ECS 36A, May 19, 2025

System Calls

- Direct interface between the applications and the operating system
- They vary among operating systems
  - We will deal with Linux system calls
- We'll look primarily at the file system calls

Opening a File: Basic Ideas

- Files represented by an integer
  - 0 refers to the standard output
  - 1 refers to the standard output
  - 2 refers to the standard error
- To get a file descriptor from a file stream:

  `int fileno(FILE *`*`fp`*`)`

  returns the file descriptor associated with file pointer *fp*
- You can now mix system and stdio calls *provided you use the file descriptor in the same way you use the file pointer*
  - For example, if the file *fp* points to is open for reading, using the file descriptor to write to it will give you an error

Opening a File: Basic Ideas

- To get a file pointer from a file descriptor:

  `FILE *fdopen(int` *`fd`*`, char *`*`mode`*`)`

  creates a file pointer (and corresponding structure) to file descriptor *fd*, which was opened as mode indicates
- You have to set *mode* the same way as you opened it
- The system maintains a rw-pointer at the spot (the *file offset*) where the next read or write will take place
  - Unless there is an *fseek* or *fsetpos*, which moves the rw-pointer

Opening a File: Details

```
int open(const char *name, int flags)
```

- Opens the file name in the way flags indicate:
  - **O_RDONLY**: open file for reading
  - **O_WRONLY**: open file for writing
  - **O_RDWR**: open file for reading and writing
- Other flags augment these
  - **O_APPEND**: with **O_WRONLY** or **O_RDWR**, append rather than overwrite
  - **O_CREAT**: create the file if it does not exist
  - **O_EXCL**: with **O_CREAT**, fail if the file exists

Detour: File Permissions

- File protection, or *mode*, is 12 bits long; for us, the first 3 bits are irrelevant
- The other 9 bits are arranged in groups of 3:

r w x r w x r w x

- First 3 refer to *owner* (also called *user*)
- Second 3 refer to *group*
- Last 3 refer to everyone else (sometimes called *other* or *world*)
- Each r (read), w (write), x (execute) is a bit; 1 means allowed, 0 means not allowed

Detour: *umask*

- *Umask* is a shell variable designed to *mask* file creation permissions
- Each bit of *umask* turns off the corresponding bit in the permissions when a file is created
  - It's a safety mechanism so the file creator doesn't accidentally give others access they should not have
- Example: file is created with permission 666 (anyone can read or write it)
  - Not a good idea!
- Set *umask* to 022 (group and other write bits set here)
- Result: the file is created with permission 644 (anyone can read it, but only the owner can write to it)

Creating a File

- When creating a file, a third argument specifies permissions:

```
int open(const char *name, int flags, mode_t mode)
```

- The file permissions are set to

$$mode\&\sim umask$$

- Example: if *umask* is 077, and mode is 0644 (owner can write, everyone can read), the file is created with protection mode

$$0644\&\sim 077 = 0644\&0700 = 0600$$

  so only the owner can read or write the file

Example: *showopen.c*

Here, *mask* is the new umask, *mode* the desired protection mode.

- Both entered on command line as octal digits

Here's a sample run:

```
> showopen xyzzy 0666 022
> ls -l xyzzy
-rw-r--r-- 1 bishop bishop 0 May 22 21:58 xyzzy
```

First argument is file name
Second argument is protection mode
Third argument is umask

The main code:

```
umask(mask);
```

Set the umask

Set the protection mode; if omitted, defaults to 666 (anyone can read, write the file) but then the umask is applied

```
if ((fd = open(argv[1], O_WRONLY|O_CREAT, mode)) < 0){
        error(argv[1], ":", " ");
        error(strerror(errno), "\n", "");
}
```

Reading

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

- Read *count* bytes from file descriptor *fd* and save them in the area *buf* points to
  - You have to allocate *buf* or create an array or variable to give the address of
  - On success, returns the number of bytes read; this is never more than *count* but may be less
  - If it returns 0, you've reached the end of file
  - If it returns –1, an error occurred, and *errno* is set to indicate the error

Writing

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

- Writes *count* bytes from the address *buf* contains to file descriptor *fd*
  - *buf* is the address of what you want written
  - *count* is the number of bytes to write; it does *not* stop at the NUL ('\0') byte
  - On success, returns the number of bytes written; this is never more than *count* but may be less
  - If it returns 0, nothing was written
  - If it returns –1, an error occurred, and *errno* is set to indicate the error

Example: main loop of *catsys.c*

```
for(i = 1; i < argc; i++){
        if ((fd = open(argv[i], O_RDONLY)) < 0){
                write(2, "Could not read ", 15);
                write(2, argv[i], (size_t) strlen(argv[i]));
                write(2, "\n", 1);
        }
        else{
                rv -= cat(fd);
                (void) close(fd);
        }
}
```

Main loop: process arguments one by one

open file for reading; on error, *open* returns 1; on success, it returns the file descriptor *fd*

These write to the standard error (file descriptor 2)

open worked; *cat* displays file (*fd* is the file descriptor); *cat* returns 0 on success, 1 on failure

close file disassociates *fd* from file; we ignore the return value Not really necessary but it's good computing hygiene

Example: First Part of *catsys.c*

```
int cat(int fd)
{
        char x;         /* space for char that is read */
        int rv;        /* number of chars read (or -1 on error) */

        while((rv = read(fd, &x, 1)) == 1){
                if (write(1, &x, 1) != 1){
                        (void) write(2, "Could not write to screen\n", 26);
                        return(-1);
                }
        }
```

Function definition; notice it uses a file *descriptor*, not pointer

Main loop: read in one char

Write it out

This writes to the standard error (file descriptor 2)

Return -1 to indicate error

Example: A Better First Part of *catsys.c*

```
int cat(int fd)
{
        char buffer[BUFSIZ];        /* space for what is read */
        int rv;        /* number of chars read (or -1 on error) */

        while((rv = read(fd, buffer, BUFSIZ)) > 0){
                if (write(1, buffer, rv) != rv){
                        (void) write(2, "Could not write to screen\n", 26);
                        return(-1);
                }
        }
```

Function definition; notice it uses a file *descriptor*, not pointer

Main loop: read in a block of characters rather than just 1— hence "better"

Write it out; check you wrote it all out – may read less than BUFSIZ chars, hence test against *rv*

This writes to the standard error (file descriptor 2)

Return -1 to indicate error

Example: Second Part of *catsys.c*

```
if (rv != 0){
        (void) write(2, "Error reading\n", 13);
        return(-1);
}
return(0);
```

if you get here and *rv* != 0, then *read* returned -1, and an error occurred

This writes to the standard error (file descriptor 2)

Return -1 to indicate error

Return 0- to indicate there were no errors (success!)

Seeking

```
off_t lseek(int fd, off_t offset, int whence)
```

- Move the rw-pointer associated with the file descriptor *fd* to offset according to whence
  - *whence* is **SEEK_SET** (beginning), **SEEK_CUR** (current position), **SEEK_END** (end of file)
  - It returns new rw-pointer offset in bytes from beginning of file
- Error handling
  - If it returns –1, an error may have occurred, and if so *errno* is set to indicate the error
  - Note: off_t is unsigned long int, so that could be a valid value of a very big file

Detecting Error in *lseek*

- If you are moving the rw-pointer to a given position *x*, this works:

```
if (lseek(fd, x, SEEK_SET) != x) . . .
```

- Otherwise, do this:

```
errno = 0;          /* clear any existing error code */
if (lseek(fd, offset, whence) == -1 && errno != 0){
   /* . . . Handle error . . . */
```

Redoing *stats2.c*

- Earlier program used standard I/O functions *fopen*, *fpos*, *fread, etc.*
- *stats2s.c* uses system calls instead
  - The only standard I/O function used is *sprintf* (because it lets us convert integers and floating-point numbers into strings easily)

Example: Output and Error Functions

```
/* write to standard output */
void outdump(char *s)
{
        (void) write(1, s, strlen(s) * sizeof(char));
}
/* write to standard error */
void errdump(char *s)
{
        (void) write(2, s, strlen(s) * sizeof(char));
}
```

s is the output string

File descriptor 1 means the standard output

s is the error message

File descriptor 2 means the standard error

Example: Print System Error Message

```
/* write a system error message to standard error */
/* handled like perror(3)                         */
void perrdump(char *s)
{
        int oops = errno;       /* remember the current error number */

        errdump(s);
        errdump(": ");
        errdump(strerror(oops));
        errdump("\n");
}
```

s is the error string

Save the error number in case *errdump* resets it

Print the user part of the message

Translate the saved error number into a printable string

Example: main loop of *stats2s.c*

```
for (i = optind; argv[i] != NULL; i++){
        /* open the file */
        if ((fd = open(argv[i], O_RDONLY)) < 0){
                perrdump(argv[i]);
                rv++;
                continue;
        }
        /* do the test */
        bintest(argv[i], fd);
        /* done with this file */
        (void) close(fd);
}
```

Main loop: process arguments one by one

open file for reading; on error, *open* returns 1; on success, it returns the file descriptor *fd*

Write out the system error message, add 1 to buber of failures, and continue

open worked; *bintest* develops statistics for file (*fd* is the file descriptor)

close file disassociates *fd* from file; we ignore the return value Not really necessary but it's good computing hygiene

Example: *bintest* Main Loop

```
/* go to the position, and check for errors */
errno = 0;
if (lseek(fd, skip, SEEK_CUR) == -1 && errno != 0){
        perrdump("lseek");
        break;
}
/* if done, break out of the loop */
if ((x = read(fd, &ch, sizeof(char))) == 0)
        break;
else if (x != 1){
        perrdump("read");
        break;
}
```

Here, *fd* is the file descriptor of the file

Need to check *errno* if *lseek* returns -1, as -1 is a value *lseek* might legitimately return

if *read* returns 0, at EOF and so drop out of loop

if *read* returns 1, it read a char; otherwise it's an error