Donghui Li(dl882) and Jason Dao(jnd88)

Assumptions we made while coding:

- Worker_exit will always be called when each thread exists
- We change main's priority to the lowest level when it yields from worker_join through implementation of a flag because it starves other threads as it constantly yields in worker_join.

1. To implement worker create, we created the thread which the benchmark code created and put it in the queue. In each thread, the status, the priority, the stack pointer, the stack, and the thread ID were saved in a TCB. The time it was enqueued was also saved in the TCB for calculations. On the first time, it goes to the schedule method to call the first thread. Through implementation of flag, we created a main thread on the first time the function is invoked by getting the context and setup a tcb and a threadnode for the main method, which is then inserted into the run queue. We also created a scheduler context on the first invocation of the function, which will take us to the scheduler after the first invocation of the the work_creat function.

   To implement worker yield, we changed the status from running to ready. We got the time that it ran and added it up to see how much time it ran to see if it was ready to increase the priority (rule 5) for only multi-level queue. We saved the time so next time it runs, it uses up the remaining quanta is has left. Then the yield goes back to scheduler to schedule the next thread.

   For worker exit, we saved the thread id and return value pointer in a record struct to save the return value pointer long term until main exits. We got the time the thread finished and recorded it and subtracted that and the start time to get the turnaround time and added it to the global turnaround time. We also recorded the thread's response time to the global variable. We got the time that it ran and added it up in a global variable to compare to S to see if we need a priority reset for multi-level queue. Next, we set the status to completed, and then we freed the stack, and the TCB. After, it goes to the scheduler to schedule the next thread.

   For worker join, first we search through the queues to find the thread, if the status has is still not terminated, we set a flag for the thread that it is waiting for and yield. Once the thread terminates, we find its current location in the queue, and pop it and remove it from the queue and free it. If there is a return value, we pass the pointer from record struct that we saved when the thread exit to the passed in return value pointer and then returns.

   For mutex init, we set the flag to 0.

   For mutex lock, if the lock is 0, the thread acquires it, set the flag to 1 and returns. If the lock is 1 (currently taken), set the thread's status to blocked and yield to go back to the scheduler.

   For mutex unlock, we release the lock by setting the flag back to 0 and wake up the blocked threads that are waiting for the lock.

   For mutex destroy, we set the flag to 0 to release the lock.

For schedule round robin, on the first time, it gets the first thread, and sets the status to scheduled, and then it saved the time it started. It switches to the context of the thread. If the thread finished and main has not called worker join, we free it and set a flag to tell the main thread it has been freed. When we go back to the scheduler after the timer runs out, we set the thread to ready and move it to the back. Then we look in the queue for the next thread that is ready and then we make that at the top. We set the status to scheduled, restart the timer to QUANTUM seconds, and swap to the context of the thread.

For schedule multi-level queue, on the first time, it gets the first thread, and sets the status to scheduled, and then it saved the time it started. It switches to the context of the thread. If the thread finished and main has not called worker join, we free it and set a flag to tell the main thread it has been freed. When we go back to the scheduler after the timer runs out, we set the thread to ready and move it to the queue of what priority it is (if it is main yielding from work_join, we put it in last priority since constant yielding while waiting tends to starve the other threads). If it went through the timer handler, the priority decrease and it is put in a lower queue. Next, we check the globaltime(time run so far) and see if it is greater than the time we set to increase the priority. If it is, we set every thread to high level priority and reset the globaltime. Next, we go through every priority queue to find the next thread with a ready state and schedule that. We set the status to scheduled, start the timer to QUANTUM*QUANTUM2(difference between each level queue time slice) milliseconds or if it still had time left before it yielded, the time left which was saved and swap to the context of the thread.

2. NOTE: We think there is an error in the calculation of the runtime in the benchmark cases but we used that time anyway for our report. It is listed as (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec - start.tv_nsec) / 1000000) microseconds, which should be millisecond instead or use (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec - start.tv_nsec) / 1000) to get microseconds; In our time for average response time/turnaround time we correctly find the answer in microseconds.

   For round robin, our quantum is 10 milliseconds. For round robin, for vector multiple for 10 threads it took 7191 ms (it is listed at microseconds in the benchmark code, but it really should be milliseconds. We will list it as ms in our report), for 50 threads it took 8098 ms, and for 100 threads it took 8970 ms. For round robin, for external calc for 10 threads it took 25162 ms, for 50 threads it took 27634 ms, and for 100 threads it took 30796 ms. For round robin, for parallel cal for 10 threads it took 2594 ms, for 50 threads it took 2596 ms, and for 100 threads it took 2599 ms. For multi-level queue, our S was 100 milliseconds, quantum was 10 milliseconds, and the difference between each level queue time slice was 5 milliseconds (level 1 runs for 10, 2 for 15 etc). For multi-level queue, for vector multiple for 10 threads it took 6861 ms, for 50 threads it took 8008 ms, and for 100 threads it took 8824 ms. For multi-level queue, for external calc for 10 threads it took 24934 ms, for 50 threads it took 27567 ms, and for 100 threads it took 30090 ms. For multi-level queue, for parallel cal for 10 threads it took 2639 ms, for 50 threads it took 2647 ms, and for 100 threads it took 2676 ms. We also tried different S values with 100 threads

in multi-level queue. We also tried S with 1000 milliseconds. For 100 threads, for vector multiply it took 8507 ms, parallel cal took 2602 ms, and external_cal took 30960 ms. From the different s values, it shows that sometimes with longer resetting priority times, it allows turnaround time to be faster, allowing jobs to finish faster.

3. For the pthread library, for 100 threads, it took 327 ms for vector multiply, 1717 ms for external cal, and 614 ms for parallel cal. We think our threads were much slower for external cal (20 to 30x slower) because since we did not yield to i/o. It took up a longer duration and so for our program, it waited a while to receive something instead of scheduling something else. We think our vector multiply was slower (20-30x) because we scheduled one thread at a time to run to ensure the threads went in the right priority and since each thread had to do 3000000 times in an array, it slowed it down. We believe our parallel cal was only slightly slower (3x) because there were 2 arrays doing math at once, making it faster.