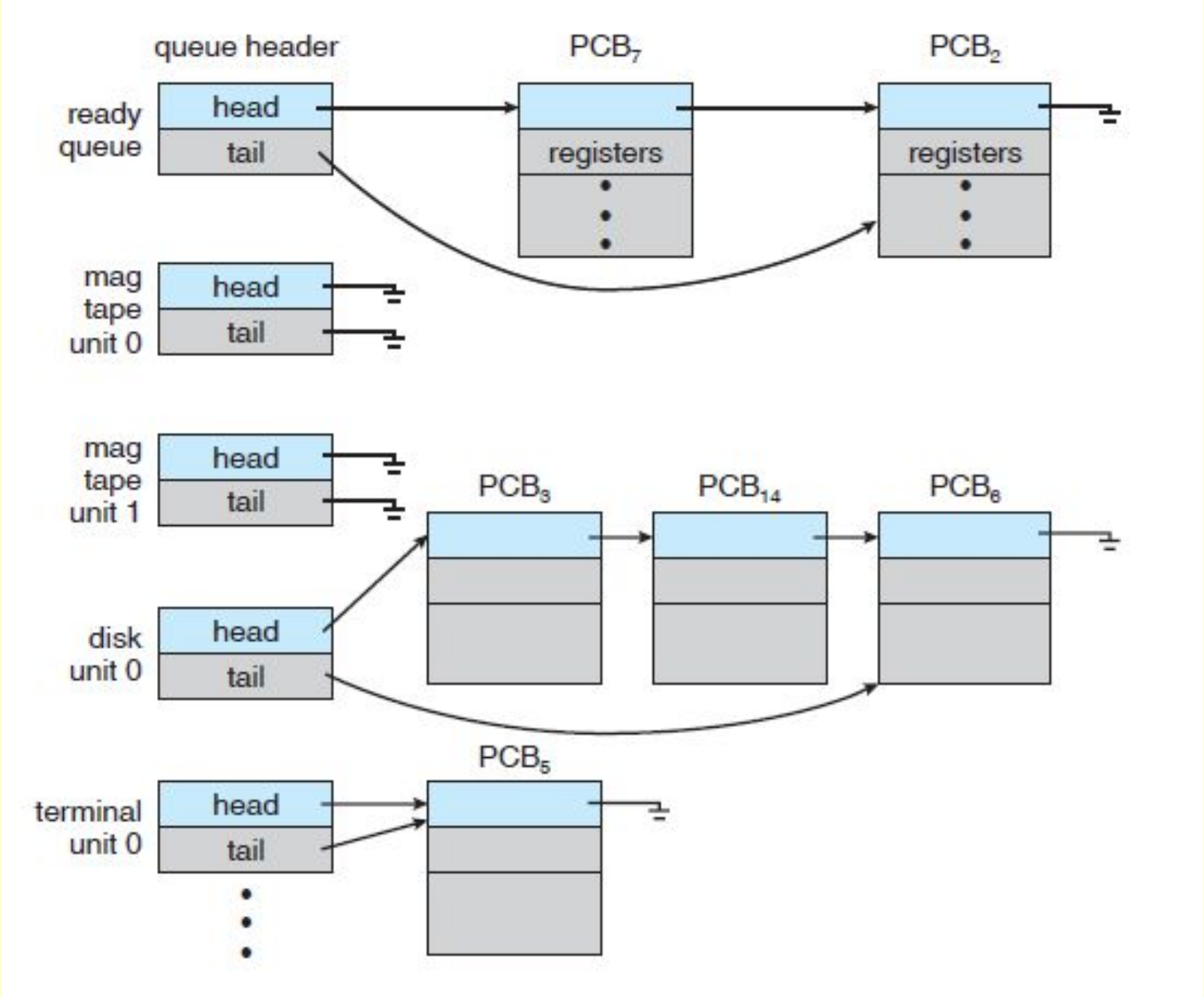
Process Scheduling: Cooperative O.S

- A multitasking operating system is cooperative
  - When it allows several applications to run
  - and occupy memory slots, leaving it up to these applications to manage this occupation,
  - at the risk of blocking the whole system.

Process Scheduling : Scheduling Queues

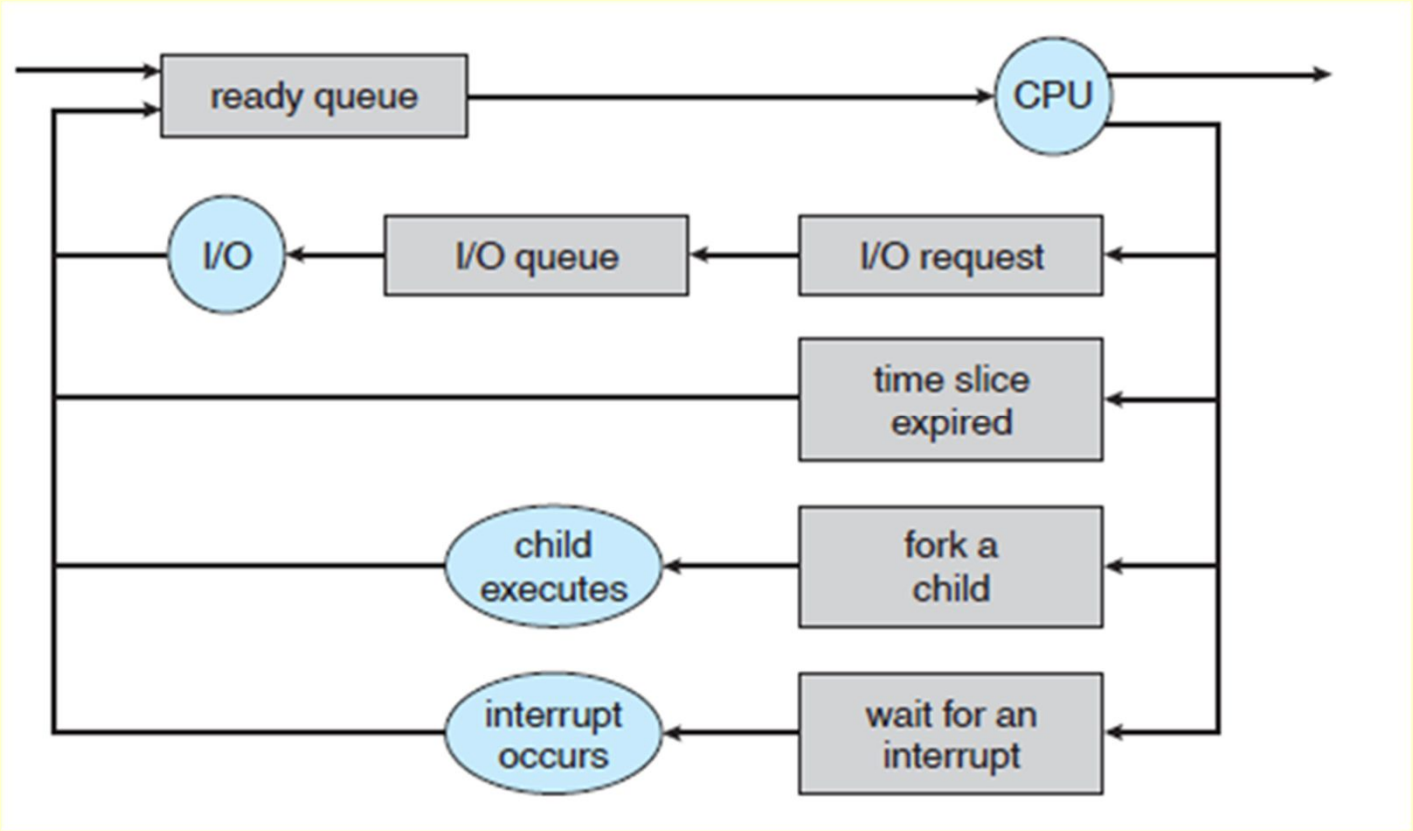
- As soon as processes are taken over by the O.S, they are placed in a queue (***job_queue***), which contains all the processes in the O.S.
- Processes that are stored in main memory and are ready and waiting to be executed are kept in a list called ***ready_queue***.
  - *This queue is usually implemented as a chained list.*
  - *The ready_queue contains a header with two pointers:*
    - *the first to the first PCB,*
    - *the second to the last PCB in the list.*
  - *Each PCB has a pointer to the next PCB in the ready_queue*

- The system also includes other queues.
  - *When a process is allocated the CPU, it executes for a certain amount of time and then exits, pauses or waits for a particular event to occur, such as the completion of an I/O request.*
  - *Suppose the process makes an I/O request to a shared device, such as a disk.*
    - *As there are many processes in the system, the disk may be busy with another process's I/O request.*
    - *The process may therefore have to wait for the disk.*
    - *The list of processes waiting for an I/O device is called a* ***device_queue****.*
    - *Each device has its own device_queue.*


queue header
PCB7
PCB2
ready queue
head
tail
registers
registers
mag tape unit 0
head
tail
mag tape unit 1
head
tail
PCB3
PCB14
PCB6
disk unit 0
head
tail
PCB5
terminal unit 0
head
tail

Cont'

- A common representation of process scheduling is a queue diagram.



- There are two types of queues:
  - *ready_queue and a set of device_queues.*

Cont'

- A new process is first placed in the ready_queue.
  - *It waits in this queue until it is selected for execution, or is dispatched.*
- Once the process has been allocated the CPU, and is running, one of these events could occur:
  - *The process could issue an I/O request and then be placed in an I/O queue.*
  - *The process could create a new sub-process and wait for the sub-process to finish.*
  - *The process could be interrupted, forcibly removed from the CPU, and put on hold in a device_queue.*

Process Scheduling: Schedulers

- A process migrates between different scheduling queues throughout its lifetime.
  - *The O.S must select the processes in these queues for scheduling purposes.*
- The selection process is carried out by the scheduler.

Cont'

- The long-term scheduler selects the programs to be admitted to the system for execution.
- Admitted programs become ***ready*** processes.
- Admission depends on the capacity of the system (degree of multi-programming) and the level of performance required.

Cont'

- Generally the number of processes submitted is greater than the immediate processing capacity of the system.
  - *These processes are queued on a mass storage device (usually a disk), where they are held for later execution.*
  - *The **job scheduler** selects processes from this pool and loads them into memory for later execution.*
  - *The **CPU scheduler** selects which processes are ready to run and allocates the CPU to one of them.*

Cont'

- The task of the short-term scheduler is to manage ready processes.
  - Based on a certain policy, it selects the next process to be executed.
  - It also performs context switching for processes.
  - It can implement preemptive, non-preemptive or cooperative scheduling.
- The scheduler is activated by an event
  - timer interruption,
  - peripheral interrupt,
  - system call or signal.

- The main distinction between these two schedulers is the frequency of execution.
  - *The short-term scheduler must frequently select a new process for the CPU.*
  - *A process may run for only a few milliseconds before an I/O request.*
- The short-term scheduler often runs at least once every 100 milliseconds.
  - *Because of the short period between executions, the short-term scheduler must be fast.*
  - *If it takes 10 milliseconds to decide to run a process for 100 milliseconds, 10 / (100 + 10) = 9% of the CPU is used (lost) simply to schedule the work.*

- The long-term scheduler runs much less frequently
  - *minutes may separate the creation of two processes*
- The long-term scheduler controls the degree of multi-programming (the number of processes in memory).
- If the degree of multiprogramming is stable
  - *then the average process creation rate must be equal to the average of the departure rates of the processes that leave the system.*
- The long-term scheduler is invoked when a process leaves the system.

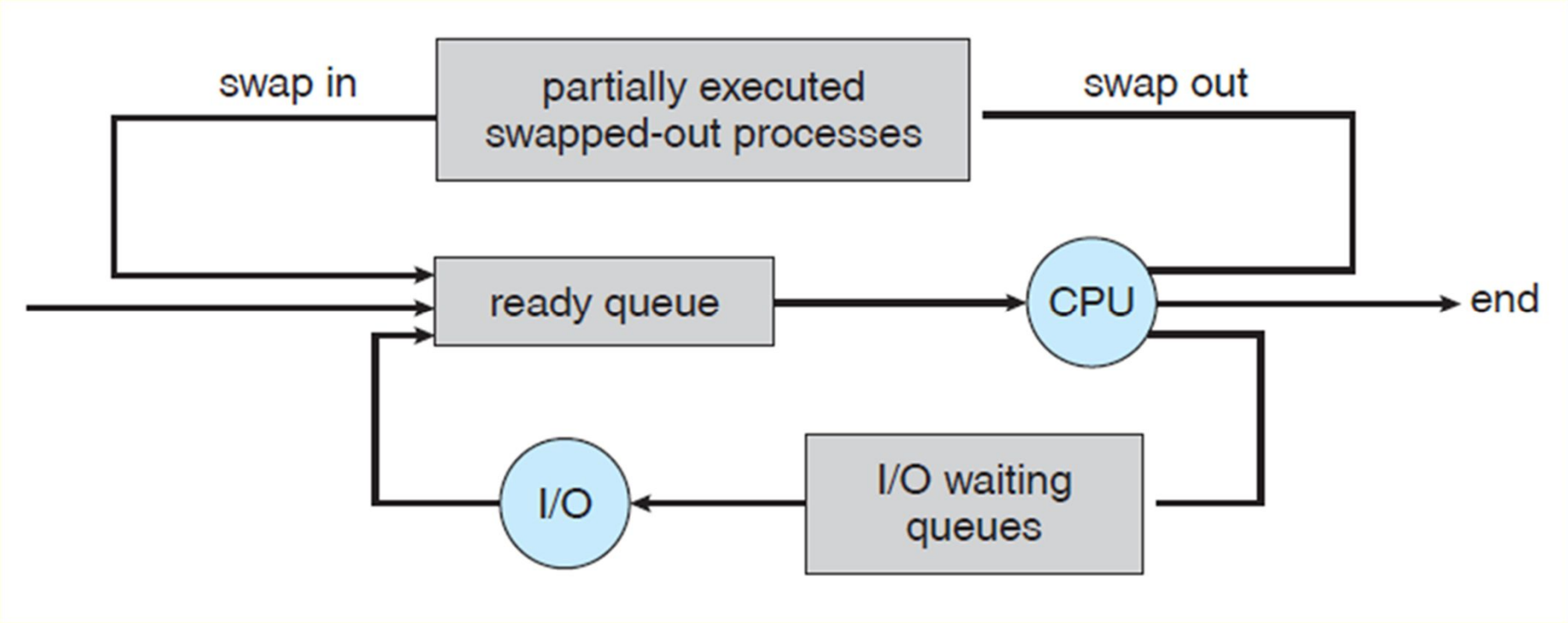
- The goals of a scheduler are:
  - *Ensure that each process waiting to run receives its share of processor time.*
  - *Minimize response time.*
  - *Use the processor at 100%.*
  - *Balanced use of resources.*
  - *Take priorities into account.*

- Most processes can be described as :
  - *I/O-bound processes or CPU-bound processes.*
- A process is CPU-bound
  - *Rarely generates I/O,*
  - *Spends more time doing calculations on the CPU,*
  - *Its progress is limited by the CPU:*
    - *if the CPU was faster, it would have completed its execution more quickly.*
- A process is I/O-bound
  - *Generates a large number of I/Os (files/disk accesses/...),*
  - *Its progress is limited by I/O.*

- It is important that the long-term scheduler makes a balanced selection between the two types of process.
- If all the processes selected were I/O-bound
  - *the ready-queue would almost always be empty,*
  - *the short-term scheduler will have very little to do.*
- If all the processes selected are CPU-bound
  - *the I/O queue will almost always be empty,*
  - *devices (disks/...) will be under-utilized*
  - *once again, the system will be unbalanced.*
- The system with the best performance should have a combination of CPU-bound and I/O-bound processes.

- On some systems, the involvement of the long-term scheduler may be absent or minimal.
  - *For example, time-sharing systems such as UNIX and Microsoft Windows often do not have a long-term scheduler,*
  - *They simply load each new process into the short-term scheduler's memory,*
  - *The stability of these systems depends on either a physical limit (such as the number of terminals available) or the self-adjustment of the human user (who quits their sessions because of a drop in system performance).*
- These systems introduce an intermediate level of scheduler: medium-term schedulers.

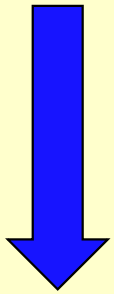
- The main idea behind a medium-term scheduler is that, sometimes,
  - *it may be advantageous to remove processes from memory (and from contention on the CPU) and thus reduce the degree of multi-programming.*
  - *Later, the process can be reintroduced into memory, and its execution can continue where it left off.*
- This scheduling scheme is called ***swapping***.


swap in
partially executed swapped-out processes
swap out
ready queue
CPU
end
I/O
I/O waiting queues

Process Scheduling : Context Switch

- When an interrupt occurs
  - *the system saves the current context of the running process on the processor so that it can restore this context when it is processed,*
    - *The context is represented by the process PCB*
- In generic terms, the O.S saves the state of the CPU (whether in user mode or kernel mode) and then restores the state to resume execution.

- Switching the CPU from one process to another requires
  - *executing a state backup of the current process,*
  - *and restoring the state of the next process.*



This task is known as a ***context switch***.

- When a context switch occurs
  - *the kernel saves the context of the old process in its PCB,*
  - *and loads the saved context of the new process to run.*
- The context switching time is considered to be an overhead.
  - *The O.S does not execute any tasks during this time,*
  - *Its speed varies from machine to machine, depending on the speed of the memory, the number of registers that need to be copied, and the existence of special instructions (such as a single instruction to load or store all the registers).*
  - *Typical speeds are a few milliseconds.*

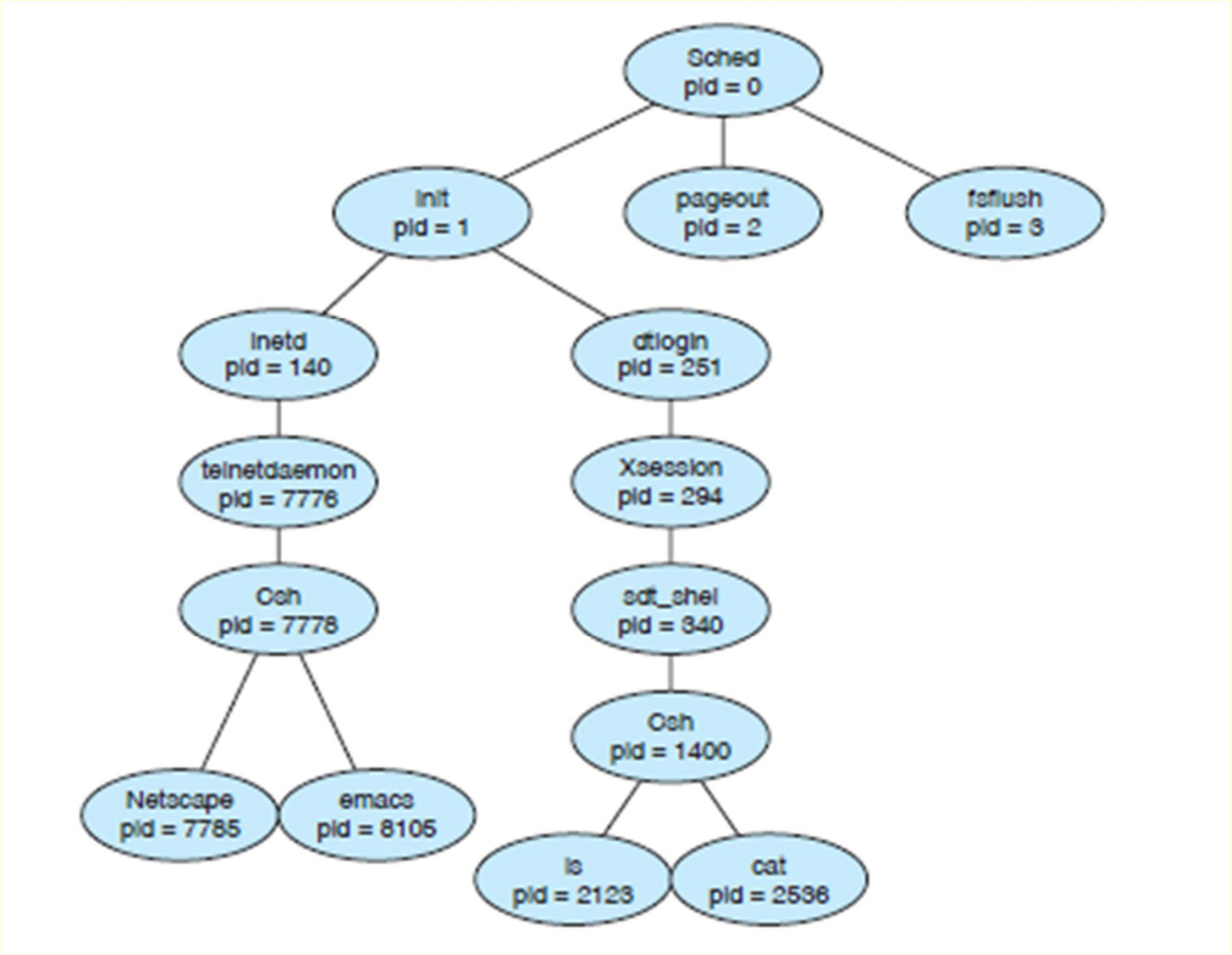
Process operations

- Processes in most systems can run concurrently, and they can be dynamically created and removed.
- These systems must therefore provide a mechanism for creating and terminating processes.

Création de processus

- A process (***parent process***) can create several new processes (***child processes***) via a system call during execution.
- Each of these new processes can in turn create other processes, forming a process tree.
- Most operating systems identify a process via a PID, which is typically an integer.

Process tree in Solaris



Cont'

- On Solaris, the process at the top of the tree is the process ***sched***, with a pid of 0.
- The sched process creates several child processes, including the ***pageout*** and ***fsflush*** processes.
  - *These processes are responsible for memory and file system management.*
- The process sched also creates the process ***init***, which serves as the root parent process for all user processes.
  - ***init*** *has two child processes: netd and dtlogin.*
    - ***inetd*** *is responsible for network services such as telnet and ftp,*
    - ***dtlogin*** *is the process that provides the user with a login screen.*
    - *When a user logs in, dtlogin creates an X-Windows session (Xsession), which in turn creates the process* ***sdt_shel****.*

Process creation

- In general, a process needs certain resources (CPU time, memory, files, I/O devices) to perform its task.
- When a process creates a child process
  - *the child process may be able to obtain its resources directly from the operating system,*
  - *or it may be limited to a subset of the resources of the parent process.*
    - *The parent may have to partition its resources between its children,*
    - *or it may be able to share certain resources (such as memory or files) between several of its children.*

- When a process creates a new process, there are two execution options:
  - *The parent continues to run at the same time as its children.*
  - *The parent waits until all or some of its children have terminated.*
- There are also two options for managing the address space of the new process:
  - *The child process is a copy of the parent process (it has the same program and data as the parent process).*
  - *The child process has a new program loaded.*

- To illustrate these differences, we will first look at the UNIX operating system.
- A new process is created by fork() of the calling process.
  - *The new process includes a copy of the address space of the original process.*
    - *This mechanism allows the parent process to communicate easily with its child process.*
  - *Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference:*
    - *the return code from the fork() is zero for the new(child) process, while the (non-zero) PID of the child process is returned to the parent process.*

- As a general rule, the ***exec()*** system call is used after a ***fork()*** system call by one of the two processes in order to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and begins execution.
- In this way, the two processes are able to communicate and then separate.
- The parent can then create several children, or, if it has nothing else to do while the child process is executing, it can issue a ***wait()*** system call to put itself out of the ready_queue until the child process terminates.

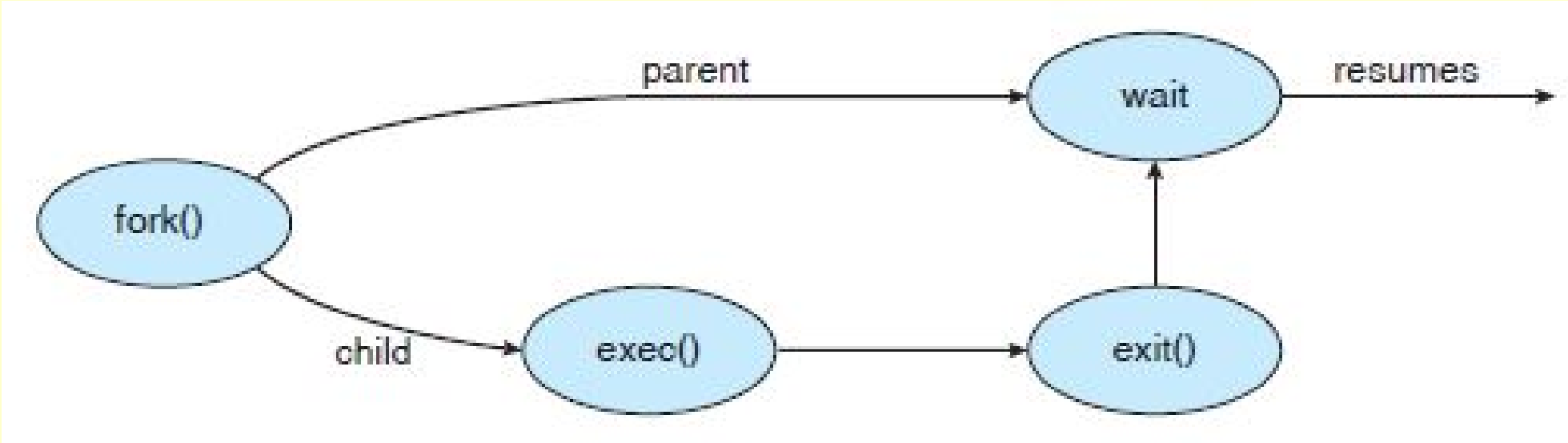
Example (2)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include  <sys/wait.h>
int main()
    {
      pid_t pid;
      pid = fork();   /* fork a child process */
      if (pid < 0) {
          /* error occurred */
          fprintf(stderr, "Fork Failed");
          return 1;
        else if (pid == 0) {  /* child process */
          printf("Child Process says Hello !\n");
          execlp("/bin/ls","ls",NULL);
        }
        else {  /* parent process */
           /* parent will wait for the child to complete */
           printf("Parent Process says Hello !\n");
           wait(NULL);
           printf("Child Complete");
        }
    return 0;
}
```

The function **execlp()** can be used to launch an application which will be searched in the directories specified in the PATH environment variable, by supplying arguments in the form of a variable list terminated by a NULL pointer. The prototype of execlp() is as follows:

*int execlp (const char * application, const char * arg, ...);*

Process creation



Identification by PID

- The first process in the system, ***init***, is created directly by the kernel at boot time.
- The only way to create a new process is to call the ***fork()*** system call, which will duplicate the calling process.
- When this system call returns, two identical processes will continue to execute the code following fork().
  - *The essential difference between these two processes is an identification number.*
  - *This distinguishes between the original process, traditionally called the parent process, and the new copy, the child process.*
  - *The fork() system call is declared in <unistd.h>, as follows*

pid_t fork(void);

- As the two processes can be distinguished by their PID, it is possible to execute two different codes when the fork() system call returns.
  - *For example, the child process can ask to be replaced by the code of another executable program on the disk.*
- To find out its own PID, use the ***getpid()*** system call, which takes no arguments and returns a value of type pid_t.
  - *This is, of course, the PID of the calling process. This system call declared in <unistd.h>,*

    pid_t getpid (void);

Importance of creating processes

- In most current applications, the purpose of creating a child process is to allow two independent parts of the program to communicate (using signals, shared memory, etc.).
- The child process can easily access its parent's PID (denoted PPID) using the ***getppid()*** system call, declared in <unistd.h> :

  pid_t getppid (void);

- This routine behaves like *getpid()*, but returns the PID of the parent of the calling process. On the other hand, the parent process can only find out the number of the new process being created when fork() returns.

Example

```
$ ps axj
PPID   PID  PGID   SID  TTY   TPGID  STAT   UID   TIME COMMAND
   0     1     0     0  ?        -1  S        0   0:03 init
   1     2     1     1  ?        -1  SW       0   0:03 (kflushd)
   1     3     1     1  ?        -1  SW<      0   0:00 (kswapd)
   1     4     1     1  ?        -1  SW       0   0:00 (nfsiod)
   [...]
   1   296   296   296  6       296  SW       0   0:00 (mingetty)
 297   301   301   297  ?        -1  S        0  45:56 usr/X11R6/bin/X
 297 25884 25884   297  ?        -1  S        0   0:00 (xdm)
```

- When a process is created by fork(), it has a copy of its parent's data, as well as its parent's environment and a certain number of other elements (file descriptor table, etc.).
  - *This is known as inheritance from the parent.*
- Under Linux, the fork() system call is very economical because it uses a "copy on write" method.
  - *This means that all the data that needs to be duplicated for each process (file descriptors, allocated memory, etc.) will not be copied immediately. As long as neither process has modified any information in these memory pages, there is only one copy on the system.*
  - *However, as soon as one of the processes writes to the area in question, the kernel ensures that the data is actually duplicated.*

- In the event of an error, fork() returns the value -1, and the global variable ***errno*** contains the error code, defined in <errno.h>.
- This error code can be :
  - *Either ENOMEM , which indicates that the kernel no longer has enough memory available to create a new process.*
  - *Or EAGAIN, which indicates that the system has no more free space in its process table but that there will probably be some soon.*
    - *A process is therefore authorized to repeat its duplication request once it has obtained an EAGAIN error code.*

Example (3)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include  <sys/wait.h>
int main (void)
{
    pid_t pid_fils;


        do {
        pid_fils = fork ( ) ;
    }while ((pid_fils == -1) && (errno == EAGAIN));


    if (pid_fils == -1) {
        fprintf(stderr,"fork( ) impossible, errno= %d\n", errno);
        return (1);
    }

    if (pid_fils == 0) {
            fprintf (stdout, "Fils : PID=%d, PPID=%d\n",getpid( ),getppid( ))
            return (0);

    } else {
        fprintf (stdout, "Pére :PID=%d,PPID=%d,PID FILS= %d\n",getpid( )
    ,getppid( ),pid_fils);
            wait(NULL);
        return(0);
        }
    return 0;
    }
```

- Note that we have introduced a ***wait(NULL)*** system call at the end of the parent code.
  - *This is used to wait for the child to finish executing.*
- If we hadn't used this system call, the parent process could have terminated before the child, giving control back to the shell, which would then have displayed its prompt symbol ($) before the child had printed its information.

Identification by PID (no wait)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include  <sys/wait.h>
int main (void)
{
    pid_t pid_fils;
    int i;

        do {
        pid_fils = fork ( ) ;
    }while ((pid_fils == -1) && (errno == EAGAIN));


    if (pid_fils == -1) {
        fprintf(stderr,"fork( ) impossible, errno= %d\n", errno);
        return (1);
    }

    if (pid_fils == 0) {

        for (i=0;i<10;i++)
        fprintf (stdout, "Fils : PID=%d, PPID=%d\n",getpid( ),getppid( ))
    return (0);

    } else {
        fprintf (stdout, "SANS WAIT, Pére :PID=%d,PPID=%d,PID FILS=
    %d\n",getpid( ) ,getppid( ),pid_fils);
        return(0);
        }
    return 0;
    }
```

Identification of the process user

- Unlike single-user systems (Dos, Windows 95/98...), a Unix system is particularly focused on user identification.
- Any activity undertaken by a user is subject to strict controls on the permissions assigned to them.
- To achieve this, each process runs under a specific identity.
  - *In most cases, this is the identity of the user who invoked the process and is defined by a numerical value: the UID (User IDentifier).*

Termination of a process

- A process can end normally or abnormally.
- In the first case
  - *the application is abandoned at the user's request, or the task to be performed is finished.*
- In the second case
  - *a malfunction is discovered which is so serious that it does not allow the program to continue its work.*

Orphan and Zombie

Orphan processes :

- *If a parent dies before its child, the latter becomes an orphan.*

Zombie process :

- *If a child terminates while still having a PID, it becomes a zombie process (e.g. without its parent having read its exit code (via wait)).*

- A child process can become orphaned if its parent terminates before it does, in which case the kernel arranges for it to be "adopted" by a system process (init), so the child process can pass on its termination status to it.
- A process is said to be zombie if it has terminated, but still has a PID and is therefore still visible in the process table.
- When a process terminates, the system deallocates the resources it still has but does not destroy its control block.
  - *The system then sets the process status to TASK_ZOMBIE (usually represented by a Z in the "status" column when listing processes with the ps command).*

Normal termination of a process

- The SIGCHLD signal is then sent to the parent process of the terminated process to inform it of this change.
  - *As soon as the parent process has obtained the end code of the terminated process using the wait or waitpid system calls, the terminated process is permanently removed from the process table.*
- SIGCHLD: is a signal used to wake up a process whose child has just died.

- The _exit( ) system call performs the following tasks:
  - *It closes the file descriptors.*
  - *The child processes are adopted by process 1 (init), which will read their return code as soon as they finish, to prevent them remaining in the zombie state for an extended period of time.*
  - *The parent process receives a SIGCHLD*

- The process then becomes a zombie, i.e. it waits for its parent process to read its return code.
  - *If the parent process explicitly ignores SIGCHLD, the kernel automatically reads it.*
- If the parent process has already terminated, ***init*** temporarily adopts the zombie status, just long enough to read its return code.
  - *Once this has been done, the process is removed from the list of tasks on the system.*

Abnormal termination of a process

- A program can also terminate abnormally.
- This is the case, for example, when a process executes an illegal instruction, or tries to access the contents of an incorrectly initialized pointer.
- These actions trigger a signal which, by default, stops the process by creating a core memory image file.

- A 'clean' way of interrupting a program abnormally (for example when a bug is discovered) is to invoke the abort() function.

  void abort(void)

- This immediately sends the signal SIGABRT to the process, unblocking it if necessary and restoring the default behavior if the signal is ignored.
- The problem with the abort() function or stops due to signals is that it is difficult to determine afterwards where in the program the malfunction occurred.

- It is possible to consult the core file (provided you have included the debugging information when compiling with the -g option in gcc), but this is sometimes an arduous task.
- Another way of automatically detecting bugs is to systematically use the ***assert()*** function in critical parts of the program.
  - ***assert()*** *evaluates the expression passed as an argument.*
  - *If the expression is true, it does nothing. On the other hand, if it is false, assert() stops the program after writing a message to the standard error output, indicating the source file concerned, the line of code and the text of the failed assertion.*
  - *It is then very easy to refer back to the point described to find the bug.*

Example (4)

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

void fonction_reussissant (int i);
void fonction_echouant (int i);

    int main (void)
    {
          fonction_reussissant(5);
          fonction_echouant(5);
          return (EXIT_SUCCESS);
    }

    void fonction_reussissant (int i)
    {
          /* Cette fonction nécessite que i soit positif */
          assert (i >= 0);
          fprintf (stdout, "Ok, i est positif \n");
    }

    void fonction_echouant (int i)
    {
          /* Cette fonction nécessite que i soit négatif */
          assert (i <= 0);
          fprintf (stdout, "Ok, i est négatif \n");
    }
```

Wait for the end of a child process.

- You can therefore imagine the importance that can be attached to reading the return code of a process.
  - *This importance is such that a process which terminates automatically goes into a special state, zombie, while waiting for the parent process to read its return code.*
- If the parent process does not read its child's return code, the child can remain in the zombie state indefinitely.

Example (5)

- The child process waits two seconds before terminating, while the parent process regularly displays the status of its child by invoking the command *ps*.

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

int main (void)
{
        pid_t pid;
        char  commande[128];

        if ((pid = fork()) < 0) {
            fprintf(stderr, "echec fork()\n");
            exit(EXIT_FAILURE);
        }

        if (pid == 0) {
            /* processus fils */
            sleep(2);
            fprintf(stdout, "Le processus fils %ld se termine\n",
                    (long) getpid());
            exit(EXIT_SUCCESS);
```

```
        } else {
            /* processus pere */
            snprintf(commande, 128, "ps %ld", (long)pid);
            system(commande);
            sleep(1);
            system(commande);
            sleep(1);
            system(commande);
            sleep(1);
            system(commande);
            sleep(1);
            system(commande);
            sleep(1);
            system(commande);
        }
        return EXIT_SUCCESS;
}
```

```
$ ./exemple_zombie_1
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  S       0:00  ./exemple_zombie_1
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  S       0:00  ./exemple_zombie_1
Le processus fils 949 se termine
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  Z       0:00  [exemple_zombie_<defunct>]
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  Z       0:00  [exemple_zombie_<defunct>]
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  Z       0:00  [exemple_zombie_<defunct>]
   PID   TTY    STAT    TIME  COMMAND
   949   pts/0  Z       0:00  [exemple_zombie_<defunct>]
$ ps 949
   PID   TTY    STAT    TIME  COMMAND
$
```

Wait for the end of a child process

- When the parent process terminates, the command ***ps*** is invoked manually and the zombie child is found to have disappeared.
  - *In this case, the process init (pid=1) adopts the orphaned child process and reads its return code, causing it to disappear.*

Example (6)

- The parent process will terminate after 2 seconds, while the child will continue to display its parent's PID regularly.

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

int main (void)
{
        pid_t pid;


        if ((pid = fork()) < 0) {
             fprintf(stderr, "echec fork()\n");
             exit(EXIT_FAILURE);
        }


        if (pid != 0) {
             /* processus père */
             fprintf(stdout, "Pere : mon PID est %ld\n", (long)getpid());
             sleep(2);
             fprintf(stdout, "Pere : je me termine\n");
             exit(EXIT_SUCCESS);
```

```
} else {
            /* processus fils */
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
            sleep(1);
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
            sleep(1);
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
            sleep(1);
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
            sleep(1);
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
            sleep(1);
            fprintf(stdout, "Fils : mon pere est %ld\n",
    (long)getppid());
        }
        return EXIT_SUCCESS;
}
```

```
$ ./exemple_zombie_2
Père : mon PID est 1006
Fils : mon père est 1006
Fils : mon père est 1006
Père : je me termine
$ Fils : mon père est 1
Fils : mon père est 1
Fils : mon père est 1
```

Wait for the end of a child process

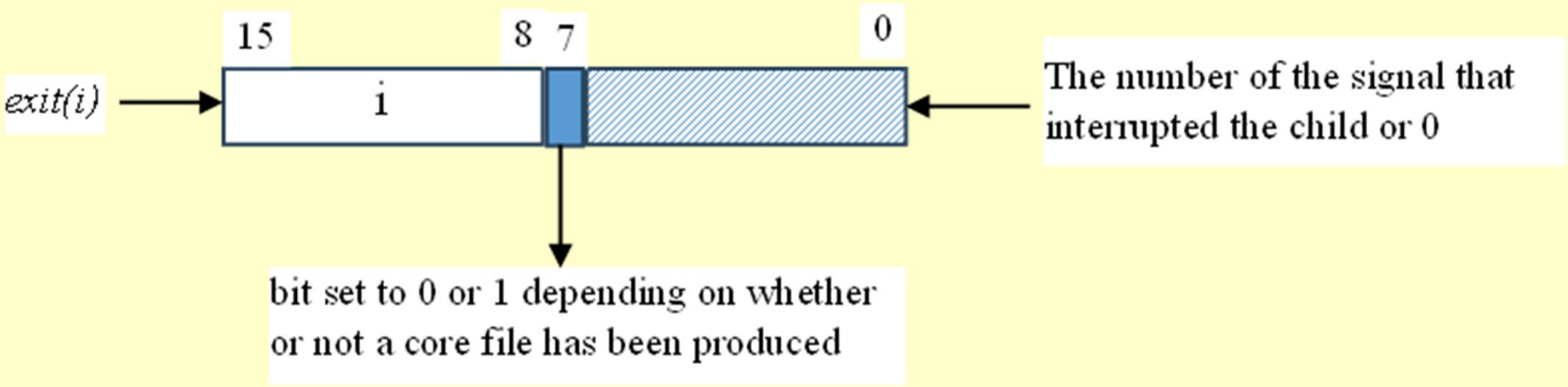
- There are four functions for reading the return code of a child process***: wait( ), waipid(), wait3()*** and *wait4()*.
  - *The first three are actually library functions implemented by invoking wait4( ), which is the only true system call.*
  - *The function **wait( )** is declared in <sys/wait.h>, as follows:*

    pid_t wait (int * status);
  - *When invoked, it blocks the calling process until one of its children terminates. It then returns the PID of the terminated child. If the status pointer is non-NULL, it is filled with a value giving information about the circumstances of the child's death.*
    - *If you are not interested in the circumstances of the termination of the child process, it is perfectly possible to supply a NULL argument.*

Synchronization of parent and child processes

- Synchronization between the parent and child processes is achieved using the following functions:
  - ***exit(i)*** *:Terminates a process, i is a byte (so possible values: 0 to 255) returned in a variable of type int to the parent process.*
  - ***wait(&status)****:Puts the parent process on hold until one of its children processes has finished.*
    - *When a process terminates, the signal SIGCHILD is sent to its parent.*
      - *Receiving this signal switches the parent process from the blocked state to the ready state. The parent process therefore exits the function wait.*
    - *The return value of wait is the number of the child process that has just terminated.*
    - *When there are no (or no longer any) child processes to wait for, the wait function returns -1.*

- Each time a child terminates, the parent process exits ***wait*** and consults "***status***" to obtain information about the child that has just terminated.
  - *"**status**" is a pointer to a two-byte word:*
    - *the high byte contains the value returned by the child (i from the exit(i) function),*
    - *the low byte: contains 0 in the general case,*
      - *If the child process terminates abnormally, this low byte contains the value of the signal received by the child.*
      - *This value is increased by 80 in hexadecimal (128 decimal), if this signal has caused the process memory image to be saved in a core file.*


15
8
7
0
exit(i)
i
The number of the signal that interrupted the child or 0
bit set to 0 or 1 depending on whether or not a core file has been produced

Relation between processes: Competition/ Cooperation

- In a system, several processes can take place simultaneously.
- During their evolution, the processes in a system interact with each other.
- Depending on whether or not the processes know each other, there are two types of interaction: Co-operation/ Competition.
  - Competition:
    - This is the situation in which several processes must simultaneously use an exclusive-access resource (a resource that can only be used by one process at a time).
    - Example: Printer, global variable.
    - One possible solution (but not the only one) for managing competition is to make the requesting processes wait until the current occupant has finished (first come, first served): FIFO

- *Co-operation:*
  - *If processes know each other, this is called co-operation.*
  - *This is the situation in which several processes collaborate on a common task and must synchronize to carry out this task.*
  - *A process is said to be cooperative if it can affect other processes running in the system or be affected by execution.*
  - *Examples:*
    - *P1 produces a file, P2 prints the file,*
    - *P1 updates a file, P2 consults the file,*

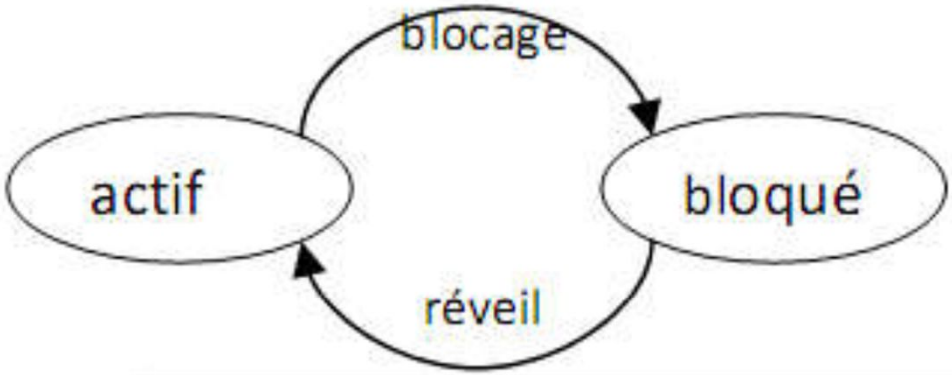
- Synchronization means that one process has to wait for another process to reach a certain point in its execution.
- For both types of relation (competition or cooperation), a process must be made to wait. How can we do this?
  - **Solution 1: active waiting**

P1	P2

`while (resource occupée)`
`  {}`
`ressource occupée = true;`
	`ressource occupée = true;`
	`utiliser resource;`
	`ressource occupée = false ;`

  - Disadvantage: active waiting monopolizes the processor and therefore degrades the overall performance of the machine.

- **Solution 2 : blocking the process**
  - *We define a new state for processes, the blocked state. The execution of a blocked process is stopped until it is explicitly woken up by another process or by the system.*




```
…
 sleep(5);/* se bloquer pendant 5 secondes */
…
```