Contents

1 File System Implementation **2**

1.1 The Way To Think . . . 2
1.1.1 data structures of the file system. . . . 2
1.1.2 access methods. . . . 3
1.2 Overall Organization . . . 3
1.2.1 divide the disk into blocks; . . . 3
1.2.2 our view of the disk partition . . . 3
1.2.3 what we need to store in these blocks to build a file system. . . . 3
1.2.3.1 user data. . . . 3
1.2.3.2 metadata, . . . 4
1.2.3.3 inodes . . . 4
1.2.3.4 allocation structures . . . 4
1.2.3.5 superblock, . . . 4
1.3 The Inode . . . 4
1.3.1 The name . . . 4
1.3.2 Each inode . . . 5
1.3.3 To read inode number 32, . . . 5
1.3.4 the sector address sector of the inode block can be calculated as follows: . . . 5
1.3.5 Inside each inode . . . 5
1.3.6 The Multi-Level Index . . . 5
1.3.6.1 One common idea . . . 5
1.3.6.2 an inode may have . . . 5
1.3.6.3 support even larger files. . . . 6
1.3.6.4 this imbalanced tree is referred to as . . . 6
1.3.6.5 why use an imbalanced tree like this? . . . 6
1.4 Directory Organization . . . 6
1.4.1 a directory . . . 6
1.4.2 The on-disk data for dir might look like: . . . 6
1.4.2.1 each entry has . . . 6
1.4.3 file systems treat directories as a special type of file. . . . 6
1.5 Free Space Management . . . 7
1.5.1 when we create a file, . . . 7
1.5.1.1 A similar set of activities . . . 7
1.5.2 Some other considerations might also come into play . . . 7
1.6 Access Paths: Reading and Writing . . . 7
1.6.1 For the following examples, . . . 7
1.6.2 Reading A File From Disk . . . 7
1.6.2.1 you issue an open("/foo/bar", O RDONLY) call, . . . 7
1.6.2.2 All traversals . . . 7
1.6.2.3 the root inode number . . . 8

1.6.2.4 The next step . . . 8
1.6.2.5 In this example, . . . 8
1.6.2.6 The final step of open() . . . 8
1.6.2.7 Once open, . . . 8
1.6.2.8 The read . . . 8
1.6.2.9 At some point, the file will be closed. . . . 8
1.6.3 ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES 8
1.6.4 Figure 40.3 . . . 9
1.6.5 the amount of I/O generated by the open . . . 9
1.6.6 Writing A File To Disk . . . 9
1.6.6.1 process. . . . 9
1.6.6.2 writing to the file may also . . . 9
1.6.6.3 When writing out a new file, . . . 10
1.6.6.4 each write to a file logically generates five I/Os: . . . 10
1.6.6.5 To create a file, . . . 10
1.6.7 Figure 40.4 . . . 11
1.7 Caching and Buffering . . . 11
1.7.1 most file systems . . . 11
1.7.2 fixed-size cache . . . 12
1.7.2.1 This fixed-size cache . . . 12
1.7.3 dynamic partitioning approach. . . . 12
1.7.4 the file open example with caching. . . . 12
1.7.5 TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING . 12
1.7.5.1 The static approach . . . 12
1.7.5.2 The dynamic approach . . . 12
1.7.6 effect of caching on writes. . . . 12
1.7.6.1 write buffering . . . 12
1.7.6.2 for example, . . . 12
1.7.6.3 by buffering a number of writes . . . 13
1.7.6.4 some writes are avoided altogether . . . 13
1.7.6.5 trade-off: . . . 13
1.7.6.6 Some applications (such as databases) don't enjoy this trade-off. . . . 13

1 File System Implementation

1.1 The Way To Think

1.1.1 data structures of the file system.

what types of on-disk structures are utilized by the file system to organize its data and metadata?

1.1.2 access methods.

How does it map the calls made by a process, such as open(), read(), write(), etc., onto its structures?

Which structures are read during the execution of a particular system call?

Which are written?

How efficiently are all of these steps performed?

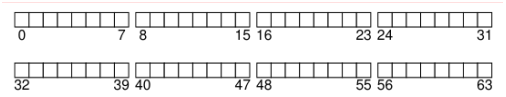
1.2 Overall Organization

1.2.1 divide the disk into blocks;

simple file systems use just one block size, Let's choose a commonly-used size of 4 KB.

1.2.2 our view of the disk partition

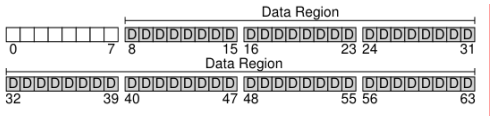
a series of blocks, each of size 4 KB. The blocks are addressed from 0 to N − 1, in a partition of size N 4-KB blocks.



1.2.3 what we need to store in these blocks to build a file system.

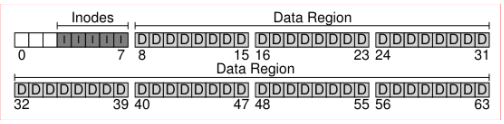
1.2.3.1 user data.

Let's call the region of the disk we use for user data the **data region,**



1.2.3.2 metadata,

tracks things like which data blocks comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information. Let's call this portion of the disk the **inode table,** which simply holds an array of on-disk inodes.



1.2.3.3 inodes

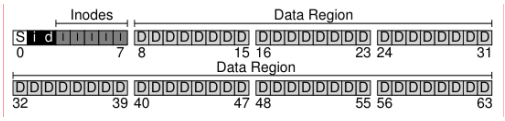
are typically not that big, for example 128 or 256 bytes. a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes. some way to track whether inodes or data blocks are free or allocated.

1.2.3.4 allocation structures

We choose a bitmap, one for the data region and one for the inode table

1.2.3.5 superblock,

The superblock contains information about this particular file system, how many inodes and data blocks are in the file system where the inode table begins and so forth. It will likely also include a magic number of some kind to identify the file system type



1.3 The Inode

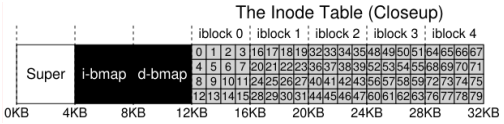
1.3.1 The name

is short for index node,

is the generic name to describe the structure that holds the metadata for a given file,

1.3.2 Each inode

is implicitly referred to by a number (called the i-number), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located.



1.3.3 To read inode number 32,

the file system would first calculate the offset into the inode region ($32 \cdot$ sizeof(inode) or 8192), add it to the start address of the inode table on disk (inodeStartAddr = 12KB), and thus arrive upon the correct byte address of the desired block of inodes: 20KB.

1.3.4 the sector address sector of the inode block can be calculated as follows:

blk = (inumber * sizeof(inode_t)) / blockSize;

sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;

1.3.5 Inside each inode

is virtually all of the information you need about a file: - its type - its size, - the number of blocks allocated to it, - protection information - some time information, - information about where its data blocks reside on disk

We refer to all such information about a file as **metadata**;

1.3.6 The Multi-Level Index

1.3.6.1 One common idea

is to have a special pointer known as an **indirect pointer.** it points to a block that contains more pointers, each of which point to user data.

1.3.6.2 an inode may have

some fixed number of direct pointers (e.g., 12), and a single indirect pointer. If a file grows large enough, an indirect block is allocated Assuming 4-KB blocks and 4-byte disk addresses, that adds another 1024 pointers;

1.3.6.3 support even larger files.

To do so, just add another pointer to the inode: the **double indirect pointer.** This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks.

1.3.6.4 this imbalanced tree is referred to as

the multi-level index approach to pointing to file blocks.

1.3.6.5 why use an imbalanced tree like this?

most files are small. if most files are indeed small, it makes sense to optimize for this case.

1.4 Directory Organization

1.4.1 a directory

basically just contains a list of (entry name, inode number) pairs.

1.4.2 The on-disk data for dir might look like:

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

1.4.2.1 each entry has

- an inode number, - record length (the total bytes for the name plus any left over space), - string length (the actual length of the name), - and finally the name of the entry.

Note that each directory has two extra entries, . “dot” and .. “dot-dot”;

1.4.3 file systems treat directories as a special type of file.

a directory has an inode, somewhere in the inode table (with the type field of the inode marked as “directory” instead of “regular file”). The directory has data blocks pointed to by the inode these data blocks live in the data block region of our simple file system.

1.5 Free Space Management

1.5.1 when we create a file,

we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information.

1.5.1.1 A similar set of activities
take place when a data block is allocated.

1.5.2 Some other considerations might also come into play

some Linux file systems, will look for a sequence of blocks (say 8) that are free by finding such a sequence the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance.

1.6 Access Paths: Reading and Writing

1.6.1 For the following examples,

let us assume that the file system has been mounted and thus that the superblock is already in memory. Everything else (i.e., inodes, directories) is still on the disk.

1.6.2 Reading A File From Disk

let us first assume that you want to simply open a file (e.g., /foo/bar), read it, and then close it. For this simple example, let's assume the file is just 12KB in size (i.e., 3 blocks).

1.6.2.1 you issue an open("/foo/bar", O RDONLY) call,
the file system first needs to find the inode for the file bar, The file system must traverse the pathname and thus locate the desired inode.

1.6.2.2 All traversals
begin at the root of the file system, in the root directory which is simply called /. the first thing the FS will read from disk is the inode of the root directory.

1.6.2.3 the root inode number
In most UNIX file systems, the root inode number is 2.

1.6.2.4 The next step
is to recursively traverse the pathname until the desired inode is found.

1.6.2.5 In this example,
the FS reads the block containing the inode of foo and then its directory data, finally finding the inode number of bar.

1.6.2.6 The final step of open()
is to read bar's inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.

1.6.2.7 Once open,
the program can then issue a read() The first read will read in the first block of the file,

1.6.2.8 The read
will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

1.6.2.9 At some point, the file will be closed.
the file descriptor should be deallocated, that is all the FS really needs to do.

1.6.3 ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES

Allocation structures, such as bitmaps, are only accessed when allocation is needed.

1.6.4 Figure 40.3

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**			**read**				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

the open causes numerous reads to take place in order to finally locate the inode of the file. Afterwards, reading each block requires the file system to first consult the inode, then read the block, and then update the inode's last-accessed-time field with a write.

1.6.5 the amount of I/O generated by the open

is proportional to the length of the pathname. Making this worse would be the presence of large directories; writing out a file is even worse.

1.6.6 Writing A File To Disk

1.6.6.1 process.

- First, the file must be opened - Then, the application can issue write() - Finally, the file is closed.

1.6.6.2 writing to the file may also

allocate a block

1.6.6.3 When writing out a new file,

each write not only has to write data to disk but has to first decide

which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode).

1.6.6.4 each write to a file logically generates five I/Os:

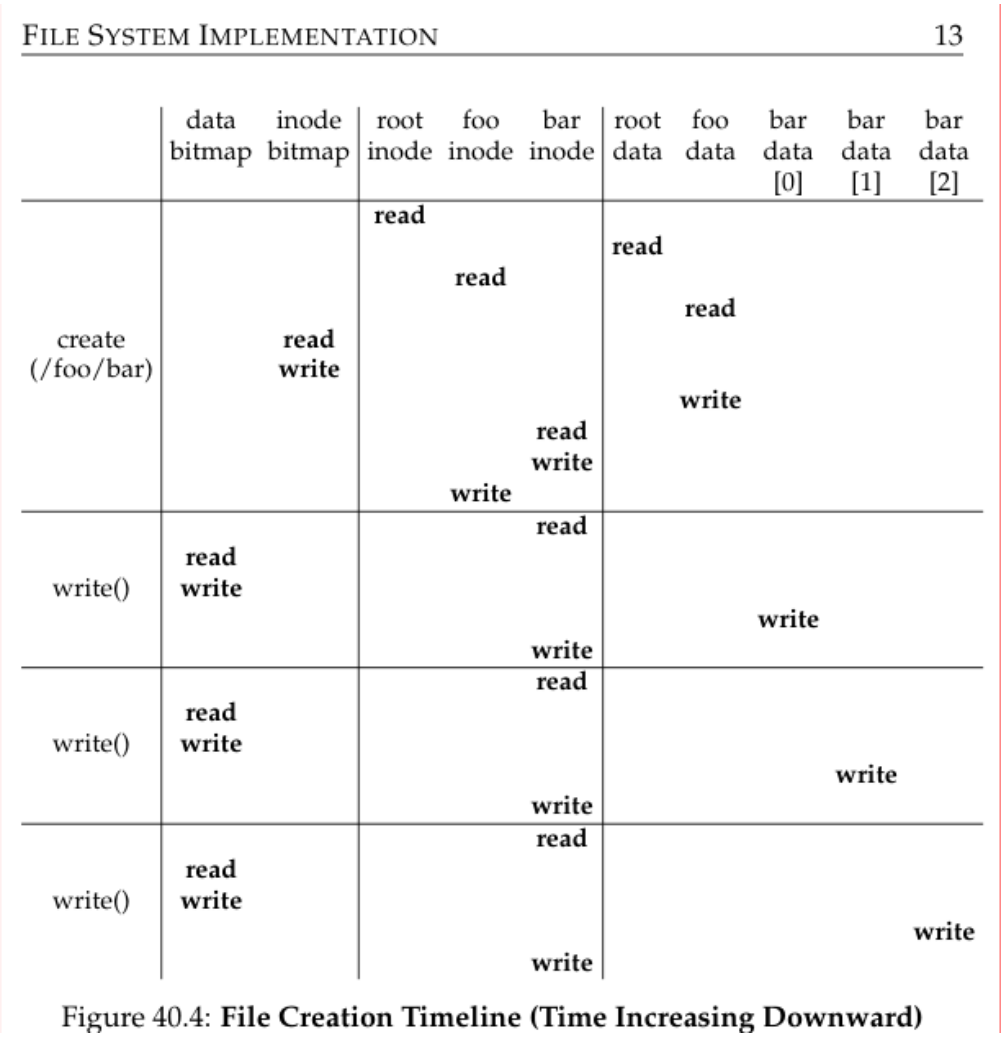
- one to read the data bitmap (which is then updated to mark the newly-allocated block as used),
- one to write the bitmap (to reflect its new state to disk),
- two more to read and then write the inode (which is updated with the new block's location),
- and finally one to write the actual block itself.

1.6.6.5 To create a file,

- one read to the inode bitmap (to find a free inode),
- one write to the inode bitmap (to mark it allocated),
- one write to the new inode itself (to initialize it),
- one to the data of the directory (to link the high-level name of the file to its inode number),
- and one read and write to the directory inode to update it.

If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too.

1.6.7 Figure 40.4



FILE SYSTEM IMPLEMENTATION 13

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

1.7 Caching and Buffering

1.7.1 most file systems

aggressively use system memory (DRAM) to cache important blocks. without caching, every file open would require at least two reads for every level in the directory hierarchy

1.7.2 fixed-size cache

to hold popular blocks. strategies such as LRU and different variants would decide which blocks to keep in cache.

1.7.2.1 This fixed-size cache
would usually be allocated at boot time to be roughly 10% of total memory.

1.7.3 dynamic partitioning approach.

many modern operating systems integrate virtual memory pages and file system pages into a unified page cache memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time.

1.7.4 the file open example with caching.

The first open may generate a lot of I/O traffic to read in directory inode and data, subsequent file opens of that same file will mostly hit in the cache and thus no I/O is needed.

1.7.5 TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING

1.7.5.1 The static approach
simply divides the resource into fixed proportions once;

1.7.5.2 The dynamic approach
is more flexible, giving out differing amounts of the resource over time;

1.7.6 effect of caching on writes.

write traffic has to go to disk in order to become persistent.

1.7.6.1 write buffering
by delaying writes, the file system can **batch** some updates into a smaller set of I/Os;

1.7.6.2 for example,
if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update.

1.7.6.3 by buffering a number of writes

in memory, the system can then schedule the subsequent I/Os and thus increase performance.

1.7.6.4 some writes are avoided altogether

by delaying them; if an application creates a file and then deletes it, delaying the writes to reflect the file creation to disk avoids them entirely.

1.7.6.5 trade-off:

if the system crashes before the updates have been propagated to disk, the updates are lost; by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

1.7.6.6 Some applications (such as databases) don't enjoy this trade-off.

Thus, to avoid unexpected data loss due to write buffering, they simply force writes to disk, or by using the raw disk interface and avoiding the file system altogether2.