

POSIX threads (programmation concurrente)

Oum-el-kheir AKTOUF François BROQUEDIS Renaud
LACHAIZE Grégory MOUNIÉ Frédéric PÉTROT Vivien
Quéma

d'après les œuvres de Jacques Mossière et Yves Denneulin

20 octobre 2016

Les fils d'exécution : "threads" ou processus légers

- La création d'un processus est une opération "lourde"
- Les processus sont isolés dans des espaces mémoire différents
- La communication est une opération lourde (fichier, pipe, socket, etc.)
- D'où l'idée de faire coexister plusieurs activités parallèles à l'intérieur d'un même espace mémoire
- Ces activités sont appelées des " threads " (fils d'exécutions, processus légers)

Introduction

Pthread

Moniteur

Exclusion mutuelle

Conditions

Sémaphore

Divers

Les threads

- Ils partagent l'espace mémoire de leur processus
- Ils ont en propre une pile et les registres
- Ils peuvent se communiquer des informations par l'intermédiaire de la mémoire commune

Support dans les OS et les langages

Les systèmes modernes ont un support des threads : Windows, UNIXes : Linux, FreeBSD, Netbsd...

Les langages ont aussi un support des threads :

- Java : Interface ancienne limitée ; remonte l'interface POSIX et bien plus dans les bibliothèques (Java > 1.4) ;
- Ada (select : opérateur de synchronisation entre les Tasks Ada par rendez-vous)
- C-11 : partiellement disponible dans votre gcc (>= 4.7 : -std=gnu11) préféré. Plus simple. Fonctions atomiques.
- C++-11 : idem C-11 mais avec des fonctionnalités supplémentaires aussi avancées que Java,
- mais aussi en Scala, Go, Rust, D, Pascal, Python, C#, etc.

Nous verrons l'API UNIX (Posix thread).

L'interface UNIX : les threads POSIX (Pthreads)

Fonctions

- création/destruction de threads
- synchronisation : moniteur (conditions, mutex) et sémaphore
- ordonnancement, priorités
- signaux

Attributs d'un thread POSIX

- Caractérisation de son état
- Pile
- Données privées

Création d'un thread POSIX

pthread_create

Crée un thread avec les attributs attr, exécute la fonction start_routine avec arg comme argument tid : identificateur du thread créé (équivalent au pid UNIX) join et synchronisation

```
int
pthread_create (pthread_t *tid ,
pthread_attr *attr ,
void* (*start_routine)(void *),
void *arg);
```

Exemple simple (1)

```
#include <pthread.h>
void * ALL_IS_OK = (void *)123456789L;
char *mess[2] = { "boys", "girls" };

void *
writer(void * arg) {
    int i, j;

    for(i=0;i<10;i++) {
        printf("Hi_%s!_(I'm_%lx)\n",
            arg, pthread_self());
        j = 800000; while(j--);
    }
    return ALL_IS_OK;
}
```

Exemple simple (2)

```
int
main(void)
{
    void * status;
    pthread_t writer1_pid, writer2_pid;

    pthread_create(&writer1_pid, NULL, writer,
        (void *)mess[1]);
    pthread_create(&writer2_pid, NULL, writer,
        (void *)mess[0]);
}
```

Exemple simple (3)

```
pthread_join(writer1_pid, &status);
if(status == ALL_IS_OK)
    printf("Thread_%lx_completed_ok.\n",
        writer1_pid);

pthread_join(writer2_pid, &status);
if(status == ALL_IS_OK)
    printf("Thread_%lx_completed_ok.\n",
        writer2_pid);

return 0;
}
```

Exclusion mutuelle

Exemple

```
pthread_mutex_t mon_mutex;

pthread_mutex_init(&mon_mutex, NULL);
...
pthread_mutex_lock(&mon_mutex);
<section critique>
pthread_mutex_unlock(&mon_mutex);
...
//fin du programme
pthread_mutex_destroy(&mon_mutex);
```

Exercice

Exercice

Lancer deux threads (create, join), puis, au choix :

- faire la somme des éléments d'un tableau
- faire l'addition de deux matrices
- faire v++ sur la même variable globale (+ mutex)

Les conditions

pthread_cond_t type condition

pthread_cond_init initialisation d'une condition

pthread_cond_wait mise en attente sur une condition et sortie de mutex, reprise du mutex au réveil

pthread_cond_signal réveil sur une condition

Le signal est différent de celui de Hoare !

Attention le thread signalé ne prend pas immédiatement le contrôle.

Réalisation d'un moniteur

- Un mutex pour assurer l'exclusion mutuelle
- Chaque procédure du moniteur est parenthésée par `pthread_mutex_lock()` et `pthread_mutex_unlock()`
- Chaque variable de condition est une variable `pthread_cond_t`
- **Le thread réveillé n'est pas activé immédiatement** par `pthread_cond_signal()`
- Généralement il faut **réévaluer la condition de blocage** (en pratique, emploi d'un `while` plutôt qu'un `if`)
- Le réveil en cascade ne fonctionne pas toujours! En général, il faut mettre `pthread_cond_signal` juste avant de terminer la procédure (juste avant `unlock`)

Réalisation d'un moniteur

- Un mutex pour assurer l'exclusion mutuelle
- Chaque procédure du moniteur est parenthésée par `pthread_mutex_lock()` et `pthread_mutex_unlock()`
- Chaque variable de condition est une variable `pthread_cond_t`
- **Le thread réveillé n'est pas activé immédiatement** par `pthread_cond_signal()`
- Généralement il faut **réévaluer la condition de blocage** (en pratique, emploi d'un `while` plutôt qu'un `if`)
- Le réveil en cascade ne fonctionne pas toujours! En général, il faut mettre `pthread_cond_signal` juste avant de terminer la procédure (juste avant `unlock`)

Généralités et points particuliers

Ces conseils sont des généralités mais parfois ils ne correspondent pas à la synchronisation voulue!

Un exemple : l'allocateur(1/2)

```

int nlibre = 123;
pthread_cond_t c ; pthread_mutex_t mutex;
pthread_cond_init (&c, NULL) ;

void allouer (int n) {
    pthread_mutex_lock(&mutex);
    while (n > nlibre) {
        pthread_cond_wait (&c, &mutex) ;
    }
    nlibre = nlibre + n ;
    pthread_mutex_unlock (&mutex) ;
}

```

Un exemple : l'allocateur (2/2)

```

void liberer (int m) {
    pthread_mutex_lock (&mutex) ;
    nlibre = nlibre + m ;
    pthread_cond_broadcast (&c) ;
    pthread_mutex_unlock (&mutex) ;
}

```

Attention au réveil en cascade!(1/2)

```

int nlibre = 123;
pthread_cond_t c ; pthread_mutex_t mutex;
pthread_cond_init (&c, NULL) ;

void liberer (int m)
{...
    pthread_cond_signal (&c) ; // !!!!!!!
}

```

Attention au réveil en cascade! (2/2)

```

void allouer (int n) {
    pthread_mutex_lock(&mutex);
    while (n > nlibre) {
        pthread_cond_wait (&c, &mutex) ;
        pthread_cond_signal (&c) ; // mouvement
    } // perpetuel
    nlibre = nlibre + n ;
    pthread_mutex_unlock (&mutex) ;
}

```

Nommés ou anonymes

Les sémaphores POSIX peuvent être nommés ou non nommés.

Sémaphores anonymes

Un sémaphore non nommé n'est accessible que par sa position en mémoire. Il permet de synchroniser des threads, qui partagent par définition le même espace de mémoire ; et des processus ayant mis en place des segments de mémoire partagée.

Un sémaphore nommé est utilisable pour synchroniser des processus connaissant son nom.

- persistant, indépendamment des processus

Sémaphore

```
sem_t mon_sem;

sem_init(&mon_sem, NULL, 3); // anonyme, pour thread
...                          // initialisé a 3
sem_wait(&mon_sem); // P()
sem_post(&mon_sem); // V()
...
sem_destroy(&mon_sem);
```

Autres détails et opérations utiles

`sleep(t)` bloque le thread courant pendant t secondes

`pthread_cancel(threadid)` détruit le thread *threadid*

`pthread_cond_broadcast(&cond)` réveille l'ensemble des threads en attente de la condition

Tests `pthread_mutex_trylock()`, `sem_trywait()`

Timer `pthread_cond_timedwait()`, `sem_timedwait()`

Les man sont vos amis

Par exemple, sur l'initialisation à la création des variables.

Compilation

Entêtes des fonctions dans `#include <pthread.h>`

Le code des fonctions est dans la bibliothèque `libpthread` (à l'édition de lien : `-lpthread`, comme le `-lm` pour la bibliothèque mathématique `libm`).

Gdb et les threads

Il est possible d'explorer l'état d'un processus composé de plusieurs threads

info threads donne la liste des threads et leur numéros,

thread 4 déplace le contexte du débogueur vers le thread numéro 4,

where, up, down, print, ... fonctionne pour le thread courant.

Documentations

- ensiwiki
- Les pages de man

Deux petits tutoriaux

<https://computing.llnl.gov/tutorials/pthreads/>

<http://www.lix.polytechnique.fr/liberti/public/computing/parallel/threads/threads-tutorial/tutorial.html>

Valgrind et les threads

En plus de vérifier vos accès mémoire, valgrind est aussi capable de vérifier vos synchronisations. Il y a même deux détecteurs différents.

-tool=helgrind : détecteur de condition de courses, lock et usage incorrecte de la bibliothèque Pthread

-tool=drd : idem et + (openmp, ...)

NB : il faut que les accès mémoires soient corrects !

Travail demandé

- Implanter le sujet présent sur ensiwiki
- Création et initialisation des variables de synchronisation et des threads
- Faire correctement les synchronisations
- Le programme doit fournir une trace d'exécution montrant le démarrage et la fin de chaque thread, les appels, les blocages et les réveils