Green Threads

Calin Capitanu
<capitanu@kth.se>
KTH Royal Institute of Technology

December 4, 2020

Introduction

This report is going to evaluate and detail the process of writing a standard library from scratch that does library calls for **getcontext()**, **makecontext()**, **setcontext()** and **swapcontext()**. This library that I have written is a smaller and less competent copy of the **pthread library**. During this research I have examined performance and issues with creating multi-threaded applications and having a main handler for all of the threads.

Implementation

First implementation

The first implementation of my library only had 3 functions to call: **green_create()**, **green_join()** and **green_yield()**. While working on these function, debugging and understanding their true purpose was one of the hardest problems that I found. The fact that a second thread starts executing as soon as the first one is "joined" was one part that I couldn't conceptualize, thus spent a long time trying to figure out a way of making it start only when **green_join()** was called. But here was the second problem, I wanted my second thread to run in turns with the first one, that would mean that the first one would need to return to the main thread just to let the main thread also join the second thread, thus making it an almost impossible task (unless hardcoding, which I obviously didn't want to adhere to). Again, as before mentioned, the debugging process was a little

harsh, thus I resided on the key ability of every engineer: abstraction! Due to abstractions with helper functions, I has able to pin-point problems in a much easier fashion: after knowing that functions like **enqueue()** or **find_next()** worked as expected, there was just a matter of time until I figured the next parts and how to assemble them into the main functions that I needed to build.

```
void enqueue(green_t *thread){
  if(ready == NULL){
    ready = thread;
  } else {
    green_t *n = ready;
```

```
    while(n->next != NULL)
      n = n->next;
    n->next = thread;
  }
}

green_t *find_next(green_t *thread){
  green_t *n = ready;
  green_t *temp;
  ready = n->next;
  n->next = NULL;
  return n;
}
```

End of the list

I feel like I need to write a separate paragraph on this part: the end of the list pointer. I usually don't forget to end a linked list, but that is not an excuse: this was for me one of the biggest issue in this whole assignment, as part of the debugging process, I forgot to take out the next pointer at the end of the list after I enqueue something. What can I say? It was an intense hour and a half to figure out why my program would never end. PS: I'd recommend anyone to just take the pointers towards a list off of an object as soon as it is taken out of the list, thus you can save lots of time!

One or two lists?

One thing that I had to dig deep into was the number of lists that I needed to create. However in the assignment paper it is quite clearly specified that the main thread that is handling the other threads only needs to keep track of the ready list and the currently running process, in my mind, it was pretty clear that I needed to create a list of running processes as well, which was a concept hard to get out and get back on track with how to actually implement this.

Benchmarks

This primitive implementation of thread management turned out to work for simple examples, but it was soon dethroned by big numbers. As there is no mutex saving threads to run over eachother and fight on some common data structure, you would expect things can gen horribly wrong when dealing with big number. You might be right, only if that would be possible, because when the ready list and the data in each thread grows - not even that much to be honest - , you would get a nice **Segmentation fault**. Not that nice, but let us move to the first improvement: conditional variables.

Conditional Variables

This section will deal with conditional variables and how I implemented them. The concept is not hard, and less is the implementantion: there is a data structure that can hold a list of suspended threads in the form of a linked list and there are two functions that can either suspend a thread or take it out of the suspended list. There is nothing more to it, just figuring out how to not let the ready list die empty because all the threads are in the suspended list, but that is also part of the user's part not to suspend tasks without taking them out of the list (implemented here in the tests, it does not end well...or at all)
With this in place, threads can now actually take controlled turns on who and when to run.

```
void green_cond_signal(green_cond_t *cond){
  green_t *temp = cond->list;
  if(temp != NULL){
    green_t *next = cond->list;
    cond->list = next->next;
    temp->next = NULL;
    enqueue(temp);
  }
}
```

Timer Interrupt

Timing the amount of time each thread is allowed to take in running is crucial for the OS, as otherwise everyone would implement their applications to never yield, and keep in the running state forever. With a simple clock that would interrupt a running thread in favor of a waiting thread, you could be more equal with how your threads work. There were no big issues in implementing this, it was a bit harded to test and it actually revealed some older bugs in my code.

```
  while(loop > 0){
    printf("thread %d: %d\n", id, loop);
    loop--;
    flag = (id + 1) % 2;
    if(id == 1){
      for(int i = 0; i < 8239000; i++);
    }
    else {
      for(int i = 0; i < 823900; i++);
    }
  }
  printf("finished thread: %d\n", id);
}
```

Even though this might look weird with the useless for loops in there, I

prefered to have a controled delay over the **sleep()** function that environment provides, as there might be some optimizations and whatnot with that. Finding the right type of test was hard, and even though this might not be very good, it shows that the timer functionality works, you can have one thread take longer and instead of them printing the output one at the time, you could see one finishing 3 out of the 4 prints before the other one actually prints the first statement.

```
gcc -c green.c && gcc green.o test3.c -o test3 && ./test3
thread 0: 4
thread 0: 3
thread 0: 2
thread 0: 1
thread 1: 4
finished thread: 0
thread 1: 3
thread 1: 2
thread 1: 1
finished thread: 1
done
```

Mutex Lock

Coming back to the initial test that we have, two threads operating on the same data structure at the same time, here is the fix: the mutex. The mutex acts as a semaphore in a way, or maybe a better way of describing it with a funny and silly example is the public toilets: they have a small red strip on the door when someone is in, and that turns green when it is empty (or unlocked). This is the same thing with the mutex, but in code. As soon as I understood this concept, which is nevertheless easy to understand, coding it was a piece of cake.
Now that we have that in place, we can ask a common counter of the two threads to actually run large numbers without even dropping a segmentation fault on us. Maybe not always, but still better than before! And when we start increasing the number, we don't see any weird behaviour such as lower than expected numbers, that's good!

Conclusion

The one statement that could summarize this whole assignment is "hard to debug". I have to point out one thing that actually helped me a lot: drawing things on the paper and starting to write how the list changes in time. Contrary to anyones belief, actually writing things down is sometimes better then just printing things in the console, at least for me. It was fun and challenging to figure out how to implement a small library that resembles pthread library. Nevertheless, I had a lot to learn by doing this!