Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T30 File System Implementation

Referência principal
Ch. 40 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 14 de novembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

How to implement a simple file system

How can we build a simple file system?

What structures are needed on the disk?

What do they need to track?

How are they accessed?

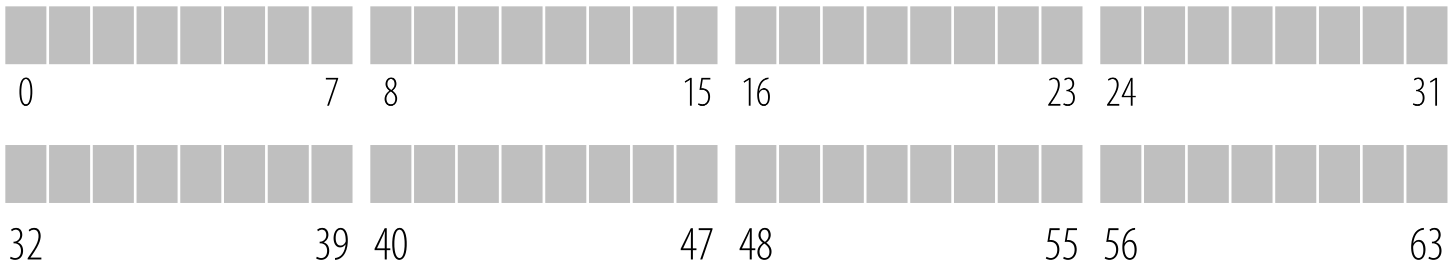
The Way To Think

- To implement a file system, there are two main aspects to think about
  - Data structures
    - What types of on-disk structures does the file system need to organize its data and metadata?
  - Access methods
    - How does a file system 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 structures are written?
    - How do such operations affect the performance of an application?

Overall Organization

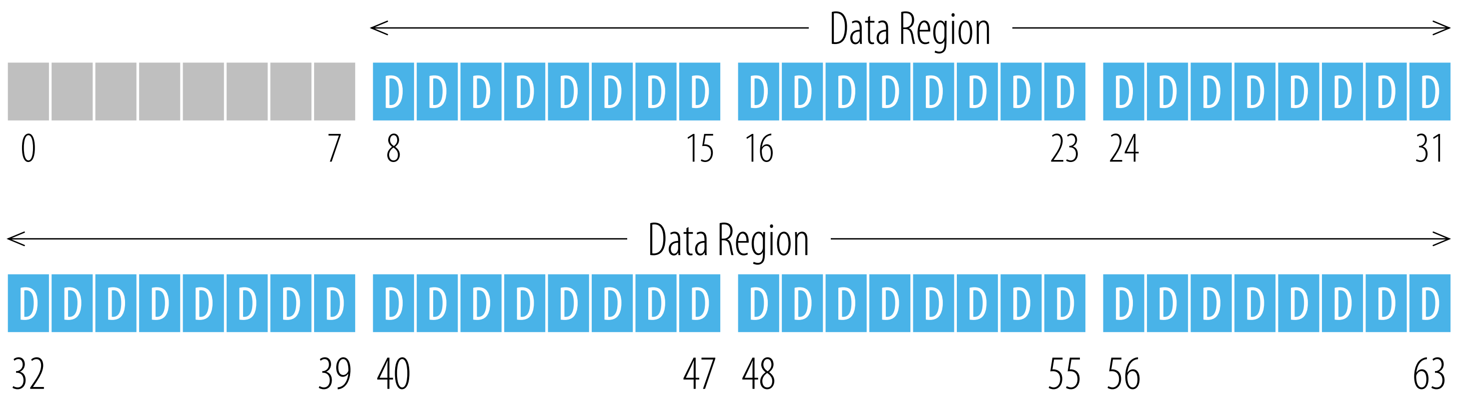
- Let's develop the overall on-disk organization of a very simple file system's data structures.

- Assume we have a very small disk with only 256KB.
  - Let's divide it into 4KB blocks.
  - In this case, there will be 64 blocks, numbered from 0 to 63.



Data region in the file system

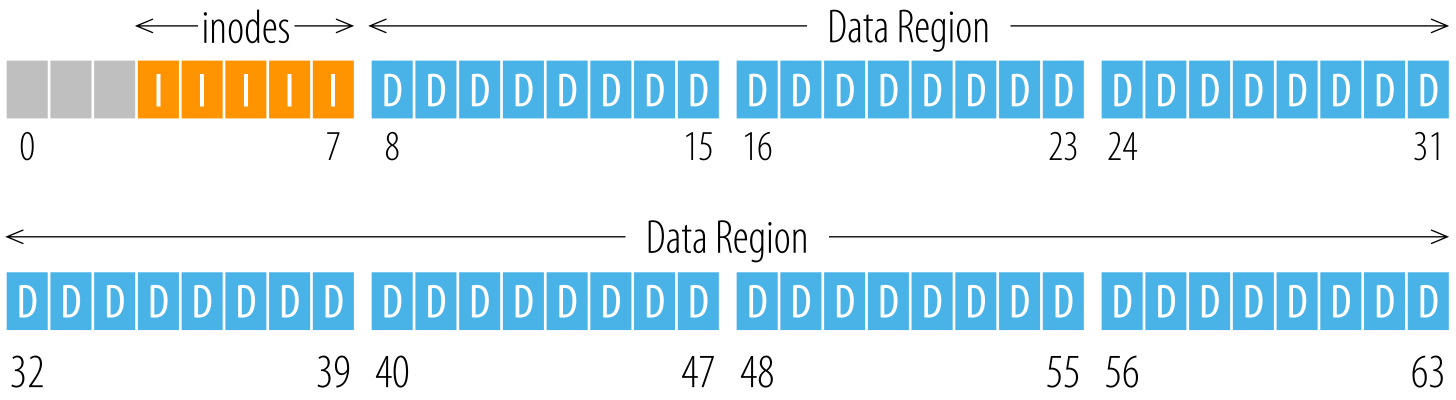
- A large fixed region of the disk should be reserved to store user data.
  - Let's use the last 56 of the 64 blocks for that.



- What do we need next?
  - The file system must track which data blocks belong to a file, the size of the file, its owner and their access rights, last access and modify times, etc.
  - Such information is usually stored on a structure called an **inode**.

Inode table in the file system

- Let’s assume an inode occupies 256B and reserve some space for an **inode table**, which will hold an array of on-disk **inodes**.
  - Let’s use 5 blocks (say 3 to 7) for the inode table.
    - Since a 4KB block can hold 16 inodes, the file system may contain up to 80 files.



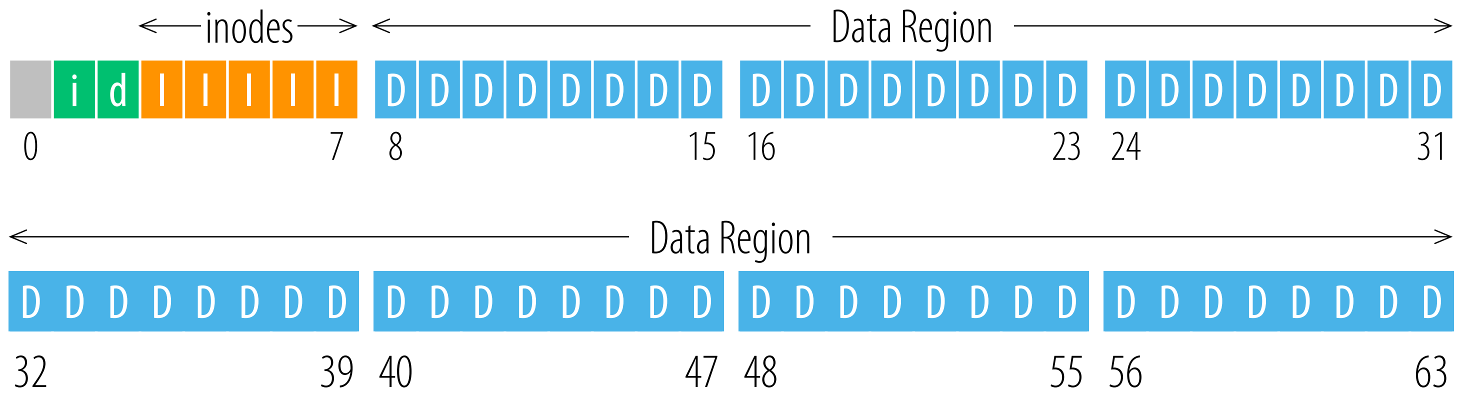
- Is there anything still missing in our design?

Allocation structures

- We need some **allocation structures** to track whether inodes or data blocks are free or allocated.
- Of course, there are many possible allocation-tracking methods, e.g.
  - A **free list** that points to the first free block, which then points to the next free block, and so forth.
  - A simple structure called a **bitmap**, where each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1).
- In our example, we will take the bitmap approach.

Allocation structures

- Let’s use two bitmaps, one for the inodes (i) and another for the data (d).
  - We will reserve one block for each.

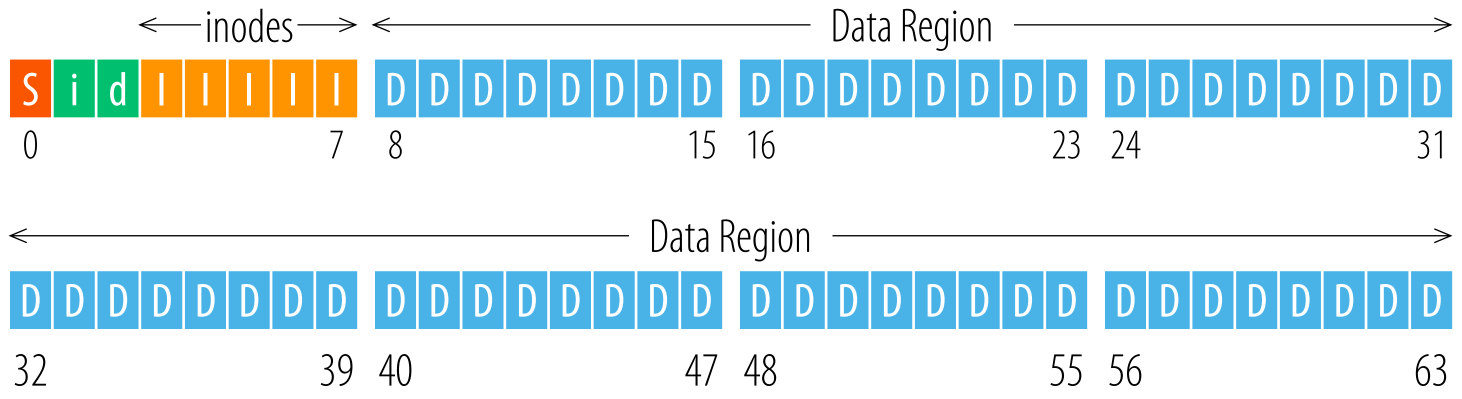

inodes
Data Region
i d I I I I I
D D D D D D D D D D D D D D D D D D D D D D D D
0
7
8
15
16
23
24
31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32
39
40
47
48
55
56
63

- What do you think about the size of our bitmaps?

- Is there anything still missing in our design?

The superblock

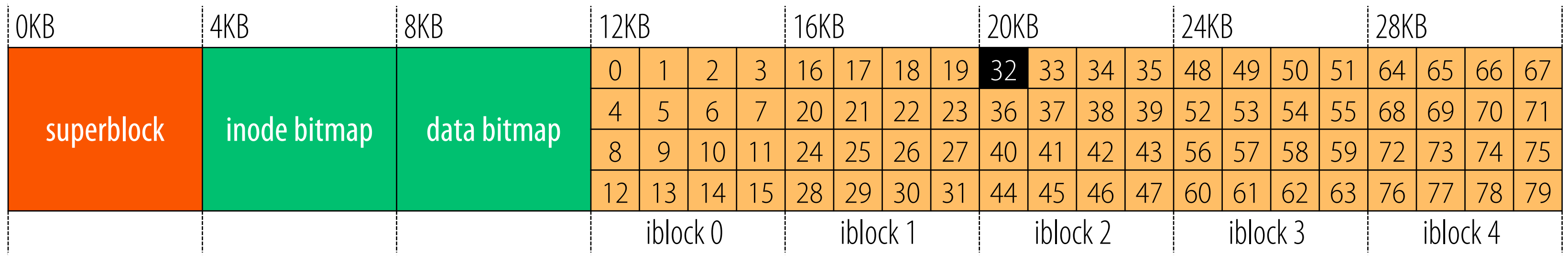
- We still need a **superblock** containing information about the file system, including the number of inodes and data blocks, the location of the inode table and the data region, etc.
- When mounting a file system, the OS will read the superblock first to initialize various parameters before attaching the volume to the file-system tree.
- This completes the design of the layout of our very simple file system.



inodes
Data Region
S i d I I I I I D D D D D D D D D D D D D D D D D D D D D D D D
0 7 8 15 16 23 24 31
Data Region
D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D D
32 39 40 47 48 55 56 63

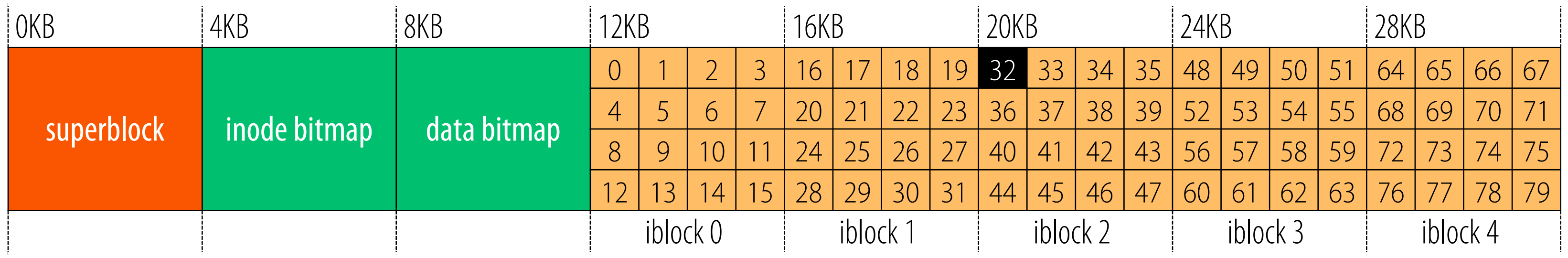
File Organization: inodes

- Each inode is referred to by its **inumber**.
- In a simple file system like ours, given an inumber, it is possible to calculate where the corresponding inode is on the disk.
  - For example, given inumber 32 the file system would
    - Calculate its offset into the inode table $= 32 \cdot \text{sizeof}(inode) = 32 \cdot 256 = 8192$ and add the start address of the inode table $= 8192 + 12288 = 20480$ in order to obtain its byte address on the disk $(20K)$.



File Organization: inodes

- However, disks are not byte addressable, but sector addressable.
- The disk consists of a large number of addressable sectors.
  - 512B is a usual size for a sector.
  - In our example, to fetch inode number n the system calculates the sector address $iaddr$ of the inode block as
    - $blk = (inumber * \text{sizeof}(inode))/blocksize = (32 \cdot 256)/4096 = 2$
    - $iaddr = (blk \cdot blocksize + inodeStartAddr)/sectorSize = (2 \cdot 4096 + 12288)/512 = 40$



The inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
4	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
2	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

- The inode holds virtually all the information you need about a file.
  - The table on the left shows the structure of a simplified Ext2 inode.
- Look at the highlighted row.
  - What can you do with it?
  - What files can you address with 60 blocks?

Multi-Level Index

- One way to support bigger files is to use a multi-level index.
  - On a multi-level index, an indirect pointer points to a block that contains more pointers.

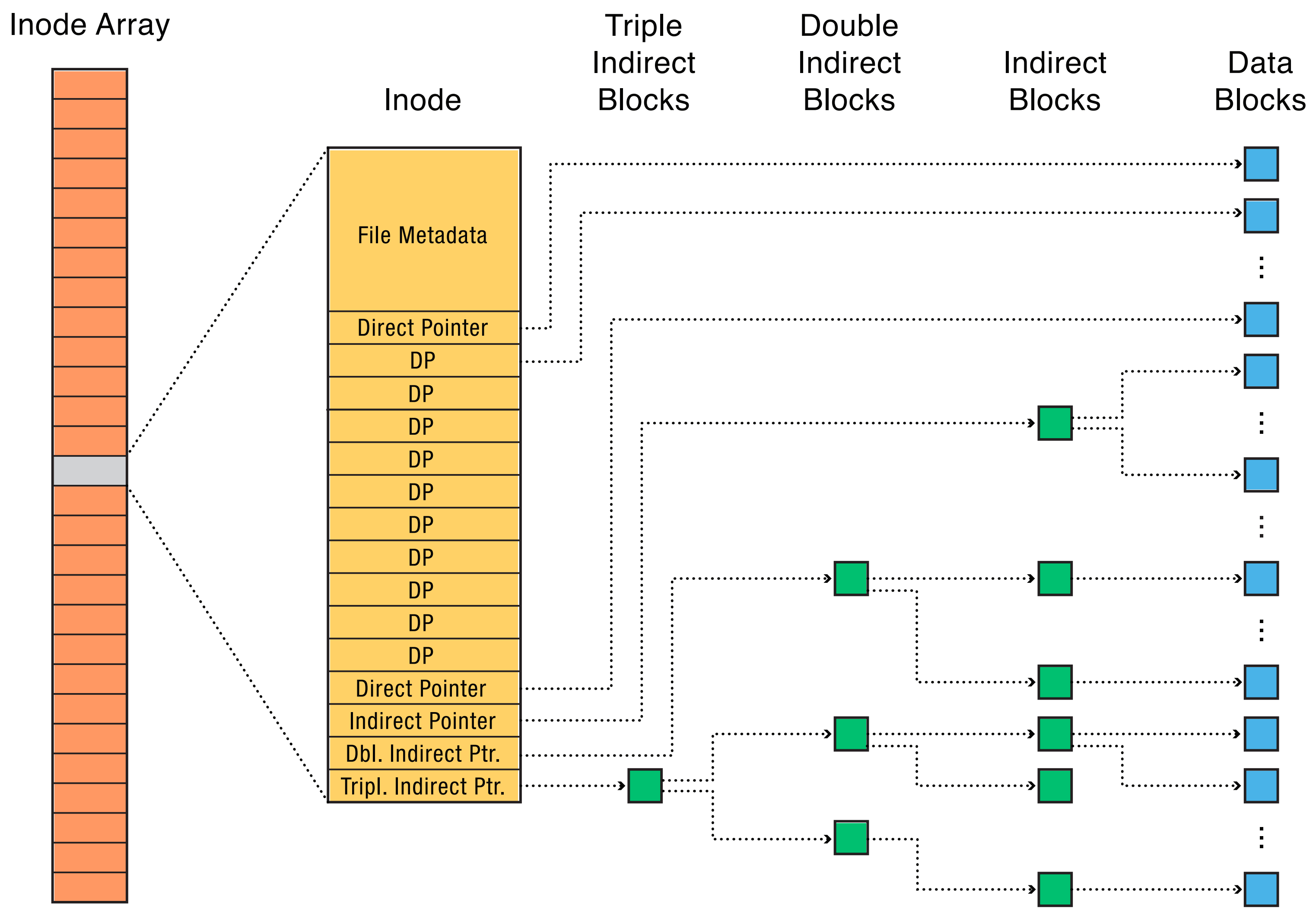
- Assume that an inode has a fixed number of direct pointers (e.g. 12) and a single indirect pointer.
  - If a file grows larger than 12 blocks, an indirect block is allocated (from the data region of the disk) and the inode's slot for an indirect pointer is set to point to it.
    - Assuming 4KB blocks and 4B disk addresses, the indirect block can hold 1024 pointers.
    - The maximum file size becomes $(12 + 1024) \times 4096 = 4243456 = 4144KB$

- What if our file is greater than that?

Multi-Level Index

- For larger files, a double indirect pointer points to a block that contains indirect blocks and thus allows a file to grow an additional 1024 x 1024 or 1 million 4KB blocks.
  - Twelve direct pointers, a single and a double indirect block support files of size up to $(12 + 1024 + 1024^2)\times 4KB > 4GB$.
- To support even larger files, a triple indirect pointer would point to a block that contains double indirect blocks.
  - Can you determine the maximum file size on such a system?
- Linux EXT2, EXT3, NetApp's WAFL and the original Unix file system, for instance, use a multi-level index.
  - Linux EXT4 and MS NTFS use extents instead of simple pointers.

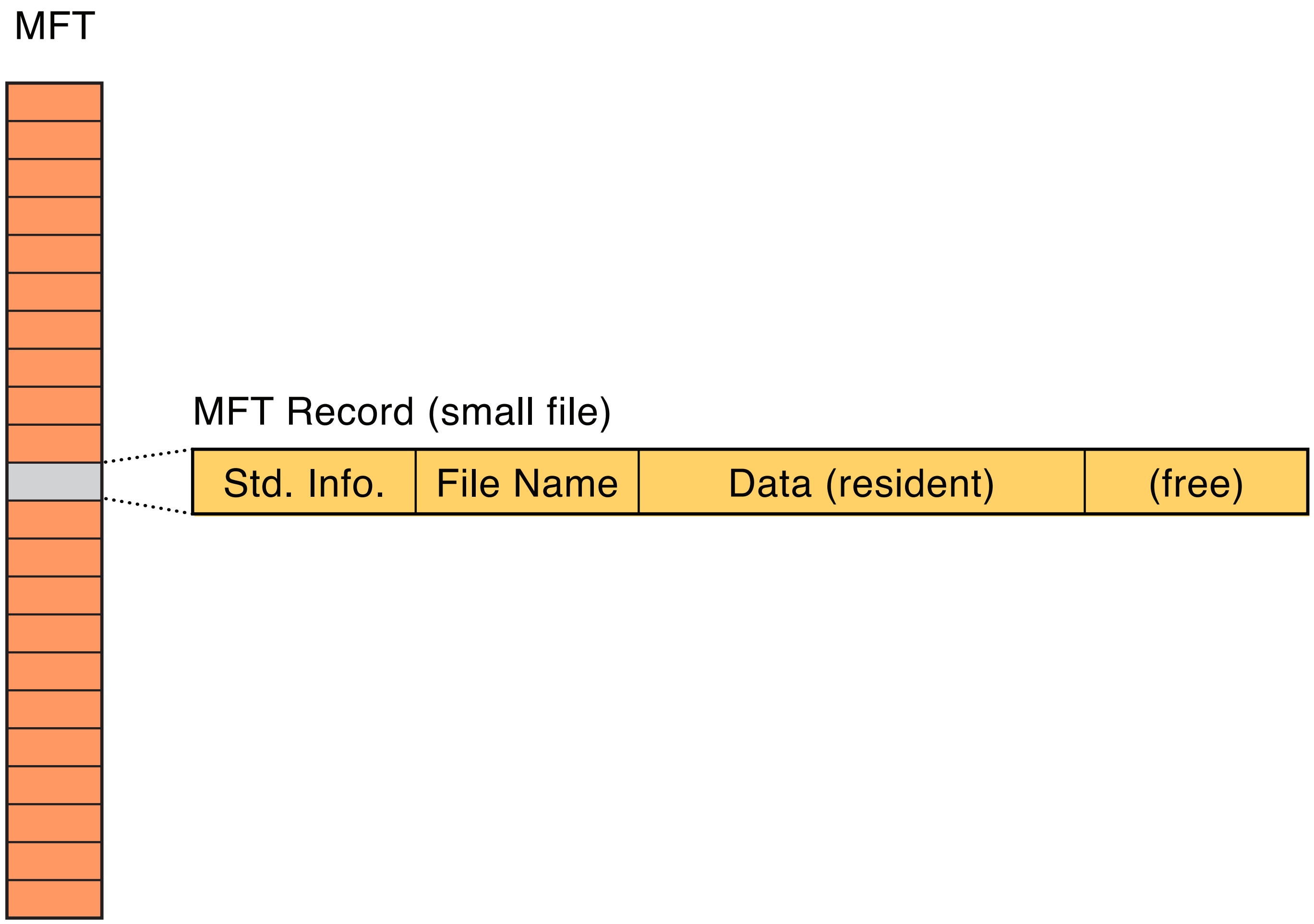
Multi-Level Index



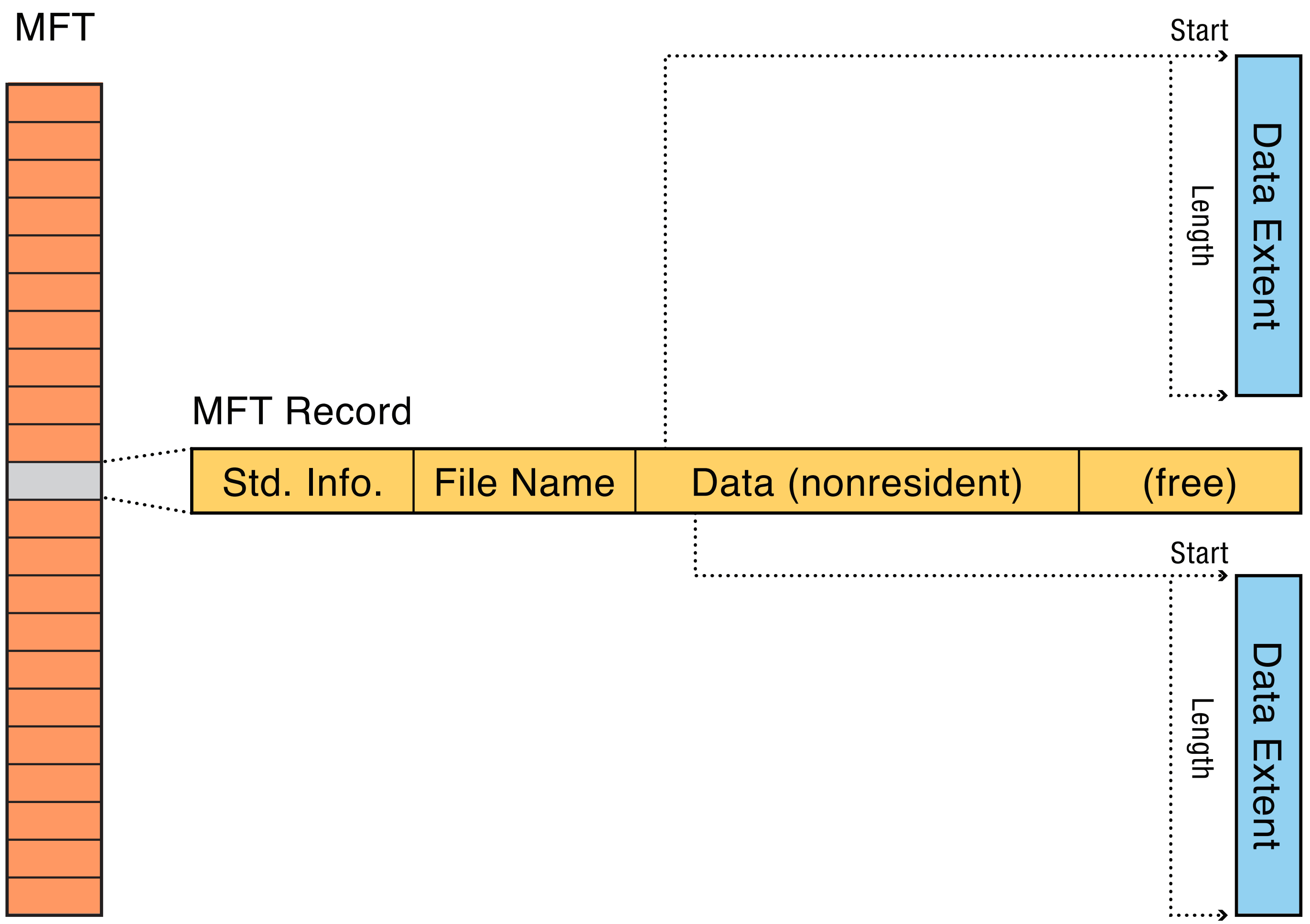
What are extents?

- An extent is simply a disk pointer plus a length (in blocks).
  - Instead of a pointer for each block, only a pointer and a length are needed to specify the on-disk location of a file.
  - Just one single extent is limiting, because it may be difficult to find a large-enough contiguous chunk of on-disk free space when allocating a file.
  - To give the file system more freedom during file allocation, extent-based file systems often allow for more than one extent.
- In comparing the two models…
  - Pointer-based approaches are the most flexible but use a large amount of metadata per file (particularly for large files).
  - Extent-based approaches are less flexible but more compact.
    - They work particularly well when there is enough free space on the disk and files can be laid out contiguously.

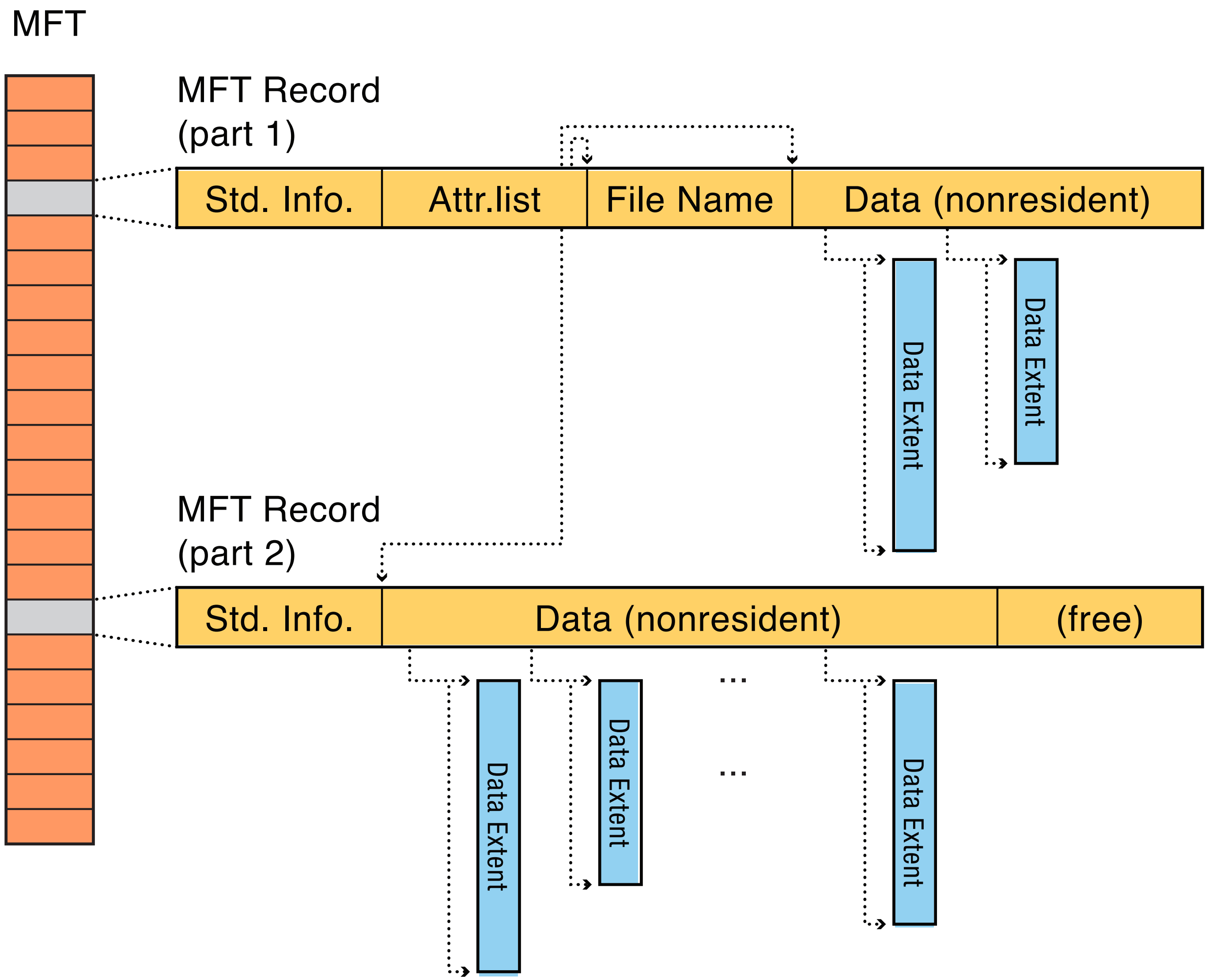
NTFS index structure



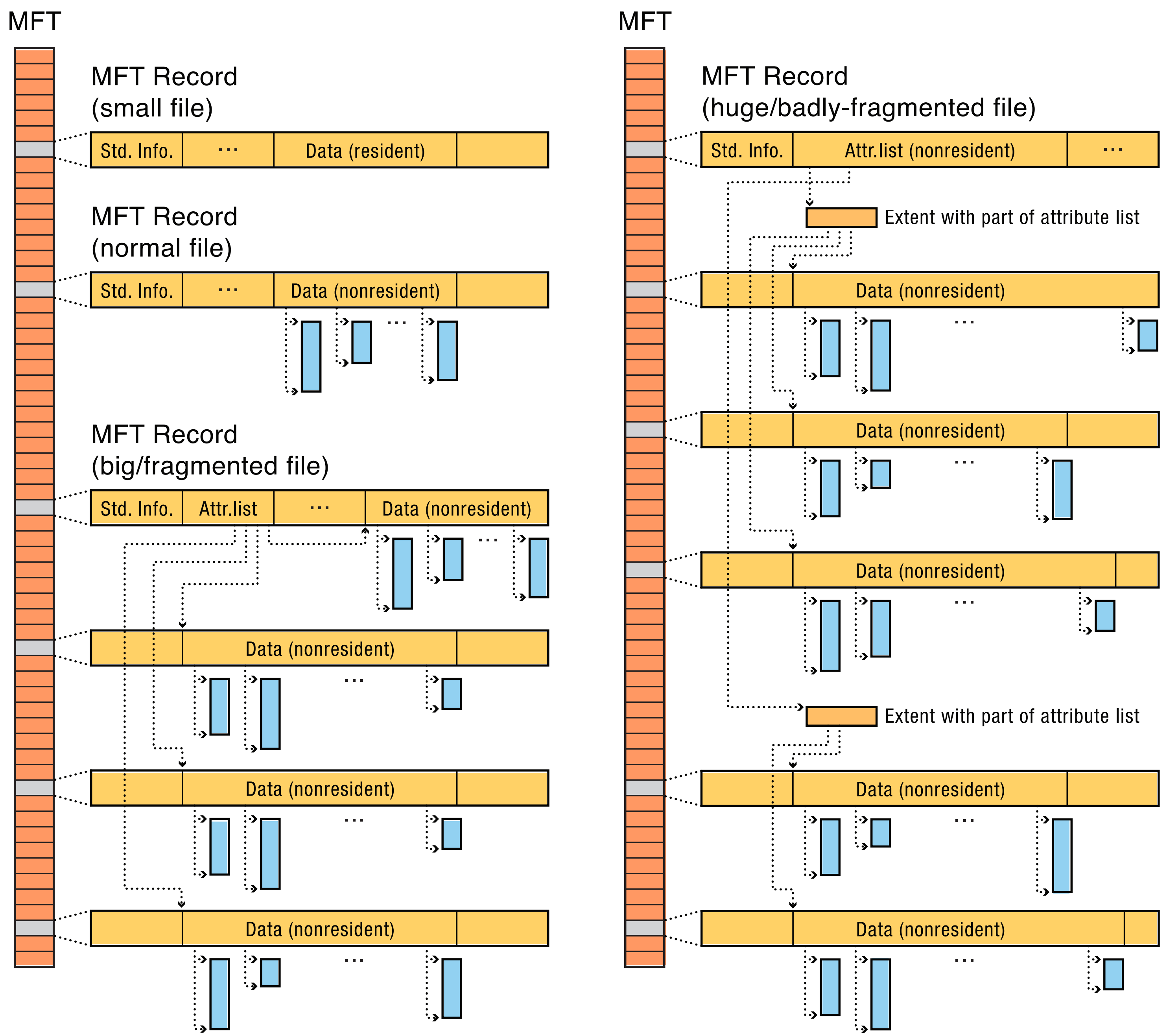
NTFS index structure



NTFS index structure



NTFS index structure



The Multi-Level Index

- Why use such an imbalanced tree? Why not a different approach?
  - Most files are small, so it makes sense to optimize for this case. Look at the table below.
  - However, many other possibilities exist; after all, the inode is just a data structure...
    - Microsoft, for instance, has been working on a proprietary Resilient File System (ReFS), codenamed "Protogon", with the intent of becoming the "next generation" file system after NTFS for quite some time.

Agrawal's File System Measurement Summary (2007)	
Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file system remain -50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Directory Organization

- Directory contains a list of (entry name, inode number) pairs.
- Each directory has two extra files "**.**" (dot) for the current directory and "**..**" (dot-dot) for the parent directory.
  - For example, assume a directory `dir` (inode number 5) with three files in it (`foo`, `bar`, and `foobar`, with inode numbers 12, 13, and 24, respectively).
  - The on-disk data for `dir` might look like this, where each entry shows the inode number (`inum`), record length (`reclen`, the total bytes for the name plus any left-over space), string length (`strlen`, the actual length of the name), and finally the name of the entry (`name`).

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Where are directories stored?

- Directories are often treated as a special type of file.
  - For exemple, a directory may have 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 (and perhaps, indirect blocks) which live in the data block region of our simple file system.
  - Our on-disk structure thus remains unchanged.
- A simple linear list of entries is not the only way to store directory information.
  - For example, XFS stores directories in B-tree form, making file create operations faster than in systems with simple lists that must be scanned in their entirety.

How to manage free space?

- The file system must track whether inodes and data blocks are free.
- There are many ways to manage free space.
  - Some early file systems used linked lists of free blocks.
  - Modern file systems use more sophisticated data structures, e.g. B-trees.
  - In our example, we have used two simple bitmaps.
    - When creating a file, the file system will search the bitmap for a free inode, allocate it to the file; mark it as used and update the on-disk bitmap.
    - A similar sequence of activities takes place when a data block is allocated.
  - Allocating data blocks for a new file also brings other concerns.
    - For example, some Linux file systems, such as ext2 and ext3, will look for a sequence of blocks (say 8) that are free when a new file is created.
    - This guarantees that at least a portion of the file will be contiguous on the disk, thus improving performance.
    - Such a pre-allocation policy is a commonly-used heuristic when allocating space for data blocks.

Access Paths: Reading a File From Disk

- Issue an open(“/foo/bar”, O_RDONLY),
  - Traverse the pathname and thus locate the desired indoe.
  - Begin at the root of the file system (/)
    - In most Unix file systems, the root inode number is 2
  - Filesystem reads in the block that contains inode number 2.
  - Look inside of it to find pointer to data blocks (contents of the root).
  - By reading in one or more directory data blocks, It will find “foo” directory.
  - Traverse recursively the path name until the desired inode (“bar”)
  - Check finale permissions, allocate a file descriptor for this process and returns file descriptor to user.

Access paths

- Now that we have an idea of how files and directories are stored on disk, we can discuss the flow of operations while reading or writing a file.
- For the following examples, let us assume that
  - The file system has been mounted and thus the superblock is already in memory.
  - Everything else (i.e., inodes, directories) is still on the disk.
- The file reading example also assumes that a file `/foo/bar` (just 12KB in size, i.e., 3 blocks) must be opened, read and then closed.
- The file writing example will go through the steps needed to create that file.

Reading a File Timeline

	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					

time

Reading a File From Disk

Opening the file

- Issue an `open("/foo/bar", O_RDONLY)`
  - The system must traverse the pathname and locate the desired inode.
  - It begins at the root of the file system `(/)`
    - In most Unix file systems, the root inode number is 2
  - File system reads in the block that contains inode number 2.
  - It looks inside it to find pointer to data blocks (contents of the root).
  - By reading one or more directory data blocks, it will find "`foo`" directory.
  - Traverse recursively the path name until the desired inode ("`bar`")
  - Read `bar`'s inode into memory, check permissions, allocate a file descriptor for this process and returns the file descriptor to the user.

Reading and closing the file

- Issue `read()` to read from the file.
  - Read in the first block of the file, consulting the inode to find the location of such a block.
    - Update the inode with a new last-accessed time.
    - Update in-memory open file table for file descriptor, the file offset.
- When file is closed…
  - File descriptor should be deallocated, and that is all the file system really needs to do.
  - No disk I/Os take place.

Reading a File Timeline

	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					

time

Writing to file timeline

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

Writing to Disk

- Issue write() to update the file with new contents.
- File may allocate a block (unless the block is being overwritten).
  - Need to update data block, data bitmap.
  - It generates five I/Os:
    - one to read the data bitmap
    - one to write the bitmap (to reflect its new state to disk)
    - two more to read and then write the inode
    - one to write the actual block itself.
  - To create a file, it also allocates space for the directory, causing high I/O traffic.

Writing to file timeline

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

Caching and Buffering

- Reading and writing files are expensive, incurring many I/Os.
  - For example, long pathname (`/1/2/3/….../100/file.txt`)
    - One to read the inode of the directory and at least one read its data.
    - Literally perform hundreds of reads just to open the file.
- In order to reduce I/O traffic, file systems aggressively use system memory (DRAM) to cache.
  - Early file systems used fixed-size cache to hold popular blocks.
    - Static partitioning of memory can be wasteful.
  - Modem systems use dynamic partitioning approach, unified page cache.
- Excessive read I/O can be avoided by large cache.

Caching and Buffering

- Write traffic has to go to disk for persistency and cannot be reduced by caching.
- File systems use write buffering for write performance benefits.
  - Delaying writes (file system batches some updates into a smaller set of I/Os).
  - By buffering several writes in memory, the file system can then schedule subsequent I/Os and thus increase performance.
  - By avoiding writes.
- Some applications force data flushing to disk by calling `fsync()` or doing direct I/O.