File Systems

Operating Systems

Department of Computer Science
Rutgers University

File System

File system is an abstraction of the disk

- File → Tracks/sectors
- File Control Block stores mapping info (+ protection, timestamps, size, etc)
- To a user process
  - A file looks like a contiguous block of bytes (Unix)
  - A file system provides a coherent view of a group of files
  - A file system provides protection

API: create, open, delete, read, write files

Performance: throughput vs. response time

Reliability: minimize the potential for lost or destroyed data

- E.g., RAID could be implemented in the OS (disk device driver)

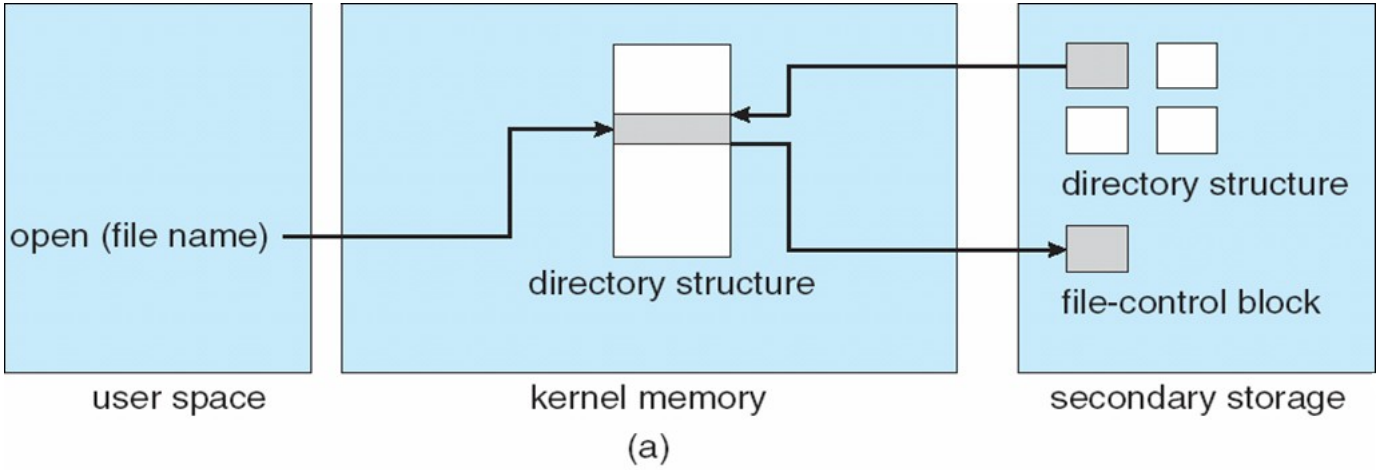
File API

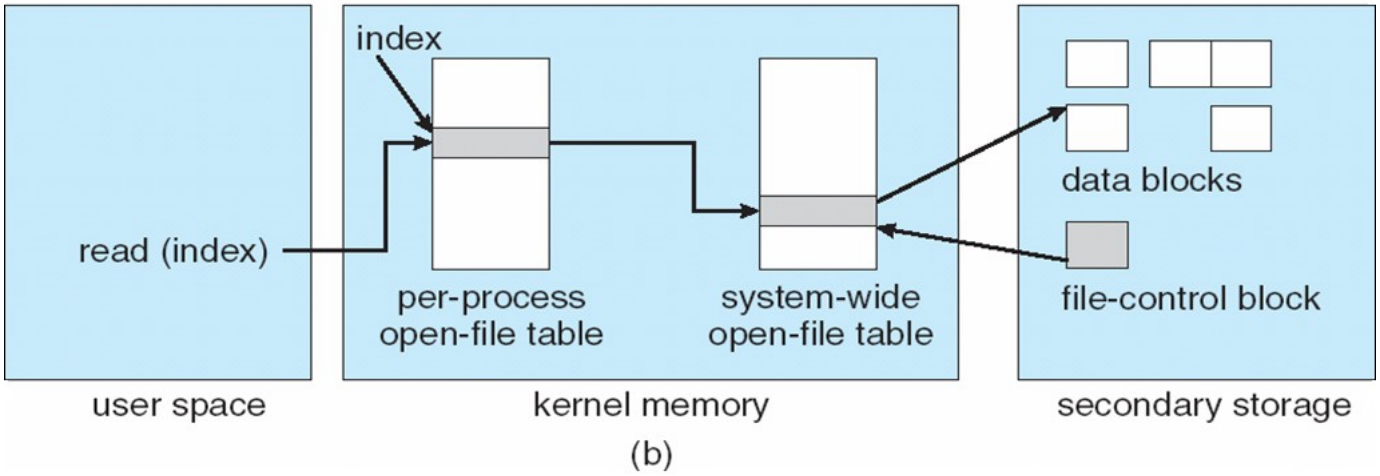
To read or write, need to open

- open() returns a handle to the opened file
- OS associates a (per-process) data structure with the handle
- This data structure maintains current "cursor" position in the stream of bytes in the file
  - Read and write takes place from the current position
  - Can specify a different location explicitly

When done, should close the file

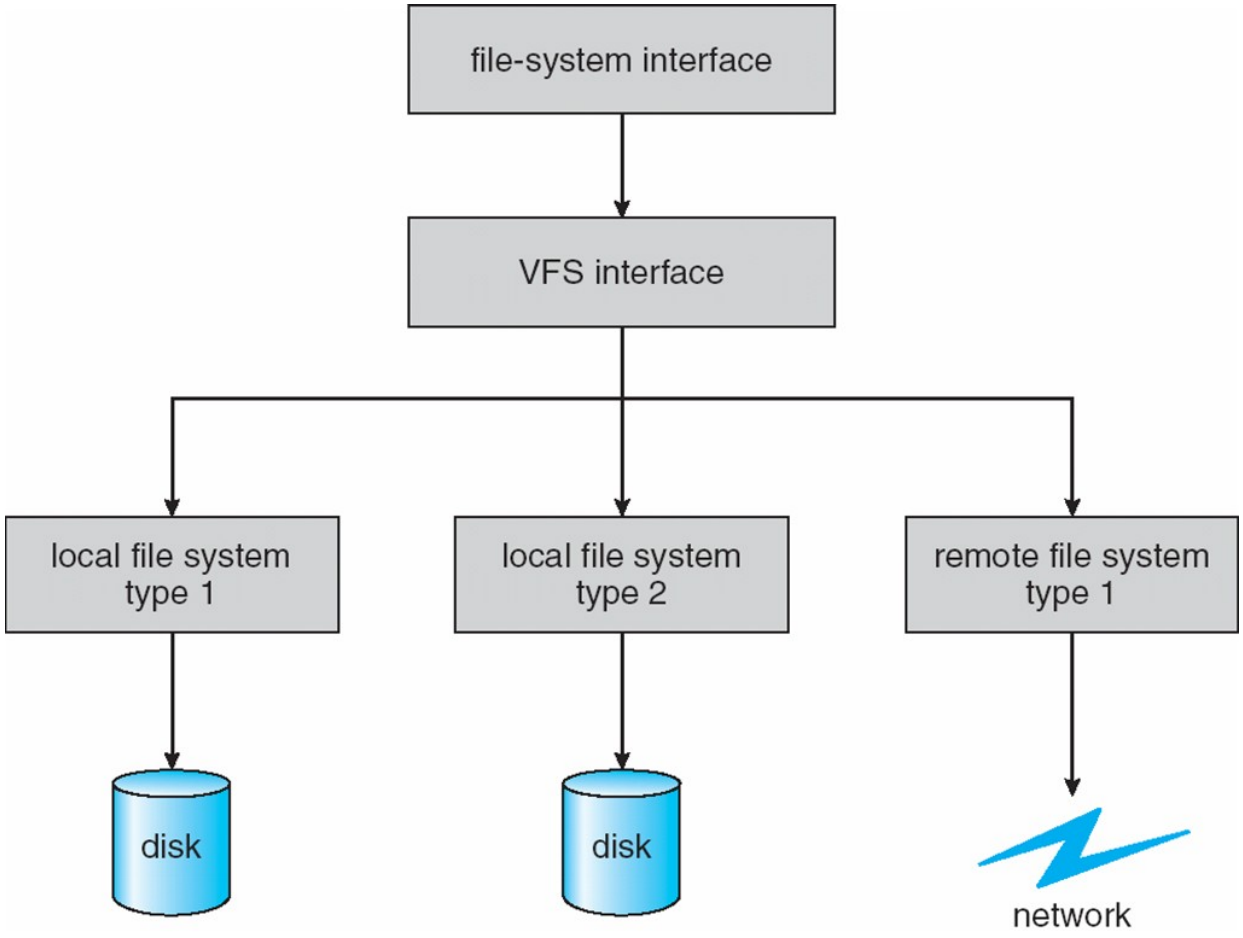
In-Memory File System Structures



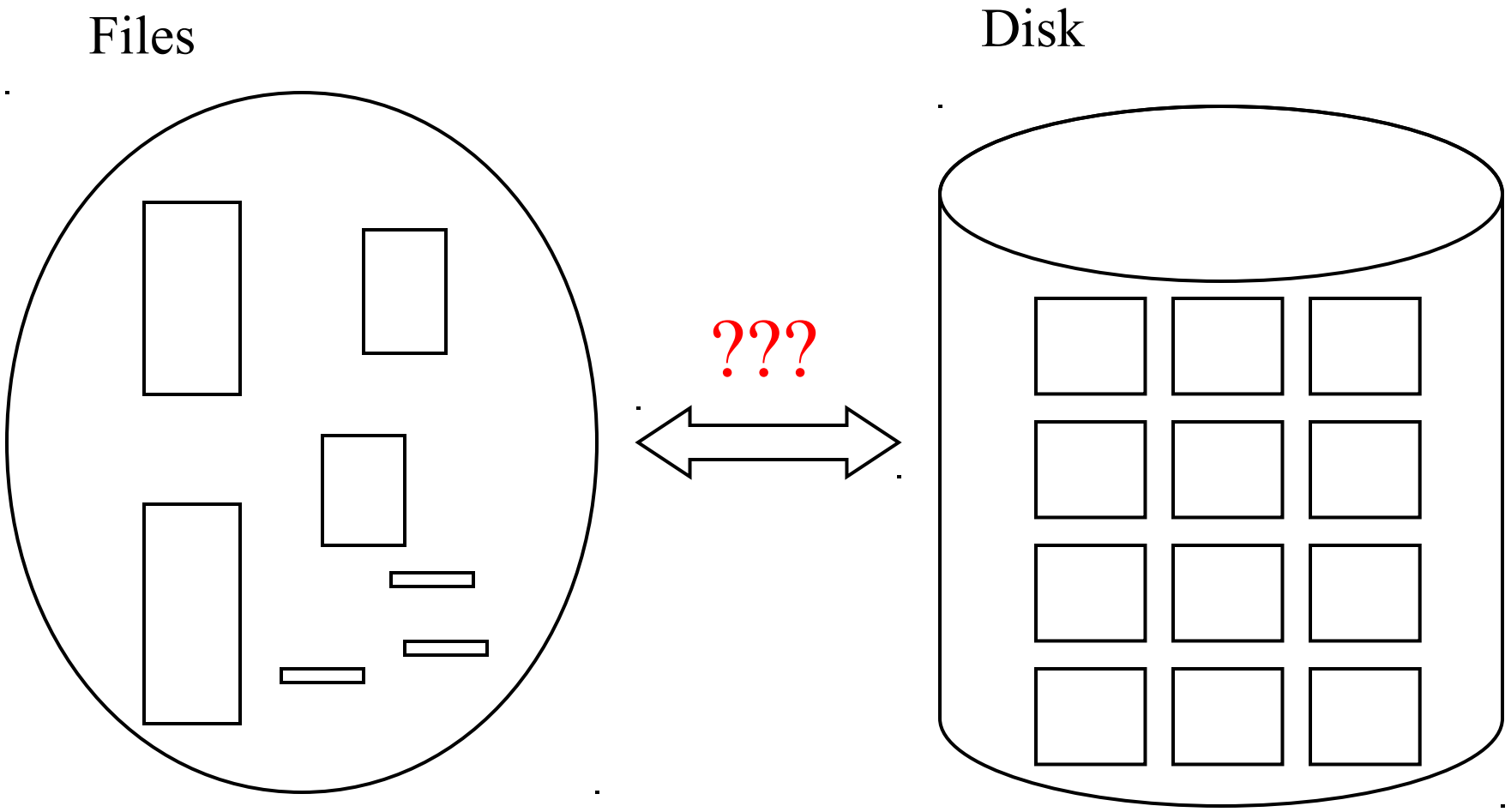


Virtual File Systems

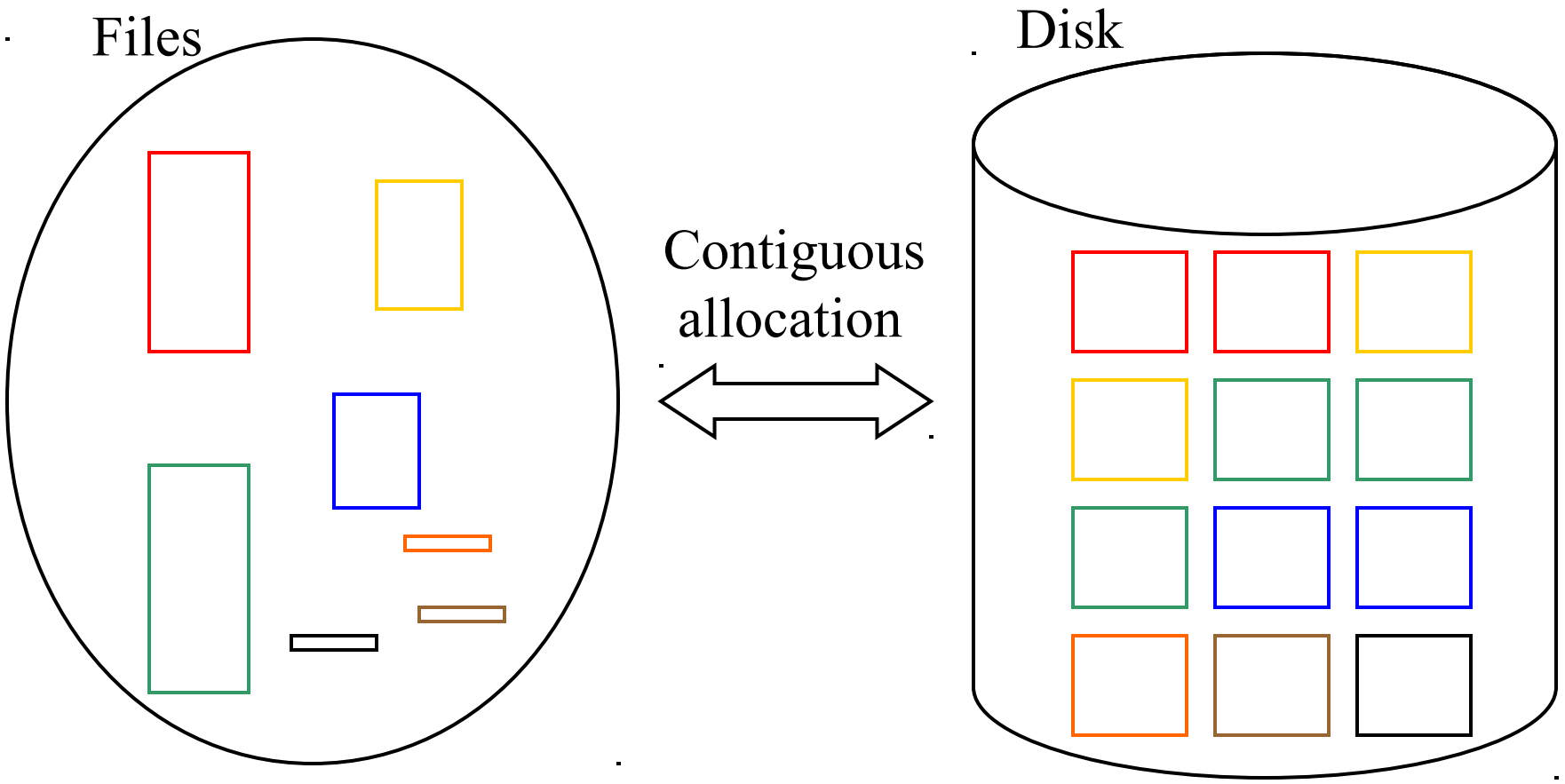
- Virtual file systems allow the same API to be used by different types of file systems
- The API is to the VFS, rather than any specific type of FS



Files vs. Disk

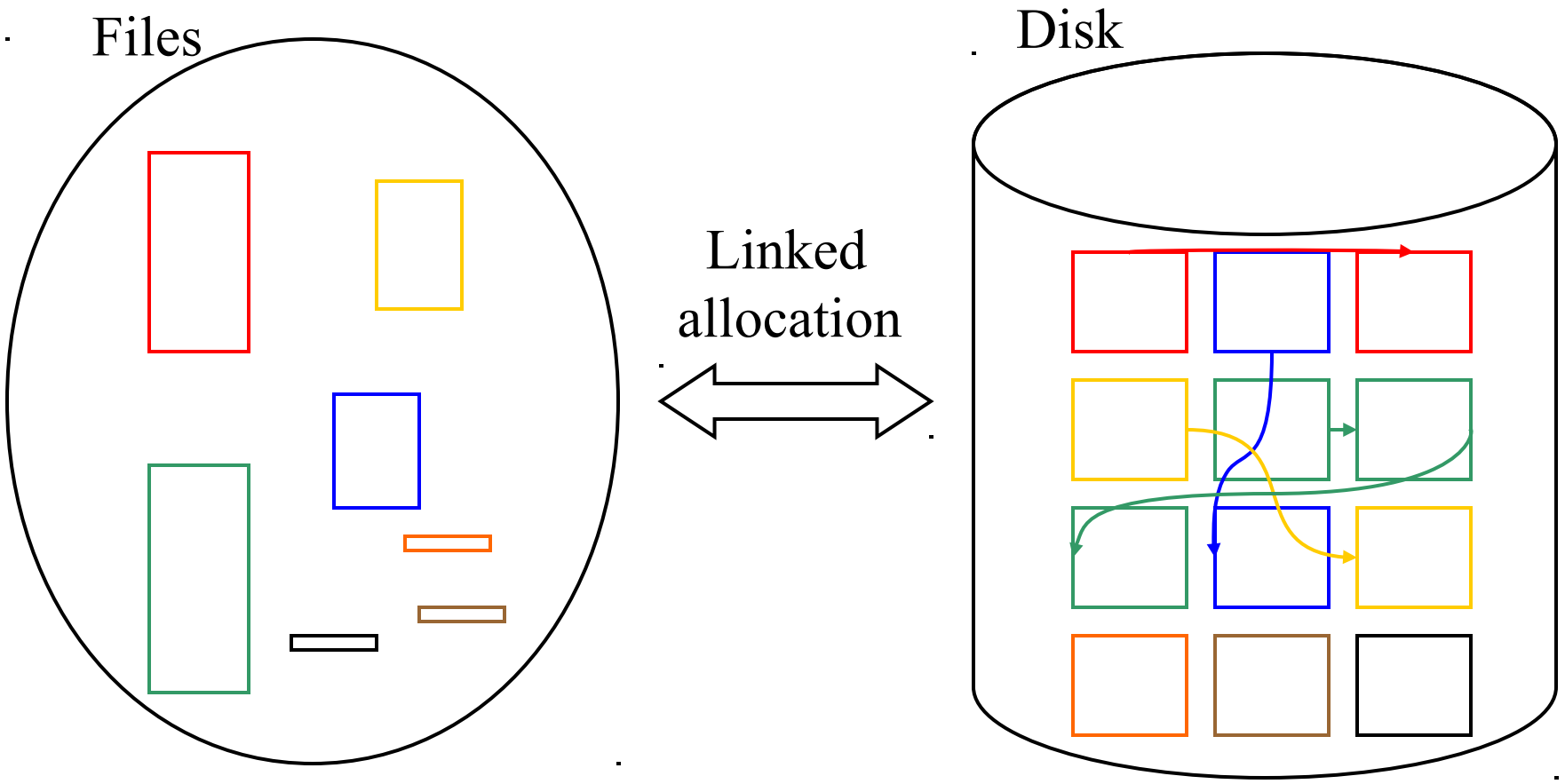


Files vs. Disk



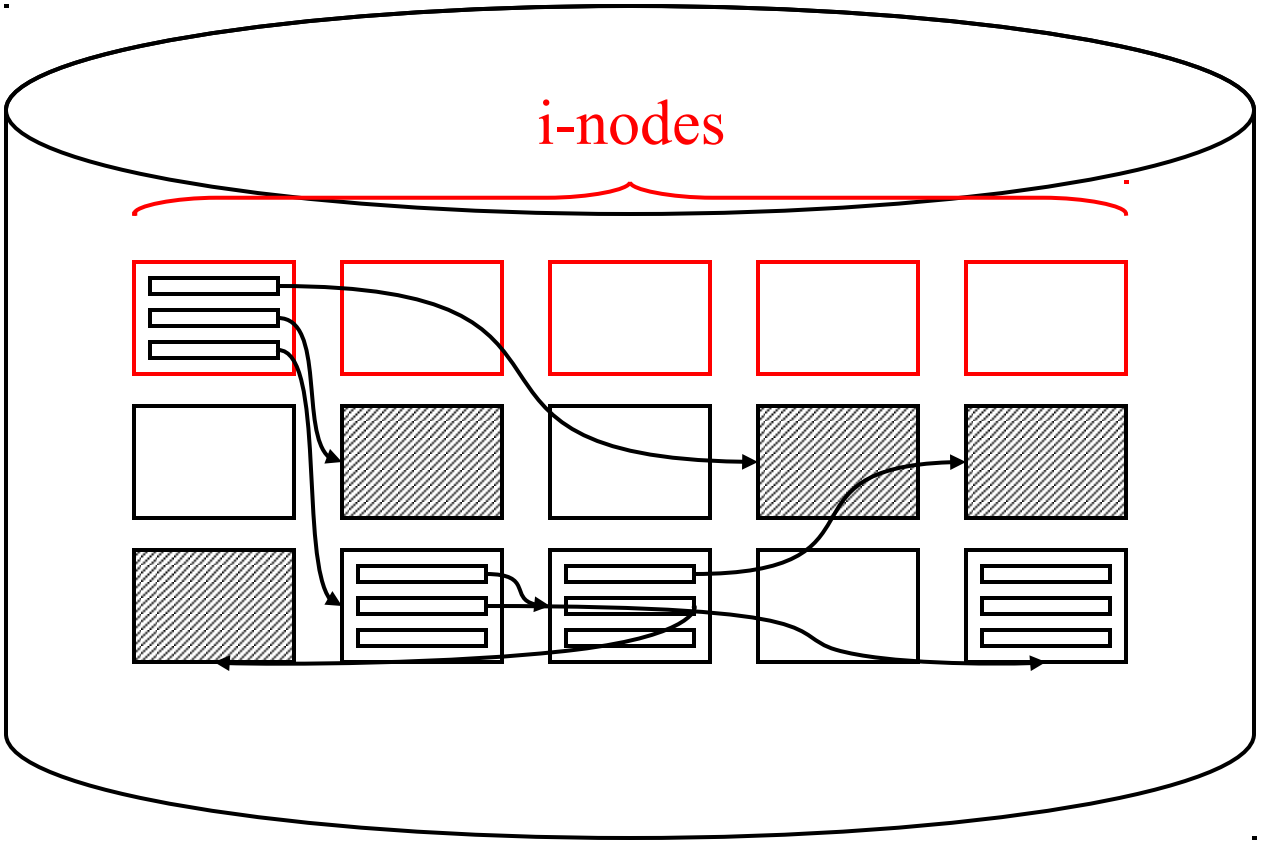
What's the problem with this mapping function?
What's the potential benefit of this mapping function?

Files vs. Disk

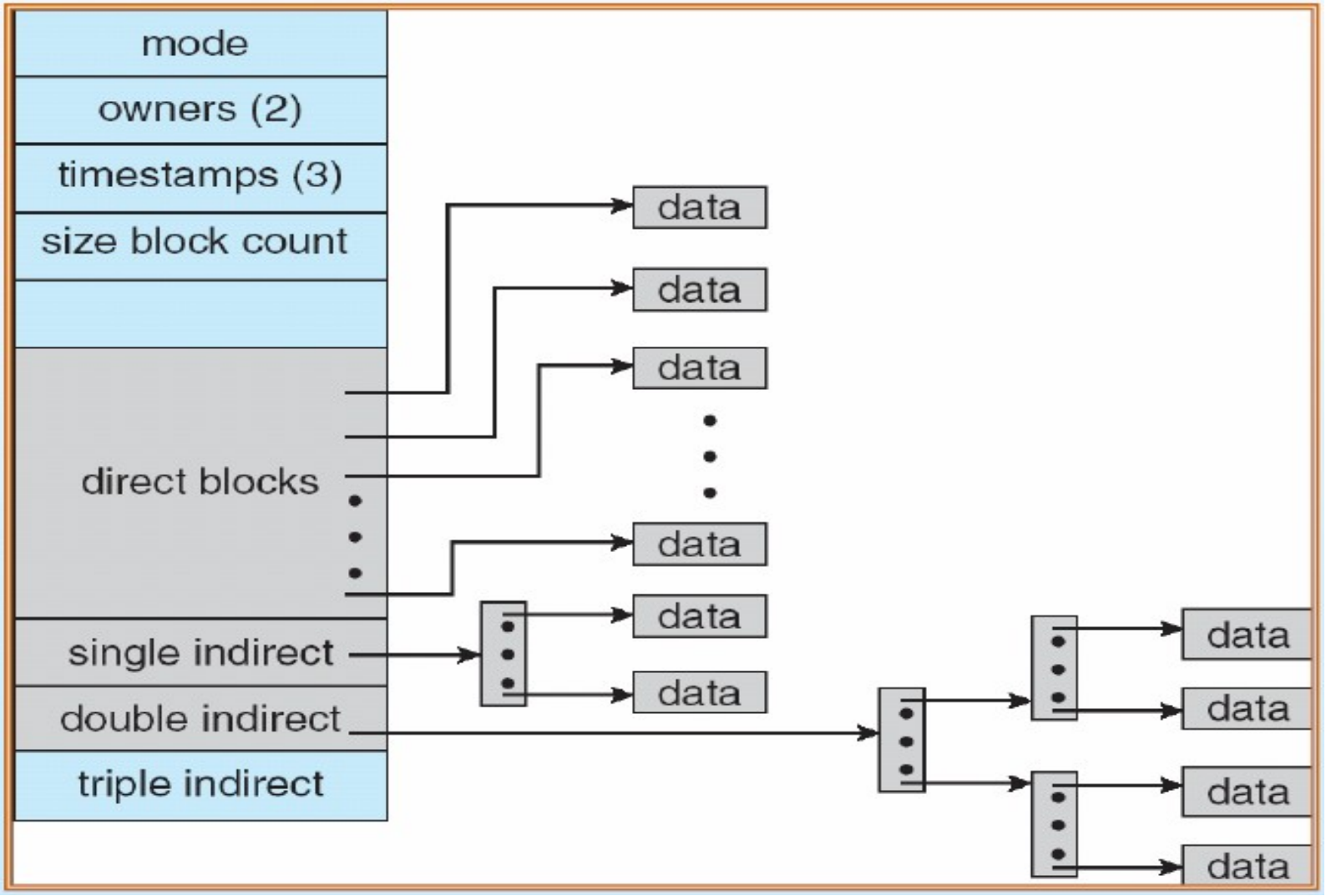


What's the problem with this mapping function?

Indexed Allocation: UNIX File



Indexed Allocation: UNIX File



De-fragmentation

Want index-based organization of disk blocks of a file for efficient random access and no fragmentation

Want sequential layout of disk blocks for efficient sequential access

How to reconcile?

De-fragmentation (cont'd)

Base structure is index-based

Optimize for sequential access

De-fragmentation: move blocks around to get sequential layout of files

Group allocation of blocks: group tracks together (cylinders). Try to allocate all blocks of a file from a single cylinder group so that they are close together. This style of grouped allocation was first proposed for the BSD Fast File System and later incorporated in ext2 (Linux).

Extents: on each write that extends a file, allocate a chunk of consecutive blocks. Some modern systems use extents, e.g. VERITAS (supported in many systems like Linux and Solaris), the first commercial journaling file system. Ext4 can use them also (extents are not the default option, though).

Free Space Management

No policy issues here – just mechanism

Bitmap: one bit for each block on the disk

Good to find a contiguous group of free blocks

Files are often accessed sequentially

For 1TB disk and 4KB blocks, 32MB for the bitmap

Chained free portions: pointer to the next one

Not so good for sequential access (hard to find sequential blocks of appropriate size)

Index: treats free space as a file

File System

OK, we have files

How can we name them?

How can we organize them?

File Naming

Each file has an associated human-readable name

E.g., usr, bin, mid-term.pdf, design.pdf

File name must be globally unique

Otherwise how would the system know which file we are referring to?

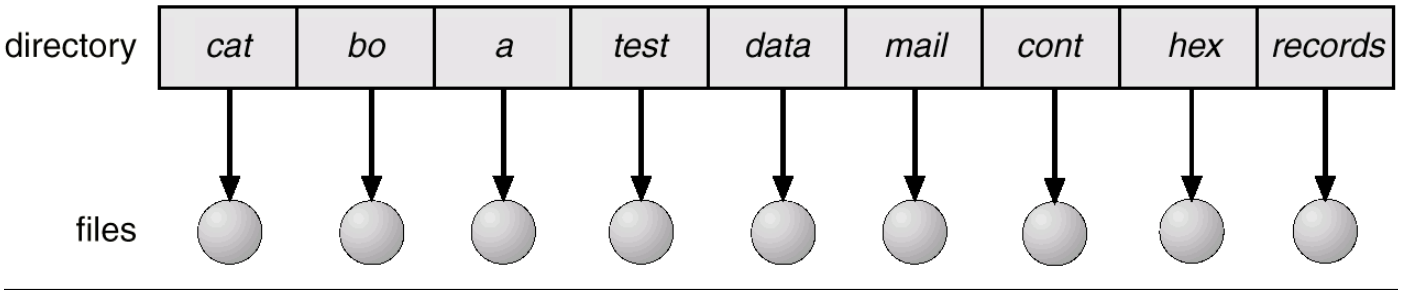
OS must maintain a mapping between a file name and the set of blocks belonging to the file

In Unix, this is a mapping between names and i-nodes

Mappings are kept in directories

Single-Level Directory

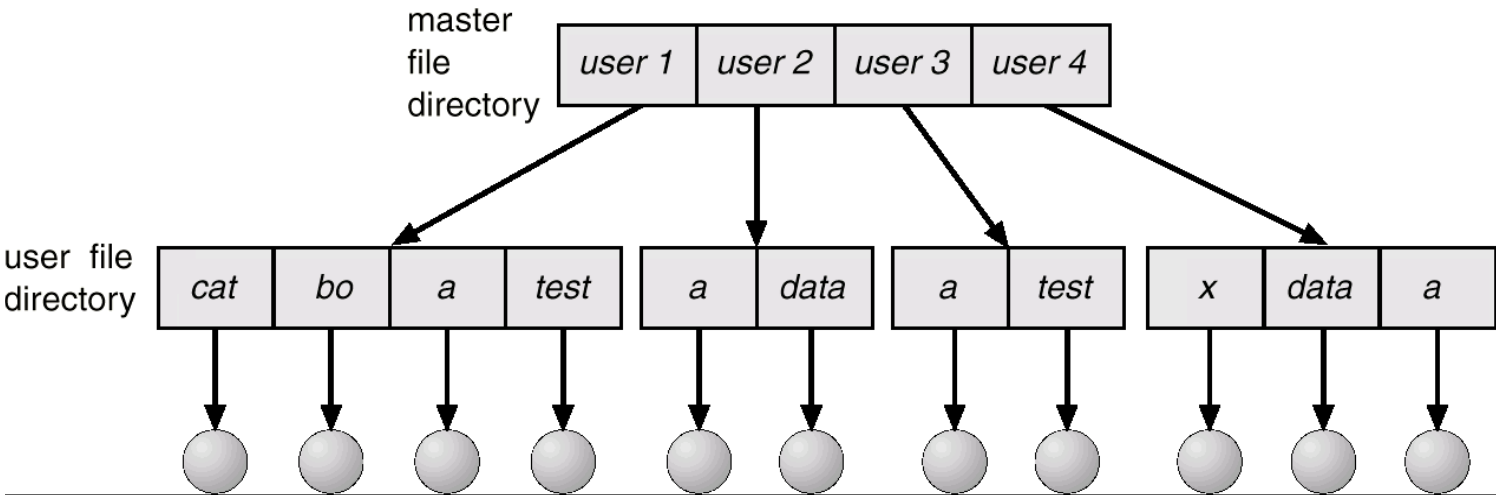
Have a directory to associate names with files



What's wrong with such a flat structure?

Two-Level Directory

Separate directory for each user

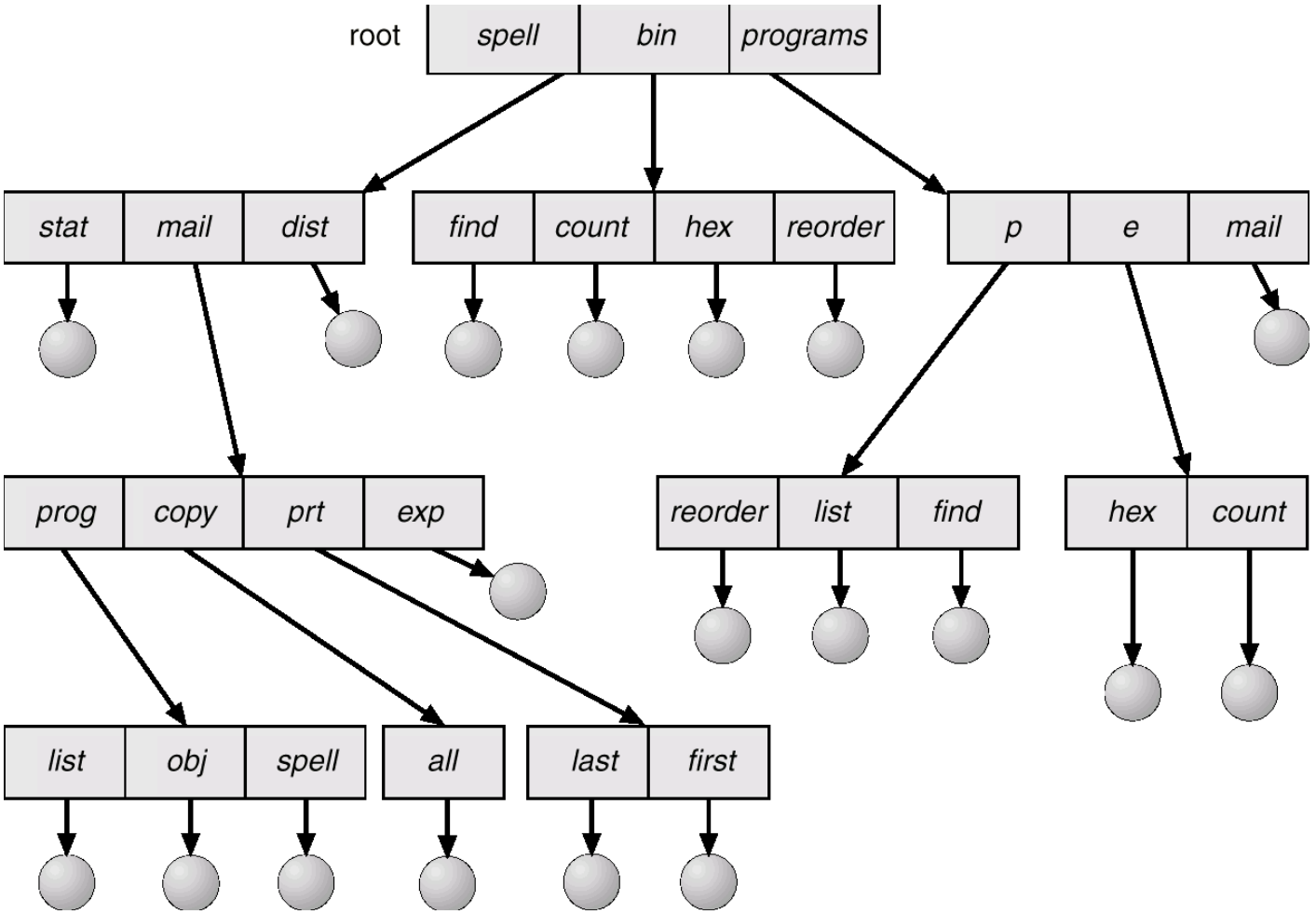


A file name is now the concatenation of the user name and the file name (with a / added between the two names)

Different users can now have files with the same file name

Is this enough?

Tree-Structured Directories



Tree-Structured Directories (Cont.)

Efficient searching

Grouping capability

Idea of current directory (working directory)

cd /spell/mail/prog

type list

Tree-Structured Directories (Cont.)

Absolute or **relative** path name

Creating a new file is done in current directory

Delete a file

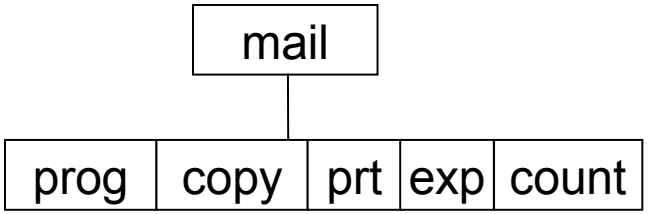
rm <file-name>

Creating a new subdirectory is done in current directory.

mkdir <dir-name>

Example: if in current directory **/mail**

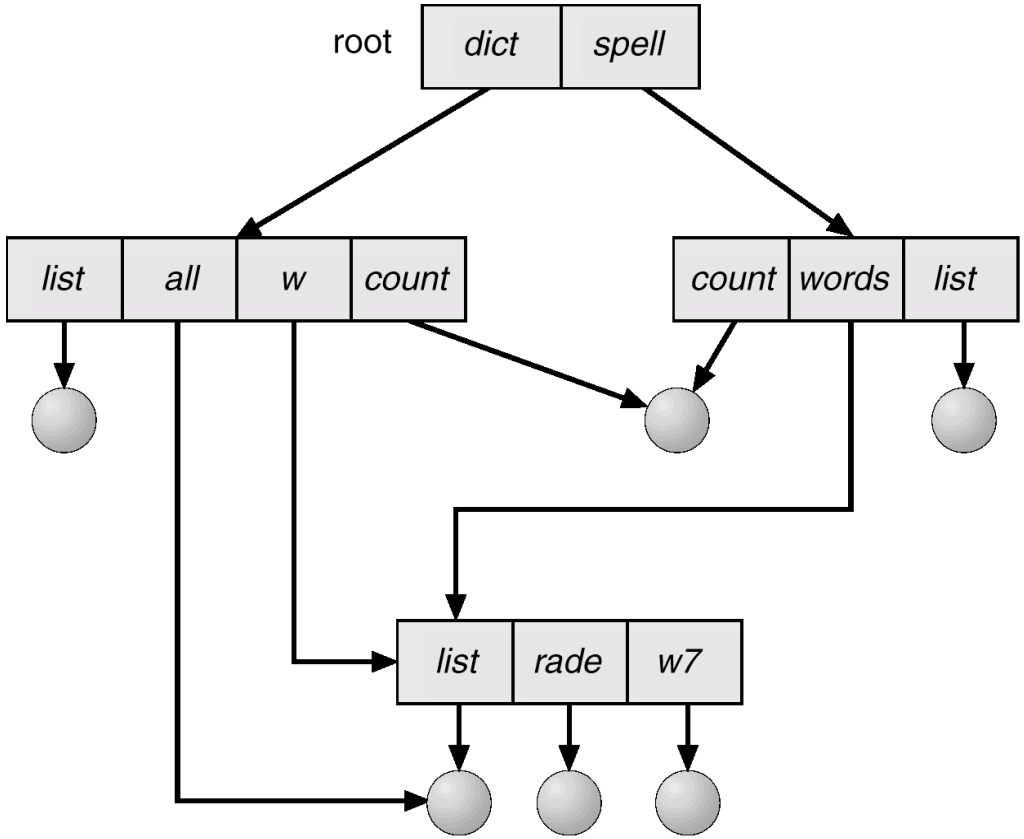
mkdir count



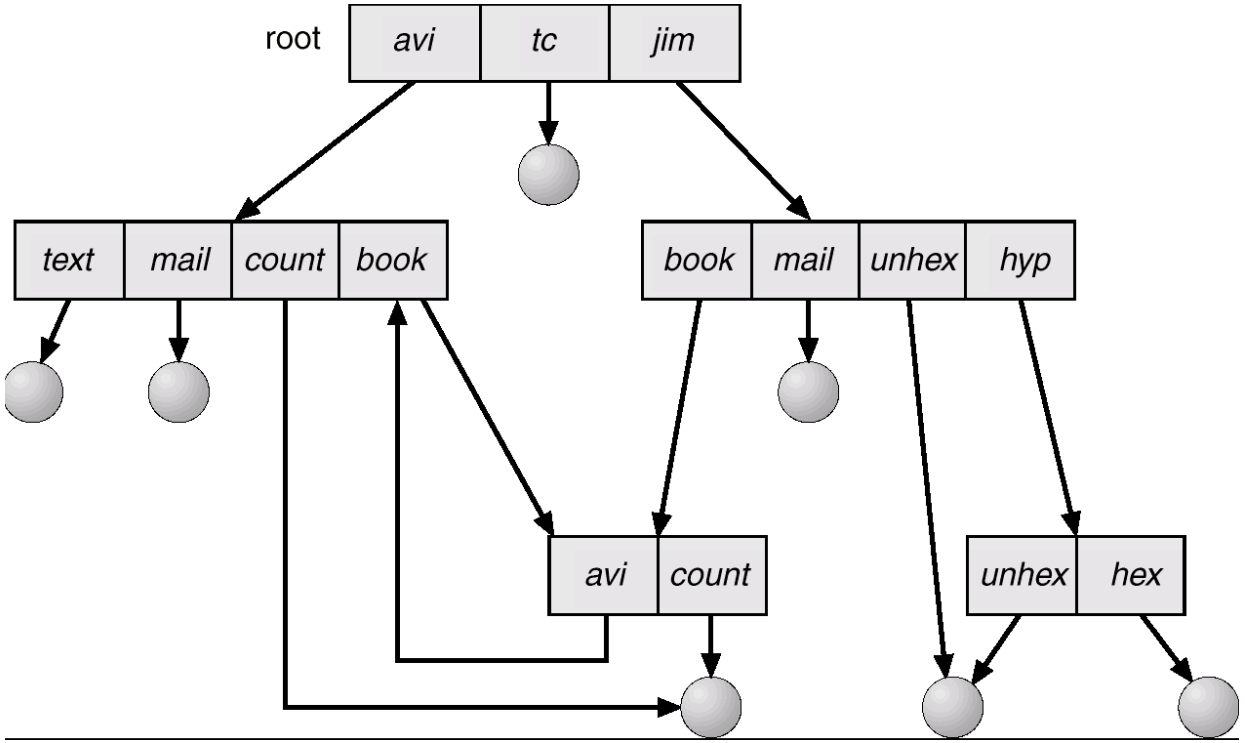
Deleting "mail" ⇒ deleting the entire subtree rooted by "mail".

Acyclic-Graph Directories

Have shared subdirectories and files



General Graph Directory



Unix File System

Ordinary files (uninterpreted)

Directories

Directory is differentiated from ordinary file by a bit in the i-node

File of files: consists of records (directory entries), each of which contains info about a file and a pointer to its i-node

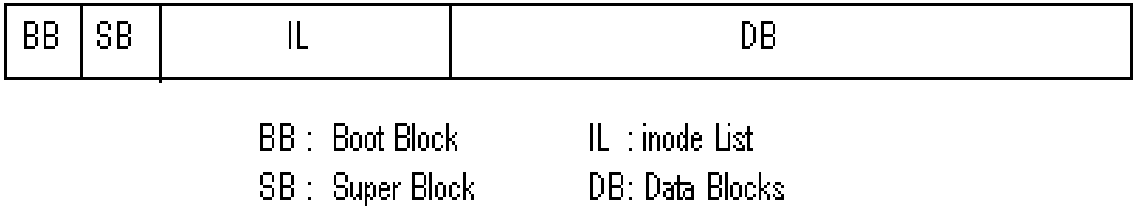
Organized as a rooted tree

Pathnames (relative and absolute)

Contains links to parent, itself

Multiple links to files can exist: hard (points to the actual file data) or symbolic (symbolic path to a hard link). Both types of links can be created with the ln utility. Removing a symbolic link does not affect the file data, whereas removing the last hard link to a file will remove the data.

Unix Storage Organization



Info stored on the SB: size of the file system, number of free blocks, list of free blocks, index to the next free block, size of the I-node list, number of free I-nodes, list of free I-nodes, index to the next free I-node, locks for free block and free I-node lists, and flag to indicate a modification to the SB

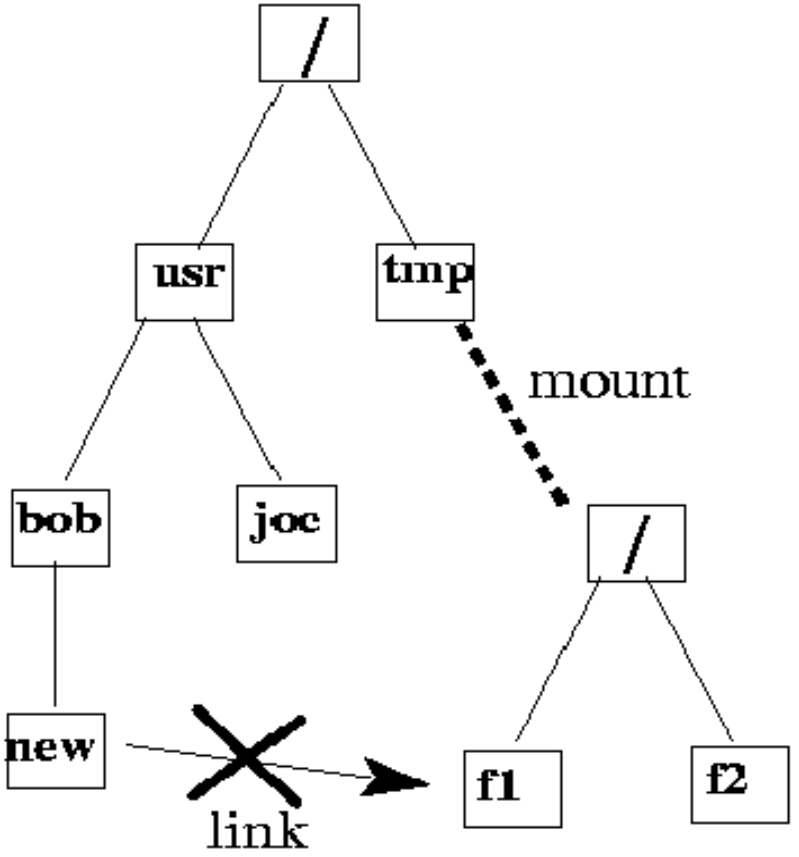
I-node contains: owner, type (directory, file, device), last modified time, last accessed time, last I-node modified time, access permissions, number of links to the file, size, and block pointers

Unix File Systems (Cont'd)

Tree-structured file hierarchies

Mounted on existing space by using mount

No hard links between different file systems



Name Space

In UNIX, "devices are files"

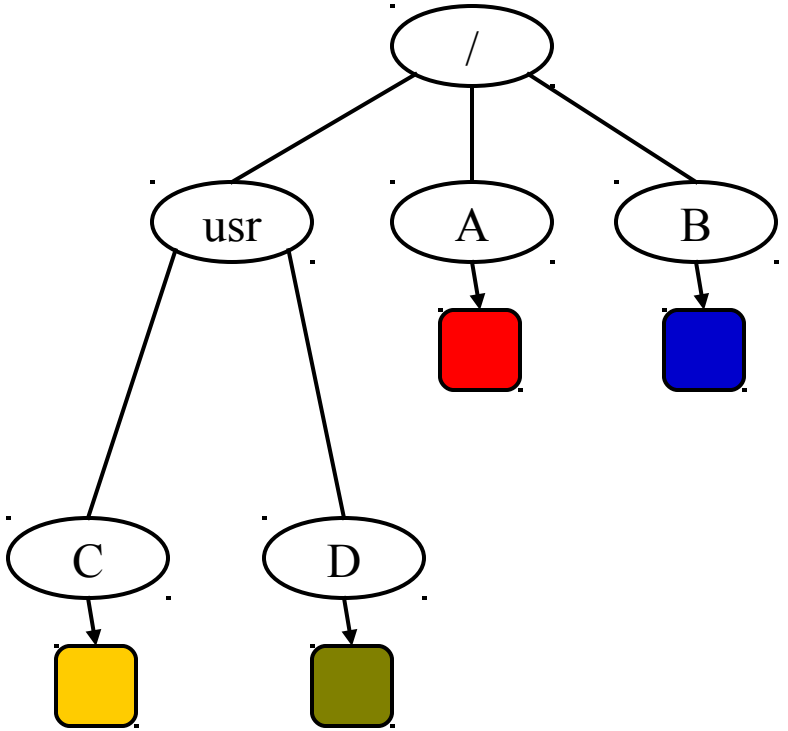
E.g., /dev/cdrom, /dev/tape

User process accesses devices by accessing corresponding file

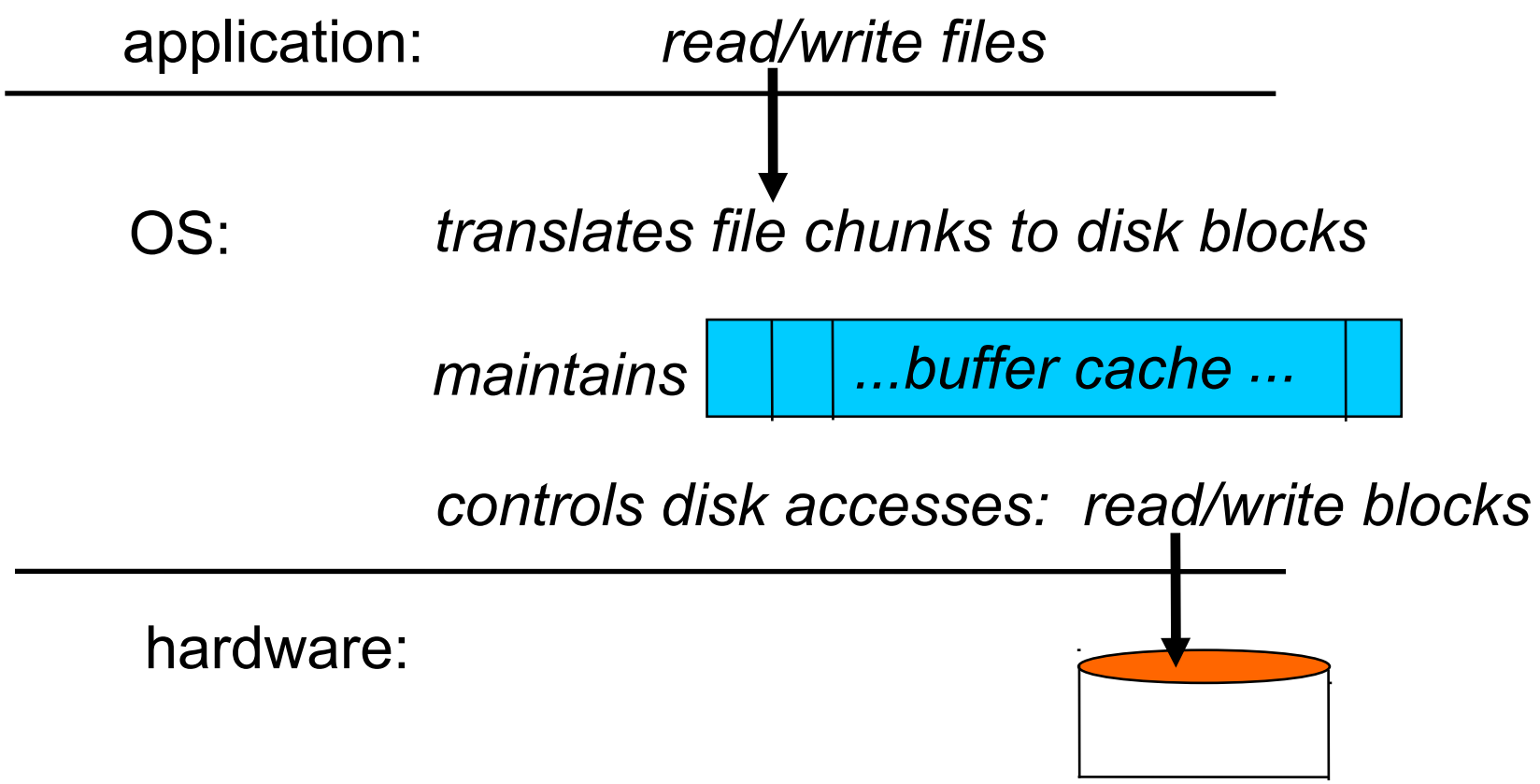
A name space

Name ↔ object

Objects may support same API (as in Unix) or different APIs (object-oriented systems)



File System Buffer Cache



Any problems?

File System Buffer Cache

Disks are "stable" while memory is volatile

What happens if you buffer a write and the machine crashes before the write has been saved to disk?

Can use write-through but write performance will suffer

In UNIX

Use unbuffered I/O when writing i-nodes or pointer blocks

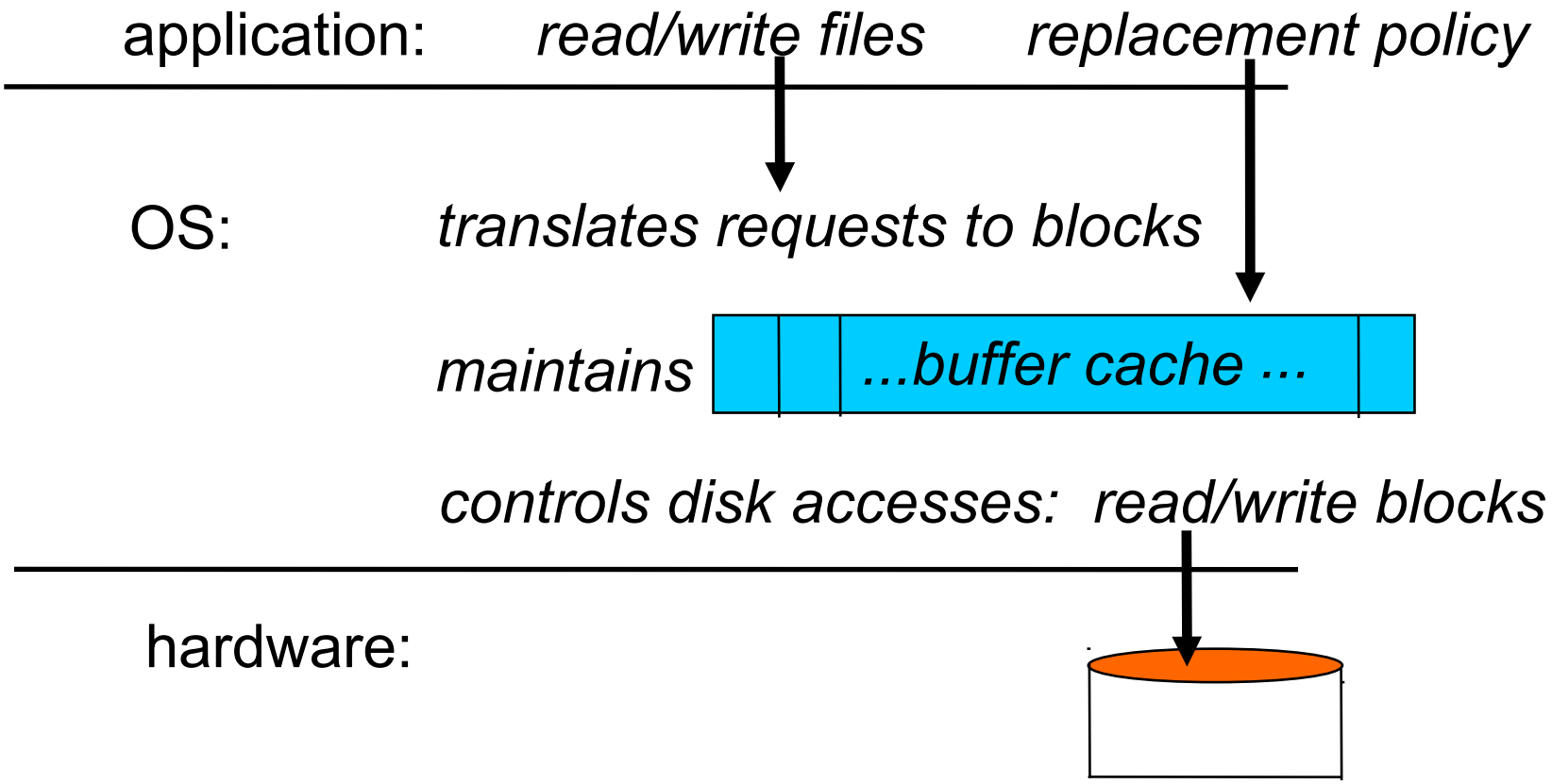
Use buffered I/O for other writes and force sync every 30 seconds

Will talk more about this in a few slides

What about replacement?

How can we further improve performance?

Application-Controlled Caching



File Sharing and Consistency

Can multiple processes open the same file at the same time?

What happens if two or more processes write to the same file?

What happens if two or more processes try to create the same file at the same time?

What happens if a process deletes a file when another has it opened?

File Sharing and Consistency (Cont'd)

Several possibilities for file sharing semantics

Unix semantics: file associated with single physical image

- Writes by one user are seen immediately by others who also have the file open.
- One sharing mode allows file pointer to be shared.

Session semantics (AFS): file may be associated temporarily with several images at the same time

- Writes by one user are not immediately seen by others who also have the file open.
- Once a file is closed, the changes made to it are visible only in sessions starting later.

Immutable-file semantics: file declared as shared cannot be written.

File System Consistency on Crashes

File system almost always uses a buffer/disk cache for performance reasons

Two copies of a disk block (buffer cache, disk) → consistency problem if the system crashes before all the modified blocks are written back to disk

This problem is critical especially for the blocks that contain control information: i-node, free-list, directory blocks

Utility programs for checking block and directory consistency

Write back critical blocks from the buffer cache to disk immediately

Data blocks are also written back periodically: sync

More on File System Consistency

To maintain file system consistency, the ordering of updates from buffer cache to disk is critical

Writing to a file may involve updating several pieces of metadata. For example, extending a file requires updates to the directory entry (file size, last access), to the i-node (extra block), and to the free list (one fewer block free).

Example problem: if the directory block is written back before the i-node and the system crashes, the directory structure will be inconsistent

Similar case when i-node and free list are updated

A more elaborate solution: use dependencies between blocks containing control data in the buffer cache to specify the ordering of updates

More on File System Consistency

Even with a pre-specified ordering of metadata updates, it might be impossible to re-establish consistency after a crash.

Hence, another solution: Journaling (e.g., ext3 for Linux)

How does it work? OS writes metadata updates synchronously to a log and returns control to user process. In the background, the log is replayed with transaction semantics. When a set of related operations (i.e., a transaction) is performed across the actual file system, they are deleted from the log. On a crash, any incomplete transactions are rolled back.

Side advantage: metadata updates perform quickly because log is sequential.

Protection Mechanisms

Files are OS objects (like other resources, such as a printer): they have unique names and a finite set of operations that processes can perform on them

Protection domain defines a set of {object,rights} where right is the permission to perform one of the operations on the object

At every instant, each process runs in some protection domain

In Unix, a protection domain is {uid, gid}

Protection domain in Unix is switched when running a program with SETUID/SETGID set or when the process enters the kernel mode by issuing a system call

How to store the info about all the protection domains?

Protection Mechanisms (cont'd)

Access Control List (ACL): associate with each object a list of all the protection domains that may access the object and how.

In Unix, an ACL defines three protection domains: owner, group and others

Capability List (C-list): associate with each process a list of objects that may be accessed along with the operations

C-list implementation issues: where/how to store them (hardware, kernel, encrypted in user space) and how to revoke them

Most systems use a combination of ACLs and C-Lists. Example: In Unix, an ACL is checked when first opening a file. After that, system relies on kernel information (per-process file table) that is established during the open call. This obviates the need for further protection checks.

Research in FSs: An Energy-Aware File System

Large file/storage servers are pretty common these days.

For these servers (and the data centers where they reside), power and energy are serious problems. Power affects installation and cooling investments. Energy affects electricity costs.

Given the replication of resources in these servers, can conserve energy by turning resources off, just like in laptops or other battery-operated devices.

Can you think of how to do this at the file- or storage-system level?

Leveraging Redundancy

Idea 1: segregate "original" and "redundant" files/blocks onto different sets of disks.

At each server, can turn off disks that store redundant data under light and moderate loads. Extrapolate to entire servers, so that whole nodes can be turned off.

Somehow keep logs of writes so that redundant disks can be updated when they are turned back on.

Problems: keeping write logs, deciding when to turn on the redundant disks, etc.

Leveraging Popularity

Idea 2: segregate popular and unpopular files/blocks, accepting some performance degradation for the latter.

At each server, can turn off disks that contain unpopular files/blocks. Extrapolate to all servers, so that whole nodes can be turned off.

Implement file/block migration and/or replication to adjust to variations in popularity.

Problems: provisioning, determining popularity, policies for migration and replication, etc.