Chapitre 5 - Synchronisation des Processus

Table des matières

1 - Introduction

Les processus étant dans des espaces mémoires séparés, il est nécessaire d'introduire des mécanismes de communication pour plusieurs raisons :

- Echanger et/ou partager des informations entre processus,
- Protéger les accès concurrents aux ressources partagées,
- Dépendances entre processus,

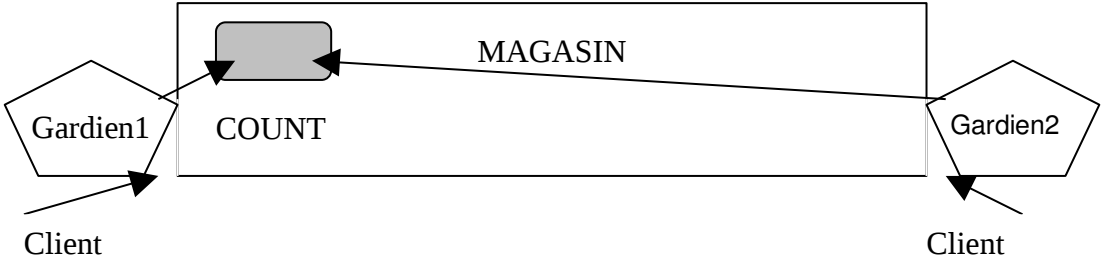
Traditionnellement il existe les mécanismes IPC (InterProcess Communication) suivants :

- **Files de messages** : Qui permet d'échanger des messages entre processus,
- **Mémoire partagée** : Qui permet de déclarer un espace mémoire commun à plusieurs processus,
- **Sémaphores** : Pour gérer les sections critiques que nous allons détailler,

2 - Concurrence

La notion de concurrence est caractérisée lorsque plusieurs entités cherchent à accéder à une même ressource au même instant.

Imaginons deux gardiens situés aux deux entrées d'un magasin, qui comptent le nombre de clients au cours de la journée en incrémentant, au fur et à mesure, une variable commune nommée « count ».



On peut représenter l'activité des gardiens par deux processus « gardien1 » et « gardien2 » correspondant à deux procédures identiques donc le texte est :

```
tant que <le magasin est ouvert> faire
    si <entrée> alors
        count = count + 1
    fin si
fin tant que
```

Le programme parallèle décrivant cette activité est alors :

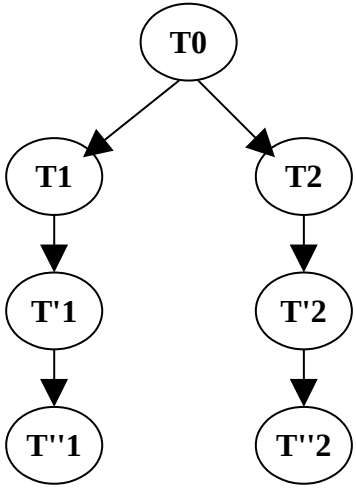
```
début
    count = 0
    <ouvrir le magasin>
    parbegin
        Gardien1;
        Gardien2;
    parend
    <afficher count>
fin
```

Le processus d'initialisation va initialiser la variable « count » et ouvrir la magasin, on va considérer pour simplifier qu'il réalise une tâche T0=d0f0.

Le processus Gardien1 va exécuter une chaîne de tâches T1=d1f1, T'1=d'1f'1, T"1=d"1f"1 … qui sont les incrémentations successives de la variable « count » correspondant aux entrées de la porte surveillée par le premier gardien

Le processus Gardien2 va exécuter une chaîne de tâches T2=d2f2, T'2=d'2f'2, T"2=d"2f"2 … qui sont les incrémentations successives de la variable « count » correspondant aux entrées de la porte surveillée par le second gardien.

Le graphe de correspondance sera alors :



On va maintenant examiner le début de l'exécution de ce système de tâches, prenons le cas où par chaque porte d'entrée est entré un seul client :

Comportement	Valeur du compteur
d0f0d1d2f2f1	Count = 1
d0f0d1d2f1f2	Count = 1
d0f0d2d1f2f1	Count = 1
d0f0d2d1f1f2	Count = 1
d0f0d1f1d2f2	Count = 2
d0f0d2f2d1f1	Count = 2

Pour que la variable « count » contienne la valeur correcte, il est indispensable de rendre les tâches T1 et T2 indivisibles ou encore atomiques.

On voudrait avoir l'un des (sous)graphes de précédence suivant :

3 - Section critique et exclusion mutuelle

Quelques définitions :

- **Ressource critique** : c'est une entité dont l'utilisation ne doit être faite que pour un seul processus à la fois, donc en accès exclusif,
- **Section critique** : désigne une séquence d'actions d'un processus pendant laquelle il doit être seul à utiliser la ressource critique.
- **Exclusion mutuelle** : il s'agit d'une condition de fonctionnement qui assure à un processus l'usage exclusif d'une ressource critique.

Dans le problème précédent, la **ressource critique** est la variable « count », l'opération « `count = count + 1` » est ce que l'on appelle une **Section Critique**. En effet, un seul processus ne doit accéder au même moment à cette variable, dans une condition d'**Exclusion Mutuelle**.

Pour résoudre le problème précédent, plusieurs solutions sont envisageables :

- **Utiliser une variable de verrou** : Par exemple une variable « lock » qui prend la valeur 0 pour indiquer que l'on peut entrer en section critique, et une valeur 1 pour indiquer qu'un autre processus y est déjà :

```
while (lock == 1) /* Attente */ ;
lock = 1 ;
/* Section Critique */
lock = 0 ;
```

→ Même problème sur la variable « lock » que sur la variable partagée.

- **Alternance stricte** : Dans le cas de deux processus, une variable indique le numéro du processus qui a le droit de rentrer en section critique, elle est modifiée à la fin de chaque section critique :

`while (choix!=1) /* Attente */;`
`/* Section Critique */`
`choix = 2 ;`
	`while (choix!=2) /* Attente */;`
	`/* Section Critique */`
	`choix = 1 ;`

→ Cette solution pose des problèmes notamment si il n'y a pas d'alternance stricte pour l'entrée en section critique.

- **Solution de Peterson** : L'idée est toujours d'utiliser une variable représentant le numéro du processus, mais également un tableau d'état composé de deux cases, représentant les besoins des deux processus de rentrer en section critique. Cette solution résout le problème précédent mais ne résout pas un autre problème : **L'attente active**.

L'attente active par le biais d'une boucle consomme du temps CPU !!

4 - Sémaphores

4.1 - Définition

Les sémaphores constituent la solution proposée par Dijkstra pour résoudre le problème de **l'exclusion mutuelle** : plusieurs processus jouant le jeu pourront ainsi s'assurer l'usage exclusif d'une ressource particulière en utilisant un sémaphore associé à celle-ci, en évitant l'attente active.

Le principe réside dans l'utilisation de deux opérations :

- **Opération P** : Qui décrémente la variable sémaphore, si cette variable est égale à 1 le processus est mis en sommeil,
- **Opération V** : Qui incrémente la variable sémaphore, et réveil les processus en sommeil,

Les opérations P et V sont donc les suivantes :

- Opération P(S)

```
Si (S == 0) alors sommeil ;
S = S - 1 ;
```

- Opération V(S)

```
S = S + 1 ;
Réveil d'un seul processus en sommeil ;
```

La particularité de ce mécanisme est qu'il est réalisé par le système d'exploitation, de manière atomique, ce qui garantit son fonctionnement.

Si l'on reprend l'exemple des gardiens, le programme principal sera :

```
début
    count = 0
    semaphore = 1
    <ouvrir le magasin>
    parbegin
        Gardien1;
        Gardien2;
    parend
    <afficher count>
fin
```

Et le code des gardiens :

```
tant que <le magasin est ouvert> faire
    si <entrée> alors
        P(semaphore)
        count = count + 1
        V(semaphore)
    fin si
fin tant que
```

5 - Les sémaphores Unix « System V »

5.1 - La gestion des clés

La primitive `ftok` construit une clé à partir d'une référence de fichier existant et d'un nombre entier : attention si le fichier est déplacé, la clé sera modifiée ! ! !

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

L'exemple de la page suivante montre la création d'une clef à partir d'une référence fournie sur la ligne de commande du programme.

Une clé particulière `IPC_PRIVATE` réserve l'utilisation de l'objet au processus créateur et à sa descendance.

5.2 - Exemple de création de clé System V

```
/* C. Drocourt - LIS - Ch8 - ipc1.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>

int main(int argc,char *argv[]) {
  key_t cle;

  if(argc != 3) {
    fprintf(stderr,"syntaxe:%s <ref_cle> <nbr>\n",argv[0]);
    exit(1);
  }

  if((cle = ftok(argv[1],atoi(argv[2]))) == -1) {
    fprintf(stderr,"Probleme de cle\n");
    exit(2);
  }
}
```

5.3 - Les commandes IPC System V

En principe, les applications doivent supprimer les objets qu'elles n'utilisent plus en appelant les primitives appropriées (`semctl`) avec l'option `IPC_RMID` mais il est néanmoins prévu de supprimer ces objets sur la ligne de commandes avec :

```
ipcrm  -s <semid>
```

La commande « `ipcs` » permet la consultation des trois tables d'I.P.C. du système, de plus, l'option « `-l` » donne les limites alors que l'option « `-u` » donne la consommation actuelle.

Information sur les sémaphores :	`ipcs -s`
Destruction d'un ensemble de sémaphores :	`ipcrm -s <semid>`

5.4 - Les sémaphores « System V »

L'implémentation System V des sémaphores est plus complète encore puisque les différentes primitives utilisées manipulent non pas un sémaphore unique mais des ensembles de sémaphores.

On peut donc résoudre le problème de l'acquisition simultanée d'exemplaires multiples de plusieurs ressources différentes ! ! !

5.5 - La table des sémaphores

Chaque ensemble de sémaphores possède un **semid** (semaphore identification) qui référence une entrée dans cette table.

5.6 - La primitive semget

Elle permet, soit de créer un nouvel ensemble de sémaphores, soit d'accéder à un ensemble déjà existant. Dans les deux cas, la primitive renvoie un nombre entier qui est le **semid** de l'ensemble de sémaphores ou –1 en cas d'échec.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t clef, int nb, int drap) ;
```

Le second paramètre est le nombre de sémaphores de l'ensemble : ces sémaphores seront numérotés de 0 à *nb*-1.

La clef d'accès à l'ensemble de sémaphores est passée en premier paramètre.

Cas général : Elle est calculée avec la fonction **ftok** .

Cas particulier : Elle est égale à la macro-constante **IPC_PRIVATE** : le **semid** du nouvel ensemble ne pourra pas être demandé ultérieurement au système et par conséquent, seule la filiation du processus créateur aura connaissance (par héritage) de ce **semid** et sera en mesure d'accéder à l'ensemble.

Néanmoins un processus connaissant cette identification pourra accéder quand même à l'ensemble si les droits d'accès sont suffisants.

Dans ce cas le troisième paramètre ne sert qu'à préciser les droits d'accès.

Le troisième paramètre règle le comportement de la primitive :

- création d'un ensemble : **IPC_CREAT| *0xyz***
- création exclusive : **IPC_CREAT|IPC_EXCL|*0xyz***
- recherche d'un ensemble : **0**

5.7 - La primitive semctl

Cette primitive permet de consulter/modifier la structure **semid_ds** mais aussi d'initialiser les sémaphores, de consulter leurs valeurs et d'autres données associées à un ensemble de sémaphores.

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

Le second champ est le numéro du sémaphore dans l'ensemble, le troisième précise le type de l'opération de contrôle réalisée et le quatrième est constitué de données supplémentaires adaptées à l'option choisie.

5.8 - La primitive semop

La structure **sembuf** correspond à une opération (P/V/Z) sur un sémaphore.

```
struct sembuf
{
 /* numéro de sémaph. ds l'ens    */
 unsigned short int  sem_num ;
 short           sem_op ;  /* opération sur le sémaphore */
 short           sem_flg ; /* option  */
}
```

Le signe de **sem_op** détermine l'opération :

- Signe négatif : opération Pn
- Signe positif : opération Vn
- Valeur nulle : opération Z.

La primitive **semop** permet la réalisation atomique d'un ensemble d'opérations P, V et Z sur un ensemble de sémaphores : si une des opérations ne peut être réalisée sur le champ, toutes les opérations déjà effectuées par la primitive sont annulées.

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

Le troisième paramètre précise le nombre d'opérations à réaliser.

Le second est l'adresse de la première opération.

5.9 - Exemple de programme de sémaphore System V

```
/* C. Drocourt - ipc2.c */
int main(int argc,char *argv[]) {
  key_t cle;
  struct sembuf operation;
  int semid;

  if(argc != 3) {
    fprintf(stderr,"syntaxe:%s <ref_cle> <nbr>\n",argv[0]);
    exit(1);
  }
  /* Le fichier doit exister avant !!! */
  if((cle = ftok(argv[1],atoi(argv[2]))) == -1) {
    perror("Probleme de cle ");
    exit(2);
  }
  /* Suite du programme */

  exit(0);
}
```

5.10 - Exemple de création de sémaphores System V

```
/* Création du sémaphore */
if((semid=semget(cle, 1, IPC_CREAT|IPC_EXCL|0600)) == -1) {
  perror("Erreur de création de sémpahore");
  exit(1);
}
/* Initialisation du sémaphore à 0 */
semctl(semid,0,SETVAL,0);

printf("T1 - Début section critique, 5 secondes... \n");
sleep(20);

printf("T1 : Fin section critique, operation V\n");
operation.sem_num =  0;
operation.sem_op  =  1;
operation.sem_flg =  0;
if (semop(semid,&operation,1)==-1)  {
  perror("Impossible d'incrémenter le sémaphore");
  exit(3);
}
printf("Fin de T1\n");
```

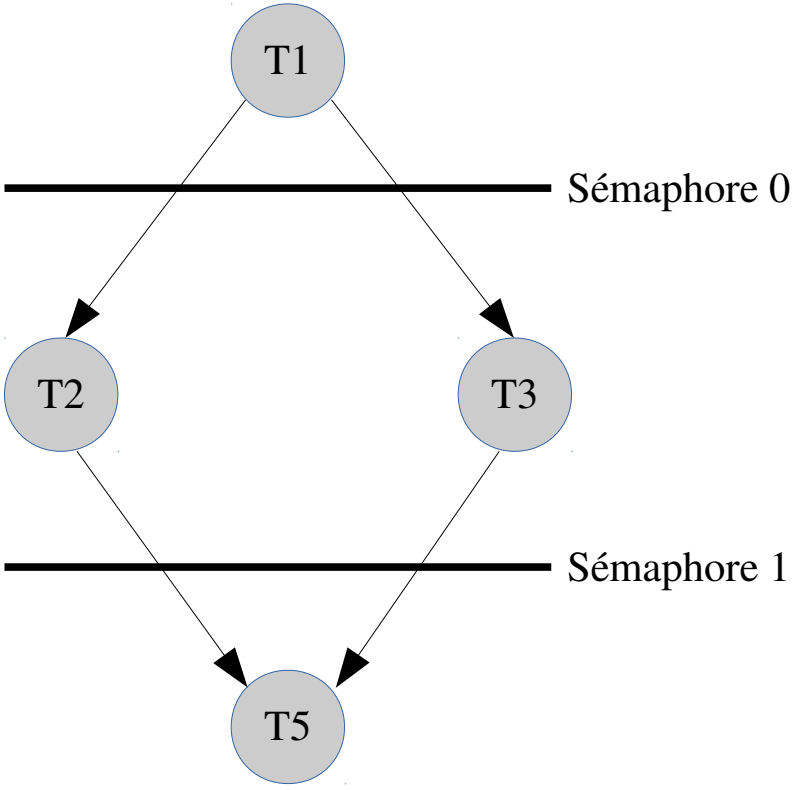
5.11 - Exemple d'utilisation de sémaphore System V

```
/* Attachement au sémaphore */
if((semid=semget(cle,1,0))==-1) {
  perror("Erreur d'accès au sémaphore");
  exit(1);
}

printf("T2 : Attente du sémaphore, opération P ... \n");
operation.sem_num =  0;
operation.sem_op  = -1;
operation.sem_flg =  0;
if (semop(semid,&operation,1)==-1)  {
  perror("Impossible de décrémenter le sémaphore");
  exit(3);
}
printf("T2 : Début section critique 5 secondes ... \n");
sleep(10);
printf("Fin de T2\n");

/* Destruction du sémaphore */
semctl(semid,0,IPC_RMID);
```

5.12 - Exemple de synchronisation avec des sémaphores System V



Fonctions P et V

```
/* C. Drocourt - ipc3.c */
int P(int semid, int ns) {          /* Opération P */
  struct sembuf oper;

  oper.sem_num=ns; oper.sem_op=-1; oper.sem_flg=0;
  if (semop(semid,&oper,1)==-1)  {
    perror("\nImpossible de décrémenter le sémaphore");
    exit(3);
  }
}

int V(int semid, int ns) {          /* Opération V */
  struct sembuf oper;

  oper.sem_num=ns; oper.sem_op=1; oper.sem_flg=0;
  if (semop(semid,&oper,1)==-1)  {
    perror("\nImpossible d'incrémenter le sémaphore");
    exit(4);
  }
}
```

Programme

```
int main(void) {
  int semid,i,status; pid_t retour;

  /* Création des deux sémaphores */
  if((semid=semget(IPC_PRIVATE,2
      ,IPC_CREAT|IPC_EXCL|0600))==-1) {
    perror("\nErreur de création de sémpahores");
    exit(1);
  }

  /* Initialisation des deux sémaphores à 0 */
  semctl(semid,0,SETVAL,0); semctl(semid,1,SETVAL,0);

  for (i=1;i<=4;i++) {
    if ((retour=fork())==-1) {
      perror("\nEchec de fork");
      exit(2);
    }
    if (retour==0) {
      switch(i) {
```

```
        case 1 : printf("Début de T1 ... \n");
                 sleep(3);
                 V(semid,0); V(semid,0);
                 printf("\n ... Fin de T1\n");
                 exit(0);
        case 2 : P(semid,0);
                 printf("\nDébut de T2 ... \n");
                 sleep(20);
                 V(semid,1);
                 printf("\n ... Fin de T2\n");
                 exit(0);
        case 3 : P(semid,0);
                 printf("\nDébut de T3 ... \n");
                 sleep(1);
                 V(semid,1);
                 printf("\n ... Fin de T3\n");
                 exit(0);
        case 4 : P(semid,1); P(semid,1);
                 printf("\nDébut de T4 ... \n");
                 sleep(2);
                 printf("\n ... Fin de T4\n");
                 exit(0);
```

```
      }
    }
  }

  /* Attente des 4 processus */
  for (i=1;i<=4;i++) wait(&status);

  /* Destruction des deux sémaphores */
  semctl(semid,0,IPC_RMID);

  exit(0);
}
```