TTK4147 - Real-time Systems
Exercise 3 - Concurrency

September 2020

Remember to check in at the lab, https://innsida.ntnu.no/checkin/room/3066, to allow for easier infection control in the event of a Covid-19 infection on campus.

In this exercise you will learn about the fundamental building blocks for concurrent execution – creating threads, and using semaphores to synchronize execution of threads.

If you have questions or want to get your assignment approved, please apply for a spot (using your NTNU-user) in the queue here: https://s.ntnu.no/rts.

Start the lab PC in generic Ubuntu (ie. not Xenomai) by selecting "Advanced options for Ubuntu" in the boot menu, then selecting the "117-generic" option (NOT recovery mode).

1 Threads

A uniprocessor computer can only do one thing at the time, but since it is able to execute operations extremely fast it can share its time between different processes and threads. This is called concurrency. To switch between two processes or two threads is called context switching and is performed fast enough so it appears as they are running at the same time.

In a multiprocessor system (dual/quad core) two processes or threads can run at the same time on different cores. This is called parallelism.

A process is an instance of a computer program. It consists of the program code and its current state. Each process can contain several threads that can execute their code in parallel. Threads within the same process share resources like memory, which means that communication between threads is simpler than between processes. For that reason, we will be using threads in this exercise, as well as most of the rest of the course.

For this exercise, we will be using the `clang` compiler, as it has better support for some advanced tooling (and as an extra bonus also gives better error messages).

TASK A

Create a program that contains a global `long` initialized to 0, and starts two threads. Each thread should create a local variable (also initialized to 0), and increment both the global and local variables in a for-loop (say, 50M iterations), then prints out both. Before terminating, the main function should wait until both threads complete by joining with the two threads. Here is some code to get you started:

```c
#include <pthread.h>

// Note the argument and return types: void*
void* fn(void* args){
    return NULL;
}

int main(){
    pthread_t threadHandle;
    pthread_create(&threadHandle, NULL, fn, NULL);
    pthread_join(threadHandle, NULL);
}
```

Function descriptions can be found here http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html

Compile with `clang -lpthread cfile.c`, and any of your other favorite flags. Run the program a few times, and observe the effects. What happens?

TASK B

You should have gotten some strange results from the previous task. Compile the same program again, but now add the flags `-g -fsanitize=thread`. Use Thread Sanitizer to help you find the problem. What was the problem?

2 Semaphores

A semaphore is a counter with atomic increment and decrement operations, which can be used to keep track of the number of available resources of some kind. Whenever a thread starts using one of the resources the semaphore is decremented. It is then incremented when the thread is no longer using the resource. If the counter in the semaphore is zero, there are no available resources and any threads that need to use a resource must wait. POSIX semaphores are defined by `semaphore.h`, and the most important functions are `sem_init()`, `sem_wait()` and `sem_post()`. More info can be found here http://pubs.opengroup.org/onlinepubs/7908799/xsh/semaphore.h.html

TASK C

Fix the problem you found in **TASK B** by using a semaphore to synchronize the threads' access to the problematic variable. Use time on the command line to compare the time the program takes to execute before and after you added the synchronization.

Which one uses more system time, and why?

And the same question, but user time?

Why is the real time used lower than the sum of the other two?

3 Double checked locking pattern

A popular design pattern is the singleton, an object that there can only exist a single one of during a program's lifespan. In order to ensure the single-ness of the singleton, it is not possible to create

a new instance of the singleton yourself, but you must instead call a function that returns a pointer to the one instance that does exist.

Now the question is: how do we construct the object to begin with? One common way is to use lazy initialization, where the function that returns the singleton pointer also initializes the singleton if it doesn't already exist. For example:

```
static struct Singleton* g_singleton = NULL;

struct Singleton* getSingleton(){
    if(!g_singleton){
        g_singleton = malloc(sizeof(struct Singleton));
        // Performing some initialization...
    }
    return g_singleton;
}
```

The problem with this is that it is not thread-safe: Two threads checking if the singleton pointer is null at the same time will both initialize the singleton, thereby creating more than one singleton. We must therefore use a semaphore (or other lock) to enforce that only one thread at a time can perform this initialization. This can be done like this:

```
static struct Singleton* g_singleton = NULL;
static sem_t initSem;

struct Singleton* getSingleton(){
    sem_wait(&initSem);
    if(!g_singleton){
        g_singleton = malloc(sizeof(struct Singleton));
        // Performing some initialization...
    }
    sem_post(&initSem);
    return g_singleton;
}
```

But now we have introduced a new problem. While using a semaphore solved the concurrent access, it adversely affected the performance for the most common case, which is when the singleton is already initialized and we just want to get the pointer. To alleviate this problem, the double-checked locking is introduced:

```
static struct Singleton* g_singleton = NULL;
static sem_t initSem;

struct Singleton* getSingleton(){
    if(!g_singleton){
        sem_wait(&initSem);
        if(!g_singleton){
            g_singleton = malloc(sizeof(struct Singleton));
            // Performing some initialization...
        }
        sem_post(&initSem);
    }
    return g_singleton;
}
```

Now, the expensive calls to the semaphore are only performed when the singleton is uninitialized, and the semaphore lets us protect against double initialization, by protecting concurrent reads and writes of the pointer in case a thread detects that it was uninitialized.

TASK D

Download and run the file `double_checked.c`. The compiler command is on the first line of the file, alternatively you can run the file directly as a "script" by running `./double_checked.c` (though you may have to run `chmod a+x double_checked.c` first).

See if you can spot the bug.

TASK E

Add `-fsanitize=thread` to the compiler arguments, and see if you can spot the bug now. It turns out that double checked locking is not really "fixable", at least not in a portable way. Also, it is a solution to a problem that doesn't really need to exist in the first place: just initialize the singleton once at the start of the program, or just live with the synchronization overhead. If you want to nerd out more, you can read what Andrei and Scott wrote about it: http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405726

4 The dining philosophers

Fixing problems where the output of a program depends on the order the operating system interleaves the threads is not always as simple as just slapping synchronization primitives at it until it behaves. Sometimes there is such a thing as too much synchronization, causing a deadlock.

Deadlock is a problem that occurs when two (or more) threads are blocking each other. For example, this can happen if thread A has locked resource 1 and wants to lock resource 2, while thread B has locked resource 2 and wants to lock resource 1. If both threads acquire one resource each, then neither thread can continue.

The dining philosophers problem is a classic example of a deadlock. First, read more about the problem here: https://en.wikipedia.org/wiki/Dining_philosophers_problem

TASK F

Create a program that implements the dining philosophers problem. The philosophers should be implemented as threads, and the forks should be mutexes. A mutex is a binary semaphore (ie. can only be 0 or 1) with ownership semantics, meaning it can only be unlocked by the same thread that locked it. By using a mutex instead of a semaphore, we can be sure that the philosopher that puts down a fork is the same one that picked it up. An example of Mutex-usage is shown here:

```
#include <pthread.h>

int main(){
    pthread_mutex_t mtx;

    // 2nd arg is a pthread_mutexattr_t
    pthread_mutex_init(&mtx, NULL);

    pthread_mutex_lock(&mtx);
    // Critical section
    pthread_mutex_unlock(&mtx);

    pthread_mutex_destroy(&mtx);
}
```

First, implement it naiively – you should end up with a deadlock. Then, fix the deadlock. There are several ways to do this – choose the solution you want. Make sure you are compiling with the Thread Sanitizer enabled to make sure you don't have any data races in your program.

If you also want to check for pointer bugs with Address Sanitizer, note that you can only run one of the sanitizers at a time.