[10] COMMUNICATION

OUTLINE

- Communication
  - Requirements
  - Inter-Thread Communication
  - Inter-Host Communication
  - Inter-Process Communication
- Inter-Process Communication
  - Concept
  - `fork(2)`, `wait(2)`
  - Signals
  - Pipes
  - Named Pipes / FIFOs
  - Shared Memory Segments
  - Files
  - Unix Domain Sockets

COMMUNICATION

- **Communication**
    - **Requirements**
    - **Inter-Thread Communication**
    - **Inter-Host Communication**
    - **Inter-Process Communication**
- Inter-Process Communication

REQUIREMENTS

For meaningful communication to take place, two or more parties have to exchange information according to a **protocol**:

- Data transferred must be in a commonly-understood format (**syntax**)
- Data transferred must have mutually-agreed meaning (**semantics**)
- Data must be transferred according to mutually understood rules (**synchronisation**)

In computer communications, the parties in question come in a range of forms, typically:

- Threads
- Processes
- Hosts

Ignore problems of discovery, identification, errors, etc. for now

INTER-THREAD COMMUNICATION

It is a common requirement for two running threads to need to communicate

- E.g., to coordinate around access to a shared variable

If coordination is not implemented, then all sorts of problems can occur. Range of mechanisms to manage this:

- Mutexes
- Semaphores
- Monitors
- Lock-Free Data Structures
- ...

Not discussed here!

- You'll get into the details next year in **Concurrent and Distributed Systems**
- (Particularly the first half, on *Concurrency*)

INTER-HOST COMMUNICATION

Passing data between different hosts:

- Traditionally different physical hosts
- Nowadays often virtual hosts

Key distinction is that there is now no shared memory, so some form of transmission medium must be used — **networking**

Also not discussed here!

- In some sense it is "harder" than IPC because real networks are inherently:
  - **Unreliable**: data can be lost
  - **Asynchronous**: even if data is not lost, no guarantees can be given about when it arrived
- You'll see a lot more of this next year in **Computer Networking**

INTER-PROCESS COMMUNICATION

In the context of this course, we are concerned with **Inter-Process Communication** (IPC)

- What it says on the tin — communication between processes on the same host
- Key point — it is possible to share memory between those processes

Given the protection boundaries imposed by the OS, by design, the OS must be involved in any communication between processes

- Otherwise it would be tantamount to allowing one process to write over another's address space
- We'll focus on POSIX mechanisms

INTER-PROCESS COMMUNICATION

- Communication
- **Inter-Process Communication**
    - **Concept**
    - **`fork(2)`,`wait(2)`**
    - **Signals**
    - **Pipes**
    - **Named Pipes / FIFOs**
    - **Shared Memory Segments**
    - **Files**
    - **Unix Domain Sockets**

CONCEPT

For IPC to be a thing, first you need multiple processes

- Initially created by running processes from a shell
- Subsequently may be created by those processes, ad infinitum
- (...until your machine dies from your fork bomb...)

Basic process mechanisms: `fork(2)` followed by `execve(2)` and/or `wait(2)`

Will look at that plus several other common POSIX mechanisms

`FORK(2)`,`WAIT(2)`

Simply put, `fork(2)` allows a process to clone itself:

- **Parent** process creates **child** process
- Child receives copy-on-write (COW) snapshot of parent's address space

Parent typically then either:

- Detaches from child — hands responsibility back to `init` process
- Waits for child — calling `wait(2)`, parent blocks until child exits

SIGNALS

Simple asynchronous notifications on another process

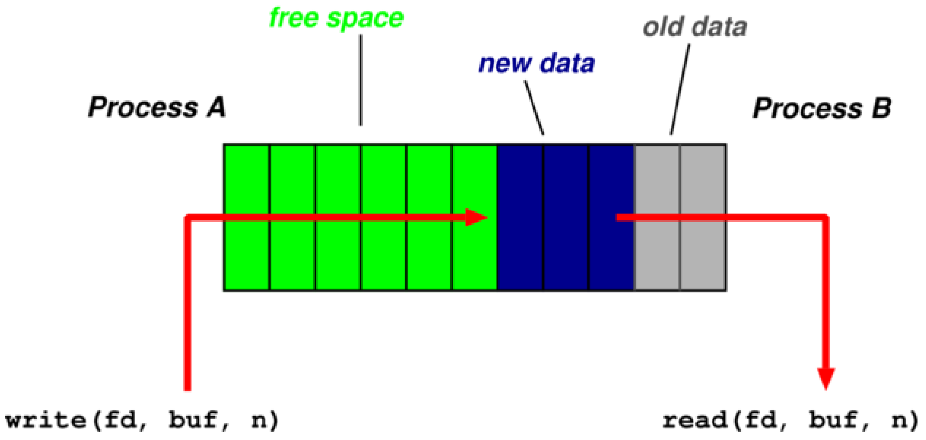
- A range of signals (28 at my last count), defined as numbers
- Mapped to standard `#defines`, a few of which have standard mappings to numbers

Among the more common ones:

- `SIGHUP`: hangup the terminal (1)
- `SIGINT`: terminal interrupt (2)
- `SIGKILL`: terminate the process [cannot be caught or ignored] (9)
- `SIGTERM`: terminate process (15)
- `SIGSEGV`: segmentation fault — process made an invalid memory reference
- `SIGUSR1/2`: two user signals [system defined numbers]

Use `sigaction(2)` to specify what function the signalled process should invoke on receipt of a given signal

PIPES



Simplest form of IPC: `pipe(2)` returns a pair of **file descriptors**

- (`fd[0]`, `fd[1]`) are the (read, write) fds

Coupled with `fork(2)`, can now communicate between processes:

- Invoke `pipe(2)` to get read/write fds
- `fork(2)` to create child process
- Parent and child then both have read/write fds available, and can communicate

NAMED PIPES / FIFOS

The same as `pipe(2)` — except that it has a name, and isn't just an array of two fds

- This means that the two parties can coordinate without needing to be in a parent/child relationship
- All they need is to share the (path)name of the FIFO

Then simply treat as a file:

- `open(2)`
- `read(2)`
- `write(2)`

`open(2)` will block by default, until some other process opens the FIFO for reading

- Can set non-blocking via `O_NDELAY`

SHARED MEMORY SEGMENTS

What it says on the tin — obtain a segment of memory that is shared between two (or more) processes

- `shmget(2)` to get a segment
- `shmat(2)` to attach to it

Then read and write simply via pointers — need to impose concurrency control to avoid collisions though

Finally:

- `shmdt(2)` to detach
- `shmctl(2)` to destroy once you know no-one still using it

FILES

Locking can be mandatory (enforced) or advisory (cooperative)

- Advisory is more widely available
- `fcntl(2)` sets, tests and clears the lock status
- Processes can then coordinate over access to files
- `read(2)`, `write(2)`, `seek(2)` to interact and navigate

Memory Mapped Files present a simpler — and often more efficient — API

- `mmap(2)` "maps" a file into memory so you interact with it via a pointer
- Still need to lock or use some other concurrency control mechanism

UNIX DOMAIN SOCKETS

Sockets are commonly used in network programming — but there is (effectively) a shared memory version for use between local processes, having the same API:

- `socket(2)` creates a socket, using `AF_UNIX`
- `bind(2)` attaches the socket to a file
- The interact as with any socket
  - `accept(2)`, `listen(2)`, `recv(2)`, `send(2)`
  - `sendto(2)`, `recvfrom(2)`

Finally, `socketpair(2)` uses sockets to create a full-duplex pipe

- Can read/write from both ends

SUMMARY

- Communication
  - Requirements
  - Inter-Thread Communication
  - Inter-Host Communication
  - Inter-Process Communication
- Inter-Process Communication
  - Concept
  - `fork(2),wait(2)`
  - Signals
  - Pipes
  - Named Pipes / FIFOs
  - Shared Memory Segments
  - Files
  - Unix Domain Sockets