

# File Systems: Allocation Issues, Naming, and Performance

CS 111

Operating Systems

Peter Reiher

# Outline

- Allocating and managing file system free space
- File naming and directories
- File volumes
- File system performance issues

# Free Space and Allocation Issues

- How do I keep track of a file system's free space?
- How do I allocate new disk blocks when needed?
  - And how do I handle deallocation?

# The Allocation/Deallocation Problem

- File systems usually aren't static
- You create and destroy files
- You change the contents of files
  - Sometimes extending their length in the process
- Such changes convert unused disk blocks to used blocks (or visa versa)
- Need correct, efficient ways to do that
- Typically implies a need to maintain a free list of unused disk blocks

# Creating a New File

- Allocate a free file control block
  - For UNIX
    - Search the super-block free I-node list
    - Take the first free I-node
  - For DOS
    - Search the parent directory for an unused directory entry
- Initialize the new file control block
  - With file type, protection, ownership, ...
- Give new file a name
  - Naming issues will be discussed in the next lecture

# Extending a File

- Application requests new data be assigned to a file
  - May be an explicit allocation/extension request
  - May be implicit (e.g., write to a currently non-existent block – remember sparse files?)
- Find a free chunk of space
  - Traverse the free list to find an appropriate chunk
  - Remove the chosen chunk from the free list
- Associate it with the appropriate address in the file
  - Go to appropriate place in the file or extent descriptor
  - Update it to point to the newly allocated chunk

# Deleting a File

- Release all the space that is allocated to the file
  - For UNIX, return each block to the free block list
  - DOS does not free space
    - It uses garbage collection
    - So it will search out deallocated blocks and add them to the free list at some future time
- Deallocate the file control lock
  - For UNIX, zero inode and return it to free list
  - For DOS, zero the first byte of the name in the parent directory
    - Indicating that the directory entry is no longer in use

# Free Space Maintenance

- File system manager manages the free space
- Getting/releasing blocks should be fast operations
  - They are extremely frequent
  - We'd like to avoid doing I/O as much as possible
- Unlike memory, it matters what block we choose
  - Best to allocate new space in same cylinder as file's existing space
  - User may ask for contiguous storage
- Free-list organization must address both concerns
  - Speed of allocation and deallocation
  - Ability to allocate contiguous or near-by space

# DOS File System Free Space Management

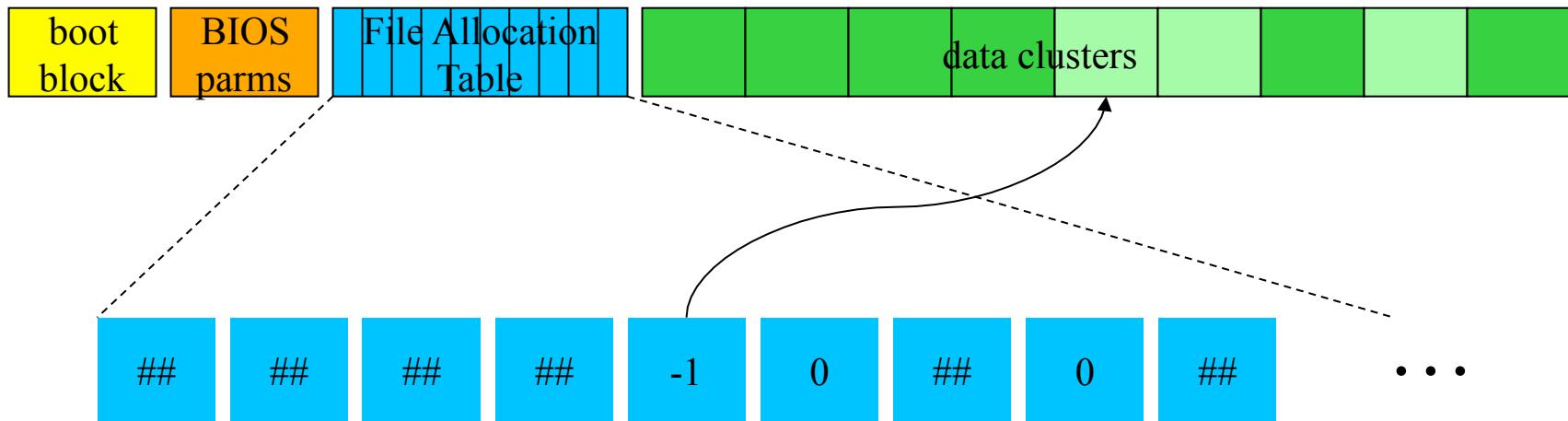
- Search for free clusters in desired cylinder
  - We can map clusters to cylinders
    - The BIOS Parameter Block describes the device geometry
  - Look at first cluster of file to choose the desired cylinder
  - Start search at first cluster of desired cylinder
  - Examine each FAT entry until we find a free one
- If no free clusters, we must garbage collect
  - Recursively search all directories for existing files
  - Enumerate all of the clusters in each file
  - Any clusters not found in search can be marked as free

CS 111 This won't be fast . . .

# Extending a DOS File

- Note cluster number of current last cluster in file
- Search the FAT to find a free cluster
  - Free clusters are indicated by a FAT entry of zero
  - Look for a cluster in the same cylinder as previous cluster
  - Put -1 in its FAT entry to indicate that this is the new EOF
  - This has side effect of marking the new cluster as “not free”
- Chain new cluster on to end of the file
  - Put the number of new cluster into FAT entry for last cluster

# DOS Free Space



Each FAT entry corresponds to a cluster, and contains the number of the next cluster.

A value of zero indicates a cluster that is not allocated to any file, and is therefore free.

# The BSD File System

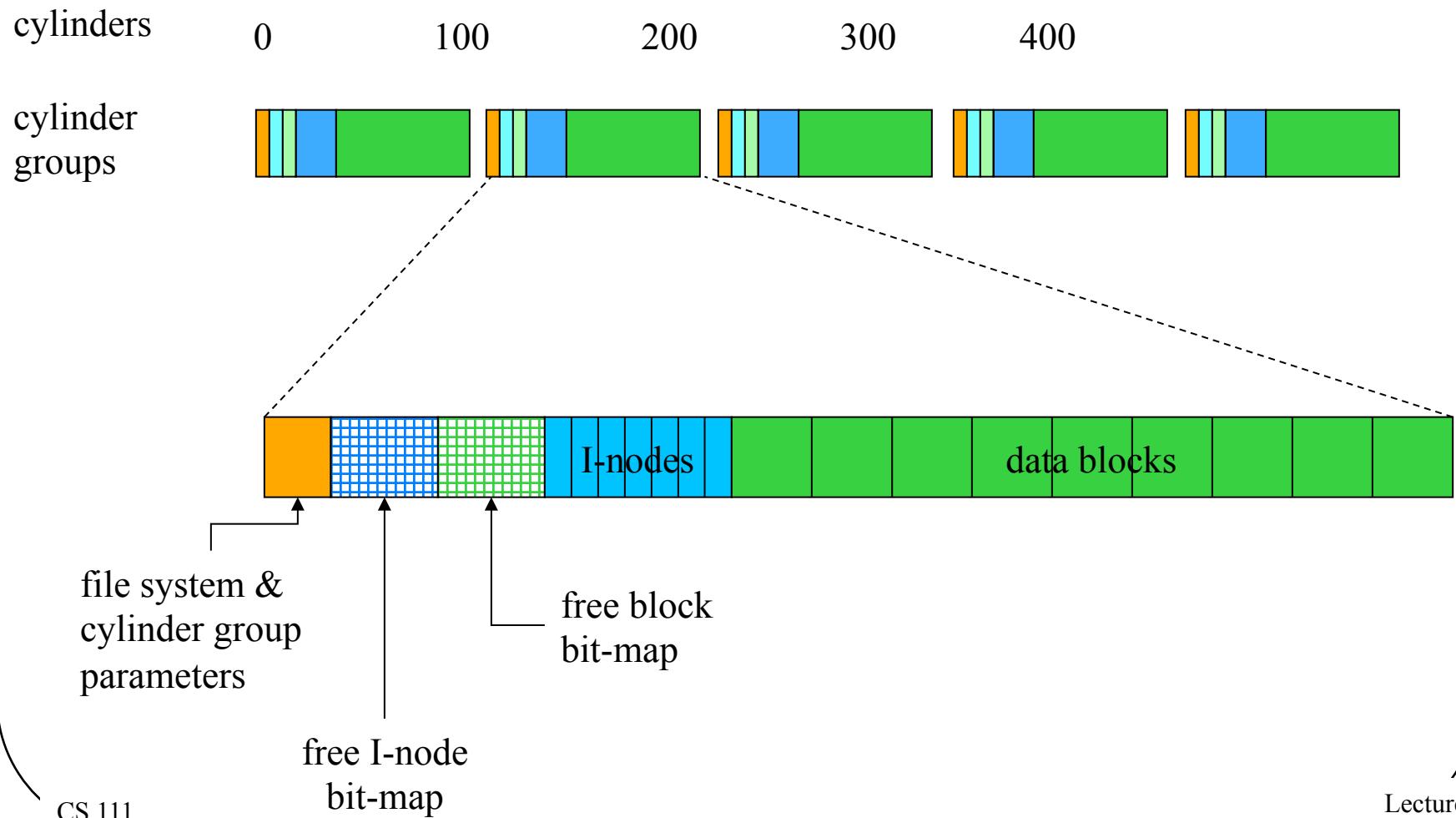
## Free Space Management

- BSD is another version of Unix
- The details of its inodes are similar to those of Unix System V
  - As previously discussed
- Other aspects are somewhat different
  - Including free space management
  - Typically more advanced
- Uses bit map approach to managing free space
  - Keeping cylinder issues in mind

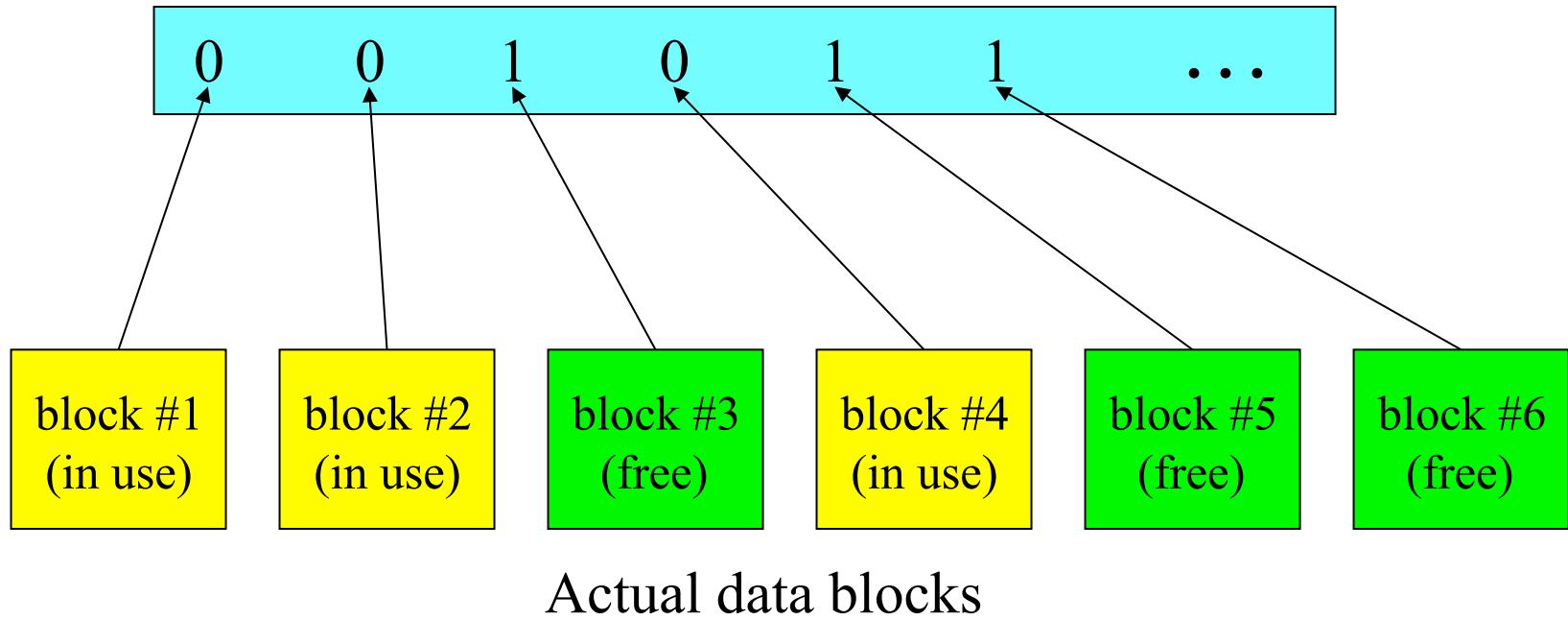
# The BSD Approach

- Instead of all control information at start of disk,
- Divide file system into cylinder groups
  - Each cylinder group has its own control information
    - The *cylinder group summary*
  - Active cylinder group summaries are kept in memory
  - Each cylinder group has its own inodes and blocks
  - Free block list is a bit-map in cylinder group summary
- Enables significant reductions in head motion
  - Data blocks in file can be allocated in same cylinder
  - Inode and its data blocks in same cylinder group
  - Directories and their files in same cylinder group

# BSD Cylinder Groups and Free Space



# Bit Map Free Lists



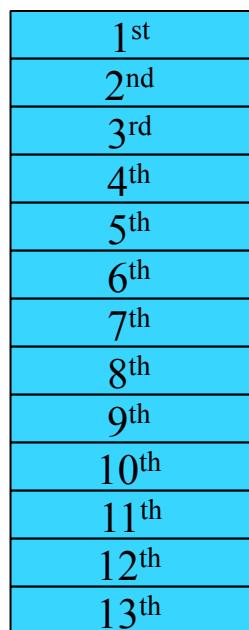
BSD Unix file systems use bit-maps to keep track of both free blocks and free I-nodes in each cylinder group

# Extending a BSD/Unix File

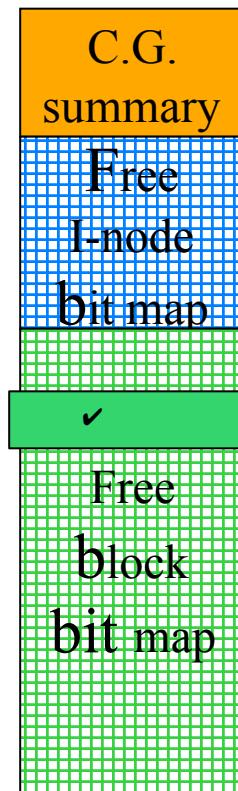
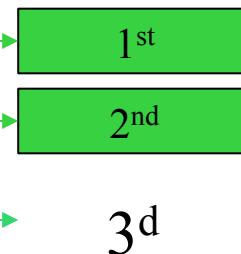
- Determine the cylinder group for the file's inode
  - Calculated from the inode's identifying number
- Find the cylinder for the previous block in the file
- Find a free block in the desired cylinder
  - Search the free-block bit-map for a free block in the right cylinder
  - Update the bit-map to show the block has been allocated
- Update the inode to point to the new block
  - Go to appropriate block pointer in inode/indirect block
  - If new indirect block is needed, allocate/assign it first
  - Update inode/indirect to point to new block

# Unix File Extension

block pointers  
(in I-node)



1. Determine cylinder group and get its information
2. Consult the cylinder group free block bit map
3. Allocate the block to the file to find a good block
- 3.2 Set appropriate block pointer to it
- 5.2 Update the free block bit map



# Naming in File Systems

- Each file needs some kind of handle to allow us to refer to it
- Low level names (like inode numbers) aren't usable by people or even programs
- We need a better way to name our files
  - User friendly
  - Allowing for easy organization of large numbers of files
  - Readily realizable in file systems

# File Names and Binding

- File system knows files by descriptor structures
- We must provide more useful names for users
- The file system must handle name-to-file mapping
  - Associating names with new files
  - Finding the underlying representation for a given name
  - Changing names associated with existing files
  - Allowing users to organize files using names
- *Name spaces* – the total collection of all names known by some naming mechanism
  - Sometimes all names that *could* be created by the mechanism

# Name Space Structure

- There are many ways to structure a name space
  - Flat name spaces
    - All names exist in a single level
  - Hierarchical name spaces
    - A graph approach
    - Can be a strict tree
    - Or a more general graph (usually directed)
- Are all files on the machine under the same name structure?
- Or are there several independent name spaces?

# Some Issues in Name Space Structure

- How many files can have the same name?
  - One per file system ... flat name spaces
  - One per directory ... hierarchical name spaces
- How many different names can one file have?
  - A single “true name”
  - Only one “true name”, but aliases are allowed
  - Arbitrarily many
  - What’s different about “true names”?
- Do different names have different characteristics?
  - Does deleting one name make others disappear too?
- Do all names see the same access permissions?

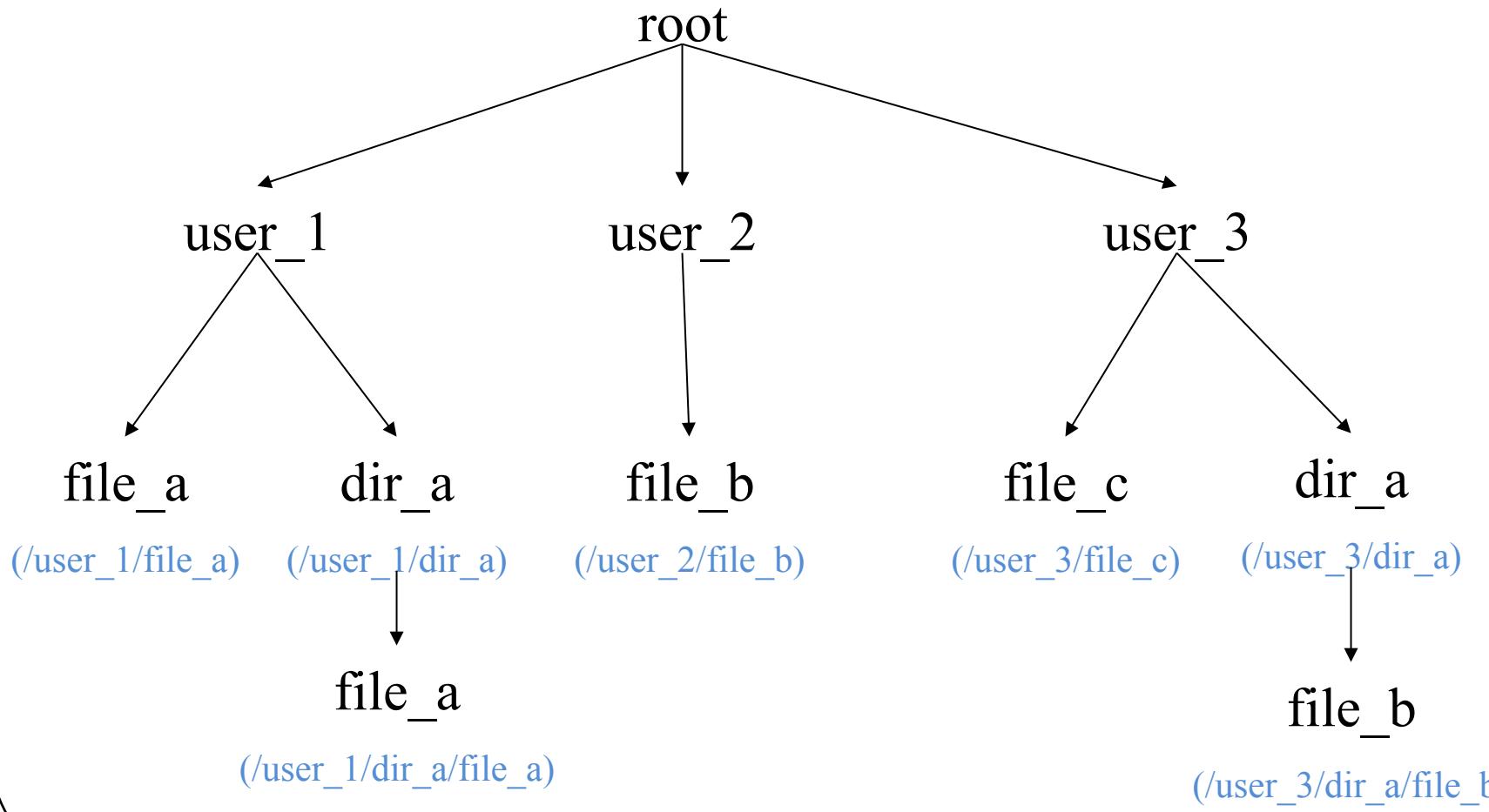
# Flat Name Spaces

- There is one naming context per file system
  - All file names must be unique within that context
- All files have exactly one true name
  - These names are probably very long
- File names may have some structure
  - E.g., CAC101CS111SECTION1SLIDESLECTURE3
  - This structure may be used to optimize searches
  - The structure is very useful to users
  - But the structure has no meaning to the file system
- No longer a widely used approach

# Hierarchical Name Spaces

- Essentially a graphical organization
- Typically organized using directories
  - A file containing references to other files
  - A non-leaf node in the graph
  - It can be used as a naming context
    - Each process has a *current directory*
    - File names are interpreted relative to that directory
- Nested directories can form a tree
  - A file name describes a path through that tree
  - The directory tree expands from a “root” node
    - A name beginning from root is called “fully qualified”
  - May actually form a directed graph
    - If files are allowed to have multiple names

# A Rooted Directory Tree



# Directories Are Files

- Directories are a special type of file
  - Used by OS to map file names into the associated files
- A directory contains multiple directory entries
  - Each directory entry describes one file and its name
- User applications are allowed to read directories
  - To get information about each file
  - To find out what files exist
- Usually only the OS is allowed to write them
  - Users can cause writes through special system calls
  - The file system depends on the integrity of directories

# Traversing the Directory Tree

- Some entries in directories point to child directories
  - Describing a lower level in the hierarchy
- To name a file at that level, name the parent directory and the child directory, then the file
  - With some kind of delimiter separating the file name components
- Moving up the hierarchy is often useful
  - Directories usually have special entry for parent
  - Many file systems use the name “..” for that

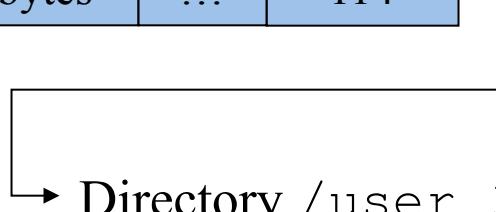
# Example: The DOS File System

- File & directory names separated by back-slashes
  - E.g., \user\_3\dir\_a\file\_b
- Directory entries are the file descriptors
  - As such, only one entry can refer to a particular file
- Contents of a DOS directory entry
  - Name (relative to this directory)
  - Type (ordinary file, directory, ...)
  - Location of first cluster of file
  - Length of file in bytes
  - Other privacy and protection attributes

# DOS File System Directories

Root directory, starting in cluster #1

file name	type	length	...	1 <sup>st</sup> cluster
user_1	DIR	256 bytes	...	9
user_2	DIR	512 bytes	...	31
user_3	DIR	284 bytes	...	114



→ Directory /user\_3, starting in cluster #114

file name	type	length	...	1 <sup>st</sup> cluster
..	DIR	256 bytes	...	1
dir_a	DIR	512 bytes	...	62
file_c	FILE	1824 bytes	...	102

# File Names Vs. Path Names

- In some flat name space systems files had “true names”
  - Only one possible name for a file,
  - Kept in a record somewhere
- In DOS, a file is described by a directory entry
  - Local name is specified in that directory entry
  - Fully qualified name is the path to that directory entry
    - E.g., start from root, to user\_3, to dir\_a, to file\_b
  - But DOS files still have only one name
- What if files had no intrinsic names of their own?
  - All names came from directory paths

# Example: Unix Directories

- A file system that allows multiple file names
  - So there is no single “true” file name, unlike DOS
- File names separated by slashes
  - E.g., /user\_3/dir\_a/file\_b
- The actual file descriptors are the inodes
  - Directory entries only point to inodes
  - Association of a name with an inode is called a *hard link*
  - Multiple directory entries can point to the same inode
- Contents of a Unix directory entry
  - Name (relative to this directory)
  - Pointer to the inode of the associated file

# Unix Directories

But what's this  
“.” entry?

It's a directory  
entry that  
points to the  
directory  
itself!

We'll see why  
that's useful

Directory /user/~~late~~<sup>ter</sup>, inode #114

inode #	file name
114	.
1	..
194	dir_a
307	file_c

Root directory, inode #1  
inode #      file name

1	.
1	..
9	user_1
31	user_2
114	user_3

Here's a “..”  
entry, pointing to  
the parent  
directory

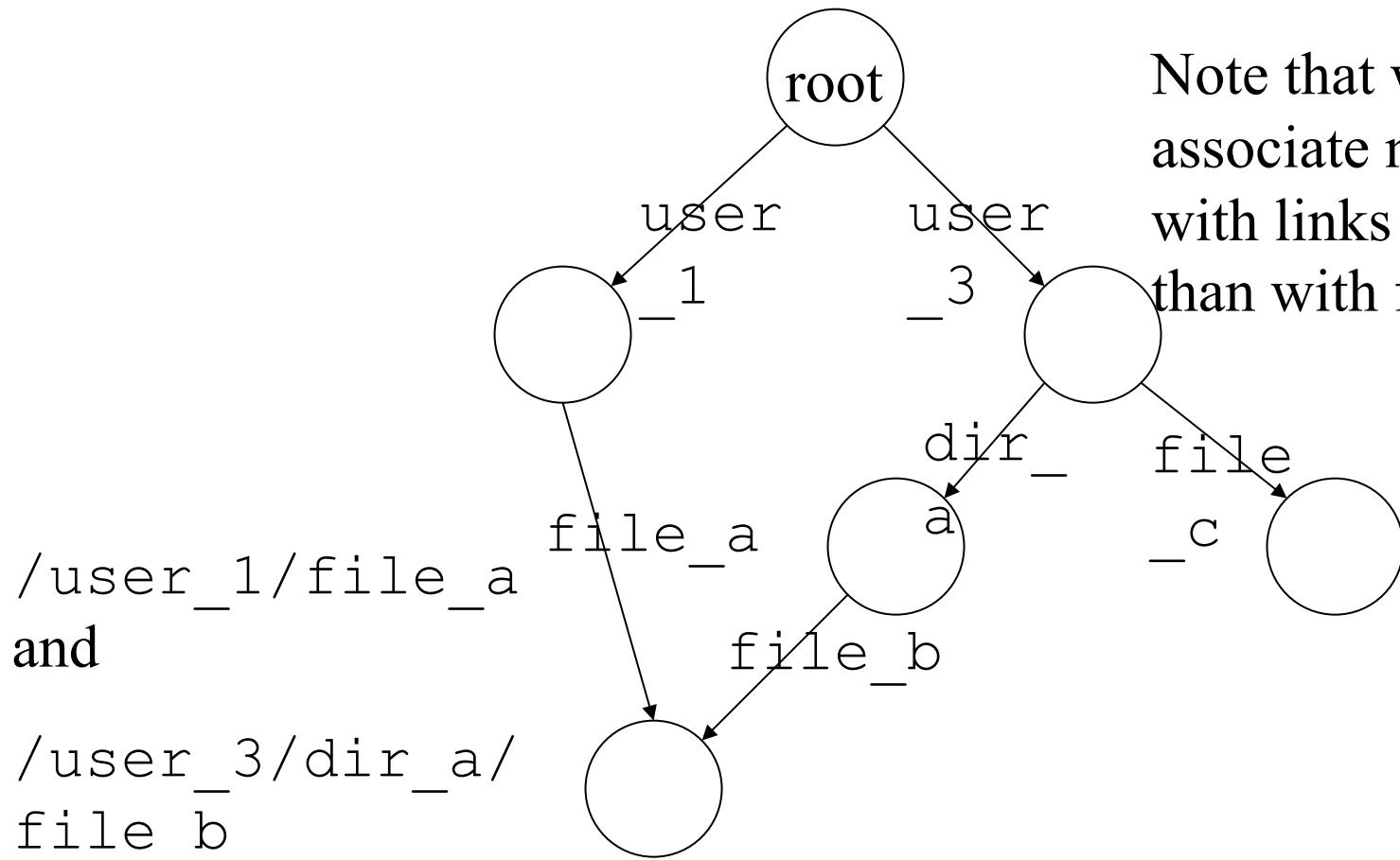
# Multiple File Names In Unix

- How do links relate to files?
  - They're the names only
- All other metadata is stored in the file inode
  - File owner sets file protection (e.g., read-only)
- All links provide the same access to the file
  - Anyone with read access to file can create new link
  - But directories are protected files too
    - Not everyone has read or search access to every directory
- All links are equal
  - There is nothing special about the first (or owner's) link

# Links and De-allocation

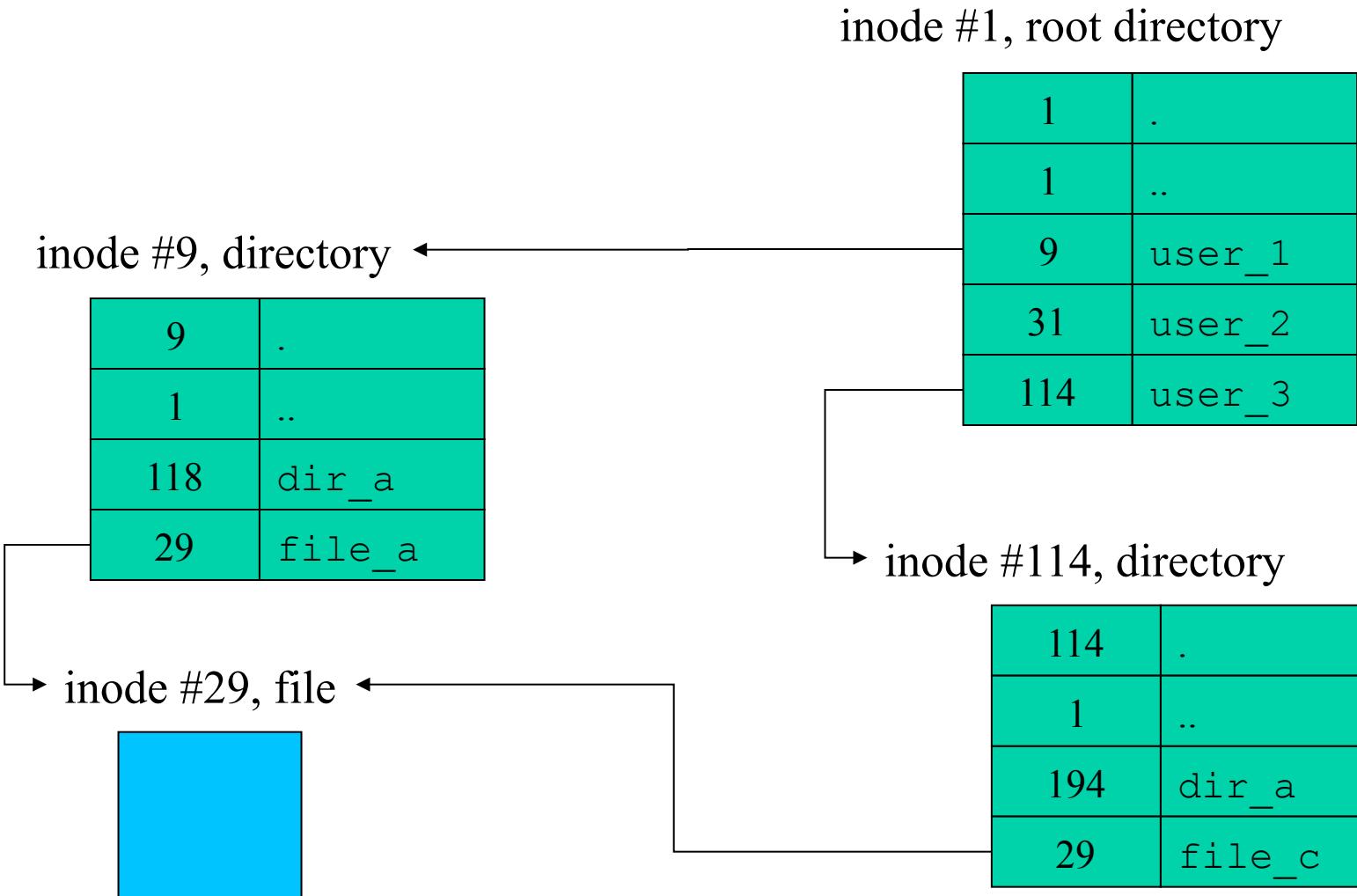
- Files exist under multiple names
- What do we do if one name is removed?
- If we also removed the file itself, what about the other names?
  - Do they now point to something non-existent?
- The Unix solution says the file exists as long as at least one name exists
- Implying we must keep and maintain a reference count of links
  - In the file inode, not in a directory

# Unix Hard Link Example



CS 141  
Fall 2015  
are both links to the  
same inode

# Hard Links, Directories, and Files

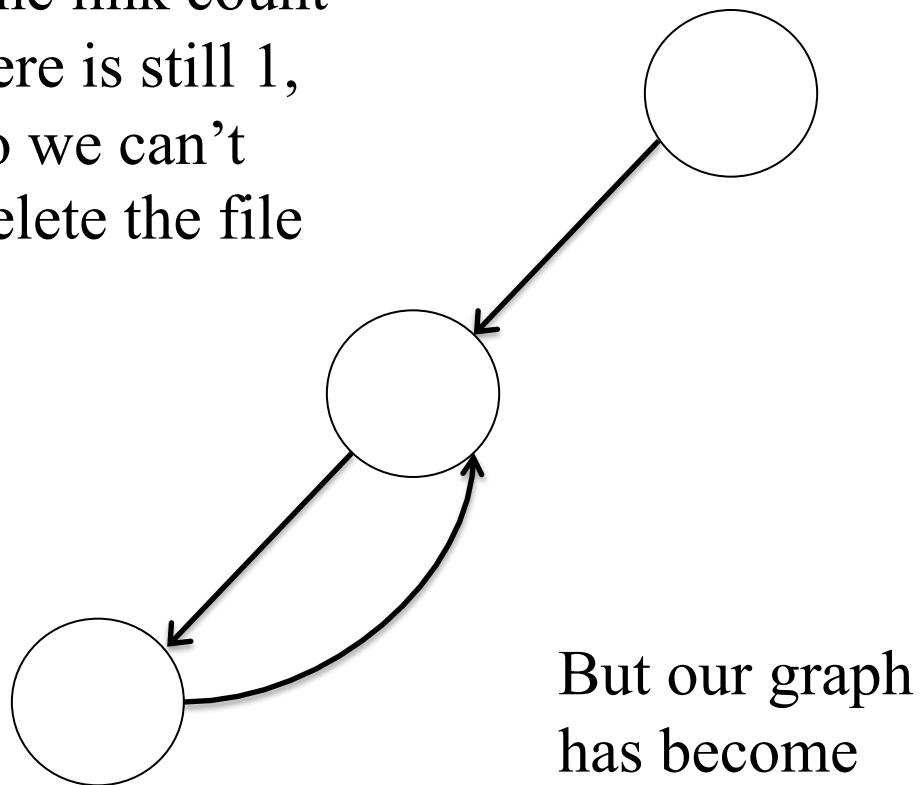


# A Potential Problem With Hard Links

- Hard links are essentially edges in the graph
- Those edges can lead backwards to other graph nodes
- Might that not create cycles in the graph?
- If it does, what happens when we delete one of the links?
- Might we not disconnect the graph?

# Illustrating the Problem

The link count  
here is still 1,  
so we can't  
delete the file



But our graph  
has become  
disconnected!

Now let's add a link

And now let's  
delete a link

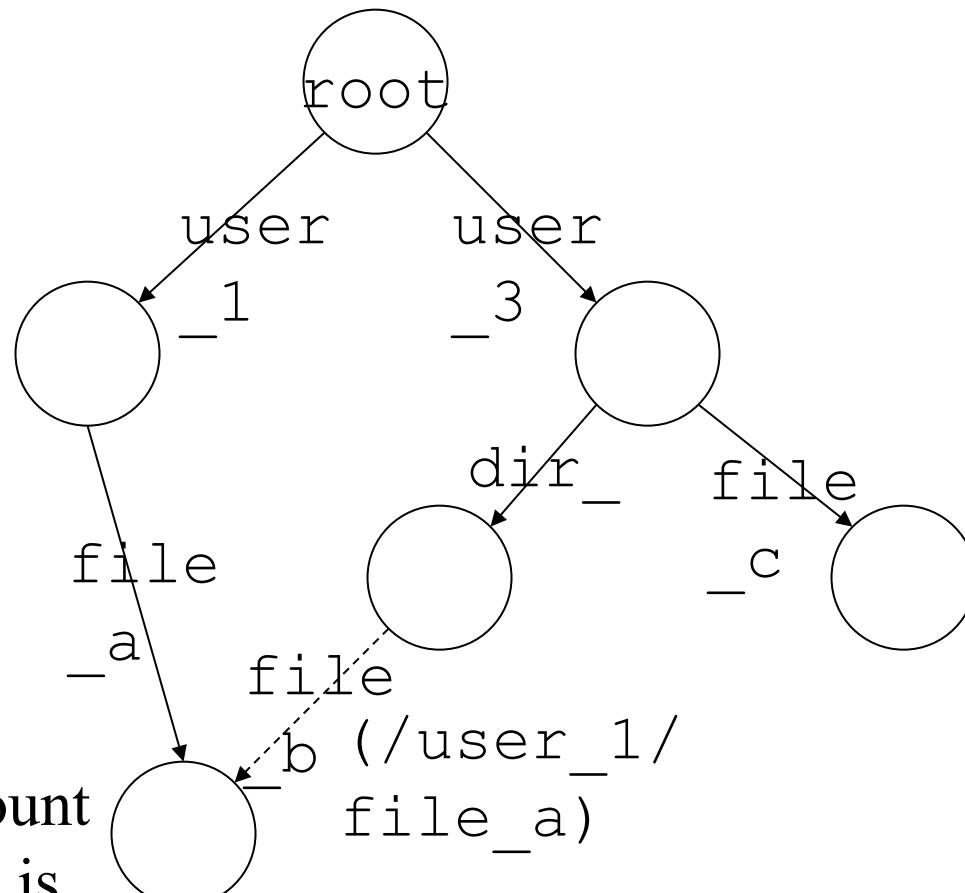
# Solving the Problem

- Only directories contain links
  - Not regular files
- So if a link can't point to a directory, there can't be a loop
- In which case, there's no problem with deletions
- This is the Unix solution: no hard links to directories
  - The “.” and “..” links are harmless exceptions

# Symbolic Links

- A different way of giving files multiple names
- Symbolic links implemented as a special type of file
  - An indirect reference to some other file
  - Contents is a path name to another file
- OS recognizes symbolic links
  - Automatically opens associated file instead
  - If file is inaccessible or non-existent, the open fails
- Symbolic link is not a reference to the inode
  - Symbolic links will not prevent deletion
  - Do not guarantee ability to follow the specified path
  - Internet URLs are similar to symbolic links

# Symbolic Link Example



The link count  
for this file is  
still 1, though

# Symbolic Links, Files, and Directories

inode #1, root directory

inode #9, directory

9	.
1	..
118	dir_a
29	file_a

inode #29, file



Link count  
still equals 1!

inode #46, symlink  
/user\_1/file\_a

1	.
1	..
9	user_1
31	user_2
114	user_3

inode #114, directory

114	.
1	..
194	dir_a
46	file_c

# What About Looping Problems?

- Do symbolic links have the potential to introduce loops into a pathname?
  - Yes, if the target of the symbolic link includes the symbolic link itself
  - Or some transitive combination of symbolic links
- How can such loops be detected?
  - Could keep a list of every inode we have visited in the interpretation of this path
  - But simpler to limit the number of directory searches allowed in the interpretation of a single path name
  - E.g., after 256 searches, just fail
  - The usual solution for Unix-style systems

# File Systems and Multiple Disks

- You can (and often do) attach more than one disk to a machine
- Would it make sense to have a single file system span the several disks?
  - Considering the kinds of disk specific information a file system keeps
  - Like cylinder information
- Usually more trouble than it's worth
  - With the exception of RAID . . .
- Instead, put separate file system on each disk
- Or several file systems on one disk

# Working With Multiple File Systems

- One machine can have multiple independent file systems
  - Each handling its own disk layout, free space, and other organizational issues
- How will the overall system work with those several file systems?
- Treat them as totally independent namespaces?
- Or somehow stitch the separate namespaces together?
- Key questions:
  1. How does an application specify which file it wants?
  2. How does the OS find that file?

# Finding Files With Multiple File Systems

- Finding files is easy if there is only one file system
  - Any file we want must be on that one file system
  - Directories enable us to name files within a file system
- What if there are multiple file systems available?
  - Somehow, we have to say which one our file is on
- How do we specify which file system to use?
  - One way or another, it must be part of the file name
  - It may be implicit (e.g., same as current directory)
  - Or explicit (e.g., every name specifies it)
  - Regardless, we need some way of specifying which file system to look into for a given file name

# Options for Naming With Multiple Partitions

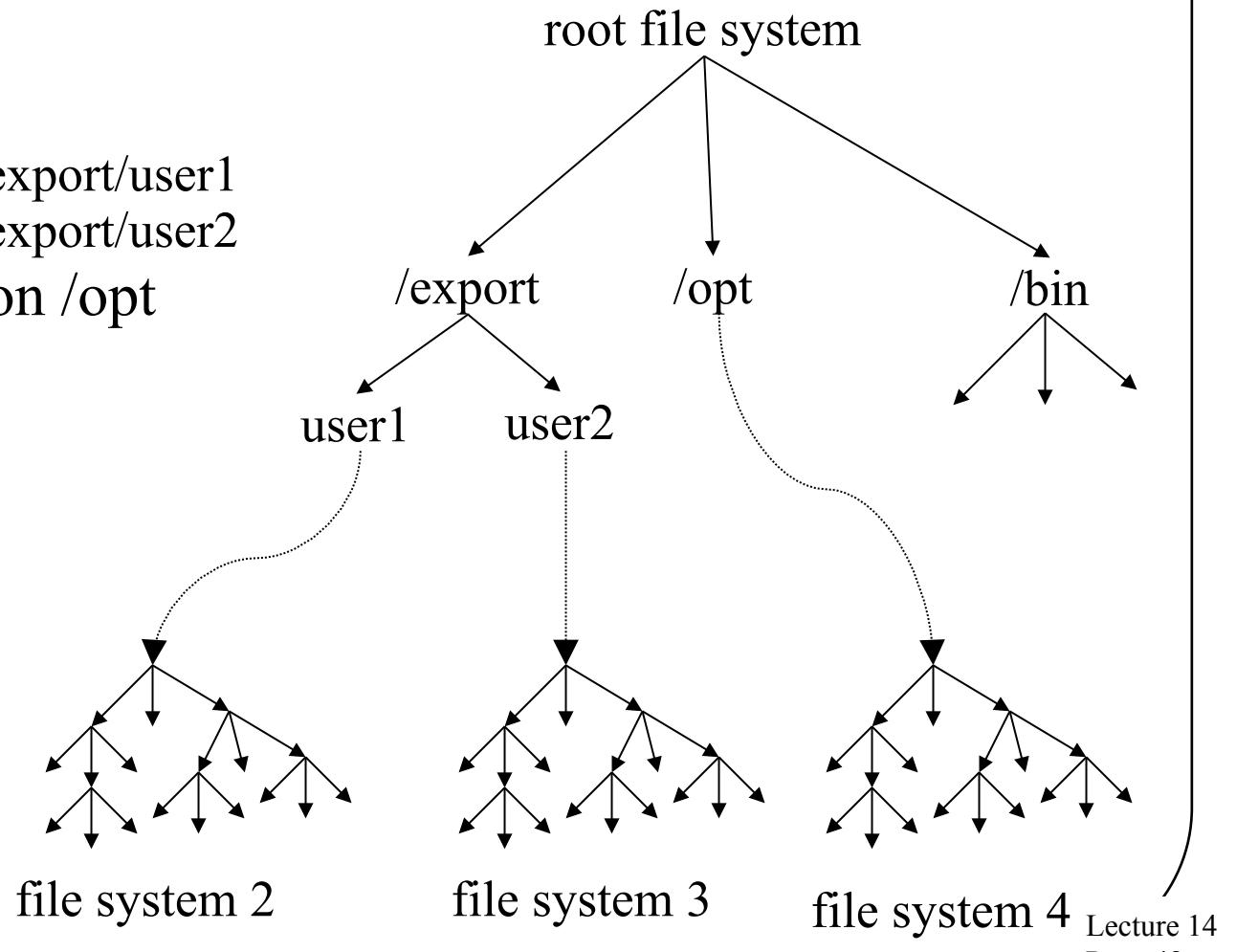
- Could specify the physical device it resides on
  - E.g., `/devices/pci/pci1000,4/disk/lun1/partition2`
    - that would get old real quick
- Could assign logical names to our partitions
  - E.g., “A:”, “C:”, “D:”
    - You only have to think physical when you set them up
    - But you still have to be aware multiple volumes exist
- Could weave a multi-file-system name space
  - E.g., Unix mounts

# Unix File System Mounts

- Goal:
  - To make many file systems appear to be one giant one
  - Users need not be aware of file system boundaries
- Mechanism:
  - *Mount device on directory*
  - Creates a warp from the named directory to the top of the file system on the specified device
  - Any file name beneath that directory is interpreted relative to the root of the mounted file system

# Unix Mounted File System Example

mount filesystem2 on /export/user1  
mount filesystem3 on /export/user2  
mount filesystem4 on /opt



# How Does This Actually Work?

- Mark the directory that was mounted on
- When file system opens that directory, don't treat it as an ordinary directory
  - Instead, consult a table of mounts to figure out where the root of the new file system is
- Go to that device and open its root directory
- And proceed from there

# What Happened To the Real Directory?

- You can mount on top of any directory
  - Not just in some special places in the file hierarchy
  - Not even just empty directories
- Did the mount wipe out the contents of the directory mounted on?
- No, it just hid them
  - Since traversals jump to a new file system, rather than reading the directory contents
- It's all still there when you unmount

# File System Performance Issues

- Key factors in file system performance
  - Head motion
  - Block size
- Possible optimizations for file systems
  - Read-ahead
  - Delayed writes
  - Caching (general and special purpose)

# Head Motion and File System Performance

- File system organization affects head motion
  - If blocks in a single file are spread across the disk
  - If files are spread randomly across the disk
  - If files and “meta-data” are widely separated
- All files are not used equally often
  - 5% of the files account for 90% of disk accesses
  - File locality should translate into head cylinder locality
- So how can we reduce head motion?

# Ways To Reduce Head Motion

- Keep blocks of a file together
  - Easiest to do on original write
  - Try to allocate each new block close to the last one
  - Especially keep them in the same cylinder
- Keep metadata close to files
  - Again, easiest to do at creation time
- Keep files in the same directory close together
  - On the assumption directory implies locality of reference
- If performing compaction, move popular files close together

# File System Performance and Block Size

- Larger block sizes result in efficient transfers
  - DMA is very fast, once it gets started
  - Per request set-up and head-motion is substantial
- They also result in internal fragmentation
  - Expected waste:  $\frac{1}{2}$  block per file
- As disks get larger, speed outweighs wasted space
  - File systems support ever-larger block sizes
- Clever schemes can reduce fragmentation
  - E.g., use smaller block size for the last block of a file

# Read Early, Write Late

- If we read blocks before we actually need them, we don't have to wait for them
  - But how can we know which blocks to read early?
- If we write blocks long after we told the application it was done, we don't have to wait
  - But are there bad consequences of delaying those writes?
- Some optimizations depend on good answers to these questions

# Read-Ahead

- Request blocks from the disk before any process asked for them
- Reduces process wait time
- When does it make sense?
  - When client specifically requests sequential access
  - When client seems to be reading sequentially
- What are the risks?
  - May waste disk access time reading unwanted blocks
  - May waste buffer space on unneeded blocks

# Delayed Writes

- Don't wait for disk write to complete to tell application it can proceed
- Written block is in a buffer in memory
- Wait until it's “convenient” to write it to disk
  - Handle reads from in-memory buffer
- Benefits:
  - Applications don't wait for disk writes
  - Writes to disk can be optimally ordered
  - If file is deleted soon, may never need to perform disk I/O
- Potential problems:
  - Lost writes when system crashes

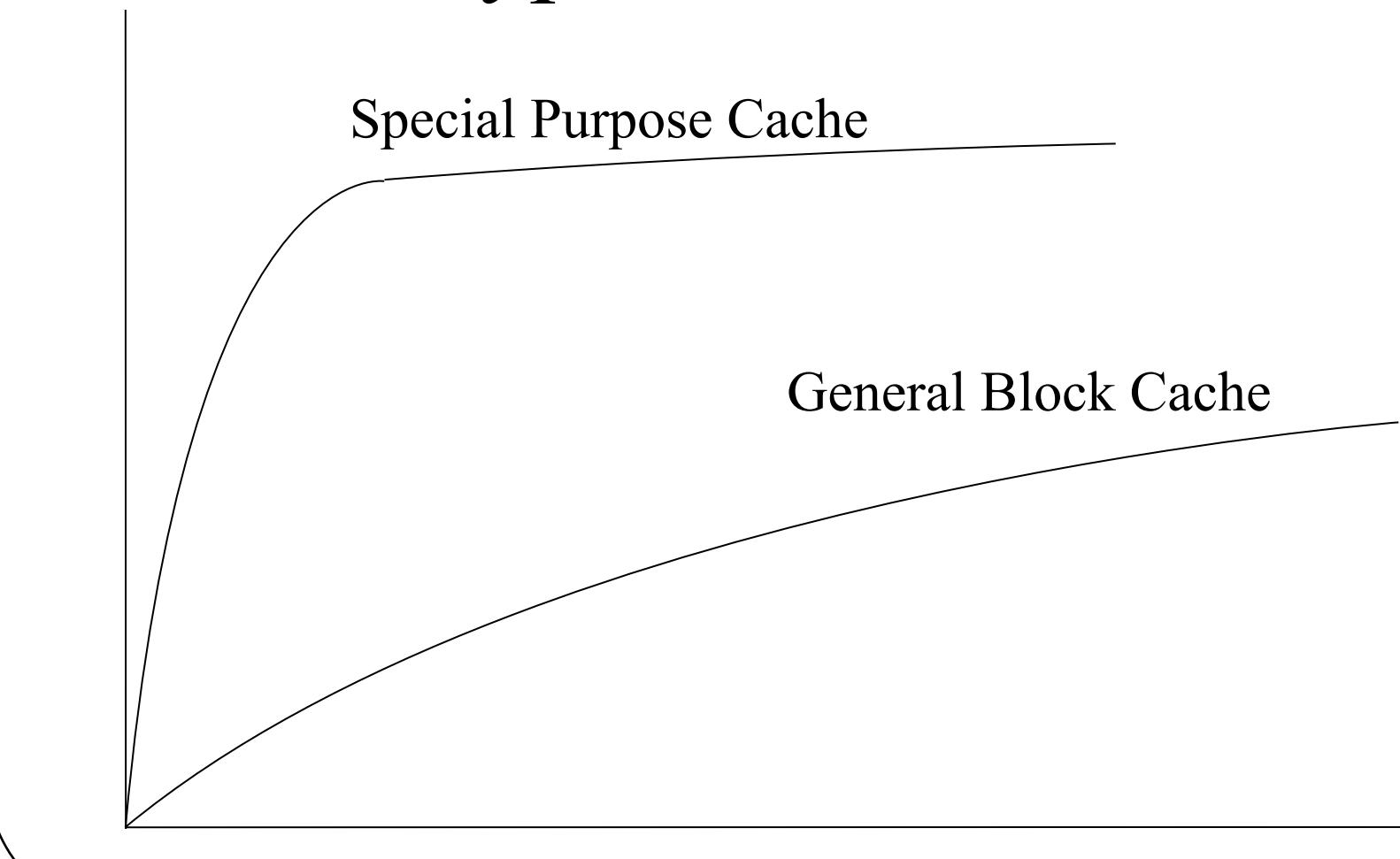
# Caching and Performance

- Big performance wins are possible if caches work well
  - They typically contain the block you’re looking for
- Should we have one big LRU cache for all purposes?
- Should we have some special-purpose caches?
  - If so, is LRU right for them?

# Common Types of Disk Caching

- General block caching
  - Popular files that are read frequently
  - Files that are written and then promptly re-read
  - Provides buffers for read-ahead and deferred write
- Special purpose caches
  - Directory caches speed up searches of same dirs
  - Inode caches speed up re-uses of same file
- Special purpose caches are more complex
  - But they often work much better

# Performance Gain For Different Types of Caches



# Why Are Special Purpose Caches More Effective?

- They match caching granularity to their need
  - E.g., cache inodes or directory entries
  - Rather than full blocks
- Why does that help?
- Consider an example:
  - A block might contain 100 directory entries, only four of which are regularly used
  - Caching the other 96 as part of the block is a waste of cache space
  - Caching 4 entries allows more popular entries to be cached
  - Tending to lead to higher hit ratios