Environment

- setting environment for program execution

MY_VAR='myVar'
export MY_VAR

- There are positional parameters passed in through argc and argv.
- There are also environment variables and special parameters that you can pass in by name instead of by position.
- getenv.c (listed in the class code part of content) uses both of these. You pass in the name of the environment variable through argv, and it prints it out with getenv().
- > HELLO=WORLD ./getenv HELLO
  value for HELLO is WORLD
- > ./getenv HELLO
  No value for HELLO
- ./getenv PATH
  value for PATH is
  a/bunch/of/different:paths/separted/by/colons:this/will/vary/from/computer/to:computer/because:PATH/is/an/environment/variable/already/in/the/shell
- In these examples, I'm pretty sure HELLO would be called a positional parameter, HELLO=WORLD is a special parameter, and PATH is an environment variable.

- retrieving environment (e.g., from within C)

https://www.systutorials.com/241139/how-to-set-and-get-an-environment-variable-in-c-on-linux/
https://www.tecmint.com/set-unset-environment-variables-in-linux/

File descriptors

- stdin, stdout, stderr (0, 1, and 2, respectively)
- redirection / duplication within a program
- pipe concepts

A pipeline is a sequence of one or more commands separated by one of the control operators | or |&. The format for a pipeline is:

[time [-p]] [!] command [[|â",|&] command2 ...]

The standard output of command is connected via a pipe to the standard input of command2. This connection is performed before any redirections specified by the command

- dup2() concepts

int dup2(int oldfd, int newfd);

Overwrites the oldfd (file descriptor) with the newfd.

Shell usage

- file descriptor redirection / duplication
- shell pipelines
- jobs / job control
  - background / foreground / suspend

- basic usage of echo, cat, grep

Processes

Definition: A process is an instance of a running program.

One of the most profound ideas in computer science

Not the same as “program” or “processor”

- fork

int fork(void)

Returns 0 to the child process, child’s PID to parent process

Child is almost identical to parent:

Child gets an identical (but separate) copy of the parent’s virtual address space.

Child gets identical copies of the parent’s open file descriptors

Child has a different PID than the parent

fork is interesting (and often confusing) because it is called once but returns twice

- concurrency, concurrent execution

Two processes run concurrently (are concurrent) if their flows overlap in time

- synchronization with wait() / waitpid()
- exec
- orphaned processes ("daemons")
- zombies/reaping
- process groups, setpgid(), how a shell handles groups with signals, etc.

Stop and kill signals are sent to groups. The shell needs to split off children into a different group so it doesn’t get killed/stopped when it passes on the kill/stop signals

- retrieving status from exited child processes with macros: WIFEXITED/WIFSTOPPED/WIFSIGNALED/etc.

Exceptions

- asynchronous vs. synchronous

Caused by events external to the processor

Indicated by setting the processor’s interrupt pin

Handler returns to “next” instruction

Examples:

Timer interrupt

Every few ms, an external timer chip triggers an interrupt

Used by the kernel to take back control from user programs

I/O interrupt from external device

Hitting Ctrl-C at the keyboard

Arrival of a packet from a network

Arrival of data from a disk

Synchronous: Caused by events that occur as a result of executing an instruction:

- traps, faults, aborts

Traps

Intentional

Examples: system calls, breakpoint traps, special instructions

Returns control to “next” instruction

Faults

Unintentional but possibly recoverable

Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions

Either re-executes faulting (“current”) instruction or aborts

Aborts

Unintentional and unrecoverable

Examples: illegal instruction, parity error, machine check

Aborts current program

- system calls

Each x86-64 system call has a unique ID number

Example:

User calls: open(filename, options)

Calls __open function, which invokes system call instruction syscall

Signals

- Signal blocking

There are 2 bit arrays (Received, and blocked) that handle signals

- Sending vs. receiving a signal

Use kill command to send signals

When a signal is received the appropriate bit is set to 1

- SIGCHLD, SIGINT, SIGTSTP, SIGCONT, SIGTERM, SIGKILL

SIGCHLD: A child has terminated

SIGINT: A ‘would you please terminate’ signal (ctrl-c keyboard interrupt, can be caught)

SIGTSTP: Program is to stop running

SIGCONT: Program is set to running

SIGTERM: ‘I really really want you to terminate’ signal (can be caught/ignored)

SIGKILL: You dead son (CANNOT BE CAUGHT/IGNORED)

- Default actions, overriding with handlers or with SIG_DFL (default) or SIG_IGN (ignore).
- Signal blocking with sigprocmask()

Sets the Blocked bitvector to a given bitvector. 1’s are blocked, 0’s are not blocked

Virtual Memory

- Virtual addressing and pages
- Page table / Page table entries (PTE)
- Identifying VPN and VPO given a VA

- Identifying PPN and PPO given a VA and a PTE
- Identifying a PA given a PPN and PPO
- TLB - access using TLB index, TLB tag
- Find a byte of data given a virtual address

Network protocols

- client/server model
  - Client and server. 2 separate things talking back and forth. Two boxes with arrows going between.
- IP content: source IP address; destination IP address
  - Client has an IP address, and server has a separate IP address that stuff gets sent to.
  - (protocol)http: (URL)//www.byu.edu (port):80 (URI) /index.html
- TCP/UDP: source port; destination port - What is meant by this?
  - The client has a source port the information/sockets come from. There is also a destination port that the information/sockets get sent to.
- Notion of well-known ports (you should know the well-known ports for HTTP and DNS)
  - **Said to only worry about HTTP 80 and DNS 53**
  - **SSH** - 22
  - **HTTP -** 80
  - **HTTPS -** 443
  - **DNS -** 53
  - **SMTP -** 25 (simple mail transfer protocol)
- TCP vs. UDP characteristics, TCP three-way handshake
  - UDP
    - No “Frills” - refers to bare-bones delivery
    - Best effort delivery
    - Connectionless
    - Uses packets
  - TCP
    - Connection-oriented
    - Reliable delivery
    - Byte stream
  - Three-Way handshake
    - Method to guarantee delivery of packets.
    - **Step 1 (SYN):** In the first step, the client wants to establish a connection with the server, so it sends a segment with SYN(Synchronize Sequence Number) which informs server that client is likely to start communication and with what sequence number it starts segments with
    - **Step 2 (SYN + ACK):** Server responds to the client request with SYN-ACK signal bits set. Acknowledgement(ACK) signifies the response of segment

it received and SYN signifies with what sequence number it is likely to start the segments with

        - **Step 3 (ACK):** In the final part the client acknowledges the response of server and they both establish a reliable connection with which they will start the actual data transfer

- Unique identification of a "connection" between two hosts (i.e., four-tuple)0
    - Source Port
    - Destination Port
    - Source IP
    - Destination IP
- IPv4 address characteristics, length, presentation
    - 32 Bit decimal number
    - A common way to represent the ip address is to break your 32 bits into 4 bytes, write each byte in decimal, and then join with dots. For example:
        - Raw 32 bit ip address in hex: BB591FE2
        - Break into 4 bytes: BB 59 1F E2
        - Convert to decimal: 187 89 31 226
        - End result 187.89.31.226
    - Maybe converting hex to IP address
    - Common to put a colon and a port after the ip address. I.e. 187.89.31.226:80

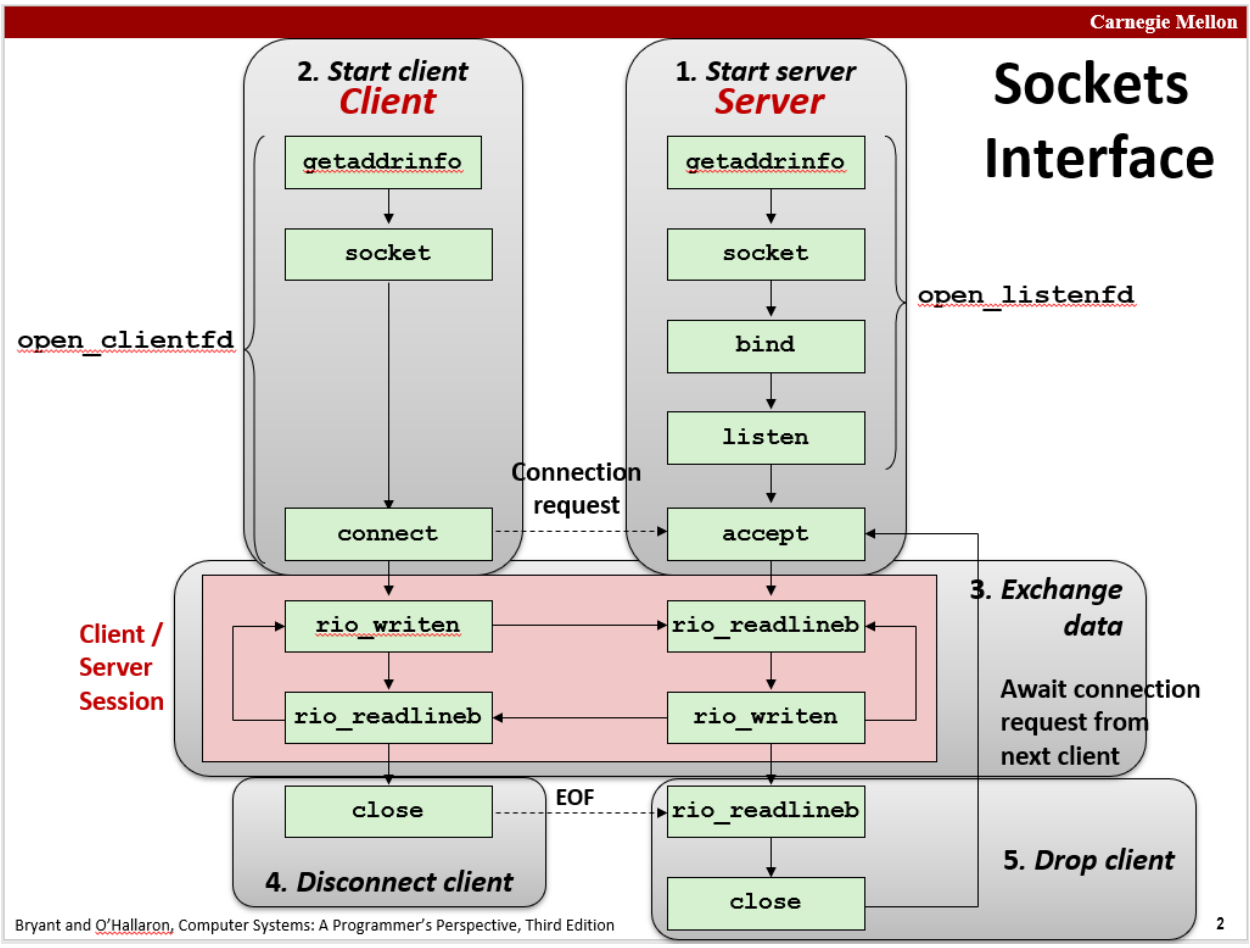
Sockets

- types: SOCK_DGRAM and SOCK_STREAM
    - SOCK_DGRAM - UDP
        - Man Page - “Provides datagrams, which are connectionless messages of a fixed maximum length. This type of socket is generally used for short messages, such as name server or time server since the order and reliability of message delivery is not guaranteed.”
    - SOCK_STREAM - TCP
        - Man Page - “communication protocols are designed to prevent the loss or duplication of data. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable period of time, the connection is broken. When this occurs, the **socket** subroutine indicates an error with a return value of -**1** and the **errno** global variable is set to **ETIMEDOUT**. If a process sends on a broken stream, a **SIGPIPE** signal is raised. Processes that cannot handle the signal

terminate. When out-of-band data arrives on a socket, a **SIGURG** signal is sent to the process group. “

- underlying transport protocol
  - Datagrams for UDP
  - Sockets for TCP

- socket functions:



  - socket(), bind(), connect(), listen(), accept(), close()
    - Socket(ipv4, sock_dgram) - initializes a socket
    - connect(s, struct, IP, port) - creates a connection, only used for TCP sockets ← is this a correct statement? <— Sort of. You can call connect for UDP but it won’t result in a “connection” - instead it just does some surface level error checking and sets the default destination address to whatever is specified. I don’t believe it is a necessary call for UDP but it can be useful. See https://www.geeksforgeeks.org/udp-client-server-using-connect-c-implementation/ and the relevant man pages. Note that in the Geeks For Geeks example code, null is passed in sendto() as the

destination address structure because the default was set in the call to connect().

- DR DECIO FROM SLACK - bind() - typically assigns a *local* IP address *and* port with a socket.
- **`int listen(int sockfd, int backlog);`**
  - marks the socket referred to by *sockfd* as a passive socket,that is, as a socket that will be used to accept incoming connection requests using accept(2).
- **int accept(int *sockfd*, struct sockaddr **addr*, socklen_t **addrlen*); - returns new client file descriptor**
  - The **accept**() system call is used with connection-based socket types (**SOCK_STREAM**, **SOCK_SEQPACKET**). It extracts the first connection request on the queue of pending connections for the listening socket, *sockfd*, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket *sockfd* is unaffected by this call. The argument *sockfd* is a socket that has been created with socket(2), bound to a local address with bind(2), and is listening for connections after a listen(2).
  - The argument *addr* is a pointer to a *sockaddr* structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned *addr* is determined by the socket's address family (see socket(2) and the respective protocol man pages). When *addr* is NULL, nothing is filled in; in this case, *addrlen* is not used, and should also be NULL. The *addrlen* argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by *addr*; on return it will contain the actual size of the peer address. The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call. If no pending connections are present on the queue, and the socket is not marked as non-blocking, **accept**() blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, **accept**() fails with the error **EAGAIN** or **EWOULDBLOCK**.

        - Close(fileDescriptor) - deallocates the file descriptor and destroys the socket. If it is in connection with a linger flag set then it will block until all of the data has been sent.
    - how each differs or is applicable (or not) for each socket type (i.e., SOCK_DGRAM and SOCK_STREAM)
        - Used by Both:
            - socket()
            - bind()
            - connect() - I don't think this is used by UDP see Kendall Response to Jason M in slack channel
            - close()
        - Only used by SOCK_STREAM / TCP:
            - listen()
            - accept()
- how each differs or is applicable (or not) to client or server
    - Applicable to both:
        - socket()
        - close()
    - Used only by client:
        - connect()
            - Underneath the covers, client gets a port. Different from bind() in that we don't care which port the client is using.
    - Used only by server:
        - bind()
            - Server needs to specify which port it receives data on. That's why it uses bind() instead of connect.
        - listen()
        - accept()
- kernel- vs. user-level functionality associated with each
- read()/write() operations on blocking sockets of both types (i.e., SOCK_DGRAM and SOCK_STREAM)
    - depending on what data is in the buffer, if any
        - For both UDP and TCP, it blocks and waits if there is no data in the buffer. It does not return zero.
        - If you are sent 100 bytes of data via UDP, but you only read 70, the last 30 bytes will be lost.
        - If you are sent 100 bytes of data via TCP, but you only read 70, the last 30 will hang around for the next read.

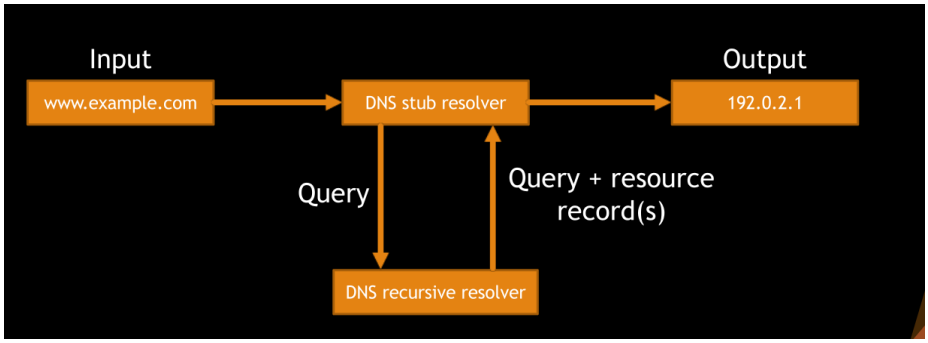
    - what happens if peer closes connection
        - Reads and writes will be stopped. You'll be sent an end of file type of signal.
- getaddrinfo() usage
    - getaddrinfo(const char * host, const char * service, const struct addrinfo *hints, struct addrinfo ** result)
        - host — string with domain name you're interested in.
        - service — Can either be service (i.e. http) or decimal port number.
        - hints — can set the following fields of addrinfo struct:
            - ai_family (AF_INET | AF_INET6) — ipv4 or ipv6
            - ai_socktype (SOCK_DGRAM | SOCK_STREAM) — udp or tcp
            - ai_protocol (we've usually set this to 0)
            - ai_flags (set this to AI_PASSIVE if a server)
            - Everything else in hints should be zero (use memset())
        - result — on successful return, will point to head of linked list of addrinfo structs
            - (*result) will be a addrinfo struct (the head)
            - *(result->ai_next) will be node after head.
            - result->ai_family, result->ai_socktype, and result->ai_protocol can be used as arguments to socket()
            - result->ai_addr and result->ai_addrlen can be used as arguments to bind() and connect()
            - If socket() fails, you can try the next thing in the list by using result = result->ai_next
    - Before you call getaddrinfo, you spend a few lines setting up `hints`.
    - After you call getaddrinfo, you spend lines calling socket() and bind()/connect() with the values you can access from `results`.
    - Be sure to free the list that `results` will point to with freeaddrinfo()

URL

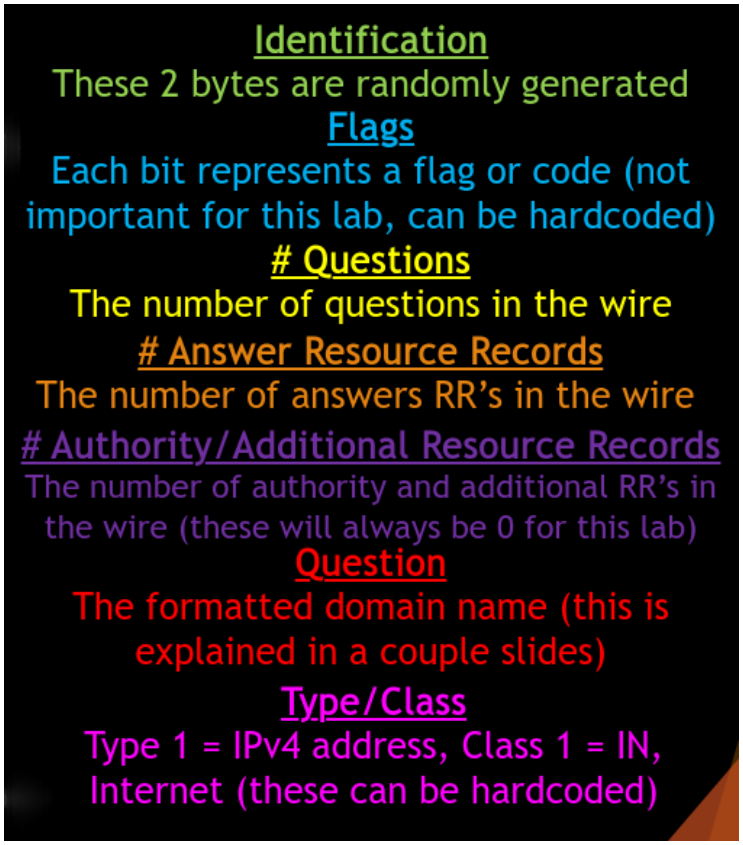
- parts breakdown: protocol, hostname, port, URI (including query string)
    - <Protocol> :// <Hostname> : <Port> / <URI>
        - Ex. https://www.example.com:8080/index.html

DNS

- Name resolution basics

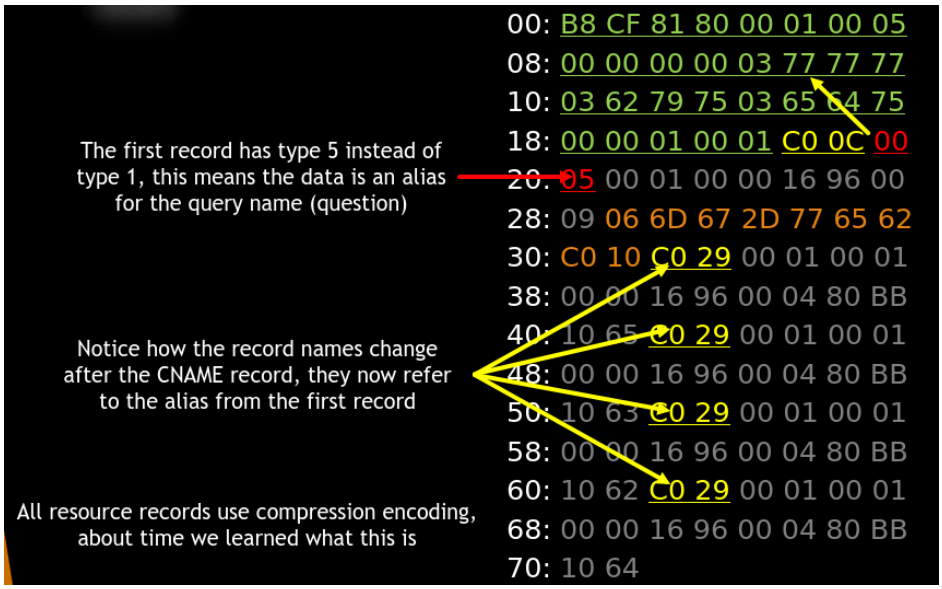

Input
www.example.com
DNS stub resolver
Output
192.0.2.1
Query
Query + resource record(s)
DNS recursive resolver

- 

- 

- Types: A, CNAME
  - A (Authoritative) - IpV4 address
  - CNAME - Alias

- Name encoding in DNS message
  - 

- Name compression in DNS messages (the concepts from the lab!)
  - 

  - 

- HTTP
- Request line:
  - <Method> <URL> <Version>
    - Ex. GET https://www.example.com:8080/home/index.htm HTTP/1.0
- Response line:
- Headers / end-of-headers signal
  - <Name> : <Data>
- Content-length header
  - Tells the recipient how long the request/response BODY is
- How a query string is formed
- GET vs. POST
  - Get request does NOT include a body

    - Get response DOES include a body
    - Post request DOES include a body
    - Post response DOES include a body
    - Get is asking for a resource, Post is asking to create a resource
- Request body
- Response body

CGI - Common Gateway Interface

- How CGI is implemented by a Web server (Web server perspective)
    - Run in the webserver, must follow the rules of the server not the web browser
- How a CGI program operates inside a Web server (CGI program perspective)
    - CGI program is fork/execve'd from the web server process.
    - Http request (without the first line) is on stdin
    - QUERY_STRING (stuff after question mark in URL) is an environment variable you can get with getenv()
    - Not sure about this, but I think the URI might be passed ~~as an environment variable as well~~.
        - G. 957-958 Tiny Web Server - lines 11-14
            - Reads the first line in the Http request to get method, uri, and version
        - G. 963 - CGI program uses the passed in cgiargs from the URI before setting QUERY_STRING to cgiargs, dup2 its stdout to the connectionfd and then execve'd
            - If (Fork() == 0) {
                - setenv("QUERY_STRING", cgiargs, 1);
                - Dup2(fd, STDOUT_FILENO);
                - Execve(filename, emptylist, environ);
            - }
    - The URI minus the query string is typically the path to the CGI program (This is what the server gives to execve()) - stuff before the "?"
- How a client passes data (both GET and POST data) to a Web server
    - As Params in the query string (both GET & POST) or in request body (POST)

Threads / Semaphores

- what is shared between threads, what is not
    - Definitely shared: File descriptors, global variables, static variables.
        - All threads have access to these blocks of memory. Easy for threads to access the same memory location. No need to pass pointers.

        - Technically, the thread needs to be in the function where the static variable is declared to access it.
    - Sorta shared: Heap (stuff you've malloc'd)
        - All threads still have access to the same heap, but you'll need to pass pointers to reliably access the same memory location.
    - Not shared (technically): Stack (local variables you declare in a function)
        - Each thread gets their own stack. Pretty hard for threads to access each other's stacks without pointers. Using something on someone else's stack is not very wise. That value can get changed to something else if they return from one function and then call another. Also, if that thread terminates, the kernel will probably start messing with their stack to free it up.
        - In the conceptual model, stacks are not shared at all.
    - Really no way to share: Thread ID, and Registers (Program Counter, Stack Pointer, and other general purpose registers)
- sharing of variables across threads - global, static, stack
- basic properties of a semaphore
    - Like a globally shared, safe integer
    - Conceptually, you can think of it like "keys" to certain parts of your code. Processes can check out a key, but they have to return it
    - If all the keys are checked out, processes have to wait for them to be turned back in
- semaphore initialization (sem_init), wait (sem_wait), post (sem_post)
    - Sem_init
    - Sem_wait
        - If semaphore > 0:
            - Decrement and continue
        - Else:
            - Wait until semaphore >0
    - Sem_post
        - Increment the semaphore
        - continue
- data sharing / protection / thread safety
- creating and joining threads, detached vs. joinable threads

https://thispointer.com/posix-detached-vs-joinable-threads-pthread_join-pthread_detach-examples/

A thread can either be in detached mode or joinable mode. It starts in joinable.

joinable - set by calling pthread_join(thread). won't release any resources even after thread runs its course

detached - call pthread_detach(thread). releases its resources on exit

- Using a binary semaphore as a mutex
- Shared data paradigms - principles and example code
  - Producer-consumer
    - How to optimize when we have producer and consumers without corrupting data
    - Have 3 semaphores
      - Mutex On data - tells us if we can access the shared data. Binary semaphore, data is available to change 1, or data is not 0
      - Items - Represents items available for the consumers. Starts at 0, increments when the producer calls post on items and puts an item on the queue
      - Slots - Represents the number of slots we have in our Queue, initialized to 4 if our queue size is 4.
    - Producer
      - First - Producer calls wait on slots - checks to see if there is a slot on the queue, this decrements slots.
      - Producer then calls wait on Mutex - checks to see if data is available to be touched.
      - Producer puts the item on the Queue
      - Calls post on Mutex to say data can be changed again. Calls post on Mutex because it is the most contested thing, could have producers or consumers waiting on the mutex
      - Finally calls Post on Items to let the consumer know there is an item
    - Consumer
      - Calls wait on items - If yes decrements items
      - Calls wait on Mutex
      - If both return true, it pulls the item off the queue
      - Calls post on Mutex
      - Calls post on slots - It pulled the item off the queue and so there are most spots in the queue

        - Executes whatever was on the queue...
    - Readers-writers - Data shared that is accessed and shared that people are changing - **Has been on previous tests**
        - You have a file or some other resource and you have readers who just look at the file, and writers who are modifying the file.
        - Because the readers are not modifying the file in any way, you can have as many readers
        - If a writer has a resource checked out then no readers are allowed to read to avoid corrupt data.
        - The most common solution - Writer Mutex and a reader semaphore. The readers when they access the file checkout the writer mutex or try to get that mutex.
        - Have a reader lock and a writer lock. Rlock = 1, Wlock = 1. Reader comes along and calls semwait() checking out the reader lock and the writer lock. Also calls semwait() on number lock. Everything is now at 0. Calls post on the number lock and the reader lock. Then we read, (writer is still locked). After we are finished reading IF we are the last reader, we put the writer lock back

level

Parallelization

- Formulas for speedup and efficiency
    - Speedup of entire code (conceptually):
        - $\frac{time\ without\ parallelization}{time\ with\ parallelization}$
    - Speedup: $S_p = T_1 / T_p$ where p = # cores
        - S_p is *relative speedup* if T_1 is running time of parallelized version of code running on 1 core
        - S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core
    - Efficiency: $E_p = S_p / p = T_1 / (pT_p)$
        - Reported as a percentage in the range (0,100]
        - Measures the overhead due to parallelization
- Amdahl's Law
    - There's a mathematical limit to sequential speedup gains
    - The sequential part of the code is limiting the speedup. In other words, the total speed of a program is most affected by its largest bottleneck.
- Limits of parallelization (e.g., number of cores, size of task)
- OpenMP
    - Difference between dynamic and static scheduling
        - Static scheduling:

                - OpenMP at compile time determines how much to divide up work (like a for loop) between threads.
                - Use when operations are expected to be the same in size.
                - Pro: Less runtime overhead.
            - Dynamic scheduling:
                - OpenMP at runtime determines how much to divide work between threads.
                - Use when uncertain if every iteration requires the same amount of work.
                - Pro: when specifying a chunk size you can increase the amount of work done disproportionately to the amount of overhead to get the work done.
    - Pros and cons of small and large "chunk" sizes

I/O multiplexing

- Advantages/Disadvantages over thread-based programming model
    - Advantages:
        - Epoll flows are faster than the time taken to spawn threads
        - Threads consume a non-trivial amount of memory because each thread has a stack.
        - Epoll doesn't require a context-switch overhead.
        - Gives the program the control over scheduling, whereas threads give the system the control over scheduling.
    - Disadvantages:
        - Obscures the logical flow of the program. Programmer has to do extra bookkeeping.
- Concepts behind between select() and epoll(), including their differences, advantages/disadvantages of each, etc.
- Level-triggered (default) vs. Edge-triggered monitoring with epolL
    - Level-triggered: Notified regardless of new data ready on an FD
    - Edge-triggered: Notified only when there is NEW data ready on on an FD
- Socket operations with blocking vs. non-blocking I/O
    - Non-blocking:
    - Blocking:
    - Which can be used with epoll/select, and under which circumstances
        - For a blocking I/O, epoll_wait() tells you when you can read from a buffer in the file descriptor table stored by the kernel. This isn't needed for non-blocking I/O.