Synchronization, Part 1: Mutex Locks

Scott Bigelow edited this page on Mar 11 · 4 revisions

What is a Critical Section?

A critical section is a section of code that can only be executed by one thread at a time, if the program is to function correctly. If two threads (or processes) were to execute code inside the critical section at the same time then it is possible that program may no longer have correct behavior.

Is just incrementing a variable a critical section?

Possibly. Incrementing a variable (`i++`) is performed in three individual steps: Copy the memory contents to the CPU register. Increment the value in the CPU. Store the new value in memory. If the memory location is only accessible by one thread (e.g. automatic variable `i` below) then there is no possibility of a race condition and no Critical Section associated with `i`. However the `sum` variable is a global variable and accessed by two threads. It is possible that two threads may attempt to increment the variable at the same time.

```c
#include <stdio.h>
#include <pthread.h>
// Compile with -pthread

int sum = 0; //shared

void *countgold(void *param) {
    int i; //local to each thread
    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

Clone this wiki locally

https://github.com/angrave/SystemPr

Clone in Desktop

Typical output of the above code is `ARGGGH sum is 8140268` A different sum is printed each time the program is run because there is a race condition; the code does not stop two threads from reading-writing `sum` at the same time. For example both threads copy the current value of sum into CPU that runs each thread (let's pick 123). Both threads increment one to their own copy. Both threads write back the value (124). If the threads had accessed the sum at different times then the count would have been 125.

How do I ensure only one thread at a time can access a global variable?

You mean, "Help - I need a mutex!" If one thread is currently inside a critical section we would like another thread to wait until the first thread is complete. For this purpose we can use a mutex (short for Mutual Exclusion).

For simple examples the smallest amount of code we need to add is just three lines:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
pthread_mutex_unlock(&m); //end of Critical Section
```

Once we are finished with the mutex we should also call `pthread_mutex_destroy(&m)` too. Note, you can only destroy an unlocked mutex. Calling destroy on a destroyed lock, initializing an initialized lock, locking an already locked lock, unlocking an unlocked lock etc are unsupported (at least for default mutexes) and usually result in undefined behavior.

If I lock a mutex, does it stop all other threads?

No, the other threads will continue. It's only when a thread attempts to lock a mutex that is already locked, will the thread have to wait. As soon as the original thread unlocks the mutex, the second (waiting) thread will acquire the lock and be able to continue.

Are there other ways to create a mutex?

Yes. You can use the macro PTHREAD_MUTEX_INITIALIZER only for global ('static') variables. m = PTHREAD_MUTEX_INITIALIZER is equivalent to the more general purpose `pthread_mutex_init(&m,NULL)`. The init version includes options to trade performance for additional error-checking and advanced sharing options.

```
pthread_mutex_t *lock = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
//later
pthread_mutex_destroy(lock);
free(lock);
```

So `pthread_mutex_lock` stops the other threads when they read the same variable?

No. A mutex is not that smart - it works with code (threads), not data. Only when another thread calls `lock` on a locked mutex will the second thread need to wait until the mutex is unlocked.

Can I create mutex before fork-ing?

Yes - however the child and parent process will not share virtual memory and each one will have a mutex independent of the other.

(Advanced note: There are advanced options using shared memory that allow a child and parent to share a mutex if it's created with the correct options and uses a shared memory segment. See stackoverflow example)

If one thread locks a mutex can another thread unlock it?

No. The same thread must unlock it.

Can I use two or more mutex locks?

Yes! In fact it's common to have one lock per data structure that you need to update.

If you only have one lock then they may be significant contention for the lock between two threads that was unnecessary. For example if two threads were updating two different counters it might not be necessary to use the same lock.

However simply creating many locks is insufficient: It's important to be able to reason about critical sections e.g. it's important that one thread can't read two data structures while they are being updated and temporarily in an inconsistent state.

Is there any overhead in calling lock and unlock?

There is a small overhead amount of overhead of calling `pthread_mutex_lock` and `_unlock`; however this is the price you pay for correctly functioning programs!

Simplest complete example?

A complete example is shown below

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

// Compile with -pthread
// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
    int i;

    //Same thread that locks the mutex must unlock it
    //Critical section is just 'sum += 1'
    //However locking and unlocking a million times
    //has significant overhead in this simple answer

    pthread_mutex_lock(&m);

    // Other threads that call lock will have to wait until we call unlock

    for (i = 0; i < 10000000; i++) {
    sum += 1;
    }
    pthread_mutex_unlock(&m);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

In the code above, the thread gets the lock to the counting house before entering. The critical section is only the `sum+=1` so the following version is also correct but slower -

```
    for (i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&m);
        sum += 1;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}
```

This process runs slower because we lock and unlock the mutex a million times, which is expensive - at least compared with incrementing a variable. (And in this simple example we didn't really need threads - we could have added up twice!) A faster multi-thread example would be to add one million using an automatic(local) variable and only then adding it to a shared total after the calculation loop has finished:

```
    int local = 0;
    for (i = 0; i < 10000000; i++) {
       local += 1;
    }

    pthread_mutex_lock(&m);
    sum += local;
    pthread_mutex_unlock(&m);

    return NULL;
}
```

How do I find out more?

Play! Read the man page!

- pthread_mutex_lock man page
- pthread_mutex_unlock man page
- pthread_mutex_init man page
- pthread_mutex_destroy man page