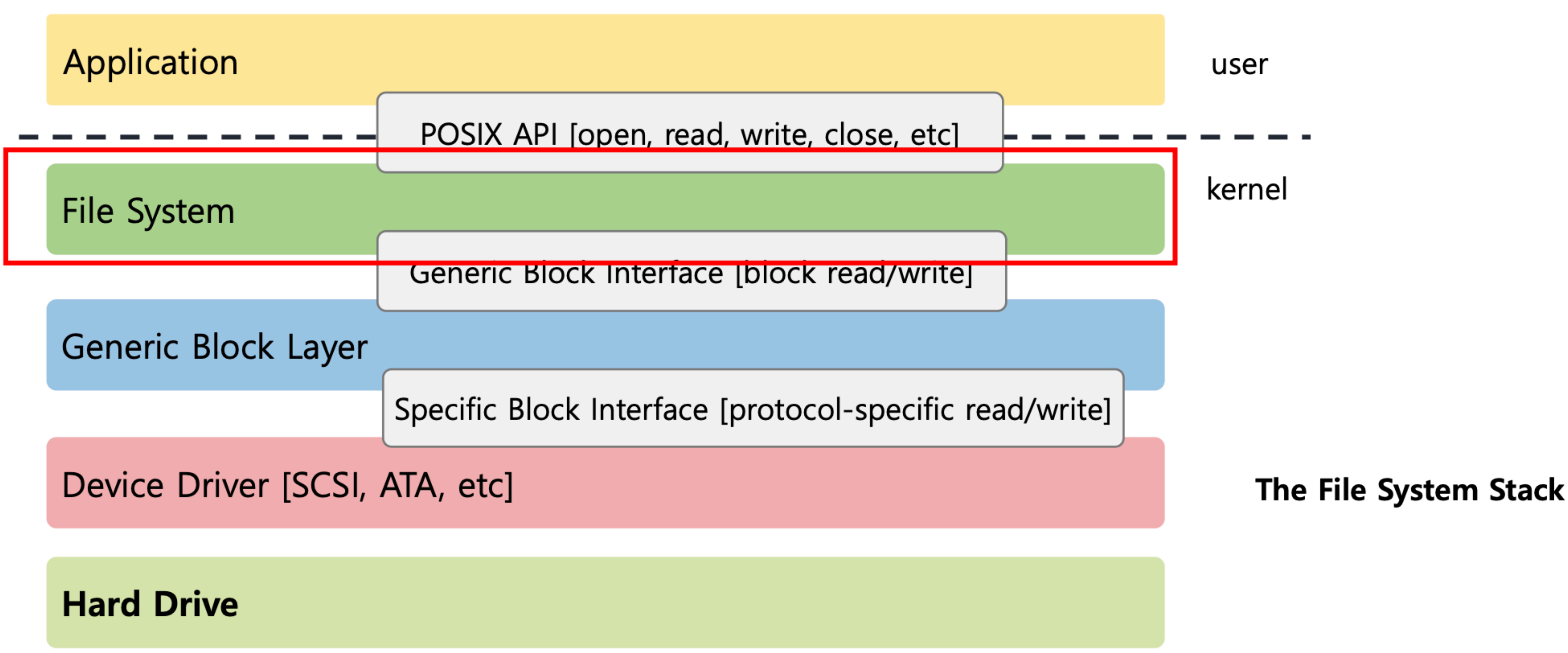
File System

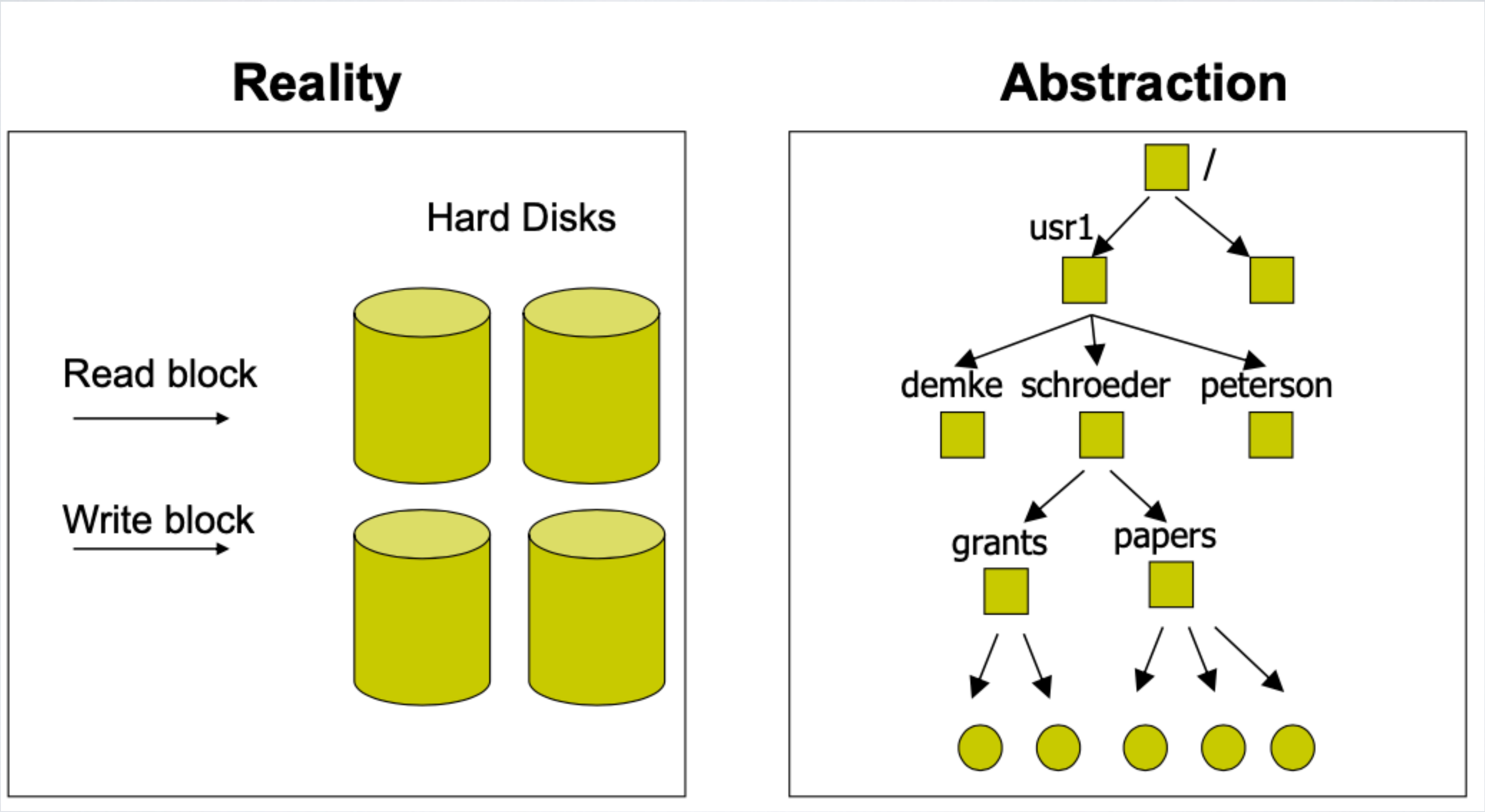
Thierry Sans

(recap) File System Abstraction

File system specifics of which disk class it is using
It issues block read/write requests to the **generic block layer**



Provide an abstraction



Goals

- Implement an abstraction (files) for secondary storage
- Organize files logically (directories)
- Permit sharing of data between processes, people, and machines
- Protect data from unwanted access (security)

Files

File - named bytes on disk that encapsulate data with some properties: contents, size, owner, last read/write time, protection, etc.

A file can also have a type

- Understood by the file system: block device, character device, link, FIFO, socket, etc.
- Understood by other parts of the OS or runtime libraries:
  text, image, source, compiled libraries (Unix `.so` and Windows `.dll`), executable, etc.

A file's type can be encoded in its name or contents

- Windows encodes type in name: `.com`, `.exe`, `.bat`, `.dll`, `.jpg`, etc.
- Unix encodes type in contents:
  magic numbers, initial characters (e.g., `#!` for shell scripts)

File Access Method

Sequential access (used by file systems - most common)
read bytes one at a time, in order (read/write next)

Random access (used by file systems)
random access given block/byte number (read/write bytes at offset n)

Indexed access (used by databases)

- file system contains an index to a particular field of each record in a file
- reads specify a value for that field and the system finds the record via the index

Record access (used by databases)

- file is array of fixed-or-variable-length records
- read/written sequentially or randomly by record number

Basic File operations

Unix

- `create(name)`
- `open(name, how)`
- `read(fd, buf, len)`
- `write(fd, buf, len)`
- `sync(fd)`
- `seek(fd, pos)`
- `close(fd)`
- `unlink(name)`

Windows

- `CreateFile(name, CREATE)`
- `CreateFile(name, OPEN)`
- `ReadFile(handle, …)`
- `WriteFile(handle, …)`
- `FlushFileBuffers(handle, …)`
- `SetFilePointer(handle, …)`
- `CloseHandle(handle, …)`
- `DeleteFile(name)`
- `CopyFile(name)`
- `MoveFile(name)`

How to Track File's Data

Disk management

- Need to keep track of where file contents are on disk
- Must be able to use this to map byte offset to disk block
- Structure tracking a file's blocks is called an index node or **inode**
- inodes must be stored on disk, too

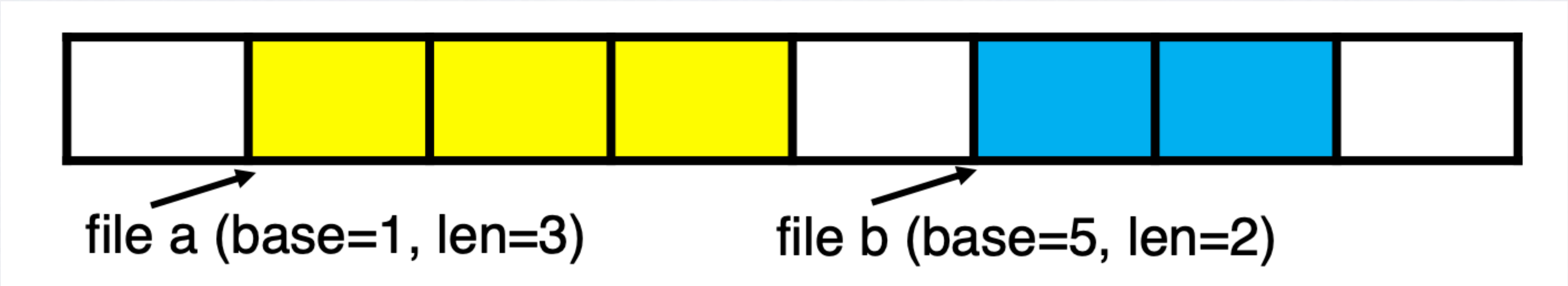
Things to keep in mind while designing file structure

- Most files are small
- Much of the disk is allocated to large files
- Many of the I/O operations are made to large files
- Want good sequential and good random access (what do these require?)

Straw Man : Contiguous Allocation

"Extent-based" - allocate files like segmented memory

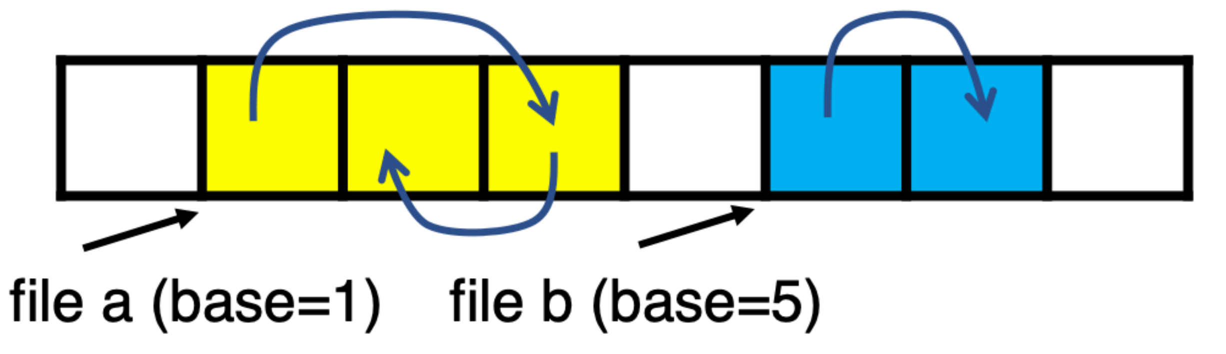
- When creating a file, make the user pre-specify its length and allocate all space at once
- Inode contents : location and size



✓ Simple, fast access, both sequential and random

◉ External fragmentation (similar to VM)

Straw Man #2 : Linked Files



Basically a linked list on disk

- Keep a linked list of all free blocks
- Inode contents : a pointer to file's first block
- In each block, keep a pointer to the next one

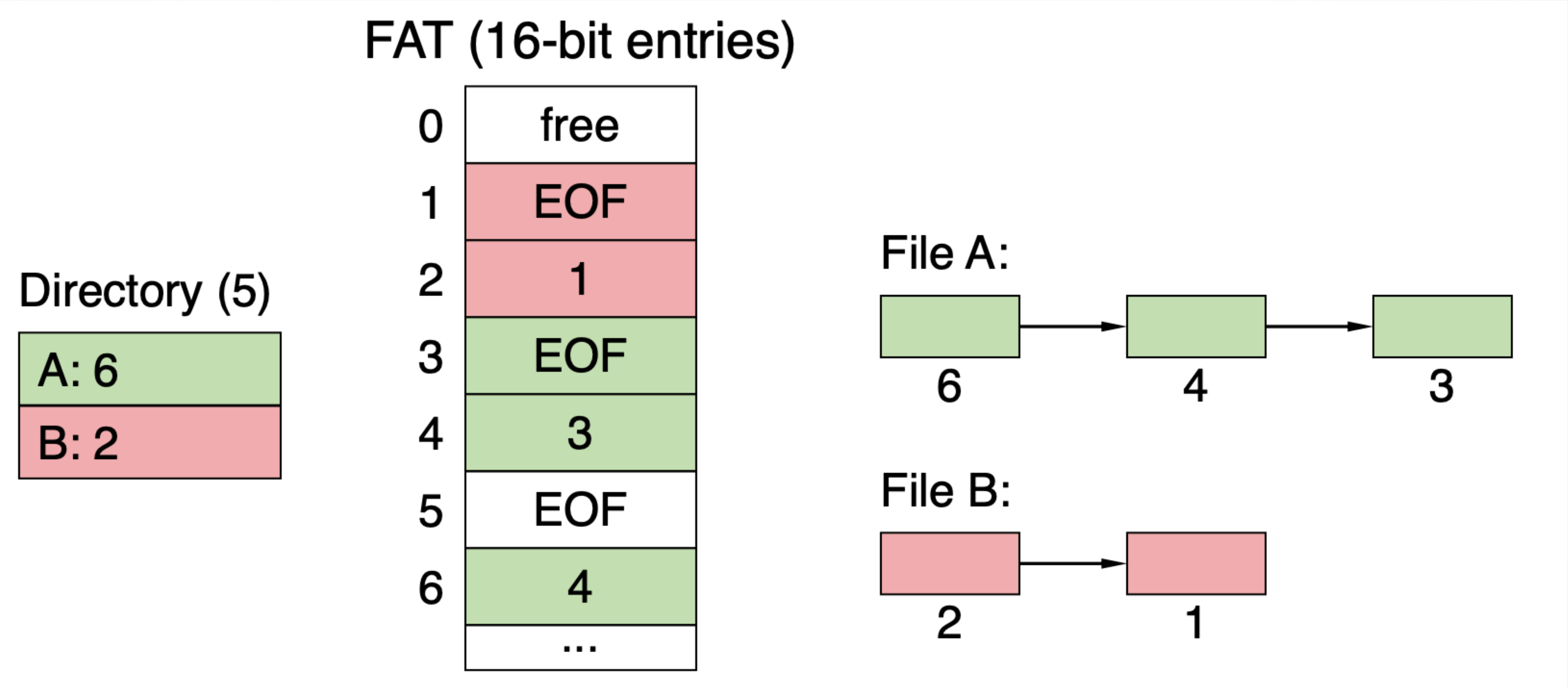
✓ Easy dynamic growth & sequential access, no fragmentation

◉ Linked lists on disk a bad idea because of access times
Random very slow (e.g., traverse whole file to find last block)
Pointers take up room in block, skewing alignment

DOS FAT (simplified)

➡ Linked files with key optimization: puts links in fixed-size "file allocation table" (FAT) rather than in the blocks

◉ Still do pointer chasing



About FAT

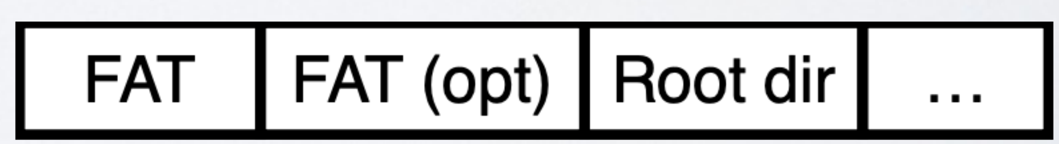
Given entry size = 16 bits (initial FAT16 in MS-DOS 3.0), what's the maximum size of the FAT? **65,536**

Given a 512 byte block, what's the maximum size of FS? **32MB**

What is the space overhead ?
2 bytes / 512 byte block = ~ 0.4%

How to protect against errors?
Create duplicate copies of FAT on disk
(state duplication a very common theme in reliability)

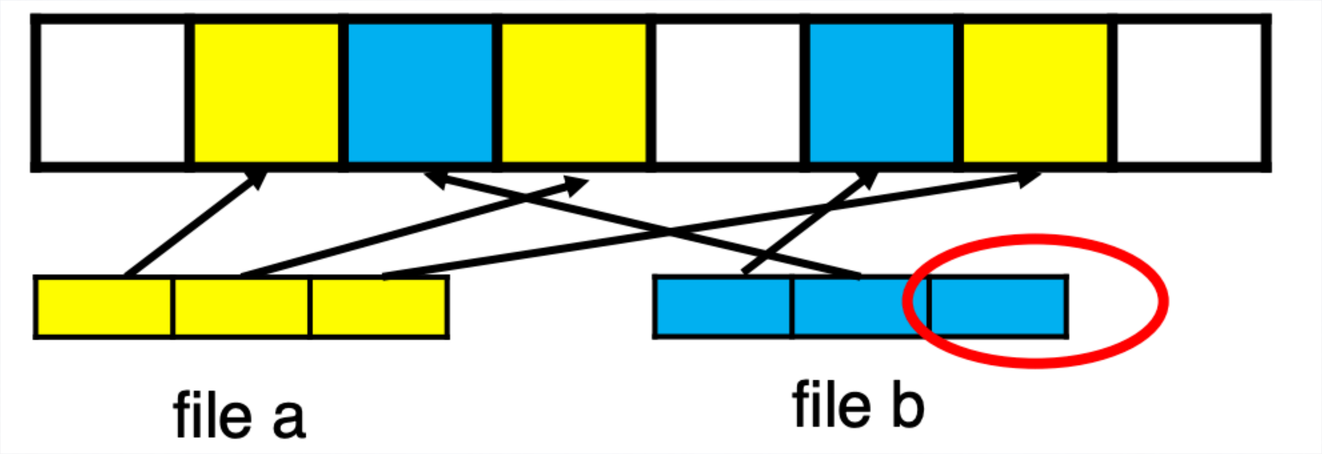
Where is root directory?
Fixed location on disk



Another Approach : Indexed Files

Each file has a table holding all of its block pointers

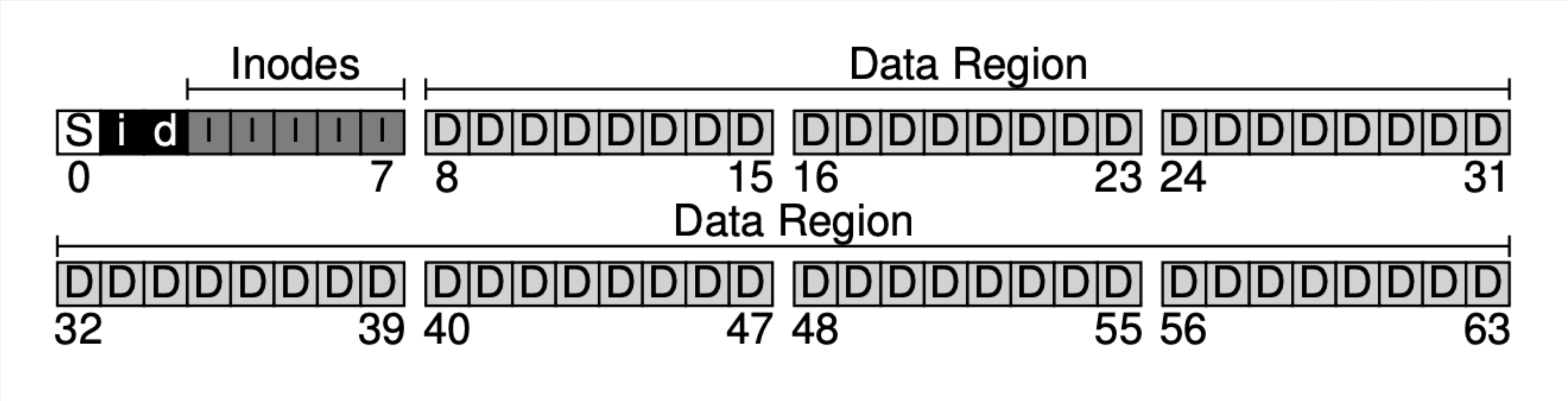
- Max file size fixed by table's size
- Allocate table to hold file’s block pointers on file creation
- Allocate actual blocks on demand using free list



✓ Both sequential and random access easy

◉ Mapping table requires large chunk of contiguous space

Unix File System



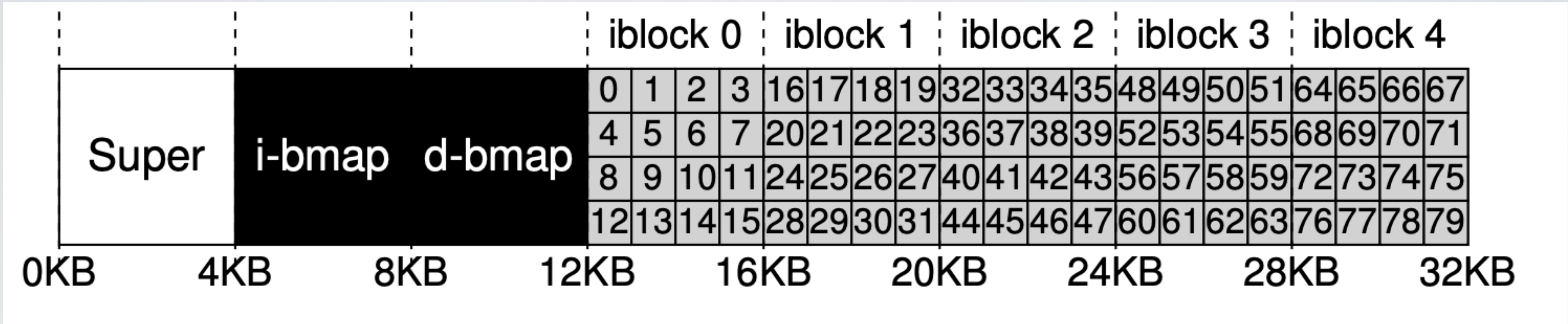
The disk is (physically) divided into sectors (usually 512 bytes per sector)

The file system is (logically) divided into blocks (e.g., 4 KB per block)

➡ Disk space is allocated in granularity of blocks

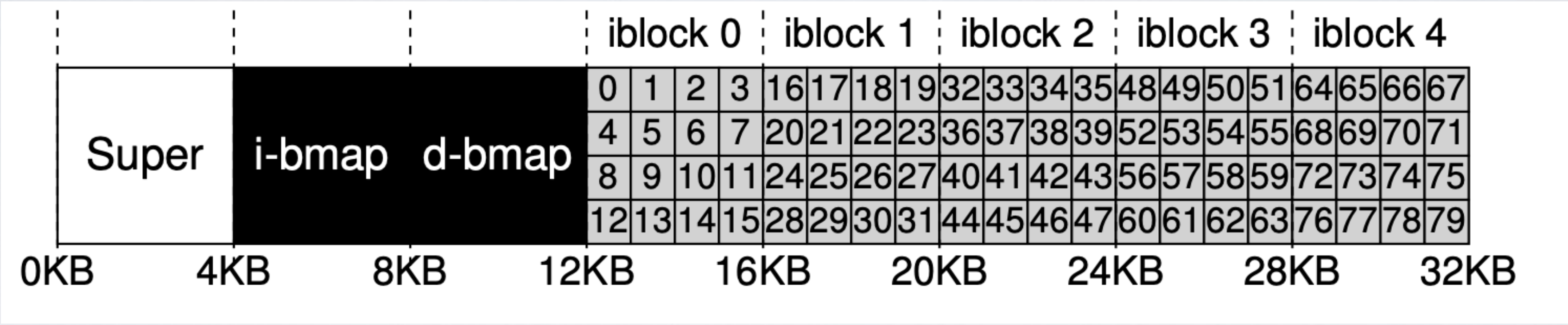
1. The data blocks "D" stored files (and directories) content
2. The inodes blocks "I" stores the inode table
3. The data bitmap "d" block d tacks which data block is free or allocated (one bit per block on the disk)
4. The inode bitmap "i" block i tracks which inode is free or allocated (one bit per inode)
5. The Superblock "S" (a.k.a Master Block or partition control block) contains:
    - a magic number to identify the file system type
    - the number of blocks dedicated to the two bitmaps and inodes

The Inode Table



- **Physical Disk capacity in our example** (64 blocks of 4KB each)
  4 x 64 = 256 KB
- **Logical capacity (8 blocks are reserved for the inode table)**
  4 x 56 = 224 KB (the actual data storage space)
- **Maximum number of inodes** (each inode is 256 bytes)
  (5 * 4 * 1024) / 256 = 80 inodes (i.e max number of files)
- **Size of the inode bitmap** (1 bit per inode)
  1 x 80 inodes = 80 bits (out of 32K bits)
- **Size of the data bitmap** (1 bit per storage block)
  1 bit x 56 blocks = 56 bits (out of 32K bits, max data storage 128 MB)

Decoding inodes



What disk sector to read to retrieve inode 32?

1. Calculate the offset (each inode is 256 bytes)
   $32 \times 256 = 8{,}192$
2. Add the start of the address of the inode table (12K)
   $8{,}192 + 12 \times 1{,}024 = 20{,}480$ (20 KB)
3. Find the corresponding disk sector (each sector is 512 bytes)
   $(20 \times 1{,}024) / 512 = 40$

Unix Inode (simplified)

Size	Name	Description
2	`mode`	can the file be read/written/executed
2	`uid`	file owner id
4	`size`	the file size in bytes
4	`time`	time the file was last accessed
4	`ctime`	time when the file created
4	`mtime`	time when the file was last modified
4	`dtime`	time when the inode was deleted
2	`gid`	file group owner id
2	`links_count`	number of hard links pointing to this file
4	`blocks`	the number of blocks allocated to this file
60	`block`	disk pointers (15 in total)
4	`file_acl`	ACL permissions
4	`dir_acl`	ACL permissions

Block pointers and maximum file size

So far, each inode has 15 block pointers

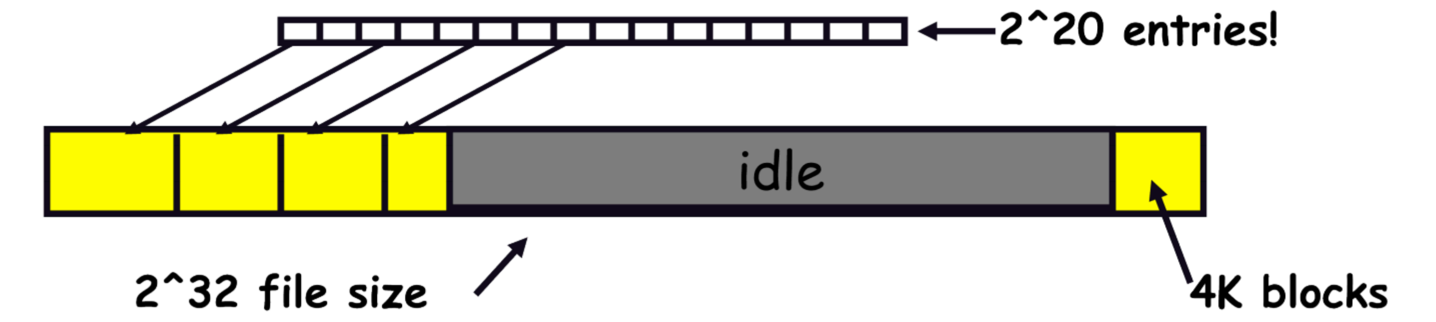
➡ The maximum file size can be 15 * 4 KB = 60 KB (only?!)

- Should we increase the number of block pointers to increase the file size?

More issues with indexed Files

Large file size with lots of unused entries means

- The mapping table requires large chunk of contiguous space

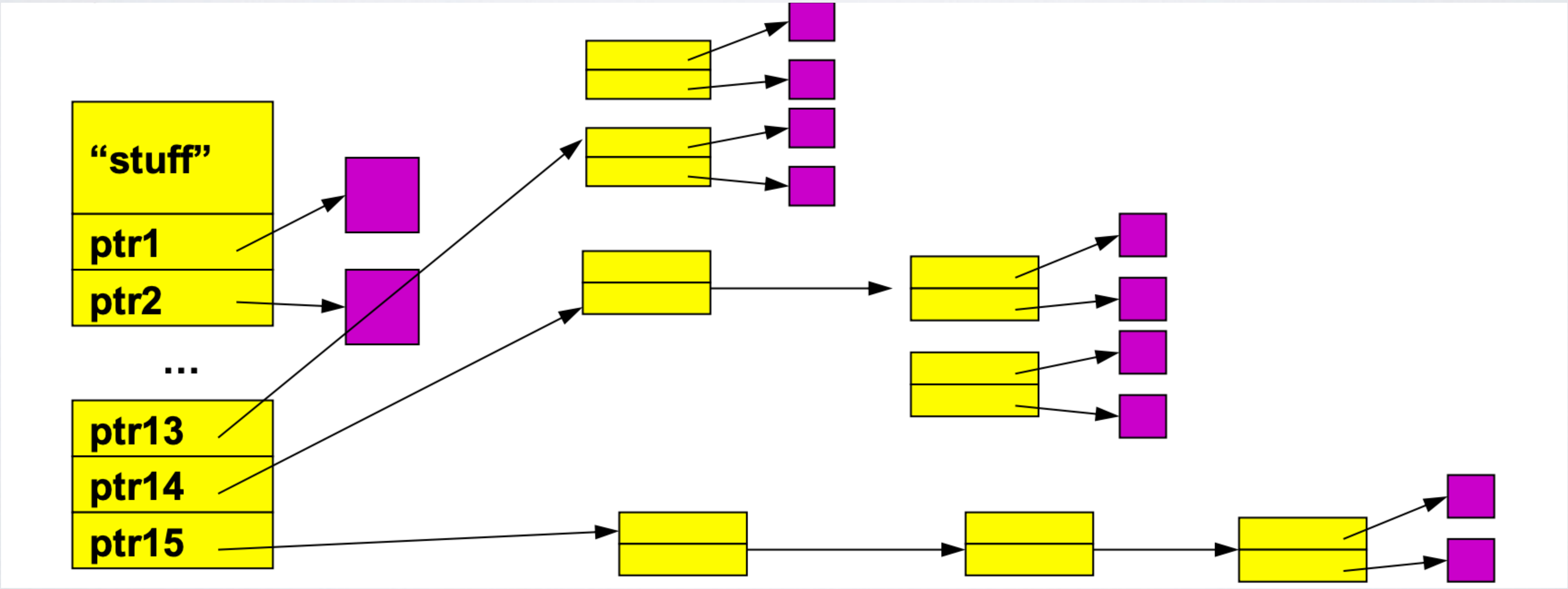


➡ Solution : mapping table structured as a multi-level index array

- but ... you know the story (similar to VM two-level page lookup)

Multi-level indexed files : Unix File System

- First 12 pointers are direct blocks
  solve problem of first blocks access slow
- Then single, double, and triple indirect block pointers



File size with multi-level indexed files

File size using 12 direct blocks : 12×4 KB = 48 KB

➡ Adding single indirect block : $(12 + 1024) \times 4$ KB ~ 4 MB

➡ Adding a double indirect block :
$(12 + 1024 + 1024^2) \times 4$ KB) ~ 4 GB

➡ Adding a triple indirect block :
$(12 + 1024 + 1024^2 + 1024^3) \times 4$ KB) ~ 4 TB

Rationale behind multi-level index files

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

Directories

Directories serve two purposes

- For users, they provide a structured way to organize files by using digestible names rather than inode numbers directly
- For the File System, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk

Basic Directory Operations

Unix

- ➡ Directories implemented in file and a C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

Windows

- ➡ Explicit dir operations
  - `CreateDirectory(name)`
  - `RemoveDirectory(name)`
  - `FindFirstFile(pattern)`
  - `FindNextFile()`

A Short History of Directories

Approach 1 : Single directory for entire system

- Put directory at known location on disk
- Directory contains `hname`, `inumberi` pairs
- If one user uses a name, no one else can
- Many ancient personal computers work this way

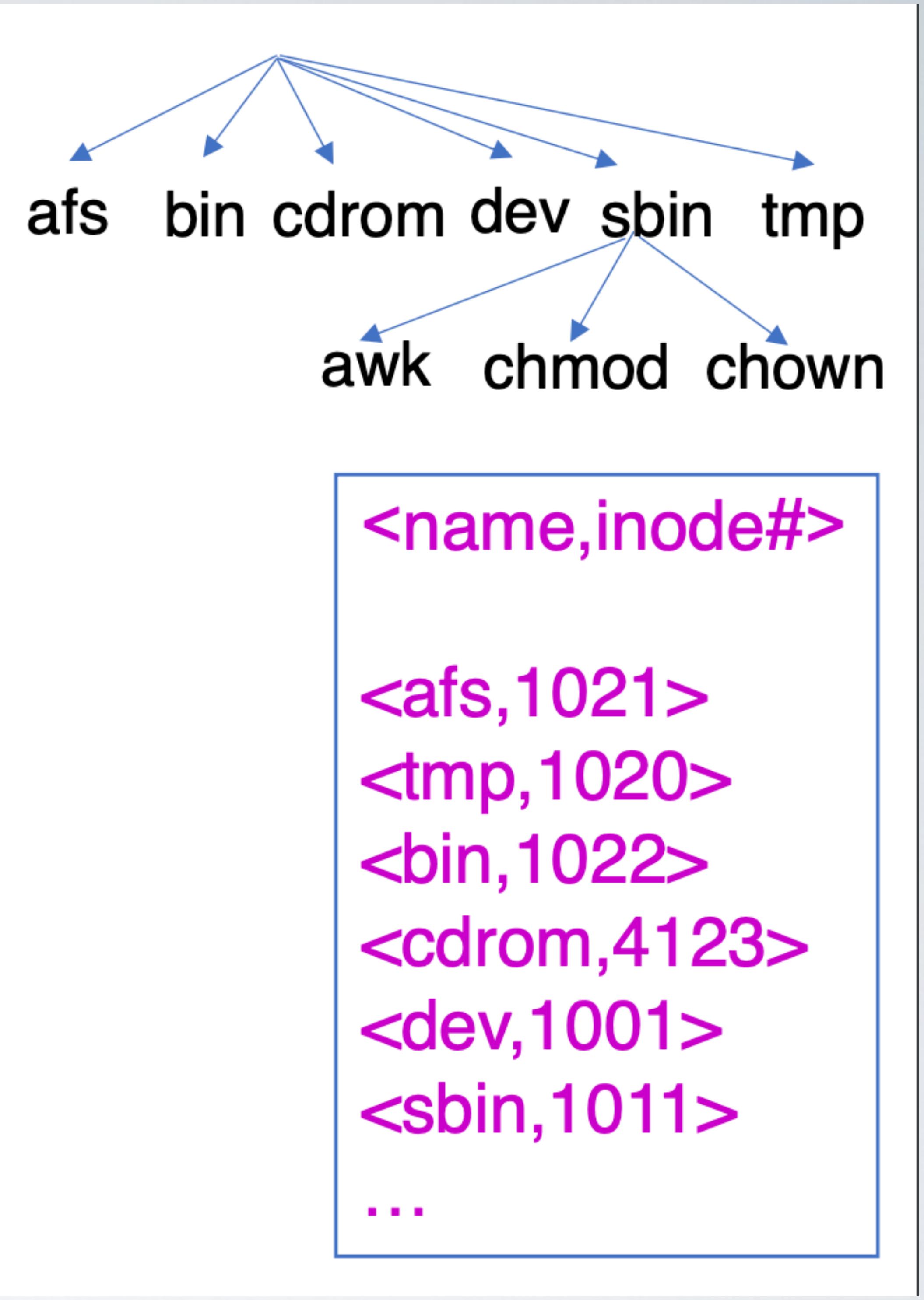
Approach 2 : Single directory for each user

- Still clumsy, and `ls` on 10,000 files is a real pain

Approach 3 : Hierarchical name spaces

- Allow directory to map names to files or other directories
- File system forms a tree (or graph, if links allowed)
- Large name spaces tend to be hierarchical
  (ip addresses, domain names, scoping in programming languages, etc.)

Hierarchical Directory



➡ Used since CTSS (1960s)
Unix picked up and used really nicely

Directories stored on disk just like regular files

- Special inode type byte set to directory
- User's can read just like any other file
- Only special syscalls can write
- Inodes at fixed disk location
- File pointed to by the index may be another directory
- Makes FS into hierarchical tree

✓ Simple, plus speeding up file ops speeds up dir ops!

Naming Magic

Bootstrapping
Root directory always inode #2 (0 and 1 historically reserved)

Special names

- Root directory : "`/`"
- Current directory : "`.`"
- Parent directory : "`..`"

Some special names are provided by shell, not FS

- User's home directory : "~"
- Globing : "`foo.*`" (expands to all files starting "`foo.`")

Using the given names, only need two operations to navigate the entire name space

- `cd name` : move into (change context to) directory name
- `ls` : enumerate all names in current directory (context)

Unix inodes and path search

Unix inodes are not directories

- Inodes describe where on the disk the blocks for a file are placed
- Directories are files, so inodes also describe where the blocks for directories are placed on the disk

Directory entries map file names to inodes

1. To open "`/one`", use Master Block to find inode for "`/`" on disk
2. Open "`/`", look for entry for "`one`"
3. This entry gives the disk block number for the inode for "`one`"
4. Read the inode for "`one`" into memory
5. The inode says where first data block is on disk
6. Read that block into memory to access the data in the file

Default Context : Working Directory

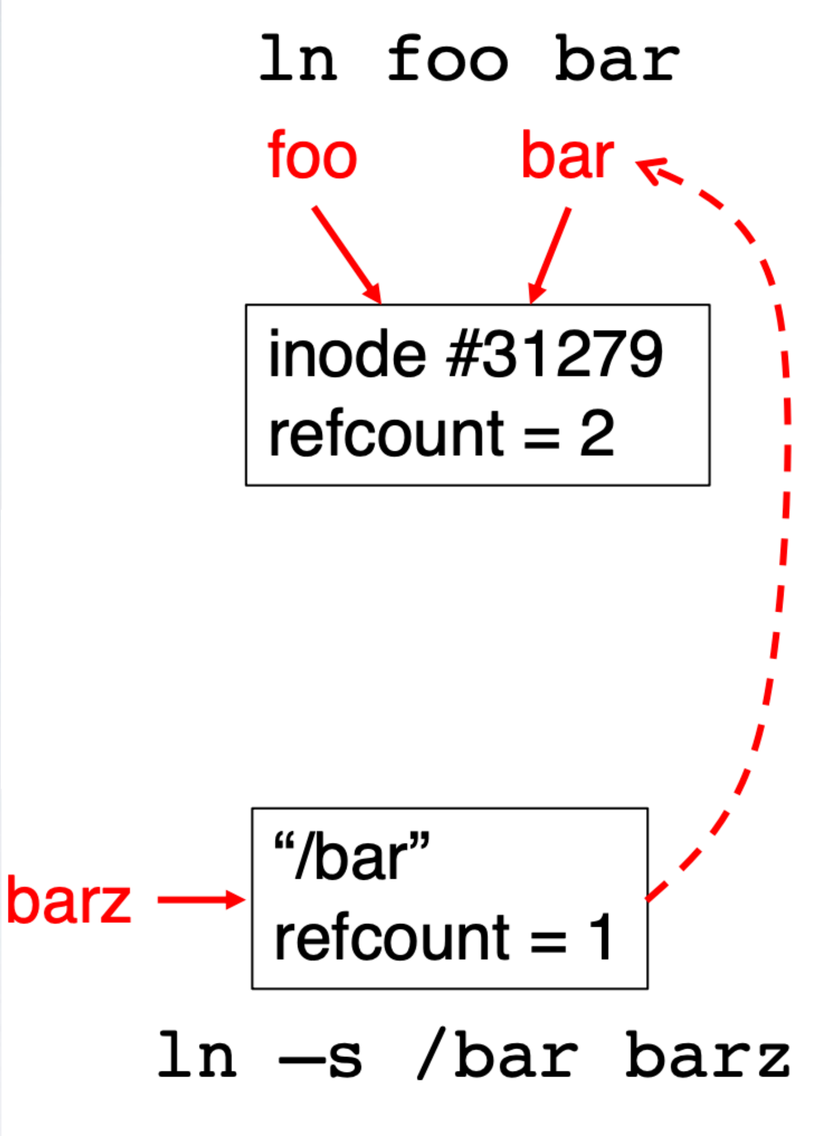
Cumbersome to constantly specify full path names

- In Unix, each process has a "current working directory" (cwd)
- File names not beginning with "/" are assumed to be relative to cwd; otherwise translation happens as before

Shells track a default list of active contexts

- A "search path" for programs you run
- Given a search path A:B:C, the shell will check in A, then B, then C
- Can escape using explicit paths: "./foo"

Hard and Soft Links (synonyms)



More than one dir entry can refer to a given file

- Hard link creates a synonym for file
- Unix stores count of pointers ("hard links") to inode
- If one of the links is removed (e.g., rm), the data are still accessible through any other link that remains
- If all links are removed, the space occupied by the data is freed

Soft symbolic links = synonyms for names

- Point to a file/dir name, but object can be deleted from underneath it (or never exist)
- Unix implements like directories: inode has special "symlink" bit set and contains name of link target
- When the file system encounters a soft link it automatically translates it (if possible).

File Buffer Cache

Applications exhibit significant locality for reading and writing files

➡ Idea : cache file blocks in memory to capture locality
Called the file buffer cache

- Cache is system wide, used and shared by all processes
- Reading from the cache makes a disk perform like memory
- Even a small cache can be very effective

◉ Issues

- The file buffer cache competes with VM (tradeoff here)
- Like VM, it has limited size
- Need replacement algorithms again (LRU usually used)

Caching Writes

On a write, some applications assume that data makes it through the buffer cache and onto the disk

➡ As a result, writes are often slow even with caching

OSes typically do write back caching

- Maintain a queue of uncommitted blocks
- Periodically flush the queue to disk (30 second threshold)
- If blocks changed many times in 30 secs, only need one I/O
- If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed

✓ Unreliable, but practical

- On a crash, all writes within last 30 secs are lost
- Modern OSes do this by default; too slow otherwise
- System calls (Unix: fsync) enable apps to force data to disk

Read Ahead

Many file systems implement "read ahead"

- FS predicts that the process will request next block
- FS goes ahead and requests it from the disk
- This can happen while the process is computing on previous block
- Overlap I/O with execution
- When the process requests block, it will be in cache
- Compliments the disk cache, which also is doing read ahead

✓ For sequentially accessed files can be a big win

- Unless blocks for the file are scattered across the disk
- File systems try to prevent that, though (during allocation)

File Sharing

File sharing has been around since timesharing

- Easy to do on a single machine
- PCs, workstations, and networks get us there (mostly)

➡ File sharing is important for getting work done (basis for communication and synchronization)

- Two key issues when sharing files
  1. Semantics of concurrent access
     - What happens when one process reads while another writes?
     - What happens when two processes open a file for writing?
     - What are we going to use to coordinate?
  2. Protection

Protection

File systems implement a protection system

- Who can access a file
- How they can access it

➡ A protection system dictates whether a given action performed by a given subject on a given object should be allowed

- You can read and/or write your files, but others cannot
- You can read "`/etc/motd`", but you cannot write it

Representing Protection

Access Control Lists (ACL)

For each object, maintain a list of subjects and their permitted actions

Capabilities

For each subject, maintain a list of objects and their permitted actions

Objects

Subjects

	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

Capability

ACL

ACLs and Capabilities

Approaches differ only in how the table is represented

- ➡ Capabilities are easier to transfer
  They are like keys, can handoff, does not depend on subject
- ➡ But ACLs are easier to manage in practice
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem

ACLs have a problem when objects are heavily shared

- The ACLs become very large
- Use groups (e.g. Unix)

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*
- Sina Meraji - teaching CS 369 at *University of Toronto*