15-746 Project 1: myFTL
Handout 1

September 2, 2020

Contents

Overview

For this project, you will be designing and implementing a Flash Translation Layer (FTL), your very own (and very creatively named) myFTL. FTLs are responsible for abstracting away the details of internal SSD architecture and characteristics from the host OS. For ease of development and debugging, you will by building myFTL on an emulated SSD.

FTLs, and by extension, this project, consists of three components: an address translation component that maps Logical Block Address (LBAs) to physical pages in the SSD; a garbage collection component that removes invalid data and makes space for new data and a wear leveling component that ensures that all the blocks in the SSD wear out at a near-uniform rate. To help ensure you progress steadily through the project goals, we have imposed four **graded** intermediate milestones (that build on one another) involving source code submissions:

Milestone	**Description**	**Deadline**
Checkpoint 0	Setup	11:59pm EST on Friday September 4, 2020
Checkpoint 1	Address translation	11:59pm EST on Tuesday September 15, 2020
Checkpoint 2	Garbage collection	11:59pm EST on Friday September 25, 2020
Checkpoint 3	Design FTL policies	TBD
Report	Final project report	TBD

Table 1: Deadlines for the three checkpoints and report.

The last submission you make for Checkpoint 3 will be your final code submission. Your final submission will be style graded following 213 guidenlines(`https://www.cs.cmu.edu/~213/codeStyle.html`). There will be a couple of days between the final code hand-in and the final project report hand-in. This will leave you a little time to think through and carefully explain your design decision. We strongly encourage you to not wait till the last minute before writing your report, because **your report is also graded**.

You must submit your code to Autolab (`http://autolab.pdl.cmu.edu`) for each milestone. We will provide you with some of the test cases used for evaluating your myFTL. You are strongly encouraged to write your own tests to ensure correctness of your myFTL. We will also be reviewing and grading your code quality (cleaniless, modularity, comments, etc).

The rest of this document describes the specification of myFTL in the context of the first two project checkpoints. We begin by describing the architecture of SSDs in section 1 followed by a discussion of FTL functionalities in section 2. Section 3 describes the SSD emulator and the evalution criteria. Section 4 details getting started with AWS. Sections 5 and 6 describe the first two main checkpoints. A separate handout will be released for checkpoint 3. Appendix A details the emulator code structure including the classes and control flow. Appendix B describes useful hints and tools for debugging your myFTL. Appendix C describes how to set up and use Amazon AWS for developing and/or testing your myFTL. Finally, Appendix D describes how to test your myFTL with real workloads. Note that running or testing with real workloads is neither a required nor a graded (i.e. no bonus points) part of this project.

1 Solid State Disk Architecture

Solid state disks (SSDs) are storage devices that use flash chips (typically in the form of NAND chips) to store information. Since you will be building software for them, what better place to start than looking at the architecture of SSDs?

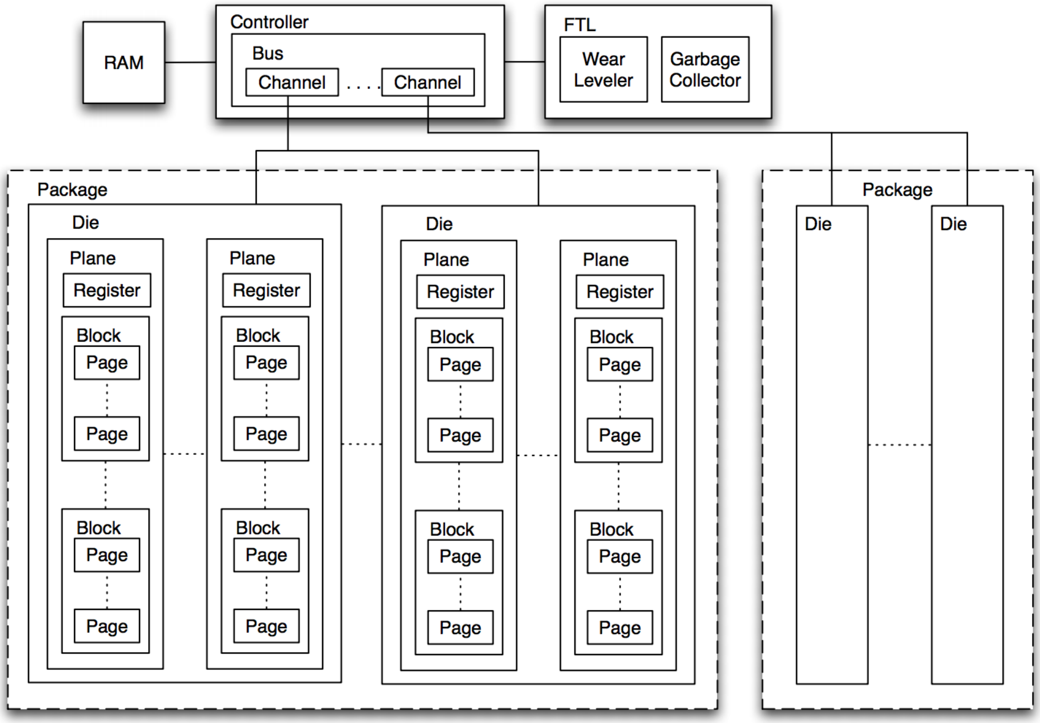


Figure 1: Basic SSD Architecture

As you can see in Figure 1, the architecture of an SSD consists of multiple tiers. More specifically, a typical SSD consists of *packages*, each of which consists of multiple *dies*, each of which consists of multiple *planes*, each of which consists of multiple *blocks*, and each block is finally made up of multiple *pages* (phew!). As you will soon realize, the terms *block* and *page* carry a plethora of meanings and sizes in different contexts. In the context of SSDs, a block of data typically stores 128 KB worth of data pages. Each SSD page is typically 4 KB, and it is the smallest unit of data storage on an SSD.

SSD packages, dies, planes, blocks and pages and not visible to software layers outside the device. Upper layers, such as file systems and user programs, view the SSD as a series of Logical Block Addresses (LBAs). In the context of this project, each LBA refers to 4 KB of data, or an SSD page[1]. This illusion is created by the Flash Translation Layer (FTL), a piece of software responsible for translating Logical to Physical Block Addresses (LBA → PBA) and vice versa. The PBA is the address where the data is stored on the device, and as we will see later, this abstraction between logical and physical addresses allows the movement of data, allowing the SSD to be used more efficiently without complicating the applications that use it. The raw capacity of an SSD is calculated by the following formula:

$$SSD_{capacity} = K \times D \times L \times B \times P \times S,$$

where K is the total number of packages in the SSD, D is the number of dies per package, L is the number of planes per die, B is the number of blocks per plane, P is the number of pages per block, and S is the size of a page in bytes.

An important characteristic of SSDs is that the granularity of erase operations is different from the granularity of read/write operations[2]. In SSDs, erases can only happen at the block level, while reads and writes can take place at the page level. Since a block consists of multiple pages, an erase of a block destroys data in all pages contained

[1] Now you see why we were saying that the terms *block* and *page* can be confusing. Please bear with us, we're equally annoyed.

[2] This is unlike traditional HDDs, where a sector is the unit of reads and writes and there are no erases.

in that block. Moreover, once written, a page cannot be overwritten until the block in which it is contained is first erased. The framework designed for this project is equipped with the necessary checks to ensure correct emulation of an SSD. If your code attempts to perform an invalid read or write operation, the SSD emulator will reject it. One exception to this is attempting to read a page that was never written, which is an operation that our SSD emulator will refuse to perform by returning an error. In real life, reading a page that was never written may return garbage content (i.e. stuff already present in that page). This design choice enables better testing/debugging and, as a storage systems' developer, you must make sure you do not read an unwritten page.

SSDs can safely do only a limited number of block erases (in effect, a limited number of page rewrites). After a block has been erased a certain number of times (usually in the tens of thousands), there is a significant probability of data corruption in that block. Due to the fact that in-place page rewrites are disallowed and the only way to rewrite a page is via a block erase, a log-structured data management scheme is a good match. We can think of blocks as segments of a log-structured mechanism, which also implies block erases involve cleaning analogous to segment cleaning (or garbage collection) and similar principles as log-structured file systems can be used to deal with SSDs effectively.

SSDs can service multiple requests in parallel, rather than one-at-a-time like traditional HDDs where only one disk head can be active at a time. This aspect, along with the fact that there are no moving parts in an SSD, has resulted in SSDs being both highly performant and energy efficient.

2 Flash Translation Layer

In this project, you will build a flash translation layer (FTL) for an emulated solid state drive (SSD). As you know, SSDs rely on firmware in the drive to perform multiple functions. When SSDs were first introduced, their radically different architecture, and the asymmetry between reads and writes (due to erase operations), required a completely different way of reading or writing to them. The Flash Translation Layer (FTL) is an abstraction introduced to maintain current file system and driver code as it is. The FTL translates the commands issued by file systems and user programs to an SSD-friendly format. Usually the FTL is built into the SSD firmware.

2.1 Translation Logic

Since an SSD has little DRAM, we cannot afford to keep a giant hash table to map every LBA to a physical address. One way of reducing the mapping table size is to associate logic in deriving a PBA from an LBA, which is reminiscent of virtual-to-physical address translation in the context of virtual memory. This logic is a crucial component of this project.

2.2 Garbage Collection

As mentioned earlier, an SSD page can only be rewritten following an erase of the block that contains the page. Suppose we have a workload where a series of writes and deletes are performed on independent files (no files are overwritten). Without performing erase operaitons, the SSD will eventually fill up. At this point, the SSD utilization, i.e., percentage of the SSD that contains active data, is less than the total capacity since we have performed deletions. But, in order to perform even one more write, we need to first erase at least one block. The process of reclaiming space to use is more involved than merely erasing a block. If there is even one page that contains valid data in the block, we need to make sure we retain that page's data even after performing the erase of the block. The action of tracking live data, copying it to some other location, erasing the block, and restoring the live data in that block is called *garbage collection*. This action is very similar to the segment cleaning performed in log-structured file systems (LFS [1]), where log segments are analogous to blocks in the SSD context. You will need to do garbage collection as a part of this project in Checkpoint 2. The relevant details for this part of the project are discussed later in the handout, and the architecture of LFS will also be discussed in class before you start Checkpoint 2.

2.2.1 TRIM

Since we are discussing the issue of garbage collection in SSDs, it is incomplete without mentioning TRIM. TRIM is a (relatively) late addition to the set of commands that can be issued by the file system. TRIM is used by the layers above the SSD to inform the SSD that certain LBAs do not contain live data. For example, suppose a file was deleted. The file system can issue a series of TRIMs to the SSD for the LBAs containing the file data, to inform the SSD that the data in those LBAs is not valid anymore. What would be the incentive for doing this? The reason is garbage collection efficiency. Let us consider a situation where garbage collection has been invoked by the SSD firmware. Having knowledge that the data in certain SSD pages (which held the most recent data for the TRIM-med LBAs) is no longer valid reduces the amount of work that the firmware has to do (fewer valid pages to copy around). Thus, TRIM is simply a convenience method aimed at reducing the work the garbage collector has to perform during garbage collection. **Note:** You are not required to support TRIM for the first two checkpoints. There is a TRIM function in the `myFTL.cpp` file provided to you but you can leave it as is. The tests for the first two checkpoints will succeed even if TRIM is a no-op as long as it returns SUCCESS. However, if you do implement TRIM, make sure it is correct, because a wrong implementation might cause the tests to fail. For Checkpoint 3, you might want to implement TRIM in order to improve your FTL design (we revisit this topic later).

2.3 Wear Leveling

Wear leveling can be thought of as being analogous to load balancing in a distributed system. Since an SSD block can tolerate a limited number of erase operations before it fails due to wear, we need to ensure that we do not end up erasing a few blocks too many times, while the others are erased much less often. As an example, consider a journaling file system. In a journaling file system, typically the journal is located at a fixed area on disk. The journal is updated for every write that occurs in the file system, making the on-disk space occupied by the journal a *hot spot*, i.e. a region more frequently updated than others. Other examples of hot spots are metadata blocks of a file system, such as the block bitmap and the inode bitmap. It is unreasonable to expect legacy file system codes to change in order to take SSD wear leveling into account (however, newer file systems designed specifically for SSDs can and try to do so). As a result, wear leveling also becomes the job of the firmware, which increases the complexity of FTL mappings, since the mapping can dynamically change based on wear. In Checkpoint 3, you will need to add wear leveling to improve the efficiency of your FTL.

2.4 Over-provisioning

Usually there is a difference between the raw capacity of an SSD and the addressable capacity (i.e., the last addressable LBA) of the SSD. The difference in size is called *over-provisioning*. Over-provisioning can be useful for several reasons, but typically and for the purpose of this project, it is for the following two reasons:

Garbage collection. While garbage collecting, we copy the live data of the block that is to be cleaned into a different block, erase the original block, and optionally copy the content back to the original block. If we let the entire raw capacity of the disk fill up with live data, there will be no free blocks left which we can use for the cleaning activity. In order to avoid this, a few blocks are reserved for the purpose of garbage collection.

Durability. Another use of the overprovisioned space is for durability or wear leveling. In the case of a direct (and never-changed) one-to-one LBA to PBA mapping, if a workload overwrites the same LBA over and over again, with each overwrite, we will need to erase the block for each write to the requested LBA. Sophisticated FTLs reserve a few blocks to account for such scenarios and prevent the same block from being erased too often. These reseved blocks usually act as logs or journals and they accomodate multiple changes to the page being rewritten in quick succession with the last value being considered as the most correct value. Until the log is full, no erases need to be done, thus extending the life of the SSD.

Thus, overprovisioning is essentially a tradeoff of space for durability and efficiency. You too will have to make this tradeoff in the FTL you will design. Ideally, you would want to have as low overprovisioning as possible, keeping in mind that the durability of the SSD is kept reasonably high (for example, in this project, a given workload containing rewrites that must complete successfully before the SSD fails due to a particular block being erased more times than it is allowed to).

3 SSD Emulator and Evaluation

While an FTL is often part of an SSD's firmware, testing your code on real hardware can be tedious (not to mention, expensive!). To make this process more pleasant, we provide you with an SSD emulator that will act as an SSD. You will be working with the `746FlashSim` emulator, built here at CMU and inspired from work done at Penn State University [2].

746FlashSim is an object-oriented approach to emulating SSDs written in C++. This means you will also be developing this project in C++. This project does not require prior knowledge of object-oriented programming or C++, however. The functions you need to write are clearly highlighted in-code, and you can use C to flesh them out. You also are not expected to use or learn object-oriented programming principles such as polymorphism and inheritance. Should you happen to be versed in standard C++ data structures such as maps, sets, etc., your life may become easier. Learning them in case you don't know them already is straight-forward. Since C++ is richer than C in complex data structures, you are *not* allowed to use any other external library for data structures. Remember that other than C, most other low-level programming in the field is done in C++. So, it may not be a bad idea to learn it a little through this project. Finally, the TAs are not going to tutor you in C++ or object-oriented programming, so please refrain from asking those questions in office hours.

Your code should reside only in `myFTL.cpp`. Your job is to add logic to the `ReadTranslate` and `WriteTranslate` methods for the myFTL class already created for you in `myFTL.cpp`. The methods are currently stub functions (i.e. empty functions) that you must develop. You must not change any file other than `myFTL.cpp`. You may add as many new functions to this file as you deem fit.

```
std::pair<ExecState, Address>
  ReadTranslate(size_t lba, const ExecCallBack<PageType> &func) {
  // address translation from LBA to PBA for READ event type.
}

std::pair<ExecState, Address>
  WriteTranslate(size_t lba, const ExecCallBack<PageType> &func) {
  // address translation from LBA to PBA for WRITE event type.
}
```

The above methods are the most crucial methods in this project. As the name suggests, this method translates an LBA to a PBA. While running tests, we are going to translate addresses (by calling the above methods) to verify the translation logic implemented. You are also highly encouraged to perform and test address tranlations by yourself to ensure correctness of your logic. **The FTL logic without garbage collection or wear leveling is the objective of Checkpoint 1.**

These methods take an LBA and an instance of `ExecCallBack`, and return whether the operation was a SUCCESS or a FAILURE and the translated PBA. The provided instance of `ExecCallBack` can be used to issue reads, writes, and erases while performing an address translation (e.g., while performing garbage collection caused due to a particular write). For instance, say your SSD's BLOCK_SIZE is 32 and your FTL wants to issue a read to page 0 of block 0 followed by a write to page 0 of block 1 followed by an erasure of block 0. You would do so in the following way:

```
... some code here ...
func(OpCode::READ, Address(0, 0, 0, 0, 0))
func(OpCode::WRITE, Address(0, 0, 0, 1, 0))
/* For erasing a block, use address of any page in that block */
func(OpCode::ERASE, Address(0, 0, 0, 0, 0))
... some code here ...
```

Garbage collection is an event spawned if needed during address translation when a write request can only proceed by at least one segment cleaning cycle. There are multiple segment cleaning policies that you will be required to develop in Checkpoint 2, whose selection will be a part of the config file. A segment cleaning policy, once selected, will apply for the entire test.

Wear leveling is required to prevent *hot spots*, i.e. to prevent certain blocks from getting more worn out than others. Remember that if even one block of the SSD is worn out (i.e., it reaches its maximum limit of allowed erases), most SSDs will declare themselves defunct. You will be responsible for implementing one wear leveling policy in Checkpoint 3.

3.1 Execution

The only way to run the SSD emulator is through a C++ program. In the handout, we provide you with a few tests to check the correctness of your implementation. The tests are organized in the following directory hierarchy:

```
tests/
|
|---- checkpoint_1/
      |
      |---- test_1_1/
            |
            |---- test_1_1.conf
            |---- test_1_1.cpp
      |---- test_1_2/
      |---- ...
|---- checkpoint_2/
      |
      |---- test_2_1/
      |---- ...
|---- ...
```

The Makefile contains targets for these tests. The way to build and run these tests are as follows:

```
make run_test_<checkpoint>_<test>
E.g. to run test 2 of checkpoint 1, make run_test_1_2
```

The tests issue events (read/write) to the myFTL object which are logged in a log file created at `output/test_CHECKPOINT_TEST.log`. At the very least, a test program must have the following snippets of code in it:

```
#include "746FlashSim.h"
...
int main(int argc, char *argv[]) {
  int ret;
  size_t lba = 0;

  init_flashsim();

  Test test(argv[1]);
  uint32_t page_value = 15746;
  ret = test.Write(log_file_stream, lba, page_value);
  if(ret != 1)
    printf("FAILURE\n");
  else
    printf("SUCCESS\n");
  uint32_t read_value;
  ret = test.Read(log_file_stream, lba, &read_value);
  if(ret != 1 || read_value != page_value)
    printf("FAILURE\n");
  else
    printf("SUCCESS\n");

  deinit_flashsim();

  return ret;
 }
```

Note that we are using 4 byte (`uint32_t`) as the page contents. This is to reduce the amount of data we need to store (to cross check after reading). Actual page sizes are typically 4KB and the simulator can be configured to run with 4KB pages as well, as explained in Appendix A.3.3. Configuration file (*tests/ checkpoint_1/ test_1_1/ test_1_1.conf* above) details are explained below. You can use the `GetInteger` method from the conf object passed to the constructor of myFTL to get the values of the parameters present in the configuration file. The tests designed to test the SSD are similar C++ programs with events aimed at checking the correct functioning of the FTL logic you will develop. You are highly encouraged to write such programs of your own in order to test your FTL. The tests we provide you are a subset of the tests we will use to grade the project. Therefore, the more you test, the better.

3.2 Configuration

The configuration file contains several knobs to characterize and tune the SSD's behavior. An example config file is:

```
# Number of Packages per Ssd (size)
SSD_SIZE 4

# Number of Dies per Package (size)
PACKAGE_SIZE 8

# Number of Planes per Die (size)
DIE_SIZE 2

# Number of Blocks per Plane (size)
PLANE_SIZE 64

# Number of Pages per Block (size)
BLOCK_SIZE 16

# Number of erases in lifetime of block
BLOCK_ERASES 500

# Overprovisioning (in %)
OVERPROVISIONING 5

# Selected garbage collection policy
# Relevant for checkpoint 2
SELECTED_GC_POLICY 0
```

You should try playing around with the knobs, especially over-provisioning percentage and block erases, to check for boundary conditions and make sure your logic is sound in corner cases. Also note that the maximum size of the SSD is bounded by the data type (unsigned long) used for the package, die, plane, and block. Setting these values above the size of the datatype (on your architecture) might result in unexpected behavior. The over-provisioning, being a percentage, may result in the number of log blocks not being an integer. As a toy example, 2% over-provisioning with 64 blocks, would mean 1.28 log blocks, which is not possible. In such situations, you should round off the number of blocks to the nearest integer, e.g., 1 block in this case.

3.3 Evaluation

Usually, evaluation of FTLs occurs on three dimensions: correctness, performance and durability. We will be assessing the project on correctness and durability.

3.3.1 Correctness

The project framework has the ability to run a trace against the developed FTL. A first correctness check is to ensure that a trace completes successfully if it should (i.e., one that theoretically has to complete given the allowed over-provisioning limit and the erase cycles, and that completes on our version of the FTL). Another correctness test is that, if a test has to fail, it should fail. For example, if we issue a write/read to the last LBA of the raw SSD capacity, it must fail since over-provisioning cannot be 0%. Along with the sample traces provided to you in the code, our testing framework will have a few additional traces (which we will not share) aimed at ensuring correctness of your FTL.

3.3.2 Durability

The aspect of durability is in regards to the number of erases performed for a given workload. Although this is design-dependent, there are a few cases that definitely must not happen. For example, if a trace involves only writing a particular page twice, and it results in an erase, then the FTL is definitely not as per spec. You should be careful while designing your FTL to prevent unnecessary erasures, since they directly reduce the lifetime of the SSD.

3.4 Memory usage

Keep in mind that you are working in the disk firmware. This layer is *always* constrained for resources. We are going to read each submission and also track your memory usage in our framework to ensure that you don't use a *ridiculous* amount of memory. For the first two checkpoints, we do not impose any penalty for a high memory usage (although we will read your code and can penalize if your code has pathological memory usage pattern). For Checkpoint 3, however, your grade would be a function of your memory usage (more on that later).

3.5 Submissions

Submissions for all the checkpoints will be done through Autolab. For checkpoints 1-3, you will only have to edit and submit `myFTL.cpp` to Autolab. Checkpoints 1 and 2 have limited unpenalized submissions (25). For each additional submission, you will be penalized 10% of your checkpoint grade. Please keep in mind that tests for the checkpoints are NOT backward compatible, i.e. if you develop Checkpoint 2, some of the tests for Checkpoint 1 will not work. The submission details for Checkpoint 3 will be mentioned in the handout for Checkpoint 3.

Late submissions: You have a total of **3 grace days for this entire course**. Of the 3 grace days, you can use a maximum of 2 on any single checkpoint. After you have used up your grace days for a given checkpoint, you will be penalized 15% of your checkpoint grade for each late day.

4 Checkpoint 0: Set up development environment

The purpose of this checkpoint is to make sure you have your development environment set up correctly. Follow the directions in Appendix C to get started. If you need an AWS voucher then please request one before finishing checkpoint 0, **but you do not need to receive the voucher before submitting checkpoint 0**.

Once you have requested your voucher (if you need one) and have finished AWS setup, run the script `checkpoint0.sh` **on your AWS instance** and follow the printed directions.

IMPORTANT NOTE: If you requested a voucher, the script will ask you to provide your voucher number. In order to avoid unnecessary stress about this, we will not be using the entered voucher code when scoring your checkpoint 0 submission. Just hit the enter key without inputting anything else.

Once you have finished, the script will output the file `checkpoint0.log`. Submit this file on Autolab.

5 Checkpoint 1: Address translation with logging

In this checkpoint, you will develop an FTL that is more sophisticated than mere one-to-one direct mapping, but one that does not implement garbage collection or wear leveling. You are not expected to implement any optimization for this checkpoint beyond what the following flowcharts explain; in fact, doing so may cause some of the tests to fail.

5.1 Hybrid Log-Block Mapping Scheme [3]

Now that we have understood the fundamentals of SSDs and the challenges they pose, let us dive into an actual mapping scheme, variants of which are used in actual SSDs and a variant of which will be developed by you in this project. In this checkpoint, you will use the overprovisioned blocks for logs (henceforth referred to as log-reservation). While the number for log-reservation blocks is pre-determined by the specified overprovisioning, you are free to place them at any location in the SSD. You must be able to explain why you chose to keep the log-reservation blocks at the position that you chose. As discussed previously, your FTL has to have the intelligence of deriving a PBA from an LBA. In this case, you need to map LBAs one-to-one to the pages of the SSD beginning with package 0, die 0 plane 0, block 0 and page 0 (henceforth denoted as [0, 0, 0, 0, 0]). The mapping should be done only for the usable SSD capacity, i.e. raw capacity minus the overprovisioning. The overprovisioning should be exactly as is specified in the configuration file. Keep in mind that we will be changing the configuration files when grading on Autolab. For

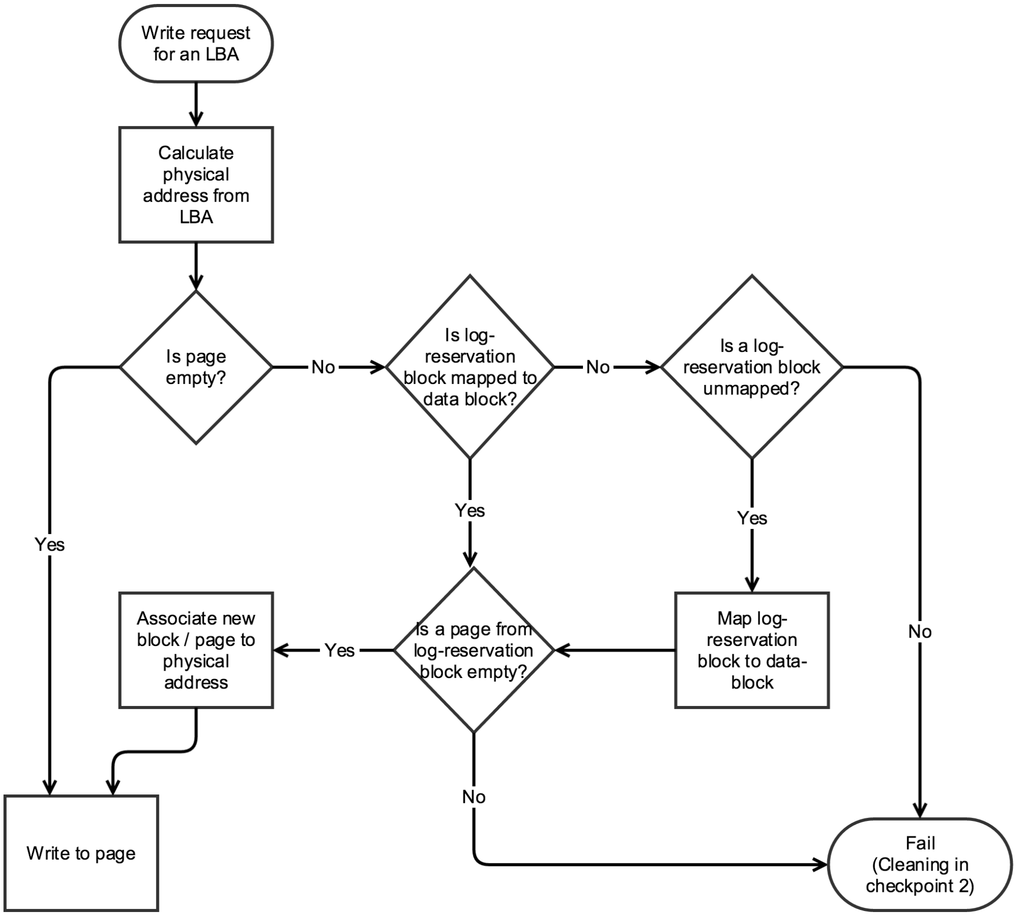

Write request for an LBA
Calculate physical address from LBA
Is page empty?
No
Is log-reservation block mapped to data block?
No
Is a log-reservation block unmapped?
Yes
Yes
Yes
No
Associate new block / page to physical address
Yes
Is a page from log-reservation block empty?
Map log-reservation block to data-block
No
Write to page
Fail (Cleaning in checkpoint 2)

Figure 2: Flowchart for FTL write control flow for checkpoint 1.

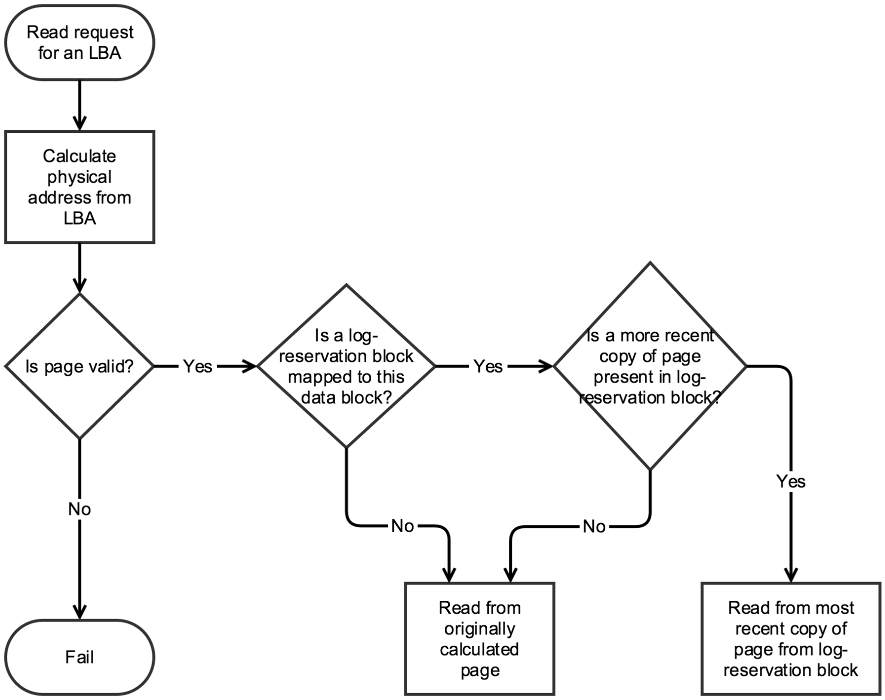

Read request for an LBA
Calculate physical address from LBA
Is page valid?
Yes
Is a log-reservation block mapped to this data block?
Yes
Is a more recent copy of page present in log-reservation block?
No
Yes
No
No
Fail
Read from originally calculated page
Read from most recent copy of page from log-reservation block

Figure 3: Flowcharts for FTL read control flow for Checkpoint 1.

example, if 5% overprovisioning was allowed, in a 100 GB disk, exactly 95 GB should be usable. Address translation can be done differently based on whether you want to issue a read or a write. The following read and write scenarios explain the behavior of the FTL as per the mapping scheme we have selected.

5.2 Write Control Flow

As the flowchart in Figure 2 indicates, if the page of the data block is *empty* the page can be directly written to. This is the simplest case in the write algorithm similar to an uncontended lock in concurrency control. In case it is a *valid* page, your code needs to map a free log-reservation block to this data block (if one is not mapped already). On doing so, you should write to the first empty page you find in the log-reservation block. In case none of the log-reservation blocks are free and if they happen to be mapped to different data blocks, your code must fail. Going ahead, in Checkpoint 2, you will clean log-reservation blocks and use them for completing the requests failing in this checkpoint.

5.3 Read Control Flow

As Figure 3 shows, read is first performed on the page which is calculated from the LBA. If that page is valid, it indicates that there is valid content in the page. This is the most straightforward read that can be performed. In case a page has been written to multiple times before a read is performed, the most recently written data may reside in a page belonging to the log-reservation block mapped to the data block. You will need to identify the most recent copy of the page from the log-reservation block. If the log-reservation block does not contain a copy of the original page, then you need to return the original page that was calculated. In case the original page itself is invalid, that means the page was never written, and you must return an error.

6 Checkpoint 2: Garbage Collection

In this checkpoint, you are going to add garbage collection (a.k.a. segment cleaning) to the code you developed in Checkpoint 1. Recall that in Checkpoint 1, the write control flow had a failure case in it. The failure to update a page was due to the fact that you filled up the log-reservation block mapped to the data block corresponding to the LBA you were about to overwrite, or you used up all the log-reservation blocks and hence you could not issue a page update. Checkpoint 2 allows you to perform garbage collection at such points and reclaim space to proceed with the writes. You are not expected to implement any optimization for this checkpoint beyond what the following flowcharts explain; in fact, doing so may cause some of the tests to fail.

6.1 Algorithm for cleaning

As Figure 4 shows, the process of cleaning is fairly complicated. As mentioned in the handout for Checkpoint 1, the firmware must reserve a few blocks for the purpose of cleaning, which behave somewhat like temporary variables (i.e., they hold the data when the original block is being erased). An alternative to reserving and writing temporary data to an SSD block might be to keep the data in memory until the block erase is complete. But, this is an unsafe way because if we were to encounter power failure during cleaning, we could lose the data we have in memory. As repeated several times in the lectures, data loss is unacceptable for storage systems.

Reiterating a design choice that was mentioned in the Checkpoint 1 handout, the location and number of log-reservation and cleaning-reservation blocks is your choice. It is not a choice to be made randomly. There are trade-offs in the choice of the number and location of these overprovisioned blocks and you are expected to justify your choice in your final report.

Note that in the naive cleaning algorithm specified above, there are three erases required to perform one cycle of cleaning. Erases are significantly slower than reads or writes. Moreover, a block can only be erased a fixed number of times before it is declared corrupt. Hence, cleaning is a crucial component of an SSD firmware and one that directly affects both performance and durability. You may notice that a few optimizations can be done for special cases, one of them, for example, is when only one page is written multiple times, and you have to clean. You can directly erase the data block and write the page from the log-reservation block to the original location and erase the log-reservation block. In this case, you could avoid the work of copying the page from the log-reservation block to a cleaning-reservation block, thus preventing work and also avoid doing an erase. The algorithm described above will

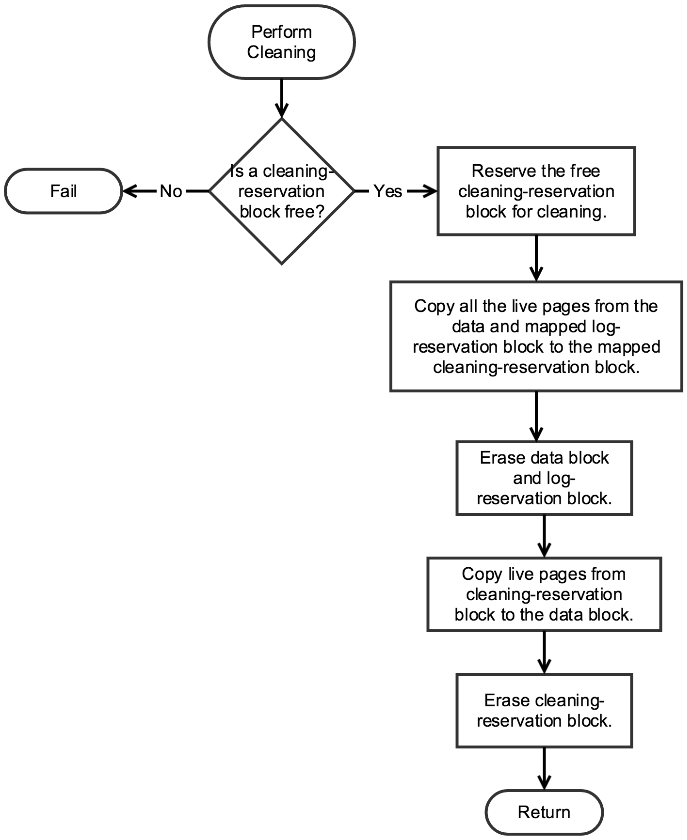

Perform Cleaning
Is a cleaning-reservation block free?
No
Fail
Yes
Reserve the free cleaning-reservation block for cleaning.
Copy all the live pages from the data and mapped log-reservation block to the mapped cleaning-reservation block.
Erase data block and log-reservation block.
Copy live pages from cleaning-reservation block to the data block.
Erase cleaning-reservation block.
Return

Figure 4: Cleaning algorithm.

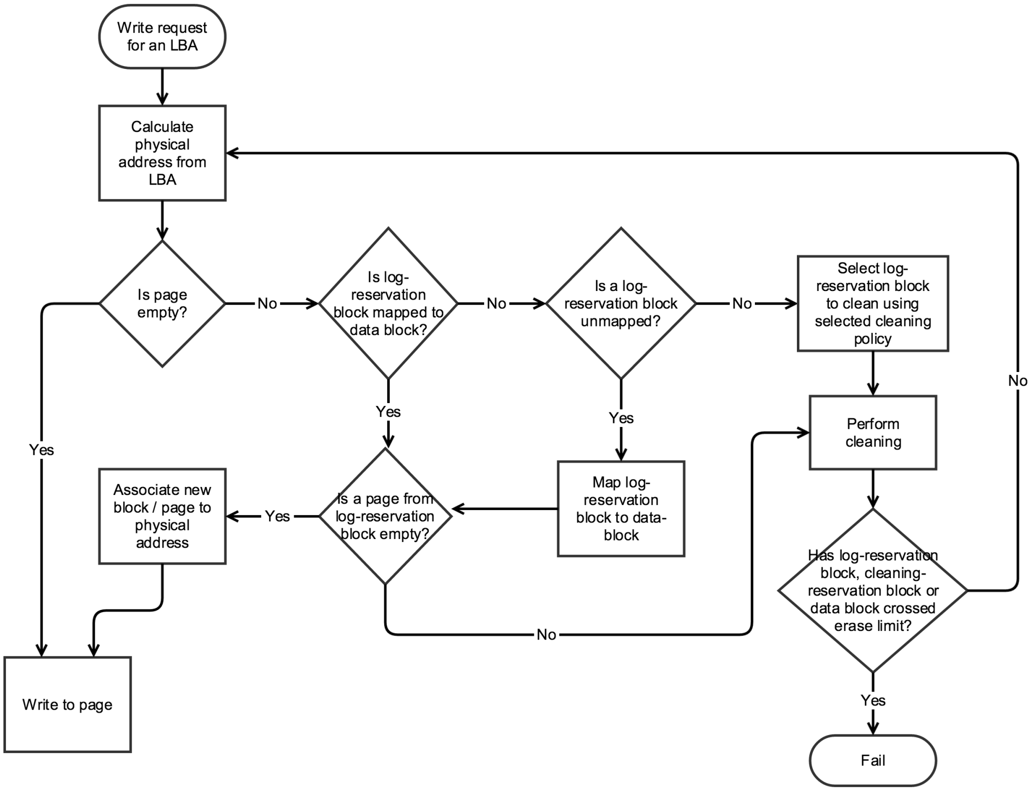

Write request for an LBA
Calculate physical address from LBA
Is page empty?
Yes
No
Is log-reservation block mapped to data block?
No
Yes
Is a log-reservation block unmapped?
No
Yes
Select log-reservation block to clean using selected cleaning policy
Perform cleaning
Has log-reservation block, cleaning-reservation block or data block crossed erase limit?
No
Yes
Fail
Map log-reservation block to data-block
Is a page from log-reservation block empty?
Yes
No
Associate new block / page to physical address
Write to page

Figure 5: Flowchart outlining the decision process for triggering cleaning.

also perform cleaning correctly, but thinking about such optimizations will make your implementation more efficient and may matter when competing for the best *wear score* in Checkpoint 3.

6.2 Policies surrounding garbage collection

There are three decisions governing any garbage collection policy.

1. *When* do we garbage collect?

Traditionally, cleaning can be both on-demand and in the background. Because cleaning a segment takes a substantial amount of time, always doing it on demand can cause some writes to be very slow. So, many systems try to perform garbage collection in the background (ideally, during idle periods) to avoid on-demand cleaning. But, it is not always possible, and it is naturally more complex. You need only support on-demand garbage collection. Figure 5 shows the time at which cleaning is to be invoked in your project.

2. *What* do we garbage collect?

When a particular cleaning operation is initiated, one of the most crucial decisions is the selection of the log-reservation block you choose to clean. A simple FIFO policy may not be the best choice, as you know from having read the log-structured file systems paper. The four policies you need to implement in Checkpoint 2 are:

- **Round Robin:**
  In this policy, the log-reservation block chosen to clean is in a Round Robin (a.k.a. FIFO) manner. This is the simplest cleaning policy indepedent of the workload and recycles log-reservation blocks in the order that they are used.

- **Least Recently Used (LRU) block:**
  As the name suggests, this policy cleans the block that was least recently used (i.e. the block whose latest page was written farthest back in time). Note that a block whose pages are written multiple times need not necessarily have its latest written page in its mapped log-reservation block. This might happen, for example, if the first page was re-written three times and then the second page was written once. This will result in the second page of the data block being the most recently written page, not any of the log-reservation ones.

- **Greedy by Minimum Effort:**
  This policy chooses its target block from the point of view of minimum effort. Effort is decided based on the number of live pages that need to be copied in the multi-fold cleaning process described above. One of the reasons for having fewer valid pages may be TRIMs issued by the file system on file deletion. But, for this project, since we are not dealing with TRIM, it suffices to simply choose the block with least work as the block whose least number of unique pages are valid.

- **LFS Cost-Benefit:**
  This is an implementation of the LFS cost-benefit policy [1]. The age of a block is how long it has been since a page in the block was last modified. That is, the the *age* of a block is the current timestamp minus the timestamp of the most recently written page of block. Note that a timestamp can simply be a monotonically increasing integer value. The utilization is the fraction of the number of valid pages present in the mapped log-reservation block and the data block, i.e.

$$utilization = \frac{(\#\, pages\, to\, be\, copied\, from\, data\, block) + (\#\, pages\, to\, be\, copied\, from\, log\, reservation\, block)}{2 * (\#\, pages\, in\, a\, block)}$$

  The cost-benefit ratio is calculated as follows:

$$\frac{benefit}{cost} = \frac{1\, -\, utilization}{1\, +\, utilization} * age$$

  The policy dictates that the block to be cleaned is the one which has the largest cost-benefit ratio.

3. *How much* should we garbage collect?

This is the policy that dictates when you stop cleaning. If you only clean for the current write to succeed, the very next write might result in a block being cleaned again. Since cleaning is an expensive operation, it may make sense to clean multiple blocks once cleaning has been invoked. Usually the benfits of background cleaning are observed in a multi-threaded environment, where the blocked write can proceed after a short amount of cleaning and the cleaner is a separate thread that interleaves its operations with the foreground workload to prevent future writes from blocking. In this project, you are dealing with a single-threaded SSD emulator and hence the right thing to do is return as soon as possible, to unblock a blocked write. Therefore, the *how much should you clean* question has a trivial answer in this case - one block at a time.

A Appendix: Code structure

A.1 Overview

The code contains the following modules:

- `746FlashSim`: This is the actual simulator for the virtual flash, which mimcs read/write/erase characteristics of (NAND) Flash.
- `myFTL`: This is the the FTL implemented by you.
- `memcheck/memory profiler`: This module keeps track of the memory usage of myFTL.

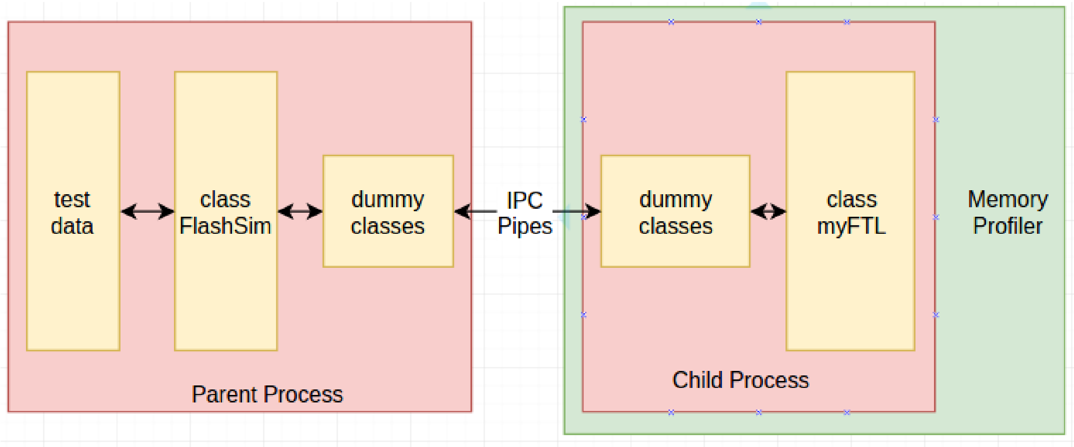


Figure 6: Simulator Architecture.

As shown in Figure 6, `746FlashSim` (parent process) and `myFTL` (child process) modules run as separate processes and communicate via Inter-Process Communication (IPC). This simplifies tracking the memory usage of just the `myFTL` component separately and also ensures that the FTL cannot access any of the test data and hence can't corrupt it.

The tests act as applications sending read/write/erase requests to the Flash simulator which forwards these requests to the FTL. The FTL in turn translates the LBA it receives to physical addresses in the SSD (along with performing other required operations such as garbage collection) and issues read/write/erase requests to the Flash simulator, if required.

Dummy classes exist for helping in implementation of IPC and encapsulating it.

A.2 Layout

The layout of the handout is as follows:

```
handout
    |---- build
    |---- fuse
    |---- iozone
    |---- output
    |---- src
    |---- SSDPlayer_v1.0
    |---- tests
```

- **src:** This directory contains all the main code. The only file that you need to edit and submit to Autolab is `myFTL.cpp`.

- **build:** This is where the binaries are stored after running make.
- **output:** Various logs that are generated during runtime are placed here.
- **tests:** This contains a subset of all the Autolab tests. (Not all tests are released. Some are only on Autolab).
- **fuse, iozone, SSDPlayer_v1.0:** These folders are not directly relevant for the project. You can use SSDPlayer to visualize your FTL and debug it (as described in B.2). Fuse and iozone can be used to run actual workloads, instead of our test cases, on your FTL. We describe each of these later on.

A.3 Classes

This section describes the various classes present in the code.

A.3.1 Classes on the `746FlashSim` side

Class `ConfBase`

This is the base class which deals with the configuration of the simulated Flash. Each test has a corresponding `.conf` file which indicates the configuration of the NAND Flash. An example of such file is the one provided in Section 3.2.

`ConfBase` class has member function who's responsibility is to parse this configuration file. The important member functions of the class are:

- `size_t GetSSDSize(void)` – Returns the count of Packages present in the SSD.
- `size_t GetPackageSize(void)` – Returns the Die count present in a Package.
- `size_t GetDieSize(void)` – Returns the count of Planes present per Die.
- `size_t GetPlaneSize(void)` – Returns the count of Blocks present per Plane.
- `size_t GetBlockSize(void)` – Returns the count of Pages present per Block.
- `size_t GetBlockEraseCount(void)` – Returns the maximum number of times a block can be erased before becoming defunct.
- `size_t GetGCPolicy(void)` - Returns the policy to be used for garbage collection (Useful for Checkpoint 2 only).
- `size_t GetOverprovisioning(void)` – Returns the percentage of Blocks (out of total present Blocks) that are overprovisioned in the SSD.

Note that Page Size is not maintained in the `.conf` file as it is immaterial for the functioning of FTL as the smallest unit of read/write is the Page.

Two classes are derived from the ConfBase class:

1. `Class FlashSimConf` - The Configuration Class on the FlashSim side. This is the class that can be used by Flash Simulator at boot time to gather configuration parameters (e.g. to allocate memory from heap based on these configuration parameters).
2. `Class FTLConf` - The dummy class on the FTL side that is used for IPC handling.

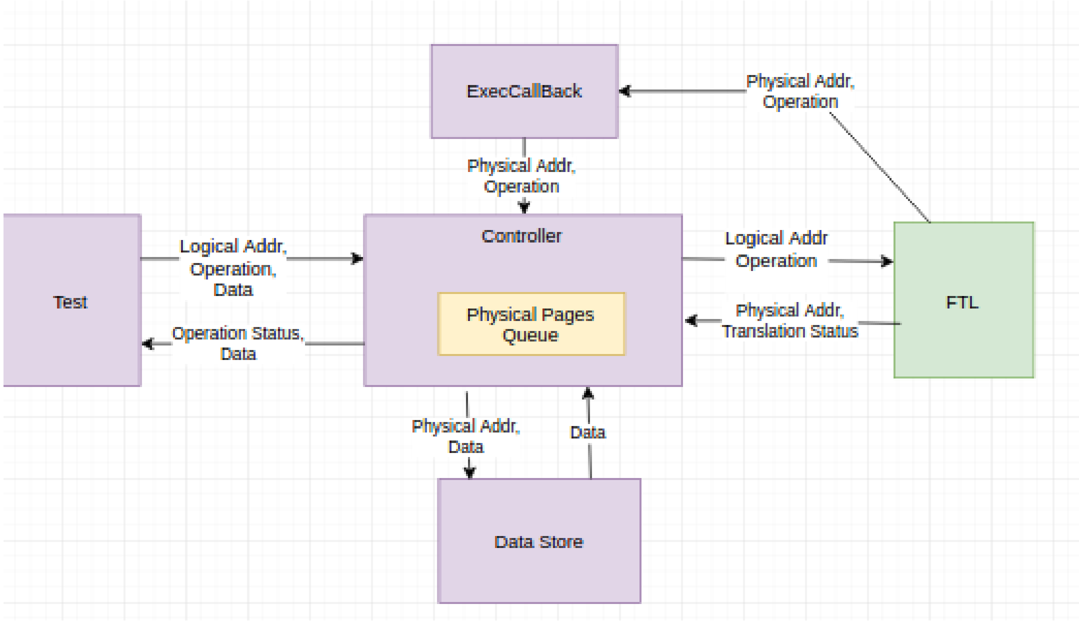


Figure 7: Data Flow in Simulator

You will need to use member functions of class `ConfBase` to gather the configuration parameters.

Class `DataStore`

This is responsible for storing the actual data contained within a Page. This simulates the raw NAND Flash. Since for testing purposes, as an optimization, we don't actually need whole of the data within a Page, the Page stored by DataStore need not be as large as 4KB. DataStore uses the facility of Spare Files in Unix to store the data (A temporary file is created for this purpose).

Class `Controller`

The function of this class is to act as a link between the tests, the FTL class and the Datastore. All requests from the tests (Read/Write/Trim) arrives at the Controller class, which then forwards the logical address of these request to FTL to get them translated to Physical Address and then acts on the Data according to this Physical Address (Read/Write/Erase on the simulated Flash in the DataStore). Remember that the application (tests in this case) knows only about the logical addresses.

Figure 7 shows the movement of Page Data. The test provides the controller with Data and its Logical Address and asks it to execute a functionality (Read/Write/Trim). The controller doesn't know the Physical Address corresponding to Logical Address hence it asks the FTL. Using the Physical Address supplied by the FTL, controller operates on the data residing in the Data store (Stored using the Physical Address).

Lets take an example to understand this flow. Assume that the test (which tries to emulate an application) sends in a request to Write at LBA #5. This request ends up in the Controller along with the Page Data (supplied in a buffer). The controller asks the FTL for a Physical Address of a Physical Page on which to write this data. Suppose the FTL returns the `Address {.package = 1, .die = 2, .plane = 3, .block = 4, .page = 5}`. The controller uses this Physical address and then ask the DataStore to store the Page Data that came along with the Write Request on this Physical Address.The over-provisioning, being a percentage, may result in the number of log blocks not being an integer. As a toy example, 2% over-provisioning with 64 blocks, would mean 1.28 log blocks, which is not possible. In such situations, you should round off the number of blocks to the nearest integer, e.g., 1 log block in this case.

Tests can ask for the following requests: ReadLBA, WriteLBA, TrimLBA. Controller correspondingly asks the FTL for the following services: ReadTranslate, WriteTranslate, Trim (Trim doesn't require an address translation as Trim is supposed to only modify the FTL internal mapping). The controller has following two components:

1. `ExecCallBack` – This class is used by the FTL to request the Controller to manipulate data in DataStore (emulated NAND Flash). E.g to carry out a Write, FTL might request Controller to erase few blocks and move some data between blocks (think garbage collection). ExecCallBack is used as a mechanism to avoid giving FTL direct access to the raw data but still providing it with ability to modify it. ExecCallBack can be used in the following manner:

   `func(OpCode, Address)` – Where func() is the ExecCallBack object of type `ExecCallBack<PageType> &)`, OpCode signify the operation (Read/Write/Erase) and Address is the Physical Address.
   (NOTE: For Erase, Physical address of any Page within the Block works).

2. `Queue` – Queue inside the controller is a method to avoid giving FTL direct access to raw data but still allow it to modify it. A careful reader might have noticed that the FTL doesn't provide Data when it calls ExecCallBack nor ExecCallBack returns data. So how can operation such as Read/Write be carried out by FTL via ExecCallBack. The key idea here is that FTL never needs to know the raw data. It just needs a mechanism to move data around. Suppose FTL wants to copy Page X to Page Y (X and Y are physical addresses) and then erase block containing Page X for the purpose of garbage collection. FTL can do this via the following function calls (func() is an instance of ExecCallBack).

   ```
   func(Opcode::READ, X)
   func(OpCode::WRITE, Y)
   func(OpCode::ERASE, X)
   ```

   Note: Each READ operation moves a page into a LIFO queue's head (push()) and each WRITE moves out a page from the queue's head (pop()). For erase operation, it is necessary that the queue must be empty. (This constraint is enforced because in real-world, erase take relatively long time during which, if there is a power loss, all data in memory will be lost).

Class `FlashSimTest`

This class is responsible for logging, executing test and reporting results.

A.3.2 Classes on the `myFTL` side

Class `FTLBase`

This is the base class from which the FTL class is derived from. This class implements the functionality of FTL. The main member function's of this class are:

1. `std::pair<ExecState, Address> ReadTranslate(size_t lba, const ExecCallBack<PageType> &)`: This function is expected to take the Logical address of a Page (lba) and returns the Physical Address corresponding to that logical address. Return argument ExecState indicates whether the operation was a success or failure (e.g. failure incase read for a page to which no previous writes exists) and ExecCallBack can be used to carry out modification operations on raw data.

2. `std::pair<ExecState, Address> WriteTranslate(size_t, const ExecCallBack<PageType> &)`: Similar to ReadTranslate() except the translation is to satisfy a Write Request sent to the controller.

3. `ExecState Trim(size_t lba, const ExecCallBack<PageType> &)`: When controller receives a TRIM command, controller doesn't carry out any functionality. Rather, it expects the FTL to carry out the operation using ExecCallBack. This function, unlike other two member functions, doesn't do any translation.

A.3.3 Miscellaneous

- `TEST_PAGE_TYPE` – This is a macro which points to the class that acts as Page for the FlashSimulator. Page is the smallest unit for read/write in NAND Flash. As an optimization, the Flash Simulator need not save the whole Page worth of data for simulation so the size of this class can be smaller than the actual Page Size of the NAND. When macro `ENABLE_LARGE_DATASTORE_PAGE` is enabled, Page Size is 4K, else it is 4 bytes.

- `Class Address` – This represents the hierarchial physical address of a Page (which Block, which Plane, which Die etc). Physical Address of Page is different from the Logical Address of the Page.
- `Enum Class ExecState` – This is used to indicate whether an operation was successful or not.

B Appendix: Debugging

In this section, we describe some mechanisms that will make debugging your FTL easier.

B.1 Running as a single process

As explained in Appendix A.1, by default the simulator runs as two processes. This can make debugging difficult. So, for debugging purposes, there exists a configuration to make the code run as a single process. Set `CONFIG_TWOPROC = 0` in the `config.mk` file to make the code run as a single process without requiring any changes to the code of myFTL.

Important: However, note that tests on Autolab will run with this config set, i.e. the tests will run with `CONFIG_TWOPROC = 1`. So while you can disable the config for development/debugging purposes, make sure to test your myFTL with the config enabled before submitting. (Also, if `CONFIG_TWOPROC = 0`, memory checker and memory tracer (more on these later in this document) will be disabled).

Note: During your development, if your program crashes with message like `"Assertion 0 && "Did child died" failed"` or `"Assertion 0 && "Parent process shouldn't have died" failed"`, it means that either the Parent/Child process died. This could happen due to various reasons such as segmentation fault. In this case, you should try to first debug with `CONFIG_TWOPROC = 0`. The bug might not be reproducable with `CONFIG_TWOPROC = 0` as the memory layout changes when you go from two process to one process architecture. Try to manually analyze your code in such scenario.

B.2 Debugging using visualization

It can be hard to visualize the effects of changes in your FTL design. For this purpose, SSDPlayer can be used to visualize internal data movement in the SSD.

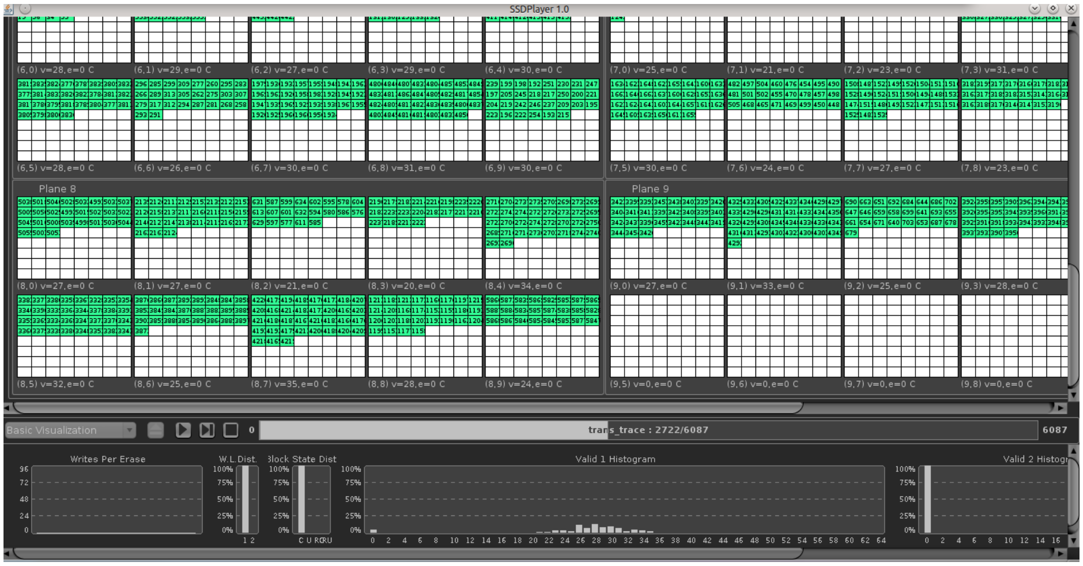


Figure 8: Visualization of Internals

SSDPlayer is an interactive simulator that takes as input a trace file of transactions (read/write/erase) issued to SSD and replay them. Figure 8 is a snapshot during a visualization of a trace in SSDPlayer. The green box indicates the physical pages that are valid and written to. The value inside those boxes indicates the logical address of the pages. White boxes are the physical pages which are not valid (not written to after last erasure of block). The grid is split into Pages/Blocks/Planes (SSDPlayer doesn't recognize Dies and Packages). As the visualization proceeds, one can get a high level understanding of how the FTL is mapping new requests (Only Writes/Erases can be visualized. Reads are not shown). Steps to create the visualization:

- Change the configuration of SSDPlayer (`SSDPlayer_v1.0/ resources/ ssd_config.xml` – Can be opened like a normal text file). Three fields can be customized (Packages and Dies count are not configurable)
    1. Number of Planes – (E.g. <planes>10</planes>)
    2. Number of Blocks per Plane (E.g. <blocks>5</blocks>)
    3. Number of Pages per Block (E.g. <pages>64</pages>)
- Make the same configuration changes to the `test_x_x.cpp` and `test_x_x.conf`. (Planes/Blocks/Pages. Set Packages and Dies to 1).
- Enable transaction tracing (`#define ENABLE_TRANS_TRACING 1`).
- Build and run the test. This will create a trace file (`output/trans_trace.log`).
- Change directory to SSDPlayer (`cd SSDPlayer/`)
- Launch the SSDPLayer (`java -jar SSDPlayer.jar`) (Requires Java).
- Set the manager to 'Basic Visualization' (Default is 'Greedy').
- Load and play the trace file.
- Analyze the trace via visualization.

Note: For Mac user, if you want to run SSDPlayer on AWS instance, you may have to installe XQuartz (`https://www.xquartz.org/`) for your Mac. Or, you can choose to run SSDPlayer locally and copy generated trace file to local disk for analysis.

You can explore more functionality of SSDPlayer by reading more about it (`http://ssdplayer.cswp.cs.technion.ac.il/`). Please let us know if you find any other feature of SSDPlayer useful or you have a better way to analysis the behaviour of FTL.

B.3 Debugging memory consumption

Since FTLs generally don't have large memory, FTL should be optimized for memory consumption also. Your score on checkpoint 3 will depend partially on the peak memory consumped by your FTL. There are chiefly 4 sources of memory consumer: (1) Heap, (2) Stack, (3) Data and BSS. Of these, major consumer of memory is heap. To aid you, we have implemented a small functionality to visualize allocations on heap. Figure 9 is an example of output of malloc tracer.

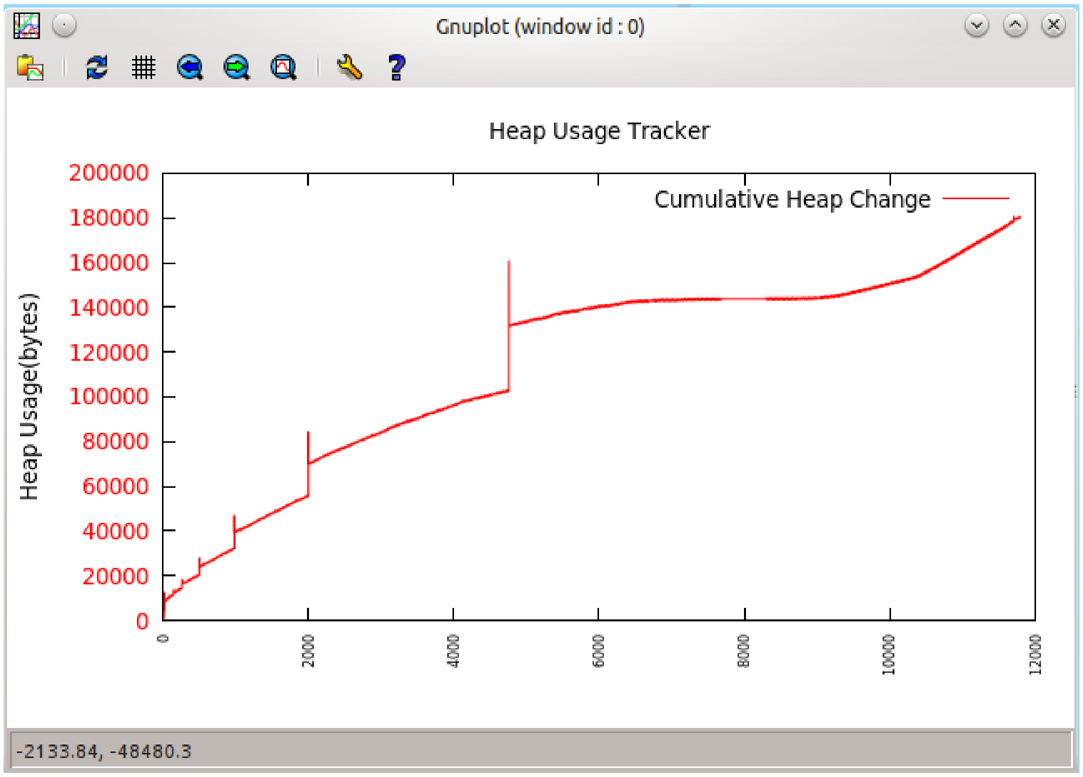


Figure 9: Example of output of malloc tracer

In Figure 9 the x axis denotes the count of call to malloc()/calloc()/free() etc (as time increases) and the y axis denotes the heap in use by the FTL. A couple of things to note:

- Functions such as new/delete internally call malloc()/free().
- The graph shows the memory in use by the FTL on heap and not the total memory allocated to heap which also includes malloc's internal and external fragmentation.

Steps to create the graph:

- Enable malloc tracer in `config.h` (`#define MALLOC_TRACE_ENABLED 1`). `CONFIG_TWOPROC` should be set to 1 in `config.mk`.
- Build and run any test. A file '`output/malloc_trace.dat`' (default name) will be created.
- Run `malloc_tracer.sh` script from within 'output' directory. A couple of things to note:
  - This script uses the `malloc_trace.dat`, so it assumes this file is present in the 'output' directory.
  - This requires gnuplot to work. You might have to install it on your machine.
  - The build might give lots of warnings after malloc tracer is enabled in `config.h`. This is because for malloc tracer to work, we are using some deprecated functions. You can safely ignore these warnings about deprecated functions.
- Visualize the graph (Run '`display malloc_trace_graph.png`').

C Appendix: Setting up and using AWS

We are providing you with an Amazon AWS EC2 image to use as your development environment. This also ensures that you are developing and testing your code in the exact same environment as the one used for Autolab tests, i.e. for marking your submission. This section provides instructions to help you setup and use an AWS account for developing/testing myFTL (as well as the subsequent CloudFS project). These instructions are also available on the course website, under the FAQ section.

C.1 Setup and Billing

1. Create an AWS account at `https://aws.amazon.com/` if you don't have one already.
   Note: If this is the first time you are using an AWS account with CMU funds, you have to use your andrew.cmu.edu email and register for AWS educate. If you have already done these two things for an earlier course, you can use the same account to get coupons for this class too. The only caveat to above is if you are using the andrew.cmu.edu email with 15-619 (Cloud Computing) class, in which case, you can't use the same account; please create a new separate AWS account with a different email and don't register with AWS educate with this new separate account

2. If your account is new, it comes with 750 hours/month (for 12 months) of free instance time. You need to use the "Free-Tier" instances in order to take advantage of that promotion.

3. If you need AWS credits for the course, visit voucher request link and enter your Andrew ID. We will be emailing you a $50 voucher. You can start using AWS without the voucher, since Amazon will not bill you until the end of the month.

4. To monitor your billing and usage, sign in to your AWS account, click on the drop-down menu with your account name and click on "My Billing Dashboard".

C.2 Starting your AWS instance

1. Sign into your AWS account.

2. Click on EC2 under "Compute Services" to be taken to the EC2 Dashboard.

3. On the right corner of the top bar, click the second drop-down menu from the right to select the datacenter you will be using. Select "N. Virginia".

4. In the sidebar on the left, click on "AMIs" under the "Images" group.

5. Click on the drop-down menu under "Launch" and select "Public images".

6. Use the bar to the right of the drop-down menu to search for '746-update10-devCopy'. Double-check that the image owner is '169965024155'. This is the VM image for the course. Use the refresh button at the top right if the image doesn't show up right away.

7. Select the AMI, and click "Launch".

8. Make sure the "t2.micro" instance type is selected (which is also Free-tier eligible, if you are using a new account that benefits from that promotion).

9. Click "Review and Launch".

10. On the next screen click "Launch".

11. In the first drop-down menu select "Proceed without a key pair", and check the checkbox. We have provided you with a key to access the instance, named '746-student.pem'. You should be able to find it on the course website under the FAQ section.

    - Remember to run "`chmod 400 746-student.pem`" to ensure your key pair file carries the right permissions

12. Click "Launch Instances", and on the next screen click "View Instances"

13. In the EC2 Dashboard Instances screen wait for your VM instance state to transition to "running", and the VM status checks to indicate "2/2 checks passed".

14. Select your instance and check the 'Public DNS' column. Make a note of the machine's FQDN, which will be of the form `X.compute-Y.amazonaws.com`

15. From your terminal, run

```
ssh -Y -i path/to/746-student.pem student@X.compute-Y.amazonaws.com
```

- Make sure the path passed to the -i option points to the 746-student.pem key pair you downloaded from the course website

16. Run the ssh command to connect to your instance!

C.3 Securing your AWS instance

It is a good idea to change the key you use to login to your AWS instance, and stop using the 746-student.pem which the AWS template comes configured with. To do so, we provide a simple script `make_secret_key.sh` on the course webpage, which will generate a new key and use it to replace `746-student.pem` on your instance. Note that if you terminate your instance and create it again from the template image, you will have to run this script again.

C.4 Terminating your AWS instance

When you are done using your VM, you can stop it or terminate it. You can stop your VM by running the shutdown command from within Ubuntu, or using Actions → Instance State → Stop.

Be warned that when your VM is stopped it still consumes resources, which will lead to usage charges for EBS storage (and you may run out of your credit faster). Instead of stopping your VM, make sure to copy out your source code when you're done, and then terminate your instance. You can do that via Actions → Instance State → Terminate.

D Optional "fun": Trying realistic workloads on your myFTL

So you got to the end of Checkpoint 2. Light is starting to creep through the blinds. Something inside you tells you to stop. Turn it in and get it over with. But you know you are not content with the tests those pesky TAs gave you. You can do better. You are ready to try the real thing. Maybe it's the sleep deprivation, maybe it's the adrenaline rush. Fear not, we have just the right fix for you. Real workloads is where it's at.

This is NOT required for your project.

No, really.

Like, no bonus points. Just sweat. And tears.

...Suit yourself.

You can run real workloads against your FTL using Filesystem in User SpacE (FUSE). FUSE allows trapping file system calls in userspace and acting on them. Using FUSE, we can trap the read/write requests that an application makes to a file and redirect those to our simulator. Hence, we can replace the tests in our simulator with FUSE with an actual file being stored in our simulated NAND Flash.

Note: For this to work, you need to install FUSE. See online materials on how to install FUSE for your linux distribution if you are using your local machine. To test if you have FUSE already installed, you can run ‘which fusermount’, If this doesn’t print out a path of fusermount bin, then FUSE is not installed. FUSE should be installed in your AWS image.

Steps to run real applications against your FTL in the simulated NAND Flash:

- Enable large page (`#define ENABLE_LARGE_DATASTORE_PAGE 1` in `config.h`)
- Run ’`make fuse`’ from your handout’s top directory.
- Run ’`$PWD/output/myFuse -c <conf-file> -f <file> -m <mount directory> \`
  `-s <reference directory> -l <log-file>`’ from top directory.
  - `<conf-file>` - .conf file that indicates the configuration of Flash (similar to the test’s conf file)
  - `<file>` - The temporary file for which the FUSE redirects read/write system calls to FlashSim.
  - `<mount directory>` - The directory where to mount FUSE filesystem. This is the directory from where the <file> should be accessed. (This folder should be empty).
  - `<ref directory>` - The reference directory for FUSE. This is the directory where the original <file> is kept at.
  - `<log-file>` - The log file.
    (Note: All paths should be absolute)

Consider an example,
`$PWD/output/myFuse -c $PWD/fuse/ref/config.conf -f $PWD/fuse/ref/text.txt \`
`-m $PWD/fuse/mount -s $PWD/fuse/ref -l $PWD/output/fuse.log`.
For this example, we can use a text editor to edit a text file stored in emulated Flash. The original file (`text.txt` in this example) should be kept at `<ref directory>`, then myFuse should be ran and then the text editor should be used to modify the file in `<mount directory>` (`$PWD/fuse/mount/text.txt` in this example). After myFuse starts running, it copies the `$PWD/fuse/ref/text.txt` contents to the FlashSim’s emulated NAND Flash. After that, all the read/write requests that the text editor or any other application makes to the `$PWD/fuse/mount/text.txt` goes to the FlashSim and not to the `$PWD/fuse/ref/text.txt`.

Note: With `ENABLE_LARGE_DATASTORE_PAGE=1`, none of the tests will compile. Set this macro true only for using FUSE.

Contact the course staff if you require assistance. Or not. Contact Saksham, it was his idea.

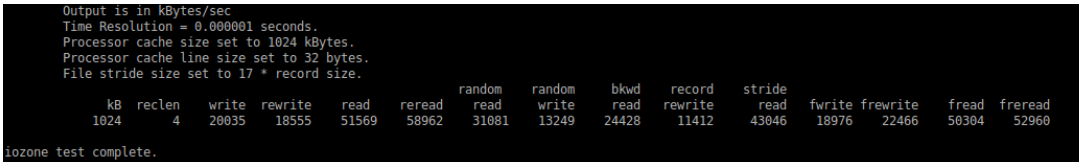
D.1 Throughput of your FTL

We also added the functionality to test the throughput of your FTL using iozone, an IO benchmarking tool, and FUSE. (Note: You may use this functionality during checkpoint 2 or checkpoint 3, when your FTL supports garbage collection).

Steps to run a sample performace test:

- Enable large page (`#define ENABLE_LARGE_DATASTORE_PAGE 1` in `config.h`)
- Edit `fuse/ref/config.h` as per your needs. (SSD should atleast support 4MB and should have sufficient block erasaure limit and overprovisioning).
- Run ‘`make perftest`’ from your handout’s top directory. (Note: You might see warnings during build. These warning are due to compilation of iozone’s code base).

Incase of some errors such as “`Transport endpoint is not connected`”, try running ‘`make veryclean`’.



```
        Output is in kBytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 kBytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                               random   random     bkwd   record   stride
              kB  reclen    write  rewrite     read   reread     read    write     read  rewrite     read   fwrite frewrite    fread  freread
            1024       4    20035    18555    51569    58962    31081    13249    24428    11412    43046    18976    22466    50304    52960

iozone test complete.
```

Figure 10: Sample output of pertest

To understand more about the iozone and meaning of it’s output, run ‘`./iozone/src/current/iozone -h`’. Note: The throughput shown might be less than actual due to overhead of FUSE. You might want to use the throughput as a relative measure between your different FTL designs. The throughput might also vary between different runs.

Let us know if you find some other interesting application to test your FTL using FUSE. For example, we were able to run a small video over the FlashSim. (To do this, run `myFuse` with the video file as the input file (this file should be present inside ’ref’ directory) and then run the pseduo video file in the mount directory using a video player of your choice. The video will be running over the FTL).

References

[1] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[2] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL’09. First International Conference on*, pages 125–131. IEEE, 2009.

[3] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5):332–343, 2009.