Threads et programmation concurrente : Pthreads, C et C++

Louis Boulanger Marie Badaroux Florence Maraninchi Grégory Mounié Alain Tchana Frédéric Wagner
libre adaptation des œuvres originales de Jacques Mossière et Yves Denneulin

20 octobre 2023

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

- Les processus sont isolés dans des espaces mémoire (virtuels) différents
- La création d'un processus est donc une opération "lourde" (création d'un espace de mémoire virtuel)
- La communication entre processus est une opération "lourde" à cause de l'isolation (utilisation de fichier, pipe, socket, etc.)
- Le changement de contexte (context switch) entre des threads de deux processus différent est lourd (les caches doivent être re-remplies avec de nouvelles données)
- 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)

Les threads

- Ils partagent l'espace mémoire (virtuel) de leur processus
- Ils ont chacun sa propre une pile et un ensemble d'état des registres
- Ils peuvent se communiquer des informations par l'intermédiaire de la mémoire (virtuelle) commune (en lisant et modifiant des variables communes)

Support dans les OS et les langages : omniprésent

Les systèmes modernes ont un support avancé des threads :
Windows et les UNIXes : Linux, Freebsd, Netbsd, MacOS X
Tous les langages courants ont un support des threads :

- `Java` : Interface ancienne limitée ; remonte l'interface POSIX et bien plus dans les bibliothèques (Java $>$ 1.4) ;
- `C-11` : disponible dans votre gcc ($>=$ 4.9 : `-std=gnu11`) préféré. Interface un peu simplifié. Fonctions atomiques.
- `C++-20-23` : de nombreuses fonctionnalités supplémentaires autour du multithreading, aussi avancées que Java,
- `OpenMP` extension de C/C++/Fortran pour le parallélisme multithreadé
- mais aussi en `Ada`, `Scala`, `Go`, `Rust`, `D`, `Pascal`, `Python`, `C#`, `Perl`, `Raku`, etc.

L'interface UNIX : les threads POSIX (Pthreads)

De nombreuses fonctions

- création/destruction de threads (joignable ou détaché)
- synchronisation : moniteur (conditions, mutex) et sémaphore
- ordonnancement, priorités, contention
- signaux
- init once, rwlock, barrier, thread local/specific

Autres API
Les PThreads sont la brique de base de l'implantation de la plupart des autres API (C++, Java, Python, etc.). Toutes les API sont donc toutes très similaires.

Attributs d'un thread POSIX

Caractérisation de son état :

- Pile (taille réglable, détection des débordements)
- Données privées (variable globale, avec une copie différente par thread)
- Affinité (cur préféré pour les NUMA)
- Joignable ou détaché

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

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

Exemple simple (1)

```c
#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)

```c
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)

```c
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;
}
```


DANGER : pointer vers des arguments (et le retour) (1/2)

Ce qui est pointé ne doit pas être modifié !

```c
#include <pthread.h>
pthread_t th;

int main() {
  // OK: passage par copie
  for (int i = 0; i < N; i++)
    pthread_create(&th, NULL, FONCTION, (void *)i);

  // FAUX: c'est le même i pour tous les threads, il
  ↪ change
  for (int i = 0; i < N; i++)
    pthread_create(&th, NULL, FONCTION, &i);
}
```

Idem pour la durée de vie du pointeur renvoyé.

DANGER : pointer vers des arguments (et le retour) (2/2)

Ce qui est pointé doit vivre suffisamment longtemps !

```c
#include <pthread.h>
pthread_t th;

int main() {
  // OK car args vie jusqu'après le join
  struct ARG args = {1, 2, 3};
  pthread_create(&th, NULL, FONCTION, &args);
  pthread_join(th, &RES);

  // FAUX dans une boucle
  for (int i = 0; i < N; i++) {
    struct ARG args = {1, 2, 3};
    pthread_create(&th, NULL, FONCTION, &args);
  }
}
```

Idem pour la durée de vie du pointeur renvoyé.

API C

Elle est calquée sur l'API POSIX, mais en se limitant au sous-ensemble portable entre les OS.
Mini-différence avec PThreads : un thread renvoie un `int` au lieu d'un `void*`. Elle a les mêmes dangers sur la durée de vie de l'argument unique.

```c
#include <threads.h>

thrd_t th;

thrd_create(&th, int (*FONCTION)(void *), void
↪  *ARG);
thrd_join(th, int *RES);
```


API C++

Le compilateur permet de passer des arguments complexes. Le type thread n'est pas un simple entier.

```c
#include <thread>
thread th{FONCTION, ARG1, ARG2, ARG3};
th.join();

thread *th2 = new thread{FONCTION, ARG1, ARG2, ARG3};
t2->join();
delete t2;

unique_ptr<thread> th3;
th3.reset(new thread{FONCTION, ARG1, ARG2, ARG3});
t2->join();
```


Exclusion mutuelle en PThreads

```c
pthread_mutex_t m = PTHREAD_MUTEX_INTITIALIZER; //
↪  init statique

void init() {
  pthread_mutex_init(&m, NULL); // si alloc
↪  dynamique
}
void f() {
  pthread_mutex_lock(&m);
  // <section critique>
  pthread_mutex_unlock(&m);
}
void FIN() { pthread_mutex_destroy(&m); }
```

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)

API C : quasi identique à PThreads

```c
#include <threads.h>
mtx_t m; // mais pas d'init statique !

void init() {
  mtx_init(&m, mtx_plain); // obligatoire
}
void f() {
  mtx_lock(&m);
  // <section critique>
  mtx_unlock(&m);
}
void fin() { mtx_destroy(&m); }
```


API C++ : plus simple grâce aux constructeurs

```cpp
#include <mutex>
mutex m; // "plain" par défaut

void f() {
  m.lock();
  // DANGER: si exception, pas de unlock
  m.unlock();
}
void g() {
  unique_lock<mutex> lg{m}; // lock à la création
  // Exception proof
} // unlock à la destruction de lg
```


PThreads : les conditions

PThreads

```c
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INTITIALIZER;

void init() {
  pthread_cond_init(&c, NULL); // si alloc dynamique
}
void f() {
  pthread_mutex_lock(&m);
  while (!done)
    pthread_cond_wait(&c, &m);
  pthread_cond_signal(c); // ou pthread_cond_broadcast
  pthread_mutex_unlock(&m);
}
void FIN() { pthread_cond_destroy(&m); }
```

Réalisation d'un moniteur

Conseils généraux décliné ici avec les PThreads

Le signal est différent de celui prévu par Hoare !

Attention le thread signalé ne prend pas immédiatement le contrôle.

- **Un seul mutex** pour assurer l'exclusion mutuelle
- Chaque procédure du moniteur commence par `pthread_mutex_lock()` et termine par `pthread_mutex_unlock()`
- **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`
- Avec un réveil en cascade, le `pthread_cond_signal` se place juste avant de terminer la procédure (juste avant unlock)

Un exemple Pthread : l'allocateur(1/2)

```
1  int nlibre = 123;
2  pthread_cond_t c = PTHREAD_COND_INITIALIZER;
3  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
4
5  void allouer(int n) {
6    pthread_mutex_lock(&mutex);
7    while (n > nlibre) {
8      pthread_cond_wait(&c, &mutex);
9    }
10   nlibre = nlibre - n;
11   pthread_mutex_unlock(&mutex);
12 }
```


Un exemple PThreads : l'allocateur (2/2)

```
1  void liberer(int m) {
2    pthread_mutex_lock(&mutex);
3    nlibre = nlibre + m;
4    pthread_cond_broadcast(&c);
5    pthread_mutex_unlock(&mutex);
6  }
```


Attention au réveil en cascade !(1/2)

```
1  void liberer(int m) {
2    //...
3    pthread_cond_signal(&c); // BUG !!!!!!
4    //...
5  }
```

Attention au réveil en cascade ! (2/2)

```
1  void allouer(int n) {
2    pthread_mutex_lock(&mutex);
3    while (n > nlibre) {
4      pthread_cond_wait(&c, &mutex);
5      pthread_cond_signal(&c); // mouvement
6    }                          // perpetuel
7    nlibre = nlibre - n;
8    pthread_mutex_unlock(&mutex);
9  }
```


Un exemple C++ : l'allocateur(1/2)

```
1  int nlibre = 123;
2  condition_variable c;
3  mutex m;
4
5  void allouer(int n) {
6    unique_lock<mutex> lg{m};
7    c.wait(lg, []() { return n <= nlibre; }); // !
   ↪  test du while en Pthreads
8    nlibre = nlibre - n;
9  }
```


Un exemple C++ : l'allocateur (2/2)

```
1  void liberer(int m) {
2    unique_lock<mutex> lg{m};
3    nlibre = nlibre + m;
4    c.notify_all();
5  }
```


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 PThreads

Pas d'API C !

```c
#include <semaphore.h>
sem_t s; // pas d'init statique

void init() {
  sem_init(&s, NULL, 3); // anonyme, pour threads,
                         // initialisé a 3
}

void f() {
  sem_wait(&s); // P()
  sem_post(&s); // V()
}
void FIN { sem_destroy(&s); }
```


API C++ : les sémaphores (ajouté dans C++-20)

```c
#include <semaphore>
binary_semaphore s{0}; // ou s{1}
counting_semaphore s2{3};

void f() {
  s.acquire(); // P()
  s.release(); // V())
}
```


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 de Pthreads : `#include <pthread.h>` , `#include <semaphore.h>` , de C : `#include <threads.h>` , et de C++ :

```c
    #include <thread>
    #include <mutex>
    #include <condition_variable>
    #include <semaphore>
```

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

Gdb supporte 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.

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 !

Documentations

- Les pages de man
- Les pages info

Quelques tutoriaux sur les posix threads

`https://hpc-tutorials.llnl.gov/posix/` `http://www.lix.polytechnique.fr/~liberti/public/computing/parallel/threads/threads-tutorial/tutorial.html`

Travail demandé

- Implanter le sujet présent sur le site web
- Création et initialisation des variables de synchronisation et des threads
- Faire correctement les synchronisations