Week 11 - Monday

COMP 2400

Last time

- What did we talk about last time?
- Exam 2 Post Mortem
- Users and groups

Questions?

Project 5

Quotes

Weeks of programming can save you hours of planning.

Anonymous

Binary Files

What is a binary file?

- Technically, **all** files are binary files
    - They all carry data stored in binary
- But some of those binary files are called **text files** because they are filled with human readable text
- When most people talk about binary files, they mean files with data that is only computer readable

Why use binary files?

- Wouldn't it be easier to use all human readable files?
- Binary files can be more efficient
  - In binary, all **`int`** values are the same size, usually 4 bytes
- You can also load a chunk of memory (like a WAV header) into memory with one function call

Integer	Bytes in text representation
`0`	1
`92`	2
`789`	3
`4551`	4
`10890999`	8
`204471262`	9
`-2000000000`	11

Changes to `fopen()`

- To specify that a file should be opened in binary mode, append a **`b`** to the mode string

```
FILE* file = fopen("output.dat", "wb");
```

```
FILE* file = fopen("input.dat", "rb");
```

- On some systems, the **`b`** has no effect
- On others, it changes how some characters are interpreted

`fread()`

- The **`fread()`** function allows you to read binary data from a file and drop it directly into memory
- It takes
  - A pointer to the memory you want to fill
  - The size of each element
  - The number of elements
  - The file pointer

```c
double data[100];
FILE* file = fopen("input.dat", "rb");
fread(data, sizeof(double), 100, file);
fclose(file);
```

fwrite()

- The **fwrite()** function allows for binary writing
- It can drop an arbitrarily large chunk of data into memory at once
- It takes
  - A pointer to the memory you want to write
  - The size of each element
  - The number of elements
  - The file pointer

```
short values[50];
FILE* file = NULL;
//fill values with data
file = fopen("output.dat", "wb");
fwrite(values, sizeof(short), 50, file);
fclose(file);
```

Seeking

- Binary files can be treated almost like a big chunk of memory
- It is useful to move the location of reading or writing inside the file
  - Some file formats have header information that says where in the file you need to jump to for data
- **`fseek()`** lets you do this
- Seeking in text files is possible but much less common

fseek()

- The **fseek()** function takes
  - The file pointer
  - The offset to move the stream pointer (positive or negative)
  - The location the offset is relative to
- Legal locations are

SEEK_SET	From the beginning of the file
SEEK_CUR	From the current location
SEEK_END	From the end of the file (not always supported)

```
FILE* file = fopen("input.dat", "rb");
int offset;
fread(&offset,sizeof(int),1,file); //get offset
fseek(file, offset, SEEK_SET);
```

Example 1

- Write a program that prompts the user for an integer ***n*** and a file name
- Open the file for writing in binary
- Write the value ***n*** in binary
- Then, write the ***n*** random numbers in binary
- Close the file

Example 2

- Write a program that reads the file generated in the previous example and finds the average of the numbers
- Open the file for reading
- Read the value ***n*** in binary so you know how many numbers to read
- Read the ***n*** random numbers in binary
- Compute the average and print it out
- Close the file

Low Level File I/O

Low level I/O

- You just learned how to read and write files
  - Why are we going to do it again?
- There's a set of Unix/Linux system commands that do the same thing
- Most of the higher level calls (**`fopen()`**, **`fprintf()`**, **`fgetc()`**, and even trusty **`printf()`**) are built on top of these low level I/O commands
- These give you direct access to the file system (including pipes)
- They can be more efficient
- You'll use the low-level file style for networking
- All low level I/O is binary

Includes

- To use low level I/O functions, include headers as follows:

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

- You won't need all of these for every program, but you might as well throw them all in

File descriptors

- High level file I/O uses a **`FILE*`** variable for referring to a file
- Low level I/O uses an **`int`** value called a **file descriptor**
- These are small, nonnegative integers
- Each process has its own set of file descriptors
- Even the standard I/O streams have descriptors

Stream	Descriptor	Defined Constant
`stdin`	0	**`STDIN_FILENO`**
`stdout`	1	**`STDOUT_FILENO`**
`stderr`	2	**`STDERR_FILENO`**

open()

- To open a file for reading or writing, use the **open()** function
  - There used to be a **creat()** function that was used to create new files, but it's now obsolete
- The **open()** function takes the file name, an **int** for mode, and an (optional) **int** for permissions
- It returns a file descriptor

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

Modes

- The main modes are

`O_RDONLY`	Open the file for reading only
`O_WRONLY`	Open the file for writing only
`O_RDWR`	Open the file for both

- There are many other optional flags that can be combined with the main modes
- A few are

`O_CREAT`	Create file if it doesn't already exist
`O_DIRECTORY`	Fail if pathname is not a directory
`O_TRUNC`	Truncate existing file to zero length
`O_APPEND`	Writes are always to the end of the file

- These flags can be combined with the main modes (and each other) using bitwise OR

```c
int fd = open("output.dat", O_WRONLY | O_CREAT | O_APPEND );
```

Permissions

- Because this is Linux, we can also specify the permissions for a file we create
- The last value passed to **`open()`** can be any of the following permission flags bitwise ORed together

`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

```
int fd = open("output.dat", O_WRONLY | O_CREAT | O_APPEND,
S_IRUSR | S_IRGRP );
```

An alternative for permissions

- The constants on the previous slides are a perfectly good way to specify permissions
- They're (sort of) readable
- Another way is by using octal
- First, use a single bit for the permissions for read, write, and execute for each of the roles user, group, and others

1	1	1	1	0	1	1	0	0
Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
User			Group			Others		

- Then, convert the binary into octal
- Each group of three permissions is a single octal digit:
  - 111 = 7, 101 = 5, 100 = 4, yielding 0754 in octal
  - Remember that octal literals in C (and Java) start with zero

Permission practice

- Convert the following permissions into an octal number:
  - User: Read and write
  - Group: Read
  - Others: Execute

- Convert the octal value 0742 into permissions

read()

- 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];
read( fd, buffer, sizeof(int)*100 );
```

write()

- 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, 0777);
int buffer[100];
int i = 0;
for( i = 0; i < 100; i++ )
     buffer[i] = i + 1;
write( fd, buffer, sizeof(int)*100 );
```

close()

- To close a file descriptor, call the **close()** function
- Like always, it's a good idea to 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 );
```

lseek()

- It's possible to seek with low level I/O using the **lseek()** function
- Its arguments are
  - The file descriptor
  - The offset
  - Location to seek from: **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**

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

Upcoming

Next time…

- Networking
- Start sockets

Reminders

- Work on Project 5
- Keep reading LPI chapters 13, 14, and 15