Synchronisation et concurrence

1 Processus, threads et algorithmes concurrents

1.1 Processus

Définition 1.1. Les tâches d'un système sont appelés des processus. Ils possèdent un numéro d'identification (PID) et sont gérés par le noyau.

Remarque 1.1. Le noyau exécute les tâches les unes à la suite des autres en changeant de tâche (processus) à intervalles réguliers. Le choix est fait par l'ordonnanceur.

Pour changer de processus, l'ordonnanceur interrompt le programme, sauvegarde son contexte, choisit un autre processus et restaure son contexte.

Le système exécute, de manière non déterministe, un entrelacement des programmes. Cette éxécution est non déterministe au sens que si on relance le programme, on obtiendra pas le même entrelacement. En effet, l'ordonnanceur peut ne pas éxécuter les programmes dans le même ordre.

Cela permet aussi que les processus qui sont en attente, ne consomment pas de ressources.

1.2 Fils d'exécution ou threads

La gestion de la concurrence avec des processus a deux défauts : le coût de la communication entre processus ainsi que celui du changement de contexte sont importants.

Définition 1.2. On utilise alors des processus légers nommés aussi fils d'exécution ou threads. Il s'agit d'exécuter plusieurs programmes au sein d'un même processus. Ils ont alors accès au même environnement global.

En **C** et en **Ocaml**, on utilise des threads **POSIX** également appelés pthreads.

Remarque 1.2. Avant la version 5.0 d'**Ocaml** (du 16/12/22), les threads sont tous exécutés sur le même cœur de la machine (pas de multi-cœur). On ne pourra donc pas observer d'accélération des programmes sur ces versions.

La bibliothèque `pthread` en `C`

Il faut d'abord inclure la bibliothèque :

```
# include <pthread.h>
```

Il faut aussi penser à indiquer au compilateur que le programme utilise des threads :

```
gcc -pthread -o test test.c
```

Le fichier **pthread.h** définit un type **pthread_t** dont on peut définir un élément avec :

```
pthread_t mon_thread
```

On dispose des fonctions suivantes :

- **pthread_create(pthread_t * thread_id, pthread_attr_t * attr, void * (* fonction) (void *), void * arguments)**.
  Cela crée un thread identifié par **thread_id**, il exécute la fonction **fonction** avec les arguments **arguments**. La valeur **attr** vaudra toujours **NULL**.

  Remarque 1.3. Une fonction peut prendre une fonction en argument avec la syntaxe ci-dessus.

  Remarque 1.4. Le type **void *** permet de passer ce que l'on veut en argument à l'aide de casts de types.
- **pthread_join(pthread_t thread_id, void ** ret)** suspend l'exécution du thread appelant jusqu'à ce que le thread identifié par **thread_id** ait terminé. La valeur **ret** vaudra toujours **NULL**.
- **sched_yield(void)** : le thread indique à l'ordonnanceur qu'il peut être interrompu. Cela permet d'augmenter l'entrelacement.

Le module `thread` en `Ocaml`

Pour compiler le programme, on utilise :

```
ocmalc -I +threads unix.cma  threads.cma -o test test.ml
```

Les fonctions sont les suivantes :

- `Thread.create : ('a -> 'b) -> 'a -> Thread.t`
  La valeur de retour de `f` n'est pas utilisé.
- `Thread.join : Thread.t -> unit` : pareil qu'en `C`
- `Thread.yield : unit -> unit` : pareil qu'en `C`

Remarque 1.5. On remarque que l'entrelacement des programmes peut produire des résultats non désirés.

Définition 1.3. Quand l'ordre d'exécution change le résultat du programme, on dit qu'on est dans une situation de compétition (en anglais : *race condition*).

Exercice 1.1. Étant données 2 matrices A, B de taille $N \times N$, chaque coefficient du produit peut être calculé de manière indépendante. Utiliser cette propriété pour écrire un programme C qui fait ce produit à l'aide de threads.

1.3 Algorithmes concurrents

Définition 1.4. Une instruction atomique est une instruction qui ne peut pas être interrompue avant sa fin.

Un fil d'exécution (ou thread) est une séquence d'instructions atomiques.

Un programme concurrent est un ensemble fini de fils d'exécution.

L'exécution d'un programme concurrent est un entrelacement des instructions des fils d'exécutions.

2 Algorithmes de synchronisation

2.1 Le problème de l'exclusion mutuelle

Définition 2.1. Une solution au problème de l'exécution mutuelle doit vérifier au moins les deux propriétés suivantes :

- Exclusion mutuelle : à tout instant, un seul fil d'exécution exécute sa section critique.
- Absence d'interblocage : si au moins un processus cherche à entrer dans sa section critique, alors un processus (pas nécessairement le même) entrera dans sa section critique à un moment.

Définition 2.2. On peut ajouter d'autres contraintes à ce problème :

- Absence de famine : un fil d'exécution qui essaie d'entrer en section critique finit par y arriver.
- Attente bornée : si un fil d'exécution est bloqué dans son entrée en section critique, on peut borner par une constante le nombre de fois où chaque autre fil entrera en section critique avant lui.

Définition 2.3. On classe généralement ces deux propriétés en deux catégories : les propriétés de sûreté qui garantissent l'absence des mauvais états et les propriétés de vivacité qui garantissent qu'un bon état sera atteint.

Exercice 2.1. Donner la catégorie de chacune.

Remarque 2.1. On fait les hypothèses suivantes :

- On ne suppose rien sur la section non-critique excepté qu'elle n'influence pas les autres fils.
- Les objets définis dans le code d'entrée et de sortie ne peuvent pas être utilisés ailleurs.
- Un processus ne peut pas s'arrêter dans le code d'entrée, la section critique ou le code de sortie.
- Un processus exécute toujours sa section critique et son code de sortie en un nombre fini d'étapes.

2.2 Algorithme de Petersen

On reprend l'idée des barrières, mais on ajoute une priorité pour éviter les interblocages.

C'est un algorithme pour deux processus.

```
1  tour:=0 ; b[0]:=Faux ; b[1]:=Faux ;
2
3  Entrer(i) :
4      b[i] := Vrai
5      tour := 1-i
6      Attendre ( b[1-i]=Faux OU t=i)
7
8  Sortir(i) :
9      b[i] := Faux
10
11 Exemple pour le fil A :
```

```
12 (A1) : SNC
13 (A2) : b[0] := Vrai
14 (A3) : tour := 1
15 (A4) : Attendre ( b[1]=Faux OU tour=0)
16 (A5) : SC
17 (A6) : b[0] := Faux
```

Exemple d'exécution : `(A1)` ; `(B1)` ; `(A2)` ; `(B2)` ; `(A3)` ; `(A4)` ; `(B3)` ; `(A5)` ; `(A6)` ; `(B1)` ; `(B5)`

Théorème 2.1. *L'algorithme de Petersen satisfait l'exclusion mutuelle.*

Théorème 2.2. *L'algorithme de Petersen est sans interblocage.*

Théorème 2.3. *L'algorithme de Petersen est sans famine.*

Remarque 2.2. On a la propriété d'attente bornée avec $r = 1$.

2.3 Algorithme de la boulangerie de Lamport

On généralise au cas de $n > 2$ fils. On utilise un «numéro de passage».

On dispose d'un tableau d'entiers `numero` et d'un tableau de booléens `choisis` qui indique si un fil est en train de choisir son numéro.

On initialise `choisis` à `Faux` et numéro à 0.

```
1  Entrer(i) :
2      choisis[i] := Vrai
3      t := 0
4      Pour j:= 0 à n-1 :
5          t := max (t,nj)
6      numero[i] := t+1
7      choisis[i] := Faux
8      Pour j:= 0 à n-1 :
9          Attendre ( choisis[j]=Faux)
10         Attendre ( nj = 0 ou (ni,i) ⩽_lex (nj,j))
11
12 Sortir(i) :
13     numero[i] := 0
```

Remarque 2.3. `choisis[i]` et `numero[i]` sont uniquement modifiés par le fil `i`. Les autres fils se contentent de les lire.

Théorème 2.4. *L'algorithme de la boulangerie de Lamport satisfait l'exclusion mutuelle.*

Théorème 2.5. *L'algorithme de la boulangerie de Lamport est sans interblocage.*

Théorème 2.6. *L'algorithme de la boulangerie de Lamport satisfait la propriété «premier arrivé, premier servi» (FIFO), c'est à dire qu'un fil en train d'attendre (aux lignes 8, 9 ou 10) entrera avant tout fil en train d'éxécuter sa SNC.*

Corollaire 2.7. *L'algorithme de Petersen est sans famine et satisfait l'attente bornée.*

3 Programmation concurrente

3.1 Mutex

Définition 3.1. Pour implémenter une section critique, on utilise un verrou ou mutex. Il dispose de 2 opérations : une pour acquérir la propriété du verrou et une autre pour le déverrouiller ensuite. Ce mutex garantit l'exclusion mutuelle : un seul processus à la fois peut verrouiller le mutex.

Remarque 3.1. Il vérifie aussi l'absence d'interblocage et l'absence de famine.

Les mutex en `C`

Il s'agit toujours de la bibliothèque pthread. Le fichier `pthread.h` définit un type `pthread_mutex_t` :

```
pthread_mutex_t mon_mutex
```

On dispose des fonctions suivantes :

- `pthread_mutex_init( pthread_mutex_t * m, pthread_mutexattr_t * attr)`
  L'attribut vaudra à priori NULL.
- `pthread_mutex_lock( pthread_mutex_t *m)`
- `pthread_mutex_unlock( pthread_mutex_t *m)`
- `pthread_mutex_destroy( pthread_mutex_t *m)`

Le module `Mutex` en `Ocaml`

Les fonctions sont les suivantes :

- `Mutex.create : unit -> Mutex.t`
- `Mutex.lock : Mutex.t -> unit`
- `Mutex.unlock : Mutex.t -> unit`

Exercice 3.1. Récrire le programme OCaml pour incrémenter un compteur avec un mutex.

3.2 Sémaphores

Définition 3.2. Un sémaphore est un entier que les fils peuvent incrémenter. Ils peuvent aussi le décrémenter quand il est non nul. Sinon, ils attendent qu'il soit non-nul.

L'exclusion mutuelle est garantie par le système.

Les sémaphores en `C`

Le fichier `semaphore.h` définit un type `sem_t` :

```
sem_t mon_semaphore
```

On dispose des fonctions suivantes :

- `sem_init( sem_t * s, int sh, unsigned int v)`
  Initialise le sémaphore s à la valeur v. L'attribut sh vaudra à priori 0.
- `sem_wait( sem_t *s)` Décrémenter ou attendre.
- `sem_post( sem_t *s)` Incrémenter.
- `sem_destroy( sem_t *s)`

Le module `Semaphore.Counting` en `Ocaml`

Avec l'alias `module Sem = Semaphore.Counting`, les fonctions sont les suivantes :

- `Sem.create : unit -> Sem.t`
- `Sem.acquire : Sem.t -> unit`
- `Sem.release : Sem.t -> unit`

3.3 Le problème du rendez-vous

Dans le problème du rendez-vous, des fils d'exécution effectuent une première phase de calcul puis doivent s'attendre avant de passer à une deuxième phase de calcul. Aucun processus ne doit commencer la deuxième phase sans les autres.

On appelle aussi cela une barrière de synchronisation.

3.4 Le problème producteur-consommateur

On dispose de deux types de fils d'exécution : les producteurs qui produisent des données et les consommateurs qui utilisent ces donnés. Ils communiquent à l'aide d'un tampon (buffer) partagé entre tous.

On ne veut pas qu'un consommateur soit bloqué alors qu'il y a des données à utiliser.

Exercice 3.2. 1. Écrire en Ocaml un type tampon et des fonctions cree_tampon, produit et consomme qui résolvent le problème précédent à l'aide d'une file partagée

2. Adapter au cas où le tampon est un tableau à N cases.

Dans ce cas, on ne veut pas non plus qu'un producteur soit bloqué alors qu'il y a de la place pour écrire ses données.