CS 385 –Operating Systems – Fall 2011
Homework Assignment 2
CPU Scheduling Simulation

Due: **Thursday October 20th. Electronic copy due at 3:00, optional paper copy at the beginning of class.** (Note: It is recommended that some significant progress be made on this assignment before the mid-term exam on Wednesday October 12th.)

Overall Assignment

For this assignment, you are to write a discrete event simulation to analyze different CPU scheduling algorithms. A number of simultaneous processes will be simulated, each alternating between bursts of CPU usage and I/O waiting. The process data will be read in from a data file.

The general process is shown in Figure 5.17 from our textbook:

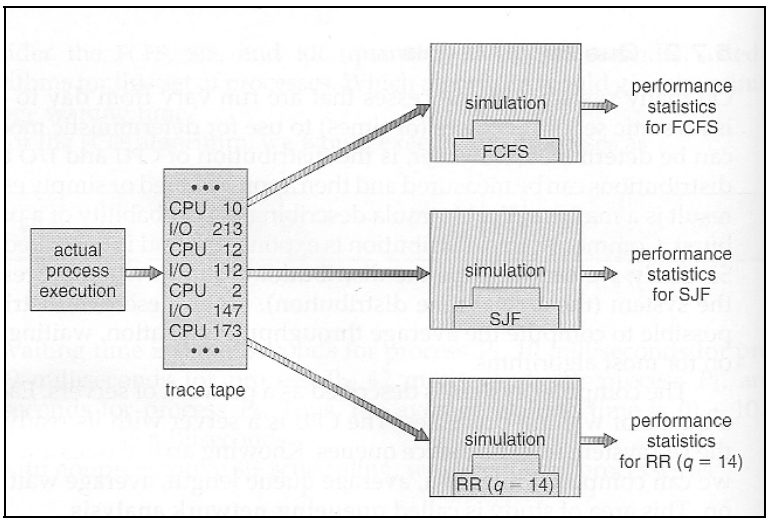


Figure 5.17 Evaluation of CPU schedulers by simulation

Background: Discrete Event Simulation

When conducting a simulation in which events occur at different times, there are two basic approaches:

1. Set up a loop in which each iteration of the loop corresponds to a particular time step, such as a second, millisecond, day, or year. This approach works well when events occur at regularly occurring intervals, but is problematic when the events are irregularly spaced, because the time step has to be small enough to match the resolution of the event timing, and if nothing happens at many of the time steps, then those are just wasted cycles. This is **NOT** the approach we will be taking for this assignment.
2. Set up a loop, but this time, 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, (DES for the remainder of this document.) You can think of this as jumping forward in time from one event on your personal planning calendar to the next, and dealing with each event as it occurs.

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 we 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. Alternatively, if we can predict that a particular event is destined to be canceled, then we can simply refrain from inserting it into the heap to begin with.)

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.)

The general pseudocode for a DES is as follows:

```
while( heap not empty ) {

     extract an Event from the heap.

     update time to match the Event.

     switch( type of Event ) {

          Process this event, possibly adding new Events to
          the heap.

     } // switch

} // while

Process statistics collected during Event processing & report.
```

Background: Templates and the Standard Template Library (STL) priority_queue Data Type

C++ introduced the concept of ***templates,*** which can be used by the compiler to generate functions and/or classes as needed, which vary only by the type of data acted upon or included. The Standard Template Library, STL, defines a number of very valuable templates, including a number of data structures such as vectors, lists, sets, maps, hash tables, and most importantly for this assignment, priority queues.

Templates are invoked by giving the name of the template, following by one or more type parameters in < angle brackets >, followed by the data variables to be declared. So for example:

```
vector< int > myList;
```

declares myList to be an STL vector of ints.

For this assignment we will be using the priority_queue template, which may be invoked (in its most general form) as:

```
priority_queue< Type, Container, ComparatorFunction >
```

where:

- Type is the type of data stored in the priority queue, e.g. int.
- Container is the type of underlying container used for the implementation of the priority queue, e.g. vector< int >. The default value if this parameter is not specified is vector< Type >.
- ComparatorFunction is the function used to order the items in the priority queue. Specifically the function must take two arguments of type Type, and return a boolean value that is "true" if the first argument is "less than" the second argument. In practice it is up to the author of the comparator function to determine how to evaluate "less than". If this parameter is not provided then the default comparator is the STL function less< Type >.

The priority queue template creates a class object having the following important methods (among others):

- void push(const Type &); - Adds (a copy of) a new data item to the priority queue, and increases the size of the queue by one.
- bool empty(void) const; - Returns true if and only if the queue is empty.
- const Type & top( ) const; - Returns a (reference to) the top (first) element of the priority queue, without actually changing the queue or removing the item.
- void pop( ); - Removes and discards the top (first) element of the priority queue, without returning it.

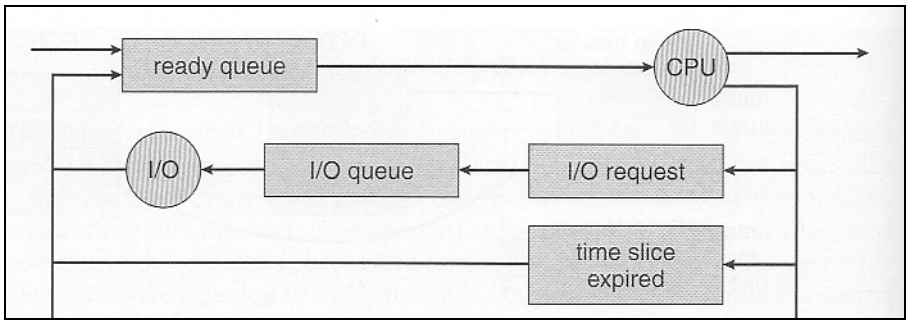
Documentation for STL, including tutorials and examples, can be found at http://www.sgi.com/tech/stl/ or at http://cplusplus.com/reference/stl/ . (In addition to priority queues, you may also be interested in vectors, and/or other containers in the STL.)

Other notes:

- It is necessary to #include<queue> in order to use the STL priority queue.
- The STL priority queue normally delivers the "largest" item to the top of the queue, assuming the comparator returns true when its first element is "less than" the second element. In order to produce a priority queue that delivers the "smallest" item to the top of the queue, it is only necessary to reverse the "directionality" of the comparator function. For example, priority_queue< int, vector< int >, greater< int > > would declare a priority queue of ints where the smallest int is always at the top of the queue.

Process Scheduler Simulation

The scheduler that we will be simulating is illustrated in the following portion of Figure 3.7, with the life cycle of processes illustrated in Figure 3.2:



Portion of Figure 3.7 to be simulated for this assignment

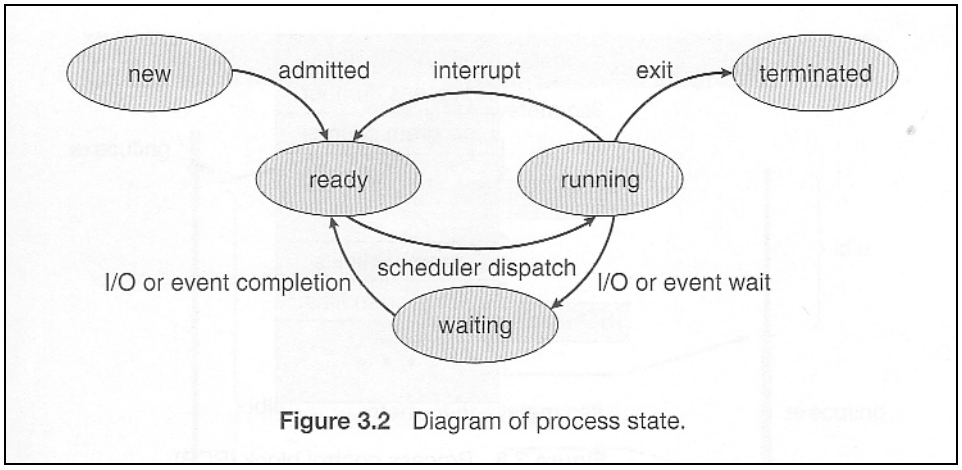


Figure 3.2 Diagram of process state.

Required Data Structures

It is envisioned that your simulation will require <u>at least</u> the following data structures, plus additional variables (and fields in some of the below structures) for the accumulation of statistical data.

- Process - Define either a struct or class type named "Process", having <u>at a minimum</u> the following fields:
    - enum state { new, ready, running, waiting, IO, terminated }; // waiting indicates it is in the I/O waiting queue, and IO indicates it is on the IO device.
    - double CPUburst[MAXCPUBURST];
    - double IOburst[MAXCPUBURST - 1];
    - int nCPUbursts; // The number of CPU bursts this process actually uses.
    - int priority, type; // Not always used
    - int currentBurst; // Indicates which of the series of bursts is currently being handled.
    - Notes: The state variable indicates whether currentBurst refers to a CPU burst or an I/O burst. Additional fields will need to be defined to keep track of statistics, such as total CPU time executed (so far), total waiting time, etc.

- Process Table - an integer-indexed collection of Processes, implemented either as an array or a vector. A process's position within the collection is essentially its PID.

- CPU, I/O Device - These can both be simple integers, indicating which process is currently on the CPU or on the I/O device, where the integer is an index into the process table. Set the value to -1 if the CPU or I/O device is idle.

- Ready queue, I/O queue - These should be implemented as priority queues, containing process IDs (indices into the process table) of the processes in the queue.

    - If they are initially implemented as type priority_queue< int >, then the processes will be prioritized by PID number, with processes having a high PID number exiting the queue first. Obviously this is not the behavior we want to model!

    - They should be implemented as type priority_queue< int, vector<int>, function >, where "function" is the name of a function you write that will take two ints as arguments and return a boolean result. The function is defined as a less-than comparator, which should return "true" if the first int is "less" than the second int.

      In practice your comparator should examine the two processes indicated by the PIDs given to it, and return true or false depending on the relative process priorities, or arrival times, or remaining burst times, or other criteria depending on the scheduling algorithm being implemented.

- Event - Define a struct or class named "Event", having at a minimum the following fields:

    - enum eventType { Arrival, CPUburstCompletion, IOcompletion, TimerExpired };

    - double time; // Defined as time units since the start of the simulation

- Event queue - priority_queue< Event, vector<Event>, function >. Alternatively this could be implemented to hold Event pointers instead of Events.

- Notes: In order to keep your data structures assignable and copy constructable without having to write assignment operators or copy constructors, they should only contain basic data types. I.e. they should not contain any pointer variables. (This avoids concerns over deep copies versus shallow copies when using STL storage structures.)

Specific Comparator Functions to be Written

Your program will need to include at least the following three comparator functions:

- bool readyQueueComparator(int, int); - Used in the priority queue for the ready queue. Implementation of alternative algorithms can be achieved either by having a series of different comparator functions for different scheduling algorithms, or perhaps better, having a single function that examines a global variable to determine what criteria to use for its comparison.
- bool IO_QueueComparator(int, int); - This needs to be a separate function from the readyQueueComparator, probably implementing FCFS based on arrival time in the I/O queue.
- bool eventQueueComparator(Event, Event); - Returns true if the time of the first Event is <u>greater than</u> the time of the second Event, (so that the earliest Events exit the queue first.)

Events to be Processed

The event heap will initially be populated with a single "Arrival" event, generated by reading the first byte of the input data file. Event objects all contain the time at which the event is scheduled to occur, an indicator of what type of event it is, and additional information pertinent to the particular event. For this particular simulation, the events to be considered will include at least the following, (and possibly others.)

It is recommended that the switch for the DES loop be developed one case at a time, in this order:

- The default case – Prints all the information about the Event, and nothing more. This will handle any Event types that have not yet had their cases written. Initially it will report the Arrival event used to initialize the event queue, which you can use to test that the queue was properly initialized.

- The Arrival case – Read in data corresponding to the arriving process, and create a new entry in the process table. (See Data File Interpretation, below.) 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 no preemption.) Before completing this case read in the time of the next process arrival, and add a new Arrival event to the event queue.

- The CPU Burst Completion case – If this was the last of the CPU bursts for this process, its state can be set to terminated. Otherwise if the IO device is idle it can be placed on the IO device with IO status and an IOcompletion event added to the event queue. Otherwise it gets put into the IO queue with status waiting. 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. (For initial testing you may want to set maxCPUbursts to 1, so that each process completes in a single CPU burst, with no I/O.)

- I/O Completion – When a process has completed its I/O task, then this process is moved to the ready queue. If the CPU is idle, then a process from the ready queue can be selected and moved to the CPU. The I/O queue is checked to see if there are any other processes waiting for the I/O device, and if so, then one is assigned to the device and a new IOcompletion event is added to the event heap. See also "preemption" below.

- 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.

  (Note: In a real system the duration of CPU bursts is not known ahead of time, and so the issue of expired timers for which the process has already left the CPU needs to be handled. In your simulation however, because the CPU burst time is known in advance, you can intelligently place timer events in the event heap only for processes whose next desired burst exceeds the time quantum used in the simulation.) Note that when limited time slices are implemented, then the system must keep track for each process not only the total duration of its next desired CPU burst, but also what fraction of that burst has already been satisfied by previous time slices.

- Preemption - This is not a specific event type, but may be triggered by either the Process Arrival or I/O Completion events. If the scheduling algorithm being simulated includes preemption, and the process currently on the CPU has a lower priority than the process which just arrived in the ready queue (either from a new arrival or from the I/O Queue), then the current process may be moved to the ready queue and a new process dispatched, similar to the processing of a timer expiration event. Note that if timers are being used, then the timer will need to be reset when preemption occurs.

Data File Interpretation

The data file should be interpreted as a series of process information blocks, read as a series of binary numbers, having the following interpretation:

- arrivalTimeIncrement - read in one byte (unsigned char) and divide by 10.0 to yield the time at which the next process arrives after the arrival of the previous process. Note that the result of the division should be a double in the range 0.0 to 25.6, not an integer.
- nCPUbursts = unsigned char % maxCPUbursts + 1
- priority = unsigned char.
- type = unsigned char. May be ignored, or used to implement scheduling algorithms that depend on the type of a process, such as real-time, interactive, batch, system, etc.
- [CPU_bursts] - a series of nCPUbursts bytes, each divided by 25.6
- [IO_bursts] - a series of (nCPUbursts - 1) bytes, each divided by 25.6

Sample Data Files

Any data file whatsoever may be used as a data source. A set of sample binary data files (images) will be provided on the web site, having different properties of variability.

Additional Details

- The syntax for running the scheduler shall be:

```
scheduler filename maxProcesses maxCPUbursts  quantum ...
```

  - The filename is the name of a data file to be read for process data.
  - maxProcesses is optional and indicates how many processes to simulate. If not present, or less than 1, or if larger than the number of processes actually in the input data file, then all the processes in the input data file are simulated.
  - maxCPUbursts is optional, and puts a limit on the maximum number of CPU bursts a process can have. This value must be less than or equal to MAXCPUBURSTS, the size of the array in the Process data structure, and should have a default value of MAXCPUBURSTS. It is recommended that MAXCPUBURSTS be initially set to 10. (Increasing maxCPUbursts effectively increases the load on the system.)
  - quantum is optional - the time quantum to use for algorithms such as round-robin.
  - . . . Other optional parameters are permitted, to study specific algorithm parameter adjustments. These will need to be clearly documented in your "readme" file.

- Statistics to be collected may include run times, I/O times, wait times (for CPU and for I/O), and/or other statistics needed to evaluate common CPU scheduling criteria.

- At least two different algorithms are to be analyzed, under a range of different load levels.

- If the end of file is encountered when reading in a new process, then that process should be ignored, and no further Arrival events processed.

Required Output

- All programs should print your name and ACCC account ID as a minimum when they first start.
- Beyond that, this program should print run statistics suitable for algorithm analysis.
- In addition to a program printout, a short memo / report shall be written with the results of the algorithm analysis. What does your simulation tell you about the different scheduling algorithms that you have investigated, and how does your data back up your conclusions? Can you determine anything about which scheduling algorithms work best under which conditions, and how parameter adjustment affects the performance of the algorithms? Can you make any recommendations based on your findings? (Either regarding the scheduling algorithms if your results are meaningful, or how to improve the simulation to make it more effective otherwise?)

Other Details:

- The TA should be able to build your program by typing "make scheduler". Provide a makefile if the default make rules are not sufficient. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department machines.

What to Hand In:

1. Your code, **including a makefile if necessary, and a readme file,** should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) should be handed in **at the beginning of class** on the date specified above.
6. Make sure that your **name and your CS account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:

- More than two algorithms, possibly involving multiple ready queues.
- Explore parameter adjustments within algorithms, such as the length of a time slice.
- Find real (measured) process data, and analyze algorithms using that data. How do your results compare to what you found using randomly generated data?
- Implement multiple ready and/or I/O queues with different scheduling algorithms and/or access time properties.
- Incorporate dispatch latency, and see how it interacts with varying time slice durations.
- Simulate multiple CPUs (real or virtual), each having their own scheduler.
    - You could explore a single ready queue versus separate ready queues for each processor.
    - You could explore push versus pull migration, and the parameters which control it.
    - You could explore symmetric versus asymmetric multiprocessing.
    - Processor affinity. (May require a penalty for cache rebuilds.)
    - The schedulers do not necessarily need to follow the same algorithm, particularly in the case of asymmetric multiprocessing.