CPSC 457

Filesystems

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Overview

- using filesystems
  - file structure, types, file access, attributes, operations
  - mount points, path names
- implementation of filesystems
  - vfs
  - file block allocation, contiguous, linked / FAT, inodes
  - free space management
- UNIX permissions

Long term storage

What properties do we want in a long-term information storage?

- It must be possible to store a very large amount of information.
- Information must survive termination of a process using it.
- Multiple processes must be able to access information concurrently.
- Easy search / management.

Disks without filesystems

Think of a disk as a linear sequence of **fixed-size blocks** and supporting two operations:

- `read-block i`
- `write-block i`

Similar to memory, but

- **block addressable,**
- **persistent**, and
- **much slower**.

need to load whole block then get bit we want then load it back to mem

can only access blocks of data, so we can't ask for single bits

Questions that quickly arise:

- How do you find information?
- How do you know which blocks are free?
- How do you keep one user from reading another user's data?
- How do programs/users share data?
- How do you (re)organize data?

The answer: implement a filesystem.

Files

- file is an *abstraction* of long term storage, implemented by OS
  - OS allows processes to see a file through contiguous logical address space
  - file contains a sequence of bytes, which can be individually addressed
  - OS maps files onto physical devices
  - OS (generally) does not care about the contents of files
- **file's creator** decides on the contents of the file (file format / internal structure)
  - can create an even higher level abstraction

    eg. treat file as a sequence of bits, numbers, records, …
  - decides on the meaning of the file's contents
- what can be in a file?
  - anything, as long as it can be organized into a **sequence of bytes**
  - eg. source code, executable, images, movies, text, ...

File attributes

- files have contents but also attributes
- file attributes vary from one OS to another but typically consist of these:
  - filename: the symbolic file name is the only information kept in human readable form
  - identifier: unique tag that identifies the file within the FS
  - special type: needed for systems that support different file types (eg. block device)
  - location: a pointer to the location of the file contents on the device
  - size: size of the file
  - time/date: time of creation/last modification/last access, used for usage monitoring
  - user ID, group ID: identifies owner(s) of the file
  - protection information: access control information (eg. read/write/execute)
- many variations, including extended file attributes — such as file checksum
- this information is usually kept separate from file contents, for example in the directory structure

File naming

- names are given to files at creation time, but usually can be changed later as well
- different file-naming rules on different systems, eg.:
  - maximum filename length
  - allowed/restricted characters
  - capitalization
  - filename extensions, enforced vs conventions

unix does not allow "/" in naming

Special file types

- most systems have **special file types** eg.:
  - regular files: both text or binary
  - directories: special files for maintaining FS structure
  - character special files: for I/O on character devices, eg. `/dev/random`
  - block special files: for I/O on block devices, eg. `/dev/sdb0`
  - links: "pointers" to other files
  - sockets, pipes, ...

Special files have random letters in front of permissions

```
$ ls -l /dev
crw-rw-rw- 1 root root      1,   8 Apr 17  2017 random
brw-rw---- 1 root disk      8,   0 Apr 17  2017 sda
brw-rw---- 1 root disk      8,   1 Apr 17  2017 sda1
lrwxrwxrwx 1 root root          15 Apr 17  2017 stderr -> /proc/self/fd/2
drwxr-xr-x 2 root root          60 Apr 17  2017 raw
```

File format (file type)

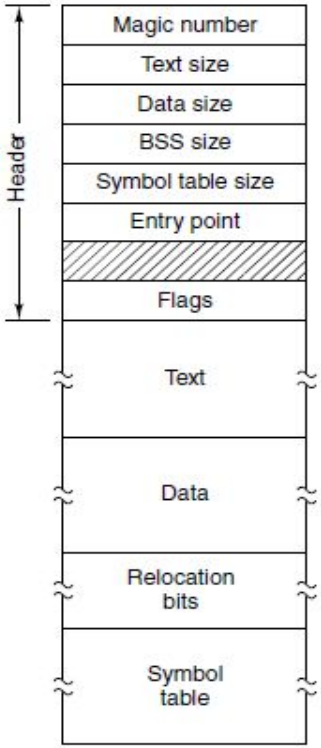
- regular files can have custom types as well (aka. file format or file type)
  - determined by file creator
  - if OS recognizes the file format, it can operate on the file in reasonable ways

    eg. automatically using an appropriate program to open a file
- Windows uses file extension to determine file format, eg. ".jpg", ".xls"
- UNIX uses *magic number* technique to determine file format, extension is only a convention
  - format inferred by inspecting the contents of the file, often first few bytes
  - eg. `#!/bin/bash` as the first line → file contains a bash script, `%PDF` → pdf file, ...

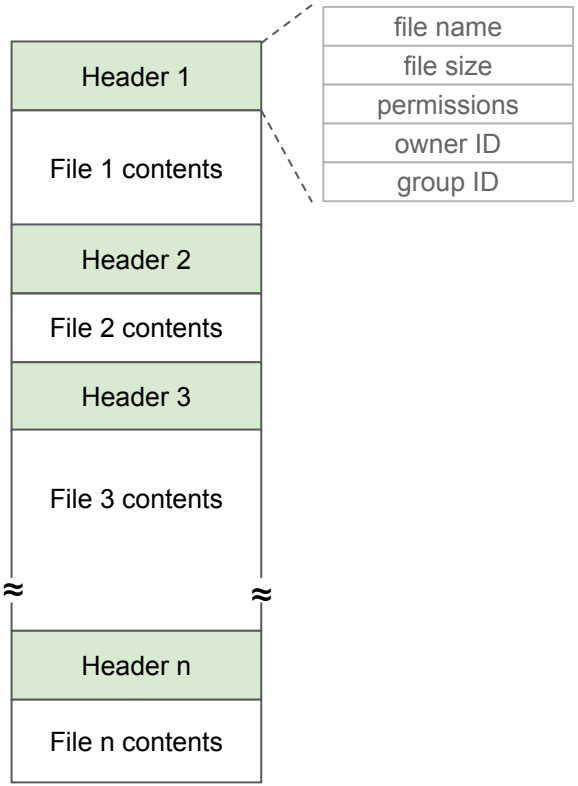
```
$ file file.c file /dev/hda .
file.c:   C program text
file:     ELF 32-bit LSB executable
/dev/hda: block special (3/0)
.:        directory
$ man file
$ man magic
```

looks at the

Example file formats



executable file



archive

```
#include <stdio.h>

main()
{
    printf("Hello World");
}
```

#	i	n	c	l	u	d	e
	<	s	t	d	i	o	.
h	>	\n	\n	m	a	i	n
(	)	\n	{	\n			
	p	r	i	n	t	f	(
"	H	e	l	l	o		W
o	r	l	d	"	)	;	\n
}	\n						

text file

File operations

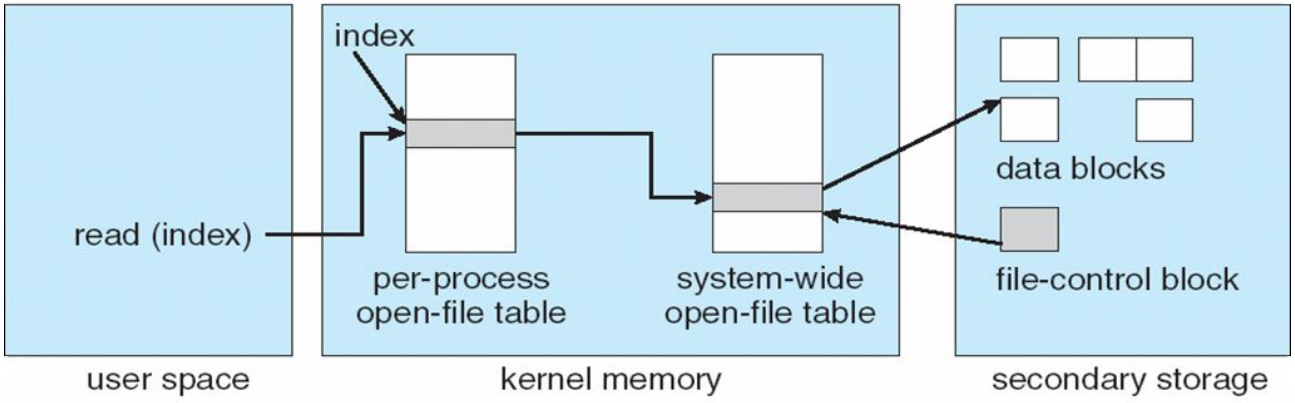
Most systems allow the following operations on regular files:

- create — empty file is created, with no data
- delete* — files can be deleted to free up disk space
- open — before using a file, a process must open it. OS can fetch and cache file attributes, such as list of disk addresses into main memory, for rapid access on subsequent calls
- close — free up space in memory associated with open file, flush unwritten data
- read — read contents of an opened file from current position
- write — overwrite data of an opened file at current position
- append — write new data at the end of file, results in file growing, usually implemented via write
- seek — change current position, affecting subsequent reads/writes
- get attributes* — eg. size
- set attributes* — eg. permissions
- rename* — change filename

* *operation could be on a directory rather than a file*

Open files

- OS needs to manage open files, and allow fast access to data in these files
- to this end OS keeps several data structures in memory
- open-file table: tracks open files, per-process tables, and a system-wide table
  - file pointer: pointer to last read/write location, per process
  - file-open count: number of times a file is open – to allow removal of data from open-file table when last processes closes it, system wide
  - permissions, pointer to file contents, system wide



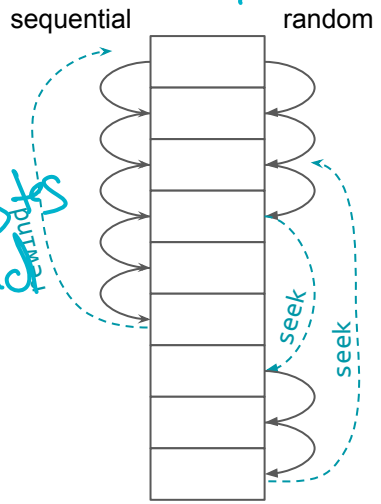
More on in-memory structures

- OS keeps various bits of information related to filesystems in various data structures (in memory), to make FS management possible as well as to improve performance
- examples:
  - system-wide open-file table: entry for each open file, eg. starting block #
  - per-process open-file table: eg. pointers into system-wide open-file table + file pointer
  - mount table: information about each mounted volume
  - buffer cache: caches FS blocks, to reduce the number of raw reads/writes to files, to speed up access to frequently accessed directories, etc.

Sequential and random file access

- two general types of accessing files: sequential & random
  - apply to both reading and writing
- **sequential access** - most common
  - bytes in the file are accessed sequentially, from beginning to end
  - no skipping, no out-of-order access, although files usually can be rewound
  - eg. `open, read, read, read, rewind, read … close`
- **random access**
  - can access any byte in any order
  - usually implemented using `seek(position)` API
  - eg. `open, read, read, read, seek, read, read, seek, read, … close`

always want to do sequential access
↳ most bytes per second



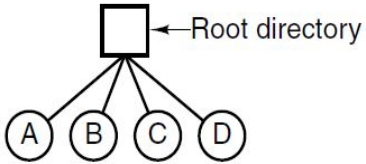
↳ uses seek to go anywhere in the file
performance is terrible

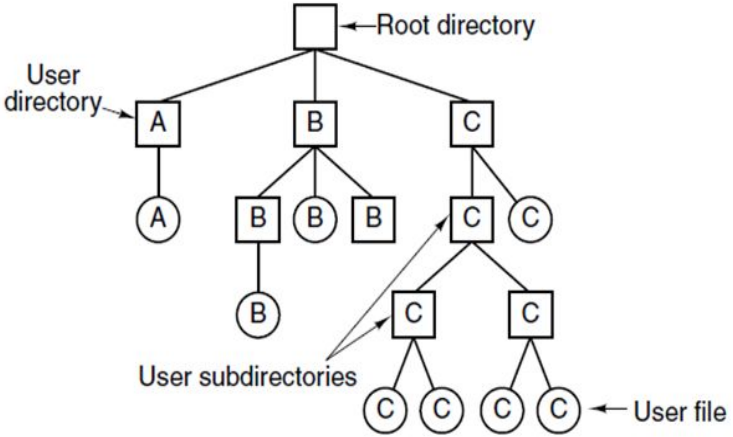
Directories

- a filesystem is a collection of files,
  where files are the basic units in a filesystem
- to help us with organizing files, we use the concept of directories
  - directories allow us to organize files hierarchically in a directory structure,
    a tree structure with one or more levels
  - root node of the tree is the root directory
  - internal nodes = directories, leaf nodes = files
  - path in a tree = filepath

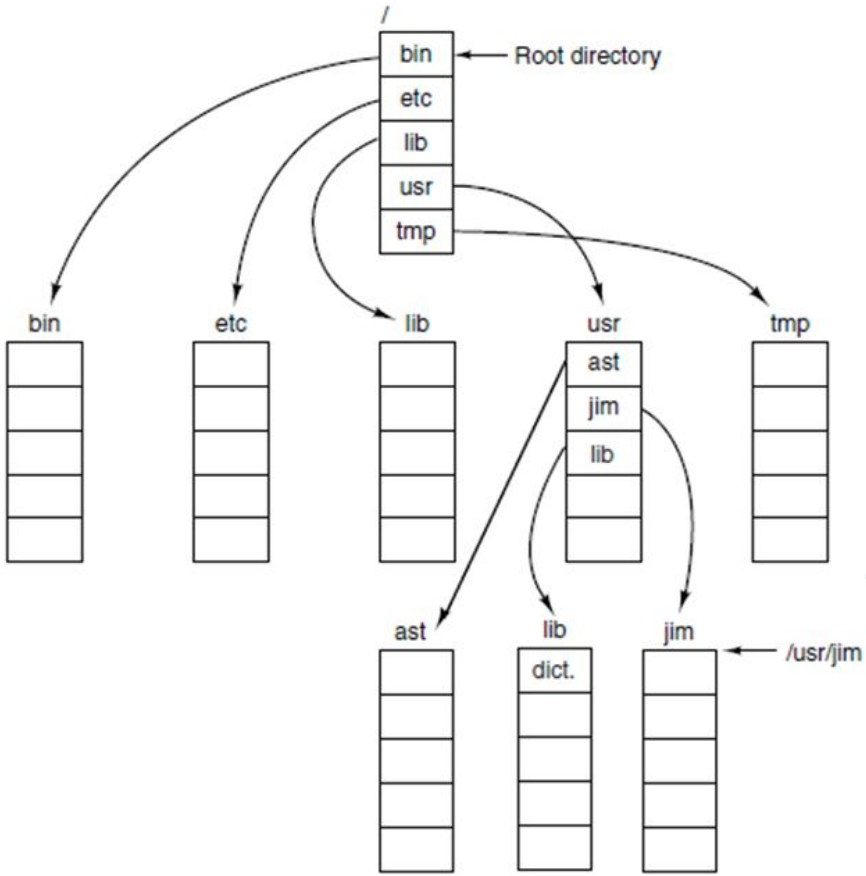
Directory

- a directory is usually implemented as a special file
- directory file contains directory entries (dentry)
  - dentry contains file attributes, such filename, size, etc.
  - dentry can represent a file or a directory (subdirectory)
  - if subdirectories not allowed → single-level directory system (limited use, eg. cameras)
  - if subdirectories allowed → hierarchical directory system (widespread use)





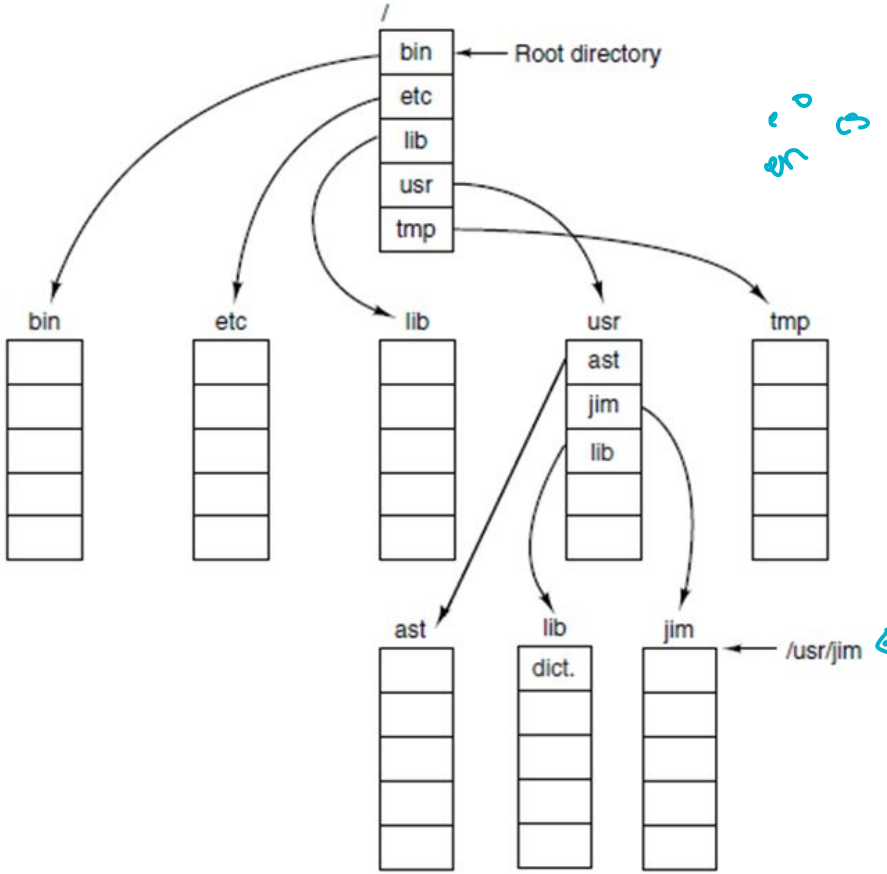
Pathnames in a UNIX directory tree



- path separator: `/` (forward slash)
- pathname: `dir1/dir2/.../dirn/filename`
- root directory path: `/`
- an absolute path name begins at root, eg: `/usr/jim`
- a relative path name defines a path from the current directory, eg. `./banker` or `../../bin/cat` or `1.txt`
- every process has a working (current) directory
- can be changed using `chdir()` sys. call:

```
int chdir(const char *path);
```

Pathnames in a UNIX directory tree



- every directory has at least 2 entries:
  - pointer to current directory: `.` (dot)
  - pointer to parent directory: `..` (dotdot)
- dot and dotdot entries:
  - cannot be deleted
  - they are just pointers
  - directory containing only . and .. entries is considered empty
- weird but true example:

`/usr/jim`

`/./etc/../lib/./../usr/pavol/../jim`

`../../../../../../../usr/jim`

all refer to the same file

Directory operations in UNIX

- create – an empty directory is created (with . and .. entries)
- delete – only empty directories can be deleted ('.' and '..' entries do not count)
- opendir – analogous to open for files
- closedir – analogous to close for files
- readdir – returns the next entry in an open directory
- rename – just like file rename
- link – technique that allows a file to appear in more than one directory
- unlink – a directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. In UNIX, the system call for deleting files (discussed earlier) is, in fact, **unlink**.

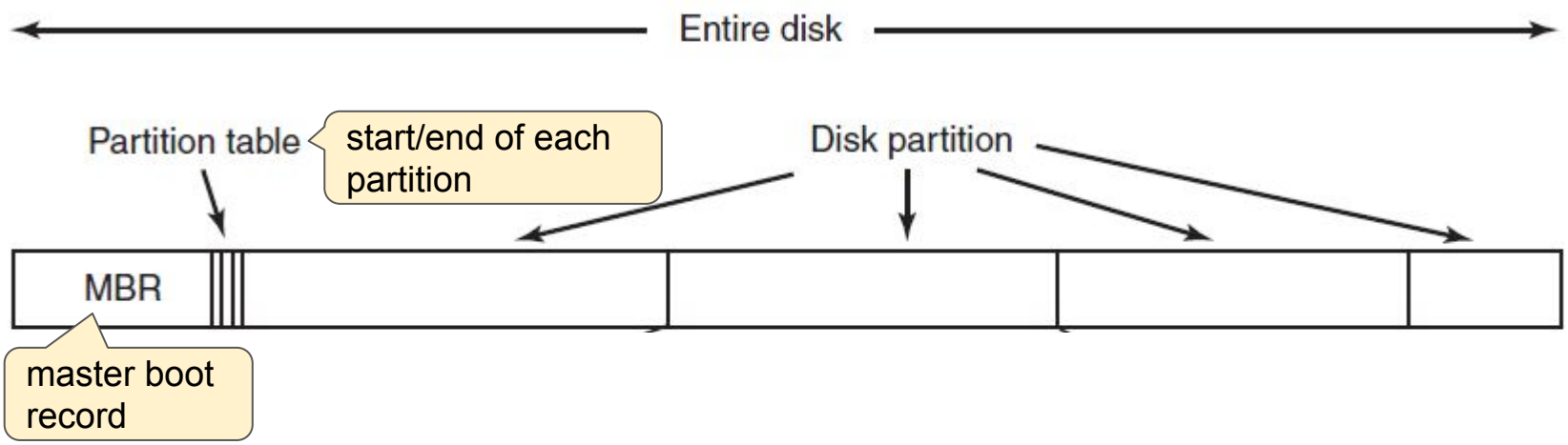
so you don't delete anything important yourself

Can always rename or delete a directory even if someone else is working on it ↳ on UNIX

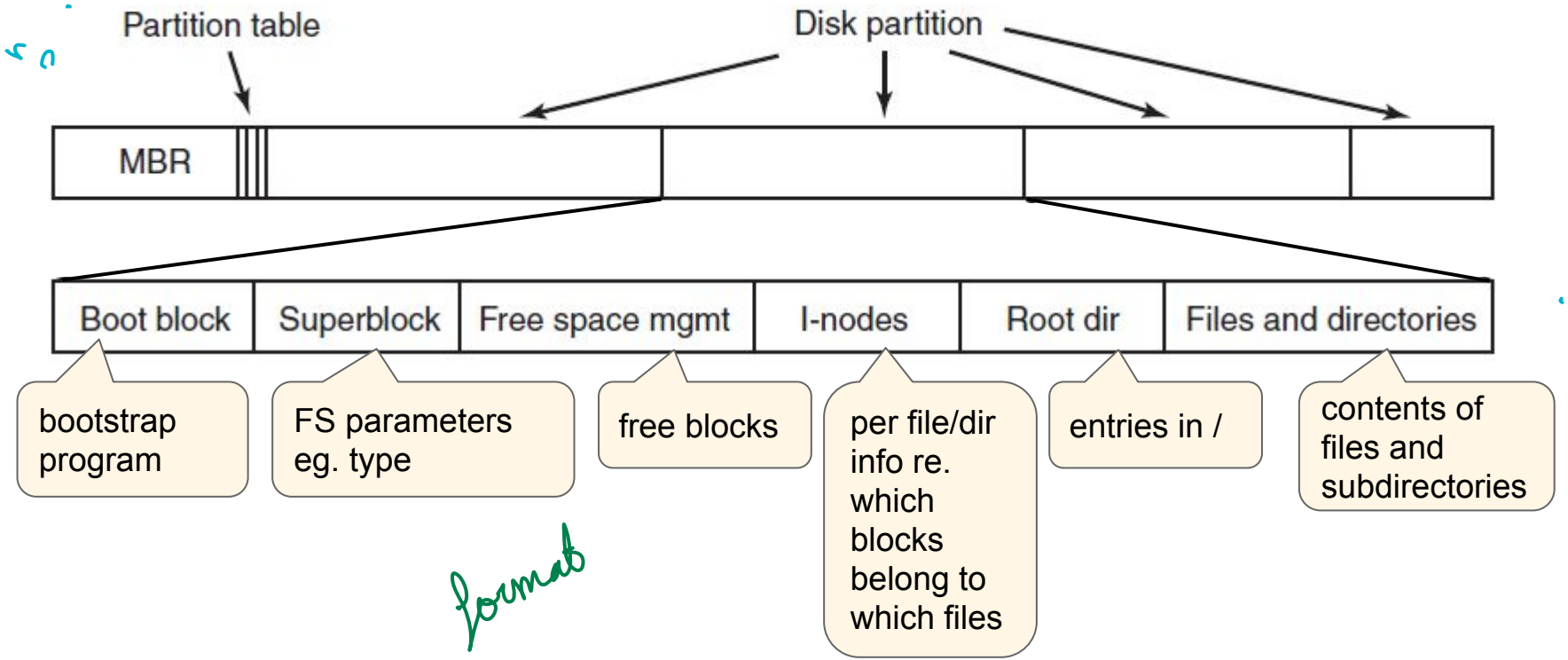
Disk partitions

- a physical disk can be subdivided into separate regions, called partitions
- partition is an abstraction, creating the illusion there are more disks
- OS can manage partitions independently, as if they were separate disks
- information about partitions is stored in a partition table

Partitions are continuous



Typical filesystem layout



Partitions and mounting

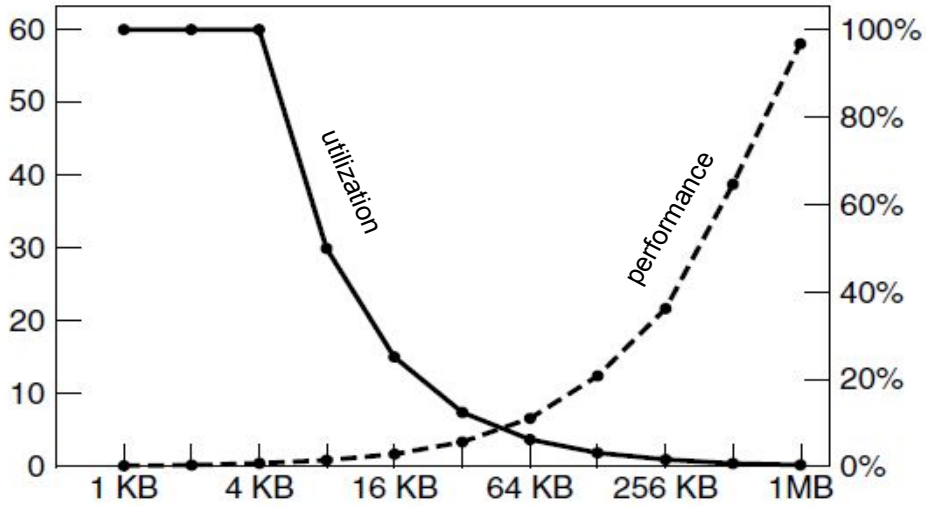
- partition can be:
  - formatted to contain a filesystem, it must be mounted to access
  - or it can be a raw partition (unformatted)
- root partition with a filesystem contains the OS
  - mounted at boot time as root directory '/'
- other partitions can hold other OSes, other file systems, or be raw
  - can mount automatically during boot, or manually after booting
- at mount time, file system consistency is checked
  - Is all metadata correct?
  - If not, fix it, try mounting again
  - If yes, add to mount table, allow access

can be slow because it does the checking

goes around this by checking the errors every so boots or so

Filesystem blocks

- nearly all filesystems split files up into fixed-size blocks
  - must round file size up to the nearest multiple
  - most filesystems suffer from internal fragmentation
- filesystem block size is usually a multiple (2^n) of the underlying disk block size
- FS blocks of one file not necessarily adjacent
  - fragmented file
  - seek time performance issues
- performance and space utilization are inherently in conflict



Virtual File Systems

OS doesn't care where your files are

API = App programming interface

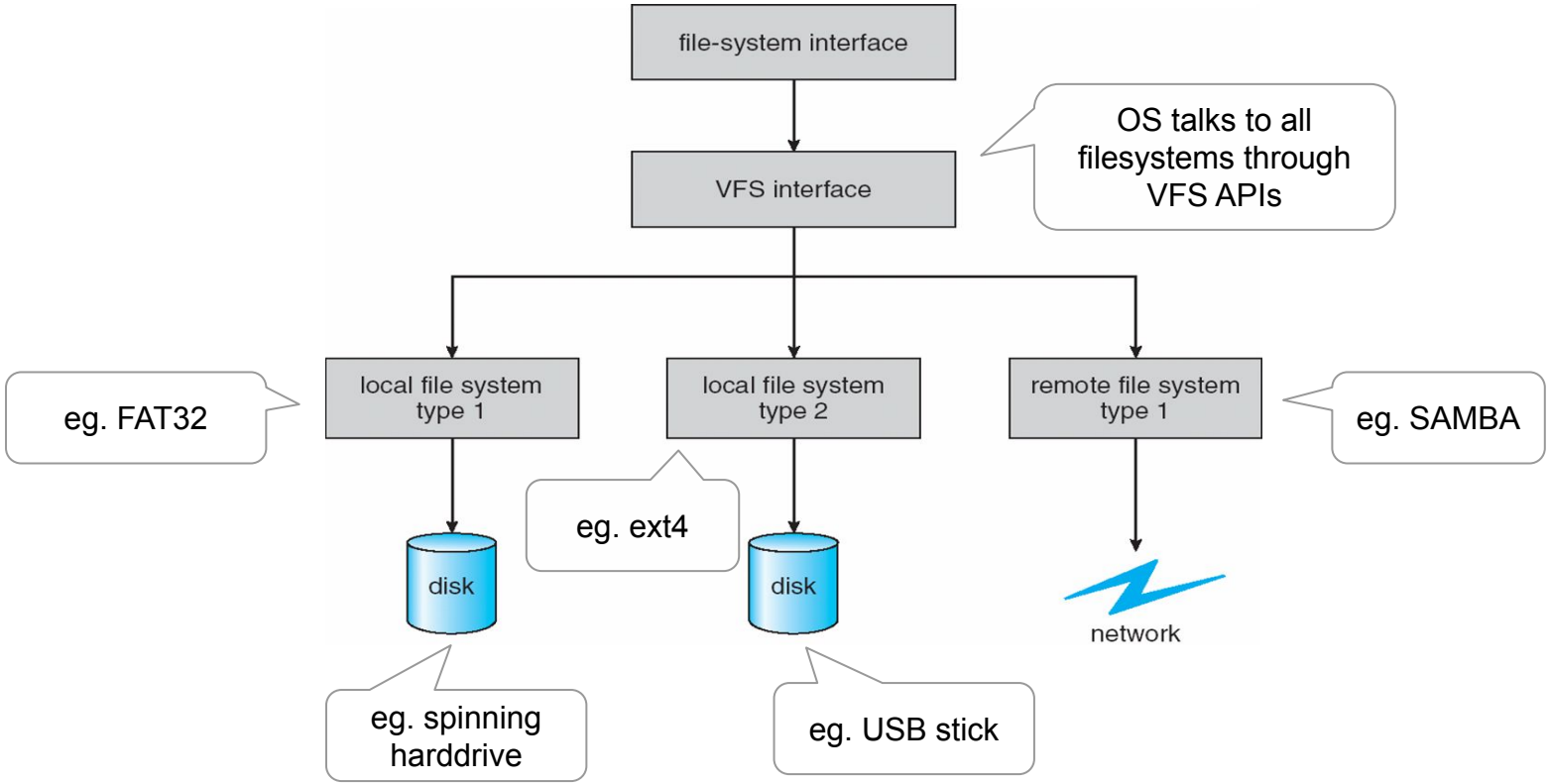
- Virtual File Systems (VFS) provides an 'object-oriented' way of implementing file systems (Linux)
- VFS allows the same system call interface (the API) to be used for different file systems
- VFS separates generic file-system operations from implementation details
- VFS implementation can be disk filesystem, RAM FS, archive FS, or even network based FS ...
- VFS dispatches operation to appropriate filesystem implementation routines

Comp Sci lab

fast → but once computer shuts down → RIP

Virtual File Systems

- OS accesses all filesystems through the same VFS interface



Virtual File System Implementation

- for example, Linux has four object types:
  - inode, file, superblock, dentry
- VFS defines set of operations on the objects that must be implemented
  - every object has a pointer to a function table
  - function table contains addresses of routines that implement that function on that object
  - example:
    - `int open(...)` — open a file
    - `int close(...)` — close an already-open file
    - `ssize_t read(...)` — read from a file
    - `ssize t write(...)` — write to a file
    - `int mmap(. . .)` — memory-map a file

need this to implement a new file system (just the basics)

- a developer of a new FS only needs to implement VFS API
- then the FS can be mounted by Linux

Directory implementation

- linear list of file names with pointer to the file blocks
  - simple to program
  - but O(n) search time
  - could be maintained in sorted order eg. using B+ tree, then O(log n) search
- hash table – linear list with hash data structure
  - potentially O(1) search time
  - needs good hash function to limit collisions, and the 'right' size table
  - big table → lot of wasted space, small table → too many collisions
  - dynamically resizable hash table could be used to solve this

- Linux (ext3/4) - use special data structure called htrees

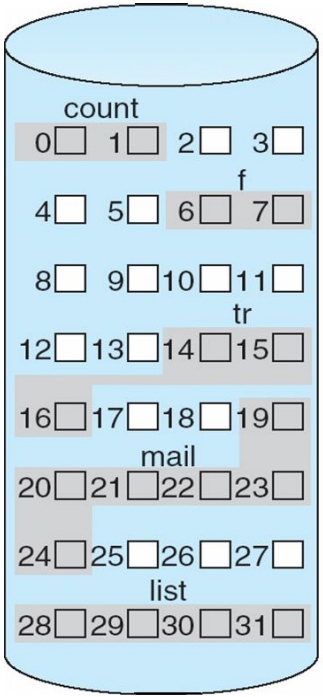
File allocation methods

- a file allocation method refers to how disk blocks are allocated to files
- we will discuss:
  - contiguous allocation
  - linked allocation
  - indexed allocation

Contiguous allocation

- contiguous allocation – each file occupies a set of contiguous blocks
- results in best performance in most cases
- simple – only starting location (block #) and length (number of blocks) are required
- problems include
  - finding space for file,
  - either knowing file size at creation, or complications with growing a file
  - external fragmentation after file deletion,
  - need for compaction off-line (downtime) or on-line (reduced performance)
    - aka defragmentation
- not very common
- useful for tapes & read-only devices such as CD-ROMs

Contiguous allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- mapping from logical to physical address:

  assuming block size is a power of 2

  logical address:

q	r

  q = upper bits

  r = lower bits

  physical address computation:

  block = q + "address of first block"

  displacement within block = r

Linked allocation

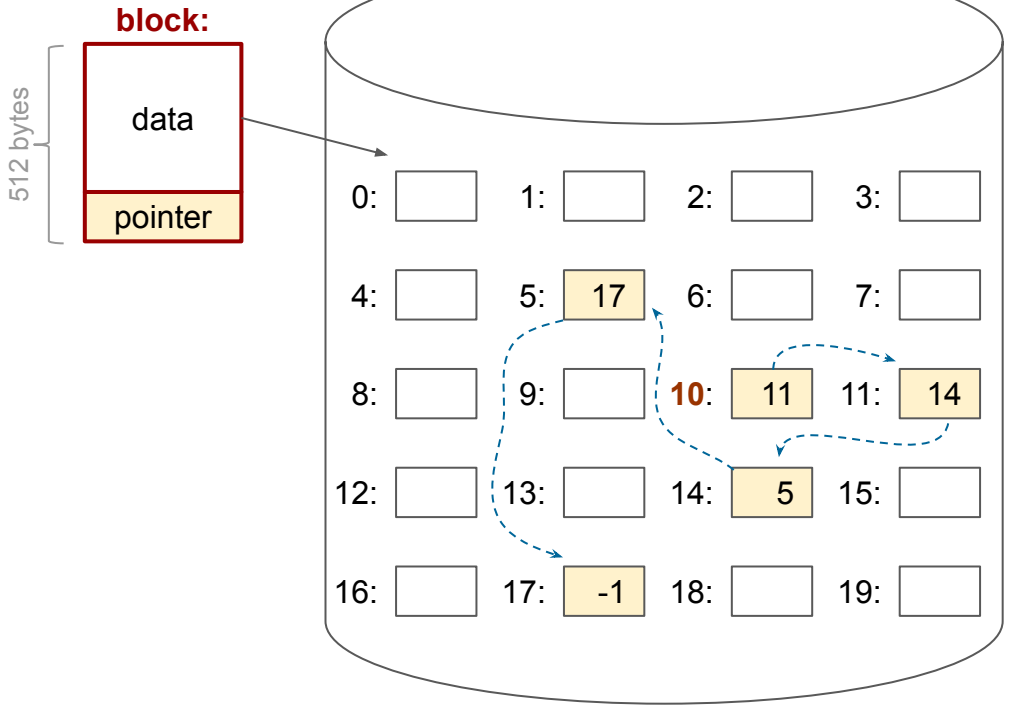
- in linked allocation each file is stored in a linked list of blocks
- each block contains file content, plus a pointer to the next block
- file ends at a block with NULL pointer
- no external fragmentation → no compaction needed
- separate free space management needed - eg. linked list of free blocks
- reliability can be a problem – imagine losing a block due to disk failure
- major problem: locating a block can take many I/Os and disk seeks
  - logical address to physical address mapping requires traversing the list
  - we could cache the 'next' pointers, but would still need to read entire file first
- we could improve efficiency by clustering blocks into larger groups but that increases internal fragmentation

Linked allocation example

appending info to a file would be painfully slow

we need the remainder

So the size isn't a multiple of the file size



Example directory entry:

filename	start block	size
test.txt	10	2100

contents of `test.txt` spread over blocks:

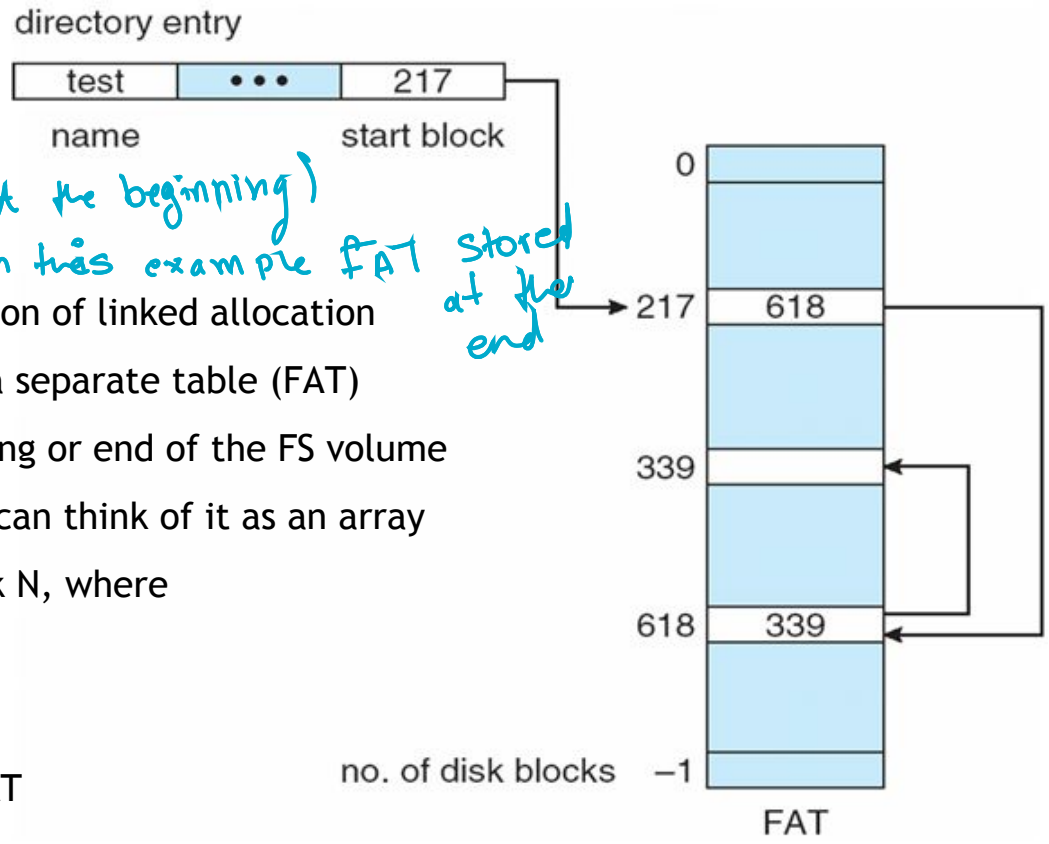
10, 11, 14, 5, 17

Why do we need 'size' in dentry?

because 5 * 512 != 2100
files can have arbitrary sizes

File Allocation Table (FAT)

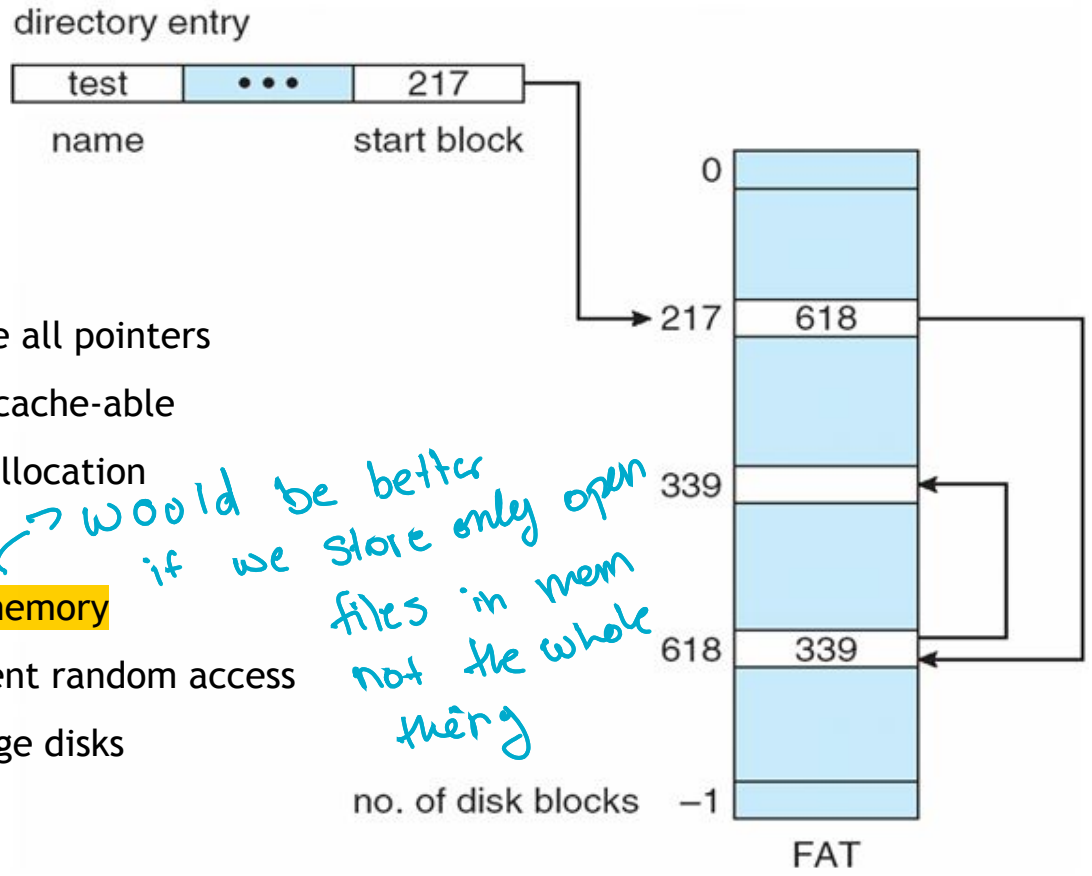
Same idea as linked but all the pointers are stored together in a small portion of the disk (usually at the beginning) but in this example FAT stored at the end



- FAT (File Allocation Table) is a variation of linked allocation
  - all 'next' pointers are stored in a separate table (FAT)
- FAT can be located eg. at the beginning or end of the FS volume
- FAT is indexed by block number, you can think of it as an array
  - `fat[N]` = next pointer for block N, where

    "-1" could denote NULL ptr
- one FAT for the entire disk
- directory entry contains index into FAT

File Allocation Table (FAT)



- much like a linked list, but because all pointers stored together, FAT is faster and cache-able
- easier random access than linked allocation
- issues with FAT:
  - the entire table must be in memory at all times to achieve efficient random access
  - table can be quite big for large disks

would be better if we store only open files in mem not the whole thing

Indexed Allocation (inodes)

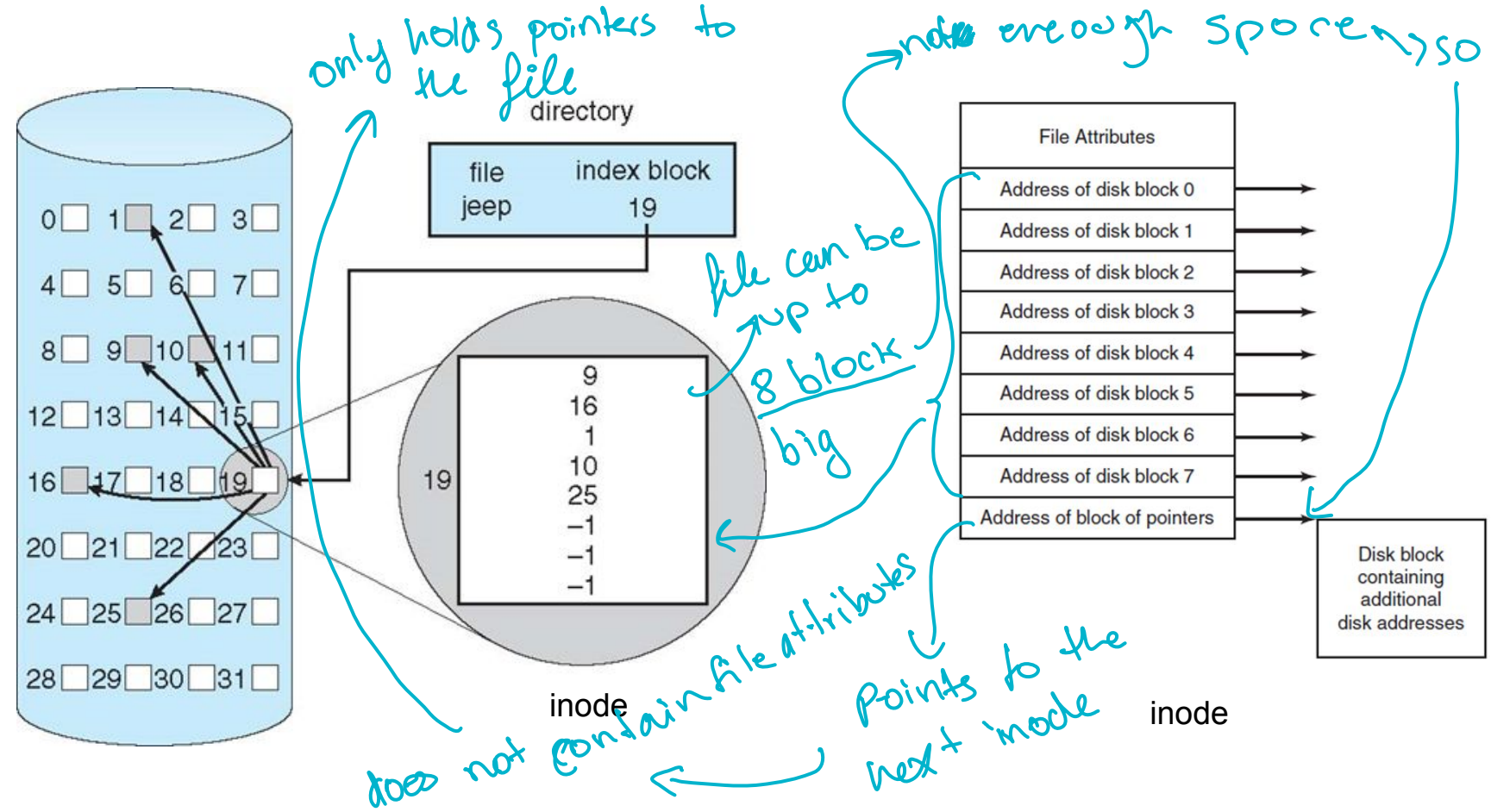
- basic idea behind indexed allocation is to:
  - store a per-file FAT-like structure
  - then we don't need to cache pointers for all files, only the open files
- each file has its own index block(s), called inodes
- an inode block contains:
  - direct pointers to blocks with file contents, or more indirect pointers to even more inodes
  - optionally, inode can contain various file attributes:
    - file size in bytes, device ID, owner, permissions, timestamps, link count, …
    - however, inode **does not** contain a filename
- dentry is used to associate filename with the inode
  - dentry = filename + pointer to inode
  - possible to have different filenames associated with the same inode
    - called hard links

contain meta data (file size, direct pointers to blocks of the file)

we only store it when the files are actually open

↳ two different file pointing to the same inode

Example of Indexed Allocation



inodes in Linux (ext2)

- example: block size 1KB, block address 4 bytes
- single inode with 12 direct entries

  → max file size 12KB
- if we add a single indirect pointer to inode:

  1KB block can have 1KB/4B=256 entries

  → max file size 256+12 blocks = 268KB
- adding double indirect pointer as well:

  or 256 blocks each with 256 addresses

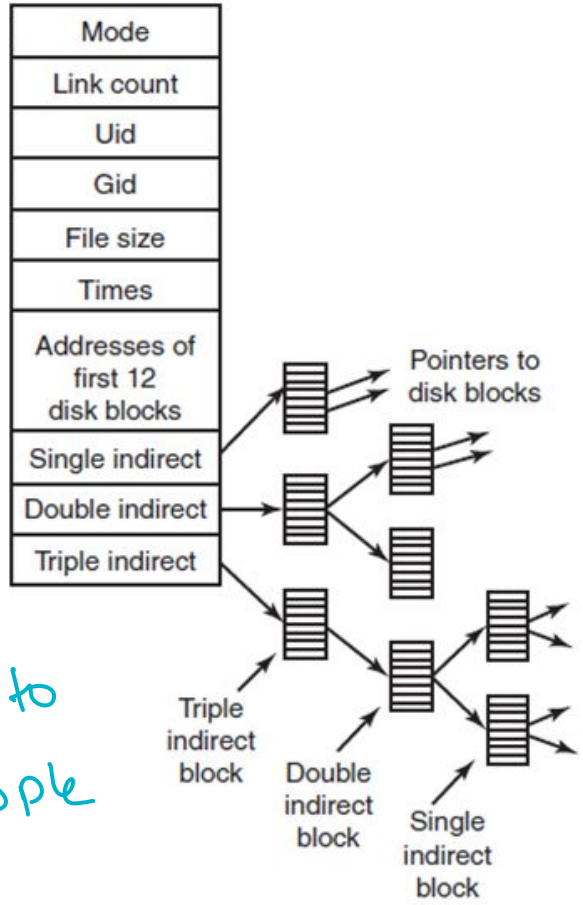
  → max file size ~ 2^{16} blocks ~= 64MB
- adding triple indirect pointer:

  → max file size ~ 2^{24} blocks ~= 16GB
- ext3 max file size = 2TB
- ext4 max file size = 16TB (using 48bit addresses)

need more space

need more space

↳ can go to quadruple



inodes

- advantages:
  - random access reasonable — only need to keep the inodes for opened files in memory
  - file size is not limited (practically)
  - files can have holes
- disadvantages:
  - at least one additional block is required for each file

Hard link vs soft link

1. create file.txt

```
$ echo "Hello" > file.txt
```

2. create hard link file1.txt that points to file.txt

```
$ ln file.txt file1.txt
```

a hard link points to the same inode

if we delete file.txt, file1.txt will still work

file.txt and file1.txt are indistinguishable

3. create soft link file2.txt that points file.txt

```
$ ln -s file.txt file2.txt
```

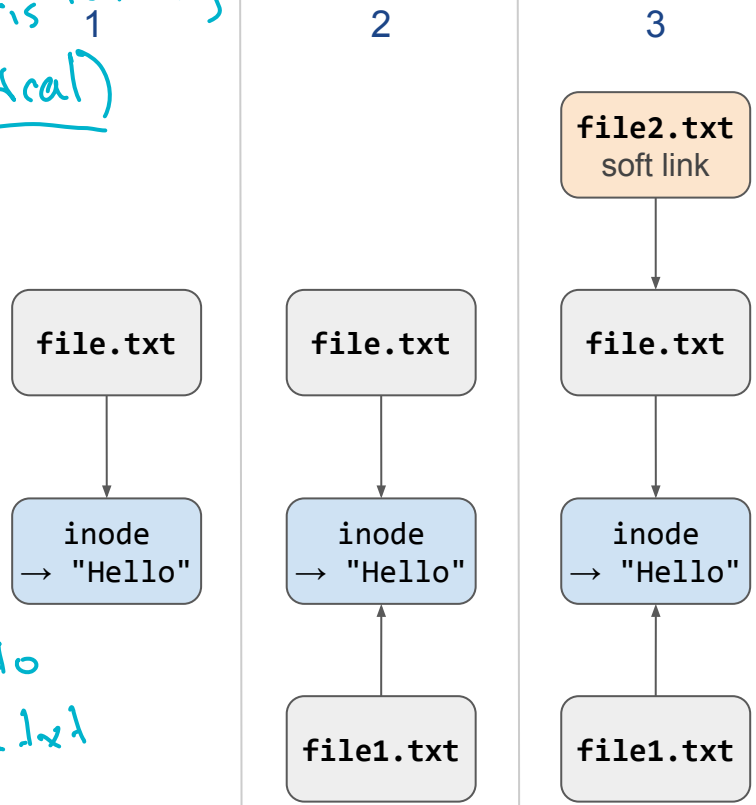
soft link points to a filename

if we delete file.txt, file2.txt will be broken

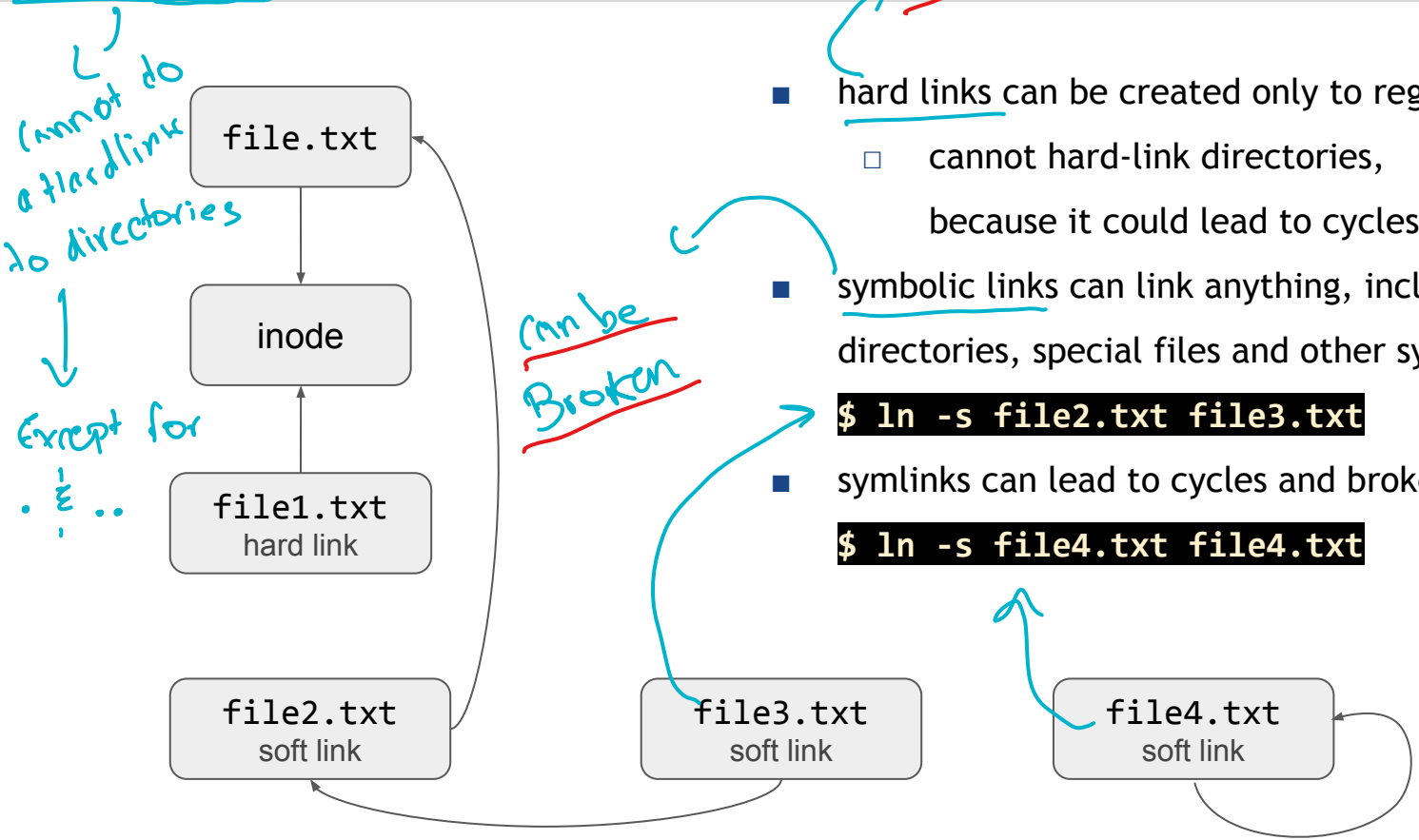
file1.txt is not pointing to file.txt but to the thing file.txt is pointing to (they are identical)

file2.txt will point to file.txt

file.txt will become a special file



Hard link vs soft link



- hard links can be created only to regular files
  - cannot hard-link directories,
    because it could lead to cycles in FS
- symbolic links can link anything, including directories, special files and other symbolic links

```
$ ln -s file2.txt file3.txt
```

- symlinks can lead to cycles and broken links

```
$ ln -s file4.txt file4.txt
```

Performance

- newer CPUs (2016) can do ~300,000 MIPS
- typical disk drive (7200RPM) can do about 100 IOPS
  - CPU can do ~3 billion instructions during one disk I/O
- fast SSD drives can deliver ~100,000 IOPS
  - still ~3 million instructions during one disk I/O
- expensive SSD arrays can deliver ~10,000,000 IOPS
  - still about 30,000 instructions during one disk I/O

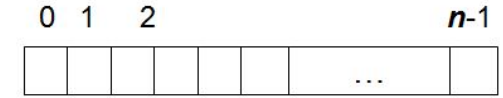
- important to try to minimize the number of I/O operations
  - try to group and combine reads/writes

300 billion operations per second

* minimize IO and when you have to do it try to group it and do as much of it at one time together

Free space management - bitmaps

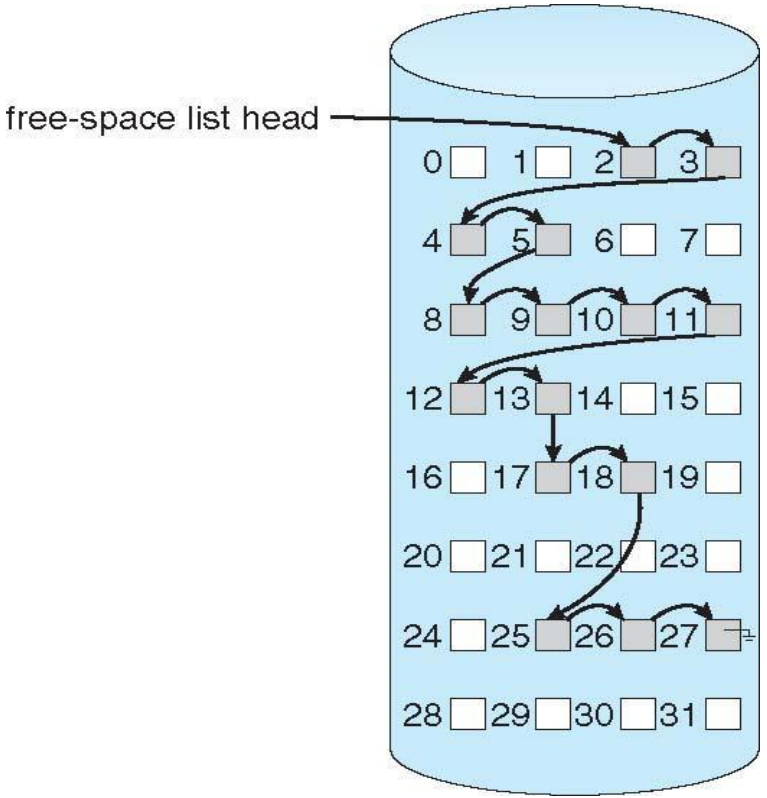
- file systems maintain free-space list to track available blocks
- can be implemented as a bit vector or bitmap
- OS can reserve some blocks for the bitmap
- example
  - block size = 4 KiB = 2^{12} bytes
  - disk size = 1 TiB = 2^{40} bytes
  - total number of blocks = $2^{40}/2^{12} = 2^{28}$ blocks
  - we need 2^{28} bits in bitmap = 2^{25} bytes = 32 MiB bitmap, or 2^{13} reserved blocks
  - if using clusters of 4 blocks instead → only 2^{11} reserved blocks
- cons: requires searching the bitmap to find free space, wastes some blocks
- pros: fairly straightforward to obtain contiguous blocks



$$\text{bit}[i] = \begin{cases} 1 \Rightarrow \text{block}[i] \text{ free} \\ 0 \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Free space management - linked list

- linked free space list (free list)
  - all free blocks are linked together
  - pointers stored inside the blocks
- pros: no waste of space
- cons: cannot get contiguous space easily



Free space management - linked list

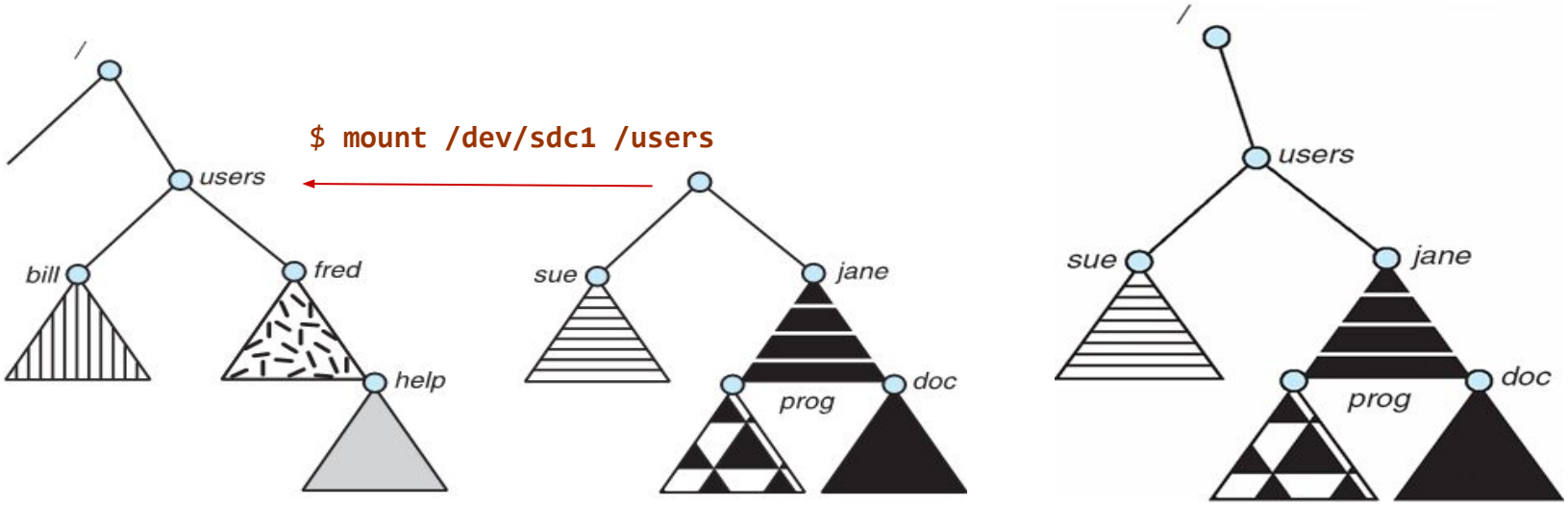
- grouping
  - instead of storing just one pointer, utilize the space of the entire free block
  - store addresses of next n-1 free blocks in first free block, plus a pointer to next block that contains more free-block-pointers (like this one)
- counting
  - takes advantage of the fact that space is frequently contiguously used and freed
  - so keep address of first free block plus the count of following free blocks
  - free space list then has entries containing addresses and counts
- space maps
  - divides device space into metaslab units, each representing a chunk of manageable size
  - within each metaslab a counting algorithm is used to keep track of free space

File locking

- provided by some operating systems and/or file systems
  - similar to reader-writer locks
  - shared lock similar to reader lock – several processes can acquire concurrently
  - exclusive lock similar to writer lock
- mediates access to a file to multiple processes during `open()`
- types:
  - mandatory – access is denied depending on locks held and requested
  - advisory – processes can find status of locks and decide what to do

File System Mounting

- a filesystem must be mounted before it can be accessed
- OS boots with (essentially) empty root filesystem
- other filesystems are later mounted into it, during or after boot
- all mounted filesystems appear as part of one big filesystem



root filesystem before mounting filesystem 2

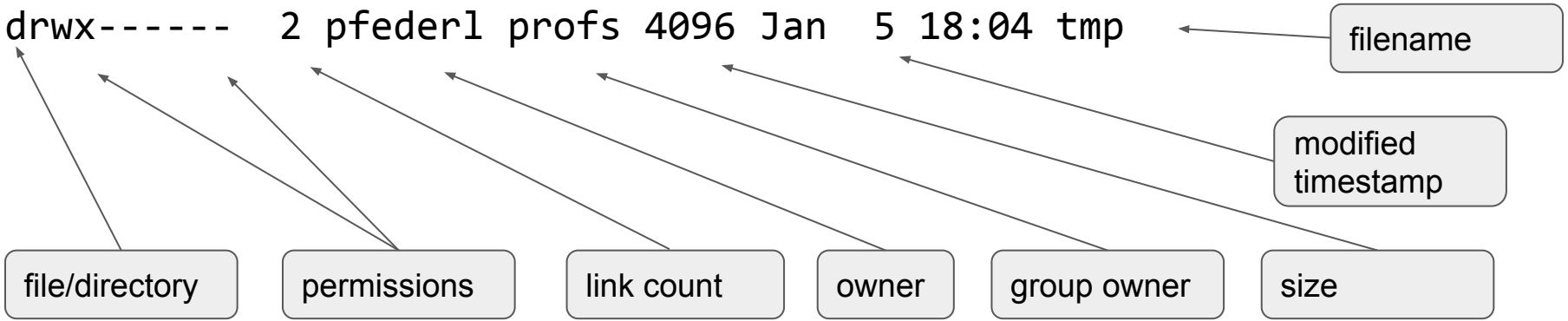
filesystem 2 on /dev/sdc1

root filesystem after mounting filesystem 2

Unix File System & Permissions

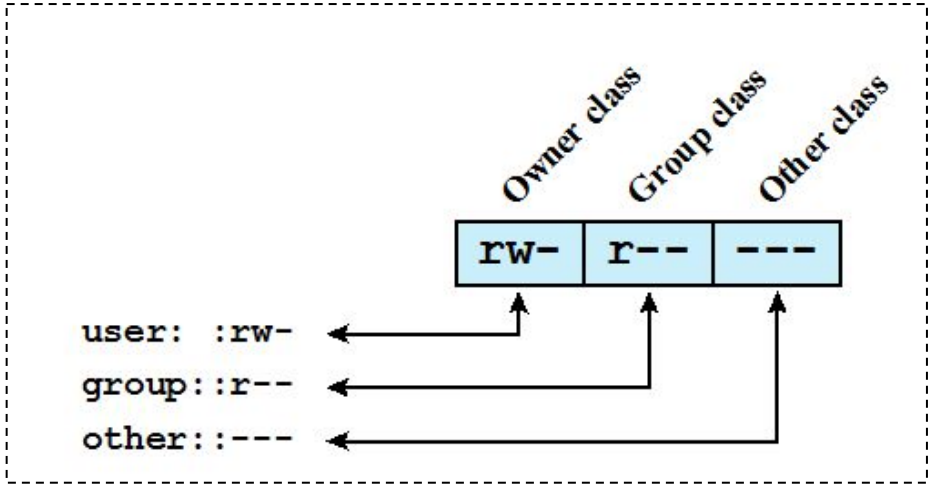
- every file is owned by a user and a group
- permissions usually displayed in compact 10-character notation

```
$ ls -l /home/profs/pfederl
-rw-r-----  1 pfederl profs  134 Oct 13 13:02 test.py
drwx------  2 pfederl profs 4096 Jan  5 18:04 tmp
```



Unix File System

- 9 permission bits per file: specify **R**ead, **W**rite, and e**X**ecute permission for the owner of the file, members of the group and all other users (aka world)
- The owner ID, group ID, and protection bits are part of the file's inode



Examples - permissions for files

`-rw-r--r--`	read/write for owner, read-only for everyone else
`-rw-r-----`	read/write for owner, read-only for group, forbidden for everyone else
`-rwx--x--x`	read/write/execute for owner, execute-only for everyone else
`-r--r--r--`	ready-only for everyone
`-rwxrwxrwx`	read/write/execute for everyone (bad idea)
`----rwxrwx`	yes it’s possible, owner has no rights, everyone else does...

Examples - permissions for directories

- permission bits are interpreted slightly differently for directories
- **read** bit allows listing of file/directory names
- **write** bit allows creating and deleting files in directory
- **execute** bit allows entering the directory and getting attributes of files in the directory
- not all combinations make sense: eg. read without execute

`drwxr-xr-x`	all can enter and list the directory, only owner can add/delete files
`drwxrwx---`	full access to owner and group, off limits to world
`drwx--x--x`	full access to owner, while group & others can access only known files
`drwxrwxrwx`	anyone can do anything

Linux File System

permission check algorithm for given user, filepath

step 1:

make sure all parent directories in path have appropriate execute permissions

step 2:

```
if file.owner == user then
     use file.userPermissions
else if file.group in user.groups then
     use file.groupPermissions
else
     use file.worldPermissions
```

Linux File System

- Set user ID (SetUID) bit, only on executable files
  - system temporarily uses rights of the file owner in addition to the real user's rights when making access control decisions
  - enables privileged programs to access files/resources not generally accessible
  - eg. `passwd`
- Set group ID (SetGID) bit
  - on executable files → similar effect to SetUID but for groups
  - on directories → new files/subdirectories will inherit the group owner
- Sticky bit (12th bit)
  - When applied to a directory it specifies that only the owner of a file in the directory can rename, move, or delete that file.
  - Usually set on `/tmp` and `/scratch` or similar directories.

Root (superuser, UID = 0)

- is exempt from usual access control restrictions
  - has system-wide access
  - dangerous, but necessary, and actually OK with good practices
- how to become root:
  - **`su`** (requires root password)
    - changes home dir, PATH and shell to root, leaves environment variables intact
  - **`su -`**
    - logs in as root
  - **`su - <user>`**
    - become someone else <user>
  - **`sudo <command>`** (requires user password)
    - run one command as root — recommended way, leaves an audit trail
    - what does "`sudo su -`" do?

Changing permissions

- permissions are changed with **`chmod`** or via a GUI
- only the file owner or root can change permissions.
- if a user owns a file, the user can use **`chgrp`** to set file's group to any group of which the user is a member
- root can change file ownership with **`chown`** (and can optionally change group in the same command)
- `chown`, `chmod`, and `chgrp` can take the -R option to recursively apply changes through subdirectories.

Changing Permissions Examples

`chown -R root dir1`	changes owner of dir1 to root, and recursively everything inside dir1
`chmod g+w,o-rwx f1 f2`	adds group write access to files f1 and f2, and removes all access to f1 and f2 for the world
`chmod -R o-rwx .`	removes access for the world to current directory and everything inside it (recursively)
`chmod u+rw,g+rw,u-x,g-x,o-rwx f1`	f1 will allow read/write to owner & group, everyone else will have no access
`chmod 660 f1`	same as above but makes you look “pro”
`chmod +x f1`	f1 will be executable to everyone
`chmod og-rwx f1`	disable all group/world access from f1

Limitations of Unix/Linux Permissions

- Unix standard/basic permissions are great, but not perfect
  - not expressive enough
  - eg. user 'bob' cannot **easily** give user ‘john’ read access to his files
- most Linux based OSes support POSIX ACLs
  - builds on top of traditional Unix permissions
  - several users and groups can be named in ACLs, each with different permissions
  - allows for much finer-grained access control
- each ACL is of the form `type:name:rwx`
  - **type** is user or group
  - **name** is user name or group name
  - **rwx** refers to the bits set
  - setuid, setgid and sticky bits are not possible

Linux Access Control Lists (ACLs)

- **getfacl** lists the ACL for a file
- **setfacl** command assigns ACLs to a file/directory
- any number of users and groups can be associated with a file
  - read, write, execute bits
  - a file does not need to have an ACL

```
$ ls -l proxy.py
-rw-rw-r-- 1 pfederl pfederl proxy.py
$ getfacl proxy.py
# file: proxy.py
# owner: pfederl
# group: pfederl
user::rw-
group::rw-
other::r--
```

```
$ setfacl -m u:bob:rw proxy.py
$ getfacl proxy.py
# file: proxy.py
# owner: pfederl
# group: pfederl
user::rw-
user:bob:rw
group::rw-
mask::rwx
other::r--
$ ls -l proxy.py
-rw-rw-r--+ 1 pfederl pfederl proxy.py
```

Default ACLs

- a directory can have an additional set of ACLs, called default ACLs
- default ACLs will be inherited by files & directories created inside directory
  - subdirectories inherit the parent directory's default ACLs as both their default and their regular ACLs
  - files inherit the parent directory's default ACLs only as their regular ACLs, since files have no default ACLs
- the inherited permissions for the user, group, and other classes are logically ANDed with the traditional Unix permissions specified to the file creation procedure

NTFS

- each file/directory has
  - an owner
  - zero or more ACEs (access control entries)
- ACE format: **<principal> <operation> (allow|deny)**
  - principal = user or group
  - operation = read, write, execute, full control, list, modify
- ACEs support inheritance
  - directory's ACEs can propagate to children
- similar to UNIX with ACLs, but
  - NTFS also supports 'deny' ACE entries, UNIX has only 'allow' ACL entries
  - NTFS file permission algorithm only checks the file's ACEs, UNIX checks entire path
  - NTFS is more expressive, but also more complicated
    - Prof: can Bob access this file in my directory?

Summary

- using filesystems
  - file structure, types, file access, attributes, operations
  - mount points, path names
- implementation of filesystems
  - vfs
  - file block allocation, FAT, inodes, links
  - free space management*
- UNIX permissions*

** if time permits*

Reference: 4.1 - 4.3.3 (Modern Operating Systems)

10.1 - 10.3 , 11.1 - 11.8 (Operating System Concepts)

Review

- Which file block allocation scheme suffers from external fragmentation?
  - Contiguous or Linked
- Describe the main difference between FAT and inode.
- After deleting a file, all hard links to the file will report an error when accessed.

  True or False
- After deleting a file, all soft links to the file will report an error when accessed.

  True or False

Questions?