INF2610 – Noyau du système d'exploitation

Laboratoire 2 – Hiver 2018

Communication inter-processus

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

Mise en situation

Au précédent laboratoire, nous avons vu qu'un processus ne peut accéder à la mémoire d'un autre processus directement. Nous avons également vu qu'il est nécessaire de disposer d'un moyen de contrôle d'accès concurrent à une ressource, comme une variable partagée. C'est la fonction des **mécanismes de communication inter-processus** (Inter-Process Communication, ou IPC), dont il est question dans ce laboratoire.

Directives

- L'activité se fait en équipe de deux. Inscrivez les noms 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 produite par *make dist*).
- 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.

Après avoir décompressé les sources, exécutez le script *fixperms.sh* pour corriger les permissions puis n'oubliez pas de configurer votre projet :

```
tar xzvf inf2610-lab2-2.0.tar.gz
cd inf2610-lab2-2.0/
sh fixperms.sh
./configure
```

1 Classification des verrous [10 pts]

Vous travaillez sur une librairie dont la fonction est de **calculer des statistiques sur des nombres**

entiers. Plusieurs fils d'exécution pourront ajouter simultanément des valeurs, il est donc nécessaire de protéger la **section critique** par un verrou.

La question que l'on peut se poser est de savoir lequel choisir entre un *mutex*, un *sémaphore* ou un *spinlock* ? Quel est leur principe de fonctionnement ? Cet atelier vous permettra d'en faire l'expérience et d'observer leurs **interactions avec le système d'exploitation**.

Implémentation

Complétez le code source du programme `multilock` de manière à protéger la section critique d'accumulation des statistiques. Les fichiers à compléter sont les suivants :

- **mutex.c** : utilisation d'un seul mutex *PThread* comme simple verrou partagé.

- **semrelay.c** : utilisation d'un sémaphore `sem_t` **par fil d'exécution**. Lorsqu'un fil a terminé sa section interne, il signale le fil suivant et attend que tous les autres fils se soient exécutés. Ce cas particulier de sémaphore est appelé sémaphore chaîné. Dans ce qui suit, lorsque l'on vous pose des questions sur les semrelay, concentrez vos explications sur les sémaphores chaînés étudiés ici plutôt que sur les sémaphores génériques.

- **spinlock.c** : utilisation de `minispinlock` (voir le fichier minispinlock.asm).

Dans chaque fichier se trouvent 3 fonctions :

- **Initialisation** (ex : `*_init()`) : allouer le verrou et l'initialiser.

- **Finalisation** (ex : `*_destroy()` : libérer la mémoire allouée lors de l'initialisation.

- **Fonction principale** (ex : `*_main()` : calcul des statistiques. Insérez le verrouillage et le déverrouillage dans la boucle extérieure.

L'option `--lib` du programme `multilock` sélectionne la version à exécuter (voir l'aide pour la liste des versions). Pour exécuter toutes les versions, utiliser la commande `--check`. Pour afficher les résultats du calcul et celui attendu, passer l'option `--verbose`. Par défaut, 4 fils d'exécutions sont lancés, ce qui est suffisant pour l'expérience, mais l'option `--thread` permet de lancer plus de fils d'exécution.

Le code de la section critique de l'expérience est de longueur variable. Le paramètre `inner` augmente le nombre d'opérations effectuées dans la section critique, tandis que le paramètre `outer` sert à répéter plusieurs fois la prise et la libération du verrou. Dans la réalité, il est préférable de réduire au minimum le nombre d'instructions dans une section critique. Voici un exemple du lancement du programme.

```
./multilock --lib serial --verbose
label             N             sum             mean
actual        200000   4999900000.000         24999.500
expected      200000   4999900000.000         24999.500
```

```
PASS serial
```

Si la section critique est bien protégée, alors le programme retournera « PASS ». Il renverra « FAIL » dans le cas contraire.

Traçage de l'exécution

Une fois le code complété, tracez l'ensemble des tests avec la commande suivante :

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

Le script exécute deux **granularités de section critique** avec 4 fils d'exécution :

1. Une section critique très courte répétée un grand nombre de fois;
2. Une section critique très longue répétée un petit nombre de fois.

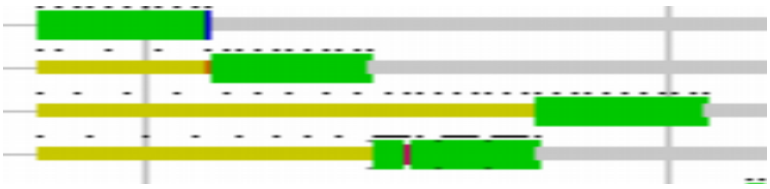
Importez la trace multilock-all.sh-k dans TraceCompass, puis complétez le tableau 1.

Attention! Pour *certaines lignes* du tableau, il vous est demandé de justifier vos réponses à l'aide de captures d'écran. Vous insérerez les captures dans ce document dans le tableau 2 ci-dessous, en les numérotant et en vous référant à leur numéro pour justifier vos réponses dans le tableau 1 quand c'est nécessaire. Vous limiterez vos captures d'écran à la partie intéressante de l'écran seulement.

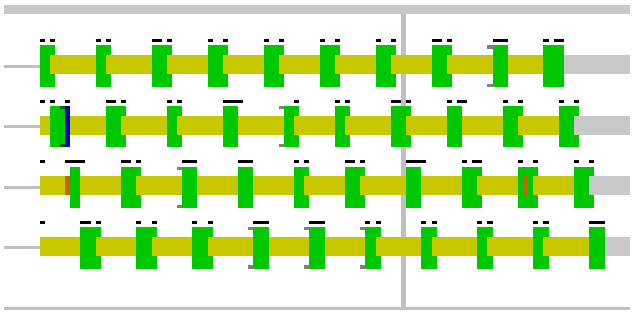
Propriété	Mutex	Semrelay	Spinlock
Est-ce que le fil d'exécution en attente bloque pour l'accès à la section critique ? (WAIT_BLOCKED) **Si oui, illustrez avec une capture d'écran pour chaque verrou concerné.**	Oui (screen 1)	Oui (screen 2)	Non
onEst-ce qu'il se produit des appels système liés au verrouillage? Lesquels? **Si oui, illustrez avec une capture d'écran pour chaque verrou concerné.**	Oui, on les voit bien dans le strace. (screen 3)	Oui. (screen 4)	Non.
Est-ce qu'une famine est possible? Si oui, comment est-elle visible ? **Si oui, illustrez avec une capture d'écran pour chaque verrou concerné**	Non.	Non.	Oui, car tous les processus peuvent prendre la main en meme temps. (screen 5) On le voit quand un fil dexecution reste gris longtemps
La granularité (paramètres *outer* et *inner*) affecte-elle le comportement du verrou ? Si oui, comment ?	Oui, outer affecte le nombre dappels systemes tandis que inner augmente la duree de veouillage	Oui, pareil que pour le mutex.	Non

Tableau 1: Résumé du comportement des verrous

Capture 1



Capture 2



Capture 3

```
clone(child_stack=0x7f803132eff0, flags=CLONE_VM|CLONE_FS|CLO
tls=0x7f803132f700, child_tidptr=0x7f803132f9d0) = 16951
futex(0x7f8032b329d0, FUTEX_WAIT, 16948, NULL) = 0
futex(0x7f8031b309d0, FUTEX_WAIT, 16950, NULL) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}
write(1, "label                N          "..., 60) = 60
```

Capture 4

```
mprotect(0x7f8617f78000, 4096, PROT_NONE) = 0
clone(child_stack=0x7f8618777ff0, flags=CLONE_VM|CLONE
tls=0x7f8618778700, child_tidptr=0x7f86187789d0) = 169
futex(0x7f8619f7b9d0, FUTEX_WAIT, 16984, NULL) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1
write(1, "label                N           "..., 60) =
write(1, "actual          400000    1999980"..., 60) =
```

Capture 5

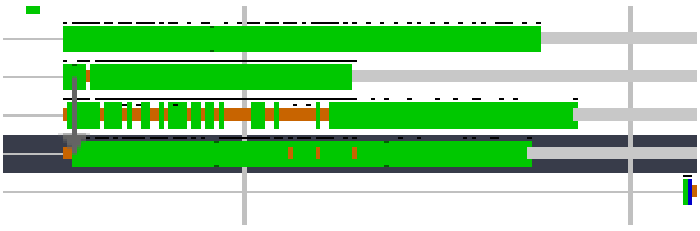


Tableau 2: Captures d'écran

1.1 Implémentation des verrous (initialisation, destruction et utilisation). /3 pt

1.2 Réponses du tableau 1 et justifications à l'aide de captures d'écran (-0,5pt par erreur ou omission). /3 pt

Comportement en cas d'erreurs

Il peut arriver qu'un des fils d'exécution qui ajoute des valeurs à l'ensemble des données ne se lance pas correctement ou s'arrête pendant l'exécution de manière inopinée. Le programme `multilock` simule ce cas d'erreur grâce à l'option `--unstable`. Lorsque cette option est fournie, un des threads annule son exécution en plein milieu de la section critique avec une certaine probabilité.

1.3 Lancer le programme `multilock` avec l'option `--unstable` pour chacun des types de verrous (`mutex`, `semrelay` et `spinlock`). Que se passe-t-il? Comment l'expliquez-vous? /0.5 pt

Réponse : Il n y a rien apres lexecution du programme. Le thread en pleine section critique ne s unlock jamais, il ne donne donc jamais la chance aux autres threads de sexecuter.

1.4 Vous voulez résoudre le problème précédent, c'est-à-dire rendre votre programme plus robuste à la défaillance d'un fil d'exécution qui est dans sa section critique. Expliquez comment vous vous y prendriez dans le cas des librairies `mutex` et `spinlock`. Pourquoi le cas de `semrelay` est-il plus compliqué à gérer? /1.5 pt

Réponse : mutex :

On peut utiliser pthread_mutex_timedlock qui permet de specifier un temps max dattente dun mutex.

On peut utiliser trylock a la place dunlock et utiliser un attribut pour le rendre recursif.

Spinlock : Grace a la fonction mini_spinlock_retry qui a la meme fonction que pour le mutex(trylock).

SemRelay : il contient plusieurs executions simultanées, cest un mutex avec plusieurs executions en parallèles, il faudrait donc verifier pour plusieurs des fils dexecutions, cela rend ce cas plus complique a gerer.

1.5 Implémentez votre correctif dans le cas de la librairie `mutex`. Votre programme ne fournira pas le résultat statistique attendu mais doit être capable de terminer, même /2 pt

en mode instable.

1.6 *BONUS : Implémentez un correctif dans le cas de la librairie* `semrelay`*.* */2 pt*
Ce cas étant plus complexe, il est conseillé d'y revenir à la toute fin si vous avez le temps!

2 Détection d'interblocage [4 pts]

Un interblocage survient lorsqu'il existe un cycle d'attente pour une ressource. Le principal problème lié à un interblocage concerne les possibilités de le détecter. Une manière de faire consiste à **surveiller à intervalles réguliers le progrès de l'exécution**. Si aucun progrès ne se produit, alors il est possible qu'un interblocage soit survenu. Un tel mécanisme de détection est communément appelé *watchdog*.

Vous désirez réaliser un détecteur d'interblocage tel que décrit ci-dessus. Le programme `interblocage` simule un traitement d'une durée variable, avec un accès concurrent aux variables `x` et `y`. Les deux fils d'exécution vont verrouiller `lock_one` et `lock_two` dans un ordre bien choisi pour provoquer un interblocage.

La fonction `watchdog()` est appelée toutes les 100 ms. Son comportement par défaut est d'afficher "watchdog" sur la console. C'est donc à vous d'implémenter un mécanisme correct. On rappelle qu'ici, le rôle du watchdog est de vérifier périodiquement si les traitements effectués par le programme sur certaines variables partagées ont avancé ou non.

2.1 Complétez le verrouillage de `lock_one` et `lock_two` de manière à provoquer un interblocage. Exécutez le programme `interblocage` à plusieurs reprises. /2 pt

- Expliquez comment vous avez placé les verrous. Pourquoi est-ce que votre choix provoque un interblocage?
- L'interblocage survient-il toujours pour les mêmes valeurs de `x` ? Expliquez pourquoi.
- Est-ce possible que l'interblocage ne survienne jamais ? Si oui, proposez un exemple concret.

Réponses :

- Nous avons placé les lock en ordre opposé entre les deux fonctions. Foo utilise les lock dans l'ordre one – two tandis que Bar les utilise dans le sens two – one. Ainsi lorsque les fonction sont exécutées simultanément, Foo capture le premier lock et Bar le second, mais lorsque Foo veut prendre possession du lock two ou Bar du

lock one, ceux-ci sont déjà verrouillés. Un interblocage survient alors.

- L'interblocage ne survient pas toujours sur les mêmes valeurs de x. En effet les valeurs de count déterminées par la fonction random_hog() ne seront pas les même pour les deux fonction (Foo et Bar) puisque qu'elles seront déterminé par des valeurs aléatoires. Ainsi nous avons des temps d'exécution différents à chaque fois et l'interblocage ne survient pas sur les mêmes valeurs de x. Cependant il survient toujours sur des valeurs de x et y égales.

- Il est possible que l'interblocage ne survienne jamais. Un des cas les plus simple serait un enchainement de random_hog() si différents entre les deux fonctions que la première aurait fini d'utiliser les locks quand la seconde voudrait commencer à les utiliser.

2.2 Implémentez la fonction `watchdog()` pour détecter un interblocage. Lorsqu'un interblocage est détecté, on fera un appel à `exit()` pour terminer le programme et tous les fils d'exécution. /1 pt

  - Votre détection d'interblocage est-elle parfaite (i.e détecte bien tous les interblocages) ?

  - Quel est le risque principal d'une telle solution ?

  - Plutôt que de tenter de détecter l'interblocage, comment auriez-vous effectué le verrouillage de `lock_one` et `lock_two` de manière à ne jamais avoir d'interblocage? Expliquez.

Réponses :

- La détection d'interblocage semble parfaite puisqu'elle ne dépend d'aucune variable. En effet elle est déterminée par une horloge et les temps d'exécutions des fonctions. Si l'une met plus de WATCHDOG_UDELAY 100000 (soit 100ms) à s'exécuter, alors on suppose qu'il s'agit d'un deadlock et sigalrm déclenche la fonction watchdog qui met fin au programme.

- Le risque de cette solution est qu'elle peut se déclencher simplement si le temps d'exécution est long mais qu'il n'y a pas effectivement d'interblocage. Par exemple si Foo ou Bar met plus de 100ms à incrémenter x ou si le programme demandait plus de temps que prévu pour s'exécuter (avec plus d'opérations par exemple), watchdog() serait appelée alors qu'il n'y a pas vraiment d'interblocage.

- Il semblerait que l'interblocage puisse être évité simplement en gardant le même ordre de possession des locks dans les deux fonctions Foo et Bar, par exemple one-two (ou inversement) dans les deux fonctions. Ainsi la première à prendre

possession du premier lock pourra librement prendre possession du second lock pendant que l'autre fonction attends. Une fois les lock libérés par la première fonction, la seconde pourra librement prendre possession des deux lock, évitant ainsi tout inter blocage.

2.3 Placez l'appel à `pthread_barrier_wait()` dans les fonctions `worker_foo()` et `worker_bar()` de manière à **forcer l'interblocage dès le premier cycle**. /1 pt

ATTENTION : l'interblocage doit bien être réalisé sur les verrous `lock_one` et `lock_two` et non sur la barrière elle-même.

3 Communication par tubes [6 pts]

L'analyse lexicale est à la base de la classification des documents réalisée par un moteur de recherche. Pour comparer des documents, on pourra par exemple utiliser des méthodes d'analyse des fréquences d'apparition des mots.

Le programme `lexique` est un prototype qui calcule la fréquence des mots dans un texte. Le but de cet exercice est de lire le texte à analyser depuis l'entrée standard (*stdin*) et d'afficher ensuite le résultat sur la sortie standard (*stdout*).

Le traitement comporte deux étapes, **chacune effectuée dans un processus**, tel que montré sur la figure 1.

- La première étape consiste à séparer l'entrée en mots et à calculer leurs tailles.
- La deuxième étape est consacrée à compter les fréquences d'apparitions des mots. Les résultats de cette analyse sont stockés dans une table de hachage.

L'échange d'informations entre les deux processus se fait par le biais de deux tubes (pipes) anonymes. L'un sert à transmettre la taille des mots, tandis que l'autre permet la transmission des mots eux-mêmes.

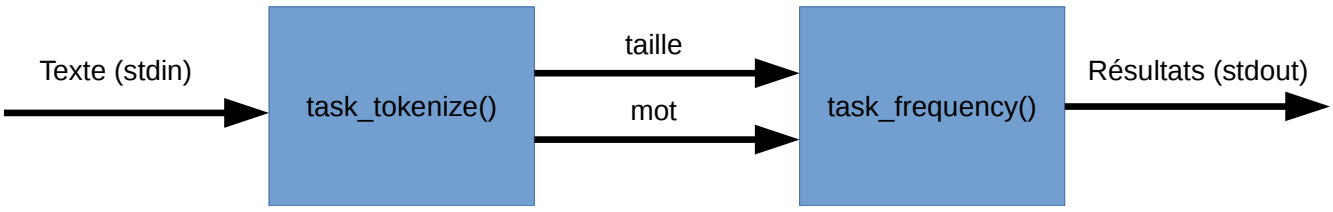


Illustration 1: Schéma de fonctionnement du programme lexique

Les fonctions `task_tokenize()` et `task_frequency()` sont fournies. Vous devez compléter la fonction `main()` du fichier `lexique.c` pour initialiser les tubes et démarrer ces deux tâches en tant que processus. Les fichiers `text-small`, `text-linux` et `text-galaxy` sont fournis à des fins de test. Voici un exemple

d'exécution, qui envoie le fichier `text-small` à l'entrée standard du programme `lexique` :

```
./lexique < text-small
```

3.1 Implémentation du programme (création des tubes, passage des paramètres aux fonctions `task_tokenize()` et `task_frequency()`). /4 pt

3.2 Actuellement, les statistiques de fréquences sont affichées seulement à la fin du traitement. En appuyant sur les touches CTRL-C, le signal SIGINT est envoyé au programme. Celui-ci quitte et les données ne sont pas affichées. /2 pt

Modifiez le gestionnaire d'événements pour afficher les données **avant de quitter**. (Astuce : en envoyant la taille de mot -1 à `task_frequency()`, alors le processus affiche les données et cesse le traitement).

3.3 **Question bonus :** Lorsqu'un tube est plein, alors le processus écrivain bloque en attendant que les données soient consommées. À l'aide de cette technique, complétez le programme remplissage (fichier lexique/remplissage.c) pour déterminer expérimentalement la taille des tubes sous Linux. /1 pt

4 Correction

Points des questions 1 à 4	0
Pénalité de retard	0
Qualité de la langue (-0,5% par faute)	0
Points bonus	0
Note finale sur 20	0