Contents

Introduction 2

Algorithm and Design 2

Issues 5

Compilation and Execution 6

Sample Output 8

Conclusion 13

Vagner Machado

Professor John Svadlenka

CSC 340

25 Mar. 2020

Project 3 – Scheduler Programming

# Introduction

Aiming to strengthen the understanding of operating system scheduling algorithms, students were instructed to design and code an application using the C/C++ languages that implements a priority based preemptive scheduler with round robin algorithm for processes with the same priority. Priority is based on an integer value and a process with the highest priority value has the highest priority of execution. The round robin scheduling algorithm for the processes with the same priority has a time quantum of 10 time units, which starts counting from the admission of the first process into the running state. The following sections contain information related to how I designed the algorithm and data structures needed to implement the project, reported issues faced while implementation, designed support for optional arguments, provided compilation and execution instructions, and produced a list of program outputs based on input arguments.

# Algorithm and Design

During the design step of the algorithm for the proposed scheduler, I outlined additional classes and data structures need to support an organized and clean programming of the scheduler.

First, I defined the class Process to handle the instance data and behaviors of a process to be scheduled to run on the CPU. Each process has instance data to indicate arrival time, process number, burst time, time left, priority, waiting time, turnaround time, running state, and next process. Since the instance data names self explain their purposes, I will avoid elaborating on each one of them. Other than a parameterized constructor, a Process has a toString() function to access a print out of the instance data for a Process.

Next came the implementation for a singly LinkedList data structure containing functions that enable for the storage and manipulation of Process objects. Each LinkedList object has as instance data head and tail pointers to Process objects, which are, respectively, the first and last Process object in the LinkedList object. The LinkedList class functions allow the program to add a Process to the tail, remove the process in the head, remove a Process by process number, remove a Process by priority number, print processes in the LinkedList, and find the largest priority amongst all Process objects in the LinkedList.

Lastly, I designed the class Scheduler that will do the bulk of the work in its only public function, run. This class has additional private functions to aid the run function in accomplishing reading a process from a input file, deleting a line from the input file when a Process is instantiated, incrementing turnaround and wait time if needed, building a timeline of execution, building a report for default output, and printing program usage instructions. The run function contains the code developed to implement the following algorithm for the scheduler:

1. Validate text file and optional arguments.
2. Loop from time t = 0 to t = 100 (adding support for non busy status of CPU).
   1. Read time t and check file for arriving process at that time.
      1. If there is a process, instantiate it and add to ready queue.
   2. Find process with largest priority in the ready queue.
   3. If the CPU is not busy:
      1. Admit all processes with highest priority into running queue
      2. Set head to run and if more than one process in running queue, set round robin true, set CPU busy.
   4. If CPU is busy and highest priority process in the ready queue is higher than currently running process priority:
      1. Place all processes in running queue into ready queue
      2. Admit all processes with highest priority into running queue
      3. Set head to run and if more than one process in running queue, set round robin true, set CPU busy.
   5. If CPU is busy and highest priority process in the ready queue is same as currently running process:
      1. Admit all processes with highest priority into running queue.
      2. Set round robin to true case it was false, set CPU busy.
   6. Decrement round robin timer and check if it has reached zero.
      1. Case true, place head process in running queue into the tail and make its running state false.
      2. Set head of running process queue as true.
   7. Increment wait time, turnaround time, left time and other data as needed.
   8. Check if running process has time left = 0.
      1. If so, place head of running queue into terminated queue.
      2. Move head to next and set head to running case not NULL.
   9. If there are not processes running, set round robin and CPU busy to false.
   10. Goto top of loop and check if time t is in range.
   11. If so continue to step (a) else provide output required.

This was an intricate yet interesting algorithm to develop, especially when it came to fully enabling the round robin algorithm for processes with the same priority. That required extra attention to detail in order to properly perform a context switch at the right time.

# Issues

While not facing any major dead-end during design time or implementation, I would like to bring attention to how important it is to understand C/C++ pointer passing and handling. I have a good understanding of how pointers and references are utilized in those programming languages. However, I encountered a situation that required special attention to debug a problem caused by misusing a pointer. This situation arose when a process ended: That caused it to be removed from the running queue and inserted into the terminated queue. When printing the terminated queue, the program produced an unexpected printout that proved time consuming to debug. After inspecting several times the print statements and statements leading to it without any luck of finding the bug, I decided to back track and review the removal of the ended process from a linked list and subsequent insertion of it into another. Upon an exhaustive analysis and a myriad of print statements, I realized that the process next to the one finished was never set to NULL, causing the print statement to traverse into processes not yet finished. Fortunately, runningProcess.head->next = NULL was all I needed to add to fix this printing issue caused by the misuse of a pointer. This was a simple fix, yet it shows the power of pointers, the added difficulty in debugging errors caused by pointers, and how dangerous mishandling them can be - especially in more complex applications.

As mentioned in the previous section, implementing the round robin algorithm for processes with the same priority number also needed some extra attention, but nothing compared to the time needed to debug the previous printing error. This situation arose when a process with priority X was already running for a fraction of its time quantum of 10 time units. At that point, another process with same priority X was added to the running queue. Originally, I was setting roundRobin to true only when the second process was admitted, which allowed the currently running process to go beyond its time quantum. After a quick analysis and a few print statements, I set roundRobin to true whenever a process started running, forcing a possible context switch after 10 time units case another process with same priority is in the running queue.

Those were two noteworthy issues that arose while implemented the scheduler. The first illustrates the power and dangers of pointers while the second denotes an intricate detail of the round robin algorithm implementation for processes with same priority number.

# Compilation and Execution

Compiling and running this application is very straightforward. Despite the project’s specification not requiring arguments other than a tab delimited input file, I decided that as a good practice, to include the -help optional parameter to provide users with some information about the application. While inspecting and testing the application, I noticed I had already built functions that could provide the user with some extra information about the scheduling process. Hence, I also added the optional arguments –timeline, which prints a timeline for the processes’ execution, and –report, which prints a detailed report with all instance data for processes scheduled. The user can chose to pass none, just one or both optional arguments in any order. The argument –help overrides any other argument at any location and prints program usage information.

The first parameter must be a valid input file with data in each line formatted as follows:

{processNumber (String)} tab {priority (int)} tab {burstTime (int)} tab {arrivalTime (int) } newline

Hence, a program line containing P12\t8\t18\t3\n indicates that Process number 12 has priority 8, burst time of 18 time units and arrives at time unit 3. The program output can be accessed on the system console and also on the output file outputVagner.txt created during runtime. Below are the instruction for compilation and execution.

Compiles the source code in file vagner.cpp into object file: **g++ -c vagner.cpp**

Links the object file vagner.o to executable vagnerApp.exe: **g++ -o vagnerApp.exe vagner.o**

Runs the application and produces only required output: **./vagnerApp.exe inputFile.txt**

Runs the application and prints additional execution timeline and process report

**./vagnerApp.exe –inputFile.txt –timeline -report**

Prints useful information about the application to console and file:  
**./vagnerApp.exe -help**

Note that a successful run of the program will erase the contents of the input file, hence to run the program multiple times, possibly with different arguments, the data needs to be saved into the input file again. Using the compilation and optional parameters during the execution, the application produces the set of outputs shown in the next section.

# Sample Output

In this section, the output produced by the different executions of the application is shown. When running the program, the user can run it without any optional parameter to print only the required output; run the program with parameter -report to print a detailed report about processes handled by the scheduler; run the application with –timeline to print a process execution timeline; run the application with parameter -help to print to console and file some useful information about the application. Below are the outputs generated by the application using the options, compilation, and execution steps previously described.

### Not passing any optional argument into the program: **./vagner.exe inputFile.txt**

EXECUTION DETAILS

Process P1 admitted from file into ready Queue at time 0

Process P2 admitted from file into ready Queue at time 0

Process P1 started to execute at time 0

Process P1 terminated execution at time 14

Process P2 started to execute at time 15

Process P3 admitted from file into ready Queue at time 20

\*\* Context switch at time 20: Process preempted: P2, Process admitted: P3

Process P3 started to execute at time 20

Process P4 admitted from file into ready Queue at time 25

\*\* Context switch at time 30: Process preempted: P3, Process admitted: P4

Process P4 started to execute at time 30

\*\* Context switch at time 40: Process preempted: P4, Process admitted: P3

Process P3 started to execute at time 40

Process P5 admitted from file into ready Queue at time 45

\*\* Context switch at time 45: Process preempted: P3, Process admitted: P5

Process P5 started to execute at time 45

Process P5 terminated execution at time 49

Process P3 started to execute at time 50

Process P3 terminated execution at time 54

Process P6 admitted from file into ready Queue at time 55

\*\* Context switch at time 55: Process preempted: P4, Process admitted: P6

Process P6 started to execute at time 55

Process P6 terminated execution at time 69

Process P4 started to execute at time 70

Process P4 terminated execution at time 79

Process P2 started to execute at time 80

Process P2 terminated execution at time 94

PROCESSS EXECUTION REPORT

---------------------------------------------------------------

PROCESS P1 P5 P3 P6 P4 P2

WAITING TIME 0 0 15 0 35 75

TURNAROUND TIME 15 5 35 15 55 95

---------------------------------------------------------------

### **Passing –report** into the program will produce the output in item (A.) in addition to the one below by executing the program with arguments:

### **./vagner.exe –inputFile.txt –report**

FULL PROCESS REPORT

-------------------------

Process # ........ P1

Arrival Time ..... 0

Burst Time ....... 15

Time Left ........ 0

Priority ......... 8

Waiting Time ..... 0

Turnaround Time .. 15

Running Status ... false

Process # ........ P5

Arrival Time ..... 45

Burst Time ....... 5

Time Left ........ 0

Priority ......... 5

Waiting Time ..... 0

Turnaround Time .. 5

Running Status ... false

Process # ........ P3

Arrival Time ..... 20

Burst Time ....... 20

Time Left ........ 0

Priority ......... 4

Waiting Time ..... 15

Turnaround Time .. 35

Running Status ... false

Process # ........ P6

Arrival Time ..... 55

Burst Time ....... 15

Time Left ........ 0

Priority ......... 5

Waiting Time ..... 0

Turnaround Time .. 15

Running Status ... false

Process # ........ P4

Arrival Time ..... 25

Burst Time ....... 20

Time Left ........ 0

Priority ......... 4

Waiting Time ..... 35

Turnaround Time .. 55

Running Status ... false

Process # ........ P2

Arrival Time ..... 0

Burst Time ....... 20

Time Left ........ 0

Priority ......... 3

Waiting Time ..... 75

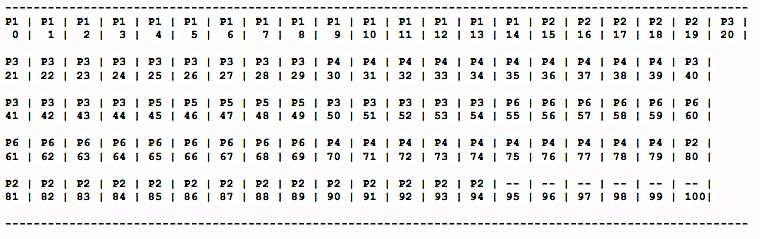
Turnaround Time .. 95

Running Status ... false

-------------------------

### **Passing –timeline** into the program will produce the output in item (A.) in addition to the one below by executing the program with arguments:

### **./vagner.exe –inputFile.txt –timeline**

****\*\*\* PROCESS EXECUTION TIMELINE \*\*\*

### **Passing -help into the program: ./vagner.exe -help:**

\*\*\* HERE IS SOME HELPFUL INFORMATION \*\*\*

\* This C++ program implements a priority based preemptive scheduler where:

- Process with highest integer value has highest priority.

- Round Robin algorithm for processes with same priority.

- Round Robin has time quantum 10 time units.

- Default program output prints execution details, waiting   
 and turnaround times for processes.

\*\* USAGE \*\*

\* The program needs a tab separated .txt file as argument 1 in the following format for each line

processnumber as (String) tab priority as (int) tab burstTime as (int) tab arrivalTime as (int) newline

i.e.: P12\t8\t18\t3\n indicates Process 12 has priority 8, burst time of 18 time units and arrives at time unit 3.

\*\* OPTIONAL ARGUMENTS \*\*

-help -> Prints this message to console.

-timeline -> Prints a timeline for the process execution.

-report -> Prints a detailed report of all processes   
 executed

# These optional arguments can be passed in any order AFTER the input text file argument.

# The parameter -help at any position overrides all other parameters.

# Any argument not listed here will cause this message to be printed.

\*\* COMPILATION AND EXECUTION \*\*

**g++ -c vagner.cpp**

compiles the source file into object file.

**g++ -o app.exe vagner.o**

links object file to executable.

**./app.exe {inputFile.txt | -help} [-help] [-timeline] [-report]**

runs program with data from inputFile.txt and optional arguments if desired.

\*\* Output is displayed onto the console and written to   
 vagnerOutput.txt \*\*

\*\* Note that a successful run will delete all contents from   
 input file \*\*

The provided outputs were base on the input file containing the following data:

P1 8 15 0

P2 3 20 0

P3 4 20 20

P4 4 20 25

P5 5 5 45

P6 5 15 55

# Conclusion

Students were required to design and implement a priority based preemptive scheduler to strengthen the understanding of scheduling algorithms for CPU allocation. To accomplish this task, I implemented additional classes and data structures to support the development of the proposed algorithm. Even with an organized and well-planned algorithm implementation, it took some additional time to debug a pointer that was not set to NULL and caused a print error. By following the detailed compilation and execution instruction, the user is capable of accessing a detailed report for executed processes, and a timeline for their execution. Implementing this project enabled me to closely analyze and better understand a system’s scheduling algorithm.