PA5: Concurrent Hard Disk Emulation

CMPSC 473, FALL 2017

Due: December 10, 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 studied HDDs in some depth in Lecture 25 (Nov. 30).

You will work with a simple emulated HDD (E-HDD) that we have written for you. 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 size 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 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, defined as `int disk[SECTORS*TRACKS]`. Both `TRACKS` and `SECTORS` are defined as macros in `driver.h`. 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 request 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`. Both the `read_disk()` and `write_disk()` are void functions, i.e., they don't return any value but simply print their outputs on the console.
- `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 handled by a newly-created thread whose properties relevant to our emulation are defined within 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. The difference between arrival time and exit time will represent 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. Each buffer node has a request id given by `int req_id` to help the disk thread signal the appropriate device driver thread after its request has been serviced. The buffer has an upper limit on the number of requests it can hold, given by the variable `limit`. The buffer limit is hard coded in the code base, its value can be changed inside the `init()` function. The requests are issued to the disk head whenever the buffer fully 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:

- 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 to the buffer concurrently with the disk head servicing requests.

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

The `disk_ops(int algo)` function reads the buffer and services all the requests issuing them to the disk. The parameter `algo` tells the function whether to use FCFS or Elevator as the disk scheduling algorithm. 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 service requests 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. Your task is to implement the Elevator algorithm. You should do this by adding the algorithm in the existing `disk_ops(int algo)` function. The parameter `algo` should be interpreted as follows:

- `0`: FCFS
- `1`: Elevator

3.3 Task 3: Extra Credit

During relatively idle periods, requests may wait in the buffer for too long. One way to overcome this problem is to start a timer whenever requests are issued to the disk. The expiration of this timer is then used as the next time to issue any requests in the buffer even if the buffer has not fully filled up by then. You are to implement this mechanism. 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

Since there can be multiple concurrent reads and writes issued to the same sector, our device driver threads may face a readers-writers like problem. You are to solve this problem to offer the following behavior. Whenever a request comes in the `read_disk()` or `write_disk()` operation, before adding itself to the buffer, it calls `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 should be allowed to add itself to the buffer, if not, must be 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 call `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 appropriate blocked request(s) before finally returning to the `_thread_handler()` function.

4 Submission and Grading

Same as the previous assignment, PA5 will be done in groups of 2. A group may choose either members repository as their workplace. Do remember to list both members names in all files you submit (including your report). Your project won't be graded if you don't have a name on your report. You are supposed to submit your assignment with your private GitHub repository. Therefore, accept the invitation to access your private repository. If you still need instructions for working with git or adding your teammate to your repository, refer to the included Git manual file. The TAs will grade you by inspecting your code, running some test cases for your code (apart from the ones already provided as sample test cases), and by looking at your report.

4.1 Code and Report

You must push and commit to your private repository all your source files (*.c and *.h) and report file and a Makefile that the TAs will use to compile your code. You should also create a short README with instructions on how to compile and run your code.
The quality and performance of your code along with the clarity of report will contribute towards your grade. Roughly, the TAs will be looking at these aspects when assigning your grade: (i) the quality and clarity of your implementation (complemented by reading your description in your report), (ii) adherence to constraints regarding the buffer and readers/writers protocol (iii) correct synchronization with no deadlock. Your report should have the following two components:

- Pseudo code explaining the synchronization, implementation of readers/writers protocol [if attempted] and buffer timer [if attempted].
- Average service times for the provided inputs for FCFS vs. Elevator. If you are doing the extra credit tasks, you need to report times with the extra functionality.

4.2 Grading Rubric

- **Task 1 : Implementing Buffer - 5 Points**
  Major synchronization bugs : 3 point deduction
  Other minor implementation issues : 2 point deduction

- **Task 2 : Elevator Algorithm - 5 Points**
  Partial grading based on implementation
- **Task 3 (Extra Credit) : Implementing buffer timer - 4 Points**
  No partial grading
- **Task 4 (Extra Credit) : Readers/Writers Protocol - 6 Points**
  No partial grading

5 Additional Information

- 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`. Your repository will have some sample input files. To run the executable with an input, type:
  `disk_simulation < sample_input.txt`
- We have included 4 sample input files along with their expected corresponding output files.

  The buffer limits while testing against the input files were set as follows:

Sample Input File	Buffer Limit
sample_input.txt	1
sample_input_2.txt	2
sample_input_3.txt	3
sample_input_4.txt	5

  These buffer limits are Hard Coded in the Code Base itself, by default having a value of 1. You can change this value within the `init()` function.
  Note that these outputs are expected outputs, the order of thread execution and/or output may change depending on actual implementation. Regardless of the order, the output must follow the buffer constrains [and readers/writers constraints].