Chapter 1 - Introduction and Essential Concepts

- system software interfaces directly with the kernel and core system libraries
- Linux is very similar to Unix but diverges in places

System Programming

- the focus is on the system level API
- must be aware of the architecture, OS, and hardware
- distinct from application programming
- mostly C
- kernel development and device driver software
- system calls, the C library, and the C compiler

System Calls

- aka syscalls
- roughly 300 in Linux in i386
- differs by architecture
- 90% are is a subset shared by most architectures

Invoking system calls

- user space applications cannot be linked with kernel space applications
- user space application "signals" the kernel that it wants to make a system call and then the kernel allows the user space application to "trap" into the kernel for the system call
- on i386, the software interrupt instruction "int" or 0x80 is used, which is the system call handler
- the application passes which system call to use and which parameters to use by placing them in machine registers. The call number is placed in eax.
- uses ebx, ecx, edx, esi, and edi. If it uses more, then one register is used to point to a buffer in user space with all parameters.
- semantics may differ from system to system but generally the same

The C Library

- libc or glibc, provides the C library, wrappers for system calls, threading support, and basic application facilities

The C Compiler

- generally gcc on Linux, both the collection of GNU compilers but also the binary

APIs and ABIs

APIs

- the abstraction by which one piece of software interacts with another
- just the interface, not the implementation

ABIs

- API provides source code interface whereas ABI provides a low-level bianry interface
- e.g. how an application interacts with the kernel and other pre-compiled binaries that only expose their interface through the process interaction
- ABIs dictate calling conventions, byte ordering, register use, system call invocation, linking, library behavior, and the binary object/file format
- ABIs are not consistent and are dictated by an OS and machine name (x86-64)
- enforced by the toolchain
- important if writing assembly code or working on the toolchain

Standards

- Covered by the Single UNIX Specification (SUS), POSIX, and IEEE

C Language Standards

- K&R C started in 1978 and covered the standard until the ANSI C standard started in 1983 and continued to 1989, with C90 released in 1990
- updates in 95 and 99

Linux and the Standards

- aims towards POSIX and SUS compliance with the optional real-time and threading support

Concepts of Linux Programming

- Linux follows the mutual set of Unix abstractions and interfaces

Files and the Filesystem

- "everything is a file"
- open files have a descriptor, a mapping from an open file back to the file itself
- much Linux system programming involves opening, manipulating and closing a file

Regular files

- bytes of data organized into a linear array called a byte stream
- other filesystems (like VMS) support structured byte streams and data from regular files, but Linux does not
- any byte in a file can be read or written and is referenced from the file offset (or file position)
- file writing generally occurs at the end, not possible to a expand a file by writing to the middle of it
- position in a file is limited by the size of the C type used to store it (with 64 bits it is 2^64)
- files are referenced by inodes (information nodes) and this can be retrieved by the filename which is actually stored with the directory data
- inode has mod time, owner, type, length, and data location but no name

Directories and links

- Directories maps file names to inodes (direct access via inode is a security issue)
- Access to any file first requires access to the containing directory file
- walks each directory entry (dentry) by opening the parent directory, getting the inode for the next directory and continuing
- kernel also stores a dentry cache
- directories are viewed as normal files but the kernel does not allow access (i.e. manipulation) of directory files in the same way as regular files

Hard links

- when multiple names map to the same inode
- can be in the same directory or in separate directories
- to remove a file it must be unlinked
- inodes contain a link count and a file's inode and data is not removed until the link count drops to 0

Symbolic links

- an inode number has no meaning outside of it's filesystem
- symbolic links allow links that can span filesystems
- looks like a regular file, has its own inode and data chunk
- the data chunk contains the full pathname of the linked file
- a symlink to nonexistent file is called a broken link
- incurs more overhead than a hard link because it requires the path resolution
- symlinks act like shortcuts

Special files

- kernel objects represented as files
  - block device files
  - character device files
  - named pipes (aka FIFOs)
  - Unix domain sockets
- device access via device files
  - character device is accessed as a linear queue of bytes
  - block device is accessed as an array of bytes (a block) which allows user space applications to access the bytes in the array in any order
- named pipes are used for IPC first in first out message passing
- sockets are an advanced means of IPC both on the same machine and via the network

Filesystems and namespaces

- Linux provides a global and unified namespace of files and directories
- Other filesystems separate namespaces, e.g. by disk drive, etc.
- mounting and unmounting is the act of adding and removing a filesystem to another filesystem via a mount point
- filesystems usually exist on physical media but Linux also supports filesystems that only exist in memory and networked filesystems
- some disk drives, etc., are partitionable, or, in other words, are able to be divided up into multiple separate filesystems
- Linux supports multiple types of filesystems via the virtual filesystem
- block devices access data in units that are multiples of the sector size (512 bytes is common)
- the smallest logically addressable unit on a filesystem is the block

- the block is an abstraction of the filesystem and they are generally larger than the sector size and smaller than the page size (the smallest unit addressable by the MMU) common block sizes are 512, 1024, and 4096 bytes

Processes

- object code in execution
- consist of data, resources, state and a virtualized computer
- consist of a file in ELF (others like COFF)
- most important seections are the text section, the data section, and the bss section
- text - executable code and read only data
- data - initialized variables, generally readable and writable
- bss - uninitialized global variables (block started by symbol or block storage segment)
- absolute - nonrelocatable
- undefined
- process associated with system resources like timers, signals, files, network connections, hardware, and IPC mechanisms
- a process is a virtualization abstraction, running as though it has sole control of the system
- the kernel manages the sharing of system resources

Threads

- a process has one or more threads of execution
- most are single threaded
- some are multithreaded
- each thread has its own stack
- global data etc within a process is shared among all threads
- Linux views threads as normal processes that share some resources, mainly the address space
- pthreads are the main threading components on Linux

Process hierarchy

- process has a PID
- processes form a process tree, started by the init process (generally init(8))
- new processes are created by fork, with parent and child
- an orphaned child process is reparented to the init process
- when a process is terminated a portion of it is kept in memory to allow a parent to query the child's status upon termination

- this is called waiting on - a child process that is terminated but not waited on is a zombie

Users and Groups

- users get a uid and each process is associated with a real uid
- root is 0 and the login program checks users via /etc/passwd
- processes have a real uid, an effective uid, a saved uid, and a filesystem uid

Permissions

- same as Unix
- files have an owning user and an owning group, stored in the inode
- permissions

```
bit     octal value     text value      permission
8       400             r--------       owner - read
7       200             -w-------       owner - write
6       100             --x------       owner - execute
5       040             ---r-----       group - read
4       020             ----w----       group - write
3       010             -----x---       group - execute
2       004             ------r--       all - read
1       002             -------w-       all - write
0       001             --------x       all - execute
```

- Linux supports access control lists (ACLs) which provide more detailed access control and security measures at the cost of increased complexity

Signals

- one way async notifications
- kernel -> process
- process -> process
- process -> itself
- roughly 30 depending on the architecture
- name and constant (i.e. SIGHUP = 1 on i386)
- SIGKILL and SIGSTOP always kill and stop a process
- all other signals can be handled on a case-by-case basis by processes
- programs can provide signal handler functions which will cause execution to jump to the function and then return

Interprocess Communication

- Linux uses standard Unix IPC mechanisms with a few extras
- pipes, named pipes, semaphores, message queues, shared memory, and futexes

Headers

- a handful of headers provide the Linux system APIs

Error Handling

- glibc provides errno support for library and system calls
- defined in <errno.h> as

```
extern int errno;
```

- valid only immediately after an errno setting function
- can be read and written
- standard errno constants:

```
define      description
------      -----------
E2BIG       arg list too long
EACCESS     permission denied
EAGAIN      try again
EBADF       bad file number
EBUSY       device or resource busy
ECHILD      no child process
EDOM        math arg outside of domain
EEXIT       file already exists
EFAULT      bad address
EFBIG       file too larges
EINTR       system call was interrupted
EINVAL      invalide arg
EIO         I/O error
EISDIR      is a directory
EMFILE      too many open files
EMLINK      too many links
ENFILE      file table overflow
ENODEV      no such device
ENOENT      no such file or directory
ENOEXEC     exec format error
ENOMEM      out of memory
ENOSPC      no space left on device
ENOTDIR     not a directory
ENOTTY      inappropriate I/O control operation
ENXIO       no such device or address
```

```
EPERM       operation not permitted
EPIPE       broken pipe
ERANGE      result too large
EROFS       read-only filesystem
ESPIPE      invalid seek
ESRCH       no such process
ETXTBSY     text file busy
EXDEV       improper link
```

- perror can be used to report errors to stderr
- strerror and strerror_r can be used to get a a string describing the error defined by errno
- strerror is thread safe whereas strerror_r is not
- save errno across multiple function calls as many functions can modify it

Chapter 2 - File I/O

- kernel maintains per process list of open files called the file table
- child process receives a copy of the parent's file table
- file descriptors are int type (0 - 1024 by default, 1048576 potential max)
- by convention each process has 0 - stdin, 1 - stdout, 2 - stderr (STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO)

Opening Files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

- flags must be one of O_RDONLY, O_WRONLY, or O_RDWR
- these three flags can be bitwise or'ed with the following to mdify the open behavior
    - O_APPEND - the file will be opened in append mode. That is, before each write, the file position will be updated to point to the end of the file
    - O_ASYNC - a signal (SIGIO by default) will be generated when the specified file becomes readable or writable. This flag is available only for terminals and sockets, not for regular files
    - O_CREAT - if the file denoted by name does not exist, the kernel will create it. If the file already exists, this flag has no effect unless O_EXCL is also given
    - O_DIRECT - the file will be opened for direct I/O
    - O_DIRECTORY - if name is not a directory, the call to open() will fail. This flag is used internally by the opendir() library call

    - O_EXCL - when given with O_CREAT, this flag will cause the call to open() to fail if the file given by name already exists. This is used to prevent race conditions on file creation
    - O_LARGEFILE - the given file will be opened using 64-bit offsets, allowing files larger than two gigabytes to be opened. This is implied on 64-bit architectures.
    - O_NOCTTY - if the given name refers to a terminal device (say, /dev/tty), it will not become the process' controlling terminal, even if the process does not currently have a controlling terminal. This flag is not frequently used.
    - O_NOFOLLOW - if name is a symbolic link, the call to open() will fail. Normally, the link is resolved, and the target file is opened. If other components in the given path are links, the call will still succeed.
    - O_NONBLOCK - if possible, the file will be opened in nonblocking mode. Neither the open() call, nor any other operation will cause the process to block (sleep) on the I/O. This behavior may be defined only for FIFOs.
    - O_SYNC - the file will be opened for synchronous I/O. No write operation will complete until the data has been physically written to disk; normal read operations are already synchronous, so this flag has no effect on reads. POSIX additionally defines O_DSYNC and O_RSYNC; on Linux, these flags are synonymous with O_SYNC.
    - O_TRUNC - if the file exists, it is a regular file, and the given flags allow for writing, the file will be truncated to zero length. Use of O_TRUNC on a FIFO or terminal device is ignored. Use on other file types is undefined. Specifying O_TRUNC with O_RDONLY is also undefined, as you need write access to the file in order to truncate it.

Owners of New Files

- uid of the file is the effective uid of the creating process
- gid of the file is generally determined the same way
- in BSD the the gid is the gid fo the parent directory

Permissions of New Files

- the mode argument is ignored unless a file is created but required if O_CREAT is included in the flags set
- the mode provides permissions for the newly created file
- on creation the mode is not checked so modes that are contradictory can be used
- mode is the familiar bitset (0644, 0755, etc) but POSIX provides these constants:
    - S_IRWXU - owner has read, write, and execute permission
    - S_IRUSR - owner has read permission
    - S_IWUSR - owner has write permission
    - S_IXUSR - owner has execute permission
    - S_IRWXG - group has read, write, and execute permission
    - S_IRGRP - group has read permission

    - S_IWGRP - group has write permission
    - S_IXGRP - group has execute permission
    - S_IRWXO - everyone else has read, write, and execute permission
    - S_IROTH - everyone else has read permission
    - S_IWOTH - everyone else has write permission
    - S_IXOTH - everyone else has execute permission
- the actual permission bits to disk are actually determined by binary &-ing the mode bits with the complement fo the user's file creation mask (umask)
- NOTE: do not need to use these, can just call with:

```
open (file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

The creat() Function

- replaces open with O_WRONLY | O_CREAT | O_TRUNC

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *name, mode_t mode);
```

- creat can be a system call or can be a wrapper around open (as in glibc)

Return Values and Error Codes

- on error, open and creat return -1

Reading via read()

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t len);
```

- reads up to len bytes of file fd
- returns -1 on error and errno is set
- NOTE: errno is a global variable (or a symbol since it is generally defined as a macro returned from a function), set on error, with the variable and various errors defined in errno.h
- basic usage:

```
unsigned long word;
ssize_t nr;
```

```
nr = read (fd, &word, sizeof (unsigned long));
if (nr == -1)
    /* error */
```

- read may return a positive non-zero value less than len if it cannot read full length
- if it returns 0 it means it read EOF
- some errors for read are recoverable (i.e. EINTR)
- results possibilities:
    - return == len - expected outcome, all bytes are read and stored in buffer
    - 0 < return < len
        - interrupt signal was received
        - error occurred in the middle of the read
        - more than 0 but less than len bytes of data was available
        - EOF was reached before len bytes were read
    - 0 - EOF no bytes to read
    - call blocks because no data is available - won't happen in nonblocking mode
    - -1 & errno == EINTR - signal was received and the call can be reissued
    - -1 & errno == EAGAIN - read would block because no data, wait and reissue (only in nonblocking mode)
    - -1 & errno == other error - more serious error

Reading All the Bytes

```
ssize_t ret;
while (len != 0 && (ret = read (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("read");
        break;
    }
    len -= ret;
    buf += ret;
}
```

Nonblocking Reads

- call to read returns immediately when no data is available (i.e. nonblocking I/O)
- use o_NONBLOCK
- check EAGAIN

```
char buf[BUFSIZ];
ssize_t nr;
```

```
start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
    if (errno == EINTR)
            goto start; /* oh shush */
    if (errno == EAGAIN)
            /* resubmit later */
    else
            /* error */
}
```

- NOTE: this trivial example should not be used

Other Error Values

- EBADF - the given file descriptor is invalid, or not open for reading
- EFAULT - the pointer provided by buf is not inside the calling process' address space
- EINVAL - the file descriptor is mapped to an object that does not allow reading
- EIO - a low-level I/O error occurred

Size Limits on read()

- size_t and ssize_t - store values used to measure sizes in bytes (ssize_t is signed)
- SIZE_MAX and SSIZE_MAX limit the number of bytes that can be read
- calling read with a value of 0 causes read to return 0 immediately

Writing with write()

```
#include <unistd.h>

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

- files that do not support seeking (e.g. character devices) always start at the head
- returns 0 for no bytes written, num bytes written, or -1 for error
- basic usage:

```
const char *buf = "My ship is solid!";
ssize_t nr;
/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
    /* error, check errno */
else if (nr != count)
    /* possible error, but 'errno' not set */
```

Partial Writes

- write is less likely to write partially than read
- for regular files writes do not need to be performed in a loop
- for sockets a loop may be required to check that the full write was performed
- sometimes the second pass will reveal the error where the first only returned a partial write

```
ssize_t ret, nr;
while (len != 0 && (ret = write (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("write");
        break;
    }
    len -= ret;
    buf += ret;
}
```

Append Mode

- in append mode, writes do not occur at the file position but always at the end
- this becomes more significant when two processes are working on the same file as one may update the stored file position
- use mainly where it is logical, like updating log files

Nonblocking Writes

- with O_NONBLOCK writes, a normally blocking call will return -1 and errno will be set to EAGAIN - the call should then be reissued later

Other [write] Error Codes

- EBADF - the given file descriptor is not valid, or is not open for writing.
- EFAULT - the pointer provided by buf points outside of the process' address space.
- EFBIG - the write would have made the file larger than per-process maximum file limits, or internal implementation limits.
- EINVAL - the given file descriptor is mapped to an object that is not suitable for writing.
- EIO - a low-level I/O error occurred
- ENOSPC - the filesystem backing the given file descriptor does not have sufficient space.

- EPIPE - the given file descriptor is associated with a pipe or socket whose reading end is closed. The process will also receive a SIGPIPE signal. The default action for the SIGPIPE signal is to terminate the receiving process. Therefore, processes receive this errno value only if they explicitly ask to ignore, block, or handle this signal.

Size Limits on write()

- if count is larger than SSIZE_MAX the results are undefined
- calling write with a value of 0 causes write to return 0 immediately

Behavior of write()

- a successful call to write copies the data into a kernel buffer after performing a few checks
- later, the kernel gathers "dirty" buffers and writes them optimally when performing a "writeback"
- the kernel handles looking for cached data vs reading from disk and interleaving reads and writes
- some potential issues can occur with crashes and disk failures
- to minimize this, the kernel has a max buffer age
- can be configured with /proc/sys/vm/dirty_expire_centiseconds

Synchronized I/O

- do not think about buffered writes in general, otherwise use synchronization calls

fsync() and fdatasync()

```
#include <unistd.h>

int fsync (int fd);
```

- ensures all dirty data associated with a file is written to disk
- writes data and metadata
- fsync does not know if the data was written to the drive's cache or actually written to the drive

```
#include <unistd.h>

int fdatasync (int fd);
```

- fdatasync just flushed data but does not guarantee that metdata is written to disk

- usage:

```
int ret;
ret = fsync (fd);
if (ret == -1)
    /* error */
```

Return values and error codes

- EBADF - the given file descriptor is not a valid file descriptor open for writing
- EINVAL - the given file descriptor is mapped to an object that does not support synchronization
- EIO - a low-level I/O error occurred during synchronization. This represents a real I/O error, and is often the place where such errors are caught
- fdatasync may not be implemented on a system, can try fsync if fdatasync returns EINVAL

sync()

- sync synchronizes all dirty buffers to disk

```
#include <unistd.h>
void sync (void);
```

- sync is not required to wait until data is flushed and for many systems it is best to call sync multiple times
- Linux does require sync to wait
- prefer fsync and fdatasync

The O_SYNC Flag

- use the o_SYNC flag with open to ensure that all file I/O should be synchronized
- can look at it as all write calls implicitly call fsync before returning (but more efficiently)
- adds some overhead

O_DSYNC and O_RSYNC

- on Linux these are synonymous with O_SYNC
- O_DSYNC specifies that only data be synced after each write (i.e. implicit fdatasync)
- O_RSYNC means reads will be synchronized also and is used with O_SYNC and O_DSYNC. It generally only means that reads will update metadata which generally means that it update the access time.

- The closest behavior on Linux is invoking fdatasync after each read which is unnecessary

Direct I/O

- It is best to not bypass the system level tools because you will rarely be able to improve performance
- using O_DIRECT will limit the kernel I/O management
- bypasses page cache and makes I/O synchronous -> nothing will return until complete
- requires that request length, buffer alignment and file offsets be integer multiples of the underlying device's sector size, generally 512 bytes
- before 2.6 it required that everything be aligned in the filesystem's logical block size -> for compatibility align to this despite the additional requirements

Closing Files

```
#include <unistd.h>
int close (int fd);
```

- unmaps file descriptor, returns 0 on success and -1 on failure
- has no bearing on when a file is flushed to disk
- frees kernel data structure related to the file
- unpins the in memory copy of the inode
- if unlinked (removed) it may not be removed until the inode is removed from memory

Error Values

- always check the return code of close
- EBADF - invalid file descriptor
- EIO - low-level I/O error
- never returns EINTR even though LINUX allows it

Seeking with lseek()

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fd, off_t pos, int origin);
```

- just updates the offset stored for the file

- SEEK_CUR - the current file position of fd is set to its current value plus pos, which can be negative, zero, or positive. A pos of zero returns the current file position value.
- SEEK_END - the current file position of fd is set to the current length of the file plus pos, which can be negative, zero, or positive. A pos of zero sets the offset to the end of the file.
- SEEK_SET - the current file position of fd is set to pos. A pos of zero sets the offset to the beginning of the file.
- usage:

```
off_t ret;
ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) -1)
    /* error */
```

Seeking Past the End of a File

- if lseek sets the current position past the end of a file, a read will return EOF but a write will create a hole in the file
- In Unix, holes do not occupy disk space so the total size of all files can add up to more than the disk space
- called a sparse file
- reading the hole will return 0s

Error Values

- EBADF - the given file descriptor does not refer to an open file descriptor.
- EINVAL - the value given for origin is not one of SEEK_SET, SEEK_CUR, or SEEK_END, or the resulting file position would be negative. The fact that EINVAL represents both of these errors is unfortunate. The former is almost assuredly a compile-time programming error, whereas the latter can represent a more insidious runtime logic error.
- EOVERFLOW - the resulting file offset cannot be represented in an off_t. This can occur only on 32-bit architectures. Currently, the file position is updated; this error indicates just that it is impossible to return it.
- ESPIPE - the given file descriptor is associated with an unseekable object, such as a pipe, FIFO, or socket.

Limitations

- limited by size of off_t
- no problem for 64 bit architectures but can cause EOVERFLOW errors on 32 bit

Positional Reads and Writes

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

- use pread and pwrite (positional read and positional write) for atomic reads and writes with lseek functionality

Error Values

- for pread any errno returned by read or lseek is possible
- for pwrite any errno returned by write or lseek is possible

Truncating Files

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

```
#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

- both set the file length to 0
- ftruncate takes a file descriptor whereas truncate takes the path to a writable file
- both can "truncate" to a larger size and neither update the current file position

Multiplexed I/O

- juggling multiple files via their descriptors often happens in programs, especially GUI based or event driven programs
- without threads a single program cannot block on more than one file at the same time

- one possibility is non-blocking I/O but this inefficient because I/O requests need to constantly be repeated or the program needs to be repeated until the I/O is possible
- multiplexed I/O allows a program to concurrently block on many files
  - wait for any file descriptors to become ready for I/O
  - sleep otherwise
- Linux provides 3 solutions

select()

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

- blocks until the fd set is ready for I/O or until a timeout has elapsed
- on successful return each fd set is modified so that it only contains fds ready for the I/O specified by that set
- timeout is a pointer to a timeval struct

```
#include <sys/time.h>
struct timeval {
    long tv_sec;         /* seconds */
    long tv_usec;        /* microseconds */
};
```

- FD_CLR - removes a file descriptor from a given set
- FD_ISSET - tests whether an fd is part of a given set
- FD_SET - adds an fd to a given set
- FD_ZERO - removes all fds from a given set

Return values and error codes

- on success returns the number of fds ready for I/O
- EBADF - an invalid file descriptor was provided in one of the sets
- EINTR - a signal was caught while waiting, and the call can be reissued
- EINVAL - the parameter n is negative, or the given timeout is invalid

- ENOMEM - insufficient memory was available to complete the request

select() example

- see page 50

Portable sleeping with select( )

- can be used as a more portable mechanism for subsecond sleep

```
/* sleep for 500 microseconds */
select (0, NULL, NULL, NULL, &tv);
```

pselect()

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>
int pselect (int n,
             fd_set *readfds,
             fd_set *writefds,
             fd_set *exceptfds,
             const struct timespec *timeout,
             const sigset_t *sigmask);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

- POSIX version of select with three main differences
  - uses timespec which theoretically provides nanosecond resolution over microsecond resolution but neither reliably provides microsecond resolution
  - does not modify the timeout parameter and does not require reinitialization
  - select does not have the sigmask parameter (when NULL behaves pselect behaves like select)
- the sigmask parameter attempts to solve a race condition (see pg 53 for more info)
- before 2.6.16 it was just a glibc wrapper but is now a system call

poll()

- System V multiplexed I/O solution with fixes for select() issues
- select is still used more often

```
#include <sys/poll.h>
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

- uses a single array of nfds pollfd structs

```
#include <sys/poll.h>
struct pollfd {
    int fd;
    short events;
    short revents;
};
```

- events are watched events
- revents are returned witnessed events
- both are bitmasked fields
- valid events:
    - POLLIN - there is data to read
    - POLLRDNORM - there is normal data to read
    - POLLRDBAND - there is priority data to read
    - POLLPRI - there is urgent data to read
    - POLLOUT - writing will not block
    - POLLWRNORM - writing normal data will not block
    - POLLWRBAND - writing priority data will not block
    - POLLMSG - a SIGPOLL message is available
- additional valid revents:
    - POLLER - error on the given file descriptor
    - POLLHUP - hung up event on the given file descriptor
    - POLLNVAL - the given file descriptor is invalid

Return values and error codes

- on success returns number of structs with non-zero events otherwise -1
- EBADF - an invalid file descriptor was given in one or more of the structures
- EFAULT - the pointer to fds pointed outside of the process' address space
- EINTR - a signal occurred before any requested event. The call may be reissued
- EINVAL - the nfds parameter exceeded the RLIMIT_NOFILE value
- ENOMEM - insufficient memory was available to complete the request

poll example()

- see page 55

ppoll()

- a pselect-like interface that is Linux only
- provides the same timeout options and sigmask options

poll() Versus select()

- poll is superior to select for these reasons:
    - no need to calculate and pass the highest fd
    - more efficient for large valued fds
    - select's fd sets are statically sized - poll can handle arrays of exactly the right size
    - select requires that sets are reconstructed and reinitialized on each return whereas poll separates the watched events and the returned events
- benefits of select:
    - select is more portable
    - select provides better timeout resolution
- prefer epoll over both (Linux specific)

Kernel Internals

The Virtual Filesystem

- used to allow Linux to access various filesystem types without being concerned for the details
- uses the common file model as the basis
- VFS uses inodes, superblocks, and directory entries and forces FAT and NTFS, etc to adapt
- allows for system calls to work with any filesystem on any medium

The Page Cache

- an in-memory store of recently accessed data from an on-disk filesystem
- exploits temporal locality to speed disk access

- first place that the kernel looks for filesystem data
- dynamically sized
- if increases past available memory, page cache pruning is implemented
- swap is used in place of pruning where less used in-memory pages are written to disk
- use /proc/sys/vm/swappiness to tune swap vs cache balance (0-100 - higher prefers swap)
- kernel also uses readahead to take advantage of sequential locality -> reads an additional chunk or two for reads in many cases
- managed dynamically, growing and reducing depending on the usage in a process
- can generally ignore these except to prefer not implementing a user space cache
- prefer sequential file I/O to random access

Page Writeback

- dictates when dirty buffers are written to disk
  - when free memory drops below a configurable threshold to facilitate removing dirty and free buffers
  - when a dirty buffer ages above a configurable threshold
- performed by pdflush threads
- pdflush use is governed for congestion avoidance
- buffers are represented by the buffer_head data structure
- in older Linux versions the buffer cache and the page cache were separate
- 2.4 unified the two
- risks data loss on power failure, something that can be avoided with synchronized I/O

Chapter 3 - Buffered I/O

- blocks and block size are the main focus of I/O
- poor block size choice can cause system performance degradation buffered I/O and the C standard library I/O functions handle much of this

User-Buffered I/O

- many small I/Os to regular files -> user-buffered I/O
- for example 2M written in 1byte chunks takes 18x longer than 2M in 1024 byte chunks which takes ~ .75x of 1130 byte chunks
- use stat() command to determine block size
- can use larger multiples of the block size to reduce system calls but do not do this for many small reads and writes
- can write buffering by hand but more common to use the C standard library

Standard I/O

- generally glibc with Linux implementing some deviations from the standard

File Pointers

- stdio routines use file pointers
- open file is a stream, input, output, and input/output

Opening Files

```
#include <stdio.h>
FILE * fopen (const char *path, const char *mode);
```

Modes

- r - start
- r+ - start (r + w)
- w - start + truncate or create
- w+ - start + truncate or create (w + r)
- a - end + create if necessary, writes append at end
- a+ - end + create if necessary, writes append at end (w + r)
- b - ignored on Linux (binary)

Opening a Stream via File Descriptor

- takes an already open fd

```
#include <stdio.h>
FILE * fdopen (int fd, const char *mode);
```

- do not use fd for I/O after

```
 FILE *stream;
int fd;
fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == &#8722;1)
    /* error */
stream = fdopen (fd, "r");
if (!stream)
    /* error */
```

Closing Streams

```
#include <stdio.h>
int fclose (FILE *stream);
```

- flushes before closing
- returns EOF on failure and sets errno

Closing All Streams

```
#define _GNU_SOURCE
#include <stdio.h>
int fcloseall (void);
```

- Linux only
- always returns 0

Reading from a Stream

- must be opened with standard I/O function in any mode other than w or a

Reading a Character at a Time

```
#include <stdio.h>
int fgetc (FILE *stream);
```

- casts to int to ensure EOF can be handled
- always store in int!

```
int c;
c = fgetc (stream);
if (c == EOF)
    /* error */
else
    printf ("c=%c\n", (char) c);
```

Putting the character back

```
#include <stdio.h>
int ungetc (int c, FILE *stream);
```

- returns c on success, EOF on failure
- POSIX guarantees success of 1, Linux allows infinite based on available memory
- calling seek in between will discard all pushed back chars

Reading an Entire Line

```
#include <stdio.h>
char * fgets (char *str, int size, FILE *stream);
```

- reads up to one less than size bytes from stream into str
- '\0' stored in buffer after bytes are read in
- reading stops after '\n' or EOF are reached
- returns NULL on failure
- LINE_MAX is defined in limits.h
- Linux is not limited by this but portable programs can use it for safety

Reading arbitrary strings

- fgets often causes issues

```
char *s;
int c;
s = str;
while (--n > 0 && (c = fgetc(stream)) != EOF)
    *s++ = c;
*s = '\0';
```

- read to delimiter

```
char *s;
int c = 0;
s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF && (*s++ = c) != d)
    ;
if (c == d)
    *--s = '\0';
```

```
else
    *s = '\0';
```

- probably slower but not for same reason as dd

Reading Binary Data

```
#include <stdio.h>
size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

- reads nr elements (each element of 'size' bytes) into buffer
- returns number of elements read, failure returns less than nr
- must use ferror() or feof() to determine type of failure

```
char buf[64];
size_t nr;
nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0)
    /* error */
```

Writing to a Stream

Issues of Alignment

- processors access memory in chunks of bytes (2, 4, 8, 16)
- can only read in integer multiples of this number
- consequently, C vars must be stored and accessed on alignment boundaries
- usually naturally aligned
- usually do not need to deal with alignment but saving binary data, dealing with memory management, and networking may make alignment important

Writing a Single Character

```
#include <stdio.h>
int fputc (int c, FILE *stream);
```

- casts int to unsigned char and returns c or EOF (w/ errno)

Writing a String of Characters

```
#include <stdio.h>
int fputs (const char *str, FILE *stream);
```

- returns non-negative number or EOF

```
FILE *stream;
stream = fopen("journal.txt", "a");
if (!stream)
    /* error */
if (fputs ("The ship is made of wood.\n", stream) == EOF)
    /* error */
if (fclose (stream) == EOF)
    /* error */
```

Writing Binary Data

```
#include <stdio.h>
size_t fwrite (void *buf, size_t size, size_t nr, FILE *stream);
```

- advances file pointer by total number of bytes written

Seeking a Stream

```
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
```

- whence - SEEK_SET - file position set to offset
- whence - SEEK_CUR - file position set to current position + offset
- whence - SEEK_END - file position set to end + offset

```
#include <stdio.h>
int fsetpos (FILE *stream, fpos_t *pos);
// with fgetpos();
```

- works like SEEK_SET above
- may be the only option on certain platforms

```
#include <stdio.h>
void rewind (FILE *stream);
```

- resets to start

Obtaining the Current Stream Position

```
#include <stdio.h>
long ftell (FILE *stream);
```

- returns -1 on error and sets errno

```
#include <stdioh.h>
int fgetpos (FILE *stream, fpos_t *pos);
```

- 0 on success, -1 w/ errno on failure

Flushing a Stream

```
#include <stdio.h>
int fflush (FILE *stream);
```

- writes unwritten data in user buffer to kernel buffer
- if passed NULL, all open input streams are flushed
- 0 on success, EOF + errno on failure
- a call to fflush followed by fsync will ensure a full write to disk

Errors and End-of-File

```
include <stdio.h>
int ferror (FILE *stream);
```

- checks for error on a stream, 0 for no error

```
include <stdio.h>
int feof (FILE *stream);
```

- checks for EOF on a stream, 0 for no error

```
#include <stdio.h>
void clearerr (FILE *stream);
```

- clears both error and EOF for a stream

Obtaining the Associated File Descriptor

```
#include <stdio.h>
int fileno (FILE *stream);
```

- returns fd or -1 with errno set to EBADF

Controlling the Buffering

- unbuffered - data goes straight to kernel (stderr by default)
- line-buffered - buffer submitted to kernel on each newline character (default for terminals)
- block-buffered - aka full-buffering, buffer passed to kernel at end of block (default for file streams)

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

- _IONBF - unbuffered
- _IOLBF - line-buffered
- _IOFBF - block buffered
- must be called on open stream
- must close the stream before a buffer falls out of scope

Thread Safety

- must synchronize access to shared data between threads or make the data thread-local
- use mutual exclusion but not always adequate
- may need to block (expand the 'critical region') on many calls at once
- may need lock-free safety for access
- stdio functions are inherently thread safe (use lock, lock count, and owning thread)
- within single function calls, stdio function calls are atomic

Manual File Locking

```
#include <stdio.h>
void flockfile (FILE *stream);
```

```
#include <stdio.h>
void funlockfile (FILE *stream);
```

- just increments the lock count and waits for a file to become free or decrements lock count

```
#include <stdio.h>
int ftrylockfile (FILE *stream);
```

- only locks if stream is unlocked or does nothing and returns non-zero value

```
flockfile (stream);
fputs ("List of treasure:\n", stream);
fputs ("    (1) 500 gold coins\n", stream);
fputs ("    (2) Wonderfully ornate dishware\n", stream);
funlockfile (stream);
```

Unlocked Stream Operations

- Linux provides "unlocked" cousins to stdio functions to allow an application programmer to manually manage file locking to improve performance

```
#define _GNU_SOURCE
#include <stdio.h>
int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf, size_t size, size_t nr,
                       FILE *stream);
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr,
                        FILE *stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked (FILE *stream);
int fileno_unlocked (FILE *stream);
void clearerr_unlocked (FILE *stream);
```

- POSIX has some but these are for Linux

Critiques of Standard I/O

- some bad functions
- do not use gets
- double copy has performance impact
- highly optimized user buffering libraries exist

Chapter 4 - Advanced File I/O

Scatter/Gather I/O

- 1 system call writes to a vector of buffers from 1 data stream or a vector of buffers into 1 stream (aka vectored I/O)
  - natural for cases that match
  - efficient and performant
  - atomic - no interleaving
- can be achieved manually

readv() and writev()

```
#include <sys/uio.h>
ssize_t readv (int fd, const struct iovec *iov, int count);
```

```
#include <sys/uio.h>
ssize_t writev (int fd, const struct iovec *iov, int count);
```

```
include <sys/uio.h>
struct iovec {
    void *iov_base;    /* pointer to start of buffer */
    size_t iov_len;    /* size of buffer in bytes */
};
```