

Matière : PRS (Programmation Système)

Année : LA1

Année scolaire : 2019-2020

Enseignant : Isabelle Le Glaz

TP n°3 - Sémaphores

Ce troisième TP aborde l'utilisation des sémaphores pour résoudre les problèmes de concurrence.

Ce TP fera l'objet d'un compte-rendu contenant :

- Les réponses aux diverses questions présentes dans cet énoncé,
- Les codes sources commentés des programmes réalisés,
- Les jeux d'essais obtenus,
- Une conclusion sur le travail réalisé, les difficultés rencontrées, les connaissances acquises ou restant à acquérir, ...

Il sera tenu compte de la qualité de vos pratiques de production de code :

- Lisibilité du code : emploi de commentaires, indentation, constantes symboliques
- Automatisation : Fichiers makefile avec macros ...
- Robustesse : Vérification des valeurs de retours des appels systèmes

Rappels

En général, une section dite « critique » est une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsque le processus (ou un thread) entre dans la section critique, il modifie la condition pour les autres processus/threads.

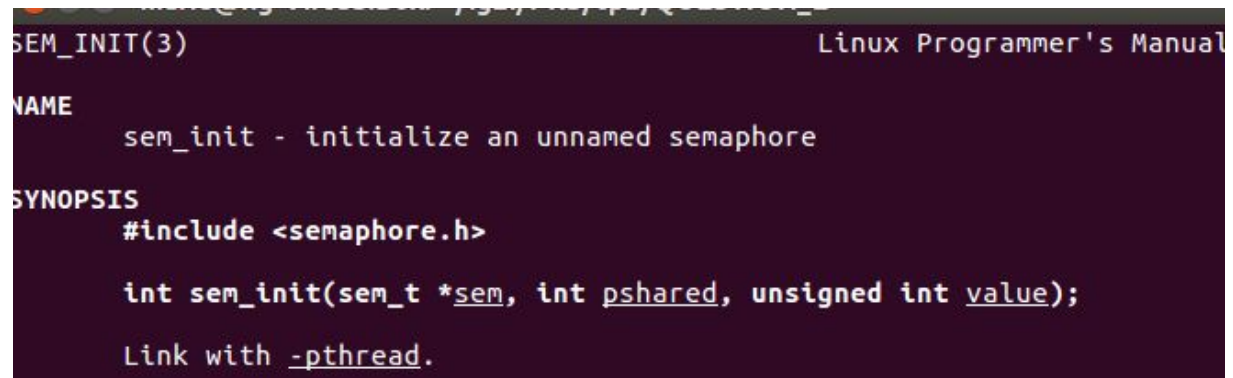
Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de n threads : Avant de rentrer dans la section critique, un thread doit vérifier qu'au plus $n-1$ threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie lui-même la condition sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique.

La difficulté est qu'on ne peut pas utiliser une simple variable comme compteur. En effet, si le test sur le nombre de thread et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, si l'on joue de malchance, un autre thread pourrait tester la condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critique.

Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est-à-dire qu'aucun autre processus/thread ne peut rien exécuter entre le test et la modification. C'est une opération atomique appelée *Test and Set Lock*.

Les sémaphores sont un type de variable nommé **sem_t** et un ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. **On entre dans une section critique si la valeur du compteur est strictement positive.**

Pour utiliser un sémaphore, on doit le déclarer et l'initialiser à une certaine valeur avec la fonction **sem_init**.



```
SEM_INIT(3) Linux Programmer's Manual
NAME
    sem_init - initialize an unnamed semaphore
SYNOPSIS
    #include <semaphore.h>

    int sem_init(sem_t *sem, int pshared, unsigned int value);

    Link with -pthread.
```

Le premier argument est un passage par adresse du sémaphore, le deuxième argument indique si le sémaphore peut être partagé par plusieurs processus, ou seulement par les threads du processus appelant (pshared égale 0). Enfin, le troisième argument est la valeur initiale du sémaphore. Cette valeur initiale peut être à 0, c'est à dire que la section est bloquée au démarrage, des événements survenant dans la suite du programme lui permettant de se débloquer par la suite.

Après utilisation, il faut systématiquement libérer le sémaphore avec la fonction **sem_destroy**.

SEM_DESTROY(3)	Linux Programmer's Manual	SEM_DESTROY(3)
<p>NAME <i>top</i></p> <p> <i>sem_destroy</i> - destroy an unnamed semaphore</p>		
<p>SYNOPSIS <i>top</i></p> <pre>#include <semaphore.h> int sem_destroy(sem_t *sem);</pre> <p>Link with <i>-pthread</i>.</p>		

Les primitives de bases sur les sémaphores sont :

- `sem_wait` : Reste bloquée si le sémaphore est nul et sinon décrémente le compteur (opération atomique) ;
- `sem_post` : incrémente le compteur ;
- `sem_getvalue` : récupère la valeur du compteur dans une variable passée par adresse ;
- `sem_trywait` : teste si le sémaphore est non nul et décrémente le sémaphore, mais sans bloquer. Provoque une erreur en cas de valeur nulle du sémaphore. Il faut utiliser cette fonction avec précaution car elle est prompte à générer des bogues.

Les prototypes des fonctions `sem_wait`, `sem_post` et `sem_getvalue` sont :

`sem_wait (sem_t * semaphore)`

SEM_WAIT(2)	Linux Programmer's Manual	SEM_WAIT(2)
<p>NAME <i>top</i></p> <p> <i>sem_wait</i>, <i>sem_timedwait</i>, <i>sem_trywait</i> - lock a semaphore</p>		
<p>SYNOPSIS <i>top</i></p> <pre>#include <semaphore.h> int sem_wait(sem_t *sem); int sem_trywait(sem_t *sem); int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);</pre> <p>Link with <i>-pthread</i>.</p> <p>Features that Macro Requirements for glibc (see <i>features_test_macros(7)</i>):</p> <p><i>sem_timedwait()</i>: _POSIX_C_SOURCE >= 200111L</p>		

`sem_post (sem_t * semaphore)`

SEM_POST(3)	Linux Programmer's Manual	SEM_POST(3)
NAME	<p>sem_post – unlock a semaphore</p>	
SYNOPSIS	<pre> #include <semaphore.h> int sem_post(sem_t *sem); Link with -pthread.</pre>	

sem_getvalue (sem_t * semaphore , int * v a l e u r)

SEM_GETVALUE(3)	Linux Programmer's Manual	SEM_GETVALUE(3)
NAME	<p>sem_getvalue – get the value of a semaphore</p>	
SYNOPSIS	<pre> #include <semaphore.h> int sem_getvalue(sem_t *sem, int *sval); Link with -pthread.</pre>	

Question 1 :

Ecrire un programme qui réalise les actions suivantes :

- récupère un paramètre de lancement lui **donnant le nombre maximum de threads pouvant réaliser le traitement critique en même temps** (Cf + bas)
- lance 10 threads. Chaque thread exécute une même fonction.
- Attend la fin des 10 threads pour se terminer.

La fonction de thread réalise les actions suivantes :

- Affiche un message en début de traitement pour indiquer son N° de thread (de 1 à 10)
- Exécute un nombre aléatoire de fois (compris entre 1 et 5) le traitement critique suivant :
 - Affiche un message pour signaler le début d'un passage critique avec son n° de thread, son N° d'itération et le nombre d'itérations à exécuter.
 - Dort un nombre aléatoire de secondes compris entre 1 et 5 secondes.
 - Affiche un message pour signaler la fin d'un passage critique
- Affiche un message en sortie de thread pour indiquer que le thread se termine

Toutes ces opérations doivent être réalisées avec la technique des sémaphores anonymes.

Question 2 - Retour sur les threads mutex

2.1 L'exercice précédent ressemble très fortement à l'exercice du TP précédent concernant les threads mutex. Cependant, il existe une différence sur l'usage d'un sémaphore et d'un thread mutex. Laquelle ?

2.2 Si on devait qualifier un thread mutex de sémaphore, quel particularité aurait-il ?

2.3 Quelles sont les deux situations qui ne permettent pas d'utiliser un thread_mutex et nécessitent un sémaphore ?

Question 3 - Les sémaphores nommés

Rappel :

Les sémaphores nommés permettent de partager une ressource critique entre processus, et non seulement entre threads. Pour cela, on dispose du même procédé que pour les sémaphores, mais en exploitant le système de fichiers d'UNIX.

Pour gérer des sémaphores nommés, on dispose de trois fonctions supplémentaires

- sem_open : ouverture / init du sémaphore nommé
- sem_close : suppression du lien entre le processus et le sémaphore
- sem_unlink : décrémentation du nombre de processus attachés au sémaphore, et éventuellement destruction si le nombre est passé à zéro.

3.1 Réaliser une nouvelle version du programme de la question 1 en utilisant la notion de sémaphore nommé et en transformant les threads en processus. Afficher la valeur du sémaphore à chaque itération d'un processus fils.

3.2 Dans quel répertoire les sémaphores nommés se trouvent-ils ? Quelles sont les permissions sur votre sémaphore nommé ? Votre sémaphore peut-il être utilisé par un processus appartenant à un autre utilisateur ?

Question 4 - Le barbier

Un barbier dispose d'un seul fauteuil pour accueillir son client et le raser, et d'une salle d'attente comportant 8 sièges pour accueillir les clients.

Quand un client se présente dans le salon, il prend un fauteuil en salle d'attente. S'il n'y a pas plus de fauteuil de libre, il reste dehors.

Quand il n'y a personne en salle d'attente, le barbier occupe son propre fauteuil et fait une sieste.

Dès qu'un client se présente, il libère le siège, rase le client.

Procédons par étape !

Etape 1 Génération de clients.

Écrivez un programme **barbier_01** qui lance l'exécution d'un nouveau thread dès que l'utilisateur appuie sur la touche ENTER.

Chaque thread représente un client. Dans cette première étape, le traitement associé à ce thread est une boucle prenant un certain temps (un temps aléatoire de lecture d'un magazine ...) , puis une sortie du thread.

Etape 2 Gestion de la salle d'attente

La salle d'attente comporte 8 sièges.

Chaque client créé ne peut venir s'asseoir en salle d'attente que s'il reste au moins un siège de libre. Sinon il attend dehors.

Modifier votre programme en version **barbier_02** pour gérer cette attente dehors; afficher les informations nécessaires pour connaître le nombre de clients dehors à chaque nouvel événement (arrivée d'un client, sortie d'un client de la salle d'attente, ...).

Etape 3 Gestion du rasage

Créer une version **barbier_03** pour générer un thread représentant le barbier. Le barbier attend indéfiniment l'arrivée de clients. Si un client entre en salle d'attente, il dit bonjour, c'est qu'il faut se réveiller et se mettre au travail ! Dans ce cas, il invite un client à s'installer confortablement sur le siège, le rase, puis le libère poliment .

Par la même occasion, le client libère donc un siège en salle d'attente, et c'est tant mieux pour un des clients qui attend au froid dehors ...

Les clients en salle d'attente ne sont plus bloqués par leur lecture, c'est le barbier qui mène la danse : c'est lui qui leur permet de quitter la salle d'attente, et non la fin de leur lecture de magazine (un salon de rasage n'est pas une bibliothèque, tout de même).

Etape 4 : Un petit tableau d'affichage s'impose ...

Proposer une solution pour afficher une information claire et sûre aux clients : combien y-a-t-il de clients dehors, combien y-a-t-il de clients en salle d'attente ?

Ecrire le programme **barbier_04** qui permet d'ajouter cette nouvelle fonctionnalité.