```
/************************************************
* lab2.cpp
*
* Name:  Mark Randles
* Class: CS408
* Date:  2006-02-28
*
* Description: Real-time simulation of disk
*  access, based on a total service time for
*  1000 disk access requests.
************************************************/

/**********************************************************
* INCLUDES
**********************************************************/
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;

#include <map>
using std::map;

#include <unistd.h>
#include <pthread.h>

#include "ddr.h"

/**********************************************************
* GLOBALS
**********************************************************/
bool                    done = false;                   // set to true if the di
sk drive has completed REQUESTS requests
RequestQueue    *request_queue = NULL;  // the request queue for the disk drive
pthread_mutex_t mutex_request_queue = PTHREAD_MUTEX_INITIALIZER;
bool                    complete[N];                    // set to true when the
disk_drive has completed the last request

/**********************************************************
* PROTOTYPES
**********************************************************/
void *disk_drive();
void *request_thread(void*);

/**********************************************************
* MAIN FUNCTION
**********************************************************/
int main(int argc, char* argv[]) {
        pthread_t threads[N];   // index of threads

        request_queue = NULL;

        cout << argc << endl;

        if(argc > 1 && (strcmp(argv[1],"--debug") == 0 || strcmp(argv[1],"-d") =
= 0)) {
                debug = true;
                DEBUG_PRINT("Debug ON" << endl)
                if(argc >= 3) {
                        if(strcmp(argv[2],"--scan") == 0)
                                request_queue = new RequestQueueSCAN();
                } else
```

```
                        request_queue = new RequestQueueFIFO();
        } else if(argc > 1 && (strcmp(argv[1],"--scan") == 0))
                request_queue = new RequestQueueSCAN();
        else
                request_queue = new RequestQueueFIFO();

        // make sure a request queue exists
        if(request_queue == NULL) {
                cerr << "No request queue created!";
                return(1);
        }

        // seed the random # generator
        srand(0);

        // create the request threads
        for(int i=0; i < N; i++)
                pthread_create(&threads[i], NULL, request_thread, (void *)i

        // run the disk_drive() process, so we've only got N+1 threads
        disk_drive();

        // wait for all the threads to rejoin
        for(int i=0; i < N; i++)
                pthread_join(threads[i],NULL);

        // print some stats for the queue
        request_queue->print_stats();

        return(0);
}

/**********************************************************
* FUNCTIONS
**********************************************************/
void *disk_drive() {
        double total_time = 0.0; // sum of all the times
        int served = 0; // the total number of request processed, should = R
STS
        double seek_time = 0.0; // the seek time for the request
        request *r = NULL;

        // do a service loop until we've processed N requests
        while(served < REQUESTS) {
                // see if there's a waiting request
                while(request_queue->request_count() <= 0) {
                        usleep(1); // sleep the thread for a bit if there i
request
                }

                // get the next request
                pthread_mutex_lock(&mutex_request_queue);
                r = request_queue->next_request();
                DEBUG_PRINT("Service #: " << served << " (" << r->thread <<
<< r->track << "," << r->time_offset << ")" << endl);
//              request_queue->queue_dump();
                pthread_mutex_unlock(&mutex_request_queue);

                // get the seek time
                seek_time = V + (r->track * M);

                // add the current seek time to the total service time
                total_time += seek_time;

                // since we’ve successfully served one request, inc our counter
                served++;

                // sleep for the seek time
                SLEEP(seek_time);

                // set the request completed flag
                complete[r->thread] = true;
        }

        // set the done flag to signal the request processes to terminate
        done = true;

        // unblock all of the request threads that are still blocked
        memset(&complete,true,sizeof(bool) * N);

        // print out the average service time
        cout << "Average Service Time = " << (float)(total_time / served) << end
l;
//      request_queue->queue_dump();

        // exit the thread
        return(0);
//      pthread_exit(NULL);
}

void *request_thread(void *thread_id) {
        double delay = 0.0;
        double total_delay = 0.0;
        int total_requests = 0;
        request *r = NULL;

        while(!done) {
                // create a new request object
                r = NULL;
                r = new request;

                // setup the new request
                delay = (double)(RANDOM() * S);
                r->track = (int)(RANDOM() * T);
                r->time_offset = delay;
                r->thread = (int)thread_id;
                complete[(int)thread_id] = false;

                // add the delay time to the total delay time for this thread
                total_delay += delay;

                // sleep until the request needs to be posted
                SLEEP(delay);

                // insert the new request into the queue
                pthread_mutex_lock(&mutex_request_queue);
                request_queue->new_request(r);
                DEBUG_PRINT("Thread " << (int)thread_id << " created new request
 (" << r->track << "," << delay << ")" << endl);
//              request_queue->queue_dump();
                pthread_mutex_unlock(&mutex_request_queue);

                // incriment the total requests count for this thread
                total_requests++;
```

```

        }

        cout << "Thread id: " << (int)thread_id << "; Average Request Time: 
 (double)(total_delay / (double)total_requests) << "; Total Requests Made: "
total_requests << endl;

        // exit the thread
        pthread_exit(NULL);
}
#ifndef __DDR_H__
#define __DDR_H__

/**********************************************************
* INCLUDES
**********************************************************/
#include <queue>
using std::queue;

#include <vector>
using std::vector;

#include <map>
using std::map;

#include <string>
using std::string;

#include <math.h>
#include <stdlib.h>
#include <unistd.h>

/**********************************************************
* MACROS
**********************************************************/
#define RANDOM() ((double)(rand() / (double)RAND_MAX))
#define DEBUG_PRINT(a) if(debug) { cout << a; }
#define SLEEP(a) usleep((useconds_t)(double)(a * (double)1000))

/**********************************************************
* CONSTANTS
**********************************************************/
const int       N = 8;                          // number of processes
const float     S = 120.0;                      // request time
const int   T = 256;                    // number of tracks
const float     M = 0.05;                       // track seek time
const float V = 3.0;                    // overhead time per request
const int       REQUESTS = 10;  // number of requests to run the sim for

/**********************************************************
* GLOBALS
**********************************************************/
bool                    debug = false;                  // set to true then
g info will print

/**********************************************************
* STRUCTURES
**********************************************************/

// a structure to hold information about a request to the disk drive
typedef struct request {
        int             thread;                 // thread id of the request
```

```
hread
        int             track;                  // track # of the request
        double  time_offset;    // time offset from the last request
};

// abstract base class for request queues
class RequestQueue {
public:
        // base constructor
        RequestQueue() {
                total = 0;
        }

        // base destructor
        virtual ~RequestQueue() {

        };

        // returns the next requests that should be processed
        virtual request* next_request() = 0;

        // returns the current # of requests pending
        virtual int request_count() = 0;

        // returns the total requests processed;
        int request_total() {
                return(total);
        }

        // adds a new request to the queue
        virtual bool new_request(request *r) = 0;

        // performs a dump of the current queue state
        virtual void queue_dump() = 0;

        virtual void print_stats() = 0;

protected:
        int total;
        string name;
};

// a request queue which implements a first-in, first-out priority
class RequestQueueFIFO : public RequestQueue {
public:
        RequestQueueFIFO() {
                name = "FIFO";
        }

        virtual ~RequestQueueFIFO() {
                // clean up any requests left in the queue
                while(!fifo.empty()) {
                        delete fifo.front();
                        fifo.pop();
                }
        }

        request* next_request() {
                request *r;     // temp request pointer

                // get the top request and pop the value off the queue
                r = fifo.front();
```

```
                fifo.pop();

                // return the value of the pointer
                return(r);
        }

        int request_count() {
                // return the count
                return(fifo.size());
        }

        bool new_request(request* r) {
                // make sure we didn’t get a null object
                if(r == NULL)
                        return(false);

                // push the new request into the queue
                fifo.push(r);

                // incriment the total counter
                total++;

                // well everything work so return something that represents
                return(true);
        }

        void queue_dump() {
                queue<request*> t = this->fifo;
                request* r = NULL;
                int i = 0;
                cout << "BEGIN QUEUE DUMP -------------------------" << endl
                while(!t.empty()) {
                        r = t.front();
                        t.pop();

                        cout << "#" << i << ": request = (" << r->thread <<
 << r->track << ", " << r->time_offset << ")" << endl;
                        i++;
                }
                cout << "END QUEUE DUMP ---------------------------" << endl
        }

        void print_stats() {
                cout << "Queue Type: " << name << endl;
                cout << "Total Requests: " << total << endl;
        }

private:
        queue<request*> fifo;   // STL queue class which is a FIFO queue

};

// A request queue which implements the SCAN priority algorithm
class RequestQueueSCAN : public RequestQueue {
public:
        RequestQueueSCAN() {
                head_position = 0;
                direction = 1;
                name = "SCAN";
        }

        virtual ~RequestQueueSCAN() {
                // clean up any requests left in the queue
                for(int i=0; i < T; i++)
                        for(vector<request*>::iterator j=requests[i].begin(); j
!= requests[i].end(); ++j)
                                delete *j;
        }

        request* next_request() {
                request *r;     // temp request pointer

                while(requests[head_position].size() <= 0) {
                        head_position += direction;

                        if(head_position < 0) {
                                head_position = 0;
                                direction *= -1;
                        }

                        if(head_position >= T) {
                                head_position = T - 1;
                                direction *= -1;
                        }
                }

                // get the first element in the
                r = requests[head_position].front();
                requests[head_position].erase(requests[head_position].begin());

                // decriment the counter
                count--;

                // return the value of the pointer
                return(r);
        }

        int request_count() {
                // return the count
                return(count);
        }

        bool new_request(request* r) {
                // make sure we didn't get a null object
                if(r == NULL)
                        return(false);

                // insert the request into the vector for that track
                requests[r->track].push_back(r);

                // incriment the total counter
                total++;
                count++;

                DEBUG_PRINT(requests[r->track].size() << endl);

                // well everything work so return something that represents that
                return(true);
        }

        void queue_dump() {

        }
```

```
        void print_stats() {
                cout << "Queue Type: " << name << endl;
                cout << "Total Requests: " << total << endl;
        }

private:
        map<int, vector<request*> > requests;
        int head_position;
        int direction;
        int count;
};

#endif
FLAGS = -g
LIBS = -lpthread

ddr: ddr.o
        $(CXX) $(CXXFLAGS) $(FLAGS) -o ddr ddr.o $(LIBS)

ddr.o: ddr.cpp ddr.h
        $(CXX) $(CXXFLAGS) $(FLAGS) -c -o ddr.o ddr.cpp

.PHONY: clean
clean:
        @rm -f *.o ddr
1
Average Service Time = 9.915
Thread id: 2; Average Request Time: 51.7728; Total Requests Made: 4
Thread id: 0; Average Request Time: 74.3906; Total Requests Made: 3
Thread id: 6; Average Request Time: 42.8138; Total Requests Made: 5
Thread id: 3; Average Request Time: 34.9343; Total Requests Made: 6
Thread id: 5; Average Request Time: 58.642; Total Requests Made: 4
Thread id: 4; Average Request Time: 45.1473; Total Requests Made: 5
Thread id: 1; Average Request Time: 92.4354; Total Requests Made: 3
Thread id: 7; Average Request Time: 73.3606; Total Requests Made: 4
Queue Type: FIFO
Total Requests: 34
2
Average Service Time = 10.885
Thread id: 2; Average Request Time: 51.7728; Total Requests Made: 4
Thread id: 0; Average Request Time: 74.3906; Total Requests Made: 3
Thread id: 6; Average Request Time: 42.8138; Total Requests Made: 5
Thread id: 3; Average Request Time: 34.9343; Total Requests Made: 6
Thread id: 5; Average Request Time: 58.642; Total Requests Made: 4
Thread id: 4; Average Request Time: 45.1473; Total Requests Made: 5
Thread id: 1; Average Request Time: 92.4354; Total Requests Made: 3
Thread id: 7; Average Request Time: 73.3606; Total Requests Made: 4
Queue Type: SCAN
Total Requests: 34
```