CS 521: Systems Programming

Inter-Process Communication

Lecture 10

Inter-Process Communication

- We previously discussed how the host OS tries its best to isolate processes
  - Processes should not be able to interfere with one another
  - To do ***privileged*** operations, we need to go through the kernel with **system calls**
- However, it's often useful to have processes communicate
  - Inter-Process Communication (IPC)
- IPC gives us safe, well-defined ways to communicate

Why IPC?

- Processes need to share data
- “Data” can mean a lot of things:
  - Plain text
  - An image, video, program
  - A message containing commands or other types of information
- Without a well-defined interface, getting processes to communicate descends into madness

An Example

1. You double-click a web link saved to your desktop
2. The OS determines which program is responsible for handling HTTP/S URIs
3. The program is launched if it isn't already running
4. The OS delivers a message to the program:
5. OPEN https://google.com

Types of IPC

- We will cover three types of IPC in this class (although there are many others):
  - Files
  - Signals
  - Pipes
- You might be surprised that files could be considered a form of IPC, but it's actually one of the easiest and simplest ways to communicate between processes!

Today's Schedule

- Files
- Signals
- Pipes

Today's Schedule

- **Files**
- Signals
- Pipes

Files

- Save a file to disk with one application, open it with another application
  - Needs a *file system* to make this happen
  - On our VMs, we're using `ext4`. A recent Mac might use `apfs`, and Windows `NTFS` (or maybe `XFAT`…)
- What happens when two applications open the same file?
  - Coordinate via **file locks**
  - Can lock an entire file or only a portion

Opening a File

- We have used `fopen()` to open and read files
- There is another, more low-level option: `open()`
  - This is a system call
  - (and, technically, `fopen` from the C library calls `open` on Linux)
- `open` returns a ***file descriptor*** – an integer that represents the opened file
  - This decouples the file's absolute path in the file system (e.g., `/usr/bin/something`) from I/O operations

File Descriptors

- `stdin`, `stdout`, and `stderr` have file descriptors
- The file abstraction is used thoroughly in Unix systems (see `/dev` for devices)
- Once you've opened a file descriptor, you can read/write the contents of the file or even redirect the stream somewhere else

Redirecting Streams: dup2

- `dup2` allows us to redirect streams
  - `int dup2(int fildes, int fildes2);`
- Let's say we want to make our standard output stream go to a file we just opened
- We'll do:
  - `dup2(fd, STDOUT_FILENO);`
- This also deallocates (closes) the second fd
  - You won't see text printing directly to your terminal anymore

Example: Redirecting to a File

- Combine `open` and `dup2`:
  - `int output = open("output.txt",`
    `O_CREAT | O_WRONLY | O_TRUNC, 0666);`
  - `dup2(output, STDOUT_FILENO);`
- This is exactly what our shell does when we use `<` and `>`
- `cat /etc/passwd > some_file`
  - Opens "some_file" and then redirects the output of the child process to that file instead!

Redirection Workflow

Let's say your shell encounters `>` in the command line...

1. Use `fork` to create a new process
2. Open the file that comes after the `>`
3. Redirect `stdout` to the file with `dup2`
4. Call `exec` to execute the program
   - This is part of the reason why `fork` and `exec` are split into two separate parts

Demo: io-redir.c

Today's Schedule

- Files
- **Signals**
- Pipes

Signals

- Signals are software-based *interrupts*
  - Basically a notification sent to the process
- The kernel uses signals to inform processes when events occur
- Handling a signal causes a jump in your program's logic to a *signal handler*
  - You can use a null signal handler to ignore particular signals

Demo: signal.c

Events

- What kind of events are reported via signals?
- It depends on the kernel
- To find out, use:

```
/bin/kill -l
```

- Wait, what?!
  - That's right: kill is used to send signals to processes
  - It doesn't necessarily 'kill' the process in doing so
    - But it can!

Terminating a Process

- You've already been using signals quite a bit (but maybe didn't realize)
  - Ever hit Ctrl+C to stop a running program?
  - it sends SIGINT to the process
- Each signal is prefixed with SIG
- Processes can choose how to deal with signals when they are received
  - Including ignoring them… usually

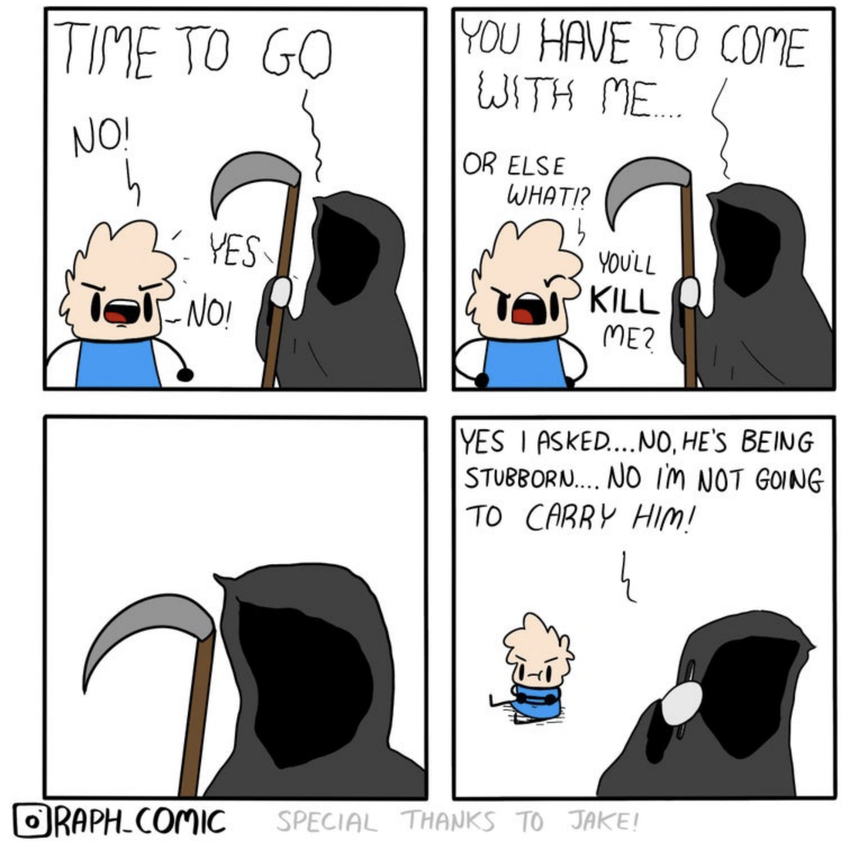
Demo: unkillable.c

Special Signals

- SIGSTOP and SIGKILL cannot be caught or ignored
- SIGSTOP – stops (pauses) the process: Ctrl+Z
  - SIGCONT tells a paused process to continue
- SIGKILL – terminates the process, no questions asked
  - You may have heard of `kill -9 <pid>`
  - 9 is SIGKILL

Using kill -9

- Occasionally a process will not respond to a SIGTERM, SIGINT, etc.
- This is the appropriate time to use SIGKILL



Signal Handling

- Set up a signal handler with `signal`:

```
signal(SIGINT, sigint_handler);
```

  - Will call `sigint_handler` every time a SIGINT is received

- Then implement the signal handling logic:

```
void sigint_handler(int signo) { ... }
```

OS Signal Transmission Process

1. First, a process initiates the signal
   - Terminal Emulator: user pressed Ctrl+C, so
   - I should send SIGTERM to the current process
2. The kernel receives the signal request
3. Permissions are verified
   - Can this user really send a signal to PID 3241?
4. The signal is delivered to the process

Reacting to a Signal

- If a process is busy doing something, it will be **interrupted** by the signal
- Jumps from the current instruction to the signal handler
  - (or performs the default operation if there is no signal handler)
- Jumps back to where it was when the handler logic completes

Segmentation Violation

- Our good friend, the segmentation violation (aka segfault) is also a signal
  - SIGSEGV
- Bus error: SIGBUS
- So if segfaults are getting you down, try blocking them!
  - What could go wrong?!

Sending a Signal

- Not all signals are sent via key combinations from the shell… We can send them programmatically or via the command line
- Let's send a SIGUSR1 signal to process 324:

```
kill -s SIGUSR1 324
```

- Simple as that!
- Or, in C:

```
int kill(pid_t pid, int signum);
```

Tracking Children

- `SIGCHLD` is sent to the parent of a child process when it exits, is interrupted, or resumes execution
- Useful in scenarios where the parent process needs to be notified about child events
  - or, in other words, when the parent is not already `wait()`ing on the child
  - Job list in the shell: when `SIGCHLD` is received, do a non-blocking `waitpid` to determine which process exited and remove it from the list (if backgrounded)

Today's Schedule

- Files
- Signals
- **Pipes**

Pipes [1/2]

- Pipes are a common way for programs to communicate on Unix systems
  - `cat /etc/something | sort | head -n5`
- Most useful for sharing unstructured data (such a text) between processes
- They work like how they sound: if you want to send data to another process, send it through the pipe

Pipes [2/2]

- Pipes are one of the fundamental forms of Unix IPC
- With pipes, we can "glue" several utilities together:
  - `grep neato file.txt | sort`
  - This will search for "neato" in file.txt and print each match
    - Next, these matches get sent over to the 'sort' utility
- Just like with I/O redirection, this is facilitated by `dup2`

In the Shell

- As we've seen, pipes are used frequently in the shell
- We can mix and match different utilities, and they all work well together
  - Awesome!
- Some genius must have designed all these programs to work this way, right?
  - Well, no. They all just read from `stdin` and then write to `stdout` (and `stderr`)
  - No coordination required between developers

Builtins vs. External Programs

- When you enter 'ls' in your shell, you're running a program
- This functionality is **NOT** built into your shell. Bash simply finds and runs the 'ls' program. That's it!
- There are some shell "commands" that actually aren't programs, called **built-ins**
  - `history`
  - `exit`
  - `cd` – why does this need to be a built-in?

Going to the Source

- I have posted a video from Bell Labs on the schedule that discusses several design aspects of Unix
- Discussion on pipes starts right around the 5 minute mark

The pipe function

- Now back to pipes: we can create them with the pipe() function
  - Returns a set of file descriptors: the **input** and **output** sides of the pipe
- Pipes aren't very useful if they aren't connected to anything, though
  - We can do this by `fork()`ing another process

Piping to Another Process

- After calling `fork()`, both processes have a copy of the pipe file descriptors
- Pipes only operate in one direction, though, so we need to close the appropriate ends of the pipe
  - You can think of a forked() pipe as one with four ends: two input and output ends each
  - We eliminate the ends we don’t need to control the direction of data flow
  - Amazing ASCII art drawing: `>---<`

Controlling Flow

- To control data flow through the pipe, we close the ends we won't use
- For example:
  - Child process closes **FD 0** and reads from **FD 1**
  - Parent process closes **FD 1** and writes to **FD 0**

Async Process Creation

- You may be wondering: what good are pipes when we have to start all the cooperating processes?
- There's actually another option: **FIFO**s, aka **named pipes**
- Create with the `mkfifo` command, then open as you would a regular file descriptor

Redirecting Streams to a Pipe

- Let's say we want to make our standard output stream go through the pipe we just created
  - `int fd[2];`
    `pipe(fd);`
- We'll do:
  - `dup2(fd[0], STDOUT_FILENO);`

Wrapping Up

- We've seen only a few possibilities for IPC!
- Another option: **sockets**
  - Communication… even over the network!
- Many Unix systems use ***D-Bus*** for more advanced IPC
- Windows has a similar concept: ***Windows Messages***
  - Windows applications are **event based**
    - Almost everything that happens on Windows has an **event** associated with it (`WM_MOUSEMOVE`, changing resolution, etc.)

Fun: Undelivered Events