


Chapter 14

Advanced I/O



Cheng-Hsin Hsu

*National Tsing Hua University
Department of Computer Science*

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Outline

- Non-blocking I/O
- Record Locking
- I/O Multiplexing
- Asynchronous I/O
- readv and writev functions
- readn and writen functions
- Memory-mapped I/O

Non-Blocking I/O

- Many I/O operations may be slow or blocked forever
 - read from pipe, terminal devices, and network devices
 - write to a pipe (if buffer is full) and network devices (if flow control is enabled)
 - open a pipe for write, but no reader is available
 - read or write of files that have mandatory record locking enabled
 - ioctl operations
 - Some other IPC functions
- Non-blocking I/O ensures that an I/O operation not be blocked
 - If an operation cannot be completed, an error is returned
 - It may return partial results

Enable Non-Blocking I/O

- Pass O_NONBLOCK flag when opening a file (use the open system call)
- Use fcntl to turn on O_NONBLOCK for an opened file
- A sample function to set fcntl flag: set_fl()

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

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

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

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

A Non-Blocking I/O Example

```
char buf[500000];
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; }
    }
    clr_fl(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */
    exit(0);
}
```

A Non-Blocking I/O Example

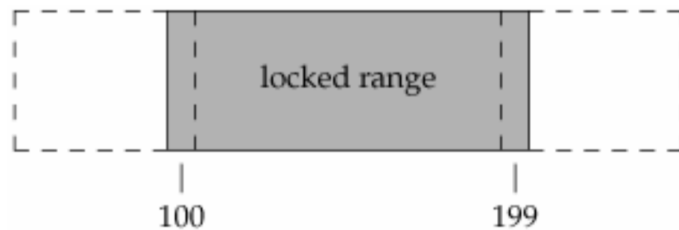
```
$ dd if=/dev/urandom bs=1k count=500 | hexdump -C > /tmp/data
$ ls -la /tmp/data
-rw-rw-r-- 1 chuang  chuang  2528009 Apr 29 18:17 /tmp/data
$ ./a.out < /tmp/data > /tmp/output
read 500000 bytes
nwrite = 500000, errno = 0
$ ./a.out < /tmp/data | cat > /dev/null
read 500000 bytes
nwrite = 65536, errno = 0      ; this depends on how cat reads
nwrite = -1, errno = 11
nwrite = -1, errno = 11
nwrite = 65536, errno = 0
nwrite = 65536, errno = 0
nwrite = 65536, errno = 0
nwrite = 65536, errno = 0
nwrite = 65536, errno = 0
nwrite = 65536, errno = 0
nwrite = 41248, errno = 0
```

On Linux:

```
#define EAGAIN      11      /* Try again */
```

Record Locking

- What happens when two people edit the same file at the same time?
 - Usually only contains the content from the last writer
- Sometimes, we may have to ensure that there is only one writer
 - For example, database systems
- Record locking
 - To prevent other processes from modifying a region of a file
 - The record locking is actually “byte-range” locking
 - An entire file can be locked
 - Lock and unlock operations



File after locking bytes 100 through 199



File after unlocking byte 150

Record Locking with fcntl

```
int fcntl(int filedes, int cmd, ... /* struct flock *flockptr */ );
```

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

- Lock operations: reader lock, writer lock, and unlock
 - Reader lock can be shared
 - Writer lock is exclusive
- Lock the entire file
 - `l_start = 0, l_whence = SEEK_SET, l_len = 0`

Record Locking with fcntl (Cont'd)

- To obtain a read lock, the descriptor must be open for reading
- To obtain a write lock, the descriptor must be open for writing
- fcntl commands
 - F_GETLK: Determine whether the lock described by flockptr is blocked by some other lock – check lock type (l_type)
 - F_SETLK: Set the lock described by flockptr – It has to follow the compatibility rules (see next page)
 - F_SETLKW: This command is a blocking version of F_SETLK. The process wakes up either when the lock becomes available or when interrupted by a signal.

Compatibility between Different Lock Types

		Request for ...	
		reader lock	writer lock
Region currently has ...	No locks	OK	OK
	One or more reader locks	OK	denied
	One writer locks	denied	denied

Compatibility between Different Lock Types (Cont'd)

- The compatibility rule applies to lock requests made from **different processes**
 - Not to multiple lock requests made by a single process
- If a process has an existing lock on a range of a file ...
 - Subsequent attempts to place a lock on the same range by the same process will **replace the existing lock** with the new one
- For example
 - Suppose a process already has a write lock on bytes 16–32 of a file
 - If it attempts to place a read lock on bytes 16–32, the request will succeed
 - **The original write lock will be replaced by the new read lock**

Sample Functions and Macros to Lock and Unlock a Region

```
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));
}
```

```
#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))
```

Sample Functions and Macros to Test Locking Condition

```
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");
                                /* terminate the process with an error message */
    if (lock.l_type == F_UNLCK)
        return(0);             /* false, region is not locked by another proc */
    return(lock.l_pid);       /* true, return pid of lock owner */
}

#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)
```

Record Locking and Deadlock

- Deadlock occurs when two processes are each waiting for a resource that the other has locked
 - Suppose we have two processes Pa and Pb
 - Pa locks region #1 and then region #2
 - Pb locks region #2 and then region #1
- A sample function to lock one byte

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

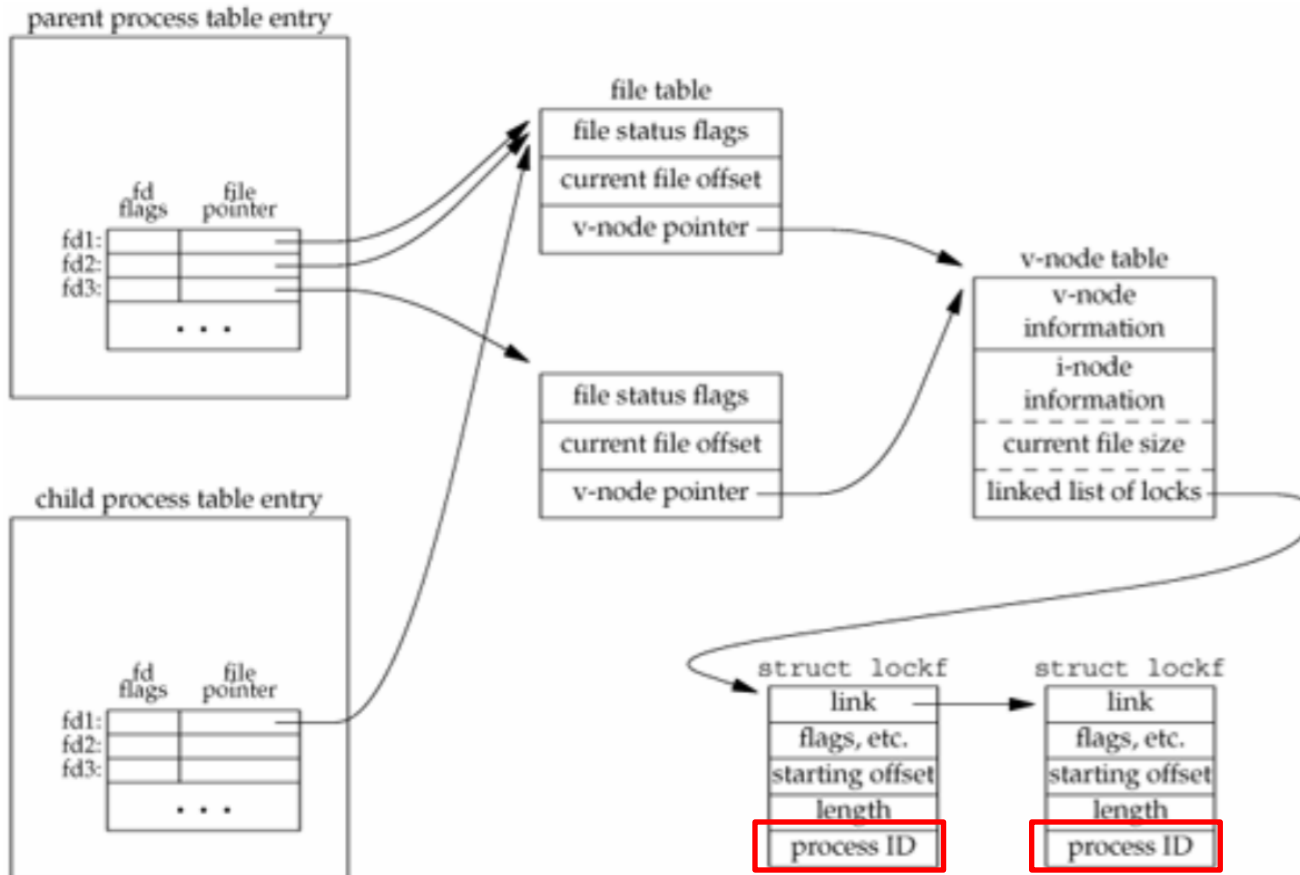
A Deadlock Example

```
int main(void) {
    int fd;
    pid_t pid;
    if ((fd = creat("templock", FILE_MODE)) < 0)
        err_sys("creat error");
    if (write(fd, "ab", 2) != 2)
        err_sys("write error");
    TELL_WAIT();
    if ((pid = fork()) < 0) { err_sys("fork error"); }
    else if (pid == 0) {
        lockabyte("child", fd, 0);
        TELL_PARENT(getppid()); /* notify parent to continue its execution */
        WAIT_PARENT();
        lockabyte("child", fd, 1);
    } else {
        lockabyte("parent", fd, 1);
        TELL_CHILD(pid); /* notify child to continue its execution */
        WAIT_CHILD();
        lockabyte("parent", fd, 0);
    }
    exit(0);
}
```

Implied Inheritance and Release of Locks

- Three basic rules
- Rule #1: 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
- Rule #2: 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
- Rule #3: Locks are **inherited** by a new program across an `exec`

Sample Lock Implementation



Another Example – lockfile: Write Lock on an Entire File

- Recall: A daemon can use a lock on a file to ensure that only one copy of the daemon is running
- The lockfile function

```
int lockfile(int fd) {
    struct flock fl; fl.l_type = F_WRLCK;

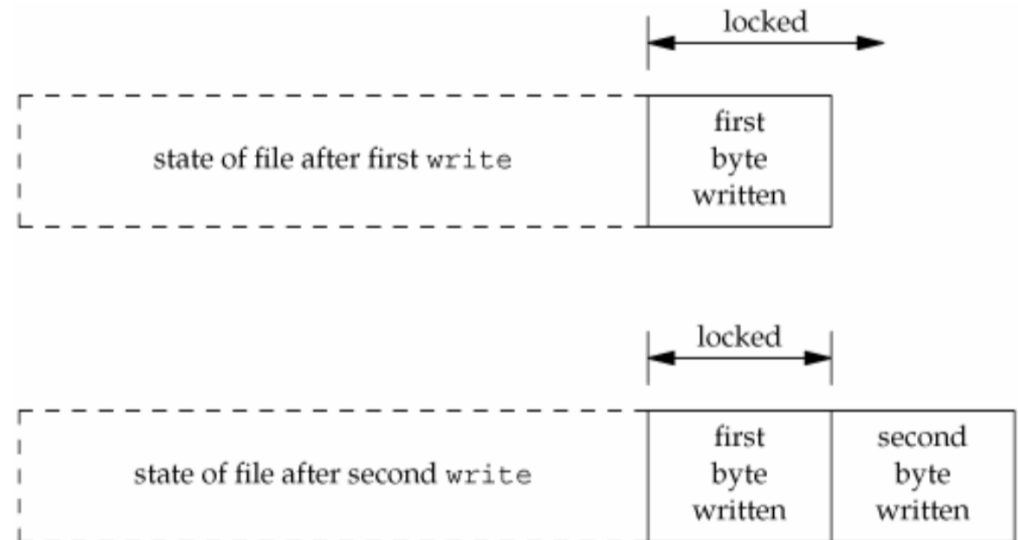
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;

    return(fcntl(fd, F_SETLK, &fl));
}
```

Locks at End of File

- Most implementations convert an l_w whence value of SEEK_CUR or SEEK_END into an absolute file offset
- An erroneous implementation

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



Advisory VS Mandatory Locks

- Most systems only implement advisory locks
- Advisory locks
 - All processes have to follow the same procedures to access protected files
 - lock -> read/write -> unlock
 - If a (uncooperating) process **DOES NOT** follow the procedure, i.e., does not call lock/unlock routines before read/write, it could break the protection
- Mandatory locks
 - Once a process has locked a file, read/write access to the file may be not granted, **EVEN** if a process does not call lock/unlock routines before read/write
 - On Linux, a filesystem can be mount with “**mand**” option to enable mandatory locks
 - Some systems enable mandatory lock if a file has enabled SGID but disabled group-execute bits

Effect of Mandatory Locks

Type of existing lock on region held by other process	Blocking descriptor, tries to		Nonblocking descriptor, tries to	
	read	write	read	write
read lock	OK	blocks	OK	EAGAIN
write lock	blocks	blocks	EAGAIN	EAGAIN

- open usually succeeds even if a file is locked by a mandatory lock
- However, if a file is opened with O_TRUNC or O_CREAT, open returns EAGAIN immediately

Record Locking – fcntl Alternatives

- flock
 - Lock a file descriptor `int flock(int fd, int operation);`
 - LOCK_SH: place a shared lock
 - LOCK_EX: place a exclusive lock
 - LOCK_UN: remove an existing lock
- lockf `int lockf(int fd, int cmd, off_t len);`
 - Lock starting from the current file position
 - F_LOCK: set an exclusive lock, could be blocked
 - F_TLOCK: same as F_LOCK, but never blocks – returns an error if a lock failed
 - F_UNLOCK: unlock
 - F_TEST: test if a region is locked or not

Record Locking Supported by Various UNIX systems

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

SUS: Single UNIX Specification

XSI: X/Open System Interfaces

Test of Mandatory Locks (1/3)

```
int main(int argc, char *argv[]) {
    int fd; pid_t pid;
    char buf[5];
    struct stat statbuf;
    if (argc != 2) {
        fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(1);
    }
    if ((fd = open(argv[1], O_RDWR | O_CREAT | O_TRUNC, FILE_MODE)) < 0)
        err_sys("open error");
    if (write(fd, "abcdef", 6) != 6)
        err_sys("write error");
    /* enable SGID and disable group execution */
    if (fstat(fd, &statbuf) < 0)
        err_sys("fstat error");
    if (fchmod(fd, (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("fchmod error");
    TELL_WAIT();
    if ((pid = fork()) < 0) { err_sys("fork error"); }
```

(continue to the next page ...)

Test of Mandatory Locks (2/3)

```
else if (pid > 0) {      /*parent*/
    /* write lock entire file */
    if (write_lock(fd, 0, SEEK_SET, 0) < 0)
        err_sys("write_lock error");
    TELL_CHILD(pid);
    if (waitpid(pid, NULL, 0) < 0)      err_sys("waitpid error");
} else {                /* child */
    WAIT_PARENT();      /* wait for parent to set lock */
    set_fl(fd, O_NONBLOCK);
    if (read_lock(fd, 0, SEEK_SET, 0) != -1) /* no wait */
        err_sys("child: read_lock succeeded");
    printf("read_lock of already-locked region returns %d\n", errno);
    /* now try to read the mandatory locked file */
    if (lseek(fd, 0, SEEK_SET) == -1)      err_sys("lseek error");
    if (read(fd, buf, 2) < 0)
        err_ret("read failed (mandatory locking works)");
    else
        printf("read OK (no mandatory locking), buf = %2.2s\n", buf);
}
exit(0);
}
```

Test of Mandatory Locks (3/3)

- Running the example
- On Linux (without mandatory locks)

- Note: errno 11 means EAGAIN

```
$ ./fig14.12-mandatory /tmp/xxx  
read_lock of already-locked region returns 11  
read OK (no mandatory locking), buf = ab
```

- On Linux (with mandatory locks)

- We mount a RAM disk with “mand” option first (require root privileges)

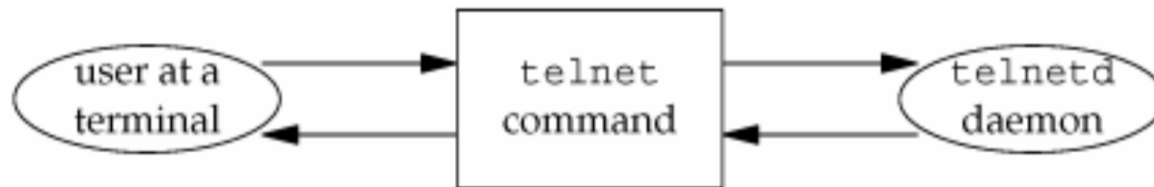
```
# mkdir /tmp/mand  
# mount -t tmpfs -o mand tmpfs /tmp/mand  
# chmod 1777 /tmp/mand/.  
$ ./fig14.12-mandatory /tmp/mand/xxx  
read_lock of already-locked region returns 11  
read failed (mandatory locking works): Resource temporarily unavailable
```

I/O Multiplexing – Why?

- For one-way communication, usually we can read from one descriptor and write to another

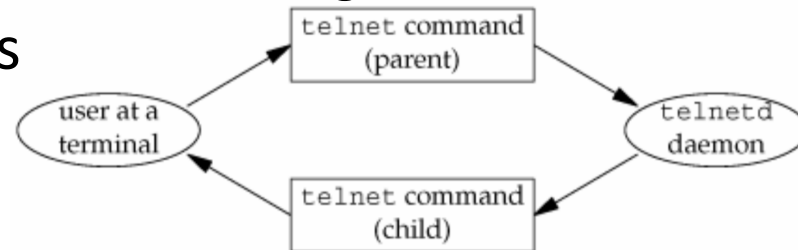
```
while ((n = read(STDIN_FILENO, buf, BUFSIZ)) > 0)
    if (write(STDOUT_FILENO, buf, n) != n)
        err_sys("write error");
```

- However, for two-way communication, the above implementation does not work



Possible Solutions*

- Work with two processes
 - Each process is used for handle one-way communication
 - When the child receives EOF, it terminates and the parent receives SIGCHLD
 - When the parent receives EOF, it has to notify the child to stop. We can use a signal for this, e.g., SIGUSR1
- Work with two threads
- Work with polling
 - Polling in a busy loop
 - Set the descriptors to non-blocking
 - Perform reads on both descriptors and forward when data is available



Possible Solutions (Cont'd)

- Asynchronous I/O (discussed later)
 - The kernel notifies us with a signal when a descriptor is ready for I/O
 - But not all systems support this feature
 - It may work only on descriptors that refer to terminal devices or networks
 - We have only one SIGIO or SIGPOLL per process – how to differentiate multiple descriptors?
 - You have to check each (non-blocking) descriptors
- I/O multiplexing (discussed later, before asynchronous I/O)
 - That's what we have in this chapter: Work with select and poll
 - Highest compatibility, usually $O(n)$ complexity
- Modern UNIX systems supports event multiplexing
 - kqueue for BSD and epoll for Linux
 - $O(1)$ algorithms: Much faster than traditional I/O multiplexing

The select Function

- Lets us do I/O multiplexing under all POSIX-compatible platforms

- Synopsis

```
int select(int maxfdp1,  
           fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  
           struct timeval *tvptr);
```

- The arguments

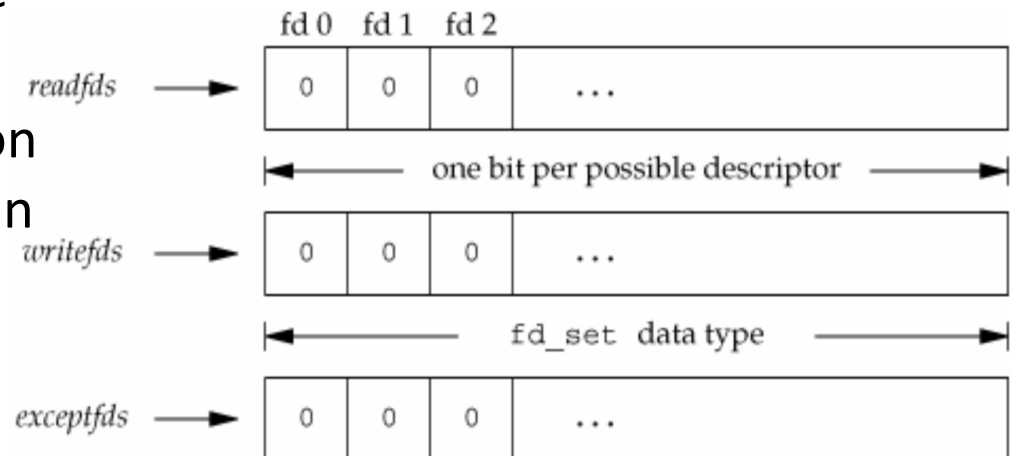
- Which descriptors we are interested in
- What conditions we are interested in for each descriptor (read, write, exception)
- How long we want to wait

- Returns

- The total count of the number of descriptors that are ready
- Which descriptors are ready for each of the three conditions

Arguments for the select Function

- The “maxfdp1”
 - The maximum descriptor number in the descriptor sets **plus one**
- The “tvptr” – How long we want to wait?
 - NULL: Wait infinitely
 - Not NULL, but tvptr->tv_sec == 0 and tvptr->tv_usec == 0: No wait
 - Not NULL, and tvptr->tv_sec != 0 or tvptr->tv_usec != 0: Wait for the given amount of time
- The descriptor sets
 - Read, write, and exception
 - A number of functions can be used to manipulate the descriptor sets



Manipulate Descriptor Sets

- Functions

```
void FD_ZERO(fd_set *fdset);          /* empty the set */
void FD_SET(int fd, fd_set *fdset);   /* add a fd into the set */
void FD_CLR(int fd, fd_set *fdset);   /* remove a fd from the set */

int FD_ISSET(int fd, fd_set *fdset); /* determine if a fd is in the set */
```

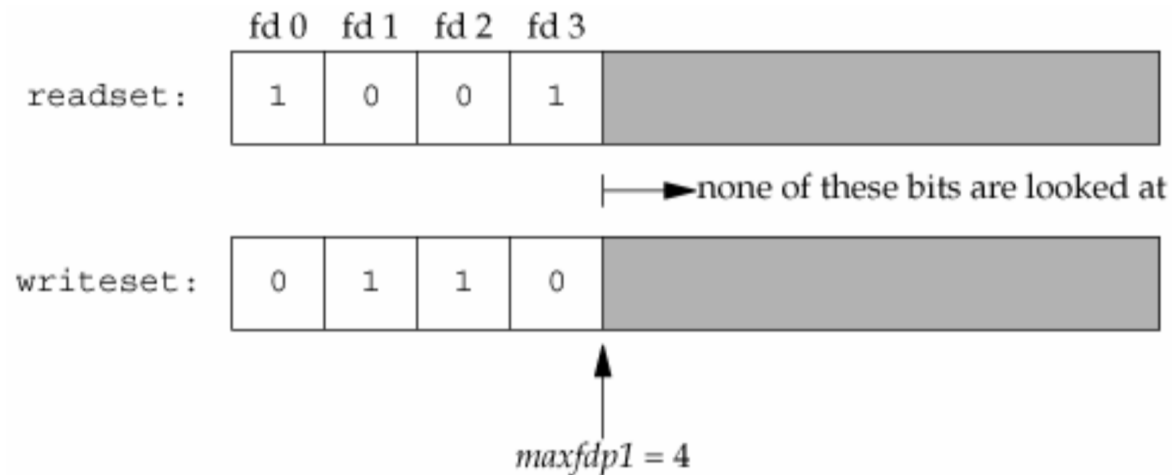
- Examples

```
fd_set rset;
int fd;

FD_ZERO(&rset);
FD_SET(fd, &rset);
FD_SET(STDIN_FILENO, &rset);
... (after returned from select)
if(FD_ISSET(fd, &rset)) { ... }
```


A select Example

```
fd_set readset, writeset;  
FD_ZERO(&readset);   FD_ZERO(&writeset);  
FD_SET(0, &readset); FD_SET(3, &readset);  
FD_SET(1, &writeset); FD_SET(2, &writeset);  
select(4, &readset, &writeset, NULL, NULL);
```



Return from select

- The return value
 - The number of descriptors in “ready” state, 0 on timeout, or -1 on error
- The select function changes the values stored in readfds, writefds, and exceptfds to reflect the state of interested descriptors
- The select function may optionally change the values in tvptr
- What does “ready” mean?
 - readfds: if a read from that descriptor will not block
 - writefds: if a write to that descriptor will not block.
 - exceptfds: if an exception condition is pending on that descriptor, it depends on the type of a descriptor
- A descriptor is blocking or not does not affect whether select blocks
 - It only depends on the “tvptr” argument

The pselect Function

- It is equivalent to the select function, with additional benefits
 - It uses a high resolution timeout structure (in nano-second)
 - Setup signal mask: an atomic operation for sigprocmask and select
- Synopsis

```
int pselect(int maxfdp1,  
            fd_set *readfds, fd_set *writefds, fd_set *exceptfds,  
            const struct timespec *tsptr,  
            const sigset_t *sigmask);
```

The poll Function

- Similar to select, with a different programming interface
- Instead of building a set of descriptors for each condition (read, write, and exception), we pass an array of descriptors
- Synopsis

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

– Returns: Number of ready descriptors, 0 on timeout, or -1 on error

- The pollfd structure

```
struct pollfd {
    int fd;           /* file descriptor to check, or <0 to ignore */
    short events;    /* events of interest on fd */
    short revents;   /* events that occurred on fd */
};
```

The events and revents Flags

Name	Input to events?	Results from revents?	Description
POLLIN	•	•	Equivalent to POLLRDNORM POLLRDBAND
POLLRDNORM	•	•	Normal data (priority 0) can be read without blocking
POLLRDBAND	•	•	Data from a nonzero priority band can be read without blocking
POLLPRI	•	•	High-priority data can be read without blocking
POLLOUT	•	•	Normal data can be written without blocking
POLLWRNORM	•	•	Same as POLLOUT
POLLWRBAND	•	•	Data for a nonzero priority band can be written without blocking
POLLERR		•	An error has occurred
POLLHUP		•	A hangup has occurred
POLLNVAL		•	The descriptor does not reference an open file

The poll Function (Cont'd)

- Note: a “hangup” descriptor
 - We can no longer write to the descriptor
 - However, we may still read data from the descriptor
- The “timeout” value
 - How long we want to wait
 - -1: Wait infinitely
 - 0: No wait
 - > 0: Wait for “timeout” milliseconds

Asynchronous I/O

- BSD asynchronous I/O
 - Asynchronous notification using signal
 - Supported on Linux, BSD, and Mac OS X
- POSIX asynchronous I/O
 - Provide more flexibilities
 - In addition to using signals, it is able to launch a thread to handle the notifications
 - Not widely supported

BSD Asynchronous I/O

- Asynchronous notification from the kernel by signals
- Only one signal per process
 - Unable to tell which descriptor the signal corresponds to when the signal is delivered
 - Have to check all the involved descriptors
- Procedures to work with BSD asynchronous I/O
 1. Register a handler for SIGIO or SIGPOLL
 2. Set the owner process (group) for the descriptor – `fcntl(F_SETOWN)`
 3. Enable asynchronous I/O – `fcntl(F_SETFL)` and add `O_ASYNC` flag
- You may also need to register a handler for SIGURG if the descriptor supports out-of-band data
- Step #3 may be failed on unsupported type of descriptors
 - It may work with terminals, pseudoterminals, sockets, pipes and FIFOs

BSD Asynchronous I/O – Echo Example

```
void sig_io(int s) {
    int rlen;
    char buf[8192];
    if((rlen = read(0, buf, sizeof(buf))) > 0)
        write(1, buf, rlen);
}

int main() {
    int flag;
    signal(SIGIO, sig_io);
    if(fcntl(STDIN_FILENO, F_SETOWN, getpid()) < 0)
        err_sys("fcntl(F_SETOWN)");
    if((flag = fcntl(STDIN_FILENO, F_GETFL)) < 0)
        err_sys("fcntl(F_GETFL)");
    if(fcntl(STDIN_FILENO, F_SETFL, flag | O_ASYNC) < 0)
        err_sys("fcntl(F_SETFL)");
    for(;;) pause();
    return 0;
}
```

POSIX Asynchronous I/O

- More flexible
- Procedures to work with POSIX asynchronous I/O
 1. Fill the AIO control block structure
 2. Initiate an AIO operation by calling `aioread` or `aiowrite` function
 3. Optionally, call `aiofsync` to perform all pending write operations
 4. Use `aioerror` function to check the completion status
 5. Use `aio_return` function to get AIO operation's return value
- We can suspend an AIO process by calling `aio_suspend` function
- We can cancel an AIO operation by calling `aio_cancel`

POSIX AIO Control Block

```
struct aiocb {
    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 */
};
```

- Compared to read and write system call

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

The sigevent Structure

```
union sigval {          /* Data passed with notification */
    int     sival_int;    /* Integer value */
    void    *sival_ptr;  /* Pointer value */
};

struct sigevent {
    int     sigev_notify; /* Notification method:
                           SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD */
    int     sigev_signo; /* Notification signal */
    union sigval sigev_value; /* Data passed with notification */
    void    (*sigev_notify_function) (union sigval);
                           /* Function used for thread
                           notification (SIGEV_THREAD) */
    void    *sigev_notify_attributes;
                           /* Attributes for notification thread
                           (SIGEV_THREAD) */
};
```

POSIX AIO Basic Functions

- Initiate an asynchronous read operation

```
int aio_read(struct aiocb *cb)
```

- Initiate an asynchronous write operation

```
int aio_write(struct aiocb *cb);
```

- Request to perform data synchronization

```
int aio_fsync(int op, struct aiocb *cb);
```

- Returns: 0 if OK, or -1 on error

POSIX AIO: Completion Status

- Check completion status

```
int aio_error(const struct aiocb *cb);
```

- Returns
- 0: Success
- -1: Error, check errno
- EINPROGRESS: operation is still pending
- Other: error code numbers

- Get return value of a completed operation

```
int aio_return(const struct aiocb *cb);
```

- Can be called only once for each AIO operation
- Return: the return value of read, write, or fsync operation, or -1 on error

POSIX AIO Example: cat

```
static void aio_completed(union sigval param);
static char buf[8192];
static struct aiocb cb = {
    .aio_buf = buf,
    .aio_nbytes = sizeof(buf),
    .aio_sigevent.sigev_notify = SIGEV_THREAD,
    .aio_sigevent.sigev_notify_function = aio_completed
};

int
main(int argc, char *argv[]) {
    if(argc < 2)    cb.aio_fildes = 0;
    else           cb.aio_fildes = open(argv[1], O_RDONLY);
    if(cb.aio_fildes < 0)    err_sys("open");
    if(aio_read(&cb) < 0)    err_sys("main/aio_read");
    for(;;) pause();
    return 0;
}
```

POSIX AIO Example: cat (Cont'd)

```
void aio_completed(union sigval param) {
    int err, len;
    off_t off;
    if((err = aio_error(&cb)) != 0) {
        if(err == -1)            err_sys("error");
        if(err == EINPROGRESS)  return;
        fprintf(stderr, "error: %s\n", strerror(err));
        exit(-1);
    }
    len = aio_return(&cb);
    write(1, (char*) cb.aio_buf, len);
    if((off = lseek(cb.aio_fildes, 0, SEEK_CUR)) >= 0)
        cb.aio_offset = off;
    if(off >= 0 && len != cb.aio_nbytes)
        exit(0);
    if(aio_read(&cb) < 0)
        err_sys("aio_read");
}
```


readv and writev Functions

- Read into and write from multiple noncontiguous buffers in a single function call
- These operations are called scatter read and gather write
- Synopsis

```
ssize_t readv(int filedes, const struct iovec *iov , int iovcnt);  
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

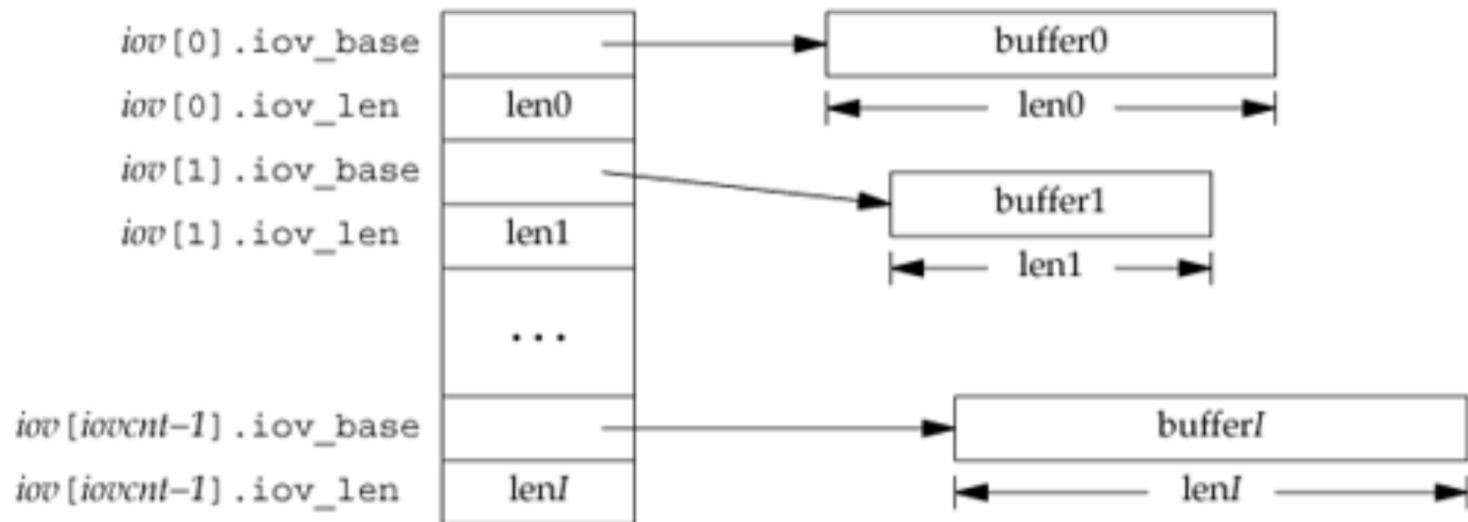
– Returns: Number of bytes read or written, or -1 on error

- The iovec structure

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

readv and writev Functions (Cont'd)

- We can combine multiple data reads into a single one system call
- Similarly, we can combine multiple data writes into a single write system call



readn and writen Functions

- Pipes, FIFOs, terminals, networks, and some devices have the two properties
 - A read operation may return less than we asked for when the data available is less than the buffer size or an EOF is encountered
 - A write operation may also return less than we specified if there is a flow control mechanism applied
- We can use the two functions to read or write EXACTLY N bytes of data
- These two functions simply call read or write as many times as required to read or write the entire N bytes of data
- Synopsis

```
ssize_t readn(int filedes, void *buf, size_t nbytes);
ssize_t writen(int filedes, void *buf, size_t nbytes);
```

 - Returns: Number of bytes read or written, or -1 on error

The Implementation of readn

```
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 */
}
```

The Implementation of `written`

```
ssize_t          /* Write "n" bytes to a descriptor */
written(int fd, 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 lets us map a file on disk into a buffer in memory
- When we fetch bytes from the buffer, the corresponding bytes of the file are read
- 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
- We can tell the kernel to map a given file to a region in memory using the mmap function

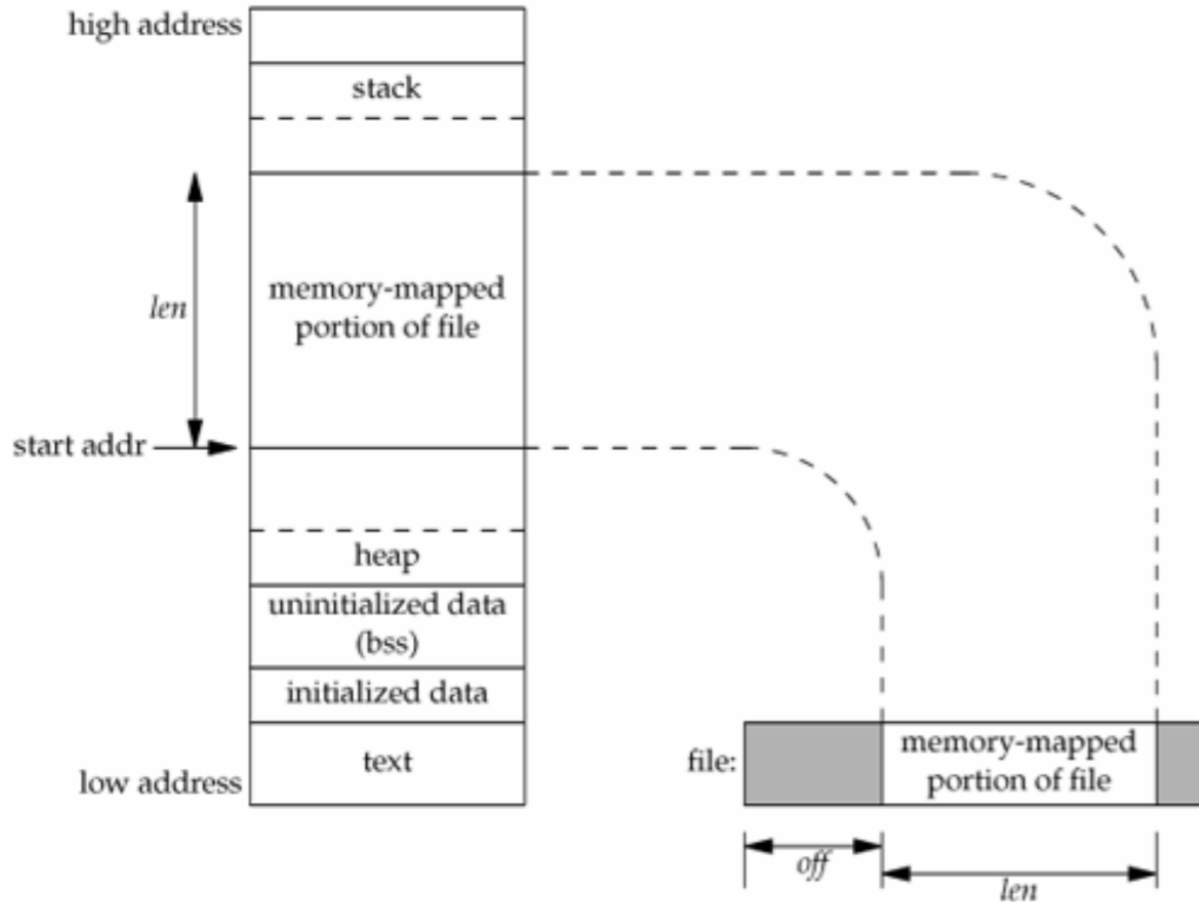
```
void *mmap(void *addr, size_t len, int prot, int flag,  
           int filedes, off_t off);
```

- Returns: starting address of mapped region if OK, or MAP_FAILED on error

mmap Function Arguments

- **addr**: Specify the address we want the mapped region to start
 - We normally set this to 0 to allow the system to choose the starting address
- **prot**: Specify the protection of the mapped region
 - PROT_NONE, OR of PROT_READ, PROT_WRITE, and PROT_EXEC
 - The protection specified for a region cannot allow more access than the open mode of the file. Read-only files cannot be mmap'ed with PROT_WRITE
- **filedes, off, and length**
 - The opened file descriptor, offset, and length
- **flags** – One of MAP_SHARED and MAP_PRIVATE must be used
 - MAP_FIXED: The return value must equal addr – not recommended
 - MAP_SHARED: Store operations modify (write) the mapped file
 - MAP_PRIVATE: Store operations cause a **private copy** of the mapped file to be created – Any modifications **affect the copy, not the original file**

Illustration of mmap in Action



mmap Relevant Functions

- mprotect changes the protection of a mapped region

```
int mprotect(void *addr, size_t len, int prot);
```

- addr must be aligned to a page boundary

- msync flushes changes to the file

```
int msync(void *addr, size_t len, int flags);
```

- Only works for MAP_SHARED regions
- Flags must be either MS_ASYNC or MS_SYNC, plus optional MS_INVALIDATE
- MS_INVALIDATE asks to invalidate other mappings of the same file

mmap Relevant Functions (Cont'd)

```
int munmap(caddr_t addr, size_t len);
```

- A memory-mapped region is automatically unmapped when the process terminates or by calling `munmap` directly
- Closing the file descriptor `filedes` **DOES NOT** unmap the region
- Modifications to memory in a `MAP_PRIVATE` region are **discarded** when the region is unmapped

mmap: Final Notes

- mmap length can be larger than the file size, but data written to those additional spaces does not append to the file
- mmap regions are inherited by a child, but are not inherited across exec
- Many mmap flags are platform dependent
 - See mmap manual pages on your platform!
- Memory allocation with mmap
 - Special case to use flag `MAP_ANONYMOUS`
 - A memory region is not associated with a file descriptor
 - The argument `filedes` and `off` are ignored
 - Some implementations require `filedes` to be `-1`



Source: <https://s-media-cache-ak0.pinimg.com/originals/e0/e7/43/e0e7432216c9c2fc59664c190337e886.jpg>