4190.308 Computer Architecture, Spring 2016
Disk Lab: Understanding Disks
Assigned: Wed, May 25, Due: Wed, June 8, 23:59

1 Overview

In this lab you will write a disk simulator for rotating harddisks.

2 Downloading the assignment

Download the handout file `disklab.tgz` from eTL (`Computer Architecture` → `Projects` → `Disk Lab`) and save it in a directory of your choice. Next, extract the tarball by issuing the command

```
$ tar xvzf disklab.tgz
```

This will create a directory called `disklab` that contains a number of files. You will be modifying two files: `hdd.cpp` and `hdd.h`. The simulator main file that parses command line arguments and reads the memory trace is provided (and fully functional) in `driver.cpp`. To compile your simulator run

```
$ make clean
$ make
```

3 The Disk Simulator Framework

Note: source code documentation is provided in the `doc/html/` directory. Simply open `doc/html/index.html` in the browser of your choice to open the documentation.

In this lab, you will implement a simulator for rotating hard disks (HDD). The file `disk.h` provides the abstract class definition for any block device such as HDDs, SDDs, USB sticks, etc. For rotating harddisks, a skeleton implementation is provided in `hdd.cpp/h`.

The `Disk` class defined in `disk.h` contains two abstract methods `read/write` that need to be implemented in all concrete subclasses.

The `read` method is called whenever the operating system requests data to be read from a block device. The method takes a timestamp `ts`, the disk address `adr`, and the size (in bytes), `size`, of the access as parameters. The return value is the time when the request finishes (i.e., $ts + latency(read)$).

In a similar way, the `write` method gets invoked by the operating system when data needs to be written to disk. The parameters and return value of the `write` method are identical to that of the `read` method. For some block devices, the `read` and `write` methods are identical.

```
class Disk {
  public:
    Disk(void) {};
    virtual ~Disk(void) {};

    // read 'size' bytes from 'adr'
    // Parameters:
    //   ts time of the event
    //   adr starting address (in bytes) of data to read
    //   size number of bytes to read
    // Return value: time when the read access completes
    virtual double read(double time, uint64 adr, uint64 size) = 0;

    // write 'size' bytes from 'adr'
    // Parameters:
    //   ts time of the event
    //   adr starting address (in bytes) of data to write
    //   size number of bytes to write
    // Return value: time when the write access completes
    virtual double write(double ts, uint64 adr, uint64 size) = 0;
};
```

One instance of a concrete subclass to Disk is provided in `hdd.cpp/h`: the `HDD` class. In this project, you will only be modifying the `hdd.cpp/h` files. The HDD represents rotating harddisks with magnetic platters.

The constructor of the HDD class in `hdd.cpp/h` takes the following parameters:

- `uint32 surfaces` number of surfaces
- `uint32 tracks_per_surface` number of tracks per surface
- `uint32 sectors_innermost_track` number of sectors on the innermost track of the surface
- `uint32 sectors_outermost_track` number of sectors on the outermost track of the surface
- `rpm` rotations per minute
- `sector_size` size of a sector (in bytes)

Your job is to implement the different methods of the `HDD` class based on these parameters.

The input to the disk simulator are disk access traces of a real operating system obtained by tracking accesses of a virtual machine. More about these traces and the trace generation can be found below.

The lab contains a reference implementation in `disklab-ref`. You are encouraged to compare your output with that of the reference implementation. It is possible that the reference implementation still contains bugs. If you suspect that this should be the case, please notify us.

You can use the reference implementation executable as follows:

```
$ disklab-ref < trace/test1.dat
```

3.1 Usage

A provided driver program, `driver.cpp`, is used to run and test your implementation. The driver program first reads the configuration of the HDD from standard input and then instantiates a HDD class. The driver program runs a few basic tests on your implementation such as querying the average rotational latency (`wait_time()`. Then, the driver program reads one trace line after another and generated either a read or write request to your HDD implementation.

We provide a few test traces are provided and can be used with the driver program as follows:

```
$ disklab < trace/test1.dat
```

This command will produce the following output:

```
HDD:
  surfaces:                  8
  tracks/surface:            25000
  sect on innermost track:   4000
  sect on outermost track:   14000
  rpm:                       7200
  sector size:               512
  number of sectors total:   1799900008
  capacity (GB):             921.549

avg. seek time:    0.016500
seek 1 track:      0.004001
avg. rot. latency: 0.004167
read 1 sector:     0.000002
write 1 sector:    0.000002


read(0.000000, 0, 1048576) = 0.008433
read(0.000000, 900000000, 1048576) = 0.012465
read(0.000000, 0, 1048576) = 0.012487
read(0.000000, 900000000, 1048576) = 0.012465
read(0.000000, 0, 1073741824) = 4.894608
read(0.000000, 900000000, 1073741824) = 4.870984
read(0.000000, 0, 1073741824) = 4.894673
read(0.000000, 900000000, 1073741824) = 4.870984
```

3.2 Implementing Your Disk Simulator

Your task is to implement a disk simulator that takes the same command line arguments and produces the identical output as the reference simulator.

You need to implement the following methods:

- constructor and destructor of the class
- the `seek_time`, `wait_time`, `read/write_time` methods
- the `decode` method which takes an address in bytes and translates it into a disk position and
- the `read/write` methods

The disk is implemented in `hdd.cpp/h`. You will find a number of functions with `TODO` markers followed by a short explanation that give you an idea what to do.

Programming Rules

- You can expect valid input parameters only, but should still check for errors. Perform parameter validation and print an error if a parameter is invalid (for example, if the number of outermost tracks is smaller than that of the innermost).
- To receive credit for this lab, you must implement all methods mentioned above. We will use the driver program with our own disk trace files to test your submissions.

3.3 Reference Trace Files

The `traces` subdirectory of this lab contains a collection of *reference trace files* that we will use to evaluate the correctness of your disk simulator. The trace files were generated by tracing disk accesses of a Linux operating system running inside a virtual machine.

The trace files first contain the HDD configuration, followed by a list of read/write accesses.

Use the pipe operator to cat a trace into the simulator. For uncompressed traces, the command is (replace `test1.trace` with any uncompressed trace file of your choice)

```
$ cat traces/test1.trace | ./disklab
```

The compressed traces need to be decompressed before feeding them into the simulator. This can be acheived by running the decompressor `bunzip2` and piping the output directly into the simulator as follows

```
$ bunzip2 -c traces/long.trace.bz2 | ./disklab
```

4 Evaluation

We will evaluate your simulator as follows

- code quality: 10 points
  You get full points if your code compiles without warnings, is indented correctly and reasonably commented.
- basic methods `seek_time, wait_time, and read/write_time`: 5 points each
- position decoding `decode`: 10 points
- per passed trace file: 10 points

We have six tracefiles with which we will evaulate your submission, i.e., the total score is 100 points.

We will give points for (partial) implementations that produce wrong results if you were on the right track, so do not remove code even if it may not work 100% correctly.

5 Handing in Your Work

Once you are ready to submit your work, type the following:

```
$ make handin ID=0000-00000 NAME=FirstnameLastName
```

This will create a file `<studentid>.tgz` which you must send to the TA (`comparch@csap.snu.ac.kr`) by email.