Lecture # 3 – Files and Directories (Chapters 4)

- Stat, fstat, and lstat functions

```
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);                    # on open files
int lstat(const char *pathname, struct stat buf);            # for links
```

Returns 0 if OK, -1 on error

Some key elements of the stat structure:

st_mode - file type and permissions
st_uid - UID of the owner
st_gid - GID of the group owner
st_size - file size in bytes
st_blocks – number of allocated blocks
st_atime - last access time
st_mtime - last modification time
st_ctime - last inode change time

- File Types

1. Regular file - S_ISREG()
2. Directory file - S_ISDIR()
3. Character special file - S_ISCHR()
4. Block special file - S_ISBLK()
5. FIFO - S_ISFIFO()
6. Socket - S_ISSOCK()
7. Symbolic link - S_ISLNK()

Example:

```
struct stat        buf;

if (lstat("/tmp/file.txt", &buf) < 0)
{
        perror("lstat error");
        exit(1);
}

if (S_ISREG(buf.st_mode))          printf("regular file\n");
else if (S_ISDIR(buf.st_mode))     printf("directory\n");
        <etc…>
else                               printf("unknown file type\n");
```

- Set User Id and Set Group Id

  There are several ID's associated with a running process:

  Real user ID / group ID
  Effective user ID / group ID (Supplementary group ID's)
  Saved user ID / group ID

  Real UID/GID is who we really are (i.e. from the passwd file for our account)
  Effective UID/GID are used for resource permission checks (i.e. file access)

  Normally, effective UID/GID is equal to real UID/GID.

  There are two bits in the mode (st_mode) called set UID, and set GID.

  When the SUID bit is set and the file is executed, the effective UID is set to the owner of the file, not the UID of the person logged in.

  When the SGID bit is set and the file is executed, the effective GID is set to the group owner of the file, not the GID of the person logged in.

  Setting SUID / SGID:

  SUID has a value of 4, SGID has a value of 2
  These two form an optional digit to chmod with a max value of 6.

```
chmod 4755 filename     # Set SUID bit (rwsr-xr-x)
chmod 2755 filename     # Set SGID bit (rwxr-sr-x)
chmod 6755 filename     # Set SUID and SGID (rwsr-sr-x)
```

- File Access Permissions

St_mode mask	Meaning
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

  st_mode also encodes file permission bits.
  9 bits for permissions, 4 for file type, 1 for SUID, 1 for SGID, 1 for sticky bit

Note: Review of file access rules particularly for directories

File access tests:

1. If the EUID of the process is 0 (root), access is allowed.
2. If the EUID of the process equals the owner of the file and the appropriate access bit is set then access is allowed, and denied otherwise.
3. If the EGID and a supplementary group ID equals the GID of the file and the appropriate access but is set then access is allowed, and denied otherwise.
4. If the appropriate other access permission bit is set, then access is allowed.

- Ownership of new files and directories

UID of a new file is set to the EUID of the process.
GID is set to one of EGID of the process, or the GID of the directory where created.

Note: GID value depends on flavor of operating system.

- Access function

int access(const char *pathname, int mode);
Returns 0 if OK, -1 on error;

Test for file access based on the real UID/GID.
Mode is bitwise OR'd from R_OK, W_OK, X_OK, F_OK

Example (Program is SUID root, file is uucp:sys 640, we are richj:users)

```
int main(int argc, char **argv)
{
        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <pathname>\n", argv[0]);
                exit(1);
        }

        if (access(argv[1], R_OK) < 0)
                perror("access error");
        else
                printf("read access OK\n");

        if (open(argv[1], O_RDONLY) < 0)
                perror("open error");
        else
                printf("open for reading OK\n");
}
```

- Umask function

  mode_t umask(mode_t cmask);

  The umask function sets the file creation mask, and returns the previous value.
  Any bits that are on in the file mode creation mask are turned off in the file's mode.

  Example:

```
int main(void)
{
        umask(0);

        if (creat("foo", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH) < 0)
                perror("creat error");

        umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);

        if (creat("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
                S_IROTH | S_IWOTH) < 0)
                perror("creat error");

        exit(0);
}
```

```
$  umask
02
$  test_prog
$ ls -l foo bar
-rw-------      <…>   bar
-rw-rw-rw-     <…>   foo
```

- Chmod and fchmod functions

  int chmod(const char *pathname, mode_t mode);
  int fchmod(int filedes, mode_t mode);

  Returns 0 if OK, -1 on error

  To change permissions on a file, the EUID of the process must equal the owner of the file, or the process must have superuser priviledges.

  Note: Additional mode constants
  (S_ISUID, S_ISGID, S_ISVTX, S_IRWXU, S_IRWXG, S_IRWXO)

Example:

```
int main(void)
{
        struct stat        statbuf;

        if (stat("foo", &statbuf) < 0)
                perror("stat failure");

        /* Turn on SGID, and off group execute */

        if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
                perror("chmod error");


        /*  Set absolute to "rw-r--r--" */

        if (chmod ("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
                perror("chmod error");

        exit(0);
}
```

- Sticky bit

  Historical significance: If set on an executable, then on first execution, the program's text segment was copied to the swap area when the process terminated (i.e. program would stay memory resident). This allowed faster program loads on subsequent program executions.

  Current use: If sticky bit is set on a directory, a file can be removed or renamed only if the user has write permission for the directory and either owns the file, owns the directory, or is root. This is commonly done for public temporary directories.

- Chown, fchown, and lchown functions

  ```
  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);
  ```

  Note: Berkely based systems only allow chown by root, while System V systems allow for root or owner of file.

- File Size

  st_size (from stat structure) contains the size of the file in bytes.
  st_blksize is the preferred block size for I/O to the file.
  st_blocks is the actual number of blocks (often 512 bytes) allocated to the file.

  Holes can be created in files by "seeking" past the current end of file, and writing.

- File Truncation

  int truncate(const char *pathname, off_t length);
  int ftruncate(int filedes, off_t length);

  These functions truncate an existing file to length bytes.

  Note: This function is not extremely portable.

- Filesystems

  We can think of a disk being divided into one or more partitions, and each partition can contain a filesystem. Perhaps one of the most popular filesystems is UFS (Unified File System) which is a variation on the Berkely fast file system.

  See pictures 4.13 and 4.14 from textbook: The file system contains a boot block(s), super block, an inode list, and a group of directory and data blocks.

  I-nodes are fixed length entries. Typical sizes are 64 bytes or 128 bytes long.

  The I-node contains all the information about the file: its type, permission bits, size, pointers to the data blocks, etc. Most of the information from the stat structure is obtained from the I-node.

  I-nodes numbers are unique only for a given filesystem.

  Only two items are stored in the directory entry - the filename, and I-node number.

  When renaming a file without changing filesystems, the actual contents of the file need not be changed, only a new directory entry created…

  When two directory entries point to the same I-node, this is called a hard link. Every I-node has a link count that contains the number of directory entries pointing to that file. This is why "unlinking" a file does not necessarily "delete" the file. Unlink only decrements the link count. The file is deleted when the link count reaches zero.

- Link, unlink, remove, and rename functions

```
int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);
int remove(const char *pathname);
int rename(const char *oldname, const char *newname);
```

The link function creates a new hard link "newpath" for the file "oldpath".
The unlink function decrements the link count in the I-node, and deletes the file if 0.
The remove function equals unlink for files, and rmdir for directories.
The rename function is equivalent to "mv" in the shell.

Example (unlink and temp files):

```
int main(void)
{
        if (open("tempfile", O_RDWR) < 0)
                perror("open");

        if (unlink("tempfile") < 0)
                perror("unlink");

        printf("file unlinked\n");

        sleep(30);
        printf("done\n");

        exit(0);
}
```

Note: file is technically deleted when unlink completes, but space not returned until program terminates.

This property of unlink can be used by a program to assure that a temporary file it creates will not be left around in case the program crashes.

- Symbolic links

  A symbolic link is an indirect pointer to a file (i.e. a file that contains another filename).
  Does not suffer from same restrictions as hard links (i.e. can cross filesystems).
  Some fct follow symbolic links and others do not (i.e. some operate on the link itself).

  Note: It is possible to introduce loops into the filesystem via symlinks (see page 100).
  Note: See top of page 100 for list of functions that follow symlinks.
  Note: It is possible to have dangling symlinks (i.e. point to a file that does not exist).

- Symlink and readlink functions

  int symlink(const char *actualpath, const char *sympath);
  int readlink(const char *pathname, char *buf, int bufsize);

  The symlink function creates a symbolic link (i.e. ln -s).
  Since open follows symlinks, we need a function to open the link itself (i.e. readlink).

  Readlink opens the link, and places the filename that the link points to into "buf".

- File times

  There are 3 time values maintained for each file (stored in the inode):

Stat Field	Description	Example	Ls(1) option
st_atime	Last access time	read	-u
st_mtime	Last mod time	write	Default
st_ctime	Last change of inode	chmod, chown	-c

  Note: UNIX does not track access times to the inode (i.e. fct like and stat doesn't change atime).

- Utime function (Note: review UNIX time values from lecture 1)

  int utime(const char *pathname, const struct utimbuf *times);

```
struct utimbuf {
        time_t actime;      /* access time */
        time_t modtime;     /* modification time */
}
```

  It is possible to change the access and modification times with the **utime** function.

  If "times" is a null pointer, then both access and mod times are set to the current time.
  If "times" is a nonnull pointer, then the EUID must be the owner or root.
  Note: this function is used by touch(1), and utilities like tar(1) and cpio(1).

Example: (truncate files offered as cmd line args, but keep current times)

```
int main(int argc, char **argv)
{
        int             i;
        struct stat     statbuf;
        struct utimbuf  timebuf;

        for (i = 1; i < argc; i++)
        {
                if (stat(argv[i], &statbuf) < 0)
                {
                        perror("stat failed");
                        continue;
                }

                if (open(argv[i], O_RDWR | O_TRUNC) < 0)
                {
                        perror("open to truncate failed");
                        continue;
                }

                timebuf.actime = statbuf.st_atime;
                timebuf.modtime = statbuf.st_mtime;

                if (utime(argv[i], &timebuf) < 0)
                {
                        perror("utime error");
                        continue;
                }
        }

        exit(0);
}
```

- Mkdir and rmdir functions

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

These are system calls for the popular utilities (mkdir and rmdir).

- Reading directories

```
DIR             *opendir(const char *pathname);
struct dirent   *readdir(DIR *dp);
void            rewinddir(DIR *dp);
int             closedir(DIR *dp);

struct dirent {
        ino_t   d_ino;                  /* inode number */
        char    d_name[NAME_MAX + 1];   /* null-terminated filename */
}
```

These are functions to allow portable reading of directory files.

- Chdir, fchdir, and getcwd functions

```
int     chdir(const char *pathname);        /* same as "cd" */
int     fchdir(int filedes);                /* same as "cd" but to open file */
char    *getcwd(char *buf, size_t size);    /* same as "pwd" */
```

Example:

```
int main(void)
{
        char    *ptr = malloc(1024+1);
        int     size = 1024;

        if (chdir("/tmp/richj") < 0)
        {
                perror("chdir failed");
                exit(1);
        }

        if (getcwd(ptr, size) == NULL)
        {
                perror("getcwd failed");
                exit(1);
        }

        printf("cwd = %s\n", ptr);
        exit(0);
}
```

- Special device files

  All device drivers in UNIX can also be represented as files.
  The system stores most special device files in the "/dev" directory, but not required.

  Each device file has two numbers that identify it to the kernel (i.e. major and minor).

  The major number identifies the driver within the kernel that supports this device file.
  The minor number specifies "options" to that driver.

  Note: Use <sys/sysmacros.h> for macros to determine major and minor numbers.

  Example:

```
int main(int argc, char **argv)
{
        int             i;
        struct stat     buf;

        for (i = 1; i < argc; i++)
        {
                printf("%s: ", argv[i]);

                if (lstat(argv[i], &buf) < 0)
                {
                        perror("lstat failed");
                        exit(1);
                }

                printf("dev = %d / %d", major(buf.st_dev), minor(buf.st_dev));

                if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode))
                {
                        printf(" (%s) rdev = %d / %d",
                                (S_ISCHR(buf.st_mode)) ? "char" : "block",
                                major(buf.st_rdev),
                                minor(buf.st_rdev));
                }
                printf("\n");
        }

        exit(0);
}
```

- Sync and fsync functions

```
void    sync(void);
int     fsync(int filedes);
```

Most UNIX systems have a buffer cache in the kernel through which most I/O passes.

The sync function queues modified block buffers for writing and returns.
Sync is normally called every 30 seconds by the operating system.

The function fsync performs a similar action for a single file, but waits on I/O to finish.