SIMULATION: CPU SCHEDULING ALGORITHMS COMPARISON

CSC 716 Advanced Operating Systems

Professor Huo, Yumei

Fall 2020

BY

Dardan Bajraktari

Nan Jia

Joon Kim

Introduction

The purpose of this project is to simulate four different scheduling algorithms and to compare which one will generate favorable results for at least 50 processes. The algorithms used are:

- First Come, First Served (FCFS)

Processes are assigned to the CPU in order of arrival to the ready queue.

- Shortest Job First (SJF)

Processes are assigned to the CPU based on the shortest CPU burst time available in the ready queue.

- Shortest Remaining Time Next (SRTN)

It is similar to SJF with the added requirement of preemption. If a new process arrives to the ready queue with a CPU service time less than that of the current process, it is preempted and replaced by the new shortest job.

- Round Robin (RR)

Processes in the ready queue are viewed as a circular queue. It is similar to FCFS with the added requirement of a time quantum. For instance, with a time quantum of 10, processes in the CPU will only be able to run for a maximum of 10 time units before relinquishing the CPU for replacement by another process. For this project, we have options for a time quantum of 10, 50, and 100.

When given a set of processes with CPU and I/O requirements, the simulator may be invoked to execute the aforementioned scheduling algorithms. Initially, we started with a simple version without I/O time to see if our basic algorithms worked. After, the program was upgraded to include CPU and I/O bursts. Additionally, we developed code to generate sample processes for testing using exponential distribution.

Development

The scheduler was developed using the C++ coding language. Coding was done in steps to build the program from the base upward. The best approach was to divide the workload into manageable chunks as follows:

- Input Format
- Output Format
- Scheduling Algorithms
- Simulation Execution

Overcoming Problems

It took some time to figure out the format for input data. There are two ways to do so. One is a simple version that consists of arrival time and cpu burst time. This helped us understand how to implement the complex version with I/O bursts, as seen below:

```
4 5             # number_of_processes   process_switch
1 0 6           # process_number        arrival_time    number1
1 15 400        # 1                     cpu_time        io_time
2 18 200        # 2                     cpu_time        io_time
3 15 100        # 3                     cpu_time        io_time
4 15 400        # 4                     cpu_time        io_time
5 25 100        # 5                     cpu_time        io_time
6 240           # number1               cpu_time
2 12 4          # process_number        arrival_time    number2
1 4 150         # 1                     cpu_time        io_time
2 30 50         # 2                     cpu_time        io_time
3 90 75         # 3                     cpu_time        io_time
4 15            # number2               cpu_time
3 27 4          # process_number        arrival_time    number3
1 4 400         # 1                     cpu_time        io_time
2 810 30        # 2                     cpu_time        io_time
3 376 45        # 3                     cpu_time        io_time
4 652           # number3               cpu_time
4 28 7          # process_number        arrival_time    number4
1 37 100        # 1                     cpu_time        io_time
2 37 100        # 2                     cpu_time        io_time
3 37 100        # 3                     cpu_time        io_time
4 37 100        # 4                     cpu_time        io_time
5 37 100        # 5                     cpu_time        io_time
6 37 100        # 6                     cpu_time        io_time
7 37            # number4               cpu_time
```

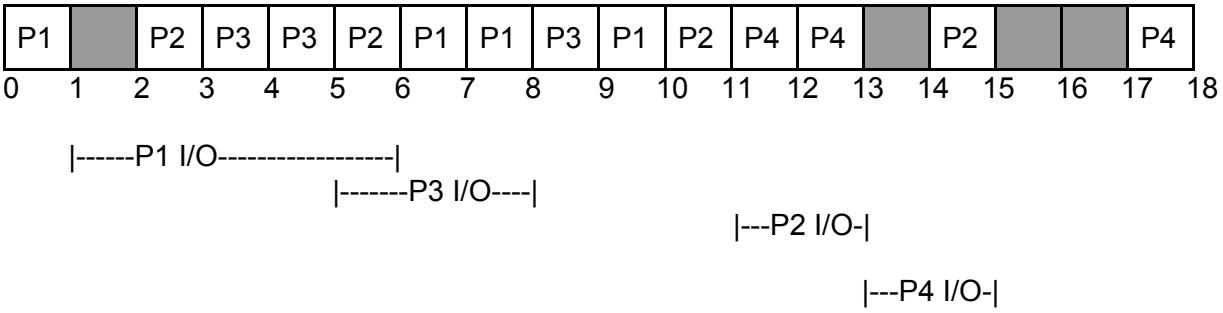
The initial required information were number of processes and process switch overhead time. What followed was the process number and arrival time, with its respective CPU bursts and I/O times enclosed between how many CPU bursts are expected within each process.

Another issue we are facing is converting the simple scheduling simulation into one with CPU and I/O bursts. We tested with 4 processes in a preemptive priority-based scheduling (small number has higher priority) as following:

	Arrival	Priority	CPU Burst	I/O Burst	CPU Burst
P1	0	2	1	5	3
P2	2	3	3	3	1

P3	3	1	2	3	1
P4	3	4	2	4	1

Gantt chart



During upgrades to the input file, we needed to find how to sort processes based on each algorithm, especially when some of the processes do not come at the same time. We needed to use the queue data structure to develop ready and wait queues necessary for our chosen algorithms.

Furthermore, there is an exponential distribution function we created to generate the random arrival interval and burst times for newly created input files. Exponential distribution may be used to describe the nature of most general computer tasks being small while having occasional large processes. With logarithmic functions and random number generators from the C/C++ standard library we were able to generate our own input files.

Conclusions

It is unfair to propose a best or worst algorithm. For example, when using round robin, if the time quantum is too small for process bursting, there will be a lot of context switches and subsequently lower CPU utilization and throughput. With overhead costs, this will cause the round robin algorithm to take too much time. On the contrary, if time quantum is too large, the round robin algorithm will degenerate into a first come first serve policy. Based on our code of simulation, the SJF is the best algorithm we chose. As for the effects of decreasing the context switch from 1 time unit to 10 time unit, we find that larger overheads will increase the waiting time as well as turnaround time.