Série d'exercice #4

Solution

IFT-2245

12 février 2019

4.1 Condition de course

Dans beaucoup de systèmes, il peut y avoir des conditions de course. Prenons l'exemple d'un compte en banque partagé par une femme et son mari, et imaginons que le mari fait un appel à `withdraw` au même moment que la femme fait un appel à `deposit`. Décrire tout ce qui peut se passer dans ce genre de circonstance, et comment éviter les différents problèmes.

4.1 Condition de course (Solution)

Dans beaucoup de systèmes, il peut y avoir des conditions de course. Prenons l'exemple d'un compte en banque partagé par une femme et son mari, et imaginons que le mari fait un appel à `withdraw` au même moment que la femme fait un appel à `deposit`. Décrire tout ce qui peut se passer dans ce genre de circonstance, et comment éviter les différents problèmes.

Solution optimiste

Balance initial : 5000

Homme	Femme
b = Read balance	b = Read balance
	b = b+1000
b = b-500	Store b
Store b	

Balance final : 4500

4.1 Condition de course (Solution)

Dans beaucoup de systèmes, il peut y avoir des conditions de course. Prenons l'exemple d'un compte en banque partagé par une femme et son mari, et imaginons que le mari fait un appel à `withdraw` au même moment que la femme fait un appel à `deposit`. Décrire tout ce qui peut se passer dans ce genre de circonstance, et comment éviter les différents problèmes.

Solution optimiste

Balance initial : 5000

Homme	Femme
b = Read balance	b = Read balance
	newB = b+1000
newB = b-500	CheckAndStore b newB
CheckAndStore b newB	

Balance final : 5500

4.2 Mutex avec swap

Utiliser l'instruction spéciale atomique `swap` pour implanter des primitives d'exclusion mutuelle `acquire` et `release` qui assurent l'attente *limité*.

4.2 Mutex avec swap (Solution)

Utiliser l'instruction spéciale atomique `swap` pour implanter des primitives d'exclusion mutuelle `acquire` et `release` qui assurent l'attente *limité*.

Solution swap

```
void acquire(bool *lock) {
  bool flag = true;
  while(flag)
    swap(flag, lock);
}
void release(bool *lock) {
  *lock = false;
}
```

4.2 Mutex avec swap (Solution)

Utiliser l'instruction spéciale atomique `swap` pour implanter des primitives d'exclusion mutuelle `acquire` et `release` qui assurent l'attente *limité*.

Solution compare&swap

```
void acquire(bool *lock) {
  while(!compare&swap(lock, false, true));
}
void release(bool *lock) {
  *lock = false;
}
```

4.3 Atomicité

Montrer que si les opérations `wait` et `signal` des sémaphores ne sont pas exécutée atomiquement, alors l'exclusion mutuelle peut être violée.

4.3 Atomicité (Solution)

Montrer que si les opérations `wait` et `signal` des sémaphores ne sont pas exécutée atomiquement, alors l'exclusion mutuelle peut être violée.

Définition

Atomicité : Propriété d'une instruction de s'exécuté uniquement de façon complète. Une instruction atomique est indivisible.

Solution

Le wait entraine une lecture puis une décrémentation du sémaphore. Il serait donc possible pour les 2 threads d'effectuer une lecture avant que la décrémentation de l'autre thread s'effectue.
Si le sémaphore était à un alors les deux threads pourraient rentrer dans la section critique.

4.4 Section critique à 2

Voici un candidat-solution au problème de la section critique :

```
bool flag[2] = { false, false };
int turn;

void enter (void) {
  flag[myself] = true;
  while (flag[other] && turn == other) /*wait*/;
}

void leave (void) {
  turn = other;
  flag[myself] = false;
}
```

Pour chacune des trois propriétés désirées (exclusion mutuelle, progrès, et attente limitée), prouver qu'elle est vérifiée ou montrer un contre exemple.

4.4 Section critique à 2 (Solution)

Exclusion mutuelle

`turn` n'est pas initialisé correctement. Si les deux processus entre environ dans en même temps (avant l'exécution de `leave`) le teste de `turn==other` sera `false` pour les 2. Les deux exécuterons leurs section critique. Le problème peut être évité si `turn=other` est mis dans la fonction `enter` au lieu de `leave`.

Progrés

Attente limitée

4.4 Section critique à 2 (Solution)

Exclusion mutuelle

`turn` n'est pas initialisé correctement. Si les deux processus entre environ dans en même temps (avant l'exécution de `leave`) le teste de `turn==other` sera `false` pour les 2. Les deux exécuterons leurs section critique. Le problème peut ètre évité si `turn=other` est mis dans la fonction `enter` au lieu de `leave`.

Progrés

P_0 ne peut être pris dans une attente que si `flag[1]==true` et `turn==1`

Attente limitée

4.4 Section critique à 2 (Solution)

Exclusion mutuelle

`turn` n'est pas initialisé correctement. Si les deux processus entre environ dans en même temps (avant l'exécution de `leave`) le teste de `turn==other` sera `false` pour les 2. Les deux exécuterons leurs section critique. Le problème peut être évité si `turn=other` est mis dans la fonction `enter` au lieu de `leave`.

Progrés

P_0 ne peut être pris dans une attente que si `flag[1]==true` et `turn==1`

Attente limitée

Si P_1 n'est pas prêt d'entrer dans sa section critique alors, `flag[1]==false` et P_0 peut s'exécuter sans attente. Lorsque P_0 sort de la section critique il assigne `turn==1` et `flags[0]=false` alors P_1 peut alors rentrer dans la section critique.

4.5 Inhiber les interruptions

Expliquer pourquoi inhiber les interruptions n'est pas une technique appropriée pour implémenter les primitives de synchronisation dans un système multiprocesseur.

4.5 Inhiber les interruptions (Solution)

Expliquer pourquoi inhiber les interruptions n'est pas une technique appropriée pour implémenter les primitives de synchronisation dans un système multiprocesseur.

Solution

Si un processus possède le droit d'inhiber les interruptions, il peut alors empêcher la préemption faite par le timer du processeur et ainsi utiliser le processeur tant qu'il le veut.

4.6 Spinlocks

Expliquer pourquoi les *spinlock* sont inappropriés dans un système monoprocesseur, mais sont souvent utilisés dans les systèmes multiprocesseurs ?

4.6 Spinlocks (Solution)

Expliquer pourquoi les *spinlock* sont inappropriés dans un système monoprocesseur, mais sont souvent utilisés dans les systèmes multiprocesseurs ?

Solution

Le processeur risque d'être monopolisé par le thread qui est en spinlock. Il est donc impossible pour les autres processus de relâcher la ressource attendu par le spinlock tant que le spinlock est en exécution.
Dans un système multiprocesseur ils sont utilisé pour éviter le coût associer au changement de contexte.

4.7 Section critique à 3

Prendre la solution de peterson au problème de la section critique et le généraliser à 3 processus.

4.7 Section critique à 3 (Solution)

Prendre la solution de peterson au problème de la section critique et le généraliser à 3 processus.

Solution ?

Soit les `threads` 0, 1 et 2 et les 2 salles d'attentes(`level`) et un tableau indiquant le dernier dans la salle `l`(`last_to_enter`)

```
for(l = 0; l < 3; ++l) {
  level[myself] = l;
  last_to_enter[l] = myself;
  while(last_to_enter[l] == myself
  /* Pour i != myself -> level[i] > l */
        && exists_gt(level, i, l))
    wait();
}
```