COMP 3430

Operating Systems

June 3rd, 2019

~~Next~~ *This* week

Let's take a look at the schedule.

Pixabay License

Assignment 2

Let's take a look at A2, and **run** part 1.

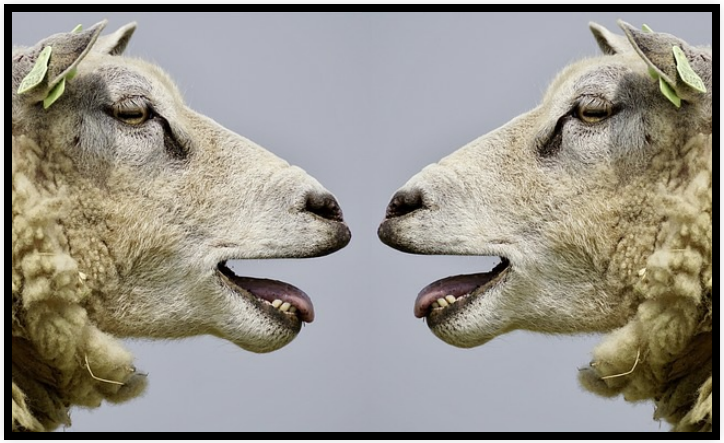
Goals

By the end of today's lecture, you should be able to:

- Describe different strategies for message passing
- Write a UNIX program that handles signals (syscalls)
- Explain how pipes and signals can be used as a messaging mechanism
- Write a UNIX program that uses pipes for messaging
- Evaluate and choose a strategy for IPC given a problem.

IPC

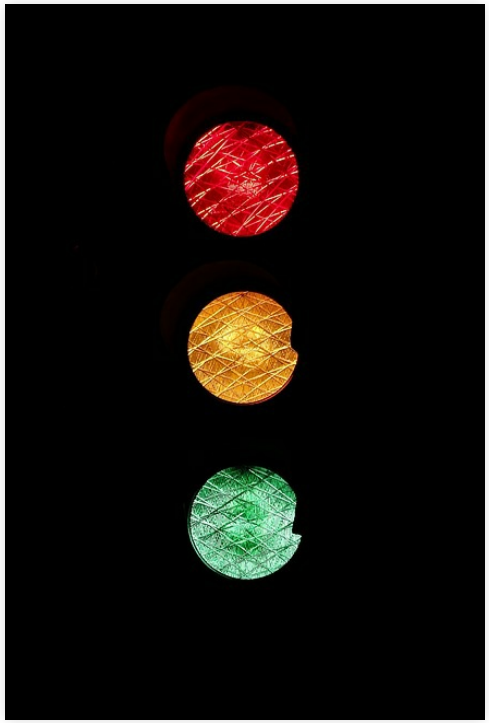
- Signals ✔
- Files (pipes)
- Shared memory



Pixabay License

Quiz

Let’s make sure we know about signals.

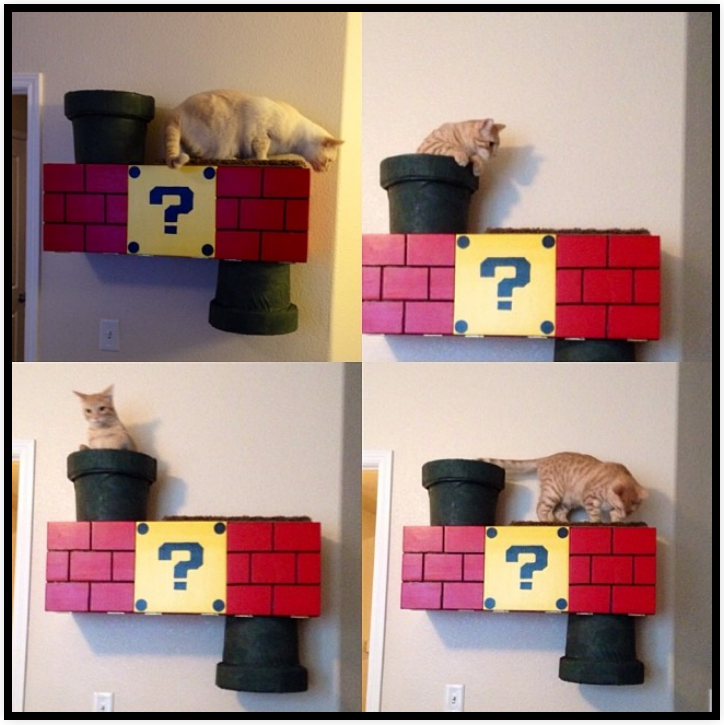


Pixabay License

Pipes are … *better*?

Let's take a look at some examples.

1. Anonymous pipes. (`anonymous_pipe.c`) ✔
2. Named pipes. (`client.c` and `server.c`)



More like *this* kind of pipe. © Wes Woodward

What's the kernel doing?

- The kernel *mediates* this whole process.
  - The kernel performs a *switch* (context? mode? …?)
- Think about it: What kind of **data structures** might the kernel use to implement pipes?
  - Let's take a look at `pipe.c`
  - Observations to make: data structures, locking, etc.

Pixabay License

Pipes!

Pipes are *much* more flexible than signals.

- We can actually get *information* with the message.
  - … but we still need *structure* (protocols) (COMP 3010)
- Kernel *still* mediates the whole process.
- We can send messages to processes on *other* machines (it's just a file!)
  - *Highly* similar to `sockets`.
- Pipes **do not** solve all problems.
  - Pipes can be **slow** .
  - Pipes have limitations on numbers of readers and writers.

Shared memory

- Shared memory with threads?
  - It's just there.
- Shared memory with processes?
  - Use `mmap` (`man 2 mmap`)
- Let's look at the `man` page.

Pixabay License

Shared memory

- *Straightforward*. (kind of like `malloc`)
- Fast!
- Dangerous!
- Fast!!
- Dangerous!!
  - Remember `pthread_mutex_init` et al?
  - Yeah. We've got the same bag of problems.
- There is no strict limit on the number of readers and writers.
  - It's *your* problem to manage access.

Which one?

Let’s take a look at a couple of scenarios/problems and try to decide which IPC mechanism is the most appropriate.

Pixabay License

Scenario 1

I need to write a program that can download multiple files from the internet simultaneously. I'm going to write this program using `fork`, where I will `fork` *n* child processes, and each one will be responsible for downloading a specific file.

I want to know when each file has been completed downloading, so the children will have to communicate this information to me.

Which kind of IPC should I use?

Scenario 2

I have the same problem (multiple file downloads concurrently), but this time I want to show **progress** as the files are downloaded by the processes.

Which kind of IPC should I use?

Scenario 3

I need to write a program that will implement a **process pool**. Each of the processes will be part of a web server where

1. Requests can be handled by *any* of the processes in the pool,
2. Requests can have *session* data (e.g., login information).

What kind of IPC should I use?

IPC

- Now we've got three methods for communicating among processes.
- One of those methods is *still* heavily file-based.
  - … but that's OK.
- These are *all* mediated by the kernel w/ system calls.
  - They *sometimes* share the same problems as threads.

"Businessman". Pixabay License

IPC decisions

- Choosing an IPC method means considering tradeoffs:
  1. Complexity of implementation.
  2. Communication requirements.
  3. Speed.

Semaphores

- Locks are a primitive protection mechanism.
  - Only **one**… *thing* can have access to the locked region.
- A semaphore is a **generalization** of locking.
  - Locks/mutexes answer the question: "Is this thing available?" (binary yes/no)
  - Semaphore answers the question: "How many of this thing is available?" (may be 0 or more)

Public Domain