CMU 18-746/15-746 **Storage Systems** **Assigned: November 07, 2024**

Project 2: CloudFS – Handout 2 **Due: November 21, 2024**

Contents

Overview

For this project your mission will be to build CloudFS, a cloud-backed local file system. Specifically, a CloudFS instance stores all file metadata and some file data on a local Solid State Disk (SSD), and the rest of the file data using a cloud storage service such as Amazon S3, Microsoft Azure, Dropbox, etc. To make the development process easier, you will be building CloudFS using the FUSE (Filesystem in USErspace) software interface.

CloudFS, and by extension this project, consists of four components: a core file system leveraging the properties of local SSD and cloud storage for making data placement decisions; a component that takes advantage of redundancy in data to reduce storage capacity; a component that adds the ability to create, delete, and restore file system snapshots while minimizing cloud storage costs; and a component that uses local on-device and in-memory caching to improve performance and reduce cloud operation costs. To help ensure you progress steadily through the project goals, we have imposed three **graded** intermediate milestones involving source code submissions:

Milestone	**Description**	**Deadline**
Checkpoint 1	Hybrid FUSE file system	November 07, 2024
Checkpoint 2	Deduplication	November 21, 2024
Checkpoint 3	Snapshots and Caching	December 06, 2024
Report	Final Project Report	December 06, 2024

The last submission you make for Checkpoint 3 will be your final code submission and the report. Note that the final report and Checkpoint 3 will be due on the same date; you should start writing the report early on, because the report is also graded. You should target finishing (most, if not all of) Checkpoint 3 at least two days before the Checkpoint 3 deadline and think of the real deadline as a built-in extension [1]. You must submit your code to Autolab for each milestone; automated testing will be available through Autolab, but we will also review and grade your design for each milestone based on your report.

You are allowed 25 submissions for each of the checkpoints. Any additional submission after the first 25 submissions will incur a penalty of 10% of the checkpoint grade. Autolab shows infinite submissions because we do not impose a hard limit, but rather have a penalty per submission after the first 25 submissions. Furthermore, you have a total of three grace days (i.e., unpenalized late days) for the entire semester. Each checkpoint has a due date and an end date. Grace days can be used to avoid late penalties for submissions past the due date. There will be a late penalty of 10% of the checkpoint grade per day if you run out of grace days. **No submissions are accepted after the end date**.

The rest of this document describes the specification of CloudFS in the context of the first two project checkpoints. In Section 1, we provide a description of the FUSE interface and the AWS instance setup you will be using to simulate your local SSD and the cloud storage service. Section 2 describes CloudFS in detail, including how different file system operations will be carried out in a hybrid SSD/cloud environment. The work items described in this Section make up your submission for Checkpoint 1. Then, Section 3 builds on the presented CloudFS core to achieve a new goal: leverage redundancy in file data to reduce

[1]Finishing checkpoint 3 before exam 2 might also make you better prepared for exam 2 :)

cloud storage cost through fine-grained data deduplication. This work will make up your Checkpoint 2 submission. In Section 4 we provide information on the starter code you received, and Section 5 describes the expected project deliverables. Finally, Appendix A includes detailed information on the Amazon S3-like cloud storage service API, and the Rabin Segmenting API you will be using in the first two checkpoints. A separate handout will be released for Checkpoint 3.

1 Project Environment and Tools

You will be developing CloudFS using the FUSE (File system in USEr space) framework. FUSE provides an interface that can be used to run file system code in user space. To achieve that, FUSE comprises a kernel module and a user-level library. The kernel module plugs into the kernel's VFS layer as a file system, and routes all relevant VFS calls to `libfuse`, the user space library. `libfuse` then proceeds to invoke the appropriate call for the user space file system. As shown in Figure 1, file system calls from applications are routed through the kernel back to user space for execution. The reverse path is then followed to return any output to the application. You will implement CloudFS as user-level code that uses `libfuse` to service file system calls of test applications attempting to access the SSD. These applications can either be those found in the project distribution we provide, or new test applications that you write yourself.

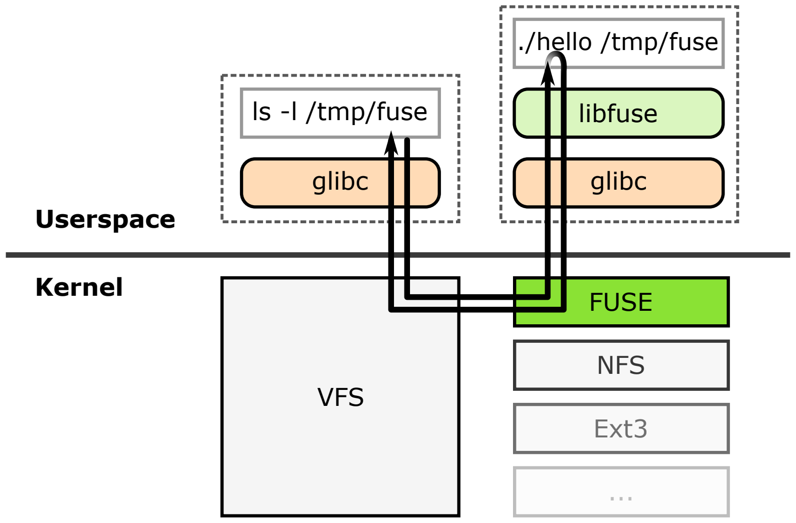

Figure 1: A flow-chart diagram showing how FUSE works (Source: Wikipedia).

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 (potentially) faster performance. This requires careful synchronization, and it is **not** required to accomplish this project. We recommend that you use the FUSE option `-s` to limit the number of threads (concurrent operations from the application through the VFS) to one. This should make debugging your code considerably easier.

To enable CloudFS development, you will continue using the VM instance you have been using so far in the course. The image contains support for simulations of a dedicated SSD and external cloud service (for more information see Appendix A).

Inside the AWS instance, you will run a Linux OS with a virtual disk that contains two partitions: one for the Linux OS (Ubuntu 14.04) and one that will simulate the CloudFS SSD. It is inside this instance that you will run your CloudFS user-level hybrid file system, and it will make file system calls on the SSD partition (not the OS partition) and a *local* cloud storage service.

We will be providing you a few test scripts that you can run in the AWS instance with a path argument that resolves to a directory within the FUSE file system. These scripts will typically extract a TAR file containing sample files and directories into CloudFS, and then perform operations on the resulting file system. The scripts are written to detect the effect of CloudFS on storage components using the command `vmstat` and a statistics report from the cloud storage service. These measurements are useful for quantifying the effectiveness of your CloudFS implementation in distributing data between the local SSD and the cloud service (Checkpoint 1), deduplicating redundant data segments (Checkpoint 2), taking file system snapshots, and caching file data (Checkpoint 3, described in a separate handout). We will also be providing an object storage server binary which implements an interface similar to that of Amazon S3. You will run this as a web server in your instance, so HTTP accesses to the cloud server will use the loopback network stack and be redirected back into the same machine. The disk storage for the cloud will actually reside in the Linux OS virtual disk, although locating it in the cloud should (in principle) not require any changes to your code.

Finally, here are a few suggestions for the design of your project:

- Try to keep the CloudFS design modular, to make your life easier across checkpoints. Think of the various parts (SSD/cloud data partitioning, deduplication, snapshots, caching) as layers of function stacked on top of the cloud storage. Designing clear and pre-documented interfaces between layers should simplify the design and save you lots of wasted development effort.
- **Do not** rely **only** on in-memory data-structures; this would be an unrealistic design for most real-world file systems because you will lose memory contents on a system crash, or in the event that you do not have enough memory to hold all of the file system metadata. Although the test cases for this project are not big enough to cause problems with solely in-memory metadata structures, we will not accept in-memory only as a correct solution and you will get a poor grade for using such an approach.
- You should design out-of-core data structures that are persistent across mounts. Your CloudFS can use storage space on the SSD to store any state associated with your approach (including special files that are known only to CloudFS), but certainly not the data of all large files.

2 Checkpoint 1: Hybrid file system spanning SSD and cloud storage

For the first checkpoint of the project you will have to implement a hybrid file system with two different storage components: a local SSD and a cloud storage service similar to Amazon S3. Cloud storage services provide storage capacity using a "pay as you go" cost model, dynamic scalability, and high availability, as long as you are connected to the network.

In this part of the project, you will get to extend your limited personal storage (the local SSD) using cloud storage. Compared to cloud storage, SSDs and particularly NAND flash devices are: (1) significantly faster, (2) subject to wear out after a number of writes, (3) subject to limited capacity per dollar, and (4) free to use as they induce no charge (billing) per operation or for data transfer. Users want the best of both worlds: the extensibility of cloud storage, the small random access speed of SSDs, the low operational cost of local SSDs, and the freedom of not having to worry about device endurance or reliability, all at a minimal cost. In this project, we assume that the SSD storage controller and its Flash Translation Layer completely handle wear-leveling to improve the lifespan of the device, so you need not worry about wear out.

The goal of the first checkpoint of this project is to build a basic hybrid file system, called CloudFS, that realizes the properties described above for a system that uses both a local SSD and the cloud for storage. The basic idea is to put all the small objects on the SSD and all the big data objects on the cloud storage. An ideal implementation strategy would be to modify a local Linux file system mounted on an SSD. This modified file system would manage lists of free blocks on SSD, allocating an object on the appropriate storage component, and maintaining pointers between the two as appropriate. Since building an in-kernel file system is complex and beyond the scope of a course project, you will be taking a user-level approach through FUSE.

In our project, the SSD device will have a dedicated local file system mounted on it (specifically: the ext4 file system), and you will not modify the code for this local file system. Cloud storage can be accessed via the Amazon S3 object store interface. You will use the FUSE API to write the high-level interposition layer that serves file system calls in user space. However, rather than using a raw device for storage, you will use the local file system interface to the SSD, and the Amazon S3 interface to the cloud storage.

The cloud storage has a different interface from the traditional file system API. 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 to be chosen so that objects are addressable via an HTTP URL in the form: `http://s3_sever_hostname/bucket/key`. The namespace is not hierarchical, and only has a depth of one directory level. Amazon S3 uses a simple set of API calls, such as LIST, PUT, GET, and DELETE to access objects. The LIST operation retrieves all the bucket names in a S3 server, or the names (keys) and attributes of all objects in a given bucket. The PUT operation puts an entire object into the S3 server. The GET operation reads a whole object. The DELETE operation can delete an object or a bucket.

Unlike with local SSDs, users of cloud storage are billed on a monthly basis. Cloud storage services have different pricing models. For instance, Amazon S3 charges for the capacity of disk space you consume, the amount of data transferred, and each command you submit, e.g. object retrieval operations. Dropbox follows a different cost model, charging only for the number of bytes used, i.e., storage capacity. For this project, we choose an Amazon-style cost model, which is shown in Table 1. One of your design goals is to minimize the cost incurred for as wide a range of workloads as possible, especially the ones in our tests.

Type	**Price used in our tests**	**Real price of Amazon S3**
Capacity	$ 0.03 per MB (our tests bill you for your max usage during each test)	$ 0.023 per GB per month for the first 50 TBs
Operation pricing	$ 0.01 per request	PUT, COPY, POST, LIST: $ 0.005 per 1,000 requests. GET: $ 0.0004 per 10,000 requests.
Data transfer pricing	$ 0.09 per MB (outgoing from S3 only)	$ 0.09 per GB for up to 10 TB (outgoing from S3 only)

Table 1: Cost model for Cloud Storage API (Amazon pricing source: `http://aws.amazon.com/s3/pricing`)

Your CloudFS will provide two primary features, both of which are described in more detail in the following subsections:

- Size-based placement to leverage the high number of IO operations per second (IOPS) provided by SSDs and
- Attribute replication to avoid performing small IOs on Cloud storage.

2.1 Size-based data placement

The CloudFS data placement policy, i.e., placing small objects on a local SSD and large objects in cloud storage, is implemented through redirection when a file is created and written to. In CloudFS, the file system namespace is created on the SSD, small files are written to the SSD, and big files (or files that grow larger than a predetermined threshold) are moved to the cloud storage. This migration replaces a small file, which was previously stored on the SSD, with a user-defined link pointing to the location of the big file in the cloud storage. When opening such a file, CloudFS parses the path through the user-defined link, creates a temporary file on the SSD, and copies an entire cloud object into the temporary file on the SSD. Finally, CloudFS returns the file descriptor of this temporary file as the return value of the `open` system call. When a big file is closed, you shall flush the temporary file into cloud storage if the file is dirty (modified), or delete it if the big file is clean (unmodified). For design simplicity, you can assume that the SSD always has enough space for keeping copies of all currently open files. For Checkpoint 2 you may need to re-design the handling of open files, because we will be dropping the assumption that all open files will fit in the SSD.

Because CloudFS is a FUSE-based file system, most of the code will be written to implement functions called through the VFS interface. Note that 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`, `link`, `symlink`, `readlink`, `unlink`, `rmdir`, and `truncate`. Your implementation of CloudFS will have to make various design decisions including deciding whether, and when a file is placed on the SSD or the cloud storage, detecting when a file gets big, copying it to the cloud storage and updating the file system namespace on the SSD using a user-defined link. Your CloudFS file system should run from the command line with the following syntax:

```
./CloudFS --hostname Hostname --threshold MigrateThreshold
          --ssd-path SSDMount --fuse-path FUSEMount
          --ssd-size SSD_size [--more_args]
```

where `MigrateThreshold` is the maximum size of a file that should be stored (permanently) in the SSD (specified in KB, and defaulting to 64 KB), `SSDMount` is the mount point for the SSD device in the AWS instance (defaulting to `/mnt/ssd`) and `FUSEMount` is the mount point for CloudFS (defaulting to `/mnt/fuse`). Hostname is hostname of the web server that runs the Amazon S3 simulation (it should probably always be set to `localhost:8888`). `SSD_size` specifies the capacity of SSD (specified in KB, not counting open temporary files). You can assume that there is no capacity limit on the cloud storage.

A key goal is to migrate a file from the SSD to the cloud storage based on its size. While it is possible to implement the correct behavior of a single operation in a FUSE file system by opening the path, seeking, performing the operation, and closing the file on every operation, 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 from all future `read`/`write` calls on the open file. On `close`, you make the decision to migrate the file to cloud depending on the file size. If the file does need to be stored in the Cloud, this will have to be done in a manner transparent to users of the filesystem. This means you have to store all the attributes of the file as well as the mapping to an object name in the cloud somewhere on the SSD.

2.2 Saving the attributes of migrated files

Based on the current description of CloudFS, attributes such as size, timestamps, and permissions of a big file need to be stored with a stub or proxy file on the SSD. You don't need to store attributes of big files in the cloud as well. CloudFS needs to work hard to make sure these small accesses go to the SSD, not to the cloud. 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 to incur the cost and latency of cloud storage access. Note that you should maintain all metadata fields generally made available via `stat`. Check `struct stat` in the man page for `stat` for a detailed breakdown.

The metadata about files that have been migrated to the cloud can be managed on SSD in a couple of ways. One of them is to have a proxy file on the SSD to represent the file in cloud. Reporting the attributes of this proxy file is **not** the correct information. So if a user makes a `stat` call to get the attributes of a file in the cloud, CloudFS should look into the proxy file, and fetch the cloud file's information from the proxy file's user defined metadata structure. Note that there will be no test cases that generate files or directories that have names starting with period (.), so you can use the proxy file names accordingly, if you want. There are many other ways that you could store the attributes of big files in the SSD: in a stand-alone database, as a special directory representation with extended attributes, or as data in an hidden file (such as resource forks in OS X). Whichever technique you use, it is important to tolerate machine failures and CloudFS process crashes, so that the file system is in a consistent state after rebooting and restarting. If you decide to keep metadata information in extended attributes of a proxy file, you could invent user-defined attribute names, beginning with the string "user", for example, one for every attribute of the migrated file. Then, you will have to ensure that they always also stay updated on non-volatile storage[2]. Also, the total bytes that can be used for names and values of the extended attributes of a file cannot be greater than 4096 bytes.

2.3 Mapping files to objects in the cloud

The cloud storage can be viewed as a key-value store. Therefore, in order to put a file in the cloud, you have to first come up with a key for the file you want to upload, i.e., a name for the object in the cloud. You also need to be able to regenerate or remember the key in order to read the file back from the cloud into the SSD. One way to do this is to use the original path name of the file as a name in the cloud storage so that you can correctly infer the file's location in the cloud storage from the original path name. This might be desirable because it does not incur extra storage space for a mapping table and you need not take special care of consistency. You might need to replace "/" characters in the original path name with other characters that adhere to Amazon's naming rules for S3, such as "+". Another way is to use a unique value as the object name, such as a creation sequence number, the creation timestamp, or the MD5 hash value of the data contents. You may need to revisit this decision in Checkpoint 2. Refer to Appendix A for more information on the usage of the cloud API.

2.4 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. Source code documentation will be a part of the grading; please write useful and readable documentation (without generating an excess amount of it). Also, we encourage you to use C++ strings when you need to manipulate strings, as we've seen many students suffering from C string manipulations (but it's your choice).

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, meaning that we expect you to develop your own tests. Note that your project will be graded using a different set of scripts and data-sets that we will not be releasing.

We use `vmstat -d` for the SSD and cloud S3 stats (number of operations, bytes transferred, maximum capacity usage) before and after each of the three operations and then parse the output using a helper program. You can read the man page for `vmstat` if you need help understanding its output format. The README file in the scripts directory has details about using this script.

There are three correctness requirements for Checkpoint 1:

- The files in CloudFS should be identical (including their data as well as their metadata) to the files in the tarball,
- No closed file larger than the threshold is stored in SSD, and

[2] Be warned that some implementations of the UNIX `tar` utility may not provide support for extended attributes, effectively ignoring them. You will be using this utility in Checkpoint 3 to snapshot CloudFS, and we will be reminding you in that handout to take extra care in order to ensure that the extended attributes are retained. You could avoid this hassle, however, by choosing a different approach here.

- Reading metadata only (e.g. `ls -laR`) should not cause any cloud storage access.

Checkpoint 1 grade is determined by tests for these correctness requirements. When grading the report, we may give extra score points for designs that efficiently maintain the metadata on the SSD.

3 Checkpoint 2: Block Level Deduplication

In Checkpoint 1, you were able to transparently increase the size of the file system by backing it up with "infinite" storage capacity provided by the cloud. Every file that was stored in the cloud increased the cost of storage. If you have a lot of files that have the same or nearly the same content, you pay the price of storing the duplicate content again and again. Your work for this checkpoint will attempt to reduce cloud storage costs by eliminating such duplicates, i.e., you will "deduplicate" the data in the cloud.

Deduplication is a term used when a storage system tries to discover duplication among unrelated files and store duplicate content only once.

The simplest way to think about deduplication is to compute a checksum or hash of the entire file, or perhaps each file block, storing the file's or block's hash in a large lookup table. When a new file (or block) is written to the storage system (i.e., added or modified), compute its hash and look up this hash in the lookup table. If the lookup table does not contain the hash, then this data is new, because identical content ensures identical hashes. Next, you will store the new data, insert the new hash and the location of the new data into the lookup table, and set the file (or block) metadata to point to the newly stored data. If the lookup table does contain the hash, then the data *might* already be stored in the storage system. Some deduplication systems do a bit by bit comparison of the new data and any stored data with the same hash, but others select hashes with a lot of bits and strong randomness properties (MD5 for example) so that they can assert that the chance of an accidental collision of two different data objects with the same hash is much much lower than the chance of the data being returned by storage devices incorrectly. For example, recall that for many hard disks 1 bit error can occur for every 10^{14} bits accessed. These systems do not verify a hash collision, but instead assume identical hashes mean identical data without checking.

If the hashes and data match, the lookup table also provides the location of the stored copy of the data, so you can drop the new copy and use the location of the stored copy in the file (or block) metadata. Storage used in the cloud will now be less than if we had not computed and checked hashes.

3.1 Choosing the granularity of deduplication

Using MD5 it is pretty easy to notice identical files. For example:

```
$ ls -l bigfile
-rw-r--r-- 1 palampal palampal 20480 Mar 6 05:03 bigfile

$ cp bigfile copy-of-bigfile

$ md5sum bigfile copy-of-bigfile
7fbc9616d4d275d05629e5cf9415495e bigfile
7fbc9616d4d275d05629e5cf9415495e copy-of-bigfile
```

You could build a deduplicating cloud storage system based on whole-file deduplication like this, where the granularity of deduplication is an entire file "object". It is the easiest to implement, but if the file is an email message with an embedded 10 MB PowerPoint presentation file, and the PowerPoint file is already stored as a separate file in the cloud, then the mail message including the 10 MB attachment is not identical to the PowerPoint file in the cloud. As a result, we will have to store a duplicate of the data in the cloud, for a total of 20MB of data. To make matters worse, if the presentation was broadcast to everyone in the department, wasting gigabytes of storage capacity to store duplicates would be a likely scenario.

Unfortunately, whole file deduplication works only for identical files that are bitwise equivalent. To see why this is problematic, consider the following common file editing scenarios: editing files by gradually appending data to them over time, or making file copies and proceeding to edit those instead. A single bit change in the file, due to editing in place, prepending, or appending data will cause the MD5 hash values to differ across file copies, thereby forcing us to store two full copies of nearly identical content.

```
$ ls -l smallfile
-rw-r--r-- 1 palampal palampal     1 Mar  6 05:06 smallfile

$ cat ./bigfile ./smallfile > ./bigfile-smallfile

$ md5sum ./bigfile ./bigfile-smallfile
7fbc9616d4d275d05629e5cf9415495e  bigfile
3367778e0f263a7879daf956439439d8  bigfile-smallfile
```

One solution to this problem is to split the file into smaller blocks of fixed size, and compute the hash for every block. Now the granularity of detecting duplicated content comes down to the block-size, typically something like 4KB. Note that most of the content in the two files above are identical, except for the last byte that was appended. We can apply the same deduplication technique as before, but at the block level.

In the following example, we split the files into segments of 4KB each. The first five segments from each file will be the same and have the same MD5 hash. So we need to only store the sixth segment of the second file. All other segments can be deduplicated, thus saving us the cost of storage.

```
$ split -b 4096 -d ./bigfile bigfile

$ md5sum ./bigfile0*
463829a1a37bb5fbd36197d1b4b459bc  bigfile00
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile02
130702081e45ffd7facaab4a52da91f2  bigfile03
e13dd6730e63115d52afe056939c5573  bigfile04

$ cat ./bigfile ./smallfile > ./bigfile-smallfile

$ split -b 4096 -d ./bigfile-smallfile  bigfile-smallfile

$ md5sum bigfile-smallfile0*
463829a1a37bb5fbd36197d1b4b459bc  bigfile-smallfile00
```

```
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile-smallfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile-smallfile02
130702081e45ffd7facaab4a52da91f2  bigfile-smallfile03
e13dd6730e63115d52afe056939c5573  bigfile-smallfile04
9fe0f7244a7da1d3f5b3d21f9b1e1ea8  bigfile-smallfile05
```

Block level deduplication has solved our problem in the scenario where new data is appended to a file. However, this naive style of block level deduplication, where each file is divided into fixed-size 4KB blocks, does not work as well if content is duplicated but misaligned.

To demonstrate this problem, we will revisit the previous example. Instead of appending data to the end of `bigfile`, we will now prepend (i.e., add at the beginning) data to `bigfile` to see how the fixed block size deals with this scenario.

```
$ cat ./smallfile ./bigfile  > ./smallfile-bigfile

$ split -b 4096 -d ./smallfile-bigfile  smallfile-bigfile

$ md5sum ./bigfile0*
463829a1a37bb5fbd36197d1b4b459bc  bigfile00
98e8b11cc2c8702066d2323d0ad5c3fa  bigfile01
ccaad733a231d62c5fdd777d808f15b2  bigfile02
130702081e45ffd7facaab4a52da91f2  bigfile03
e13dd6730e63115d52afe056939c5573  bigfile04

$ md5sum smallfile-bigfile0*
f7290d75e8f81c8acd21ee350f759bfe  smallfile-bigfile00
353d2c12147583f287de0576768f957f  smallfile-bigfile01
0c8d327ab961c780d74b2bbd9d880b07  smallfile-bigfile02
fc69cbbec0683ed5250cfe59091a0d5e  smallfile-bigfile03
dbd09bbbbf0fa8657fcb3c01f7203ed6  smallfile-bigfile04
15f41a2e96bae341dde485bb0e78f485  smallfile-bigfile05
```

We can clearly see in the above output that all the MD5 hashes are now different, even though we prepended just one byte of data to the beginning of `bigfile`.

The problem here is that arbitrarily splitting a file into fixed size segments is a solution that is vulnerable to data misalignment. What we need is an algorithm that divides up data sequences into segments based on the content itself, so that files with bit patterns that are identical are likely to be split so the parts that are identical become separate, duplicate segments.

We are **not** asking you to invent such a content specific splitting algorithm. This invention exists, it is called **Rabin Fingerprinting**, and has significantly impacted the way we communicate and store information. The key idea is that a specific pattern/value of a short hash calculated using a rolling window over the file's contents can be declared as a content boundary. Content between two subsequent such boundaries is considered to be part of the same *segment*. Duplicates are then detected at the segment level, and since segment boundaries depend on content, they do not suffer from misalignment issues.

In the scope of this project you are not required to understand Rabin Fingerprinting in depth. In fact, we will provide you with a library that segments files based on the Rabin algorithm. If you feel so inclined, however, you can learn more about the algorithm and its applications through the following publications:

- Udi Manber, *Finding Similar Files in a Large File System*, USENIX Winter 1994 Technical Conference Proceedings, Jan. 17-21, 1994, San Francisco, CA.
- Benjamin Zhu, Kai Li, Hugo Patterson, *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System*, 6th USENIX Conference on File and Storage Technologies, Feb 26-29, 2008, San Jose, CA.

An important consequence of using a splitting scheme like Rabin Fingerprinting is that segments are *not all the same size*. In fact, segment sizes may vary significantly. As a result, an important parameter of the Rabin algorithm is the average segment size, which can be tuned to different values. This average segment size represents the deduplication granularity in this context. For practical reasons, it is also likely that segment sizes will be required to lie between a minimum and a maximum size.

The deduplication granularity is one of the fundamental choices when designing a deduplication system, as it determines the granularity over which you wish to detect duplicate content. As you have probably come to expect in this class by now, there is a trade-off here. A larger granularity requires less metadata (pointers in the files and entries in the lookup table) and less frequent lookups, while a smaller granularity allows us to detect more duplication in the data (as we're looking at smaller groups of bits), allowing for higher space savings.

3.2 Design Specification

The goal for Checkpoint 2 is to implement variable size segment deduplication for the data content stored in CloudFS. To simplify the implementation, we ask you to apply deduplication only to files that are to be stored in the cloud (i.e., files bigger than a given threshold, not all files). So instead of automatically shipping the large file to the cloud (as you did in Checkpoint 1), you will hand it over to a dedup code layer, which should then do its magic and only store unique segments in the cloud, thereby reducing the data capacity consumption on cloud storage and the corresponding data transfer costs. Your design and implementation of a deduplicating system will be measured against lowering these cost metrics, however **you should not implement any type of caching to improve on cloud costs yet**. We will be examining caching in Checkpoint 3.

You should be able to enable/disable the de-duplication for one run of your operation through an argument on this invocation command line. Deduplication should be enabled by default. Specifying `--no-dedup` should run the system without deduplication. This will aid you in debugging and us in testing your program. A layered design with clearly defined interfaces will help you here and throughout the project. You can assume that we will not be switching dedup modes at runtime, i.e., while your file system is mounted. The expected command line syntax for this parameter when starting CloudFS should be as follows:

```
./CloudFS [--no-dedup] [--other-args]
```

There are various approaches to designing the dedup subsystem, but all designs should be able to cope with addition and deletion of a few bytes in the files without losing (all of) the cost savings benefit. The implementation will be measured against the total cost savings in the cloud.

3.3 Identifying the segment boundaries

We will be providing you with a library that implements the Rabin Fingerprinting algorithm and uses it to partition file data into segments. A sample program that prints out the MD5 checksums of segments in a given file will also be included along with the support code.

To understand the Rabin Fingerprinting algorithm you should first look at the way it uses a rolling hash algorithm, hashing a window of data in the file at every byte offset. A fast implementation incrementally calculates the hash values over a given window of bytes (say w bytes). We start at the first byte and slide the window forward, one byte at a time, while calculating the Rabin fingerprint at every byte offset in the file. To determine a segment boundary, we look for a specific bit pattern in the generated Rabin fingerprints. For example, we can mark a segment boundary every time all b least significant bits of the Rabin fingerprint are equal to zero, so that 2^b bytes will be the average segment size. To guard against a string of nulls in the data, or a huge segment, there will also be a minimum segment size and for ease of implementation, a maximum segment size.

Your CloudFS implementation should support four parameters for the Rabin Fingerprinting algorithm: the average segment size (`--avg-seg-size`), minimum segment size (`--min-seg-size`), maximum segment size (`--max-seg-size`), and window size (`--rabin-window-size`), allowing users to override the defaults. You are required to use the values provided in these parameters, (i.e., you are not permitted to overwrite these parameters in `cloudfs_init`). These parameters extend the CloudFS command-line syntax as follows:

```
./CloudFS [--avg-seg-size <Kilobytes>] [--min-seg-size <Kilobytes>]
          [--max-seg-size <Kilobytes>] [--rabin-window-size <number>]
          [--other-args]
```

We will parse these arguments for you, and fill up appropriate fields in the instance of the `struct cloudfs_state` that is made available to CloudFS. You should use the same values in your code.

3.4 Identifying the duplicated segments

Consider a file system having 40 GB of data with an average segment size of 4 KB. In this case you will end up storing and searching through 10 million hash values. And to be fair, 40 GB is not a lot of data for a file system backed by the cloud. Thus, you need a good algorithm to search through the hash values efficiently. The choices range from a simple hash table implementation to databases with Bloom Filters or B+ trees.

Another problem is that you will most likely be unable to fit everything in this lookup table into main memory. By saying this, we are not challenging you to attempt it! In fact, in the real world this data structure needs to be mostly on storage, with only as much of it in memory as is needed to achieve sufficient performance. The cost of reconstructing this data structure at startup can also be prohibitive, so many implementations of this data structure have to be stored persistently on disk, and kept up-to-date.

To simplify the design and implementation, we allow you to assume that all the hash values of all segments in the file system fit into main memory. In this case a simple hash table should be sufficient, if it is persistently backed on SSD. We will also allow you to recreate the data structure by doing a LIST on the segments stored in the cloud, provided you have a good naming scheme for the segments.

We are not looking for a fancy data structure; make it fast, persistent, as small as possible, but above all: simple. Note that the hash values that are being stored and searched in this lookup table are MD5 hashes of the content in the segments; these hashes have nothing to do with the hash that Rabin Fingerprinting algorithm generates internally to select the segment boundaries.

3.5 Handling I/O on segmented files

So far we have discussed the mechanisms involved in segmenting and deduplicating a file. These mechanisms kick in as soon as the file exceeds the size threshold that requires it to be moved to cloud storage. In other words, starting with Checkpoint 2 **you should assume that when a file exceeds the size threshold you may not have enough memory or space on the SSD to buffer its entire contents**. Consequently, you should start segmenting the file immediately, sending segments to the cloud. This subsection will guide you in handling read and write requests to files that already reside in the cloud, in deduplicated segments.

Reading from a segmented file should not require that you read in the entire file from cloud storage. In fact, you **must** assume that *a file stored in the cloud may not fit entirely on the SSD, or in memory*. As a result, you need to make sure that you only read the relevant segments in.

Writing to a segmented file must not violate the assumption that the file's segment boundaries have been decided by Rabin's algorithm. This means that if one or more bytes at any offset of the file are overwritten by a write request, you must rerun Rabin's algorithm to determine if (and how) the segment boundaries have changed.

Thus, on each random write, you would have to read the affected segments from cloud storage, modify the bytes, re-segment the bytes using Rabin's algorithm, update your segment mappings, and write the new segments back to cloud storage. This might lead you to read data of more segments than those that were actually touched by the write, because the segment boundaries previously defined by Rabin's algorithm might have changed. The important point we wish to make, is that **at all points, the data stored in segments must have been defined by Rabin's algorithm**. As mentioned above, you should not buffer the entire file's contents in memory or on SSD, as you have no guarantee there will be sufficient memory or space on the SSD.

Finally, writes in traditional file systems can occur beyond the end of a file. Those writes are used to create *sparse files*, for which we store only blocks that have been written to, and reads to other offsets return zeros. You are **not** expected to support sparse files in CloudFS, and we will not be testing or awarding points for supporting this feature. We will, however, appreciate the effort.

3.6 Mapping files to segments

The last piece of the dedup puzzle is to tie down the newly created segments to the file they belong to. Segment names and associated metadata are internal to your file system and are not exposed to the user. You have to build and maintain a mapping between the big file stored in the cloud, and its list of segments. You can come up with various ways of achieving this mapping. You could create a on-SSD data structure that maps a key (filename) to values (all the segments that belong to the file). One of the simplest approaches is to use the hidden proxy file that you may have used in Checkpoint 1 to store the attributes of the file in cloud. You could store the segment identifiers in this file. Do note that you need to also maintain the correct ordering of the segments.

3.7 Deleting segments on file deletion

You must be able to recover the cloud storage space used by a segment whenever all files to which the segment belongs get deleted. To achieve this, you should come up with a reference counting scheme for the segments stored in the cloud. One option you could choose is storing the reference counts as part of the lookup table used to search for duplicate segments.

3.8 Test and Evaluation

Your grade on Checkpoint 2 will be determined by tests for correctness as well as tests for capacity and cost reduction achieved through deduplication. As with checkpoint 1, we may penalize for inefficient (in terms of performance and/or cost) designs and/or award extra score points for efficient designs when grading the report.

The test script for Checkpoint 2 generates a large number of big files. The files will sometimes share data segments with other files. At the end of test, capacity consumption of cloud storage is measured. As in Checkpoint 1 tests, the Checkpoint 2 test is also wrapped around two measurement sources: `vmstat` and cloud service statistics. In addition to the correctness tests from checkpoint 1, checkpoint 2 also tests that deduplication should result in cloud storage cost savings and that you design adheres to the constraints.

4 Project Resources

The following resources are available in the project distribution that can be downloaded from Autolab.

- **CloudFS starter code**

  The files inside `src/cloudfs/` are the skeleton code that you will modify and extend for your project. `cloudfs.h` and `cloudfs.cc` contain the skeleton code for FUSE file system. You can refer to the fuse header (`/usr/include/fuse.h` in your VM) to learn about supported FUSE operations and their parameters. `cloudapi.h` and `cloudapi.c` contain wrapper functions for the libs3 C library. The file `cloud-example.c` gives you an example of how to use our wrapper of libs3 to communicate with our Amazon S3-like server. The file `rabin-example.c` gives you an example for Rabin Segmentation API. Use the `make` command under `src/cloudfs/` to create the binary code of `cloudfs` in the directory `src/build/bin/cloudfs`.

- **AWS instance deployment**

  Detailed instructions on the usage of AWS instances for CloudFS development are included in the distribution under `src/aws/README.aws`. The key necessary for accessing the instance is also available in the same directory, under the name `746-student.pem`.

- **AWS instance setup scripts**

  There are three scripts: `format_disks.sh`, `mount_disks.sh`, and `umount_disks.sh`, that are required to manage the AWS instance environment with the SSD. 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. Tools such as `gdb`, `blktrace`, and `valgrind` have already been installed in the provided AWS template.

- **Amazon S3 simulation web server**

  The file `src/s3-server/s3server.pyc` is compiled python code that simulates the Amazon S3-like cloud storage service. To run the web server, you can use the command: `python s3server.pyc`. The web server depends on the Tornado web server framework (`http://www.tornadoweb.org/`), which has been already installed in the provided AWS template. It stores all the files by default in `/tmp/s3/` (do not change this). To enable logging, you can simply run it with the `--verbose` option. You can see more options by using the `--help` option.

5 Deliverables

The homework source code and project report will be graded based on the following criteria (Note: this is a tentative breakdown and is subject to change).

- **Checkpoint 1:** Hybrid file system spanning SSD and cloud storage **30%** of grade
- **Checkpoint 2:** Block-level deduplication **30%** of grade
- **Checkpoint 3:** Snapshots and Caching **30%** of grade
- **Final project report:** Project report and source code documentation **10%** of grade

For each of the Checkpoint 1, 2, and 3 milestones you should submit a .tar file that contains only a directory `src/` with the source files of CloudFS. You should use the same code structure as in the archive you downloaded from Autolab, and make sure that there exists a Makefile that can generate the binary `src/build/bin/cloudfs`. We will test this in Autolab, and your code should compile correctly (Make sure to test this yourself!).

In your final project report submission you should submit a file `AndrewID.tar.gz`, where `AndrewID` is *your* Andrew ID, which should include at least the following items (and structure):

- A `src/` directory with any test suites you used for evaluation in your report. Please keep the size of test suite files small ($\leq$ 1MB), otherwise omit them. You will have already made your final code submission in Checkpoint 3, so **do not resubmit code here.**
- A `AndrewID.pdf` file containing your final project report, which should consist of **4 pages or fewer**. More information on the contents of this report follow.
- A `suggestions.txt` file with suggestions about this project, i.e., what you liked and disliked about the project and how we can improve it for the next offerings of this course.

5.1 Source code documentation

The `src/` directory should contain all your source code files. Each source code file should be well commented to highlight the key aspects of each function, while avoiding very long descriptions. This applies to each checkpoint submission you will make. Feel free to look at well-known open source projects' codebases to get an idea of how to structure and document your source code.

5.2 Final 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`. By now, you should know how annoying it can be to your instructors when you forget to present your Andrew ID front and center!

The report should contain design and evaluation of all three checkpoints, i.e., discuss Checkpoint 1, Checkpoint 2 and Checkpoint 3 goals, design, evaluation and evidence of success. 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.

Your report must answer the following questions explicitly:

- Explain all the cost/performance trade-offs you encountered as part of this project
- Explain the trade-offs in choosing the right average segment size for deduplication
- Explain how deduplication is used in your design
- Explain your snapshot design, and the techniques you used to minimize cloud costs
- Explain your cache replacement policy, and how your policies translate into (hopefully) observed performance improvement and cost savings

5.3 Submissions

As always, you will submit your work through Autolab and will not be penalized for the first 25 submissions. Autolab test results will be used in your grade, but we will also perform additional tests that are not available to you. For the final submission, make sure to use the same directory structure provided in the code handout.

As a final reminder, make sure to **use the SSD for all CloudFS data that must be stored locally**. Specifically, you should not utilize locations outside your FUSE-mounted file system to store CloudFS data (or metadata). This excludes any information directed to `/tmp/cloudfs.log` for logging purposes.

5.4 Useful Pointers

- `https://github.com/libfuse/libfuse` is the de-facto source of information on FUSE. If you download the latest FUSE source code, there are examples included in the source. In addition, the FUSE documentation might prove helpful in understanding FUSE and its data structures:

  `http://libfuse.github.io/doxygen/`

  You can search Google for tutorials on 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 it 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 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/`

A Appendix

A.1 Autolab debugging

Please direct all information you want to log to `/tmp/cloudfs.log`. The contents of this file will be displayed in your Autolab output.

```
logfile = fopen("/tmp/cloudfs.log", "w");
/* To ensure that a log of content is not buffered */
setvbuf(logfile, NULL, _IOLBF, 0);
int fuse_stat = fuse_main(argc, argv, &cloudfs_operations, NULL);
```

Use this code snipped to debug your code. Please note that there is a limit on the number of log lines we can print on Autolab, so be conservative about logging. Also, please ensure the placement of `fopen` is before `fuse_main()`.

A.2 Amazon S3 API Specifications

To simulate the Amazon S3 cloud storage environment, we provide you with a web server running locally in AWS instance. 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 an open-source S3 client-library called libs3 in FUSE to allow CloudFS to communicate with the 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. In this project, however, we only need to use a subset of its full functionality. For your convenience, we provide a wrapper of libs3 C library 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 code snippet 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 libs3 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 the 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 the `cloud_get_object` call, it retrieves data from the 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/cloud-example.c`.

A.3 Rabin Segmentation API Specifications

The Rabin segmentation API should be used to define the segment boundaries. First you need to initialize the data structures by calling `rabin_init()`. Once initialized, use `rabin_segment_next()` to run the contents of a file through the Rabin fingerprinting algorithm. You may have to call the latter function multiple times in a loop. Once you are done with all the fingerprinting for one file, you can call `rabin_free()` at the end. You can use `rabin_reset()` to re-initialize the data structure. We suggest you perform initialization only once at startup and then use `rabin_reset()` to use the same data structure for multiple files. An example program that uses this API is provided with the source code. It prints out the segment lengths and their MD5 sums. To build the example do `make rabin-example`.

```
cloudfs $ make rabin-example
build/obj/rabin-example.o: Compiling object
build/bin/rabin-example: Building executable

cloudfs$ ls -l /tmp/bigfile /tmp/smallfile
-rw-r--r-- 1 guest guest 20480 2013-03-17 17:06 /tmp/bigfile
-rw-r--r-- 1 guest guest     1 2013-03-17 17:07 /tmp/smallfile

cloudfs$ cat /tmp/bigfile | ./build/bin/rabin-example
3190 cb26f4d170a93009e0d1c5b29b31796e
7862 409348f2fdd9aa2d18641a0d3d113108
3868 5e27fd72f47bb5a263f6443e39a8c5d7
5560 d61085ec3b8749ff57c4bbb4590760b3

cloudfs$ cat /tmp/smallfile /tmp/bigfile | ./build/bin/rabin-example
3191 c62235153f6148d2cc9fb94ef576b57b
7862 409348f2fdd9aa2d18641a0d3d113108
3868 5e27fd72f47bb5a263f6443e39a8c5d7
5560 d61085ec3b8749ff57c4bbb4590760b3
```

That right here is Rabin Fingerprinting at work!

Below is the API Interface header `dedup.h`:

```
/**
 * @file dedup.h
 * @author Pavan Kumar Alampalli
 * @date 15-mar-2013
 * @brief Interface header for dedup library
 *
 * This header file defines the interface for the dedup library.
 * The interface follows the basic init, call-in-a-loop, free
 * pattern.  It also declares an opaque structure (rabinpoly_t)
 * that will be used by all the functions in the library to maintain
 * the state.
 *
 * Note that the memory allocated by rabin_init() has to be freed
 * the calling rabin_free() in the end. You can reuse the same
 * rabinpoly_t structure by calling a rabin_reset().
 */

#ifndef _DEDUP_H_
#define _DEDUP_H_

/**
 * Rabin fingerprinting algorithm structure declaration
 */
struct rabinpoly;
typedef struct rabinpoly rabinpoly_t;

/**
 * @brief Initializes the rabin fingerprinting algorithm.
 *
 * This method has to be called in order to create a handle that
 * can be passed to all other functions in the library. The handle
 * should later be free'ed by passing it to rabin_free().
 *
 * The window size is the size of the sliding window that the
 * algorithm uses to compute the rabin fingerprint (~32-128 bytes)
 *
 * @param [in] window_size Rabin fingerprint window size in bytes
 * @param [in] avg_segment_size Average desired segment size in KB
 * @param [in] min_segment_size Minumim size of the produced segment in KB
 * @param [in] max_segment_size Maximum size of the produced segment in KB
 *
 * @retval rp Pointer to a allocated rabin_poly_t structure
 * @retval NULL Incase of errors during initialization
 */
rabinpoly_t *rabin_init(unsigned int window_size,
                        unsigned int avg_segment_size,
                        unsigned int min_segment_size,
                        unsigned int max_segment_size);

/**
 * @brief Find the next segment boundary.
 *
 * Consumes the characters in the buffer and returns when it
 * finds a segment boundary in the given buffer. The segments
 * defined by this function will never be longer than max_segment_size
 * and will never be shorter than min_segment_size.
 *
 * It returns the number of bytes processed by the rabin fingerprinting
 * algorithm. Note that it can be <= the number of bytes in the
 * input buffer depending on where the new segment was found (similar
 * to shortcounts in write() system call). So you may need to call
 * this function in a loop to consume all the bytes in the buffer.
 *
 * @param [in] rp Pointer to the rabinpoly_t structure returned by rabin_init
 * @param [in] buf Pointer to a characher buffer containing data
 * @param [in] bytes Number of bytes to read from the buf
 * @param [out] is_new_segment Pointer to an integer flag indicating segment
 *                             boundary. 1: new segemnt starts here
 *                                       0: otherwise.
 *
 * @retval int Number of bytes processed by the rabin algorithm.
 *         -1  Error
 */
int rabin_segment_next(rabinpoly_t *rp,
                       const char *buf,
                       unsigned int bytes,
                       int *is_new_segment);

/**
 * @brief Resets the Rabin Fingerprinting algorithm's datastructure
 *
 * Call this function to reuse the rp for a different file or stream.
 * It has the same effect as calling rabin_init(), but does not do
 * any allocation of rabinpoly_t.
 *
 * @param [in] rp Pointer to the rabinpoly_t structure returned by rabin_init
 *
 * @retval void None
 */
void rabin_reset(rabinpoly_t *rp);

/**
 * @brief Frees the Rabin Fingerprinting algorithm's datastructure
 *
 * This function should be called at the end to free all the memory
 * allocated by rabin_init.
 *
 * @param [in] p_rp Address of the pointer returned by rabin_init()
 *
 * @retval void None
 */
void rabin_free(rabinpoly_t **p_rp);

#endif /* _DEDUP_H_ */
```