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 IO requests from multiple users.

2 High-level Design

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 IO 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 3.1, 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. IO requests from anywhere on the disk.

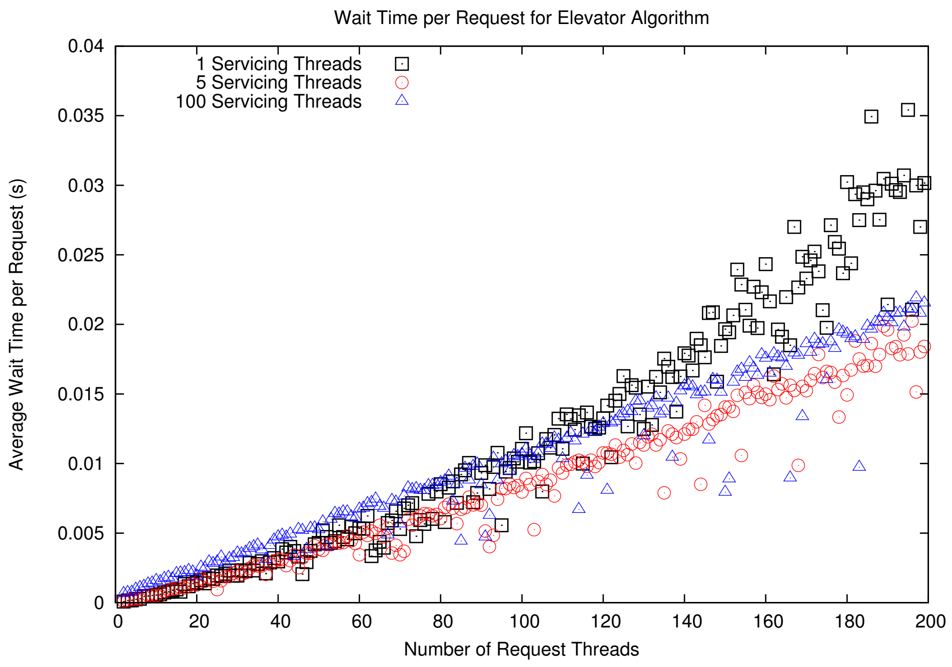

Figure 1: The wait time per request is improved by using more than one servicing thread.

From the graphs, we can make a few observations. Firstly, we can see that the performance of both algorithms are comparable using this metric, on this input data. This happens because the behaviour of the two algorithms are 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 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 a small number of servicing threads, but more than one.

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 IO 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 scans the disk back and forth. Unfortunately (or perhaps fortunately!) it is very difficult to construct a test

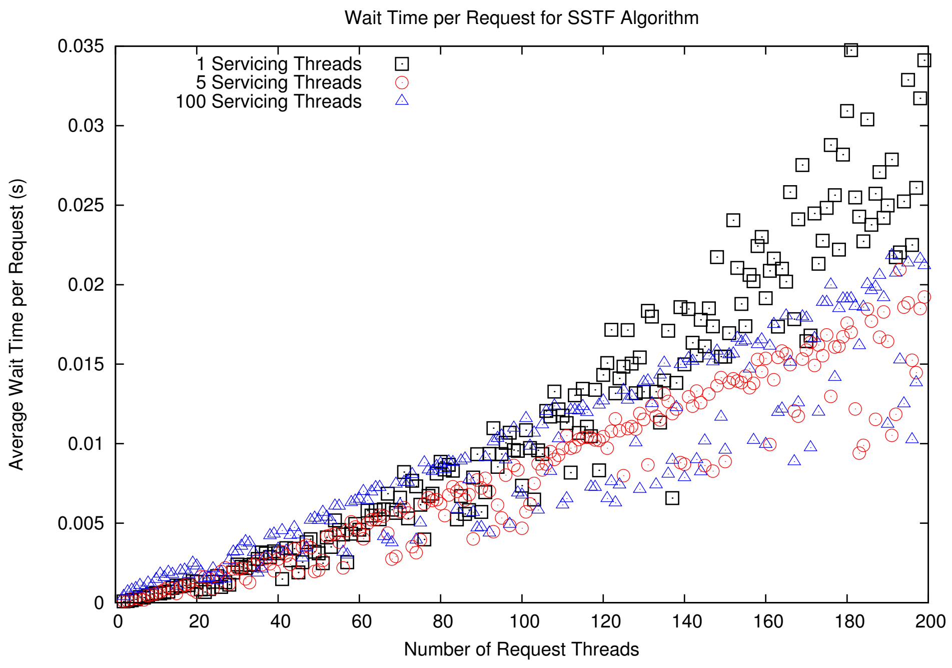

Figure 2: The wait time per request is improved by using more than one servicing thread.

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 on the time to service a request, the analysis assumed that all jobs took the same amount of time to execute, and so the time 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 IO head will require some time to move to the desired track, and so this might 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 3.2 and 3.2 we show this quantity with respect to the number of request threads in the simulation, for the Elevator and SSTF algorithms, respectively.

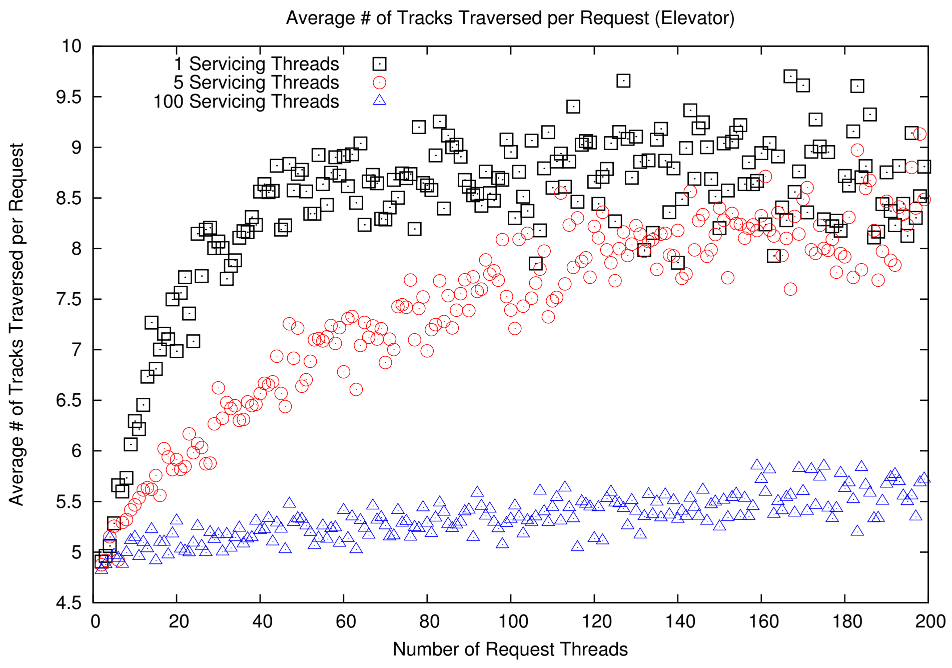

Figure 3: The average number of tracks traversed by the IO head per request can depend significantly on the number of servicing threads in use.

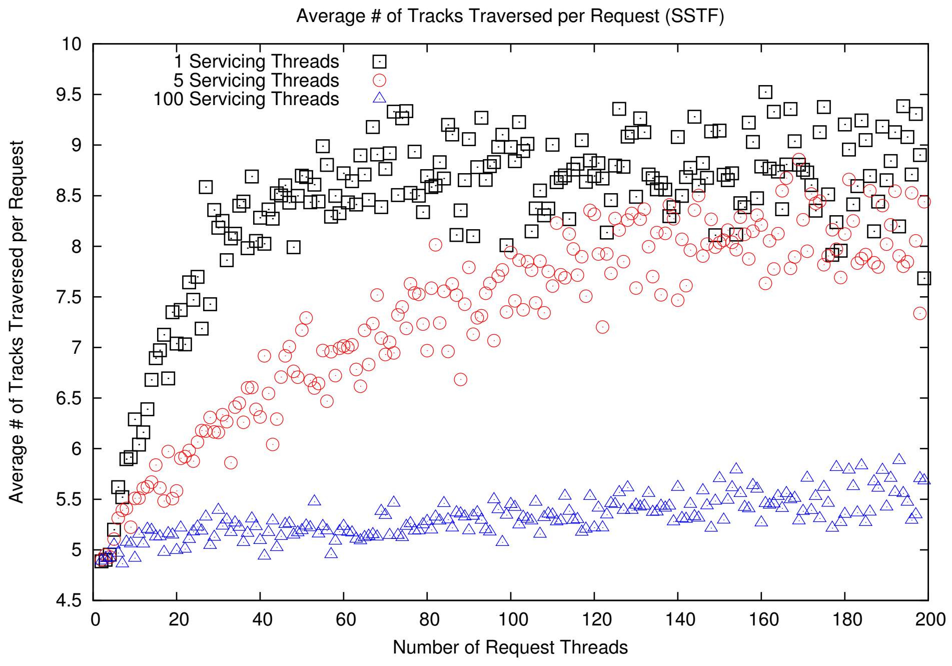

Figure 4: Performans of SSTF under the distance metric is comparable to Elevator on random input data.

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 IO 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 IO 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 IO head to reverse direction would take some time, and it may be a desirable quantity to minimize. Figures 3.3 and 3.3 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. When there is only one request thread, the SSFT algorithm had runs whose *average* number of reversals was greater than one. This means that IO head had to reverse direction on almost every request, and sometimes more than once. This is definitely not very good behaviour if reverals 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 reverals rarely were more than 0.8.

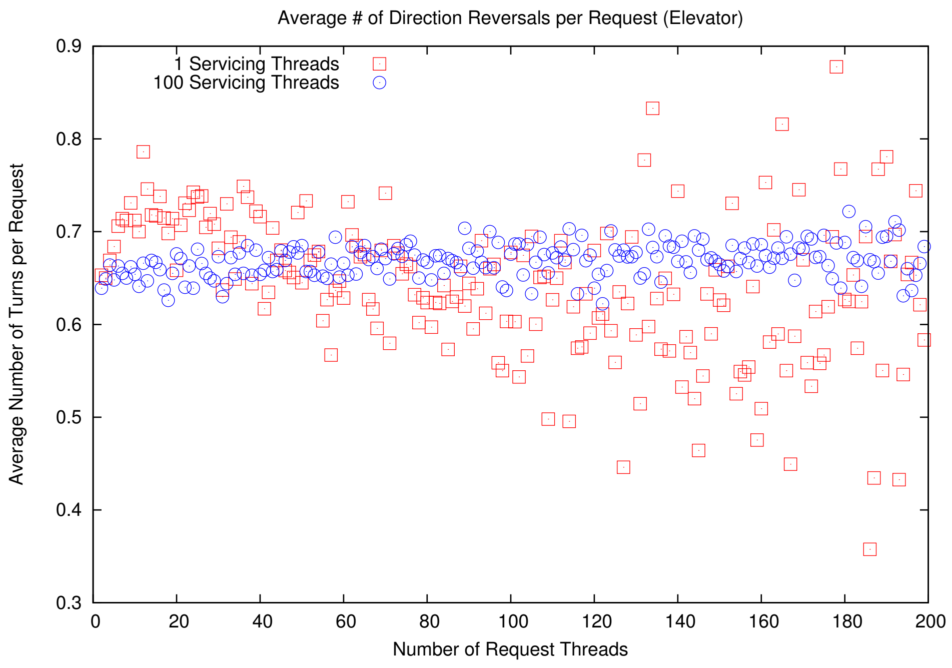

Figure 5: Due to the nature of the Elevator algorithm, the number of turns per request is guaranteed never to be more than one.

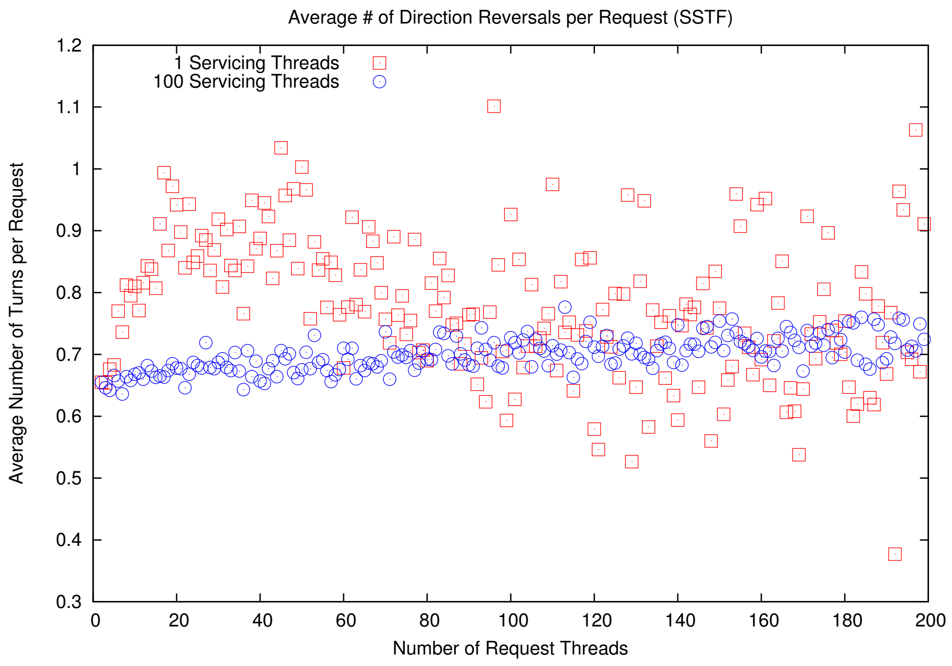

Figure 6: The number of reversals when using SSTF can be more than one per request.

4 Listings

4.1 Monitor Interface

```
/*
 * Monitor.h
 * Created by Andrew Inwood on 11/15/11
 * for CMPT 300 Project 2.
 *
 * All rights reserved.
 *
 *
 * Defines the interface for a faked Monitor construct, with synchronization
 * of shared resources implemented using the pthreads library.
 */
#ifndef MONITOR_H
#define MONITOR_H
#include <pthread.h>
#include <map>
typedef pthread_cond_t *condition;
class Monitor{
    public:
        //Constructor
        Monitor();
        void wait(condition &cond);
        void timedwait(condition &cond, int t);
        void signal(condition cond);
        void EnterMonitor();
        void LeaveMonitor();
        void InitializeCondition(condition &c);
    protected:
        /*The pthreads implementation of condition variables,
         *pthread_cond_t, requires that each condition variable be
         *associated with a mutex. This association is kept track of using
         *a map.
         */
    //    std::map<condition, pthread_mutex_t*> condMutexes;
    private:
        /*The occupied mutex ensures that only one process is present in the
         *mutex at one time.
         */
        pthread_mutex_t *occupied;
};
#endif
```

Monitor.h

4.2 Monitor Implementation

```
/*
 * Monitor.cpp
 *
 * Created by Andrew Inwood on 11/15/11
 * for CMPT 300 Project 2.
 *
 * All rights reserved.
 *
 * Implementation of methods of the Monitor class contained in Monitor.h
 *
 */
#include "Monitor.h"
#include <pthread.h>
#include <ctime>
#include <cstdio>
using namespace std;
```

```
Monitor::Monitor(){
    occupied = new pthread_mutex_t();
    pthread_mutex_init(occupied, NULL);
}
void Monitor::InitializeCondition(condition &c){
    c = new pthread_cond_t();
    pthread_cond_init(c, NULL);
}

/*
 * wait(condition cond).
 *
 * The executing thread releases the monitor, and is put to sleep until the
 * condition variable is signalled. The first time a condition variable is
 * passed to wait, it is initialized and associated with a mutex.
 */
void Monitor::wait(condition &cond){
    int ret;
    if((ret = pthread_cond_wait(cond, occupied))){
        printf ("wait failed with %d\n", ret);
    }
}
void Monitor::timedwait(condition &cond, int t){
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_nsec = ts.tv_nsec + t;
    pthread_cond_timedwait(cond, occupied, &ts);
}

/*
 * signal(condition cond).
 * Signals all threads waiting on condition variable cond. The system selects
 * one thread to execute once the Monitor is released.
 *
 */
void Monitor::signal(condition cond){
    //pthread_mutex_lock(condMutexes[cond]);
    pthread_cond_broadcast(cond);
    //pthread_mutex_unlock(&occupied);
    //pthread_mutex_unlock(condMutexes[cond]);
}

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

/*
 * LeaveMonitor().
 * When LeaveMonitor() completes, the Monitor is empty.
 *
 */
void Monitor::LeaveMonitor(){
    int ret;
    if((ret = pthread_mutex_unlock(occupied))){
        printf("Mutex unlock returned an error: $d\n", ret);
    }
}
```

Monitor.cpp

4.3 Hard Disk Monitor Interface

```
/*
 * HDMonitor.h
 *
 * Created by Andrew Inwood on 11/15/11
 * for CMPT 300 Project 2.
 *
 * All rights reserved.
 *
 * 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
 * request IO operations on individual tracks, and request that a queued IO
 * operation be performed.
 *
 * This Monitor schedules the next IO request based on the elevator algorithm.
 *
 */

#ifndef HD_MONITOR_H
#define HD_MONITOR_H
#include "Monitor.h"
#include <list>       //for request list
#include <ctime>      //for request time
#include <map>
#include "Request.h"

const int WAIT_FOR_X_REQUESTS = 1;
const int WAIT_X_NSECONDS = 1000000;

class RequestWrap {
  public:
    request *r;
            virtual bool operator< (const RequestWrap &o) const {
      return *r < *(o.r);
    }
    bool operator==(const request *o) const{
      return r == o;
    }
};

typedef std::list<RequestWrap> RequestList;
/*
 * HDMonitor (Hard Drive Monitor)
 *
 */
class HDMonitor : protected Monitor{
    public:
        condition areRequests; //Allows the scheduler to wait for requests
        //Constructors
        HDMonitor(int N);
        ~HDMonitor();

        /*
         * Request(int track, int duration)
         * Creates a new request to the hard drive, and is put on the queue for
         * scheduling.
         */
  void Request(int track, int duration, int &numRequests, double &T, int & turns,
    int & dist);
        /*
         * DoNextJob().
         * Selects the next job from the list based on the scheduling rule.
         *
         */
        void DoNextJob();
        void NumberOfRequests(int &N);
        int currentTrack; //track that read/write head is on [1, N]
        int direction; //direction of read/write head {-1, 1}
    private:
        int numWaitingToWork;
        int numTracks; // Equal to N
        RequestList jobsList;
        std::map<request*, int> NumAtRequestComplete;
        int numTurns;
        int distance;
        std::map<request*, int> numTurnsMap;
        std::map<request*, int> distanceMap;
};
#endif
```

HDMonitor.h

4.4 Hard Disk Monitor Implementation

```
/*
 * HDMonitor.cpp
 *
 * Created by Andrew Inwood on 11/15/11
 * for CMPT 300 Project 2.
 *
 * All rights reserved.
 *
 * 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
 * request IO operations on individual tracks, and request that a queued IO
 * operation be performed.
 *
 * This Monitor schedules the next IO request based on the elevator algorithm.
 *
 */

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

#include "SSTF.h"
#include "Elevator.h"

using namespace std;

/*
 * HDMonitor (Hard Drive Monitor)
 *
 */
HDMonitor::HDMonitor(int N){
    numWaitingToWork = 0;
    direction = 1;
    currentTrack = 1;
    numTracks = N;
    //InitializeCondition(areRequests);
    areRequests = new pthread_cond_t();
    pthread_cond_init(areRequests, NULL);
    numTurns = 0;
    distance = 0;
}
HDMonitor::~HDMonitor(){
}

/*
```

```
 * Request(int track, int duration)
 * Creates a new request to the hard drive, and is put on the queue for
 * scheduling.
 */
void HDMonitor::Request(int track, int duration, int &numRequests, double &T, int
&turns, int &dist){
    EnterMonitor();
    T = clock()/(double)CLOCKS_PER_SEC;
    int before = jobsList.size();
    int startTurns = numTurns;
    int startDistance = distance;
    condition c;
    InitializeCondition(c);
    request *r = new SSTF(track, time(NULL), duration, this, c);

    RequestWrap wrap;
    wrap.r = r;
    jobsList.push_back(wrap);
    NumAtRequestComplete.insert(pair<request*,int>(r, 0));
    //printf("The size was %d\n", jobsList->size() +1);
    //printf("Just pushed track %d for %d microseconds\n", track, duration);
    if(numWaitingToWork && jobsList.size() >= WAIT_FOR_X_REQUESTS){
    //if(jobsList->size() && !before && numWaitingToWork) {
        signal(areRequests);
    }
    while(find(jobsList.begin(), jobsList.end(), r) !=
        jobsList.end()){
        timedwait(c, WAIT_X_NSECONDS);
    }
    T = ( clock()/(double)CLOCKS_PER_SEC) - T;
    /*
    dist = distance - startDistance;
    turns = numTurns - startTurns;
    */
    dist = distanceMap[r] - startDistance;
    turns = numTurnsMap[r] - startTurns;
    distanceMap.erase(r);
    numTurnsMap.erase(r);

    //printf("The size is %d\n", jobsList->size() +1);
    numRequests = NumAtRequestComplete[r];
    NumAtRequestComplete.erase(r);
    delete r;
    LeaveMonitor();
}
/*
 * DoNextJob().
 * Selects the next job from the list based on the scheduling rule.
 *
 */
void HDMonitor::DoNextJob(){
    EnterMonitor();
    if(!jobsList.size()) { //wait until requests are available
        ++numWaitingToWork;
        //The loop is necessary, or else pthreads will wake up a thread
        //that has been inactive for some time.
        while(!jobsList.size()){
            //printf("going to wait\n");
            //wait(areRequests);
            timedwait(areRequests, WAIT_X_NSECONDS);
        }
        --numWaitingToWork;
    }
    //get next job
    RequestList::iterator nextRequest = min_element(jobsList.begin(), jobsList.end());
    request *r = nextRequest->r;
    //change direction if necessary
    if((direction == -1 && r->track > currentTrack) ||
        (direction == 1  && r->track < currentTrack)) {
          direction *= -1;
          nextRequest = min_element(jobsList.begin(), jobsList.end());
  r = nextRequest->r;
    ++numTurns;
    //printf("Have turned %ld times.\n", numTurns);
    }
    int delta = r->track - currentTrack;
    if(delta > 15) {
        //printf("Found delta with %d. Next track at $d, current track at $d\n", delta,
        //r->track, currentTrack);
    }
    distance += (delta > 0) ? delta : -1*delta;
    currentTrack = r->track;
    //printf("Working on track %d for %d micro seconds\n", r->track, r->duration);
    //printf("Have travelled %ld\n", distance);
    NumAtRequestComplete[r] = jobsList.size();
    int sleepytime = r->duration;
    jobsList.erase(nextRequest);
    distanceMap.insert(pair<request*,int>(r, distance));
    numTurnsMap.insert(pair<request*,int>(r, numTurns));
    signal(r->c);
    //delete r;
    usleep(1);
    //usleep(sleepytime); //Do some "work"
    LeaveMonitor();
}

/*
 * NumberOfRequests()
 * Returns the number of requests pending
 *
 */
 void HDMonitor::NumberOfRequests(int & N){
    EnterMonitor();
    N = jobsList.size();
    LeaveMonitor();
 }
```

HDMonitor.cpp

4.5 Request Class Definition

```
#ifndef REQUEST_H
#define REQUEST_H

#include <ctime>
#include "Monitor.h"

class HDMonitor;

/*
 * request
 * The HDMonitor encapsulates an IO request as an instance of the request class,
 * and maintains a list of these objects.
 *
 */
class request {
    public:
        int track;
        time_t tor; //Time of Request
        int duration;
        condition &c;
        static int dist(int a, int b){ return a < b ? b - a : a - b;}
        static int delta(int a, int b) { return b - a;}
        request(int atrack, time_t ator, int aduration, HDMonitor *aHD, condition &ac):c (ac
            ){
```

```
            track = atrack;
            tor = ator;
            duration = aduration;
            H = aHD; //The comparison operator needs the currentTrack from H
        }
        /*
         * operator<
         * This is required to find the next request.
         */
        virtual bool operator< (const request & r) const = 0;
        bool operator==(const request & r) const{
            return track == r.track && tor == r.tor && duration == r.duration && H
            == r.H && c == r.c;
        }
    protected:
        HDMonitor *H;
};


#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

```
#include "Elevator.h"
#include "HDMonitor.h"

/*
 * Implement the < operator for requests.
 */
bool Elevator::operator< (const request &r) const{
    switch(H->direction){
        case 1:
            return ( request::dist(H->currentTrack, track) < request::dist(H->currentTrack,
                r.track) &&
                    request::delta(H->currentTrack, track) >= 0 ) ||
                  ( track == r.track && request::delta(H->currentTrack, track) >= 0 &&
                    tor < r.tor);
          break;
        case -1:
            return ( request::dist(H->currentTrack, track) < request::dist(H->currentTrack,
                r.track) &&
                    request::delta(H->currentTrack, track) <= 0) ||
                  ( track == r.track && request::delta(H->currentTrack, track) <= 0 &&
                    tor < r.tor);
        break;
        default:
            return false;
            break;
    }
}
```

Elevator.cpp

Shortest Seek-Time First Algorithm

```
#include "SSTF.h"
#include "HDMonitor.h"

/*
 * Implement the < operator for requests.
 */
bool SSTF::operator< (const request &r) const{
  return dist(H->currentTrack, track) < dist(H->currentTrack, r.track);
}
```

SSTF.cpp

4.7 Main Program

```
#include "HDMonitor.h"
#include <cstdio>
#define NUM_TRACKS 15
#include <pthread.h>
#include <cstdlib>
#include <ctime>
#include <unistd.h>
#include <map>
#define WAIT_TIME 1
static int NUM_THREADS = 100;
static int NUM_WORK_THREADS = 1;
using namespace std;

pthread_mutex_t readmutex;

void *Schedule(void* Mon);
void *DoNext(void* Mon);
int main(int argc, char *argv[]){
    if(argc > 2) {
        NUM_THREADS = atoi(argv[1]);
        NUM_WORK_THREADS = atoi(argv[2]);
    }
    HDMonitor M(NUM_TRACKS);
    pthread_t threads[NUM_THREADS + NUM_WORK_THREADS];
    int rc;
    long t;
    void* status;
    pthread_mutex_init(&readmutex, 0);

    for(t=0; t < NUM_WORK_THREADS ; t++){
      rc = pthread_create(&threads[t], NULL, DoNext,  &M);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
    }
    for(t=NUM_WORK_THREADS; t<NUM_THREADS + NUM_WORK_THREADS ; t++){
      rc = pthread_create(&threads[t], NULL, Schedule, & M);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
    }
    //rc = pthread_create(&threads[NUM_THREADS-1], NULL, DoNext, & M);

    if (rc){
     printf("ERROR; return code from pthread_create() is %d\n", rc);
     exit(-1);
    }
    for(t = NUM_WORK_THREADS; t < NUM_THREADS + NUM_WORK_THREADS; ++t){
        rc = pthread_join(threads[t], &status);
    }
    return 0;
}


void* Schedule(void* Mon){
    int track, duration;
    FILE* TimingFP = fopen("TimePerRequest.txt", "w");
    fclose(TimingFP);
    HDMonitor* M = (HDMonitor*) Mon;
    while(true){
  pthread_mutex_lock(&readmutex);
  int ret = scanf("%d %d", &track, &duration);
  pthread_mutex_unlock(&readmutex);
        if( ret == 2){
            clock_t start = clock();
            int N;
            double T;
            int  turns, distance;
            M->Request(track, duration, N, T, turns, distance);
            //M->NumberOfRequests(N);
            TimingFP = fopen("TimePerRequest.txt", "a");
            //fprintf(TimingFP, "%d %16.14f\n", N, (clock()/(double)CLOCKS_PER_SEC) -
            //                              start/(double)CLOCKS_PER_SEC);
            fprintf(TimingFP, "%d %16.14f %d %d\n", N, T, turns, distance);

            fclose(TimingFP);
            usleep(rand() % WAIT_TIME);
        } else if (ret == EOF) {
    break;
  }
    }
}
void* DoNext(void* Mon){
    HDMonitor* M = (HDMonitor*) Mon;
    while(true){
        M->DoNextJob();
    }
}
```

MonitorDriver.cpp

Contents