# Case study: ext2 FS



# The ext2 file system

- Second Extended Filesystem
  - The main Linux FS before ext3
  - Evolved from Minix filesystem (via “Extended Filesystem”)
- Features
  - Block size (1024, 2048, and 4096) configured at FS creation
  - inode-based FS
  - Performance optimisations to improve locality (from BSD FFS)
- Main Problem: unclean unmount →**`e2fsck`**
  - Ext3fs keeps a journal of (meta-data) updates
  - Journal is a file where updates are logged
  - Compatible with ext2fs

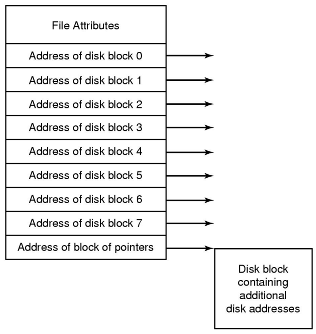


# Recap: i-nodes

- Each file is represented by an inode on disk
- Inode contains all of a file’s metadata
  - Access rights, owner,accounting info
  - (partial) block index table of a file
- Each inode has a unique number
  - System oriented name
  - Try ‘ls –i’ on Unix (Linux)
- Directories map file names to inode numbers
  - Map human-oriented to system-oriented names



# Recap: i-nodes






# Ext2 i-nodes

| |
|---|
| mode |
| uid |
| gid |
| atime |
| ctime |
| mtime |
| size |
| block count |
| reference count |
| direct blocks (12) |
| single indirect |
| double indirect |
| triple indirect |

- Mode
  - Type
    - Regular file or directory
  - Access mode
    - rwxrwxrwx
- Uid
  - User ID
- Gid
  - Group ID



# Inode Contents

| |
|---|
| mode |
| uid |
| gid |
| atime |
| ctime |
| mtime |
| size |
| block count |
| reference count |
| direct blocks (12) |
| single indirect |
| double indirect |
| triple indirect |

- atime
  - Time of last access
- ctime
  - Time when file was created
- mtime
  - Time when file was last modified

mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
single indirect
double indirect
triple indirect

# Inode Contents

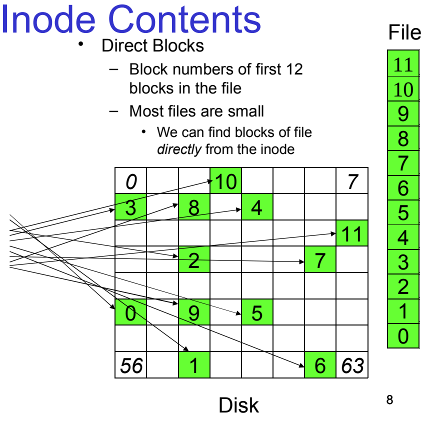
- Size
  - Size of the file in bytes
- Block count
  - Number of disk blocks used by the file.
- Note that number of blocks can be much less than expected given the file size
  - Files can be sparsely populated
    - E.g. write(f,"hello"); lseek(f, 1000000); write(f, "world");
    - Only needs to store the start an end of file, not all the empty blocks in between.
      - Size = 1000005
      - Blocks = 2 + overheads



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
40,58,26,8,12,
44,62,30,10,42,3,21
single indirect
double indirect
triple indirect

# Inode Contents

- Direct Blocks
  - Block numbers of first 12 blocks in the file
  - Most files are small
    - We can find blocks of file *directly* from the inode






# Problem

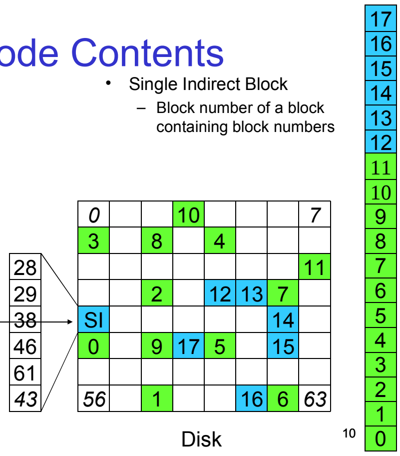
- How do we store files greater than 12 blocks in size?
  - Adding significantly more direct entries in the inode results in many unused entries most of the time.



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
40,58,26,8,12,
44,62,30,10,42,3,21
single indirect: 32
double indirect
triple indirect

# Inode Contents

- Single Indirect Block
  - Block number of a block containing block numbers






# Single Indirection

- Requires two disk access to read
  - One for the indirect block; one for the target block
- Max File Size
  - Assume 1Kbyte block size, 4 byte block numbers
  - 12 * 1K + 1K/4 * 1K = 268 Kbytes
- For large majority of files (< 268 K), given the inode, only one or two further accesses required to read any block in file.



mode
uid
gid
atime
ctime
mtime
size
block count
reference count
direct blocks (12)
40,58,26,8,12,
44,62,30,10,42,3,21
single indirect: 32
double indirect
triple indirect

# Inode Contents

- Double Indirect Block
  - Block number of a block containing block numbers of blocks containing block numbers

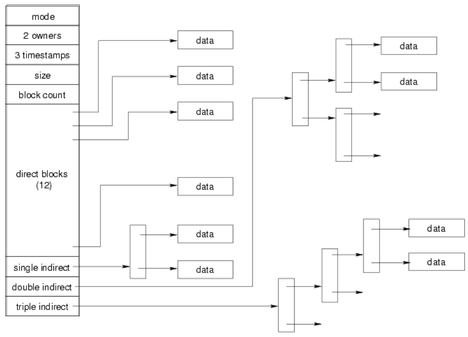
| |
|---|
| mode |
| uid |
| gid |
| atime |
| ctime |
| mtime |
| size |
| block count |
| reference count |
| direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21 |
| single indirect: 32 |
| double indirect |
| triple indirect |

# Inode Contents

- Double Indirect Block
  - Block number of a block containing block numbers of blocks containing block numbers
- Triple Indirect
  - Block number of a block containing block numbers of blocks containing block numbers of blocks containing block numbers ☺



# UNIX Inode Block Addressing Scheme






# Max File Size

- Assume 4 bytes block numbers and 1K blocks
- The number of addressable blocks
  - Direct Blocks = 12
  - Single Indirect Blocks = 256
  - Double Indirect Blocks = 256 * 256 = 65536
  - Triple Indirect Blocks = 256 * 256 * 256 = 16777216
- Max File Size
  - 12 + 256 + 65536 + 16777216 = 16843020 blocks = 16 GB



# Where is the data block number stored?

- Assume 4K blocks, 4 byte block numbers, 12 direct blocks
- A 1 byte file produced by

  lseek(fd, 1048576, SEEK_SET) /* 1 megabyte */

  write(fd, “x”, 1)
- What if we add

  lseek(fd, 5242880, SEEK_SET) /* 5 megabytes */

  write(fd, “x”, 1)



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| ? | Single-indirect block |
| | Double-indirect blocks |



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| ? | Double-indirect blocks |

# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=?



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=1048576/4096=256



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=1048576/4096=256

Block number=256 ==> index in the single-indirect block=?



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=1048576/4096=256

Block number=256 ==> index in the single-indirect block=256-12=244

Address = 5242880 ==> block number=5242880/4096=1280

Block number=1280 ==> double-indirect block number=?



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=1048576/4096=256

Block number=256 ==> index in the single-indirect block=256-12=244

Address = 5242880 ==> block number=5242880/4096=1280

Block number=1280 ==> double-indirect block number=(1280-1036)/1024=244/1024=0

Index in the double indirect block=?



# Solution

| block # | location |
|---|---|
| 0 through 11 | Direct block |
| 12 through (11 + 1024 = 1035) | Single-indirect block |
| 1036 through (1035+1024*1024 = 1049611) | Double-indirect blocks |

Address = 1048576 ==> block number=1048576/4096=256

Block number=256 ==> index in the single-indirect block=256-12=244

Address = 5242880 ==> block number=5242880/4096=1280

Block number=1280 ==> double-indirect block number=(1280-1036)/1024=244/1024=0

Index in the double indirect block=244

# Some Best and Worst Case Access Patterns

Assume Inode already in memory

- To read 1 byte
  - Best:
    - 1 access via direct block
  - Worst:
    - 4 accesses via the triple indirect block
- To write 1 byte
  - Best:
    - 1 write via direct block (with no previous content)
  - Worst:
    - 4 reads (to get previous contents of block via triple indirect) + 1 write (to write modified block back)



# Worst Case Access Patterns with Unallocated Indirect Blocks

- Worst to write 1 byte
  - 4 writes (3 indirect blocks; 1 data)
  - 1 read, 4 writes (read-write 1 indirect, write 2; write 1 data)
  - 2 reads, 3 writes (read 1 indirect, read-write 1 indirect, write 1; write 1 data)
  - 3 reads, 2 writes (read 2, read-write 1; write 1 data)
- Worst to read 1 byte
  - If reading writes a zero-filled block on disk
    - Worst case is same as write 1 byte
  - If not, worst-case depends on how deep is the current indirect block tree.

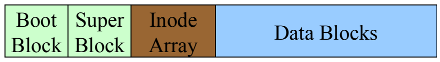


# Inode Summary

- The inode contains the on disk data associated with a file
  - Contains mode, owner, and other bookkeeping
  - Efficient random and sequential access via *indexed allocation*
  - Small files (the majority of files) require only a single access
  - Larger files require progressively more disk accesses for random access
    - Sequential access is still efficient
  - Can support really large files via increasing levels of indirection



# Where/How are Inodes Stored




- System V Disk Layout (s5fs)
  - Boot Block
    - contain code to bootstrap the OS
  - Super Block
    - Contains attributes of the file system itself
      - e.g. size, number of inodes, start block of inode array, start of data block area, free inode list, free data block list
  - Inode Array
  - Data blocks



# Some problems with s5fs

- Inodes at start of disk; data blocks end
  - Long seek times
    - Must read inode before reading data blocks
- Only one superblock
  - Corrupt the superblock and entire file system is lost
- Block allocation was suboptimal
  - Consecutive free block list created at FS format time
    - Allocation and de-allocation eventually randomises the list resulting the random allocation
- Inodes also allocated randomly
  - Directory listing resulted in random inode access patterns



# Berkeley Fast Filesystem (FFS)

- Historically followed s5fs
  - Addressed many limitations with s5fs
  - ext2fs mostly similar

# Layout of an Ext2 FS

| Boot Block | Block Group 0 | …. | Block Group *n* |
|---|---|---|---|

- Partition:
  – Reserved boot block,
  – Collection of equally sized *block groups*
  – All block groups have the same structure



# Layout of a Block Group

| Super Block | Group Descrip-tors | Data Block Bitmap | Inode Bitmap | Inode Table | Data blocks |
|---|---|---|---|---|---|
| 1 blk | *n* blks | 1 blk | 1 blk | *m* blks | *k* blks |

- *Replicated* super block
  – For e2fsck
- Group descriptors
- Bitmaps identify used inodes/blocks
- All block groups have the same number of data blocks
- Advantages of this structure:
  – Replication simplifies recovery
  – Proximity of inode tables and data blocks (reduces seek time)



# Superblocks

- Size of the file system, block size and similar parameters
- Overall free inode and block counters
- Data indicating whether file system check is needed:
  – Uncleanly unmounted
  – Inconsistency
  – Certain number of mounts since last check
  – Certain time expired since last check
- Replicated to provide redundancy to aid recoverability



# Group Descriptors

- Location of the bitmaps
- Counter for free blocks and inodes in this group
- Number of directories in the group



# Performance considerations

- EXT2 optimisations
  – Block groups cluster related inodes and data blocks
  – Read-ahead for directories
    - For directory searching
  – Pre-allocation of blocks on write (up to 8 blocks)
    - 8 bits in bit tables
    - Better contiguity when there are concurrent writes
- FFS optimisations
  – Aim to store files within a directory in the same group



# Thus far…

- Inodes representing files laid out on disk.
- Inodes are referred to by number!!!
  – How do users name files? By number?

# Ext2fs Directories

| inode | rec_len | name_len | type | name… |
|---|---|---|---|---|

- Directories are files of a special type
  - Consider it a file of special format, managed by the kernel, that uses most of the same machinery to implement it
    - Inodes, etc…
- Directories translate names to inode numbers
- Directory entries are of variable length
- Entries can be deleted in place
  - inode = 0
  - Add to length of previous entry
  - use null terminated strings for names



# Ext2fs Directories

- “f1” = inode 7
- “file2” = inode 43
- “f3” = inode 85

| | |
|---|---|
| 7 | Inode No |
| 12 | Rec Length |
| 2 | Name Length |
| ‘f’ ‘1’ 0 0 | Name |
| 43 | |
| 16 | |
| 5 | |
| ‘f’ ‘i’ ‘l’ ‘e’ | |
| ‘2’ 0 0 0 | |
| 85 | |
| 12 | |
| 2 | |
| ‘f’ ‘3’ 0 0 | |
| 0 | |



# Hard links

- Note that inodes can have more than one name
  - Called a *Hard Link*
  - Inode (file) 7 has three names
    - “f1” = inode 7
    - “file2” = inode 7
    - “f3” = inode 7

| | |
|---|---|
| 7 | Inode No |
| 12 | Rec Length |
| 2 | Name Length |
| ‘f’ ‘1’ 0 0 | Name |
| 7 | |
| 16 | |
| 5 | |
| ‘f’ ‘i’ ‘l’ ‘e’ | |
| ‘2’ 0 0 0 | |
| 7 | |
| 12 | |
| 2 | |
| ‘f’ ‘3’ 0 0 | |
| 0 | |



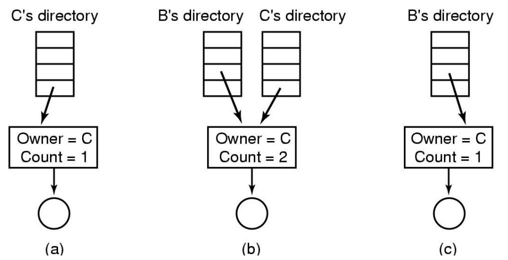
# Inode Contents

| |
|---|
| mode |
| uid |
| gid |
| atime |
| ctime |
| mtime |
| size |
| block count |
| reference count |
| direct blocks (12) 40,58,26,8,12, 44,62,30,10,42,3,21 |
| single indirect: 32 |
| double indirect |
| triple indirect |

- We can have many names for the same inode.
- When we delete a file by name, i.e. remove the directory entry (link), how does the file system know when to delete the underlying inode?
  - Keep a *reference count* in the inode
    - Adding a name (directory entry) increments the count
    - Removing a name decrements the count
    - If the reference count == 0, then we have no names for the inode (it is unreachable), we can delete the inode (underlying file or directory)



# Hard links




(a) Situation prior to linking

(b) After the link is created

(c)After the original owner removes the file

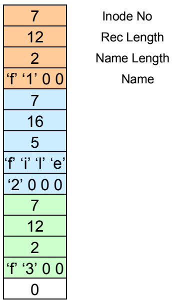


# Symbolic links

- A symbolic link is a file that contains a reference to another file or directory
  - Has its own inode and data block, which contains a path to the target file
  - Marked by a special file attribute
  - Transparent for some operations
  - Can point across FS boundaries

# Ext2fs Directories

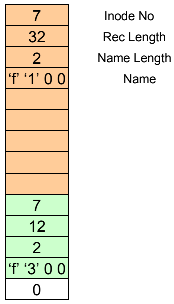
- Deleting a filename
  - rm file2






# Ext2fs Directories

- Deleting a filename
  - rm file2
- Adjust the record length to skip to next valid entry



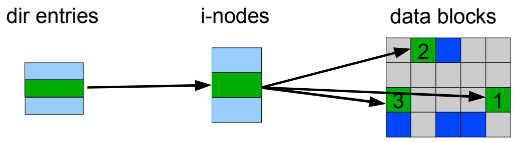



# FS reliability

- Disk writes are buffered in RAM
  - OS crash or power outage ==> lost data
  - Commit writes to disk periodically (e.g., every 30 sec)
  - Use the `sync` command to force a FS flush
- FS operations are non-atomic
  - Incomplete transaction can leave the FS in an inconsistent state



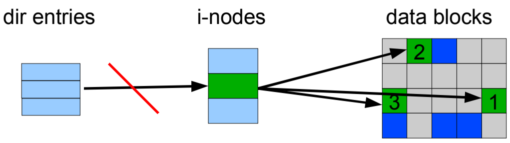
# FS reliability




- Example: deleting a file
  1. Remove the directory entry
  2. Mark the i-node as free
  3. Mark disk blocks as free



# FS reliability

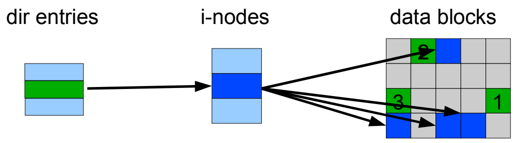



- Example: deleting a file
  1. Remove the directory entry--> crash
  2. Mark the i-node as free
  3. Mark disk blocks as free

The i-node and data blocks are lost



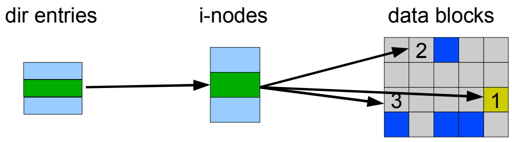
# FS reliability




- Example: deleting a file
  1. Mark the i-node as free --> crash
  2. Remove the directory entry
  3. Mark disk blocks as free

The dir entry points to the wrong file

# FS reliability




- Example: deleting a file
  1. Mark disk blocks as free --> crash
  2. Remove the directory entry
  3. Mark the i-node as free

The file randomly shares disk blocks with other files



# FS reliability

- e2fsck
  - Scans the disk after an unclean shutdown and attempts to restore FS invariants
- Journaling file systems
  - Keep a journal of FS updates
  - Before performing an atomic update sequence, write it to the journal
  - Replay the last journal entries upon an unclean shutdown
  - Example: ext3fs