TD VI – Librairies de sémaphores

1. Introduction

Dans ce TD, nous allons étudier les sémaphores et voir comment ils protègent les sections critiques de programmes s'exécutant en même temps. Nous allons d'abord devoir créer une librairie avec les fonctions de base des sémaphores car elles ne sont pas implémentées. Nous allons ensuite tester l'utilisation des sémaphores dans le cas de 2 programmes concourant. Enfin, nous traiterons un problème de producteur-consommateur en utilisant la librairie créée.

2. Création d'une bibliothèque

On initialise une variable globale « semid » à -1.

2.1. Initialisation d'un sémaphore

Pour créer un groupe de sémaphore, on vérifie déjà que « semid » ne soit pas positif (auquel cas le groupe de sémaphores a déjà été créé car la primitive « semget » renvoie un entier positif). Si « semid » vaut -1, on crée le groupe de sémaphores avec « semget » et si la valeur de retour de la fonction est positive, la création est un succès. On initialise donc les sémaphores à 0 grâce à la primitive « semctl » avec l'option SETALL. Si la valeur de retour est -1, il y a eu une erreur et on affiche un message d'information.

2.2. Destruction de sémaphore

Cette fonction vise à détruire un groupe de sémaphores. Si « semid » est à -1, le groupe n'existe pas (init_semaphore() jamais appelée ou erreur lors de son appel). Sinon, on détruit le groupe grâce à la primitive « semctl » et à l'option IPC_RMID.

2.3. Attribution d'une valeur à un sémaphore

Cette fonction vise à attribuer une valeur à un sémaphore. On vérifie que le groupe de sémaphore existe bien et on attribue une valeur au champ « val » d'une union semun « value ». On attribue ensuite la valeur au sémaphore avec la primitive « semctl » en lui donnant le groupe de sémaphore, le numéro de sémaphore, l'option SETVAL, et la valeur value.val. On vérifie la valeur de retour afin de s'assurer qu'il n'y a pas eu d'erreur.

2.4. Fonction P

Cette fonction est l'une des principales dans la librairie de sémaphore, elle sert à décrémenter la valeur d'un sémaphore. On commence comme toujours par vérifier que le groupe existe, puis on boucle de manière infinie si la valeur du sémaphore est nulle (La valeur étant récupérée grâce à semctl(semid, sem, GETVAL)). Si le test est passé, la valeur du sémaphore n'est pas nulle et on va donc décrémenter sa valeur. Pour cela, on initialise une *struct sembuf op* où l'on va préciser le numéro de sémaphore (sem_num), l'opération (-1 pour effectuer un P) et un flag (0 ici). On effectue l'opération grâce à la primitive « semop » avec en argument le groupe de sémaphores et la structure sembuff. On vérifie qu'il n'y a pas eu d'erreur en vérifiant la valeur de retour de « semop » (-1 si erreur)

2.5. Fonction V

L'opération V est l'autre fonction principale dans la librairie de sémaphore, elle sert à incrémenter la valeur d'un sémaphore. On commence comme toujours par vérifier que le groupe existe, puis on va donc incrémenter sa valeur. Pour cela, on initialise une *struct sembuf op* où l'on va préciser le numéro de sémaphore (sem_num), l'opération (1 pour effectuer un V) et un flag (0 ici). On effectue l'opération grâce à la primitive « semop » avec en argument le groupe de sémaphores et la structure sembuff. On vérifie qu'il n'y a pas eu d'erreur en vérifiant la valeur de retour de « semop » (-1 si erreur)

3. Test

On vérifie que notre librairie fonctionne correctement avec le petit programme de test. Nous avons rajouté un affichage dans la librairie pour suivre l'évolution des valeurs des sémaphores. On a bien :

- Valeur du sémaphore n°2 qui vaut 1 ;
- Puis le P le met à 0 ;
- V le remet à 1
- Sémaphore détruit

On vérifie bien que « detruire_semaphore() » fonctionne en faisant un ipcs pendant l'exécution du programme : On voit bien notre groupe apparaitre dans la liste des sémaphores . Une fois le programme terminé, le groupe n'apparait plus dans la liste : il a bien été supprimé.

4. Section critique sans exclusion mutuelle

Dans cette partie, nous allons créer un programme qui va instancier un processus fils. On crée un segment de mémoire partagée entre le processus père et le processus fils qui vont partager un entier E. Chacun des deux processus va :

1. Affecter E à une variable A, entière, locale au processus.
2. Attendre entre 20 et 100 ms.
3. Incrémenter A.
4. Affecter la variable locale A à la variable "partagée" E.
5. Attendre entre 20 et 100 ms.
6. Affichage dans le processus père de la valeur de E.

Ceci est fait 100 fois pour chaque processus.

Nous rappelons brièvement la méthode pour créer un segment de mémoire partagée :

```
int shmid ;
int* ptr;
// On crée le segment en utilisant IPC_PRIVATE car le segment de mémoire est partagé entre
//un processus père et un processus fils
shmid = shmget(IPC_PRIVATE, 100, IPC_CREAT | 0600 );
//on  attache ce segment à l'espace virtuel du processus
ptr = shmat(shmid, 0, 0);
```

Nous avons implémenté une fonction qui retourne un nombre aléatoire entre 2 bornes a et b :

```
int rand_a_b(int a, int b){
        return rand()%(b-a) +a;
}
```

Nous prenons bien soin de détacher le segment de mémoire partagée, et de le supprimer à la fin du programme (shmdt() et shmctl () avec l'option IPC_RMID).

Nous constatons que la valeur finale de E n'est pas égale à 200 alors que c'est le comportement voulu. Ceci est dû au fait que l'entrelacement des instructions des deux processus provoque l'erreur de mise à jour de la variable commune. La partie où l'on modifie la variable commune E est une section critique qu'il faut à tout prix contrôler si l'on veut obtenir une valeur de 200. Il faut donc utiliser les sémaphores pour protéger l'exécution de la section critique.

5. Section critique avec sémaphore

Dans cette partie, nous allons mettre en place un système d'exclusion mutuelle pour palier au problème rencontré dans le programme précédent. On reprend exactement le même programme que précédemment, mais on crée cette fois un groupe de sémaphores. On initialise le sémaphore S à 1 pour effectuer l'exclusion mutuelle. Ensuite, on « entoure » la section critique de chaque processus, c'est-à-dire le contenu de la boucle, par un P(S) avant le code de la section critique, et un V(S) après la dernière ligne de code de cette section critique. On obtient ainsi le comportement voulu, à savoir obtenir une valeur pour E de 200.

Grâce à ce système, les deux parties du code qui modifient la variable partagée s'excluent. En effet, lorsqu'on entre dans une section critique, c'est que le sémaphore est à 1 et donc que l'autre partie du code n'est pas en train d'être exécutée. Lorsqu'on est dans une des deux sections critiques la valeur du sémaphore et à 0 et on a donc la certitude que si la CPU donne la main à l'autre processus, le P() initial précédant la partie critique va bloquer le processus et ne va donc pas rentré dans la section critique. On garantit ainsi que la variable partagée est mise à jour de manière bien séparée.

Enfin, on prend bien soin de supprimer le segment de mémoire partagée ET le groupe de sémaphore à la fin du programme (shmctl(shmid, IPC_RMID, 0); et detruire_semaphore() ;).

6. Problème de Producteur/Consommateur

7. Conclusion