Advanced Storage Systems 15/18-746:
Project #2: Hybrid SSD/HDD/Cloud Storage System

Due on Monday, April 16, 2012

Overview

In this project, you will build a file system, called CLOUDFS, to integrate into one system with a set of heterogeneous storage devices, particularly solid-state devices (SSDs), hard disk drives (HDDs) and cloud storage systems (Amazon S3, etc.). To ease development, this file system will be built using the "file system in user-space" (FUSE) API. CLOUDFS includes two parts: a core file system that leverages the properties of SSDs and HDDs for data placement, and a second part that takes advantage of cloud storage as an inexpensive way to extend storage capacity.

The deadline to submit the CLOUDFS source and project report is **11.59pm EST on April 16, 2012**. To help you make steady progress, we have two graded intermediate milestones: the first part should be completed by **March 28, 2012** and the second part should be completed by **April 11, 2012**; this will leave you a few days to further enhance your system performance and to work on your performance evaluation report. You must submit your code to Autolab for each milestone. The rest of this document describes the specification of CLOUDFS in detail: Section 1 gives you tips about FUSE and VirtualBox setup, Section 2 and 3 present the specification of CLOUDFS on an SSD/HDD hybrid system and Amazon S3-like cloud service respectively, and Section 4 provides logistics for the code and report for this project.

1 Project Environment and Tools

CLOUDFS will be developed using the file system in user-space (FUSE) toolkit which provides a framework for implementing a file system at user level. FUSE comes with most recent implementations of Linux and is available in other operating systems as well. FUSE has a small kernel module (FUSE in Figure 1) which plugs into the VFS layer in the kernel as a file system and then communicates with a user-level process that does all the work and employs the FUSE library (libfuse in Figure 1) to communicate with the kernel module. IO requests from an application are redirected by the FUSE kernel module to this user level process for execution and the results are returned back to the requesting application.

You will implement CLOUDFS as user-level code that uses libfuse to communicate with test applications (in the project distribution we provide or tests that you write yourself), and uses regular file system calls with different paths (mount points for each) to access the two lower layer file systems, one for one HDD and one for one SSD.

By default, FUSE is multi-threaded, so multiple system calls from user applications can be running at the same time in the user-level process, allowing higher parallelism and probably faster performance. This requires careful synchronization, however, and *it is not required to accomplish this project*. We recommend that you use the FUSE option "-s" which limits the number of threads (concurrent operations from the application through the VFS) to one, to make your debugging simpler.

To enable CLOUDFS development on your own machines (without dedicated SSDs or HDDs), you will use a virtual machine environment. You will use Virtual Box, a free product developed by Sun/Oracle and supported on Windows, Linux and Macintosh machines (project development testing was done with version 4.1.4). Inside a virtual

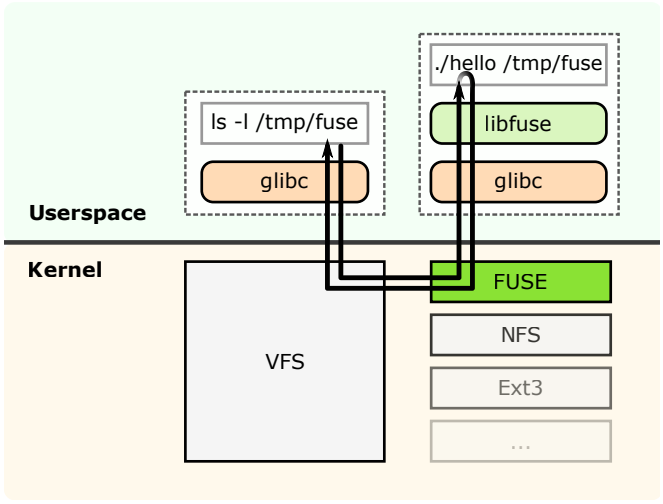

Figure 1: Overview of FUSE Architecture (from `http://en.wikipedia.org/wiki/Filesystem_in_Userspace`)

machine we will give you, you will run a Linux OS with three virtual disks: one for the Linux OS (Ubuntu 10.10), one for your SSD and one for your HDD. It is inside this virtual machine that you will mount FUSE and run your CLOUDFS user-level hybrid file system that will make file system calls on the two virtual disks (not the OS virtual disk).

Note that virtual machine images are 5-10 GB in size, and you will need at least this much free space on the machine you use to run Virtual Box. We have given you three compressed disk images in the project distribution (along with the README with instructions). Please make sure you can run Virtual Box with the root OS partition as soon as possible. If you have problems with this early, we may be able to help. If you have problems with this late, you may be in big trouble.

We will be giving you a few test scripts that you run at user level with a pathname that resolves into the FUSE filesystem. These scripts will typically extract a TAR file containing sample files and directories into CLOUDFS, and then do something with the resulting file system. It will detect the effect of CLOUDFS on each virtual disk using the command vmstat, which reports kernel maintained disk access statistics. These statistics are useful for seeing the effect of splitting the file system between the two devices. We will also give you a script to summarize the vmstat output. This test environment is provided in the project distribution.

We will also provide in the project distribution a web server binary which implements the Amazon S3 interface. You will run this in your virtual machine, so http accesss to this web server uses the loopback network stack to be connected from the same machine. And the disk storage for the cloud will actully be in the Linux OS virutal disk, although it could in principle be in the cloud.

2 Hybrid File System for SSD+HDD (Part 1)

Compared to HDDs, SSDs, particularly NAND flash devices, are (1) much more expensive per byte, (2) very much faster for small random access, even per dollar, (3) comparable for sequential data transfer rates, especially per dollar, and (4) wear out if written too often. Users want the best of both worlds: the price per byte of HDDs, the small random access speed of SSDs and storage that does not wear out. In this project, we will assume that the storage controller (and the flash translation layer) handle wear-leveling that helps improve the life span of SSDs (you don't need to worry about wear out).

The goal of the first part of this project is to build a hybrid file system, called CLOUDFS, that realizes the properties described above for a system that uses both an SSD and an HDD for storage. The basic idea is to put all the small

objects on the SSD and all the big objects on the HDD. We will assume, at least for this project, that big objects are accessed in large sequential blocks, and as a result the performance of these accesses on the HDD will be comparable with accessing the same large sequential block if it were on the SSD. And small objects, such as small file and directories, will be on the SSD where seeks for such objects will be nearly free.

A real implementation strategy would be to modify the Linux Ext (2,3 or 4) file system to be mounted on a device driver that offers *N* HDDs and *M* SSDs. This modified Ext file system would manage lists of free blocks on each, allocating an object on the appropriate device, and have block pointers pointing between the two as appropriate. Since building an in-kernel file system is complex and beyond the scope of a course project, you will use a more layered approach. Each device will have a separate, isolated local file system (e.g. ext2) mounted on it (and you will not modify the code for this per-device local file system). Instead you will write a higher level interposition layer, using the FUSE API, that plugs in as a file system, but rather than use a raw device for its storage, it uses the other two local file systems.

2.1 Design Specifications

Your CLOUDFS that will provide two primary features: size-based placement to leverage high IOPS provided by SSDs and attribute replication to avoid performing small random IO on HDDs.

- **Size-based data placement**

  CLOUDFS data placement - small objects on a SSD and large objects on a HDD - is implemented through redirection when a file is created and is written to. In CLOUDFS, the file system namespace is created on the SSD, small files are written to the SSD and big files (files that grow larger than a threshold) are moved to the HDD. This migration replaces a small file, which was previously stored on the SSD, with a symbolic link containing the actual path on the HDD. When opening such file, CLOUDFS parse the path through the symbolic link file, and opens the actual file from HDD.

  Because CLOUDFS is a FUSE-based file system, most of the code will be implementations of functions called through the VFS interface. Section 1 gave a quick summary of FUSE. In fact, for this project, you don't need to support all the VFS functions; you can build a working prototype using a subset of VFS calls including *getattr, getxattr, setxattr, mkdir, mknod, open, read, write, release, opendir, readdir, init, destroy, access, utimens, chmod, chown, unlink, rmdir*.

  Your implementation of CLOUDFS will have to make various design decisions including deciding whether a file is placed on the SSD or the HDD, detecting when a file gets big, copying it to the HDD and updating the file system namespace on the SSD using a symbolic link. Your CLOUDFS file system should run from command line as follow:

  ```
  ./CloudFS -t MigrateThreshold -s SSDMount -d HDDMount -f FUSEMount
  ```

  where MigrateThreshold is the maximum size of a file stored in SSD (specified in KB, with a default of 64KB), SSDMount and HDDMount are the mount points (in the virtual machine's local file system) for the SSD and HDD respectively, and FUSEMount is the mount point for CLOUDFS.

  The key goal is to migrate a file from the SSD to the HDD based on its size. While it is possible to implement correct behavior in a FUSE file system by opening the path, seeking, performing an access, and closing the file on every read/write call, it is very inefficient. A better way is to open the path on the FUSE `open()` call, save the file descriptor, and re-use that file descriptor on subsequent `read()` or `write()` operations. FUSE provides a mechanism to make this easier: the `open()` call receives a struct `fuse_file_info` pointer as an argument. CLOUDFS may set a value in the `fh` field of struct `fuse_file_info` during open() and that value will be available to all `read()/write()` calls on the open file.

  CLOUDFS faces an additional challenge: after migrating a file from the SSD to the HDD in response to a write, its open file descriptor will change. Unfortunately, you cannot update the `fh` field of struct `fuse_file_info` on a `write()` call with a new file descriptor (it will simply be ignored and the next `write()` call will see the old

value). Your goal is to come up with a cool way to tackle this problem. As in most cases in computer science, a layer of indirection and some additional data-structures may help :-) Odds are quite good that you will still use the `fh` field of struct `fuse_file_info` in your solution, although perhaps not for a file handle. We can imagine solutions that do not use that field at all.

- **Replicating attributes**

  Based on the CLOUDFS description so far, the attributes (such as size, timestamps and permissions) of a big file are stored in the inode for that file on the HDD. In the file system namespace, stored in the SSD, this big file is represented by a symbolic link. Recall that a symbolic link is itself a small file, with its own inode, but the attributes of the symbolic link describe the small file that contains the path to the real big file (stored on the HDD). Reporting the attributes of the symbolic link is not the correct information. So if we use `stat()` for getting the attributes, CLOUDFS will follow the symbolic link and fetch information from the inode on the HDD for a big file. If these small IOs, such as fetching the attributes of a big file, keep going to the HDD then we are not effectively leveraging the high random read performance provided by SSDs.

  CLOUDFS needs to work harder to make these small accesses go to the SSD. In particular, `ls -lR` reports attributes of all files, a small amount of information per file, and it does not read the data of any file, so we would like it to not use the HDD at all. CLOUDFS must replicate attributes of the big files in the SSD, so that attribute scans like `ls -lR` do not need to consult the HDD. There are many ways that you could store the attributes of the big files in the SSD - as a stand-alone database, as a special directory representation with embedded attributes, as extended attributes on the symbolic link, or as an hidden file (called resource forks in OS X) for each big file containing the attributes as data or containing the replicated attributes as extended attributes. Whichever technique is used, it is important to tolerate machine failures and CLOUDFS process crashes, so that the file system is in a consistent state.

  Ideally we would prefer to use extended attributes on the symbolic link, because you will have pulled that inode into memory in order to have seen the symbolic link. In this scheme you would invent user-defined attribute names, beginning with the string "user", one for every attribute of the big file you want to replicate from the disk device into the flash, and then replicate value changes from the disk device to the corresponding extended attribute every time an attribute changes in the HDD. For example, the permissions and mode stored in the `st_mode` attribute of the big file could be replicated in the `user.st_mode` extended attribute of the symbolic link.

  Unfortunately, some local file systems (for example, the one distributed in VirtualBox images for this project) do not support extended attributes on symbolic links. Another alternative is to create a hidden file in the SSD for each big file migrated to HDD and use the extended attributes of this hidden file. Traditionally hidden file are named starting with a ., for example `.xattr_foo` for big file `foo`. For our purposes, file names beginning with `.xattr` can be assumed to be never created by anyone except CLOUDFS. This is not the only way, and is not the most transparent way, but it is good enough for this project.

2.2 Test and Evaluation

To get you started with FUSE, the project distribution includes skeleton code for CLOUDFS. These source files compile without errors and include comments with a few pointers about how to use the FUSE library to build a file system. You are expected to build your code using this skeleton code and include a Makefile to compile it. As described later in Section 4, source code documentation will be an important part of the grading; please write useful and readable documentation (without generating an excess amount of it).

The project distribution also includes scripts to facilitate code testing. Note that these scripts will help you with correctness and performance checks, but you may have to dig deeper to debug (e.g. develop your own tests) performance issues. And your projects will be graded using a different set of scripts and data-sets.

The initial test script for Part 1 of this project is called test_part1.sh. This script performs three operations: extracts a TAR file into the CLOUDFS mount point, reads all the files in the directory (to compute md5sum for each file) and then reads attributes of all objects in the system (using $ls - laR$). We also provide three TAR files of different sizes to help you test your source code. Each of these three operations is wrapped around two measurement related actions.

First, the script will unmount and then re-mount the CLOUDFS file system to eliminate the benefits of caching from the performance numbers. Second, each operation will generate statistics of block IO that happened during each operation. We use *vmstat -d* before and after each of the three operations and then parse the output using a helper program called "stat_summarizer" (which you do not need to change) that summarizes the number of blocks written and read during this operation. Read the man page for *vmstat* to understand its output format. The README file in the *scripts/* directory has details about using this script.

3 Extend Hybrid File System with Cloud Storage (Part 2)

The second part of this project will focus on extending your FUSE filesystem from the first part with cloud storage services such as Amazon S3 and Windows Azure. Cloud storage provides a large storage capacity with pay-as-you-go cost, dynamic scalability and high availability (as long as you have network access). In this project, our goal is to utilize cloud storage to extend your personal machine's storage capacity. After finishing this project, CLOUDFS could be used as a simple version of DropBox.

To understand a cloud storage service, let's take Amazon S3 as an example. Amazon S3 provides a simple web interface that can be used to store and retrieve arbitrary objects (files). Objects are organized into buckets as a flat namespace. Each bucket is essentially a directory that stores a number of objects, which are identified within each bucket by a unique, user-assigned key. Buckets names and keys are chosen so that objects are addressable via an HTTP URL in the form: `http://s3_sever_hostname/bucket/key`. So the namespace is not hierarchical, and only has one level of directory.

Amazon S3 uses simple APIs such as LIST, PUT, GET, DELETE to access objects. The LIST operation retrieves all the bucket names in a S3 server, or the name (key) and attributes of all objects in a bucket. The PUT operation simply writes an object and puts it into the S3 server. The GET operation reads the whole object or a portion of the data. The DELETE operation can delete an object or a bucket.

Cloud storage services have different pricing models. Amazon S3, for example, charges accessing to the quantity of disk space consumed (per month), the amount of data transferred and number of operations performed. (`http://aws.amazon.com/s3/#pricing`). For this project, we choose a similar model, as shown in Table 1, but increase the price since the data-sets used in your tests are comparatively small.

Type	Price
Capacity pricing	$0.125 per MB (Max usage during one test)
Request pricing	$0.01 per request
Data Transfer Pricing	$0.12 per MB (Out from S3 only)

Table 1: Cloud storage pricing model used in class project.

The goal of this project is to extend your code from the first part to support cloud storage, so that users are able to use cloud storage service seamlessly to extend the size of their local HDD storage. Cloud storage acts much like another layer in the storage hierarchy, beyond SSD and HDD in the first part. However, it presents new design considerations that make it distinct from other layers:

- The high latency to the cloud necessitates aggressive caching, otherwise performance of your machine will suffer a lot.
- Cloud storage interfaces often only support writing complete objects in an operation, preventing the efficient update of just a portion of a stored object.
- Cloud storage has elastic capacity and provides operation service times independent of spatial locality, thus greatly easing free space management and data layout. (Since you can't be smart about layout, you do not have to try :)).

- Monetary cost is an important metric for optimization: Although cloud storage capacity might be elastic, it still requires careful management to minimize storage costs over time. Removing redundant data from storage helps reduce storage costs. Providers also charge a small cost for each operation. This motivates you to decide carefully when you are considering moving every small objects instead of a single large one. Total size, toal operations and bytes read from the cloud all cost money.

3.1 Design Specifications

The key idea is to extend CLOUDFS to support Amazon S3-like cloud services as another layer of storage. For the class project, we will limit the complexity and use a web server running locally in the virtual machine to simulate the remote Amazon S3 service. This local web server supports four APIs including LIST, PUT, GET and DELETE. A client library for accessing Amazon S3 servers and its wrapper is also provided, which can communicate with the local web server (as well as real the Amazon S3 service). In order to fulfill the projects requirements, your CLOUDFS will at least support the following features:

- **Objects placement and caching policy**

  Assume that the capacity of the SSD and HDD is limited (their size will be given as input parameters to CLOUDFS), and the capacity of cloud storage is unlimited but its cost is part of your grade. Your implementation of CLOUDFS should consider how to organize file data in different storage layers to minimize response time as well as cloud storage costs.

  Following the design of the first part and making minimal changes, we can still preserve all metadata (directory entries and inode attributes) inside SSD (the SSD will be big enough for this). However, thise meta-data will now need to identify some objects in the cloud (similar to symbolic links to the SSD). You can add more extended attributes if needed, describing each in your documentation.

  A simple policy for what is in the HDD is to store as much as possible locally. Unless local storage will soon run out of space, CLOUDFS does not need to move data from local storage to the cloud. What files should be moved to the cloud when you must? One can imagine that frequently accessed files should reside in local storage, to prevent unneccessary transfers from the cloud, to keep small file access response times as well as data transferring costs. Note that moving many, small, rarely read files to the cloud may cost more than moving a slightly more recently used large file. You are encouraged to design more sophisticated replacement algorithms to reduce response time and cloud storage costs. If you design a new policy, remember to write down the trade-off and evaluations clearly in your final report.

- **Whole-file De-duplication**

  In realistic workloads, file systems tend to have lots of redundant file data. Some study from Microsoft shows that, block-based deduplication of live file systems of 857 desktop computers at Microsoft, can lower storage consumption to as little as 32% of its original requirements. This suggests removing redundant information can greatly reduce the capacity cost in cloud.

  To simplify your task, this project only requires whole-file de-duplication, that is, store only one copy of files having the same data in cloud. As suggested from the same study, whole-file de-duplication achieves about three quarters of the space savings of the most aggressive block-level de-duplication for the same workload. To achieve whole-file de-duplication efficiently, we need methods to detect redudancy quickly. One method is to use "compare-by-hash" to detect duplicated files. For each created or modified file, CLOUDFS computes a cryptographic function (e.g., SHA-1, SHA-256) to calculate a hash value of the file's entire data. If another file with the same hash value already exists, then this new file is assumed to be the same as the other file, and therefore you only need to create a reference link pointing to the other file. Compare-by-hash can declare the two files identical without verifying, since according to the birthday paradox (`http://en.wikipedia.org/wiki/Birthday_problem`), the probability of SHA-1 hash collisions within 5×10^{19} random files is less than 10^{-9}. With this probability, you may accept equivalence and know that some code bugs is for more likely to cause data loss.

In your implementation of CLOUDFS, you should implement a persistent data structure that keeps hash values of all files in cloud at least. Every file object should also maintain a reference count, so that deleting or modifying a file object with multiple hard links can be handled properly.

- **Sharing files in cloud**

  Consider the DropBox example again. It would be nice if CLOUDFS helped you share specific files with your control; that is, specific files (shared documents, music, or pictures) should be placed to the cloud immediately to facilitate sharing. It is acceptable to assume that these files do not need any attributes other than a new specifical name in the cloud, and the new specific name can be part of the URL to the object in the cloud. Based on this observation, your CLOUDFS should implement the following simplified mechanism to share files in cloud:

  When users want to share a file into cloud, they use "setxattr" to specify an extended attribute "user.location" to be "cloud" to notify CLOUDFS. They also specify the category of shared files by using a extended attribute "user.type". Each category has a corresponding bucket (directory) in Cloud, and all files with same category are stored in the same bucket. To preserve the original path name of a shared file it desired, a user can translate an original path name into its key name in the cloud, by replacing the delimit or symbol '/' to '+'. For example, if a user wanting to share the music file "`/home/adele/21.mp3`", they execute a user program `attr` to set the extended attributes:

  ```
  attr -s user.location -V cloud /home/adele/21.mp3
  attr -s user.type -V music /home/adele/21.mp3
  ```

  CLOUDFS should then copy this file object as "`music/home+adele+21.mp3`" into Amazon S3 server, where "music" is a bucket that corresponds the category "music", and "`home+adele+21.mp3`" is the key name translated from the original path name. Notice that shared files do not contain their original metadata in cloud (permissions, time stamps, etc.).

  For simplicity, you only need to synchronize a shared file to the cloud after "`close()`" has been called. During the "`close()`" call, CLOUDFS check the extended attribute "user.location". If its value is "cloud", then CLOUDFS transfer the shared file into the cloud. You only need to use four pre-defined categories for testing: music, movie, photo and document, and it is okay to assume that the symbol '+' does not appear in the original path name.

In part 2, your CLOUDFS file system should take three additional input parameters from the command line:

```
./CloudFS -t MigrateThreshold -s SSDMount -d HDDMount -f FUSEMount -h Hostname -a SSDSize -b HDDSize
```

where Hostname is hostname of the web server that runs the Amazon S3 simulation (probably always set to `localhost:8888`), and SSDSize and HDDSize specify the capacity of the SSD and HDD respectively. The HDD size in particular is very important to decision of which private files are put into the cloud, and tests for grading may change this.

Your implementation should also follow the following instructions:

- DO NOT rely ONLY on in-memory data-structures; this is an unrealistic design for most real-world file systems because you lose memory on crash-n-reboot and because you may not have enough memory to maintain all the file system metadata. Although the test cases for this project are not too big for an in-memory structure of all metadata, we will not accept in-memory only as a correct solution and you will get a poor grade for using such an approach.
- It is appropriate to design out-of-core data structures that are persistent through crash-n-reboot. Your CLOUDFS can use extra space on the SSD to store any state associated with your approach, including special files that are known only to CLOUDFS, but not the data of large files.

3.2 Amazon S3 API Specifications

To simulate the Amazon S3 cloud storage environment, we provide you with a web server running locally in Virtual Box. This web server supports basic Amazon S3 compatible APIS including: LIST, GET, PUT, DELETE on buckets and objects. On the client slide, you will use a open-source S3 client-library called "libs3" in FUSE to allow CLOUDFS to communicate with web server.

The libs3 C library (`http://libs3.ischo.com/index.html`) provides an API for accessing all of S3's functionality, including object accessing, access control and so on. However, in this project, we only need to use a subset of its full functionality. For your convenience, we provide a wrapper of libs3 C libaray in files "`cloudapi.h`" and "`cloudapi.c`", although you are free to use original libs3 C library for better performance. All functions in the wrapper are listed in "`cloudapi.h`". The following example shows how to use these wrapper functions:

```
1  FILE *outfile;
2  int get_buffer(const char *buffer, int bufferLength) {
3     return fwrite(buffer, 1, bufferLength, outfile);
4   }
5
6  void test() {
7    cloud_init("localhost:8888");
8    outfile = fopen("./test", "wb");
9    cloud_get_object("test_bucket", "test", get_buffer);
10   fclose(outfile);
11   cloud_destroy();
12 }
```

To use any wrapper function, you first have to initialize a lib3 connection by calling "`cloud_init(HOSTNAME)`" (shown in line 7), where HOSTNAME specifies the IP address that the S3 web server binds to. Line 8 uses the call "`cloud_get_object`" to download the file "`S3://test_bucket/test`" from cloud to a local file "`./test`". The "`cloud_get_object`" call, takes a bucket name, a file name, and a callback function as input parameters. In the internal implementation of "`cloud_get_object`" call, it retrieves data from S3 server into a buffer, and once the buffer is full or the whole object is downloaded, it will then pass the buffer to the callback function for data processing. Line 2 to 4 shows a callback function that simply writes the received data into the local file system. For more examples of using the wrapper of libs3, look at the sample code "`src/cloudfs/example.c`".

3.3 Evaluation and Testing

Part 2 will be evaluated by two metrics: correctness and performance.

- Correctness: This criteria is used to determine if your CLOUDFS programs returns the correct results for the evaluation workload that we will use for grading. You should implement the three features mentioned in previous section. Also remember to keep key data structures persistent through normal mount/umount.
- Performance: This criteria will determine the efficiency of your programs. Ideally measuring the completion time of different queries would be a good way to evaluate performance, but using VirtualBox (and, in general, a virtual machine) makes timing related measurements unpredictable and unrepeatable. Instead, we will use the disk traffic and cloud costs to measure the efficiency of your implementation, and compare it with a naive approach as the baseline. Both disk traffic and cloud costs should be low, but you must decide and document how you balance between them.

To facilitate testing, we will provide you a test script "`test_part2.sh`" that will emulate the kind of tests that your program is expected to pass. Of course, your homework will be graded on a different set of scripts :-)

Similar to the earlier test script (test_part1.sh), this script runs a set of workloads into your CLOUDFS. The total size of files in the workload will exceed the capacity of the HDD, which forces CLOUDFS to store data into the S3 web server. The workload also contains duplicate files to test your de-duplication functionality. Finally, the script tests the file-sharing functionality by setting pre-defined extended attributes mentioned in previous sections. At the end of the test, the script will retrieve cloud storage cost information from the S3 web server as the evaluation for performance. To test the correctness of your CLOUDFS more completely, you should extend these scripts by designing new test cases. And your own test suites can be also used to illustrate the performance improvement brought by your design in the final report.

4 Project Logistics

4.1 Resources

The following resources are available in the project distribution on the course website.

- **CLOUDFS skeleton code**: The files inside `src/cloudfs/` are the skeleton code that you will modify and extend for your project. "`cloudfs.h`" and "`cloudfs.c`" contain the skeleton code for FUSE file system. "`cloudapi.h`" and "`cloudapi.c`" contain the wrapper functions of libs3 C library. The file `example.c` gives you an example of how to use our wrapper of `libs3` to communicate with the Amazon S3 simulation server. Use the "`make`" command under `src/cloudfs/` to create the binary code of "`cloudfs`" in the directory "`src/build/bin/cloudfs`".
- **Amazon S3 simulation web server**: The file "`src/s3-server/s3server.pyc`" is the compiled python code that simulates Amazon S3 storage service. To run the web server, you can use the command line: "`python s3server.pyc`". The web server depends on the Tornado web server framework (`http://www.tornadoweb.org/`), which has been already installed inside the Virtual Box Image. It stores all the files by default in `/tmp/s3/` (do not change this). To enable logging, you can simply run it with parameter "`--verbose`". More options can be checked out by using "`--help`" option.
- **VirtualBox disk images**: There are three images used by Virtual Box: ubuntu10.10-OS.vdi which is the OS disk image, SSD.vdi which is the SSD disk image and HDD.vdi which is the HDD disk image. Instructions to setup VirtualBox using these .vdi files are included in the README file after you unpack the vbox_images.tar.gz files.
- **VirtualBox setup scripts**: There are three scripts, format_disks.sh, mount_disks.sh and umount_disks.sh, that are required to manage the VirtualBox environment with the SSD and the HDD
- **test_part1.sh**: This script is used to test your solution for both correctness and performance. It allows you to extract three different sizes of TAR files in the CLOUDFS mount point and then perform two kinds of operations (`md5sum` and `ls -alR`) on the file system. This script also generates the relevant blockIO statistics using `vmstat -d` and a helper binary called `stat_summarizer`. You should read about the output format of `vmstat -d` to understand the results.
- **test_part2.sh**: This script is used to test Part 2 of your project and is similar to test_part1.sh except that it runs a different set of workloads to test "cloud" features of CLOUDFS. As described in previoius section, it will test cloud storage, deduplication as well as file sharing. During each test, the script will extract cloud cost infomation from the S3 web server by via URL: `http://localhost:8888/admin/stat` (where `localhost:8888` is IP address that the S3 web server binds to).

All the scripts are placed in the `scripts/` directory and have a README file that describes their usage in details. NOTE that these scripts are provided for your assistance; it is a good idea to write your own scripts to debug and test your source code.

4.2 Deliverables

The homework source code and report is due on **April 16, 2012** and will be graded based on the criteria given below (note: this is the rough criteria and is subject to change).

Fraction	Graded Item
40%	Part 1 (Hybrid FS)
40%	Part 2 (Cloud Storage)
20%	Project report, source code documentation, etc.

What to submit?

For the milestones, you should submit a tar file that contains only a directory "src/" with the source files. You should use the same code structure as given in handout, and make sure that there exists a **Makefile** that can generate the binary code "src/build/bin/cloudfs". We will test this in Autolab, and your code should compile correctly (Test this yourself!).

In your final submission, you should submit a file "AndrewId.tar.gz" which should include at least the following items (and structure):

- **src/ directory** with the source files, and any test suites you used for evaluation in your report. (Please keep the size of test suite files small ($\leq$ 1MB), otherwise omit them.)
- **AndrewId.pdf** containing your 4-page report
- **suggestions.txt** file with suggestions about this project (what you liked and disliked about the project and how we can improve it for the next offerings of this course).

Source code documentation: The src/ directory should contain all your source files. Each source file should be well commented highlighting the key aspects of the function without a very long description. Feel free to look at well-known open-source code to get an idea of how to structure and document your source distribution.

Project report: The report should be a PDF file no longer than 4 pages in a single column, single-spaced 10-point Times Roman font with the name AndrewID.pdf. The report should contain design and evaluation of both parts, i.e. two pages for each part. The design should describe the key data-structures and design decisions that you made; often figures with good descriptions are helpful in describing a system. The evaluation section should describe the results from the test suite provided in the hand-out and your own test suite. Describe your understanding for the disk access counts and cloud storage costs by answering questions such as "why do these counts show that the SSD is being used well?" "why do they show that your caching strategy is better than the simple approach?"

How to submit?

All your submissions will go to Autolab. For the milestones, Autolab's resulsts are used in your grade, but the final grade will do many the Autolab does not run for you, and Autolab only runs compile scripts and basic test scripts we provided in handout. The goal of milestone submission is to ensure that you make steady progress. For the final submission, include both source code and project report as described above. We will run different tests on your code outside Autolab. Please use the same directory structure provided in handout to ease our grading.

4.3 Useful Pointers

- `http://fuse.sourceforge.net/` is the de facto source of information on FUSE. If you download the latest FUSE source code, there are a bunch on the examples included in the source. In addition, the documentation

about FUSE internals is helpful in understanding the behavior of FUSE and its data-structures:
`http://fuse.sourceforge.net/doxygen/`

You can Google for tutorials about FUSE programming. Some useful tutorials can be found at:
`http://www.ibm.com/developerworks/linux/library/l-fuse/`
`http://www.cs.nmsu.edu/pfeiffer/fuse-tutorial/`

- Disk IO stats are measured using vmstat -d. More information on can be found using the man pages. btrace and blktrace are useful tools for tracing block level IO on any device. Read their man pages to learn about using these tools and interpreting their output.

- We have provided instructions to setup VirtualBox in a README file in the `vbox_images.tar.gz` for this project. More information about VirtualBox can be found at the following URLs:
`http://www.virtualbox.org`
`http://www.virtualbox.org/wiki/End-user_documentation`

- We have provide instructions on Amazon S3 API specifications. More information about Amazon S3 API specifications can be found at the following URLs:
`http://libs3.ischo.com.s3.amazonaws.com/index.html`
`http://aws.amazon.com/s3/`