CMPS 105
Systems Programming
Prof. Darrell Long
E2.371
darrell@ucsc.edu
Chapter 14: Advanced I/O
Advanced I/O

- Advanced I/O
  - Non-blocking
  - Record locking
  - Multiplexing
  - Asynchronous
  - Memory-mapped
Blocking I/O

- Slow system calls can block forever
  - Reads can block forever if data is not present
  - Writes can block forever if the data isn’t accepted
  - Opens can block until a certain condition occurs
  - Read/write of files that with record locking enabled
  - Certain ioctl() operations
  - etc…
Nonblocking I/O

- Nonblocking I/O will return with an error if the operation could not be completed

- Two ways to specify nonblocking I/O
  - Specify the O_NONBLOCK flag
  - Call fcntl() with O_NONBLOCK
    - For already opened descriptors
Record Locking

- Record locking: a process preventing other processes from modifying a region of a file while the first process is reading.

- The kernel doesn’t have a notion of records in a file.
  - Better name is byte-range locking.
Record locking history

- Databases require record locking, early UNIX didn’t have it
- flock() was added to BSD, it locked entire files
- SVR3 added it using fcntl()
  - lockf() was an interface that used fcntl()
- POSIX chose the fcntl() approach
```c
#include "apue.h"
#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));
}
```

**Figure 14.5** Function to lock or unlock a region of a file
Record locking: `fnctl()` again

- `fnctl()` – file control
  - `int fnctl(int fd, int cmd, ...);`

```c
struct flock {
    short l_type;    /* Type of lock */
    short l_whence;  /* SEEK_SET, SEEK_CUR, SEEK_END*/
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock */
    ...
};
```
## Record locking compatibility

<table>
<thead>
<tr>
<th>Region currently has</th>
<th>Request for</th>
<th>read lock</th>
<th>write lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>no locks</td>
<td>OK</td>
<td>OK</td>
<td></td>
</tr>
<tr>
<td>one or more read locks</td>
<td>OK</td>
<td>denied</td>
<td></td>
</tr>
<tr>
<td>one write lock</td>
<td>denied</td>
<td>denied</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 14.3** Compatibility between different lock types
Record locking byte-range

File after locking bytes 100 through 199

File after unlocking byte 150

Figure 14.4 File byte-range lock diagram
```c
#include <apue.h>
#include <fcntl.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)
        err_sys("fcntl error");

    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 */
}
```

**Figure 14.6** Function to test for a locking condition
Inheritance and release of locks

- Three rules for inheritance and release of record locks
  - Locks are associated with a process and a file
    - When a process terminates locks are released
    - When a descriptor is closed locks are released
  - Locks are never inherited by the child across fork()
    - Child has to can fcntl() to obtain its own locks
    - Why would a child inheriting a lock be bad?
  - Locks are inherited by a new program across and exec()
Figure 14.8 The FreeBSD data structures for record locking
Lock entire file

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

int
lockfile(int fd)
{
    struct flock fl;
    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return(fcntl(fd, F_SETLK, &fl));
}
```

Figure 14.9  Place a write lock on an entire file
Locks at EOF

- Be careful when locking or unlocking ranges relative to the end of the file

**Figure 14.10** File range lock diagram
Advisory vs mandatory locking

- Cooperating processes all handle record locking in the same manner (example: database access routines)
  - This works as long as other processes don’t modify the file
- Mandatory locking causes the kernel to check if the calling process is violating a lock of the file
I/O Multiplexing

- Protocols like telnet need to read and write to two descriptors for full duplex communication
  - Can fork(), and parent/child handle read/write separately
    - Communication between parent child may be complicated
  - Can using polling
    - CPU intensive
  - Asynchronous I/O – kernel sends notification when fd is ready
    - Portability issues
    - Hard to differentiate which fd sent the signal
- I/O Multiplexing…
Figure 14.13  Overview of telnet program

Figure 14.14  The telnet program using two processes
What is I/O Multiplexing?

- **I/O Multiplexing**
  - Build a list of file descriptors
  - Call function that waits for descriptors to become ready

- **select()** – Check file descriptor list for available fds
  - `int select(int maxfdp1, fd_set* restrict readfds, fd_set* restrict writefds, fd_set* restrict exceptfds, struct timeval* restrict tvptr);`
  - `select()` returns:
    - `-1` on error
    - `0` if no fds are ready
    - `#` of fds ready (sum of read/write/other)
### Figure 14.15
Specifying the read, write, and exception descriptors for `select`
fd_set functions: FD_ISSET(), FD_CLR(), FD_SET(), FD_ZERO()

- **FD_ISSET()** – test if bit in fd set is turned on
  - ```c
  int FD_ISSET(int fd, fd_set *set);
  ```

- **FD_CLR()** – clear single bit in fd set
  - ```c
  void FD_CLR(int fd, fd_set *set);
  ```

- **FD_SET()** – set single bit in fd set
  - ```c
  void FD_SET(int fd, fd_set *set);
  ```

- **FD_ZERO()** – zero out fd set
  - ```c
  void FD_ZERO(fd_set *set);
  ```
**Figure 14.16** Example descriptor sets for `select`
pselect() – similar to select with a higher resolution time variable and mask option

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout, const sigset_t *sigmask);
poll() – similar to select() different implementation

- int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);

struct pollfd {
    int fd;   /* file descriptor to check */
    short events;  /* events of interest on fd */
    short revents; /* events that occurred on fd */
};
## Event flags

<table>
<thead>
<tr>
<th>Name</th>
<th>Input to events?</th>
<th>Result from revents?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POLLIN</td>
<td>•</td>
<td>•</td>
<td>Data other than high priority data can be read without blocking (equivalent to POLLRDNORM</td>
</tr>
<tr>
<td>POLLRDNORM</td>
<td>•</td>
<td>•</td>
<td>Normal data can be read without blocking.</td>
</tr>
<tr>
<td>POLLRDBAND</td>
<td>•</td>
<td>•</td>
<td>Priority data can be read without blocking.</td>
</tr>
<tr>
<td>POLLPRI</td>
<td>•</td>
<td>•</td>
<td>High-priority data can be read without blocking.</td>
</tr>
<tr>
<td>POLLOUT</td>
<td>•</td>
<td>•</td>
<td>Normal data can be written without blocking.</td>
</tr>
<tr>
<td>POLLWRNORM</td>
<td>•</td>
<td>•</td>
<td>Same as POLLOUT.</td>
</tr>
<tr>
<td>POLLWRBAND</td>
<td>•</td>
<td>•</td>
<td>Priority data can be written without blocking.</td>
</tr>
<tr>
<td>POLLERR</td>
<td>•</td>
<td>•</td>
<td>An error has occurred.</td>
</tr>
<tr>
<td>POLLHUP</td>
<td>•</td>
<td>•</td>
<td>A hangup has occurred.</td>
</tr>
<tr>
<td>POLLNVAL</td>
<td>•</td>
<td>•</td>
<td>The descriptor does not reference an open file.</td>
</tr>
</tbody>
</table>

**Figure 14.17** The `events` and `revents` flags for `poll`
Asynchronous I/O

- Asynchronous I/O
  - Use signals to notify when an fd is open
  - Usually multithreaded

- Problems with asynchronous I/O
  - Portability
  - Three sources of errors for each async operation
    - submission of the operation
    - result of the operation
    - status of operation
  - Recovering from errors is difficult
  - Interfaces involve extra setup and processing rules
Variations of Asynchronous I/O

- **System V: STREAMS devices and STREAMS pipe**
  - Signal: SIGPOLL
  - call ioctl()

- **BSD: Combination of two signals**
  - SIGURG – signal for arrival out of band data on network
  - SIGIO – general async I/O
    - Establish a signal handler
    - Call fcntl with F_SETOWN (process/process group ID)
    - Call fcntl with F_SETFL to enable asynchronous I/O (O_ASYNC flag)
POSIX Asynchronous I/O

- AIO control blocks describe I/O operations

```
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 for list I/O */
    /* Various implementation-internal fields not shown */
};
```
POSIX Asynchronous I/O: aio_read() and aio_write()

- aio_read() - asynchronous read
  - int aio_read(struct aiocb *aiocbp);

- aio_write() - asynchronous write
  - int aio_write(struct aiocb *aiocbp);

- Successful returns indicate the asynchronous I/O request has been queued for processing by the OS
  - Be aware that this means that read/write errors can still occur even on successful return
POSIX Asynchronous I/O: aio_fsync() and aio_error()

- **aio_fsync()** - asynchronous file synchronization
  - int aio_fsync(int op, struct aiocb *aiocbp);

- **aio_error()** - get error status of asynchronous I/O operation
  - int aio_error(const struct aiocb *aiocbp);
  - Return values:
    - 0: success
    - -1: aio_error() failed
    - EINPROGRESS: action is pending
    - other: error code corresponding to failed operation
AI/O: \texttt{aio\_return()}, \texttt{aio\_suspend()}, and \texttt{aio\_cancel()}

- \texttt{aio\_return()} - get return status of asynchronous I/O operation
  
  \begin{verbatim}
  ssize_t aio_return(struct aiocb *aiocbp);
  \end{verbatim}

- \texttt{aio\_suspend()} - wait for asynchronous I/O operation or timeout
  
  \begin{verbatim}
  int aio_suspend(const struct aiocb * const aiocb_list[], int nitems, const struct timespec *timeout);
  \end{verbatim}

- \texttt{aio\_cancel()} - cancel an outstanding asynchronous I/O request
  
  \begin{verbatim}
  int aio_cancel(int fd, struct aiocb *aiocbp);
  \end{verbatim}
AI/O: lio_listio()

- lio_listio() - initiate a list of I/O requests
  - int lio_listio(int mode, struct aiocb *const aiocb_list[], int nitems, struct sigevent *sevp);

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Minimum acceptable value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIO_LISTIO_MAX</td>
<td>maximum number of I/O operations in a single list I/O call</td>
<td>_POSIX_AIO_LISTIO_MAX (2)</td>
</tr>
<tr>
<td>AIO_MAX</td>
<td>maximum number of outstanding asynchronous I/O operations</td>
<td>_POSIX_AIO_MAX (1)</td>
</tr>
<tr>
<td>AIO_PRIO_DELTA_MAX</td>
<td>maximum amount by which a process can decrease its asynchronous I/O priority level</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 14.19** POSIX.1 runtime invariant values for asynchronous I/O
readv() and writev()

- **readv()** – read data into multiple buffer
  - ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
  - Read iovcnt buffers from fd into buffers iov

- **writev()** - write data into multiple buffer
  - ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
  - Write iovcnt buffers described by iov into fd

```c
struct iovec {
    void* iov_base; /* starting address of buffer */
    size_t iov_len; /* size of buffer */
};
```
Figure 14.22 The iovec structure for readv and writev
- For pipes, FIFOs, terminals, networks, and other devices
  - Read operations may return less than asked for
  - Write operations can return less than we asked for

- **readn()** – read n bytes from a descriptor
  - `ssize_t readn(int filedes, void *buff, size_t nbytes);`

- **writen()** – write n bytes to a descriptor
  - `ssize_t writen(int filedes, const void *buff, size_t nbytes);`
Memory Mapped I/O: mmap() and munmap()

- Memory mapped I/O lets us map a file on disk into a buffer in memory

- mmap() - map files or devices into memory
  - void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

- munmap() - unmap files or devices into memory
  - int munmap(void *addr, size_t length);
Figure 14.26  Example of a memory-mapped file
Memory Mapped I/O: mprotect() and msync()

- mprotect() - set protection on a region of memory
  - int mprotect(const void *addr, size_t len, int prot);

- msync() - synchronize a file with a memory map
  - int msync(void *addr, size_t length, int flags);
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define COPYINCR (1024*1024*1024) /* 1 GB */

int
main(int argc, char *argv[])
{
    int fdin, fdout;
    void *src, *dst;
    size_t copyysz;
    struct stat sbuf;
    off_t fsz = 0;

    if (argc != 3)
        err_quit("usage: %s <fromfile> <tofile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

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

    if (fstat(fdin, &sbuf) < 0) /* need size of input file */
        err_sys("fstat error");

    if (ftruncate(fdout, sbuf.st_size) < 0) /* set output file size */
        err_sys("ftruncate error");

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

        if ((src = mmap(0, copyysz, PROT_READ, MAP_SHARED, fdin, fsz)) == MAP_FAILED)
            err_sys("mmap error for input");

        if ((dst = mmap(0, copyysz, PROT_READ | PROT_WRITE, MAP_SHARED, fdout, fsz)) == MAP_FAILED)
            err Sys("mmap error for output");

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