Week 4 - Friday

COMP 3400

Last time

- What did we talk about last time?
- Pipes

Questions?

Project 1

Pipes

Pipes and shell commands

- Let's go back to our command-line example:

```
sort foo.txt | grep -i error | head -n 10
```

- What's happening behind the scenes?
- The shell is calling **`fork()`** and **`exec()`** to run each of those processes
- Then, each process is linked to the next one with a pipe
- But how do those arbitrary processes know to read from or write to a pipe?
- **They don't**, so the shell magically changes **`stdout`** or **`stdin`** to pipe file descriptors

dup2()

- The **dup2()** function closes a new file descriptor and replaces it with an old file descriptor

```
int dup2 (int oldfd, int newfd);
```

- This function is used by the shell to close their **stdin** or **stdout** and replace it with an end of a pipe
- The syntax is confusing:
  - We keep the first file descriptor
  - We replace the second one

dup2 () example

- The output of Child 2 becomes the input of Child 1

```
assert ((child_pid = fork ()) >= 0); // Child 1
if (child_pid == 0)
  {
    close (pipefd[1]); // Close write end of pipe
    dup2 (pipefd[0], STDIN_FILENO); // Reading from stdin reads from pipe
    char *buffer = NULL;
    size_t size = 0;
    getline (&buffer, &size);  // Function that reads a line, resizing buffer as needed
    printf ("Received: '%s'\n", buffer);
    free (buffer);
    exit (0);
  }

assert ((child_pid = fork ()) >= 0); // Child 2
if (child_pid == 0)
  {
    close (pipefd[0]); // Close read end of pipe
    dup2 (pipefd[1], STDOUT_FILENO); // Writing to screen writes to pipe
    printf ("Now is the winter of our discontent\n");
    exit (0);
  }
close (pipefd[0]); // Parent closes both ends of the pipe for itself
close (pipefd[1]);
wait (NULL);  // Wait for children to finish
```

FIFOs

FIFOs

- Pipes are great for parent and child processes
  - Create the pipes in the parent, use them in the children
- But what if two unrelated processes want to share a pipe?
- **FIFOs** or **named pipes** are pipes associated with a file name
- These files can be seen in the file system, but they're special files intended only for use as pipes
- Naming:
  - In Linux, it's common to put these files in the `/tmp/` directory
  - It's important to pick a file name that's unlikely to collide with other FIFOs

The `mkfifo()` function

- The **`mkfifo()`** function is used to create a FIFO

```
int mkfifo (const char *path, mode_t mode);
```

- The **`mode`** is a bitwise OR of the permissions you want the FIFO to have (who can read and write)
- Using it creates the FIFO (which looks like a file), but programs still have to open it to use it and close it when done
- After the FIFO is done being used, the **`unlink()`** function removes the path from the file system

```
int unlink (const char *path);
```

FIFO example reader

- The following code creates a FIFO and reads **`int`** values until it gets a **0**

```
const char *FIFO = "/tmp/MY_FIFO";
assert (mkfifo (FIFO, S_IRUSR | S_IWUSR) == 0);
int fifo = open (FIFO, O_RDONLY);  // Open FIFO, delete if fails
if (fifo == -1)
  {
    fprintf (stderr, "Failed to open FIFO\n");
    unlink (FIFO);
    return 1;
  }

bool done = false;
while (!done)
  {
   int value = 0;
   if (read (fifo, &value, sizeof (int)) == sizeof (int)) {
       if (value == 0)
               done = true;
       else
               printf ("%d\n", value);
   }
close (fifo);
unlink (FIFO);
```

FIFO example writer

- The following code opens the FIFO and writes 6 **`int`** values to it

```
const char *FIFO = "/tmp/MY_FIFO";

int fifo = open (FIFO, O_WRONLY);
assert (fifo != -1);

for (int index = 5; index >= 0; index--)
  {
    write (fifo, &index, sizeof (int));
    sleep (1); // Sleep for a second before writing more
  }

close (fifo);
```

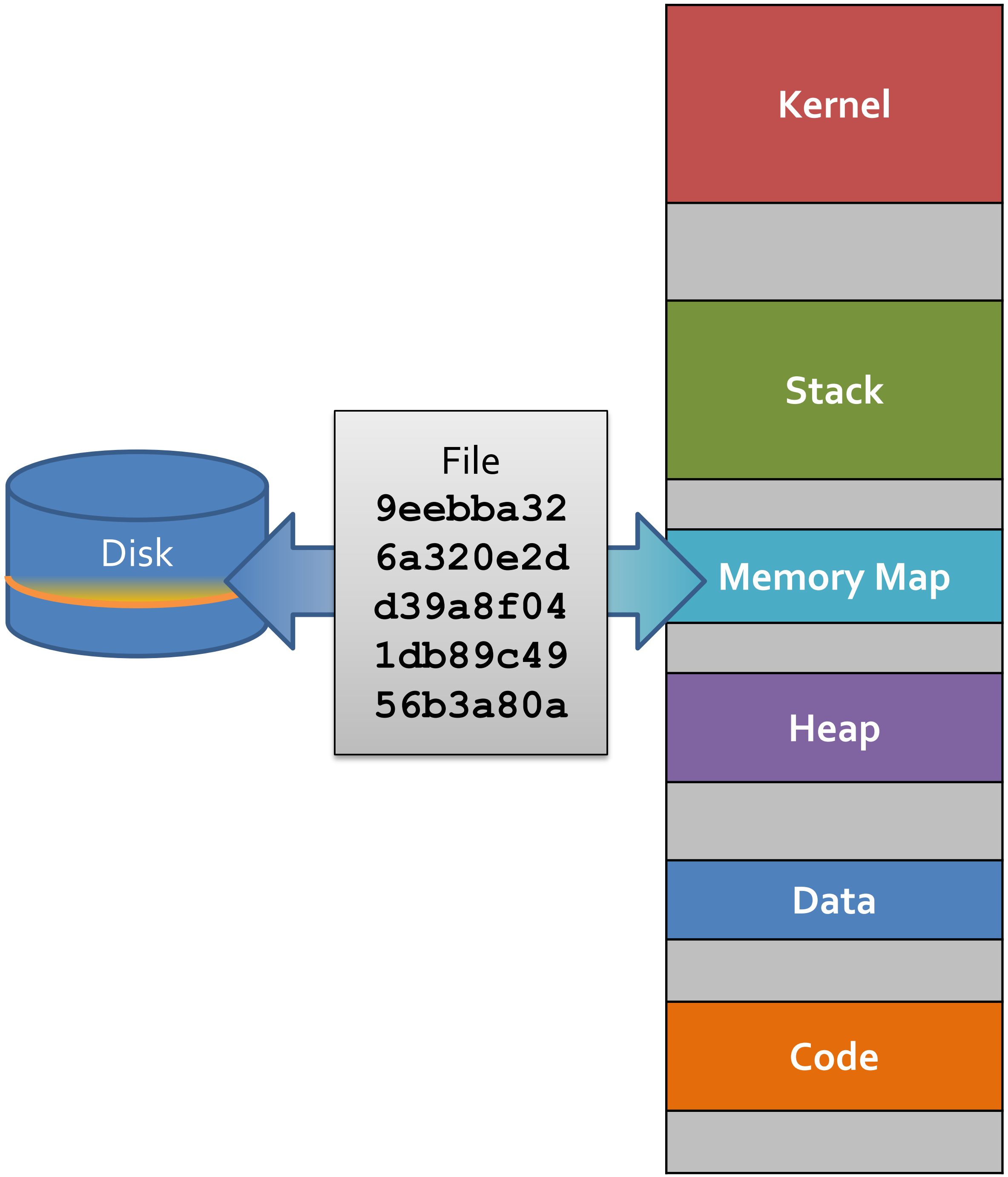
Memory-Mapped Files

Memory-mapped files

- Having covered pipes and FIFOs, we'll jump to the other side of the fence and talk about shared memory
- One shared memory technique are **memory-mapped files**
- A normal file is *mapped* into the virtual memory of a process
- Data can be read and written into that memory using normal pointer operations
  - And the data will magically get read and written to the file!
- One process can use memory-mapped files to interact with a file without using **`read()`** or **`write()`** calls
- But two or more processes can use memory-mapped files to exchange data directly

Visualization

- There's actually a special segment we haven't talked about in virtual memory before used just for memory mapping
  - Between the heap and the stack
- The virtual memory system is able to read only needed parts of the file into memory (often a page at a time)
- Storing data into this memory is eventually written back to the file



Advantages

- Over regular file access
  - Multiple processes can have read-only access to a common file
    - Often done with shared libraries, so that many different processes are able to access, for example, the same code for **`printf()`**
  - Programs can sometimes be simpler because there's no need to use **`fseek()`** to jump around a file
  - Reading files can be more efficient because the file contents don't have to be copied into the kernel's buffer cache
- Compared to other kinds of IPC
  - Writable memory-mapped files are fast for IPC
  - Unlike message passing, data continues to exist and can be read repeatedly

Mechanics

- The **mmap()** function returns memory mapped to a particular file descriptor

```
void *mmap (void *addr, size_t length, int prot, int flags,
  int fd, off_t offset);
```

  - **addr** is a suggestion for where the memory goes but should usually be **NULL**
  - **length** is how many bytes to map
  - **prot** are flags shown on the right that can be combined
  - **flags** are **MAP_SHARED** or **MAP_PRIVATE** (and others), depending on whether the area is shared
  - **fd** is an open file descriptor for a file
  - **offset** is the starting point inside the file

Protection	Actions permitted
PROT_NONE	May not be accessed
PROT_READ	Region can be read
PROT_WRITE	Region can be modified
PROT_EXEC	Region can be executed

Other useful functions

- The **`munmap()`** function unmaps an existing map

```
void munmap (void *addr, size_t length);
```

  - **`addr`** is the start of the mapped address
  - **`length`** is how much to unmap
- The **`msync()`** function synchronizes the file with the mapped memory

```
void msync (void *addr, size_t length, int flags);
```

  - **`MS_ASYNC`** flag returns immediately and **`MS_SYNC`** waits for the sync to complete

When updates happen

- If the goal is simply easy file interaction, you don't have to worry about when any updates are made
- But if you're trying to do IPC, the timing of when memory writes become disk writes becomes important
- The OS might occasionally write updated memory to files
- But some file systems won't write changes to files until the connection is closed
- If it's important, call the `msync()` function to make the updates happen

Example

- The following example checks to make sure that the 2nd, 3rd, and 4th bytes of an executable are "ELF", a marker of the executable and linking format used by Linux

```
int fd = open ("/bin/bash", O_RDONLY);
assert (fd != -1);

struct stat file_info;
assert (fstat (fd, &file_info) != -1);

// Map whole file for reading, unshared
char *mapping = mmap (NULL, file_info.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
assert (mapping != MAP_FAILED);

// Bytes 1 - 3 of the file must be 'E', 'L', 'F'
if (mapping[1] == 'E' && mapping[2] == 'L' && mapping[3] == 'F')
   printf("Valid executable!\n");
else
   printf("Invalid executable!\n");

munmap (mapping, file_info.st_size); // Unmap file and close it
close (fd);
```

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
};
```

Upcoming

Next time…

- POSIX vs. System V IPC
- Message queues

Reminders

- Finish Project 1!
  - Due **Monday** by midnight!
- Read sections 3.5 and 3.6