Test Cases

## Program requirement: execute –v Program\_1.osx <arriving time> Program\_2.osx <arriving time> … Program\_n.osx <arriving time>

**This requirement has been met and can be demonstrated with the following test cases:**

*Note on switches available*

*-v = Verbose - showing decode output*

*-g = Gantt chart, omitting verbose output and instead, showing <Clock cycle> <process ID> <PCB state> for each process that is loaded.*

* First demonstration shows that I am checking for memory overlap upon load, I have created an OSX file with an overlapping load location called badLoadAddress.osx

execute -g IOBound.osx 10 test\_cases.osx 13

Results:

Load address overlap exception.

Program ended with exit code: 1

* The second demonstration shows that when executing two or more programs, it will run them based on arrival time. To demonstrate this, run the first command and then the second command and notice that the order is different based on the arrival time modification.

execute -g prog1.osx 10 prog2.osx 12

Result: looking at the Gantt chart, you can see that prog1 ran first and prog2 ran second

execute -g prog1.osx 10 prog2.osx 8

Result: looking at the Gantt chart, you can see that prog2 ran first and prog1 ran second, this is because I adjusted the arrival time on prog2 to 8 which is lower than the arrival time of prog1 which is 10.

## Successfully implement 5 states model and PCB is updated

**This requirement has been met and can be demonstrated by using the -g switch which shows the different states. When running the following command, IOBound will require that you press 'q' to continue. After IOBound is complete, CPUBound will execute, which as a fork in it, so there will be a status of Wait from the parent shown.**

execute -g IOBound.osx 10 CPUBound.osx 12

Result: IOBound and CPUBound started in the "New" state, which can be seen at the very top where the PCBs are printed, IOBound then enters the running state while CPUBound enters the ready state, as shown in the gantt chart. After pressing 'q', IOBound enters the terminated state, then CPUBound enters the running state. CPUBound then forks into PID 0 and the parent PID, which is 102 in this case. The Parent enters the Waiting state while the child enters the running state. After the child terminates, the parent terminates as well. I was not able to get the final "Terminated" state for the parent to print out, but it did enter that state, which is why the program ended back the prompt.

## Successfully implement job queue, ready queue, I/O queue and PCB is updated

**This requirement has been met as demonstrated by the previous test case. Please let me know if there is something else you would like to see to demonstrate that this requirement has been met.**

## Successfully implement Wait system call and resources are released

**This requirement has been meet as seen by the parent entering the wait state while the child finishes. Once the child finishes execution, the child terminates and memory is cleaned up then the parent finishes execution. Demonstration from the Gantt chart when running:**

execute -g IOBound.osx 10 CPUBound.osx 12

## Successfully run osx under your shell

**This requirement has been met and can be demonstrated by running:**

osx IOBound.asm 800

Result:

osx version: 0.3

osX Assembly Complete see file IOBound.osx

## Successfully implement fork( ) – execute( )

**I have implemented fork() as demonstrated when executing CPUBound.osx. The fork actually loads a new copy of CPUBound into memory at the tail end of memory allocation. I am hoping this covers the requirement for exec() as well.**