UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

Faculté des Sciences — Département d'Informatique

ULB

INFO-F-201 – Systèmes d'exploitation

Synchronisation

Joël Goossens (N8.107) Olivier Markowitch (N8.106)
Arnaud Leponce (N8.213) Yannick Molinghen (N8.210)
Alexis Reynouard (N8.215)

Résumé

Lorsque plusieurs fils d'exécution ont lieu simultanément, certains problèmes de ***concurrence*** peuvent survenir, c'est-à-dire que plusieurs fils tentent d'accéder à la même ressource partagée au même moment. La portion de code sujette à des problèmes de concurrence est appelée ***section critique***.
Cette ressource peut être de plusieurs natures (fichier, zone mémoire, …). Dans ce TP, nous allons nous intéresser à l'un des problèmes de concurrence les plus courants : la synchronisation d'accès multi-threads à une zone mémoire commune.

concurrence

section critique

Note : *Il est important de distinguer la mémoire partagée entre deux processus et créée explicitement (avec `shm` ou `mmap`), de la zone mémoire commune à deux threads d'un même processus.*

1 Les Mutexes

Les ***mutex*** (pour ***mut****ual* ***ex****clusion*, exclusion mutuelle) sont des structures de données qui permettent de verrouiller des ressources partagées afin de garantir un accès exclusif à ces dernières.

mutex

Concrètement, cela signifie que lorsqu'un thread verrouille une mutex, n'importe quel autre thread qui tente de verrouiller la même mutex est mis en pause (état bloqué) jusqu'à ce que la mutex soit déverrouillée.

L'appel système permettant d'initialiser une mutex est le suivant :

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Le second paramètre peut être `NULL` pour initialiser une mutex avec les attributs de la mutex par défaut. On peut également initialiser une mutex avec les valeurs par défaut en utilisant la macro `PTHREAD_MUTEX_INITIALIZER`.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Une fois initialisée, il existe deux appels système pour verrouiller la mutex (demander l'accès à la zone critique d'exécution).

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Le premier est un appel bloquant. En d'autres termes, le thread est mis en attente (état ***bloqué***) par l'OS jusqu'à ce que la mutex soit libérée. Le second quant à lui retourne la valeur `EBUSY` si le verrouillage de la mutex n'est pas réalisable. En cas de réussite, les deux appels retournent 0.

bloqué

Pour déverrouiller une mutex (une fois que le traitement sur la ressource partagée est terminé), le thread doit exécuter l'appel système :

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Cet appel est très important puisque la ressource partagée ne sera accessible par les autres threads que lorsque la mutex aura été déverrouillée par le thread qui l'a verrouillée (seul ce dernier est autorisé à déverrouiller la mutex).

Les opérations `lock` et `unlock` sont des ***opérations atomiques*** (atomique = qui ne peut pas être coupé), ce qui signifie que leur implémentation garantit que le thread ne sera pas interrompu par l'ordonnanceur au cours de cette opération.

opérations atomiques

Enfin, la fonction `pthread_mutex_destroy` permet de détruire une mutex.

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Exercice 1. *Écrivez un programme qui lance deux threads qui incrémentent chacun 10^5 fois un compteur initialisé à zéro. Le thread principal attend la fin des threads enfants avant de terminer le programme en affichant la valeur finale du compteur.*

- *Écrivez une première version sans mécanisme de synchronisation.*
- *Écrivez une seconde version qui utilise une `mutex` pour éviter les accès concurrentiels.*

ATTENTION : les fonctions sur les mutexes ne sont pas "async-signal safe". Cela signifie qu'elles ne doivent pas être appelées à l'intérieur d'un gestionnaire de signaux (signal handler).

2 Les types atomiques

Le langage C, comme la plupart des langages de programmation, proposent des ***types atomiques*** qui implémentent eux-mêmes (et plus efficacement que les mutexes) la gestion de la concurrence.

types atomiques

Ces types répondent à des usages courants qui nécessitent de la synchronisation tout en simplifiant grandement la mise en place de celle-ci. On trouve par exemple les `atomic_int` qui s'utilisent comme n'importe quel `int` et dont l'utilisation est brièvement illustrée dans le Listing 1.

```
#include <stdatomic.h>
int main() {
    atomic_int x = 0;
    x++;
    ...
}
```

LISTING 1 – Utilisation d'un `atomic_int`.

Exercice 2. *Écrivez une nouvelle version de l'exercice 1 où la synchronisation est gérée par un compteur atomique.*

3 Les sémaphores

Un ***sémaphore*** est une généralisation du concept d'exclusion mutuelle dans laquelle il y a (généralement) plus d'une seule ressource disponible. S'il n'y a qu'une seule ressource disponible, un sémaphore est (presque [1]) identique à une mutex.

sémaphore

Techniquement un sémaphore est un "compteur de places libres". Lorsque le sémaphore a une valeur supérieure à 0, un thread peut le décrémenter et accéder à la ressource. Lorsqu'un processus libère une ressource, le sémaphore est incrémenté.

On illustre régulièrement les sémaphores via un problème de producteur/consommateur où le producteur crée des données et le consommateur les utilise.

Ce qu'il faut bien distinguer des mutexes, c'est que plusieurs consommateurs peuvent consommer en même temps s'il y a plusieurs ressources disponibles.

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
// Link with -pthread
```

Listing 2 – Les appels systèmes relatifs aux sémaphores.

`sem_init` initialise le sémaphore à la valeur `value`. Mettre `pshared` à une valeur non nulle permet de créer un sémaphore inter-processus (et non inter-thread). Ceci dépasse l'objet de ce TP.

`sem_wait` décrémente le sémaphore (verrouille une ressource). Si le sémaphore est déjà à 0, l'appel bloque jusqu'à ce que la valeur du sémaphore ré-augmente ou qu'un signal interrompe l'appel.

Pour l'exhaustivité, nous notons qu'il en existe une variante non bloquante `sem_trywait` et une variante partiellement bloquante `sem_timedwait` qui bloque au plus jusqu'à un instant absolu dans le temps donné puis retourne une erreur si le sémaphore vaut toujours 0 à ce moment.

`sem_post` incrémente (déverrouille une ressource) le sémaphore. Si le sémaphore devient alors supérieur à 0, un autre thread (ou processus) bloqué par `sem_wait` peut alors être réveillé et effectuer son lock du sémaphore.

Tous ces appels retournent 0 en cas de succès.

Exercice 3. *Écrivez un programme avec une fonction `producteur` et une fonction `consommateur`. Le producteur écrit des nombres un à un dans un tableau de 10 entiers et le consommateur les y lit puis les affiche.*

Après avoir produit un nombre, le producteur s'endort pendant cinq secondes après un message qui indique "Au dodo !".

Exercice 4. *Modifiez le programme précédent pour que le producteur produise aléatoirement entre 5 et 20 entiers avant de s'endormir. Comment le comportement de la production change-t-il ?*

1. Le sémaphore reste cependant moins restrictif car, pour une mutex, l'opération de verrouillage puis celle de déverrouillage se font obligatoirement dans le même thread contrairement aux sémaphores.