July 27, 1993

Using Asynchronous I/O and Direct I/O in IRIX 5.X

Bill Mannel, Huy Nguyen, Kris Solem
Silicon Graphics Inc.

This paper is intended to provide guidance in programming applications using asynchronous I/O and direct I/O. It builds upon work done by Jay McCauley and Dave Ciemiewicz, of Silicon Graphics, and uses materials, drawings, and code examples developed by them.

Introduction

Asynchronous I/O and direct I/O are two new features available with the 5.X release of IRIX. They provide finer control of I/O operations than those provided by standard UNIX^{TM} I/O mechanisms, and will have many applications in real-time processing, fast data transfer, and on-line transaction processing (OLTP).

Asynchronous I/O

With IRIX 5.X, support for asynchronous I/O was added in accordance with the specification in POSIX 1003.4a Draft 12. With asynchronous I/O, a user can queue read and write requests to a device, and optionally receive a queued signal when the request completes. The read or write function call will return when the request is queued, rather than blocking the process. As such, a process can simultaneously queue a number of requests, without waiting for any of them to complete.

Asynchronous I/O is performed via *sproc(2)*'ed processes. When a process first executes an aio request, an aio initialization routine is started. This routine, by default, creates four sproc processes to handle requests. These sprocs then wait on a semaphore for work. When a user issues an *aio_read(3)* or *aio_write(3)* request, the request is attached to a linked list of outstanding requests, and the semaphore is incremented to 'kick off' a slave process to service this transaction.

Normally aio operations are performed in the order of process priority. The user can optionally reduce the priority of a particular request.

Also, a signal can be sent to the process upon completion of the transfer.

Using Asynchronous I/O

Asynchronous I/O uses a control block to set up the transaction. This control block is of type *aiocb_t* and is defined in */usr/include/aio.h*. The following elements in the control block can be modified by the user:

Member Type	Member Name	Member Description
int	aio_fildes	file descriptor to perform aio on
off_t	aio_offset	file offset position; used only if O_APPEND is not set for aio_fildes
volatile void *	aio_buf	buffer to write from/read into
size_t	aio_nbytes	number of bytes to read/write
int	aio_reqprio	aio priority
struct sigevent	aio_sigevent	signal to be generated on completion
int	aio_lio_opcode	opcode for lio_listio() call; one of LIO_READ, LIO_WRITE or LIO_NOP

In order to queue an aio request, the user must:

- open a file using *open(2)* to get a file descriptor
- fill at least the first four elements of the aio control block
- use *aio_read(3)*,*aio_write(3)*, *lio_listio(3)* to queue the request

When filling the aio control block, the user specifies a file descriptor, an offset (use 0 if starting at the beginning of the file), a pointer to a buffer to write into/read from, and the number of bytes to transfer.

Normally aio requests are queued in the order of the priority of the processes initiating the requests. However, the user has the option of lowering the priority (making it worse) by assigning a value other than 0 to *aio_reqprio* in the aio control block. This value will be added to the process priority to determine the order.

If *aio_sigevent.sevt_signo* is set to a valid signal number (see *signal(5)*), the async I/O sproc will send the requested signal to the user process at the completion of the aio transfer. Setting this value to 0 will prevent a signal from being sent.

One of the more powerful features of asynchronous I/O is that a user process can simultaneously enqueue a number of requests using the lio_listio(3) call:

*int lio_listio(int mode, aiocb_t **list, int nent, sigevent_t *sig)*

Here the process requests *nent* aio transactions. If *mode* is set to *LIO_WAIT*, the process will block until all *nent* transactions are complete. If set to *LIO_NOWAIT*, the process will not wait, but instead will be signaled with the signal specified in *sig.sevt_signo*. (If *sig* is set to *NULL* or *sig.sevt_signo* is set to 0, no signal will be delivered). In the case of *lio_listio(3)*, any setting of signal delivery in the individual control blocks will have no effect.

The user process can choose to synchronously wait for aio completion with the *aio_suspend(3)* call:

*int aio_suspend(const aiocb_t **aiocbp, int cnt, timespec_t *timeout)*

The *aiocbp* argument is a *cnt* list of pointers to aio control blocks, and the *timeout* is the amount of time to wait for completion before returning from the call. This gives a program the capability of queuing a number of aio requests, and then waiting until:

- at least one of them is completes
- the program is interrupted by a signal
- the timeout specified in the call expires

The user has limited control of the aio request once it has been queued. Requests can only be cancelled or queried.

aio_cancel(3) can be used to cancel a pending aio request:

*int aio_cancel(int fildes, struct aiocb *aiocbp)*

Setting *aiocbp* to *NULL* will cause all outstanding aio requests on that file descriptor to be canceled.

The function call will return:

Return Value	Description
AIO_CANCELED	all outstanding aio requests were canceled
AIO_NOTCANCELED	some requests, but not all, were canceled
AIO_ALLDONE	all requests were completed before canceling

Canceling an aio request will cause any requested signal to be delivered to the process that initiated the request.

There are two ways to query aio completion status:

- When using *aio_suspend(3)* along with *aio_error(3)* and *aio_return(3)*, the user process incurs the least amount of overhead using aio. Upon return from *aio_suspend(3)*, *aio_error(3)* and *aio_return(3)* can be applied to the individual aio control block for completion status.
- Instead of waiting for the aio completion synchronously, the process can continue its execution and receive the completion notice via the IRIX signal mechanism. The signal type to be delivered is specified as described previously either in the *aio_sigevent.sevt_signo* field of the aio control block, or as the *sig.sevt_signo* argument to *lio_listio(3)*. Because of the overhead of asynchronous signal delivery, this method is most appropriate when:
  - the number of outstanding async I/O requests is small
  - the process cannot afford to block, as it has other time sensitive tasks to complete

Asynchronous I/O Example

The following simple example shows an aio read being initiated:

```
#include <aio.h>
#define   READ_BUFFER_SIZE 4096 * 4
/* Signal handler for read completion */
extern void read_complete_handler();


aiocb_t   aio_request;     /* AIO request control block */
aiocb_t *aio_p = &aio_request;
off_t     offset;          /* Program maintained offset */
char      read_buf[READ_BUFFER_SIZE];   /* Read req buf */

void read_complete_handler(){
       /* Signal handlers should be as short as possible
        * so as not to cause additional overhead
        * /
       printf("aio transfer complete\n");
}


main(int argc, char* argv[]) {
       int  fd, retval;
       timespec_t timeout;
       /* open file for asynchronous reads */
       if ((fd = open(argv[1], O_RDONLY)) == -1) {
          perror("open");
          exit(1);
       }
       /* establish the handler for read completion */
       sigset(SIGUSR1, read_complete_handler);

       offset = 0;         /* Start read at beginning */
```

```
        for (;;) {
            int status = 0;

            /* Set up asynchronous read control block */
            aio_request.aio_fildes        = fd;
            aio_request.aio_offset        = offset;
            aio_request.aio_buf           = read_buf;
            aio_request.aio_nbytes        = READ_BUFFER_SIZE;
            aio_request.aio_reqprio       = 0;
            aio_request.aio_sigevent.sigev_signo = SIGUSR1;

            /* Enqueue asynchronous read request */
            aio_read(&aio_request);

            /* suspend until the aio is complete, a signal is
            received, or the timeout expires */
            timeout.tv_sec = 1;
            timeout.tv_nsec = 0;
            retval = aio_suspend((const aiocb_t **)&aio_p,
                                      1, &timeout);

            /* Should not get here until read is done */
            status = aio_return(&aio_request);
            if (status > 0) {
                    printf("bytes read %d\n", status);
                    offset += status;
            } else if (status == 0) {
                    printf("<EOF>\ntotal read = %d\n", offset);
                    exit(0);
            } else if (status == -1) {
                    printf("total read = %d\n", offset);
                    printf("aio_read: %s\n",
                            strerror(aio_error(&aio_request)));
                    exit(1);
            }
        }
}
```

Asynchronous I/O Caveats

Regardless of the number of aio slave processes and processors available, no more than one request can occur simultaneously *on a given file descriptor in a regular file system*. This is because the file table entry is locked for use by one process at a time, in order to ensure consistency and avoid race conditions in the update of the file offset. To get around this problem, a regular file can be opened multiple times, thereby providing multiple file table entries. Concurrent aio request control blocks can be assigned to a different file descriptor of the same file; the user then must take care to ensure consistency of writes into/reads from this file. (When writing to a *raw device*, multiple requests can occur simultaneously on a *single file descriptor*. Since writing via the raw device file bypasses the normal file system mechanism, there are no file table entries to lock and unlock.)

Asynchronous I/O Applications

There are several key applications which can benefit from the use of asynchronous I/O. It allows a real-time process to process I/O requests without blocking, and reduces the latency involved with an I/O operation. When used in conjuction with direct I/O, asynchronous I/O can be used to give an application complete control over I/O buffering, thereby customizing a buffering scheme for a particular use, such as fast video data processing. An OLTP process can collect a series of requests, enqueue them, then return to further input processing while the transactions are completed.

Direct I/O

Direct I/O allows an application to bypass the RAM buffer cache, and gain I/O performance when reading and writing large files.

Normal EFS I/O utilizes a delay write and read ahead mechanism whereby data to and from disk files are buffered in memory. Traditional UNIX implementations provided a specified area of memory called the *buffer cache* in which to buffer this file system I/O. The number and size of these buffers were typically set by UNIX kernel reconfiguration. Starting with IRIX 3.3, the concept of a buffer cache limited in size was obsoleted in favor of an *integrated page and data cache*, in which the size of the buffer cache can grow and shrink with I/O demands. In this scheme, perhaps a significant portion of available RAM memory can be used for buffering file system I/O.

Data cached in the integrated page and data cache is written out by the *bdflush* daemon, which operates every five seconds. Pages of data that have sufficiently aged (not been modified recently) are then written out to disk.

The original intention of caching file system data in memory was to reduce the amount of actual disk I/O taking place. The design favored a usage pattern in which blocks written to disk are frequently read back in, modified, and written back. Finding the data more often in memory than out on disk should improve overall I/O performance. However, this caching system has one shortfall: data must be copied from user and kernel space (an operation called a *bcopy*), in order to become a part of the cache. When reading and writing large files, the amount of extra copies can cause significant performance degradation.

Through the use of direct I/O, this performance degradation can be avoided, because data is transferred directly from the user process to the disk (a process called Direct Memory Access, or DMA), without being cached in memory:

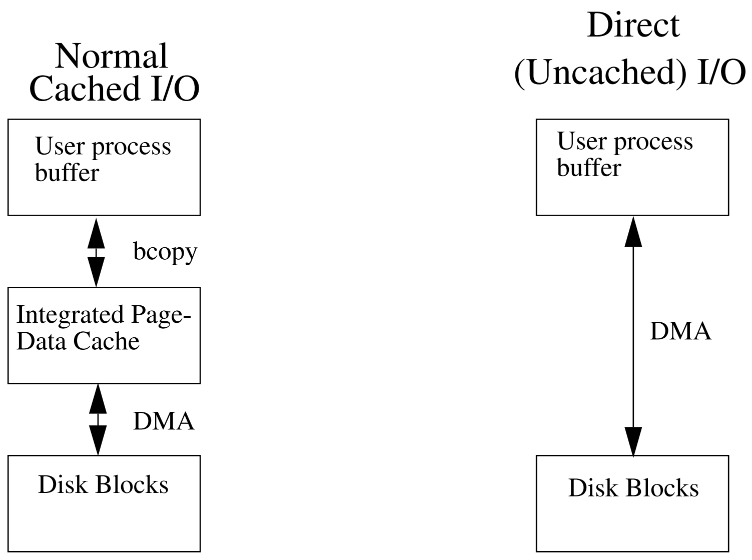

Figure 1 Comparison between normal and direct I/O

One large disadvantage of direct I/O is that the data is always written *synchronously*, i.e. the user process will not return from the read or write until the data is transferred. Since this synchronous behavior may in turn decrease performance, it is suggested that marrying asynchronous I/O with direct I/O will provide the best performance gains.

Using Direct I/O

When performing direct I/O on a file, the user buffers must conform to the characteristics of the raw disk partition. The requirement sets a minimum transfer size (the *block size* of the file system) and alignment of data along set boundaries. Since these parameters can vary from file system to file system, the user program should use an *fcntl(2)* call to query for this information:

```
retval = fcntl(fd, F_DIOINFO, &dioinfo)
```

where *dioinfo* is of type *struct dioattr*:

```
struct dioattr {
            unsigned d_mem;          /* buffer alignment */
            unsigned d_miniosz;      /* min transfer size */
            unsigned d_maxiosz;      /* max transfer size */
}
```

The value returned in *d_mem* can then be used as an argument in *memalign(3C)* to malloc buffer space on the proper boundary:

*buffer = memalign(dioinfo.d_mem, 100 * dioinfo.d_miniosz)*

The *d_miniosz* field specifies the smallest size, in bytes, that can be transferred at one time. The size of all transfers must be some multiple of *d_miniosz*, not to exceed *d_maxiosz*. (The defaults for EFS file systems are 512 byte boundaries, minimum transfer size of 512 bytes, and a maximum transfer size of 4 Mb.)

To initiate direct I/O on a file, the file must be *open(2)*'ed using the *O_DIRECT* flag:

retval = open("myfile", O_RDONLY | O_DIRECT)

If the file has been previously opened without the *O_DIRECT* flag, the *FDIRECT* flag of *fcntl(2)* can be used:

retval = fcntl(fd, F_SETFL, FDIRECT)

Direct I/O example

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

main()
{
       struct dioattr dioinfo;
       int fd, retval, size;
       int *data;
       fd = open("/usr/tmp/test_dio", O_CREAT|O_DIRECT);
       retval = fcntl(fd, F_DIOINFO, &dioinfo);

       /* allocate a buffer 1000 * the min size */
       size = 1000 * dioinfo.d_miniosz;
       data = (int *) memalign(dioinfo.d_mem, size);

       /* fill buffer with pixel data */

       write(fd, (void *)data, size);

}
```

Direct I/O Caveats

Direct I/O is an SGI extension available with the SGI proprietary Extent File System (EFS) only.

Direct I/O Applications

Users will find direct I/O especially useful when transfering large video and audio files back and forth between memory and disk. Coupled with asynchronous I/O, direct I/O can give an application the opportunity to use its own customized buffering scheme to most efficiently move large amounts of data.