Assignment 2 Report
Name: Miguelangel Tamargo
Panther ID: 5866999

In this assignment, I was responsible for creating a scheduling algorithm program in C that efficiently manages the execution of multiple processes. The program supports various scheduling algorithms, including Shortest Job First and First Come First Serve. To streamline the process management, I implemented a structure that encapsulates essential attributes such as arrival time and burst time, making it easier to perform calculations and comparisons.

A significant design choice was opting for quicksort from the C standard library instead of bubble sort. I found quicksort to be far more efficient, especially when dealing with larger sets of processes. Additionally, I maintained both an arrival list and a burst-sorted list to enhance process selection based on the scheduling algorithm being used. This decision improved the overall responsiveness of the program.

One of the main challenges I encountered was effectively managing the dynamic list of arrived processes. It was crucial to track the process clock accurately, ensuring that only processes that had actually arrived were considered for execution. I addressed this by creating a dedicated function that updates the arrival list after each process completes, allowing for precise evaluation of which processes were ready to run.

To address the segmentation faults encountered during the development of my scheduling algorithm program, I conducted a thorough review of my memory management practices and data structures. The first step was to ensure that all pointers were correctly initialized before use, as uninitialized pointers were a frequent source of access violations. I also carefully examined array bounds, particularly when iterating through process lists and completed processes, to avoid accessing out-of-bounds elements. Additionally, I utilized debugging tools and print statements to trace the program's flow and identify the specific lines where the segmentation faults occurred, helping me pinpoint the conditions under which these faults were triggered. I implemented proper checks for dynamic memory allocation, ensuring that memory was allocated before being accessed and was appropriately freed after use to prevent memory leaks. These steps helped stabilize the program, significantly reducing the occurrence of segmentation faults and enhancing its overall robustness.

In reflection, this project enhanced my understanding of process scheduling in operating systems while emphasizing the importance of clean design patterns. It was a valuable opportunity to refine my skills in modular coding, huge in building memory management, and algorithm implementation. Overcoming these challenges was not only educational but also enjoyable, providing a great platform to strengthen my grasp of core computer science concepts.