Week 3 - Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Process lifecycle
- Started files

Questions?

Assignment 2

Project 1

Files

UNIX file abstraction

- The UNIX file abstraction uses two key ideas:
  - A file is a sequence of bytes
  - Everything is a file
- This abstraction is different from the traditional idea of files in a few ways:
  - Moving backwards and forwards within a file isn't always possible
  - Files don't always have names or live in a particular place
  - Files don't always have a set structure
- Even so, creating, deleting, opening, closing, reading, and writing can be treated the same

Opening files

- To open a file for reading or writing, use the **`open()`** function
- The **`open()`** function takes the file name, an **`int`** for mode, and an (optional) **`mode_t`** for permissions
- The name refers to an entity somewhere in the directory structure that might or might not be a normal file
- It returns a file descriptor as an **`int`**

```
int fd = open ("input.dat", O_RDONLY);
```

Constants

- A number of constants specify whether the opening is for reading or writing
- The optional permissions value has other constants to set the permissions of the file when creating a new one
- Both sets of constants can be bitwise ORed together to make complicated values

Access	Meaning
`O_RDONLY`	Open for reading only
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing
`O_NONBLOCK`	Do not block on opening while waiting for data
`O_CREAT`	Create the file if it does not exist, requires `mode_t` argument
`O_TRUNC`	Truncate to size o
`O_EXCL`	Error if `O_CREAT` and the file exists

Name	Description
`S_IRUSR`	Read (user)
`S_IWUSR`	Write (user)
`S_IXUSR`	Execute (user)
`S_IRGRP`	Read (group)
`S_IWGRP`	Write (group)
`S_IXGRP`	Execute (group)
`S_IROTH`	Read (other)
`S_IWOTH`	Write (other)
`S_IXOTH`	Execute (other)

Example with other constants

- The following example shows how to open a file
  - For writing
  - By creating it
  - Truncating its size to 0 if there's already something in the file
  - Making it readable and writable to the user and readable to others

```
int fd = open ("output.dat", O_CREAT | O_TRUNC |
O_WRONLY, S_IRUSR | S_IWUSR | S_IROTH);
```

- It's also common to use numbers in octal for permissions, where the 64's place is permission for the user, the 8's place is permission for the group, and the 1's place is permission for others
  - **S_IRUSR | S_IWUSR | S_IROTH = 110 000 100 = 0604**

Reading from files

- Opening the file is actually the hardest part
- Reading is straightforward with the **`read()`** function
- Its arguments are
  - The file descriptor
  - A pointer to the memory to read into
  - The number of bytes to read
- Its return value is the number of bytes successfully read

```
int fd = open ("input.dat", O_RDONLY);
int buffer[100];
// Fill with something
read (fd, buffer, sizeof(int)*100);
```

Closing files

- To close a file descriptor, call the **`close()`** function
- Close files when you're done with them

```
int fd = open ("output.dat", O_WRONLY | O_CREAT | O_TRUNC,
0644);
// Write some stuff
close (fd);
```

Special files

- Linux provides some "special" files
  - **`/dev/full`**
    - A file that's says the device is full if you try to write to it, gives unlimited zeroes if you try to read from it
  - **`/dev/null`**
    - A file you can write to forever but simply discards the data (while saying that the write succeeded)
  - **`/dev/random`**
    - A file you can read a stream of random bytes from
  - **`/dev/zero`**
    - A file you can read an unlimited stream of zero bytes from
- They're not actually files, but you can treat them as if they are
- They can be useful for testing and sometimes even for the operation of program

Reading random data

- Let's open **`/dev/random`** and read data to fill 10 random **`int`** values

Polling files

- For some devices, it can be useful to see if the "file" that represents the device is ready to be read from
- If not, the program can do other things and come back
- The **`poll()`** function lets us check to see if a file is ready, instead of blocking
  - First parameter is an array of **`pollfd`** structs containing the file descriptor and the kind of access you want
  - Second parameter is the length of the array
  - Third parameter is how long to wait for information, in milliseconds

```
int successes = poll (array, length, 100);
```

Polling example

- The following example shows what code is like to check to see if a file is ready to be read (using the **POLLIN** constant)

```
// Set up a single pollfd for the file descriptor fd
struct pollfd fds[1];
fds[0].fd = fd;
fds[0].events = POLLIN; // Looking for input data

if (poll (fds, 1, 100) == 0) // Wait for 100 ms
  printf ("Poll failed: %d\n", fds[0].revents);
else
  {
    printf ("Poll successful!\n");
    // Read from the file
  }
```

Writing to files

- Writing to a file is almost the same as reading
- Arguments to the **`write()`** function are
  - The file descriptor
  - A pointer to the memory to write from
  - The number of bytes to write
- Its return value is the number of bytes successfully written

```
int fd = open ("output.dat", O_WRONLY | O_CREAT | O_TRUNC, 0644);
int buffer[100];
for (int i = 0; i < 100; ++i)
     buffer[i] = i + 1;
write (fd, buffer, sizeof(int)*100);
```

Seeking to locations

- It's possible to move the current location within the file using the **`lseek()`** function
- Its arguments are
  - The file descriptor
  - The offset (positive or negative)
  - Location to seek from:
    - **`SEEK_SET`** (beginning of file)
    - **`SEEK_CUR`** (current location)
    - **`SEEK_END`** (end of file)
- Seeking is more common when reading, but you can seek while writing too

```
int fd = open ("input.dat", O_RDONLY);
lseek (fd, 100, SEEK_SET);
```

File metadata

- The data in the file is the sequence of bytes it contains
- The **metadata** of a file gives information about the file itself
  - Obscure OS stuff like inode number and hard links to the file
  - User ID of the owner
  - Group ID of the owner
  - Device type
  - File size
- This information can be stored in a **`struct stat`** and retrieved with:
  - **`fstat()`** Gets information from a file descriptor
  - **`stat()`** Gets information from a path

Interpreting metadata

- The following shows some fields in **`struct stat`**
- The **`st_mode`** field is a bitwise OR of permissions and other information from the table on the right

```
struct stat {
  dev_t     st_dev;     // device of inode
  ino_t     st_ino;     // inode number
  mode_t    st_mode;    // protection mode
  nlink_t   st_nlink;   // hard links to file
  uid_t     st_uid;     // user ID of owner
  gid_t     st_gid;     // group ID of owner
  dev_t     st_rdev;    // device type
  off_t     st_size;    // file size in bytes
  // Other fields depending on OS ...
};
```

Name	Description
`S_IFIFO`	Named pipe (IPC)
`S_IFCHR`	Character device (terminal)
`S_IFDIR`	Directory file type
`S_IFBLK`	Block device (disk drive)
`S_IFREG`	Regular file type
`S_IFLNK`	Symbolic link
`S_IFSOCK`	Socket (IPC, networks)

Example getting file metadata

- The following code finds out how big a file (stored with file descriptor **`fd`**) is in bytes:

```
struct stat metadata;
fstat (fd, &metadata);
printf ("File size: %lld bytes\n",
        (long long)metadata.st_size);
```

Practice

- Let's write a program that:
  - Prompts the user for a file name
  - Uses **`stat()`** to get metadata about the file
  - Print out the size of the file in bytes
  - Use the **`getpwuid()`** function get login information about the owner of the file
  - Print out the user's login name (the **`pw_name`** member of the **`passwd struct`**)

Ticket Out the Door

Upcoming

Next time…

- Finish files
- Events and signals

Reminders

- Finish Assignment 2
  - Due Friday by midnight
- Start working on Project 1
- Read section 2.7