TP C avancé n^o 8 Licence Informatique L3 - 2021-2022
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux UFR SFA - Université de Poitiers

Multi-threads

Table des matières

1 Résumé et cadre de travail

Dans ce TP il s'agit de manipuler les processus légers (appelés également threads).

De tels processus ont besoin (généralement) de se synchroniser via des sémaphores ou des mutex. Outre les sémaphores IPC, les threads possèdent leur propre mécanisme de mutex.

Rappel d'une directive importante : pour tout appel système, (ouverture de fichier, lancement d'un thread, ...) vous devez tester la valeur de retour pour détecter toute anomalie ; on ne présuppose jamais qu'un appel système fonctionne. Un *assert* suffira amplement.

2 Principe de fonctionnement des threads

Point de cours

Un thread est un processus dit léger qui s'exécute au sein d'un processus lourd ; et plusieurs threads peuvent s'exécuter concurremment dans un processus lourd.
La mémoire du processus lourd est partagée par tous les threads.

Techniquement, lorsqu'un thread est lancé, il exécute le code d'une fonction (qui s'exécute donc en concurrence avec les autres threads). Lorsque la fonction se termine, le thread se termine par la même occasion.

Le thread principal est celui qui exécute le programme principal (fonction *main*). Si celui-ci se termine, alors tous les autres threads sont terminés de manière brutale (au contraire des processus lourds avec *fork*). Le thread principal doit donc attendre la fin de tous les autres threads avant de se terminer lui-même (et on retrouve ici un comportement analogue à celui des *fork*, si ce n'est que le devoir d'attente des threads est absolu).

Nous utiliserons obligatoirement les fonctions suivantes :

- *pthread_create* : pour créer et lancer un thread
- *pthread_join* : pour attendre la fin d'un thread fils
- *pthread_exit* : pour terminer (proprement) un thread (un thread se termine également proprement lorsqu'il arrive à la fin de sa fonction support)
- *pthread_self* : renvoie l'identifiant du thread courant (analogue à *getpid*)

Voici d'autres fonctions que nous n'utiliserons pas :

- *pthread_cancel* : arrêter un autre thread
- *pthread_detach* : rendre un thread indépendant de son créateur
- *pthread_equal* : compare les identifiants de deux threads
- *pthread_attr_<something>* : pour paramétrer le comportement d'un thread (nous utiliserons uniquement le paramétrage par défaut)

3 Création de threads

Point de cours

Un thread exécute une fonction dont le prototype est obligatoirement :

```
void * nom_fonction(void *arg);
```

Les arguments de la fonction sont fournis via un pointeur sans type que nous étudierons ultérieurement.

La fonction support d'un thread retourne un pointeur sans type (NULL par convention si on n'attend aucun retour).

La fonction *pthread_create* prend 4 arguments :
- un pointeur sur une variable de type *pthread_t* qui sera remplie avec l'identifiant du thread créé (pour être utilisé par *pthread_join* par exemple).
- le paramétrage du thread : *NULL* pour nous.
- le nom de la fonction support du thread.
- les arguments de la fonction support du thread : *NULL* dans un premier temps pour nous.

La fonction renvoie 0 en cas de succès.

la fonction *pthread_join* attend la fin d'un thread fils et prend 2 arguments :
- une variable de type *pthread_t* qui identifie la thread à attendre.
- un paramètre permettant de récupérer le code de retour du thread : *NULL* si ce code ne nous intéresse pas.

La fonction renvoie 0 en cas de succès.

Écrivez un programme qui lance plusieurs threads ; chaque thread affiche son *pid*.

Le squelette *exo3.c* est fourni.

Les différents threads affichent-ils un *pid* différent ? La réponse est non, pourquoi ?

4 Création de threads avec paramètres

Point de cours

Lorsqu'on veut envoyer des paramètres à la fonction thread (ce qui est souvent le cas), il faut les passer via le 4^{me} paramètre de la fonction *pthread_create*.

La fonction thread n'accepte qu'un paramètre qui est un pointeur sur ce qu'on veut (*void **). S'il n'y a qu'un seul paramètre, il suffit de passer un pointeur dessus. S'il y a plusieurs paramètres à transmettre, il faudra passer par une structure intermédiaire et transmettre un pointeur sur cette structure.

La fonction thread reçoit un *void **; elle devra caster ce pointeur dans le type attendu.
Attention il n'y a aucune vérification du compilateur (c'est impossible), il faut donc être vigilant pour caster dans le bon type.

Il ne faut lire le libellé d'une question qu'une fois les précédentes programmées (il y a des éléments de réponse d'une question dans les questions suivantes).

Les squelettes des programmes sont fournis.

a) Il s'agit de passer à la fonction thread son numéro d'ordre (1 pour le premier lancement, 2 pour le deuxième et ainsi de suite). Pour cela on déclare un compteur dans la fonction *main* et on passe un pointeur dessus lors de la création des thread). Le thread affiche le numéro qu'elle reçoit (après une attente d'une seconde).
Il se passe un comportement a priori surprenant ; expliquez-le.

b) Pour pallier le problème précédent, une solution est que dans le *main* il y ait un tableau de numéros : chaque thread recevra un pointeur sur une case qui lui est propre.
Programmez le nouveau code.
Lancez plusieurs fois le programme pour voir si les affichages sont toujours dans le même ordre.

c) Reprenez le programme précédent, mais faites volontairement une erreur de type : on envoie toujours un entier au thread mais ce dernier le caste en *float*. Regardez le résultat.

5 Création de threads avec paramètres (bis repetita)

Le but est le suivant :

- la fonction *main* déclare un tableau d'entiers, tableau qui a autant de cases qu'il y a de threads.
- chaque thread a la charge d'une case du tableau.
- chaque thread reçoit 2 entiers, il doit calculer la somme des entiers de l'intervalle et stocker le résultat dans la case du tableau qui lui est attribuée.
- le *main* affiche le tableau une fois les calculs finis.

Il y a donc plusieurs paramètres à passer à un thread :

- un pointeur sur la case qu'il doit remplir
- le début de l'intervalle
- la fin de l'intervalle

Il faut donc passer par une structure intermédiaire.

Le squelette est fourni.

6 Mutex posix

Point de cours

Qui dit mémoire partagée dit synchronisation, par exemple :

- pour qu'un processus attente un autre
- pour qu'une zone mémoire ne puisse pas être accédée par plus d'un thread à la fois (exclusion mutuelle)

Il est possible d'utiliser les sémaphores IPC. Mais ces derniers sont destinés à la communication entre les processus lourds et nécessitent beaucoup de ressources système.

Les threads ont une version particulièrement restreinte des sémaphores : les mutex.
Voici les caractéristiques d'un mutex :

- il ne fonctionne qu'au sein d'un seul processus lourd, pour ses threads
- il est moins gourmand en ressources qu'un sémaphore IPC
- il ne peut prendre que deux valeurs : 0 ou 1. On parle plutôt d'états "locked" (verrouillé) et "unlocked" (déverrouillé).
- seul le thread qui a verrouillé le mutex a le droit de le déverrouiller
- un thread voulant verrouiller un mutex déjà verrouillé est bloqué jusqu'à son déverrouillage (fonctionnement normal d'un sémaphore)
- déverrouiller un mutex déjà déverrouillé est une erreur
- bref il sert uniquement aux sections critiques

Un mutex est donc beaucoup moins puissant qu'un sémaphore, mais pour les sections critiques il est préférable de l'utiliser (vu son faible impact sur les ressources).

Nous utiliserons les fonctions suivantes :

- initialisation : par affectation directe lors de la déclaration de la variable de type mutex :

```
pthread_mutex_t monMutex = PTHREAD_MUTEX_INITIALIZER;
```

- *pthread_mutex_lock* : verrouillage (et blocage si déjà verrouillé)
- *pthread_mutex_trylock* : comme *lock* mais ne se bloque pas si le mutex est déjà verrouillé, il renvoie un code d'erreur à la place
- *pthread_mutex_unlock* : déverrouillage
- *pthread_mutex_destroy* : libère les ressources d'un mutex (qui ne doit donc plus être utilisé)

Nous n'utiliserons pas les fonctions du type *pthread_mutexattr_<something>*.

Les squelettes des programmes sont fournis.

a) Écrivez un programme qui lance plusieurs threads, chacun faisant une grande quantité d'incrémentations (+1) sur une variable partagée.
L'adresse de la variable partagée (qui est déclarée en local dans le *main*) est passée en paramètre au thread. Il en est de même pour le nombre d'incrémentations à effectuer.
Il n'y a pas de section critique pour protéger l'accès à la variable.
Vérifiez si le résultat est cohérent. A priori ce n'est pas la cas, pourquoi ?

b) On reprend le programme précédent, mais chaque incrémentation est protégée dans une section critique.
Note : le mutex est déclaré en variable globale.
Vérifiez que le résultat est maintenant correct.
Comparez le temps d'exécution avec celui de la première version. Pourquoi est-ce si long ?

c) Il s'agit du même programme, mais cette fois-ci toutes les incrémentations sont faite sur une variable locale du thread, et la mise à jour de la variable partagée est faite une seule fois (et en section critique) à la fin du thread.
Comparer le nouveau temps d'exécution avec l'ancien.

d) Il s'agit du même programme, mais cette fois-ci le mutex est déclaré en local dans la fonction *main* et est passé en paramètre aux threads.

7 Sémaphore anonyme (facultatif)

Point de cours

Il existe également un type sémaphore dédié plus particulièrement aux threads : *sem_t* .
Il est aussi possible de partager de tels sémaphores entre des processus lourds mais nous n'utiliserons pas cette possibilité.

Le comportement est très proche de celui des sémaphores IPC, mais d'une puissance moindre cependant :

- la variation de la valeur d'un sémaphore ne se fait que par incrément de ± 1
- la possibilité de se bloquer jusqu'à ce que la valeur du sémaphore atteigne 0 (zéro) n'existe pas.

Nous utiliserons les fonctions suivantes :

- *sem_init* : création et initialisation d'un sémaphore
- *sem_wait* : tentative de décrémenter (-1) un sémaphore
- *sem_post* : incrémentation (+1) d'un sémaphore
- *sem_destroy* : libère les ressources d'un sémaphore

Nous n'utiliserons pas les fonctions suivantes :
sem_open sem_close sem_unlink sem_trywait sem_timedwait

La fonction *sem_init* prend 3 arguments :

- un pointeur sur une variable de type *sem_t* qui sera remplie avec l'identifiant du sémaphore créé.
- un paramètre indiquant si le sémaphore est partagé entre des processus lourds : *0* pour nous, indiquant que le sémaphore n'est partagé que par les threads du processus.
- un entier indiquant la valeur initiale.

La fonction renvoie 0 en cas de succès.

Le thread principal lance plusieurs threads fils. Un thread fils commence par un temps d'initialisation, puis continue par un temps de travail.
Le thread principal, une fois qu'il a lancé les fils, doit attendre, avant de continuer, que tous les fils aient fini leurs initialisations.

Le squelette est fourni : il ne reste qu'à gérer le sémaphore et l'attente du thread principal