Today: File System Functionality

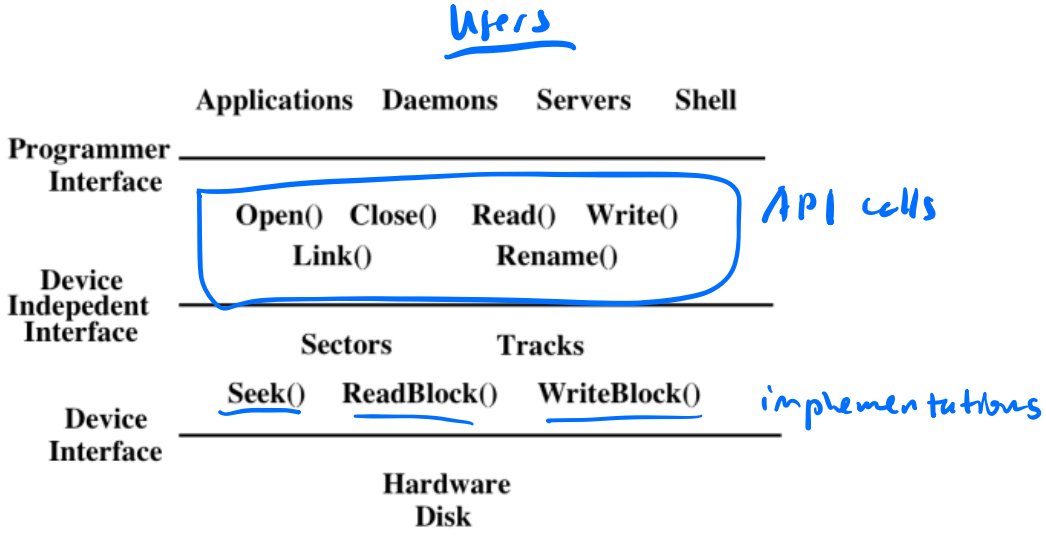
- Remember the high-level view of the OS as a translator from the user abstraction to the hardware reality

virtual

User Abstraction		Hardware Reality
Processes/Threads	<=OS=>	CPU
Address Space	<=OS=> page table	Memory
Files	<=OS=> File system	Disk

a bunch of named bytes → FS → a collection of disk blocks (sectors)

File System Abstraction



User Requirements on Data

- **Persistence:** data stays around between jobs, power cycles, crashes
- **Speed:** can get to data quickly
- **Size:** can store lots of data
- **Sharing/Protection:** users can share data where appropriate or keep it private when appropriate
- **Ease of Use:** user can easily find, examine, modify, etc. data

Hardware/OS Features

- Hardware provides:
  - Persistence: Disks provide non-volatile memory
  - Speed: Speed gained through random access
  - Size: Disks keep getting bigger (typical disk on a PC = 500GB)
- OS provides:
  - Persistence: redundancy allows recovery from some additional failures
  - Sharing/Protection: Unix provides read, write, execute privileges for files
  - Ease of Use
    - Associating names with chunks of data (files)
    - Organize large collections of files into directories
    - Transparent mapping of the user's concept of files and directories onto locations on disks
    - Search facility in file systems (Spotlight in Mac OS X)

Files

a collection of files
a directory structure
Sectors

- File: Logical unit of data on a storage device
  - Formally, named collection of related information recorded on secondary storage
  - Example: reader.cc, a.out
- Files can contain programs (source, binary) or data
- Files can be structured or unstructured
  - Unix implements files as a series of bytes (unstructured)
  - IBM mainframes implements files as a series of records or objects (structured)
- File attributes: name, type, location, size, protection, creation time

File Names

- Three types of names (abstractions)
  - inode (low-level names) index node
  - path (human readable)
  - file descriptor (runtime state)

Inodes

- Each file has exactly one inode number
- Inodes are unique (at a given time) within a FS
- Numbers may be recycled after deletes

- Show inodes via `stat`
  - `$ stat <file or dir>`

Path (multiple directories)

- A directory is a file
  - Associated with an inode
- Contains a list of <user-readable name, low-level name> pairs

e.g., test.c

e.g., 10 (inode)

<"foo", 10>

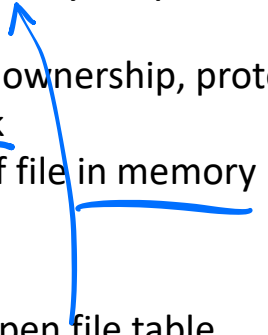
<"bar", 12>

User Interface to the File System

- Common file operations:
  - Data operations:
    - Create(), Delete(), Open(), Close(), Read(), Write(), Seek()
  - Naming operations:
    - HardLink(), SoftLink(), Rename(), SetAttribute(), GetAtrribute()

OS File Data Structures

1. Open file table - shared by all processes with an open file.
   - open count
   - file attributes, including ownership, protection information, access times, ...
   - location(s) of file on disk
   - pointers to location(s) of file in memory
2. Per-process file table
   - for each file,
   - pointer to entry in the open file table
   - current position in file (offset)
   - mode in which the process will access the file (r, w, rw)
   - pointers to file buffer



PCB

a list of opened files

r, w, x

File Operations: Creating a File

- Create(name)
  - Allocate disk space (check disk quotas, permissions, etc.)
  - Create a file descriptor for the file including name, location on disk, and all file attributes.
  - Add the file descriptor to the directory that contains the file.
  - Optional file attribute: file type (Word file, executable, etc.)

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);
```

only written to

truncate to zero

File Operations: Deleting a File

- Delete(name)
  - Find the directory containing the file.
  - Free the disk blocks used by the file.
  - Remove the file descriptor from the directory.
  - Behavior dependent on hard links

File Operations: Open and Close

- fileId = Open(name, mode)
  - Check if the file is already open by another process. If not,
    - Find the file.
    - Copy the file descriptor into the system-wide open file table.
  - Check the protection of the file against the requested mode. If not ok, abort
  - Increment the open count.
  - Create an entry in the process's file table pointing to the entry in the systemwide file table. Initialize the current file pointer to the start of the file.
- Close(fileId)
  - Remove the entry for the file in the process's file table.
  - Decrement the open count in the system-wide file table.
  - If the open count == 0, remove the entry in the system-wide file table.

File Operations: Reading a File

- Read(fileID, from, size, bufAddress)
  - random/direct access – OS reads “size” bytes from file position “from” into “bufAddress”

```
for (i = from; i < from + size; i++)
      bufAddress[i - from] = file[i];
```

- Read(fileID, size, bufAddress) - sequential access
  - OS reads “size” bytes from current file position, fp, into “bufAddress” and increments current file position by size

```
for (i = 0; i < size; i++)
      bufAddress[i] = file[fp + i];
fp += size;
```

File Operations

- **Write** is similar to reads, but copies from the buffer to the file.
- **Seek** just updates fp.
- **Memory mapping** a file
  - Map a part of the portion virtual address space to a file
  - Read/write to that portion of memory \implies OS reads/writes from corresponding location in the file
  - File accesses are greatly simplified (no read/write call are necessary)

File Access Methods

random

sequential

- Common file access patterns from the programmer's perspective
  - Sequential: data processed in order, a byte or record at a time.
    - Most programs use this method
    - Example: compiler reading a source file.
  - Direct: address a block based on a key value.
    - Example: database search, hash table, dictionary
- Common file access patterns from the OS perspective:
  - Sequential: keep a pointer to the next byte in the file. Update the pointer on each read/write.
  - Direct: address any block in the file directly given its offset within the file.

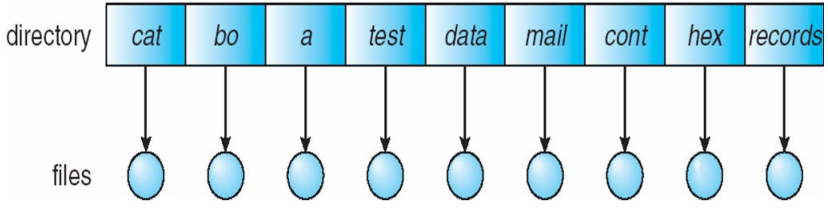
Naming and Directories

- Need a method of getting back to files that are left on disk.
- OS uses numbers for each files
  - Users prefer textual names to refer to files.
  - Directory: OS data structure to map names to file descriptors
- Naming strategies
  - Single-Level Directory: One name space for the entire disk, every name is unique.
    1. Use a special area of disk to hold the directory.
    2. Directory contains pairs.
    3. If one user uses a name, no one else can.
    4. Some early computers used this strategy. Early personal computers also used this strategy because their disks were very small.
  - Two Level Directory: each user has a separate directory, but all of each user's files must still have unique names

Single-Level Directory

O(n)

- A single directory for all users

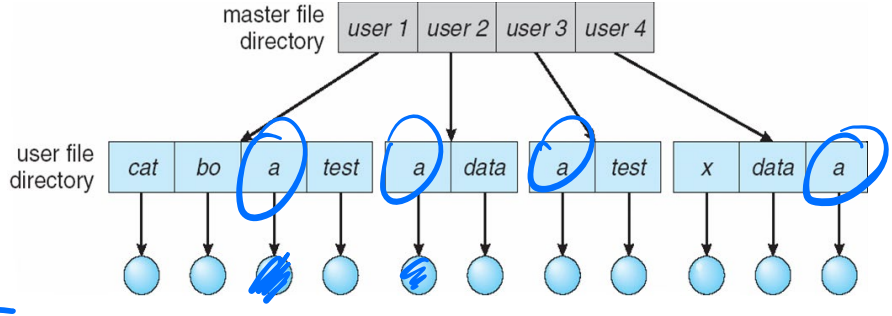


- Naming problem
- Grouping problem

Two-Level Directory

- Separate directory for each user

n-ary search



- Path name
- Can have the same file name for different user
- Efficient searching
- No grouping capability

/user1/a

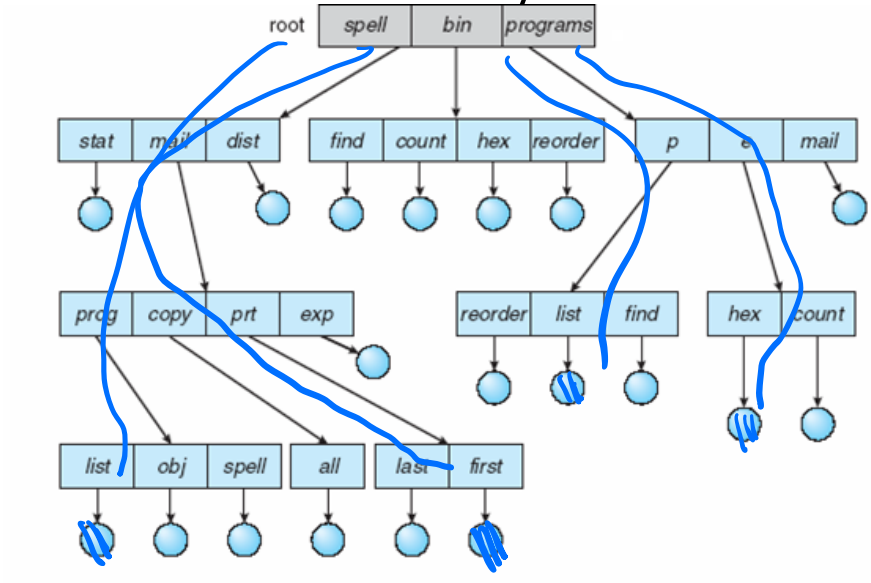
/user2/a

Naming Strategies (cont.)

- Multilevel Directories - tree structured name space (Unix, and all other modern operating systems).
  1. Store directories on disk, just like files except the file descriptor for directories has a special flag bit.
  2. User programs read directories just like any other file, but only special system calls can write directories.
  3. Each directory contains pairs in no particular order. The file referred to by a name may be another directory.
  4. There is one special root directory. Example: How do we look up name: /usr/bin/ls
- Limitations with basic tree structure
  - Difficult to share file across directories and users
  - Can’t have multiple file names

Tree-Structured Directory

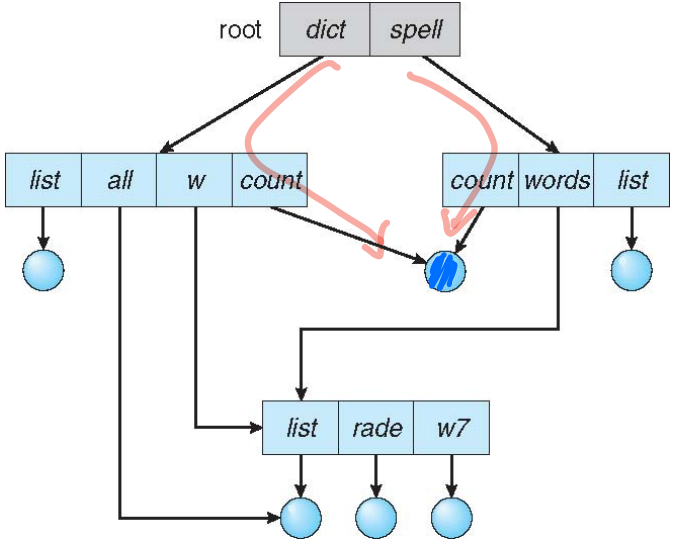
path



Acyclic-Graph Directory

/dict /count

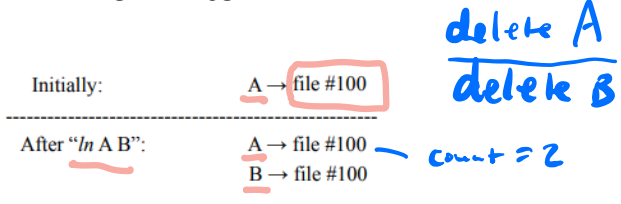
/spell/count



Referential Naming

non-symbolic

- Hard links (Unix: `ln` command)
  - A hard link adds a second connection to a file
  - Example: creating a hard link from B to A



  - OS maintains reference counts, so it will only delete a file after the last link to it has been deleted.

Referential Naming (cont.)

- Soft links (Unix: `ln -s` command)
  - A soft link only makes a symbolic pointer from one file to another.
  - Example: creating a soft link from B to A

Initially:	A → file #100
After "*ln* A B":	A → file #100 B → A

  - removing B does not affect A
  - removing A leaves the name B in the directory, but its contents no longer exists

Directory Operations

- Search for a file: locate an entry for a file
- Create a file: add a directory listing
- Delete a file: remove directory listing
- List a directory: list all files (ls command in UNIX)
- Rename a file
- Traverse the file system

Protection

ls -l

ACL

Capability List

object (e.g., file)

Subject (e.g., user, process, domain ...)

- The OS must allow users to control sharing of their files => control access to files
- Grant or deny access to file operations depending on protection information
- Access lists and groups (Windows NT)
  - Keep an access list for each file with user name and type of access
  - Lists can become large and tedious to maintain
- Access control bits (UNIX)
  - Three categories of users (owner, group, world)
  - Three types of access privileges (read, write, execute)
  - Maintain a bit for each combination (111101000 = rwxr-x---)

r+w+x

chmod 777

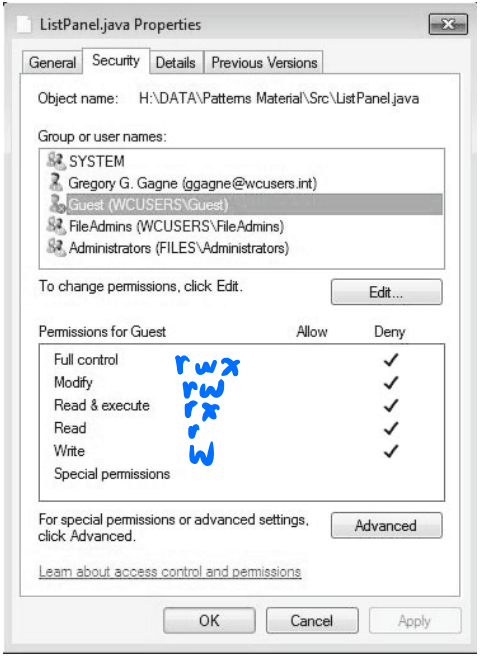
1 x
2 w
4 r

rwx

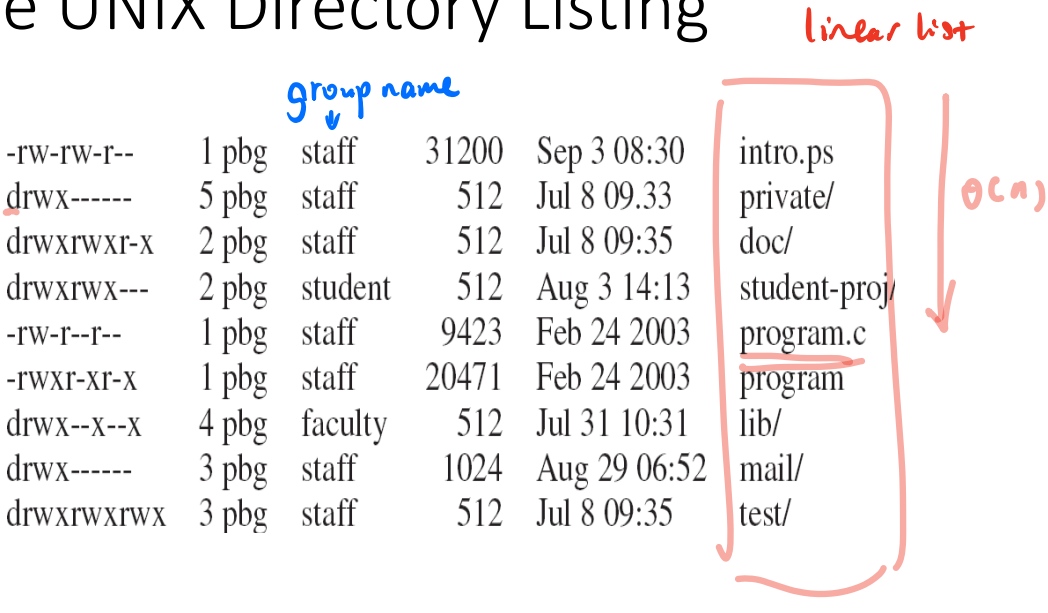
0 or 7

owner group world

Windows 7 Access-Control List Management



A Sample UNIX Directory Listing



-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Today: File System Implementation

- Disk management
  - Brief review of how disks work (already discussed two lectures ago)
  - How to organize data on disks

File Organization on Disk

- The information we need:

  fileID 0, Block 0 → Platter 0, cylinder 0, sector 0

  fileID 0, Block 1 → Platter 4, cylinder 3, sector 8

  …

- Key performance issues:

1. We need to support sequential and random access
2. What is the right data structure in which to maintain file location information?
3. How do we lay out the files on the physical disk?

File Organization: On-Disk Data Structures

- The structure used to describe where the file is on the disk and the attributes of the file is the *file descriptor* (*FileDesc*). File descriptors have to be stored on disks just like files

- Most systems fit the following profile:

1. Most files are small
2. Most disk space is taken up by large files
3. I/O operations target both small and large files

a few

==> The per-file cost must be low, but large files must also have good performance

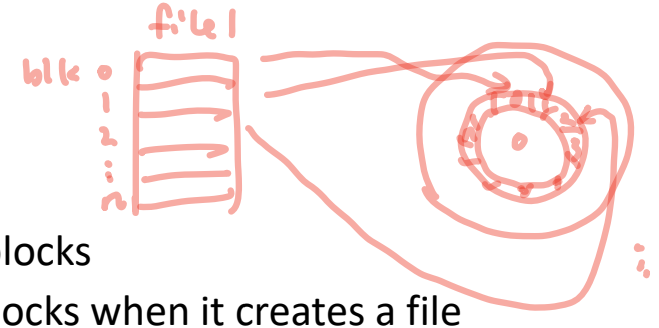
Directory Implementation

<high-level name, low-level name>

platter 0 cylinder 0
foo → sector 1
bar → ...

- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
    - Linear search time O(n)
    - Could keep ordered alphabetically via linked list or use B+ tree O(log n)
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method

Contiguous Allocation



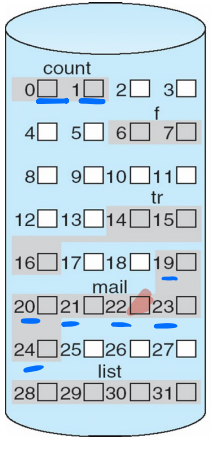
- OS maintains an ordered list of free disk blocks
- OS allocates a contiguous chunk of free blocks when it creates a file
- Need to store only the start location and size in the file descriptor
- Advantages
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Access time? Number of seeks? (sequential and random access)
- Disadvantages
  - Changing file sizes
  - Fragmentation? Disk management?
    - External fragmentation, need for compaction
- Examples: IBM OS/360, write-once disks, early personal computers

base + offset

segmentation

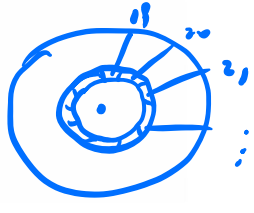
disk defragmentation

Contiguous Allocation (cont.)



directory

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

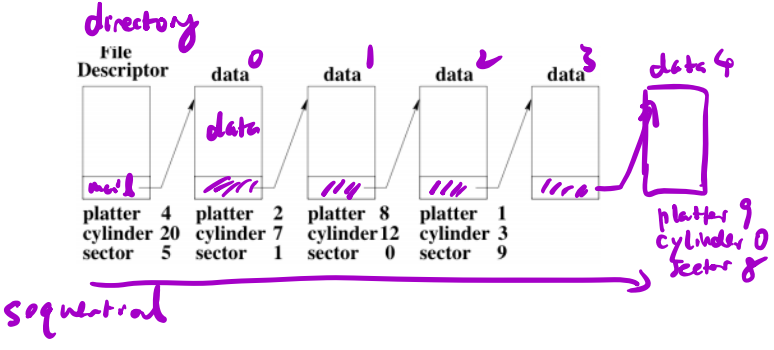


Extent-Based Systems

- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme
- Extent-based file systems allocate disk blocks in extents
- An extent is a contiguous block of disks
  - Extents are allocated for file allocation
  - A file consists of one or more extents

Linked Files

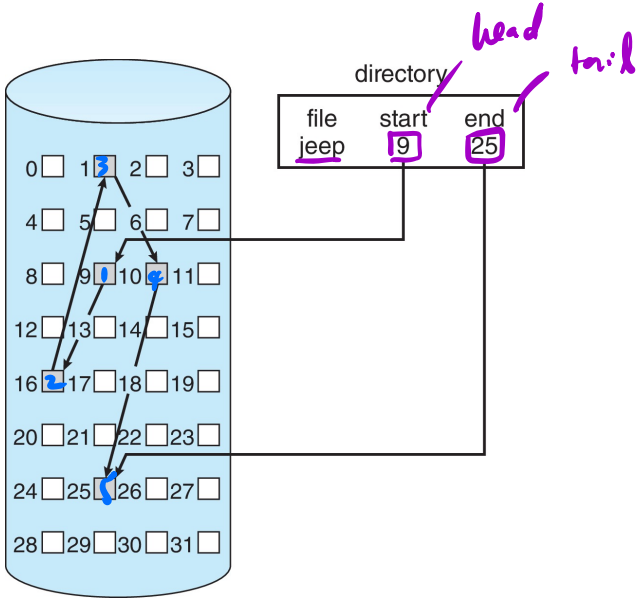
- Keep a list of all the free sectors/blocks
- In the file descriptor, keep a pointer to the first sector/block
- In each sector, keep a pointer to the next sector



Linked Files (cont.)

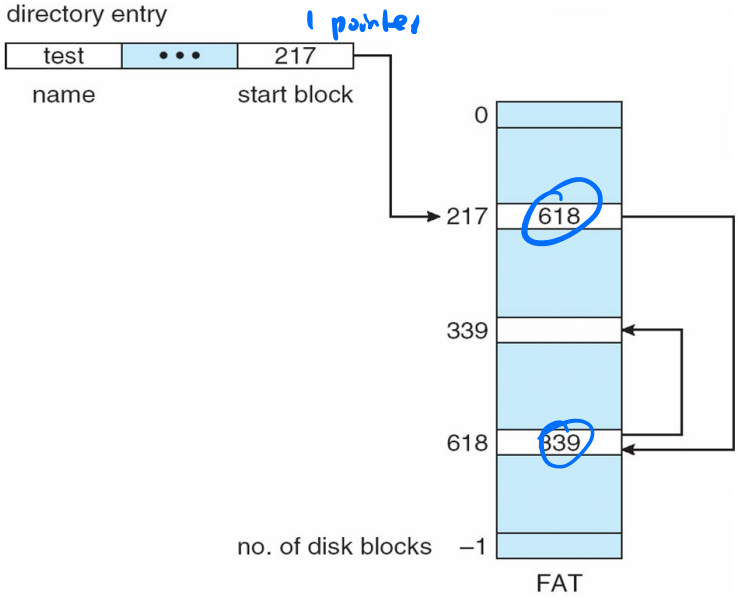
- Advantages:
  - Fragmentation? no external fragmentation
  - File size changes? ✓
  - Efficiently supports which type of access? sequential
- Disadvantages:
  - Does not support which type of access? Why? random
  - Number of seeks? non-contiguous
- Examples: FAT, MS-DOS

Linked Allocation



File-Allocation Table

FAT



flash drive

512 B

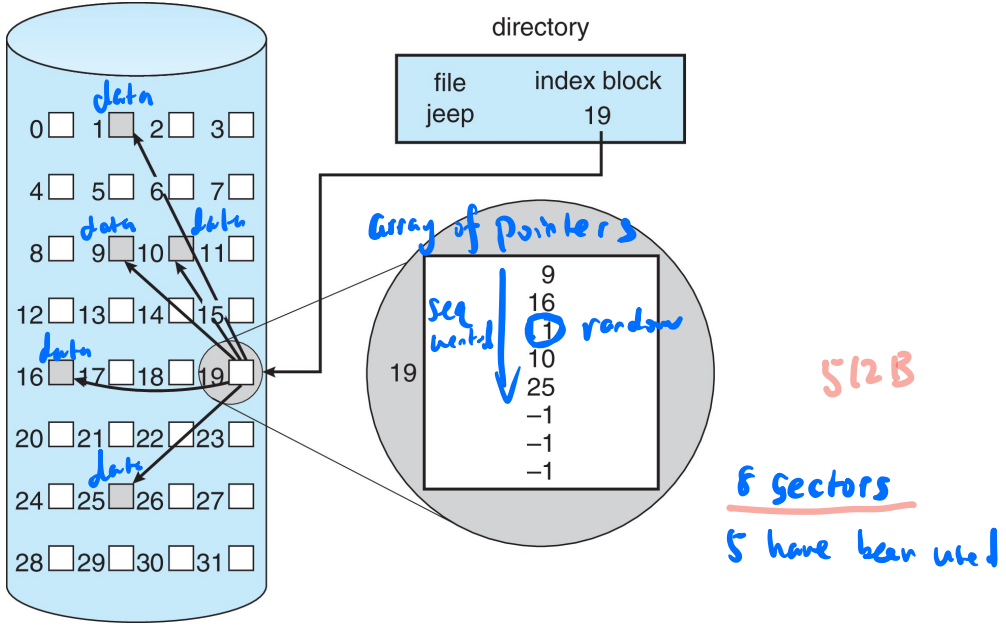
508 B data

4 B pointer

Indexed Files

- OS keeps an array of block pointers for each file
- The user or OS must declare the maximum length of the file when it is created
- OS allocates an array to hold the pointers to all the blocks when it creates the file, but allocates the blocks only on demand
- OS fills in the pointers as it allocates blocks

Example of Indexed Allocation

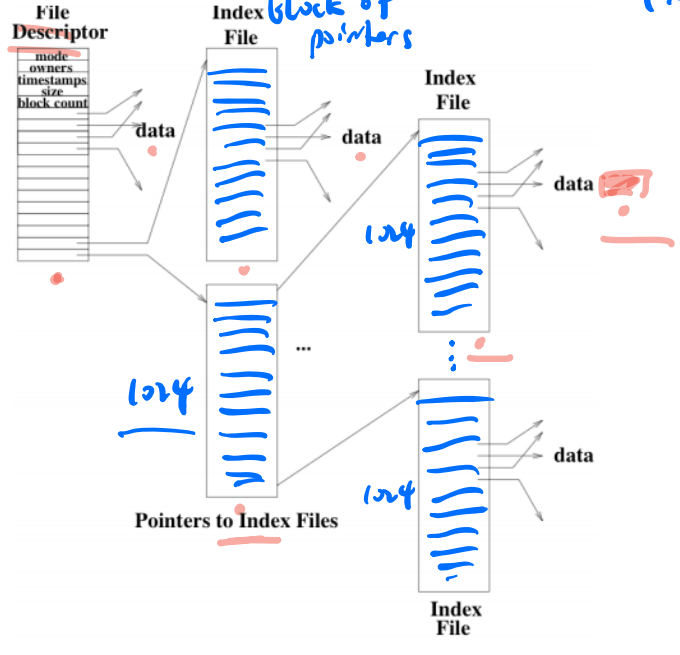


Indexed Files (cont.)

- Advantages:
  - Not much wasted space for files
  - Both sequential and random accesses are easy
- Disadvantages:
  - Wasted space in file descriptors
  - Sets a maximum file size
  - Lots of seeks because data is not contiguous

Multilevel Indexed Files

- Each file descriptor contains (for example) 14 block pointers
- First 12 pointers point to data blocks
- 13th pointer points to a block of 1024 pointers to 1024 more data blocks (One indirection)
- 14th pointer points to a block of pointers to indirect blocks (Two indirections)

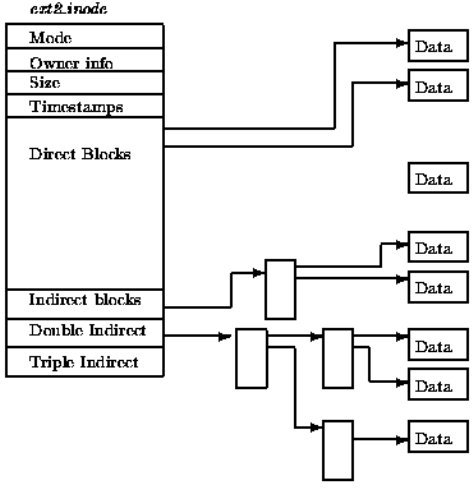


Multilevel Indexed Files

- Advantages
  - Supports incremental file growth
  - Small files?
- Disadvantages
  - Indirect access is inefficient for random access to very large files
  - Lots of seeks because data is not contiguous
- What could the OS do to get more contiguous access and fewer seeks?
- Examples: BSD Unix 4.3

The EXT2 inode

- The inode is the basic building block; every file and directory is described by one and only one inode
- Contains the following fields:
  - Mode
  - Owner info
  - Size
  - Timestamps
  - Datablocks
    - The first twelve are pointers to the physical blocks and the last three pointers contain more levels of indirections



Performance

- Best method depends on file access type
  - Contiguous great for sequential and random
- Linked good for sequential, not random
- Declare access type at creation -> select either contiguous or linked
- Indexed more complex
  - Single block access could require 2 index block reads then data block read
  - Clustering can help improve throughput, reduce CPU overhead

Free-Space Management

- Need a free-space list to keep track of which disk blocks are free (just as we need a free-space list for main memory)
- Need to be able to find free space quickly and release space quickly => use a bitmap
  - The bitmap has one bit for each block on the disk.
  - If the bit is 1, the block is free. If the bit is 0, the block is allocated.
- Can quickly determine if any page in the next 32 is free, by comparing the word to 0. If it is 0, all the pages are in use. Otherwise, you can use bit operations to find an empty block.

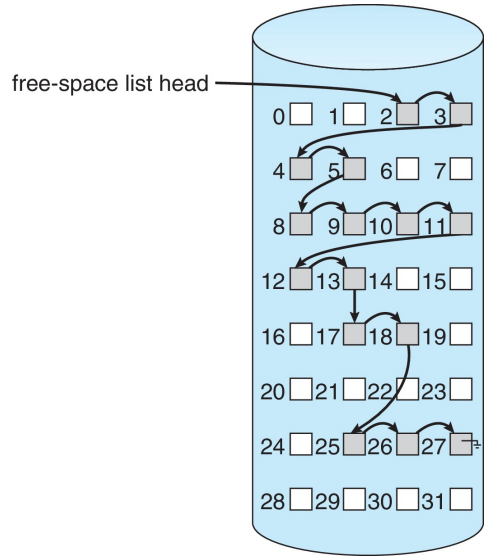
  11000010010001111111 0…
- Marking a block as freed is simple since the block number can be used to index into the bitmap to set a single bit.

Free-Space Management (cont.)

- Problem: Bitmap might be too big to keep in memory for a large disk. A 2 TB disk with 512 byte sectors requires a bitmap with 4,000,000,000 entries (500,000,000 bytes = 500 MB).
- If most of the disk is in use, it will be expensive to find free blocks with a bitmap.

Linked Free Space List on Disk

- An alternative implementation is to link together the free blocks.
  - The head of the list is cached in kernel memory. Each block contains a pointer to the next free block.
  - How expensive is it to allocate a block?
  - How expensive is it to free a block?
  - How expensive is it to allocate consecutive blocks?



Summary

- Many of the concerns and implementations of file system implementations are similar to those of virtual memory implementations.
  - Contiguous allocation is simple, but suffers from external fragmentation, the need for compaction, and the need to move files as they grow.
  - Indexed allocation is very similar to page tables. A table maps from logical file blocks to physical disk blocks.
  - Free space can be managed using a bitmap or a linked list.

Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn't understand