Week 5 - Monday

COMP 3400

Last time

- What did we talk about last time?
- FIFOs
- Memory-mapped files

Questions?

Assignment 3

Project 1

Programming practice

- Memory map a bitmap file read in from the user
- Then, write out the contents of the header, which should match the following **`struct`**:

```
struct BitmapHeader {
    unsigned char type[2];                // always contains 'B' and 'M'
    unsigned int size;                    // total size of file
    unsigned int reserved;                // always 0
    unsigned int offset;                  // start of data from front of file
    unsigned int header;                  // size of header, always 40
    unsigned int width;                   // width of image in pixels
    unsigned int height;                  // height of image in pixels
    unsigned short planes;                // planes in image, always 1
    unsigned short bits;                  // color bit depths, always 24
    unsigned int compression;             // always 0
    unsigned int dataSize;                // size of color data in bytes
    unsigned int horizontalResolution;    // unreliable, use 72 when writing
    unsigned int verticalResolution;      // unreliable, use 72 when writing
    unsigned int colors;                  // colors in palette, use 0 when writing
    unsigned int importantColors;         // important colors, use 0 when writing
};
```

Problem with the example

- When we do this, we'll get unexpected values for **`size`**, **`width`**, and **`height`**
- The problem is one that's important when dealing with memory directly
- Struct members are typically packed to fall on certain boundaries
  - In this case, the **`unsigned int`** values will fall on 4-byte boundaries
  - That means that the struct we defined expects two unused bytes after **`type`** but before **`size`**
- To fix this problem, we surround the struct declaration with the following statements:
  - **`#pragma pack(push, 2) // Set packing size to 2 bytes`**
  - **`#pragma pack(pop) // Pop 2 off, restoring old size`**

The getopt() function

- Assignments and projects for this class frequently use command-line options
- Dealing with them can be annoying, so POSIX provides **getopt()** to help:

```
int getopt(int argc, char * const argv[], const char *optstring);
```

  - **argc** and **argv** are the usual argument values passed into main()
  - **optstring** is a string containing:
    - Characters for any flag you want to give (such as **g** for a **-g** flag)
    - With a colon afterwards when there are arguments (such as **o:** if there's an argument for the **-o** flag)

Use of `getopt()`

- Typically, **`getopt()`** is called repeatedly
  - Whenever a legal option is found, the **`char`** value associated with that option is returned
    - If the option has an argument, it's stored in the global variable **`optarg`**
  - For unrecognized options, **`'?'`** is returned
  - When all options have been processed
    - **`getopt()`** returns **`-1`**
    - The global variable **`optind`** contains the index of the first element in **`argv`** that isn't an option or option argument
- **`getopt()`** moves around the contents of **`argv`** so that all the options appear first

getopt() example

- Consider a program that runs the following code in its **main()**

```
int value = 0;
while ((value = getopt(argc, argv, "co:")) != -1)
{
   switch (value)
   {
      case 'c': printf ("Compile but do not link\n"); break;
      case 'o': printf ("Output: %s\n", optarg); break;
   }
}
printf ("Current argument: %s\n", argv[optind]);
```

- It's looking for:
  - A **-c** option with no argument
  - A **-o** option with an argument

`getopt()` example continued

- Now this executable (**`program`**) is run:

```
./program goats.c -o result -c
```

- The output will be:

```
Output: result
Compile but do not link
Current argument: goats.c
```

- Likewise, **`argv`** will have been rearranged so that all options are first:

argv

0	1	2	3	4	5
./program	-o	result	-c	goats.c	NULL

Programming practice

- Write a program that uses **`getopt()`** to respond to the following command-line options:
  - **`-a`** Print "**`aardvark`**"
  - **`-b`** Print "**`bat`**"
  - **`-c`** Print "**`cat`**"
  - **`-m name`** Print "**`a mammal of type`** ***`name`***"
  - Any other flag Print "**`unknown animal`**"
- After all the flags have been processed, print how many non-flag arguments are left

POSIX IPC

POSIX

- POSIX is a series of standards for operating systems tied closely to UNIX standards
  - macOS is POSIX compliant in many ways but **not** for the IPC topics we're doing now
  - Linux is mostly POSIX compliant
  - Windows is not POSIX compliant, but there are environments like Cygwin that create mostly POSIX compliant environments
- For this kind of IPC, you have to use System V standards on macOS

POSIX IPC

- POSIX IPC function refer to IPC object named with a string that follows a particular format:
  - It must start with a slash
  - It must have one or more non-slash characters
  - Example: **`/comp3400_mqueue`**
- Object names must be unique
- These objects often appear as files in the file system, but you shouldn't interact with them using normal file commands
- POSIX IPC connections also have two other (familiar) values:
  - **`oflag`**: Access needed, a bitwise OR of flags like **`O_RDONLY`**, **`O_WRONLY`**, **`O_RDWR`**, **`O_CREAT`**, and **`O_EXCL`**
  - **`mode`**: Permissions, a bitwise OR of flags like **`S_IWUSR`** and **`S_IRGRP`**

Message Queues

Message queues

- Message queues are a form of message-passing IPC
- But don't we already have pipes and FIFOs?
- Differences from pipes:
  - Messages are sent as units: one whole message is retrieved at a time
  - Message queues use identifiers, not file descriptors, requiring special functions instead of **`read()`** and **`write()`**
  - Messages have priorities, not just first-in-first-out
  - Messages exist in the kernel, so killing off the sending process won't destroy them
- The big difference is structure:
  - Pipes and FIFOs send bytes, and the reader can read any number of available bytes at a time
  - Message queues send messages as units

POSIX message queues

- POSIX message queues have additional features that other implementations, like System V, might not have
- POSIX message queues:
  - Are only removed once they're closed by all processes using them
  - Include an asynchronous notification feature that allows processes to alerted when a message is available
  - Have priority levels for messages
  - Allow application developers to specify attributes (such as message size or capacity of the queue) via optional parameters passed when opening the queue

POSIX message queue functions

- `mqd_t mq_open (const char *name, int oflag, ...`
  `/* mode_t mode, struct mq_attr *attr */);`
  - Open (and possibly create) a POSIX message queue.
- `int mq_getattr(mqd_t mqdes, struct mq_attr *attr);`
  - Get the attributes associated with a given message queue
- `int mq_close (mqd_t mqdes);`
  - Close a message queue
- `int mq_unlink (const char *name);`
  - Remove a message queue's name (and the message queue itself, when all processes close it)
- `int mq_send (mqd_t mqdes, const char *msg_ptr,`
  `size_t msg_len, unsigned int msg_prio);`
  - Send a message with a given length and priority
- `ssize_t mq_receive (mqd_t mqdes, char *msg_ptr,`
  `size_t msg_len, unsigned int *msg_prio);`
  - Receive a message into a buffer and get its priority

Upcoming

Next time…

- Finish message queues
- Shared memory
- Semaphores

Reminders

- Finish Project 1
  - Due tonight by midnight!
- Read sections 3.7 and 3.8
- Exam 1 next Monday!