# 4190.308 Computer Architecture, Spring 2017
# Disk Lab: Understanding Disks and Caches
# Due: Sun, June 18, 14:00 (hard deadline)

## 1 Introduction

In this lab, we implement a simulator for rotating harddisks and test it with a number of disk access traces as issued by a real operating system. The lab is written in the C++ language and uses the Standard Template Library (STL). After completing this lab, you will have an in-depth understanding of how disk drives work and some familiarity with C++ and the STL.

The lab comprises two main parts. In Part A, we simulate a simple disk drive. You will need to consider the seek overhead, rotational latency, and the time required to access the data from the rotating surfaces. In Part B, we extend our disk model with an integrated cache and observe the effect of caching on disk accesses.

The simulation framework and skeleton code for the disk drive and the cache are provided in the handout. A reference implementation for the disk drive and a test program for your cache are included. Your submission includes the source code and a report describing your implementation and results. Note that the submission deadline is final, i.e., no grace days can be used for this lab.

## 2 Logistics

You can work alone or in teams of two people on this lab. Clarifications to the assignment will be posted on the course web page.

### 2.1 Downloading and Unpacking the Handout

Download the handout from eTL and save the tarball to a directory in which you plan to do your work. Then unpack the handout with the command `tar xvzf disklab-handout.tgz`. The handout contains the necessary C++ files, a Makefile guiding your build process, several disk configurations and disk access traces, and source code documentation in HTML form.

### 2.2 Handing in Your Work

The tarball for your handin is generated by the `make handin` command. Email the tarball along with your report to the CA TAs.

# 3 Handout and Build Instructions

The handout contains the following files and directories:

| File/directory | Description |
|---|---|
| `Makefile` | Makefile guiding the build and handin process |
| `disklab.pdf` | This document |
| `types.h` | Basic type definitions |
| `disk.h` | Disk interface definition |
| `hdd.h/hdd.cpp` | HDD interface/implementation |
| `cache.h/cache.cpp` | Cache interface/implementation |
| `disk_driver.cpp` | Disk simulation driver |
| `cache_driver.cpp` | Cache test driver |
| `disklab-ref` | Disk reference implementation |
| `cache-ref` | Cache reference implementation |
| `config/` | Disk configuration files |
| `traces/` | Disk access traces |
| `doc/` | Source code documentation |

The `Makefile` controls the build process and is also used to generate the tarball for the handin. Before you begin working on your lab, make sure to set the variables `ID` and `NAME` defined `Makefile` to your student ID and name (use only English characters).

```
$ vi Makefile
#-------------------------------------------------------------------------------
# Student ID / Name
# replace the values for ID/NAME below with your student ID/name
#
ID   = "2016-00000"
NAME = "Invalid"
#-------------------------------------------------------------------------------
```

The `Makefile` implements the following commands:

| Command | Description |
|---|---|
| `disklab` | Build the disklab simulator (`disklab`) (default) |
| `test` | Build the cache test driver (`cache`) |
| `handin` | Create the handin for submission (`<Student-ID>.tgz`) |
| `clean` | Delete object files and generated executables |

Commands are executed using GNU Make

```
$ make <command>
```

If no command is given, the disklab simulator is built.

# 4 Disk Organization and Operation

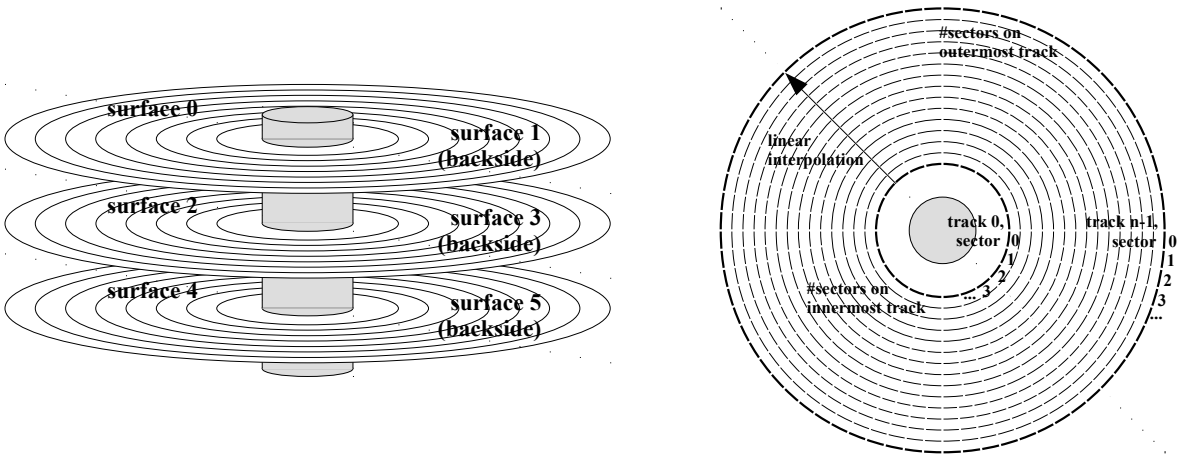
In our framework, a disk is defined by the following nine parameters:

| Parameter | Description |
|---|---|
| *surfaces* | Number of surfaces |
| *tracks_per_surfaces* | Number of tracks per surface |
| *sectors_innermost_track* | Number of sectors on innermost track |
| *sectors_outermost_track* | Number of sectors on outermost track |
| *rpm* | Rotations per minute |
| *sector_size* | Size of one sector, in bytes |
| *seek_overhead* | Base overhead of seek operation, in seconds |
| *seek_per_track* | Linear seek overhead per track, in seconds |
| *cache_blocks* | Number of cache blocks in integrated cache |

Table 1: Configuration parameters of a disk.

## 4.1 Surface, Track, and Sector Layout

The surfaces are organized on a counterclockwise rotating spindle as shown in Figure 1 (a). The tracks and sectors are layout on a surface as displayed in Figure 1 (b): track 0 is the innermost track, sectors are organized in clockwise order. The number of sectors per track is given for the innermost and the outermost track only; for tracks in between, use linear interpolation to compute the number of tracks (round down to the next lower integer). All surfaces use an identical organization and are aligned, i.e., all the sectors $s$ of a certain track $t$ for the different surfaces are located on top of each other when viewed from above.




(a) (b)

Figure 1: Surface, track, and sector layout.

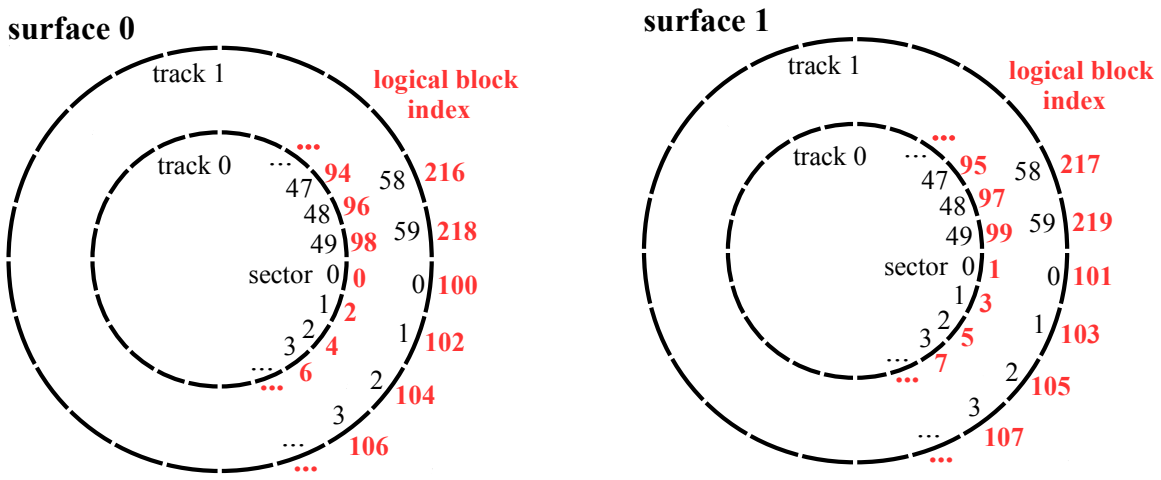



Figure 2: Logical block mapping to surface, track, and sector.

### 4.2 Logical Blocks versus Surface, Track, and Sector Position

Modern disk drives present the data of the disk as a series of logical blocks indexed from 0 to the total number of blocks minus 1. Internally, the disk drive translates this logical block number into a position on the disk. A position on the disk is a triple $(surface, track, sector)$.

As an example, consider a disk with two surfaces and only two tracks. The inner track has 50, the outer 60 sectors. To increase throughput of sequential accesses, logically consecutive blocks are distributed across the surfaces. This is because the read/write head of the disk for all surfaces is always over the same track and sector position. In other words, logical block 0 is mapped to surface 0, track 0, sector 0. Block 1 is located on surface 1, track 0, sector 0. Block 2 on surface 0, track 0, sector 1. Block 3 on surface 1, track 0, sector 1, and so on. This situation is illustrated in Figure 2.

To understand how this increases read/write throughput of consecutive logical blocks, consider a read access to sectors 2-3. With the layout described above, the disk controller has to seek the read/write head to track 0, sector 1, then read sector 1 on surface 0 (block 2) and sector 1 on surface 1 (block 3), i.e., the two sectors can be read *in parallel*. In general, a disk with $n$ surfaces can read up to $n$ blocks in parallel. You will have to consider this when translating logical block indices to surface, track, and sector positions.

### 4.3 Access Latency

The latency of an access comprises three parts: seek overhead, rotational latency, and the data transfer time.

$$t_{access} = t_{seek} + t_{rot.lat} + t_{transfer}$$

**Seeking** is only necessary if the current track position of the disk's read/write head is different from the desired track position. If the access requires a seek, the seek overhead needs to be accounted for. Seeking to a different track involves two kinds of overheads: (1) the *base seek overhead* and (2) the *seek overhead per track*. The former occurs once every time a seek operation is initiated; the second overhead is linear in

the number of tracks crossed. Consider, for example, the situation as illustrated in Figure 3 where we move the head from track 2 to 5. The seek overhead is then computed by

$$t_{seek} = seek\_overhead + (5 - 2) * seek\_per\_track$$

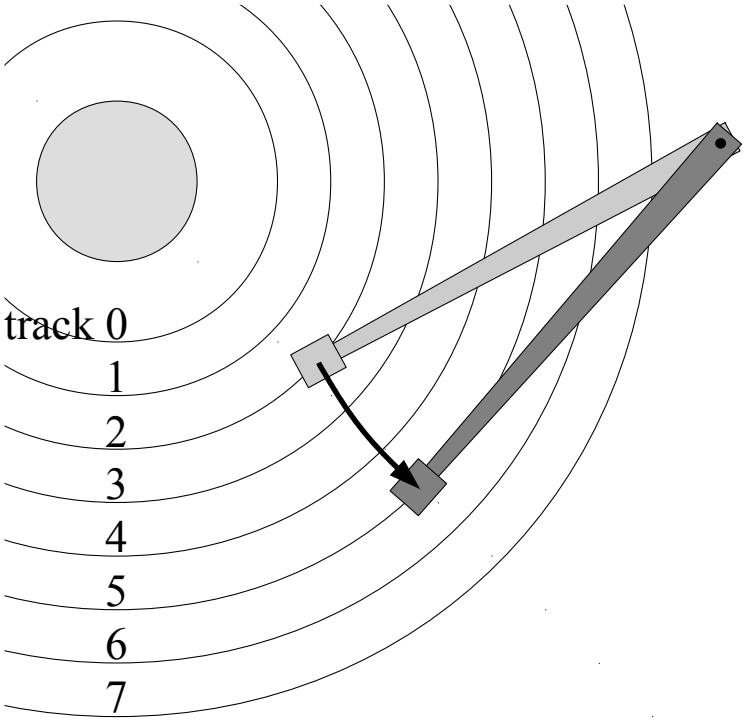



Figure 3: Seek operation from track 2 to track 5.

When the disk starts up, you can assume that the head is positioned above track 0.

The **rotational latency** occurs for every access (even if the head position does not need to be moved) and after every seek operation. In an actual disk, the rotational latency depends on the position of the read/write head and the first sector to read, but for our simulation we assume that the rotational overhead is always half a rotation, that is, the rotational latency is constant for our purposes.

$$t_{rot.lat} = \frac{1}{2} * \frac{1}{rpm} * \frac{60sec}{1min}$$

The **data transfer time**, finally, is given by the time required to read $s$ sectors from a given track $t$. Note that the time to read one sector depends on the track and is not a constant for the entire disk. Assuming that $s$ denotes the number of sectors to read from track $t$ and $t_{sector}(t)$ returns the time to read one sector from track $t$, the data transfer time is given by

$$t_{transfer} = s * t_{sector}(t)$$

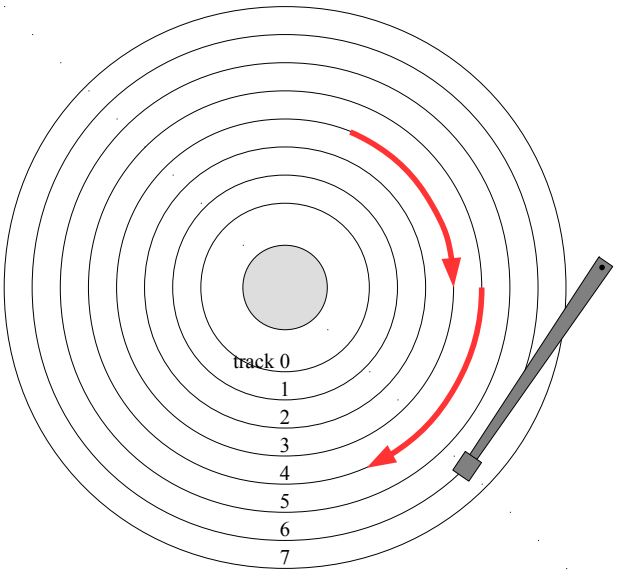



Figure 4: Accesses spanning several tracks.

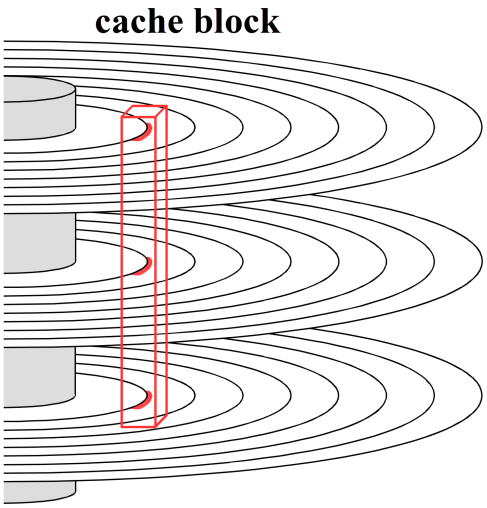



Figure 5: One cache block contains the sectors of all $n$ surfaces for a given track $t$ and sector $s$.

### 4.4 Accessing Consecutive Logical Blocks

A single access to the disk may span several tracks. Consider, for example, a disk with a single surface. The access starts somewhere on track 3 but involves more sectors than are available to the end of the track as shown in Figure 4. In that case, the disk controller reads up to the end of the track, seeks to the following track, and then reads the remaining sectors. Since the disk continues to rotate while the head moves from track 3 to 4, the controller has to wait until sector 0 of track 4 is positioned under the read/write head, i.e., consider the rotational latency.

The access time for the example shown in Figure 4 is the sum of the latencies of the following operations:

1. Seek to track 3
2. Rotational latency
3. Read from start sector to end of track 3
4. Seek to track 4
5. Rotational latency
6. Read from start of track 4 to end sector

### 4.5 Integrated Block Cache

Virtually all modern disks integrate a disk block cache into their drives to speed up the slow accesses to the rotating disks. This block cache typically uses SDRAM to store recently accessed disk blocks and is managed in software by the harddisk controller. Our simulated disk also supports an optional disk block cache. One *cache block* comprises the $n$ 'parallel' sectors of a drive with $n$ surfaces (Figure 5). For the disk shown in Figure 2, for example, one cache block holds two sectors.

# 5 The Disk Simulation Framework

The disk simulator framework consists of a driver program, `disk_driver.cpp` and the implementation of the rotating disks in `hdd.cpp/h`. Types and the abstract `Disk` superclass are defined in `types.h` and `disk.h`, respectively. The block cache is implemented in `cache.cpp/h`.

## 5.1 Disk Simulation Driver

The disk simulation driver program takes as command line arguments the filename of a harddisk configuration and an optional disk access trace file. If no trace file is given, the trace is read from standard input. The command

```
$ ./disklab -c config/hdd1.cfg -t traces/test1.trace
```

for example, initializes a disk as configured in `config/hdd1.cfg` and uses it to simulated the disk access trace given in `traces/test1.trace`.

Disk configuration files contain the nine parameters required to initialize a disk (see Table 1) plus a *verbose* flag. If set to 1, the driver and disk implementation operate in verbose mode and print in more detail what is going on. The contents of the file `config/hdd1.verbose.cfg`, for example, are the following ten parameters:

```
$ cat config/hdd1.verbose.cfg
4 25000 4000 14000 7200 512 0.004 0.01 0 1
```

The parameters are listed in the same order as shown in Table 1, followed by the *verbose* flag.

The disk access traces were obtained by tracing a virtual machine (VM) running Ubuntu Linux. A trace file is a sequence of lines, each line defining a disk access through the four parameters *timestamp*, *access type*, *byte offset*, and *length*. The timestamp gives the time in seconds when the access was issued by the VM. The access type is either `r` (read) or `w` (write). The byte offset holds the byte position on the disk, and the length represents the amount of bytes requested. The following five lines show an excerpt from a trace:

```
1464236826.491567  r  29639296  31744
1464236826.505433  r  29640000  26112
1464236826.510269  r  29640744  12288
1464236826.511261  w  54817623  20992
1464236826.511862  r  29640064  25600
```

Most of the provided access trace files are compressed, in which case you need to decompress the contents first. A simple way is to send the decompressed output to stdout and create a pipe to the disk driver as follows:

```
$ bunzip2 -c traces/vm.sequence1.bz2 | ./disklab -c config/hdd1.cfg
```

The disk simulation driver translates the byte offset into the logical disk block and the access length into the number of blocks requested before forwarding the request to the disk simulator.

The driver program is complete and does not need to be modified. Have a look at its source code if you are curious how it is implemented.

### 5.2 The Disk Interface

The file `disk.h` provides the abstract class definition for a block device such as a rotating harddisk, an SSD, or a USB stick, etc. For rotating harddisks, a skeleton implementation is provided in `hdd.cpp/h`.

The interface of the block device only contains two methods: `read` and `write`. Both methods take a timestamp `ts`, the logical block index `block`, and the number of blocks to read/write `nblocks` as input parameters and return the time when the access completes (i.e., `ts` + *access latency*).

### 5.3 The HDD Class

The HDD class implements the two abstract methods of the `Disk` class plus a few other methods that are needed by the disk simulation driver program. The skeleton code in the handout omits the code of most of the methods; your job is to implement these methods.

## 6 Part A: Simulating a Rotating Disk Drive

In this first part of the lab, your task is to complete the skeleton code of the HDD class (files `hdd.cpp` and `hdd.h`) so that it operates in the same way as discussed in Section 4.

### 6.1 Testing

The provided driver program, `disk_driver.cpp`, is used to run and test your implementation. The simulation driver first initializes an instance of your HDD class and then runs a few basic tests on it such as querying the average rotational latency. The driver then reads the provided input trace and forwards the requests to the `HDD:read()` and `HDD::write()` methods.

You are encouraged to test your code against the provided reference implementation. To understand what is going on, we suggest to use verbose mode. We provide a verbose and non-verbose version of every disk configuration in `config/` so you can easily switch between the two. In addition to real traces obtained from profiling a VM, we also include a test trace to test a few basic operations. This test is located in `traces/test1.trace`. Feel free to copy and extend it to include more special cases.

The following shows an excerpt of running the reference implementation, `disklab-ref`, with the verbose `hdd1` configuration and the `test1` trace:

```
$ cat traces/test1.trace | ./disklab-ref -c config/hdd1.verbose.cfg
HDD:
  surfaces:                  4
  tracks/surface:            25000
  sect on innermost track:   4000
  sect on outermost track:   14000
  rpm:                       7200
  sector size:               512
  cache blocks:              0
  number of sectors total:   899950004
  capacity (GB):             460.774
```

```
avg. seek time:    125.0040000
seek 1 track:      0.0140000
avg. rot. latency: 0.0041667
read 1 sector:     0.0000021
write 1 sector:    0.0000021
(all units in milliseconds)

reading trace from stdin...

// sector 0 on surface 0: read 1 'parallel sector'
read (       0,    2) =
HDD::read(0.0000000, 0, 2)
  head on track: 0
  HDD::decode(0) = surface 0 / track 0 / sector 0 / max.sect 16000
  read              2
  from              0
  to                0
  psec              1
  sectors read:     4
  sectors to read:  0
  block position:   4
  HDD::wait() =  0.0041667
  cumulative time:  0.0041687
0.0041687 ms

...
```

Note that the verbose output of your implementation may be different, and you may also not be able to understand all the output generated by the reference implementation. We will evaluate your implementation in non-verbose mode.

### 6.2 Implementing the HDD Class

Your task is to implement a disk simulator that takes the same command line arguments and produces output identical to the reference simulator. You need to implement the following methods:

- the constructor and destructor of the class
- the `capacity` method that returns the capacity of the disk
- the `seek_time`, `wait_time`, `read/write_time` methods
- the `decode` method which takes a logical block address 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.

# 7 Part B: Simulating a Disk with an Integrated Block Cache

In this second part, your job is to extend your disk to support a block cache.

## 7.1 The Block Cache

A skeleton implementation of a block cache is provided in `cache.cpp/cache.h`. You are to implement a write-through fully-associative cache using the LRU replacement policy. You need to implement the following methods:

- the constructor and destructor of the class
- the `miss_rate` method that returns the miss rate (a float between 0.0 and 1.0) of the cache
- the `has` methods that checks whether a block is currently in the cache
- the `get` and `put` methods that retrieve and store a block in the cache and
- the `dump` method to visualize the contents of the cache

You can test your cache implementation using the provided cache driver (`cache_driver.cpp`). The handout also contains a reference implementation to give you an idea how our cache behaves. To build and run the cache simulator, simply execute

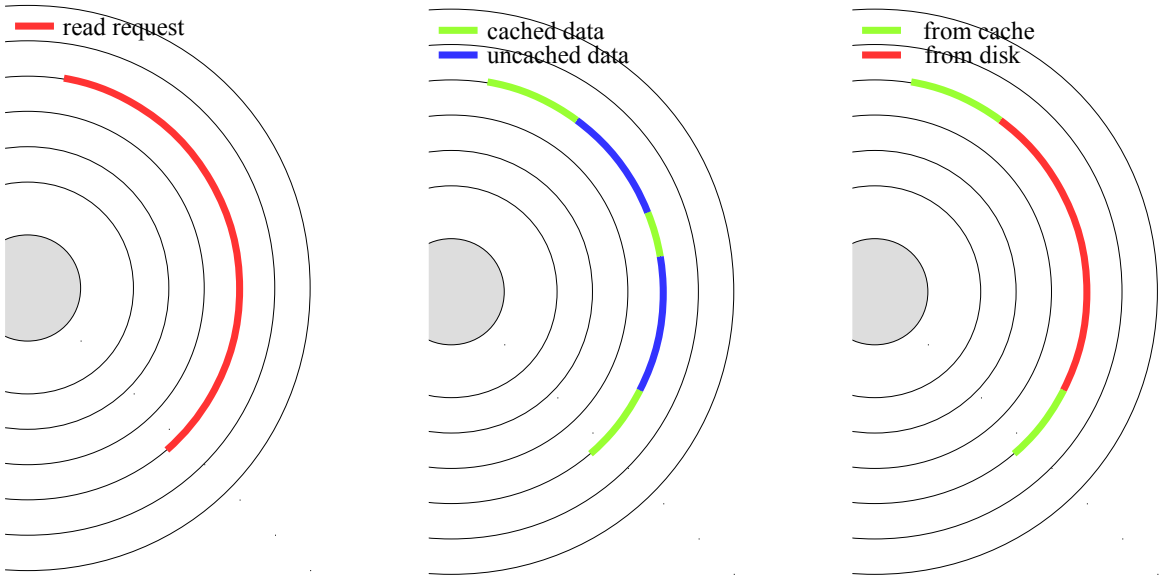
```
$ make test
$ ./cache
```

Compare this to the output of the reference implementation

```
$ ./cache-ref
------------------------------------------------------------
BlockCache:
  cache blocks:                2
...
BlockCache::get(0): miss
BlockCache::dump()
  #hit/miss:  0 / 1
  miss rate:  100%
       lru             block      prev      next  array idx
         0                 0         -         0          1
         1           invalid         1         -          0

BlockCache::get(0): hit
BlockCache::dump()
  #hit/miss:  1 / 1
  miss rate:  50%
       lru             block      prev      next  array idx
         0                 0         -         0          1
         1           invalid         1         -          0
...
```

Again, you may not fully understand the output of the reference implementation, and your implementation does not have to print the same output. However, the miss rates of both caches should be identical.




(a) Read request. (b) Cached/uncached data. (c) Executed disk access.

Figure 6: Interplay of disk and cache accesses.

### 7.2 Integrating the Block Cache with the Disk

To integrate the block cache into your HDD implementation, you need to create an instance of the cache in the constructor if the parameter `cache_blocks` is bigger than zero and access the cache for each `read` and `write` access to the disk.

It is worthwhile thinking a bit about how the cache is used in disk devices. For rotating disks, it may make sense to access the disk even if some data is already in the cache. Consider the situation illustrated in Figure 6. Figure 6 (a) shows the read request as received by the disk. Checking the cache contents reveals that not all data is cached but some sectors at the beginning, in the middle, and at the end of the request are present in the cache (Figure 6 (b)). The disk could read the two uncached parts of the request, but in our implementation that would mean that the second access incurs another rotational latency overhead. In our simplified implementation of what to read from the disk and what to get from the cache, we use the following rules:

- cached areas at the beginning of a request are taken from the cache
- cached areas at the end of a request are taken from the cache
- everything in between, even if partially cached, is read from disk

This scenario is shown in Figure 6 (c).

Also note that the cache is write-through, i.e., data written to the cache is *always* written directly to disk (and also cached in the cache).

# 8 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.
- Part A: 50 points
- Part B: 25 points
- Report: 25 points
  Describe your implementation, results, and experiences/difficulties with this lab.

We will give points for incomplete/wrong implementations if you were on the right track, so do not remove code even if it does not work 100% correctly. Also note that we use different configuration and trace files than the ones provided in the handout for the evaluation.

# 9 Programming Rules and Hints

## 9.1 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).

## 9.2 Hints

The code is fully documented using Doxygen-style annotations. The generated HTML documentation can be found in `doc/html/index.html` under your disklab directory.

**Good luck!**