Environment

- setting environment for program execution

Putenv “name=value”

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

int main(char \*argv, int argc, char \*env)

getenv(“name”)

File descriptors

- stdin, stdout, stderr

0 1 2

- redirection / duplication within a program

Open pipe, fork, exec v, connect stdout of parent and stdin of child.

- pipe concepts

  Int pipe(int pipefd[2]) [0] is read end [1] is write end.

- dup2() concepts

Int dup2(int oldfd, int newfd) -> refer to the old fd on the new fd.

Dup2(4,1) = 1 refers to disk file pointet at by 4.

Shell usage

- file descriptor redirection / duplication

1> file name (stdout to file)

2 > filename (stderr to file)

& > filename (redirect both stdout and stderr to file

2>&1 redirects stderr to stdout. (& clarifies it is not filename)

\*Fd defaults to 1 if not set.

- shell pipelines

Process | process to feed output into

- jobs / job control

[1] 25647 ([job#] pid)

Refer to jid with %, nothing for pid

Ctrl-z = pause current job.

Jobs – list jobs

Fg &[jid] resume job in foreground

Bg [%jid] – resume job in background, no id = resume last job.

Run job in background – program &

Kill %[jid]/pid – stop job.

- basic usage of echo, cat, grep

Echo writes arguments to stdout.

Cat concat files and print on stdout

Grep searches for patterns in each file. Grep [pattern] [filename]

Outputs: every line where there is a match.

-w = whole word matches. -i ignore case, -v find non-match.

Processes

- fork

Returns pid of child for parent, 0 for child process.

- concurrency, concurrent execution

Concurrent if flows overlap in time, execv, fork, etc.

  - synchronization with wait() / waitpid()

  Wait in parent process to synchronize parent action with exit of child process.

- execve

Execv but passes in environment variables.

 - orphaned processes ("daemons")

  Killed the parent process without waiting the child process- process gets adopted by kernel.

- zombies/reaping

 Int wait(int \*child\_status) or

Waitpid(pid\_t pid, int &status, int options) – returns pid of waited child.

- process groups, setpgid(), how a shell handles groups with

signals, etc.

Any signal sent to one process in process group, they all receive it.

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

Check the value of the child\_status variable in wait.

Exceptions

- asynchronous vs. synchronous

Asynch = events external to processor, sync = result of executing instruction.

Interrupt = asynch. (timer interrupt, i/o interrupt from device)

- traps, faults, aborts

All synch.

Trap = intentional (system call, breakpoint, special) return to next.

Fault (unintentional but recoverable) page fault, floating point. Rececutes faulting instruction or aborts.

-aborts = (unintentional unrecoverable) Illegal instruction, machine check. Aborts program.

- system calls

All system calls create traps then pass control to kernel.

Signals

Sigset\_t mask, prev\_mask

  - Signal blocking

Sigemptyset(&mask), sigaddset (&mask, SIGINT), sigprocmask(SIG\_BLOCK, &mask, &prev\_mask), sigprocmask(SIG\_SETMASK, &prev\_mask, NULL)

    - Sending vs. receiving a signal

Send signal with kill, keyboard or bash. Receive signals on signal handlers. When signal received, the bit is set so it cant tell

    - kill() system call

Send signals in program kill(pid\_t pid, int sig)

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

SIGCHLD (child stopped) SIGINT (interrupt), SIGCONT (continue) SIGTERM (terminate) SIGKILL(kill -9 cant be blocked).

    - Default actions, overriding with handlers or with SIG\_DFL (default) or SIG\_IGN (ignore).

\*unblocked signals processed after being blocked and held.

**Midterm 2**

Client:

Int getaddrinfo()

Int socket(int domain, int type, int protocol) type = AF\_INET, protocol = SOCK\_STREAM for TCP, SOCK\_DGRAM for UDP

Int connect(int socfd, const struct sockaddr \*addr, socklen\_t addrlen)

Server:

Int getaddrinfo()

Int socket(int domain, int type, int protocol)

Int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)

Int listen(int sockfd, int backlog)

Int accept(int sockfd, struct sockaddr \*restrict addr, socklen\_t \*restrict addrlen, int flags)

Network protocols  
  - IP content: source IP address; destination IP address  
  - TCP/UDP: source port; destination port  
  - Notion of well-known ports (you should know the well-known ports for HTTP and DNS)

Echo sever 7, ssh server 22, email server 25, web servers 80.  
  - TCP vs. UDP characteristics, TCP three-way handshake. UDP sends in datagrams (fixed chunk sizes), unreliable datagram deliver from process to process cant guarantee delivery. TCP sends byte streams from process-to-process over open connections.  
  - Unique identification of a "connection" between two hosts (i.e., four-tuple)

A connection is uniquely identified by the socket address of its endpoints (cliaddr:clipart, servaddr:servport).  
  - IPv4 address characteristics, length, presentation

32 bit ip addresses stored in IP address truct. Always stored in network byte order (big-endian). Getaddrinfo/getnameinfo to convert between ip address and dotted decimal.

Network prefix and host number.

Ex) if ip addr = 128.143.137.144 -> netmask = 255.255.0.0, network id = 128.143, host number = 137.144

Sometimes given with prefix form 128.143.137.144/16 (prefix is 16 bits long)

Sockets  
  - types: SOCK\_DGRAM and SOCK\_STREAM  
    - underlying transport protocol

DGRAM = UDP, STREAM = TCP  
    - how each differs or is applicable (or not) for each socket type (i.e., SOCK\_DGRAM and SOCK\_STREAM)

Client doesn’t call connect, just sends. Server does not need to call listen. Once it is binded, it can receive data from recv)  
    - how each differs or is applicable (or not) to client or server  
    - kernel- vs. user-level functionality associated with each

socket is a system call? Kernel assigns ports, redirects incoming requests to the correct ports.  
  - 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  
    - what happens if peer closes connection  
  - getaddrinfo() usage, give it port number and other settings and it converts it from string to socket address structures.

URL  
  - parts breakdown: protocol, hostname, port, URI (including query string)

Ex) <http://www.cmu.edu:80/index.html>

Protocol = http

Hostname = [www.cmu.edu](http://www.cmu.edu)

Port = 80

Uri = index.html

HTTP  
  - Request / response line

Request line: <method> <uri> <version>

Response line: <version> <status code> <status msg> (common codes 200 = OK, 301 = Move, 404 = Not Found)  
  - Headers / end-of-headers signal

Format: <header name>: <header data> “\r\n” signals end of header. Double \r\n signals end of headers start of content.

- Content-length header

How many bytes are being sent/read.  
  - How a query string is formed

After uri, ?query\_1&query\_2 etc. ?= start of arg list. & = argument divider. +/%20 = space.  
  - GET vs. POST

Get only sends a request to get an object/site/program. Post sends data to be processed (request to a database with login info, etc.)  
  - Request body = none, unless using post.  
  - Response body = the content you requested.

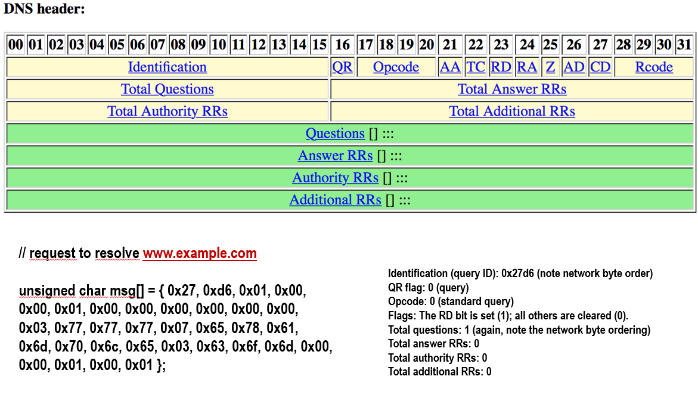
CGI  
  - How CGI is implemented by a Web server (Web server perspective)

The web server handles the request, extracts arguments from the request and saves to a query\_string environment variable which is passed to cgi program through execve. Server uses dup2 to dup stdout to the connected socket.  
  - How a CGI program operates inside a Web server (CGI program perspective)

A cgi program is run as a forked child of the web server process, where output is returned to the client via the duped stdout fd.  
  - How a client passes data (both GET and POST data) to a Web server.

Client uses a query string to pass in arguments for the cgi program to use.

DNS  
  - Name resolution basics  
  - Types: A, CNAME (compression name)  
  - Name encoding and name compression in DNS messages (the concepts from the lab!)



Table

Description automatically generated

Threads / Semaphores  
  - what is shared between threads, what is not: share all code and data (except local stacks)

Libraries, run-time heap, read/write data. Not shared: data registers, condition codes SP2, PC2  
  - sharing of variables across threads - global, static, stack

Any of these variable types can be affected by action in a thread, sometimes to negative effect  
  - basic properties of a semaphore

Binary/normal (binary only 0,1, normal non-negative).

Semaphore starts, and thread calls P (wait) decrements semaphore by 1. If semaphore == 0, then thread is suspended. When a thread calls V (post), increments semaphore by 1. If there is a waiting thread, now one can be released, decrement p and continue.  
  - semaphore initialization (sem\_init), wait (sem\_wait), post (sem\_post) (all return 0 on ok)  
int sem\_init(sem\_t \*sem, int pshared, unsigned int value) – Initializes semaphore at to value. Pshared = 0 means shared by threads not processes.

Int sem\_wait(sem\_t \*sem) – decrements semaphore pointed to by sem.

Int sem\_post(sem\_t \*sem) increments semaphore.

- data sharing / protection / thread safety

Thread safe functions produce correct results when called repeatedly from multiple functions

Class 1: functiosn that do not protect shared variables – use P/V to fix

Class 2: functions that keep state across multiple invocations – just pass in state as arg

Class 3: functions that return a pointer to a static – pass in address of variable to thread.

Class 4: functions that call thread-unsafe functions – don’t do this.

- creating and joining threads, detached vs. joinable threads

Threads created joinable by default (must be explicitly reaped by another thread).

Pthread\_detach(pthread\_t tid) sets thread to detached (memory is freed authomatically when thread terminates). Must be done explicitly for each thread.

  - Using a binary semaphore as a mutex

  - Shared data paradigms - principles and example code  
    - Producer-consumer

Sbuf – sbuf\_insert creates items and adds them to the buffer. Sbuf\_remove takes the item off it. Producer consumer must manage scheduling (if buffer is full, pause producer, if no items pause consumers)  
    - Readers-writers

Unbounded number of readers and writers accessing shared buffer. Only one writer can be in the critical section at a time, but only the first reader locks the semaphore and the last one unlocks it, since it is not dangerous to only read.

Virtual Memory

- Virtual addressing and pages

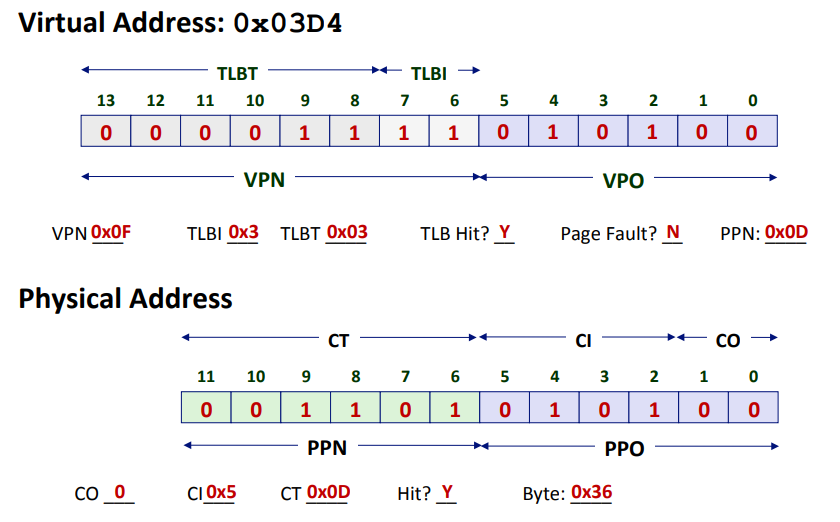
N = 2^n: Number of address in virtual address space.

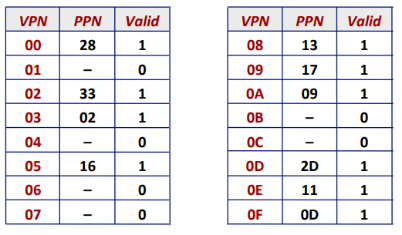
M = 2^m: number of addresses in physical address space.

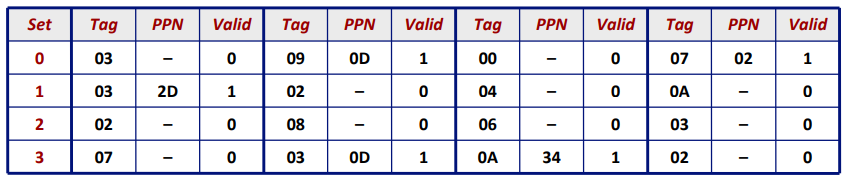
P = 2^p: page size (bytes)

- Page table / Page table entries (PTE)

P = 2^p, there are 2^n-p possible pages. Ex) n=16 P = 4k (4k = 2^12) 2^16-12 = 2^4 = 16 PTE



****

****

**Table

Description automatically generated**

Ex) Given n = 32 virtual addr bits and m = 24 phsycial add bits: P = 1K. VPO/PPO (Offset) = log2(1K) = 10 bits for VPO/PPO. Rest are allocated to VPN/PPN.

**Final Exam**  
   
Parallelization  
  - Formulas for speedup and efficiency – p = processor cores, Tk is running time using k cores.

Sp = T1 / Tp – Sp is relative speed up if T1 is running time of parallel version of the code running on 1 core. Sp is absolute if T1 is running time of running time of sequential version of code on 1 core. (absolute speed up is truer)

Efficiency: Ep = Sp / p = T1/(pTp) = reported as percentage in range (0,100]  
  - Amdahl's Law

T = total sequestional time required. P = Fraction of total that can be sped up (0 <= p <= 1), k = speed up factor. Tk = pT/k + (1-p)T

Ex) T = 10, p = 0.9 k = 9.

T9 = .9\*10/9 + .1\*10 = 1.0 + 1.0 = 2.0  
 - Limits of parallelization (e.g., number of cores, size of task)

Diminishing returns with core numbers, certain tasks are more costly than others, especially synchronous ones.  
   
I/O multiplexing  
  - Advantages/Disadvantages over thread-based programming model

Doesn’t have the synchronous costs of running multiple threads, single threaded but still can handle concurrency. Non-blocking reduced time spent passing control to kernel and back.  
  - epoll concepts – EPOLL\_IN/OUT | EPOLL\_ET  
  - Socket operations with blocking vs. non-blocking I/O  
    must check for -1/EGAIN/EWOULDBLOCK for errno.State machine needed to keep track of where each call goes to when a socket is acted on.