



Architect Korner

File Management

Software architects have a lot to do with reading and writing to databases, files, devices and memory locations. Sending data in and out of files efficiently and synchronously is very crucial in this field. Files are important not only from an operating system perspective, but also for application programming. This article introduces you to how architects use files to build their applications, middleware and entire software products on the Linux platform.

The word 'file' originates from the Latin *filum*, which means a thread to hold loose papers, especially arranged for reference. But in computing science, the word 'file' is used as a collection of data stored with one name, which is the file name. Linux treats everything as a file. As an example, data, devices, inter-process communication mechanisms, and links are all dealt with as

files. There are different types of files—namely, regular or ordinary files, directory files, device files, link files, FIFO and socket files. Device files are of two types: character and block device files. There are two types of link files: soft and hard [1] link files. In Linux, these files are arranged in an inverted tree data structure, for easy maintenance.

How are these files stored on a storage medium like a hard disk, USB storage or

CD-ROM? Linux uses a file system to manage file storage. It supports various kinds of file systems [2], which are depicted in Figure 1. Fundamentally, a file system is used to organise, manipulate and retrieve stored files from a storage medium. A comprehensive list of file systems, with a brief description of each, can be found at [3] and a comparison of different file systems with distinct features is available at [4].

The Linux file system

Linux used to use the Third Extended file system (ext3) as the default file system, but recent kernel versions (2.6.28 onwards) use the Fourth Extended file system (ext4), which has many more desirable features compared to ext3. The complete features of the ext4 file system are available at [5]. The major differences between ext3 and ext4 are given in Table 1.

File System	Addressing	Maximum file system size	Maximum file size
Ext3	32-bit	16 Terabyte (TB)	2 Terabyte (TB)
Ext4	48-bit	1 Exabyte (EB)	16 Terabyte (TB)

Table 1: Comparison between ext3 and ext4 file systems

Executing `fdisk /dev/<device file name>` will list the specified storage device's partitions, as shown in Figure 2. Type `L` to obtain a list of known partition types, as shown in Figure 3.

Normally, we use the `mount` command to mount a file system, and the `umount` command to unmount the mounted file system. However, we can configure the necessary file systems to be automatically mounted during system start-up, in the `/etc/fstab` file, and thus it is available for use as soon as you log in to the system.

In our day-to-day life, we use different kinds of devices, which in turn use different kinds of file systems. But Linux uses a set of commands to access a file from any device or file system. How is this possible? Linux accesses files through the Virtual File System (VFS), which helps a user to access any device with the same set of commands. VFS completely hides the complexity of different file systems from the user. Figure 4 illustrates how a user accesses a file through VFS. Please refer to Bovet and Cesati [6], to understand the internals of the Linux file system.

The `extundelete` utility [7] can recover deleted files on both ext3 and ext4 partitions. It uses the stored partition's journal information to do this.

Linux uses the `resize2fs` command-line utility to resize (shrink or expand) existing ext2/ext3/ext4 file systems. If you need a graphical utility, then `gparted` [8] can be used.

How is a file accessed by a process? Let us explore this.



Figure 1: Linux-supported file systems

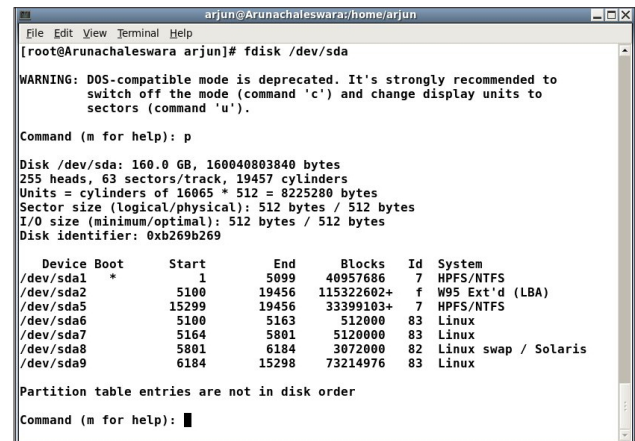


Figure 2: Screenshot of fdisk /dev/sda

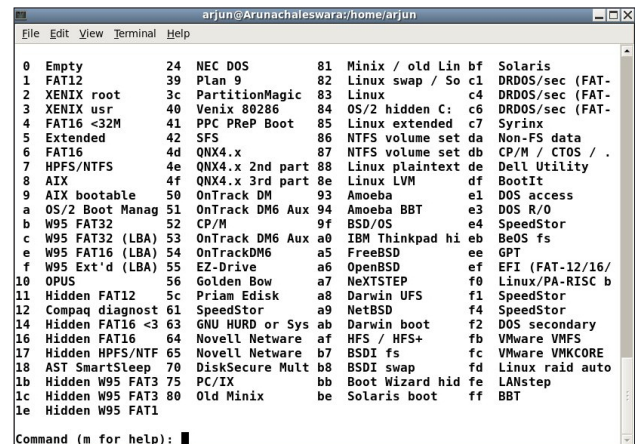


Figure 3: Screenshot—list of partition types

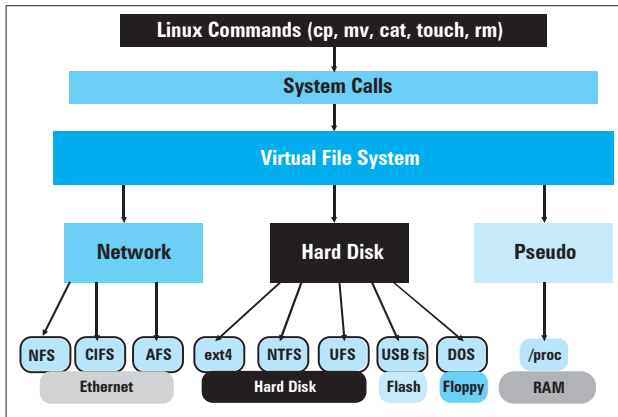


Figure 4: File system interface

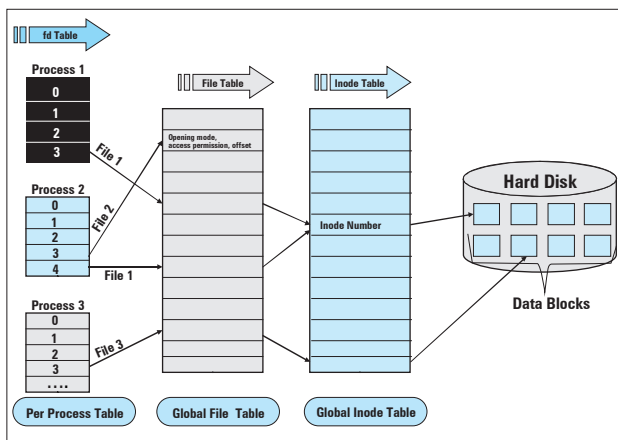


Figure 5: Interaction between processes and VFS objects (normal files)

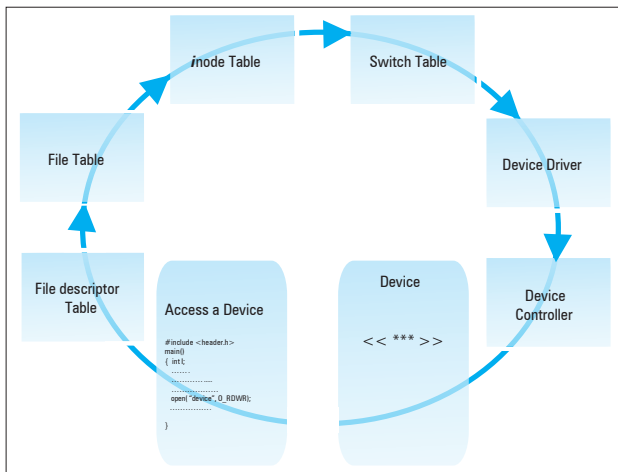


Figure 6: A user process accesses a device-special file

Accessing normal/ordinary files

Linux uses *system calls* to access a file. Let's begin with the *open* system call, which is used to open a file, and which returns a file descriptor for access to the file. This file descriptor is an integer, and its value normally starts at 3, since file descriptors 0, 1 and 2 are assigned for the

process' standard input (*stdin*), standard output (*stdout*) and standard error (*stderr*) respectively. Each process has a file descriptor table, in which file descriptors are linked to the file table, and to the inode table, subsequently. The inode table stores a unique inode number for each file—so the total number of inodes in the table is equivalent to the total number of files in the file system. Both the file table and inode table are global tables. The given inode number is routed to a data block on the hard disk/storage device, where the file is stored. An overview of the file-opening operation is shown in Figure 5.

Accessing device-special files

An ordinary file is stored in a hard disk data block, so we can access it easily. But what happens with a device-special file? Since the device file is an entity used to access a device, if you store/write any data to such a file, it will be sent to the device, and not actually stored in a file. Device-special files are in the */dev* directory. Each device file is associated with two numbers -- its major and minor numbers. The major number is used to identify the device type, and the minor number is used to identify the instance of that device type. In this case, the inode table points to a switch table.

There are two switch tables in Linux—character and block device switch tables. Using the major number, the switch table routes to a corresponding device driver, and the device driver accesses the device. The complete flow of a user process accessing a device is shown in Figure 6.

Linux system calls and strace

Linux has approximately 338 system calls. The system call table is stored in *unistd.h* (*/usr/src/linux-2.6.34/arch/x86/include/asm/unistd_32.h*), and each system call is identified by a unique system-call number. The file- and file-system-related system calls are given in Table 2, with their call numbers and a brief description of each call.

File system related system calls are available in Table 2 carried in the LFY CD bundled with this issue.

Let's say you want to trace a system call. You want to find out which system calls an executable program uses, how many times each system call is invoked, and how much time it takes for each invocation. We can get all this information from the *strace* command-line utility. For example, assume I want to create a named pipe, i.e., a FIFO file, to communicate between two processes. To do this, I'd run the *mkfifo* command. We can execute *mkfifo* via *strace* to obtain system call coverage information, as shown in Figures 7 and 8.

Figure 8 shows the execution of *strace -c -F mkfifo*, and lists the information in a different format. Using this, we can see which system calls were used by *mkfifo*, how many times they were used, and how much time each call took. This reveals that *mkfifo* calls *mknod* to create a file; indeed, *mkfifo* is just a wrapper around the *mknod* system call. Thus,

calling *mknod* instead of *mkfifo* will reduce the execution time required to create the FIFO file.

The *proc* file system

The *proc* file system is one of the important virtual file systems in Linux. It helps to maintain information about currently running processes, system limitations, and system information (like CPU, RAM, IRQ, etc). The output of the *mount* command lists this line for *proc*:

```
proc on /proc type proc (rw)
```

Here, the first *proc* is a pseudo-device created in kernel memory. It is mounted at the */proc* mount-point, with the *proc* file-system type. The size of */proc* is always zero, because the data accessible via */proc* is generated on the fly. For example, if you execute a program (which becomes a running process), then a directory is created in */proc* with the process identification number (PID). The kernel maintains process-related information like its state, PID, PPID, GID, fd table, and other parameters, in the process ID directory, as long as the process is executing in the system. Once the process has ended, the directory is removed. More complete information about */proc* was published in an earlier issue of LFY [9].

Developers can also create their own entries in */proc* to store necessary information about their application programs, by interfacing the *proc* file system with a kernel module. An excellent resource for more information on this is reference [10].

Synchronisation of a shared file

Linux is a multi-processing operating system, and if more than one process wants to access the same file, or the same record in a file, it leads to a race condition. For example, if the first process opens the file, the kernel sets the file pointer to the file. Then, the second process opens the same file. The first pointer is updated with the second pointer, so whatever you modify with the first process will not be reflected in the file; only modifications by the second process will be reflected.

This undesirable situation, caused by the race condition, can be avoided if we synchronise the file with the help of a suitable synchronisation mechanism. Linux supports all kinds of synchronisation mechanisms, at user level as well as kernel level. At the user level, we have semaphores to synchronise a critical section, but to synchronise a file, it may not be a good choice. Instead, file locking is more suitable. In multi-threaded programming, *pthread_mutex* is used to synchronise a thread.

In this section, we discuss file-locking mechanisms: we have two locking mechanisms, which can be implemented by *flock* and *fcntl*. The *fcntl* type of locking is more robust, flexible and fine-grained. Figures 9 through 12 explain the different types of locking architectures. You can get information about the Linux kernel locking mechanisms,

```

arjun@Arunachaleswara:/home/arjun
File Edit View Terminal Help
execve("/usr/bin/mkfifo", ["mkfifo", "myfifo"], [{"/* 50 vars */}] = 0
brk(0) = 0x8fd4000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7887000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
.....
mknod("myfifo", S_IFIFO|0666) = 0
close(1) = 0
close(2) = 0
exit_group(0) = 7

```

Figure 7: Running strace mkfifo

```

arjun@Arunachaleswara:/home/arjun
File Edit View Terminal Help
[root@Arunachaleswara arjun]# strace -c -F mkfifo myfifo
% time    seconds  usecs/call   calls   errors syscall
-----
94.87    0.000462      462         1         0 mknod
5.13     0.000025       2         11         0 mmap2
0.00     0.000000       0          3         0 read
0.00     0.000000       0          5         0 open
0.00     0.000000       0          7         0 close
0.00     0.000000       0          1         0 execve
0.00     0.000000       0          1         1 access
0.00     0.000000       0          3         0 brk
0.00     0.000000       0          1         0 munmap
0.00     0.000000       0          4         0 mprotect
0.00     0.000000       0          5         0 fstat64
0.00     0.000000       0          1         0 set_thread_area
0.00     0.000000       0          1         0 statfs64
-----
100.00    0.000487                44         1 total
[root@Arunachaleswara arjun]#

```

Figure 8: Running strace -c -F mkfifo

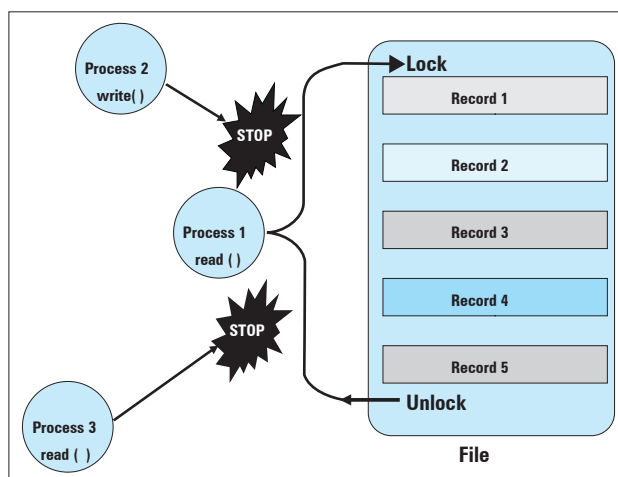


Figure 9: Non-shared lock—Sys V semaphore

and semaphore usage in programming at the user level, from references [10-12].

File-locking implementations: *fcntl*

The *flock* structure is used to declare and initialise a file lock. The structure contains the following members:

```

struct flock {
short l_type ;
short l_whence ;
off_t l_start ;
off_t l_len ;
pid_t l_pid ;
};

```

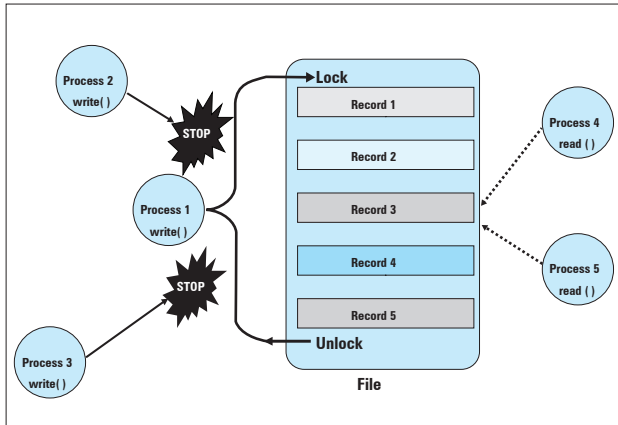


Figure 10: Non-shared lock—mandatory file locking

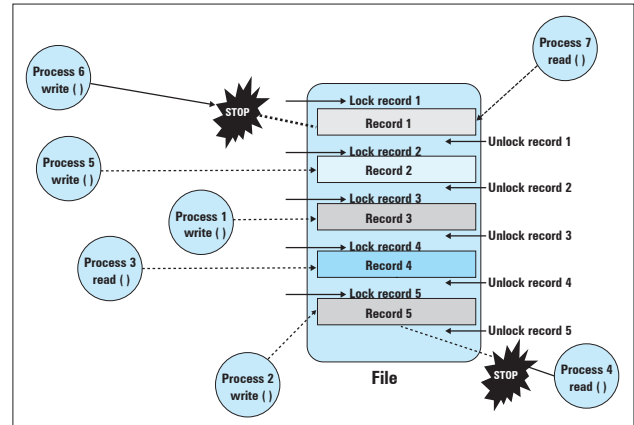


Figure 12: Shared lock—advisory or record locking

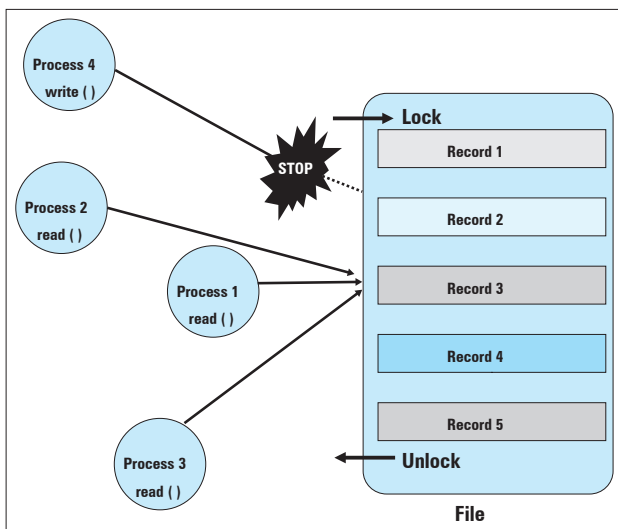


Figure 11: Shared lock—mandatory file locking

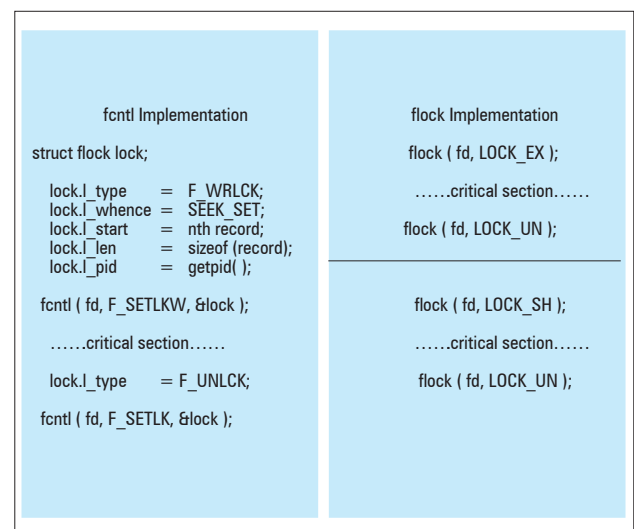


Figure 13: Sketch of file-locking implementation

Let's look at the members of the struct:

- *l_type* specifies the lock type, and is one of the following: read lock (*F_RDLCK*), write lock (*F_WRLCK*) or unlock a locked file (*F_UNLCK*). A read lock will allow many readers to access the file at the same time, but no writer is allowed. A write lock will allow only one writer at a time.
- 1. *l_whence* is used to set the file pointer relative to the current cursor position. If, for example, you want to go to the starting location of a file, then pass *SEEK_SET*, or if the file is already open and you subsequently want to set the lock from the current cursor position, then pass *SEEK_CUR*. If you want to go to the end of the file, then pass *SEEK_END*.
- 2. *l_start* member of the structure is used to tell the starting point of the file, which could be from the 10th byte.
- 3. *l_len* is used to specify till what point you want to lock the file. It could be till the 100th byte.
- *l_pid* is the process ID of the process that holds the lock.

If the process that is holding the lock exits abnormally, then the kernel will release the lock, and the next deserving process can acquire the lock.

If we want to lock the entire file, we set the *l_start* and *l_len* fields to 0. To improve locking granularity, however, each process can lock a specific record, or range of bytes in a file, by setting appropriate values in the *l_start* and *l_len* fields. Figure 13 explains how to approach this design of locking a specific record or byte, by the suitable setting of the *l_start* and *l_len* fields.

The locking and unlocking is done by calling the *fcntl* system call. Its syntax is:

```
int fcntl (int fd, int cmd, struct flock *lock);
```

We have three commands: *F_SETLKW*, *F_SETLK* and *F_GETLK* to acquire, release or check the availability of the lock. The first two commands are used to acquire or release the lock, but the difference between them is that

SETLKW will wait till the lock is released, if it's held by another process, but *SETLK* will return with an error if the lock is not available. *F_GETLK* will just check if the lock is available.

File-locking implementations: *flock*

The usage of *flock* is simpler than *fcntl*. On the flip side, *flock* does not have many features that a developer needs. Use *flock* just to lock or unlock, by passing a suitable operation flag. The *flock* command manages locks within shell scripts, or at the command line. The *flock* system call is used to apply or remove an advisory lock on an open file. Information on the *flock* system call is provided below.

Syntax:

```
int flock (int fd, int operation);
```

Available operations:


- *LOCK_SH* (shared lock, more than one process can hold)
- *LOCK_NB* (don't block when locking)
- *LOCK_EX* (exclusive lock; only one process at a time)

- *LOCK_UN* (remove an existing lock)

Examples of file-locking implementations of both *fcntl* and *flock* types are shown in Figure 13.

To sum up, let's look at the procedure for lock usage:

- a. Declare a lock
- b. Initialise the lock
- c. Lock the critical section
- d. Unlock the critical section

In this article, we have looked at how a file is accessed on the Linux platform, and its complete sequence. We showed different file-related system calls, and discussed how they work. We explained different file-synchronisation mechanisms available on Linux, and the procedures to access different kinds of files. This information is valuable to software architects who create entire software products on the Linux platform. **END** 

Acknowledgement

The authors would like to acknowledge Krishna Sudhakaran for his valuable assistance in the preparation of the view graphs.

References

1. Kernel Corner: *Starting with Linux Device Drivers*, by Dr B. Thangaraju, *LINUX For You*, May 2003, p 83
2. Linux supported file systems: <http://www.linux-tutorial.info/modules.php?name=MContent&pageid=243>
3. List of file systems: http://en.wikipedia.org/wiki/List_of_file_systems
4. A comparison of file systems: http://en.wikipedia.org/wiki/Comparison_of_file_systems
5. Ext4 fs features: <http://kernelnewbies.org/Ext4>
6. Understanding the Linux Kernel, (3rd Edition) by Daniel P. Bovet, Marco Cesati, O'Reilly Publications, 2005
7. extundelete details: <http://extundelete.sourceforge.net/>
8. gparted documentation: <http://gparted.sourceforge.net/documentation.php>
9. Kernel Corner: *Examining Process Information*, by Dr B. Thangaraju, *LINUX For You*, January 2004, pgs 84-87
10. Kernel Corner: *Interfacing the proc file system with a kernel module*, by Dr B. Thangaraju, *LINUX For You*, February 2004, pgs 90-92
11. Kernel Corner: *Linux Kernel Locking Mechanisms for Kernel Programming*, by Dr B. Thangaraju, *LINUX For You*, September 2003, pgs 81-83;
12. Basics of System V Semaphore, by Dr B. Thangaraju, *LINUX For You*, July 2006, pgs 86-89;
13. The Intricacies of System V Semaphore, by V. Shobana and Dr B. Thangaraju, *LINUX For You*, April 2009, pgs 40-43.

By: Gururajan Narasimhan Erode and Dr B. Thangaraju

Gururajan Narasimhan Erode currently works with Wipro Technologies as a lead consultant in the Wireless and Embedded Domain. He has 16 years of experience after a masters degree in engineering, and has worked with various multinational companies in the USA, Germany and India. Gururajan's specialisation is in wireless communication, algorithms and data structures, and Linux systems programming. He can be reached at gururajan.erode@wipro.com.

Dr B. Thangaraju received his Ph.D. in Physics and worked as a research associate in the Indian Institute of Science from 1996 to 2001. From 2001, he has been working at Wipro Technologies as a senior consultant, and his core expertise is in the Linux kernel and embedded and real-time Linux. He has published more than 35 papers on Linux in renowned international and national journals, and has presented more than 20 technical papers at national and international conferences. He can be reached at balat.raju@wipro.com.