Code(quoi);

🇺🇸 English

Threads, Mutexes and Concurrent Programming in C

>_ Mia Combeau >_ C >_ November 2, 2022

▼ Table of Contents

For efficiency or by necessity, a program can be concurrent rather than sequential. Thanks to its concurrent programming and with its child processes or threads and mutexes, it will be able to perform multiple tasks simultaneously.

In a previous article, we came to understand how to create child processes, which are one way to implement concurrent programming. Here, we will concentrate on threads and how to deal with the dangers that come with their shared memory with mutexes.

Concurrent Programming

As opposed to sequential programming, **concurrent programming** allows a program to perform several tasks simultaneously instead of having to wait for the result of one operation to move onto the next. The operating system itself uses this concept to meet its users expectations. If we had to wait for a song to finish to be able to open our browser, or if we had to restart the computer to kill a program caught in an infinite loop, we'd die of frustration!

There are three ways to implement concurrency in our programs: processes, threads, and multiplexing. Let's concentrate on threads.

What is a Thread?

An execution **thread** is a logical sequence of instructions inside a process that is automatically managed by the operating system's kernel. A regular sequential program has a single thread, but modern operating systems allow us to create several threads in our programs, all of which run in parallel.

Each one of a process's threads has its own context: its own ID, its own stack, its own instruction pointer, it's own processor register. But since all of the threads are part of the same process, they share the same virtual memory address space: the same code, the same heap, the same shared libraries and the same open file descriptors.

A thread's context has a smaller footprint in terms of resources than the context of a process. Which means that it is much faster for the system to create a thread than it is to create a process. Switching from one thread to the other, compared to switching from one process to another is also quicker.

Threads don't have the strict parent-child hierarchy that processes do. Rather, they form a group of peers regardless of which thread created which other thread. The only distinction the "main" thread has is being the first one to exist at the beginning of the process. This means that within the same process, any thread can wait for any other thread to complete, or kill any other thread.

Additionally, any thread can read and write to the same virtual memory, which makes communication between threads much easier than communication between processes. We will later examine the problems that can arise from this shared memory.

Using POSIX Threads

The standard interface in C to manipulate threads is POSIX with its **<pthread.h>** library. It contains around sixty functions to create and join threads, as well as to manage their shared memory. We will only study a fraction of these in this article. In order to compile a program using this library, we can't forget to link it with `-pthread`:

```
gcc -pthread main.c
```

Creating a Thread

We can create a new thread from any other thread of the program with the `pthread_create` function. Its prototype is:

```
int pthread_create(pthread_t *restrict thread,
                          const pthread_attr_t *restrict attr,
                          void *(*start_routine)(void *),
                          void *restrict arg);
```

Let’s examine each argument we must supply:

- **thread**: a pointer towards a `pthread_t` type variable, to store the ID of the thread we will be creating.
- **attr**: an argument that allows us to change the default attributes of the new thread. This is beyond the scope of this article, and in general, passing `NULL` here suffices.
- **start_routine**: the function where the thread will start its execution. This function will have as its prototype: `void *function_name(void *arg);` . When the thread reaches the end of this function, it will be done with its tasks.
- **arg**: a pointer towards an argument to pass to the thread’s `start_routine` function. If we’d like to pass several parameters to this function, we will need to give it a pointer to a data structure.

When the `pthread_create` function ends, the thread variable we gave it should contain the newly created thread’s ID. The function itself returns 0 if the creation was successful, or en error code if not.

Joining or Detaching Threads

In order to block the execution of a thread until another thread finishes, we can use the `pthread_join` function:

```
int pthread_join(pthread_t thread, void **retval);
```

Its parameters are as follows:

- **thread**: the ID of the thread that this thread should wait for. The specified thread must be joinable (meaning not detached - see below).
- **retval**: a pointer towards a variable that can contain the return value of the thread’s routine function (the `start_routine` function we supplied at its creation). Here, we will not need this value: a simple `NULL` will suffice.

The **`pthread_join`** function returns 0 for success, or an error code for failure.

Let's note that we can only wait for the termination of a specific thread. There is no way to wait for the first terminated thread without specifying an ID, as the **`wait`** function for child processes does.

But in some cases, it is possible and preferable to not wait for the end of certain theads at all. In that case, we can detach the thread to tell the operating system that it can reclaim its resources right away when it finishes its execution. For that, we use the **`pthread_detach`** function (usually right after that thread's creation):

```
int pthread_detach(pthread_t thread);
```

Here, all we have to supply if the thread's ID. We get 0 in return if the operation was a success, or non-zero if there was an error. After detaching the thread, other threads will not be able to kill or wait for this thread with **`pthread_join`**.

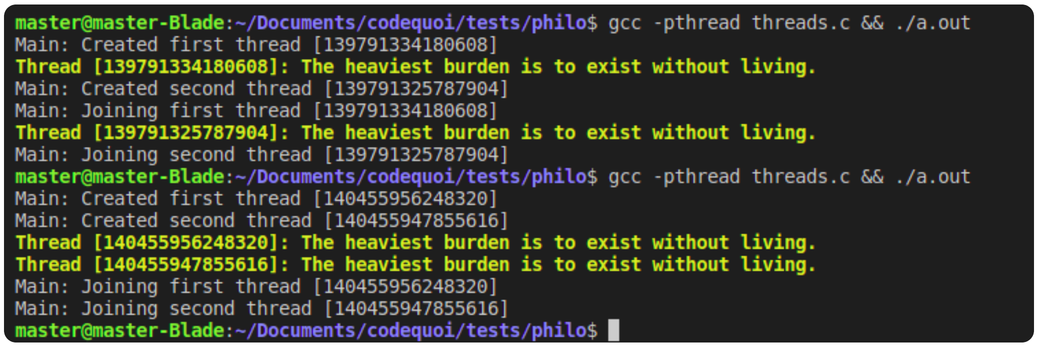
A Practical Example of Threads

Let's write a small, simple program that creates two threads and joins them. The routine of each thread only consists of writing its own ID followed by a roughly-translated philosophic quote from the French novelist Victor Hugo.

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

# define NC "\e[0m"
# define YELLOW "\e[1;33m"

// thread_routine is the function the thread invokes right after its
// creation. The thread ends at the end of this function.
void *thread_routine(void *data)
```

```
{
 pthread_t tid;

 // The pthread_self() function provides
 // this thread's own ID.
 tid = pthread_self();
 printf("%sThread [%ld]: The heaviest burden is to exist without living.%s\
  YELLOW, tid, NC);
 return (NULL); // The thread ends here.
}

int main(void)
{
 pthread_t tid1; // First thread's ID
 pthread_t tid2; // Second thread's ID

 // Creating the first thread that will go
 // execute its thread_routine function.
 pthread_create(&tid1, NULL, thread_routine, NULL);
 printf("Main: Created first thread [%ld]\n", tid1);
 // Creating the second thread that will also execute thread_routine.
 pthread_create(&tid2, NULL, thread_routine, NULL);
 printf("Main: Created second thread [%ld]\n", tid2);
 // The main thread waits for the new threads to end
 // with pthread_join.
 pthread_join(tid1, NULL);
 printf("Main: Joining first thread [%ld]\n", tid1);
 pthread_join(tid2, NULL);
 printf("Main: Joining second thread [%ld]\n", tid2);
 return (0);
}
```

When we compile and run this test, we can see that both threads were created and print their IDs correctly. If we run the program several times in a row, we might notice that the threads are always created in order, but sometimes, the main writes its message before the thread and vice versa. This shows that each thread is indeed executing in parallel to the main thread, and not sequentially.



```
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Created first thread [139791334180608]
Thread [139791334180608]: The heaviest burden is to exist without living.
Main: Created second thread [139791325787904]
Main: Joining first thread [139791334180608]
Thread [139791325787904]: The heaviest burden is to exist without living.
Main: Joining second thread [139791325787904]
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Created first thread [140455956248320]
Main: Created second thread [140455947855616]
Thread [140455956248320]: The heaviest burden is to exist without living.
Thread [140455947855616]: The heaviest burden is to exist without living.
Main: Joining first thread [140455956248320]
Main: Joining second thread [140455947855616]
master@master-Blade:~/Documents/codequoi/tests/philo$
```

Managing Threads' Shared Memory

One of the greatest qualities of threads is that they all share their process's memory. Each thread does have its own stack, but the other threads can very easily gain access to it with a simple pointer. What's more, the heap and any open file descriptors are totally shared between threads.

This shared memory and the ease with which a thread can access another thread's memory clearly also has its share of danger: it can cause nasty synchronization errors.

Synchronization Errors

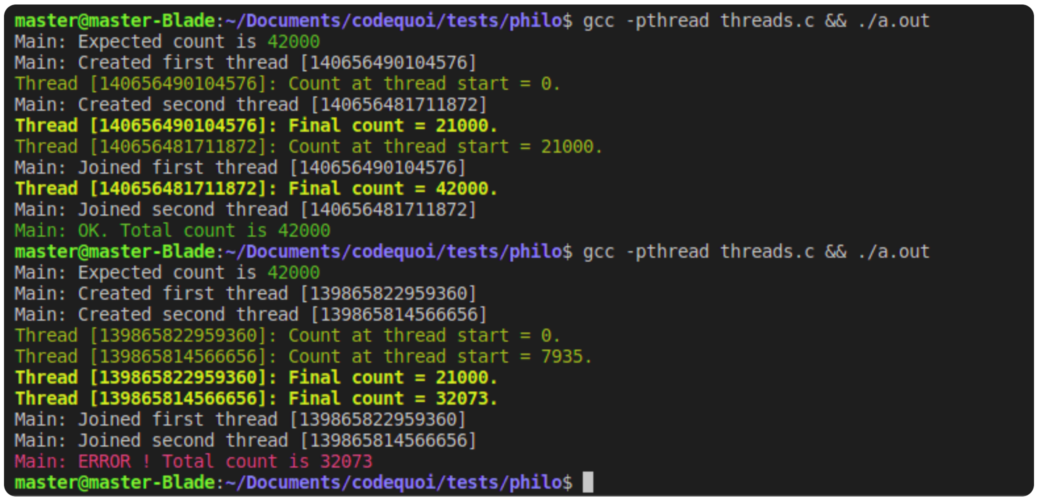
Let's go back to our previous example and modify it to see how the shared virtual memory of threads can cause issues. We will create two threads and give each of them a pointer towards a variable in the main containing an unsigned integer, `count`. Each thread will iterate a certain number of times (defined in the `TIMES_TO_COUNT` macro) and increment the count at each iteration. Since there are two threads, we will of course expect the final count to be exactly twice `TIMES_TO_COUNT`.

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

// Each thread will count TIMES_TO_COUNT times
```

```
#define TIMES_TO_COUNT 21000

#define NC "\e[0m"
#define YELLOW "\e[33m"
#define BYELLOW "\e[1;33m"
#define RED "\e[31m"
#define GREEN "\e[32m"

void *thread_routine(void *data)
{
 // Each thread starts here
 pthread_t tid;
 unsigned int *count; // pointer to the variable created in main
 unsigned int i;

 tid = pthread_self();
 count = (unsigned int *)data;
 // Print the count before this thread starts iterating:
 printf("%sThread [%ld]: Count at thread start = %u.%s\n",
  YELLOW, tid, *count, NC);
 i = 0;
 while (i < TIMES_TO_COUNT)
 {
  // Iterate TIMES_TO_COUNT times
  // Increment the counter at each iteration
  (*count)++;
  i++;
 }
 // Print the final count when this thread
 // finishes its own count
 printf("%sThread [%ld]: Final count = %u.%s\n",
  BYELLOW, tid, *count, NC);
 return (NULL); // Thread ends here.
}

int main(void)
{
 pthread_t tid1;
 pthread_t tid2;
 // Variable to keep track of the threads' counts:
 unsigned int count;

 count = 0;
 // Since each thread counts TIMES_TO_COUNT times and that
 // we have 2 threads, we expect the final count to be
 // 2 * TIMES_TO_COUNT:
 printf("Main: Expected count is %s%u%s\n", GREEN,
     2 * TIMES_TO_COUNT, NC);
 // Thread creation:
 pthread_create(&tid1, NULL, thread_routine, &count);
 printf("Main: Created first thread [%ld]\n", tid1);
```

```
  pthread_create(&tid2, NULL, thread_routine, &count);
  printf("Main: Created second thread [%ld]\n", tid2);
  // Thread joining:
  pthread_join(tid1, NULL);
  printf("Main: Joined first thread [%ld]\n", tid1);
  pthread_join(tid2, NULL);
  printf("Main: Joined second thread [%ld]\n", tid2);
  // Final count evaluation:
  if (count != (2 * TIMES_TO_COUNT))
   printf("%sMain: ERROR ! Total count is %u%s\n", RED, count, NC);
  else
   printf("%sMain: OK. Total count is %u%s\n", GREEN, count, NC);
  return (0);
 }
```

Output:



```
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Expected count is 42000
Main: Created first thread [140656490104576]
Thread [140656490104576]: Count at thread start = 0.
Main: Created second thread [140656481711872]
Thread [140656490104576]: Final count = 21000.
Thread [140656481711872]: Count at thread start = 21000.
Main: Joined first thread [140656490104576]
Thread [140656481711872]: Final count = 42000.
Main: Joined second thread [140656481711872]
Main: OK. Total count is 42000
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Expected count is 42000
Main: Created first thread [139865822959360]
Main: Created second thread [139865814566656]
Thread [139865822959360]: Count at thread start = 0.
Thread [139865814566656]: Count at thread start = 7935.
Thread [139865822959360]: Final count = 21000.
Thread [139865814566656]: Final count = 32073.
Main: Joined first thread [139865822959360]
Main: Joined second thread [139865814566656]
Main: ERROR ! Total count is 32073
master@master-Blade:~/Documents/codequoi/tests/philo$ 
```

Completely by chance, the first time we run the program, the result could be correct. But things aren’t always as they appear! The second time we run it, the result is totally incorrect. If we continue to run the program several more times in a row, we’ll even come to realize that it is wrong much more often than right… And it isn’t even predictably wrong: the final count varies a lot from one run to the next. So what is happening, here?

The Danger of Data Races

If we examine the results closely, we can see that the final count is correct if and only if the first thread finishes counting before the second one starts. Whenever their executions overlap, the result is wrong, and always less than the expected result.

So the problem is that both threads often access the same memory area at the same time. Let's say the count is currently 10. Thread 1 reads the value 10. More accurately, it copies that value 10 to its register in order to manipulate it. Then, it adds 1 to get a result of 11. But before it can save the result in the memory area pointed to by the count variable, thread 2 reads the value 10. Thread 2 then increments it to 11 as well. Both threads then save their result and there we have it! Instead of incrementing the count once for each thread, they ended up only incrementing it by one in total… That's why we're loosing counts and our final result is so wrong.

This situation is called a **data race**. It happens when a program is subject to the progression or timing of other uncontrollable events. It is impossible to predict if the operating system will choose the correct sequencing for our threads.

Indeed, if we compile the program with the **`-fsanitizer=thread`** and **`-g`** options and then run it, like this:

```
gcc -fsanitize=thread -g threads.c && ./a.out
```

We will get an alert: "WARNING: ThreadSanitizer: data race".

So is there a way to stop a thread from reading a value while another one modifies it? Yes, thanks to mutexes!

What is a Mutex ?

A mutex (short for " **mut** ual **ex** clusion") is a synchronization primitive. It is essentially a lock that allows us to regulate access to data and prevent shared resources being used at the same time.

We can think of a mutex as the lock of a bathroom door. One thread locks it to indicate that the bathroom is occupied. The other threads will just have to patiently stand in line until the door is unlocked before they can take their turn in the bathroom.

Declaring a Mutex

Thanks to the `<pthread.h>` header, we can declare a mutex type variable like this:

```
pthread_mutex_t    mutex;
```

Before we can use it, we first need to initialize it with the `pthread_mutex_init` function which has the following prototype:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexattr);
```

There are two parameters to supply:

- **mutex**: the pointer to a variable of `pthread_mutex_t` type, the mutex we want to initialize.
- **mutexattr**: a pointer to specific attributes for the mutex. We will not worry about this parameter here, we can just say `NULL` .

The `pthread_mutex_init` function only ever returns 0.

Locking and Unlocking a Mutex

Then, in order to lock and unlock our mutex, we need two other functions. Their prototypes are as follows:

```
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

If the mutex is unlocked, **`pthread_mutex_lock`** locks it and the calling thread becomes its owner. In this case, the function ends immediately. However, if the mutex is already locked by another thread, **`pthread_mutex_lock`** suspends the execution of the calling thread until the mutex is unlocked.

The **`pthread_mutex_unlock`** function unlocks a mutex. The mutex to be unlocked is assumed to be locked by the calling thread, and the function only sets it to unlocked. Let’s be careful to note that this function does not check if the mutex is in fact locked and that the calling thread is actually its owner: a mutex could therefore be unlocked by a thread that did not lock it in the first place. We will need to be careful about arranging **`pthread_mutex_lock`** and **`pthread_mutex_unlock`** in our code, otherwise, we might get “lock order violation” errors.

Both of these functions return 0 for success and an error code otherwise.

Destroying a Mutex

When we no longer need a mutex, we should destroy it with the following **`pthread_mutex_destroy`** function:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

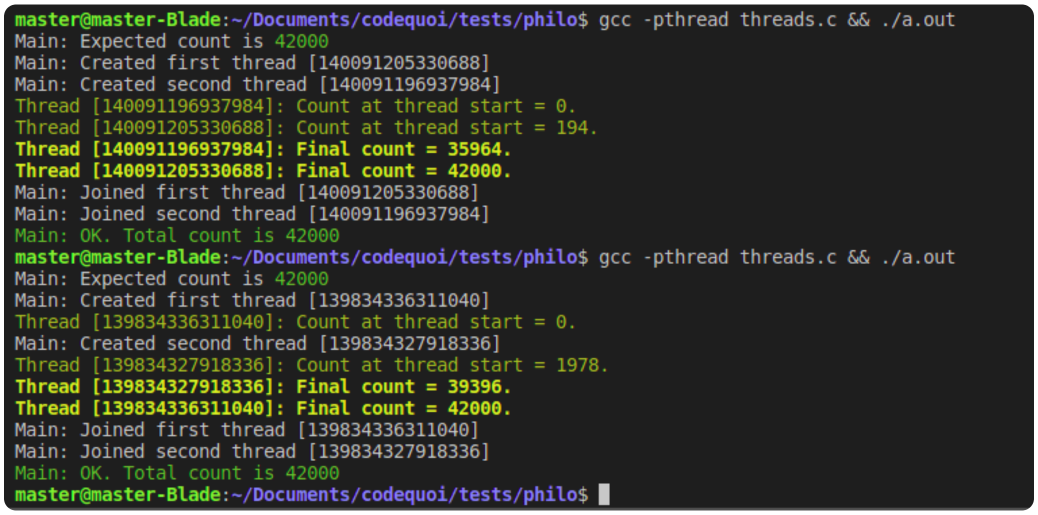
This function destroys an *unlocked* mutex, freeing whatever resources it might hold. In the LinuxThreads implementation of POSIX threads, no resources are associated with mutexes. In that case, **`pthread_mutex_destroy`** doesn't do anything other than check that the mutex isn't locked.

Example Implementation of a Mutex

We can now solve our previous example's problem of incorrect final count by using a mutex. For this, we need to create a small structure that will contain our **`count`** variable and the mutex that will protect it. We can then pass this structure to our threads' routines.

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

// Each thread will count TIMES_TO_COUNT times
#define TIMES_TO_COUNT 21000

#define NC "\e[0m"
#define YELLOW "\e[33m"
#define BYELLOW "\e[1;33m"
#define RED "\e[31m"
#define GREEN "\e[32m"

// This structure contains the count as well as the mutex
// that will protect the access to the variable.
typedef struct s_counter
{
 pthread_mutex_t count_mutex;
 unsigned int count;
} t_counter;

void *thread_routine(void *data)
{
 // Each thread starts here
 pthread_t tid;
 t_counter *counter; // pointer to the structure in main
 unsigned int i;

 tid = pthread_self();
 counter = (t_counter *)data;
 // Print the count before this thread starts iterating.
 // In order to read the value of count, we lock the mutex:
 pthread_mutex_lock(&counter->count_mutex);
 printf("%sThread [%ld]: Count at thread start = %u.%s\n",
  YELLOW, tid, counter->count, NC);
 pthread_mutex_unlock(&counter->count_mutex);
 i = 0;
 while (i < TIMES_TO_COUNT)
 {
  // Iterate TIMES_TO_COUNT times
  // Increment the counter at each iteration
  // Lock the mutex for the duration of the incrementation
  pthread_mutex_lock(&counter->count_mutex);
  counter->count++;
  pthread_mutex_unlock(&counter->count_mutex);
  i++;
 }
 // Print the final count when this thread finishes its
 // own count, without forgetting to lock the mutex:
 pthread_mutex_lock(&counter->count_mutex);
 printf("%sThread [%ld]: Final count = %u.%s\n",
  BYELLOW, tid, counter->count, NC);
 pthread_mutex_unlock(&counter->count_mutex);
 return (NULL); // Thread termine ici.
}

int main(void)
{
 pthread_t tid1;
 pthread_t tid2;
 // Structure containing the threads' total count:
 t_counter counter;

 // There is only on thread here (main thread), so we can safely
 // initialize count without using the mutex.
 counter.count = 0;
 // Initialize the mutex :
 pthread_mutex_init(&counter.count_mutex, NULL);
 // Since each thread counts TIMES_TO_COUNT times and that
 // we have 2 threads, we expect the final count to be
 // 2 * TIMES_TO_COUNT:
 printf("Main: Expected count is %s%u%s\n", GREEN,
     2 * TIMES_TO_COUNT, NC);
 // Thread creation:
 pthread_create(&tid1, NULL, thread_routine, &counter);
 printf("Main: Created first thread [%ld]\n", tid1);
 pthread_create(&tid2, NULL, thread_routine, &counter);
 printf("Main: Created second thread [%ld]\n", tid2);
 // Thread joining:
 pthread_join(tid1, NULL);
 printf("Main: Joined first thread [%ld]\n", tid1);
 pthread_join(tid2, NULL);
 printf("Main: Joined second thread [%ld]\n", tid2);
 // Final count evaluation:
 // (Here we can read the count without worrying about
 // the mutex because all threads have been joined and
 // there can be no data race between threads)
 if (counter.count != (2 * TIMES_TO_COUNT))
  printf("%sMain: ERROR ! Total count is %u%s\n",
     RED, counter.count, NC);
 else
  printf("%sMain: OK. Total count is %u%s\n",
     GREEN, counter.count, NC);
 // Destroy the mutex at the end of the program:
 pthread_mutex_destroy(&counter.count_mutex);
 return (0);
}
```

Let's see if our result is still incorrect now:



```
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Expected count is 42000
Main: Created first thread [140091205330688]
Main: Created second thread [140091196937984]
Thread [140091196937984]: Count at thread start = 0.
Thread [140091205330688]: Count at thread start = 194.
Thread [140091196937984]: Final count = 35964.
Thread [140091205330688]: Final count = 42000.
Main: Joined first thread [140091205330688]
Main: Joined second thread [140091196937984]
Main: OK. Total count is 42000
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Expected count is 42000
Main: Created first thread [139834336311040]
Thread [139834336311040]: Count at thread start = 0.
Main: Created second thread [139834327918336]
Thread [139834327918336]: Count at thread start = 1978.
Thread [139834327918336]: Final count = 39396.
Thread [139834336311040]: Final count = 42000.
Main: Joined first thread [139834336311040]
Main: Joined second thread [139834327918336]
Main: OK. Total count is 42000
master@master-Blade:~/Documents/codequoi/tests/philo$ 
```

There! Our result is now always right, every time we run the program, even if the second thread starts counting before the first is finished.

Beware of Deadlocks

However, mutexes can often provoke **deadlocks**. It's a situation in which each thread waits for a resource held by another thread. For example, thread T1 acquired mutex M1 and is waiting for mutex M2. Meanwhile thread T2 acquired mutex M2 and is waiting for mutex M1. In this situation, the program stays perpetually pending and must be killed.

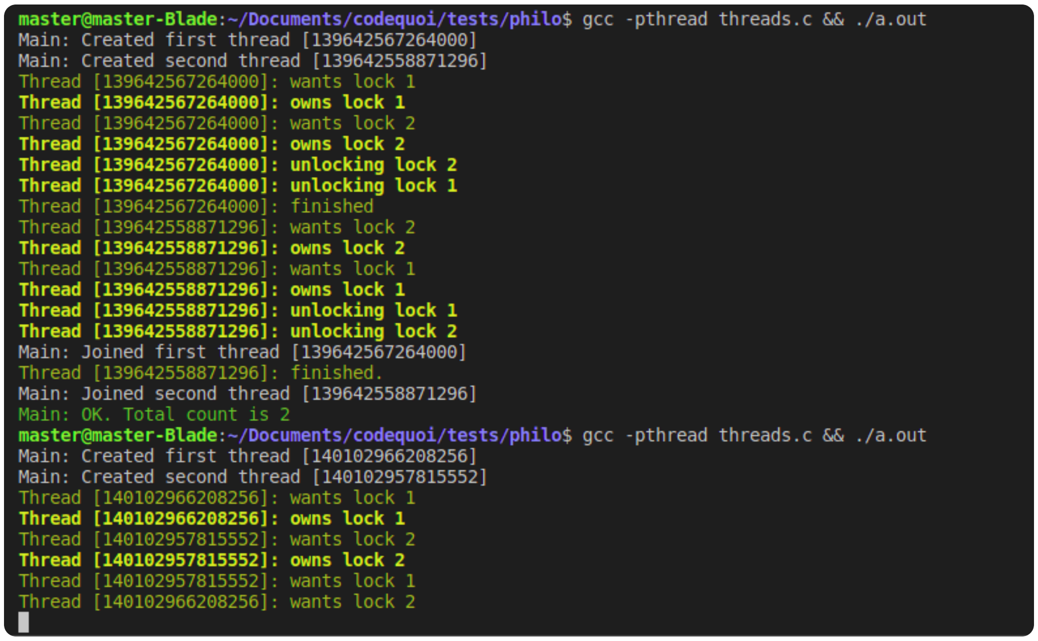
A deadlock can also happen when a thread is waiting for a mutex that it already owns!

Let’s try to demonstrate a deadlock. In this example, we will have two threads that need to lock two mutexes, **`lock_1`** and **`lock_2`** before being able to increment a counter. The routines of the two threads will be slightly different: the first thread will lock **`lock_1`** first, while thread 2 will start by locking **`lock_2`** …

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

#define NC "\e[0m"
#define YELLOW "\e[33m"
#define BYELLOW "\e[1;33m"
#define RED "\e[31m"
#define GREEN "\e[32m"

typedef struct s_locks
{
 pthread_mutex_t lock_1;
 pthread_mutex_t lock_2;
 unsigned int count;
} t_locks;

// The first thread invokes this routine:
void *thread_1_routine(void *data)
{
 pthread_t tid;
 t_locks  *locks;

 tid = pthread_self();
 locks = (t_locks *)data;
 printf("%sThread [%ld]: wants lock 1%s\n", YELLOW, tid, NC);
 pthread_mutex_lock(&locks->lock_1);
 printf("%sThread [%ld]: owns lock 1%s\n", BYELLOW, tid, NC);
 printf("%sThread [%ld]: wants lock 2%s\n", YELLOW, tid, NC);
 pthread_mutex_lock(&locks->lock_2);
 printf("%sThread [%ld]: owns lock 2%s\n", BYELLOW, tid, NC);
 locks->count += 1;
 printf("%sThread [%ld]: unlocking lock 2%s\n", BYELLOW, tid, NC);
 pthread_mutex_unlock(&locks->lock_2);
 printf("%sThread [%ld]: unlocking lock 1%s\n", BYELLOW, tid, NC);
 pthread_mutex_unlock(&locks->lock_1);
 printf("%sThread [%ld]: finished%s\n", YELLOW, tid, NC);
 return (NULL); // The thread ends here.
```

```
}

// The second thread invokes this routine:
void *thread_2_routine(void *data)
{
 pthread_t tid;
 t_locks  *locks;

 tid = pthread_self();
 locks = (t_locks *)data;
 printf("%sThread [%ld]: wants lock 2%s\n", YELLOW, tid, NC);
 pthread_mutex_lock(&locks->lock_2);
 printf("%sThread [%ld]: owns lock 2%s\n", BYELLOW, tid, NC);
 printf("%sThread [%ld]: wants lock 1%s\n", YELLOW, tid, NC);
 pthread_mutex_lock(&locks->lock_1);
 printf("%sThread [%ld]: owns lock 1%s\n", BYELLOW, tid, NC);
 locks->count += 1;
 printf("%sThread [%ld]: unlocking lock 1%s\n", BYELLOW, tid, NC);
 pthread_mutex_unlock(&locks->lock_1);
 printf("%sThread [%ld]: unlocking lock 2%s\n", BYELLOW, tid, NC);
 pthread_mutex_unlock(&locks->lock_2);
 printf("%sThread [%ld]: finished.%s\n", YELLOW, tid, NC);
 return (NULL); // The thread ends here.
}

int main(void)
{
 pthread_t tid1; // ID of the first thread
 pthread_t tid2; // ID of the second thread
 t_locks  locks; // Structure containing 2 mutexes

 locks.count = 0;
 // Initialize both mutexes :
 pthread_mutex_init(&locks.lock_1, NULL);
 pthread_mutex_init(&locks.lock_2, NULL);
 // Thread creation:
 pthread_create(&tid1, NULL, thread_1_routine, &locks);
 printf("Main: Created first thread [%ld]\n", tid1);
 pthread_create(&tid2, NULL, thread_2_routine, &locks);
 printf("Main: Created second thread [%ld]\n", tid2);
 // Thread joining:
 pthread_join(tid1, NULL);
 printf("Main: Joined first thread [%ld]\n", tid1);
 pthread_join(tid2, NULL);
 printf("Main: Joined second thread [%ld]\n", tid2);
 // Final count evaluation:
 if (locks.count == 2)
  printf("%sMain: OK. Total count is %d\n", GREEN, locks.count);
 else
  printf("%sMain: ERROR ! Total count is %u\n", RED, locks.count);
 // Mutex destruction:
 pthread_mutex_destroy(&locks.lock_1);
 pthread_mutex_destroy(&locks.lock_2);
 return (0);
}
```

As we can see in the following output, most of the time, there is no problem with this configuration because the first thread has a small head start on the second one. But sometimes, both threads lock their first mutexes exactly at the same time, in which case the program stays blocked because the threads are caught in a deadlock.



```
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Created first thread [139642567264000]
Main: Created second thread [139642558871296]
Thread [139642567264000]: wants lock 1
Thread [139642567264000]: owns lock 1
Thread [139642567264000]: wants lock 2
Thread [139642567264000]: owns lock 2
Thread [139642567264000]: unlocking lock 2
Thread [139642567264000]: unlocking lock 1
Thread [139642567264000]: finished
Thread [139642558871296]: wants lock 2
Thread [139642558871296]: owns lock 2
Thread [139642558871296]: wants lock 1
Thread [139642558871296]: owns lock 1
Thread [139642558871296]: unlocking lock 1
Thread [139642558871296]: unlocking lock 2
Main: Joined first thread [139642567264000]
Thread [139642558871296]: finished.
Main: Joined second thread [139642558871296]
Main: OK. Total count is 2
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Created first thread [140102966208256]
Main: Created second thread [140102957815552]
Thread [140102966208256]: wants lock 1
Thread [140102966208256]: owns lock 1
Thread [140102957815552]: wants lock 2
Thread [140102957815552]: owns lock 2
Thread [140102957815552]: wants lock 1
Thread [140102966208256]: wants lock 2
```

Studying this second result, we can clearly see that the first thread locked `lock_1` and the second locked `lock_2`. The first thread now wants to lock `lock_2` and the second wants `lock_1`, but neither have any way of getting those mutexes. They are deadlocked.

Dealing with Deadlocks

There are several ways to deal with deadlocks like these. Among other things, we can:

- ignore them, but only if we can prove that they will never happen. For example when the time intervals between resource requests are very long and far between.
- correct them when they happen by killing a thread or by redistributing resources, for example.
- prevent and correct them before they happen.
- avoid them by imposing a strict order for resource acquisition. This is the solution to our previous example: the threads should both ask for `lock_1` first.
- avoid them by forcing a thread to release a resource before asking for new ones, or before renewing its request.

There is no "best" solution that will solve all deadlock cases. The best method to deal with a particular deadlock depends on the situation.

Tips for Testing a Program's Threads

The most important thing to remember when testing any program that makes use of threads, is to test the same thing many times in a row. Oftentimes, synchronization errors won't be apparent on the first or second or even third run. It depends on the order the operating system chooses for the execution of each thread. By running the same test over and over, we may see a large variation in the results.

There are some tools we can use to help us detect thread-related errors like possible data races, deadlocks and lock order violations:

- The `-fsanitize=thread -g` flag we can add at compilation. The `-g` option displays the specific files and line numbers involved.
- The thread error detection tool Helgrind that we can run our program with, like this: `valgrind --tool=helgrind ./philo <args>`.
- DRD, another thread error detection tool that we can also run our program with, like this: `valgrind --tool=drd ./philo <args>`.

Beware of using both `-fsanitize=thread` and valgrind, they do not play well together!

And as always, we can't forget to check for memory leaks with `-fsanitize=address` and `valgrind`!

A little tip to share, a nagging question to ask, or a strange discovery to discuss about threads or mutexes? I'd love to read and respond to it all in the comments. Happy coding !

Sources and Further Reading

- Bryant, R., O'Hallaron, D., 2016, *Computer Systems: a Programmer's Perspective*, Chapter 12: Concurrent Programming, p. 1007 - 1076
- Arpaci-Dusseau R., Arpaci-Dusseau, A., 2018, *Operating Systems: Three Easy Pieces*, Part II: Concurrency [OSTEP]
- Wikipedia, *Concurrent computing* [Wikipedia]
- Wikipedia, *Mutual exclusion* [Wikipedia]
- The Linux Programmer Manual:
  - *pthread_create(3)* [man]
  - *pthread_join(3)* [man]
  - *pthread_detach(3)* [man]
  - *pthread_mutex_init/lock/unlock(3)* [man]
- Valgrind User Manual, *Helgrind: a thread error detector* [valgrind.org]
- Valgrind User Manual, *DRD: a thread error detector* [valgrind.org]

Tags : Concurrency Data race Deadlock Mutex Thread

Comments

Your comment here

Sign In

Styling with **Markdown** is supported

Sort by **Recently updated**

Related Posts

How to Prepare for the 42 Piscine

>_ Mia Combeau >_ Methodology

One year ago to the day, I crossed the threshold of 42 school in Paris for the very first time.

Read More

The Internet's Layered Network Architecture

>_ Mia Combeau >_ Internet | Network

We all know the Internet. It’s the network that enables data transfer on a global scale.

Read More

Why I No Longer Write Articles About 42 School Projects

>_ Mia Combeau >_ Methodology

Following a meeting with 42 school's pedagogical team, I decided to remove all articles directly related to 42 projects.

Read More

About Legal Notice