PA5: Concurrent Hard Disk Simulation

CMPSC 473, FALL 2017

Due: December 8, 2017, 11:59:59pm

Ata Fatahi Baarzi, Aman Jain and Bhuvan Urgaonkar

1 Overview

In this programming assignment you will design a multi-threaded device driver for an emulated hard disk drive (HDD). This assignment has two required tasks and two extra credit tasks.

2 Background

A hard disk drive (HDD) is a device that allows storage and retrieval of digital information using one or more rigid rapidly rotating disks (platters) coated with a magnetic material. Each platter is paired with a "disk head," which can read/write to the platter surface. We will study HDDs in some depth in Lecture 24 (Nov. 28).

You will work with a simple emulated HDD (E-HDD) that we have written for you. Our E-HDD has a single platter consisting of 100 Tracks and each of these tracks is in turn composed of 10000 Sectors. A sector is 512 bytes in size and is the granularity of read/write for our disk (meaning both that it is the smallest amount of a request to the disk and that a request to read/write a sector is guaranteed to be atomic by the disk). E-HDD has a fixed length buffer (in terms of number of requests, irrespective of their sizes) for requests coming from the device driver. Requests may be re-ordered once the buffer fills up according to the disk's scheduling algorithm. Requests are issued to the disk head whenever the buffer fills up or a timer expires, whichever occurs earlier. This timer is reset every time requests are issued to the disk head. Clearly, the number of requests coming from the device driver threads can sometimes exceed the buffer size. This requires a certain type of synchronization to be achieved so that none of the requests are lost despite the finite buffer size.

3 Description of Tasks

We are providing you the code for a simple emulated disk (E-HDD) that exposes the following interface to our multi-threaded device driver:
(i) `int read_disk(int sector_number)`, and

(ii) `int write_disk(int data, int sector_number)`
If an access is made to an invalid sector, the disk outputs `BAD SECTOR`, else prints the value read (in case of read) or the value written (in case of write) in the sector. E-HDD is provided as a 1-dimensional array with two components: `Tracks` and `Sectors`, both of them defined as macros.
Since this is an emulation, we have introduced a delay caused by movement of head from one track to another. The delay is calculated as: |source track - destination track|x1 + 2 msec (where 2 msec is the average rotational delay for a HDD with rotational speed of 15,000 RPM) .

Each device driver thread will receive a sequence of two types of requests from (implicit) applications with one device driver thread servicing one application-level thread in our design. These requests will be specified via an input file and will have the following aspects:

- `op_name`: IO call which is either "read" or "write", return values have same interpretation as those for `read_disk` and `write_disk`
- `sector number`: argument to the above call
- `arrival_time`: describes when this request is to be issued by a device driver thread (real time) as an offset from the start time of the program.

We will use `pthreads` for creating our multi-threaded device driver. Each incoming request is launched as a thread which is defined following a structure called `thread_info`. Each of these requests/threads will have: `int tid` which is the thread id, `char[] op_name`, `int sector_number`, `int data`, `int arrival_time` and `struct timespec exit_time`. These threads will call the `_thread_handler()` function which in turn will call either `read_disk()` or `write_disk()` based on `op_name`. After a request has been serviced, it will record the current time as an offset from the start time of the program and store it in the `exit_time` in the thread structure. Difference between arrival time and exit time will show how long it took for the request to get serviced.

We are going to assume that the device driver threads have access to the disk buffer through shared memory. The disk buffer/head will be implemented as a separate thread in the same address space as the device driver threads. In practice, they must make use of privileged IO instructions. A disk operation has the effect of appending a request to the disk buffer if there is room. If there is no room, the device driver thread blocks waiting for there to be room. A new disk request is added as a `buffer_node` which has: `op_name` denoting the operation to be performed by the head, `sector_number` and `data` which is to be written onto the given sector number if its a write operation. The buffer has an upper limit on the number of requests it can hold, given by the variable `limit`. The requests are issued to the disk head whenever the buffer fills up.

3.1 Task 1: Implement Disk Buffer

You are to implement a mechanism to attach disk requests emerging from the device driver to the buffer and issue them to the disk head at appropriate times. Keep in mind

that the incoming requests are being issued by concurrent threads accessing a shared buffer. Therefore, this task requires that you prevent any data races due to this. You must ensure the following holds:

- The number of requests in the buffer should not exceed its limit.
- Requests are only issued to the disk when the buffer is full.
- Requests are be allowed to be added concurrently to the buffer with the disk head servicing requests but issuance should only occur when limit is reached.
- Both the `read_disk()` and the `write_disk()` operations are allowed to add requests to the buffer.

3.2 Task 2: Implement the Elevator Algorithm for Disk Head Scheduling

The `disk_ops(int algo)` function reads the buffer and service all the requests performing them on the disk. The parameter `algo` tell the function whether to use FCFS or Elevator Algorithm to service the requests. Both `read_disk()` and `write_disk()` functions can call the `disk_ops()` function when required.

The function `disk_ops(int algo)`, by default, uses FCFS to execute operations stored in the buffer, i.e., the disk executes the operations in the same order as they were appended to the buffer. This can cause unwanted movement of the head resulting in timely overheads.

Since we need to optimize on the disk performance, your task is to make the buffer follow the "Elevator Algorithm". You should do this by adding the algorithm in the existing `disk_ops(int algo)` function. The parameter `algo` should follow:

- `0`; To use FCFS
- `1`; Io use Elevator

3.3 Task 3: Extra Credit

There are two ways to issue the requests to the disk. Either issue the requests when the buffer fills up or when a fixed amount of time has passed, whichever occurs earlier.
Create a timer which issues the requests to the disk after it expires. This timer is reset every time requests are issued to the disk head.
Make sure that the requests are issued to the disk head if the buffer is full before the timer expires.

3.4 Task 4: Extra Credit

As described in the problem statement, one of the issues faced by the incoming concurrent requests is the concurrency itself, i.e., since there can be multiple reads and writes

issued on the same sector, there should be an order to be followed within those requests.

This will be achieved by using the `ENTER_OPERATION()` and `EXIT_OPERATION()` functions. Whenever a request comes in the `read_disk()` or `write_disk()` operation, before adding itself to the buffer, it should go into the `ENTER_OPERATION(char *op_name, int sector_number)` function. This function decides whether an incoming request follows the Readers/Writers Protocols correctly or not. If it does, the request is allowed to add itself to the buffer, if not, it is simply blocked.

But this is not all, after the request is serviced, it should wake up any request(s) which was/were blocked in the `ENTER_OPERATION()`. This is done by making the request go through the `EXIT_OPERATION(char *op_name, int sector_number)` before leaving the `read_disk()` or `write_disk()` function. The purpose of this function is to signal the blocked requests before finally returning to the `_thread_handler()` function.

Note: This problem is extra credit. You will be awarded points only if this task is completed correctly. No partial credits to be given here.

4 Additional Information

- Information about grading will be released within a few days. Be sure to download an updated version of this document to see information about the grading.
- Before starting, make sure you understand the code base and its components.
- You are given a `Makefile` for the code base which generates an executable named `disk_simulation`. Along with it, your respository will have one sample input file. To run the executable with this input, type:
  `disk_simulation < sample_input.txt`
- You will be provided some test cases with expected outputs for validation. Additionally, you must do your own tests also as your subm will also be evaluated using surprise tests.
- You should create your own test cases to validate your code as your submission will also be evaluated using surprise tests.
- The current limit of the buffer is set as '1'. This will be changed while testing your code. Make sure you test your code against different sizes of buffer.