Systems Design and Implementation

I.5 – File Systems

System Architecture Group, SS 2009

University of Karlsruhe

May 26, 2009

Jan Stoess

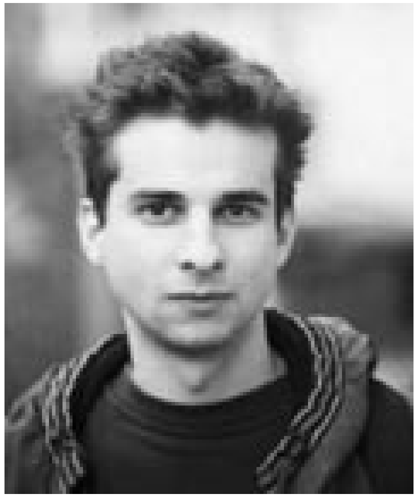
Philipp Kupferschmied

University of Karlsruhe

Reminder

- All SDI Groups:
  Please contact the tutor before your presentations take place!

- Tutor
  - Marcel Noe
  - Consultation Time:
    Monday, 16.00 - 18.00
  - R154, 50.34



Overview

- Introduction
  - Motivation
  - File types, attributes, access, operations
  - Directory types, operations
  - Implementing files and directories
  - *[Parts taken from A.Tanenbaums slides on modern OSes]*
- Case Studies:
  - FAT
  - NFS

Why files?

- Tanenbaum's motivation for files:
  - Enable storing large amount of data
  - Make data survive termination of processes or the system
  - Let processes access persistent data concurrently
- My 2cents
  - Structure your data

Source: Andy Tanenbaum: **Modern Operating Systems, 2nd edition**. Supplementary powerpoint slides, http://www.cs.vu.nl/~ast/books/book_software.html

File naming and structuring

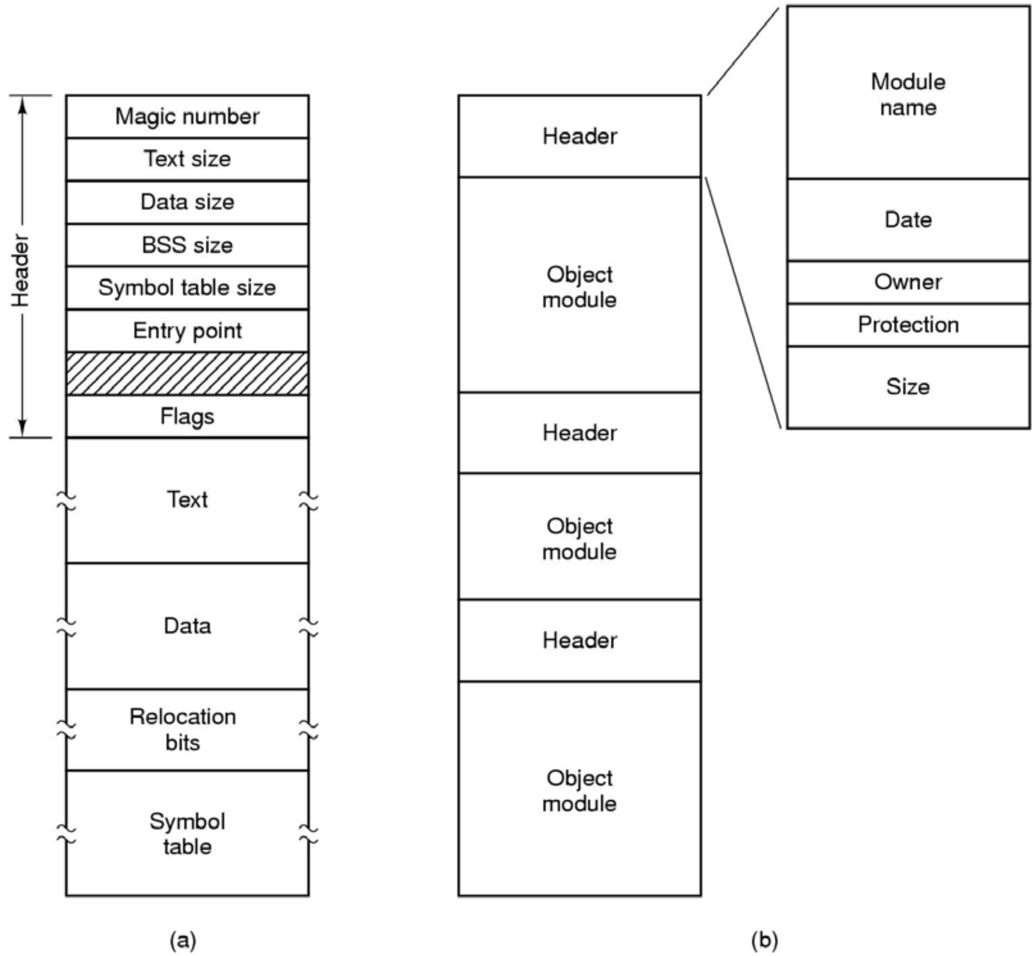
Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Typical file extensions.

File Structure

- Three kinds of files
  - byte sequence
  - record sequence
  - tree

File Types



(a) An executable file (b) An archive

File Attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Possible file attributes

File Access

- Sequential access
  - read all bytes/records from the beginning
  - cannot jump around, but rewind
  - convenient when medium was mag tape
- Random access
  - bytes/records read in any order
  - essential for data base systems
  - read can be ...
    - move file marker (seek), then read or ...
    - read and then move file marker

File Operations

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attributes
10. Set Attributes
11. Rename

Example Program Using File System Calls

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                     /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);          /* ANSI prototype */

#define BUF_SIZE 4096                      /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                   /* protection bits for output file */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);               /* syntax error if argc is not 3 */
```

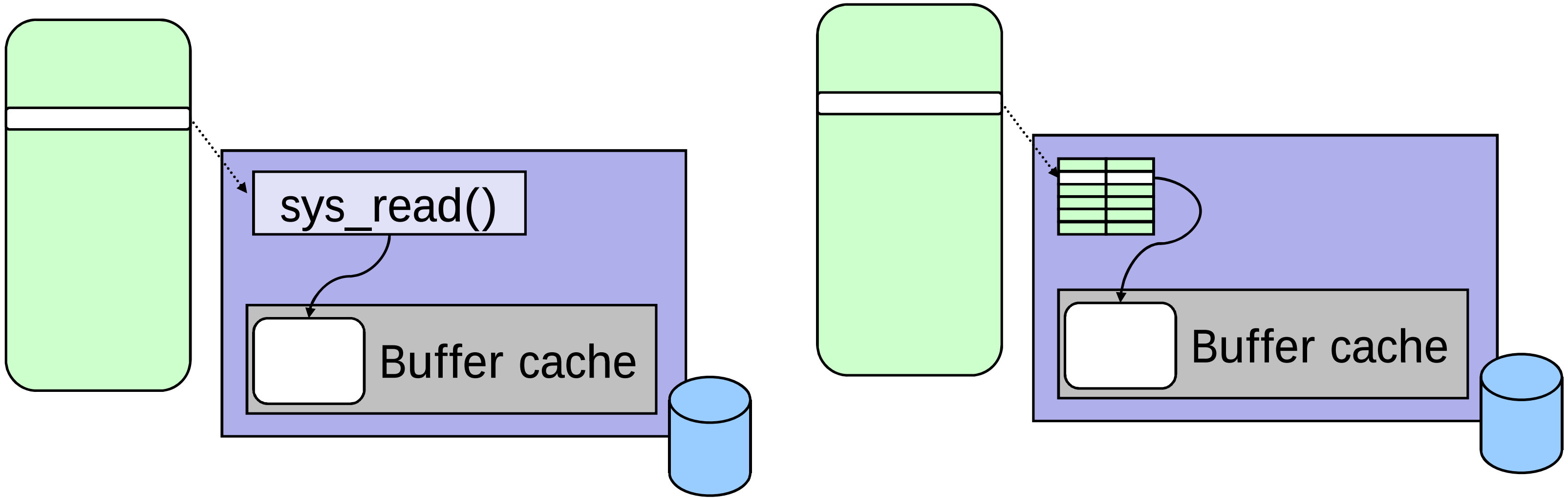
Example Program Using File System Calls

```
/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY);     /* open the source file */
if (in_fd < 0) exit(2);              /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
if (out_fd < 0) exit(3);             /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
if (rd_count <= 0) break;            /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);  /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0)                   /* no error on last read */
        exit(0);
else
        exit(5);                     /* error on last read */
}
```

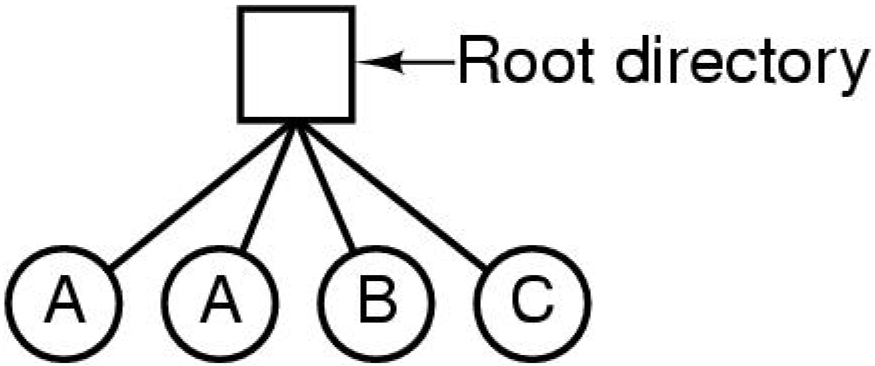
Memory-Mapped Files



(a) Reading files using file system calls

(b) Reading files using memory mappings

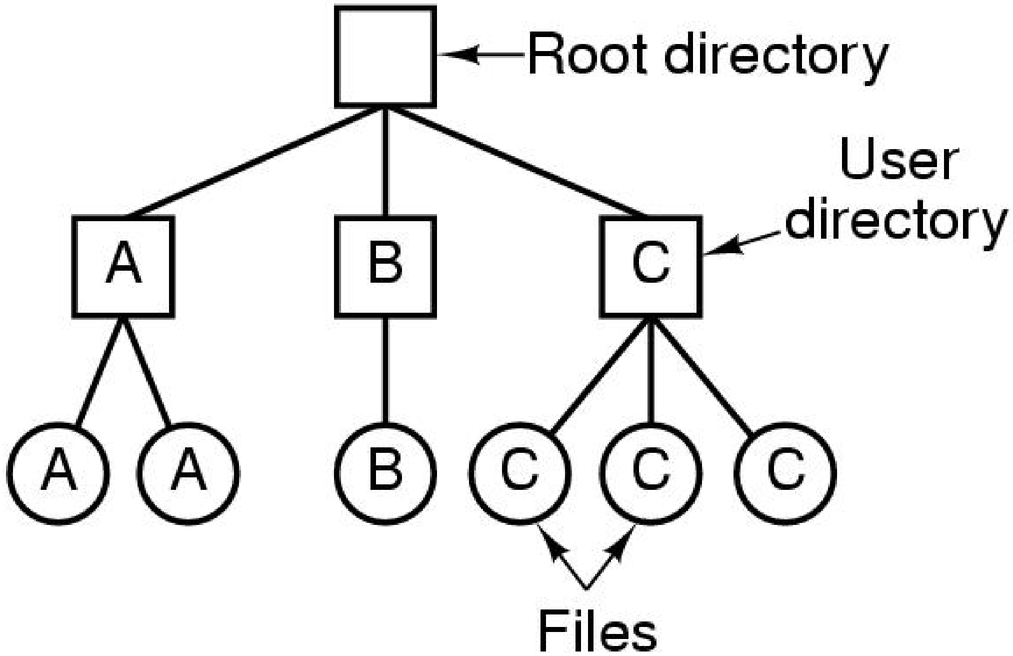
Directories
Single-Level Directory Systems



- A single level directory system
  - contains 4 files
  - owned by 3 different people, A, B, and C
- (Letters indicate *owners* of the directories and files)

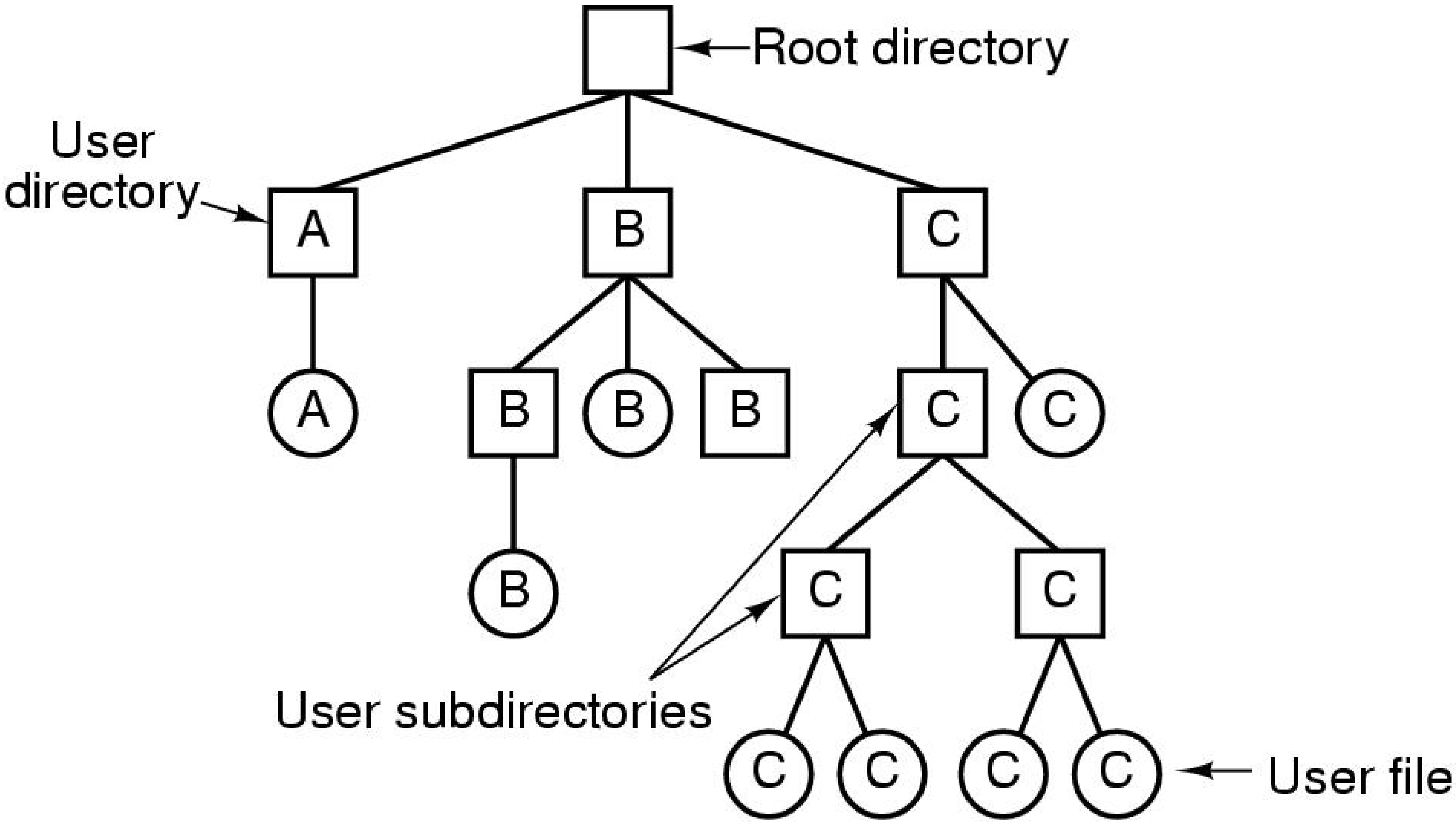
Directories

Two level Directory Systems



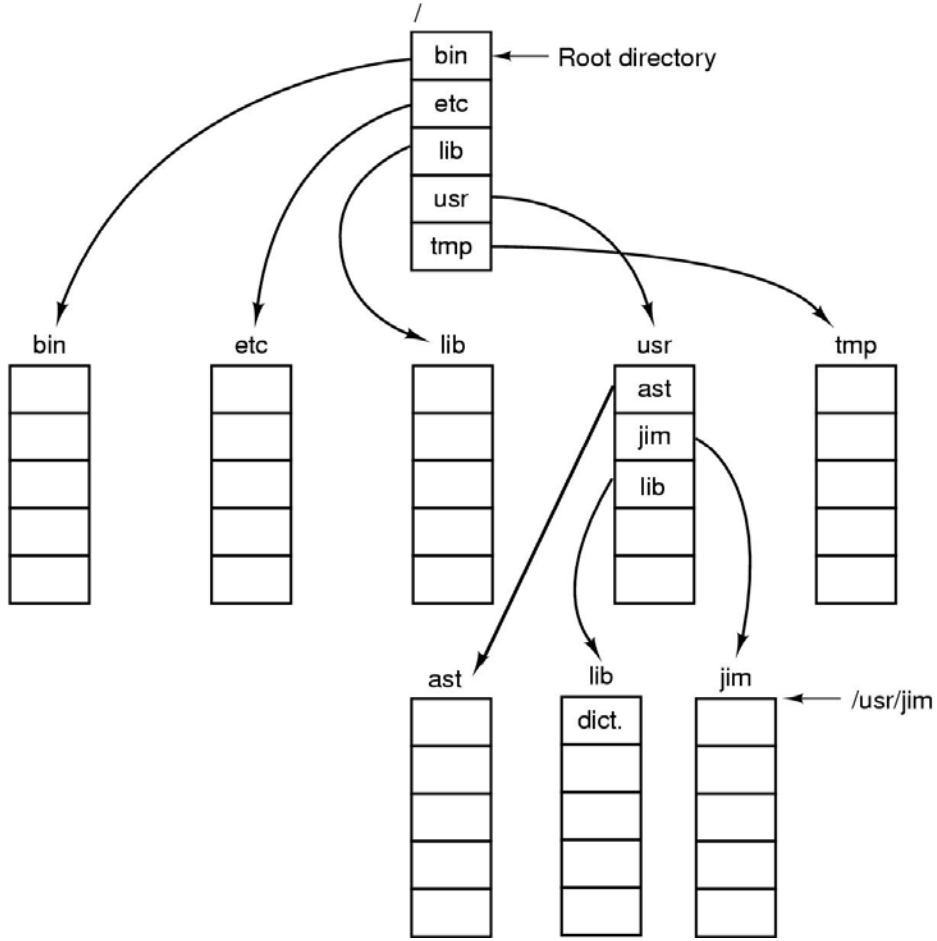
Directories

Hierarchical Directory Systems



A hierarchical directory system

Directory and Path Names

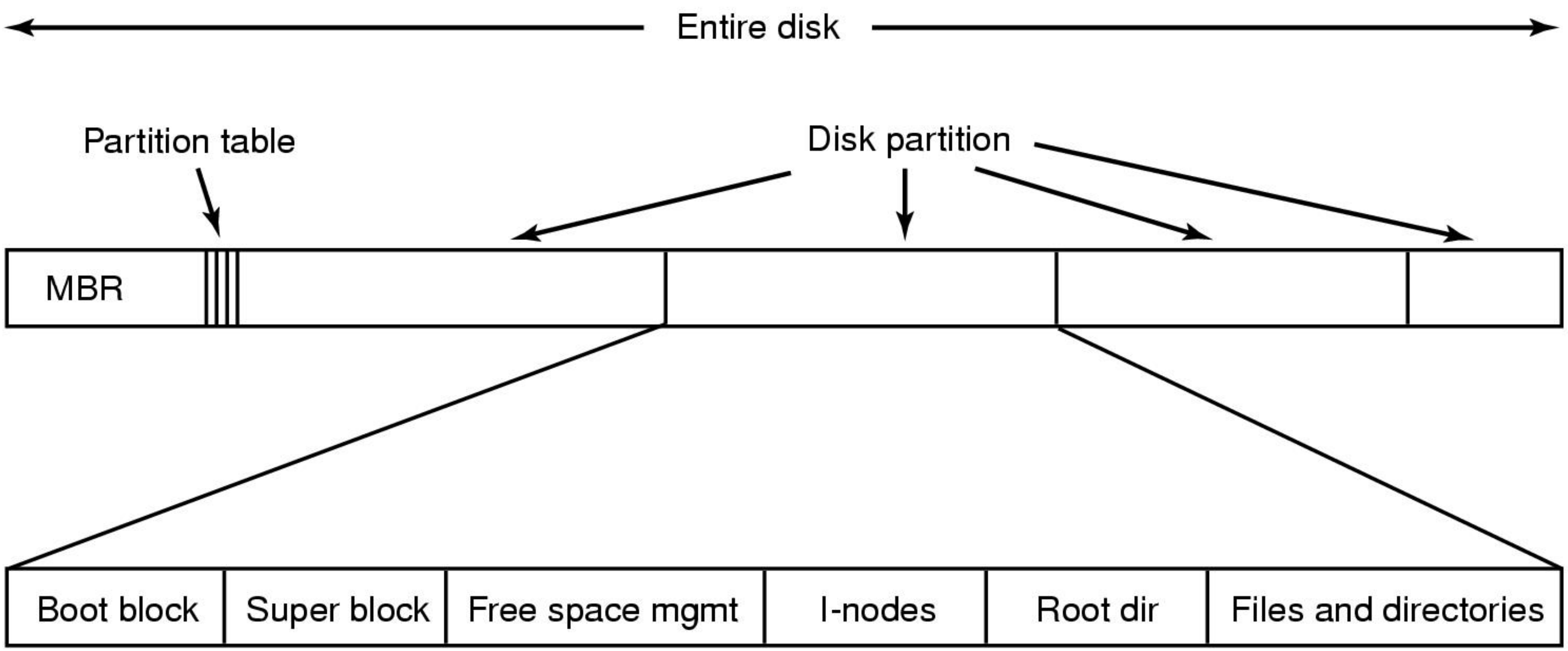


A UNIX directory tree

Directory Operations

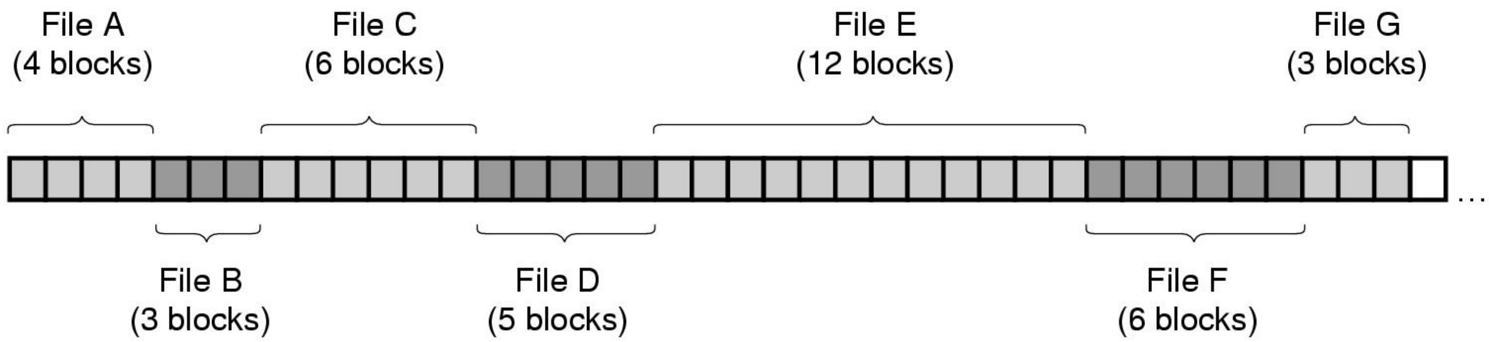
1. Mkdir
2. Rmdir
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

File System Implementation



A possible file system layout

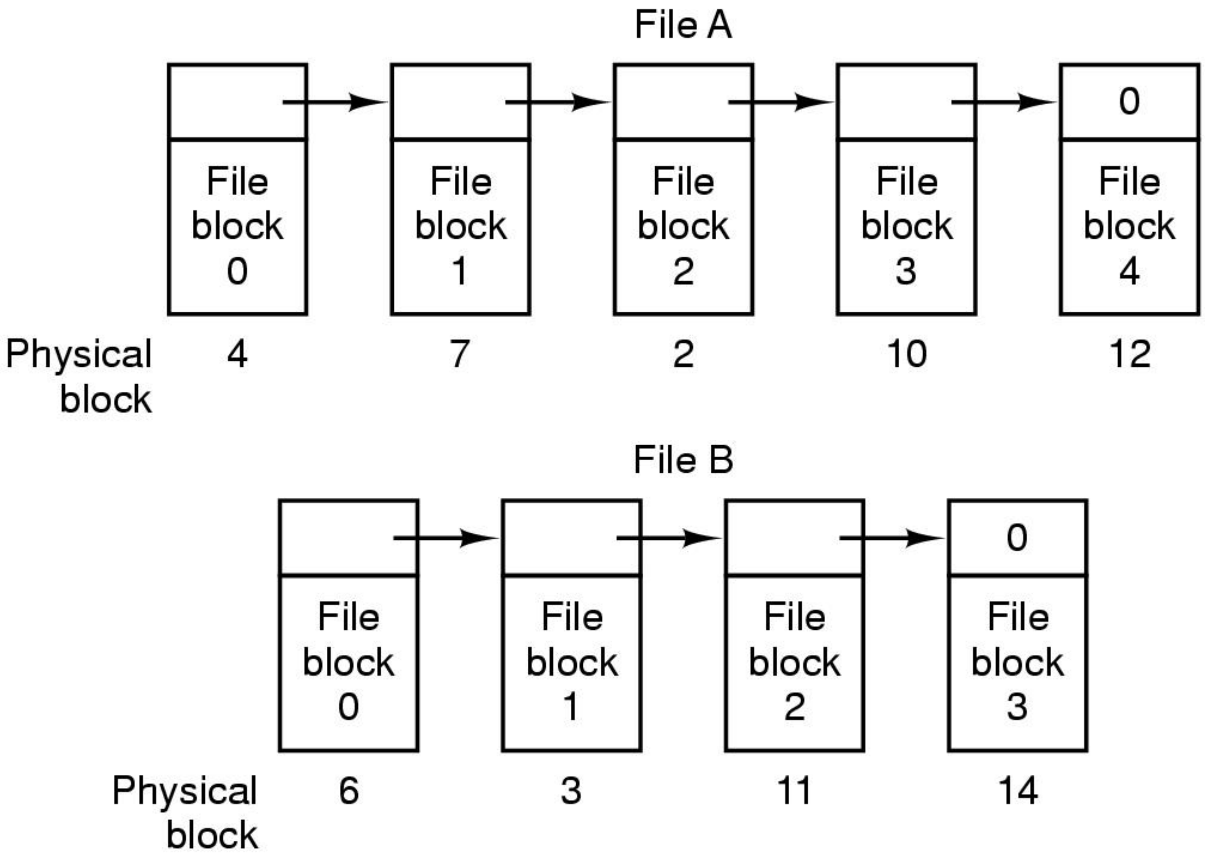
Implementing Files



(a) Contiguous allocation of disk space for 7 files

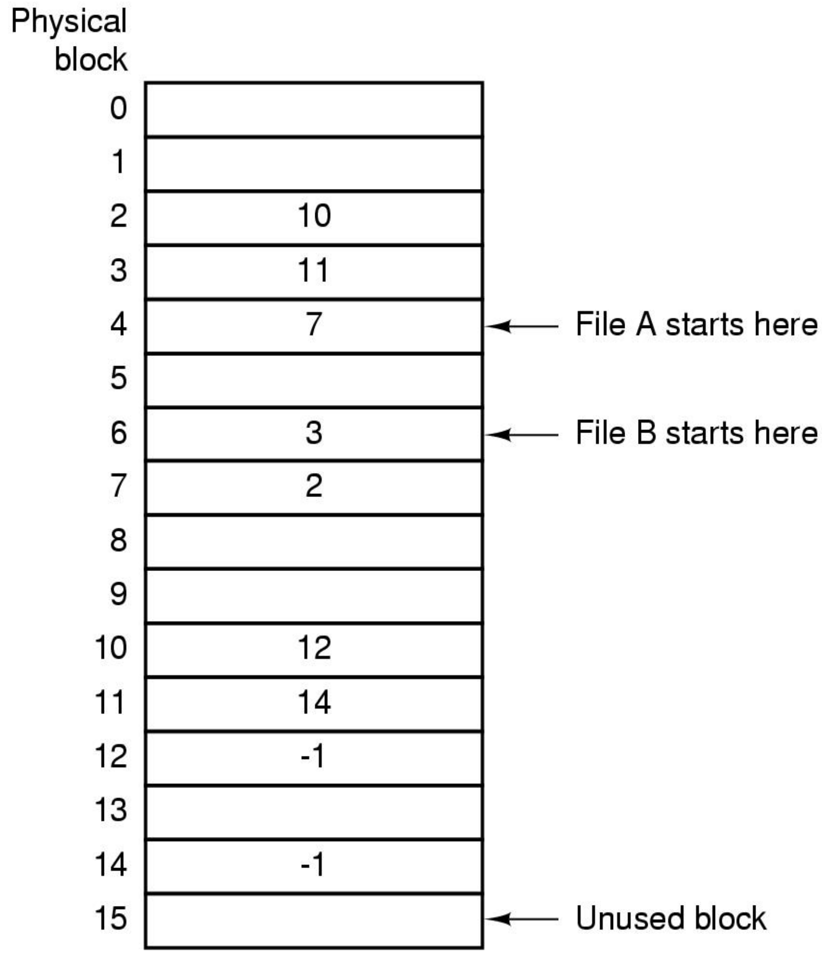
(b) State of the disk after files *D* and *E* have been removed

Implementing Files



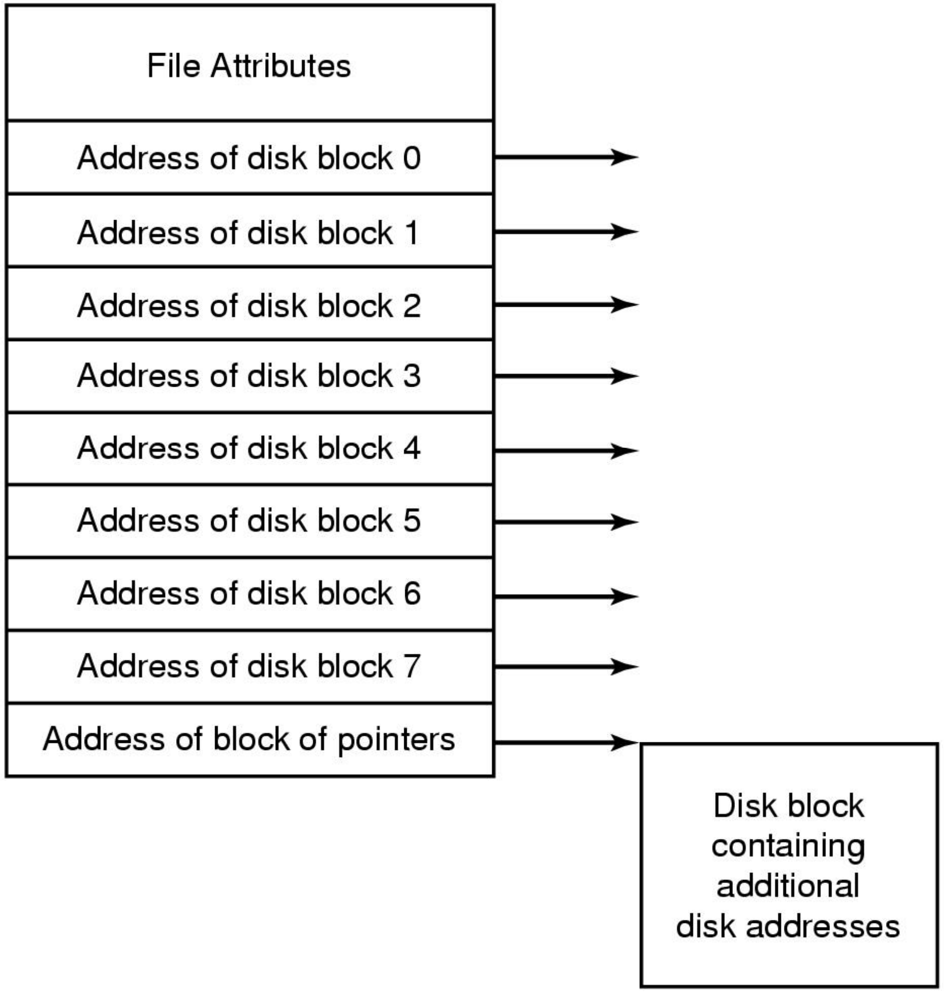
Storing a file as a linked list of disk blocks

Implementing Files



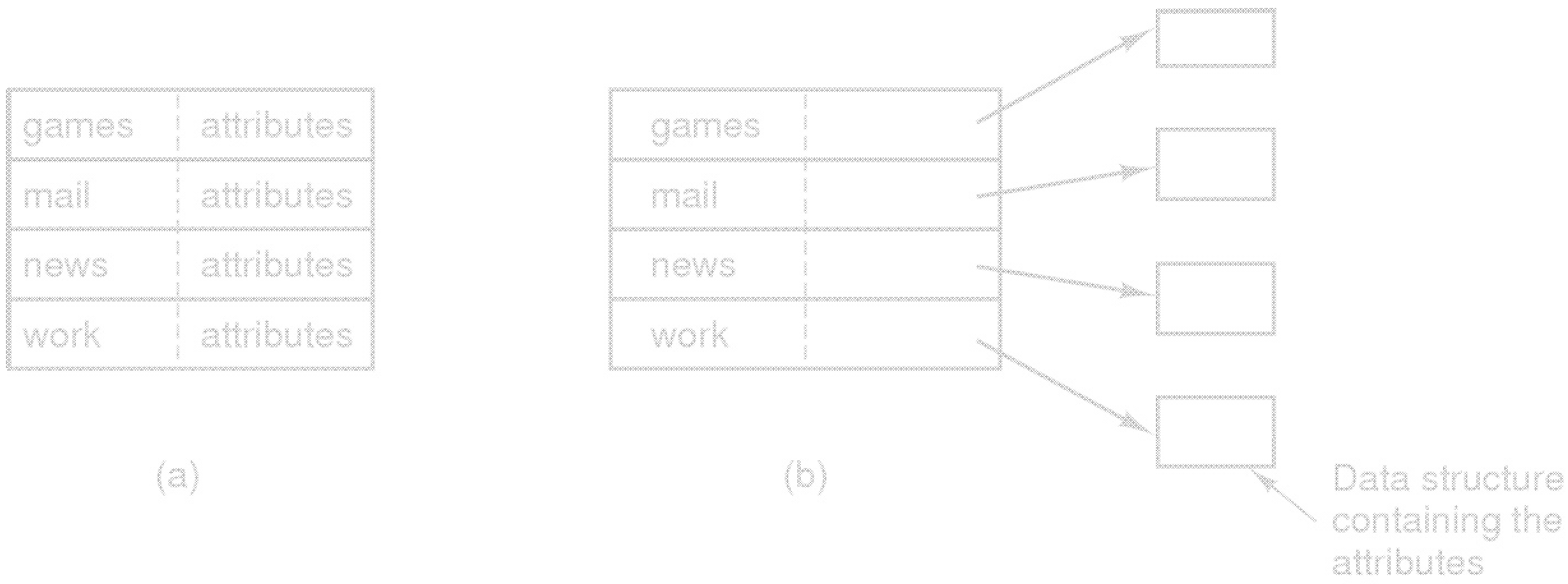
Linked list allocation using a file allocation table in RAM

Implementing Files



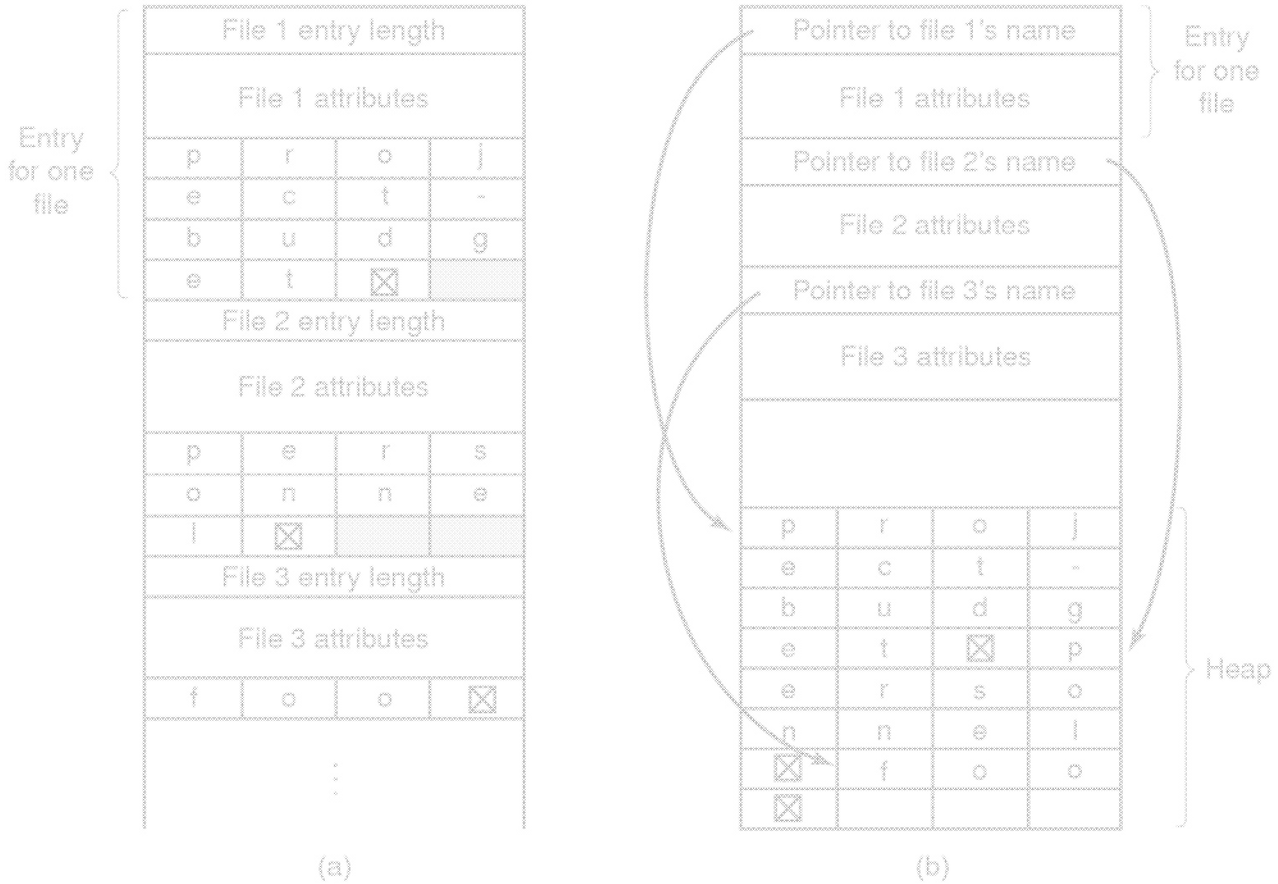
An example i-node

Implementing Directories



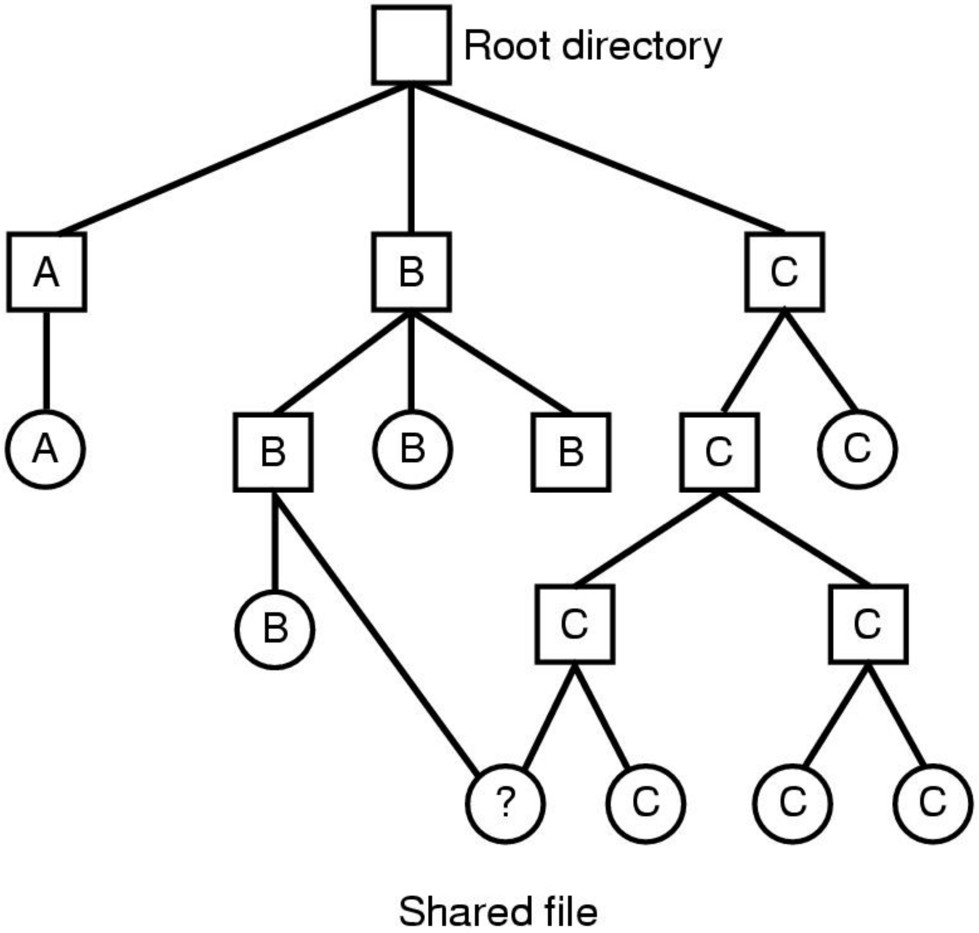
- A simple directory
  - fixed size entries
  - disk addresses and attributes in directory entry
- Directory in which each entry just refers to an i-node

Implementing Directories



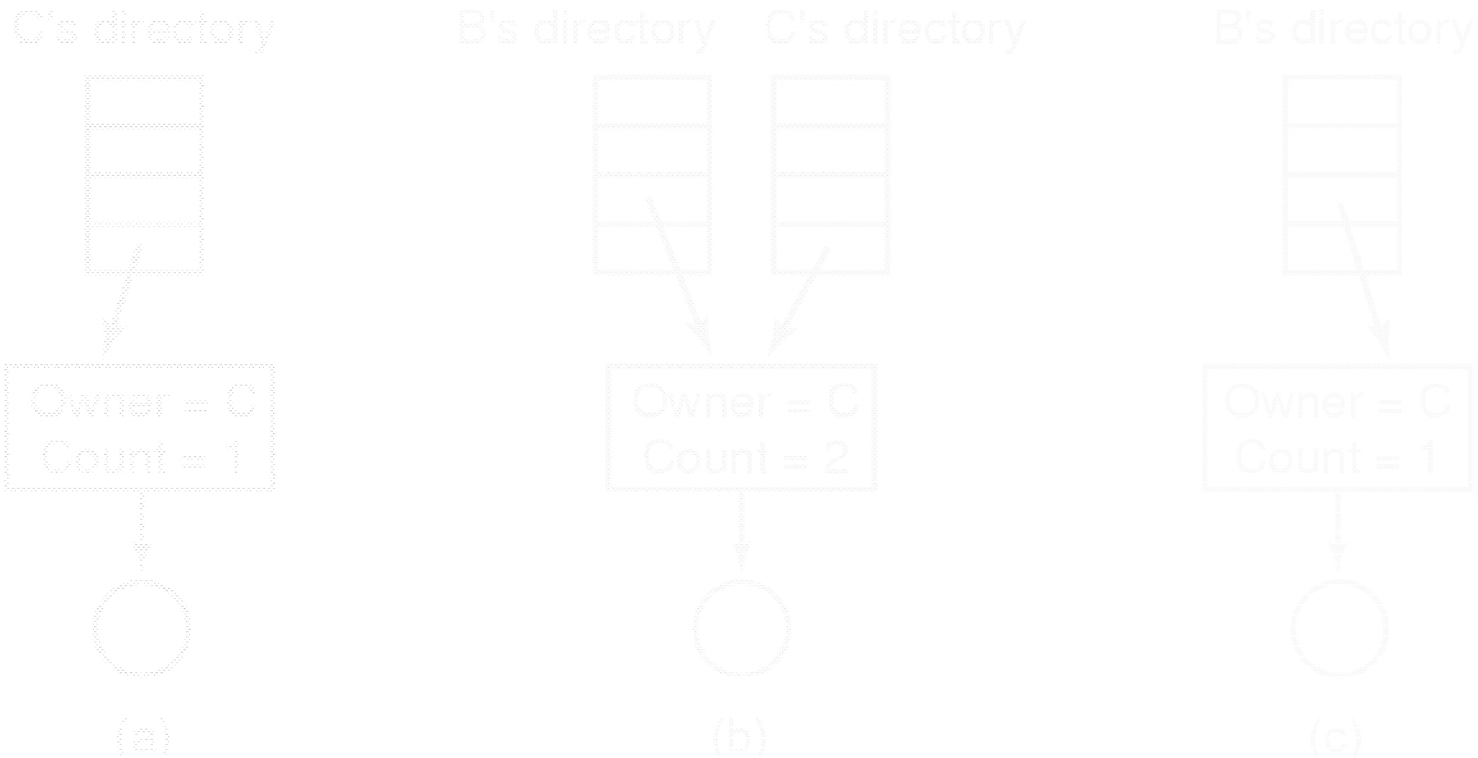
- Two ways of handling long file names in directory
  - In-line
  - In a heap

Shared Files



File system containing a shared file

Shared Files



- Situation prior to linking
- After the link is created
- After the original owner removes the file

The FAT file system

- Origins in the late 1970s
- Simple file system
  - Floppy disks
  - less than 500K size.
- Enhanced to support larger data.
- FAT = file allocation table
  - Specifies used/free areas of disk
- 3 FAT file system types
  - FAT12
  - FAT16
  - FAT32
  - Specifies #bits/entry in FAT structure

Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

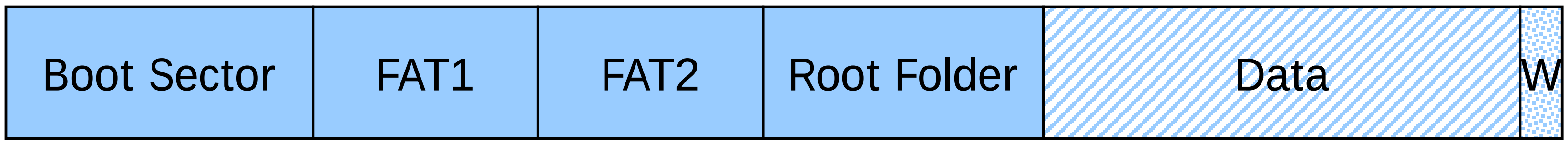
FAT structures

- Boot record
- Cluster
  - Group of data sectors on disk
  - Used to store file and directory data
  - Number of sectors stored in boot record
- File allocation table (FAT)
  - Simple array of 12/16/32 bit entries
  - Singly linked list of cluster chains (files)
  - 2 synchronized copies / disk

Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

FAT structures

- Root directory
  - Normal directory without ".."
  - Location hardcoded after FAT
- Data area
  - Arranged in clusters
- Wasted sectors
  - #sectors % sizeof(cluster)



Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

FAT entries

- Simple bit field

0x00000000	Free
0x00000001	Reserved
0x00000002-0xFFFFEFFF	Used; value points to next cluster
0xFFFFFFF7	Bad
0xFFFFFFF8-0xFFFFFFFF	Last cluster in file

Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

Directories

- Directories are special files
  - Table of file entries
- Structure of file entries
  - Name + Extension (fixed size)
  - Attributes
  - Create time
  - Last access date
  - Last modified time
  - Last modified date
  - Starting cluster number
  - File size
- Long file names
  - Phony entries (invalid volume attribute)
  - Ignored by most old DOS programs
  - New programs can retrieve LFN from entry

Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

Opening a file

- Go to parent directory
- Search for file entry
  - Retrieve first cluster number
  - Retrieve data from cluster
- For more clusters
  - Go to FAT
  - Retrieve entry of first cluster
  - Follow chain of clusters
  - Retrieve data from clusters

Source: Microsoft Corporation. **Microsoft Extensible Firmware Initiative FAT32 File System Specification**. FAT: General Overview of On-Disk Format. Version 1.03, 2000

NFS – The Network File System

- Invented by Sun Microsystems, mid 1980s
- Idea:
  - Transparent, remote access to filesystems
  - Portability to different OSes and architectures
- Approach:
  - specified using external data representation (XDR)
    - describes protocols machine-independently
  - based on RPC package
    - Simplify protocol definition, implementation, maintenance

Source: R.Sandberg et al. ***Design and Implementation of the Sun Network Filesystem.*** Proceeding of the USENIX 1985 Summer Conference

NFS – The Network File System

- First implementation
  - UNIX 4.2 kernel
  - Completely new kernel interface
  - Separates generic from specific filesystem implementations
  - Two basic parts
    - VFS: operations on a filesystem
    - VNode: operations on a file

Source: R.Sandberg et al. ***Design and Implementation of the Sun Network Filesystem.*** Proceeding of the USENIX 1985 Summer Conference

NFS Design considerations

- Goals:
  - Machine and OS independence
  - Crash recovery
  - Transparent access
  - Maintain UNIX semantics on client
  - Reasonable performance

Source: R.Sandberg et al. ***Design and Implementation of the Sun Network Filesystem.*** Proceeding of the USENIX 1985 Summer Conference

NFS Design considerations

- Basic design
  - Uses RPC mechanism
    - Protocol defined as a set of procedures, arguments and results
    - Synchronous behavior
  - Stateless protocol
    - Each call contains all information to complete the call
    - Stateful alternative discarded since
      - Client would need to detect server crashes
      - Server would need to detect client crashes (why?)
    - No recovery needed after crash
    - No difference between crashed and slow server

NFS Design considerations

- Basic design
  - RPC package is transport independent
    - First implementation uses UDP/IP
  - Most common parameter: file handle
    - Provided by server
    - Used by client as reference
    - Opaque for client

NFS protocol procedures

- **null**() returns ()
- **lookup**(*dirfh*, *name*) returns (*fh, attr*)
- **create**(*dirfh*, *name*, *attr*) returns (*newfh, attr*)
- **remove**(*dirfh*, *name*) returns (*status*)

- **getattr**(*fh*) returns (*attr*)
- **setattr**(*fh,* attr) returns (*attr*)

- **read**(*fh, offset, count*) returns (*attr, data*)
- **write**(*fh , offset, count, data*) returns (*attr*)
- **rename**(*dirfh, name, tofh, toname*) returns (*status*)

NFS protocol procedures

- **link**(*dirfh, name, tofh, toname*) returns (*status*)
- **symlink**(*dirfh, name, string*) returns (*status*)
- **readlink**(***fh***) returns (*string*)

- **mkdir**(*dirfh*, *name, attr*) returns (*fh, newattr*)
- **rmdir**(*dirfh*, *name*) returns (*status*)
- **readdir**(*dirfh, cookie, count*) returns(*entries*)

  ↑ *next*

- **statfs**(*fh*) returns (*fsstats*)

NFS protocol procedures

- Filesystem root obtained via external *mount* protocol
  - Takes UNIX directory pathname
  - Checks permissions
  - Returns filehandle
  - Idea:
    - Easy extension of filesystem access checks
    - Only place where UNIX names are used
- External data representation XDR
  - Similar to IDL
  - Specification of data types
  - Specification of procedures
  - Defines size, byte order, alignment of data types
  - C-like definition

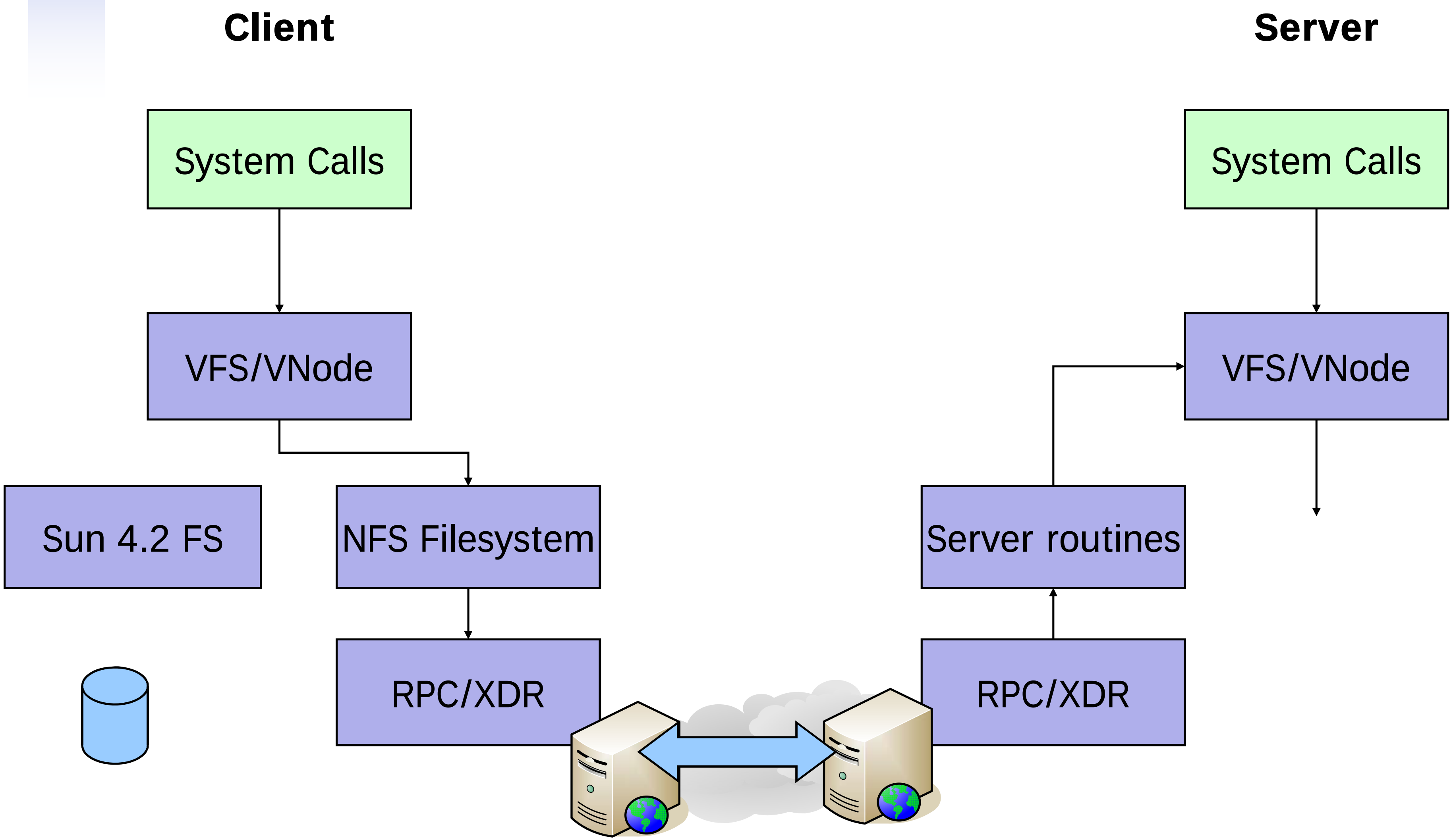
NFS Server Side

- Stateless server
  - No server-internal caching
  - Server flushes modified data immediately
- Filehandle generation
  - Filehandle =
    <filesystem id, inode number, inode generation number>
  - NFS introduces filesystem IDs
  - NFS introduces inode generation number (what for?)

NFS Client Side

- Need transparent access to remote files
  - Do not change path name structure
  - Explicit <host:/path> not backwards compatible
  - Approach:
    - Do hostname lookup and file address binding once
    - Attach remote filesystem to local path
    - Use *mount* protocol
  - Implementation:
    - Add new filesystem interface to the kernel
      - VFS: operations on a remote file system
      - VNode: operations on files within a file system

NFS Filesystem Interface



NFS Filesystem Interface

- Filesystem operations
  - per filesystem
  - **mount, mount_root**
- VFS operations
  - per mounted filesystem
  - **unmount, root, statfs, sync**
- VNode operations
  - **lookup, create, remove, rename**
  - **open, close, rdwr, ioctl, select**
  - **getattr, setattr, access**
  - **mkdir, rmdir, readdir**
  - **link, symlink, readlink**
  - ...

NFS Filesystem Interface

- VNode operations
  - some operations map to NFS procedures, some not
- Pathname lookup
  - Problem:
    - Pathname could contain mountpoint
    - Mount information is contained in the client, above the VNode layer
    - Server cannot keep track of client mount points
  - Approach:
    - Break path into components
    - Do lookup per component
    - Cache lookups in the client

NFS implementation

- Completed around 1984
  - Implemented VNodes in the kernel
  - RPC, XDR ported to kernel
  - User-level mount service
  - User-level NFS server daemon allows for sleeping

NFS Problems

- Root filesystems
  - Sharing root file systems not possible
    - /tmp: names of temporary files are created with local names (process id)
    - /dev: no remote device access system
    - Approach:
      - Share root FS partly, e.g., /usr only
- Filesystem naming
  - Client can mount a filesystem several times
  - Different names for the same file system
  - Increases confusion
  - Approach:
    - Structure mountpoint names, e.g., /usr/server1

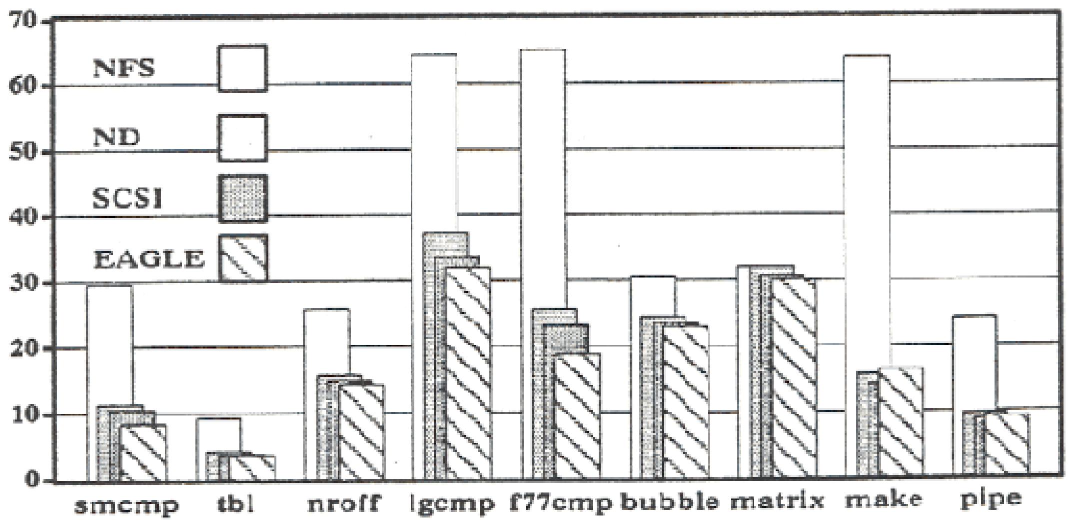
NFS Problems

- Credentials and security
  - Wanted UNIX style permissions
  - Possible via RPC permission model
    - Pass authentication parameters with RPC
    - UID, GID
  - Problem: global UIDs, GIDs required
    - Administrative hassle
    - Solution: Yellow pages (YP)
      Database-like networked user/group administration
  - Problem: remote *root* access
    - Remote root should not be equal to local root
    - Solution: map root access to a special UID (nobody)
    - Problem: root may have fewer rights to files than users!
      root still can impersonate every local user

NFS Problems

- Concurrent access:
  - No agreed-on concurrency model for files
  - Thus, NFS does not provide file locking
- UNIX open file semantics:
  - Problem:
    - can **open** a file and **unlink** afterwards
    - strange but necessary semantics (tmp files)
  - Solution:
    - Rename a file temporarily on server
    - Client removes file after close
  - Similar problem: file access changes on open file
- Time skew:
  - E.g., making dependencies on remote files
  - Solution: NTP (planned)

Initial NFS Performance



- First version had pretty bad performance

NFS Performance Optimizations

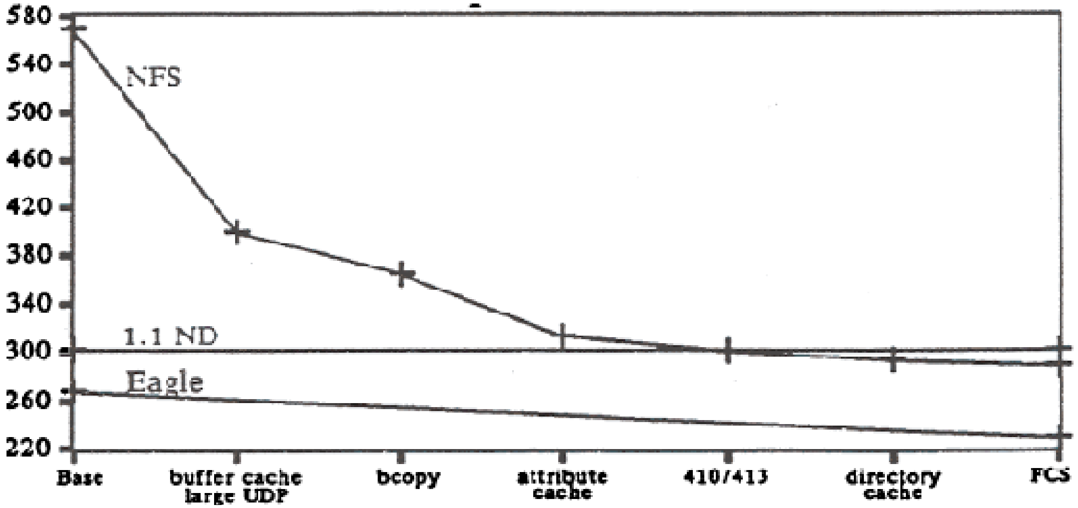
- Decrease number of **read** and **write** calls
  - Add client cache
  - Flush cache on close
  - Helped a lot
- Avoid extensive copying
  - Do XDR translation in place
  - Saves 1 buffer copy
- **gettattr** accounted for 90% of server calls
  - **stat** on client produces 11 (!) getattr RPCs
  - Add attribute cache
  - Flushed periodically (every 3 seconds)
  - Dropped to 10%

NFS Performance Optimizations

- Make sequential reads faster
  - Add read ahead in the server
  - For on-demand executables:
    - Cluster on-demand loading requests
    - For small programs, load all pages at once
- Increase **lookup** performance
  - Add client name lookup cache
  - Contains vnodes for remote directory names
  - Flushed when retrieved attributes (modify time) don't match cached vnode attributes

NFS Performance Optimizations

- Performance after optimizations



- Problems remaining:
  - Frequently executed **stat** calls are costly
  - **write** is synchronous by design

NFS Future Work (anno 1984)

- Future work
  - Diskless mode for clients
  - Remote file locking
  - Other filesystem types
  - Performance
  - Security improvements
  - Automatic mounting

SDI File Service Design

- File names maintained by name server
- Names translate into a session handle as seen by the client
- The session handle maps to disk blocks in the file server

SDI File Service Design

- Fileserver design
  - Stateful
  - Stateless
- Fileserver interfaces
  - File handle layout
  - Operations on files
  - Operations on directories
  - File attributes (basic)
- Fileserver implementation
  - Fileserver / Nameserver relationship
  - Data transfer: copying, mapping
  - Stateful fileserver: which state to hold

SDI File Service Design Groups (2)

- Groups
  - SDI 3
  - SDI 6
- Presentation
  - June 04, 2009
- Please don’t forget to discuss your slides with Marcel beforehand