Birla Institute of Technology & Science, Pilani Hyderabad Campus

CS F372: Operating Systems

Assignment 2 - (Scheduling, Synchronization and Deadlocks)

Assigned: 15/10/2018 Date of Submission: 03/11/2018

Total Marks: 35

Problem 1: Scheduling: Discrete event simulation to analyze performance of different CPU scheduling algorithms.

A discrete-event simulation (DES) models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next. (This contrasts with continuous simulation in which the simulation continuously tracks the system dynamics over time).

You will simulate the scheduling of a set of simultaneous processes with different arrival time and different CPU bursts derived from an exponential distribution. Also, you will compare their performances using any suitable metrics studied in the course.

Background: **Discrete Event Simulation**
When conducting a simulation in which events occur at different times, you should use the below approach:

Set up a loop, jump forward in time with each iteration to whenever the next meaningful event occurs. Some time steps may be small (even 0 if two or more things happen at the same time) and some may be large. Some events may trigger other events, which are then put on the schedule to be processed when their time comes. This approach is called Discrete Event Simulation.

Key to a DES are event objects, which are stored in an **event heap**, so that the next Event to occur can be readily acquired. The ordering of the remainder of the events in the heap is irrelevant. It is also key that new events can be added to the heap at any time. (It would also be convenient in some cases to be able to remove events from the heap if they get cancelled for some reason. Unfortunately, the data structures you will be using for this assignment do not allow the random removal of events from the heap, so they will just have to be marked as cancelled, and then discarded whenever they are removed from the heap in the normal course of the simulation.

The general algorithm of a DES is a while loop which continues as long as there are events remaining in the heap to be processed. (The heap must obviously be populated with at least one initial event before the while loop commences). The while loop extracts the next event from the event heap, determines what type of event it is, and processes it accordingly (generally with a switch on the event type). Depending on the type of event and other conditions, new (future) events may be generated, which are then added into the event heap to be processed when their time arrives.

There is also a “time” variable that is updated whenever an event is extracted from the event heap. (Since each event has an associated time at which it occurs, we can update the time variable to match the time of the event which was just extracted from the heap)

You may use the generic pseudocode for the DES as given below:

```
while (heap not empty) {
    1. extract an Event from the heap;
    2. update time to match the Event;
    3. switch (type of Event) {
            process this event, possibly adding new Events to the heap;
    } // switch
} // while
```

Process statistics collected during Event processing & report.

You have to simulate 2 scheduling algorithms using DES:

- First Come First Serve (FCFS)
- Multilevel Feedback Queue

and compare their average wait times (AWTs).

Below data structures may be used:

```
typedef struct Process {
int pid;
char state;
int arrival_time;
int cpu_burst;
int wait_time;
char* scheduling_policy;
int time_quantum;
bool preemption;
// add other fields which you feel are necessary
} Process;
```

Define the event struct as:

```
typedef struct Event {
enum eventType { Arrival, CPUburstCompletion, TimerExpired };
double time; // time units since the start of the simulation
// add other fields which you feel are necessary
} Event;
```

- Implement the **ready queue** using heap. For example, for FCFS Scheduling, one might implement a min-heap with key as arrival time of the process. Ready queue would contain process ids (PIDs) of the processes.
    - Processes are stored in a global **process table** which is an array of Process. You can also use the index of a process in process table as the PID.
    - **CPU**: global integer, contains PID of currently running process and -1 if it is idle

You need to process following events:

The default case – Prints all the information about the Event.

The Arrival case – Read in data corresponding to the arriving process, and create a new entry in the process table. If the CPU is currently idle, the new process can be placed in the running state directly on the CPU, and a CPUburstCompletion event created and added to the event queue. Otherwise, set the process state to ready and add it to the ready queue. (Assuming there is no preemption). In case if preemption bit is set add Preemption event to event heap. Before completing this case read in the time of the next process arrival, and add a new Arrival event to the event queue.

A process would 'arrive' if current simulation time > arrival time of process. If multiple processes have arrival time < current time, pick one with minimum arrival time.

The CPU Burst Completion case – If this was the last of the CPU bursts for this process, its state can be set to be terminated. If the ready queue is not empty, then a new process is moved from the ready queue to the CPU and a new CPUburstCompletion event added to the event queue.

Timer Expiration – If a scheduling algorithm is being simulated which limits the maximum time slice that a process may receive, then a timer expiration event gets added to the event heap whenever a process is loaded onto the CPU. If the timer expires before the process completes its CPU burst, then the timer expiration event triggers a context switch, moving that process from the CPU to the ready queue and selecting a new process from the ready queue to run next. If the process finishes its CPU burst before the timer expires, then the timer expiration event is a null event, and can just be discarded when it exits the event heap.

Break ties by giving higher priority to process with smaller PID in case there are multiple processes with same key.

In the end, when event heap is empty, print:

- average wait time (AWT) of all processes.

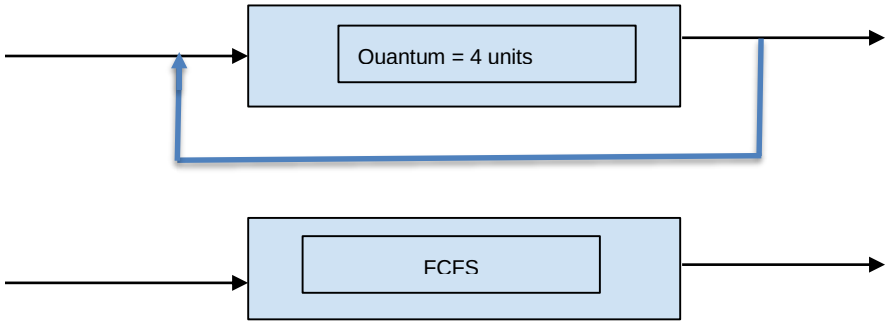
as stored and calculated from processes in the process table. You must also print to console when the following conditions happen for a process:

- it's time-quantum expires or,
- it finishes i.e. CPU burst completes or,
- selected from ready queue.
- gets inserted to ready queue or,
- when it is running.

Scheduling algorithms to be implemented:

1. **First Come First Server (Assuming that there is no Convoy effect created by your DES)**

2. **Multilevel queues**
   - **Two** queues as shown in below figure.
   - Bottom queue with **FCFS scheduling** and top queue with **RR scheduling** policy.
   - A process enters into one of the queues based on its type. For example, you could consider processes with CPU burst of 8 time units or less entering into top queue and all other processes entering into bottom queue.
   - Ignore Context switch time.



Data Generation:
You can generate the process arrival time and burst time data using following python snippet:

```
import numpy as np
# exp_dist <-> f(x, beta) = (1/beta) * exp(-(1/beta) * x)

beta = 10
process_count = 20
a=np.round(np.random.exponential(scale=beta, size=(process_count,2)))
np.savetxt("process.csv", a, delimiter=",",fmt="%i")
```

Assign PID to each row in the file while reading it. First column corresponds to arrival time and second column is the CPU Burst.

[8 Marks + 12 Marks]

Problem 2: Synchronization: Assume that in the system there are 8 processes. Each of the 8 processes need some resources to run, and there are 4 types of resources, i.e. A, B, C and D. Now, to run, each process requires different type of resources which are listed below:

Process	Resources
1	**A, B, C**
2	**B, C, D**
3	**A, C, D**
4	**A, B, D**
5	**A**
6	**B**
7	**C**
8	**D**

At an instant (say, t = 0), all the processes running on the CPU completed their execution and operating system currently has some amount of each resource (say 5 of A, 5 of B, 5 of C and 5 of D) available. Other than that, the OS also collects these resources from the processes which no longer need it, with some probabilities as:

Resource	Probability of Collection
A	**2/3**
B	**3/4**
C	**3/5**
D	**2/3**

When the OS collects a resource, its count in the pool increases by 1. When a process uses a resource, its count in the pool decreases by 1. **OS will try to collect resources only when a process finishes**.

Now, implement this synchronization problem using semaphores. You must ensure that:

- every process runs exactly k times;
- the same instance of a resource is not used by multiple processes at the same time;
- a process (say Process 4) can start again only if Process 4 is not already running;
- since there is only one CPU more than 1 process cannot run simultaneously;
- deadlocks don't happen - say at a moment quantity of A - 1, B - 1, C - 1, D - 1 and Process 1 acquires lock on A, 2 on B, D and 3 on C => deadlock.

Input:

```
k
a b c d
```

First line contains 'k', the number of times each process should run.
Next line contains a, b, c, d, the initial quantity of the resources.

Output:

On the console, you must print

- the resources (A/B/C/D) every time they get collected,
- when a process starts
- when the process is running,
- and when it finishes, the resources it is waiting upon.

You must print this information sequentially as and when it happens.

[10 Marks]

Problem 3: Deadlocks (no coding): Prepare one-page report on:

"A Banker's Solution for Deadlock Avoidance in FMS with Flexible Routing and Multiresource States", J. Ezpeleta, et al. IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 18, NO. 4, AUGUST 2002. OR, The Theory of Deadlock Avoidance via Discrete Control, Yin Wang, et al. POPL'09, ACM SIGPLAN.

[05 Marks]

Submission Instructions: Pl maintain same grouping as previous assignment. All your programs must run over Ubuntu machines. Submit source files. Put all your deliverables into a tar file (like, *f20160xxx.tar*) and send it to p20150005@hyderabad.bits-pilani.ac.in as a mail attachment. Copied codes will be awarded zero marks. This is the last assignment and tentative date for demo of this assignment is 1st week of Nov 2018.

For any queries, you may contact I/C or the following:

- Rajesh Kumar Shrivastava <p20150005@hyderabad.bits-pilani.ac.in>
- Sanket Mishra <p20150408@hyderabad.bits-pilani.ac.in>
- Bhavesh G <f20150116@hyderabad.bits-pilani.ac.in>