Introduction aux variables de condition

Orestis Malaspinas, Steven Liatti

1 Les variables de condition

Le contenu de ce chapitre est basé sur l'excellent livre de R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau [1].

Après avoir discuté les verrous, et leur construction, nous allons discuter d'une autre structure qui est nécessaire pour construire des applications concurrentes: les **variables de conditions**. Les variables de condition sont utilisées lorsque qu'un fil d'exécution doit d'abord vérifier une condition avant de continuer.

Le problème peut s'illustrer de la façon suivante

```
void *foo() {
    printf("Nous y sommes.\n");
    // On veut signaler qu'on a terminé:
    // mais comment faire?
    return NULL;
}

int main() {
    printf("Début du programme.\n");

    pthread_t tid;
    pthread_create(&tid, NULL, foo, NULL);
    // On aimerait attendre sur la fonction foo
    // mais comment faire?
    printf("Fin du programme.\n");

    return EXIT_SUCCESS;
}
```

Une solution serait d'utiliser une variable partagée entre le thread principal et le thread enfant comme dans l'exemple ci-dessous

```
bool done = false;

void *foo() {
    printf("Nous y sommes.\n");
    done = true;
```

1. R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ed. 0.91, (2015).

```
    return NULL;
}

int main() {
    printf("Début du programme.\n");

    pthread_t tid;
    pthread_create(&tid, NULL, foo, NULL);
    while (!done) {
        // do nothing, just wait
    }
    printf("Fin du programme.\n");

    return EXIT_SUCCESS;
}
```

Cette solution, bien que très simple, est très inefficace. En effet, le thread principal va tourner dans la boucle **while** et gaspiller des ressources pour rien. Par ailleurs, il est assez simple d'imaginer des situations où on peut produire du code qui sera simplement faux.

1.1 Définition

Pour pallier aux problèmes d'inefficacité et de justesse que nous venons de discuter, on utilise les **variables de condition**. Une variable de condition est une file d'attente, dans laquelle les fils d'exécution peuvent se mettre lorsqu'une condition n'est pas satisfaite. Ils y attendront que la dite condition soit remplie. Un autre thread peut alors, lorsque l'état de la condition change, réveiller un ou plusieurs fils qui sont en attente pour leur permettre de continuer. Il va **signaler** aux threads en attente de se réveiller.

Les variables de conditions dans la librairie POSIX se déclare via le type `pthread_cond_t`. Une telle variable se manipule via les fonctions [2]

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

Ces deux fonctions, sont les appels fondamentaux des variables de condition. La fonction `cond_signal()` (on va utiliser ici les raccourcis `cond_wait()` et `cond_signal()` car je suis flemmard) est utilisée pour signaler qu'une variable a changé d'état et réveille un thread. La fonction `cond_wait()` a la responsabilité de mettre le thread qui l'appelle en attente. Il faut noter que cette fonction prend un `mutex` en paramètre. Par ailleurs, `cond_wait()` fait l'hypothèse que le `mutex` est verrouillé et a la responsabilité de libérer le verrou et de mettre le thread en attente. Puis, lorsque le dit thread doit se réveiller (parce qu'un autre fil d'exécution le lui a signalé) `cond_wait()` doit acquérir le verrou à nouveau et continuer l'exécution du thread. Cette partie est quelque peu complexe. Pour la comprendre considérons le code ci-dessous

2. Il ne faut pas oublier qu'il faut également initialiser les variables de condition et éventuellement les détruire aussi.

```
bool done = false;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void cond_wait() {
    pthread_mutex_lock(&mutex);
    while (!done) {
        pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
}

void cond_signal() {
    pthread_mutex_lock(&mutex);
    done = true;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}

void *foo() {
    printf("Nous y sommes.\n");

    cond_signal();

    return NULL;
}

int main() {
    printf("Début du programme.\n");

    pthread_t tid;
    pthread_create(&tid, NULL, foo, NULL);

    cond_wait();

    printf("Fin du programme.\n");
    return EXIT_SUCCESS;
}
```

Il y a deux cas distincts:

1. Le thread principal crée un thread enfant et continue son exécution. Il acquière le `mutex` et est mis en attente lorsqu'il appelle `cond_wait()` et libère le verrou. A un moment ou un autre, le fil enfant s'exécutera. En appelant la fonction `cond_signal()`, il verrouillera le `mutex`, assignera la valeur `true` à la variable `done` et appellera `pthread_cond_signal()`, ce qui aura pour effet de réveiller le thread principal. Le thread principal sera réveillé et aura le `mutex` dans un état verrouillé, sortira de la boucle **`while`** déverrouillera le `mutex` et terminera son exécution.
2. Le thread enfant est exécuté immédiatement, il appelle `cond_signal()`

et il modifie `done`. Comme il n'y a pas de thread en attente, rien ne se passe. Le thread principal peut donc finir son exécution tranquillement.

Remarque 1

Comme nous allons le voir plus bas, il est très important d'utiliser la boucle `while` bien qu'ici cela ne soit pas strictement nécessaire.

Il se pourrait que vous soyez tenté · e · s de modifier quelque peu l'implémentation ci-dessus. N'en faites rien, cela pourrait causer des dommages irréversibles à vos codes. Changeons un peu les différentes parties du programme pour voir ce qui peut se passer. Si nous remplaçons les fonctions `cond_wait()` et `cond_signal()` par les codes ci-dessous (il va y avoir plusieurs exemples faux)

```
void cond_wait() {
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);
}

void cond_signal() {
    pthread_mutex_lock(&mutex);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Question 1

Pourquoi cette façon de faire est-elle problématique?

Dans ce cas, on utilise pas la variable `done`. Hors, si on imagine qu'on est dans le cas où le thread enfant est exécuté immédiatement, il va effectuer la signalisation et retourner immédiatement. Le thread principal lui va ensuite attendre indéfiniment, car plus aucun autre thread ne lui signalera jamais de se réveiller.

L'exemple suivant est également faux. Nous nous débarrassons cette fois du verrou

```
void cond_wait() {
    if (!done) {
        pthread_cond_wait(&cond, &mutex);
    }
}

void cond_signal() {
    done = true;
```

```
    pthread_cond_signal(&cond);
}
```

Question 2

Pourquoi cette façon de faire est-elle également problématique?

Il y a ici un problème d'accès concurrent. Si le thread enfant appelle `cond_wait()` et voit que `done == false` il voudra se mettre en attente. Hors, si à ce moment précis il y a un changement de contexte et `cond_signal()` est appelé, il signalera la condition, mais comme aucun fil n'est en attente, ce signal sera perdu. Le thread enfant sera à un moment ou un autre ré-ordonnancé et sera mis en attente. Comme aucun autre signal ne sera jamais émis il restera endormi à jamais.

De ces deux mauvais exemples, on peut tirer un enseignement. Même si cela n'est pas toujours nécessaire, il faut **toujours** détenir le verrou avant de signaler une condition. A l'inverse, il est toujours nécessaire d'avoir un verrou lors de l'appel de `pthread_cond_wait()`. La fonction `pthread_cond_wait()` présuppose que le verrou est détenu lors de son appel. La règle générale est finalement: quoi qu'il arrive avant d'attendre ou de signaler acquérir le verrou (et le libérer ensuite).

1.2 Le problème producteurs/consommateurs

Le problème des **producteurs/consommateurs** a été à l'origine du développement des variables de condition par Dijkstra. On peut imaginer un ou plusieurs fils d'exécution produisant des données et les plaçant dans une mémoire tampon (buffer). Le ou les consommateurs vont chercher les données dans la mémoire tampon et les utilisent.

Un exemple typique est la fonction `pipe` d'un programme dans un autre dans les systèmes UNIX:

```
grep hop file.txt | wc -l
```

Ici un processus exécute la fonction `grep`, qui écrit les lignes du fichier `file.txt` où `hop` est présent sur la sortie standard (enfin c'est ce que `grep` croit). Le système d'exploitation redirige en fait la sortie dans un "tuyau" (*pipe*) UNIX. L'autre côté du *pipe* est connecté à l'entrée du processus `wc` qui compte les lignes dans le flot de données en entrée et affiche le résultat. On a donc que `grep` produit des données et `wc` les consomme avec un buffer entre deux.

Question 3

Pouvez-vous imaginer une autre application de ce type?

Comme la mémoire tampon est une ressource partagée, il faut trouver un mécanisme de synchronisation pour éviter les accès concurrents.

Pour simplifier, représentons le buffer par un simple entier [3]. On a ensuite deux fonctions:

— la fonction `put()` qui va insérer une valeur dans la mémoire tampon.
— la fonction `get()` qui va récupérer une valeur de la mémoire tampon.

Finalement, on a également besoin de savoir combien de données sont stockées dans le buffer. Pour cela on utilise un simple compteur.

```
int buffer;
int counter = 0; // le compteur

void put(int value) {
    assert(counter == 0 && "trying to put into a full buffer");
    counter = 1;
    buffer = value;
}

int get() {
    assert(counter == 1 && "trying to get from empty buffer");
    counter = 0;
    return buffer;
}
```

La fonction `put()` suppose que le **`buffer`** ne contient rien (`counter == 0`) et vérifie l'état avec une `assert()`. Ensuite elle incrémente la valeur du compteur à 1 et assigne la valeur `value` au `buffer`. A l'inverse la fonction `get()` fait l'hypothèse que le **`buffer`** contient une valeur (`counter == 1`) et vérifie également cet état avec une `assert()`.

A présent, il faut des fonctions qui auront la sagesse nécessaire pour appeler les fonctions `put()` et `get()` au moments opportuns (quand **`counter`** a la bonne valeur, zéro ou un respectivement). Grâce au design astucieux de `put()` et `get()` on saura très vite si on fait quelque chose de faux, car les assertions vont faire planter le programme.

Ces actions seront effectuées par deux types de threads: les threads **producteurs** et les threads **consommateurs**. Pour simplifier, écrivons d'abord une version complètement fausse et naive mais qui illustre ce que ferait un thread producteur qui met `loops` valeurs dans le buffer et un thread consommateur qui récupère à l'infini les valeurs stockées dans le buffer.

```
// rien n'est initialisé c'est normal c'est une illustration
// complètement absurde
void *producer(void *arg) {
    int loops = (int)*arg;
    for (int i = 0; i < loops; ++i) {
        put(i);
    }
}
```

3. Cela pourrait aussi être un pointeur vers une structure de donnée plus complexe.

```
void *consumer(void *arg) {
    while (1) {
        int i = get();
        printf("%d\n", i);
    }
}
```

Il est évident que ce code ne peut pas fonctionner. Les `assert()` passeraient le temps à tout arrêter. On a des sections critiques dans `put()` et `get()` qui ne sont pas du tout gérées. Mais vous voyez l'idée de ce qu'on essaie de faire. Juste utiliser un verrou pour protéger, `put()` et `get()` comme on l'a fait jusqu'ici ne peut pas fonctionner.

Question 4

Pourquoi?

Il nous faut en fait un moyen de prévenir chacun des threads de l'état du buffer afin qu'il puisse effectuer une action appropriée.

1.2.1 Une solution, fausse

La solution la plus simple serait d'ajouter une variable de condition et son verrou associé. Un code du genre de celui se trouvant ci-dessous par exemple

```
int loops = 10; // une initialisation au hasard
pthread_cond_t cond;
pthread_mutex_t mutex

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        if (count == 1) {
            pthread_cond_wait(&cond, &mutex);
        }
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(*void arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        if (count == 0) {
            pthread_cond_wait(&cond, &mutex);
        }
        int tmp = get();
        pthread_cond_signal(&cond);
```

```
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Si nous avons un thread producteur et un thread consommateur ce code fonctionne. Par contre, si nous ajoutons plus de threads cela ne fonctionne plus.

Question 5

Que se passe-t-il dans le cas où nous avons deux threads consommateurs pour un producteur?

Le premier problème est relié à l'instruction `if` avant l'appel à `wait()`. Le second est relié à l'utilisation d'une unique variable de condition. Voyons à présent ce qui peut se passer.

Soient les threads consommateurs T_{c_1} et T_{c_2} et le thread producteur T_p. Supposons que T_{c_1} s'exécute. Il acquière le verrou et vérifie s'il y a quelque chose à consommer dans le frigo... euh... dans le buffer. Comme il n'y a rien il se met en attente et libère le verrou. Ensuite le thread T_p s'exécute. Il acquière le verrou et vérifie si le buffer est vide. Comme il l'est il peut y déposer une valeur. Il va ensuite signaler que le buffer est rempli. Cela va mettre T_{c_1} en état de réveil et prêt à être exécuté (il ne s'exécute pas encore). Finalement pour T_p, il se rend compte que le buffer est plein et va se mettre en sommeil. A ce moment précis, T_{c_2} est ordonnancé. Il acquière le verrou, lit la valeur dans le buffer, signale qu'il a lu et libère le verrou. A ce moment précis, si T_{c_1} est ordonnancé, il va acquérir le verrou et continuer son exécution là où elle c'était arrêtée. Il va donc tenter de lire à son tour dans le buffer, mais celui-ci sera vide! L'assertion dans `get()` nous sautera au visage et arrêtera l'exécution. Au moins, nous sommes sauvés d'un comportement erroné que nous croyions correct.

Pour prévenir cette erreur, nous aurions dû empêcher T_{c_1} de tenter de consommer une valeur déjà consommée. En fait le signal envoyé par T_p doit être considéré comme une indication que quelque chose a changé. Il n'y a en revanche aucune garantie que cela sera toujours le cas lorsque le thread mis en sommeil sera réveillé. La solution ici est de remplacer le `if` par un `while`. De cette façon T_{c_1} aurait vérifié que la condition a bien changé après son réveil et avoir acquis le verrou. Ainsi il n'aurait pas essayé de consommer une valeur déjà consommée.

Règle 1

Toujours utiliser une boucle `while` et jamais de `if`. [4]

Voilà un premier problème de réglé, mais nous ne sommes pas sortis d'affaire. En effet, un autre problème existe. Imaginons que T_{c_1} et T_{c_2} s'exécutent en premier

4. C'est pas toujours nécessaire, mais ça ne fait pas de mal!

et sont mis en sommeil. T_p s'exécute alors, met une valeur dans le buffer et réveille un thread, disons T_{c_1}. T_p refait un tour de boucle, libère et réacquière le verrou et réalise que le buffer est plein. Il se met en sommeil. T_{c_1} peut s'exécuter (alors que T_{c_2} et T_p sont endormis). Il refait un tour dans la boucle **`while`**, trouve le buffer plein et le vide. Il signale qu'il a fini et va se rendormir. Si à présent c'est T_{c_2} qui se réveille (on a aucune garantie sur l'ordre de l'exécution), il va trouver le buffer vide et se rendormir. On a donc les trois threads en sommeil avec aucun moyen de les réveiller (pas de prince · esse charmant · e, ni rien)!

On a besoin d'un autre signal qui soit dirigé: un consommateur doit réveiller un producteur et vice-versa. Pour corriger ce bug, nous pouvons simplement utiliser **deux** variables de condition: une relié au remplissage du buffer, l'autre à sa vidange comme dans le code ci-dessous.

```
pthread_cond_t fill, empty; // pas oublier d'initialiser hein
pthread_mutex_t mutex;

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == 1) {
            pthread_cond_wait(&empty, &mutex);
        }
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}

vois *consumer(*void arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == 0) {
            pthread_cond_wait(&fill, &mutex);
        }
        int tmp = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Dans ce code, on voit que les threads producteurs attendent sur la condition `empty` et signalent la condition `fill`, alors que les threads consommateurs attendent sur la condition `fill` et signalent la condition `empty`.

1.2.2 Aller plus haut

On vient de voir comment résoudre le problème producteurs/consommateurs pour un buffer unique. Cette solution est généralisable pour permettre des applications plus concurrentes et efficaces. Pour ce faire, on va créer un buffer

avec plus d'espaces. Ainsi il sera possible de produire plus de valeurs et de les consommer avant que les threads se mettent en sommeil.

Pour ce faire, nous créons un buffer qui sera cette fois un tableau statique. On doit garder une trace du taux de remplissage, ainsi que de la position à laquelle nous devons lire. On modifie donc les fonctions `put()` et `get()` de la façon suivante

```
#define MAX 100;

int buffer[MAX];
int fill_pos = 0;
int read_pos = 0;
int count = 0;

void put(int value) {
    buffer[fill_pos] = value;
    fill_pos = (fill_pos + 1) % MAX;
    count += 1;
}

int get() {
    int tmp = buffer[read_pos];
    read_pos = (read_pos + 1) % MAX;
    count -= 1;
    return tmp;
}
```

Une toute petite adaptation à la vérification nécessaire avant le `wait()` ou `signal()` et le tour est joué

```
pthread_cond_t empty, fill;
pthread_mutex_t mutex;

void *producer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == MAX) {
            pthread_cond_wait(&empty, &mutex);
        }
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    for (int i = 0; i < loops; ++i) {
        pthread_mutex_lock(&mutex);
        while (count == 0) {
            pthread_cond_wait(&fill, &mutex);
```

```
        }
        int value = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", value);
    }
}
```

On voit que le producteur ne se met en sommeil que lorsque le buffer est plein et à l'inverse le consommateur n'appelle `wait()` que lorsque le buffer est vide.

1.3 Signalisation globale

Dans certains cas, on aimerait être un peu plus large lorsqu'on réveille des threads. En particulier, on aimerait pouvoir réveiller tous les threads endormis à la fois. Afin d'illustrer le problème considérons l'exemple d'une application d'allocation de mémoire sur la pile. La pile a une taille maximale `MAX_HEAP_SIZE`. Nos threads vont allouer un certain nombre d'octets `void *allocate(int size)` et libérer un certain nombre de bytes de la mémoire `void free(void *ptr, int size)`. Lorsque la mémoire est pleine, les fils d'exécution qui sont en charge de l'allocation doivent se mettre en sommeil. Lorsque de la mémoire se libère il faut signaler que de la place est à nouveau disponible. Une approche simple serait le code suivant

```
int avail_bytes = MAX_HEAP_SIZE; // nombre d'octets disponibles

// comme d'habitude, une variable de condition et son verrou
pthread_cont_t cond;
pthread_mutex_t mutex;

void *allocate(int size) {
    pthread_mutex_lock(&mutex);
    while (avail_bytes < size) { // on doit avoir assez de mémoire
        pthread_cond_wait(&cond, &mutex);
    }
    void *ptr = ...; // on récupère la mémoire depuis le tas
    avail_bytes -= size;
    pthread_mutex_unlock(&mutex);
    return ptr;
}

void free(void *ptr, int size) {
    pthread_mutex_lock(&mutex);
    avail_bytes += size;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Question 6

Ce code a un problème. Lequel?

Imaginons qu'à un moment donné on ait plus de place en mémoire (`avail_bytes == 0`). Un premier thread T_1 appelle `allocate(100)` et un deuxième thread T_2 appelle `allocate(10)`. Chacun de ces threads va appeler `wait()` et se mettre en sommeil. A présent, un troisième thread appelle `free(40)`, il libère 40 bytes et signale qu'il y a de la mémoire disponible. Si T_1 est réveillé, il retournera en sommeil car il n'y a pas la place suffisante pour allouer et le programme se retrouve à ne rien faire alors qu'il pourrait. Il suffisait de réveiller T_2 bon sang! La solution la plus simple ici est de réveiller **tous** les threads grâce à la fonction `pthread_cond_broadcast()`

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Tous les threads attendant après l'appel de `wait()` seront réveillés, ils vérifieront le nombre de bytes disponibles et se remettront en sommeil immédiatement si l'espace n'est pas disponible.