Chapter 2 - UNIX Standardization and Implementations

- ISO C standard - used for all UNIX systems
- 24 areas based on the following headers
    - assert.h - veriy program functionality with assertions
    - complex.h - complex arithmetic
    - ctype.h - character types
    - errno.h - error codes
    - fenv.h - floating point environment
    - float.h - floating point constants
    - inttypes.h - integer type format conversion
    - iso646.h - alternate relational operator macros
    - limits.h - implementation constants
    - locale.h - locale categories
    - math.h - mathematical constants
    - setjmp.h - nonlocal got statements
    - signal.h - signals
    - stdarg.h - variable argument lists
    - stdbool.h - boolean types and values
    - stddef.h - standard definitions
    - stdint.h - integer types
    - stdlib.h - utility functions
    - string.h - string operations
    - tgmath.h - type-generic math macros
    - time.h - time and date
    - wchar.h - extended multibyte and wide character support
    - wctype.h - wide character classification and mapping support

IEEE POSIX

- Portable Operating System Interface
- Institute of Electrical and Electronics Engineers
- define additional C OS standards
    - dirent.h - directory entries (Section 4.21)

- fcntl.h - file control (Section 3.14)
- fnmatch.h - filename-matching types
- glob.h - pathname pattern-matching types
- grp.h - group file (Section 6.4)
- netdb.h - network database operations
- pwd.h - password file (Section 6.2)
- regex.h - regular expressions
- tar.h - tar archive values
- termios.h - terminal I/O (Chapter 18)
- unistd.h - symbolic constants
- utime.h - file times (Section 4.19)
- wordexp.h - word-expansion types
- arpa/inet.h - Internet definitions (Chapter 16)
- net/if.h - socket local interfaces (Chapter 16)
- netinet/in.h - Internet address family (Section 16.3)
- netinet/tcp.h - Transmission Control Protocol definitions
- sys/mman.h - memory management declarations
- sys/select.h - select function (Section 14.5.1)
- sys/socket.h - sockets interface (Chapter 16)
- sys/stat.h - file status (Chapter 4)
- sys/times.h - process times (Section 8.16)
- sys/types.h - primitive system data types (Section 2.8)
- sys/un.h - UNIX domain socket definitions (Section 17.3)
- sys/utsname.h - system name (Section 6.9)
- sys/wait.h - process control (Section 8.6)
- cpio.h - cpio archive values
- dlfcn.h - dynamic linking
- fmtmsg.h - message display structures
- ftw.h - file tree walking (Section 4.21)
- iconv.h - codeset conversion utility
- langinfo.h - language information constants
- libgen.h - definitions for pattern-matching function
- monetary.h - monetary types
- ndbm.h - database operations
- nl_types.h - message catalogs
- poll.h - poll function (Section 14.5.2)
- search.h - search tables
- strings.h - string operations

    - syslog.h - system error logging (Section 13.4)
    - ucontext.h - user context
    - ulimit.h - user limits
    - utmpx.h - user accounting database
    - sys/ipc.h - IPC (Section 15.6)
    - sys/msg.h - message queues (Section 15.7)
    - sys/resource.h - resource operations (Section 7.11)
    - sys/sem.h - semaphores (Section 15.8)
    - sys/shm.h - shared memory (Section 15.9)
    - sys/statvfs.h - file system information
    - sys/time.h - time types
    - sys/timeb.h - additional date and time definitions
    - sys/uio.h - vector I/O operations (Section 14.7)
    - aio.h - asynchronous I/O
    - mqueue.h - message queues
    - pthread.h - threads (Chapters 11 and 12)
    - sched.h - execution scheduling
    - semaphore.h - semaphores
    - spawn.h - real-time spawn interface
    - stropts.h - XSI STREAMS interface (Section 14.4)
    - trace.h - event tracing
- POSIX symbolic constants are defined to define which groups of POSIX APIs are available (in unistd.h)
    - _POSIX_ADVISORY_INFO - advisory information (real-time)
    - _POSIX_ASYNCHRONOUS_IO - asynchronous input and output (real-time)
    - _POSIX_BARRIERS - barriers (real-time)
    - _POSIX_CPUTIME - process CPU time clocks (real-time)
    - _POSIX_CLOCK_SELECTION - clock selection (real-time)
    - _POSIX_FSYNC - extension to ISO C standard
    - _POSIX_IPV6 - file synchronization IPv6 interfaces
    - _POSIX_MAPPED_FILES - memory-mapped files
    - _POSIX_MEMLOCK - process memory locking (real-time)
    - _POSIX_MEMLOCK_RANGE - memory range locking (real-time)
    - _POSIX_MONOTONIC_CLOCK - monotonic clock (real-time)
    - _POSIX_MEMORY_PROTECTION - memory protection
    - _POSIX_MESSAGE_PASSING - message passing (real-time)
    - _POSIX_PRIORITIZED_IO - IEC 60559 floating-point option prioritized input and output

    - _POSIX_PRIORITIZED_SCHEDULING - process scheduling (real-time)
    - _POSIX_RAW_SOCKETS - raw sockets
    - _POSIX_REALTIME_SIGNALS - real-time signals extension
    - _POSIX_SEMAPHORES - semaphores (real-time)
    - _POSIX_SHARED_MEMORY_OBJECTS - shared memory objects (real-time)
    - _POSIX_SYNCHRONIZED_IO - synchronized input and output (real-time)
    - _POSIX_SPIN_LOCKS - spin locks (real-time)
    - _POSIX_SPAWN - spawn (real-time)
    - _POSIX_SPORADIC_SERVER - process sporadic server (real-time)
    - _POSIX_THREAD_CPUTIME - thread CPU time clocks (real-time)
    - _POSIX_TRACE_EVENT_FILTER - trace event filter
    - _POSIX_THREADS - threads
    - _POSIX_TIMEOUTS - timeouts (real-time)
    - _POSIX_TIMERS - timers (real-time)
    - _POSIX_THREAD_PRIO_INHERIT - thread priority inheritance (real-time)
    - _POSIX_THREAD_PRIO_PROTECT - thread priority protection (real-time)
    - _POSIX_THREAD_PRIORITY_SCHEDULING - thread execution scheduling (real-time)
    - _POSIX_TRACE - trace
    - _POSIX_TRACE_INHERIT - trace inherit
    - _POSIX_TRACE_LOG - trace log
    - _POSIX_THREAD_ATTR_STACKADDR - thread stack address attribute
    - _POSIX_THREAD_SAFE_FUNCTIONS - thread-safe functions
    - _POSIX_THREAD_PROCESS_SHARED - thread process-shared synchronization
    - _POSIX_THREAD_SPORADIC_SERVER - thread sporadic server (real-time)
    - _POSIX_THREAD_ATTR_STACKSIZE - thread stack address size
    - _POSIX_TYPED_MEMORY_OBJECTS - typed memory objects (real-time)
    - _XOPEN_UNIX - X/Open extended interfaces
    - _XOPEN_STREAMS - XSI STREAMS
- the single UNIX specification is a superset of POSIX and adds APIs
    - Encryption: denoted by the _XOPEN_CRYPT symbolic constant
    - Real-time: denoted by the _XOPEN_REALTIME symbolic constant
    - Advanced real-time
    - Real-time threads: denoted by the _XOPEN_REALTIME_THREADS symbolic constant
      Advanced real-time threads
    - Tracing
    - XSI STREAMS: denoted by the _XOPEN_STREAMS symbolic constant
    - Legacy: denoted by the _XOPEN_LEGACY symbolic constant

- Major UNIX systems
    - System V Release 4
    - BSD
    - FreeBSD
    - Linux
    - Mac OSX
    - Solaris
- limits.h for size limits, etc
- runtime limits are determined using sysconf, pathconf, and fpathconf with specific arguments for queries
- may need to use PATH_MAX, OPEN_MAX, etc
- many other limits, macros, feature test macros, etc outlined in chapter 2

Common Primitive Data Types

- caddr_t - core address (Section 14.9)
- clock_t - counter of clock ticks (process time) (Section 1.10)
- comp_t - compressed clock ticks (Section 8.14)
- dev_t - device numbers (major and minor) (Section 4.23)
- fd_set - file descriptor sets (Section 14.5.1)
- fpos_t - file position (Section 5.10)
- gid_t - numeric group IDs
- ino_t - i-node numbers (Section 4.14)
- mode_t - file type, file creation mode (Section 4.5)
- nlink_t - link counts for directory entries (Section 4.14)
- off_t - file sizes and offsets (signed) (lseek, Section 3.6)
- pid_t - process IDs and process group IDs (signed) (Sections 8.2 and 9.4)
- ptrdiff_t - result of subtracting two pointers (signed)
- rlim_t - resource limits (Section 7.11)
- sig_atomic_t - data type that can be accessed atomically (Section 10.15)
- sigset_t - signal set (Section 10.11)
- size_t - sizes of objects (such as strings) (unsigned) (Section 3.7)
- ssize_t - functions that return a count of bytes (signed) (read, write, Section 3.7)
- time_t - counter of seconds of calendar time (Section 1.10)
- uid_t - numeric user IDs
- wchar_t - can represent all distinct character codes

Chapter 3 - File I/O

- describes unbuffered I/O, i.e. each call invokes a system call (not ISO C but POSIX)

File Descriptors

- int as a file handle
- 0 - STDIN_FILENO
- 1 - STDOUT_FILENO
- 2 - STDERR_FILENO
- 0 - OPEN_MAX
- 1024 or 1048576

open Function

```
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

- open flags
    - O_RDONLY - Open for reading only
    - O_WRONLY - Open for writing only
    - O_RDWR - Open for reading and writing
- optional flags
    - O_APPEND
    - O_CREAT
    - O_EXCL
    - O_TRUNC
    - O_NOCTTY
    - O_NONBLOCK
- synchronized I/O
    - O_DSYNC
    - O_RSYNC
    - O_SYNC

Filename and Pathname Truncation

- If _POSIX_NO_TRUNC is in effect, errno is set to ENAMETOOLONG, and an error status is returned if the entire pathname exceeds PATH_MAX or any filename component of the pathname exceeds NAME_MAX

creat Function

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

- same as

```
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

close Function

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

lseek Function

- used to explicitly set the current file offset (0 by default on open unless O_APPEND is specified)

```
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
```

- the offset argument is interpreted differently depending on whence
    - If whence is SEEK_SET, the file's offset is set to offset bytes from the beginning of the file.
    - If whence is SEEK_CUR, the file's offset is set to its current value plus the offset. The offset can be positive or negative.
    - If whence is SEEK_END, the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.
- can use lseek with 0 to find the current position

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

- if a file cannot be seeked because of the type then it returns a 1 (-1 ?)

read Function

```
#include <unistd.h>
ssize_t read(int filedes, void *buf, size_t nbytes);
```

- nbytes if successful
- 0 if EOF
- When reading from a regular file, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
- When reading from a terminal device. Normally, up to one line is read at a time. (We'll see how to change this in Chapter 18.)
- When reading from a network. Buffering within the network may cause less than the requested amount to be returned.
- When reading from a pipe or FIFO. If the pipe contains fewer bytes than requested, read will return only what is available.
- When reading from a record-oriented device. Some record-oriented devices, such as magnetic tape, can return up to a single record at a time.
- When interrupted by a signal and a partial amount of data has already been read. We discuss this further in Section 10.5.
- older signature (before POSIX.1)

```
int read(int filedes, char *buf, unsigned nbytes);
```

write Function

```
#include <unistd.h>
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

- nbytes for success, otherwise error

I/O Efficiency

- buffer size choice depends on the filesystem

- look at st_blksize (ext2 - 4096)
- read ahead means the system will try to predict reads after sequential reads and read in more data than requested to improve performance
- for ext2, performance improvements seem to stop after a buffer size of 128K
- measuring read performance can be misleading due to system caching
- to avoid this use a different copy of the file on each read

File Sharing

- following implementations may vary depending on OS
- every process has an entry in the process table with each process table entry containing a table of open file descriptors
    - each entry contains:
        - file descriptor flags
        - a pointer to a file table entry

```
process table:
        fd flags    file pointer
fd 0:   ________    _____________ -> points to a file table
fd 1:   ________    _____________
fd 2:   ________    _____________

file table:
single entry:
file status flags:      _____
current file offset:    _____
v-node pointer:         _____ -> points to a v-node table

v-node table:
single entry:
v-node information:     _____
i-node information:     _____
current file size:      _____
```

- the kernel maintains a file table for all open file tables
    - each entry contains:
        - the file status flags for the file (i.e. read, write, append, etc)
        - the current file offset
        - the pointer to the v-node table entry for the file
- each file table entry points to a v-node structure
    - a v-node structure contains
        - info about the type of file
        - pointers to the functions that operate on the file

        - usually pointers to the i-node for the file (see i-node info later)
- for writes -> the offset is incremented by the number of bytes written
    - if the offset exceeds the current file size, the file size is set to the current offset
- if a file is opened with O_APPEND -> sets the file status flag entry in the file table
    - for each write -> the offset is set to the file size before writing
- if lseek seeks to the end of the file then the offset is set to the file size
    - NOTE: this is distinct functionality from the O_APPEND functionality
- lseek only modifies the offset, no I/O is performed
- it is possible for more than one file descriptor to point to the same file table entry
- multiple processes opening the same file results in each having its own file table entry
- when multiple processes write to the same file issues can arise -> this is where atomic operations come in

Atomic Operations

Appending to a File

- in older systems it required both a seek and a write
- if two processes are working on the same file, these might become interleaved resulting in undefined behavior
- using O_APPEND makes this atomic

pread and pwrite Functions

- Single UNIX and XSI offer pread and pwrite to perform seek and read/write atomically

```
#include <unistd.h>
ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);

ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);
```

Creating a File

- if open with O_CREAT or creat did not exist then the open call would need to be called and checked before creating the file
- if this is split and two processes try to create the same file errors may occur

dup and dup2 Functions

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

- dup returns the lowest possible file descriptor
- dup2 returns the specified file descriptor, closing it if it is already open
- can also use the fcntl function to duplicate a file descriptor

```
dup(filedes);
// equivalent to
fcntl(filedes, F_DUPFD, 0);

dup2(filedes, filedes2);
// roughly equivalent to
close(filedes2);
fcntl(filedes, F_DUPFD, filedes2);
```

- differences:
  - dup2 is atomic
  - some errno difference between the two

sync, fsync, and fdatasync Functions

- most UNIX Systems have a buffer cache or page cache that I/O passes through
- fsync, fdatasync, and sync provide the ability to write the kernel buffers to disk

```
#include <unistd.h>
int fsync(int filedes);
int fdatasync(int filedes);
void sync(void);
```

- sync queues all dirty buffers for writing and returns without waiting
- usually called about every 30 seconds by a system daemon called update (mirrored by sync command)
- fsync writes buffers to disk for one file and waits for the writes before returning
- fdatasync just guarantees writing of the data portions of a file (not necessarily the metadata portions of a file)
- FreeBSD and Mac OSX do not support fdatasync

fcntl Function

- modify the properties of an open file via its file descriptor

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, ... /* int arg */ );
```

- used for the following:

  i. Duplicate an existing descriptor (cmd = F_DUPFD)
  ii. Get/set file descriptor flags (cmd = F_GETFD or F_SETFD)
  iii. Get/set file status flags (cmd = F_GETFL or F_SETFL)
  iv. Get/set asynchronous I/O ownership (cmd = F_GETOWN or F_SETOWN)
  v. Get/set record locks (cmd = F_GETLK, F_SETLK, or F_SETLKW)

- cmd flags:
  - F_DUPFD - 0 - duplicate file descriptor
  - F_GETFD - 1 - get file descriptor flags
  - F_SETFD - 2 - set file descriptor flags
  - F_GETFL - 3 - get file status flags
  - F_SETFL - 4 - set file status flags
  - F_GETOWN - 5 - get SIGIO/SIGURG proc/pgrp
  - F_SETOWN - 6 - set SIGIO/SIGURG proc/pgrp
  - F_GETLK - 7 - get record locking information
  - F_SETLK - 8 - set record locking information
  - F_SETLKW - 9 - F_SETLK; wait if blocked
- need to first get status flags and modify from there

ioctl Function

- a catchall for I/O operations not previously described
- previously used for terminal I/O (this was replaced by other POSIX functions)

```
#include <unistd.h> /* System V */
#include <sys/ioctl.h> /* BSD and Linux */
#include <stropts.h> /* XSI STREAMS */
int ioctl(int filedes, int request, ...);
```

- the prototype is different depending on the system

- normally, additional device specific ioctl calls require additional headers
  - disk labels DIOxxx <sys/disklabel.h>
  - file I/O FIOxxx <sys/filio.h>
  - mag tape I/O MTIOxxx <sys/mtio.h>
  - socket I/O SIOxxx <sys/sockio.h>
  - terminal I/O TIOxxx <sys/ttycom.h>

/dev/fd

- entries in /dev/fd are 0, 1, 2, etc and opening them is the same as opening fd 0, 1, 2, etc.
- usage:

```
fd = open("/dev/fd/0", mode);
```

- mode must be a subset of the original open mode

```
// equivalent to
fd = dup(0);
```

- contradictions between this and the original open modes will be controlled by the original open mode

Chapter 4 - Files and Directories

stat, fstat, and lstat Functions

```
#include <sys/stat.h>
int stat(const char *restrict pathname, struct stat *restrict buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *restrict pathname, struct stat *restrict buf);
```

- returns a structure of information about a path

```
struct stat {
    mode_t              st_mode;                /* [XSI] Mode of file (see below)
    ino_t       st_ino;             /* [XSI] File serial number */
        dev_t           st_dev;                 /* [XSI] ID of device containing f
        dev_t           st_rdev;                /* [XSI] Device ID */
        nlink_t         st_nlink;               /* [XSI] Number of hard links */
        uid_t           st_uid;                 /* [XSI] User ID of the file */
        gid_t           st_gid;                 /* [XSI] Group ID of the file */
        off_t           st_size;                /* [XSI] file size, in bytes */
        time_t      st_atime;       /* time of last access */
```

```
    time_t      st_mtime;       /* time of last modification */
    time_t      st_ctime;       /* time of last file status change */
        blksize_t       st_blksize;             /* [XSI] optimal blocksize for I/O
        blkcnt_t        st_blocks;              /* [XSI] blocks allocated for file

        // Additional
    uint32_t    st_flags;               /* user defined flags for file */
        uint32_t        st_gen;                 /* file generation number */
};
```

- used most commonly by ls -l commands

File Types

1. Regular file - most common, text or binary. Binary needs extra information to pass to the kernel about the location of text/code and data
2. Directory file - a file that contains names of other files and pointers to the file info
3. Block special file - buffered I/O access in fixed blocks to devices such as disk drives
4. Character special file - unbuffered I/O access in variable size units to devices NOTE: all device access is either block or character
5. FIFO - (named pipe) used for interprocess communication
6. Socket - used for network communication between processes. Also used for communication between processes on the same host, i.e. interprocess communication
7. Symbolic link - a file that points to another file

- file type is encoded in the st_mode field
- Macros (take the st_mode field)
    - S_ISREG() - regular file
    - S_ISDIR() - directory file
    - S_ISCHR() - character special file
    - S_ISBLK() - block special file
    - S_ISFIFO() - pipe or FIFO
    - S_ISLNK() - symbolic link
    - S_ISSOCK() - socket
- IPC file type macros (take a pointer to the full stat structure)
    - S_TYPEISMQ() - message queue
    - S_TYPEISSEM() - semaphore
    - S_TYPEISSHM() - shared memory object
- the first set of macros just logically & the st_mode value with the S_IFMT mask and compare the result with the constants

Set-User-ID and Set-Group-ID

- real user/group - the actual user and group set on login - only possible to change by a superuser process
- effective user/group - determine the file access permissions
- saved user/group - passed to exec processes
- when a file is executed, if the file has superuser privileges then the process has superuser privileges regardless of the real user ID

File Access Permissions

- S_IRUSR - user-read
- S_IWUSR - user-write
- S_IXUSR - user-execute
- S_IRGRP - group-read
- S_IWGRP - group-write
- S_IXGRP - group-execute
- S_IROTH - other-read
- S_IWOTH - other-write
- S_IXOTH - other-execute
- access rules:
    - to open must have access permission, plus all directories in path
    - the read permission for a file determines whether user can open an existing file for reading: the O_RDONLY and O_RDWR flags for the open function
    - the write permission for a file determines whether the user can open an existing file for writing: the O_WRONLY and O_RDWR flags for the open function
    - must have write permission for a file to specify the O_TRUNC flag in the open function
    - cannot create a new file in a directory unless we have write permission and execute permission in the directory
    - to delete an existing file, we need write permission and execute permission in the directory containing the file. User does not need read permission or write permission for the file itself.
    - execute permission for a file must be on if we want to execute the file using any of the six exec functions. The file also has to be a regular file

- kernel combines tests for process user access as follows:
    - superuser (0) has full access through entire filesystem
    - if eid of process == id of file owner access is allowed if the appropriate access bit is set and denied otherwise
    - if gid of process == gid of the file access is allowed if the appropriate bit is set
    - if the appropriate other acces bit is set access is allowed

Ownership of New Files and Directories

- file and directory ownership rules are identical for creation
- the uid of a new file == eid of creating process
- for gid:
    - the gid can be effective gid of the process
    - the gid of the file can be the gid of the containing directory

access Function

- provides a means for a process to test for access

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

- mode:

```
mode    description
----    -----------
R_OK    test for read permission
W_OK    test for write permission
X_OK    test for execute permission
F_OK    test for existence of file
```

umask Function

- sets the the file mode creation value and returns the previous value
- does not return errors

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

- formed with bitwise or of the nine acces constants
- not usually changed, must be changed when running
- shell also supports a symbolic form (i.e. u=rwx,g=rwx,o=rwx

chmod and fchmod Functions

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int filedes, mode_t mode);
```

- chmod takes a path, fchmod takes an open file by file descriptor
- uses additional constants
    - S_ISUID - set-user-ID on execution
    - S_ISGID - set-group-ID on execution
    - S_ISVTX - saved-text (sticky bit)
    - S_IRWXU - read, write, and execute by user (owner)
    - S_IRUSR - read by user (owner)
    - S_IWUSR - write by user (owner)
    - S_IXUSR - execute by user (owner)
    - S_IRWXG - read, write, and execute by group
    - S_IRGRP - read by group
    - S_IWGRP - write by group
    - S_IXGRP - execute by group
    - S_IRWXO - read, write, and execute by other (world)
    - S_IROTH - read by other (world)
    - S_IWOTH - write by other (world)
    - S_IXOTH - execute by other (world)
- the chmod function automatically sets bits when:
    - an attempt is made to set the sticky bit on a regular file without superuser privileges it is automatically turned off
    - it is possible for the gid for a new file to be a group that the user of the calling process does not belong to. In this case the set-group_ID bit is automatically turned off
- NOTE:

Sticky Bit

- has a weird history
- the sticky bit is usually set on directories to indicate that only the file's owner, the directory's owner, or root can rename or delete the file
- /tmp and /var/spool/mail etc are good candidates for the sticky bit

chown, fchown, and lchown Functions

- change the user ID and gid of a file

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int filedes, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

- each are roughly the same but lchown will not modify the file pointed to by a symbolic link
- if the _POSIX_CHOWN_RESTRICTED is in effect for a file
    - only superuser can change the uid of the file
    - a nonsuperuser process can changed the gid of a file if the process owns the file

File Size

- use the st_size field of a stat structure
- only meaningful for a regular file, directory, or symbolic link
- for a directory the size is usally the multiple of a number
- for a symlink the file size is the number bytes in the name
- modern Unix systems provide
    - st_blksize - preferred block size for file I/O
    - st_blocks - actual number of 512 byte blocks allocated for a file (different for different systems)

Holes in a File

- stat and du will have a discrepancy if holes are in a file

File Truncation

```
#include <unistd.h>
int truncate(const char *pathname, off_t length);
int ftruncate(int filedes, off_t length);
```

- setting it to larger than the current file size is system dependent
- setting it to smaller than the current file then the data beyond the new size is inaccessible

File Systems

- most Unix systems support many different types of filesystems

```
disk drive:
_____________________________________________
|           |                   |           |
| Partition |     Partition     | Partition |
|___________|___________________|___________|

partition (contains a filesystem):
_____________________________________________________________
|   |   |               |               |   |                |
|   |   |Cylinder Grp 0 |Cylinder Grp 1 |...|Cylinder Grp n  |
|___|___|_______________|_______________|___|________________|
Boot block(s)
    super block

cylinder group:
_____________________________________________________________
|   |   |   |   |                       |                   |
|   |   |   |   |       i-nodes         |   data blocks     |
|___|___|___|___|_______________________|___________________|
super block copy
    cylinder group info
        i-node map
            block bitmap

i-nodes section:
_____________________________________________________________
|               |               |           |                |
|   i-node      |   i-node      |   ...     |   i-node       |
|_______________|_______________|___________|________________|
```

```
i-nodes and data blocks in more detail
_____________________________________________________________
|               |   |       |   |       |   |       |       |
| i-node array  |...|       |...|       |...|       |       |
|_______________|___|_______|___|_______|___|_______|_______|
```

```
                    data block              data block
                                directory block     etc

i-node array
____________________
|       |       |   |
|i-node |i-node |...|
|_______|_______|___|
an i-node will contain
pointers to various data blocks (above)

directory block
_____________________________________
|   |       |   |   |       |   |
|   |       |...|   |       |...|
|___|_______|___|___|_______|___|
i-node number   i-node number
    filename        filename
```

```
Additional info about UFS (from https://docs.oracle.com/cd/E19683-01/806-4073/6jd6
Cylinder group - a disk slice is divided into these groups which are made up
of one or more disk cylinders

Boot block - objects used to boot the filesystem (blank if unneeded)
Only appears in cylinder group 0 as the first 8KB in a slice

Superblock - duplicated across each cylinder group and offset by a different amoun
(so appears on each platter of a drive)
    - size and status of the filesystem
    - label (filesystem name and volume area)
    - size of the system logical block
    - date and time of the last update
    - cylinder group size
    - number of data blocks in a cylinder group
    - summary data block
    - file system state
    - path name of the last mount point

Inodes - contains info about a file except its name (kept in directory)
- type of file
    - regular
    - directory
    - block special
    - character special
    - FIFO or named pipe
    - symbolic link
    - socket
    - other inodes - attribute directory and shadow (used for ACLs)
- file permissions mode
- number of hard links
- UID
- GID
- size in bytes
- 15 disk block array
- last access timestamp
```

```
- last modified timestamp
- created timestamp

The disk block array is arranged differently depending on the size
1 - 12 - direct addresses
13 - points to an indirect block (if larger than the first 12)
14 - a double indirect block (if larger)
15 - a triple indirect block (if larger)


__________
0 - 11 -> storage blocks
12 -> indirect block -> storage blocks
13 -> double indirect block -> indirect block -> storage blocks
                            -> indirect block -> storage blocks
14 -> triple indirect block -> double indirect block -> indirect block -> storage
                                                     -> indirect block -> storage
                            -> double indirect block -> indirect block -> storage
                                                     -> indirect block -> storage
                            -> double indirect block -> indirect block -> storage
                                                     -> indirect block -> storage

data blocks or storage blocks - on UFS allocated in 8KB logical block sizes and
1KB fragment size. For a regular file, data blocks contain file contents. For
directories, data blocks contain entries with the inode number and name of files.

free blocks - unused blocks are marked as free in the cylinder group map (which al
fragments)

NOTE: the ordering is not as above for each cylinder group, it is modular with the
superblock starting at a different offset depending on the cylinder group
```

- two directory entries can point to the same inode (a hard link)
- a symbolic link's file contents are the name of the file that the symbolic link points to
- i-node contains all of the information about the file (see above). i-node number data type is ino_t
- i-node numbers have to point to an i-node in the same filesystem (hence the hard link limitations)
- when moving a file within the same filesystem, all that needs to be done is to create a link in the new location and unlink the old location
- NOTE: a directory entry does not contain the full path, just a single name
- a directory link adds an entry to the containing directory with the i-node number and base name of the directory, which points to the i-node, which points to the linked directories block

!!!! Missing a few sections here

Chapter 5 - Standard I/O Library

- ISO C and extenstions, covers over buffer allocation and chunk size optimization
- written in roughly 1975

Streams and FILE Objects

- rather than working with a file descriptor considers a file handle to be a stream
- with ASCII - single char = 1 byte, international char > 1 byte
- on creation, stream has no orientation then it can be specified
- use fwide to set a stream's orientation

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *fp, int mode);
```

- arg - negative - tries to make the specified stream byte-oriented
- arg - positive - tries to make the specified stream wide-oriented
- arg - zero - tries to set the orientation, but will still return a value identifying the stream's orientation.
- Returns: positive if stream is wide-oriented, negative if stream is byte-oriented, or 0 if stream has no orientation

Standard Input, Standard Output, and Standard Error

- fd versions - STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO
- file pointer versions - stdin, stdout, and stderr
- defined in the <stdio.h> header
- (usually 0, 1, 2)

Buffering

- goal is to minimize read and write calls and automate it, covering over application programmer's need to handle it
- causes confusion

- three types:
    - fully buffered - I/O takes place when the standard I/O buffer is filled. Buffer usually created with malloc. Flushes on fill or fflush call. In a terminal driver flushing discards the data.
    - line buffered - I/O on newline char. Usually used for terminals. NOTE: max line is fixed and will cause I/O. NOTE: I/O requests from unbuffered or line buffered streams will cause a flush
    - unbuffered - no buffering (generally stderr is unbuffered)
- stdin and stdout are generally fully buffered for non-interactive devices
- stderr is never fully buffered and generally always unbuffered
- all other streams (stdin and stdout included) are line buffered if they are a terminal device and unbuffered otherwise

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

- call after stream is open and set buffering
    - _IOFBF - fully buffered
    - _IOLBF - line buffered
    - _IONBF - unbuffered
- NOTE: see summary 5.1 for details

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

- force a stream to be flushed

Opening a Stream

```
#include <stdio.h>
FILE *fopen(const char *restrict pathname, const char *restrict type);
FILE *freopen(const char *restrict pathname, const char *restrict type, FILE *rest
FILE *fdopen(int filedes, const char *type);
```

- fopen - opens file
- freopen - opens a file on a specified stream, clearing orientation if necessary
- fdopen - creates a stream from an open file descriptor NOTE: streams and other special file types cannot be opened with fdopen and must be opened and then passed to fdopen

- open type arg:
  - r or rb - read
  - w or wb - write
  - a or ab - append (write to end of file), create if necessary
  - r+, r+b, rb+ - read and write
  - w+, w+b, wb+ - truncate or create and read or write
  - a+, a+b, ab+ - open or create for reading and writing from the end of the file
- fdopen alters type args - truncate does not work as expected and more
- for read and write
  - need to call fflush, fseek, fsetpos, or rewind between a read/write switch or when EOF is encountered

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

- use fclose to close an open stream, flushes before closing ad releases buffer
- on process exit, all streams are flushed and closed

Reading and Writing a Stream

- three types of I/O
  - char - read and write on character at a time (if buffered, buffered behind scenes)
  - line - use fgets or fputs
  - direct - fread/fwrite - read or write in a specified size (corollary to reading blocks) NOTE: aka binary I/O, object-at-a-time I/O, record-oriented I/O, structure-oriented I/O
- character input:

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

- getchar == getc(stdin)
- getc can be macro, fgetc cannot
- getc arg should have no side effects
- fgetc has an address and can be passed to other functions
- fgetc calls usually take longer
- returns character converted to int

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

```
int feof(FILE *fp);
void clearerr(FILE *fp);
```

- Both return: nonzero (true) if condition is true, 0 (false) otherwise
- each file pointer generally maintains:
  - error flag
  - EOF flag
- clear with clearerr

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

- use ungetc to push a character back to a stream (only for files that will accept this) NOTE: does not write back to file, just pushes back to kernel buffer
- output

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

- same semantics as above

Line-at-a-time I/O

```
#include <stdio.h>
char *fgets(char *restrict buf, int n, FILE *restrict fp);
char *gets(char *buf);
```

- fgets takes max for line size and stores newline in buffer
- !!! do not use gets - causes buffer overflow

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

- fputs does not write null character or newline
- puts is not unsafe but avoid using it in favor of fputs

I/O Efficiency

- fastest to slowest - fgets, getc, fgetc

Binary I/O