CS 385 –Operating Systems – Spring 2006
Homework Assignment 4
Disk Scheduling Simulation

Due: **Friday 8 December. Electronic copy due at 3:00, paper copy at the beginning of class.**

Overall Assignment

For this assignment, you are to write a discrete event simulation to analyze different disk scheduling algorithms. A given number of disk requests will be processed, spanning a 2000 unit time span.

References

This assignment will use data structures and methodologies similar to those in HW3 – See that assignment for further background information.

Additional Details

- The hypothetical disk upon which this simulation is based has 1000 cylinders, 10 heads, and 16 sectors per cylinder.

- The time required to satisfy a disk access request is (1.0 + 0.003 * "distance"), where "distance" is measured in cylinders from the current cylinder position to the next one.

- The number of disk requests to process during the 2000 time unit period is a variable. It is expected that the range in which the results will be interesting and meaningful will fall somewhere between 1000 and 2000. (The average time to process a request, not counting time spent waiting in the queue should be about 2.0 for FCFS, which means 1000 requests would just about fill the 2000 time units available. Other algorithms will have a lower average access time, requiring more requests to see any significant time spent waiting in the queue. One criterion for evaluating different algorithms is the size of the backlog of requests remaining at the end of the 2000 time unit period, or alternatively, how many processes can be handled in 2000 time units without generating a significant backlog at the end of the time period.)

- In order to compare algorithms with a common set of input requests, your program should take a data file name as a command line argument, and should read the disk requests from the data file. The file should contain one line per disk request, with the following information:

  time head sector cylinder

  where time is a double and the rest are ints, separated by white space. (Your particular program may choose to ignore head and sector information, but a common format is requested for all programs.)

- You may optionally accept a second command line argument indicating how many requests to read from the file. This would allow you to generate a single 10,000-item file and then read whatever portion of it was suitable for a particular run. You may also optionally provide a command line argument indicating which algorithm to use for a particular run. Make sure that any such arguments you support are fully documented in your readme file.

- A separate program will be needed to generate the input data file. A sample data file may or may not be provided. (You may use an alternative program, such as MS Excel, to generate your input data file if you wish, so long as the result is a plain text file with the given file format.

- (Note: The assignment is written to use discrete event simulation, but if the input data file is sorted in order of disk requests, then it may be possible to implement an alternative algorithm that accomplishes the same results. This is acceptable, but development of the alternative algorithm is your responsibility, and must be documented thoroughly in your readme file. Note also that this idea is in conflict with having an overly large input file and only reading the first N lines of the file.)

- Two primary data structures are foreseen for this assignment: A priority heap containing Events, and a collection (queue, list, array, heap, or whatever is appropriate) containing currently pending disk requests.

- Two Event types are foreseen for this simulation, with the following responses needed for each type:

    - Disk Request Arrival – If the disk is currently busy, then add the request to the collection of current pending requests. Otherwise, process the request, which includes putting a Disk Request Completion Event into the heap.

    - Disk Request Completion – If there are any pending requests, then select one to process, and put a new Disk Request Completion Event into the heap. If there are no pending requests, then put the disk controller into an idle state. (Or do some read-ahead if you want to implement caching. See Optional Enhancements.)

- Statistics to be collected may include the average time to process requests, average waiting time in the queue, average total (waiting plus access) time, number of processes which can be processed efficiently in the 2000 unit time period, maximum number of processes waiting in the queue (presumably at time = 2000 for a busy system), etc. You should apply your judgment to decide what data you need to perform a suitable analysis.

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

Required Output

- All programs should print your name and CS 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 can you conclude about the capacity and efficiency of each algorithm, and under what conditions it would be preferable to choose one over another?

Other Details:

- A makefile is required, that will allow the TA to easily build your program. 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 Linux machines.

What to Hand In:

1. Your code, **including a makefile, and a readme file,** should be handed in electronically using the turnin command (see below).

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.

Other Notes:

- The correct **`turnin`** command for this assignment is:

```
turnin –v -c cs385 -p hw4 files
```

- **Note carefully** that every time you submit files to a given project using turnin, it **replaces** any files that you had previously submitted for the same project. It is therefore necessary to submit **all** files for a given assignment with a single turnin command.

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:

- A more realistic allocation of disk requests than total randomness. One option would be to generate a sinusoidal sequence of a bunch of requests followed by relatively idle periods.
- Sequential access could be simulated by generating a sequence of successive reads, distributed over time.
- Caching of tracks, blocks, etc.
- More than two algorithms.
- Explore parameter adjustments within algorithms. (And try to reach some conclusion regarding the "best" value of the parameters.)
- Find real (measured) process data, and analyze algorithms using that data. How do your results compare to what you found using randomly generated data?