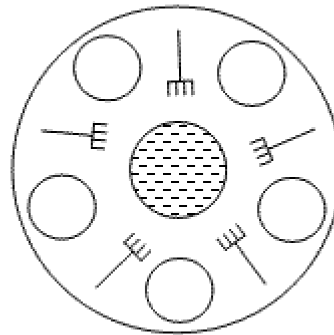
Systèmes d'Exploitation

La communication inter-processus

Version 1



Paternité - Pas d'Utilisation Commerciale :
http://creativecommons.org/licenses/by-nc/2.0/fr/

Table des matières

Objectifs 5

I - La Communication Inter-Processus 7

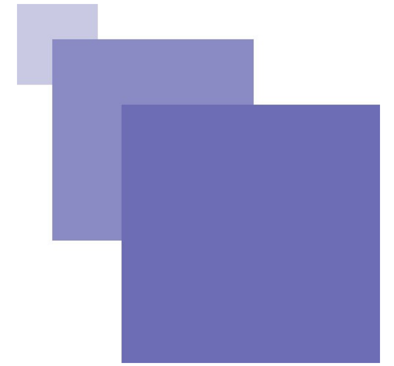
A. Les pipes.......7

1. Les pipes anonymes.......7
2. Les pipes nommés.......9

B. Les IPC System V.......10

1. Gestion des IPC System V.......10
2. Les segments de mémoire partagés.......13
3. Exclusion mutuelle et section critique.......18
4. Les sémaphores.......23
5. Problèmes classiques de synchronisation [beauquier].......30

Objectifs



- Réaliser une communication inter-processus à travers différents outils systèmes
- Analyser une situation d'exclusion mutuelle
- Assurer la communication inter-processus à travers des sémaphores

La Communication Inter-Processus

I

Les pipes	7
Les IPC System V	10

Il existe plusieurs mécanismes de communication entre processus.

- Les pipes anonymes : pipe()
- Les pipes nommés : mkfifo()
- Les IPC System V :
  - Les segments de mémoire partagée : shmem
  - Les sémaphores : semop
  - Files de messages : msgget (non abordé dans ce cours)
- Les sockets : non abordés dans ce cours

A. Les pipes

Il existe deux types de pipes :

- Les pipes anonymes permettant la communication entre deux processus ayant un ancêtre commun.
- Les pipes nommés permettant la communication entre n'importe quels processus en passant par le système de fichier.

1. Les pipes anonymes

Définition : Pipe anonyme

Un pipe anonyme est un tube qui permet la communication entre deux processus ayant un ancêtre commun. Ce qui est écrit dans une extrémité du pipe peut être lu de l'autre extrémité.

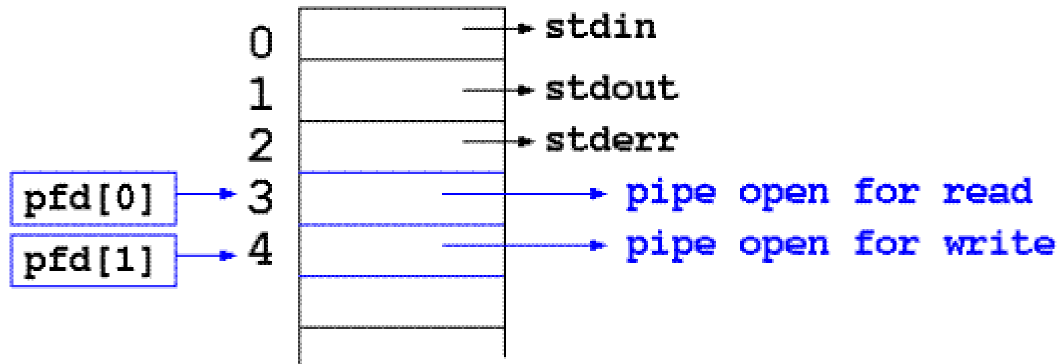
Méthode : Procédure de fonctionnement

- Le pipe est créé par l'appel système : pipe()
- Le processus créateur crée un files par fork()
- Le père et le fils choisissent le sens de communication dans le pipe commun
- Ils communiquent par read() et write() sur le pipe commun, mêmes appels systèmes que sur les fichiers

Création d'un pipe

Un pipe est défini par deux descripteurs :

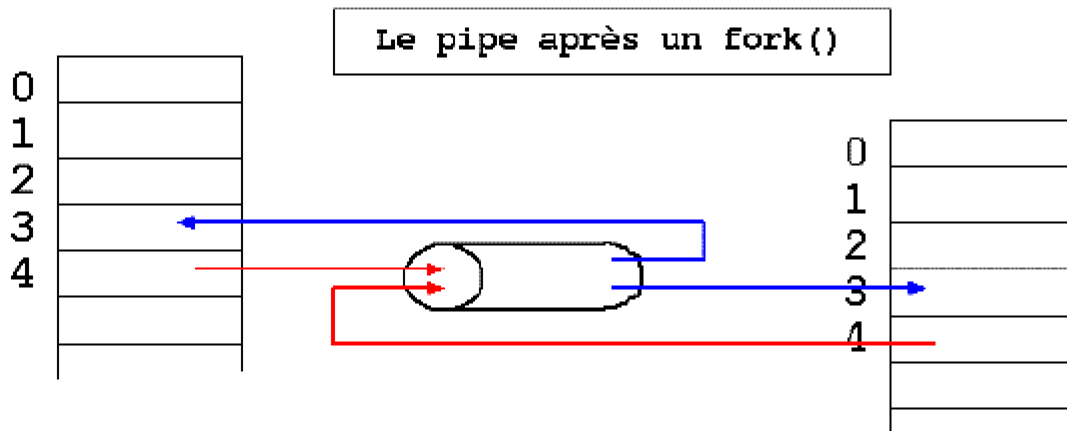
```
int pfd[2]; /* déclaration des descripteurs */
pipe(pfd) ; /* création du pipe ==> deux descripteurs sont
alloués dans la table de descripteurs */
```



Création d'un pipe

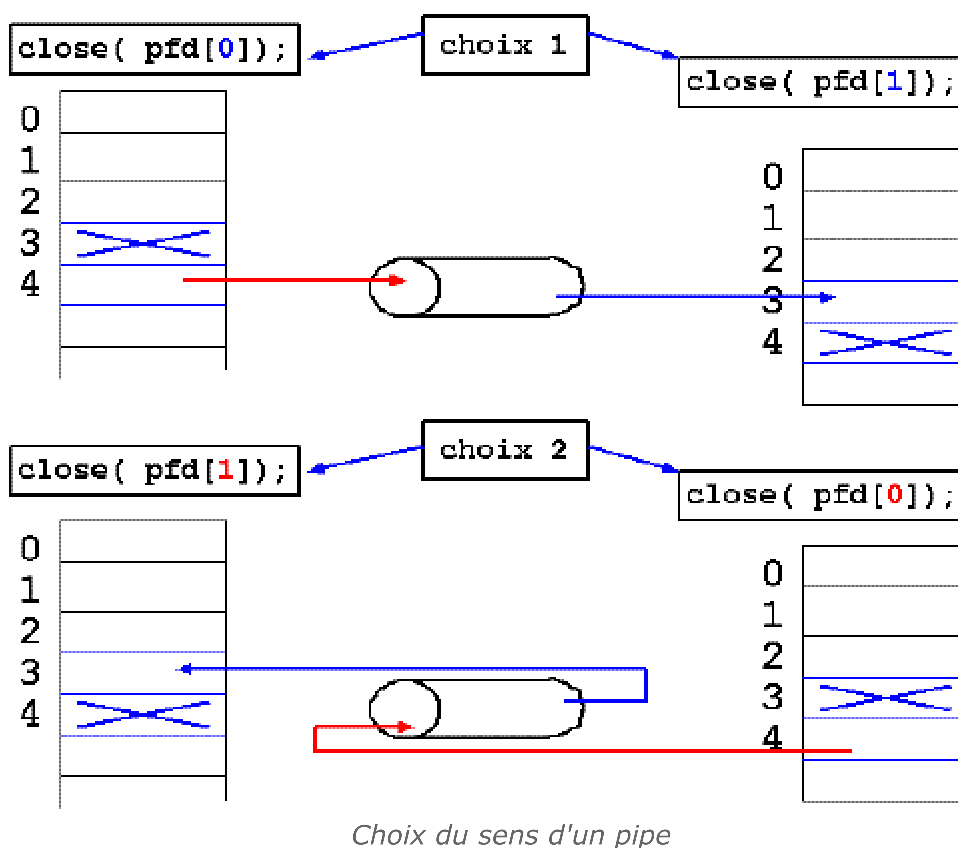
Héritage du pipe

Après le fork() le pipe est dupliqué dans le processus fils puisque la table des descripteurs fait partie du PCB du processus père.



Le pipe après un fork()

Il y'aura deux accès dans chaque sens (un accès pour le père et un autre pour le fils). Il faudra que chaque processus choisisse le sens d'utilisation.



Choix du sens d'un pipe

Remarque : Risque de blocage

La coordination entre le processus lecteur et le processus rédacteur doit être prévue, sinon il y'a risque de blocage de l'un des processus :

- buffer plein ==> write(p[1], ...) bloque
- buffer vide ==> read(p[0], ...) bloque

Par contre,

- p[1] closed et buffer vide ==> read(p[0], ...) returns 0 (end of file)

Les lectures / écritures peuvent être rendues non-bloquantes en utilisant l'option O_NODELAY (System V), ou O_NONBLOCK (POSIX)

```
status = fcntl(pfd[0], G_GETFL) ;
fcntl(pfd[0], F_SETFL, status | O_NONBLOCK) ;
```

2. Les pipes nommés

Définition : Pipe nommé

Un pipe nommé est un fichier spécifique de type FIFO. Il est similaire à un pipe anonyme sauf qu'il fait partie du système de fichier. Il peut être ainsi ouvert par plusieurs processus en lecture ou en écriture.

Lorsque des processus s'échangent des données à travers le pipe nommé, le noyau fait passer les données en interne sans écriture sur le système de fichier.

Création d'un pipe nommé

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char* pathname, mode_t mode) ;
```

B. Les IPC System V

En plus des mécanisme de communication entre processus : les signaux, les pipes anonymes et les pipes nommés, la branche System V d'UNIX propose un ensemble de mécanismes connus sous le nom de IPC System V (segments de mémoire partagés, sémaphores et files de messages). Dans ce qui suit nous étudierons les mécanismes de gestion communs à ces différents IPC puis nous étudierons en particulier les segments de mémoire partagés et les sémaphores.

1. Gestion des IPC System V

a) Descripteur d'un IPC

L'appel get

Pour chaque type d'IPC le système gère une table globale.

Chaque mécanisme d'IPC dispose d'un appel :

id=xxxget(Clé, ...)

avec xxx=shm, sem, msg

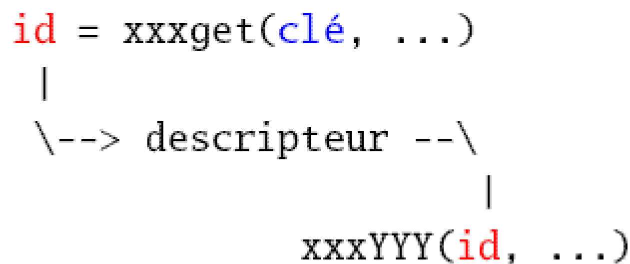
L'appel get permet de créer une nouvelle entrée ou retrouver une entrée existante.

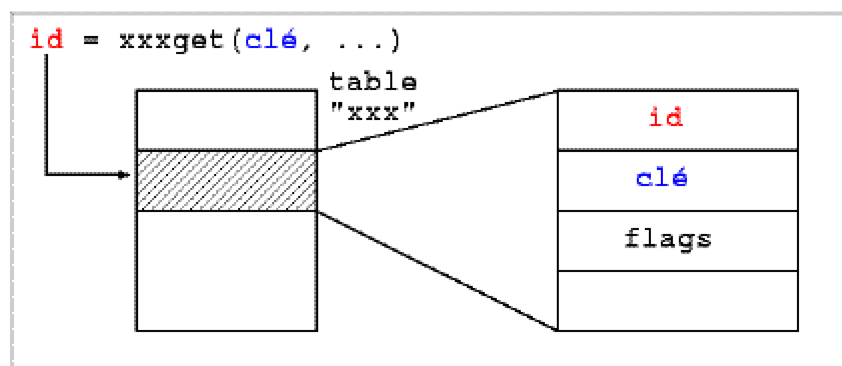
Cette entrée est allouée par le système dans la table de l'IPC correspondant.

Donc toute utilisation d'un IPC commence par un appel xxxget().

Descripteur d'un IPC

Les appels get renvoient un descripteur destiné à être utilisé lors des appels aux autres fonctions qui permettent de manipuler l'IPC (par exemple : shmat, shmctl, shmdt, etc.)





Descripteur d'un IPC

Une entrée d'un IPC désignée par un descripteur

Chaque entrée possède des champs d'autorisation qui contiennent :

- l'UID et le GID du processus qui a créé cette entrée
- un UID et un GID remplis par l'appel à la primitive control
- un ensemble de bits rwx pour user-group-world semblables à ceux des fichiers

Chaque entrée contient des champs qui mémorisent des informations d'états telles que :

- PID du dernier processus ayant modifié cette entrée
- date du dernier accès ou de la dernière mise à jour

Chaque mécanisme définit un appel xxxctl qui permet :

- d'obtenir des infos sur l'état de l'entrée
- de modifier ces infos d'état
- de retirer l'entrée de la table

Pour toutes ces opérations le système vérifie que le processus appelant possède les droits correspondants (bits rwx et le UID et GID du processus)

Méthode : Utilisation d'un IPC

L'utilisation d'un IPC par un programme A obéit à la séquence :

- allouer une entrée dans la table id=xxxget() ;
- utiliser cette entrée xxxYYY(id, ...) ;
- libérer cette entrée

Remarque : Réallocation des descripteurs

Afin d'éviter qu'un processus ne réutilise un descripteur obsolète après l'avoir fermé et accède ainsi à un descripteur qui ne lui appartient pas, le système utilise la méthode suivante pour allouer les descripteurs :

Supposons que la table du mécanisme contiennent NE=100 entrée :

Lorsque le descripteur 1 est fermé après usage par un processus le système incrémente le descripteur associé à cette entrée de la longueur de la table, ainsi les descripteurs successifs de l'entrée 1 seront : 101, 201, 301, etc.

Si un processus obtient le descripteur 101, après sa fermeture, le prochain processus obtiendra la valeur 201. Si le premier processus tente d'accéder à son descripteur 101, le système consulte l'entrée 1 et trouve qu'elle est passée à 201 et renvoie donc une erreur.

b) Espaces de noms d'IPC

Pour créer ou accéder un IPC il faut fournir une clé : xxxget(Clé, ...)

C'est cette clé qui permet à différents processus coopérants d'accéder à la même instance de l'IPC.

L'ensemble des valeurs possibles de ce nom est appelé "espace de noms".

La "Clé" est de type key_t ; un entier de 32 bits.

Au lieu de mémoriser cette valeur entière, le système fournit une fonction "ftok" qui permet de convertir le nom d'un fichier en une valeur de type key_t.

Ainsi les processus désirant utiliser la même instance d'un IPC doivent mémoriser ce nom de fichier et dériver la clé associée en utilisant la fonction "ftok".

Syntaxe : La fonction ftok

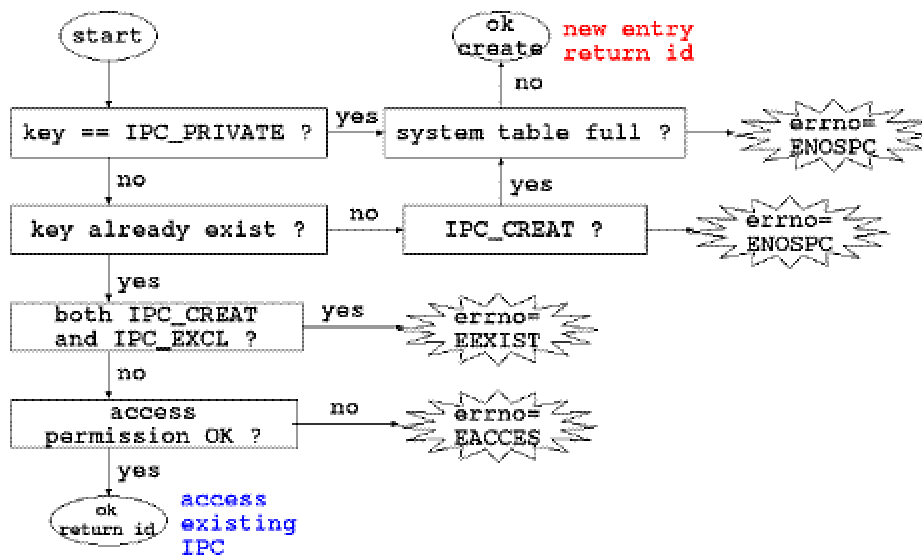
```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char* pathname, char id) ;
```

Le pathname doit exister. S'il n'existe pas ou pas accessible au processus appelant, ftok retourne -1.

Si des processus ont besoin de plusieurs canaux de communication, ils appellent ftok plusieurs fois avec le même pathname mais des valeurs différentes de "id" (le deuxième argument)

c) Logique de création d'un IPC

La création ou accès à un IPC se fait avec l'appel xxxget(...) dont le fonctionnement dépend de deux paramètres : la clé, et un flag :



Logique de création d'un IPC

Suivant les paramètres de l'appel xxxget() la réponse du système peut être :

	argument flag	clé n'existe pas	clé existe déja
1	aucun	error, errno= ENOENT	OK
2	IPC_CREAT	OK, crèe nvelle entrée	OK
3	IPC_CREAT|IPC_EXCL	OK, crèe entrée	errno=EEXIST

Réponse du système à un appel xxxget(...)

- A la ligne 2, on ne sait pas si on a créé une nouvelle entrée ou si on a utilisé une entrée existante, alors qu'à la ligne 3 on demande la création seulement si l'entrée n'existe pas déjà.
- L'appel à xxxget(Clé, ...) avec Clé=IPC_PRIVATE permet d'obtenir un IPC sans clé. Il ne pourra être partagé que par les processus fils du créateur de l'IPC.
- Quand un nouveau canal est créé les 9 bits de poids faible du flag initialise le champ "mode protection" de l'entrée créée. De plus les champs suivants sont initialisés :
  - uid=cuid= UID du processus créateur

- gid=cgid= GID du processus créateur

cuid et cgid ne changent plus jusqu'à la destruction de l'IPC alors que uid et gid peuvent être modifié avec la fonction xxxctl().

d) Commandes sur IPC

La commande ipcs = status

Cette commande retourne des informations sur l'état des IPC dans le système

Exemple

```
> ipcs
------ Shared Memory Segments --------
key        shmid   owner    perms   bytes    nattch  status
0x00000000 65536   vayssade 600     393216   2       dest
0x00000000 98305   vayssade 600     393216   2       dest

------ Semaphore Arrays --------
key        semid    owner    perms   nsems
------ Message Queues --------
key        msqid    owner    perms   used-bytes  messages
```

ipcs

La commande ipcrm

Cette commande permet de détruire un IPC

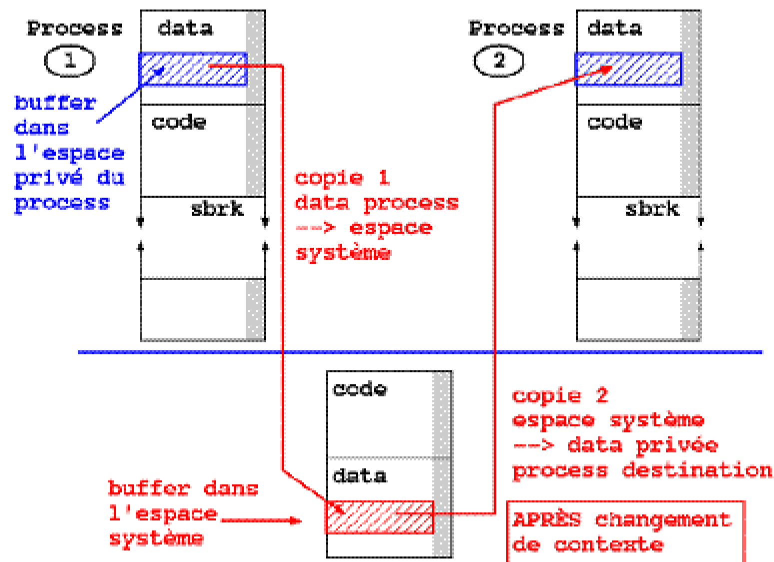
Exemple

ipcrm -m key

Détruit la file de message de clé "key"

2. Les segments de mémoire partagés

Les IPC de type pipe, FIFO, message transmettent les données du contexte du processus émetteur vers le contexte du processus destinataire en les faisant transiter par l'espace système. Ceci entraîne deux copies des données :

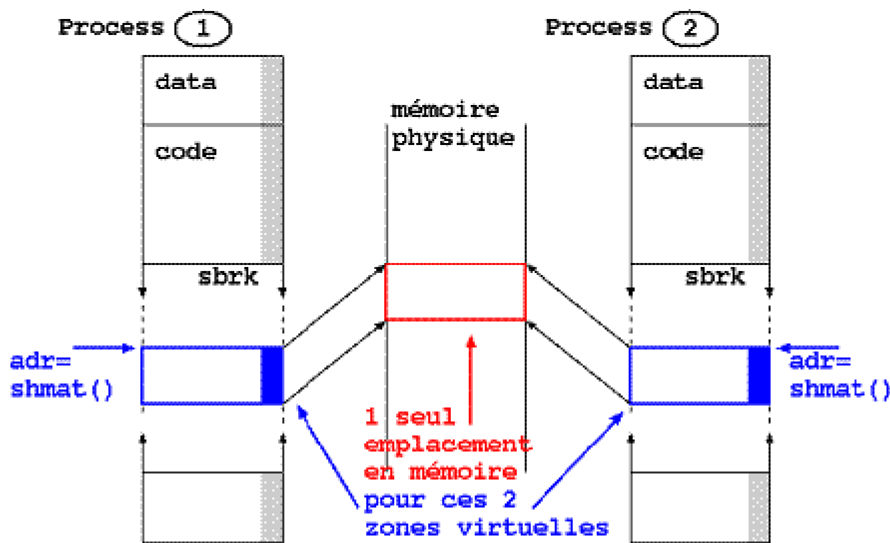


Problème de duplication des données

a) Partage d'un segment de mémoire

Le partage d'un segment de mémoire entre deux (ou plusieurs) processus permet des échanges de données sans copie, puisque les deux processus peuvent lire les mêmes mots de mémoire physiques.

Cet espace physique sera associé à des espaces virtuels dans les contextes des processus le partageant.



Un segment de mémoire physique partagé

Dans ce cas, c'est aux processus de se coordonner dans leurs accès à cette mémoire commune.

b) Création d'un segment de mémoire partagé

Méthode : Création d'un segment de mémoire partagée

La création d'un segment de mémoire partagé se fait avec la primitive shmget(...)

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

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

Exemple

```
#define CLE-SH 33 /* notre choix */
#define PROT 0666 /* Protection */
main(){
  shmid=shmget(CLE-SH, taille, PROT|IPC_CREAT) ;
  if(shmid==-1) perror("Création échoue") ;
```

A ce stade le segment de mémoire est créé, une entrée est allouée pour lui dans la table système globale des segments de mémoire partagée, l'identificateur permet d'accéder à cette entrée.

c) Utilisation d'un segment de mémoire partagée

Méthode : Attacher un segment à l'espace virtuel d'un processus

Pour utiliser un segment de mémoire partagée, il faut d'abord l'attacher à l'espace virtuel du processus avec la primitive shmat(...) :

```
char* shmat(int shmid, char* shmaddr, int shmflag) ;
```

- si shmaddr=0, alors le système choisit l'emplacement dans l'espace virtuel
- si shmaddr#0,
  - si shmflag=0, le segment est attaché à l'adresse shmaddr
  - si shmflag=SHM_RND (round) le segment est attaché à l'adresse spécifiée dans shmaddr, arrondie vers le bas par la constante (SHMLBA)
- si shmflag contient SHM_RDONLY le segment est attaché en lecture seule, sinon en lecture + écriture

Exemple

```
char* adr ;
adr=shmat(shmid,0,0) ;
/* puis accès par */
*adr=
/* ou */
strcpy(adr, p1) ;
```

Complément

Les protections (données dans le paramètre flag de l'appel shmget) sont définies de la façon suivante :

valeur numérique	constante symbolique	description
0400	SHM_R	Read par propriétaire
0200	SHM_W	Write par propriétaire
0040	SHM_R >> 3	Read par groupe
0020	SHM_W >> 3	Write par groupe
0004	SHM_R >> 6	Read par world
0002	SHM_W >> 6	Write par world
	IPC_CREAT	créer
	IPC_EXCL	créer seulmt si existe pas

Options de protection d'un segment partagé

Méthode : Détachement et destruction d'un segment de mémoire partagée

Après utilisation d'un segment de mémoire partagée il faut faire un détachement avec la primitive shmdt(...) :

```
int shmdt(char* adr) ;
```

où *adr est le pointeur renvoyé par la fonction shmat(...)

Ceci ne détruit pas le segment, pour cela il faut appeler la fonction shmctl avec un paramètre cmde égal à IPC_RMID (remove ID)

```
int shmctl(int shmid, int cmde, struct shmid_ds * buf) ;
```

Par exemple, on fait :

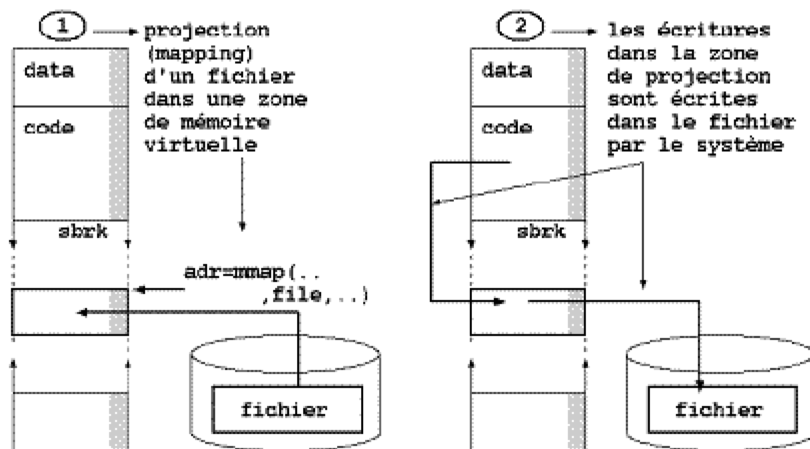
```
shmctl(shmid, IPC_RMID, 0) ;
```

d) Projection de fichiers en mémoire

Projection d'un fichier en mémoire

La norme POSIX a introduit la possibilité de projeter (mapper) un fichier dans une zone de l'espace virtuel avec la primitive mmap(...).

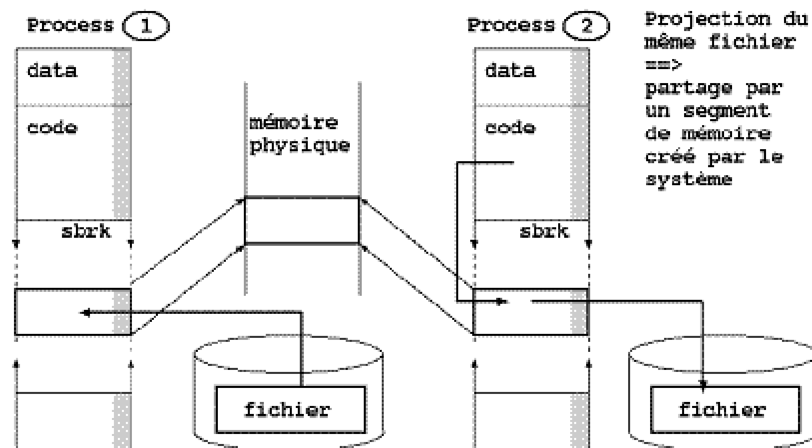
Le contenu du fichier est alors vus comme un simple tableau et la lecture et l'écriture dans le fichier (directives read() et write()) sont remplacées par de simples accès dans un tableau (tab[i]).



Mapper un fichier dans un espace virtuel

Partage d'un fichier en mémoire

Lorsque plusieurs processus projettent un même fichier en mémoire, celui-ci est partagé par l'intermédiaire d'un segment de mémoire partagée, créé automatiquement par le système.



Partage d'un fichier projeté en mémoire

Complément

C'est de cette façon que Solaris partage le code d'un programme ou celui d'une bibliothèque partageable (.so) quand ils sont utilisés par plusieurs processus simultanément.

Exemple : Utilisation de la primitive mmap

```
/* map1.c - UTC UV SR02 - (c) Michel.Vayssade@utc.fr
   compil: gcc -o map1 map1.c  usage:  ./map1 fichier
   démo utilisation mmap()                                 */
#include <stdio.h>  /* pour NULL */
#include <fcntl.h>  /* pour O_RDWR */
#include <sys/mman.h>

int main (int argc, char **argv) {
  int fd; char c, *adr;
  if((fd=open(argv[1],O_RDWR))==-1)
     {perror(argv[1]);exit(0);}
  adr = (char *)mmap(NULL,1024,
                  PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
  if (adr==MAP_FAILED) { perror("mmap"); exit(0);}

  printf("adr=%p car.3,4,5=%c%C%c\n",
              adr,adr[3],adr[4],adr[5]);
  adr[4]++;
  printf(" car.3,4,5=%c%C%c\n",adr[3],adr[4],adr[5]);
  printf("pid=%d\n",getpid());  sleep(10);
}
```

Exemple d'utilisation de mmap

```
$ gcc -o map1 map1.c
$ cat map1.txt
abcdefghijklmnopqrstuvwxyz
$ ./map1 map1.txt
adr= 0x40000000  car.3,4,5=def
 car.3,4,5=dff                    adr[4]++;
pid=10582
$ cat map1.txt
abcdffghijklmnopqrstuvwxyz
$
```

Résultat de l'exécution

Remarque : MAP_SHARED vs. MAP_PRIVATE

Il est possible avec la primitive mmap de projeter en mémoire plusieurs fois le même fichier, à des adresses virtuelles différentes.

Des paramètres de la fonction mmap permettent de choisir le mode de projection, en particulier MAP_SHARED ou MAP_PRIVATE.

Les écritures dans une projection SHARED sont répercutées dans le fichier d'origine, et éventuellement dans les mémoires des autres processus ayant projeté le même fichier en mode SHARED, alors que les écritures dans une projection PRIVATE n'affectent pas le fichier d'origine (le comportement dans l'autre sens n'étant pas spécifié).

3. Exclusion mutuelle et section critique

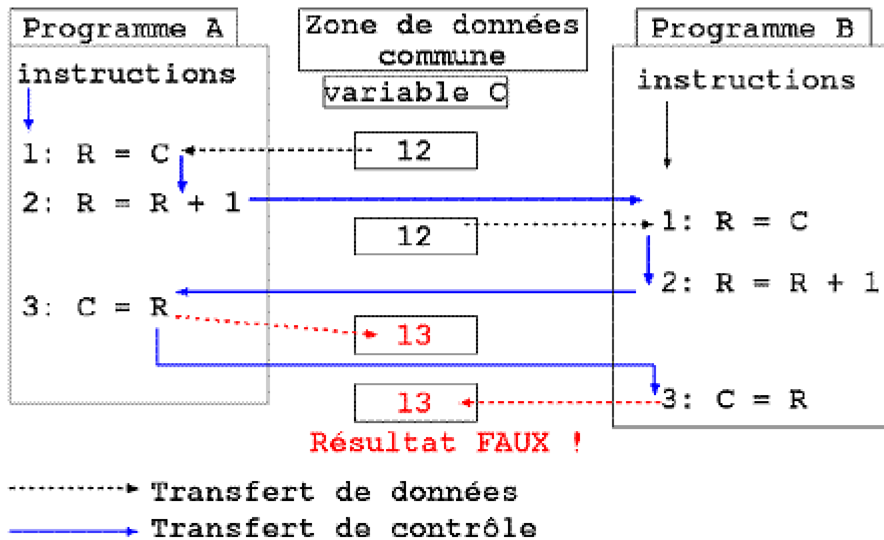
Nous allons étudier maintenant un des problèmes clés des systèmes d'exploitation : celui du partage de données par plusieurs processus ayant des exécutions indépendantes.

Sans certaines précautions, l'exécution simultanée de processus partageant des données peut conduire des programmes corrects à produire des résultats faux.

a) Section critique

Notion de section critique

Considérons l'exemple suivant où deux programmes A et B partage une variable C.



Partage de donnée et section critique

Ce schéma montre que l'exécution simultanée des deux programmes corrects peut conduire à une erreur lorsque l'entrelacement des instructions des deux programmes provoque l'erreur de mise à jour de la variable commune.

On dit que la séquence d'instructions qui modifient cette variable commune constitue une section critique.

Il va falloir s'assurer que lorsque l'un des deux programmes a déjà commencé l'exécution de sa section critique, l'autre se trouvera bloqué à l'entrée de la sienne, et ne sera autorisé à poursuivre son exécution que lorsque le premier aura quitté la section critique.

b) Exclusion mutuelle

Notion d'exclusion mutuelle

Dans l'exemple de la variable C, l'erreur vient du fait que le programme B utilise C avant que le processus A ait fini de s'en servir.

Il faut alors empêcher la présence simultanée des deux processus dans la section critique en introduisant dans le code ce qu'on appelle un mécanisme d'exclusion mutuelle :

```
obtenir_acces_exclusif ;
  manipuler les données communes ;
relacher_acces_exclusif ;
```

Conditions d'exclusion mutuelle

Il a été démontré que pour être correct dans tous les cas, un mécanisme d'exclusion mutuelle doit respecter 5 conditions :

1. exclusion mutuelle de la section critique
2. pas d'hypothèse sur les vitesses relatives des processus
3. progression : un processus suspendu en dehors de la section critique ne doit pas bloquer les autres
4. attente bornée et absence de famine : un processus qui demande l'entrée en section critique l'obtient en un temps fini,
5. pas d'interblocage dans le mécanisme de contrôle

c) Exclusion mutuelle : Algorithme 1

Test sur un drapeau

La première tentative de résolution de l'exclusion mutuelle est de mettre en début de la section critique un code (cf. 'Test sur un drapeau' p 19) qui permet de garantir l'exclusion mutuelle.

```
        initialiser drapeau à 0
test:   si (drapeau == 1) aller_à test   ! si occupée, réessayer
          drapeau = 1                    ! verouiller section
          exécuter la section critique
          drapeau = 0                    ! libérer section
```

Test sur un drapeau

Cet algorithme ne garanti pas l'exclusion mutuelle. Un programme peut être suspendu entre le test et le changement d'état du drapeau.

```
        Prog.A                          Prog.B
          |                              ...
si (drapeau == 1)                        ...
          ......suspendu ----------> reprise
          ...                              |
          ...                           si (drapeau == 1)
          reprise <---------------suspendu ...
          |                              .....
        drapeau = 1                      ....
        ......suspendu ----------> reprise
        .....                           drapeau = 1
        .....                              |
      *** tous les deux sont en section critique ***
```

L'algorithme 1 ne garanti pas l'exclusion mutuelle

d) Exclusion mutuelle : Algorithme 2

Test sur deux drapeaux

Dans cet algorithme (cf. 'Tentative d'exclusion mutuelle par test sur 2 drapeaux' p 20) on tente de contrôler deux programmes P0 et P1 avec deux drapeaux D[i].

P0 entre en section critique si D[1]=0 et inversement. Lorsque Pi entre en section critique il met D[i] à 1. Il le remet à 0 en sortant de la section critique.

```
          initialiser D[0]=D[1]=0
          // drapeau D[i]=1 si Pi est en section critique
Entree(i):| Tant_que D[j] Faire: rien;   (ins 1)
          | D[i] = 1;                    (ins 2)
          | ret

Sortie(i):| D[i] = 0;                    (ins 3)
          | ret
```

Tentative d'exclusion mutuelle par test sur 2 drapeaux

Cet algorithme ne garanti pas l'exclusion mutuelle :

```
        Prog.A                          Prog.B
          |                              ...
(ins 1) D[1]==1 ??
          | non
          ......suspendu ----------> reprise
          ...                              |
          ...                           (ins 1) D[0]==1 ??
          ...                              | non
          reprise <---------------suspendu ...
(ins 2) D[0]= 1                     ....
          ...                 (ins 2) D[1]= 1

      *** tous les deux sont en section critique ***
```

Algorithme 2 ne garanti pas l'exclusion mutuelle

e) Exclusion mutuelle : Algorithme 2 bis

Test sur deux drapeaux en inversant le test

Dans cette variante (cf. 'Tentatvie d'exclusion mutuelle par test sur 2 drapeaux (bis)' p 21) de l'algorithme précédent, on change d'abord l'état du drapeau puis on teste :

```
              initialiser D[0]=D[1]=0
              // drapeau D[i]=1 si Pi est en section critique
     Entree(i):| D[i] = 1;                  (ins 1)
               | Tant_que D[j] Faire: rien;  (ins 2)
               | ret

     Sortie(i):| D[i] = 0;                  (ins 3)
               | ret
```

Tentatvie d'exclusion mutuelle par test sur 2 drapeaux (bis)

Cet algorithme ne garanti pas l'exclusion mutuelle :

```
       Prog.A                        Prog.B
         |                            ...
  (ins 1) D[0]=1
         | non
        ......suspendu ----------> reprise
        ...                            |
        ...                       (ins 1) D[1]=1
        ...                            | non
        reprise <---------------suspendu ...
  (ins 2) D[1]==1 ??
         | oui
         attendre                     ....
        ...                  (ins 2) D[0]==1 ??
                                        | oui : attendre
      *** tous les deux sont bloqués ***
```

L'algorithme 2 bis ne garanti pas l'exclusion mutuelle

f) Exclusion mutuelle : Algorithme de Peterson (1981)

Attention

Dans les tentatives logicielles précédentes, le problème n'est pas résolu car le contrôle repose chaque fois sur la modification d'une variable et un test. Les deux opérations étant réalisées par des instructions différentes, le programme peut être interrompu entre les deux.

Algorithme de Peterson

```
          initialiser D[0]=D[1]=0 et Tour=0
          // drapeau D[i]=1 si Pi est en section critique
Entree(i):| D[i] = 1;                ("je demande")
          | Tour = j                 ("je cède mon tour")
          | Tant_que D[j]==1 ET Tour==j Faire: rien;  (attendre)
          | ret

Sortie(i):| D[i] = 0;                (libérer)
          | ret
```

Algorithme de Peterson 1981

"Tour" ne peut avoir qu'une valeur. Donc, même si les deux instructions ont lieu "en même temps", il n'y a qu'un seul des tests "Tant que D[j]==1 ET Tour==j" qui sera faux, l'autre laissant un processus bloqué.

Complément : Vérification de l'algorithme de Peterson

- Exclusion mutuelle: Si les deux processus sont en section critique, alors (D[2] =0 ou Tour vaut 1) et (D[1]=0 ou Tour =2) et (D[1]=1 et D[2]=1) ce qui est impossible.
- Absence de blocage: Si les deux processus sont bloqués, alors (D[2]=1 et Tour =2) et (D[1]=1 et Tour =1), ce qui est également impossible.
- Progression: La solution étant symétrique, on peut supposer que P2 n'a pas demandé l'entrée en section critique, donc D[2]=0. Si P1 ne peut pas entrer en section critique, alors D[2]=1 et Tour=2, ce qui est une contradiction.
- Attente bornée: Si les deux processus ont demandé l'entrée en section critique, supposons que P1 ait pu y accéder. Donc D[1]=1 et D[2]=1, Tour =1 et P2 est en attente. Si P1 demande une nouvelle fois l'entrée en section critique, après avoir exécuté sa section de sortie, il affecte à D[1] la valeur 1 et à Tour la valeur 2, ce qui lui interdit l'entrée en section critique, attend au plus un passage de l'autre en section critique pour y entrer à son tour.

Remarque : Hypothèse : écriture atomique

L'algorithme de Peterson suppose tout de même que l'écriture d'un mot mémoire (ici de la variable "Tour") est atomique. C'est-à-dire que si P0 écrit dans Tour, quand P1 va lire Tour, il voit la valeur écrite par P0. Cela peut paraître évident, mais ce peut être faux sur des machines multiprocesseurs avec des caches multiples si on ne programme pas correctement (en provoquant volontairement une mise à jour des caches).

Remarque : En pratique ?

En pratique, il existe des mécanismes matériels plus efficaces. Entre autres, certains processeur disposent dans leur jeu d'instruction, une instruction particulière qui permet de tester et mettre à jour un mot mémoire d'une manière atomique. Une telle instruction peut être utilisée pour entrer en section critique tout en assurant l'exclusion mutuelle.

g) Solutions matérielles

Désarmement des interruptions [beauquier]

Dans un système monoprocesseur, la solution la plus simple consiste à désarmer les interruptions pendant toute la durée de la section critique, les réarmer lorsque le processus en sort. En conséquence, l'unité centrale reste allouée à ce processus jusqu'à ce qu'il ait terminé sa section critique, ce qui garantit bien l'exclusion mutuelle. Remarquons qu'une telle méthode ne s'adapte pas à certains systèmes

multiprocesseurs. En effet, si les interruptions sont désarmées sur un processeur particulier, un autre processeur peut parfaitement accéder aux variables partagées. Toutefois, si les interruptions sont gérées par un seul processeur, cette méthode est utilisable. Néanmoins, il est dangereux de rendre le désarmement des interruptions accessible aux processus utilisateurs.

Instruction Test-And-Set [beauquier]

De nombreux processeurs disposent d'une instruction élémentaire exécutée par le matériel, qui permet de lire d'écrire le contenu d'un mot de la mémoire de manière indivisible. Cette instruction est généralement appelée Test-And-Set notée TS ou TAS. Cette instruction a deux opérandes un registre a et un mot b de la mémoire centrale. Elle copie le mot dans le registre a et place la valeur 1 dans le mot b :

```
TAS a, b :
DEBUT
  a :=b ;
  b :=1 ;
END
```

Le point fondamental est que cette instruction est toujours exécutée d'une manière indivisible.

Méthode : Instruction Test-And-Set et le contrôle de section critique

Les processus partagent une variable, que nous appelons verrou. Chaque processus Pi possède une variable locale, notée $Test_i$, et sa section d'entrée consiste à exécuter l'instruction TAS, puis à consulter la valeur de cette variable. Si sa valeur est 1 (signifiant que le verrou est mis), le processus doit recommencer et si sa valeur est 0, il peut entrer en section critique. La variable verrou est initialisée à 0 et le processus Pi exécute :

```
<section restante>
Fermer :
  TAS(Testi , Verrou);
  Si Testi=1 Alors GOTO Fermer ;
  <section critique> /* ici Verrou = 1 */
Libérer :
  Verrou :=0 ;
```

Puisque l'instruction TAS est indivisible, le premier processus qui l'exécute trouve verrou à 0 et peut entrer en section critique. Tant qu'il n'en est pas sorti, un autre processus désirant entrer en section critique trouve verrou à 1 et doit attendre. Lorsque le processus entré en section critique sort, il remet verrou à 0 et le premier processus qui exécute alors une instruction TAS pourra entrer à son tour.

Remarque : Instruction TAS et condition d'attente bornée

Notons que l'instruction TAS de garantit pas l'attente bornée qui doit être assurée par d'autres moyens.

4. Les sémaphores

a) Concept des sémaphores

Définition : Sémaphore

Un sémaphore est un objet "S" définit par le système d'exploitation dont le fonctionnement peut être décrit par deux opérations P et V :

```
P(S): Si S > 0
        | Alors: S = S - 1
        | Sinon: S'endormir
        Fin_Si

V(S): Si (au moins un process bloqué sur S)
        | Alors: En Réveiller UN,
        | Sinon: S = S + 1
        Fin_Si
```

Sémaphore

Ces deux fonctions ont été proposées par Dijkstra en 1965. C'est lui qui leur a donné les noms P et V qui sont les initiales en néerlandais de "verrouiller" et "libérer". P(S) et V(S) sont des opérations garanties indivisibles par le système d'exploitation. C'est-à-dire que deux opérations sont implantées de telle sorte que le SE garanti qu'elles vont s'exécuter complètement (soit jusqu'à la sortie, soit jusqu'à la mise en sommeil) sans qu'un autre processus puisse démarrer leur exécution.

b) Sémaphore et section critique

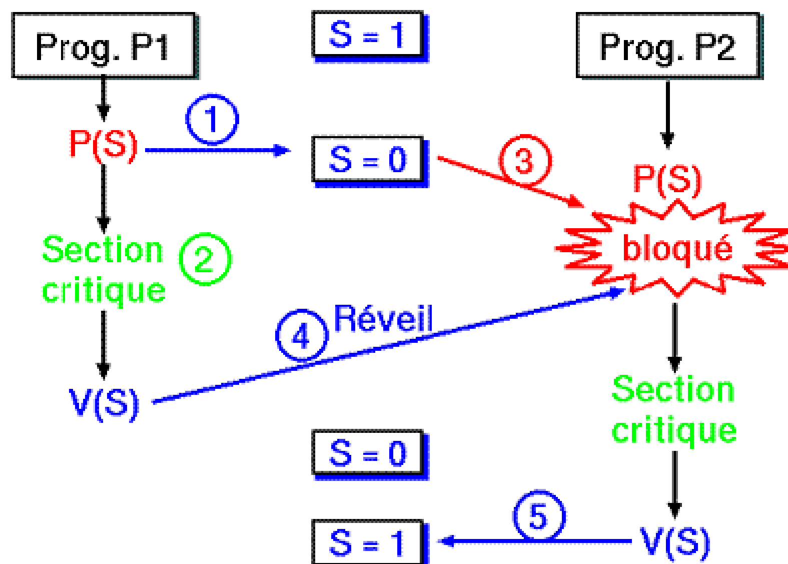
Contrôle de section critique avec un sémaphore

L'opération P(S) correspond à la demande d'entrer en section critique, et l'opération V(S) correspond au signalement de sortie de section critique :

P(S) // entrée

<Section critique>

V(S) // sortie



Contrôle d'une section critique avec un sémaphore

Remarque : Sémaphore unaire et n-aire

Si le sémaphore S est initialisé à 1, on dit que c'est un sémaphore unaire : il

autorise exactement UN processus en section critique (exclusion mutuelle).

S'il est initialisé à N, on parle de sémaphore N-aire qui autorise "N" processus simultanément en section critique (Il y a des cas où ceci est utile).

Remarque : Exécution de P et V par le système

Le système d'exploitation garanti l'exécution complète des fonctions P et V. Bien sûr, un processus en mode utilisateur ne pourrait pas fournir cette garantie, puisqu'il peut être à tout moment interrompu par le système.

c) Implémentation des sémaphores

Structure d'un sémaphore

Un sémaphore peut être représenté par une structure du type suivant :

```
type sémaphore = enregistrement
  valeur : entier ;
  liste_d'attente : liste de PCB ;
fin ;
```

Un sémaphore contient une liste pour stocker les PCB des processus endormis ayant exécuté P(S) alors que S.valeur<= 0.

Implémentation des opérations P et V

Compte tenu de la structure d'un sémaphore présentée ci-dessus, les opérations P et V peuvent être implémentées comme suit :

```
P(S) :
SI S.valeur <=0 ALORS
  DEBUT
    <Ajouter le PCB du processus à S.Liste_d'attente> ;
    <Mettre le processus en attente> ;
    <Interruption logicielle ==> Appel ordonnanceur pour
allouer le CPU à un autre processus> ;
  FIN
SINON S.valeur := S.valeur-1 ;
```

```
V(S) :
SI S.Liste_d'attente non vide ALORS
  DEBUT
    <Choisir et enlever un PCB de S.Liste_d'attente> ;
    <Faire passer à l'état prêt le processus choisi> ;
  FIN
SINON S.valeur := S.valeur +1 ;
```

Attention : Opérations P et V et section critique

Les opérations P et V sur un sémaphore sont supposées être exécutées de manière indivisible, vus qu'elles manipulent des données partagées. Ceci signifie que, pendant qu'un processus exécute une opération P ou V sur un sémaphore S, aucun autre processus ne peut exécuter P ou V sur ce même sémaphore S.

Et donc, pour implémenter dans un système la notion de sémaphore qui permet de résoudre facilement le problème de l'exclusion mutuelle, il faut être capable de réaliser, dans cette implémentation une section critique. A l'évidence, on ne pourra pas utiliser les sémaphores pour implémenter la routine d'implémentation des sémaphores !

Pour cela, le système passe en mode noyau et utilise un mécanisme matériel d'exclusion mutuelle (voir cours précédent).

Le plus souvent on utilise l'instruction Test-And-Set (présentée dans le cours

précédent) pour assurer l'exclusion mutuelle (protéger la section critique) des opérations P et V.

d) Sémaphores sous UNIX

Sémaphore et IPC System V

Les sémaphores ont été implantés sous unix par les primitives des IPC System V : semget(...), semop(...), et semctl(...).

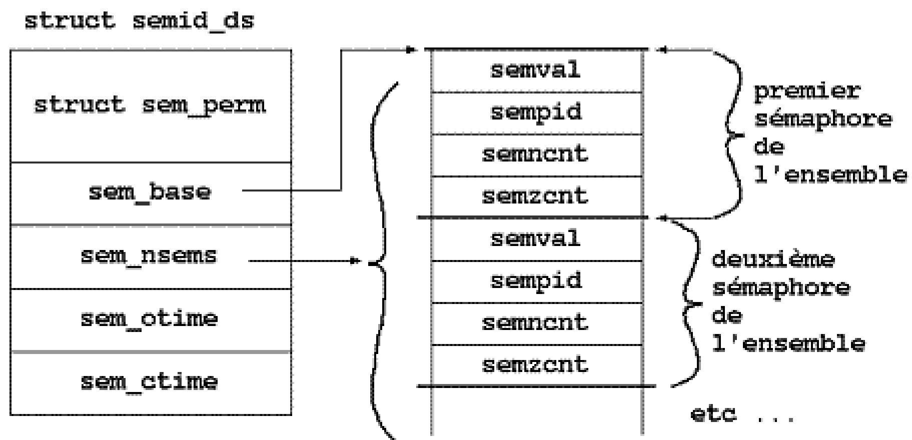
L'implémentation System V des sémaphores est plus complexe que les simples opérations P et V vus qu'elle permet la gestion de plusieurs sémaphores simultanément.

Méthode : Création de sémaphores

La primitive semget(...) va soit créer un ensemble de sémaphores ou récupérer le ID d'un ensemble existant de sémaphores.

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

Lors de la création une structure de type semid_ds est créée et représente dans le système cet ensemble de sémaphores :



Structure d'une entrée dans la table des sémaphores

Méthode : Opérations sur les sémaphores

Les opérations sur un ensemble de sémaphores sont réalisées à travers la primitive semop(...) qui manipulera simultanément un sous ensemble de sémaphores sélectionnés parmi un ensemble de sémaphores préalablement créés et identifiés par un semid :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *tab_op, int nb_op) ;
```

Pour cela, semop prend en deuxième argument un tableau de structures de type struct sembuf :

```
struct sembuf{
  unsigned short int sem_num ; /* numéro de sémaphore cible
de l'opération */
```

```
  short sem_op ; /* opération sur le sémaphore */
  short sem_flg ; /* option : IPC_NOWAIT, SEM_UNDO */
```

L'opération effectuée dépend de la valeur n de sem_op :

- si n est positif : l'opération est Vn. La valeur du sémaphore est alors augmentée de n. Tous les processus en attente de l'augmentation de la valeur de ce sémaphore sont réveillés (leur nombre correspond à la valeur du champ semnctl du sémaphore (struct sem)).
- Si n est nul : l'opération est une opération Z. Le processus est bloqué tant que le sémaphore n'est pas nul.
- si n est négatif : l'opération est Pn. Si l'opération n'est pas possible, le processus est, sauf demande contraire à travers IPC_NOWAIT, bloqué et le nombre de processus en attente de l'augmentation de ce sémaphore est incrémenté. Dans le cas où l'opération est possible, si la valeur du sémaphore devient nulle, tous les processus en attente de la nullité du sémaphore sont réveillés.

En fait, la valeur associée au sémaphore indique le "nombre de places" disponibles dans la section critique. Si ce nombre est inférieur au nombre demandé, alors le processus demandeur est mis en attente. Chacune de ces opérations peut être rendue non bloquante en positionnant IPC_NOWAIT dans le champs de flags sem_flg de cette opération.

Attention : Atomicité de l'ensemble des opérations demandées par semop

L'ensemble des opérations demandées dans un semop est exécuté d'une manière atomique. C'est-à-dire que toutes les opérations sont réalisées ou aucune ne l'est.

Remarque : Opération Z et notion de rendez-vous

L'opération Z permet une implémentation aisée de la notion de rendez-vous :

- on initialise un sémaphore à n,
- chaque processus qui fait un Z sur ce sémaphore provoque une décrémentation de 1,
- pour les premiers arrivés, comme le sémaphore n'est pas nul, ils sont mis en attente,
- le dernier qui arrive fait passer le sémaphore à zéro libérant tous les autres et lui-même

Méthode : Contrôle des sémaphores avec semctl

La primitive semctl permet de réaliser, outre les opérations communes aux xxxctl des IPC System V, des initialisations de sémaphores et la consultation de différentes valeurs (sémaphores, nombre de sémaphores en attente d'augmentation ou de nullité de valeur,... :

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

Selon la valeur du paramètre op, le paramètre semnum sera interprêté soit comme un numéro de sémaphore, soit comme un nombre de sémaphores. La valeur de op détermine également si un paramètre supplémentaire est nécessaire, auquel cas il doit être du type suivant, qui doit être explicitement défini dans l'application :

```
union semun{
```

```
  int val ; // pour SETVAL
  struct semid_ds *buf ; // pour IPC_SET et IPC_STAT
  ushort_t *array ; // pour GETALL et SETALL
} ;
```

op	semnum	union	effet
GETNCNT	numéro	-	Nombre de processus en attente d'augmentation du sémaphore
GETZCNT	numéro	-	Nombre de processus en attente de nullité du sémaphore
GETVAL	numéro	-	Valeur du sémaphore
GETALL	nombre	array	Le tableau array contient la valeur des semnum premiers sémaphores
SETVAL	numéro	val	Initialisation du sémaphore à val
SETALL	nombre	array	Initialisation des semnum premiers sémaphores au contenu de arg
IPC_RMID	-	-	Supression de l'entrée dans la table des sémaphores
IPC_STAT	-	buf	Extraction de l'entrée dans la table des sémaphores
			Modification de l'entrée dans la table

Tableau 1 : Paramètres et comportement de semctl

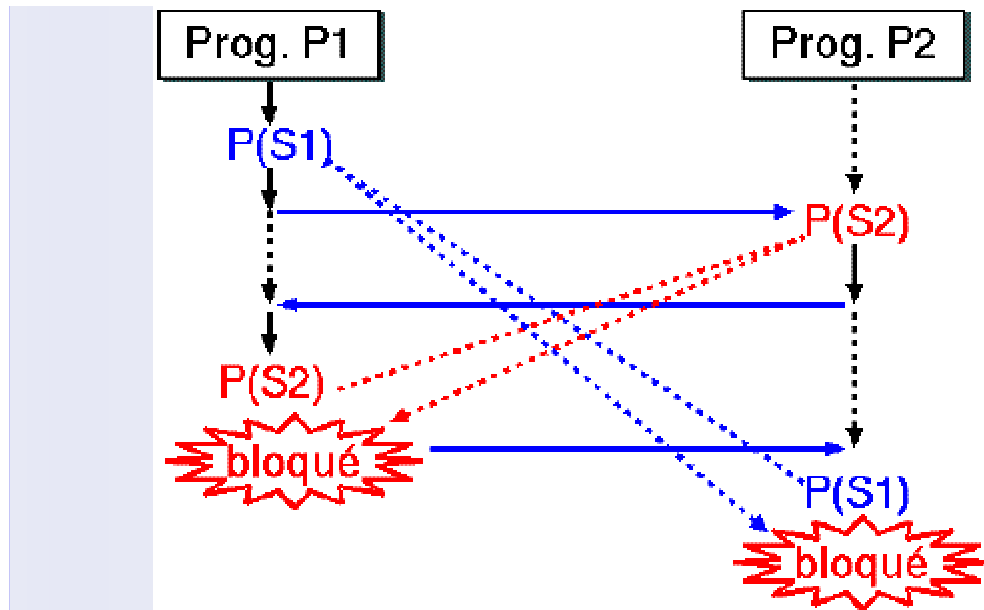
e) Précaution d'utilisation des sémaphores

Attention : **Risque d'interblocage**

Si plusieurs sémaphores sont demandés par des programmes concourants, il faut :

1. soit que tous les programmes demandent les sémaphores dans le même ordre,
2. soit, quand un processus est bloqué :
   - relâcher les ressources déjà acquises,
   - s'endormir en attendant celle qui est occupée,
   - au réveil, refaire la séquence d'acquisition.

Voici un scénario typique de demande de sémaphores provoquant un interblocage :



Interblocage de deux processus

L'API des sémaphores UNIX résout le problème à la place du programmeur par la possibilité de demander l'acquisition d'un ensemble de sémaphores (tous accordés ou aucun).

Attention : Risque d'oubli et difficulté de maintenance

L'utilisation des sémaphores conduit à des algorithmes plutôt non-structurés et donc facilement sujets à erreurs :

- il est facile d'oublier un P() ou un V() dans une des branches de l'algorithme
- il est souvent difficile, à la relecture du code, d'identifier l'utilisation de tel ou tel appel P() ou V().

Ceci a conduit à proposer des mécanismes de plus haut niveau dont la sémantique soit mieux définie, ou dont l'utilisation soit moins sujette à erreur, par exemple les moniteurs de Java.

Définition : Moniteur de Java

Le moniteur est le mécanisme d'exclusion mutuelle de Java. On définit une section critique par le mot-clé synchronized. Il peut porter sur une fonction ou bien sur un bloc.

```
class protectcompt{
  int compteur = 0 ;
  public synchronized void ajouter(int val){
    compteur += val ;
  }
  public int lire(){ return compteur ;}
  }
}
```

L'exécution normale de la fonction ajouter prend plusieurs instructions. Elle peut donc être interrompue au mauvais moment. Le mot-clé synchronized empêche l'exécution simultanée de "ajouter" par plusieurs processus.

5. Problèmes classiques de synchronisation [beauquier]

a) Le problème des producteurs / consommateurs

Définition : Producteur / consommateur

Un processus producteur produit des informations qui sont consommées par un processus consommateur.

Exemple : Clavier

Le processus clavier produit des caractères qui sont consommés par le processus d'affichage.

Exemple : Imprimante

Le pilote de l'imprimante produit des lignes de caractères, consommées par l'imprimante.

Contrainte d'ordre

Une première contrainte provient du fait que les objets sont consommés dans l'ordre où ils ont été produits, c'est-à-dire suivant une politique FIFO.

Contrainte de partage d'un tampon

Pour que les processus puissent s'exécuter en parallèle, avec des rythmes différents pour la production et pour la consommation, ils partagent un tampon, dans lequel le producteur place les objets qu'il produit et à partir duquel le consommateur prend les objets qu'il consomme.

Une seconde contrainte concerne donc l'accès au tampon, qui constitue une section critique pour chacun des processus.

Remarque : Deux variantes du problème

Ce problème admet deux variantes, suivant que le tampon est ou non supposé borné.

Si le tampon n'est pas borné, seul le consommateur doit attendre, quand le tampon est vide.

Si le tampon est borné, il faut, en plus, garantir que le producteur attend lorsque le tampon est plein.

i - Tampon non borné

Actions

On distingue pour le producteur la production d'un objet et l'adjonction de cet objet au tampon, et pour le consommateur, de façon symétrique, le prélèvement d'un objet dans le tampon et la consommation de cet objet.

Sémaphores

Les deux opérations d'accès au tampon doivent être réalisées en exclusion mutuelle, ce qui conduit à définir un sémaphore ex_mut, initialisé à 1. De plus, le fait qu'une opération de dépôt dans le tampon précède une opération de retrait, elle est réalisée par un autre sémaphore, initialisé à 0, que nous appelons nombre_de_places_occupées, et dont l'interprétaion est la suivante : si la valeur du sémaphore est nulle, le tampon est vide, ce qui interdit une opération de retrait. Au contraire une opération de dépôt augmente d'une unité le nombre de places occupées.

Méthode : Algorithmes

Le producteur et le consommateur exécutent donc respectivement :

```
init(ex_mut,1) ;
init(nombre_de_places_occupées,0) ;
```

```
Producteur :
répéter
  produire un objet ;
  P(ex_mut) ;
  ajouter cet objet au tampon ;
  V(ex_mut) ;
  V(nombre_de_places_occupées) ;
jusqu'à faux ;
```

```
Consommateur :
répéter
  P(nombre_de_places_occupées) ;
  P(ex_mut) ;
  prendre un objet dans le tampon ;
  V(ex_mut) ;
  consommer ce objet ;
jusqu'à faux ;
```

ii - Tampon borné

Un sméphore supplémentaire

Dans le cas d'un tampon borné, comportant n emplacements, il suffit de définir un nouveau sémaphore, représentant le nombre de places libres du tampon et dont la valeur est donc initialisée àn. Lorsque cette valeur est nulle, le tampon est plein et les opérations de dépôts sont interdites. Toute opération de retrait entraîne l'augmentation d'une unité, du nombre de places libres.

Méthode : Algorithmes

```
init(ex_mut, 1) ;
init(nombre_de_places_occupées, 0) ;
init(nombre de places_libres, n) ;
```

```
Producteur ;
répéter
  produire un objet ;
  P(nombre_de_places_libres) ;
  P(ex_mut) ;
  ajouter cet objet au tampon ;
  V(ex_mut) ;
  V(nombre_de_places_occupées) ;
jusqu'à faux ;
```

```
Consommateur ;
répéter
  P(nombre_de_places_occupées) ;
  P(ex_mut) ;
  prendre un objet dans le tampon ;
  V(ex_mut) ;
  V(nombre_de_places_libres) ;
  consommer cet objet ;
jusqu'à faux ;
```

b) Problème des lecteurs / rédacteurs

Définition : Lecteur / rédacteur

Le problème des lecteurs/rédacteur est le suivant : il s'agit de partager un fichier (ou une base de données) entre des processus lecteurs et des processus rédacteur. La cohérence est assurée si l'accès en écriture est exclusif, par contre plusieurs lectures peuvent se faire en parallèle.

Remarque : Variantes

On peut distinguer plusieurs variantes selon que les lectures soient plus prioritaires que les écritures et vice versa.

Lecture prioritaire dès la première lecture

```
init(Fichier, 1) ;
init(mutex,1) ;
Rédacteur :
P(Fichier) ;
<écrire>
V(Fichier) ;
Lecteur :
P(mutex) ;
nl :=nl+1 /* nombre de demandes de lecture */
Si nl =1 alors P(Fichier) ;
V(mutex) ;
<lecture>
P(mutex) ;
nl :=nl-1 ;
Si nl=0 alors V(Fichier) ;
V(mutex) ;
```


Y. Challal, H. Bettahar, M. Vayssade, J-M Berenguier/ UTC