**Project 1 Report**

**I. Contributors & Responsibilities**

This project was a collaboration between Dan Shadel, Chalet Shelton, and Peyton Pritchard, for CSCE 4600 with Professor Mikler. The objective of the project was to create algorithms that could optimize process scheduling for sets of processors in different circumstances. These circumstances could be variance in processor memory, difference in processor speed, and/or a change in the way processes arrive to be scheduled. In order to make sure all group members were able to contribute a fair amount, we delegated parts of the project to different group members by perceived difficulty and time requirement.

As a single unit, the group collaborated to complete Part 1 of the project, in which all processors are identical. This problem is discussed in this paper in Section II. Identical Processors. The group also collaborated to proofread this report before submission to make sure all of the given information was accurate.

Dan Shadel set up a Git repository for the project at https://github.com/DanShadel/OSProj1, where all commits to the project can be found. He also completed Part 2 of the project, in which the processors each have distinct memory requirements. The discussion for the algorithm for this program can be found in Section III. Processors with Memory Constraints.

Chalet Shelton set up a platform for group communication in order to facilitate collaboration between members. She also created a solution for Part 4 of the project, where processes arrive to be scheduled sequentially instead of appearing as a single chunk before being assigned to a processor. Part 4 of the project is discussed in Section V. Sequential Process Arrival.

Peyton Pritchard wrote this report, which outlines the algorithms behind each scheduling algorithm and describes difficulties that each group member faced. In addition to this, she solved Part 3 of the project, in which each processor has a different speed which must be accounted for when scheduling. This program is discussed in Section IV. Processors with Distinct Speeds.

**II. Identical Processors**

For Part 1 of the project, all of the processors have identical memory and speed capabilities, and the processes that are being scheduled have a random distribution of memory requirements (ranging from .25 MB to 8GB) and cycle count (ranging from 10E6 cycles to 50E12 cycles). Once completed, we could build on our solution to Part 1 in order to form solutions to the other parts of the project, so to make it easier to read each other’s code, we tried to make the class objects contain all necessary variables that will be needed for all parts of the project.

First, we set up schema for three kinds of class object: CPU, process, and monitor. The CPU object contains information that will be relevant to the scheduling algorithm. It has *mem*, an integer which is the amount of available memory on the processor in megabytes, as it is easier to convert gigabytes to megabytes than vice versa; additionally, the CPU object also has an integer *speed*, which is its cycle speed in gigahertz, which will be necessary to solve Part 3, and it has a boolean *occupied*, whose value depends on whether or not the CPU is able to take on another process. This boolean will be useful in Part 4 of the project, where a processor’s ability to take on a newly queued process directly affects the speed of the algorithm.

The second class object is process, which represents an individual process being scheduled. Per the project requirements, these are not true processes, but objects that act as processes for the purpose of the program without needing to use system calls such as *fork()*. Each process object contains a few variables: an integer *pid*, used to identify processes and ensure each process is unique; an integer *cycles*, which represents the number of cycles needed to complete the process; a floating point *mem*, which represents the memory requirement of the process in megabytes; and time variables for the arrival and completion of the process, which will be used to calculate the time taken for the process to complete.

The third and final class object is the monitor, which is used to synchronize the assigning task without allowing processes to be scheduled to more than one processor. The monitor object contains vectors of pointers to processes and to processors. The monitor is also responsible for generating the vector of 200 processes, generating the vector of 5 processors, assigning processes to the appropriate processor, printing the processes that are queued to each CPU, and performing insertion sort on the list of processes. As the processes are dispatched SJF (shortest job first), this is done in ascending order so the jobs at the beginning of the list are shorter and therefore scheduled first.

Once the object schema are set up, the rest of the program is somewhat simple. We first generate a list of processes and sort them in ascending order to make SJF scheduling easier. Then, the CPUs are generated, and the processes are evenly distributed across the processors, since the CPUs are identical and there is no reason to distinguish between them.

As mentioned before, turnaround time is found by using variables of the process class. The first, *arrival*, notes the time that the process arrives to be scheduled. For parts 1 through 3 of the project, arrival time should be the same for all processes. The second variable is *completion*, which marks the time that the process completes execution. The difference in these two numbers is the turnaround time for the individual process, which is what our group used to analyze the performance of the algorithms. This process for finding time is more or less the same for each of the programs. In part 1, the turnaround time for the last process to terminate comes out to be around 1.9835e-05 seconds.

**III. Processors with Memory Constraints**

Memory’s main impact on processor speed is that a processor can read more quickly from its own registers than it can read from other memory sources, such as RAM and disk. As such, the solution to Part 2 of the project, where the processors have different memory capabilities, is optimal when the amount of wasted memory is minimalized. That is, we try to fill each CPU to its maximum memory capability in order to minimize the number of times the CPU has to read from storage -- and, if possible, make it so the CPU never has to read from storage -- while still maintaining the SJF property where possible.

The class objects are constructed the same as they are in Part 1 of the project except for the constructors. In Part 1, all processors were initialized to be exactly the same, while the processors in Part 2 are initialized with different memory limits. As each processor has different memory capabilities, they can no longer be treated the same. Instead, each processor’s available memory is taken into account, and when/if that memory is filled, the monitor ceases to assign processes to that particular CPU. If all CPUs happen to be filled, the processes are queued and assigned SJF to whichever processors are expected to become available first.

The main difficulty behind developing a solution to Part 2 of the project was that process assignment can only happen once - that is, once a process is given to a processor to run, we cannot move that process to a different processor to complete execution, lest this project of exploring scheduling algorithms should become a full-fledged scheduler. As such, each process is able to track the time when it arrives and terminates on the processor, and each processor may keep track of whether or not it is currently occupied with some process, but the monitor’s inability to reassign processes makes this information less beneficial.

Like in part 1, the arrival time for each of the processes is the same, making calculation of the longest turnaround time fairly straightforward. For part 2, the last process to terminate has a turnaround time of about 2.458e-05 seconds. Note that this is slower than the time calculated in part 1, assumedly because the processes must be deliberately placed in certain queues rather than being given to any processor without consequence, meaning some amount of sorting must take place.

**IV. Processors with Distinct Speeds**

When running a finite number of jobs of various lengths on processors with various speeds, we ideally want all of the processors to empty their queue at the same time so none of the CPUs are left idle. To do this, we want to put the longest jobs on the fastest processors and the shortest jobs on the slowest processors. In theory, since all the processes and their speeds are given at the same time, we could calculate exactly where each process goes in order to make sure all the processors are freed at the exact same time, but as the processors would be idle while that calculation is made, it would not necessarily be ideal to do so.

As with the other programs, the processes are sorted in ascending order, but in this case they are not equally distributed across all of the processes. Instead, they are assigned to each processor in chunks - the first chunk, which contains the shortest processes, is distributed equally between the 2 slowest processors. The second chunk, with processes that take a moderate amount of time, is distributed between PC and PD, which are neither the fastest nor the slowest processors. The third chunk, which contains the most time-intensive jobs, is assigned to PE, which is the fastest processor.

Although the shortest jobs would be able to be completed incredibly quickly if assigned to the fastest processor, this does not lend itself well to achieving the optimal total time, as it may force longer jobs to run on slower processors or force the slower CPUs to sit idle as they wait for faster processors to complete their jobs. In both cases, overall time is wasted, so the algorithm used in the delivered program is designed to avoid that.

Each process arrives at the same time, so for the sake of calculating turnaround time, we use the same method used in parts 1 and 2. For part 3, the last process to terminate has a turnaround time of about 2.0969e-05 seconds. Compare this to the time taken by part 2, which had to go through a similar scheduling process to that of part 3; it is similar to part 3’s time. We believe this is due to the fact that similar algorithms were used to sort the processes for each of these problems.

**V. Sequential Process Arrival**

Part 4 of the project is about creating a scheduling algorithm that can handle processes that arrive sequentially, as opposed to parts 1 through 3 where all processes arrive simultaneously and instantaneously. Optimization of process scheduling is limited, as the monitor that assigns jobs to processors has a limited knowledge of which processes are available. Additionally, we are given the limitation that processes cannot be moved to a different processor once assigned (stated in class on March 7), meaning processes cannot be dispatched by shortest remaining time (SRT).

The driving problem in this program is to prevent processes from being blocked indefinitely. Specifically, if a process were to be continually pushed to the back of the queue as new processes arrived, it would fail to progress to the critical section, which would be a failing of the algorithm. Adding to this, the time necessary to repeatedly re-sort the same array would almost definitely outweigh the time taken by any one process to execute, meaning that it could be wasteful to sort processes as they arrive. To prevent indefinite blocking and unnecessary sorting, processes are dispatched first-in-first-out (FIFO) instead of SJF. By using FIFO order, processes do not have to wait for an undefined amount of time in order to complete, and no sorting whatsoever takes place when determining order.

The time calculation for part 4 differs from parts 1, 2, and 3 in that the processes do not have the same beginning time. This means we cannot simply see which process ends last to calculate the total elapsed time; instead, we find the total elapsed time for each process, then find which one of those is longest, even if its completion time is not the latest. Here, the longest turnaround time was approximately 1.923e-05 seconds. Though this it is generally not ideal, using FIFO instead of SJF in this case had a positive effect on the turnaround time, as it kept processes from waiting to execute for an extended period of time. With SJF, this time may have been significantly worse.

**VI. Concluding Remarks**

A major difficulty faced by the group was trying to wrap our heads around what exactly the programs needed to work. When we started working on the project, the first thing we did was grab a whiteboard and try to write out what each part of the project entailed, and that quickly turned from writing brief instruction sets to drawing pictures of abstract data types, erasing and re-writing structures repeatedly, and getting confused again the second someone thought they understood something. Taking instructions given by the teacher, turning it into the pseudocode for a working algorithm, then turning that pseudocode into an executable program could be difficult at times. Luckily, the team had a great dynamic, and with some time and patience, we managed to overcome these obstacles.

After facing the trials and tribulations that are inherent to peer programming, we ended up with a product that we are satisfied with. The purpose of this project was to create scheduling algorithms to optimize process assignment under different circumstances, and we believe that we have succeeded in that, as each part of the project is able to execute quickly and efficiently.