

**CS518: OS Design**  
**Assignment 1: Thread Scheduling**

**Members**

Roshan Patel	rrp78
Daniel Hidalgo	dh624
Saad Ismail	sni7

**Overview**

Our scheduler implementation was vigorously tested for correct execution using a series of debugging tests that can be found in the debugging.c file.

First, we decided to performance test our scheduler by varying the design parameters: Running\_Time and Priority\_Levels. Please see the readme.pdf for more information about how these variables affect the scheduler design.

The performance criteria we are designing for is execution time of a test program. This test function will spawn N number of threads. Half of these threads will attempt to claim a lock to edit global memory while the other half do likewise using another lock and different memory. These threads unlock and exit when they are complete. A few threads join on others in this process. Lastly, the main thread joins all N threads. The test program will terminate when all threads are complete.

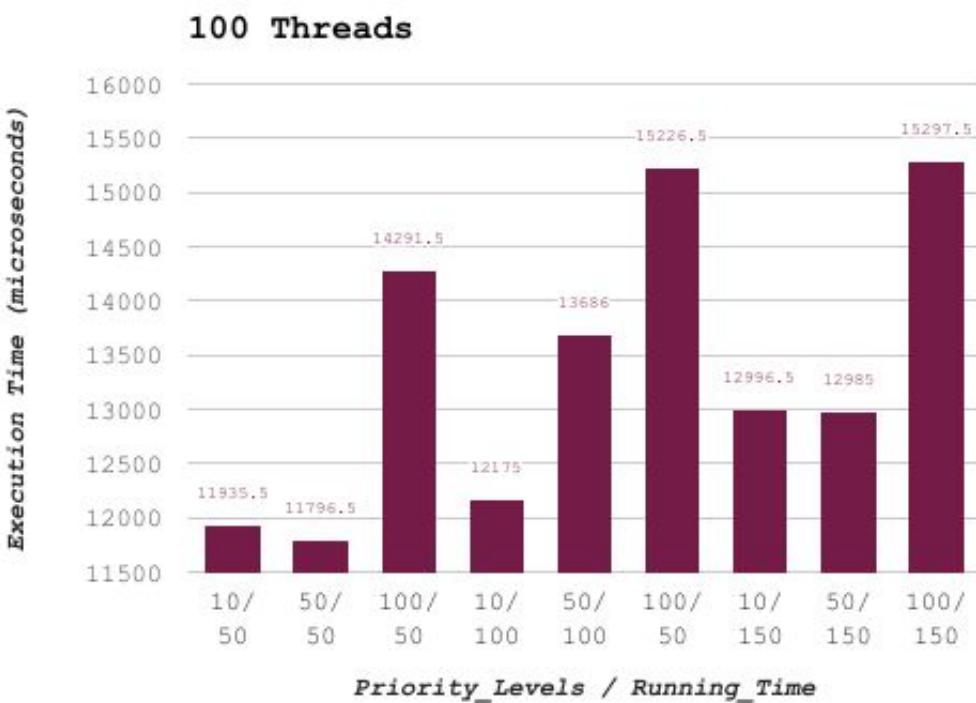
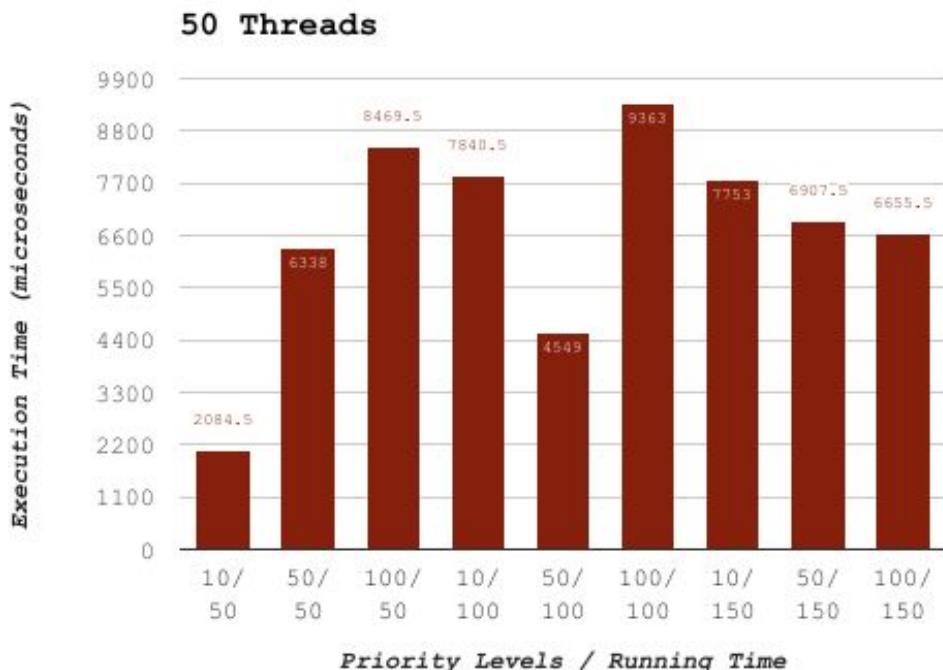
The scheduler is tested with the following configurations with N (number of threads) varying from 50, 100, 500, 1000, and 2000. There will be 3 runs and the average of these will be counted for analysis.

Priority Levels	Running Time
10	50
10	100
10	150
50	50
50	100
50	150
100	50
100	100
100	150

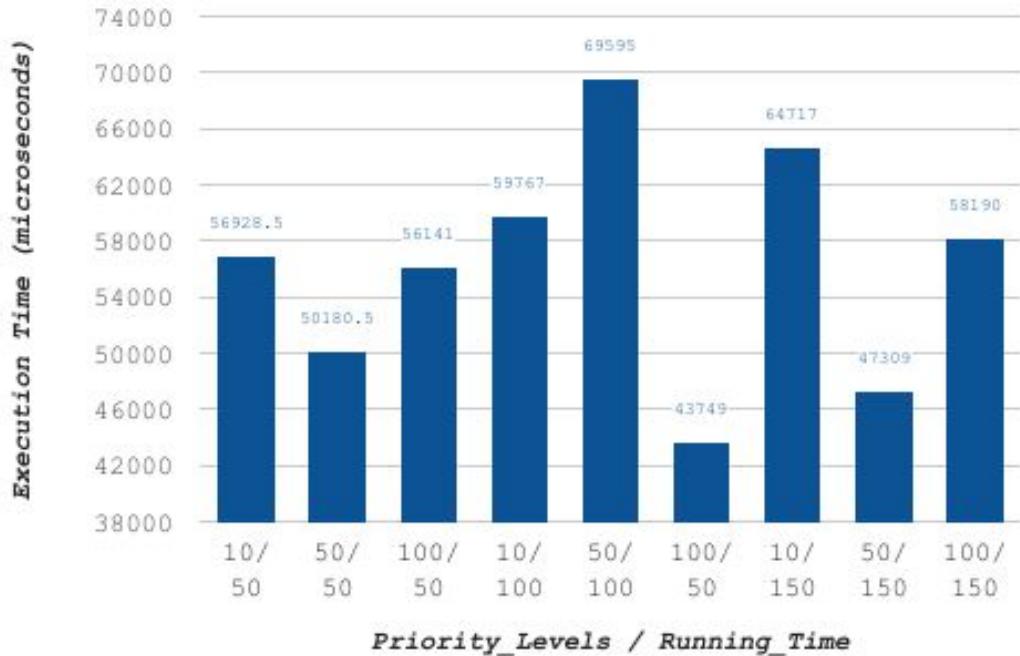
Table 1: Configurations to test

Using ilab machine: java.cs.rutgers.edu

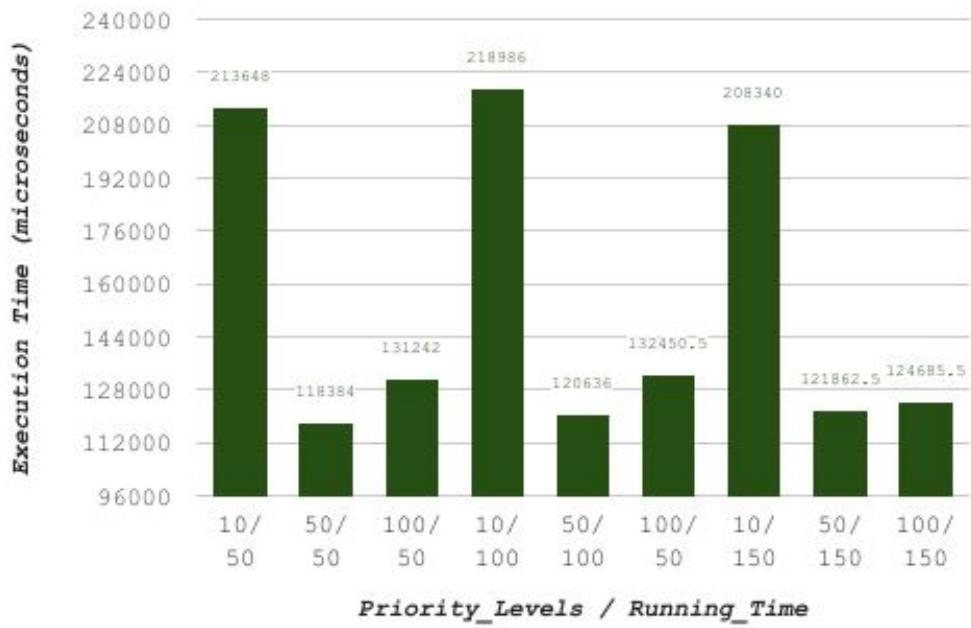
Please see page 2-4 for bar charts that depict how the configurations compare for a constant number of threads, N.

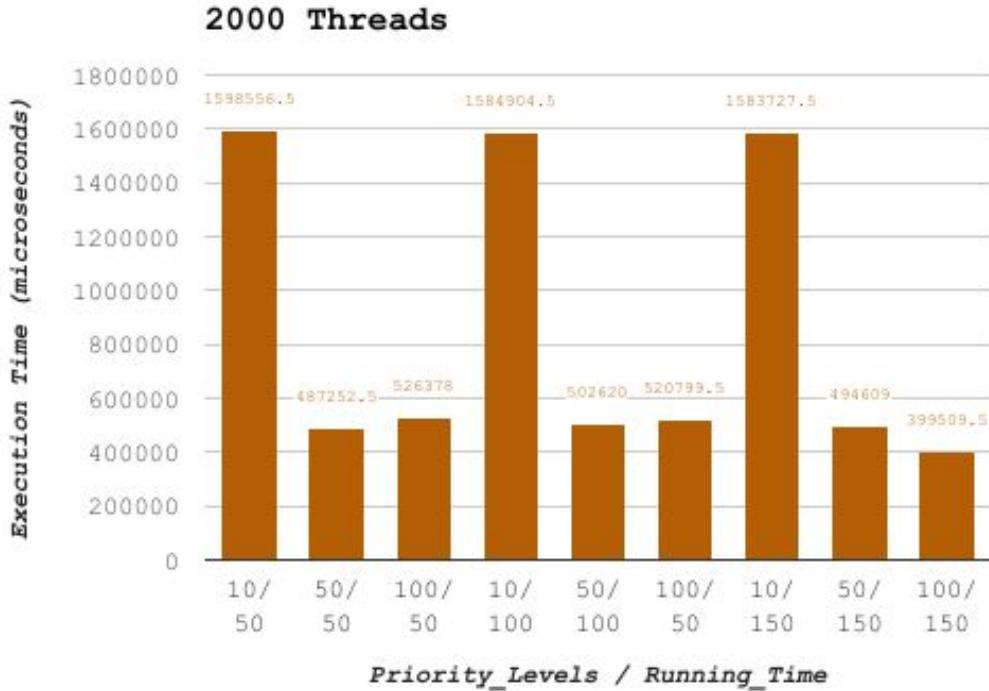


### 500 Threads



### 1000 Threads

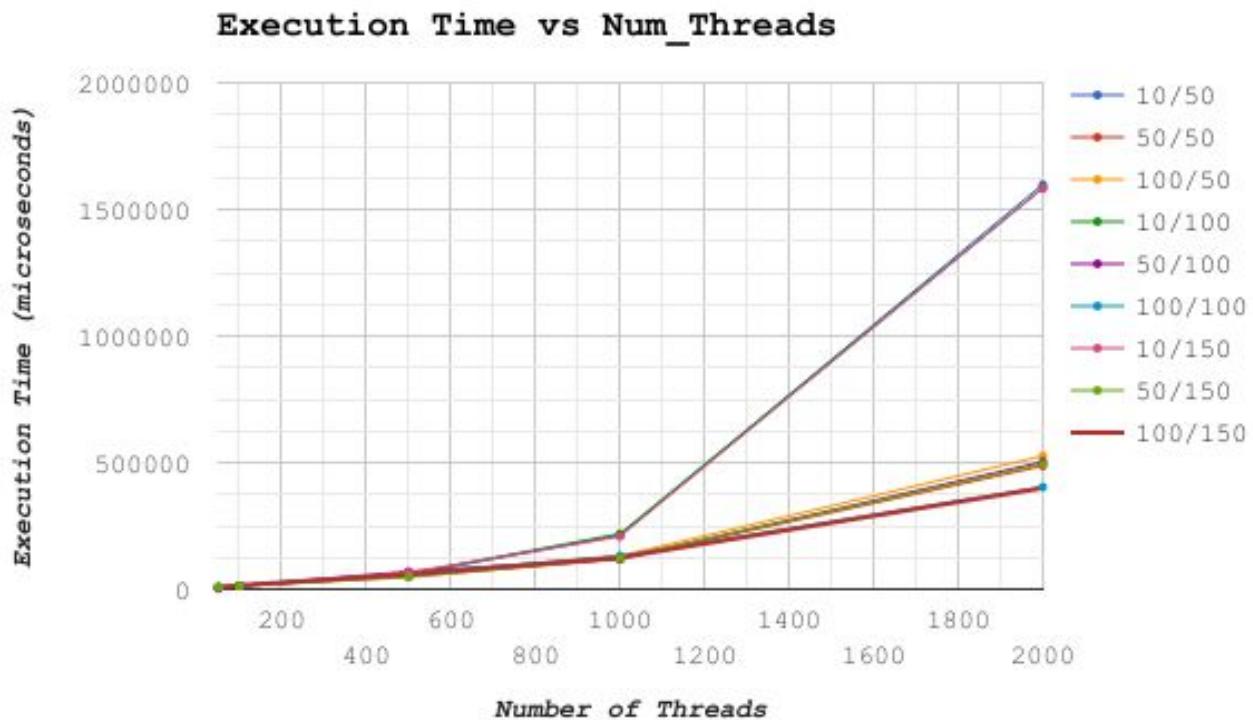




It is evident that the configurations with smaller values for Priority Levels (PL) and Running Time (RT) perform better than the others when the number of threads is smaller. However, they are outperformed by about 300% in terms of execution time when the number of threads reaches greater than 1000.

In the cases with 1000 and 2000 threads, the configurations with more priority levels available performed faster with little impact from increasing running time. This makes sense with our testing algorithm as very few threads will be ready at any time. The test skews our performance statistics in this way. In the case where many threads are in a ready state, there may be more disparity when varying this parameter.

The following page shows a line graph aggregating all of the above bar graphs to compare the performance (in terms of execution time) of each configuration and the number of input threads increases.



This plot highlights that the configurations converge into three categories based on their performance as the number of threads increases.

The best performing configurations are:

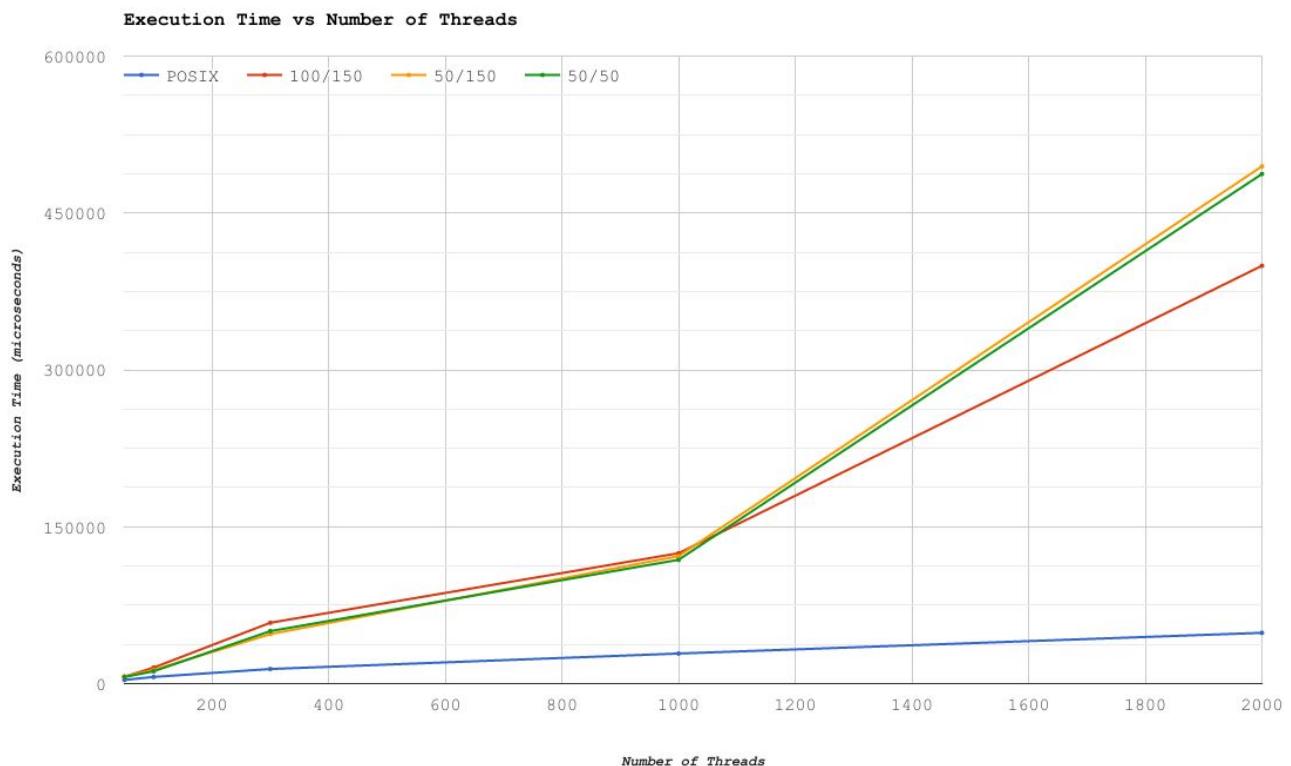
50 PL / 50 RT  
 50 PL / 150 RT  
 100 PL / 150 RT

Out of these three, 100/150 appears to scale the most favorably with increasing number of threads.

Lastly, to depress ourselves, we will test the best performing configuration of our scheduler against the POSIX pthread library. The same test file will be used along with the same testing strategy ( $N = 50, 100, 500, 1000, 2000$ ). Results are shown on page 6.

Number of Threads (N)	Execution Time (microseconds)
50	3510
100	6387
500	13964
1000	28759
2000	48550

Table 2: Results for POSIX's pthread library



The above line chart compares the top performing configurations with the POSIX library implementation.

The POSIX library outperforms our thread scheduler with this testing program. It scales very well as the number of threads increases. Its performance with respect to execution time is very optimal in comparison.

However, there are some caveats to these measurements. Our thread scheduler implementation is entirely using user-level threads while the POSIX library uses kernel-level threads. Since kernel-level threading has the opportunity to take advantage of parallelism, it is likely that the true execution time (without any parallelism) of the POSIX implementation would be significantly higher. Our thread scheduler implementation is limited to one core and non parallel operation.

Despite our scheduler presenting no competition with the POSIX pthread library, it is still a fairly optimal user-level thread design. Depending on the exact program specifics (number of threads, utilizing mutexes, lots of joining) and what kind of metrics are favorable (execution time, response time, etc), different configurations of the thread scheduler would be more optimal. In this assessment, we determined a set of configurations that were optimal for execution time for a program that heavily used mutexes and thread joining.