EE441 HOMEWORK 3

Operating System Process Scheduling

Submission

- Use **Code::Blocks IDE** and choose **GNU GCC Compiler** while creating your project. Name your Project as "**eXXXXXXX_HW3**", where X's are your **7-digit student ID number**. Send the whole Project folder compressed in a rar or zip file. Name your submission as "**eXXXXXXX_ee441_hw3.rar**". You will **not** get full credit if you fail to submit your project folder as required.
- The homework must be written in **C++** (**not** in C or any other language).
- Your C++ program should follow object oriented principles, including proper class and method usage and should be correctly structured including private and public components. Your work will be graded on its correctness, efficiency, clarity and readability as a whole.
- You should insert **comments** in your source code at appropriate places without including any unnecessary detail. Comments will be graded.
- **Late submissions** are welcome, but are penalized according to the following policy:
  - 1 day late submission : HW will be evaluated out of **70**.
  - 2 days late submission : HW will be evaluated out of **50**.
  - 3 days late submission : HW will be evaluated out of **30**.
  - Later submissions : HW will NOT be evaluated.
- You should **not** call any external programs in your code.
- **Check** what you upload. Do not send corrupted, wrong files or unnecessary files.
- Programs giving compilation errors will not be evaluated!
- The homework is to be prepared **individually**. Group work is **not** allowed. Your code will be **checked** for cheating.
- The design should be your original work. However, if you partially make use of a code from the Web, give proper **reference** to the related website in your comments. Uncited use is unacceptable.
- **METU honor code is essential**. Do **not** share your code. Any kind of involvement in cheating will result in a **zero** grade, for **both** providers and receivers.

Introduction

A single core CPU can typically run a single process at a time. However, computers are required to run several processes concurrently, e.g. a web browser, a word processor and a printer spooler. Due to the architecture of CPU, such processes have to run sequentially. Deciding which process runs at a certain time instant is one of the main tasks of operating systems, known as 'process scheduling'.

The process scheduler keeps a data structure to hold the processes that will be executed, which is known as the '**Ready Queue (RQ)**'. When the user starts a new process, the process is inserted into the RQ according its **priority. The RQ stores the processes in decreasing priority order** such that the highest priority process is always at the front of the RQ. When CPU is ready, the process scheduler selects the next process to execute as the process at the front of the RQ. When the CPU finishes the execution, the process is deleted from RQ. Then, the scheduler selects the next process to be executed from the RQ.

In this homework, you are asked to implement a simple OS process scheduler.

Data Representation

For simplicity, the processes you will consider will be represented as four positive integers, `processID`, `arrivalTime`, `runTime` and `priority`. `arrivalTime` describes the time instant at which the process arrives. `runTime` describes the clock cycles required to execute the process. When a process finishes execution, your code will print its `processID` to the console. `priority` is defined similar to the practical applications such that a smaller value indicates a higher priority.

Running Your program

Your program is required to read the process list from a text file, processes.txt, which will be in the same directory as your executable. The first line of the file contains an integer, `numProcesses`, that is the number of processes in the file. The following `numProcesses` lines contain four space separated integers, `processID`, `arrivalTime`, `runTime`, `priority`, the parameters of the corresponding process. You can assume that the processes are **sorted by their `arrivalTime`**. A sample **processes.txt** file is distributed with the homework. You might use **Code1** to read processes into an array.

A pseudocode for the process scheduler is given in **Code2**. Your task is to convert this pseudocode into valid C++ and implement the classes and member functions your code requires.

CODE 1: Reading the Contents of 'processes.txt' into the Array

```
#include <fstream>

/* reading processes from file */
ifstream processesFile; /* input file stream */
processesFile.open ("processes.txt");

int numProcesses; /* number of processes */
processesFile >> numProcesses; /* read from file */

Node process[numProcesses]; /* create Node array */
int processID, arrivalTime, runTime, priority;

for(int i=0; i<numProcesses; ++i){
       /* read 4 integers from file */
       processesFile >> processID >> arrivalTime >>   runTime >> priority;
       /* construct i'th element of the array */
       process[i] = Node(processID, arrivalTime, runTime, priority);
}
processesFile.close();
/* end of reading from file */
```

CODE 2: Pseudocode of the Process Scheduler

```
time := 0 // CPU time
CPUBusy := false // is the CPU currently running a process
currentProcess := NULL // pointer to the process CPU is currently running
processIndex := 0 // index of the process awaiting insertion in RQ

while processIndex < numProcesses OR RQ is NOT empty OR CPUBusy = true
  if processIndex < numProcesses // there are processes not inserted into RQ
    if processs[processIndex].arrivalTime = time // if the process arrived now
      insert process[processIndex] into RQ // insert it into RQ
      processIndex := processIndex + 1 // next process awaits insertion

  if CPUBusy = true // CPU is running a process
    currentProcess.runTime := currentProcess.runTime - 1
                // 1 clock cycle elapsed
  if currentProcess.runTime = 0 // if it was last cycle of the process
    print currentProcess.processID // process finished execution
    CPUBusy := false // CPU is no longer running a process

  if CPUBusy = false AND RQ is NOT empty
    // CPU is not running a process but there are processes awaiting execution
    currentProcess := highest priority process in RQ // select new process
    delete currentProcess from RQ // remove it from RQ
    CPUBusy := true // CPU is now running a process

  time := time + 1 // 1 clock cycle elapsed
```

Two Different Scheduler Implementations and Comparing Their Performance

You will implement the scheduler using two different data structures to store RQ: Linked List and Binary Search Tree (BST). You will evaluate these two schedulers' performances by comparing the **total** number of nodes visited to insert **all** arrived processes into RQ (insertion), the **total** number of nodes visited to find the next processes from RQ (search) and the **total** number of nodes visited to delete the executed processes from RQ (deletion).

Linked List Scheduler

In this part, you are going to implement the scheduler explained above using a Linked List to keep RQ. The nodes should be kept **sorted** in descending order of priority. Insertion of new processes must confirm to the priority order.

Binary Search Tree Scheduler

In this part, you are going to implement the scheduler explained above using a Binary Search Tree to keep RQ. The tree does not have to be balanced.

An example operation of BST scheduler is in Fig 1.

**You can refer to the course source books for node deletion on binary search trees.*

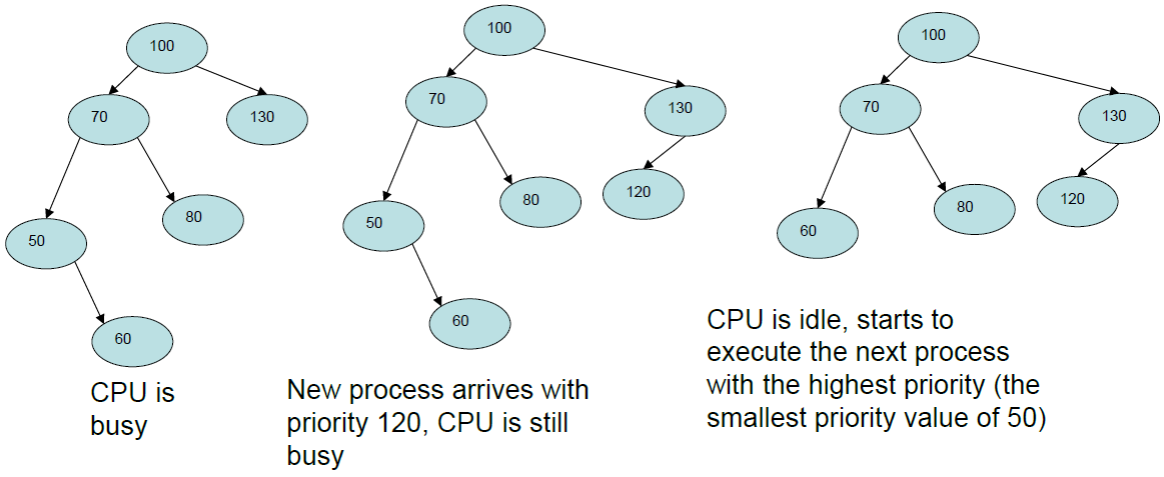

Figure 1: BST Scheduler operation example

Remarks for Performance Evaluation

- Use the sample input/output data in sampleData.zip to evaluate your algorithm's performance
- processesN.txt (N=100, 500, 1000) files are sample inputs with numProcesses =N processes
- Copy processesN.txt files, one file at a time, to your project directory, rename it as processes.txt and run both linked list and binary search tree scheduler codes

- sampleOutputN_X.txt file is the expected output for the input processesN.txt using X data structure for RQ (X=linked list, binary search tree)
- Your algorithm's performance results might be slightly different from sample output. Process execution order should not.
- You don't need to save your outputs as text files. However, you do need to run your code on sample inputs and briefly comment on your results, either in a separate text file or in your main.cpp

Program Execution Example

```
5
2 0 2 2
1 2 4 3
3 3 1 4
5 4 3 5
4 5 2 1
```

Figure 2: Input (processes.txt)

```
Linked List RQ implementation

2
1
4
3
5

5 nodes visited for searching
6 nodes visited for insertion

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

Figure 3: Output

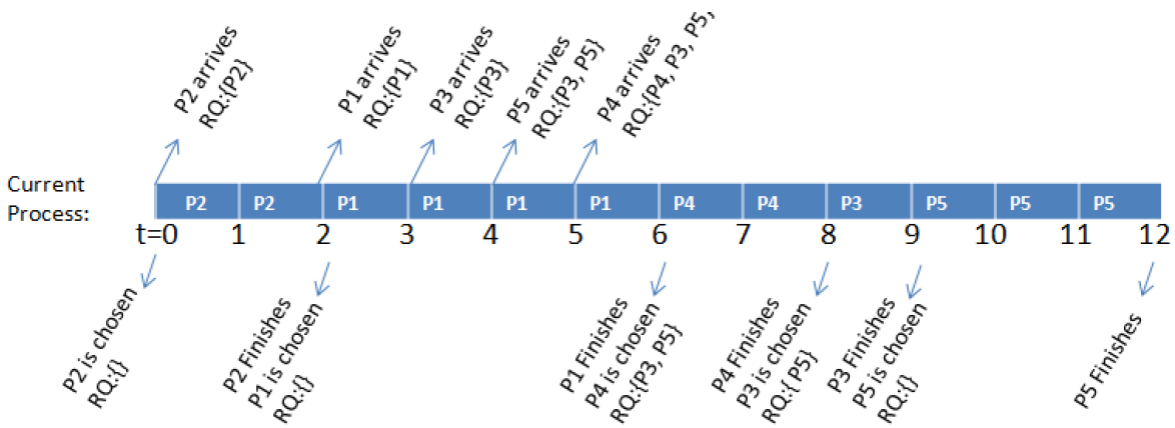

Figure 4: Explanation (CPU Timing)