The UNIX File System

- Everything in the UNIX system is a file.
- When the first version of the system was being designed, before it even had a name, the discussion focused on the structure of a file system that would be clean and easy to use.
- The file system is central to the success and convenience of UNIX. It is a good example of the "keep it simple" philosophy, and the power achieved by careful implementation of a few well chosen ideas.
- Details of using the file system:
  –what files are
  –how they are represented
  –directories and the file system hierarchy
  –permissions
  –inodes (system's internal record of files)
  –device files

- A file is a sequence of bytes. No structure is imposed on the file by the system, and no meaning is attached to its contents—the meaning of the bytes depends solely on the program that interprets the file.

- This is true not just of disc files, but of peripheral devices as well. Magnetic tapes, mail messages, characters typed on the keyboard, line printer output, data flowing in pipes—each of these files is just a sequence of bytes as far as the system and the programs in it are concerned.

Create a file:

```
~$ ed
a
now is the time
for all good people
.
w junk
36
q
~$ ls -l junk
-rw-rw-r--    1 you   you    36 Mar  9 07:56 junk
~$
```

junk is a file with 36 bytes—the 36 characters typed while appending. To see the file:

```
~$ cat junk
now is the time
for all good people
~$
```

The command od (octal dump) prints a visible representation of all the bytes of a file:

```
~$ od -c junk
0000000   n   o   w       i   s       t   h   e
0000020   f   o   r       a   l   l       g   o
0000040   p   l   e  \n
0000044
~$
```

- The 7-digit numbers down the left side are positions in the file, that is, the ordinal numbers of the next character shown, in octal. (The emphasis on octal numbers is a holdover from the PDP-11, for which octal was a preferred notation.)

- Each line ends with a character with octal value 012. This is the ASCII newline character; it is what the system places in the input when you press the RETURN key. By a convention borrowed from C, the character representation of a newline is `\n`. But this is just a convention to make it easy to read—the value stored in the file is the single byte 012.

- Other special characters include backspace (octal value 010, printed as `\b`), tab (011, `\t`), etc.

- It is important to distinguish between how a character is stored in a file and how it is interpreted in various situations. For example, when you type a backspace on your keyboard, the kernel intrprets it to mean that you want to discard whatever character you typed previously. Both that character and the backspace disappear, but the backspace is echoed to your terminal, where it makes the cursor move one position backward. (You can insert a backspace character in a file by typing \← .)

- The UNIX system is unusual in its approach to representing control information, particularly its use of newlines to terminate lines.

- Many systems provide instead “records,” one per line, each of which contains not only your data but also a count of the number of characters in the line (and no newline).

- Other systems terminate each line with a carriage return and a newline, because that sequence is necessary for output on most terminals. (The word “linefeed” is a synonym for newline, so this sequence is often called “CRLF.”)

- The UNIX system does neither—there are no records, no record counts, and no bytes in any file that you or your programs did not put there. A newline is expanded into a carriage return and a newline when sent to a terminal, but programs need only deal witht he single newline character, because that is all they see.

- The kernel keeps track of file lengths, so there is no need to add a special end-of-file character at the end; a program encounters end-of-file when it has processed all the bytes in the file.

- Programs retrieve the data in a file by a system call (a subroutine in the kernel) called

  `read`

  Each time `read` is called, it returns the next part of a file. It also says how many bytes of the file were returned, so end of file is assumed when 0 bytes are returned.

- This scheme makes a lot of sense: all files must end. Since all files must be accessed through `read`, returning zero is an interpretation independent way to represent the end of file without introducing a new special character.

- **What's in a file?** The format of a file is determined by the program that uses it; it is not determined by the file system. Thus, the kernel cannot tell you the type of a file, because it does not know it.

- The command `file` makes an educated guess:

```
~$ file /bin /bin/ed junk
/bin:    directory
/bin/ed: ELF 32-bit LSB executable, Intel 80386, version 1,
dynamically linked (uses shared libs), stripped
junk:    ASCII English text
~$
```

- To determine the file types, `file` didn't pay any attention to the file names (although it could have), because naming conventions are just conventions.

- Instead, `file` reads the first few hundred bytes of a file and looks for clues to the file type. Sometimes the clues are obvious: a runnable program is marked by a binary "magic number" at its beginning. `od` with no options dumps the file in 16-bit (2-byte) words and makes the magic number visible:

  ```
  ~$ od /bin/ed
  0000000 042577 043114 000401 000001 000000 000000 000000 000000
  0000020 000002 000003 000001 000000 110600 004004 000064 000000
  0000040 014464 000001 000000 000000 000064 000040 000006 000050
  ...
  ```

  The octal value 42577 marks an executable. Specific magic numbers are system dependent.

- For text files, the clues may be deeper in the file, so `file` looks for words line `#include` to identify C source, or line beginning with a period to identify `nroff` or `troff` input.

Directories and Filenames

- All files have unambigous names. If you have a file `/home/you/junk` and you type `ls junk`, then you obtain: `junk`, not `/home/you/junk`, why?

- Because each process has a **current directory**, and all file names are implicitly assumed to start with the name of that directory, unless they begin directly with a slash. Your login shell and `ls`, therefore have a curent directory. The command `pwd` (print working directory) identifies the current directory:

  ```
  $ pwd
  /home/you
  $
  ```

- The current directory is an attribute of a process, not a person or a program—people have login directories, processes have current directories. If a process creates a child process, the child inherits the current directory of its parent. If the child changes to a new directory, the parent is unaffected.

- The notion of current directory is a notational convenience, but its real purpose is organizational.

Permissions

- Every file has a set of **permissions** associated with it, which determine who can do what with the file. However, there is a special user, the **super-user**, who can read and modify any file in the system, (login name `root`.)

- You can, however, encrypt files in your directory (using PGP, for example). It is believed that in real life, most security breaches are due to passwords that are given away or easily guessed.

- When you log in, you type a name and a password, The system actually recognizes you by your user-id, **uid**. Besides a uid you are assigned a **group-id**, which places you in a class of users. The file system, and the UNIX system in general, determines what you can do by the permissions granted to your uid and group-id.

- The file `/etc/passwd` is the password file; it contains all the login information about each user. You can discover your uid and group-id, as does the system, by looking up your name in `/etc/passwd`:

  ```
  $ grep you /etc/passwd
  you:gkmbCTrJ04COM:500:500:You:/home/you:/bin/bash
  $
  ```

  The fields in the password file are separated by colons and are laid out like this:

  login-id:encrypted-password:uid:group-id:miscellany:login-directory:shell

- The file is ordinary text, but the field definitions and separator are a convention agreed upon by the programs that use the information in this file.

- Note that your password appears here in an encrypted form, since anyone can read this file. When you give your password at login, it is encrypted and compared with the results in `/etc/passwd`.

- ```
  ~$ ls -l /etc/passwd
  -rw-r--r-- 1 root root 1271 Feb 12 12:03 /etc/passwd
  ~$
  ```
  ```

- Anybody can open `/etc/passwd` and make changes, but only root can write the changes back. The program to change passwords is called `passwd` and:

```
~$ ls -l /usr/bin/passwd
-r-s--x--x 1 root root 13476 Aug 7 2001 /usr/bin/passwd
~$
```

- The `s` instead of `x` in the execute field for the file owner states that, when the command is run, it is to be given the permissions corresponding to the file owner, in this case root. Because `passwd` is "set-uid" to root, any user can run the `passwd` command to edit the password file.

- The set-uid bit is a simple and elegant idea (patented by Dennis Ritchie) that solves a number of security problems. For example, the author of a game program can make the program set-uid to the owner, so that it can update a score file that is otherwise protected form other users' access.

- But the set-uid concept is potentially dangerous. `passwd` has to be correct; if it were not, it could destroy system information under root's auspices. If, for example, it had the permissions:

  ```
  -rwsrwxrwx
  ```

  it could be overwritten by any user, who could therefore replace the file with a program that does anything.

- The set-uid bit is powerful, but used primarily for a few system programs such as `passwd`. A more ordinary file:

  ```
  ~$ ls -l /usr/bin/who
  -rwxr-xr-x 1 root root 14364 Jul 23 2001 /usr/bin/who
  ~$
  ```

  is executable by everybody, and writable by root and the group root. When you type `who` to the shell, the shell "finds" the file `who`, and checks if it has execute permission. If it does, the shell calls the kernel to run it.

- Note that a program is just a file with execute permission.

- Directory permissions operate a little differently, but the basic idea is the same:

```
~/somedir$ ls -ld .
drwxrwx--x 46 you you 4096 Jan 31 11:48 .
~/somedir$
```

  - The leading `d` signifies that `.` is indeed a directory.
  - An `r` means that you can read the directory, with the `ls` command, for example.
  - A `w` means that you can create and delete files in this directory, because that requires modifying and therefore writing the directory file. (There are system calls that create and remove files, and only through them is it possible to change the contents of a directory—the `w` tells you who can use these system calls.)
  - Permission to remove a file is independent of the file itself. If you have write permission in a directory, you may remove files there, even files that are protected against writing.
  - The `x` field in the permissions on a directory does not mean execution; it means "search." The mode `--x` for other users means that users may access any file that they know about in that directory, but may not run `ls` on it or read it (with `od`) to see what files are there.