





B. Gestion des exécutions programmes

processus, ordonnancement, threads

Sommaire

-  B1. NOTION DE PROCESSUS
-  B2. ORDONNANCEMENT SUR L'UNITÉ CENTRALE
-  *B3. SYNCHRONISATION ET COMMUNICATION ENTRE PROCESSUS*
-  B.4. COMPLEMENT : NOTION DE PROCESSUS LÉGER OU THREAD

Introduction

Dans ce **chapitre**, nous nous **intéressons** à la **fonction d'exécution** qui recouvre principalement **deux notions** :

- celle de **processus** qui correspond à l'**image** d'un **programme** qui **s'exécute**
- et celle d'**ordonnancement** qui correspond au **problème** de l'**allocation** du **processeur** et donc du **partage** du **processeur** entre **différents processus**.

Enfin, nous terminons cette partie en abordant les problèmes de synchronisation et de communication entre processus.

NOTION DE PROCESSUS

Définitions

Démo : diapos4

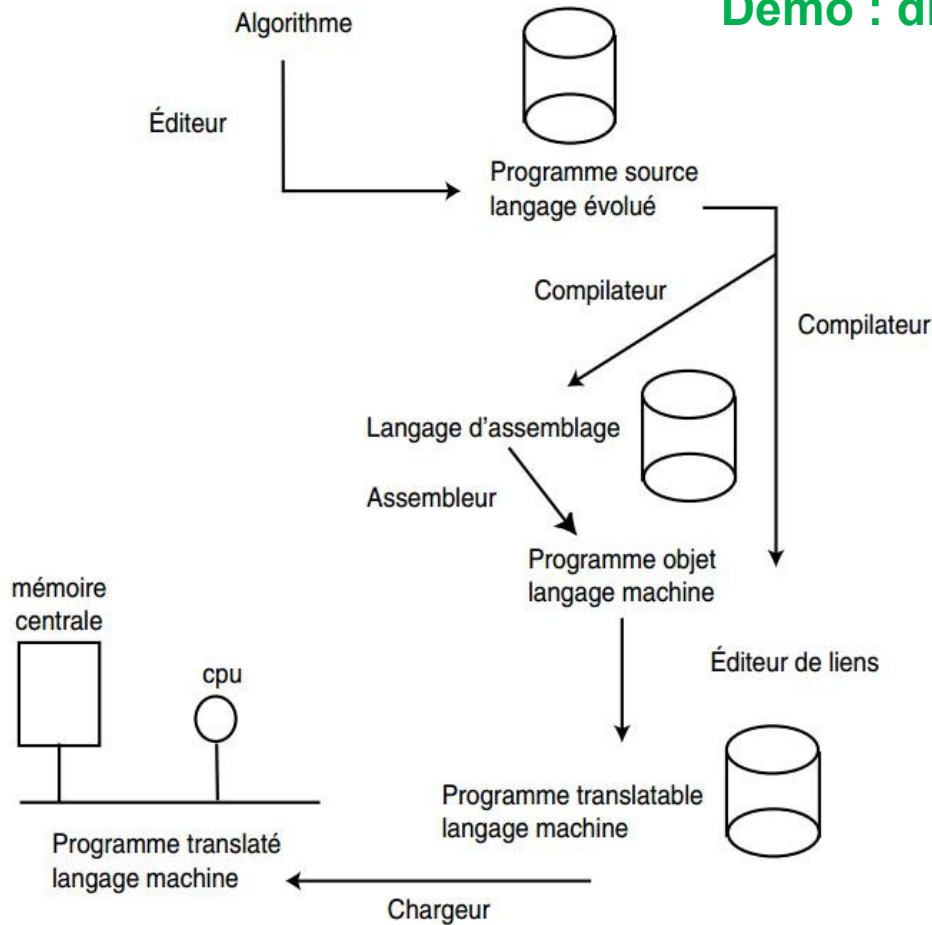


Figure 1 : Du programme au processus (note 1)

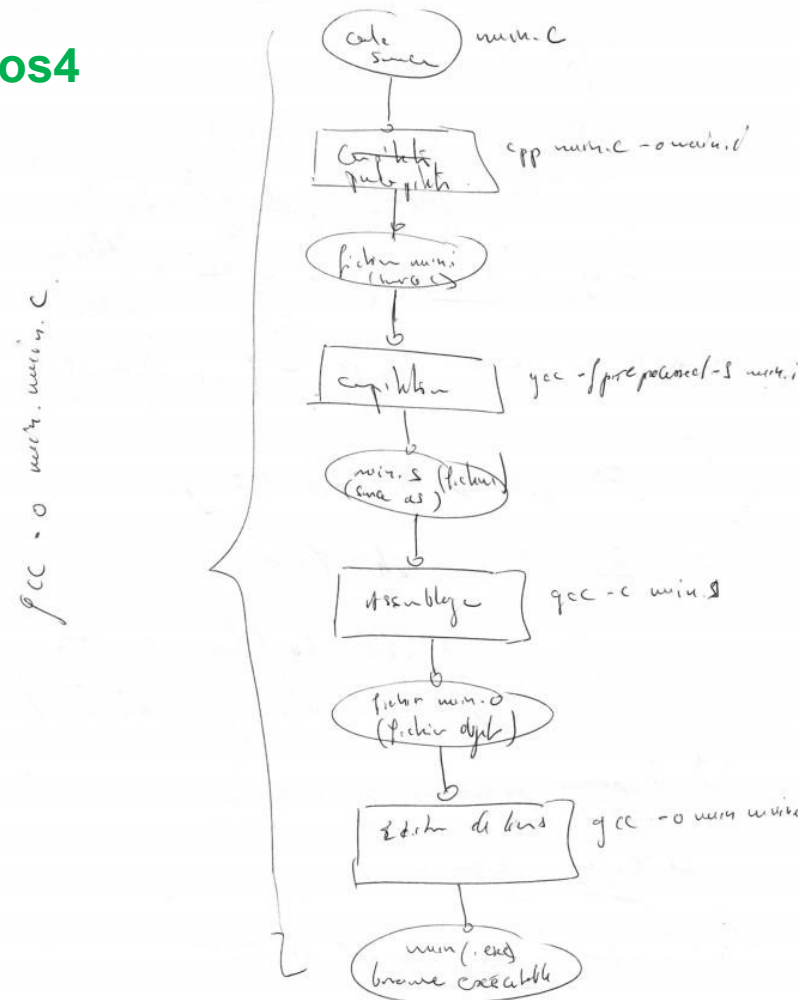


Figure 2 : Chaîne de production de programmes en C.

NOTION DE PROCESSUS

Définitions

- Le **programme à exécuter** est **placé en mémoire centrale** à partir de l'emplacement **d'adresse 102**.
- Le processeur commence **l'exécution du programme** :
 - la **première instruction** `loadIm R1 20` de celui-ci est chargée dans le registre instruction1 (**RI**)
 - et le compteur ordinal (**CO**) **contient l'adresse** de la **prochaine instruction** à exécuter soit **103**.
 - Lorsque **l'instruction courante** a été **exécutée**, le **processeur charge** dans le registre **RI** l'instruction pointée par le **CO**, soit par exemple l'instruction `add Im R1 5` et le **compteur ordinal** prend la valeur **104**.
 - L'exécution de l'instruction `add Im R1 5` **modifie** le contenu du **registre PSW** puisque c'est une instruction arithmétique : les drapeaux de signe, de nullité etc. sont mis à jour.
- A **chaque étape** d'exécution du programme, le **contenu des registres du processeur évolue**.
- De même le **contenu de la mémoire centrale** peut être **modifié** par des opérations d'écriture ou de lecture.
- On appelle **processus** l'**image** de l'**état** du **processeur** et de la **mémoire** au cours de **l'exécution** d'un **programme** :
 - Le programme est statique
 - et le processus représente la dynamique de son exécution.

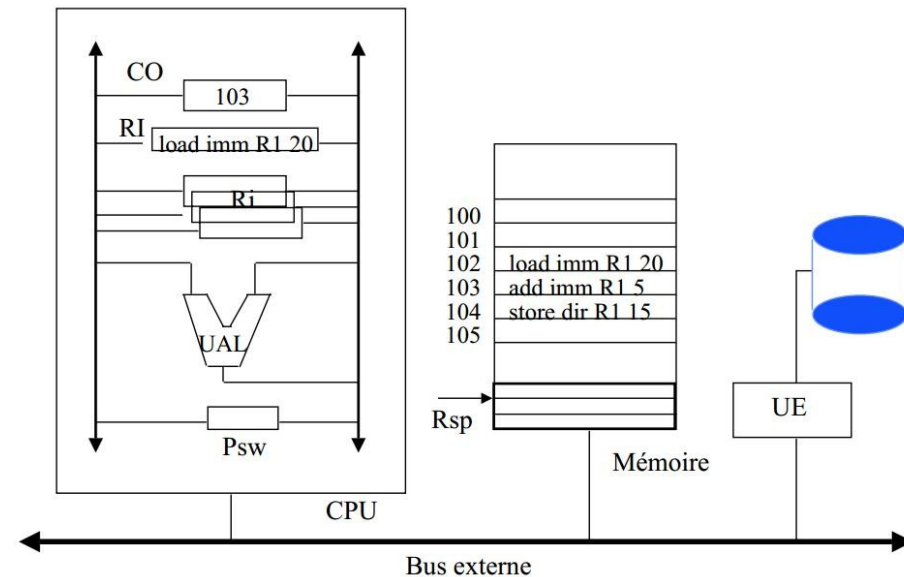


Figure 3 : Exécution d'un programme

NOTION DE PROCESSUS

Définitions

- Un **processus** est un **programme** en cours **d'exécution** auquel est **associé** un **environnement processeur** (CO, PSW, RSP, registres généraux) et un environnement mémoire (zone de code, de données et de pile) **appelés contexte** du **processus**;
- un processus est l'**instance dynamique** d'un **programme** et incarne le fil **d'exécution** de celui-ci dans un **espace d'adressage protégé**, c'est-à-dire sur une plage mémoire dont l'accès lui est réservé.
- un **programme réentrant** est un programme pour lequel il **peut exister plusieurs** instances d'exécutions simultanées (**processus**).
- Une **ressource** désigne toute **entité** dont a **besoin** un **processus** pour **s'exécuter** :
 - Ressource **matérielle** (processeur, périphérique)
 - Ressource **logicielle** (variable)
- Une ressource est **caractérisé** :
 - par un **état** : libre / occupée
 - par son **nombre de points d'accès** (nombre de processus pouvant l'utiliser en même temps)

NOTION DE PROCESSUS

États d'un processus

- Lors de son **exécution**, un **processus** est caractérisé par un **état** :
 - L'état **élu** est l'état **d'exécution** du **processus** ;
 - L'état **bloqué** est l'état **d'attente** d'une **ressource** autre que le processeur ;
 - L'état **prêt** est l'état **d'attente** du **processeur**.

(note 2)

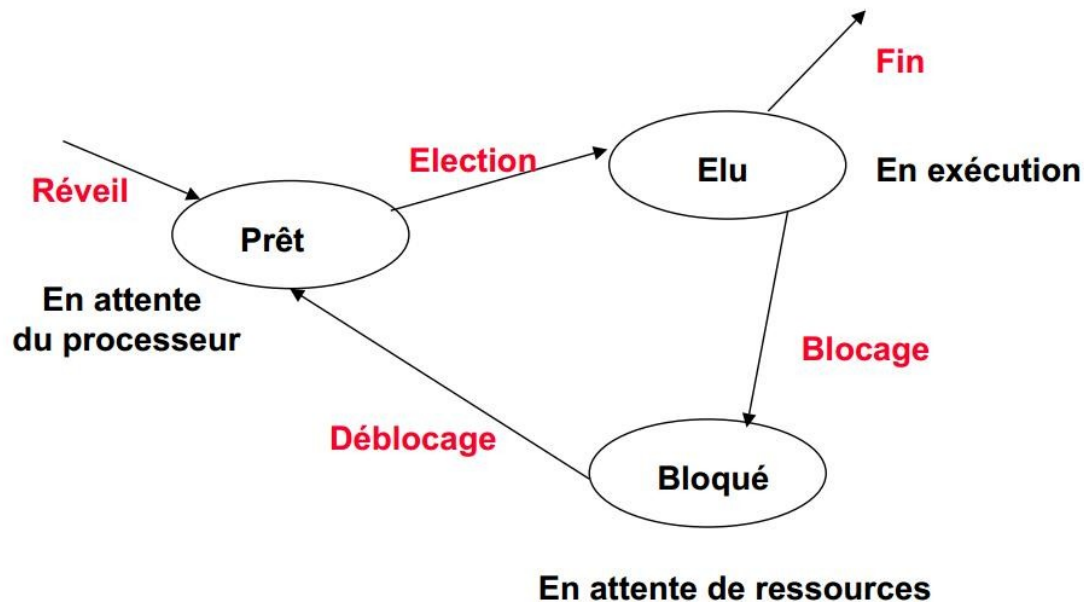


Figure 4 : Diagramme d'états d'un processus.

- Un **processus** est toujours **créé** dans l'état **prêt**. Un processus se **termine** toujours à **partir** de l'état élu (sauf anomalie).

NOTION DE PROCESSUS

Bloc de contrôle du processus
(*PCB : Process Control Block*)

- En même temps que le **chargement** du programme en MC, le **système d'exploitation crée une structure de description du processus associé au programme exécutable, le PCB. (note 3)**
- Le PCB **permet la sauvegarde et la restauration du contexte mémoire et du contexte processeur** lors des opérations de **commutations de contexte**.

Identificateur processus
État du processus
Compteur ordinal contexte pour reprise (registres et pointeurs, piles...)
Chaînage selon les files de l'ordonnanceur priorité (ordonnancement)
Informations mémoire (limites et tables pages / segments)
Informations sur les ressources utilisées fichiers ouverts, outils de synchronisation, entrées-sorties
Informations de comptabilisation

Figure 5 : Bloc de contrôle de processus.

NOTION DE PROCESSUS

Opérations sur les processus

- Le **système d'exploitation** offre généralement les **opérations** suivantes pour la **gestion** des **processus** : création de processus, destruction de processus, suspension de l'exécution et reprise de celle-ci :
 - **Création de processus** : Un **processus** peut **créer** un ou plusieurs **autres processus** en invoquant un **appel système** de création de processus (**fork**, sous Linux, **note 4**), se développe un **arbre** de **filiation** entre processus (*père -> fils -> petit-fils ->*).
 - **Destruction de processus** intervient :
 - **processus** a **terminé** son exécution, il **s'autodétruit** en appelant une **routine système** de fin d'exécution (par exemple **exit()** sous Unix);
 - processus **commet** une **erreur irrécouvrable**. Une **trappe** est **levée** et le processus est **terminé** par le **système**.
 - lorsqu'un **autre processus** **demande** la **destruction** du processus, **par** le biais d'un **appel** à une routine **système** telle que **kill** sous Unix.
 - Lors de la **destruction** d'un processus, le **contexte** de celui-ci est **démantelé** : les **ressources** allouées au processus sont **libérées** et son **bloc de contrôle** est **détruit**.
 - **Suspension d'exécution** :
 - opération qui **consiste** à momentanément **arrêter l'exécution** d'un **processus** pour la **reprendre ultérieurement**.
 - le **contexte** du **processus** est **sauvegardé** dans son **PCB** afin de pouvoir **reprendre l'exécution**, là où elle a été suspendue.
 - Le processus suspendu **passé** dans l'état **bloqué**.
 - Lors de sa **reprise** d'exécution, le processus franchit la transition de **déblocage** et entre dans l'état **prêt**.
 - Sous les systèmes Linux ou Unix, l'**appel système** **sleep(duree)** permet de **suspendre** l'exécution d'un **processus** pour un temps égal à **duree secondes**.

NOTION DE PROCESSUS

Un exemple de processus : les processus Unix

- Le système **Unix** est entièrement **construit** à partir de la notion de **processus**.
- Au **démarrage** du système, un **premier** processus est créé, le **processus 0**. Ce processus 0 **crée à son tour** un **autre processus**, le **processus 1** encore appelé **processus init**.
- Ce processus **init** **lit** le fichier **/etc/inittab** et **crée** chacun des **deux types** de **processus** décrits dans ce fichier.

Notes :

- systemd** est un **gestionnaire** de **système** et de **services** pour **Linux**. Il est le **système d'init** par défaut dans Debian depuis Debian 8 (« Jessie »).
- Systemd est **compatible** avec les scripts d'init SysV et LSB. Il peut fonctionner comme un remplaçant de **sysvinit**.

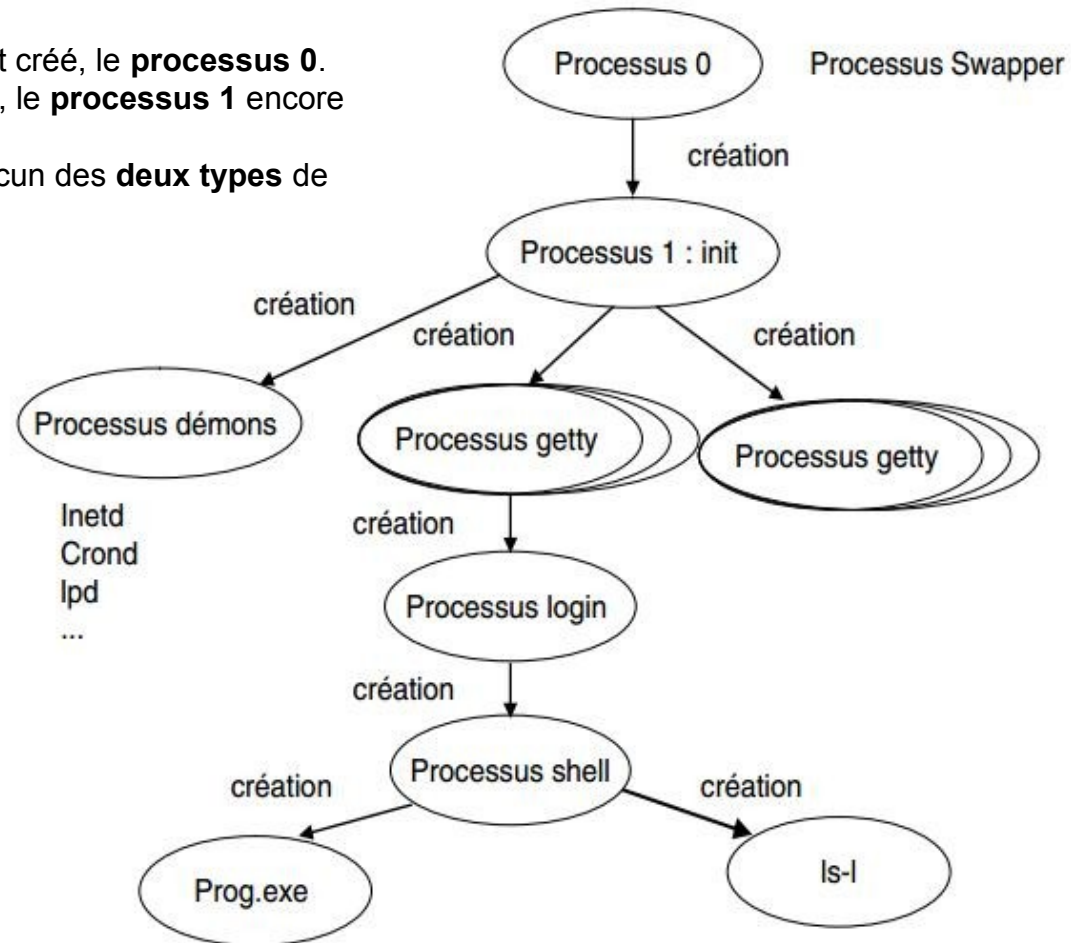


Figure 6 : Arborescence des processus Unix.

NOTION DE PROCESSUS

Un exemple de processus : les processus Unix

- Un **processus** Unix est **décrit** par un **bloc de contrôle** qui est **divisé en deux parties** :
 - chaque processus **dispose** d'une **entrée** dans une table générale du système, la **table des processus**, **contient** les **informations** sur le processus **utiles** au **système** : **pid**, **l'état** du processus, les **informations d'ordonnancement**, les **informations mémoire**.
 - chaque processus **dispose** également d'une autre structure, la **Zone U**, **contient** d'autres **informations** concernant le processus qui **peuvent** être temporairement **déplacées** sur le **disque**, quand le **processus** est dans l'état **bloqué** depuis un **certain temps**.

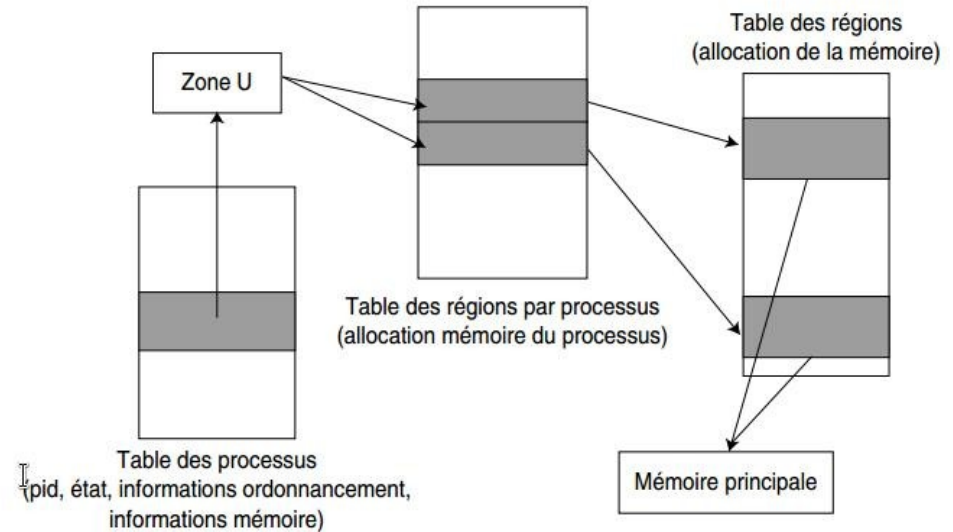


Figure 7 : Bloc de contrôle d'un processus Unix.

NOTION DE PROCESSUS

Un exemple de processus : les processus Unix

- Un **processus** Unix évolue entre **trois modes** au cours de son **exécution** :
 - le **mode utilisateur** : mode **classique** d'exécution
 - le **mode noyau en mémoire**, le **processus** se trouve dans un des états suivants, passé en **mode superviseur** :
 - processus **élu** ,
 - processus **prêt**,
 - ou processus **bloqué** (endormi).
 - et le **mode swappé** qui est le mode dans lequel se trouve un **processus bloqué** (endormi) **déchargé** de la **mémoire** centrale. Ces processus **réintègrent** la **mémoire** centrale lorsqu'ils **redeviennent prêts** (transition de l'état prêt swappé vers l'état prêt).

- Un **processus** Unix qui se **termine** passe dans un **état** dit **zombi**. Il y reste tant que son **PCB** n'est **pas entièrement démantelé** par le système.

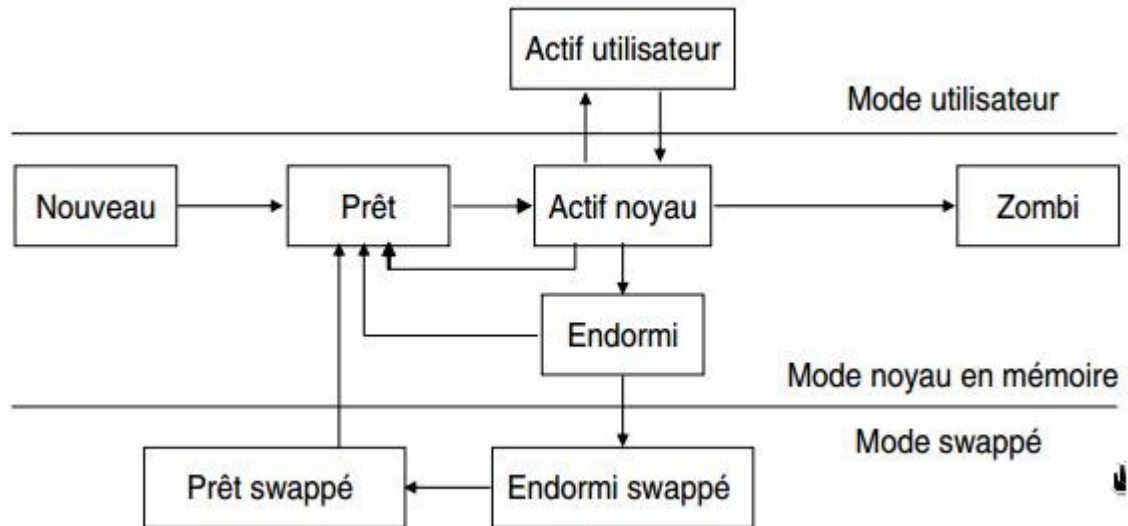


Figure 8 : Diagramme d'états simplifié d'un processus Unix

Démo : [diapos12/exemple-status.c](#)

NOTION DE PROCESSUS

Programmation de processus : l'exemple de LINUX

Principaux attributs

- Chaque **processus Linux** est **caractérisé** par un numéro unique appelé **PID** (**entier non signé de 32 bits**) qui lui est **attribué** par le système au moment de sa **création**.
- Par ailleurs, chaque **processus Linux** pouvant **créer** lui-même un **autre processus**, chaque **processus** est également **caractérisé** par l'**identifiant** du processus qui l'a créé (**son père**), appelé **PPID**.
- Les **primitives** données ci-dessous **permettent** à un **processus** respectivement de **connaître** la valeur de **son PID**, du **PID de son père** :

```
#include <unistd.h>
pid_t getpid(void); retourne le PID du processus appelant.
pid_t getppid(void); retourne le PPID du processus appelant.
Exemple : pid_t ret ; - déclaration variable ret de type pid_t
ret = getpid() ;
```

Création d'un processus Linux

- Sous le système Linux, tout **processus** peut **créer** un **nouveau processus** qui est une **exacte copie** de lui-même.
- Le **processus créateur** (le **père**) par un **appel** à la **primitive `fork()`** crée un **processus fils** qui est une **copie exacte** de lui-même (code et données).
- Le **prototype** de la fonction est le suivant:

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

- Le **code retour** du **fork** qui est différent chez le **fils** (toujours **0**) et le **père** (**PID du fils créé**).

NOTION DE PROCESSUS

Programmation de processus : l'exemple de LINUX

Création d'un processus Linux : Exemple (note 5)

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

int main()
{
    pid_t ret;
    ret = fork();
    if (ret == 0)
    {
        printf("je suis le fils; mon pid est %d\n", getpid());
        printf("pid de mon père, %d\n", getppid());
        return 0;
    }
    else
    {
        printf("je suis le père; mon pid est %d\n", getpid());
        printf("pid de mon fils, %d\n", ret);
        wait(NULL); // Pour attendre la fin d'exécution du fils.
        return 0;
    }
}
```

NOTION DE PROCESSUS

Programmation de processus : l'exemple de LINUX

Terminaison d'un processus Linux

- Un **processus termine** normalement son **exécution** en **achevant l'exécution** du **code** qui lui est associé. Cette **terminaison s'effectue** par le biais d'un **appel** à la **primitive** `exit()`.
- Le **prototype** de la fonction `exit()` est le suivant:

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

- **status** est un code retour **compris** entre **0** et **255** qui est **transmis** au **père** par le **processus défunt**, **0** caractérise une **terminaison normale** du processus et une valeur supérieure à 0 code une fin **anormale**.
- Lors de la **terminaison** d'un **processus**, le **système désalloue** les **ressources**, mais ne **détruit pas** le **bloc de contrôle**.
- Le processus **passse** à la valeur `TASK_ZOMBIE` puis **avertit** le processus **père** de la **terminaison** de son **fils**.
- Un processus **fils** défunt **reste zombie jusqu'à** ce que son **père** ait pris **connaissance** de sa **mort**.
- Un **processus** fils **orphelin**, suite au décès de son père (le processus **père** s'est **terminé avant** son **fils**) est toujours **adopté** par le **processus** numéro 1 (Init). **(faire démo : exemple-zombie-1.c et exemple-zombie-2.c)**

Synchronisation avec le père

- Un **processus père** peut **se mettre** en **attente** de la **mort** de l'un de ses **fils**, par le **biais** de la **primitive** `wait()`.
- Lorsque le processus père prend connaissance de la **mort** de l'un de ses **fils**, il **détruit** le **bloc de contrôle** du processus **fils**.
- Le **prototype** de la fonction `wait()` est le suivant:

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

- La **fonction retourne** immédiatement le **PID** du **fils terminé** et le **code retour** de celui-ci **dans** la variable **status**.
- L'**exécution** du **processus père** est **suspendue** jusqu'à ce qu'un **processus fils** se **termine**.

NOTION DE PROCESSUS

Programmation de processus : l'exemple de LINUX

Terminaison du processus fils et affichage de sa valeur de retour : Exemple

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    pid_t ret, fils_mort;
    int status;
    ret = fork();
    if (ret == 0)
    {
        printf("je suis le fils; mon pid est %d\n", getpid());
        printf("pid de mon père, %d\n", getppid());
        exit(0);
    }
    else
    {
        printf("je suis le père; mon pid est %d\n", getpid());
        printf("pid de mon fils, %d\n", ret);
        fils_mort = wait(&status);
        printf("je suis le père; le pid de mon fils mort est %d\n",
               fils_mort);
        if (WIFEXITED(status))
            printf("je suis le père; le code retour de mon fils est %d\n",
                   WEXITSTATUS(status)); // vrai si le processus fils s'est terminé par un
appel à la primitive exit() ;
    }
}
```


NOTION DE PROCESSUS

Langage de commandes Processus : l'exemple de Linux

- Depuis son **terminal**, un utilisateur dispose de **commandes** lui permettant de **visualiser** et gérer ses **processus**.

Commande ps : connaître les processus existants

- Permet d'obtenir des **informations** pour l'**ensemble** des **processus** en cours **d'exécution**.

```
(base) └─(komo ☿ kali)-[~]  
└─$ ps -lu komo
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	1000	1710	1	0	80	0	-	4825	do_epo	?	00:00:00	systemd
5	S	1000	1711	1710	0	80	0	-	42331	-	?	00:00:00	(sd-pam)
0	S	1000	1726	1710	0	69	-11	-	20220	do_epo	?	00:08:28	pipewire

- Affiche les **processus** appartenant à l'utilisateur spécifié (ici **komo**).
- Dans les **traces** ci-dessus, les **champs** affichés **correspondent** aux informations suivantes (**note 8**) :
 - utilisateur (UID)** : indique le **nom** ou l'**identifiant** de l'utilisateur ayant lancé le **programme** ;
 - pid (Processus Identfier)** : correspond à l'**entier attribué** par le **système** à un **processus** pour l'**identifier** ;
 - ppid (Parent Processus Identifier)** : correspond à l'**identifiant** du **processus parent** ayant **engendré** le **processus** ;
 - état (S, STA ou STATE)** : **état défini** par l'**ordonnanceur** du système, plusieurs valeurs sont possibles :
 - S** pour **Stopped** si le **processus** est en sommeil (état **endormi** ou **bloqué**) ;
 - R** pour **Running** si le **processus** est en **exécution** ou **prêt** à s'exécuter ;
 - Z processus** dans l'état **zombie**.
- Plus généralement, la commande **ps** **aux** permet d'**afficher toutes** les **informations** associées à **tous** les **processus** en cours **d'exécution** dans le système.

NOTION DE PROCESSUS

Langage de commandes Processus : l'exemple de Linux

Commande kill : arrêter l'exécution d'un processus

- Permet d'envoyer un **signal** à un **processus**.
- Un signal est un moyen de **communication entre processus**.
- Chaque **signal** est **identifié** par un **nom** et un **numéro**.

```
kill -numerosignal pid
```

- **envoie** le signal « **numerosignal** » au **processus** de numéro « **pid** ».
- Lors de la **prise en compte** de l'arrivée d'un **signal**, un **processus** va **exécuter** un **traitement** par défaut associé à ce signal.
- Un **processus** peut **attacher** un **signal** une **fonction** qu'il souhaite voir exécutée : le **signal** est dit **capté**.
- **Deux signaux** permettent à l'utilisateur de **forcer l'arrêt** d'un **processus**. Ce sont les signaux :
 - **SIGTERM** de numéro **15**. Ce signal **demande l'arrêt d'un processus**. Il peut être **capté**.
 - **SIGKILL** de numéro **9**. Ce signal **force le processus à se terminer** ; il ne peut **pas** être **capté**.

```
(base) └─(komo ☉ kali)-[~]  
└─$ cat >  essai  
sleep 100
```

```
(base) └─(komo ☉ kali)-[~]  
└─$ kill 841105
```

```
(base) └─(komo ☉ kali)-[~]  
└─$ ./essai&  
[1] 841103
```

```
Terminated  
[1] + exit 143    ./essai  
(base) └─(komo ☉ kali)-[~]  
└─$ ps  
      PID TTY          TIME CMD  
  288073 pts/1        00:00:01 zsh  
  841246 pts/1        00:00:00 ps
```

```
(base) └─(komo ☉ kali)-[~]  
└─$ ps
```

Les traces ci-dessus montre l'action de la commande **kill**.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Introduction

- 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**.

Système multiprocessus

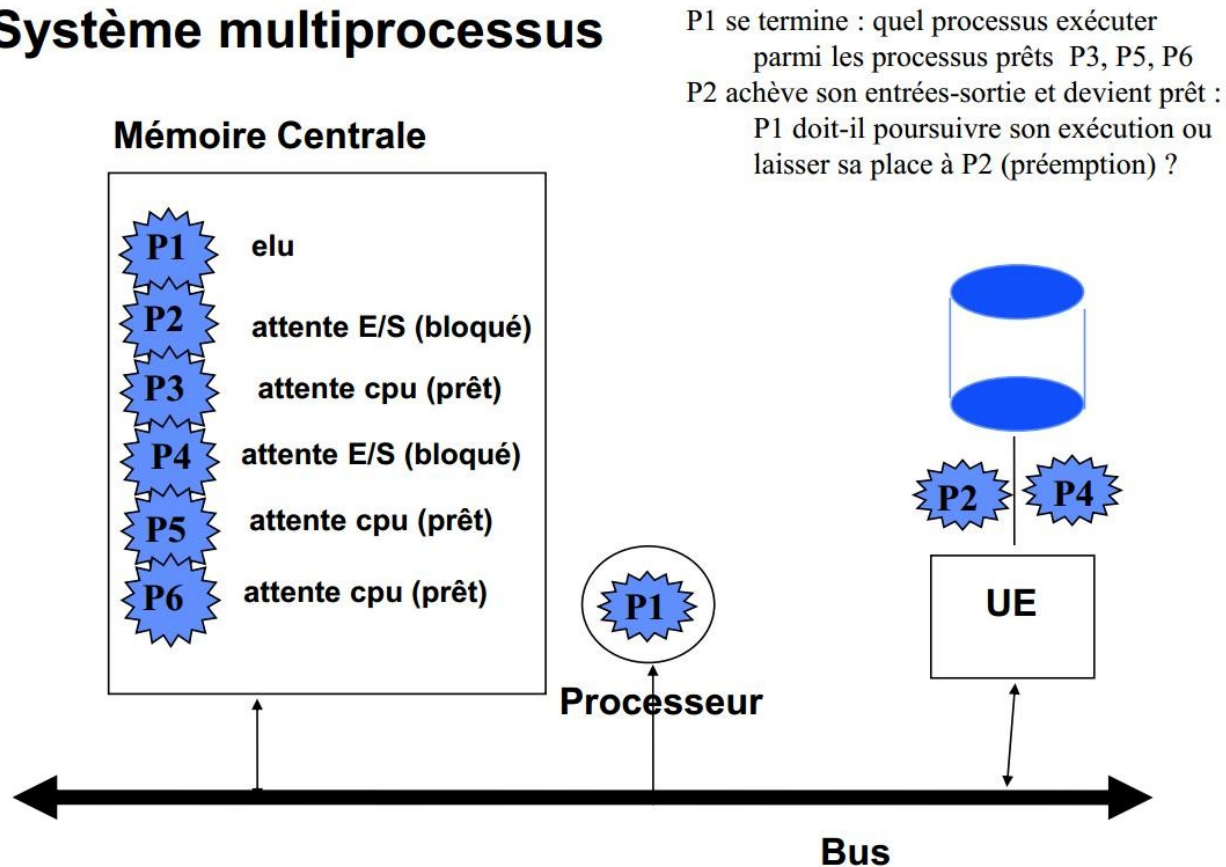


Figure 9 : Exemple d'ordonnancement de processus par le système (note 9).

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Ordonnancement préemptif et non préemptif

- Le **passage** de l'état élu vers l'état **prêt** a été **ajouté** par **rapport** à la figure 4 (**diapos 7**) : il correspond à une **réquisition** du **processeur**, c'est-à-dire que le **processeur** est **retiré** au **processus élu** alors que celui-ci dispose de toutes les ressources nécessaires à la poursuite de son exécution. Cette réquisition porte le nom de **préemption**.
- Ainsi selon **si l'opération de réquisition est autorisée** ou **non**, l'**ordonnancement** est **qualifié** d'ordonnancement **préemptif** ou **non préemptif** :
 - si l'ordonnancement est **non préemptif**, la **transition** de l'état élu vers l'état prêt est **interdite** : un processus quitte le processeur s'il a terminé son exécution ou s'il se bloque ;
 - si l'ordonnancement est **préemptif**, la **transition** de l'état élu vers l'état prêt est **autorisée** : un **processus** quitte le processeur s'il a terminé son exécution, s'il se bloque ou si le processeur est réquisitionné.

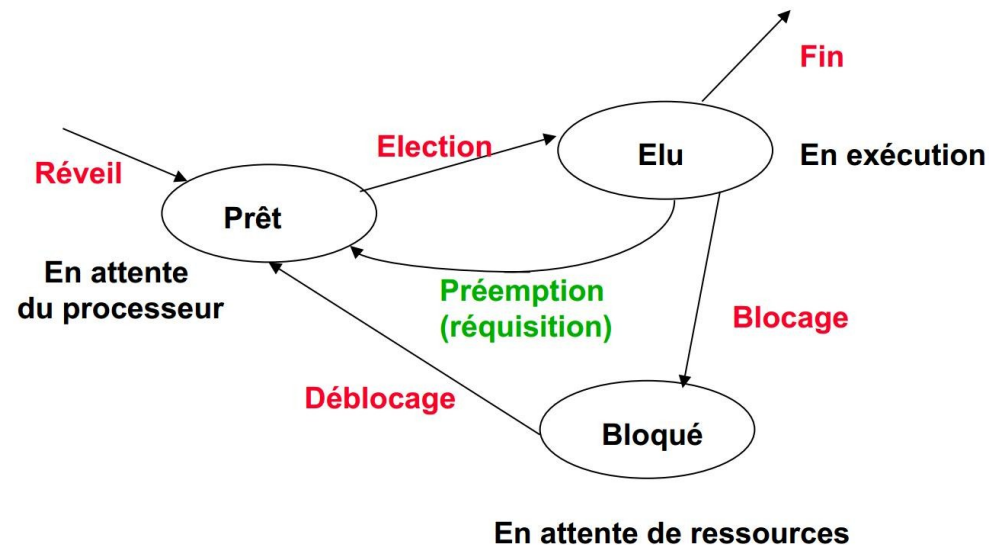


Figure 10 : Opérations d'élection et de préemption.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Ordonnancement préemptif et non préemptif

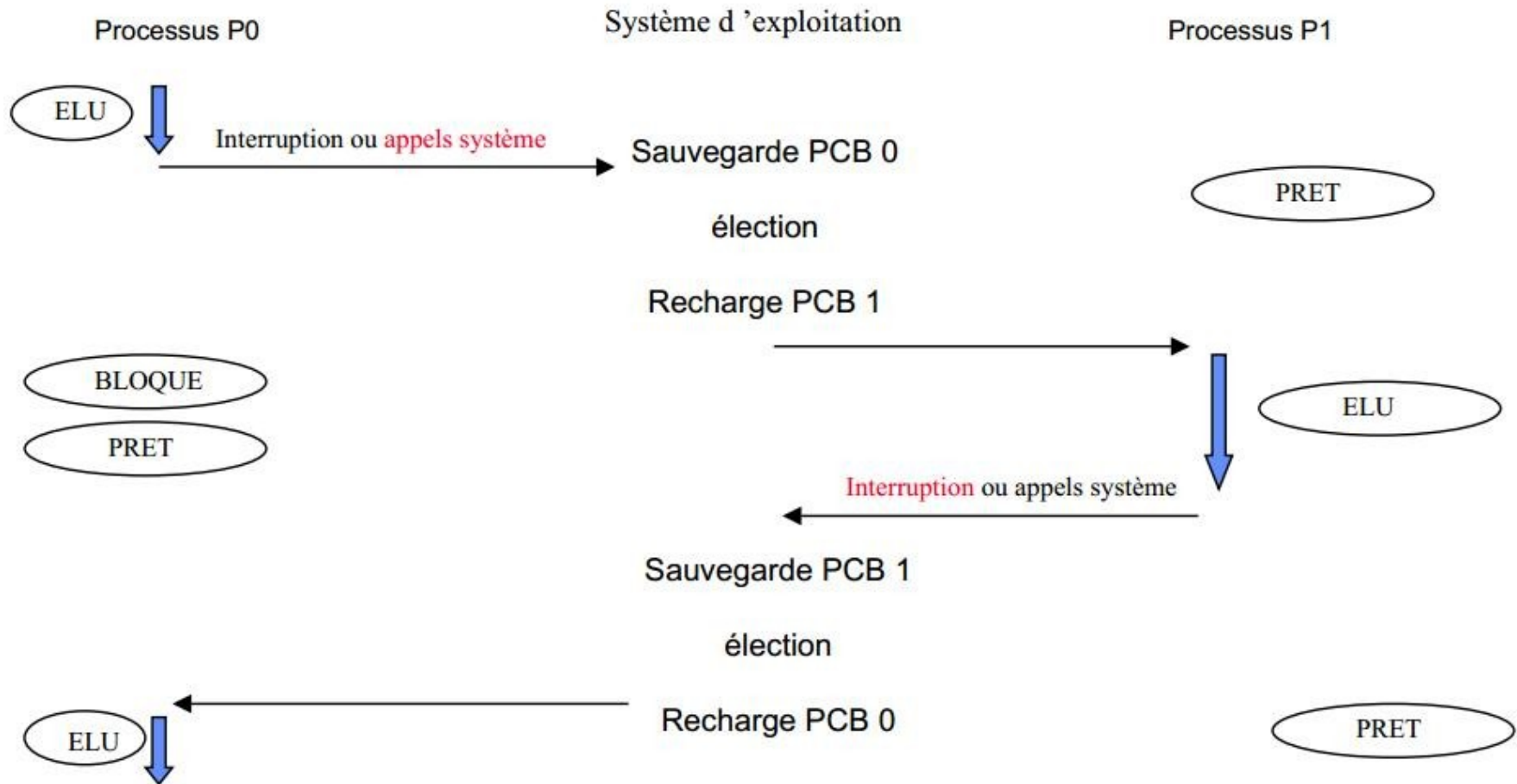


Figure 11 : Déroulement des opérations d'ordonnancement. (note 10)

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Entités systèmes responsable de l'ordonnancement

- Les **processus prêts** et les **processus bloqués** sont **gérés** dans **deux files d'attente distinctes** qui chaînent leur PCB.
- Le module **ordonnanceur** (scheduler) **trie** la **file** des processus **prêts** de telle sorte que le **prochain processus** à élire soit toujours en **tête de file**.
- Le **tri s'appuie** sur un critère donné spécifié par la **politique d'ordonnancement**.

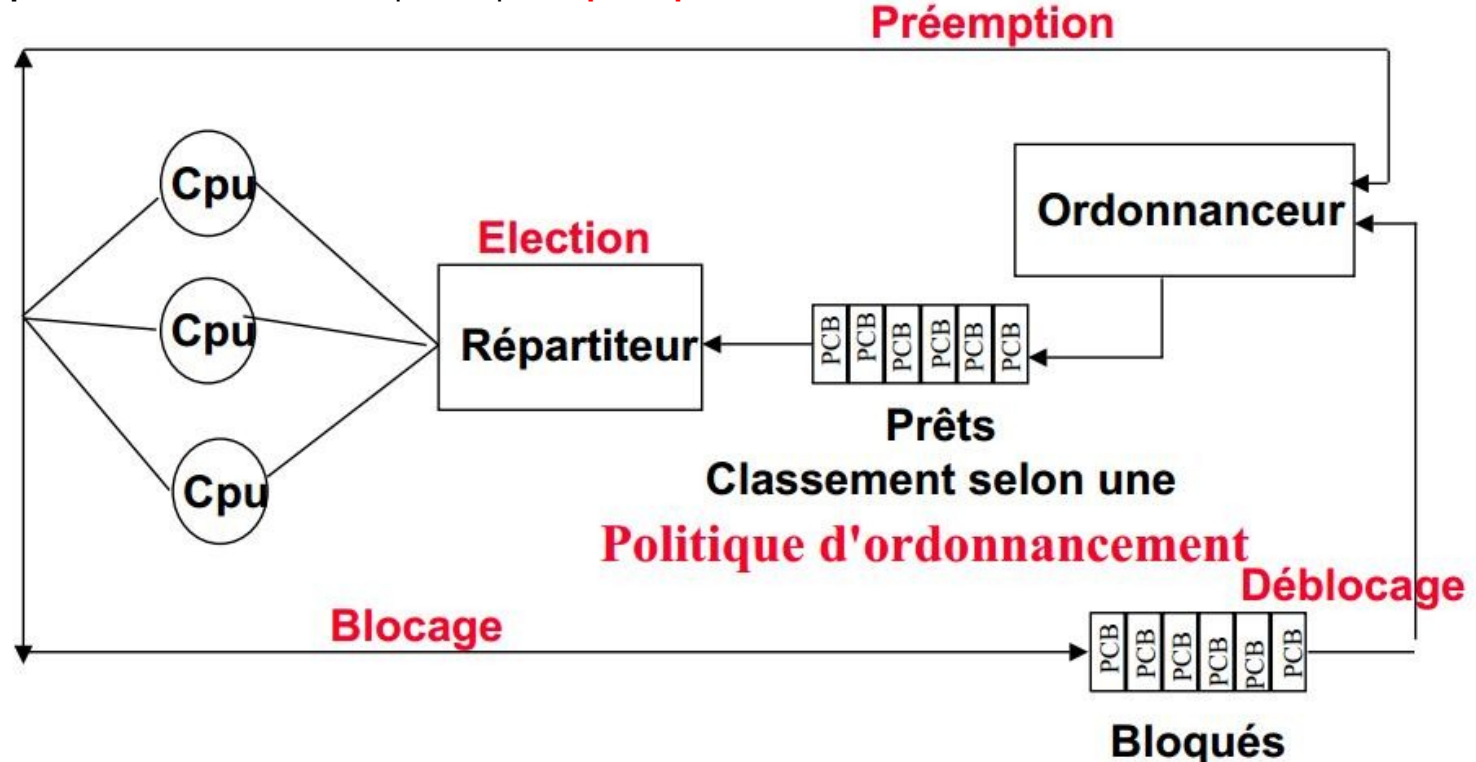


Figure 12 : Ordonnanceur et répartiteur. (note 11)

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Politiques d'ordonnancement

- La **politique d'ordonnancement détermine** quel sera le **prochain processus élu**. Selon si la **préemption est autorisée** ou non, la **politique d'ordonnancement** sera de **type préemptive** ou **non**.

Objectifs des politiques

- Les **objectifs à atteindre** en matière **d'ordonnancement diffèrent** selon les **types de systèmes** considérés.
- Une **politique** se révélera plus **appropriée** à un certain **type de système** plutôt qu'à un **autre** :
 - pour un système à **traitements par lots** : **maximiser** le **débit** du **processeur** ou capacité de traitement,
 - pour un système en **temps partagé** : **maximiser** le taux **d'occupation** du **processeur** tout en **minimisant** le **temps de réponse** des **processus**.
 - pour un système **temps réel** : le but recherché est de **respecter** les **contraintes temporelles** des processus.
- Différents **critères** sont utilisés pour **mesurer** les **performances** des **politiques d'ordonnancement** (note 12):
 - le taux d'occupation du processeur ;
 - la capacité de traitement du processeur ;
 - le temps d'attente des processus ;
 - le temps de réponse des processus.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Politiques d'ordonnancement

Présentation des politiques

- Nous **présentons** à présent les **politiques d'ordonnancement** les plus **courantes**.

A. Politique Premier Arrivé, Premier Servi

- Les **processus** sont **élus** selon l'**ordre** dans lequel ils **arrivent** dans la **file d'attente** des **processus** prêts.
- Il n'y a **pas** de **réquisition**.
- L'**avantage** de cette politique est sa **simplicité**.
- Inconvénient** : les **processus** de **petit temps d'exécution** sont **pénalisés** en terme de **temps de réponse** par les **processus** de **grand temps d'exécution** qui se **trouvent** **avant** eux dans la **file d'attente**.

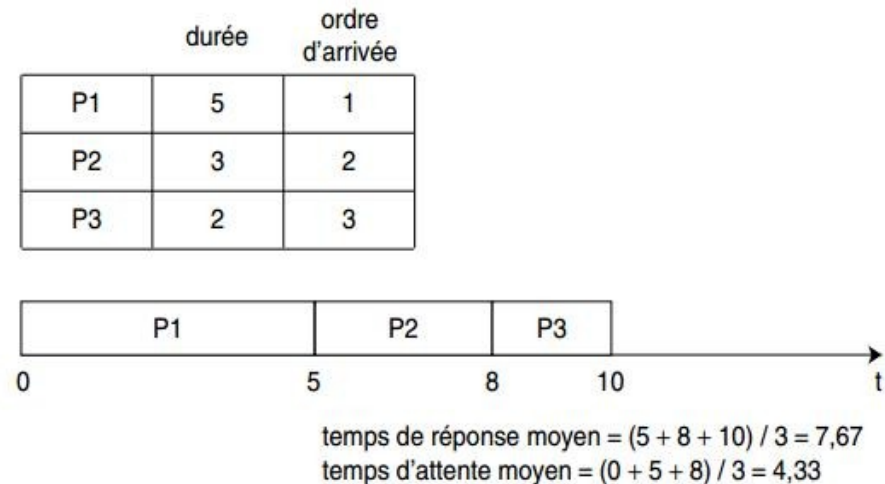


Figure 13 : Politique « Premier Arrivé, Premier Servi ».

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Politiques d'ordonnancement

B. Politique Plus Court d'Abord

- Le **processus** de **plus petit temps d'exécution** est celui qui est **ordonné** en **premier**.
- La politique est **sans réquisition**.
- **Remède** à l'**inconvenient** cité pour la **politique** précédente du « **Premier Arrivé, Premier Servi** ».
- **Politique** est **optimale** dans le sens où elle permet d'obtenir le **temps de réponse** moyen minimal pour un ensemble de processus donné.
- La **difficulté** : la **connaissance** a priori des **temps d'exécution** des processus. Cette **connaissance** n'est **pas disponible** dans un **système interactif**.
- **Politique** est essentiellement **mise en œuvre** dans les **systèmes** de **traitement** par **lots**.

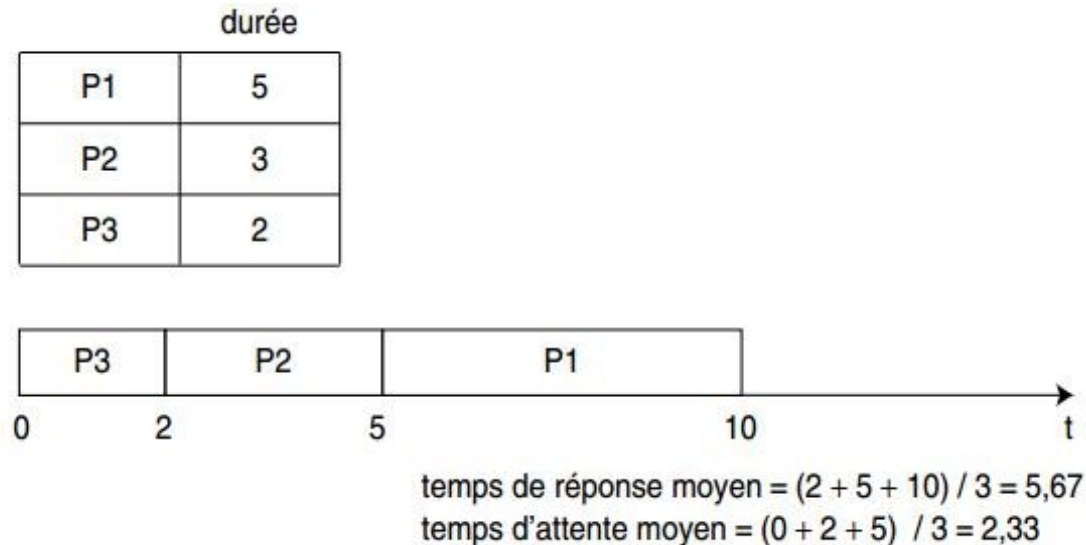


Figure 14 : Politique « Plus Court d'Abord ».

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Politiques d'ordonnancement

C. Politique par priorité

- La **politique** très **courante**.
- Chaque **processus possède** une **priorité**, un **nombre positif**. Selon le système d'exploitation, une valeur basse est plus prioritaire qu'une valeur haute ou inversement.
- Cette politique se **décline** en **deux versions** selon si la **réquisition** est **autorisée** ou **non**. La **figure 16** donne un **exemple d'application** de cette **politique** en mode **préemptif** et en mode **non préemptif**.
- Un **inconvenient** de cette politique est le **risque** de **famine** pour les **processus** de **petite priorité** si il y a de **nombreux processus** de **haute priorité**. On dit aussi qu'il y a **coalition** des processus de forte priorité contre les processus de faible priorité.

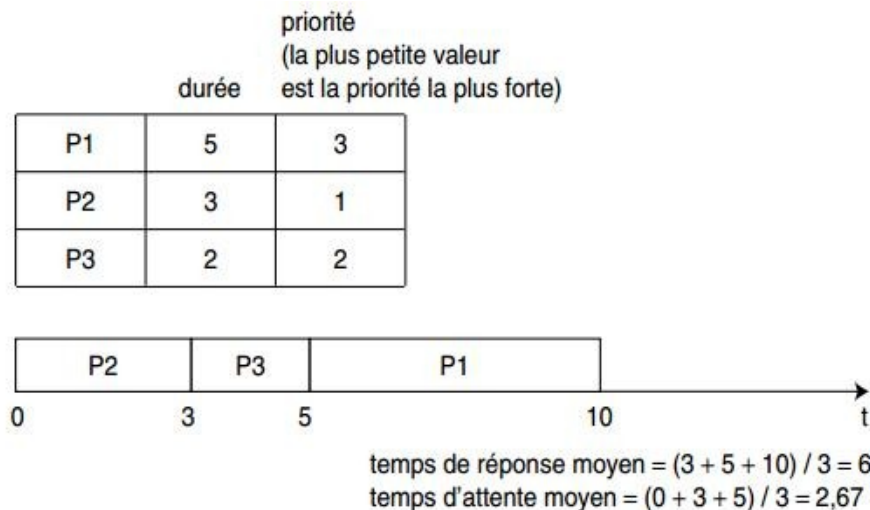


Figure 15 : Politique par priorité.

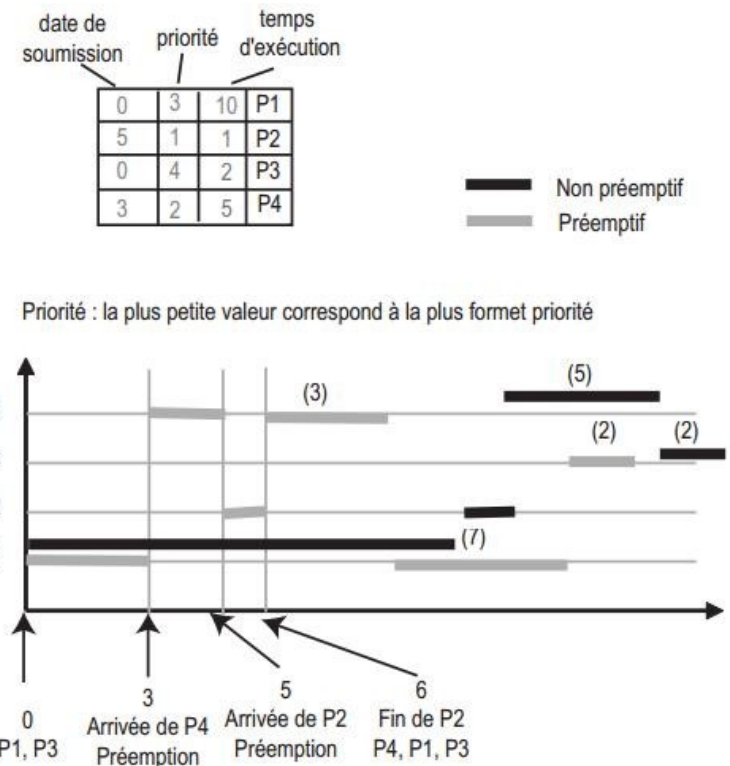


Figure 16 : Politique par priorité préemptive et non préemptive.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Politiques d'ordonnancement

D. Politique du tourniquet (round robin)

- Politique mise en œuvre dans les systèmes dits en temps partagé.
- Le temps est découpé en tranches nommées **quantums de temps**.
- Un processus est élu, il s'exécute au plus durant un **quantum de temps**.
- Si le processus n'a pas terminé son exécution à l'issue du quantum de temps, il est **préempté** et il réintègre la file des processus prêts mais en fin de file.
- Le processus en tête de file de la file des processus prêts est alors à son tour élu pour une durée égale à un quantum de temps.
- La valeur du quantum constitue un **facteur important** de performance de la politique, elle influe directement sur le nombre de **commutations de contexte**.

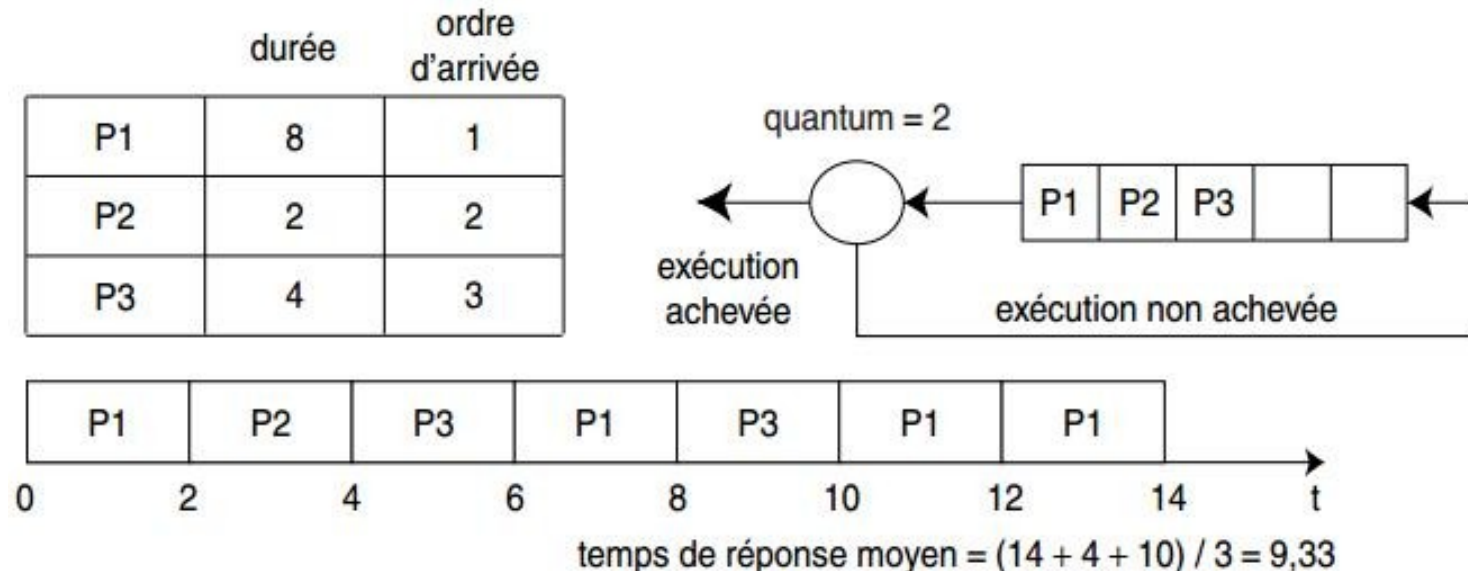


Figure 17 : Politique du tourniquet.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Exemples

- Les **systèmes actuels combinent** très souvent deux des **politiques** que nous avons vues : celles des **priorités fixes** et celles du **tourniquet**.
- La **file** des processus prêts est en fait **divisée** en autant de sous **files** qu'il **existe** de **niveaux** de **priorité** entre les processus.
- Chaque **file** de **priorité P_i** est **gérée** en **tourniquet** avec éventuellement un **quantum q_i** de temps propre.
- Pour **remédier** au **problème** de **famine** des **processus** de plus **faible priorité**, un **mécanisme d'extinction** de **priorité** peut être **mis en œuvre**, la **priorité** d'un processus **baisse** au **cours** de son **exécution**.

Prêt

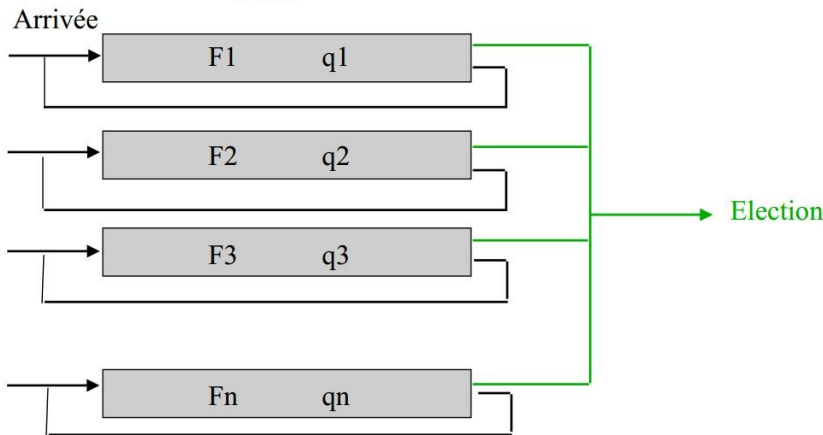


Figure 17 : multfiles sans extinction

Prêt

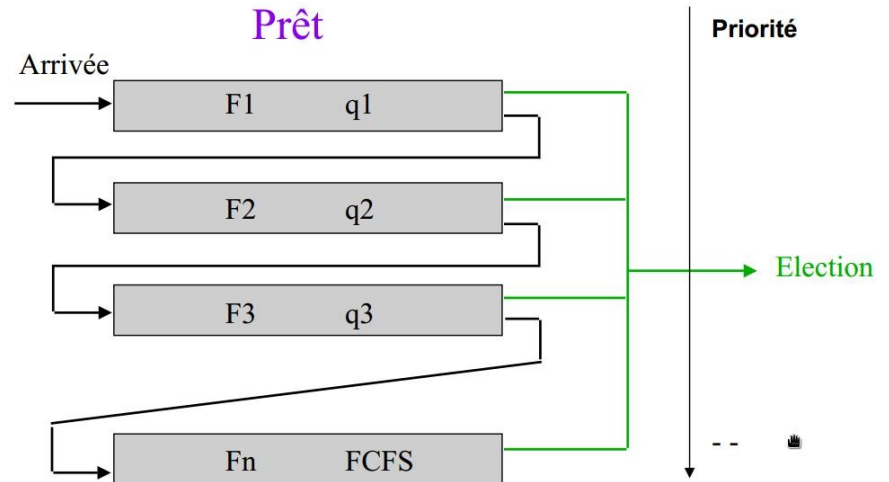


Figure 18: multfiles avec extinction

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Exemples

Ordonnancement sous Unix

- L'**ordonnanceur** Unix est un ordonnanceur de **type tourniquet**, avec **plusieurs niveaux** de **priorités**.
- À un instant t , le **processus élu** est celui de **plus forte priorité**.
- Le **système** effectue un **recalcul** des **priorités** mettant en œuvre un **principe d'extinction** de **priorité** afin de **garantir** une **équité d'accès** au **processeur**.

Exemple

- Supposons que **toutes** les **secondes**, le **système recalcule** les **priorités** des **processus élus** et **prêts** selon le principe suivant :
 - **Extinction** : $\text{compte_UC} = \text{compte_UC} / 2$ où compte_UC est le **temps CPU consommé** par le **processus**.
 - **Priorité** = $\text{compte_UC} / 2 + 40$ où **40** est une **priorité de base** de niveau utilisateur.
- Soient trois processus **P1, P2 et P3**, **P1** de priorité **40** et **P2, P3** de priorité **45**. **P1 s'exécute**.
 - Au bout d'**1 seconde** :
 - Priorité P1 : $\text{compte_UC} = 60/2 = 30$ et $\text{priorité} = 15 + 40 = 55$
 - Priorité P2, inchangée : 45
 - Priorité P3, inchangée : 45
 - **P2 est élu**
 - Au bout de **2 secondes** :
 - Priorité P1 : $\text{compte_UC} = 30/2 = 15$ et $\text{priorité} = 7 + 40 = 47$
 - Priorité P2 : $\text{compte_UC} = 60/2 = 30$ et $\text{priorité} = 15 + 40 = 55$
 - Priorité P3, inchangée : 45
 - **P3 est élu**
 - Au bout de **3 secondes** :
 - Priorité P1 : $\text{compte_UC} = 15/2 = 7$ et $\text{priorité} = 3 + 40 = 43$
 - Priorité P2 : $\text{compte_UC} = 30/2 = 15$ et $\text{priorité} = 17 + 40 = 47$
 - Priorité P3 : $\text{compte_UC} = 60/2 = 30$ et $\text{priorité} = 15 + 40 = 55$
 - **P1 est de nouveau élu**

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Exemples

Ordonnancement sous Linux

Démo : [diapos30/primitives_ordonnancemen.c](#)

Principe de l'ordonnancement

- Dans le système Linux, chaque **processus** est **qualifié** par une **priorité**.
- le système Linux offre **trois politiques d'ordonnancement** différentes :
 - **SCHED_FIFO** : élit à tout instant le **processus** de **plus forte priorité** parmi les **processus** attachés à **cette classe**.
 - **SCHED_RR** : est une politique de type **tourniquet** entre **processus** de **même priorité**.
 - **SCHED_OTHER** : implémente une politique à **extinction de priorité**.
- Un **processus créé** est **attaché** à l'une des **politiques** par un **appel** à la fonction **système** `sched_setscheduler()`.
- Les **processus attachés** aux politiques **SCHED_FIFO** et **SCHED_RR** sont **plus prioritaires** que les **processus attachés** à la politique **SCHED_OTHER**.
- Le système Linux, offre un ensemble de **primitives systèmes** pour la **gestion** de l'**ordonnancement**.

Modification ou récupération des paramètres d'ordonnancement

- Les **prototypes** de ces fonctions déclarées dans le fichier `<sched.h>` sont :
 - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param*param)` : **modification** de la **politique d'ordonnancement** `policy` du **processus identifié** par `pid`. Le paramètre `policy` prend une des trois valeurs **SCHED_FIFO**, **SCHED_RR** et **SCHED_OTHER**. Cette modification ne peut être **réalisée** qu'avec des droits équivalents à ceux du **super-utilisateur**. Si `pid` est nul, le processus courant est concerné.
 - `int sched_getscheduler(pid_t pid)` : **récupération** de la **politique d'ordonnancement associée** au **processus** identifié par `pid`. Si `pid` est nul, le processus courant est concerné.
 - `int sched_setparam(pid_t pid, const struct sched_param *param)` : **modification** des **paramètres d'ordonnancement** du **processus** identifié par `pid`. La structure `param` contient un seul champ, **correspondant** à la **priorité statique** du **processus**. Si `pid` est nul, le processus courant est concerné.
 - `int sched_getparam(pid_t pid, const struct sched_param *param)` : **récupération** des **paramètres d'ordonnancement** du **processus** identifié par `pid`. Si `pid` est nul, le processus courant est concerné.

ORDONNANCEMENT SUR L'UNITÉ CENTRALE

Exemples

Ordonnancement sous Linux

Priorités et quantum

- Les **primitives** `sched_get_priority_min()`, `sched_get_priority_max()` et `sched_rr_get_interval()` permettent respectivement de connaître les valeurs des priorités statiques minimale et maximale associées à une politique d'ordonnancement ainsi que la valeur du quantum de temps associé à un processus dans le cadre d'un ordonnancement **SCHED_RR**. Les **prototypes** de ces fonctions déclarées dans le fichier `<sys/wait.h>` sont :
 - `int sched_get_priority_min(int policy);`
 - `int sched_get_priority_max(int policy);`
 - `int sched_rr_get_interval(pid_t pid, struct timespec *interval);`
- Enfin, les **primitives** `nice()`, `setpriority()` et `getpriority()` permettent aux processus de modifier ou connaître leur priorité ou la priorité d'un groupe de processus.
- **Seul un processus privilégié peut augmenter sa priorité.**
- La primitive `nice(int inc)` permet de **baiss**er la priorité de base d'un processus de la valeur `inc`.
- Les primitives `setpriority()` et `getpriority()` permettent de **baiss**er ou de **connaître** la **priorité** d'un processus.
- Leurs **prototypes** sont :
 - `#include <unistd.h>`
 - `int nice(int inc);`
 - `#include <sys/wait.h>`
 - `int setpriority(int which, int who, int prio);`
 - `int getpriority(int which, int who);`

Démo : diapos31 : exemple-nice.c, exemple-getpriority.c, exemple-get-priority-min-max.c

Conclusion

 A FAIRE

TP

TP : A FAIRE