ECE 344: Operating Systems
Lecture 6

Basic IPC

1.0.1

Jon Eyolfson
September 20/21, 2022

IPC is Transferring Bytes Between Two or More Processes

Reading and writing files is a form of IPC

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
```

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

Most Operations Are Non-Blocking

A non-blocking call returns immediately, and you check if something occurs

To turn `wait` into a non-blocking call, use `waitpid` with `WNOHANG` in `options`

To react to changes to a non-blocking call, we can either use a *poll* or *interrupt*

Polling Continuously Calls the Function and Checks for Changes

See: `lecture-06/wait-poll-example.c`

We call `waitpid` over and over until the child exits

Note: some hardware behaves like this,
the kernel may have to check for changes

What's the drawback of this approach?

An Interrupt Instead Occurs Right After the Change

See: `lecture-06/wait-interrupt-example.c`

Instead of calling `wait` or `waitpid` from `main`, we can do it in the interrupt handler

The kernel sends the `SIGCHLD` whenever one of its children exit

This idea also applies to the kernel, hardware can generate interrupts

Interrupt Handlers Run to Completion

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

An interrupt may occur while an interrupt handler is already running

All interrupt handler code must be reentrant

> You need to be able to pause execution,
> execute another call (to the same function),
> and resume execution

On a CPU, There's 3 Kinds of "Interrupts"

Interrupt

Triggered by external hardware,
handled by the kernel (needs to respond quickly)

Trap

Triggered by users (a system call is a trap),
handled by the kernel (calling process suspended)

Exception

Triggered by abnormal control flow (divide by zero, illegal memory access),
default handler is the kernel (calling process suspended),
the process can optionally handle these themselves

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
- Signals

Signals are like interrupts for user processes

The kernel has to handle all 3 kinds of “interrupts”