Operating Systems

Operating Systems

Sina Meraji

U of T

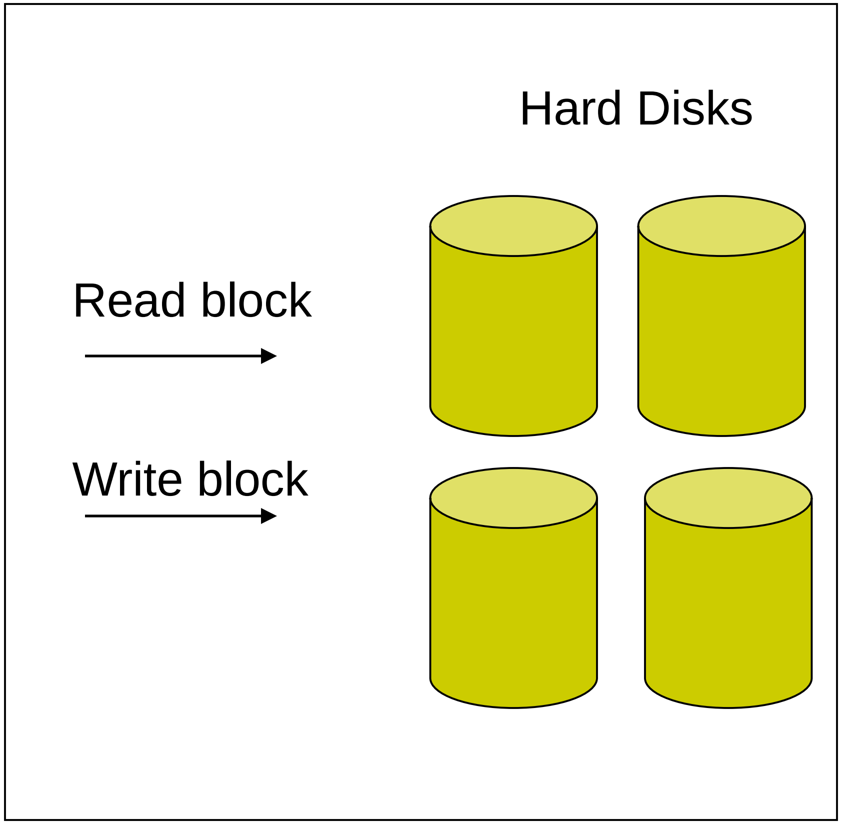
New topic:

- File systems!

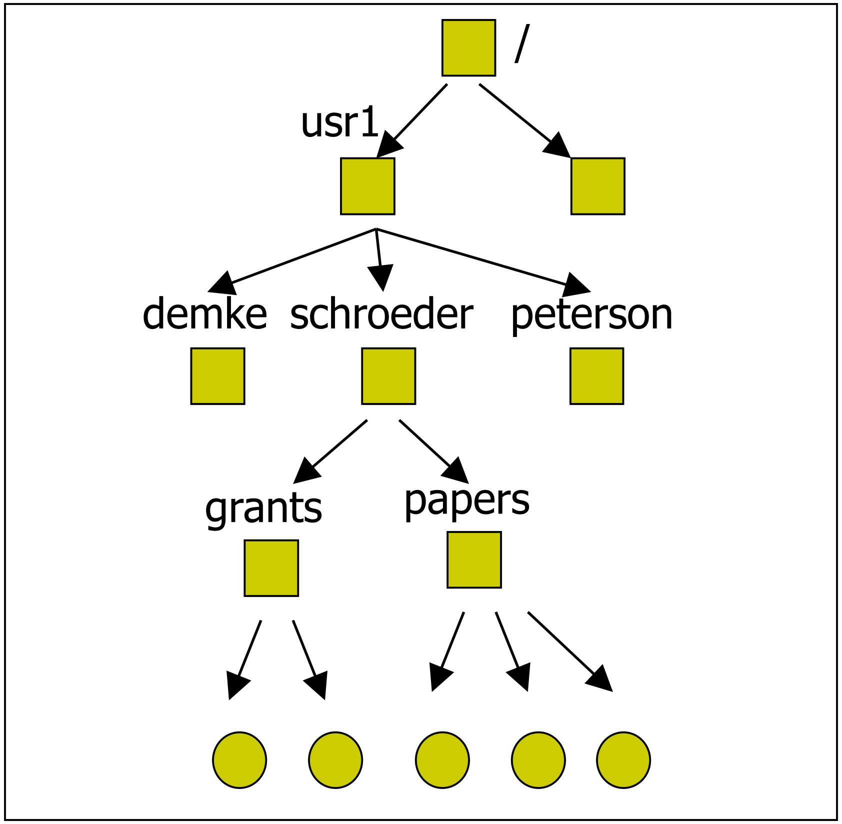
What do file systems do?

- They provide a nice abstraction of storage:

Reality



Abstraction



File Systems

- File management systems
  - 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)

File Concept

- A file is named collection of data with some **attributes**
  - Name
  - Owner
  - Location
  - Size
  - Protection
  - Creation time
  - Time of last access

File Types

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers

- A file's type can be encoded in its name or contents
  - Windows encodes type in name
    - .exe, .doc, .jpg, etc.
  - Unix encodes type in contents (sometimes)
    - Magic numbers, initial characters (e.g., #! for shell scripts)

le Operation

Where in the file do write and read operations operate?

Unix (C library)

- Create
- Write
- Read
- Repositioning within file
- Delete
- Truncating a file
- Open
- Close

- creat(name)
- write(fd, buf, len)
- read(fd, buf, len)
- seek(fd, pos)
- unlink(name)
- truncate(fd, length)
- open(name, mode)
- close(fd)

File Access Methods

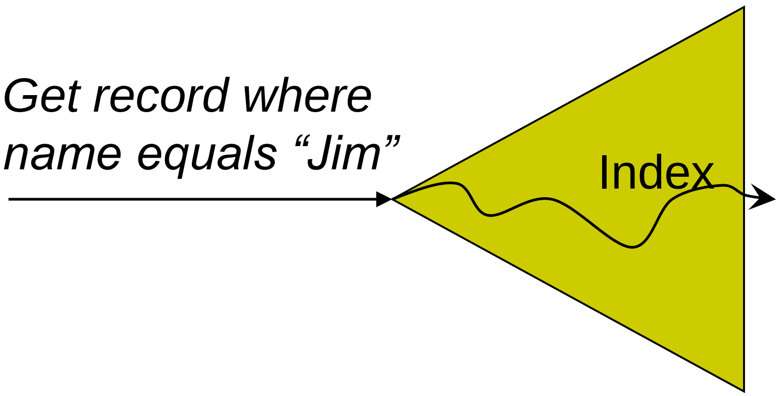
- General-purpose file systems support simple methods
  - Sequential access – read bytes one at a time, in order
    - read next
    - write next
  - Direct access – random access given block/byte number
    - read n (byte at offset n)
    - write n
- What does Unix use?
  - both

Unix (C library)

- read(fd, buf, len)
- write(fd, buf, len)
- seek(fd, pos)

File Access Methods

- Database systems support more sophisticated methods
  - Record access
  - Indexed access



John	75837	A
David	63548	C+
Alice	92746	A

- Modern OS file systems support only simple methods (direct access, sequential access)

Conceptual File Operation

- Create
- Write
- ...
- ...ng
- Dele...
- Truncating a file
- Open
- Close

Why do we need open and close operations?

Unix (C library)

- creat(name)
- write(fd, buf, len)
- read(fd, buf, len)
- seek(fd, pos)
- unlink(name)
- truncate(fd, length)
- open(name, mode)
- close(fd)

Handling operations on files

- Involves searching the directory for the entry associated with the named file
  - when the file is first used actively, store its attribute info in a system-wide open-file table; the index into this table is used on subsequent operations ⇒ no searching

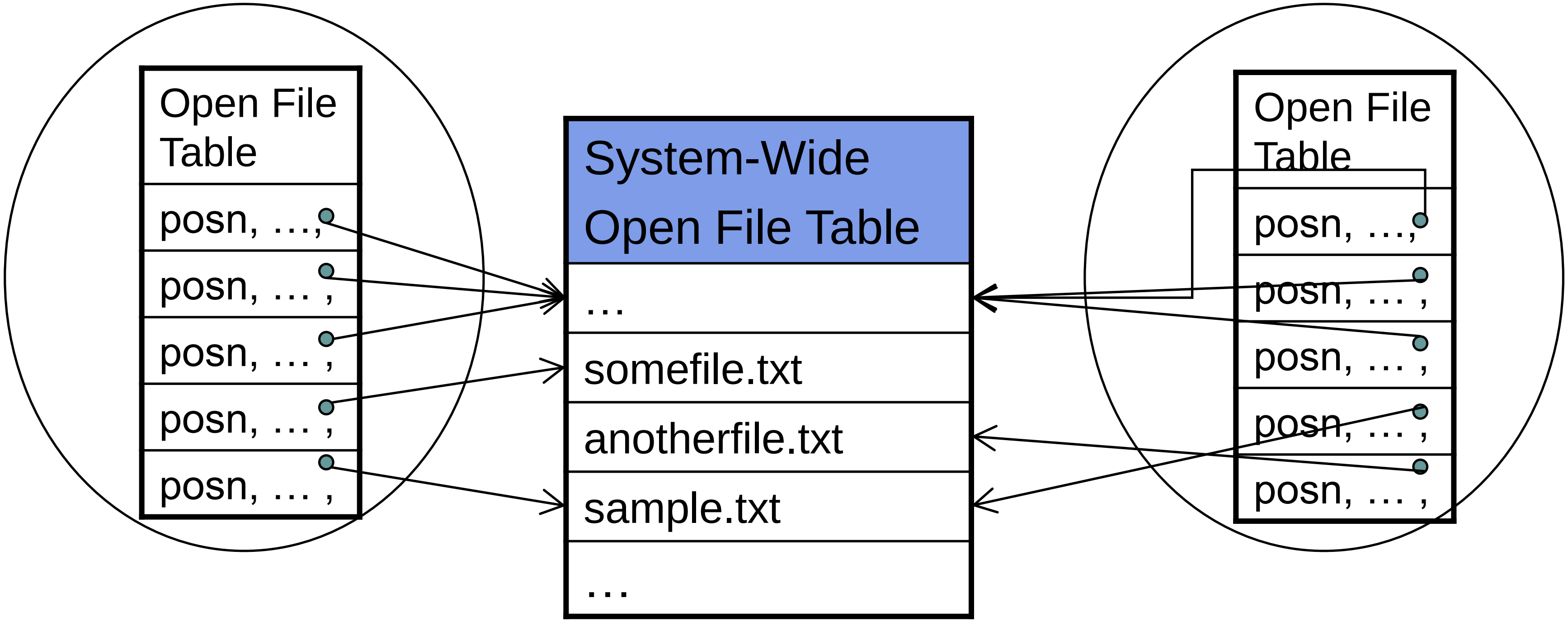
Unix example (open, read, write are syscalls):

```
main() {
    char onebyte;
    int fd = open("sample.txt", "r");
    read(fd, &onebyte, 1);
    write(STDOUT, &onebyte, 1);
    close(fd);
}
```

Open File Table
…
…
sample.txt
…
…

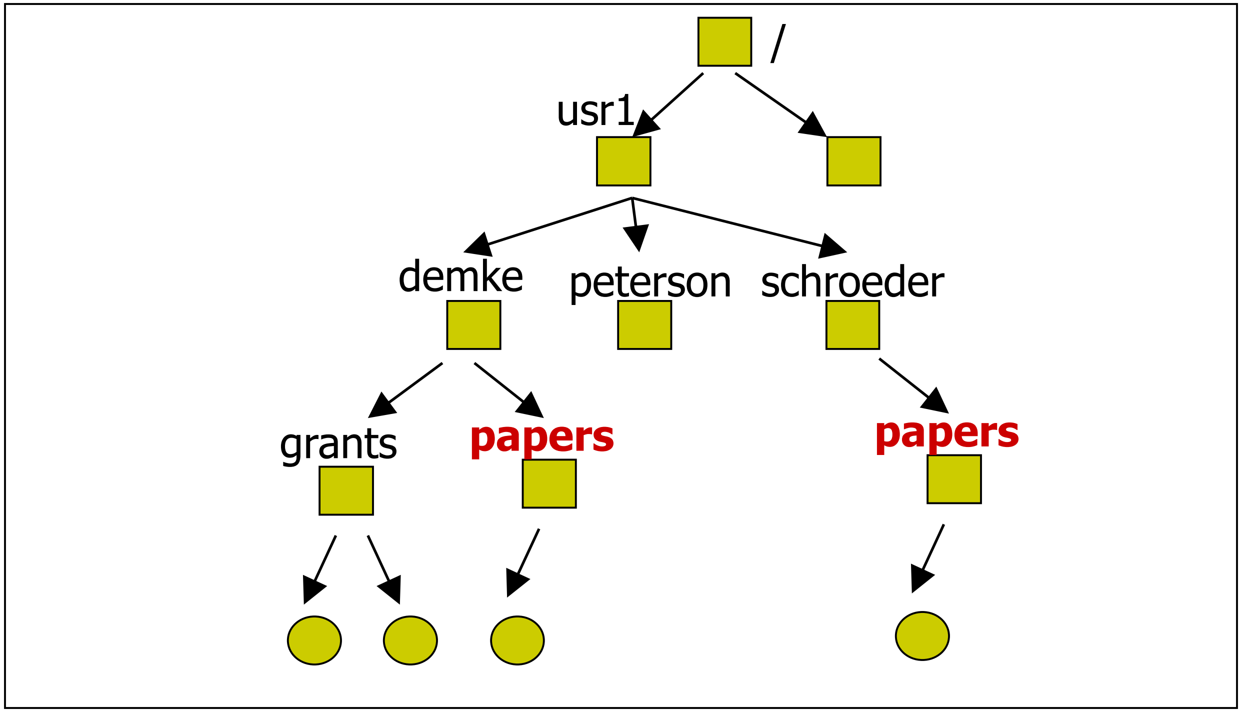
Shared open files

- There are actually 2 levels of internal tables
  - a per-process table of all files that each process has open (this holds the current file position for the process)
  - each entry in the per-process table points to an entry in the system-wide open-file table (for process independent info)



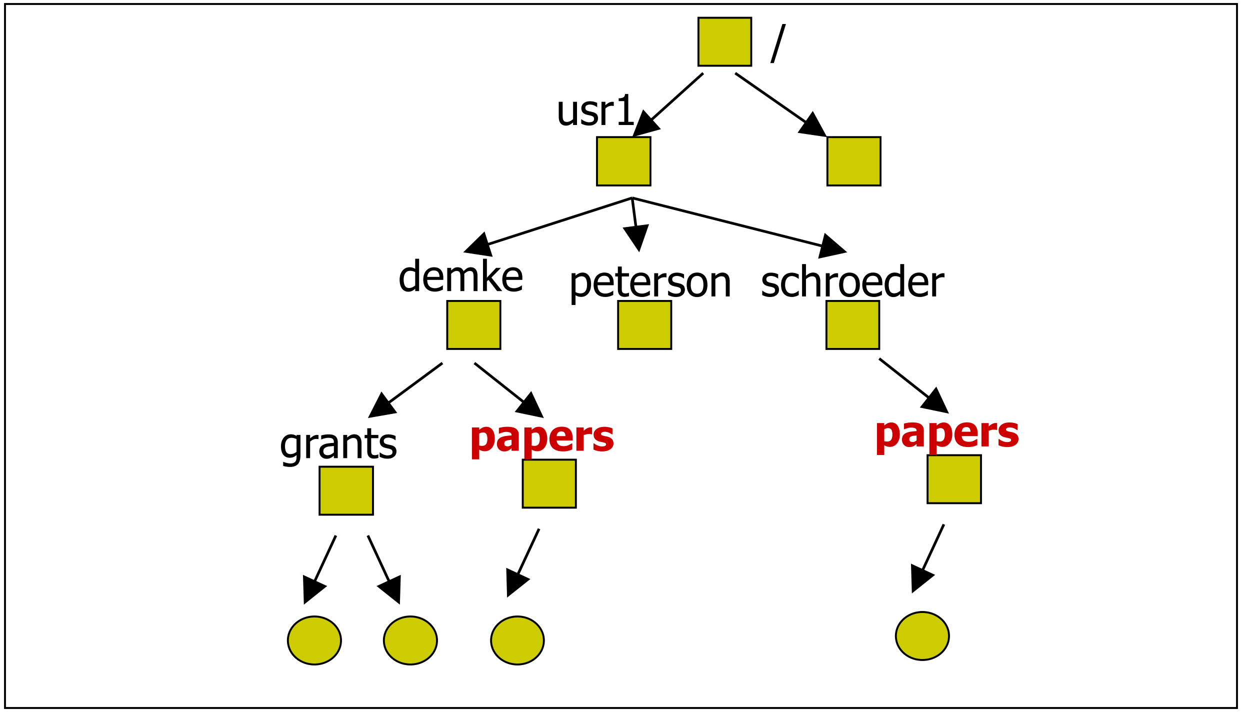
Directories

- Directories serve multiple purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
  - Also store information about files (owner, permission, etc.)



Directories

- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, …)



What is a directory at the OS level?

- A directory is a list of entries – names and associated *metadata*
  - Metadata is not the data itself, but information that describes properties of the data (size, protection, location, etc.)
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
- Directories typically stored in files
  - Only need to manage one kind of secondary storage unit

Operations on Directories

- Search
  - Find a particular file within directory
- Create file
  - Add a new entry to the directory
- Delete file
  - Remove an entry from the directory
- List directory
  - Return file names and requested attributes of entries
- Update directory
  - Record a change to some file's attributes

Example Directory Operations

Unix

- Directories implemented in files
  - Use file ops to create dirs
- C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR *dir)
  - seekdir(DIR *dir)
  - closedir(DIR *dir)

Path Name Translation

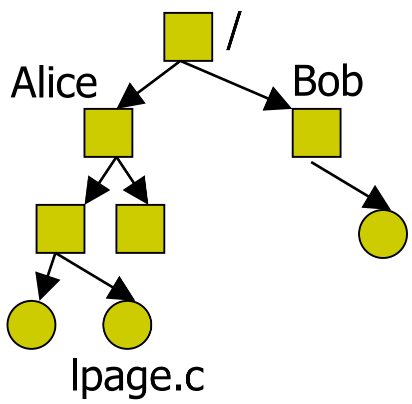
Why do we need open and close operations?

- Let’s say you want to open “/one/tw
- What does the file system do?
  - Open directory “/” (the root, well known, ca ays find)
  - Search for the entry “one”, get location of e” (in directory entry)
  - Open directory “one”, search for “two”, location of “two”
  - Open directory “two”, search for “thre get location of “three”
  - Open file “three”
- Systems spend a lot of time walking directory paths
  - This is why open is separate from read/write
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share “/a” prefix

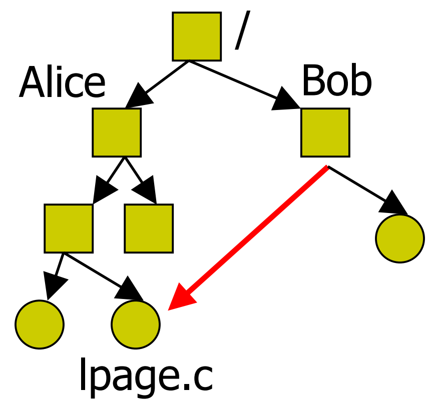
Possible Directory Implementations

- single-level, two-level, tree-structured
- acyclic-graph directories: allows for shared directories
  - the *same* file or subdirectory may be in 2 different directories

Tree-structured:



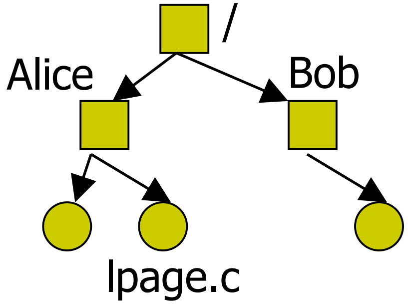
Acyclic graph:



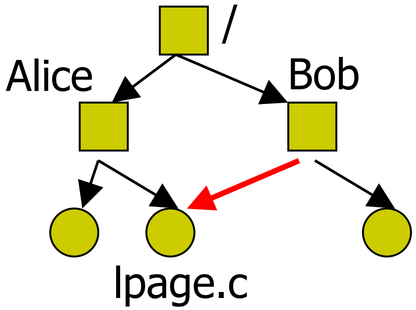
File Links

- Sharing can be implemented by creating a new directory entry called a *link* : a pointer to another file or subdirectory
  - **Symbolic, or soft, link**
  - **Hard links**

Tree-structured:

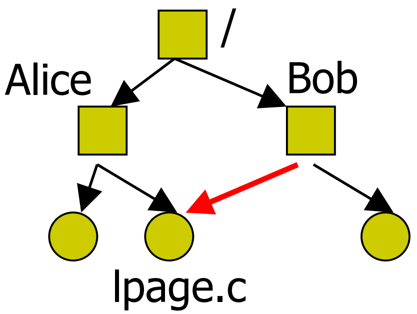


Acyclic graph:



Symbolic vs. Hard Links

- Symbolic, or soft, link:
  - Directory entry contains “true” path to the file
- Hard links:
  - Second directory entry identical to the first



'~Alice' directory

File Name	Start Block	Type
…	…	…
lpage.c	42	file
...	…	…

'~Bob' directory (hard link)

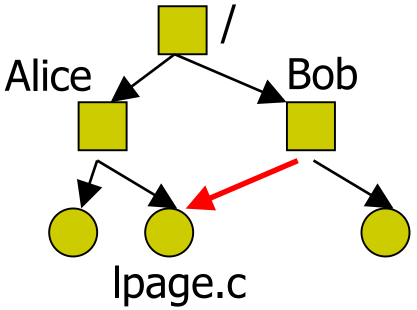
File Name	Start Block	Type
…	…	…
lpage.c	42	file
…	…	…

'~Bob' directory (soft link)

File Name	Start Block	Type
…	…	…
lpage.c	215	link
…	…	…

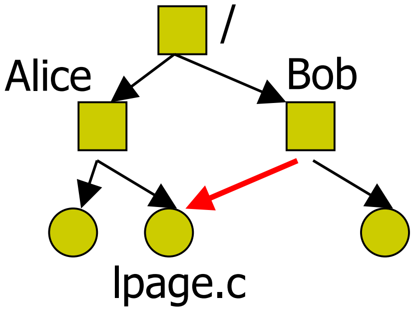
Issues with Acyclic Graphs

- With links, a file may have multiple absolute path names
  - Traversing a file system should avoid traversing shared structures more than once
- Sharing can occur with duplication of information, but maintaining consistency is a problem
  - E.g. updating permissions in directory entry with hard link



Issues with Acyclic Graphs

- Deletion: when can the space allocated to a shared file be deallocated and reused?
  - Somewhat easier to handle with symbolic links
    - Deletion of a link is OK; deletion of the file entry itself deallocates space and leaves the link pointers dangling
  - Keep a reference count for hard links

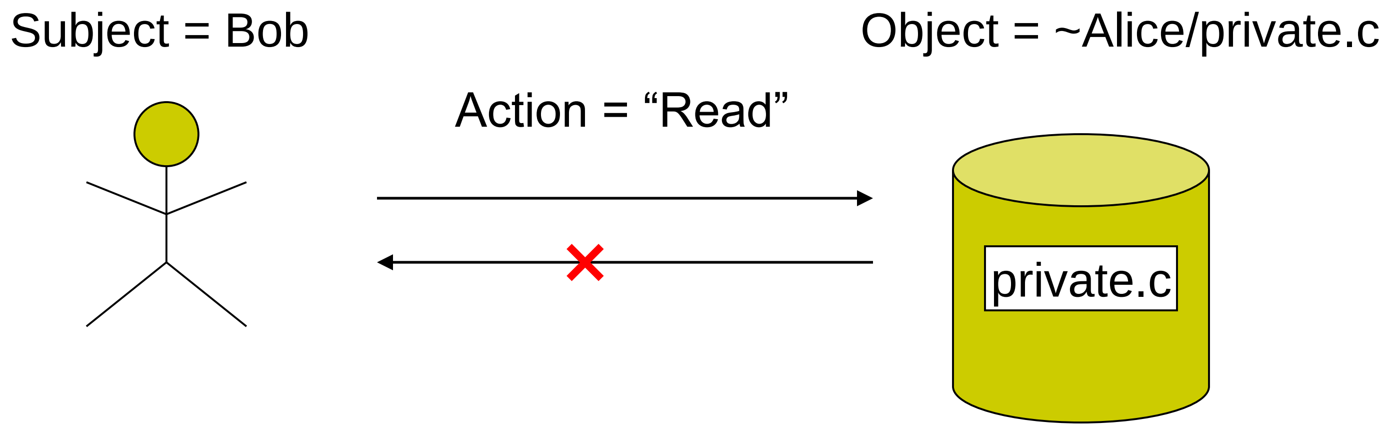


File Sharing

- 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 incredibly important for getting work done
  - Basis for communication and synchronization
- Two key issues when sharing files
  - Semantics of concurrent access
    - What happens when one process reads while another writes?
    - What happens when two processes open a file for writing?
  - Protection

Protection

- File systems implement some kind of protection system
  - Who can access a file
  - How they can access it
- A protection system dictates whether given action 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



Types of Access

- None
- Knowledge
- Execution
- Reading
- Appending
- Updating
- Changing Protection
- Deletion

- Unix provides only Read/Write/Execute permissions

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

ACL

Objects

Subjects

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

Capability

ACLs and Capabilities

- The approaches differ only in how the table is represented
  - What approach does Unix use?
- Capabilities are easier to transfer between users
  - They are like keys, can handoff, does not depend on subject
- In practice, ACLs are easier to manage
  - 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

File System Implementation

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A "Master Block" determines location of root directory (aka *partition control block, superblock)*
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

Directory Implementation

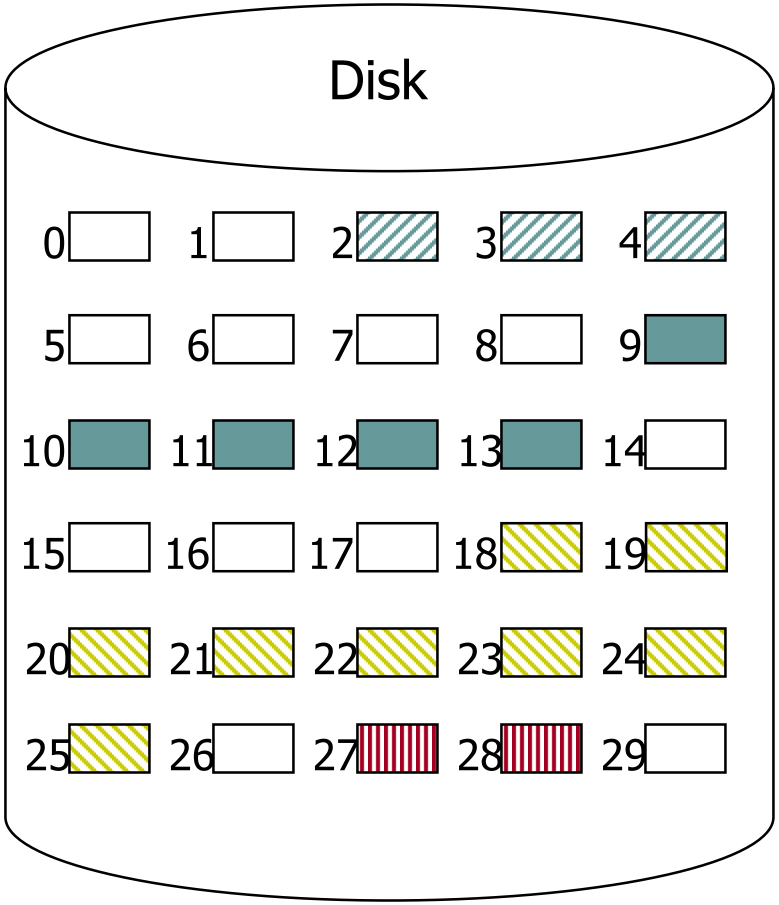
- Option 1: Linear List
  - Simple list of file names and pointers to data blocks
  - Requires linear search to find entries
  - Easy to implement, slow to execute
    - And directory operations are frequent!
- Option 2: Hash Table
  - Add hash data structure to linear list
  - Hash file name to get pointer to the entry in the linear list

Disk Layout Strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation
   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction
2. Linked, or chained, structure
   - Each block points to the next, directory points to the first
   - Good for sequential access, bad for all others
3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)

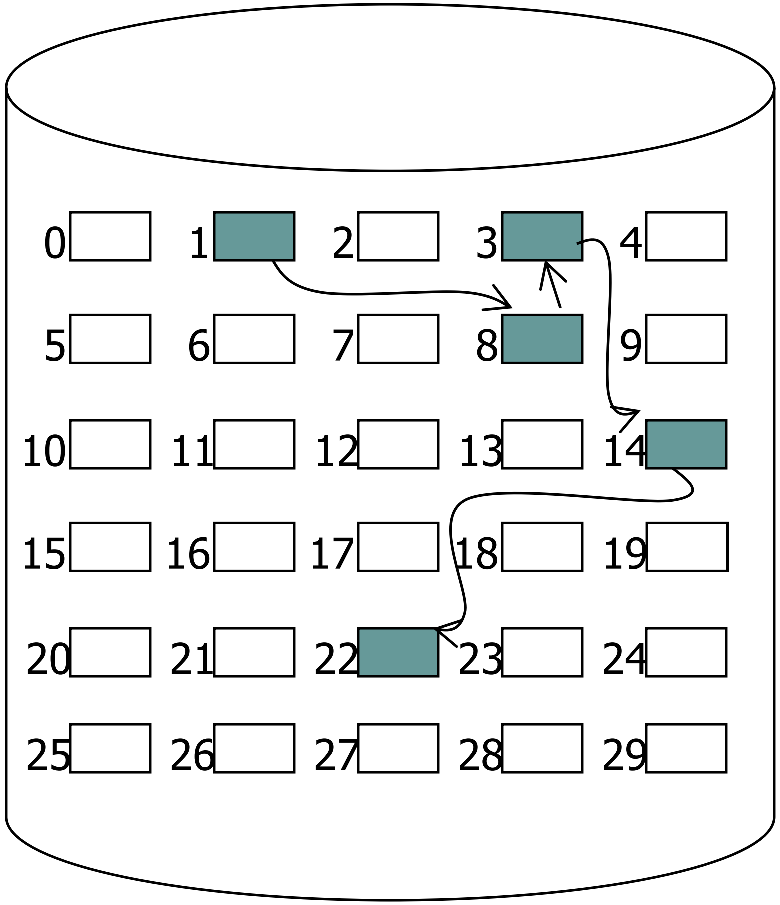
Contiguous Allocation



directory

File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

Linked Allocation

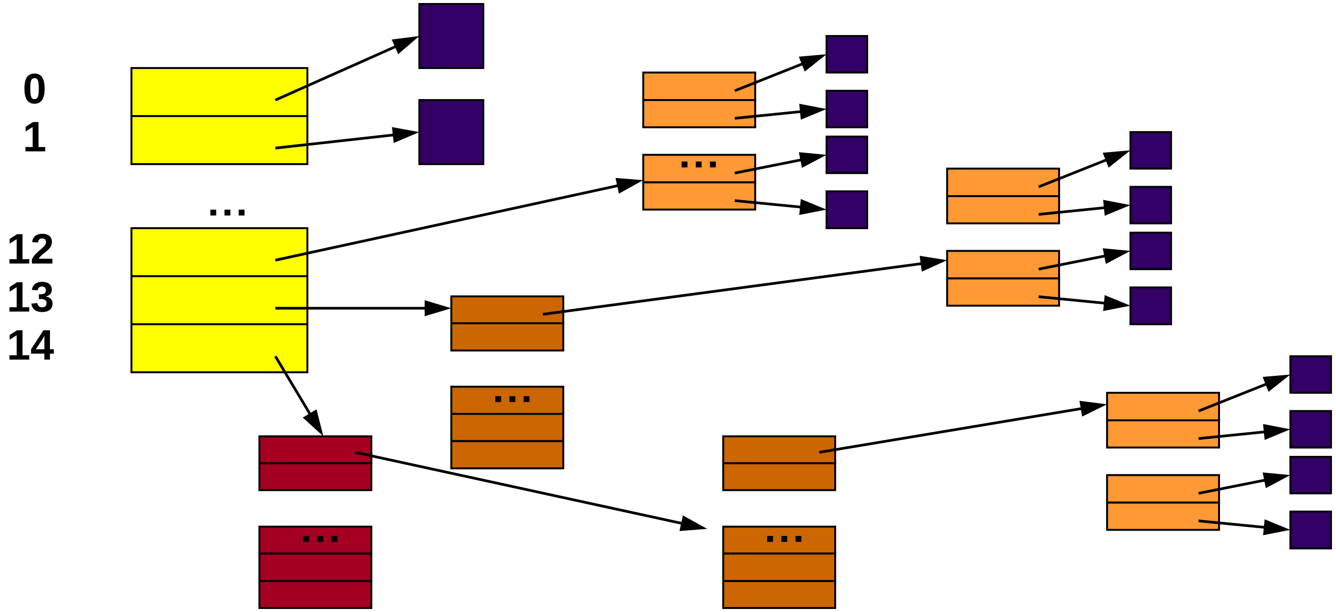


directory

File Name	Start Blk	Last Blk
…	…	…
File B	1	22
…	…	…

Indexed Allocation: Unix Inodes

- Unix inodes implement an indexed structure for files
- Each inode contains 15 block pointers
  - First 12 are direct block pointers (e.g., 4 KB data blocks)
  - Then single, double, and triple indirect



Unix Inodes and Path Search

- Unix Inodes are not directories
- They 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
  - To open “/one”, use Master Block to find inode for “/” on disk and read inode into memory
  - inode allows us to find data block for directory “/”
  - Read “/”, look for entry for “one”
  - This entry gives locates the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality
  - This is 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 4 MB 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
- Several ways to compensate for this
  - “write-behind”
    - Maintain a queue of uncommitted blocks
    - Periodically flush the queue to disk
    - Unreliable
  - Battery backed-up RAM (NVRAM)
    - As with write-behind, but maintain queue in NVRAM
    - Expensive
  - Log-structured file system
    - Always write contiguously at end of previous write

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 on-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)

Summary

- Files
  - Operations, access methods
- Directories
  - Operations, using directories to do path searches
- Sharing
- Protection
  - ACLs vs. capabilities
- File System Layouts
  - Unix inodes
- File Buffer Cache
  - Strategies for handling writes
- Read Ahead

Next time…

- More details on space management, implementations, recovery