Green

13 december 2020

Philip Salqvist

phisal@kth.se
CINTE
1992-04-17

Innehåll

1 Problem description

The problem of this assignment was to create one's own pthread library in C, to gain a deeper understanding of how a library like this works under the hood. During the assignment, we were faced with a number of challenges. One of the main challenges concerned how to handle scheduling of threads. Furthermore, the library needs to enable the threads to suspend on a queue. Finally, we need to consider the fact that unpredictable behaviour can occur in a program when several threads are operating on a global variable, which is addressed using a lock implementation.

2 Program description

Initially, the scheduling of threads was managed using the yield operation. This procedure simply places the thread at the end of the ready queue, finds the next thread to be executed, updates the global running variable and uses *swapcontext* to run the new thread.

The scheduler should be able to place a thread in three different states: running, ready or suspended. To achieve this, we enable the thread to be suspended on a condition and let the condition structure represent the suspended queue. Two main procedures were created to manage the condition queue, *green_cond_wait*, which suspends the thread on the condition, and *green_cond_signal*, which signals to the first thread in the ready queue that it is time to wake up, by moving it from the conditional queue to the ready queue.

In addition to this, we introduce a timer that on a given interval signals to the running thread to suspend and to schedule the first thread in the run queue. At this point, a running thread can be interrupted at any point during execution. This will be problematic when several threads are manipulating a global variable and will cause unwanted unpredictable behavior in our program. To handle this issue we implement a mutex lock structure. First of all, we create the *lock* and *unlock* procedures, which enables the threads to take the lock and to pass it along. Only one thread at a time can be the holder of the lock. We therefore need to enable threads to suspend on the mutex whenever they want to take the look, but it is already taken. To achieve this, we let the mutex structure hold a list that threads can suspend on while waiting on the lock.

Now, the current state of the implementation is problematic. Segmentation faults occasionally occur when running the following test:

```c
while (loop > 0) {
    green_mutex_lock(&mutex);
    while (flag != id) {
        green_mutex_unlock(&mutex );
        green_cond_wait(&cond);
        green_mutex_lock(&mutex);
    }
    flag = (id+1) % 2;
    green_cond_signal(&cond);
    green_mutex_unlock(&mutex);
    loop--;
}
```

Consider the edge case when a thread T_1 enters the first loop, it's id matches the flag, so it enters the second loop and right after calling the unlock procedure, the thread is suspended by a timer interupt. Now T_2 enters, it takes the lock, updates the flag and signals that T_1 should be placed in the ready queue. But since T_1 is not yet suspended, it won't wake up. T_2 will enter the second loop after taking the lock, where it will suspend on the condition. We now have a deadlock.

This edge case is solved by letting the unlock, wait and lock procedures occur in one single atomic operation. We pass the mutex to *green_cond_wait*, and if the mutex isn't NULL, we perform all of the operations in this single procedure:

```c
while (loop > 0) {
    green_mutex_lock(&mutex);
    while (flag != id) {
        green_cond_wait(&cond, &mutex);
    }
    flag = (id+1) % 2;
    green_cond_signal(&cond);
    green_mutex_unlock(&mutex);
    loop--;
}
```

3 Benchmarks

The examine the performance of our *green_thread* library compared to the standard *pthread* library, we need appropriate benchmarks. The key objective to measure is going to be execution time. When constructing the benchmarks, we want to take notice of the fact that the *pthread* library probably will perform better in a benchmark that where parallelism can be used. On the other hand, it is possible that the *green_thread* library is going to execute faster when there is a lot of context switching. In both benchmarks, we create two threads T_1 and T_2 and have both of the threads run the procedure *test* that runs for one million iterations. Both of the benchmarks are exemplified using the *green_thread* library, although the identical procedures were created using the *pthread* library as well.

3.1 Benchmark 1

In benchmark 1, our goal is to create a test that enable the *pthread* library to utilize parallelism. We simply increment a global variable *count* and protect the increment operation using the lock and unlock procedures:

```
void *test(void *arg) {
  int loop = 1000000;
  while(loop > 0) {
    green_mutex_lock(&mutex);
    count++;
    green_mutex_unlock(&mutex);
    loop--;
  }
}
```

3.2 Benchmark 2

In benchmark 2, we want to prevent the *pthread* library from being able to use parallelism. We achieve this by giving the threads unique id's. In the test, we have the running thread check if a variable flag is equal to it's id. If it is, the thread T_1 suspends on the condition. The other thread, T_2, will update the flag, place T_1 on the ready queue. In the next iteration, T_2 will find that it isn't its turn and suspend on the condition. This way, the threads aren't able to execute parallel to one another. The fact that we make calls to *green_cond_wait* in this benchmark would also mean that we increase the number of context switches being made in relation to the first benchmark. We can hypothesize that the *green_thread* library would be faster to execute in this situation:

```
void *test(void *arg) {
```

```
    int id = *(int*)arg;
    int loop = 1000000;
    while(loop > 0) {
      green_mutex_lock(&mutex);
      while(flag != id) {
        green_cond_wait(&cond, &mutex);
      }
      flag = (id + 1) % 2;
      green_cond_signal(&cond);
      green_mutex_unlock(&mutex);
      loop--;
    }
}
```

4 Result

The executions was performed on an Ubuntu system running on a virtual machine. The benchmarks are in part suppose to measure if there is a difference in execution time between the libraries when parallelism can be used. Therefore the virtual machine is set to run on two cores, otherwise the difference couldn't be measured. The results from the two benchmarks is visualized in Table 1. When using benchmark 1, we can see that the *pthread* library is faster by approximately a factor of 70. The difference between the libraries wasn't as substantial when using benchmark 2. Although we can see that the *green_thread* library performs faster by almost a factor of 2,3.

Benchmark	Library	Execution time (ms)
$Benchmark1$	*green_thread*	$t\_green_1 = 5251ms$
$Benchmark1$	*pthread*	$t\_p_1 = 75ms$
		$t\_green_1 \div t\_p_1 = 70,01$
$Benchmark2$	*green_thread*	$t\_green_2 = 11056ms$
$Benchmark2$	*pthread*	$t\_p_2 = 24893ms$
		$t\_p_2 \div t\_green_2 = 2,25$

Tabell 1: Execution time results comparing benchmarks and thread libraries.