ID1206 Operating Systems Seminar 3

Amiran Dugiev

December 16, 2021

1 Introduction

The purpose of this report is to document the implementation of a thread library using C. The thread simulates the pthread library and uses context handling to handle multithreading. I also implement elements such as mutexes later on. The report is finished off by comparing the execution tests of our created library with that of generic Pthread library functions.

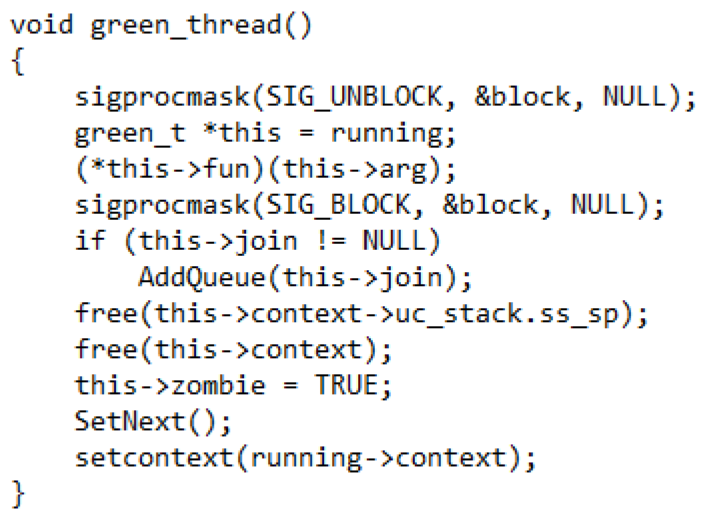
2 Contexts

Before writing our thread library I investigated the structure of the Pthread functions to get a better understanding of how our own implementation should look. We encapsulate the information a thread should contain into a structure that contains a pointer to its function and a marker that designates it as a zombie thread along with other information.

In our implementation the green thread creation function is given an uninitialized green thread structure, a designated function and arguments. It then creates some stack space, sets up pointers and inserts itself into the queue, ready to execute its function when the context switches to it.

Once green threads have been created and put in the queue we can start running the functions they contain with the green thread function which saves the result of the execution and marks the thread as a zombie thread which has already executed its function.

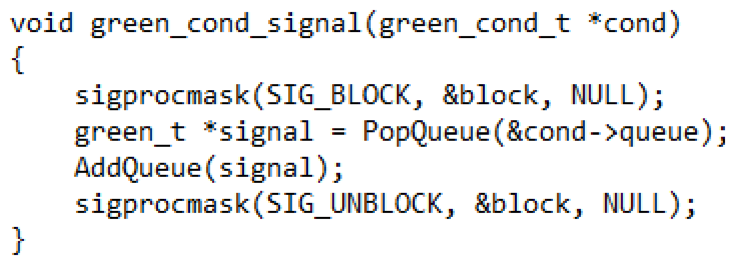
The final two functions we implement are a join and a yield function. Green threads call the yield function to put themselves at the back of the queue and let the next ready thread take over, this function is responsible for context switching. The join function is used to make sure that threads are run in a correct order, if a thread is dependent on another to run the join function is invoked and waits for the other thread to terminate and then frees its space once its a zombie thread.



```
void green_thread()
{
    sigprocmask(SIG_UNBLOCK, &block, NULL);
    green_t *this = running;
    (*this->fun)(this->arg);
    sigprocmask(SIG_BLOCK, &block, NULL);
    if (this->join != NULL)
        AddQueue(this->join);
    free(this->context->uc_stack.ss_sp);
    free(this->context);
    this->zombie = TRUE;
    SetNext();
    setcontext(running->context);
}
```

3 Suspension

So far our implementation changes context switching through the yield function but another way is the use of conditional variables that suspend the currently running thread after some condition is met. This prevents resource heavy threads from hogging all the processing that other threads could use for example.



```
void green_cond_signal(green_cond_t *cond)
{
    sigprocmask(SIG_BLOCK, &block, NULL);
    green_t *signal = PopQueue(&cond->queue);
    AddQueue(signal);
    sigprocmask(SIG_UNBLOCK, &block, NULL);
}
```

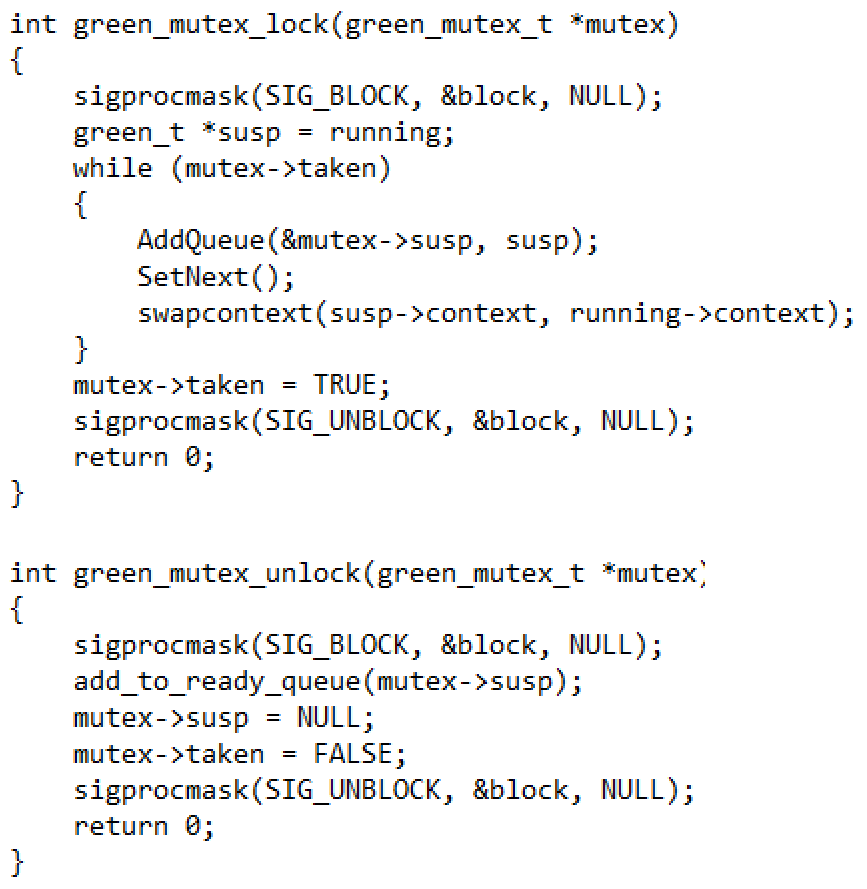
In this signaling function we pop a suspended thread from the queue and add it to the ready queue.

4 Interrupts

A potential problem with our previous methods of context switching is that they rely on the thread itself to determine when it should be swapped out which is reliant on the functions own internal implementation. We can remedy this by adding a timer and stopping the thread whether it has finished its task or not. We implement a timer handler that lets each thread run for a predetermined amount of time and then forces a yield. We also stop segmentation faults by preventing interrupts during the yield operation.

5 Mutex

While using a timer handler solves some problems it also introduces unstable behaviours such as two concurrent threads potentially manipulating each other's resources leading to errors. We implement locking in order to solve this problem.



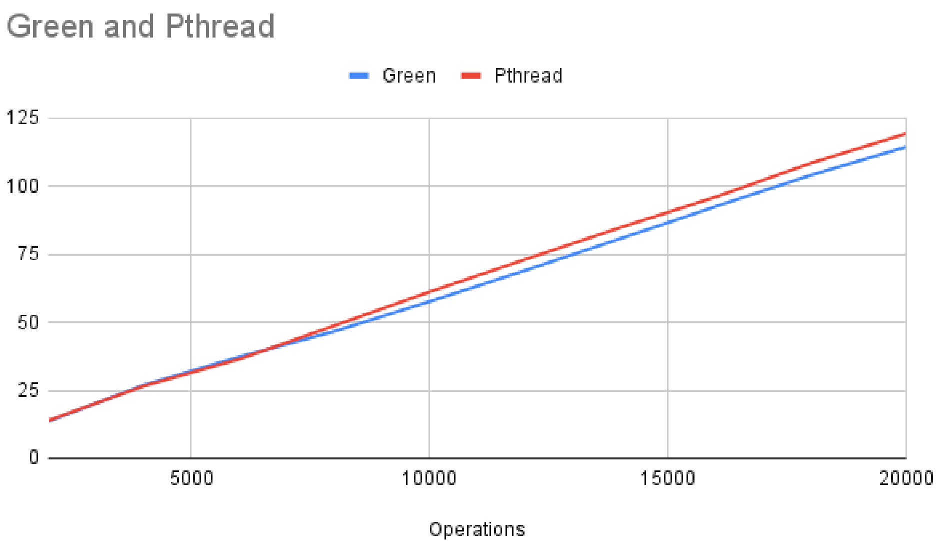
```
int green_mutex_lock(green_mutex_t *mutex)
{
    sigprocmask(SIG_BLOCK, &block, NULL);
    green_t *susp = running;
    while (mutex->taken)
    {
        AddQueue(&mutex->susp, susp);
        SetNext();
        swapcontext(susp->context, running->context);
    }
    mutex->taken = TRUE;
    sigprocmask(SIG_UNBLOCK, &block, NULL);
    return 0;
}

int green_mutex_unlock(green_mutex_t *mutex)
{
    sigprocmask(SIG_BLOCK, &block, NULL);
    add_to_ready_queue(mutex->susp);
    mutex->susp = NULL;
    mutex->taken = FALSE;
    sigprocmask(SIG_UNBLOCK, &block, NULL);
    return 0;
}
```

In this implementation of mutex we force the thread to yield if the lock is already taken by another thread. We also force threads to relinquish the lock once their execution time is up and let another thread take it, as well as preventing any interrupts during this exchange.

There is a potential error that might happen which is that the thread might be interrupted by a timer before calling the wait function leading to a deadlock, we fix this by making the wait function an atomic function that incorporates conditional lock releasing.

6 Benchmarking

We conclude this report by comparing our thread library to that of the standard Pthread implementation. We test the efficiency of these libraries by setting up a producer and a consumer function that both change a global variable to different values using the threads, the testing function was also altered to use mutex and conditional variables with an atomic wait function. We repeat this 20000 times on a Linux Mint system using both libraries. As we can see our implementations efficiency is comparable to the standard.


Green and Pthread
Green
Pthread
Operations