I used the given python script to generate the graphs but I am not too familiar with python or excel so I struggled a bit. In the graph the different colors represent different processes, I didn't know how to change the key labels. Also I didn't know how to expand the x-axis range to really see the processes in Q0, so you will not see it on the graph for test1 but when running in on terminal, you will see it correctly outputted.

Test 1:
In the first test I implemented a mixed workload that has both I/O intensive process and CPU intensive process. After forking, the parent process runs the I/O intensive workload printing “I/O Intensive” in the loop. The child process runs the CPU intensive process which is basically looping through for loop 1000000000ULL times and setting i as volatile unsigned long long i. In graph 1 the scheduler prioritizes q0,q1, and q2 respectively. The 2 processes enter about the same time and you can see this alternating effect amongst each other starting at q0 then to q1 then ending at q2. Both of the processes run 1 tick at q0, 2 ticks at q1 and 8 ticks in q2. If one of the processes ends before the other one, you will see that the process that hasn't finished will be displayed simultaneously next to each other towards the end. When running this test, there will be a slight pause due to CPU intensive before it runs after IO Intensive prints out.

Test 2:
In the second test I am testing to see if the boosting mechanism improves performance of a long running CPU bound process. In the test, I forked 7 times; each of the processes does an cpu intensive function that prints and generates random numbers by moduling the variables of the for loops together. With this implementation, there are a lot of processes and it takes a very long time for them to complete each process, so there will be a process that waits 50 or more ticks. With that, this process then gets boosted to the higher priority queue. In graph 2 you can see boosting occurring where processes in the lower queue are boosted up to the higher queue after 50 or more ticks. When running the test for this, there will be a slight pause due to CPU intensive before it runs.

Test 3:
In the third test, I am basically cheating 1 process into staying in the highest priority queue by creating a loop that iteratively calls sleep(1) to relinquish the processor before the time interrupt occurs. Calling sleep(1) 20 times basically yields the CPU before the end of the tick count and never getting down to the lower priority queue. In the graph you can see that it stays in the lower priority queue for almost the entire time.