# Preamble

## Work Breakdown

* Project.java – Nikki Pruitt
* Generator.java – Nikki Pruitt, Joshua Hester
* Job.java – Nikki Pruitt, Joshua Hester
* Segments.java – Nikki Pruitt, Joshua Hester
* Memory.java – Nikki Pruitt

## Integration

The integration for this program was handled through the use of GitHub. As we both worked individually on the project, we periodically synched our work so that the other member would have access to the changes we had made. From GitHub, we could download the latest version of the project. We also held two group meetings to work together on the project. During the first group meeting, we worked together to create the Project.java, Memory.java, Job.java, and Segments.java files. After this meeting, we both individually to create the Generator.java file and get it working correctly. At the second group meeting, we tested the program and finished up the project.

# Design Details

## Project Class

The Project class is the main for this program. It creates a Memory object and three instances of the Generator class. After each instance of the Generator class has had a chance to execute and display their results, the Memory object will reset all jobs and segments to the specifications decided at the beginning of the program.

## Generator Class

The Generator class deals with the execution of the three test cases. It takes in a Memory object, the ordering algorithm, and the memory allocation policy. In order to run each case, the Generator instance in the Project class will call the Generator’s execute method. This method will run for 30 seconds. Every second, it calls the updateReadyQueue method, which updates the program based on the specified order and allocation schemes. This method takes into consideration the fact that it is using four processors that work in parallel. The numbers of finished and waiting jobs are both calculated in their own methods, as well as the amount of wasted space. The Generator class also handles displaying the results of each time unit to the command line as well as printing the results to an appropriate text file.

## Job Class

The Job class creates and maintains a Job object. It holds the identification number, amount of memory requested, the execution time requested, the segment number assigned to the job, the execution time remaining, and the job’s status. Upon creation, the Job object will generate a random memory request between 16-64 and a random time request between 2 and 10. The Job object also sets the default status to “waiting” and the default segment number to -1. Job objects have the ability to decrement their execution time, update the time and status, reset to the original information generated upon declaration, and check to see if it is finished execution.

## Segments Class

The Segments class deals with the creation of individual memory segments. It holds the segment number, segment size, if the segment is occupied, the job assigned to the segment, and the amount of wasted space in the segment. It takes in the specified size of the segment. Upon creation, the Segment object sets the occupied value to false, the default job to null, and the wasted space to zero. Segment objects have the ability to remove a job, calculate the wasted space, update, and reset.

## Memory Class

The Memory class works as the memory for the hypothetical operating system. The Memory class creates the required amount of Segment objects with the sizes specified for the project. The Memory class also creates the specified amount Job objects. The Memory class has the ability to reset the segments and jobs to the values that were created at the start of the program. This is used after the test cases finish so that it is possible to use the same data for all test cases.

# Conclusion

Typically, the most efficient test case was the third, which used the shortest job first ordering algorithm and the best-fit allocation policy. This can be explained through the fact that the shortest job first algorithm sorts the jobs so that the jobs are in descending order based on their requested execution times.

The shortest job first algorithm allows the hypothetical operating system to complete more jobs in the time allotted. The other two test cases for the most part behaved the same. Both cases were less efficient than the third test case. This can be attributed to the fact that it uses the first come, first serve algorithm. This algorithm takes in the first job in the ready queue, regardless of size or burst time. This means that if the smaller jobs come in towards the end, they will not be able to execute until the larger jobs finish, which might not happen during the given timeframe. Although there was not usually a difference between the two cases, in the instances where there was a difference, test case two tended to be more efficient. This is due to the fact that the second test case uses the best-fit allocation policy, which fits jobs into the segment that it best fits into.