Linux Metadata

```
struct stat {
  dev_t       st_dev;     /* device */
  ino_t       st_ino;     /* inode */
  mode_t      st_mode;    /* protection */
  nlink_t     st_nlink;   /* number of hard links
  uid_t       st_uid;     /* user ID of owner */
  gid_t       st_gid;     /* group ID of owner */
  dev_t       st_rdev;    /* device type (if inoc
  off_t       st_size;    /* total size, in bytes
  blksize_t   st_blksize; /* blocksize for filesy
  blkcnt_t    st_blocks;  /* number of blocks all
  time_t      st_atime;   /* time of last access
  time_t      st_mtime;   /* time of last modific
  time_t      st_ctime;   /* time of last status
};
```

Where is Metadata Stored?

- In the file?
- In the directory entry?
- Elsewhere?
- Split?

Metadata in the File

- (Sort of) done by Apple: resource and data forks
- Not very portable — when you copy the file to/from other systems, what happens to the metadata?
- No standardized metadata exchange format

Metadata in the Directory

- Speeds access to metadata
- Makes hard links difficult — need to keep copies of the metadata synchronized
- Makes directories larger; often, one doesn't need the metadata
- Many newer systems keep at least a few bits of metadata in the directory, notably file type — knowing if something is or isn't a directory speeds up tree walks considerably

Crash Recovery

- Must ensure that file systems are in a consistent state after a system crash
- Example: don't write out directory entry before the metadata
- Example: File systems are generally trees, not graphs; make sure things always point somewhere sane
- What if the file has blocks but the freelist hasn't been updated?
- Principle: order writes to ensure that the disk is always in a *safe* state

Repairing Damage

- At boot-time, run a consistency checker except after a clean shutdown
- Example: `fsck` (Unix) and `scandisk` (Windows)
- Force things to a safe state; move any allocated but unreferenced blocks and files to a recovery area

Log-Structured File Systems

- Instead of overwriting data, append the changes to a journaling area
- The file system is thus always consistent, as long as writes are properly ordered.
- Hmm — is that a reasonable assumption?

Modern Disks

- Modern disks do a lot of buffering
- Cache size on new Seagate drives: 2-16M bytes
- The drive will reorder writes to optimize seek times
- If a bad block has been relocated, you can't even predict when seeks will occur; only the drive knows
- What if the power fails when data is buffered?

The Unix Filesystem

The Unix Filesystem

- Let's take a high-level look at the Unix file system
- Though details differ (a lot) for, say, Windows, at a high level things are pretty similar
- We'll discuss the actual code paths next time
- Note: all modern operating systems support multiple file system types; differences hidden by abstraction layer.

From the Process

- The process has two crucial directory attributes, the current root and the current working directory
- Paths that start with a / (known as *absolute paths*) start from the current root; those that do not start with a / (*relative paths*) start from the current working directory
- (Roots other than the real root are a security mechanism; we'll discuss that later in the semester.)

Directories

- On some Unix systems, directories can be read like any other file with `read()`; on Linux, you must (and on other Unix systems you should) use `readdir()`
- A directory entry consists of a variable-length name and an *i-node* number
- (What's an i-node?)
- By convention, on most Unix systems the first two entries in a directory are `.` and `..` — the current and parent directories
- Don't count on them being there; they're not guaranteed by the spec!

Finding a File

- Find the next component in the pathname
- Read the current directory looking for it
- If there's another component to the path name and this element is a directory, repeat, starting from this element
- When we're done, the result is an i-node number

. and ..

- Note how . and .. work
- . has the i-node number of the current directory — you start again from this node for the next component
- It's just another directory; to (this part of) the kernel, there's nothing special about it
- The same is true for .. — it "happens" to point up a level in the directory tree.
- Following a search path does not rely on the directory structure being a tree! That's primarily needed for orderly tree walks.
- (Mental exercise: symbolic links do introduce the possiblity of loops. How can this be dealt with?)

I-Nodes

- What's an i-node?
- An i-node holds most of the metadata for a Unix file — on classic Unix, it holds all but the i-node number itself
- The i-list is a disk-resident array i-nodes
- The i-node number is just an index into this array
- Looked at another way, a directory entry maps a name to an array entry
- Files are actually described by i-nodes, not by names

What's in an I-Node?

- All of the fields from the `stat` structure
- A few other pieces of user-settable metadata (flags)
- Disk block information — where on disk the file resides?
- How many blocks is enough?
- Put another way, how big can a file be?

Disk Blocks in the I-Node

- If we have a small array, we limit the size of a file too much
- If we have a large array, we waste space in the i-list, because most files aren't huge
- We have a modest-size array of block addresses, followed by the address of an *indirect block*
- The indirect block is an array of disk addresses
- Hmm — suppose the i-node points to ten 4K blocks, followed by an indirect block. The maximum size of a file is then $4096 \times 10 + (4096/4) \times 4096$
- That's 4,235,264 bytes – not nearly big enough

Multiple Layers of Indirection

- "Any problem in software can be solved by adding another layer of indirection" —*David Wheeler*
- The second indirect block is a *double indirect block*
- That gives us $4096 \times 10 + (4096/4) \times 4096 + ((4096/4) \times 4096/4) \times 4096$ bytes — about 4G
- Is that enough? Some systems today have triple indirect blocks...
- Metanote: PDP-11 Unix had triple indirect blocks; when file system blocks became 4K (more or less), there was no need for them. But disks and files grew bigger...

Opening a File

- When a file is opened, the i-node is read into memory (if necessary) and its *reference count* is incremented
- A *file table entry* is created for it
- The index in the file table is passed back to the application as the file descriptor
- Virtually all operating systems have this notion — a *file handle* — that is a short way of referring to an open file.
- More complex for special files — wait a few days

Creating a File

- See if there's a directory entry.
- If there is, it's like opening a file (but you may have to truncate it)
- If there's no entry, create one.
- That involves writing to the directory, which is a lot like writing to a regular file
- It also involves finding and allocating a free i-node
- Directory entries are free if the i-node number is 0; i-nodes are free if the link count is 0

Reading a File

- Convert the current byte offset to a block number
- Read that block from the disk
- Pass the proper bytes back to the user
- Update the current byte offset
- Optional: if access to the file appears to be sequential, start — but don't wait for — the read of the next block
- Get it in the buffer cache ahead of use, to improve performance

Writing a File

- Are we writing to the middle of a block?
- If so, read that block in
- If not, allocate a new block from the freelist and add it to the i-node
- Copy the data from the user to a buffer
- Mark that buffer dirty, so that it will (eventually) be written out
- Update the file offset pointer

Seek

- Simply change the current byte offset
- Does not actually move the disk arm
- Doing that is probably pointless on a multitasking system

Closing a File

- Decrement the i-node's reference count
- Delete the file table entry
- If the i-node's link count is 0, the file has been deleted; see below
- Everything else is automatic

Linking to a File

- Create a new directory entry
- But — the i-node number is the number of the existing file
- Increment the i-node's link count

Unlinking a File

- Decrement the link-count
- If the link-count is still non-zero, the file has other names; do nothing more
- If the link-count is now 0 (and the in-memory reference count is 0), the file is being deleted
- Return all of its blocks to the free list
- (Note: you can unlink an open file; it isn't actually deleted until it's closed. Query: what happens if the system crashes with a file in that state?)

Updating Metadata

- When a file is read, the *access time* needs to be updated
- When a file is written, the *modified time* needs to be updated
- If the user changes things like file permissions, make the appropriate changes
- Mark the in-memory i-node as dirty
- Also update the i-node change time
- Periodically, dirty i-nodes are rewritten to disk

Creating Directories

- Similar to creating a file
- But — the kernel first writes the `.` and `..` entries
- Increment the link count in the parent directory — `..` points to it
- (All writes to directories are done by specialized system calls; user programs cannot write directories directly via `write()` on any modern Unix)

Deleting Directories

- First make sure the directory is empty except for . and ..
- Then delete it the same way a normal file is deleted
- But – the link count in the parent directory is decremented
- It's possible to delete the current working directory, or even its parent!

Renaming

- Create a link with the new name
- Remove the link with the old name
- But — it must be done in the kernel, since the first step involves creating a hard link to a directory

I’ve Glossed Over Stuff

- There’s a fair mount of locking going on, to ensure consistency
- I have not mentioned proper order of operations to ensure file system consistency
- There are a variety of less-important system calls
- File permissions
- Special files and symbolic links

Components of a File System

- Boot record, superblock
- i-list
- Freelist
- Data area

Boot Record and Superblock

- The boot record is at the start of the disk; it's used by the BIOS for booting (called the *Master Boot Record* (MBR) on PCs)
- This area also stores the *disk label* — information on how the disk is partitioned
- Next is the *superblock* — contains essential file system parameters
- Among other things: how to find the i-list; the "clean shutdown" flag, to indicate that the file system is believed to be in a consistent state

The I-List

- Once, the i-list was an array of blocks just after the superblock
- Today, it's distributed — a piece of the i-list is in each *cylinder group*
- A cylinder group contains a portion of the i-list, a freelist for blocks within the cylinder group, and a data area
- Newly-created files get their initial block allocations within the cylinder group; later blocks are allocated in groups in other cylinder groups
- What is the purpose of cylinder groups?
- Locality of reference — try to avoid long seeks

Data Area

- Most of the file is allocated in blocks of 4-8K bytes
- Left-over pieces of the file are stored in *fragments*, which are composed of 512-byte blocks
- Dual block size saves RAM and disk space for large files, but doesn't waste too much for short files

The Windows FAT File System

- Limited-size root directory
- 8.3 file names
- Metadata in the directory entry
- Directory points to FAT table entry

File Allocation Table

- The FAT keeps track of allocated blocks
- Each entry in the FAT — on disk and in RAM — is just a pointer to the next block
- Implements a linked list of blocks, without the need to read each block
- Maximum file size limited by number of bits in a disk block address — current is using 28-bit addresses

Supporting Long Names

- For Windows 98, they wanted to support longer names than 8.3 permitted
- But — must maintain substantial name compatibility with older versions of Windows and DOS
- First: use recognizable part of long name for 8.3 version
- Second: create fake, invalid directory entries that precede the real one
- DOS will ignore them, because they appear invalid
- Have a checksum in case the real, short-name version of the file is deleted while running DOS

Dumping a Disk

- Must have a way to dump and restore disks — some people make backups
- More than — we want a way to to create *incremental* dumps: all files changed since the last dump
- Unix has multiple levels of backup: 0 is everything; 1 is everything since the last level 1; 2 is everything since the last level 1; etc.
- Can select files by date modified or by a "dirty" bit
- Some systems have a way to exclude some files from being dump (i.e., swap files or very sensitive files)

Dump Strategies

- Could do a physical image dump
- Safe and simple, but wasteful — dumps empty blocks, can't do incremental dump, might try to dump bad blocks, etc.
- only restorable to disk with identical geometry
- Instead — dump a filesystem

Mapping a Filesystem

- Must map the file system to see what files must be dumped
- For incremental dumps, must be able to "dump" deletions — those are changes to the parent directory
- Must also dump all parent directories, up to the root, of any dumped files
- Dump selection is based on metadata (and metadata must be dmped)

Dumping a Filesystem

- The actual dump can't go through the file system on Unix; want to avoid dumping file system "holes"
- The actual dump file is based on i-node numbers, not file names; the file names are in the dumped directories
- Restores can be done through the file system
- To do incremental restores, must restore each level in sequence, to build the proper directory structure