Introduction aux systèmes d ’exploitation

Partie 3 : Les processus

Exclusion mutuelle

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

Partie 3 : Les processus

Exclusion mutuelle

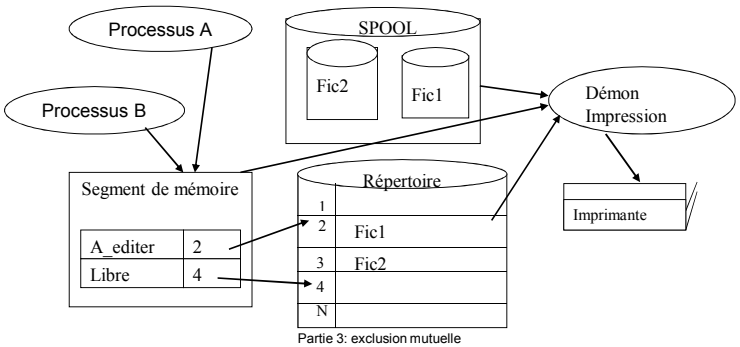
1. Objectif de l’exclusion mutuelle
2. Accès concurents
3. Section critique
4. Exclusion mutuelle par attente active
5. Exclusion mutuelle sans attente active
   1. Le problème du producteur et du consommateur
   2. Les primitives SLEEP et WAKEUP
6. Les sémaphores

1. Objectif de l'exclusion mutuelle

- L'exclusion mutuelle a pour but de limiter l'accès à une ressource à un ou à un nombre donné de processus.
- Ceci concerne, par exemple, un fichier de données que plusieurs processus/utilisateurs désirent mettre à jour.
  - L'accès à ce fichier doit être réservé à **un seul** utilisateur pendant le moment où il le modifie, autrement son contenu risque de ne plus être cohérent.
- On appelle ce domaine d'exclusivité, **une section critique.**

2. Accès concurrents (1)

- Fonctionnement d'un spool d'impression :
  - Quand un processus veut imprimer un fichier, il doit placer le nom de ce fichier dans le ***répertoire de spool***.
  - Un autre processus, le ***démon d'impression***, vérifie périodiquement s'il faut imprimer des fichiers. Si c'est le cas, il les imprime et retire leur nom du répertoire de spool.

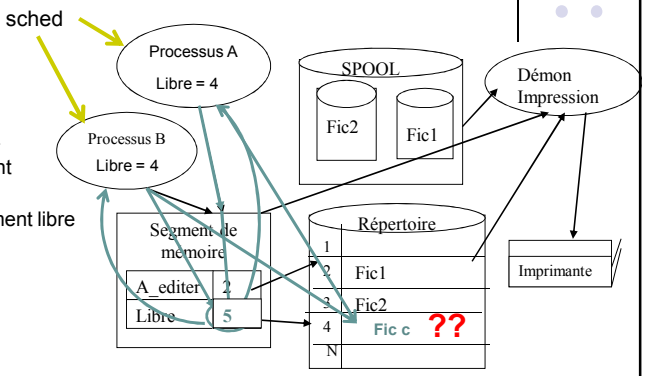


2. Accès concurrents (2)

- On fait les hypothèses suivantes :
  - Le répertoire de spool (d'impression) a un très grand nombre d'emplacements, numérotés de 0 à N.
  - Chacun de ces emplacements peut contenir le nom d'un fichier à imprimer.
  - Il existe deux variables partagées qui sont sauvegardées dans un segment de mémoire partagée et accessible à tous les processus :
    - *« A_editer »* qui pointe sur le prochain fichier à imprimer ,
    - *« Libre »* qui pointe sur le prochain emplacement libre du répertoire.

2. Accès concurrents (3)

- Situation courante
  - La position 1 est libre.
  - Les positions 2 à 3 sont occupées.
  - Le prochain emplacement libre est le 4.



- Imaginons maintenant que pratiquement au même moment, les processus A et B veulent placer chacun un fichier dans la file d'impression.

2. Accès concurrents (4)

- Il risque de se produire la chose suivante :
  - Le processus A lit la variable *libre* et mémorise la valeur 4.
  - Une interruption horloge se produit et l'usage du processeur est retiré au processus A pour être alloué au processus B.
  - Le processus B lit également la variable *libre* qui vaut toujours 4 et place le nom du fichier B à l'emplacement 4. Ensuite le processus B incrémente libre qui passe à 5 puis poursuit ensuite son travail.
  - Au bout d'un certain temps le processus A est relancé par l'ordonnanceur et reprend son exécution.
  - Le processus A place son nom de fichier à imprimer à la position 4 du répertoire en effaçant le nom du fichier qui y avait été mis par B.
  - Le processus A calcule ensuite la place_libre suivante + 1, obtient 5 et place cette valeur dans libre.

2. Accès concurrents (5)

- Le répertoire d'impression est alors dans un état cohérent et le démon d'impression ne détectera pas la moindre anomalie. Pourtant, le fichier du processus B ne sera jamais imprimé.

- Les situations de ce type, où deux processus ou plus
  - lisent et/ou écrivent des données partagées,
  - et où le résultat dépend de l'ordonnancement des processus,

  sont qualifiées **d'accès concurrents**.

3. Section critique

- La partie de programme où peut se produire un conflit d'accès s'appelle une **section critique**.
- Pour éviter les conflits d'accès if faut un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois.
  - En d'autres termes, il faut un **mécanisme d'exclusion mutuelle** qui empêche les autres processus d'accéder à un objet partagé si cet objet est en train d'être utilisé par un processus.
- Toutes les ressources dites partagées au sein d'un système d'exploitation ne peuvent être accédées qu'en EXCLUSION MUTUELLE.
  - Le choix des primitives qui assurent l'exclusion mutuelle est un des **points fondamentaux** de la conception d'un système d'exploitation.

3. Section critique (2)

- Le problème des conflits d'accès est résolu, si on peut s'assurer que ***2 processus ne sont jamais en section critique en même temps.***
- Cette condition est suffisante pour éviter les conflits d'accès mais elle ne permet pas aux processus de coopérer.
- Quatre conditions sont dans ce cas nécessaires :
  1. Deux processus ne peuvent être en même temps en section critique.
  2. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
  3. Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.
  4. Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

4. Exclusion mutuelle par attente active

- Il existe plusieurs méthodes pour réaliser l'exclusion mutuelle par attente active.
- Seules quelques unes de ces méthodes seront présentées dans ce cours.
  1. Le masquage des interruptions
  2. Les variables de verrouillage
  3. L'alternance
  4. L'instruction TSL
  5. L'attente active sous Unix

4.1. Le masquage des interruptions (1)

- Il s'agit d'un moyen matériel qui consiste à masquer les interruptions du processeur.
  1. Le processus masque les interruptions avant d'entrer dans une section critique.
  2. Il les restaure à la fin de la section.
- Conséquences:
  - **L'interruption horloge** ne peut avoir lieu lorsque les interruptions sont masquées.
  - Le processeur ne pourra donc plus être alloué à un autre processus puisque ce sont les interruptions (d'horloge ou autres) qui provoquent le changement d'allocation.

4.1. Le masquage des interruptions (2)

- Deux inconvénients majeurs :
  - Il est très dangereux de permettre aux processus utilisateur de masquer les interruptions car si l'un d'eux oubliait de les restaurer ce serait la fin du système.
  - Si le système possède plus d'un processeur, le masquage des interruptions n'aura lieu que sur le processeur qui a demandé l'interdiction. Les autres processeurs pourront continuer à accéder normalement à la mémoire partagée.

4.2. Les variables de verrouillage (1)

- Soit une variable (ou **verrou**) partagée, unique, qui a initialement la valeur 0 et dont le contenu permettra de savoir s'il y a déjà un processus en section critique :
  - 0 : aucun processus n'est en section critique
  - 1 : il y a déjà un processus en section critique
- Un processus doit tester ce verrou avant d'entrer en section critique.
  - Si le verrou vaut 0, le processus le met à 1 et entre en section critique.
  - Si le verrou est déjà à 1, le processus attend qu'il repasse à 0.

4.2. Les variables de verrouillage (2)

Processus 1	Processus 2

while (verrou ==1) ;
verrou = 1 ;
section_critique();
verrou = 0 ;
	while (verrou ==1) ;
	verrou = 1 ;
	section_critique();
	verrou = 0 ;

- Cette approche comporte un vrai défaut entre
  - le moment où le processus lit le verrou et s 'aperçoit qu'il est à zéro (while (verrou==1)),
  - et le moment où il peut le mettre à 1 (verrou = 1).

4.3. L'alternance (1)

- Soit la variable tour, qui a initialement la valeur 1 et dont le contenu permettra de savoir qui peut rentrer en section critique
  - 1 : le processus 1 peut rentrer en SC
  - 2 : le processus 2 peut rentrer en SC

Processus 1	Processus 2

while (1)
{
while (tour != 1) ;
section_critique () ;
tour = 2 ;
section_non_critique () ;
}
	while (1)
	{
	while (tour != 2) ;
	section_critique () ;
	tour = 1 ;
	section_non_critique () ;
	}

4.3. L'alternance (2)

- Cette approche basée sur l'alternance présente un gros défaut s'il y a une **grande différence de vitesse**.
  - Si le processus 2 sort très rapidement de sa SC, tour vaut 1 et le processus 1 peut rentrer en SC.
  - Les deux processus se trouvent tous les deux dans leur section **non** critique.
  - Le processus 2 reste très longtemps dans sa SNC (Section Non critique).
  - Le processus 1 sort très rapidement de SNC.
    - Exécute de nouveau sa SC car tour vaut 1.
    - Sort de sa section critique et positionne tour à 2.
    - Exécute sa SNC et se retrouve de nouveau en entrée de SC.
    - Ne peut cependant pas rentrer en SC car tour vaut 2.
- Bilan
  - Cette solution va à l'encontre de la règle N°3 : le processus 1 est bloqué par le processus 2 qui n'est pas en SC.

4.4. L'instruction machine TSL (1)

- Si on peut faire une lecture suivie d'une écriture de façon **indivisible**, alors l'algorithme de passage en section critique devient plus simple.
- En réalité, la plupart des processeurs disposent d'une instruction TEST AND SET LOCK (TSL)
  - Elle charge le contenu d'un mot mémoire dans un registre (lecture).
  - Puis met une valeur non nulle à cette adresse mémoire (écriture).
- Les opérations de lecture et d'écriture du mot sont garanties comme étant indivisibles (atomiques).

4.4. L'instruction machine TSL (2)

- Pour illustrer l'instruction TSL, nous nous servirons de la variable partagée *drapeau* :
  - Lorsque drapeau vaut 0, tout processus peut lui donner la valeur 1 au moyen de l'instruction TSL.
  - Ce processus peut ensuite rentrer en section critique.
  - Lorsqu'il a terminé, il remet drapeau à 0 grâce à l'instruction MOVE classique.

4.4. L'instruction machine TSL (3)

- La solution comprend deux sous-programmes
  - *Entrer_SC* et *Sortir_SC*
  - Ecrits dans un langage assembleur fictif.

Entrer_SC	Sortir_SC
boucler : **tsl registre,drapeau** **// registre<-drapeau** **// ET drapeau=1** **cmp registre, #0** **jnz boucler** **ret**	**mov drapeau, #0** **ret**

5. Exclusion mutuelle sans attente active

- L'exclusion mutuelle avec attente active consomme beaucoup de *temps processeur*.
  - Le processus exécute une ***boucle infinie*** jusqu'à ce qu'il soit autorisé à entrer en section critique.
  - Cette approche doit en général être **évitée**.
- Pour éviter cela, les systèmes d'exploitation disposent de primitives IPC (Inter Process Communication) qui se bloquent au lieu de perdre du temps processeur, lorsqu'elles ne sont pas autorisées à entrer en SC.
  - Les primitives SLEEP et WAKEUP.
  - Les sémaphores.

5.1. Le producteur et le consommateur (1)

- Deux processus se partagent un tampon de données de taille fixe (N) initialement vide :
  - Un premier processus (le **PRODUCTEUR**) produit des données et les écrit dans le tampon.
  - Un second processus (le **CONSOMMATEUR**) consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture.

5.1. Le producteur et le consommateur (2)

- Les problèmes surviennent lorsque :
  - Le producteur veut mettre des informations alors que la mémoire tampon est déjà pleine.
    - Il devra attendre (dormir) en attendant d'être réveillé par le consommateur.
    - Lorsque ce dernier aura retiré une ou plusieurs informations.
  - De la même manière si le consommateur tente de retirer une information alors que la mémoire tampon est vide.
    - Il devra dormir en attendant d'être réveillé par le producteur.
    - Lorsque ce dernier aura déposé une ou plusieurs informations.

5.2. Les primitives SLEEP et WAKEUP (1)

```
#include "prototypes h"

#define N       100         /* nbre d'emplacement dans le tampon*/

int compteur = 0 ;    /*nbre d'objet dans le tampon*/

void producteur(void)
{
      int objet;
      while (TRUE)
      {
            produire_objet(&objet);
            if (compteur == N)
                  SLEEP () ;       /*si tampon plein dormir*/
            mettre_objet(objet); /*mettre l'objet dans le tampon*/
            compteur = compteur + 1;/*incrémenter le nbr d'objets*/
            if (compteur == 1) /*réveiller le consommateur si besoin*/
                  WAKEUP (consommateur);
      }
}
```

5.2. Les primitives SLEEP et WAKEUP (2)

```
#include "prototypes h"

#define N     100

int compteur = 0 ;

void consommateur (void)
{
        int objet;
        while (TRUE)
        {
            if (compteur == 0)
                SLEEP() ;                /*si tampon vide dormir*/
            retirer_objet(objet);        /*retirer l'objet du tampon*/
            compteur = compteur - 1;  /*decrementer le compteur*/
            if (compteur == N-1) /*réveiller le producteur si besoin*/
                WAKEUP (producteur);
            consommer_objet (objet) ; /*utiliser l'objet ..*/
        }
}
```

interruption

5.2 Les primitives SLEEP et WAKEUP (3)

- Cette solution ne marche pas car le signal WAKEUP envoyé à un processus qui ne dort pas **est perdu**:
  - La mémoire tampon est vide et le consommateur vient de trouver la valeur 0 dans le compteur.
  - L'ordonnanceur décide à cet instant de suspendre le consommateur avant qu'il n'ait eut le temps de s'endormir.
  - Le producteur est alors relancé :
    - Il met un objet dans le tampon.
    - Il incrémente le compteur.
    - Il constate qu'il vaut désormais 1 et réveille donc le consommateur.
    - Le signal WAKEUP est perdu car le consommateur ne dormait pas.
  - Le consommateur est relancé par l'ordonnanceur :
    - Il testera la valeur du compteur qu'il avait lu avant d'être suspendu.
    - Il trouvera 0 et se mettra en sommeil.
  - Le producteur finira par remplir la mémoire et ira lui aussi dormir. Les 2 processus dormiront …. pour toujours.

6. Les sémaphores

1. Généralités sur les sémaphores
2. Mécanismes fournis par le noyau
3. Le problème des producteurs et des consommateurs
4. Les sémaphores sous UNIX
5. Les commandes SHELL de manipulation des sémaphores

6.1 Généralités sur les sémaphores (1)

- Un sémaphore est un mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée.
  - Sur les voies ferrées, un sémaphore empêche deux trains d'entrer en collision sur un tronçon de voie commun.
  - Sur les voies ferrées comme en informatique, les sémaphores ne sont **qu'indicatifs**.

6.1 Généralités sur les sémaphores (2)

- Un **sémaphore binaire** n'a que deux états :
  - 0 verrouillé (ou occupé),
  - 1 déverrouillé (ou libre).

- Un **sémaphore général** peut avoir un très grand nombre d'états car il s'agit d'un compteur qui :
  - Décroît d'une unité quand il est acquis (« verrouillé »).
  - Croît d'une unité quand il est libéré (« déverrouillé »).
  - Quand il vaut zéro
    - un processus tentant de l'acquérir doit attendre qu'un autre processus ait augmenté sa valeur
    - car il ne peut jamais devenir négatif.

6.1 Généralités sur les sémaphores (3)

- L'accès à un sémaphore se fait généralement par deux opérations :
  - P pour l'acquisition en néerlandais: *Proberen*, tester.
  - V pour la libération: *Verhogen*, incrémenter

- Un moyen mnémotechnique :
  - P(uis-je) accéder à une ressource.
  - V(as-y) la ressource est disponible.

6.1 Généralités sur les sémaphores (4)

- Si nous devions écrire en C ces deux primitives nous aurions le résultat suivant.

P	*V*
Void P (int *sem) **{** ** while (*sem <= 0) ;** ** (*sem) -- ;** **}**	**Void V (int *sem)** **{** ** (*sem) ++ ;** **}**

6.2 Un mécanisme fourni par le noyau

- Lorsqu'un processus effectue l'opération **P(sema)** :
  - si la valeur de *sema* est supérieure à 0
    - il y a alors des ressources disponibles, P(séma) décrémente séma
    - et le processus poursuit son exécution,
  - sinon ce processus sera mis dans **une file d'attente** jusqu'à la libération d'une ressource.
- Lorsqu'un processus effectue l'opération **V(sema)** :
  - si il n'y a pas de processus dans la file d'attente, V(sema) incrémente la valeur de sema,
  - sinon un processus en attente est débloqué.

6.3 Les producteurs et les consommateurs (1)

- Synchronisation des processus producteur et consommateur grâce à des sémaphores :
  - Le producteur et le consommateur doivent accéder de manière exclusive au tampon, le temps d'une lecture ou d'une écriture.
    - Un sémaphore d'exclusion mutuelle est donc nécessaire : ***mutex***.
  - Les ressources du producteur sont les **emplacements vides** du tampon qu'il est donc possible de matérialiser.
    - Un sémaphore production dont la valeur initiale correspond à la taille du tampon.
  - Les ressources du consommateur sont les **emplacements pleins** du tampon qu'il est donc possible de matérialiser.
    - Un sémaphore consommation dont la valeur initiale est nulle.

6.3 Les producteurs et les consommateurs (2)

```
#define N 100

séma mutex = 1;
séma production = N;
séma consommation = 0;

void consommateur()
{
    while (1)
      {
        P(consommation);  /*une place pleine en moins*/
        P(mutex);         /*section critique*/
        vider_case();
        V(mutex);         /*fin de section critique*/
        V(production);    /*une place vide en plus*/
    }
}
```

6.3 Les producteurs et les consommateurs (3)

```
#define N 100

séma mutex = 1;
séma production = N;
séma consommation = 0;

void producteur()
{
    while (1)
    {
        P(production);        /*une place vide en moins*/
        P(mutex);             /*section critique*/
        remplir_case();
        V(mutex);             /*fin de section critique*/
        V(consommation) ; /*une place pleine en plus*/
    }
}
```


6.4 Les sémaphores sous UNIX

1. Les primitives
   1. ftok
   2. semget
   3. semop
   4. semctl
2. Exemple de manipulation

6.4.1 la primitive ftok

- Le problème de la construction des clés des objets manipulés est un problème centrale en particulier du point de vue de la portabilité des applications développées.
- Un appel à la fonction *ftok* génère à partir de la référence *ref* et de l'entier *num* une clé unique.

```
#include <sys/ipc.h>
key_t ftok(const char *ref, int num);
```


6.4.2. La primitive semget

- Il s'agit d'une primitive dont l'appel est un préalable à toute utilisation d'un ensemble de sémaphores par un processus.
- La primitive ***semget*** traduit une *clé* en un identificateur représentant un ensemble de sémaphores.

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

- A partir d 'une clé, elle permet :
  - la création d'un nouvel ensemble de sémaphores,
  - ou la recherche de l'identification d'un ensemble existant,
- Si le bit IPC_CREAT de *option* est positionné, l'ensemble est créé s'il n'existe pas déjà.

6.4.3. La primitive semop

- C'est par son intermédiaire qu'un processus connaissant l'identification d'un ensemble de sémaphores pourra réaliser des ensembles d'opérations sur les sémaphores de cet ensemble.
- Les nb_op opérations placées dans tab_op sont réalisées atomiquement, c'est-à-dire qu'elles le sont toutes ou qu'aucune ne l'est.

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


La structure sembuf

- Elle correspond à une opération sur un sémaphore :
  - L'augmenter d'une valeur entière.
  - La diminuer d'une valeur entière.
  - Attendre qu'il devienne nul.

```
struct sembuf {
     unsigned short  int sem_num ; /* numéro de sémaphore */
     short               sem_op ;  /* opération sur le sémaphore */
     short               sem_flg ;  /* option*/
} ;
```

- Il est important de noter que :
  - Les numéros de sémaphores commencent à 0.
  - C'est le signe du champ sem_op qui détermine l'opération :
    - Négatif => opération P
    - Positif => opération V
    - 0 => opération Z

6.4.3. La primitive semop (2)

- La nature de l'opération est définie par la valeur n du champ *sem_op* de l'élément *sembuf*.
  - Si n est strictement positif (opération V), la valeur du sémaphore est augmentée de n.
  - Si n est négatif (opération P) deux cas peuvent se présenter :
    - L'opération n'est pas réalisable, le processus est, sauf demande contraire, bloqué.
    - L'opération est possible.
- Le champ *sem_flgs* permet de définir des options.
  - IPC_NOWAIT permet de rendre une opération non bloquante et c'est au travers du code retour que l'on peut connaître le résultat.
    - 0 : opération réussie
    - -1 : opération échouée

6.4.4. La primitive semctl (1)

- Cette primitive permet
  - d'une part d'accéder aux informations contenues dans l'entrée de la table des sémaphores,
  - et d'autre part d'en modifier certains attributs :
    - suppression d'un ensemble de sémaphores,
    - initialisation de sémaphores,
    - …

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

6.4.4. La primitive semctl (2)

- Effet de la primitive semctl et interprétation des différents paramètres en fonction de la valeur du paramètre op.

op	semnum	arg	Valeur de retour et effet
GETNCNT	N° sémaphore		Nb sémaphores en attente d'augmentation du sémaphore.
GETZNCNT	N° sémaphore		Nb sémaphores en attente de la nullité du sémaphore.
GETVAL	N° sémaphore		Valeur du sémaphore.
GETALL	Nb sémaphores	Tableau d'entiers courts non signés	0 ou –1 Le tableau arg contient les valeurs des semnum premiers sémaphores.
GETPID	N° sémaphore		Identification du dernier processus ayant réalisé une opération sur le sémaphore.
SETVAL	N° sémaphore	entier	0 ou –1 Initialisation du sémaphore à arg.
SETALL	Nb sémaphores	Tableau d'entiers courts non signés	0 ou –1 Initialisation des semnum premiers sémaphores au contenu de arg.
IPC_RMID			Suppression de l'entrée dans la table des ensembles de sémaphores.
IPC_STAT		Pointeur sur struct semid_ds	Extraction de l'entrée dans la table des ensembles de sémaphores.
IPC_SET		Pointeur sur struct semid_ds	Modification de l'entrée dans la table des ensembles de sémaphores.

6.4.5 Exemple de manipulation des sémaphores (1)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
 ushort semarray[2] ;
 int idSema ;
 int res ;
 struct sembuf sops[2];
 main ( ) {

        key_t cle;
       if (cle = ftok('/home/ens/kiki/toto', 1) == -1)
            { printf("error ftok \n"); exit(-1);}

        /* Création d'un ensemble de 2 sémaphores */
        If (idSema = semget(cle,2,IPC_CREAT | 0600) == -1)
           { printf("error semget \n"); exit(-2);}

        /* Initialisation des 2 sémaphores */
        semarray[0] = 1;
        semarray[1] = 2;
        if (semctl(idSema, 2, SETALL, semarray);==-1)
        { printf("error semctl \n"); exit(-2);}
        …
```

6.4.5 Exemple de manipulation des sémaphores (2)

```
        /* P sur le 1er sémaphore */
        sops [0].sem_num = 0 ;
        sops [0].sem_op = -1 ;
        sops [0].sem_flgs = 0 ;

        /* V sur le 2ème sémaphore */
        sops [1].sem_num = 1 ;
        sops [1].sem_op = +1 ;
        sops [1].sem_flgs = 0 ;

        /* Exécution des opérations P et V */
        If (res = semop (idSema, sops, 2)==-1)
        { printf("error semop \n"); exit(-3);}
        /* Suppression de l'ensemble de sémaphore */
        If (semctl(idSema, 2, IPC_RMID)==-1)
        { printf("error semctl \n"); exit(-4);}
       return 0; /* exit(0); */
}
```

6.5 Les commandes SHELL de manipulation des sémaphores

- Les deux commandes shell essentielles pour la manipulation des sémaphores sont :
  - ipcs –s pour visualiser l'état des sémaphores.
  - ipcrm –s pour la destruction d'un ensemble de sémaphore.
- Il s'agit des commandes standards de consultation et de destruction de tous les IPC :
  - Files de messages (q).
  - Segments de mémoire partagée (m).
  - Ensemble de sémaphores (s).

6.5.1 La commande ipcs

- Elle permet la consultation des tables d'IPC du système.
  - T : le type de l'objet :
  - ID : identification interne de l'objet.
  - KEY : la clé de l'objet (sous forme hexadécimale).
  - MODE : les droits d'accès à l'objet.
  - OWNER : l'identité du propriétaire de l'objet.
  - GROUP : l'identité du groupe propriétaire de l'objet.

```
mini2440@mini2440-VirtualBox:~$ ipcs

------ Segment de mémoire partagée --------
clef       shmid      propriétaire perms      octets     nattch     états
0x00000000 0          mini2440   777         98304      2          dest
0x00000000 32769      mini2440   777         3047424    2          dest
0x00000000 65538      mini2440   777         17028      2          dest
0x00000000 98307      mini2440   777         17424      2          dest
0x00000000 131076     mini2440   777         25476      2          dest
0x00000000 163845     mini2440   777         25212      2          dest
0x00000000 262150     mini2440   777         26532      2          dest
0x00000000 294919     mini2440   777         26532      2          dest
0x00000000 327688     mini2440   777         23496      2          dest
0x00000000 360457     mini2440   777         30360      2          dest
0x00000000 458762     mini2440   777         20196      2          dest
0x00000000 425995     mini2440   600         393216     2          dest
0x00000000 491532     mini2440   777         27324      2          dest
0x00000000 524301     mini2440   777         34056      2          dest
0x00000000 557070     mini2440   777         25080      2          dest

------ Tableaux de sémaphores --------
clef       semid      propriétaire perms      nsems

------ Files de messages --------
clef       msqid      propriétaire perms      octets utilisés messages

mini2440@mini2440-VirtualBox:~$
```

47

6.5.2 La commande ipcrm

- Elle permet de demander la suppression d'une entrée dans une des tables.
- L'entrée à supprimer peut être désignée :
  - soit par la clé associée,
  - soit par son identification interne.
- Seul le créateur ou propriétaire d'un objet peut demander sa suppression.
- La commande « ipcrm –s 100 » est la demande de suppression de l'ensemble de sémaphore dont l'identification est 100.