INF2610 – Noyau du système d'exploitation

Laboratoire 1 – Hiver 2018

Fils d'exécution

Nom, prénom, matricule des membres de l'équipe	Bouis Constantin 1783438 Nicolet Olivier 1726277 Timothee Laborde 1782257
Note finale sur 20	

Mise en situation

Comment démarrer un nouveau programme ?

Quelle est la différence entre un processus et un fil d'exécution ?

Quel est l'effet sur la mémoire accessible par un processus ?

Ce laboratoire a pour but de répondre à ces questions en étudiant les mécanismes d'exécution d'une tâche sous Linux.

Directives

- L'activité se fait en équipe de deux. Inscrivez les noms dans le cadre ci-haut.
- Répondez directement dans le questionnaire *odt* avec *Libre Office*.
- La correction se fera directement dans le document électronique.
- Remettez sur Moodle le questionnaire rempli et le code (archive *tar.gz* produite par `make dist`). **Vérifiez** le contenu de votre archive **avant** de la rendre.
- Une seule personne de l'équipe doit effectuer la remise sur Moodle.
- Utilisez les machines du laboratoire pour obtenir vos résultats.
- 10% de la note finale peut être enlevée pour la mauvaise qualité de la langue.
- Les deux colonnes de droite sont réservées pour la correction.

Après avoir décompressé les sources, exécutez le script `fixperms.sh` pour corriger les permissions :

```
tar xzvf inf2610-lab1-2.1.tar.gz
cd inf2610-lab1-2.1/
```

```
sh fixperms.sh
```

N'oubliez pas de vous référer aux commandes fournies dans le sujet du laboratoire 0 en cas de doutes sur la compilation et l'exécution des tests.

1 Opérations bancaires [9 pts]

Le programme `banque` simule des transactions sur un compte accédé par plusieurs guichets automatiques (ATM). Quatre versions sont disponibles : **série, processus (fork), fil d'exécution (pthreads) et fil d'exécution utilisateur (pth)**. La version série est complétée et vous devez compléter les trois autres versions. Pour les versions pthreads, fork et pth, il faut lancer les différents fils d'exécution l'un après l'autre sans attendre la fin du précédent. Bien que cela soit tout à fait possible et acceptable en pratique, nous supposons ici que le fil d'exécution principal ne fasse aucune opération directe sur les ATM et que seuls les fils créés à partir du fil principal effectuent de telles opérations. Observez les résultats obtenus.

Une fois le programme fonctionnel, tracez l'exécution de chaque version du programme avec LTTng. Le script `banque-all.sh` est disponible pour réaliser cette opération. La commande est la suivante :

```
lttng-simple -k -c -s -- ./banque-all.sh
```

Le résultat sera disponible dans le répertoire `banque-all.sh-k`. **Importez cette trace dans Trace Compass**, puis retrouvez l'exécution du script pour comparer les différentes exécutions.

Résumez vos observations dans le tableau 1 pour chaque type d'exécution. Pour compter le nombre de PID et TID créés pendant l'exécution, tenez compte de tous les processus et tous les fils d'exécution créés, **y compris le processus initial**.

Version du programme	Nombre de PID créé(s)	Nombre de TID créé(s)	Exécution simultanée?	Résultat correct? (égal au cas série)
Série (serial)	1	1	Non	Oui
Processus (fork)	4	4	Oui	Non
Fil d'exécution (pthread)	1	4	Oui	Non
Fil d'exécution utilisateur (pth)	1	1	Non	Oui

Tableau 1: Création des tâches

Le programme `perf` permet de récupérer des statistiques sur un grand nombre d'événements de bas niveau déclenchés par un programme, tels que les défauts de pages, les défauts de cache et les changements de contexte. La commande `perf list` donne une liste complète des événements disponibles.

Utilisez le script

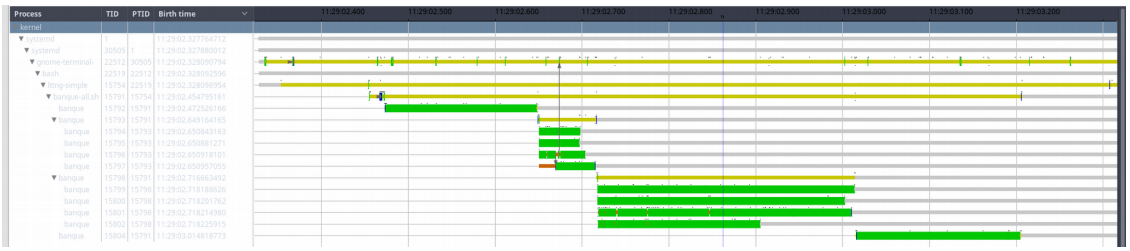
```
./banque-perf.sh
```

afin de récupérer des statistiques qui vous permettront de mieux comprendre la performance de chaque version du programme. Le fichier résultant, `perf.csv`, peut être ouvert avec *LibreOffice Calc*.

1.1 Implémentation du programme. Une partie du barème porte sur la qualité du code lui-même. / 3 pt

1.2 Résultats du tableau 1. (-0.5pt par erreur ou omission) / 2 pt

1.3 Tracez l'application à l'aide de LTTng (voir TP0). Justifiez vos réponses du tableau 1 à l'aide de captures d'écran dans TraceCompass et expliquez. / 1 pt



On voit clairement les 4 manières dexecuter le programme.

Série = 1 bande verte longue : toutes les actions sont faites a la suite sur un même et long fil.

Processus = 4 bandes vertes courtes : 4 processus en simultanée exécutent les instructions.

Pthread = 4 bandes vertes longues : 4 fils dexecutions exécutent les instructions .

Fil utilisateur : 1 seule bande verte qui exécute les instructions

1.4 Dans quel appel système se produit la création d'un nouveau processus (`fork`) et celle d'un nouveau fil d'exécution (`pthread`) ? Quelle est la différence entre les deux ? **Copiez ici les appels système impliqués tels qu'observés en exécutant le programme avec `strace` et expliquez brièvement**. / 1 pt

Exemple appel système :

clone(child_stack=NULL,flags=CLONE_CHILD_CLEARTID
CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f74c7611e10) = 8487

Astuce : utilisez l'option -o de `strace`, l'outil `grep` pour la recherche, et `man` pour

la documentation de l'appel système.

Fork : On cree des nouveaux processus Grace aux appels a clone().

PThread : On cree des nouveaux fils grace aux appels a clone(). Cependant ces derniers ont tous le meme PID.

1.5 Expliquez le comportement à l'égard du **résultat du calcul** pour les versions d'exécution `fork` et `pthread`. Une condition de concurrence (accès simultané en lecture et écriture à une donnée partagée) se produit-elle ? Si oui, pour quelle version d'exécution ? Pourquoi pas pour l'autre ? / 2 pt

Explication :

Fork copie les variables du processus principal. Modifier ces variables ne change pas la variable du processus pere, donc aucune operation nest effectuee sur le processus principal.

Les threads partagent les variables entre eux, ainsi lors de lecriture sur la variable on ne peut pas savoir lequel des threads va écrire en dernier sur cette variable. Mais la derniere ecriture sera celle prise en compte; cest pourquoi on obtient des resultats differents a chaque execution¸du programme.

Resumons :

- fork : pas acces simultane car lors de lecriture une copie se fait

- Thread : Acces simultane en ecriture

2 Descripteurs de fichiers [3 pts]

Pour les questions suivantes, exécutez le programme `descripteurs` du dossier nommé « descripteurs » et observez la sortie.

2.1 Remarquez que le texte du premier `printf` est affiché avant celui du `write`, alors que le texte du deuxième `printf` est affiché après celui du `write`. Remarquez aussi que le texte du deuxième `printf` est affiché par les deux processus bien que cette instruction soit exécutée par le processus parent uniquement. Expliquez la cause de ces comportements. / 1 pt

2.2 Utilisez la commande suivante pour faire le traçage du programme : / 1 pt

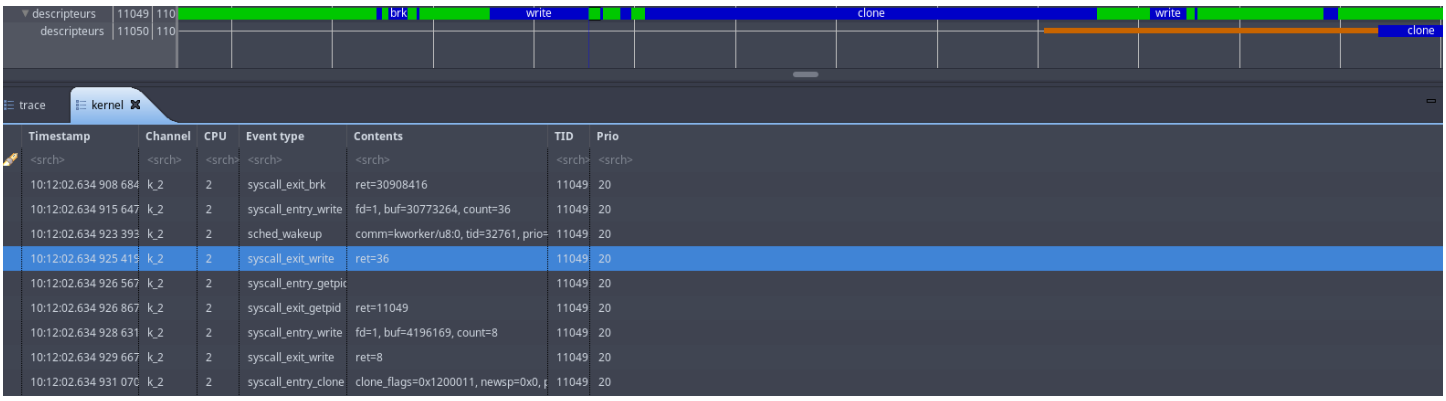
```
Lttng-simple -k -s -e sched -c -- ./descripteurs
```

Quel est l'appel système invoqué par `printf` pour afficher la sortie sur le terminal? Quand est-ce que `printf` invoque cet appel système ? Illustrez votre réponse à l'aide d'une ou de plusieurs captures d'écran dans TraceCompass.

2.3 Modifiez le code de `descripteurs.c` de façon à ce que le texte du second `printf` soit seulement affiché par le processus parent (évidemment sans déplacer le `printf`). Expliquez pourquoi vos modifications règlent le problème. / 1 pt

2.1 Le premier *printf* comporte un \n, le buffer du flux utilisé est donc vidé. À contrario le second *printf* n'en comporte pas, il est donc « bufferisé » et se retrouve dans les deux processus (le parent et le fils). Il est alors affiché avant le troisième *printf*.

2.2



10:12:02.634 908 684	k_2	2	syscall_exit_brk	ret=30908416	11049	20
10:12:02.634 915 647	k_2	2	syscall_entry_write	fd=1, buf=30773264, count=36	11049	20
10:12:02.634 923 393	k_2	2	sched_wakeup	comm=kworker/u8:0, tid=32761, prio=	11049	20
10:12:02.634 925 419	k_2	2	syscall_exit_write	ret=36	11049	20
10:12:02.634 926 567	k_2	2	syscall_entry_getpic		11049	20
10:12:02.634 926 867	k_2	2	syscall_exit_getpid	ret=11049	11049	20
10:12:02.634 928 631	k_2	2	syscall_entry_write	fd=1, buf=4196169, count=8	11049	20
10:12:02.634 929 667	k_2	2	syscall_exit_write	ret=8	11049	20
10:12:02.634 931 070	k_2	2	syscall_entry_clone	clone_flags=0x1200011, newsp=0x0, p	11049	20

On peut observer sur la première capture et sur le détail de la 3éme capture que l'appel système « write » est invoqué plusieurs fois suivit de l'appel getpid, enfin un dernier appel write est fait avant l'invocation du clone. On peut en déduire que les printf correspondent aux appels de write suivis des appels de getpid tandis que les write dans le fichier .c ne font appel qu'aux write seul.

2.3 Pour que le printf ne soit affiché que dans le processus parent il suffit de vider le buffer. Pour cela nous pouvons ajouter un \n ou un fflush(stdout). Le second ne modifie pas l'affichage prévu, tandis que le premier ajout un retour à la ligne ce qui pourrait être plus clair.

3 Exécution en chaîne [8 pts]

Votre ami vient demander votre aide pour réussir un défi : exécuter trois programmes différents, l'un à la suite de l'autre, sans la création de nouveau processus pour chacun. Les programmes sont `foo`, `baz` et `bar`. Chaque programme utilise `whoami()` pour fournir des informations sur l'exécution, dont le PID et une variable globale `rank`, qui est incrémentée dans chaque programme. Le lancement de la chaîne d'exécution se fait depuis le programme `chaine`. Le programme actuellement fourni n'exécute que `foo`.

Apportez les modifications nécessaires pour que les trois programmes s'exécutent séquentiellement. Le programme doit accepter un argument pour spécifier le nombre de cycle « `baz foo bar` » à exécuter (Ex : `./chaine 2` exécute `baz, foo, bar, baz, foo, bar` et se termine). Ensuite, répondez aux questions suivantes :

3.1 Implémentation du programme. Le numéro de processus est toujours le même, les programmes sont bien exécutés, l'argument du nombre de cycle fonctionne / 4 pt

correctement. Une partie du barème porte sur la qualité du code.

3.2 Dans la situation de départ, pourquoi seul le programme `foo` s'exécute? / 2 pt

3.3 La variable `rank` est incrémentée dans chaque programme, alors pourquoi la valeur de `rank` affichée par `whoami()` est-elle toujours la même? / 2 pt

3.2 Au départ, seul foo est exécuté car execlp() écrase le reste du code. Ainsi chaine est écrasée par le code de foo.

3.3 De la même manière, execlp() écrase le code. Ainsi, même si rank est incrémenté, rank sera écrasé puis réinitialise . C'est pour cela que whoami() renvoi toujours la même valeur.

4 Correction

Points des sections 1 à 2	0
Pénalité de retard (-5pts par jour de retard)	0
Qualité de la langue (-0,25% par faute, max -2pt)	0
Points bonus (le cas échéant)	0
Note finale sur 20	0