15 Lecture: Filesytem wrapup, for sure, for sure

Outline:

Announcements
Lab03 stats
Example: A recursive traversal for codes?
A note on the lab
From last time: Filesystem functions
File Status Information: the stats
Some Examples
    Example: mv
    Example: A minimally functional ls
Directories
Example: microls.c
Identity Issues Revisited
File Protection Modes Revisited
Where a new file gets its values
Changing these values
File Permissions: `umask(2)`
Catchall calls: fcntl, ioctl
    fcntl() — modifying an open file descriptor
    ioctl() — the kitchen sink
Linking internal and external identity
Just a Matter of `time(2)`
    Time Display Routines <`time.h`>
    Time Manipulation Routines: utime()
Implementation details: the buffer cache

15.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn3	hencode/hdecode	Fri	Nov 3	23:59	
lab05	mypwd	Mon	Nov 6	23:59	
asgn4	mytar	Mon	Nov 27	23:59	
asgn5	mytalk	Fri	Dec 1	23:59	
lab07	forkit	Mon	Dec 4	23:59	
asgn6	shell	Fri	Dec 8	23:59	

  Use your own discretion with respect to timing/due dates.

- Use the test harness
- Calloc is not a panacea
- `sizeof(char)` is 1
- Remember not to open your input files for writing, e.g.
- DTA until 11/24

- From email:

  > We are hosting our largest pitch competition of the year, Startup Launch Weekend, from November 10-12. This Shark Tank-themed event will start downtown in the CIE Hothouse with guest speakers, catered food, and a business model canvas workshop. Throughout the weekend, students will form groups and develop a business idea, for the chance to win over $2100 in prizes!

15.2 Example: A recursive traversal for codes?

15.3 A note on the lab

Implementing:

```
char *getcwd(char *buf, size_t size);
         NULL and ERANGE on error
```

Inodes are unique to a filesystem, but inode and device are sufficient.

- current working directory and chdir()
- opendir, readdir, closedir
- Inodes on mounted filesystems.

15.4 From last time: Filesystem functions

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
int unlink(const char *pathname);

int symlink(const char *oldpath, const char *newpath);
int readlink(const char *path, char *buf, size_t bufsiz);
         returns character count or -1 (e.g., ENAMETOOLONG)

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

int chdir(const char *path);
int fchdir(int fd);

char *getcwd(char *buf, size_t size);
         NULL and ERANGE on error


DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

15.5 File Status Information: the stats

`stat()`, `lstat()`, and `fstat()` are the system calls for finding out about a file. (`stat(2)` on linux, `stat(3HEAD)` on solaris) They all populate a `struct stat`, shown in Figure 64.

```
NAME
       stat, fstat, lstat - get file status

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <unistd.h>

       int stat(const char *file_name, struct stat *buf);
       int fstat(int filedes, struct stat *buf);
       int lstat(const char *file_name, struct stat *buf);
```

EBADF	filedes is bad.
ENOENT	A component of the path file_name does not exist, or the path is an empty string.
ENOTDIR	A component of the path is not a directory.
ELOOP	Too many symbolic links encountered while traversing the path.
EFAULT	Bad address.
EACCES	Permission denied.
ENOMEM	Out of memory (i.e. kernel memory).
ENAMETOOLONG	File name too long.

```
struct stat
{
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    blksize_t     st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t      st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

Figure 64: struct stat returned by stat()

Structure Info.	Access Info.	Audit Info.
• What is it? (type) • Number of links • Where is it? Device(`st_dev`) and inode (`st_ino`) • How big is it? • If it's a device file, which device (`st_rdev`)	• owner • group • other	• atime • mtime • ctime

Consider how you can identify a **mounted** filesystem.

15.6 Some Examples

15.6.1 Example: mv

For an example of directory manipulation, see the example of renaming a file in Figure 65.

15.6.2 Example: A minimally functional ls

An example of listing the contents of a directory is found in Figure 66.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]){
  char *old, *new;
  struct stat sb;
  int status = 0;

  if ( argc != 3 ) {
    fprintf(stderr,"usage:  %s oldname newname\n",argv[0]);
    exit(-1);
  }

  old = argv[1];
  new = argv[2];

  if ( stat(old,&sb) ) {    /* make sure source exists */
    perror(old);
    status++;
  } else {
    if ( link(old, new) ) {     /* link the new name */
      perror(new);
      status ++;
    } else {
      if ( unlink(old) ) {      /* unlink the old one */
        perror(old);
        status++;
      }
    }
  }

  return status;
}
```

Figure 65: Renaming a file with link() and unlink()

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>


void listdir(char *path){
  /* list the contents of the given directory */
  DIR *d;
  struct dirent *ent;

  printf("Contents of %s:\n", path);

  if ( NULL == (d = opendir(path))) {
    perror(path);
  } else {
    while( (ent = readdir(d)) ) {
      printf("   %s\n", ent->d_name);
    }

    closedir(d);
  }
}

int main(int argc, char *argv[]) {
  int i;
  for(i=1; i < argc; i++)
    listdir(argv[i]);
  return 0;
}
```

Figure 66: A minimally functional implementation of ls.

15.7 Directories

Note, that even though there are many fields described in the `struct dirent`, only `d_ino` and `d_name` are required by POSIX.1 and its extension. (POSIX.1 only requires `d_name`.)

```
#include <dirent.h>

        struct dirent
        {
            long d_ino;                 /* inode number */
            off_t d_off;                /* offset to this dirent */
            unsigned short d_reclen;    /* length of this d_name */
            char d_name [NAME_MAX+1];   /* file name (null-terminated) */
        }


DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
off_t telldir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
int closedir(DIR *dir);
```

15.8 Example: microls.c

We developed something like Figure 67 in class today.

15.9 Identity Issues Revisited

Any process has several simultaneous ids:

real user id
real group id
effective user id
effective group id
(saved suid)
(saved sgid)

A file has two identities: user and group. The uid is the effective UID of the user who created it. The gid is either the effective group ID of the creating user, or the GID of the directory in which it is created.

15.10 File Protection Modes Revisited

The following POSIX macros are defined to check the file type:

```
#include<dirent.h>
#include<limits.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>

void ls_file(const char *path) {
  struct stat sb;
  if ( -1 == stat(path,&sb)) {
    perror(path);
  } else {
    printf("%s %ld\n",path,(long)sb.st_size);
  }
}

void ls_dir(const char *path) {
  DIR *d;
  struct dirent *ent;
  char name[PATH_MAX+1];

  if ( NULL == ( d = opendir(path))) {
    perror(path);
  } else {
    while(NULL != (ent=readdir(d))) {
      snprintf(name,PATH_MAX,"%s/%s",path,ent->d_name);
      ls_file(name);
    }
  }
}

int main(int argc, char *argv[]) {
  int i;
  struct stat sb;
  for(i=1;i<argc;i++) {
    if ( -1 == stat(argv[i],&sb)) {
      perror(argv[i]);
    } else if ( S_ISDIR(sb.st_mode)) {
      ls_dir(argv[i]);
    } else {
      ls_file(argv[i]);
    }
  }
  return 0;
}
```

Figure 67: A less-minimally functional implementation of ls.

Macro	Meaning
S_ISLNK(m)	is it a symbolic link?
S_ISREG(m)	regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	fifo?
S_ISSOCK(m)	socket?

RWX permissions and their meanings

SUID—change user id on execution

SGID—change group id on execution

Sticky (saved-text)—various

See Figure 68 for the meanings of the particular bits. In general, use the macros.

Mode Bits		
Macro	**Value**	**Meaning**
S_IFMT	0170000	bitmask for the file type bitfields
S_IFSOCK	0140000	socket
S_IFLNK	0120000	symbolic link
S_IFREG	0100000	regular file
S_IFBLK	0060000	block device
S_IFDIR	0040000	directory
S_IFCHR	0020000	character device
S_IFIFO	0010000	fifo
S_ISUID	0004000	set UID bit
S_ISGID	0002000	set GID bit (see below)
S_ISVTX	0001000	sticky bit (see below)
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others (not in group)
S_IROTH	00004	Others have read permission
S_IWOTH	00002	Others have write permission
S_IXOTH	00001	Others have execute permission

Figure 68: Macros for bits in file modes

15.11 Where a new file gets its values

uid	effective uid of creator
gid	egid of creator or gid of directory (BSD/solaris/linux)
	Linux is configurable(mount(8)): grpid or bsdgroups / nogrpid or sysvgroups
permissions	`mode && ~umask`

15.12 Changing these values

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


int     chmod(const char *path, mode_t mode);
int     fchmod(int fildes, mode_t mode);
mode_t  umask(mode_t mask);
int     chown(const char *path, uid_t owner, gid_t group);
int     fchown(int fd, uid_t owner, gid_t group);
int     lchown(const char *path, uid_t owner, gid_t group);
```

Restrictions on chmod() and chown():

- Only the owner can chown() or chmod() a file.
- **chmod()** clears sticky bit if not root. Sgid turned off by chmod() if owner is not a member of the group.(WHY?)
- suid and sgid turned off on write by a process is not owner (or not root).
- **chown()** Can only chown to self if not root. Can only chgrp to group where one is a member.

15.13 File Permissions: `umask(2)`

The umask us used to remove particular permissions during the process of file-creation with `open(2)` or `creat(2)`. When one of those functions is called, the given permissions are ANDed with the bitwise inverse of the umask to get the final permissions.

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

mode_t umask(mode_t mask);
```

15.14 Catchall calls: fcntl, ioctl

15.14.1 fcntl() — modifying an open file descriptor

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

       int fcntl(int fd, int cmd);
       int fcntl(int fd, int cmd, long arg);
       int fcntl(int fd, int cmd, struct flock * lock);
```

Of interest now:

- **File descriptor flags:** `F_GETFD`, `F_SETFD`

  There is only one flag of interest here: `FD_CLOEXEC`, close on exec.

- **File status flags:** `F_GETFL`, `F_SETFL`

  Allows you to examine or change the way file IO is handled on an open file descriptor: `O_RDONLY`, `O_WRONLY`, `O_RDWR` cannot be changed, but `O_APPEND` (and `O_NONBLOCK`, `O_SYNC` `O_ASYNC` for that matter) can be.

15.14.2 ioctl() — the kitchen sink

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

int ioctl(int d, int request, ...);
```

ioctl() is the traditional catchall for device manipulation. Its semantics are defined by each particular device driver.

15.15 Linking internal and external identity

```
int chown(const char *path, uid_t owner, gid_t group);

#include <pwd.h>
#include <sys/types.h>
struct passwd *getpwnam(const char * name);
struct passwd *getpwuid(uid_t uid);

#include <grp.h>
#include <sys/types.h>
struct group *getgrnam(const char *name);
struct group *getgrgid(gid_t gid);
```

```
struct passwd {
        char    *pw_name;       /* user name */
        char    *pw_passwd;     /* user password */
        uid_t   pw_uid;         /* user id */
        gid_t   pw_gid;         /* group id */
        char    *pw_gecos;      /* real name */
        char    *pw_dir;        /* home directory */
        char    *pw_shell;      /* shell program */
};
```

```
struct group {
        char    *gr_name;       /* group name */
        char    *gr_passwd;     /* group password */
        gid_t   gr_gid;         /* group id */
        char    **gr_mem;       /* group members */
};
```

15.16 Just a Matter of `time(2)`

Unix represents time as the number of seconds since Midnight, January 1st, 1970, UTC (coordinated universal time). Anything else is just a prettification.

`SYNOPSIS`

```
#include <time.h>
```

```
       time_t time(time_t *t);

DESCRIPTION
       time  returns the time since the Epoch (00:00:00 UTC, Jan-
       uary 1, 1970), measured in seconds.

       If t is non-NULL, the return value is also stored  in  the
       memory pointed to by t.
```

The three times associated with any file are:

stat(2)	ls(1) -l	Meaning
atime	-u	Time of last access
mtime	<none>	Time of last modification
ctime	-c	Time of last change (incl. of the inode)

15.16.1 Time Display Routines <time.h>

```
NAME
       asctime,  ctime,  gmtime,  localtime,  mktime  - transform
       binary date and time to ASCII

SYNOPSIS
       #include <time.h>

       char *asctime(const struct tm *timeptr);

       char *ctime(const time_t *timep);

       struct tm *gmtime(const time_t *timep);

       struct tm *localtime(const time_t *timep);

       time_t mktime(struct tm *timeptr);

       extern char *tzname[2];
       long int timezone;
       extern int daylight;

       size_t strftime(char *s, size_t max, const char *format,
                       const struct tm *tm);



              struct tm
              {
                      int     tm_sec;         /* seconds */
                      int     tm_min;         /* minutes */
                      int     tm_hour;        /* hours */
```

```
                int     tm_mday;        /* day of the month */
                int     tm_mon;         /* month */
                int     tm_year;        /* year */
                int     tm_wday;        /* day of the week */
                int     tm_yday;        /* day in the year */
                int     tm_isdst;       /* daylight saving time */
        };
```

15.16.2 Time Manipulation Routines: utime()

```
UTIME(2)            Linux Programmer's Manual            UTIME(2)

NAME
       utime, utimes - change access and/or modification times of
       an inode

SYNOPSIS
       #include <sys/types.h>
       #include <utime.h>

       int utime(const char *filename, struct utimbuf *buf);

DESCRIPTION
       utime changes the access and  modification  times  of  the
       inode  specified  by  filename  to  the actime and modtime
       fields of buf respectively.  If  buf  is  NULL,  then  the
       access  and  modification times of the file are set to the
       current time.  The utimbuf structure is:

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

15.17 Implementation details: the buffer cache

```
NAME
       sync - commit buffer cache to disk.

SYNOPSIS
       #include <unistd.h>

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