Introduction aux verrous

Orestis Malaspinas, Steven Liatti

1 Les verrous

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 plus en détails le concept de verrous. Nous allons d'abord voir comment nous pourrions tenter de construire un verrou dans un programme et se rendre compte que cela est quelque chose de compliqué pour que le verrou soit effectivement un mécanisme d'exclusion mutuelle (qu'il remplisse bel et bien son rôle) et qu'il soit efficace.

1.1 Les verrous: généralités

Le but d'un verrou est de s'assurer qu'uniquement un fil d'exécution à la fois peut accéder à une **section critique** d'un code. Une section critique contient des variables (contenant des données) partagées par plusieurs threads qui y sont modifiées (si on ne fait que lire des données il n'y a pas de problème en principe). Un exemple, de protection peut se voir avec l'exemple du compteur des exercices de la section sur l'api des threads POSIX

```
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

// du code

pthread_mutex_lock(&verrou);
counter += 1;
pthread_mutex_unlock(&verrou);

// encore du code
```

Un verrou est une variable, qui peut être dans deux états:

1. Libre (déverrouillé, disponible): aucun thread ne possède le verrou.
2. Occupée (acquis, verrouillé): exactement un thread possède le verrou et se trouve dans une section critique du programme.

On constate en premier lieu que dans l'interface POSIX un verrou est un `mutex` pour **mutual exclusion**.

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

La syntaxe pour le verrouillage/déverrouillage du verrou est très simple. Lorsque que `pthread_mutex_lock()` est appelé dans un thread, le fil d'exécution tente d'acquérir le verrou. S'il y parvient (cela veut dire qu'aucun autre thread ne l'a verrouillé) il devient le **propriétaire** du verrou, celui-ci entre dans l'état occupé (c'est pour cela que nous passons **la référence** au verrou à `pthread_mutex_lock()`), et le thread entre dans une section critique. S'il n'y parvient pas, il va rester bloqué dans cette fonction, jusqu'à ce qu'il puisse acquérir le verrou.

Quand le propriétaire d'un verrou (il doit l'avoir verrouillé avant), appelle la fonction `pthread_mutex_unlock()`, à la fin d'une section critique. Un autre thread peut donc en devenir le propriétaire et entrer dans sa section critique. Il n'y a aucune garantie quant à l'ordre de l'acquisition du verrou par plusieurs threads, si plusieurs sont en attente à leur fonction `pthread_mutex_lock()`. En revanche, si aucun fil d'exécution n'attend pour devenir propriétaire du verrou, il passe dans l'état libre (c'est pour cela que nous passons **la référence** au verrou à `pthread_mutex_lock()`).

On peut également noter qu'un verrou n'est rien d'autre qu'une variable. On peut donc en définir différents tout au long de l'exécution d'un programme. Chacun peut protéger des structures de données différentes dépendant de la **granularité**, **fine** ou **grossière** que chacun veut donner aux section critiques à protéger. Utiliser une approche plus fine permet "d'augmenter la concurrence" mais cela vient aussi avec une plus grande chance d'introduire des bugs.

Remarque 1

Il est important de noter que l'api des threads ne permet aucun contrôle sur l'ordre de leur exécution (tout est géré par l'OS). Les verrous permettent de récupérer un peu de contrôle en garantissant qu'un seul thread à la fois entre une section critique.

1.2 Construisons notre verrou

Comme nous allons le voir dans cette section, construire un verrou n'est pas simple du tout. En particulier, il serait tentant d'utiliser une simple variable booléenne pour garantir l'exclusion mutuelle. En fait, on se rendra compte que cela n'est pas possible sans l'aide du matériel et de l'OS.

Notations 1

Pour plus de généralité (et de simplicité dans les notations), nous noterons l'acquisition du verrou par la fonction `lock()` et sa libération `unlock()` et n'utiliserons pas l'api POSIX dans ce qui suit.

Avant de construire un verrou à proprement parler, nous devons d'abord définir quelles propriétés il doit avoir:

1. Il doit garantir **l'exclusion mutuelle**. Cela est la tâche primordiale d'un verrou. S'il ne garantit pas qu'un seul thread à la fois peut accéder à la partie du code protégée par le verrou, alors le verrou est inutile.
2. Il doit être aussi **équitable** que possible. Tous les fils d'exécution doivent avoir la possibilité d'acquérir le verrou, sinon ils risquent une terrible **famine** et n'auront jamais accès à la section critique.
3. La **performance** doit être bonne. Il y a deux cas de figure possibles:
   — le cas où il n'y a qu'un fil qui effectue le verrouillage/déverrouillage.
   — le cas où plusieurs threads sur un seul processeur combattent pour acquérir le verrou.
   — le cas où plusieurs threads, sur plusieurs CPUs sont en concurrence pour acquérir le verrou. Quel est le coût de calcul de cette opération dans ces cas? Quelle sera la performance d'un verrou dans chacun de ces cas?

Dans ce qui suit, nous allons considérer plusieurs technique et voir comment elles se comparent entre-elles pour chacun des critères.

1.2.1 Les interruptions

Une des premières façon de protéger une section critique a été de désactiver les **interruptions**[2]. Ainsi, en empêchant notre programme de s'interrompre pendant qu'il entre dans une section critique, nous nous assurions qu'aucun autre fil ne peut interrompre le code se trouvant dans une section critique. Puis lorsque la section critique est terminée, le processeur peut à nouveau interrompre le thread s'il le souhaite.

Une façon de se représenter ce comportement en pseudo-c, serait d'avoir les fonctions `lock()` et `unlock()`.

```
void lock() {
    disable_interrupts();
}

void unlock() {
    enable_interrupts();
}
```

Cette façon de procéder a l'avantage d'être d'une extrême simplicité. Il y a en revanche plusieurs problèmes fondamentaux également.

Question 1

Lesquels voyez-vous? Réfléchissez fort! Plus fort! C'est toujours pas assez fort!

Voilà c'est mieux.

---

2. Sur un système mono-processeur, une interruption est une instruction matérielle permettant de signaler au processeur qu'il doit sauver l'état dans lequel il se trouve et s'interrompre pour gérer un événement de plus haute priorité.

Le premier problème est que la désactivation/réactivation des interruptions est une action *privilégiée* qu'on ne peut pas laisser tout le monde effectuer et certainement pas des threads contrôlés par un utilisateur: on ne peut avoir aucune confiance que ce genre d'opération privilégiée sera utilisée de façon raisonnable. Imaginons que notre confiance soit mal placée en un programme appelant `lock()` au début de son code et rendant le processeur complètement inaccessible jusqu'à ce qu'il se termine? Imaginons qu'en plus il ne se termine jamais. La seule solution serait donc de relancer l'ordinateur. Comme on le voit ici, il faut un degré de confiance dans les applications qui est beaucoup trop élevé si on utilise les interruptions comme outil de synchronisation.

Un deuxième problème est que ce système ne peut pas fonctionner sur des systèmes comprenant plusieurs processeurs. Si les threads tournent sur des processeurs différents désactiver les interruptions, n'empêchera pas les threads se trouvant sur des CPUs différents d'entrer dans des sections critiques. Même en supposant qu'on a assez confiance dans nos applications, cette solution ne marchera simplement pas sur des systèmes multi-processeurs.

Troisièmement, désactiver les interruptions peut entraîner la "perte" de certaines interruptions primordiales (émises par d'autres processus). Cela peut arriver lorsqu'il faut réveiller un thread après la fin d'une opération de lecture sur le disque, ou l'arrivée d'un paquet réseau.

Finalement, cette approche est inefficace. C'est probablement la raison la moins importante, mais sur le matériel actuel, effectuer ce genre d'opérations est très lent.

La désactivation des interruptions est par conséquent plus utilisées dans les ordinateurs, mais est encore très présente dans les systèmes embarqués. Néanmoins, il y a quelques cas extrêmes où le système d'exploitation lui-même utilise ce mécanisme pour garantir l'atomicité de certains accès à des structures internes. Le système opérationnel se faisant confiance à lui-même, le problème de devoir faire confiance à un processus disparaît.

1.2.2 Un verrou raté

Essayons pour jouer de construire un verrou uniquement grâce à des commandes logicielles. Pour ce faire, nous allons définir notre verrou comme une simple variable booléenne, `bool locked`. Le code ci-dessous implémente le verrou à l'aide d'une variable booléenne.

```
typedef struct {
    bool locked;
} lock_t;

void init(lock_t *mutex) {
    mutex->locked = false;
}

void lock(lock_t *mutex) {
    while (mutex->locked == true) {
        // do nothing;
```

```
    }
    mutex->locked = true;
}

void unlock(lock_t *mutex) {
    mutex->locked = false;
}
```

Il fonctionne de la façon suivante:

- La fonction `init()` initialise le verrou à `false`.
- La fonction `lock()` vérifie dans une boucle **`while`**. Si le verrou est libre il est mis à `true`.
- La fonction `unlock()` met simplement la valeur du verrou à `false`.

Le fonctionnement de ce verrou est très simple. Le premier thread appelant `lock()` changera `locked` à `true` et pourra entrer dans sa section critique. N'importe quel autre thread essayant de devenir propriétaire du verrou entrera dans la boucle `while` et sera en **attente active** (**spin-wait** en anglais) jusqu'à ce que le verrou soit libéré par le premier thread et qu'il assigne `false` à la variable `locked`. A ce moment là, le verrou peut être acquis par un autre thread et ainsi entrer dans la section critique.

"Pour tout problème complexe, il existe une solution simple et élégante... et fausse" [3]. Ici nous sommes dans ce cas. Bien que simple et élégante cette solution est également fausse.

Question 2

Pourquoi cette solution est-elle fausse?

Réfléchissez avant de lire la suite. Faites même un dessin si cela peut vous aider.

On peut assez voir sur le listing ci-dessous qu'il n'y a pas d'exclusion mutuelle.

Thread 1	Thread 2
`lock()`	
`while (locked == true)`	
interruption on passe au thread 2	
	`lock()`
	`while (locked == true)`
	`locked = true`
	interruption on passe au thread 1
`locked = true // A-R-G-L!`	

Mais que se passe-t-il donc? Le thread 1 tente d'acquérir le verrou. Alors qu'il teste la valeur de `locked` et sort du **`while`**. A ce moment précis, le thread un

3. Citation attribuée à H. L. Mencken.

est préempté, un changement de contexte a lieu et l'exécution est passée au thread 2. Celui-ci appelle également la fonction `lock()`. La valeur de `locked` étant toujours `false` il sort de la boucle **`while`** et assigne la valeur `true` à la variable `locked`. A ce point le verrou ne devrait plus pouvoir être acquis par un autre thread. Hors si une seconde interruption se produit et la main est repassée au thread un, il continue exactement où il s'est arrêté, c'est-à-dire après être sorti de la boucle **`while`**. Il va donc également assigner la valeur `true` à `locked` et continuer son exécution. On voit donc ici que deux threads distincts peuvent acquérir le verrou et entrer dans leurs sections critiques respectives.

Ce verrou **ne fonctionne pas**.

Par ailleurs, mais cela est secondaire étant donné que ce verrou n'en est pas un, la technique consistant à avoir une attente active avec le **`while`** gaspille beaucoup de ressources sans raison. Cela est même absolument terrible dans le cas d'un système avec un seul CPU.

Question 3

A votre avis pourquoi?

Je veux voir vos méninges se tordre!

1.2.3 Un verrou qui marche: attente active et test-and-set

Comme il est en pratique impossible de se baser sur le mécanisme des interruptions pour construire des mécanismes d'exclusion mutuelle, les ingénieurs systèmes ont dû créer du matériel qui supporte des mécanismes de verrou. Le plus simple de tous à comprendre est l'instruction **test-and-set** (ou **échange atomique**). Cette instruction est **atomique**, c'est-à-dire qu'il est garanti qu'elle ne sera **jamais** interrompue. Cette garantie est apportée par le matériel directement. Une telle instruction aurait une syntaxe en C qui serait la suivante

```
bool test_and_set(bool *old_ptr, bool new) {
    bool old = *old_ptr; // on stocke la valeur pointée par d'old_ptr
    *old_ptr = new;      // on assigne `new` à `*old_ptr`
    return old;          // on retourne l'ancienne valeur d'`*old_ptr`
}
```

Souvenez-vous que cette instruction n'est jamais implémentée comme cela dans un code. Cette commande si vous l'implémentez comme cela ne sera pas atomique (e système d'exploitation fera tout son possible pour ruiner vos plans), il vous faut le support du matériel pour y arriver.

Remarque 2

Dans ces pseudo-codes, on utilise des booléens pour des questions de clarté. En pratique, ce sont des entiers qui sont utilisés (voire des bits uniquement).

Avec l'instruction `test_and_set()`, on peut construire un verrou avec attente active de la façon suivante

```
typedef struct {
    bool locked;
} lock_t;

void init(lock_t *mutex) {
    mutex->locked = false;
}

void lock(lock_t *mutex) {
    while (test_and_set(&mutex->locked, true) == true) {
        // do nothing;
    }
}

void unlock(lock_t *mutex) {
    mutex->locked = false;
}
```

Examinons ce qui se passe à présent lorsqu'un thread appelle `lock()` et que le verrou est libre (`locked == false`). Quand ce thread appelle la fonction `test_and_set()`, il recevra en retour l'ancienne valeur de `locked`, donc `false`, et sortira de la boucle **`while`** immédiatement. De plus, le thread assignera également **atomiquement** la valeur `true` à `locked`. Le verrou sera donc acquis et aucun autre thread ne pourra y accéder jusqu'à ce que ce même thread appelle la fonction `unlock()`.

L'autre possibilité est quand le verrou est déjà possédé par un thread. Un autre thread appelant la fonction `lock()`. Il appellera la fonction `test_and_set(locked, true)`. Cette fonction retournera la valeur stockée dans `locked` qui se trouve être `true` et lui assignera `true` à nouveau. Le thread entrera donc dans la boucle **`while`** et répétera l'opération jusqu'à ce que le verrou soit libéré.

Question 4

Quelle est la grande différence entre cette version du verrou, et notre version avec la simple variable booléenne?

La différence est que dans la version `test_and_set()` le test et l'assignation sont **une seule opération atomique** qui ne peut pas être interrompue (c'est une garantie du matériel). De cette façon nous garantissons qu'un seul thread peut acquérir le verrou à la fois.

Ce verrou est dit à **attente active** (ou **spin-lock**). En résumé il gaspille des cycles CPU jusqu'à ce que le verrou soit libéré. Ce système ne fait pas de sens sur les systèmes CPU s'il n'y a pas un ordonnanceur préemptif (un ordonnanceur qui va préempter un processus après un certain temps quoi qu'il arrive). En effet,

si le processus se retrouve dans la boucle `while` en attendant que le verrou soit libéré, il n'en sortira jamais. Il faut que ce thread soit préempté et que le thread possédant le verrou sorte de sa section critique (et ainsi libère le verrou) pour que le programme puisse continuer à s'exécuter, sinon...

1.2.4 Évaluation de l'efficacité des spin-lock

Maintenant que nous avons construit un verrou qui verrouille et garantit l'exclusion mutuelle (c'était notre première règle pour avoir un "bon" verrou) nous pouvons nous intéresser aux autre critères que nous avons énumérés précédemment.

Le deuxième critère est l'équité.

Question 5

Est-ce que le spin-lock est équitable? Les threads ont-ils tous une chance d'acquérir le verrou?

La réponse à cette question est **non** de ce que nous avons brièvement discuté précédemment. Il n'y a aucune garantie qu'un thread particulier puisse acquérir un spin-lock à un moment ou un autre. Il n'est pas du tout garanti non plus que les threads ne subissent pas la famine. Et comme tout ce qui n'est pas garanti il faut penser que cela va se passer, et en général cela sera au pire moment possible!

Le troisième critère est la performance.

Question 6

Quelle est la performance d'un spin-lock? Que se passe-t-il pour plusieurs fils d'exécution sur un CPU unique? Et qu'en est-il pour plusieurs CPUs?

Le problème de la performance des spin-locks se manifeste surtout lorsqu'on se trouve sur un système avec un seul CPU (ou lorsqu'il y a plus de threads que de CPUs). Dans ce cas, si le thread se trouve préempté alors qu'il est dans la section critique, les autres threads devront attendre que le thread ayant acquis le verrou soit à nouveau exécuté pour enfin pouvoir accéder au verrou.

Dans le cas de systèmes multi-CPUs où le nombre de thread est plus petit ou égal au nombre de CPUs, le problème se pose moins. Les autres processeurs se contenteront de tourner dans leur boucle `while` tant que le verrou ne sera pas libéré. Mais cela n'a pas un coût notable, on utilise juste des cycles CPUs à ne rien faire.

1.2.5 L'atomicité dans la vraie vie: `compare-and-exchange`

La primitive hardware utilisée dans les systèmes `x86` est le `compare-and-exchange`. Le pseudo-code C de cette instruction ressemblerait à

```
bool compare_and_exchange(bool *ptr, bool expected, bool new) {
    bool actual = *ptr;
    if (actual == expected) {
        *ptr = new;
    }
    return actual;
}
```

Dans cette instruction, on teste d'abord si la valeur contenue à l'adresse `ptr` est égale à une valeur attendue, `expected`. Si cela est le cas, on met à jour cette valeur à une nouvelle valeur `new`.

Exercice 1

Comment construirait-on un verrou avec attente active avec cette instruction?

Mais pourquoi introduire une seconde instruction alors que le `test_and_set()` faisait l'affaire? En fait, le `compare_and_exchange()` est une instruction est un peu plus puissante que le `test_and_set()` mais pas pour construire un verrou avec attente active. Si par miracle nous avons le temps, nous en dirons plus si nous parlons de synchronisation sans verrou.

1.2.6 Un verrou équitable

Pour ce verrou, nous avon besoin d'un autre type d'instruction atomique. Une d'un type permettant d'incrémenter une valeur, tout en retournant l'ancienne. Ce type d'instruction s'appelle `fetch-and-add`. En pseudo-code C, cela ressemblerait à

```
int fetch_and_add(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Avec cette instruction, nous pouvons construire le verrou par ticket (ticket lock). Le verrou à ticket est un peu plus complexe qu'un simple `flag`. Nous allons utiliser un numéro de tour, ainsi qu'un ticket pour créer le verrou. Le code pour ce type de verrou serait du genre

```
typedef struct {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}
```

```
void lock(lock_t *lock) {
    int myturn = fetch_and_add(&lock->ticket);
    while (lock->turn != myturn) {
        // do nothing
    }
}

void unlock(lock_t *lock) {
    lock->turn += 1;
}
```

Avant de tenter d'acquérir le verrou, celui-ci incrémente la valeur du ticket et retourne son ancienne valeur. Si le ticket avait inscrit comme valeur, la valeur du tour du thread en question, il peut sortir de la boucle `while`. Sinon, il doit attendre. Lorsque le thread qui a acquis le verrou sort de sa section critique et qu'il déverrouille le verrou, il incrémente la valeur du tour se trouvant dans le verrou qui est partagé par tous les threads. Le threads dont c'est le tour en prochain (s'il y en a un) peut acquérir le verrou.

1.2.7 L'inversion de priorité

Une autre raison de ne pas utiliser les verrous à attente active est que sur certains systèmes ils ne fonctionnent pas... Ce problème est connu sous le nom de **l'inversion de priorité**.

Imaginons un système avec deux fils d'exécution, T_1 et T_2. De plus T_2 a une priorité d'ordonnancement plus élevée que T_1, et donc l'ordonnanceur préemptera toujours T_1 pour exécuter T_2 lorsque les deux threads sont exécutables. T_1 ne pourra s'exécuter que lorsque T_2 est bloqué (en train d'effectuer une opération I/O). Imaginons à présent que T_2 est bloqué (en train d'effectuer une opération I/O par exemple). T_1 s'exécute et acquière le verrou et pénètre dans sa section critique. Si à ce moment là, T_2 repasse en attente d'exécution, T_1 sera préempté, car T_2 a une priorité plus élevée. T_2 essaiera d'acquérir le verrou, et sera donc en attente dans la boucle `while`. Mais comme T_1 ne sera jamais ordonnancé (à cause de sa plus faible priorité) le système sera bloqué et rien ne va se passer.

En fait, le problème d'inversion de priorité peut arriver même avec des verrous sans attente active. En effet, imaginons à présent trois threads T_1, T_2, et T_3 avec T_3 ayant une priorité plus haute que T_1 et T_2, qui lui a une priorité plus élevée que T_1. Si T_1 acquière le verrou et que T_3 entre dans l'état d'attente d'exécution, il sera immédiatement ordonnancé et T_1 préempté. Mais comme T_1 est propriétaire du verrou, T_3 est bloqué. Si à présent T_2 commence à s'exécuter, comme il a une priorité plus élevée il passera toujours avant T_1, et donc il y a une chance pour que T_1 ne soit jamais ordonnancé, bloquant T_3 étant donné qu'il attend que T_1 libère le verrou.

Il existe différentes façons de se prémunir contre les inversions de priorité. La plus simple étant d'avoir tous les threads avec les mêmes priorités. Une autre façon serait d'avoir un mécanisme où les threads avec des priorités plus élevées attendant sur des threads avec une priorité plus basse puisse augmenter la priorité des threads avec une priorité plus basse.

1.3 Aller plus loin que le verrou à attente active

Comme nous l'avons discuté plus haut, les verrous à attente active marchent et sont très simples. En revanche, ils sont inéquitables et inefficaces. En particulier, si un changement de contexte intervient lorsqu'un thread est dans une section critique, nous avons un problème. Pour pouvoir aller plus loin, il nous faut le soutient du système d'exploitation.

1.3.1 L'approche simple: céder

Afin d'aller plus loin, comme on vient de le dire, on va avoir besoin de l'aide du système d'exploitation. Pour ce faire la stratégie sera très simple dans un premier temps. Plutôt que de tourner dans le vide, on va céder (yield) le contrôle du CPU à un autre thread.

A titre d'illustration, En pseudo-c un verrou ressemblerait à quelque chose de ce genre

```
void init(lock_t *mutex) {
    mutex->locked = false;
}

void lock(lock_t *mutex) {
    while (compare_and_exchange(&mutex->locked, false, true) == true) {
        yield(); // on passe le CPU
    }
}

void unlock(lock_t *mutex) {
    mutex->locked = false;
}
```

où `yield()` est une primitive du système d'exploitation ayant pour effet de faire passer le fil d'exécution qui l'appelle de l'état **en cours d'exécution** à l'état **prêt**. Ainsi, l'ordonnanceur du système d'exploitation peut promouvoir un autre thread à l'état **en cours d'exécution**.

Exemple 1 *(favorable)*

Considérons un processus tournant sur un CPU avec deux threads, T_1 et T_2. Supposons que T_1 a acquis le verrou et se trouve dans sa section critique. Si un changement de contexte intervient à ce moment-là, et que T_2 essaie d'acquérir le verrou, il le trouvera verrouillé et entrera dans la boucle **`while`** et sera mis dans l'état **en attente**, donnant une chance au système d'exploitation d'ordonnancer T_1 à nouveau. Celui-ci pourra terminer sa section critique.

En fait, cette méthode de céder le CPU lorsque le verrou est déjà acquis par un autre thread, fait reposer toute la responsabilité sur le système d'exploitation. On vient de voir que c'est une approche raisonnable pour un cas avec peu de

threads. Lorsque leur nombre devient trop nombreux cela est beaucoup moins le cas.

Exemple 2 *(moins favorable)*

Imaginons à présent cent threads donnant un combat à mort pour l'acquisition du verrou, T_{1-100}. Supposons que T_{48} a acquis le verrou et est entré dans sa section critique. C'est ce moment précis que choisit le système d'exploitation (détestant T_{48}) pour le préempter. Un autre thread sera ordonnancé et trouvera le verrou acquis, il appellera donc `yield()` et sera mis **en attente**. Si nous supposons que l'ordonnanceur a une stratégie qui consiste à passer la main à chaque thread à tour de rôle, nous aurons 99 changements de contexte avant que T_{48} puisse reprendre la main et en finir avec sa section critique. Cela reste beaucoup mieux que le spin-lock qui aurait dû être préempté 99 fois pour revenir à T_{48} mais ce n'est pas parfait, car cela nécessite quand même un grand nombre de changements de contexte.

Bien que bien plus efficace que les spin-locks, ce type de verrou n'est pas parfait. Comme nous venons de le voir avec notre exemple à 100 threads, il est nécessaire d'effectuer des changements de contexte et passer la main pour acquérir le verrou. Cela peut être une opération coûteuse et n'est pas très satisfaisante. Par ailleurs, il n'est pas exclu qu'un fil d'exécution se retrouve coincé dans une boucle où il cède le CPU indéfiniment et va se retrouver dans un état de **famine**: il n'entrera jamais dans sa section critique. Ce problème est encore plus sérieux que celui de la performance, donc il faut trouver une meilleure solution.

1.3.2 L'utilisation des files

Afin de remédier aux problèmes que nous avons énoncé plus tôt (le gaspillage de ressources, mais plus important encore la famine), il est nécessaire d'utiliser une autre approche. En fait, le problème est dû à la trop grande confiance que nous devons avoir en l'ordonnanceur du système d'exploitation. Une solution est d'exercer un plus grand contrôle sur quel thread doit être le prochain à pouvoir acquérir un verrou quand il a été libéré par son précédent propriétaire.

Pour ce faire, nous allons utiliser un système de file et le support du système d'exploitation. Pour simplifier, nous allons utiliser le support du système de `Solaris` pour réveiller et mettre en sommeil des threads. L'API est la suivante:

— `park()` qui met le thread qui appelle cette fonction en attente;
— `unpark(tid)` qui réveille the thread dont l'identifiant est `tid`.

Avec ces deux appels, il est possible de construire un verrou qui:

1. Mettra en attente un thread qui tentera d'acquérir un verrou déjà verrouillé.
2. Réveillera ce thread lorsque le verrou devient libre.

Le code suivant permet de créer un verrou selon ce modèle.

```
typedef {
    bool locked;
    bool guard;
    queue_t *q; // une file
} lock_t;

void init(lock_t *mutex) {
    mutex->locked = false;
    mutex->guard = false;
    queue_init(mutex->q);
}

void lock(lock_t *mutex) {
    while (test_and_set(&lock->gard, true) == true) {
        // la valeur de guard est acquise en
        // tournant dans la boucle while
    }
    if (mutex->locked == false) {
        mutex->locked = true; // le verrou est maintenant acquis
        mutex->guard = false; // la garde est maintenant fausse
    } else {
        queue_add(mutex->q, get_id()); // on met l'id du thread dans la queue
        m->guard = false;
        park(); // on met le thread en sommeil
    }
}

void unlock(lock_t *mutex) {
    while (test_and_set(&lock->gard, true) == true) {
        // la valeur de guard est acquise en
        // tournant dans la boucle while
    }
    if (queue_empty(mutex->q)) {
        mutex->locked = false; // le verrou est libéré si aucun thread
                               // est en attente
    } else {
        unpark(queue_remove(mutex->q)); // si un thread est en attente
                                        // lui garder le thread
    }
    mutex->guard = false;
}
```

Remarque 3

On peut constater qu'il y a un verrou à attente active caché dans ce verrou. En effet, lors de la manipulation de la variable `locked` et de la file, on utilise l'instruction `compare_and_exchange()` combiné à un **`while`**. Néanmoins, cela est moins problématique que lorsque le verrou protège des sections critiques

étant donné que le nombre d'instructions est limité (on met à jour une variable et ajoute un élément dans une file) en comparaison de la protection d'une section critique qui peut contenir beaucoup de calculs et ainsi avoir plus de chances d'être préemptée par le système d'exploitation.

Ce verrou fonctionne à l'aide de deux variables booléennes et d'une file d'attente. En premier lieu, nous avons la variable `lcoked` qui détermine si le verrou est acquis ou non. En second lieu, vient la variable `guard` qui elle protège `locked` et la file d'attente contre les accès concurrents. Finalement, la file d'attente enregistre les identifiants des threads tentant de prendre possession du verrou mais ayant été recalés.

Lorsque qu'on appelle la fonction `lock()` et que le verrou est déjà détenu par un autre thread, on ajoute l'identifiant (`tid`) du thread se battant pour l'acquisition du verrou dans une file d'attente et en mettant `gard` à `false` avant d'utiliser la fonction `park()`.

Question 7

Que se passe-t-il si on met `gard` à false après `park()` plutôt?

En fait, on voit bien que si `gard` est mise à jour après `park()` on ne déverrouille jamais le verrou utilisé pour protéger la file et `locked`. De plus, on ne met jamais `locked` à `true` après le réveil du thread (après `unpark(tid)`). Ceci est dû au fait que lors du réveil, le thread continue son exécution après l'appel à `park()`. Comme à ce moment-là il n'as pas acquis `guard`, il serait très dangereux de lui faire modifier `locked`. Enfin, ce code contient un accès concurrent potentiel. Si par malheur le système d'exploitation, préemptait le thread entre le moment où on libère `gard` et l'appel à `park()`. A ce moment-là, le thread pense que le verrou est acquis par un autre thread. Si à ce moment-là un autre thread prend la main et parvient à libéré le verrou, lorsque notre premier thread reprendrait la main, il pourrait se retrouver en attente infinie de libération d'un verrou déjà libéré...

Pour ce prémunir contre ce problème, un autre appel a été créé dans Solaris: `setpark()`. En appelant cette fonction, un thread indique au système qu'on est sur le point d'appeler `park()`. Ainsi, si `unpark()` est appelé avant que `park()` soit effectivement appelé, `park()` retourne immédiatement plutôt que de se mettre en sommeil. De cette façon on évite le sommeil à durée indéterminée.

1.3.3 Et sous Linux?

Comme on l'a vu avec les verrous appels systèmes, il n'existe pas de façon unique de définir les appels vers le noyaux pour fabriquer un verrou. Dans les systèmes linux, on a à disposition le **futex** qui est similaire aux fonctionnalités de Solaris, mais où chaque futex a un espace mémoire spécifique, ainsi qu'une file d'attente qui est directement implémentée dans le noyau. Ci-dessous, vous trouverez l'imprémentation des fonctions `lock()` et `unlock()` pour le `mutex`

(qui n'est rien d'autre qu'un entier) dans la librairie `libc` (cela se trouve dans `lowlevellock.h`).

```
void mutex_lock (int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (the fastpath) */
    if (atomic_bit_test_set (mutex, 31) == 0)
        return;
    atomic_increment (mutex);
    while (1) {
        if (atomic_bit_test_set (mutex, 31) == 0) {
            atomic_decrement (mutex);
            return;
        }
        /* We have to waitFirst make sure the futex value
            we are monitoring is truly negative (locked). */
        v = *mutex;
        if (v >= 0)
            continue;
        futex_wait (mutex, v);
    }
}
void mutex_unlock (int *mutex) {
    /* Adding 0x80000000 to counter results in 0 if and
       only if there are not other interested threads */
    if (atomic_add_zero (mutex, 0x80000000))
        return;
    /* There are other threads waiting for this mutex,
       wake one of them up. */
    futex_wake (mutex);
}
```

Regardons ce qui se passe ici. En premier lieu, nous voyons que notre verrou n'est rien d'autre qu'un entier. Dans le bit de poids fort, on stocke l'information sur le verrou (est-il acquis ou pas) et dans le reste des bits, on stocke le nombre de threads qui attendent. Donc si `mutex` est négatif le verrou est déjà détenu par un thread (si ce bit est mis à 1, l'entier est négatif).

Pour l'acquisition : * Si le verrou n'est pas acquis, on incrémente atomiquement l'entier, puis ont attend activement pour voir si le verrou est toujours (vraiment) acquis. On vérifie par conséquent si la valeur de l'entier est bien négative. Puis, on met le thread en attente via `futex_wait(mutex, v)`. Cet appel système met le thread le thread appelant en sommeil, si la valeur de `mutex` est la même que celle de `v`. Sinon il retourne immédiatement (pour se prémunir de l'attente indéterminée qu'on a vue plus haut).

Pour la libération: * Si le verrou est à zéro (et que la file est vide), on ajoute `0x80000000`. On a donc que la valeur du verrou est: `10000000000000000000000000000000`. Sinon, on reveille le thread suivant avec `futex_wait(mutex)`, où `mutex` est l'adresse du thread à réveiller.

Il faut noter que cette méthode est très efficace quand il n'y a pas de contention

pour acquérir le verrou. En effet, lorsqu'un seul thread combat avec lui-même pour acquérir le verrou, il ne fera que deux opération très simples: un `test_and_set` atomique sur un seul bit pour l'acquisition, puis une addition atomique pour la libération.