CPSC 457 – Spring 2019
Assignment 3 – Tutorial 1

Eric Austin – 30037742

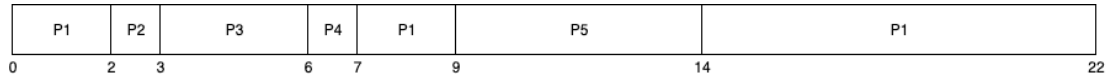
June 3, 2019

1

The CPU utilization rate is the probability that at least one process will need to execute instructions on the CPU, which is equal to 1 minus the probability that no process needs to execute on the CPU (ie all processes are waiting on I/O). Given that we are running 8 processes and each has a probability of waiting on I/O of $75\% = 3/4$, the utilization rate is:

$$1 - (\frac{3}{4})^8 \approx 0.90 = 90\%$$

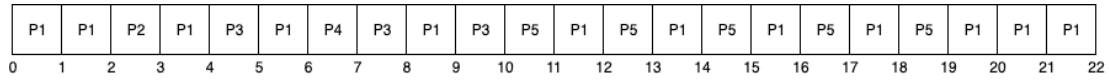
2



Process	Arrival	Burst	Start	Finish	Turnaround	Waiting
P1	0	12	0	22	22	10
P2	2	1	2	3	1	0
P3	3	3	3	6	3	0
P4	5	1	6	7	2	1
P5	9	5	9	14	5	0

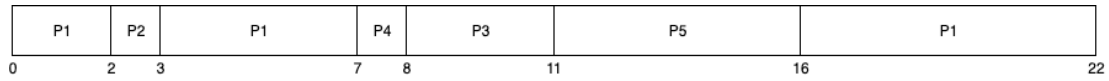
$$AverageWaitTime = \frac{10+0+0+1+0}{5} = \frac{11}{5} = 2.2s$$

3



There are 19 context switches.

4



5

See uploaded C file *count.c*.

6

Test File: medium.txt			
# of Threads	**Observed Timing**	**Observed Speedup**	**Expected Speedup**
original program	30.008	1.0	1.0
1	31.916	0.97	1.0
2	17.422	1.72	2.0
3	11.585	2.59	3.0
4	8.721	3.44	4.0
8	4.410	6.80	8.0
16	5.678	5.28	16.0

Test File: hard.txt			
# of Threads	**Observed Timing**	**Observed Speedup**	**Expected Speedup**
original program	10.164	1.0	1.0
1	10.230	0.99	1.0
2	5.153	1.97	2.0
3	3.476	2.92	3.0
4	2.592	3.92	4.0
8	1.741	5.84	8.0
16	1.371	7.41	16.0

Test File: hard2.txt			
# of Threads	**Observed Timing**	**Observed Speedup**	**Expected Speedup**
original program	10.199	1.0	1.0
1	10.126	1.01	1.0
2	5.362	1.90	2.0
3	4.313	2.36	3.0
4	3.430	2.97	4.0
8	1.401	7.28	8.0
16	1.376	7.41	16.0

In all three cases, we observe an improvement in speed that increases with the number of threads used to solve the problem. However, the improvement is not as great as the increase in the number of threads, eg. using 16 threads doesn't give us a 16x speed improvement. With smaller numbers of threads, we are close to a one-for-one improvement, ie. two threads is about twice as fast, three threads is about three times as fast. The improvement slows as the number of threads grows; we observe diminishing returns to scale. I hypothesize that this is due to the overhead of setting up threads and the data structures they use. As the time needed to find the actual solution shrinks, this fixed cost represents a greater share of the total time and a 'floor' for the execution time.

Another issue with my code may be the lack of communication between threads. In the single threaded solution, the looping search breaks as soon as it finds a factor and returns not prime. In the multi-threaded solution, any thread that finds a factor will break and return not prime. However, the other threads continue to run and the main thread is waiting for all to finish. Some method to alert the other threads that a factor has been found and to quit searching may help to speed things up, but the extra code needed to set up this messaging and getting the threads to communicate may also add overhead that would slow things down.

7

See uploaded C file *scheduler.c*.