Nikoloz Chichua (nc733) | **Course**: CS416-518 | **Machine:** iLab3.cs.rutgers.edu

Project 2 Report:

1. Before I started implementing the thread-worker library, I first defined a couple of data structures for it. Specifically, I designed a doubly linked list that would take, as nodes, TCB structures. I also created a couple of methods for this data structure to manipulate each linked list as I liked. Then I started implementing the API:

   - ***worker_create***, for its first call, initializes all needed data structures for both PSJF and MLFQ scheduling, creates scheduler context, termination context, and main context, and configures the timer, the timer signal handler, and the timer signal blocker (for blocking timer interrupts during scheduling). For the first and all the subsequent calls, this method also initializes the incoming thread and appends it to the run-queue, while also recording the arrival time for the thread.

   - ***worker_yield*** sets the running threads yielded field to true (which is needed in MLFQ scheduling to determine if the thread yielded before the timer interrupt) and immediately context switches to the scheduler context to handle the scheduling of the next thread, therefore yielding the thread.

   - ***worker_exit*** sets the return value of the exiting thread to the incoming argument from the call and context switches to the termination context, which marks the thread as descheduled and context switches to the scheduler context to schedule the next thread.

   - ***worker_join*** finds the thread responsible for blocking the running thread, if it has not yet terminated, and sets the running thread's state to blocked while also assigning to the blocking thread a pointer to the thread it is blocking. Afterwards, *worker_yield* is called and then when the thread is descheduled through the scheduler, the return value of the thread is assigned to the argument of the function.

   - ***worker_mutex_init*** initializes and releases the incoming mutex.

   - ***worker_mutex_lock*** uses the built in test-and-set atomic function to test the mutex, and if the mutex is acquired successfully, enters the critical section.

Otherwise, it pushes the running thread into the blocked queue and calls *worker_yield*, which deals with proper context switching.

- ***worker_mutex_unlock*** uses the built in test-and-set atomic function to test the mutex and if the mutex is acquired successfully, moves its blocked threads to the run queue and makes them available to compete for the mutex later.

- ***worker_mutex_init*** destroys and releases the incoming mutex.

- ***sched_psjf***, first increments the counter for how many times a thread has ran and then decides what to do with the thread that is about to get context switched out. If this thread is marked descheduled, it deschedules the thread (unblocking any threads that it was blocking) and moves it to the descheduled queue. If it is blocked, it gets moved to the blocked queue, and if it is neither than it goes to the back of the ready queue. Then the scheduler finds the next thread to context switch in, which will be the thread that has ran the least amount of times, and then it context switches to the thread, while also recording the time for its first run, if it is being scheduled for the very first time.

- ***sched_mlfq***, first checks the state of the thread that is about to get context switched out. If the thread has been descheduled, it deschedules the thread (unblocking any threads that it was blocking) and moves it to the descheduled queue. If the thread has yielded then it increments the counter that counts how many times a thread has yielded. If the yielded field reaches the yield limit it's priority is reduced. Afterwards, it checks if the thread has been blocked and if it has it gets moved to the blocked queue, otherwise it's state is ready and it goes to the back of the same level of the queue as is its priority. Then the scheduler finds the topmost ready thread and context switches to that thread, while also setting the timer interval to the calculated time quantum (the lower the level is the higher time quantum) for the level that the thread resided in.

2.

Number of Threads	*PSJF*	*MLFQ*
default	**./vector_multiply** run time: 4659 context switches: 314 avg turnaround: 4.508312 avg response: 0.008231 **./external_cal** run time: 16659 context switches: 1110 avg turnaround: 16.094304 avg response: 0.009324 **./parallel_cal** run time: 3551 context switches: 267 avg turnaround: 3.283198 avg response: 0.025802	**./vector_multiply** run time: 3854 context switches: 56 avg turnaround: 4.329067 avg response: 0.009824 **./external_cal** run time: 16294 context switches: 182 avg turnaround: 15.550944 avg response: 0.008540 **./parallel_cal** run time: 3707 context switches: 77 avg turnaround: 3.447521 avg response: 0.027568
10-100	**./vector_multiply 25** run time: 4394 context switches: 342 avg turnaround: 4.303573 avg response: 0.182061 **./external_cal 45** run time: 16827 context switches: 1207 avg turnaround: 3.706270 avg response: 0.124551 **./parallel_cal 95** run time: 3305 context switches: 429 avg turnaround: 2.799542 avg response: 0.709831	**./vector_multiply 25** run time: 4416 context switches: 150 avg turnaround: 3.982602 avg response: 0.383382 **./external_cal 45** run time: 16916 context switches: 457 avg turnaround: 3.965312 avg response: 0.275673 **./parallel_cal 95** run time: 3210 context switches: 287 avg turnaround: 2.563693 avg response: 0.947849

3. Comparing the runtimes of PSJF and MLFQ to POSIX pthread library, we can clearly see that POSIX is much faster, about 5 times as fast as either of the two schedulers. Other measurements can't be done since we can't measure anything else on POSIX.