Seminar 3 Report

Cem Cebeci

December 12, 2019

1 Introduction

This project is an implementation of a library for managing concurrent threads. The implementation will be described in parts. We will start with a very simple library, indicate some problems our library can not handle and introduce new tools do deal with those problems. Ultimately, we aim to mimic the Linux `pthreads` library.

2 Basic Implementation

Let us begin with describing the functionality we want to have. We want to have multiple threads running on the CPU. The threads should be different instances of the same program that share the same address space. This means that they should share their heap, global and static variables and code segment. Since the threads will belong to the same process, they will also share the same open file table and other per-process attributes assigned by the operating system. However, the threads should have different stack pointers and program counters to be able to run without interfering with each others' control flow.

Then, there are two main questions we need to answer:

- How do we keep track of each thread's program counter and stack pointer?
- How do we manage the control flow between threads?

2.1 Contexts

Of course, we could manually attribute a stack pointer and a program counter to every context and it would be relatively simple to manage the stack pointers. The program counters, however, would be a bit more challenging since we want to implement our library entirely in C and since C is a structured programming language, it does not (by default) have the tools to access the program counter and jump to an arbitrary location in the code segment. Therefore, we will use the `ucontext` library to assign a new context with a freshly allocated stack to every thread. One fact we should make note of is that every `ucontext_t` has its own signal mask. This is not relevant at all for the problem we are discussing right now but will be very important in a later section.

2.2 Control Flow

Let us define being in control. A thread being in control means that the CPU has that thread's context's values in it's dedicated registers (including but not limited to the stack pointer and the program counter).

One possible way of managing the control flow is to have a controller thread that keeps a record of all other current threads in some data structure. We could make sure that every other thread takes the control from the controller thread and yields the control back to it. This intuitive approach would be much more manageable than the approach we are going to take. However, this solution is not discussed in the assignment description and I have neither the time nor the courage to not follow the assignment's instructions.

The approach we took in the actual implementation is to keep a list of all the threads that are ready to run. Since we chose to implement that list as a queue, we will refer to that list as the ready queue. Note that we should also make sure that every suspended thread is pointed to by at least one thread in the ready queue.

In our implementation, we have two global pointers `running` and `end` that point to the front and the back of the ready queue respectively. Since those pointers are global, every thread can enqueue and dequeue new threads to the queue. In the ready queue's consistent state, running points to the thread in control.

The only way for a thread to be suspended in the basic implementation is calling the `green_join()` method. In that case, we will dequeue the running thread and store a pointer to it in the thread it's meant to join. When any thread finishes executing, it will add the thread joining it back to the ready queue.

Whenever a thread yields control, it changes `running` the next thread in the ready queue and requests a context switch to running's attributed context.

3 Timer Interrupts

One problem with the current implementation is that the only way for the running thread to change is the running thread to yield control willingly. This could cause a problem in two different ways. Firstly, one of the threads could be malicious or poorly written and never yield control, locking the process indefinitely. Secondly, if the threads depend on other each other to modify some global variables, the process could be stuck in a loop forever unexpectedly because no other thread will be able to change any global variable while a thread is in a loop waiting for that variable to change. Therefore, we need to make sure that the running thread changes regularly.

To be able to change the thread in regular intervals, we need some operating system support again. We simply configure a timer to interrupt the process regularly and in the ISR, we simply make the current thread yield control.

The problem with our very simple solution is that we need to make sure that the running thread hands the control over to the next thread with the ready queue in a consistent state. With timer interrupts at arbitrary points, we can not be sure of that. Therefore we need to block the interrupts signals we get whenever we are executing critical segments of code that handle the ready queue.

3.1 An important clarification

Here is an extremely abstract outline of how a context yields control:

```
block signals
update the ready queue
request a context switch
unblock signals
```

One might wonder: Would the thread we're switching to not be blocked entirely, since we do not unblock until we get back in control?

The answer is no, it would not. Because each context has its own signal mask.

4 Conditional Variables

Now processes can depend on each other to change some flags safely, because the running process changes with regular intervals. However, we still have another problem to fix regarding those dependencies. Suppose a user wants to write a program in which two threads take turns modifying a global variable. The user could of course implement this using a global flag and spinlocks. However, that strategy would cause the user to waste on average half of the processor's clock cycles in a loop that accomplishes nothing. As a better solution to that problem, we want threads to tell each other that some global conditions (such as flags) have changed.

We will allow a thread to suspend itself, waiting on a **conditional variable**. Conditional variables will have some data structure to keep track of the threads waiting for them to change. In our implementation those data structures are stacks but that decision is completely arbitrary. Other threads will be able to put the threads waiting on conditional variables to the ready queue when they change the global condition. That way, the users can mutually dependent threads much more efficiently.

5 Mutex Locks

Now our user can efficiently handle dependencies on global flags but what if the user wants to update the same global variable from two different threads concurrently?

The problem lies in the fact that updating a variable is not an atomic operation and since our threads can be interrupted almost anywhere now, they might end up reading a global value before another thread finished updating it, which definitely would lead to some errors.

A simple solution to that is adding **mut**ually **ex**clusive locks. A mutex lock is taken by at most one thread at a given time. We make sure that no two threads try to take the lock at the same time by blocking the interrupts while a process tries to take the lock. If a thread updates shared variables only when it has the lock, we avoid any concurrency issues.

In the implementation, each a mutex lock keeps track of all the threads trying to take it in a stack (arbitrarily chosen). When a threads releases the lock, it passes the lock to the next waiting thread by simply popping it from the stack and adding it to the ready queue without changing the lock's taken status.

6 Atomic Release - Signal

There is one last problem left to solve with the implementation. Suppose thread a takes a lock and suspends on a condition and thread b waits tries to take the lock thread a took earlier. Thread a should definitely release the lock before suspending or thread b will never signal thread a and the system will be in a deadlock. However, if thread a gets interrupted after releasing the lock and before suspending on the condition, thread b might end up signaling thread a before it suspends and when thread a does suspend, there is no one left to signal thread a back, resulting in a deadlock again.

The only way to solve that situation is to have an atomic operation that releases the lock and suspends on the condition. Obviously, we can not add an atomic instruction to the architecture but we can fake atomicity by blocking the interrupt signals just as we did with the mutex locks.

The small modification we need to make is to change `green_cond_wait` to take a mutex argument (`NULL` by default) and make the thread release the lock before suspending and take it back before returning from the `green_cond_wait` method. This way, we will never have a suspended thread holding a lock.