IPC and Sockets:

* Key Idea: Communication between processes and across the world looks like File I/O
* Introduce Pipes and Sockets
* Introduce TCP/IP Connection set up for Web Server

pid\_t fork() 🡪 Copy current process

* State of original process duplicated in parent and child
* Address Space (Memory), FD, etc…
* Return value from fork():
  + When > 0: Running in original Parent process
  + When = 0: Running in new Child process
  + When < 0: Error!

Uniformity – Everything is a File!

* File operations, device I/O, and IPC through open, read/write, close
* Allows for simple composition of programs (find | grep | wc… )
* Open before use:
  + Provides opportunity for access control and arbitration
  + Sets up underlying machinery (ie, data structures)
* Byte-oriented
  + Even if blocks are transferred
  + (agnostic, doesn’t matter what data looks like)
* Kernel buffered reads
  + To give byte-oriented access (streaming and block devices looks the same)
* Kernel buffered writes
* A picture containing text, screenshot, diagram, font

  Description automatically generatedExplicit close

Putting it together: Web server (single process)

* Steps 3, 8 are kernel buffered reads
* Step 11 is kernel buffered write

High Level File API – Streams

* Operates on “streams”, unformatted sequence of bytes
* fopen returns FILE\*, data structure:
  + Error reported by returning nullptr
  + Pointer used in subsequent operations on stream
  + Data buffered in user space

Low-Level File I/O (RAW system-call interface):

* A screenshot of a computer code

  Description automatically generated with low confidenceInteger return from open() is a file descriptor
  + File descriptor used in subsequent operations on the file
* Streams (opened with fopen()) have file descriptor inside of them!
  + Retrievable with fileno(FILE\* stream) 🡪 internal file descriptor

Representation of Process (inside kernel):

A diagram of a file

Description automatically generated with low confidence

Can change position with seek operations

Communication between processes:

* What if processes wish to communicate with one another?
  + Why? Shared Task, Cooperative Venture w/ Security Implications
* Process Abstraction designed to Discourage IPC
* So, must do something special (and agreed upon by both processes)
  + Must “Punch Hole” in security
* Called “Interprocess Communication” 🡺 “IPC”

Recall: Processes protected from each other (translation mapping)

* Producer (writer) and consumer (reader) may be distinct processes
  + Potentially separated in time
  + How to allow selective communication?
* A picture containing text, screenshot, font, logo

  Description automatically generatedSimple option: Use a file!
  + Parents and children share file descriptions
  + Very slow, requires persisting information to disk just to allow processes to communicate
  + On the right track… processes are communicating with file descriptors
* Can use shared memory through translation mapping (both processes have shared piece of virtual memory that maps onto same physical address)

Suppose we ask Kernel to help?

* Consider an in-memory queue?
* Accessed via system calls (for security reasons)
* Data written by A is held in memory until B reads it
  + Same interface as we use for files
  + Internally more efficient, since nothing goes to disk
  + How to set it up?
    - What if A generates data faster than B can consume it?
    - What if B generates data faster than A can produce it?
    - A green and pink rectangle with black text

      Description automatically generated with low confidenceUse WAIT

Example: POSIX/Unix PIPE

* Memory buffer is finite
  + If producer(A) tries to write when buffer is full, it blocks
  + If consumer(B) tries to read when buffer empty, it blocks
* int pipe(int fileds[2]);
  + Allocates two new file descriptors in the process
  + Writes to fileds[1] read from fileds[0]
  + A green and black logo

    Description automatically generated with low confidenceImplemented as fixed-size queue in kernel memory (use syscalls to read, write)

A screenshot of a computer code

Description automatically generated with low confidence

Kernel knows when A writes and B reads (when A writes whether B needs to be woken and vice versa)

A picture containing text, screenshot, font

Description automatically generatedHow to have two processes communicate using pipe:

How to create channels from Child 🡺 Parent and Parent 🡺 Child

A picture containing text, diagram, screenshot, line

Description automatically generatedA picture containing text, diagram, screenshot, line

Description automatically generated

A picture containing text, diagram, screenshot, line

Description automatically generated

When do we get EOF on a pipe?

* After last “write” is closed, pipe is effectively closed
  + Reads return only “EOF”
* After last “read” descriptor is closed, write generates SIGPIPE signals
  + If process ignores, then the write fails with an “EPIPE” error

Once we have communication, we need a protocol

* A protocol is an agreement on how to communicate
* Includes:
  + Syntax: how a communication is specified and structured
    - Format, order messages are sent and received
  + Semantics: what a communication means
    - Actions taken when transmitting, receiving, or when a time expires
* Described formally by a state machine
  + Often represented as a message transaction diagram
* In fact, across network may need a way to translate between different representations for numbers, strings, etc
  + Such translation typically part of Remote Procedure Call (RPC) facility
  + Don’t worry about this now, but it is clearly part of the protocal

Examples of Protocols:

* Telephone:
  + Pick up
  + Listen for dial tone
  + Dial
  + Should hear ringing…
  + Callee: \*ring\* Hello?
  + Caller: Hi it’s John
  + Caller: \*blah blah blah\*
  + Callee: \*blah blah blah8
  + Caller: Bye
  + Callee: Bye
  + Hang Up

Client Server Protocols: Cross-Network IPC

* Many clients accessing a common server
* File servers, www, FTP, databases
* Each unique communication has IP address, Port, communication protocol on both sides

Client-Server Communication

* Client is “sometimes on”
  + Sends server requests for services when interested
  + Doesn’t communicate directly with other clients
  + Needs to know server’s address
* Server is “always on”
  + Services requests for many clients
  + Doesn’t initiate contact with clients
  + Needs a fixed, well-known address

What is a Network connection?

* Bidirectional stream of bytes between two processes on possibly different machines
* Abstractly a connection between endpoints A, B consists of :
  + A queue (bounded buffer) for data sent from A to B
  + A queue (bounded buffer) for data sent from B to A

Socket Abstraction:

* Key Idea: communication across the world looks like File I/O
* Socket: Endpoint for communication
  + A picture containing text, screenshot, font, diagram

    Description automatically generatedQueues to temporarily hold results
* Connection: Two sockets connected over the network 🡺 IPC over network

Socket: An abstraction for one endpoint of a network connection

* Another mechanism for IPC
* Most OS provide this, even if they don’t copy UNIX I/O
* Standardized by POSIX
* First introduced by 4.2 Berkley Standard Distribution Unix (BSD)
* Same abstraction over any kind of network
  + Local (within same machine), Internet (TCP/IP, UDP/IP), etc
* Looks just like a file with file descriptor
  + Corresponds to a network connection (two queues)
  + Writes adds to output queue
  + Read removes from input queue
  + Some operations (ie, lseek) don’t work
* How can we use sockets to support real applications?
  + A bidirectional byte stream isn’t useful on its own
  + May need messaging facility to partition stream into chunks
  + May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

Simple Example: Echo server



A screen shot of a computer

Description automatically generated with medium confidence

A picture containing text, diagram, screenshot, line

Description automatically generated

What assumptions are we making?

* Reliable
  + Write to a file 🡺 Read it back
  + Write a TCP socket 🡺 Read from the other side
  + Like pipes
* In order (sequential stream)
  + Write X then write Y 🡺 Read gets X then reads gets Y
* When ready?
  + File read gets whatever is there at the time
  + Assumes writing already took place
  + Blocks if nothing has arrived yet
  + Like pipes!

Socket Creation:

* File systems provide a collection of permanent objects in a structured name space
* Files exist independently of processes
* Processes open, read/write/close them
* Easy to name what file to open()

Pipes: **One-way** communication between processes on same (physical machine

* Single queue
* Created transiently by a call to pipe()
* Passed from parent to children (descriptors inherited from parent process)
* (If both try to write, data will be interleaved) 🡺 Can close
* For bi-direction create two separate pipes

Sockets: Two-way communication between processes on same or different machine

* Two queues (one in each direction)
* Processes can be on separate machines? No common ancestor
* How do we *name* the objects we are opening?
* How do these completely independent programs know that the other wants to “talk” to them?

Naming System -> Namespaces for Communication over IP

* Hostname
  + [www.eecs.berkley.edu](http://www.eecs.berkley.edu)
* IP address
  + 128.32.244.172 (IPv4, 32-bit Integer)
  + 2607:f140:0:81::f (IPv6, 128-bit Integer)
* Port Number
  + (many programs on a single machine have same IP address, need to be able to distinguish between them)
  + 0-1023 “well-known” or “system” ports
  + 1024 – 49151 are “registered” ports
  + 49152-65535 are ”dynamic” or “private”

Connection Setup over TCP/IP

* Special kind of socket: **server socket**
  + Has fd
  + Can’t read or write
* Two operations:
  + 1) listen(): start allowing clients to connect
  + 2) accept(): create a *new socket* for a *particular* client

Ports are what make connection unique:

5-Tuple identify each connection:

* Source IP Address
* Destination IP Address
* A picture containing text, font, screenshot, diagram

  Description automatically generatedSource Port Number
* Destination Port Number
* Protocol

Often Client Port “randomly” assigned (done by OS during client socket setup)

Server Port often “well known”

* 80 (web), 443 (secure web), 25 (sendmail)

Server: Client:

* Create Server Socket - Create Client Socket
* Bind it to an Address - Connected to server (host: port)
* Listen for Connction - Connection Socket
* Accept Syscall (Loop) - Write request
* Connected Socket
* Read Request

A picture containing text, font, screenshot, line

Description automatically generated

A picture containing text, font, screenshot, algebra

Description automatically generatedA picture containing text, screenshot, font

Description automatically generatedClient Server (Server Socket is for listening, Connection Socket is for reading/writing, so child closes the listening server socket)

This server only makes one connection at a time

A picture containing text, screenshot, font, line

Description automatically generated

A concurrent server can handle and service a new connection before the previous client disconnects

A picture containing text, screenshot, font, line

Description automatically generated

Can use thread pools to bound the number of threads (create fixed number of threads, incoming connections can be placed on queue. When thread frees up, it can dequeue connection and serve it)

IPC

* Communication facility between protected environments (processes)

Pipes are an abstraction of a single queue

* One one write-only, one end read-only
* Used for communication between multiple processes on a single machine
* File descriptors obtained via inheritance

Sockets are an abstraction of two queues

* Can or write to either end
* Used for communication between multiple processes on different machines
* File descriptors obtained via socket/bind/connect/listen/accept
* Inheritance of file descriptors on fork() facilitates handling each connection in a separate process