CS3204 Operating Systems Arthur

Programming Assignment 2

You are to write a program that simulates the movement of jobs starting at the hold queue, going through the various process states, and ending up in a termination list. You will be required to implement the following "queues":

HOLD Queue:	jobs will initially be placed in the hold queue (FIFO within SJN order) and moved to the ready queue at the appropriate time (see psuedo code).
READY Queue:	the ready queue holds processes for which resources have been allocated and that are ready to run.
RUNNING State:	the running state is a unique state that holds one process, i.e. the process that is currently executing.
BLOCKED List:	the blocked list holds processes that have an I/O request pending. When an I/O request is satisfied for a process on the blocked list, that process is moved to the end of the ready queue.
TERMINATION List:	when a process completes, it will be placed on the termination list.

You will implement a Shortest Job Next (SJN) job scheduler for the hold queue (FIFO on ties), and a ROUND ROBIN process scheduling algorithm to move processes to, from and among the principal process states (ready, running and blocked).

Input to your program will be in the form of independent ascii text files, each containing combinations of six (6) distinct line formats distinguished by the first integer on each line. These lines will be used to build an event list using the associated event times. The lines will have the following format:

(0) the first line will always be the one corresponding to initial system parameters. It will have a '0' in position 1, followed by memory capacity (integer), system tape resources (integer), initial system clock value (real), system time quantum (real), context switch overhead time (real).

(1) the "job arrival" line will have a '1' in position 1, followed by the job/process name (char), event time (real), memory requirements (integer), tape requirements (integer), and required cpu time (real).

(2) the "process complete" line will have a '2' in position 1, followed by the process name (char) and event time (real). This type of line signifies that the indicated process is to be terminated regardless of the remaining required CPU time. Note: a process can only be terminated while in the running state.

(3) the "i/o request" line will have a '3' in position 1, followed by the name of the process (char) and the event time (real). This type of line signifies that the indicated process is to be blocked

for I/O (i.e. moved to the blocked state) and remain there until an "i/o complete" event is encountered for that process. Note: a process can only be blocked while in the running state.

(4) the "i/o complete" line will have a '4' in position 1, followed by the name of the process (char) and the event time (real). The indicated process is to be removed from blocked list and to be put on the ready queue.

(5) the "print stats" line will have a '5' in position 1, followed by an event time (real). When this event occurs you will print out:
   (a) the current system status:
      - current memory available,
      - current number of tape resources available, and
      - current time on the system clock,
      - current event list entries (event, process name, event time)
   (b) for each queue/list/state (hold, ready, run, block, termination) and each entry in that queue/list/state:
      - process name,
      - memory and tape resources currently allocated to it,
      - arrival, start, finish times, and
      - estimated time required, time actually used, and estimated time left, and
      - turnaround time (Ti) for any processes is in termination queue (finish - arrival, where arrival time is when the process arrived at the Hold queue.
   (c) summary statistics:
      - if applicable, the average turn around (T) and weighted turnaround time (W) using actual CPU time used by each process,
      - the total time spent in context switching and process movement,
      - the total CPU time used by all processes,
      - how much time the CPU spent idle,

For convenience, the system clock is incremented by a context switch time whenever a job/process migrates between any two states/queues/lists. For example, a process that has used its time quantum will be placed on the ready queue and a new process will be moved to the running state... this sequence of operations will cause the system clock to be incremented twice. The system clock will also be incremented by the amount of a context switch when moving from the hold queue to the ready queue, as well as from the run state to the termination list. A job arrival event (i.e., moving a process from the event list to the hold queue) is NOT considered part of the job/process migration scenario – hence, the system clock is not incremented.

PART 1 – DUE: Oct 18 (1.5 Weeks)
Part 1 reflects an interim product. You will turn in to me a listing that contains a complete dump of the **event list**, **hold queue** (SJN scheduling), and the **ready queue**. This will require that you implement the event list manager, the hold queue with SJN scheduling, and resource allocation. I will provide you with an input file (it can be ftp'ed from arthur.cs.vt.edu). This part will count 12 points* toward your final grade.

PART 2 – DUE: Oct 27 (3 Weeks Total)
Part 2 reflects a completed product. You will turn in to me (a) a printed copy of the program, (b) a printed copy of the output of your program. You will ftp to the appropriate "incoming" directory on arthur.cs.vt.edu a **tar'ed** and **gzipped** file containing: the program source code, a compiled version of

the source code, files containing the input to your program, files containing the output of your program, and a README file containing your name, SSN, and how to invoke your program..

Some simple sample data and expected output will be provided. THE FINAL SET OF TEST DATA WILL BE PROVIDED BY ME AT A LATER DATE – THE RESULTS OF RUNNING YOUR PROGRAM ON THE FINAL SET OF DATA SHOULD MATCH MY RESULTS EXACTLY!

Attend class for further details, answers to questions, and hints on how one might construct such a simulator. I WILL be stating additional requirements IN CLASS. You WILL be responsible for ALL that I say relative to this programming assignment!

Notes, Considerations and Requirements

(1) You may use a linked lists or circular buffer implementation strategy.

(2) The maximum number of processes that will ever be given to you is 20.

(3) Although I give you data in "real" quantities (because they reflect a more realistic OS flavor), you will convert them to integer by multiplying all real numbers by 1000 (implement a "round up" computation because real numbers whose internal representation cannot be stored exactly might be off by .001), do your work using integers, and then convert your results back to real numbers by dividing the appropriate quantities by 1000. I highly recommend this course of action -- it simplifies your programming considerations.

(4) Whenever a process transitions from one active state to another, i.e. hld=>rdy, rdy=>run, run=>rdy, run=>blk, run=>term and blk=>rdy, a time cost is incurred. The first action you should take in handling a transition is to add the “context switch” (CTX) time to the system clock. THEN, take the appropriate actions that physically move the process. This requirement helps your timestamping keep in sync with mine. For example, starttime is when a process enters the ready queue for the first time… this can be off by one CTX if you move the process record to the ready queue, interrogate the system time clock for the current time (to get process starttime), and THEN charge the CTX time. Note: the timestamp for a process termination (or completion) is when it completes processing in the runstate... not on entry into the termination state (a difference of one CTX time).

(5) Because our process migrations may be a few CTX's out of sync (and VERY few), and because I/O requests and Process terminations requests must be initiated from the run state, some form of process record “flagging” must be implemented. Effectively, if an I/O request or Process termination request is given for a process that is NOT currently in the running state, that action will be delayed until the designated process actually enters the running state. When the appropriate process does enter the running state, the designated action will IMMEDIATELY be applied against that process. No “flagging” is necessary for those requests whose designated processes are in the running state at the time the I/O or termination request event occurs.

(7) If a process’ quantum is up and there is no other process in the ready queue, let the process in the run state continue to execute for another quantum.

(8) The event list is checked after each quantum has been run and charged to the system clock, and after each CTX has been charged to the system clock.

The following is an algorithmic approximation of how to program your simulator

```
READ SYSTEM PARAMETER LINE (TYPE 0) AND INITIALIZE SYSTEM PARAMETERS.

READ IN ALL EVENTS AND STORE IN EVENT LIST (ASSUME EVENTS ARE IN ASCENDING ORDER)

PROCESS INITIAL EVENTS (WHERE EVENT TIMES = INITIAL SYSTEM START TIME)

WHILE (EVENT LIST NOT EMPTY) || (PROCESSES REMAIN IN NON-TERMINATED STATE) {

       IF (RUN STATE NOT EMPTY) THEN {

              IF (TERMINATION FLAG NOT SET) && (I/O REQUEST FLAG NOT SET) THEN {

                     GIVE PROCESS 1 QUANTUM OF EXECUTION TIME *

                     IF PROCESS HAS COMPLETED EXECUTION THEN
                            MOVE PROCESS TO TERMINATION LIST **
                     ELSE   MOVE PROCESS TO READY QUEUE **

                     PLACE NEW PROCESS INTO RUN STATE **
              }

              ELSE
                     While (TERMINATION OR I/O REQUEST FLAG SET) THEN PROCESS CONDITION **
       }

       ELSE
              PLACE NEW PROCESS INTO RUN STATE **

}

PRINT CURRENT SYSTEM VALUES
PRINT DETAILED PROCESS INFORMATION FOR EACH QUEUE/LIST/STATE
PRINT SUMMARY STATISTICS

================================================================================

EVENT LIST PROCESSING:

       WHILE (EVENT LIST IS NOT EMPTY) && (NEXT EVENT TIME <= SYSTEM CLOCK) {

              GET EVENT

              CASE EVENT TYPE

                     ARRIVAL:           CREATE PROCESS DESCRIPTOR
                                        MOVE PROCESS TO HOLD QUEUE

                     TERMINATION:       IF PROCESS IS IN RUN STATE THEN
                                             RELEASE RESOURCES
                                             MOVE PROCESS TO TERMINATION LIST **
                                        ELSE FLAG PROCESS FOR TERMINATION

                     I/O REQUEST:       IF PROCESS IS IN RUN STATE THEN
                                             MOVE PROCESS TO BLOCK LIST **
                                        ELSE FLAG PROCESS FOR BLOCKING

                     I/O COMPLETION:    MOVE PROCESS FROM BLOCK LIST TO READY QUEUE ***


                     PRINT STATS:       PRINT STATISTICS
       }

       IF ARRIVAL OR TERMINATION HAS OCCURRED ALLOCATE RESOURSES IF POSSIBLE ***

================================================================================
```

The event list is checked each time the system clock has been incremented. The system time clock is incremented after (1) each quantum has been allocated [*], (2) after each CTX [**], and (3) anytime there is a movement of a process from one queue/list/state to another queue/list/state that is not due to a CTX [***]. There is no time charged for moving process from event list to hold queue.