Network connections are handled using sockets. There are several different kinds of sockets.

A socket is a means of connecting two programs, often living on two separate computers. It’s kinda like a pipe, except that it is set up after the programs are already running. The process for using a socket is a bit different depending on whether you want to connect to another socket or listen for incoming connections

To connect to another socket:

1. Create a socket with socket
2. Connect the socket to the server with connect
3. Read and write data from the socket

To listen for incoming connections:

1. Create a socket with socket
2. Bind it to a network address with bind
3. Listen for connections with listen
4. Accept incoming connections with accept, yielding a connected socket

5. Read and write data from accepted connection

6. Repeat for subsequent connections

We want non-blocking input

• read returns right away, even if there is no data

• no data is an error (-1 return code), with errno of EAGAIN

• write might not write all the data

How do we get non-blocking input?

• The O\_NONBLOCK flag to open

• But we didn’t open standard input, or our socket

• So we use fcntl:

intfcntl(int fd,int cmd, ...);

So we can set the non-blocking flag:

fcntl(0, F\_SETFL, O\_NONBLOCK);

Now we can read from both sides!

select. It watches multiple file descriptors and returns when there is data

available on at leastone.

Takes:

1. Very strange nfds argument. Not the # of file descriptors, but rather the

Largest file descriptor (plus one).

2. Read FD set

3. Write FD set

4. Exception FD set

5. Timeout

Returns:

• Number of FDs ready (0 for timeout) (actual number, not this highest-FD stuff)

• -1 on error

The **fopen**() function opens the file whose name is the string pointed to by *path* and associates a stream with it.

Fdopen on the stream

• fclose when we are done, this automatically closes the FD (socket)

• can’t use same stream for reading and writing

FILE \*stream = fdopen(sock, "r+"); //sock is an int

Processes

• create a process with fork()

• fork()returns twice: once in parent process, once in child.

• new process is a copy of parent process – keeps file descriptors open – almost always: only read/write from one process

a **port** is a software construct serving as a communications endpoint in a computer's host operating system. A port is always associated with an [IP address](http://en.wikipedia.org/wiki/IP_address) of a host and the protocol type of the communication. It completes the destination or origination address of a communications session. A port is identified for each address and protocol by a 16-bit number, commonly known as the **port number**.

The connection and bind operations will require a network

address

Addresses come in many forms:

• UNIX domain socket paths (/var/run/pgsql.sock)

•IP addresses (192.168.3.49)

•IPv6 addresses

•Host names (e.g. google.com)

TCP (Transmission Control Protocol) is the most commonly used protocol on the Internet. The reason for this is because TCP offers error correction. When the TCP protocol is used there is a "guaranteed delivery." This is due largely in part to a method called "flow control." Flow control determines when data needs to be re-sent, and stops the flow of data until previous packets are successfully transferred. This works because if a packet of data is sent, a collision may occur. When this happens, the client re-requests the packet from the server until the whole packet is complete and is identical to its original.   
UDP (User Datagram Protocol) is anther commonly used protocol on the Internet. However, UDP is never used to send important data such as webpages, database information, etc; UDP is commonly used for streaming audio and video. Streaming media such as Windows Media audio files (.WMA) , Real Player (.RM), and others use UDP because it offers speed! The reason UDP is faster than TCP is because there is no form of flow control or error correction. The data sent over the Internet is affected by collisions, and errors will be present. Remember that UDP is only concerned with speed. This is the main reason why streaming media is not high quality.

Signals

Now, we have a little issue. Let’s look at ps, and see the zombies. Once a process exits, it is kept around until its parent asks for its status. This allows us to get the exit code.We can do this with wait or waitpid.

How do we know when we need to do that?

• just know, and call wait from time to time (e.g. in select timeout handler).

• handle the SIGCHLD signal Signals are mechanisms for the system to notify processes of events. There are several commonly-used signals:

• SIGTERM, indicating that a process should stop

• SIGKILL, kills a program (cannot be caught)

•SIGCHLD, something happened to a child

•SIGINT, the program should be interrupted (Ctrl-C sends this)

•SIGUSR1,SIGUSR2, used for user-defined programs

•SIGHUP, the output terminal hung up (for old dial-up systems). This signal is often used to tell adaemon to reload its configuration files. We can send a signal with Kill (the command or the C function). We can handle a signal with a signal handler . This is a little function that is run when a signal is caught. We can register a signal handler with the

Sigaction function. The signal action struct has a few things:

•the handler

•the flags; we can ignore the signal, set default status, set a handler, etc.

•a signal set to block (see sigsetops for how to work with this) Let’s use this to make

Fingerd reap its children. Let’s also make fingerd terminate cleanly. For this we need a handler for SIGINT and SIGTERM, and a volatile variable. Volatile is needed whenever we have a variable (such as a shutdown flag) that will be changed from a signal handler.

Process Reparenting

If a process terminates, it is a zombie until its exit status is collected. The parent process is responsible for this. If a process terminates, all its child processes are reparented to process 1

(the init process). The init process ‘reaps’ (waits for) all of its children, so the process will be cleaned up if its parent is terminated. Note: the process is not reparented to the grandparent or anything like that. Signal Handling on a File Descriptor On Linux, the process can also receive its signals on a file descriptor. This does not work on Mac, BSD, etc. (some of them may provide a similar function, but the API is different). We can do this with signalfd, and then integrate it into our select loop. This is often easier than using

signal handlers (and is better-behaved in edge cases) but is non-portable.

Process Management

Suppose we want to run another program. We use the exec call. It replaces the current process with another. So to run another program: fork followed by exec in the child.

Between fork and exec we can do interesting things:

•modify file descriptors

•set environment variables

•clean up environment (close file descriptors that aren’t needed, for example)

Useful:

O\_CLOEXEC, makes sure the FD will be closed on exec.

Let’s write a new program:

inetd. Listens for network connections, passes them off to external processes

such that:

•standard output will be socket

•standard input will be socket

•standard error will be parent’s standard error (so error messages show up)

•environment variables contain remote information (host and port)

Key pieces:

•dup2 rearranges file descriptors

•setenv sets environment variables

•we will also create a process group using setpgrp, so we can kill everything at once.

Pipes

We can fork, launch processes, etc. How can we talk between them?

pipe

to the rescue.

pipe

creates a pair of file descriptors wired to each other.

Let’s write a string-prefixing output processor

•runs a command

•prefixes each line of standard output with ‘out’

Review of communication

•file descriptors

•sockets

•processes

•signals

How else can processes communicate?

Shared Memory and Semaphores

•semaphores

allow different processes to wait on things

•shared memory

allows different processes to access a common memory area.

File System Operations

We’ve done basic file system operations:

•open

There’s a lot more:

•mkdir

(takes initial permissions)

•open

with different options

–O\_CREAT

–O\_EXCL

–O\_TRUNC

•unlink

•link

•rename

There’s super-useful behavior of

link

and

rename

:

•link is atomic, and fails if the target file already exists

•rename is atomic, and overwrites the target file

Useful for implementing locking protocols, concurrent access protocols.

•lock a directory: create a unique file name, then

link

it to a well-known name. On success, you have

the lock.

–needs care on network file systems

–also check the link count (needed on NFS... - may fail but really succeed)

•concurrent key-value store:

–unique file name + link + unlink, to try-replace

–unique file name + rename, to replace unconditionally These tricks are hard, if not impossible, on Windows...

File Locking

You can lock a file, so other programs cannot access it.

• advisory locks, doesn’t actually prohibit writing but allows for coordination.

• two interfaces:

-flock, locks whole file

–lockf, locks part of a file

–third:fcntl locking

–coordinating programs must use same interface

Memory-Mapped File I/O

In addition to read and write, you can map a chunk of a file into memory and read it like bytes.

•the mmap call

•munmap unmaps

•msync writes the data

•can protect data:

mprotect

