TD 1

Création destruction de threads

1.1 But de ce TP

Ce TD a pour but de vous initier à la création et à la destruction des threads. Il vous permettra de créer 2 threads et de synchroniser ou pas leur terminaison.

1.2 Création de thread

Objectif: *création de threads et vérification de l'exécution en parallèle, ps*
Vous créerez un thread autre que le thread principal (main). Le thread créé effectuera une boucle N fois qui affiche tous les 1 seconde l'expression "et mon courroux". Le thread principal fera lui aussi une boucle M fois qui affiche tous les 1 seconde le mot "coucou".

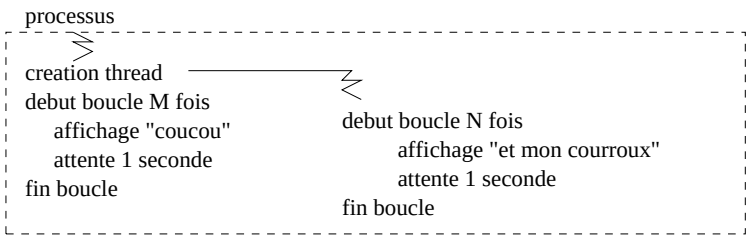

Figure 1.1: Exécution de 2 threads

1.2.1 paramètres

Demandez à l'utilisateur la valeur de N et M et passer la valeur de N en paramètre du thread

1.2.2 valeurs des paramètres

Faites varier les valeurs de M et N et commenter ce qu'il se passe.

- M égal à N
- M plus grand que N
- M plus petit que N

1.3 Terminaison de thread

Objectif: *test de différentes terminaisons possible pour un thread*

1. Vous utiliserez les fonctions **pthread_exit** pour terminer chaque thread
2. Dans le thread principal, vous attendrez la terminaison de l'autre thread avec la fonction **pthread_join**. Vous récupérerez dans le main la valeur fournie dans le pthread_exit du thread en utilisant pthread_join
3. Retirez la fonction de la question précédente et utilisez la fonction de demande d'arrêt d'un thread: **pthread_cancel**

Pour chaque question vous modifierez les valeurs de N et M et expliquerez ce qu'il s'est passé lors de l'exécution.

TD 2

Utilisation des mutex

2.1 But de ce TD

Ce TD a pour but de vous faire manipuler les mutex entre threads à l'intérieur d'un même processus.

2.2 Exemple de problème

Objectif: *passage d'un argument au thread et exemple d'un programme buggé*
Vous utiliserez le programme du TD1 permettant de créer 2 threads (faites une copie du programme). Vous modifierez ensuite la copie comme suit:

1. Vous déclarerez une variable **globale** "px" du type pointeur sur un réel que vous initialiserez à NULL.
2. La fonction exécuté par le thread prendra un paramètre permettant de récupérer une valeur de type entière (max). la fonction fera une boucle "max" fois et affichera à chaque fois la valeur pointée par px: $*px$.
3. le thread principal fera l'algorithme suivant:
   - déclaration d'un réel "x" initialisé à 1
   - affectation de "px" avec l'adresse de x
   - création du thread
   - boucle pour i de 0 à max
   - affectation de "px" à NULL
   - affichage de la valeur de i
   - affectation de "px" avec l'adresse de x
   - fin boucle

   Exécutez plusieurs fois ce programme. Que constatez vous ?

2.3 Mise en place du mutex

Objectif: *manipulation d'un mutex pour protéger des sections critiques*

Vous metterez en place un mutex afin de protéger les sections de codes posant un problème.

2.4 Joker

Supposez maintenant que vous avez 3 threads, et une autre variable du même genre que px que vous appelerez py. Le thread 1 manipule px, le thread 2 manipule py et le thread 3 manipule px et py en même temps. On voudrait:

- permettre au thread 1 et 2 de travailler en même temps puisqu'ils ne travaillent pas sur la même variable.
- assurer que quand le thread 3 travaille sur px et py les deux autres threads n'y ont pas accès
- ne pas avoir la situation où le thread 1 travaille sur px, le thread 3 voudrait travailler sur px et py mais doit attendre la fin du travail du thread 1, le thread 3 bloque le thread 2 qui aimerait aussi travailler sur py.

Proposez et implémentez une solution.

TD 3

Utilisation des variables de condition

3.1 But de ce TP

Ce TD a pour but de vous faire manipuler les variables condition entre threads à l'intérieur d'un même processus.

3.2 Problème à résoudre

Objectif: *gestion de la demande de ressources, manipulation des variables conditions*

3.2.1 Description

Votre programme dispose de "M" ressources. Le programme est multi-threadé. Un thread peut demander à avoir "N" ressources avec $N <= M$. Une fois les ressources utilisées, le thread les libère. A chaque fois que des ressources sont allouées à un thread, elles ne sont plus disponibles pour les autres threads. Il peut donc arriver qu'un thread demande "N" ressources alors que le nombre de ressources disponible est inférieur à N. Dans ce cas, le thread est bloqué jusqu'à tant que sa demande puisse être satisfaite. Quand des ressources seront libérées, la requête du thread en attente doit être réévaluée.

Exemple de fonctionnement (le systeme dispose de 4 ressources):

- thread 1: demande de 3 ressources => requete satisfaite
- thread 2: demande de 2 ressources => thread bloqué
- thread 3: demande de 1 ressources => requete satisfaite
- thread 3: libère 1 ressources => demande bloquée réévaluée
- thread 2: on ne peut toujours pas satisfaire la demande du thread 2
- thread 1: libère les 3 ressources => demande bloquée réévaluée
- thread 2: le thread 2 prend 2 ressources
- thread 2: libère 2 ressources

3.2.2 Implémentation

Proposez une solution au problème précédent en utilisant un mutex et une variable conditionnelle.

3.3 Améliorations

Proposez une modification afin de ne pas “signaler” quand cela n’est pas nécessaire.

3.4 Joker

Reprenez la dernière question du TD2 et proposez une solution avec une variable conditionnelle.

TD 4

Manipulation des fonctions d'exécution unique et des clés

4.1 But de ce TP

Ce TD a pour but de vous faire manipuler les fonctions à ne réaliser qu'une fois et les clés permettant d'associer des variables uniques aux threads.

4.2 ExÈcution unique

Objectif: *gestion d'une fonction appelée par tous les threads mais exécutée une seule fois*

Ecrivez un programme manipulant 3 threads (2 threads créés + le thread principal). Chaque thread appel la fonction : "void fonction_init()" et réalise N fois une boucle comprenant un "printf" et un "sleep".

La fonction "fonction_init" fait un printf pour indiquer qu'elle a été exécutée.

Vous devez garantir que même si les 3 threads appellent la fonction "fonction_init", cette dernière ne sera exécutée qu'une fois. Il vous faut pour cela manipuler: **pthread_once**.

4.3 Manipulation d'une clé

4.3.1 Programme erroné

Ecrivez une fonction "int incrementation()" qui déclare en local un entier statique qui est incrémenté d'un à chaque appel. La fonction retourne la nouvelle valeur de l'entier statique.

Affichez le résultat de l'appel de la fonction "incrementation" dans la boucle de chaque thread. Que constatez vous?

En fait, on voudrait que: quand un même thread appel deux fois de suite (quel que soit le temps écoulé entre les 2 appels) la fonction "incrementation" la valeur retournée ne soit différente que de 1 et ne pas avoir une valeur d'incrémentation quelconque.

4.3.2 Programme corrigée

Utilisez une clé pour obtenir le comportement désiré vis à vis de la fonction "incrementation". Pour cela vous devrez:

- déclarez une clé en variable globale
- initialisé la clé dans la fonction: "fonction_init" (**pthread_key_create**)
- allouée une variable spécifique au début de chaque thread que vous attacherez à la clé (malloc + **pthread_setspecific**). Afin de mieux comprendre ce qu'il se passe intialiser cette variable avec le numéro du thread (**pthread_self**)
- modifiez la fonction "incrementation" afin que cette dernière récupère la variable accrochée à ce thread (**pthread_getspecific**) et la modifie en la renvoyant.

TD 5

Manipulation des points d'arrêt et des signaux

5.1 Les points d'arrêt

Objectif: *manipulation du cancel et des mutex sans point d'arrêt dans un premier temps, puis mise en place de la gestion des points d'arrêt*

5.1.1 bug

vous utiliserez un thread en plus du thread principal. Le thread réalisera les actions suivantes (max prendra une valeur de 6):

- "lock" d'un mutex"
- affichage "j'ai le mutex"
- boucle max fois
- affiche son tid
- sleep(1)
- fin boucle
- "unlock" du mutex

le "main" réalise les actions suivantes:

- boucle 2 fois
- création du thread
- boucle max/2 fois
- affichage du tid
- sleep(1)
- fin boucle
- cancel du thread

- sleep(1)
- fin boucle

Que constatez vous ? Pourquoi ?

5.1.2 Correction du programme

Utilisez les fonctions **pthread_cleanup_push** et **pthread_cleanup_pop** pour mettre en place un handler corrigeant le problème précédent.

5.2 Les signaux

Objectif: *manipulation des signaux dans les threads*

Créer un thread masquant tous les signaux sauf SUGUSR1. Le thread principal masquera tous les signaux sauf SIGUSR2. La réception de SIGUSR1 fait afficher quelquechose a l'écran. La réception de SIGUSR2 arrête le programme.

Que constatez vous sous LINUX?

TD 6

Joker 1: Calcul de π en parallèle

6.1 But de ce TD

Ce TD a pour but de calculer π en parallèle en utilisant le multi-threading. Il vous faudra aussi gérer la fin des threads afin de calculer π.

6.2 Principe

Vous disposez d'une cible ronde de diamètre D insérée dans un carré de coté D. Un archer très habile tire F flèches qui touchent toutes la cible et se répartissent de manière uniforme dans le carré.
On désire estimer la probabilité P_f pour une flèche de tomber dans la cible ronde.
Comme le tirage des flèches est uniforme dans le carré, la probabilité est égale au rapport des surfaces entre le cercle et le carré : $P_f = (\pi * (D/2)^2)/D^2 = \pi/4$

Cette probabilité correspond aussi au rapport du nombre de flèches dans le cercle (F_c) divisé par le nombre de flèches total : $P_f = F_c/F$. on obtient donc $\pi = 4 * F_c/F$.

6.3 Réalisation

On va créer T threads. Chaque thread tire aléatoirement F/T flèches. On tire aléatoirement les coordonnées x et y d'une flèche $x \in [-D/2, D/2]$ et $y \in [-D/2, D/2]$. Une flèche est dans le cercle si $x^2 + y^2 < (D/2)^2$. Afin d'estimer les performances de votre programme, celui ci fournira son temps d'exécution. Vous fournirez en paramètre de l'exécutable : le nombre de thread à lancer, le nombre de flèches total et un nom de fichier où vous écrirez après chaque exécution :

1. temps
2. nombre de threads
3. nombre de flèches

Analyser l'evolution du temps en fonction du nombre de flèches et de threads.

Pour faire cela vous aurez besoin de :

1. rand pour tirer aléatoirement des valeurs
2. gettimeofday pour prendre le temps

TD 7

Joker 2 : Client/serveur et multi-threading

L'objectif est de créer un système de boîtes aux lettres avec des clients déposant des messages à destination d'un autre client. Les messages sont stockés sur un processus serveur. Ce dernier est multi-threadé afin de traiter en parallèle plusieurs clients. Quand un client envoie un message ce dernier est stocké dans la boîte aux lettres du destinataire. Quand un client lit un message, il est détruit de sa boîte aux lettres.

Un message est constitué d'un texte et des informations de communication : émetteur et récepteur.

7.1 Système de boîtes aux lettres

Créer une structure pour les messages et les boîtes aux lettres ainsi que les fonctions nécessaires à leur manipulation. Tenez compte de l'accès concurrent par plusieurs threads aux boîtes aux lettres. Testez vos fonctions.

7.2 Clients serveur

Créer un squelette de client et de serveur multi-threadé basé sur TCP permettant d'envoyer une chaîne de caractères de taille fixe et de recevoir un message d'acquittement coté client.

7.3 Outil de messagerie

Insérer le système de boîtes aux lettres dans le client serveur.