



1 Fonctions utiles

Description extraite du site http://docs.mandraptor.org/files/Operating_systems/Linux/Man_fr/

#include <semaphore.h>

Semaphore.h est la librairie à inclure pour permettre l'utilisation des sémaphores dans votre programme.

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

"sem_init" initialise le sémaphore pointé par "sem". Le compteur associé au sémaphore est initialisé à "valeur". L'argument "pshared" indique si le sémaphore est local au processus courant (pshared initialisé à zéro) ou partagée entre plusieurs processus (pshared n'est pas nulle). LinuxThreads ne gère actuellement pas les sémaphores partagés entre plusieurs processus, donc "sem_init" renvoie toujours l'erreur "ENOSYS" si "pshared" n'est pas nulle.

int sem_wait(sem_t * sem);

"sem_wait" suspend le thread appelant jusqu'à ce que le sémaphore pointé par "sem" ait un compteur non nul. Alors, le compteur du sémaphore est atomiquement décrémenté.

int sem_post(sem_t * sem);

"sem_post" incrémente atomiquement le compteur du sémaphore pointé par "sem". Cette fonction ne bloque jamais et peut être utilisée de manière fiable dans un gestionnaire de signaux.

#include <pthread.h>

Semaphore.h est la librairie à inclure pour permettre l'utilisation des processus dans votre programme.

int pthread_create(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg);

"pthread_create" crée un nouveau thread s'exécutant concurremment avec le thread appelant. Le nouveau thread exécute la fonction "start_routine" en lui passant "arg" comme premier argument. Le nouveau thread s'achève soit explicitement en appelant "pthread_exit" ou implicitement lorsque la fonction "start_routine" s'achève. Ce dernier cas est équivalent à appeler "pthread_exit" avec la valeur renvoyée par "start_routine" comme code de sortie.

L'argument "attr" indique les attributs du nouveau thread. Voir "pthread_attr_init" pour une liste complète des attributs. L'argument "attr" peut être "NULL", auquel cas, les attributs par défaut sont utilisés: le thread créé est joignable (non détaché) et utilise la politique d'ordonnancement usuelle (pas temps-réel).

int pthread_join(pthread_t th, void **thread_return);

"pthread_join" suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par "th" achève son exécution, soit en appelant "pthread_exit" soit après avoir été annulé.

Si "thread_return" ne vaut pas "NULL", la valeur renvoyée par "th" y sera enregistrée. Cette valeur sera soit l'argument passé à "pthread_exit" (3), soit "PTHREAD_CANCELED" si le thread "th" a été annulé.

Le thread joint "th" doit être dans l'état joignable: il ne doit pas avoir été détaché par "pthread_detach" ou par l'attribut "PTHREAD_CREATE_DETACHED" lors de sa création par "pthread_create".

Quand l'exécution d'un thread joignable s'achève, ses ressources mémoires (descripteur de thread et pile) ne sont pas désallouées jusqu'à ce qu'un autre thread le joigne en utilisant pthread_join. Aussi, "pthread_join" doit être appelée une fois pour chaque thread joignable pour éviter des "fuites" de mémoire.

Au plus un seul thread peut attendre la mort d'un thread donné. Appeler "pthread_join" sur un thread "th" dont un autre thread attend déjà la fin renvoie une erreur.

Pour des informations complémentaires, utilisez la commande man.

2 Mise en avant des problèmes de concurrences

Le code source fourni (fichier tp1_compteur.c) permet à 4 processus d'incrémenter un compteur partagé. Afin de vérifier le bon fonctionnement du compteur, chaque processus possède un compteur privé dont lui seul à accès et qu'il incrémente également. A la fin du programme, la valeur du compteur partagé et la somme des compteurs privés sont affichées.

Compilez et exécutez le code source. Que constatez-vous ?

Modifiez le programme afin qu'il ait un comportement correct.

3 Le problème du dîner des philosophes

Considérons cinq philosophes, installés autour d'une table circulaire, et qui passent leurs temps à penser et à manger.

La table est mise avec *cinq couverts* qui sont disposés *entre chacun* des philosophes.

NB : le nombre des philosophes peut être quelconque, mais il doit être au moins égal à cinq pour garantir le bon fonctionnement du programme.

De temps en temps, un philosophe a faim et essaye de prendre les couverts qui sont immédiatement à cotés de lui (ceux qui sont entre lui et son voisin de gauche et de droite). Un *philosophe a besoin de deux couverts pour manger*, et ne peut évidemment pas prendre un couvert qui est dans la main d'un voisin.

Quand un philosophe affamé a ses deux couverts dans les mains en même temps, il mange sans libérer ses couverts. Dans le cas contraire, il doit attendre que ceux-ci deviennent libres.

Enfin, quand il a fini de manger, il repose ses deux couverts et commence à penser à nouveau.

Précisons que les *philosophes mangent et pensent* durant des *temps aléatoires différents*, leur *changements d'état*, se produisent donc de manière totalement *asynchrone*.

Le problème même du dîner des philosophes consiste donc à *réguler les changements d'état* de ces derniers suivant la disponibilité des couverts, qui dépend bien entendu de l'état des voisins du philosophe concerné.

Par exemple, dans notre hypothèse d'un dîner de cinq philosophes, *seulement deux philosophes peuvent manger à un instant donné* car les couverts ne sont pas suffisants.

Dans ce cas, trois philosophes n'ont la possibilité que de penser ou d'être en attente de vouloir manger.

NB : quelque soit le nombre de philosophes, on ne peut jamais avoir deux philosophes mangeant cote à cote, pour de "conflit de couverts".

Réaliser un programme permettant d'illustrer ce problème. Chaque philosophe est représenté par un processus. Le programme ne doit pas arriver à une situation d'interblocage et les chances de manger pour les philosophes doivent être équitables.