Linux File System Study Guide

1. Describe the role of a file manager.

A file manager has four basic roles:

i. Translate file names (user-defined data units or UDDUs) into volume relative addresses (disk blocks).
ii. Provide protection in a multi-user environment.
iii. Move data between user space buffers and logical OS blocks fragmenting and re-assembling as required.
iv. Traverse a mapping structure in which the file manager allocates and de-allocates logical operating system (OS) blocks to individual UDDUs.

The first two roles are public (part of the OS's user interface) while the last two are internal to the kernel. Historically, the public view of the Unix file manager has changed little, but the mapping structure changes with each implementation of Unix as well as among versions of the same Unix implementation.

2. Describe Linux's file manager.

Other Unix implementations have just one file manager, but Linux combines one file manager front-end (file name translation and protection) with many back-ends (mapping structure and user buffer management). The Linux "native" root file system is usually "ext2" format but may be a MINIX file format. The more common mounted volumes are proc, NFS, msdos, Novell, OS/s, Windows NT, CD-ROM, or Windows file sharing. There are also file manager back-ends for other versions of Unix such as Xenix, System V, and Coherent.

The ability to have multiple file manager back-ends in Linux is a fundamental design innovation not seen in any other common operating system. Multiple back-ends allow Linux to be configured and adapt to many unique situations that other operating systems are unable to handle.

3. Describe a typical file manager mapping structure that allocates and de-allocates logical operating system (OS) blocks.

The hard disk (partition) is divided into four areas: an administrative description of the hard disk (super block), one or more areas of index lists (inodes), allocated file data pointed to by the index lists (directories and regular files), and lists of yet-to-be-used blocks (the free list).

Before a file system can be used (mounted), it must be pre-formatted (mkfs) for the file manager. Mounted file systems present a "point-of-no-return" problem for the file manager. Usually the file manager just checks the super block to see if the volume has the correct format and, if so, assumes the other structures (like the inode list) are valid. Depending on the type of file manager and where the

volume is mounted, a corrupted partition will crash the operating system since the file manager must rely upon the file structure for access to system programs.

4. **Explain the role of Linux's "special" files and describe five examples.**

   Like Unix, Linux uses the file manager to create "system abstractions" or special purpose files. These constructs are not files at all, but kernel device drivers and other "logical" drivers that perform unique functions when accessed as a file.

   a. /dev/null - The "rat hole" dumps input into the "bit bucket" when written and returns end-of-file when read.
   b. /dev/zeros - When read, it returns a buffer full of zeros.
   c. /dev/tty - When accessed, it always points to the program's specific controlling /dev/ttyx.
   d. /dev/hda1 - Provides direct access to the hard disk without file manager interpretation.
   e. /dev/ram - Makes high memory look like a hard disk (for rescue).

5. **Explain how the "/proc" abstraction has evolved in Linux compared to other Unix implementations.**

   Historically, Unix allowed user access to kernel variables and the currently running user process through /dev/kmem and /dev/umem abstractions. Assuming the kernel or user object modules were not stripped of their symbol tables, a debugger could be used to access to the kernel's or users variables through /dev/kmem or /dev/umem. In more recent versions of Unix, the /proc directory was created and combined with a logical driver. This design was better than before, but it still required "ioctl" service calls to access kernel variables.

   Linux has extended this concept by creating a separate file system manager back-end called "/proc" which is mounted like other file systems. The /proc manager separates parts of the kernel into directories. Accessing a given /proc file yields the variable name and value. The /proc abstraction also permits access to user process as well as many device drivers.

6. **Describe how the file system structure and the mount command work to create a transparent file system.**

   . The first block of the file system (super block) contains a volume relative index to one or more "inode" sections.
   i. Inodes, in turn, index to directory files and regular files.
   ii. First inode always "points" to root directory file.
   iii. The booted root file system is "hard wired" into the kernel image (see rdev).
   iv. A mounted directory is the root directory of the mounted file system.

   To access a file, the file manager must first decode a series of directories or "path components." As the inode number for each component is discovered in a directory, the mount table is consulted to see if that inode number had been previously used to mount another volume (file system). If the inode number was in the mount table, the contents of the inode are ignored and, instead, inode number one in the new file system is accessed.

7. **Describe the arguments to the mount command and explain the role of the /etc/fstab file. Also, explain why some mount commands without arguments seem to work.**

   The Linux command mount -t fstype device mount-directory takes at least three arguments. -t fstype refers to one of the 12 possible file managers. device refers to the device driver that can access specific storage

devices such as /dev/fd0, and mount-directory refers to an existing directory in the file hierarchy. Note that any files contained in the directory will be hidden by the mount operation.

The /etc/fstab configuration file contains a list of file systems and arguments. The mount, umount, and fsck commands read this file to discover details about the available file systems.

```
file system  mount point  type    options                 dump  pass
-----------  -----------  -------  ------------------------  ----  ----
/dev/sda1    /          ext2    defaults,errors=remount-ro  0     1
/dev/sda2    none         swap    sw                      0     0
proc         /proc        proc    defaults                0     0
/dev/sda3    /usr         ext2    defaults                0     2
/dev/fd0     /floppy      msdos   noauto,conv=auto,user       0     0
/dev/cdrom   /cdrom       iso9660 noauto,ro,user              0     0
```

If the mount command is given with just one argument and it matches one of the rows in fstab, the other fields in the row are taken as arguments for the command. Given the above fstab file and the command mount /cdrom, the resulting command would be mount -t cdrom -o ro /dev/cdrom /cdrom. The "noauto" keeps mount from automatically mounting the floppy or CD at boot time while the "user" option allows a non-superuser account to mount the floppy or CD.

8. Explain where the file manager back-ends are located and how they are activated.

File manager back-ends are either compiled directly into the kernel image or dynamically loaded as "device driver" modules. To see which back-ends a given kernel contains, examine the /proc/filesystems file. Its contents might look like this:

```
        ext2
        minix
        msdos
nodev   proc
        iso9660
        umsdos
nodev   nfs
```

indicating that, at the moment, this kernel supports seven file manager back-ends.

Access to additional file manager back-ends are provided through the load module library and are found in the /lib/modules/2.x.x/fs directory. These file manager back-ends are dynamically loaded into the kernel when the volume is mounted. Example loadable back-ends are:

```
autofs.o  hpfs.o    minix.o   nfs.o     sysv.o    umsdos.o  xiafs.o
ext.o     isofs.o   ncpfs.o   smbfs.o   ufs.o     vfat.o
```

A file manager back-end is activated when its file system is entered as part of a path traversal to reach a desired file. Linux usually begins with its native ext2 file manager and switches as it enters the mounted volume.

mounted volume.

9. **Explain how the MS-DOS file manager back-end maps incompatibilities between MS-DOS and Linux files.**

   Linux (Unix) ASCII files use one character (the LF or \n) to indicate end-of-line. MS-DOS employs two characters (the CRLF or \r\n pair) to show end-of-line. Linux (Unix) employs multi-group permissions and other file attributes while MS-DOS has just a few attributes (hidden, archive, read only).

   The MS-DOS file manager back-end, therefore, must map a Linux (Unix) \n to \r\n as the file is written onto the MS-DOS volume. Coming from a MS-DOS volume, the file manager back-end must give additional permissions (rwxrwxr-x root root) and map \r\n to \n. File manager back-end mappings are controlled by additional arguments added to the mount command. For example, `mount -t msdos -o conv=auto /dev/fd0 /floppy` tells the file manager back-end **not** to do the above mapping if the MS-DOS file has a common binary extension such as "exe."

10. **Explain the conflicts that removable media introduce into a multi-user, data cached operating system and possible solutions.**

    Use of the mount and umount commands indicate that the file manager maintains "state" information about a volume. Users allocate entries in kernel data structures when they "open" a given file. If a mounted CD-ROM or floppy is removed, then the file manager is unable to complete its accounting information for users accessing the volume. Furthermore, files written to the volume may still be in the data cache and not yet written to the floppy.

    The solution is to prevent a user from removing the media. The eject button on the CD-ROM drive, for example, is disabled while the volume is mounted. In the case of the floppy disk, there is no solution other than to restrict floppy disk access to the console terminal. Linux allows users access to any mounted volume.

11. **When using the umount command, explain what the error message "/dev/xxx busy" really means.**

    The "state" information described above includes a "count" of the number of processes accessing a given inode. If one attempts to unmount an inode with a non-zero reference count, then the inode is "busy" since the other processes wish to read/write to or through the mounted inode.

    The "busy" count could also refer to the person issuing the umount command if their current working directory is on or below the mounted directory.

12. **Explain why one would receive the error message "mount: /dev/cdrom is not a valid block device."**

    The mount utility attempts minimal error checking before it really mounts the new volume. Once the first block is read in, its contents are compared to a key constant (magic number) to double check that it is a valid file system for that type of file manager back-end.

13. Explain how the fdformat and mkfs commands differ.

14. Describe the role of fsck and describe the three basic ways a block could be missing or duplicated in a fully-indexed file system.

0. Free list block numbers, file block numbers, and inode block numbers may be double referenced in one list, referenced in more that one list, or have no reference.
1. An allocated inode may not have a directory entry (orphaned file) or a directory entry may reference a free inode.
2. Size of a file may differ from its block count.

15. Describe the different types of pages and how they relate to swapping.

Linux divides up memory into process, shared process, shared library, and data cache areas. After process and library memory has been allocated, the data cache will grow to fill most of memory. As new process are activated, memory space will be reclaimed from the data cache. After a point, the data cache will no longer give up memory and idle pages are moved to the swapping disk to make room for new processes. On the Intel architecture, a "page" is 4096 bytes. The read only shared pages have high priority and tend to stay in memory, but when their space is needed, they are overwritten to improve swapping speed.

16. Describe the results of the free command.

```
17.             total       used       free     shared    buffers     cached
18.     Mem:      14092      12288       1804      17456       1100       5588
19.     Swap:     40156         24      40132
```

Physical memory is 16 MB and about 2 MB is used by the kernel. Thus the "total" remaining is 14 MB, and about 12 MB has been "used" or allocated to user programs and 2 MB are "free" or held in reserve for new processes.

Of the "used" 12 MB, 1 MB has been allocated to "buffers" or permanent data cache while another 5.5 MB of memory has been "cached" or taken over by the data cache mechanism. The remaining 3.5 MB (used - free + buffers + cached) holds about 17 MB of "shared" code. Thus, processes share 3.5 MB of physical memory while executing as if they had 17 MB of memory.

Even though there is 40 MB of swap space, only 24 KB of it have been used.

20. Explain the following setup commands:

```
21.     # dd if=/dev/zero of=/swapfile bs=1024 count=8192
22.     # mkswap swapfile 8192
23.     # sync
24.     # swapon swapfile
```

/dev/zero is a system abstraction that returns a continuous stream of null bytes. The device dump (dd) program reads 8KB of zeros and writes a contiguous (and hopefully sequential) file. mkswap initializes a free list of swap blocks within the file. sync moves file blocks left in the data cache out to the hard disk, and swapon directs the system swapper to be used in the file.

25. Contrast "special," "block special," and "character special" devices.