Week 12 - Wednesday

CS222

Last time

- What did we talk about last time?
- Bit fields
- Unions

Questions?

Project 5

Exam 2 Post Mortem

Quotes

Weeks of programming can save you hours of planning.

Anonymous

Stack initialization

- Initializing the stack isn't hard
  - We give it an initial capacity (perhaps 5)
  - We allocate enough space to hold that capacity
  - We set the size to 0

```
Stack stack;
stack.capacity = 5;
stack.values = (double*)
    malloc(sizeof(double)*stack.capacity );
stack.size = 0;
```

Push, pop, and top

- We can write simple methods that will do the operations of the stack ADT

```
void push(Stack* stack, double value);
```

```
double pop(Stack* stack);
```

```
double top(Stack* stack);
```

Postfix notation

- You might recall postfix notation from CS221
  - It is an unambiguous way of writing mathematical expressions
- Whenever you see an operand, put it on the stack
- Whenever you see an operator, pop the last two things off the stack, perform the operation, then put the result back on the stack
- The last thing should be the result
- Example: **`5 6 + 3 –`** gives $(5 + 6) – 3 = 8$

Evaluate postfix

- Finally, we have enough machinery to evaluate an array of postfix terms
- Write the following function that does the evaluation:

```
double evaluate(Term terms[], int size);
```

- We'll have to see if each term is an operator or an operand and interact appropriate with the stack

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 is 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 are often 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

```
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 );
```

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 );
//fill with something
```

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);
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);
//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 );
```

Example

- Use low level I/O to write a hex dump program
- Print out the bytes in a program, 16 at a time, in hex, along with the current offset in the file, also in hex
- Sample output:

```
0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
```

File descriptors revisited

- A file descriptor is not necessarily unique
  - Not even in the same process
- It's possible to duplicate file descriptors
  - Thus, the output to one file descriptor also goes to the other
  - Input is similar

Duplicating descriptors on the command line

- **`stderr`** usually prints to the screen, even if **`stdout`** is being redirected to a file

```
./program > output.txt
```

- What if you want **`stderr`** to get printed to that file as well?

```
./program > output.txt 2>&1
```

- You can also redirect only **`stderr`** to a file

```
./program 2> errors.log
```

dup() and dup2()

- If you want a new file descriptor number that refers to an open file descriptor, you can use the **`dup()`** function

```
int fd = dup(1); //makes a copy of stdout
```

- It's often more useful to change an existing file descriptor to refer to another stream, which you can do with **`dup2()`**

```
dup2(1, 2);
//makes 2 (stderr) a copy of 1 (stdout)
```

- Now all writes to **`stderr`** will go to **`stdout`**

I/O buffering in files

- Reading from and writing to files on a hard drive is expensive
- These operations are buffered so that one big read or write happens instead of lots of little ones
  - If another program is reading from a file you've written to, it reads from the buffer, not the old file
- Even so, it is more efficient for your code to write larger amounts of data in one pass
  - Each system call has overhead

Buffering in stdio

- To avoid having too many system calls, **`stdio`** uses this second kind of buffering
  - This is an advantage of **`stdio`** functions rather than using low-level **`read()`** and **`write()`** directly
- The default buffer size is 8192 bytes
- The **`setvbuf()`**, **`setbuf()`**, and **`setbuffer()`** functions let you specify your own buffer

Flushing a buffer

- Stdio output buffers are generally flushed (sent to the system) when they hit a newline (**`'\n'`**) or get full
  - When debugging code that can crash, make sure you put a newline in your **`printf()`**, otherwise you might not see the output before the crash
- There is an **`fflush()`** function that can flush **`stdio`** buffers

```
fflush(stdout); //flushes stdout
//could be any FILE*
fflush(NULL); //flushes all buffers
```

Quiz

Upcoming

Next time…

- Files and atomicity
- File systems
- Lab 12

Reminders

- Finish Project 5
  - Due Friday by midnight
- Read LPI chapters 13, 14, and 15