41

Locality and The Fast File System

When the UNIX operating system was first introduced, the UNIX wizard himself Ken Thompson wrote the first file system. Let's call that the "old UNIX file system", and it was really simple. Basically, its data structures looked like this on the disk:

The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

The good thing about the old file system was that it was simple, and supported the basic abstractions the file system was trying to deliver: files and the directory hierarchy. This easy-to-use system was a real step forward from the clumsy, record-based storage systems of the past, and the directory hierarchy was a true advance over simpler, one-level hierarchies provided by earlier systems.

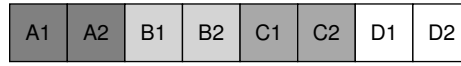
41.1 The Problem: Poor Performance

The problem: performance was terrible. As measured by Kirk McKusick and his colleagues at Berkeley [MJLF84], performance started off bad and got worse over time, to the point where the file system was delivering only 2% of overall disk bandwidth!

The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs. For example, the data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks of a file (a pretty common operation).

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:

As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. A side note: this problem is exactly what disk **defragmentation** tools help with; they reorganize on-disk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. Thus, the problem:

THE CRUX:
HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE

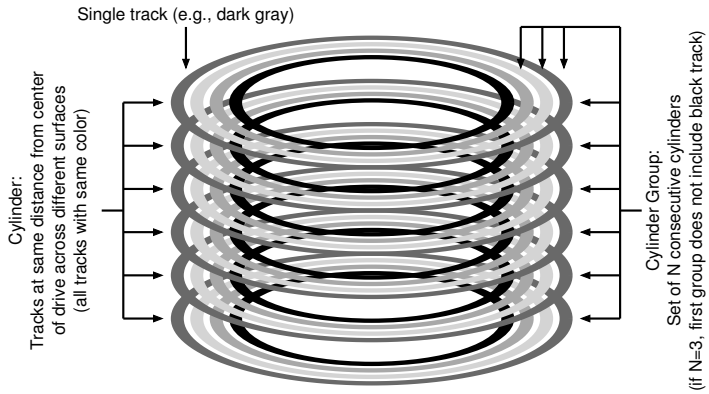
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

41.2 FFS: Disk Awareness Is The Solution

A group at Berkeley decided to build a better, faster file system, which they cleverly called the **Fast File System (FFS)**. The idea was to design the file system structures and allocation policies to be "disk aware" and thus improve performance, which is exactly what they did. FFS thus ushered in a new era of file system research; by keeping the same *interface* to the file system (the same APIs, including `open()`, `read()`, `write()`, `close()`, and other file system calls) but changing the internal *implementation*, the authors paved the path for new file system construction, work that continues today. Virtually all modern file systems adhere to the existing interface (and thus preserve compatibility with applications) while changing their internals for performance, reliability, or other reasons.

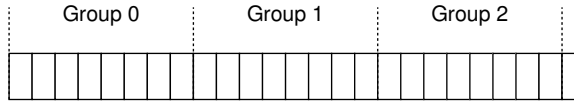
41.3 Organizing Structure: The Cylinder Group

The first step was to change the on-disk structures. FFS divides the disk into a number of **cylinder groups**. A single **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive; it is called a cylinder because of its clear resemblance to the so-called geometrical shape. FFS aggregates each N consecutive cylinders into group, and thus the entire disk can thus be viewed as a collection of cylinder groups. Here is a simple example, showing the four outer most tracks of a drive with six platters, and a cylinder group that consists of three cylinders:



Note that modern drives do not export enough information for the file system to truly understand whether a particular cylinder is in use; as discussed previously [AD14a], disks export a logical address space of blocks and hide details of their geometry from clients. Thus, modern file

systems (such as Linux ext2, ext3, and ext4) instead organize the drive into **block groups**, each of which is just a consecutive portion of the disk's address space. The picture below illustrates an example where every 8 blocks are organized into a different block group (note that real groups would consist of many more blocks):



Whether you call them cylinder groups or block groups, these groups are the central mechanism that FFS uses to improve performance. Critically, by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

To use these groups to store files and directories, FFS needs to have the ability to place files and directories into a group, and track all necessary information about them therein. To do so, FFS includes all the structures you might expect a file system to have within each group, e.g., space for inodes, data blocks, and some structures to track whether each of those are allocated or free. Here is a depiction of what FFS keeps within a single cylinder group:

Let's now examine the components of this single cylinder group in more detail. FFS keeps a copy of the **super block** (S) in each group for reliability reasons. The super block is needed to mount the file system; by keeping multiple copies, if one copy becomes corrupt, you can still mount and access the file system by using a working replica.

Within each group, FFS needs to track whether the inodes and data blocks of the group are allocated. A per-group **inode bitmap** (ib) and **data bitmap** (db) serve this role for inodes and data blocks in each group. Bitmaps are an excellent way to manage free space in a file system because it is easy to find a large chunk of free space and allocate it to a file, perhaps avoiding some of the fragmentation problems of the free list in the old file system.

Finally, the **inode** and **data block** regions are just like those in the previous very-simple file system (VSFS). Most of each cylinder group, as usual, is comprised of data blocks.

ASIDE: **FFS FILE CREATION**
As an example, think about what data structures must be updated when a file is created; assume, for this example, that the user creates a new file `/foo/bar.txt` and that the file is one block long (4KB). The file is new, and thus needs a new inode; thus, both the inode bitmap and the newly-allocated inode will be written to disk. The file also has data in it and thus it too must be allocated; the data bitmap and a data block will thus (eventually) be written to disk. Hence, at least four writes to the current cylinder group will take place (recall that these writes may be buffered in memory for a while before they take place). But this is not all! In particular, when creating a new file, you must also place the file in the file-system hierarchy, i.e., the directory must be updated. Specifically, the parent directory `foo` must be updated to add the entry for `bar.txt`; this update may fit in an existing data block of `foo` or require a new block to be allocated (with associated data bitmap). The inode of `foo` must also be updated, both to reflect the new length of the directory as well as to update time fields (such as last-modified-time). Overall, it is a lot of work just to create a new file! Perhaps next time you do so, you should be more thankful, or at least surprised that it all works so well.

41.4 Policies: How To Allocate Files and Directories

With this group structure in place, FFS now has to decide how to place files and directories and associated metadata on disk to improve performance. The basic mantra is simple: *keep related stuff together* (and its corollary, *keep unrelated stuff far apart*).

Thus, to obey the mantra, FFS has to decide what is "related" and place it within the same block group; conversely, unrelated items should be placed into different block groups. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. FFS employs a simple approach: find the cylinder group with a low number of allocated directories (to balance directories across groups) and a high number of free inodes (to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group. Of course, other heuristics could be used here (e.g., taking into account the number of free data blocks).

For files, FFS does two things. First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data (as in the old file system). Second, it places all files that are in the same directory in the cylinder group of the directory they are in. Thus, if a user creates four files, `/a/b`, `/a/c`, `/a/d`, and `b/f`, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

Let's look at an example of such an allocation. In the example, assume that there are only 10 inodes and 10 data blocks in each group (both

unrealistically small numbers), and that the three directories (the root directory `/`, `/a`, and `/b`) and four files (`/a/c`, `/a/d`, `/a/e`, `/b/f`) are placed within them per the FFS policies. Assume the regular files are each two blocks in size, and that the directories have just a single block of data. For this figure, we use the obvious symbols for each file or directory (i.e., `/` for the root directory, `a` for `/a`, `f` for `/b/f`, and so forth).

```
group inodes     data
    0 /--------- /---------
    1 acde------ accddee---
    2 bf-------- bff-------
    3 ---------- ----------
    4 ---------- ----------
    5 ---------- ----------
    6 ---------- ----------
    7 ---------- ----------
    ...
```

Note that the FFS policy does two positive things: the data blocks of each file are near each file's inode, and files in the same directory are near one another (namely, `/a/c`, `/a/d`, and `/a/e` are all in Group 1, and directory `/b` and its file `/b/f` are near one another in Group 2).

In contrast, let's now look at an inode allocation policy that simply spreads inodes across groups, trying to ensure that no group's inode table fills up quickly. The final allocation might thus look something like this:

```
group inodes     data
    0 /--------- /---------
    1 a--------- a---------
    2 b--------- b---------
    3 c--------- cc--------
    4 d--------- dd--------
    5 e--------- ee--------
    6 f--------- ff--------
    7 ---------- ----------
    ...
```

As you can see from the figure, while this policy does indeed keep file (and directory) data near its respective inode, files within a directory are arbitrarily spread around the disk, and thus name-based locality is not preserved. Access to files `/a/c`, `/a/d`, and `/a/e` now spans three groups instead of one as per the FFS approach.

The FFS policy heuristics are not based on extensive studies of file-system traffic or anything particularly nuanced; rather, they are based on good old-fashioned **common sense** (isn't that what CS stands for after all?)[1]. Files in a directory *are* often accessed together: imagine compiling a bunch of files and then linking them into a single executable. Because such namespace-based locality exists, FFS will often improve performance, making sure that seeks between related files are nice and short.

[1]Some people refer to common sense as **horse sense**, especially people who work regularly with horses. However, we have a feeling that this idiom may be lost as the "mechanized horse", a.k.a. the car, gains in popularity. What will they invent next? A flying machine??!!

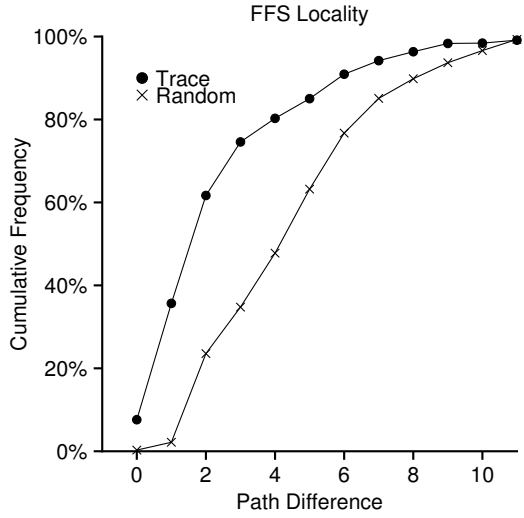


Figure 41.1: **FFS Locality For SEER Traces**

41.5 Measuring File Locality

To understand better whether these heuristics make sense, let's analyze some traces of file system access and see if indeed there is namespace locality. For some reason, there doesn't seem to be a good study of this topic in the literature.

Specifically, we'll use the SEER traces [K94] and analyze how "far away" file accesses were from one another in the directory tree. For example, if file `f` is opened, and then re-opened next in the trace (before any other files are opened), the distance between these two opens in the directory tree is zero (as they are the same file). If a file `f` in directory `dir` (i.e., `dir/f`) is opened, and followed by an open of file `g` in the same directory (i.e., `dir/g`), the distance between the two file accesses is one, as they share the same directory but are not the same file. Our distance metric, in other words, measures how far up the directory tree you have to travel to find the *common ancestor* of two files; the closer they are in the tree, the lower the metric.

Figure 41.1 shows the locality observed in the SEER traces over all workstations in the SEER cluster over the entirety of all traces. The graph plots the difference metric along the x-axis, and shows the cumulative percentage of file opens that were of that difference along the y-axis. Specifically, for the SEER traces (marked "Trace" in the graph), you can see that about 7% of file accesses were to the file that was opened previously, and that nearly 40% of file accesses were to either the same file or to one in the same directory (i.e., a difference of zero or one). Thus, the FFS locality assumption seems to make sense (at least for these traces).

Interestingly, another 25% or so of file accesses were to files that had a distance of two. This type of locality occurs when the user has structured a set of related directories in a multi-level fashion and consistently jumps between them. For example, if a user has a `src` directory and builds object files (`.o` files) into an `obj` directory, and both of these directories are sub-directories of a main `proj` directory, a common access pattern will be `proj/src/foo.c` followed by `proj/obj/foo.o`. The distance between these two accesses is two, as `proj` is the common ancestor. FFS does *not* capture this type of locality in its policies, and thus more seeking will occur between such accesses.

For comparison, the graph also shows locality for a "Random" trace. The random trace was generated by selecting files from within an existing SEER trace in random order, and calculating the distance metric between these randomly-ordered accesses. As you can see, there is less namespace locality in the random traces, as expected. However, because eventually every file shares a common ancestor (e.g., the root), there is some locality, and thus random is useful as a comparison point.

41.6 The Large-File Exception

In FFS, there is one important exception to the general policy of file placement, and it arises for large files. Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). Filling a block group in this manner is undesirable, as it prevents subsequent "related" files from being placed within this block group, and thus may hurt file-access locality.

Thus, for large files, FFS does the following. After some number of blocks are allocated into the first block group (e.g., 12 blocks, or the number of direct pointers available within an inode), FFS places the next "large" chunk of the file (e.g., those pointed to by the first indirect block) in another block group (perhaps chosen for its low utilization). Then, the next chunk of the file is placed in yet another different block group, and so on.

Let's look at some diagrams to understand this policy better. Without the large-file exception, a single large file would place all of its blocks into one part of the disk. We investigate a small example of a file (`/a`) with 30 blocks in an FFS configured with 10 inodes and 40 data blocks per group. Here is the depiction of FFS without the large-file exception:

```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
    ...
```

As you can see in the picture, `/a` fills up most of the data blocks in Group 0, whereas other groups remain empty. If some other files are now created in the root directory (`/`), there is not much room for their data in the group.

With the large-file exception (here set to five blocks in each chunk), FFS instead spreads the file spread across groups, and the resulting utilization within any one group is not too high:

```
group inodes     data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ---------- ---------- ----------
    5 ---------- aaaaa----- ---------- ---------- ----------
    6 ---------- ---------- ---------- ---------- ----------
    ...
```

The astute reader (that's you) will note that spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 29 in order). And you are right, oh astute reader of ours! But you can address this problem by choosing chunk size carefully.

Specifically, if the chunk size is large enough, the file system will spend most of its time transferring data from disk and just a (relatively) little time seeking between chunks of the block. This process of reducing an overhead by doing more work per overhead paid is called **amortization** and is a common technique in computer systems.

Let's do an example: assume that the average positioning time (i.e., seek and rotation) for a disk is 10 ms. Assume further that the disk transfers data at 40 MB/s. If your goal was to spend half our time seeking between chunks and half our time transferring data (and thus achieve 50% of peak disk performance), you would thus need to spend 10 ms transferring data for every 10 ms positioning. So the question becomes: how big does a chunk have to be in order to spend 10 ms in transfer? Easy, just use our old friend, math, in particular the dimensional analysis mentioned in the chapter on disks [AD14a]:

$$\frac{40\ \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1\ \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms} = 409.6\ KB \tag{41.1}$$

Basically, what this equation says is this: if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek in order to spend half your time seeking and half your time transferring. Similarly, you can compute the size of the chunk you would need to achieve 90% of peak bandwidth (turns out it is about 3.69MB), or even 99% of peak bandwidth (40.6MB!). As you can see, the closer you want to get to peak, the bigger these chunks get (see Figure 41.2 for a plot of these values).

FFS did not use this type of calculation in order to spread large files across groups, however. Instead, it took a simple approach, based on the structure of the inode itself. The first twelve direct blocks were placed in the same group as the inode; each subsequent indirect block, and all the blocks it pointed to, was placed in a different group. With a block size of 4KB, and 32-bit disk addresses, this strategy implies that every

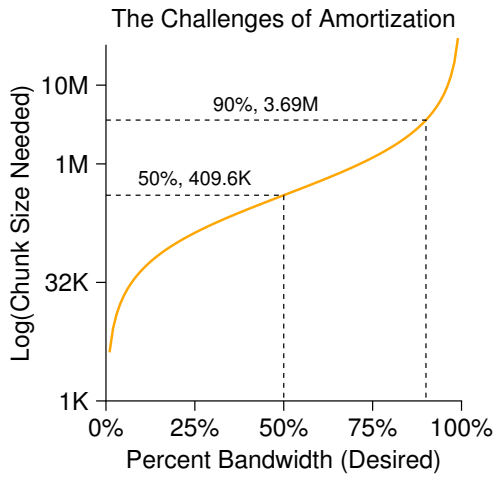


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

1024 blocks of the file (4MB) were placed in separate groups, the lone exception being the first 48KB of the file as pointed to by direct pointers.

Note that the trend in disk drives is that transfer rate improves fairly rapidly, as disk manufacturers are good at cramming more bits into the same surface, but the mechanical aspects of drives related to seeks (disk arm speed and the rate of rotation) improve rather slowly [P98]. The implication is that over time, mechanical costs become relatively more expensive, and thus, to amortize said costs, you have to transfer more data between seeks.

41.7 A Few Other Things About FFS

FFS introduced a few other innovations too. In particular, the designers were extremely worried about accommodating small files; as it turned out, many files were 2KB or so in size back then, and using 4KB blocks, while good for transferring data, was not so good for space efficiency. This **internal fragmentation** could thus lead to roughly half the disk being wasted for a typical file system.

The solution the FFS designers hit upon was simple and solved the problem. They decided to introduce **sub-blocks**, which were 512-byte little blocks that the file system could allocate to files. Thus, if you created a small file (say 1KB in size), it would occupy two sub-blocks and thus not waste an entire 4KB block. As the file grew, the file system will continue allocating 512-byte blocks to it until it acquires a full 4KB of data. At that point, FFS will find a 4KB block, *copy* the sub-blocks into it, and free the sub-blocks for future use.

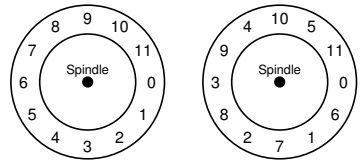


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

You might observe that this process is inefficient, requiring a lot of extra work for the file system (in particular, a lot of extra I/O to perform the copy). And you'd be right again! Thus, FFS generally avoided this pessimal behavior by modifying the `libc` library; the library would buffer writes and then issue them in 4KB chunks to the file system, thus avoiding the sub-block specialization entirely in most cases.

A second neat thing that FFS introduced was a disk layout that was optimized for performance. In those times (before SCSI and other more modern device interfaces), disks were much less sophisticated and required the host CPU to control their operation in a more hands-on way. A problem arose in FFS when a file was placed on consecutive sectors of the disk, as on the left in Figure 41.3.

In particular, the problem arose during sequential reads. FFS would first issue a read to block 0; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.

FFS solved this problem with a different layout, as you can see on the right in Figure 41.3. By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk *how many* blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

You might be thinking: this scheme isn't so great after all. In fact, you will only get 50% of peak bandwidth with this type of layout, because you have to go around each track twice just to read each block once. Fortunately, modern disks are much smarter: they internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason). Then, on subsequent reads to the track, the disk will just return the desired data from its cache. File systems thus no longer have to worry about these incredibly low-level details. Abstraction and higher-level interfaces can be a good thing, when designed properly.

Some other usability improvements were added as well. FFS was one of the first file systems to allow for **long file names**, thus enabling more expressive names in the file system instead of the traditional fixed-size approach (e.g., 8 characters). Further, a new concept was introduced

TIP: MAKE THE SYSTEM USABLE
Probably the most basic lesson from FFS is that not only did it introduce the conceptually good idea of disk-aware layout, but it also added a number of features that simply made the system more usable. Long file names, symbolic links, and a rename operation that worked atomically all improved the utility of a system; while hard to write a research paper about (imagine trying to read a 14-pager about "The Symbolic Link: Hard Link's Long Lost Cousin"), such small features made FFS more useful and thus likely increased its chances for adoption. Making a system usable is often as or more important than its deep technical innovations.

called a **symbolic link**. As discussed in a previous chapter [AD14b] , hard links are limited in that they both could not point to directories (for fear of introducing loops in the file system hierarchy) and that they can only point to files within the same volume (i.e., the inode number must still be meaningful). Symbolic links allow the user to create an "alias" to any other file or directory on a system and thus are much more flexible. FFS also introduced an atomic `rename()` operation for renaming files. Usability improvements, beyond the basic technology, also likely gained FFS a stronger user base.

41.8 Summary

The introduction of FFS was a watershed moment in file system history, as it made clear that the problem of file management was one of the most interesting issues within an operating system, and showed how one might begin to deal with that most important of devices, the hard disk. Since that time, hundreds of new file systems have developed, but still today many file systems take cues from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). Certainly all modern systems account for the main lesson of FFS: treat the disk like it's a disk.

References

[AD14a] "Operating Systems: Three Easy Pieces"
Chapter: Hard Disk Drives
Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau
There is no way you should be reading about FFS without having first understood hard drives in some detail. If you try to do so, please instead go directly to jail; do not pass go, and, critically, do not collect 200 much-needed simoleons.

[AD14b] "Operating Systems: Three Easy Pieces"
Chapter: File System Implementation
Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau
As above, it makes little sense to read this chapter unless you have read (and understood) the chapter on file system implementation. Otherwise, we'll be throwing around terms like "inode" and "indirect block" and you'll be like "huh?" and that is no fun for either of us.

[K94] "The Design of the SEER Predictive Caching System"
G. H. Kuenning
MOBICOMM '94, Santa Cruz, California, December 1994
According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems, 2:3, pages 181-197.
August, 1984. *McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.*

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
Keynote Lecture at the ACM SIGMOD Conference (SIGMOD '98)
June, 1998
A great and simple overview of disk technology trends and how they change over time.

Homework

This section introduces `ffs.py`, a simple FFS simulator you can use to understand better how FFS-based file and directory allocation work. See the README for details on how to run the simulator.

Questions

1. Examine the file `in.largefile`, and then run the simulator with flag `-f in.largefile` and `-L 4`. The latter sets the large file exception to 4 blocks in a group before moving on to the next one. What do you think the file system allocation will look like? Then run with `-c` enabled to see the actual layout.
2. Now run with `-L 30`. What do you expect to see? Once again, turn on `-c` to see if you were right. You can also use `-S` to see exactly which blocks were allocated to the file `/a`.
3. Now we will compute some statistics about the file. The first is something we call *filespan*, which is the max distance between any two data blocks of the file or between the inode and any data block. the file and the data block that is furthest away from it. Calculate the filespan of `/a`. Run `ffs.py -f in.largefile -L 4 -T -c` to see what it is. Do the same with `-L 100`. What difference do you expect in filespan as the large-file exception parameter changes from low values to high values?
4. Now let's look at a new input file, `in.manyfiles`. How do you think the FFS policy will lay these files out across groups? (you can run with `-v` to see what files and directories are created, or just `cat in.manyfiles`). Run the simulator with `-c` to see if you were right.
5. A new metric we will use to evaluate FFS is called *dirspan*. This metric calculates the spread of files within a particular directory, specifically the max distance between the inodes and data blocks of all the files in the directory as well as the inode and data block of the directory itself. Run with `in.manyfiles` and the `-T` flag, and see if you can figure out the dirspan of the three directories. Run with `-c` to see if you were right. How good of a job does FFS do in minimizing dirspan?
6. Now change the size of the inode table per group to 5 (`-I 5`). How do you think this will change the layout of the files? Run with `-c` to see if you were right. How does it affect the dirspan?
7. One policy that can affect FFS effectiveness is which group to place the inode of a new directory in. The default policy (in the simulator) simply looks for the group with the most free inodes. A slightly different policy, specified with `-A`, looks for a group of groups with the most free inodes. For example, if you run with `-A 2`, when allocating a new directory, the simulator will look at groups in pairs and pick the best pair for the allocation. Now you should run

`./ffs.py -f in.manyfiles -I 5 -A 2 -c` to see how allocation changes with this strategy. How does it affect dirspan? Why might this policy be a good idea?

8. One last policy change we will explore relates to file fragmentation. Run `./ffs.py -f in.fragmented -v` and see if you can predict how the files that remain are allocated. Run with `-c` to confirm your answer. What is interesting about the data layout of file `/i`? Why is it problematic?
9. A new policy, which we call *contiguous allocation* and enabled with the `-C` flag, tries to ensure that each file is allocated contiguously. Specifically, with `-C n`, the file system tries to ensure that `n` contiguous blocks are free within a group before allocating a block. Run `./ffs.py -f in.fragmented -v -C 2 -c` to see the difference in layout. How does layout change as the parameter passed to `-C` increases? Finally, how does `-C` affect filespan and dirspan?

42

Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, A and B, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

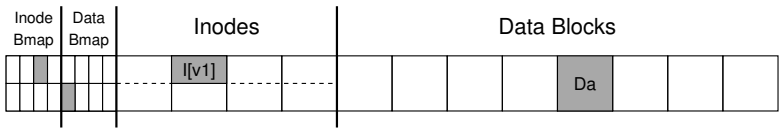
The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

42.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:



If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted I[v1], as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of I[v1], we see:

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

In this simplified inode, the `size` of the file is `1` (it has one block allocated), the first direct pointer points to block `4` (the first data block of the file, Da), and all three other direct pointers are set to `null` (indicating

that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

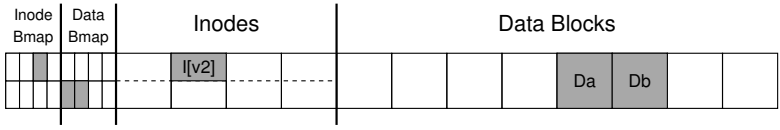
When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block as well as have a bigger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music perhaps?

What we would like is for the final on-disk image of the file system to look like this:



To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk. Unfortunately, a crash may occur and thus interfere with these updates to the disk. In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency[1].

- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

  Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.

- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.

- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

[1] However, it might be a problem for the user, who just lost some data!

The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another **atomically** (e.g., after the inode, bitmap, and new data block have been written to disk). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: **fsck**[2]. `fsck` is a UNIX tool for finding such inconsistencies and repairing them [M86]; similar tools to check and repair a disk partition exist on different systems. Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

The tool `fsck` operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (`fsck` assumes that no other file-system activity is on-going while it runs); once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Here is a basic summary of what `fsck` does:

- **Superblock:** `fsck` first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, `fsck` scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.

[2] Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `fsck` makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by `fsck`; the inode bitmap is correspondingly updated.
- **Inode links:** `fsck` also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, `fsck` scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the `lost+found` directory.
- **Duplicates:** `fsck` also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered "bad" if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can't do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `fsck` performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

As you can see, building a working `fsck` requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging [G+08]. However, fsck (and similar approaches) have a bigger and perhaps more fundamental problem: they are *too slow*. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours. Performance of `fsck`, as disks grew in capacity and RAIDs grew in popularity, became prohibitive (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAIDs) grew, researchers and practitioners started to look for other solutions.

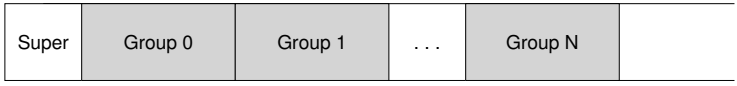
42.3 Solution #2: Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

The basic idea is as follows. When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the "write ahead" part, and we write it to a structure that we organize as a "log"; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group has an inode and data bitmap as well as inodes and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:



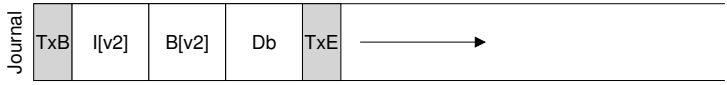
Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

The real difference is just the presence of the journal, and of course, how it is used.

Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), as well as some kind of **transaction identifier** (**TID**). The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block Db to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Thus, to **checkpoint** the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes I[v2], B[v2], and Db to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the file system and are basically done. Thus, our initial sequence of operations:

1. **Journal write:** Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
2. **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

In our example, we would write TxB, I[v2], B[v2], Db, and TxE to the journal first. When these writes complete, we would complete the update by checkpointing I[v2], B[v2], and Db, to their final locations on disk.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue

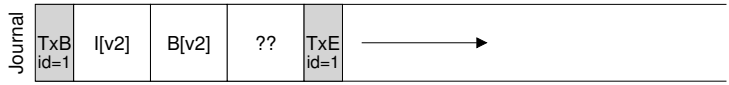
ASIDE: **FORCING WRITES TO DISK**

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, A and B, was easy: just issue the write of A to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of B.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:



Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

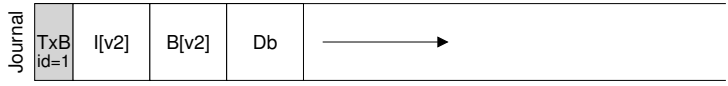
ASIDE: **OPTIMIZING LOG WRITES**

You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).

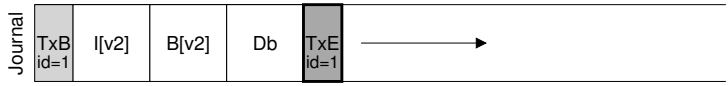
One of our former graduate students, Vijayan Prabhakaran, had a simple idea to fix this problem [P+05]. When writing a transaction to the journal, include a checksum of the contents of the journal in the begin and end blocks. Doing so enables the file system to write the entire transaction at once, without incurring a wait; if, during recovery, the file system sees a mismatch in the computed checksum versus the stored checksum in the transaction, it can conclude that a crash occurred during the write of the transaction and thus discard the file-system update. Thus, with a small tweak in the write protocol and recovery system, a file system can achieve faster common-case performance; on top of that, the system is slightly more reliable, as any reads from the journal are now protected by a checksum.

This simple fix was attractive enough to gain the notice of Linux file system developers, who then incorporated it into the next generation Linux file system, called (you guessed it!) **Linux ext4**. It now ships on millions of machines worldwide, including the Android handheld platform. Thus, every time you write to disk on many Linux-based systems, a little code developed at Wisconsin makes your system a little faster and more reliable.

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):



When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:



An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte

write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block. Thus, our current protocol to update the file system, with each of its three phases labeled:

1. **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be **committed**.
3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.

Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped. If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover** the update as follows. When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus **replayed** (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

Note that it is fine for a crash to happen at any point during checkpointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about[3].

Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocated a new inode), the newly-created inode of the file, the

[3]Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

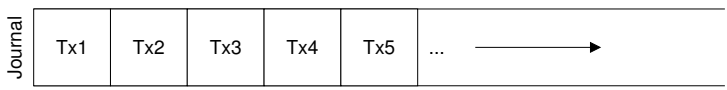
data block of the parent directory containing the new directory entry, as well as the parent directory inode (which now has a new modification time). With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and assuming they even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

Making The Log Finite

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

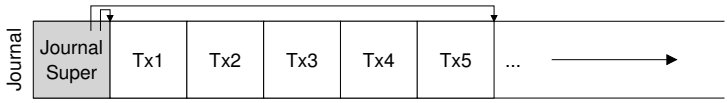
However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?



Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as a **circular log**. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused. There are many ways to achieve this end; for example, you could simply mark the

oldest and newest non-checkpointed transactions in the log in a **journal superblock**; all other space is free. Here is a graphical depiction:



In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

Thus we have our final data journaling protocol. But there is still a problem: we are writing each data block to the disk *twice*, which is a heavy cost to pay, especially for something as rare as a system crash. Can you figure out a way to retain consistency without writing data twice?

Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:

The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of "write the pointed to object before the object with the pointer to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use some form of metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to issue data writes as well as the transaction-begin block and metadata to the journal; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

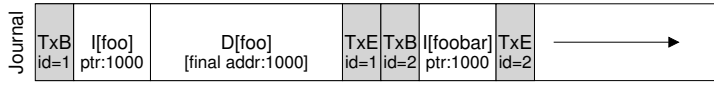
Tricky Case: Block Reuse

There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

> "What's the hideous part of the entire system? ... It's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated." [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are *not* journaled). Let's say you have a directory called `foo`. The user adds an entry to `foo` (say by creating a file), and thus the contents of `foo` (because directories are considered metadata) are written to the log; assume the location of the `foo` directory data is block 1000. The log thus contains something like this:

At this point, the user deletes everything in the directory as well as the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say `foobar`), which ends up reusing the same block (1000) that used to belong to `foo`. The inode of `foobar` is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of `foobar` is committed to the journal; the newly-written data in block 1000 in the file `foobar` is *not* journaled.



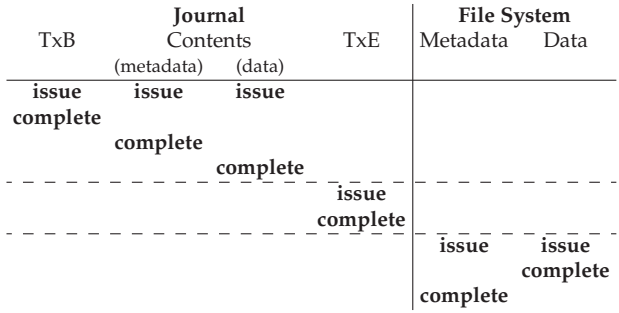


Figure 42.1: **Data Journaling Timeline**

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `foobar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `foobar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data as well as metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note that the data write can logically be issued at the same time as the writes

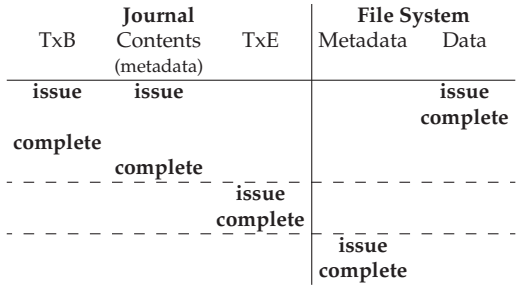


Figure 42.2: **Metadata Journaling Timeline**

to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on `fsck`, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Patt. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk *before* the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Another approach is known as **copy-on-write** (yes, **COW**), and is used in a number of popular file systems, including Sun's ZFS [B07]. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures. Doing so makes keeping the file system consistent straightforward. We'll be learning more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible and uses a generalized form of the **transaction checksum** [P+05], as well as a few other techniques, to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log), thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata as well as user data.

References

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.

[C+12] "Consistency Without Ordering"
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '12, San Jose, California
A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!

[C+13] "Optimistic Crash Consistency"
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '13, Nemacolin Woodlands Resort, PA, November 2013
Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` a lot, performance can be greatly improved.

[GP94] "Metadata Update Performance in File Systems"
Gregory R. Ganger and Yale N. Patt
OSDI '94
A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.

[G+08] "SQCK: A Declarative File System Checker"
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
OSDI '08, San Diego, California
Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit"
Robert Hagmann
SOSP '87, Austin, Texas, November 1987
The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.

[M+13] "ffsck: The Fast File System Checker"
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.

[MK96] "Fsck - The UNIX File System Check Program"
Marshall Kirk McKusick and T. J. Kowalski
Revised in 1996
Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM Transactions on Computing Systems.
August 1984, Volume 2:3
You already know enough about FFS, right? But yeah, it is OK to reference papers like this more than once in a book.

[P+05] "IRON File Systems"
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.

[PAA05] "Analysis and Evolution of Journaling File Systems"
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
USENIX '05, Anaheim, California, April 2005
An early paper we wrote analyzing how journaling file systems work.

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling"
Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
DSN '11, Hong Kong, China, June 2011
Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.

[T98] "Journaling the Linux ext2fs File System"
Stephen C. Tweedie
The Fourth Annual Linux Expo, May 1998
Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.

[T00] "EXT3, Journaling Filesystem"
Stephen Tweedie
Talk at the Ottawa Linux Symposium, July 2000
olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
A transcript of a talk given by Tweedie on ext3.

[T01] "The Linux ext2 File System"
Theodore Ts'o, June, 2001.
Available: http://e2fsprogs.sourceforge.net/ext2.html
A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.

43

Log-structured File Systems

In the early 90's, a group at Berkeley led by Professor John Ousterhout and graduate student Mendel Rosenblum developed a new file system known as the log-structured file system [RO91]. Their motivation to do so was based on the following observations:

- **System memories are growing**: As memory gets bigger, more data can be cached in memory. As more data is cached, disk traffic increasingly consists of writes, as reads are serviced by the cache. Thus, file system performance is largely determined by its write performance.
- **There is a large gap between random I/O performance and sequential I/O performance**: Hard-drive transfer bandwidth has increased a great deal over the years [P98]; as more bits are packed into the surface of a drive, the bandwidth when accessing said bits increases. Seek and rotational delay costs, however, have decreased slowly; it is challenging to make cheap and small motors spin the platters faster or move the disk arm more quickly. Thus, if you are able to use disks in a sequential manner, you gain a sizeable performance advantage over approaches that cause seeks and rotations.
- **Existing file systems perform poorly on many common workloads**: For example, FFS [MJLF84] would perform a large number of writes to create a new file of size one block: one for a new inode, one to update the inode bitmap, one to the directory data block that the file is in, one to the directory inode to update it, one to the new data block that is a part of the new file, and one to the data bitmap to mark the data block as allocated. Thus, although FFS places all of these blocks within the same block group, FFS incurs many short seeks and subsequent rotational delays and thus performance falls far short of peak sequential bandwidth.
- **File systems are not RAID-aware**: For example, both RAID-4 and RAID-5 have the **small-write problem** where a logical write to a single block causes 4 physical I/Os to take place. Existing file systems do not try to avoid this worst-case RAID writing behavior.

TIP: DETAILS MATTER

All interesting systems are comprised of a few general ideas and a number of details. Sometimes, when you are learning about these systems, you think to yourself "Oh, I get the general idea; the rest is just details," and you use this to only half-learn how things really work. Don't do this! Many times, the details are critical. As we'll see with LFS, the general idea is easy to understand, but to really build a working system, you have to think through *all* of the tricky cases.

An ideal file system would thus focus on write performance, and try to make use of the sequential bandwidth of the disk. Further, it would perform well on common workloads that not only write out data but also update on-disk metadata structures frequently. Finally, it would work well on RAIDs as well as single disks.

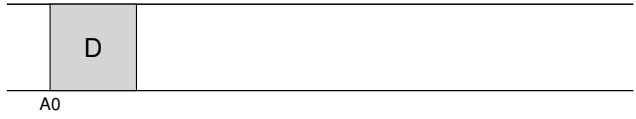
The new type of file system Rosenblum and Ousterhout introduced was called **LFS**, short for the **Log-structured File System**. When writing to disk, LFS first buffers all updates (including metadata!) in an in-memory **segment**; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather *always* writes segments to free locations. Because segments are large, the disk is used efficiently, and performance of the file system approaches its zenith.

THE CRUX:
HOW TO MAKE ALL WRITES SEQUENTIAL WRITES?

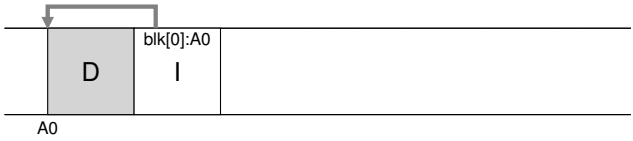
How can a file system transform all writes into sequential writes? For reads, this task is impossible, as the desired block to be read may be anywhere on disk. For writes, however, the file system always has a choice, and it is exactly this choice we hope to exploit.

43.1 Writing To Disk Sequentially

We thus have our first challenge: how do we transform all updates to file-system state into a series of sequential writes to disk? To understand this better, let's use a simple example. Imagine we are writing a data block D to a file. Writing the data block to disk might result in the following on-disk layout, with D written at disk address $A0$:



However, when a user writes a data block, it is not only data that gets written to disk; there is also other **metadata** that needs to be updated. In this case, let's also write the **inode** (I) of the file to disk, and have it point to the data block D. When written to disk, the data block and inode would look something like this (note that the inode looks as big as the data block, which generally isn't the case; in most systems, data blocks are 4 KB in size, whereas an inode is much smaller, around 128 bytes):



This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS. If you understand this, you get the basic idea. But as with all complicated systems, the devil is in the details.

43.2 Writing Sequentially And Effectively

Unfortunately, writing to disk sequentially is not (alone) enough to guarantee efficient writes. For example, imagine if we wrote a single block to address A, at time T. We then wait a little while, and write to the disk at address $A + 1$ (the next block address in sequential order), but at time $T + \delta$. In-between the first and second writes, unfortunately, the disk has rotated; when you issue the second write, it will thus wait for most of a rotation before being committed (specifically, if the rotation takes time $T_{rotation}$, the disk will wait $T_{rotation} - \delta$ before it can commit the second write to the disk surface). And thus you can hopefully see that simply writing to disk in sequential order is not enough to achieve peak performance; rather, you must issue a large number of *contiguous* writes (or one large write) to the drive in order to achieve good write performance.

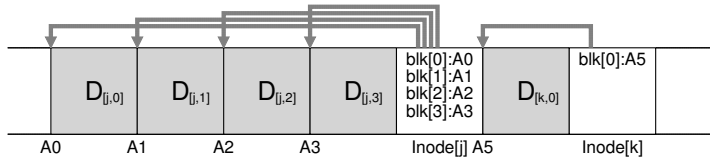
To achieve this end, LFS uses an ancient technique known as **write buffering**[1]. Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk.

The large chunk of updates LFS writes at one time is referred to by the name of a **segment**. Although this term is over-used in computer systems, here it just means a large-ish chunk which LFS uses to group writes. Thus, when writing to disk, LFS buffers updates in an in-memory

[1] Indeed, it is hard to find a good citation for this idea, since it was likely invented by many and very early on in the history of computing. For a study of the benefits of write buffering, see Solworth and Orji [SO90]; to learn about its potential harms, see Mogul [M94].

segment, and then writes the segment all at once to the disk. As long as the segment is large enough, these writes will be efficient.

Here is an example, in which LFS buffers two sets of updates into a small segment; actual segments are larger (a few MB). The first update is of four block writes to file j; the second is one block being added to file k. LFS then commits the entire segment of seven blocks to disk at once. The resulting on-disk layout of these blocks is as follows:



43.3 How Much To Buffer?

This raises the following question: how many updates should LFS buffer before writing to disk? The answer, of course, depends on the disk itself, specifically how high the positioning overhead is in comparison to the transfer rate; see the FFS chapter for a similar analysis.

For example, assume that positioning (i.e., rotation and seek overheads) before each write takes roughly $T_{position}$ seconds. Assume further that the disk transfer rate is R_{peak} MB/s. How much should LFS buffer before writing when running on such a disk?

The way to think about this is that every time you write, you pay a fixed overhead of the positioning cost. Thus, how much do you have to write in order to **amortize** that cost? The more you write, the better (obviously), and the closer you get to achieving peak bandwidth.

To obtain a concrete answer, let's assume we are writing out D MB. The time to write out this chunk of data (T_{write}) is the positioning time $T_{position}$ plus the time to transfer D ($\frac{D}{R_{peak}}$), or:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{43.1}$$

And thus the effective *rate* of writing ($R_{effective}$), which is just the amount of data written divided by the total time to write it, is:

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}. \tag{43.2}$$

What we're interested in is getting the effective rate ($R_{effective}$) close to the peak rate. Specifically, we want the effective rate to be some fraction F of the peak rate, where $0 < F < 1$ (a typical F might be 0.9, or 90% of the peak rate). In mathematical form, this means we want $R_{effective} = F \times R_{peak}$.

At this point, we can solve for D:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \tag{43.3}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \tag{43.4}$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \tag{43.5}$$

$$D = \frac{F}{1 - F} \times R_{peak} \times T_{position} \tag{43.6}$$

Let's do an example, with a disk with a positioning time of 10 milliseconds and peak transfer rate of 100 MB/s; assume we want an effective bandwidth of 90% of peak ($F = 0.9$). In this case, $D = \frac{0.9}{0.1} \times 100\ MB/s \times 0.01\ seconds = 9\ MB$. Try some different values to see how much we need to buffer in order to approach peak bandwidth. How much is needed to reach 95% of peak? 99%?

43.4 Problem: Finding Inodes

To understand how we find an inode in LFS, let us briefly review how to find an inode in a typical UNIX file system. In a typical file system such as FFS, or even the old UNIX file system, finding inodes is easy, because they are organized in an array and placed on disk at fixed locations.

For example, the old UNIX file system keeps all inodes at a fixed portion of the disk. Thus, given an inode number and the start address, to find a particular inode, you can calculate its exact disk address simply by multiplying the inode number by the size of an inode, and adding that to the start address of the on-disk array; array-based indexing, given an inode number, is fast and straightforward.

Finding an inode given an inode number in FFS is only slightly more complicated, because FFS splits up the inode table into chunks and places a group of inodes within each cylinder group. Thus, one must know how big each chunk of inodes is and the start addresses of each. After that, the calculations are similar and also easy.

In LFS, life is more difficult. Why? Well, we've managed to scatter the inodes all throughout the disk! Worse, we never overwrite in place, and thus the latest version of an inode (i.e., the one we want) keeps moving.

43.5 Solution Through Indirection: The Inode Map

To remedy this, the designers of LFS introduced a **level of indirection** between inode numbers and the inodes through a data structure called the **inode map (imap)**. The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the

TIP: USE A LEVEL OF INDIRECTION

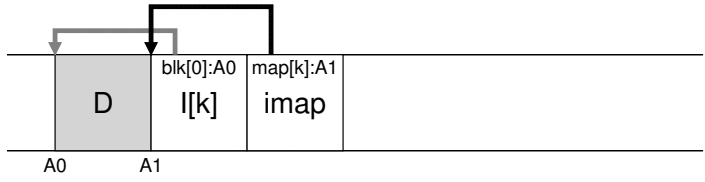
People often say that the solution to all problems in Computer Science is simply a **level of indirection**. This is clearly not true; it is just the solution to *most* problems (yes, this is still too strong of a comment, but you get the point). You certainly can think of every virtualization we have studied, e.g., virtual memory, or the notion of a file, as simply a level of indirection. And certainly the inode map in LFS is a virtualization of inode numbers. Hopefully you can see the great power of indirection in these examples, allowing us to freely move structures around (such as pages in the VM example, or inodes in LFS) without having to change every reference to them. Of course, indirection can have a downside too: **extra overhead.** So next time you have a problem, try solving it with indirection, but make sure to think about the overheads of doing so first. As Wheeler famously said, "All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections."

inode. Thus, you can imagine it would often be implemented as a simple *array*, with 4 bytes (a disk pointer) per entry. Any time an inode is written to disk, the imap is updated with its new location.

The imap, unfortunately, needs to be kept persistent (i.e., written to disk); doing so allows LFS to keep track of the locations of inodes across crashes, and thus operate as desired. Thus, a question: where should the imap reside on disk?

It could live on a fixed part of the disk, of course. Unfortunately, as it gets updated frequently, this would then require updates to file structures to be followed by writes to the imap, and hence performance would suffer (i.e., there would be more disk seeks, between each update and the fixed location of the imap).

Instead, LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk, as follows:



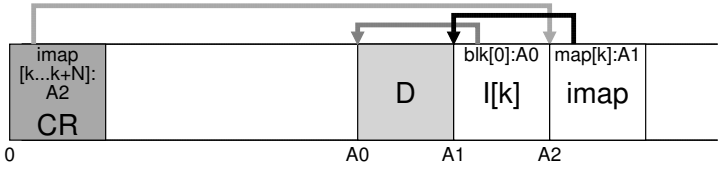
In this picture, the piece of the imap array stored in the block marked *imap* tells LFS that the inode k is at disk address $A1$; this inode, in turn, tells LFS that its data block D is at address $A0$.

43.6 Completing The Solution: The Checkpoint Region

The clever reader (that's you, right?) might have noticed a problem here. How do we find the inode map, now that pieces of it are also now spread across the disk? In the end, there is no magic: the file system must have *some* fixed and known location on disk to begin a file lookup.

LFS has just such a fixed place on disk for this, known as the **checkpoint region (CR)**. The checkpoint region contains pointers to (i.e., addresses of) the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first. Note the checkpoint region is only updated periodically (say every 30 seconds or so), and thus performance is not ill-affected. Thus, the overall structure of the on-disk layout contains a checkpoint region (which points to the latest pieces of the inode map); the inode map pieces each contain addresses of the inodes; the inodes point to files (and directories) just like typical UNIX file systems.

Here is an example of the checkpoint region (note it is all the way at the beginning of the disk, at address 0), and a single imap chunk, inode, and data block. A real file system would of course have a much bigger CR (indeed, it would have two, as we'll come to understand later), many imap chunks, and of course many more inodes, data blocks, etc.



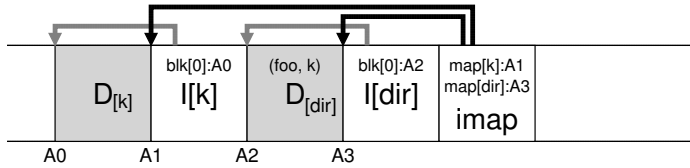
43.7 Reading A File From Disk: A Recap

To make sure you understand how LFS works, let us now walk through what must happen to read a file from disk. Assume we have nothing in memory to begin. The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers (i.e., disk addresses) to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-disk-address mapping in the imap, and reads in the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirect pointers as need be. In the common case, LFS should perform the same number of I/Os as a typical file system when reading a file from disk; the entire imap is cached and thus the extra work LFS does during a read is to look up the inode's address in the imap.

43.8 What About Directories?

Thus far, we've simplified our discussion a bit by only considering inodes and data blocks. However, to access a file in a file system (such as `/home/remzi/foo`, one of our favorite fake file names), some directories must be accessed too. So how does LFS store directory data?

Fortunately, directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings. For example, when creating a file on disk, LFS must both write a new inode, some data, as well as the directory data and its inode that refer to this file. Remember that LFS will do so sequentially on the disk (after buffering the updates for some time). Thus, creating a file `foo` in a directory would lead to the following new structures on disk:



The piece of the inode map contains the information for the location of both the directory file dir as well as the newly-created file f. Thus, when accessing file `foo` (with inode number k), you would first look in the inode map (usually cached in memory) to find the location of the inode of directory dir ($A3$); you then read the directory inode, which gives you the location of the directory data ($A2$); reading this data block gives you the name-to-inode-number mapping of (`foo`, k). You then consult the inode map again to find the location of inode number k ($A1$), and finally read the desired data block at address $A0$.

There is one other serious problem in LFS that the inode map solves, known as the **recursive update problem** [Z+12]. The problem arises in any file system that never updates in place (such as LFS), but rather moves updates to new locations on the disk.

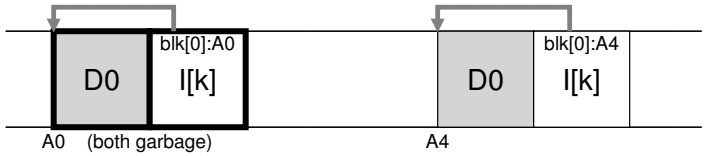
Specifically, whenever an inode is updated, its location on disk changes. If we hadn't been careful, this would have also entailed an update to the directory that points to this file, which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree.

LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inumber mapping. Thus, through indirection, LFS avoids the recursive update problem.

43.9 A New Problem: Garbage Collection

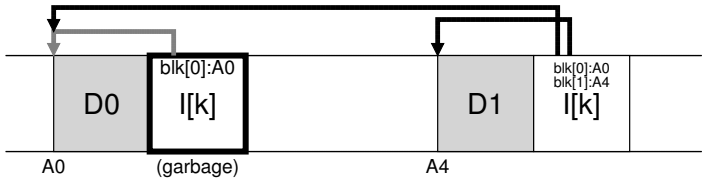
You may have noticed another problem with LFS; it repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This process, while keeping writes efficient, implies that LFS leaves old versions of file structures scattered throughout the disk. We (rather unceremoniously) call these old versions **garbage**.

For example, let's imagine the case where we have an existing file referred to by inode number k, which points to a single data block $D0$. We now update that block, generating both a new inode and a new data block. The resulting on-disk layout of LFS would look something like this (note we omit the imap and other structures for simplicity; a new chunk of imap would also have to be written to disk to point to the new inode):



In the diagram, you can see that both the inode and data block have two versions on disk, one old (the one on the left) and one current and thus **live** (the one on the right). By the simple act of (logically) updating a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on the disk.

As another example, imagine we instead append a block to that original file k. In this case, a new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it is still live and very much part of the current file system:



So what should we do with these older versions of inodes, data blocks, and so forth? One could keep those older versions around and allow users to restore old file versions (for example, when they accidentally overwrite or delete a file, it could be quite handy to do so); such a file system is known as a **versioning file system** because it keeps track of the different versions of a file.

However, LFS instead keeps only the latest live version of a file; thus (in the background), LFS must periodically find these old dead versions of file data, inodes, and other structures, and **clean** them; cleaning should

thus make blocks on disk free again for use in a subsequent writes. Note that the process of cleaning is a form of **garbage collection**, a technique that arises in programming languages that automatically free unused memory for programs.

Earlier we discussed segments as important as they are the mechanism that enables large writes to disk in LFS. As it turns out, they are also quite integral to effective cleaning. Imagine what would happen if the LFS cleaner simply went through and freed single data blocks, inodes, etc., during cleaning. The result: a file system with some number of free **holes** mixed between allocated space on disk. Write performance would drop considerably, as LFS would not be able to find a large contiguous region to write to disk sequentially and with high performance.

Instead, the LFS cleaner works on a segment-by-segment basis, thus clearing up large chunks of space for subsequent writing. The basic cleaning process works as follows. Periodically, the LFS cleaner reads in a number of old (partially-used) segments, determines which blocks are live within these segments, and then write out a new set of segments with just the live blocks within them, freeing up the old ones for writing. Specifically, we expect the cleaner to read in M existing segments, **compact** their contents into N new segments (where $N < M$), and then write the N segments to disk in new locations. The old M segments are then freed and can be used by the file system for subsequent writes.

We are now left with two problems, however. The first is mechanism: how can LFS tell which blocks within a segment are live, and which are dead? The second is policy: how often should the cleaner run, and which segments should it pick to clean?

43.10 Determining Block Liveness

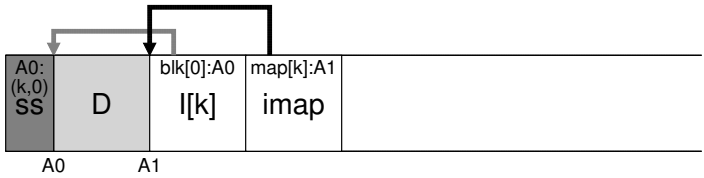
We address the mechanism first. Given a data block D within an on-disk segment S, LFS must be able to determine whether D is live. To do so, LFS adds a little extra information to each segment that describes each block. Specifically, LFS includes, for each data block D, its inode number (which file it belongs to) and its offset (which block of the file this is). This information is recorded in a structure at the head of the segment known as the **segment summary block**.

Given this information, it is straightforward to determine whether a block is live or dead. For a block D located on disk at address A, look in the segment summary block and find its inode number N and offset T. Next, look in the imap to find where N lives and read N from disk (perhaps it is already in memory, which is even better). Finally, using the offset T, look in the inode (or some indirect block) to see where the inode thinks the Tth block of this file is on disk. If it points exactly to disk address A, LFS can conclude that the block D is live. If it points anywhere else, LFS can conclude that D is not in use (i.e., it is dead) and thus know that this version is no longer needed. A pseudocode summary of this

process is shown here:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

Here is a diagram depicting the mechanism, in which the segment summary block (marked SS) records that the data block at address $A0$ is actually a part of file k at offset 0. By checking the imap for k, you can find the inode, and see that it does indeed point to that location.



There are some shortcuts LFS takes to make the process of determining liveness more efficient. For example, when a file is truncated or deleted, LFS increases its **version number** and records the new version number in the imap. By also recording the version number in the on-disk segment, LFS can short circuit the longer check described above simply by comparing the on-disk version number with a version number in the imap, thus avoiding extra reads.

43.11 A Policy Question: Which Blocks To Clean, And When?

On top of the mechanism described above, LFS must include a set of policies to determine both when to clean and which blocks are worth cleaning. Determining when to clean is easier; either periodically, during idle time, or when you have to because the disk is full.

Determining which blocks to clean is more challenging, and has been the subject of many research papers. In the original LFS paper [RO91], the authors describe an approach which tries to segregate *hot* and *cold* segments. A hot segment is one in which the contents are being frequently over-written; thus, for such a segment, the best policy is to wait a long time before cleaning it, as more and more blocks are getting over-written (in new segments) and thus being freed for use. A cold segment, in contrast, may have a few dead blocks but the rest of its contents are relatively stable. Thus, the authors conclude that one should clean cold segments sooner and hot segments later, and develop a heuristic that does exactly that. However, as with most policies, this policy isn't perfect; later approaches show how to do better [MR+97].

43.12 Crash Recovery And The Log

One final problem: what happens if the system crashes while LFS is writing to disk? As you may recall in the previous chapter about journaling, crashes during updates are tricky for file systems, and thus something LFS must consider as well.

During normal operation, LFS buffers writes in a segment, and then (when the segment is full, or when some amount of time has elapsed), writes the segment to disk. LFS organizes these writes in a **log**, i.e., the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written. LFS also periodically updates the checkpoint region. Crashes could clearly happen during either of these operations (write to a segment, write to the CR). So how does LFS handle crashes during writes to these structures?

Let's cover the second case first. To ensure that the CR update happens atomically, LFS actually keeps two CRs, one at either end of the disk, and writes to them alternately. LFS also implements a careful protocol when updating the CR with the latest pointers to the inode map and other information; specifically, it first writes out a header (with timestamp), then the body of the CR, and then finally one last block (also with a timestamp). If the system crashes during a CR update, LFS can detect this by seeing an inconsistent pair of timestamps. LFS will always choose to use the most recent CR that has consistent timestamps, and thus consistent update of the CR is achieved.

Let's now address the first case. Because LFS writes the CR every 30 seconds or so, the last consistent snapshot of the file system may be quite old. Thus, upon reboot, LFS can easily recover by simply reading in the checkpoint region, the imap pieces it points to, and subsequent files and directories; however, the last many seconds of updates would be lost.

To improve upon this, LFS tries to rebuild many of those segments through a technique known as **roll forward** in the database community. The basic idea is to start with the last checkpoint region, find the end of the log (which is included in the CR), and then use that to read through the next segments and see if there are any valid updates within it. If there are, LFS updates the file system accordingly and thus recovers much of the data and metadata written since the last checkpoint. See Rosenblum's award-winning dissertation for details [R92].

43.13 Summary

LFS introduces a new approach to updating the disk. Instead of overwriting files in places, LFS always writes to an unused portion of the disk, and then later reclaims that old space through cleaning. This approach, which in database systems is called **shadow paging** [L77] and in file-system-speak is sometimes called **copy-on-write**, enables highly efficient writing, as LFS can gather all updates into an in-memory segment and then write them out together sequentially.

TIP: TURN FLAWS INTO VIRTUES

Whenever your system has a fundamental flaw, see if you can turn it around into a feature or something useful. NetApp's WAFL does this with old file contents; by making old versions available, WAFL no longer has to worry about cleaning quite so often (though it does delete old versions, eventually, in the background), and thus provides a cool feature and removes much of the LFS cleaning problem all in one wonderful twist. Are there other examples of this in systems? Undoubtedly, but you'll have to think of them yourself, because this chapter is over with a capital "O". Over. Done. Kaput. We're out. Peace!

The downside to this approach is that it generates garbage; old copies of the data are scattered throughout the disk, and if one wants to reclaim such space for subsequent usage, one must clean old segments periodically. Cleaning became the focus of much controversy in LFS, and concerns over cleaning costs [SS+95] perhaps limited LFS's initial impact on the field. However, some modern commercial file systems, including NetApp's **WAFL** [HLM94], Sun's **ZFS** [B07], and Linux **btrfs** [M07], and even modern **flash-based SSDs** [AD14], adopt a similar copy-on-write approach to writing to disk, and thus the intellectual legacy of LFS lives on in these modern file systems. In particular, WAFL got around cleaning problems by turning them into a feature; by providing old versions of the file system via **snapshots**, users could access old files whenever they deleted current ones accidentally.

References

[AD14] "Operating Systems: Three Easy Pieces"
Chapter: Flash-based Solid State Drives
Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau
A bit gauche to refer you to another chapter in this very book, but who are we to judge?

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Copy Available: http://www.ostep.org/Citations/zfs_last.pdf
Slides on ZFS; unfortunately, there is no great ZFS paper (yet). Maybe you will write one, so we can cite it here?

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Spring '94
WAFL takes many ideas from LFS and RAID and puts it into a high-speed NFS appliance for the multi-billion dollar storage company NetApp.

[L77] "Physical Integrity in a Large Segmented Database"
R. Lorie
ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104
The original idea of shadow paging is presented here.

[M07] "The Btrfs Filesystem"
Chris Mason
September 2007
Available: oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf
A recent copy-on-write Linux file system, slowly gaining in importance and usage.

[MJLF84] "A Fast File System for UNIX"
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry
ACM TOCS, August, 1984, Volume 2, Number 3
The original FFS paper; see the chapter on FFS for more details.

[MR+97] "Improving the Performance of Log-structured File Systems
with Adaptive Methods"
Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello,
Randolph Y. Wang, Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France
A more recent paper detailing better policies for cleaning in LFS.

[M94] "A Better Update Policy"
Jeffrey C. Mogul
USENIX ATC '94, June 1994
In this paper, Mogul finds that read workloads can be harmed by buffering writes for too long and then sending them to the disk in a big burst. Thus, he recommends sending writes more frequently and in smaller batches.

[P98] "Hardware Technology Trends and Database Opportunities"
David A. Patterson
ACM SIGMOD '98 Keynote Address, Presented June 3, 1998, Seattle, Washington
Available: http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html
A great set of slides on technology trends in computer systems. Hopefully, Patterson will create another of these sometime soon.

[RO91] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum and John Ousterhout
SOSP '91, Pacific Grove, CA, October 1991
The original SOSP paper about LFS, which has been cited by hundreds of other papers and inspired many real systems.

[R92] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum
http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/CSD-92-696.pdf
The award-winning dissertation about LFS, with many of the details missing from the paper.

[SS+95] "File system logging versus clustering: a performance comparison"
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995
A paper that showed the LFS performance sometimes has problems, particularly for workloads with many calls to `fsync()` (such as database workloads). The paper was controversial at the time.

[SO90] "Write-Only Disk Caches"
Jon A. Solworth, Cyril U. Orji
SIGMOD '90, Atlantic City, New Jersey, May 1990
An early study of write buffering and its benefits. However, buffering for too long can be harmful: see Mogul [M94] for details.

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
Our paper on a new way to build flash-based storage devices. Because FTLs (flash-translation layers) are usually built in a log-structured style, some of the same issues arise in flash-based devices that do in LFS. In this case, it is the recursive update problem, which LFS solves neatly with an imap. A similar structure exists in most SSDs.

44

Data Integrity and Protection

Beyond the basic advances found in the file systems we have studied thus far, a number of features are worth studying. In this chapter, we focus on reliability once again (having previously studied storage system reliability in the RAID chapter). Specifically, how should a file system or storage system ensure that data is safe, given the unreliable nature of modern storage devices?

This general area is referred to as **data integrity** or **data protection**. Thus, we will now investigate techniques used to ensure that the data you put into your storage system is the same when the storage system returns it to you.

CRUX: HOW TO ENSURE DATA INTEGRITY

How should systems ensure that the data written to storage is protected? What techniques are required? How can such techniques be made efficient, with both low space and time overheads?

44.1 Disk Failure Modes

As you learned in the chapter about RAID, disks are not perfect, and can fail (on occasion). In early RAID systems, the model of failure was quite simple: either the entire disk is working, or it fails completely, and the detection of such a failure is straightforward. This **fail-stop** model of disk failure makes building RAID relatively simple [S90].

What you didn't learn is about all of the other types of failure modes modern disks exhibit. Specifically, as Bairavasundaram et al. studied in great detail [B+07, B+08], modern disks will occasionally seem to be mostly working but have trouble successfully accessing one or more blocks. Specifically, two types of single-block failures are common and worthy of consideration: **latent-sector errors** (**LSEs**) and **block corruption**. We'll now discuss each in more detail.

	Cheap	Costly
LSEs	9.40%	1.40%
Corruption	0.50%	0.05%

Figure 44.1: **Frequency Of LSEs And Block Corruption**

LSEs arise when a disk sector (or group of sectors) has been damaged in some way. For example, if the disk head touches the surface for some reason (a **head crash**, something which shouldn't happen during normal operation), it may damage the surface, making the bits unreadable. Cosmic rays can also flip bits, leading to incorrect contents. Fortunately, in-disk **error correcting codes** (**ECC**) are used by the drive to determine whether the on-disk bits in a block are good, and in some cases, to fix them; if they are not good, and the drive does not have enough information to fix the error, the disk will return an error when a request is issued to read them.

There are also cases where a disk block becomes **corrupt** in a way not detectable by the disk itself. For example, buggy disk firmware may write a block to the wrong location; in such a case, the disk ECC indicates the block contents are fine, but from the client's perspective the wrong block is returned when subsequently accessed. Similarly, a block may get corrupted when it is transferred from the host to the disk across a faulty bus; the resulting corrupt data is stored by the disk, but it is not what the client desires. These types of faults are particularly insidious because they are **silent faults**; the disk gives no indication of the problem when returning the faulty data.

Prabhakaran et al. describes this more modern view of disk failure as the **fail-partial** disk failure model [P+05]. In this view, disks can still fail in their entirety (as was the case in the traditional fail-stop model); however, disks can also seemingly be working and have one or more blocks become inaccessible (i.e., LSEs) or hold the wrong contents (i.e., corruption). Thus, when accessing a seemingly-working disk, once in a while it may either return an error when trying to read or write a given block (a non-silent partial fault), and once in a while it may simply return the wrong data (a silent partial fault).

Both of these types of faults are somewhat rare, but just how rare? Figure 44.1 summarizes some of the findings from the two Bairavasundaram studies [B+07,B+08].

The figure shows the percent of drives that exhibited at least one LSE or block corruption over the course of the study (about 3 years, over 1.5 million disk drives). The figure further sub-divides the results into "cheap" drives (usually SATA drives) and "costly" drives (usually SCSI or FibreChannel). As you can see, while buying better drives reduces the frequency of both types of problem (by about an order of magnitude), they still happen often enough that you need to think carefully about how to handle them in your storage system.

Some additional findings about LSEs are:

- Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
- For most drives, annual error rate increases in year two
- The number of LSEs increase with disk size
- Most disks with LSEs have less than 50
- Disks with LSEs are more likely to develop additional LSEs
- There exists a significant amount of spatial and temporal locality
- Disk scrubbing is useful (most LSEs were found this way)

Some findings about corruption:

- Chance of corruption varies greatly across different drive models within the same drive class
- Age effects are different across models
- Workload and disk size have little impact on corruption
- Most disks with corruption only have a few corruptions
- Corruption is not independent within a disk or across disks in RAID
- There exists spatial locality, and some temporal locality
- There is a weak correlation with LSEs

To learn more about these failures, you should likely read the original papers [B+07,B+08]. But hopefully the main point should be clear: if you really wish to build a reliable storage system, you must include machinery to detect and recover from both LSEs and block corruption.

44.2 Handling Latent Sector Errors

Given these two new modes of partial disk failure, we should now try to see what we can do about them. Let's first tackle the easier of the two, namely latent sector errors.

CRUX: HOW TO HANDLE LATENT SECTOR ERRORS
How should a storage system handle latent sector errors? How much extra machinery is needed to handle this form of partial failure?

As it turns out, latent sector errors are rather straightforward to handle, as they are (by definition) easily detected. When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data. In a mirrored RAID, for example, the system should access the alternate copy; in a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group. Thus, easily detected problems such as LSEs are readily recovered through standard redundancy mechanisms.

The growing prevalence of LSEs has influenced RAID designs over the years. One particularly interesting problem arises in RAID-4/5 systems when both full-disk faults and LSEs occur in tandem. Specifically, when an entire disk fails, the RAID tries to **reconstruct** the disk (say, onto a hot spare) by reading through all of the other disks in the parity group and recomputing the missing values. If, during reconstruction, an LSE is encountered on any one of the other disks, we have a problem: the reconstruction cannot successfully complete.

To combat this issue, some systems add an extra degree of redundancy. For example, NetApp's **RAID-DP** has the equivalent of two parity disks instead of one [C+04]. When an LSE is discovered during reconstruction, the extra parity helps to reconstruct the missing block. As always, there is a cost, in that maintaining two parity blocks for each stripe is more costly; however, the log-structured nature of the NetApp **WAFL** file system mitigates that cost in many cases [HLM94]. The remaining cost is space, in the form of an extra disk for the second parity block.

44.3 Detecting Corruption: The Checksum

Let's now tackle the more challenging problem, that of silent failures via data corruption. How can we prevent users from getting bad data when corruption arises, and thus leads to disks returning bad data?

CRUX: HOW TO PRESERVE DATA INTEGRITY DESPITE CORRUPTION
Given the silent nature of such failures, what can a storage system do to detect when corruption arises? What techniques are needed? How can one implement them efficiently?

Unlike latent sector errors, *detection* of corruption is a key problem. How can a client tell that a block has gone bad? Once it is known that a particular block is bad, *recovery* is the same as before: you need to have some other copy of the block around (and hopefully, one that is not corrupt!). Thus, we focus here on detection techniques.

The primary mechanism used by modern storage systems to preserve data integrity is called the **checksum**. A checksum is simply the result of a function that takes a chunk of data (say a 4KB block) as input and computes a function over said data, producing a small summary of the contents of the data (say 4 or 8 bytes). This summary is referred to as the checksum. The goal of such a computation is to enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and then confirming upon later access that the data's current checksum matches the original storage value.

TIP: THERE'S NO FREE LUNCH

There's No Such Thing As A Free Lunch, or TNSTAAFL for short, is an old American idiom that implies that when you are seemingly getting something for free, in actuality you are likely paying some cost for it. It comes from the old days when diners would advertise a free lunch for customers, hoping to draw them in; only when you went in, did you realize that to acquire the "free" lunch, you had to purchase one or more alcoholic beverages. Of course, this may not actually be a problem, particularly if you are an aspiring alcoholic (or typical undergraduate student).

Common Checksum Functions

A number of different functions are used to compute checksums, and vary in strength (i.e., how good they are at protecting data integrity) and speed (i.e., how quickly can they be computed). A trade-off that is common in systems arises here: usually, the more protection you get, the costlier it is. There is no such thing as a free lunch.

One simple checksum function that some use is based on exclusive or (XOR). With XOR-based checksums, the checksum is computed by XOR'ing each chunk of the data block being checksummed, thus producing a single value that represents the XOR of the entire block.

To make this more concrete, imagine we are computing a 4-byte checksum over a block of 16 bytes (this block is of course too small to really be a disk sector or block, but it will serve for the example). The 16 data bytes, in hex, look like this:

```
365e c4cd ba14 8a92 ecef 2c3a 40be f666
```

If we view them in binary, we get the following:

```
0011 0110 0101 1110    1100 0100 1100 1101
1011 1010 0001 0100    1000 1010 1001 0010
1110 1100 1110 1111    0010 1100 0011 1010
0100 0000 1011 1110    1111 0110 0110 0110
```

Because we've lined up the data in groups of 4 bytes per row, it is easy to see what the resulting checksum will be: perform an XOR over each column to get the final checksum value:

```
0010 0000 0001 1011    1001 0100 0000 0011
```

The result, in hex, is 0x201b9403.

XOR is a reasonable checksum but has its limitations. If, for example, two bits in the same position within each checksummed unit change, the checksum will not detect the corruption. For this reason, people have investigated other checksum functions.

Another basic checksum function is addition. This approach has the advantage of being fast; computing it just requires performing 2's-complement addition over each chunk of the data, ignoring overflow. It can detect many changes in data, but is not good if the data, for example, is shifted.

A slightly more complex algorithm is known as the **Fletcher checksum**, named (as you might guess) for the inventor, John G. Fletcher [F82]. It is quite simple to compute and involves the computation of two check bytes, $s1$ and $s2$. Specifically, assume a block D consists of bytes $d1$ ... dn; $s1$ is defined as follows: $s1 = s1 + d_i \ mod \ 255$ (computed over all d_i); $s2$ in turn is: $s2 = s2 + s1 \ mod \ 255$ (again over all d_i) [F04]. The fletcher checksum is known to be almost as strong as the CRC (described next), detecting all single-bit errors, all double-bit errors, and a large percentage of burst errors [F04].

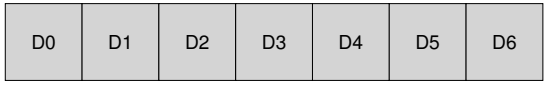
One final commonly-used checksum is known as a **cyclic redundancy check** (**CRC**). Assume you wish to compute the checksum over a data block D. All you do is treat D as if it is a large binary number (it is just a string of bits after all) and divide it by an agreed upon value (k). The remainder of this division is the value of the CRC. As it turns out, one can implement this binary modulo operation rather efficiently, and hence the popularity of the CRC in networking as well. See elsewhere for more details [M13].

Whatever the method used, it should be obvious that there is no perfect checksum: it is possible two data blocks with non-identical contents will have identical checksums, something referred to as a **collision**. This fact should be intuitive: after all, computing a checksum is taking something large (e.g., 4KB) and producing a summary that is much smaller (e.g., 4 or 8 bytes). In choosing a good checksum function, we are thus trying to find one that minimizes the chance of collisions while remaining easy to compute.

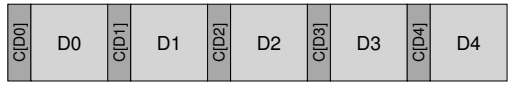
Checksum Layout

Now that you understand a bit about how to compute a checksum, let's next analyze how to use checksums in a storage system. The first question we must address is the layout of the checksum, i.e., how should checksums be stored on disk?

The most basic approach simply stores a checksum with each disk sector (or block). Given a data block D, let us call the checksum over that data $C(D)$. Thus, without checksums, the disk layout looks like this:



With checksums, the layout adds a single checksum for every block:



Because checksums are usually small (e.g., 8 bytes), and disks only can write in sector-sized chunks (512 bytes) or multiples thereof, one problem that arises is how to achieve the above layout. One solution employed by drive manufacturers is to format the drive with 520-byte sectors; an extra 8 bytes per sector can be used to store the checksum.

In disks that don't have such functionality, the file system must figure out a way to store the checksums packed into 512-byte blocks. One such possibility is as follows:

In this scheme, the n checksums are stored together in a sector, followed by n data blocks, followed by another checksum sector for the next n blocks, and so forth. This scheme has the benefit of working on all disks, but can be less efficient; if the file system, for example, wants to overwrite block $D1$, it has to read in the checksum sector containing $C(D1)$, update $C(D1)$ in it, and then write out the checksum sector as well as the new data block $D1$ (thus, one read and two writes). The earlier approach (of one checksum per sector) just performs a single write.

44.4 Using Checksums

With a checksum layout decided upon, we can now proceed to actually understand how to *use* the checksums. When reading a block D, the client (i.e., file system or storage controller) also reads its checksum from disk $C_s(D)$, which we call the **stored checksum** (hence the subscript C_s). The client then *computes* the checksum over the retrieved block D, which we call the **computed checksum** $C_c(D)$. At this point, the client compares the stored and computed checksums; if they are equal (i.e., $C_s(D)$ == $C_c(D)$, the data has likely not been corrupted, and thus can be safely returned to the user. If they do *not* match (i.e., $C_s(D)$!= $C_c(D)$), this implies the data has changed since the time it was stored (since the stored checksum reflects the value of the data at that time). In this case, we have a corruption, which our checksum has helped us to detect.

Given a corruption, the natural question is what should we do about it? If the storage system has a redundant copy, the answer is easy: try to use it instead. If the storage system has no such copy, the likely answer is to return an error. In either case, realize that corruption detection is not a

magic bullet; if there is no other way to get the non-corrupted data, you are simply out of luck.

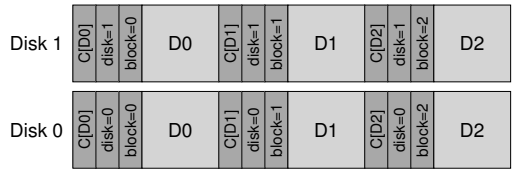
44.5 A New Problem: Misdirected Writes

The basic scheme described above works well in the general case of corrupted blocks. However, modern disks have a couple of unusual failure modes that require different solutions.

The first failure mode of interest is called a **misdirected write**. This arises in disk and RAID controllers which write the data to disk correctly, except in the *wrong* location. In a single-disk system, this means that the disk wrote block D_x not to address x (as desired) but rather to address y (thus "corrupting" D_y); in addition, within a multi-disk system, the controller may also write $D_{i,x}$ not to address x of disk i but rather to some other disk j. Thus our question:

CRUX: HOW TO HANDLE MISDIRECTED WRITES
How should a storage system or disk controller detect misdirected writes? What additional features are required from the checksum?

The answer, not surprisingly, is simple: add a little more information to each checksum. In this case, adding a **physical identifier** (**physical ID**) is quite helpful. For example, if the stored information now contains the checksum $C(D)$ as well as the disk and sector number of the block, it is easy for the client to determine whether the correct information resides within the block. Specifically, if the client is reading block 4 on disk 10 ($D_{10,4}$), the stored information should include that disk number and sector offset, as shown below. If the information does not match, a misdirected write has taken place, and a corruption is now detected. Here is an example of what this added information would look like on a two-disk system. Note that this figure, like the others before it, is not to scale, as the checksums are usually small (e.g., 8 bytes) whereas the blocks are much larger (e.g., 4 KB or bigger):



You can see from the on-disk format that there is now a fair amount of redundancy on disk: for each block, the disk number is repeated within each block, and the offset of the block in question is also kept next to the

block itself. The presence of redundant information should be no surprise, though; redundancy is the key to error detection (in this case) and recovery (in others). A little extra information, while not strictly needed with perfect disks, can go a long ways in helping detect problematic situations should they arise.

44.6 One Last Problem: Lost Writes

Unfortunately, misdirected writes are not the last problem we will address. Specifically, some modern storage devices also have an issue known as a **lost write**, which occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is left is the old contents of the block rather than the updated new contents.

The obvious question here is: do any of our checksumming strategies from above (e.g., basic checksums, or physical identity) help to detect lost writes? Unfortunately, the answer is no: the old block likely has a matching checksum, and the physical ID used above (disk number and block offset) will also be correct. Thus our final problem:

CRUX: HOW TO HANDLE LOST WRITES
How should a storage system or disk controller detect lost writes? What additional features are required from the checksum?

There are a number of possible solutions that can help [K+08]. One classic approach [BS04] is to perform a **write verify** or **read-after-write**; by immediately reading back the data after a write, a system can ensure that the data indeed reached the disk surface. This approach, however, is quite slow, doubling the number of I/Os needed to complete a write.

Some systems add a checksum elsewhere in the system to detect lost writes. For example, Sun's **Zettabyte File System** (**ZFS**) includes a checksum in each file system inode and indirect block for every block included within a file. Thus, even if the write to a data block itself is lost, the checksum within the inode will not match the old data. Only if the writes to both the inode and the data are lost simultaneously will such a scheme fail, an unlikely (but unfortunately, possible!) situation.

44.7 Scrubbing

Given all of this discussion, you might be wondering: when do these checksums actually get checked? Of course, some amount of checking occurs when data is accessed by applications, but most data is rarely accessed, and thus would remain unchecked. Unchecked data is problematic for a reliable storage system, as bit rot could eventually affect all copies of a particular piece of data.

To remedy this problem, many systems utilize **disk scrubbing** of various forms [K+08]. By periodically reading through *every* block of the system, and checking whether checksums are still valid, the disk system can reduce the chances that all copies of a certain data item become corrupted. Typical systems schedule scans on a nightly or weekly basis.

44.8 Overheads Of Checksumming

Before closing, we now discuss some of the overheads of using checksums for data protection. There are two distinct kinds of overheads, as is common in computer systems: space and time.

Space overheads come in two forms. The first is on the disk (or other storage medium) itself; each stored checksum takes up room on the disk, which can no longer be used for user data. A typical ratio might be an 8-byte checksum per 4 KB data block, for a 0.19% on-disk space overhead.

The second type of space overhead comes in the memory of the system. When accessing data, there must now be room in memory for the checksums as well as the data itself. However, if the system simply checks the checksum and then discards it once done, this overhead is short-lived and not much of a concern. Only if checksums are kept in memory (for an added level of protection against memory corruption [Z+13]) will this small overhead be observable.

While space overheads are small, the time overheads induced by checksumming can be quite noticeable. Minimally, the CPU must compute the checksum over each block, both when the data is stored (to determine the value of the stored checksum) as well as when it is accessed (to compute the checksum again and compare it against the stored checksum). One approach to reducing CPU overheads, employed by many systems that use checksums (including network stacks), is to combine data copying and checksumming into one streamlined activity; because the copy is needed anyhow (e.g., to copy the data from the kernel page cache into a user buffer), combined copying/checksumming can be quite effective.

Beyond CPU overheads, some checksumming schemes can induce extra I/O overheads, particularly when checksums are stored distinctly from the data (thus requiring extra I/Os to access them), and for any extra I/O needed for background scrubbing. The former can be reduced by design; the latter can be tuned and thus its impact limited, perhaps by controlling when such scrubbing activity takes place. The middle of the night, when most (not all!) productive workers have gone to bed, may be a good time to perform such scrubbing activity and increase the robustness of the storage system.

44.9 Summary

We have discussed data protection in modern storage systems, focusing on checksum implementation and usage. Different checksums protect

against different types of faults; as storage devices evolve, new failure modes will undoubtedly arise. Perhaps such change will force the research community and industry to revisit some of these basic approaches, or invent entirely new approaches altogether. Time will tell. Or it won't. Time is funny that way.

References

[B+07] "An Analysis of Latent Sector Errors in Disk Drives"
Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri Schindler
SIGMETRICS '07, San Diego, California, June 2007
The first paper to study latent sector errors in detail. As described in the next citation [B+08], a collaboration between Wisconsin and NetApp. The paper also won the Kenneth C. Sevcik Outstanding Student Paper award; Sevcik was a terrific researcher and wonderful guy who passed away too soon. To show the authors it was possible to move from the U.S. to Canada and love it, he once sang us the Canadian national anthem, standing up in the middle of a restaurant to do so.

[B+08] "An Analysis of Data Corruption in the Storage Stack"
Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
The first paper to truly study disk corruption in great detail, focusing on how often such corruption occurs over three years for over 1.5 million drives. Lakshmi did this work while a graduate student at Wisconsin under our supervision, but also in collaboration with his colleagues at NetApp where he was an intern for multiple summers. A great example of how working with industry can make for much more interesting and relevant research.

[BS04] "Commercial Fault Tolerance: A Tale of Two Systems"
Wendy Bartlett, Lisa Spainhower
IEEE Transactions on Dependable and Secure Computing, Vol. 1, No. 1, January 2004
This classic in building fault tolerant systems is an excellent overview of the state of the art from both IBM and Tandem. Another must read for those interested in the area.

[C+04] "Row-Diagonal Parity for Double Disk Failure Correction"
P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar
FAST '04, San Jose, CA, February 2004
An early paper on how extra redundancy helps to solve the combined full-disk-failure/partial-disk-failure problem. Also a nice example of how to mix more theoretical work with practical.

[F04] "Checksums and Error Control"
Peter M. Fenwick
Copy Available: http://www.ostep.org/Citations/checksums-03.pdf
A great simple tutorial on checksums, available to you for the amazing cost of free.

[F82] "An Arithmetic Checksum for Serial Transmissions"
John G. Fletcher
IEEE Transactions on Communication, Vol. 30, No. 1, January 1982
*Fletcher's original work on his eponymous checksum. Of course, he didn't call it the Fletcher checksum, rather he just didn't call it anything, and thus it became natural to name it after the inventor. So don't blame old Fletch for this seeming act of braggadocio. This anecdote might remind you of Rubik and his cube; Rubik never called it "**Rubik's cube**"; rather, he just called it "my cube."*

[HLM94] "File System Design for an NFS File Server Appliance"
Dave Hitz, James Lau, Michael Malcolm
USENIX Spring '94
The pioneering paper that describes the ideas and product at the heart of NetApp's core. Based on this system, NetApp has grown into a multi-billion dollar storage company. If you're interested in learning more about its founding, read Hitz's autobiography "How to Castrate a Bull: Unexpected Lessons on Risk, Growth, and Success in Business" (which is the actual title, no joking). And you thought you could avoid bull castration by going into Computer Science.

[K+08] "Parity Lost and Parity Regained"
Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '08, San Jose, CA, February 2008
This work of ours, joint with colleagues at NetApp, explores how different checksum schemes work (or don't work) in protecting data. We reveal a number of interesting flaws in current protection strategies, some of which have led to fixes in commercial products.

[M13] "Cyclic Redundancy Checks"
Author Unknown
Available: http://www.mathpages.com/home/kmath458.htm
Not sure who wrote this, but a super clear and concise description of CRCs is available here. The internet is full of information, as it turns out.

[P+05] "IRON File Systems"
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP '05, Brighton, England, October 2005
Our paper on how disks have partial failure modes, which includes a detailed study of how file systems such as Linux ext3 and Windows NTFS react to such failures. As it turns out, rather poorly! We found numerous bugs, design flaws, and other oddities in this work. Some of this has fed back into the Linux community, thus helping to yield a new more robust group of file systems to store your data.

[RO91] "Design and Implementation of the Log-structured File System"
Mendel Rosenblum and John Ousterhout
SOSP '91, Pacific Grove, CA, October 1991
Another reference to this ground-breaking paper on how to improve write performance in file systems.

[S90] "Implementing Fault-Tolerant Services Using The State Machine Approach: A Tutorial"
Fred B. Schneider
ACM Surveys, Vol. 22, No. 4, December 1990
This classic paper talks generally about how to build fault tolerant services, and includes many basic definitions of terms. A must read for those building distributed systems.

[Z+13] "Zettabyte Reliability with Flexible End-to-end Data Integrity"
Yupu Zhang, Daniel S. Myers, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
MSST '13, Long Beach, California, May 2013
Our own work on adding data protection to the page cache of a system, which protects against memory corruption as well as on-disk corruption.

45

Summary Dialogue on Persistence

Student: *Wow, file systems seem interesting(!), and yet complicated.*

Professor: *That's why me and my spouse do our research in this space.*

Student: *Hold on. Are you one of the professors who wrote this book? I thought we were both just fake constructs, used to summarize some main points, and perhaps add a little levity in the study of operating systems.*

Professor: *Uh... er... maybe. And none of your business! And who did you think was writing these things? (sighs) Anyhow, let's get on with it: what did you learn?*

Student: *Well, I think I got one of the main points, which is that it is much harder to manage data for a long time (persistently) than it is to manage data that isn't persistent (like the stuff in memory). After all, if your machines crashes, memory contents disappear! But the stuff in the file system needs to live forever.*

Professor: *Well, as my friend Kevin Hultquist used to say, "Forever is a long time"; while he was talking about plastic golf tees, it's especially true for the garbage that is found in most file systems.*

Student: *Well, you know what I mean! For a long time at least. And even simple things, such as updating a persistent storage device, are complicated, because you have to care what happens if you crash. Recovery, something I had never even thought of when we were virtualizing memory, is now a big deal!*

Professor: *Too true. Updates to persistent storage have always been, and remain, a fun and challenging problem.*

Student: *I also learned about cool things like disk scheduling, and about data protection techniques like RAID and even checksums. That stuff is cool.*

Professor: *I like those topics too. Though, if you really get into it, they can get a little mathematical. Check out some the latest on erasure codes if you want your brain to hurt.*

Student: *I'll get right on that.*

Professor: *(frowns) I think you're being sarcastic. Well, what else did you like?*

Student: *And I also liked all the thought that has gone into building technology-aware systems, like FFS and LFS. Neat stuff! Being disk aware seems cool. But will it matter anymore, with Flash and all the newest, latest technologies?*

Professor: *Good question! And a reminder to get working on that Flash chapter... (scribbles note down to self) ... But yes, even with Flash, all of this stuff is still relevant, amazingly. For example, Flash Translation Layers (FTLs) use log-structuring internally, to improve performance and reliability of Flash-based SSDs. And thinking about locality is always useful. So while the technology may be changing, many of the ideas we have studied will continue to be useful, for a while at least.*

Student: *That's good. I just spent all this time learning it, and I didn't want it to all be for no reason!*

Professor: *Professors wouldn't do that to you, would they?*

I

Flash-based SSDs

After decades of hard-disk drive dominance, a new form of persistent storage device has recently gained significance in the world. Generically referred to as **solid-state storage**, such devices have no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory and processors. However, unlike typical random-access memory (e.g., DRAM), such a **solid-state storage device** (a.k.a., an **SSD**) retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data.

The technology we'll focus on is known as **flash** (more specifically, **NAND-based flash**), which was created by Fujio Masuoka in the 1980s [M+14]. Flash, as we'll see, has some unique properties. For example, to write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**), which can be quite expensive. In addition, writing too often to a page will cause it to **wear out**. These two properties make construction of a flash-based SSD an interesting challenge:

CRUX: HOW TO BUILD A FLASH-BASED SSD

How can we build a flash-based SSD? How can we handle the expensive nature of erasing? How can we build a device that lasts a long time, given that repeated overwrite will wear the device out? Will the march of progress in technology ever cease? Or cease to amaze?

I.1 Storing a Single Bit

Flash chips are designed to store one or more bits in a single transistor; the level of charge trapped within the transistor is mapped to a binary value. In a **single-level cell** (**SLC**) flash, only a single bit is stored within a transistor (i.e., 1 or 0); with a **multi-level cell** (**MLC**) flash, two bits are encoded into different levels of charge, e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels. There is even **triple-level cell** (**TLC**) flash, which encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

TIP: BE CAREFUL WITH TERMINOLOGY

You may have noticed that some terms we have used many times before (blocks, pages) are being used within the context of a flash, but in slightly different ways than before. New terms are not created to make your life harder (although they may be doing just that), but arise because there is no central authority where terminology decisions are made. What is a block to you may be a page to someone else, and vice versa, depending on the context. Your job is simple: to know the appropriate terms within each domain, and use them such that people well-versed in the discipline can understand what you are talking about. It's one of those times where the only solution is simple but sometimes painful: use your memory.

Of course, there are many details as to exactly how such bit-level storage operates, down at the level of device physics. While beyond the scope of this book, you can read more about it on your own [J10].

I.2 From Bits to Banks/Planes

As they say in ancient Greece, storing a single bit (or a few) does not a storage system make. Hence, flash chips are organized into **banks** or **planes** which consist of a large number of cells.

A bank is accessed in two different sized units: **blocks** (sometimes called **erase blocks**), which are typically of size 128 KB or 256 KB, and **pages**, which are a few KB in size (e.g., 4KB). Within each bank there are a large number of blocks; within each block, there are a large number of pages. When thinking about flash, you must remember this new terminology, which is different than the blocks we refer to in disks and RAIDs and the pages we refer to in virtual memory.

Figure I.1 shows an example of a flash plane with blocks and pages; there are three blocks, each containing four pages, in this simple example. We'll see below why we distinguish between blocks and pages; it turns out this distinction is critical for flash operations such as reading and writing, and even more so for the overall performance of the device. The most important (and weird) thing you will learn is that to write to a page within a block, you first have to erase the entire block; this tricky detail makes building a flash-based SSD an interesting and worthwhile challenge, and the subject of the second-half of the chapter.

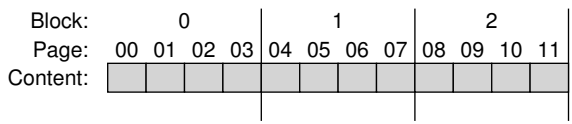


Figure I.1: **A Simple Flash Chip: Pages Within Blocks**

I.3 Basic Flash Operations

Given this flash organization, there are three low-level operations that a flash chip supports. The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write. The details:

- **Read (a page)**: A client of the flash chip can read any page (e.g., 2KB or 4KB), simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast, 10s of microseconds or so, regardless of location on the device, and (more or less) regardless of the location of the previous request (quite unlike a disk). Being able to access any location uniformly quickly means the device is a **random access** device.
- **Erase (a block):** Before writing to a *page* within a flash, the nature of the device requires that you first **erase** the entire *block* the page lies within. Erase, importantly, destroys the contents of the block (by setting each bit to the value 1); therefore, you must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) *before* executing the erase. The erase command is quite expensive, taking a few milliseconds to complete. Once finished, the entire block is reset and each page is ready to be programmed.
- **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1's within a page to 0's, and write the desired contents of a page to the flash. Programming a page is less expensive than erasing a block, but more costly than reading a page, usually taking around 100s of microseconds on modern flash chips.

One way to think about flash chips is that each page has a state associated with it. Pages start in an `INVALID` state. By erasing the block that a page resides within, you set the state of the page (and all pages within that block) to `ERASED`, which resets the content of each page in the block but also (importantly) makes them programmable. When you program a page, its state changes to `VALID`, meaning its contents have been set and can be read. Reads do not affect these states (although you should only read from pages that have been programmed). Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides. Here is an example of states transition after various erase and program operations within a 4-page block:

		`iiii`	*Initial: pages in block are invalid* (`i`)
Erase()	→	`EEEE`	*State of pages in block set to erased* (`E`)
Program(0)	→	`VEEE`	*Program page 0; state set to valid* (`V`)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	`VVEE`	*Program page 1*
Erase()	→	`EEEE`	*Contents erased; all pages programmable*

A Detailed Example

Because the process of writing (i.e., erasing and programming) is so unusual, let's go through a detailed example to make sure it makes sense. In this example, imagine we have the following four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example); each page is `VALID` as each has been previously programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Now say we wish to write to page 0, filling it with new contents. To write any page, we must first erase the entire block. Let's assume we do so, thus leaving the block in this state:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Good news! We could now go ahead and program page 0, for example with the contents `00000011`, overwriting the old page 0 (contents `00011000`) as desired. After doing so, our block looks like this:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

And now the bad news: the previous contents of pages 1, 2, and 3 are all gone! Thus, before overwriting any page *within* a block, we must first move any data we care about to another location (e.g., memory, or elsewhere on the flash). The nature of erase will have a strong impact on how we design flash-based SSDs, as we'll soon learn about.

Summary

To summarize, reading a page is easy: just read the page. Flash chips do this quite well, and quickly; in terms of performance, they offer the potential to greatly exceed the random read performance of modern disk drives, which are slow due to mechanical seek and rotation costs.

Writing a page is trickier; the entire block must first be erased (taking care to first move any data we care about to another location), and then the desired page programmed. Not only is this expensive, but frequent repetitions of this program/erase cycle can lead to the biggest reliability problem flash chips have: **wear out**. When designing a storage system with flash, the performance and reliability of writing is a central focus. We'll soon learn more about how modern SSDs attack these issues, delivering excellent performance and reliability despite these limitations.

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure I.2: **Raw Flash Performance Characteristics**

I.4 Flash Performance And Reliability

Because we're interested in building a storage device out of raw flash chips, it is worthwhile to understand their basic performance characteristics. Figure I.2 presents a rough summary of some numbers found in the popular press [V12]. Therein, the author presents the basic operation latency of reads, programs, and erases across SLC, MLC, and TLC flash, which store 1, 2, and 3 bits of information per cell, respectively.

As we can see from the table, read latencies are quite good, taking just 10s of microseconds to complete. Program latency is higher and more variable, as low as 200 microseconds for SLC, but higher as you pack more bits into each cell; to get good write performance, you will have to make use of multiple flash chips in parallel. Finally, erases are quite expensive, taking a few milliseconds typically. Dealing with this cost is central to modern flash storage design.

Let's now consider reliability of flash chips. Unlike mechanical disks, which can fail for a wide variety of reasons (including the gruesome and quite physical **head crash**, where the drive head actually makes contact with the recording surface), flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The primary concern is **wear out**; when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.

The typical lifetime of a block is currently not well known. Manufacturers rate MLC-based blocks as having a 10,000 P/E (Program/Erase) cycle lifetime; that is, each block can be erased and programmed 10,000 times before failing. SLC-based chips, because they store only a single bit per transistor, are rated with a longer lifetime, usually 100,000 P/E cycles. However, recent research has shown that lifetimes are much longer than expected [BD10].

One other reliability problem within flash chips is known as **disturbance**. When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as **read disturbs** or **program disturbs**, depending on whether the page is being read or programmed, respectively.

TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY
Backwards compatibility is always a concern in layered systems. By defining a stable interface between two systems, one enables innovation on each side of the interface while ensuring continued interoperability. Such an approach has been quite successful in many domains: operating systems have relatively stable APIs for applications, disks provide the same block-based interface to file systems, and each layer in the IP networking stack provides a fixed unchanging interface to the layer above.

Not surprisingly, there can be a downside to such rigidity, as interfaces defined in one generation may not be appropriate in the next. In some cases, it may be useful to think about redesigning the entire system entirely. An excellent example is found in the Sun ZFS file system [B07]; by reconsidering the interaction of file systems and RAID, the creators of ZFS envisioned (and then realized) a more effective integrated whole.

I.5 From Raw Flash to Flash-Based SSDs

Given our basic understanding of flash chips, we now face our next task: how to turn a basic set of flash chips into something that looks like a typical storage device. The standard storage interface is a simple block-based one, where blocks (sectors) of size 512 bytes (or larger) can be read or written, given a block address. The task of the flash-based SSD is to provide that standard block interface atop the raw flash chips inside it.

Internally, an SSD consists of some number of flash chips (for persistent storage). An SSD also contains some amount of volatile (i.e., non-persistent) memory (e.g., SRAM); such memory is useful for caching and buffering of data as well as for mapping tables, which we'll learn about below. Finally, an SSD contains control logic to orchestrate device operation. See Agrawal et. al for details [A+08]; a simplified block diagram is seen in Figure I.3 (page 7).

One of the essential functions of this control logic is to satisfy client reads and writes, turning them into internal flash operations as need be. The **flash translation layer**, or **FTL**, provides exactly this functionality. The FTL takes read and write requests on *logical blocks* (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying *physical blocks* and *physical pages* (that comprise the actual flash device). The FTL should accomplish this task with the goal of delivering excellent performance and high reliability.

Excellent performance, as we'll see, can be realized through a combination of techniques. One key will be to utilize multiple flash chips in **parallel**; although we won't discuss this technique much further, suffice it to say that all modern SSDs use multiple chips internally to obtain higher performance. Another performance goal will be to reduce **write amplification**, which is defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) is-

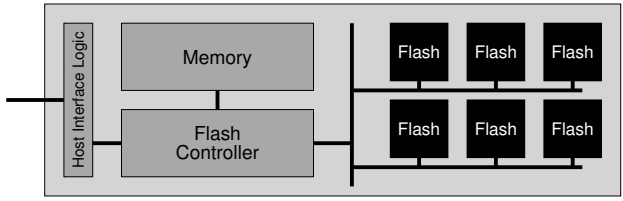


Figure I.3: **A Flash-based SSD: Logical Diagram**

sued by the client to the SSD. As we'll see below, naive approaches to FTL construction will lead to high write amplification and low performance.

High reliability will be achieved through the combination of a few different approaches. One main concern, as discussed above, is **wear out**. If a single block is erased and programmed too often, it will become unusable; as a result, the FTL should try to spread writes across the blocks of the flash as evenly as possible, ensuring that all of the blocks of the device wear out at roughly the same time; doing so is called **wear leveling** and is an essential part of any modern FTL.

Another reliability concern is program disturbance. To minimize such disturbance, FTLs will commonly program pages within an erased block *in order*, from low page to high page. This sequential-programming approach minimizes disturbance and is widely utilized.

I.6 FTL Organization: A Bad Approach

The simplest organization of an FTL would be something we call **direct mapped**. In this approach, a read to logical page N is mapped directly to a read of physical page N. A write to logical page N is more complicated; the FTL first has to read in the entire block that page N is contained within; it then has to erase the block; finally, the FTL programs the old pages as well as the new one.

As you can probably guess, the direct-mapped FTL has many problems, both in terms of performance as well as reliability. The performance problems come on each write: the device has to read in the entire block (costly), erase it (quite costly), and then program it (costly). The end result is severe write amplification (proportional to the number of pages in a block) and as a result, terrible write performance, even slower than typical hard drives with their mechanical seeks and rotational delays.

Even worse is the reliability of this approach. If file system metadata or user file data is repeatedly overwritten, the same block is erased and programmed, over and over, quickly wearing it out and potentially losing data. The direct mapped approach simply gives too much control over wear out to the client workload; if the workload does not spread write load evenly across its logical blocks, the underlying physical blocks containing popular data will quickly wear out. For both reliability and performance reasons, a direct-mapped FTL is a bad idea.

I.7 A Log-Structured FTL

For these reasons, most FTLs today are **log structured**, an idea useful in both storage devices (as we'll see now) and file systems above them (as we'll see in the chapter on **log-structured file systems**). Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block; we call this style of writing **logging**. To allow for subsequent reads of block N, the device keeps a **mapping table** (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.

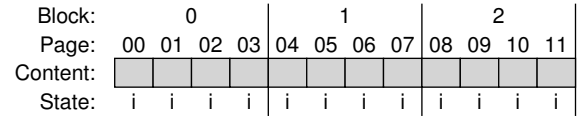
Let's go through an example to make sure we understand how the basic log-based approach works. To the client, the device looks like a typical disk, in which it can read and write 512-byte sectors (or groups of sectors). For simplicity, assume that the client is reading or writing 4-KB sized chunks. Let us further assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages; these parameters are unrealistic (flash blocks usually consist of more pages) but will serve our didactic purposes quite well.

Assume the client issues the following sequence of operations:

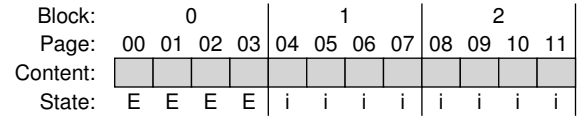
- Write(100) with contents `a1`
- Write(101) with contents `a2`
- Write(2000) with contents `b1`
- Write(2001) with contents `b2`

These **logical block addresses** (e.g., 100) are used by the client of the SSD (e.g., a file system) to remember where information is located.

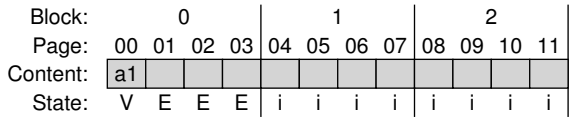
Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data. Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. Here we show the initial state of our SSD, with all pages marked `INVALID` (`i`):



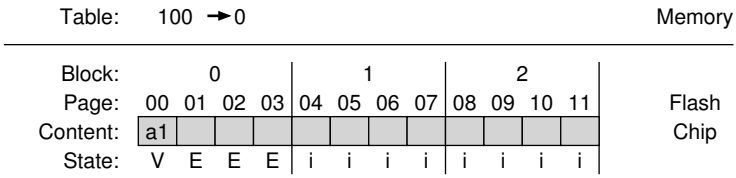
When the first write is received by the SSD (to logical block 100), the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. Because the block is not erased, we cannot write to it yet; the device must first issue an erase command to block 0. Doing so leads to the following state:



Block 0 is now ready to be programmed. Most SSDs will write pages in order (i.e., low to high), reducing reliability problems related to **program disturbance**. The SSD then directs the write of logical block 100 into physical page 0:

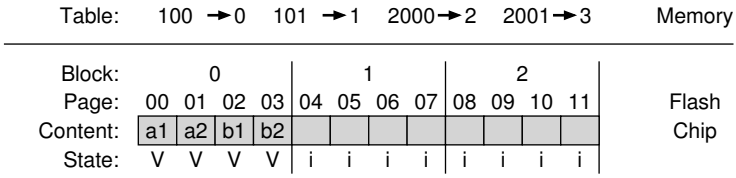


But what if the client wants to *read* logical block 100? How can it find where it is? The SSD must transform a read issued to logical block 100 into a read of physical page 0. To accommodate such functionality, when the FTL writes logical block 100 to physical page 0, it records this fact in an **in-memory mapping table**. We will track the state of this mapping table in the diagrams as well:



Now you can see what happens when the client writes to the SSD. The SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block's contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to **translate** the logical block address presented by the client into the physical page number required to read the data.

Let's now examine the rest of the writes in our example write stream: 101, 2000, and 2001. After writing these blocks, the state of the device is:



The log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether), and greatly enhances reliability. The FTL can now spread writes across all pages, performing what is called **wear leveling** and increasing the lifetime of the device; we'll discuss wear leveling further below.

ASIDE: **FTL MAPPING INFORMATION PERSISTENCE**

You might be wondering: what happens if the device loses power? Does the in-memory mapping table disappear? Clearly, such information cannot truly be lost, because otherwise the device would not function as a persistent storage device. An SSD must have some means of recovering mapping information.

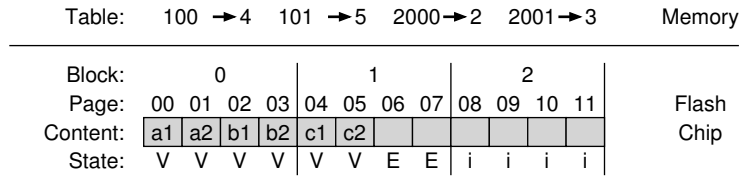
The simplest thing to do is to record some mapping information with each page, in what is called an **out-of-band** (**OOB**) area. When the device loses power and is restarted, it must reconstruct its mapping table by scanning the OOB areas and reconstructing the mapping table in memory. This basic approach has its problems; scanning a large SSD to find all necessary mapping information is slow. To overcome this limitation, some higher-end devices use more complex **logging** and **checkpointing** techniques to speed up recovery; we'll learn more about logging later when we discuss file systems.

Unfortunately, this basic approach to log structuring has some downsides. The first is that overwrites of logical blocks lead to something we call **garbage**, i.e., old versions of data around the drive and taking up space. The device has to periodically perform **garbage collection** (**GC**) to find said blocks and free space for future writes; excessive garbage collection drives up write amplification and lowers performance. The second is high cost of in-memory mapping tables; the larger the device, the more memory such tables need. We now discuss each in turn.

I.8 Garbage Collection

The first cost of any log-structured approach such as this one is that garbage is created, and therefore **garbage collection** (i.e., dead-block reclamation) must be performed. Let's use our continued example to make sense of this. Recall that logical blocks 100, 101, 2000, and 2001 have been written to the device.

Now, let's assume that blocks 100 and 101 are written to again, with contents `c1` and `c2`. The writes are written to the next free pages (in this case, physical pages 4 and 5), and the mapping table is updated accordingly. Note that the device must have first erased block 1 to make such programming possible:



The problem we have now should be obvious: physical pages 0 and 1, although marked `VALID`, have **garbage** in them, i.e., the old versions of blocks 100 and 101. Because of the log-structured nature of the device, overwrites create garbage blocks, which the device must reclaim to provide free space for new writes to take place.

The process of finding garbage blocks (also called **dead blocks**) and reclaiming them for future use is called **garbage collection**, and it is an important component of any modern SSD. The basic process is simple: find a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing.

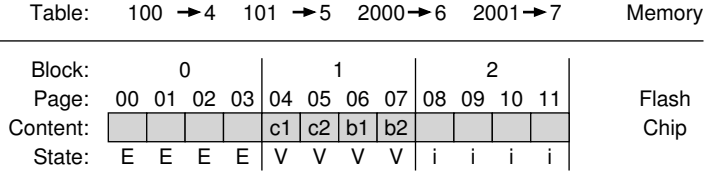
Let's now illustrate with an example. The device decides it wants to reclaim any dead pages within block 0 above. Block 0 has two dead blocks (pages 0 and 1) and two lives blocks (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To do so, the device will:

- Read live data (pages 2 and 3) from block 0
- Write live data to end of the log
- Erase block 0 (freeing it for later usage)

For the garbage collector to function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead. One natural way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

From our example above (before the garbage collection has taken place), block 0 held logical blocks 100, 101, 2000, 2001. By checking the mapping table (which, before garbage collection, contained `100->4, 101->5, 2000->2, 2001->3`), the device can readily determine whether each of the pages within the SSD block holds live information. For example, 2000 and 2001 clearly are still pointed to by the map; 100 and 101 are not and therefore are candidates for garbage collection.

When this garbage collection process is complete in our example, the state of the device is:



As you can see, garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages; in this case, the block can immediately be erased and used for new data, without expensive data migration.

To reduce GC costs, some SSDs **overprovision** the device [A+08]; by adding extra flash capacity, cleaning can be delayed and pushed to the **background**, perhaps done at a time when the device is less busy. Adding more capacity also increases internal bandwidth, which can be used for cleaning and thus not harm perceived bandwidth to the client. Many modern drives overprovision in this manner, one key to achieving excellent overall performance.

I.9 Mapping Table Size

The second cost of log-structuring is the potential for extremely large mapping tables, with one entry for each 4-KB page of the device. With a large 1-TB SSD, for example, a single 4-byte entry per 4-KB page results in 1 GB of memory needed the device, just for these mappings! Thus, this **page-level** FTL scheme is impractical.

Block-Based Mapping

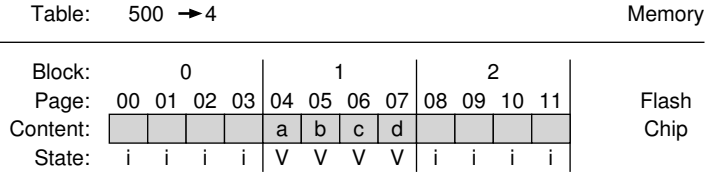
One approach to reduce the costs of mapping is to only keep a pointer per *block* of the device, instead of per page, reducing the amount of mapping information by a factor of $\frac{Size_{block}}{Size_{page}}$. This **block-level** FTL is akin to having bigger page sizes in a virtual memory system; in that case, you use fewer bits for the VPN and have a larger offset in each virtual address.

Unfortunately, using a block-based mapping inside a log-based FTL does not work very well for performance reasons. The biggest problem arises when a "small write" occurs (i.e., one that is less than the size of a physical block). In this case, the FTL must read a large amount of live data from the old block and copy it into a new one (along with the data from the small write). This data copying increases write amplification greatly and thus decreases performance.

To make this issue more clear, let's look at an example. Assume the client previously wrote out logical blocks 2000, 2001, 2002, and 2003 (with contents, `a`, `b`, `c`, `d`), and that they are located within physical block 1 at physical pages 4, 5, 6, and 7. With per-page mappings, the translation table would have to record four mappings for these logical blocks: 2000→4, 2001→5, 2002→6, 2003→7.

If, instead, we use block-level mapping, the FTL only needs to record a single address translation for all of this data. The address mapping, however, is slightly different than our previous examples. Specifically, we think of the logical address space of the device as being chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a chunk number and an offset. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

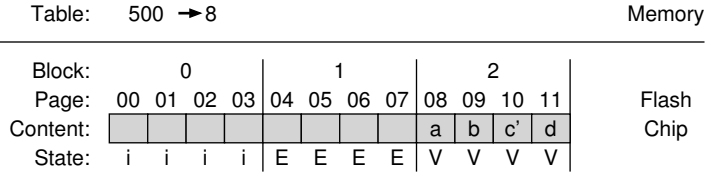
Logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500), and have different offsets (0, 1, 2, and 3, respectively). Thus, with a block-level mapping, the FTL records that chunk 500 maps to block 1 (starting at physical page 4), as shown in this diagram:



In a block-based FTL, reading is easy. First, the FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address. Then, the FTL looks up the chunk-number to physical-page mapping in the table. Finally, the FTL computes the address of the desired flash page by *adding* the offset from the logical address to the physical address of the block.

For example, if the client issues a read to logical address 2002, the device extracts the logical chunk number (500), looks up the translation in the mapping table (finding 4), and adds the offset from the logical address (2) to the translation (4). The resulting physical-page address (6) is where the data is located; the FTL can then issue the read to that physical address and obtain the desired data (c).

But what if the client writes to logical block 2002 (with contents c')? In this case, the FTL must read in 2000, 2001, and 2003, and then write out all four logical blocks in a new location, updating the mapping table accordingly. Block 1 (where the data used to reside) can then be erased and reused, as shown here.



As you can see from this example, while block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often. Thus, a better solution is needed. Can you sense that this is the part of the chapter where we tell you what that solution is? Better yet, can you figure it out yourself, before reading on?

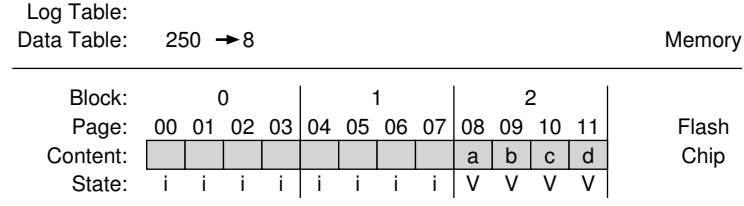
Hybrid Mapping

To enable flexible writing but also reduce mapping costs, many modern FTLs employ a **hybrid mapping** technique. With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps *per-page* mappings for these log blocks.

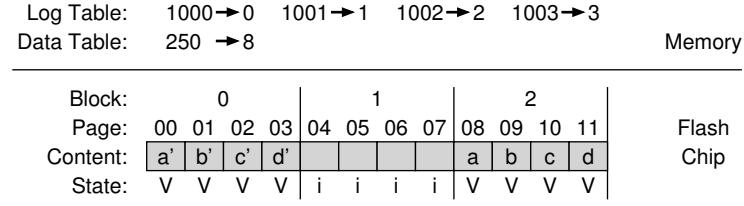
The FTL thus logically has two types of mapping table in its memory: a small set of per-page mappings in what we'll call the *log table*, and a larger set of per-block mappings in the *data table*. When looking for a particular logical block, the FTL will first consult the log table; if the logical block's location is not found there, the FTL will then consult the data table to find its location and then access the requested data.

The key to the hybrid mapping strategy is keeping the number of log blocks small. To keep the number of log blocks small, the FTL has to periodically examine log blocks (which have a pointer per page) and *switch* them into blocks that can be pointed to by only a single block pointer. This switch is accomplished by one of three main techniques, based on the contents of the block [KK+02].

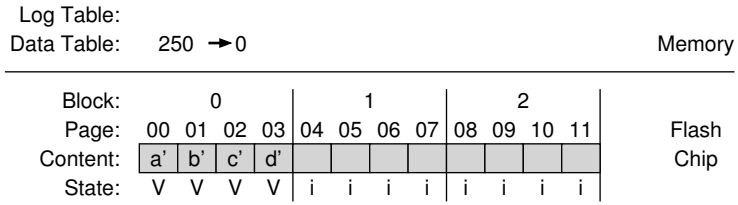
For example, let's say the FTL had previously written out logical pages 1000, 1001, 1002, and 1003, and placed them in physical block 2 (physical pages 8, 9, 10, 11); assume the contents of the writes to 1000, 1001, 1002, and 1003 are `a`, `b`, `c`, and `d`, respectively.



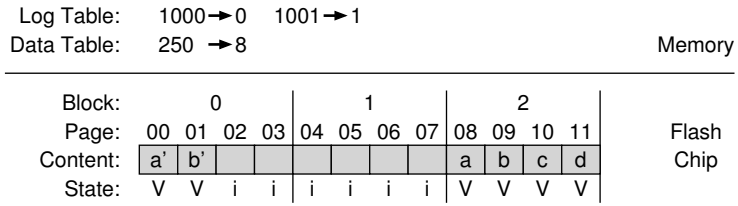
Now assume that the client overwrites each of these pages (with data `a'`, `b'`, `c'`, and `d'`), in the exact same order, in one of the currently available log blocks, say physical block 0 (physical pages 0, 1, 2, and 3). In this case, the FTL will have the following state:



Because these blocks have been written exactly in the same manner as before, the FTL can perform what is known as a **switch merge**. In this case, the log block (0) now becomes the storage location for pages 0, 1, 2, and 3, and is pointed to by a single block pointer; the old block (2) is now erased and used as a log block. In this best case, all the per-page pointers required replaced by a single block pointer.



This switch merge is the best case for a hybrid FTL. Unfortunately, sometimes the FTL is not so lucky. Imagine the case where we have the same initial conditions (logical blocks 0, 1, 2, and 4 stored in physical block 2) but then the client overwrites only logical blocks 0 and 1:



To reunite the other pages of this physical block, and thus be able to refer to them by only a single block pointer, the FTL performs what is called a **partial merge**. In this operation, 2 and 3 are read from block 4, and then appended to the log. The resulting state of the SSD is the same as the switch merge above; however, in this case, the FTL had to perform extra I/O to achieve its goals (in this case, reading logical blocks 2 and 3 from physical pages 18 and 19, and then writing them out to physical pages 22 and 23), thus increasing write amplification.

The final case encountered by the FTL known as a **full merge**, and requires even more work. In this case, the FTL must pull together pages from many other blocks to perform cleaning. For example, imagine that pages 0, 4, 8, and 12 are written to log block A. To switch this log block into a block-mapped page, the FTL must first create a data block containing logical blocks 0, 1, 2, and 3, and thus the FTL must read 1, 2, and 3 from elsewhere and then write out 0, 1, 2, and 3 together. Next, the merge must do the same for logical block 4, finding 5, 6, and 7 and reconciling them into a single data block. The same must be done for logical blocks 8 and 12, and then (finally), the log block A can be freed. Frequent full merges, as is not surprising, can seriously hurt performance [GY+09].

I.10 Wear Leveling

Finally, a related background activity that modern FTLs must implement is **wear leveling**, as introduced above. The basic idea is simple: because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. In this manner, all blocks will wear out at roughly the same time, instead of a few "popular" blocks quickly unusable.

The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well. However, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load.

To remedy this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again. This process of wear leveling increases the write amplification of the SSD, and thus decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate. Many different algorithms exist in the literature [A+08, M+14]; read more if you are interested.

I.11 SSD Performance And Cost

Before closing, let's examine the performance and cost of modern SSDs, to better understand how they will likely be used in persistent storage systems. In both cases, we'll compare to classic hard-disk drives (HDDs), and highlight the biggest differences between the two.

Performance

Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM, in that they are "random access" devices. The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes; while a typical disk drive can only perform a few hundred random I/Os per second, SSDs can do much better. Here, we use some data from modern SSDs to see just how much better SSDs perform; we're particularly interested in how well the FTLs hide the performance issues of the raw chips.

Table I.4 shows some performance data for three different SSDs and one top-of-the-line hard drive; the data was taken from a few different online sources [S13, T15]. The left two columns show random I/O performance, and the right two columns sequential; the first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row shows performance for a **hard disk drive** (or **HDD**), in this case a Seagate high-end drive.

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure I.4: **SSDs And Hard Drives: Performance Comparison**

We can learn a few interesting facts from the table. First, and most dramatic, is the difference in random I/O performance between the SSDs and the lone hard drive. While the SSDs obtain tens or even hundreds of MB/s in random I/Os, this "high performance" hard drive has a peak of just a couple MB/s (in fact, we rounded up to get to 2 MB/s). Second, you can see that in terms of sequential performance, there is much less of a difference; while the SSDs perform better, a hard drive is still a good choice if sequential performance is all you need. Third, you can see that SSD random read performance is not as good as SSD random write performance. The reason for such unexpectedly good random-write performance is due to the log-structured design of many SSDs, which transforms random writes into sequential ones and improves performance. Finally, because SSDs exhibit some performance difference between sequential and random I/Os, many of the techniques we will learn in subsequent chapters about how to build file systems for hard drives are still applicable to SSDs; although the magnitude of difference between sequential and random I/Os is smaller, there is enough of a gap to carefully consider how to design file systems to reduce random I/Os.

Cost

As we saw above, the performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O. So why haven't SSDs completely replaced hard drives as the storage medium of choice? The answer is simple: cost, or more specifically, cost per unit of capacity. Currently [A15], an SSD costs something like $150 for a 250-GB drive; such an SSD costs 60 cents per GB. A typical hard drive costs roughly $50 for 1-TB of storage, which means it costs 5 cents per GB. There is still more than a 10× difference in cost between these two storage media.

These performance and cost differences dictate how large-scale storage systems are built. If performance is the main concern, SSDs are a terrific choice, particularly if random read performance is important. If, on the other hand, you are assembling a large data center and wish to store massive amounts of information, the large cost difference will drive you towards hard drives. Of course, a hybrid approach can make sense – some storage systems are being assembled with both SSDs and hard drives, using a smaller number of SSDs for more popular "hot" data and delivering high performance, while storing the rest of the "colder" (less used) data on hard drives to save on cost. As long as the price gap exists, hard drives are here to stay.

I.12 Summary

Flash-based SSDs are becoming a common presence in laptops, desktops, and servers inside the datacenters that power the world's economy. Thus, you should probably know something about them, right?

Here's the bad news: this chapter (like many in this book) is just the first step in understanding the state of the art. Some places to get some more information about the raw technology include research on actual device performance (such as that by Chen et al. [CK+09] and Grupp et al. [GC+09]), issues in FTL design (including works by Agrawal et al. [A+08], Gupta et al. [GY+09], Huang et al. [H+14], Kim et al. [KK+02], Lee et al. [L+07], and Zhang et al. [Z+12]), and even distributed systems comprised of flash (including Gordon [CG+09] and CORFU [B+12]).

Don't just read academic papers; also read about recent advances in the popular press (e.g., [V12]). Therein you'll learn more practical (but still useful) information, such as Samsung's use of both TLC and SLC cells within the same SSD to maximize performance (SLC can buffer writes quickly) as well as capacity (TLC can store more bits per cell). And this is, as they say, just the tip of the iceberg. Dive in and learn more about this "iceberg" of research on your own, perhaps starting with Ma et al.'s excellent (and recent) survey [M+14]. Be careful though; icebergs can sink even the mightiest of ships [W15].

References

[A+08] "Design Tradeoffs for SSD Performance"
N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy
USENIX '08, San Diego California, June 2008
An excellent overview of what goes into SSD design.

[A15] "Amazon Pricing Study"
Remzi Arpaci-Dusseau
February, 2015
This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!

[B+12] "CORFU: A Shared Log Design for Flash Clusters"
M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis
NSDI '12, San Jose, California, April 2012
A new way to think about designing a high-performace replicated log for clusters using Flash.

[BD10] "Write Endurance in Flash Drives: Measurements and Analysis"
Simona Boboila, Peter Desnoyers
FAST '10, San Jose, California, February 2010
A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100×.

[B07] "ZFS: The Last Word in File Systems"
Jeff Bonwick and Bill Moore
Available: http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf
Was this the last word in file systems? No, but maybe it's close.

[CG+09] "Gordon: Using Flash Memory to Build Fast,
Power-efficient Clusters for Data-intensive Applications"
Adrian M. Caulfield, Laura M. Grupp, Steven Swanson
ASPLOS '09, Washington, D.C., March 2009
Early research on assembling flash into larger-scale clusters; definitely worth a read.

[CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives"
Feng Chen, David A. Koufaty, and Xiaodong Zhang
SIGMETRICS/Performance '09, Seattle, Washington, June 2009
An excellent overview of SSD performance problems circa 2009 (though now a little dated).

[G14] "The SSD Endurance Experiment"
Geoff Gasior
The Tech Report, September 19, 2014
Available: http://techreport.com/review/27062
A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.

[GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications"
L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf
IEEE MICRO '09, New York, New York, December 2009
Another excellent characterization of flash performance.

[GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings"
Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar
ASPLOS '09, Washington, D.C., March 2009
This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.

[H+14] "An Aggressive Worn-out Flash Block Management Scheme
To Alleviate SSD Performance Degradation"
Ping Huang, Guanying Wu, Xubin He, Weijun Xiao
EuroSys '14, 2014
Recent work showing how to really get the most out of worn-out flash blocks; neat!

[J10] "Failure Mechanisms and Models for Semiconductor Devices"
Report JEP122F, November 2010
Available: http://www.jedec.org/sites/default/files/docs/JEP122F.pdf
A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems"
Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho
IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002
One of the earliest proposals to suggest hybrid mappings.

[L+07] "A Log Buffer-Based Flash Translation Layer
Using Fully-Associative Sector Translation"
Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song
ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007
A terrific paper about how to build hybrid log/block mappings.

[M+14] "A Survey of Address Translation Technologies for Flash Memories"
Dongzhe Ma, Jianhua Feng, Guoliang Li
ACM Computing Surveys, Volume 46, Number 3, January 2014
Probably the best recent survey of flash and related technologies.

[S13] "The Seagate 600 and 600 Pro SSD Review"
Anand Lal Shimpi
AnandTech, May 7, 2013
Available: http://www.anandtech.com/show/6935/seagate-600-ssd-review
One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.

[T15] "Performance Charts Hard Drives"
Tom's Hardware, January 2015
Available: http://www.tomshardware.com/charts/enterprise-hdd-charts/
Yet another site with performance data, this time focusing on hard drives.

[V12] "Understanding TLC Flash"
Kristian Vatto
AnandTech, September, 2012
Available: http://www.anandtech.com/show/5067/understanding-tlc-nand
A short description about TLC flash and its characteristics.

[W15] "List of Ships Sunk by Icebergs"
Available: http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs
Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes"
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
FAST '13, San Jose, California, February 2013
Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.