File systems

UNIX owes much of its success to the simplicity and flexibility of the facilities it offers for file handling, generally called the *file system*. This term can have two different meanings:

1. It can be a part of a disk or floppy which can be accessed as a collection of files. It includes regular files and directories. A floppy is usually a single file system, whereas a hard disk can be partitioned into several file systems and possibly also non-file system parts, such as swap space and bad track areas.
2. It can be the software in the kernel which accesses the file systems above.

UNIX has a single file hierarchy, unlike MS-DOS, which uses a separate letter for each file system (`A:` and `B:` for floppies, `C:` to `Z:` for local and network accessible disks). MS-DOS determines the drive letter for the file systems at boot time, whereas UNIX only determines the location of the root file system */* at boot time. You add the other file systems to the directory tree by mounting them:

```
$ mount /dev/usr /usr
```

This mounts the file system on the disk partition */dev/usr* onto the directory */usr*, so if the root directory of */dev/usr* contains a file called *foo*, after mounting you can access it as */usr/foo*.

Anything useful is bound to attract people who want to make it more useful, so it should come as no surprise that a large number of “improvements” have been made to the file system in the course of time. In the rest of this chapter, we’ll look at the following aspects in more detail:

- File systems introduced since the Seventh Edition.
- Differences in function calls, starting on page 206.
- Non-blocking I/O, starting on page 220.
- File locking, starting on page 226.
- Memory-mapped files, starting on page 232.

File system structures

The original Seventh Edition file system is—at least in spirit—the basis for all current file system implementations. All UNIX file systems differ in one important point from almost all non-UNIX file systems:

- At the lowest level, the file system refers to files by numbers, so-called *inodes*. These are in fact indices in the *inode table*, a part of the file system reserved for describing files.
- At a higher level, the directory system enables a file to be referred to by a name. This relationship between a name and an inode is called a *link*, and it enables a single file to have multiple names.

One consequence of this scheme is that it is normally not possible to determine the file name of an open file.

The Seventh Edition file system is no longer in use in modern systems, though the System V file system is quite similar. Since the Seventh Edition, a number of new file systems have addressed weaknesses of the old file system:

- New file types were introduced, such as *symbolic links*, *fifos* and *sockets*.
- The performance was improved.
- The reliability was increased significantly.
- The length of the file names was increased.

We'll look briefly at some of the differences in the next few sections.

The Berkeley Fast File System

The first alternative file system to appear was the Berkeley *Fast File System*, (*FFS*), now called the *Unix File System* (*ufs*).* It is described in detail in *A Fast File System for UNIX*, by Kirk McKusick, Bill Joy, Sam Leffler and Robert Fabry, and *The Design and the Implementation of the 4.3BSD UNIX Operating System* by Sam Leffler, Kirk McKusick, Mike Karels, and John Quarterman. Its main purpose was to increase speed and storage efficiency. Compared to the Seventh Edition file system, the following differences are relevant to porting software:

- The maximum file name size was increased from 14 to 255 characters.
- The size of the inode number was increased from 16 to 32 bits, thus allowing an effectively unlimited number of files.
- *Symbolic links* were introduced.

A symbolic link differs from a normal link in that it points to another file name, and not an inode number.

* Don't confuse the Berkeley FFS with SCO's *afs*, which is sometimes referred to as a Fast File System. In fact, *afs* is very similar to *s5fs*, though later versions have symbolic links and longer file names.

Symbolic links

A symbolic link is a file whose complete contents are the name of another file. To access via a symbolic link, you first need to find the directory entry to which it is pointing, then resolve the link to the inode. By contrast, a traditional link (sometimes called *hard link*) links a file name to an inode. Several names can point to the same inode, but it only takes one step to find the file. This seemingly minor difference has a number of consequences:

- A definite relationship exists between the original file and the symbolic link. In a normal link, each of the file names have the same relationship to the inode; in a symbolic link, the symbolic link name *refers* to the main file name. This difference is particularly obvious if you remove the original file: with a normal link, the other name still works perfectly. With a symbolic link, you lose the file.
- There's nothing to stop a symbolic link from pointing to another symbolic link—in fact, it's quite common, and is moderately useful. It also opens the possibility of looping: if the second symbolic link points back to the first, the system will give up after a few iterations with the error code `ELOOP`.
- Symbolic links have two file permissions. In practice, the permission of the link itself is of little consequence—normally it is set to allow reading, writing and execution for all users (on an *ls -l* listing you see `lrwxrwxrwx`). The permission that counts is still the permission of the original file.
- Symbolic links allow links to different file systems, even (via NFS) to a file system on a different machine. This is particularly useful when using read-only media, such as CD-ROMs. See Chapter 3, *Care and feeding of source trees*, page 39, for some examples.
- Symbolic links open up a whole new area of possible errors. It's possible for a symbolic link to point to a file that doesn't exist, so you can't access the file, even if you have a name and the correct permissions.

Other file systems

Other file systems have emerged since ufs, including:

- The *System V file system, s5fs*, a minor evolution of the Seventh Edition File system with some performance and stability modifications, and without multiplexed files. Even in System V, ufs has replaced it. For all practical purposes, you can consider it to be obsolete.
- The *Veritas File System, vxfs* and the *Veritas Journalling File system, vjfs*. From the point of view of porting, they are effectively compatible with ufs.
- The *Network File System, NFS*,* a method of sharing file systems across networks. It allows a system to mount file systems connected to a different machine. NFS runs on

* People just don't seem to be able to agree whether to write file system names in upper case (as befits an abbreviation), or in lower case (the way most *mount* commands want to see them). It appears that NFS is written in upper case more frequently than the other names.

just about any system, including System V.3 and DOS, but unfortunately not XENIX. It can offer a partial escape from the "14 character file limit, no symlinks" syndrome. It is reasonably transparent, but unfortunately does not support device files.

- *Remote File Sharing, rfs*. This is AT&T's answer to NFS. Although it has a number of advantages over NFS, it is not widely used.

Along with new file systems, new file types have evolved. We have already looked at symbolic links, which we can think of as a new file type. Others include *FIFOs* (First In First Out) and *sockets*, means of interprocess communications that we looked at in Chapter 12, *Kernel dependencies*.

In practice, you run into problems only when you port software developed under ufs, vjfs or vxfs to a s5fs system. If you can, you should change your file system. If you can't do that, here are some of the things that could give you headaches:

- *File name length*. There's very little you can do about this: if the file names are longer than your kernel can understand, you have to change them. There are some subtle problems here: some 14-character file systems accept longer names and just silently truncate them, others, notably SCO, signal an error. It should be fairly evident what your file system does when you try to do it. If your system has the `pathconf` system call, you can also interrogate this programmatically (see page 212).
- *Lack of symbolic links* is another big problem. You may need far-reaching source changes to get around this problem, which could bite you early on in the port: you may have an archive containing symbolic links, or the configuration routines might try to create them.

Another, more subtle difference is that BSD and System V do not agree on the question of group ownership. In particular, when creating a file, the group ownership may be that of the directory, or it may be that of the process that creates the file. BSD always gives the file the group of the directory; in System V.4, it is the group of the process, *unless* the "set group ID" bit is set in the directory permissions, in which case the file will belong to the same group as the directory.

Function calls

The Seventh Edition left a surprising amount of functionality up to the system library. For example, the kernel supplied no method to create a directory or rename a file. The methods that were used to make up for these deficiencies were not always reliable, and in the course of the time these functions have been implemented as system calls. Current systems offer the following functions, some of them system calls:

chsize

`chsize` changes the end of file of an open file.

```
int chsize (int fd, long size);
```

It originated in XENIX and has been inherited by System V.3.2 and System V.4. It corresponds both in function and in parameters to the System V version of `ftruncate`: if the new end-of-file pointer is larger than the current end-of-file pointer, it will extend the file to the new size.

dup2

All systems offer the system call `dup`, which creates a copy of a file descriptor:

```
int dup (int oldd);
```

`oldd` is an open file descriptor; `dup` returns another file descriptor pointing to the same file. The problem with `dup` is that you don't have any control over the number you get back: it's the numerically smallest file descriptor currently not in use. In many cases, you want a specific number. This is what `dup2` does:

```
int dup2 (int oldd, int newd);
```

With `newd` you specify the number of the new descriptor. If it's currently allocated, `dup2` closes it first. You can fake this with `dup`, but it's painful. The `F_DUPFD` subfunction of `fcntl` does the same thing as `dup2`, so you can use it if it is available (see page 208). `dup2` is available on nearly every UNIX system, *including* the Seventh Edition. Somehow some earlier versions of System V don't have it, however—recall that System V derived from the Sixth Edition, not the Seventh Edition. See Chapter 1, *Introduction*, page 4.

fchdir and friends

Various systems offer functions with names like `fchdir`, `fchmod`, `fchown`, and `fchroot`. These are effectively the same as the corresponding functions `chdir`, `chmod`, `chown`, and `chroot`, except they take the number of an open file instead of its name. For example:

```
#include <sys/stat.h>

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

You can replace them with a corresponding call to `ch*` if you know the name of the file associated with the file descriptor; otherwise you could be in trouble.

fcntl

All modern versions of UNIX supply a function called `fcntl`, which is rather like an `ioctl` for disk files:

```
#include <sys/fcntl.h>

int fcntl (int fd, int cmd, union anything arg);
```

Table 14-1 shows common command values.

Table 14–1: `fcntl` commands

Command	System	Meaning
F_DUPFD	all	Duplicate a file descriptor, like `dup`. Return the lowest numbered descriptor that is higher than the `int` value `arg`.
F_GETFD	all	Get the close-on-exec flag associated with `fd`.
F_SETFD	all	Set the close-on-exec flag associated with `fd`.
F_FREESP	SVR4, Solaris 2.X	Free storage space associated with a section of the file `fd`. See the section on file locking on page 230 for more details.
F_GETFL	all	Get descriptor status flags (see below).
F_SETFL	all	Set descriptor status flags to `arg`.
F_GETOWN	BSD	Get the process ID or the complement of the process group currently receiving `SIGIO` and `SIGURG` signals.
F_GETOWN	SVR4	Get the user ID of the owner of the file. This function is not documented for Solaris 2.X.
F_SETOWN	BSD	Set the process or process group to receive `SIGIO` and `SIGURG` signals. If `arg` is negative, it is the complement of the process group. If it is positive, it is a process ID.
F_SETOWN	SVR4	Set the user ID of the owner of the file. This function is not documented for Solaris 2.X.
F_GETLK	all	Get file record lock information. See the section on locking on page 226, for more details.
F_SETLK	all	Set or clear a file record lock.
F_SETLKW	all	Set or clear a file record lock, waiting if necessary until it becomes available.
F_CHKFL	SVR3	Check legality of file flag changes.
F_RSETLK	SVR4	Used by *lockd* to handle NFS locks.
F_RSETLKW	SVR4	Used by *lockd* to handle NFS locks.
F_RGETLK	SVR4	Used by *lockd* to handle NFS locks.

As you can see from the table, `arg` is not always supplied, and when it is, its meaning and type vary depending on the call.

A couple of these functions deserve closer examination:

- F_SETFD and F_GETFD manipulate the *close on exec* flag. This is normally defined in *sys/fcntl.h* as 1. Many programs use the explicit constant 1, which is theoretically non-portable, but which works with current systems.

  By default, exec inherits open files to the new program. If the close on exec flag is set, exec automatically closes the file.

- F_GETOWN and F_SETOWN have very different meanings for BSD and System V.4. In BSD, they get and set the process ID that receives SIGIO and SIGURG signals; in System V.4, they get and set the file owner, which can also be done with stat or fstat. There is no direct equivalent to the BSD F_SETOWN and F_GETOWN in System V, since the underlying implementation of non-blocking I/O is different. Instead, you call ioctl with the I_SETSIG request—see page 225 for more details.

- The request F_CHKFL is defined in the System V.3 header files, but it is not documented.

- F_GETFL and F_SETFL get and set the file status flags that were initally set by open. Table 14-2 shows the flags.

Table 14−2: fcntl file status flags

Flag	System	Meaning
O_NONBLOCK	all	Do not block if the operation cannot be performed immediately. Instead, the read or write call returns -1 with errno set to EWOULDBLOCK.
O_APPEND	all	Append each write to the end of file.
O_ASYNC	BSD	Send a SIGIO signal to the process group when I/O is possible.
O_SYNC	System V	write waits for writes to complete before returning.
O_RDONLY	System V	Open for reading only.
O_RDWR	System V	Open for reading and writing.
O_WRONLY	System V	Open for writing only.

getdents and getdirentries

getdents (System V.4) and getdirentries (BSD) are marginally compatible system calls that read a directory entry in a file-system independent format. Both systems provide a header file */usr/include/sys/dirent.h*, which defines a struct dirent, but unfortunately the structures are different. In System V, the structure and the call are:

```
struct  dirent
  {
  ino_t d_ino;
  off_t d_off;
```

```
  unsigned short d_reclen;
  char d_name[1];
  };

int getdents(int fd, struct dirent *buf, size_t nbyte);
```

`getdirentries` is the corresponding BSD system call:

```
struct dirent
  {
  unsigned long d_fileno;     /* "file number" (inode number) of entry */
  unsigned short d_reclen;    /* length of this record */
  unsigned short d_namlen;    /* length of string in d_name */
  char d_name[MAXNAMLEN + 1]; /* name must be no longer than this */
};

int getdirentries(int fd, char *buf, int nbytes, long *basep);
```

Because of these compatibility problems, you don't normally use these system calls directly—you use the library call `readdir` instead. See the description of `readdir` on page 213 for more information.

getdtablesize

Sometimes it's important to know how many files a process is allowed to open. This depends heavily on the kernel implementation: some systems have a fixed maximum number of files that can be opened, and may allow you to specify it as a configuration parameter when you build a kernel. Others allow an effectively unlimited number of files, but the kernel allocates space for files in groups of about 20. Evidently, the way you find out about these limits depends greatly on the system you are running:

- On systems with a fixed maximum, the constant `NOFILE`, usually defined in */usr/include/sys/param.h*, specifies the number of files you can open.
- On systems with a configurable maximum, you will probably also find the constant `NOFILE`, only you can't rely on it to be correct.
- On some systems that allocate resources for files in groups, the size of these groups may be defined in */usr/include/sys/filedesc.h* as the value of the constant `NDFILE`.
- BSD systems offer the function `getdtablesize` (no parameters) that returns the maximum number of files you can open.
- Modern systems offer the `getrlimit` system call, which allows you to query a number of kernel limits. See Chapter 12, *Kernel dependencies*, page 169, for details of `getrlimit`.

ioctl

`ioctl` is a catchall function that performs functions that weren't thought of ahead of time. Every system has its own warts on *ioctl*, and the most common problem with `ioctl` is a call with a request that the kernel doesn't understand. We can't go into detail about every *ioctl* function, but we do examine terminal driver *ioctl* calls in some depth in Chapter 15, *Terminal drivers*, starting on page 252.

lstat

`lstat` is a version of `stat`. It is identical to `stat` unless the pathname specifies a symbolic link. In this case, `lstat` returns information about the link itself, whereas `stat` returns information about the file to which the link points. BSD and System V.4 support it, and it should be available on any system that supports symbolic links.

ltrunc

`ltrunc` truncates an open file in the same way that `ftruncate` does, but the parameters are more reminiscent of `lseek`:

```
int ltrunc (int fd, off_t offset, int whence);
```

`fd` is the file descriptor. `offset` and `whence` specify the new end-of-file value:

- If `whence` is `SEEK_SET`, `ltrunc` sets the file size to `offset`.
- If `whence` is `SEEK_CUR`, `ltrunc` sets the file size to `offset` bytes beyond the current seek position.
- If `whence` is `SEEK_END`, `ltrunc` increases the file size by `offset`.

No modern mainstream system supports `ltrunc`. You can replace a call `ltrunc (fd, offset, SEEK_SET)` with `ftruncate (fd, offset)`. If you have calls with `SEEK_CUR` and `SEEK_END`, you need to first establish the corresponding offset with a call to `lseek`:

```
ftruncate (fd, lseek (fd, offset, SEEK_CUR)); or SEEK_END
```

mkdir and rmdir

Older versions of UNIX did not supply a separate system call to create a directory; they used `mknod` instead. Unfortunately, this meant that only the superuser could create directories. Newer versions supply `mkdir` and `rmdir`. The syntax is:

```
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode)

#include <unistd.h>
int rmdir (const char *path)
```

If your system does not have the `mkdir` system call, you can simulate it by invoking the

`mkdir` utility with the library function `system`.

open

Since the Seventh Edition, `open` has acquired a few new flags. All modern versions of UNIX support most of them, but the following differ between versions:

- `O_NDELAY` is available only in earlier versions of System V. It applies to devices and FIFOs (see Chapter 12, *Kernel dependencies*, page 165, for more information on FIFOs) and specifies that both the call to `open` and subsequent I/O calls should return immediately without waiting for the operation to complete. A call to `read` returns 0 if no data is available, which is unfortunately also the value returned at end-of-file. If you don't have `O_NDELAY`, or if this ambiguity bugs you, use `O_NONBLOCK`.
- `O_NONBLOCK` specifies that both the call to `open` and subsequent I/O calls should return immediately without waiting for completion. Unlike `O_NDELAY`, a subsequent call to `read` returns -1 (error) if no data is available, and `errno` is set to `EAGAIN`.
- System V.4 and 4.4BSD have a flag, called `O_SYNC` in System V.4 and `O_FSYNC` in 4.4BSD, which specifies that each call to write `write` should write any buffered data to disk and update the inode. Control does not return to the program until these operations complete. If your system does not support this feature, you can probably just remove it, though you lose a little bit of security. To *really* do the Right Thing, you can include a call to `fsync` after every I/O.

pathconf and fpathconf

`pathconf` and `fpathconf` are POSIX.1 functions that get configuration information for a file or directory:

```
#include <unistd.h>
long fpathconf (int fd, int name);
long pathconf (const char *path, int name);
```

The parameter `name` is an `int`, not a name. Despite what it is called, it specifies the *action* to perform:

Table 14–3: `pathconf` actions

name	Function
`_PC_LINK_MAX`	Return the maximum number of links that can be made to an inode.
`_PC_MAX_CANON`	For terminals, return the maximum length of a formatted input line.
`_PC_MAX_INPUT`	For terminals, return the maximum length of an input line.
`_PC_NAME_MAX`	For directories, return the maximum length of a file name.

Table 14−3: `pathconf` actions (continued)

name	Function
_PC_PATH_MAX	Return the maximum length of a relative path name starting with this directory.
_PC_PIPE_BUF	For FIFOs, return the size of the pipe buffer.
_PC_CHOWN_RESTRICTED	return TRUE if the chown system call may not be used on this file. If fd or path refer to a directory, then this information applies to all files in the directory.
_PC_NO_TRUNC	return TRUE if an attempt to create a file with a name longer than the maximum in this directory would fail with ENAMETOOLONG.
_PC_VDISABLE	For terminals, return TRUE if special character processing can be disabled.

read

The function `read` is substantially unchanged since the Seventh Edition, but note the comments about `O_NDELAY` and `O_NONBLOCK` in the section about `open` on page 212.

rename

Older versions of UNIX don't have a system call to rename a file: instead, they make a link and then delete the old file. This can cause problems if the process is stopped in the middle of the operation, and so the atomic `rename` function was introduced. If your system doesn't have it, you can still do it the old-fashioned way.

revoke

`revoke` is used in later BSD versions to close all file descriptors associated with a special file, even those opened by a different process. It is *not* available with System V.4. Typically, this call is used to disconnect serial lines.

After a process has called `revoke`, a call to `read` on the device from any process returns an end-of-file indication, a call to `close` succeeds, and all other calls fail. Only the file owner and the super user may use this call.

readdir and friends

In the Seventh Edition, reading a directory was simple: directory entries were 16 bytes long and consisted of a 2-byte *inode number* and a 14 byte file name. This was defined in a `struct direct`:

```
struct direct
  {
  ino_t d_ino;
  char  d_name[DIRSIZ];
```

```
  };
```

With the introduction of *ufs*, which supports names of up to 256 characters, it was no longer practical to reserve a fixed-length field for the file name, and it became more difficult to access directories. A family of directory access routines was introduced with 4.2BSD:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *filename);
struct dirent *readdir (DIR *dirp);
long telldir (const DIR *dirp);
void seekdir (DIR *dirp, long loc);
void rewinddir (DIR *dirp);
int closedir (DIR *dirp);
int dirfd (DIR *dirp);
```

Along with the `DIR` type, there is a `struct dirent` that corresponds to the Seventh Edition `struct direct`. Unfortunately, System V defines `struct dirent` and `DIR` differently from the original BSD implementation. In BSD, it is

```
struct dirent                   /* directory entry */
  {
  unsigned long d_fileno;     /* file number of entry */
  unsigned short d_reclen;    /* length of this record */
  unsigned short d_namlen;    /* length of string in d_name */
  char d_name [255 + 1];      /* maximum name length */
  };

/* structure describing an open directory. */
typedef struct _dirdesc
  {
  int   dd_fd;                  /* directory file descriptor */
  long  dd_loc;                 /* offset in current buffer */
  long  dd_size;       /* amount of data from getdirentries */
  char  *dd_buf;       /* data buffer */
  int   dd_len;                 /* size of data buffer */
  long  dd_seek;       /* magic cookie from getdirentries */
  } DIR;
```

System V defines

```
struct dirent
  {
  ino_t d_ino;                  /* inode number of entry */
  off_t d_off;                  /* offset of directory entry */
  unsigned short d_reclen;    /* length of this record */
  char d_name [1];              /* name of file */
  };

typedef struct
  {
  int dd_fd;                    /* file descriptor */
  int dd_loc;                   /* offset in block */
  int dd_size;                  /* amount of valid data */
```

```
char *dd_buf;                /* directory block */
} DIR;              /* stream data from opendir() */
```

There are a number of ugly incompatibilities here:

- The field `d_fileno` in the BSD `dirent` struct is not a file descriptor, but an inode number. The System V name `d_ino` makes this fact clearer, but it introduces a name incompatiblity.
- A number of the BSD fields are missing in the System V structures. You can calculate `dirent.d_namlen` by subtracting the length of the other fields from `dirent.d_reclen`. For example, based on the System V `dirent` structure above:

```
d_namlen = dirent.d_reclen
          - sizeof (ino_t)   /* length of the d_ino field */
          - sizeof (d_off)   /* length of the d_off field */
          - sizeof (unsigned short); /* length of the d_reclen field */
```

System V.4 has two versions of these routines: a System V version and a BSD version. Many reports have claimed that the BSD version is broken, though it's possible that the programmers were using the wrong header files. If you *do* run into trouble, you should make sure the header files match the flavour of `dirent` and `DIR` that you have.

readv and writev

`readv` and `writev` perform a so-called *scatter read* and *gather write*. These functions are intended to write to a file a number of pieces of data spread in memory, or to read from a file to a number of places.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>
in sys/uio.h is the definition:
struct iovec
  {
  caddr_t iov_base;
  int     iov_len;
  };

int readv(int d, struct iovec *iov, int iovcnt);
int writev (int d, struct iovec *iov, int iovcnt);
```

Each `iovec` element specifies an address and the number of bytes to transfer to or from it. The total number of bytes transferred would be the sum of the `iov_len` fields of all `iovcnt` elements. `readv` and `writev` are available only for BSD and System V.4 systems—if you don't have them, it's relatively easy to fake them in terms of `read` or `write`. The reasons why these calls exist at all are:

- Some devices, such as tape drives, write a physical record for each call to `write`. This can result in a significant drop in performance and tape capacity.

- For tape drives, the only alternative is to copy the data into one block before writing. This, too, impacts performance, though not nearly as much as writing smaller blocks.
- Even for devices that don't write a physical block per `write`, it's faster to do it in the kernel with just a single function call: you don't have as many context switches.

statfs and statvfs

`statfs` or `statvfs` return information about a file system in a format referred to as a *generic superblock*. All current UNIX versions supply one or the other of these functions, but the information they return varies greatly. XENIX, System V.3, BSD, and BSD-derived SunOS operating systems supply `statfs`. System V.4 supplies `statvfs`.

BSD systems define `statfs` like this:

```
typedef quad fsid_t;

#define MNAMELEN 32            /* length of buffer for returned name */

struct statfs
  {
  short   f_type;      /* type of filesystem (see below) */
  short   f_flags;             /* copy of mount flags */
  long    f_fsize;             /* fundamental file system block size */
  long    f_bsize;             /* optimal transfer block size */
  long    f_blocks;            /* total data blocks in file system */
  long    f_bfree;             /* free blocks in fs */
  long    f_bavail;            /* free blocks avail to non-superuser */
  long    f_files;             /* total file nodes in file system */
  long    f_ffree;             /* free file nodes in fs */
  fsid_t  f_fsid;      /* file system id */
  long    f_spare[6];          /* spare for later */
  char    f_mntonname[MNAMELEN]; /* mount point */
  char    f_mntfromname[MNAMELEN]; /* mounted filesystem */
  };
```

SunOS 4.1.3 defines them as:

```
#include <sys/vfs.h>

typedef struct
  {
  long    val[2];
  } fsid_t;
struct statfs
  {
  long    f_type;      /* type of info, zero for now */
  long    f_bsize;             /* fundamental file system block size */
  long    f_blocks;            /* total blocks in file system */
  long    f_bfree;             /* free blocks */
  long    f_bavail;            /* free blocks available to non-super-user */
  long    f_files;             /* total file nodes in file system */
  long    f_ffree;             /* free file nodes in fs */
```

```
  fsid_t  f_fsid;      /* file system id */
  long    f_spare[7];          /* spare for later */
  };
```

System V.3 and XENIX define:

```
struct  statfs
  {
  short f_fstyp;       /* File system type */
  long  f_bsize;       /* Block size */
  long  f_frsize;      /* Fragment size (if supported) */
  long  f_blocks;      /* Total number of blocks on file system */
  long  f_bfree;       /* Total number of free blocks */
  long  f_files;       /* Total number of file nodes (inodes) */
  long  f_ffree;       /* Total number of free file nodes */
  char  f_fname[6];            /* Volume name */
  char  f_fpack[6];            /* Pack name */
  };

int statfs (const char *path, struct statfs *buf);
int fstatfs (int fd, struct statfs *buf);
```

System V.4 and Solaris 2.X use `statvfs`, which is defined as

```
#include <sys/types.h>
#include <sys/statvfs.h>

struct statvfs
  {
  u_long  f_bsize;             /* preferred file system block size */
  u_long  f_frsize;            /* fundamental filesystem block size */
  u_long  f_blocks;            /* total # of blocks on file system */
  u_long  f_bfree;             /* total # of free blocks */
  u_long  f_bavail;            /* # of free blocks available */
  u_long  f_files;             /* total # of file nodes (inodes) */
  u_long  f_ffree;             /* total # of free file nodes */
  u_long  f_favail;            /* # of inodes available */
  u_long  f_fsid;      /* file system id (dev for now) */
  char    f_basetype [FSTYPSZ];   /* target fs type name */
  u_long  f_flag;      /* bit mask of flags */
  u_long  f_namemax;           /* maximum file name length */
  char    f_fstr [32];         /* file system specific string */
  u_long  f_filler [16];      /* reserved for future expansion */
  };

#define ST_RDONLY  0x01        /* read-only file system */
#define ST_NOSUID  0x02        /* does not support setuid/setgid */
#define ST_NOTRUNC 0x04        /* does not truncate long file names */

int statvfs (const char *path, struct statvfs *buf);
int fstatvfs (int fd, struct statvfs *buf);
```

There's not much to say about these functions: if you have problems, hopefully this information will help you figure out what the author intended.

symlink

`symlink` creates a symbolic link in file systems that support symbolic links:

```
#include <unistd.h>

int symlink (const char *real_name, const char *symbolic_name);
```

A symbolic link `symbolic_name` is created to the name `real_name`.

sysfs

`sysfs` is a System V function that returns information about the kinds of file systems configured in the system. This function has the rather strange property of not being compatible with ANSI C—the parameters it accepts depend on the function supplied:

```
#include <sys/fstyp.h>
#include <sys/fsid.h>

int sysfs ((int) GETFSIND, const char *fsname);
```

This call translates `fsname`, a null-terminated file-system type identifier, into a file-system type index.

```
int sysfs ((int) GETFSTYP, int fs_index, char *buf);
```

This call translates `fs_index`, a file-system type index, into a `NUL`-terminated file-system type identifier in the buffer pointed to by buf.

```
int sysfs((int) GETNFSTYP);
```

This call returns the total number of file system types configured in the system.

truncate and ftruncate

These functions set the EOF pointer of a file. `truncate` finds the file via its file name, and `ftruncate` requires the file number of an open file.

```
#include <unistd.h>
int truncate (const char *path, off_t length);
int ftruncate (int fd, off_t length);
```

These functions are available with BSD and System V.4. There is a subtle difference between the way the BSD and System V.4 versions work: if the file is smaller than the requested length, System V.4 extends the file to the specified length, while BSD leaves it as it is. Both versions discard any data beyond the end if the current EOF is longer.

If your system doesn't have these functions, you may be able to perform the same function with `chsize` (page 206) or the `fcntl` function `F_FREESP` (page 208).

ustat

`ustat` returns information about a mounted file system, and is supported by System V and SunOS 4, but not by BSD. The call is:

```
struct ustat
  {
  daddr_t f_tfree;              /* Total blocks available */
  ino_t f_tinode;               /* Number of free inodes */
  char f_fname [6];             /* File system name */
  char f_fpack [6];             /* File system pack name */

int ustat (dev_t dev, struct ustat *buf);
```

On BSD systems, you can get this information with the `statfs` system call, which requires a path name instead of a device number.

utime and utimes

`utime` is available in all versions of UNIX.

```
#include <sys/types.h>
#include <utime.h>

int utime (const char *path, const struct utimbuf *times);
```

`utime` sets the modification timestamp of the file defined by `path` to the time specified in `times`. In the Seventh Edition, `times` was required to be a valid pointer, and only the file owner or `root` could use the call. All newer versions of UNIX allow `times` to be a `NULL` pointer, in which case the modification timestamp is set to the current time. Any process that has write access to the file can use `utime` in this manner. BSD implements this function in the C library in terms of the function `utimes`:

```
#include <sys/time.h>
sys/time.h defines:
struct timeval
  {
  long tv_sec;                  /* seconds */
  long tv_usec;                 /* and microseconds */
};
int utimes (const char *file, const struct timeval *times);

#include <sys/types.h>
#include <utime.h>
utime.h defines:
struct utimbuf
  {
  time_t actime;       /* access time */
  time_t modtime;      /* modification time */
  };

int utime (char *path, struct utimbuf *times);
```

The difference between `utime` and `utimes` is simply in the format of the access time: `utime` supplies the time in `time_t` format, which is accurate to a second, whereas `utimes` uses the `timeval` struct which is (theoretically) accurate to one microsecond. BSD systems supply the `utime` function as a library call (which, not surprisingly, calls `utimes`). On XENIX and early System V systems you can fake `utimes` using `utime`.

Non-blocking I/O

In early versions of UNIX, all device I/O was *blocking*: if you made a call to `read` and no data was available, or if you made a call to `write` and the device wasn't ready to accept the data, the process would sleep until the situation changed. This is still the default behaviour.

Blocking I/O can be restrictive in many situations, and many schemes have been devised to allow a process to continue execution before the I/O operation completes. On current systems, you select non-blocking I/O either by supplying the flag `O_NONBLOCK` to `open`, or by calling the `fcntl` function `F_SETFL` with the `O_NONBLOCK` flag (see page 209).

One problem with non-blocking I/O is that you don't automatically know when a request is complete. In addition, if you have multiple requests outstanding, you may not really care which finishes first, you just want to know when one finishes.

Two approaches have been used to inform a process when a request completes. One is to call a function that returns information about current request status, and that may optionally block until something completes. Traditionally, BSD uses `select` to perform this function, whereas System V uses `poll`.

The other solution is to send a signal (`SIGPOLL` in System V, `SIGIO` or `SIGURG` in BSD) when the request finishes. In both systems, this has the disadvantage of not supplying any information about the request that completed, so if you have more than one request outstanding, you still need to call `select` or `poll` to handle the situation.

select

`select` is called with the following parameters:

```
#define FD_SETSIZE 512  my maximum FD count, see below
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
```

These header files define the structs:

```
typedef struct fd_set
  {
  fd_mask fds_bits [howmany (FD_SETSIZE, NFDBITS)];
  } fd_set;

struct timeval
  {
  long  tv_sec;                /* seconds */
  long  tv_usec;       /* and microseconds */
```

```
};

int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *timeout);
```

The parameters `readfds`, `writefds`, and `exceptfds` are bit maps, one bit per possible file descriptor. Recall that file descriptors are small non-negative integers. `select` uses the file descriptor as an index in the bit map.

This gives us a problem when porting: we don't know how many files our implementation supports. In modern systems, there is usually no fixed limit. The solution chosen is a kludge: "choose a sufficiently high number". The expression `howmany (FD_SETSIZE, NFDBITS)` evaluates to the number of words of `NFDBITS` required to store `FD_SETSIZE` bits:

```
#define howmany(bits, wordsize)  ((bits + wordsize - 1) / wordsize)
```

In 4.4BSD `FD_SETSIZE` defaults to 256 (in *sys/types.h*). Nowadays, a server with many requestors could quite easily exceed that value. Because of this, you can set it yourself: just define `FD_SETSIZE` before including */usr/include/sys/types.h*, as indicated in the syntax overview above.

Setting variables of type `fd_mask` is tricky, so a number of macros are supplied:

```
FD_SET (fd, &fdset)            /* set bit fd in fdset*/
FD_CLR (fd, &fdset)            /* clear bit fd in fdset */
FD_ISSET (fd, &fdset)          /* return value of bit fd in fdset */
FD_ZERO (&fdset)       /* clear all bits in fdset */
```

`select` examines the files specified in `readfds` for read completion, the files specified in `writefds` for write completion and the files specified in `exceptfds` for exceptional conditions. You can set any of these pointers to `NULL` if you're not interested in this kind of event. The action that `select` takes depends on the value of `timeout`:

- If `timeout` is a `NULL` pointer, `select` blocks until a completion occurs on one of the specified files.
- If both `timeout->tv_sec` and `timeout->tv_usec` are set to 0, `select` checks for completions and returns immediately.
- Otherwise `select` waits for completion up to the specified timeout.

`select` returns -1 on error conditions, and the number of ready descriptors (possibly 0) otherwise. It replaces the contents of `readfds`, `writefds`, and `exceptfds` with bit maps indicating which files had a corresponding completion.

So far, we haven't even mentioned `nfds`. Strictly speaking, it's not needed: you use it to indicate the *number* of file descriptors that are worth examining. By default, `open` and `dup` allocate the lowest possible file descriptors, so `select` can save a lot of work if you tell it the highest file number that is worth examining in the bit maps. Since file descriptors start at 0, the *number* of file descriptors is 1 higher than the highest file descriptor number.

This baroque function has a couple of other gotchas waiting for you:

- The state of `readfds`, `writefds`, and `exceptfds` is undefined if `select` returns 0 or -1. System V clears the descriptors, whereas BSD leaves them unchanged. Some System V programs check the descriptors even if 0 is returned: this can cause problems if you port such a program to BSD.
- The return value is interpreted differently in BSD and System V. In BSD, each completion event is counted, so you can have up to 3 completions for a single file. In System V, the number of files with completions is returned.
- On completion without timeout, Linux decrements the value of `timeout` by the time elapsed since the call: if `timeout` was initially set to 30 seconds, and I/O completes after 5 seconds, the value of `timeout` on return from `select` will be 25 seconds. This can be of use if you have a number of outstanding requests, all of which must complete in a certain time: you can call `select` again for the remaining file descriptors without first calculating how much time remains.

  In Linux, this feature can be disabled by setting the `STICKY_TIMEOUTS` flag in the COFF/ELF personality used by the process. Other versions of UNIX do not currently suppport this feature, although both System V and BSD suggest that it will be implemented. For example, the man pages for 4.4BSD state:

  > **Select**() should probably return the time remaining from the original timeout, if any, by modifying the time value in place. This may be implemented in future versions of the system. Thus, it is unwise to assume that the timeout value will be unmodified by the *select*() call.

If you find a system without `select` that does support `poll`, you can probably replace `select` with `poll`—it's just a SMOP.*

Typical use of select

Programs which use `select` generally start a number of I/O transfers and then go to some central place to wait for something to happen. The code could look like:

```
if (select (maxfnum,           /* number of files to check */
            &readfds,          /* mask of read completions */
             &writefds,        /* mask of write completions */
            &exceptfds,        /* mask of exception completions */
            NULL) > 0)         /* no timeout */
  {                            /* we have completions, */
  int fd;
  for (fd = 0; fd < maxfnum; fd++)
    {
    if (FD_ISSET (fd, readfds)) /* this file has a read completion */
      read_completion (fd);   /* process it */
    if (FD_ISSET (fd, writefds)) /* this file has a write completion */
      write_completion (fd);  /* process it */
    if (FD_ISSET (fd, exceptfds)) /* this file has a exception completion */
```

* To quote the New Hacker's Dictionary: *SMOP: /S-M-O-P/ [Simple (or Small) Matter of Programming] n. 2. Often used ironically . . . when a suggestion for a program is made which seems easy to the suggester, but is obviously (to the victim) a lot of work.*

```
        exception_completion (fd); /* process it */
      }
```

As we saw above, FD_ISSET is a macro which checks if bit fd is set in the bit mask. The *foo*_completion functions do whatever is needed on completion of I/O for this file descriptor. See *Advanced Programming in the UNIX environment*, by Richard Stevens, for further information.

poll

poll takes a different approach from select:

```
#include <stropts.h>
#include <poll.h>

...  in poll.h is the definition
struct pollfd
  {
  int fd;                       /* file descriptor */
  short events;                 /* requested events */
  short revents;              /* returned events */
  };

int poll (struct pollfd *fds, unsigned long nfds, int timeout);
```

For each file of interest, you set up a pollfd element with the file number and the events of interest. events and revents are again bit maps. events can be made up of the following values:

Table 14−4: poll event codes

Event	Meaning
POLLIN	Data other than high priority data is available for reading.
POLLRDNORM	Normal data* (priority band = 0) is available for reading.
POLLRDBAND	Data from a non-zero priority band is available for reading.
POLLPRI	High priority data is available for reading.
POLLOUT	Normal data may be written without blocking.
POLLWRNORM	The same as POLLOUT: normal data may be written without blocking.
POLLWRBAND	Priority data (priority band > 0) may be written without blocking.

When it succeeds, poll sets the corresponding bits in revents to indicate which events

* STREAMS recognizes 256 different *data priority bands*. Normal data is sent with priority band 0, but urgent data with a higher priority can "leapfrog" normal data. See *UNIX Network Programming*, by W. Richard Stevens, for further information.

occurred. In addition, it may set the following event bits:

Table 14−5: `poll` result codes

Event	Meaning
`POLLERR`	An error has occurred on the device or stream.
`POLLHUP`	A hangup has occurred.
`POLLNVAL`	The specified fd is not open.

Timeout processing is nearly the same as for `select`, but the parameter `timeout` is specified in milliseconds. Since it is an `int`, not a pointer, you can't supply a `NULL` pointer; instead, you set the value to `INFTIM` (defined in *stropts.h*) if you want the call to block.. To summarize:

- If `timeout` is set to `INFTIM`, `poll` blocks until a completion occurs on one of the specified files.
- If `timeout` is set to 0, a check is made for completions and `poll` returns immediately.
- If `timeout` is non-zero, `poll` waits for completion up to the specified timeout.

Typical use of poll

Like `select`, programs which use `poll` generally start a number of I/O transfers and then go to some central place to wait for something to happen. In this case, the code could look like:

```
if (poll (pollfds, maxfnum, NULL) > 0) /* wait for something to complete */
  {
  int fd;
  for (fd = 0; fd < maxfnum; fd++)
    {
    if (pollfds [fd].revents) /* something completed */
      ... check the result bits which interest you and
      perform the appropriate actions
    }
  }
```

The code for starting the request and enabling `SIGIO` and `SIGURG` for the line assumes that the file has been opened and the number stored in an array of file numbers.

rdchk

`rdchk` is a XENIX function that checks if data is available for reading on a specific file descriptor:

```
int rdchk (int fd);
```

It returns 1 if data is available, 0 if no data is currently available, and -1 on error (and `errno` is set). If you don't have it, you can implement it in terms of `select` or `poll`.

SIGPOLL

System V systems can arrange to have the signal SIGPOLL delivered when a request completes. It is not completely general: the file in question must be a STREAMS device, since only STREAMS drivers generate the SIGPOLL signal.

The ioctl call I_SETSIG enables SIGPOLL. The third parameter specifies a bit mask of events to wait for:

Table 14−6: I_SETSIG event mask bits

Mask bit	Event
S_INPUT	A normal priority message is on the read queue.
S_HIPRI	A high priority message is on the read queue.
S_OUTPUT	The write queue is no longer full.
S_WRNORM	The same thing as S_OUTPUT: The write queue is no longer full.
S_MSG	A signal message is at the front of the read queue.
S_ERROR	An error message has arrived at the stream head.
S_HANGUP	A hangup message has arrived at the stream head.
S_RDNORM	A normal message is on the read queue.
S_RDBAND	An out of band message is on the read queue.
S_WRBAND	We can write out of band data.
S_BANDURG	In conjunction with S_RDBAND, generate SIGURG instead of SIGPOLL.

In addition to the call to ioctl, the process needs to set up a signal handler for SIGPOLL—the default disposition is to terminate the process, which is probably not what you want.

SIGIO

BSD systems have a similar mechanism to SIGPOLL, called SIGIO. Like SIGPOLL, it also has its restrictions: it can be applied only to terminal or network devices. In addition, when out-of-band data* arrives, a second signal, SIGURG, is generated. SIGIO and SIGURG are enabled by the O_ASYNC flag to open and a couple of calls to fcntl—see page 209 for more details:

- First, specify the process or process group that should receive the signals, using the fcntl subfunction F_SETOWN in order to enable reception of SIGURG.
- If you want to use SIGIO, set the O_ASYNC file status flag with the fcntl subfunction F_SETFL.
- As with System V, you need to define a signal handler for SIGIO and SIGURG.

* Sockets use the term *out-of-band* to refer to data which comes in at a higher priority, such as TCP urgent mode. Like STREAMS priority data, this data will be presented ahead of normal data.

File locking

The Seventh Edition did not originally allow programs to coordinate concurrent access to a file. If two users both had a file open for modification at the same time, it was almost impossible to prevent disaster. This is an obvious disadvantage, and all modern versions of UNIX supply some form of file locking.

Before we look at the functions that are available, it's a good idea to consider the various kinds of lock. There seem to be two of everything. First, the *granularity* is of interest:

file locking — applies to the whole file.

range locking — applies only to a range of byte offsets. This is sometimes misleadingly called *record locking*.

With file locking, no other process can access the file when a lock is applied. With range locking, multiple locks can coexist as long as their ranges don't overlap. Secondly, there are two types of lock:

Advisory locks — do not actually prevent access to the file. They work only if every participating process ensures that it locks the file before accessing it. If the file is already locked, the process blocks until it gains the lock.

mandatory locks — prevent (block) read and write access to the file, but do not stop it from being removed or renamed. Many editors do just this, so even mandatory locking has its limitations.

Finally, there are also two ways in which locks cooperate with each other:

exclusive locks — allow no other locks that overlap the range. This is the only was to perform file locking, and it implies that only a single process can access the file at a time. These locks are also called also called *write locks*.

shared locks — allow other shared locks to coexist with them. Their main purpose is to prevent an exclusive lock from being applied. In combination with mandatory range locking, a write is not permitted to a range covered by a shared lock. These locks are also called *read locks*.

There are five different kinds of file or record locking in common use:

- *Lock files*, also called *dot locking*, is a primitive workaround used by communication programs such as *uucp* and *getty*. It is independent of the system platform, but since it is frequently used we'll look at it briefly. It implements advisory file locking.
- After the initial release of the Seventh Edition, a file locking package using the system call `locking` was introduced. It is still in use today on XENIX systems. It implements mandatory range locking.
- BSD systems have the system call `flock`. It implements advisory file locking.
- System V, POSIX.1, and more recent versions of BSD support range locking via the `fcntl` system call. BSD and POSIX.1 systems provide only advisory locking. System V supplies a choice of advisory or mandatory locking, depending on the file permissions. If you need to rewrite locking code, this is the method you should use.

- System V also supplies range locking via the `lockf` library call. Again, it supplies a choice of advisory or mandatory locking, depending on the file permissions.

The decision between advisory and mandatory locking in System V depends on the file permissions and not on the call to `fcntl` or `lockf`. The *setgid* bit is used for this purpose. Normally, in executables, the *setgid* bit specifies that the executable should assume the effective group ID of its owner group when `exec`ed. On files that do not have group execute permission, it specifies mandatory locking if it is set, and advisory locking if it is not set. For example,

- A file with permissions 0764 (`rwxrw-r--`) will be locked with advisory locking, since its permissions include neither group execute nor setgid.
- A file with permissions 0774 (`rwxrwxr--`) will be locked with advisory locking, since its permissions don't include setgid.
- A file with permissions 02774 (`rwxrwsr--`) will be locked with advisory locking, since its permissions include both group execute and setgid.
- A file with permissions 02764 will be locked with mandatory locking, since it has the setgid bit set, but group execute is not set. If you list the permissions of this file with *ls -l*, you get `rwxrwlr--` on a System V system, but many versions of *ls*, including BSD and GNU versions, will list `rwxrwSr--`.

Lock files

Lock files are the traditional method that *uucp* uses for locking serial lines. Serial lines are typically used either for dialing out, for example with *uucp*, or dialing in, which is handled by a program of the *getty* family. Some kind of synchronization is needed to ensure that both of these programs don't try to access the line at the same time. The other forms of locking we describe only apply to disk files, so we can't use them. Instead, *uucp* and *getty* create *lock files*. A typical lock file will have a name like */var/spool/uucp/LCK..ttyb*, and for some reason these double periods in the name have led to the term *dot locking*.

The locking algorithm is straightforward: if a process wants to access a serial line */dev/ttyb*, it looks for a file */var/spool/uucp/LCK..ttyb*. If it finds it, it checks the contents, which specify the process ID of the owner, and checks if the owner still exists. If it does, the file is locked, and the process can't access the serial line. If the file doesn't exist, or if the owner no longer exists, the process creates the file if necessary and puts its own process ID in the file.

Although the algorithm is straightforward, the naming conventions are anything but standardized. When porting software from other platforms, it is absolutely essential that all programs using dot locking should be agreed on the lock file name and its format. Let's look at the lock file names for the device */dev/ttyb*, which is major device number 29, minor device number 1. The *ls -l* listing looks like:

```
$ ls -l /dev/ttyb
crw-rw-rw-   1 root     sys       29,   1 Feb 25  1995 /dev/ttyb
```

Table 14-7 describes common conventions:

Table 14−7: uucp lock file names and formats

System	*Name*	PID format
4.3BSD	*/usr/spool/uucp/LCK..ttyb*	binary, 4 bytes
4.4BSD	*/var/spool/uucp/LCK..ttyb*	binary, 4 bytes
System V.3	*/usr/spool/uucp/LCK..ttyb*	ASCII, 10 bytes
System V.4	*/var/spool/uucp/LK.032.029.001*	ASCII, 10 bytes

A couple of points to note are:

- The digits in the lock file name for System V.4 are the major device number of the disk on which */dev* is located (32), the major device number of the serial device (29), and the minor device number of the serial device (1).
- Some systems, such as SCO, have multiple names for terminal lines, depending on the characteristics which it should exhibit. For example, */dev/tty1a* refers to a line when running without modem control signals, and */dev/tty1A* refers to the same line when running with modem control signals. Clearly only one of these lines can be used at the same time: by convention, the lock file name for both devices is */usr/spool/uucp/LCK..tty1a*.
- The locations of the lock files vary considerably. Apart from those in the table, other possibilities are */etc/locks/LCK..ttyb*, */usr/spool/locks/LCK..ttyb*, and */usr/spool/uucp/LCK/LCK..ttyb*.
- Still other methods exist. See the file *policy.h* in the Taylor *uucp* distribution for further discussion.

Lock files are unreliable. It is quite possible for two processes to go through this algorithm at the same time, both find that the lock file doesn't exist, both create it, and both put their process ID in it. The result is not what you want. Lock files should only be used when there is really no alternative.

locking system call

`locking` comes from the original implementation introduced during the Seventh Edition. It is still available in XENIX. It implements mandatory range locking.

```
int locking (int fd, int mode, long size);
```

`locking` locks a block of data of length `size` bytes, starting at the current position in the file.

mode can have one of the following values:

Table 14–8: locking operation codes

Parameter	Meaning
LK_LOCK	Obtain an exclusive lock for the specified block. If any part is not available, sleep until it becomes available.
LK_NBLCK	Obtain an exclusive lock for the specified block. If any part is not available, the request fails, and errno is set to EACCES.
LK_NBRLCK	Obtains a shared lock for the specified block. If any part is not available, the request fails, and errno is set to EACCES.
LK_RLCK	Obtain a shared lock for the specified block. If any part is not available, sleep until it becomes available.
LK_UNLCK	Unlock a previously locked block of data.

flock

flock is the weakest of all the lock functions. It provides only advisory file locking.

```
#include <sys/file.h>
(defined in sys/file.h)
#define   LOCK_SH   1          /* shared lock */
#define   LOCK_EX   2          /* exclusive lock */
#define   LOCK_NB   4          /* don't block when locking */
#define   LOCK_UN   8          /* unlock */

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

flock applies or removes a lock on fd. By default, if a lock cannot be granted, the process blocks until the lock is available. If you set the flag LOCK_NB, flock returns immediately with errno set to EWOULDBLOCK if the lock cannot be granted.

fcntl locking

On page 207 we discussed fcntl, a function that can perform various functions on open files. A number of these functions perform advisory record locking, and System V also offers the option of mandatory locking. All locking functions operate on a struct flock:

```
struct flock
  {
  short l_type;               /* lock type: read/write, etc. */
  short l_whence;             /* type of l_start */
  off_t l_start;              /* starting offset */
  off_t l_len;                /* len = 0 means until end of file */
  long  l_sysid;              /* Only SVR4 */
  pid_t l_pid;                /* lock owner */
```

```
};
```

In this structure,

- `l_type` specifies the type of the lock, listed in Table 14-9.

Table 14–9: `flock.l_type` values

value	Function
`F_RDLCK`	Acquire a *read* or *shared* lock.
`F_WRLCK`	Acquire a *write* or *exclusive* lock.
`F_UNLCK`	Clear the lock.

- The offset is specified in the same way as a file offset is specified to `lseek`: `flock->l_whence` may be set to `SEEK_SET` (offset is from the beginning of the file), `SEEK_CUR` (offset is relative to the current position) or `SEEK_EOF` (offset is relative to the current end of file position).

All `fcntl` lock operations use this struct, which is passed to `fcntl` as the `arg` parameter. For example, to perform the operation `F_`*`FOO`*`LK`, you would write:

```
struct flock flock;
error = fcntl (myfile, F_FOOLK, &flock);
```

The following `fcntl` operations relate to locking:

- `F_GETLK` gets information on any current lock on the file. when calling, you set the fields `flock->l_type`, `flock->l_whence`, `flock->l_start`, and `flock->l_len` to the value of a lock that we want to set. If a lock that would cause a lock request to block already exists, `flock` is overwritten with information about the lock. The field `flock->l_whence` is set to `SEEK_SET`, and `flock->l_start` is set to the offset in the file. `flock->l_pid` is set to the pid of the process that owns the lock. If the lock can be granted, `flock->l_type` is set to `F_UNLK` and the rest of the structure is left unchanged,
- `F_SETLK` tries to set a lock (`flock->l_type` set to `F_RDLCK` or `F_WRLCK`) or to reset a lock (`flock->l_type` set to `F_UNLCK`). If a lock cannot be obtained, `fcntl` returns with `errno` set to `EACCES` (System V) or `EAGAIN` (BSD and POSIX).
- `F_SETLKW` works like `F_SETLK`, except that if the lock cannot be obtained, the process blocks until it can be obtained.
- System V.4 has a further function, `F_FREESP`, which uses the `struct flock`, but in fact has nothing to do with file locking: it frees the space defined by `flock->l_whence`, `flock->l_start`, and `flock->l_len`. The data in this part of the file is physically removed, a read access returns `EOF`, and a write access writes new data. The only reason this operation uses the `struct flock` (and the reason we discuss it here) is because `struct flock` has suitable members to describe the area that needs to be freed. Many file systems allow data to be freed only if the end of the region corresponds with the end of file, in which case the call can be replaced with `ftruncate`.

lockf

`lockf` is a library function supplied only with System V. Like `fcntl`, it implements advisory or mandatory range locking based on the file permissions. In some systems, it is implemented in terms of `fcntl`. It supports only exclusive locks:

```
#include <unistd.h>

int lockf (int fd, int function, long size);
```

The functions are similar to those supplied by `fcntl`. `l_type` specifies the type of the lock, as shown in Table 14-10.

Table 14−10: `lockf` functions

value	Function
F_ULOCK	Unlock the range.
F_LOCK	Acquire exclusive lock.
F_TLOCK	Lock if possible, otherwise return status.
F_TEST	Check range for other locks.

`lockf` does not specify a start offset for the range to be locked. This is always the current position in the file—you need to use `lseek` to get there if you are not there already. The following code fragments are roughly equivalent:

```
flock->ltype = F_WRLK;          /* lockf only supports write locks */
flock->whence = SEEK_SET;
flock->l_start = filepos;     /* this was set elsewhere */
flock->l_len = reclen;          /* the length to set */
error = fcntl (myfile, F_GETLK, &flock);
```

...and

```
lseek (myfile, SEEK_SET, filepos); /* Seek the correct place in the file */
error = lockf (myfile, F_LOCK, reclen);
```

Which locking scheme?

As we've seen, file locking is a can of worms. Many portable software packages offer you a choice of locking mechanisms, and your system may supply a number of them. Which do you take? Here are some rules of thumb:

- *fcntl* locking is the best choice, as long as your system and the package agree on what it means. On System V.3 and V.4, `fcntl` locking offers the choice of mandatory or advisory locking, whereas on other systems it only offers advisory locking. If your package expects to be able to set mandatory locking, and you're running, say, 4.4BSD, the package may not work correctly. If this happens, you may have to choose `flock` locking instead.

- If your system doesn't have `fcntl` locking, you will almost certainly have either `flock` or `lockf` locking instead. If the package supports it, use it. Pure BSD systems don't support `lockf`, but some versions simulate it. Since `lockf` can also be used to require mandatory locking, it's better to use `flock` on BSD systems and `lockf` on System V systems.
- You'll probably not come across any packages which support `locking`. If you do, and your system supports it, it's not a bad choice.
- If all else fails, use lock files. This is a very poor option, though—it's probably a better idea to consider a more modern kernel.

Memory-mapped files

Some systems offer a feature called *memory mapped files*: the data of a file is mapped to a particular area of memory, so you can access it directly rather than by calling `read` and `write`. This increases performance, since the virtual memory system is more efficient than the file system. The following function calls are used to implement memory mapping:

- You need to open the file with the file system calls `open` or `creat`.
- `mmap` maps the file into memory.
- `msync` ensures that updates to the file map are flushed back to the file.
- `munmap` frees the mapped file data.

In the following sections, we'll look at these functions more closely.

mmap

`mmap` maps a portion of a file to memory.

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap (caddr_t addr, int len, int prot, int flags, int fd, off_t offset);
```

- `addr` specifies the address at which the file should be mapped. Unless you have good reasons to do otherwise, you should specify it as `NULL` and let `mmap` choose a suitable address itself. If `mmap` can't place the memory where it is requested, the subsequent behaviour depends on the flag `MAP_FIXED`—see the discussion of flags below.
- `len` specifies the length to map.
- `prot` specifies the accessibility of the resultant memory region, and may be any combination of `PROT_EXEC` (pages may be executed), `PROT_READ` (pages may be read) or `PROT_WRITE` (pages may be written). In addition, System V.4 allows the specification `PROT_NONE` (pages may not be accessed at all).

- `flags` is a bit map that specifies properties of the mapped region. It consists of a combination of the following bit-mapped flags:
  - `MAP_ANON` specifies that the memory is not associated with any specific file. In many ways, this is much the same thing as a call to `malloc`: you get an area of memory with nothing in it. This flag is available only in BSD.
  - `MAP_FILE` specifies that the region is mapped from a regular file or character-special device. This flag, supplied only in BSD, is really a dummy and is used to indicate the opposite of `MAP_ANON`: if you don't have it, ignore it.
  - `MAP_FIXED` specifies that `mmap` may use only the specified `addr` as the address of the region. The 4.4BSD man page discourages the use of this option.
  - `MAP_INHERIT` permits regions to be inherited across `exec` system calls. Only supported in 4.4BSD.
  - `MAP_PRIVATE` specifies that modifications to the region are private: if the region is modified, a copy of the modified pages is created and the modifications are copied to them. This flag is used in debuggers and to perform page-aligned memory allocations: `malloc` doesn't allow you to specify the address you want. In some systems, such as System V.4, `MAP_PRIVATE` is defined as 0, so this is the default behaviour. In others, such as SunOS 4, you must specify either `MAP_PRIVATE` or `MAP_SHARED`—otherwise the call fails with an `EINVAL` error code.
  - `MAP_SHARED` specifies that modifications to the region are shared: the virtual memory manager writes any modifications back to the file.
- On success, `mmap` returns the address of the area that has been mapped. On failure, it returns `-1` and sets `errno`.

msync

Writes to the memory mapped region are treated like any other virtual memory access: the page is marked dirty, and that's all that happens immediately. At some later time the memory manager writes the contents of memory to disk. If this file is shared with some other process, you may need to explicitly flush it to disk, depending on the underlying cooperation between the file system and the virtual memory manager.

System V.4 maps the pages at a low level, and the processes share the same physical page, so this problem does not arise. BSD and older versions of System V keep separate copies of memory mapped pages for each process that accesses them. This makes sharing them difficult. On these systems, the `msync` system call is used to flush memory areas to disk. This solution is not perfect: the possibility still exists that a concurrent read of the area may get a garbled copy of the data. To quote the 4.4BSD man pages:

> *Any required synchronization of memory caches also takes place at this time. Filesystem operations on a file that is mapped for shared modifications are unpredictable except after an msync.*

Still, it's better than nothing. The call is straightforward:

```
void msync (caddr_t addr, int len);
```

`addr` must be specified and must point to a memory mapped page; `len` may be 0, in which case all modified pages are flushed. If `len` is not 0, only modified pages in the area defined by `addr` and `len` are flushed.

munmap

`munmap` unmaps a memory mapped file region:

```
void munmap (caddr_t addr, int len);
```

It unmaps the memory region specified by `addr` and `len`. This is not necessary before terminating a program—the region is unmapped like any other on termination—and it carries the danger that modifications may be lost, since it doesn't flush the region before deallocating. About the only use is to free the area for some other operation.