CS-118-02 Week 12

Varun Narravula

2022-04-18

Avoiding Race Conditions With Mutexes

In a race condition, let's say one thread is reading and another is writing. There is no locking, so if the reader thread reads before the other thread writes, then it will get the value before the write (which may not make sense). This is why you need locking. In the `pthread` library, there is something called a "mutex", or a "**mut**ual **ex**clusion" lock, where only one thread can lock the mutex, use any resources it needs related to it while it is locked, and unlock it after. Let's see this in action; say you have an array on the heap that is a globally accessible variable. A mutex associated with the array would look like this:

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

int *array;
pthread_mutex_t array_lock;

int main() {
  array = (int *)malloc(sizeof(int) * 4);

  int err = pthread_mutex_init(&array_lock, NULL);
  if (err) {
    printf("Mutex initialization failed\n");
    return err;
  }

  err = pthread_mutex_destroy(&array_lock);
  if (err) {
    printf("Mutex destruction failed\n");
    return err;
  };

  return 0;
}
```

It looks similar to initializing a thread with `pthread_create`, but simpler. An `int` array of size 4 is initialized with `malloc`, and an associated variable of type `pthread_mutex_t` (the mutex lock itself) is initialized with `pthread_mutex_init`, a function that takes two arguments: a reference to where to keep the mutex, and mutex attributes (kind of like thread attributes, also assigned `NULL` here for defaults). Notice that the mutex does not have any data in itself to store; it is not a container for data. Think of a real-life lock; locks do not actually house the data (unless you have a weirdly large lock that stores things, I can't help you there), but only prevent access to things it secures. The same concept is here: the mutex is only a lock for data that needs to be manually associated with a piece of data by you, the programmer.

When you want to lock access to some data using a mutex, you do it by first calling `pthread_mutex_lock` with a reference to the mutex as an argument. When a thread calls this function, and the mutex is not already locked, then the thread locks it and moves on. However, if a thread tries to lock an already existing mutex, it is blocked from doing so, and the thread is blocked and is placed in a queue for that mutex. This is what allows for safety from race conditions; if the thread can only proceed when it locks the mutex, then it is guaranteed access to the data behind the mutex that will not rely on race conditions in order to access/modify it properly. Eventually, when you are done using those resources, you will need to unlock the mutex you have locked in order to give access to other threads. This is done using `pthread_mutex_unlock` or `pthread_mutex_destroy` (which destroys the mutex instance, though it can be reinitialized with a new, unrelated mutex instance; more often than not you will probably only use `pthread_mutex_destroy` to get rid of the mutex when the thread that was using it is joined back completely).

Here's a common question: why can't you use a single boolean variable for this? Technically, yes you can, but then you would have to manage all of the complexity behind managing the different threads that attempt to lock an already-locked thread by implementing a scheduler, and you would have reinvented a part of Linux itself, along with a crude implementation of a mutex. The `pthread` library takes care of interfacing with the OS's scheduler for you with its mutex implementation, so use it. Don't try and reinvent the wheel unless you are doing embedded programming without an OS, which is a whole other world that does require reimplementing standard library pieces at times.

Let's see an example of locking a resource in action. In the C++ standard library, there is a collection of data structures and methods called the Standard Template Library, or STL, which contains many useful implementations of data structures and other objects, such as dynamic-size lists (`std::vector`), queues (`std::queue`), and even object-oriented wrappers around C structs such as `pthread` (`std::thread`), among many other things (For this last one, though, try to stick with using the C-style methods for managing threads, like the ones mentioned above for now). However, these data structures are not thread-safe. How do we work with these when multithreading then? You would have to

manage threading by hand, which is more difficult, but still doable. Let's instantiate a global instance of a vector with an initial size of 100 and fill it with zeros on a separate thread:

```
#include <iostream>
#include <vector>
#include <pthread.h>

struct args {
  std::vector<int>* list;
  pthread_mutex_t *mutex;
};

void* initialize_vector(void *arg) {
  auto p = reinterpret_cast<args *>(arg);

  pthread_mutex_lock(p->mutex);

  *(p->list) = std::vector<int>(100, 0);

  pthread_mutex_unlock(p->mutex);

  return NULL;
}

int main() {
  std::vector<int> x;
  pthread_mutex_t m;

  int err = pthread_mutex_init(&m, NULL);
  if (err) {
    std::cout << "Mutex initialization failed" << "\n";
    return err;
  }

  struct args d = {&x, &m};

  pthread_t thread;

  err = pthread_create(&thread, NULL, &initialize_vector, &d);
  if (err) {
    std::cout << "Thread initialization failed" << "\n";
    return err;
  }

  err = pthread_join(thread, NULL);
```

```
  if (err) {
    std::cout << "Thread join failed" << "\n";
    return err;
  }

  err = pthread_mutex_destroy(&m);
  if (err) {
    std::cout << "Mutex destruction failed" << "\n";
  }

  return 0;
}
```

This builds on the very first `pthread_create` example from last time, and initializes the vector passed to the thread instead of sleeping for a number of seconds in a thread-safe manner. However, what would happen if some other thread tried to initialize the vector variable without checking if the mutex is locked? Well, they can actually do that without causing any errors directly, because the mutex and array themselves are not related as far as the compiler is concerned. It is your responsibility to check if the mutex that corresponds to the vector is locked, and if you fail to do so, you'll probably start seeing race conditions sooner or later.

The section that initializes the vector between the locking and unlocking of the mutex is called the "critical section"; in this part of code, it is critical (see what I did there?) to ensure access is only done by one thread at a time. However, for most situations, there is an exception to this. There can be multiple "reader" threads that only ever read data from a single data source, but only one thread at a time should be able to write to it. This is only semi-related, but in a language called Rust, this is actually enforced by the compiler for even primitive language constructs. A mutable variable can have any number of immutable references, but can only have one mutable reference that you can change the value of the variable with. While this seems extremely limiting at first, it proves to be extremely powerful and scalable for safe concurrency applications.

Important terminology: You may have noticed that I use the terms "blocking", "non-blocking", "synchronous", and "asynchronous". When I say a statement or function is "blocking" or "synchronous", this means that the statement needs to finish executing before the thread can continue executing the next statements. "Non-blocking" or "asynchronous" means that the statement starts immediately and does not do this, allowing you to continue on to the next statement immediately without necessarily finishing. An example of asynchronous execution is running a function using a separate thread; when you launch a thread using `pthread_create`, it does not have to finish before you go on. Examples of synchronous execution include `pthread_join` and `pthread_mutex_lock`; this function stops execution of the thread it was called on until the thread is joined and its resources are cleaned up.