File Systems

key concepts: file, directory, link, open/close, descriptor, read, write, seek, file naming, block, i-node, crash consistency, journaling

Zille Huma Kamal

David R. Cheriton School of Computer Science
University of Waterloo

Spring 2022

Disk vs. Memory

	Disk	MLC NAND Flash	DRAM
Smallest write	sector	page	byte
Atomic write	sector	page	byte/word
Random read	8 ms	75 μs	50 ns
Random write	8 ms	300 μs*	50 ns
Sequential read	100 MB/s	250 MB/s	$>$ 1 GB/s
Sequential write	100 MB/s	170 MB/s*	$>$ 1 GB/s
Cost	$0.04/GB	$0.65/GB	$10/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*Flash write performance degrades over time

Edited from Dr. Ali Mashtizadeh - Unveristy of Waterloo -CS350 Fall 2021 Slides

Files and File Systems

- there is a need to efficiently organize large storage
- storage units work with bulk data, e.g. in HDD the unit of atomicity sector
- **user's view:** a file is single logical sequence of bytes
- **file system - an operating system abstraction:** to provide persistent, named data.
- **files**: named data objects
  - data consists of a sequence of numbered bytes, each byte is an offset from the start of the sequence
  - file may change size over time
  - file has associated meta-data (e.g., type, timestamp, access controls)
- **file systems**: the data structures and algorithms used to store, retrieve, and access files
  - **logical file system**: high-level API, what a user sees
  - **virtual file system**: abstraction of lower level file systems, presents multiple different underlying file systems to the user as one
  - **physical file system**: how files are actually stored on physical media

Directories, Volumes and Mounts

- file systems can be organized as a directory tree
- **path** identifies file and directories
- directroy is a special file with a list of mappings from `filenames` to file umber
- therefore it is used to translate filename to file number
- the `hard link` is the association of a file number and its filename
- `soft link` or `symbolic link` is a directory mapping for a file name to another filename. Windows shortcuts and MacOS alias are similar counterparts to symbolic links
- **volume** - logical mass storage system composed of a collection of physical storage device(s)
- **mount** - allows a single computer to use mulitple file systems, by creating a mapping from an existing file system to the root directory of a mounted file system

File Interface: Basics

- file access types: sequential or random access
- open
  - `open` returns a **file identifier** (or handle or descriptor), which is used in subsequent operations to identify the file.
  - other operations (e.g., read, write) require file descriptor as a parameter
- `close`
  - kernel tracks while file descriptors are currently valid for each process
  - close invalidates a valid file descriptor
- `read`, `write`, `seek`
  - read copies data from a file into a virtual address space
  - write copies data from a virtual address space into a file
  - seek enables non-sequential reading/writing
- get/set file meta-data, e.g., Unix `fstat`, `chmod`, `ls -la`

File Position and Seeks

- each file descriptor (open file) has an associated **file position**
  - the position starts at byte 0 when the file is opened
- read and write operations
  - start from the current file position
  - update the current file position as bytes are read/written
- this makes sequential file I/O easy for an application to request
- seeks (`lseek`) are used for achieve non-sequential file I/O
  - `lseek` changes the file position associated with a descriptor
  - next read or write from that descriptor will use the new position

Sequential File Reading Example

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time.

File Reading Example Using Seek

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f,(100-i)*512,SEEK_SET);
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time, in reverse order.

`lseek` **does not** modify the file. It also does not check if the new file position is valid (i.e., in the file). It will not return an error or throw an exception if the position is invalid. However, the subsequent read or write operation **will**.

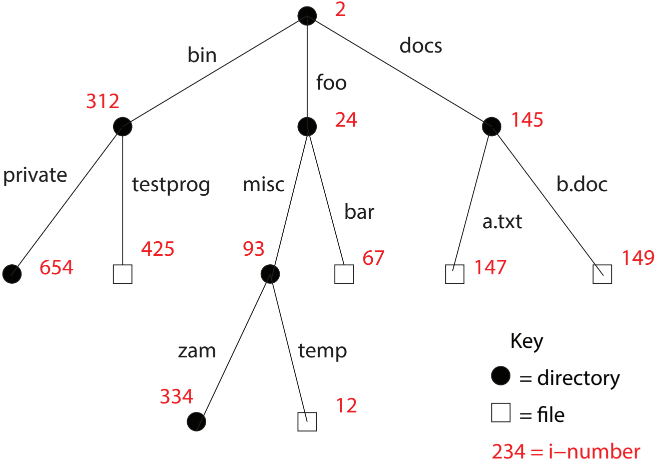
Directories and File Names

- A directory maps **file names** (strings) to **i-numbers**
  - an i-number is a unique (within a file system) identifier for a file or directory
  - given an i-number, the file system can find the data and meta-data for the file
- Directories provide a way for applications to group related files
- Since directories can be nested, a filesystem's directories can be viewed as a tree, with a single **root** directory.
- In a directory tree, files are leaves
- Files may be identified by **pathnames**, which describe a path through the directory tree from the root directory to the file, e.g.:

  `/home/user/courses/cs350/notes/filesys.pdf`
- Directories also have pathnames
- Applications refer to files using pathnames, not i-numbers

> Only the kernel is permitted to edit directories. Why?

Hierarchical Namespace Example

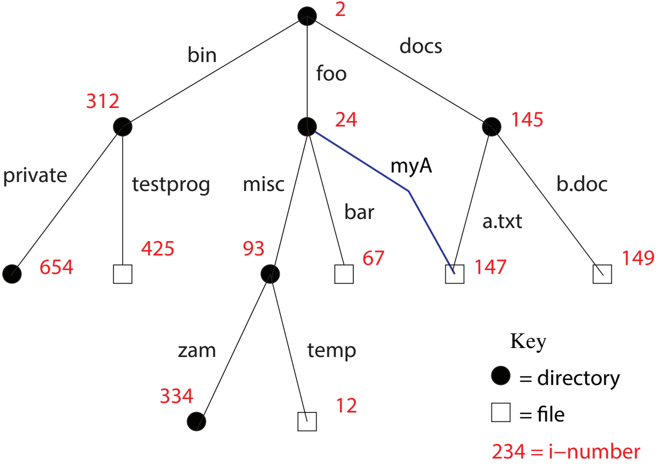


/docs/b.doc is the path for file 149.

Links

- a **hard link** is an association between a name (string) and an i-number
    - each entry in a directory is a hard link
- when a file is created, so is a hard link to that file
    - `open(/foo/misc/biz,O_CREAT|O_TRUNC)`
    - this creates a new file if a file called `/foo/misc/biz` does not already exist
    - it also creates a hard link to the file in the directory `/foo/misc`
- Once a file is created, **additional** hard links can be made to it.
    - example: `link(/docs/a.txt,/foo/myA)` creates a new hard link `myA` in directory `/foo`. The link refers to the i-number of file `/docs/a.txt`, which must exist.
- linking to an existing file creates a new pathname for that file
    - each file has a unique i-number, but may have multiple pathnames
- Not possible to `link` to a directory (to avoid cycles)

Hierarchical Namespace Example



/foo/myA and /docs/a.txt are two different paths to the same file, number 147.

Unlinking

- hard links can be removed:
  - `unlink(/docs/b.doc)`
  - this removes the link `b.doc` from the directory `/docs`
- when the last hard link to a file is removed, the file is also removed
  - since there are no links to the file, it has no pathname, and can no longer be opened

Multiple File Systems

- it is not uncommon for a system to have multiple file systems
- some kind of global file namespace is required
- two examples:

  DOS/Windows: use two-part file names: file system name, pathname within file system

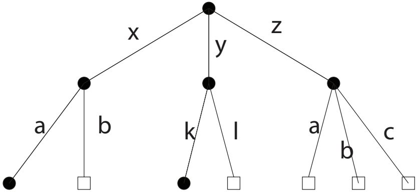
  - example: `C:\user\cs350\schedule.txt`

  Unix: create single hierarchical namespace that combines the namespaces of two file systems

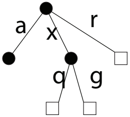
  - Unix `mount` system call does this
- mounting does **not** make two file systems into one file system
  - it merely creates a single, hierarchical namespace that combines the namespaces of two file systems
  - the new namespace is temporary - it exists only until the file system is unmounted

Unix mount Example

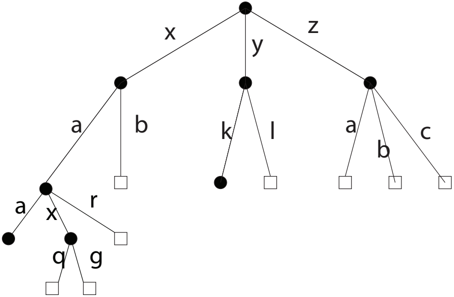
"root" file system



file system X



result of mount (file system X, /x/a)



File System Implementation

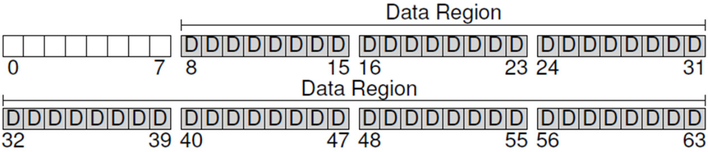
- organize metadata, to construct translations of file offsert to disk addresses
- what needs to be stored persistently?
    - file data
    - file meta-data
    - directories and links
    - file system meta-data
- non-persistent information
    - per process open file descriptor table
        - file handle
        - file position
    - system wide:
        - open file table
        - **cached** copies of persistent data

File System Example

- Use an extremely small disk as an example:
  - 256 KB disk!
  - Most disks have a sector size of 512 bytes
    - Memory is usually *byte addressable*
    - Disk is usually “sector addressable”
  - 512 total sectors on this disk
- Group every 8 consecutive sectors into a block
  - Better spatial locality (fewer seeks)
  - Reduces the number of block pointers (we’ll see what this means soon)
  - 4 KB block is a convenient size for demand paging
  - 64 total blocks on this disk

VSFS: Very Simple File System (1 of 5)

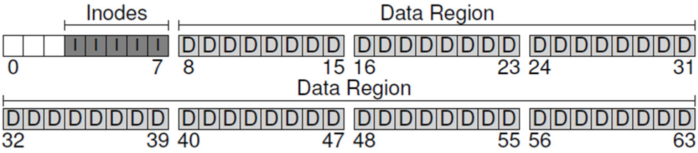
- Most of the blocks should be for storing user data (last 56 blocks)



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
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

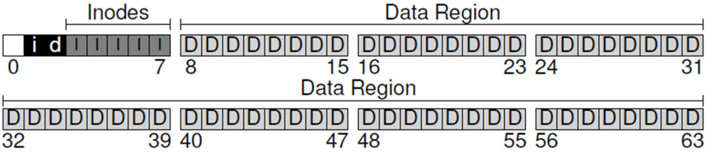
VSFS: Very Simple File System (2 of 5)

- Need some way to map files to data blocks
- Create an array of i-nodes, where each i-node contains the meta-data for a file
- Store the array of i-nodes in a i-node table (**Inodes**)
    - The index into the array is the file's index number (i-number)
- Assume each i-node is 256 bytes, and we dedicate 5 blocks for i-nodes
    - This allows for 80 total i-nodes/files



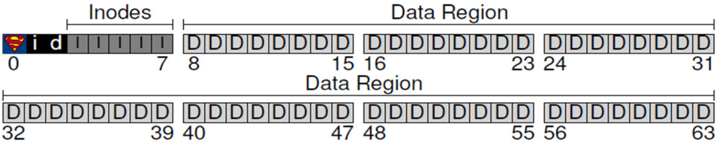
VSFS: Very Simple File System (3 of 5)

- We also need to know which i-nodes and data blocks are unused
- Many ways of doing this:
  - In VSFS, we use a bitmap for free inodes (**i**) and free data blocks(**d**)
  - Can also use a free list instead of a bitmap
- A block size of 4 KB means we can track 32K i-nodes and 32K blocks, since one bit is used to track each i-node or block
  - This is far more than we actually need for this disk

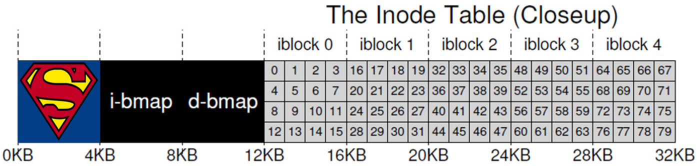


VSFS: Very Simple File System (4 of 5)

- Reserve the first block as the **superblock**
- A superblock contains meta-information about the entire file system
  - e.g., how many i-nodes and blocks are in the system, where the i-node table begins, etc.



VSFS: Very Simple File System (5 of 5)

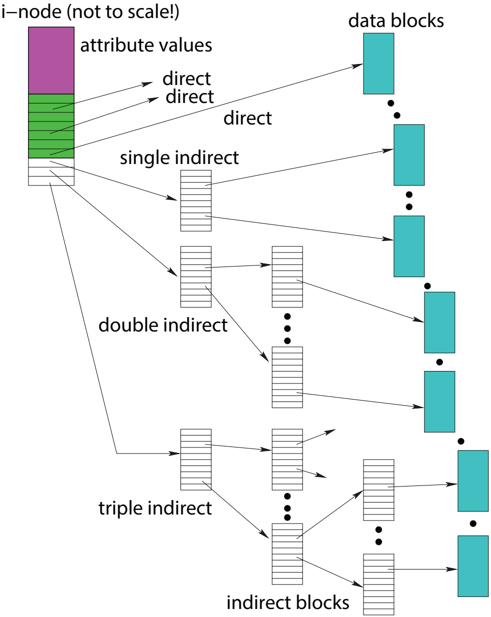


iblock 0				iblock 1				iblock 2				iblock 3				iblock 4			
0	1	2	3	16	17	18	19	32	33	34	35	48	49	50	51	64	65	66	67
4	5	6	7	20	21	22	23	36	37	38	39	52	53	54	55	68	69	70	71
8	9	10	11	24	25	26	27	40	41	42	43	56	57	58	59	72	73	74	75
12	13	14	15	28	29	30	31	44	45	46	47	60	61	62	63	76	77	78	79

i-nodes

- An i-node is a *fixed size* index structure that holds both file meta-data and a small number of pointers to data blocks
- i-node fields may include:
    - file type
    - file permissions
    - file length
    - number of file blocks
    - time of last file access
    - time of last i-node update, last file update
    - number of hard links to this file
    - direct data block pointers
    - single, double, and triple indirect data block pointers

i-node Diagram



VSFS: i-node

- Assume disk blocks can be referenced based on a 4 byte address
  - 2^{32} blocks, 4 KB blocks
  - Maximum disk size is 16 TB
- In VSFS, an i-node is 256 bytes
  - Assume there is enough room for 12 direct pointers to blocks
  - Each pointer points to a different block for storing user data
  - Pointers are ordered: first pointer points to the first block in the file, etc.
- What is the maximum file size if we only have direct pointers?
  - 12 * 4 KB = 48 KB
- Great for small files (which are common)
- Not so great if you want to store big files

VSFS: Indirect Blocks

- In addition to 12 direct pointers, we can also introduce an **indirect pointer**
  - An indirect pointer points to a block full of direct pointers
- 4 KB block of direct pointers = 1024 pointers
  - Maximum file size is: (12 + 1024) * 4 KB = 4144 KB
- This is more than enough for any file that can fit on our tiny 256KB disk, but what if the disk was larger?
- Add a **double indirect pointer**
  - Points to a 4 KB block of indirect pointers
  - (12 + 1024 + 1024 * 1024) * 4 KB
  - Just over 4 GB in size (is this enough?)
- Still not enough? **use a triple indirect pointer**

Reading from a File (/foo/bar)

First, the root i-node is read.

operation	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							

root's i-node will provide the position of root's data, which is where the links are stored.

Reading from a File (/foo/bar)

root's data is read to find the link to foo.

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

In this example, we assume that the directory links fit into a single block.

Reading from a File (/foo/bar)

foo's i-node is read next, providing the location of foo's data.

operation	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						

Reading from a File (/foo/bar)

foo's data is read to find bar's link.

operation	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			

Again, for this example we assume that the links contained in directory foo fit into a single block. This may not always be true.

Reading from a File (/foo/bar)

bar's i-node is read

1. the permissions are checked
2. a file descriptor is returned and added to the processes's file descriptor table
3. the file is added to the kernel's list of open files

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

The file is now open and ready for reads and writes. The position of the file is byte 0. Opening this file required 5 disk reads!

Reading from a File (/foo/bar)

Reading data from /foo/bar, one block at a time.

1. bar's i-node is read
2. a pointer to the correct data block is found

operation	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					

If bar's i-node is not in the i-node cache, it must be read from disk.

Reading from a File (/foo/bar)

1. the data block for /foo/bar is read

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

Reading from a File (/foo/bar)

1. bar's i-node is written to update the access time

operation	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					

Reading from a File (/foo/bar)

Two more data blocks are read.

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

Even if the user wants a single byte out of the middle of a block, the entire block must be read. Disks typically do not permit byte-based addressing, only block or sector addressing.

Creating a File (/foo/bar)

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

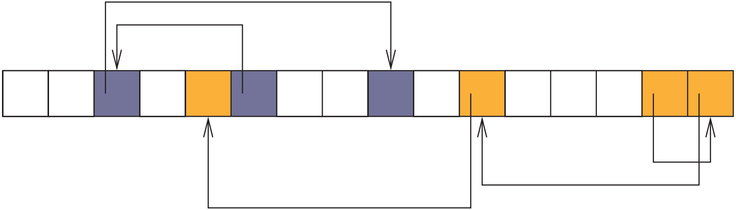
When writing a partial block, that block must be read first. When writing an entire block, no read is required.

Chaining

- VSFS uses a per-file index (direct and indirect pointers) to access blocks
- Two alternative approaches:
  - **Chaining:**
    - Each block includes a pointer to the next block
  - **External chaining:**
    - The chain is kept as an external structure
    - Microsoft's File Allocation Table (FAT) uses external chaining

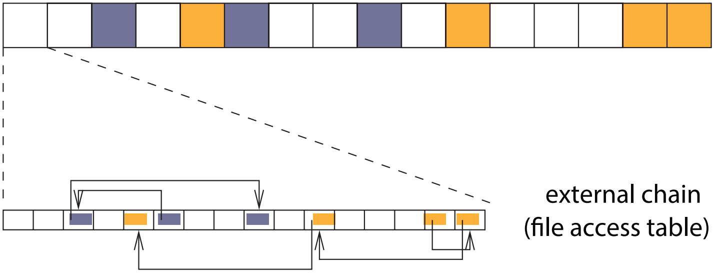
Chaining

- Directory table contains the name of the file, and each file's starting block
- Acceptable for sequential access, very slow for random access (why?)



External Chaining

- Introduces a special file access table that specifies all of the file chains



external chain
(file access table)

File System Design

- File system parameters:
  - *How many i-nodes should a file system have?*
  - *How many direct and indirect blocks should an i-node have?*
  - *What is the "right" block size?*
- For a general purpose file system, design it to be efficient for the common case
  - most files are small, 2KB
  - average file size growing
  - on average, 100 thousand files
  - typically small directories (contain few files)
  - even as disks grow large, the average file system usage is 50%

What about exceptional cases?
What if the files were mostly large, 50GB minimum?
What if each file is less than 1KB?

Problems Caused by Failures

- a single logical file system operation may require several disk I/O operations
- example: deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file's data blocks free in free space index
- what if, because of a failure, some but not all of these changes are reflected on the disk?

> - system failure will destroy in-memory file system structures
> - persistent structures should be **crash consistent**, i.e., should be consistent when system restarts after a failure

Fault Tolerance

- special-purpose consistency checkers (e.g., Unix `fsck` in Berkeley FFS, Linux ext2)
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structures, e.g.:
    - file with no directory entry
    - free space that is not marked as free
- journaling (e.g., Veritas, NTFS, Linux ext3), **write-ahead logging**
  - record file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation
  - **after** changes have been journaled, update the disk data structures (**write-ahead logging**)
  - after a failure, redo journaled updates in case they were not done before the failure