Atharva Patil
CS416
anp166
Cheese machine

Project Write Up

API Functions:

worker_create:

- In this function, we essentially take care of a series of tasks. We create the scheduler's context, which is stored in a global variable. We create the main thread's context and assign it to a tcb that is inputted into the run queue. We also create the timer and the appropriate signal handler. We also then create the worker threads context and tcb. It is important to note that the main thread, scheduler context, timer, and signal handler are all created in the first call to this method. Then in every subsequent call, only a worker thread is created.
- More specifically, the tcb holds the thread id, the status which will be runnable, the context, the elapsed quantum, the thread it is waiting on to finish, and entry_time and scheduled to help calculate response and turnaround time.
- NOTE - `node * ptr = list_of_tcbs_pjsf;`
- This line of code is present within this method to avoid my library from seg faulting. I genuinely could not figure out why this line was needed.

worker_yield:

- We simply change the current running thread to runnable, save the context, and context switch to the scheduler.

worker_exit:

- We essentially free the current block that holds the current thread because it is exiting. Then we must unblock all threads that could have been blocked because they were waiting on the completion of this thread. We also calculate the turnaround time here. Then we invoke the scheduler to run as we need to schedule a new thread.

worker_join:

- In this method, we wish to block the current thread. Now it is possible that the thread that is joining could have exited before the call to join, so if it has, we should free this thread. Then we should also set the current thread to wait on the thread that called the join. After we invoke the scheduler.

worker_mutex_init:

- Initialize the lock to be 0 and the list of threads that are blocked to be NULL

worker_mutex_lock:
- We apply the sync lock test and set to see if the thread accessing the mutex has the right to continue. If the test is successful, we can block the thread and input the thread into the block list of the mutex and invoke the scheduler. Otherwise, we allow the thread to bypass the mutex.

worker_mutex_unlock:
- We simply reset the lock to 0 and free all elements in the block list as well as set the status of these threads to Runnable.

worker_mutex_destroy:
- We invoke worker_mutex_unlock because it frees any dynamic memory that was created.

schedule:
- No alterations made

sched_psjf:
- In this method I accounted for four cases
- If previous thread is not null and next thread is not null
    - In this case we are context switching from one thread to another. Therefore, I will add the previous thread back to run queue and then switch context to the next thread so the next thread can run. We also set the timer and signal action in this step. We update the quantum for the previous thread as well.
- If previous thread is not null and next thread is null
    - In this case we will increase the elapsed quantum and run the previous thread again.
- If previous thread is null and next thread is not null
    - We simply run the next thread
- If previous thread is null and next thread is null
    - We invoke the scheduler.
- In all these cases we also set the timer and the signal action.

sched_mlfq():
- Not implemented

add_to_tcb_LL_pjsf:
- We add the thread to the runqueue based on its time quantum so the list is ordered from least to greatest in terms of elapsed time.

get_next_tcb_LL_pjsf:

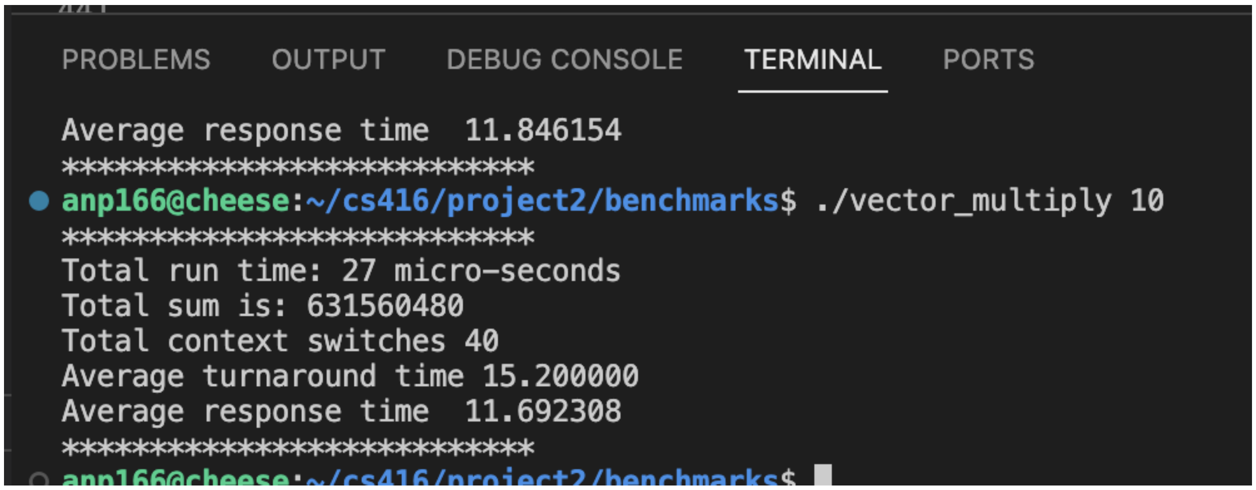
- We will get the next thread that is ready to run or runnable. We just traverse through the list and remove the tcb and make sure the list is still appropriately linked.

Unblock_threads:

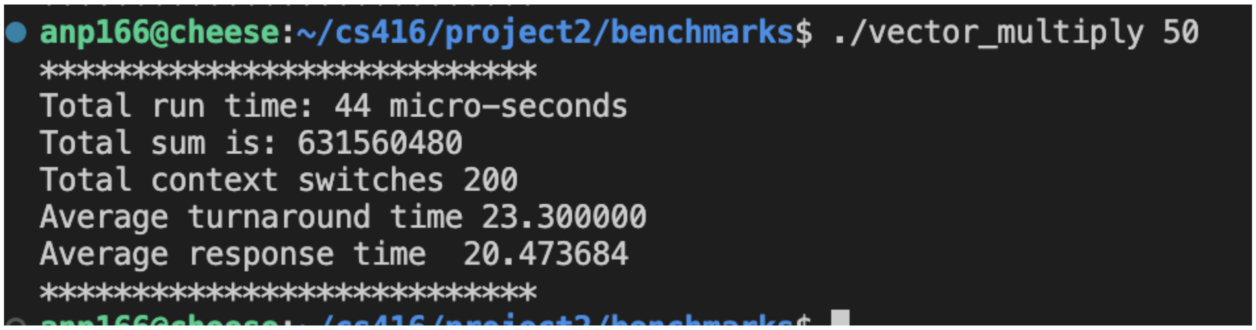
- Traverse through tcb list and find the thread that is waiting on the thread that has just finished with is inputted into the function as a thread id. Then just set the status of the thread to runnable.

Is_in_run_queue:

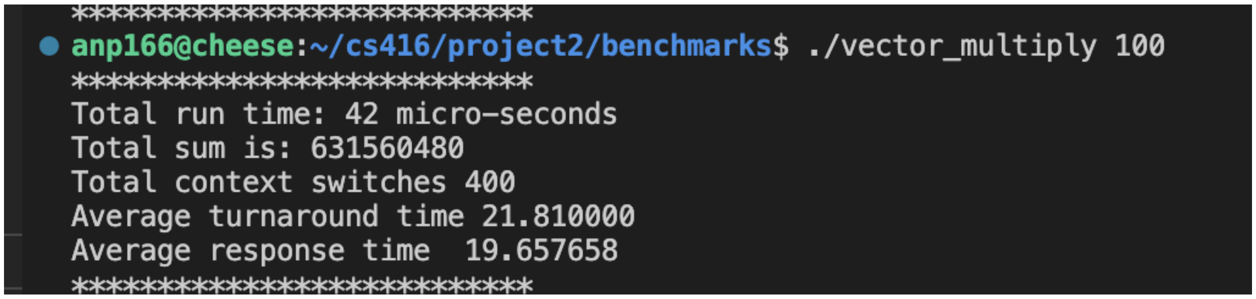
- This is just used as a check that a job is not inputted into the run queue, so it does not get inputted in again.



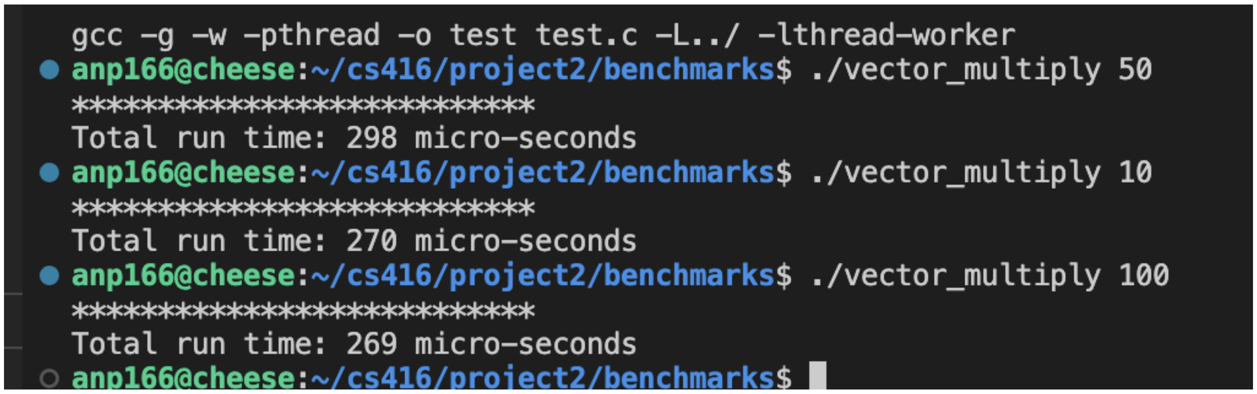
```
Average response time  11.846154
***************************
anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 10
***************************
Total run time: 27 micro-seconds
Total sum is: 631560480
Total context switches 40
Average turnaround time 15.200000
Average response time  11.692308
***************************
anp166@cheese:~/cs416/project2/benchmarks$
```



```
anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 50
***************************
Total run time: 44 micro-seconds
Total sum is: 631560480
Total context switches 200
Average turnaround time 23.300000
Average response time  20.473684
***************************
anp166@cheese:~/cs416/project2/benchmarks$
```



```
****************************
● anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 100
****************************
Total run time: 42 micro-seconds
Total sum is: 631560480
Total context switches 400
Average turnaround time 21.810000
Average response time  19.657658
****************************
```



```
gcc -g -w -pthread -o test test.c -L../ -lthread-worker
● anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 50
****************************
Total run time: 298 micro-seconds
● anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 10
****************************
Total run time: 270 micro-seconds
● anp166@cheese:~/cs416/project2/benchmarks$ ./vector_multiply 100
****************************
Total run time: 269 micro-seconds
○ anp166@cheese:~/cs416/project2/benchmarks$ 
```

In terms of run time compared to pthread library, my library runs considerably quicker. However, the issue is that my program is not 100% correct in ./vector_multiply, but for the most part it does return the correct solution. I think some of the issues could be cleaning memory which my library is not doing fully correctly.