Mutex et sémaphores

PCO

4 - Mutex

Résumé du document

Definition

Table des matières

1. Attente active vs passive

Les mutex et sémaphores viennent nous permettre d'éviter l'attente active lors-ce que des threads veulent accéder aux mêmes ressources.

1.1. Rappel

Attente active

- Le processus utilise une boucle pour vérifier continuellement une condition, consommant des ressources processeur.
- `while (!condition) { }`
  - Consomme de la ressource pour faire avancer la boucle `while`

Attente passive

- Le processus est mis en pause par le système jusqu'à ce qu'une condition soit remplie, libérant le processeur.
- Utilisation de `mutex` ou `semaphore`

2. Mutex

Un mutex est/a

- une variable booléenne
- possède une liste d'attente
- est manipulé par deux opérations **atomiques**
  - `Verouille(v)`
  - `Deverouille(v)`
- Un mutex possède un **propriétaire**: le thread ayant obtenu le verrou est propriétaire du mutex jusqu'à ce qu'il le déverouille. **Seul le propriétaire peut déverouiller son mutex**.
- Les fonctions `mutex.lock()` et `mutex.unlock()` sont atomiques.
- Verrouiller un mutex déjà verrouillé bloque le thread **appelant** (deadlock).

2.1. Utilisation du mutex

Grâce au mutex nous pouvons donc protéger une section critique

```
...
mutex.lock();
/* section critique */
mutex.unlock();
...
```

2.2. PcoMutex

Une classe `PcoMutex` est mise à disposition dans le cours de PCO. Cette classe offre la possibilité de créer des mutex et d'utiliser certaines fonctions dessus.

```
#include <pcosynchro/pcomutex.h>

PcoMutex mutex;         // Après initialisation toujours déverouillé

mutex.lock();           // Verrouille le mutex
```

- Si un thread tente de reverouiller un mutex verrouillé par lui il y a deadlock
- Si le mutex est déjà verrouillé par un autre thread, le thread appelant est suspendu jusqu'à ce que le mutex soit déverrouillé

```
mutex.unlock();         // Déverrouille le mutex
```

- Déverrouille (libère) le mutex; le mutex est supposé être verrouillé par le thread appelant avant l'appel à `unlock()`

2.3. Mutex récursif

- Peut être verrouillé plusieurs fois par le même thread; un compteur mémorise le nombre de verrouillages effectués et le verrou sait quel thread en est le propriétaire
- La fonction `unlock()` doit être appelée autant de fois que la fonction `lock()` pour que le verrou se retrouve déverrouillé

```
PcoMutex mutex(PcoMutex::RecursionMode::Recursive);
```

Non récursif

```
void function() {
    PcoMutex mutex;
    mutex.lock();
    mutex.lock();  <-- Blocage

    mutex.unlock();
    mutex.unlock();
}
```

Récursif

```
void function() {
    PcoMutex mutex(PcoMutex::Recursive);
    mutex.lock();
    mutex.lock();  <-- Pas de blocage

    mutex.unlock();
    mutex.unlock();
}
```

2.3.1. Bonne pratique

De manière générale il est préférable de verrouiller et deverrouiller un mutex dans la même fonction c-à-d au plus proche de la section critique.

Mauvais exemple

```
class BadClass {
private:
    PcoMutex mutex;

public:
    void function1() {
        mutex.lock();
        doSomething();
        mutex.unlock();
    }

    void doSomething() {
        mutex.lock();  <-- Blocage
        ...
        mutex.unlock();
    }
}
```

Bon exemple

```
class GoodClass {
private:
    PcoMutex mutex(PcoMutex::Recursive);

public:
    void function() {
        mutex.lock();
        doSomething();
        mutex.unlock();
    }

    void doSomething() {
        mutex.lock();  <-- Pas de blocage
        ...
        mutex.unlock();
    }
}
```

Mauvais exemple

```
class BadClass {

    PcoMutex mutex;

    void function1() {
        mutex.lock();
        doSomething();
    }

    void doSomething() {
        // really do
        ...
        mutex.unlock();
    }
}
```

Bon exemple

```
class GoodClass {

    PcoMutex mutex;

    void function() {
        mutex.lock();
        ...
        mutex.unlock();
    }

    void doSomething() {
        mutex.lock();
        ...
        mutex.unlock();
    }
}
```

3. Sémaphores

Les sémaphores sont:

- une généralisation des mutex
- proposé par Djikstra en 62 ou 63
- **comprennent une variable entière plutôt qu'un booléen**
- opérations d'accès (atomiques)
  - `P(s)` → pour tester
  - `V(s)` → pour incrémenter

3.1. Section critique avec sémaphore

- Une section critique peut être protégée par un sémaphore
- Pour un nombre quelconque de tâches
- Avec un sémaphore qu'on appelle `mutex`, initialisé à 1

```
...
P(mutex);
/* section critique */
V(mutex);
...
```

3.2. Section contrôlée

- En initialisant le sémaphore `mutex` à $v > 1$
- Jusqu'à v tâches peuvent être admises simultanément dans la section critique
- on parle alors de section contrôlée

3.3. PcoSemaphore

Comme pour la librairie `PcoMutex` le cours PCO met à disposition la librairie pour les sémaphores avec les fonctions suivantes:

```
#include <pcosynchro/pcosemaphore.h>

PcoSemaphore sem;       // valeur par défaut est 0

sem.acquire();
```

- Décrémente (bloque) le sémaphore spécifié. Si la valeur du sémaphore est > 0, sa valeur est décrémentée et la fonction retourne immédiatement
- Si sa valeur est égale à 0, alors l'appel bloque jusqu'à ce que le thread soit relâché par un appel release().

```
sem.release();
```

- Incrémente (débloque) le sémaphore spécifié
- Si la valeur du sémaphore est ≤ 0, alors un autre thread bloqué dans un appel à acquire sera débloqué de la file d'attente.

3.4. Strong vs. weak

Il existe deux types de sémaphores

- Sémaphore fort (strong)
  - La file d'attente des sémaphores est ordonnées (FIFO)
  - Les threads sont donc réveillés dans l'ordre de leur arrivée
- Sémaphore faible (weak)
  - La file d'attente n'est pas ordonnée
  - Les threads peuvent être réveillés dans un ordre quelconque
- En général les implémentations offrent des sémaphores forts
  - C'est le cas de **PcoSemaphore**

4. Coordination

- Un sémaphore peut être utilisé pour synchroniser l'exécution de deux tâches, assurant que l'une s'exécute avant l'autre.

Exemple sans coordination :

```
void T1() {
    std::cout << "T1: I1" << std::endl; // I_1
}
void T2() {
    std::cout << "T2: I2" << std::endl; // I_2
}
int main() {
    PcoThread thread1(T1);
    PcoThread thread2(T2);
    thread1.join();
    thread2.join();
    return 0;
}
```

Exemple avec coordination via un sémaphore :

```
static PcoSemaphore sync(0);
void T1() {
    PcoThread::usleep(1000000); // I_1
    std::cout << "T1: fini sleep" << std::endl;
    sync.release();
}
void T2() {
    std::cout << "T2: avant acquire" << std::endl;
    sync.acquire();
    std::cout << "T2: apres acquire" << std::endl; // I_2
}
int main() {
    PcoThread thread1(T1);
    PcoThread thread2(T2);
    thread1.join();
    thread2.join();
    return 0;
}
```