Chapitre 2

Les processus

2.1 Introduction

Le processus est un concept clé dans un système d'exploitation. Un processus est un programme en cours d'exécution. C'est-à-dire, un programme à l'état actif. Un processus regroupe un programme exécutable, sa zone de données, sa pile d'exécution, son compteur ordinal ainsi que toutes autres informations nécessaire à l'exécution du programme.

Un seul processus est exécuté à la fois sur un processeur. Comme le processeur commute entre les différents processus, on a une impression de parallélisme. Le compteur ordinal permet de garder en mémoire la prochaine instruction à exécuter.

Il est clair que deux processus peuvent être associés au même programme. Prenons comme exemple deux utilisateurs éditant sous `vi`. Dans ce cas les sections de texte sont équivalentes et les zones de données peuvent différer. Pour expliciter la différence entre processus et programme, prenons cet exemple :

Une maman réalise de la confiture. Pour ce faire, elle dispose d'une recette et des ingrédients nécessaires (mures, framboises, sucre, ...). Ici la recette représente le programme et les ingrédients, les données. La maman joue le rôle du processeur. Le processus est l'activité consistant à lire la recette, trouver les ingrédients et faire cuire la confiture. On suppose maintenant que la fille de la maman vienne interrompre cette activité parce qu'elle est tombé dans les orties. La maman marque alors l'endroit où elle en est dans la recette (l'état du processus est sauvegardé). Elle va chercher un livre médical et soigne sa fille (ce processus est prioritaire). Lorsqu'elle a terminé ces premiers soins, elle reprend son activité de cuisinière là où elle l'avait laissé.

Cet exemple illustre la répartition de tâche sur un même processeur.

2.2 les différents états d'un processus

L'état d'un processus est défini par l'activité courante de celui-ci. Les processus peuvent interagir. Dans la commande shell Unix `who | wc -l`, le premier processus liste les utilisateurs connectés et envoie ses résultats dans un tube. Le second compte le nombre de lignes lui parvenant par le tube. Ces deux processus sont lancés de manière concurrente. Le processus `wc` peut être prêt

à s'exécuter mais il est obligé d'attendre les informations provenant du processus `who`. Dans ce cas le processus `who` se bloque.

Un processus se bloque lorsqu'il ne peut poursuivre son exécution. Chaque processus peut se trouver dans un des états suivants :

Nouveau Le processus est en cours de création.

Élu Le processus est en cours d'exécution sur le processeur.

Éligible (ou prêt) Le processus attend d'être selectionné.

En attente (ou bloqué) Le processus attend qu'un évènement extérieur se produise.

Terminé Le processus a fini son exécution.

On peut représenter le passage d'un état à un autre par le graphe suivant

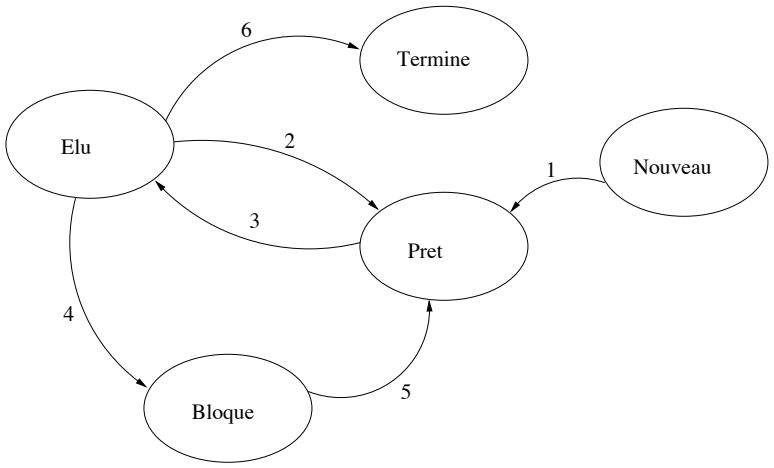


FIG. 2.1 – Graphe de transition des états d'un processus

Chaque transition correspond à un changement d'état. Ces changements peuvent avoir lieu dans les differents cas suivants :

1. Un processus est créé. Son code est chargé. Les données sont initialisées pour l'ordonnanceur.
2. le quantum de temps a expiré, l'ordonnanceur choisit un nouveau processus.
3. L'ordonnanceur choisit ce processus.
4. le processus est en attente d'une ressource.
5. La ressource ou les données deviennent disponible.
6. Le processus sort des files d'attente de l'ordonnanceur.

Il faut aussi mentionner des transitions supplémentaires ayant pour effet de faire éliminer de la mémoire centrale le programme correspondant au processus. Celui-ci retourne alors dans l'état bloqué en mémoire auxiliaire, s'il ne dispose des ressources nécessaires à son exécution, ou prêt en mémoire auxiliaire sinon. Sous Unix un processus particulier appelé le swapper est chargé du va et vient de la mémoire auxiliaire à la mémoire centrale. Son fonctionnement sera détaillé dans le chapitre sur le gestion de la mémoire.

2.3 Structure de données associée au processus

Pour mettre en œuvre le principe de processus, le système gère une table de structures appelée table des processus. Chaque processus y est représenté par une structure de donnée que l'on nomme le bloc de contrôle des processus (BCP). Le BCP contient des informations sur l'état d'un processus. En particulier on y trouve :

- L'état d'un processus,
- le compteur ordinal ou compteur d'instruction qui indique l'instruction suivante devant être exécutée par ce processus,
- les registres de l'UC (pointeur de pile, ...),
- des informations sur le scheduling (ordonnancement)
  - priorité du processus,
  - pointeurs sur file d'attente,
- informations sur la gestion de la mémoire,
- informations de comptabilité (temps CPU utilisé, ...),
- informations sur l'état des Entrée/Sorties,
  - liste des E/S allouées à ce processus
  - liste des fichiers ouverts

Le contenu du BCP varie d'un système à l'autre. Nous prenons ici comme exemple la structure du BCP d'unix :

Gestion des processus	**Gestion de la mémoire**	**Gestion des fichiers**
compteur ordinal	pointeur su le segment code	masque UMASK
mot d'état du programme	pointeur sur segment données	répertoire racine
pointeur de pile	pointeur sur le segment bss	répertoire de travail
état du processus	statut de fin d'exécution	descripteurs de fichiers
date lancement du processus	statut de signal	uid effectif
temps UC utilisé	pid	gid effectif
date de la prochaine alarme	divers indicateurs	divers indicateurs
pointeurs sur file de messages		
bits des signaux en attente		

2.3.1 Le changement de contexte

Considèrons le cas d'Unix. Pour donner l'impression de parallélisme, le processeur commute rapidement les processus. Lors de la commutation, le noyau doit assurer le changement de contexte. Pour ce faire, le noyau doit assurer la sauvegarde du contexte. Cela se produit dans trois cas distincts que nous allons détailler.

1. Le système reçoit une interruption (top d'horloge, disque, terminaux, ...)
2. Un processus exécute un appel système.
3. Le noyau effectue un changement de contexte.

1. Le système a la charge de gérer les interruptions matérielles (horloge, périphériques,...), logicielles ou les exeptions (défaut de page). Si l'UC est en exécution a un niveau d'in-

terruption plus bas que le niveau de l'interruption arrivante, il accepte celle-ci et relève le niveau d'interruption du processeur à celui de la nouvelle interruption de manière à ne pas être interrompu par une nouvelle interruption de niveau inférieur.
   – Le contexte est alors sauvegardé.
   – Il y a recherche de la source de l'interruption et obtention du vecteur d'interruption.
   – La procédure de traitement de l'interruption termine son travail. Le noyau exécute une restauration de contexte. Le processus reprend son exécution.
2. Dans le cas d'un appel système, le noyau doit tout d'abord déterminer l'adresse de celui-ci ainsi que le nombre de paramètres qui lui est nécessaire. Ces paramètres sont copiés dans une zone fixée. Le code de l'appel système est exécuté. Le noyau teste ensuite s'il y a eu une erreur durant l'exécution auquel cas la variable **errno** est positionné. Sinon le noyau renvoie la valeur de retour de l'appel système vers le processus.
3. Il existe différents cas dans lesquels le contexte d'un processus en cours d'exécution doit être sauvegardé.
   – Le processus se met lui même en sommeil (`sleep`).
   – Le processus demande à se terminer (`exit`)
   – Le processus revient d'un appel système et il n'est pas le plus éligible.
   – Le processus repasse en mode utilisateur après une interruption et il n'est pas le plus éligible.

   Pour effectuer un changement de contexte, le noyau doit vérifier l'intégrité de ses données (files proprement chainées, pas de verrous inutiles, ...). Prenons par exemple le cas où le noyau alloue un tampon pour lire un bloc de fichier. Le processus s'endort en attente de fin de transmission de l'Entrée/Sortie depuis le disque. Il laisse le tampon verouillé pour qu'aucun autre processus ne puisse l'altérer.

2.3.2 Le noyau et les appels système

Le **noyau** ou superviseur est l'entité permettant la gestion des processus. Celui-ci regroupe la gestion des E/S, l'ordonnancement des processus, la gestion de la mémoire, la gestion des interruptions. Seul le noyau a des accès directs à toutes les ressources.

Un processus utilisateur souhaitant acquérir un périphérique devra en faire la demande au travers d'un **appel-système**. Un appel-système est une requête traité par le noyau pour le compte d'un processus. Ces notions de noyau et d'appel-système permettent de protéger les données sensibles, celles du noyau, car seul celui-ci peut les modifier. De plus, un appel-système n'est pas interruptible, ce qui fait qu'il n'y a pas de concurrence entre ceux-ci.

2.4 Création et destruction de processus : le cas d'Unix

La création de processus est faite par le système dans la plupart des cas sur demande d'un autre processus. La création d'un processus consiste en l'activation d'un programme. Les différentes actions suivantes sont réalisées lors de la création :

– acquisition d'un bloc de contrôle de processus,
– allocation de la place nécessaire pour le code et les données du nouveau processus,
– chargement de celui-ci,
– complément des différents champs du BCP.

On peut noter que ce processus n'est pas forcément activé immédiatement. Son moment d'activation dépend de l'ordonnanceur. Il existe des liens entre le processus demandant la création d'un autre processus et ce dernier. Le créateur est appelé le père, le nouveau processus, le fils. Un ensemble de mécanismes de communication et de synchronisation est souvent disponible entre eux. La structure des processus est donc dans ce cas arborescente.

La destruction d'un processus se produit en général après la fin de l'exécution du programme correspondant. Il faut alors :
– libérer les ressources détenues par le processus (périphériques, mémoire,...),
– libérer le bloc de contrôle du processus,
– refaire l'allocation du processeur.

Que faire au cas où ce processus a des fils ? Dans le cas d'Unix, tous les fils sont rattachés au processus de numéro 1 qui se nomme *init*.

Un processus peut également être détruit par un autre. Il faut dans ce cas que le processus voulant détruire possède des droits particuliers sur celui à détruire.

2.4.1 Les appels-système Unix

L'identification

Le processus courant est identifié de manière unique dans le système par un numero (**pid**). Des appels-systèmes permettent d'obtenir cet identifiant ainsi que celui de son père.

```
#include<unistd.h>
pid_t getpid (void);          /* identite du processus */
pid_t getppid (void);         /* identite du processus pere */
```

Il existe évidemment des liens entre processus et utilisateur. Un processus appartient à un utilisateur et à un groupe d'utilisateurs. Il possède également un propriétaire effectif et un groupe propriétaire effectif. Ces propriétaires et groupes effectifs déterminent les droits du processus vis à vis des objets du système.

Le propriétaire (resp. groupe) effectif diffère du propriétaire (groupe) réel lorsque le *setuid (setgid)* bit est positionné sur le fichier binaire servant de base au processus. Cette astuce permet dans ce cas au processus d'obtenir les droits du propriétaire (groupe propriétaire) du fichier exécuté.

```
#include<unistd.h>
uid_t getuid(void);          /* proprietaire reel  */
uid_t getgid(void);          /* groupe proprietaire reel  */
uid_t geteuid(void);         /* proprietaire effectif  */
uid_t getegid(void);         /* groupe proprietaire effectif  */
```

Rappelons également que le système Unix crée deux processus qui sont, le *swapper* de numéro 0 et l'*init* de numéro 1. Tous les autres processus sont des descendants de l'*init*.

Par défaut ces processus appartiennent au super-utilisateur. Lors de la connection d'un utilisateur, il faut donc être capable de changer le propriétaire et le groupe propriétaire du nouveau processus. Pour ceci nous disposons des deux primitives suivantes :

```
#include<unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
```

vie et mort d'un processus

La création de processus se fait uniquement par duplication. On peut parler de génétique des processus. Le processus créateur (père) fait appel à la fonction :

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

Cet appel système crée, si cela est possible, un nouveau processus (fils). Le nouveau processus est en tout point identique au père (même code, copie des mêmes données). La seule chose qui va differer est la valeur de retour de l'appel système *fork*. Cette valeur est

- **0** dans le processus fils,
- **l'identité du processus fils** dans le processus père.

Si la primitive échoue, la valeur de retour est -1.

La terminaison d'un processus peut survenir soit à la demande du processus lui même, soit du fait de la délivrance d'un signal de demande d'interruption (voir les signaux). Dans le premier cas on utilise la fonction

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

Cet appel provoque la terminaison du processus avec le code de retour *valeur*. La fonction réalise la libération des ressources utilisées par le processus. Seul subsiste un bloc de contrôle de processus contenant le code de retour. Un processus dans cet état est appelé **zombie**.

Un ensemble de primitives permet d'attendre la mort d'un processus fils et ou de récupérer des informations concernant sa mort.

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

Pour *wait* nous avons trois cas :

- Le processus n'a pas de fils : retour immédiat avec la valeur -1.

– Le processus n'a pas de fils zombie : il est bloqué jusqu'à ce que, soit un de ses fils devient zombie, ou soit l'appel-système est interrompu par un signal.
– Le processus a au moins un fils zombie : la primitive renvoie l'identifiant de l'un de ceux-ci et *exit-status* donne des informations sur sa mort.

La primitive *waitpid* permet de selectionner le processus fils dont on doit récupérer les informations. Nous vous renvoyons au *man* pour plus d'indication.

La création de processus ne se faisant que par duplication, nous avons également besoin d'un mécanisme permettant de charger un nouveau code binaire. Les appels-système de la famille **exec** permettent cette opération par une méthode que l'on appelle recouvrement. En fait lors de l'appel à exec, le code passé en paramêtre est chargé en lieu et place de celui qui exécute cet appel-système.

Il existe six primitives **exec** différentes permettant soit un paramétrage un peu différent, et tenant compte du PATH et ou de l'environnement. Nous en donnons deux ici.

```
int execl(const char *ref, const char *arg0, const char *arg1, ..., NUL
```

Dans ce cas le fichier de nom *ref* est chargé et les paramêtres arg_i sont passés au *main*.

```
int execv(const char *ref, const char *arg[ ]);
```

Cette primitive se comporte de la même façon sauf que les arguments sont cette fois stockés dans un tableau.

2.5 Ordonnancement

L'ordonnancement de l'UC est la base d'un système d'exploitation multiprogrammé. Plusieurs processus peuvent être prêt simultanément à s'exécuter. Le choix du processus à exécuter revient à l'ordonnanceur (scheduleur). La stratégie utilisée dans ce cadre s'appelle l'algorithme d'ordonnancement. Le rôle majeur de l'ordonnancement est de maximiser l'utilisation des ressources. Un bon algorithme d'ordonnancement doit être capable de :

1. s'assurer que chaque processus reçoit sa part du temps processeur,
2. utiliser le temps processeur à 100% (au maximum),
3. minimiser le temps de réponse des processus interactifs,
4. maximiser le nombre de travaux effectués dans un interval de temps.

Les processus ne fonctionnent évidemment pas tous de la même façon. Certains vont être gourmands en temps de calcul alors que d'autres vont être le plus souvent bloqués sur des Entrée/Sorties.

L'ordonnanceur doit prévoir que chaque processus a un comportement unique et imprévisible. Pour s'assurer qu'un processus ne prendra pas tout le temps CPU, chaque ordinateur dispose d'une horloge électronique qui génère une interruption plusieurs fois par seconde. A chaque interruption de l'horloge le noyau reprend la main et décide si le processus doit continuer ou donner la main.

Cette stratégie permettant de suspendre des processus est appelée ordonnancement avec réquisition. Elle s'oppose à la stratégie qui consiste a laisser finir la tâche.

Dans la méthode d'ordonnancement avec réquisition (préemptive), un processus peut être interrompu à n'importe quel moment pour laisser la place à un autre processus, ce qui peut conduire à des conflits d'accès qu'il faut savoir traiter. Ce thème sera traité au chapitre suivant. Les algorithmes d'ordonnancement sans réquisition ne sont pas adaptés à un système multi-utilisateur. Imaginez un utilisateur décidant le calcul exact de $10000!$. La tâche serait si longue qu'aucun autre processus n'aurait jamais le CPU.

Les décisions d'ordonnancement peuvent avoir lieu dans l'une des quatres circonstances suivantes :

1. Un processus commute de l'état en exécution vers l'état en attente (requête d'E/S).
2. Un processus commute de l'état en exécution vers l'état prêt (quantum de temps utilisé).
3. Un processus commute de l'état en exécution vers l'état prêt (terminaison d'une E/S).
4. Un processus se termine.

Dans les circonstances 1 et 4, il n'existe aucun choix en fonction de l'ordonnanceur. Celui-ci doit choisir un nouveau processus. Nous parlerons dans ce cas d'ordonnancement sans réquisition. Dans les cas 2 et 3, l'ordonnanceur peut choisir d'interrompre le processus courant pour allouer le processeur à une autre tâche. Nous parlerons dans ce cas d'ordonnancement avec réquisition.

L'ordonnancement avec réquisition pose le problème de l'actualisation des données. Supposons un processus p_1 en train de mettre à jour des données. Ce processus est interrompu en cours de tâche. Le processus p_2 est élu. Ce dernier va lire les données précédentes qui sont cependant dans un état instable. Des mécanismes de synchronisation sont alors nécessaires. La réquisition affecte également le comportement du noyau. Le noyau peut être réquisitionné au nom d'un processus et peut modifier des données importantes (file d'attente des BCP). Imaginons que celui-ci soit interrompu alors que les données sont dans un état instable. Ceci nous entraine vers la fin du fonctionnement du système. Pour remédier à ce problème, on considère qu'un appel système n'est pas interruptible (fin naturel ou attente d'E/S).

2.5.1 Ordonnancement sans réquisition

Nous allons décrire succintement deux algorithmes d'ordonnancement sans réquisition.

Premier arrivé, premier servi (FCFS)

C'est le plus simple des algorithmes d'ordonnancement. Il peut être mis en œuvre simplement grâce à une file d'attente FIFO. Quand un processus rentre dans la file d'attente des processus prêts, son BCP est enchainé en queue de file. Dès que l'UC est libre, elle est allouée au processus en tête de file.

Examinons l'exemple suivant :

Processus	Temps de cycle
P_1	24
P_2	3
P_3	3

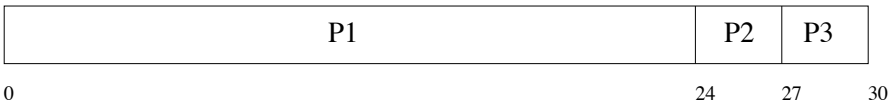


FIG. 2.2 – Diagramme de Gantt représentant le temps d'exécution des différents processus

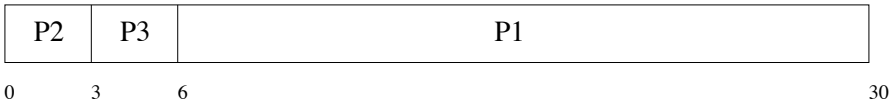
Le temps moyen d'attente est de $(0 + 24 + 27)/3 = 17ms$

On voit ici que le temps d'attente moyen pour l'allocation du processeur à un processus n'est pas optimal.

Le plus court d'abord (SJF)

Dans cet algorithme, on associe à chaque processus la longueur de son prochain cycle. Si deux processus ont la même longueur de cycle alors l'algorithme FCFS est utilisé sinon le processeur est alloué au processus de plus court cycle.

Reprenons l'exemple précédent



Le temps moyen d'attente est cette fois de $(0 + 3 + 6)/3 = 2ms$

Cet algorithme est optimal, mais cependant il est difficile de pouvoir estimer la longueur du prochain cycle d'UC du processus.

2.5.2 L'ordonnancement avec réquisition

Le modèle du tourniquet (RR)

L'un des algorithmes les plus anciens, les plus simples et les plus fiables est celui du tourniquet. On définit une petite tranche de temps appelée quantum. La file d'attente des processus prêts est traitée comme une file FIFO. Chaque processus est donc autorisé à s'exécuter pendant son quantum de temps. Deux cas peuvent alors se produire :

- Le processus utilise entièrement son quantum de temps. Il est donc placé en queue de file et on alloue le processeur au processus en tête de file.
- Le processus se termine ou est bloqué sur une E/S. Il est alors placé dans une file d'attente adéquate et le processeur est alloué au processus en tête de la file des processus prêts.

Le principal problème de ce modèle est de fixer la taille du quantum de temps. Supposons que la tranche de temps soit de $20ms$ et que le temps de commutation du contexte soit de $5ms$. Le processeur travaille alors $20ms$ au profit du processus puis passe $5ms$ pour la commutation soit 20% du temps. On voit ici que la perte de temps processeur est importante. Supposons maintenant que la tranche de temps est de $500ms$. Le temps perdu est cette fois inférieur à 1%, mais cette fois, c'est le temps d'attente moyen qui devient important. Supposons dix processus lancés quasiment simultanément. Dans le premier cas le dixième processus commencera à s'exécuter au bout de $5 \times 25ms = 225ms$. Dans le second cas il faudra attendre $9 \times 505ms$ soit plus de $4,5s$. Ce temps

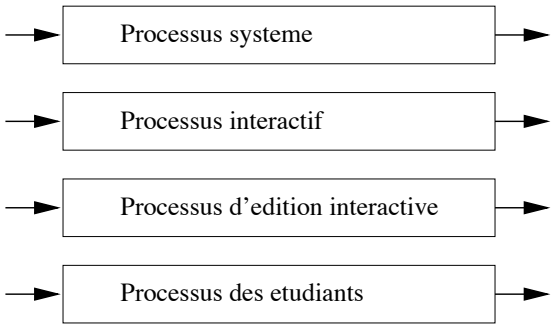
d'attente est trop long pour les utilisateurs. Il faut donc trouver un juste équilibre afin d'obtenir à la fois un temps d'attente convenable et une perte de temps processeur raisonnable.

L'ordonnancement avec priorité

Le modèle précédent présume que tous les processus ont la même importance. On peut considérer que c'est rarement le cas. Considérons le cas d'un processus interactif et d'un processus en arrière plan. Le processus interactif aura besoin d'avoir une priorité plus importante pour répondre rapidement à la demande utilisateur.

Un algorithme d'ordonnancement avec des files d'attente à multiniveaux regroupe les processus prêts selon leur priorité. Chaque file d'attente possède son propre algorithme d'ordonnancement. On doit également avoir un algorithme d'ordonnancement entre les files.

Supposons le schéma suivant :



Chaque file d'attente est absolument prioritaire par rapport à la file immédiatement inférieur. Un processus interactif ne sera élu que lorsque la file des processus système sera vide. Une autre possibilité consiste à assigner des tranches de temps aux files. Chaque file obtient un pourcentage du temps processeur.

Les files multiples avec déplacements

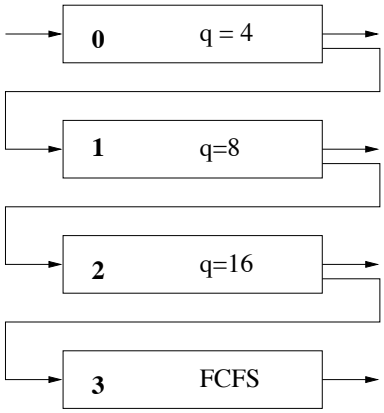
L'idée ici est d'allouer un quantum de temps particulier pour les processus de chaque file. On alloue des quanta de temps plus faible aux files les plus prioritaires. Supposons un ordonnanceur de files multiples à quatre niveaux.

Pour qu'un processus de la file 2 puisse s'exécuter, il faut que les files 0 et 1 soient vides. Un processus entrant dans la file 1 provoquera l'interruption d'un processus de la file 2. Le fonctionnement général est le suivant. Un processus rentre dans la file 0, il dispose donc d'un quantum de temps de $4ms$. Si au bout de ce temps son cycle n'est pas terminé (fin du processus, demande d'E/S), il passe à la queue de la file immédiatement inférieure, ainsi de suite.

Cet algorithme donne donc une priorité importante au processus effectuant beaucoup d'E/S. Les processus longs sombrent automatiquement dans les files les plus basses.

Pour définir ce type d'algorithmes, nous avons besoin des paramêtres suivants :

- le nombre de files d'attente,
- l'algorithme d'ordonnancement pour chaque file,



- la méthode utilisée pour passer un processus à une file supérieure,
- la méthode utilisée pour passer un processus à une file inférieure,
- la méthode déterminant dans quelle file doit rentrer un processus.

Cet algorithme est le plus général et donc le plus puissant mais également le plus difficile à mettre en œuvre.

Exercice corrigé

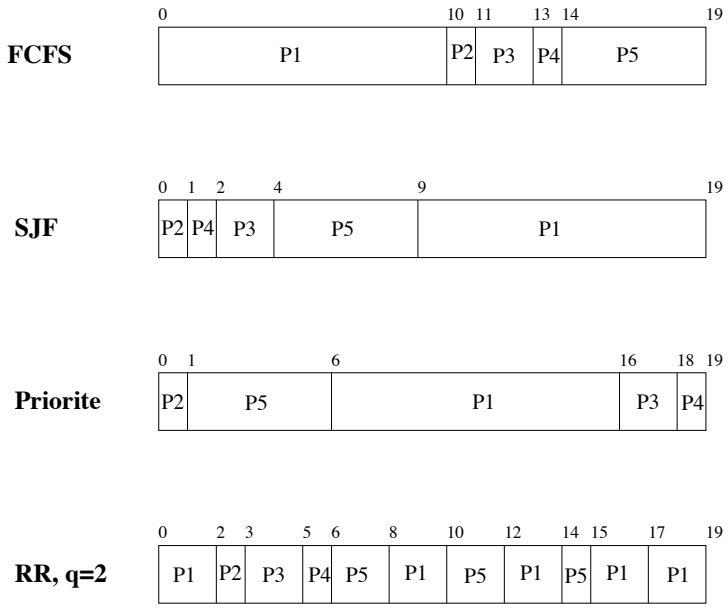
Envisagez l'ensemble suivant de processus, avec la durée du cycle d'UC donnée en millisecondes :

Processus	Temps du cycle	Priorité
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

On suppose que les processus sont arrivés dans l'ordre P_1,P_2,P_3,P_4,P_5, à l'instant 0.

1. Dessinez quatre diagrammes de Gantt illustrant l'exécution de ces processus en utilisant les schedulings FCFS, SJF, une priorité sans réquisition (plus petit numéro indique plus forte priorité) et RR (avec le quantum vaut 2).
2. Quel est le temps moyen de restitution d'un processus pour chacun des algorithmes d'ordonnancement de l'exercice 1 ?
3. Quel est le temps moyen d'attente d'un processus pour chacun des algorithmes d'ordonnancement de l'exercice 1 ?
4. lequel des ordonnancement de l'exercice 1 obtient-il le temps moyen d'attente minimal (sur tous les processus) ?

1. Schéma de Gantt
2. Restitution
   - FCFS = $13,4ms$
   - SJF = $7ms$



- Priorité = $12ms$
- RR = $9,6ms$

3. Attente
   - FCFS = $9,6ms$
   - SJF = $3,6ms$
   - Priorité = $8,2ms$
   - RR = $3,2ms$

2.5.3 L'exemple d'Unix

Sous Unix, les processus sont répartis dans différentes files selon leur priorité. La plage des priorités est partitionnée en deux classes : les priorités utilisateurs et les priorités du noyau. Chaque classe contient plusieurs priorités et chaque priorité est associée à une file de processus qui lui est propre.

Le noyau recalcule régulièrement la priorité de chaque processus. Il met en corrélation un niveau de priorité avec l'attente d'un évènement. Les priorités les plus hautes sont non interruptibles.

Pour les processus du noyau, de simple files d'attente sont utilisées avec possibilité de doubler les processus endormis de la même liste (seuls les processus ayant l'information de la fin de leur E/S seront réveillés).

Au niveau utilisateur, la méthode du tourniquet est utilisée. Les priorités étant recalculées périodiquement, les processus changent de file en fonction de celles-ci.

Appels-système Unix

Il est possible pour un utilisateur de modifier la priorité de son processus

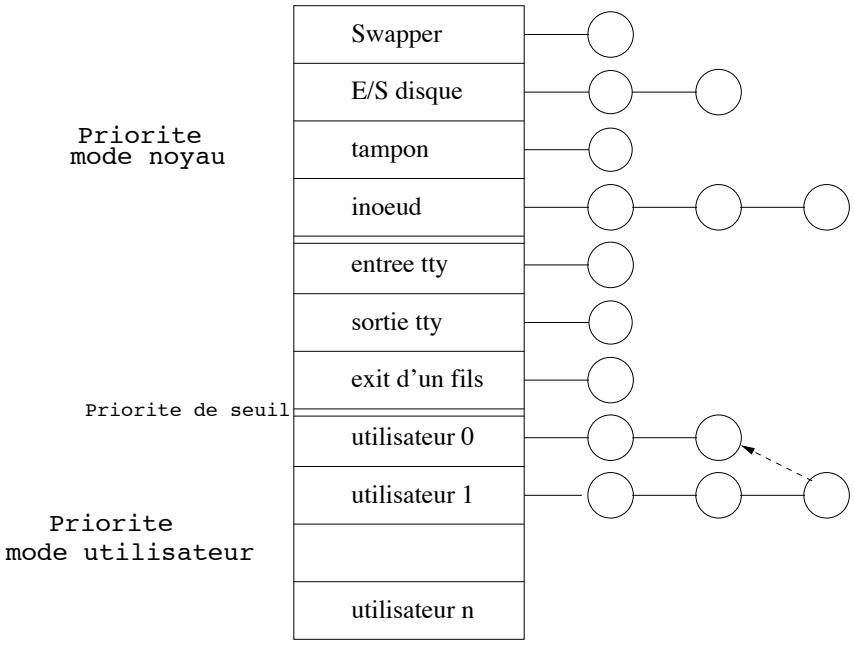


FIG. 2.3 – Files de priorité sous Unix

```
#include <unistd.h>
int nice(int valeur);
```

Seul le super-utilisateur peut se donner une priorité plus importante. Cependant, cet appel ne permet pas à un processus de modifier la priorité d'un autre processus. Ce qui signifie que si un processus est trop ¡¡gourmand¿¿ en CPU, le super-utilisateur n'a pas d'autre choix que de le supprimer.

2.6 La synchronisation des processus

2.6.1 Accès concurrents

Considérons le cas classique du Producteur/Consommateur. Un producteur écrit dans un tampon un ensemble d'objet qu'un consommateur va prendre (tampon d'impression). Cet exemple pose différents problèmes que nous énumérons ici.

- Le tampon est-il vide ?
- Le tampon est-il plein ?
- L'accès au tampon est-il réservé ?

Le pseudo-code pour les processus Producteurs et consommateurs pourrait être le suivant :

Algorithme Producteur	Algorithme Consommateur
repeter	**repeter**
produire un élément dans suivant	**tant que** compteur = 0 **faire**
tant que compteur = n **faire**	rien
rien	suivant⟵ tampon[out]
tampon[in]⟵ suivant	out ⟵ (out +1)
in ⟵ (in +1) mod n	compteur ⟵ compteur -1
compteur ⟵ compteur +1	consommer suivant
jusqu'à faux	**jusqu'à** faux

Dans ce cas nous disposons pour les processus Producteurs et Consommateurs d'une variable partagée `compteur`.

Bien que séparément chacun des deux algorithmes soit correct, ils peuvent ne pas fonctionner ensemble car ils accèdent simultanément à la même variable `compteur` en lecture et en écriture. Regardons ce qui se passe au niveau du fonctionnement machine. L'instruction *compteur* ⟵ *compteur+1* peut être écrite en langage d'assemblage de la manière suivante :

(1) *reg1*[1] ⟵ *compteur*
(2) *reg1* ⟵ *reg1 +1*
(3) *compteur* ⟵ *reg1*

Il en va de même pour l'instruction *compteur* ⟵ *compteur -1*. Les différentes instructions machines peuvent être intercalés au niveau du processeur d'une manière quelconque. Prenons cet exemple :

P	*reg1* ⟵ *compteur*	*[reg1=5]*
P	*reg1* ⟵ *reg1 +1*	*[reg1=6]*
C	*reg2* ⟵ *compteur*	*[reg2=5]*
P	*compteur* ⟵ *reg1*	*[compteur=6]*
C	*reg2* ⟵ *reg2 -1*	*[reg2=4]*
C	*compteur* ⟵ *reg2*	*[compteur=4]*

En sortie nous obtenons la valeur erronée *compteur = 4*. La valeur est incorrecte car nous avons permis aux deux processus de manipuler la variable *compteur* en concurrence. Le but des algorithmes qui suivent va être de résoudre ce genre de problème.

2.6.2 Sections critiques

Lorsqu'il y a partage de la mémoire, des fichiers ou de tout autre objet, il faut trouver un moyen d'interdire la lecture ou l'écriture à plus d'un processus à la fois. Dans l'exemple précédent, l'erreur vient du fait que le processus Consommateur manipule la variable *compteur* alors que le Producteur n'a pas fini de s'en servir. La partie de programme ou se produise les conflits d'accès est appelée **section critique**. Le problème des conflits d'accès serait résolu si on pouvait s'assurer que deux processus ne soient jamais en section critique en même temps.

Pour que le problème de la section critique soit résolu efficacement, nous avons besoin des conditions suivantes :

- Deux processus ne peuvent être en même temps en section critique.
- Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
- Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.
- Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

Exemple sous Unix

Un exemple de zone critique sous Unix est le segment de mémoire partagée. Un segment de mémoire partargée représente une ou plusieurs pages. Les pages partagées vont donc devenir des ressources critiques et les accès devront y être synchronisés.

Ces segments de mémoire partagée font partie d'un ensemble d'outils appelée IPC (Inter-Processus Communication). Il regroupe trois mécanismes qui sont :

- Les segments de mémoire partagée,
- les sémaphores (outil de synchronisation),
- les files de messages (outil de communication).

Chacun des deux autres outils sera détaillé dans la section adéquat.

Ce qu'il faut savoir sur les I.P.C.

- Le système assure la gestion de chacun des objets par l'intermédiaire d'une table spécifique à chacun de ceux-ci.
- Un objet dispose d'un identifiant interne (propre au processus) et un identifiant externe (connu par le système). Pour utiliser un tel objet, un processus devra en connaitre un identifiant interne (héritage ou demande au système par nom externe).
- les I.P.C. utilisents des fonctions qui sont construites selon le même type, les préfixes pour chacun des objets étant différents :

  shm pour les segments de mémoire partagée,

  sem pour les sémaphores,

  msq pour les files de messages.

  Nous disposons donc de deux types de fonctions
  - Un mécanisme de création/ouverture de suffixe **get** (*shmget*),
  - un mécanisme de consultation/modification/suppression de suffixe **ctl** (*msqctl*).

Nous trouvons dans le fichier <`sys/ipc.h`> un certain nombre de définitions de structures et la macro-définition de constantes symboliques utilisées pour la manipulation des I.P.C. Nous en donnons ici un tableau récapitulatif :

Contantes	Rôle	Primitives
IPC_PRIVATE	clé privé, pas de nom externe pour l'objet	get
IPC_CREAT	crée l'objet s'il n'existe pas	get
IPC_EXCL	utilisé avec IPC_CREAT, si l'objet existe déjà alors erreur	get
IPC_NOWAIT	opération non bloquante	
IPC_RMID	suppression de l'IPC	ctl
IPC_STAT	extraction de caractéristiques	ctl
IPC_SET	modification de caractéristiques	ctl

On trouve également dans ce fichier la structure `ipc_perm` regroupant des informations sur le propriétaire et les droits sur le mécanisme I.P.C. utilisé.

Chaque objet de type I.P.C. a une structure carte d'identité qui lui est associé. Voici celle pour le segment de mémoire partagée :

```
struct shmid_ds
{
  struct ipc_perm shm_perm;          /* droits d'acces */
  int shm_segsz;                     /* taille du segment en octet */
  time_t shm_atime;                  /* date du dernier attachement */
  time_t shm_dtime;                  /* date du dernier detachement() */
  time_t shm_ctime;                  /* date du dernier changement par shm
  pid_t shm_cpid;                    /* pid createur   */
  pid_t shm_lpid;                    /* pid ayant fait la derniere operati
  unsigned short int shm_nattch;     /* nombre d'attachements */
};
```

La primitive de création d'un segment est la suivante :

```
#include <sys/shm.h>
int shmget(key_t cle, int taille, int option);
```

Cette fonction renvoie l'identifiant interne d'un segment de mémoire partagée. L'*option* est une combinaison booléenne des constantes symboliques, la `taille` spécifie la taille du segment et `cle` est l'identifiant externe de l'objet.

Une fois le segment créé, il faut pouvoir l'attacher à l'espace d'adressage.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *adr, int option);
```

Cette primitive permet l'attachement du segment d'identifiant interne `shmid` à l'adresse `*adr`. Dans la plupart des cas, on laissera le système choisir l'adresse d'attachement en donnant *NULL* comme paramêtre. Parmi les options on peut vouloir attacher un segment en lecture seul (`option`

= SHM_RDONLY). La valeur de retour est l'adresse d'attachement.

Le détachement s'effectue de la manière suivante.

```
#include <sys/shm.h>
int shmdt(const void *adr);
```

La terminaison d'un processus entraîne le détachement de tous les segments attachés par lui.

L'appel système

```
#include <sys/shm.h>
int shmctl(int shmid, int op, struct shmid_ds *p_shmid);
```

permet les différentes actions suivantes en fonction de la valeur de *op.*

- **IPC_RMID :** Suppression du segment. Celle-ci est différée si le segment est encore attaché par un processus.
- **IPC_STAT :** Remplit la structure `*p_shmid` avec les informations correspondantes.
- **IPC_GET :** Sauvegarde les nouvelles valeursde la structure pointée par `p_shmid`.

2.6.3 Différentes méthodes d'exclusion mutuelle

Nous allons voir dans cette section différents algorithmes garantissant l'unicité de l'accès à la section critique.

Le masquage des signaux

Cette solution permet à un processus de masquer l'ensemble des signaux d'interruption du processeur pendant le temps d'accès à la section critique. Un processus entrant en mémoire partagée ne pourra être interrompu qu'après avoir terminé son accès à celle-ci. Cependant ce procédé révèle de graves lacunes. Tout d'abord, il est extrêmement dangereux qu'un processus utilisateur puisse réquisitionner le processeur à son propre compte. Si par malheur il oublie de démasquer les signaux en sortie alors c'est la fin du système.

Ce procédé peut cependant être utilisé à l'intérieur du noyau pour gérer certains conflits d'accès mais il est inapproprié pour assurer l'exclusion mutuelle en processus utilisateur.

Les verrous

Une solution qui, on le verra, n'en est pas vraiment une consiste à utiliser une variable partagée appelée *verrou.* Un processus désirant entrer en section critique consulte une variable verrou. Si cette variable est à 0, le processus passe celle-ci à 1 et entre dans la section critique. Cette solution n'est pas adéquate car supposons les processus P_0 et P_1 effectuant la séquence suivante :

P_0 lire verrou (verrou=0)
P_1 lire verrou (verrou=0)
P_1 positionner verrou 1
P_0 positionner verrou 1
Les processus peuvent entrer simultanément en section critique.

Une solution pour deux processus

L'alternance

L'alternance consiste pour deux processus à partager une variable *tour* que chaque processus test avant l'entrée dans sa section critique. Si la variable *tour = i* dans le processus P_i alors celui-ci peut entrer en section critique. Quand le processus P_i termine, il bascule cette variable à $1 - i$.

Le problème de cette solution est qu'il requiert une stricte alternance entre les deux processus. Une des conditions de résolution du problème de section critique n'est pas rempli. P_1 peut être prêt à entrer en section critique alors que P_0 est en section restante mais *verrou* vaut 0.

L'algorithme de Peterson

En combinant l'idée du tour de rôle et des variables verrous, Peterson a proposé une solution au problème de l'exclusion mutuelle respectant les quatre contraintes. Les processus partagent deux variables :

```
drap   : tableau[0..1] de booleens
verrou : 0..1
```

Au départ, on a *drap[0]=drap[1]=faux* et *verrou* vaut soit 0 soit 1. La variable *drap* permet à chacun des processus de dire qu'il est prêt à entrer en section critique. Pour que $P_{i(i=0,1)}$ puisse entrer en section critique, il faut que (1) soit ce soit son tour ($verrou = i$), (2) soit que P_{1-i} ne soit pas prêt a y entrer lui même.

L'algorithme est donc le suivant :

Algorithme de Peterson

repeter
drap[i]⟵vrai
tour ⟵ 1-i
tant que drap[1-i] **et**
tour=$1 - i$ **faire**
rien

section critique

drap[i]⟵ faux

section restante

jusqu'à faux

Ainsi le processus P_i affirme avant d'entrer en section critique que c'est le tour de P_{1-i}. Si les deux processus sont prêts en même temps, `verrou` vaudra soit 0 soit 1 sinon, si P_{1-i} est en section restante alors P_i peut entrer en section critique. Montrons que les deux processus ne peuvent pas être en section critique en même temps. Tout d'abord lors de l'exécution de la boucle *tant que*, *drap[i]=drap[1-i]=vrai*, or *tour* vaut soit i soit $1-i$, donc un des deux processus attendra dans cette boucle car les deux conditions du *tant que* sont vraies.

Solution pour plusieurs processus

Nous allons détailler ici un algorithme permettant de résoudre le problème de l'exclusion mutuelle pour n processus. Cet algorithme est souvent connu sous le nom d'algorithme du boulanger. Chaque client entrant dans la boulangerie se voit affecter un numéro. Le prochain client à être servi est celui de plus petit numéro. Malheuresement, on ne peut garantir que deux clients n'obtiendront pas un ticket de même numéro. Dans ce cas, chaque client (processus) possedant lui-même un nom, c'est celui de nom le plus petit qui sera servi.

Under Construction

2.6.4 Un peut d'aide de la part du matériel

Test and set

Nous avons vu précédemment qu'entre la lecture d'une variable et son affectation, un autre processus pouvait intervenir pour lire celle-ci. Ce problème peut être résolu de manière matériel grâce à une instruction atomique Test-and-Set. Cette instruction charge le contenu d'un mot mémoire dans un registre puis affecte une valeur non-nulle (vraie) à ce registre et le réécrit. Cette instruction est indivisible. Le processeur ne peut être requis pour aucune autre tâche entre la lecture et l'écriture.

Si deux instructions Test-and-Set sont exécutées simultanément sur deux processeurs, elles seront en fait exécutées séquentiellement de manière aléatoire quant à l'ordre.

Algorithme d'exclusion mutuelle (Test-and-Set)

repeter

tant que Test-and-Set(verrou) **faire**

rien

section critique

verrou⟵faux

section restante

jusqu'à faux

swap

L'instruction swap échange le contenu de deux booléens de manière atomique. Elle peut être défini comme suit

fonction swap (a,b)
variable temp : **booléen**
début
temp ⟵ a
a⟵ b
b⟵ temp
fin

Cette instruction atomique permet à nouveau d'implanter l'algorithme d'exclusion mutuelle.

Algorithme d'exclusion mutuelle (Swap)
repeter
cle ⟵ vrai
repeter
swap(verrou,cle)
jusqu'à cle = faux

section critique

verrou⟵faux

section restante

jusqu'à faux

Attente limité avec Test-and-Set

Les deux algorithmes précédant ne garantissent cependant pas une attente limitée pour l'entrée en section critique d'un processus. Supposons une file de $n-1$ processus bloqués sur l'instruction *swap*. Si le temps d'exécution de la section restante est inférieur ou égal à un quantum de temps alors l'ordonnanceur choisira parmi $n-1$ processus à débloquer à chaque fois. L'attente risque donc d'être longue. Nous donnons ici un algorithme utilisant *Test-and-Set* et garantissant une attente limitée.

Algorithme d'exclusion mutuelle (Test-and-Set et attente limitée)

variable j :$0..n-1$
cle :**booléen**

repeter
attend[i]⟵vrai
cle⟵vrai
tant que attend[i] et cle **faire**
cle⟵Test-and-Set(verrou)
attend[i]⟵faux

section critique

j⟵i+1 mod n
tant que j≠i et ¬ attend[i] **faire**
j⟵j+1 mod n
si j=i **alors**
verrou ⟵ faux
sinon
attend[j]⟵faux
fsi

section restante

jusqu'à faux

On prouve tout d'abord que le besoin d'exclusion mutuelle est respecté. En effet le premier processus à exécuter *Test-and-Set* trouvera *verrou* à faux. Tous les autres devront attendre. Après la section critique, un processusen débloque un autre, soit en mettant *attend[i]* à faux, soit en mettant *verrou* à faux. En fait *verrou* passe à faux si aucun autre processus n'attend.

L'attente est limité car si plusieurs processus sont en attente, le premier débloqué par P_i sera celui de plus petit numéro supérieur à i modulo n. L'attente sera donc limitée à $n-1$ processus.

2.6.5 L'attente passive

Les différentes solutions présentées précédemment sont correctes mais elles engendrent ce que l'on appelle de l'attente active. Cette attente active est consommatrice de temps CPU. En effet un processus en attente pour entrer en section critique reste éligible. S'il est élu, il y aura une permutation de contexte ainsi qu'une boucle n'effectuant aucune opération. Le processeur n'est donc pas utilisé à bon essient. Cette attente active peut également engendrer des interblocages. Supposons un processus P_1 en attente d'entrer en section critique avec une forte priorité et un processus P_2 en section critique avec une faible priorité. P_1 risque d'être élu, ce qui ne permet pas à P_2 de sortir de sa section critique.

Un solution apparaît évidente. Plutôt que d'attendre de manière consommatrice, le processus peut s'endormir si la section critique n'est pas libre. Celui-ci sera reveillé par le processus abandonnant sa section critique. Les primitives *SLEEP* et *WAKEUP* permettent de réveiller et d'endormir un processus. Celles-ci ont en paramêtre l'évènement permettant de les faire correspondre. Cette solution n'est toujours pas parfaite comme nous allons le voir.

Les primitives SLEEP et WAKEUP

Nous pouvons tout d'abord modéliser le problème de l'exclusion mutuelle grâce à ces primitives.

Algorithme d'exclusion mutuelle (SLEEP/WAKEUP et test-and-set)

repeter
  tant que test-and-set(verrou) **faire**
    sleep()

  section critique

  verrou⟵faux
  wakeup(p_i)

  section restante

jusqu'à faux

Les processus endormis sont stockés dans une file d'attente. Ce schéma fonctionne parfaitement malgré les signaux *wakeup* perdus. Voyons maintenant le problème du Producteur/Consommateur. On peut écrire les algorithmes Producteur et Consommateur de la manière suivante.

Algorithme Producteur

```
repeter
    produire un élément dans suivant
    si compteur = n alors
        sleep()
    tampon[in]⟵ suivant
    in ⟵ (in +1) mod n
    compteur⟵compteur + 1
    si compteur = 1 alors
        wakeup(consommateur)
jusqu'à faux
```

Algorithme Consommateur

```
repeter
    si compteur = 0 alors
        sleep()
    suivant⟵tampon[out]
    out ⟵ (out +1) mod n
    compteur⟵compteur - 1
    si compteur = n-1 alors
        wakeup(producteur)
    consommer suivant
jusqu'à faux
```

Ce schéma peut ne pas fonctionner. On peut se trouver dans la situation suivante : le tampon est vide et le consommateur vient de trouver la valeur à 0 dans *compteur*. L'ordonnanceur décide à ce moment de passer la main au producteur. Celui-ci incrémente *compteur*, trouve *compteur* à 1 et envoie un wakeup vers le consommateur. Le consommateur est réélu, il s'endort. Le producteur va maintenant remplir le tampon puis s'endormir à son tour. Nous arrivons alors dans une situation d'interblocage.

Cette solution est cependant partiellement utilisé sous Unix pour établir la synchronisation, à ceci près que c'est le noyau qui gère les appels à *SLEEP* et *WAKEUP* dans les appels système.

2.6.6 Sémaphores

Les sémaphores ont été inventé par Dijkstra en 1965 pour résoudre les problèmes de synchronisation. Un sémaphore S est composé d'une variable entière et d'une file d'attente. On accède au sémaphore S par les trois opérations P, V et $Init$. Chacune de ces opérations est atomique (elle n'est pas décomposable). On peut représenter un sémaphore par la structure de données suivante :

```
Type semaphore = enregistrement
    val : entier
    file : file d'attente
fin
```

Les algorithmes des opérations d'accès sont les suivants :

```
Init(S : Sémaphore, nombre : entier)
début
    S.val ⟵ nombre
    S.file ⟵ NIL
fin
```

```
P(S : Sémaphore)
début
    S.val ⟵ S.val - 1
    Si S.val < 0 alors
        bloquer(S.file)
    fin si
fin


V(S : Sémaphore)
début
    S.val ⟵ S.val + 1
    Si S.val ≤ 0 alors
        (* extraire un processus de file et
        le mettre parmi les processus prêts *)
    fin si
fin
```

On peut considérer que la fonction *bloquer* endort le processus courant et le place dans la file d'attente lié au sémaphore S. Lorsqu'une opération V est effectué, un processus se trouve débloqué. Il faudra assurer une politique d'ordonnancement correct pour gérer la file d'attente du sémaphore de manière à ce qu'aucun processus ne passe un temps trop long dans cette file (famine).

La valeur d'initialisation correspond au nombre de processus pouvant accéder au sémaphore sans être bloqués. Si $nombre - S.val$ est positif ou nul alors cette valeur correspond au nombre de processus ayant acceder au sémaphore sans se bloquer. Si cette valeur est négative, sa valeur absolue correspond au nombre de processus dans la file d'attente.

Les sémaphores permettent de garantir l'exclusion mutuelle sur une zone critique. Il suffit pour cela que le sémaphore soit initialisé à 1.

Algorithme d'exclusion mutuelle (sémaphore)

```
repeter
    P(S)

    section critique

    V(S)

    section restante

jusqu'a faux
```

Exercice : Résoudre le problème du Producteur/Consommateur avec un tampon de n places grâce aux sémaphores.

- Le tampon devra être en accès exclusif.
- Un objet ne peut être pris dans le tampon que si celui-ci n'est pas vide.
- Un objet ne peut être ajouté dans le tampon que s'il y a une place libre.

Algorithme Producteur	Algorithme Consommateur
repeter	**repeter**
produire un élément dans suivant	P(nb_places_occupees)
P(nb_places_libres)	P(ex_mut)
P(ex_mut)	suivant⟵ tampon[out]
tampon[in]⟵ suivant	out ⟵ (out +1)
in ⟵ (in +1) mod n	V(ex_mut)
V(ex_mut)	V(nb_places_libres)
V(nb_places_occupees)	consommer suivant
jusqu'à faux	**jusqu'à** faux

Au départ, on effectue les initialisations suivantes :

Init(ex_mut,1)
Init(nb_places_libres,n)
Init(nb_places_occupees,0)

Interblocage

L'implantation d'un sémaphore avec une file d'attente peut provoquer une situation où deux processus ou plus attendent indéfiniment un évènement qui ne peut être produit que par un des processus en attente. Quand un tel état est atteint, on dit que ces processus sont dans une situation d'interblocage.

Voici un exemple illustrant un interblocage. On dispose de deux sémaphores S et Q initialisés à 1.

P_0	P_1
P(S)	P(Q)
P(Q)	P(S)
...	...
...	...
V(S)	V(Q)
V(Q)	V(S)

Si P_0 exécute P(S) puis P_1 P(Q) alors aucun des deux processus ne pourra continuer sa course, car P_0 devra débloquer P_1 et réciproquement.

Implantation sous Unix

L'implantation des sémaphores sous Unix est beaucoup plus riche que la simple implémentation des opérations P et V.

- Les opérations P_n et V_n sont implantés. Elles permettent d'obtenir ou de libérer n occurences de la ressource et ceci de manière atomique.
- Unix implante les ensembles de sémaphores. Un processus peut obtenir ou liberer n occurences de la ressource R_1 et m occurences de la ressource R_2. Le modèle s'en trouve considérablement enrichi.
- Cette implantation propose également une nouvelle opération appelée Z qui attend qu'un sémaphore devienne nul.

La structure sémaphore se décrit en deux parties sous Unix. Tout d'abord une structure `__sem` décrivant un sémaphores, puis une structure `semid_ds` décrivant un ensemble de sémaphores. Nous donnons ici la définition de ces structures :

```
struct __sem {   /* egalement la structure dans les anciennes versions Un
        unsigned short int semval;   /* valeur du semaphore */
        unsigned short int sempid;   /* pid du dernier processus utilisat
        unsigned short int semncnt;  /* nombre de processus attendant */
                                     /* l'augmentation du semaphore */
        unsigned short int semzcnt;  /* nombre de processus attendant */
                                     /* la nullite du semaphore */
            };


struct semid_ds {
     struct ipc_perm  sem_perm;  /* structure decrivant les droits */
     struct __sem     *sem_base; /* pointeur sur premier sem de l'ensembl
     time_t           sem_otime; /* date de derniere operation */
```

```
    time_t          sem_ctime; /* date de dernier changement par semctl
    ushort          sem_nsems; /* nb de semaphores de l'ensemble */
       };
```

La primitive de création d'un ensemble de sémaphores est la suivante

```
#include <sys/sem.h>
int semget (key_t cle, int nb_sem, int option);
```

Le fonctionnement de la primitive et le rôle des paramêtres *cle* et *option* sont les mêmes que pour la primitive *shm_get*. La variable *nb_sem* correspond au nombre de sémaphores de l'ensemble.

Une opération sur un sémaphore est décrite grâce à la structure `sembuf`

```
struct sembuf {
     ushort sem_num;   /* numero du semaphore */
     short sem_op;     /* operation sur le semaphore */
     short sem_flg;    /* option */
     };
```

sem_num correspond au numéro du sémaphore dans l'ensemble. La numérotation commence à 0. La valeur du champ *sem_op* détermine la nature de l'opération :

- $n > 0$: Opération V_n. La valeur du sémaphore est augmentéede n. Au plus n processus en attente sur le sémaphore sont réveillés (théorie).
- $n < 0$: Opération P_{-n}. Si l'opération n'est pas possible, le processus est bloqué. Si l'opération est possible, le sémaphoreest diminué de n. S'il devient nul, tout les processus en attente de la nullité sont réveillés.
- $n = 0$: Opération Z. Si le sémaphore n'est pas nul, alors le processus est bloqué.

La variable *sem_flg* représente une conjonction des options possibles. Parmi les options, nous distinguons les deux cas suivants :

- **IPC_NOWAIT** qui rend l'opération non bloquante (*man* pour plus de détails),
- **SEM_UNDO** qui annule l'effet de l'opération en cas de terminaison intempestive du processus.

Ces opérations peuvent être utilisées grâce à la commande

```
#include <sys/sem.h>
int semop (int semid, struct sembuf *tab, int n);
```

Dans ce cas *semid* est l'identificateur de l'ensemble de sémaphores, *tab* est un tableau d'opérations et *n* le nombre des opérations de ce tableau à prendre en compte.

L'ensemble des opérations est réalisé atomiquement. Si une opération ne peut être réalisée, alors aucune opération n'est effectuée et le processus est mis en sommeil.

L'opération de contrôle sur un ou plusieurs sémaphores d'un ensemble est la suivante :

```
#include <sys/sem.h>
int semctl (int semid, int semnum, int op , ... /*arg */);
```

Nous retrouvons tout d'abord les opérations classiques sur les I.P.C. Dans ce cas la variable *op* vaut :

- **IPC_RMID :** L'ensemble de sémaphores identifié par *semid* est supprimé.
- **IPC_STAT :** L'adresse d'une structure de type *semid_ds* doit être passé en argument pour en extraire un contenu.
- **IPC_SET :** Comme précédemment, mais cette fois, l'entrée dans la table des sémaphores est ajustée. Dans ce cas et le précédent *semnum* n'a pas de signification.

Nous trouvons également des cas spécifiques aux sémaphores. Ces cas ainsi que l'interprétation des différents paramêtres sont regroupés dans le tableau 2.1.

Valeur de *op*	**Interprétation de** *semnum*	**Interprétation de** *arg*	**Effet**
`GETNCNT`	numéro d'un sémaphore	–	Nombre de processus en attente d'augmentation du sémaphore
`GETZCNT`	numéro d'un sémaphore	–	Nombre de processus en attente d'augmentation du sémaphore
`GETVAL`	numéro d'un sémaphore	–	Valeur du sémaphore
`GETALL`	nombre de sémaphores	tableau d'entiers courts non signés	Le tableau *arg* contient les valeurs des *semnum* premiers sémaphores
`GETPID`	numéro d'un sémaphore	–	Identification du dernier processus ayant réalisé une opération sur le sémaphore
`SETVAL`	numéro d'un sémaphore	entier	Initialisattion du sémaphore à *arg*
`SETALL`	nombre de sémaphores	tableau d'entiers courts non signés	Initialisation des *semnum* premiers sémaphores au contenu de *arg*

TAB. 2.1 – valeurs de *op* spécifiques aux sémaphores dans `semctl`

2.6.7 Les moniteurs

Nous avons vu précédemment que les sémaphores pouvaient résoudre le problème de l'exclusion mutuelle mais qu'il y avait en cas de problème de programmation ou de synchronisation des risques d'interblocage.

Pour faciliter l'écriture de programme corrects, HOARE et BRINCH-HANSEN ont proposés une structure de synchronisation de plus haut niveau appelée moniteur. Cette structure regroupe les variables partagées ainsi que les fonctions qui les manipulent. Les moniteurs ont une propriété essentielle leur permettant d'assurer l'exclusion mutuelle : Seul un processus à la fois peut être actif dans un moniteur. L'interêt principal des moniteurs vient du fait que c'est au compilateur de s'assurer qu'un seul processus à la fois exécute le code du moniteur. Si un processus demande à

entrer dans un moniteur alors qu'il existe déjà un processus en cours d'exécution de celui-ci alors le premier est mis en attente.

La structure d'un moniteur est celle d'une procédure sans paramêtre et comporte :
- La déclaration de variables partagées accessible uniquement à l'intérieur du moniteur.
- Des procédures et des fonctions internes au moniteur manipulant les variables partagées. Ces fonctions correspondent aux sections critiques.
- Un corps comportant l'initialisation des variables partagées.

Cependant, il faut encore enrichir les moniteurs de manière à non seulement garantir l'exclusion mutuelle mais également garantir un accès conditionnel à certaines variables. Par exemple dans le cas du problème du Producteur/Consommateur, il faut garantir qu'aucun nouvel objet ne pourra être placé dans le tampon si celui-ci est plein.

Pour obtenir la puissance d'expression des sémaphores, on définit un type condition. Des variables de ce type peuvent être déclarées à l'intérieur d'un moniteur. Sur ces variables seules deux opérations peuvent être réalisées :
- L'opération *wait* permet de bloquer un processus et provoque la libération de l'accès au moniteur.
- L'opération *signal* permet de réactiverun processus qui s'était bloqué par un *wait*. Si aucun processus n'était bloqué, l'opération est sans effet.

Il y a cependant un problème quant à l'opération *signal*. En effet, deux processus ne peuvent se trouver simultanément dans le moniteur, or cette opération libère un processus. Deux solutions sont alors proposées :
- Le processus ayant effectué l'opération *signal* se bloque s'il a liberé un processus (HOARE)
- L'opération *signal* est la dernière opération effectuée par le processus dans le moniteur (BRINCH-HANSEN).

Pour plus de simplicité nous retiendrons la deuxième solution.

L'implantation des moniteurs est effectué dans de rare langage (*concurrent pascal*). Nous laissons à titre d'exercice l'implantation du moniteur grâce au sémaphores en C.

Nous donnons ici l'exemple du Producteur/Consommateur résolu grâce aux moniteurs.

```
moniteur ProducteurConsommateur
    condition plein,vide ;
    entier compteur,in,out ;

    procedure mettre
    début
        si compteur = N alors
            wait(plein)
        fin si
        tampon[in]⟵suivantP
        in⟵in+1 mod n
        compteur⟵compteur + 1
        si compteur = 1 alors
            signal(vide)
        fin si
    fin

    procedure retirer
    début
        si compteur = 0 alors
            wait(vide)
        fin si
        suivantC⟵tampon[out]
        out⟵out+1 mod n
        compteur⟵compteur - 1
        si compteur = N-1 alors
            signal(plein)
        fin si
    fin

    compteur=0
    in = 0
    out = 0
fin moniteur
```

```
procedure producteur
début
    repeter
        produire un objet dans suivantP
        ProducteurConsommateur.mettre
    jusqu'a faux
fin

procedure consommateur
début
    repeter
        ProducteurConsommateur.retirer
        consommer suivantC
    jusqu'a faux
fin
```

2.7 Communication inter-processus

Il existe une étroite corrélation entre synchronisation et communication. Certains mécanismes de communication peuvent être utilisés à des fins de synchonisation. Nous verrons principalement dans cette section l'utilisation de messages ainsi que leur implantation sous Unix. Nous donnerons

également les fondements théoriques et l'utilisation du mécanisme de communication par signaux relativement pauvre par rapport à la communication par messages.

2.7.1 Les messages

La fonction d'un système de messages est de permettre aux processus de communiquer entre eux sans avoir recours à des variables partagées. On évite ainsi le problème de la section critique. Une fonction de ce type propose au moins deux primitives qui sont :

- *envoyer (message)*
- *recevoir (message)*

  L'implantation d'un système de messages amène de nombreuses questions
  - Comment sont établies les liaisons ?
  - Est-ce qu'une liaison peut-être associée à plus d'un processus ?
  - Quelle est la taille des messages (taille variable, taille fixe) ?
  - La liaison est elle unidirectionnelle ou bidirectionnelle ?
  - La communication est elle directe ou indirecte ?
  - La communication est elle symétrique ou assymétrique ?
  - Les messages sont ils bufferisés ?

Pour communiquer, les deux processus doivent pouvoir se nommer entre eux. Ils peuvent utiliser la communication direct (type liaison téléphonique) ou la communication indirecte (type poste).

Communication directe

Dans le schéma de la communication directe, les processus désirant comuniquer sont explicitement nommés. Les primitives *envoyer* et *recevoir* sont alors définies ainsi :

- *envoyer (P_1,message)*
- *recevoir (P_2,message)*

Dans ce schéma la communication a les propriétés suivantes :
- Une liaison est affectée à exactement deux processus.
- Il existe au plus une liaison entre chaque paire de processus.
- La liaison est en général bidirectionnelle.
- Chaque processus voulant communiquer n'a besoin de connaitre que l'identité de celui avec lequel il veut communiquer.

Nous pouvons donner avec ce schéma et dans le cas d'une bufferisation une solution au problème du Producteur/Consommateur.

Si aucun élément n'a encore été produit, alors la réception du message par le consommateur ne peut se faire et le processus est bloqué jusqu'à ce qu'un message soit envoyé.

Algorithme Producteur

repeter
    produire un élément dans suivantP
    envoyer (consommateur, suivantP)
jusqu'à faux

Algorithme Consommateur

repeter
    recevoir (producteur, suivantC)
    consommer suivantC
jusqu'à faux

Dans ce cas, l'adressage est symétrique (les processus Producteur et Consommateur doivent se connaitre). On peut également proposer une solution asymétrique auquel cas seul l'émetteur connait son destinataire et le récepteur récupère le nom de l'émetteur en plus du message.

- *envoyer (P,message)* : Envoie du message vers le processus P.
- *recevoir (id,message)* : Reçoit un message d'un processus quelconque dont l'identificateur est récupéré dans `id`.

Le schéma de la communication directe pose le problème de la portabilité. Si un processus P_1 veut changer de destinataire, alors il faut vérifier toutes les occurences d'envoi et de récupération de messages pour remplacer l'ancien destinataire par le nouveau.

Communication indirecte

Dans le cas de la communication indirecte, les messages sont envoyés et reçus dans des boites-aux-lettres. L'objet boite-aux-lettres possède alors un identificateur unique. Dans ce schéma, deux processus peuvent communiquer à partir du moment où ils partagent au moins une boite-aux-lettres.

Les primitives de communication sont alors définis ainsi :

- *envoyer (A,message)* : Envoie un message dans la boite-aux-lettres A.
- *recevoir (A,message)* : Retire un message de la boite-aux-lettres A.

Dans ce schéma la communication a les propriétés suivantes :
- Une liaison n'est établie entre deux processus que s'ils partage une boite-aux-lettres.
- Il peut exister plusieurs liaisons (boites-aux-lettres) entre deux processus.
- Une boite-aux-lettres peut être paratagée par plus de deux processus.
- La liaison peut-être soit unidirectionnelle, soit bi-directionnelle.

Dans ce schéma, un problème se pose : *Soit un processus P envoyant un message dans une boite-aux-lettres A, et soient Q et R, deux processus faisant une opération recevoir simultanée. Qui reçoit le message ?*

A ce problème, on peut apporter plusieur solutions.
- Un seul *recevoir* peut être exécuté à la fois.
- Le système choisit arbitrairement le processus récepteur.

On peut distinguer plusieurs types de boites-aux-lettres. Une boite-aux-lettres peut être un objet appartenant au processus. Dans ce cas, le processus déclare une variable de type boite-aux-lettres et peut recevoir des messages depuis celle-ci. Tout autre processus connaissant un identifiant de

cette boite-aux-lettres peut émettre (s'il en a bien entendu le droit) des messages vers celle-ci. Le processus créateur est alors le propriétaire de la boite-aux-lettres. Une boite-aux-lettres peut également appartenir au système auquel cas elle est indépendante et existe par elle-même. Le système doit alors fournir des mécanismes de gestion pour celle-ci.

- Création d'une boite-aux-lettres.
- Suppression d'une boite-aux-lettres.
- Envoie et reception de messages.

Dans ce schéma, le processus créateur est alors le propriétaire. Il est au départ le seul à pouvoir émettre et recevoir. Il peut cependant déléguer ces droits à d'autres processus. Par héritage, sa descendance reçoit également les droits d'émission et de reception sur cette boite-aux-lettres.

Au cas où plus aucun processus ne peut acceder à la boite-aux-lettres, le système devra être capable de libérer la place prise par celle-ci grâce à un système de ramasse-miettes (garbage collector). Ce principe de boite-aux-lettres est celui implanté sous Unix.

Implantation sous Unix

L'implantation sous Unix est appelé file de messages. Elle fait parti des mécanismes I.P.C. dont nous avons déjà vu deux exemples (mémoire partagée et sémaphores).

La définition sous Unix obeït à ces différentes règles :

- Les files de messages sont des objets système.
- Pour communiquer, deux processus doivent partager une file de messages.
- Il peut exister plusieurs files de messages entre deux processus.
- Une file de messages peut être paratgée par plus de deux processus.
- Il existe une différenciation à l'intérieur d'une même file de plusieurs type de messages.

La structure d'une file de message est la suivante :

```
struct msqid_ds {
    struct ipc_perm  msg_perm;    /* droits d'acces \'a l'objet */
    struct __msg     *msg_first;  /* pointeur sur le premier message  */
    struct __msg     *msg_last;   /* pointeur sur le dernier message  */
    u_short          msg_qnum;    /* nombre de messages dans la file */
    u_short          msg_bytes;   /* nombre maximum d'octets */
    pid_t            msg_lspid;   /* pid du dernier processus emetteur */
    pid_t            msg_lrpid;   /* pid du dernier processus recepteur *
    time_t           msg_stime;   /* date de derniere emission (msgsnd) *
    time_t           msg_rtime;   /* date de derniere reception (msgrcv)
    time_t           msg_ctime;   /* date de dernier changement (msgctl)
    u_short          msg_cbytes;  /* nombre total actuel d'octets */
  };
```

Un message est une structure regroupant au minimum deux champs. Le premier est le type du message. Ce type n'a aucun sens pour le système. Il sert juste à pouvoir partitionner les messages dans une file. Les autres champs forment le corps du message. Le système donne une définition générique d'un message.

```
struct msgbuf {
    long mtype;    /* type du message  */
    char mtext[1]; /* texte du message */
  };
```

mais chaque utilisateur devra définir sa propre structure sur cette base. Le type du message est un nombre entier strictement positif.

La primitive de création d'une file de messages est la suivante.

```
#include<sys/msg.h>
int msgget(key_t cle, int option);
```

Le fonctionnement de la primitive et le rôle des paramêtres est le même que pour la primitive *shmget*.

L'envoi d'un message se fait grâce à l'appel-système :

```
#include<sys/msg.h>
int msgsnd(int msgid, const void *p_msg, int lg, int option);
```

Cette primitive a pour effet d'envoyer dans la file identifié par *msgid* le message pointé par *p_msg*. Le texte de ce message est sur une longueur *lg*. L'envoi dans une file est bloquant si la file est pleine sauf si le paramêtre *option* est positionné à la valeur **IPC_NOWAIT**.

L'extraction dans une file de message se fait grâce à la primitive :

```
#include<sys/msg.h>
int msgrcv(int msgid, void *p_msg, int taille, long type, int option);
```

Cet appel-système lit un message dans la file identifiée par *msgid* et le place à l'adresse *p_msg*. Le texte du message doit être de longueur inférieure ou égale à *taille*. Le paramêtre *option* est une combinaison des deux valeurs booléennes suivantes :

- **IPC_NOWAIT :** La lecture est non bloquante
- **MSG_NOERROR :** Si le texte du message à extraire est de taille supérieure à *taille*, le message est alors tronqué.

Le paramêtre type permet de spécifier le type du message à extraire

$type = 0$ **:** Le message le plus ancien est extrait.

$type > 0$ **:** Le message le plus ancien du type spécifié est extrait.

$type < 0$ **:** Le message le plus ancien de *type* $\leq |type$ est extrait.

La valeur renvoyée est la longueur du texte du message qui a été lu dans la file.

L'opération de contrôle sur une file de messages est la suivante :

```
#include<sys/msg.h>
int msgctl(int msgid, int op, .../* struct msqid_ds *p_msqid */);
```

Nous retrouvons ici les trois opérations classiques sur les I.P.C. Selon que *op* vaut :

- **IPC_RMID :** La file de message est supprimé.
- **IPC_STAT :** L'adresse d'une structure de type *msqid_ds* doit être passé en argument pour en extraire le contenu.
- **IPC_SET :** Comme précédemment, mais cette fois, l'entrée dans la table des files de messages est ajustée.

2.7.2 La gestion des signaux sous Unix

Définition

Un signal peut être vu comme une sonnerie annonçant un évènement. Il existe bien entendu plusieurs type de sonnerie. On peut également les voir comme des interruptions logicielles.

Mécanisme

L'étude des mécanismes de gestion des signaux va nous permettre de répondre aux questions suivantes :

1. Quelles informations un signal véhicule-t-il ?
   L'information véhiculé va dépendre d'un numéro de signal. Nous donnons ici une liste non exhaustive du type d'info véhiculé.
   - Signaux ayant rapport avec la fin d'un processus.
   - Signaux ayant rapport avec des exeptions (division par zéro, violation de l'espace d'adressage,...).
   - Signaux provoquéspar une situation innatendus lors d'un appel-système (*lseek* avec mauvais offset).
   - Signaux utilisateur (alarm).
   - signaux d'interaction avec le terminal (ctrl C, ...).
2. Comment les signaux sont ils émis ?
   Les signaux sont émis grâce à l'appel-système *kill*. C'est donc le noyau qui gère l'émission des signaux. Pour ce faire, il positionne dans le champ signal du BCP du recepteur un bit à 1 (correspondant au type du signal reçu). Si le processus est endormi avec une priorité interruptible, il est alors réveillé par le noyau. Attention, un processus ne peut pas stocker plusieurs

occurence du même signal non encore traité. Ceci est lié à la faiblesse de la structure de données.

3. Quand les signaux sont ils pris en compte ?
   Le noyau examine la reception d'un signal quand un processus est sur le point de passer du mode noyau au mode utilisateur. Un signal n'a donc pas un effet instantanée sur un processus.
4. Comment réagissent les processus à la prise en compte du signal ?
   Il y a trois cas de traitement des signaux.
   – Le signal provoque un *exit* du processus qui le reçoit.
   – le signal est ignoré par le processus.
   – Une fonction utilisateur est exécutéeà la prise en compte du signal.
   L'action par défaut est d'appeler *exit* en mode noyau.

Structure de données

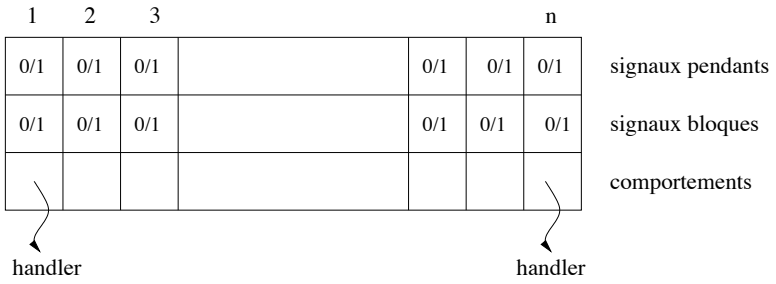


FIG. 2.4 – Structure de données des signaux dans le BCP

– Un signal est **pendant** s'il a été envoyé à un processus mais pas encore pris en compte par celui-ci.
  Exemple : On sonne à votre porte mais tant que vous n'ouvrez pas, le signal n'est pas pris en compte.
– Un signal est délivré à un processus lorsque le processus le prend en compte. Si un signal est émis alors qu'il en existe un exemplaire pendant, il est perdu. Lors de la délivrance, le bit est basculé de 1 à 0.

Les primitives de la norme POSIX pour la manipulation des signaux

Sur un système, il existe *NSIG* signaux différents. Chacun est identifié par un nombre ou une constante symbolique. Le fichier de définition est <`signal.h`>. La primitive d'envoi d'un signal est la suivante :

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

Cette appel-système envoi le signal identifié par le numéro *sig* au processus de numéro *pid*. Si *sig* vaut 0, l'existance du processus *pid* est testé. Pour qu'un signql puisse être envoyé, il faut, soit que le processus emetteur et recepteur aient même propriétaire, soit que le processus emetteur ait pour *euid* celui de root.

A chaque signal est associé une fonction (handler) par défaut (*SIG_DFL*). Cette fonction définit le comportement par défaut à reception de ce signal. Le comportement est parmi :

- Terminaison du processus (avec ou sans vidage mémoire).
- Ignorance du signal.
- Suspension du processus.
- Reprise du processus.

On peut attacher un handler particulier à chaque type de signal (sauf pour les signaux KILL, STOP et CONT). Ces handler sont de deux types :

- **SIG_IGN** permet l'ignorance du signal.
- fonction définit par l'utilisateur. Ces fonctions sont de type void et reçoivent comme paramêtre le numéro du signal.
  `void hand(int sig);`

La norme POSIX permet la manipulation d'ensemble de signaux. On peut par exemple ajouter une type de signal à un ensemble, en retirer un, les ajouter tous.

fonction	effet
`int sigemptyset(sigset_t *p_ens);`	$*p\_ens = \emptyset$
`int sigfillset(sigset_t *p_ens);`	$*p\_ens = \{1, \ldots, NSIG\}$
`int sigaddset(sigset_t *p_ens, int sig);`	$*p\_ens = *p\_ens \cup \{sig\}$
`int sigdelset(sigset_t *p_ens, int sig);`	$*p\_ens = *p\_ens \setminus \{sig\}$
`int sigismember(sigset_t *p_ens, int sig);`	$sig \in *p\_ens$

TAB. 2.2 – primitives de manipulation d'ensemble de signaux

Il existe une primitive de masquage des signaux. Cette appel-système permet une prise en compte ultérieure de ceux-ci. Un processus ne se verra délivré les signaux que lorsqu'ils auront été démasqués.

```
#include <signal.h>
int sigprocmask (int op, const sigset_t *p_ens, sigset_t *p_ens_ancien)
```

Cette primitive effectue l'opération suivante. Un ensemble de types de signaux sont masqués dépendant du paramêtre *op*.

• **SIG_SETMASK :** Les signaux définit dans $*p\_ens$ sont masqués.

• **SIG_BLOCK :** Les signaux définit dans $*p\_ens \cup *p\_ens\_ancien$ sont masqués.

• **SIG_UNBLOCK :** Les signaux définit dans $*p\_ens\_ancien \setminus *p\_ens$ sont masqués.

On récupère dans la variable $*p\_ens\_ancien$ le masque tel qu'il était avant l'appel à la fonction.

On peut également obtenir la liste des signaux pendants qui sont bloqués (masqués). Ces signaux sont ceux qui ont été reçu par le processus mais qui ne seront traités qu'ultérieurement (après démasquage).

```
#include <signal.h>
int sigpending (sigset_t *p_ens);
```

Ceux-ci sont placés dans $*p\_ens$.

Comme nous l'avons vu, il est possible que l'utilisateur définisse sa propre fonction pour le comportement à reception d'un type de signal. La manipulation des handlers se fait grâce à la structure *sigaction*.

```
struct sigaction{
  void (*sa_handler)(); /* SIG_IGN, SIG_DFL ou pteur comme defini ici */
  sigset_t sa_mask;     /* signaux supplementaire a masquer lors de
                           l'execution du handler        */
  int sa_flags;         /* option (pas a la norme POSIX */
};
```

L'ensemble *sa_mask* correspond à l'ensemble des signaux masqués lors de l'exécution du handler. Le signal lui même est toujours ajouté à cet ensemble. On utilise ensuite pour positionner ce handler la fonction :

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *p_action, struct sigact
```

Le handler pointé par *p_action* est positionné, et l'ancien handler est récupéré à l'adresse *p_anc_action*.

La dernière fonction que nous présentons dans cette section permet de réalisé atomiquement plusieurs opérations.

```
#include <signal.h>
int sigsuspend(const sigset_t *p_ens);
```

- Cette opération installe le masque pointé par *p_ens*.
- Le processus courant est mis en sommeil jusqu'à l'arrivée d'un signal non bloqué.

Au retour (après le réveil du processus), le masque d'origine est réinstallé.