CS 326: Operating Systems

Inter-Process Communication

Lecture 8

Today's Schedule

- IPC Background
- Basic IPC
- Signals

Today's Schedule

- **IPC Background**
- Basic IPC
- Signals

Inter-Process Communication

- We've spent considerable time discussing how processes are isolated
  - Memory space
  - Illusion of complete ownership of the CPU
- It's often useful to have processes communicate with each other
  - 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

How do we do this?!

- Given what we know about operating systems so far, what are some ways we can accomplish IPC?
- …

IPC Mechanisms

- Files
- Signals
- Pipes
- Message Queues
- Mapped Memory
- Sockets
- Shared communication channels (e.g., D-Bus)

So Many Options!

- Wow, there sure are a lot of ways to do IPC!
- It's almost like we *want* processes to talk often
  - Why did we set up all these barriers?
- Maybe we could design a new OS where processes have access to each others' memory
  - Wait, what?!
  - No! Stop! **NO**! No! No!!!

Today's Schedule

- IPC Background
- **Basic IPC**
- Signals

Basic IPC Concepts

- We can’t study each IPC mechanism in depth, so we will focus on the most common
- But first, let’s look at:
  - Files
  - Message Queues
  - Shared communication channels

Files

- The most basic form of IPC is done through files
- Save a file to disk with one application, open it with another application
- What happens when two applications open the same file?
  - Coordinate via **file locks**
  - Can lock an entire file or only a portion

Message Queues

- POSIX Message Queues provide a direct communication mechanism between processes
- Messages are simple text strings
- Queues are identified by a **name**
- Great for sending text, but extremely old and out of date
  - Not supported by macOS
  - The Linux implementation (apparently?) has bugs that haven't been fixed for years

D-Bus

- The modern way to send messages (and do much more) on Unix systems is D-Bus
  - multiplatform **system message bus**
- Allows for IPC, process discovery, broadcasting
- Additionally provides **remote procedure calls**
  - Process A calls a method exposed by process B
  - One of the basic building blocks of distributed computing

Windows Messages [1/2]

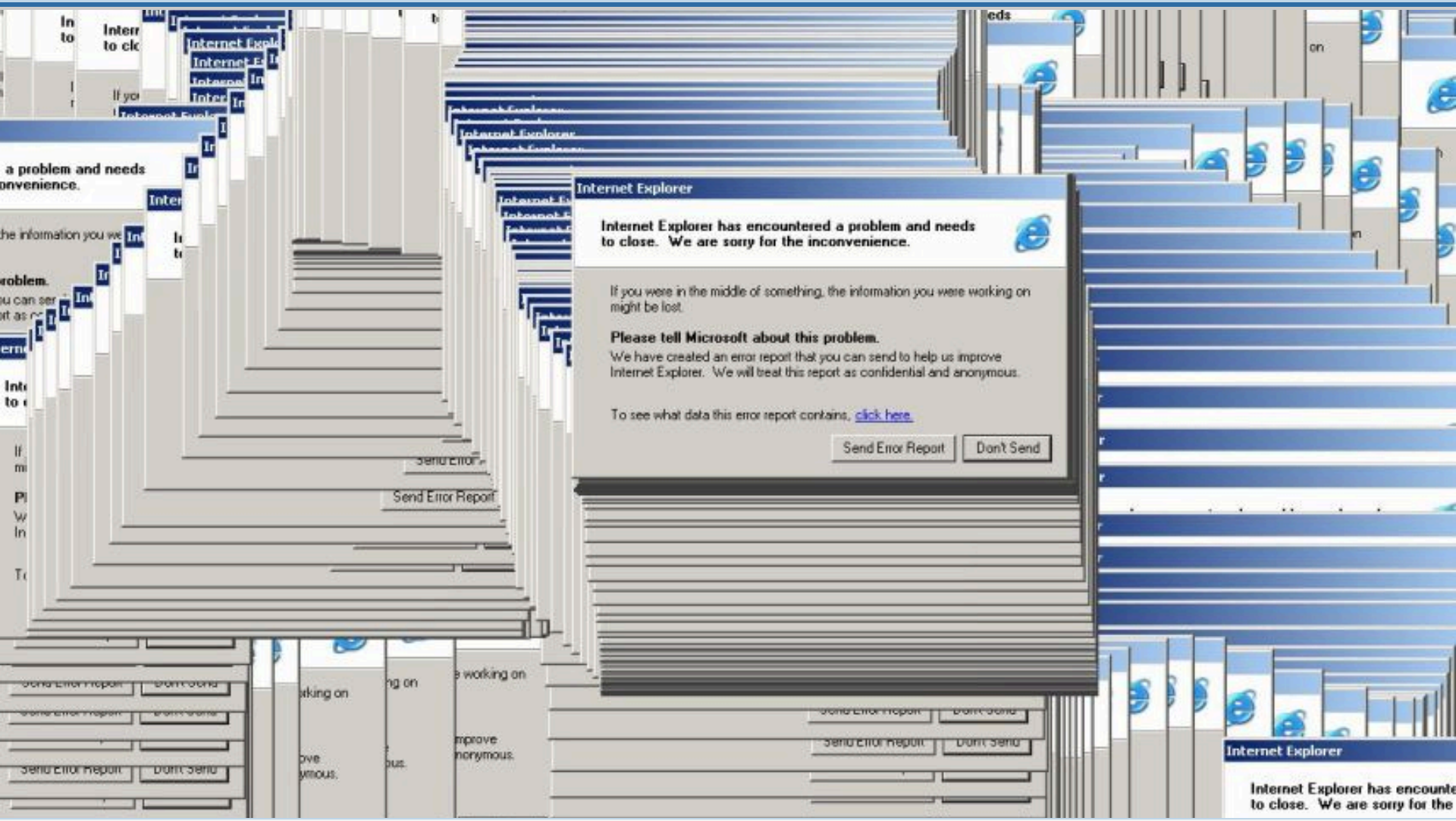
- Windows has a similar concept: Windows Messages
- Windows applications are **event based**
  - A process is automatically subscribed to a default set of events
  - Can subscribe to more if needed
- Almost everything that happens on a Windows system is exposed as these events
  - Display resolution changed, user logged out, etc.

Windows Messages [2/2]

- Even UI elements are triggered via Messages:
  - WM_MOUSEMOVE
- Each Windows process has a message queue associated with it and has some logic along the lines of the following:

```
while (true) {
   GetMessage(&msg, NULL, 0, 0);
...
}
```

Undelivered Events



Shared Communication Channels

- Operating systems tend to support their own types of messaging protocols
  - The most efficient way to communicate is influenced by the design of the kernel
- macOS: microkernel, emphasis on message passing
- Minimalistic OS: get basic mapped memory working

Today's Schedule

- IPC Background
- Basic IPC
- **Signals**

Signals

- The first type of IPC we'll work with in class is **signals**
- Signals are software-based interrupts
  - (as opposed to the hardware-based interrupts we've been talking about with scheduling)
- The kernel uses signals to inform processes when events occur
- **NOTE**: xv6 does *not* support 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
```

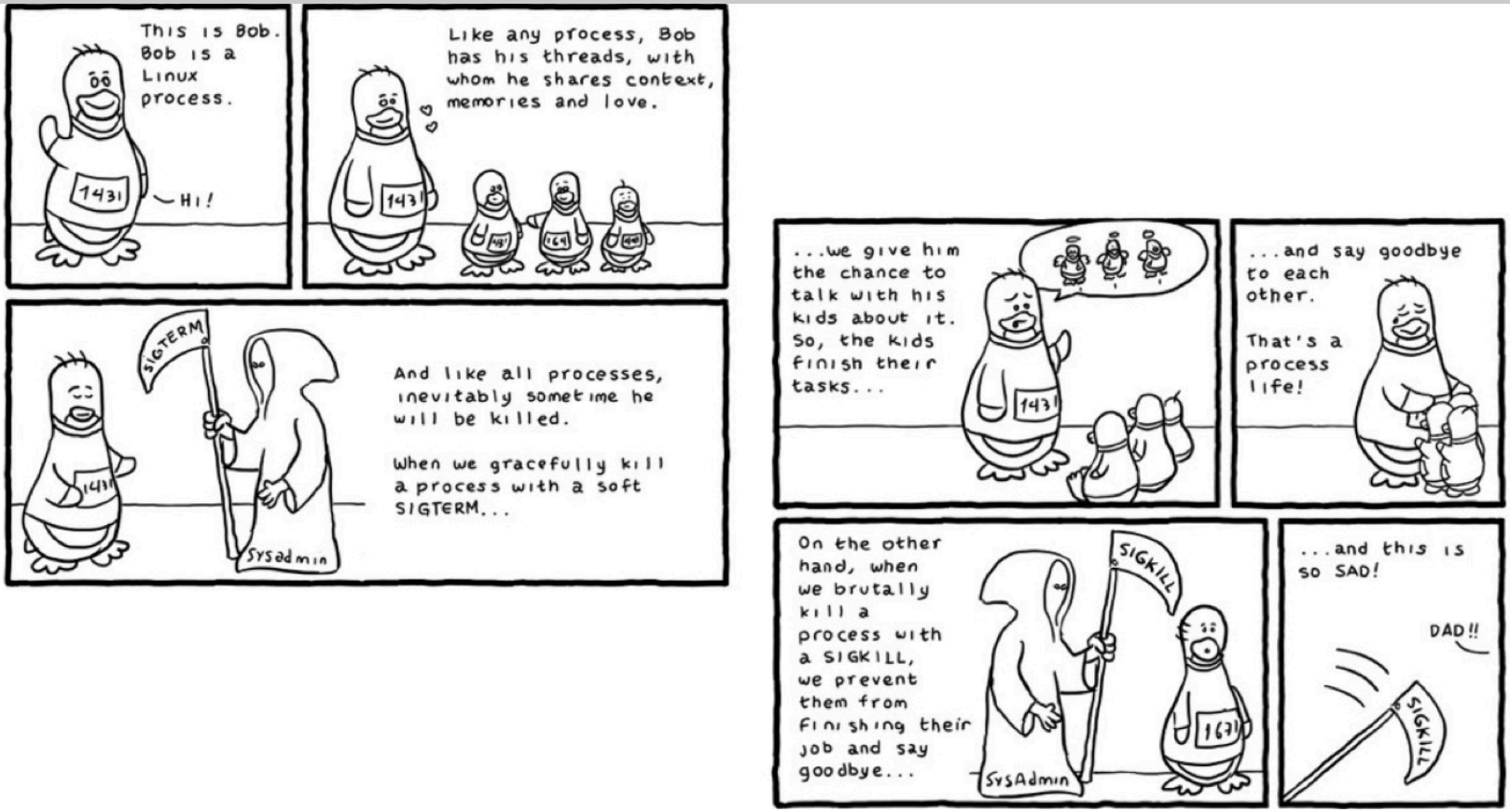
- Wait, what?!
  - That's right: kill is used to send signals to processes
  - It doesn't necessarily 'kill' the process in doing so
    - But it can!

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

The Life and Times of Processes



The Life and Times of Processes

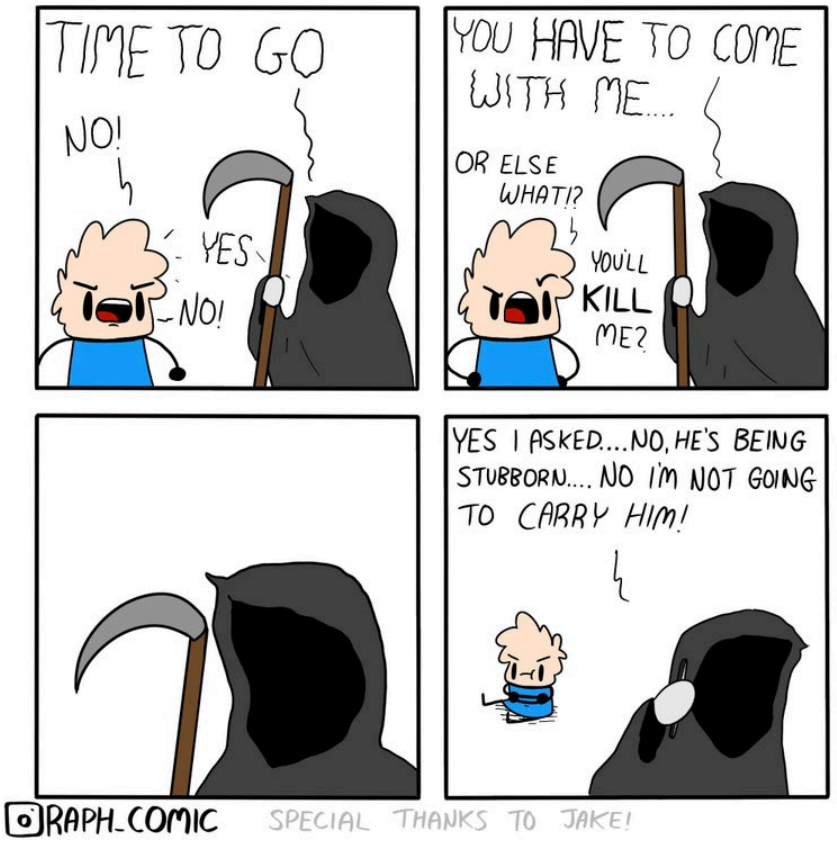


Special Signals

- SIGSTOP and SIGKILL cannot be caught or ignored
- SIGSTOP – stops (pauses) the process: Ctrl+Z
- 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



Stop and Continue

- SIGSTOP pauses a running process
  - What process state will be entered in this case?
- SIGCONT tells a paused process to continue
- These signals are used for job control in the shell: you can suspend a process with ^Z (Ctrl+Z)
- What else might STOP/CONT be useful for?

Alarms

```
#include <unistd.h>
int alarm(unsigned int seconds);
```

- Here's another useful signal: SIGALRM
- Use the `alarm` function to set up a timer
  - When the timer elapses, your program will receive the SIGALRM event

Custom Signals

- You can also handle program-specific signals
- SIGUSR1 and SIGUSR2
  - Programs can implement their own custom signal-handling logic for these
  - Added because some programs were doing crazy things like overriding SIGTERM for custom functions
- Unfortunately, you can’t invent your own signals
  - No SIGAWESOME… yet

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);
```

Signals: Pros

- Easy to use
- They’re simple!
- No need to know anything other than the process ID
- Custom functionality: SIGUSR* signals
- Example: SIGHUP on apache/nginx/etc

Signals: Cons

- They’re simple
- Can’t send a picture as a signal
- Causes a jump in your program logic, whether you want that or not
  - (ex) sockets: we can listen when we want messages

Wrapping Up

- Given how basic signals are
  (just a symbolic message… basically an **int**)
  there has to be more to this story
- The next type of IPC we look at will be **pipes**