Seminar III Report
ID1206 Operating Systems

Azer Hojlas

January 6, 2022

1 Introduction

1.1 Disclaimer

My code is partially based on my classmate Adeel Hussains code. The benchmark and queue in particular.

1.2 General description

The purpose of the assignment is to illustrate the inner workings of the pthread library. This is achieved by implementing a simplified version of said library without actually using it, which will give the student an enhanced understanding of it's machinations. As the real thread library isn't allowed to be used, the implementations will have a "green" prefix. Threads can be viewed as separate processes, but with shared space and data. The following description of the pthread library is offered by the system manual:

```
POSIX threads are a set of functions that support applications with
requirements for multiple flows of control, called threads, within a
process.  Multithreading is used to improve the performance of a program.
```

The main permitted library calls include but are not limited to: getcontext(), makecontext(), setcontext() and swapcontext(). The following description of the context calls are offered by the system manual:

```
The getcontext() function saves the current thread's execution context in the
structure pointed to by ucp. This saved context may then later be
restored by calling setcontext(). The setcontext() function makes a previously saved
current thread context. The makecontext() function modifies the user thread context
The swapcontext() function saves the current thread context in *oucp and
makes *ucp the currently active context.
```

2 Implementation of the list handling system

I had difficulty implementing this on my own, but a classmate introduced me to the concept of double pointers (embarrassing, I know) which ensured that the management of threads could be handled flawlessly. Below follows the implementation of how the list management works:

```
void enqueue(green_t **list, green_t *thread) {
     if (*list == NULL) {
         *list = thread;
     } else {
         green_t *susp = *list;
         while (susp->next != NULL)  susp = susp->next;
         susp->next = thread;
     }
}
green_t *dequeue(green_t **list) {
     if (*list == NULL) {
         return NULL;
     } else {
         green_t *thread = *list;
         *list = (*list)->next;
         thread->next = NULL;
         return thread;
     }
}
```

3 Implementation of additional functionality

3.1 Conditional variables

A conditional variable in operating system programming is a special kind of variable that is used to determine if a certain condition has been met or not. It is used to communicate between threads when certain conditions become true.

A conditional variable is like a queue. A thread stops its execution and enters the queue if the specified condition is not met (i.e we suspend it and put it on the queue). Once another thread makes that condition true, it sends a signal to the leading thread in the queue to continue its execution.

The following functions are necessary to implement suspending on a condition:

`green_cond_init(green_cond_t*): initialize a green condition variable`

`green_cond_wait(green_cond_t*): suspend the current thread on the condition`

`green_cond_signal(green_cond_t*): move the first suspended thread to the ready queue`

3.2 Adding a timer interrupt

In order to prevent starvation, it is desired to enforce that all threads share resources as equally as possible. With the addition of timer interrupts, we can ensure that each thread gets to run a set amount of time by introducing a time driven scheduler. Before this introduction, threads could keep executing without ever calling on the yield function (inducing lifelock or deadlock?). The time driven schedule eliminates this problem by calling the yield function after a set period of time. If a timer interrupt is received during a operation that affects the thread state it could lead to unforeseen errors. This is handled by implementing a function sigprocmask() that to me is reminiscent of a semaphore, thus ensuring that any operation between the function start and end call is "safe" or atomic.

3.3 Mutex locks

Due to the fact that threads can be interrupted because of the latest implementation, shared data structures can consequently get unpredictably updated. In order to resolve this, we introduce the concept of mutex locks. The mutex structure stores suspended threads once a lock has been reserved so no more than one thread at a time tries to reserve said lock. The following functions and structure represent the mutex addition.

```
int green_mutex_init ( green_mutex_t *mutex ) ;
int green_mutex_lock ( green_mutex_t *mutex ) ;
int green_mutex_unlock ( green_mutex_t *mutex ) ;
typedef struct green_mutex_t { volatile int taken;
    struct green_t *list;
}green_mutex_t;
```

The general idea behind this is that mutual exclusion locks can "lock" certain operations or code, thus making said operation exclusive for a thread that has the lock.

4 Implementation of benchmark

The benchmark compares the performance (in terms of execution time) between our thread implementation and the standard library pthread POSIX

implementation. This is done by initiating said threads and letting them take turns in updating a global variable. This will be done in 10 iterations and each successive iteration will increment the number of updates by 10000. The runtime of each thread will be saved and displayed in a chart below. Whilst updating the global variable, one thread will hold the lock while the other one waits (in order to not get interrupted). After one thread is done updating, it will release the lock and send a signal reanimating the suspended thread. This will go on indefinitely until the test is complete.

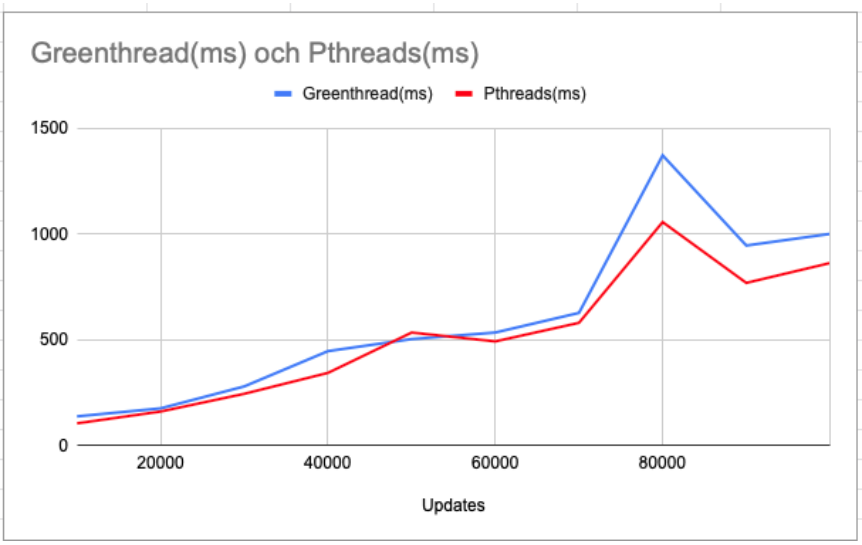

Figure 1: Regular HEAD vs smaller HEAD with 150 runs

5 Conclusion

With regards to presented material, it is safe to assume that the standard library threads are faster. Initially the differences in runtime are miniscule, but as the size of input increases, so does the difference in execution time. This difference in time complexity, judging by the curves, appears to be by a constant k. Even if my implementation of pthreads did not perform as good as the original, it did a fine job of mimicking the POSIX one. Out of all the seminars, this one was the toughest for me to grasp. Without assistance from my fellow classmates this would have been impossible to understand and implement.