eBook Gratuit

APPRENEZ pthreads

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#pthreads

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: pthreads

It is an unofficial and free pthreads ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official pthreads.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec pthreads

Remarques

Cette section fournit une vue d'ensemble de ce que pthreads est et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans pthreads, et établir un lien avec les sujets connexes. La documentation de pthreads étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

Installation ou configuration

Instructions détaillées sur la configuration ou l'installation de pthreads.

Minimal "Hello World" avec pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>

/* function to be run as a thread always must have the same signature:
   it has one void* parameter and returns void */
void *threadfunction(void *arg)
{
  printf("Hello, World!\n"); /*printf() is specified as thread-safe as of C11*/
  return 0;
}

int main(void)
{
  pthread_t thread;
  int createerror = pthread_create(&thread, NULL, threadfunction, NULL);
  /*creates a new thread with default attributes and NULL passed as the argument to the start
routine*/
  if (!createerror) /*check whether the thread creation was successful*/
    {
      pthread_join(thread, NULL); /*wait until the created thread terminates*/
      return 0;
    }
  fprintf("%s\n", strerror(createerror), stderr);
  return 1;
}
```

Passer des arguments aux threads

```
#include <stdio.h>
#include <pthread.h>
```

```
void *thread_func(void *arg)
{
    printf("I am thread #%d\n", *(int *)arg);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int i = 1;
    int j = 2;

    /* Create 2 threads t1 and t2 with default attributes which will execute
    function "thread_func()" in their own contexts with specified arguments. */
    pthread_create(&t1, NULL, &thread_func, &i);
    pthread_create(&t2, NULL, &thread_func, &j);

    /* This makes the main thread wait on the death of t1 and t2. */
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("In main thread\n");
    return 0;
}
```

Comment compiler:

```
$ gcc -pthread -o hello hello.c
```

Cela imprime:

```
I am thread #1
I am thread #2
In main thread
```

Résultat de retour du fil

Un pointeur sur un type de données concret, converti en `void *` , peut être utilisé pour transmettre des valeurs et renvoyer des résultats à partir de la fonction thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct thread_args
{
    int a;
    double b;
};


struct thread_result
{
    long x;
    double y;
```

```
};

void *thread_func(void *args_void)
{
    struct thread_args *args = args_void;
    /* The thread cannot return a pointer to a local variable */
    struct thread_result *res = malloc(sizeof *res);

    res->x  = 10 + args->a;
    res->y = args->a * args->b;
    return res;
}

int main()
{
    pthread_t threadL;
    struct thread_args in = { .a = 10, .b = 3.141592653 };
    void *out_void;
    struct thread_result *out;

    pthread_create(&threadL, NULL, thread_func, &in);
    pthread_join(threadL, &out_void);
    out = out_void;
    printf("out -> x = %ld\tout -> b = %f\n", out->x, out->y);
    free(out);

    return 0;
}
```

Dans de nombreux cas, il est inutile de transmettre une valeur de retour - par exemple, l'espace dans l'argument struct peut également être utilisé pour renvoyer des résultats ou un pointeur sur une structure de données partagée peut être transmis au thread et les résultats stockés .

Lire Démarrer avec pthreads en ligne: https://riptutorial.com/fr/pthreads/topic/5669/demarrer-avec-pthreads

Chapitre 2: Condition de course dans les pthreads

Introduction

Lors de l'écriture d'applications multithread, l'un des problèmes les plus courants concerne les conditions de course. Donc, nous documentons comment vous les détectez? et comment les manipulez-vous?

Examples

Exemple: Consider aura deux threads T1 et T2.

Comment les détectez-vous?

Si la même *variable / ressource / emplacement mémoire* est accessible par plusieurs threads et qu'au moins le thread modifie la valeur de la *variable / ressource / emplacement mémoire* , alors une situation de **concurrence** peut survenir. Parce que si un thread modifie la valeur de la *variable / ressource / emplacement mémoire* et qu'un autre thread essaie de lire la même chose, il n'obtiendra pas la valeur mise à jour.

Remarque : Si tous les threads ne font que lire la *variable / ressource / emplacement mémoire,* alors la **condition de** concurrence ne se produira pas.

Exemple: le programme souffre d'une condition de course

```
#include <stdio.h>
#include <pthread.h>

int x= 0;

void* fun(void* in)
{
    int i;
    for ( i = 0; i < 10000000; i++ )
    {
        x++;
    }
}

int main()
{
    pthread_t t1, t2;
    printf("Point 1 >> X is: %d\n", x);

    pthread_create(&t1, NULL, fun, NULL);
    pthread_create(&t2, NULL, fun, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
```

```
    printf("Point 2 >> X is: %d\n", x);
    return 0;
}
```

La sortie sur mon écran est la suivante:

```
Point 1 >> X is: 0
Point 2 >> X is: 9925047
```

Votre sortie variera. Mais à coup sûr, ce ne sera pas 20 000 000. Comme les deux threads exécutent la même boucle et ont la variable globale `int x;`

```
for ( i = 0; i < 10000000; i++ )
{
   x++;
}
```

Donc, la valeur finale de `x` dans la ligne `Point 2 >> X is: 9925047` devrait être `Point 2 >> X is: 9925047` . Mais ce n'est pas le cas.

L'état de x peut être modifié par un autre thread pendant le temps écoulé entre la lecture de x et la réécriture.

Disons qu'un thread récupère la valeur de x, mais ne l'a pas encore stockée. Un autre thread peut également récupérer la même valeur de x (car aucun thread n'a encore changé) et ensuite ils stockeront tous les deux la même valeur (x + 1) dans x!

Exemple:

Thread 1: lit x, la valeur est 7

Fil 1: ajouter 1 à x, la valeur est maintenant 8

Thread 2: lit x, la valeur est 7

Fil 1: stocke 8 en x

Thread 2: ajoute 1 à x, la valeur est maintenant 8

Fil 2: stocke 8 en x

Comment les manipulez-vous?

Les conditions de course peuvent être évitées en utilisant une sorte de mécanisme de verrouillage avant le code qui accède à la ressource partagée ou une exclusion mutuelle.

Le programme suivant est modifié:

Exemple: problème de condition de course résolu

```
#include <stdio.h>
#include <pthread.h>

int x= 0;
//Create mutex
pthread_mutex_t test_mutex;

void* fun(void* in)
{
    int i;
    for ( i = 0; i < 10000000; i++ )
    {
        //Lock mutex before going to change variable
        pthread_mutex_lock(&test_mutex);
        x++;
        //Unlock mutex after changing the variable
        pthread_mutex_unlock(&test_mutex);
    }
}

int main()
{
    pthread_t t1, t2;
    printf("Point 1 >> X is: %d\n", x);

    //Initlize mutex
    pthread_mutex_init(&test_mutex, NULL);

    pthread_create(&t1, NULL, fun, NULL);
    pthread_create(&t2, NULL, fun, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    //Destroy mutex after use
    pthread_mutex_destroy(&test_mutex);
    printf("Point 2 >> X is: %d\n", x);
    return 0;
}
```

Voici le résultat:

```
Point 1 >> X is: 0
Point 2 >> X is: 20000000
```

Ici, la réponse est 20 000 000 à chaque fois.

Remarque : le programme modifié, qui ne contient aucune erreur de condition de course, prendra beaucoup de temps à exécuter. Parce que le `mutex` verrouille et déverrouille.

Lire Condition de course dans les pthreads en ligne:
https://riptutorial.com/fr/pthreads/topic/8243/condition-de-course-dans-les-pthreads

Chapitre 3: Variables conditionnelles

Introduction

Les variables conditionnelles sont utiles dans les cas où vous souhaitez qu'un thread attend quelque chose qui se produit dans un autre thread. Par exemple, dans un scénario producteur / consommateur avec un ou plusieurs threads producteurs et un seul thread consommant, les variables conditionnelles peuvent être utilisées pour signaler le thread consommant que de nouvelles données sont disponibles.

Remarques

Processus général

Une attente sur une variable conditionnelle (queueCond dans l'exemple producteur / consommateur) est toujours couplée à un mutex (le queueMutex dans l'exemple producteur / consommateur) et doit toujours être couplée à une variable d'état "normale" (queue.empty ( ) dans l'exemple producteur / consommateur). Utilisé correctement, cela garantit qu'aucune nouvelle donnée n'est manquée chez le consommateur.

En général, le processus devrait être:

- Pour le fil de signalisation:
    1. Verrouille le mutex
    2. Mettre à jour les données et les variables d'état
    3. Signalez la variable de condition
    4. Débloquer le mutex
- Pour le thread en attente:
    1. Verrouille le mutex
    2. Faites une `while` boucle sur la variable d'état, en boucle tant que les données ne sont pas prêts
    3. Dans le `while` en boucle, faire une attente sur la variable d'état avec `pthread_cond_wait()`
    4. Lorsque le `while` en sortie de la boucle, nous sommes maintenant sûr que les nouvelles données sont prêtes, et que le mutex est verrouillé
    5. Faire quelque chose avec les données
    6. Débloquer le mutex et répéter

Avec ce schéma, peu importe quand les threads de signalisation et d'attente sont programmés, le thread en attente ne manquera jamais de données (comme dans, il ne sera jamais bloqué en attente pour toujours avec des données valides prêtes). Cela peut être réalisé en essayant manuellement de parcourir les étapes pour le thread de signalisation, en enregistrant les états des variables mutex, condition et state, pour chacune des étapes du thread en attente.

`pthread_cond_wait` et le mutex

Pour faciliter le processus ci-dessus, il est nécessaire d'appeler `pthread_cond_wait()` avec le mutex verrouillé. Lorsqu'il est appelé, `pthread_cond_wait()` déverrouillera alors le mutex avant de mettre le thread en veille, et, juste avant de revenir pour quelque raison que ce soit, le mutex sera libéré. Cela signifie également que si un autre thread a actuellement le mutex verrouillé, `pthread_cond_wait()` attendra que le mutex soit déverrouillé, et jusqu'à ce que le thread en attente puisse réellement acquérir le mutex - il se heurtera à tout autre thread essayant de se verrouiller le mutex en même temps.

Réveils faux

, Il peut sembler aussi comme si `while` boucle d' attente sur la variable d'état pourrait être remplacé par un simple , `if` la déclaration. Cependant, la `while` en boucle est nécessaire, car la norme Posix permet `pthread_cond_wait()` de le faire soi-disant « parasites » redémarrages pendant l'attente, sans être réellement signalé. Ainsi, le code doit revérifier la variable d'état pour voir si `pthread_cond_wait()` est renvoyé en raison de la signalisation effective ou de l'un de ces réveils intempestifs.

Examples

Exemple producteur / consommateur

```
pthread_mutex_t queueMutex;
pthread_cond_t queueCond;
Queue queue;

void Initialize() {
    //Initialize the mutex and the condition variable
    pthread_mutex_init(&queueMutex, NULL);
    pthread_cond_init(&queueCond, NULL);
}

void Producer() {
    //First we get some new data
    Data *newData = MakeNewData();

    //Lock the queue mutex to make sure that adding data to the queue happens correctly
    pthread_mutex_lock(&queueMutex);

    //Push new data to the queue
    queue.push(newData);

    //Signal the condition variable that new data is available in the queue
    pthread_cond_signal(&queueCond);

    //Done, unlock the mutex
    pthread_mutex_unlock(&queueMutex);
}


void Consumer() {

    //Run the consumer loop
    while(1) {
```

```
        //Start by locking the queue mutex
        pthread_mutex_lock(&queueMutex);

        //As long as the queue is empty,
        while(queue.empty()) {
            // - wait for the condition variable to be signalled
            //Note: This call unlocks the mutex when called and
            //relocks it before returning!
            pthread_cond_wait(&queueCond, &queueMutex);
        }

        //As we returned from the call, there must be new data in the queue - get it,
        Data *newData = queue.front();
        // - and remove it from the queue
        queue.pop();

        //Now unlock the mutex
        pthread_mutex_unlock(&queueMutex);

        // - and process the new data
        ProcessData(newData);
    }
}
```

Lire Variables conditionnelles en ligne: https://riptutorial.com/fr/pthreads/topic/8614/variables-conditionnelles

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec pthreads	Armali, ArturFH, caf, Community, cse, EOF, nachiketkulk
2	Condition de course dans les pthreads	cse, Mohan
3	Variables conditionnelles	sonicwave