Synchronisation : moniteurs et sémaphores en Pthread, C et C++

Grégory Mounié

2023

1 Les différences entre les trois API

L'interface du langage C a été calquée sur les Pthreads/C (Bibliothèque POSIX Threads utilisée en C), donc les différences sont minimes. L'interface de C est moins verbeuse que Pthreads/C mais n'offre pas de fonctionnalités aussi avancées.

Les différences entre C++ et C sont plus importantes. C++ permet d'initialiser les types plus facilement. C++ empêche leurs copies (ce n'est pas visible dans les exemples ici). C++ propose de lier le lock d'un mutex à une variable locale, qui sera forcément toujours détruite. Le wait des variables de condition utilise cette garde. Ces différences sont illustrées dans ce document.

La documentation sur les POSIX Threads est disponible dans les pages *man* de votre Linux préféré (`man pthreads` , `man pthread_create` , etc.). La documentation sur les threads C est disponible dans les pages info de la librairie C GNU (`info libc threads "iso c threads"`). Une documentation de référence sur C++, avec les évolutions du standard est disponible en ligne sur `https://en.cppreference.com/w/`.

Table des matières

1 Les différences entre les trois API **1**

2 Moniteurs **2**
- 2.1 Les types . . . 2
- 2.2 Initialisation en C et Pthreads/C . . . 2
- 2.3 Utilisation des mutex en C++ . . . 3
- 2.4 Utilisation des conditions en C++ et C . . . 4

3 Un petit exemple complet de moniteur en C et C++ **5**

4 Sémaphores **7**

5 Les autres points qui ne sont pas montrés dans ce document 7

2 Moniteurs

Les trois API proposent les mêmes types (mutex, variables de condition) mais avec des variations dans le nom exact des types.

2.1 Les types

```
// en pthread
#include <pthread.h>
pthread_mutex_t m;
phtread_cond_t c;

// en C
#include <threads.h>
mtx_t m;
cnd_t c;
```

```
// en C++
#include <mutex>
mutex m;

#include <condition_variable>
condition_variable c;
```

2.2 Initialisation en C et Pthreads/C

En C++ les constructeurs font le travail d'initialisation. Il suffit de définir les variables avec les bons types. En C, il faut faire ce travail à la main, à l'initialisation de la création (en Pthread) ou dans une fonction d'initialisation (en Pthread et en C).

Sous Linux, l'implantation des opérateurs de synchronisation est à base d'entiers manipulés avec des opérations atomiques. Comme Linux initialise à 0 la mémoire contenant les variables globales, et les grosses allocations dynamiques de mémoire (malloc/free), l'absence d'initialisation peut passer complètement inaperçu. La compilation du code sur d'autres systèmes apporte alors son lot de bugs imprévus.

```cpp
// en C++
mutex m;
condition_variable c;
// Il n'y a rien à faire de plus
// dans le cas standard
```

```c
// en C
mtx_t m;
cnd_t c;

// seul l'initalisation dynamique
↪  est standardisée
void init() {
 mtx_init(&m, mtx_plain);
 cnd_init(&c);
}
```

```c
// en Pthread/C
// initalisation à la création,
//  existe en pthread, pas en C
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

// initialisation dynamique
void init() {
 // nullptr depuis C-23, ou NULL
 pthread_mutex_init(&m, nullptr);
 pthread_cond_init(&c, nullptr);
}
```

2.3 Utilisation des mutex en C++

En C++, à cause des exceptions, l'utilisation effective des mutex ne se fait pas avec un lock et un lock "à la main", mais avec une variable locale de type `unique_lock`, qui sera toujours détruite en sortie de la fonction, que ce soit par le chemin normal ou par une exception. Par défaut, le constructeur de la variable appelle `lock` sur le mutex et le destructeur appelle `unlock`.

En java, cela est traité avec la construction `m.lock(); try { section critique } finally { m.unlock()}`.

```cpp
// en C++

mutex m;

void A() {
 unique_lock<mutex> lg{m};

 // section critique

}
```

```cpp
// en C++, à la main cela compile
↪  mais...
// DANGER: SENSIBLE AUX EXCEPTIONS
↪  qui ne feront pas unlock
// PROBLÈME: les condition_variable
↪  demande un paramètre de type
↪  unique_lock

mutex m;

void A() {
 m.lock();

 // section critique

 m.unlock();
}
```

2.4 Utilisation des conditions en C++ et C

En C++, l'appel à `wait` est bloquant et prend deux arguments : un `unique_lock` et une fonction booléenne de passage qui débloque le thread lorsqu'elle vaut `true`.

Attention, entre C et C++, les deux expressions booléenes sont inversées : en C, c'est un ńăblocage tant queăż alors qu'en C++ c'est ńăpassage si est seulement siăż.

```cpp
// en C++
mutex m;
condition_variable c;
bool fini;

void A() {
 unique_lock<mutex> lg{m};

 c.wait(lg, [] { return fini; });
 c.notify_one();
 c.notify_all();
}
```

```cpp
// en C
mtx_t m;
cnd_t c;
bool fini;

void A() {
 mtx_lock(&m);
 while (!fini)
  cnd_wait(&c, &m);
 cnd_signal(&c);
 cnd_broadcast(&c);
 mtx_unlock(&m);
}
```

3 Un petit exemple complet de moniteur en C et C++

Un exemple artificiel, mais petit et complet avec deux threads exécutant deux fonctions du moniteur `A()` et `B()`. Le thread exécutant `A()`, modifie une variable `a`. Le thread exécutant `B()` attend la fin de la modification pour afficher la valeur de la variable `a`.

```c
// En C
// TODO à la sortie de gcc-13 avec
↪  support C23
// ajouter: nullptr
// enlever: stdbool
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <threads.h>

mtx_t m;
cnd_t c;
int a = 0;
bool done;

void init() {
 mtx_init(&m, mtx_plain);
 cnd_init(&c);
}

int A(void *) {
 mtx_lock(&m);
 a = 42;
 done = true;
 cnd_signal(&c);
 mtx_unlock(&m);
 return 0;
}

int B(void *) {
 mtx_lock(&m);
 while (!done)
  cnd_wait(&c, &m);
 printf("%d\n", a);
 mtx_unlock(&m);
 return 0;
}

int main(int, char **) {
 thrd_t ta, tb;
 init();

 thrd_create(&ta, A, NULL);
 thrd_create(&tb, B, NULL);

 int resa, resb;
 thrd_join(ta, &resa);
 thrd_join(tb, &resb);
}
```

```cpp
// En C++
#include <cassert>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std;
mutex m;
condition_variable c;
int a = 0;
bool done;

void A() {
 unique_lock<mutex> lg{m}; // mutex
↪  lock à la création du lg
 a = 42;
 done = true;
 c.notify_one();
} // mutex unlock à la destruction du
↪  lg

void B() {
 unique_lock<mutex> lg{m};
 // while sur "done" avec une fonction
↪  anonyme
 c.wait(lg, [] { return done; });
 cout << a << endl;
}

int main(int, char **) {
 thread ta{A};
 thread tb{B};

 ta.join();
 tb.join();
}
```

4 Sémaphores

Les sémaphores ne sont pas standardisés dans le langage C, seulement dans les PThreads/C. Pour être précis, les sémaphores existaient dans la norme POSIX avant les Pthreads, car ils sont aussi utilisables pour synchroniser des processus. Ils ont été ajoutés dans C++ 20. C++ propose de différencier les sémaphores binaires, qui ne proposent qu'un seul passage sans bloquer, des autres. L'initialisation des sémaphores est incluse dans les constructeurs de C++.

```c
// POSIX/C
#include <semaphore.h>

sem_t s;

// seul l'initalisation dynamique
↪  est standardisée
// 0: synchro entre threads
// !0: synchro entre processus, le
↪  sémaphore doit être en mémoire
↪  partagée
void init() {
 sem_init(&s, 0, 42); // 42 jetons
}

void A() {
 sem_wait(&s); // P()
 sem_post(&s); // V()
}
```

```c
// C++
#include <semaphore>

counting_semaphore s{42}; // entier
↪  positif
binary_semaphore s2{1}; //
↪  initialisation à 0 ou 1

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

5 Les autres points qui ne sont pas montrés dans ce document

Ce document ne discute que des opérateurs fondamentaux. Pthreads/C offre par exemple une interface pour régler (si possible) l'ordonnancement des threads. Toutes les API, pour toutes les fonctions bloquantes (wait sur les moniteurs et les sémaphores) peuvent être associées avec un délai de garde (timeout). Certaines API proposent des opérateurs supplémentaires. En particulier, des barrières de synchronisation de threads, à usage unique (Latch C++) ou à usage multiple en Pthreads/C et C++ (Barrier en C++ et `pthread_barrier` en Pthreads/C). C++ proposent également de nombreuses variations autour des threads (Async), des tâches parallèles (Coroutine) et des communications asynchrones (Future/Promise). L'extension OpenMP de C et C++ (`https://www.openmp.org`), disponible dans GCC et Clang offre un support de plus haut niveau dans la gestion des exécutions parallèles. Les interactions entre les exé-

cutions multi-threadées et les signaux (comme SIGINT de votre Control-C, pas celui des variables de condition) sont complexes et si besoin, leurs détails doivent être regardés dans la documentation de votre système. En particulier, certains appels bloquants peuvent être débloqués à la réception d'un signal par le thread.