CPSC 457 - Assignment 4

Due date is posted on D2L.
Individual assignment. No group work allowed.
Weight: 7% of the final grade.

Q1. Written question – SRTN scheduling (4 marks)

Process	Arrival Time	Burst Time	Process Priority
P1	0	12	High
P2	2	1	Medium
P3	3	3	Low
P4	5	1	Medium
P5	9	5	High

Table 1: 5 processes with arrival times, CPU burst times an priorities. All times are given in seconds. Only Q3 uses priorities.

Draw a Gantt chart to illustrate the execution of processes in Table 1 using shortest-remaining-time-next scheduling algorithm. Ignore process priorities for this question.

Calculate the average wait time when using SRTN scheduling.

Q2. Written question – RR scheduling (4 marks)

Draw a Gantt chart to illustrate the execution of processes in Table 1 using round-robin scheduling algorithm with quantum of 1s. Ignore process priorities for this question.

Count the number of context switches for RR scheduling with 1sec time slice.

Use FCFS to break break ties. A tie could occur if a new process arrives exactly at the same time as a currently executing process exceeds its time slice. Since both processes must be inserted into the ready, we need to decide which one to insert first. By using FCFS to break ties, the already executing job (the older of the two) will be inserted into the ready queue first, and the newly arrived job second.

Q3. Written question – MLFQ scheduling (4 marks)

For processes in Table 1, draw a gantt chart using multi-level feedback queue scheduling algorithm. Use 3 queues: Q1, Q2 and Q3, where Q1 is a high priority queue, scheduled using RR with 2s quantum, Q2 is medium priority queue, scheduled using RR with 4s quantum, and Q3 is low priority queue scheduled using FCFS scheduling.

When P1 and P5 arrive, they go straight to Q1. When P2 and P4 arrive, they go straight to Q2. When P3 arrives it goes straight to Q3.

Processes will move down in priority if they exceed the corresponding time slice - for example P1 starts in Q1, but it could eventually end up in Q3.

Processes cannot move up from lower priority queue to higher priority.

In general, a running process belonging to a lower priority queue could be preempted by a process arriving into a higher priority queue. For this question you will ignore such source of preemption.

Use FCFS to break ties.

Q4. Programming question – scheduler simulation (20 marks)

Write a program `scheduler.c` or `scheduler.cpp` that simulates two different scheduling algorithms: non-preemptive shortest-job-first and preemptive round-robin.

Command line arguments

Your program will accept 2 or 3 command-line arguments, depending on the scheduling algorithm:

1. The name of the configuration file. Your program will read the configuration file to obtain the description of processes.
2. The name of the scheduling algorithm: 'RR' for round-robin or 'SJF' for shortest job first.
3. The time quantum for the RR scheduling algorithm. If 2nd argument is SJF, the 3rd argument will not be specified.

For example, the command line below should invoke your simulator on file `config.txt` using RR scheduler and time slice of 3:

```
$ ./scheduler config.txt RR 3
```

To run the simulation on the same file using SJF schedule, you would start your simulator like this:

```
$ ./scheduler config.txt SJF
```

If the user does not provide correct arguments, you should print out an informative error message and abort the program. For example, if the user specifies time-slice for SJF, you should report this as an error and abort the program. You must support the uppercase strings "RR" and "SJF", but if you wish you can also upport lower case versions.

Configuration file

The configuration file contains description of processes that your simulator will schedule. Each process will be described in the file on a seprate line. Each line contains 2 non-negative integers: the first one denotes the arrival time of the process, and the second one the cpu burst length.A sample configuration file `config.txt` is below:

```
1 10
3 5
5 3
```

This file contains information about 3 processes: P0, P1 and P2. The 2nd line "3 5" means that process P1 arrives at time 3 and it has a burst of 5 seconds.

You may make the following assumptions about the configuration file:

- The processes are sorted by their arrival time, in ascending order.
- For output purposes you need to give the processes names in the format ‘Px’, where x is the process ID. Assign IDs consecutively starting from 0.
- All processes are 100% CPU-bound, i.e., a process will never be in the WAITING state.
- There will be between 0 and 30 processes.
- Process arrival time will be in the range [0…1000], and bursts in the range [1…100].

Simulation

You can use the simualtion loop pseudocode presented during lectures to write your solution. In your simulation, you should advance the ‘current time’ of the simulation by 1 unit at a time.

Simulation output

Your program will print the state of every processes at every simulated time step to standard output. Your program will also report the time that each process spent waiting in the ready queue, as well as the overall average wait time. Example ouput of your simulator is below. Please make sure your program output matches it as closely as possible.

```
$ ./scheduler config.txt RR 3
Time P0 P1 P2
------------------------------------------------------------
   0
   1  #
   2  #
   3  #  .
   4  .  #
   5  .  #  .
   6  .  #  .
   7  #  .  .
   8  #  .  .
   9  #  .  .
  10  .  .  #
  11  .  .  #
  12  .  .  #
  13  .  #
  14  .  #
  15  #
  16  #
  17  #
  18  #
------------------------------------------------------------
P0  waited 8.000 sec.
P1  waited 7.000 sec.
P2  waited 5.000 sec.
Average waiting time = 6.667 sec.
```

The output has two parts: a table discribing the state of each process for every simulation time step, followed by a summary, which includes the wait time for each process and the averaget wait time for all processes.

The first column in the table is the simulation time. There is also one column for each process to describe its state for the given simulation time. Use "." to denote READY state, "#" to denote RUNNING state, and a empty space " " to denote a process that has not yet arrived or a finished process. Make sure the output of your program is nicely aligned like the example above.

Q5. Written question (3 marks)

In a system consisting of 5 processes and 3 resource types, the current allocation, requests and available resources are as follows:

Process	Allocation	Request	Available
P1	0 0 1	1 2 1	0 1 0
P2	0 1 0	0 1 1	
P3	1 1 2	1 2 2	
P4	1 0 0	0 2 1	
P5	0 0 1	0 1 0	

Is this system in a deadlock? If your answer is yes, give a partial execution sequence. If your answer is no, give a full execution sequence.

Q6. Programming question (20 marks)

For this question you will write a program `deadlock.c` or `deadlock.cpp` that examines multiple system states and for each of them determines whether the system is in a deadlock, and if it is, which processes are deadlocked.

Each system state will consist of some number of processes and resources. You will assume a single instance per resource type. Hint: you should implement a cycle-detection algorithm.

Command line

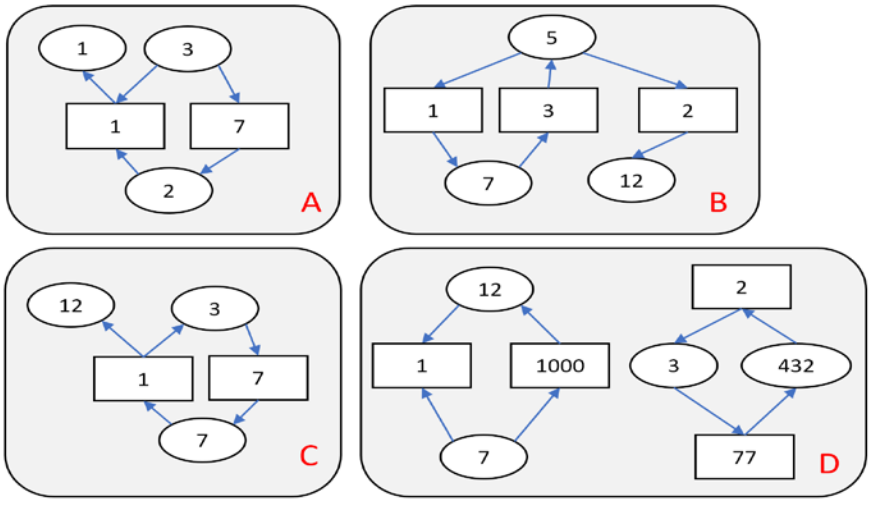
Your program will take no command line arguments. Your program will read the standard input to get state descriptions.

Input

Your program will read the descriptions of each system state from standard input. The input will be line-based, with 3 types of lines: one represeting an assignment edge, one representing a request edge and one representing an end of state description.

- a line `N -> M` represents a request edge, i.e. process N is requesting resource M;
- a line `N <- M` represents an assignment edge, i.e. process N holds resource M;
- a line that starts with `#` represents the end of state description.
- "N" and "M" above will be non-negative integers.

As an example, consider the following 4 system states:



The 4 system states above would be represented by the input below, and the output your program should produce on this input is shown on the right.

Input:		Output:
1 <- 1 3 -> 1 3 -> 7 2 -> 1 2 <- 7 # end of A 5 -> 1 5 <- 3 5 -> 2 7 <- 1 7 -> 3 12 <- 2 # end of B	12 <- 1 3 <- 1 3 -> 7 7 -> 1 7 <- 7 # end of C 12 -> 1 12 <- 1000 7 -> 1 7 -> 1000 3 -> 77 432 <- 77 432 -> 2 3 <- 2	Deadlocked processes: none Deadlocked processes: 5 7 Deadlocked processes: 3 7 Deadlocked processes: 3 432

Notice there is no explicit end of state line `#` at the end of the input above, as it is implied by the end-of-file. You may assume the following limits on input:

- process numbers will be in range [0 … 100000];
- resource numbers will be in range [0 … 100000];
- number of edges per state description will be in range [1 … 100000];
- number of states per input will be in range [1 … 20].

Your solution must be efficient enough to be able run on any input within the above limits in less than 10 seconds.

Output

Your program will write the results to standard output. For each state it reads, it will print out how the list of processes involved in the deadlock, sorted in ascending order. If there is no deadlock, it should print out the word ‘None’. Your output should match the sample output above exactly.

Notes

The overall structure of your program should look something like this:

```
loop:
    set graph to empty
    loop:
        read line from stdin
        if line starts with ‘#’, or EOF reached:
            compute, sort and display deadlocked processes from graph
            break
        else:
            convert line to an edge
            add edge to graph
    if EOF was encountered:
        break
```

Submission

Submit 3 files to D2L:

`report.[pdf|txt]`	answers to all written questions
`scheduler.[c|cpp]`	solution to the first programming question in C or C++
`deadlock.[c|cpp]`	solution to the second programming question in C or C++

General information about all assignments:

- All assignments must be submitted before the due date listed on the assignment. Late assignments or components of assignments will not be accepted for marking without approval for an extension beforehand. What you have submitted in D2L as of the due date is what will be marked.
- Extensions may be granted for reasonable cases, but only by the course instructor, and only with the receipt of the appropriate documentation (e.g. a doctor's note). Typical examples of reasonable cases for an extension include: illness or a death in the family. Cases where extensions will not be granted include situations that are typical of student life, such as having multiple due dates, work commitments, etc. Forgetting to hand in your assignment on time is not a valid reason for getting an extension.
- After you submit your work to D2L, make sure that you check the content of your submission. It's your responsibility to do this, so make sure that you submit your assignment with enough

time before it is due so that you can double-check your upload, and possibly re-upload the assignment.

- All assignments must include contact information, including full name, student ID and tutorial section, at the very top of each file submitted.
- Assignments must reflect individual work. Group work is not allowed in this class nor can you copy the work of others. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html.
- You can and should submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It's better to submit incomplete work for a chance of getting partial marks, than not to submit anything.
- Assignments will be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA then you can contact your instructor.