

OPERATING SYSTEMS

FILE SYSTEMS

Jerry Breecher

FILE SYSTEMS

This material covers Silberschatz Chapters 10 and 11.

File System Interface

The user level (more visible) portion of the file system.

- Access methods
- Directory Structure
- Protection

File System Implementation

The OS level (less visible) portion of the file system.

- Allocation and Free Space Management
- Directory Implementation

FILE SYSTEMS INTERFACE

File Concept

- A collection of related bytes having meaning only to the creator. The file can be "free formed", indexed, structured, etc.
- The file is an entry in a directory.
- The file may have attributes (name, creator, date, type, permissions)
- The file may have structure (O.S. may or may not know about this.) It's a tradeoff of power versus overhead. For example,
 - a) An Operating System understands program image format in order to create a process.
 - b) The UNIX **shell** understands how directory files look. (In general the UNIX **kernel** doesn't interpret files.)
 - c) Usually the Operating System understands and interprets file types.

FILE SYSTEMS INTERFACE

File Concept

A file can have various kinds of structure

- None - sequence of words, bytes
- Simple record structure
 - Lines
 - Fixed length
 - Variable length
- Complex Structures
 - Formatted document
 - Relocatable load file
- Who interprets this structure?
 - Operating system
 - Program

FILE SYSTEMS INTERFACE

File Concept

Attributes of a File

- **Name** – only information kept in human-readable form
- **Identifier** – unique tag (number) identifies file within file system
- **Type** – needed for systems that support different types
- **Location** – pointer to file location on device
- **Size** – current file size
- **Protection** – controls who can do reading, writing, executing
- **Time, date, and user identification** – data for protection, security, and usage monitoring

- Information about files is kept in the directory structure, which is maintained on the disk.

FILE SYSTEMS INTERFACE

File Concept

What can we find out about a Linux File?

```
jbreecher@younger:~$ stat A_File
  File: `A_File'
  Size: 6491          Blocks: 16          IO Block: 4096   regular file
Device: 14h/20d Inode: 20938754    Links: 1
Access: (0600/-rw-----)  Uid: ( 1170/jbreecher)   Gid: ( 100/   users)
Access: 2006-11-15 15:38:17.000000000 -0500
Modify: 2006-09-27 17:44:10.000000000 -0400
Change: 2006-09-27 17:44:10.000000000 -0400
```

```
jbreecher@younger:~/public/os/Code$ stat protos.h
  File: `protos.h'
  Size: 2889          Blocks: 8          IO Block: 4096   regular file
Device: 14h/20d Inode: 28442631    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1170/jbreecher)   Gid: ( 100/   users)
Access: 2006-11-16 03:56:17.000000000 -0500
Modify: 2006-08-27 12:45:57.000000000 -0400
Change: 2006-08-27 13:25:24.000000000 -0400
```

FILE SYSTEMS INTERFACE

File Concept

Note: The command "LDE" - Linux Disk Editor - does amazing things but requires root privilege.

```
-rw-rw-rw- 1 jbrecherusers 56243 Mon Dec 18 14:25:40 2006
```

```
TYPE: regular file LINKS: 1 DIRECT BLOCKS= 0x002462CA
MODE: \0666 FLAGS: \10 0x002462CB
UID: 01170(jbrecher)ID: 00100(users) 0x002462CC
SIZE: 56243 SIZE(BLKS): 128 0x002462CD
0x002462CE
0x002462CF
0x002462D0
0x002462D1
0x002462D2
0x002462D3
0x002462D4
0x002462D5
0x002462D6
INDIRECT BLOCK=
2x INDIRECT BLOCK=
3x INDIRECT BLOCK=
```

Expanded on next page

FILE SYSTEMS INTERFACE

File Concept

```
lde v2.6.1 : ext2 : /dev/mapper/VolGroup00-LogVol01
Inode:      1170636 (0x0011DCCC)  Block:      2384586 (0x002462CA)  0123456789!@$%^
462CA000    74 68 69 73 20 6D 61 6E : 79 20 6E 6F 74 20 77 6F  this many not wo
462CA010    72 6B 20 74 68 69 73 20 : 6D 61 6E 79 20 6E 6F 74  rk this many not
462CA020    20 77 6F 72 6B 20 74 68 : 69 73 20 6D 61 6E 79 20  work this many
462CA030    6E 6F 74 20 77 6F 72 6B : 20 74 68 69 73 20 6D 61  not work this ma
462CA040    6E 79 20 6E 6F 74 20 77 : 6F 72 6B 20 74 68 69 73  ny not work this
462CA050    20 6D 61 6E 79 20 6E 6F : 74 20 77 6F 72 6B 0A 74  many not work.t
462CA060    68 69 73 20 6D 61 6E 79 : 20 6E 6F 74 20 77 6F 72  his many not wor
```

```
lde v2.6.1 : ext2 : /dev/mapper/VolGroup00-LogVol01
Inode:      1170636 (0x0011DCCC)  Block:      2384598 (0x002462D6)  0123456789!@$%^
462D6000    D7 62 24 00 D8 62 24 00 : 00 00 00 00 00 00 00 00  .b$..b$.....
462D6010    00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  .....
462D6020    00 00 00 00 00 00 00 00 : 00 00 00 00 00 00 00 00  .....
```


FILE SYSTEMS INTERFACE

File Concept

Blocking (packing) occurs when some entity, (either the user or the Operating System) must pack bytes into a physical block.

- a) Block size is fixed for disks, variable for tape
- b) Size determines maximum internal fragmentation
- c) We can allow reference to a file as a set of logical records (addressable units) and then divide (or pack) logical records into physical blocks.

What does it mean to “open” a file??

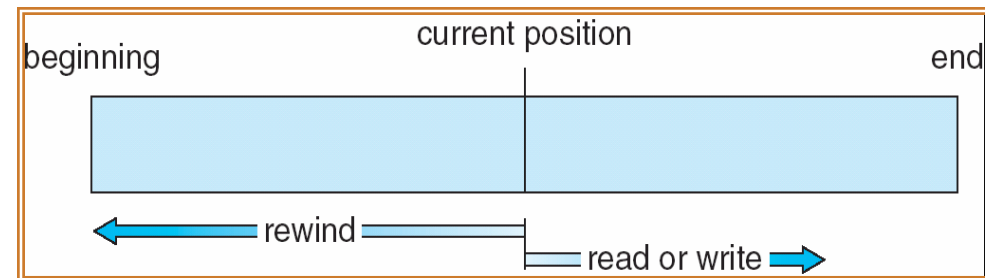
FILE SYSTEMS INTERFACE

Access Methods

If files had only one "chunk" of data, life would be simple. But for large files, the files themselves may contain structure, making access faster.

SEQUENTIAL ACCESS

- Implemented by the filesystem.
- Data is accessed one record right after the last.
- Reads cause a pointer to be moved ahead by one.
- Writes allocate space for the record and move the pointer to the new End Of File.
- Such a method is reasonable for tape



FILE SYSTEMS INTERFACE

Access Methods

DIRECT ACCESS

- Method useful for disks.
- The file is viewed as a numbered sequence of blocks or records.
- There are no restrictions on which blocks are read/written in any order.
- User now says "read n" rather than "read next".
- "n" is a number relative to the beginning of file, not relative to an absolute physical disk location.

FILE SYSTEMS INTERFACE

Access Methods

OTHER ACCESS METHODS

Built on top of direct access and often implemented by a user utility.

Indexed ID plus pointer.

An index block says what's in each remaining block or contains pointers to blocks containing particular items. Suppose a file contains many blocks of data arranged by name alphabetically.

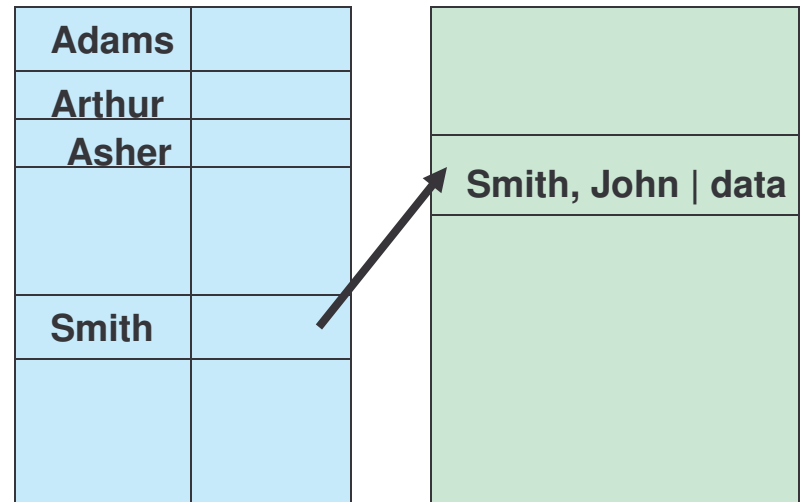
Example 1: Index contains the name appearing as the first record in each block. There are as many index entries as there are blocks.

Example 2: Index contains the block number where "A" begins, where "B" begins, etc. Here there are only 26 index entries.

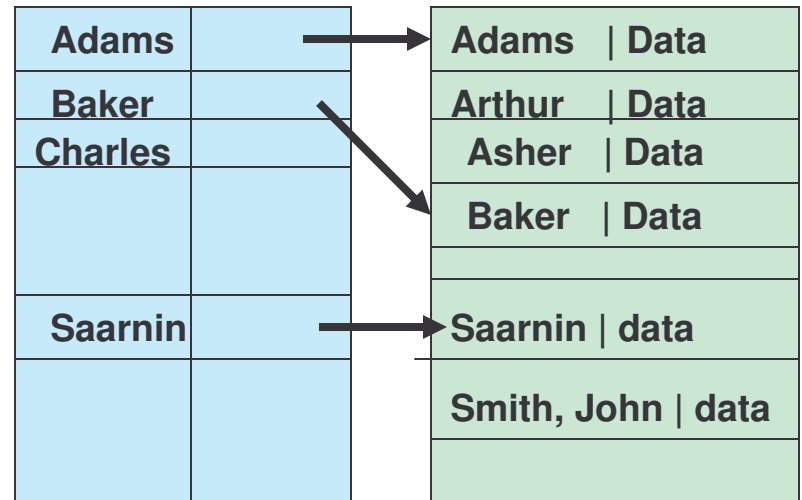
FILE SYSTEMS INTERFACE

Access Methods

Example 1: Index contains the name appearing as the first record in each block. There are as many index entries as there are blocks.



Example 2: Index contains the block number where "A" begins, where "B" begins, etc. Here there are only 26 index entries.



FILE SYSTEMS INTERFACE

Directory Structure

Directories maintain information about files:

For a large number of files, may want a directory structure - directories under directories.

Information maintained in a directory:

Name	The user visible name.
Type	The file is a directory, a program image, a user file, a link, etc.
Location	Device and location on the device where the file header is located.
Size	Number of bytes/words/blocks in the file.
Position	Current next-read/next-write pointers. ← In Memory only!
Protection	Access control on read/write/ execute/delete.
Usage	Open count ← In Memory only!
Usage	time of creation/access, etc.
Mounting	a filesystem occurs when the root of one filesystem is "grafted" into the existing tree of another filesystem.

There is a need to **PROTECT** files and directories.

Actions that might be protected include: **read, write, execute, append, delete, list**

FILE SYSTEMS INTERFACE

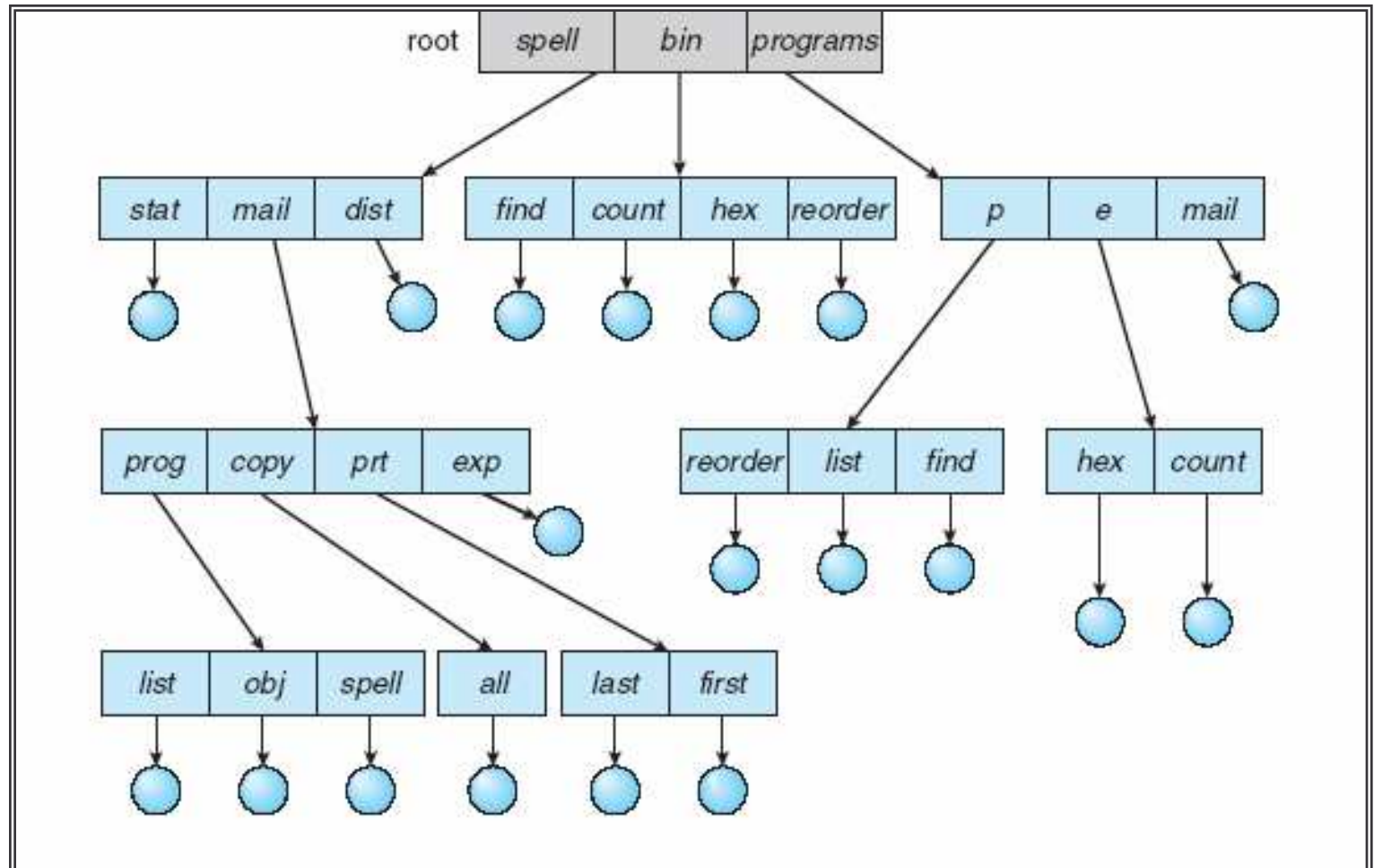
Directory Structure

```
jbreecher@younger:~/public/os$ stat Code
  File: `Code'
  Size: 4096          Blocks: 8          IO Block: 4096   directory
Device: 14h/20d Inode: 28606492    Links: 2
Access: (0755/drwxr-xr-x)  Uid: ( 1170/jbreecher)  Gid: ( 100/  users)
Access: 2006-11-16 14:52:11.000000000 -0500
Modify: 2006-11-16 14:52:01.000000000 -0500
Change: 2006-11-16 14:52:01.000000000 -0500
```

FILE SYSTEMS INTERFACE

Directory Structure

Tree-Structured Directory



FILE SYSTEMS INTERFACE **Other Issues**

Mounting:

- **Attaching portions of the file system into a directory structure.**

Sharing:

- Sharing must be done through a **protection** scheme
- May use networking to allow file system access between systems
 - Manually via programs like FTP or SSH
 - Automatically, seamlessly using **distributed file systems**
 - Semi automatically via the **world wide web**
- **Client-server** model allows clients to mount remote file systems from servers
 - Server can serve multiple clients
 - Client and user-on-client identification is insecure or complicated
 - **NFS** is standard UNIX client-server file sharing protocol
 - **CIFS** is standard Windows protocol
 - Standard operating system file calls are translated into remote calls

FILE SYSTEMS INTERFACE

Protection

■ File owner/creator should be able to control:

- what can be done
- by whom

■ Types of access

- Read
- Write
- Execute
- Append
- Delete
- List

- Mode of access: read, write, execute
- Three classes of users

a) **owner access 7** ⇒

RWX

1 1 1

RWX

b) **group access 6** ⇒

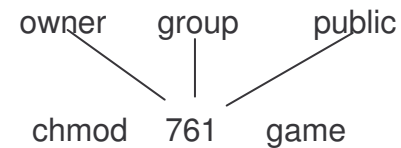
1 1 0

RWX

c) **public access 1** ⇒

0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



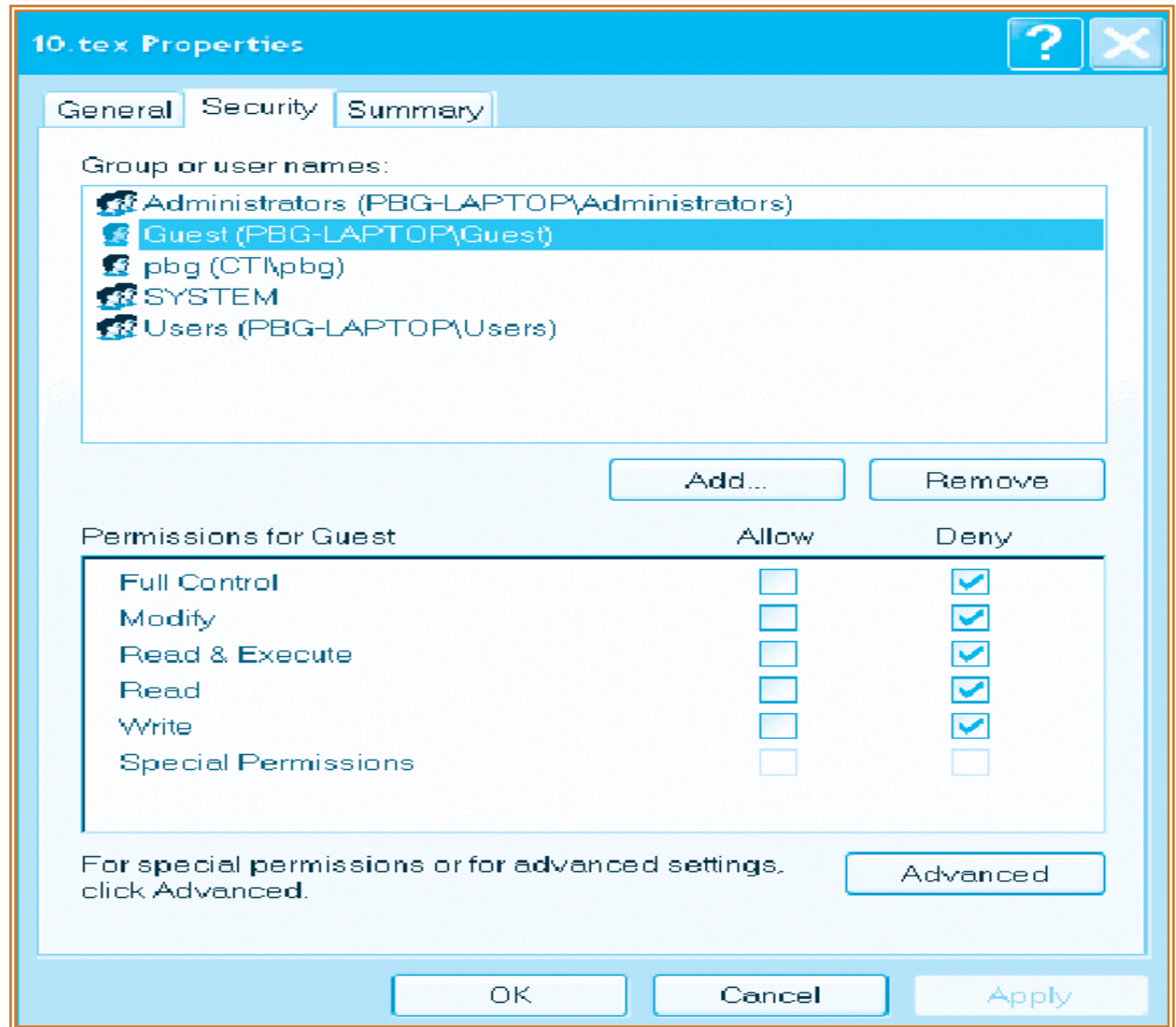
Attach a group to a file

“chgrp G game”

FILE SYSTEMS INTERFACE

Protection

Example on
Windows XP



FILE SYSTEM IMPLEMENTATION

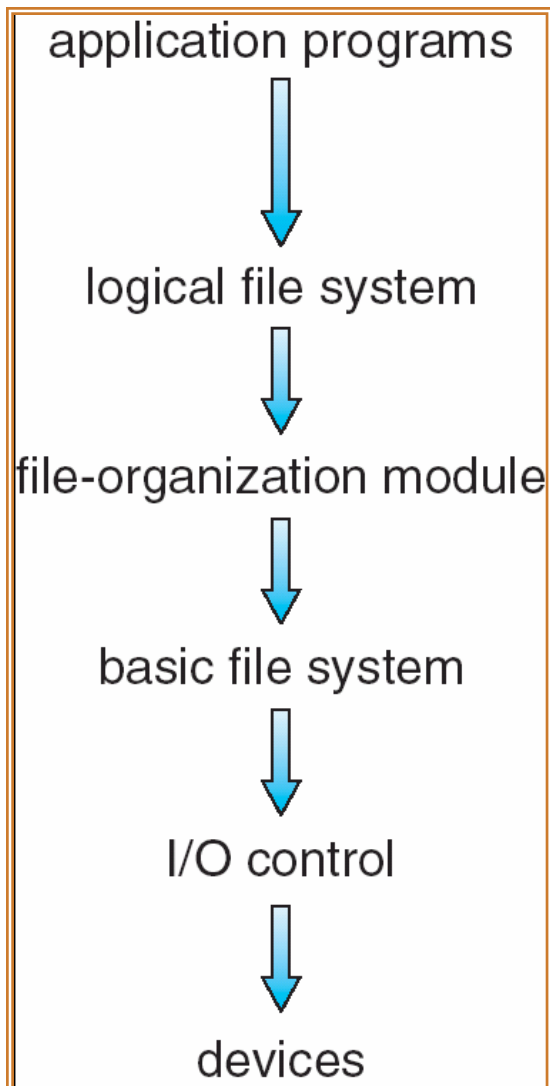
FILE SYSTEM STRUCTURE:

When talking about “the file system”, you are making a statement about both the rules used for file access, and about the algorithms used to implement those rules. Here’s a breakdown of those algorithmic pieces.

Application Programs	The code that's making a file request.
Logical File System	This is the highest level in the OS; it does protection, and security. Uses the directory structure to do name resolution.
File-organization Module	Here we read the file control block maintained in the directory so we know about files and the logical blocks where information about that file is located.
Basic File System	Knowing specific blocks to access, we can now make generic requests to the appropriate device driver.
IO Control	These are device drivers and interrupt handlers. They cause the device to transfer information between that device and CPU memory.
Devices	The disks / tapes / etc.

FILE SYSTEM IMPLEMENTATION

Layered File System



Handles the CONTENT of the file. Knows the file's internal structure.

Handles the OPEN, etc. system calls. Understands paths, directory structure, etc.

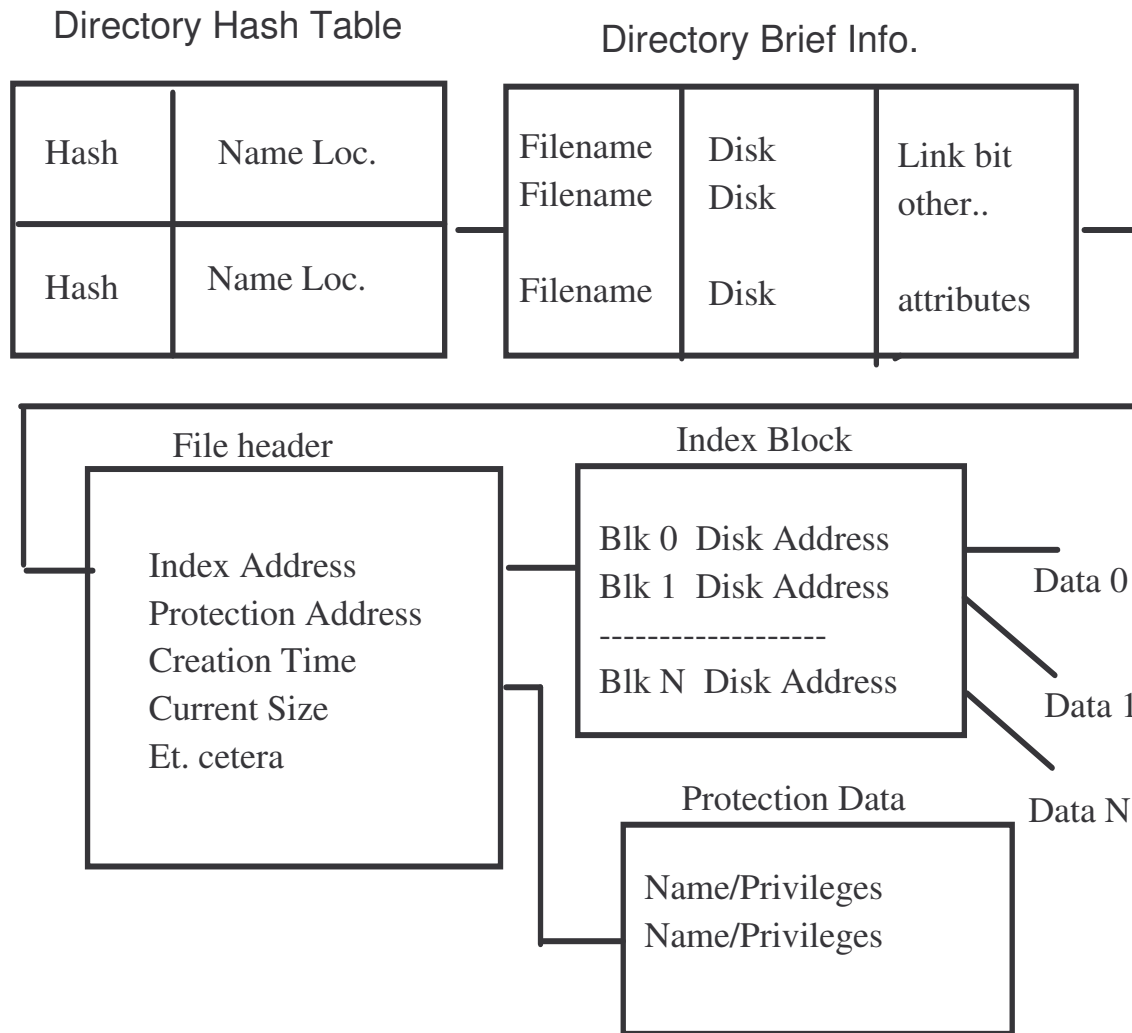
Uses directory information to figure out blocks, etc. Implements the READ. POSITION calls.

Determines where on the disk blocks are located.

Interfaces with the devices – handles interrupts.

FILE SYSTEM IMPLEMENTATION

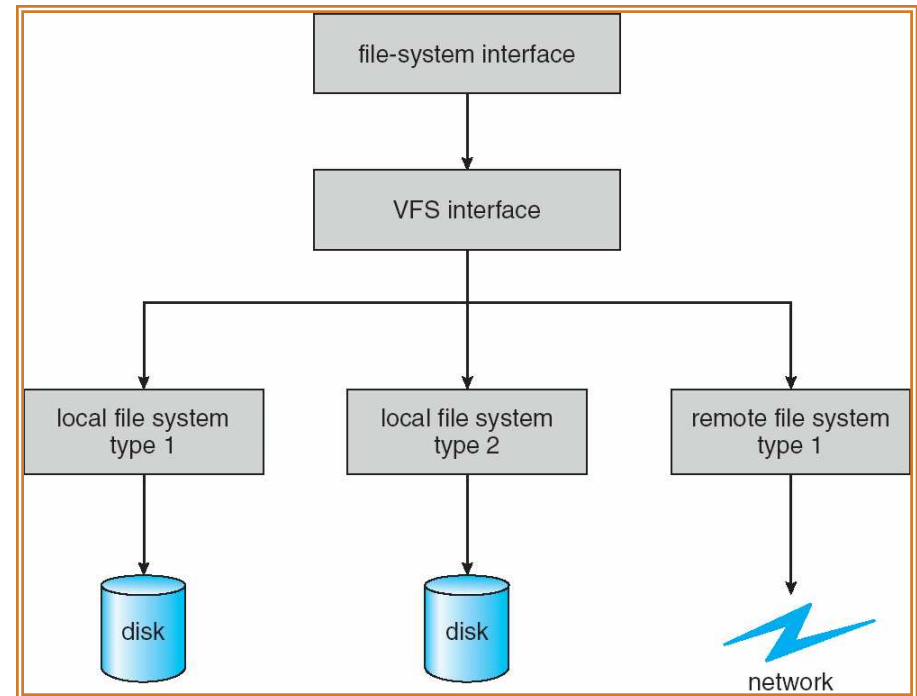
Example of Directory and File Structure



FILE SYSTEM IMPLEMENTATION

Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.



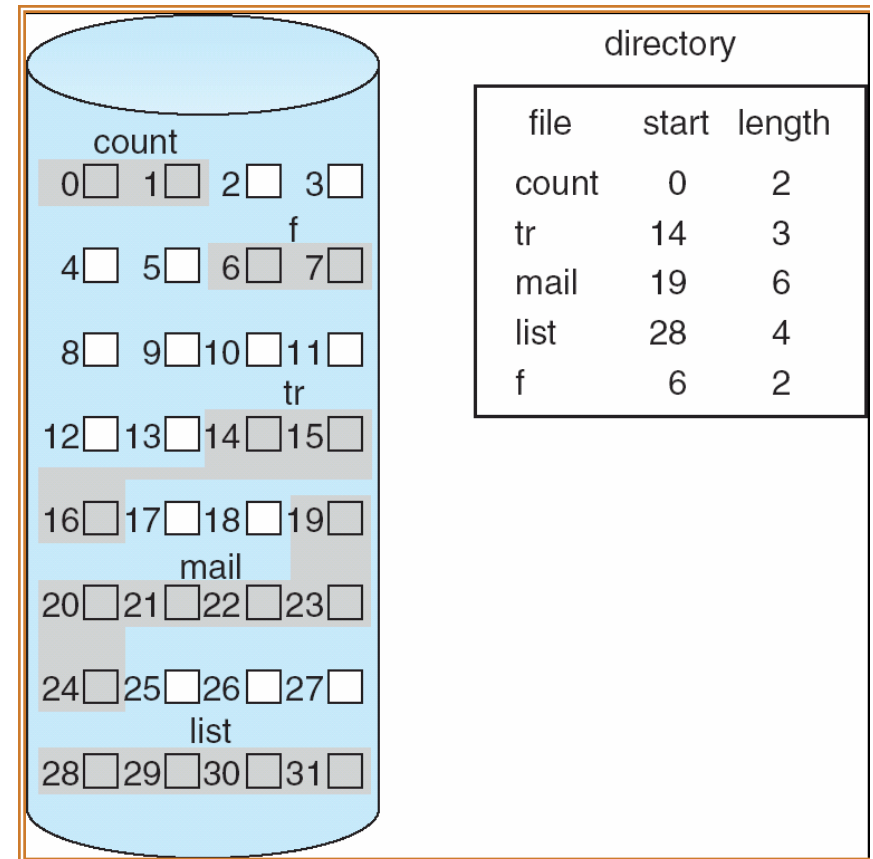
FILE SYSTEM IMPLEMENTATION

CONTIGUOUS ALLOCATION

Allocation Methods

- Method: Lay down the entire file on contiguous sectors of the disk. Define by a dyad $\langle \text{first block location, length} \rangle$.

- Accessing the file requires a minimum of head movement.
- Easy to calculate block location: block i of a file, starting at disk address b , is $b + i$.
- Difficulty is in finding the contiguous space, especially for a large file. Problem is one of dynamic allocation (first fit, best fit, etc.) which has external fragmentation. If many files are created/deleted, compaction will be necessary.



- It's hard to estimate at create time what the size of the file will ultimately be. What happens when we want to extend the file --- we must either terminate the owner of the file, or try to find a bigger hole.

FILE SYSTEM IMPLEMENTATION

Allocation Methods

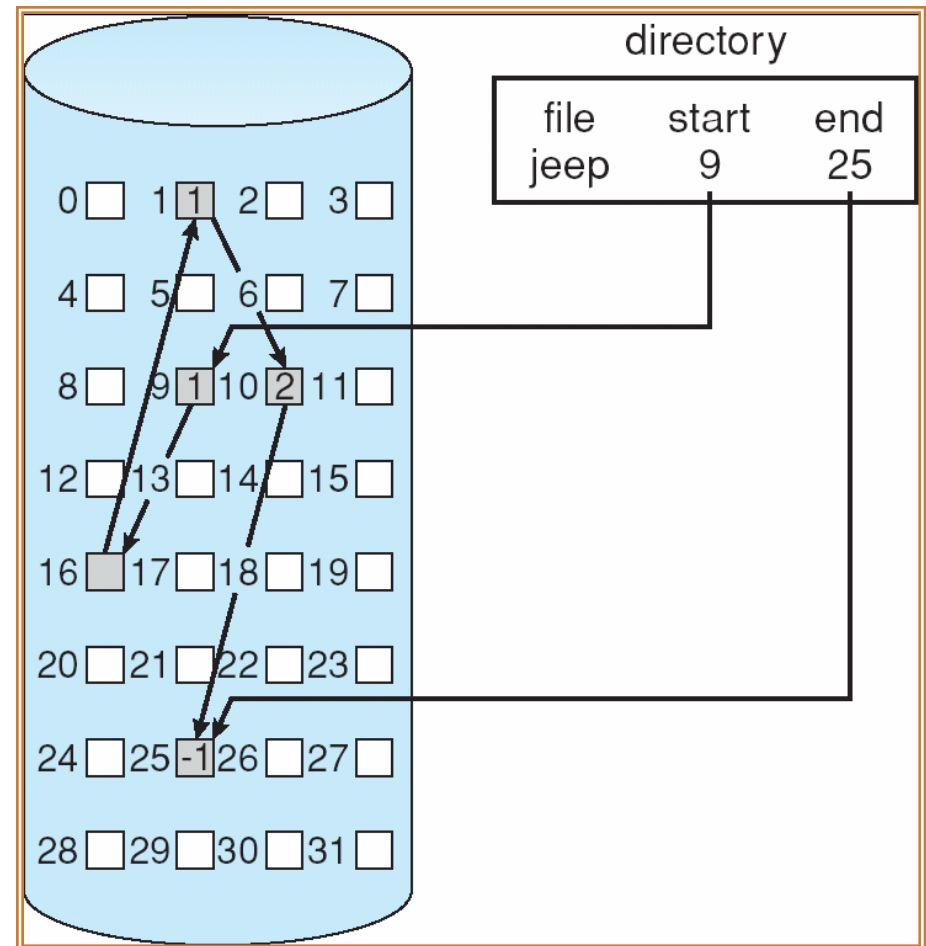
LINKED ALLOCATION

Each file is a linked list of disk blocks, scattered anywhere on the disk.

At file creation time, simply tell the directory about the file. When writing, get a free block and write to it, enqueueing it to the file header.

There's no external fragmentation since each request is for one block.

Method can only be effectively used for sequential files.



FILE SYSTEM IMPLEMENTATION

Allocation Methods

LINKED ALLOCATION

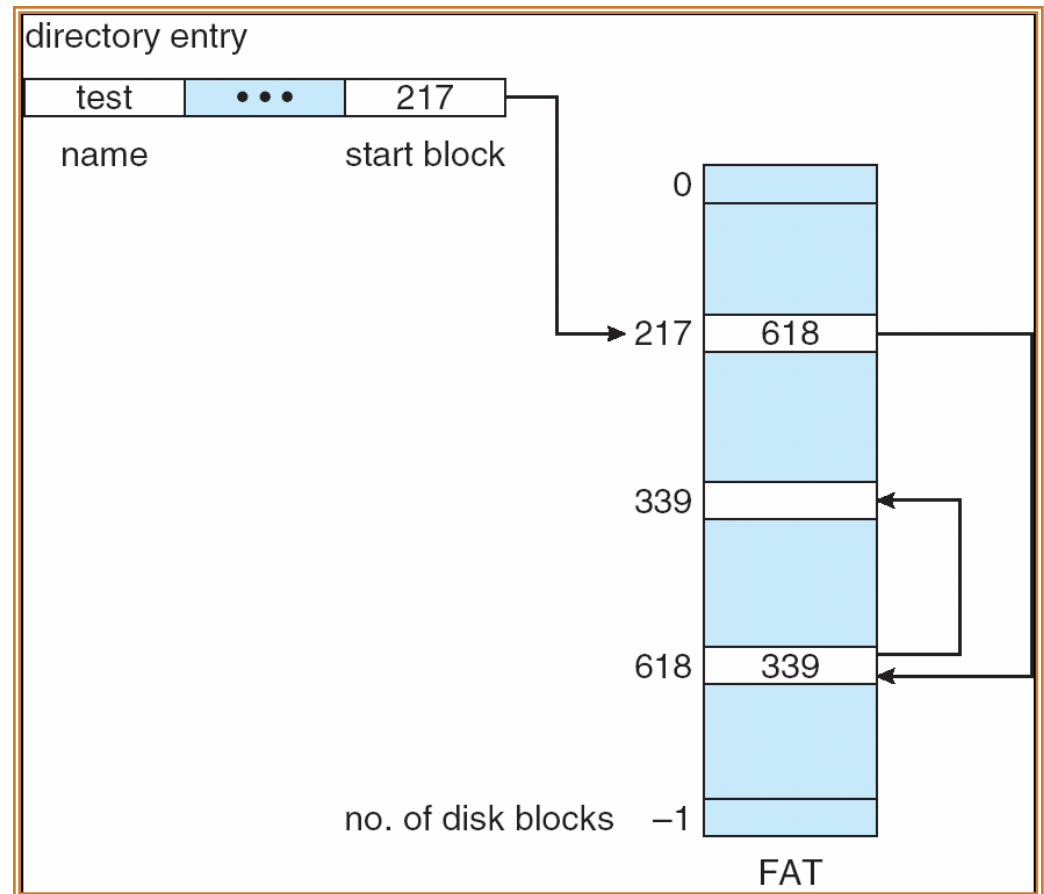
Pointers use up space in each block. Reliability is not high because any loss of a pointer loses the rest of the file.

A File Allocation Table is a variation of this.

It uses a separate disk area to hold the links.

This method doesn't use space in data blocks. Many pointers may remain in memory.

A FAT file system is used by MS-DOS.

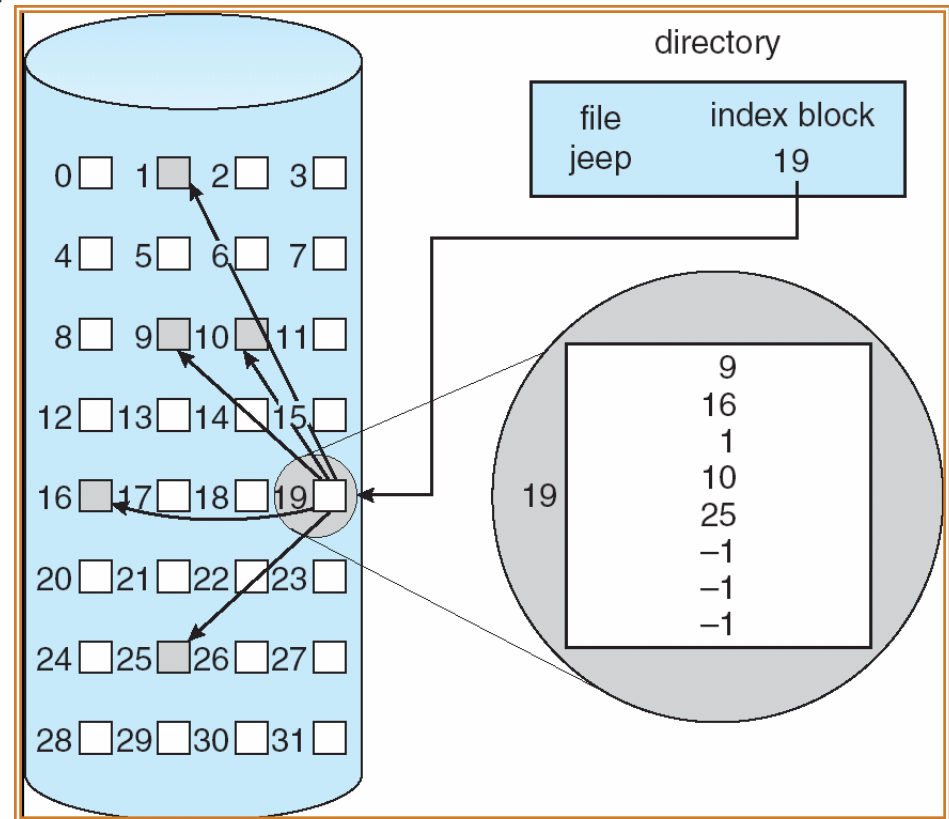


FILE SYSTEM IMPLEMENTATION

INDEXED ALLOCATION

- Each file uses an index block on disk to contain addresses of other disk blocks used by the file.
- When the i th block is written, the address of a free block is placed at the i th position in the index block.
- Method suffers from wasted space since, for small files, most of the index block is wasted. What is the optimum size of an index block?
- If the index block is too small, we can:
 - a) Link several together
 - b) Use a multilevel index

Allocation Methods



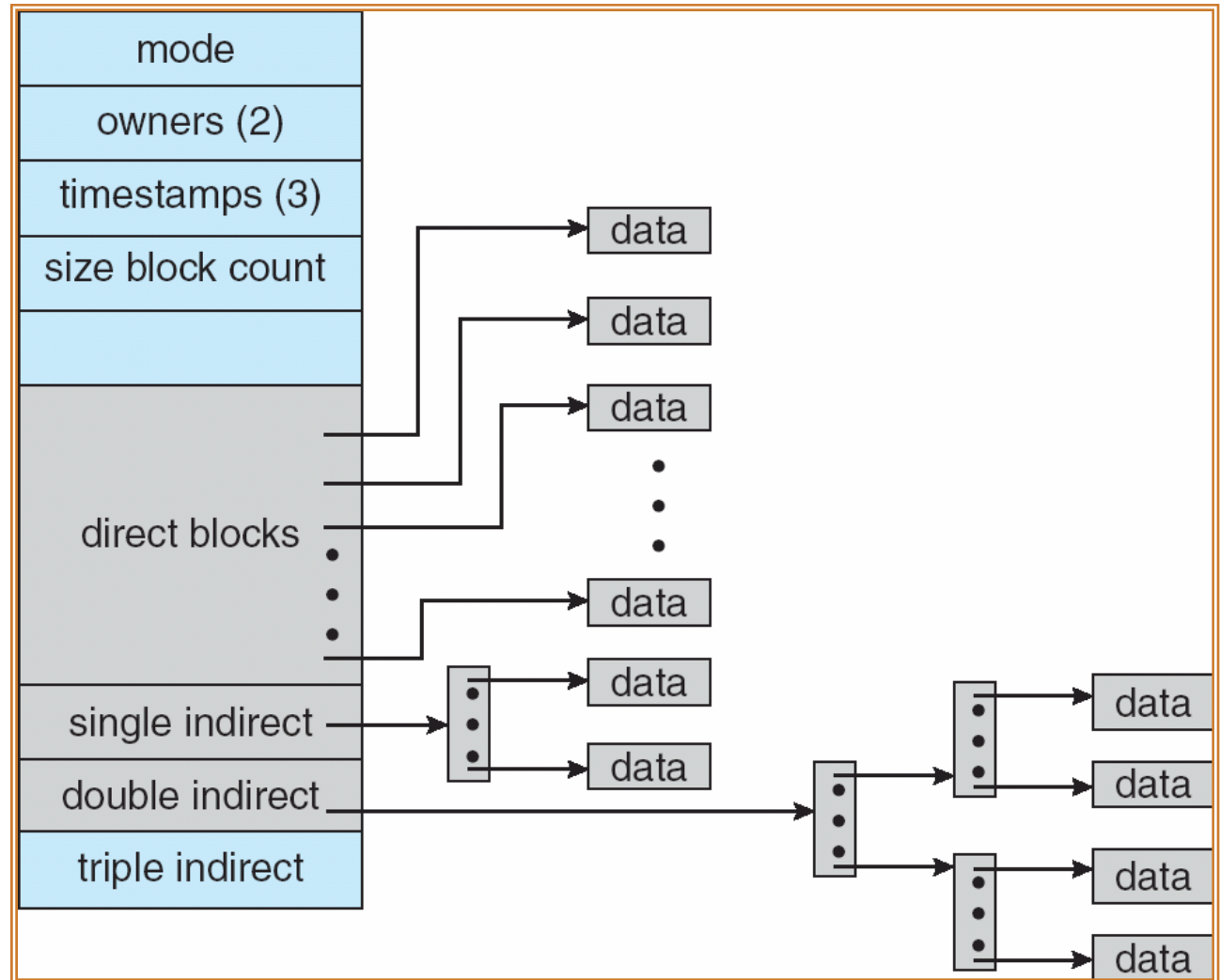
UNIX keeps 12 pointers to blocks in its header. If a file is longer than this, then it uses pointers to single, double, and triple level index blocks.

FILE SYSTEM IMPLEMENTATION

Allocation Methods

UNIX METHOD:

Note that various mechanisms are used here so as to optimize the technique based on the size of the file.



FILE SYSTEM IMPLEMENTATION

Allocation Methods

PERFORMANCE ISSUES FOR THESE METHODS

It's difficult to compare mechanisms because usage is different. Let's calculate, for each method, the number of disk accesses to read block i from a file:

- contiguous:** 1 access from location **start + i**.
- linked:** $i + 1$ accesses, reading each block in turn. (is this a fair example?)
- index:** 2 accesses, 1 for index, 1 for data.

FILE SYSTEM IMPLEMENTATION

Free Space Management

We need a way to keep track of space currently free. This information is needed when we want to create or add (allocate) to a file. When a file is deleted, we need to show what space is freed up.

BIT VECTOR METHOD

- Each block is represented by a bit

1 1 0 0 1 1 0 means blocks 2, 3, 6 are free.

- This method allows an easy way of finding contiguous free blocks. Requires the overhead of disk space to hold the bitmap.
- A block is not REALLY allocated on the disk unless the bitmap is updated.
- What operations (disk requests) are required to create and allocate a file using this implementation?

FILE SYSTEM IMPLEMENTATION

Free Space Management

FREE LIST METHOD

- Free blocks are chained together, each holding a pointer to the next one free.
- This is very inefficient since a disk access is required to look at each sector.

GROUPING METHOD

- In one free block, put lots of pointers to other free blocks. Include a pointer to the next block of pointers.

COUNTING METHOD

- Since many free blocks are contiguous, keep a list of dyads holding the starting address of a "chunk", and the number of blocks in that chunk.
- Format < disk address, number of free blocks >

FILE SYSTEM IMPLEMENTATION

Directory Management

- The issue here is how to be able to search for information about a file in a directory given its name.
- Could have **linear list** of file names with pointers to the data blocks. This is:
 - simple to program **BUT** time consuming to search.
- Could use hash table - a linear list with hash data structure.
 - a) Use the filename to produce a value that's used as entry to hash table.
 - b) Hash table contains where in the list the file data is located.
 - c) This decreases the directory search time (file creation and deletion are faster.)
 - d) Must contend with collisions - where two names hash to the same location.
 - e) The number of hashes generally can't be expanded on the fly.

FILE SYSTEM IMPLEMENTATION

Directory/File Management

GAINING CONSISTENCY

Required when system crashes or data on the disk may be inconsistent:

Consistency checker - compares data in the directory structure with data blocks on disk and tries to fix inconsistencies. For example, What if a file has a pointer to a block, but the bit map for the free-space-management says that block isn't allocated.

Back-up- provides consistency by copying data to a "safe" place.

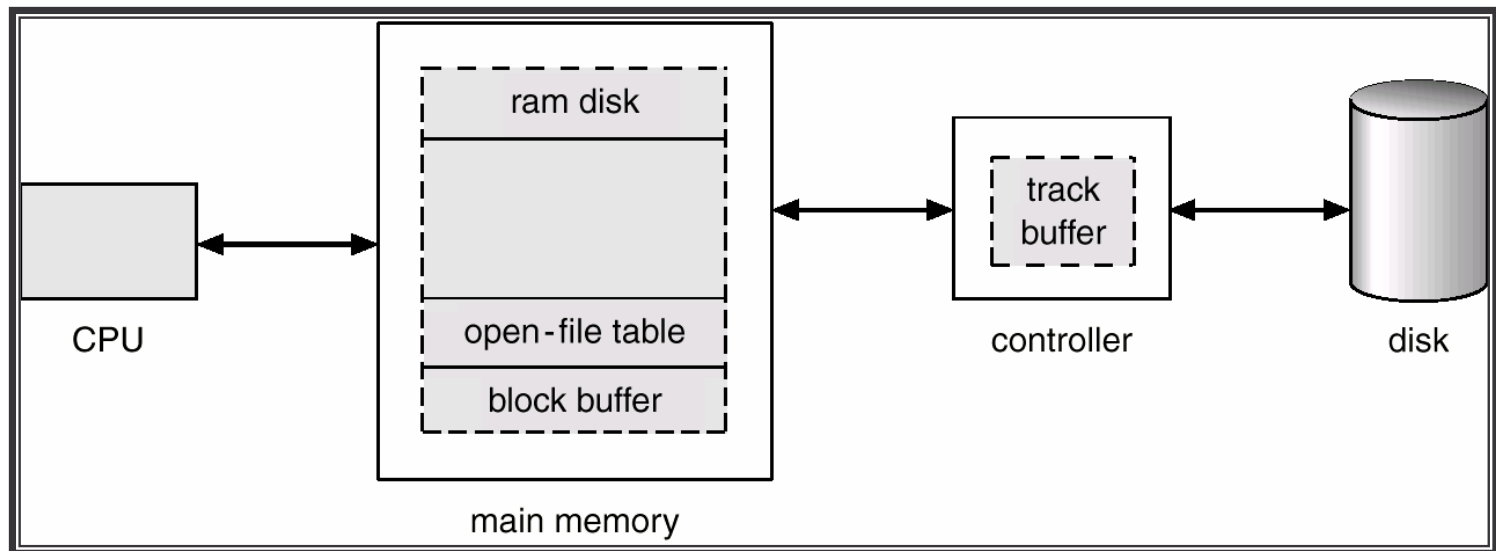
Recovery - occurs when lost data is retrieved from backup.

FILE SYSTEM IMPLEMENTATION

Efficiency and Performance

THE DISK CACHE MECHANISM

- There are many places to store disk data so the system doesn't need to get it from the disk again and again.

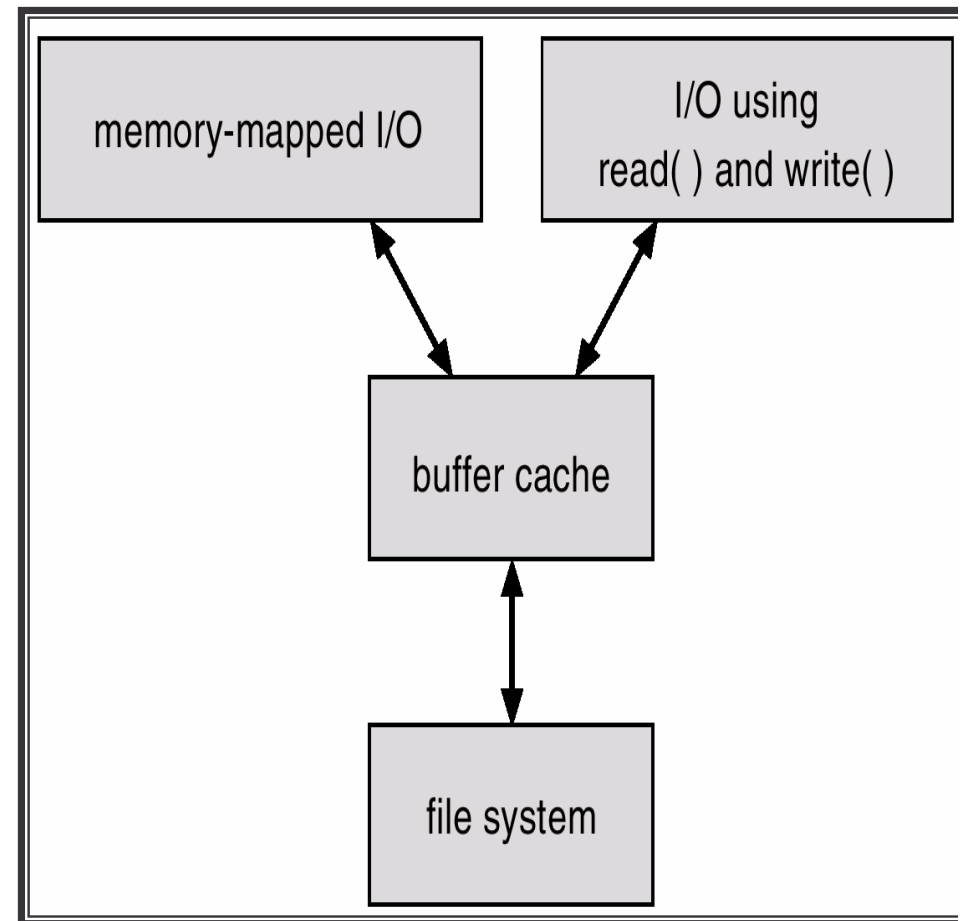


FILE SYSTEM IMPLEMENTATION

Efficiency and Performance

THE DISK CACHE MECHANISM

- This is an essential part of any well-performing Operating System.
- The goal is to ensure that the disk is accessed as seldom as possible.
- Keep previously read data in memory so that it might be read again.
- They also hold on to written data, hoping to aggregate several writes from a process.
- Can also be “smart” and do things like read-ahead. Anticipate what will be needed.



DISTRIBUTED FILE SYSTEMS

SUN Network File System

OVERVIEW:

- Runs on SUNOS - NFS is both an implementation and a specification of how to access remote files. It's both a definition and a specific instance.
- The goal: to share a file system in a transparent way.
- Uses client-server model (for NFS, a node can be both simultaneously.) Can act between any two nodes (no dedicated server.) Mount makes a server file-system visible from a client.

mount server:/usr/shared client:/usr/local

- Then, transparently, a request for /usr/local/dir-server accesses a file that is on the server.
- Can use heterogeneous machines - different hardware, operating systems, network protocols.
- Uses RPC for isolation - thus all implementations must have the same RPC calls. These RPC's implement the mount protocol and the NFS protocol.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

THE MOUNT PROTOCOL:

The following operations occur:

1. The client's request is sent via RPC to the mount server (on server machine.)
2. Mount server checks export list containing
 - a) file systems that can be exported,
 - b) legal requesting clients.
 - c) It's legitimate to mount any directory within the legal filesystem.
3. Server returns "file handle" to client.
4. Server maintains list of clients and mounted directories -- this is state information! But this data is only a "hint" and isn't treated as essential.
5. Mounting often occurs automatically when client or server boots.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

THE NFS PROTOCOL:

RPC's support these remote file operations:

- a) Search for file within directory.
- b) Read a set of directory entries.
- c) Manipulate links and directories.
- d) Read/write file attributes.
- e) Read/write file data.

Note:

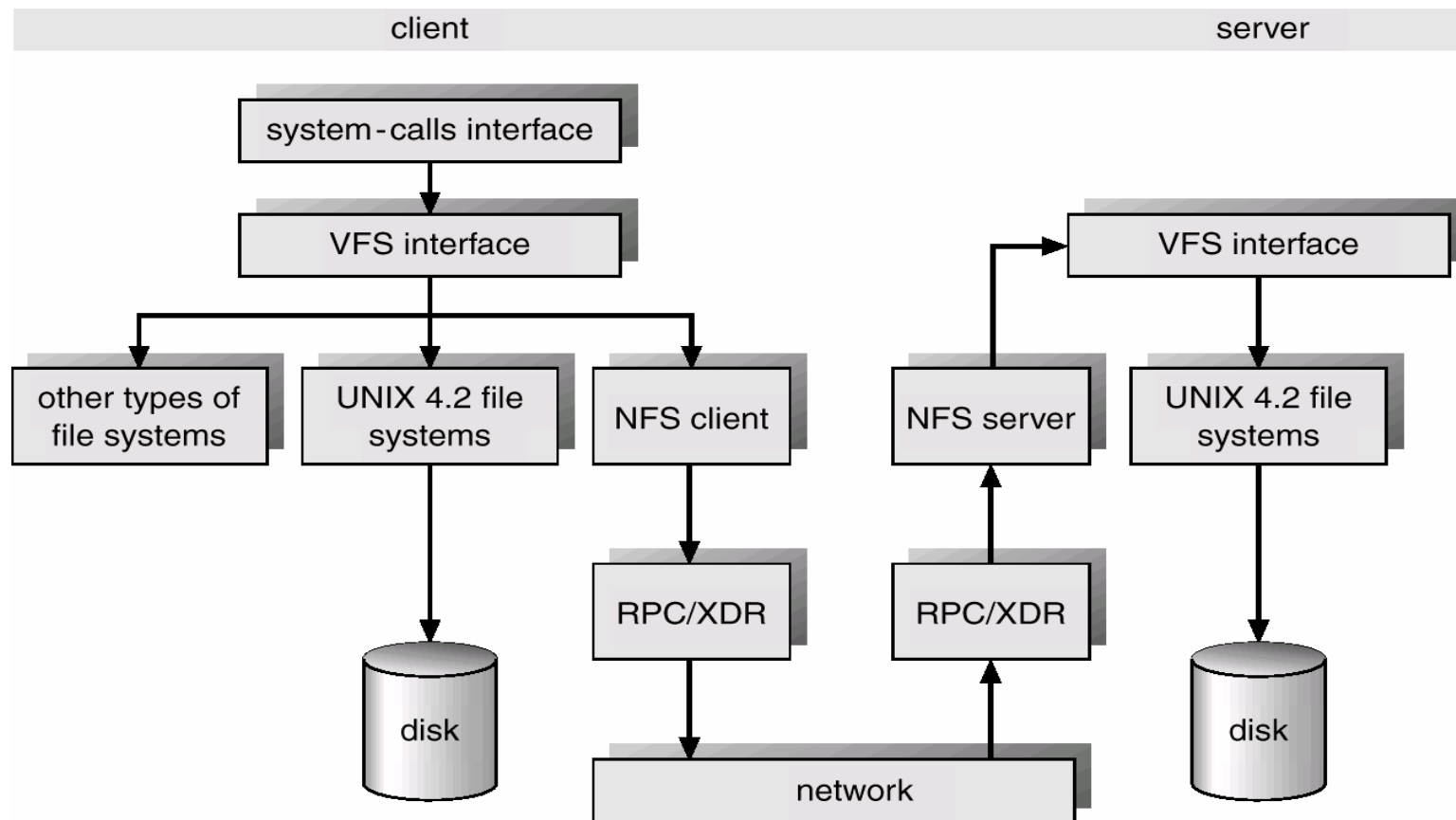
- Open and close are **absent** from this list. NFS servers are **stateless**. Each request must provide all information. With a server crash, no information is lost.
- Modified data must actually get to server disk before client is informed the action is complete. Using a cache would imply state information.
- A single NFS write is **atomic**. A client write request may be broken into several atomic RPC calls, so the whole thing is NOT atomic. Since lock management is stateful, NFS doesn't do it. A higher level must provide this service.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

NFS ARCHITECTURE:

Follow local and remote access through this figure:



DISTRIBUTED FILE SYSTEMS

SUN Network File System

NFS ARCHITECTURE:

1. UNIX filesystem layer - does normal open / read / etc. commands.
2. Virtual file system (VFS) layer -
 - a) Gives clean layer between user and filesystem.
 - b) Acts as deflection point by using global vnodes.
 - c) Understands the difference between local and remote names.
 - d) Keeps in memory information about what should be deflected (mounted directories) and how to get to these remote directories.
3. System call interface layer -
 - a) Presents sanitized validated requests in a uniform way to the VFS.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

PATH-NAME TRANSLATION:

- Break the complete pathname into components.
- For each component, do an NFS lookup using the
component name + directory vnode.
- After a mount point is reached, each component piece will cause a server access.
- Can't hand the whole operation to server since the client may have a second mount on a subsidiary directory (a mount on a mount).
- A directory name cache on the client speeds up lookups.

DISTRIBUTED FILE SYSTEMS

SUN Network File System

CACHES OF REMOTE DATA:

- The client keeps:
 - File block cache - (the contents of a file)
 - File attribute cache - (file header info (inode in UNIX)).
- The local kernel hangs on to the data after getting it the first time.
- On an open, local kernel, it checks with server that cached data is still OK.
- Cached attributes are thrown away after a few seconds.
- Data blocks use read ahead and delayed write.
- Mechanism has:
 - Server consistency problems.
 - Good performance.

FILE SYSTEMS

Wrap Up

In this section we have looked at how the file is put together. What are the components that must be present in the file and implicitly, what procedures must be in the Operating System in order to act on these files.

We've also examined the internal structure of files.

This gives a file system knowledge about how to get around in the file – especially how to find the required data block.