Project Report #4

Memory Allocation Schemes

Sujeet Ojha
sxo160430@utdallas.edu
April 7, 2020

I. Problem Statement

The assigned task was to implement, explore and analyze distinct memory partitioning algorithms. Four algorithms: best fit, first fit, next fit, and worst fit were required to be analyzed. Furthermore, it was required to run these algorithms for a total of a thousand tasks and to count the length of time required to process the task stream for every algorithm.

II. Importance of Placement Algorithms

Placement algorithms are extremely beneficial as memory is one of the biggest aspects of an operating system. Therefore, it is imperative for the OS designer to design a clear and calculated system that assigns processes to memory in an efficient manner. Since processes need to be loaded and swapped out quite frequently, having an efficient algorithm that frees and allocates blocks as necessary is an extremely important task

III. Algorithm Usage

The four different algorithms described in this report are very different in the way they interact with memory. The best fit algorithm chooses a block of memory that is closest in size to the task's request. The first fit algorithm sequentially scans the memory and chooses the first location where the task can be placed. The next fit algorithm keeps track of the last placement and uses that information to place a new task in memory. Finally, the worst fit algorithm allocates the largest free partition available in memory.

This report will utilize the algorithms each time a new task needs to be placed in memory. These algorithms will also run an equal amount of times under similar circumstances in order to ensure results are unbiased.

III. Approach

The programming language selected for this project is C. The approach taken was to first generate a random task stream consisting of the size needed for allocation and duration. Keeping the limited units of memory available into account, an algorithm was used to place the tasks in memory. A compaction sequence was utilized in order to ensure that the program makes efficient usage of memory. This sequence of events repeated a thousand times for each algorithm. Once this series of tasks was completed, the total amount of time consumed by the stream was returned for further analysis.

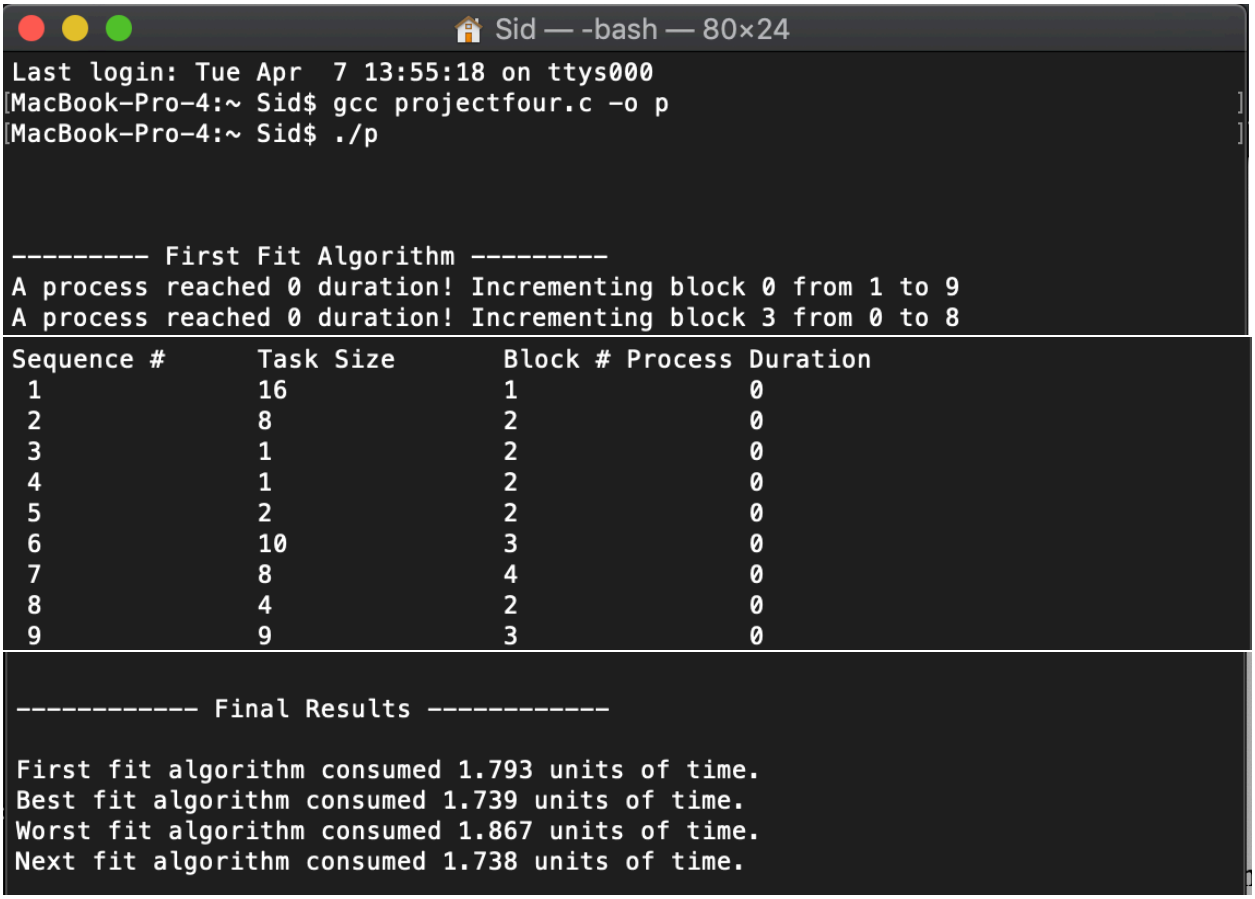
IV. Solution

Implementing the four distinct algorithms proved to be a difficult, but manageable task. Since each task stream needed to hold the size, along with the duration, a struct was utilized. The program first creates an array of a thousand structures and fills the size and duration with randomized values from 1-16. Once the task stream is created, the program creates an array that represents the 56 units of memory. This array is initialized with the same values for each algorithm {16, 16, 16, 8, 2}. However, the compaction scheme used ensures that each algorithm utilizes the memory in a distinct and dynamic manner. The program then processes the task stream by using each algorithm and keeps account of the time consumed. The general strategy utilized was to increment the time consumed by the algorithm by one every time a task cannot be placed while decrementing the time remaining for all tasks in memory by one. Tasks were removed from memory in order to free up space if the time remaining for the task reached 0. Once each task was processed, the largest time remaining for tasks in memory was added to the time consumed by the algorithm. This resulted in the total amount of time consumed by the stream.

The information regarding the total amount of time consumed by the stream was then used to analyze each algorithm and make conclusions. Furthermore, in order to explore the algorithms in greater depth, the algorithms were run with different sample sizes of tasks. A plot was then created using the various amount of data gathered in order to visualize the results.

Figures:

- *Plot (1):* Plot (1) depicts the build and execution of the program in the terminal.
- *Plot (2):* Plot (2) depicts additional testing of the program using a new set of data.
- *Plot (3):* Plot (3) depicts a plot of the results with distinct task stream sizes.

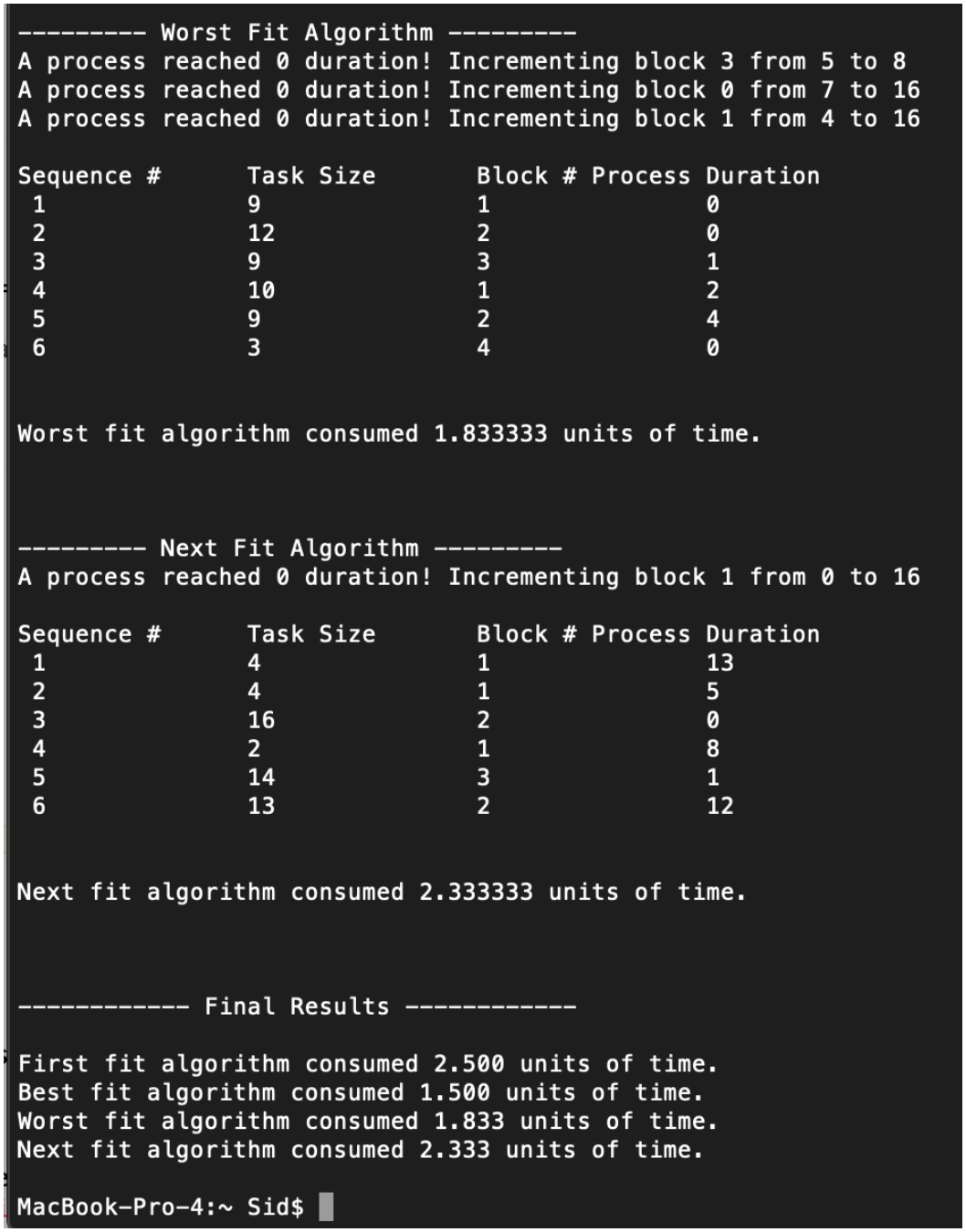


Plot (1): Build and Execution of the program

The local terminal was used to build and execute the code. The program was traditionally compiled without added statements. Upon execution of the program, the user is able to visualize the events such as compaction that occur for each algorithm. Furthermore, a visual representation of the tasks in memory is displayed. The program also displays the final results that make it easy for the user to analyze the time consumed by each task stream.

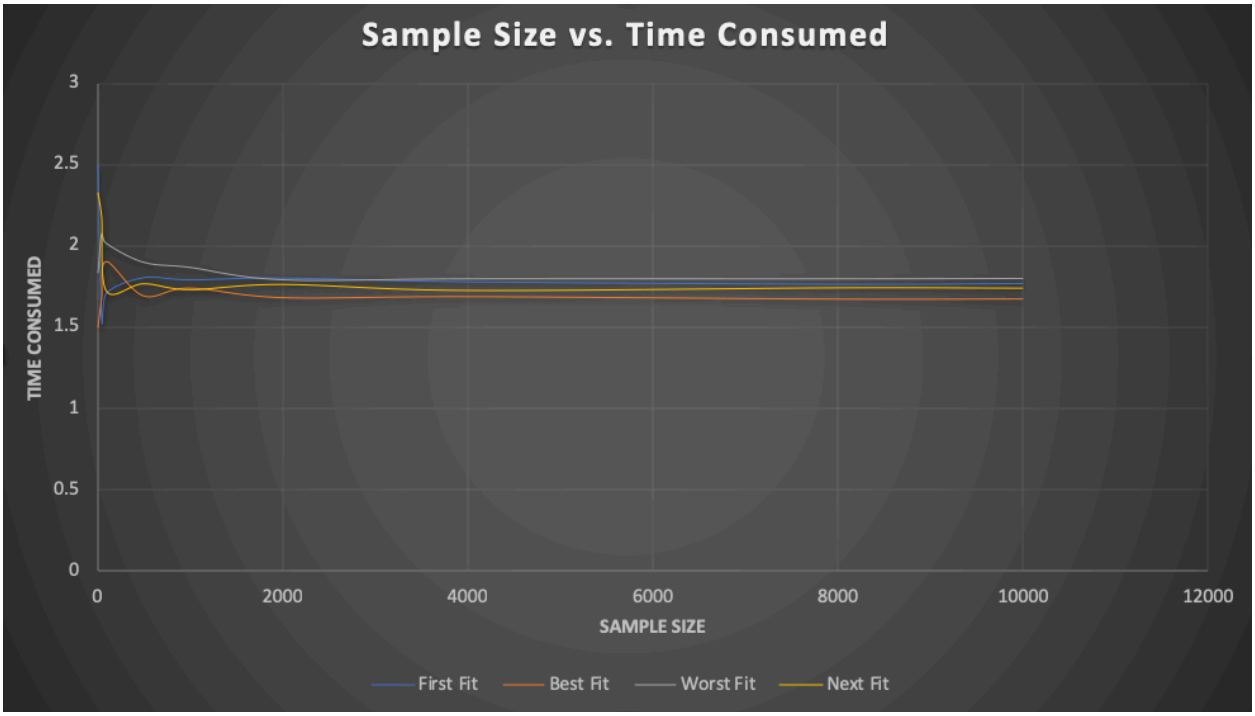
With a sample size of a thousand tasks, it is clear that the next fit algorithm runs the fastest while the worst fit algorithm tends to be the slowest one.

Note: The program can also be executed on the CS1 Linux server.



Plot (2): Additional Testing

A new sample size was utilized to analyze the four algorithms. In this test, a sample size of 10 tasks was utilized. The results of this test vary distinctly from the results of the test with a 1000 samples. In this test, the best fit algorithm is the fastest algorithm while the first fit algorithm is the slowest algorith. The general trend that can be observed by the program is that with smaller sample sizes, less realiable information is given. Since memory is generally large, considerations about algorithm speed should be made analyzing the larger sample sizes.



Plot (3): Plot with various samples

The plot above depicts the time consumed by each alogirthm with respect to the amount of tasks being processed. An interesting observation that can be made is that with a smaller sample size, the time consumed seems to vary drastically for each algorthm. However, as the sample size increases, all aglorthms seem to have a steady “time consumed.” Using the plot, it can be concluded that as a higher number of tasks are processed, the best-fit algorithm works in the most efficent manner while the worst-fit algorithm is the least efficent.