CptS 360 (System Programming)
Unit 15: Interprocess Communication

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- Processes need to talk to each other.
- Two processes on the same system can communicate more efficiently than two processes on separate systems.
- Daemons and some servers depend on IPC.

References

- Stevens & Rago Ch. 15
- *man* pages
- Rochkind, "Advanced Unix Programming" (classic)
- Stones & Matthew "Beginning Linux Programming"

Overview

- IPC is how processes talk to each other intra-system.
- This is mostly old stuff that's been in UNIX for many years.
- It's still heavily used.
- Linux has both BSD and System V facilities.

Pipes

- In UNIX from day 1.
- Originally driven by limited (16 bit) address space.
- *pipe(2)*
  - creates a pair of pipes
  - `fd[0]` opened for reading
  - `fd[1]` opened for writing.
- Don't confuse with *dup(2)*.
- Half-duplex. (One way.)
- Only works between processes with a common ancestor.
- Pipes classed as FIFOs for purposes of *fstat(2)*.

(see `demos/d`n`_pipe_call` and `demos/d`n`_pipe_comm`)

Pipes and Forking

- Pipes are pretty useless within a single process.
- But during a `fork()`:
  - Child inherits parent's open `fd`'s, including `pipe()`'d ones.
  - Each process closes one `fd[]` element.
  - Can use this to redirect stdin or stdout to another program.
- a prepackaged way to do this is...

popen(3) and *pclose(3)*

- unidirectional
- 1st argument passed to `/bin/sh`
  - so it can even be a shell command, like `"cd /home/bobl; find ."`
- note use of shell syntax

(see `demos/d`n`_popen_pager`)

Parent/Child Synchronization

- A process reading from a pipe will block until the process at the other end writes something.
- A process writing to a pipe will block until the process at the other end reads it.
- We can use this to sychronize parent and child processes.

(see `demos/d`n`_tell_wait`)

Coprocesses

- Pipe unidirectionality seems limiting: Can we do better?
- Coprocesses: two or more processes passing data back and forth between them.
- Two implementations:
  - If pipe's are bidirectional (full duplex), use a single bidirectional pipe
  - If pipe's are unidirectional (half duplex), use two unidirectional pipes (This is the Linux case.)

FIFOs I

- otherwise known as "named pipes"
- unidirectional
- *mkfifo(3)*
- There's also a shell command:

  ```
  $ mkfifo myfifo
  $ echo "hello, fifo" >myfifo
  ```

  (in another window, but same directory)

  ```
  $ cat <myfifo
  $ rm myfifo
  ```

  Then use standard or low-level I/O as usual.
- can be used to duplicate streams

FIFOs II

- Also works for client-server *if* name of server's FIFO is advertised ("well-known").
- Sending stuff back to the client is difficult unless the client sends its PID or some other identifier. Then server can open client-specific FIFO.
- To prevent a server getting EOF every time the number of clients drops to 0, server may open well-known FIFO read/write, even though it never writes anything there.
- May be used with *select(2)*.
- Problems:
  - Hard for server to tell when client goes away.
  - Messages that are too big may be broken up, leading to interspersed requests.

(run `demos/dn_fifo`)

XSI IPC

- XSI: X/Open System Interface
  - Nothing to do with X11.
- Derived from System V IPC.
- Goal was to produce more flexible IPC than pipes and FIFOs.
- Three paradigms:
  - sending messages
  - sharing memory
  - semaphores

IPC Keys

- IPC based on "identifiers"
  - arbitrary integers "handles" created by the kernel
  - kind of like system-wide file descriptors
  - but you need to start with a "key" first.
- key (`key_t`, usually a `long int`) is passed to *msgget()*, *shmget()*, or *semget()*, all of which return an identifier.
  - A key of `IPC_PRIVATE` returns a private identifier for a new mechanism.
- but this must somehow be communicated among all participants, so ...

Where Does the Key Come From?

- A predetermined key can be stored in a shared header or mutually agreed-upon file but the key could already be assigned, so the **get()* calls will fail.
- Alternative: *ftok(3)*
- pathname/project ID $\rightarrow$ `ftok()`
- All programs that agree on a given path name and a project ID will return the same key.
  - Only the lower 8 bits of the project ID count.

Mapping the Key to an IPC Identifier

- As mentioned above, an identifier is like a persistent file descriptor that is meaningful to the whole system.
- `IPC_CREAT` bit needed to create the identifier in a **get()* function
  - but should only be called by one participant
  - the server, maybe?

Permission Structure

- passed to the **get()* functions
- look like the usual 9-bit file permission bitmask, except
  - that they don't describe files
  - the search/execute bit is not currently used by Linux

Configuration Limits

- Be aware of these.
- Set by kernel configuration.
- On Linux:

```
$ ipcs -l
```

To XSI IPC or Not to XSI IPC?

Advantages:

- reliable
- flow controlled
- record oriented
- can be processed in nonsequential order

Disadvantages:

- No reference counting – messages remain in system until read or deleted.
- IPC structures don't exist in filesystem.
- Much functionality already in the filesystem had to be duplicated.
- Identifiers aren't exactly file descriptors, so there's no multiplexed I/O.

Message Queues I

- record oriented
- Messages have
  - a "message type" (`msgbuf.mtype`)
  - a length (`n`)
  - a series of `n` bytes.
- *msgget(2)*
  - to establish or connect to a message queue
- *msgsnd(2)*
  - to send a message
  - note return: `ssize_t` (signed size) vs. `size_t`

Message Queues II

- *msgrcv(2)*
  - to receive a message
  - the ("`typ`") argument lets us screen messages:
    - `typ == 0`
      first message on queue
    - `typ > 0`
      first message of message type `typ`
    - `typ < 0`
      first message with message type $\leq$ |`typ`|
- *msgctl(2)*
  - does miscellaneous operations on message queues
    - deleting them
    - setting them for nonblocking I/O
  - kind of like *ioctl(2)* on devices or *fcntl(2)* on files.

Message Queues Examples

- unidirectional message passing:
  (run `demos/d`n`_ipc_chat`)
- bidirectional message passing:
  (run `demos/d`n`_ipc_oracle`)

Message Queues Reconsidered

- According to Stevens & Rago, message queues for “normal” messages don’t offer much advantages over local `AF_UNIX` (next unit) sockets.
- They’re still in use, though.
- Don’t use them for new stuff.

Semaphores I

- slightly more flexible than a mutex
- (originally: visual signalling device)
- Simple ones start at one and become zero when resource is in use. (i.e. they're mutexes)
- XSI IPC semaphores
  - start at a positive integer
  - resource is locked when the count reaches zero
  - somewhat more useful than mutexes, this allows you to restrict the number of users of a resource to something other than one
- *semget(2)*
  gets one (or more) semaphores
- *semctl(2)*
  does miscellaneous operations (such as deleting semaphores)

Semaphores II

- *semop(2)*
  - (look at `struct sembuf`)
  - the "thermometer":
    - `sem_op > 0`
      releasing resources by the process
    - `sem_op < 0`
      obtaining resources by the process
- S & R on record locking vs. semaphores:
  Record locking is slower, but easier.

Shared Memory

Maps the same area of memory into two or more processes.

- *shmget(2)*
- *shmctl(2)*
- *shmat(2)*
  - attaches shared memory to address space
  - virtual addresses may differ in two clients
- *shmdt(2)*

  detaches (like an unlink) shared memory from address space
- Access to shared memory is often controlled by semaphores.
- Where does shared memory fit into a virtual address space?

(run `demos/dn_shm`)

Client-Server Properties

- fork-exec
- client forks and execs server
  - bidirectional pipes can be used
  - server can be SetUID, looking at clients real UID (which it inherits) to verify permission beyond filesystem
- server can only send data, not – for instance – a file descriptor, back to a parent

Communication with Daemons

- can't use pipes for this
- FIFOs possible, but message queues better.
- single queue per daemon
- multiple queues, one per client (but no *select()* call available.)
- can use memory segments with semaphores as an alternative to message queues