The Second Extended File-System (ext2)

Ohad Rodeh

Outline

■ Ext2 is a Linux file-system

■ The material for this lecture was taken from:

`http://en.tldp.org/LDP/khg/HyperNews/get/fs/ext2intro.html`

■ Ext2 is a relatively simple file-system, that is why we start with it.

Relationship to FFS

- Ext 2 heavily borrows from FFS.

- This original FFS paper is: **A Fast File System for UNIX** Marshall K. McKu sick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, 1984

- All Unix file-systems, to this day, are based to some extent on FFS.

- The main difference between FFS and Ext2 is that FFS uses fragments

File system history

FFS

Ext2

LFS

WAFL
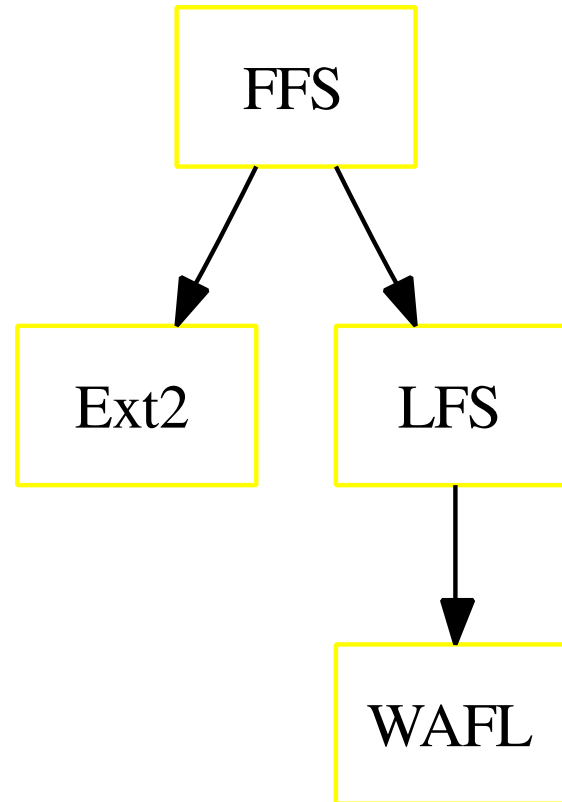
File systems on Unix

- Every Unix file-system implements a basic set of common concepts derived from the Unix operating system

- Files are represented by i-nodes

- Directories are simply files containing a list of entries

- Devices can be accessed by requesting I/O on special files

I-nodes

- Each file is represented by an i-node

- Each i-node contains the description of the file

  1. file type

  2. access rights

  3. owners

  4. timestamps: mtime, ctime, atime.

  5. Size and length

  6. pointers to data blocks

I-nodes II

- The addresses of data blocks allocated to a file are stored in its i-node.

- The size of an i-node on disk, depending on the file-system, can be is 128,256,512 bytes.

- It is normally no larger than a sector (512bytes) to ensure atomic writes to it.
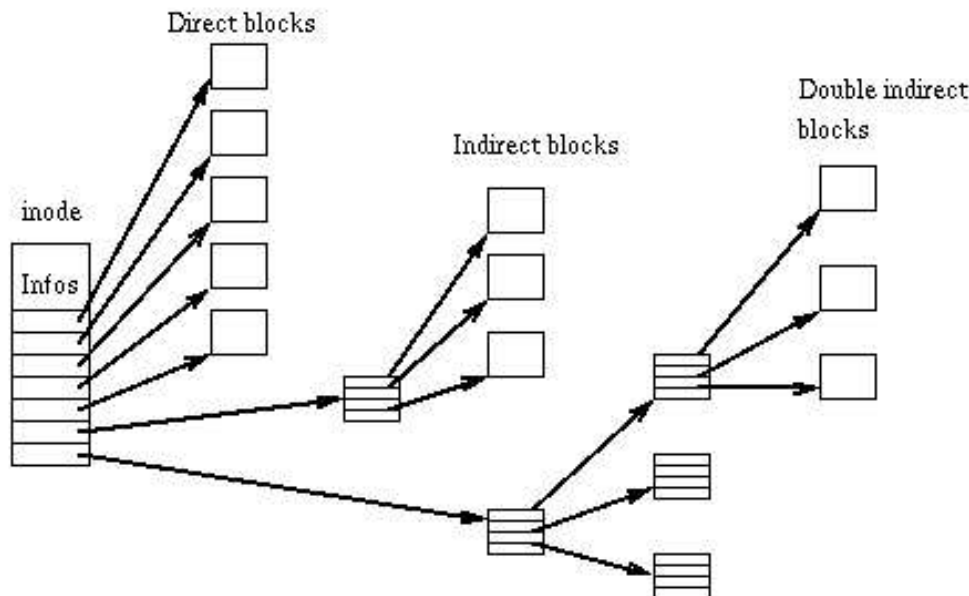
Directories

- Directories are structured in a hierarchical tree

- Each directory can contain files and subdirectories

- Directories are implemented as a special type of files.

- Actually, a directory is a file containing a list of entries.

- Each entry contains an i-node number and a file name.

- When a process uses a pathname, the kernel code searches in the directories to find the corresponding i-node number.

- After the name has been converted to an i-node number, the i-node is loaded into memory and is used by subsequent requests.

Inode table

Directory

i1	name1
i2	name2
i1	name3
i4	name4
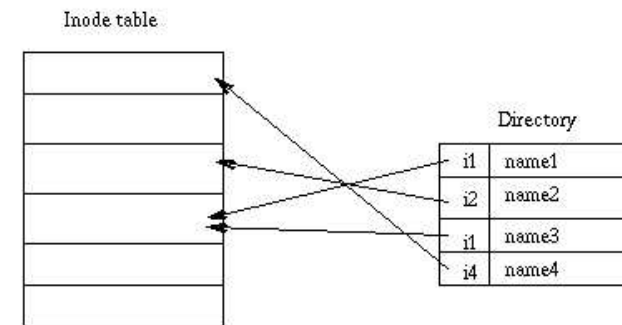
Readdir

- When performing an "ls", the readdir call is called repeatedly

- Every call returns a couple of entries. Entries are not sorted alphabetically.

- This means that "ls" first reads all entries from the directory, sorts them, and then displays them. Does not work very well for large directories.

Hard links

- Unix files-systems implement the concept of a hard-link

- Several names can be associated with an i-node

- This means that several directory names can associate a name to i-node 17

- Requires a link-count field in an i-node

- When the count reaches zero, the i-node and data can be deleted. Note: if the file is still open, it will be erased only when the last user closes it.

Limitations and problems

- A hard-link can only be used within a single file-system.

- Hard links can only point to files. This prevents cycles in the directory tree.

- Hard links complicated locking.

- Hard links are not supported in all file-systems.

Symbolic links

- A symbolic-link is simply a file that contains a filename

- When the kernel encounters a symbolic link during a pathname to i-node conversion, it replaces the name of the link by its contents, i.e. the name of the target file, and restarts the pathname interpretation.

Symbolic links cont.

- Symbolic links are useful because
    - They can point to files on another file-system
    - They can point to any type of file
    - Note: a symbolic link can point to a non-existent file
- However:
    - They use disk space an i-nodes
    - They add overhead to pathname to i-node conversion

Device special files

- Unix employs an "everything is a file philosophy"

- In Unix-like operating systems devices can be accessed via special files

- A device special file does not use any space on the file-system

- It is only an access point to the device driver

- Device special files will not be handled in this course

The virtual file-system

- The Unix kernel contains a Virtual File System layer which is used during system calls acting on files

- The VFS is an indirection layer which handles the file oriented system calls and calls the necessary functions in the physical file-system code to do the I/O

- This indirection mechanism is frequently used in Unix-like operating systems to ease the integration and the use of several file-system types

The virtual file-system

- Common functionality is located in the VFS

- For example:
  - File-name and i-node caches
  - Page cache

- Each file-system only implements its own data-structures and IO path
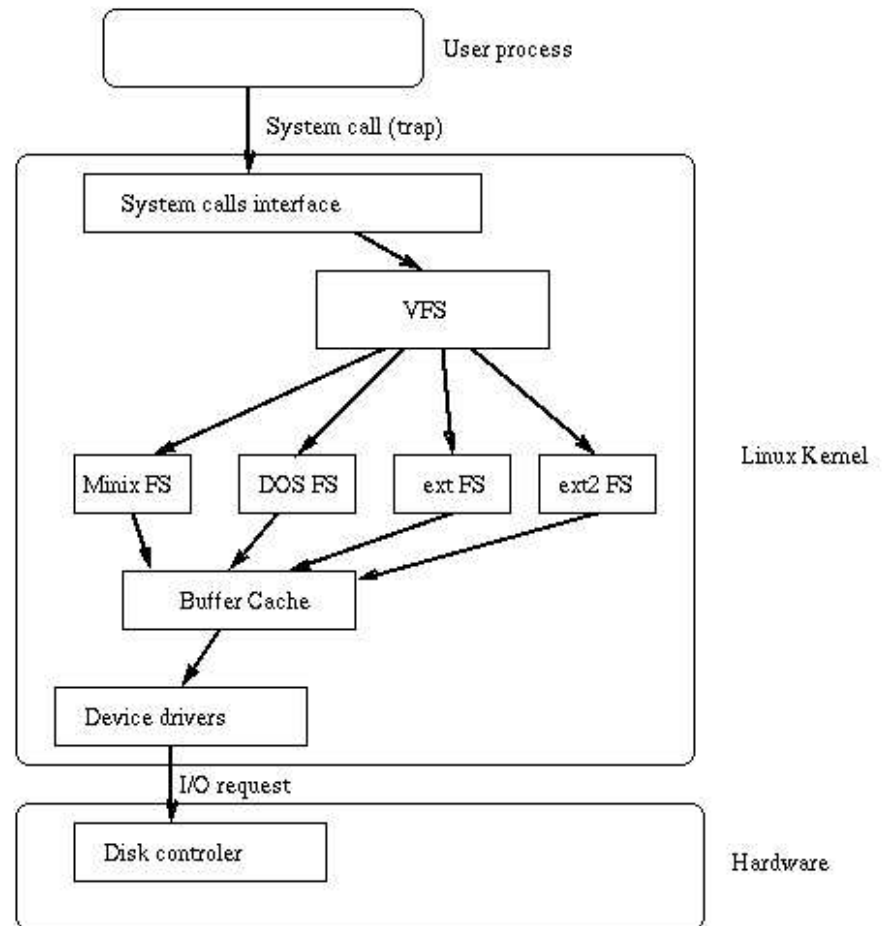
On-disk structure

■ An FFS file-system is made up of block-groups

block group 1	. . .	block group N

■ Each block-group contains

■ A redundant copy of crucial file-system control information (super-block and the file-system descriptors)

■ Part of the file-system: i-nodes, data

Super block	FS descriptors	block bitmap
i-node bitmap	i-node table	data blocks

Block groups

- Using block groups is a big win in terms of reliability
  1. Control structures are replicated
  2. Easy to recover the super-block
- Helps to get good performance
  1. Reducing the distance between the i-node table and the data blocks
  2. It is possible to reduce the disk head seeks during I/O on files

Block groups II

- The amount of i-nodes is preconfigured and cannot be changed

- Block size is pre-configured

  1. Common size: 4KB

  2. Problem: small files waste a lot of space

Directories

Directories are managed as linked lists of variable length entries

i-node number	Entry length	Name length	File-name
i1	7	3	"foo"
i2	18	14	"E. Britannica"
i3	8	3	"bar1"

Read-ahead

- Takes advantage of the page-cache

- When a page is read, the kernel code requests the I/O on several contiguous blocks Hopefully, the next block read by the application will already be loaded into the page cache

- Performed during:
  - Sequential reads on files
  - Directory reads

Allocation optimizations

- Block groups are used to cluster together related i-nodes and data

- Try to allocate data blocks for a file in the same group as its i-node

- Intended to reduce disk-seeks when reading an i-node and its data blocks

When writing data to a file

- Pre-allocate up to 8 adjacent blocks when allocating a new block

- Preallocation hit rates are around 75% even on very full file-systems

- Achieves good write performances under heavy load

- Allows contiguous blocks to be allocated to files

- Speeds up future sequential reads

File-system check (FSCK)

- FSCK can be invoked if
  - The file-system is not shutdown cleanly
  - Every X reboots

- The FSCK process in Ext2 has 5 phases

Pass 1

- Iterates over all i-nodes in the file-system

- Checks each i-node as an unconnected object

- These checks do not require any cross-checks to other file-system objects.

- For example:
  - Make sure the file mode is legal
  - Check that all the blocks in the i-node are valid block numbers

- Bitmaps indicating which blocks and i-nodes are in use are compiled

Pass 1 cont.

- Pass 1 takes the longest time to execute

- All of the i-nodes have to be read into memory and checked

- To reduce the I/O time, critical file-system information is cached in memory

- For example, the location on disk of all of the directory blocks on the file-system is cached

- Obviates the need to re-read the directory i-node structures during pass 2 to obtain this information

Pass 2

- Pass 2 checks directories as unconnected objects

- Directory entries do not span disk blocks

- Each directory block can be checked individually without reference to other directory blocks

- Allows e2fsck to sort all of the directory blocks by block number

  - Check directory blocks in ascending order

  - Decreases disk seek time

Pass 2 (II)

- Directory blocks are checked to make sure that
    - Directory entries are valid
    - Contain references to in-use i-node numbers (as determined by pass 1)

- For each directory check the '.' and '..' entries
    - Make sure they exist
    - Verify that the i-node number for the '.' entry matches the current directory
    - Note: the '..' entry is not checked until pass 3

Pass 2 (III)

- Cache information concerning the parent directory in which each directory is linked
  - If a directory is referenced by more than one directory, the second reference of the directory is treated as an illegal hard link, it is removed

Pass 2 (IV)

- Note:
  - At the end of pass 2, nearly all of the disk I/O for e2fsck is complete
  - Information required by passes 3, 4 and 5 are cached in memory
  - The remaining passes are largely CPU bound
  - They take less than 5-10% of the total running time

Pass 3

- In pass 3, the directory connectivity is checked

- Trace the path of each directory back to the root
  - Use information cached in pass 2.
  - The '..' entry is validated.
  - Any directory that can not be traced back to the root is linked to the /lost+found directory

Pass 4

- In pass 4 the reference-counts for all i-nodes is checked

- Iterate over all the i-nodes

  - Compare the link counts (which were cached in pass 1) against internal counters computed during passes 2 and 3

Pass 5

- In pass 5, e2fsck checks the validity of the file-system summary information

- It compares the block and i-node bitmaps which were constructed during the previous passes against the actual bitmaps on the file-system, and corrects the on-disk copies if necessary.

Journaling

- Journaling is the practice of writing a log entry prior to modifying meta-data

- Used in Ext3

- If journaling was used, FSCK would be needed only in cases of disk-errors

Transactions and consistency

- Many file-system operations requiring modifying several locations on-disk atomically.

- For example, performing a write-append to a file:

  1. Allocation: modify free-space bitmap

  2. Modify file-attributes: modification time, file-length, file-size

  3. Modify indirect blocks

- Ext2 uses fsck

- Newer file systems (ext3) use transactions or equivalent techniques

Performance

- Machine (very old one!)
    - i486DX2 processor
    - 16 MB
    - two 420 MB IDE disks
- We want to see what the performance of Ext2

Bonnie benchmark

- Tests I/O speed on a big file, File size = 60 MB
  - Write data to the file with character based I/O
  - Rewrite the entire file
  - Write data using block based I/O
  - Read the file using character I/O and block I/O
  - Seek into the file

Bonnie benchmark results

- Numbers are in KB/sec

	Char Write	Block Write	Rewrite	Char Read	Block Read
Ext2-fs	452	1237	536	397	1033

Analysis of Bonnie results

- Ext2 is very good in block oriented I/O
    - Writes are fast because data is clustered
    - Reads are fast because contiguous blocks have been allocated to the file
    - There is no head seek between two reads
    - The read-ahead optimizations work

- It is not so good at character oriented I/O. A better character IO library is needed.

The Andrew benchmark

- Developed at Carnegie Mellon University

- It is a script that operates on a collection of files constituting an application program.

- The operations are intended to represent typical user actions.

- The input to the benchmark is a source tree of about 70 files.

- The files total about 200KB in size.

The Andrew benchmark (II)

- Five phases

1. MakeDir - construct a target subtree that is identical to the source subtree.

2. Copy - copy every file from the source subtree to the target subtree.

3. Stat - traverse the target subtree and examine the status of every file in it.

4. Grep - scan every byte of every file in the target subtree, search for a particular pattern.

5. Compile - compile and link all files in the target subtree.

Andrew benchmark results

- Numbers are in milliseconds

	P1 MakeDir	P2 Copy	P3 Stat	P4 Grep	P5 Compile
Ext2-fs	790	4791	7235	11685	63210

Summary

- Ext2 is a fairly simple file system

- Uses various optimizations to improve performace

- Uses FSCK to recover from crashes