



SIMON FRASER UNIVERSITY

CMPT 300: 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 28, 2011

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.

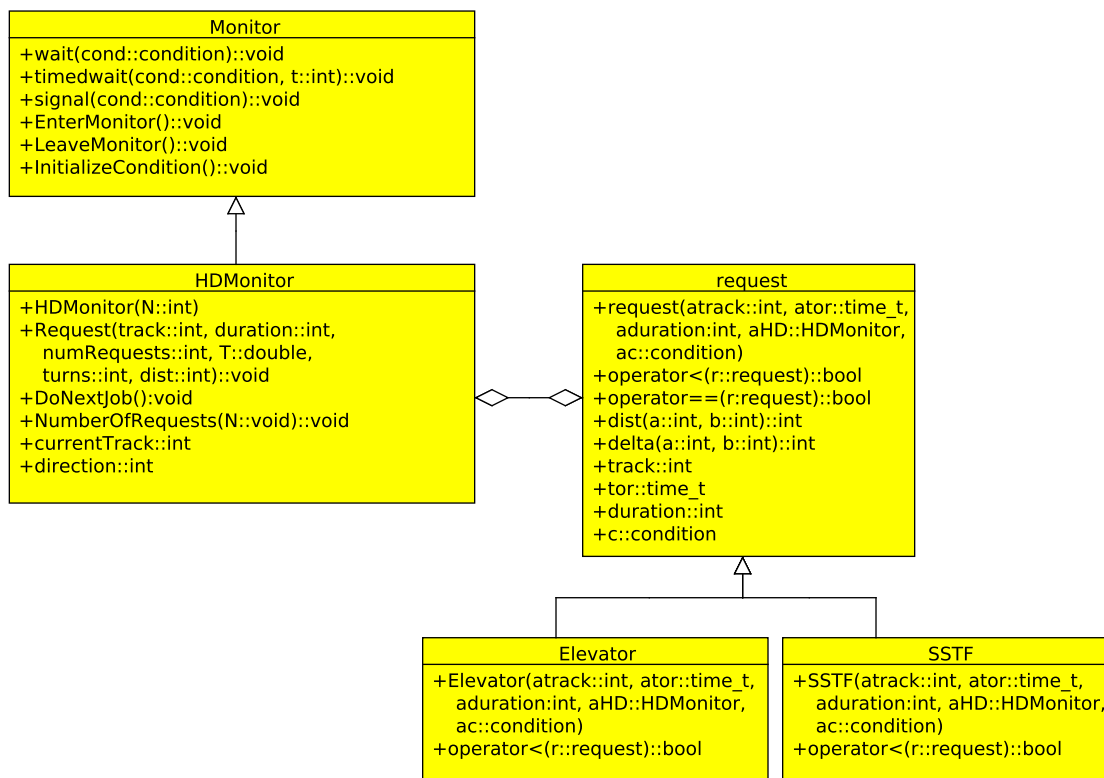


Figure 2.1: UML diagram from the hard drive simulation.

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 listings 4.1 and 4.2.

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 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.

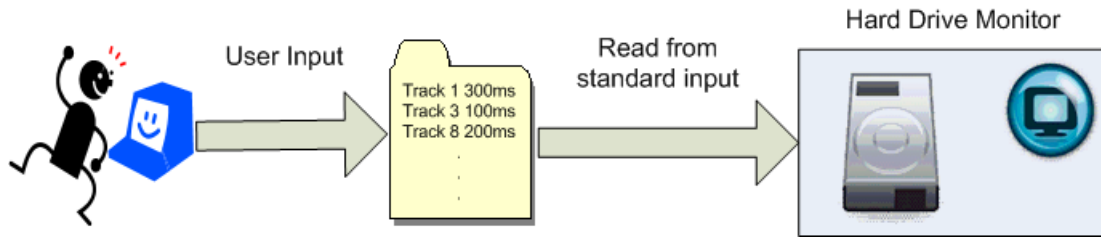


Figure 2.2: Scheduling environment for the hard drive simulation.

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 listing 4.7.

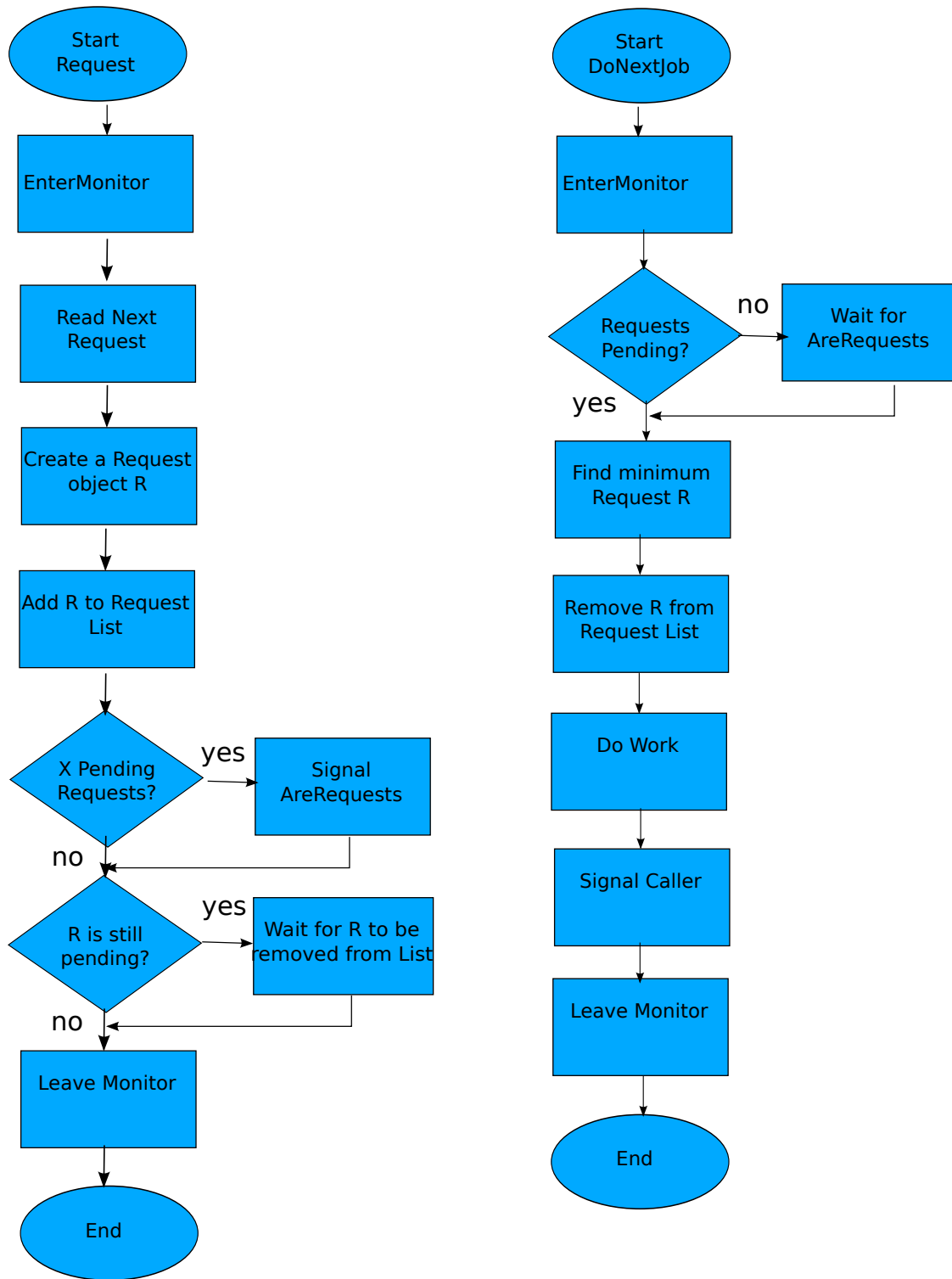


Figure 2.3: Flow of execution for HDMonitor functions responsible for requesting I/O and servicing requests.

In listings 4.8 and 4.8, 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 4.8. 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.

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 unserved 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.

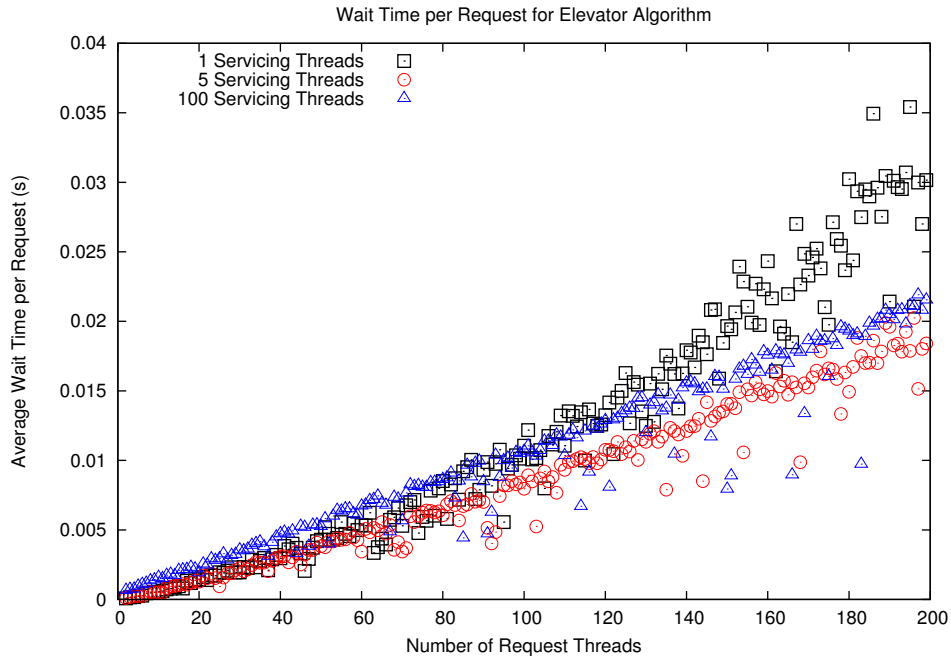


Figure 3.1: The wait time per request is improved by using more than one servicing thread.

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 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

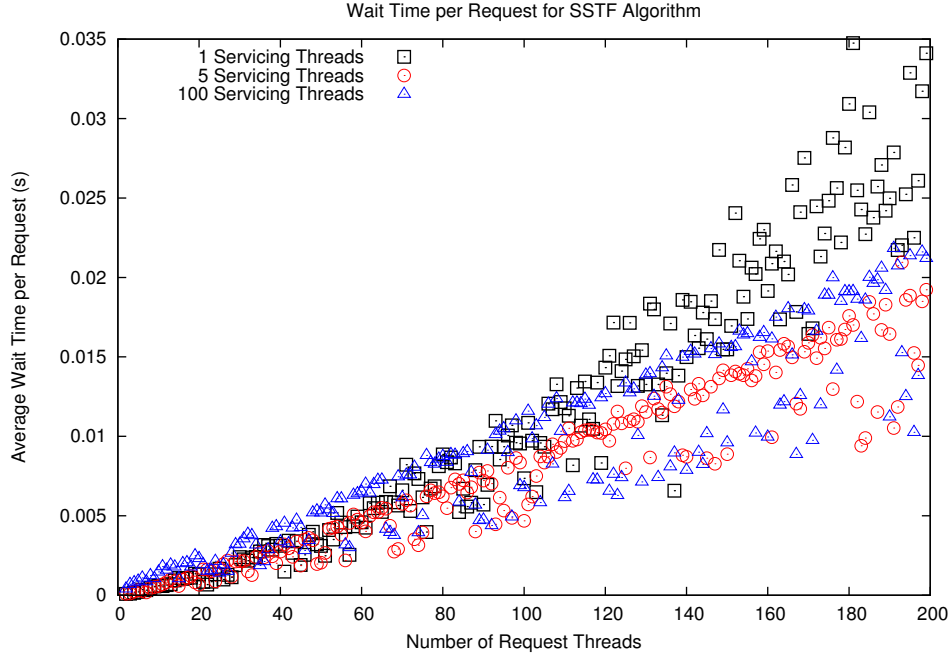


Figure 3.2: The wait time per request is improved by using more than one servicing thread.

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.

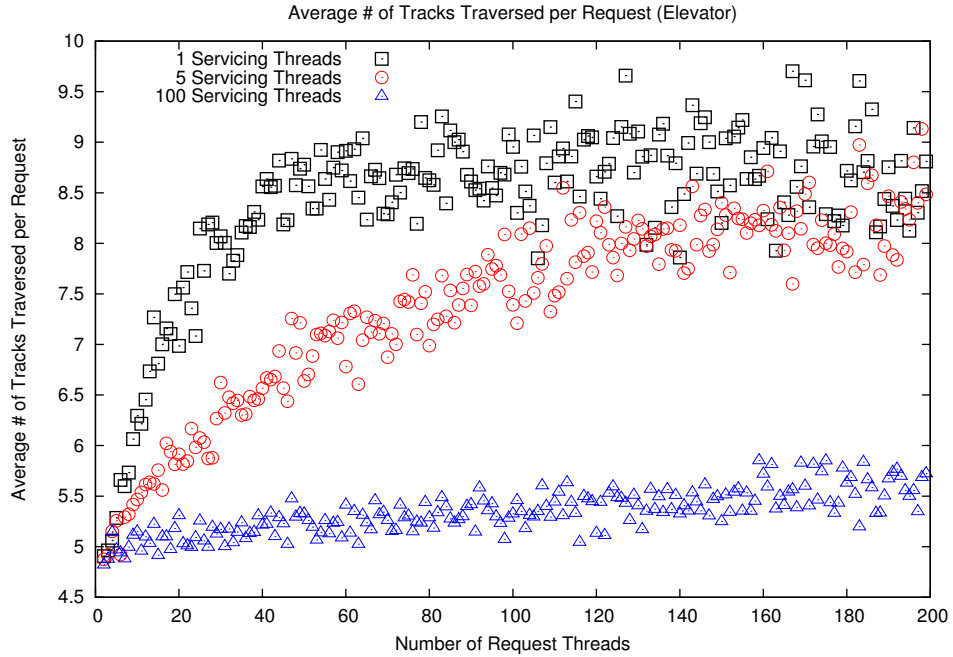


Figure 3.3: The average number of tracks traversed by the I/O head per request can depend significantly on the number of servicing threads in use.

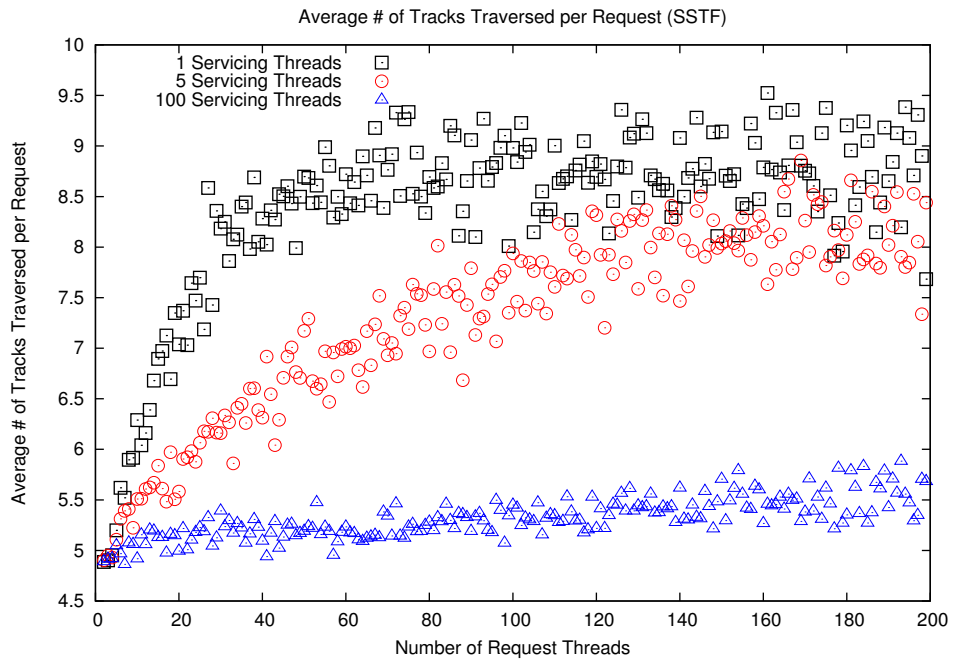


Figure 3.4: Performance of SSTF under the distance metric is comparable to Elevator on random input data.

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.

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

Plugging in $N = 15$ yields $E(D) = 5.04$, 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.

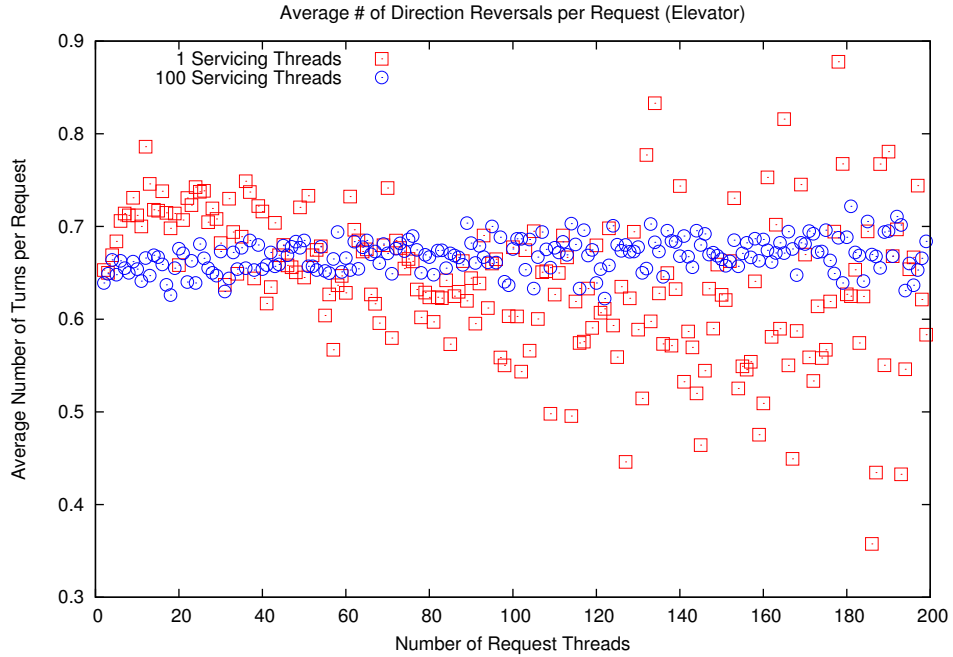


Figure 3.5: Due to the nature of the Elevator algorithm, the number of turns per request is guaranteed never to be more than one.

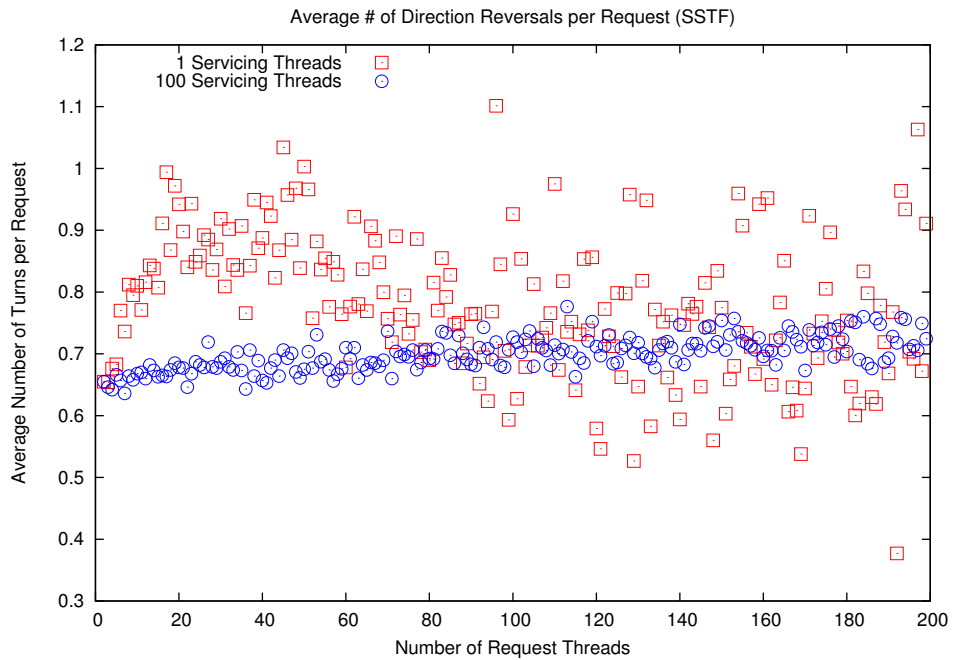


Figure 3.6: The number of reversals when using SSTF can be more than one per request.

4 Listings

4.1 Monitor Interface

```
1  /*
2  *  Monitor.h
3  *  Created by Andrew Inwood on 11/15/11
4  *  Modified by Joel Teichroeb 11/27/11
5  *  for CMPT 300 Project 2.
6  *
7  *  All rights reserved.
8  *
9  *
10 *  Defines the interface for a faked Monitor construct, with synchronization
11 *  of shared resources implemented using the pthreads library.
12 */
13 #ifndef MONITOR.H
14 #define MONITOR.H
15 #include <pthread.h>
16
17 typedef pthread_cond_t *condition;
18
19 class Monitor{
20 public:
21     //Constructor
22     Monitor();
23     void wait(condition &cond);
24     void timedwait(condition &cond, int t);
25     void signal(condition cond);
26     void EnterMonitor();
27     void LeaveMonitor();
28     void InitializeCondition(condition &c);
29 private:
30     /*The occupied mutex ensures that only one process is present in the
31     *mutex at one time.
32     */
33     pthread_mutex_t *occupied;
34 };
35 #endif
```

Monitor.h

4.2 Monitor Implementation

```
1  /*
2  *  Monitor.cpp
3  *
4  *  Created by Andrew Inwood on 11/15/11
5  *  Modified by Joel Teichroeb 11/27/11
6  *  for CMPT 300 Project 2.
7  *
8  *  All rights reserved.
9  *
10 *  Implementation of methods of the Monitor class contained in Monitor.h
11 */
12 #include "Monitor.h"
13 #include <pthread.h>
14 #include <ctime>
15 #include <cstdio>
16 using namespace std;
17 Monitor::Monitor(){
18     occupied = new pthread_mutex_t();
19     pthread_mutex_init(occupied, NULL);
20 }
21 }
```

```

23  /*
   *  InitializeCondition(condition &c).
25  *
   *  Creates and initializes a condition for immediate use.
27  *
   */
29  void Monitor::InitializeCondition(condition &c){
   c = new pthread_cond_t();
31  pthread_cond_init(c, NULL);
   }
33
   /*
35  *  wait(condition cond).
   *
37  *  The executing thread releases the monitor, and is put to sleep until the
   *  condition variable is signalled.
39  *
   */
41  void Monitor::wait(condition &cond){
   int ret;
43  if((ret = pthread_cond_wait(cond, occupied))){
   printf ("wait failed with %d\n", ret);
45  }
   }
47
   /*
49  *  timedwait(condition &cond, int t).
   *
51  *  The executing thread releases the monitor, and is put to sleep until the
   *  condition variable is signalled or t nanoseconds pass.
53  *
   */
55  void Monitor::timedwait(condition &cond, int t){
   struct timespec ts;
57  clock_gettime(CLOCK_REALTIME, &ts);
   ts.tv_nsec = ts.tv_nsec + t;
59  pthread_cond_timedwait(cond, occupied, &ts);
   }
61
   /*
63  *  signal(condition cond).
   *
65  *  Signals all threads waiting on condition variable cond. The system selects
   *  one thread to execute once the Monitor is released.
67  *
   */
69  void Monitor::signal(condition cond){
   pthread_cond_broadcast(cond);
71  }

73  /*
   *  EnterMonitor().
75  *
   *  When EnterMonitor() completes, the calling thread has control of the Monitor.
77  *  This function must be the first line of all functions defined in the Monitor,
   *  and enables faking the Monitor construct in C++.
79  *
   */
81  void Monitor::EnterMonitor(){
   while(pthread_mutex_lock(occupied));
83  }

85  /*
   *  LeaveMonitor().
87  *
   *  When LeaveMonitor() completes, the Monitor is empty.
89  *

```

```

91  */
void Monitor::LeaveMonitor(){
    int ret;
93  if((ret = pthread_mutex_unlock(occupied))){
        printf("Mutex unlock returned an error: %d\n", ret);
95  }
}

```

Monitor.cpp

4.3 Hard Disk Monitor Interface

```

2  /*
   * HDMonitor.h
   *
4  * Created by Andrew Inwood on 11/15/11
   * Modified by Joel Teichroeb 11/27/11
6  * for CMPT 300 Project 2.
   *
8  * All rights reserved.
   *
10 * The HDMonitor provides the interface to a hard drive simulation, consisting
   * of N tracks and a moveable read/write head. Users of the HDMonitor can
12 * request IO operations on individual tracks, and request that a queued IO
   * operation be performed.
14 *
16 * This Monitor schedules the next IO request based on the elevator algorithm.
   */
18
19 #ifndef HD_MONITOR_H
20 #define HD_MONITOR_H
21 #include "Monitor.h"
22 #include <list>           //for request list
23 #include <ctime>          //for request time
24 #include <map>
25 #include "Request.h"
26
27 const int WAITFOR_XREQUESTS = 1;
28 const int WAIT_XNSECONDS = 1000000;
29
30 class RequestWrap {
31 public:
32     request *r;
33     virtual bool operator< (const RequestWrap &o) const {
34         return *r < *(o.r);
35     }
36     bool operator==(const request *o) const{
37         return r == o;
38     }
39 };
40
41 typedef std::list<RequestWrap> RequestList;
42
43 /*
44  * HDMonitor (Hard Drive Monitor)
45  */
46 class HDMonitor : protected Monitor{
47 public:
48     condition areRequests; //Allows the scheduler to wait for requests
49     //Constructors
50     HDMonitor(int N);
51
52     /*
53      * Request(int track, int duration)
54      * Creates a new request to the hard drive, and is put on the queue for

```

```

    * scheduling.
56 */
    void Request(int track, int duration, int &numRequests, double &T, int &turns,
58 int &dist);
    /*
60 * DoNextJob().
    * Selects the next job from the list based on the scheduling rule.
62 *
    */
64 void DoNextJob();
    void NumberOfRequests(int &N);
66 int currentTrack; //track that read/write head is on [1, N]
    int direction; //direction of read/write head {-1, 1}
68 private:
    int numWaitingToWork;
70 int numTracks; // Equal to N
    RequestList jobsList;
72 /*Attributes for Testing*/
    int numTurns;
74 int distance;
    std::map<request*, int> NumAtRequestComplete; // counts pending requests
76 std::map<request*, int> numTurnsMap; //counts reversals
    std::map<request*, int> distanceMap; //counts tracks traversed
78 };
#endif

```

HDMonitor.h

4.4 Hard Disk Monitor Implementation

```

1 /*
    * HDMonitor.cpp
3 *
    * Created by Andrew Inwood on 11/15/11
5 * Modified by Joel Teichroeb 11/27/11
    * for CMPT 300 Project 2.
7 *
    * All rights reserved.
9 *
    * The HDMonitor provides the interface to a hard drive simulation, consisting
11 * of N tracks and a moveable read/write head. Users of the HDMonitor can
    * request IO operations on individual tracks, and request that a queued IO
13 * operation be performed.
    *
15 * This Monitor schedules the next IO request based on the elevator algorithm.
    *
17 */

19 #include "Monitor.h"
    #include "HDMonitor.h"
21 #include <algorithm> //for min_element
    #include <cstdio> //for output messages
23 #include <list> //for request list
    #include <ctime> //for request time
25 #include <unistd.h>

27 #include "SSTF.h"
    #include "Elevator.h"
29
    using namespace std;
31
    /*
33 * HDMonitor (Hard Drive Monitor)
    *
35 */
    HDMonitor::HDMonitor(int N){

```

```

37     numWaitingToWork = 0;
38     direction = 1;
39     currentTrack = 1;
40     numTracks = N;
41     //InitializeCondition(areRequests);
42     areRequests = new pthread_cond_t();
43     pthread_cond_init(areRequests, NULL);
44     numTurns = 0;
45     distance = 0;
46 }
47
48 /*
49  * Request(int track, int duration)
50  * Creates a new request to the hard drive, and is put on the queue for
51  * scheduling.
52  */
53 void HDMonitor::Request(int track, int duration, int &numRequests, double &T, int
&turns, int &dist){
54     EnterMonitor();
55     T = clock()/(double)CLOCKS_PER_SEC; //Test data
56     int startTurns = numTurns;
57     int startDistance = distance;
58
59     condition c;
60     InitializeCondition(c);
61     //create Request object
62     request *r = new Elevator(track, time(NULL), duration, this, c);
63     RequestWrap wrap;
64     wrap.r = r;
65     jobsList.push_back(wrap); //Add request to list
66     NumAtRequestComplete.insert(pair<request*,int>(r, 0));
67     //printf("The size was %d\n", jobsList->size() +1);
68     printf("Just pushed track %d for %d microseconds\n", track, duration);
69     if(numWaitingToWork && jobsList.size() >= WAIT_FOR_X_REQUESTS){
70         signal(areRequests);
71     }
72     while(find(jobsList.begin(), jobsList.end(), r) !=
73         jobsList.end()){
74         timedwait(c, WAIT_X_NSECONDS);
75     }
76     //update test data
77     T = (clock()/(double)CLOCKS_PER_SEC) - T;
78     dist = distanceMap[r] - startDistance;
79     turns = numTurnsMap[r] - startTurns;
80     distanceMap.erase(r);
81     numTurnsMap.erase(r);
82
83     //printf("The size is %d\n", jobsList->size() +1);
84     numRequests = NumAtRequestComplete[r];
85     NumAtRequestComplete.erase(r);
86     delete r;
87     LeaveMonitor();
88 }
89
90 /*
91  * DoNextJob().
92  * Selects the next job from the list based on the scheduling rule.
93  *
94  */
95 void HDMonitor::DoNextJob(){
96     EnterMonitor();
97     if(!jobsList.size()) { //wait until requests are available
98         ++numWaitingToWork;
99         //The loop is necessary, or else pthreads will wake up a thread
100         //that has been inactive for some time.
101         while(!jobsList.size()){
102             printf("going to wait\n");
103             //wait(areRequests);
104             timedwait(areRequests, WAIT_X_NSECONDS);

```

```

105     }
106     --numWaitingToWork;
107 }
108 //get next job
109 RequestList::iterator nextRequest = min_element(jobsList.begin(), jobsList.end());
110 request *r = nextRequest->r;
111 //change direction if necessary
112 if((direction == -1 && r->track > currentTrack) ||
113     (direction == 1 && r->track < currentTrack)) {
114     direction *= -1;
115     nextRequest = min_element(jobsList.begin(), jobsList.end());
116     r = nextRequest->r;
117 ++numTurns;
118 //printf("Have turned %ld times.\n", numTurns);
119 }
120 int delta = r->track - currentTrack;
121 if(delta > 15) {
122     //printf("Found delta with %d. Next track at $d, current track at $d\n", delta,
123     //r->track, currentTrack);
124 }
125 distance += (delta > 0) ? delta : -1*delta;
126 currentTrack = r->track;
127 printf("Working on track %d for %d micro seconds\n", r->track, r->duration);
128 //printf("Have travelled %ld\n", distance);
129 NumAtRequestComplete[r] = jobsList.size();
130 int sleepytime = r->duration;
131 jobsList.erase(nextRequest);
132 distanceMap.insert(pair<request*,int>(r, distance));
133 numTurnsMap.insert(pair<request*,int>(r, numTurns));
134 signal(r->c);
135 usleep(1);
136 //usleep(sleepytime); //Do some "work"
137 LeaveMonitor();
138 }
139
140 /*
141 * NumberOfRequests()
142 * Returns the number of requests pending
143 *
144 */
145 void HDMonitor::NumberOfRequests(int & N){
146     EnterMonitor();
147     N = jobsList.size();
148     LeaveMonitor();
149 }

```

HDMonitor.cpp

4.5 Request Class Definition

```

1 #ifndef REQUEST_H
2 #define REQUEST_H
3
4 #include <ctime>
5 #include "Monitor.h"
6
7 class HDMonitor;
8
9 /*
10 * request
11 * The HDMonitor encapsulates an IO request as an instance of the request class,
12 * and maintains a list of these objects.
13 *
14 */
15 class request {
16 public:

```



```

17     int track;
18     time_t tor; //Time of Request
19     int duration;
20     condition &c;
21     static int dist(int a, int b){ return a < b ? b - a : a - b;}
22     static int delta(int a, int b) { return b - a;}
23     request(int atrack, time_t ator, int aduration, HDMonitor *aHD, condition &ac):c (ac
24     ){
25         track = atrack;
26         tor = ator;
27         duration = aduration;
28         H = aHD; //The comparison operator needs the currentTrack from H
29     }
30     /*
31      * operator<
32      * This is required to find the next request.
33      */
34     virtual bool operator< (const request & r) const = 0;
35     bool operator==(const request & r) const{
36         return track == r.track && tor == r.tor && duration == r.duration && H
37         == r.H && c == r.c;
38     }
39     protected:
40         HDMonitor *H;
41 };
42
43 #endif

```

Request.h

4.6 Scheduling Algorithm Implementations

The scheduling algorithm is implemented by defining an order on the Request objects in the HDMonitor class. The order is defined by implementing the less-than operator(<), and this is done in separate files for each algorithm and linked at compile time.

Elevator Algorithm

```

1 /*
2  * Elevator.cpp
3  *
4  * Created by Joel Teichroeb on 11/25/11
5  * for CMPT 300 Project 2.
6  *
7  * All rights reserved.
8  *
9  * The Elevator class as defined in Elevator.h.
10  *
11  */
12 #include "Elevator.h"
13 #include "HDMonitor.h"
14
15 /*
16  * Implement the < operator for requests.
17  */
18 bool Elevator::operator< (const request &r) const{
19     switch(H->direction){
20     case 1:
21         return ( request::dist(H->currentTrack, track) < request::dist(H->currentTrack,
22         r.track) &&
23         request::delta(H->currentTrack, track) >= 0 ) ||
24         ( track == r.track && request::delta(H->currentTrack, track) >= 0 &&
25         tor < r.tor);
26     }
27     break;
28 }

```

```

26         case -1:
            return ( request::dist(H->currentTrack, track) < request::dist(H->currentTrack,
28                    r.track) &&
                    request::delta(H->currentTrack, track) <= 0) ||
                    ( track == r.track && request::delta(H->currentTrack, track) <= 0 &&
30                      tor < r.tor);
            break;
32         default:
            return false;
34         break;
        }
36 }

```

Elevator.cpp

Shortest Seek-Time First Algorithm

```

2  /*
   * SSTF.cpp
   *
   * Created by Joel Teichroeb on 11/25/11
   * for CMPT 300 Project 2.
   *
   * All rights reserved.
   *
   * The SSTF class as defined in SSTF.h.
10  */
12 #include "SSTF.h"
13 #include "HDMonitor.h"
14
15 /*
16  * Implement the < operator for requests.
17  */
18 bool SSTF::operator< (const request &r) const{
19     return dist(H->currentTrack, track) < dist(H->currentTrack, r.track);
20 }

```

SSTF.cpp

4.7 Main Program

```

1  /*
   * MonitorDriver.cpp
   *
   * Created by Andrew Inwood on 11/15/11
   * Modified by Joel Teichroeb 11/27/11
   * for CMPT 300 Project 2.
   *
   * All rights reserved.
   *
   *
11  */
13
14 #include "HDMonitor.h"
15 #include <cstdio>
16 #define NUMTRACKS 15
17 #include <pthread.h>
18 #include <cstdlib>
19 #include <ctime>
20 #include <unistd.h>
21 #include <map>
22 #define WAIT_TIME 1

```

```

23 static int NUMTHREADS = 100;
   static int NUMWORKTHREADS = 1;
24 using namespace std;

25
26 // The mutex used so that only one thread calls scanf at a time
   pthread_mutex_t readmutex;
27
28
29 void *Schedule(void* Mon);
   void *DoNext(void* Mon);
30 int main(int argc, char *argv[]) {
31     if (argc > 2) {
32         NUMTHREADS = atoi(argv[1]); //Number of threads are command line arguments
33         NUMWORKTHREADS = atoi(argv[2]);
34     }
35     HDMonitor M(NUMTRACKS);
36     pthread_t threads[NUMTHREADS + NUMWORKTHREADS];
37     int rc;
38     long t;
39     void* status;
40     pthread_mutex_init(&readmutex, 0);
41
42     // Create the worker threads
43     for (t=0; t < NUMWORKTHREADS; t++){
44         rc = pthread_create(&threads[t], NULL, DoNext, &M);
45         if (rc){
46             printf("ERROR; return code from pthread_create() is %d\n", rc);
47             exit(-1);
48         }
49     }
50
51     // Create the request threads
52     for (t=NUMWORKTHREADS; t<NUMTHREADS + NUMWORKTHREADS; t++){
53         rc = pthread_create(&threads[t], NULL, Schedule, &M);
54         if (rc){
55             printf("ERROR; return code from pthread_create() is %d\n", rc);
56             exit(-1);
57         }
58     }
59
60     if (rc){
61         printf("ERROR; return code from pthread_create() is %d\n", rc);
62         exit(-1);
63     }
64
65     // Wait for all the request threads to finish before exiting the program
66     for (t = NUMWORKTHREADS; t < NUMTHREADS + NUMWORKTHREADS; ++t){
67         rc = pthread_join(threads[t], &status);
68     }
69     return 0;
70 }
71
72 void *Schedule(void *Mon){
73     int track, duration;
74     FILE* TimingFP = fopen("TimePerRequest.txt", "w");
75     fclose(TimingFP);
76     HDMonitor* M = (HDMonitor*) Mon;
77     while(true){
78         pthread_mutex_lock(&readmutex);
79         int ret = scanf("%d %d", &track, &duration);
80         pthread_mutex_unlock(&readmutex);
81         if (ret == 2){
82             clock_t start = clock();
83             int N; //These variables will hold test data
84             double T;
85             int turns, distance;
86             M->Request(track, duration, N, T, turns, distance);
87             TimingFP = fopen("TimePerRequest.txt", "a");
88             fprintf(TimingFP, "%d %16.14f %d %d\n", N, T, turns, distance);

```

```

91         fclose(TimingFP);
92         usleep(rand() % WAIT_TIME);
93     } else if (ret == EOF) {
94         // if scanf outputs that there is no more data left stop the thread
95         break;
96     }
97 }
98 }
99 }
101 void* DoNext(void* Mon){
102     HDMonitor* M = (HDMonitor*) Mon;
103     while(true){
104         M->DoNextJob(); //Asks for a request to be serviced indefinitely
105     }
106 }

```

MonitorDriver.cpp

4.8 Sample Output

Elevator

```

going to wait
2 Just pushed track 13 for 100 microseconds
Working on track 13 for 100 micro seconds
4 going to wait
Just pushed track 15 for 100 microseconds
6 Just pushed track 15 for 100 microseconds
Just pushed track 12 for 100 microseconds
8 Just pushed track 2 for 100 microseconds
Just pushed track 15 for 100 microseconds
10 Working on track 15 for 100 micro seconds
Working on track 15 for 100 micro seconds
12 Working on track 15 for 100 micro seconds
Working on track 12 for 100 micro seconds
14 Working on track 2 for 100 micro seconds
going to wait
16 Just pushed track 6 for 100 microseconds
Working on track 6 for 100 micro seconds
18 going to wait
Just pushed track 9 for 100 microseconds
20 Just pushed track 4 for 100 microseconds
Working on track 9 for 100 micro seconds
22 Working on track 4 for 100 micro seconds
going to wait
24 Just pushed track 15 for 100 microseconds
Just pushed track 7 for 100 microseconds
26 Working on track 7 for 100 micro seconds
Working on track 15 for 100 micro seconds
28 going to wait
Just pushed track 10 for 100 microseconds
30 Working on track 10 for 100 micro seconds
going to wait
32 Just pushed track 1 for 100 microseconds
Working on track 1 for 100 micro seconds
34 going to wait
Just pushed track 14 for 100 microseconds
36 Working on track 14 for 100 micro seconds
going to wait
38 Just pushed track 5 for 100 microseconds
Working on track 5 for 100 micro seconds
40 going to wait
Just pushed track 3 for 100 microseconds
42 Working on track 3 for 100 micro seconds
going to wait

```

```

44 Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
46 going to wait
Just pushed track 6 for 100 microseconds
48 Working on track 6 for 100 micro seconds
going to wait
50 Just pushed track 12 for 100 microseconds
Working on track 12 for 100 micro seconds
52 going to wait
Just pushed track 5 for 100 microseconds
54 Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
56 Working on track 5 for 100 micro seconds
going to wait
58 Just pushed track 9 for 100 microseconds
Working on track 9 for 100 micro seconds
60 going to wait
Just pushed track 2 for 100 microseconds
62 Working on track 2 for 100 micro seconds
going to wait
64 Just pushed track 7 for 100 microseconds
Working on track 7 for 100 micro seconds
66 going to wait
Just pushed track 6 for 100 microseconds
68 Working on track 6 for 100 micro seconds
going to wait
70 Just pushed track 14 for 100 microseconds
Working on track 14 for 100 micro seconds
72 going to wait
Just pushed track 9 for 100 microseconds
74 Working on track 9 for 100 micro seconds
going to wait
76 Just pushed track 14 for 100 microseconds
Working on track 14 for 100 micro seconds
78 going to wait
Just pushed track 4 for 100 microseconds
80 Working on track 4 for 100 micro seconds
going to wait
82 Just pushed track 12 for 100 microseconds
Working on track 12 for 100 micro seconds
84 going to wait
Just pushed track 10 for 100 microseconds
86 Working on track 10 for 100 micro seconds
going to wait
88 Just pushed track 7 for 100 microseconds
Working on track 7 for 100 micro seconds
90 going to wait
Just pushed track 10 for 100 microseconds
92 Working on track 10 for 100 micro seconds
going to wait
94 Just pushed track 4 for 100 microseconds
Working on track 4 for 100 micro seconds
96 going to wait
Just pushed track 10 for 100 microseconds
98 Just pushed track 12 for 100 microseconds
Working on track 10 for 100 micro seconds
100 Working on track 12 for 100 micro seconds

```

sample_elevator_output.txt

SSTF

```

going to wait
2 Just pushed track 13 for 100 microseconds
Just pushed track 15 for 100 microseconds
4 Just pushed track 15 for 100 microseconds

```

```

Just pushed track 12 for 100 microseconds
6 Working on track 12 for 100 micro seconds
Working on track 13 for 100 micro seconds
8 Working on track 15 for 100 micro seconds
Working on track 15 for 100 micro seconds
10 going to wait
Just pushed track 2 for 100 microseconds
12 Working on track 2 for 100 micro seconds
going to wait
14 Just pushed track 9 for 100 microseconds
Just pushed track 15 for 100 microseconds
16 Just pushed track 6 for 100 microseconds
Just pushed track 4 for 100 microseconds
18 Working on track 4 for 100 micro seconds
Working on track 6 for 100 micro seconds
20 Working on track 9 for 100 micro seconds
Working on track 15 for 100 micro seconds
22 going to wait
Just pushed track 15 for 100 microseconds
24 Just pushed track 7 for 100 microseconds
Working on track 15 for 100 micro seconds
26 Just pushed track 10 for 100 microseconds
Just pushed track 1 for 100 microseconds
28 Working on track 10 for 100 micro seconds
Just pushed track 14 for 100 microseconds
30 Working on track 7 for 100 micro seconds
Working on track 1 for 100 micro seconds
32 Working on track 14 for 100 micro seconds
going to wait
34 Just pushed track 5 for 100 microseconds
Just pushed track 3 for 100 microseconds
36 Working on track 5 for 100 micro seconds
Working on track 3 for 100 micro seconds
38 going to wait
Just pushed track 6 for 100 microseconds
40 Just pushed track 10 for 100 microseconds
Just pushed track 5 for 100 microseconds
42 Just pushed track 12 for 100 microseconds
Working on track 5 for 100 micro seconds
44 Working on track 6 for 100 micro seconds
Working on track 10 for 100 micro seconds
46 Working on track 12 for 100 micro seconds
going to wait
48 Just pushed track 10 for 100 microseconds
Working on track 10 for 100 micro seconds
50 going to wait
Just pushed track 6 for 100 microseconds
52 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

Contents

1	Introduction	1
2	High-level Design	1
	2.1 Monitor Design	1
	2.2 Hard Drive Simulator Design	2
	2.3 Hard Drive Simulation Experiment	2
3	Analysis	5
	3.1 Time to Service a Request	5
	3.2 Distance per Request	6
	3.3 Reversals per Request	8
4	Listings	10
	4.1 Monitor Interface	10
	4.2 Monitor Implementation	10
	4.3 Hard Disk Monitor Interface	12
	4.4 Hard Disk Monitor Implementation	13
	4.5 Request Class Definition	15
	4.6 Scheduling Algorithm Implementations	16
	4.7 Main Program	17
	4.8 Sample Output	19