With the exception that named pipes and mount points are not supported, an application program need not be aware that it is doing input or output to files in a shared file system.

The same system calls that are used for I/O in standard NC1FS file systems are used for files in a shared file system. Files are created with the open(2) system call, updated using read(2) and write(2), reada(2) and writea(2), or listio(2), and closed or unlinked with close(2) and unlink(2) when they are no longer needed. The fact that a file is part of an SFS configuration can be completely transparent at the user level.

It is possible, however, to take advantage of extensions to several system calls to control allocation of and access to files in an SFS configuration more closely, as well as to enhance input and output performance.

If an application is written that uses features that are exclusive to shared file systems, it should use the sysconf library routine to determine whether or not the UNICOS SFS feature is currently available on the host system. If it is, the sysconf call returns the SMP port number to which the host is connected. If not, it returns a negative number. The following code fragment demonstrates the use of sysconf.

```
if ( sysconf ( _SC_CRAY_SFS ) < 0 ) {
        printf ( "SFS not active on this machine\n" );
        exit;
} else {     /* continue processing */
        .
        .
        .
}
```

In the sections that follow, extensions to system calls that are exclusive to SFS file systems are specified as such. In addition, some features that are available to both regular NC1FS and SFS file systems, but which may profitably be used in conjunction with exclusive features, are described.

8.1 Data locking and SFS systems

A fundamental requirement of shared file systems is the preservation of data integrity. Considerable effort, and therefore system time, is expended to insure that a shared file is protected from inadvertent simultaneous events. For example, if two

processes running on different Cray Research systems were simultaneously extending the same file, it is important that all parties involved maintain a common view of where the end of the file is located.

In the UNICOS SFS environment, any time a user process does anything to a file that changes any of the attributes of that file, such as size, modification date, ownership, and so on, the UNICOS SFS feature must update the on-media inode, under protection of an inode semaphore.

The notable exception to the above statement happens when the UNICOS SFS feature determines that the file update is occurring under control of one of the three supported data locks:

- Exclusive open lock
- Read lock
- Write lock

If a shared file is currently *open-locked*, *read-locked*, or *write-locked*, the UNICOS SFS feature defers inode updates, and therefore inode semaphore operations, until the lock is relinquished. In this *data-locked environment*, the UNICOS SFS feature is able to deliver device speed I/O performance.

8.1.1 Exclusive open lock

A UNICOS SFS exclusive open lock is granted if the user process used the `O_SFSXOP` flag on the `open`(2) request. Upon return from the `open`(2), the user process is guaranteed that no other process in the SFS cluster has this file open. While that process owns the open lock, the process may execute an `unlink`(2) call on the file, thus causing all other pending `open`(2) calls for this file to fail with an `ENOENT` error. The UNICOS SFS exclusive open lock can only be obtained through the `open`(2) call.

If a process tries to open a file without the `O_SFSXOP` flag, when the file is already open by another process with an exclusive open lock, the resulting behavior is determined by the presence or absence of either the `O_NDELAY` or the `O_NONBLOCK` flags. If either is set on the attempted open, the open will fail with an `EAGAIN` error. If neither flag is set, the process will sleep until the file has been closed by all other processes on all Cray Research systems.

If the same process opens a file with the exclusive open flag, and then attempts a subsequent non-exclusive open, the second open attempt will fail with an `EDEADLK` error.

Use of the `O_SFSXOP` flag makes opening and locking a file an atomic operation, which closes the timing window between what would otherwise require an `open` call followed by an `fcntl` system call to lock the file.

8.1.2 Read lock

A UNICOS SFS read lock is granted if no other process in the SFS cluster has a current open lock, or write lock. A read lock is requested by using the normal `flock`(2) or `fcntl`(2) UNICOS kernel interfaces. As many as 500 processes on each system in the SFS cluster may concurrently own a read lock on a shared file.

8.1.3 Write lock

A UNICOS SFS write lock is granted if no other process in the SFS cluster has a current open lock, read lock, or write lock. A write lock is requested using the normal `flock`(2) or `fcntl`(2) UNICOS kernel interface. Only one process in the entire SFS cluster may own a write lock on a shared file.

8.1.4 Setting write locks and read locks

An write lock or a read lock may be assigned to or released from an open file by calling the `fcntl`(2) UNICOS kernel interface with a pointer to an appropriately populated `struct flock` structure (shown below), which is defined in `<sys/fcntl.h>`. Note that because file data locking in a shared file system applies to the entire file, fields `l_whence, l_start`, and `l_len` are forced to zero by the data locking functions of the `fcntl`(2) system call.

```
struct flock {
       short l_type;
       short l_whence;
       off_t l_start;
       off_t l_len;
       short l_sysid;
       pid_t l_pid;
};
```

Both a blocking and a non-blocking version of file locking commands in `fcntl` exist. If `F_SETLK` is used and the lock cannot be set, the `fcntl` call returns immediately with a -1 value. If `F_SETLKW` is used and the read or write lock cannot be set, the process will sleep until it can be set.

8.1.5 Adding a write lock

If a process needs to update a file and wants to disallow both reading and writing by all other processes, it may set a write lock by using the `fcntl`(2) system call. While

one process has a write lock set on an open file, no other process may set it, nor may a read lock be set, and, of course, no exclusive open operations of the file are allowed.

If a call is made to `fcntl` to set a write lock on a file that was opened with the `O_SFSXOP` flag, the exclusive open lock will be retained for the file. That is, it will have both an exclusive open and a write lock.

The following example demonstrates how to add a write lock.

```
#include <sys/fcntl.h>
struct flock flock, *fp = &flock;

main {
     int fd, rc;

     if ( ( fd = open ( "shared_file", O_CREAT | O_RDWR ) ) == -1 ) {
            exit ( 1 );
     }

     /*
      * Set the write lock. Note that another process
      * could obtain a write lock in this window
      */
      fp->l_type = F_WRLCK;
      if( ( rc = fcntl ( fd, F_SETLKW, fp ) ) == -1 ) {
             exit ( 1 );
      }
}
```

8.1.6 Adding a read lock

A file may have a read lock set. This lock is obtained by means of the `fcntl`(2) system call. The read lock is unique in that more than one process may hold the lock at the same time. Currently, 500 processes may hold read locks on the same file. If a file has one or more read locks, it will block write operations, exclusive open operations and the setting of a write lock until all processes holding the read lock release it.

If an `fcntl` call is made to set a read lock on a file that was opened with the `O_SFSXOP` flag, the exclusive open lock for the file will be released for the file. That is, a file cannot have both an exclusive open and a read lock set.

The following example shows how to set a read lock for a file.

```
#include <sys/fcntl.h>
struct flock flock, *fp = &flock;
```

```
main {
     int fd, rc;

     if ( ( fd = open ( "shared_file", O_CREAT | O_READ ) ) == -1 ) {
            exit ( 1 );
     }

     /*
      * Set the read lock. Note that another process
      * could obtain an write lock in this window
      */
     fp->l_type = F_RDLCK;
     if( ( rc = fcntl ( fd, F_SETLKW, fp ) ) == -1 ) {
            exit ( 1 );
     }
}
```

8.1.7 Removing data locks

A process may explicitly relinquish a data lock by using the `flock`(2) or `fcntl`(2) UNICOS kernel interface, or implicitly by simply closing the file. For open and write locks, where only one process in the SFS cluster can own the lock, the lock is relinquished immediately. Because read locks can be held by many processes, a reference counter mechanism is used. In this case, the lock is relinquished when all processes that own a read lock have relinquished their read locks.

When the `l_type` field of the `flock` structure is set to `F_UNLCK`, a lock currently associated with the file's data are removed, including the exclusive open lock. If this process is the last (or only) process holding a read lock on the file, other processes will be allowed to acquire a write lock. Otherwise, the file is still considered to be locked against such attempts due to other holders of the read lock. The following code example shows how to clear a file data lock.

```
#include <sys/fcntl.h>
struct flock flock, *fp = &flock;

main {
     int fd, rc;

     if ( ( fd = open ( "shared_file", O_CREAT | O_READ ) ) == -1 ) {
            exit ( 1 );
     }
```

```
        /*
         * Set the read lock.
         */
        fp->l_type = F_RDLCK;
        if( ( rc = fcntl ( fd, F_SETLKW, fp ) ) == -1 ) {
               exit ( 1 );
        }

        /*
         * Clear all file data locks held by this process
         */
        fp->l_type = F_UNLCK;
        if ( ( rc = fcntl ( fd, F_SETLK, fp ) ) == -1 ) {
                exit ( 1 );
        }
}
```

8.2 File allocation issues

NC1FS file systems, as well as SFS file systems, may be configured to have a *primary allocation area* and a *secondary allocation area.* In general, each of these two allocation areas will have distinctive characteristics. For example, the primary area might be mirrored to provide redundancy for file system structures such as inodes, while the secondary area could be striped to provide high bandwidth I/O. In cases such as this, it becomes desirable to be able to direct certain types or sizes of files to one or the other allocation area.

Note: Mirroring is not supported on CRAY T3D or CRAY T3E systems.

In the absence of explicit direction from the process that opened the file, where a file's data is initially allocated is a function of the allocation unit sizes of the areas and the quantity of data written to the file at a time. This assumes that neither allocation area is full at the time the file is opened.

Another aspect of file allocation that might be important is whether or not the file's data should be allowed to reside partially in both allocation areas. Again, in the absence of explicit direction from the user, a file that is initially allocated in the primary allocation area will have subsequent allocations in the secondary area if the file grows beyond a system defined size (BIGFILE, <sys/param.h>), or if the primary allocation area becomes full.

Yet another consideration is whether a file should be allowed to increase in size once it has reached a user-specified size. For example, if a process uses the ialloc

routine to preallocate space for a file, the user may wish that space to be the maximum size that the file can become. By default, simply preallocating a file does not keep it from growing beyond the preallocated size.

All of these issues (initial placement of data, residence in both allocation areas, and limiting file size) can be controlled by options on various system calls, which are discussed in the following sections.

8.2.1 The open(2) system call

The O_BIG flag may be used at file open time to control allocation of a file in file system configurations with both a primary and secondary allocation area.

By including the O_BIG flag in the open system call, an application program forces initial allocation of the file's data to the secondary allocation area.

8.2.2 The ialloc(2) system call

The ialloc system call allows a user to preallocate data space for a previously opened file.

The form of the ialloc system call is:

```
long ialloc ( int fildes, long nb, int flag, int part, long *avl );
```

Note: Although the user requests space allocation in *bytes*, the kernel actually allocates in multiples of *allocation units*, so it is possible that the total amount of space available for a file is somewhat larger than what the user requested. This is discussed in the fcntl section.

Several flags may be used to precisely control the characteristics of the preallocated file space.

- The IA_CONT flag specifies that the space is to be allocated contiguously in the file system. If the amount of space requested is unavailable, no space is allocated, and *ialloc* returns a -1, unless the IA_BEST flag also is set.
- If the IA_BEST flag is set, then either the amount of space requested or the amount of space available is allocated, whichever is smaller, and the amount allocated is returned in the system call return value.
- If the IA_RAVL flag is set, then either the requested amount is allocated, or nothing is allocated and an error is returned, and the amount of space available is returned in the *avl* parameter.

- The IA_PART flag restricts allocation to the current allocation area. If the O_BIG flag was set when the file was opened, the initial allocation area is the secondary allocation area.

All of the flags can be used in combination. For example, to allocate the largest contiguous chunk of space in the allocation area, the IA_CONT, IA_BEST, and IA_PART flags should all be set in the ialloc call.

8.2.3 The fcntl(2) system call

It may be desirable to restrict a file to a maximum size. This can be accomplished by setting the nogrow flag for a file.

The nogrow flag can be set for a file with a call similar to the following:

```
prev_flags = fcntl ( fd, F_SETALF0, S_ALF_NOGROW ) ;
```

The return value is the state of the flags prior to the call.

This flag is intended for situations where a file is preallocated with ialloc, and its size is not allowed to either increase or decrease. Note that calls to ialloc supply the number of *bytes* to be allocated for the file, but file space is actually allocated in multiples of *allocation units*. Thus, the actual size of the file might be somewhat larger than was specified in the ialloc call. This notwithstanding, if a file has the nogrow flag set, additional space will not be allocated for it beyond what it has at the time the fcntl call is made.

Even if a file has the nogrow flag set, the trunc system call may still be used to decrease the logical size of the file, but the physical space allocated to it is not released.

If, at some point during processing, it becomes desirable to allow file size changes to a file that has the nogrow flag set, it may be cleared with a call similar to the following:

```
prev_flags = fcntl ( fd, F_CLRALF, S_ALF_NOGROW ) ;
```

8.2.4 The join(2) and fjoin(2) system calls

The join and fjoin system calls are supported for both NC1FS and SFS file systems. There forms are:

```
int fjoin ( int fildes1, int fildes2 );
```

These calls are used to concatenate two files without moving file data. Instead, the extent descriptors in *file2*'s inode are appended to *file1*'s extent descriptors. If any

allocated but unused blocks exist at the end of *file1*, they are deallocated before *file2*'s extents are appended. At the completion of the system call, *file2* has been truncated to zero blocks.

Note: With the `join` system call, the file being joined to (first argument) must end on an allocation unit boundary. For example, if the allocation unit was one megabyte, then the first file would have to be an integral number of megabytes in size.

8.2.5 The open(2) system call

In addition to the exclusive open data lock described above, the `open`(2) system call supports the `O_SFS_DEFER_TM` flag, which may be useful when writing applications that use shared files.

The `O_SFS_DEFER_TM` flag may be useful in reducing file system overhead by declaring to the UNICOS kernel that updates to file inode time stamps may be done less frequently than they otherwise would. If it is not critical to the application that the time stamps returned by the `stat`(2) or `fstat`(2) system calls be highly accurate, then setting this flag on very active files is advisable. The number of inode updates to media will be reduced, which implies a reduction in overhead and a corresponding increase in performance.

The `O_SFS_DEFER_TM` flag is only valid for files using the UNICOS SFS feature.