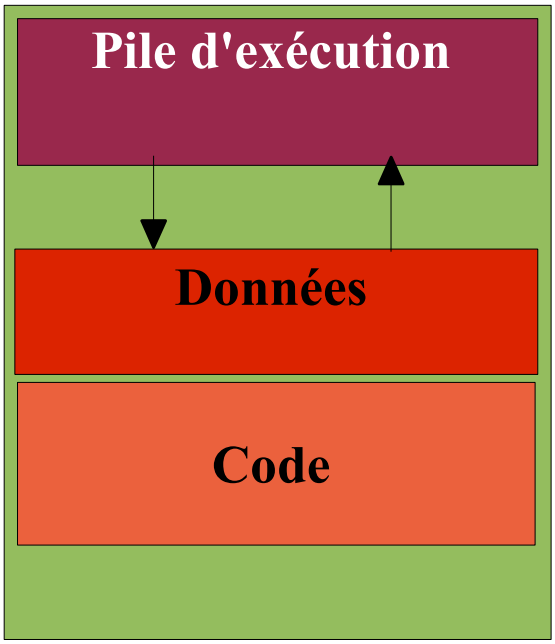
# Chapitre 2
# Création et ordonnancement de Processus

# Plan

# 2. Création et Ordonnancement de Processus

## Rappels

Les processus sont composés d'un espace de travail en mémoire formé de 3 segments :




**Le code correspond aux instructions, en langage d'assemblage, du programme à exécuter.**

**La zone de données contient les variables globales ou statiques du programme ainsi que les allocations dynamiques de mémoire.**

**Enfin, les appels de fonctions, avec leurs paramètres et leurs variables locales, viennent s'empiler sur la pile.**

Les zones de pile et de données ont des frontières mobiles qui croissent en sens inverse lors de l'exécution du programme.

> Un processus est donc un «programme» qui s'exécute et qui possède son propre compteur ordinal (@ de la prochaine instruction exécutable), ses registres et ses variables.

Le concept de processus n'a donc de sens que dans le cadre d'un contexte d'exécution. Conceptuellement, chaque processus possède son propre processeur virtuel, en réalité, le **vrai processeur commute entre plusieurs processus**.

Le noyau maintient une table pour gérer l'ensemble des processus. Cette table contient la liste de tous les processus avec des informations concernant chaque processus.

> Le nombre des emplacements dans cette table des processus est limité pour chaque système et pour chaque utilisateur.

# A. L'ordonnanceur

**Objectif** : Sur un intervalle de temps assez grand, faire progresser tous les processus, tout en ayant, à un instant donné, un seul processus actif (dans le processeur).

**Rôle de l'ordonnanceur** : Prendre en charge la **commutation des tâches**, qui assure le passage d'une tâche à une autre.

=> **multi-programmation**

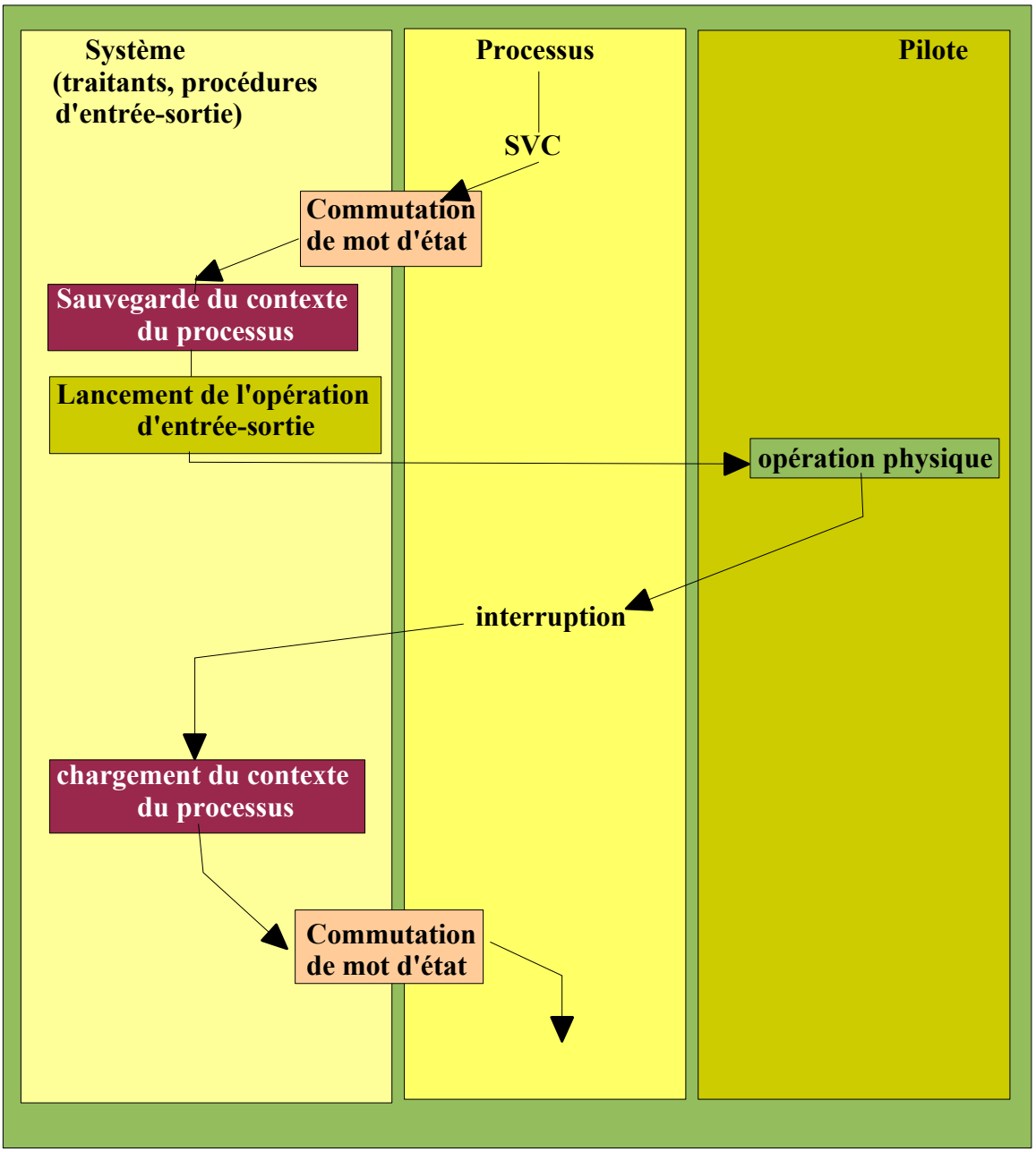
Réalisée par l'ordonnanceur au niveau le plus bas du système, elle est activée par des interruptions d'horloge, de disque et de terminaux.

**But de la multiprogrammation** : maintenir le taux d'utilisation du ou des processeurs le plus élevé possible.

Il faut donc agir pour que le processeur soit le moins inactif possible.

Un PROCESSUS ⇒ une succession de phases de calcul (où un processeur est actif) et d'Entrées/Sorties (largement indépendantes des processeurs).

***Idée de base*** : tenter de faire se recouvrir une phase d'E/S d'un processus avec des phases de calcul d'autres processus, en effectuant une **commutation de contexte** dès qu'un processus entame une phase d'E/S.




SVC (SuperVisor Call) est un appel système réalisé par un processus.

L'objectif est d'optimiser ces grandeurs :

**Taux d'utilisation de l'unité centrale** : rapport entre la durée où l'unité centrale est active et la durée totale.

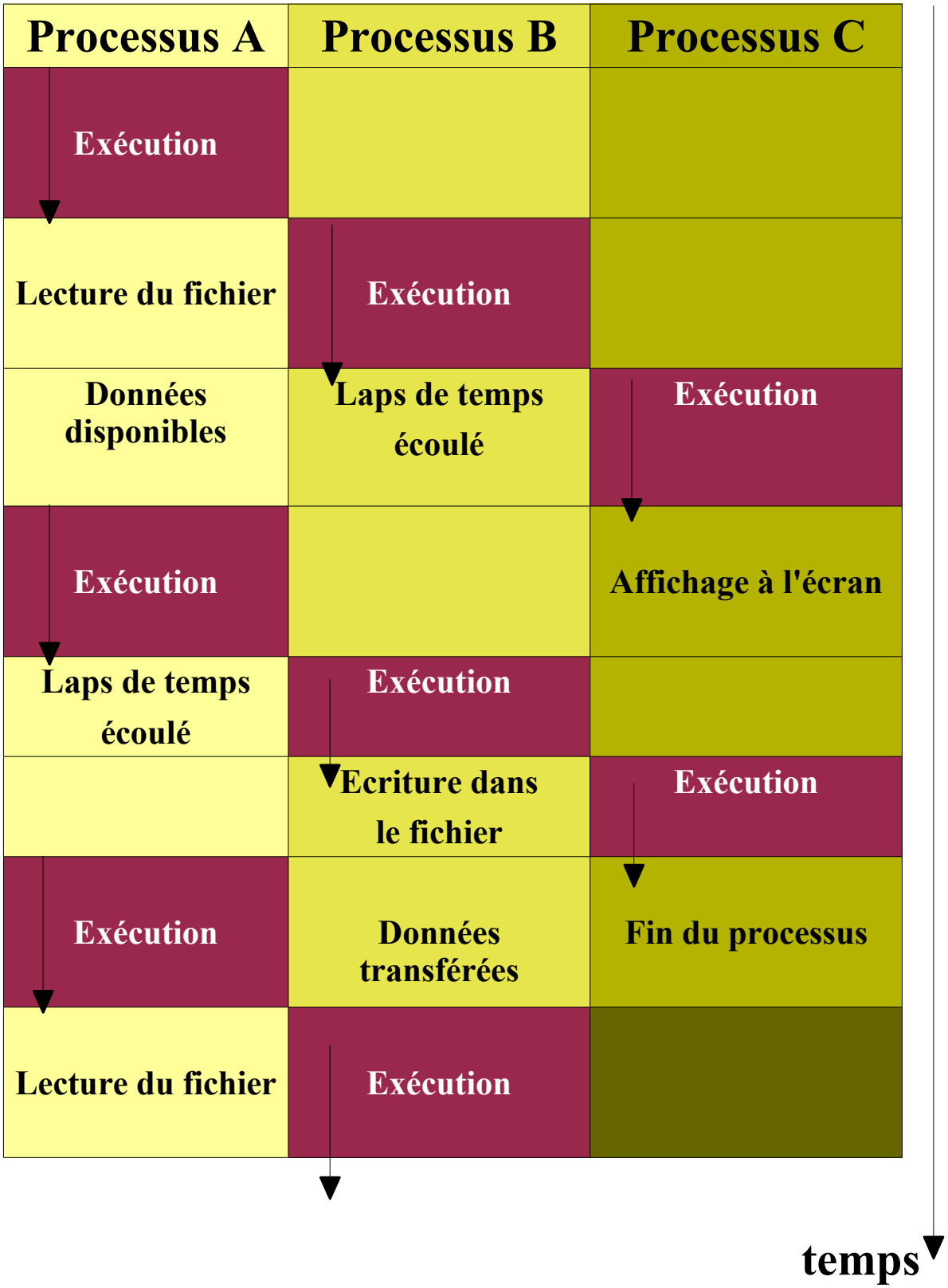
En pratique on obtient des valeurs comprises entre 40 et 95 %.

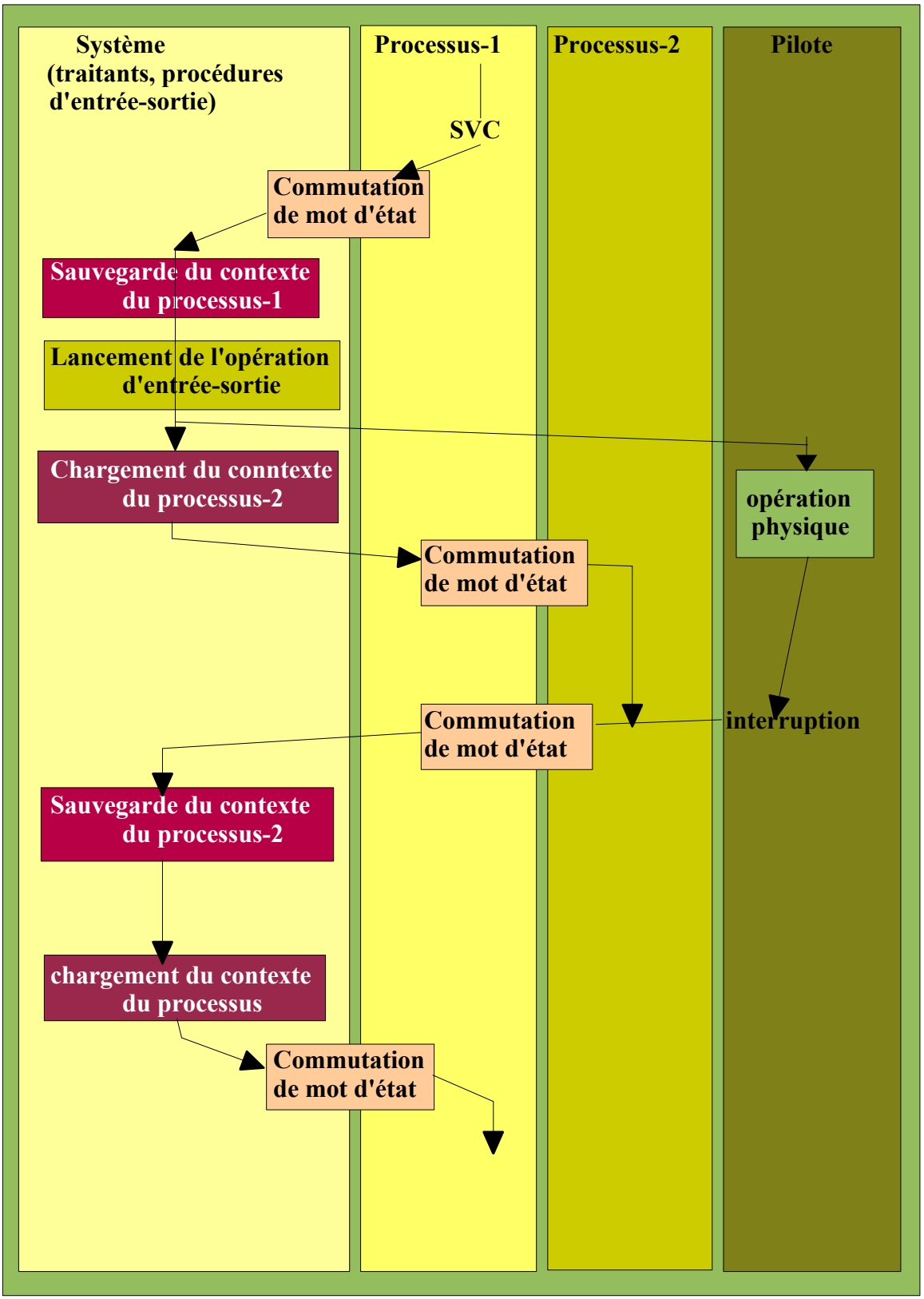
**Débit** = nombre de programmes utilisateurs traités en moyenne par unité de temps.

**Temps de traitement moyen** = moyenne des intervalles de temps séparant la soumission d'une tâche de sa fin d'exécution.

**Temps de traitement total** = pour un ensemble de processus donné.

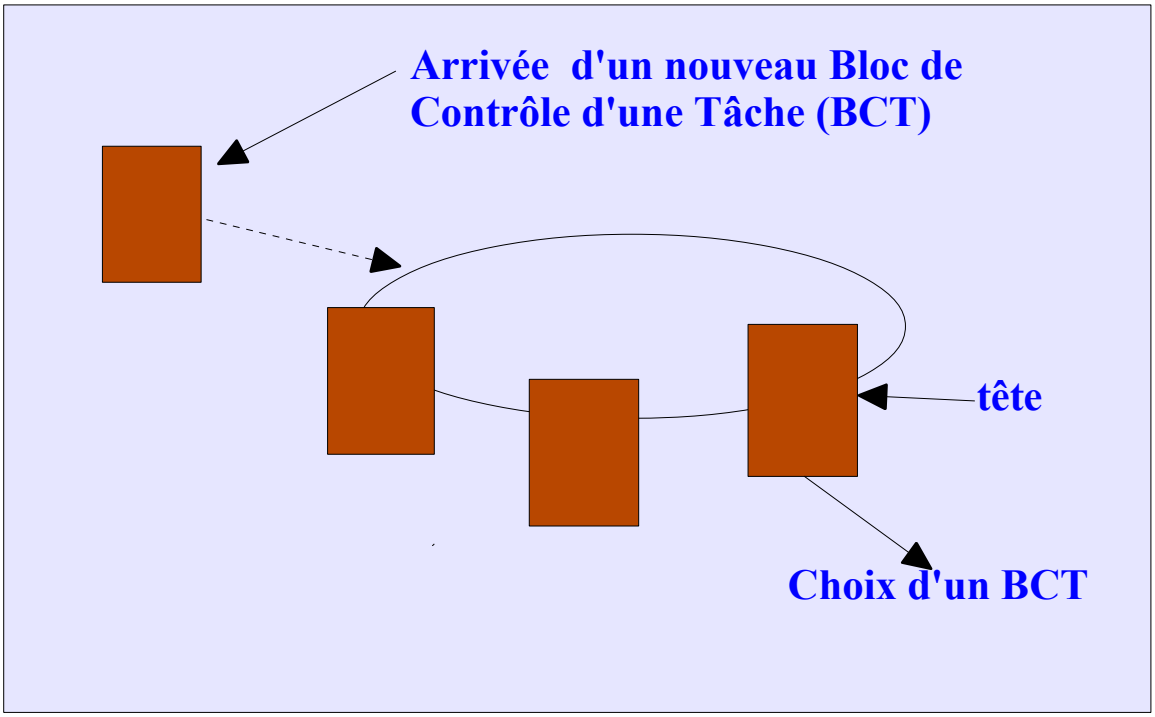
**Temps de réponse maximum** = maximum des durées séparant la soumission d'une requête par un processus de son accomplissement.


Processus A
Processus B
Processus C
Exécution
Lecture du fichier
Exécution
Données disponibles
Laps de temps écoulé
Exécution
Exécution
Affichage à l'écran
Laps de temps écoulé
Exécution
Ecriture dans le fichier
Exécution
Exécution
Données transférées
Fin du processus
Lecture du fichier
Exécution
temps




L'ordonnanceur associe dynamiquement (recalculées périodiquement) des priorités à chaque processus.

L'algorithme du tourniquet («**round robin**») permet d'arbitrer les conflits entre les processus de même priorité, ou dont les phases de calcul sont trop longues.



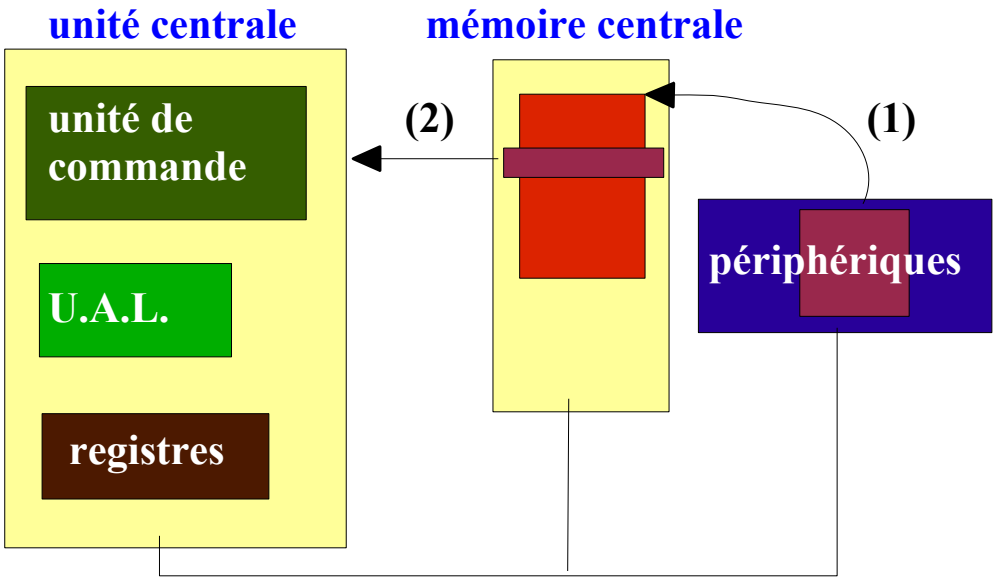

*La file d'attente de l'unité centrale est organisée comme une liste circulaire.*

Le système d'exploitation partage l'unité centrale entre toutes les tâches qui la demandent.

Pour construire un ordonnanceur, il suffit de disposer :

- **d'un mécanisme d'interruption**
- **d'un mécanisme de commutation de contexte**
- **et d'une horloge.**

On choisit un **intervalle de temps** (quantum) et on génére une **interruption** à la fin de chaque quantum. Cette interruption provoque une commutation de contexte qui redonne le contrôle à l'ordonnanceur.




1. Pour être exécuté et donner naissance à un processus, un programme et ses données sont chargées en mémoire centrale.

2. Puis les instructions sont transférées individuellement de la mémoire centrale vers l'unité centrale, où elles sont exécutées.

L'unité centrale contient entre autre l'UAL (Unité Arithmétique et Logique) pour effectuer les calculs et un ensemble de registres :

**- des registres pour ranger des valeurs de manière temporaire,**

**- des registres contenant certaines informations sur son propre statut (active ou inactive, mode système ou utilisateur, ...) et sur le processus en cours d'exécution (zone mémoire accessible et droits d'accès correspondants, niveau de priorité, ...).**

Tous ces registres forment ce que l'on appelle :
un **mot d'état** (PSW, "Processus Status Word").

A tout instant, un processus est donc entièrement caractérisé par :

**1) le programme sous-jacent et ses données (contexte en mémoire),**

**2) la valeur de son mot d'état (contexte d'unité centrale).**

## La commutation de mot d'état

1) sauvegarde de la valeur du mot d'état dans une zone précise.

2) chargement d'une nouvelle valeur, à partir d'une zone précise.

## Interruption

C'est une commutation de mot d'état provoquée par un **signal** généré par le matériel.

Ce signal est la conséquence d'un évènement qui peut être **interne** au processus et résultant de son exécution, ou bien **extérieur** et indépendant de cette exécution.

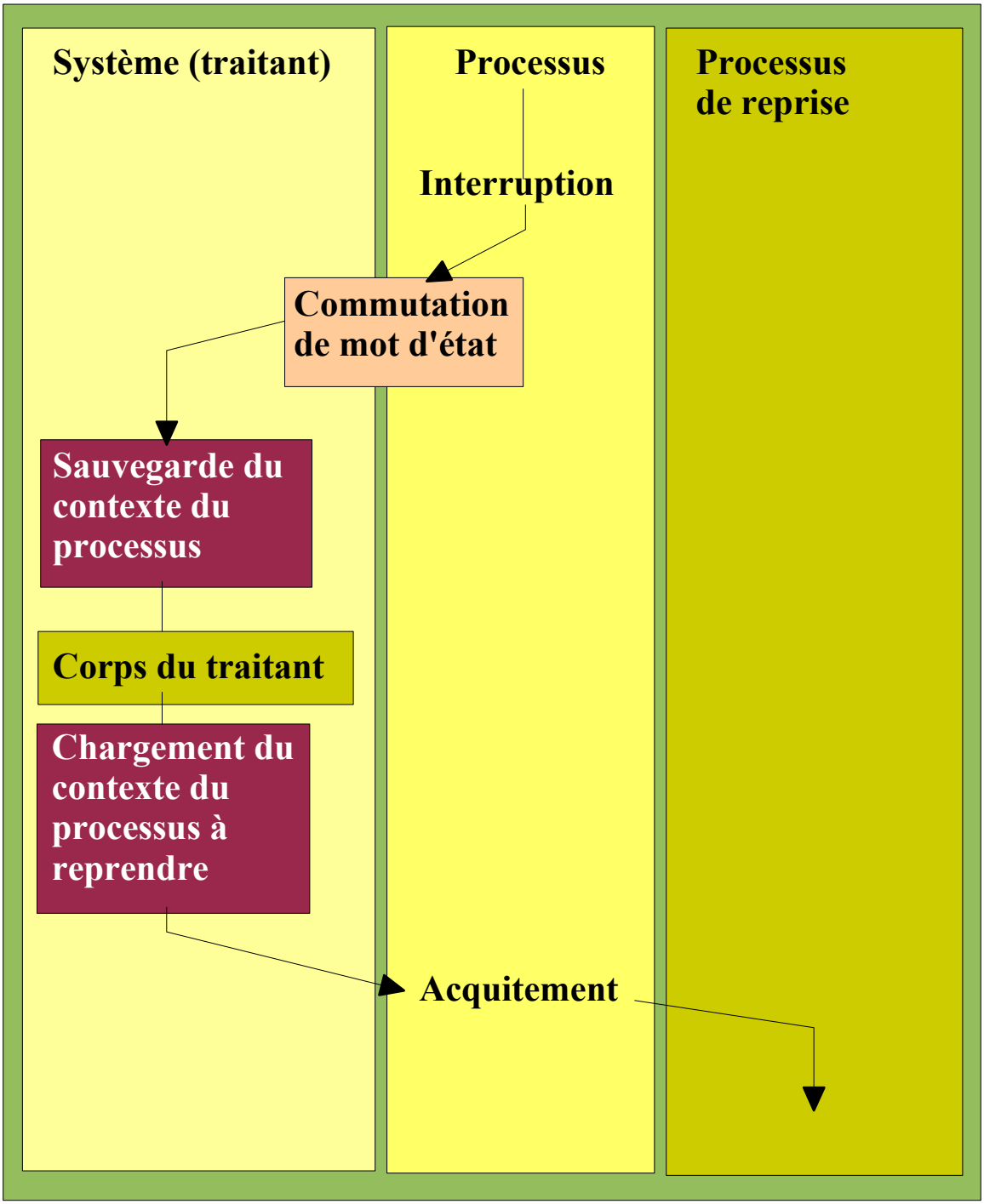
Le signal modifie la valeur d'un indicateur qui est consulté régulièrement par le système.

Celui-ci doit alors déterminer la cause de l'interruption.

Un **vecteur d'interruption** correspond à plusieurs bits du mot d'état. C'est un emplacement mémoire contenant une adresse. L'arrivée de l'interruption provoque le branchement à cette adresse.

A chaque cause d'interruption est associée un niveau d'interruption. Il existe trois principaux niveaux :

- **interruptions externes (panne, opérateur, ...)**
- **déroutement (erreur, division/0, débordement de tableau, ...),**
- **appels au système SVC (E/S, ...).**




Comme le processeur commute entre les processus, la vitesse d'exécution d'un processus ne sera pas uniforme et variera vraisemblablement si les mêmes processus étaient exécutés à nouveau.

**Il ne faut donc pas que les processus fassent une quelconque présomption sur le facteur temps.**

# Priorités des niveaux d'interruption

S'il existe plusieurs niveaux d'interruption, on peut avoir à un moment donné, **plusieurs indicateurs d'interruption positionnés en même temps**.

D'où l'utilisation de priorités sur les niveaux d'interruption.

Si dans une interruption, une seconde se produit, ..., les processus ne progressent plus ; on fait des commutations de mots d'état en cascade.

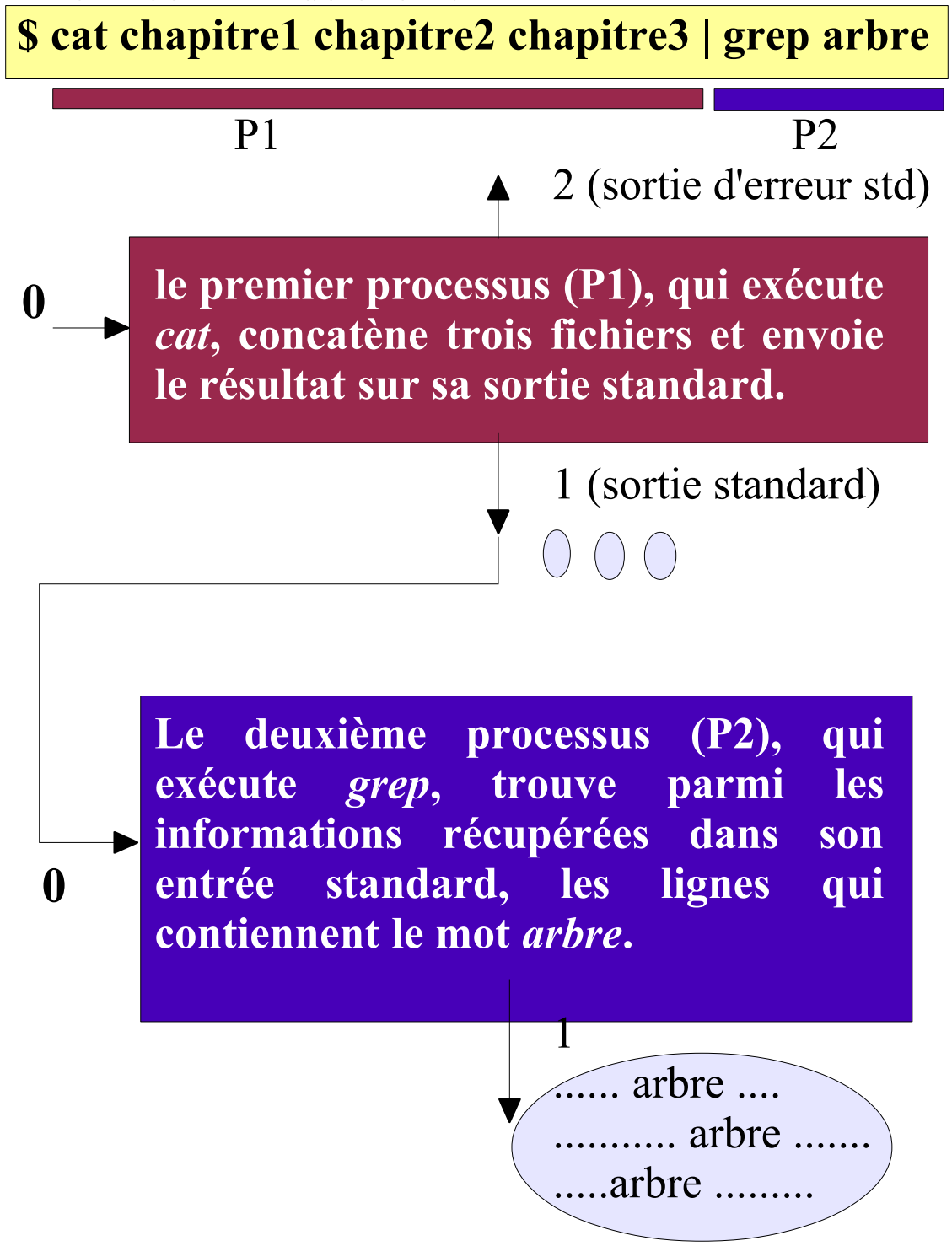
D'où le **masquage** et le **désarmement** de niveaux d'interruption.

- En **masquant** un niveau d'interruption, on retarde la prise en compte des interruptions de ce niveau.

- En **désarmant** un niveau d'interruption on annule l'effet des interruptions correspondantes. On peut réarmer un niveau désarmé.

Les processus, bien qu'étant des entités indépendantes, doivent parfois interagir avec d'autres processus. Les résultats d'un processus peuvent, par exemple, être les données d'un autre.

Dans la commande shell :




Les vitesses relatives des deux processus (qui dépendent à la fois de leur complexité respective et du temps que leur consacre le processeur) peuvent entraîner la situation suivante : *grep* est prêt à s'exécuter mais ne peut le faire faute de données. Il doit alors se **bloquer** en attendant les données.

Cet ordonnancement des processus implique la notion d'**état d'un processus** ; on trouve principalement trois états :

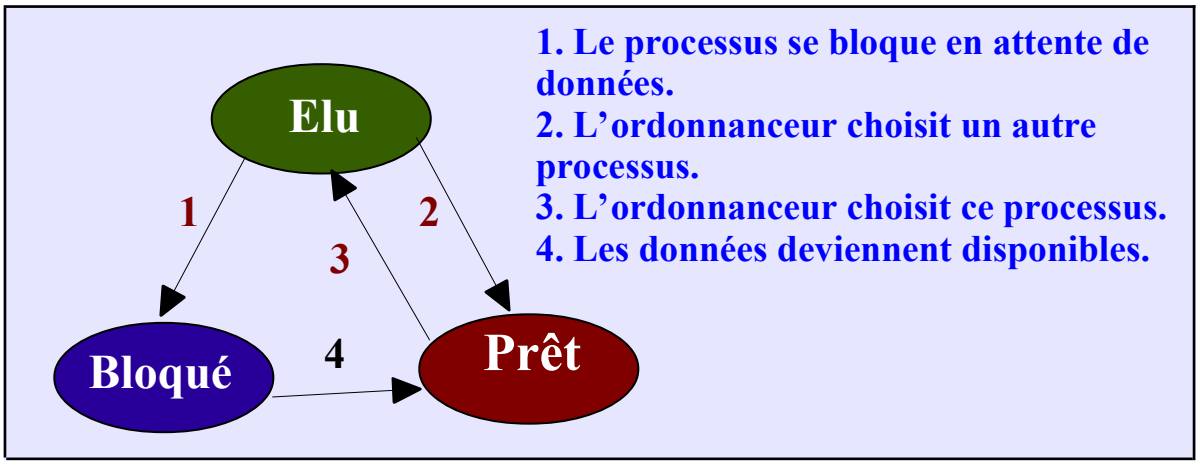
**Elu** : un processus est actif en mémoire centrale (en cours d'exécution).

**Prêt** : un processus est suspendu provisoirement en attente d'exécution, pour permettre l'exécution d'un autre processus.

**Bloqué** : un processus attend un évènement extérieur pour pouvoir continuer ; il est bloqué sur une ressource (par exemple au cours d'une lecture sur disque).

Les processus terminés ne sont pas immédiatement éliminés de la table des processus. Ils ne le seront que par l'appel à une instruction explicite de leur père. Ceci correspond à l'état ***défunt***.

Le diagramme simplifié des états (3) d'un processus comportant 4 transitions est le suivant :



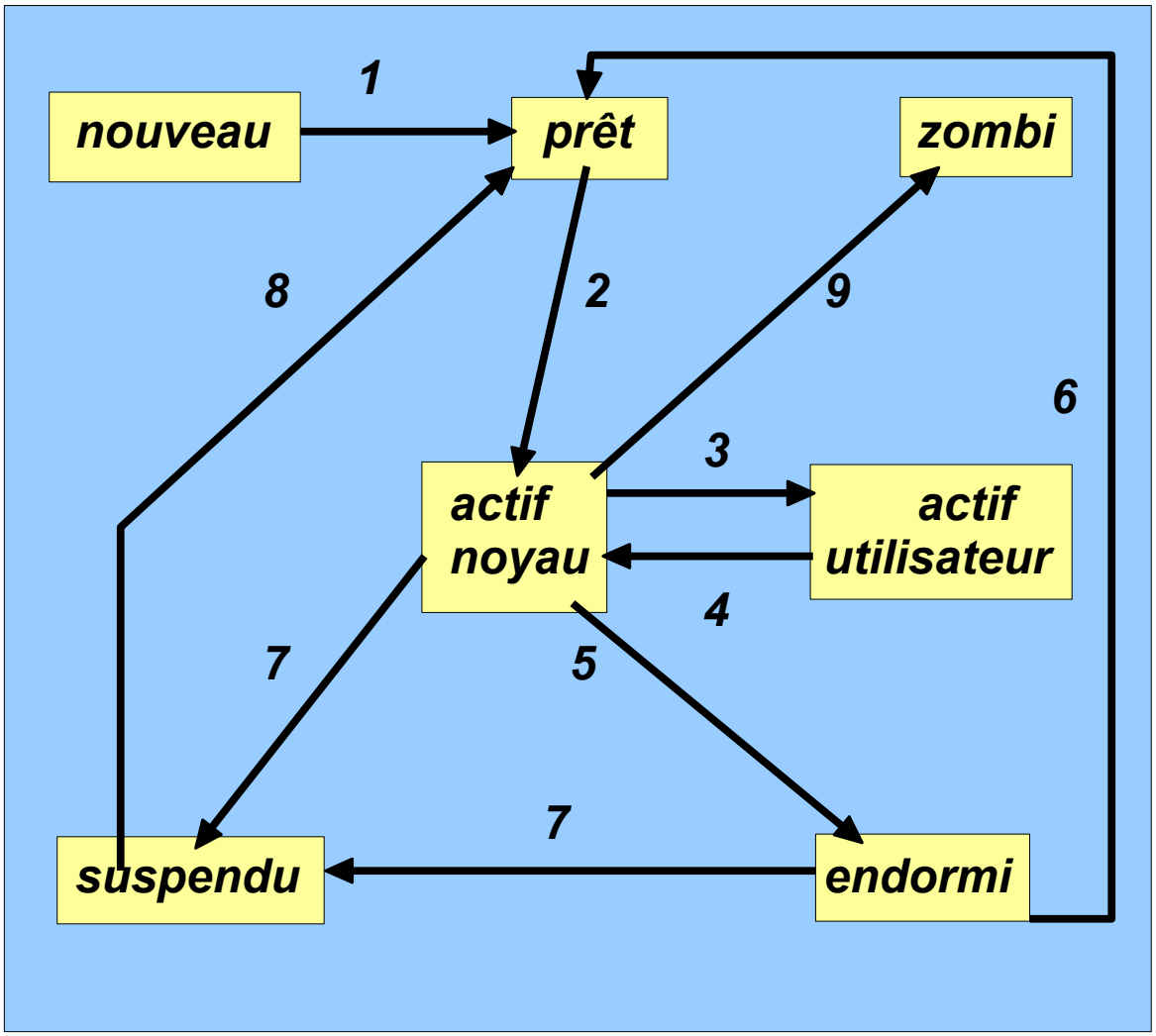

- La transition 1) " **Elu** -> **Bloqué** " se produit lorsqu'un processus ne peut plus poursuivre son exécution, par exemple lors d'une attente pour effectuer une lecture sur le disque. Le processus est retiré du processeur.
- Les transitions 2) et 3) sont provoquées par l'ordonnanceur de processus, sans que les processus s'en rendent compte. La transition 2) " **Elu** -> **Prêt** " a lieu lorsque l'ordonnanceur décide que le processus en cours s'est exécuté pendant une durée suffisante. Il lui retire le processeur et l'alloue à un autre processus, c'est la transition 3) " **Prêt** -> **Elu** ".
- La transition 4) " **Bloqué** -> **Prêt** " est provoquée si l'évènement extérieur attendu par un processus (ex. arrivée de données) se produit. Si aucun autre processus n'est en cours à cet instant, cette transition 4) sera immédiatement suivie par la 3).

Il existe de nombreux algorithmes pour choisir le processus prêt qui va être exécuté parmi la liste des processus prêts (plus vieux, plus avancé dans son traitement, celui qui a consommé le moins de temps CPU, le plus prioritaire, ...).

C'est donc le système d'exploitation qui détermine et modifie l'état d'un processus, sous l'effet d'évènements. Les évènements peuvent être internes au processus (une demande d'entrée/sortie peut faire passer de l'état en exécution ou ***élu*** à l'état ***bloqué***) ou extérieurs, provenant du système d'exploitation (une attribution de ressource fait passer de l'état ***bloqué*** à l'état ***prêt***).

Lorsque la mémoire ne peut contenir tous les processus prêts (idem pour bloqués), un certain nombre d'entre eux sont déplacés sur le disque. Un processus peut alors être prêt (bloqué) en mémoire ou prêt (bloqué) sur le disque.

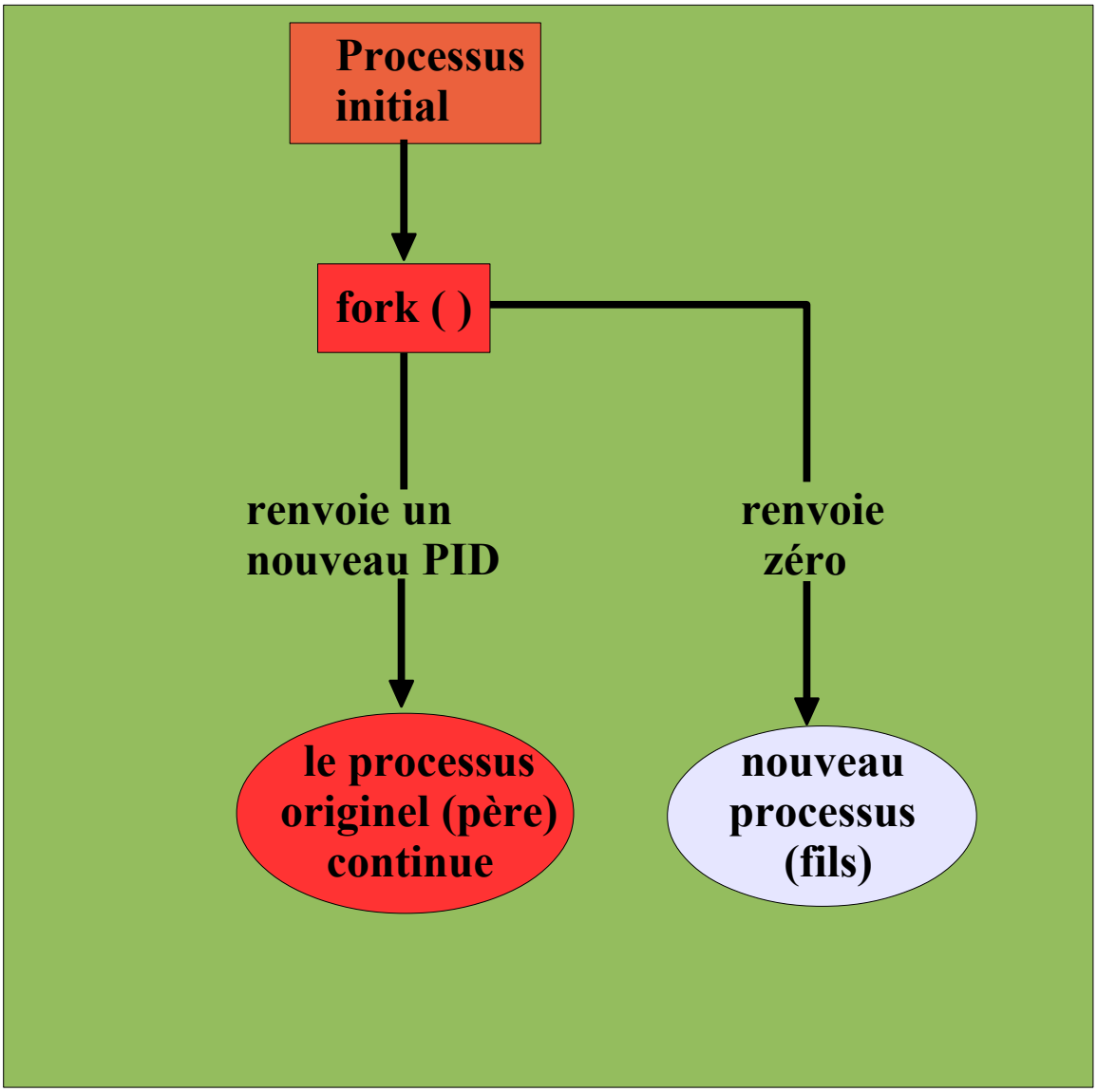
Un diagramme plus complet des états (7) d'un processus comportant cette fois 9 transitions est le suivant :




1. **Le processus a acquis les ressources nécessaires à son exécution.**
2. **Le processus vient d'être élu par l'ordonnanceur ; il y a alors un changement de contexte.**
3. **Le processus revient d'un appel système ou d'une interruption (par ex. il vient d'être élu ou il a terminé l'exécution du *handler* d'une interruption).**
4. **Le processus a réalisé un appel système ou une interruption est survenue (périphérique ou horloge).**
5. **Le processus se met en attente d'un événement : appel système bloqué par un événement interne au système (libération d'une ressource ou terminaison d'un processus) ou extérieur (terminaison d'une E/S d'un périphérique).**
6. **L'événement attendu par le processus s'est produit.**
7. **Délivrance d'un signal particulier (SIGSTOP).**
8. **Réveil du processus par le signal SIGCONT.**
9. **Le processus se termine.**

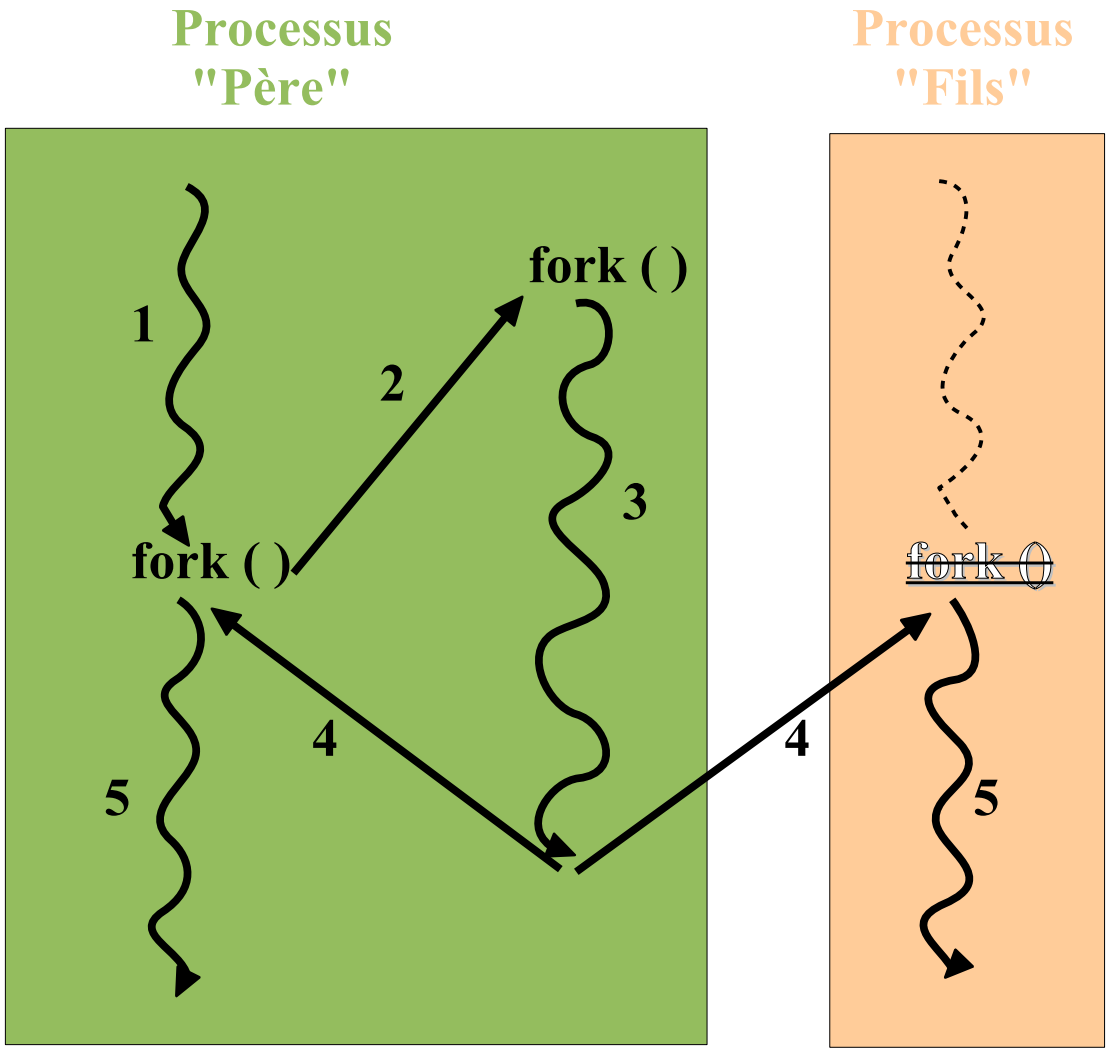
# B. Création de processus : fork ()

Cet appel système permet de créer dynamiquement (en cours d'exécution) un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé.




L'appel système fork() crée donc un nouveau processus ainsi que son contexte initial. Alors que dans certains systèmes, le contexte initial d'un processus est un contexte prédéfini, **la fonction fork() d'UNIX, crée le nouveau contexte par duplication du contexte du processus parent au moment de l'appel.**

**Le nouveau processus exécute donc le même code que le processus parent et démarre comme lui au retour du fork().**




**Les processus "père" et "fils" deviennent concurrents (phase n°5) vis-à-vis de l'ordonnaceur.**

Le processus ainsi créé (fils) hérite, du processus qui l'a créé (père), un certain nombre d'attributs dont :

- **le même code,**
- **une copie de la zone de données,**
- **l'environnement,**
- **la priorité,**
- **les différents propriétaires,**
- **les descripteurs de fichiers courant (mêmes caches et mêmes pointeurs),**
- **le traitement des signaux.**

Le seul moyen de distinguer le processus fils du processus père est que la valeur de retour de la fonction fork est 0 dans le processus fils créé et est égale au numéro du processus fils nouvellement créé, dans le processus père et -1 en cas d'échec.

L'utilisation classique de la fonction **fork** est la suivante :

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

```
switch(pid=fork())
{
      case -1: /*code correspondant au
               traitement de l'échec :
               printf(" panique: le fork()
               a échoué"); break; */

       case 0:  /* code correspondant à
              l'exécution du processus fils
              break; */

      default: /* code correspondant à
              l'exécution du processus
              père break; */
}
```

Pratiquement, après réalisation de la fourche, deux processus vont se dérouler de manière concurrente sur le même code (en principe réentrant) et le processus fils va recevoir une copie physique de la zone de données du processus père.

Le fils reçoit par ailleurs une nouvelle entrée (une nouvelle structure *PROC* et *USER* est allouée) dans la table des processus. Cette entrée est initialisée avec les informations contenues dans celle du père.

Les principales différences sont les suivantes : *PID* et espace mémoire occupée (piles différentes).

La fonction ***fork( )*** réalise les tâches suivantes :

1. test d'existence de ressources mémoires suffisantes pour permettre la création du nouveau processus,

2. calcul du *PID* du processus enfant. Le noyau explore ensuite la table *proc[]* pour vérifier que ce numéro n'est pas en cours d'utilisation. Si le numéro existe dans la table, le calcul est relancé,

3. recherche d'une entrée libre dans la table des processus,

4. duplication du contexte processeur et mémoire du processus parent,

5. mise à jour des tables système modifiées par la création du nouveau processus.

**Les processus créés par des *fork()* s'exécutent de façon concurrente avec leur père**. Ne pouvant présumer de la politique et de l'état de l'ordonnanceur du système, il sera impossible de déterminer quels processus se termineront avant tels autres (y compris leur père).

Dans certains cas le père meurt avant la terminaison de son fils ; à la terminaison du fils, on dira que ce dernier est dans un état **zombi** (dans l'affichage de la commande "ps", S=Z). Le noyau rattache ces processus au processus *init* (1).

D'où l'existence, dans certains cas, d'un problème de synchronisation.

# C. Primitive de synchronisation : wait()

La primitive *wait* permet l'élimination des processus zombis et la synchronisation d'un processus sur la terminaison de ses descendants avec récupération des informations relatives à cette terminaison. Elle provoque la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine.

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

La sémantique de cette primitive est la suivante :

- **si le processus appelant ne possède aucun fils, la primitive renvoie la valeur -1 et la variable *errno* a comme valeur *ECHILD*,**

- **si le processus appelant possède au moins un fils zombi, la primitive renvoie l'identité de l'un de ses fils zombis (c'est le système qui choisit lequel) et si l'adresse *pointeur_status* est différente de *NULL*, la valeur **pointeur_status* fournit des informations sur la terminaison de ce processus zombi (qui disparaît effectivement des tables du système).**

- **si le processus appelant possède des fils mais aucun fils zombi, le processus est bloqué jusqu'à ce que :**
  - **l'un de ses fils devienne zombi,**
  - **ou que l'appel système soit interrompu par un signal "non mortel". La valeur retournée par la primitive est alors -1 et la variable errno a comme valeur EINTR.**

# La primitive waitpid()

Introduite dans la norme POSIX, elle permet de sélectionner parmi les fils du processus appelant un processus particulier.

**Cette 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 *pointeur_status*.**

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

- 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** |

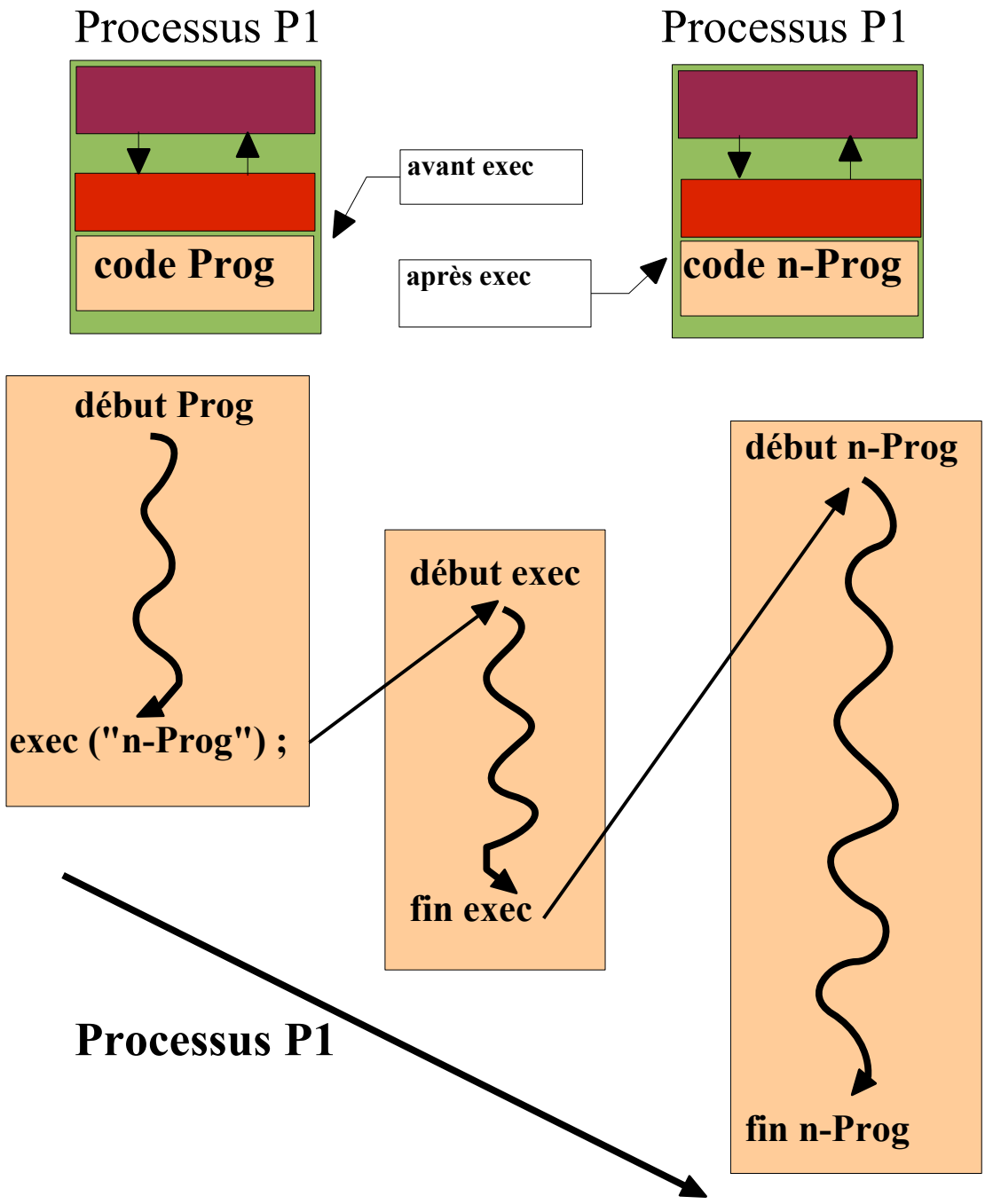
- Le paramètre ***options*** permet entre autre de choisir entre les modes bloquant ou non-bloquant.

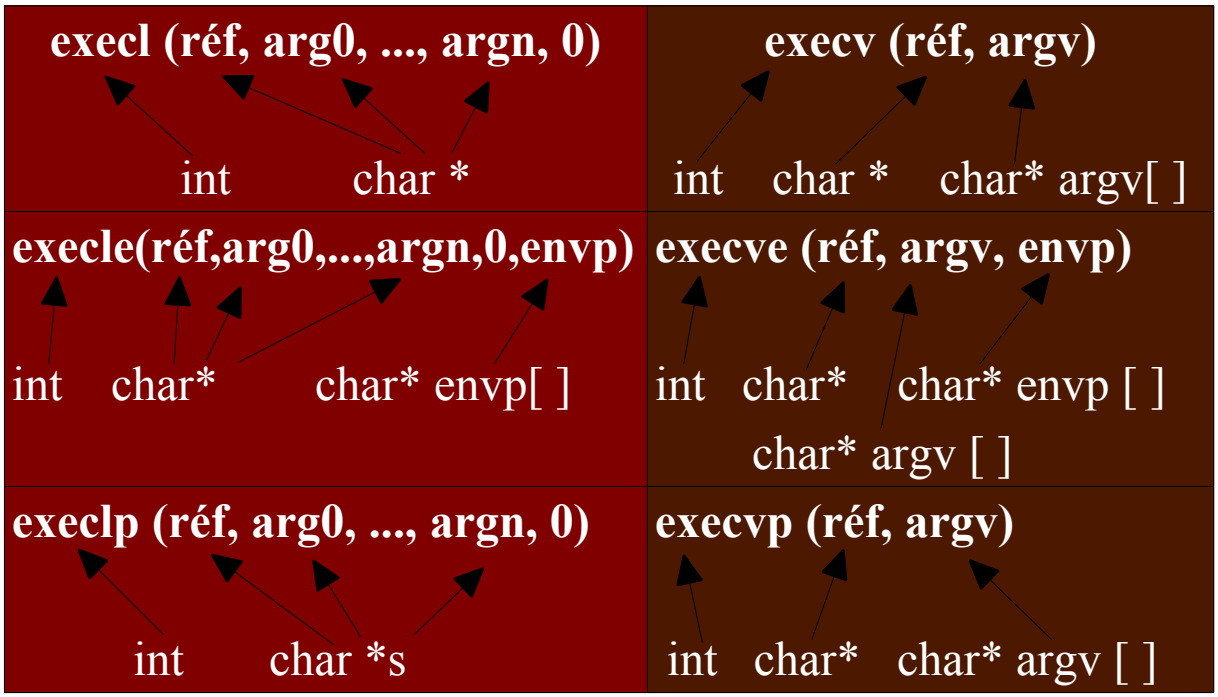
- La fonction renvoie :

| | |
|---|---|
| **-1** | **en cas d'erreur (en mode bloquant ou non bloquant) ;** |
| **0** | **en cas d'échec (processus demandé existant mais ni terminé ni stoppé) en mode non bloquant ;** |
| **le pid du processus fils zombi** | **pris en compte sinon.** |

## D. Primitives de recouvrement : exec() ...

Elles permettent le lancement de l'exécution par le processus appelant d'un nouveau programme ; le texte de ce nouveau programme recouvre alors celui de l'ancien programme exécuté. Il n'y a donc pas de retour d'un exec réussi, car on a écrasé l'ancien code.






**execl** : le nombre d'arguments est connu, mais le nombre de paramètres est variable (liste).

**execv** : le nombre d'arguments n'est pas connu, mais le nombre de paramètres est fixe.

**execle, execve** : avec passage d'environnement (redirection, priorité, ...).

**execlp, execvp** : il y a une recherche de référence (ref) dans le PATH (ensemble des chemins d'accès).

Prototype d'une fonction ***main*** appelée au début de l'exécution d'un fichier binaire :

**int main (int *argc*, char **argv*[ ], char **arge*[ ]);**

dans laquelle :

***argc*** est le nombre de composantes de la commande (nom de la commande et arguments) ;

***argv*** est un tableau de pointeurs sur caractères permettant l'accès aux différentes composantes de la commande (argv[0] étant le nom de la commande) dont la fin est marquée par un élément de valeur NULL ;

***envp*** est un tableau de pointeurs sur caractères permettant l'accès à l'environnement du processus.

```
#include <stdio.h>

main(int argc, char** argv, char** envp) {
  printf ("Variables d'environnement du processus :");
  while(*envp)
        printf(" \n %s",*envp++);

  printf ("\n");
}
```

# E. Visualisation de processus

***La commande "ps" permet de visualiser les processus existant à son lancement sur la machine.***

```
$ ps -cflj
F S UID  PID  PPID PGID  SID  CLS PRI ADDR  SZ  WCHAN STIME TTY   TIME     CMD
0 S c1   4208 4206 4208  4208  -   26  -     443 wait4 13:41 pts/1 00:00:00 bash
0 R c1   4273 4208 4273  4208  -   20  -     632 -     14:20 pts/1 00:00:00 ps -cflj
$
```

***Sans option, elle ne concerne que les processus associés au terminal depuis lequel elle est lancée.***

| Nom | Interprétation | Options(s) |
|---|---|---|
| S | Etat du processus<br>S -> endormi<br>R -> actif<br>T -> tracé ou stoppé<br>Z -> terminé (zombi) | -l |
| F | Indicateur de l'état du processus en mémoire (swappé, enn mémoire, tracé par un autre, ...) | -l |
| PPID | Identificateur du processus père | -l -f |
| UID | Utilisateur propriétaire | -l -f |
| PGID | N° du groupe de processus | -j |
| SID | N° de session | -j |
| ADDR | Adresse du processus | -l |
| SZ | Taille du processus (en nbre de pages) | -l |
| WCHAN | Adresse d'événements attendus (raison de sa mise en sommeil s'il est endormi) | -l |
| STIME | Date de création | -l |
| CLS | Classe de priorité (TS-> temps partagé, SYS->système, RT->temps réél). | -cf -cl |
| C | Utilisation du processeur par le processus | -f -l |
| PRI | Priorité du processus | -l |
| NI | Valeur "*nice*" | -l |

Un exemple où l'on voit un processus "**zombi**" :

```
$ cat p_zombi.c
/* binaire : p_zombi */
#include   <stdio.h>

main ( ) {
if  ( fork ( ) = = (pid_t)  0 ) {
   printf ( " Fin du processus fils de N°  %d\n", getpid ( ) );
   exit (2);
   }
sleep (30);
}

$ p_zombi  &
Fin du processus fils de N° 1263
[1] 1262
$ ps -l


F    S    UID      PID      PPID   ...  TTY          CMD
1    S    202      364      180         ttyp4        ksh
1    Z    202      1263     1262        ttyp4        <defunct>
1    S    202      1262     364         ttyp4        p_zombi
1    R    202      1264     364         ttyp4        ps



$
```

C'est l'instruction "sleep (30)" qui en endormant le père pendant 30 secondes, rend le fils zombi (état "Z" et intitulé du processus : "<defunct>").

# F. Identificateurs réels et effectifs

A chaque processus sont en fait associés deux groupes d'identifications :

**1. l'UID et le GID *réels* identifient l'utilisateur qui a lancé le processus.**

**2. l'UID et le GID *effectifs* (EUID et EGID), identifient les droits des processus.**

Le plus souvent, les identités réelles et effectives sont identiques. Cependant, il peut être nécessaire de modifier les droits d'un processus.

Le fichier **/etc/passwd** est protégé en écriture. Aucun utilisateur n'a le droit d'écrire dans ce fichier. Cependant, en utilisant la commande **passwd** (*/bin/passwd*), vous écrivez quand même dans ce fichier, puisque vous modifiez votre mot de passe.

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root sys 7100 Aug 22 15:21 /etc/passwd
$ type passwd
passwd is /bin/passwd
$ ls -l /bin/passwd
-r-sr-sr-x 1 root sys 19040 Dec 19 1985 /bin/passwd
$
```

Remarquez que pour le fichier exécutable **/bin/passwd**, **ls** affiche un caractère '*s*' à la place du caractère '*x*' habituel, montrant par là que ce fichier a le mode «*set user ID*» et «*set group ID*».

Dans ce cas, l'EUID et l'EGID du processus sont ceux du propriétaire du fichier exécuté (et non pas ceux de l'utilisateur qui l'a lancé).

Ainsi quiconque lance l'exécutable **passwd**, travaille avec l'UID effectif de *root* et GID de *sys*.

Unix manipule une entité plus légère que le processus que l'on appelle un fil d'exécution (***thread***).

Dans ce cas on passe l'adresse de la fonction que l'on veut exécuter sur une ***thread***.

L'interface POSIX des threads est la suivante :

```
#include <pthread.h>
```

```
/* crée un nouveau thread (comme le "fork" pour un
    processus)  */

int pthread_create(pthread_t *thread,
    pthread_attr_t *attr, void * (*lancer_routine)
    (void *), void *arg);

// termine une thread

void pthread_exit (void *retval);

/* attendre la fin d'un thread (comme "wait" pour les
    processus)  */

int pthread_join (pthread_t th, void **thread_return);
```