System Programming

Week 13: Advanced I/O

Seongjin Lee

Updated: 2016-11-16
10_adv-io

insight@hanyang.ac.kr
http://esos.hanyang.ac.kr
Esos Lab. Hanyang University

Table of contents

Introduction

This chapter covers

- Nonblocking I/O
- Record Locking
- Asynchronous I/O
- memory-mapped I/O (`mmap`)

Nonblocking I/O

Nonblocking I/O

Disk I/O are not considered slow

Nonblocking I/O lets us issue an I/O operation, such as an `open`, `read`, or `write`, and not have it block forever.

- if the operation cannot be completed, the call returns immediately with an error noting that the operation would have blocked

Two ways to specify nonblocking I/O for a given descriptor

- open with `O_NONBLOCK` flag
- For already opened descriptors, use `fcntl` to turn on the `O_NONBLOCK` file status flag

Nonblocking I/O Example: codes/nonblocking.c I

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

char buf[500000];



/* flags are the file status flags to turn off */
void
clr_fl(int fd, int flags)
{
   int  val;

   if ((val = fcntl(fd, F_GETFL, 0)) < 0){
      fprintf(stderr, "fcntl F_GETFL error");
      exit(1);
   }

   val &= ~flags;  /* turn flags off */
```

```

   if (fcntl(fd, F_SETFL, val) < 0){
      fprintf(stderr, "fcntl F_SETFL error");
      exit(1);
   }
}

/* flags are file status flags to turn on */
void
set_fl(int fd, int flags)
{
   int  val;

   if ((val = fcntl(fd, F_GETFL, 0)) < 0){
      fprintf(stderr, "fcntl F_GETFL error");
      exit(1);
   }

   val |= flags;  /* turn on flags */

   if (fcntl(fd, F_SETFL, val) < 0){
      fprintf(stderr, "fcntl F_SETFL error");
      exit(1);
```

```
   }
}

int
main(void)
{
   int ntowrite, nwrite;
   char *ptr;
   ntowrite = read(STDIN_FILENO, buf, sizeof(buf));
   fprintf(stderr, "read %d bytes\n", ntowrite);

   set_fl(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */

   ptr = buf;

   while (ntowrite > 0) {
      errno = 0;
      nwrite = write(STDOUT_FILENO, ptr, ntowrite);
      fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);
      if (nwrite > 0) {
         ptr += nwrite;
         ntowrite -= nwrite;
      }
```

Nonblocking I/O Example: codes/nonblocking.c IV

```
    }

    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */

    exit(0);
}
```

running the example

```
James@maker:codes$ ls -lh /etc/services
-rw-r--r--  1 root  wheel   662K Jul 31 04:32 /etc/services
James@maker:codes$ ./nonblocking < /etc/services > temp
read 500000 bytes
nwrite = 500000, errno = 0
James@maker:codes$ ls -lh  temp
-rw-r--r--  1 James  staff   488K Nov 28 15:29 temp
James@maker:codes$ ./nonblocking < /etc/services 2>stderr
#
# Network services, Internet style
# ...
```

Nonblocking I/O Example: `codes/nonblocking.c` V

understanding the result

`errno = 35` is EAGAIN

```
James@maker:codes$ head stderr
read 500000 bytes
nwrite = 999, errno = 0
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = -1, errno = 35
nwrite = 1001, errno = 0
nwrite = 1002, errno = 0
```

Nonblocking I/O Example: `codes/nonblocking.c` VI

Finding the total sum of writes

```
grep "nwrite" stderr |
    grep -v "\-1" |
    awk  -F "," '{print $1}' |
    awk -F "=" '{sum=sum+$2};END{print sum}'
```

Finding the total number of errors

```
grep "nwrite" stderr |
    grep  "\-1" | wc -l
```

Record Locking

Record Locking

What happens when two people edit the same file at the same time?

- the final state of the file corresponds to the last process that wrote the file

In some applications, a process needs to be certain that it alone is writing to a file

- To provide this capability, commercial UNIX systems provide record locking

Record Locking

Record Locking

It is the term normally used to describe the ability of a process to prevent other processes from modifying a region of a file while the first process is reading or modifying that portion of the file

A better term is byte-range locking

Record Locking: History

Early Berkeley releases supported only the `flock` function

- This function locks only entire files, not regions of a file.

Record locking was added to System V Release 3 through the `fcntl` function.

- The `lockf` function was built on top of this, providing a simplified interface.
- These functions allowed callers to lock arbitrary byte ranges in a file, ranging from the entire file down to a single byte within the file.

Record Locking: History cnt'd

POSIX.1 chose to standardize on the `fnctl` approach

System	Advisory	Mandatory	`fcntl`	`lockf`	`flock`
SUS	•		•	XSI	
FreeBSD 8.0	•		•	•	•
Linux 3.2.0	•	•	•	•	•
Mac OS X 10.6.8	•		•	•	•
Solaris 10	•	•	•	•	•

Figure: Forms of record locking supported by various UNIX systems

Prototype for the `fcntl` function

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
// Returns: depends on cmd if OK (see following), 1 on error
```

- *cmd*: F_GETLK, F_SETLK, F_SETLKW
- *flockptr*: is a pointer to an `flock` structure

```
struct flock {
   short  l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
   short  l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
   off_t  l_start;  /* offset in bytes, relative to l_whence */
   off_t  l_len;    /* length, in bytes; 0 means lock to EOF */
   pid_t  l_pid;    /* returned with F_GETLK */
};
```

`fcntl` Record Locking cnt'd

Rules in specification of the region to be locked or unlocked

- The two elements that specify the starting offset of the region are similar to the last two arguments of the `lseek` function.
  - the `l_whence` member is specified as `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If `l_len` is `0`, it means that the lock extends to the largest possible offset of the file.
- To lock the entire file, we set `l_start` and `l_whence` to point to the beginning of the file and specify a length (`l_len`) of `0`. (There are several ways to specify the beginning of the file, but most applications specify `l_start` as `0` and `l_whence` as `SEEK_SET`.)

`fcntl` Record Locking cnt'd

The basic rule

- any number of processes can have a shared read lock on a given byte
- only one process can have an exclusive write lock on a given byte.

Furthermore, if there are one or more read locks on a byte, there can't be any write locks on that byte;

if there is an exclusive write lock on a byte, there can't be any read locks on that byte.

`fcntl` Record Locking cnt'd

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Figure: Compatibility between different lock types

If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process **will replace the existing lock with the new one**.

`fcntl` Record Locking cnt'd

Three commands for the `fcntl` function

- `F_GETLK`: Determine whether the lock described by *flockptr* is blocked by some other lock.
  - If a lock exists, the information on that existing lock overwrites the information pointed to by *flockptr*.
  - If no lock exists, the structure pointed to by *flockptr* is left unchanged except for the `l_type` member, which is set to `F_UNLCK`.
- `F_SETLK`: Set the lock described by *flockptr*.
  - If the compatibility rule prevents the system from giving us the lock, `fcntl` returns immediately with errno set to either `EACCES` or `EAGAIN`.
- `F_SETLKW`: a blocking version of `F_SETLK`.
  - If the requested lock cannot be granted the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

`fcntl` Record Locking cnt'd

Be aware that testing for a lock with `F_GETLK` and then trying to obtain that lock with `F_SETLK` or `F_SETLKW` is not an atomic operation.

When setting or releasing a lock on a file, the system combines or splits adjacent areas as required.

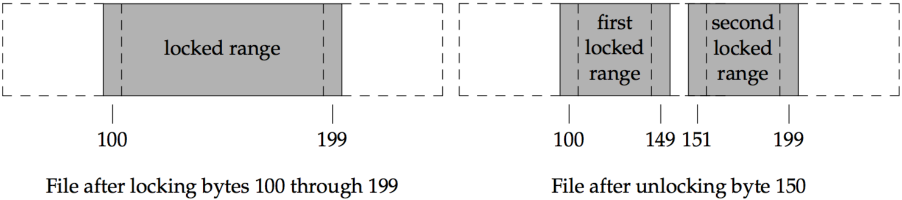

Figure: File byte-range lock diagram

Example: Requesting and Releasing a Lock

Definition of `lock_reg`

```
#include <fcntl.h>
int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;
    lock.l_type = type; /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset; /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len; /* #bytes (0 means to EOF) */
    return(fcntl(fd, cmd, &lock));
}
```

Example: Requesting and Releasing a Lock

Use following five macros

```
#define read_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
    lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))
```

Example: Requesting and Releasing a Lock I

```c
#include <fcntl.h>
#include <stdlib.h>

pid_t lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;
    lock.l_type = type;  /* F_RDLCK or F_WRLCK */
    lock.l_start = offset; /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;  /* #bytes (0 means to EOF) */

    if (fcntl(fd, F_GETLK, &lock) < 0){
       fprintf(stderr, "fcntl error");
       exit(1);
    }

    if (lock.l_type == F_UNLCK)
       return(0);  /* false, region isn't locked by another proc */
       return(lock.l_pid); /* true, return pid of lock owner */
}
```

Example: Requesting and Releasing a Lock

- If a lock exists, this function returns the process ID of the process holding the lock.
- Otherwise, the function returns 0 (false).

macros

```
#define is_read_lockable(fd, offset, whence, len) \
      (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
      (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)
```

Example: codes/deadlock.c I

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

// tellwait.c
void TELL_WAIT(void);  /* parent/child from {Sec race_conditions} */
void TELL_PARENT(pid_t);
void TELL_CHILD(pid_t);
void WAIT_PARENT(void);
void WAIT_CHILD(void);

#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

#define read_lock(fd, offset, whence, len) \
   lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
   lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
   lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
   lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
```

Example: codes/deadlock.c II

```c
#define un_lock(fd, offset, whence, len) \
  lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

int
lock_reg(int fd, int cmd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;
    lock.l_type = type; /* F_RDLCK, F_WRLCK, F_UNLCK */
    lock.l_start = offset; /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len; /* #bytes (0 means to EOF) */
    return(fcntl(fd, cmd, &lock));
}

pid_t
lock_test(int fd, int type, off_t offset, int whence, off_t len)
{
    struct flock lock;
    lock.l_type = type;  /* F_RDLCK or F_WRLCK */
    lock.l_start = offset; /* byte offset, relative to l_whence */
    lock.l_whence = whence; /* SEEK_SET, SEEK_CUR, SEEK_END */
    lock.l_len = len;  /* #bytes (0 means to EOF) */

```

```
        if (fcntl(fd, F_GETLK, &lock) < 0){
            fprintf(stderr, "fcntl error");
            exit(1);
        }

        if (lock.l_type == F_UNLCK)
            return(0);  /* false, region isn't locked by another proc */
            return(lock.l_pid); /* true, return pid of lock owner */
}

static void
lockabyte(const char *name, int fd, off_t offset)
{
    if (writew_lock(fd, offset, SEEK_SET, 1) < 0){
        fprintf(stderr, "%s: writew_lock error", name);
        exit(1);
    }
    printf("%s: got the lock, byte %lld\n", name, (long long)offset);
}

int
main(void)
{
```

```
    int  fd;
    pid_t pid;

    /*
     * Create a file and write two bytes to it.
     */
    if ((fd = creat("templock", FILE_MODE)) < 0){
       fprintf(stderr, "creat error");
       exit(1);
    }
    if (write(fd, "ab", 2) != 2){
       fprintf(stderr, "write error");
       exit(1);
    }

    TELL_WAIT();
    if ((pid = fork()) < 0) {
       fprintf(stderr, "fork error");
       exit(1);
    } else if (pid == 0) {   /* child */
       lockabyte("child", fd, 0);
       TELL_PARENT(getppid());
       WAIT_PARENT();
```

Example: codes/deadlock.c V

```
        lockabyte("child", fd, 1);
    } else {        /* parent */
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid);
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

When a deadlock is detected, the kernel has to choose one process to receive the error return.

```
James@maker:codes$ ./deadlock
parent: got the lock, byte 1
child: got the lock, byte 0
child: writew_lock errorparent: got the lock, byte 0
```

implied Inheritence and Release of Locks

Three rules govern the automatic inheritance and release of record locks

- Locks are associated with a process and a file.
  - when a process terminates, all its locks are released
  - whenever a descriptor is closed, any locks on the file referenced by that descriptor for that process are released.

Example

After the `close(fd2)`, the lock on `fd1` is released

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
fd2 = dup(fd1);
close(fd2);
```

The same would happen if we replace `dup` with `open`

```
fd1 = open(pathname, ...);
read_lock(fd1, ...);
// if pathname is same
fd2 = open(pathname, ...);
close(fd2);
```

implied Inheritence and Release of Locks cnt'd

Three rules govern the automatic inheritance and release of record locks

- Locks are associated with a process and a file.
- Locks are never inherited by the child across a `fork`.
  - The child has to call `fcntl` to obtain its own locks on any descriptors that were inherited across the `fork`.
  - locks are meant to prevent multiple processes from writing to the same file at the same time.
- Locks are inherited by a new program across an `exec`.

To understand rule 1

```
fd1 = open(pathname, ...);
write_lock(fd1, 0, SEEK_SET, 1);
if ((pid = fork()) > 0) {
    fd2 = dup(fd1);
    fd3 = open(pathname, ...);
} else if (pid == 0) {
/* parent write locks byte 0 */
/* parent */
   read_lock(fd1, 1, SEEK_SET, 1); /* child read locks byte 1 */
}
pause();
```

FreeBSD Implementation cnt'd

- `lockf` structures that are linked together from the i-node structure
- Each `lockf` structure describes one locked region for a given process
- In the parent, closing any one of fd1, fd2, or fd3 causes the parent's lock to be released. When any one of these three file descriptors is closed, the kernel goes through the linked list of locks for the corresponding i-node and releases the locks held by the calling process.

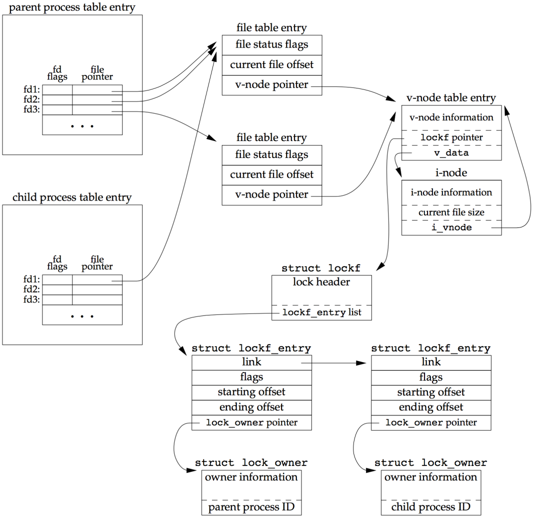

Figure: The FreeBSD data structures for record locking

We need to use caution when locking or unlocking byte ranges relative to the end of file.

- Most implementations convert an `l_whence` value of `SEEK_CUR` or `SEEK_END` into an absolute file offset, using `l_start` and the file's current position or current length.

```
writew_lock(fd, 0, SEEK_END, 0);
write(fd, buf, 1);
un_lock(fd, 0, SEEK_END);
write(fd, buf, 1);
```

The unlock operation that follows has the effect of removing the locks for future writes that append data to the file, but it leaves a lock on the last byte in the file.

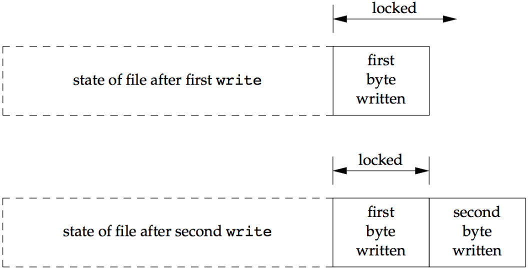

Figure: File range lock diagram

Advisory versus Mandatory Locking

Cooperating processes

If all the functions in the library handle record locking in a consistent way, then we say that any set of processes using these functions to access a database are *cooperating processes*.

It is feasible for these database access functions to use **advisory locking**, if they are the only ones being used to access the database.

Mandatory Locking or *enforcement-mode locking* causes the kernel to check `open`, `read`, `write` to verify that the calling process isn't violating a lock on the file being accessed

Asynchronous I/O

Asynchronous I/O

The cost of using Asynchronous I/O

- we complicate the design of our application by choosing to juggle multiple concurrent operations.
- We have to worry about three sources of errors for every asynchronous operation
  - one associated with the submission of the operation,
  - one associated with the result of the operation itself,
  - and one associated with the functions used to determine the status of the asynchronous operations.
- The interfaces themselves involve a lot of extra setup and processing rules compared to their conventional counterparts
- Recovering from errors can be difficult.

POSIX Asynchronous I/O

The POSIX asynchronous I/O interfaces give us a consistent way to perform asynchronous I/O, regardless of the type of file. The

asynchronous I/O interfaces use AIO control blocks to describe I/O operations.

```
struct aiocb {
  int                aio_fildes;       /* file descriptor */
  off_t              aio_offset;       /* file offset fro I/O */
  volatile void     *aio_buf;          /* buffer for I/O */
 size_t            aio_nbytes;     /* number of bytes to transfer */
  int                aio_reqprio;      /* priority  */
  struct sigevent aio_sigevent;        /* Signal information  */
  int               aio_lio_opcode;   /* operation for list I/O */
};
```

POSIX Asynchronous I/O cnt'd

The `aio_sigevent` field controls how the application is notified about the completion of the I/O event.

```
struct sigevent {
     int    sigev_notify;             /* nofity type */
     int    sigev_signo;              /* signal number */
     union sigval  sigev value;       /* notify argument */
     void (*sigev_notify_function)(union sigval); /* notify function */
     pthread_attr_t *sigev_notify_attributes;     /* notify attrs */
};
```

`sigev_notify` field controls the type of notification

1. `SIGEV_NONE`: The process is not notified when the asynchronous I/O request completes.
2. `SIGEV_SIGNAL`: The signal specified by the `sigev_signo` field is generated when the asynchronous I/O request completes.
3. `SIGEV_THREAD`: The function specified by the `sigev_notify_function` field is called when the asynchronous I/O request completes. The function is executed in a separate thread

To perform asynchronous I/O, we need to initialize an AIO control block

```
#include <aio.h>
int aio_read(struct aiocb *aiocb); int aio_write(struct aiocb *aiocb);
// Both return: 0 if OK, 1 on error
```

When these functions return success,

- the asynchronous I/O request has been queued for processing by the operating system.

The return value bears no relation to the result of the actual I/O operation

POSIX Asynchronous I/O: Persistency

To force all pending asynchronous writes to persistent storage without waiting

```
#include <aio.h>
int aio_fsync(int op, struct aiocb *aiocb);
// Returns: 0 if OK, 1 on error
```

The `aio_fildes` field in the AIO control block indicates the file whose asynchronous writes are synched. If the op argument is set to

- `O_DSYNC`: the operation behaves like a call to `fdatasync`.
- `O_SYNC`: the operation behaves like a call to `fsync`.

Returns when the synch is scheduled. The data won't be persistent until the asynchronous synch completes.

POSIX Asynchronous I/O: Status

To determine the completion status of an asynchronous read, write, or synch operation

```
#include <aio.h>
int aio_error(const struct aiocb *aiocb);
Returns: 0, -1, EINPROGRESS, others
```

Four return types

- `0`: The asynchronous operation completed successfully.
- `-1`: The call to `aio_error` failed
- `EINPROGRESS`: The asynchronous read, write, or synch is still pending
- *anything esle*: error code corresponding to the failed asynchronous operation.

POSIX Asynchronous I/O: Success

If the asynchronous operation succeeded, we can call the `aio_return` function to get the asynchronous operation's return value

```
#include <aio.h>
ssize_t aio_return(const struct aiocb *aiocb);
// Returns: ...
```

Until the asynchronous operation completes, we need to be careful to avoid calling the aio_return function

- Results undefined until the operation completes
- call `ait_return` only one time per asynchronous I/O operation

POSIX Asynchronous I/O: suspend

When we have completed the processing and find that we still have asynchronous operations outstanding, we can call the `aio_suspend` function to block until an operation completes.

```
#include <aio.h>
int aio_suspend(const struct aiocb *const list[], int nent,
               const struct timespec *timeout);
// Returns: 0 if OK, 1 on error
```

Three things cause `aio_suspend` to return

- `-1`: Interrupted by a signal, `errno` is set to `EINTR`
- `-1`: *timeout* argument expires without any of the I/O operations completing, `errno` is set to `EAGAIN`
- `0`: if any of the I/O operations complete

list is a pointer to an array of AIO control block

nent indicates the number of entries in the array

POSIX Asynchronous I/O: Cancel

When we have pending asynchronous I/O operations that we no longer want to complete, we can attempt to cancel them with the `aio_cancel` function.

```
#include <aio.h>
int aio_cancel(int fd, /* file descriptor */
               struct aiocb *aiocb); /* NULL: all oustanding AIO */
// Returns: (see following)
```

Four return values

- `AIO_ALLDONE`: All doen before attempt to cancel
- `AIO_CANCELED`: All are canceled
- `AIO_NOTCANCELED`: At least one of the requested opeartion could not be canceled
- `-1`: Failed. Error code stored in `errno`

POSIX Asynchronous I/O: set of I/O requests

The `lio_listio` function submits a set of I/O requests described by a list of AIO control blocks.

```
#include <aio.h>
int lio_listio(int mode, struct aiocb *restrict const list[restrict],
               int nent, struct sigevent *restrict sigev);
// Returns: 0 if OK, 1 on error
```

- `mode`
  - `LIO_WAIT`: won't return until all I/Os in the list are complete (`sigev` is ingored)
  - `LIO_NOWAIT`: returns as soon as the I/O requests are queued. Process is notified asynchronously when all of the I/O operations complete
- `sigev`: can be set to NULL if don't want to be notified
- `list`: list of AIO control blocks specifying the I/O operations to perform
- `nent`: number of elements in the array

Example: codes/rot13a.c I

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

#define BSZ 4096
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

unsigned char buf[BSZ];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= ’n’)
            c -= 13;
        else if (c >= ’a’)
            c += 13;
        else if (c >= ’N’)
            c -= 13;
        else
```

Example: codes/rot13a.c II

```
         c += 13;
   }
   return(c);
}

int
main(int argc, char* argv[])
{
   int ifd, ofd, i, n, nw;

   if (argc != 3){
      fprintf(stderr, "usage: rot13 infile outfile");
      exit(1);
   }
   if ((ifd = open(argv[1], O_RDONLY)) < 0){
      fprintf(stderr, "can't open %s", argv[1]);
      exit(1);
   }
   if ((ofd = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, FILE_MODE)) < 0){
      fprintf(stderr, "can't create %s", argv[2]);
      exit(1);
   }

```

Example: codes/rot13a.c III

```c
   while ((n = read(ifd, buf, BSZ)) > 0) {
      for (i = 0; i < n; i++)
         buf[i] = translate(buf[i]);
      if ((nw = write(ofd, buf, n)) != n) {
         if (nw < 0){
            fprintf(stderr, "write failed");
            exit(1);
         }
         else {
            fprintf(stderr, "short write (%d/%d)", nw, n);
            exit(1);
         }
      }
   }

   fsync(ofd);
   exit(0);
}
```

Example: codes/rot13c2.c I

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>
#include <sys/stat.h>

#define BSZ 4096
#define NBUF 8
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

enum rwop {
    UNUSED = 0,
    READ_PENDING = 1,
    WRITE_PENDING = 2
};

struct buf {
    enum rwop op;
```

```c
    int last;
    struct aiocb aiocb;
    unsigned char data[BSZ];
};

struct buf bufs[NBUF];

unsigned char
translate(unsigned char c)
{
    if (isalpha(c)) {
        if (c >= 'n')
            c -= 13;
        else if (c >= 'a')
            c += 13;
        else if (c >= 'N')
            c -= 13;
        else
            c += 13;
    }
    return(c);
}
```

```c
int
main(int argc, char* argv[])
{
    int     ifd, ofd, i, j, n, err, numop;
    struct stat   sbuf;
    const struct aiocb *aiolist[NBUF];
    off_t    off = 0;

    if (argc != 3){
        fprintf(stderr, "usage: rot13 infile outfile");
        exit(1);
    }
    if ((ifd = open(argv[1], O_RDONLY)) < 0){
        fprintf(stderr, "can't open %s", argv[1]);
        exit(1);
    }
    if ((ofd = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0){
        fprintf(stderr, "can't create %s", argv[2]);
        exit(1);
    }
    if (fstat(ifd, &sbuf) < 0){
```

```
        fprintf(stderr, "fstat failed");
        exit(1);
    }

    /* initialize the buffers */
    for (i = 0; i < NBUF; i++) {
        bufs[i].op = UNUSED;
        bufs[i].aiocb.aio_buf = bufs[i].data;
        bufs[i].aiocb.aio_sigevent.sigev_notify = SIGEV_NONE;
        aiolist[i] = NULL;
    }

    numop = 0;
    for (;;) {
        for (i = 0; i < NBUF; i++) {
            switch (bufs[i].op) {
            case UNUSED:
                /*
                 * Read from the input file if more data
                 * remains unread.
                 */
                if (off < sbuf.st_size) {
```

```
                bufs[i].op = READ_PENDING;
                bufs[i].aiocb.aio_fildes = ifd;
                bufs[i].aiocb.aio_offset = off;
                off += BSZ;
                if (off >= sbuf.st_size)
                    bufs[i].last = 1;
                bufs[i].aiocb.aio_nbytes = BSZ;
                if (aio_read(&bufs[i].aiocb) < 0){
                    fprintf(stderr, "aio_read failed");
                    exit(1);
                }
                aiolist[i] = &bufs[i].aiocb;
                numop++;
            }
            break;

        case READ_PENDING:
            if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
                continue;
            if (err != 0) {
                if (err == -1){
                    fprintf(stderr, "aio_error failed");
```

```
                exit(1);
            }
            else{
                fprintf(stderr, "read failed");
                exit(1);
            }
        }

        /*
         * A read is complete; translate the buffer
         * and write it.
         */
        if ((n = aio_return(&bufs[i].aiocb)) < 0){
            fprintf(stderr, "aio_return failed");
            exit(1);
        }
        if (n != BSZ && !bufs[i].last){
            fprintf(stderr, "short read (%d/%d)", n, BSZ);
            exit(1);
        }
        for (j = 0; j < n; j++)
```

Example: codes/rot13c2.c VII

```
                bufs[i].data[j] = translate(bufs[i].data[j]);
            bufs[i].op = WRITE_PENDING;
            bufs[i].aiocb.aio_fildes = ofd;
            bufs[i].aiocb.aio_nbytes = n;
            if (aio_write(&bufs[i].aiocb) < 0){
                fprintf(stderr, "aio_write failed");
                exit(1);
            }
            /* retain our spot in aiolist */
            break;

        case WRITE_PENDING:
            if ((err = aio_error(&bufs[i].aiocb)) == EINPROGRESS)
                continue;
            if (err != 0) {
                if (err == -1){
                    fprintf(stderr, "aio_error failed");
                    exit(1);
                }
                else{
                    fprintf(stderr, "write failed");
```

```
                    exit(1);
                }
            }

            /*
             * A write is complete; mark the buffer as unused.
             */
            if ((n = aio_return(&bufs[i].aiocb)) < 0){
                fprintf(stderr, "aio_return failed");
                exit(1);
            }
            if (n != bufs[i].aiocb.aio_nbytes){
                fprintf(stderr, "short write (%d/%d)", n, BSZ);
                exit(1);
            }
            aiolist[i] = NULL;
            bufs[i].op = UNUSED;
            numop--;
            break;
        }
    }
```

Example: codes/rot13c2.c IX

```
        if (numop == 0) {
            if (off >= sbuf.st_size)
                break;
        } else {
            if (aio_suspend(aiolist, NBUF, NULL) < 0){
                fprintf(stderr, "aio_suspend failed");
                exit(1);
            }
        }
    }

    bufs[0].aiocb.aio_fildes = ofd;
    if (aio_fsync(O_SYNC, &bufs[0].aiocb) < 0){
        fprintf(stderr, "aio_fsync failed");
        exit(1);
    }
    exit(0);
}
```

`readv` AND `writev` FUNCTIONS

readv and writev Functions

The readv and writev functions let us read into and write from multiple noncontiguous buffers in a single function call. These operations are called *scatter read* and *gather write*.

```
#include <sys/uio.h>
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
// Both return: number of bytes read or written, 1 on error
```

iov is a pointer to an array of iovec structures

```
struct iovec {
       void   *iov_base;  /* starting address of buffer */
       size_t  iov_len;   /* size of buffer */
};
```

The number of elements in the iov array is specified iovcnt

readv and writev Functions

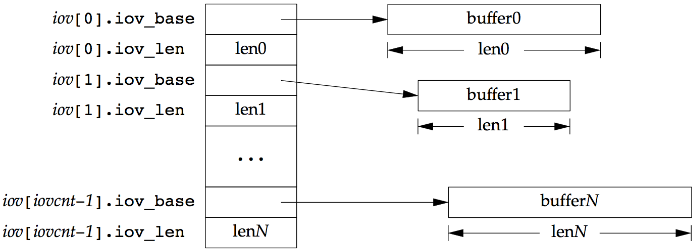

Figure: the iovec structure for readv and writev

- The `writev` function gathers the output data from the buffers in order: `iov[0]`, `iov[1]`, through `iov[iovcnt1]`
  - `writev` returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.
- The `readv` function scatters the data into the buffers in order, always filling one buffer before proceeding to the next.
  - `readv` returns the total number of bytes that were read.

`readn` AND `writen` FUNCTIONS

`readn` and `writen` Functions

Pipes, FIFOs, and some devices—notably terminals and networks—have the following two properties.

1. A `read` operation may return less than asked for, even though we have not encountered the end of file.
   - This is not an error, and we should simply continue reading from the device.
2. A `write` operation can return less than we specified. This may be caused by kernel output buffers becoming full.
   - it's not an error, and we should continue writing the remainder of the data.

   Generally, when we read from or write to a pipe, network device, or terminal, we need to take these characteristics into consideration.

`readn` and `writen` Functions

We can use the `readn` and `writen` functions to read and write `N` bytes of data, respectively, letting these functions handle a return value that's possibly less than requested.

- These two functions simply call read or write as many times as required to read or write the entire N bytes of data.

```
ssize_t readn(int fd, void *buf, size_t nbytes);
ssize_t writen(int fd, void *buf, size_t nbytes);
// Both return: number of bytes read or written, 1 on error
```

readn Functions

```c
ssize_t /* Read "n" bytes from a descriptor */
readn(int fd, void *ptr, size_t n)
{
   size_t  nleft;
   ssize_t  nread;

   nleft = n;
   while (nleft > 0) {
      if ((nread = read(fd, ptr, nleft)) < 0) {
         if (nleft == n)
            return(-1); /* error, return -1 */
         else
            break; /* error, return amount read so far */
      } else if (nread == 0) {
         break; /* EOF */
      }
      nleft -= nread;
      ptr += nread;
   }
   return(n - nleft); /* return >= 0 */
}
```

writen Functions

```c
ssize_t /* Write "n" bytes to a descriptor */
writen(int fd, const void *ptr, size_t n)
{
   size_t  nleft;
   ssize_t  nwritten;

   nleft = n;
   while (nleft > 0) {
      if ((nwritten = write(fd, ptr, nleft)) < 0) {
         if (nleft == n)
            return(-1); /* error, return -1 */
         else
            break; /* error, return amount written so far */
      } else if (nwritten == 0) {
         break;
      }
      nleft -= nwritten;
      ptr += nwritten;
   }
   return(n - nleft); /* return >= 0 */
}
```

Memory-Mapped I/O

Memory-Mapped I/O

Memory-mapped I/O lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read` or `write`.

Memory-Mapped I/O

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flag, int fd, off_t off );
// Returns: starting address of mapped region if OK, MAP_FAILED on error
```

- `addr`: specify the address where we want the mapped region to start (Normally set to 0 to let system chooose)
- `fd`: file descriptor specifying the file that is to be mapped (file must be opened before mapping)
- `len`: number of bytes to map
- `off`: starting offset in the file of the bytes to map
- `prot`: READ, WRITE, and EXEC can be ORed
    - `PROT_READ`: Region can be read
    - `PROT_WRITE`: Region can be written
    - `PROT_EXEC`: Region can be executed
    - `PROT_NONE`: Region cannot be accessed

Memory-Mapped I/O

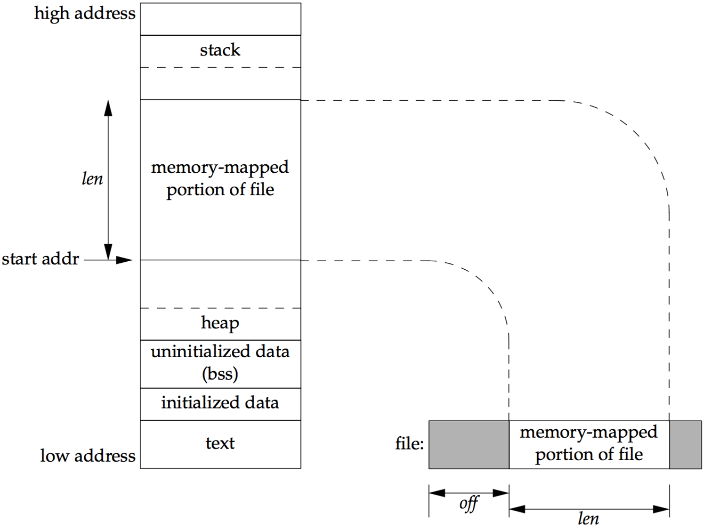

Figure: Example of a memory-mapped file

Memory-Mapped I/O

`flag` argument affects various attributes of the mapped region

- `MAP_FIXED`: The return value must equal `addr`. Use of this flag is discouraged (Portability issue)
- `MAP_SHARED`: store operations into the mapped region by this process
- `MAP_PRIVATE`: store operation into the mapped region cause a private copy of the mapped file to be created. All successive references to the mapped region then reference the copy

Memory-Mapped I/O

The value of `off` and the value of addr are usually required to be multiples of the system's virtual memory page size What happens if

the length of the mapped region isn't a multiple of the page size?

- File size 12 bytes
- Page size is 512 bytes
- system provides a mapped region of 512 bytes
- final 500 bytes of the this region are set to 0
- any changes to the final 500 bytes are not reflected to the file

Memory-Mapped I/O cnt'd

- A memory-mapped region is inherited by a child across a `fork`
- But, is not inherited by the new program across an `exec`.

Permission can be changed by calling `mprotect`

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
// Returns: 0 if OK, 1 on error
```

Memory-Mapped I/O cnt'd

When we modify pages that we've mapped into our address space using the `MAP_SHARED` flag, the changes aren't written back to the file immediately. kernel daemons decide when dirty pages are written

back based on

- system load
- configuration parameters meant to limit data loss in the event ofa system failure

When the changes are written back, they are written in units of pages.

Memory-Mapped I/O cnt'd

If the pages in a shared mapping have been modified, we can call `msync` to flush the changes to the file that backs the mapping.

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
// Returns: 0 if OK, 1 on error
```

Two values for `flags`

- `MS_SYNC`: wait for the writes to complete before returning
- `MS_ASYNC`: simply schedule the pages to be written

Memory-Mapped I/O cnt'd

A memory-mapped region is automatically unmapped when the process terminates or we can unmap a region directly by calling the `munmap` function.

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
// Returns: 0 if OK, 1 on error
```

Example: CP with MMAP codes/mcopy2.c I

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>

#define COPYINCR (1024*1024*1024) /* 1 GB */
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int
main(int argc, char *argv[])
{
    int     fdin, fdout;
    void    *src, *dst;
    size_t  copysz;
    struct stat sbuf;
```

```
    off_t  fsz = 0;

    if (argc != 3){
        fprintf(stderr, "usage: %s <fromfile> <tofile>", argv[0]);
        exit(1);
    }

    if ((fdin = open(argv[1], O_RDONLY)) < 0){
        fprintf(stderr, "can't open %s for reading", argv[1]);
        exit(1);
    }

    if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,
       FILE_MODE)) < 0){
        fprintf(stderr, "can't creat %s for writing", argv[2]);
        exit(1);
    }

    if (fstat(fdin, &sbuf) < 0){   /* need size of input file */
```

```c
        fprintf(stderr, "fstat error");
        exit(1);
    }

    if (ftruncate(fdout, sbuf.st_size) < 0){ /* set output file size */
        fprintf(stderr, "ftruncate error");
        exit(1);
    }

    while (fsz < sbuf.st_size) {
        if ((sbuf.st_size - fsz) > COPYINCR)
            copysz = COPYINCR;
        else
            copysz = sbuf.st_size - fsz;

        if ((src = mmap(0, copysz, PROT_READ, MAP_SHARED,
           fdin, fsz)) == MAP_FAILED){
            fprintf(stderr, "mmap error for input");
            exit(1);
```

Example: CP with MMAP codes/mcopy2.c IV

```
        }
        if ((dst = mmap(0, copysz, PROT_READ | PROT_WRITE,
          MAP_SHARED, fdout, fsz)) == MAP_FAILED){
            fprintf(stderr, "mmap error for output");
            exit(1);
        }

        memcpy(dst, src, copysz); /* does the file copy */
        munmap(src, copysz);
        munmap(dst, copysz);
        fsz += copysz;
    }
    exit(0);
}
```

```
dd if=/dev/urandom of=testfile count=1000000
time ./mcopy2 testfile mcopyout
time cp testfile mcopyout
```