Stéphane Lavirotte, Erick Gallesio, Hélène Collavizza
Polytech'Nice – Sophia
SI3
2019-2020

TD n° 4
Processus et Thread

Pour réaliser ce TD, nous vous fournissons une partie du code pour vous permettre de vous concentrer sur les parties importantes. Vous pouvez récupérer l'archive à l'adresse suivante :

```
https://lms.univ-cotedazur.fr/mod/resource/view.php?id=82517
```

1 Premiers threads, PID et Thread ID

Un thread (« fil d'exécution » ou « tâche ») est une portion de code (fonction) qui se déroule en parallèle au thread principal (aussi appelé `main`).

Exercice n°1:

Le programme `tres_simple` illustre la création d'un thread. Après avoir constaté que ce code est très simple, commencez par exécuter le code tel qu'il vous est fourni. Puis, supprimez l'appel à `pthread_join`. Quelle(s) différence(s) constatez-vous ? Que pouvez vous conclure sur la durée de vie d'un thread ?

Remettez l'appel à `pthread_join` et ajoutez un `exit(0)` après le `printf("Hello from the thread")`. Que constatez-vous ? Enlevez le `exit(0)` dans le thread et mettez le avant le pthread_join. Que pouvez-vous conclure sur `exit` et `pthread_exit` ?

Exercice n°2:

Ecrire un programme `threads` qui reçoit en argument deux entiers `n1` et `n2` puis crée deux threads. Le programme commencera par afficher son numéro de PID. Puis on créera deux threads, chacune exécutant une boucle, parcourue 5 fois, qui affiche sur la sortie standard son identifiant de processus (`getpid()`) et son identifiant de thread Posix (`pthread_self()`) puis se met en attente `n1` secondes pour la première, `n2` secondes pour la seconde, avant de reboucler. Le thread renvoie son statut d'exécution : `0` si l'argument utilisé pour le `sleep` (i.e. `n1` ou `n2`) est positif, `1` si l'argument est négatif. Quant au père, il doit attendre la fin des deux threads qu'il a créées (`pthread_join`) et afficher leur statut d'exécution.

Faites varier `n1` et `n2` pour vérifier que votre programme se comporte correctement.

Exercice n°3:

Modifiez le programme précédent pour simplement ajouter l'affichage du numéro de thread géré par le noyau Linux. Pour cela, vous utiliserez le code suivant :

```
#include <sys/syscall.h>
#include <sys/types.h>

printf("Linux Thread ID: %ld\n", syscall(SYS_gettid));
```

Bien entendu, ce code n'est pas portable (il ne fait pas partie de l'API Posix) et est dépendant du noyau Linux. En fait ce code fait appel à l'appel système du noyau Linux pour récupérer le numéro de thread (tel que représenté dans celui-ci). Vous pourrez constater que vous obtenez un numéro de thread ID qui est différent du numéro obtenu avec `pthread_self()`, qui est l'appel standard Posix.

> **Remarque :** Ceci s'explique par le fait, comme nous avons pu le voir en cours, que l'implémentation des threads sous Linux est un patch au système, qui historiquement a été réalisé par deux méthodes différentes : en espace utilisateur, en espace noyau). Contrairement à d'autres systèmes (comme le noyau Windows NT), cette notion de thread a été ajoutée après le développement du noyau (débutée en 1991) et en particulier l'abstraction de processus au cœur du système d'exploitation. L'ajout de la normalisation des threads en Posix datant de 1995 et celle-ci étant postérieure au démarrage du développement de Linux, l'ajout a été réalisé de manière plus ou moins propre.

TD n° 4
Processus et Thread

2 Différences Processus et Threads

Les threads sont donc des unités plus légères (unité d'exécution) que les processus (unité de ressources), comme nous avons pu l'expliquer en cours. Nous allons mettre en évidence assez simplement cette légèreté des threads par rapport aux processus en créant une grande quantité de ces deux types d'entités et constater que le temps pour les créer est bien inférieur dans le cas des threads que des processus.

> **Attention** : Pour mesurer quelque chose d'un peu plus pertinent, 1 seul cœur de calcul doit être utilisé par votre processus. Pour cela, vous lancerez celui-ci grâce à la commande :
> `taskset -c 0 ./mon_prog.exe`

Exercice n°4:

Vous disposez du programme `multiple_fork`. Lors de son exécution, celui-ci prend en paramètre le nombre de processus à créer. Pouvez-vous créer autant de processus que vous le souhaitez ?

Exercice n°5:

Sur le modèle du programme `multiple_fork`, vous réaliserez un programme `multiple_threads` qui créera des threads à la place de processus.

Comparez le temps de création d'un processus par rapport à un thread (vous tenterez de créer `5.000` processus ou threads). Quelle différence constatez-vous ?

> **Remarque** : Dans le cadre de ce TD, le facteur de temps entre la création d'un processus et d'un thread est assez faible par rapport aux chiffres qui ont été annoncés en cours (d'un facteur `5` à un facteur `100`). Ceci est dû au fait que nous ne mesurons que le temps de création du thread ou du processus.

Cette création consiste en la duplication des informations nécessaire à la gestion des éléments qui composent ces entités. Or dans le cadre d'un système d'exploitation, les structures de données ne sont pas obligatoirement dupliquées. En effet la technique de copie sur écriture (ou en anglais « *copy on write* ») est une stratégie d'optimisation largement utilisée dans les systèmes d'exploitation et en programmation. L'idée fondamentale est que si de multiples appelants demandent l'accès à des ressources initialement identiques, il est possible de leur donner des pointeurs vers la même ressource. Cette approche n'est possible que jusqu'à ce qu'un appelant modifie « sa copie » de la ressource. Dis dans d'autres termes, tant que tout le monde ne fait que des lectures dans les données, elles peuvent être partagées et il n'y a pas d'obligation de dupliquer les ressources (cette duplication engendrant une copie de blocs mémoire, qui peut être très coûteuse en temps, suivant le volume de données à dupliquer). Quand un des appelant veut modifier une des données, donc faire une écriture, alors une copie privée de celle-ci est réalisée (et uniquement pour le bloc de données qui est modifié). Cela permet de faire de très grosses économies du temps nécessaire à la création et d'optimiser au cours du temps en ne dupliquant que ce qui est nécessaire.

Dans notre exemple, comme on ne fait que la création des entités processus et thread, seule la duplication des morceaux qui vont différer entre les deux entités est réalisée. La structure pour gérer un thread étant plus légère que la structure pour gérer un processus, on constate cette différence dans le temps mis pour réaliser cette duplication.

3 Observation des Processus et Threads

Exercice n°6:

Pour constater la différence entre processus et thread, vous pourrez utiliser tour à tour les commandes suivantes : `ps aux` et la commande `htop`. Nous vous conseillons de lancer ces commandes dans une fenêtre terminal dédiée (pas dans le terminal de l'environnement Visual Studio Code).

TD n° 4
Processus et Thread

Pour la commande `htop`, il vous faudra configurer l'affichage du nombre de thread d'un processus (F2 setup / columns / ajouter la colonne NLWP pour afficher le nombre de thread d'un processus / F10) puis configurer le filtrage des processus/thread par leur nom (F4 / nom du processus ou thread).

En lançant tour à tour les programmes `multiple_fork` et `multiple_threads`, vous pourrez noter la différence entre la création de processus et de threads vue de l'espace utilisateur. Vous constaterez la gestion des identifiants de threads sous Linux. Comme nous l'avons constaté avec l'exercice 2, que sous Linux, les threads disposent d'un numéro spécifique comme les processus (même si la fonction `getpid()` dans le processus donne le même numéro pour chacune des thread d'un processus). Linux implémente les threads Posix comme des processus légers[1] et à ce titre, ont un numéro de `pid`. Mais bien entendu, les données ne sont pas dupliquées, seul le fil d'exécution l'est.

4 Programmes (presque) justes

Considérez le programme C défini dans le fichier `juste_presque.c`. Essayez d'exécuter ce programme tel qu'il vous est fourni. Son fonctionnement semble correct à première vue.

Exercice n°7:

Enlevez l'appel à la fonction `sleep` de la fonction `main`. Le programme vous semble-t-il toujours s'exécuter suivant la même logique que ce que vous avez constaté précédemment (il faudra peut-être plusieurs exécutions avant que vous constatiez un problème) ? Est-ce qu'il vous semble raisonnable qu'un programme soit dépendant d'une instruction d'attente ?

Expliquez ce qui est à l'origine de l'erreur et justifiez votre réponse. Corrigez là dans un fichier `juste.c`. Dans un cas aussi simple que cet exemple, il est possible de s'en sortir simplement.

Exercice n°8:

Mais il peut être nécessaire de partager une structure de données entre les différents threads. C'est le cas par exemple dans l'implémentation du jeu des allumettes avec des threads fourni dans le programme `jeu.c`. Lancez le programme `jeu` dans un terminal autre que celui de Visual Studio Code et constatez que le programme ne fonctionne pas correctement à toutes les parties (un joueur peu prendre une allumette alors qu'il n'en reste plus sur le plateau de jeu). Enlevez l'appel à la fonction `sleep` dans le thread des joueurs. Quel nouveau dysfonctionnement constatez-vous ?

Remettez l'appel à la fonction `sleep`. Vous pouvez aussi constater que votre programme une fois lancé consomme `100`% du CPU ! Utilisez la commande `htop` vue précédemment pour constater cela. C'est tout de même beaucoup pour un programme qui ne fait qu'un tirage aléatoire du nombre d'allumettes à retirer et attend `1` seconde. D'où vient cette consommation anormale des ressources de calcul ?

> **Remarque** : Nous ne faisons là que mettre en évidence des problématiques autour de l'utilisation des threads. Nous n'irons pas plus loin dans l'étude de ces problématiques dans le cadre de ce cours. Vous pourrez voir dans le cours « Programmation Concurrente » en 4ème année comment régler proprement ce type de problèmes (sans utiliser des délais bien sûr !).

Exercice n°9:

Afin de pouvoir vous coucher ce soir en n'étant pas trop frustré, vous pourrez consulter une version presque correcte du programme dans `jeu_correct.c`. Vous trouverez ce fichier sur Moodle.

[1] http://opensourceforu.com/2011/08/light-weight-processes-dissecting-linux-threads/

TD n° 4
Processus et Thread

5 Processus et Threads sous Win32

S'il est possible de créer des processus et des threads sous Windows via l'interface Posix Thread grâce à la bibliothèque pthreads-win32[2], Windows possède sa propre interface de programmation pour créer ces entités. La différence fondamentale entre Windows et Linux pour les threads et les processus est :

- Sous Linux, l'unité basique pour l'exécution est le processus, la notion de thread a été ajoutée a posteriori.
- Sous Windows, l'unité basique d'exécution est le thread, un processus étant un conteneur incluant au moins un thread.

Pour aller plus loin sur les différences entre l'approche Win32 et l'approche Posix, vous pourrez consulter la ressource suivante :

https://fr.slideshare.net/abufayez/pthreads-vs-win32-threads

Exercice n°10:

Faites un programme qui réalise l'équivalent d'un « `fork` + `exec` » avec l'API Win32 sous Windows.

Exercice n°11:

Faites un programme qui crée un thread supplémentaire au thread « main » à l'aide de l'API Win32 sous Windows.

Exercice n°12:

Pour faire une synthèse que le sujet, maintenant que vous avez pu tester les deux méthodes pour le lancement d'un nouveau processus, que préférez-vous ? L'approche Posix avec « `fork` + `exec` » ou bien l'approche Win32 avec « `CreateProcess` » ? Justifiez votre réponse.

Et pour les threads, quelle approche en Posix et sa mise en œuvre sous Linux ou Win32 vous semble la plus adaptée et pourquoi ? Justifiez votre réponse.

[2] https://sourceware.org/pthreads-win32/