Programming Project 2
Storage Management in a Hybrid SSD/HDD File System *

15/18-746 (Spring 2011), Carnegie Mellon University
Due Date: April 11, 2011

Overview

In this project, you will build a user-level file system, called MELANGEFS, for systems with heterogeneous storage devices, particularly solid-state devices (SSDs) and hard disk drives (HDDs). To ease development, this file system will be built using the file system in user-space (FUSE) API. MELANGEFS includes two parts: a core file system that leverages the properties of SSDs and HDDs for data placement and a metadata-specific optimization for efficient storage management queries such as `aggregator` and `top-K`.

The deadline to submit the MELANGEFS source and project report is **11.59pm EST on April 11, 2011**. To help you make steady progress, we have two intermediate milestones: the first part should be completed by *March 25, 2011* and the second part should be completed by *April 6, 2011*; this will leave you a few days to further enhance your system performance or to work on your performance evaluation report. Note that there are no deliverables due on these dates.

The rest of the document describes the specification of MELANGEFS in details: Section 1 gives you tips about FUSE and VirtualBox setup, Section 2 and 3 present the specification of MELANGEFS on an SSD/HDD hybrid system and the extensions for storage management applications respectively, and Section 4 provides implementation guidelines and hints for this project.

---

1 Project Environment and Tools

MELANGEFS 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 MELANGEFS as user-level code that uses "libfuse" to communicate with test applications (in the project distribution we provide), and uses regular file system calls with different paths (mount points for each) to access the two lower layer file systems, one for the HDD and one for the 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 parallel 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 enable MELANGEFS development on the Andrew clusters or 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. Inside a virtual machine 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

*Acknowledgements - Lin Xiao answered many questions about the first part of this project based on her experiences from Spring 2010.

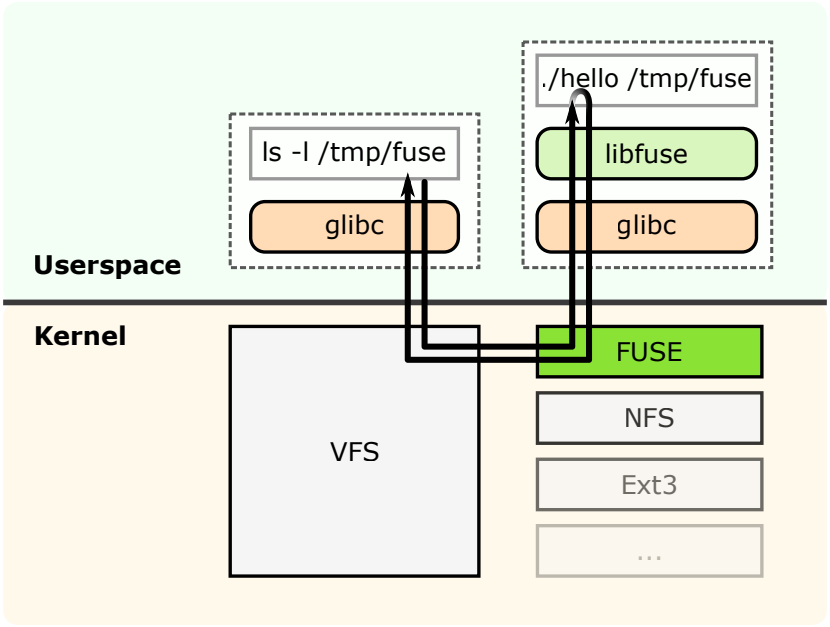

Figure 1 – Overview of FUSE Architecture (from `http://en.wikipedia.org/wiki/Filesystem_in_Userspace`)

virtual machine that you will mount FUSE and run your MELANGEFS 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 then do "something" with the resulting file system. It will detect the effect of MELANGEFS 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.

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: price per byte of HDDs, 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.

The goal of the first part is to build a hybrid file system, called MELANGEFS, 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 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 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 Specification

You will implement MELANGEFS that will provide two 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 –**

  MELANGEFS data placement – small objects on a SSD and large objects on a HDD – is implemented through redirection when file is created and is written to. In MELANGEFS, the file system namespace is created on the SSD, small files are written to the SSD and big 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 to the file that is now stored on the HDD. The VFS layer will interpret the symbolic link by following it (for most common use cases) and opening the "large" file stored on the HDD without having to do any special handling in MELANGEFS implementation.

  Because MELANGEFS 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, mkdir, mknod, open, read, write, release, opendir, readdir, init, access, utimens, unlink, rmdir.

  Your implementation of MELANGEFS 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 MELANGEFS file system should run from command line as follow:

  ./MELANGEFS *MigrateThreshold SSDMount HDDMount FUSEMount*

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

  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. MELANGEFS 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.

  MELANGEFS 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. But we can imagine solutions that do not use that field at all.

- **Replicating attributes –**

  Based on the MELANGEFS 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 it's 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, MELANGEFS will follow the symbolic link and fetch information from the inode on the HDD for a big file. If these small IOs, such a 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.

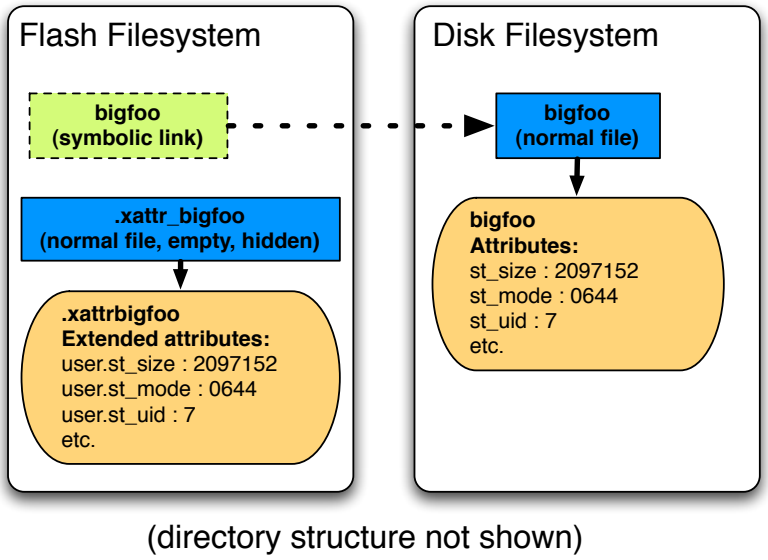

Figure 2 – Attribute replication in MELANGEFS to eliminate small IO accesses going to the HDD.

MELANGEFS 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. MELANGEFS 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. Irrespective of which technique is used, it is important to tolerate machine failures and MELANGEFS 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 values 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 would replicated in the `user.st_mode` extended attribute of the symbolic link. (Note that we will be lax when it comes to create time as it is wrong in the disk device anyway – stretch goal for some of you would be to get the create time right.)

Unfortunately, some local file systems (for example, the one distributed in VirtualBox images for this project) don't support extended attributes on symbolic links. We recommend you 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", as shown in Figure 2 which illustrates a subset of the attributes that must be replicated for `ls -lR` to be run without accessing the disk.

For our purposes, file names beginning with ".xattr" can be assumed to be never created by anyone except MELANGEFS. This is not the only way, and is not the most "transparent" way, but it is good enough for this project; another stretch goal may be to come up with a way to replicate the attributes so that no use of other tools on MELANGEFS shows a difference between the case where all files are on the SSD, versus the hybrid split over SSD and HDD.

2.2 Evaluation and Testing

To get you started with FUSE, the project distribution includes skeleton code for MELANGEFS: two files, `melangefs.c` and `melangefs.h`. These source files compile without errors (because they do anything useful :-)) 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 is an important part of the grading; please write useful and readable documentation (without generate an excess 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 performance issues. And your projects will be graded using different set of scripts and data-sets.

The test script for Part 1 of this project is called "test_part1.sh". This script performs three operations: extracts a TAR file in MELANGEFS 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 MELANGEFS file system to eliminate the benefits of caching from the performance numbers. And 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 don't need to worry about) 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 Faster Storage Management (part 2)

The second part of this project will focus on speeding up storage management applications, such as `du`, `find [pattern]` and *top-k*, for a hybrid file system developed in the first part. Storage management is a class of applications that generate metadata-intensive access patterns, particularly read traffic, that may span the entire file system. Such applications are useful for both system administrators, who need to monitor the system usage, and end users, who need mechanisms to search for desired files. Other storage management applications include context search in desktops, data backups and snapshots.

Let's look at a few illustrative examples of storage management in UNIX. The first example is simple file search: `find / -newer ttt -name '*.c' -print` is used to find (and print) all the `.c` files in the file system that are created after file `ttt` was created. In this particular case, the `find` program uses regular-expression based filtering to return the results. Another example, `du -S /` is used to print a sum of sizes of all the files. The general problem is that these queries may run for a long time. File systems today contain about a million files, and at 5ms per file, storage management queries can easily take well over an hour to run – and future file systems will have billions to trillions of files.

The key requirement of most storage management applications is to scan the (entire) file system looking at attributes of all objects and returning the desired results that match the user-specified filter. This results in a tradeoff between how much work the storage management application needs to do to get the desired result and how much overhead the file system incurs to support storage management. And this tradeoff can be addressed using one of the two extreme approaches.

On one end of the spectrum, the file system provides no special support for efficient storage management queries but the storage management applications end up scanning the entire file system namespace to answer the appropriate queries. In most traditional file systems, this whole file system scan involves a complete traversal of the file system namespace, first (recursively) performing a `readdir()` to read contents of each directory and then doing a `stat()` on each file returned by the directory scan. Clearly file system developers are happy about not adding more (complex) code to facilitate storage management, but storage management applications incur high cost leading to queries that run for hours to days, in terms of larger number of random reads, required for metadata-intensive file system scans. This problem may be exacerbated if storage management queries are done frequently and when the file system is being used by other applications.

The opposite end of the spectrum is to rely on the file system to provide sophisticated interfaces and optimized structures that speed up the storage management applications. For instance, file systems can create and update in-memory "maps" of all attributes of the files in the file system in an on-line manner. While this approach has significant speed-up benefits, it is infeasible to assume that all the file system metadata can fit in-memory for high performance storage management; for instance, a "mid-sized" file system with 100 million files, each with 128 bytes of attributes, will need 12.8GB of memory – even if a system has so much memory, it is not the best use of the resource!

The goal then is to explore designs between these two ends of the spectrum, i.e. to share the work done by the storage management applications and the underlying file system. for a class of storage management queries with minimal whole file system scan through the use of on-disk representations.

3.1 Problem Specification

The key idea is to extend MELANGEFS to provide interfaces and/or storage layouts that facilitate storage management applications. For the class project, we will limit the scope of storage management to two kinds of queries: `top-K` and `aggregation`

- `top-K` queries – These queries return the top "K" files that meet desired criteria. You will write a user-level program called `top-K` with the following syntax:

  `top-K` *Path NumRecords [condition] [filter]*

  where,
  `[condition]` is one or both of *[-s Size | -n FileName]*
  `[filters]` are **optional** arguments including one or both of *[-u UID | -g GID]*

  The program returns, in a descending size order, a maximum of *NumRecords* of files in the sub-tree rooted at the directory *Path* that meet one (or both of the following condition: files that are greater than *Size* bytes and/or files that were created after the file "FileName" was last *accessed*. The results can be filtered using optional arguments including the *UID* and/or the *GID*.

- `aggregation` queries – These queries *aggregate* the desired statistics from the underlying file system. You will write a user-level program called `aggregator` with the following syntax:

  `aggregator` *Path [-c | -s | -o | -y]*

  The program returns a single value corresponding to the condition specified for a given directory *Path*. Your program should support four conditions: `-c` returns the total number of files in *Path*, `-s` returns the size of the largest file *Path*, `-o` returns the timestamp of the oldest file in *Path* and `-y` returns the timestamp of the newest file in *Path*. Report the timestamp in "MONTH/DAY/YEAR HOURS:MINUTES:SECONDS" format.

You are required to write these two user-level storage management programs and extend the MELANGEFS to improve the efficiency of these programs. Both these tasks should stick with the following instructions.

- DO NOT rely ONLY on in-memory data-structures; this is an unrealistic assumption for most real-world file system workloads because you lose them 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, but we will not accept that as a correct solution and you will get a *poor grade* for using such an approach.

- It is okay to design out-of-core data structures that are persistent (and get written to storage) after crash-n-reboot. And your proposed MELANGEFS extensions can use extra space on the SSD to store any state associated with your approach, including special files that are known only to the storage management programs.

- Storage management applications, `top-K` and `aggregator`, are allowed to perform read (or lookup) operations ONLY. Any storage management functions that create, write and mutate must be handled in the FUSE-based MELANGEFS layer – after all a file system should be allowed to perform all types of operations :-)

Overview of Storage Management in Modern File systems

In order to pay a little overhead all the time and reduce the response time of storage management queries a lot, various approaches have been taken in research and products. By far the most common is a periodic (nightly, weekly, etc.) run of a complete "find" script (slow and expensive, but only once) that records all attribute information in a stand-alone file. Then storage management queries are run as a sequential pass over the stored result of the last periodic find. The biggest problem with this approach is the changes that may have happened since the last periodic find was run. For the purposes of this project, historical queries that omit recent changes are considered inappropriate.

Some folks have proposed that a separate database should be maintained, with every change in metadata and attributes synchronously reflected in that database. This enables a rich class of queries on the metadata to be applied – anything that can be expressed in SQL for example. But it means that the database has to keep up with the rate of changes in file system metadata – a feat that few databases can do – and the cost (in disk accesses and latency) of the database modification counts in the total overhead. If you try this, make sure that the database is mounted on the SSD or HDD and that it is writing to the media synchronously. And we expect it to be a rather expensive solution, enabling much richer storage management queries than we need.

Microsoft NTFS has a partial solution. Directories are typically contiguous on disk, but the stat() on the individual files maybe anywhere in the various parts of the inode tables. NTFS replicates the key attributes of each file in the file's directory entry, if there is only one hard link to the file. Then scanning the directories fetches attributes without seeks. This means that the stat() calls hit in the readdir() data in memory.

IBM GPFS has an interesting middle solution. GPFS adds an ioctl() call to "dump to a file" sequential scans of the inode tables and all directories. Then, like the periodic scan, user-level storage management application code runs against the scan output file.

Your solution cannot change the ext2 file systems mounted on the SSD and HDD. You can only manipulate files in these file systems from your FUSE user-level code. Your solution might use parts of the above solutions, if you can rationalize that these techniques improve total throughput, including storage management queries, and provide much better storage management query response time.

3.2 Evaluation and Testing

Both `top-K` and `aggregator` will be evaluated using two metrics: correctness and performance.

- Correctness – This criteria is used to determine if your storage management programs return the correct results for the evaluation workload that we will use for grading.
- 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 number of SSD and HDD IOs before and during storage management query, and compare it with the number of disk IOs required for the naive approach of scanning the entire file system.

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 :-)

Unlike, the earlier test script ("test_part1.sh"), this script only provides simple storage management queries that emulate the "top-k" and "aggregator" programs that you are implementing. These queries will run using the "simple" approach of scanning the (entire) file system sub-tree; the block IO measurements will serve as the baseline performance for storage management.

To test the performance of your implementation of "top-k" and "aggregator", you should extend these scripts to include your programs that perform the same operations as the simple queries, and then compare the results of using your approach with the results from using the "simple" approach.

To test the correctness of you results, you should extend these scripts by inserting a few file system operations that modify the attributes of the file and check that the result of your storage management programs correctly reflects these changes. For example, if `aggregator -s` returns that the largest file is 1345 bytes, and then the application performs

deletes some data from that file and adds 4KB of data to another file, then the next time `aggregator -s` is expected to return the size of "new" largest file.

4 Project Logistics

4.1 Resources

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

- `melangefs skeleton code`: The melangefs.c and melangefs.h are two skeleton code that you will modify and extend for your project.
- *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 configure 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 MELANGEFS mount point and then perform two kinds of operations (`md5sum` and `ls -alR`) on the file system. This script also generates the relevant block IO 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 few storage management queries using the simple approach of scanning the entire file system. This gives you a baseline performance to compare your implementations of `aggregator` and `top-K`.

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 okay 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 11, 2011** and will be graded based on the criteria given below (note: this is the rough criteria and are subject to some changes).

Fraction	Graded Item
35%	Part 1 (hybrid FS)
35%	Part 2 (storage management)
30%	Project report, source code documentation, etc.

What to submit?

You should submit a file `<AndrewId.tar.gz>` which should include at least the following items (and structure):

- `src/` directory with the source files
- `AndrewId.pdf` containing your 4-page report
- `results.txt` file with the output from your tests

- `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 very long description. In addition, include a README file that describes your project and a INSTALL file that has instructions about how to run the program. 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 **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 description are helpful in describing a system. The evaluation section should describe the results from the test suite provided in the hand-out. Describe your understanding for the SSD and HDD access counts by answering questions such as why do these counts show that the SSD is being used well? why do they show that your storage management solution is better than the simple approach? Finally, you may use an additional page to provide feedback on the project and how we can improve it further next year.

How to submit?

The dropbox is at `/afs/ece/usr/ganger/public_html/746.spring11/proj2` handin. We will create a handin directory named your Andrew id and setup the access permissions for you. In order to do this, we need you to run the following commands at `unix.andrew.cmu.edu`:

```
unix49 % aklog ece.cmu.edu
unix49 % pts createuser <username>@andrew.cmu.edu -cell ece.cmu.edu
```

Please do this as soon as possible, otherwise we will be unable to create your directory and grade your project. Copy the tarball to the directory named your andrew id. For example, if your andrew id is "johndoe", you will copy the tarball the directory:

`/afs/ece/usr/ganger/public html/746.spring11/proj2_handin/johndoe`.

In your tarball, please include a file "suggestion.txt" to tell your friendly T.A. what you like and dislike about this laboratory, and whether there is anything you would suggest we change (to make it easier to understand, more challenging, etc.)

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`