R305 Programmation Système: Thread

Abdelkader Gouaïch

2022

INTRODUCTION

Définitions

Programmes, Processus et Thread

Programme C => Programme binaire

Programme binaire + CPU => Processus

Processus = Thread**s**

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 dispose d'une mémoire contiguë

Taille plus importante que la mémoire physique (RAM).

Une construction réalisée par la **pagination**:

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

Mémoire Virtuelle et Processeur Virtuel

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

Le processeur virtuel (processus)

Le processus croit qu'il est seul à disposer du processeur

Un algorithme d'ordonnancement gère l'accès au processeur

Les opérations de chargement et déchargement du processeur sont transparentes pour le processus.

Linux est préemptif: le processus est déchargé à son insu.

Mémoire Virtuelle et Processeur Virtuel

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

Le processeur virtuel (threads)

Les threads vont aussi avoir l'illusion de disposer de plusieurs processeurs virtuels au sein du processus.

Un algorithme de scheduling de thread va partager le processeur physique.

Ce partage du processeur physique est transparent pour les threads.

Mémoire Virtuelle et Processeur Virtuel

Synthèse

L'OS nous offre une machine virtuelle:

- 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 ?

Justifier et bien comprendre les avantages des threads avant de les utiliser

Le multithreading

Avantage: un design pattern

Application demande des traitements indépendants => associer à chaque thread un traitement particulier de l'application

La solution à notre problème vient de la coopération entre les différents threads.

Le multithreading

Avantage: un design pattern

Exemple:

Un serveur web qui reçoit les requêtes de différents clients.

Solution 1:

- Associer à chaque client une activité qui va le prendre en charge et répondre à toutes ses requêtes.
- Les threads peuvent être créés pour chaque nouvelle session.

Solution 1.bis:

- Thread tirés d'un pool de thread fixe
- Avantage: limiter le nombre de threads

Le multithreading

Avantage: améliorer l'interactivité

Imaginons un programme avec une interface graphique.

Dans le cas monothread:

- cette interface va se figer (IO, traitement)
- elle ne peut pas se rafraîchir les éléments
- frustration !

Avec les threads:

- un thread va présenter l'interface graphique
- un autre thread va s'occuper des traitements algorithmiques.
- => l'utilisateur pourra toujours interagir avec son interface

Le multithreading

Avantage: le parallélisme

Plusieurs unités de calcul => un vrai parallélisme. Dans ce cas, un gain de temps important est possible.

Attention au partage des données et à la synchronisation!

Le multithreading

Avantage: partage des données

Les processus ne partagent pas les données

Les threads vont partager le segment mémoire de leur processus.

Partage des données est immédiat !

Mais attention à la synchronisation

Le multithreading

Avantage: changement de contextes

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

Les inconvénients des thread

Difficulté à tester et s'assurer que notre application multithread ne comporte pas de bugs.

Perte de déterminisme !

Nous avons un paramètre nouveau dans notre système : l'ordonnancement des threads.

Race Condition = Condition de course

Alternatives aux threads

Programmation événementielle asynchrone

Ce modèle de programmation propose un seul thread qui va exécuter de petites unités de traitement.

Des fonctions ‘callbacks’ vont attendre des événements

L’idée est de ne jamais avoir un blocage ou une attente active.

Alternatives aux threads

Co-routines

Ce modèle propose de ne pas utiliser d'ordonnanceur mais d'introduire de nouvelles primitives pour libérer explicitement le processeur.

L'instruction 'yield' va permettre à une fonction de libérer le processeur.
La fonction conserve son état et peut continuer son d'exécution.

Les co-routines vont **coopérer** pour utiliser le processeur contrairement aux threads qui sont en **compétition** pour le processeur.

Problème de synchronisation

Exemple

```
 int withdraw (struct account *account, int
montant)
 {

     const int balance = account->balance; //↙(A)
     if (balance < montant)
        {
            return -1;
        }
    account->balance = balance - montant; //↙(B)

    distribue_argent (montant);
}
```

API THREAD

PThread API

Pthread est une librairie Linux, elle ne fait pas partie de la librairie standard C.

Informer le compilateur que le programme est multithread.

Pour compiler un programme multithread:

```
gcc -pthread programme.c -o nomexecutable
```

PThread API

```
gcc -pthread programme.c -o nomexecutable
```

Le flag -pthread informe gcc que le programme est multithread

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

- une implémentation monothread sans synchronisation (variables globales)
- une implémentation multithread qui est thread-safe

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

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`

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);
```

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

L'argument `const pthread_attr_t *attr` est une liste d'attributs.

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.

Cycle de vie d'un thread

Exemple

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

Cycle de vie d'un thread

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);
```

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);
```

Cycle de vie d'un thread

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.

Cycle de vie d'un 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()`

Cycle de vie d'un thread

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 arrêté par le mécanisme `cancel`, il doit mettre son flag *cancellable* à vrai. Ce qui est la valeur par défaut pour tous les threads.

Si par contre un thread ne souhaite pas être arrêté 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.

Join et Detach

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.

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

initialisation

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
```

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

Pthread mutexe

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.

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

Pthread mutexe

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

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.

Pthread mutexe

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);
}
```