Scheduler README

Alfonso Buono (ajb53), Ali Mohamad (aam345)

March 2020

1 Introduction

The goal of this project was to implement a thread library and scheduler in C. After working on the assignment we were able to learn why there are limitations to user threads, why certain design choices are made for user threads, what contexts are (and how to use them), and how to implement different scheduling algorithms.

2 API

Our general design intuition was to make it such that our scheduling functions do not actually execute threads or contexts - it simply just maintains our data structures and returns a pointer to the correct thread to execute.

This being said, our rp_thread() functions execute far more functionality than our scheduler functions as the bulk of the work is done in the schedule() function. Here the function receives the next thread (with its context) to run next and swaps to its context. It allows the thread / function that the user passes to run and after the allotted time quantum (we used 5ms) passes a signal goes off. Once this signal goes off the state of the thread is changed to ready (if it is still running) and then the schedule function is invoked again. The recently running thread is then passed into its correct location (dependent on if MLFQ or STCF). Then our scheduling functions (MLFQ OR STCF) are called again to get the next thread to run, the timer starts again, and then context is swapped. This is then repeated until the threads have terminated.
Each of the main API functions were implemented in conjunction with this schedule() function in order to create the user-level thread library.

2.1 rpthread_create()

rpthread_create() creates a Thread Control Block - we use get_context() to create and initialize the given context. We then assign the context a stack (and some other values associated with the stack) and call makecontext in order to pass it the function that the user wishes to create a thread for. We then also add data that is needed for the scheduling / library inside the TCB. This includes the

threadID (which is based off a global counter incrementing every time a thread is created), its status (which we defined as an enum to contain whether the current thread is ready, running, terminated, or blocked), its context's stack, its priority (to implement MLFQ) and the amount of times that it has run (for STCF). Immediately after we update our TCB we add it to our runqueue, and schedule it.
On top of these steps that are run every time that a thread is created, we have a special case in which it is the first time a thread is being created. Here we have to create two more contexts. These contexts correspond to the user's main function as well as a context for the schedule() function. These should only be created once and are vital to the library as the user main function needs to continue running and the schedule() context is needed to context switch between the threads.

2.2 rpthread_yield()

rpthread_yield() simply stops the running timer (to prevent extra quantums from running and interfering with context switches / scheduling), changes its status from running to ready, and swaps to the next scheduled context. This allows a thread to easily voluntarily give up control and let another thread run (if there are any. If there are none then the schedule() function will context switch to the thread that called yield as it is the only thread running).

2.3 rpthread_exit()

rpthread_exit() assigns the return value (the value_ptr param if it is non-null) in to the TCB. This allows the user to have exit status and allows any thread that joins the thread that exists to receive the exit status. It also stops the timer (in order to prevent any mishaps with the schedule() function), switches its status to terminated and simply switches to the schedule() context. Because of the status change the schedule() context then enqueues this thread into our global terminated list that we need whenever we try to join on exited threads.

2.4 rpthread_join()

rpthread_join() first calls our helper, findTCB(), which simply searches through all TCB lists (runqueue, our MLFQ, terminated, and blocked) to find a given thread by thread id to join the current thread to based on its status. It then waits for the current thread to finish executing with a blocking infinite loop. It then sets the provided return value, if non-null, by going into the TCB of the thread that corresponds to the provided ID and setting it equal to the val saved in the TCB.

2.5 rpthread_mutex_init()

Our implementation of rpthread_muetx_init() takes in a mutex pointer. Here we have to malloc the size of a mutex in order to initialize and store the information that is needed to implement the mutex. We then set the pointer that is passed in equal to the malloc'd mutex in order to "give" the user the mutex. In our struct, we hold a tcb* owner that points to the tcb that is currently holding the mutex (if any). It also holds a queue Head and Tail. These are to allow us to easily queue and dequeue tcbs that correspond to threads that are waiting for the mutex to unlock. Finally, we also have an "isLocked" volatile int. This is our flag that changes with test_and_set so threads can understand if they can pass the lock or not.

2.6 rpthread_mutex_lock()

Our rpthread_mutex_lock() implementation is quite simple. In order to safely and correctly implement a mutex we need to rid the program of any possibility of having race conditions and multiple threads accessing the critical section at the same time. In order to prevent this, we use the built-in __sync_lock_test_and_set() function. Here it both returns the previous value and also sets it to a new value all in one step. If instead of using this we used two statements, such as an if statement to check isLocked and a setting statement isLocked = 1, there is a possibility that the timer would go off switching to a different thread and thus potentially having two or more threads in the critical section.
Furthermore, after this statement (if the mutex was already locked) we add the current thread into the list of waiting threads in the mutex. We do this so that we know whenever this mutex is unlocked which thread should be allowed to access it next. Finally we switch the current thread's status to blocked and switch it back to the scheduler. This will then allow the scheduler to add it into the blocked queue so that it will not run (and waste time that can be used for other threads).
If the thread that calls lock is the first one to do so, it gains control of the mutex and the mutex's value "owner" is set equal to the current thread.

2.7 rpthread_mutex_unlock()

In our rpthread_mutex_unlock() function we need to first check to see if the calling thread owns the mutex it is calling on. If it does not, then we print to the user that it does not have access to the mutex and then return without unlocking.
If the calling thread does hold this mutex, then three things need to occur:

1. Find the next thread that should access the mutex (if there are any) and then remove it from the mutex's list
2. Remove the next thread from the blocked queue
3. Change the next thread's status to ready and enqueue it into the runqueue.

In order to do (1) we go into the mutex's list of threads (if any exist) and pop from the tail. This is because we use FIFO for that queue and it is performed quickly as we have access to both ends of the queue inside the mutex.
If there exists another thread that is blocked by this mutex we can easily perform (2) as we have access to its corresponding tcb. Since our TCBs are essentially nodes of a doubly linked list it is easy and quick to remove it from the blocked queue. We just have to set the values of the two encapsulating nodes to each other and this will take the next thread out of the queue.
For (3) we then are easily able to change the status of the new thread to ready and then push it into the running queue (as we have a reference to the next thread to have access to the mutex).

2.8 rpthread_mutex_destroy()

For our mutex_destroy(), we simply free our mutex pointer and return if the current thread is the mutex owner. If its null, we return an invalid input, and if the current thread is not the owner, we print an error and return.

2.9 sched_mlfq()

Our sched_mlfq() uses four global linked list pointers. After dereferencing the priority of a TCB, it is added to the proper queue by using our dequeue() function and providing its priority (priority 0, 1, 2, and 3 correspond to one of the 4 queues).

In order to figure out which thread to pop (run next) we check the queues starting from the highest priority queue. Here we check if there are any elements in the queue and if there are any it returns the thread that is at our tail (basic FIFO). This thread is then returned to the schedule() function which handles context switching and changing of statuses (among other things).

Upon being popped, after the scheduled thread runs its priority is incremented (our schedule for demoting is one time quantum), and if it doesn't complete, it is added back to the appropriate queue.

For promotion, we have a counter that increments after every quantum. After ten quantums, every node in the bottom-priority queue is added into the top priority queue (runqueue) by invoking our promote() function. Furthermore, every thread's priority is changed to 0 (the highest priority).

2.10 sched_stcf()

For sched_stcf(), we added a timeElapsed property to our TCB struct that keeps track of the number of quantums a context has run for. We maintain a global linked list of all scheduled TCBs, and using a loop and a TCB pointer, we iterate through the entire linked list to find the TCB with the lowest elapsed quantums and return it. It is incremented immediately after it is run.

3 Benchmark

Upon running ./external_cal.sh with 2, 5, 10, 15, 25, 50 threads, our results are as follows:

- 2 threads: 13,565 microseconds (STCF), 13,464 microseconds (MLFQ).
- 5 threads: 10,650 microseconds (STCF), 10,619 microseconds (MLFQ).
- 10 threads: 9889 microseconds (STCF), 9755 microseconds (MLFQ).
- 15 threads: 9840 microseconds (STCF), 9743 microseconds (MLFQ).
- 25 threads: 9786 microseconds (STCF), 9838 microseconds (MLFQ).
- 50 threads: 9698 microseconds (STCF), 9787 microseconds (MLFQ).

Upon running ./parallel_cal.sh with 2, 10, 50 threads, our results are as follows:

- 2 threads: 3650 microseconds (STCF), 3666 microseconds (MLFQ).
- 5 threads: 2928 microseconds (STCF), 2976 microseconds (MLFQ).
- 10 threads: 2682 microseconds (STCF), 2684 microseconds (MLFQ).
- 15 threads: 2609 microseconds (STCF), 2618 microseconds (MLFQ).
- 25 threads: 2532 microseconds (STCF), 2586 microseconds (MLFQ).
- 50 threads: 2485 microsecond (STCF), 2500 microseconds (MLFQ).

Upon running ./vector_multiply.sh with 2, 10, 50 threads, our results are as follows:

- 2 threads: 62 microseconds (STCF), 50 microseconds (MLFQ).
- 5 threads: 61 microseconds (STCF), 44 microseconds (MLFQ).
- 10 threads: 39 microseconds (STCF), 38 microseconds (MLFQ).
- 15 threads: 40 microseconds (STCF), 40 microseconds (MLFQ).
- 25 threads: 46 microseconds (STCF), 47 microseconds (MLFQ).
- 50 threads: 68 microseconds (STCF), 76 microseconds (MLFQ).

As we can see, as we add more threads it becomes faster, but asymptotically so - occasionally, the cost of creating multiple threads is more than just running the program, and as a result time suffers. Additionally, all of our results matched the verified result.

As it appears to us, MLFQ is consistently, but very negligibly so, faster than STCF. This could be due to how often promote() runs, reducing the number of context switches which in turn actually allows some threads to complete, versus STCF which is essentially a revolving door.

4 Comparison

When we tested these scripts with the pthread library, the pthread library was consistently, substantially faster (about twice or three times as fast). We believe that this is due to optimizations in runtime (i.e. using heap data structures instead of linear searches, optimizing the number of time quantums to demote / promote) that unfortunately we did not have time to explore.

The above, however did not prove true for ./vector_multiply, where once again, we see the effects of excessive thread usage / management on time efficiency.