

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

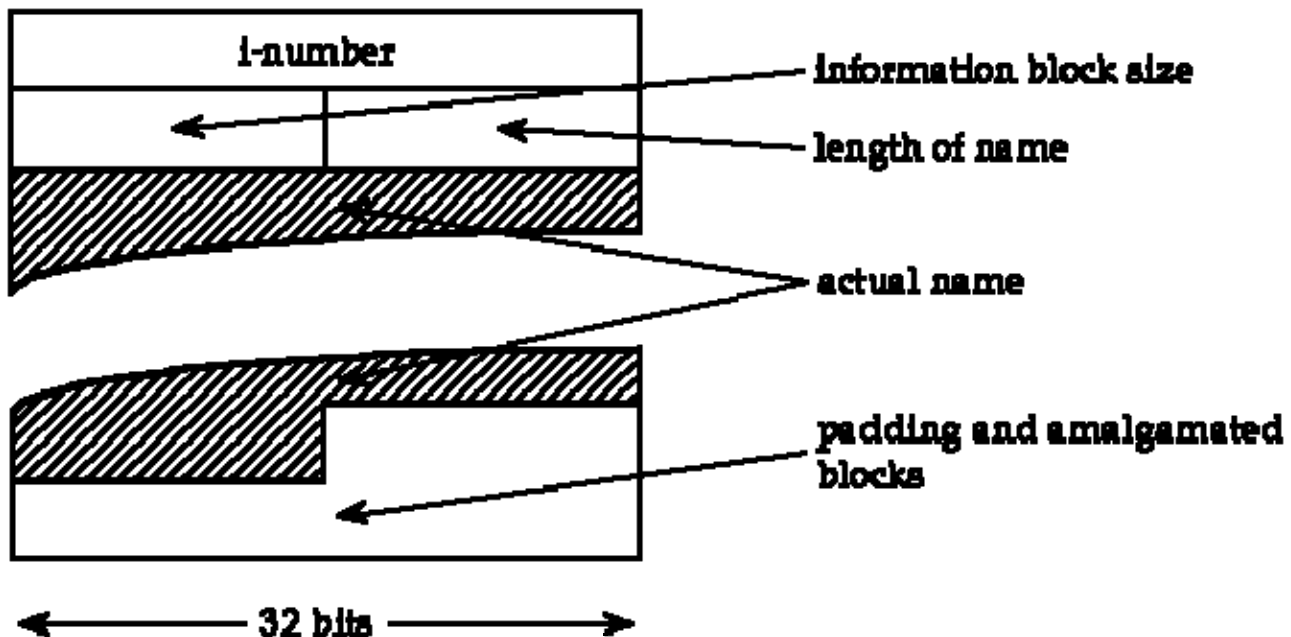
THE UNIX FILE SYSTEM

Under UNIX we can think of the file system as everything being a file. Thus directories are really nothing more than files containing the names of other files and so on. In addition, the file system is used to represent physical devices such as tty lines or even disk and tape units.

DIRECTORY STRUCTURES

All the information about a file, apart from its name, is stored in the inode. The file name and the i-number which identifies the inode is stored in a directory entry. A directory is a regular file, however, in order to avoid file system integrity problems, the kernel will not allow a directory to be opened for writing using normal file system calls.

Older versions of Unix (System 5 and earlier) used a directory format consisting of a sequence of 16 byte records, the first 2 bytes held the i-number and the remaining 14 bytes held the file name. There was, thus, a limit of 14 characters on the maximum size of a file name. When a file was deleted the i-number of the associated directory record was set to zero to mark the record as free, however the name was not set to null. More recent versions of Unix have adopted the more flexible structure for directory entries shown in the diagram.



When a file is deleted the space occupied by the directory entry is amalgamated with the free space at the end of the previous entry and the information block size field is incremented. This technique has the advantage of allowing usefully long file names without requiring large directory records. The disadvantage is that the manipulation of directories is more complex, directories can only be read via a special system call (`getdent()`). Normal manipulation of directories from within programs is handled via library routines.

INODES

Each file on the system has what is called an inode that contains information on the file. To see the fields of the inode look at manual page of the `stat` system call. This shows the following fields:

```
struct stat {
```

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

```
dev_t  st_dev;    /* device inode resides on */
ino_t  st_ino;   /* this inode's number */
u_short st_mode; /* protection */
short  st_nlink; /* number of hard links to the file */
short  st_uid;   /* user-id of owner */
short  st_gid;   /* group-id of owner */
dev_t  st_rdev;  /* the device type, for inode that is device */
off_t  st_size;  /* total size of file */
time_t st_atime; /* file last access time */
int st_spare1;
time_t st_mtime; /* file last modify time */
int st_spare2;
time_t st_ctime; /* file last status change time */
int st_spare3;
long st_blksize; /* optimal blocksize for file system i/o ops */
long st_blocks; /* actual number of blocks allocated */
long st_spare4;
u_long st_gennum; /* file generation number */
};
```

The key fields in the structure are `st_mode` (the permission bits), `st_uid` the UID, `st_gid` the GID, and `st_*time` (assorted time fields).

The `ls -l` command is used to look at all of those fields:

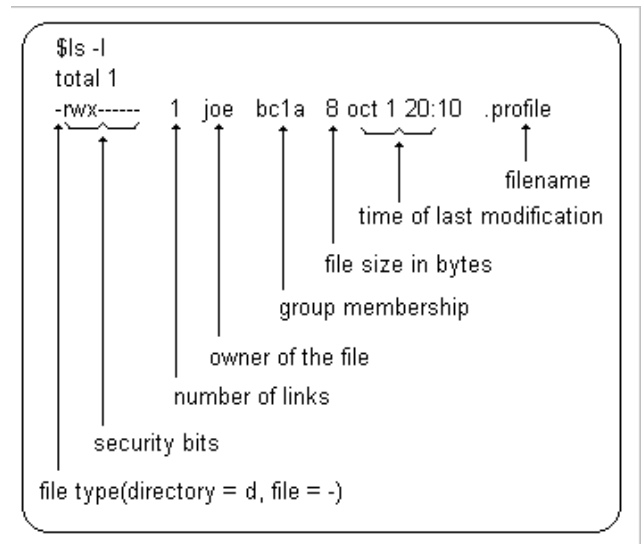
```
umbc4:gopher> ls -l dead.letter
-rw-rw-rw- 1 gopher 307 Sep 14 12:33 dead.letter
```

```
umbc4:gopher> ls -F www
acsinfo/ index.html irc.giflogo.gif umbcfaq/
```

There is an inode on disc for every file on the disc and there is also a copy in kernel memory for every open file. All the information about a file, other than its name, is stored in the inode. This information includes:

- File access and type information, collectively known as the mode.
- File ownership information.
- Time stamps for last modification, last access and last mode modification.
- Link count.
- File size in bytes.
- Addresses of physical blocks.

An inode is a file system data structure that keeps track of a file. Inodes store information about each file not stored within the file itself or within the local directory. Where a file contains only its own content and a directory holds only the files name, the inode contains all the other information describing a file -- disk location; owner; permission matrix; dates and times for the file's creation, last access, and last modification; and the number of links to the file. Basically, a lot of important information.



UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

Upon creation, a file system reserves space for inodes. The default setting creates an inode for every 2K bytes contained in the file system. Hence, a disk that is 512 bytes in size (which I believe it is), would reserve 20% of each file system for inodes.

A file system can run out of space in two ways: It can run out of file space or it can run out of inodes. In a typical file system, file space goes first because most system's average file size is larger than 2K bytes. To construct a file system for large files, you might change the ratio between file space and inode space when you create the file system (e.g., with newfs) to account for this files. Similarly, a file system housing only, or predominantly, small files might allot more space to inodes and less to file contents.

If, for example, you expect the average file size on a new disk to be 1024 bytes, then set up a file system reserving as much space for inodes as for files. Do this by using the following option and argument with newfs:

```
# newfs -i 1024 /dev/rdsk/c0t0d0s6
mkfs -F ufs -o N /dev/rdsk/c0t1d0s6 2052288 72 14 8192 1024 16 6 90 1024 t 0 -1 8 128
/dev/rdsk/c0t1d0s0: 2052288 sectors in 2036 cylinders of 14 tracks, 72 sectors 1002.1MB in 128 cyl
groups (16 c/g, 7.88MB/g, 7552 i/g)
super-block backups (for fsck -F ufs -o b=#) at:
 32, 16240, 32448, 48656, 64864, 81072, 97280, 113488, 129696, 145904,
 162112, 178320, 194528, 210736, 226944, 243152, 258080, 274288, 290496,
306704, 322912, 339120, 355328, 371536, 387744, 403952, 420160, 436368, 452576,
468784, 484992, 501200, 516128, 532336, 548544, 564752, 580960, 597168,
613376, 629584, 645792, 662000, 678208, 694416, 710624, 726832, 743040,
759248, 774176, 790384, 806592, 822800, 839008, 855216, 871424, 887632,
903840, 920048, 936256, 952464, 968672, 984880, 1001088, 1017296, 1032224,
1048432, 1064640, 1080848, 1097056, 1113264, 1129472, 1145680, 1161888,
1178096, 1194304, 1210512, 1226720, 1242928, 1259136, 1275344, 1290272,
1306480, 1322688, 1338896, 1355104, 1371312, 1387520, 1403728, 1419936,
1436144, 1452352, 1468560, 1484768, 1500976, 1517184, 1533392, 1548320,
1564528, 1580736, 1596944, 1613152, 1629360, 1645568, 1661776, 1677984,
1694192, 1710400, 1726608, 1742816, 1759024, 1775232, 1791440, 1806368,
1822576, 1838784, 1854992, 1871200, 1887408, 1903616, 1919824, 1936032,
1952240, 1968448, 1984656, 2000864, 2017072, 2033280, 2049488
```

To examine existing file system, the `df -k` command and the `df -t` command looks at file space first, and then at the number of free inodes. With these commands, you can check your file system's balance. Has a file system that is 80% full used 80% of its inodes or far more/fewer? In file system that never exceed 50% full, it may not matter if space for files and inodes is getting used at an uneven rate. However, if you risk exhausting one or the other, then make corrections.

To modify the rate between files and inodes, back up the file system, recreate it with different parameters, and then reload it from your backups. The default values are probably good enough to give you good disk usage if you have no idea what to expect.

The layout of the components of a System V disc inode is shown in the following diagram. It occupies 16 32-bit words.

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

There are 13 physical block addresses in an inode, each of these addresses is 3 bytes long. The first ten block addresses refer directly to data blocks, the next refers to a first level index block (which holds the addresses of further data blocks), the next refers to a second level index block (which holds the addresses of further index blocks) and the last refers to a third level index block (which holds the addresses of further second level index blocks).

Type/ Mode	Link Count
User Id	Group Id
File Size (in bytes)	
Last Accessed Time	
Last Modified Time	
Created Time	

12 data block addresses

First level index block address
Second level index block address
Third level index block address

Extensions (7 4 byte words)

--

← 32 bits →

All physical addresses associated with a file are implicitly assumed to reside on the same disc, there is no facility whereby a file could span more than one disc. There is no requirement that the physical addresses of a file should be contiguous (i.e. adjacent) and with multiple files being handled on a disc it is unlikely that contiguity would offer any advantages for performance. There is also, more surprisingly, no requirement that all logical blocks should map to physical blocks, it is quite permissible for files to have "holes" and this is quite likely to happen with large sparsely populated direct access files.

Assuming 512 byte blocks and 3 bytes per address which is equivalent to a disc capacity of about 8 GByte. An index block of 512 bytes is capable of holding 170 3 byte addresses. The size of the largest file can be calculated thus:

1. Directly addressed blocks $10 \times 512 \text{ byte} = 5120 \text{ bytes}$
2. Blocks addressed via first level index block $170 \times 512 \text{ byte} = 87040 \text{ bytes}$
3. There will be 170 index blocks addressed via the second level index block. This will address $170 \times 170 \times 512 \text{ bytes} = 14796800 \text{ bytes}$
4. Via the third level index block there will be $170 \times 170 \times 170 \times 512 \text{ bytes}$ of addressable data. This comes to 2515456000 bytes.

The total addressable space comes to 2530344960 bytes (approximately 2.5 Gbytes).

BSD and other more recent versions of Unix use a larger disc inode format that consists of 32 4 byte words. The block addresses now occupy 4 bytes rather than 3 and various other fields are larger as will be seen from the diagram below.

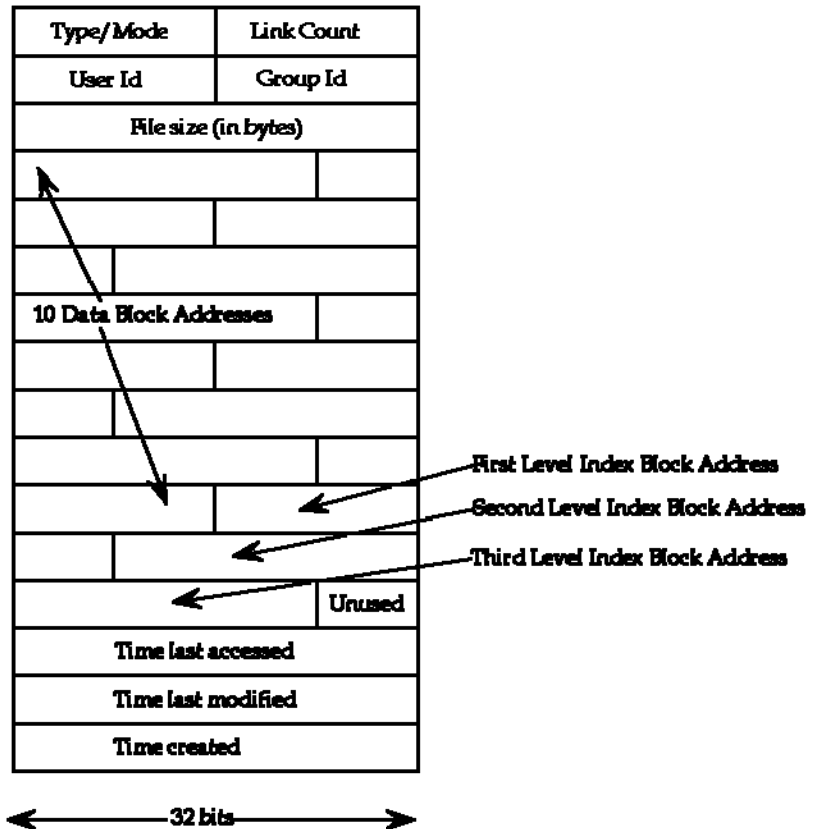
UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

The extensions include space for 32-bit user and group ids and an inode generation number incremented when a free inode is used for a different file. The generation number is used by the network file system for file handle calculation. It should also be noted that the time fields have expanded to 64 bits so that the year 2031 problem (when the Unix standard time format wraps) can be avoided.

Once a file has been opened the in-memory (the phrase "in-core" is traditionally used) version of the inode contains significant extra information. This extra information includes

- In-core inode status indicating whether:
 1. the inode is locked
 2. a process is waiting for the inode to be unlocked
 3. the in-core inode is dirty, i.e. differs from the disc version
 4. file modifications have been made that haven't been written to disc



- The device number of the disc the file belongs to.
- The inode number sometimes known as the i-number. On disc the inodes form an array and the i-number is inferred from the inode's position in this array.
- Pointers to other in-core inodes.
- A reference count indicating the number of instances of the file being active or open.

FILE LIMITS

Although most Unix users may not be aware of it, files have several limitations. These include:

- the number of files that can exist in any particular file system
- the number of files that a single process can have open
- the number of files that can be referenced in the inode table cache (in memory).

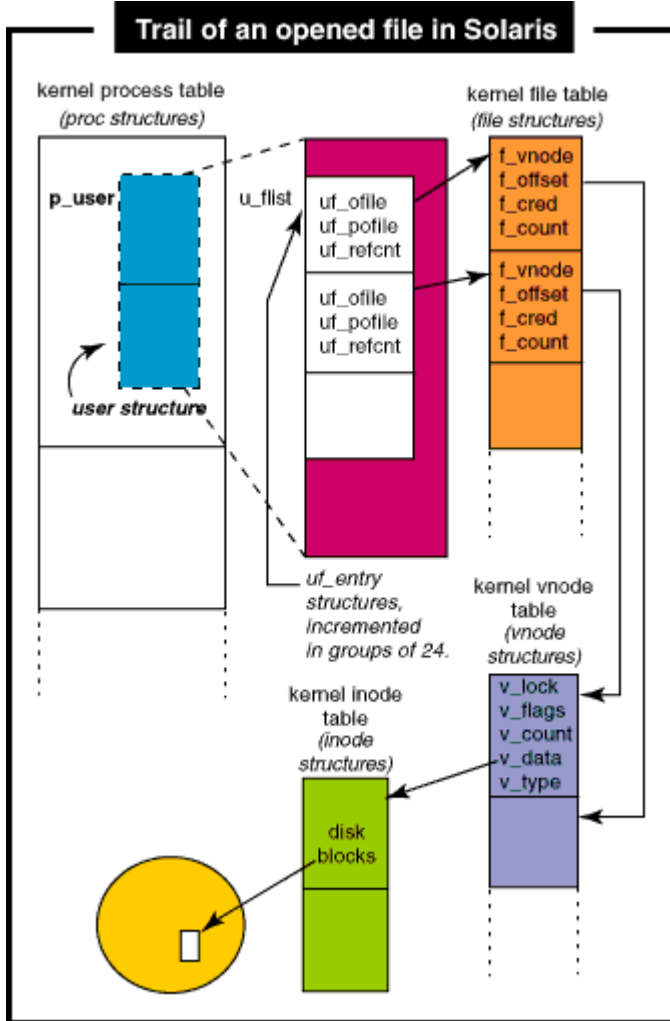
The first limit relates to the number of allocated file information nodes (inodes) when a file system is created. For most file systems, an inode is allocated for every 2K bytes of available space. If a file system runs out of inodes, then no additional files can be added – even if plenty of disk space remains. This rarely happens. In general, more inodes are allocated than ever will be used. In other words, the average file size of most file systems is larger than 2K and you'll run out of disk space

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

before you run out of inodes. The `df -t` command will report on the number of free inodes in each of your file systems.

The number of files that a process can have open depends on limits set on the system -- accessible through the `ulimit` command. If you enter the command `ulimit -a`, then you will see a number of limits that control resources available to your current shell. The "nofiles" (number of files) limit determines the number of files that can be open simultaneously. The Solaris default is 64. This and other limits reported by `ulimit` keep any particular user or process from overwhelming the system.



The `nofiles` parameter affects network connections as well as open files -- sockets use file descriptors just as open files do. To view the file descriptors used by any particular process, grab the process ID from the `ps` output use the `pfiles` command (`/usr/proc/bin/pfiles`). You should see something like this:

```
# /usr/proc/bin/pfiles 254
254: /usr/dt/bin/dtlogin -daemon
Current rlimit: 2014 file descriptors
0: S_IFDIR mode:0755 dev:32,24 ino:2 uid:0
gid:0 size:512
O_RDONLY|O_LARGEFILE
1: S_IFDIR mode:0755 dev:32,24 ino:2 uid:0
gid:0 size:512
O_RDONLY|O_LARGEFILE
2: S_IFREG mode:0644 dev:32,24 ino:143623
uid:0 gid:0 size:41
O_WRONLY|O_APPEND|O_LARGEFILE
3: S_IFCHR mode:0666 dev:32,24 ino:207727
uid:0 gid:3 rdev:13,12
O_RDWR
4: S_IFSOCK mode:0666 dev:174,0 ino:4686
uid:0 gid:0 size:0
O_RDWR|O_NONBLOCK
5: S_IFREG mode:0644 dev:32,24 ino:143624
uid:0 gid:0 size:4
O_WRONLY|O_LARGEFILE
advisory write lock set by process 245
```

```
7: S_IFSOCK mode:0666 dev:174,0 ino:3717 uid:0 gid:0 size:0
O_RDWR
8: S_IFDOOR mode:0444 dev:179,0 ino:65516 uid:0 gid:0 size:0
O_RDONLY|O_LARGEFILE FD_CLOEXEC door to nsdc[171]
```

This listing shows the files open by the `dtlogin` process. Notice how easy it is to decipher the file types in this output. We have:

- S_IFDIR directory files
- S_IFREG regular files
- S_IFCHR character mode device

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

S_IFSOCK sockets
S_IFDOOR a "door" file

Limits on the number of files that a process can open can be changed system-wide in the `/etc/system` file.

If you support a process that opens a lot of sockets, then you can monitor the number of open files and socket connections by using a command such as this:

```
# /usr/proc/bin/pfiles <proclD> | grep mode | wc -l
```

The third limit determines how many file references can be held in memory at any time (in the inode cache). If you're running the `sar` utility, then a `sar -v` command will show you (in one column of its output (`inod-sz`)) the number of references in memory and the maximum possible. On most systems, these two numbers will be oddly stable throughout the day. The system maintains the references even after a process has stopped running -- just in case it might need them again. These references will be dropped and the space reused as needed. The `sar` output might look like this:

```
** 00:00:00 proc-sz ov inod-sz ov file-sz ov  
11:20:00 400/20440 0 41414/46666 0 1400/1400 0 0/0
```

The 4th field reports the number of files currently referenced in the inode cache and the maximum that can be stored.

FILE TIMES

Unix records three file times in the inode, these are referred to as `ctime`, `mtime`, and `atime`. The `ctime` field refers to the time the inode was last changed, `mtime` refers to the last modification time of the file, and `atime` refers to the time the file was last accessed.

The `ctime` file of the inode is updated whenever the file is written too, protections are changed, or the ownership changed. Usually, `ctime` is a better indication of file modification than the `mtime` field. The `mtime` and `atime` fields can easily be changed via a system call in C (or a perl script). The `ctime` field is a little harder to change, although not impossible.

File times are important because they are used in many ways by system administrators. For example, when performing backups, an incremental dump will check the `mtime` of the inode to see if a file has been modified and should be written to tape. Also, system administrators often check the `mtime` of certain key system files when looking for signs of tampering (while sometimes useful, a hacker with sufficient skill will reset the `mtime` back). Finally, when managing disk space, some sites have a policy where files not accessed in a certain time are marked for archival, it is not uncommon to have certain users deliberately set the `atime` or `mtime` to defeat this policy.

FILE PERMISSIONS

File permissions are used to control access to files on the system. Clearly in a multi-user system some method must be devised that allows users to select files for sharing with other users while at the same time selecting other files to keep private. Under Unix, the inode maintains a set of 12 mode bits. Three of the mode bits correspond to special permissions, while the other nine are general user permissions.

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

The nine general file permissions are divided into three groups of three. The three groups correspond to owner, group, and other. Within each group there are three distinct permissions, read, write, and execute. The nine general file permissions are listed via the `ls -l`. The following table summarizes the file permissions:

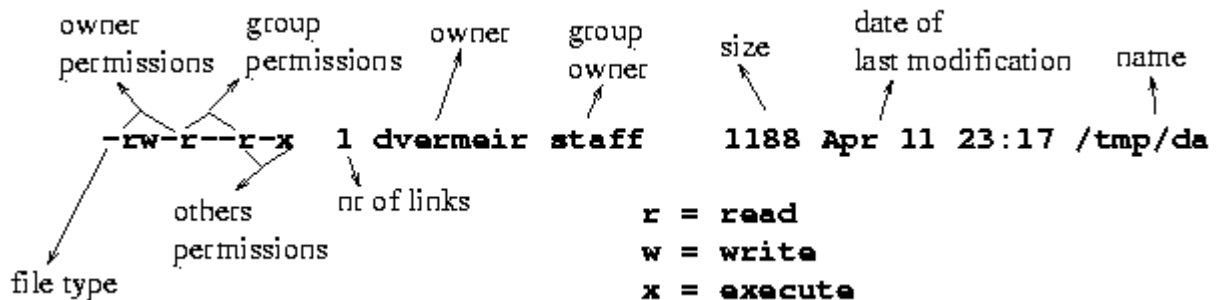
- **Read (r)**
Read access means you can open the file with the open system call and can read the contents of the file with the read system call.
- **Write (w)**
Write access means you can overwrite the file or modify its contents. It gives you access to the system calls write and truncate.
- **Execute(x)**
Execute access means you can specify the path of this file and run it as a program. When a file name is specified to the shell the shell examines the file for execute access and calls the exec system call. The first two bytes of the file are checked for the system magic number, signifying the file is an executable. If the magic number is not contained in the first two bytes the file is assumed to be a shell script.

owner	group	others
rwx	rw-	---

The owner can read write and execute the file
Group members can read and write the file
Other users have no access

The file permissions described above apply to plain files, devices, sockets, and FIFOs. These permissions do not apply to directories and symbolic links. Symbolic links have no permission control on the link, all access is resolved by examining the permissions on the target of the link.

Some anomalies can develop, for example, it is possible to set permissions so that a program can be run but the file cannot be read. Also, it is possible to set permissions so that anyone on the system, except members of your group can read the file.



DIRECTORY PERMISSIONS

While directories are considered to be part of the file system the protection the mode bits take on different meanings in the context of directory files.

- **Read (r)**
Read access permits the `opendir` and `readdir` system calls to be performed on the directory file. Read access on a directory will allow you to see the file names in the directory file.
- **Write (w)**

UNIX FILESYSTEM STRUCTURE BASICS

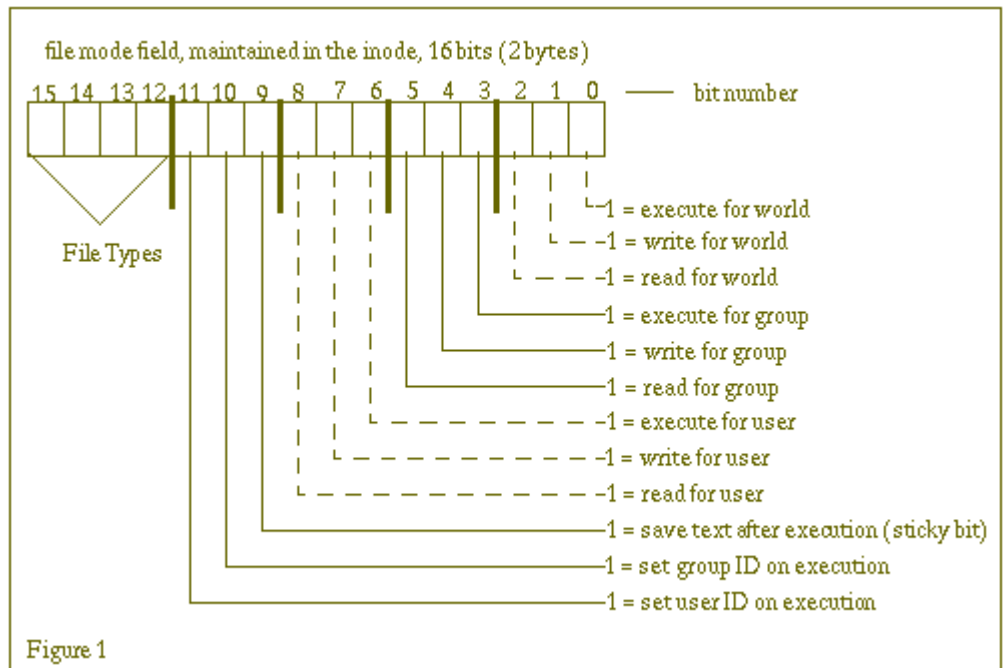
By Mark E. Donaldson

Write access allows you to add or remove file names from the directory file. With write access I can add files to a particular directory or use the rm command to delete a file.

- Execute(x)
Execute access allows access to the stat system call to displays the values in the inode (such as the ls command does). You must have execute access to the directory to make it your current directory or to open files inside that directory.

SUID, SGID, & STICKY BIT FILE PERMISSIONS

Unix allows programs to be endowed with privileges that belong to another user (such as root). Unix uses three of the twelve mode bits to support special permissions. These permissions are named SetUID (SUID), SetGID (SGID), and sticky bit permissions. Files that have the SUID bit set will run with effective user UID of the owner of the file. Files that have the SGID bit set will run with the effective group ID of the group owner of the file. Files with the sticky bit have special properties. Regular files with the sticky bit set are supposed to remain in the swap file after they have finished execution. This was to provide better performance to the system and not force commonly accessed programs to be loaded from swap each time. On directory files, the sticky bit is interpreted in such a way that only the owner of the file in that directory can delete a file. This is generally used with the /tmp directory so that users cannot delete other users files even though all users.



The SUID and SGID permissions are indicated with the ls -l command. A s in the execute field for owner or group indicates SUID or SGID respectively. The sticky bit is indicated in the ls -l command by a t in the execute bit for others.

The setuid and setgid bits provide the ability to have a user execute a file that requires permissions to do things that the user may normally not have. The most common example is the Solaris passwd(1) command. In order for users to change their passwords, they must be able to write to the /etc/passwd and /etc/shadow files. Obviously, it would be a serious security hole to leave these files writeable by anyone. For this reason, the /usr/bin/passwd(1) command is a setuid and setgid file, which means that when the command is executed, the user's effective UID and GID become that of the file, which in this case is root (user) and sys (group). Thus, the setuid and setgid mode bits tell the system to alter the UID and GID of the user executing the file to that of the file they wish to execute. Once execution has completed, the user's UID and GID are restored to whatever they were prior to executing the file. Note that in order for the file owner to set the setuid or setgid mode bits, the file must be executable

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

first (the execute mode must be true for the corresponding setid bit: user execution for `setuid` and group execution for `setgid`).

As said, there are 16 bits (two bytes) available for the file modes, but a long listing via `ls -l` displays only nine of these. Therefore, depending on the file mode, the system will alter what appears in a particular mode field in the output. For `setuid` and `setgid`, a lowercase `s` appears where the execution bit is normally displayed. For executable files that are not `setuid` or `setgid`, the familiar lowercase `x` appears.

It should be obvious that `setuid` programs that make the user "root" have security implications, and thus should be written and implemented with care. Note also that in this example I used the octal notation for the mode bits I wished to set for the file. The `chmod(1)` command also takes ASCII text flags as command line arguments for those users not familiar with octal representation.

The mode bit for `setgid`, bit 10 (see Figure 1), plays a dual role. If bit 10 is set, and the group execute bit is also set (bit 3), bit 10 provides `setgid` behavior as described above. However, if bit 10 is set and bit 3 (group execute) is clear, this means that mandatory locking has been set on the file. Solaris supports two methods of file locking for regular files, *advisory* and *mandatory*. Advisory locking requires that the developer writing the file I/O code uses the proper interfaces for setting and getting file locks. This is typically done with the `fcntl(2)` system call. It is up to the program to follow the rules of checking for locks prior to issuing a read or write to the file. It's analogous to using mutex locks in multithreaded code; the kernel does not prevent a process from grabbing a shared resource that is locked -- the code must explicitly check for the locks. The same applies for advisory file locks.

STICKY BIT

Finally, we get to the *sticky bit*, which is bit 9 in the inode's file mode field. The sticky bit originated as a means of telling the operating system to keep the text portion of an executable file available on the swap device if it had to be moved out of physical memory due to space constraints. It provided a method of keeping commonly used executable text on the faster swap device so the system could retrieve the pages faster if they were needed again after being paged out.

The sticky bit in Solaris affects different file types in different ways. For simple text files, it basically has no affect. For directories, it provides an additional level of security for non-root users that do not own the directory file. If the directory has the sticky bit set, the user must own the directory or the target file in order to remove or rename a file in it, whereas without the sticky bit set on the directory file, a user can remove or rename the file in the directory even if they do not own the file or have write permission to it, as long as they have write permission to the directory.

Put another way, removing a file from a directory requires the user to have write permission for the directory, but not necessarily write or ownership of the underlying file. With the sticky bit set for the directory, the user would not be allowed to remove the file under the same set of conditions unless the user owned the directory, owned the target file, or had write permissions to the target file.

As far as the sticky bit and executable files go, Solaris no longer implements the traditional "save-text-in-swap" functionality that originated with the sticky bit. Again, with the natural evolution of the virtual memory system design, much faster disks, shared object libraries, and memory page locking capability, this type of functionality is somewhat dated. Solaris does, however, use the sticky bit for one other case. For swap files that are created using the `mkfile(1M)` command, Solaris sets a flag in the vnode that identifies the file as a swap file. This flag instructs the kernel not to cache this file in the page cache (because it's a swap file, it's already in memory1).

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

Reference the `chmod(1)` and `chmod(2)` manual pages for information on how to set file modes and descriptions of the various mode fields.

SETTING FILE PROTECTIONS

umask

The `umask` command sets the default file protection for your process. The default file protection on the system is 0666, a `umask` value is subtracted from the default to give a new user specified default. Thus a `umask` of 0077 denies all users except your own. A `umask` of 022 eliminates execute mode.

chmod

The `chmod [-Rf] mode filelist` command (change mode) sets the file permission for a set of files. The command supports two options `-R` and `-f`. The `-R` option is used to recursively move down through the file system and select files matching the filelist. The `-f` option is used to keep errors from being reported back (when used in shell scripts). The values you can specify are either a absolute mode mask or a set of letters. When using the absolute mode mask the value is specified as an three digit octal number and each octal number corresponds to a user category (owner, group, or other). Since an octal number requires three bits each bit of the number corresponds to a file permission (read, write, or execute). Thus using the octal number 7 which has a binary value of 111 which implies read, write, and execute access.

4000 - Setuid on execution

2000 - setgid on execution

1000 - set sticky bit

0400 - read by owner

0200 - write by owner

0100 - execute by owner

0040 - read by group

0020 - wr

0010 - execute by group

0004 - read by others

0002 - write by others

0001 - execute by others

A Common file setting:

0755 Owner has RWX, Group has RX, Others have RX.

0700 Owner has RWX, Group has none, Other have none

4755 Owner has RWX, Group has RX, Others have RX, SUID bit is set.

PROBLEMS WITH SUID

SUID programs have a place on the system, in fact it would be difficult, if not impossible, to run Unix without SUID programs. For example, the file `/bin/passwd` is a SUID program owned by root. This makes sense, you want to restrict access to changing passwords in some way and SUID is a good way to do that. That said, SUID programs are the most frequent way that security is compromised. Problems with SUID programs fall into the following two categories.

Carelessness - The root user leaves an unattended root window and someone uses that window to create a backdoor into the system.

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson

Poor Software Design - Often programs are written SUID that can be written without being SUID root. Also, many programmers don't fully understand the consequences of the code they write and leave unintended back doors into the system.

To identify SUID programs on your system use the command `find`. The `find` command with the correct options will list all files that are SUID or SGID on the system.

```
find / -perm -004000 -o -perm -002000 -type f -print
```

CHANGING OWNERSHIP OF A FILE

The `chown` command is used to change the ownership of files on the system. The format of the command is:

```
chown [-Rf] owner.group
```

Under BSD based systems you must be root to use the `chown` command. Under SYSV, the `chown` command can be run by any user. This has interesting implications when running disk quotas. You will occasionally see users give files to other users so the file will count against the disk quota of the new owner. There is also a `chgrp` command for changing group ownership only. The `chown` command can set both the owner and group of a file and is the command generally used.

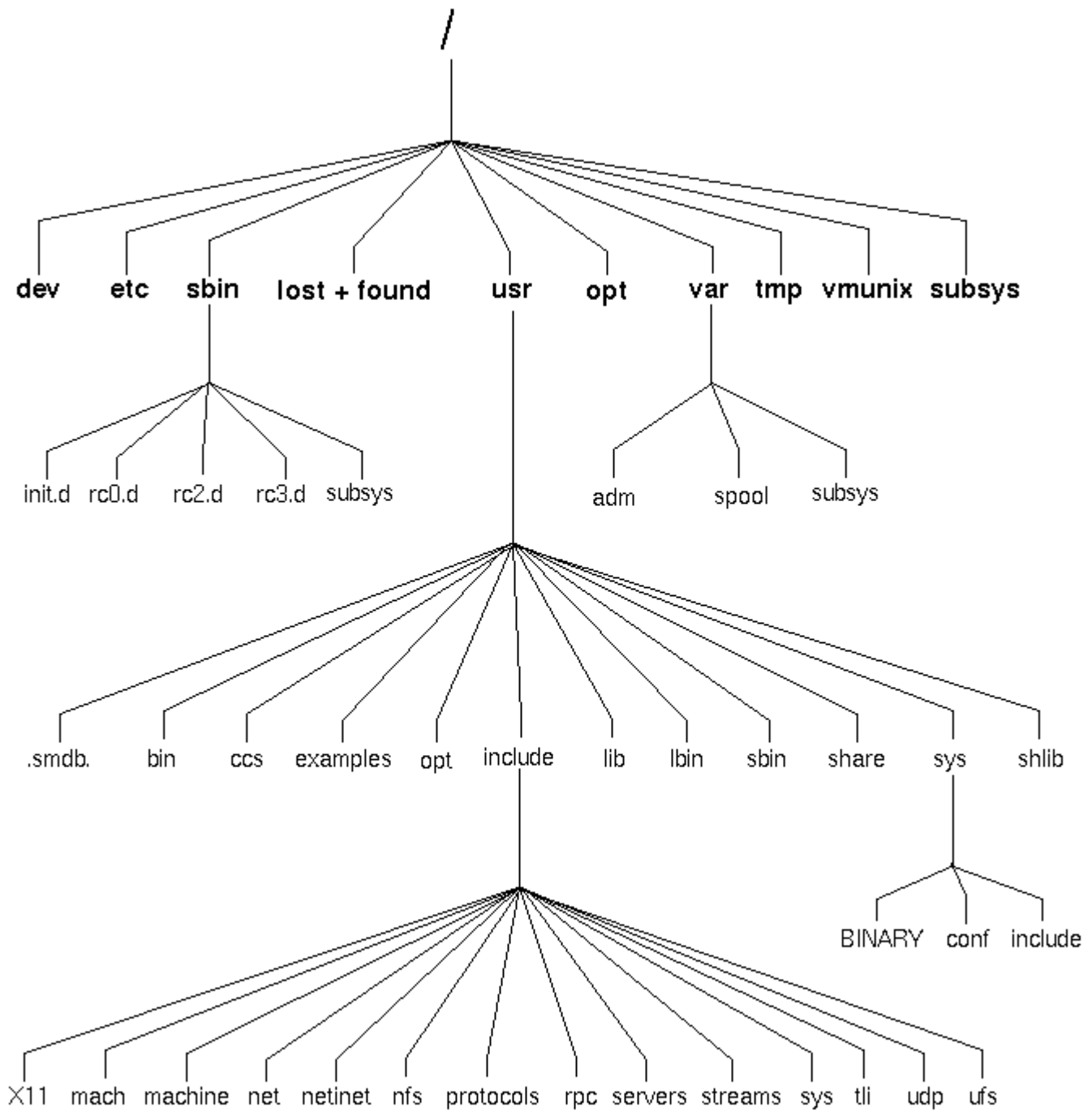
CREATING LINKS

Unix provides both hard and symbolic links to files. Hard links must reside on the same file system. A hard link is essentially a direct reference to the file by a another name. Both files share the same inode and the inode increases the reference count of the file by one for each link. The `ln` command creates hard links. The format of the command is `ln existing-file new-file`.

A symbolic link is a file that indirectly points to another file. While the affect is the same as a hard link a symlink does not increase the reference count of a file. A symbolic link also can point to files on other file systems and files that don't even exist! The `ln -s` command creates a symbolic link. The format of the command is `ln -s existing-file new-file`. Symbolic links can greatly aid a system administrator when file systems must be redone. Through the use of symbolic links you can create your own view of the file system to keep up consistency for users.

UNIX FILESYSTEM STRUCTURE BASICS

By Mark E. Donaldson



ZK-0851U-R