Input/Output Interfaces 4

Files and I/O

Files that are organized as a sequence of data are called *regular* files. These can contain ASCII text, text in some other encoding binary data, executable code, or any combination of text, data, and code. The file has two components:

- The control data, called the *inode*. These data include the file type, the access permissions, the owner, the file size, and the location(s) of the data blocks.
- The file contents: an nonterminated sequence of bytes.

Solaris provides three basic forms of file input/output interfaces.

- One form is the traditional style of file I/O described in “Basic File I/O” on page 52.
- The second form is the “standard file I/O”. The buffering provided by standard I/O provides an easier interface and improved efficiency to an application that is run on a system without virtual memory. In an application running in a virtual memory environment, such as the Solaris 2.x system, standard file I/O is a very inefficient anachronism.
- The third form of file I/O is provided by the memory mapping interface described in “Memory Management Interfaces” on page 94. Mapping files is the most efficient and powerful form of file I/O for most applications run in the Solaris 2.x environment.

Note that it is not necessary to use traditional file I/O to obtain locking of file elements. The lighter weight synchronization mechanisms described in *Multithreaded Programming Guide* can be used more effectively with mapped files.

Basic File I/O

These functions perform basic operations on files:

Table 4-1 Basic File I/O Functions

Function Name	Purpose
`open`	Open a file for reading or writing
`close`	Close a file descriptor
`read`	Read from a file
`write`	Write to a file
`creat`	Create a new file or rewrite an existing one
`unlink`	Remove a directory entry
`lseek`	Move read/write file pointer

Code Example 4-1, below, demonstrates the use of the basic file I/O interface. `read`(2) and `write`(2) both transfer no more than the specified number of bytes, starting at the current offset into the file. The number of bytes actually transferred is returned. The end of a file is indicated, on a `read`(), by a return value of zero.

Code Example 4-1 Basic file I/O

```
#include    <fcntl.h>
#define     MAXSIZE     256

main()
{
    int     fd, n;
    char    array[MAXSIZR]

    fd = open ("/etc/motd", O_RDONLY);
    if (fd == -1) {
        perror ("open");
        exit (1);
    }
```

```
    while ((n = read (fd, array, MAXSIZE)) > 0)
        if (write (1, array, n) != n)
            perror ("write");
    if (n == -1)
        perror ("read);
    close (fd);
}
```

Always close a file when you are done reading or writing it.

Offset into an open file are changed by `read`()s, `write`()s, or by calls to `lseek`(2). Some examples of using lseek() are:

Code Example 4-2 Seek code

```
off_t   start, n;
struct  record  rec;

/* record current offset in start */
start = lseek (fd, 0L, SEEK_CUR);

/*go back to start */
n = lseek (fd, start, SEEK_SET);
read (fd, (char *)&rec, sizeof (rec));

/* rewrite previous record */
n = lseek (fd, -sizeof (rec), SEEK_CUR);
write (fd, (char *)&rec, sizeof (rec));
```

Advanced File I/O

These functions create and remove directories and files, create links to existing files, and obtain or modify file status information:

Table 4-2 Advanced File I/O Functions

Function Name	Purpose
`link`	Link to a file
`access`	Determine accessibility of a file
`mknod`	Make a special or ordinary file
`chmod`	Change mode of file

`chown`
`lchown`
`fchown`
	Change owner and group of a file

Table 4-2 Advanced File I/O Functions (Continued)

Function Name	Purpose
`utime`	Set file access and modification times

`stat`
`lstat`
`fstat`
	Get file status

`fcntl`	Perform file control functions
`ioctl`	Control device
`fpathconf` `pathconf`	Get configurable path name variables

`opendir`
`readdir`
`closedir`
	Perform directory operations

`mkdir`	Make a directory
`readlink`	Read the value of a symbolic link
`rename`	Change the name of a file
`rmdir`	Remove a directory
`symlink`	Make a symbolic link to a file

File System Control

These functions allow you to control various aspects of the file system:

Table 4-3 File System Control Functions

Function Name	Purpose
`ustat`	Get file system statistics
`sync`	Update super block
`mount`	Mount a file system
`unmount`	Unmount a file system
`statvfs` `fstatvfs`	Get file system information
`sysfs`	Get file system type information

4

File and Record Locking

You lock files, or portions of files, to prevent the errors that can occur when two or more users of a file try to update information at the same time.

File locking and record locking are really the same thing, except that file locking implies that the whole file is affected, while record locking means that only a specified portion of the file is locked. (In the SunOS 5.x system, file structure is undefined: a record is a concept of the programs that use the file.)

Supported File Systems

Both advisory and mandatory locking are supported on the following types of file systems:

- `ufs`—the default disk-based file system
- `fifofs`—a pseudo file system of named pipe files that give processes common access to data.
- `namefs`—a pseudo file system used mostly by STREAMS for dynamic mounts of file descriptors on top of files.
- `specfs`—a pseudo file system that provides access to special character and block devices.

only advisory file locking is supported on NFS.

File locking is not supported for the `proc` and `fd` file systems.

Choosing A Lock Type

Mandatory locking suspends a process until the requested file segments are free. Advisory locking returns a result indicating whether the lock was obtained or not. Processes can ignore the result and do the I/O anyway. You cannot have both mandatory and advisory file locking on the same file at the same time. The mode of the file at the time it is opened determines whether the existing locks on the file are treated as mandatory or advisory.

Of the two basic locking calls, `fcntl`(2) is more portable, more powerful, and less easy to use than `lockf`(3C). `fcntl`() is specified in Posix 1003.1 standard. `lockf`() is provided to be compatible with older applications.

4

Terminology

Some important definitions:

record	An arbitrary sequence of bytes in a file. The UNIX operating system supports no record structure. Programs that use the files can impose any desired record structure.
cooperating processes	Processes use some technique to access a shared resource, often a synchronization mechanism.
read lock	Used to control access to segments of files. A read lock lets other processes also lock the same segment to read, but lets no other process have a write lock on an overlapping segment of the file.
write lock	Used to control access to segments of files. No other process can read or write the record.
advisory lock	The form of record lock that does not suspend execution of other processes. Advisory locking is not enforced on `creat`(2), `open`(2), `read`(2), or `write`(2) operations. Advisory locking lets calling processes poll the state of a lock and do other work if a lock is not available.
mandatory lock	A record lock that suspends execution of processes that do not hold the lock. Access to locked records is enforced on `creat`(2), `open`(2), `read`(2), and `write`(2) operations.

Setting a File Lock

There are several ways to set a lock on a file. Choice of method depends on how the lock interacts with the rest of the program, performance, and portability. To lock an entire file, set the offset to zero, and, by convention, set the size to zero.

The first example uses the POSIX standard-compatible `fcntl`(2) function. It tries to lock a file until one of the following happens:

- The file is successfully locked
- There is an error

- `MAX_TRY` is exceeded, and the program gives up trying to lock the file

```
#include <fcntl.h>

...
    struct flock lck;

...
    lck.l_type = F_WRLCK;/* setting a write lock */
    lck.l_whence = 0;/* offset l_start from beginning of file */
    lck.l_start = (off_t)0;
    lck.l_len = (off_t)0;/* until the end of the file */
    if (fcntl(fd, F_SETLK, &lck) <0) {
        if (errno == EAGAIN || errno == EACCES) {
            (void) fprintf(stderr, "File busy try again later!\n");
            return;
        }
        perror("fcntl");
        exit (2);
    }
    ...
```

The second example uses `lockf`(3C):

```
#include <unistd.h>

...

    /* make sure the file pointer is at the beginning of the file. */
    lseek(fd, (off_t)0, 0);
    if (lockf(fd, F_TLOCK, 0L) < 0) {
        if (errno == EAGAIN || errno == EACCES) {
            (void) fprintf(stderr,"File busy try again later!\n");
            return;
        }
        perror("lockf");
        exit(2);
    }
...
```

Note that the `lockf`(3C) example is simpler, but the `fcntl`(2) example shows more flexibility. Using `fcntl`(2), you can set the type and start of the lock request by setting a few structure variables. The `lockf` method sets only write (exclusive) locks; an additional function, `lseek`, is required to specify the start of the lock.

Opening a File for Record Locking

A lock can only be requested on a file with a valid open descriptor. For read locks, the file must be opened with at least read access. For write locks, the file must also be opened with write access. In the example, a file is opened for both read and write access:

```
...
    filename = argv[1];
    fd = open (filename, O_RDWR);
    if(fd < 0) {
        perror(filename);
        exit(2);
    }
    ...
```

Note – Mapped files cannot be locked with `flock`(). See `mmap`(2).

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except that the starting point and length of the lock segment are not set to zero.

Plan a failure strategy for when you cannot obtain all the required locks. Contention for records is why you use record locking, so different programs might:

- Wait a certain amount of time, then try again
- Abort the procedure and warn the user
- Let the process sleep until signaled that the lock has been freed
- Do some combination of the above

This example shows locking a record using `fcntl`():

```
{
    struct flock lck;

...
    lck.l_type = F_WRLCK;/* setting a write lock */
    lck.l_whence = 0;/* offset l_start from beginning of file */
    lck.l_start = here;
    lck.l_len = sizeof(struct record);

    /* lock "this" with write lock */
```

```
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* "this" lock failed. */
        return (-1);
...
}
```

The next example shows the `lockf` function:

```
#include <unistd.h>

...
    /* lock "this" */
    (void) lseek(fd, this, SEEK_SET);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Lock on "this" failed. Clear lock on "here". */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
```

Locks are removed the same way they are set—only the lock type is different (F_ULOCK). An unlock cannot be blocked by another process and affects only locks placed by the process. The unlock affects only the segment of the file specified in the preceding locking call.

Getting Lock Information

You can determine which process, if any, is holding a lock. Use this as a simple test or to find locks on a file. A lock is set, as in the previous examples, and `F_GETLK` is used in the `fcntl` call. The next example finds and prints indentifying data on all the locked segments of a file:

```
struct flock lck;

    lck.l_whence = 0;
    lck.l_start = 0L;
    lck.l_len = 0L;
    do {
        lck.l_type = F_WRLCK;
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK) {
            (void) printf("%d %d %c %8d %8d\n", lck.l_sysid,
                    lck.l_pid, (lck.l_type == F_WRLCK) ? 'W' : 'R',
                    lck.l_start, lck.l_len);
```

```
            /* If this lock goes to the end of the address space, no
             * need to look further, so break out. */
            if (lck.l_len == 0) {
            /* else, look for new lock after the one just found. */
                    lck.l_start += lck.l_len;
            }
        }
    } while (lck.l_type != F_UNLCK);
```

The `fcntl` function with the `F_GETLK` command can sleep while waiting for a server to respond, and it can fail (returning ENOLCK) if there is a resource shortage on either the client or server.

The `lockf` function with the `F_TEST` command can also be used to test if a process is blocking a lock. This function does not return information about where the lock is and which process owns the lock.

```
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero (0). to test until the
   end of the file address space. */
if (lockf(fd, (off_t)0, SEEK_SET) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unexpected error <%d>\n", errno);
            break;
    }
```

Forking Locks

When a process forks, the child receives a copy of the file descriptors that the parent opened. Locks are not inherited by the child because they are owned by a specific process. The parent and child share a common file pointer for each file. Both processes can try to set locks on the same location in the same file.

This problem happens with both `lockf`(3C) and fcntl(2). If a program holding a record lock forks, the child process should close the file and reopen it to set a new, separate file pointer.

Deadlock Handling

The UNIX locking facilities provide deadlock detection/avoidance. Deadlocks can happen only when the system is about to put a record locking function to sleep. A search is made to determine whether process A will wait for a lock that B holds while B is waiting for a lock that A holds. If a potential deadlock is detected, the locking function fails and sets `errno` to indicate deadlock. Processes setting locks using `F_SETLK` do not cause a deadlock because they do not wait when the lock cannot be granted immediately.

Selecting Advisory or Mandatory Locking

For mandatory locks, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Get mandatory enforcement as follows:

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

int mode;
struct stat buf;
    ...
    if (stat(filename, &buf) < 0) {
        perror("program");
        exit (2);
    }
    /* get currently set mode */
    mode = buf.st_mode;
    /* remove group execute permission from mode */
    mode &= ~(S_IEXEC>>3);
        /* set 'set group id bit' in mode */
    mode |= S_ISGID;
    if (chmod(filename, mode) < 0) {
        perror("program");
        exit(2);
    }
    ...
```

Files to be record locked should never have any execute permission set. This is because the operating system ignores record locks when executing a file.

The `chmod`(1) command can also be used to set a file to permit mandatory locking. For example:

```
$ chmod +l file
```

(Note that this is letter "l" and not the number "1".) This command sets two permission bits in the file mode, which indicates mandatory locking on the file. The two bits in the mode are .1./.../..0/... An individual file cannot simultaneously be enabled for mandatory locking and have the set-group-ID on execution bit set. Nor can an individual file be enabled for mandatory locking and for group execution.

The `ls`(1) command shows this setting when you ask for the long listing format with the `-l` option:

```
$ ls -l file
```

displays following information:

```
-rw---l--- 1 user group size mod_time file
```

The letter "l" in the permissions indicates that the set-group-ID bit is on, so mandatory locking is enabled, as well as the normal semantics of set group ID.

Cautions about Mandatory Locking

- Mandatory locking works only for local files. It is not supported when accessing files through NFS.
- Mandatory locking protects only the segments of a file that are locked. The remainder of the file can be accessed according to normal file permissions.
- If multiple reads or writes are needed for an atomic transaction, the process should explicitly lock all such segments before any I/O begins. Advisory locks are sufficient for all programs that perform in this way.
- Arbitrary programs should not have unrestricted access permission to files on which record locks are used.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Terminal I/O

These functions deal with a general terminal interface for controlling asynchronous communications ports:

Table 4-4 Terminal I/O Functions

Function Name	Purpose
`tcgetattr` `tcsetattr`	Get and set terminal attributes

`tcsendbreak`
`tcdrain`
`tcflush`
`tcflow`
	Perform line control functions

`cfgetospeed`
`cfgetispeed`
`cfsetispeed`
`cfsetospeed`
	Get and set baud rate

`tcgetpgrp` `tcsetpgrp`	Get and set terminal foreground process group ID
`tcgetsid`	Get terminal session ID

4