Shubham Rustagi – sr1034
Sameet Hegde – srh155

OS Project 2 – User Level Thread Library

1. Detailed logic of how you implemented each API function and scheduler

General Detail:

Thread control blocks (TCB) are maintained in a run queue which is implemented as a doubly linked list. The general structure of the linked list is: Head <-> T0 <-> T1 <-> T2 <-> Tail. The head and tail are meaningless thread control blocks but are used to make enqueueing and dequeuing easier. Also, the first TCB after head (T0) is always denoted as the current running process.

The timer is set to expire every quantum (which in our case is every 5 milliseconds). When it expires, a SIGPROF signal is sent and a handler function catches the signal and calls the scheduler function.

One important implementation detail is that whenever a function within our library is called, we always stop the timer. When a function is completed, we resume the timer from the moment it stopped. Two exceptions to this are when the scheduler and mypthread_create are called. Since the scheduler switches to the context of another TCB, instead of resuming the timer, we simply reset the timer to the original quantum so the next thread can run for the whole quantum. Calling mypthread_create simply starts the timer over again for simplicity.

The resume feature was added so a thread cannot game our library and our scheduler to allow it to keep running and starve the other threads. For some functions, like mypthread_join, it was a little difficult to implement one stop/resume and so all read/write instructions were surrounded with a stop/resume, guaranteeing the atomicity for any read or write. This is all to say that this allows us to have proper synchronization when we switch between thread contexts and data is not corrupted.

mypthread_create:

The first time this function is called, we initialize our library, our run queue and our terminated queue, our initial main parent thread, and we register our SIGPROF handler. Afterwards, we create a thread control block which has a ucontext_t struct in it that maintains the thread stack. We then enqueue the created thread into the run queue as it is now ready to run, and we continue letting the parent thread execute until its new quantum runs out.

mypthread_yield:

We mark the current running process as “ready” and go to the scheduler, which will, among other things, use the Preemptive SJF algorithm to decide which thread to run next

mypthread_exit:
We mark the current running process as "terminated" and keep track of the return value in the TCB struct. We then call the scheduler function, which will, among other things, remove the terminated TCB from the run queue and add it to terminated queue, where it can be joined on later. The scheduler will now pick a new TCB to run from the run queue.

Mypthread_join:
First, we determine the location of the TCB the user is trying to join on. If it doesn't exist, we return -1. However, if it exists in the run queue, we keep yielding the current thread until the thread we want to join on has been marked as "terminated" by calling mypthread_exit. Once that thread has terminated, or it was already terminated to begin with (before join was even called), we can get the return value of the joined thread and we can remove it from our terminated queue.

Since there are multiple read/writes in this method, they are surrounded by a call to stop the timer before the read/write is called and resume the timer at the value that it was stopped at after the read/write is done. This will guarantee that we will stay within the context of the executing thread during any read/writes, thereby making read or write operations "atomic"

mypthread_mutex_init:
Since the mutex is on the stack, we don't need to allocate any dynamic memory within this function. We initialize our mutex struct by initializing our binary semaphore to 1 and initializing the queue of the mutex where blocked TCBs will be placed. The semaphore will allow us to make sure only one thread is in a critical section at any given time and no other thread can access that section until the thread releases the lock

mypthread_mutex_lock:
We do an atomic read/write instruction using test and set (which isn't really necessary since we have already stopped our timer at the beginning of this function call) and if the read value of our semaphore is 0, we mark the current thread as "blocked" and call the scheduler function. The schedule will, among other things, remove the blocked thread from the run queue and place it in the queue of the mutex struct since this thread tried accessing the critical section when it wasn't allowed to. The scheduler will then pick a new thread to run based on our scheduling algorithm.

However, if the read value of our semaphore is 1, we simply resume the timer at the point in time where it was stopped and continue to execute the running thread which is now in the critical section and has the lock. Note that because we used test and set, we wrote a 0 into the memory of our binary semaphore, letting all other threads know that we are using this lock and that it is unavailable for them.

We only do 1 single test and set because since this is a user level thread library, only one thread is running on the CPU at any given point. Therefore, we don't need to be continuously checking if the lock is available throughout our quantum and waste our CPU cycles. Thus, to increase

efficiency, only 1 check is done. If the lock is available, the thread can use it; otherwise, the thread is considered blocked

mypthread_mutex_unlock:
We take all the blocked TCBs that are in the mutex queue, mark them as “ready” and put them back into our run queue. All other threads are now allowed to compete for the lock again. The semaphore value is reset to 1, marking it as available.

This approach of making all threads ready and adding them to the run queue could create problems if a significantly large number of threads were used since all threads would be woken up at the same time and they will all compete for the lock. However, since only one thread is running on the CPU at any given point in our user level thread library, this may not raise too much of an issue.

mypthread_mutex_destroy:
We simply free the memory allocated for our mutex struct. Since head and tail are pointers to “dummy” TCBs, this memory is freed.

schedule:
The scheduler is invoked on timer interrupts and when other library functions call it. The current running process has its time of execution counter incremented to let us know how many quanta it has run for. We then clean up our run queue by checking if the current running thread has terminated or blocked. If it is terminated, we add it to the terminated queue and if it is blocked, we add it to the queue of the respective mutex, removing it from the run queue in both cases.

We then check to see if we have any processes left to run. If the run queue is empty, we free all allocated memory and exit the process. Otherwise, we call our preemptive SJF algorithm, which will go through our run queue and determine which TCB has run for the least amount of time. Once we find that TCB, we mark it as “running”, start our timer, and swap our context to the new running TCB.

3. Benchmark results of your thread library with different configurations of thread numbers.

All tests done on man iLab machine at approximately 9:30 pm 10/21/2020.

```
sr1034@man:~/os/Proj2/benchmarks$ ./external_cal 20
running time: 23361 micro-seconds
sum is: -882643677
verified sum is: -882643677
sr1034@man:~/os/Proj2/benchmarks$ ./parallel_cal 20
running time: 2097 micro-seconds
sum is: 83842816
verified sum is: 83842816
sr1034@man:~/os/Proj2/benchmarks$ ./vector_multiply 20
running time: 5002 micro-seconds
res is: 631560480
verified res is: 631560480
```

Figure 1. All sums match verified for 20 threads

```
sr1034@man:~/os/Proj2/benchmarks$ ./external_cal 50
running time: 23302 micro-seconds
sum is: -1127014619
verified sum is: -1127014619
sr1034@man:~/os/Proj2/benchmarks$ ./parallel_cal 50
running time: 2095 micro-seconds
sum is: 83842816
verified sum is: 83842816
sr1034@man:~/os/Proj2/benchmarks$ ./vector_multiply 50
running time: 5235 micro-seconds
res is: 631560480
verified res is: 631560480
```

Figure 2. All sums match verified for 50 threads

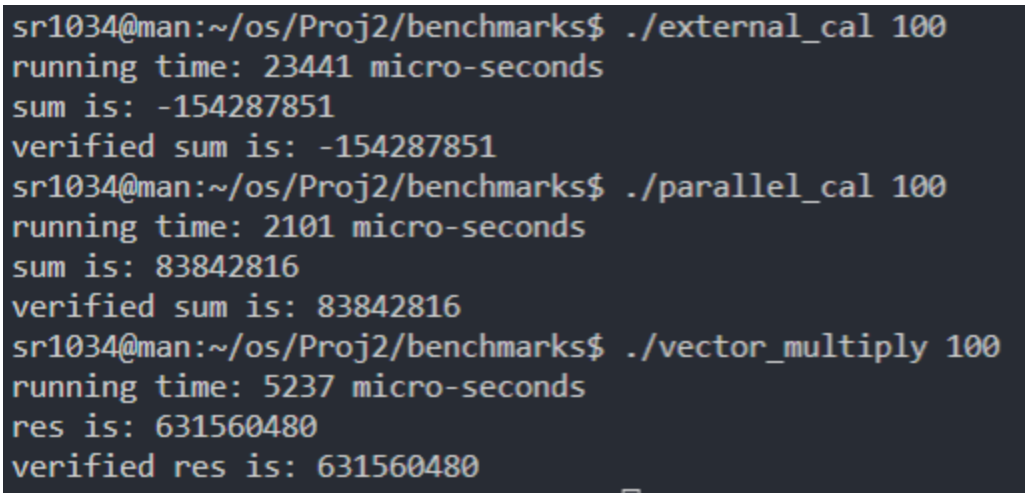

Figure 3. All sums match verified for 100 threads

4. What was the most challenging part of implementing the user-level thread library/scheduler?

The most challenging part was understanding the differences between how our library, a user level thread library, is supposed to work relative to the pthread library, a kernel level thread library. For instance, when we lock the mutex, we simply do 1 test-and-set to check if we can enter the critical section. The pthread library most likely continues to do test-and-set until the lock is available.

Another challenge was thinking of the best data structure to hold all TCB and how we could make as many operations O(1), or as efficient as possible. Since our quantum was 5 milliseconds, operations couldn't be too computationally expensive because that would lead to greater run times in the benchmark tests cases. We realized the best approach was to use linked lists to allow for a flexible number of thread control blocks and ease in enqueuing and dequeuing TCBs.

Another challenge was just integrating all these methods and getting them all to work since many functions overlap and there are a lot of moving parts. To best combat this, we tried to create a lot of modularity within the code and thoroughly tested each method before moving forward. This allowed us to have minimal number of bugs and allowed for a relatively painless debugging session at the end.

5. Is there any part of your implementation where you thought you can do better?

- When a thread yields and calls the scheduler, if it has the minimum number of quanta it has run for in the run queue, it will be scheduled to run again. Because we called the scheduler function, the number of quanta the thread has run for will also increase. There are a couple problems with this.

First, if we have two threads, t0 and t1. If t0 has run for 1 quantum and t1 has run for 100 quanta. If t0 yields, it will take 99 (or possibly 100) calls to schedule for t1 to actually run since the quantum for t0 will keep getting incremented, but since it is still smaller than the other threads quantum, we will keep running t0 and it will keep yielding. This is inefficient, but is necessary because without incrementing the quantum, t1 will never run given our scheduling algorithm. Furthermore, we are "faking" the quanta that t0 has run for by artificially incrementing it. If a thread has only taken a fraction of its quantum, but uses yield, it will be considered as having executed for 1 quantum. This is done for simplicity but could be improved upon.

- We also stop the timer at the beginning of library functions and resume it at the end. We could improve this by encapsulating this stop/resume functionality between any read or write. This would be more effective so one thread doesn't monopolize the CPU by making extra library calls. For instance, mypthread_create currently resets the timer at the end of the function. Theoretically, one main parent thread could keep calling mypthread_create right as its quantum was about to run out and reset its quantum to continue execution.

- We could also improve the algorithm for the preemptive SJF to be O(log n), instead of O(n). We currently go through the run queue to find the TCB to execute next. However, we could use a min heap (prioritized by the quanta of execution) and simply get the next TCB to execute in log n time. This could significantly improve our run times since our scheduler is called many times (due to our 5-millisecond quantum).