NAME

aio − POSIX asynchronous I/O overview

DESCRIPTION

The POSIX asynchronous I/O (AIO) interface allows applications to initiate one or more I/O operations that are performed asynchronously (i.e., in the background). The application can elect to be notified of completion of the I/O operation in a variety of ways: by delivery of a signal, by instantiation of a thread, or no notification at all.

The POSIX AIO interface consists of the following functions:

aio_read(3)
: Enqueue a read request. This is the asynchronous analog of **read**(2).

aio_write(3)
: Enqueue a write request. This is the asynchronous analog of **write**(2).

aio_fsync(3)
: Enqueue a sync request for the I/O operations on a file descriptor. This is the asynchronous analog of **fsync**(2) and **fdatasync**(2).

aio_error(3)
: Obtain the error status of an enqueued I/O request.

aio_return(3)
: Obtain the return status of a completed I/O request.

aio_suspend(3)
: Suspend the caller until one or more of a specified set of I/O requests completes.

aio_cancel(3)
: Attempt to cancel outstanding I/O requests on a specified file descriptor.

lio_listio(3)
: Enqueue multiple I/O requests using a single function call.

The *aiocb* ("asynchronous I/O control block") structure defines parameters that control an I/O operation. An argument of this type is employed with all of the functions listed above. This structure has the following form:

```
#include <aiocb.h>

struct aiocb {
    /* The order of these fields is implementation-dependent */

    int             aio_fildes;     /* File descriptor */
    off_t           aio_offset;     /* File offset */
    volatile void  *aio_buf;        /* Location of buffer */
    size_t          aio_nbytes;     /* Length of transfer */
    int             aio_reqprio;    /* Request priority */
    struct sigevent aio_sigevent;   /* Notification method */
    int             aio_lio_opcode; /* Operation to be performed;
                                       lio_listio() only */

    /* Various implementation-internal fields not shown */
};

/* Operation codes for 'aio_lio_opcode': */

enum { LIO_READ, LIO_WRITE, LIO_NOP };
```

The fields of this structure are as follows:

aio_fildes
The file descriptor on which the I/O operation is to be performed.

aio_offset
This is the file offset at which the I/O operation is to be performed.

aio_buf
This is the buffer used to transfer data for a read or write operation.

aio_nbytes
This is the size of the buffer pointed to by *aio_buf*.

aio_reqprio
This field specifies a value that is subtracted from the calling thread's real-time priority in order to determine the priority for execution of this I/O request (see **pthread_setschedparam**(3)). The specified value must be between 0 and the value returned by *sysconf(_SC_AIO_PRIO_DELTA_MAX)*. This field is ignored for file synchronization operations.

aio_sigevent
This field is a structure that specifies how the caller is to be notified when the asynchronous I/O operation completes. Possible values for *aio_sigevent.sigev_notify* are **SIGEV_NONE**, **SIGEV_SIGNAL**, and **SIGEV_THREAD**. See**sige vent**(7) for further details.

aio_lio_opcode
The type of operation to be performed; used only for **lio_listio**(3).

In addition to the standard functions listed above, the GNU C library provides the following extension to the POSIX AIO API:

aio_init(3)
Set parameters for tuning the behavior of the glibc POSIX AIO implementation.

ERRORS

EINVAL
The *aio_reqprio* field of the *aiocb* structure was less than 0, or was greater than the limit returned by the call *sysconf(_SC_AIO_PRIO_DELTA_MAX)*.

VERSIONS

The POSIX AIO interfaces are provided by glibc since glibc 2.1.

STANDARDS

POSIX.1-2001, POSIX.1-2008.

NOTES

It is a good idea to zero out the control block buffer before use (see **memset**(3)). The control block buffer and the buffer pointed to by *aio_buf* must not be changed while the I/O operation is in progress. These buffers must remain valid until the I/O operation completes.

Simultaneous asynchronous read or write operations using the same *aiocb* structure yield undefined results.

The current Linux POSIX AIO implementation is provided in user space by glibc. This has a number of limitations, most notably that maintaining multiple threads to perform I/O operations is expensive and scales poorly. Work has been in progress for some time on a kernel state-machine-based implementation of asynchronous I/O (see **io_submit**(2), **io_setup**(2), **io_cancel**(2), **io_destroy**(2), **io_getevents**(2)), but this implementation hasn't yet matured to the point where the POSIX AIO implementation can be completely reimplemented using the kernel system calls.

EXAMPLES

The program below opens each of the files named in its command-line arguments and queues a request on the resulting file descriptor using **aio_read**(3). The program then loops, periodically monitoring each of the I/O operations that is still in progress using **aio_error**(3). Each of the I/O requests is set up to provide notification by delivery of a signal. After all I/O requests have completed, the program retrieves their status using **aio_return**(3).

The **SIGQUIT** signal (generated by typing control-\) causes the program to request cancelation of each of the outstanding requests using **aio_cancel**(3).

Here is an example of what we might see when running this program. In this example, the program queues two requests to standard input, and these are satisfied by two lines of input containing "abc" and "x".

```
$ ./a.out /dev/stdin /dev/stdin
opened /dev/stdin on descriptor 3
opened /dev/stdin on descriptor 4
aio_error():
    for request 0 (descriptor 3): In progress
    for request 1 (descriptor 4): In progress
abc
I/O completion signal received
aio_error():
    for request 0 (descriptor 3): I/O succeeded
    for request 1 (descriptor 4): In progress
aio_error():
    for request 1 (descriptor 4): In progress
x
I/O completion signal received
aio_error():
    for request 1 (descriptor 4): I/O succeeded
All I/O requests completed
aio_return():
    for request 0 (descriptor 3): 4
    for request 1 (descriptor 4): 2
```

Program source

```
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <aio.h>
#include <signal.h>

#define BUF_SIZE 20     /* Size of buffers for read operations */

#define errExit(msg) do { perror(msg); exit(EXIT_FAILURE); } while (0)

struct ioRequest {      /* Application-defined structure for tracking
                           I/O requests */
    int           reqNum;
    int           status;
    struct aiocb *aiocbp;
};

static volatile sig_atomic_t gotSIGQUIT = 0;
                        /* On delivery of SIGQUIT, we attempt to
                           cancel all outstanding I/O requests */

static void             /* Handler for SIGQUIT */
quitHandler(int sig)
{
```

```
        gotSIGQUIT = 1;
    }

    #define IO_SIGNAL SIGUSR1   /* Signal used to notify I/O completion */

    static void                 /* Handler for I/O completion signal */
    aioSigHandler(int sig, siginfo_t *si, void *ucontext)
    {
        if (si->si_code == SI_ASYNCIO) {
            write(STDOUT_FILENO, "I/O completion signal received\n", 31);

            /* The corresponding ioRequest structure would be available as
                    struct ioRequest *ioReq = si->si_value.sival_ptr;
               and the file descriptor would then be available via
                    ioReq->aiocbp->aio_fildes */
        }
    }

    int
    main(int argc, char *argv[])
    {
        struct sigaction sa;
        int s;
        int numReqs;        /* Total number of queued I/O requests */
        int openReqs;       /* Number of I/O requests still in progress */

        if (argc < 2) {
            fprintf(stderr, "Usage: %s <pathname> <pathname>...\n",
                    argv[0]);
            exit(EXIT_FAILURE);
        }

        numReqs = argc - 1;

        /* Allocate our arrays. */

        struct ioRequest *ioList = calloc(numReqs, sizeof(*ioList));
        if (ioList == NULL)
            errExit("calloc");

        struct aiocb *aiocbList = calloc(numReqs, sizeof(*aiocbList));
        if (aiocbList == NULL)
            errExit("calloc");

        /* Establish handlers for SIGQUIT and the I/O completion signal. */

        sa.sa_flags = SA_RESTART;
        sigemptyset(&sa.sa_mask);

        sa.sa_handler = quitHandler;
        if (sigaction(SIGQUIT, &sa, NULL) == -1)
            errExit("sigaction");

        sa.sa_flags = SA_RESTART | SA_SIGINFO;
```

```
        sa.sa_sigaction = aioSigHandler;
        if (sigaction(IO_SIGNAL, &sa, NULL) == -1)
            errExit("sigaction");

        /* Open each file specified on the command line, and queue
           a read request on the resulting file descriptor. */

        for (size_t j = 0; j < numReqs; j++) {
            ioList[j].reqNum = j;
            ioList[j].status = EINPROGRESS;
            ioList[j].aiocbp = &aiocbList[j];

            ioList[j].aiocbp->aio_fildes = open(argv[j + 1], O_RDONLY);
            if (ioList[j].aiocbp->aio_fildes == -1)
                errExit("open");
            printf("opened %s on descriptor %d\n", argv[j + 1],
                    ioList[j].aiocbp->aio_fildes);

            ioList[j].aiocbp->aio_buf = malloc(BUF_SIZE);
            if (ioList[j].aiocbp->aio_buf == NULL)
                errExit("malloc");

            ioList[j].aiocbp->aio_nbytes = BUF_SIZE;
            ioList[j].aiocbp->aio_reqprio = 0;
            ioList[j].aiocbp->aio_offset = 0;
            ioList[j].aiocbp->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
            ioList[j].aiocbp->aio_sigevent.sigev_signo = IO_SIGNAL;
            ioList[j].aiocbp->aio_sigevent.sigev_value.sival_ptr =
                                    &ioList[j];

            s = aio_read(ioList[j].aiocbp);
            if (s == -1)
                errExit("aio_read");
        }

        openReqs = numReqs;

        /* Loop, monitoring status of I/O requests. */

        while (openReqs > 0) {
            sleep(3);       /* Delay between each monitoring step */

            if (gotSIGQUIT) {

                /* On receipt of SIGQUIT, attempt to cancel each of the
                   outstanding I/O requests, and display status returned
                   from the cancelation requests. */

                printf("got SIGQUIT; canceling I/O requests: \n");

                for (size_t j = 0; j < numReqs; j++) {
                    if (ioList[j].status == EINPROGRESS) {
                        printf("    Request %zu on descriptor %d:", j,
                                ioList[j].aiocbp->aio_fildes);
```

```
                    s = aio_cancel(ioList[j].aiocbp->aio_fildes,
                            ioList[j].aiocbp);
                    if (s == AIO_CANCELED)
                        printf("I/O canceled\n");
                    else if (s == AIO_NOTCANCELED)
                        printf("I/O not canceled\n");
                    else if (s == AIO_ALLDONE)
                        printf("I/O all done\n");
                    else
                        perror("aio_cancel");
                }
            }

            gotSIGQUIT = 0;
        }

        /* Check the status of each I/O request that is still
           in progress. */

        printf("aio_error():\n");
        for (size_t j = 0; j < numReqs; j++) {
            if (ioList[j].status == EINPROGRESS) {
                printf("    for request %zu (descriptor %d): ",
                        j, ioList[j].aiocbp->aio_fildes);
                ioList[j].status = aio_error(ioList[j].aiocbp);

                switch (ioList[j].status) {
                case 0:
                    printf("I/O succeeded\n");
                    break;
                case EINPROGRESS:
                    printf("In progress\n");
                    break;
                case ECANCELED:
                    printf("Canceled\n");
                    break;
                default:
                    perror("aio_error");
                    break;
                }

                if (ioList[j].status != EINPROGRESS)
                    openReqs--;
            }
        }
    }

    printf("All I/O requests completed\n");

    /* Check status return of all I/O requests. */

    printf("aio_return():\n");
    for (size_t j = 0; j < numReqs; j++) {
        ssize_t s;
```

```
            s = aio_return(ioList[j].aiocbp);
            printf("    for request %zu (descriptor %d): %zd\n",
                    j, ioList[j].aiocbp->aio_fildes, s);
        }

        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

io_cancel(2), **io_destroy**(2), **io_getevents**(2), **io_setup**(2), **io_submit**(2), **aio_cancel**(3), **aio_error**(3), **aio_init**(3), **aio_read**(3), **aio_return**(3), **aio_write**(3), **lio_listio**(3)

"Asynchronous I/O Support in Linux 2.5", Bhattacharya, Pratt, Pulavarty, and Morgan, Proceedings of the Linux Symposium, 2003, ⟨https://www.kernel.org/doc/ols/2003/ols2003−pages−351−366.pdf⟩