COMP124

Computer Systems

16 | Linux File Systems | EXT4 | I/O Streams

Dr Stuart Thomason

Files Everywhere

- The fundamental concept in Unix (and hence Linux) is that everything is a file

- Even things that we wouldn't normally expect to be a file
    - Pipes
    - Sockets
    - Processes
    - USB ports
    - Printers
    - Hard drives
    - Kernel data structures

- Allows us to use the same system calls (and commands) to access everything
- Presents a clean system call interface to applications and developers

Linux File Systems

- Users (and programs) see a unified view of the file system as a tree-like structure of nested directories and files

- This is an abstraction (logical representation) because in reality…
  - Some parts of the tree might be stored on separate disks
  - Some might be stored on a network (remotely)
  - Some might be stored in memory (ie. virtual file system)
  - Each can be formatted with a different physical disk layout

- We use the phrase file system in two slightly different ways
  - The overall tree-like view of the entire logical structure
  - The physical format of each disk (how its blocks are arranged and used)

Top Level Directories

- Every Linux file system has certain top level directories

Directory	Content
/bin	Executable files that users can run (such as shell commands)
/boot	Files needed while the system is booting (such as kernel image)
/dev	Virtual files mapped to physical devices
/etc	Settings and configuration files for system software
/home	Home directories for every user on the system
/lib	Shared library files (similar to DLL files in Windows)
/proc	Virtual files mapped to running processes (plus housekeeping)
/root	Home directory of the root user (super user)
/sbin	Executable files that only the super user can run
/usr	System software (with symbolic links from /bin and elsewhere)
/var	Log files, lock files, cache data (ie. things that change a lot)

Logical System Tree

- The Linux file system tree is a logical structure that could be provided by multiple physical disks, networks, or other devices
  - /dev
    - Virtual file system (stored in memory)
    - Each subdirectory maps onto a physical device
  - /home
    - Often located on network drives (eg. in our Linux farm)
    - Not available until the system has fully booted
  - /proc
    - Virtual file system
    - Stores dynamic data about current processes

- Each part of the logical file system could use a different physical disk format

Root and Boot

- There are two directories that need to be available while the system is booting and the kernel is forking its initial processes
  - /root
    - Home directory of the super user
    - Cannot be stored with other home directories (might not be mounted yet)
  - /boot
    - Stores kernel image and other required files
    - Bootloader knows where it is and how to read its files

- Other parts of the logical file system are mounted in the correct place by the kernel after it has finished loading
- Use the df command to see mount points and physical disks

The EXT4 File System

- One of the most common Linux file systems is EXT4
  - Multiple decades of development (EXT2 dates back to 1993)
  - Proven stability, reliability and performance
  - Requires minimal maintenance
  - Default file system on many Linux installations
- Can support very large disk and file sizes
  - Disks up to 1 exabyte (1 billion GB)
  - Individual files up to 16 terabytes (16,000 GB)
  - Standard disk block size is 4 KB (but can go up to 64 KB)
- Provides several features that improve performance, stability and security

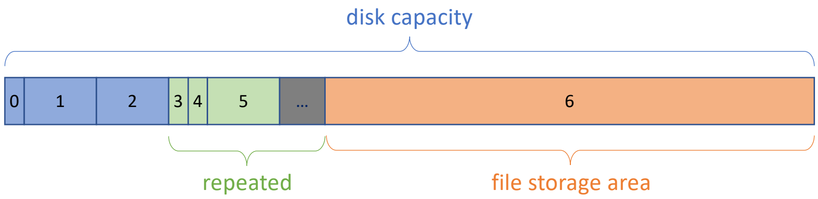
- These slides present a simplified version of how EXT4 works

EXT4 Block Groups

- EXT4 uses the concept of block groups
  - Free block bitmap (free list) is stored in just one block
  - Free inode bitmap is stored in another single block
  - 4 KB block size implies each bitmap can hold 32,768 records
  - Enough blocks to store 128MB of actual data

- A disk (with gigabytes of space) will be split into many of these small block groups
  - Each group will be 128MB in size
  - Each will need its own free block and free inode lists
  - The first portion of the disk will be used to store all this metadata
  - The inodes themselves will also be stored alongside the free lists
  - Rest of the disk is used for actual storage of data

EXT4 Physical Disk Format



Key	Name	Size	Stores
0	Super block	1	EXT4 settings and flags (housekeeping info)
1	Group descriptors	M	Block locations of 3/4/5 for each block group
2	Reserved space	M	Free space to allow more groups to be added
3	Data bitmap	1	Free list for blocks in a single group
4	Inode bitmap	1	Free inode list for inodes in a single group
5	Inode table	M	Table of inode data for a single group
...	(3/4/5 repeated)	M	Bitmaps and tables for every block group
6	Data blocks	M	Physical data for files

1 = Uses exactly one block M = Spans multiple blocks

Inode Data Structure

- Every file (and directory) has an inode associated with it
  - Inodes are stored in the inodes table
  - Which inodes are free/used is stored in the inodes bitmap
- Each file inode stores (among much else)…
  - Size (of actual file on disk)
  - Owner, group, and octal permissions value
  - Type of inode (normal file, directory, symbolic link, etc.)
  - Timestamps (created, modified, accessed)
  - Start and end block numbers of actual data on disk
    (if file is fragmented, will be a list of start/end fragments)

Directory Data Structure

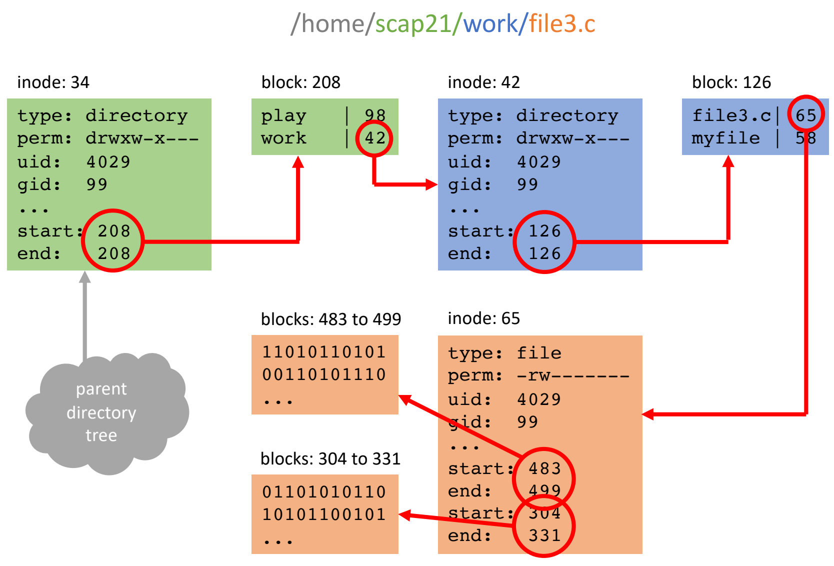
- A file inode does not store the name of the file

- Content of each directory is stored as a normal file in a block somewhere on disk
  - Contains the filename of each file in the directory
  - And its corresponding inode number

- Permissions and other metadata about each directory are stored in the inode that points to that directory data block
- Since directories can be nested, some of the inodes listed in the directory block could point to other directory blocks (and so on)

- Kernel has to follow a trail of inode numbers and block indexes before it can finally read the actual data blocks for a file

File Data Diagram



Symbolic Links (Hard and Soft)

- We can give a file a second filename with the ln command (ln file1 file2)
  - Files can be in different directories (but not different physical partitions)
  - Adds file2's name to the relevant directory data file
  - Along with file1's inode reference
  - Known as a hard link
  - A single inode is referenced from multiple directory entries

- Or we can create a soft link to the file (ln -s file1 file2)
  - Creates a new file called file2 and stores file1's name in it
  - Adds file2 and its new inode reference to the relevant directory data file
  - Soft links can refer to files across different physical disks (hard links cannot do this)
  - Adds a layer of indirection
  - Each directory entry references a different inode

EXT4 Optimisation Features

- Journaling
  - Part of the disk is used to store a journal of changes made to inodes and data blocks
  - Changes are applied to the disk after they are recorded in the journal
  - Writing data is slightly slower (needs to write everything twice)
  - Fast recovery from a system crash (only check and apply journal entries)

- Scattering
  - New files are spread evenly across the disk
  - Large gaps between each file
  - Allows room to grow contiguously
  - Reduces chance of files becoming fragmented

EXT4 Optimisation Features

- Delayed allocation
  - Creates virtual blocks in memory while file is written
  - Commits to physical blocks when the final size is known
  - Allows system to find contiguous free space on disk
  - But could lose data if memory is not written to disk
  - Contentious and can be turned off for specific disks

- Persistent pre-allocation
  - A system call (fallocate) can be used to reserve space for a file
  - All necessary blocks are allocated and filled with zeroes before real data arrives
  - Can be used to guarantee a contiguous set of blocks for the file
  - Improves read/write speed for media streaming and database applications

Standard File Streams

- Every Linux system has three standard streams
  - stdin – Standard input
  - stdout – Standard output
  - stderr – Standard error (where error messages are sent)

- For processes spawned by shell, these are linked to keyboard and terminal of the user
  - Soft links created at /dev/stdin, /dev/stdout, /dev/stderr
  - Point to entries in /proc/self (soft link to current process)

- Child processes inherit the same streams as their parent

- Can be treated as files, so programs can read from standard input and write to standard output via normal file-handling system calls

Standard Streams in Program Code

- Programs can use system calls to access the standard streams
- They have different names depending on the language

Language	Standard In	Standard Out
C / C++	stdin	stdout
C#	System.Console.In	System.Console.Out
Java	System.in	System.out
Python	sys.stdin	sys.stdout

- Under Linux, since everything is treated as a file, we can 'open' virtual files associated with processes and devices
- Reading from a network socket behaves in the same way as reading from a normal file, with the same system call interface

File Descriptors

- Every open file has a file descriptor (a positive integer)
  - Used by system calls instead of file names
  - File manager maps file descriptors onto physical files
  - Recorded in the PCB of processes using the file

- Standard streams always have the same descriptors (stdin is 0, stdout is 1, stderr is 2)
  - Remember the Linux file system stores process details in a virtual directory
  - The output stream of the current process is a soft link to its file descriptor

```
$ echo "Hello" > /proc/self/fd/1
Hello
```

- The C printf and scanf subroutines are wrappers around system calls that use file descriptors 1 and 0 internally