L3 – Cours de Système – Travaux dirigés

# Les conditions — Présentation et implémentation dans Nachos autour d'un exemple

EMMANUEL AGULLO, EDDY CARON, NICOLAS VEYRAT CHARVILLON

On considère un programme multithreadé utilisant une variable globale `compteur`. Les accès à cette variable sont protégés par un lock `lock`. Les threads peuvent incrémenter le compteur. Ils peuvent également le décrémenter, mais uniquement s'il est supérieur à 3. Sinon, ils doivent attendre qu'il soit incrémenté par d'autres threads jusqu'à être supérieur à 3.

```
...
/* Incrémentation */
lock->Acquire();
compteur++;
lock->Release();
...

...
/* Décrémentation, dans un autre thread, à un autre moment */
while ( compteur <= 3 );
lock->Acquire();
compteur--;
lock->Release();
```

Ce code est faux car un changement de contexte entre la boucle `while` et `Acquire` peut changer `compteur`. Il faut donc entrer la boucle `while` dans la section critique.

```
...
/* Décrémentation */
lock->Acquire();
while ( compteur <= 3 );
compteur--;
lock->Release();
...
```

Ce code est également faux car `lock` n'est jamais relâché entre les différentes itérations de la boucle `while`. Le thread va donc boucler indéfiniment et ne jamais relâcher `lock`, ce qui va probablement bloquer tous les autres threads voulant accéder à `compteur`. On relâche donc `lock` entre les itérations de `while`.

```
...
/* Décrémentation */
label:
lock->Acquire();
```

```
  if ( compteur <= 3 ) {
    lock->Release();
    goto label;
  }
  compteur--;
  lock->Release();
  ...
```

Ce code marche, enfin ! Mais il utilise un `goto`, ce qui est mal :-).

Le vrai problème vient de la boucle d'attente active qui consomme du temps processeur. De plus, les nombreux appels à `Acquire` et `Release` désactivent les interruptions, ce qui réduit la réactivité du système. L'idéal serait donc de pouvoir endormir le thread jusqu'à ce que le compteur ait été incrémenté.

```
  ...
  /* Décrémentation */
  lock->Acquire();
  if ( compteur <= 3 ) {
    < Je préviens tout le monde que je vais dormir et que
      je veux être réveillé quand compteur sera incrémenté >
    lock->Release();
    currentThread->Sleep();
    lock->Acquire();
  }
  compteur--;
  lock->Release();
  ...
```

On définit une classe `Condition` qui implémente ce modèle. On lui associe une liste de threads en attente qui seront réveillés par les threads incrémentant `compteur`.

```
class Condition {
  List *queue;
};
```

Un thread attendant que le compteur soit incrémenté va appeler la fonction `Wait` qui l'ajoute à la queue des threads en attente, puis l'endort. Cette fonction doit évidemment relâcher `lock` avant de dormir.

```
void Condition::Wait(Lock *lock) {
  queue->Append(currentThread);
  lock->Release();
  currentThread->Sleep();
  lock->Acquire();
}
```

Une fonction `Signal` est utilisée pour que les threads modifiant compteur puissent réveiller les threads en attente pour leur signaler que compteur a été incrémenté.

```
void Condition::Signal() {
  Thread *sleepingThread = (Thread*)queue->Remove();
  scheduler->ReadyToRun(sleepingThread);
}
```

Le modèle initial devient alors :

```
  ...
  /* Incrémentation */
  lock->Acquire();
  compteur++;
  condition->Signal();
  lock->Release();
  ...

  ...
  /* Décrémentation */
  lock->Acquire();
  if ( compteur <= 3 )
    condition->Wait(lock);
  compteur--;
  lock->Release();
```

Ce code n'est pas tout à fait correct car un thread peut incrémenter `compteur` sans pour autant le rendre supérieur à 3. On reteste donc la condition à la sortie de `Wait`.

```
  /* Décrémentation */
  lock->Acquire();
  while ( compteur <= 3 )
    condition->Wait(lock);
  compteur--;
  lock->Release();
```

Il est possible que `Signal` soit appelé sans qu'aucun thread ne soit en attente. On doit donc tester que `Remove` a bien obtenu un thread avant de le réordonnancer.

De plus, plusieurs appels concurrents à `Signal` ou `Wait` peuvent provoquer des accès concurrents à `queue`. On peut envisager d'utiliser également `lock` pour protéger ces accès. Mais cela peut poser des problèmes si on veut pouvoir appeler `Signal` sans être forcément dans la section critique. Il vaut donc mieux protéger ces accès avec un autre `Lock` (celui-là sera mis directement dans la classe `Condition`), voire même en désactivant les interruptions dans `Signal` et dans `Wait`.

```
void Condition::Signal() {
  Thread *sleepingThread;
  IntStatus oldLevel = interrupt->SetLevel (IntOff);
  sleepingThread = (Thread*)queue->Remove();
  if ( sleepingThread )
    scheduler->ReadyToRun(sleepingThread);
  (void) interrupt->SetLevel (oldLevel);
}
```

Il faut noter que ce modèle doit être scrupuleusement respecté. Le programme appelant `Wait` devra donc avoir acquis le `lock`. Notez que la désactivation des interruptions y était en fait imposée par la méthode `Sleep` (voir dans son code).

```
void Condition::Wait(Lock *lock) {
  IntStatus oldLevel;
  ASSERT(lock->isHeldByCurrentThread());
  oldLevel = interrupt->SetLevel (IntOff);
  queue->Append(currentThread);
  lock->Release();
  currentThread->Sleep();
  lock->Acquire();
  (void) interrupt->SetLevel (oldLevel);
}
```

La condition testée peut évidemment prendre n'importe quelle forme (tests sur des entiers, des pointeurs, utiliser des fonctions, ...). La classe `Condition` n'a pas de lien direct avec la condition en elle-même. La classe sert uniquement à fournir un modèle de programmation utilisable dans le cas où le test de la condition est intrinsèquement lié à une section critique.

Ce modèle peut-être par exemple utilisé pour implémenter un modèle producteur consommateur avec file de produits. Les producteurs ajoutent des produits à la file tandis que les consommateurs attendent que la file soit non-vide pour y prendre un produit.