Linux Basics 2: Filesystem Contd.

To work effectively on a Linux system, it is necessary to understand a number of the basic aspects of Linux:

- **The Filesystem (directories and files)**
  - the logical filesystem tree (directory structure)
  - pathnames
  - file naming scheme
  - key commands
  - **standard directories**
  - **regular files vs. special files**
  - **physical filesystems**
  - **additional commands**
- Users, Groups, and File Permissions (security mechanisms)
- Processes (running programs and their attributes)

Standard Directories

There has been a major effort in recent years to standardize the layout of directories in the Linux filesystem, so most distributions are very similar.

This project is called the Filesystem Hierarchy Standard:
www.pathname.com/fhs

The project specifies a set of directories that must exist, and what their purpose is, so things like system libraries are always in standard locations.

Standard Directories (contd.)

The standard "top-level" directories are:

/bin – (essential) binary/executable files

/boot – bootable kernel files

/dev – device files

/etc – configuration files

/home – user home directories (as /home/username)

/lib – (essential) libraries and kernel modules

/media – mount points for removable media (newer)

/mnt – mount points for removable/temporary media

/opt – optional software

Standard Directories (contd.)

The standard "top-level" directories (contd.):

/proc – pseudo files that display/change OS params

/root – home for the root user

/sbin – system binaries and executables (usually only in root's search path)

/tmp – temporary files (world writable)

/usr – non-host specific, read-only information, secondary hierarchy: e.g., `/usr/bin`, `/usr/lib`, etc.

contains `/usr/local` for purely local/non-system files (cannot be overwritten by standard system software), and yet another hierarchy: e.g., `/usr/local/bin`, etc.

/var – variable files like log files

Regular vs. Special Files

In Linux, many objects that can be opened and read from or written to are treated *largely as if they are disk files*, including:

- **regular files** (normal disk files)
- **directories**
- **symlinks**
- **FIFOs** and **sockets** (IPC mechanisms)
- **devices** (hardware):

All non-regular files are called **special files**.

Regular vs. Special Files (contd.)

Directories are considered *special files* because though they are disk files, they have special format and special API calls:

- they contain a list of *records*
- each record contains an **inode number** and filename
- the first two directory records are always:
  . (dot) a link to the directory itself
  .. (double dot) a link to the parent directory

Physical Filesystems

The term "**filesystem**" can also be used to refer to the *format used to store files on a storage device*, i.e., a **physical filesystem**.

As noted above, the single logical filesystem tree is composed from multiple partitions and/or storage devices, each mounted at some point in the tree.

All of these partitions/storage devices make use of a physical filesystem to organize the data they contain.

There are a wide variety of different physical filesystems in common use, including: FAT, NTFS, ext2/ext3/ext4, JFS, HFS+, ZFS, ISO 9660, UDF, exFAT, JFFS2, LTFS, etc.

Physical Filesystems

A filesystem is designed for a particular type of physical storage device.

The largest set of filesystems are for *"disk systems,"* which allow relatively rapid *random access*.

Disk filesystems include: FAT, NTFS, ext2/ext3/ext4, JFS, HFS+, ZFS, etc.

ISO 9660 and UDF are designed for *optical discs* (CD-ROM, DVD-ROM, BR).

exFAT, JFFS2, etc. are for *flash memory* storage
(note that most consumer devices present a "disk interface,"
so are used with disk filesystems).

Physical Filesystems (contd.)

Linux has the ability to work with many disk filesystems.

There are several “native” Linux disk filesystems:

- ext2, ext3, ext4, JFS, XFS, ReiserFS, Btrfs
- all of these except ext2 are **journaled**

Linux can also mount Windows filesystems:

- there is full read/write support for FAT32 and NTFS
- this means you can access your Windows files from Linux on a dual-boot machine or from a USB storage device

Linux currently has only read support for journaled HFS+ filesystems used on Macs.

Physical Filesystems (contd.)

Linux physical filesystems store a file's *contents* separate from the file's **metadata** (information about the file).

The metadata is stored in an **inode structure**, which includes the following information:

- inode number
- file owner (a user) and file group
- mode (permissions plus type)
- size in bytes
- last access time (**atime**)
- last modification time (**mtime**)
- last metadata change time (**ctime**)
- link count (explained below)

Physical Filesystems (contd.)

The **link count** attribute of an inode structure warrants further explanation.

We typically think of files as being "contained in" a directory.

What is actually stored inside a Linux directory, however, is a *file record*.

Each file record contains two pieces of information:

- a *name* for the file
- a *link* to the file itself (i.e., to the file's *inode*)

Physical Filesystems (contd.)

Each directory entry (record) is termed a **hard link** to the file.

A file (i.e., the contents) can be *linked multiple times*.

This means that a file can have *multiple names* in the filesystem, in different directories or even in the same directory.

Most regular files will have a link count of one.

Directories have a link count of *at least two*, because each directory contains two special records:

- . (dot) – a link to the directory itself
- .. (double dot) – a link to the parent directory

Physical Filesystems (contd.)

A second type of "file link" is the **symbolic link/soft link/symlink**.

A symlink is a type of *special file* that contains the pathname of another file (as text).

Many commands/functions automatically *"follow symlinks,"* which means they operate on the target file in the symlink rather than on the symlink file itself.

This means that symlinks often function much like hard links in terms of accessing the target file contents.

Symlinks make it easy to see that a file "points to" another file.

Symlinks are much used to provide multiple names for system libraries and the like (e.g., libx.so and libx.so.3 are symlinks to the true library file libx.so.3.2).

Additional Filesystem Commands

ln – create hard/soft links

- `ln existing-name second-name`
- `ln -s existing-name symlink-name`