R305 Programmation Système: Thread

Notes de cours

Abdelkader Gouaïch

2022

Introduction

Définitions

Programmes, Processus et Thread

En programmation système, nous utiliserons le langage `C` pour exprimer nos programmes.

Ces programmes sont compilés pour produire des fichiers binaires qui codent nos instructions dans un format compréhensible par l'unité de traitement: *le processeur (CPU)*.

Nous avons vu dans le chapitre précédent que le processus est l'activité qui résulte de l'exécution d'un programme sur un processeur.

Un processus va contenir un ou plusieurs *threads*. Dans le cas où un processus utilise plus d'un thread, on parlera alors d'un processus *multithread*.

Les threads sont les *vraies* unités qui permettent d'interpréter les instructions des fonctions sur le processeur et le système d'exploitation nous propose une vision dans laquelle chacun de ces threads dispose d'une CPU virtuelle qui lui permet d'interpréter ses instructions.

C'est le rôle du système d'exploitation de maintenir cette vision et de faire croire au programmeur que plusieurs CPUs sont disponibles, même si, en réalité, il n'existe qu'une seule CPU physique.

Mémoire Virtuelle et Processeur Virtuel

Linux offre au programmeur un ordinateur virtuel qui dispose d'une mémoire virtuelle et de multiples processeurs virtuels.

La mémoire virtuelle

Le processus croit qu'il dispose d'une mémoire contigue de taille plus importante que la mémoire physique (RAM). Cette mémoire n'est qu'une construction virtuelle réalisée par le système d'exploitation grâce au mécanisme de pagination:

— la mémoire est organisée en page
— les pages sont créées au besoin et ensuite chargées/déchargées de la RAM vers le disque dure au besoin

Le processeur virtuel

Le processus croit qu'il est seul à disposer du processeur pour exécuter ses instructions. En réalité, plusieurs processus partagent le processeur. Avec un algorithme d'ordonnancement, scheduling, le processeur est occupé successivement par les processus. Les opérations de chargement et déchargement du processeur sont transparentes pour le processus. Cela veut dire aussi que le processus ne contrôle pas à quel moment il peut occuper ou perdre le processeur, nous parlons alors de préemption. Linux est certes préemptif mais garantit au processus que son état sera restitué intégralement à son retour. Les threads vont aussi avoir l'illusion de disposer de plusieurs processeurs virtuels au sein du processus. Chaque thread va donc s'exécuter indépendamment dans son propre processeur virtuel. Encore une fois, si nous disposons que d'un seul processeur physique (avec un coeur d'exécution) alors un algorithme de scheduling de thread va permettre aux différents threads d'un même processus de partager le processeur physique. Ce partage du processeur physique est transparent pour les threads du moment où l'état du processeur est restitué intégralement.

Synthèse

Avec les abstractions de la mémoire virtuelle et des processeurs virtuelles l'OS nous offre une machine:

— chaque processus dispose de son propre segment mémoire virtuelle de taille importante (32G ou 64G)
— chaque processus dispose de son processeur virtuel pour exécuter ses instructions
— dans le cas multithread, chaque thread dispose de son propre processeur virtuelle pour exécuter ses instructions
— les threads partagent le segment de mémoire virtuelle associé à leur processus

Le multithreading

Quels sont les avantages des threads ?

Ce n'est pas une question dénuée de sens, car la programmation multithread est plus complexe que la programmation monothread. Nous devons alors justifier et bien comprendre les avantages des threads pour les utiliser dans une application. Parmi les avantages des threads nous pouvons citer les suivants.

Un design pattern

Les solutions à certains problèmes peuvent s'exprimer naturellement comme des traitements indépendants. Dans ce cas, nous pourrions associer à chaque thread un traitement particulier et la solution à notre problème vient de la coopération entre les différents threads. Prenons l'exemple d'un serveur web qui reçoit les requêtes de différents clients. Nous pouvons modéliser cela facilement en associant à chaque client une activité qui va le prendre en charge et répondre à toutes ses requêtes. Ainsi, il serait naturel d'associer à chaque session d'un client un thread qui va la traiter. Les threads peuvent soit être créés pour chaque nouvelle session ou bien être tirés d'un pool de thread fixe. La deuxième solution à l'avantage de limiter et de contrôler le nombre de thread mais va introduire de l'attente chez les clients si le nombre de sessions, à un instant donné, est plus important que la taille du pool de threads.

Améliorer l'interactivité

Imaginons un programme qui propose une interface graphique. Dans le cas monothread, cette interface va se figer à chaque opération d'entrée/sortie ou à chaque traitement algorithmique complexe. En effet, pendant que notre activité effectue des calculs, elle ne peut pas rafraîchir les éléments de l'interface graphique. Nous pouvons facilement imaginer la frustration créée par une telle application chez les utilisateurs. Les threads vont apporter une solution, à ce problème: un thread va présenter l'interface graphique et un autre thread va s'occuper des traitements algorithmiques. L'utilisateur pourra toujours interagir avec son interface pendant que les traitements seront exécutées par le thread des traitements.

Le parallélisme

Sur machine qui dispose de plusieurs unités de calcul, nous pouvons exécuter les threads avec un vrai parallélisme. Dans ce cas, un gain de temps important est possible. Mais attention ce gain de temps n'est pas automatique et dépend aussi des mécanismes de partage d'information entre les threads. En effet, les

dépendances entre threads sur les données vont créer de l'attente qui peut réduire le bénéfice de parallélisme.

Partage des données

Les processus ne partagent pas leurs données contrairement aux threads. Il serait alors plus judicieux de réaliser les activités avec des threads qui vont partager le segment mémoire qu'avec des processus qui vont utiliser les IPCs, plus lourd, pour partager les données.

Changement de contextes

Le partage du processeur entre les différents processus entraîne des coûts pour le chargement et la sauvegarde de l'état de chaque processus. Ces coûts peuvent détériorer les performances de façon significative si nous avons beaucoup de processus en cours d'exécution. Le changement de contexte pour les threads est beaucoup plus léger que celui entre les processus. Nous avons moins d'informations à sauvegarder/charger pour un thread car tous les threads vont partager la mémoire virtuelle.

Alternatives aux threads

Introduction

Les threads souffrent aussi de certains inconvénients que nous devons comprendre. Le premier concerne la difficulté à tester et s'assurer que notre application multithread ne comporte pas de bugs. En effet, le parallélisme des threads et l'algorithme d'ordonnancement, que nous ne contrôlons pas, nous font perdre une propriété importante pour certifier la qualité de notre application: le déterminisme. Le déterminisme est la propriété qui nous permet d'avoir exactement la même séquence d'instructions et donc le même résultat à la fin pour un même jeu de données en entrée. Un algorithme classique séquentiel possède cette propriété et il est par conséquent possible de déterminer si une erreur se produit pour un jeu de données fixé. Avec le déterminisme, si une erreur se produit avec un jeu de données alors elle va toujours se reproduire pour ce même jeu de donnée. Et si l'exécution est correcte pour un jeu de données fixé, alors les résultats seront toujours justes pour ce jeu de données. Avec les multithread nous perdons cette propriété car nous avons un paramètre nouveau dans notre système : l'ordonnancement des threads. Ceci a comme conséquence que pour un jeu de données fixe, il est possible qu'une séquence particulière de l'ordonnanceur produise une erreur et que cette erreur ne ne produit pas avec d'autres séquences. Nous appelons cela en anglais, race condition, et cela veut dire que le comportement de notre application dépend des conditions de course entre les threads. Ceci rend le debug

des applications multithread très complexe. Pour éviter ce problème, certains architectes vont éviter les threads et utiliser des alternatives qui vont maintenir le déterminisme.

Programmation évènementielle asynchrone

Ce modèle de programmation propose un seul thread qui va exécuter de petites unités de traitement. L'idée est de ne jamais avoir un blocage ou une attente active. Par exemple, si nous effectuons une opération E/S alors nous allons enregistrer une unité de traitement (fonction) qui sera exécutée ultérieurement quand un événement signalant la disponibilité des données E/S sera reçu. Pendant ce temps, d'autres unités de traitement seront exécutées. Ce style de programmation orientée événements, va utiliser des opérations asynchrones qui ne bloque jamais l'activité en attendant un résultat. L'attente d'un résultat est transformée comme l'enregistrement d'une fonction (callback) qui sera exécutée à la réception de d'un événement.

Co-routines

Ce modèle propose de ne pas utiliser d'ordonnanceur mais d'introduire de nouvelles primitives au langage de programmation pour libérer explicitement le processeur. Par exemple l'instruction 'yield' va permettre à une fonction de libérer le processeur. Mais quand cette fonction va retrouver le processeur, elle va le retrouver à partir de son dernier yield. Elle conserve ainsi son état et elle peut continuer son d'exécution. Les co-routines vont coopérer pour utiliser le processeur contrairement aux threads qui sont en compétition pour le processeur.

Modèle de programmation concurrente

Thread système et utilisateurs

Linux nous offre la possibilité de créer des threads systèmes. Ce sont des threads gérés explicitement par le noyau Linux.

Si nous disposons du plusieurs processeurs (multicore) alors ces threads vont être exécutés en 'vrai' parallélisme.

Mais nous pouvons aussi implémenter notre propre vision des threads. Nous devons alors gérer nous même l'ordonnancement sur les unités d'exécution. Dans ce cas, le noyau ne va voir d'un seul thread système.

Ce thread système va exécuter (simuler) plusieurs threads utilisateurs.

L'exemple que nous pouvons donner est celui des threads Java qui sont des threads utilisateur définis dans la machine virtuelle Java. Les threads Java partagent le processeur virtuel Java. Mais tout le processeur virtuel Java est simulé par un seul thread système Linux.

Design patterns

Nous allons présenter ici quelques patrons de conception à utiliser avec les threads. Un patron de conception est une règle métier/pattern que nous pouvons utiliser pour traiter des problèmes applicatifs.

Thread par connexion

Ce thread s'applique aux applications client-serveur. Nous allons affecter un thread pour s'occuper de la session d'un client. Nous avons dans ce cas une relation 1-1 entre les sessions des clients et les threads.

L'avantage est que les informations pour une session sont gérées et maintenues par un seul thread. Le thread termine avec la fin de la session associée.

Traitement événementiel

Dans ce pattern, nous allons définir une source d'événements. Un bassin (pool) de thread vont attendre l'arrivée des événements pour les consommer (traiter). Si la source n'est pas vide alors chaque thread va retirer un événement de la source et le traiter jusqu'à épuisement de la source. L'avantage de cette approche est que nous pouvons paramétrer la taille de notre pool de threads et que cela ne dépend pas des événements extérieures à notre application.

La synchronisation

Race condition

La programmation multithread est plus difficile que la programmation séquentielle car nous pourrions être dans des situations où un programme nous semble correct intuitivement si ses activités sont étudiées séparément.

Mais si les activités sont exécutées en parallèle alors des erreurs peuvent se produire.

Comme nous l'avons évoqué les threads sont en concurrence pour accéder au processeur. Nous pouvons les imaginer dans une sorte de course (race) et ce sont les conditions de cette course qui vont influencer le résultat final.

Nous perdons alors le déterminisme dans nos applications car nous ne contrôlons par les conditions de course.

Exemple:

```
int retirer (struct account *account, int montant)
{
    const int balance = account->balance; // (A)
    if (balance < montant) return -1;
    account->balance = balance - montant // (B);
    distribue_argent (montant);
    return 0;
}
```

Le code cei-dessus présente un gestion d'un compte bancaire. Ce code nous parait correct à première vue.

Supposons maintenant que ce code soit exécuté en parallèle pour gérer plusieurs transactions.

Nous réalisons deux achats au même instant, une première transaction de 200 euros et une deuxième transaction de 400 euros.

La première transaction va exécuter la ligne (A.1) et à cet instant la deuxième transaction récupère le processeur. Elle va exécuter également la même ligne (A.2).

La première transaction récupère le processeur et retire 200 de la variable balance et sauvegarde le nouveau montant du compte (B.1).

La transaction 2 fait de même et exécute (B.2).

En faisant cela, B.2 a écraser le travail réalisé par B.1 et nous nous retrouvons avec 200 euros retirés mais pas comptabilisés dans notre compte !

Il faut remarquer que si nous avions un ordre d'exécution A.1; B1 et ensuite A.2; B.2, aucun problème ne serait signalé.

Pour éviter que les race conditions influencent le résultat final de notre programme, nous allons introduire des mécanismes de synchronisation qui vont nous permettre d'avoir un peu plus de contrôle sur l'ordre, du moins partiel, d'exécution de nos threads.

Mutexe

Le mutex est un mécanisme qui permet de verrouiller une section du code. Ainsi la section verrouillée par une activité ne permet pas à une autre activité de l'exécuter. Elle sera bloquée en attendant la libération du verrou.

Nous avons deux primitives: lock() et unlock()

lock() Cette primitive permet à un thread de demander l'autorisation pour rentrer dans une section critique. Si le verrou est libre alors le thread continue son activité et prend le verrou. Si le verrou est fermé alors le thread suspend son activité en attendant sa libération.

unlock() Cette primitive permet de libérer un verrou et aussi de réveiller un thread suspendu qui était en attente.

Exemple

```
int retirer (struct account *account, int montant)
{
    lock();
    const int balance = account->balance; // (A)
    if (balance < montant) return -1;
    account->balance = balance - montant // (B);
    unlock(); distribue_argent (amount);
    return 0;
}
```

Avec le mécanisme de mutex, nous avons défini une section critique entre (A) et (B).

Cette section critique va nous garantir qu'un thread numéro n va toujours exécuter A.n ensuite B.n avant qu'un autre thread numéro m puisse réaliser A.m ensuite B.m

Avec ce mécanisme notre programme est correct et cela indépendamment des race conditions.

L'interblocage

L'utilisation des verrous peut aussi introduire des interblocages entre les threads. Il est tout à fait possible de prendre un verrou par exemple et de demander un autre verrou.

Il est facile alors de voir le problème: si une autre activité a récupéré le verrou que nous attendons et se trouve en attente du verrous que nous possédons !

Ce problème se généralise avec l'interblocage circulaire entre trois activités ou plus. Chacun des threads est en possession d'un verrou et en attente d'un autre.

API Thread

PThread API

Pthread offre une API complète pour la programmation multithread sous Linux.

Il faut noter que Pthread est une librairie Linux, elle ne fait pas partie de librairie standard C.

Nous devons dès lors informer notre compilateur que notre programme va utiliser cette libraire et qu'il est multithread.

Pour compiler un programme multithread vous devez exécuter la commande:

`gcc -pthread programme.c -o nomexecutable`

Le flag -pthread informe gcc que le programme est multithread et que la libraire Pthread sera utilisée lors de l'édition des liens.

Ce flag est très important car certaines fonctions disponibles en librairies `C` peuvent offrir deux implémentations:

- une implémentation monothread standard qui pourrait utiliser des variables globales sans se soucier des problèmes de synchronisation.
- une implémentation multithread qui prendra en charge les problèmes liés à l'utilisation de variables globales et synchronisation.

Le flag -pthread permettra de choisir la bonne version du code à importer.

Pthread est organisé en deux groupes de fonctions:

- le premier groupe est consacré au cycle de vie d'un thread (création, attente, destruction)
- le deuxième groupe est consacré à la synchronisation

Cycle de vie d'un thread

création

```
#include <pthread.h>
int pthread_create (pthread_t *thread,
                                          const pthread_attr_t *attr,
                                          void *(*start_routine) (void *),
                                          void *arg);
```

Cette fonction va créer une nouvelle activité qui va commencer son exécution par la fonction `start_routine`

La fonction `start_routine` aura comme argument le pointeur `void *arg`.

L'argument `const pthread_attr_t *attr` est une liste d'attributs que nous pouvons utiliser pour changer le comportement du nouveau thread. Nous pouvons par exemple changer la priorité du thread ainsi que la taille de sa pile. Si cette valeur vaut `NULL` alors les paramètres par défaut seront utilisés.

Après la création, la structure `pthread* thread` va récupérer des informations sur le nouveau thread comme par exemple son identifiant.

La fonction de démarrage d'un thread doit avoir une signature bien spécifique: `void* (start) (void)`

C'est donc une fonction qui prend un pointeur quelconque et qui retourne un pointeur quelconque.

En `C`, dire que notre pointeur et de type `void*` revient à ne pas poser de contrainte sur les valeurs que nous souhaitons. En effet, ce pointeur est une adresse qui peut tout désigner.

Pour être plus spécifique, nous devons utiliser un cast pour restreindre le type à certaines valeurs souhaitées par notre fonction.

Les valeurs de retour de la fonction `pthread_create` sont:

- un 0 sera retourné pour indiquer un succès de création du thread.
- en cas de problème une valeur différente de zéro est retournée et peut être:
  - `EAGAIN`: vous ne pouvez pas créer de nouveaux threads.
  - `EINVAL`: vous avez utilisé une valeur invalide pour un attribut
  - `EPERM`: vous cherchez à modifier un attribut et le processus n'a pas le droit suffisant pour le faire.

Exemple

```
pthread_t tread;
int ret;
ret = pthread_create (&thread, NULL, start_routine, NULL);
if (!ret) {
    errno = ret;
    perror("pthread_create");
    return -1;
}
```

Thread ID

Chaque thread au sein d'un processus va avoir un identifiant thread ID (TID)

Nous pouvons avoir le TID d'un thread que nous venons de créer avec le pointeur donné pour la fonction `pthread_create`.

Un thread peut aussi récupérer son TID avec la fonction `pthread_self`:

```
#include <pthread.h> pthread_t pthread_self (void);
```

Attention, le TID n'est pas nécéssairement un entier (comme pour PID)

Pour vérifier l'égalité entre deux valeurs nous devons utiliser la fonction pthread_equal

```
#include <pthread.h>
int pthread_equal (pthread_t t1, pthread_t t2);
```

Terminer un thread

Les cas de terminaison pour un thread sont les suivants:

- la fonction de départ du thread fait un `return`
- la fonction `pthread_exit()` est invoquée
- le thread recoit un événement `cancel` via la fonction `pthread_cancel`. Cet événement va venir d'un autre thread.

Terminer le thread par `exit()`

```
#include <pthread.h>
void pthread_exit (void *retval);
```

Cette fonction termine l'activité et la valeur `retval` sera sauvegardée et donnée au thread qui est en attente de terminaison par la fonction `pthread_join()`

Terminer un thread par cancel()

```
#include <pthread.h>
int pthread_cancel (pthread_t thread);
```

Nous pouvons considérer que cette fonction est analogue à envoyer un signal de terminaison à un thread.

Pour qu'un thread soit arrété par le mécanisme `cancel`, il doit mettre son flag *cancellable* à vrai. Ce qui est la valeur par défault pour tous les threads.

Si par contre un thread ne souhaite pas être arrété par `cancel` il peut changer l'état de son flag *cancellable* à faux.

```
#include <pthread.h>
int pthread_setcancelstate (int state, int *oldstate);
```

C'est la fonction `setcancelstate` qui permet de modifier l'état du flag cancellable pour le thread.

Join et Detach

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **retval);
```

L'appel à cette fonction permet de bloquer le thread appelant jusqu'à la terminaison du thread identifié par thread. Elle nous permet d'agir comme une barrière qui va attendre la fin d'un thread pour continuer.

Exemple

```
int ret;
ret = pthread_join (thread, NULL);
if (ret) {
    errno = ret;
    perror ("pthread_join");
    return -1;
    }
```

Detach

```
#include <pthread.h>
int pthread_detach (pthread_t thread);
```

Tous les threads sont joignables par défaut. Cela signifie que n'importe quel thread peut réaliser une opération join, attendre leur fin et récupérer leur résultat. La conséquence est que le noyau devra sauvegarder des informations sur tous les threads terminés en attendant un potentiel join.

La fonction detach informe le noyau que le thread en question ne peut pas être joint. Le noyau dans ce cas va libérer toutes les ressources du thread dès sa terminaison.

Pthread mutexe

Nous présentons maintenant les fonctions de gestion des mutexes.

initialisation

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Cette ligne permet d'initialiser un mutex en utilisant la macro `PTHREAD_MUTEX_INITIALIZER`

lock

```
#include <pthread.h>
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Cette fonction demande le verrou. Si le verrou est libre alors le thread continue son exécution et récupère le verrou pour lui.

Si le verrou est déjà pris par un autre thread, alors l'appel à cette fonction suspend l'activité du thread appelant jusqu'à la libération du verrou par la fonction `unlock`.

Voici les valeurs de retour:

- la valeur 0 est retourné si tout se passe bien.
- EDEADLK: ce code erreur indique que le verrou est déjà en posséssion du thread appelant.
- EINVAL: ce code erreur indique que la structure mutex n'est pas valide.

unlock

```
pthread_mutex_unlock (&mutex);
```

Cette fonction libère le verrou du mutex. Si des threads sont suspendus sur ce mutex alors ils seront réveillés et un seul peut récupérer le verrou et les autres seront suspendus de nouveau.

Voici les valeurs de retour: - la valeur 0 est retournée pour indiquer un fonctionnement normal - EINVAL: ce code erreur est retourné pour indiquer que la structure mutex n'est pas valide. - EPERM: ce code erreur indique que nous cherchons à libérer un verrou qui ne nous appartient pas.

Exemple

```
static pthread_mutex_t the_mutex = PTHREAD_MUTEX_INITIALIZER;
int withdraw (struct account *account, int montant)
{
    pthread_mutex_lock (&the_mutex);
    const int balance = account->balance;
    if (balance < montant)
       {
           pthread_mutex_unlock (&the_mutex);
           return -1;
       }
   account->balance = balance - montant;
   pthread_mutex_unlock (&the_mutex);
```

```
    distribue_argent (montant);
}
```