File Systems

CPSC 457: Principles of Operating Systems
Winter 2024

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

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

- files – types, formats, attributes, operations, access methods
- directories – paths, operations
- file block allocation, contiguous, linked / FAT, inodes, free space management

Long-term Storage

Long term storage

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

1. persistent storage of large amount of information
2. information must survive termination of a process using it
3. access to information from multiple processes/threads
4. easy search & management

Disks without filesystems

Most storage devices, e.g. disks, are essentially arrays of **fixed-size blocks** (e.g. 512 bytes), and support two operations:

```
- data = read_block(block_number);
- write_block(block_number,data);
```

Similar to memory, but

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

Disks without filesystems

Most storage devices, e.g. disks, are essentially arrays of **fixed-size blocks** (e.g. 512 bytes), and support two operations:

```
- data = read_block(block_number);
- write_block(block_number,data);
```

Similar to memory, but

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

… but:

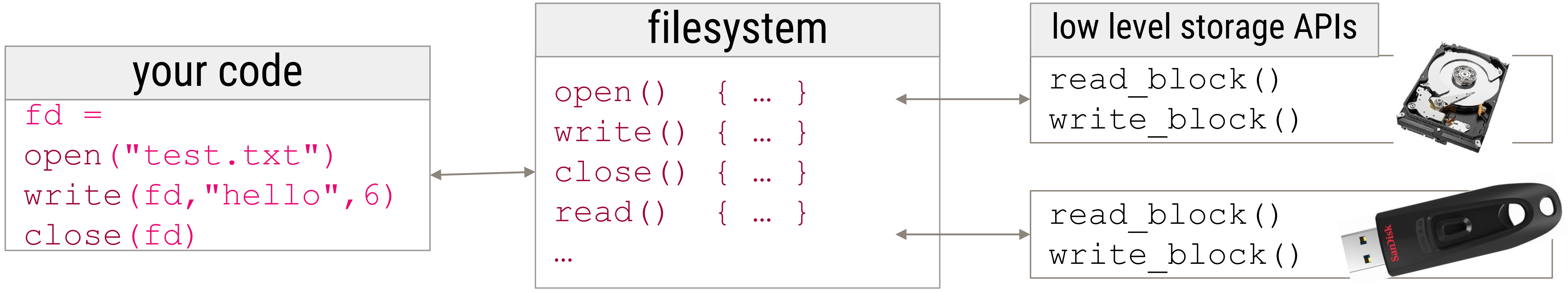
- How do we find information? Which blocks contain what data?
- How do we know which blocks are free?
- How do we keep one user from reading another user's data?
- How do programs/users share data?
- How do we (re)organize data?

Filesystems addresses all of the above via much more convenient and higher level APIs.

Filesystems

Filesystem

- filesystem is a higher level abstraction of storage, implemented using clever data structures, stored in storage but also in memory (when in use)
- filesystem acts as a library with nice APIs built on top of `read_block()` and `write_block()`



```
fd =
open("test.txt")
write(fd,"hello",6)
close(fd)
```

```
open()   { … }
write()  { … }
close()  { … }
read()   { … }
…
```

```
read_block()
write_block()
```

```
read_block()
write_block()
```

- it is common for OSes to support multiple filesystem implementations, and multiple filesystems can be usually mounted simultaneously
- basic unit of a filesystem is a file
- files can be usually grouped in directories and subdirectories

Files

- a file is presented to a process as a virtual sequence of bytes, which can be addressed individually
- OS maps individual addresses to physical blocks (which are often not contiguous)
  i.e. OS translates all read/write operations on files into appropriate `read_block`/`write_block` operations
- for example, consider appending a single byte to an existing file…
- OS (generally) does not care about the contents of files

File formats

UNIVERSITY OF CALGARY

Files

- what can be stored in a file?
  - anything, as long as it can be organized into a sequence of bytes

- file's creator decides:
  - on the contents of the file and their meaning
  - this determines the file format

- developers can then create even higher level abstractions
-

  e.g. treat file as a sequence of bits, numbers, text, …

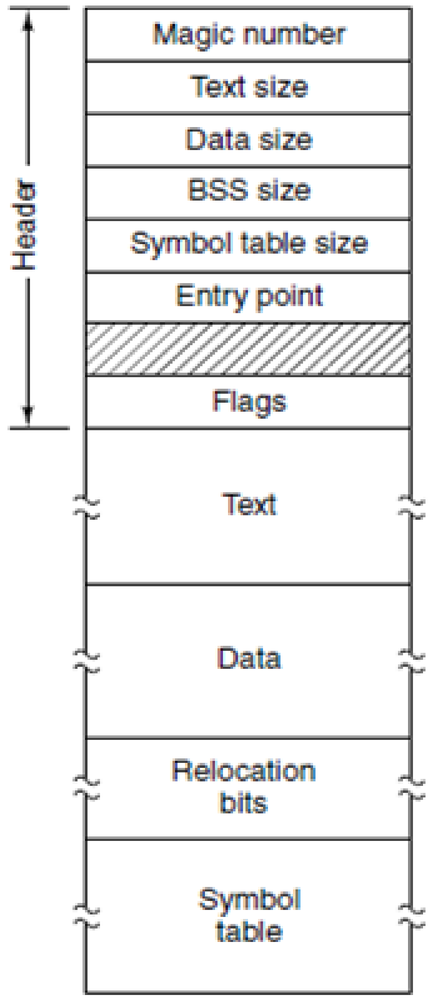
  source code, executables, images, movies, spreadsheets, …

Example file formats

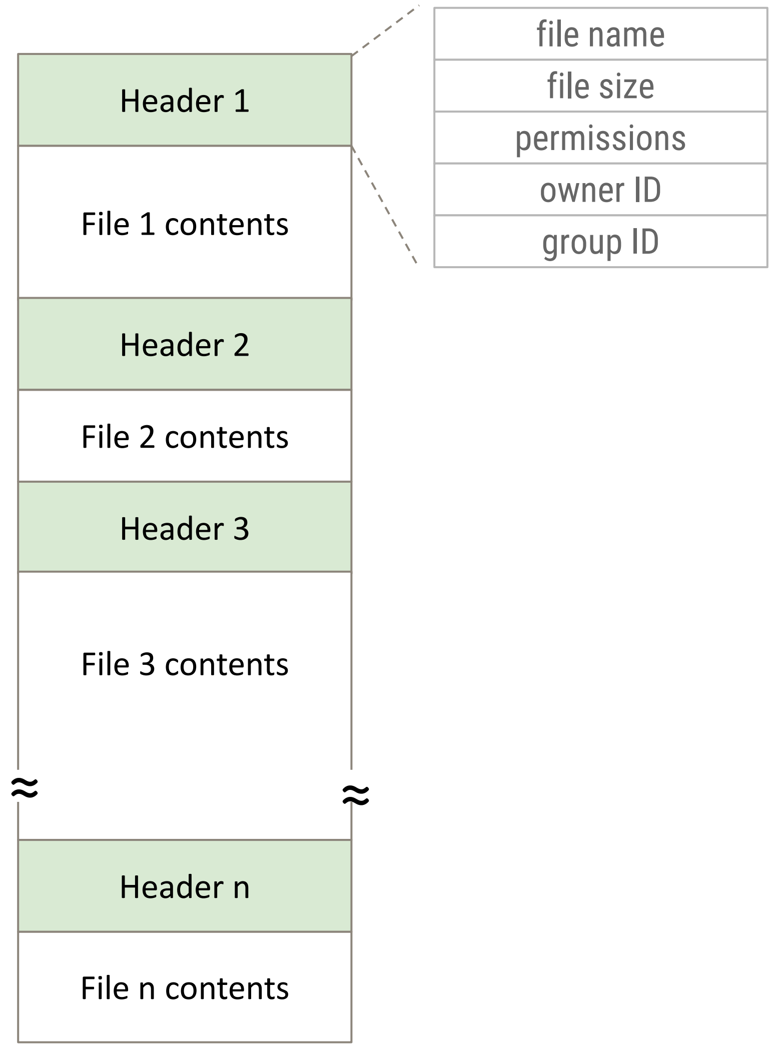
```
#include <stdio.h>

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



executable file



archive

File format (file type)

- if OS recognizes the file format, it can operate on the file in reasonable ways
  e.g. automatically selecting an appropriate program to open a file
- Windows uses file extension to determine file format, e.g. `.jpg`, `.xls`
- UNIX uses magic number technique to guess file format, and extensions are only a convention
  - format inferred by inspecting the contents of the file, often just the first few bytes
  - e.g. if `"#!/bin/bash"` is the first line, then file is a bash script
    or if `"%PDF"` are the first 4 bytes, then it is a 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
```

File Information

File attributes

- files have contents but also attributes (metadata)

- 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
  - size: size of the file in bytes
  - location: a pointer to the location of the file contents on the device
  - special type: needed for systems that support different file types, e.g. directory, link
  - time/date: time of creation/last modification/last access
  - ownership: identifies owner(s) of the file, e.g. user ID, group ID
  - protection information: access control information, e.g. read/write/execute

- attributes are often kept separate from file contents, e.g. in directory structure or inodes

- many variations, e.g. file checksum

File naming

- filenames are given to files at creation time, but usually can be changed later

- different file-naming rules on different systems, e.g.
  - maximum filename length
  - allowed/restricted characters
  - capitalization
  - filename extensions, enforced vs conventions

UNIVERSITY OF CALGARY

Special file types

- on top of regular files, most OSs also support special files
  regular files are the ones that users can create, such as .txt, .cpp, .jpg etc.
  - directories – special files for maintaining FS structure
  - character and block special files – that represent devices, e.g. `/dev/sdb0`
  - pseudo files – for accessing kernel routines and datastructures
    e.g. `/dev/random` and `/proc/cpuinfo`
  - links – pointers to other files
  - sockets, pipes, …

```
$ 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 Operations

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

More on in-memory structures

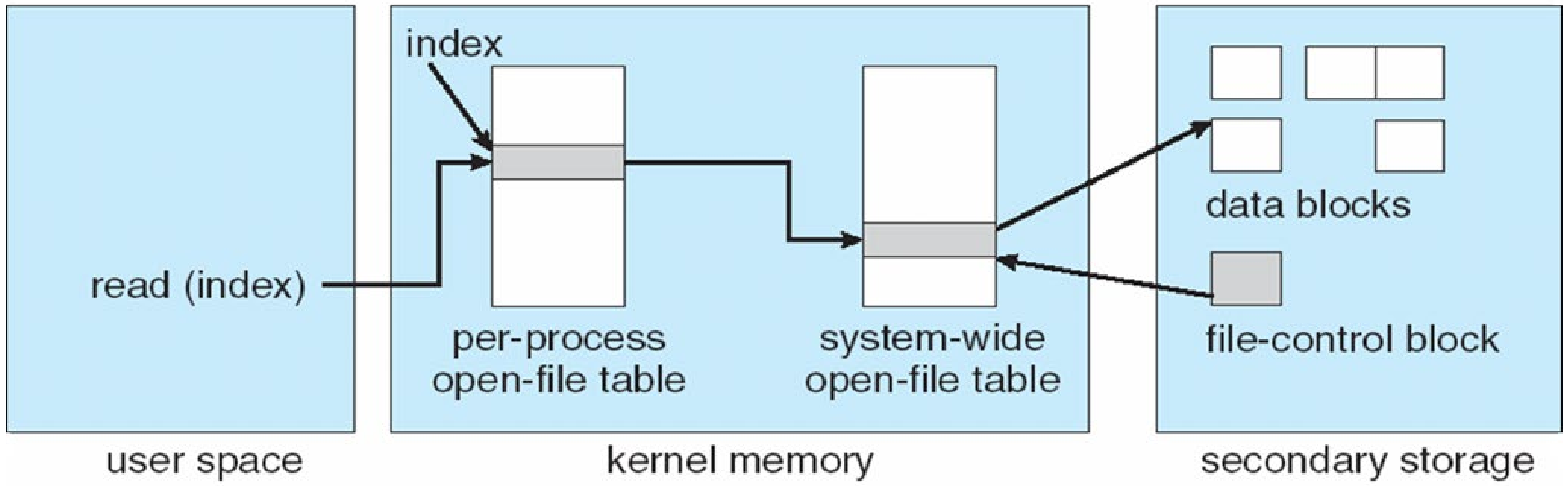
- to improve performance, OS keeps various bits of information related to filesystems in various data structures in memory

- examples:
  - system-wide open-file table: entry for each open file, e.g. first block address, permissions
  - per-process open-file table: e.g. pointers into system-wide open-file table + file pointer
  - mount table: information about each mounted file-system
  - buffer cache: caches FS blocks, to reduce the number of raw reads/writes to files, to speed up access to frequently accessed directories, etc.

Open files

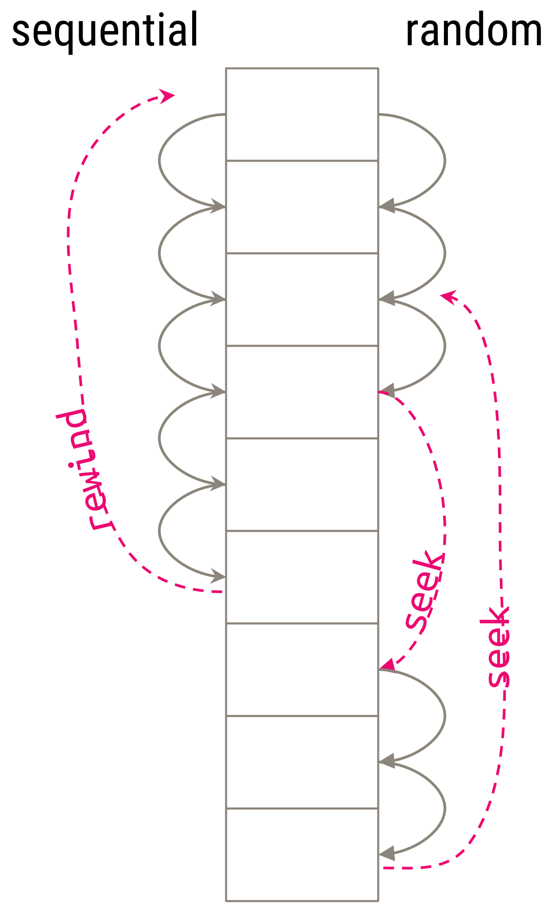
- OS needs to manage open files, and allow fast access to data in these files

- 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



Sequential and random file access

- two general types of accessing files: sequential & random
  - apply to both reading and writing
- **sequential access**
  - bytes in the file are accessed sequentially, from beginning to end
  - no out-of-order access, although files usually can be rewound
  - e.g. `open, read, read, read, rewind, read … close`
  - most common, most optimized
- **random access**
  - can access any byte in any order
  - usually achieved by calling `seek()` or `mmap()` syscalls
  - e.g. `open, read, read, read, seek, read, read, seek, read, … close`
  - usually much (magnitudes) slower than sequential access!

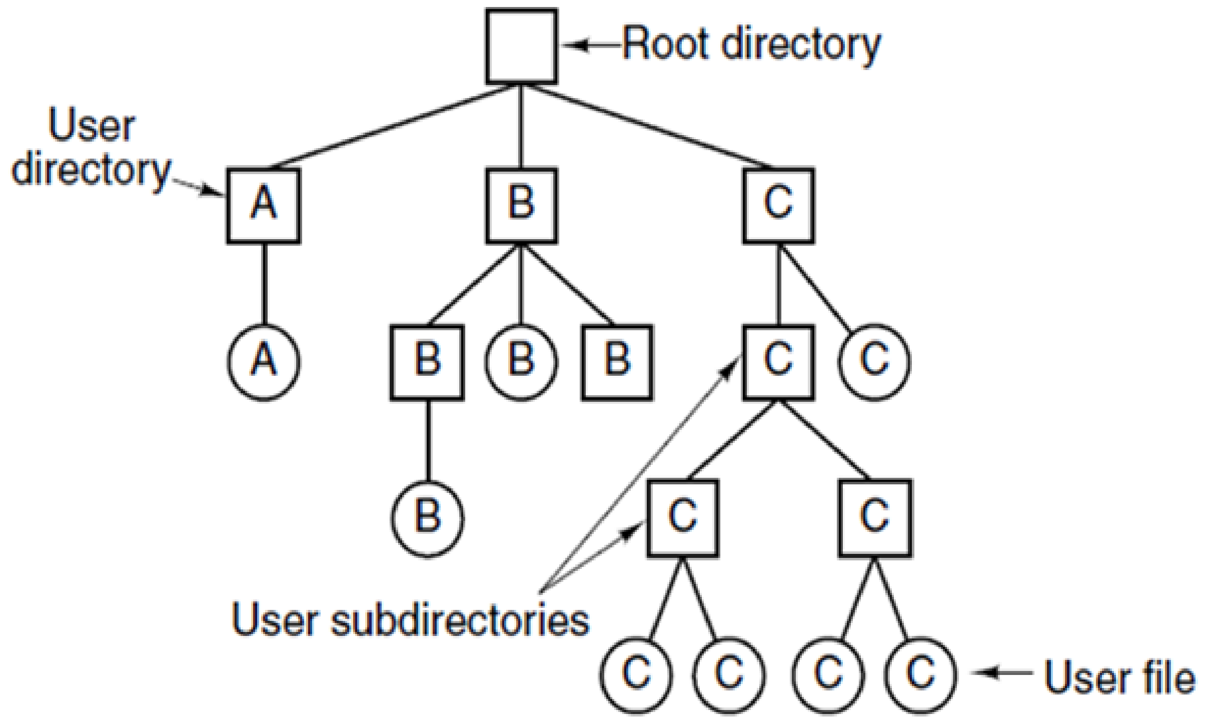


Directories

Directories

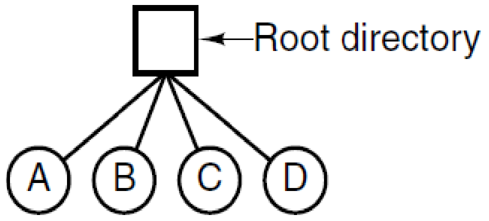
- filesystems use directories to help us with organizing files
- used to organize files hierarchically using tree

  structure (directory structure)
- root node of the tree is the root directory
- internal nodes are directories
- leaf nodes are either files or empty directories
- path in a tree = filepath



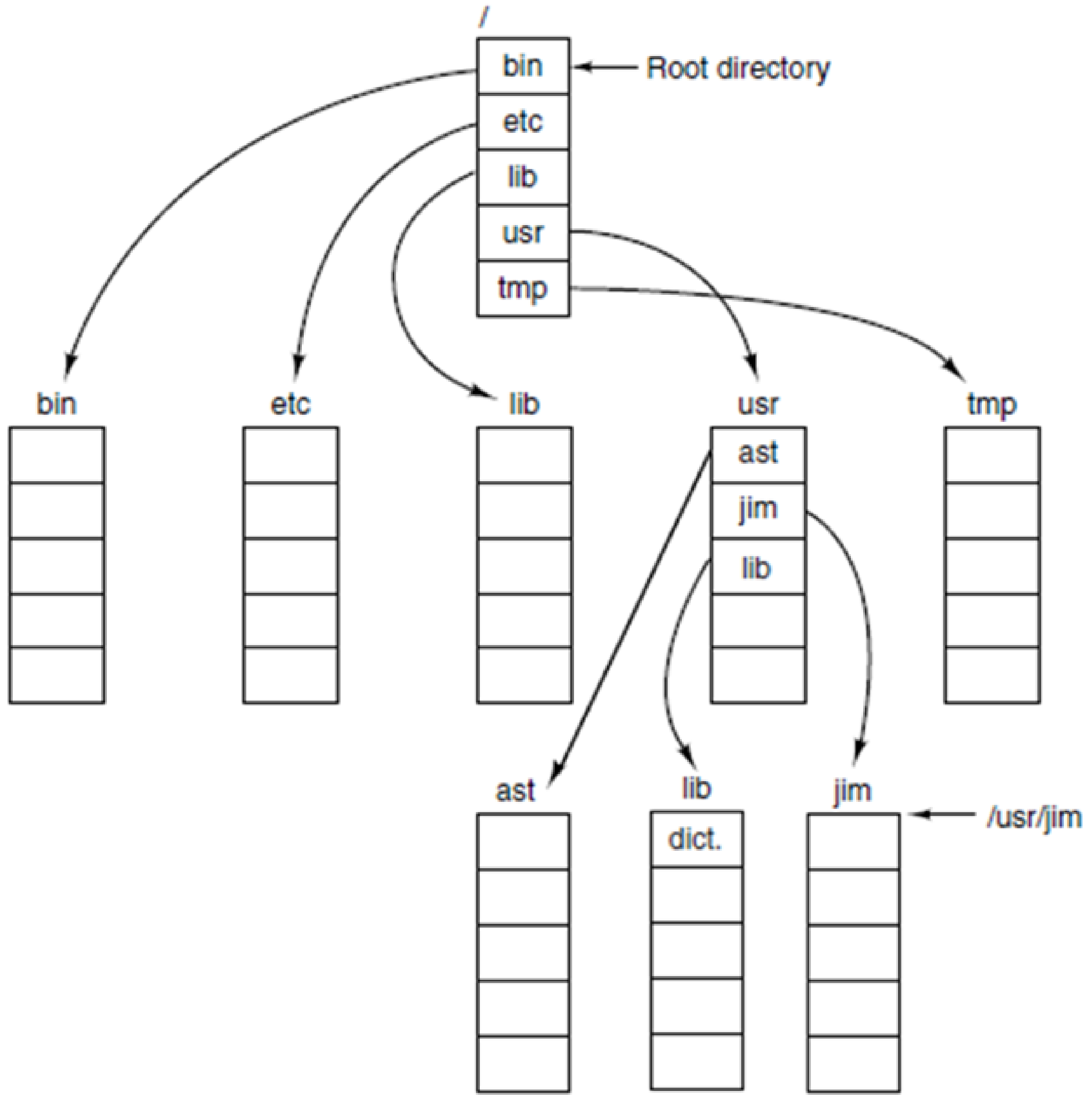
Directory

- a directory is usually implemented as a special file, containing a list of directory entries (dentry)
- contents of dentry depend on the type of filesystem, e.g. filename, size, first block of file contents
- each dentry can represent a file or a directory (subdirectory)
  - if subdirectories allowed → hierarchical directory system (widespread use)
  - if subdirectories not allowed → single-level directory system (not common anymore)



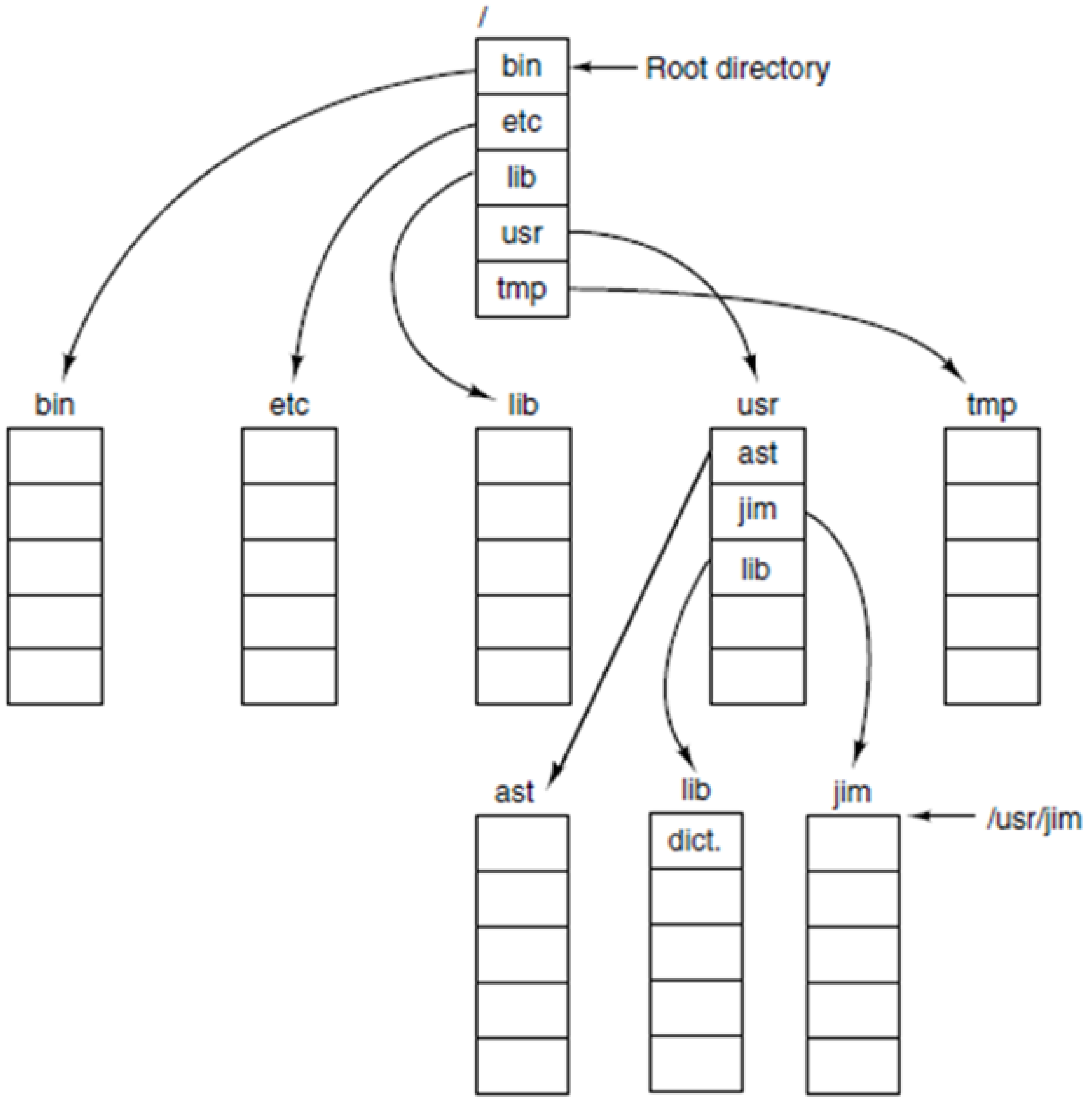
Directories (Unix)

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, e.g.
  `./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/lib/../jim`
  - ☐ `../../../../../../../usr/jim`

  all refer to the same directory

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 for files
- 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`.

Directory implementation

Directory implementation

- linear list of dentries
  - 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) - uses special data structure called HTree

Filesystem blocks

- filesystems store files in fixed-size blocks
  - similar to fixed partitioning scheme for memory management (will be covered later)
  - number of blocks per file = ceil(file_size / block_size)
  - filesystems often suffer from internal fragmentation (wasted space within blocks)
- filesystem block size is usually a multiple (2^n) of the underlying disk block size
- blocks belonging to the same file not necessarily adjacent
  - fragmented file = file stored in non-consecutive blocks
  - fragmented files suffer from performance issues on mechanical hard-drives
  - as filesystem becomes full, files become fragmented, the entire filesystem slows down
  - TLDR; don't fill up your filesystem

File allocation methods

File allocation methods

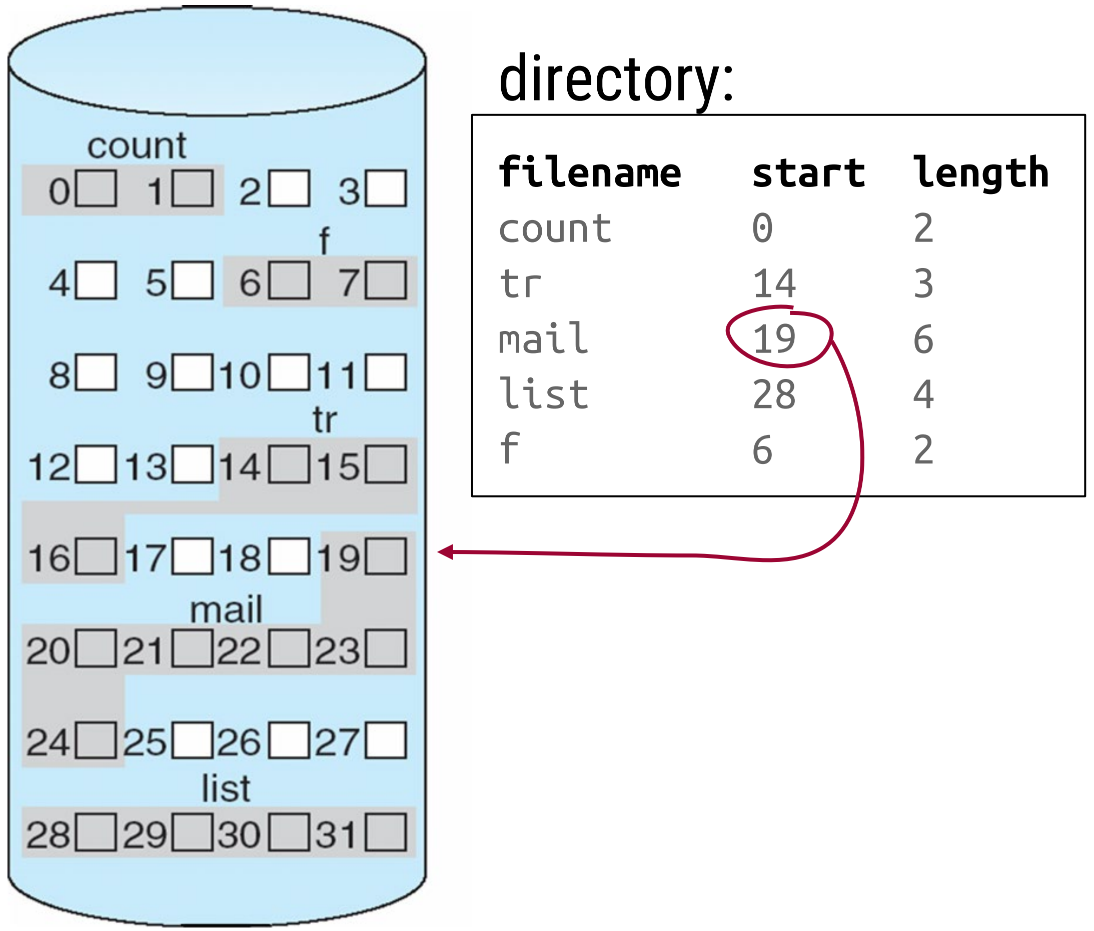
- file's contents and attributes may need to be stored in multiple blocks
- FS needs to keep track of all used and free blocks
    - which blocks belong to file xxx?
    - where are unused blocks located?
- a file allocation method refers to how disk blocks are allocated and tracked
- we will discuss:
    - contiguous allocation
    - linked allocation + FAT
    - indexed allocation

Contiguous allocation

Contiguous allocation

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

Contiguous allocation



directory:

filename	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 byte address:

q	r

  q = upper bits
  r = lower bits

  physical address computation:

block number = q + "address of first block"
displacement within block = r

Linked allocation

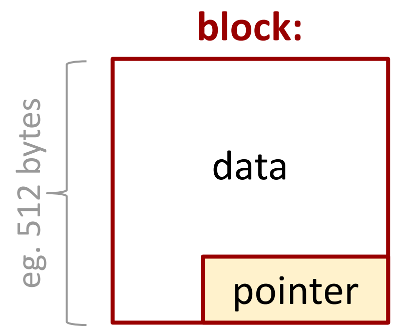
Linked allocation

- 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 which has NULL pointer as next block

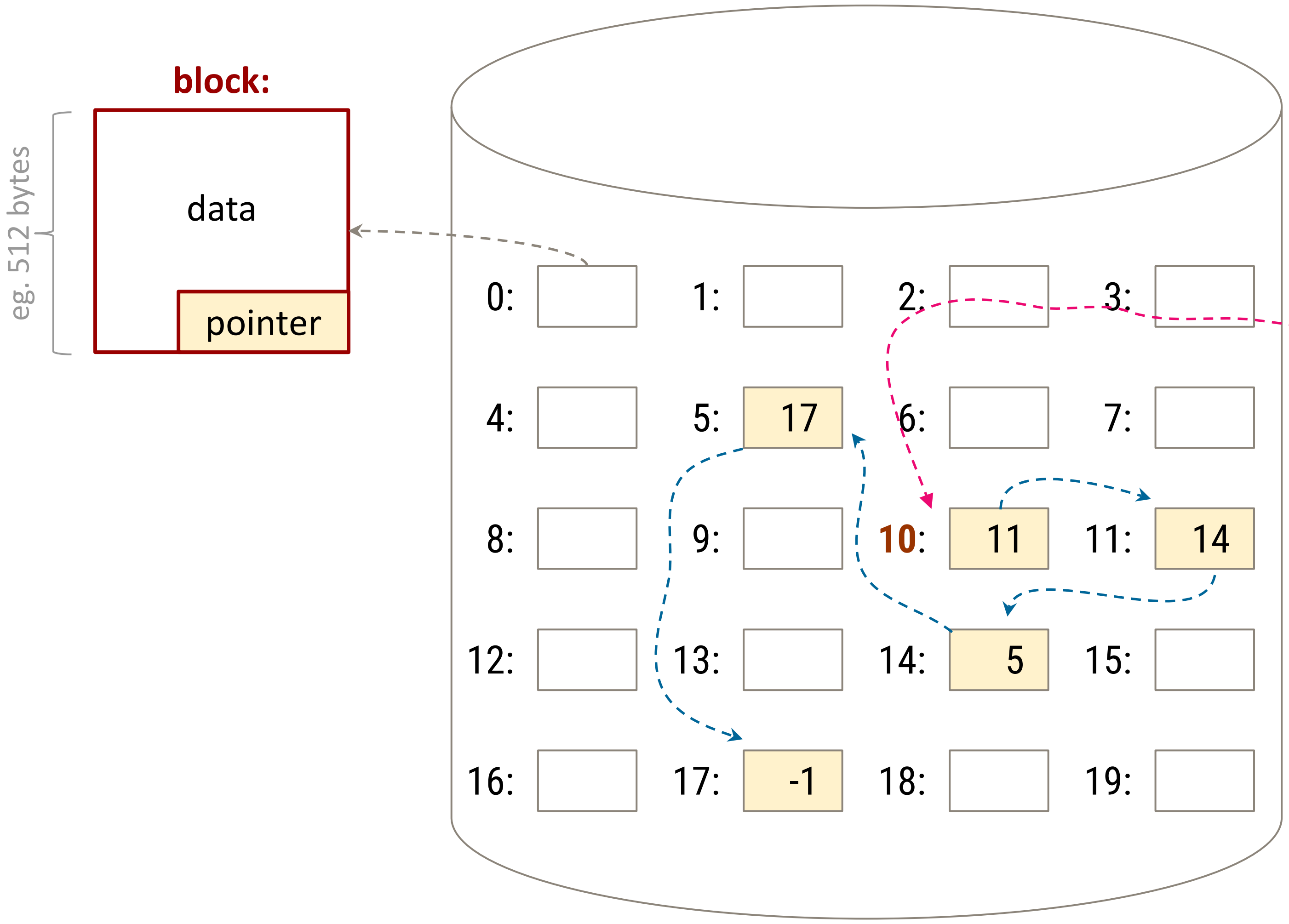
- no external fragmentation → compaction needed only for performance
- separate free space management needed - eg. linked list of free blocks

- 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



Example directory entry:

filename	start block	size
`test.txt`	10	2100

contents of `test.txt` spread over 5 blocks:

10, 11, 14, 5, 17

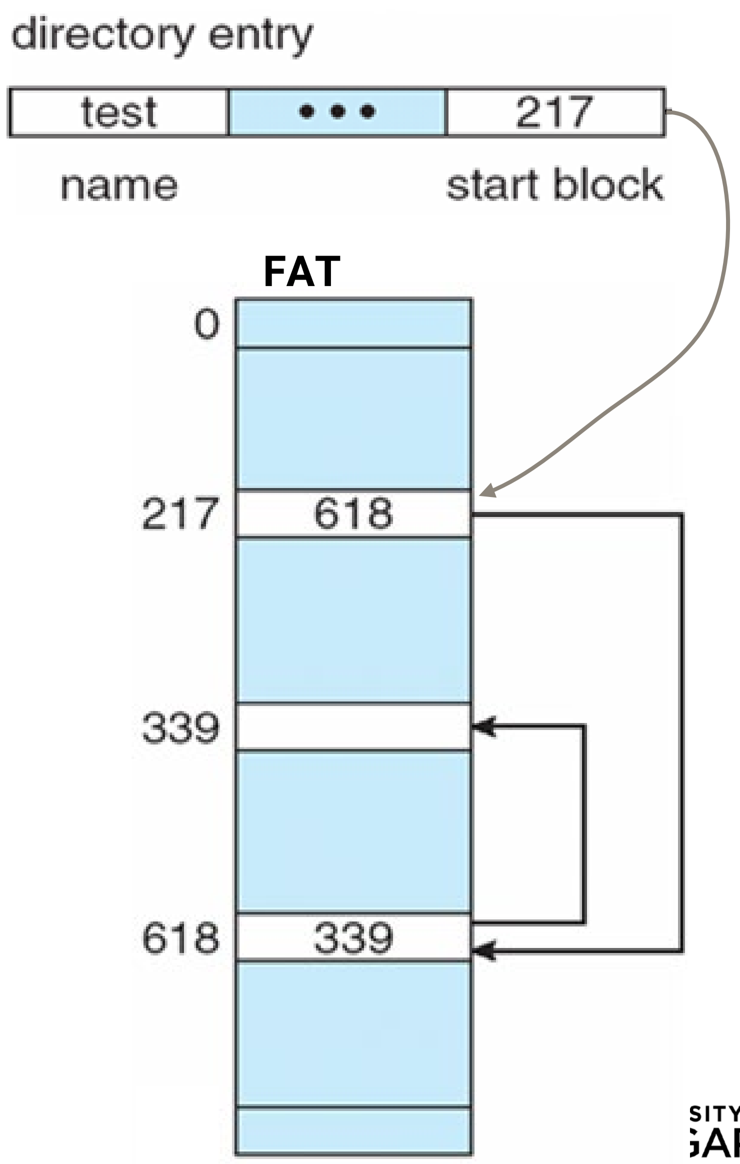
Why do we need 'size' in dentry?

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

File Allocation Table (FAT)

File Allocation Table (FAT)

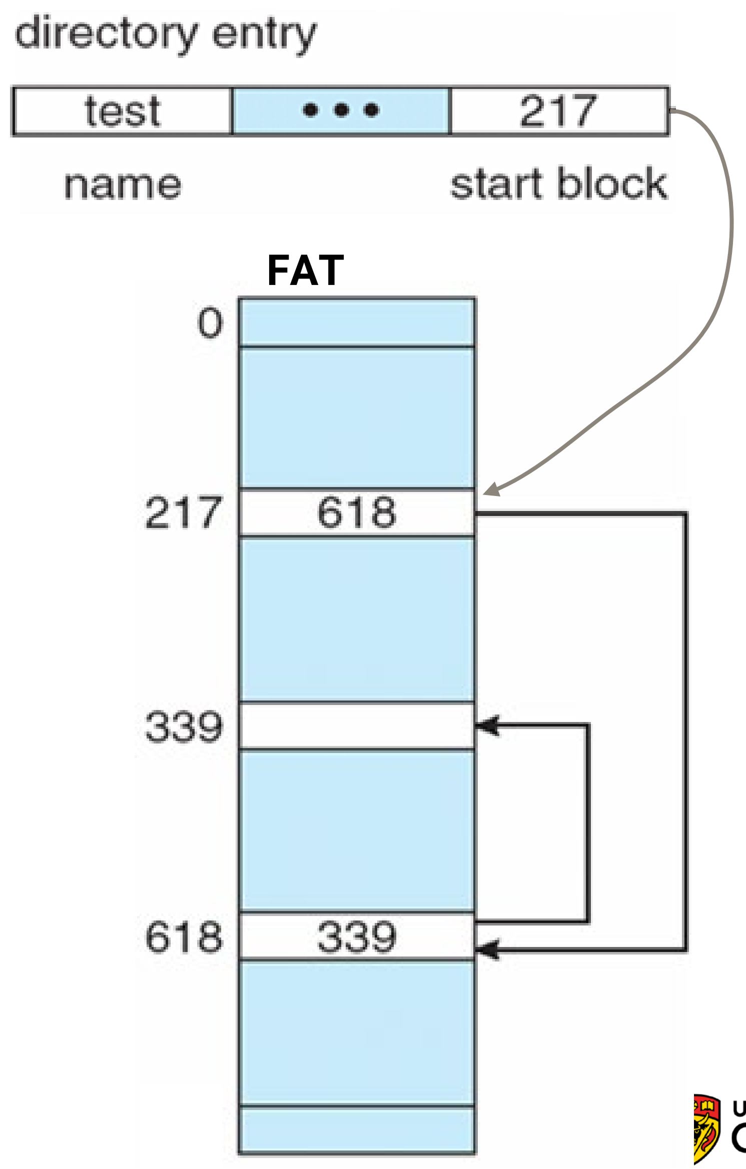
- FAT (File Allocation Table) is a variation of linked allocation
- all 'next' pointers are stored in a separate table (FAT)
- FAT can be located eg. at the beginning or end of the FS volume
- FAT is essentially an array containing block numbers, indexed by block numbers:
  - `fat[i]` = next pointer for block `i`
  - value -1 could represent NULL pointer
    i.e. if `fat[i]==-1` then block `i` is the last block of file
- can store multiple copies of FAT to add redundancy
- dentry contains first block # of the file, which is also an index into FAT to find next block, etc.



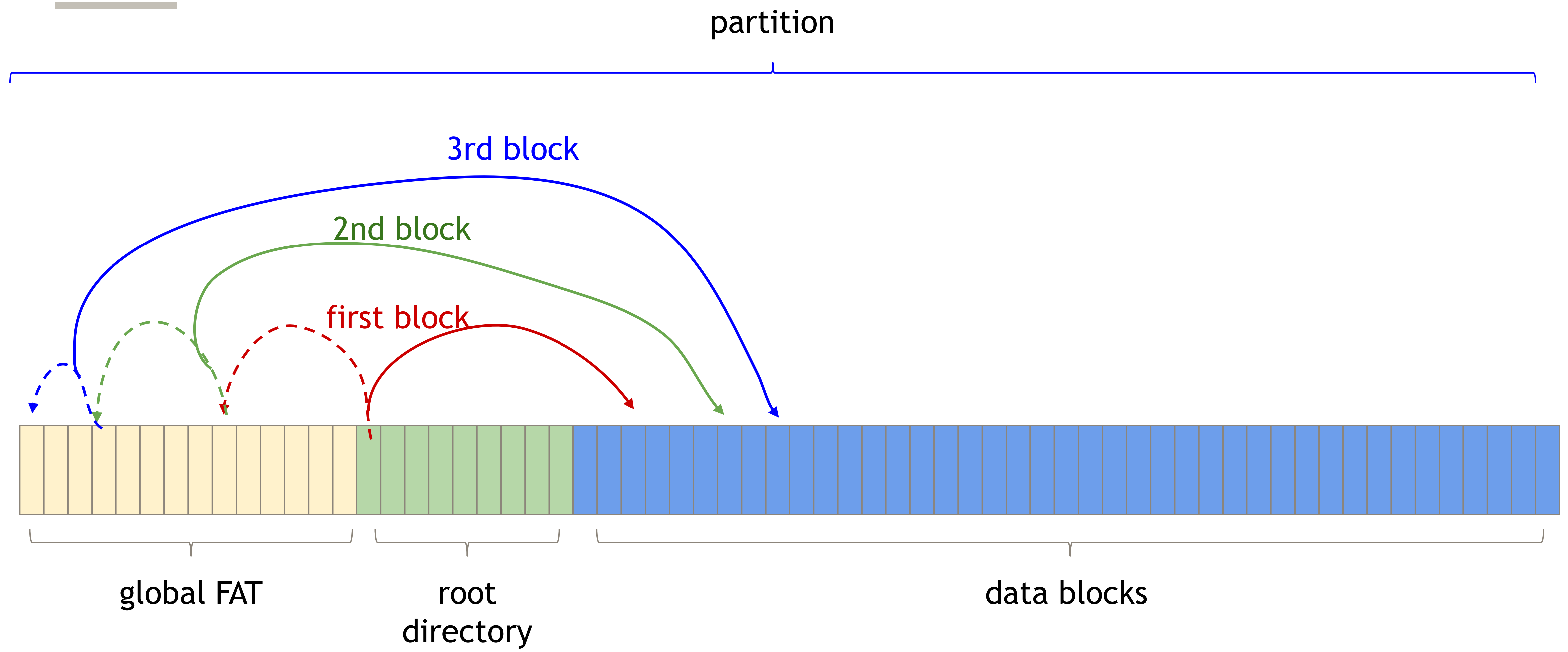
File Allocation Table (FAT)

- compared to linked allocation:
  - easier random access since all pointers stored together, and therefore cache-able *

- 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



File Allocation Table (FAT)

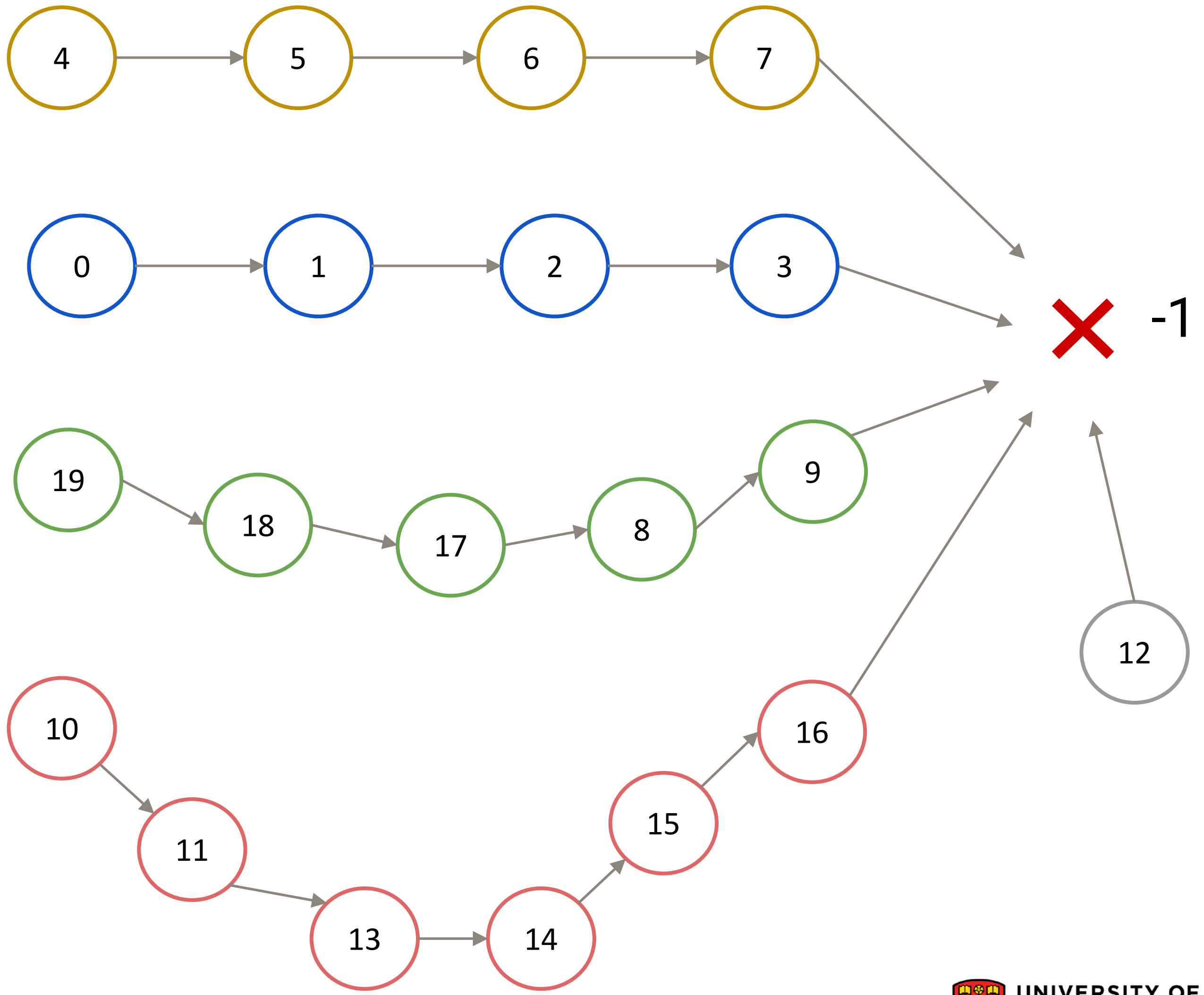


File Allocation Table (FAT)

Example FAT:

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:
1	2	3	-1	5	6	7	-1	9	-1

10:	11:	12:	13:	14:	15:	16:	17:	18:	19:
11	13	-1	14	15	16	-1	8	17	18



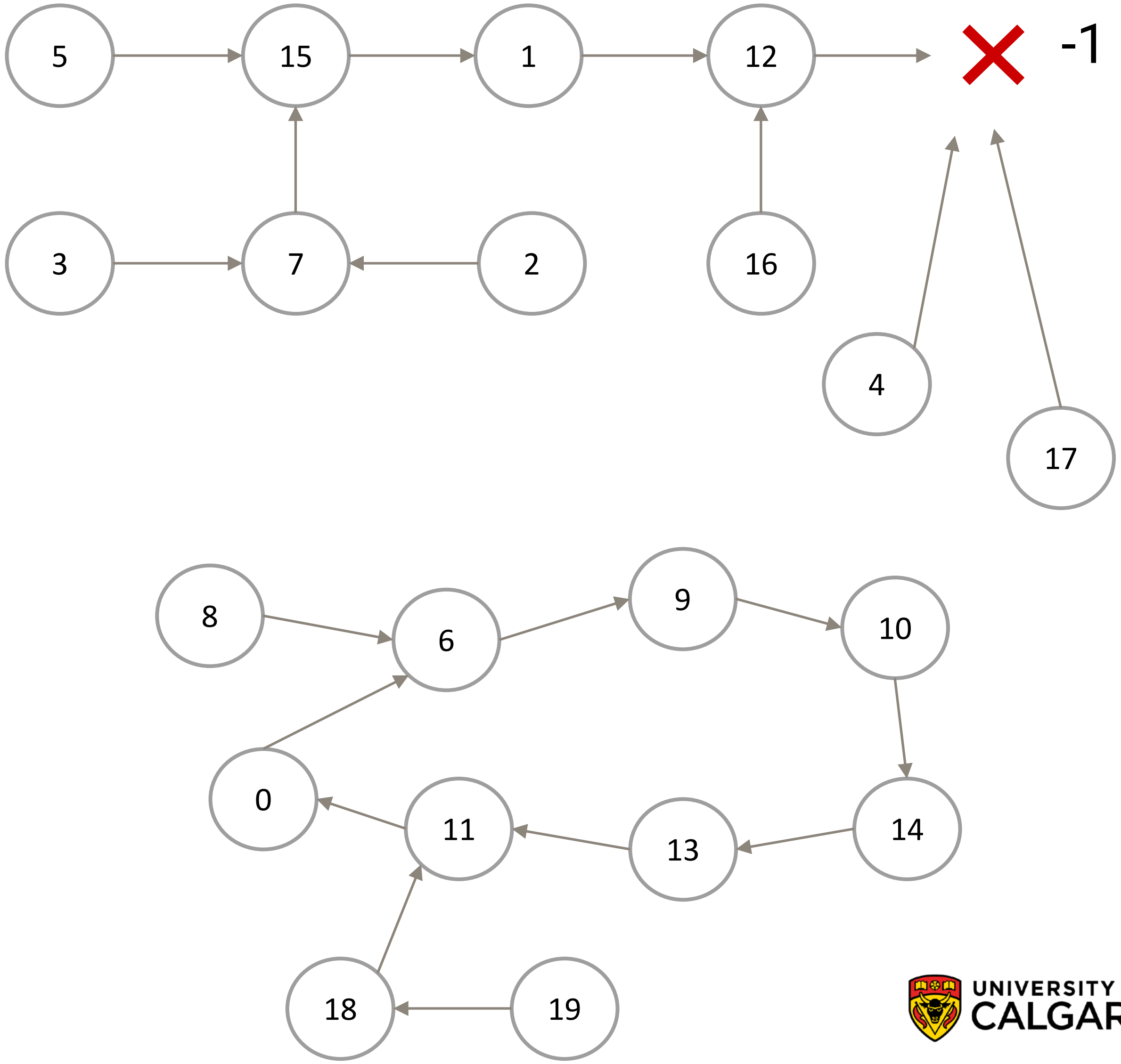
the above FAT contains 5 files, starting on blocks 4, 0, 19, 10 and 12

File Allocation Table (FAT)

Example of a corrupted FAT:
contains at most 3 files

0:	1:	2:	3:	4:	5:	6:	7:	8:	9:
6	12	7	7	-1	15	9	15	6	10

10:	11:	12:	13:	14:	15:	16:	17:	18:	19:
14	0	-1	11	13	1	12	-1	11	18



Largest possible file starts either on block 3 or block 2.

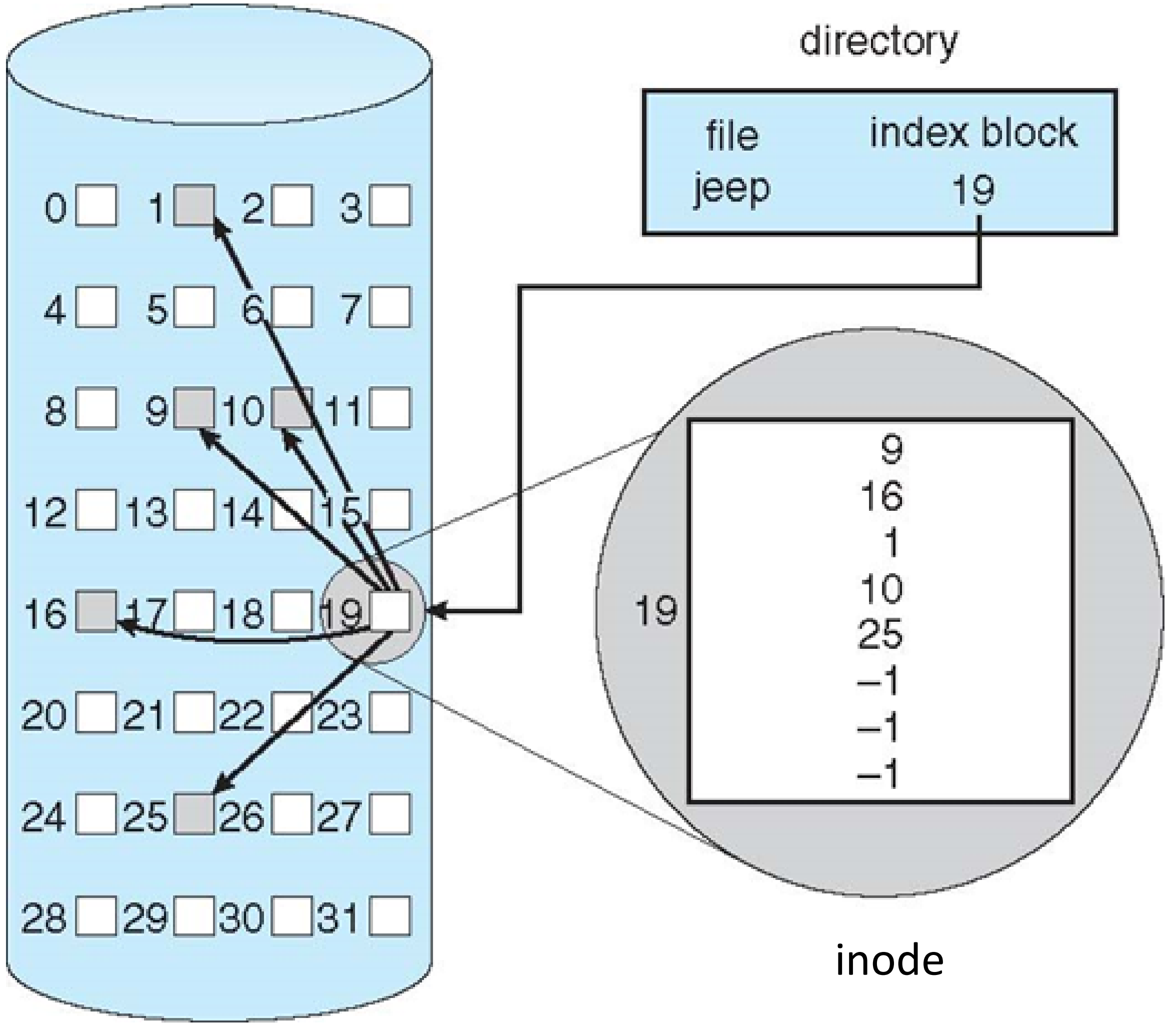
Two more files of length 1 start on nodes 4 and 17.

Indexed Allocation (inodes)

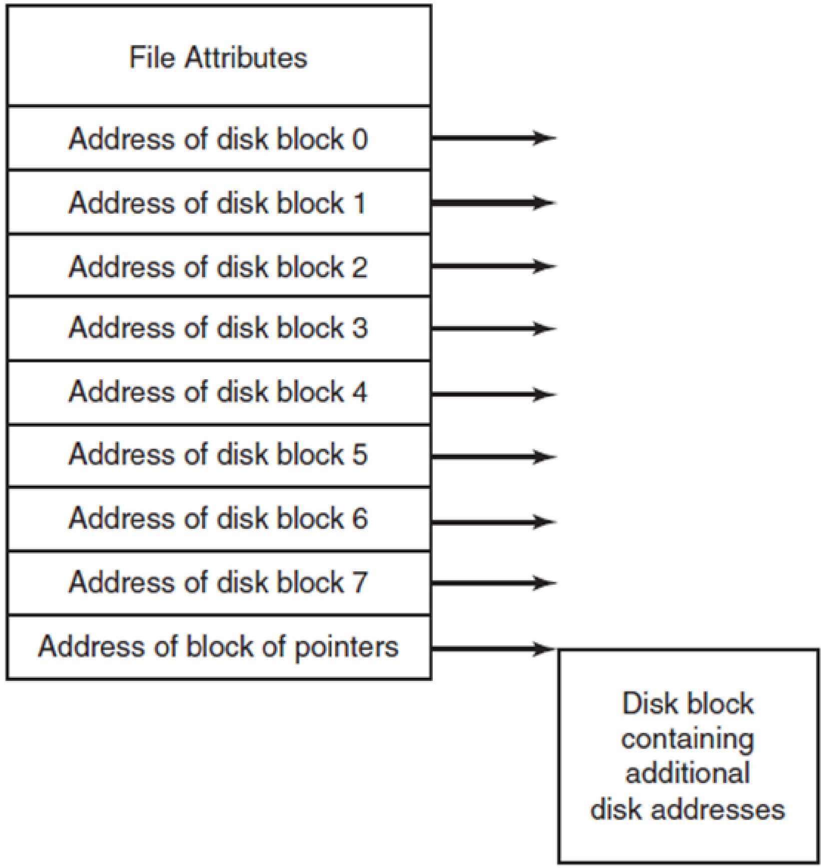
Indexed Allocation (inodes)

- basic idea behind indexed allocation is to maintain per-file FAT-like structure
- this way we don't need to cache pointers for all files, only for the open files
- we call this structure inode, and it contains:
  - direct pointers to blocks with file contents
  - indirect pointers to blocks that contain even more pointers
  - various file attributes:
    - file size in bytes, device ID, owner, permissions, timestamps, link count, …
  - inode **does not** contain a filename
- dentry is used to associate filename with the inode
  - dentry = filename + pointer to disk block where the inode is stored
  - it is possible to have different filenames associated with the same inode
    - called hard links

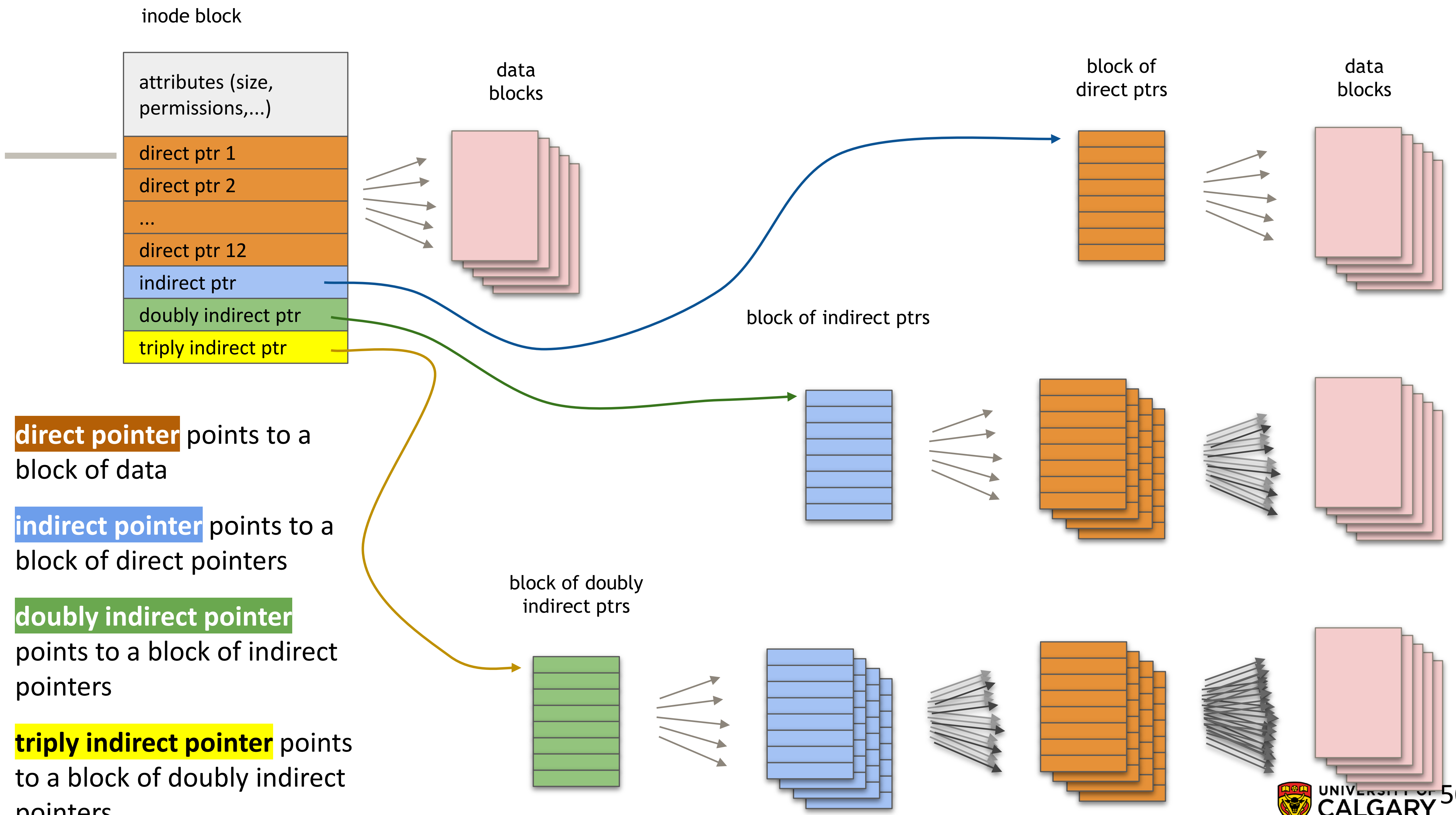
Example of Indexed Allocation



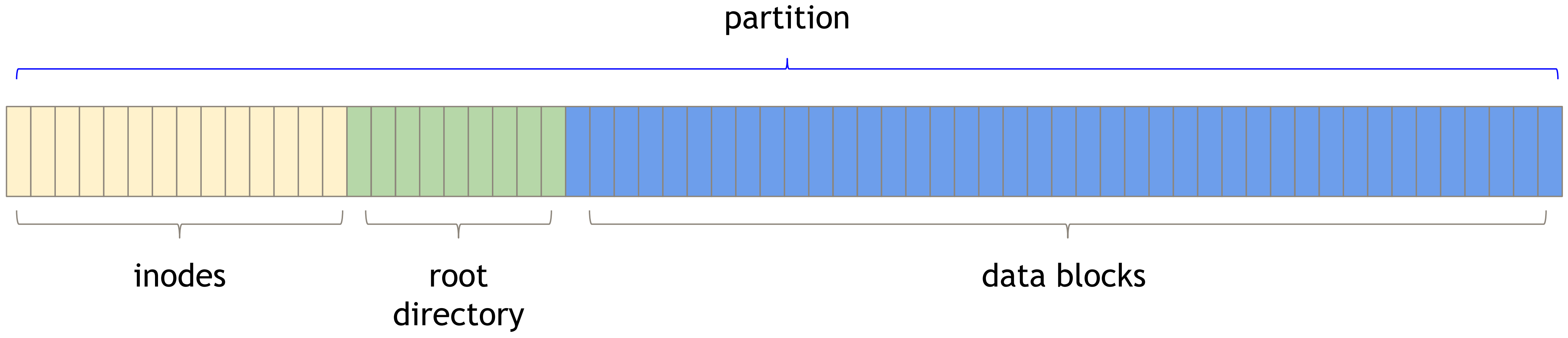
inode

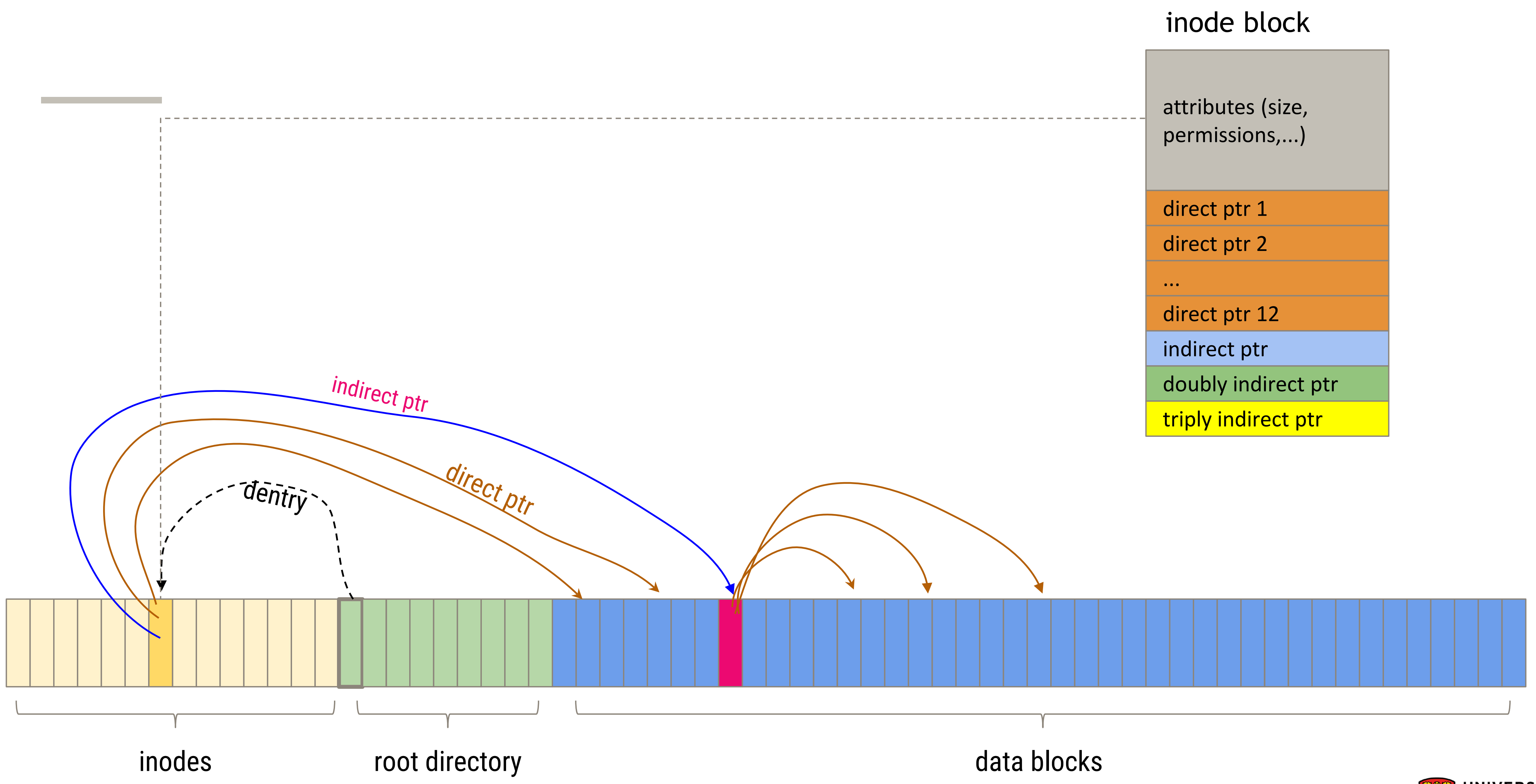


inode



- **direct pointer** points to a block of data
- **indirect pointer** points to a block of direct pointers
- **doubly indirect pointer** points to a block of indirect pointers
- **triply indirect pointer** points to a block of doubly indirect pointers


partition
inodes
root directory
data blocks


inode block
attributes (size, permissions,...)
direct ptr 1
direct ptr 2
...
direct ptr 12
indirect ptr
doubly indirect ptr
triply indirect ptr
indirect ptr
direct ptr
dentry
inodes
root directory
data blocks

inodes in Linux (ext2)

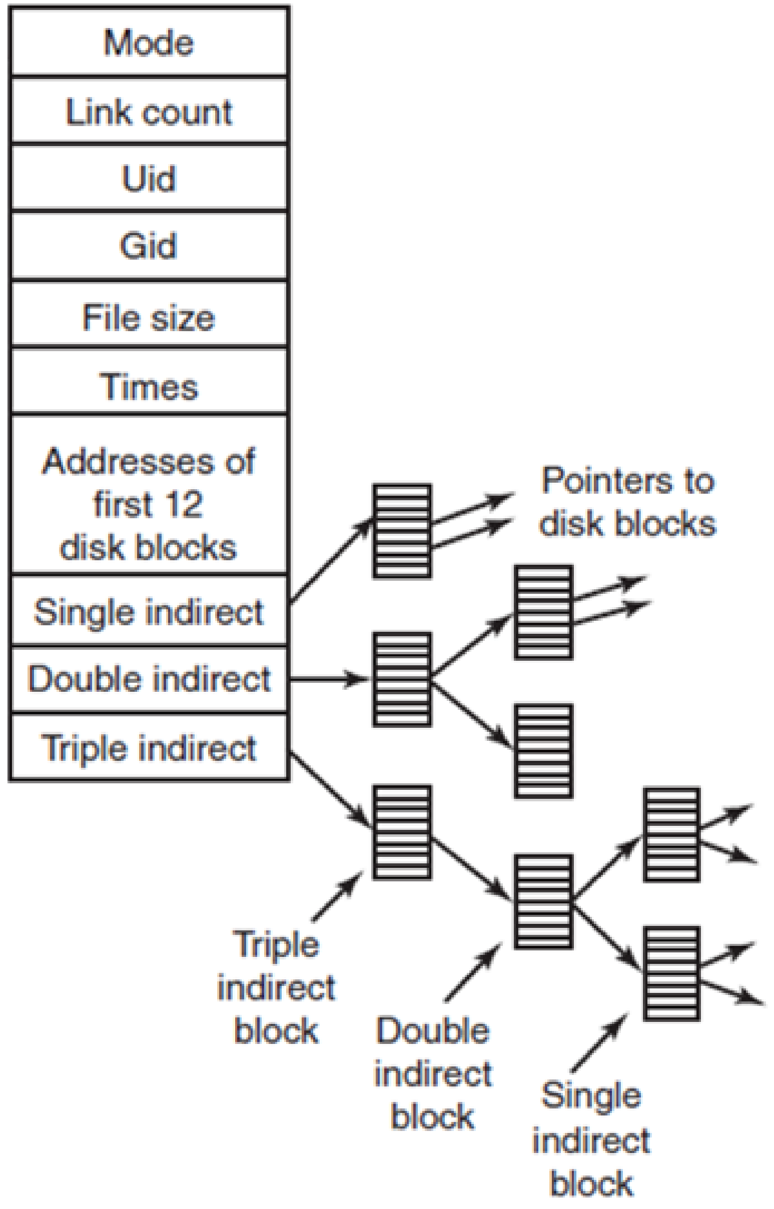
- example: block size 1KB, block address 4 bytes
- if inode can only store 12 direct pointers
  → max file size 12KB

- if inode can also store 1 indirect pointer:
  1KB block can store1KB/4B=256 additional pointers
  → max file size 12+256 blocks = 268KB

- if inode can also store 1 double-indirect pointer:
  or 256 blocks each with 256 addresses
  → max file size 12 + 256 + 256^2 blocks ~= 64MB

- adding triple-indirect pointer:
  → max file size 12 + 256 + 256^2 + 256^3 blocks ~= 16GB

- ext3 max file size = 2TB
- ext4 max file size = 16TB (using 48bit addresses and extents)



inodes

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

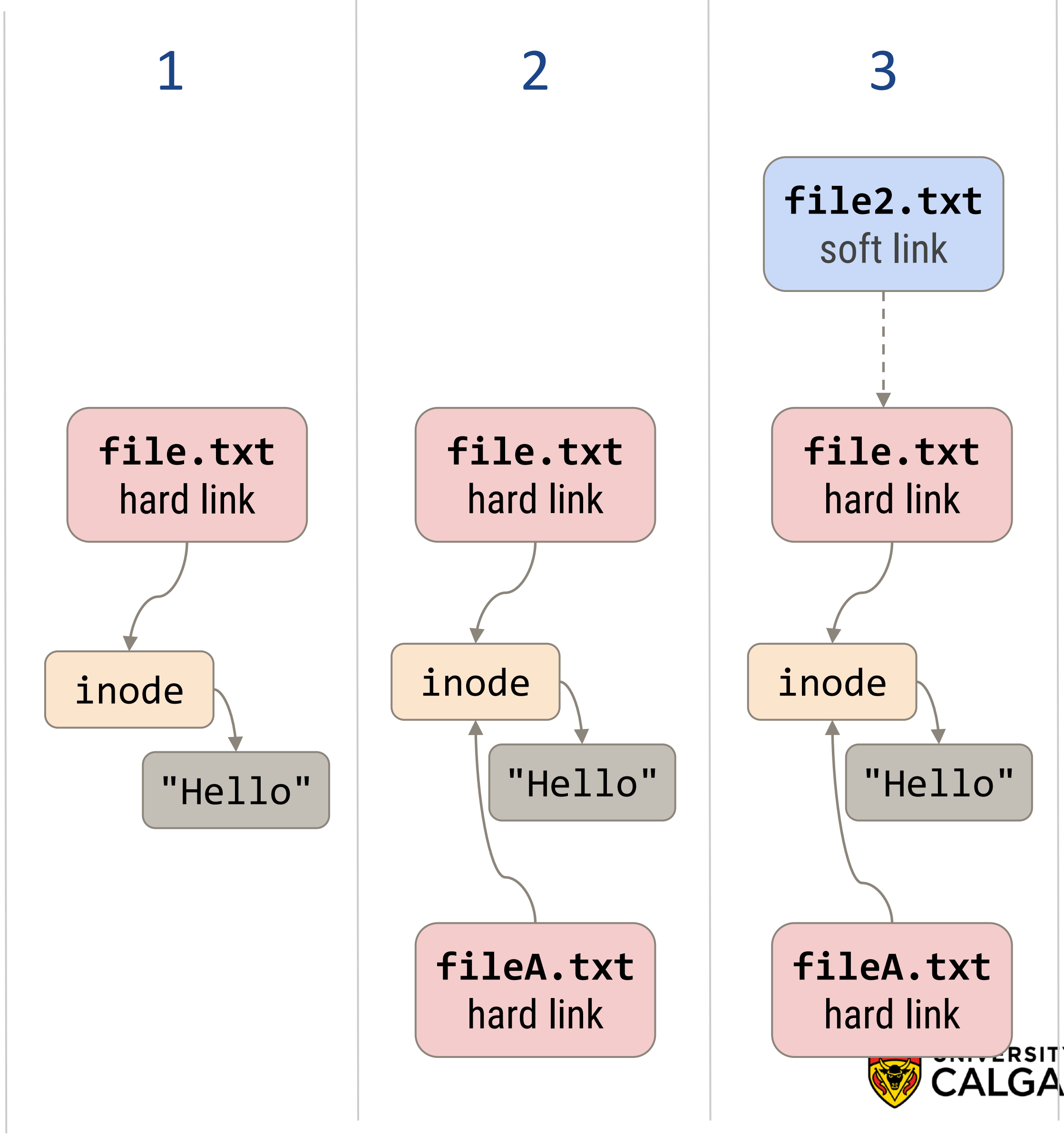
Hard link vs soft link

Hard link vs soft link

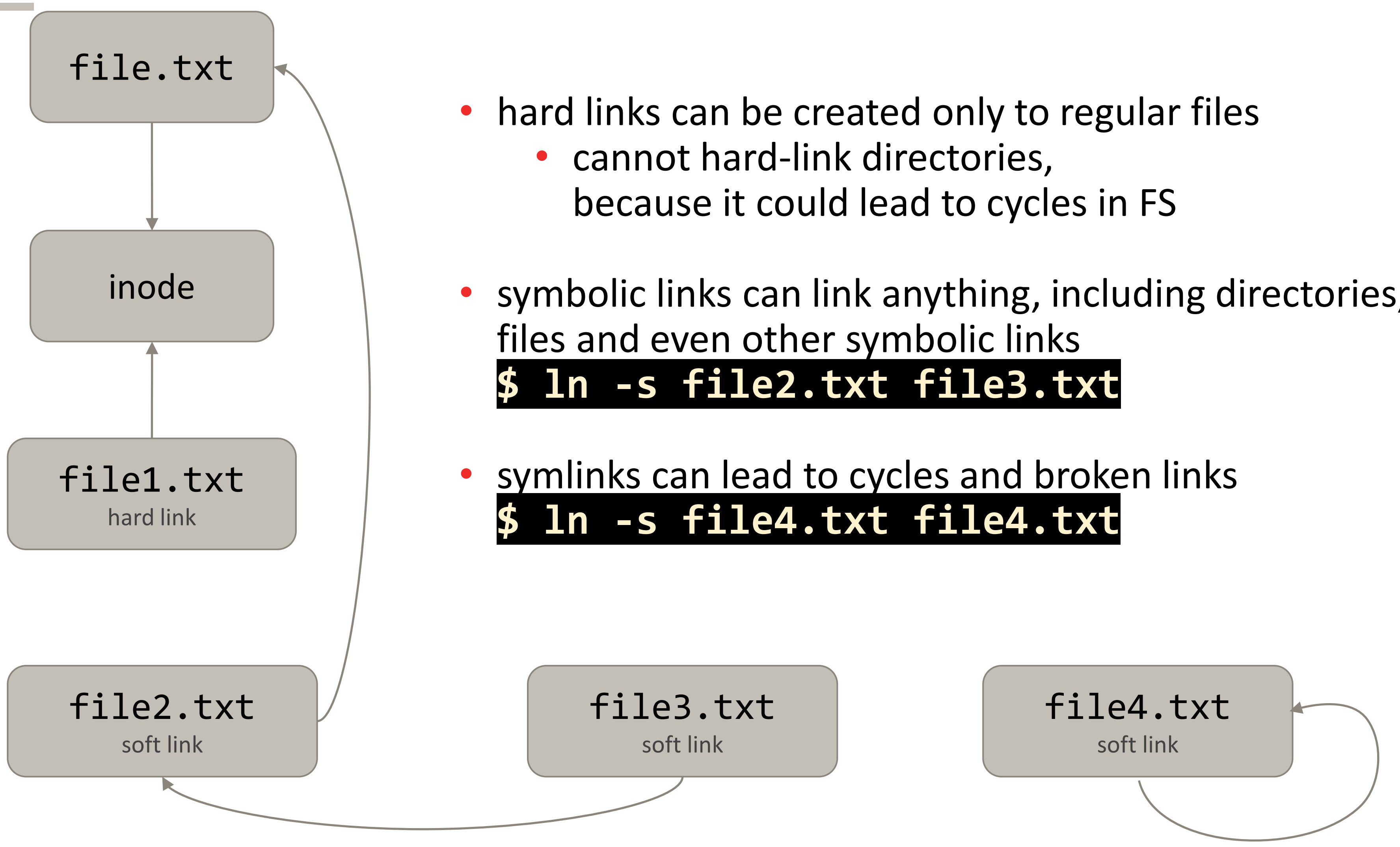
1. create file.txt
   ```
   $ echo "Hello" > file.txt
   ```
   this actually creates hard link

2. create hard link fileA.txt that points to file.txt
   ```
   $ ln file.txt fileA.txt
   ```
   a hard link points to the same inode
   if we delete file.txt, fileA.txt will still work
   file.txt and fileA.txt are indistinguishable

3. create soft link fileB.txt that points file.txt
   ```
   $ ln -s file.txt fileB.txt
   ```
   soft link points to a filename
   if we delete file.txt, fileB.txt will be broken



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 even other symbolic links

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

- symlinks can lead to cycles and broken links

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

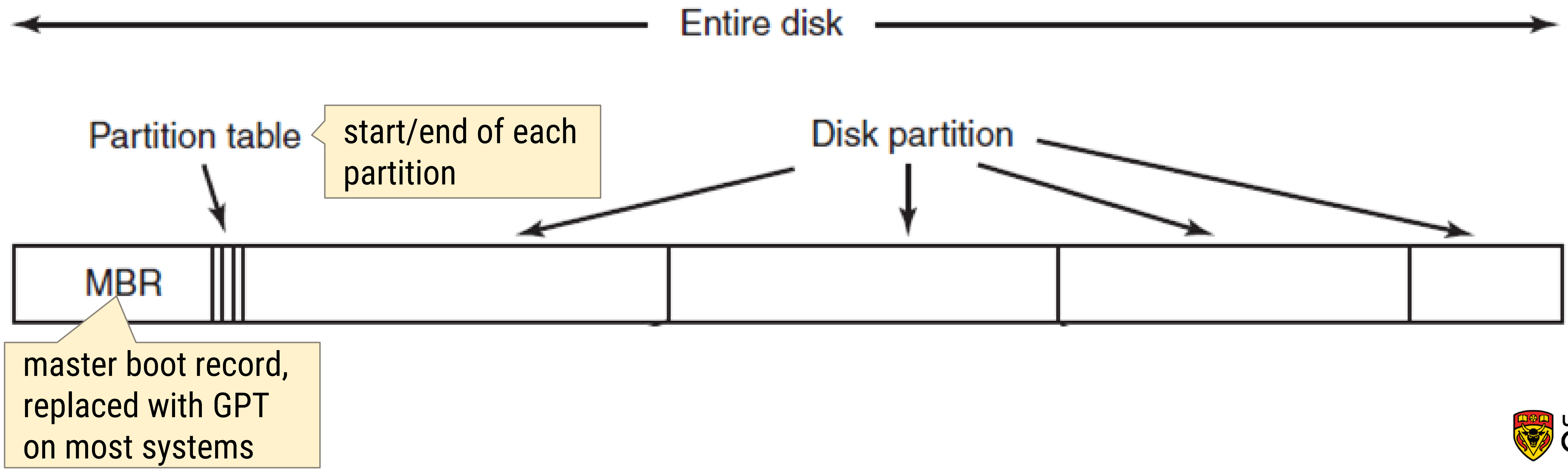
Performance

- 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
  - minimize random access

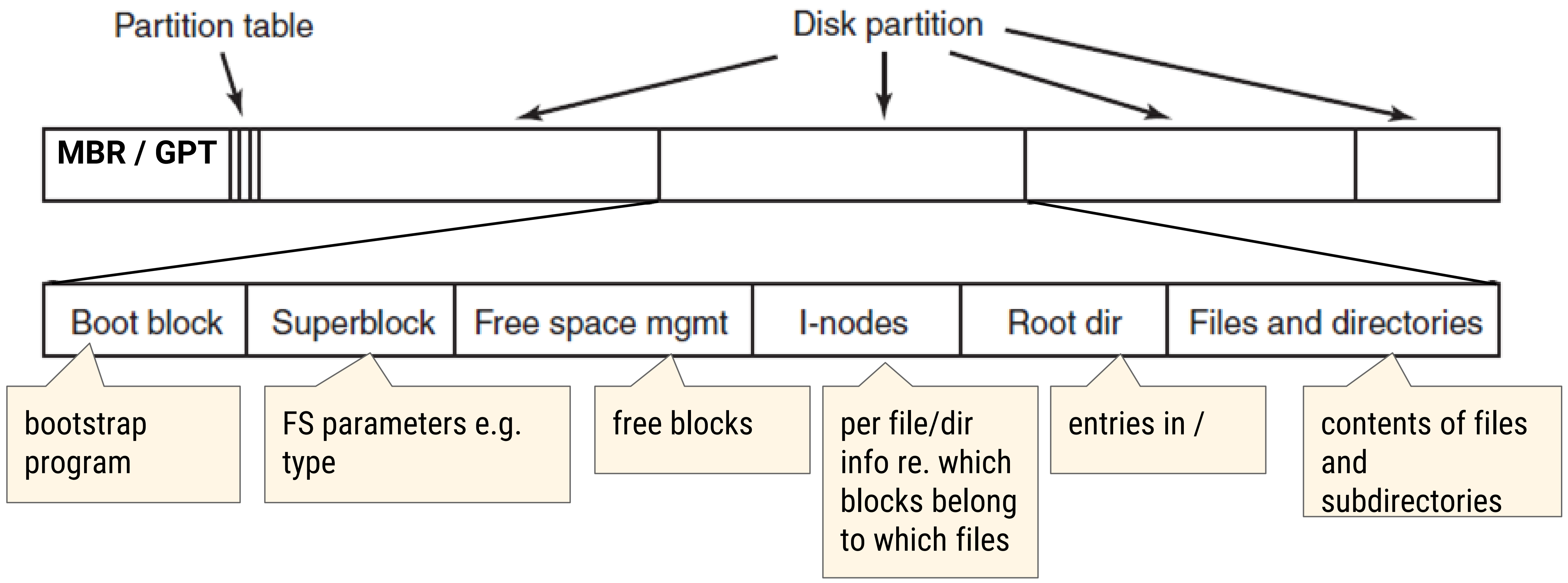
Disk partitions

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


Entire disk
Partition table
start/end of each partition
Disk partition
MBR
master boot record, replaced with GPT on most systems

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 '/’

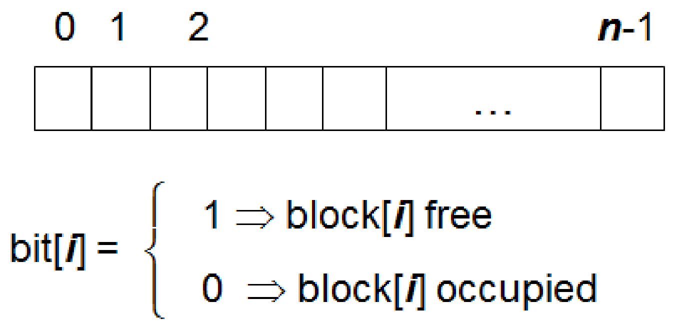
- other partitions can hold other OSes, other file systems, or be raw
  - can mount automatically during boot, or manually after booting

- 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

Free space management

Free space management - bitmaps

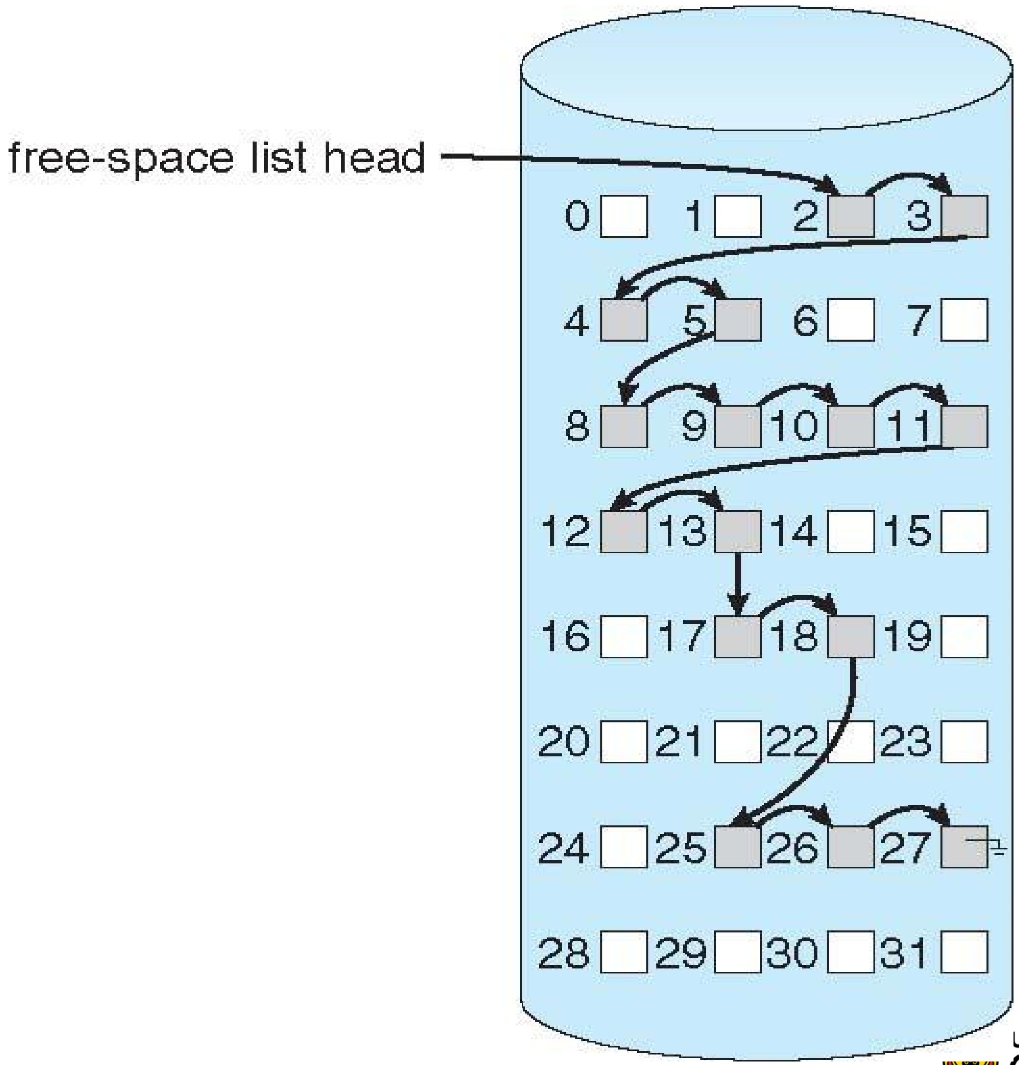
- file systems maintain free-space list to track available blocks
- can be implemented as a bit vector or bitmap
- OS can reserve some blocks for the bitmap



- 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

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



UNIVERSITY OF CALGARY

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

UNIVERSITY OF CALGARY

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

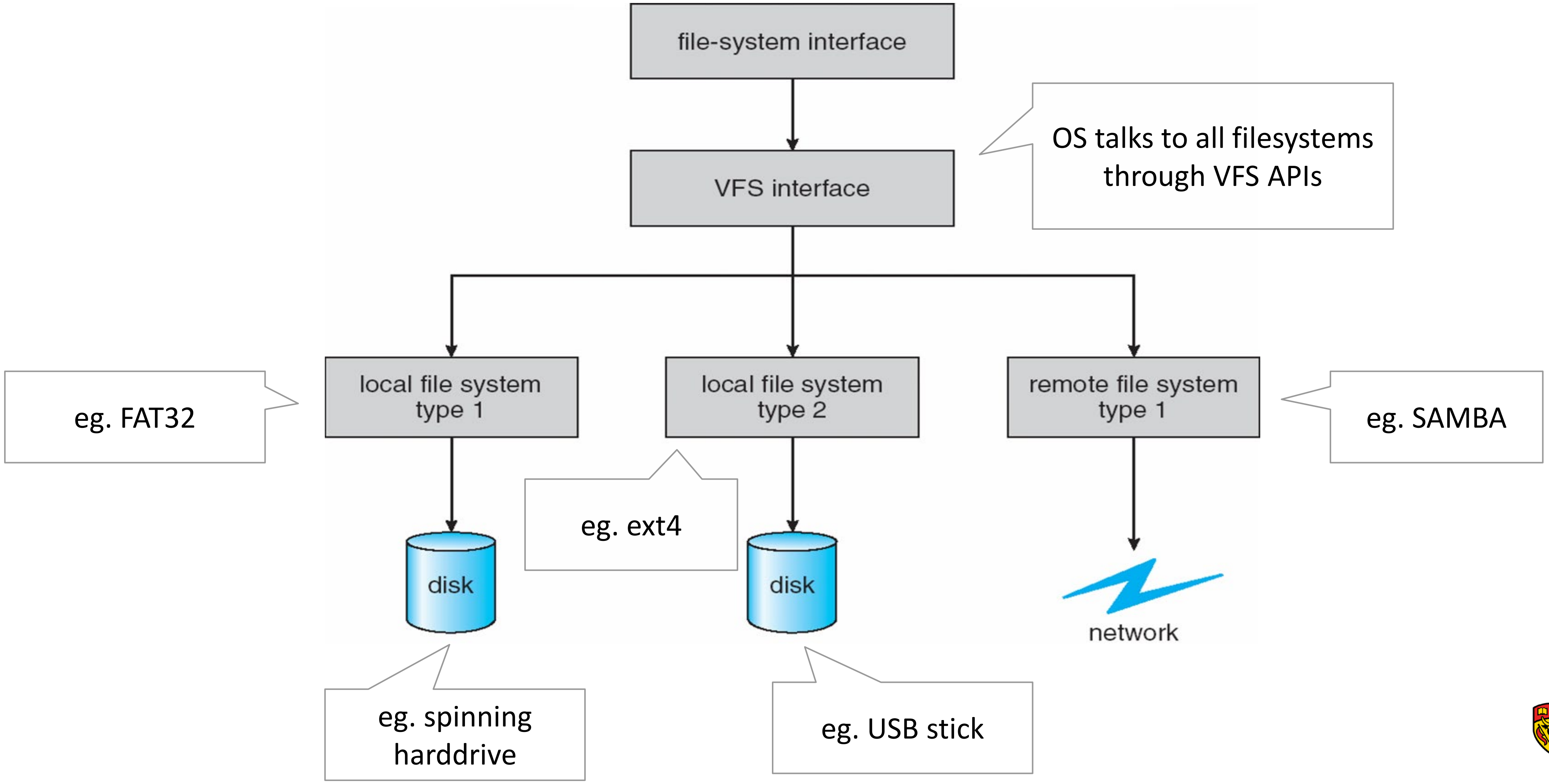
Virtual File Systems

Virtual File Systems

- Virtual File System (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

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

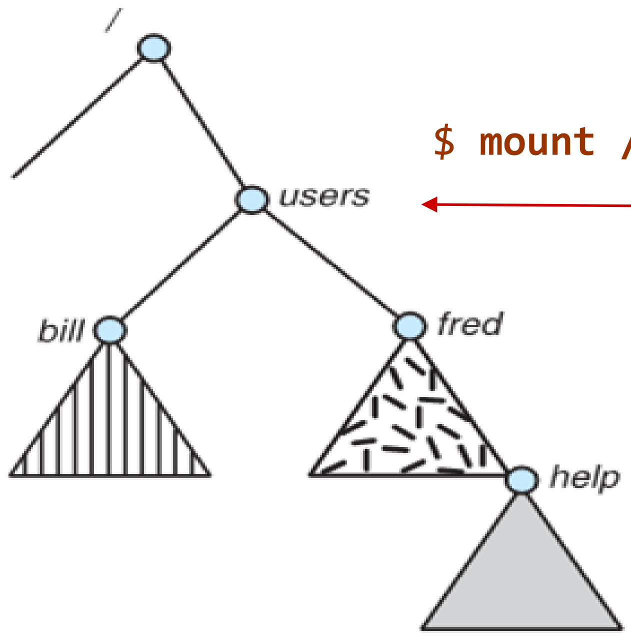
- a developer of a new FS only needs to implement VFS API

- then the FS can be mounted by Linux

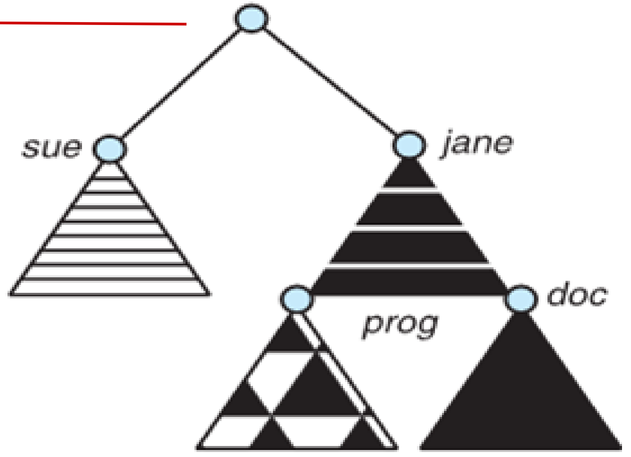
File System Mounting

- 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

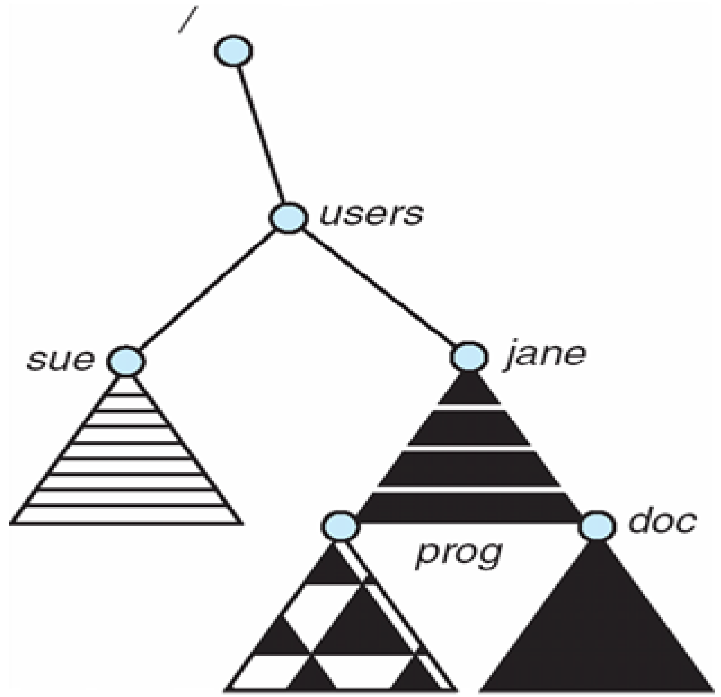
```
$ mount /dev/sdc1 /users
```



root filesystem before mounting filesystem 2



filesystem 2 on /dev/sdc1

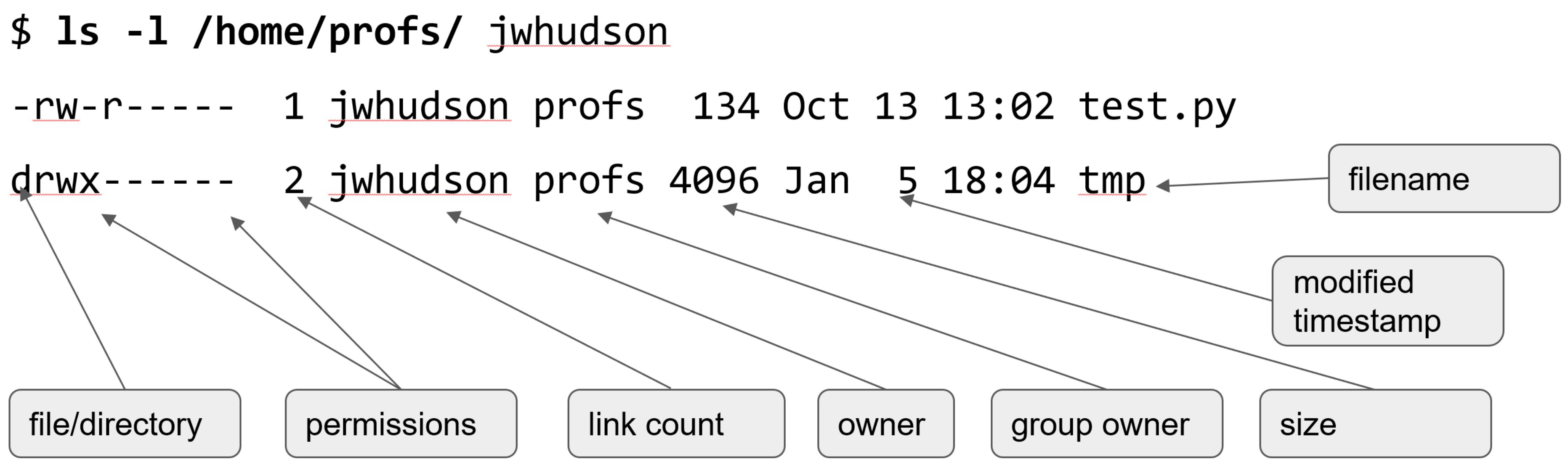


root filesystem after mounting filesystem 2

Unix File System & Permissions

Unix File System & Permissions

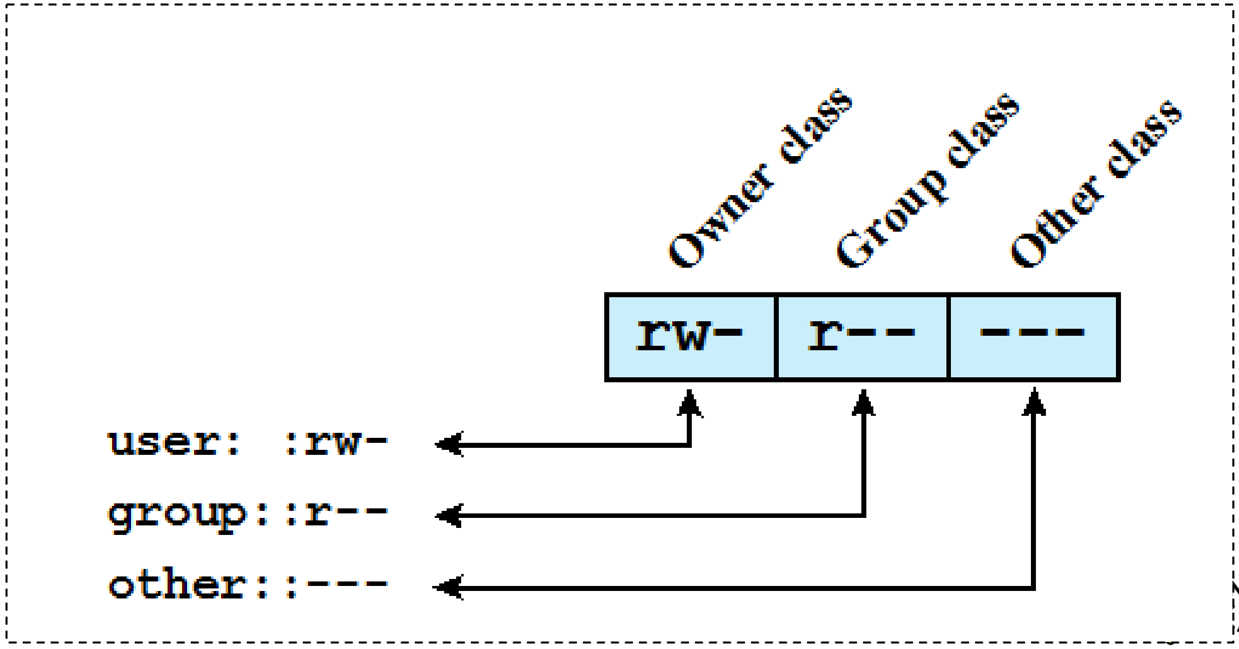
- every file is owned by a user and a group
- permissions usually displayed in compact 10-character notation



```
$ ls -l /home/profs/ jwhudson
-rw-r-----  1 jwhudson profs  134 Oct 13 13:02 test.py
drwx------  2 jwhudson 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)

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

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?

Review

UNIVERSITY OF CALGARY

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

Onward to … memory

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY