FREE eBook

LEARNING pthreads

Free unaffiliated eBook created from
Stack Overflow contributors.

#pthreads

Table of Contents

About

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

Chapter 1: Getting started with pthreads

Remarks

This section provides an overview of what pthreads is, and why a developer might want to use it.

It should also mention any large subjects within pthreads, and link out to the related topics. Since the Documentation for pthreads is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

Detailed instructions on getting pthreads set up or installed.

Minimal "Hello World" with 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;
}
```

Passing arguments to 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;
}
```

How to compile:

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

This prints:

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

Returning result from thread

A pointer to a concrete data type, converted to `void *`, can be used to pass values to and return results from the thread function.

```
#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;
}
```

In many cases it is unnecessary to pass a return value in this way - for example, space in the argument struct can also be used to return results, or a pointer to a shared data structure can be passed to the thread and the results stored there.

Read Getting started with pthreads online: https://riptutorial.com/pthreads/topic/5669/getting-started-with-pthreads

Chapter 2: Conditional Variables

Introduction

Conditional variables are useful in cases where you want a thread to wait for something that happens in another thread. For instance, in a producer/consumer scenario with one or or more producing threads and one consuming thread, conditional variables can be used to signal the consuming thread that new data is available.

Remarks

General process

A wait on a conditional variable (queueCond in the producer/consumer example) is always coupled to a mutex (the queueMutex in the producer/consumer example), and should always be coupled to a "normal" state variable also (queue.empty() in the producer/consumer example). When used correctly, this ensures that no new data is missed in the consumer.

In general, the process should be:

- For the signalling thread:
    1. Lock the mutex
    2. Update any data and state variables
    3. Signal the condition variable
    4. Unlock the mutex
- For the waiting thread:
    1. Lock the mutex
    2. Do a `while` loop on the state variable, looping as long as the data is not ready
    3. In the `while` loop, do a wait on the condition variable with `pthread_cond_wait()`
    4. When the `while` loop exits, we are now sure that new data is ready, and that the mutex is locked
    5. Do something with the data
    6. Unlock the mutex and repeat

With this scheme, no matter when the signalling and waiting threads are scheduled, the waiting thread will never miss data (as in, it will never be stuck waiting forever with valid data ready). This can be realized by manually trying to run through the steps for the signalling thread, recording the states of the mutex, condition and state variables, for each of the steps in the waiting thread.

`pthread_cond_wait` and the mutex

To facilitate the above process, it is required to call `pthread_cond_wait()` with the mutex locked. When called, `pthread_cond_wait()` will then unlock the mutex before putting the thread to sleep, and, just before returning for whatever reason, the mutex will be relocked. This also means that if some other thread currently has the mutex locked, `pthread_cond_wait()` will wait for the mutex to be unlocked, and until the waiting thread can actually acquire the mutex - it will contend on it together

with any other threads trying to lock the mutex at the same time.

Spurious wakeups

Also, it may seem as if the `while` loop waiting on the state variable could be substituted for a simple `if` statement. However, the `while` loop is needed, as the Posix standard allows `pthread_cond_wait()` to do so-called "spurious" wakeups during the wait, without actually being signalled. Thus, the code needs to recheck the state variable to see if `pthread_cond_wait()` returned due to actually being signalled, or due to one of these spurious wakeups.

Examples

Producer / consumer example

```
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);
    }
}
```

Read Conditional Variables online: https://riptutorial.com/pthreads/topic/8614/conditional-variables

Chapter 3: Race condition in pthreads

Introduction

When writing multi-threaded applications, one of the most common problems experienced are race conditions. So we document the How do you detect them? and How do you handle them?

Examples

Example: Consider will have two threads T1 and T2.

How do you detect them?

If the same *variable/resource/memory location* is accessible by multiple threads and at least of the thread is changing the value of *variable/resource/memory location*, then **Race Condition** can occurred. Because if a thread is changing the value of *variable/resource/memory location* and another thread tries to read the same then it will not get the updated value.

Note: If all threads are just reading the *variable/resource/memory location* then **Race Condition** will not occur.

Example: Program suffers from Race Condition

```
#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;
}
```

The output on my screen is:

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

Your output will vary. But for sure it will not be 20,000,000. Since both Thread executing the same loop and having global variable `int x;`

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

So final value of `x` in line `Point 2 >> X is: 9925047` should be 20,000,000. But it is not so.

The state of x can be changed by another thread during the time between x is being read and when it is written back.

Let's say a thread retrieves the value of x, but hasn't stored it yet. Another thread can also retrieve the same value of x (because no thread has changed it yet) and then they would both be storing the same value (x+1) back in x!

Example:

Thread 1: reads x, value is 7

Thread 1: add 1 to x, value is now 8

Thread 2: reads x, value is 7

Thread 1: stores 8 in x

Thread 2: adds 1 to x, value is now 8

Thread 2: stores 8 in x

How do you handle them?

Race conditions can be avoided by employing some sort of locking mechanism before the code that accesses the shared resource or mutual exclusion.

Following is modified program:

Example: Race Condition problem resolved

```
#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;
}
```

Following is the output:

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

Here, the answer comes out as 20,000,000 every time.

Note: Modified program, which is free from race condition error, will take much longe to execute. Because there is overburden on `mutex` lock and unlock.

Read Race condition in pthreads online: https://riptutorial.com/pthreads/topic/8243/race-condition-in-pthreads

Credits

S. No	Chapters	Contributors
1	Getting started with pthreads	Armali, ArturFH, caf, Community, cse, EOF, nachiketkulk
2	Conditional Variables	sonicwave
3	Race condition in pthreads	cse, Mohan