

File Systems – Naming and
Reliability
CS 111
Summer 2025
Operating System Principles
Peter Reiher

Outline

- How do we name files for user and process use?
 - And translate those names for the OS' use
- How can we ensure more reliable file systems?

Naming in File Systems

- Each file needs some kind of handle to allow us to refer to it
- The OS likes simple numbers as names
 - But those aren't usable by people or 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 systems know 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 *That's what we mean by binding*
 - 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 means 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
 - Graph-based name spaces
 - Can be a strict hierarchical 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?

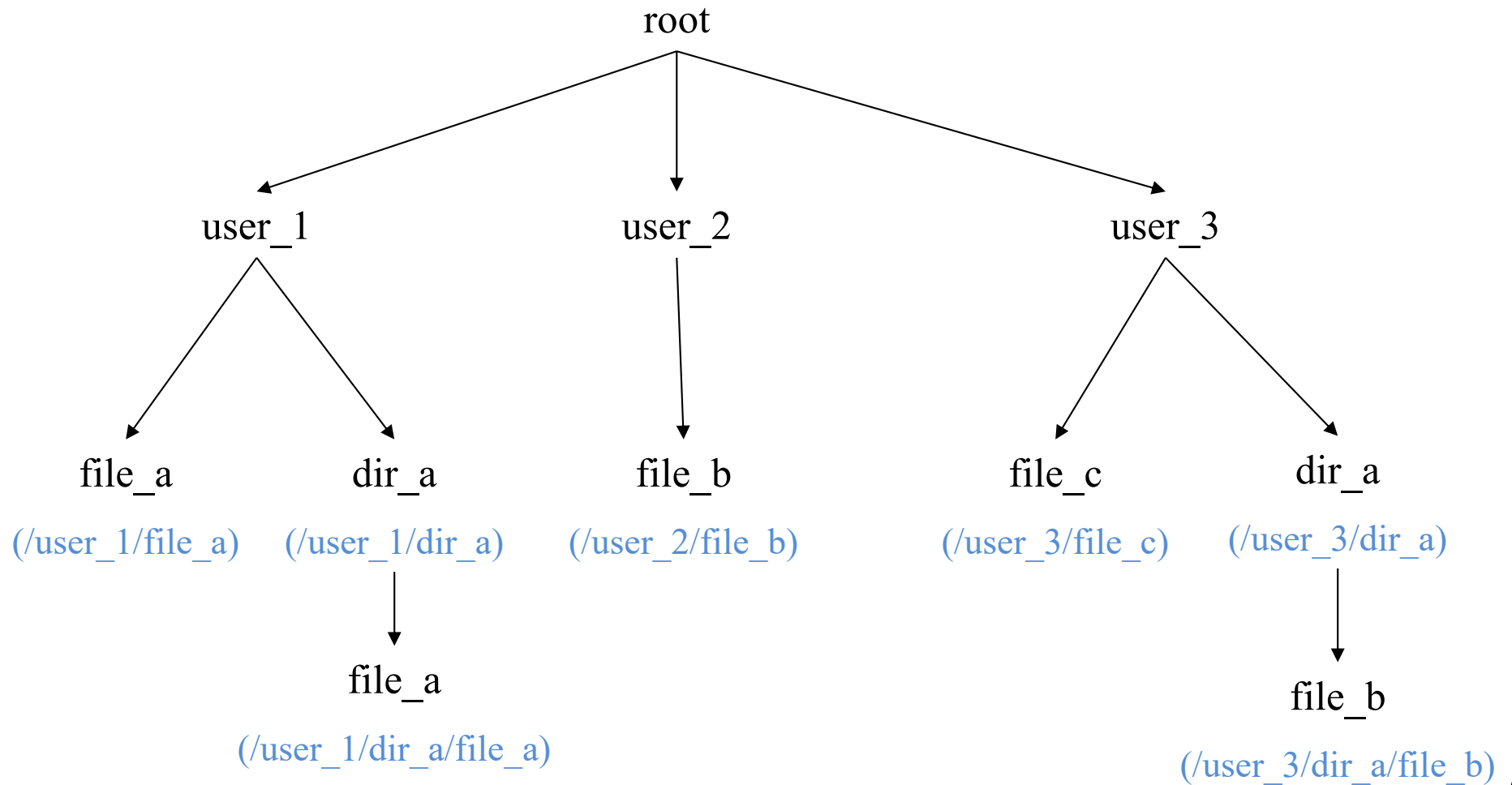
Some Issues in Name Space Structure

- 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
 - The file system depends on the integrity of directories
 - Users can cause writes through special system calls

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!

Useful if we use defaults to attach prefixes to file names.

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

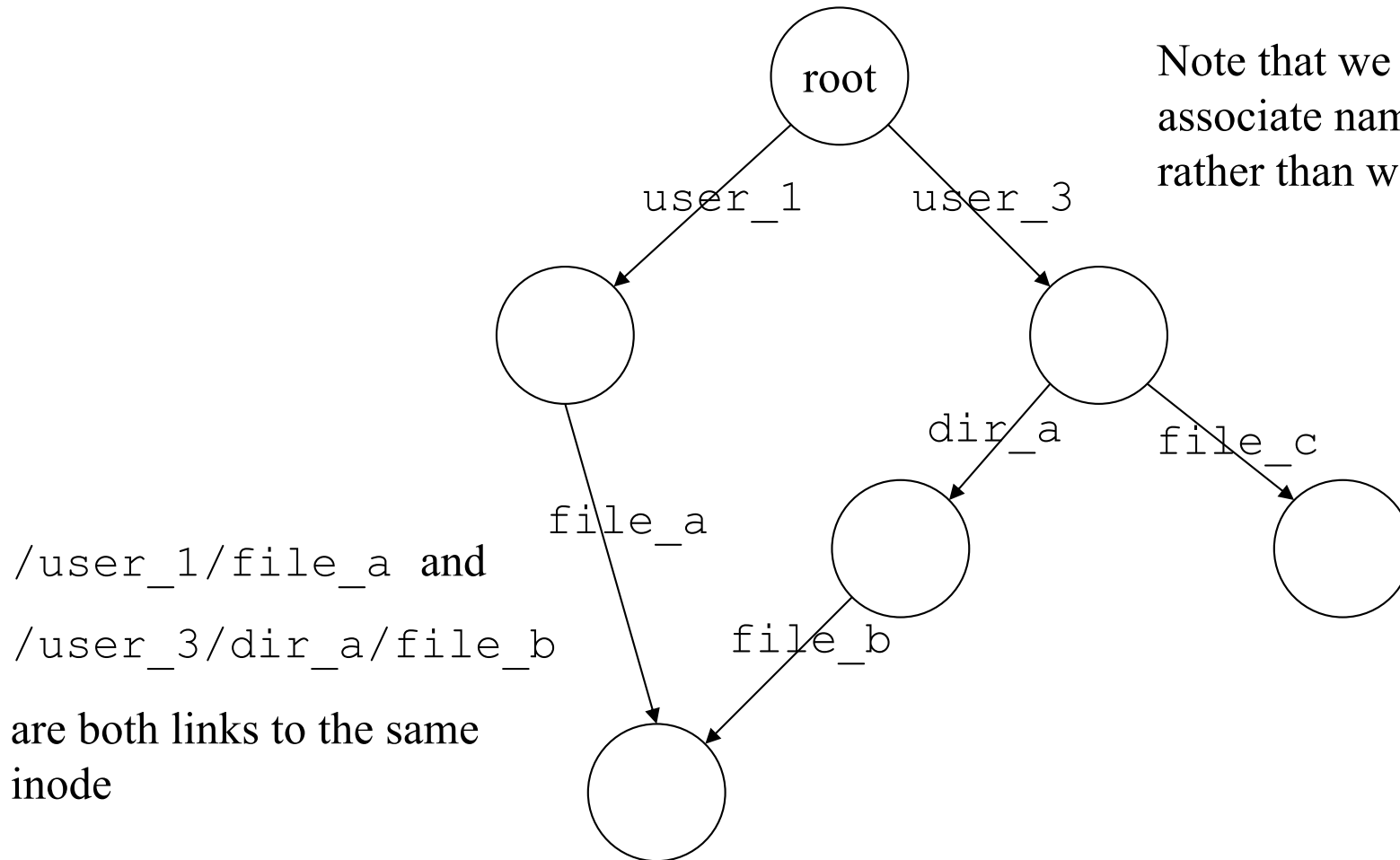
- How do links relate to files?
 - They're the names only
- All other metadata is stored in the file inode
 - File owner sets file protection (e.g., read-only)
- All links provide the same access to the file
 - Anyone with read access to file can create new link
 - But directories are protected files 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

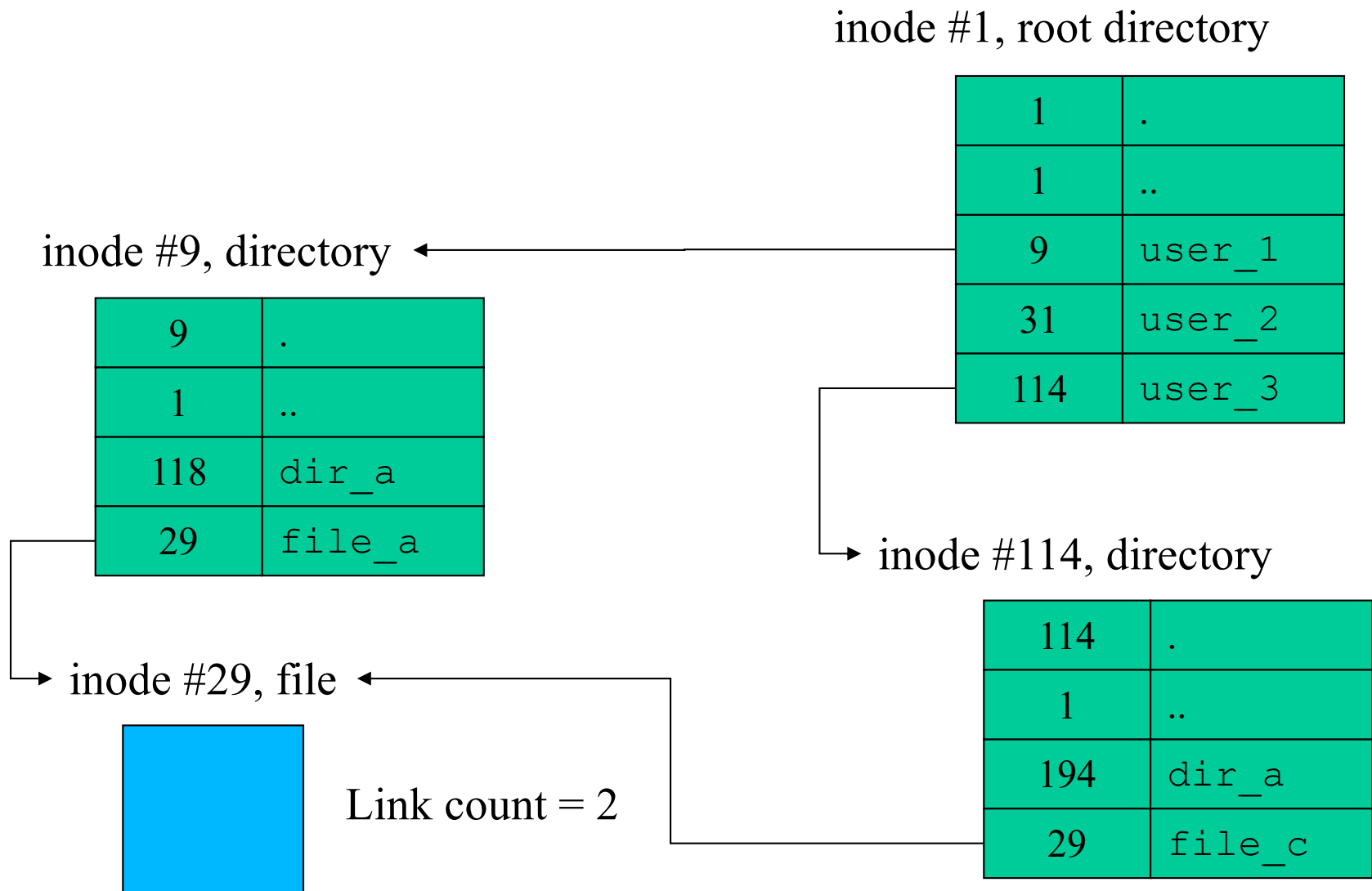
Unix Hard Link Example

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

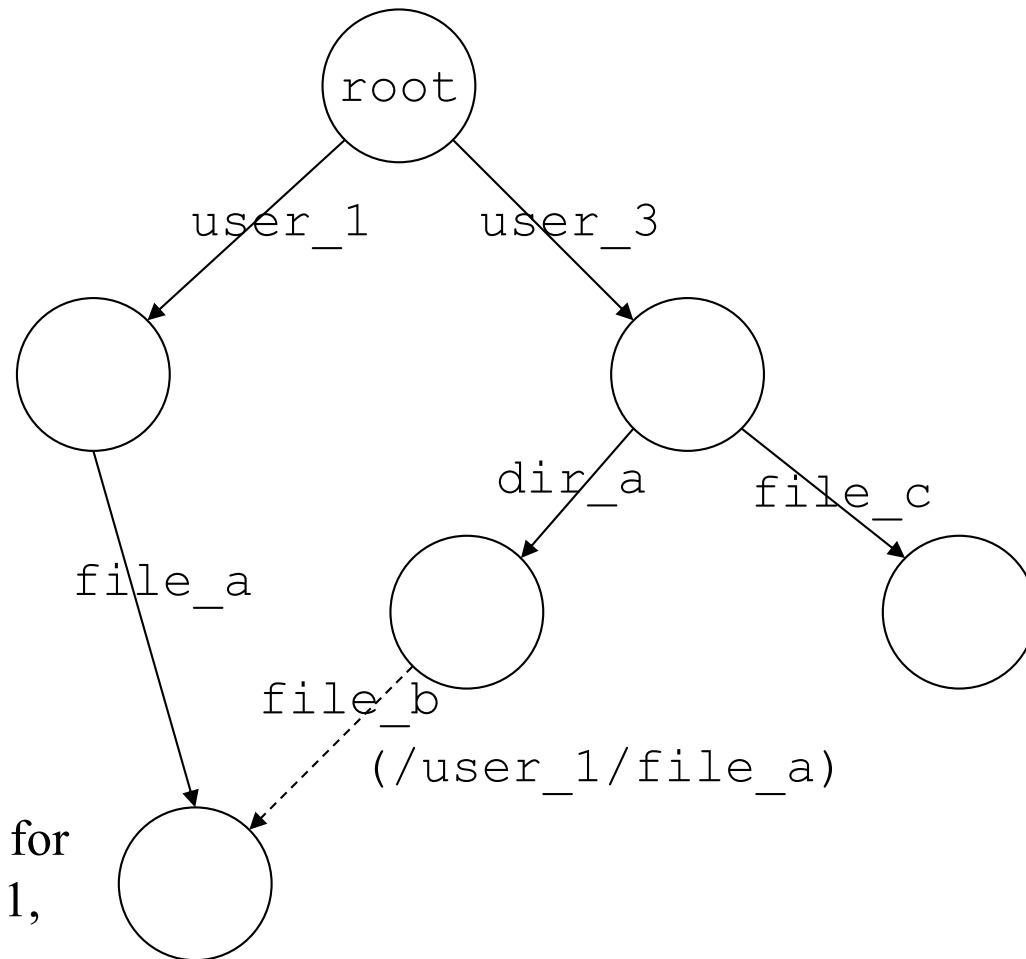
Hard Links, Directories, and Files



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



The link count for
this file is still 1,
though

Symbolic Links, Files, and Directories

inode #1, root directory

1	.
1	..
9	user_1
31	user_2
114	user_3

inode #9, directory

9	.
1	..
118	dir_a
29	file_a

inode #114, directory

114	.
1	..
194	dir_a
46	file_c

inode #29, file



Link count
still equals 1!

inode #46, symlink

/user_1/file_a

File Systems Reliability

- What can go wrong in a file system?
 - System crashes can result in incorrect file system state
 - Data loss due to HW failure or SW error
 - File data is no longer present or readable
 - Some/all of data has been accidentally altered
 - 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

The Core Reliability Problem

- File system writes typically involve multiple operations
 - Not just writing a data block to disk/flash
 - But also writing one or more metadata blocks
 - The inode, the free list, maybe directory blocks
- All must be committed to disk for the write to succeed
- But each block write is a separate hardware operation

Deferred Writes – A Worst Case Scenario

So we
need to
write a
free list
block.

- 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

So we
need to
write an
inode.

Application Expectations When Writing

- Applications make system calls to perform writes
- When system call returns, the application (and user) expect the write to be “safe”
 - Meaning it will persist even if the system crashes
- We can block the writing application until really safe
- But that might block application for quite a while . . .
- Crashes are rare
 - So persistence failure caused by them are also rare
 - Must we accept big performance penalties for occasional safety?

Buffered Writes

- Don't wait for the write to actually be persisted
- Keep track of it in RAM Of course, this is a lie.
- Tell the application the write is complete
- At some later point, actually write to persistent memory
- Up-sides:
 - Less application blocking
 - Deeper and optimizable write queuesParticularly complicated if one user-level write requires multiple disk writes.
- Down-side:
 - What if there's a crash between lying and fixing the lie?

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
- Audit file system for correctness and use redundant info to enable automatic repair
- Used to be standard practice
- No longer practical
 - Checking a 2TB FS at 100MB/second = 5.5 hours
- We need more efficient partial write solutions

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

A Journaling Example

Write to /a/foo

Let's say that's two pages of data



Replacing two existing pages

Put a *start* record in the journal

Plus metadata about where to put the two blocks

Then write the two pages to the journal

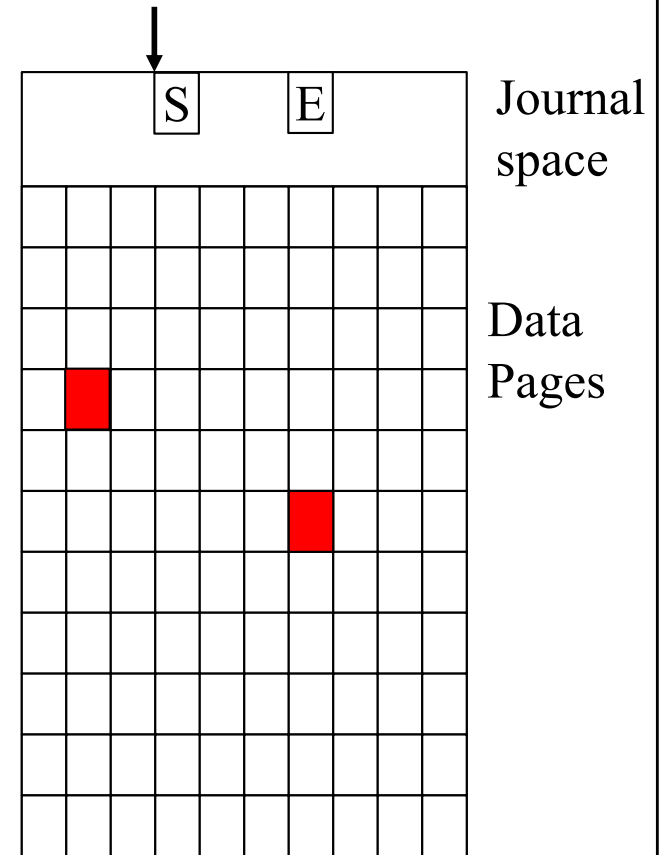
Put an *end* record in the journal

Tell the writing process that it's done

When the OS has spare time, copy the data pages to their true locations

Then get rid of the log entry

Head of journal



The storage device

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 small 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 are much faster than data writes
 - All journal writes are sequential • • • 
- 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
- Journal pages may contain information for multiple files
 - Performed by different processes and users

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

A Metadata Journaling Example

Write to /a/foo

Let's say that's two pages of data



Replacing two existing pages

Put a *start* record in the journal

Plus *new* metadata about where to put the two blocks

Then write the two pages to the new locations

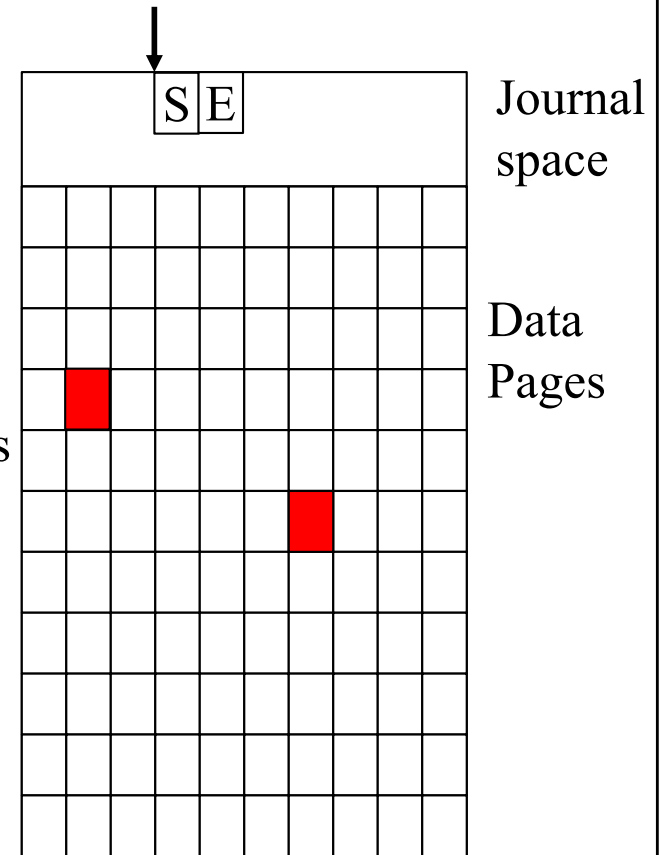
Put an *end* record in the journal

Tell the writing process that it's done

When the OS has spare time, update the file's metadata to show the new page locations

Then free the old data pages and get rid of the log entry

Head of journal



The storage device

Log Structured File Systems

- The journal is the file system
 - All inodes and data updates written to the log
 - Updates are Redirect-on-Write
 - An 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

Don't overwrite the old data.

Write it elsewhere and change the metadata (inode) pointer to it.

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

A Log Structured File System Example

The current head of the log

Write to /a/foo

Let's say that's two pages of data



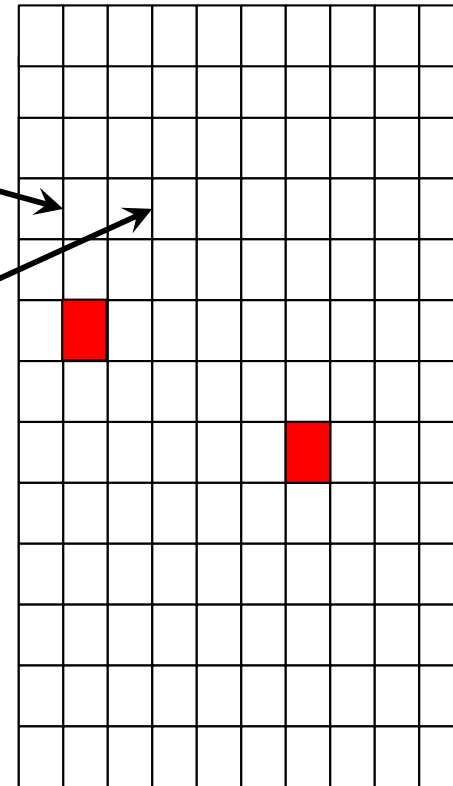
Replacing two existing pages

Write the new pages to the head of the log

Move the head of the log pointer

*But how can we find the
new pages of foo?*

*Foo's inode still points to
the old version of the pages*



The storage device

Continuing the Example

Foo's inode would still point to the old versions

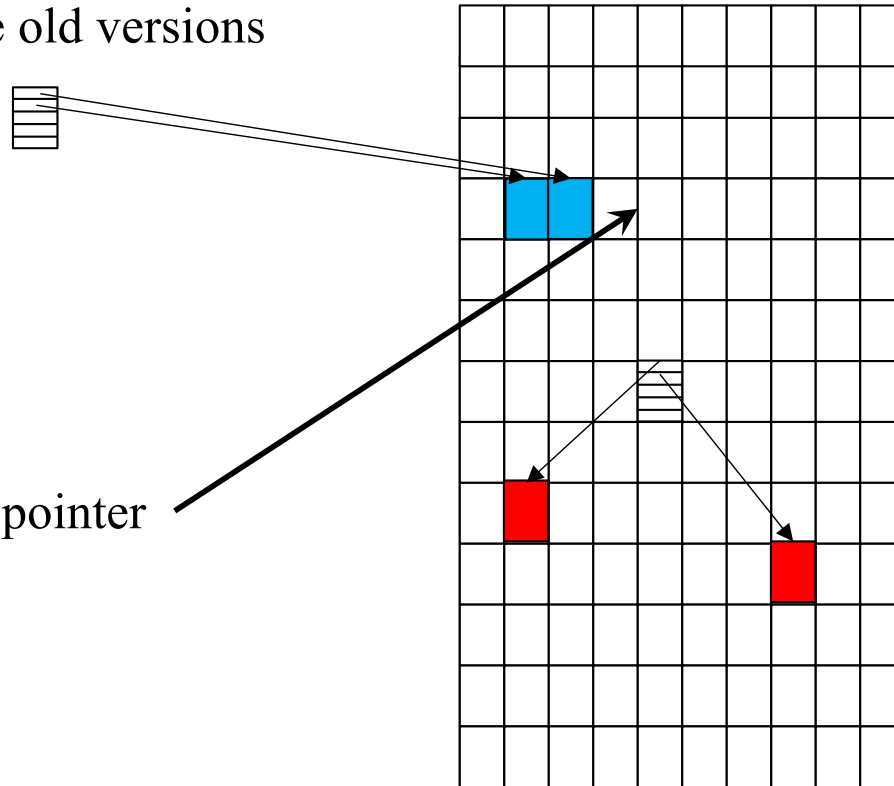
So create a new inode for foo

Pointing to the new pages

Where do we put the new inode?

In the log!

And we move the head of the log pointer



The storage device

But . . .

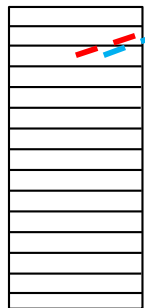
How do we find the inode for this file?

The `/a` directory's entry for `foo` points to the old inode

Traditional Linux file systems keep all inodes in one part of the disk

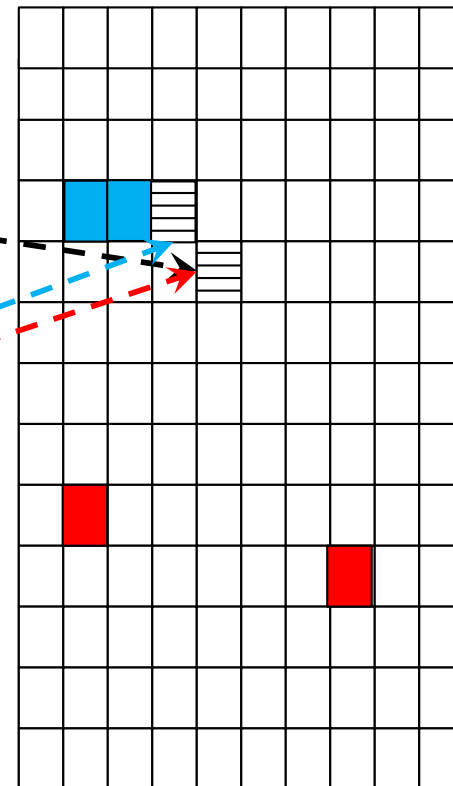
LFS scatters them all over the log

So use an inode map to keep track of them



The old location

The new location



The storage device

One Thing Leads to Another

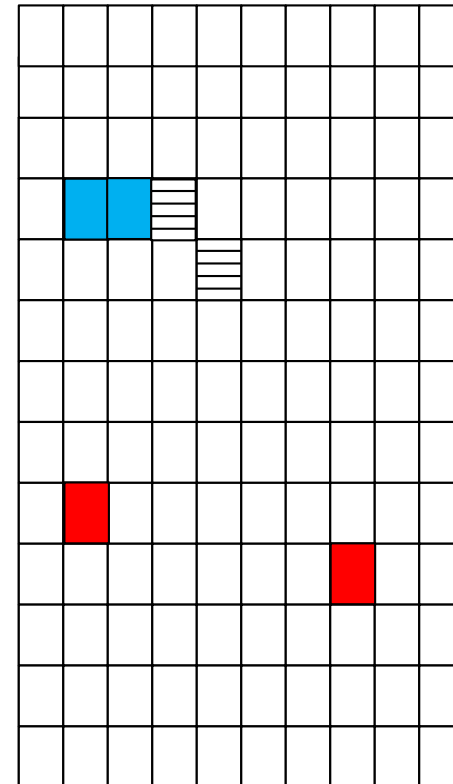
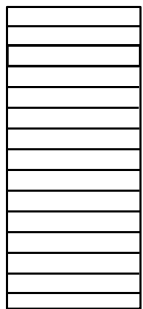
The inode map better be persistent

So we need to store it on disk

Where?

How about in the log?

Maybe just a relevant piece of it, though



The storage device

But how do we find the inode
map's pieces?

Read the book to find out

Conclusion

- We must have some solution to how to manage space on a persistent device
- We must have some scheme for users to name their files
 - And for the file system to match names to locations
- Performance and reliability are critical for file systems
- How the file system works under the covers matters a lot for those properties