0.1 Implementation Logic

At a high level, the main components that we had to to implement in this project were the thread control blocks and mutex structures, a specialized list structure to hold threads, the scheduler, and the actual thread and mutex functions. We approached the project in essentially that order – completing the data structures first followed by the thread and mutex functions and the scheduler. This project implements Preemptive Shortest Job First (PSJF) which is also known as Shortest Time to Completion First (STCF). We had 3 global variables that we declared which were a list of all the thread control blocks called allThreadControlBlocks, an integer called threadIDs that kept track of the total number of threads that had been made and gave each thread a unique ID, and lastly a pointer to the currently running thread's thread control block which was conveiently called currentlyRunningThreadBlock. The following list breaks down all of the remaining function and data structures that interact with those global variables.

1. Structures In Header File

   - **mypthread_status:** This is an enum to hold the various states that a thread can be in. The states are self-explanatory – run is for when a thread can be run, block is for when a thread is waiting/blocked, done is for when a thread has finished execution, and destroy is the state signifying that the thread can be destroyed and its content can be freed.
   - **tcb:** This structure represents the thread control block and it holds the important information that is kept for every thread. It holds the thread's ID, its status, its context, its quantum, the thread that it is waiting on (if any), a pointer to its value (if any), and its return value (if any).
   - **threadControlList:** This data structure is at the heart of the program. This is essentially a priority queue, where each element consists of a pointer to a tcb struct and a pointer to another threadControlList struct. This data structure is an intuitive way to store all of the threads in one place in ascending order based on their quantum (amount of times ran).
   - **mypthread_mutex_t:** This structure is for the mutex and so it holds a value for the lock and a threadControlList pointer that holds all the tcb's that are waiting for this mutex which is called waitList.

2. Methods for the List Data Structure

   - **addThreadToTCB():** This method adds threads to the global variable allThreadControlBlocks, which is a pointer to the start of the list that tracks every single thread control block. This method adds threads in ascending order based on their elapsedTime attribute, which is the counter that keeps track of how many quantum any thread has ran for. In essence, this method ensures that the linked list is

maintained as a priority queue, where the threads with the lowest quantum have higher priority and are therefore at the front of the list.

- **getNextJob():** This method retrives the next ready thread from allThreadControlBlocks. Because we keep track of every thread in one single linked list, this method iterates through the entire allThreadControlBlocks until it finds the first ready thread. Although, we keep track of every thread in one linked list, the thread with the lowest quantum elapsed is found at the head. So the return value will be the first ready thread which will be the ready thread with the lowest quantum.

3. Helper Functions

- **unblockThread():** This function takes a thread ID as an argument and traverses the thread control block list for any thread that was blocked by the threadID in the argument. If there is a match, then the thread gets woken up, else nothing happens.
- **freeNode():** This function takes a pointer to a thread control block and frees all of the memory associated with it, like the thread's stack, the memory allocated for the thread's control block, and finally node itself.
- **createMainThread():** One of the global variables we make use of is threadIDs. The purpose of threadIDs is to assign each new thread a unique ID. However, a special threadID is reserved for the main thread, which we must keep track of and must ensure that it is the first one that's handled. Therefore, whenever a new thread is created, we check to see if the main thread has already been created, if not, then createMainThread is invoked, which is where the main thread is created and assigned as the running thread right away.
- **create_tcb():** This method takes in a thread ID and a boolean called createContext and then returns a tcb pointer. The goal of this function is to have all of the tcb creation done in one location. It malloc's space for the tcb struct and then populates the fields that we know. If the createContext flag is true, then it malloc's space for the context – the space size is SIGSTKSZ. In the end it returns a pointer to the newly made tcb.
- **getTCB():** The argument of this method is a thread ID and the sole purpose of this function is to help us look for a particular thread. In the function we iterate through our list of threads and look for a thread that matches the thread ID that was passed in as argument. If such a thread is found, we return the control block of the thread, else we return NULL to indicate that no thread with the indicated thread ID exists in the list.
- **setupAction():** This method creates the sigaction struct and sets the handler to sched_stcf function. This is important for the scheduler so that everytime the SIGPROF signal is recieved, our scheduler is the handler and does the correct scheduling.

- **setupTimer():** This method creates the timer that is used for the preemptive scheduling. It uses the quantum value when creating the timer.

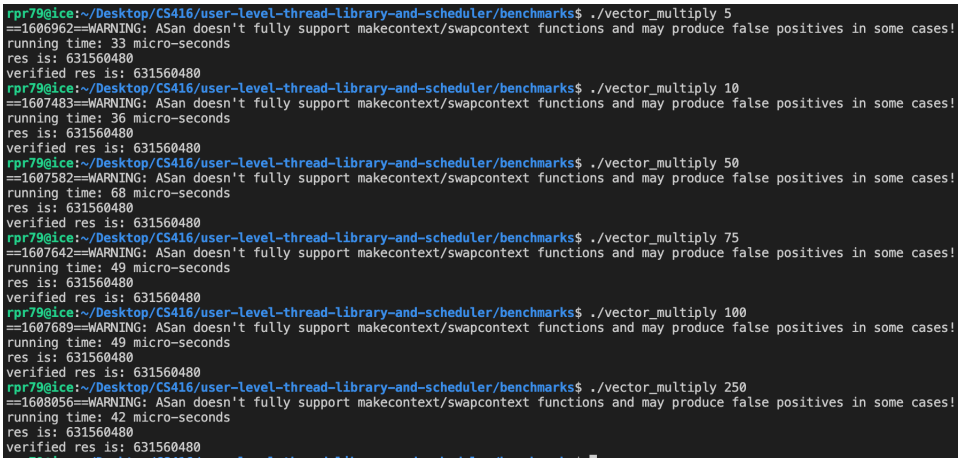
4. Threading Functions

- **mypthread_create():** This function creates a new thread. If there are no threads currently running then this function will first call the createMainThread() helper method to make the main thread. After confirming there is a main thread, it creates a new tcb using create_tcb() and it adds the tcb to the global list. The function and arg parameters passed to this method are used when making the thread context.
- **mypthread_yield():** This function is supposed to give the CPU possession to other threads voluntarily. However, this job is covered by the scheduler so we simply reroute any calls to this method directly to the scheduler.
- **mypthread_exit():** This function should terminate a thread. First we mark the currently running thread's status as done. Then, if it has a valuePtr, we mark its status as destroy. Otherwise we set its return value to be the valuePtr parameter passed to exit(). At the end, we unblock any threads that were waiting on this one using the unblockThread() helper method. Lastly, since possibly many threads have been unblocked we call the scheduler and let it decide what should run next.
- **mypthread_join():** This function checks the thread block list to see if the thread to be joined is already finished. If so, then we set the thread's status to destroy, indicating that we can free its memory and return any values if value_ptr is not NULL. If the thread to be joined has not finished, then we put the current thread to sleep, give the value passed as an argument to the thread to be joined, and then invoke the scheduler to run the next thread. Where we put the current thread to sleep, we record the ID of the thread that was responsible for putting it to sleep, that way when that thread exits, it wakes up the thread it interrupted.

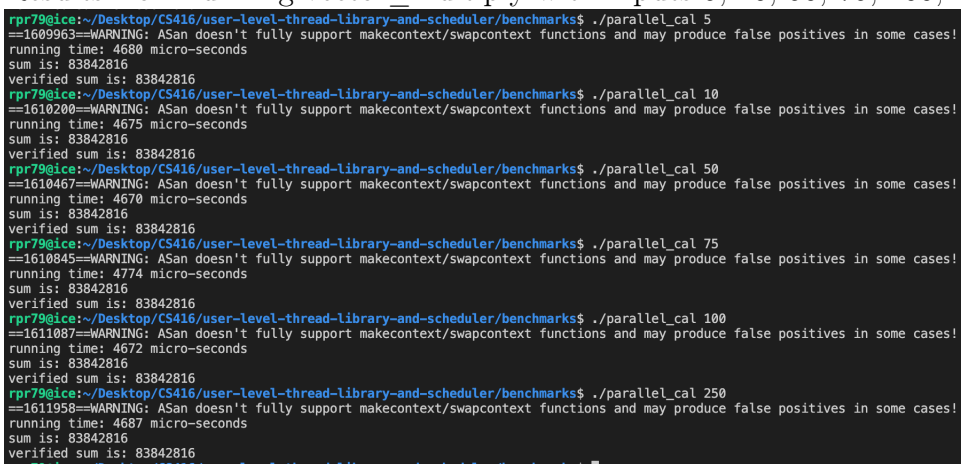
5. Mutex Functions

- **mypthread_mutex_init():** Here we simply set the mutex values to their defaults which means waitList is NULL and lock is 0.
- **mypthread_mutex_lock():** This method should acquire the mutex lock. We use atmoic_flag_test_and_set to check the mutex lock. If this is successful then we enter the critical section. Otherwise we mark the currently running thread as blocked, add it to the mutex's waitlist, and then run the scheduler.
- **mypthread_mutex_unlock():** Here we run through the list of threads that are waiting for the mutex, mark their tcb status as run, and then free the entire waiting list in the mutex, so that the threads can compete for the lock again later.
- **mypthread_mutex_destroy():** Here we simply make a call to the previous unlock(), since it also handles 'destroying' the threads waiting and resets the lock.

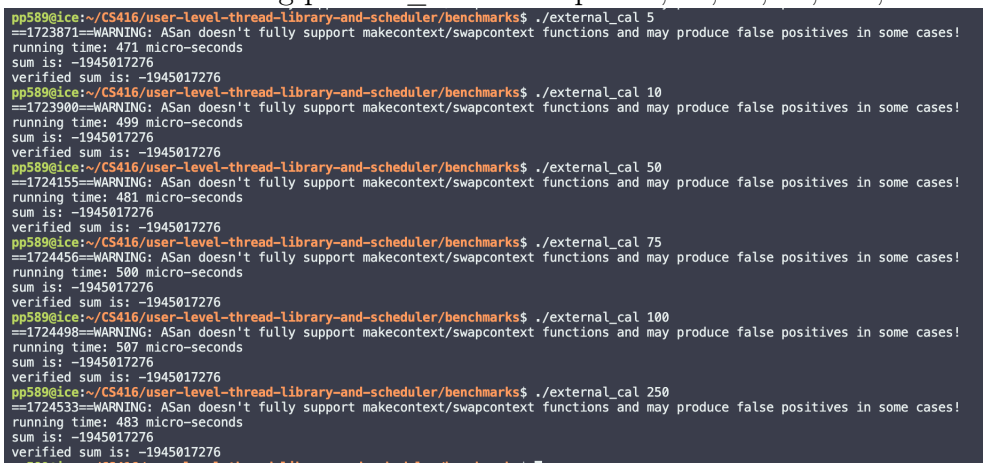
0.2 Benchmark Results



Results from running vector_multiply with inputs 5, 10, 50, 75, 100, 250



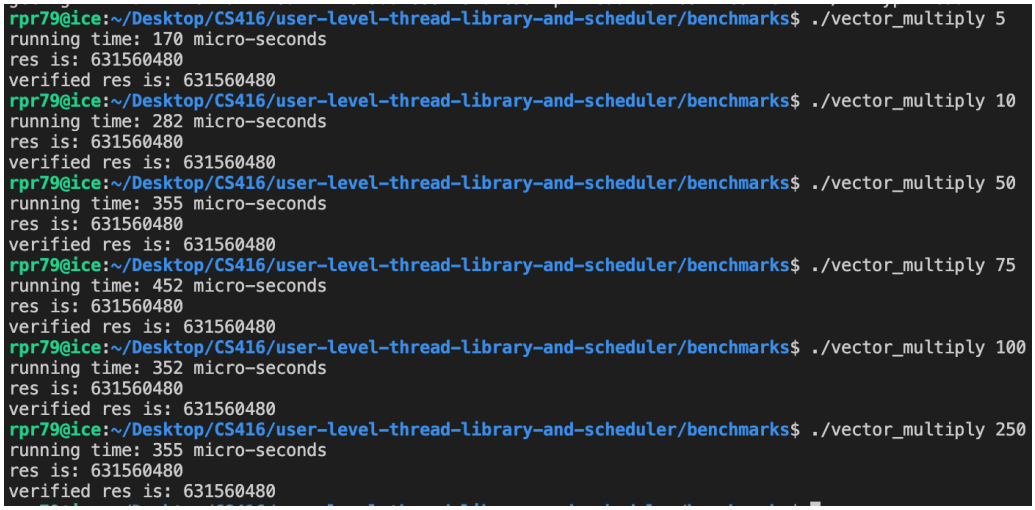
Results from running parallel_cal with inputs 5, 10, 50, 75, 100, 250



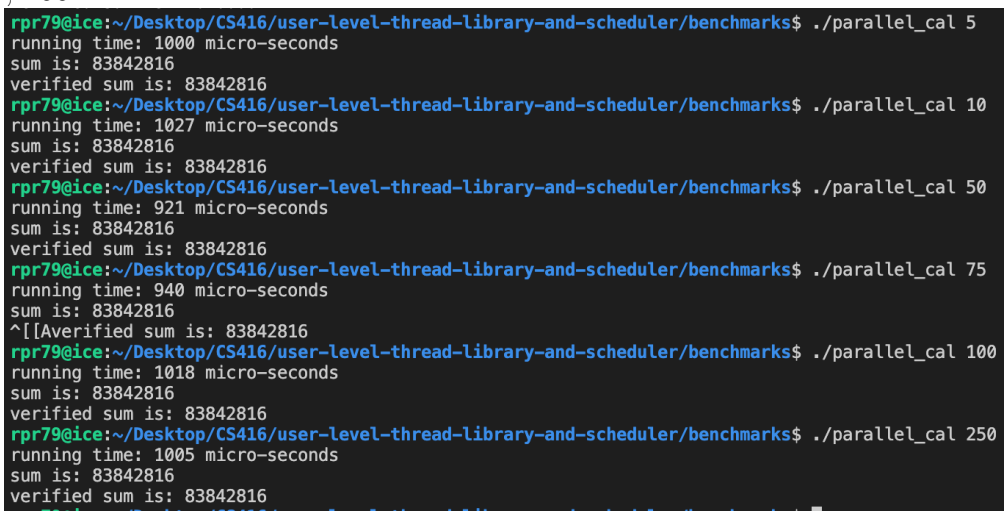
Results from running external_cal with inputs 5, 10, 50, 75, 100, 250

As we can see from the images above, for vector_multiply the runtime became slower when increasing the number of threads when the threads were very small. Once we got more threads though, the runtime started to decrease as we added more. For parallel_cal, the runtimes seem to be averaging around 4670. For external_cal, the runtimes seem to be averaging around 490.

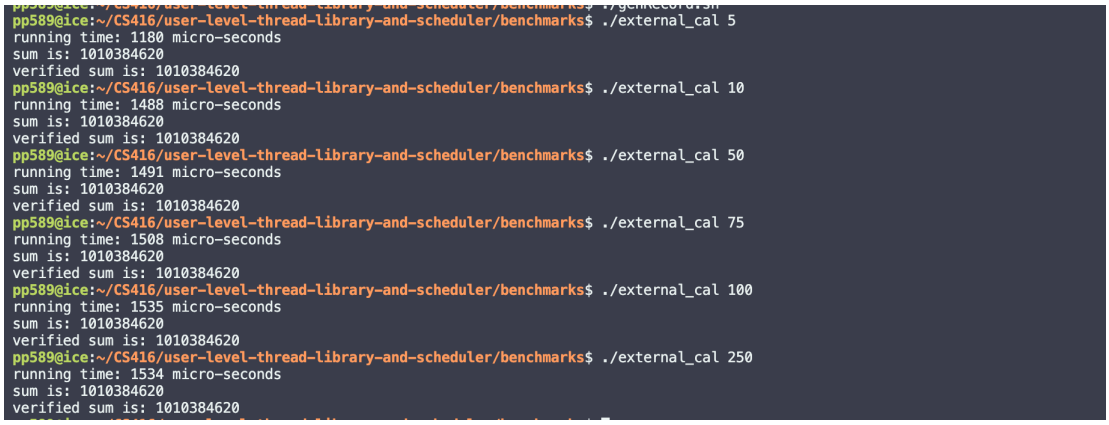
Next, we will compare the above runtimes to the standard pthread library.



Results from running vector_multiply using standard pThreads with inputs 5, 10, 50, 75, 100, 250.



Results from running parallel_cal using standard pThreads with inputs 5, 10, 50, 75, 100, 250.



Results from running external_cal using standard pThreads with inputs 5, 10, 50, 75, 100, 250.

As we can see from the above images, the vector_multiply and and parallel_cal behaved similarly to when they were run with our threads. Specifically, for vector_multiply, the runtime became slower for small number of threads, and faster once we started adding more threads beyond a certain threshold. Also, for parallel_cal, the runtimes average around a 1000. As for external_cal, the pThread version seemed to take longer as more threads were added.

A interesting observation to make here is that our vector_multiply and external_cal runtime seems to be faster than pThread's runtime, but our parallel_cal seems to be significantly slower than pThread's runtime.

0.3 Ending Questions

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

   We both found that the most challenging part of implementing the user-level thread library was the scheduling implementation. While the data structures and the intuition on how these concepts work makes sense, practically implementing them was a lot harder than we anticipated.

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

   We both are fairly certain that there are memory leaks and/or race conditions within the code we wrote. Given more time, we both think that would be the first and most important aspect of the program that needs to be fixed.