Cours « système d'exploitation »
2ème année
IUT de Caen, Département d'Informatique
(François Bourdon)

Chapitre 6

Communication entre Processus :

les IPC

Plan

6.1 La gestion des clés

Les IPC (*Inter Process Communication*) UNIX System V représentent trois outils de communication entre des processus situés sur une même machine.

- les segments de mémoire partagée,
- les files de messages,
- les sémaphores.

Ces mécanismes sont en dehors du SGF => il existe une **table système par type d'objet**. On ne désigne donc pas ces objets par des descripteurs. On ne peut pas rediriger les E/S standards d'un processus sur un objet de ce type.

Apport de fonctionnalités nouvelles et augmentation des performances en matière de partage d'objets.

Dispositif **interne** (au système) et **externe** (clé) de nommage.

Pour partager un objet IPC, les processus partagent la **clé externe** qui lui est associée et utilisent les méthodes propres à chaque type d'objet IPC (notion de "classe d'objets").

La commande **ipcs** permet de consulter les tables systèmes, alors que la commande **ipcrm** supprime une entrée de la table :

```
$ ipcs
IPC status from /dev/kmem as of Tue Oct 20 08:56:30 1998
T     ID          KEY         MODE        OWNER    GROUP
Message Queues:
q     100         0x00000000  --rw-------  root     info
q     51          0x00000000  --rw-------  root     info
q     2           0x49179e95  --rw-rw-rw-  root     root
q     155         0x00000000  --rw-rw----  jmr      ens
Shared Memory:
m     0           0x41440014  --rw-rw-rw-  root     root
m     1           0x41442041  --rw-rw-rw-  root     root
Semaphores:
s     0           0x41442041  --ra-ra-ra-  root     root
s     1           0x4144314d  --ra-ra-ra-  root     root
$
```

où :

T est le type de l'objet (***q*** pour **f.d.m.**, ***m*** pour **s.m.p.** et ***s*** pour sémaphore),

ID est l'identification interne de l'objet,

KEY (valeur hexadécimale) est la clé de l'objet (identification externe) qui identifie l'objet de manière unique au niveau système,

MODE représente les droits d'accès à l'objet,

OWNER et **GROUP** représentent respectivement l'identité du propriétaire de l'objet et l'identité du groupe propriétaire de l'objet.

Chaque objet IPC possède :

- Un **identificateur interne** (ID) équivalent pour les fichiers aux descripteurs dans la table des fichiers ouverts de chaque processus.
- Une **clé externe** équivalente aux références (noms symboliques) pour les fichiers.

Pour qu'un processus utilise un objet IPC, il doit connaître son ID. Pour cela il utilise la ***clé externe*** associée à cet objet.

La fonction **ftok** permet à l'utilisateur de créer ses propres clés, en reliant l'espace de nommage des objets IPC à celui du système de gestion des fichiers.

Une clé unique va être créée à partir d'une référence de fichier et d'une constante.

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t  ftok  (char pathname, char proj) ;
                          /* ou "int proj"  */
```

ce paramètre permet pour un même nom de fichier du SGF d'UNIX de générer plusieurs clés différentes.

La clé est calculée par la fonction suivante (opérations bit à bit) :

Cle = (st_ino & 0xFFFF) | ((st_dev & 0xFF) << 16) | (proj << 24)

Ce calcul est réalisé par une combinaison entre le numéro de l'i–noeud du fichier, le numéro du périphérique sur lequel se trouve le fichier et le dernier paramètre (proj), de telle manière qu'un nombre unique soit généré.

La fonction **ftok** utilise en fait l'identité interne du fichier donné en paramètre. Ceci implique en particulier que :

- la référence utilisée doit être celle d'un fichier existant ;
- si un fichier est physiquement déplacé (changement de i−noeud) entre deux appels à la fonction **ftok**, la clé fournie par les deux appels sera différente.

Exemple :

```
$ cat generer_cle.c
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
struct stat buf;
main(int argc, char * argv[ ]){
   printf("nom du fichier : %s\n", argv[1]);
   if(stat(argv[1], &buf)==−1) {
     printf("fichier %s inexistant\n", argv[1]);
     exit 0;
   }
   printf("numero d'i−noeud : %x\n",buf.st_ino);
   printf("cle associee au numero 0 :
                              %x\n",(int)ftok(argv[1], 0));
   printf("cle associee au numero 1 :
                              %x\n",(int)ftok(argv[1], 1));
}
$
```

```
$ generer_cle   cle
nom du fichier : cle
numero d'i−noeud : 58c1
cle associe au numero 0 : 4458c1
cle associe au numero 1 : 14458c1
$ mv   cle2   cle
$ generer_cle   cle
nom du fichier : cle
numero d'i−noeud : 58c3
cle associe au numero 0 : 4458c3
cle associe au numero 1 : 14458c3
$
```

ici le fichier a été déplacé et a donc changé de i-noeud ; la clé fournie est donc différente !!!

6.2 Les segments de mémoire partagée

Lors de l'échange de volumes importants de données (fichier, tube ou file de messages), il faut recopier les informations échangées dans un tampon système ; d'où un passage en mode noyau du processus à l'origine de cet échange.

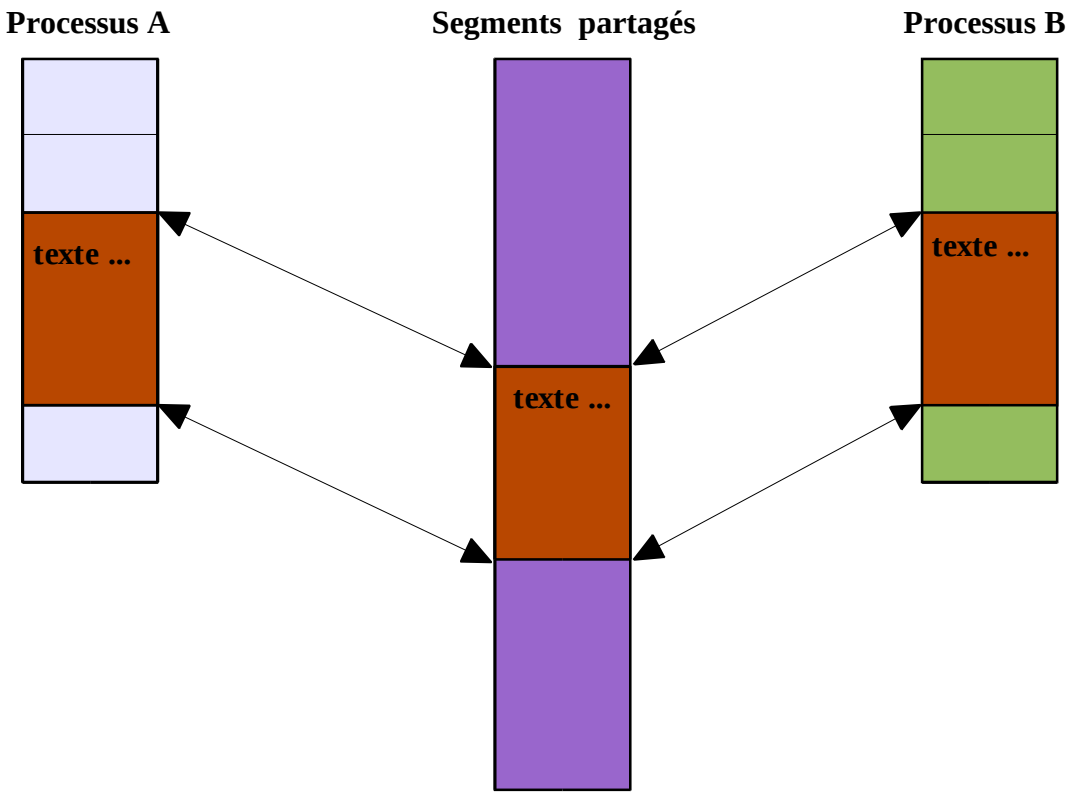
Avec le **fork** UNIX, les processus fils héritent d'une copie des données des processus pères. Il n'y a pas partage de ces données après recopie.

> Les segments de mémoire partagée permettent à deux processus distincts de partager physiquement des données.

Le partage est réalisé par l'espace d'adressage de chacun des processus, dont une zone pointe vers un segment de mémoire partagée.

Les segments de mémoire partagée existent indépendamment des processus.

L'attachement d'un segment par un processus consiste à faire correspondre à un segment donné, une adresse dans l'espace d'adressage du processus concerné.



La projection des fichiers en mémoire

Un fichier est dit **projeté en mémoire** lorsqu'il est accessible directement dans l'espace d'adressage d'un processus, comme le sont les segments de mémoire partagée.

Le fichier est alors un tableau **t** de caractères en mémoire et le **$i^{ème}$** caractère du fichier est désigné directement par l'expression **t[i − 1].**

Un fichier projeté en mémoire par plusieurs processus définit un **espace partagé** par les différents processus.

La primitive **mmap** permet d'associer une adresse de l'espace d'adressage du processus à un segment de fichier d'une certaine longueur commençant à une certaine position, le fichier étant identifié par un descripteur.

Les fonctions permettant de manipuler des fichiers directement en mémoire sont les suivantes :

```
#include <sys/mman.h>
```

/* **mmap** positionne (code retour) un pointeur sur une zone mémoire associée au contenu du fichier accessible via un descripteur de fichier ouvert. */

```
void *mmap(void *adr, taille_t lon, int prot, int drapeaux, int descripteur_de_fichier, off_t off);
```

adr : permet de demander une adresse mémoire particulière pour le segment de mémoire ; si "0", le pointeur est automatiquement attribué.

lon : longueur du segment de mémoire (quantité de données accessibles).

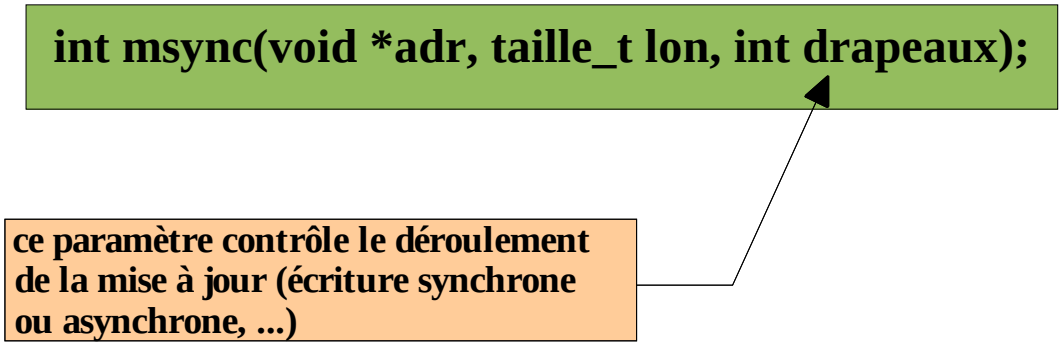
prot : définit les droits d'accès au segment (PROT_READ, PROT_WRITE, PROT_EXEC ou PROT_NONE).

drapeaux : ce paramètre contrôle l'influence des modifications apportées au segment par le programme (MAP_SHARED pour rendre effectives au fichier les modifications faites par le programme).

descripteur_de_fichier : donne accès au fichier visé.

off : permet de modifier le début des données du fichier qui sont accessibles dans le segment.

/* **msync** génère l'écriture des changements dans tout ou partie du segment de mémoire sur le fichier associé.



/* **munmap** libère le segment de mémoire.

```
int munmap(void *adr, taille_t lon);
```

Pour illustrer l'usage de ces fonctions nous allons prendre l'exemple (issu du livre "Programmation Linux" aux éditions Eyrolles/Wrox) décrit au chapitre 3 pp131−132.

/* On commence par définir une structure appelée **RECORD** et on crée **NRECORDS** exemplaires, contenant chacun son numéro de création (1 pour la première, ...). Toutes ces structures sont enregistrées dans le fichier "**records.dat**". */

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

typedef struct {
    int integer;
    char string[24];
} RECORD;

#define NRECORDS (100)

int main()
{
    RECORD record, *mapped;
    int i, f;
    FILE *fp;

    fp = fopen("records.dat","w+");
    for(i=0; i<NRECORDS; i++) {
        record.integer = i;

sprintf(record.string,"RECORD−%d",i);
        fwrite(&record,sizeof(record),1,fp);
    }
    fclose(fp);
```

/* Ensuite on modifie la valeur entière du 43ème enregistrement, à 143 au lieu de 43, et on l'écrit dans la chaîne de caractères de ce 43ème enregistrement */

```
// ouverture du flux vers le fichier "records.dat"
fp = fopen("records.dat","r+");
// déplacement de l'offset au 43ème enregistrement
fseek(fp,43*sizeof(record),SEEK_SET);
// lecture du 43ème enregistrement
fread(&record,sizeof(record),1,fp);
/* mise à jour des données de cet enregistrement dans la structure "record" */
record.integer = 143;
sprintf(record.string,"RECORD−%d",record.integer);
/* écriture de cette structure "record" maj dans le fichier */
fseek(fp,43*sizeof(record),SEEK_SET);
fwrite(&record,sizeof(record),1,fp);
fclose(fp);
```

/* Maintenant on charge dans le segment de mémoire le fichier et on accède via la mémoire au 43ème enregistrement, afin d'y changer son champ entier pour passer de 143 à 243 et mettre à jour également le champ chaîne de caractères. */

```
f = open("records.dat",O_RDWR);
// association du fichier au segment de mémoire
mapped = (RECORD *) mmap(0,
      NRECORDS*sizeof(record),
      PROT_READ|PROT_WRITE, AP_SHARED,
      f, 0);
/* mise à jour directe du 43ème enregistrement dans
le segment de mémoire */
mapped[43].integer = 243;
sprintf(mapped[43].string,"RECORD−%d",
                                  mapped[43].integer);
 // mise à jour des écritures sur le fichier (<=> fflush)
msync((void *) mapped,
      NRECORDS*sizeof(record), MS_ASYNC);

// libération du segment de mémoire
munmap ((void *) mapped,
                  NRECORDS*sizeof(record));

// fermeture du descripteur de fichier ouvert (flux)
close(f);

exit(0);
 }
```

6.3 Les files de messages

C'est une implantation UNIX du concept de boîte aux lettres, qui permet la communication indirecte entre des processus.

Les messages étant typés, chaque processus peut choisir les messages qu'il veut lire (extraire de la file).

```
struct msgbuf {
        long mtype;       /* type du message */
        char mtext[1];    /* texte du message */
        int tab[4];       /* texte du message (suite) */
        };
```

Les informations sont stockées dans les files de messages en **DATAGRAM** (un message à la fois). Les files sont de type **FIFO** (**F**irst **I**n, **F**irst **O**ut).

Un processus peut émettre des messages vers plusieurs processus, par l'intermédiare d'une même file de message (**multiplexage**).

Dans un message l'ensemble des informations doit être contigu en mémoire. Ceci interdit l'usage de pointeurs d'indirection.

msgget :	fournit l'identification d'une f.d.m. de clé donnée.
msgsnd :	utilisée par un processus pour envoyer un message sur la f.d.m. connue du processus.
msgrcv :	permet à un processus connaissant une f.d.m. donnée, d'y extraire des messages.
msgctl :	pour accéder et modifier les informations de la table des f.d.m.

L'interface détaillée de l'utilisation des Files de Messages

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>
```

Création et recherche des files de messages

int msgget (key_t cle, int option);

- **cle** : n° de clé d'une FdM (existante ou non) ou **IPC_PRIVATE** (FdM non nommée)
- **option** : **IPC_CREAT**, **IPC_EXCL** (création exclusive), **0666** (droits d'accès)

Contrôle des files de messages

int msgctl (int msqid, int cmd, struct ms_qid_dsn *buf);

- **msqid** : identificateur de la FdM visée
- **cmd** : type d'opération à effectuer sur la FdM, par exemple **IPC_RMID** pour détruire la FdM
- **buf** : pour récupérer la table associée à la FdM

Emission des messages

```
int msgsnd (int msqid, struct msgbuf *msgp, int msgtaille, int msgopt);
```

- **msqid** : identificateur de la FdM visée
- **msgp** : le message à inclure dans la FdM
- **msgtaille** : taille de l'objet place dans la FdM
- **msgopt** : option de fonctionnement du processus vis−à−vis de la FdM ; par exemple **IPC_NOWAIT** permet au processus appelant d'éviter l'attente active

Réception des messages

```
int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
```

- **msqid** : identificateur de la FdM visée
- **msgp** : zone de récupération du message à lire dans la FdM
- **msgsz** : taille maximale de la zone mémoire pointée par **msgp**
- **msgtyp** : type du message à lire ; 0 signifie que l'on prend le 1er message de la FdM quelque soit son type (le plus vieux dans FIFO)
- **msgflg** : option de fonctionnement du processus appelant (**IPC_NOWAIT** évite l'attente passive).

6.4 Les sémaphores

C'est un mécanisme de synchronisation entre processus. Un sémaphore **S** est une variable à valeurs entières positives ou nulle, manipulable par l'intermédiaire de deux opérations **P** (***proberen***) et **V** (***verhogen***) :

P (S) : si S ≤ 0, alors mettre le processus en attente ;
sinon : S := (S − 1).
V (S) : S := (S + 1) ; réveil d'un processus en attente.

P (S) : Cette opération correspond à une tentative de franchissement. S'il n'y a pas de jeton pour la section critique alors attendre, sinon prendre un jeton et entrer dans la section.

V (S) : Rendre son jeton à la sortie de la section critique. Chaque dépôt de jeton V (S) autorise le passage d'une personne. Il est possible de déposer des jetons à l'avance.

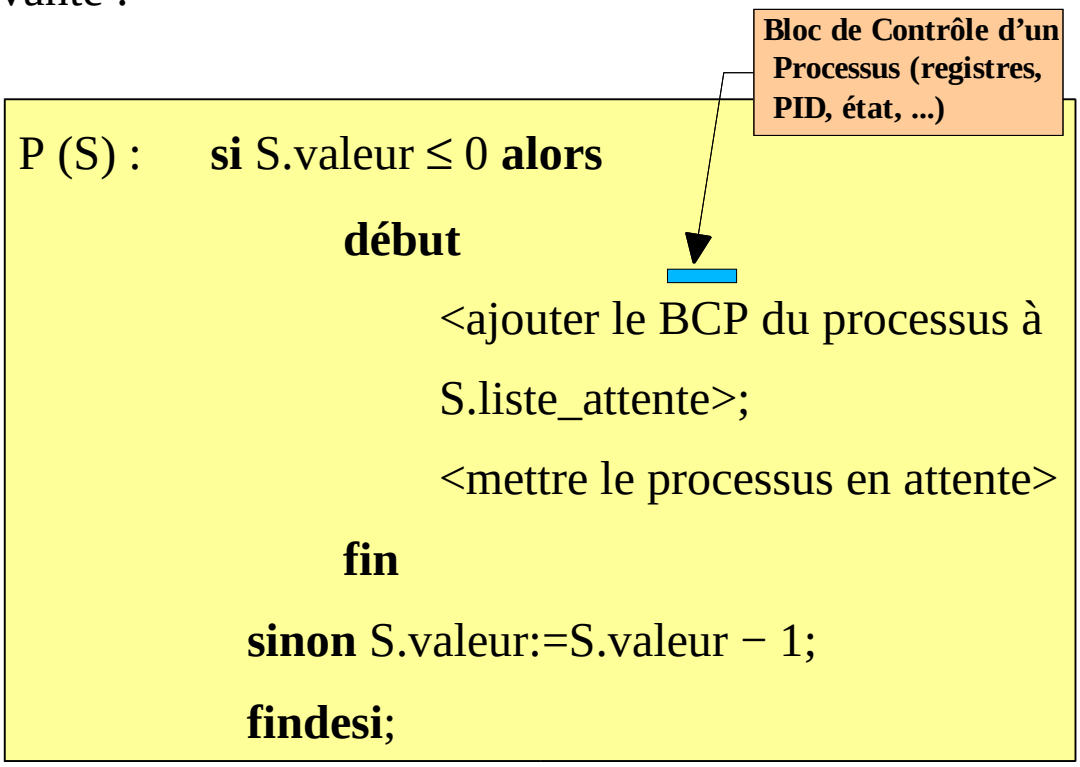
Toute implantation de ce mécanisme repose sur :

- l'**atomicité** des opérations **P** et **V**, c'est−à−dire sur le fait que la suite d'opérations les réalisant (section critique) est non interruptible (afin d'éviter les conflits entre processus), et

- l'existence d'un mécanisme de **file d'attente**, permettant de mémoriser les demandes d'opération **P** non satisfaites et de réveiller les processus en attente.

L'atomicité des opérations **V** et **P** revient à bloquer l'accès à la variable **S** (en test et en modification) dès lors qu'un processus modifie la valeur de **S** soit par **P** ou par **V**.

Une implantation détaillée des fonctions **P** et **V** est la suivante :



avec le compteur à valeurs entières **S.valeur** et la file **S.liste_attente**, initialisés respectivement à **n** et à **file vide**.

La file permet d'éviter de faire de l'attente active, c'est−à−dire que les processus soient bloqués sur une boucle du type :

tant que S ≤ 0 faire rien.

En attente passive les processus ne concourent pas pour obtenir le processeur.

```
V (S) :     si S.liste_attente non vide alors
              début
                    <choisir et enlever un BCP de
                     S.liste_attente>;
                    <faire passer à l'état prêt le
                                        processus choisi>
              fin
            sinon S.valeur:=S.valeur + 1;
            findesi;
```

Tous les problèmes de synchronisation ne sont pas solubles avec les sémaphores simples de Djiskstra, tels qu'ils ont été décrits précédemment.

Par exemple : la demande de **m** ressources différentes, dans un ensemble de **n** ressources (**m <= n**).

Soit les **n** sémaphores **S1, S2, ..., Sn** permettant de bloquer les **n** ressources. Si **P1** demande **R1, R2, R3** et **R4** et **P2** demande **R2, R3, R4** et **R5** alors on a :

P1 −> **P(S1)**, **P(S2)**, **P(S3)**, P(S4) et
P2 −> **P(S3)**, **P(S4)**, **P(S5)**, **P(S2)**

On a un risque d'**interblocage**.

Si les 4 opérations **P** étaient **atomiques**, on éviterait les interblocages (*deadlock*) :

P (S1, S2, S3, S4)
P (S2, S3, S4, S5)

Les solutions adoptées sur "SYSTEM V" (et LINUX) consistent à réaliser des ensembles de sémaphores avec la propriété qu'une opération de l'ensemble ne pourra être réalisée que si toutes les autres peuvent l'être également.

C'est le noyau qui garantit cette atomicité (sur P et V), en lançant lui−même ces opérations.

Au niveau de chaque sémaphore on définit les opérations V_n et P_n de la façon suivante :

$P_n(S)$: si **S < n**, alors attendre que **S** soit égal à **(k*n)**;

sinon **S := S − n ;**

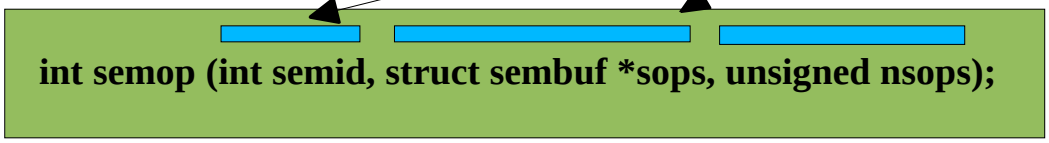
$V_n(S)$: **S := S − n ;**

réveil d'un ou de plusieurs processus en attente

Ces opérations permettent de diminuer ou d'augmenter de façon atomique la valeur d'un sémaphore de n. On pourra donc réaliser atomiquement un ensemble d'opérations P_n et V_n.

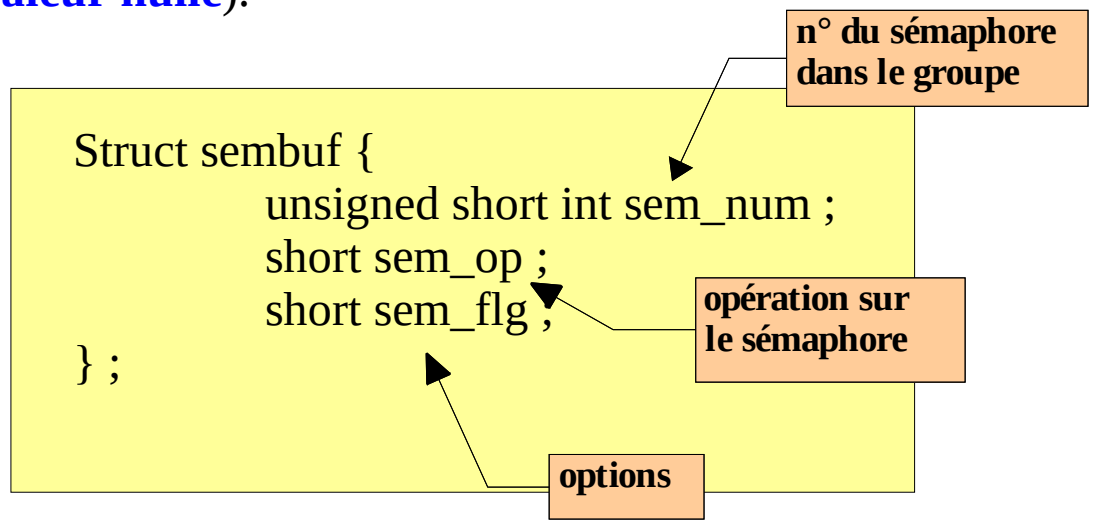
L'implantation propose également une fonction **Z** permettant d'attendre qu'un sémaphore devienne nul.

La primitive **semop()** permet de réaliser atomiquement les opérations voulues, définies à l'adresse mémoire désirée, sur l'ensemble des sémaphores identifiés.



- **semid** : descripteur du groupe de sémaphores visé
- **sops** : tableau de structures qui contient la liste des opérations
- n**sops** : nombre des opérations à réaliser lors de l'appel

La structure **sembuf** correspond à une opération sur un sémaphore (**incrémenter**, **décrémenter**, **attendre une valeur nulle**).



L’opération semop est en principe bloquante, c’est−à−dire que le processus est mis en sommeil si l’une des opérations de l’ensemble ne peut être effectuée.

Dans ce cas aucune opération de l’ensemble n’est réalisée.

Une opération élémentaire comporte un numéro de sémaphore et une opération qui peut avoir trois valeurs :

- une valeur > 0, pour une opération V_{valeur},
- une valeur = 0, pour une opération Z,
- une valeur < 0, pour une opération P_{valeur}.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
```

Création et recherche d'un groupe de sémaphores

int semget (key_t cle, int nsems, int semflg);

- **cle** : n° de clé d'un groupe de sémaphores (existant ou non) ou **IPC_PRIVATE** (groupe non nommée)
- **nsems** : nombre de sémaphores à créer dans le groupe
- **semflg** : **IPC_CREAT**, **IPC_EXCL** (création exclusive), **0666** (droits d'accès) */

Cette fonction renvoie un descripteur sur le groupe de sémaphores créé.

Contrôle des groupes de sémaphores

int semctl (int semid, int semnum, int cmd, union semun arg);

- **semid** : descripteur du groupe de sémaphores visé
- **semnum** : numéro du sémaphore choisi dans le groupe (0 pour le premier)
- **cmd** : type d'opération à effectuer sur le sémaphore, par exemple **SETVAL** pour initialiser le sémaphore à la valeur contenue dans "**arg**"
- **arg** : sert à passer des arguments aux commandes exécutées par "**cmd**"

Rendez−vous de processus : une application de l'opération Z.

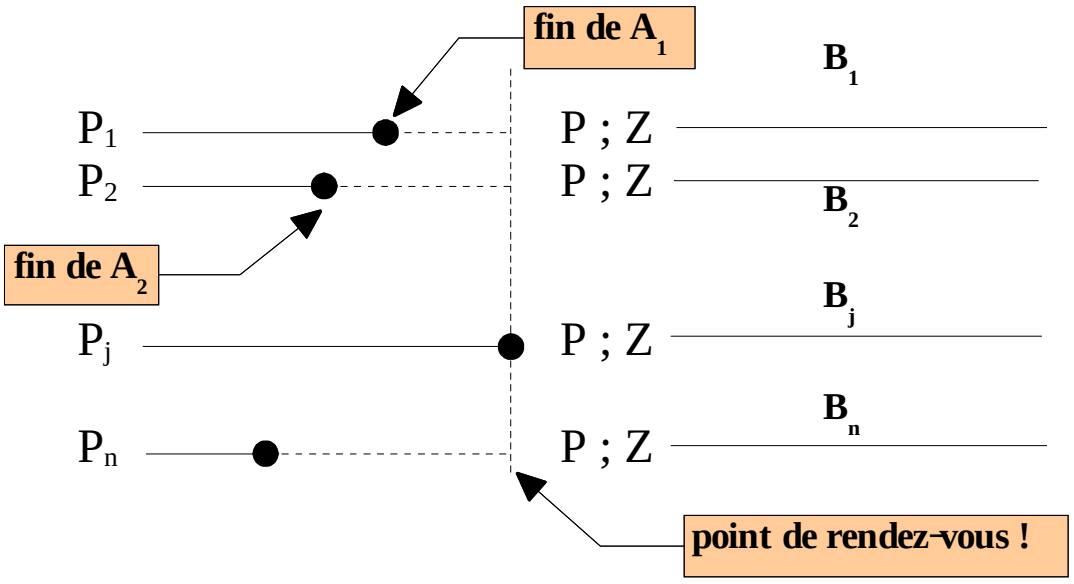
Soit une application décomposée en **N** modules qui s'exécuteront de manière concurrente et ont la particularité d'être constitués de deux parties logiques $\mathbf{A_i}$:$\mathbf{B_i}$.

L'objectif est de faire en sorte que les **N** processus ne commencent l'exécution des séquences $\mathbf{B_i}$, uniquement (point de rendez−vous) lorsque toutes les séquences $\mathbf{A_i}$ auront été complètement exécutées.

Cela signifie que les processus doivent se synnchroniser sur le processus **j** le plus lent à exécuter sa séquence $\mathbf{A_j}$; il est donc nécessaire de mettre en place entre les séquences $\mathbf{A_i}$ et $\mathbf{B_i}$ un mécanisme bloquant les processus quand ils terminent leur séquence $\mathbf{A_i}$.

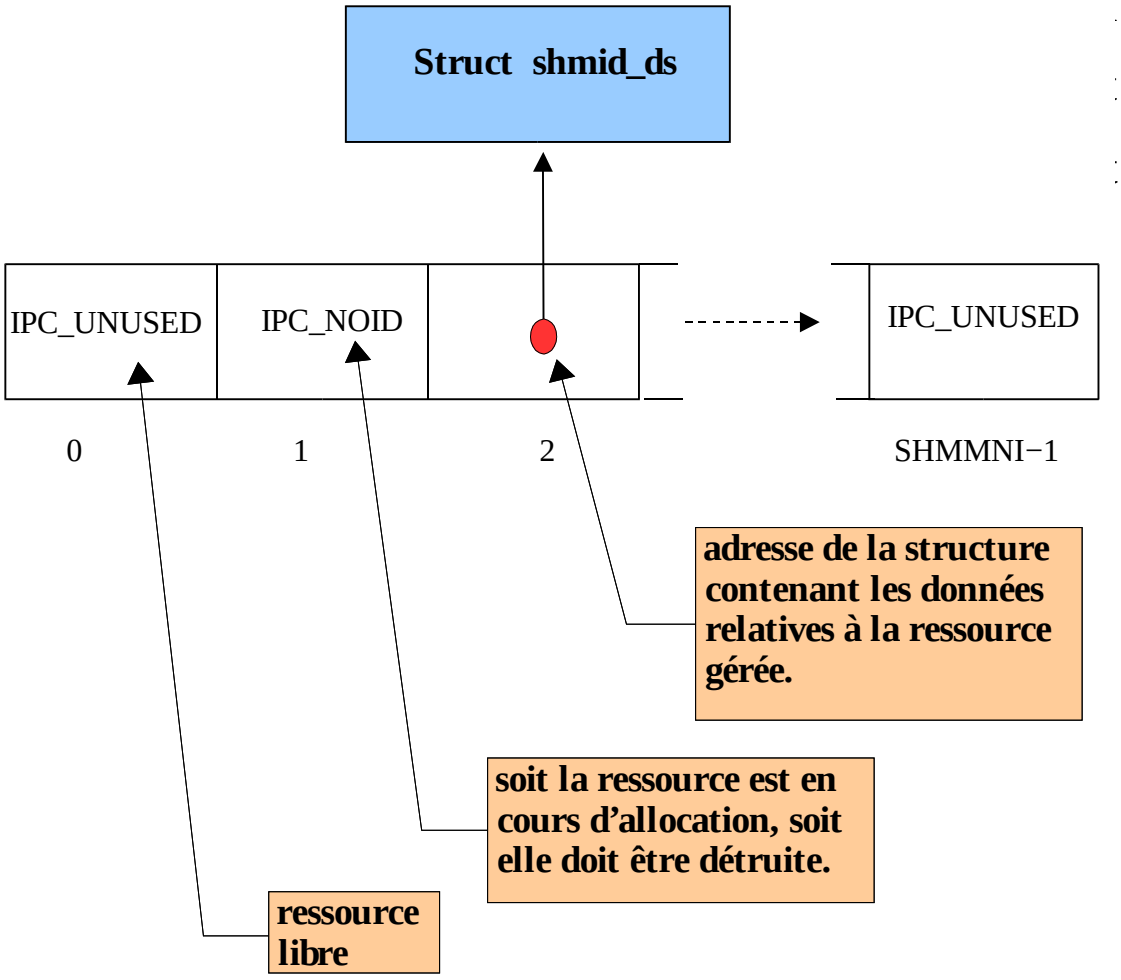
La solution consiste à initialiser un sémaphore à **N** (nombre de processus). Chaque processus exécute à la fin de la séquence $\mathbf{A_i}$, la suite d'opérations : **P ; Z**.

Cela aura pour effet de décrémenter la valeur du sémaphore et de mettre le processus en attente de la nullité (**Z**) du sémaphore.



Implantation des IPC

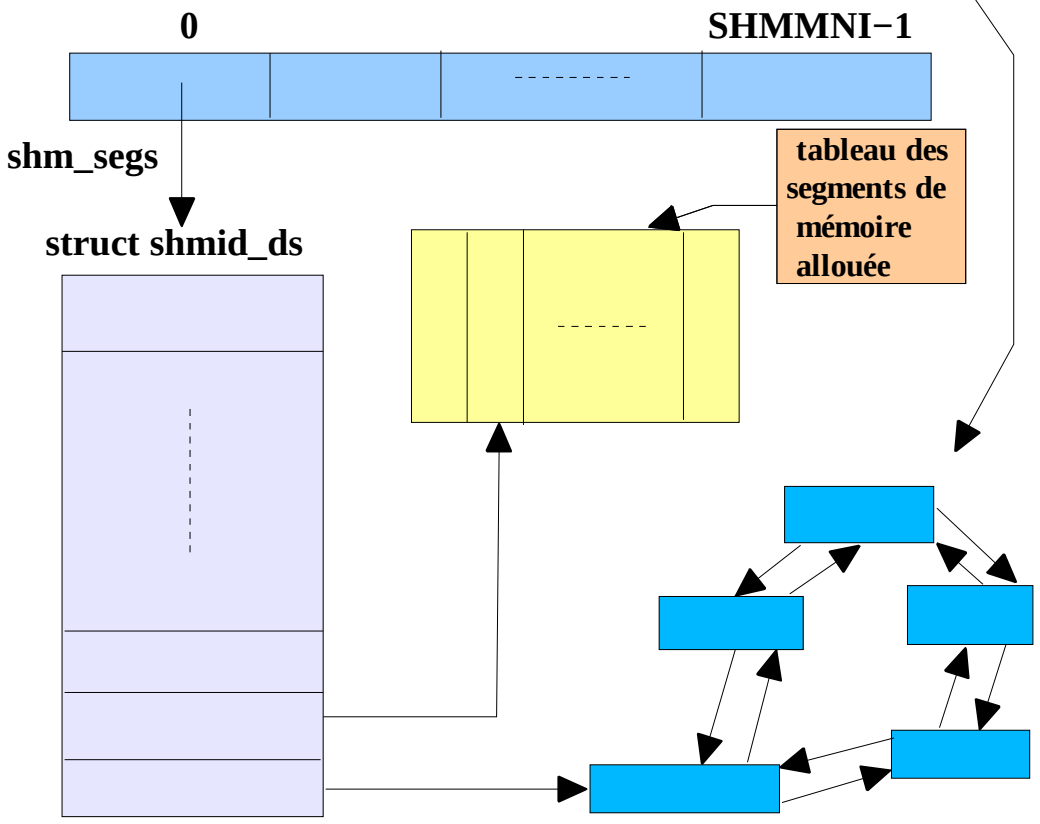
Les objets IPC sont implémentés sous la forme de trois tableaux (un par type d'IPC) de taille constante respectivement ***MSGMNI*** (file de messages), ***SEMMNI*** (sémaphores) et ***SHMMNI*** (segments de mémoire partagée). Ci− après le tableau des segments de mémoire partagée :



Représentation interne des "segments de mémoire partagée"

Les éléments fondamentaux de la structure de la mémoire partagée sont :

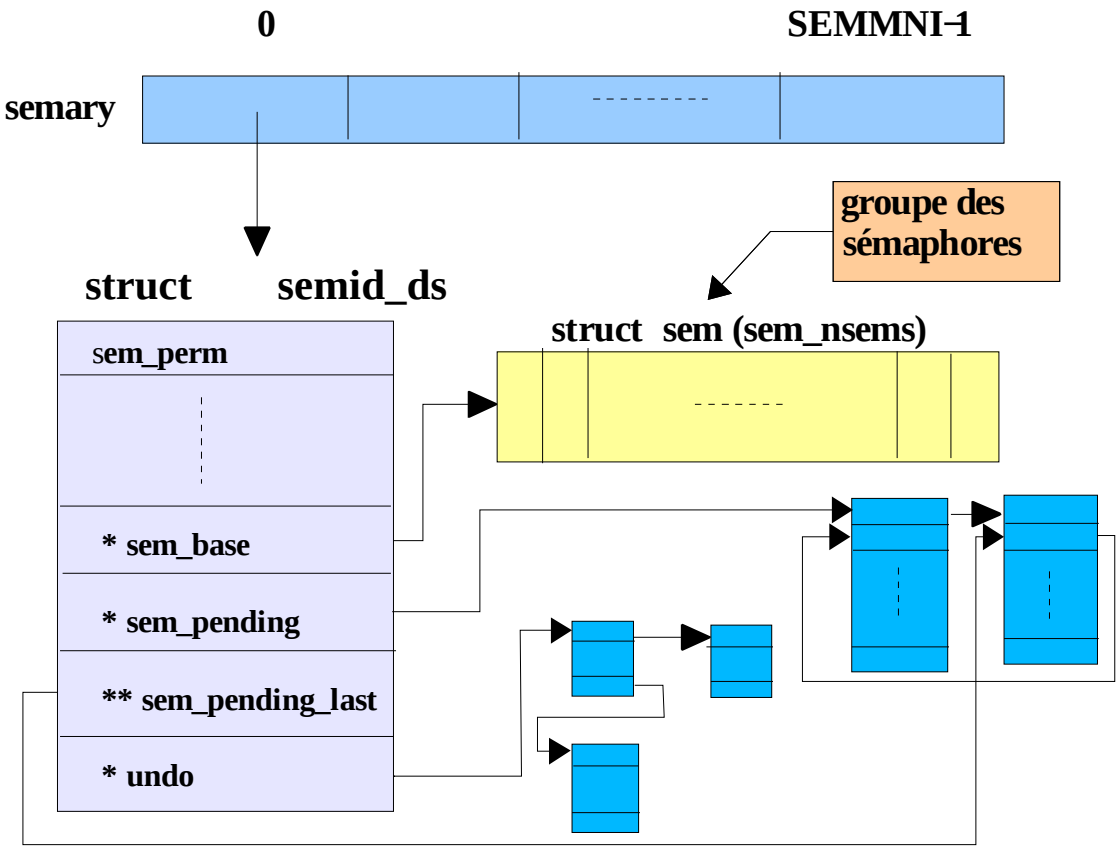
- la taille du segment en octets, et en nombre de pages ;
- un tableau de pointeurs sur les différentes *fenêtres* (pages mémoire) ;
- une liste circulaire des attachements (à des processus) de la mémoire partagée.



Représentation interne des "sémaphores"

Les sémaphores sont stockés par le système dans le tableau **semary**. Ce tableau contient des pointeurs sur la structure **semid_ds**. Sa taille indique donc le nombre maximum de groupes de sémaphores que l'on peut créer (**SEMMNI**). Chaque structure contient trois listes :

- la liste des sémaphores du groupe (**base**) stockée en mémoire sous la forme d'un tableau ;
- la liste des opérations en attente (**sem_pending**) gérée sous la forme d'une liste doublement chaînée ;
- la liste des requêtes annulables (**undo**).



Représentation interne des "files de message"

La file de message d'indice n est constituée d'éléments tels que les droits d'accès. Mais les deux éléments importants sont des pointeurs sur la file :

- un pointeur sur le premier élément ;
- un pointeur sur le dernier élément de la file.