# Notes sur la gestion des sémaphores

*Master 1$^{iere}$ année, Bruno Jacob, IC2*

## 1 Introduction

Les sémaphores constituent un mécanisme de synchronisation de processus. Ils sont utilisés pour contrôler l'accès à des sections critiques dans le code du programme qui est exécuté par un processus.

### 1.1 Rappel du problème

Des applications s'exécutant de manière concurrente et partageant des ressources communes peuvent avoir besoin d'accéder de manière exclusive à une ou plusieurs de ces ressources. On appelle ces ressources des **ressources critiques**. Le code exécuté nécessitant cet accès exclusif est appelé une **section critique**.
Une analogie courante est de comparer cela avec une route qui emprunte un pont ne supportant la charge que d'une voiture. Or la longueur et la largeur du pont font qu'il est susceptible d'être emprunté par plusieurs véhicules (plusieurs voitures proches dans le même sens, voitures dans les deux sens...). Le bon fonctionnement (et la survie des usagers) suppose qu'il faut trouver un mécanisme pour que un seul véhicule puisse utiliser, à un instant donné, la section de route correspondant au passage du pont. Ici la route représente le code et le sémaphore le mécanisme qui contrôle l'accès au pont.

D'un point de vue informatique, ce problème est fréquent. Citons-en quelques exemples:

- suite d'opérations dans un fichier ou une base de données;
- accès à une ressource telle qu'une imprimante;
- accès à une zone de mémoire centrale (segment de mémoire partagée) par plusieurs processus;
- exécution d'un appel système mettant en jeu des tables du système

## 1.2 Exemple

Pour illustrer le problème considérons l'application suivante dans laquelle deux activités concurrentes incrémentent chacune un grand nombre de fois (ici 10.000.000) une variable partagée `p_cpt` (qui joue ici le rôle de la ressource partagée) représentant un pointeur balayant un segment de mémoire partagée.

CODE

```
int *p_cpt;

static
void fonc()
{
      int m = 0;
      while(m < 10000000)
      {
            m++;
            (*p_cpt)++;
      }
      printf("m = %d *p_cpt = %d\n", m, *p_cpt);
}

int
main()
{
      int pid, shmid;

      /*-------*/

      /* Creation ressource critique */
      /* -- creation segment partage */
      shmid = shmget(IPC_PRIVATE, sizeof(int), 0666);
      /* -- rattachement */
      p_cpt = (int *) shmat(shmid, NULL, 0);
      /* -- init */
      *p_cpt = 0;

      /* Creation de 2 processus pere/fils */
      pid = fork();

      /* Boucle dans les 2 processus */
      fonc();

      /* Arret du fils */
      if(pid == 0)
            exit(0);

      /* Dans le pere */
```

```
        /* -- attente fin du fils */
        wait(NULL);
        /* -- affichage ressource critique */
        printf("Valeur finale de *p_cpt : %d\n", *p_cpt);
        /* -- destruction segment memoire partage */
        shmctl( shmid , IPC_RMID , 0 ) ;

        exit(0);
}
```

Exécutions

```
m = 10000000 *p_cpt = 7981312
m = 10000000 *p_cpt = 8642116
Valeur finale de *p_cpt : 8642116

m = 10000000 *p_cpt = 7299687
m = 10000000 *p_cpt = 11139260
Valeur finale de *p_cpt : 11139260

m = 10000000 *p_cpt = 1561030
m = 10000000 *p_cpt = 8853725
Valeur finale de *p_cpt : 8853725
```

En principe, on est en droit de s'attendre à ce que la variable `p_cpt` soit incrémentée 20.000.000 fois (10.000.000 fois par chaque processus). Or les trois exécutions d'une part ne donnent pas toutes le même résultat, d'autre part ne donne pas le résultat attendu. Ceal vient du fait que l'opération `*p_cpt++` n'est pas atomique. Du point de vue du processeur, elle correspond à une séquence de plusieurs opérations (chargement d'un registre, incrémentation de ce registre et sauvegarde du registre). Par exemple, pour incrémenter la variable `p_cpt` chacun des processus doit

1. charger la valeur de la variable `p_cpt` dans un registre
2. incrémenter le registre
3. sauvegarder le registre dans la zone mémoire affectée à `p_cpt`

Cette séquence est interruptible et donc un changement de contexte est susceptible de se produire entre chacune de ces opérations. Par exemple, admettons que `p_cpt` vaut 10, et voyons ce qui peut se passer pour une incrémentation de chacun des 2 processus:

1. chargement de `p_cpt` par le père (`p_cpt`/père $==$ 10)
2. *interruption* $\rightarrow$ *fils* : chargement de `p_cpt` par le fils (`p_cpt`/fils $==$ 10)

3. incrémentation registre dans le fils (`p_cpt`/fils == 11)
4. sauvegarde registre du fils (`p_cpt` == 11)
5. *interruption* → *père* : incrémentation registre dans le père (`p_cpt`/père == 11)
6. sauvegarde registre du père (`p_cpt` == 11)

On se retrouve avec la variable `p_cpt` ayant pour valeur 11 au lieu de 12.

# 2 Solutions

## 2.1 Solution 1 : incorrecte

**Variable partagée** : une variable `ZONE` qui peut avoir les états `LIBRE` ou `OCCUPE`

**Idée** : quand un processus entre dans la section critique, il met `ZONE` à `OCCUPE`, quand il en sort il remet `ZONE` à `LIBRE`. Un processus ne peut pas entrer dans la section critique quand `ZONE` est à `OCCUPE`.

**Avantage** : simple

**Inconvénients** : ne résout pas le problème. Nous n'avons fait que le déplacer de l'incrémentation du compteur vers l'affectation du booléen.

### 2.1.1 Code

```
/* Gestion de la Section Critique */
#define LIBRE  0
#define OCCUPE 1
int * p_zone ;

/* Section Critique */
unsigned long int * p_cpt ;

static
void fonc()
{
      int m = 0;
      /*-------*/
      while(m < 10000000)
      {
            m++;
            while( (*p_zone) == OCCUPE ) ; /* Entree */
```

```
            (*p_zone) = OCCUPE ;
            (*p_cpt)++;
            (*p_zone) = LIBRE ;/* Sortie */
        }
        printf("m = %d *p_cpt = %lu\n", m, *p_cpt);
}
```

### 2.1.2 Exécution

```
m = 10000000 *p_cpt = 8561133
m = 10000000 *p_cpt = 16324328
Valeur finale de cpt : 16324328
```

Résultat incorrect avec un temps d'exécution:

```
real        1.9
user        2.5
sys         0.0
```

## 2.2 Solution 2 : insatisfaisante

**Variable partagée** : une variable `tour`

**Idée** : indiquer au processus quand c'est son tour d'entrer en section critique

**Avantage** : simple

**Inconvénients** :

- Ne satisfait pas la contrainte de continuïté, c'est à dire qu'un processus qui ne veut pas entrer en section critique ne bloque pas les autres.
- Attente active
- Exécution longue

### 2.2.1 Code

```
/* Gestion de la Section Critique */
pid_t *p_tour ;

/* Section Critique */
unsigned long int *p_cpt;
```

```
static
void fonc( pid_t lui ,
           pid_t moi )
{
     int m = 0;
     /*-------*/

     while(m < 10000000 )
     {
          m++;
          while( (*p_tour) != moi ) ; /* Entree */
          /*       printf("entree de %ld en SC\n", moi ); */
          (*p_cpt)++;
          /* printf("sortie de %ld en SC\n", moi ); */
          (*p_tour) = lui ; /* Sortie */
     }
     printf("m = %d *p_cpt = %lu\n", m, *p_cpt);
}
```

### 2.2.2 Exécution

```
m = 10000000 *p_cpt = 19999999
m = 10000000 *p_cpt = 20000000
Valeur finale de cpt : 20000000
```

On voit que l'exécution de cette solution donne un résultat correct mais avec un temps peu performant:

```
real       16.5
user       31.4
sys         0.0
```

## 2.3 Solution 3 : interblocage

**Variable partagée** : un tableau `veut_entrer[2]`

**Idée** : chaque participant à une variable booléenne qui indique qu'il veut entrer en section critique. Les variables de tous les participants sont lisibles par tous. Ainsi si un participant veut entrer en section critique, il mettra sa variable a `VRAI` et attendra que les variables de tous les autres soient à `FAUX`

**Avantage** : cela garanti que les 2 processus ne peuvent entrer en même temps dans la section critique

**Inconvénients** : cela peut amener à un interblocage si les 2 processus mettent leur variable `veut_entrer` à `VRAI` en même temps, sans avoir testé la valeur de la variable de l'autre

### 2.3.1 Code

```
/* Gestion de la Section Critique */
int * veut_entrer ;

/* Section Critique */
unsigned long int *p_cpt;

/* Handler signal SIGINT */
void hand_int( int sig)
{
  /* -- destruction segment memoire partagee */
  shmctl( shmid_sc    , IPC_RMID , 0 );
  shmctl( shmid_entre , IPC_RMID , 0 );
  exit(0);
}

static
void fonc( int lui ,
           int moi )
{
     int m = 0;
     /*------*/

     printf( "lancement de %d\n" , moi );

     while(m < 10000000 )
     {
          m++;
          veut_entrer[moi] = VRAI ;/* Entree */
          while( veut_entrer[lui] ) ;
          printf("entree de %d en SC\n", moi );
          (*p_cpt)++;
          printf("sortie de %d en SC\n", moi );
          veut_entrer[moi] = FAUX ; /* Sortie */
     }
     printf("m = %d *p_cpt = %lu\n", m, *p_cpt);
}
```

### 2.3.2 Exécution

L'exécution conduit rapidement à une situation d'interblocage:

```
lancement de 1
entree de 1 en SC
sortie de 1 en SC
entree de 1 en SC
sortie de 1 en SC
lancement de 0
```

```
entree de 1 en SC
sortie de 1 en SC
entree de 1 en SC
sortie de 1 en SC
....(bloque).....
```

## 2.4 Solution 4: correcte (Dekker, 1965)

**Variable partagée** : une variable `tour` + un tableau `veut_entrer[2]`

**Idée** : combiner les 2 solutions précédentes

**Avantage** : solution correcte avec un coût d'exécution raisonnable par rapport aux précédentes

**Inconvénients** : l'exécution du code de l'entrée dans la section critique induit de l'attente active

### 2.4.1 Code

```
#define FAUX 0
#define VRAI 1

/* Gestion de la Section Critique */
int * veut_entrer ;
int * p_tour ;

/* Section Critique */
unsigned long int *p_cpt;

static
void fonc( int lui ,
           int moi )
{
     int m = 0;
     /*-------*/

     while(m < 10000000 )
     {
          m++;

          /* Entree */
          veut_entrer[moi] = VRAI ;
          while( veut_entrer[lui] )
          {
            if( (*p_tour) == lui )
```

```
            {
              veut_entrer[moi] = FAUX ;
              while( (*p_tour) == lui ) ;
              veut_entrer[moi] = VRAI ;
            }
          }

          /* Section critique */
          (*p_cpt)++;

          /* Sortie */
          veut_entrer[moi] = FAUX ;
          (*p_tour) = lui ;
      }
      printf("m = %d *p_cpt = %lu\n", m, *p_cpt);
}
```

### 2.4.2 Exécution

L'exécution de cette solution donne un résultat correct

```
m = 10000000 *p_cpt = 18932250
m = 10000000 *p_cpt = 20000000
Valeur finale de cpt : 20000000
```

avec un temps meilleur que dans la solution de 2.2.

```
real        5.3
user       10.2
sys         0.0
```

# 3 Généralisation à N processus

Dijkstra a donné une généralisation de la solution de Dekker à $N$ processus.

## 3.1 Théorie

Un sémaphore $S$ est une variable entière $\geqslant 0$ manipulable par 2 opérations P et V:

$$\begin{cases} P(S) & \text{si } S = 0, \text{ alors mettre le processus en attente, sinon } S \leftarrow S - 1 \\ V(S) & S \leftarrow S + 1 \text{ ; réveil d'un processus en attente} \end{cases}$$

Toute l'implantation de ce mécanisme repose:

- sur l'atomicité des opérations P et V, c'est à dire sur le fait que ces opération ne peuvent pas être interrompues
- sur l'existence d'un mécanisme permettant
  - de mémoriser et mettre en attente les processus ayant demandés une opération P qui n'a pas pu être satisfaite (quand $S$ va devenir négatif)
  - de réveiller ces processus quand leur opération P peut être satisfaite (lors d'une opération V)

Une section critique, ou une ressource critique, est protégée par un sémaphore $S$. L'entrée dans cette section critique est gérée en faisant une opération P sur le sémaphore $S$ (notée $P(S)$) et la sortie par l'opération V sur ce même sémaphore (notée $V(S)$)

## 3.2 Réalisations

Il y a différentes façons de réaliser des sémaphores sous Unix qui n'induisent pas d'attente active.

### 3.2.1 Réalisation par signaux

Pour permettre à un autre processus d'entrer dans la section critique, le processus actif lui adresse un signal `SIGUSR1`. Afin que celui-ci ne soit pas délivré trop tot au processus l'attendant, le signal est masqué en dehors de l'opération P.
Inconvénients :

- la variable sémaphore peut difficilement être supérieure à 1
- généralisation à N processus peu aisée
- temps d'exécution réel très long

Ce mécanisme de synchronisation s'apparente plus à un passage de témoin entre 2 processus.

### 3.2.2 Réalisation par tubes

Le sémaphore est un tube ordinaire. Sa valeur de départ est le nombre de caractères qu'il contient. Le blocage de l'opération P est réalisé par une lecture d'un caractère dans ce tube.

### 3.2.3 Réalisation par file de messages

Le sémaphore est une file des messages. Sa valeur de départ est le nombre de messages d'un certain type Le blocage de l'opération P est réalisé par une demande d'extraction d'un message de ce type.

### 3.2.4 Réalisation par sémaphores

Ils constituent le mécanisme de synchronisation par excellence puisqu'ils sont faits pour ça.
Leur utilisation est cependant un peu compliquée car ils permettent de réaliser atomiquement des ensembles d'opérations sur des ensembles de sémaphores et leurs structures sont donc plus complexes. L'idée est que l'on puisse manipuler plusieurs sémaphores, correspondant a plusieurs ressources, en même temps.
La réalisation de l'exclusion mutuelle du compteur est donnée en exemple d'application des sémaphores Unix en 6.4

# 4 Généralisation à N processus et M ressources

C'est la solution adoptée par le système d'exploitation Unix pour modéliser les sémaphores.

Il faut donc poser **atomiquement** un ensemble de sémaphores différents contrôlant l'accès à un ensemble de ressources différentes. La version *System V* de Unix (celle dont nous disposons ici) a organisé chaque entrée de sa table comme une structure de liste avec entête: un élément de cette liste est un sémaphore individuel qui contrôle l'accès à une ressource. Un sémaphore est identifié par un entier qui est unique dans l'ensemble auquel il appartient.
Une opération se rapporte à un sémaphore. Pour cela on associe les deux par une structure `sembuf` qui défini un couple *{sémaphore,opération}*.
Les primitives de gestion de la table des sémaphores permettent d'effectuer atomiquement un ensemble de couples `sembuf`. La figure 1 schématise ce principe.

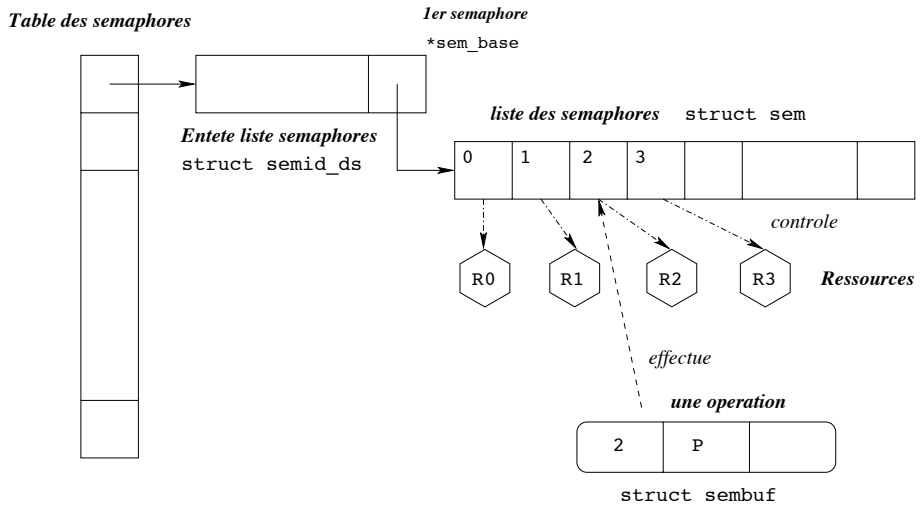


Figure 1: Schéma de la table des sémaphores

# 5 Structures

Les structures des sémaphores peuvent être trouvées dans le fichier

`/usr/include/sys/sem.h`

## 5.1 Structure `semid_ds`

L'entête de la liste (ou ensemble) des sémaphores contient entre autres les champs :

```
struct semid_ds {
  struct ipc_perm sem_perm;  /* droits d'acces */
  struct sem      *sem_base; /* ptr sur 1er semaphore */
  ushort_t        sem_nsems; /* nb de semaphore dans la liste */
  time_t          sem_otime; /* date derniere operation */
  time_t          sem_ctime; /* date dernier changement par semctl */
  ...
};
```

## 5.2 Structure `sem`

Cette structure correspond à un élément de la liste des sémaphores, c'est à dire à un sémaphore individuel. Elle contient les champs:

```
struct sem {
  ushort_t semval;   /* la variable semaphore */
  pid_t    sempid;   /* pid dernier processus ayant appele semop */
  ushort_t semncnt;  /* nb processus attendant une valeur > 0*/
  ushort_t semzcnt;  /* nb processus attandant une valeur = 0*/
  ...
};
```

A noter que c'est le champ `semval`, qui contient le nombre de processus qui peuvent entrer dans la section critique à un moment donné.

## 5.3 Structure `sembuf`

C'est la définition d'une opération que l'on peut faire sur un sémaphore

```
struct sembuf {
  ushort_t  sem_num;   /* numero du semaphore */
  short     sem_op;    /* operation sur le semaphore */
  short     sem_flg;   /* options de l'operation */
};
```

Les opérations que l'on peut effectuer sur un sémaphore sont donc des entiers. L'interprétation de leur valeur est donnée au § 6.2.1.

# 6 Primitives

Les primitives de gestion des sémaphores sont préfixées par `sem`. On retrouve les primitives de gestion des objets IPC avec `get` et `ctl` et les opérations d'entrée et de sortie de sections critiques sont réalisées par `semop`.

## 6.1 Primitive `semget`

```
int semget(key_t cle,   /* cle utilisateur */
           int nbsems, /* nb semaphores */
           int option);/* option creation */
```

Permet de retrouver l'identification d'un ensemble de `nbsems` sémaphores de clé utilisateur `cle`, avec création s'il n'existe pas. Les sémaphores ont pour numéro 0,1,2,3,...`nbsems-1`.

## 6.2 Primitive `semop`

```
int semop( int             semid,   /* ensemble semaphores */
           struct sembuf *p_op,    /* tableau d'operations */
           size_t          nbops); /* nb operations */
```

Elle permet de réaliser `atomiquement` les `nbops` opérations contenues dans le tableau `p_op` dans l'ensemble de sémaphores identifié par `semid`. Le processus est mis en sommeil si l'une des opérations du tableau ne pas être effectuées. Retour :

- 0 en cas de réussite
- -1 sinon

### 6.2.1 Nature des opérations

Une opération est une valeur qui peut être

- $> 0$ : opération du type V, sortie d'une section critique
  - La valeur du sémaphore augmente
  - Les processus en attente que la valeur de ce sémaphore augmente sont réveillés
- $= 0$ : test pour savoir si la valeur du sémaphore est nulle
  - Le processus sera bloqué si la valeur n'est pas nulle
- $< 0$ : opération du type P, entrée dans une section critique
  - La valeur du sémaphore diminue sans aller en dessous de zéro
  - Si ce n'est pas possible, alors le processus est bloqué jusqu'à ce que la valeur du sémaphore augmente.
  - Si la valeur devient nulle alors tous les processus en attente que la valeur soit nulle sont réveillés

### 6.2.2 options des opérations

C'est le troisième champ de `sembuf`

- `IPC_NOWAIT` : rend l'opération non bloquante
- `SEM_UNDO` : permet de supprimer les blocages de processus quand un autre processus s'est terminé anormalement.

## 6.3 La primitive `semctl`

```
int semctl( int semid, /* identification ens. semaphores */
            int semnum, /* num. ou nb semaphore(s) */
            int op,     /* type d'operation a faire */
            union semun{
                        int valeur ;
                        struct semid_ds *buffer,
                        unsigned short *tableau } arg );
```

Réalise différentes commandes. Les paramètres `semnum` et `arg` sont interprétés différemment en fonction de l'opération.

### 6.3.1 Paramètres de `semctl`

Le tableau ci-après donne un résumé des interprétations des paramètres de `semctl` en fonction des opérations indiquées par `op`.

| Opération | `semnum` | Interprétation de `arg` | Valeur de retour Effet |
|---|---|---|---|
| `GETNCNT` | Numéro d'un sémaphore | - | Nombre de processus en attente d'augmentation du sémaphore |
| `GETZCNT` | Numéro d'un sémaphore | - | Nombre de processus en attente de nullité du sémaphore |
| `GETVAL` | Numéro d'un sémaphore | - | Valeur du sémaphore |
| `GETALL` | Nombre de sémaphores | tableau | 0 si OK -1 sinon. Le tableau contient les `semnum` premiers sémaphores |
| `GETPID` | Numéro d'un sémaphore | - | Dernier processus ayant réalisé un `semop` |
| `SETVAL` | Numéro d'un sémaphore | valeur | 0 si OK -1 sinon. Init. du sémaphore à la valeur |
| `SETALL` | Nombre de sémaphores | tableau | 0 si OK -1 sinon. Init. des `semnum` premiers sémaphores |
| `IPC_STAT` | - | buffer | 0 si OK -1 sinon. Extraction de l'entrée de la table des sémaphores |
| `IPC_SET` | - | buffer | 0 si OK -1 sinon. Affectation de l'entrée de la table des sémaphores |
| `IPC_RMID` | - | - | Suppression de l'entrée de la table des sémaphores |

### 6.3.2 Exemples d'utilisation de `semctl`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#define NB_SEM 5
int
main()
{
  int semid ;
  int val = 5 ;
  unsigned short tab[NB_SEM] = { 2 , 4 , 6 , 8 , 10 };
```

```
struct semid_ds buf ;
int i ;
/*-----*/

/* Creation ensemble de NB_SEM semaphores avec cle PRIVEE */
if( (semid = semget( IPC_PRIVATE, 10 , IPC_CREAT | 0666 )) == -1 )
{
  perror("Pb creation semaphore");
  exit(-1) ;
}
/*
 * Exemple d'utilisation avec valeur :
 * semaphore numero 2 <-- 5
 */
if( semctl( semid , 2 , SETVAL , val ) == -1 )
{
  perror("Pb semctl SETVAL");
  exit(-1) ;
}
val = 0 ;
val = semctl( semid , 2 , GETVAL ) ;
printf("Semaphore numero 2 de l'ensemble %d --> %d\n" , semid , val );
/*
 * Exemple d'utlisation avec tableau
 * les NB_SEM semaphores sont initilises avec les valeurs de tab
 */
if( semctl( semid , NB_SEM , SETALL , tab ) == -1 )
{
  perror("Pb semctl SETALL");
  exit(-1) ;
}
for( i=0 ; i<NB_SEM ; i++ ) tab[i] = 0 ;
if( semctl( semid , NB_SEM , GETALL , tab ) == -1 )
{
     perror("Pb semctl GETALL");
    exit(-1) ;
}
for(i=0 ; i<NB_SEM ; i++ )
     printf("Semaphore numero %d --> %d\n" , i , tab[i] );
/*
 * Exemple d'utilisation avec buffer:
 * recuperation des caracteristiques des semaphore dans buf
 */
if( semctl( semid , 0 , IPC_STAT , &buf ) == -1 )
{
  perror("Pb semctl IPC_STAT");
  exit(-1) ;
}
printf("Ensemble %d:\n" ,semid );
```

```
    printf("\t- Nombre de semaphores = %d\n" , buf.sem_nsems );
    printf("\t- UID = %d\n" , buf.sem_perm.uid );
    printf("\t- GID = %d\n" , buf.sem_perm.gid );
    /*
     * Exemple utilisation sans arg
     * Destruction semaphore
     */
    semctl( semid , 0 , IPC_RMID , 0 );
    exit(0);
}
```

## 6.4 Exemple d'utilisation des sémaphores Unix

Pour illustrer cela nous reprenons l'exemple du partage du compteur avec la réalisation du sémaphore selon l'implémentation d'Unix. Nous créons ici un ensemble composé d'un seul sémaphore individuel.

### 6.4.1 Code

```
static int Sem_id ;
static struct sembuf Op_P = {0 , -1 , 0 } ; /* operation P: -1 sur le sem 0 */
static struct sembuf Op_V = {0 ,  1 , 0 } ; /* operation V: +1 sur le sem 0 */

static
void init_semaphore()
{
     int val_init = 1 ;
     /*-------*/
     /* Creation d'un ensemble contenant 1 semaphore */
     if( (Sem_id = semget( IPC_PRIVATE, 1 , 0666 )) == -1 )
     {
          perror( "init_semaphores: Pb semget\n");
          exit(-1) ;
     }
     /* Initialisation du semaphore avec un jeton */
     semctl( Sem_id , 0 , SETVAL , val_init );
}

static
void fin_semaphore()
{
     /* Destruction de l'ensemble contenant le sempahore */
     semctl( Sem_id , 0 , IPC_RMID , 0 );
}

static
void P()
{
```

```
        semop( Sem_id , &Op_P , 1) ;
}

static
void V()
{
        semop( Sem_id , &Op_V , 1) ;
}

/* Section Critique */
unsigned long int *p_cpt;
```

### 6.4.2 Temps d'exécution

```
real        4.5
user        2.7
sys         2.0
```