République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
L'université Mohamed Boudiaf - M'Sila –

Faculté de mathématiques et d'informatique

2ème Année Licence (2L)

Système d'exploitation 1 (SE1)

Semestre : 04

2023/2024

Réalise par

Dr. DABBA ALI

➢ Contactez-nous

alidabba@gmail.com

ali.dabba@univ-msila.dz

- En cas de problèmes ou de difficultés, me contacter ou contacter votre enseignant TD / TP
- Nous sommes à votre disposition pour vous aider

CHAPITRE 3

Gestion des processus

Plan

I. Introduction

II. Cycle de vie d'un processus

III. Qu'est-ce que l'ordonnancement de processus ?

IV. Les différentes politiques de scheduling

V. Les critères d’évaluation de performances entre les algorithmes d’ordonnancement

VI. Les algorithmes de scheduling

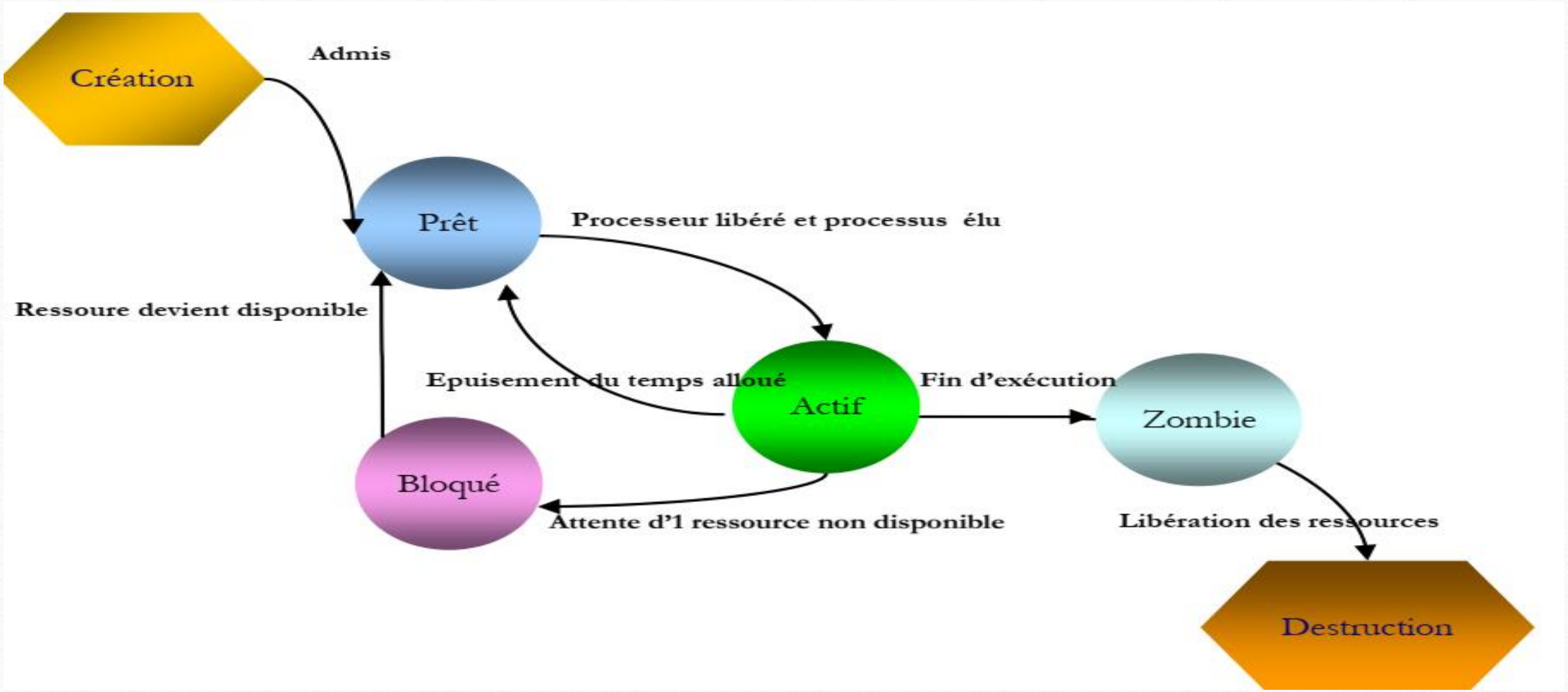
I. Introduction

Dans l'exécution des différents processus, il faut disposer de plusieurs ressources machine contribuant au long du cycle de vie d'un processus, la plus importante c'est la ressource qui effectue l'exécution elle-même, c'est le processeur. Alors, pour cette ressource, **le système d'exploitation doit doter d'un ensemble de mécanismes gérant d'une manière efficace et optimale le travail du processeur et contrôlant l'exécution des processus.**

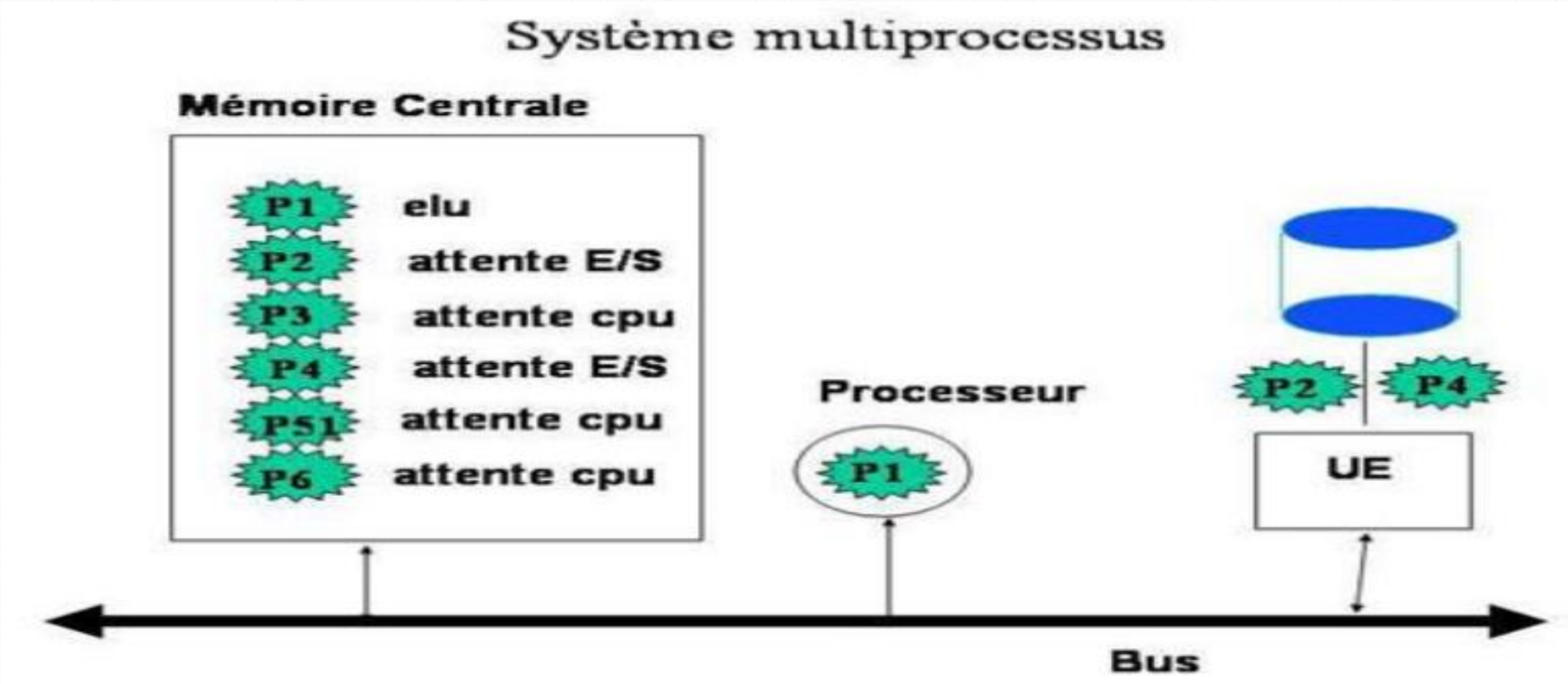
II. Cycle de vie d'un processus

Un processus possède un cycle de vie allant de sa naissance à sa mort passant par divers phases d'activité et d'attente. La figure suivante illustre le cycle de vie d'un processus dans le cadre d'un système mono processeur avec stratégie de partage de temps entre les processus.

II. Cycle de vie d'un processus



III. Qu'est-ce que l'ordonnancement de processus ?



III. Qu'est-ce que l'ordonnancement de processus ?

Définition 3.1

La fonction d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt. L'opération d'élection consiste à allouer le processeur à un processus.

Chaque fois, que le processeur devient inactif, le système d'exploitation doit sélectionner un processus de la file d'attente des processus prêts, et lui passe le contrôle. D'une manière plus concrète, cette tâche est prise en charge par deux routines système en l'occurrence le **Dispatcheur** et le **Scheduleur (Ordonnanceur).**

III. Qu'est-ce que l'ordonnancement de processus ?

- **Le Dispatcheur** : Il s'occupe de l'allocation du processeur à un processus sélectionné par l'Ordonnanceur du processeur. Une fois allouer, le processeur doit réaliser les tâches suivantes
  - **Commutation de contexte** : sauvegarder le contexte du processus qui doit relâcher le processeur et charger le contexte de celui qui aura le prochain cycle processeur.
  - **Commutation du mode d'exécution** : basculer du mode Maître (mode d'exécution du dispatcheur) en mode utilisateur (mode d'exécution du processeur utilisateur).
  - **Branchement** : se brancher au bon emplacement dans le processus utilisateur pour le faire démarrer.

III. Qu'est-ce que l'ordonnancement de processus ?

- **L'Ordonnanceur** : Certains systèmes d'exploitation utilisent une technique d'ordonnancement à deux niveaux qui intègre deux types d'Ordonnanceurs :
  - **Ordonnanceur du processeur**
  - **Ordonnanceur de travail**

III. Qu'est-ce que l'ordonnancement de processus ?

- **Questions**
  - Dans le cas de N processus prêts, à quel processus doit-on allouer le CPU ?
  - Un processus en attente est-il obliger d'attendre la fin de celui actif ?
  - Si un processus urgent (prioritaire) demande le CPU, doit-il attendre ?
  - Quelles sont les politiques (ou stratégies) d'allocation du processeur ?
  - Comment peut-on optimiser l'utilisation du CPU pour maximiser les performances du SIQ ?

III. Qu'est-ce que l'ordonnancement de processus ?

Définition 3,2

L'ordonnanceur (scheduler en anglais) est un programme du SE qui s'occupe de choisir, selon une politique (ou algorithme) d'ordonnancement donnée, un processus parmi les processus prêts pour lui affecter le processeur. Sur le plan ordonnancement, l'ordonnanceur élit un processus prêt selon un algorithme qui doit garantir :

- **L'équité** : chaque processus doit avoir sa part du temps processeur.
- **L'efficacité** : le taux d'utilisation du processeur doit être élevé.
- **Un temps de réponse minimal.**
- **Un temps moyen d'exécution minimal** de manière à ce que l'attente des processus soit minimale.
- **Un rendement maximal** de façon à ce que le nombre de travaux effectués soit élevé.

IV. Les différentes politiques de scheduling

- Deux types d'ordonnancement sont considérés dans les systèmes d'exploitation: et l'ordonnancement nan préemptif (sans réquisition).
  - **Le scheduling non préemptif (l'ordonnancement nan préemptif (sans réquisition).)**
  - **Le scheduling préemptif (l'ordonnancement préemptif (avec réquisition))**

IV. Les différentes politiques de scheduling

- **Le scheduling non préemptif**

  Un algorithme d'ordonnancement non préemptif sélectionne un processus, puis le laisse s'exécute jusqu'à ce qu'il bloque (soit sur une E/S, soit en attente d'un autre processus) ou qu'il libère volontairement le processeur. Même s'il s'exécute pendant des heures, il ne sera pas suspendu de force. En effet, aucune décision d'ordonnancement n'intervient pendant les interruptions d'horloge. Une fois le traitement de l'interruption est terminé, le processus qui était en cours d'exécution avant l'interruption est toujours relancé.

IV. Les différentes politiques de scheduling

- **Le scheduling préemptif**

  Un algorithme de scheduling préemptif sélectionne un processus et le laisse s'exécuter pendant un délai déterminé. Si le processus est toujours en cours à l'issue de ce délai, il est suspendu (mis dans la file d'attente des processus prêts), et l'ordonnanceur sélectionne un autre processus à exécuter. L'ordonnancement préemptif nécessite une interruption à la fin du délai afin de redonner le contrôle de l'UC à l'ordonnanceur.

V. Les critères d'évaluation de performances

- Utilisation du processeur
- Capacité de traitement
- Le *rendement* d'un système
- ***Le temps de réponse***

$$\text{TRM} = \sum_{i=1}^{n} TR_i/n \text{ , avec } TR_i = \text{date fin} - \text{date arrivée}$$

- ***Le temps d'attente***

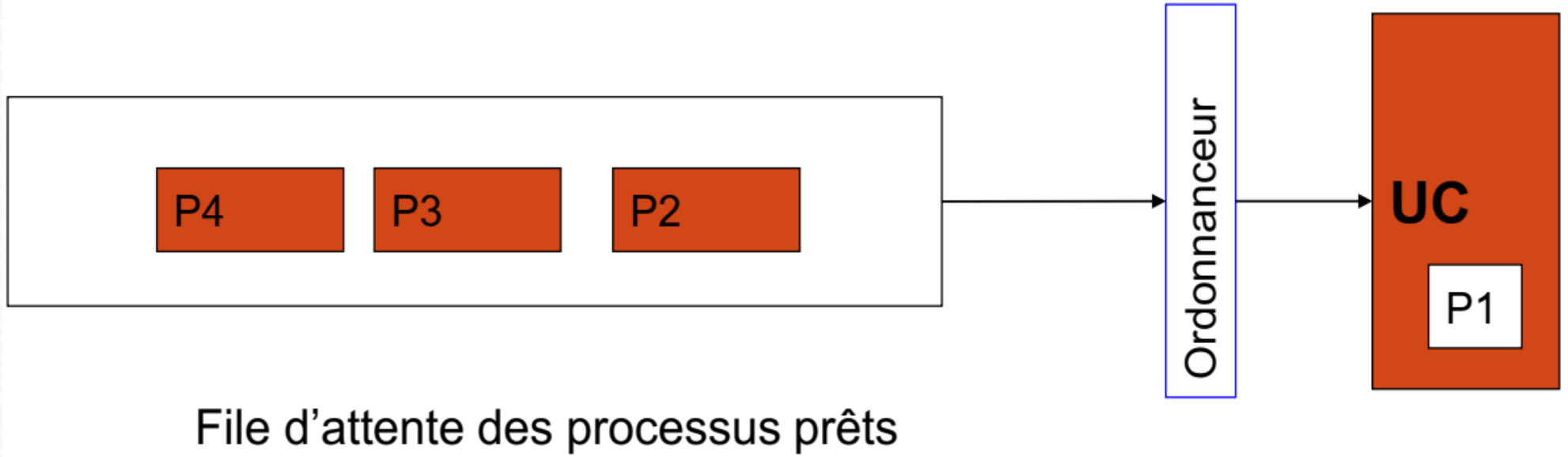
$$\text{TAM} = \sum_{i=1}^{n} TA_i/n \text{ , avec } TA_i = TR_i - \text{temps d'exécution}$$

V. Les critères d'évaluation de performances

Remarque *: Pour représenter schématiquement l'évolution dans le temps des processus, on recourt habituellement à des* ***diagrammes de Gantt.***

VI. Les algorithmes de scheduling

➢ **L'algorithme du Premier Arrivé Premier Servi (FCFS)**

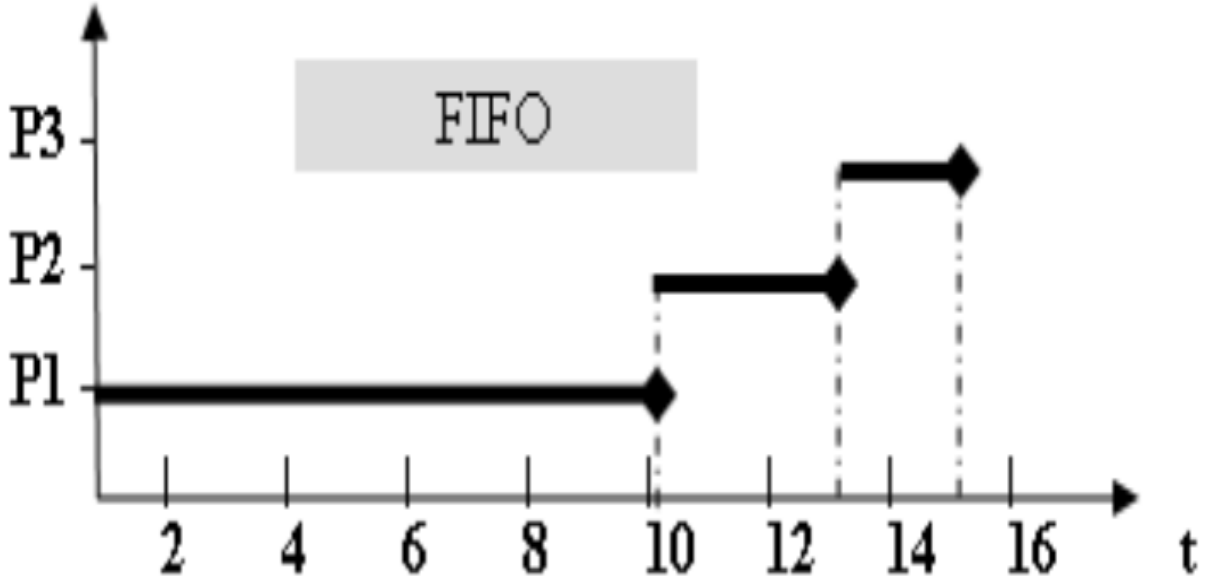


VI. Les algorithmes de scheduling

➢ L'algorithme du Premier Arrivé Premier Servi (FCFS)

EXP

Processus	Temps CPU	Ordre d'arrivée
P1	10	1
P2	3	2
P3	2	3

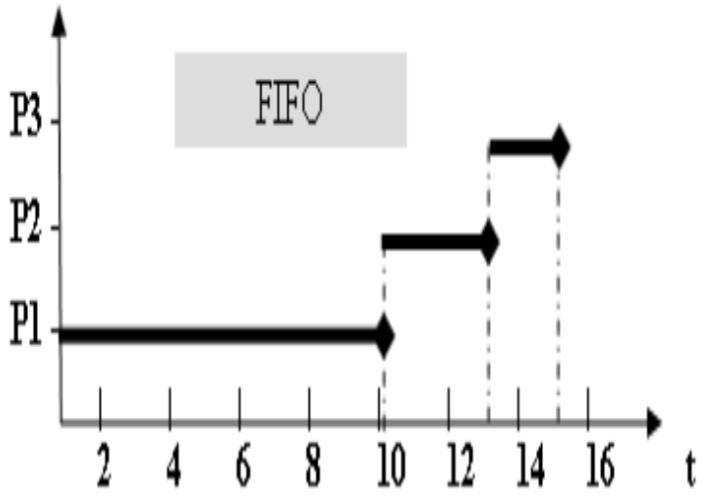


VI. Les algorithmes de scheduling

➢ **L'algorithme du Premier Arrivé Premier Servi (FCFS)**

EXP

Processus	Temps CPU	Ordre d'arrivée
P1	10	1
P2	3	2
P3	2	3



➢ **Avantages:**

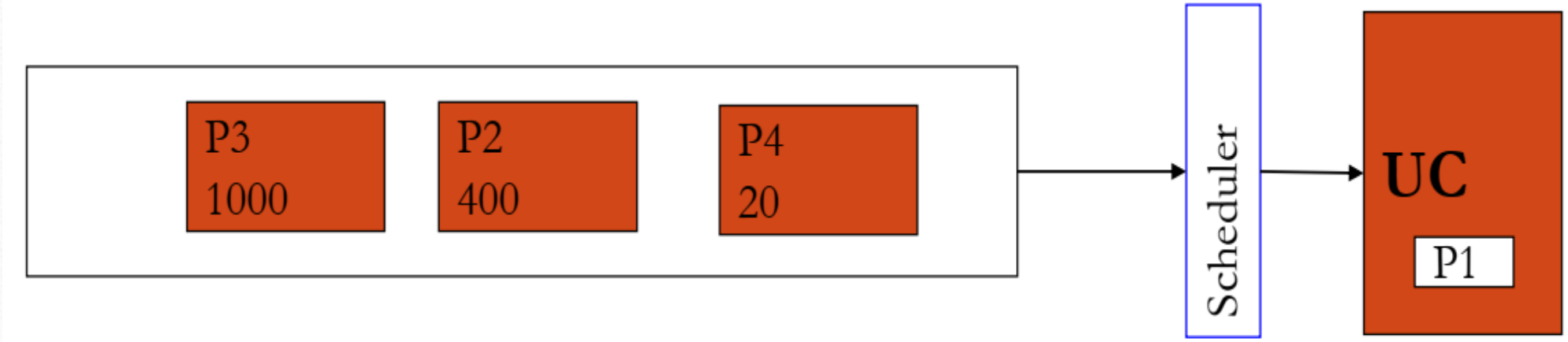
- Implémentation simple.

➢ **Inconvénients:**

- Les processus longs pénalisent les processus courts.
- Performances généralement mauvaises.

VI. Les algorithmes de scheduling

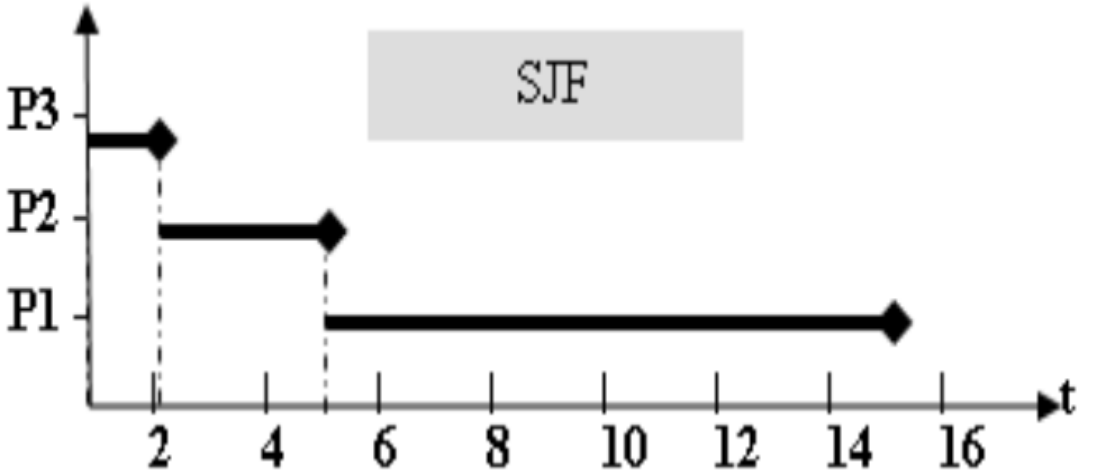
➢ **L'algorithme du Plus Court d'abord (SJF)**



VI. Les algorithmes de scheduling

➢ **L'algorithme du Plus Court d'abord (SJF)**

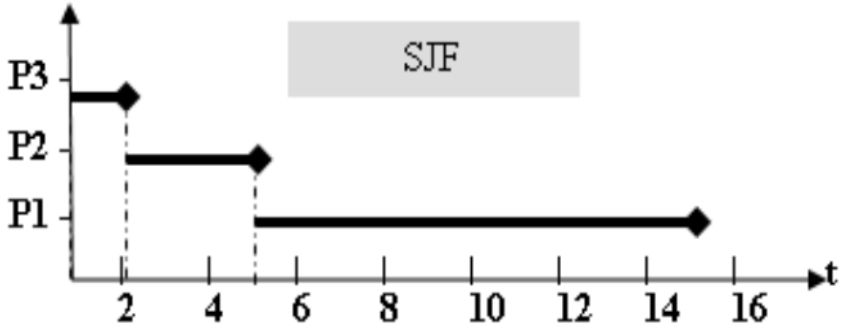
Processus	Temps CPU
P1	10
P2	3
P3	2



VI. Les algorithmes de scheduling

- **L'algorithme du Plus Court d'abord (SJF)**

Processus	Temps CPU
P1	10
P2	3
P3	2



- **Avantage**

  -Performances meilleures en temps d'attente moyen TAM.

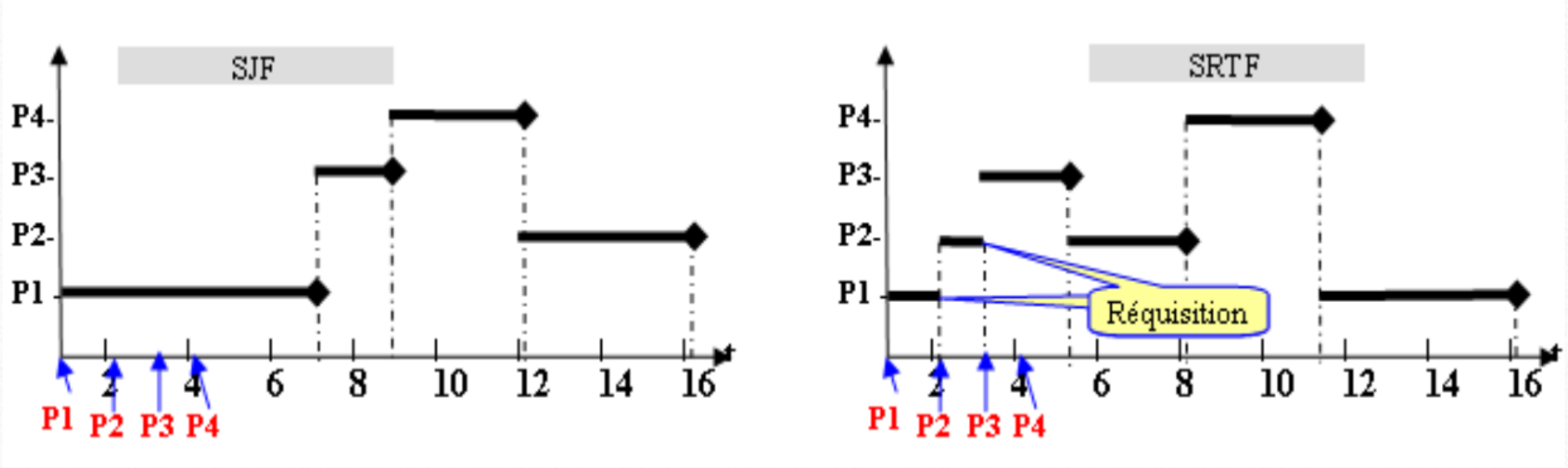
- **Inconvénient**

  - Il est difficile d'estimer le temps d'exécution d'un processus
  - Génération de la **Famine**

VI. Les algorithmes de scheduling

- **SRTF (Shortest Remaining Time First) ou SJF avec requisition**

Processus	Temps CPU	temps d'arrivée
P1	7	0
P2	4	2
P3	2	3
P4	3	4



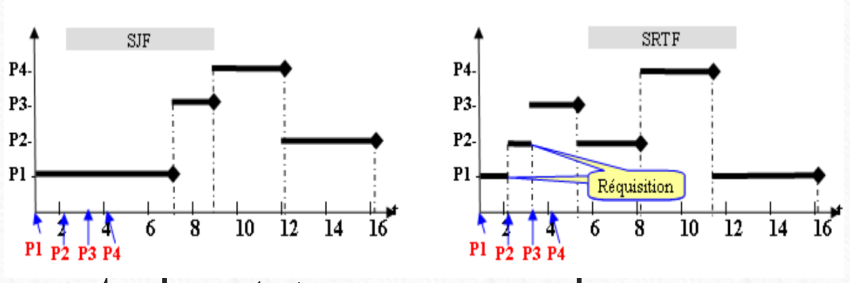
VI. Les algorithmes de scheduling

- **SRTF (Shortest Remaining Time First) ou SJF avec requisition**

Processus	Temps CPU	temps d'arrivée
P1	7	0
P2	4	2
P3	2	3
P4	3	4

- **Avantage:**

  -Les processus courts ne sont pas obligés d'attendre de longs périodes de temps, ce qui améliore le TAM.

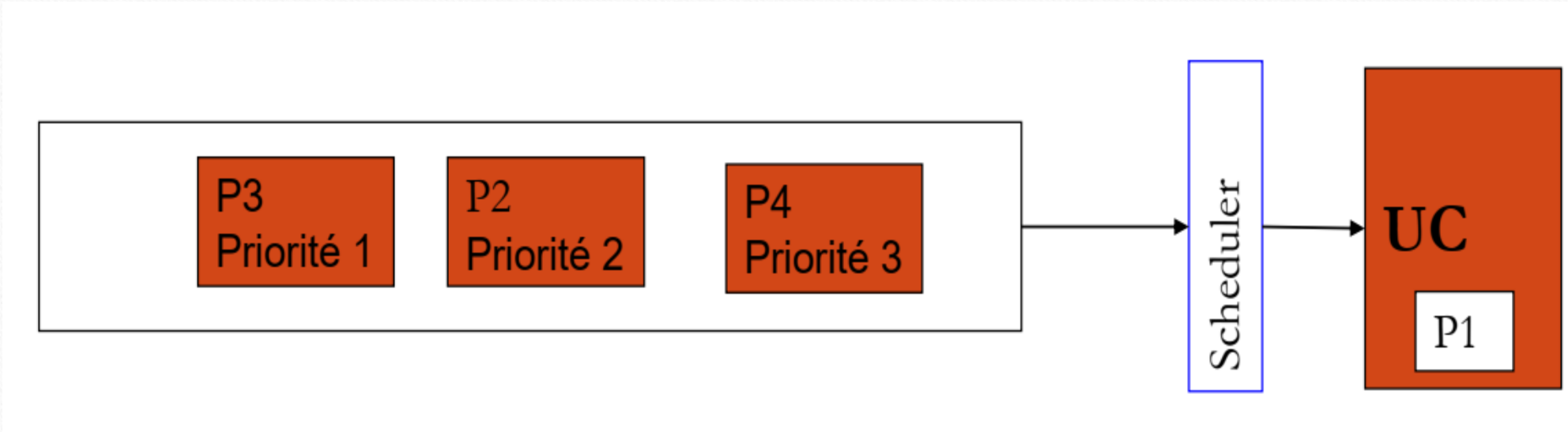


- **Inconvénient:**

  - Une famine peut être crée par le traitement des processus de court durée en laissant attendre les processus long

VI. Les algorithmes de scheduling

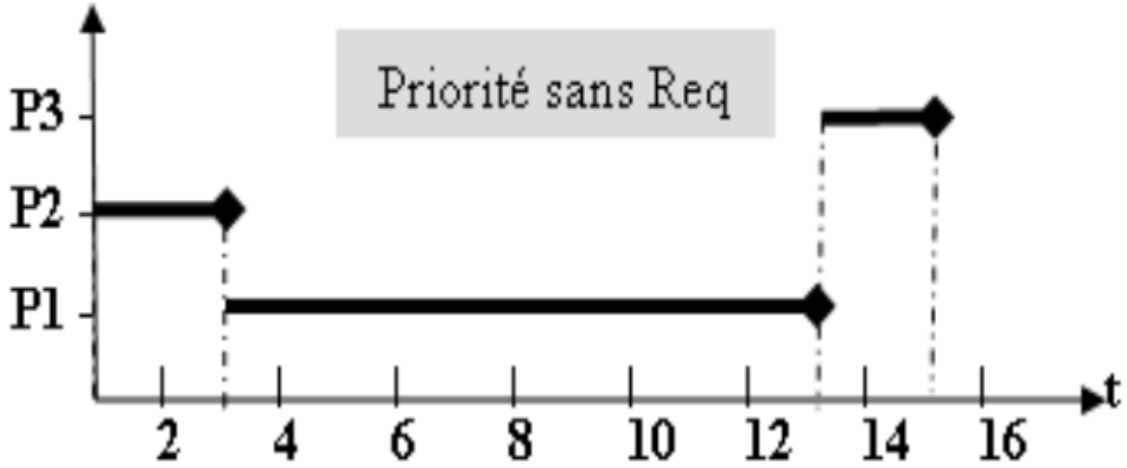
➢ **Scheduling avec priorité**



VI. Les algorithmes de scheduling

- **Scheduling avec priorité (sans réquisition)**

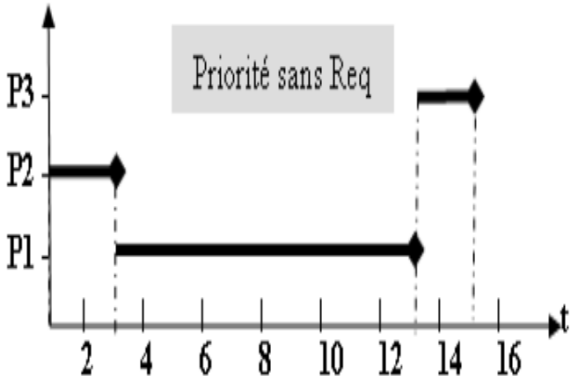
Processus	*Temps CPU*	*priorité*
P1	*10*	*2*
P2	*3*	*3*
P3	*2*	*1*



VI. Les algorithmes de scheduling

➢ **Scheduling avec priorité (sans réquisition)**

Processus	Temps CPU	priorité
P1	10	2
P2	3	3
P3	2	1



➢ **Avantage:**

- Possibilité de spécifier la priorité à priori

➢ **Inconvénient :**

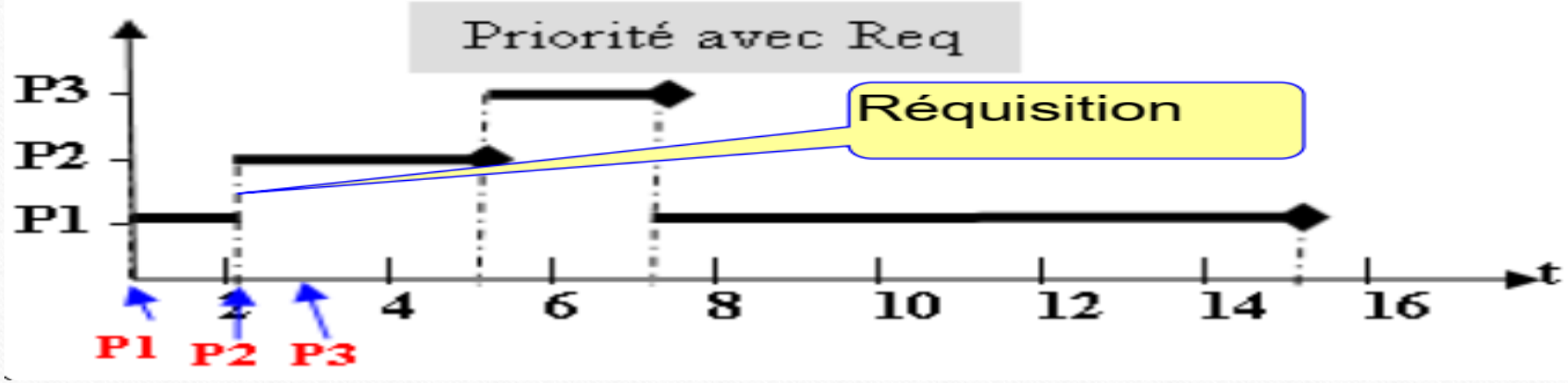
- l'arrivée successive de processus de hautes priorités pénalise les processus de faibles priorités

VI. Les algorithmes de scheduling

- **Politique par priorité avec réquisition**

Processus	Temps CPU	Temps d'arrivée	priorité
P1	10	0	1
P2	3	2	3
P3	2	3	2

t1 = Tr1

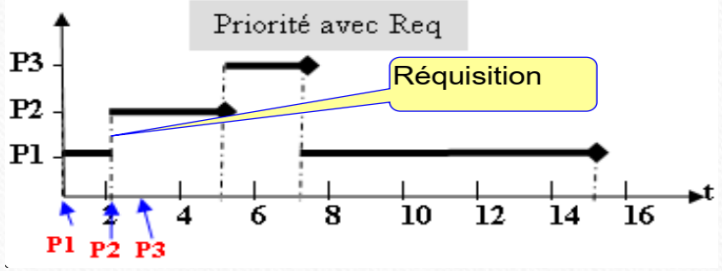


VI. Les algorithmes de scheduling

- **Politique par priorité avec réquisition**

Processus	Temps CPU	Temps d'arrivée	priorité
P1	10	0	1
P2	3	2	3
P3	2	3	2

t1 = Tr1



- ***Avantage:***

  - *Possibilité de spécifier la priorité à priori*

- ***Inconvénient:***

  - L'arrivée successive de processus de hautes priorités pénalise les processus de faibles priorités (famine).

VI. Les algorithmes de scheduling

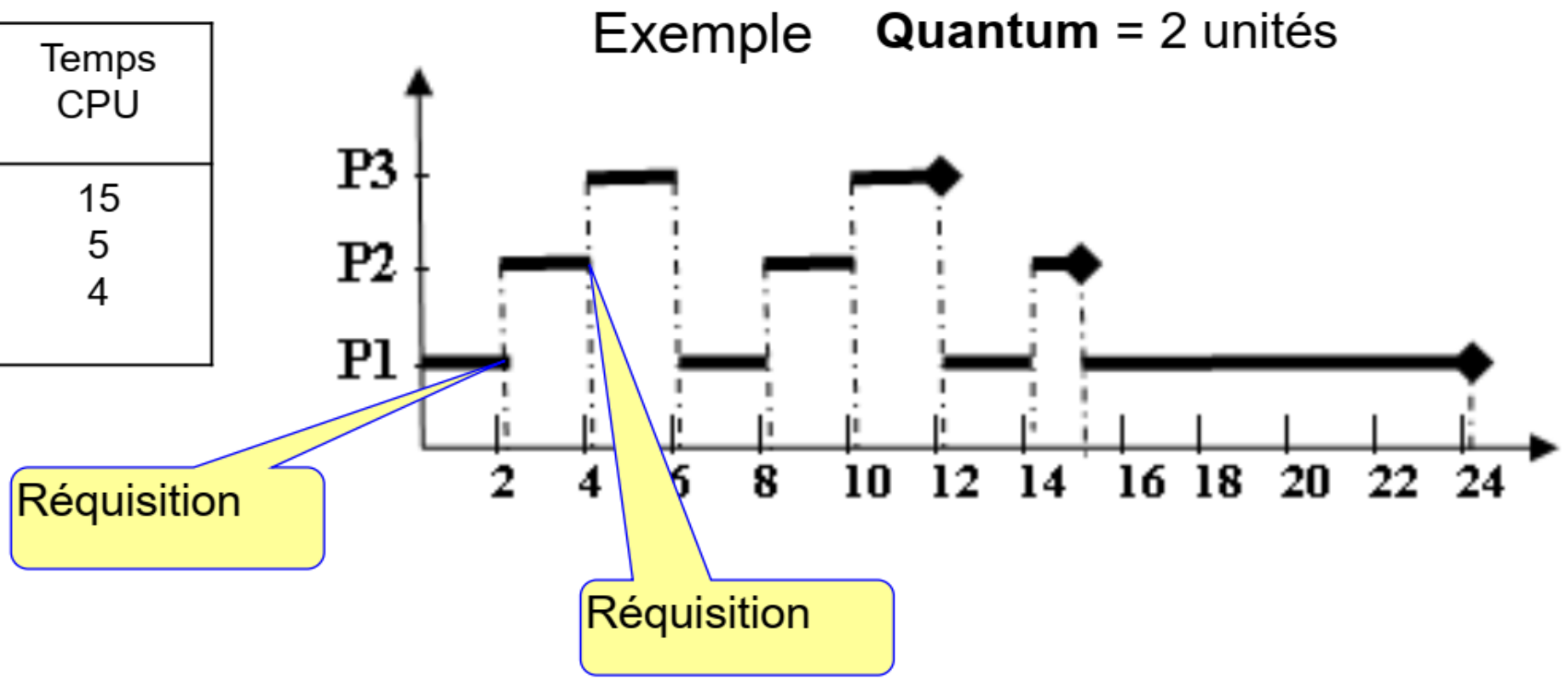
- **L'algorithme de Round Robin (Tourniquet)**

  Le CPU est alloué à chaque processus existants dans la file des prêts d'une façon FIFO circulaire pendant une durée de temps bien déterminée Q appelée **quantum de temps**. A la fin des quantum, le CPU est enlevé du processus actif au profit du processus suivant dans la file des prêts.

VI. Les algorithmes de scheduling

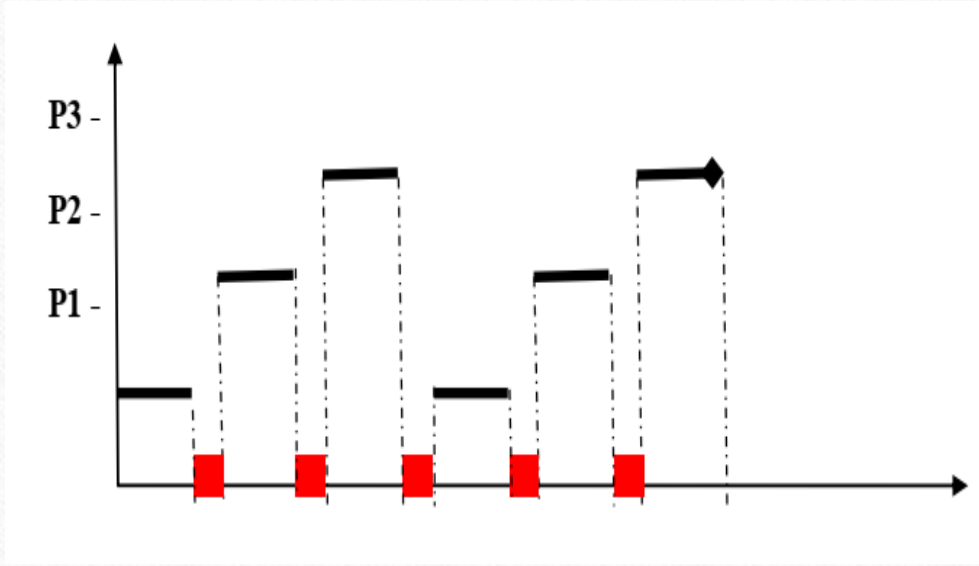
➢ L'algorithme de Round Robin (Tourniquet)

Processus	Temps CPU
P1	15
P2	5
P3	4



VI. Les algorithmes de scheduling

➢ **L'algorithme de Round Robin (Tourniquet)**



➢ **Avantage:**

-Politique qui s'adapte bien

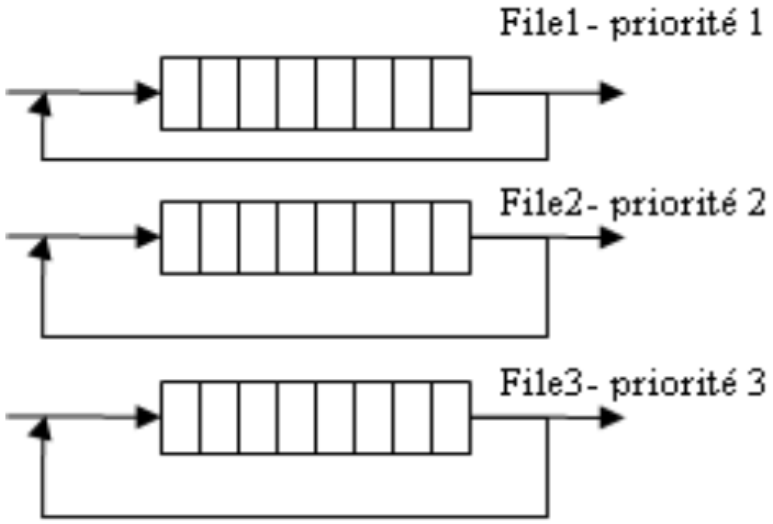
dans les systèmes à temps partagés.

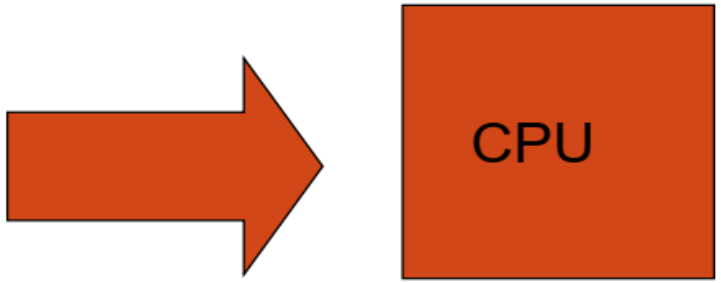
➢ **Inconvénient:**

- Le temps consommé pour chaque commutation de contexte réduit les performances du système.

VI. Les algorithmes de scheduling

➢ **Scheduling avec files d'attente multiniveaux**

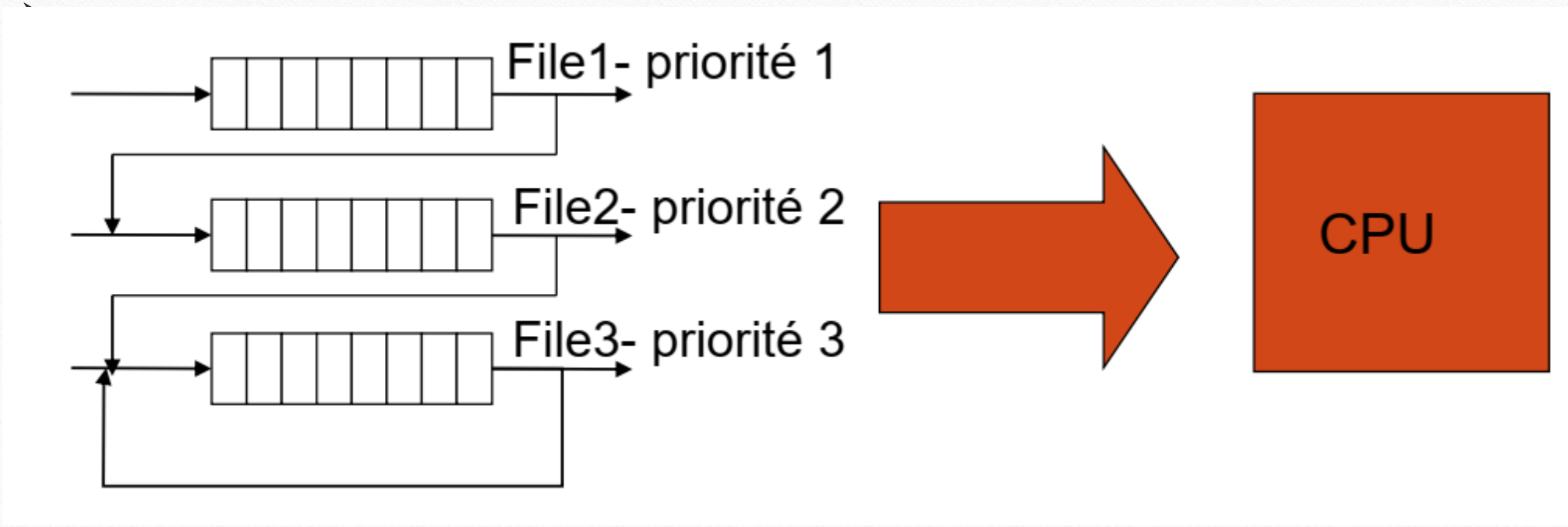




Round robin multiniveaux
avec priorités constantes

VI. Les algorithmes de scheduling

- Scheduling avec files d'attente multiniveaux et feedback



VI. Les algorithmes de scheduling

- **Question:** Quelle est la meilleure politique d'ordonnancement ?
  - **On ne peut pas juger une politique que dans un contexte bien déterminé**
  - **Une politique qui brille dans un contexte peut échouer dans un autre.**

Exemple : RR est bonne dans un système à temps partagé elle est dangereuse dans un système temps réel

VII. Processus sous UNIX

- Un processus est un programme qui s'exécute.
- Un processus possède un **identificateur** qui est un numéro : le **pid**. Il s'agit d'un entier du type **pid_t** déclare comme synonyme du type **int** ou **unsigned long int**.
- Un processus incarne — exécute — un programme ; il appartient à un utilisateur ou au noyau.
- Un processus possède une priorité et un mode d'exécution.

VII. Processus sous UNIX

- Nous distinguerons deux modes : le mode utilisateur (ou esclave) de basse priorité et le mode noyau (ou maître) de plus forte priorité.
- Généralement un processus appartient à la personne qui a écrit le programme qui s'exécute, mais ce n'est pas toujours le cas. Quand vous exécutez la commande **ls** vous exécutez le programme **/bin/ls** qui appartient à **root**. Regardez le résultat de la commande **ls -l !**

VII. Processus sous UNIX

- **Exemple**

```
% ls -l /bin/ls
-rwxr-xr-x 1 root    root        29980 Apr 24 1998  /bin/ls
%
```

- Un processus naît, crée d'autres processus (ses fils), attend la fin d'exécution de ceux-ci ou entre en compétition avec eux pour avoir des ressources du système et enfin meurt. Pendant sa période de vie, il accède à des variables en mémoire suivant son mode d'exécution. En mode noyau, tout l'espace adressable lui est ouvert ; en mode utilisateur, il ne peut accéder qu'à ses données privées.

Les identificateurs de processus

- Chaque processus possède un identificateur unique nommé **pid**. Comme pour les utilisateurs, il peut être lié à un groupe ; on utilisera alors l'identificateur pgrp. Citons les différentes primitives permettant de connaître ces différents identificateurs :

  - **pid_t getpid()** **/* retourne l'identificateur du processus */**
  - **pid_t getpgrp()** **/* retourne l'identificateur du groupe de processus */**
  - **pid_t getppid()** **/* retourne l'identificateur du père du processus */**
  - **pid_t setpgrp()** **/* positionne l'identificateur du groupe de */**

    /* processus à la valeur de son pid : crée un */

    /* nouveau groupe de processus */

Valeur retournée : nouvel identificateur du groupe de processus.

Les identificateurs de processus

- **Exemple :**

```
/* fichier test_idf.c */

#include main()

{

        printf("je suis le processus %d de père %d et de groupe

                %d\n",getpid(),getppid(),getpgrp()) ;

}
```

Les identificateurs de processus

- **Résultat de l'exécution :**

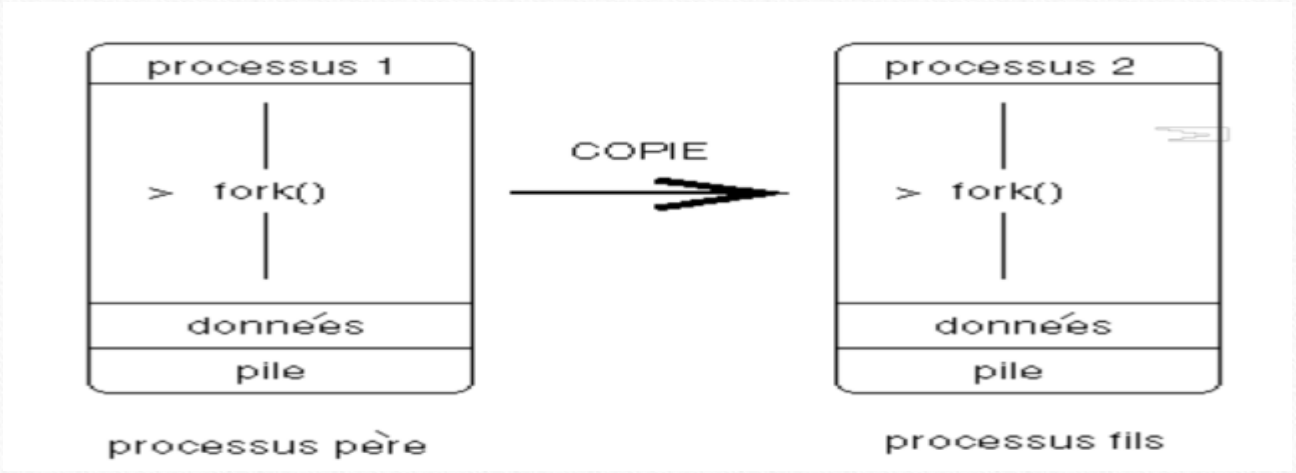
```
Systeme> ps
PID TTY TIME COMMAND
6658 ttyp5 0:04 csh
Systeme> test_idf
je suis le processus 8262 de père 6658 et de groupe 8262
```

- Notons que le père du processus exécutant **test_idf** est **csh**.

Création de processus

- Un processus est créé par une instruction spéciale **fork**. Le processus créé (le **fils**) est un **clone** (**copie conforme**) du processus créateur (le **père**).

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```



Création de processus

- Le père (**Parent**) et le fils (**Child**) ne se distinguent que par le résultat retourné par **fork**. Pour le père, cette fonction renvoie le **numéro du fils** (ou **-1** si création impossible) et pour le fils, elle renvoie **0**.

```
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
          ...
        case 0: /* Programme du fils */
          ...
        default: /* Programme du père */
          ...
}
```

L'hiérarchie des processus

- Sous UNIX, les processus forment une hiérarchie (père-fils) (Figure 3.4). La commande UNIX **ps -aeH** permet d'afficher la hiérarchie complète des processus, y compris le processus **init**.

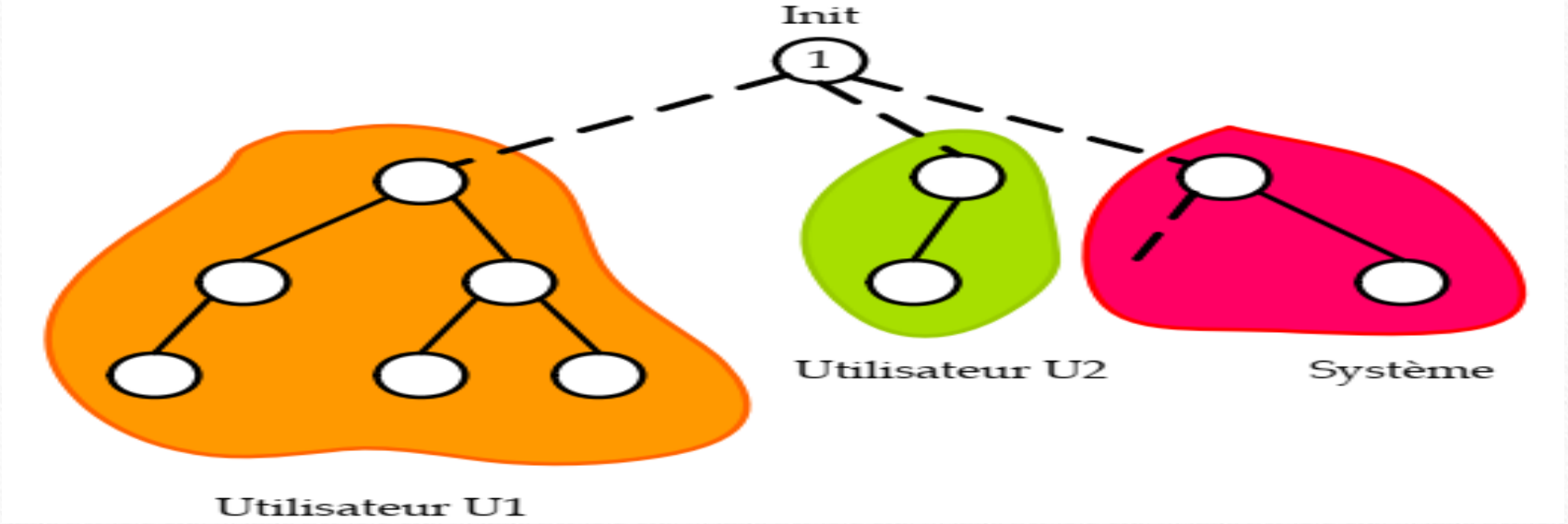
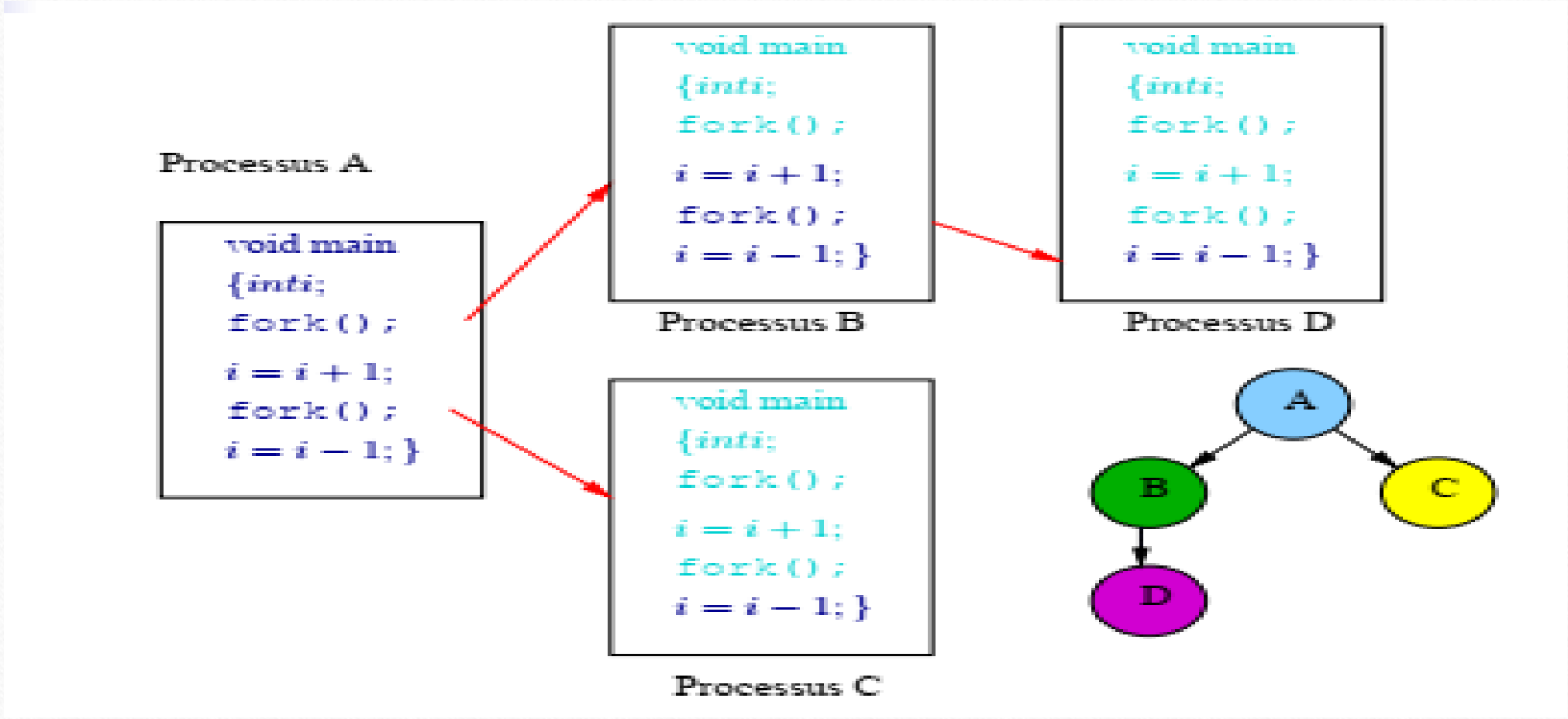


Figure 3.4 : Hiérarchie des processus

L'hiérarchie des processus



Terminaison d'un processus

- Dans des conditions normales, un processus peut se terminer de deux façons:
  - Soit la fonction main du programme se termine,
  - Soit le programme appelle la fonction exit.

```
#include <stdlib.h>
void exit (int status);
```

Terminaison d'un processus

- Comme il peut être explicitement éliminé par un autre processus, avec l'envoi d'un signal du type **kill**. Ceci peut se faire de deux façons :
  - Soit à partir d'un programme, en utilisant la fonction **kill()**.

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

Terminaison d'un processus

- **kill()** envoie le signal numéro **sig** à un processus identifié par son **pid**. En cas d'erreur elle retourne **-1**, et **0** autrement.

```
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
            ...
        case 0: /* Programme du fils */
            ...
        default: /* Programme du père */
            ...
            /*Envoi d'un signal de terminaison au fils*/
            kill(pid, SIGTERM);
}
```

Terminaison d'un processus

- **kill()** envoie le signal numéro **sig** à un processus identifié par son **pid**. En cas d'erreur elle retourne **-1**, et **0** autrement.

```
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
            ...
        case 0: /* Programme du fils */
            ...
        default: /* Programme du père */
            ...
            /*Envoi d'un signal de terminaison au fils*/
            kill(pid, SIGTERM);
}
```

- Soit à partir du shell, en invoquant la commande kill.

```
$ kill pid
```

La commande **kill** envoie par défaut un signal **SIGTERM**, ou de terminaison au processus d'identité **pid**.

Code de sortie d'un processus

Chaque processus a un code de sortie (**Exit Code**) : un nombre que le processus renvoie à son parent. Le code de sortie est l'argument passé à la fonction **exit** ou la valeur retournée depuis **main**. Par convention, le code de sortie est utilisé pour indiquer si le programme s'est exécuté correctement :

- Un code de sortie à **zéro** indique une **exécution correcte**,
- Un code **différent de zéro** indique qu'une **erreur** est survenue.

```
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
           ...
           exit(1);
        case 0: /* Programme du fils */
           ...
           exit(0);
        default: /* Programme du père */
           ...
}
```

Synchronisation père-fils

Lorsqu'un processus fils se termine, il devient un processus **zombie**. Un processus zombie ne peut plus s'exécuter, mais consomme encore des ressources (son entrée dans la table des processus n'est pas enlevée). Il restera dans cet état jusqu'à ce que son processus père ait récupéré son code de sortie. Les appels systèmes **wait** et **waitpid** permettent au processus père de récupérer ce code. A ce moment le processus termine et disparaît complètement du système.

- L'appel système **wait()**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int* status);
```

Synchronisation père-fils

L'appel système wait suspend l'exécution du processus appelant jusqu'à ce qu'un de ses fils se termine. Si un fils est déjà terminé, **wait** renvoie le PID du fils immédiatement sans bloquer. Il retourne l'identifiant du processus fils et son état de terminaison dans **status** (si **status** est différent de NULL). Si par contre le processus appelant ne possède **aucun fils**, **wait** retourne **-1**.

```
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
                ...
        case 0: /* Programme du fils */
                printf("processus fils %d\n", getpid());
                exit(10);
        default: /* Programme du père */
                printf("processus père %d\n", getpid()) ;
                pid_t id = wait (&status);
                printf("fin processus fils %d\n", id);
                exit(0);
}
```

Synchronisation père-fils

- L'appel système **waitpid()**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid , int* status , int options);
```

- L'appel système **waitpid()** permet à un processus père d'attendre un fils particulier d'identité **pid**, de façon **bloquante** (**options=0**) ou **non bloquante** (**options=WNOHANG**).
- Si l'appel réussit, il renvoie l'identifiant du processus fils et son état de terminaison dans **status** (si **status** n'est pas NULL). En cas d'erreur **-1** est renvoyé. Si l'option WNOHANG est utilisée et aucun fils n'a changé d'état, la valeur de retour est **0**.

Synchronisation père-fils

```
pid_t id;
pid_t pid=fork();
switch(pid){
case -1: /* Erreur de création */
        ...
    case 0: /* Programme du fils */
        ...;
    exit(10);
    default: /* Programme du père */
        ...
        While((id=waitpid(pid,&status,WNOHANG))==0)
        printf("processus fils non encore terminé\n");
        printf("fin processus fils %d\n", id) ;
        ...
}
```

L'appel **waitpid(-1, &status, 0)** est équivalent à l'appel **wait(&status)**

Synchronisation père-fils

Code de sortie retourné par wait() et waitpid()

- Pour extraire le code de retour du processus fils de **status**, il faut d'abord tester que le processus c'est terminé normalement. Pour cela, on utilise la macro **WIFEXITED(status)**, qui renvoie **vrai** si le processus fils c'est terminé normalement.
- Ensuite pour obtenir son code de retour, on utilise la macro **WEXITSTATUS(status)**.

Synchronisation père-fils

```
pid_t id;
pid_t pid=fork();
switch(pid){
        case -1: /* Erreur de création */
            ...
        case 0: /* Programme du fils */
            ...;
        exit(10);
        default: /* Programme du père */
            ...
            while((id=waitpid(pid,&status,WNOHANG))==0)
            printf("processus fils non encore terminé\n");
            if(WIFEXITED(status))
            printf("fils %d terminé par exit(%d)\n",id, WEXITSTATUS(status));
            ...
}
```

VIII. La communication interprocessus : les signaux

- Un signal est un moyen de communication des événements dans le système UNIX. Chaque signal a une signification particulière liée à la source qui a produit l'événement. Les signaux constituent la forme la plus simple de communication entre processus. Un signal peut être généré par le matériel, par le logiciel, ou par l'utilisateur lui-même. Par exemple, une division par zéro dans l'unité centrale de traitement provoque une interruption matérielle portée par le signal SIGFPE (SIG floating point error). Un appui simultané des touches **Ctrl et C (^C)** génère le signal d'interruption SIGINT (2). Les signal SIGCHLD est généré quand un programme termine son exécution.

VIII.La communication interprocessus : les signaux

Définition 3.3

Un signal est une **interruption logicielle** asynchrone d'un processus. Le signal peut être envoyé par un **processus** ou par le **noyau**.

Identification des signaux

- Les signaux sont identifiés par un **numéro entier** et un **nom symbolique** décrit dans **signal.h**. Le nombre de signaux disponibles dans un système varie suivant la version du système de 0 à 31, de 0 à 63, etc. Les valeurs peuvent changer d'une implémentation à l'autre; les noms symboliques restent les mêmes. La commande **kill -l** donne la liste des signaux système. Le tableau ci-après présente quelques principaux signaux :
- L'ensemble des signaux est décrit dans **/usr/include/sys/signal.h** ou dans **/usr/include/bits/signum.h**.

Nom du signal	Evénement associé
SIGTERM	Terminaison d'un processus.
SIGINT	(interrupt) signal d'interruption émis à tous les processus associés à un terminal lorsque le caractère (Ctrl – c) est tapé.
SIGKILL	Terminaison d'un processus (non déroutable)
SIGQUIT	Signal d'interruption émis à tous les processus associés à un terminal lorsque le caractère (Ctrl –\) est tapé.
SIGTSTP	Arrêt temporaire d'un processus (Ctrl – z).
SIGSTOP	Suspension du processus (non déroutable).
SIGCONT	Signal de continuation d'un processus stoppé
SIGCHLD	Terminaison d'un processus fils
SIGABRT	Terminaison anormale provoquée par l'exécution de la fonction abort.
SIGALRM	Expiration d'un timer (fonction alarm)
SIGUSR1	Signal utilisateur 1 (disponible pour les applications)
SIGUSR2	Signal utilisateur 2 (disponible pour les applications)
SIGSEGV	Violation des protections mémoire (Segmentation Fault)
SIGFPE	Erreur arithmétique (Ex. division par zéro). (Floating-Point Arithmetic Error)
SIGILL	Détection d'une instruction illégale. (ILLegal)
SIGTRAP	Emis après chaque instruction en cas d'exécution en mode trace, par un débogueur.
SIGPIPE	Ecriture dans un tube sans lecteur.

Origines d'un signal

- Les signaux Unix ont des origines diverses, ils peuvent être :
- transmis par le noyau (division par zéro, overflow, instruction interdite, …etc.),
- envoyés par l'utilisateur depuis le clavier (touches Ctrl – z , Ctrl – c, …etc.), émis par la commande kill depuis le shell,
- par un autre processus : primitives kill(), sigqueue() (émis par la commande kill depuis le shell ou par la fonction kill depuis un programme écrit en C)

Remarque : l'émetteur ne peut pas savoir si le destinataire a reçu ou non le signal.

comportement possibles pour un signal

Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal. On peut ainsi ignorer le signal, ou bien le prendre en compte, ou encore laisser le système d'exploitation appliquer le comportement par défaut, qui en général consiste à tuer le processus. Lorsqu'un processus reçoit un signal, plusieurs comportements sont possibles.

- Ignorer le signal. Pas tous les signaux pouvant être ignoré

comportement possibles pour un signal

- Appeler une routine de traitement fournie par le noyau. Il s'agit d'appliquer le comportement par défaut. Cette procédure provoque normalement la **mort du processus**. Dans certains cas, elle provoque la création d'un fichier **core**, qui contient le contexte du processus avant de recevoir le signal. Ces fichiers peuvent être examinés à l'aide d'un **débogueur**. c'est ce qu'on appelle un *core dump*.
- Appeler une procédure spécifique créée par le programmeur. Si un processus choisit de prendre en compte les signaux qu'il reçoit, il doit alors spécifier la procédure de gestion de signal. Quand un signal arrive, la procédure associée est exécutée. A la fin de l'exécution de la procédure, le processus s'exécute à partir de l'instruction qui suit celle durant laquelle l'interruption a eu lieu. Pas Tous les signaux permettent ce type d'action.

Réponse à un signal

Il existe trois manières de répondre à un signal :

- **Exécution de l'action par défaut** pour le signal. Cinq traitements par défaut sont possibles:
  - **exit** : provoque la terminaison du processus (Ex. SIGINT, SIGKILL, SIGALRM)
  - **core** : sauvegarde l'état de la mémoire et termine le processus (Ex. SIGFPE, SIGBUS, SIGSEGV)
  - **stop** : suspend l'exécution du processus (Ex. SIGSTOP)
  - **ignore** : le signal est ignoré (Ex. SIGCHLD)
  - **continue** : le processus suspendu reprend son exécution ou le signal est ignoré (Ex. SIGCONT)
- **Ignorance du signal.** On peut ignorer un certain nombre de signaux (sauf par exemple SIGKILL, SIGSTOP)
- **Interception** puis **invocation** d'une fonction en réaction à l'événement. Le signal peut être rattrapé par le processus destinataire, ce qui provoque un **déroutement** et le lancement d'une routine spécifique de traitement (**handler**). Après le traitement, le processus reprend où il a été interrompu.

Réponse à un signal

Un signal peut être dérouté soit à travers les appels systèmes **signal()** et **sigaction()** à partir d'un programme C/C++ ou en utilisant la commande **trap** à partir du shell.

- **L'appel système signal()**

```
#include <signal.h>
void (* signal(int signum, void (* handler)(int))) (int);
```

- L'appel système **signal()** permet d'installer **handler** comme fonction de traitement lors de la réception du signal **signum**. Cette fonction ne pourra prendre qu'un argument de type entier : le **numéro du signal** qui l'aura appelée. La fonction renvoie **SIG_ERR** en cas d'échec.

Réponse à un signal

Exemple

- La fonction de déroutement (handler) permet d'afficher le numéro et le nom symbolique du signal dérouté. S'il s'agit du signal SIGINT, on arrête le programme.
- La fonction **strsignal** permet de renvoyer le nom symbolique du signal numéro **sig**.

```
#include <string.h>
char *strsignal (int sig)
```

Réponse à un signal

Code

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <stdlib.h>
//fonction de déroutement
void handler (int sig) {
	printf("Bien reçu %d %s\n", sig, strsignal(sig));
	if (sig == SIGINT) {
		printf ("Fin volontaire\n");
		exit (1);
	}
}
void main () {
	signal (SIGINT, handler); //Ctrl-c : signal 2
	signal (SIGQUIT, handler); //Ctrl-\ : signal 3
	signal (SIGTSTP, handler); //Ctrl-z : signal 20
	//SIGKILL est non déroutable
	if (signal (SIGKILL, handler) == SIG_ERR) perror("SIGKILL");

	for (;;);
}
```

Réponse à un signal

Remarque

La fonction de traitement peut être remplacée par une des constantes suivantes

- **SIG_IGN**, qui indique que le signal doit être ignoré.

```
/*Rendre Ctrl-C (SIGINT) inopérant*/
/*l'arrêt de ce programme doit se faire avec kill -9 pid*/
#include <signal.h>
void main () {
  signal (SIGINT, SIG_IGN);
  for (;;);
}
```

- **SIG_DFL,** qui indique de rétablir l'action par défaut pour le signal.

Réponse à un signal

L'appel système sigaction()

La primitive sigaction() associe un signal à un contexte de déroutement, représenté par une structure **sigaction**.

```
struct sigaction{
   void (*sa_handler)(); /*SIG_DFL ou SIG_IGN ou ptr sur handler*/
   sigset_t sa_mask; /*signaux supplémentaires à bloquer*/
   int sa_flags; /*indicateurs optionnels*/
}
```

Le champ **sa_handler** est le **seul obligatoire** ; c'est un pointeur sur la fonction qui servira de handler.

```
#include <signal.h>
int sigaction (int signum, const struct sigaction* act, struct sigaction* oldact)
```

Réponse à un signal

Exemple

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <stdlib.h>
void handler (int sig) {
        printf ("Bien reçu %d %s\n", sig, strsignal(sig));
        if (sig == SIGINT) {
                printf ("Fin volontaire\n");
                exit (1);
        }
}
void main () {
        struct sigaction act;
        act.sa_handler = handler;
        sigaction (SIGINT, &act, NULL); //Ctrl-c : signal 2
        sigaction (SIGQUIT, &act, NULL); //Ctrl-\ : signal 3
        sigaction (SIGTSTP, &act, NULL); //Ctrl-z : signal 20
        //SIGKILL est non déroutable
        if (sigaction(SIGKILL, &act, NULL) == -1) perror ("SIGKILL");

        for (;;);
}
```

Réponse à un signal

La commande trap

La commande trap permet d'associer un traitement à un ou plusieurs signaux directement à partir du shell.

```
$ trap 'liste de commandes' sig1 sig2 ...
```

Exemple

La commande :

```
$ trap 'rm /tmp/* ; exit' 2 3
```

a pour effet d'exécuter la commande **rm /tmp/* ; exit** à la réception des signaux SIGINT ou SIGQUIT.

Pour désactiver un ou plusieurs signaux, il suffit de transmettre la chaîne vide comme argument de la commande trap.

Réponse à un signal

Exemple

La commande : **$ trap ” 2 3 15**

permet de masquer les signaux SIGINT, SIGQUIT et SIGTERM.
Pour rétablir l’action par défaut pour un ou plusieurs signaux, il suffit de transmettre les numéros de ces signaux comme seul argument de la commande trap.

Exemple

La commande : **$ trap 2 3 15**

permet de rétablir les actions par défaut associée aux signaux SIGINT, SIGQUIT et SIGTERM.

IX. La communication interprocessus : les tubes (pipes)

- Les tubes ou tuyaux (ou pipes en anglais) permettent à un groupe de processus d'envoyer des données à un autre groupe de processus. Ces données sont envoyées directement en mémoire sans être stockées temporairement sur disque, ce qui est donc très rapide. Tout comme un tuyau de plomberie, un tuyau de données a deux côtés : un côté permettant d'écrire des données dedans et un côté permettant de les lire. Chaque côté du tuyau est un descripteur de fichier ouvert soit en lecture soit en écriture, ce qui permet de s'en servir très facilement, au moyen des fonctions d'entrée / sortie classiques
- Le principe des tubes est assez simple : la sortie standard d'un processus est redirigée vers l'entrée d'un tube dont la sortie est dirigée vers l'entrée standard d'un autre processus. Ainsi les deux processus peuvent échanger des données sans passer par un fichier intermédiaire de l'arborescence.

IX. La communication interprocessus : les tubes (pipes)

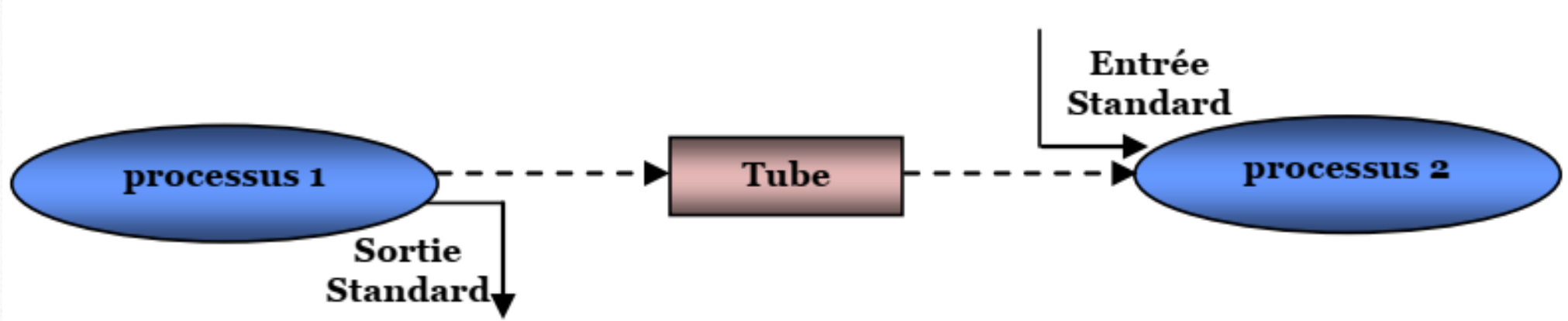


Figure 3.5 : Principe de fonctionnement d'un tube de communication interprocessus

Caractéristiques des Tubes

Un tube est un moyen de transmission de données d'un processus à un autre (Figure 3.5). Comme il est implanté par fichier, il sera désigné par des descripteurs et manipulé par les primitives **read()** et **write()**. Mais avec les particularités suivantes:

- La communication est **unidirectionnelle**: on écrit à un bout et on lit à l'autre (d'où le nom de tube). Cela entraine qu'il faut au moins deux descripteurs pour manipuler un tube.
- La communication est **faite en mode FIFO** (first in first out). Premier écrit, premier lu. Des ordres comme **Lseek()** ou autre accès directe n'ont pas de sens pour un tube.
- Ce qui est lu quitte définitivement le tube et ne peut être relu. De même, ce qui est écrit est définitivement écrit.

Caractéristiques des Tubes

- La transmission est faite en **mode flot continu d'octets**. L'envoi consécutif des deux séquence est semblable et peut être lu en totalité ou en morceaux ("abcd" et "efg" à "abcdefg").
- Pour fonctionner, un tube doit avoir au moins un lecteur et un écrivain. Il peut y en avoir plusieurs.
- Un tube a une capacité finie, et il y a une synchronisation type producteur/consommateur entre lecteurs et écrivains: un lecteur peut parfois attendre qu'il y est quelque chose d'écrite avant de lire, et un écrivain peut attendre qu'il y ait de la place dans le tube avant de pouvoir y écrire.

Utilisation d'un Tube

1) Création d'un Tube: Primitive pipe()

int pipe (int p[2])	

La primitive crée un tube et lui associe deux descripteurs rendus dans le tableau de deux entiers **p**. Par définition p[0] est le descripteur pour lire (sortie du tube) et p[1] celui pour écrire (entrée du tube). La valeur retour de **pipe()** est **0** en cas de **succès**, **-1** sinon (trop de descripteurs actifs, ou de fichiers ouverts, etc...)

Utilisation d'un Tube

Exemple:

```
int p[2];
if ( pipe(p) == -1 )
fprintf(stderr, "Impossible ouvrir tube\n");
...
```

- Comme déjà dit, la création d'un tube doit se faire avant le lancement des processus qui vont l'utiliser. Ce sera le cas par exemple dans une coopération père/fils ou fils/fils..., quand le processus père crée le tube avant de faire fork().

Remarque: Il n'y a pas la notion d'ouverture d'un tube (open()). Après sa création, un tube est directement utilisable. Par contre, la fermeture close() s'applique à un tube.

Utilisation d'un Tube

2) Lecture dans un Tube: Primitive read()

On peut lire dans un tube à l'aide de la primitive classique **read()**. On utilise le descripteur p[0] rendu par **pipe()**. Dans l'exemple qui suit :

```
char buf[100];
int p[2];
...
read(p[0], buf, 20);
```

on a une demande de lecture de 20 caractères dans le tube p. Les caractères lus sont rendus disponibles dans la zone buf. Leur nombre est la valeur retour de read().

Utilisation d'un Tube

2) Lecture dans un Tube: Primitive read()

On peut lire dans un tube à l'aide de la primitive classique **read()**. On utilise le descripteur p[0] rendu par **pipe()**. Dans l'exemple qui suit :

```
char buf[100];
int p[2];
...
read(p[0], buf, 20);
```

on a une demande de lecture de 20 caractères dans le tube p. Les caractères lus sont rendus disponibles dans la zone buf. Leur nombre est la valeur retour de read().

```
close(p[1])
read(p[0], buf, 20)
```

Cela permet d'éviter des erreurs aboutissant parfois à des situations d'interblocage (deadlock): des processus communiquent, mais chacun attend que l'autre commence.

Utilisation d'un Tube

3) Ecriture dans un Tube: Primitive write()

On peut écrire dans un tube avec la primitive classique **write()**. L'écriture est faite en utilisant le descripteur p[1] cette fois-ci. La séquence qui suit:

```
char buf[100];
int p[2];
...
close p[0];
buf = "texte a ecrire ";
write(p[1], buf, 20);
```

est une demande d'écriture de 20 caractères dans le tube de descripteur ouvert p[1]. La séquence à écrire est prise dans la zone buf. La valeur retour de **write()** est le nombre d'octets ainsi écrits. Là aussi, on a fermé le descripteur p[0] de lecture. L'écriture dans un tube est atomique (tout est écrit ou rien n'est écrit) et n'interfère pas avec d'autres écrivains éventuels. Les 20 caractères écrits ici seront consécutifs dans le tube.

Utilisation d'un Tube

3) Ecriture dans un Tube: Primitive write()

Remarque: La taille d'un tube est bien sûr limitée. Elle a pour valeur 4096 en général (constante PIPE_BUF de <**limits.h**>). S'il arrive que le nombre de caractères à écrire soit supérieur à cette limite, le message peut être écrit mais décomposé par le système en plusieurs lots. L'atomicité serait alors perdue.

Utilisation d'un Tube

3) Ecriture dans un Tube: Primitive write()

Exemple : Le père envoi au fils un message dans un tube. Message envoyé en bloc et lu caractère par caractère puis imprimé.

```
#include <stdio.h>
char message[25] = "Cela provient d'un tube";
main()
{
        /*  communication PERE --> FILS par pipe  */
        int p[2];
        int pipe(int[2]);
        if (pipe(p) == -1) {
           fprintf(stderr, "erreur ouverture pipe\n");
           exit(1);
        }
        if (fork() == 0) {          /* fils */
           char c;
           close(p[1]);
           while (read(p[0], &c, 1) != 0)
              printf("%c", c);
           close(p[0]);
           exit(0);
        } else {                    /* suite pere */
           close(p[0]);
           write(p[1], message, 24);
           close(p[1]);
           exit(0);
        }
}
```

Utilisation d'un Tube

3) Ecriture dans un Tube: Primitive write()

Remarque :

- ✓ L'arrêt de **read()** sur valeur retour nulle (réalisé par close(p[1]); dans le processus père).
- ✓ Le lancement concurrent de processus communiquant deux par deux par l'intermédiaire des tubes sera réalisé par une commande de la forme :

X. Les principales primitives de gestion de processus

1) La primitive fork ()

Un processus peut se dupliquer – et donc créer un nouveau processus – par la fonction : **pid_t fork(void)**

- Cette fonction permet la création dynamique d'un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé.
- Un appel à **fork ()** par un processus, appelé processus père, demande à UNIX de mettre en activité un nouveau processus (appelé processus-fils) qui est une copie conforme du processus courant, pour la plupart de ses attributs.

X. Les principales primitives de gestion de processus

1) La primitive fork ()

Cette fonction rend :

- **-1** en cas d'échec,
- **0** dans le processus fils,
- Le **numéro du processus** fils (PID) dans le père.

X. Les principales primitives de gestion de processus

1) La primitive fork ()

- Mis à part ce point, les deux processus sont exactement identiques et exécutent le même programme sur **deux copies distinctes** des données. Les espaces de données sont complètement séparés: la modification d'une variable dans l'un est invisible dans l'autre.
- À l'issue d'un **fork()** les deux processus s'exécutent simultanément.
- On souhaite généralement exécuter des tâches distinctes dans le processus père et le processus fils. La valeur retournée par **fork** est donc très importante pour différencier le processus père du processus fils. Un test permet d'exécuter des parties de code différentes dans le père et le fils.

X. Les principales primitives de gestion de processus

Exemple introductif

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void main() {
    pid_t p1 ;
    printf("Début de fork\n") ;
    p1 = fork() ;
    printf("Fin de fork %d\n", p1) ;
}
```

X. Les principales primitives de gestion de processus

Résultat de l'exécution :

```
$./a.out
Debut de fork
Fin de fork 16099
Fin de fork 0
```

Dans cet exemple, on voit qu'un seul processus exécute l'écriture *Début de* ***fork***, par contre, on voit deux écritures *Fin de* ***fork*** suivies de la valeur de retour de la primitive **fork().** Il y a bien apparition d'un processus : le fils, qui ne débute pas son exécution au début du programme mais à partir de la primitive **fork**.

X. Les principales primitives de gestion de processus

La génétique des processus

Un processus fils hérite de la plupart des attributs de son père, à l'exception de :

- son **PID**, qui est unique et attribué par le système d'exploitation,
- le **PID** de son père,
- ses temps d'exécution initialisés à la valeur nulle,
- les signaux en attente de traitement,
- la priorité initialisée à une valeur standard,
- les verrous sur les fichiers.

Tous les autres attributs du père sont hérités, et notamment la table des descripteurs de fichiers.

X. Les principales primitives de gestion de processus

2) La primitive exit ()

La fonction **exit** met fin au processus qui l'a émis, avec un code de retour **status**.

```
void exit (int status)
```

- Tous les descripteurs de fichiers ouverts sont fermés ainsi que tous les flots de la bibliothèque d'E/S standard.
- Si le processus à des fils lorsque **exit** est appelé, ils ne sont pas modifiés mais comme le processus père prend fin, le nom de leur processus père est changé en 1, qui est l'identifiant du processus **init**..

X. Les principales primitives de gestion de processus

2) La primitive exit ()

- Ce processus **init** est l'ancêtre de tous les processus du système excepté le processus 1 lui-même ainsi que le processus 0 chargé de l'ordonnancement des processus. En d'autres termes, c'est l'ancêtre de tous les processus qui adopte tous les orphelins.
- Par convention, un code de retour égal à zéro signifie que le processus s'est terminé correctement, et un code non nul (généralement 1) signifie qu'une erreur s'est produite. Seul l'octet de droite de l'entier **status** est remonté au processus père. On est donc limité à 256 valeurs pour communiquer un code retour à son processus père.
- Le père du processus qui effectue un **exit** reçoit son code retour à travers un appel à **wait**.

X. Les principales primitives de gestion de processus

2) La primitive exit ()

- Ce processus **init** est l'ancêtre de tous les processus du système excepté le processus 1 lui-même ainsi que le processus 0 chargé de l'ordonnancement des processus. En d'autres termes, c'est l'ancêtre de tous les processus qui adopte tous les orphelins.
- Par convention, un code de retour égal à zéro signifie que le processus s'est terminé correctement, et un code non nul (généralement 1) signifie qu'une erreur s'est produite. Seul l'octet de droite de l'entier **status** est remonté au processus père. On est donc limité à 256 valeurs pour communiquer un code retour à son processus père.
- Le père du processus qui effectue un **exit** reçoit son code retour à travers un appel à **wait**.

X. Les principales primitives de gestion de processus

3) La primitive wait ()

Lorsque le fils se termine, si son père ne l'attend pas, le fils passe à l'état **defunct** (ou **zombi**) dans la table des processus. L'élimination d'un processus terminé de la table ne peut se faire que par son père, grâce à la fonction **wait**.

```
#include <sys/types.h>
#include <sys/wait.h>
int wait (int * terminaison)
```

Avec cette instruction :

- Le père se bloque en attente de la fin d'un fils.
- Elle rendra le n° (**PID**) du premier fils mort trouvé.
- La valeur du code de sortie est reliée au paramètre **d'exit** de ce fils.

X. Les principales primitives de gestion de processus

3) La primitive wait ()

- On peut donc utiliser l'instruction **wait** pour connaître la valeur éventuelle de retour, fournie par **exit()**, d'un processus. Ce mode d'utilisation est analogue à celui d'une fonction.
- **wait()** rend –1 en cas d'erreur.
- Un processus exécutant l'appel système **wait** est endormi jusqu'à la terminaison d'un de ses fils. Lorsque cela se produit, le père est réveillé et **wait** renvoie le **PID** du fils qui vient de mourir.

X. Les principales primitives de gestion de processus

4) La primitive waitpid ()

- L'appel système **waitpid** permet de tester la terminaison d'un processus particulier, dont on connaît le **PID**.

```
#include <sys/types.h>
#include <sys/wait.h>
int waitpid (int pid, int *terminaison, int options)
```

- La primitive permet de tester, en bloquant ou non le processus appelant, la terminaison d'un processus particulier ou appartenant à un groupe de processus donné et de récupérer les informations relatives à sa terminaison à l'adresse **terminaison**.

X. Les principales primitives de gestion de processus

4) La primitive waitpid ()

Plus précisément le paramètre **pid** permet de sélectionner le processus attendu de la manière suivante :

- **<-1** tout processus fils dans le groupe | **pid** |
- **-1** tout processus fils
- **0** tout processus fils du même groupe que l'appelant
- **> 0** processus fils d'identité **pid**

X. Les principales primitives de gestion de processus

4) La primitive waitpid ()

Le paramètre options est une combinaison bit à bit des valeurs suivantes. La valeur **WNOHANG** permet au processus appelant de ne pas être bloqué si le processus demandé n'est pas terminé.

- La fonction renvoie :
  - **-1** en cas d'erreur,
  - **0** en cas d'échec (processus demandé existant mais non terminé) en mode
  - **>0** non bloquant, le numéro du processus fils zombi pris en compte.

X. Les principales primitives de gestion de processus

5) Interprétation de la valeur renvoyée

- La valeur de l'entier ***terminaison**, au retour d'un appel réussi de l'une des primitives **wait** ou **waitpid**, permet d'obtenir des informations sur la terminaison ou l'état du processus.
- De manière générale sous UNIX, pour un processus qui s'est terminé normalement c'est-à-dire par un appel **exit(n)**, l'octet de poids faible de **n** est récupéré dans le second octet de ***terminaison**, ce que symbolise le schéma ci-dessous sur une machine où mes entiers sont codés sur 32 bits.

X. Les principales primitives de gestion de processus

5) Interprétation de la valeur renvoyée

➢ **Entier n du processus fils.**

octet 3	octet 2	octet 1	octet 0

➢ **Entier *terminaison récupéré par le processus père**

? ? ? ? ? ? ? ?	? ? ? ? ? ? ? ?	octet 0	? ? ? ? ? ? ? ?

Pour un processus qui s'est terminé accidentellement à cause du signal de numéro **sig**, l'entier ***terminaison** est égal à **sig**, éventuellement augmenté de la valeur décimale 128 si une image mémoire (fichier core) du processus a été créée à sa terminaison.

X. Les principales primitives de gestion de processus

5) Interprétation de la valeur renvoyée

- L'interprétation de la valeur récupérée doit, pour des raisons de portabilité être réalisée par l'intermédiaire de macro-fonctions prédéfinies, appelées avec la valeur fournie au retour de l'appel à **wait** ou **waitpid**.

Fonction	Interprétation
WIFEXITED	Valeur non nulle si le processus s'est terminé normalement.
WEXITSTATUS	Fournit le code de retour du processus si celui-ci s'est terminé normalement.
WIFSIGNALED	Valeur non nulle si le processus s'est terminé à cause d'un signale.
WTERMSIG	Fournit le numéro du signal ayant provoqué la terminaison du processus.
…	

X. Les principales primitives de gestion de processus

6) Description de la primitive system()

La fonction **system** lance l'exécution d'un processus **SHELL** interprétant la commande passée en argument dans la chaîne de caractères **ch**.

```
int system (char *ch)
```

Cette fonction crée pour cela un nouveau processus, qui se termine avec la fin de la commande. Le processus à l'origine de l'appel de **system** est suspendu jusqu'à la fin de la commande.

X. Les principales primitives de gestion de processus

7) Principe du recouvrement

- La création d'un nouveau processus ne peut se faire que par le « recouvrement » du code d'un processus existant. Cela se fait à l'aide d'une des fonctions de la famille **exec** qui permet de faire exécuter par un processus un autre programme que celui d'origine.
- Lorsqu'un processus exécute un appel **exec**, il charge un autre programme exécutable en conservant le même environnement système :
  - ***Du point de vue du système***, il s'agit toujours du même processus : il a conservé le même **pid**.
  - ***Du point de vue de l'utilisateur***, le processus qui exécute **exec** disparaît, au profit d'un nouveau processus disposant du même environnement système et qui débute alors son exécution.

X. Les principales primitives de gestion de processus

7) Principe du recouvrement

- **execvp ().** Ces fonctions rendent **–1** en cas d'échec. Les deux fonctions de base sont:

```
int execl (char *ref, char *arg0, …, char *argn, 0)
int execv (char *ref, char *argv [ ]);
```

- Il existe d'autres fonctions analogues dont les arguments sont légèrement différents : **execle (), execlp (), execv (), execcve (),**

X. Les principales primitives de gestion de processus

8) La primitive execl

- Dans le cas de **execl**, les arguments de la commande à lancer sont fournis sous la forme d'une liste terminée par un pointeur nul :
- **ref** est une chaîne de caractères donnant le chemin absolu du nouveau programme à substituer et à exécuter.
- **arg0, arg1, …, argn** sont les arguments du programme.
- Le premier argument, **arg0**, reprend en fait le nom du programme.

X. Les principales primitives de gestion de processus

8) La primitive execl

```
void main()
{
  execl("/bin/ls", "ls", "-l", NULL);
  printf("Erreur lors de l'appel à ls \n");
}
```

Remarque:

La partie de code qui suit l'appel d'une primitive de la famille **exec** ne s'exécute pas, car le processus où elle se trouve est remplacé par un autre. Ce code ne sert donc que dans le cas où la primitive **exec** n'a pas pu lancer le processus de remplacement (le nom du programme était incorrect par exemple).

X. Les principales primitives de gestion de processus

9) La primitive execv

Dans le cas de **execv**, les arguments de la commande sont sous la forme d'un vecteur de pointeurs **(de type argv [ ])** dont chaque élément pointe sur un argument, le vecteur étant terminé par le pointeur **NULL** :

- **ref** est une chaîne de caractères donnant l'adresse du nouveau programme à substituer et à exécuter.,
- **argv [ ]** contient la liste des arguments.

X. Les principales primitives de gestion de processus

9) La primitive execv

```
#include <stdio.h>
#define NMAX 5
void main () {
    char *argv [NMAX];
    argv [0] = “ls”;
    argv [1] = “-l”;
    argv [2] = NULL;
    execv (“/bin/ls”, argv);
    printf("Erreur lors de l'appel à ls \n");
}
```

Questions ?

Exercice 2.1 (Questions de Cours)

1. Quel est le rôle de l'ordonnanceur ?

   L'ordonnanceur gère l'allocation du processeur aux différents processus.

2. Pourquoi l'algorithme d'ordonnancement SJF n'est-il pas réellement applicable ?

   car on ne peut pas prévoir la durée d'exécution d'un processus.

Exercice 2.1 (Questions de Cours)

3. Les algorithmes d'ordonnancement basés sur des priorités peuvent engendrer la famine (non-exécution) des processus à faible priorité. Comment peut-on éviter ce problème ?

On utilise un autre algorithme: tourniquet ou priorité dynamique.

Exercice 2.2 (Question à choix multiples)

4. Décrire (sur 2 à 3 phrases) la différence entre les algorithmes d'ordonnancement de programmes préemptifs et non préemptifs. Lequel qui convient mieux pour un système à temps partagé?

- L'ordonnancement de programme préemptif permet à un processus d'être interrompu au milieu de son exécution, en lui retirant la cpu et l'allouant à un autre processus.
- L'ordonnancement non préemptif s'assure qu'un processus lâche l'unité centrale de traitement seulement quand il termine avec son exécution en cours.
- C'est les algorithmes préemptifs (ou bien round robin) qui conviennent au système à temps partagé.

Exercice 2.2 (Question à choix multiples)

5. Supposons qu'on a une machine avec 4 processeurs et 3 processus qui sont en état « prêt ». Quelle est d'après vous la politique d'ordonnancement de processus qui donnerait les meilleures performances pour le système ? Argumentez votre réponse.

   Etant donné qu'il y a plus de processeurs que de processus, n'importe quelle stratégie d'ordonnancement qui affecte entièrement un processeur à un processus donne de bons résultats. Il y a plus de ressources que de demandes.

Exercice 2.2 (Question à choix multiples)

6. Décrivez ce qui se passe, du côté du système d'exploitation, lorsqu'une touche de clavier est pressée ? (maximum 3 phrases)

   Après chaque touche pressée, une interruption (de type matérielle associée au clavier) est générée.

   Le processeur interrompt son traitement pour lancer la routine d'interruption associée.

Exercice 2.2 (Question à choix multiples)

7. La stratégie d'ordonnancement de processus la plus appropriée pour un système d'exploitation es temps partagé est :

   a) **Le Court-travail Premier (SJF).**
   b) **Priorité.**
   c) **Rond-Robin.**
   d) **Premier arrivée Premier Servi (FCFS).**
   e) **tout ce qui précède.**

Exercice 2.2 (Question à choix multiples)

8. Un processus était observé de commuter depuis l'état actif vers l'état prêt. L'ordonnancement (ou le scheduling) doit être :

   a) **Le plus court job le premier (SJF)**
   b) **Non-préemptive.**
   c) **Préemptive**
   d) **Round Robin**
   e) **Aucune de ce qui précède**

Exercice 2.2 (Question à choix multiples)

9. La différence entre l'ordonnancement de programmes avec préemption et sans préemption est :
   a) **Si ou non un processus prêt peut être involontairement terminé depuis l'état prêt.**
   b) **Si ou non un processus élu est involontairement enlevé de l'état actif.**
   c) **Si ou non un processus bloqué peut avoir ses ressources involontairement lui retirées.**

Exercice 2.2 (Question à choix multiples)

10. Quel est l'intérêt du scheduling multi niveaux ?.

    Dans le scheduling multi-niveaux, la file d'attente des processus prêts n'est pas unique : elle est divisées en plusieurs files devant contenir chacune un type de processus donné. L'intérêt de cette méthode est que les processus (du système et des utilisateurs, par exemple) n'ont pas les mêmes besoins (mémoire et temps processeur) et doivent donc être ordonnancés (Schedule) différemment.