MP 2 – Using Monitors for a Disk Scheduler

CS 423 – Spring 2011
Revision 1.3

Assigned February 9, 2011
Due February 23, 2011, 11:59 PM
Extension 48 hours (penalty 20% of total points possible)

1 Change Log

1.3 Third take on getting a correct sample output. The version 1.2 example was clearly not handling putting requesting threads to sleep correctly. To facilitate your debugging your code, I have allowed you to output more information that just that required, but any additional information must be on a separate line from the required output and must start with two blank spaces. All such lines of output will be ignored.

1.2 Corrected the incorrect sample output and a LaTex error in `thread_yield`.

B.N. **I am not changing the algorithm that I am asking you to implement**, but I should comment that the algorithm described here is NOT the shortest-seek algorithm for finding track on a disk, as it is claimed. (It is, in fact, the algorithm for shortest-seek for sectors in a track, which is no longer a concern for disk scheduling.) For true shortest-seek time, you need to consider the absolute value of distance of the requests from the current position. True shortest-seek time has the problem that requests for tracks far from the middle may be starved. The algorithm you are implementing does not suffer this, but it has a gross inefficiency, in that it only picks up tracks on the way out, and then wastes the return trip. Modifying this so that it picks up tracks in both directions gets you an algorithm called the *elevator algorithm*, which is almost as overall efficient as shortest-seek time and avoids starvation for all requests.

1.1 Corrected LaTex errors in the name `thread_startfunc_t`.

1.0 Initial Release.

2 Acknowledgements

This MP is a piece of an MP used by Sam King, who in turned acquired it from Professor Peter Chen at the University of Michigan.

3 Objectives and Background

The purpose of this MP is to help the students:

- Understand the uses and restrictions of Mesa Monitors with simple multi-threaded programs.

4 Given Library

In the MP, you will be using a library for threads (distinct from pthreads). You will be asked to write a multi-threaded program using this library. This library is provided to you as a tar-gzipped directory `mp2.tgz`. When unpacked to the directory `mp2`, inside are nine files. The first is a header file `thread.h` providing an interface to this library, which we describe below. The second is `thread.o`, supplying a compiled version of the library. In the following MP, you will be asked to implement this library. The third is `libinterrupt.a` giving an interrupt library. The

fourth is a stub of a `Makefile`. Lastly, there are five sample input files. The first three files should be used *as is*, with no renaming or alterations. The file `tread.h` can be included in any file you write needing threads.

The header file provides:

```
typedef void (*thread_startfunc_t) (void *);
```

This is a pointer to a function to be called by a thread upon its creation.

```
int thread_libinit(thread_startfunc_t func, void *arg)
```

The procedure `thread_libinit` is used to initialize the thread library. This should be done exactly once, and before calling any other thread functions. When `thread_libinit` is called it creates the first thread and initializes it to run the function pointed to by `func` on a single argument pointed to by `arg`. The calling process will never execute again after calling `thread_libinit`.

```
int thread_create(thread_startfunc_t func, void *arg)
```

The procedure `thread_create` is used for creating successive new threads. As with `thread_libinit` it will initialize the new thread to run the function pointed to by `func` on a single argument pointed to by `arg`.

```
int thread_yield(void)
```

The procedure `thread_yield` causes the current thread to yield the CPU to the next runnable thread. It has no effect if there are no other runnable threads. The main purpose of `thread_yield` is used to test the thread library, and you should not use it (although, later you will be expected to implement it). A normal concurrent program should not depend on `thread_yield`; nor should a normal concurrent program produce incorrect answers if `thread_yield` calls are inserted arbitrarily.

Synchronization and Mesa Monitors are implemented by the following:

```
int thread_lock(unsigned int lock)
int thread_unlock(unsigned int lock)
int thread_wait(unsigned int lock, unsigned int cond)
int thread_signal(unsigned int lock, unsigned int cond)
int thread_broadcast(unsigned int lock, unsigned int cond)
```

A lock is identified by an unsigned integer (`0 - 0xffffffff`). Each lock has a set of condition variables associated with it (numbered `0 - 0xffffffff`), so a condition variable is identified uniquely by the tuple (lock number, cond number). Programs can use arbitrary numbers for locks and condition variables (i.e., they need not be numbered from 0 - n). The semantics of these function is as was discussed in class. Each of these functions returns -1 on failure. Each of these functions returns 0 on success, except for `thread_libinit`, which does not return at all on success.

There is one last function in `thread.h`:

```
void start_preemptions(bool async, bool sync, int random_seed);
```

The procedure `start_preemptions` is part of the interrupt library we provide (`libinterrupt.a`) and not part of the thread library (`thread.o`), but its declaration is included as part of the interface that application programs include when using the thread library. Application programs can call `start_preemptions()` to configure whether (and how) interrupts are generated during the program. These interrupts can preempt a running thread and start the next ready thread (by calling `thread_yield()`). If you want to test a program in the presence of these preemptions, have the application program call `start_preemptions()` once (and only once) in the beginning of the function started by `thread_libinit()`.

5 Disk Scheduler

The disk scheduler in an operating system gets and schedules disk I/Os for multiple threads. Threads issue disk requests by queueing them at the disk scheduler. The disk scheduler queue can contain at most a fixed number of requests (`max_disk_queue`); threads must wait if the queue is full. In the case of the disk scheduler you will be implementing, the requests will NOT be serviced in a first-come first-served (FIFO) fashion. Instead, they will be serviced in a shortest-seek-time-first (SSTF) fashion.

6 Problem

(35 pts)

1. (15 pts) You are to write a concurrent program to simulate the issuing and servicing of disk requests using the thread library (`thread.o`) we provide, as described above. Your program will be called on the command-line by `disk` with several command-line arguments. The first argument (after the name of the program) specifies the maximum number of requests (`max_disk_queue`) that the disk queue can hold. The rest of the arguments specify a list of input files (one input file per requester). I.e. the input file for requester r is `argv[`$r+2$`]`, where $0 \leq r <$ (number of requesters). The number of threads making disk requests should be deduced from the number of input files specified.

   Your program should start by creating one thread to service disk requests and the number of requester threads specified by the number of input files given on the command line to issue disk requests. Each requester thread should issue a series of requests for disk tracks (specified in its input file). Each request is synchronous; a requester thread must wait (asleep) until the servicing thread finishes handling its last request before issuing its next request. A requester thread finishes after all the requests in its input file have been serviced.

   The input file for each requester contains that requester's series of requests. Each line of the input file specifies the track number of the request (0 to 999). You may assume that input files are formatted correctly. Open each input file read-only (use ifstream rather than fstream). When a requester attempts to put a request to the server, it can only succeed and enter its request into the queue if there is room in the queue for it. If there is no room in the queue, the requester must go to sleep and wait to be woken when there is room. the sleeping requesters should be allowed to enter their requests into the queue in the order in which they went to sleep. When a requester puts successfully puts a request into the queue, it should also call

   ```
   cout << "Requester #" << requester << " wants track " << track << endl;
   ```

   Please be careful to note all the space characters in the string.

   In servicing requests, you need to meet two constraints. First, in order to minimize average seek times, you are require to only process a request when the queue is as full as possible. The queue is as full as possible if either there are `max_disk_queue` requesters in the queue, or there are fewer than `max_disk_queue` requesters still active and they are all in the queue. You will probably find it useful to keep track of the number of threads alive at any given time.

   The other constraint you must satisfy is the shortest-seek-time-first ordering for the requests already in the queue at the time a request is serviced. When the process begins the current position of the disk we are simulating will be fixed at sector 0. The disk will be assumed to have 1000 sectors, numbered 0 – 999. After a request has been serviced, the position of the disk will be the last sector serviced. You will probably want to keep track of the last sector serviced. The seek time between two sectors is calculated as follows: Given sectors a and b, if $a < b$, the seek time from a to b is $b-(a+1)$. If $b \leq a$, then the seek time from a to b is $999+b-a$. Thus the seek time from a given sector is always less to those greater than it than to those less than it (because the disk only spins in one direction), and the maximum seek time is from a sector to itself. When the server services a request, it is always required to service the request in the queue having the shortest seek time. If two different requests have the same seek time, then the one that entered the queue first should be serviced first.

   In this simulation, the server will service a request by telling the requester that its request has been met, and by calling

   ```
   cout << "Serviced requester #" << requester << " with track " << track << endl;
   ```

   A request should be off the request queue when the server prints this line, but no new request should be able to enter the queue until after this line has been completed. There should be no other output generated by your program other than what has been described above.

   Write your disk scheduler in C++ on Linux (such as the EWS machines). Begin by downloading `mp2.tgz` from the course website for MP2, and unpacking it :

```
tar xzf mp2.tgz
```

The code for your disk scheduler must be in a single file called `disk.cc`), and should be placed in the resultant directory. Compile using

```
g++ -g -ldl -m32 disk.cc thread.o libinterrupt.a -o disk
```

In addition to the output described above, you may output additional lines for your own use in tracing/debugging your code provided that they have two blank spaces (at least) at the beginning of the line; all such lines will be treated as comments and ignored by an autograding tool used.

7 Sample Input

Here is an example set of input files (disk.in0 - disk.in4). These sample input files will be in `mp2.tgz`.

disk.in0	disk.in1	disk.in2	disk.in3	disk.in4
53	914	827	302	631
785	350	567	230	11

8 Sample Output

Here is one of several possible correct outputs from running the disk scheduler with the following command:

```
>>disk 3 disk.in0 disk.in1 disk.in2 disk.in3 disk.in4
```

(The final line of the output is produced by the thread library, not the disk scheduler.)

```
Requester #1 wants track 914
Requester #0 wants track 53
Requester #2 wants track 827
Serviced requester #0 with track 53
Requester #3 wants track 302
Serviced requester #3 with track 302
Requester #3 wants track 230
Serviced requester #2 with track 827
Requester #4 wants track 631
Serviced requester #1 with track 914
Requester #0 wants track 785
Serviced requester #3 with track 230
Requester #1 wants track 350
Serviced requester #1 with track 350
Requester #2 wants track 567
Serviced requester #2 with track 567
Serviced requester #4 with track 631
Requester #4 wants track 11
Serviced requester #0 with track 785
Serviced requester #4 with track 11
Thread library exiting.
```

To help see that this example makes sense, here is a second version of it annotated with the current position (track) being read of the disk, the elements (in no particular order) of the requests in the queue, and the state of the requester threads.

```
  Current position: 0
  Current Queue:
  Current pending requests:
    Requester #0(ready): 53 785
    Requester #1(ready): 914 350
    Requester #2(ready): 827 567
    Requester #3(ready): 302 230
    Requester #4(ready): 631 11
Requester #1 wants track 914
  Current position: 0
  Current Queue: (1, 914)
  Current pending requests:
    Requester #0(ready): 53 785
    Requester #1(blocked): 350
    Requester #2(ready): 827 567
    Requester #3(ready): 302 230
    Requester #4(ready): 631 11
Requester #0 wants track 53
  Current position: 0
  Current Queue: (0, 53) (1, 914)
  Current pending requests:
    Requester #0(blocked): 785
    Requester #1(blocked): 350
    Requester #2(ready): 827 567
    Requester #3(ready): 302 230
    Requester #4(ready): 631 11
Requester #2 wants track 827
  Current position: 0
  Current Queue: (2, 827) (0, 53) (1, 914)
  Current pending requests:
    Requester #0(blocked): 785
    Requester #1(blocked): 350
    Requester #2(blocked): 567
    Requester #3(ready): 302 230
    Requester #4(ready): 631 11
Serviced requester #0 with track 53
  Current position: 53
  Current Queue: (2, 827) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(blocked): 567
    Requester #3(ready): 302 230
    Requester #4(ready): 631 11
Requester #3 wants track 302
  Current position: 53
  Current Queue: (3, 302) (2, 827) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(blocked): 567
    Requester #3(blocked): 230
    Requester #4(ready): 631 11
```

```
Serviced requester #3 with track 302
  Current position: 302
  Current Queue: (2, 827) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(blocked): 567
    Requester #3(ready): 230
    Requester #4(ready): 631 11
Requester #3 wants track 230
  Current position: 302
  Current Queue: (3, 230) (2, 827) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(blocked): 567
    Requester #3(blocked):
    Requester #4(ready): 631 11
Serviced requester #2 with track 827
  Current position: 827
  Current Queue: (3, 230) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(ready): 567
    Requester #3(blocked):
    Requester #4(ready): 631 11
Requester #4 wants track 631
  Current position: 827
  Current Queue: (4, 631) (3, 230) (1, 914)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(blocked): 350
    Requester #2(ready): 567
    Requester #3(blocked):
    Requester #4(blocked): 11
Serviced requester #1 with track 914
  Current position: 914
  Current Queue: (4, 631) (3, 230)
  Current pending requests:
    Requester #0(ready): 785
    Requester #1(ready): 350
    Requester #2(ready): 567
    Requester #3(blocked):
    Requester #4(blocked): 11
Requester #0 wants track 785
  Current position: 914
  Current Queue: (0, 785) (4, 631) (3, 230)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready): 350
    Requester #2(ready): 567
    Requester #3(blocked):
```

```
    Requester #4(blocked): 11
Serviced requester #3 with track 230
  Current position: 230
  Current Queue: (0, 785) (4, 631)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready): 350
    Requester #2(ready): 567
    Requester #3(ready):
    Requester #4(blocked): 11
Requester #1 wants track 350
  Current position: 230
  Current Queue: (1, 350) (0, 785) (4, 631)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(blocked):
    Requester #2(ready): 567
    Requester #3(ready):
    Requester #4(blocked): 11
Serviced requester #1 with track 350
  Current position: 350
  Current Queue: (0, 785) (4, 631)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready):
    Requester #2(ready): 567
    Requester #3(ready):
    Requester #4(blocked): 11
Requester #2 wants track 567
  Current position: 350
  Current Queue: (2, 567) (0, 785) (4, 631)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready):
    Requester #2(blocked):
    Requester #3(ready):
    Requester #4(blocked): 11
Serviced requester #2 with track 567
  Current position: 567
  Current Queue: (0, 785) (4, 631)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready):
    Requester #2(ready):
    Requester #3(ready):
    Requester #4(blocked): 11
Serviced requester #4 with track 631
  Current position: 631
  Current Queue: (0, 785)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready):
    Requester #2(ready):
```

```
    Requester #3(ready):
    Requester #4(ready): 11
Requester #4 wants track 11
  Current position: 631
  Current Queue: (4, 11) (0, 785)
  Current pending requests:
    Requester #0(blocked):
    Requester #1(ready):
    Requester #2(ready):
    Requester #3(ready):
    Requester #4(blocked):
Serviced requester #0 with track 785
  Current position: 785
  Current Queue: (4, 11)
  Current pending requests:
    Requester #0(ready):
    Requester #1(ready):
    Requester #2(ready):
    Requester #3(ready):
    Requester #4(blocked):
Serviced requester #4 with track 11
  Current position: 11
  Current Queue:
  Current pending requests:
    Requester #0(ready):
    Requester #1(ready):
    Requester #2(ready):
    Requester #3(ready):
    Requester #4(ready):
Thread library exiting.
```

9 Deliverables

When you are ready to turn in your assignment, you should have created a file named `disk.cc`. When you execute

```
~cs432/bin/handin cs432 -s mp2
```

the file `disk.cc` will be uploaded.