CS 111: Operating System Principles

Lecture 6

Basic IPC

3.0.1

Jon Eyolfson

October 12, 2021

IPC is Transferring Bytes Between Two or More Processes

Reading and writing files is a form of IPC

For a process you can read the input, and write the output

- Think about Lab 0

The read and write system calls allow any bytes

A Simple Process Could Write Everything It Reads

See: `lecture-06/read-write-example.c`

We `read` from standard in, and `write` to standard out

  Does this remind you of any program you’ve seen before?

If we run it in our terminal without arguments, it’ll wait for input

  Press Ctrl+D when you’re done to send end-of-file (EOF)

`read` Just Reads Data from a File Descriptor

See: `man 2 read`

There’s no EOF character, `read` just returns 0 bytes read

- The kernel returns 0 on a closed file descriptor

We need to check for errors!

- Save `errno` if you’re using another function that may set it

`write` Just Writes Data to a File Descriptor

See: `man 2 write`

It returns the number of bytes written, you can’t assume it’s always successful

- Save `errno` if you’re using another function that may set it

Both ends of the read and write have a corresponding write and read

- This makes two communication channels with command line programs

The Standard File Descriptors Are Powerful

We could close standard input (freeing file descriptor 0) and open a file instead

- Linux uses the lowest available file descriptor for new ones

See: `lecture-06/open-example.c` and `man 2 open`

Without changing the core code, it now works with multiple input types

You could type, or use a file

Your Shell Will Let You Redirect Standard File Descriptors

Instead of running `./open-example open-example.c` we could run:

```
./open-example < open-example.c
```

Your shell will do the `open` for you and replace the standard input

We didn't actually have to write that!

You could also redirect across multiple processes

```
cat open-example.c | ./open-example
```

Piping in Your Shell Connects Two Processes Together

See: `examples/lecture-06/pipe-example.c` and `man 2 pipe`

In `./p1 | ./p2` the shell connects: `p1`'s `stdout` to `p2`'s `stdin`

The kernel has a `pipe` system call that returns two file descriptors

  `fd_pair[0]` is for `read` and `fd_pair[1]` is for `write`

This forms a one-way communication channel

What happens if you leave the write end open?

Your Shell Properly Handles All the File Descriptors

This includes changing file descriptors, and closing them properly

You can use `dup2` to move a file descriptor to a new one

- If the new one is already open, it'll `close` it first

See: `man 2 dup2` and `man 2 close`

How do you give processes different file descriptors?

- `fork` copies all the file descriptors to the new process

Signals are a Form of IPC that Interrupts

You could also press Ctrl+C to stop `./open-example`

- This interrupts your programs execution and exits early

Kernel sends a number to your program indicating the type of signal

- Kernel default handlers either ignore the signal or terminate your process

Ctrl+C sends `SIGINT` (interrupt from keyboard)

If the default handler occurs the exit code will be 128 + signal number

You Can Set Your Own Signal Handlers with `sigaction`

See: `lecture-06/signal-example.c` and `man 2 sigaction`

You just declare a function that doesn't return a value, and has an `int` argument

The integer is the signal number

Some numbers are non standard, here a few from Linux x86-64:

- 2: `SIGINT` (interrupt from keyboard)
- 9: `SIGKILL` (terminate immediately)
- 11: `SIGSEGV` (memory access violation)
- 15: `SIGTERM` (terminate)

A Signal Pauses Your Process and Runs the Signal Handler

Your process can be interrupted at any point in execution

- Your process resumes after the signal handler finishes

This is an example of concurrency, your process switches execution

- You have to be careful what you write here

Run `./signal-example` and press Ctrl+C

You Need to Account for Interrupted System Calls

You should see:

```
Ignoring signal 2
read: Interrupted system call
```

We can rewrite it to retry interrupted system calls

See: `lecture-06/signal-example-2.c`

Now the program continues when we press Ctrl+C

You Can Send Signals to Processes with Their PID

You can use the command: `kill`

- It is also a system call, taking a `pid` and signal number

Find a processes' ID with `pidof`, e.g. `pidof ./signal-example-2`

After use `kill <pid>`, which by default sends `SIGTERM`

Use `kill -9 <pid>` to tell the kernel to terminate the process

- Process won't terminate if it's in uninterruptible sleep

Shared Memory Allows Two Processes to Access the Same Memory

See: `lecture-06/shared-memory-example.c` and `man 3 shm_open`

You use `shm_open` which returns a file descriptor

You can think of it as a new location to read and write bytes to

- This needs to be resized with `ftruncate`

Note: on some implementations this just opens a file in `/dev/shm`

`mmap` Allows You to Memory Map the Contents of a File Descriptor

See: `man 3 mmap`

Instead of using `read` and `write` system calls, you just access memory

- The operating system is responsible for management

Instead of accessing the file sequentially, you can access any part of it

You can `mmap` regular files as well!

We Explored Basic IPC in an Operating System

Some basic IPC includes:

- `read` and `write` through file descriptors (could be a regular file)
- Redirecting file descriptors for communcation
- Pipes (which you'll explore)
- Signals
- Shared Memory