ID1206 Operating Systems
Seminar III: Green Threads

Larisa Cof

January 2021

1 Introduction

The purpose of this assignment was to implement a green threads library and conduct testing using a benchmark program.

In order to handle multiple threads, a scheduler and context handler were implemented. Contexts were managed explicitly to implement something that behaves similar to the pthread library in order to get a better understanding of how threads work on the inside.

A timer driven scheduling event was was added which allowed several threads to execute concurrently. Although, by adding a timer driven scheduling, interrupts at any point in the execution were enabled which led to a problem when the shared data structures were updated. For this reason, a mutex construct was implemented in order to prevent race conditions.

2 Managing Contexts

The functions implemented in order to mimic the pthread library were:

- green_create(): initializes a green thread.
- green_yield(): suspends the current thread and selects a new thread for execution.
- green_join(): the current thread is suspended waiting for a thread to terminate.

The library functions that were used were getcontext(), makecontext(), setcontext() and swapcontext().

In order to execute different threads on a CPU, the managing of contexts is of great importance. The handler that was implemented needed to keep track of the currently running thread as well as a set of suspended threads ready for execution. Furthermore, the handler also needed to keep track of terminated threads. This is because the creator of the thread should be able to pick up the result when calling green_join().

Representing a Thread

Threads were represented by a structure that held all the information needed in order to manage them. The structure held a pointer to the context of the thread.

Suspended threads were added to a linked list. Pointers for the thread that is waiting to join the thread to terminate and to the final result were implemented.

A status field, zombie, which indicated if a thread had terminated or not was also implemented.

Internal State

The internal state of the handler kept track of only two things: the running threads and the so-called ready queue. Also implemented was a global context in order to store the context of the initial running process.

2.1 Creating a Thread

The process of creating a new green thread consists of two stages. When calling green_create(), the user provides an uninitialized green_t structure, the function the thread should execute and pointer to its arguments. A new context is created, attached to the thread structure and finally the thread is added to the ready queue.

When scheduled, the thread should call the function. The thread is set to call the function green_thread() which is responsible for calling the function provided by the user. Specifically, the green_thread() function will do two things:

- start the execution of the real function.
- terminate the thread (after returning from the call).

2.2 Yield the Execution

The green_yield() function will suspend the currently running thread and enqueue it into the ready queue. Then, the next thread to run is swapped with the currently running thread.

2.3 The Join Operation

The green_join() function will be used by a thread that is waiting for another thread to terminate. The calling thread will be suspended and added to the waiting queue and another thread is selected for execution. When all threads are done executing, the first joined thread will begin. Lastly, the memory allocated by the zombie thread will be freed.

3 Suspending on a Condition

For this part of the assignment, conditional variables needed to be implemented and were intended to work as the conditional variables in the pthread library. Although, this initial implementation is simpler because at this point in the assignment, there were no mutex structures that could be locked.

The following functionalities were implemented:

- void green_cond_init(green_cond_t*): initialize a green condition variable.
- void green_cond_wait(green_cond_t*): suspend the current thread on the condition.
- void green_cond_signal(green_cond_t*): move the first suspended thread to the ready queue.

The wait procedure puts a thread to sleep and the signal procedure wakes a thread.

4 Adding a Timer Interrupt

In previous implementations, the library created thus far relied on the threads themselves to yield the execution or suspend on a conditional variable, before a new thread was scheduled for execution. Nevertheless, in order to allow multiple threads to execute concurrently, a timer driver scheduling event was added.

Signals will be sent with regular intervals from the timer to the process. Upon receiving the signal, the currently running thread will be suspended and the next the next thread in the run queue will be scheduled. This procedure is exactly what green_yield() does and therefore, the timer_handler() will simply call green_yield().

Unfortunately, there is a problem with the solution above. In the case of a time interrupt in the middle of a function that manipulates the state of the threads (for example a yield operation) there is risk for errors. These interrupts needed to be prevented when changing the state. This is a simple way to block and unblock these interrupts:

```
sigprocmask(SIG_BLOCK, &block, NULL);
:
:
:
sigprocmask(SIG_UNBLOCK, &block, NULL);
```

5 A Mutex Lock

Because of the fact that a thread now can be interrupted at any point during the execution, a problem arises when the shared data structures are updated. For example, when two threads read and increment a shared counter, it will lead to very unpredictable behaviour. Hence, the implementation thus far was extended with mutual exclusion.

Concurrency issues are avoided by simply blocking the interrupts while a process tries to take the lock. This means that only when a thread has the lock and updates shared variables will the potential concurrency problems be solved. A thread that holds the lock will give the lock to the next thread in line. When a thread is woken up after having been suspended on the mutex, the lock now belongs to this thread.

In case of a thread waiting on the lock, the lock is not released but passed over to the suspended thread. This unlock function is similar to the signal operation.

6 The Final Touch

Now, one final implementation remains. As in the pthread library, a function that suspends on a conditional variable and releases a lock in one atomic operation is needed. The lock should be held when the function returns.

Therefore, the green_cond_wait() function is changed to take a mutex argument and make the thread release the lock before suspending and take it back before returning from green_cond_wait().

7 Benchmark

A benchmark program was written in C. In the benchmark program, two threads were created whereas one of them were assigned the role of a *producer* and the other a *consumer*.

During the execution, it was intended for a shared global variable to have its value updated by the two threads. When the consumer thread obtains the lock, it decrements the global variable to 0, sends a signal that the variable has been updated and releases the lock. The producer then obtains the lock, increments the global variable's value to 1 and waits.

In this benchmark program, this procedure was executed 100 000 times before it terminated and was performed both with the green threads library created in this assignment as well as with the built-in POSIX threads library.

As can be seen in Figure 1, the implemented green library performs slower than the POSIX library. Although, one should use pthreads which provides more reliability, which is crucial in order to work with multi-threading.

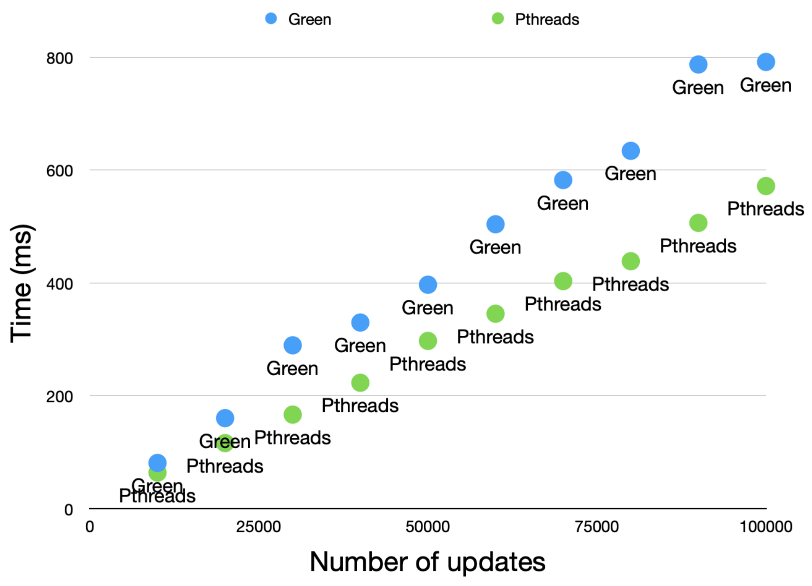

Figure 1: The time it takes for 2 threads to make 100 000 updates.

8 Problems

As often in C, there were a few bumps on the road considering pointers, especially when implementing the ready queue. I myself am a bit new to writing in C which in turn led more time being taken whilst implementing the ready queue and understanding the procedure.

In order to understand when and how to use the mutex lock implementation, the information provided in the course litterature (Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau) was very helpful. The same goes for the usage of conditional variables.