Chapter 8

Optimizing Disk I/O for a Real-Time Program

A real-time program sometimes needs to perform disk I/O under tight time constraints and without affecting the timing of other activities such as data collection. This chapter covers techniques that IRIX supports that can help you meet these performance goals, including these topics:

- "Memory-Mapped I/O" on page 133 points out the uses of mapping a file into memory.
- "Asynchronous I/O" on page 134 describes the use of the asynchronous I/O feature of IRIX version 5.3 and later.
- "Synchronous Writing and Direct Writing" on page 147 documents the performance cost of knowing when disk output is complete.
- "Guaranteed-Rate I/O" on page 150 describes the use of the guaranteed-rate feature of XFS.

Memory-Mapped I/O

When an input file has a fixed size, the simplest as well as the fastest access method is to map the file into memory (for details on mapping files and other objects into memory, see the book *Topics in IRIX Programming*). A file that represents a data base of some kind—for example a file of scenery elements, or a file containing a precalculated table of operating parameters for simulated hardware—is best mapped into memory and accessed as a memory array. A mapped file of reasonable size can be locked into memory so that access to it is always fast (see "Locking Mapped Files Into Memory" on page 51).

You can also perform output on a memory-mapped file simply by storing into the memory image. When the mapped segment is also locked in memory, you control when the actual write takes place. Output happens only when the program calls **msync()** or changes the mapping of the file. At that time the modified pages are written. (See the msync(2) reference page.) The time-consuming call to **msync()** can be made from an asynchronous process.

Asynchronous I/O

You can use asynchronous I/O to isolate the real-time processes in your program from the unpredictable delays caused by I/O.

Conventional Synchronous I/O

Conventional I/O in UNIX is synchronous; that is, the process that requests the I/O is blocked until the I/O has completed. The effects are different for input and for output.

Synchronous Input

The normal sequence of operations for IRIX input is as follows:

1. Normal code in a process invokes the system call **read()**, either directly or indirectly—for example, by accessing a new page of a memory-mapped file, or by calling a library function that calls **read()**.
2. The kernel, still operating under the identity of the calling process, enters the read entry point of the device driver.
3. The device driver initiates the input operation and blocks the calling process, for example by waiting on a semaphore in the kernel address space.
4. The kernel schedules another process to use the CPU.
5. Later, the device completes the input operation and causes a hardware interrupt.
6. The kernel interrupt handler enters the device driver interrupt entry point.
7. The device driver, finding that the data has been received, unblocks the sleeping process, for example by posting a semaphore.
8. The kernel recalculates the scheduling queues to account for the fact that a blocked process can now run.
9. Then or perhaps later, depending on scheduling priorities, the kernel schedules the original process to run on some CPU.
10. The unblocked process exits the device driver read function and returns to user code, the read being complete.

During steps 4-8, the process that requested input is blocked. The duration of the delay is unpredictable. For example, the delay can be negligible if the data is already in a buffer in memory. It can be as long as one rotation time of a disk, if the disk is positioned on the correct cylinder. It can be longer still, if the disk has to seek. The probability of seeking depends on the way the file is arranged on the disk surface and also on the I/O operations of other processes in the system.

Synchronous Output

For disk files, the process that calls **write()** is normally delayed only as long as it takes to copy the output data to a buffer in kernel address space. The device driver schedules the device write and returns. The actual disk output is asynchronous. As a result, most output requests are blocked for only a short time. However, since a number of disk writes could be pending, the true state of a file on disk is unknown until the file is closed.

In order to make sure that all data has been written to disk successfully, a process can call **fsync()** for a conventional file or **msync()** for a memory-mapped file (see the fsync(2) and msync(2) reference pages). The process that calls these functions is blocked until all buffered data has been written. (An alternative for disk output is to use direct output, discussed under “Synchronous Writing and Direct Writing” on page 147.)

Devices other than disks may block the calling process until the output is complete. It is the device driver logic that determines whether a call to **write()** blocks the caller, and for how long. Device drivers for VME devices are often supplied by third parties.

Asynchronous I/O Basics

A real-time process needs to read or write a device, but it cannot tolerate an unpredictable delay. One obvious solution can be summarized as “call **read()** or **write()** from a different process, and run that process in a different CPU.” This is the essence of asynchronous I/O. You could implement an asynchronous I/O scheme of your own design, and you may wish to do so in order to integrate the I/O closely with your own design of processes and data structures. However, a standard solution is available.

Two Implementation Versions

IRIX (since version 5.3) supports asynchronous I/O library calls conforming to POSIX document 1003.1b-1993. You use relatively simple calls to initiate input or output. The library package handles the details of

- initiating several lightweight processes to perform I/O
- allocating a shared memory arena and the locks, semaphores, and/or queues used to coordinate between the I/O processes
- queueing multiple input or output requests to each of multiple file descriptors
- reporting results back to your processes, either on request, through signals, or through callback functions

Note: In IRIX 5.2 and IRIX 6.0, asynchronous I/O was implemented to conform to POSIX standard 1003.4 Draft 12, an earlier document. Support for the later POSIX standard1003.1b was implemented in IRIX 5.3. Programs compiled in IRIX 5.2 continue to work in later releases. However, conversion of programs to POSIX standard1003.1b is recommended. The remainder of this topic describes that interface.

Only in IRIX 5.3, both the Draft-12 and 1003.1b interfaces were supported. In that release, you had to take two steps to compile with the later version of asynchronous I/O:

- Define the compiler variable _ABI_SOURCE. This indicates conformance with the MIPS Application Binary Interface.
- Include the library parameter *-labi* in the link.

In releases following 5.3, support for POSIX 1003.1b-1993 is the only version of asynchronous I/O. These steps are no longer required. It is no longer possible to compile programs that use the Draft-12 interface.

Asynchronous I/O Functions

Once you have opened the files and initialized asynchronous I/O, you perform asynchronous I/O by calling some of these functions:

aio_read(3)	Initiates asynchronous input from a file or device.
aio_write(3)	Initiates asynchronous output to a file or device.
lio_listio(3)	Initiates a list of operations to one or more files or devices.
aio_error(3)	Returns the status of an asynchronous operation.
aio_fsync(3)	Waits for all scheduled output for a file to complete.
aio_cancel(3)	Cancels pending, scheduled operations.

Each of these functions is described in detail in a reference page in volume 3.

Asynchronous I/O Control Block

Each asynchronous I/O request is represented by an instance of *struct aiocb*, a data structure that your program must allocate. The important fields are as follows.

- The file descriptor that is the target of the operation.

  File descriptors are returned by **open()** (see the open(2) reference page). A file descriptor used for asynchronous I/O can represent any file or device—not only a disk file.

- The address and size of a buffer to supply or receive the data.
- The file position for the operation as it would be passed to **lseek()** (see the lseek(2) reference page)

  The use of this value is discussed under “Multiple Operations to One File” on page 146.

- A *sigevent* structure, whose contents indicate what, if anything, should be done to notify your program of the completion of the I/O.

  The use of the *sigevent* is discussed under “Checking for Completion” on page 142.

Note: The IRIX 5.2 implementation also accepted a request priority value. Request priorities are no longer supported. The field exists for compatibility and for possible future use, but must currently contain zero.

Initializing Asynchronous I/O

You can initialize asynchronous I/O in either of two ways. One way is simple; the other gives you control over the initialization.

Implicit Initialization

You can initialize asynchronous I/O simply by starting an operation with **aio_read()**, **lio_listio()**, or **aio_write()**. The first such call causes default initialization. This is the only form of initialization described by the POSIX standard. However, in a real-time program you often need to control at least the timing of initialization.

Initializing with aio_sgi_init()

You can take greater control of asynchronous I/O by calling **aio_sgi_init()** (refer to the aio_sgi_init(3) reference page and to the declarations in */usr/include/aio.h*). The argument to this call can be a null pointer, indicating you want default values, or you can pass an *aioinit_t* structure. The principal fields of this structure specify

- the number of asynchronous processes to execute I/O (*aio_threads*)

  The default is 5 processes; the minimum is 2. Specify 1 more than the number of I/O operations that could reasonably be executed in parallel on the available hardware. For example if you will be doing asynchronous I/O to one disk file and one tape drive, there could be at most two concurrent I/O operations, so there is no need to have more than 3 (1 more than 2) asynchronous processes.

- the number of locks that the asynchronous I/O processes should preallocate (*aio_locks*)

  The default used by **aio_init()** is 3 locks; the minimum is 1. Specify the maximum number of simultaneous **lio_listio**(LIO_NOWAIT), **aio_fsync()**, and **aio_suspend()** calls that your program could execute concurrently. If in doubt, specify the number of subprocesses your program contains.

- the number of lightweight processes (sprocs) that will be sharing the use of asynchronous I/O (*aio_numusers*)

  The default is 5; the minimum is 2. Specify 1 more than the number of different sproc'd processes that will be requesting asynchronous I/O.

Other fields of the *aioinit_t* structure such as *aio_num* and *aio_usedba* are not used at this time and must be zero. Zero-valued fields are taken as a request for the default for that field. Example 8-1 shows a subroutine to initialize asynchronous I/O, given counts of devices and calling processes.

Example 8-1 Initializing Asynchronous I/O

```
int initAIO(int numDevs, int numSprocs, int maxOps)
{
    aioinit_t A = {0}; /* ensure zero'd fields */
    if (numDevs) /* we do know how many devices */
        A.aio_threads = 1+numDevs;
    if (numSprocs) /* we do know how many sprocs */
        A.aio_locks = A.aio_numusers = 1+numSprocs;
    if (maxOps) /* we do know max aiocbs at 1 time */
        A.aio_num = maxOps;
    return aioinit(&A);
}
```

When to Initialize

The time at which initialization occurs is important. If you initialize in a process that has been assigned to run on an isolated CPU, the asynchronous I/O processes will also run on that CPU. You probably want the I/O processes to run under normal dispatching on unrestricted CPUs. In that case, the proper sequence of initialization is:

- Open all file descriptors and verify that files and devices are ready.
- Initialize asynchronous I/O. The lightweight processes created by **aioinit()** inherit the attributes of the calling process, including its current priority and access to open file descriptors.
- Isolate any CPUs that are dedicated to real-time work (see "Restricting a CPU From Scheduled Work" on page 82)—or create the Frame Schedulers (see "Starting Multiple Schedulers" on page 105).
- Assign real-time processes to their CPUs.

The asynchronous I/O processes created by **aioinit()** continue to be scheduled according to their priority in whatever CPUs remain available.

Scheduling Asynchronous I/O

You schedule an input or output operation by calling **aio_read()** or **aio_write()**, passing an *aiocb* structure to describe the operation (see the aio_read(3) and aio_write(3) reference pages). The operation is queued to the file descriptor, but it will not execute until one of the asynchronous I/O processes is available. The return code from the library call says nothing about the I/O operation itself; it merely indicates whether or not the *aiocb* could be queued.

Note: It is important to use a given *aiocb* for only one operation at a time, and to not modify an *aiocb* until its operation is complete.

You can find examples of the use of **aio_read()**, **aio_write()**, and **aio_fsync()** in the program beginning on “Asynchronous I/O Example” on page 203.

You can schedule a list of operations using **lio_listio()** (see the lio_listio(3) reference page). The advantage of this function is that you can request a single notification (either a signal or a callback) when all of the operations in the list are complete. Alternatively, you can be notified of the completion of each one as it happens.

When an asynchronous I/O process is free, it takes a queued *aiocb* and performs the equivalent function to **lseek()** (if a file position is specified), then the equivalent of **read()** or **write()**. The asynchronous process may be blocked for some time. That depends on the file or device and on the options that were specified when it was opened. When the operation is complete, the asynchronous process notifies the initiating process using the method requested in the *aiocb*.

You can cancel a started operation, or all pending operations for a given file descriptor, using **aio_cancel()** (see the aio_cancel(3) reference page).

Assuring Data Integrity

With sequential I/O, you call **fsync()** to ensure that all buffered data has been written. However, you cannot use **fsync()** with asynchronous I/O, since you are not sure when the **write()** calls will execute.

The **aio_fsync()** function queues the equivalent of an **fsync()** call for asynchronous execution (see the aio_fsync(3) reference page). This function takes an *aiocb*. The file descriptor in it specifies which file is to be synchronized. The **fsync()** operation is done following all other asynchronous operations that are pending when **aio_fsync()** is called. The synchronize operation can take considerable time, depending on how much output data has been buffered. Its completion is reported in the same ways as completion of a read or write (see the next topic). The example program starting in “Asynchronous I/O Example” on page 203 contains calls to **aio_fsync()**.

Checking the Progress of Asynchronous Requests

You can test the progress and completion of an asynchronous operation by polling. Your program can be informed of the completion of an operation in a variety of ways. All of the methods discussed here are demonstrated in the example program that starts in “Asynchronous I/O Example” on page 203.

Polling for Status

You can check the progress of any asynchronous operation (including **aio_fsync()**) using **aio_error()**. As long as the operation is incomplete, this function returns EIINPROGRESS. When the operation is complete, you can check the final return code from **read()**, **write()**, or **fsync()** using **aio_return()** (see the aio_error(3) and aio_return(3) reference pages).

To see in an example of polling for status, see function **inWait0()** under “Asynchronous I/O Example” on page 203. This function is used when the aiocb is initialized with SIGEV_NONE, meaning that no notification is to be returned at the completion of the operation. The function waits for an asynchronous operation to complete using a loop in the general form shown in Example 8-2.

Example 8-2 Polling for Asynchronous Completion

```
int waitForEndOfAsyncOp(aiocb *pab)
{
    while (EINPROGRESS == (ret = aio_error(pab)))
        sginap(0);
    return ret;
}
```

The function result is the final return code from the read, write, or sync operation that was started. Under the Frame Scheduler, the call to **sginap()** would be replaced with a call to **frs_yield()**.

Checking for Completion

In the *aiocb*, the program can specify one of three things to be done when the operation is complete:

- Nothing; take no action.
- Send a signal of a specified number.
- Invoke a callback function directly from the asynchronous process.

In addition, the **aio_suspend()** function blocks its caller until one of a list of pending operations is complete (see the aio_suspend(3) reference page).

These choices give you a wide variety of design options. Your program can

- periodically poll the *aiocb* using **aio_error()** until it completes (shown in Example 8-2)
- use **aio_suspend()** to wait until one of a list of operations completes
- set up an empty signal handler function and use **sigsuspend()** or **sigwait()** to wait until a signal arrives (see the sigsuspend(2) and sigwait(3) reference pages)
- use either a signal handler function or a callback function to report completion—for example, the function can post a semaphore.

Most of these methods are demonstrated in the program starting in "Asynchronous I/O Example" on page 203.

Tip: When operating under the Frame Scheduler, a handler or callback function can simply set a flag. An activity process can test the flag in each minor frame, calling **frs_yield()** immediately if the flag is not set.

Establishing a Completion Signal

You request a signal from an asynchronous operation by setting these values in the *aiocb* (refer to */usr/include/aio.h* and */usr/include/sys/signal.h*):

aio_sigevent.sigev_notify	Set to SIGEV_SIGNAL.
aio_sigevent.sigev_signo	The number of the signal. This should be one of the POSIX real-time signal numbers (see "Signals" on page 36).

aio_sigevent.sigev_value A value to be passed to the signal handler. This can be used to inform the signal handler of which I/O operation has completed; for example, it could be the address of the *aiocb*.

When you set up a signal handler for asynchronous completion, do so using **sigaction()** and specify the SA_SIGINFO flag (see the sigaction(2) reference page). This has two benefits: any new completion signal that arrives while the first is being handled is queued; and the *aio_sigev.sigev_value* word is passed to the handler in a *siginfo* structure.

Establishing a Callback Function

You request a callback at the end of an asynchronous operation by setting the following values in the *aiocb*:

aio_sigevent.sigev_notify Set to SIGEV_CALLBACK.

aio_sigevent.sigev_func The address of the callback function. Its prototype must be `void` *functionName*`(union sigval);`

aio_sigevent.sigev_value A word to be passed to the callback function. This can be used to inform the function of which I/O operation has completed; for example, it could be the address of the *aiocb*.

The callback function is invoked from the asynchronous process when the **read()**, **write()** or **fsync()** operation finishes. This notification method has the lowest overhead and shortest latency, but it requires careful design to avoid race conditions in the use of shared variables.

The asynchronous processes are created with **sproc()**, so they share the address space of the process that initialized asynchronous I/O. They typically execute in a different CPU from the real-time processes using that address space. Since the callback function could be entered at any time, it must coordinate its use of shared data structures. This is a good place to use a lock (see "Locks" on page 33). Locks have very low overhead in cases such as this, where there is likely to be little contention for the use of the lock.

Tip: You can call **aio_read()** or **aio_write()** from within a callback function or within a signal handler. This lets you start another operation with the least delay.

The code in Example 8-3 demonstrates a hypothetical set of subroutines to schedule asynchronous reads and writes using a single *aiocb*. The principle functions and global variables it uses are:

pendingIO — An array of records, each holding one request for an I/O operation.

dontTouchThatStuff — A lock used to gain exclusive use of *pendingIO*.

scheduleRead() — A function that accepts a request to read some amount of data, from a specified file descriptor, at a specified file offset. It places the request in *pendingIO* and then, if no asynchronous operation is under way, initiates it.

yeahWeFinishedOne() — The callback function that is entered when an asynchronous operation completes. If any more operations are pending, it initiates one.

initiatePending() — A function that initiates one selected pending operation. It prepares the *aiocb* structure, including the specification of **yeahWeFinishedOne()** as the callback function. The lock *dontTouchThatStuff* must be held before this function is called.

Note: The code in Example 8-3 is not intended to be realistic and is not recommended as a model. In order to demonstrate the use of callback functions and the *aiocb*, it essentially duplicates work that could be done by the **lio_listio()** feature of asynchronous I/O.

Example 8-3 Set of Functions to Schedule Asynchronous I/O

```
#define _ABI_SOURCE
#include <signal.h>
#include <aio.h>
#include <ulocks.h>
#define MAX_PENDING 10
#define STATUS_EMPTY 0
#define STATUS_ACTIVE 1
#define STATUS_PENDING 2
static struct onePendingIO {
    int status;
    int theFile;
    void *theData;
    off_t theSize;
```

```
    off_t theSeek;
    int readNotWrite;
    } pendingIO[MAX_PENDING];
static unsigned numPending;
static struct aiocb theAiocb;
static ulock_t dontTouchThatStuff;
static unsigned scanner;
static void initiatePending(int P);
static void
yeahWeFinishedOne(union sigval S)
{
    ussetlock(dontTouchThatStuff);
    pendingIO[S.sival_int].status = STATUS_EMPTY;
    if (numPending)
    {
        while (pendingIO[scanner].status != STATUS_PENDING)
        {
            if (++scanner >= MAX_PENDING)
                scanner = 0;
        }
        initiatePending(scanner);
    }
    usunsetlock(dontTouchThatStuff);
}
static void
initiatePending(int P) /* lock must be held on entry */
{
    theAiocb.aio_fildes = pendingIO[P].theFile;
    theAiocb.aio_buf = pendingIO[P].theData;
    theAiocb.aio_nbytes = pendingIO[P].theSize;
    theAiocb.aio_offset = pendingIO[P].theSeek;
    theAiocb.aio_sigevent.sigev_notify = SIGEV_CALLBACK;
    theAiocb.aio_sigevent.sigev_func = yeahWeFinishedOne;
    theAiocb.aio_sigevent.sigev_value.sival_int = P;
    if (pendingIO[P].readNotWrite)
        aio_read(&theAiocb);
    else
        aio_write(&theAiocb);
    pendingIO[P].status = STATUS_ACTIVE;
    --numPending;
}
/*public*/ int
scheduleRead( int FD, void *pdata, off_t len, off_t pos )
{
    int j;
```

```
    if (numPending >= MAX_PENDING)
        likeTotallyFreakOut();
    ussetlock(dontTouchThatStuff);
    for(j=0; pendingIO[j].status != STATUS_EMPTY; ++j)
        ;
    pendingIO[j].theFile = FD;
    pendingIO[j].theData = pdata;
    pendingIO[j].theSize = len;
    pendingIO[j].theSeek = pos;
    pendingIO[j].readNotWrite = 1;
    pendingIO[j].status = STATUS_PENDING;
    if (1 == ++numPending)
        initiatePending(j);
    usunsetlock(dontTouchThatStuff);
}
```

Holding Callbacks Temporarily

You can temporarily prevent callback functions from being entered using the **aio_hold()** function. This function is not defined in the POSIX standard; it is added by the MIPS ABI standard. Use it as follows:

- Call **aio_hold**(AIO_HOLD_CALLBACK) to prevent any callback function from being invoked.
- Call **aio_hold**(AIO_RELEASE_CALLBACK) to allow callback functions to be invoked. Any that were held are now called.
- Call **aio_hold**(AIO_ISHELD_CALLBACK) returns 1 if callbacks are currently being held; otherwise it returns 0.

Multiple Operations to One File

When you queue multiple operations to a single file descriptor, the asynchronous I/O package does not always guarantee the order of their execution. There are three ways you can ensure the sequence of operations.

You can open any output file descriptor passing the flag O_APPEND (see the open(1) reference page). Asynchronous write requests to a file opened with O_APPEND are executed in the sequence of the calls to **aio_write()** or the sequence they are listed for **lio_listio()**. You can use this feature to ensure that a sequence of records is appended to a file in sequence.

For files that support **lseek()**, you can specify any order of operations by specifying the file offset in the *aiocb*. The asynchronous process executes an absolute seek to that offset as part of the operation. Even if the operations are not performed in the sequence they were requested, the data is transferred in sequence. You can use this feature to ensure that multiple requests for sequential disk input are stored in sequential locations.

For non-disk input operations, the only way you can be certain that operations are done in sequence is to schedule them one at a time, waiting for each one to complete.

Synchronous Writing and Direct Writing

Two options of **open()** give you more control over the timing of output.

Using Synchronous Writing

When you open a disk file and do not specify the O_SYNC flag, a call to **write()** for that file returns as soon as the data has been copied to a buffer managed by the device driver (see the open(2) reference page).

The actual disk write may not take place until considerable time has passed. A common pool of disk buffers is used for all disk files. Disk buffering is integrated with the virtual memory paging mechanism. A daemon executes periodically and initiates output of buffered blocks according to the age of the data and the needs of the system.

Tip: The number of disk blocks that are written in each output operation is set by the *dwcluster* tuning variable. The system administrator can adjust this value with *systune* (see the systune(1) reference page).

The default management of disk output improves performance in general but has two drawbacks:

- All output data must be copied from the buffer in process address space to a buffer in the kernel address space. For small or infrequent writes, the copy time is negligible, but for large quantities of data it adds up.
- You do not know when the written data is actually safe on disk. A system crash could prevent the output of a large amount of buffered data.

You can force the writing of all pending output for a file by calling **fsync()** (see the fsync(2) reference page). This gives you a way of creating a known checkpoint of a file. However, **fsync()** blocks until all buffered writes are complete, possibly a long time.

When you open a disk file specifying O_SYNC, each call to **write()** blocks until the data has been written to disk. This gives you a way of ensuring that all output is complete as it is created. If you combine O_SYNC access with asynchronous I/O, you can let the asynchronous process suffer the delay.

The O_SYNC option requires completed output even when the amount of data written is less than the physical blocksize of the disk, or when the output data does not align with the physical boundaries of disk blocks. This can lead to writing and rewriting the same disk blocks, wasting time. A file opened with O_SYNC also copies data to kernel memory before writing.

Using Direct I/O

You can avoid both sources of delay by using the option O_DIRECT. Under this option, writes to the file take place directly from your program's buffer—the data is not copied to a buffer in the kernel first. In order to use O_DIRECT you are required to transfer data in quantities that are multiples of the disk blocksize. This ensures that a block is written only once. (The requirements for O_DIRECT use are documented in the open(2) and fcntl(2) reference pages.)

Control does not return from an O_DIRECT **read()** or **write()** until the disk write is complete. However, you can open a file O_DIRECT and use the use file descriptor for asynchronous I/O.

Performance Comparison

The data displayed in Figure 8-1 was collected on a 4-processor Challenge system under IRIX 5.3, using a test program that wrote approximately 250,000 bytes of binary data using a specified blocksize and one of three options:

- default: asynchronous buffered write
- synchronous writes (option O_SYNC)
- direct writes (option O_DIRECT)

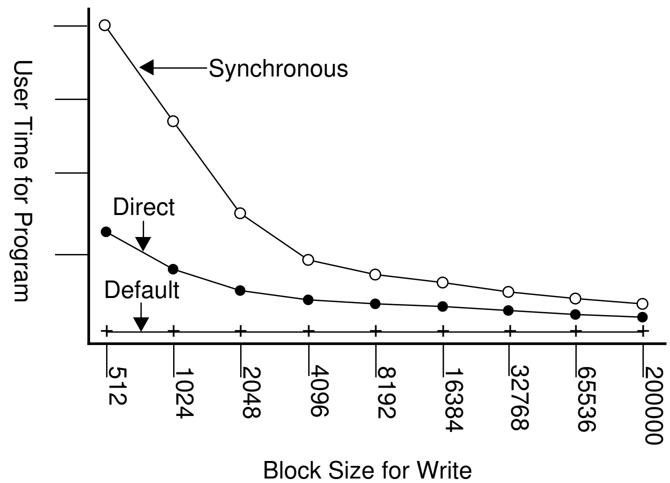

Figure 8-1 Effect of Blocksize on write() Performance

The values in Table 8-1reflect the total execution time for one run of the program, as reported by the *time* command (see the time(1) reference page).

Table 8-1 Data on Which Figure 8-1 is Based

Blocksize	O_SYNC	O_DIRECT	Asynchronous
512	40.4	13.9	2.7
1024	25.3	8.5	2.6
2048	12.9	5.8	2.6
4096	8.5	4.4	2.6
8192	6.2	3.7	2.6
16384	5.0	3.4	2.6
32768	4.4	3.1	2.5
65536	4.1	3.0	2.5
200000	3.9	2.9	2.5

Blocksize was almost irrelevant for asynchronous writes, because the only delay was the time to switch to kernel mode and block-copy the data from the program buffer to a kernel buffer. The actual disk operations occurred asynchronously, in another CPU, and so are not reflected in the *time* output. As shown in Figure 8-1, O_DIRECT is considerably faster than O_SYNC.

Using a Delayed System Buffer Flush

When your application has both clearly defined times when all unplanned disk activity should be prevented, and clearly defined times when disk activity can be tolerated, you can use the **syssgi()** function to control the kernel's automatic disk writes.

Prior to a critical section of length *s* seconds that must not be interrupted by unplanned disk writes, use **syssgi()** as follows:

```
syssgi(SGI_BDFLUSHCNT,s);
```

The kernel will not initiate any deferred disk writes for *s* seconds. At the start of a period when disk activity can be tolerated, initiate a flush of the kernel's buffered writes with **syssgi()** as follows:

```
syssgi(SGI_SSYNC);
```

Note: This technique is most useful in a uniprocessor—code executing in an isolated CPU of a multiprocessor is not affected by kernel disk writes.

Guaranteed-Rate I/O

Under specific conditions, your program can demand a guaranteed rate of data transfer. You would use this feature, for example, to ensure input of picture data for real-time video display, or to ensure disk output of high-speed telemetry data capture.

Guaranteed-Rate I/O Basics

Guaranteed-rate I/O (GRIO) is applied on a file basis. The file must have these characteristics for any guarantee to be granted:

- The file must be managed by XFS. EFS, the older IRIX file system, does not support GRIO.

- The file must be contained in the real-time subvolume of a logical volume created by XLV.

  The real-time subvolume of an XLV volume can span multiple disk partitions, and can be striped. The real-time subvolume differs from the more common data subvolume in that it contains only data, no file system management data such as directories or inodes.

  Note: Real-time subvolumes cannot include RAID partitions.

- The predictive failure analysis feature and the thermal recalibration feature of the drive firmware must be disabled, as these can make device access times unpredictable.
- A guaranteed-rate stream must be available. Unless extra-cost options are installed, a maximum of four streams can be in use at one time.

You can request either of two types of guarantee. A *hard guarantee* asks XFS and IRIX to subordinate all other considerations, including data integrity, to meet the guaranteed rate. A *soft guarantee* asks IRIX to make its best effort at the rate, accepting that error correction might cause glitches.

You can qualify either type of guarantee as being for Video On Demand (VOD), indicating a particular, special use of a striped volume. These three types of guarantee are discussed further in the following topics.

For information about using XFS, XLV, and how to prepare a real-time subvolume for GRIO, see the *IRIX Admin: Disks and Filesystems* manual (see "Other Useful Books" on page xxiii). For an example of how the **grio_request()** function is used, see the function starting in "Guaranteed-Rate Request" on page 221.

Creating a Real-time File

You can only request a guaranteed rate from a real-time disk file. A real-time disk file is identified by the fact that it is stored within the real-time subvolume of an XFS logical volume.

The file management information for all files in a volume (the directories as well as XFS management records) are stored in the data subvolume. A real-time subvolume contains only the data of real-time files. A real-time subvolume comprises an entire disk device or partition and uses a separate SCSI controller from the data subvolume. Because of these constraints, the GRIO facility can predict the data rate at which it can transfer the data of a real-time file.

You create a real-time file in the following steps, which are illustrated in Example 8-4.

1. Open the file with the options O_CREAT, O_EXCL, and O_DIRECT. That is, the file must not exist at this point, and must be opened for direct I/O (see “Using Direct I/O” on page 148).
2. Modify the file descriptor to set its extent size, which is the minimum amount by which the file will be extended when new space is allocated to it, and also to establish that the new file is a real-time file. This is done using **fcntl()** with the FS_FSSETXATTR command. Check the value returned by **fcntl()** as several errors can be detected at this point.

   The extent size must be chosen to match the characteristics of the disk; for example it might be the “stripe width” of a striped disk.
3. Write any amount of data to the new file. Space will be allocated in the real-time subvolume instead of the data subvolume because of step (2). Check the result of the first **write()** call carefully, since this is another point at which errors could be detected.

Once created, you can read and write a real-time file the same as any other file, except that it must always be opened with O_DIRECT. You can use a real-time file with asynchronous I/O, provided it is not under a guarantee (see “Sharing Access to Guaranteed Files” on page 154).

Example 8-4 Function to create a real-time file

```
#include <sys/fcntl.h>
#include <sys/fs/xfs_itable.h>
int createRealTimeFile(char *path, __uint32_t esize)
{
   struct fsxattr attr;
   bzero((void*)&attr,sizeof(attr));
   attr.fsx_xflags = XFS_XFLAG_REALTIME;
   attr.fsx_extsize = esize;
   int rtfd = open(path, O_CREAT + O_EXCL + O_DIRECT );
   if (-1 == rtfd)
      {perror("open new file"); return -1; }
   if (-1 == fcntl(rtfd, F_FSSETXATTR, &attr) )
      {perror("fcntl set rt & extent"); return -1; }
   return rtfd; /* first write to it creates file*/
}
```

Requesting a Guarantee

To obtain a guaranteed rate, a program places a reservation for a specified part of the I/O capacity of a file. In the request, the program specifies

- the file descriptor to be used
- the start time and duration of the reservation
- the time unit of interest, typically 1 second
- the amount of data required in any one unit of time

For example, a reservation might specify: now, for 90 minutes, 1 megabyte per second. A process places a reservation by calling **grio_request()** (refer to the grio_request(3X) reference page).

XFS (in a GRIO daemon) keeps information on the transfer capacity of all real-time subvolumes, as well as the capacity of the controllers and busses to which they are attached. When you request a reservation, XFS tests whether it is possible to transfer data at that rate, from that file, during that time period.

This test considers the capacity of the hardware as well as any other reservations that apply during the same time period to the same subvolume, drives, or controllers. Each reservation consumes some part of the total capacity.

When XFS predicts that the guaranteed rate can be met, it accepts the reservation. Over the reservation period, the available capacity of the subvolume is reduced by the promised rate. Other processes can place reservations against any capacity that remains.

If XFS predicts that the guaranteed rate cannot be met at some time in the reservation period, XFS returns the maximum data rate it could supply. The program can reissue the request for that available rate. However, this is a new request that is evaluated afresh.

During the reservation period, the process can use **read()** and **write()** to transfer up to the guaranteed number of bytes in each time unit. XFS raises the priority of requests as needed in order to ensure that the transfers take place. However, a request that would transfer more than the promised number of bytes within a 1-second unit is blocked until the start of the next time unit.

Releasing a Guarantee

A guarantee ends under three circumstances,

- when the process calls **grio_remove_request()** (see the grio_remove_request(3X) reference page)
- when the requested duration expires
- when all file descriptors held by the requesting process that refer to the guaranteed file are closed (an exception is discussed in the next topic)

When a guarantee ends, the guaranteed transfer capacity becomes available for other processes to reserve. When a guarantee expires but the file is not closed, the file remains usable for ordinary I/O, with no guarantee of rate.

Sharing Access to Guaranteed Files

Other processes can use a file or the hardware it resides on, even though guarantees are active. XFS never grants guarantees for the whole capacity of the I/O path; it always reserves some capacity. Non-guaranteed I/O requests are delayed within any 1-second interval until guarantees have been met, and may be executed bit by bit in smaller units, but they will finally be completed.

Once a guarantee is granted, the guarantee is uniquely identified with the file, through the I-node number, and with the process, through the process ID. However, it is possible to have the same file (I-node) open under different file descriptors. This has important implications:

- All requests from that process to that file are handled under the guarantee—even if they are issued to different file descriptors. (It is not possible for a single process to request both guaranteed and nonguaranteed I/O to the same file.)
- It is not possible for one process to have two guarantees on the same file. The second guarantee request is rejected, even if it uses a different file descriptor.
- Only the process that received a guarantee can remove the guarantee—that is, **grio_remove_request()** must be called from the same process ID that called **grio_request()**.
- A rate guarantee is not shared by other processes created with **fork()** or **sproc()**—even though they may have shared access to the file descriptor used with

grio_request(). Each process that wants guaranteed access must obtain its own guarantee.

The last point has the important implication that you cannot use a rate guarantee with asynchronous I/O. An input requested using **aio_read()** is executed by a different process than the one that requested the guaranteed rate. That read is treated as non-guaranteed, and executed on a time-available basis.

A complication can arise when a guaranteed rate is obtained by one process of a process group created with **sproc()**. When the PR_SDIR flag (synchronize file descriptors; see the sproc(2) reference page) is used, a rate guarantee obtained by one process of the group cannot be terminated simply by closing all file descriptors. It can be terminated explicitly, or by the time expiring, or by the whole process group terminating.

Hard Guarantees

When a program requests a hard guarantee, it asserts that nothing, not even data integrity, should interfere with data transfer. A hard guarantee can be given only when

- the SCSI controller or controllers that attach the real-time subvolume have only disks attached to them—no tapes or other nondisk devices

  I/O to a non-disk device can delay disk data transfer.
- sector remapping in the drive firmware, as well as any device driver retry and correction mechanisms, is disabled

  Error retry can introduce unpredictable delays in data transfer.

When your program requests I/O under a hard guarantee, any device error is returned directly to the program. No effort is made to retry the failure. If the drive contains a bad sector, the bad sector is read and returned with no indication of error.

Soft Guarantees

A soft guarantee can be granted for a subvolume that has error retry and sector remapping enabled. Your program accepts a possible, occasional failure to meet the specified rate in exchange for having errors retried and possibly corrected.

In addition, a soft guarantee can be granted when the disk controller also controls non-disk devices such as scanners and printers. Use of these devices during the guarantee period can prevent the guaranteed rate from being met.

Video On Demand (VOD) Guarantees

You specify the VOD disk layout as a modifier on either a hard or soft guarantee (see the grio_request(2) reference page and */usr/include/sys/grio.h*). A VOD guarantee can be requested only for a *striped volume*. In a striped volume, fixed-size segments of the volume space that are logically sequential ("stripes") are physically located on successive drives. The potential data rate of a striped volume is higher because the multiple drives can be used in parallel.

However, in order to achieve the higher rate, the striped volume must be used concurrently by multiple processes, each reading in a different stripe. The maximum rate is reached when as many processes are reading sequentially in stripe-sized units as the subvolume has drives.

When a program requests a VOD guarantee, it must specify a data rate that equals one stripe-width per second. VOD guarantees can be given concurrently to several processes for the same subvolume. As long as all the processes read different stripes, the guaranteed rate can be sustained for each.

When the first VOD guarantee is granted against a striped volume, the XFS system begins VOD-style I/O scheduling for that volume. This establishes a strict cyclic rotation of time intervals during which any disk in the striped volume can be read. In general, a process must be ready for access when its turn in the rotation comes up. If it is not ready, it can be delayed by as many seconds as there are disks in the volume.

- The first access by a process to a striped volume under VOD scheduling can be delayed.
- If the process fails to request its next access before the beginning of the next second of time, it can miss its assigned slot and be delayed.
- When a process uses **lseek()** to move to a stripe other than the next stripe in sequence, its next I/O request can be delayed.