Les bugs

Orestis Malaspinas, Steven Liatti

1 Les bugs les plus courants

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

Dans ce chapitre, nous allons discuter certains des bugs les plus fréquents qu'on retrouve dans les codes concurrents.

1.1 Quels sont les bugs possibles

Un certain nombre d'études ont été faites sur les bugs les plus fréquents lors de l'écriture de codes concurrents. Ce que nous allons raconter ici se base sur: *Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics*, par Shan Lu, Soyeon Park, Eunsoo Seo, et Yuanyuan Zhou. **ASPLOS 2008**, Mars 2008, Seattle, Washington. Cette étude est basée sur un certain nombre de logiciels open source populaires (MySQL, OpenOffice, Apache, et Mozilla le butineur). L'avantage de ce genre de codes est que des gens externes peuvent voir et analyser les modifications faites et ainsi examiner en particulier les bugs liés à la concurrence. Il est ainsi possible de comprendre quels types d'erreurs les gens font et tenter d'empêcher le répétition.

Il ressort qu'une grande partie des bugs sont des interblocages (ou deadlocks). En effet, sur 74 bugs de concurrence, 31 étaient dûs à des deadlocks et les 43 restants autre chose [2].

On va d'abord commencer par étudier les bugs "non-deadlocks", car ils sont plus nombreux et "plus simples" à analyser. Puis nous passerons aux bugs "deadlocks".

1.2 Les bugs pas dû à de l'interblocage

Selon l'étude de *Lu et al.* deux classes de bugs principales: la **violation d'atomicité** et la **violation d'ordre**.

1.2.1 La violation d'atomicité

La violation d'atomicité est quand un thread écrit et un autre lit des données partagées sans que celles-ci soient protégées par une primitive d'exclusion mutuelle. Par exemple le pseudo-c suivant

1. R. H. Arpaci-Dusseau et A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books, ed. 0.91, (2015).
2. Après un calcul savant on voit que $31/74 \cdot 100 \cong 42\%$ des bugs sont dû à des interblocages. 42... coincidence je ne crois pas.

```
void *t1() {
    if (thd->info) {
        // read thd->info
    }
}

void *t2() {
    thd->info = NULL;
}
```

Ici `t1()` vérifie si `thd->info` est non-`NULL` avant de lire sa valeur et de faire des opérations avec éventuellement. Dans le même temps `t2()` assigne `thd->info` à `NULL`. Si entre la vérification de `thd->info` et son utilisation `t2()` prend la main et fait son office, `t1()` va ensuite tenter de déréférencer un pointeur `NULL` ce qui va causer un plantage magistral.

Ce genre de bugs se corrige assez simplement.

Question 1

Comment?

Dans ce cas il suffit bien souvent d'insérer un verrou et de protéger la section critique où on assigne `ths->info` à `NULL` et l'endroit où on le lit.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *t1() {
    pthread_mutex_lock(&mutex);
    if (thd->info) {
        // read thd->info
    }
    pthread_mutex_unlock(&mutex);
}

void *t2() {
    pthread_mutex_lock(&mutex);
    thd->info = NULL;
    pthread_mutex_unlock(&mutex);
}
```

1.2.2 Les bugs de violation d'ordre

Une violation d'ordre se produit lorsque l'ordre dans lequel deux accès mémoires sont inversés: on veut que A soit toujours exécuté avant B, mais que cela n'est pas garantit à l'exécution.

Un exemple de ce genre d'exécution (en pseudo-c) serait:

```
// initialize something here
void *t1() {
    // do things
    thread = create_thread_do_things_and_return(other_thread, ...);
    // do more things
}

void *t2() {
    // do things
    state = thread->state;
    // do even more things
}
```

Ici, dans `t1()` on crée un thread effectue un certain nombre d'opérations et on retourne une valeur stockée dans `thread`. Dans le thread `t2()`, on suppose que `thread` est initialisé et on l'utilise. En fait, comme nous l'avons déjà vu, au moment où `t2()` arrive à la ligne `state = thread->state`, la variable `thread` n'a pas forcément de valeur, car la fonction `create_thread_do_things_and_return()` n'a pas encore retourné. L'ordre dans lequel on aimerait que soient exécutées les instructions n'est pas celui qui est imposé à l'exécution.

Question 2

Comment corriger ce code?

Une façon de corriger ce code est d'utiliser une variable de condition et signaler à `t2()` que `thread` contient une valeur

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

// initialize something here
void *t1() {
    // do things
    thread = create_thread_do_things_and_return(other_thread, ...);

    // we signal the initialization
    pthread_mutex_lock(&mutex);
    done = true;
    pthread_mutex_signal(&cond);
    pthread_mutex_unlock(&mutex);

    // do more things
}

void *t2() {
```

```
    // do things

    // we wait on the initialization
    pthread_mutex_lock(&mutex);
    done = true;
    while (!ini_done) {
        pthread_mutex_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);

    state = thread->state;

    // do even more things
}
```

On a donc forcé l'ordre d'exécution de l'initialisation de `thread` et de son utilisation à l'aide d'un booléen, d'une variable de condition et d'un verrou.

1.3 Les bugs d'interblocage

Nous avons déjà vu un exemple typique de bug d'interblocage. Ce genre de bug peut survenir lorsqu'on a deux threads qui vérouillent deux verrous dans des ordres différents. Imaginons la situation où deux threads T_1 et T_2 verrouillent/déverrouillent deux verrous V_1 et V_2.

T_1	T_2
`pthread_mutex_lock(&V_1)`	`pthread_mutex_lock(&V_2)`
`pthread_mutex_lock(&V_2)`	`pthread_mutex_lock(&V_1)`

Le tableau ci-dessus montre une situation où un interblocage **peut** se produire. En effet, si T_1 verrouille V_1 et qu'un changement de contexte se produit et T_2 verrouille V_2, les deux fils d'exécution attendent l'un sur l'autre et un interblocage survient.

Une façon de voir cette situation est via le schéma de la fig. 1. On constate sur le schéma que les demandent de verrouillage et l'acquisition des verrous forment un cycle. Ce genre de cycle est en général l'indication qu'un interblocage est possible.

Dans ce qui suit, nous allons voir quelles sont les conditions qui doivent être réunies pour l'existence des interblocages, et comment essayer les éviter.

1.3.1 Pourquoi surviennent les interblocages?

Les constructions amenant à des interblocages peuvent être beaucoup plus complexes que dans l'exemple que nous venons de voir. En fait, dans de grands projets, des dépendances très complexes entre différentes parties du programme peuvent survenir.

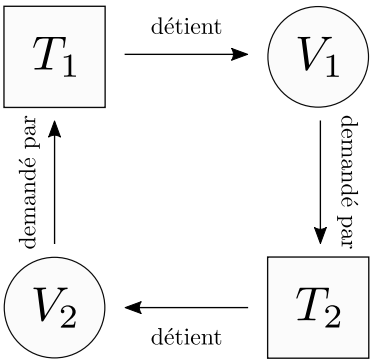

FIGURE 1 – Schéma de l'interblocage de deux thread verrouillant et tentant de verrouiller deux verrous dans un ordre différent. On voit un cycle se former ce qui peut indiquer la présence d'un interblocage potentiel.

Une autre raison est **l'encapsulation**. Le développement logiciel encourage l'encapsulation pour cacher au maximum les détails d'implémentation et ainsi construire les logiciels aussi modulaires et faciles à maintenir que possible. Hélas, cela a aussi pour effet de cacher certaines parties critiques comme les verrous.

Imaginons la situation où nous avons une fonction, `add_vectors_in_place(v1, v2)`, qui additionne les valeurs de deux tableaux de nombres `v1` et `v2`. Afin que cette fonction soit thread-safe il faut verrouiller `v1` et `v2`. Si le verrou est acquis dans l'ordre `v1` puis `v2`, et que dans un autre thread on appelle la même fonction avec les arguments inversés, `add_vectors_in_place(v2, v1)`, on peut se retrouver dans une situation d'interblocage sans qu'on ait commis une erreur explicite!

1.3.2 Conditions pour l'interblocage

Il y a quatre conditions distinctes pour l'apparition d'un interblocage:

- *L'exclusion mutuelle*: plsuieurs threads essaient d'avoir le contrôle exclusif d'une ressource (un thread acquière un verrou).
- *Hold-and-wait*: plusieurs threads possèdent des ressources (ont déjà acquis un verrou) et attendent d'en obtenir d'autres (attendent sur un autre verrou par exemple).
- *Pas de préemption*: Les ressources (un verrou par exemple) ne peuvent pas être libérées des fils qui les possèdent.
- *Une attente circulaire*: Il existe une chaîne de fils d'exécution telle que chaque fil possède une ressource qui est attendu pour verrouillage par le prochain thread dans la chaîne.

Si une de ces condition n'est pas remplie, il ne peut y avoir de deadlock. On va voir maintenant quelques techniques pour éviter les interblocages en évitant chacune des ces conditions.

1.3.3 L'attente circulaire

La façon la plus courante d'éviter les interblocages est d'écrire le verrouillage en évitant les attentes circulaires. La façon la plus simple est d'utilser un **ordre**

global pour l'acquisition des verrous. Par exemple, si nous avons deux verrous dans notre code, V_1 et V_2, les verrous seront toujours acquis dans l'ordre V_1, puis V_2 et jamais l'inverse. De cette façon, nous garantissons qu'il n'y aura jamais d'attente circulaire et donc jamais d'interblocage.

Il est commun que dans des systèmes plus complexes, il y a plus de verrous et donc un tel ordre ne peut pas toujours être garantit. On utilise alors un **ordre partiel**. Un exemple d'un tel ordre partiel peut se lire dans les commentaires du code suivant. Dans cet exemple, il y a un grand nombre d'acquisition de verrous, qui vont du peu complexe (deux verrous) à de beaucoup plus complexes, allant jusqu'à dix verrous différents. Il est évident, que ce genre de documentation est très utile pour le bon fonctionnement du code. Il est également certain que c'est très difficile pour n'importe quel programmeur ne connaissant pas parfaitement la structure d'un code de ce genre d'éviter le bugs, même s'il fait très attention. Une façon un peu plus systématique de procéder est d'utiliser les adresses des verrous pour les ordonner. Imaginons une fonction prenant deux verrous en argument, `foo(mutex_t *m1, mutex_t *m2)`. Si l'ordre d'acquisition des verrous dépend de l'ordre dans lequel on passe les arguments à la fonction, un deadlock est très vite arrivé si on appelle une fois `foo(m1, m2)` et une fois `foo(m2, m1)`. Si en revanche on utilise les adresses des verrous comme dans le code ci-dessous, on s'affranchit de ce problème

```
void foo(mutex_t *m1, mutex_t m2) {
    if (m1 > m2) { // on verrouille d'abord le verrou avec l'adresse la plus élevée
        pthread_mutex_lock(m1);
        pthread_mutex_lock(m2);
    } else if (m2 > m1) {
        pthread_mutex_lock(m2);
        pthread_mutex_lock(m1);
    } else {
        assert(false && "Les deux verrous ne peuvent pa avoir la même adresse");
    }
    // do stuff
}
```

1.3.4 Hold-and-wait

Le condition de détenir un verrou et attendre pour en acquérir un autre peut être prévenue en acquérant les verrous de façon atomique: en mettant un verrou autour de la séquence de verrous à acquérir.

```
mutex_t global, v1, v2;

pthread_mutex_lock(&global); // début de l'acquisition des verrous
pthread_mutex_lock(&v1);
pthread_mutex_lock(&v2);
// ...
pthread_mutex_unlock(&global); // fin
```

De cette façon, comme le verrou `global` protège l'acquisition de tous les autres verrous, l'interblocage est certainement évité. Si un autre thread tentait d'acquérir les verrous `v1` et `v2` dans un ordre différent, cela ne poserait pas problème car il

devrait attendre la libération du verrou `global`.

Cette méthode n'est pas idéale. En effet, l'encapsulation possible de l'acquisition des verrous, nous oblige à connaître en détail la structure et l'ordre de l'acquisition des verrous pour pouvoir mettre ce genre de solution en place. De plus, cette façon de faire pourrait diminuer de façon non négligeable la concurrence de notre application.

1.3.5 Pas de préemption

Les verrous sont en général vus comme détenus jusqu'à ce qu'ils soient déverrouillés (on ait appelé la fonction `unlock()`). Plusieurs problèmes peuvent apparaître lorsqu'on détient un verrou et qu'on attend d'en acquérir un autre. Une interface plus flexible peut nous aider dans ce cas. La fonction `pthread_mutex_trylock()` verrouille le verrou si possible et retourne `0` ou retourne un code d'erreur si le verrou est déjà acquis. On peut donc réessayer de verrouiller les verrous plus tard avec le code suivant

```
before:
    pthread_mutex_lock(&v1);
    if (pthread_mutex_trylock(&v2) != 0) { // si on peut pas verrouiller
        pthread_mutex_unlock(&v1); // on déverouille
        goto before; // on retourne à la première tentative d'acquisition
    }
```

De cette façon, même si un fil essaie de verrouiller `v2` avant `v1` on a pas de problème d'interblocage. Néanmoins, un autre problème peut survenir, celui de **l'interblocage actif** (ou **livelock**). Imaginons que deux threads tentent en même temps cette séquence d'acquisition et ratent dans leurs tentatives d'acquisition. Le programme ne serait pas bloqué (contrairement à l'interblocage "passif", les fils s'exécuteraient toujours) mais serait dans une boucle infinie de tentatives d'acquisition. Pour se prémunir de ce problème on pourrait imaginer ajouter un délai aléatoire entre les tentatives d'acquisition pour minimiser les probabilités d'avoir un interblocage actif.

Cette méthode pour éviter l'interblocage n'est évidemment pas idéale, car encore une fois, l'encapsulation ne nous aide pas: le `goto` peut être **très** compliqué à implémenter... De plus si plus d'un verrou était acquis "en chemin", il faudrait également le déverrouiller après le `trylock()`.

1.3.6 Exclusion mutuelle

Finalement, on peut aussi essayer de se débarrasser de l'exclusion mutuelle. Cela peut être difficile à concevoir, mais à l'aide des instructions matériel, on peut construire des structure de données qui sont sans verrou explicite (lock-free).

On se rappelle qu'on a vu certaines instructions permettant de construire des verrous à attente active. On avait par exemple la fonction atomique `compute_and_exchange()` qui pouvait s'écrire de la façon suivante

```
int compare_and_exchange(int *addr, int expected, int new) {
    if (*adress == expected) {
        *adress = new;
```

```
        return 1; // succès
    }
    return 0; // erreur
}
```

A l'aide de cette fonction on pourrait construire une fonction qui incrémenterait une valeur, `value`, d'une certaine quantité, `amount` de façon atomique sans explicitement avoir à utiliser un verrou.

```
void atomic_increment(int *value, int amount) {
    do {
        int old = *value;
    } while (compare_and_exchange(value, old, old + amount) == 0);
}
```

De cette façon, comme nous n'utilisons pas de verrou, un interblocage "passif" ne peut pas se produire (on peut avoir un interblocage actif, mais ils sont beaucoup plus difficiles à obtenir).

On peut également considérer un exemple plus complexe: l'insertion en tête d'une liste.

```
typedef struct __node_t {
    int value;
    __node_t *next;
} node_t;

node_t *head;

void insert(int value) {
    node_t *node = malloc(sizeof(node_t));
    assert(node != NULL);
    node->value = value;
    node->next = head;
    head = node;
}
```

En utilisant un verrou, on ferait un `lock()` avant de modifier le pointeur `next` et un `unlock()` après avoir modifié `head`.

```
pthread_mutex_t list_lock = PTHREAD_MUTEX_INITIALIZER;

void insert(int value) {
    node_t *node = malloc(sizeof(node_t));  // on sait que malloc est thread safe
    assert(node != NULL);
    node->value = value;
    pthread_mutex_lock(&list_lock); // début de section critique
    node->next = head;
    head = node;
    pthread_mutex_unlock(&list_lock); // fin de section critique
}
```

Question 1.1

Comment modifier ce code pour avoir une insertion sans verrou?

De façon similaire à l'incrémentation, on peut modifier ce code en ajoutant un `compare_and_exchange()`.

```
pthread_mutex_t list_lock = PTHREAD_MUTEX_INITIALIZER;

void insert(int value) {
    node_t *node = malloc(sizeof(node_t));  // on sait que malloc est thread safe
    assert(node != NULL);
    node->value = value;
    do {
        node->next = head;
    } while (compare_and_exchange(&head, node->next, node) == 0);
}
```

Ce code essaie de façon atomique d'échanger la tête avec un noeud nouvellement créé. Il est possible qu'un autre thread ait modifié la tête pendant la création du noeud, et donc la condition de la boucle `do ... while` ne va pas être remplie et entraîner un nouveau tour de boucle.

Les autres fonctions pour manipuler des listes peuvent être bien plus complexes. Il existe une grande littérature sur le sujet des structures de données concurrentes qui fonctionnent sans verrous (les structures **lock-free** ou **wait-free**).

1.4 Éviter les interblocages avec le scheduling

Plutôt que de prévenir les interblocages on peut également les éviter totalement. Cela se fait en ayant une connaissance approfondie des mécanismes de verrouillage du programme. On peut ainsi ordonnancer les threads de façon appropriée "à la main".

Imaginons qu'on ait deux processeurs, P_1 et P_2, et quatre fils d'exécution, T_{1-4}, qui peuvent s'ordonnancer sur ces processeurs. Supposons que le thread T_1 tente de vérouiller V_1 et V_2 à un moment donné de l'exécution, T_2 également V_1 et V_2, T_3 seuelement V_2 et T_4 n'essaie d'acquérir aucun verrou.

On peut résumer cela dans le tableau suivant:

	T_1	T_2	T_3	T_4
V_1	oui	oui	non	non
V_2	oui	oui	oui	non

De ce tableau, on peut déduire que si T_1 et T_2 ne sont jamais ordonnancés en même temps, on ne peut pas avoir d'interblocage. Une possibilité serait par exemple d'ordonnancer T_1 et T_2 à la suite sur P_1 et T_3 et T_4 à la suite sur P_2.

Question 1.2

Comment ordonnancer le tableau suivant?

	T_1	T_2	T_3	T_4
V_1	oui	oui	oui	non
V_2	oui	oui	oui	non

Bien que ces méthodes existent, elles sont très difficiles à mettre en place et sont assez rares en pratique. Sachez simplement qu'elles existent et que parfois elles peuvent être utiles.