Green Threads

Jacob Hedén Malm

January 2021

1 Create, join, yield

The first couple methods seemed fairly self-explanatory to me, once I had figured out what they were meant to accomplish, and once I had read up on the ucontext documentation. Thus, implementing them was rather painless. I chose to represent the ready queue using a singly-linked list, keeping a global green_t variable for the head of the list, similar to how we already kept track of the currently running context.

When I attempted to compile my code, I got some errors related to the ucontext libraries. This was due to me doing my development on a mac. A bit of googling/stackoverflowing suggested that I could get around this issue by defining the macro _XOPEN_SOURCE somewhere in my program. This got the program to compile. However, my first execution lead to a segmentation fault. Upon some headscratching and debugging, I realized I had missed detaching the node that got dequeued from the ready queue, meaning that when this node later got added back, it would be a circular linked list in the case of 2 nodes. This lead to an infinite loop as I iterated across the entirety of the linked list when looking for the node to dequeue, and took the first node where node-¿next was NULL.

After I remedied this issue, I realized I still had some kind of illegal memory access problem. However, to my frustration, no amount of stepping through the program execution with the debugger brought me any clarity. I was able to pinpoint the problem to the second swapcontext() call, in yield. At first I was convinced there was a problem with my program due to the fact that swapcontext was successfully called once, and all of the other context manipulation methods seemed to be functioning fine. However, after much debugging and repeatedly checking the program state right before the offending swapcontext() call was made, and verifying that it looked right, I decided to try my program on linux.

I downloaded virtualbox and a manjaro-iso, and cloned my project. Immediately it compiled and ran as expected.

2 Condition variables

For this task we were given an API to implement. The API was very similar to the Pthreads condition variables, however, these condition variables did not have to be associated with a lock. My guess was that this was due to the fact that our program was not truly multithreaded, at any given time we only had one thread running, thus even without locks our program was still deterministic.

I realized that the condition variable needed to consist of a suspended queue, that needed to support enqueue, dequeue, and contains operations. Seeing as I had already implemented these methods, I did not see it fit to re-implement them. Instead I modified the methods to accept a head node so the same methods could be used for different lists. In fact, I had to pass in a pointer to a pointer to a head node, as I needed to be able to reassign the pointer to the head node inside the method. This caused some issues for me at first, due to my inexperience programming with C and pointers.

My cond_init method looked fairly simple. I dynamically allocated memory on the heap the size of a pointer to a green_t struct.

The wait method was a little more complicated. Here I started by adding the currently running context/thread to the condition variables suspended queue, then I added a while loop where green_yield() was called repeatedly. The condition we checked against in this while loop was that the condition variables ready queue contained this context. Essentially this translates to while we are suspended, yield execution.

In the cond_signal() implementation, I called dequeue with the condition variables list structure as argument once, and queued the resulting node on the ready queue.

3 Timer interrupt

The time interrupt code was provided to us. However, when I ran the first small test with only four iterations, I was surprised to see that the order in which threads printed still alternated perfectly. I expected the timer to introduce non-determinism in the order test loops were completed. For example, if one thread prints out its id and the loop iteration, calls yield, and then immediately the timer interrupt goes off, this would cause the thread which just printed to run again and print out twice in a row. This was not seen. I decided to increase the loop iterations ran to 100,000. However, I was not able to spot a double print out. The program did cause a segmentation fault, presumably because we yield when we are already in the process of yielding.

I added a timer interrupt block during the yield operation. I reran the same test that had triggered a segmentation fault without the timer blocks, and found that the segmentation fault was gone. I did however notice that when I tried to do too many prints/writes within the timer_handler, an error was thrown. Each time the timer handler ran, I wished to print out the time taken since the last timer handler was ran. This was due to me noticing that the timer handler did not seem to execute with an even period. This did not seem to be possible. At first I thought it was perhaps due to the fact that I used printf() within the timer_handler, which should be fine since we are unlikely to throw a new timer interrupt before we are done dealing with the previous one, but switching to write() did not seem to help at first. However the problem was resolved when i removed the time measuring operations and only printed a line indicating we had reached the timer_handler.

In order to test the race condition, I created a small test program where I created four threads that all spawned on a function called increment_counter. In increment_counter, each thread loops for 100,000,000 iterations, and each iteration a shared global counter that is incremented by one. So, at the end of the program execution, we expect counter to equal 400,000,000.

If we witness race conditions, we expect there to be some non-determinism in the counter value we see. Thus, the greater the impact of these race conditions, the greater the "randomness" in counter value, and the greater the variance of counter value over program executions. We expect race conditions to become more prevalent the more timer interrupts are triggered over the course of the programs execution. In order to investigate this, I will vary the PERIOD macro defining the prevalence of the timer interrupt, and for each value of PERIOD, execute the program three times and record the standard deviation and mean of the counter variable.

PERIOD	Standard Deviation	Mean
1E8	0	400,000,000
1000	23,398,984	233,391,273
1	25,411,305	221,281,138

We can definitely spot the effect of the introduced race conditions, as the counter value strays from what it would be if the threads would not interfere with each other at all. However, shortening the period of the timer does not seem to have a great effect on the variance or the mean of the outcome, after a certain point.

4 Mutex

In order to get around the issue of race conditions, we will implement a mutex lock. The mutex lock consists of a suspended queue, similar to the condition variable, and a boolean flag indicating whether the lock is taken by a thread or not.

We will implement 3 methods, init(), lock(), and unlock(). The init() method allocates memory for the queue, and sets the taken flag to FALSE.

The lock() function checks if the lock is already taken, if it is the calling thread gets suspended, and the next thread is scheduled from the ready queue. If the lock is free, the calling thread sets it to taken and returns.

The unlock function first checks if there are any more threads in the suspended queue. If there are, one of those threads is scheduled on the ready queue, and we return. We do not change the state of the taken flag as the thread we wake up from the suspended queue now ”holds” the lock. If the queue is empty, we set the taken flag to false and return.

The program I used to verify whether or not my mutex works consisted of a method that four threads are spawned into called mutex_test(). In this method there is a loop of 1000 iterations, and each iteration the thread prints its id. I added a lock and unlock before and after this loop. If the mutex doesn't work, we expect to see a mix of id's, if the lock works, we expect to see each thread enter the loop section and print out its presence in order, each thread waiting for its predecessor to complete before beginning.

```
    void *mutex_test(void *arg){
    int id = *(int*) arg;
    int loop = 1000;
    green_mutex_lock(&mutex);
    while (loop > 0){
        loop--;
        printf("thread %d is here\n", id);
    }
    green_mutex_unlock(&mutex);
    return NULL;
}
```

The mutex does indeed work :).

5 The final touch

The code shown does indeed have an issue, if the timer interrupt goes off between the mutex unlock and the condition_wait, if there are two threads, the next thread will signal on an empty condition variable, and increment the flag, after which it will itself wait on the condition variable. When the interrupt goes off again, the next thread will wake up and immediately wait on the condition variable. Thus both threads are waiting for each other, leading to deadlock in the program.

Thus, we rework the condition_wait() method to atomically release the lock. The first thing we do is add the current thread to the condition variables wait queue. Then we release the lock. When we release the lock we add the threads in the lock's wait queue to the ready list, in order to allow a thread that had been waiting for the lock when we were using it a chance to execute. Next, we find the next thread to run from the ready queue, and allow its context to start executing.

When we return from this call, we know that we have been signalled by another process, as the only way for us to return is if we have been put on the ready queue, which only happens on a signal to our condition variable. Thus, we know that we should grab the lock, if we cannot, we join the queue for the lock and allow the next process to operate, otherwise we are good to go.

In order to demonstrate the functioning of my program, I wrote a short producer/consumer model. It consists of a producer() and a consumer() method, a shared resource defined as an int variable that is full when equal to 1 and empty when equal to 0, an empty and a full condition variable, along with one mutex. Each thread is spawned into either producer() or consumer().

The producer and consumers are ran in infinite loops. The first thing each tries to do is grab the mutex lock, this is to ensure that only one process can access the shared resource at a time. The producer then checks that the resource is empty, if this is true the producer fills the resource, and signals on the full condition variable, after this it gives up the lock. If the resource is full, the producer waits on the empty condition variable. The consumer is the mirror image of this. A successful program execution consists of alternating producers and consumers filling and emptying the shared resource, making sure only one thread at a time is in the critical secction.

```
green_cond_t empty;
green_cond_t full;
int resource = 0;
#define EMPTY 0
#define FULL 1

void *producer(void *arg) {
    while (1) {
        green_mutex_lock(&mutex);
        if (resource == EMPTY) {
            //produce
            resource = FULL;
            printf("PRODUCING");
            green_cond_signal(&full);
        }
        else{
            green_cond_wait(&empty, &mutex);
        }
        green_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg){
    while(1){
        green_mutex_lock(&mutex);
        if (resource == FULL){
            //consume
            resource = EMPTY;
            printf("CONSUMING\n");
            green_cond_signal(&empty);
        }
        else{
            green_cond_wait(&full, &mutex);
        }
        green_mutex_unlock(&mutex);
    }
}
```

6 Github

Code can be viewed at github.com/jacobhm98/ID1206-Operating_Systems