File Systems: Naming and Performance
CS 111
Operating Systems
Peter Reiher

Outline

- File naming and directories
- File volumes
- File system performance issues
- File system reliability

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?

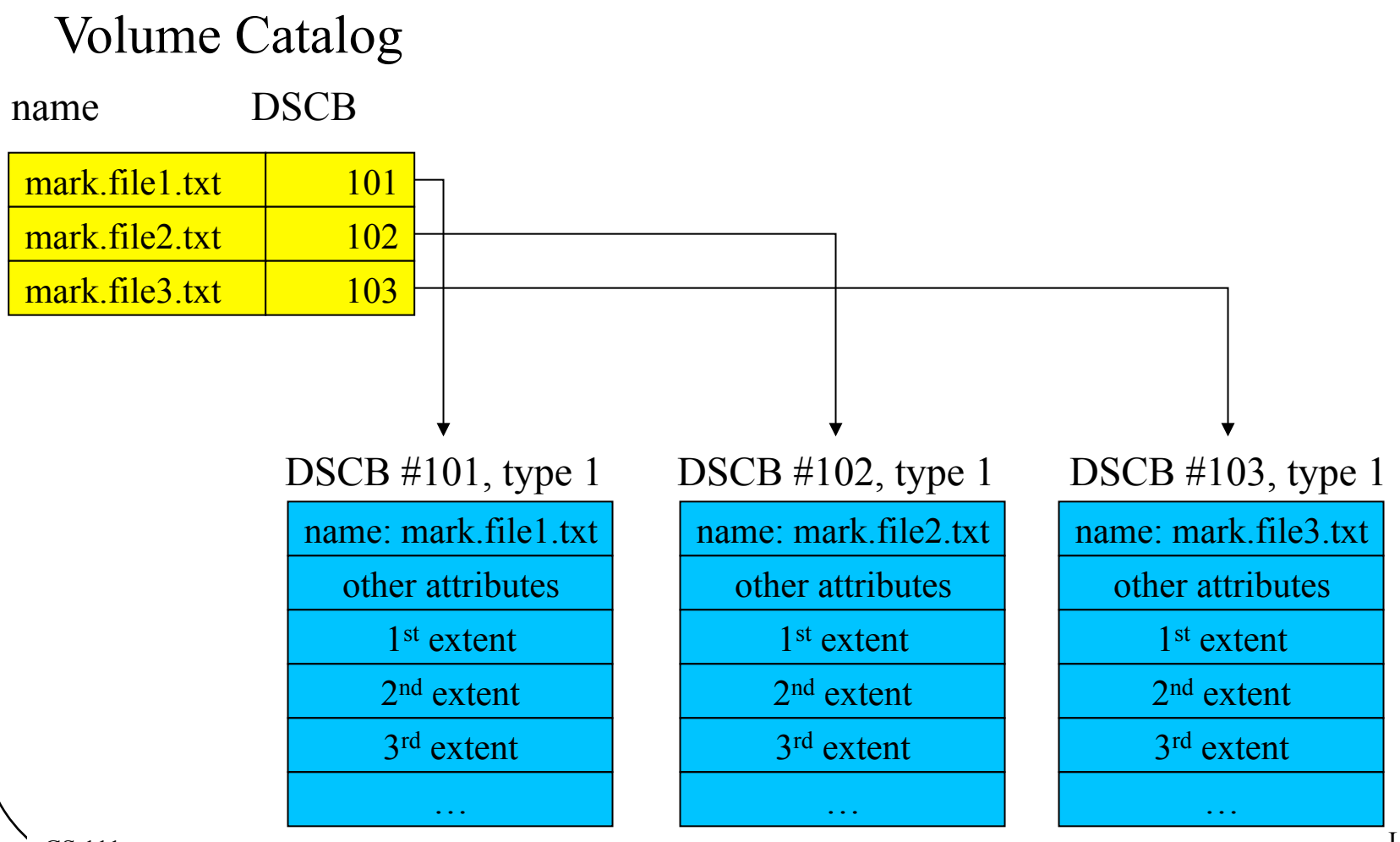
Flat Name Spaces

- 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., CAC101.CS111.SECTION1.SLIDES.LECTURE_13
  - 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

A Sample Flat File System - MVS

- A file system used in IBM mainframes in 60s and 70s
- Each file has a unique name
  - File name (usually very long) stored in the file's descriptor
- There is one master catalog file per volume
  - Lists names and descriptor locations for every file
  - Used to speed up searches
- The catalog is not critical
  - It can be deleted and recreated at any time
  - Files can be found without catalog ... it just takes longer
  - Some files are not listed in catalog, for secrecy
    - They cannot be found by “browsing” the name space

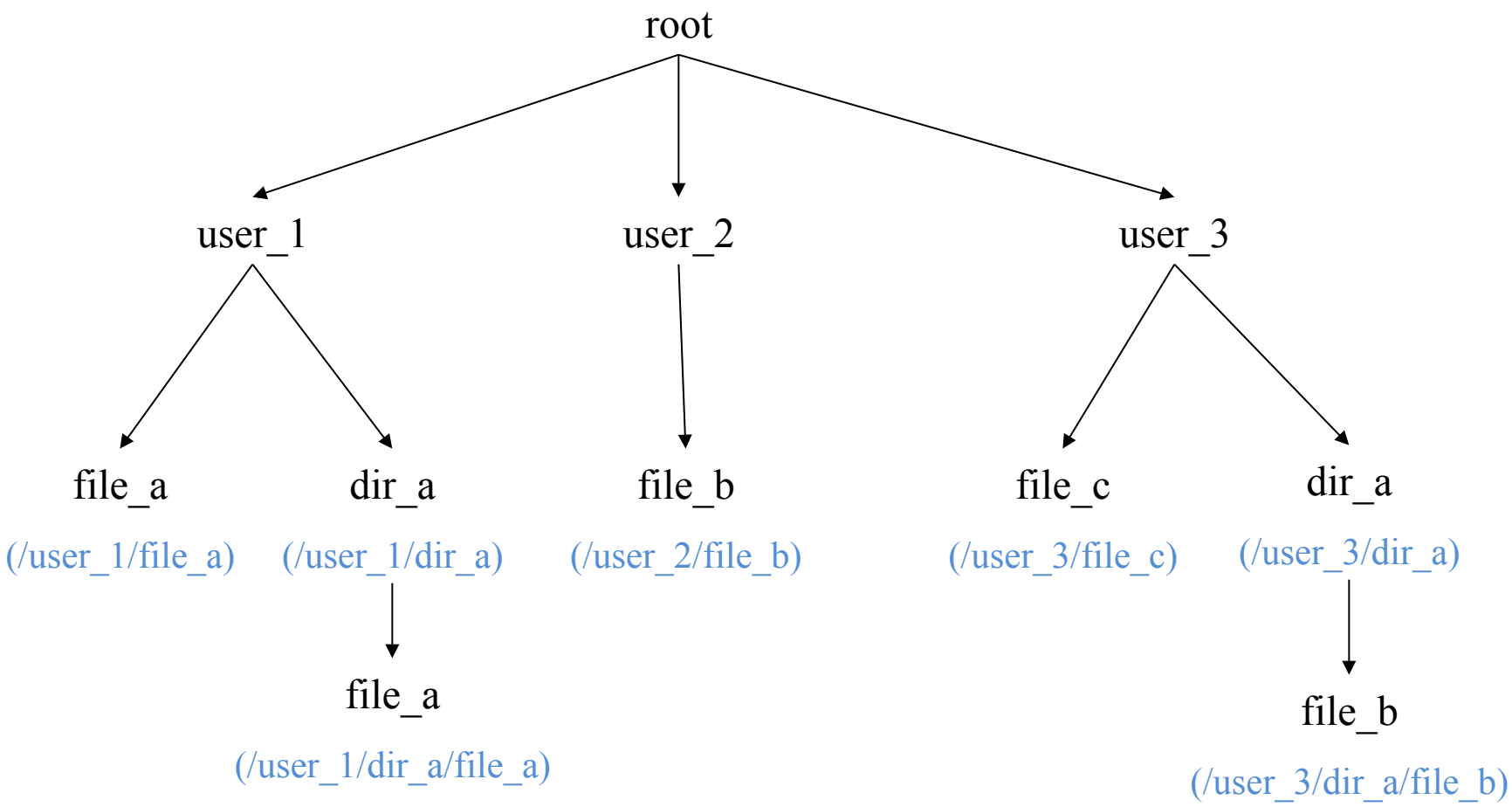
MVS Names and Catalogs



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	…	1st 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	…	1st 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 later

Root directory, inode #1

inode #	file name
1	.
1	..
9	user_1
31	user_2
114	user_3

Directory /user_3, inode #114

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

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

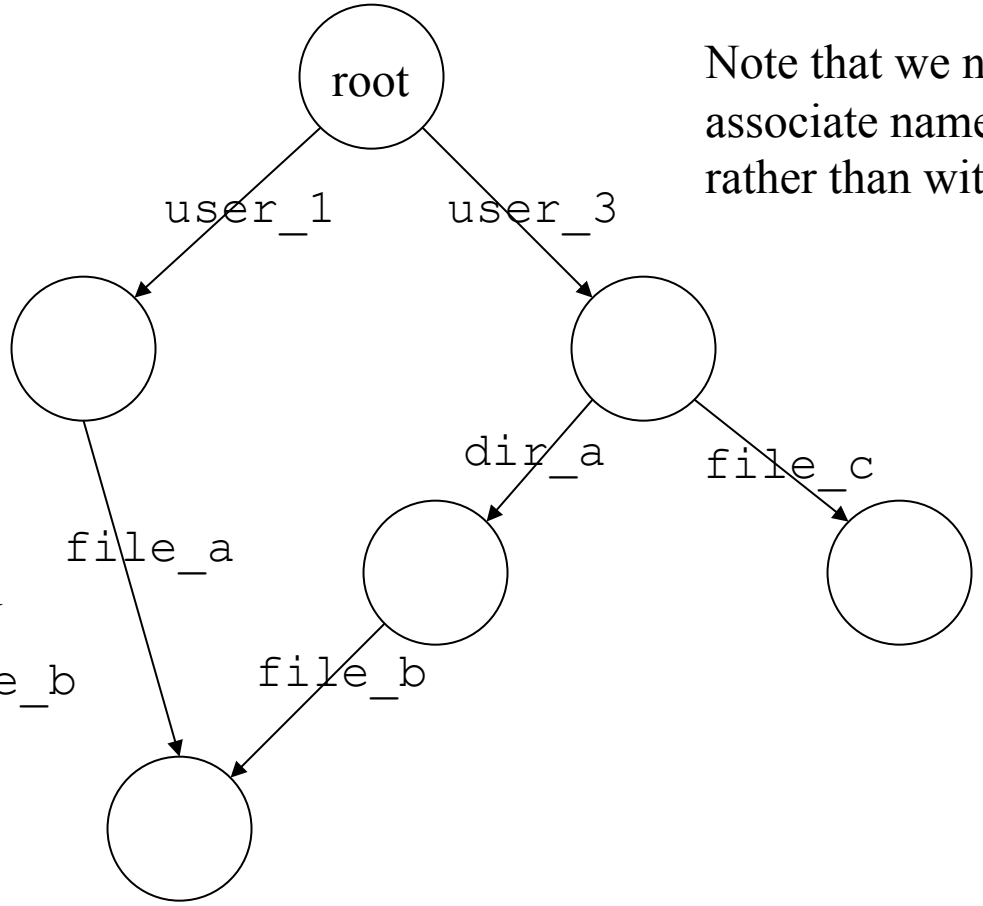
Multiple File Names In Unix

- 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



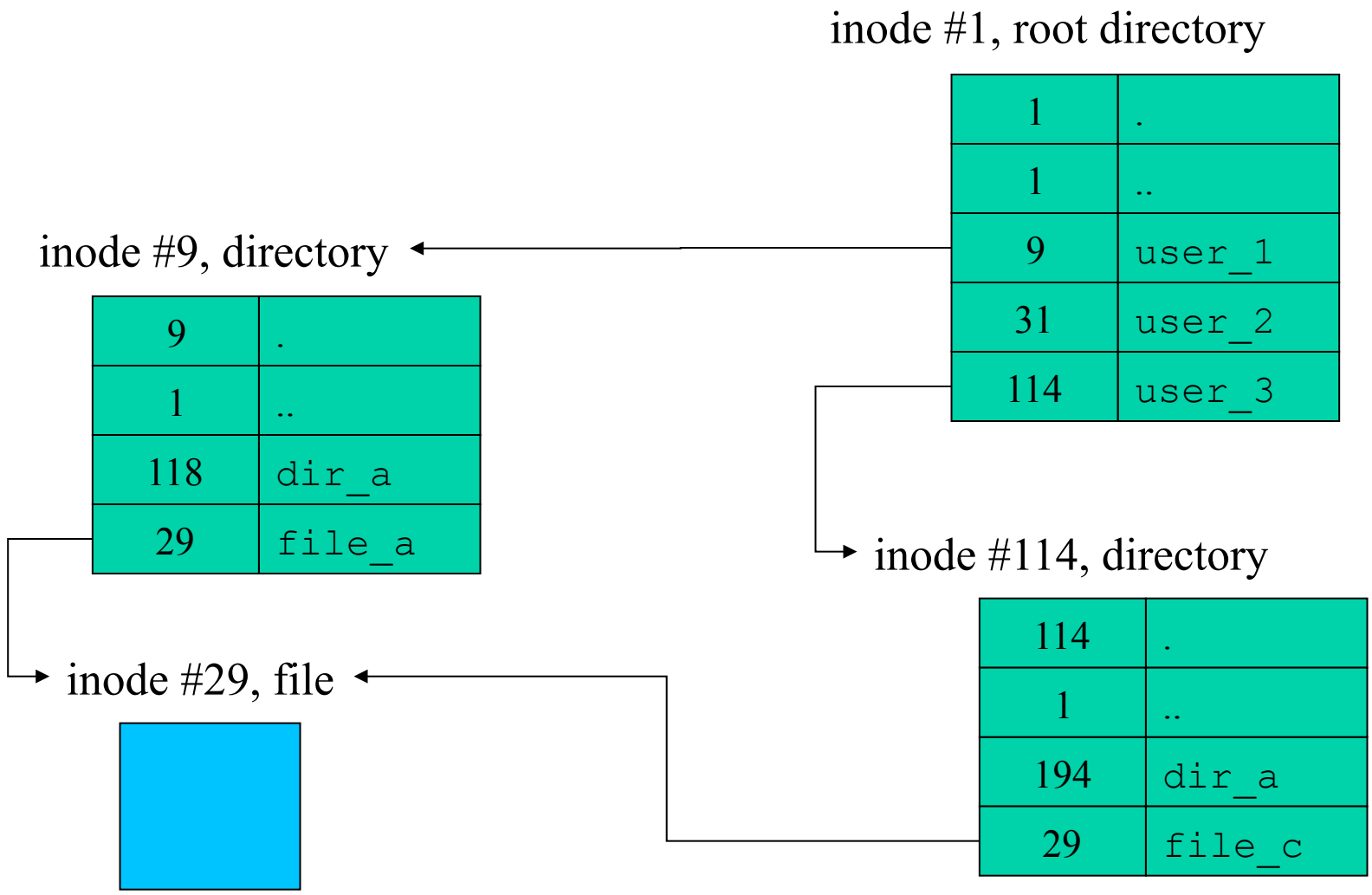
Note that we now associate names with links rather than with files.

`/user_1/file_a` and

`/user_3/dir_a/file_b`

are both links to the same inode

Hard Links, Directories, and Files

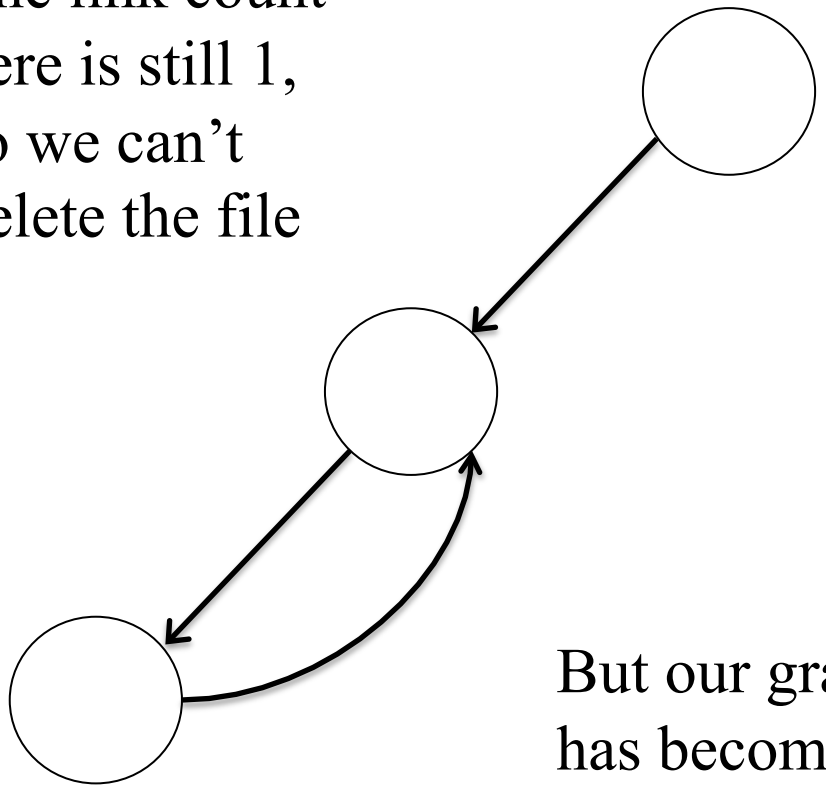


A Potential Problem With Hard Links

- 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



Now let's add a link

And now let's delete a link

But our graph has become disconnected!

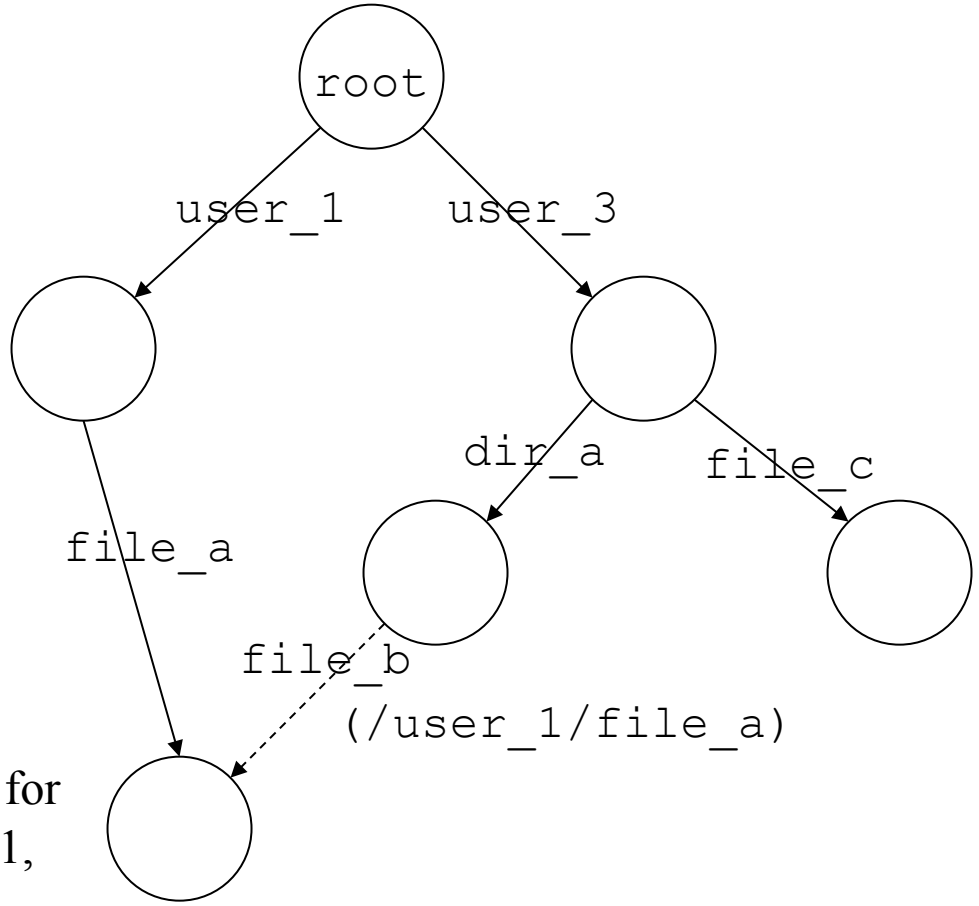
Solving the Problem

- 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

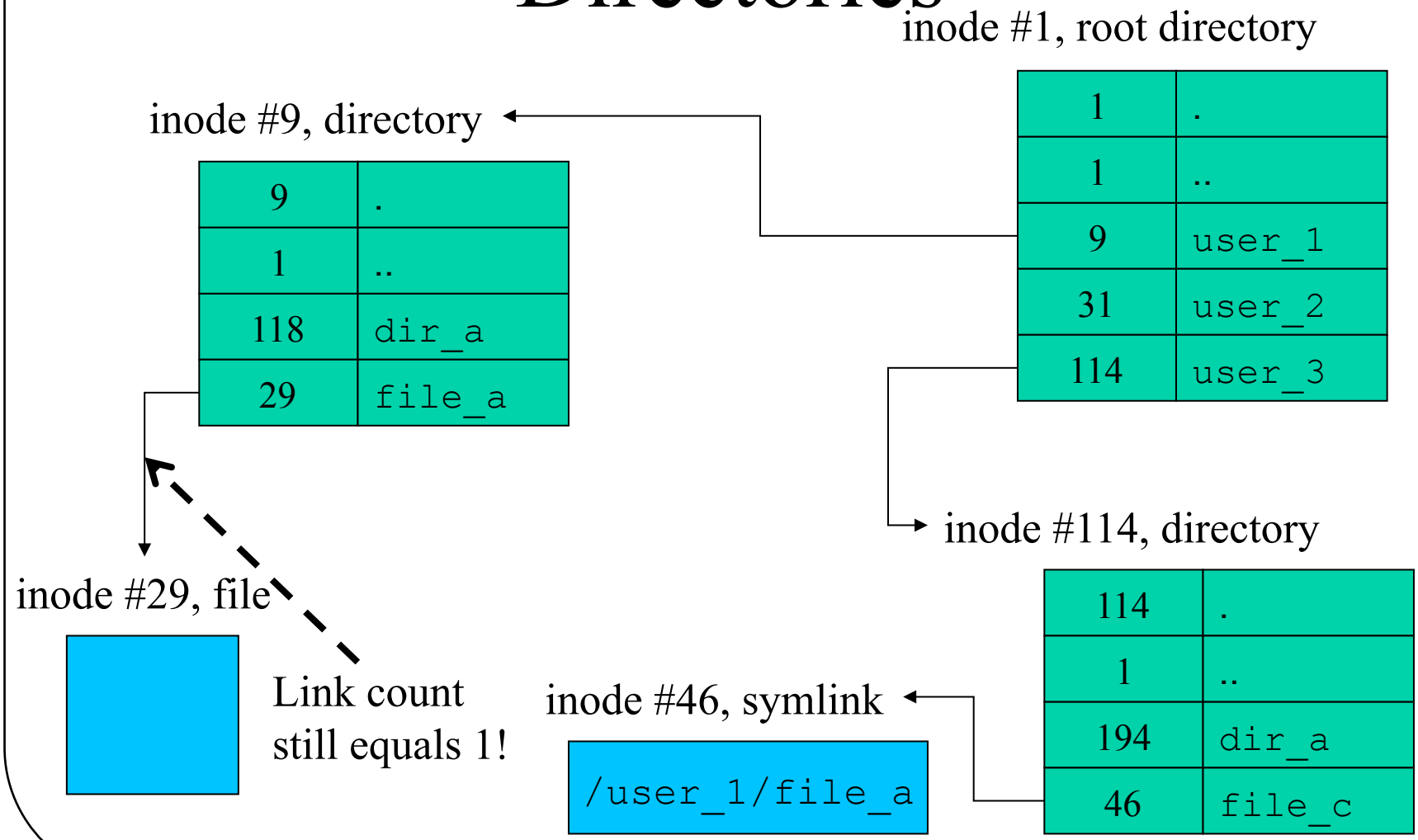
- A different way of giving files multiple names
- Symbolic links implemented as a special type of file
  - An indirect reference to some other file
  - Contents is a path name to another file
- OS recognizes symbolic links
  - Automatically opens associated file instead
  - If file is inaccessible or 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



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

How About the Other Way Around?

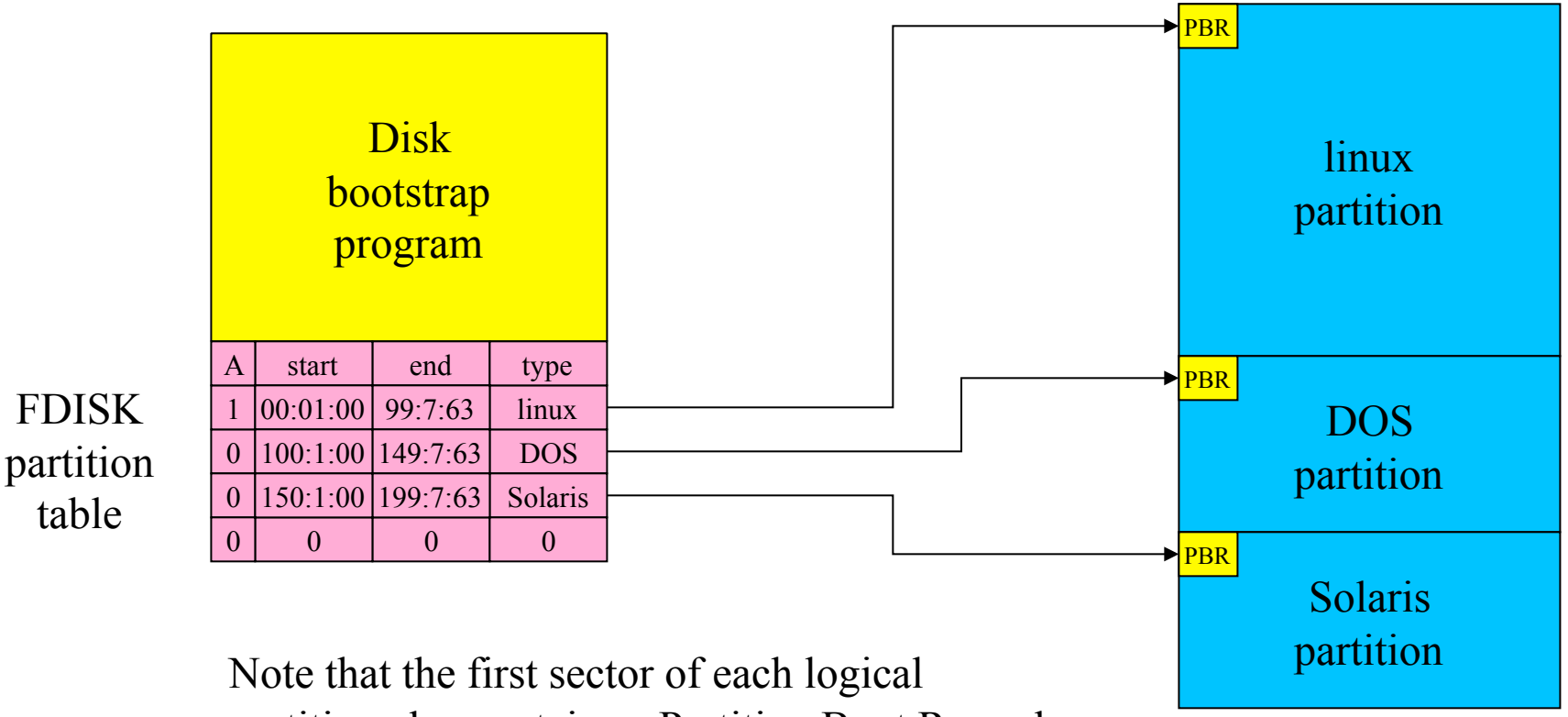
- Multiple file systems on one disk
- Divide physical disk into multiple logical disks
  - Often implemented within disk device drivers
  - Rest of system sees them as separate disk drives
- Typical motivations
  - Permit multiple OS to coexist on a single disk
    - E.g., a notebook that can boot either Windows or Linux
  - Separation for installation, back-up and recovery
    - E.g., separate personal files from the installed OS file system
  - Separation for free-space
    - Running out of space on one file system doesn't affect others

Disk Partitioning Mechanisms

- Some are designed for use by a single OS
  - E.g., Unix slices (one file system per slice)
- Some are designed to support multiple OS
  - E.g., DOS FDISK partitions, and VM/370 mini-disks
- Important features for supporting multiple OS's
  - Must be possible to boot from any partition
  - Must be possible to keep OS A out of OS B's partition
- There may be hierarchical partitioning
  - E.g., multiple UNIX slices within an FDISK partition

Example: FDISK Disk Partitioning

Physical sector 0 (Master Boot Record)



Note that the first sector of each logical partition also contains a Partition Boot Record, which will be used to boot the operating system for that partition.

Master Boot Records and Partition Boot Records

- Given the Master Boot Record bootstrap, why another Partition Boot Record bootstrap per partition?
- The bootstrap in the MBR typically only gives the user the option of choosing a partition to boot from
  - And then loads the boot block from the selected (or default) partition
- The PBR bootstrap in the selected partition knows how to traverse the file system in that partition
  - And how to interpret the load modules stored in it

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

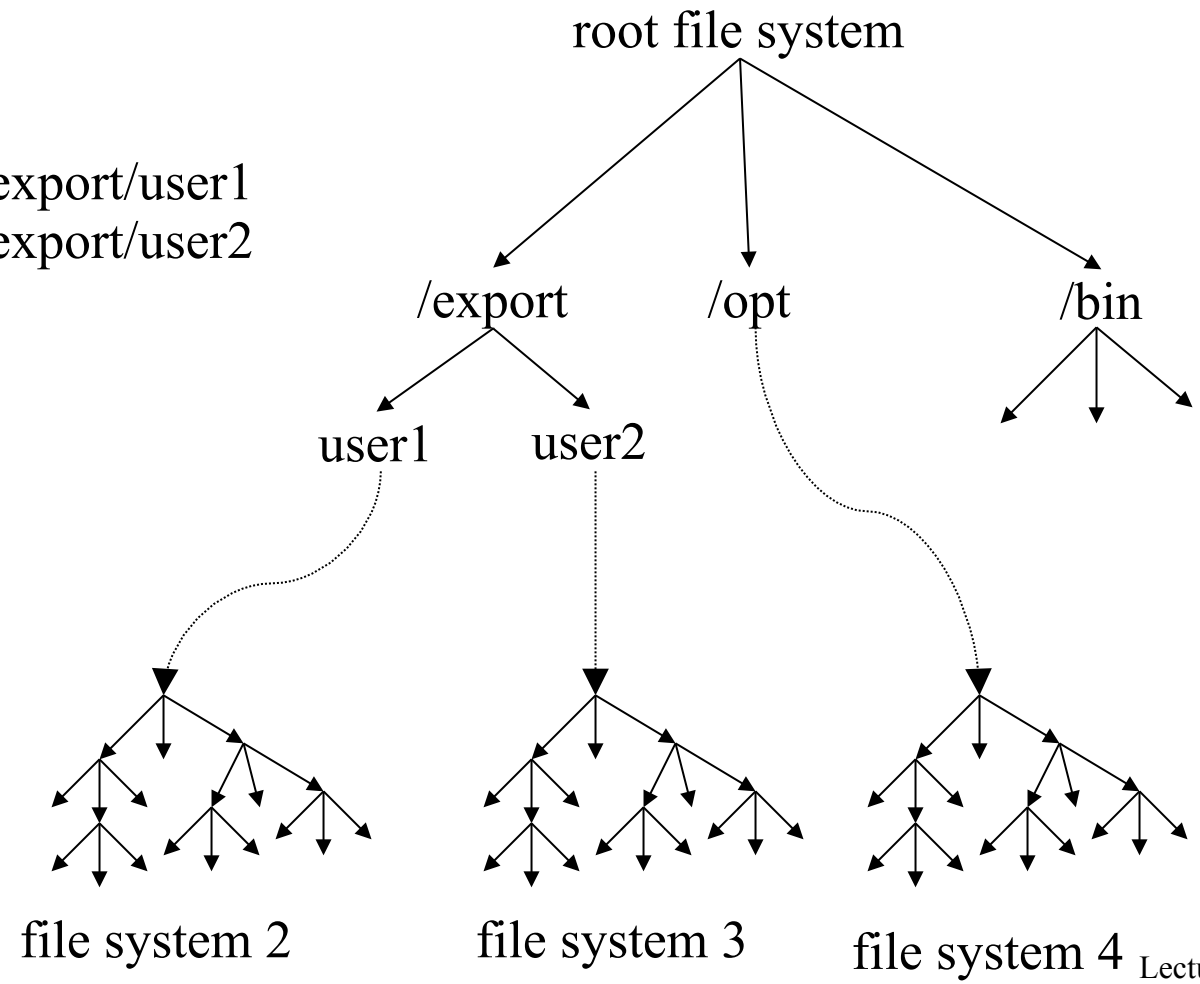
- Could specify the physical device it resides on
  - E.g., `/devices/pci/pci1000,4/disk/lun1/partition2`
    - that would get old real quick
- Could assign logical names to our partitions
  - E.g., "A:", "C:", "D:"
    - You only have to think physical when you set them up
    - But you still have to 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: ½ 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
  - Buffers holding delayed writes can't be re-used

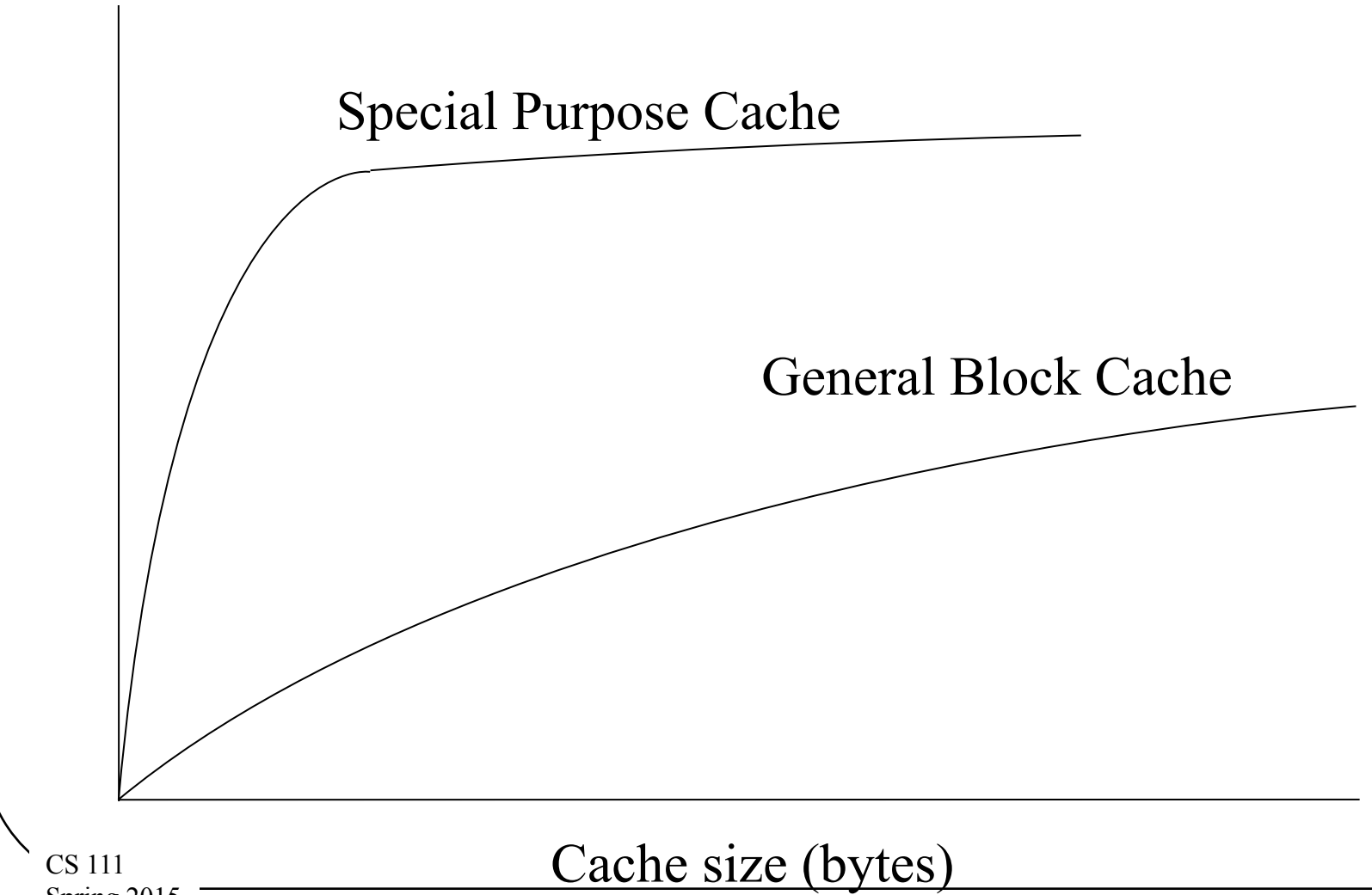
Caching and Performance

- 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

Remote File System Examples

- Common Internet File System (classic client/server)
- Network File System (peer-to-peer file sharing)
- Hyper-Text Transfer Protocol (a different approach)

Common Internet File System

- Originally a proprietary Microsoft Protocol
  - Newer versions (CIFS 1.0) are IETF standard
- Designed to enable "work group" computing
  - Group of PCs sharing same data, printers
  - Any PC can export its resources to the group
  - Work group is the union of those resources
- Designed for PC clients and NT servers
  - Originally designed for FAT and NT file systems
  - Now supports clients and servers of all types

CIFS Architecture

- Standard remote file access architecture
- State-full per-user client/server sessions
  - Password or challenge/response authentication
  - Server tracks open files, offsets, updates
  - Makes server fail-over much more difficult
- Opportunistic locking
  - Client can cache file if nobody else using/writing it
  - Otherwise all reads/writes must be synchronous
- Servers regularly advertise what they export
  - Enabling clients to "browse" the workgroup

Benefits of Opportunistic Locking

- A big performance win
- Getting permission from server before each write is a huge expense
  - In both time and server loading
- If no conflicting file use 99.99% of the time, opportunistic locks greatly reduce overhead
- When they can't be used, CIFS does provide correct centralized serialization

CIFS/SMB Protocol

- SMB (old, proprietary) ran over NetBIOS
  - Provided transport, reliable delivery, sessions, request/response, name service
- CIFS (new, IETF), uses TCP and DNS
- Scope
  - Session authentication
  - File and directory access and access control
  - File and record-level locking (opportunistic)
  - File and directory change notification
  - Remote printing

CIFS/SMB Pros and Cons

- Performance/Scalability
  - Opportunistic locks enable good performance
  - Otherwise, forced synchronous I/O is slow
- Transparency
  - Very good, especially the global name space
- Conflict Prevention
  - File/record locking and synchronous writes work well
- Robustness
  - State-full servers make seamless fail-over impossible

The Network File System (NFS)

- Transparent, heterogeneous file system sharing
  - Local and remote files are indistinguishable
- Peer-to-peer and client-server sharing
  - Disk-full clients can export file systems to others
  - Able to support diskless (or dataless) clients
  - Minimal client-side administration
- High efficiency and high availability
  - Read performance competitive with local disks
  - Scalable to huge numbers of clients
  - Seamless fail-over for all readers and some writers

The NFS Protocol

- Relies on idempotent operations and stateless server
  - Built on top of a remote procedure call protocol
  - With eXternal Data Representation, server binding
  - Versions of RPC over both TCP or UDP
  - Optional encryption (may be provided at lower level)
- Scope – basic file operations only
  - Lookup (open), read, write, read-directory, stat
  - Supports client or server-side authentication
  - Supports client-side caching of file contents
  - Locking and auto-mounting done with another protocol

NFS Authentication

- How can we trust NSF clients to authenticate themselves?
- NFS not not designed for direct use by user applications
- It permits one operating system instance to access files belonging to another OS instance
- If we trust the remote OS to see the files, might as well trust it to authenticate the user
- Obviously, don't use NFS if you don't trust the remote OS . . .

NFS Replication

- NFS file systems can be replicated
  - Improves read performance and availability
  - Only one replica can be written to
- Client-side agent (in OS) handles fail-over
  - Detects server failure, rebinds to new server
- Limited transparency for server failures
  - Most readers will not notice failure (only brief delay)
  - Users of changed files may get "stale handle" error
  - Active locks may have to be re-obtained

NFS and Updates

- An NFS server does not prevent conflicting updates
  - As with local file systems, this is application's job
- Auxiliary server/protocol for file and record locking
  - All leases are maintained on the lock server
  - All lock/unlock operations handed by lock server
- Client/network failure handling
  - Server can break locks if client dies or times out
  - "Stale-handle" errors inform client of broken lock
  - Client response to these errors are application specific
- Lock server failure handling is very complex

NFS Pros and Cons

- Transparency/Heterogeneity
  - Local/remote transparency is excellent
  - NFS works with all major ISAs, OSs, and FSs
- Performance
  - Read performance may be better than local disk
  - Replication option for scalable read bandwidth
  - Write performance slower than local disk
- Robustness
  - Transparent fail-over capability for readers
  - Recoverable fail-over capability for writers

NFS Vs. CIFS

- Functionality
  - NFS is much more portable (platforms, OS, FS)
  - CIFS provides much better write serialization
- Performance and robustness
  - NFS provides much greater read scalability
  - NFS has much better fail-over characteristics
- Security
  - NFS supports more security models
  - CIFS gives the server better authorization control

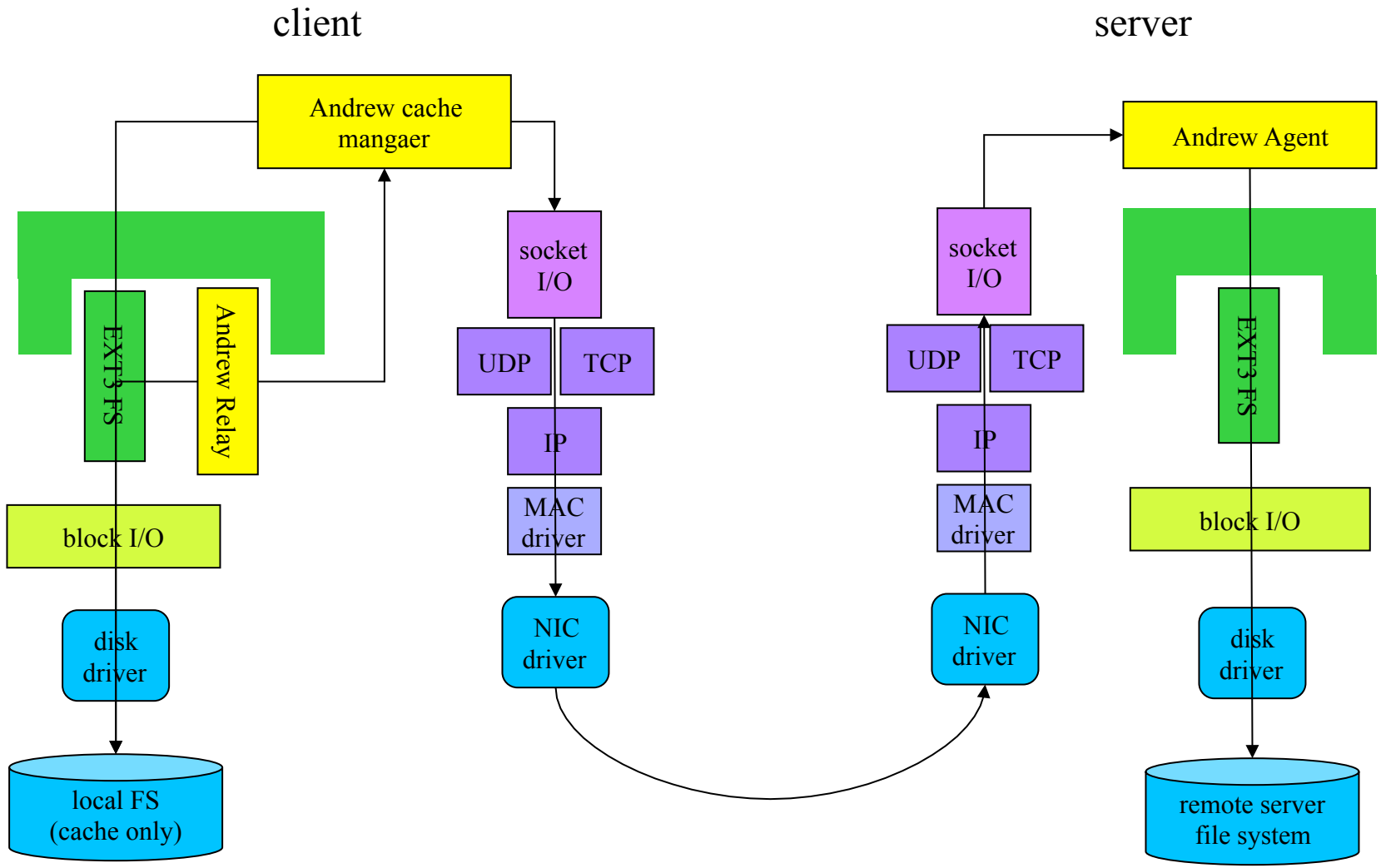
The Andrew File System

- AFS
- Developed at CMU
- Designed originally to support student and faculty use
  - Generally, large numbers of users of a single organization
- Uses a client/server model
- Makes use of whole-file caching

AFS Basics

- Designed for scalability, performance
  - Large numbers of clients and very few servers
  - Needed performance of local file systems
  - Very low per-client load imposed on servers
  - No administration or back-up for client disks
- Master files reside on a file server
  - Local file system is used as a local cache
  - Local reads satisfied from cache when possible
  - Files are only read from server if not in cache
- Simple synchronization of updates

AFS Architecture



AFS Replication

- One replica at server, possibly many at clients
- Check for local copies in cache at open time
  - If no local copy exists, fetch it from server
  - If local copy exists, see if it is still up-to-date
    - Compare file size and modification time with server
  - Optimizations reduce overhead of checking
    - Subscribe/broadcast change notifications
    - Time-to-live on cached file attributes and contents
- Send updates to server when file is closed
  - Wait for all changes to be completed
  - File may be deleted before it is closed
    - E.g., temporary files that servers need not know about

AFS Reconciliation

- Client sends updates to server when local copy closed
- Server notifies all clients of change
  - Warns them to invalidate their local copy
  - Warns them of potential write conflicts
- Server supports only advisory file locking
  - Distributed file locking is extremely complex
- Clients are expected to handle conflicts
  - Noticing updates to files open for write access
  - Notification/reconciliation strategy is unspecified

AFS Pros and Cons

- Performance and Scalability
  - All file access by user/applications is local
  - Update checking (with time-to-live) is relatively cheap
  - Both fetch and update propagation are very efficient
  - Minimal per-client server load (once cache filled)
- Robustness
  - No server fail-over, but have local copies of most files
- Transparency
  - Mostly perfect - all file access operations are local
  - Pray that we don't have any update conflicts

AFS vs. NFS

- Basic designs
  - Both designed for continuous connection client/server
  - NFS supports diskless clients without local file systems
- Performance
  - AFS generates much less network traffic, server load
  - They yield similar client response times
- Ease of use
  - NFS provides for better transparency
  - NFS has enforced locking and limited fail-over
- NFS requires more support in operating system

HTTP

- A different approach, for a different purpose
- Stateless protocol with idempotent operations
  - Implemented atop TCP (or other reliable transport)
  - Whole file transport (not remote data access)
    - **get** file, **put** file, **delete** file, **post** form-contents
  - Anonymous file access, but secure (SSL) transfers
  - Keep-alive sessions (for performance only)
- A truly global file namespace (URLs)
  - Client and in-network caching to reduce server load
  - A wide range of client redirection options

HTTP Architecture

- Not a traditional remote file access mechanism
- We do not try to make it look like local file access
  - Apps are written to HTTP or other web-aware APIs
  - No interception and translation of local file operations
  - But URLs can be constructed for local files
- Server is entirely implemented in user-mode
  - Authentication via SSL or higher level dialogs
  - All data is assumed readable by all clients
- HTTP servers provide more than remote file access
  - POST operations invoke server-side processing
- No attempt to provide write locking or serialization

HTTP Pros and Cons

- Transparency
  - Universal namespace for heterogeneous data
  - Requires use of new APIs and namespace
  - No attempt at compatibility with old semantics
- Performance
  - Simple implementations, efficient transport
  - Unlimited read throughput scalability
  - Excellent caching and load balancing
- Robustness
  - Automatic retrys, seamless fail-over, easy redirects
  - Not much attempt to handle issues related to writes

HTTP vs. NFS/CIFS

- The file model and services provided by HTTP are much weaker than those provided by CIFS or NFS
- So why would anyone choose to use HTTP for remote file access?
- It's easy to use, provides excellent performance, scalability and availability, and is ubiquitous
- If I don't need per-user authorization, walk-able name spaces, and synchronized updates,
  - Why pay the costs of more elaborate protocols?
  - If I do need, them, though, . . .

Conclusion

- Be clear about your remote file system requirements
  - Different priorities lead to different tradeoffs & designs
- The remote file access protocol is the key
  - It determines the performance and robustness
  - It imposes or presumes security mechanisms
  - It is designed around synchronization & fail-over mechanisms
- Stateless protocols with idempotent ops are limiting
  - But very rewarding if you can accept those limitations
- Read-only content is a pleasure to work with
  - Synchronized and replicated updates are very hard