Week 4 - Wednesday

COMP 3400

Last time

- What did we talk about last time?
- Pointer review
- Interprocess communication overview

Questions?

Project 1

Interprocess Communication

Message passing

- There are many IPC approaches, but they can all be categorized as either **message passing** or **shared memory**
- Message passing:
  - Sender prepares a message
  - Sender makes a system call to request a data transfer
  - Kernel copies the message into a buffer
  - Receiver makes a system call to retrieve the data
  - Receiver copes the message into its own memory

Shared memory

- Shared memory IPC is completely different
- The processes decide on a chunk of virtual memory that will be used for IPC
- The processes make system calls to request that this memory is shared
- Once it's shared, processes can read and write from shared memory just like any other data in the program
- Mediation through the kernel isn't needed after the memory is shared

Pros and cons of message passing

- Message passing requires:
  - A system call to read
  - A system call to write
  - Copying the message into kernel memory
  - Copying the message into receiver memory
- Thus, sending lots of messages can cause a lot of overhead
- However, sending a small number of messages can be less expensive than setting up shared memory
- Message passing naturally handles the problem of synchronization
  - Making sure that timing doesn't corrupt memory

Pros and cons of shared memory

- It's computationally expensive to set up the shared memory
- But that's a one-time cost
- If two processes are sharing lots of messages, it can be more efficient to use a shared memory system
- Perhaps the more significant problem with shared memory is synchronization
  - Processes reading and writing the same memory can leave the memory in an inconsistent state
  - If one process executes `x += 100` while another executes `x -= 100`, the result could be the correct `x` or the incorrect `x + 100` or `x - 100`
- Tools must be used to guarantee synchronization

The IPC zoo

- Although all IPC techniques fall under the message passing or the shared memory model, there are other ways to categorize them:
  - For data exchange or purely for synchronization
  - As a stream of bytes or data with more structure
  - For local communication or for networked communication
- Note: People sometimes use the term "shared memory" to refer only to the technique using `shm_open()` and not memory-mapped files

IPC taxonomy

- Using the categories from the previous slide, we can list all of the IPC techniques that will be covered in this class

Technique	Model	Purpose	Granularity	Network
Pipe/FIFO	Message passing	Data exchange	Byte stream	Local
Socket	Message passing	Data exchange	Either	Either
Message queue	Message passing	Data exchange	Structured	Local
`shm()`	Shared memory	Data exchange	None	Local
Memory-mapped file	Shared memory	Data exchange	None	Local
Signal	Message passing	Synchronization	None	Local
Semaphore	Message passing	Synchronization	None	Local

- We just talked about signals, which are a form of IPC but very limited
- We'll cover sockets when we talk about networking

Pipes

- Pipes are a way to do message passing between two processes
  - The bytes flow in one direction
  - There's a different file descriptor for each end
  - Think of it like a pipe where water is poured into one end and comes out the other
- Internally, the shell uses pipes to communicate between two programs when you use the | operator on the command line

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

Pipe details

- Pipes only go in one direction
  - One end is the reading end, and the other is the writing end
- Pipes preserve order
  - The bytes read come out in the same order they were written
- Pipes have limited capacity
  - If a pipe is full, trying to write to the pipe will block until more is read
- Pipes are unstructured
  - It's all just bytes, so the processes have to know what kind of data to expect
- Messages smaller than `PIPE_BUF` are sent atomically
  - Two processes writing messages to a pipe will not get their messages garbled

Pipe mechanics

- The **`pipe()`** function takes an **`int`** array of length 2 to hold file descriptors corresponding to the ends of the pipe

```
int pipe (int pipefd[2]);
```

- It's convention to use element 0 for reading and element 1 for writing
- For piping between parent and child, the call to **`pipe()`** happens before the **`fork()`**, so that both have clones of the same file descriptors
- One process reads from the pipe and the other writes
- Each process closes the end that they're not using

Pipe example

```c
int pipefd[2];
char buffer[10];
memset (buffer, 0, sizeof (buffer));
int result = pipe (pipefd); // Open the pipe
assert (result >= 0);

pid_t child_pid = fork (); // Create child process
assert (child_pid >= 0);
if (child_pid == 0)
  {
    close (pipefd[1]); //  Child closes writing end
    ssize_t bytes_read = read (pipefd[0], buffer, 10); // Read from pipe
    if (bytes_read <= 0)
      exit (1);

    printf ("Child received: '%s'\n", buffer);
    exit (0);
  }

close (pipefd[0]); // Parent closes the reading end
strncpy (buffer, "hello", sizeof (buffer));
printf ("Parent is sending '%s'\n", buffer);
write (pipefd[1], buffer, sizeof (buffer)); // Parent sends "hello"
wait (NULL); // Wait for child to terminate
```

Practice

- Let's write a program that:
  - Creates a pipe
  - Spawns a child
  - Reads words from the command line (until "exit" is entered)
  - Sends those words to the child through the pipe
  - Kills the child when done
- The child:
  - Reads words
  - Prints them out

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
```

Ticket Out the Door

Upcoming

Next time…

- FIFOs
- Shared memory with memory-mapped files

Reminders

- Keep working on Project 1
  - Due Friday by midnight!
- Read section 3.4