SCHEDULER IMPLEMENTATION FOR AN OPERATING SYSTEM

(Due Date: 25.11.2018 Sunday - 23:55)

CMPE322 project series is a set of projects that teaches how to construct an operating system, named *BUnix (written in institutional colors of Boğaziçi University ☺)*, from scratch. In the first project, we learn how to manage processes (tasks) in an operating system for a single CPU (with a single core). For this purpose, we expect from you to implement a round robin scheduler in C language.

Preemptive Priority Scheduling

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler chooses the tasks to work as per the priority, which is different from other types of scheduling, for example, a simple round robin. Priority scheduling involves priority assignment to every process, and processes with higher priorities are carried out first, whereas tasks with equal priorities are carried out on a first-come-first-served (FCFS) or round robin basis. In this project, we expect you to choose FCFS algorithm as the scheduling algorithm for equal priorities.

In preemptive priority scheduling, at the time of arrival of a process in the ready queue, its priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The one with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and nonpreemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job. Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.

PROCESS	PRIORITY	ARRIVAL TIME (ms)	TOTAL EXECUTION TIME (ms)
P1	1	0	10
P2	5	10	70
P3	2	20	30
P4	4	30	60
P5	3	40	50

Table 1 – Example for BUnix Operating System with five processes

The actions we expect from your scheduler are as follows:

Time	The Scheduler Action	The Ready Queue
0	P1 arrives. Since no other process is available, execute P1.	P1
10	P2 arrives and P1 has completed its execution. no other process is available at this time, execute P2.	P2
20	P3 arrives. Since the priority of P3 is higher than P2, the execution of P2 will be stopped and P3 will be scheduled on the CPU.	P3 | P2
50	During the execution of P3, two more processes, P4 and P5 arrives. Since both these newly arrived processes have a lower priority than P3, they cannot preempt P3. P3 completes its execution at this time, and P5 is scheduled due to its highest priority among the available processes.	P5 | P4 | P2
100	Meanwhile the execution of P5, all the processes got available in the ready queue. At this point, the algorithm will start behaving as nonpreemptive priority scheduling. Hence now, once all the processes get available in the ready queue, the OS just took the process with the highest priority and execute that process till completion. In this case, P4 will be scheduled and will be executed till the completion.	P4 | P2
160	After P4 is completed, the remaining process in the ready queue is P2. Hence P2 will be scheduled next.	P2
220	P2 is finished	

Table 2 – Preemptive priority scheduler

The Process Structure and the Code Files

As you learned in the lectures, a process contains the program code and its current activity (context). Normally, a program code is a binary file, which contains the instructions that are executable by the CPU. However, for the sake of simplicity, we will use text files (e.g., “code1.txt”) that represent the executable code of the processes in this project. The structure of the text file is shown in Table 3. Each row of the text file represents an instruction in which the first column presents the name of this instruction and the second column is the execution time required by the CPU to complete this instruction.

INSTRUCTION NAME	INSTRUCTION EXECUTION TIME (ms)
instr1	20
instr2	50
instr3	50
instr4	20
instr5	30
. . .	. . .
exit	10

Table 3 – A text based program code file example

Each of the rows is an atomic operation, which means if the CPU starts to execute an instruction, it cannot be interrupted to schedule another process until the execution of that instruction is completed. Assume that “instr3” is started to be executed, and another process with a higher priority arrived after 20 ms (30 more ms to complete “instr3”). The scheduler should stop the process after “instr3” is completed, add this process to the appropriate place in the ready queue, and start executing the newly arrived (with the highest priority) process. Finally, you might have noticed that the last instruction name is “exit” which means that this process should be finalized and it should be removed from the system after the exit instruction is executed.

In addition to the program code, we need to store the current activity (context) of a process to reschedule this process and continue to execute it with the CPU. Normally, the current activity of a process contains different elements such as the registers in the CPU and stack of the process. In *BUnix*, we just need to store the line number of the last executed instruction before sending this process to the ready queue.

Don’t get confused by the names of the instructions. These are not real machine codes and you will not really execute. The only important thing is that you have to calculate the execution times of these instructions correctly. So, you can stop a process, send it back to the ready queue, and run another process at the right time.

The Process Definition File

The process definition file (i.e., “definition.txt”) is a text file that provides the initial information of the processes in the system. The following table shows the structure of this file.

Process Name	Priority	Program Code File	Arrival Time (ms)
P1	4	code1	40
P2	3	code2	120
P3	6	code1	250
P4	2	code3	470
.	.	.	.
.	.	.	.
.	.	.	.

Table 4 – The process definition file structure

Each row represents a process in the system. You should have noticed that process names are unique, but more than one process may be assigned to the same program code.

The Scheduler Code

This component is a C code that parses the process definition file and schedules these processes using the preemptive priority scheduling algorithm. The design should have at least the following parts:

- The parsing of the process definition file (i.e., “definition.txt”) and the programming code files (e.g., “code1.txt”).
- A preemptive priority scheduler that maintains the queue.
- An output mechanism that generates a file to show the ready queue whenever it is changed by the algorithm. Output file also includes turnaround and waiting time for each process (you should calculate turnaround and waiting time within your code, do not calculate it manually). The output file should have the format that is shown in Table 5. Note that line number represents the row number that will be executed in the program code file.

Below you may find the expected output for the provided process definition file and programming code files:

```
[Time]:HEAD-[Ready Queue[line number]]-TAIL
0:HEAD--TAIL
40:HEAD-P1[1]-TAIL
130:HEAD-P2[1]-P1[5]-TAIL
260:HEAD-P2[7]-P1[5]-P3[1]-TAIL
390:HEAD-P1[5]-P3[1]-TAIL
470:HEAD-P4[1]-P1[8]-P3[1]-TAIL
620:HEAD-P1[8]-P3[1]-TAIL
630:HEAD-P3[1]-TAIL
810:HEAD--TAIL

Turnaround time for P1 = 590 ms
Waiting time for P1 = 410
Turnaround time for P2 = 270 ms
Waiting time for P2 = 10
```

```
Turnaround time for P3 = 560 ms
Waiting time for P3 = 380
Turnaround time for P4 = 150 ms
Waiting time for P4 = 0
```

Table 5 – A sample output file format

Development Platform

You have to implement your design in the Linux Platform with GCC/G++ Compiler. We strongly advise you to install Ubuntu which has pre-installed GCC/G++ compiling tools. We will test your code in this platform and if your compiled code is not compatible with the test platform. You might be invited for a demo, if necessary.

Provided Files

The following files are given together with the project:

- The process definition file. Note that number of processes, their priorities, and their arrival times will be modified to test your code.
- Four program code files. Note that number of instructions, their names and their execution times will be modified to test your code. Only exception is the name of the last instruction, which is always “exit”. Also, even though you don’t need “code4.txt” for the provided process definition file, you will need it for our test scenario.
- The expected output file for the provided process definition file and program code files.

Project Requirements

- Your project should have generated the expected output file for the modified process definition file and program code files. (80%)
- The source code documentation. The documentation depends on your design style, but at least having comments that explain your code is strongly advised. (10%)
- A short report (in PDF format) that describes the platform you use to develop and run your program, and a small how to run explanation. (10%)

Submission Policy

- You have to submit your source code and report to the Moodle as a zip file.
- The name of the zip file should be in [STUDENT_ID].zip format (e.g., “2015800054.zip”).
- Each student must work individually for the project. Teamwork is not accepted! Plagiarism and any other forms of cheating will have serious consequences, including getting “-100” as the project score and failing the course.
- If you have submitted to Moodle once and want to make any changes on your project, you should do it before the Moodle submission system closes. Your changes will not be accepted by e-mail. Connectivity problems to the Internet or to Moodle in the last few minutes are not valid excuses for being unable to submit. You should not risk leaving your submission to the last few minutes. After uploading to Moodle, check to make sure that your project appears there.
- Deadline is 25 November 2018, 23:55.
- Good luck.