

Operating System Principles:  
File Systems – Allocation, Naming, Performance, and Reliability  
CS 111  
Operating Systems

# Outline

- Allocating and managing file system free space
- Other performance improvement strategies
- File naming and directories
- File system reliability 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

# 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



## Other Performance Improvement Strategies

- Beyond disk layout issues
  - Which are only relevant for hard drives, not flash or other solid state devices
  - Though they have their own unique issues
- Transfer size
- Caching

# Allocation/Transfer Size

- Per operation overheads are high
  - DMA startup, interrupts, device-specific costs
- Larger transfer units more efficient
  - Amortize fixed per-op costs over more bytes/op
  - Multi-megabyte transfers are very good
- What unit do we use to allocate storage space?
  - Small chunks reduce efficiency
  - Large fixed size chunks -> internal fragmentation
  - Variable sized chunks -> external fragmentation
  - Tradeoff between fragmentation and efficiency

# Flash Drive Issues

- Flash is becoming the dominant technology
- Special flash characteristics:
  - Faster than hard disks, slower than RAM
  - Any location equally fast to access
  - But write-once/read-many access
    - Until you erase
  - You can only erase very large chunks of memory
- Think about this as we discuss other file system issues

# Caching

- Caching for reads
- Caching for writes

# Read Caching

- Disk I/O takes a long time
  - Deep queues, large transfers improve efficiency
  - They do not make it significantly faster
- We must eliminate much of our disk I/O
  - Maintain an in-memory cache
  - Depend on locality, reuse of the same blocks
  - Check cache before scheduling I/O

# 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

# Write Caching

- Most disk writes go to a write-back cache
  - They will be flushed out to disk later
- Aggregates small writes into large writes
  - If application does less than full block writes
- Eliminates moot writes
  - If application subsequently rewrites the same data
  - If application subsequently deletes the file
- Accumulates large batches of writes
  - A deeper queue to enable better disk scheduling

# 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 by matching cache granularities to actual needs



# 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?

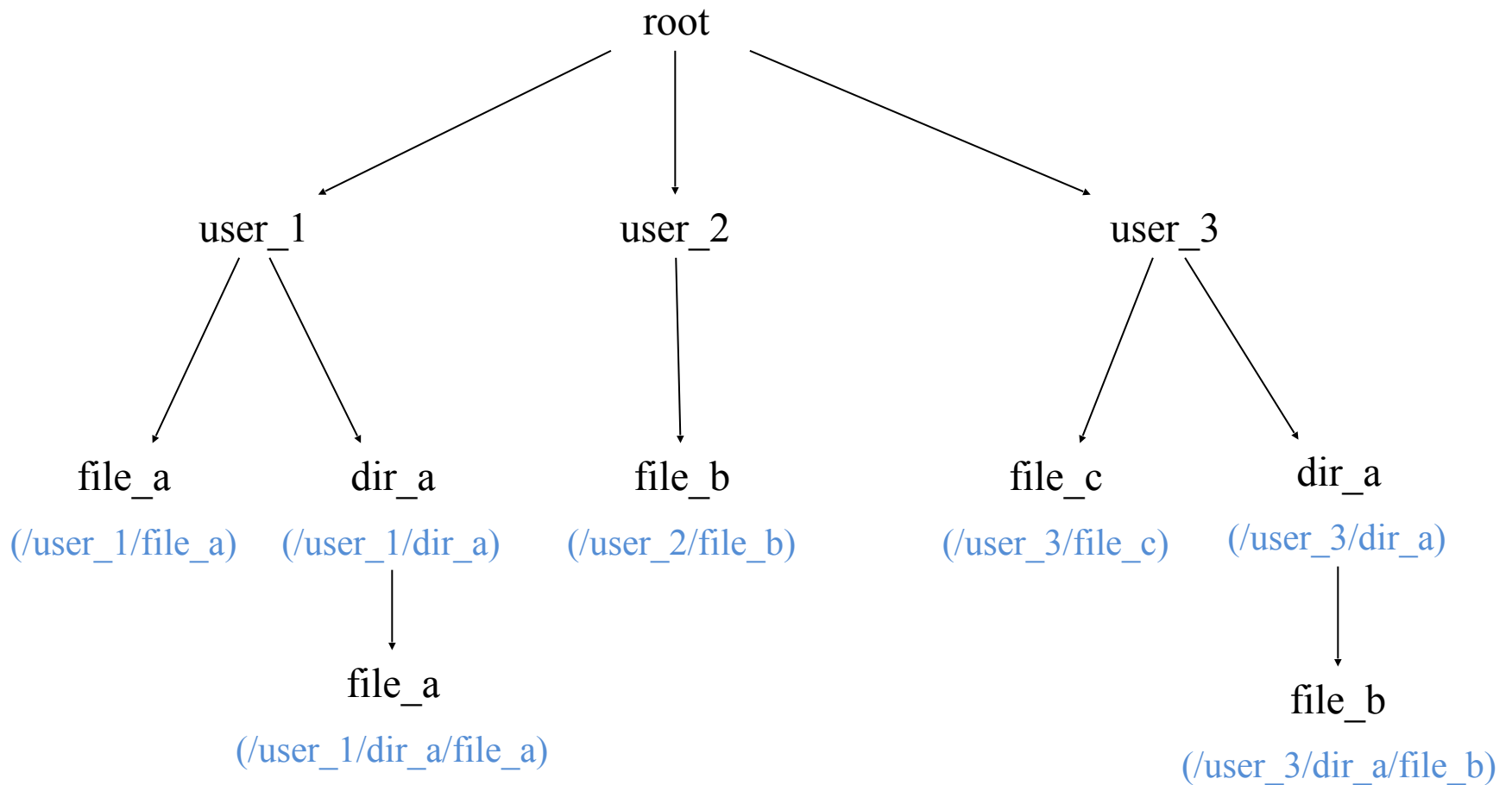
# Some Issues in Name Space Structure

- 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?

# 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



# File Names Vs. Path Names

- In some name space systems, files had “true names”
  - Only one possible name for a file,
  - Kept in a record somewhere
- E.g., 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
- 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

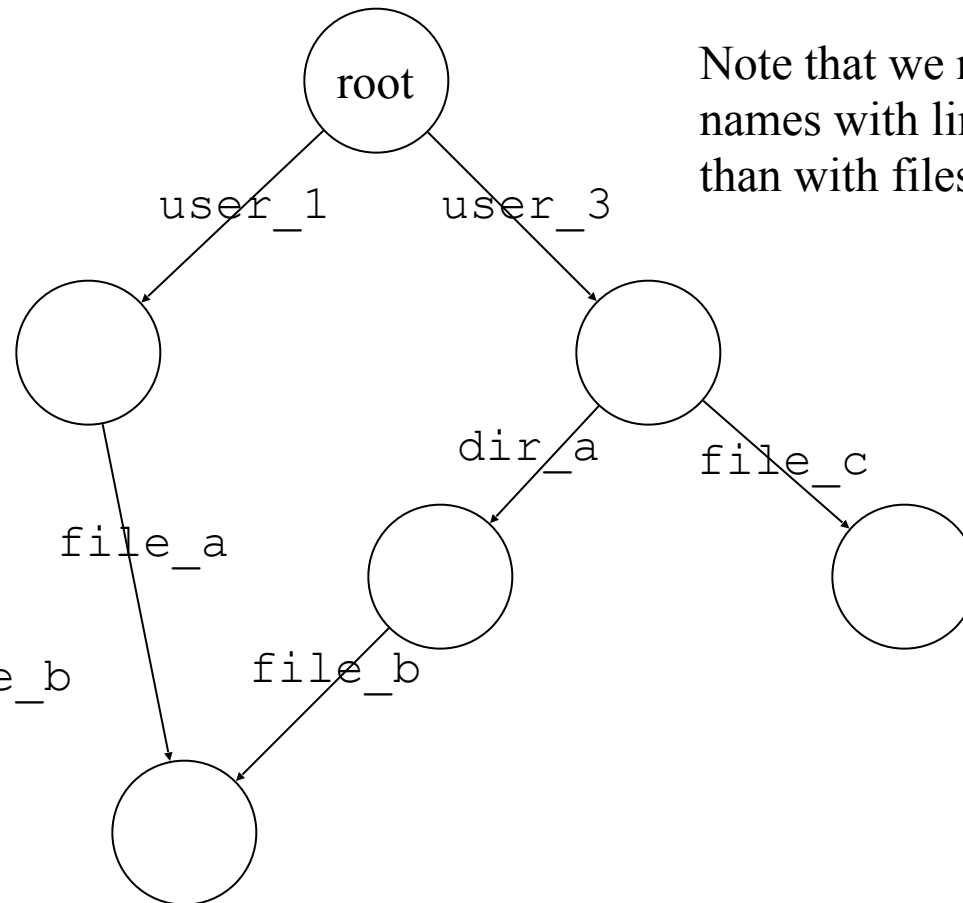
# Multiple File Names In Unix

- 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

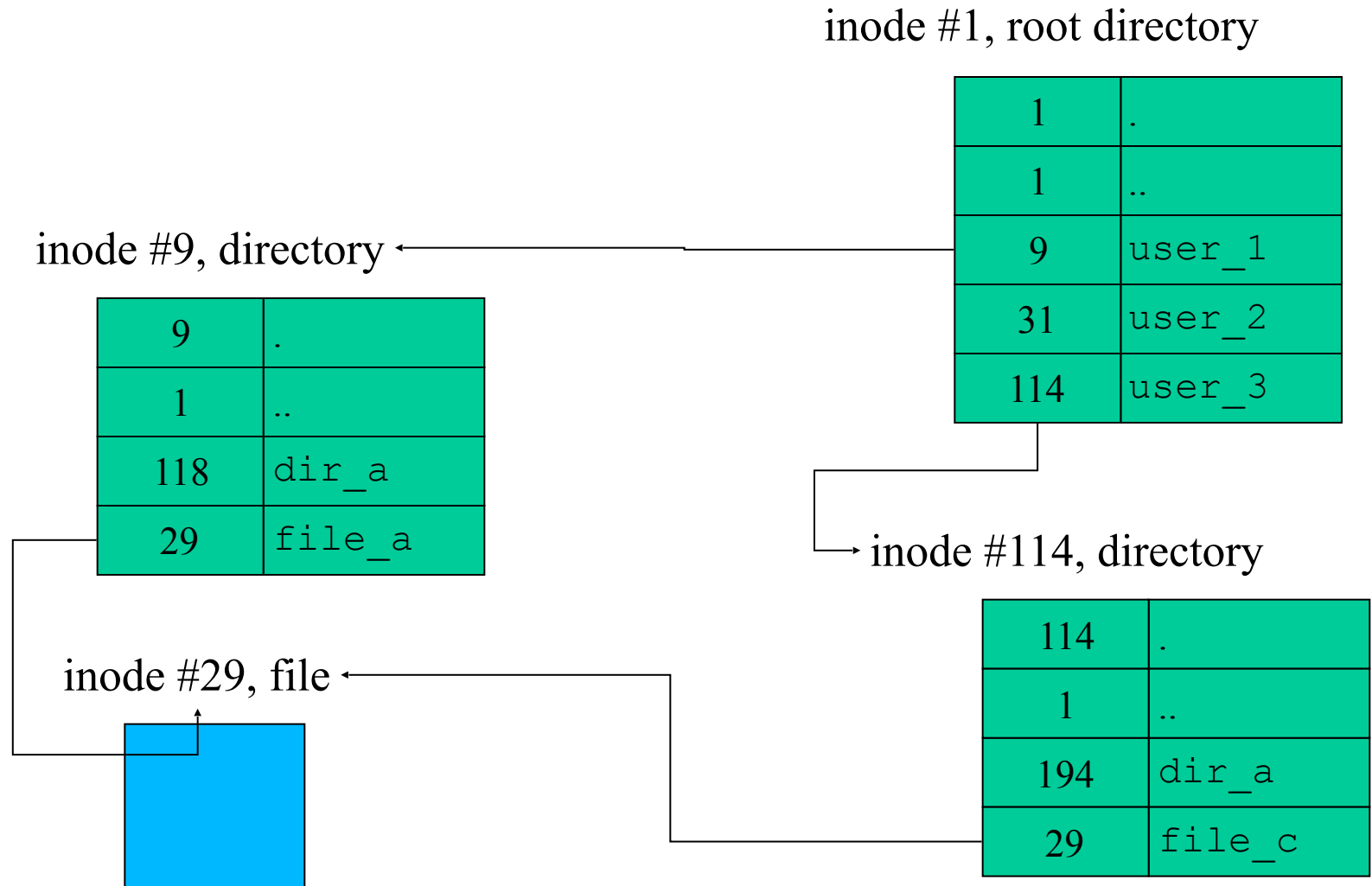
# Unix Hard Link Example



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

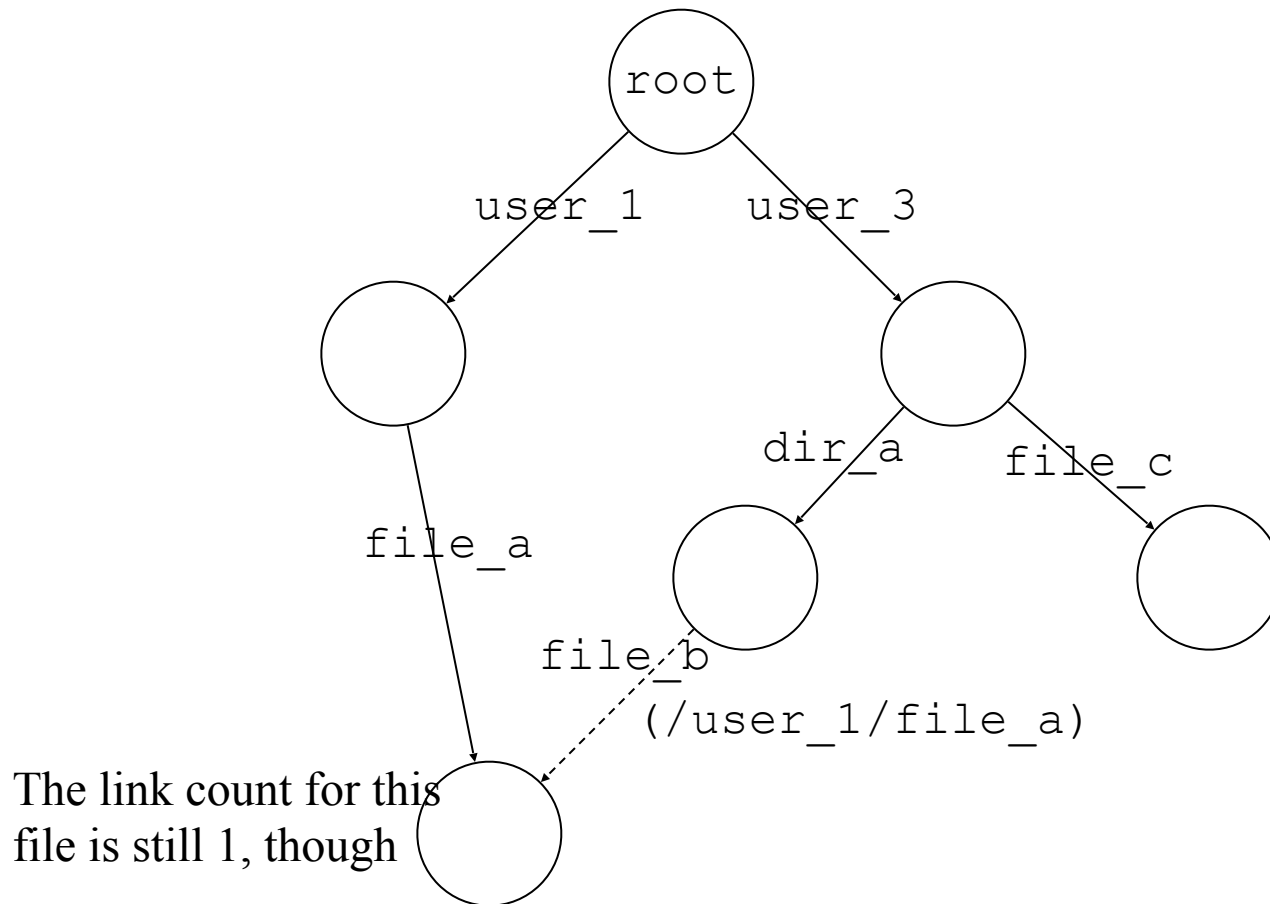


# Symbolic Links

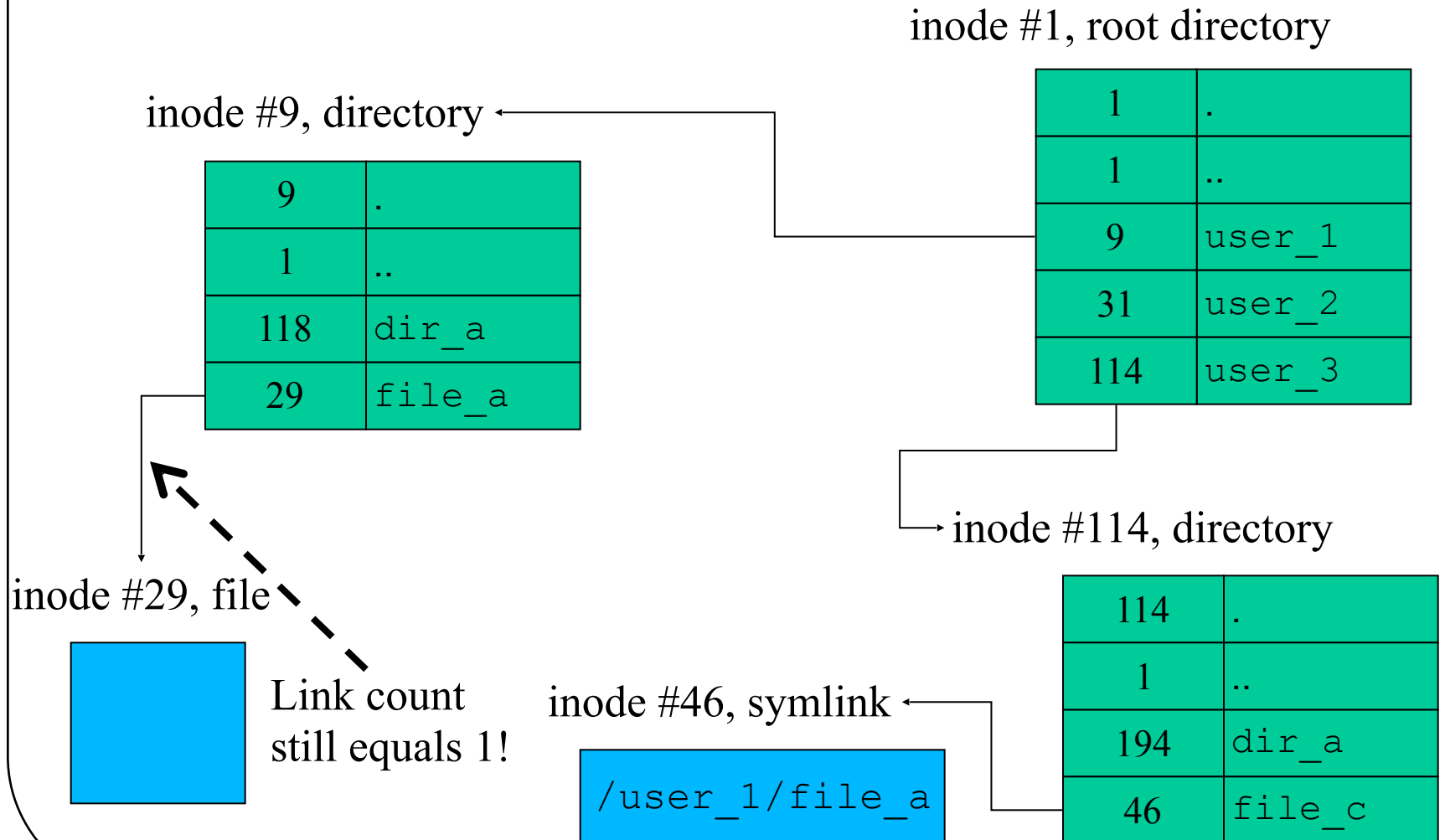
- A different way of giving files multiple names
- Symbolic links implemented as a special type of file
  - An indirect reference to some other file
  - Contents is a path name to another file
- File system recognizes symbolic links
  - Automatically opens associated file instead
  - If file is inaccessible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



# Symbolic Link Example



# Symbolic Links, Files, and Directories



# File Systems Reliability

- What can go wrong in a file system?
- Data loss
  - File or data is no longer present
  - Some/all of data cannot be correctly read back
- File system corruption
  - Lost free space
  - References to non-existent files
  - Corrupted free-list multiply allocates space
  - File contents over-written by something else
  - Corrupted directories make files un-findable
  - Corrupted inodes lose file info/pointers

# File Systems – System Failures

- Caused by system crashes or OS bugs
- Queued writes that don't get completed
  - Client writes that will not be persisted
  - Client creates that will not be persisted
  - Partial multi-block file system updates
- Can also be caused by power failures
  - Solution: NVRAM disk controllers
  - Solution: Uninterruptable Power Supply (UPS)
  - Solution: super-caps and fast flush

# Deferred Writes – A Worst Case Scenario

- Process allocates a new block to file A
  - We get a new block (x) from the free list
  - We write out the updated inode for file A
  - We defer free-list write-back (happens all the time)
- The system crashes, and after it reboots
  - A new process wants a new block for file B
  - We get block x from the (stale) free list
- Two different files now contain the same block
  - When file A is written, file B gets corrupted
  - When file B is written, file A gets corrupted

# Robustness – Ordered Writes

- Carefully ordered writes can reduce potential damage
- Write out data before writing pointers to it
  - Unreferenced objects can be garbage collected
  - Pointers to incorrect info are more serious
- Write out deallocations before allocations
  - Disassociate resources from old files ASAP
  - Free list can be corrected by garbage collection
  - Improperly shared data is more serious than missing data

# Practicality of Ordered Writes

- Greatly reduced I/O performance
  - Eliminates accumulation of near-by operations
  - Eliminates consolidation of updates to same block
- May not be possible
  - Modern devices may re-order queued requests
- Doesn't actually solve the problem
  - Does not eliminate incomplete writes
  - It chooses minor problems over major ones

# Robustness – Audit and Repair

- Design file system structures for audit and repair
  - Redundant information in multiple distinct places
    - Maintain reference counts in each object
    - Children have pointers back to their parents
    - Transaction logs of all updates
  - All resources can be garbage collected
    - Discover and recover unreferenced objects
- Audit file system for correctness (prior to mount)
  - All objects are well formatted
  - All references and free-lists are correct and consistent
- Use redundant info to enable automatic repair



# Practicality of Audit and Repair

- Integrity checking a file system after a crash
  - Verifying check-sums, reference counts, etc.
  - Automatically correct any inconsistencies
  - A standard practice for many years (see *fsck(8)*)
- No longer practical
  - Checking a 2TB FS at 100MB/second = 5.5 hours
- We need more efficient partial write solutions
  - File systems that are immune to them
  - File systems that enable very fast recovery

# Journaling

- Create a circular buffer journaling device
  - Journal writes are always sequential
  - Journal writes can be batched
  - Journal is relatively small, may use NVRAM
- Journal all intended file system updates
  - Inode updates, block write/alloc/free
- Efficiently schedule actual file system updates
  - Write-back cache, batching, motion-scheduling
- Journal completions when real writes happen

# Batched Journal Entries

- Operation is safe after journal entry persisted
  - Caller must wait for this to happen
- Small writes are still inefficient
- Accumulate batch until full or max wait time

writer:

```
if there is no current in-memory journal page
    allocate a new page
add my transaction to the current journal page
if current journal page is now full
    do the write, await completion
    wake up processes waiting for this page
else
    start timer, sleep until I/O is done
```

flusher:

```
while true
    sleep()
    if current-in-memory page is due
        close page to further updates
        do the write, await completion
        wake up processes waiting for page
```

# Journal Recovery

- Journal is a circular buffer
  - It can be recycled after old ops have completed
  - Time-stamps distinguish new entries from old
- After system restart
  - Review entire (relatively small) journal
  - Note which ops are known to have completed
  - Perform all writes not known to have completed
    - Data and destination are both in the journal
    - All of these write operations are idempotent
  - Truncate journal and resume normal operation

# Why Does Journaling Work?

- Journal writes much faster than data writes
  - All journal writes are sequential
  - There is no competing head motion
- In normal operation, journal is write-only
  - File system never reads/processes the journal
- Scanning the journal on restart is very fast
  - It is very small (compared to the file system)
  - It can read (sequentially) with huge (efficient) reads
  - All recovery processing is done in memory

# Meta-Data Only Journaling

- Why journal meta-data?
  - It is small and random (very I/O inefficient)
  - It is integrity-critical (huge potential data loss)
- Why not journal data?
  - It is often large and sequential (I/O efficient)
  - It would consume most of journal capacity bandwidth
  - It is less order sensitive (just precede meta-data)
- Safe meta-data journaling
  - Allocate new space for the data, write it there
  - Then journal the meta-data updates

# Log Structured File Systems

- The journal is the file system
  - All inodes and data updates written to the log
  - Updates are Redirect-on-Write
  - In-memory index caches inode locations
- Becoming a dominant architecture
  - Flash file systems
  - Key/value stores
- Issues
  - Recovery time (to reconstruct index/cache)
  - Log defragmentation and garbage collection

# Navigating a Logging File System

- Inodes point at data segments in the log
  - Sequential writes may be contiguous in log
  - Random updates can be spread all over the log
- Updated inodes are added to end of the log
- Index points to latest version of each inode
  - Index is periodically appended to the log
- Recovery
  - Find and recover the latest index
  - Replay all log updates since then



# Redirect on Write

Note: This will work very nicely for flash devices

- Many modern file systems now do this
  - Once written, blocks and inodes are immutable
  - Add new info to the log, and update the index
- The old inodes and data remain in the log
  - If we have an old index, we can access them
  - Clones and snapshots are almost free
- Price is management and garbage collection
  - We must inventory and manage old versions
  - We must eventually recycle old log entries