What's the Worst That Could Happen?
An In-Depth Look at File Systems

Authors:
Devin Bedari (204-318-490)
Krishna Vijayakumar (704-294-420)
Sean Kim (204-446-949)

June 19, 2018

1 Introduction

When it comes to writing a file system, we need to be very careful, as we are handling other people's data. Therefore, it is good for us to ask ourselves, what is the worst that can happen, when our code fails.

2 The GZIP Bug

To begin our discussion, let us take a look at a simplified version of the source code that is executed by gzip when it runs:

```
1     write(ofd);
2     if( close(fd) != 0 )
3        error();
4     if( close(ofd) != 0 )
5        error();
6     if( unlink(foo) != 0 )
7        error();
8     exit(0);
```

What would happen, if the gzip code were to crash after the execution of line 6, and just before the execution of line 8? According to a recent crash report, the data on disk is deleted. How is this possible? Is it not true that the unlink call has occurred successfully?

In order to understand why this bug occurs, let us look at the structure of a file system that we have in a typical Linux/BSD type file system.

3 File System Layout

Let us look at the following BSD file system layout, which we will analyze in the sections below:

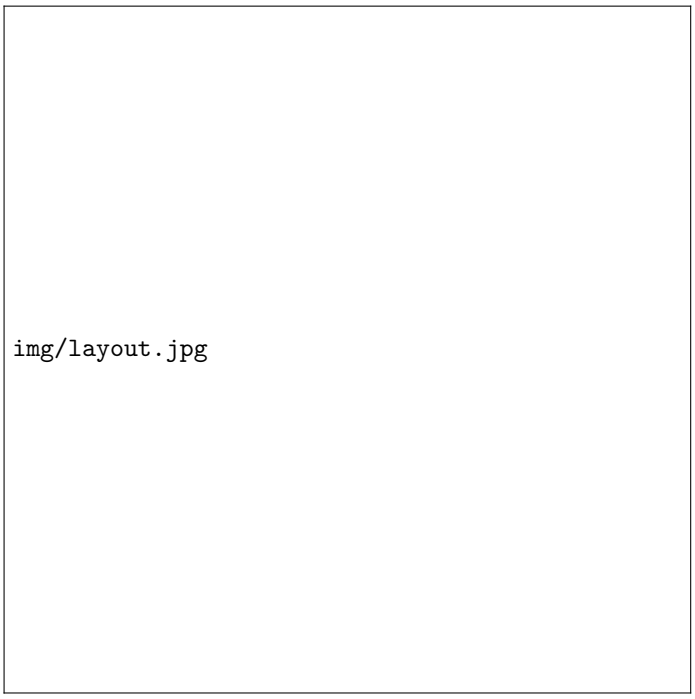

Figure 1: An overview of the layout used by the typical BSD-Linux file system.

3.1 Free Blocks Location

We argue that the aim of a file system is to arrange and allocate storage in a quick and efficient manner. When the file system decides to allocate a file, it needs to look for a free block of memory in secondary storage. In the traditional BSD type file system, there exists an area that keeps track of where our free blocks are. This is called the free block bitmap. this system was not implemented in the original Unix file system, but let us see how it saves us time on computation.

Suppose we did not have this implementation; the simple action of growing a file would require us to modify the inode table to point to a new data block, either directly or indirectly (via some indirect blocks). This can, in theory, be done by searching through the inode table, finding an unused inode, and looking for its corresponding data block through all the aforementioned direct and indirect data block links. From this, we will know, by a process of elimination, which blocks are used and which ones are not. This approach would work, but it is very slow.

We eliminate this bottleneck by using a free block bitmap, which marks off which blocks contain free memory. At any point, if we want to grow a file, we merely need to look through the free block bitmap and find its corresponding free memory block. Next we grow the file. Finally we change the value of the bitmap, to mark off the fact that this memory block is now used.

Even though this free block bitmap represents some redundant data, just marking off a small section of our storage becomes valuable in saving computations to do simple tasks. Note also, that for an 8 KiB data block, we end

up having a ratio of free blocks to data representation of 65536:1, which makes free block bitmaps highly efficient. Another useful feature of having such a data block, is that, because it is so small in size, it can be cached for even better performance. Therefore, using a relatively simple trick, we were able to boost the performance of our file system. Let us use a similar mindset to look at other performance issues that we have with disk storage, and find ways to solve these performance bottlenecks.

3.2 Performance Issues with File Systems

In the previous section, we looked at an ingenious fix to a problem that caused slow performance in performing a relatively simple task. Now, let us take a look at some other performance problems that we may run across with file system handling, and look at ways in which we may solve such problems.

Suppose we want to get a block of data off a hard disk. How long would this take us? The hard drive is a physical circular platter, and the way a sector is magnetized corresponds to 0s and 1s that would be read by the disk seek head. Putting these 0s and 1s together, we build up the file in question for I/O. We will now look at the overhead that is involved in reading data off the hard disk:

- Latency in moving the mechanical arm (seek heads). Take a lot of time (e.g. 10ms)
- Latency in moving the platter to the correct disk sector, called Rotational Latency (e.g. 8ms)
- Latency in copying the data from Disk cache to CPU or RAM (e.g. 0.1ms)

In our example, a total time of 18.1ms is taken for a simple file I/O operation. Assuming that you have a 1GHz single core processor, this time corresponds to 18 million useful instructions wasted. We therefore want to minimize this amount of work done; in fact, we would rather not talk to the secondary storage device at all - we prefer the file system data to just sit in RAM (which is not possible, most of the time). So what can we do? Let us look at the properties that define the effectiveness of a file system:

- **Utilization**: The percentage of time spent in reading or writing data. A low utilization implies that we are not using our I/O devices as much as we like.
- **Through-put**: Rate of request completion. Measured in terms of operations per second. Higher through-put implies a better file system, because more read/write requests are processed, in a period of time.
- **Latency**: The overall delay between the user making an I/O request and the file system's processing of this request. The lower the latency, the faster the file system reads/writes the data, and therefore is quicker when a smaller value.

An interesting thing to note is, when most file systems optimize **Utilization** and **Through-put**, they tend to sacrifice on **Latency**. Consider one of the ways in which file systems write to disk; suppose a series of write requests are give to

the file system. The file system writes them in order that they were issued, but when it comes to reading back the contents of these files, they were written for various different files, and you will have to wonder through them to find them. This will improve your overall through-put, but hurt your latency, because there is a delay in finding and reading from the correct file.

Back to the problem at hand; what ways could we boost our file system performance?

3.3 Solutions to Performance Problems

The idea here is to abstract the disk read/write requests, thereby having the file system handle these requests in such a way that they are more efficient than just having the file system brute force reads/writes to secondary storage. There are several techniques, but we will only go through the most popular ones:

3.3.1 Batching

Batching refers to coalesce small requests into big ones. e.g. c-library putchar()

```
1    while((c = getchar()) != EOF)
2        putchar(c);
```

in this example, the standard I/O library implements putchar() such that the characters read by putchar() are first stored in a buffer of a certain size; when this buffer becomes full, or no other read requests come through from line 1, only then will the data actually be written into memory. Batching is actually a special case of a more general type of optimization, which we will look at next.

3.3.2 Speculation

Speculation is when the file system guesses what your program will do in the near future, and acts on its guess. For example, whenever you are reading some data from a file, but the file system reads more data than you have asked form in anticipation that you will ask for more data from the same file. This is called **pre-fetching**. In the case of writing, we do the inverse of pre-fetching, which is called **dallying** (where you write to file after fetching enough data). So storing the pre-fetched or dallied buffer in the cache makes the file system perform more efficiently. The reason speculation works, and is so effective is because we can rely on the **locality of reference**.

Locality of Reference implies that we are not looking at a completely random event (like betting on a roulette wheel), but instead at something that is mostly predictable (e.g. If we read the first letter in a file, we will most probably read all of the data on the file). There are usually two types of localities:

- **Temporal Locality**: If you access a location i_0 at time t_0, you are very likely to access i at some time $t_0 + \delta t$.
- **Spatial Locality**: If you access location i_0 at time t_0, you are likely to access $i_0 + \epsilon i$ at some time $t_0 + \delta t_0$

This means that access to files are not random; they are highly structured. Locality of reference takes advantage of this, and makes file systems operate faster.

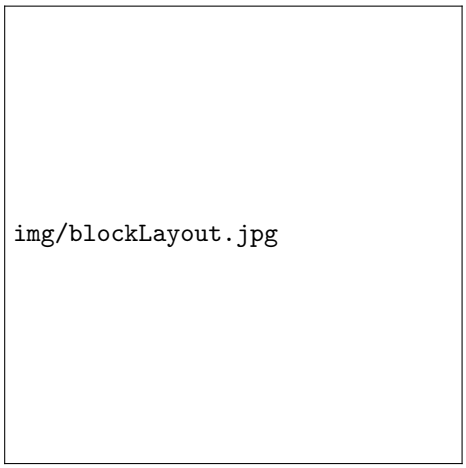

Figure 2: Block layout with cache and request queue shown

According to the locality, we can structure the disk such a way that we can make the most out of reads and writes to disk. Consider the block layer of the file system as represented in Figure 2.

1. **Caching**: We see that using the temporal locality, we can just cache the blocks of data that we have just used. This means that when we attempt to cache the data again,

2. **Algorithmic Request Handling**: We can also propose a simple model of disk access - one in which we model the disk as a big array of blocks, as in Figure 3.

   Given a read/write request array, we see that if the read head of the disk h has to traverse to some location i, then the cost for doing so is $|i - h|$. According to this model, we know that the worst case cost is 1 (as in 100%, which means we have to traverse from the beginning to the end of the disk), and the best case cost is 0 (when we are already at the disk sector that we wish to read from). What is the average cost of a read, assuming random reads? Well if the read head were at either the beginning or the end of the disk, the average read cost would be 1/2; however, when we assume that the read head is at some arbitrary position, things become more complicated. Assume that we are at the middle, then the average cost goes down to 1/4. Therefore the average cost at some arbitrary position must be $0.25 \leq$ Average Cost ≤ 0.5. Let us calculate this upper bracket:

$$\int_0^1 \left(\frac{\int_0^h (h-i)di}{h} + \frac{\int_h^1 (i-h)di}{1-h} \right) dh$$

   Evaluating this integral gives us a value of 1/3 - this is the average value involved in traversing from some arbitrary location of the read head h to some arbitrary memory block i.

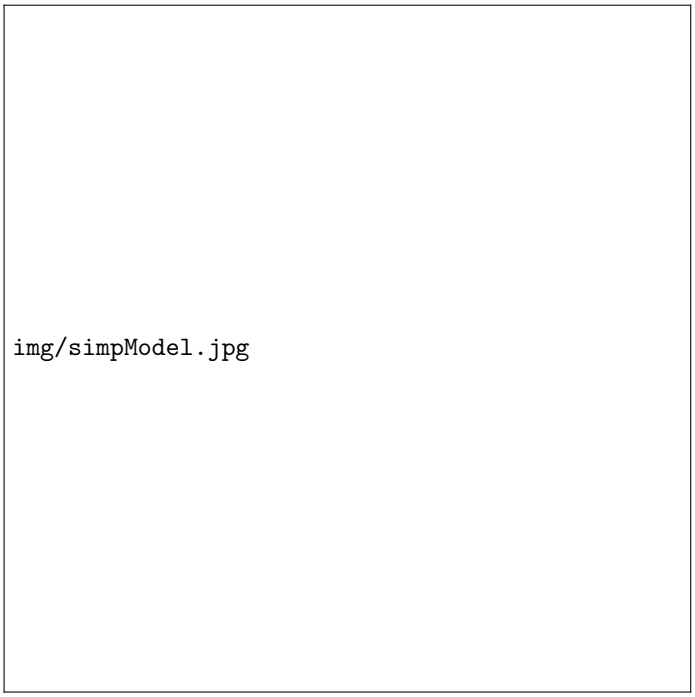

Figure 3: A simplified model of the disk array. Please note that we assume the range of the disk to be a unit (i.e. a fraction of 0.5 means we are halfway through our array in this simplified model). The initial sector of the disk is marked by a 0, and the final sector a 1

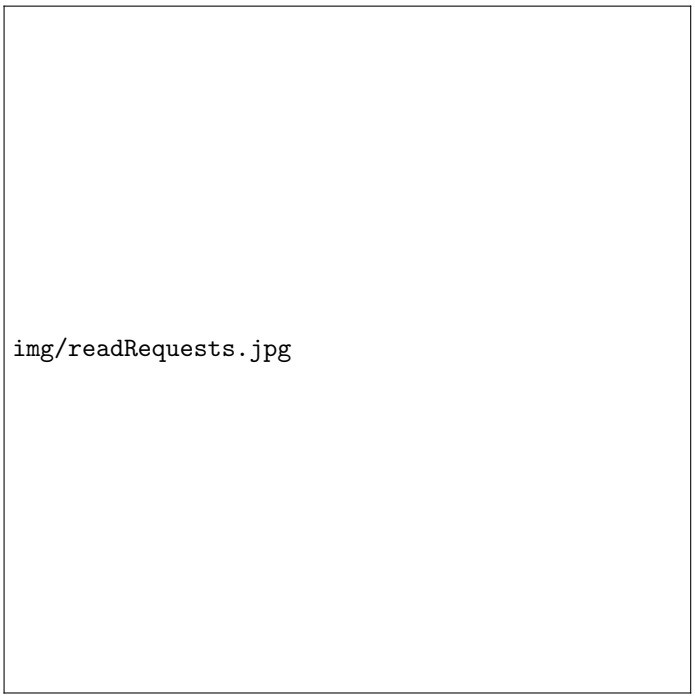

Figure 4: Suppose we were given the following data requests, where each number corresponded to a request's position in the request queue. We see that it is cheaper to process requests in an order of (this is just an example) 4,1,5,3,2 instead of in the order 1,2,3,4,5 because the read head does not waste a lot of time moving around in the secondary storage drive, and therefore maximizes through-put. The sequence that the requests are processed (i.e. 4,1,5,3,2 in our above example) depend on the algorithm that the file system applies to read/write from secondary storage

From all of this we conclude that reads and writes are not arbitrary in nature; they happen in clusters of file blocks.

We will combine this concept with the idea of dallying. We will dally all read/write requests in an array that we will call the request queue. The request queue stores are future read/write requests made by the OS, and the file system will read this data in such a way that it minimizes the distance that the read head has to move (in an out of order manner). It does so by implementing various request queue algorithms (refer to Figure 4). They are described below:

(a) **Shortest Seek Time First (SSTF)**: This algorithm states that if we have a list of read/write requests to process, then process the request that is closest to the disk arm. This algorithm is greedy, and looks only at the "Lowest hanging fruit". We note how something like this can fail. Refer to Figure 4, we see that if requests come in such that these requests are always referencing file blocks near where the read head of the hard drive is, we will see **starvation** occur for

reads that are located far away from the disk arm (read head); for example, in our figure, if requests come in such that they are located hear requests 1 and 5, requests like 2 will only be made after a long period of waiting. For this reason, there are better algorithms out there.

(b) **Elevator Algorithm**: The elevator algorithm states that the read head will move to the end that it is closest to (e.g. to the left end or beginning of the drive) until it reaches the request at the deepest point at this end, and then start processing request in the opposite direction (e.g. to the right or the end of the drive) until it reaches the last request in this direction. This algorithm is good because it does not cause starvation to any of the readers/writers unless all calls are to the same side of the movement of the read head. This is not always the case, so it is not really a problem. Let's model the problems with this algorithm as if we are on the second floor of a 10 story building, and we are waiting for the elevator in order to go up. Suppose the elevator (the read head) is at the 6^{th} floor, and is going up (there are read requests on the 7^{th}, 8^{th}, and 9^{th} file blocks, for example); this means that the person waiting on the 2^{nd} floor had to wait for the elevator to up to the last person that calls the elevator, then wait for the elevator to come down and carry all the people that it encounters on the way down (to compute requests that have just been queued as the seek head was moving up), and only then would we be allowed to use the elevator. This is starvation, but not to the extreme that we saw in SSTF. There is another algorithm that aims to solve this problem.

(c) **Circular Elevator Algorithm**: By this algorithm, we assume that the seek head only moves in one direction, and the end of disk is connected to the beginning of disk. Therefore, there is an avoidance of starvation, but the read head in practice has to move from the end to the beginning of disk; there is a cost in this, but there is fairness, so we are willing to pay this cost. It is also cheaper to move the seek head from the end to the beginning of disk (or vice versa) because the disk accelerates quickly until we are about half way to the beginning, and then decelerates until we reach the beginning. This is cheaper than making many small seeks in the secondary storage device. Again, this is another good algorithm, but are there any more?

(d) **Combining Algorithms**: Combining algorithms is a good way to solve the problems that we encountered in the above steps. An interesting example is combining *Shortest Seek Time First + First Come First Serve (SSTF+FCFS)* The idea here is that we give each request a score (this could be determined by any constant time operation; e.g. Difference between how long a request has been waiting, and the quotient of speed of the seek head and the distance of the request from the read head? Again this is just an example), and according to this score, re-number the requests by this order, and have them evaluated in this order. In a similar way, we can combine algorithms, and find better ways of implementing seeks in a file system.

Using such an algorithmic approach to reading/writing data, makes sure that disk requests are done faster, and with greater through-put. However we see that implementing these algorithms makes evident the relationship that maximizing through-put minimizes read/write latency. We implement algorithms in such a way that we get the best out of our file system, and depending on the system, we may want to maximize through-put or reduce latency. Finding a balance is key, and implementing the correct algorithms is what tweaks this balance according to the user's specifications.

What we have done is formed a layer of abstraction that the OS does not need to worry about. The OS will just issue read and write requests, and the file system will read and write the data whenever its ready. This reduces the stress on the OS (because it gets its reads/writes without having to do any work), and gives us a chance to make the file system more efficient (because the OS does not need to write directly to secondary storage). Right?

4 Problems with Blindly Applying Disk I/O Abstraction

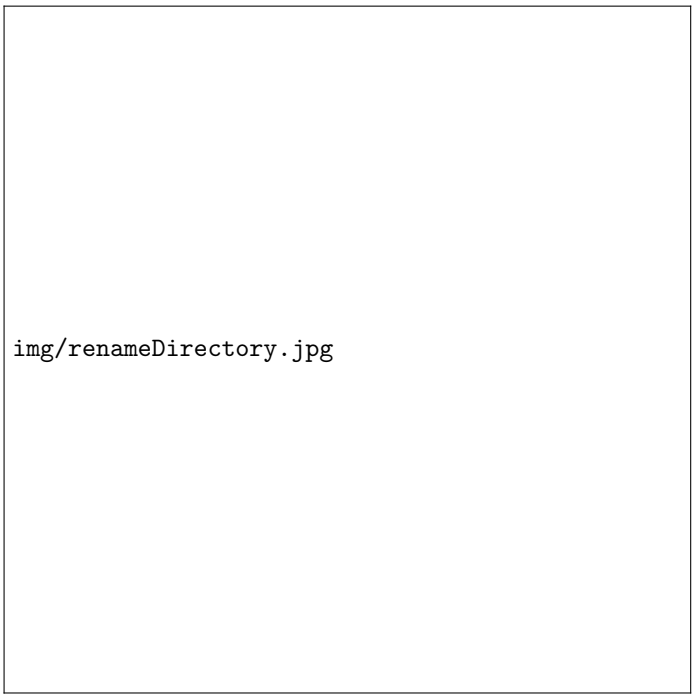

Figure 5: The process in renaming a directory in the case that we run out of memory to rename the block; we need to grab a free block by looking at the free block bitmap and then do the rename operation

Let us look at the problems that we may stumble upon when we perform an operation as simple as renaming a directory (Refer to Figure 5 for the numerical

values we use in the examples in brackets). When we want to rename a directory (directory foo, inode entry states that this is located in memory block 512), we just need to change the block data entries (for block 512) to the newly renamed value (from "foo" to "fuzzy") and the directory name size/length (from 3 to 5), and issue a write request to the request queue. This will work, and is a simple case.

The more interesting case is when the new name's length is bigger than the length that the data block can hold, then we need to get a free block (block 1937), by looking for this free block data in the free block bitmap. Next, we extend over the data from the old directory block (block 512) to the new free block (1937), so that these two new blocks store the directory information. Let us summarize the operations that happen during this system call:

1. Write the new block (1937) with information stored on the old block (512), thereby extending the block data length
2. Change the free block bitmap, to mark the new block (1937) as used
3. Change the inode data, so that the blocks (5121 and 1937) correspond to the directory location
4. Edit the old block (512) to store only the parameters that it has space for and write this to the secondary storage device

All of this is great, however does there need to be an order in which this is processed? Well it does matter, because we are trying to build a persistent data structure; that is, one in which if there are interrupts in the middle of these operations, we still retain the data; we may sacrifice on the system call (rename) failing, but this is not a problem, when compared to losing the user's data.

What happens if we process writing the free data bitmap first? Assuming that there is a crash after this process takes place, we simply have a memory leak, because the new block (1937) was set as used, when we did not actually write anything into this block.

Wait. Wouldn't it be better to write the new block (1937) first? We will be writing a free block! This is fine, because there is nothing important going on here, and the user will not even notice any changes to the file, because it simply looks like the rename operation crashed.

Next we write the free block data, because we are now marking the new block (1937) as used; doing this is great, because if the system crashes here, then the worst case is the memory leak.

So now, we better write the inode table data, because if we crash while writing the new data block, the directory structure in the inode table will not recognize that the new block (1937) is attached to the directory.

Finally, we write the new block data (1937) last. This are the four points that are listed above.

What happens after doing 3, and before doing 4? That is, what happens if we crash after overwriting the inode table, and before extending the directory block? Well, this would mean that we have two hard links for the same directory (both foo and fuzzy). This is allowed in Unix, but is not what the user wanted. At least the data is still there! A crash after 4 is no problem at all.

We should be good, right? No. If we crash after 3 and before 4, we have too many links to the file. (i.e. we will have two hard links to the file, but the link count is 1). This is a disaster, because if after a reboot, if one of those two links were deleted, the system will reclaim the space for the file, the link count will drop to zero, we will have lost the data, and one of the files would point to nothing.

How could we fix this, at some performance cost? We could do 3, and temporarily increase the link count, and then after doing 4, we bring the link count back down to its correct value. This would require us to do 6 block writes instead of 4, but it is worth the security. This is based on a concept that we will called **file system invariants**.

We see that when we are maintaining/writing/using a file system, we have to keep track of what the important properties of that file system are, on disk. This is because all of the reads and writes that we are doing above, attempt to maintain the properties of the file system for all time of the operation of the file system; these assertions that are always true, are called **File System Invariants**.

5 File System Invariants

An invariant implies that at any point in time t_1, if we were to examine the properties of the file system, they would be the same at any futuristic time t_2, as well as any past time t_0.

5.1 Invariants in a BSD-Unix File System

There are some underlying principals regarding file system invariants in a BSD style file system:

1. Every block is used for exactly one purpose. These purposes include a boot block, super block (note in a BSD Type system, the super block is scattered throughout the hard disk), bitmap, inode, and data. We should always know what bit block represents what purpose. We should never mix up something like a free bitmap block, and an inode, because overwriting one array with the data in the other array can cause us big problems.

2. All referenced blocks are appropriately initialized. This means that you do not have some uninitialized garbage sitting inside an inode, because the file system will believe that there is data stored at this garbage. Note, however, that this applies to only referenced blocks; free blocks are not referenced, so we can leave them as garbage. The rule of thumb is that if the block is visible to the user, then they have to be initialized.

3. All referenced data blocks are marked as used in the bitmap. That means that if you can traverse data from a directory, through an inode entry, through the indirect blocks, and find a data block - the bitmap should claim the data block as used.

4. All unreferenced datablocks are marked free in the bitmap.

These are not all of the invariants we may encounter in a file system, but they are a good starting point. When writing file system code, we should always keep these invariants in mind, therefore we will always know whether the code we are writing for this file system is buggy or not.

5.2 Punishment for Violating Invariants

The short answer to the question: What is the punishment for violating the file system invariants, is DISASTER (data is lost).

1. Overwriting two different referenced blocks will cause data to be lost in data blocks, inodes to point at incorrect directories, free block bitmaps to mark files as empty, and allowing them to be overwritten. To summarize, disaster.

2. If we have a reference block that is not appropriately initialized, for example, if we access an unreferenced inode structure, then we will attempt to read and write from the address of garbage locations. This is also a disaster.

3. If you do not mark referenced file blocks in the free block bitmap, you would find free space at a used file block location, and therefore overwrite any data that is contained in that file block. This is also a disaster.

4. Marking unused file blocks as used, means that if we have free space, we wont recognize it as free space, and may not be able to use that free memory block. This is not a total disaster; it is merely a memory leak - still not good.

Finally, we conclude that we should never violate the file system invariants, because the result of breaking these rules result in disasters; by disaster, we imply data loss. This goes against the importance of a file system, which is to preserve data.

5.3 Data Leaks

Therefore if we maintain invariants, our biggest problems during crashes would be data leaks. Is there any way to check for storage leaks on a drive though?

We can not check for leaks using something like opendir(); this is because the file system does not give the user access to the free list. The only way one can access the free list is by allocating some space to a file. Even structures like inodes are only half-visible in the typical BSD-Linux file system; even so, you can not really do much with the inode number.

Instead of using such system calls, we need to bypass the OS, and go straight into the blocks of the file system, because we know exactly which file system we are working with. Therefore we need some standard program, designed for your file system, that will find corrupted data blocks, which is fsck (file system checker).

fsck will basically finds free block that are not actually free, or files that are not linked to any directory (such a file was probably in the process of being removed, so its directory entry went away. The next step was to drop its link count to zero, and recover the used space, except the system crashed before this

could happen). Since fsck is root, any files that it recovers that are not linked to anywhere, will be copied into a special directory entry called lost+found (it will probably name the file by its inode number). Note that fsck marks these files as read only to root, and therefore the system administrator will have to decide what to do with such files. Therefore, we can think of fsck as being a gateway between an imperfect file system and human operators that are "sweeping up" our file debris.

6 Back to GZIP

It is very strange that GZIP would have this bug that we refer to in Section 2. File system developers have spent years writing file system code that was so carefully implemented, that it maintained the files system invariants. Why would file system code like this, allow GZIP to fail in a way that it lost user data?

So our file system code is nice enough to handle the read and write overhead, and it does them in a particular order to make sure that the read/writes are done in an order, such that these invariants are maintained - this code is not perfect, but fsck makes it such that it is pretty damn close. The problem arises when you glue this careful, write ordering code to the code of the block layer. why? Well, because the block layer code is optimized using the seek head algorithms that we described in Section 3.3.2; this code does not process requests in the ordering that the upper layer code has put so much thought into preserving; it wants to process read/writes requests as fast as possible.

If we look at the GNU/Linux file systems that we have today (ext2-ext4), we see that we are able to change some very mundane (and, to the layman, unimportant) settings of the file system. One of the options that we have, is the ordered mount option. This option forces the block data code to process requests in the order that the user wanted them to be processed; this makes our file system resilient to crashes, but makes it process read/write requests at a much slower rate.

Now suppose we run GZIP on a Linux machine that has not set the ordered mount option as true. Looking at the code in Figure 6, we can see that there is a request block that happens at unlink, on some block u (to output a new directory after doing the unlink), as well as another request block that occurs at the initial write, on some block w (to write out the last block for the file). We gave the underlying system a request to do u after w, but because ordering is not conserved, the system may do it in any order. Suppose, it does u first, then the code crashes between lines 6 and 8, then attempting to write out w after removing u will mean that we attempt to write out the compressed file, after deleting the uncompressed file. In fact, if the request list is large, then we can actually end up losing a lot of data. This is the bug.

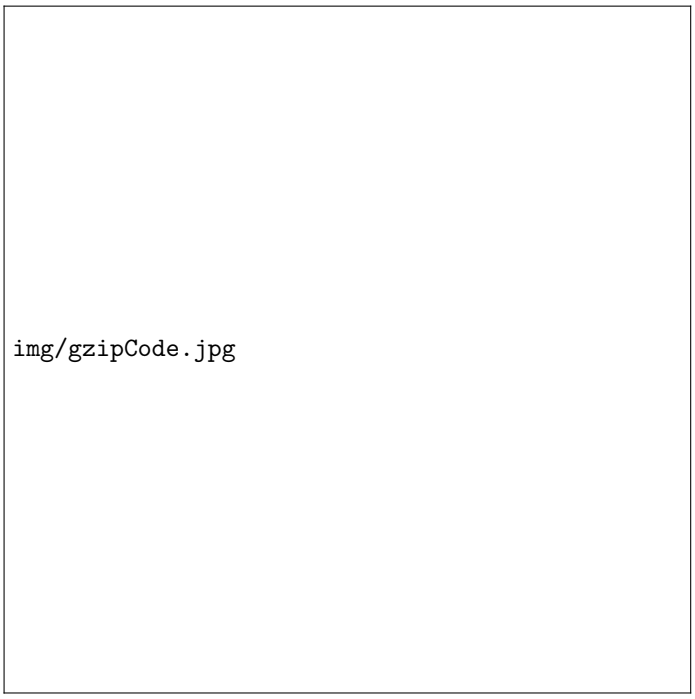

Figure 6: GZIP code that contains a bug

What does this mean? If programs as simple as GZIP have this bug, then more complicated programs are highly susceptible to have this bug as well, regardless of which file system or OS one may be using.

How can we fix this?

```
1     write(ofd);
2     if(fsync(ofd) != 0)    //fix
3        error();
4     if( close(fd) != 0 )
5        error();
6     if( close(ofd) != 0 )
7        error();
8     if( unlink(foo) != 0 )
9        error();
10    exit(0);
```

We introduce a new system call, called fsync() that forces the file system to actually write out the data, without dallying/caching/optimizing. This really hurts the performance of GZIP, but makes it more reliable. Does it have to be that bad? Notice that fsync forces GZIP to wait until all the bits of block w have hit the disk; all we want, however, is the data for block w to hit the disk before block u. Unfortunately, the kernel does not let you do this, and maybe one day, the reader can solve it.