

Introduction à la programmation concurrente

Récapitulatif

Yann Thoma, Fiorenzo Gamba

Février 2021

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



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

Méthodologie

1. Identifier le modèle général de threading
 - Modèle *délégation* (*boss-worker model*)
 - Modèle *pair* (*peer model*)
 - Le modèle *pipeline*

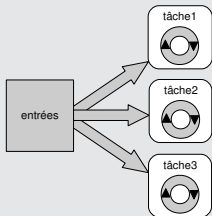
1. Identifier le modèle général de threading
 - Modèle *délégation* (*boss-worker model*)
 - Modèle *pair* (*peer model*)
 - Le modèle *pipeline*
2. Identifier les paradigmes de synchronisation
 - Producteurs-consommateurs
 - Lecteurs-rédacteurs
 - Coordination

1. Identifier le modèle général de threading
 - Modèle *délégation* (*boss-worker model*)
 - Modèle *pair* (*peer model*)
 - Le modèle *pipeline*
2. Identifier les paradigmes de synchronisation
 - Producteurs-consommateurs
 - Lecteurs-rédacteurs
 - Coordination
3. Choisir un mécanisme de synchronisation
 - Sémaphore (solution spécifique ou solution générale)
 - Moniteur de Mesa
 - Moniteur de Hoare

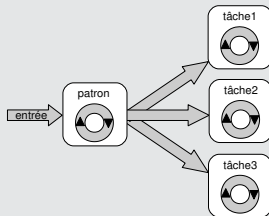
1. Identifier le modèle général de threading
 - Modèle *délégation* (*boss-worker model*)
 - Modèle *pair* (*peer model*)
 - Le modèle *pipeline*
2. Identifier les paradigmes de synchronisation
 - Producteurs-consommateurs
 - Lecteurs-rédacteurs
 - Coordination
3. Choisir un mécanisme de synchronisation
 - Sémaphore (solution spécifique ou solution générale)
 - Moniteur de Mesa
 - Moniteur de Hoare
4. Identifier les éléments clé de synchronisation
 - variables (état, conditions)
 - conditions de blocage
 - conditions de réveil

Modèles typiques de threading (rappel)

Modèle pair



Modèle délégation



Modèle pipeline



- Producteurs-consommateurs
 - Echange de données entre différents threads
- Lecteurs-rédacteurs
 - Protection de ressources
- Coordination
 - Synchronisation-coordination au sens large

Mécanismes

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
 - Ordre de traitement garanti avec des sémaphores forts
- Contre:
 - Code très *home made*
 - Plus grand risque d'erreurs
 - Plus difficile à expliquer et documenter
 - Beaucoup d'appels aux primitives de synchronisation
⇒ performance moins bonne

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
 - Exemple: Un sémaphore *fifo* qui entoure ce que fait l'algorithme général
- 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 internes pour représenter l'état du système
 - Des variables conditions (Mesa)
 -  Lors du réveil d'une tâche, celle-ci doit réacquérir le mutex
 - Il faut donc vérifier la condition
 - Association d'un compteur de threads bloqués
 - Pas forcément nécessaire, dépend des cas
- Pour:
 - Manière méthodique de résoudre les problèmes
 - Code facilement lisible par les tiers
 - Une manière de penser à appliquer à tous les problèmes
- Contre:
 - La réacquisition du mutex rend le code plus facilement erroné

Implémentations avec moniteur de Hoare

- Utilisation d'un moniteur de Hoare:
 - Un mutex
 - Des variables internes pour représenter l'état du système
 - 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
- 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

Rappels méthodologiques

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();
}
```

Idee 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

- Faire attention: Mesa ou Hoare?

⇒ 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

- Attention à revérifier la condition au réveil

```
void myFunction() {  
    mutex.lock();  
  
    while (!myCondition) {  
        condVar.wait(&mutex);  
    }  
  
    mutex.unlock();  
}
```

Comparaison

	Sémaphore	Mesa	Hoare
Attente	<code>acquire()</code>	<code>wait(&mutex)</code>	<code>cond.wait()</code>
Réveil	<code>release()</code>	<code>notifyOne()</code> <code>notifyAll()</code>	<code>cond.signal()</code>
Réveil sans attente	incrémentation	pas d'effet	pas d'effet
Après réveil	-	doit réacquérir le mutex	mutex transmis
Retester la condition	dépend	oui (en général)	non
Appel d'une fonction externe ¹	oui	non	non

¹Avant endormissement, coûteuse en temps d'exécution

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?


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

Techniques diverses

-  Attention à ne pas détruire un thread qui est en attente sur un objet de synchronisation

-  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
 - **PcoThread** offre ce mécanisme

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
 - **PcoThread** offre ce mécanisme

Exemple

```
// Dans les threads
sem.acquire();
if (shouldTerminate) {
    ...
}
// Dans un destructeur
shouldTerminate = true;
sem.release(); ← A faire le bon nombre de fois
```

Oubli de déverrouiller un mutex: utilisation d'un guard

```
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)
{
    const std::lock_guard<std::mutex>
        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;
}
```

- 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
        }
    }));
}

for (auto& worker: workers) {
    worker.join();
}
```

- Un *Futur* représente un résultat de fonction asynchrone

- Un *Futur* représente un résultat de fonction asynchrone
 - Autrement dit: encapsule une valeur disponible dans le *futur*

- Un *Futur* représente un résultat de fonction asynchrone
 - Autrement dit: encapsule une valeur disponible dans le *futur*
- Niveau d'abstraction supérieur à celui du thread

- Un *Futur* représente un résultat de fonction asynchrone
 - Autrement dit: encapsule une valeur disponible dans le *futur*
- Niveau d'abstraction supérieur à celui du thread
 - Permet le chaînage d'appels asynchrones et leur orchestration

- Un *Futur* représente un résultat de fonction asynchrone
 - Autrement dit: encapsule une valeur disponible dans le *futur*
- Niveau d'abstraction supérieur à celui du thread
 - Permet le chaînage d'appels asynchrones et leur orchestration
 - Permet la composition fonctionnelle

- Un *Futur* représente un résultat de fonction asynchrone
 - Autrement dit: encapsule une valeur disponible dans le *futur*
- Niveau d'abstraction supérieur à celui du thread
 - Permet le chaînage d'appels asynchrones et leur orchestration
 - Permet la composition fonctionnelle
- Utile avec des opérations asynchrones

Futurs: example C++11

```
std::string hello(std::string name)
{
    return "my name is " + name;
}

int main(int argc, char **argv)
{
    std::future<string> a1 = std::async(hello, "Riri");
    std::future<string> a2 = std::async(hello, "Fifi");
    a1.wait();           ← On attend qu'il termine
    a2.wait();
    std::cout << a1.get() << std::cout; ← on récupère le résultat
    std::cout << a2.get() << std::cout;
}
```

Assurer un ordre avec des variables condition (solution 1)

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

Assurer un ordre avec des variables condition (solution 2)

```
std::queue<std::PcoConditionVariable*> conditionsList;
int currentId = 0;
int nbWaiting = 0;

void myFunction() {
    mutex.lock();
    if (!myCondition) {
        nbWaiting ++;
        PcoConditionVariable *variable = new PcoConditionVariable();
        conditionsList.emplace(variable);
        variable->wait(&mutex);
        delete variable;
    }
    mutex.unlock();
}

void anotherFunction() {
    mutex.lock();
    if (nbWaiting > 0) {
        conditionsList.front()->notifyOne();
        conditionsList.pop();
        nbWaiting --;
    }
    mutex.unlock();
}
```


Conclusion

Conclusion

- La concurrence se retrouve dans *tous* les logiciels
- Les mécanismes de base seront proches de ceux présentés en PCO
- Attention à les utiliser correctement
- Exploitez les mécanismes avancés disponibles