Assignment #4: Simulation of a CPU Scheduler

CS201 Fall 2019
Part I, 25 points, due Friday, Oct 18th, 11:59 pm
Part II, 10 points, due Friday, Oct. 25th, 11:59 pm

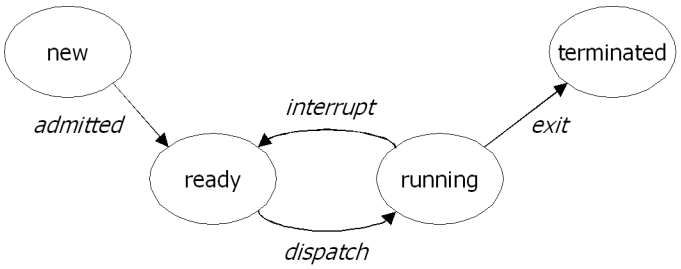
1 Overview

You'll create a discretgenExpRande-event simulator to model the creation and scheduling of CPU processes to a single-CPU system. Processes will move from the ready state to the running state (and possibly back). You'll model three scheduling algorithms:

- first-come-first-served (FCFS)
- shortest-job first (SJF)
- round robin (RR)

Each process will have a burst time, which represents the total amount of cpu time *i.e.,* time spent in the running state. You'll keep track of how long a process has to wait (in the ready state).

You'll model this part of the system:



In FCFS and SJF, a process will stay in the running state for the duration of its burst time. After that, it will leave the system.

In RR, a process will move from running to ready when the quantum expires. Each time this happens, you will subtract the length of the quantum from the burst time from the process, so that eventually the process does finish running and leaves the system.

2 Discrete-Event Simulation

In a discrete-event simulation (DES), everything that happens in the system happens in response to an event. All events in the system have an entry in the event queue, which is a priority queue.

From Wikipedia:

A discrete-event simulation 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.

Also, no event in the past can affect an event in the future, and so we really can go from event to event.

The event queue is a priority queue. The priority is determined by the scheduling algorithm that is in use. For example: in SJF, the priority is the burst time: processes with smaller burst times have higher priority and thus get to the CPU sooner. For FCFS, the priority is the time that the process was submitted.

3 The System

There are four different types of events in the system: PROCESS_SUBMITTED, PROCESS_STARTS, PROCESS_ENDS, and TIMESLICE_EXPIRES.

The program itself will consist of an event loop:

```
currentTime = getMinPriority(EPQ);
event = dequeue(EPQ);
while (event != NULL) {
   handleEvent(event);
   currentTime = getMinPriority(EPQ);
   event = dequeue(EPQ);
}
```

Each entry in EPQ, the event queue, will correspond to an event. Each event has an eventType and a pointer to the process for which this event will occur.

You can choose to handle the events in a separate function (as I show) or handle them directly inside the while loop.

Use my DESexample as a framework for your program. It shows a simple DES, with three event types. The source code for this is DESexample.c, in the course gitlab repo, under Examples. You can use your implementation of the priority queue from Assignment #3 or else my pqueue files.

3.1 Data Structures

Use this structure for a process in the system:

```
typedef struct {
   int pid;            // unique identifier for this process
   int burstTime;      // total amout of CPU time this process will need
   int waitTime;       // total amount of time this process has spent waiting
   int numPreemptions; // for RR, the # times this process was descheduled
   int lastTime;       // to keep track of a specific value of currentTime
} Process;
```

The lastTime field will let you keep track of a specific time in the past, so that you can compute some quantity. For example, when a process is submitted, put the value of currentTime in this field. Then, when the process starts at some point in the future, you'll be able to compute the time that the process has been waiting: `currentTime - process->lastTime`.

Use this structure for an event:

```
typedef struct {
   EventType eventType;
   Process *process;
} Event;
```

And use this typedef:

```
typedef enum EventTypeEnum {
   PROCESS_SUBMITTED,
   PROCESS_STARTS,
   PROCESS_ENDS,
   PROCESS_TIMESLICE_EXPIRES
} EventType;
```

3.2 The Priority Queues

There are two priority queues in the system: the event priority queue and the ready queue. Both are instances of PQueue, from Assignment #3. You can use your own implementation or use mine (pqueue.jhibbele.c, pqueue.jhibbele.h; both are in gitlab).

The event priority queue holds instances of type Event. The ready queue holds instances of type Process. When a process comes into the ready state but the CPU is busy, then that process goes into the CPU queue. The ordering of the CPU queue is determined by the scheduling algorithm.

3.3 Processes

A process will keep track of its wait time: this is the total time waiting for the CPU. For FCFS and SJF, the wait time is the difference of the time when the process is submitted and the time when the process starts. For RR, it's more complicated. Initially, it's the difference in the PROCESS_SUBMITTED and PROCESS_STARTS times, but then every time the timeslice for that process expires, the process will have to go back to the ready state; it will then incur additional waiting time until it starts again.

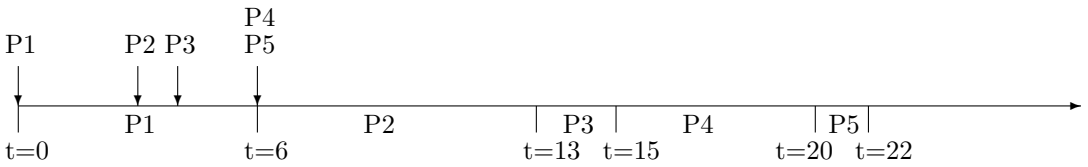
For RR, every time that the quantum expires for a process and the process still has burstTime > 0, increment the numPreemptions field for that process.

4 Testing Your System

Use these five processes:

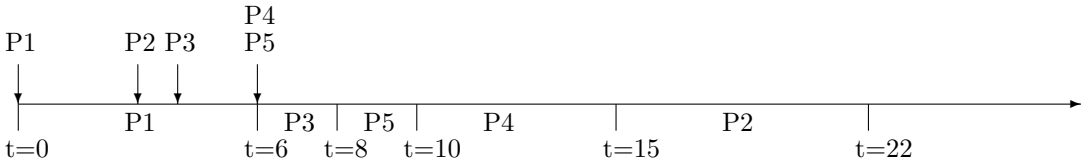
Process	Submitted at	Burst time
pid=1	t=0	6 ms
pid=2	t=3	7 ms
pid=3	t=4	2 ms
pid=4	t=6	5 ms
pid=5	t=6	2 ms

4.1 FCFS results



Mean wait time = 7 ms.

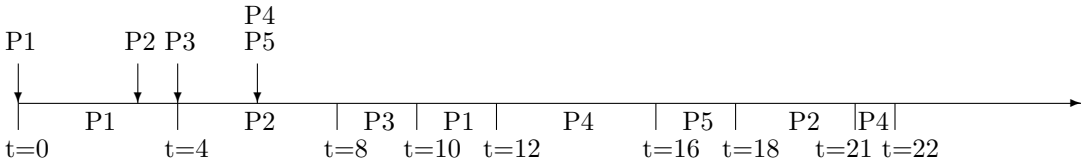
4.2 SJF results



Mean wait time = 4 ms.

4.3 RR results

Quantum = 4 ms.



Here are some details and comments about the RR results:

- at t=4, P2 starts because it's the only process in the ready queue
- at t=8, the ready queue is P3, P1, P4, P5, P2
- at t=16, the ready queue is P5, P2, P4
- P1 wait time = 6
- P2 wait time = 1 + 10 = 11
- P3 wait time = 4
- P4 wait time = 6 + 5 = 11
- P5 wait time = 10

5 What to Do

5.1 Individual or Partners

You can work individually or with a partner. If you work with a partner, then I will ask each partner to fill out a form describing the contributions of each person.

5.2 Part One: FCFS and SJF

Due Friday, Oct. 18th, at 11:59 pm. I'll accept late submissions until Monday, Oct. 21st, at 11:59pm.

Create these functions to help you test your system:

```
Process *createProcesses()
```

This will create the five processes shown in the table above and will return them in an array.

```
void enqueueProcesses(PQueueNode **eventPQueue, Process *processes, int numProcesses)
```

This will enqueue the processes in the event queue at the times specified in the table.

```
void runSimulation(int schedulerType, int quantum, PQueueNode *eventPQueue)
```

This will run the simulation. Print out information about each event and process, as shown below. You should see the following, for FCFS and SJF

5.2.1 FCFS

t = 0	PROCESS_SUBMITTED	pid = 1	
t = 0	PROCESS_STARTS	pid = 1	
t = 3	PROCESS_SUBMITTED	pid = 2	
t = 4	PROCESS_SUBMITTED	pid = 3	
t = 6	PROCESS_SUBMITTED	pid = 4	
t = 6	PROCESS_SUBMITTED	pid = 5	
t = 6	PROCESS_ENDS	pid = 1	waitTime = 0
t = 6	PROCESS_STARTS	pid = 2	
t = 13	PROCESS_ENDS	pid = 2	waitTime = 3
t = 13	PROCESS_STARTS	pid = 3	
t = 15	PROCESS_ENDS	pid = 3	waitTime = 9
t = 15	PROCESS_STARTS	pid = 4	
t = 20	PROCESS_ENDS	pid = 4	waitTime = 9
t = 20	PROCESS_STARTS	pid = 5	
t = 22	PROCESS_ENDS	pid = 5	waitTime = 14

Mean wait time = 7 ms.

5.2.2 SJF

t = 0	PROCESS_SUBMITTED	pid = 1	
t = 0	PROCESS_STARTS	pid = 1	
t = 3	PROCESS_SUBMITTED	pid = 2	
t = 4	PROCESS_SUBMITTED	pid = 3	
t = 6	PROCESS_SUBMITTED	pid = 4	
t = 6	PROCESS_SUBMITTED	pid = 5	
t = 6	PROCESS_ENDS	pid = 1	waitTime = 0
t = 6	PROCESS_STARTS	pid = 3	
t = 8	PROCESS_ENDS	pid = 3	waitTime = 2
t = 8	PROCESS_STARTS	pid = 5	
t = 10	PROCESS_ENDS	pid = 5	waitTime = 2
t = 10	PROCESS_STARTS	pid = 4	
t = 15	PROCESS_ENDS	pid = 4	waitTime = 4
t = 15	PROCESS_STARTS	pid = 2	
t = 22	PROCESS_ENDS	pid = 2	waitTime = 12

Mean wait time = 4 ms.

5.3 Part Two

Due Friday, Oct. 25th, at 11:59pm.

Implement round-robin scheduling. With a time quantum set to 4 ms, you should see these results:

t = 0	PROCESS_SUBMITTED	pid = 1	
t = 0	PROCESS_STARTS	pid = 1	
t = 3	PROCESS_SUBMITTED	pid = 2	
t = 4	PROCESS_SUBMITTED	pid = 3	
t = 4	PROCESS_TIMESLICE_EXPIRES	pid = 1	
t = 4	PROCESS_STARTS	pid = 2	
t = 6	PROCESS_SUBMITTED	pid = 4	
t = 6	PROCESS_SUBMITTED	pid = 5	
t = 8	PROCESS_TIMESLICE_EXPIRES	pid = 2	
t = 8	PROCESS_STARTS	pid = 3	
t = 10	PROCESS_ENDS	pid = 3	waitTime = 4
t = 10	PROCESS_STARTS	pid = 1	
t = 12	PROCESS_ENDS	pid = 1	waitTime = 6
t = 12	PROCESS_STARTS	pid = 4	
t = 16	PROCESS_TIMESLICE_EXPIRES	pid = 4	
t = 16	PROCESS_STARTS	pid = 5	
t = 18	PROCESS_ENDS	pid = 5	waitTime = 10
t = 18	PROCESS_STARTS	pid = 2	
t = 21	PROCESS_ENDS	pid = 2	waitTime = 11
t = 21	PROCESS_STARTS	pid = 4	
t = 22	PROCESS_ENDS	pid = 4	waitTime = 11

Mean wait time = 8.4 ms.

In addition, implement the following functions:

```
Process *createRandomProcesses(int numProcesses, double meanBurstTime);

void enqueueRandomProcesses(int numProcesses,  PQueueNode **eventQueue,
                            Process *processes, double meanIAT);
```

Assign the burst time for each process using an exponential distribution with the mean `meanBurstTime`.

Generate a random number for the time delta between the submit times of consecutive processes. In other words, if process P_i is submitted at t_i, then process P_{i+1} will be submitted at $t_i + \Delta_i$, where Δ_i is a exponentially distributed random number with mean specified by the parameter `meanIAT` (= mean interarrival time). Here's a code snippet for creating the processes:

```
for (i=0; i<numProcesses; ++i) {
    processes[i].pid = i+1; // start the process IDs at 1 instead of zero
    processes[i].waitTime = 0;
    processes[i].lastTime = 0;
    processes[i].numPreemptions = 0;
    processes[i].burstTime = (int) genExpRand(burstTimeMean);
}
```

And here's a code snippet for creating the PROCESS_SUBMITTED events:

```
t = 0;
for (i=0; i<numProcesses; ++i) {
  event = (Event *) malloc(sizeof(Event));
  memset(event, 0, sizeof(Event));
  event->eventType = PROCESS_SUBMITTED;
  event->process = &processes[i];
  enqueue(eventQueue, t, event);
  t = t + (int) genExpRand(meanIAT);
}
```

And here's a function for generating an exponentially-distributed random number with mean *mean*:

```
int genExpRand(double mean) {
  double r, t;
  int rtnval;
  r = drand48();
  t = -log(1-r) * mean;
  rtnval = (int) floor(t);
  if (rtnval == 0)
    rtnval = 1;
  return(rtnval);
}
```

This is in github also. To use this, you must #include <math.h> and <stdlib.h>.

To set a seed for the random-number generator, do this for example:

```
seed48(1);
```

6 Experiments

Create 1000 random processes with meanBurstTime = 25.0 and meanIAT = 25.0, in both cases using an exponential distribution. Use seed value = 1. Run a simulation using SJF and a second simulation using FCFS. Print the mean waiting time and standard deviation for both runs. Do this five times–using a different seed each time, but using the same seed value for the FCFS and SJF runs–and record the results for each of the ten runs in a text document that you'll submit.

Make a table like this:

	SJF		FCFS	
seed	mwt	stddev	mwt	stddev
1				
2				
3				
4				
5				

6.1 Formulas

The mean value μ of n waiting times $w_1, w_2, \ldots, w_n$:

$$\mu = \frac{1}{n} \sum_{1}^{n} w_i$$

The standard deviation σ of those values:

$$\sigma = \sqrt{\frac{1}{n} \sum_{1}^{n} (w_i - \mu)^2}$$

7 What to Submit

Name your file scheduler.netid.c. If you have a .h file, name it scheduler.netid.h. Also submit the results of your experiments.

8 Graduate Students and Extra Credit for Undergraduates

8.1 Part Three

Graduate students, and undergraduates for 3 extra-credit points: create 1000 random processes with meanBurstTime = 25.0 and meanIAT = 25.0. Do a RR simulation and vary the time quantum from 1 to 10, in increments of 1. Plot the mean waiting time as a function of the quantum. Also plot the total number of preemptions as a function of the quantum. In reality, the time for a context switch is not zero. Do some experiments to understand the effect of this overhead. Set the context time as a fraction of the mean burst time (e.g., 10%), and investigate the behavior of RR.

8.2 Part Four

Also, graduate students, and undergraduates for 3 extra-credit points, add a second CPU. There will still be a single ready queue, but let a process will go either to CPU #1 or CPU #2, depending on which CPU is idle. Do five runs of FCFS and SJF, using the same seed value for both but different seed values for each experiment, with a single CPU and a second CPU and show mean wait time and standard deviation for each.

9 Other Information

Here's the gitlab repo for the class:

https://gitlab.uvm.edu/Jason.Hibbeler/ForStudents/tree/master/CS201/