Introduction à la programmation concurrente

Design patterns, trucs et astuces

Yann Thoma, Jonas Chapuis

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Février 2018

Choix du mécanisme

- Sémaphore (solution spécifique)
- Sémaphore (solution générale)
- Moniteur de Mesa
- Moniteur de Hoare

Implémentations avec sémaphores: solution spécifique

- Utilisation des sémaphores:
  - Comme un mutex : initialisation à 1
  - Pour garantir une gestion des arrivées en FIFO: initialisation à 1
  - Pour bloquer les autres threads: initialisation dépendante de l'utilisation
- Pour:
  - Code potentiellement plus simple (lignes de code)
  - Grande flexibilité des sémaphores
  - Pratique pour du producteurs-consommateurs, plus *dangereux* dans d'autres contextes
- Contre:
  - Code très *home made*
  - Plus grand risque d'erreurs
  - Plus difficile à expliquer et documenter
  - Beaucoup d'appels aux primitives de synchronisation -> performance

Implémentations avec sémaphores: solution générale

- Utilisation des sémaphores:
  - Un mutex : initialisation à 1
  - Pour les attentes bloquantes: initialisation à 0
    - Association d'un compteur de threads bloqués
  - Des variables internes pour représenter l'état du système
- Pour:
  - Manière méthodique de résoudre les problèmes
  - Code plus lisible par les tiers
  - Une manière de penser à appliquer à tous les problèmes
- Contre:
  - Plus de lignes de code que les solutions spécifiques
  - Désolé, je ne vois pas

Mélange des deux approches sémaphores

- Potentiellement intéressant d'utiliser un sémaphore pour garantir un ordre FIFO
  - Un sémaphore *fifo* qui entoure ce que fait l'algorithme général
  - Peut faciliter ce cas précis
- Mais attention à ne pas se *mélanger les pinceaux*

Exemple

```
void myFunction() {
    fifo.acquire();
    mutex.acquire();
    ...
    if (!myCondition) {
        nbWaiting ++;
        mutex.release();
        waitingSem.acquire();
    }
    ...
    mutex.release();
    fifo.release();
}
```

Implémentations avec moniteur de Mesa

- Utilisation d'un moniteur de Mesa:
  - Un mutex
  - des variables conditions (Mesa)
  - Lors du réveil d'une tâche, celle-ci doit réacquérir le mutex
  - Il faut donc revérifier la condition
    - Association d'un compteur de threads bloqués
    - Pas forcément nécessaire, dépend des cas
  - Des variables internes pour représenter l'état du système
- Pour:
  - Manière méthodique de résoudre les problèmes
  - Code facilement lisible par les tiers
  - Une manière de penser à appliquer à tous les problèmes
- Contre:
  - Par rapport aux sémaphores, si une fonction doit être appelée depuis le moniteur elle le bloque
  - La réaquisition du mutex rend le code plus facilement erroné

Implémentations avec moniteur de Hoare

- Utilisation d'un moniteur de Hoare:
  - Un mutex
  - des variables conditions (Hoare)
  - Lors du réveil d'une tâche, celle-ci se fait offrir le mutex
  - Pas besoin de revérifier la condition
    - Pas nécessaire d'associer un compteur de threads bloqués
  - Des variables internes pour représenter l'état du système
- Pour:
  - Manière méthodique de résoudre les problèmes
  - Code facilement lisible par les tiers
  - Une manière de penser à appliquer à tous les problèmes
  - Pas de perte de section critique au réveil d'une tâche
- Contre:
  - Par rapport aux sémaphores, si une fonction doit être appelée depuis le moniteur elle le bloque
  - En général pas offert par les environnements de développement

Méthodologie

1. Identifier le pattern général
   - Producteur-consommateur
   - Lecteurs-rédacteurs
   - Synchronisation
2. Choisir un mécanisme
3. Identifier les variables nécessaires
4. Identifier les conditions de bloquage
5. Identifier les conditions qui génèrent un réveil
6. Implémenter le tout avec le mécanisme choisi

Méthodologie : sémaphores (solution générale)

- Un sémaphore `mutex` initialisé à 1
- Pour chaque attente spécifique:
  - Un sémaphore initialisé à 0
  - Un compteur de nombre de threads en attente

Mise en attente

```
waitingCounter ++;
mutex.release();
waitingSem.acquire();
// Eventuellement
mutex.acquire();
```

Réveil

```
if (waitingCounter > 0) {
    waitingCounter --;
    waitingSem.release();
}
```

Méthodologie : sémaphores (solution générale)

Mauvaise idée (Pourquoi?)

```
void myFunction() {
    mutex.acquire();
    while (!myCondition) {
        waitingCounter ++;
        mutex.release();
        waitingSem.acquire();
        mutex.acquire();
        waitingCounter --;
    }
    ...

    mutex.release();
}

void anotherFunction() {
    mutex.acquire();
    myCondition = true;
    if (waitingCounter > 0)
        waitingSem.release();
    mutex.release();
}
```

Idée correcte

```
void myFunction() {
    mutex.acquire();
    while (!myCondition) {
        waitingCounter ++;
        mutex.release();
        waitingSem.acquire();
        mutex.acquire();
    }
    ...

    mutex.release();
}

void anotherFunction() {
    mutex.acquire();
    myCondition = true;
    if (waitingCounter > 0) {
        waitingCounter --;
        waitingSem.release();
    }
    mutex.release();
}
```

Méthodologie : sémaphores (solution générale)

Idée correcte

```
void myFunction() {
    mutex.acquire();
    while (!myCondition) {
        waitingCounter ++;
        mutex.release();
        waitingSem.acquire();
        mutex.acquire();
    }
    ...

    mutex.release();
}

void anotherFunction() {
    mutex.acquire();
    myCondition = true;
    if (waitingCounter > 0) {
        waitingCounter --;
        waitingSem.release();
    }
    mutex.release();
}
```

Meilleure idée

```
void myFunction() {
  mutex.acquire();
  if (!myCondition) {
    waitingCounter ++;
    mutex.release();
    waitingSem.acquire();
  }
  ...

  mutex.release();
}

void anotherFunction() {
  mutex.acquire();
  myCondition = true;
  if (waitingCounter > 0) {
      waitingCounter --;
    waitingSem.release();
  }
  else
      mutex.release();
}
```

Méthodologie : sémaphores (solution générale)

- De manière générale éviter de relâcher l'exclusion mutuelle

Méthodologie : moniteur

- Faire attention: Mesa ou Hoare?

$\Rightarrow$ Impact sur les réveils

  - Mesa: Il faut réacquérir le mutex
  - Hoare: Le thread réveillé s'exécute en exclusion mutuelle et repasse la main ensuite au "réveilleur"

- Réveils multiples
  - Mesa : possible
  - Hoare : impossible, il faut faire des réveils en cascade

Méthodologie : moniteur de Mesa

- Attention à revérifier la condition au réveil

```
void myFunction() {
    mutex.lock();

    while (!myCondition) {
        condVar.wait(&mutex);
    }

    mutex.unlock();
}
```

Comparaison

	Sémaphore	**Mesa**	**Hoare**
Attente	`acquire()`	`wait(&mutex)`	`cond.wait()`
Fonction réveil	`release()`	`wakeOne()` `wakeAll()`	`cond.signal()`
Réveil si pas d'attente	incrémentation	Pas d'effet	Pas d'effet
Après réveil		Doit réacquérir le mutex	Mutex transmis
Retester la condition	Dépend	Oui	Non
Appel d'une fonction externe[1]	Oui	Non	Non

[1] Coûteuse en temps d'exécution

Design patterns

- Quelques patterns souvent utilisés:
  - Boss-worker : producteurs-consommateurs
  - Préchargement
  - Parallélisme pur

Parallélisme en C++11

- Exemple d'une boucle avec des itérations indépendantes
- Chaque itération très coûteuse en temps

```
for (int i = 0; i < maxIt; ++i) {
    // Do my iteration
}
```

Parallélisme en C++11

```
int nbThreads = std::thread::hardware_concurrency();

std::vector<std::thread> workers;
for(int threadId = 0;threadId< nbThreads; threadId++) {
  workers.push_back(std::thread([threadId, variable1, ...]()
      {
        // Do something nice, knowing the thread Id

        int start = threadId * (maxIt / nbThreads);
        int end   = (threadId + 1 ) * (maxIt / nbThreads);
        for (int i = start; i < end; ++i) {
            // Do my iteration
          }
        }

      }));
}
std::for_each(workers.begin(), workers.end(), [](std::thread &t)
{
  t.join();
});
```

Oubli de déverouiller un mutex: utilisation d'un locker

```
int complexFunction(int flag)
{
  mutex.lock();
  int retVal = 0;
  switch (flag) {
    case 0:
    retVal = moreComplexFunction(flag);
    break;
    case 2:
    {
      int status = anotherFunction();
      if (status < 0) {
        mutex.unlock();
        return -2;
      }
      retVal = status + flag;
    }
    break;
    default:
    if (flag > 10) {
      mutex.unlock();
      return -1;
    }
    break;
  }
  mutex.unlock();
  return retVal;
}
```

```
int complexFunction(int flag)
{
  QMutexLocker locker(&mutex);
  int retVal = 0;
  switch (flag) {
    case 0:
    return moreComplexFunction(flag);
    case 2:
    {
      int status = anotherFunction();
      if (status < 0)
          return -2;
      retVal = status + flag;
    }
    break;
    default:
    if (flag > 10)
        return -1;
    break;
  }
  return retVal;
}
```

Terminaison

- Attention à ne pas détruire un thread qui est en attente sur un objet de synchronisation
- Comment faire?
  - Avoir une variable permettant d'indiquer que le thread doit s'arrêter
    1. Mettre à jour cette variable
    2. Relâcher les threads en attente
    3. Chaque thread interprète la variable pour se terminer

Exemple

```
// Dans les threads
sem.acquire();
if (shouldTerminate) {
    ...
}

// Dans un destructeur
shouldTerminate = true;
sem.release(); // A faire le bon nombre de fois
```

Variables statiques

Exemple

```
MyObject *myFunction()
{
  static MyObject *instance = new MyObject();
  return instance;
}
```

- Que se passe-t-il si deux threads appellent cette fonction pour la première fois?
- C++11 nous garantit que l'initialisation ne sera faite qu'une fois

Assurer un ordre avec QWaitCondition

```
std::vector<int> waitingIds;
int currentId = 0;
int toReleaseId = -1;

void myFunction() {
    mutex.lock();
    if (!myCondition) {
        int id = currentId ++;
        waitingIds.push_back(id);
        while (!myCondition && (toReleaseId!= id))
            myCondition.wait(&mutex);
        waitingIds.erase(0);
    }
    mutex.unlock();
}

void anotherFunction() {
    mutex.lock();
    if (waitingIds.size() > 0) {
        toReleaseId = waitingIds[0];
        myCondition.wakeAll();
    }
    mutex.unlock();
```

Code source