15-746 Project 1 Handout: myFTL (revised for checkpoint 2)

September 14, 2016

Contents

1 Solid State Disk (SSD)

Solid state disks (SSDs) are storage devices that use flash chips to store information rather than the spinning magnetic disks used in hard drives. As a result, SSDs have a radically different architecture from hard drives.

1.1 Architecture

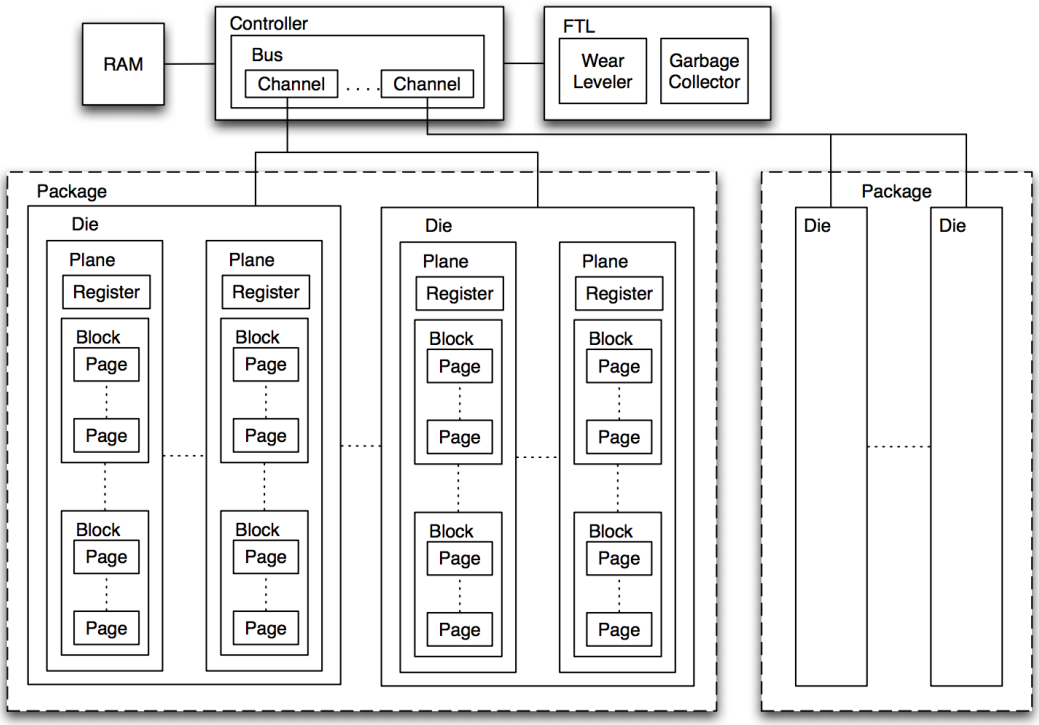

Figure 1: Example SSD Architecture

As illustrated in the above figure, an SSD has a multi-tiered architecture. The example SSD has multiple packages, each with multiple dies, containing multiple planes, which in turn have multiple blocks and finally each block has a fixed number of pages. An important note regarding the terms *block* and *page*.[1] Block sizes of SSDs are typically 128 KB, and blocks are not the finest granularity of data access. Pages, typically 4 KB, are the smallest unit of data storage on an SSD.

The file system and any other layer outside the SSD is oblivious to the packages, dies, planes, blocks and pages of the SSD. Upper layer programs only communicate with the SSD is in terms of the logical block address (LBA). In the context of this project, each LBA refers to 4 KB of data, which will be the same as the size of an SSD page. The flash translation layer (FTL) is responsible for the translation of the LBA to a physical block address (PBA) where the data resides / needs to be written. The raw capacity of an SSD is calculated by:

$$packages\_per\_ssd \times dies\_per\_package \times planes\_per\_die \times blocks\_per\_plane \times pages\_per\_block \times bytes\_per\_page$$

An important characteristic of SSDs is that the granularity of erases is different from the granularity of reads / writes. This is unlike traditional HDDs, where a sector is the unit of reads and writes and there are no erases. In SSDs, erases can only happen at the block level, while reads and writes can take place at the page level. Since a block is comprised of multiple pages, an erase of a block destroys data in all pages contained in that block. Moreover,

[1] In computer systems, these terms can mean multiple things depending on context. Here, we are talking about common in-SSD terminology, rather than file systems, virtual memory systems, etc. A file system block can vary across file systems anywhere from 4 KB to 64 MB (in the case of the Google File System) A block from an HDD's perspective can be different. Traditionally sector sizes in HDDs were 512 bytes while blocks were 4 KB. Now-a-days even sectors are 4 KB. Blocks are usually the granularity at which reads and writes take place in HDDs. The SSD page is different from a main memory page, which also happens to be 4 KB in size. In this project, when we refer to block, we mean an SSD block and by page, we mean an SSD page.

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. Unless your logic is sound, the SSD emulator will reject invalid reads or writes. A thing to note is that in this project, we assume that reading a page that was never written results in 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.

A fundamental difference between SSDs and HDDs is that HDDs can theoretically accomodate infinite sector rewrites. SSDs can only safely do 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 millions), 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 to 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 princliples as log-structured file systems can be used to deal with SSDs effectively.

SSDs effectively can service multiple requests in parallel, rather than one-at-a-time like traditional HDDs that have only one active disk head 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 (FTL)

In this project, you will build a flash translation layer (FTL) for an emulated solid state drive (SSD). As you know, SSDs too have firmware in the drive that performs multiple functions. When SSDs were first introduced, their radically different architecture required a completely different way of reading or writing to them. The flash translation layer is an abstraction introduced to maintain current file system and driver code as it is. The FTL translates the commands issued by the file systems 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 physical address from an LBA similar to virtual to physical address translation in the context of virtual memory. This logic is a crucial component of this project.

2.2 Garbage Collection (Segment Cleaning)

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 wherein we perform a series of writes and deletes to independent files not overwriting any file at all until we fill up the SSD. At this point, the SSD utilization (i.e., percentage of the SSD that is full) 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 or segment cleaning. This action is very similar to the segment cleaning performed in log-structured file systems, where log segments are analogous to blocks in the SSD context. You will need to do segment cleaning as a part of this project (in checkpoint 2), and the details of that part will be discussed in the handout for 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 to the LBAs to inform it that the LBAs used for that file do not contain valid data any more. Why do this? The reason - garbage collection. Let us consider a

situation where garbage collection has been invoked by the SSD firmware because no block appears to be free. In this situation, without the help of the file system, how can the SSD differentiate between which pages are live (i.e., they contain valid data) and which are dead (i.e., they contain invalid data, possibly data of the file that was deleted, and hence shouldn't be preseved during segment cleaning)? TRIM is simply a convenience method aimed at reducing the work the garbage collector has to perform during segment cleaning. It is a common misunderstanding that SSDs *require* TRIM. In this project, among other things, you could (if you wish to) prove that an SSD is perfectly capable of performing all operations without supporting TRIM, i.e. you can simply ignore TRIM and immediately return SUCCESS. However, proper handling of TRIM will make myFTL more efficient, which can be an important factor of your score in checkpoint 3.

2.3 Wear Leveling

Wear leveling can be thought of as being analogous to load balancing in a distributed system. Since SSD blocks have a limited number of erases before it is declared as corrupt, we need to ensure that we do not end up over-erasing a few blocks while the others are erased much less often. In a journaling file system, typically the journal is located at a fixed area on disk. The journal is written to 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. Another example of hot spots are metadata blocks of a file system viz. the block bitmap or the inode bitmap. Since FTLs enable SSDs to be drop-in replacements for HDDs, it is unreasonable to expect file system code to change in order to take SSD wear leveling into account. 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 add wear leveling as the final icing on your FTL cake.

2.4 Overprovisioning

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 overprovisioning. Overprovisioning can be for several reasons, but typically and for the purpose of this project, it is for the following two reasons:

2.4.1 Overprovisioning for segment cleaning

In segment cleaning, 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 segment cleaning.

2.4.2 Overprovisioning for 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 must complete successfully before the SSD fails due to a particular block being erased more than it is allowed to).

3 SSD Emulator and Evaluation

Ideally, the best experience writing an FTL would be on an actual SSD's firmware. But, we do not have programmable SSDs available. Moreover, your code (like all other code since the history of programming) will have bugs with probability 1, and debugging a firmware environment is much more difficult.

So, the next best thing to an actual SSD is an emulated SSD. This particular emulation uses software, viz. 746FlashSim, built in-house at CMU (and inspired by FlashSim from Kim et al. [2], researchers at Penn State University (PSU)). It is an object-oriented approach to emulating SSDs written in C++. Fortunately / unfortunately, you too will develop the project in C++. This project does not require prior knowledge of object-oriented programming or C++. The functions you need to write are clearly mentioned, and you can write C inside those functions. You also are not expected to use / learn object-oriented programming principles viz. polymorphism, inheritance and the like. In case you happen to know the standard C++ data structures viz. maps, sets, and so on, your life may become easier. Learning them in case you don't know them already *is not a big deal*. 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 *out there* 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 of the MyFTL class already created in *myFTL.cpp*. These methods are currently stub functions (i.e. empty functions) that you must develop. You must not change any other file other than *myFTL.cpp*. You may add as many functions to this file as you deem fit.

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

std::pair<FlashSim::ExecState, FlashSim::Address>
  WriteTranslate(size_t lba, const FlashSim::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 names suggest, these methods translate 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 address translations by yourself to ensure correctness of your logic. **The FTL logic without segment cleaning or wear leveling is the objective of checkpoint 1**.
These methods take an LBA and an instance of *FlashSim::ExecCallBack*, and return whether the operation was a SUCCESS or a FAILURE, and the translated Physical Address (PBA). The provided instance of *FlashSim::ExecCallBack* can be used to issue reads, writes, and erases while performing an address translation (for eg. while performing garbage cleaning 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 PBA 0 (contained in block 0) followed by a write to PBA 32 (contained in block 1), followed by an erasure of block 0. You would do so in the following way:

```
... some code here ...
func(FlashSim::OpCode::READ, FlashSim::Address(0, 0, 0, 0, 0))
func(FlashSim::OpCode::WRITE, FlashSim::Address(0, 0, 0, 1, 0))
/* For erasing a block, just pass in any PBA in that block */
func(FlashSim::OpCode::ERASE, FlashSim::Address(0, 0, 0, 0, 0))
... some more 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 test_1_1

./test_1_1 tests/checkpoint_1/test_1_1/test_1_1.conf /tmp/test_1_1.log
```

The tests issue events (read / write) to the MyFTL object. Your log file can reside in the path of your choice. 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;
  FlashSimTest 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");
  return ret;
}
```

Configuration file 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
SSD_SIZE 4

# Number of Dies per Package
PACKAGE_SIZE 8
```

```
# Number of Planes per Die
DIE_SIZE 2

# Number of Blocks per Plane
PLANE_SIZE 64

# Number of Pages per Block
BLOCK_SIZE 16

# Number of erases in lifetime of block
BLOCK_ERASES 500

# Overprovisioning (in %)
OVERPROVISIONING 5

# Selected garbage collection policy
SELECTED_GC_POLICY 0
```

You should try playing around with the knobs, especially overprovisioning 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.

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 overprovisioning 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 overprovisioning 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 dependant, 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 Submissions

Submissions for all the checkpoints will be done through 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.

4 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.

4.1 Hybrid Log-Block Mapping Scheme [1]

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). You must be able to explain why you chose to keep the log-reservation blocks in the way that you choose. 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 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.

4.2 Write Control Flow

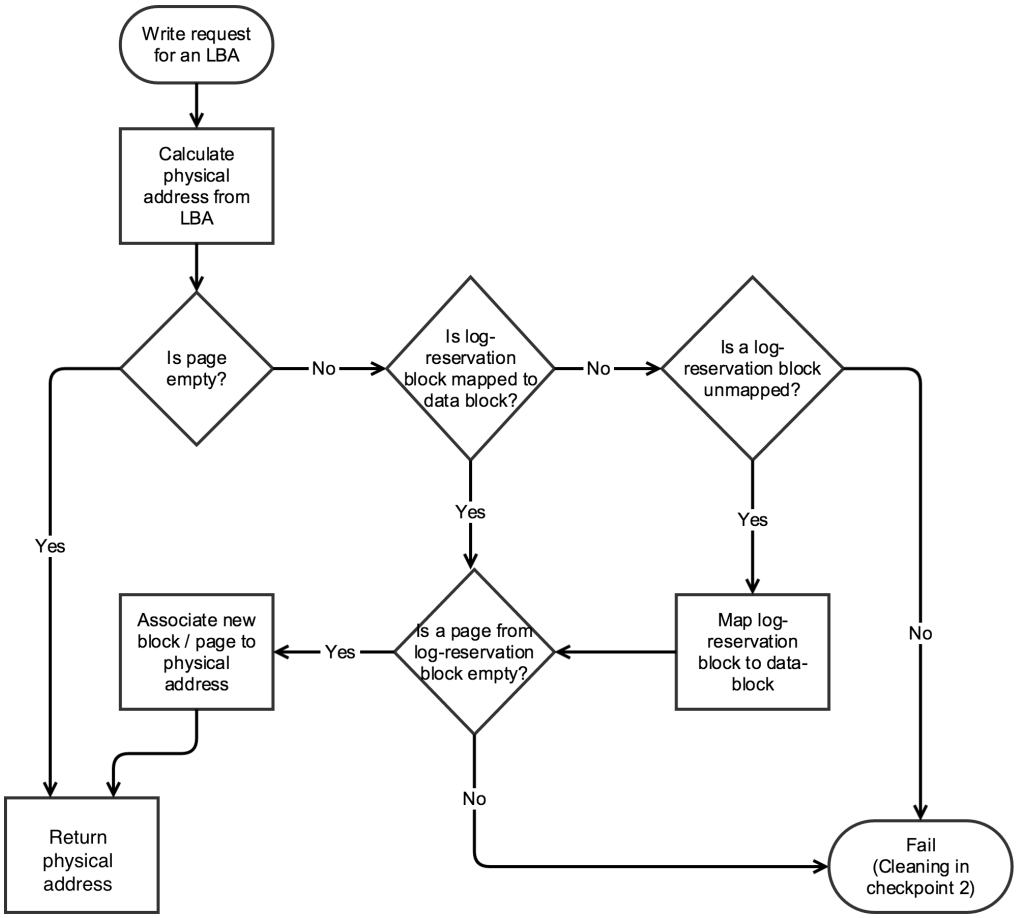

Figure 2: Flowchart for FTL write control flow for checkpoint 1

As the flowchart in 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.

4.3 Read Control Flow

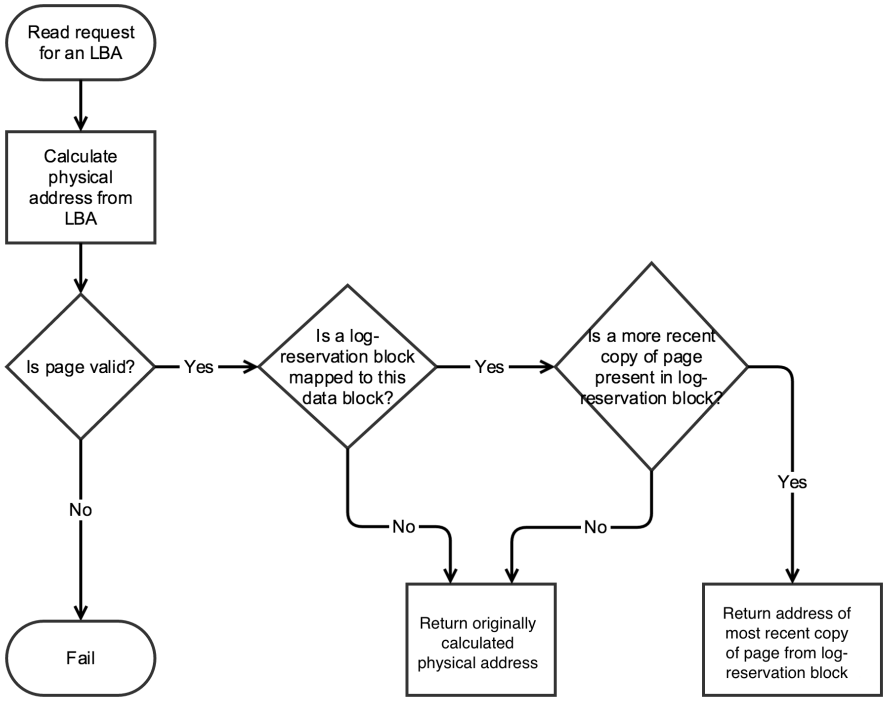

Figure 3: Flowcharts for FTL read control flow for checkpoint 1

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.

4.4 Memory usage

Keep in mind that you are working in the disk firmware. This layer is *always* constrained for resources. So, it is not acceptable for you to use a huge map (a.k.a. hash-table) to map every LBA to its corresponding PBA. A significant percentage of the grade is dependent on the fact that you don't use unnecessary memory. We are going to read each submission (and track your memory usage in our framework) in order to verify this. It is thus crucial that you spend some thought into minimizing the amount or RAM usage and explain it thoroughly in your report.

5 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.

5.1 Algorithm for cleaning

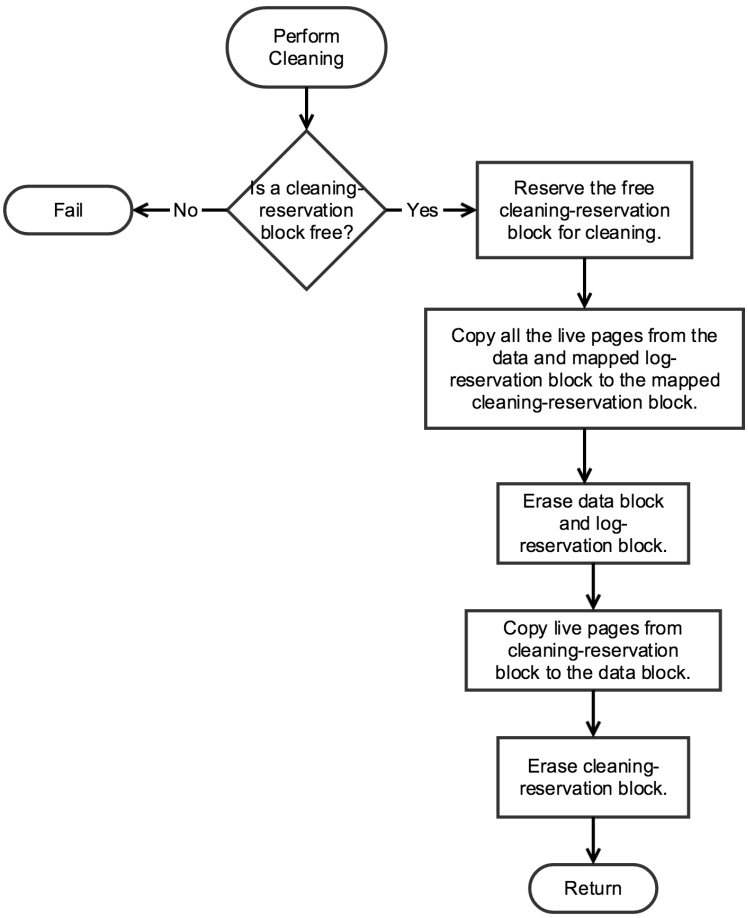

Figure 4: Algorithm for performing cleaning.

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 tradeoffs 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 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.

5.2 Policies surrounding garbage collection

There are three main policies governing cleaning:

5.2.1 When should you perform cleaning?

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.

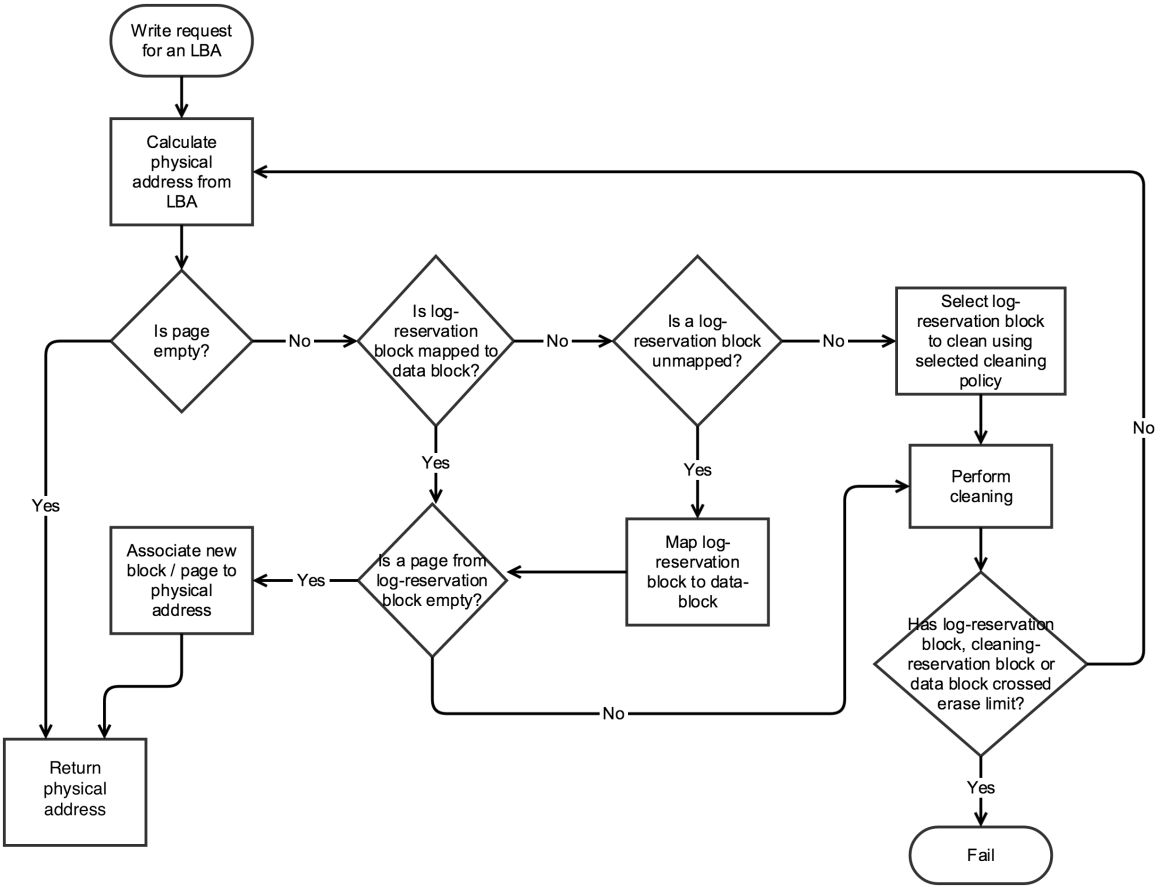

Figure 5: Flowchart to understand when to clean.

5.2.2 What should you clean?

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). Consequently, the *age* of a block is the timestamp of the most recently written page of that block. Note that a timestamp can simply be a monotonically increasing integer value. Also, the mechanism to understand the LRU block is a little involved. 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. 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 from Rosenblum et. al. [3]. The age of a block is determined as explained in the LRU cleaning policy. 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.

5.2.3 How much should you clean?

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.

5.3 TRIM

As discussed previously, TRIM is used by the layers above the SSD to inform the FTL that certain LBAs do not contain live data. In checkpoint 2, along with READs and WRITEs, our test suite will issue TRIM commands. Your FTL is free to use this information to guide itself in optimizing garbage collection. For checkpoint 2, you are not required to fully implement TRIM, but your FTL must reply with SUCCESS for the tests to pass.

6 Checkpoint 3 - Wear Leveling

Handout for checkpoint 3 will be released independently. An announcement will be made once it is posted.

References

[1] Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. A survey of flash translation layer. *Journal of Systems Architecture 55*, 5 (2009), 332–343.

[2] Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B. Flashsim: A simulator for nand flash-based solid-state drives. In *Advances in System Simulation, 2009. SIMUL'09. First International Conference on* (2009), IEEE, pp. 125–131.

[3] Rosenblum, M., and Ousterhout, J. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (1991), ACM.