Simon Fraser University

CMPT 300-D100: Project II

Design and Implementation of a Monitor Construct.

Authors

Joel Teichroeb
jteichro@sfu.ca

Matt Grandy
mgrandy@sfu.ca

Andrew Inwood
ami2@sfu.ca

November 24, 2011

Contents

1 Introduction 2
2 High-level Design 2
2.1 Monitor Design 2
2.2 Hard Drive Simulator Design 2
2.3 Hard Drive Simulation Experiment 4
3 Analysis 5
3.1 Time to Service a Request 5
3.2 Distance per Request 7
3.3 Reversals per Request 9
4 Known Bugs 12
4.1 Too Many Threads Segfaulting 12
5 Listings 13
5.1 Sample Output 13

1 Introduction

Monitors are a powerful technique for providing mutual exclusion to shared resources in a multiprogramming environment, because of their ability to simplify the programming interface for users of the shared resource. They are, however, a complex construct that must be supported by the compiler for the language that makes use of them. Seeing as the design goals of C/C++ are primarily based on speed and efficiency of the resulting code, Monitors are not supported in these languages.

In this report we design and implement a software interface that simulates the interface of a Monitor, using the pthreads package to provide the mutual exclusion required. We then use the Monitor construct to provide mutual exclusion to a hard drive simulator, which receives I/O requests from multiple users.

2 High-level Design

2.1 Monitor Design

Since the monitor construct has to be implemented as a language feature, we can only hope to simulate the interface of a monitor in C++. The strategy is to use object oriented design and write a Monitor class which has the same interface as a rudimentary monitor, along with some extra helper functions to provide the functionality which would typically be language features.

Once the Monitor class has been written, a programmer would then be able to write their own monitor as a subclass of the Monitor class, and implement their monitor functions according the documentation provided with the Monitor class. As an example, in this report, we implement a hard drive simulation, and provide mutual exclusion to the shared resources (I/O read/write head) by writing a subclass of the Monitor called HDMonitor. Figure 2.1 features a UML class diagram of the intended system.

As is seen in the diagram, the Monitor class has a very basic interface, with only a few function calls, most of which are typical monitor calls like *signal* and *wait*. Two functions which are not standard are *EnterMonitor* and *LeaveMonitor*. The monitor construct is supposed to only allow one process to use one of its function calls at a time, and the mutual exclusion is supposed to be provided automatically by the compiler. This is not possible using C++, and so when writing a monitor function as a subclass of the Monitor class, the programmer must ensure that the function calls EnterMonitor before doing anything, and calls LeaveMonitor when the function's work is done. The public interface and implementation in C++ can be found in Monitor.h and Monitor.cpp in the online submission.

2.2 Hard Drive Simulator Design

To demonstrate both the correctness of the Monitor class and how to implement a monitor as a subclass of Monitor, we implement a hard disk simulation in C++ using the Monitor interface to provide mutual exclusion to the shared resource (the I/O read/write head). All of the source code can be found in the Listings section. Multiple threads will be generated to request I/O on the tracks of the hard disk (labelled 1 to 15 for our experiments), and

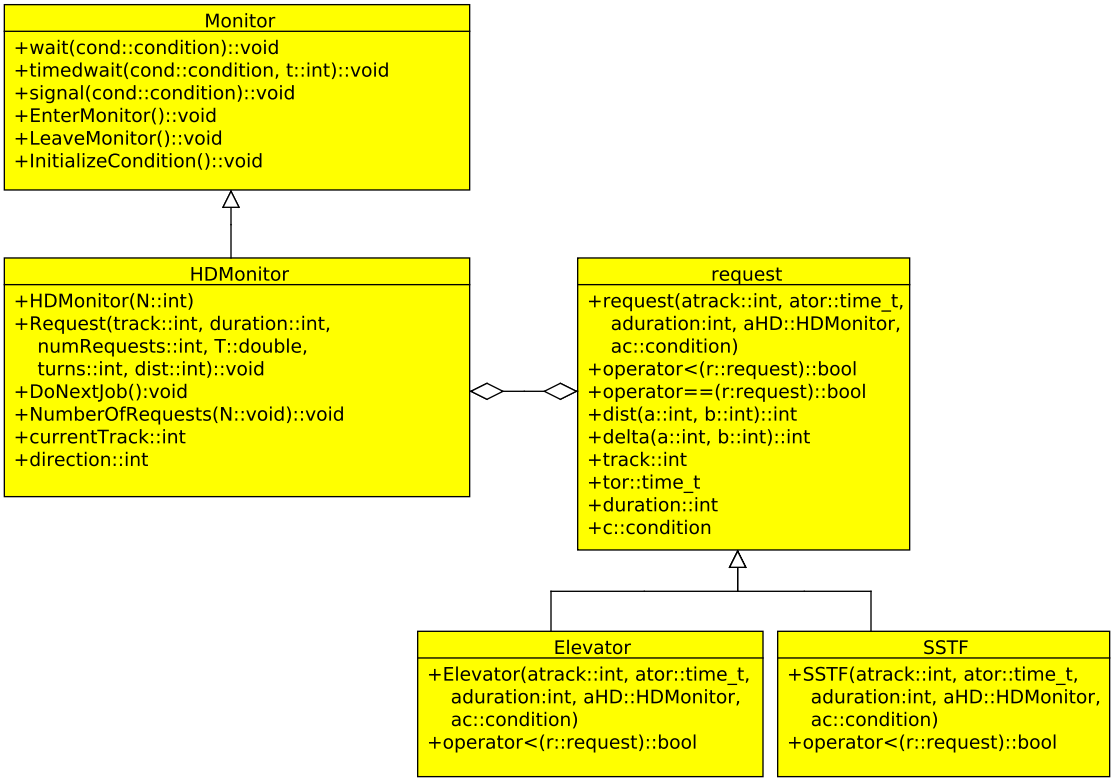

Figure 2.1: UML diagram from the hard drive simulation.

the hard disk will be responsible for managing these requests and servicing them according to a scheduling algorithm. The basic picture is illustrated in figure 2.2. In our design, the hard disk simulator will read requests from standard input, which can be generated by any number of threads.

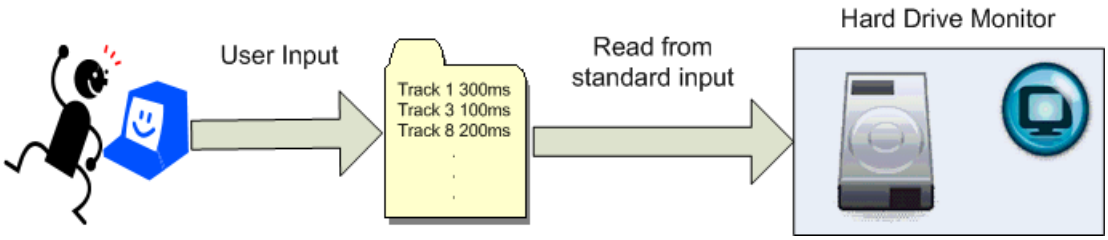

Figure 2.2: Scheduling environment for the hard drive simulation.

We will also take this opportunity to compare the performance of two popular scheduling algorithms for real hardware, namely the Elevator algorithm and the Shortest Seek-Time First (SSTF) algorithm. In the Elevator algorithm, the I/O head moves in one direction, servicing pending requests as it reaches the desired tracks, and only changes direction when it reaches the inner or outer edge. The head will also change direction if there are no more requests for tracks in the direction of travel, but there are requests for tracks in the opposite direction. The SSTF algorithm will move the I/O head to the desired track that is closest to its current position.

To help the simulation decide which request to service next, we define a Request class as was seen in figure 2.1. Not only does the Request object keep a record of all the important request information like the track number and work to be done, but it also defines a weight through the less-than operator ($<$). The definition of the less-than operator depends on the desired scheduling algorithm. A Request object $\mathcal{A}$ is said to be less than the Request object $\mathcal{B}$ if $\mathcal{A}$ is to be scheduled before $\mathcal{B}$. The simulated hard drive can then maintain a list of Request objects as they are made, and decide the next one to service by finding the smallest Request object. As seen in the UML diagram, the implementation of the less-than operator is found in a separate file from the rest of the Request definition, to make it easy to plug in different definitions to get different scheduling algorithms.

2.3 Hard Drive Simulation Experiment

In our experiment we first generated a large file of sample input (1000 requests) to be made to the hard drive. The data consists of tracks chosen from the disk at random, all with an equal amount of work. This choice was made because this kind of data is easy to generate and understand. However, by the principle of locality, it's more likely that a hard disk would receive requests for tracks that are in close proximity.

We generate 5 threads whose purpose is to read requests from the data file, and make the request call to the monitor. We also generate one servicing thread whose purpose is to request that the monitor service a pending request. The servicing thread runs indefinitely, but the requesting threads will terminate once they find there is no more data in the file.

The flow of execution for the hard disk Monitor calls *DoNextJob* and *Request* are shown in figure 2.3. These are the functions called by the threads in the main program. Notice how both functions start and end with the EnterMonitor and LeaveMonitor calls, respectively. Also of note are the calls to *signal* and *wait*, the standard monitor calls. The implementation of the main program logic can be found in MonitorDriver.cpp in the online submission.

In listings 5.1 and 5.1, we show the first few lines of the output of the experiment, using the Elevator and SSTF algorithms, respectively. There are three types of lines in the output, those that read *going to wait*, *Just pushed track X*, and *Working on track X*. These output lines let us know when the servicing thread is going to wait because there are no pending requests, when a request is made, and when a request is serviced. By examining the output, we can see that the scheduling algorithms are working as expected.

As an example, consider lines 54 to 59 of the output from the SSTF algorithm, in listing 5.1. Previously, track 10 was examined, followed by track 6, before the servicing thread output *going to wait*. Then starting at line 54, tracks 2 and 9 are requested. Since the I/O head moved down from 10 to 6, we know the current direction is down. The next lines show that track 9 was scheduled before track 2, because track 9 is closer to track 6 than is track 2. The Elevator algorithm, on the other hand, would have scheduled track 2 before track 9, since track 2 was in the direction of travel.

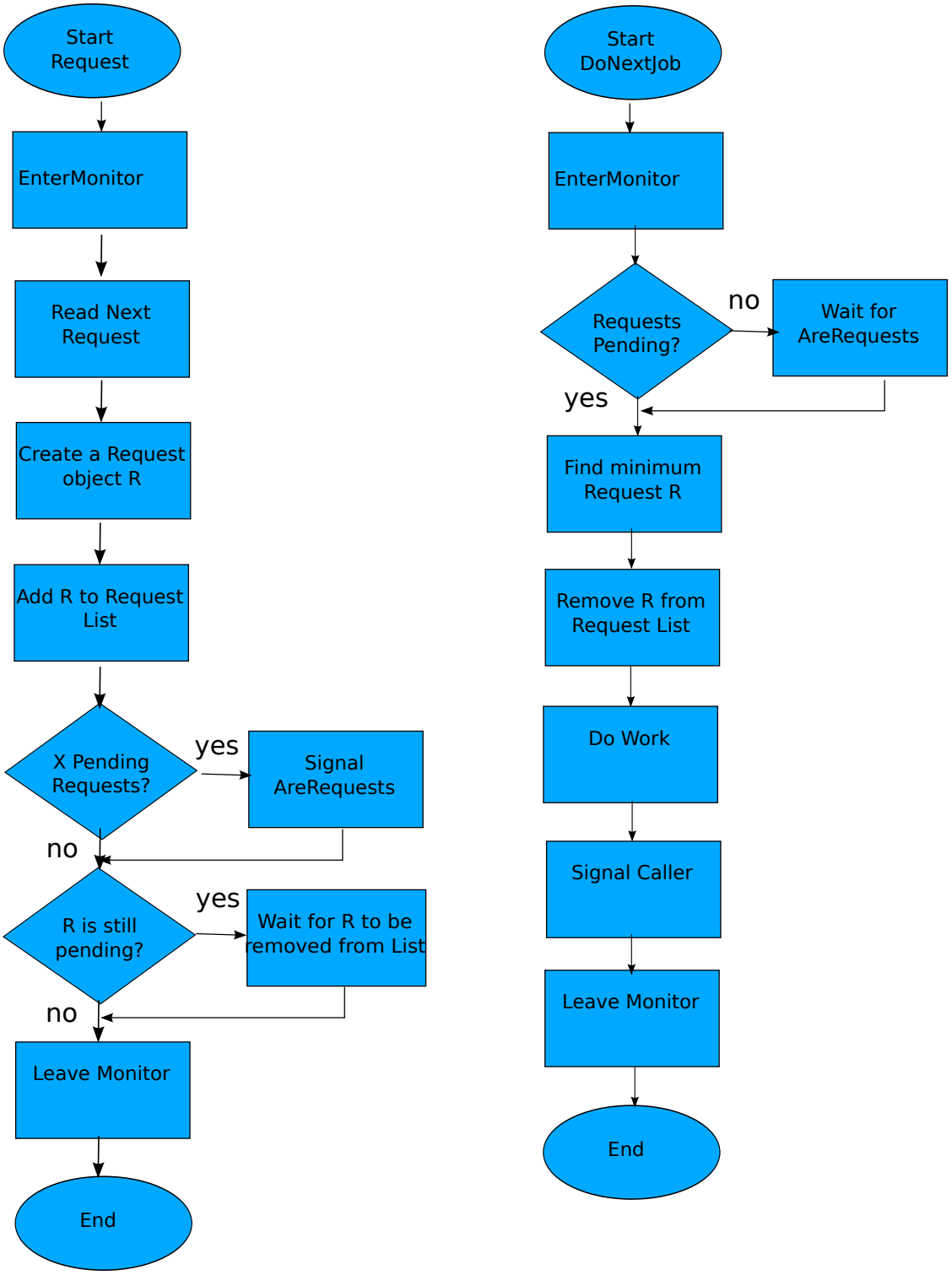

Figure 2.3: Flow of execution for HDMonitor functions responsible for requesting I/O and servicing requests.

3 Analysis

3.1 Time to Service a Request

One of the most important metrics in analyzing the performance of the hard drive is the time it takes after an I/O request is made before it is serviced by the hard drive. This interval is

expected to depend on the number of unserviced requests made to the hard drive thus far, because the servicing thread always has to scan the entire list of requests in order to find the next request according to the scheduling rule. The way we ensure that more requests are pushed into the queue is by increasing the number of requesting threads.

In figures 3.1 and 3.2, this delay interval is shown as a function of the number of request threads for the Elevator and SSTF scheduling algorithms, respectively. This data was gathered from running the simulation multiple times, with increasingly many request threads, and on input data of 1000 requests generated randomly on the disk.

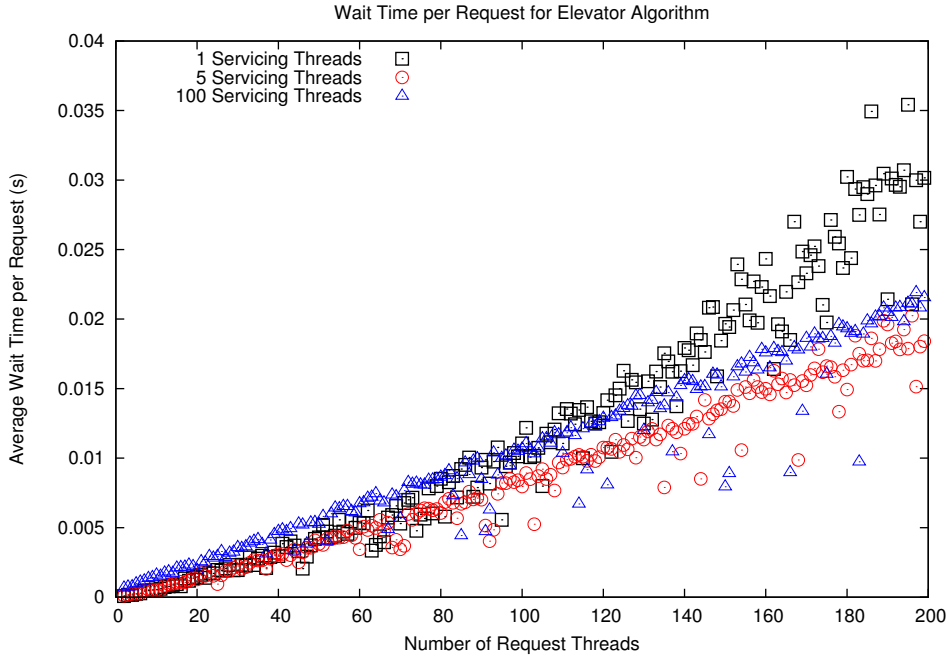

Figure 3.1: The wait time per request is improved by using more than one servicing thread.

From the graphs, we can make a few observations. Firstly, we can see that using this metric, both algorithms perform about equally well on the chosen input data. This happens because the behaviour of the two algorithms is similar when the request queue contains data from everywhere on the disk. In this situation, both algorithms will scan up and down the disk in succession, servicing requests along the way.

Secondly, we can note the change in performance when increasing number of servicing threads are employed. By increasing from one to five threads, there is a clear improvement in performance, especially when there are many request threads. With only one servicing thread, there is a divergence from linear behaviour, but this divergence is delayed when using a large number of servicing threads. However, we can see that when using 100 servicing threads, the performance is reduced from when using 5 threads. This is caused by the overhead of the system having to maintain a large number of threads. Therefore, performance by this metric is optimized by using more than one servicing threads, but not too many, and the optimal value could probably be found by tuning.

There is a test case in which SSTF performs much worse than the Elevator algorithm. The SSTF algorithm can suffer from starvation when most of the I/O requests are on one

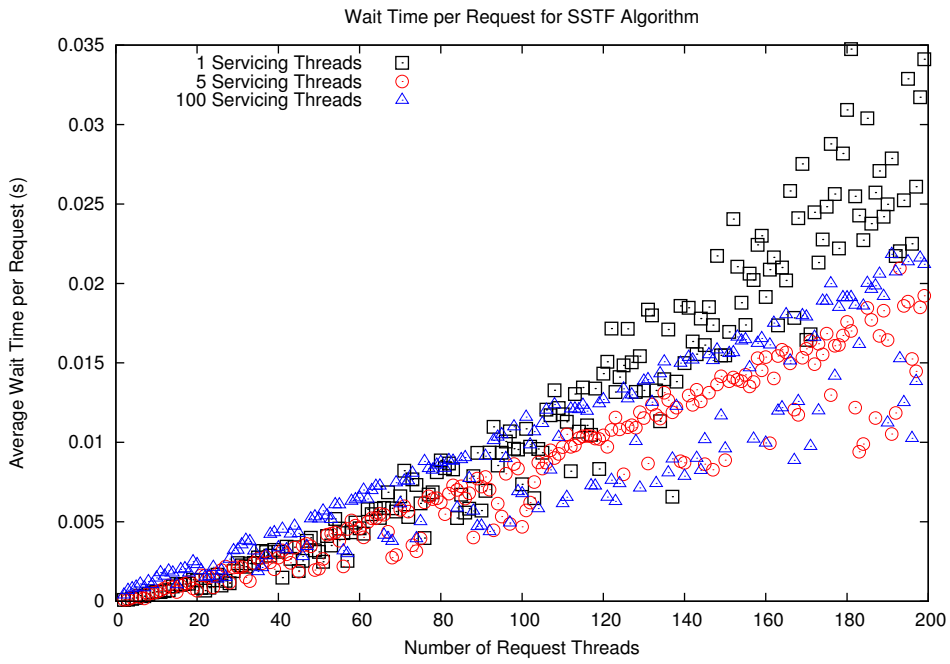

Figure 3.2: The wait time per request is improved by using more than one servicing thread.

end of the disk (say the outer edge), and then a small number of requests are made for the other end of the disk (inner edge). If requests for the outer edge keep coming in before the servicing thread has the chance to clear the queue of requests, then those requests will never run. The elevator algorithm does not suffer from this problem, of course, since it scans the disk back and forth. Unfortunately (or perhaps fortunately!) it is very difficult to construct a test case that demonstrates this behaviour; the scheduling environment makes it difficult to predict how many requests will be pending at any time, so it is not possible to predict how long the servicing thread will execute before pthreads forces it to yield.

3.2 Distance per Request

In the last section, the analysis assumed that all jobs took the same amount of time to execute, and so the waiting interval was mostly attributable to the scheduling algorithm, and the relative amount of time spent in requesting threads versus the servicing thread. In real hardware, the I/O head will require some time to move to the desired track, and so this can be a useful metric. Specifically, we measure the average number of tracks traversed between the time a request is made, and the same request is serviced. In figures 3.3 and 3.4 we show this quantity with respect to the number of request threads in the simulation, for the Elevator and SSTF algorithms, respectively.

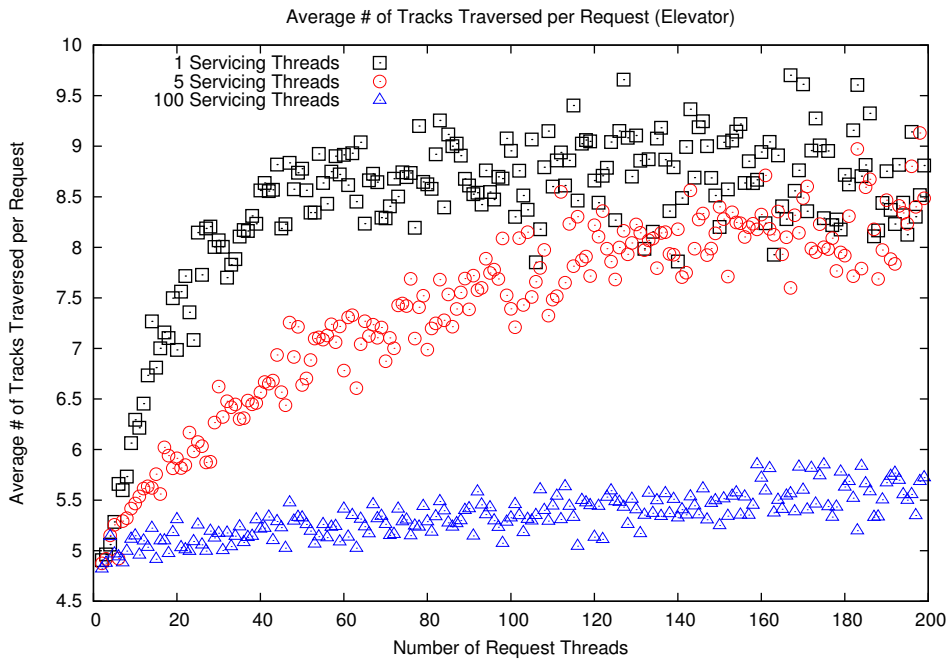

Figure 3.3: The average number of tracks traversed by the I/O head per request can depend significantly on the number of servicing threads in use.

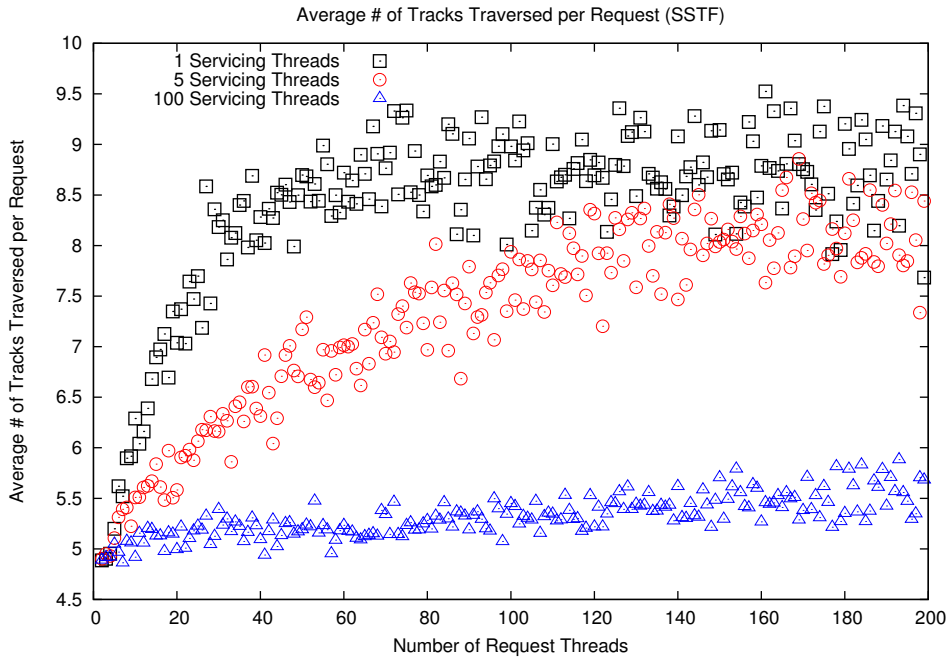

Figure 3.4: Performance of SSTF under the distance metric is comparable to Elevator on random input data.

Again, it's clear that the two algorithms have similar performance under this metric. This is again attributable to the behaviour of the two algorithms being similar on the input. Of particular interest, however, is the profile of the data. When the number of request threads is fewer than about twenty, the average distance per request increases linearly. This happens when there are relatively few requests on the queue (the result of fewer request threads), since the I/O head will not travel in a straight line to the requested track under SSTF or Elevator, but rather will go back and forth between close requests in SSTF, or in the wrong direction with Elevator. However, when there are many requests, as explained before, both algorithms will scan the entire disk up and down servicing all requests. When multiple servicing queues are used, the average distance levels off at increasingly small values. This happens because having more servicing threads reduces the number of pending requests at any time, and so the average distance travelled does not grow as much.

Note that regardless of the number of servicing threads, the average number of tracks traversed begins at around 5. This is the result of our input data being random, and it can be shown that the expected distance travelled is 5 when using a hard drive of size 15, as we do in our simulations. When there is only 1 request thread, each request is usually serviced immediately because the act of requesting when there are no requests pending signals the servicing thread to do some work. Since the requests are designed to be random, the average distance (D) travelled is the average distance between any two tracks. This is computed below.

$$
\begin{aligned}
\mathrm{E}(D) &= \sum_{i=0}^{N} D \times \mathrm{P}(D), \text{expectation formula} \\
&= \frac{N}{N^2} + \sum_{i=1}^{N} (N-i) \times \frac{2i}{N^2}, N \text{ ways for positions to be identical} \\
&= \frac{1}{N^2}\left[2N \cdot \frac{N \cdot (N+1)}{2} - 2 \cdot \left(\frac{1}{6} \cdot N \cdot (N+1) \cdot (2N+1)\right) + N\right], \text{using summation formulae} \\
&= N + 1 - \frac{(N+1) \cdot (2N+1)}{3N} + \frac{1}{N}.
\end{aligned}
$$

Plugging in $N = 15$ yields $\mathrm{E}(D) = 5.04$, which is close to the observed value.

3.3 Reversals per Request

Another hardware consideration is the number of times the I/O head will need to switch directions after a request is made and before it is serviced. Just like the distance metric, in real hardware it is expected that instructing the I/O head to reverse direction would take some time, and it may be a desirable quantity to minimize. Figures 3.5 and 3.6 show this quantity with respect to the number of request threads, for the Elevator and SSTF algorithms, respectively. For both algorithms, there is an initial increase in the number of reversals as the number of request threads increase. This again is attributable to a non-empty request queue which is sparse. This initial increase is lost when the number of servicing threads is increased, because the number of pending requests is kept very low.

This is the first metric for which there is a measurable difference in performance between the two scheduling algorithms. When there is only one request thread, the SSTF algorithm

had runs whose *average* number of reversals was greater than one. This means that I/O head had to reverse direction on almost every request, and sometimes more than once. This is definitely not very good behaviour if reversals want to be kept to a minimum. In comparison, the number of reversals for the Elevator don't vary as much, and the average number of reversals rarely were more than 0.8.

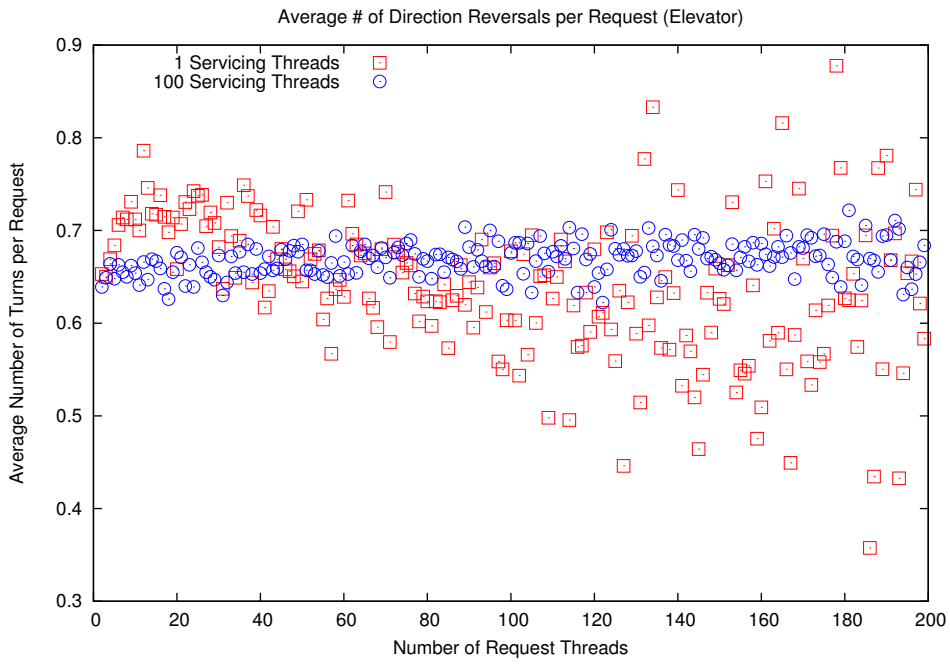

Figure 3.5: Due to the nature of the Elevator algorithm, the number of turns per request is guaranteed never to be more than one.

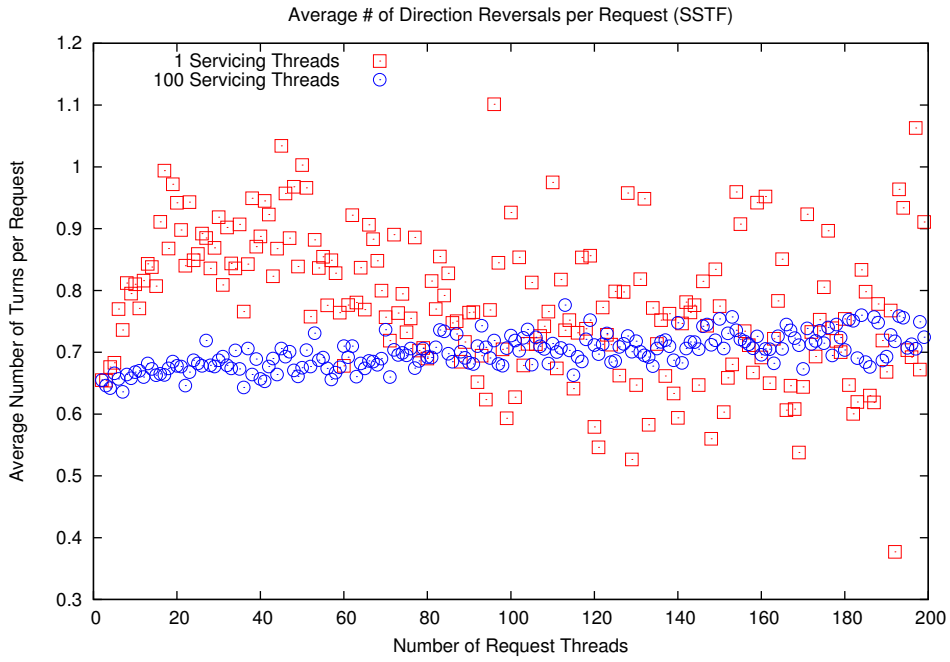

Figure 3.6: The number of reversals when using SSTF can be more than one per request.

4 Known Bugs

4.1 Too Many Threads Segfaulting

If the monitor is run with a large number of request or servicing threads pthreads the program gets frequent segfaults. When trying to debug this we found they are almost always in the list size function but we could not track down what causes the segfault.

5 Listings

5.1 Sample Output

Elevator

```
going to wait
Just pushed track 13 for 100 microseconds
Working on track 13 for 100 micro seconds
going to wait
Just pushed track 15 for 100 microseconds
Just pushed track 15 for 100 microseconds
Just pushed track 12 for 100 microseconds
Just pushed track 2 for 100 microseconds
Just pushed track 15 for 100 microseconds
Working on track 15 for 100 micro seconds
Working on track 15 for 100 micro seconds
Working on track 15 for 100 micro seconds
Working on track 12 for 100 micro seconds
Working on track 2 for 100 micro seconds
going to wait
Just pushed track 6 for 100 microseconds
Working on track 6 for 100 micro seconds
going to wait
Just pushed track 9 for 100 microseconds
Just pushed track 4 for 100 microseconds
Working on track 9 for 100 micro seconds
Working on track 4 for 100 micro seconds
going to wait
Just pushed track 15 for 100 microseconds
Just pushed track 7 for 100 microseconds
Working on track 7 for 100 micro seconds
Working on track 15 for 100 micro seconds
going to wait
Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
going to wait
Just pushed track 1 for 100 microseconds
Working on track 1 for 100 micro seconds
going to wait
Just pushed track 14 for 100 microseconds
Working on track 14 for 100 micro seconds
going to wait
Just pushed track 5 for 100 microseconds
Working on track 5 for 100 micro seconds
going to wait
Just pushed track 3 for 100 microseconds
Working on track 3 for 100 micro seconds
going to wait
Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
going to wait
Just pushed track 6 for 100 microseconds
Working on track 6 for 100 micro seconds
```

```
49 going to wait
   Just pushed track 12 for 100 microseconds
51 Working on track 12 for 100 micro seconds
   going to wait
53 Just pushed track 5 for 100 microseconds
   Just pushed track 10 for 100 microseconds
55 Working on track 10 for 100 micro seconds
   Working on track 5 for 100 micro seconds
57 going to wait
   Just pushed track 9 for 100 microseconds
59 Working on track 9 for 100 micro seconds
   going to wait
61 Just pushed track 2 for 100 microseconds
   Working on track 2 for 100 micro seconds
63 going to wait
   Just pushed track 7 for 100 microseconds
65 Working on track 7 for 100 micro seconds
   going to wait
67 Just pushed track 6 for 100 microseconds
   Working on track 6 for 100 micro seconds
69 going to wait
   Just pushed track 14 for 100 microseconds
71 Working on track 14 for 100 micro seconds
   going to wait
73 Just pushed track 9 for 100 microseconds
   Working on track 9 for 100 micro seconds
75 going to wait
   Just pushed track 14 for 100 microseconds
77 Working on track 14 for 100 micro seconds
   going to wait
79 Just pushed track 4 for 100 microseconds
   Working on track 4 for 100 micro seconds
81 going to wait
   Just pushed track 12 for 100 microseconds
83 Working on track 12 for 100 micro seconds
   going to wait
85 Just pushed track 10 for 100 microseconds
   Working on track 10 for 100 micro seconds
87 going to wait
   Just pushed track 7 for 100 microseconds
89 Working on track 7 for 100 micro seconds
   going to wait
91 Just pushed track 10 for 100 microseconds
   Working on track 10 for 100 micro seconds
93 going to wait
   Just pushed track 4 for 100 microseconds
95 Working on track 4 for 100 micro seconds
   going to wait
97 Just pushed track 10 for 100 microseconds
   Just pushed track 12 for 100 microseconds
99 Working on track 10 for 100 micro seconds
   Working on track 12 for 100 micro seconds
```

sample_elevator_output.txt

SSTF

```
going to wait
Just pushed track 13 for 100 microseconds
Just pushed track 15 for 100 microseconds
Just pushed track 15 for 100 microseconds
Just pushed track 12 for 100 microseconds
Working on track 12 for 100 micro seconds
Working on track 13 for 100 micro seconds
Working on track 15 for 100 micro seconds
Working on track 15 for 100 micro seconds
going to wait
Just pushed track 2 for 100 microseconds
Working on track 2 for 100 micro seconds
going to wait
Just pushed track 9 for 100 microseconds
Just pushed track 15 for 100 microseconds
Just pushed track 6 for 100 microseconds
Just pushed track 4 for 100 microseconds
Working on track 4 for 100 micro seconds
Working on track 6 for 100 micro seconds
Working on track 9 for 100 micro seconds
Working on track 15 for 100 micro seconds
going to wait
Just pushed track 15 for 100 microseconds
Just pushed track 7 for 100 microseconds
Working on track 15 for 100 micro seconds
Just pushed track 10 for 100 microseconds
Just pushed track 1 for 100 microseconds
Working on track 10 for 100 micro seconds
Just pushed track 14 for 100 microseconds
Working on track 7 for 100 micro seconds
Working on track 1 for 100 micro seconds
Working on track 14 for 100 micro seconds
going to wait
Just pushed track 5 for 100 microseconds
Just pushed track 3 for 100 microseconds
Working on track 5 for 100 micro seconds
Working on track 3 for 100 micro seconds
going to wait
Just pushed track 6 for 100 microseconds
Just pushed track 10 for 100 microseconds
Just pushed track 5 for 100 microseconds
Just pushed track 12 for 100 microseconds
Working on track 5 for 100 micro seconds
Working on track 6 for 100 micro seconds
Working on track 10 for 100 micro seconds
Working on track 12 for 100 micro seconds
going to wait
Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
going to wait
Just pushed track 6 for 100 microseconds
Working on track 6 for 100 micro seconds
```

```
    going to wait
 54 Just pushed track 2 for 100 microseconds
    Just pushed track 9 for 100 microseconds
 56 Working on track 9 for 100 micro seconds
    Just pushed track 7 for 100 microseconds
 58 Working on track 7 for 100 micro seconds
    Working on track 2 for 100 micro seconds
 60 going to wait
    Just pushed track 14 for 100 microseconds
 62 Working on track 14 for 100 micro seconds
    going to wait
 64 Just pushed track 9 for 100 microseconds
    Working on track 9 for 100 micro seconds
 66 going to wait
    Just pushed track 14 for 100 microseconds
 68 Just pushed track 10 for 100 microseconds
    Working on track 10 for 100 micro seconds
 70 Working on track 14 for 100 micro seconds
    going to wait
 72 Just pushed track 4 for 100 microseconds
    Just pushed track 12 for 100 microseconds
 74 Just pushed track 7 for 100 microseconds
    Just pushed track 10 for 100 microseconds
 76 Just pushed track 4 for 100 microseconds
    Working on track 12 for 100 micro seconds
 78 Working on track 10 for 100 micro seconds
    Working on track 7 for 100 micro seconds
 80 Just pushed track 10 for 100 microseconds
    Working on track 4 for 100 micro seconds
 82 Just pushed track 12 for 100 microseconds
    Working on track 4 for 100 micro seconds
 84 Working on track 10 for 100 micro seconds
    Working on track 12 for 100 micro seconds
 86 going to wait
    Just pushed track 3 for 100 microseconds
 88 Working on track 3 for 100 micro seconds
    Just pushed track 1 for 100 microseconds
 90 Just pushed track 7 for 100 microseconds
    Working on track 1 for 100 micro seconds
 92 Just pushed track 11 for 100 microseconds
    Working on track 7 for 100 micro seconds
 94 Just pushed track 7 for 100 microseconds
    Working on track 7 for 100 micro seconds
 96 Just pushed track 14 for 100 microseconds
    Just pushed track 13 for 100 microseconds
 98 Just pushed track 13 for 100 microseconds
    Working on track 11 for 100 micro seconds
100 Just pushed track 11 for 100 microseconds
```

sample_SSTF_output.txt