Green Threads - ID1206

Youssef Taoudi, yousseft@kth.se

December 13, 2018

Abstract

This report describes how a thread-handling library for context switching could be implemented in an operating system step-by-step. Our library includes condition variables, time interrupts, locks to be used to counter race conditions on shared variables when using context switching.

1 Introduction

The assignment was to implement a scheduler for handling threads concurrently by switching contexts. The library is similar to the pthreads library, but left a lot to be desired and was not very properly tested. The library was of course implemented in user space since the operating system was not be involved. The library was named greenthread.

The implementation was done in steps, starting with implementation of the initialization, creation and declaration of a green thread and the context switching functionalities yield and join for our green threads. The second step was to implement a locking mechanism by using condition variables. The functionalities to implement were initialization of a condition variable, suspending the running thread on a condition and signaling a suspended thread to get ready to execute. The third step was to add a timer interrupt to ensure that threads do not use too much time or even run forever. The previous implementation had to be modified to ensure they worked well with the newly introduced timers. The next step was then to introduce mutex locks to avoid race conditions. Lastly, the mutex locks and condition variable functionalities had to be modified slightly for it to be atomic when unlocking and waiting.

2 Implementation and Testing

2.1 Init and Defining

An interface is defined in a header that includes our green structure and library functions. The green thread holds a context that will handle storing data, the process that the thread will run, the argument for the function, a pointer to another thread to be used in a linked list, another pointer to joining threads and a variable that determines if the thread is a zombie or not.

2.2 Create and Switching

API for creating and yielding threads.

2.2.1 green_create

When creating a thread, a stack is allocated from the OS, a stack pointer is created and a context is created in the green thread function. All attributes are set and the thread is added to a ready queue.

2.2.2 green_thread

The green thread function is where a context will start from after a swap. After the user function is terminated, all joining threads are added to the end of the ready queue, the memory of the context is freed, the thread is set as a zombie and the first thread is dequeued from the ready queue and will start running from it's current context.

2.2.3 green_yield

The yield function simply adds the currently running program to the ready queue and replaces it with the first thread in the ready queue then swaps their contexts.

2.2.4 green_join

The join function passes a thread as a parameter. The currently running thread will put itself into the join queue for the thread that it wants to wait for and will then suspend itself and run the next thread in the queue. The joining thread will be put into the ready queue when the joined thread is done running (in the green thread function).

2.3 Condition Variables

Condition variables are introduced to the library so that threads can signal when a condition has been met. This way other threads that are waiting for the condition know they can get ready to run. This is a form of locking to prevent race conditions for concurrent threads.

2.3.1 green_cond_init

The initialization function sets all attributes for our newly created condition variable structure which holds a pointer to the first element in a list of threads waiting for a condition.

2.3.2 green_cond_wait

The wait functionality adds the currently running thread to the condition list for the specified condition and then schedules a new thread to run.

2.3.3 green_cond_signal

The signal function enqueues the first thread in the condition list to the ready list, and thus a thread that was previously waiting for a condition will now be ready to execute since the condition is signalled to have been met.

2.4 Timer Interrupts

Timer interrupts are introduced into the library because user processes cannot be trusted. At the moment, a thread can run for as long as it wants, it does not even have to make a call to yield or wait. This could mean that a thread can go into an infinite loop and thus never allowing any other threads to run. This will however create some problems in the code, for example, what happens when we have a timer interrupt in a library function? Odds are that there will be an interrupt when handling a queue which will result in inconsistencies in the state of the queue for different threads which inevitably will give a segmentation fault.

This is solved by blocking and unblocking the timer at certain points in the code to prevent interrupts when handling thread and queue states.

2.4.1 where do we block?

The timer interrupts are blocked at the start of every library function and are unblocked at the end of every library function. The tricky part is that this means that each time a new context is set or swapped to, it will have the timers blocked. When a context is switched, it will first enter the green thread function which will select the running thread and run the user function in blocked mode. The solution is thus to unblock at the start of the green_thread function and block as soon as the thread returns from it.

```
void green_thread(){
   sigprocmask(SIG_UNBLOCK,&block,NULL);
   green_t *this = running;
   (*this->fun)(this->arg);

   sigprocmask(SIG_BLOCK,&block,NULL);
   enqueue(this->join);
   free(this->context);
   this->zombie = TRUE;
  green_t *next = dequeue();
   running = next;
   setcontext(next->context);
   sigprocmask(SIG_UNBLOCK,&block,NULL);
}
```

2.4.2 problems

The timer interrupt is tested by running one thread that yields and one thread that will run an infinite loop. When running this test, the threads will switch context constantly, meaning that the timer will interrupt the thread trying to run in an infinite loop.

```
THREAD ONE:
void *test(void *arg){
  int i = *(int*)arg;
  int loop = LOOPS;
  while(loop>0){
    loop--;
    printf("Im running :)");
    green_yield();
  }
}
THREAD TWO:
void *infinity(void *arg){
  while(count<MAX){
    printf("Time me out!\n");
  }
}
```

Now the interrupts can only happen in the user code which seems fine at first but as always, there are times where the implementation does not work as it should.

```
void *test_cond(void *arg){
  int id= *(int*)arg;
  int loop = MAX;
  while(loop>0){
    if(flag==id){
    count++;
    loop--;
    flag = (id+1)%THREADS;
    green_cond_signal(&cond);
  }
  else{
    green_cond_wait(&cond);
  }
  }
}
```

Running the test above where we increment a global counter variable each time a thread loops, the problem becomes apparent very quickly. The result of the counter at the end of the process is non-deterministic (It will vary each runtime), meaning there is a race condition to the counter variable. A counting thread may have been interrupted in the middle of trying to increment for example. The solution to this will be introducing proper locks that provide mutual exclusion.

2.5 Mutex

Mutual exclusion locks (mutex) are introduced to the library. Once again, a new struct is created, this time called green_mutex_t that holds an integer variable called taken that signifies that a lock is taken or not and a pointer to a the first green thread in the list for threads waiting to take the lock. The lock provides three functions to the library: lock, unlock and init.

2.5.1 init

Once again, init initializes all attributes of the mutex structure.

2.5.2 lock

The lock function will check if the lock is already taken, if it is not, it will simply set the taken member to true. If not, the running thread will be put into a queue with threads that are waiting for the lock and the next thread in the ready queue will be executed. The latter process will be done in a loop so that the thread will keep suspending itself whenever the lock is still taken.

2.5.3 unlock

Unlocking is simpler, the taken flag is set as false and all waiting threads are moved to the ready queue which is the reason we will loop in the lock function.

2.6 Final Touch

Since a timer interrupt can happen inbetween unlock and wait that can result in a deadlock, the lock and condition wait functionalities were made atomic meaning they are done in the same library function. Since all library functions block timer interrups, the wait and unlock will be atomic.

Now when a thread wants calls the condition wait function, it will first suspend the running thread into the condition queue, then it will unlock the mutex lock and add all threads that are waiting for the lock to the ready queue. After that, the next queue in line will be executed.

To ensure this worked, a producer/consumer test was made where a consumer will take the lock, decrement a flag, signal to the producer that the flag has been decremented and then wait and release the lock atomically. The producer will then take the lock, increment the flag, signal to the consumer that the flag has been incremented and then unlock the lock and wait atomically. If the implementation is done correctly, this will keep going for eternity.

```
CONSUMER:
void *test_consumer(void *arg){
  int id= *(int*)arg;
    while(1){
      green_mutex_lock(&mutex);
     if(flag%2==CONSUMER){
      flag++;
      if(flag!=1){
      green_cond_signal(&produce);
    }
     }
      printf("consumer sleep\n" );
    green_cond_wait(&consume,&mutex);
    }
}

PRODUCER:
void *test_producer(void *arg){
  int id = *(int*)arg;
    while(1){
      green_mutex_lock(&mutex);
      if(flag%2==PRODUCER){
      flag++;
      if(flag!=0){
      green_cond_signal(&produce);
    }
      }
      green_cond_wait(&consume,&mutex);
    }
}
```

Obviously this test results in a segmentation fault which has not yet been fixed because the bug is impossible to find (trust me I've tried). I have however made sure that the test program itself works as intended by testing it using the standard library pthreads in the example below.

```
void *pt_consumer(void *arg){
  int id= *(int*)arg;
    while(1){
      pthread_mutex_lock(&lock);
     if(flag%2==CONSUMER){
      flag++;

      pthread_cond_signal(&produce);
     }
    pthread_mutex_unlock(&lock);
      pthread_cond_wait(&consume,&lock);
    }
}
void *pt_producer(void *arg){
  int id = *(int*)arg;
    while(1){
      pthread_mutex_lock(&lock);
      if(flag%2==PRODUCER){
```

```
      flag++;
      pthread_cond_signal(&consume);
      }
      pthread_mutex_unlock(&lock);
      pthread_cond_wait(&produce,&lock);
    }
}
```

3 Conclusion

Since we are running a multi-threaded program with timer interrupts, our program will be non-deterministic and very difficult to predict when testing and can lead to very unusual bugs as seen during implementation. The implemented library is not something that should ever be used by anyone ever, please keep using pthreads...

As a side note, I think I've caught a severe case of segmentation fault in my left brain and I profusely apologize for any bad writing in the report.