AN OPERATING SYSTEM SCHEDULER IMPLEMENTATION

(Due Date: 17.11.2017 Friday - 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.

Round Robin Algorithm

Round robin algorithm is a scheduling algorithm in which the CPU can be interrupted by the operating system while it executes a process and is forced to switch to the execution of another process in the ready queue. The interruption occurs after the process consumes its assigned time called "time slot". Then, the scheduler sends this process to the end of the ready queue and executes the process at the head of this queue. For example, assume that the time slot is 100 milliseconds in *BUnix* and the operating system has to schedule the following processes using round robin algorithm. You should have noticed that the "total execution time" of a process is not a given parameter; it depends on the execution of the program code which is explained in the next section.

PROCESS NAME	ARRIVAL TIME (ms)	TOTAL EXECUTION TIME (ms)
P1	0	210
P2	50	120
P3	120	240
P4	450	30

Table 1 – Example for BUnix Operating System with four processes

The actions we are expect from your scheduler are:

Time	The Scheduler Action	The Ready Queue
0	Execute P1	P1
50	Add P2 to the end of the queue	P1 P2
100	P1 consumes its time slot; reschedule it and execute P2	P2 P1
120	Add P3 to the end of the queue	P2 P1 P3
200	P2 consumes its time slot; reschedule it and execute P1	P1 P3 P2
300	P1 consumes its time slot; reschedule it and execute P3	P3 P2 P1
400	P3 consumes its time slot; reschedule and execute P2	P2 P1 P3
420	P2 is finished; remove it from the queue and execute P1	P1 P3
430	P1 is finished; remove it from the queue and execute P3	P3
450	Add P4 to the end of the queue	P3 P4
530	P3 consumes its time slot; reschedule and execute P4	P4 P3
560	P4 is finished; remove it from the queue and execute P3	P3
600	P3 is finished	

Table 2 – Round robin 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 (*.code) 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)
instruction_1	20
instruction_2	50
instruction_3	50
instruction_4	20
instruction_5	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 this instruction, it cannot be interrupted to schedule another process until the execution of that instruction is completed. Assume that the "time slot" is 100 ms in *BUnix* and we start executing the process with the program code in Table 3. The scheduler should stop the process after "instruction_3" (execution time 120ms), add this process to the end of ready queue, and start executing the first process in the queue. 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 (definition.ini) 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	Program Code File	Arrival Time (ms)
P1	1.code	0
P2	3.code	20
P3	1.code	130
P4	1.code	170
. . .	. . .	. . .

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 round robin algorithm. The design should have at least the following parts:

- The parsing of the "process definition file" and the programming code files.
- The process data structure, which includes at least the programming code address (or the filename) and the last executed line information.
- A FIFO queue to implement the ready queue.
- A round robin scheduler that maintains the queue.
- An output mechanism that generates a file to show the ready queue whenever it is changed by the algorithm. The output file should have the format that is shown in Table 5.

[TIME]::HEAD-[Ready Queue]-TAIL
100::HEAD—TAIL
200::HEAD-P1-P2-TAIL
340::HEAD-P2-P3-P1-TAIL
450::HEAD-P3-P1-P2-TAIL
590::HEAD-P1-P2-P4-P3-TAIL
630::HEAD-P2-P4-P3-TAIL
790::HEAD-P4-P3-P2-TAIL
930::HEAD-P3-P2-P4-TAIL
970::HEAD-P2-P4-TAIL
1090::HEAD-P4-P2-TAIL
1130::HEAD-P2-TAIL
1230::HEAD-P2-TAIL
1240::HEAD—TAIL

Table 5 – The 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 (number of processes and their arrival times will be modified to test your code).
- Four program code files (number of instructions, their names and their execution times will be modified to test your code. Only exception is the last instruction name is always "exit" and its execution time is 10 milliseconds).
- The expected output file.
- Note: The length of the time slot is a constant value in *BUnix* operating system (100 milliseconds).

Project Requirements

- Your project should have generated the expected output file for 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. (20%)

Submission Policy

- You have to submit your source code and your documentation (if it is not only comments in your source code) to the Moodle as a zip file.
- The name of the zip file should be in [STUDENT_ID].zip format (such as 2015800054.zip).
- Deadline is 17 November 2017, 23:55.
- Good Luck.