

Department of Computer Science and Software Engineering
433-332 Operating Systems
Project 1 - 2003

Project Description

The aim of this project is to give you experience with the lower layers of an operating system, specifically with disk scheduling, and with the notion of simulation as a tool for the evaluation of operating system algorithms.

In the directory `/home/subjects/332/local/proj1` are a number of files. They comprise a working program which simulates the operation of a disk head scheduling algorithm.

You should copy, compile, and familiarise yourself with this program. The program takes a set of N files as input, each of which describes the behaviour of one process. Each line of the files contains three integers: the first indicates how long the process will wait after one disk request is serviced until it issues another request, the second indicates which track the new request is for, and the third is 0 for a read request and 1 for a write. For example, suppose we have the following process file :

```
2 19 1
30 3 1
50 10 0
```

This should be read as meaning

-the process started at time zero

-it issues a write request at time 2 for track 19. This request gets completed at time x

-it issues a write request at time $x+30$ for track 3. This request gets completed at time y

-it issues a read request at time $y+50$ for track 10

The program simulates the operation of a computer that is running these N processes simultaneously (though it does not include CPU scheduling, assuming that the computer has at least N processors).

Each process file corresponds to a single simulated process. *As a result, it is impossible to create a queue of requests using just one input file, because by the time the process can cause more I/O, it has already had its previous requests serviced.*

Modern hard disks contain a significant amount of cache memory to answer requests for data without seeking. Whenever a request is made for data on a particular track, the entire track will be read into the cache. After that, some disk I/O requests may be satisfied very quickly by using this data (for the purposes of simulation, the time taken to read from the cache may be ignored).

If the disk seeks, the cost incurred has two components- the time taken to start the disk head moving and an extra cost for each track that must be seeked past. So the cost of a seek from track

n to track m would cost

$$start_time + (|n - m| * track_cost)$$

The two costs may be redefined at run-time using the `-s` and `-t` flags which set the `start_time` and `track_cost` respectively. Other run-time flags are; `-a` to select the scheduling algorithm to be used (one of `fcfs` or `sstf` or `sstfa`, described below), `-i` to select the initial disk head position and `-c` to set the number of tracks that the disk's cache can hold. You will also need to add support for two more options: `-w` and `-l`, which are described below.

The program is divided into 3 source files; `proj1.c`, `event.c`, and `disk.c` and two corresponding header files; `event.h` and `disk.h`. The first source file, `proj1.c`, does option processing and general initialisation. The second, `event.c`, does the time sequencing required for the simulation handling the time order processing of requests to the disk scheduler. This is an example of an event driven simulation. The third file, `disk.c` contains code to perform the disk simulation and scheduling.

It would be usual for each scheduling algorithm to be implemented in a separate file and linked to the driver routines to create distinct binaries. In this case, however, in the interests of easier processing at submission time a number of algorithms are to be defined in the same file, `disk.c`. However, the only disk scheduling algorithm currently implemented is first-come-first-served, possibly the simplest, but certainly not the best in general.

Your task for Parts A and B is to implement two other scheduling algorithms. You may make modifications to the files we have provided. Indeed, for part B, the details of how events are processed may need to be changed. However, you should not change the printing function, or output anything else at any point in the program, as the output will be examined for marking purposes and you will lose marks if the output format is changed.

Other files in the data directory include a number of header files and another program called `gen1` which can be used to generate data files to use for testing your implementations. The file `README.gen` describes the use of this program. Also two suites of test files are provided, `test*` and `proc*` the first suitable for initial debugging and the second for more extensive tests. A `Makefile` is also provided.

As always, a high standard of coding style and documentation is expected. Because you are implementing a simulation tool, *simplicity and clarity of code should be prioritised above algorithmic efficiency.*

The algorithms that you have to implement, are described below. Note, however:

- when the disk head is idle, assume that the head is to be left where it is. ie. It does *not* return to the center.
- When there is more than one possible request to satisfy (ie. there is more than one request on a single track), use the first one which arrived in the queue.

The following points may clarify how the cache operates:

- The cache is a hardware cache. You are simulating a disk scheduler in an operating system. The scheduler does not know what is in the cache; when it asks the disk to read a track, and the data is in the cache, it may receive an answer instantaneously without the disk head moving.
- cache semantics can in general be quite complicated, and are beyond the scope of this project. As a result, the cache code has been written for you, and you only need use it in the same way that `fcfs` does.
- The default cache size is 0. If you want to see the cache in action, you need to use a `"-c cache-size"` flag.

Part A

Your task for Part A is to implement the SSTF algorithm.

Part B

Your task for Part B is to implement a variation of the SSTF algorithm, we will call SSTFA. It is based on the notion of anticipatory scheduling described in the paper “Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O” (located on the 332 website). As motivation, consider the following extract from this paper

“Disk schedulers are typically work-conserving, since they select a request for service as soon as (or before) the previous request has completed. Now consider processes issuing requests synchronously: each process issues a new request shortly after its previous request has finished, and thus maintains at most one outstanding request at any time. This forces the scheduler into making a decision too early, so it assumes that the process issuing the last request has momentarily no further disk requests, and selects a request from some other process. It thus suffers from a condition we call deceptive idleness and becomes incapable of consecutively servicing more than one request from any process ... deceptive idleness forces a seek optimising scheduler to multiplex between requests from different processes. The ensuing head seeks can cause performance degradation by up to a factor of four ..”

Reading this paper will provide more background and motivation about anticipatory scheduling.

The algorithm SSTFA is a *simplified version* of the approach described in the paper, one that is more suitable for implementation in a student project.

SSTFA maintains a queue of disk requests in the same manner as SSTF. It requires two command line parameters: w and l (specified using command line options ‘-w value’ and ‘-l value’). w represents the maximum time the scheduler is willing to wait before choosing another request from the queue. l is an integer value ≥ 1 used to denote the ‘length’ of the scheduler’s memory about past requests issued by a process.

Let the arrival times of all the disk requests issued in the past by a process α be R_1, R_2, \dots, R_n . Let the corresponding completion times of each of these requests be C_1, C_2, \dots . When the most recent disk request n for process α completes, a timer is started. The duration of this timer is $D = \text{minimum}(k, w)$. If another request from process α arrives in the queue while the timer is running, then the timer is stopped. As soon as the timer runs out or is stopped, the scheduler selects a request from the queue using the SSTF policy. Here, $k = w$, if $n = 1$. Otherwise, $k = \lfloor \text{median}\{I_1, I_2, \dots, I_x\} \rfloor^1$, where $x = \text{minimum}(n - 1, l)$ and each I_i ($1 \leq i \leq x$) represents a time interval between the completion of request $n - i$ and the arrival of the next request $n + 1 - i$. i.e. $I_i = R_{n+1-i} - C_{n-i}$.

Observe that if $w = 0$, then SSTFA behaves exactly the same as SSTF.

Note: The code that you have been provided with implements the FCFS algorithm. Unlike FCFS, SSTFA deliberately allows the disk head to be idle at certain times. It may thus be necessary for you to change the details of how some events are handled during the simulation.

Part C

This part requires you to write a succinct description of the tests you have performed to verify the correctness of your SSTFA implementation. This description should be put in a file named *testing* and should not be longer than 30 lines of text. The exact format is left for you to decide, but you should aim to describe at least four distinct types of test cases.

Assessment

Marking: This project is worth 15% of your final mark for the subject. Your submission will be tested and marked with the following criteria:

- 3 points for SSTF in Part A giving expected output;

¹Median: The middle measurement when items are arranged in order of size; or, if there is no middle one, then the average of the two middle ones

- 5 points for SSTFA in Part B giving expected output;
- 5 points for the coding style and documentation. It is particularly important you clearly describe any modifications/extensions you make to the code that processes events.
- 2 points for test plan description in Part C

Individual Work: You are reminded that all submitted assignment work in this subject is to be your own individual work. Students submitting the work of others for assessment will be penalised by the Department, and risk prosecution under the University’s Discipline Regulations. Students who allow other students access to their work will also be penalised, even if they themselves are sole authors of the program in question. You may, of course, copy the standard skeleton files. After that, *all work must be your own and no-one else’s*. Automated similarity checking software will be used to compare submissions.

Subject Assessment Policy: The two programming projects for 332 comprise 30% of the marks. The exam is worth 70% of the marks. There are also two hurdle requirements: you must obtain a total of at least 15/30 for the projects and 35/70 for the exam. Students who fail either the exam or the project hurdle will have their mark adjusted so as to ensure that they fail the subject as a whole by the amount by which they failed the hurdle. For example, a student with a mark of 25/30 for the projects and 28/70 in the exam (seven marks short of the hurdle) will be assigned a final mark of 43.

Questions and further information: The subject WWW home page will be kept updated with any further information relevant to the project and will be considered part of the specification. Questions about the project specification should be directed to the newsgroup cs.332. *Please note that even with these resources, there may exist cases where there is ambiguity about how your program should behave. In such situations, you should make a sensible assumption and justify and document it in your program.*

Submission: You must submit all your files (including Makefile and testing file), using `submit 332 1`, not later than Tuesday, 15th of April, by 5:00pm. Late submissions will incur a deduction of 3 marks per day (or part thereof). If you have a medical or similar compelling reason for being late, you should contact Saeed Araban (araban@cs.mu.oz.au) as early as possible to ask for an extension (if e-mailing, use subject header ‘332-proj1-extension’). Please include your name, login id and student number in a comment at the top of `proj1.c` file.