perlthrtut

Table des matières

1 NAME/NOM

perlthrtut - Tutoriel sur les threads en Perl

2 DESCRIPTION

Ce tutoriel décrit les threads à la nouvelle mode introduites dans Perl 5.6.0, appelés threads interpréteur, ou **ithreads** pour faire court. Dans ce modèle, chaque thread exécute son propre interpréteur Perl, et tout partage de données entre les threads doit être explicite.

Il y a une autre, plus vieille mode de gestion des threads en Perl, appelée le modèle 5.005 ; évidemment, il s'applique aux versions 5.005 de Perl. Ce vieux modèle a des problèmes connus, est obsolète, et sera probablement retiré autour de la version 5.10. Vous êtes fortement encouragé à faire migrer aussi vite que possible tout code utilisant le modèle 5.005 vers le nouveau modèle.

Vous pouvez savoir quel modèle de threads vous avez (ou savoir si vous n'en avez aucun) en exécutant `perl -V` et en examinant la section `Platform`. Si vous avez `useithreads=define`, vous avez les ithreads ; si vous avez `use5005threads=define`, vous avez les threads 5.005. Si vous n'avez aucun des deux, votre perl ne contient pas de support pour les threads. Si vous avez les deux, vous avez un problème.

L'accès aux threads 5.005 se faisait à travers la classe *Threads*, alors que celui aux ithreads se fait grâce à la classe *threads*. Notez le t majuscule et minuscule.

3 État courant

Le code du modèle ithreads existe depuis Perl 5.6.0, et est considéré stable. L'interface utilisateur aux ithreads (la classe *threads*) est apparue dans la version 5.8.0, et aujourd'hui on peut la considérer comme stable, bien qu'il faille la traiter avec précautions comme tout ce qui est nouveau.

4 Qu'est-ce que c'est qu'un thread, d'abord ?

Un thread (*processus léger* en bon Français) est un flux de contrôle à travers un programme, muni d'un point d'exécution unique.

Ça ressemble vraiment à un processus, non ? Eh bien, c'est normal. Les threads sont un des composants d'un processus. Chaque processus a au moins un thread, et d'ailleurs jusqu'à aujourd'hui, chaque processus exécutant Perl a eu un seul thread. Avec Perl 5.8, cependant, vous pouvez créer des threads supplémentaires. Nous allons voir comment, quand, et pourquoi.

5 Modèles de programmes utilisant les threads

Il y a trois façon de structurer un programme utilisant des threads. Le choix du modèle dépend de ce que vous voulez que votre programme fasse. Pour de nombreux programmes non triviaux utilisant les threads, vous allez devoir choisir des modèles différents pour les différentes parties de votre programme.

5.1 Maître/Esclave

Le modèle maître/esclave met en scène un seul thread "maître", et un thread "esclave" ou plus. Le thread maître rassemble et génère les tâches qui doivent être accomplies, et répartit ces tâches aux threads esclaves.

Ce modèle est courant dans les programmes serveurs et à interface graphique, où un thread principal attend un évènement et le passe aux threads esclaves appropriés, pour qu'ils le traitent. Une fois l'évènement transmis, le thread maître se remet à attendre un autre évènement.

Le thread maître fait assez peu de travail. Même si les tâches ne sont pas nécessairement traitées plus vite qu'avec une autre méthode, celle-ci a tendance à présenter les meilleurs temps de réponse à l'utilisateur.

5.2 Équipe de travail

Dans le modèle de l'équipe de travail, plusieurs threads traitent de la même façon les différents morceaux de données. Cela reproduit le mode de travail du calcul parallèle classique et des processeurs vectoriels, où une série de processeurs font exactement la même chose à de nombreux morceaux de données.

Ce modèle est particulièrement utile si le système qui exécute le programme peut distribuer les threads sur les différents processeurs. Il peut aussi être utile en lancer de rayons ou dans les moteurs de rendu, quand les threads individuels peuvent renvoyer des résultats intermédiaires pour donner un retour visuel à l'utilisateur.

5.3 Travail à la chaîne

Le modèle de travail à la chaîne divise une tâche en une suite d'étapes, et consiste à passer les résultats d'une étape au thread qui s'occupe de l'étape suivante. Chaque thread fait une chose à chaque morceau de données et passe ses résultats au thread suivant de la chaîne.

Ce modèle est surtout conseillé si vous avez plusieurs processeurs, de façon à ce que plusieurs threads s'exécutent en parallèle, même si il peut aussi servir dans d'autres contextes. Il permet de garder les tâches élémentaires simples, et autorise certaines partie de la chaîne à bloquer (sur des appels système d'E/S, par exemple) alors que d'autres parties continuent à tourner. Si vous exécutez les différentes parties de la chaîne sur des processeurs différents, cela vous permet aussi de tirer avantage du cache de chaque processeur.

Ce modèle est aussi pratique pour une forme de programmation récursive dans laquelle au lieu de s'appeler elle-même, une fonction crée un autre thread. Les générateurs de nombres premiers ou de Fibonacci profitent bien du modèle de travail à la chaîne. (Une version d'un générateur de nombres premiers est présentée plus bas.)

6 Implémentations des threads dans le système d'exploitation

Il y a plusieurs façons différentes d'implémenter les threads sur un système donné. Cela dépend de la marque, et, parfois, de la version du système d'exploitation (SE). Souvent la première implémentation est relativement simple, mais celles des versions suivantes du SE sont plus sophistiquées.

Les informations présentées dans cette section sont utiles mais pas indispensables ; vous pouvez donc les sauter si elles vous rebutent.

Il y a trois sortes de threads : les threads en mode utilisateur, les threads du noyau, et les threads multiprocesseurs du noyau.

Les threads en mode utilisateur vivent entièrement à l'intérieur d'un programme et de ses bibliothèques. Dans ce modèle, le SE ne sait rien des threads. Pour ce qu'il en voit, votre processus utilisant les threads n'est qu'un processus comme les autres.

C'est la façon la plus facile d'implémenter les threads, et celle avec laquelle commencent la plupart des SE. Le gros désavantage est que, puisque le SE ne sait rien des threads, si un des thread bloque, ils bloquent tous. Parmi les activités bloquantes courantes, on trouve la plupart des appels système, des E/S, et les choses comme sleep().

Les threads du noyau sont l'étape suivante dans l'évolution des threads. Le SE connaît des threads gérés par le noyau et en tient compte. La principale différence entre un thread du noyau et un thread en mode utilisateur est le blocage. Avec les threads du noyau, un thread peut bloquer sans que les autres threads bloquent. Ce n'est pas le cas avec les threads en mode utilisateur, où le noyau bloque au niveau du processus et pas au niveau du thread.

C'est un grand pas en avant, qui peut donner à un programme utilisant les threads un bonus de performance sur les programmes ne les utilisant pas. Les threads qui bloquent au cours d'E/S, par exemple, ne bloqueront pas les threads qui font autre chose. Cependant, chaque processus ne peut avoir qu'un thread en train de s'exécuter à un instant donné, quel que soit le nombre de processeurs que le système a.

Puisque le noyau peut interrompre un thread à n'importe quel moment, cela va mettre à nu certaines des hypothèses implicites d'exclusion que vous pouvez faire dans votre programme. Par exemple, quelque chose d'aussi simple que `$a = $a + 2` peut se comporter de façon imprévisible avec des threads du noyau si $a est visible aux autres threads : un autre thread peut en effet changer $a entre le moment où $a est lue dans la partie droite, et celui où la nouvelle valeur est stockée.

Les threads multiprocesseurs du noyau sont l'étape finale dans l'évolution du support pour les threads. Avec ce modèle, sur une machine à plusieurs processeurs, le SE peut faire tourner simultanément plusieurs threads sur plusieurs processeurs.

Cela peut améliorer sérieusement les performances d'un programme utilisant les threads, puisque plusieurs s'exécuteront en même temps. En retour, cependant, tous les fourbes problèmes de synchronisation qui ne se montraient pas avec les threads du noyau vont surgir pour se venger.

Au-delà des différents niveaux d'implication des SE dans la gestion des threads, il y a différentes façon pour un SE (et pour une implémentation des threads sur un même SE) d'allouer des cycles processeur aux threads.

Les systèmes multitâches coopératifs obligent les threads qui s'exécutent à leur rendre le contrôle dans deux cas. Un thread peut demander explicitement de céder le contrôle. Il le cède aussi s'il fait quelque chose de bloquant, comme des E/S. Dans un système multitâche coopératif, un thread peut priver tous les autres de temps processeur si il le décide.

Les systèmes multitâches préemptifs interrompent les threads à intervalle régulier, et le système décide quelle thread doit être exécuté ensuite. Sur un tel système, un thread ne peut généralement pas monopoliser le processeur.

Sur certains systèmes, des threads coopératifs et préemptifs peuvent tourner simultanément. (Par exemple, Les threads avec des priorités temps-réel se comportent souvent coopérativement, alors que ceux avec des priorités normales se comportent préemptivement.)

7 De quelle sorte sont les threads de Perl ?

Si vous avez expérimenté d'autres implémentations des threads, vous pourriez avoir l'impression que les choses ne sont pas ce à quoi vous vous attendiez. Avec les threads de Perl, il est très important de garder à l'esprit qu'ils ne sont d'aucune école particulière. Ce ne sont pas des threads POSIX, ni DecThreads, ni des Green Threads de Java, ni des threads Win32. Il y a des similarités, et les concepts généraux sont les mêmes, mais si vous commencez à mettre votre nez dans les détails d'implémentation, vous allez être déçu, ou troublé. Les deux peut-être.

Cela ne veut pas dire que les threads de Perl sont différents de tout ce qui est venu avant. Le modèle de Perl doit beaucoup aux autres modèles, surtout au modèle POSIX. Mais de la même façon que Perl n'est pas C, les threads Perl ne sont pas les threads POSIX. Donc, si vous commencez à chercher des mutexes ou des priorités de threads, prenez du recul et repensez à ce que vous voulez faire, et à comment Perl peut le faire.

Cependant, il est important de se souvenir que les threads de Perl ne peuvent pas faire des choses que n'autorise pas votre système d'exploitation. Si celui-ci bloque un processus entier sur sleep(), Perl va bloquer aussi.

Les Threads de Perl Sont Différents.

8 Modules réentrants (*thread-safe*)

L'ajout des threads a changé l'intérieur de Perl substantiellement. Cela a des implications pour ceux qui écrivent des modules utilisant XS ou des bibliothèques externes. Cependant, dans la mesure où par défaut les données ne sont par partagées par les threads, les modules Perl ont de grandes chances d'être déjà réentrants, ou peuvent le devenir facilement. Il faut tester ou passer en revue le code des modules non réentrants avant de les utiliser en production.

Tous les modules que vous pouvez utiliser ne sont pas réentrants, et vous devriez toujours supposer qu'un module ne l'est pas, à moins que sa documentation ne dise explicitement le contraire. Cela tient pour les modules qui sont distribués dans la version standard de Perl. Les threads sont une caractéristique nouvelle, et même certains des modules standards ne sont pas réentrants.

Même si un module est réentrant, cela n'implique pas qu'il soit optimisé pour fonctionner avec les threads. Il est possible qu'il puisse être réécrit pour tirer parti des nouvelles fonctionnalités liées aux threads de Perl, pour augmenter les performances dans un environnement utilisant les threads.

Si vous utilisez un module qui se trouve n'être pas réentrant, vous pouvez vous protéger en ne l'utilisant que depuis un seul et unique thread. Si vous avez besoin que de plusieurs threads accèdent à ce module, il vous reste à utiliser des sémaphores et beaucoup de discipline. Les sémaphores sont décrits dans Sémaphores de base (§11.5).

Voir aussi Réentrance des bibliothèques système (§16).

9 Bases des threads

Le module standard *threads* fournit les fonctions de base nécessaires à l'écriture de programmes utilisant les threads. Dans les section suivantes, nous allons en décrire les bases, en vous montrant ce dont vous avez besoin pour faire tourner un programme avec des threads. Après ça, nous passerons en revue certaines fonctionnalités du module *threads* qui rendent la vie avec les threads plus facile.

9.1 Support de base pour les threads

Le support de Perl pour les threads est une option de compilation - il est activé ou désactivé quand Perl est compilé sur votre machine, et non quand vos propres programmes sont compilés. Si votre perl n'a pas été compilé avec le support pour les threads, toute tentative d'utiliser les threads est vouée à l'échec.

Vos programmes peuvent utiliser le module Config pour déterminer si les threads sont activés ou non. Si votre programme ne peut tourner sans eu, vous pouvez écrire quelque chose comme :

```
$Config{useithreads} or die
  "Recompilez Perl avec les threads activés pour faire tourner ce programme.";
```

Si nous devons faire usage d'un module capable d'utiliser les threads ou non, nous pouvons écrire un programme qui les utilise si possible :

```
use Config;
use MonModule;

BEGIN {
    if ($Config{useithreads}) {
        # Nous avons les threads
        require MonModule_avec_threads;
        import MonModule_avec_threads;
    } else {
        require MonModule_sans_threads;
        import MonModule_sans_threads;
    }
}
```

Du code capable de tourner avec et sans les threads est généralement très touffu, aussi est-il préférable d'isoler le code spécifique aux threads dans son propre module. Dans l'exemple ci-dessus, c'est ce que fait MonModule_avec_threads, qui est seulement importé si nous tournons dans un perl avec les threads activés.

9.2 Note à propos des exemples

Bien que le support pour les threads soit considéré comme stable, il reste un certain nombre de rugosités qui pourraient vous gêner si vous essayez les exemples ci-dessous. Dans une situation réelle, il faudrait prendre garde que tous les threads aient fini avant que le programme ne se termine. Cette précaution n'a **pas** été prise dans nos exemples, par souci de simplicité. Tels quels, il vont produire des messages d'erreur, généralement à cause de threads qui tournent encore quand le programme se termine. Ne vous en effrayez pas. Les versions futures de Perl régleront peut-être ce problème.

9.3 Créer des threads

Le module *threads* fournit les outils nécessaires à la création de threads. Comme n'importe quel autre module, vous devez dire à Perl que vous voulez l'utiliser ; `use threads` importe tous les éléments dont vous avez besoin pour créer des threads.

La façon la plus simple de créer un thread utilise new() :

```
use threads;

$thr = threads->new(\&sub1);

sub sub1 {
    print "Dans le thread\n";
}
```

La méthode new() prend une référence à une fonction et crée un nouveau thread, qui commence à s'exécuter dans ladite fonction. Le contrôle est alors à la fois dans la fonction et dans l'appelant.

Si nécessaire, votre programme peut passer des paramètres à la fonction au démarrage du thread. Il suffit d'inclure la liste des paramètres dans l'appel à `threads::new`, comme ceci :

```
use threads;

$Param3 = "toto";
$thr = threads->new(\&sub1, "Param 1", "Param 2", $Param3);
$thr = threads->new(\&sub1, @ListeDeParametres);
$thr = threads->new(\&sub1, qw(Param1 Param2 Param3));

sub sub1 {
    my @Parametres = @_;
    print "Dans le thread\n";
    print "Reçu les paramètres >", join("<>", @Parametres), "<\n";
}
```

Cet exemple illustre une autre caractéristique des threads : vous pouvez générer plusieurs threads en utilisant la même fonction. Chaque thread exécute la même fonction, mais avec un environnement séparé et des arguments potentiellement différents.

`create()` est un synonyme de `new()`.

9.4 Rendre le contrôle

On veut parfois qu'un thread rende explicitement le contrôle du processeur à un autre thread. Par exemple, votre système de gestion des threads pourrait ne pas supporter le multitâche préemptif, ou vous pourriez être en train de faire quelque chose de très lourd en calculs, et vouloir être sûr que le thread de l'interface utilisateur soit appelé assez souvent. Et il y a des cas où l'on veut juste qu'un thread lâche le contrôle du processeur.

Le système de threads de Perl fournit la fonction yield(), qui permet de faire ceci. yield() a une utilisation assez évidente :

```
use threads;
```

```
sub boucle {
        my $thread = shift;
        my $toto = 50;
        while($toto--) { print "dans le thread $thread\n" }
        threads->yield;
        $toto = 50;
        while($toto--) { print "dans le thread $thread\n" }
}

my $thread1 = threads->new(\&boucle, 'premier');
my $thread2 = threads->new(\&boucle, 'deuxième');
my $thread3 = threads->new(\&boucle, 'troisième');
```

Il est important de garder à l'esprit que yield() n'est qu'une suggestion de rendre le contrôle du processeur, et qu'il dépend de votre matériel, SE et bibliothèques de threads que cela arrive réellement. Par conséquent, il ne faut pas espérer imposer l'ordonnancement des threads avec des appels à yield(). Cela peut marcher sur votre plateforme, mais cela ne marchera pas sur toutes.

9.5 Attendre qu'un thread termine

Puisque les threads sont aussi des fonctions, elles peuvent renvoyer des valeurs. Pour attendre qu'un thread termine et utiliser les valeurs qu'il peut retourner, vous pouvez vous servir de la méthode join() :

```
use threads;

$thr = threads->new(\&sub1);

@DonneesRenvoyees = $thr->join;
print "Le thread a renvoyé @DonneesRenvoyees";

sub sub1 { return "Cinquante-six", "toto", 2; }
```

Dans cet exemple, la méthode join() retourne dès que le thread se termine. En plus d'attendre qu'un thread termine et de rassembler les valeurs qu'il peut retourner, join() effectue aussi le nettoyage nécessaire. Ce nettoyage peut prendre des ressources importantes, particulièrement pour les programmes qui tournent longtemps et génèrent de nombreux threads. Si vous ne voulez pas récupérer les valeurs de retour ni attendre que le thread finisse, vous devriez plutôt appeler la méthode detach(), décrite ci-dessous.

9.6 Ignorer un thread

join() fait trois choses : il attend qu'un thread se termine, le nettoie, et retourne les valeurs qu'il peut avoir produites. Mais que faire si vous n'êtes pas intéressé par le retour du thread, et que vous n'avez cure de quand il finit ? Tout ce qui vous importe alors est que le thread soit nettoyé quand il se termine.
Dans ce cas, utilisez la méthode detach(). Une fois qu'un thread et détaché, il continuera à s'exécuter jusqu'à ce qu'il termine, après quoi Perl le nettoiera automatiquement.

```
use threads;

$thr = threads->new(\&sub1); # Générer le thread

$thr->detach; # A partir de maintenant, nous nous désintéressons
              # officiellement du thread

sub sub1 {
    $a = 0;
    while (1) {
        $a++;
        print "\$a vaut $a\n";
        sleep 1;
    }
}
```

Une fois qu'un thread est détaché, il ne peut plus être rejoint (avec `join()`), et toute valeur de retour qu'il pourrait avoir produite (s'il avait déjà fini et attendait un join()) est perdue.

10 Threads et données

Maintenant que nous avons couvert les bases des threads, il est temps de passer au sujet suivant : les données. L'utilisation des threads introduit quelques complications à l'accès aux données dont les programmes sans threads n'ont jamais à se préoccuper.

10.1 Données partagées et non partagées

La plus grande différence entre les ithreads de Perl et les vieux threads 5.005, ou d'ailleurs n'importe quel autre système de threads, est que par défaut, aucune donnée n'est partagée. Quand un nouveau thread Perl est créé, toutes les données associées au thread courant sont copiées vers le nouveau thread, et sont dorénavant données privées du nouveau thread ! C'est similaire dans l'esprit à ce qui arrive quand un processus UNIX fait un fork(), sauf que dans notre cas les données sont copiées vers une partie différente de la mémoire à l'intérieur du même processus, sans qu'un vrai fork() ait lieu.

Pour faire bon usage des threads, cependant, on veut généralement partager des données entre eux. On peut le faire grâce au module *threads::shared* et à l'attribut `: shared` (*partagé*).

```
use threads;
use threads::shared;

my $toto : shared = 1;
my $tata = 1;
threads->new(sub { $toto++; $tata++ })->join;

print "$toto\n"; # affiche 2 car $toto est partagé
print "$tata\n"; # affiche 1 car $tata n'est pas partagé
```

Dans le cas d'un tableau partagé, tous les éléments du tableau sont partagés, et pour une table de hachage partagée, toutes les clés et les valeurs sont partagées. Cela place des restrictions sur ce qui peut être affecté à des éléments de tableaux et de tables de hachage partagés : seules des valeurs simples ou des références à des variables partagées sont autorisées - de façon à ce qu'une variable privée ne puisse accidentellement devenir partagée. Un affectation incorrecte entraîne la mort du thread (*die*). Par exemple :

```
use threads;
use threads::shared;

my $var = 1;
my $svar : shared = 2;
my %hash : shared;

... créer quelques threads ...

$hash{a} = 1;      # pour tous les threads, exists($hash{a}) et $hash{a} == 1
$hash{a} = $var;   # ok - copie par valeur : même effet que précédemment
$hash{a} = $svar;  # ok - copie par valeur : même effet que précédemment
$hash{a} = \$svar; # ok - référence à une variable partagée
$hash{a} = \$var;  # entraîne la terminaison (I<die>)
delete $hash{a};   # ok - pour tous les threads, !exists($hash{a})
```

Remarquez qu'une variable partagée garantit que si deux threads ou plus essaient de la modifier au même moment, son état interne ne sera pas corrompu. Cependant, il n'y a pas de garantie au-delà de celle-ci, comme cela est expliqué dans la prochaine section.

10.2 Pièges des threads : *race conditions*

Note de traduction : nous utilisons le terme anglais *race condition* pour désigner un cas où le comportement d'un programme dépend de l'ordre d'évènements particuliers, alors que cet ordre ne peut pas être garanti.

Les threads apportent des outils très utiles, mais amènent aussi leur lot de pièges. L'un de ces pièges est la race condition.

```
use threads;
use threads::shared;

my $a : shared = 1;
$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub2);

$thr1->join;
$thr2->join;
print "$a\n";

sub sub1 { my $toto = $a; $a = $toto + 1; }
sub sub2 { my $tata = $a; $a = $tata + 1; }
```

Quelle sera la valeur de $a, d'après vous ? Par chance, la réponse est "ça dépend". sub1() et sub2() accèdent toutes les deux à la variable globale $a, une fois en lecture et une fois en écriture. Selon différents facteurs comme l'algorithme d'ordonnancement de votre implémentation des threads ou la phase de la lune, $a peut être 2 ou 3.

Les race conditions ont pour cause un accès non synchronisé à des données partagées. Sans synchronisation explicite, il n'y a aucune façon d'être sûr que rien n'arrive aux données partagées entre le moment où vous les lisez et celui où vous les modifiez. Même un bout de code aussi simple que l'exemple suivant présente un problème :

```
use threads;
my $a : shared = 2;
my $b : shared;
my $c : shared;
my $thr1 = threads->create(sub { $b = $a; $a = $b + 1; });
my $thr2 = threads->create(sub { $c = $a; $a = $c + 1; });
$thr1->join;
$thr2->join;
```

Deux threads accèdent à $a. Chacun peut potentiellement être interrompu n'importe quand, et ils peuvent être exécutés dans n'importe quel ordre. A la fin, $a peut valoir 3 ou 4, et $b et $c peuvent chacun valoir 2 ou 3.

Même pour `$a += 5` et `$a++`, l'atomicité n'est pas garantie.

A chaque fois que votre programme accède à des données ou des ressources auxquelles peuvent aussi accéder d'autres threads, vous devez prendre des mesures pour assurer la coordination des accès, ou courir le risque de vous retrouver avec des données inconsistantes ou des race conditions. Remarquez que Perl protège son état interne contre vos race conditions, mais il ne vous protège pas de vous-même.

11 Synchronisation et contrôle

Perl fournit plusieurs mécanismes pour coordonner les interactions entre les threads et leurs données, et pour éviter les race conditions. Certains ressemblent aux techniques communes des bibliothèques de gestion des threads, comme `pthreads` ; d'autres sont spécifiques à Perl. Le plus souvent, les techniques standards (comme les attentes de condition) sont maladroites et difficiles à mettre en place correctement. Quand c'est possible, il est généralement plus facile d'utiliser les procédés perliens comme les files d'attente, qui se chargent d'une partie du travail pénible.

11.1 Contrôler l'accès : lock()

La fonction lock() prend une variable partagée et la verrouille. Aucun autre thread ne peut la verrouiller jusqu'à ce que la variable ait été déverrouillée par le thread qui a posé le verrou. Le déverrouillage arrive automatiquement quand le thread qui tient le verrou sort du premier bloc extérieur qui contient l'appel à la fonction `lock()`. L'utilisation de lock() est simple ; l'exemple suivant fait faire des calculs en parallèle à plusieurs threads, en mettant à jour un total courant de temps en temps :

```
use threads;
use threads::shared;

my $total : shared = 0;

sub calc {
    for (;;) {
        my $resultat;
        # (... faire des calculs et affecter à $resultat ...)
        {
            lock($total); # bloque jusqu'à obtenir le verrou
            $total += $resultat;
        } # le verrou est automatiquement relâché à la sortie du bloc
        last if $resultat == 0;
    }
}

my $thr1 = threads->new(\&calc);
my $thr2 = threads->new(\&calc);
my $thr3 = threads->new(\&calc);
$thr1->join;
$thr2->join;
$thr3->join;
print "total=$total\n";
```

lock() bloque le thread jusqu'à ce que la variable à verrouiller soit disponible. Quand lock() retourne, votre thread peut être sûr qu'aucun autre ne peut verrouiller la variable jusqu'à la fin du premier bloc extérieur contenant l'appel à lock().

Il est important de noter que les verrous n'empêchent pas l'accès à la variable en question, mais seulement les tentatives de verrouillage. Cela s'inscrit dans la longue amitié qu'à Perl avec la programmation bien élevée, ainsi que le verrouillage consultatif (*advisory locking*) de fichiers que flock() permet.

Vous pouvez verrouiller des tableaux et des tables de hachage, aussi bien que des scalaires. Cependant, le verrouillage d'un tableau ne bloque pas d'éventuels verrouillages ultérieurs sur ses éléments, mais juste ceux sur le tableau lui-même.

Les verrous sont récursifs, ce qui veut dire qu'un thread peut verrouiller une variable plus d'une fois. Le verrou durera jusqu'à ce que le lock() le plus extérieur sur la variable arrive au bout de sa portée. Par exemple :

```
my $x : shared;
fais();

sub fais {
    {
        {
            lock($x); # attendre le verrouillage
            lock($x); # NOOP - nous avons déjà le verrou
            {
                lock($x); # NOOP
                {
                    lock($x); # NOOP
                    verrouille_le_encore();
                }
            }
        } # *** déverrouillage implicite ici ***
    }
}
```

```
sub verrouille_le_encore {
    lock($x); # NOOP
} # rien n'arrive ici
```

Notez qu'il n'y a pas de fonction unlock() - la seule façon de déverrouiller une variable est de laisser le verrou sortir de sa portée.

Un verrou peut être utilisé pour préserver les données contenues dans la variable verrouillée, mais il peut aussi servir à garder autre chose, comme une section de code. Dans ce cas, la variable verrouillée ne contient pas de données utiles, et n'existe que pour être verrouillée. La variable se comporte alors comme une mutex ou un sémaphore simple des bibliothèques traditionnelles de gestion des threads.

11.2 Un piège des threads : interblocages (*deadlocks*)

Les verrous sont un outil pratique pour synchroniser les accès aux données, et leur bon usage est la clé de la sûreté des données partagées. Malheureusement, ils ne sont pas sans danger, en particulier quand plusieurs verrous sont en jeu. Contemplez le code suivant :

```
use threads;

my $a : shared = 4;
my $b : shared = "toto";
my $thr1 = threads->new(sub {
    lock($a);
    threads->yield;
    sleep 20;
    lock($b);
});
my $thr2 = threads->new(sub {
    lock($b);
    threads->yield;
    sleep 20;
    lock($a);
});
```

Ce programme va probablement se bloquer jusqu'à ce que vous le tuiez. La seule façon qu'il ne se bloque pas est qu'un des deux threads acquière les deux verrous en premier. Une version garantie de bloquer serait plus compliquée, mais le principe serait le même.

Le premier thread va acquérir un verrou sur $a, puis, après une pause durant laquelle le second thread a probablement eu le temps de travailler, essayer de verrouiller $b. Pendant ce temps, le second thread acquiert un verrou sur $b, puis plus tard essaie de verrouiller $a. La deuxième tentative de verrouillage pour les deux threads va bloquer, chacun attendant que l'autre relâche son verrou.

Cette situation est appelée un interblocage, et elle arrive dès que deux threads ou plus essaient d'obtenir un verrou sur des ressources que les autres ont verrouillées. Chaque thread va bloquer, en attendant que l'autre relâche un verrou sur une ressource. Cependant, cela n'arrive jamais puisque le thread qui détient la ressource est lui-même en train d'attendre qu'un autre verrou soit relâché.

Il y a plusieurs façon de régler ce genre de problèmes. La meilleure est de s'assurer que tous les threads acquièrent les verrous dans le même ordre. Si, par exemple, vous verrouillez les variables $a, $b et $c, verrouillez toujours $a avant $b, et $b avant $c. Il est aussi préférable de ne garder les verrous que pendant une courte période de temps pour minimiser les risques d'interblocage.

Les autres primitives de synchronisation décrites plus bas peuvent souffrir de problèmes similaires.

11.3 Files d'attente (*queues*) : transmettre des données

Une file d'attente est un objet spécial qui vous permet d'enfourner des données d'un côté et de les récupérer de l'autre, sans avoir à vous inquiéter des problèmes de synchronisation. C'est un animal assez simple, qui ressemble à ceci :

```
use threads;
use Thread::Queue;
```

```
my $File = Thread::Queue->new;
$thr = threads->new(sub {
    while ($Element = $File->dequeue) {
        print "Retiré $Element de la file\n";
    }
});

$File->enqueue(12);
$File->enqueue("A", "B", "C");
$File->enqueue(\$thr);
sleep 10;
$File->enqueue(undef);
$thr->join;
```

Vous pouvez créer une file d'attente avec `new Thread::Queue`. Ensuite, vous pouvez y empiler des listes de scalaires avec enqueue(), et en retirer des scalaires de l'autre côté avec dequeue(). Une file n'a pas de taille fixe, et peut grandir à volonté pour loger tout ce qui y est poussé.

Si une file est vide, dequeue() bloque jusqu'à ce qu'un autre thread y pousse quelque chose. Cela fait des files l'élément idéal pour les boucles d'évènements et les autres sortes de communication entre threads.

11.4 Sémaphores : synchroniser les accès aux données

Les sémaphores sont un genre de mécanisme de verrouillage générique. Dans leur forme la plus basique, ils se comportent tout à fait comme des scalaires munis d'un verrou, sauf qu'ils ne peuvent pas contenir de données, et doivent être explicitement déverrouillés. Dans leur forme avancée, ils agissent comme un genre de compteur, et peuvent autoriser plusieurs threads à détenir le "verrou" à un instant donné.

11.5 Sémaphores de base

Les sémaphores ont deux méthodes, down() et up() : down() décrémente le compteur de ressource, up() l'incrémente. Les appels à down() bloqueront si le compteur courant du sémaphore devait descendre sous zéro pour effectuer le down(). Ce programme en donne une rapide démonstration :

```
use threads qw(yield);
use Thread::Semaphore;

my $semaphore = new Thread::Semaphore;
my $VariableGlobale : shared = 0;

$thr1 = new threads \&sub_exemple, 1;
$thr2 = new threads \&sub_exemple, 2;
$thr3 = new threads \&sub_exemple, 3;

sub sub_exemple {
    my $NoSub = shift @_;
    my $NbTentatives = 10;
    my $CopieLocale;
    sleep 1;
    while ($NbTentatives--) {
        $semaphore->down;
        $CopieLocale = $VariableGlobale;
        print "$NbTentatives essais restants pour la sub"
         . " $NoSub (\$VariableGlobale est $VariableGlobale)\n";
        yield;
        sleep 2;
        $CopieLocale++;
        $VariableGlobale = $CopieLocale;
        $semaphore->up;
    }
}
```

```
$thr1->join;
$thr2->join;
$thr3->join;
```

Les trois appels de la fonction opèrent tous en même temps. Le sémaphore, cependant, s'assure qu'il n'y a qu'un thread à la fois qui accède à la variable globale.

11.6 Sémaphores avancés

Par défaut, les sémaphores se comportent comme des verrous, et laissent un seul thread à la fois décrémenter leur compteur avec down(). On peut cependant en faire d'autres usages.

Chaque sémaphore a un compteur attaché. Par défaut, les sémaphores sont créés avec un compteur mis à un, down() décrémente ce compteur d'une unité, et up() l'incrémente d'une unité. Nous pouvons néanmoins utiliser d'autres valeurs de la façon suivante :

```
use threads;
use Thread::Semaphore;
my $semaphore = Thread::Semaphore->new(5);
                # Crée un sémaphore avec le compteur initialisé
                # à cinq

$thr1 = threads->new(\&sub1);
$thr2 = threads->new(\&sub1);

sub sub1 {
    $semaphore->down(5); # Décrémente le compteur de cinq unités
    # Faire des choses ici
    $semaphore->up(5); # Incrémente le compteur de cinq unités
}

$thr1->detach;
$thr2->detach;
```

Si down() essaie de décrémenter le compteur au-dessous de zéro, il bloque jusqu'à ce que le compteur devienne assez grand. Notez que bien qu'un sémaphore puisse être créé avec un compteur initial de zéro, tout up() ou down() change toujours le compteur d'au moins une unité ; par conséquent, $semaphore->down(0) est identique à $semaphore->down(1).

La question, bien sûr, est pourquoi voudrait-on faire ça ? Pourquoi créer un sémaphore avec un compteur initial qui n'est pas égal à un, ou pourquoi décrémenter/incrémenter par plus d'une unité ? La réponse tient dans la disponibilité des ressources. De nombreuses ressources auxquelles on veut gérer l'accès peuvent être utilisées par plusieurs threads en même temps en toute sécurité.

Par exemple, considérons un programme centré autour d'une interface graphique. Il utilise un sémaphore pour synchroniser l'accès à l'affichage, de manière à ce qu'un seul thread à la fois puisse dessiner. C'est pratique, mais bien sûr vous ne voulez pas qu'un thread commence à dessiner avant que tout ne soit mis en place. Dans ce cas, vous pouvez créer un sémaphore avec un compteur initial de zéro, et l'incrémenter quand tout est prêt pour le dessin.

Les sémaphores avec un compteur plus grand que un sont utiles pour établir des quotas. Supposons, par exemple, que vous avez plusieurs threads qui peuvent faire des E/S en parallèle. Vous ne voulez pas que tous lisent et écrivent en même temps, car cela pourrait encombrer vos canaux d'E/S, ou épuiser le quota de descripteurs de fichiers de votre processus. Vous pouvez utiliser un sémaphore initialisé au nombre de requêtes d'E/S (ou d'ouvertures de fichiers) concurrentes que vous voulez autoriser à un instant donné, et laisser tranquillement vos threads se bloquer et se débloquer les uns les autres.

Les incréments et décréments plus grands que un sont utiles quand un thread doit libérer ou accéder à plusieurs ressources à la fois.

11.7 cond_wait() et cond_signal()

Ces deux fonctions peuvent être utilisées en conjonction avec les verrous pour notifier des threads qui coopèrent qu'une ressource est devenue disponible. Elles sont très semblables aux fonctions qu'on peut trouver dans `pthreads`. Mais dans la plupart des cas, les files d'attente sont plus simples à utiliser et plus intuitives. Voyez *threads::shared* pour plus de détails.

12 Fonctions utiles générales

Nous avons couvert les parties principales du système de gestion des threads de Perl, et avec ces outils vous devriez être bien équipé pour écrire du code et des modules utilisant les threads. Mais il reste quelques petits éléments que nous n'avons pas abordés.

12.1 Dans quel thread suis-je ?

La méthode de classe `threads->self` fournit à votre programme une façon d'obtenir un objet qui représente le thread dans lequel il s'exécute. Vous pouvez utiliser cet objet de la même façon que ceux renvoyés par les fonctions de création de threads.

12.2 Identificateur de thread

tid() est une méthode d'objet sur les threads qui renvoie l'identificateur du thread que l'objet représente. Les identificateurs sont des entiers, et celui du thread principal d'un programme est 0. Actuellement, Perl affecte un tid unique à chaque thread créé dans votre programme ; le premier thread crée reçoit un tid de 1, puis chaque nouveau thread reçoit un tid incrémenté de 1.

12.3 Est-ce que ces threads sont les mêmes ?

La méthode equal() prend deux objets thread et renvoie vrai si les objets représentent le même thread, faux sinon.

La comparaison avec == est surchargée pour les objets threads, et vous pouvez donc comparer ceux-ci comme vous le faites pour les objets ordinaires.

12.4 Quels threads sont en train de tourner ?

`threads->list` renvoie une liste d'objets threads, un pour chaque thread actuellement en train de tourner et non détaché. C'est utile à plusieurs fins, dont le nettoyage à la fin de votre programme :

```
# Parcourir la liste de tous les threads
foreach $thr (threads->list) {
    # Ne pas rejoindre le thread principal ni nous-mêmes
    if ($thr->tid && !threads::equal($thr, threads->self)) {
        $thr->join;
    }
}
```

Si certains threads n'ont pas fini de tourner quand le thread Perl principal se termine, Perl vous avertit et meurt, puisqu'il est impossible à Perl de sortir proprement alors que d'autres threads tournent encore.

13 Un exemple complet

Déjà embrouillé ? Il est temps qu'un exemple illustre une partie de ce que nous avons exposé. Ce programme trouve des nombres premiers en utilisant des threads.

```
1  #!/usr/bin/perl -w
2  # prime-pthread, avec l'aimable autorisation de Tom Christiansen
3
4  use strict;
5
6  use threads;
7  use Thread::Queue;
8
9  my $flot = new Thread::Queue;
10 my $enfant = new threads(\&teste_nombre, $flot, 2);
11
```

```
12 for my $i ( 3 .. 1000 ) {
13      $flot->enqueue($i);
14 }
15
16 $flot->enqueue(undef);
17 $enfant->join;
18
19 sub teste_nombre {
20      my ($amont, $premier_courant) = @_;
21      my $enfant;
22      my $aval = new Thread::Queue;
23      while (my $nombre = $amont->dequeue) {
24           next unless $nombre % $premier_courant;
25           if ($enfant) {
26                $aval->enqueue($nombre);
27           } else {
28                print "Ai trouvé le premier $nombre\n";
29                $enfant = new threads(\&teste_nombre, $aval, $nombre);
30           }
31      }
32      $aval->enqueue(undef) if $enfant;
33      $enfant->join if $enfant;
34 }
```

Ce programme utilise le modèle de travail à la chaîne pour générer des nombres premiers. Chaque thread dans la chaîne a une file d'entrée qui fournit des nombres à tester, un nombre premier dont il est responsable, et une file de sortie dans laquelle il enfile les nombres qui ont raté son test (de non primauté). Si le thread tombe sur un nombre qui a raté le test et qu'il n'y a pas de thread enfant, c'est qu'il doit avoir trouvé un nouveau nombre premier. Dans ce cas, un nouveau thread enfant est créé pour ce premier, et accolé au bout de la chaîne.

Cela sonne probablement un peu plus difficile que cela ne l'est vraiment ; passons donc en revue ce programme morceau par morceau, et examinons ce qu'il fait. (Pour ceux qui essaient de se rappeler ce qu'est au juste un nombre premier, c'est un nombre qui est seulement divisible par 1 et lui-même.)

Le gros du travail est fait par la fonction teste_nombre, qui prend une référence à sa file d'entrée et un nombre premier dont elle est responsable. Après avoir extrait la file et le nombre premier pour lequel elle fait les tests (ligne 20), nous créons une nouvelle file (ligne 22) et nous réservons un scalaire pour le thread que nous allons sans doute créer plus loin (ligne 21).

La boucle while de la ligne 23 à la ligne 31 retire un scalaire de la file d'entrée et le teste avec le nombre premier dont le thread est responsable. La ligne 24 détermine s'il y a un reste quand nous calculons le nombre à tester modulo notre premier. S'il y en a un, le nombre n'est pas divisible par notre premier, et nous devons donc soit le passer au thread suivant si nous en avons créé un (ligne 26) soit créer un nouveau thread si ce n'est pas encore fait.

La création du nouveau thread se trouve ligne 29. Nous passons une référence à la file que nous avons créée et le nombre premier que nous avons trouvé.

Enfin, une fois que la boucle se termine (parce que nous avons trouvé un 0 ou un undef dans la file, qui sert comme un ordre de terminer), nous notifions notre enfant et attendons qu'il sorte si nous en avons créé un (lignes 32 et 37).

Pendant ce temps, dans le thread principal, nous créons une file (ligne 9) et le thread enfant initial (ligne 10), et nous lui passons le premier nombre premier : 2. Ensuite, nous enfilons tous les nombres de 3 à 1000 pour qu'il soient testés (lignes 12-14), puis une notification de terminer (ligne 16), et nous attendons que le premier thread enfant finisse (ligne 17). Nous savons que tout va bien quand nous retournons du join(), puisqu'un enfant ne mourra pas avant que ses propres enfants ne soient morts.

Voila tout. C'est très simple ; comme beaucoup de programmes Perl, l'explication est bien plus longue que le programme.

14 Considérations de performance

La chose la plus importante à garder à l'esprit quand on compare les ithreads à d'autres modèles de gestion des threads est le fait que pour chaque thread créé, une copie complète de toutes les variables et données du thread parent doive être faite. La création d'un thread peut donc être coûteuse en termes de mémoire autant que de temps de calcul. La meilleure façon de réduire ces coûts est d'avoir un faible nombre de threads qui vivent longtemps, tous créés assez tôt - avant que le thread de base n'ait accumulé trop de données en mémoire. Bien sûr, après qu'un thread a été créé, ses performances et son occupation mémoire devraient être peu différente que celles de code ordinaire.

Notez aussi que dans l'implémentation actuelle, les variables partagées prennent un peu plus de mémoire et sont légèrement plus lentes que les variables ordinaires.

15 Changements au niveau du processus

Bien que les threads aient des chemins d'exécution séparés et que les données Perl soient privées à chaque thread à moins qu'elles ne soient explicitement partagées, les threads peuvent modifier l'état du processus complet, affectant ainsi les autres threads.

L'exemple évident de cela est le changement du répertoire de travail courant avec chdir(). Si un thread appelle chdir(), le répertoire de travail de tous les threads change.

Un exemple encore plus spectaculaire d'un changement qui intervient au niveau du processus est chroot() : le répertoire racine de tous les threads change alors, et aucun thread ne peut revenir en arrière (par opposition à chdir()).

umask() et les changements d'uid et de gid sont d'autres exemples de changement qui ont lieu au niveau du processus.

Si jamais l'envie vous prenait de mélanger fork() et les threads, allongez-vous et attendez que ça passe. Mais si vous voulez vraiment savoir, la sémantique est que fork() duplique tous les threads (sous UNIX du moins - les autres plateformes feront quelque chose de différent).

De même, vous ne devriez pas essayer de mélanger les signaux et les threads. Les implémentations dépendent de la plateforme, et même la sémantique POSIX pourrait vous surprendre (sans compter que Perl ne vous fournit pas l'API POSIX complète).

16 Réentrance des bibliothèques système

La réentrance des différents appels système est hors du contrôle de Perl. Parmi les appels connus pour ne pas être réentrants, on trouve : localtime(), gmtime(), get{gr,host,net,proto,serv,pw}*(), readdir(), rand(), et srand(), et en général, tous les appels qui dépendent d'un état externe global.

Si le système sur lequel perl est compilé a des versions réentrantes de ces fonctions, elles seront utilisées. A part ça, Perl est à la merci de la réentrance ou de la non-réentrance de ces fonctions. Consultez la documentation de votre bibliothèque C pour plus de détails.

Sur certaines plateforme, l'interface réentrante peut échouer si le tampon de résultat est trop petit (par exemple getgrent() peut renvoyer d'assez volumineuses listes de membres). Perl réessaie alors un certain nombre de fois en agrandissant le tampon, mais jusqu'à 64ko seulement (pour des raisons de sécurité).

17 Conclusion

Un tutoriel complet sur les threads pourrait remplir un livre entier (en fait, de nombreux livres ont pour seul objet ce type de tutoriel), mais avec ce que nous avons couvert dans cette introduction, vous devriez être en bonne voie pour devenir un expert de la programmation avec les threads en Perl.

18 Bibliographie

Voici une courte bibliographie, aimablement fournie par Jürgen Christoffel.

18.1 Textes introductifs

Birrell, Andrew D. An Introduction to Programming with Threads. Digital Equipment Corporation, 1989, DEC-SRC Research Report #35 en ligne sur http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-035.html (hautement recommandé)

Robbins, Kay. A., et Steven Robbins. Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading. Prentice-Hall, 1996.

Lewis, Bill, et Daniel J. Berg. Multithreaded Programming with Pthreads. Prentice Hall, 1997, ISBN 0-13-443698-9 (une introduction aux threads très bien écrite).

Nelson, Greg (sous la direction de). Systems Programming with Modula-3. Prentice Hall, 1991, ISBN 0-13-590464-1.

Nichols, Bradford, Dick Buttlar, et Jacqueline Proulx Farrell. Pthreads Programming. O'Reilly & Associates, 1996, ISBN 156592-115-1 (couvre les threads POSIX).

18.2 Références liées aux systèmes d'exploitation

Boykin, Joseph, David Kirschen, Alan Langerman, et Susan LoVerso. Programming under Mach. Addison-Wesley, 1994, ISBN 0-201-52739-1.

Tanenbaum, Andrew S. Distributed Operating Systems. Prentice Hall, 1995, ISBN 0-13-219908-4 (très bon manuel).

Silberschatz, Abraham, et Peter B. Galvin. Operating System Concepts, 4th ed. Addison-Wesley, 1995, ISBN 0-201-59292-4

18.3 Autres références

Arnold, Ken et James Gosling. The Java Programming Language, 2nd ed. Addison-Wesley, 1998, ISBN 0-201-31006-6.

FAQ de comp.programming.threads, http://www.serpentine.com/~bos/threads-faq/

Le Sergent, T. et B. Berthomieu. "Incremental MultiThreaded Garbage Collection on Virtually Shared Memory Architectures" in Memory Management : Proc. of the International Workshop IWMM 92, St. Malo, France, September 1992, Yves Bekkers and Jacques Cohen, eds. Springer, 1992, ISBN 3540-55940-X (applications pratiques des threads).

Arthur Bergman, "Where Wizards Fear To Tread", June 11, 2002, http://www.perl.com/pub/a/2002/06/11/threads.html

19 Crédits

Merci (sans ordre particulier) à Chaim Frenkel, Steve Fink, Gurusamy Sarathy, Ilya Zakharevich, Benjamin Sugars, Jürgen Christoffel, Joshua Pritikin, et Alan Burlison, pour leur aide dans la vérification et la mise au point de cet article. Un grand merci à Tom Christiansen pour sa réécriture du générateur de nombres premiers.

20 AUTEUR

Dan Sugalski (dan@sidhe.org).

Légèrement modifié par Arthur Bergman pour ajuster l'article au nouveau modèle de gestion des threads.

Légèrement retravaillé par Jörg Walter <jwalt@cpan.org> pour être plus concis sur le sujet de la réentrance du code Perl.

21 Copyrights

22 Copyright

Pour plus d'informations, voyez *threads* et *threads::shared*.

23 TRADUCTION

23.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

23.2 Traducteur

Ronan Le Hy <rlehy@free.fr>.

23.3 Relecture

Personne pour l'instant.

24 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.