TP n° 3

Programmation des Sémaphores

1. Rappel : les sémaphores

Les sémaphores sont des outils généraux de synchronisation qui servent, par exemple, à gérer l'accès en exclusion mutuelle à une ressource critique. Un sémaphore est représenté par une valeur entière **E** et une file d'attente **F**. Prenons, par exemple, le cas d'une ressource critique partagée par plusieurs processus et dont la quantité est limitée. La quantité de cette ressource disponible initialement sera donnée par la valeur du sémaphore. Si nous disposons de *n* unités de ressources initialement, alors, la valeur initiale du sémaphore sera *n*.

2. Opérations sur les sémaphores

	Algorithme	Signification
P(S)	**S.E = S.E-1** **si (S.E < 0) alors** **Bloquer le processus** **dans S.F**	Prise de la ressource. Si la ressource n'est pas disponible, le processus est bloqué.
V(S)	**S.E = S.E+1** **si (S.E <= 0) alors** **Débloquer un processus** **de S.F**	Libération de la ressource. Certains processus bloqués pourront reprendre si la ressource nécessaire est à nouveau disponible.

*Opération **P(S)**.*

Lorsqu'un processus nécessite une ressource pour fonctionner, il va tenter de puiser dans la quantité disponible. La quantité de ressources va donc être décrémentée de *1*. Si après cette opération la quantité de ressources est négative, cela veut dire que le processus ne peut pas s'exécuter correctement : il est alors suspendu dans l'attente d'une quantité suffisante de ressources.

*Opération **V(S)***

Lorsqu'un processus libère une ressource, les demandes des processus bloqués sont réexaminées et certains sont débloqués.

3. Programmation des sémaphores en C

L'implémentation des sémaphores disponible sous GNU/Linux propose les deux opérations **P** et **V** ainsi qu'une troisième notée **Z**. Un processus qui effectue une opération **Z** est suspendu jusqu'à ce que la valeur du sémaphore associé soit égale à **Z**éro.

Il est absolument fondamental de remarquer que **toute opération sur les sémaphores est atomique** : en d'autres termes, une fois que le processus entre dans une opération traitant des sémaphores, alors il ne peut pas être interrompu.

3.1 Structure sembuf

La structure **struct sembuf** permet de définir une opération sur un sémaphore particulier de l'ensemble des sémaphores. Elle a la définition suivante :

```
struct sembuf
{
  short sem_num; /* Numéro du sémaphore */
  short sem_op;  /* Opération P, V ou Z */
  short sem_flg; /* Options */
};
```

sem_num : Numéro du sémaphore sur lequel va s'appliquer l'opération. Dans un ensemble de sémaphores, les numéros commencent à 0.

sem_op : C'est le champ qui détermine l'opération :

- **sem_op = 0 :** Opération **Z**
- **sem_op = 1 :** Opération **V**
- **sem_op = -1 :** Opération **P**

sem_flg : Options sur l'opération. Son étude est hors de ce TP.

3.2. Création de sémaphores

La fonction, s'appelle **semget**, sa syntaxe est la suivante :

```
int semget(key_t clef, int nbrSem, int options)
```

Pour simplifier, une seule option spécifique est présentée : **nbrSem** qui permet de spécifier combien de sémaphores doivent être crées. La fonction retourne l'identifiant du sémaphore crée.

3.3 Initialisation de sémaphores

La fonction **semctl** permet de gérer les sémaphores. Syntaxe :

```
int semctl(int idSEM, int numSem, int opr, int val, ...)
```

Avec **idSEM** l'identifiant du sémaphore. **numSem** est le numéro du sémaphore (le numéro commence de zéro). La valeur de **opr** détermine l'opération a faire : **SETVAL** permet de donner une valeur, **val**, au sémaphore.

3.4 Opérations sur les sémaphores

La fonction **semop** permet de réaliser les opérations **P**, **V** et **Z** sur les sémaphores. La syntaxe générale est la suivante :

```
int semop(int idSEM, struct sembuf * ensOps, int nbOps)
```

Avec **idSEM** l'identifiant du sémaphore. **ensOps** est l'adresse de la structure **sembuf** et **nbOps** le nombre d'opérations à effectuer, généralement 1.

Remarque : Toutes ces fonctions sont définies dans la bibliothèque **sys/sem.h**

4. Applications

Soit le programme suivant :

```
#include <stdio.h>
#include <sys/sem.h>
void P(int semid) {
  struct sembuf op;
  op.sem_num = 0; op.sem_op = -1; op.sem_flg = 0;
  semop(semid, &op, 1) ;
}
void V(int semid) {
  struct sembuf op;
  op.sem_num = 0; op.sem_op = 1; op.sem_flg = 0;
  semop(semid, &op, 1);
}
void Z(int semid) {
  struct sembuf op;
  op.sem_num = 0; op.sem_op = 0; op.sem_flg = 0;
  semop(semid, &op, 1);
}
void init(int semid, int E) {
  semctl(semid, 0, SETVAL, E);
}
int main() {
  int i, semid;
  int val = 4;
  semid = semget(42, 1, IPC_CREAT|0666);
  init(semid, val);
  for (i=1; i<10; i++) {
    P(semid);
    printf("Opération P\n");
  }
  return 0;
}
```

1. Interpréter le résultat de l'exécution de ce programme.

2. Même question avec la fonction principale suivante :

```
int main() {
  int i, semid;
  int val = 4;
  semid = semget(42, 1, IPC_CREAT|0666);
  init(semid, val);
  for(;;) {
     P(semid); printf("opération P\n");
     P(semid); printf("opération P\n");
     V(semid); printf("opération V\n");
  }
  return 0;
}
```

3. Même question avec la fonction principale suivante :

```
int main() {
  int semid;
  int val = 0;
  semid = semget(42, 1, IPC_CREAT|0666);
  init(semid,val);
  for (;;) {
    getchar();
    V(semid);
    printf ("Opération V\n");
  }
  return 0;
}
```

4. Même question avec la fonction principale suivante :

```
int main() {
  int pid,i,j,semid;
  int max = 100;
  int val = 2;
  semid = semget(42, 1, IPC_CREAT|0666);
  init(semid, val);
  pid = fork();
  switch(pid) {
    case -1: printf("Erreur dans fork()\n");
             return -1;
    case  0: printf("Je suis le processus fils.\n");
             printf("Je boucle dans le vide...\n");
             for(i=0;i<=max;i++)
               for(j=max;j>=0;j--);
             printf("Fin de la boucle.\n");
             P(semid);
             Z(semid);
             printf("Fin du fils.\n");
             break;
    default: printf("Je suis le processus père.\n");
             printf("J'attends un caractère...\n");
             getchar();
             printf("Fin de l'attente.\n");
             P(semid);
             Z(semid);
             printf("Fin du père.\n");
             break;
  }
  return 0;
}
```

Indication : faite une exécution du programme avec max = 100 et une autre avec max = 100000.

Exercice 1 – Producteur/Consommateur

Une réécriture simplifiée du problème Producteur/Consommateur étudié dans le cours est la suivante :

```
int main() {
  int pid;
  pid = fork();
  switch(pid) {
    case -1: printf("Erreur dans fork()\n"); return -1;
    case  0: printf("Processus producteur.\n");
             while(1) {
                printf("Production d'un objet.\n");
                printf("Objet placé dans tampon.\n");
             }
             break;
    default: printf("Processus consommateur.\n");
             while(1) {
             printf("Objet retiré du tampon.\n");
             printf("Consommation d'un objet.\n");
            }
             break;
  }
  return 0;
}
```

Modifier cette fonction principale afin de respecter les contraintes suivantes :

- Le producteur ne peut pas placer un objet dans le tampon alors qu'il est plein. La taille du tampon est finie (N=10).
- Le consommateur ne peut pas retirer un objet du tampon alors qu'il est vide.
- Le producteur et le consommateur ne doivent pas utiliser le tampon en même temps.

Afficher les valeurs des différents sémaphores utilisés et étudier les scénarios suivants :

1. Tampon initialement vide.
2. Tampon initialement plein.
3. Ralentir le processus producteur.
4. Ralentir le processus consommateur.

Indications :

- Pour ralentir un processus nous pouvons utiliser la fonction **`sleep(int secondes)`**.
- La valeur d'un sémaphore peut être obtenue comme valeur de retour de la fonction **semctl** avec l'opération **GETVAL**. Syntaxe :

```
int semctl(int idSEM, int numSem, GETVAL);
```

Exercice 2

Soit la fonction principale suivante :

```
int main() {
  int pid;
  pid = fork();
  switch(pid) {
    case -1: printf("Erreur dans fork()\n"); return -1;
    case  0: printf("Traitement A.\n");
             printf("Traitement C.\n");
             break;
    default: printf("Traitement B.\n");
             printf("Traitement D.\n");
             break;
  }
  return 0;
}
```

Modifier le code de la fonction principale pour respecter chacun des scénarios suivants :

1. Le traitement B se fait après la fin du traitement C.
2. Le traitement A se fait après la fin du traitement D.
3. Les traitements sont faits dans l'ordre alphabétique.
4. Les traitements sont faits dans l'ordre alphabétique inverse.