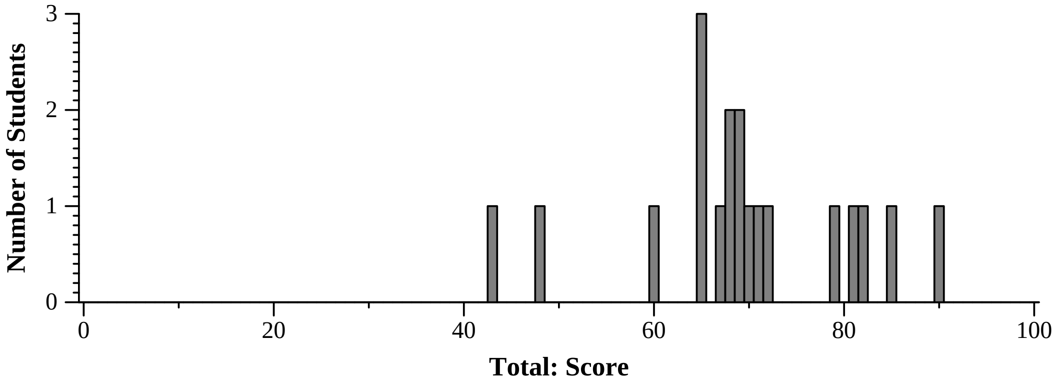
Final Exam – Questions, Answers and Grading

CS460: Operating Systems

13 December, 1995

Jim Plank

This was a very challenging exam, and I was very pleased with the results. Most of y'all demonstrated a good capacity to spout both quantitatively and qualitatively about the material, which was one of my goals for the class. The histogram of final exam grades is:



My interpretation of this is as follows:

- 75 and above: A+
- 70 to 74: A
- 67 to 69: A-/B+
- 61 to 66: B
- 45 to 60: C
- 30 to 45: D

Question 1: 10 Points (10 minutes)

Suppose you have written a distributed file system using remote procedure call (RPC).

Part 1: 4 points

Describe how clients use this mechanism to do remote file operations.

Part 2: 6 points

Describe how the server responds to clients to perform remote file operations. Why is it a good idea to have the server be threaded?

The Answer

Part 1

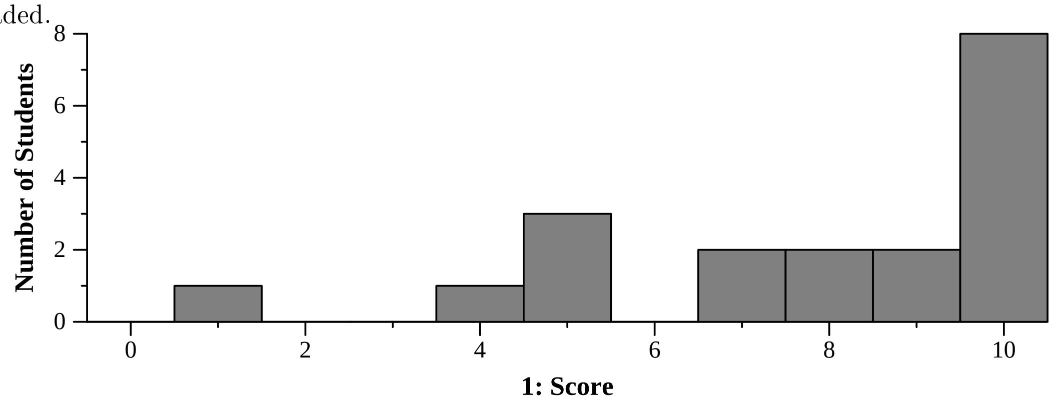
To perform a remote file operation, the client process makes a remote procedure call to the machine and port where the correct server is for that file operation. The remote procedure call looks just like a local procedure call, except its execution path is different. Instead of actually making a procedure call, a "stub" is called. Stubs are generated at compile/link time. The stub bundles up the arguments into a message (this is called "marshalling"), and sends that message to the server's machine. It then waits for a return message from the server which will contain the return contents of the procedure call. When this message comes in and is unbundled, the RPC call returns.

Part 2

The server machine listens for messages on the proper port. These messages are RPC calls. When such a message comes in, the server services the message, (in the case of file I/O, it does the file open/close/read/write/etc) and then sends the return contents back, signaling that the client's RPC call may return. A threaded system is good for this because one thread can spin and listen for messages, and when it gets a message, it forks off a thread that services the message. It then goes back to listening to messages. In this way, the server may service many messages at once. Of course, you could write code to service multiple messages without threads, or you could use multiple processes to do this (as we did in CS360). The thread mechanism is best because it has lower overhead than using processes and presents a very nice programming methodology to the programmer of the server.

Grading

Straightforward. Part 2 was worth 3 points for the description and 3 point for detailing why the server should be threaded.



Question 2: 25 Points (30 minutes)

Behold the following C file:

```
f()
{
  if (i()) {
    ii();
    if (i()) {
      fifi(i());
      if (i()) {
        f();
      }
    }
  }
  fifi();
  if (i()) {
    fifi(i());
  }
}
```

The character usage count is the following:

```
Freq:        5    Char: '{'
Freq:        5    Char: '}'
Freq:        5    Char: ';'
Freq:       12    Char: 'f'
Freq:       16    Char: ')'
Freq:       16    Char: '('
Freq:       17    Char: '^m'
Freq:       18    Char: 'i'
Freq:       62    Char: ' '
```

Part 1: 8 points

Construct a Huffman tree for the file.

Part 2: 4 points

And how many bits are saved over ASCII coding when you encode with your Huffman tree. You can give an expression instead of a number to save time.

Part 3: 7 points

Suppose you were writing a compressor for C files based on Huffman coding. How would you write it so that you don't have to store the Huffman tree with each file?

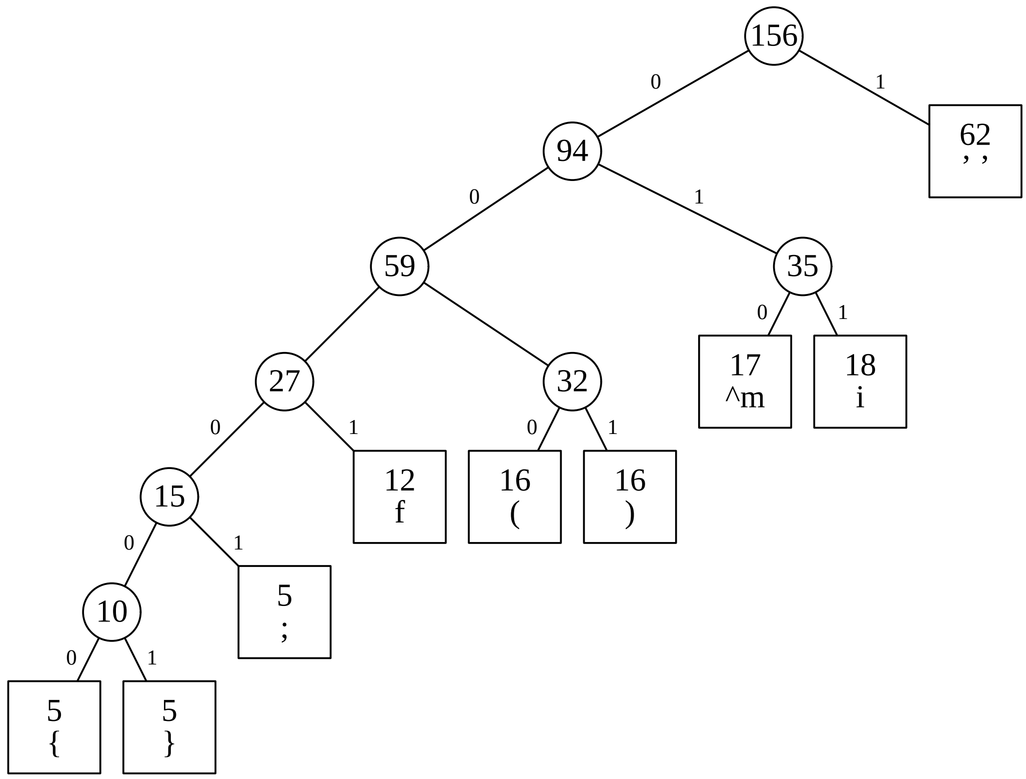
Part 4: 6 points

Give two simple ways that you could optimize your compressor for C files, yet still have it be based on Huffman coding. (When I say optimize here, I mean optimize the compression, not the speed of the compressor).

Answer

Part 1

Several Huffman trees can result from this frequency distribution. Here's one:



Remember the definition of a Huffman tree given in the lecture notes:

1. Determine the number of times each character is used. Let this be denoted by $u(c)$, where c is the character.
2. . Create a binary tree as follows: Start by making a node n for each character c. Let $u(n) = u(c)$. Put each node into a pool p. Now, iterate the following until there is just one node in p:
   - Choose the node n_1 in p such that $u(n_1) <= u(n)$ for all other nodes n in p. Remove n_1 from p.
   - Choose the node n_2 in p such that $u(n_2) <= u(n)$ for all other nodes n in p. Remove n_2 from p.
   - Make a new node n_3. Set the left child of n_3 to n_1, and the right child of n_3 to n_2. Set $u(n_3)$ to be $(u(n_1) + u(n_2))$.
   - Add n_3 to p.
3. The last remaining node in p is the root of the Huffman tree.
4. Label each left child of a node with a zero and each right child with a one.
5. The code for a character c is the label of the path from the root of the Huffman tree to c.

Part 2

Character	Code	Frequency	Savings per character	Total Savings
{	000000	5	2 bits	10 bits
}	000001	5	2 bits	10 bits
;	00001	5	3 bits	15 bits
`f`	0001	12	4 bits	48 bits
`(`	0010	16	4 bits	64 bits
`)`	0011	16	4 bits	64 bits
`^m`	010	17	5 bits	85 bits
`i`	011	18	5 bits	90 bits
`' '`	1	62	7 bits	434 bits
Total				820 bits

With ASCII coding. the file will be 1,248 bits, so Huffman coding results in a savings of 65.7%. You could give valid expressions for this, like "$5*2+5*2+5*3\ldots$", or

$$\sum_c (8 - length(code(c)) * freq(c)$$

Part 3

To write a Huffman compressor for C files, it would be best to take a large collection of C files, generate the character usage frequency for this collection and create the Huffman tree for this usage frequency. This tree can be hardwired into your compressor/decompressor so that you do not have to save the tree with the compressed files. As long as most C programs have roughly the same usage frequencies, your compressor will work well.

Part 4

One easy thing to do is treat keywords in C as characters. In other words, look for words like `int` and `for` and `++`, and include them in your Huffman tree. Since keywords are used often in C, this should help your compression very much.

The second thing to do is use some sort of run-length encoding to deal with spaces. What I would probably do is include a count of spaces after every newline character. This is because in C, most lines start with some number of spaces due to indentation. The count can either be some fixed-length number (3 or 4 bits would probably work will), or you could use the trick of encoding n by using $\log_2 n$ zeros and then encode n with $\log_2 n$ bits.

Grading

Part 1

Any valid Huffman tree received 8 points. Any binary tree with lower frequencies beneath higher frequencies was worth 4 points. Random trees with the elements at the leaves were worth 2 points.

Part 2

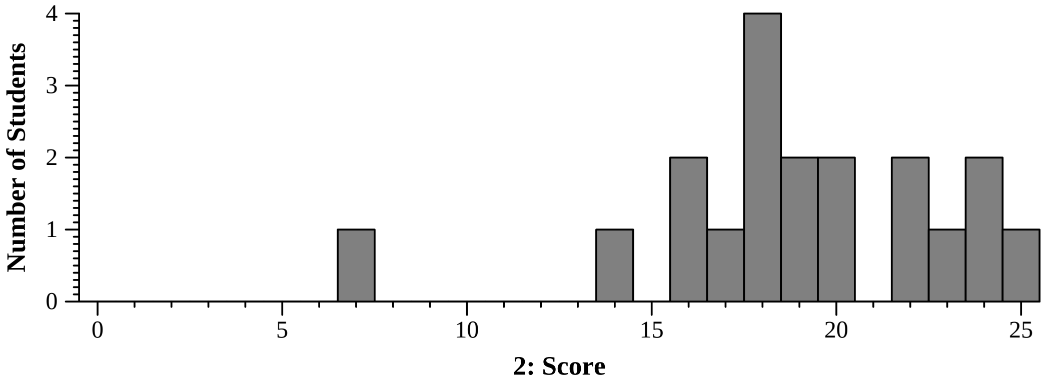
This was straightforward – you could either give the right answer/expression or you couldn't.

Part 3

You got points based on how reasonable your answer was. 2 points were taken off if you did not specify how you would generate your standard Huffman tree for C.

Part 4

Again, points were awarded on how reasonable the ideas were. Each optimization was worth 3 points – optimizations like "compress the compressed file" were not acceptable because they didn't fit within the framework of Huffman coding.



Question 3: 25 Points (30 minutes)

Suppose you have a system with 32-bit pointers and 4096-byte pages that supports two-level paging. All page table entries are 4 bytes. Discuss in detail how the hardware translates virtual addresses to physical addresses. In your discussion, make sure you address the following questions:

- Exactly how is the translation done?
- What do page table entries look like?
- What hardware is necessary so that this system is efficient, and how does this hardware work?
- How much physical memory is taken up by each process?
- What happens on a context switch?

Answer

The easiest way to answer this question was to answer the five questions individually:

- **Exactly how is the translation done?**

  Each address is broken up into three parts:

p_1: 10 bits	p_2: 10 bits	d: 12 bits

  There are two page tables – an "outer" one and an "inner" one. P_1 is an index into the outer page table. This page table contains PTEs that contain pointers to the inner page table, which is paged. In other words, the inner page table is broken up into pages that do not have to be stored contiguously. The PTE at outer[p_1] contains a pointer to the p_1-th page in the inner page table.

  P_2 is an index into the p_1-th page in the inner page table. Since PTE's are 4 bytes, there are 1024 PTE's per page. Thus, p_2 must be exactly 10 bits. The index at p_2 points to the page that holds virtual addresses starting with p_1p_2. The particular byte is the d-th byte in that page. Since pages are 4K, d must be exactly 12 bits. Since d is 12 bits and p_2 is 10 bits, p_1 must be 10 bits.

  PTE's should have several protection bits – e.g. read, write, execute, valid. The memory operation (load, store, load for execution) must agree with these bits in both sets of PTE's (inner and outer). Otherwise the hardware will generate an interrupt. Getting the page from disk is not part of page translation – it is performed in the servicing of the interrupt when the valid bit is not set.

- **What do page table entries look like?**

  Each PTE will have a pointer to the proper page plus several bits – read, write, execute, valid (resident), used, dirty which all relate to different hardware activities. This should all fit into 4 bytes. If physical memory is 2^{32} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the read, write, etc. bits.

  If the required page is not resident, then the valid bit is set to be zero, and the page's location on backing store (if there is one) is stored in the PTE's page frame pointer.

- **What hardware is necessary so that this system is efficient, and how does this hardware work?**

  Without extra hardware, performing a memory operation takes 3 actual memory operations: two page table lookups in addition to the desired memory operation. For efficiency, a "translation lookaside buffer" (TLB) is usually employed. This is an associative memory (meaning a memory that can be indexed by a value) that stores the most recent virtual-to-physical page translations. If a page translation is stored in the TLB, then a memory operation can be performed with very little overhead. If it is not in the TLB, then it will take the aforementioned 3 memory operations. Fortunately, the locality of most programs is such that the majority of memory operations hit the TLB.

Caches are another example of hardware to make the system more efficient, but they don't have too much to do with making the VM system more efficient.

- **How much physical memory is taken up by each process?**

  There will usually be a "page table base register" (PTBR) that points to the base of the outer page table. This table has 1024 entries (since p_1 is 10 bits), and therefore fits into one page. Now, suppose a process uses p pages whose virtual addresses are contiguous. Then p pages are taken for the pages themselves and $\lceil \frac{p}{1024} \rceil$ pages are taken for the inner page table. This yields a total of $(p + \lceil \frac{p}{1024} \rceil + 1)$ total pages. If there are n segments of contiguous pages, then for each segment, if the segment consists of p_s pages, then the segment contributes $(p_s + \lceil \frac{p_s}{1024} \rceil)$ pages to the system. Thus, the entire memory usage is:

$$1 + \sum_{s=1}^{n} (p_s + \left\lceil \frac{p_s}{1024} \right\rceil) \text{ pages}$$

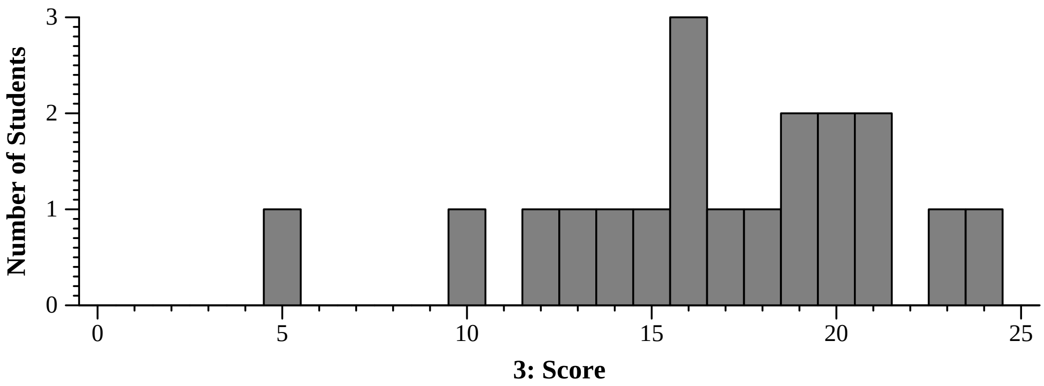
  Since the inner page table is paged, only the outer page table really needs to be resident in memory at all times. All the other pages can be faulted in from stable storage.

- **What happens on a context switch?**

  On a context switch, the PTBR of the outgoing process must be stored in that process's PCB, and the PTBR of the new process should be loaded. If there is a TLB, then it needs to be invalidated so that virtual-to-physical mappings from the old process are not used for the new process.

Grading

- Exactly how is the translation done: 8 points. If you didn't mention protection interrupts anywhere, you lost 1 point here. If you didn't break up the virtual address into bits or did it incorrectly, you lost 2 points.
- What do page table entries look like: 5 points. If you didn't mention protection bits, then you lost 2 points.
- What hardware is necessary so that this system is efficient, and how does this hardware work: 5 points
- How much physical memory is taken up by each process: 4 points. You didn't need to be as detailed as I was with the physical memory calculation, but you needed to say something quantitative and reasonable. If you said that the inner page table takes 4M (2^10 * 4K), then you lost a point, because the existence of the outer page table enables you do only have inner page table pages for those pages that are part of the process's virtual address space that is either resident or on backing store.
- What happens on a context switch: 3 points



Question 4: 40 Points (50 minutes)

A *log-structured file system (LFS)* is one which works as follows:

- A large area of memory is allocated in the operating system – it is called the *write buffer*.
- Whenever a file is written to, the write is performed in the write buffer.
- When the write buffer is full, the entire buffer is written to contiguous disk blocks.

This is straightforward when you are creating and appending to a file. However, when you are overwriting a file block, it is more subtle. What happens is that you read the old values from the disk block into the write buffer, the writes get performed in the write buffer, and eventually the writes go to disk when the write buffer is written to disk. Of course, this means that they go to a *different* disk block than before. The old disk block for that file block is deemed to be garbage. Every now and then, the operating system executes a "garbage collector process," which takes large, contiguous blocks of the disk and compacts them.

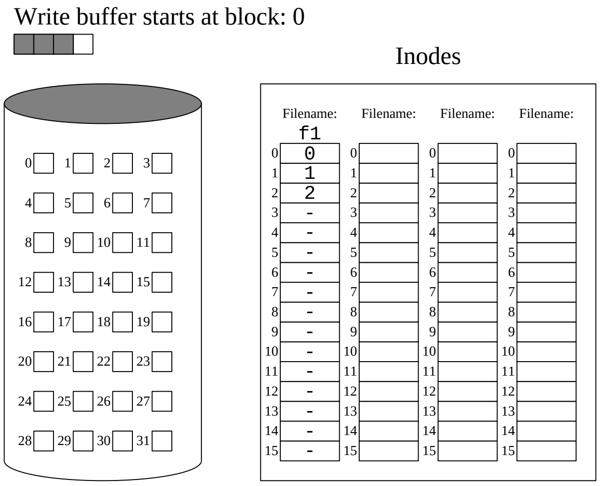
Whenever a file is deleted, its blocks on disk become garbage, and its blocks in the write buffer are freed for reuse.

Unix inodes are used to map files to disk. Inodes have the following structure:

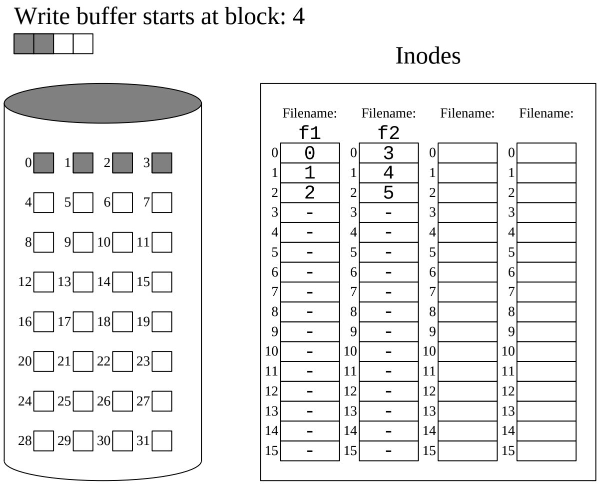
- The first 12 pointers in the inode point directly to the first 12 file blocks.
- The next pointer is to a single indirect block.
- The next pointer is to a double indirect block.
- The next pointer is to a triple indirect block.

Suppose that disk blocks are 1K and that the write buffer is 4K. The following pictures depicts the system after the following events: Free blocks are white. Used blocks are gray. Garbage blocks are black.

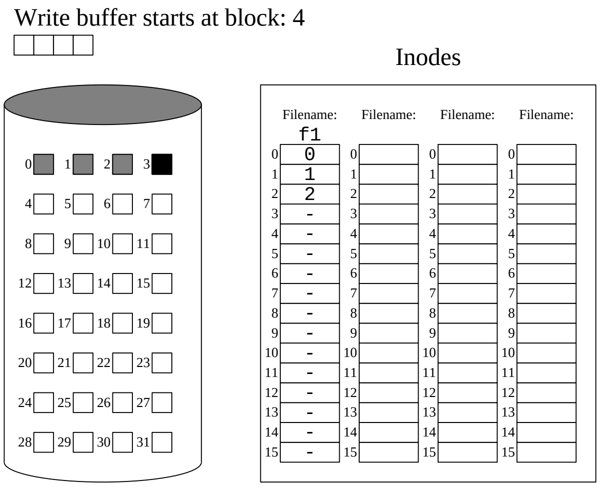
1. Process 1 creates the file "f1" and writes blocks 0 through 2.



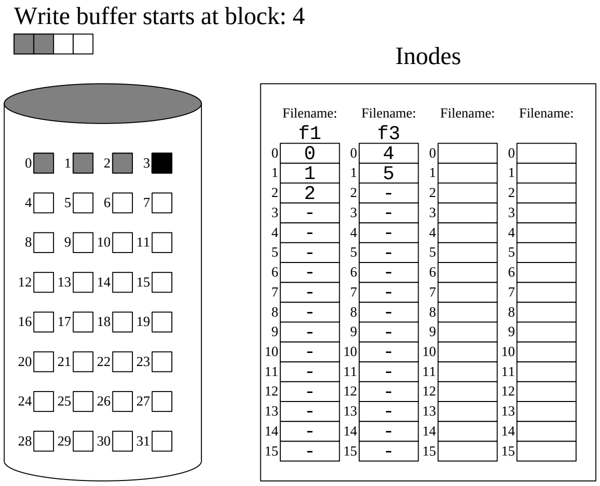
2. Process 2 creates the file "f2" and writes blocks 0 through 2.



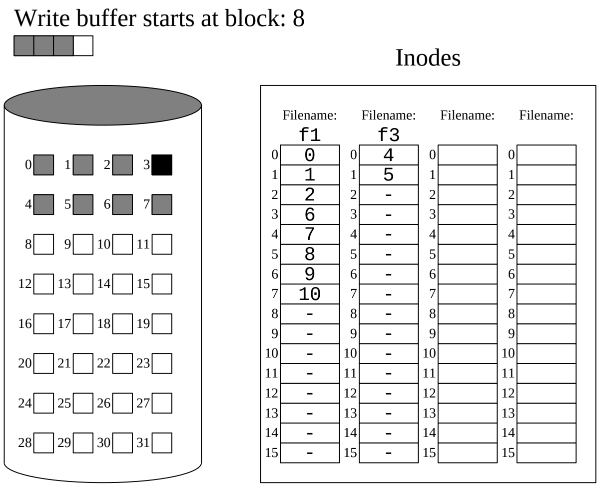
3. Process 2 deletes the file "f2".



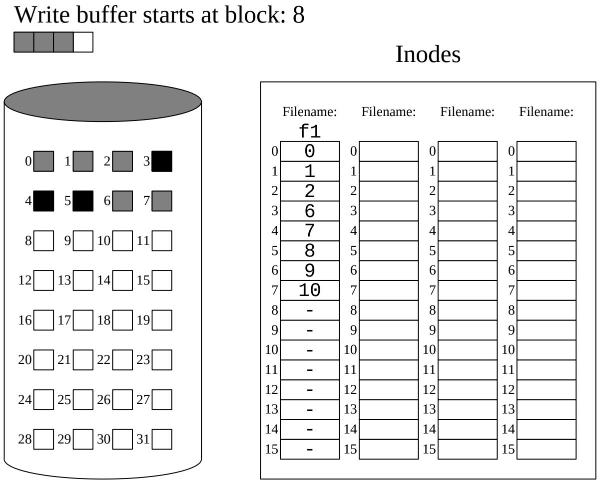
4. Process 3 opens the file "f3" and writes blocks 0 and 1.



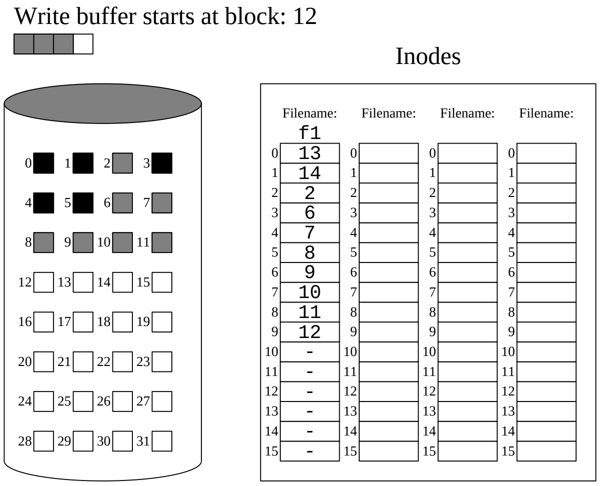
5. Process 1 writes blocks 3 through 7 of "f1" and closes it.



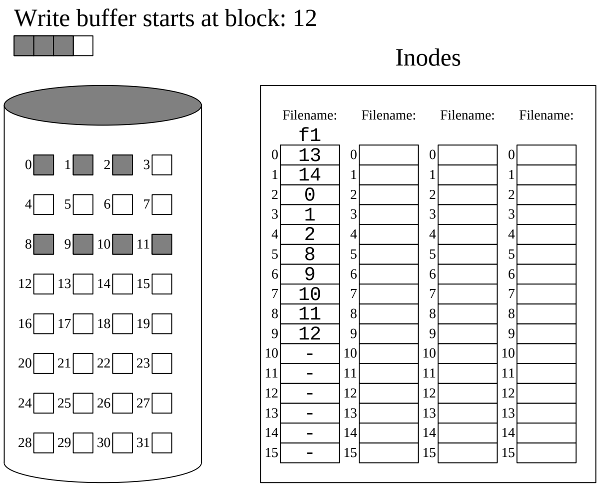
6. Process 3 deletes the file "f3".



7. Process 4 opens "f1", writes blocks 8 and 9, and overwrites blocks 0 and 1.



8. The garbage collector process runs and compacts disk blocks zero through 7



The following questions relate to log-structured file systems as described above:

Part 1: 6 points

Using the sheets at the end of this paper, show what the state of the system looks like after each of the the following events happen:

- 8. Process 4 overwrites blocks 0, 1, 3, and 4.
- 9. Process 5 creates the file `"f4"` and writes blocks 0 through 7. Then process 4 overwrites block 0 again.
- 10. Process 5 deletes the file `"f4"` and opens the file `"f5"` and writes blocks 0 through 5. Then process 4 overwrites block 0 again.

Part 2: 6 points

We'll define a "normal" file system as follows: Inodes are used to map files to disk. There is no write buffer. A standard free list is used to allocate disk blocks, and when a block is freed, it goes on the end of the free list. When a file overwrites an existing block, the disk block corresponding to that file block is overwritten. Show the state of the system after events 2, 4, 6 and 10 above on a normal file system. Use the sheets at the end of the test to hand this in.

Part 3: 18 points

There are two important things going on in a LFS. One is the use of the write buffer, and the other is how the blocks are written to disk. For most systems, the following statements are true:

- Most files are small.
- Many files are deleted very soon after they are created.
- Most files are written sequentially.

Given these characteristics, explain why a LFS with a *very* large write buffer (on the order of megabytes) should perform much better than a normal file system as far as disk writes are concerned. There are several reasons for this – try to get them all.

Part 4: 5 points

In most systems, the majority of file reads are either from a small set of files, or from files that have been recently written. Given this, explain how to improve the performance of an LFS.

Part 5: 5 points

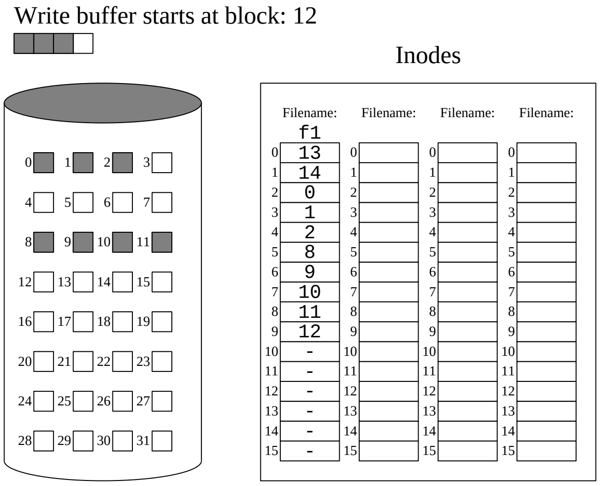
What kind of disk scheduling would be best for an LFS-based system? Why?

Answer

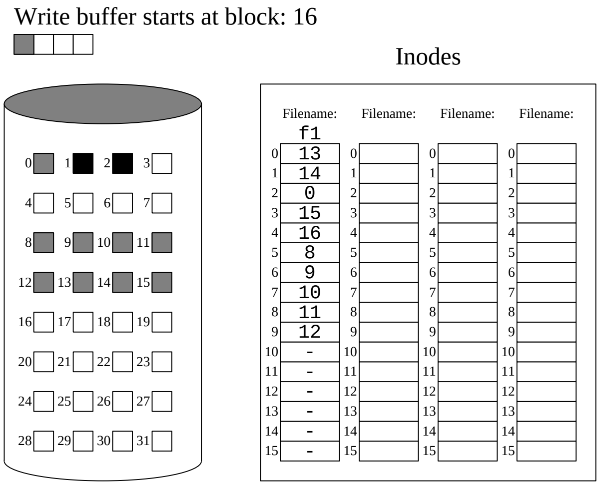
Part 1

Free blocks are white. Used blocks are gray. Garbage blocks are black.

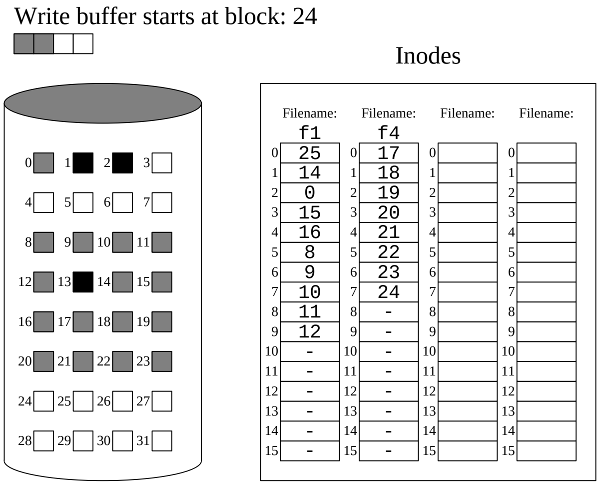
- Starting after the garbage-collection step:



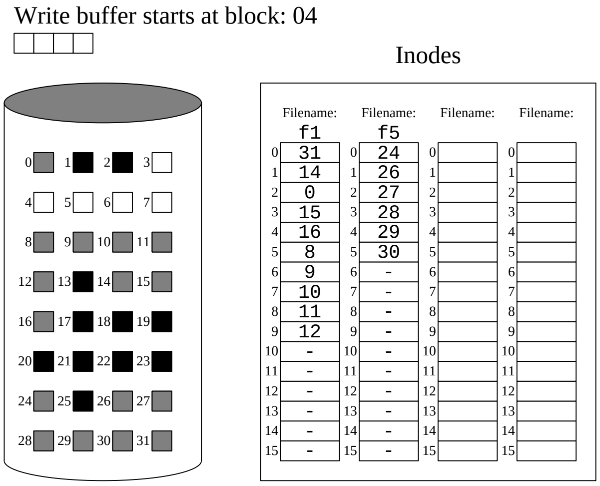
- 8. *Process 4 overwrites blocks 0, 1, 3, and 4.* Since blocks 0 and 1 are already in the write buffer, they can be overwritten there. Blocks 3 and 4 get written to blocks 15 and 16



- 9. *Process 5 creates the file* "f4" *and writes blocks 0 through 7. Then process 4 overwrites block 0 again.* The file "f4" gets blocks 17 through 24, and block 0 of file "f1" is now stored in block 25, as block 13 becomes garbage.



- 10. *Process 5 deletes the file* `"f4"` *and opens the file* `"f5"` *and writes blocks 0 through 5. Then process 4 overwrites block 0 again.* First, blocks 17-24 become garbage. Since one of those blocks (24) is in the write buffer, it is available immediately for reuse. `"f5"` gets blocks 24 and 26 through 30. Next, block 25 becomes garbage, and block 0 of file `"f1"` is now stored in block 31. The write buffer, being full, gets flushed, and now starts caching from block 4. (You could equivalently have it cache from block 3 too).

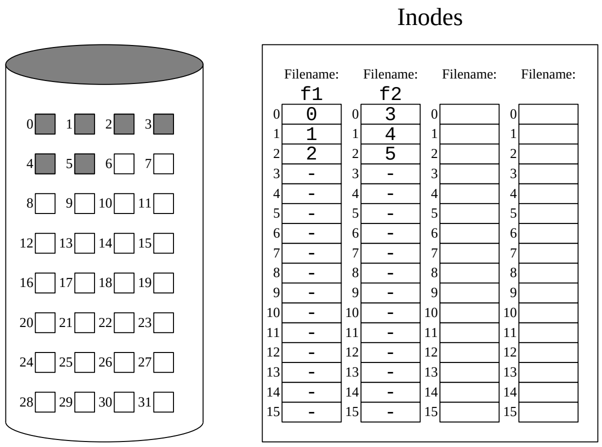


You didn't have to use the same algorithm for finding free blocks as I did (i.e. you could have used blocks 3-6 or 4-7 sooner).

Part 2

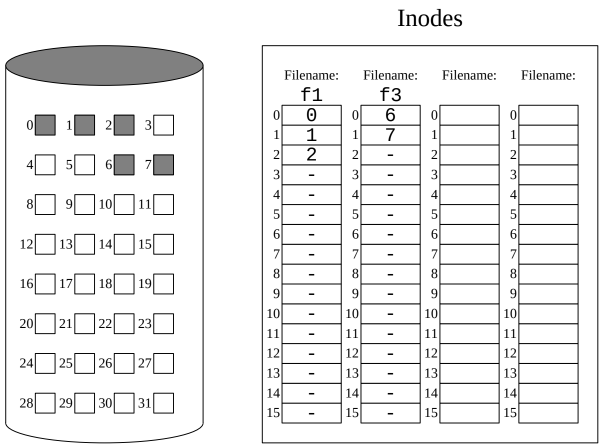
1. *Process 1 creates the file* `"f1"` *and writes blocks 0 through 2.*

2. *Process 2 creates the file* "f2" *and writes blocks 0 through 2.* Straightforward.



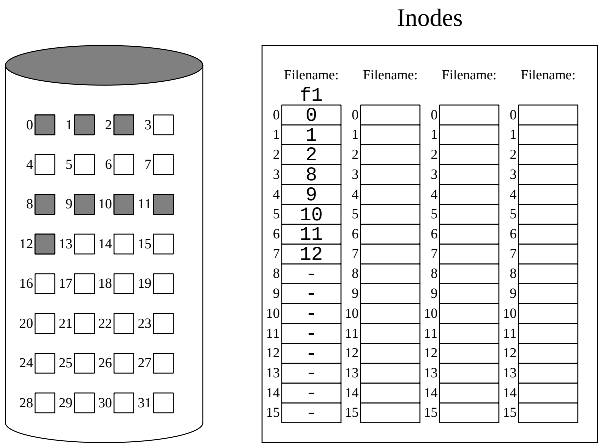
3. *Process 2 deletes the file* "f2". Blocks 3, 4, and 5 are put at the end of the freelist (I'm not drawing the freelist in these pictures).

4. *Process 3 opens the file* "f3" *and writes blocks 0 and 1.* These go in disk blocks 6 and 7.

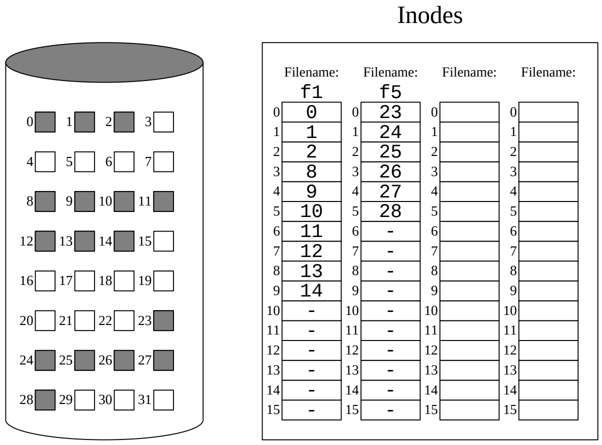


5. *Process 1 writes blocks 3 through 7 of* "f1" *and closes it.* These go in disk blocks 8 through 12.

6. *Process 3 deletes the file* "f3". Blocks 6 and 7 now go at the end of the freelist.



7. *Process 4 opens* "f1"*, writes blocks 8 and 9, and overwrites blocks 0 and 1.* File blocks 8 and 9 go into disk blocks 13 and 14. File blocks 0 and 1 get overwritten in place.
8. *Process 4 overwrites blocks 0, 1, 3, and 4.* This overwrites disk blocks 0, 1, 2, and 8.
9. *Process 5 creates the file* "f4" *and writes blocks 0 through 7. Then process 4 overwrites block 0 again.* "f4" gets disk blocks 15 through 22. Disk block 0 gets overwritten.
10. *Process 5 deletes the file* "f4" *and opens the file* "f5" *and writes blocks 0 through 5. Then process 4 overwrites block 0 again.* Disk blocks 15 through 22 get freed, and disk blocks 23 through 28 get allocated to "f5". Again, disk block 0 gets overwritten.



Part 3

There are three important conclusions that you can draw about LFS performance from this information.

1. If files are small and deleted soon after they are created, then it is likely that these files will be written to and deleted from the write buffer before the buffer is flushed to disk. This means that they are done with no disk operations, which will drastically improve the performance of the system.
2. In a normal file system, numerous small files that get deleted will lead to many individual random file blocks in the free list eventually. This means that each successive write to a new dick block will go

to a random disk block, resulting in a lot of disk arm movement. In contrast, disk writes in a LFS are performed in huge number of contiguous blocks. This virtually eliminates disk arm movement as a source of overhead in disk operations.

3. The overhead of garbage collection has the potential to outweigh the performance gains achieved by doing all these contiguous writes. Since most files are written sequentially and most deleted files will concern blocks in the write buffer, there number of garbage blocks should be very small. This means that there should not be much overhead due to garbage collection.

Part 4

To improve the performance of reads, you can also have a very large read buffer. Whenever you read a file block, if that block is not in the write buffer, then it gets read into the read buffer. You can manage the read buffer in a LRU fashion, perhaps with a readahead heuristic to improve sequential reads. Since most reads are from frequently used files or from recently written files, most blocks that you will want to read will be either in the read buffer or in the write buffer. This means that buffering works: a high percentage of reads will need no disk operations.

Part 5

If the buffers are large enough, disk operations will be very sparse, meaning that FIFO scheduling will do very well. FIFO will work fine with flushing the write buffer because that requires few disk arm movements in any algorithm. If there are still a considerable number of disk operations, then any of SCAN, C-SCAN, LOOK or C-LOOK will work to minimize the number of disk arm movements.

Grading

- **Part 1**: 6 points. 2 for each one. If you made a mistake in an early event, then you were ok for subsequent events if they are performed correctly with respect to your earlier event. I didn't care how you organized your free list.
- **Part 2**: 6 points. Same thing. You lost 2 points if you did not put free blocks at the end of your free list.
- **Part 3**: 18 points – 6 for each major observation. If you said something interesting other than the three major observations, then you got a point or two.
- **Part 4**: 5 points – you had to explain why read buffering worked in this situation.
- **Part 5**: 5 points – you had to explain that FIFO was good because the buffering should limit the number of writes. 1 point was taken off if you didn't deal with the case when there were a lot of disk operations to be done.