Nom 1 : CHAWAF Alia **Nom 2 :** MISSOUM Inès

TP Synchronisation des processus, tubes et signaux

Module F.A.R. IG3 _ Feuille à rendre en fin de TP

0 - Echauffement

Chargez les fichiers du TP depuis Moodle sur votre machine dans un dossier spécifique que vous rangerez au bon endroit, puis ouvrez un terminal positionné dans le dossier.

I - C'est du lourd

Ouvrez `cpt-lourd.c` dans un éditeur de code (sublime text par exemple, s'il n'est pas sur votre machine, téléchargez la version 2 -- gratuite pour votre S.E.).

Répondez aux questions suivantes SANS COMPILER ET EXECUTER le programme (ne pas tricher, merci ;)

1) combien de processus sont créés par ce programme

500 processus sont créés : un à chaque itération de la boucle for

2) Quel est le but de ce programme ?

Le but de ce programme est de calculer le nombre de fils créés à l'aide de la variable : somme.

3) Que va-t-il se passer lors de l'exécution, c'est-à-dire que va produire l'exécution de ce programme à l'écran ?

L'exécution du programme va afficher pour chaque processus fils (500 lignes au total) : « processus fils i : somme = S » avec i le numéro de l'itération au moment où le processus est créé et S la somme incrémenté à chaque ligne.
A la fin de ces 500 lignes s'affiche : « processus père : somme = 500 »

Ca y est, vous avez enfin le droit de compiler le programme et de l'exécuter.

4) Etes-vous surpris du résultat de l'exécution du programme ? pourquoi ? expliquez ce qui ne marche pas bien :

On est surpris du résultat car pour tous les processus fils (500 premières lignes) la somme affichée vaut 1 (elle n'est donc pas incrémenté au fur et à mesure) et pour le processus père (dernière ligne) la somme vaut 0.

Ceci s'explique par le fait que la première ligne du main (somme=0) est exécuté par tous les processus (père et fils). Dans la partie de code des processus fils, la somme est incrémenté et dans celle du père elle n'est pas changé. Ce qui explique les résultats observés.

5) Pourquoi c'est toujours la ligne du processus père qui s'affiche en dernier ?

C'est toujours la ligne du processus père qui s'affiche en dernier car le wait fait que sa partie de code ne s'exécute que lorsque tous ses processus fils ont fini leurs éxécution.

II - Se sentir plus léger

Ouvrez maintenant dans l'éditeur de code `cpt-leger.c` et répondez aux questions suivantes SANS LE COMPILER ET L'EXECUTER (hop hop hop, on ne triche pas non plus) :

1) Quelle(s) différence(s) avec le programme précédent ?

La majeur différence est que ce programme utilise des thread (processus légers) contrairement au premier ou on travailler avec des processus lourds.

2) Prévoyez-vous que son exécution va donner ou pas un résultat différent du programme précédent ? pourquoi ?

On pense que le résultat sera différent du programme précédent, à savoir que ça va marcher car on n'a pas la ligne : somme=0; dans le main.

Allez-y, vous avez maintenant le droit de compiler et exécuter ce programme.

3) Que constatez-vous ? expliquez (sauf si vous aviez deviné juste ci-dessus).

On constate que le programme fonctionne correctement c'est-à-dire comme expliqué à la question 4 de la question 1.

4) Rappelez quel est l'avantage des threads sur les processus classiques (créés par un appel système `fork()`) :

Les threads ont l'avantage de partager une zone de données contrairement aux processus classiques (lourds).

5) Mais au fait vous aurait-on menti : vérifions si les processus créés par `fork()` ont bien un numéro de processus différent les uns des autres. Comment afficher le numéro de processus (pid) de chaque processus (et du père aussi) ?

Pour afficher le numéro de processus (pid) on utilise pour les fils: getpid() et pour les pères : getppid()

6) Modifiez `cpt-lourd.c` pour que chaque processus (père et fils) affichent leur pid puis indiquez ce que vous constatez :

Chaque processus fils a un pid différent mais ils ont tous le même père (même pid père).

7) Faites la même modification pour `cpt-leger` et indiquez ce que vous constatez :

On observe le même « pid fils » à chaque fois et le même « pid père » également.

8) Dans les deux cas, si vous avez la chance que la commande `pstree` soit installée, lancez-la depuis un autre terminal pendant que chacun des programmes ci-dessus s'exécute, et vérifiez les filiations éventuelles (sinon avec la commande `ps -l`). Ce que vous observez est comme attendu ou pas (pourquoi) ?

Pour le programme lourd on observe une arborescence avec un père et 500 fils tandis qu'avec le programme léger on observe un père et un fils. Ce qui est cohérent avec les pid observé.

9) Vérifions maintenant la charge pour le système d'un programme comparé à l'autre. Après avoir consulté `man time`, expliquez à quoi sert cette commande en deux lignes (max) :

Elle permet d'obtenir le manuel qui contient les commandes générales de la bibliothèque time

10) Lancez maintenant les deux programmes précédés de la commande time. Quels temps obtenus pour `cpt-leger` en moyenne détaillez user et système :

On obtient en moyenne : User = 0,1sec et sys = 0,35sec

11) Idem pour `cpt-lourd` :

On obtient en moyenne : User = 0,005 sec et sys = 0,9 sec

12) Conclusion : quel programme est le plus coûteux pour la machine à faire tourner ? Comment expliquer ça ?

On voit nettement que le programme « lourd » est plus coûteux que en temps pour la machine c'est parce que la création des threads est plus rapide.

III Les bons comptes font les bons amis

Dans `cpt-leger` passez le nombre de processus créés de 500 à 10 000. Lancez plusieurs exécutions du programme, que constatez-vous ?

Lorsqu'on lance le programme, parfois la somme finale est bonne (=10 000) et d'autres fois elle ne l'est pas (<10 000).

1) Complétez les phrases suivantes avec les mots proposés en dessous :

Le problème constaté vient du fait que le programme contient une ..... Section critique ..... et que l'accès à celle-ci n'est pas protégé. Pour que le programme fonctionne il faut garantir une ..... Exclusion mutuelle ..... Ceci peut être réalisé grâce au mécanisme des ..... Semaphores ..... qui permet de plus d'éviter ..... Une attente active ..... contrairement à d'autres solutions logicielles.

Mots à utiliser (vous pouvez les barrer au fur et à mesure pour un total retour en enfance) : exclusion mutuelle - tube - inclusion indépendante - attente active - pâte à crêpe - sémaphore - c'est moi l' plus fort - processus - signaux - attention - section christique - section critique - attente passive - attente lascive.

2) Pour remédier au problème ci-dessus, on va maintenant regarder le programme `mutex-thread.c` . Ouvrez ce programme et remarquez comment en faisant attendre les processus fils un temps non prévisible à un endroit critique, on peut utiliser bien moins de threads pour provoquer le soucis repéré précédemment : compilez et exécutez le programme pour vérifier.

Devinette : que fait l'instruction pthread_mutex_init (&mutex, NULL); située dans le main ? (aide : consultez la diapo 73 du cours 1 et ses copines autour)

Cette instruction crée un mutex initialisé avec la valeur 1 (sémaphore binaire).

3) A quelle valeur le sémaphore est initialisé après cette instruction ? Est-il possible d'utiliser la même instruction différemment pour donner une valeur plus importante au sémaphore ? si oui comment ?

Le sémaphore est initialisé avec la valeur 1, il n'est donc pas possible d'utiliser la même instruction différemment pour donner une valeur plus importante au sémaphore .

4) Dans quel cas voudrait-on donner une valeur plus importante à un sémaphore ?

On peut donner une valeur plus importe à un sémaphore si on veut autoriser l'accès à la section critique à plusieurs threads à la fois.

5) Placez les instructions suivantes au bon endroit dans le code (à nouveau voir diapos du cours, si nécessaire) :

```
pthread_mutex_lock (&mutex);

pthread_mutex_unlock (&mutex);
```

Puis compiler et exécuter le code pour voir si le soucis de compteur est réglé ou pas (vérifiez sur plusieurs exécutions). Vérifiez déjà sur plusieurs exécutions du programme que le soucis est réglé, puis pour vérifier d'une autre façon, commentez la partie qui fait dormir les processus et passez à 10 000 processus. Est-ce ok au bout de 10 essais à 10 000 processus ?

On a crée une section critique dans la fonction fils qui modifie la variable globale somme et on a vérifié sur plusieurs exécutions, le problème est réglé. La partie qui fait dormir les processus sert à créer des problèmes de synchronisation.

IV - Signaux et tube

Revenez sur le cours numéro 2 et écrivez les programmes nécessaires pour les exercices suivant :

- exercices p13

- exercices p20

V - Le tout à la fois

Ecrivez un programme qui prend deux arguments en entrée : un nom de dossier de départ, et un nom de dossier d'arrivée. Le programme doit copier tous les fichiers (non récursivement) du dossier de départ dans le dossier d'arrivée (qu'on suppose créé à la main dans un premier temps), en faisant appel à un thread différent pour chaque fichier à copier. Si le fichier ne peut pas être copié, alors le thread enverra un signal indiquant une situation à problème en écrivant son numéro et le fichier à problème dans un fichier partagé du nom de `errors.log` . Bien sûr l'accès à ce fichier ne doit être réalisé que par un seul thread à la fois (vous utiliserez un (des ?) sémaphore(s) pour cela).

Pour tester le programme, vous choisirez un dossier de départ où vous aurez enlevé les droits d'accès à certains fichiers (`chmod`).