Like, comment, and subscribe for more content!

Basic C syntax

- ref and deref of pointers
    - If you have an array:
        - s[0][0]
        - **s
        - *s[0]

          ^ all mean the same thing. sign
- If you have: int *y = (int *)malloc(sizeof(int));

  What is contained in &y'?

  The address of y

  ^how does this differ from what is contained in y itself?

  Y contains the address of the allocated memory that you called malloc for, &y is the address of the variable y itself, and *y (dereferencing y) is the physical data that is stored within that memory address that y points to

  Y contains a location of an an int. *Y gives you the int that y is pointing to

  To find the value contained, you use *y, which is dereferencing

  ^I am asking what will be contained in y itself (without dereferencing) a point as well?

  It is the same thing, but (&) makes the pointer of type void* (I think)

  When I am trying this I am getting a different address for y and &y (Doing y and &y is not the same, y is the address of what y is pointing to, &y is the address of the pointer itself. y = &(*y))

  yes^

  http://www.c4learn.com/c-programming/c-pointer-operator/
- what causes seg faults and how to avoid them, what the mem layout is
    - Seg fault - program has attempted to access an area of memory that it is not allowed to access.

- When using array make sure the array doesn't go out of bounds. When freeing or deleting memory, make sure u malloc first. And Do not free memory twice.
- Freeing already freed pointers, dereferencing null pointers, accessing past valid array indices, accessing past string boundaries

- what goes into:
  - stack
    - Local variables
  - Heap
    - Dynamically allocated variables [malloc, realloc, calloc and free]
  - text sec
    - Contains the instructions of the program
  - BSS(block stored by symbol)
    - Uninitialized global variables
  - data sec
    - Initialized globals
- dynamic memory allocation: what each function does and how it performs it operations q
  - Malloc - takes a single argument, memory req in bytes. Does not initialized the memory allocated. (asks os for this)
  - Calloc - takes two arguments, initializes the allocated memory to zero.
  - Realloc - resize the allocated memory size (original malloc is deallocated for future use)
- why call free, what free does
  - deallocates the memory previously allocated by a call to calloc malloc or realloc. Free(ptr)
- Free can fail, how?
  - If there is nothing to free..
  - In other words, if pointer is already free (freeing twice will result in a dangling pointer) or if pointer does not exist (was not malloced in the first place).
- diff b/w dynamic and static allocation

    - Dynamic allocation - achieved using certain functions like malloc(), calloc(), realloc(), free in.
    - Static memory allocation - memory allocated at COMPILE time in stack.memory is known during compile time.
- *Function pointers:
    - Good for things like qsort: where you are going to pass another function to it (to do more stuff instead of explicitly coding sorting for all types).
        - Declaring function pointers:
            - void (*ptr)(paramtertype1, parametertype2, ...);
            - Isn't it: return_type(*ptr)(parameter_type_1, parameter_type_2….); ?
            - ptr = &function_name;
            - //to use, but the star is not required:
            - *ptr();
            - //one line declaration: void (*fun_ptr)(int) = &fun;
    - to pass a function pointer to a method: &ptr(parameters)
- *Explicit and implicit linked lists:
    - in context of malloc (where we store the meta data) the
        - Explicit linked list: will contain the next node address within the node
        - Implicit linked list: no memory address, use block size to reference to next and prev nodes (store the size of this node and the previous node).
- *Explicit free list:
    - Contains the address of the next free block

Concurrent programming

Processes

- wait, fork, signals, interrupts, what happens when u fork, what it means to be a child, what to do with children,
- Exec??? --> loads another programs executable in a child process
    - Exec in a parent? You create a zombie....
    - Wait()

        - Wait function returns child pid on success.
        - Wait until all child processes exit or change stage and no more child processes are un-waited for.
            - wait() exits upon completion of only 1 ARBITRARY process (need to call wait n times to reap n children)
    - Fork()
        - Is a system call that creates a process. The process that invoke fork is called parent process. The new process is called a child process. Has different return value, pid.
    - signal()
        - ***typedef void (*sighandler_t)(int);***
        - ***sighandler_t signal(int signum, sighandler_t handler);***
        - ***signal****() sets the disposition of the signal signum to handler, which is either* ***SIG_IGN****,* ***SIG_DFL****, or the address of a programmer-defined function (a "signal handler").*
            - *If the disposition is set to* ***SIG_IGN****, then the signal is ignored.*
            - *If the disposition is set to* ***SIG_DFL****, then the default action associated with the signal (see signal(7)) occurs.*
            - *If the disposition is set to a function, then first either the disposition is reset to* ***SIG_DFL****, or the signal is blocked , and then handler is called with argument signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.*
            - *The signals* ***SIGKILL*** *and* ***SIGSTOP*** *cannot be caught or ignored.*
    - Interrupts
        - Messages sent to the chip to halt current activity and perform our current task.

- what is not shared b/w children
    - address space (means stack and heap)
    - Pid, > 0 in parent, == 0 in child

        - ^The pid assigned to a child is not zero (pid 0 is the scheduler); the value returned by fork IN the child process is 0
    - ppid(shared? same parents)
- effects of multi process on multi core and single core,
    - multi core allows for concurrency through the use of more than one core
    - single core does not allow concurrency (serial)
- efficiency? Implementation?
    - Threads are much faster than processes. (In creation, terminating, and switching between threads) hy hy hy hy hy hyh y h
    - Threads have shared memory that can be useful, but might result in race condition as well. So you have to be careful.
- diff b/w multi process and threads
    - process: own heap, own stack, created by the OS, require more overhead, more information (pid, ppid, registers, stack, heap, file descriptors, signal actions, etc.)
    -
    - thread: shared heap, own stack
- benefits and drawbacks of one over other
    - Threads better than processes >>
        - Far less time to create a new thread than a new process
        - Less time to terminate a thread than a process
        - Less time to switch between two threads within the same process than to switch between processes
        - Threads can communicate via shared memory - processes have to rely on kernel services for IPC
    - Processes better than threads >>
        - Processes have their own signals, so something like a segfault would only stop one process, not all. Threads are all stopped by a signal
        - safer

Threads:

- threads what is shared and what is not?
    - Not Shared:
        - Stack
        - Registers
    - Shared:
        - Everything else
- what to do with child threads
    - join (wait for thread to return)
    - detach (don't wait on thread and let it run on its own)
- method calls, parameters, return values
    - pthread create
        - int pthread_create(pthread_t **thread*, const pthread_attr_t **attr*, void *(**start_routine*)(void*), void **arg*);
        - If successful, the *pthread_create()* function returns zero. Otherwise, an error number is returned to indicate the error.
    - Join
        - **int pthread_join(pthread_t** *thread***, void** ****retval*****);**
        - On success, **pthread_join**() returns 0; on error, it returns an error number.
        -
- sync constructs- locks, mutexes, what are locked and what are not
    - Critical sections are locked
    - Atomic statements
        - Atomic: either an entire action goes through or it doesn't go through at all. Appears to occur without being interrupted.
- what locks guarantee and not guarantee
    - Guarantee:
        - Mutual exclusion
        - Prevent race condition
    - Don't guarantee: deadlock.

- Semaphores
    - Structure that increases/decreases in value using sem_post(sem_t * sem)/sem_wait(sem_t * sem), respectively. When the value of the semaphore reaches 0 the calling process blocks; one of the blocked processes wakes up when another process calls sem_post
    - A semaphore is a data structure used to help threads work together without interfering with each other.
    - When to use semaphores?
        - When trying to solve the Dining Philosophers Problem, semaphores represent a suitable locking mechanism.

- ,conditional variables
    - Synchronization mechanism used to wait for a particular condition to become true
    - pthread_cond_wait(condition, lock):blocks the calling thread until the specified *condition* is signalled. This routine should be called while *mutex* is locked, and it will automatically release the mutex while it waits. After signal is received and thread is awakened, *mutex* will be automatically locked for use by the thread. The programmer is then responsible for unlocking *mutex* when the thread is finished with it.race
    - pthread_cond_signal(condition, lock): if any threads are waiting on condition, wake up one of them. Caller must hold lock, which must be the same as the lock used in the wait call.
    - pthread_cond_broadcast(condition, lock): same as signal, except wake up all waiting threads.
    - Note: after signal, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.
    - Warning: when a thread wakes up after condition_wait there is no guarantee that the desired condition still exists: another thread might have snuck in.
    - Usually used in while loops
- ?
    - Atomic instructions

        - Set of instructions whose executions cannot be split up from each other
- live lock and dead lock? what causes deadlock how to avoid it?
    - Race Condition:
        - One thing finishing before another does when the two are consecutively related
        - Can be prevented with semaphores, locks, etc..
    - Deadlock:
        - Everyone is waiting on something and no forward progress is being made
            - Circular wait
            - everyone waiting on someone else is a circle -- creates a deadlock
        - Holding Wait
            - Once a structure is acquired, you don't let go of it
            - Deadlocks could be fixed by giving up lock if you don't have resources to make forward progress
    - Livelock:
        - No forward progress is being made but the threads are awake and live (not sleeping)
        - Example:
            - Dining Philosophers Problem
            - continually trying to acquire a lock which cannot be acquired
- **How to use a semaphore as a condition variable?**
    - Substitute the condition variable with a binary semaphore… since semaphores can be incremented/decremented from any thread or process depending on the flag set during initializing.
    - Wouldnt this be an example of how to use a semaphore as a lock?
- **Combinations of locks, semaphores, and conditional variables**
- **How to use a lock and conditional variable to simulate a semaphore?**

Dynamic memory allocation

Malloc, Calloc, Realloc, what is the difference between them ? Which one is used to do something ___ ?

malloc()	Allocates requested size of bytes and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes all the bytes to zero
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space(this will free the pointer that you give it while it reallocates the space--careful when using).

- Saturation. What happens when you have a test program that allocates all of available dynamic memory?
    - Memory saturation is when all of the memory on a machine is used (dynamically allocated) and cannot be used for future mallocs.
    - In this case, any malloc after the saturation threshold has been reached will cause the machine to crash completely.
    - Some compilers kill the process that caused saturation to prevent a machine crash.
- Fragmentation. It is possible to allocate and free many small blocks in a way that leaves only small blocks available for allocation. When a request for a large block is made, the request fails even though the total amount of free memory is larger than the requested block size. There are ways to deal with this problem. First, one part of your memory resource can be reserved for small blocks, leaving the rest available for larger blocks. Second, large blocks can be allocated from one end of your memory resource and small blocks can be allocated from the opposite end.
- How to prevent fragmentation:
    - Depending on the workload, these algorithms are very useful when trying to prevent fragmentation of memory:
    - *Malloc algorithms:

        - First fit: looking for the first free block to allocate.
        - Best fit: traverse the entire list looking for the closest fit to what you try to malloc(least amount of space that can fit).
        - Worst fit: looking for most space. (allows you to have larger space if workloads were similar to block size)

Network Programming:

- IP address how diff from hardware address
    - Hardware address is always unique to every single machine/
    - IP address can be relative to a specific router and can be the same if another router is used somewhere else (for example: 192.168.1.1 is pretty common in many networks, not unique)
- what is LAN, IP, TCP, UDP, about DNS,
    - Internet Protocol(IP):
        - higher-level address for accessing device over internet
        - every device has hardware and IP address
    - IPV4:
        - 32 bits split into octets(groups of 8)
        - we're running out of IP addresses cause we can only have 2^32 bits
    - IPV6:
        - 128 bits
        - slow adoption, no one is supporting it
    - *LAN Connections: through a cable directly.
        - Local area network.
        - All machines can see each other in this network.
    - *In LAN connections, every machine is associated with a HW Address(for hardware) and IP Address(for the network).

- *UDP = datagram
  - UNRELIABLE: may not deliver the packet. If link failure occurs or interference occurs, etc, UDP may not deliver the packet.
  - If a packet gets dropped for any reason, no requests for sending it again will be placed.
  - Packets may not arrive in the correct order either.
- *TCP:
  - allows to create simulated connections between machines.
  - Tunnels
  - RELIABLE: Both sides of connections keep checking and making sure that all of the packets get to their destinations! - 3 way handshaking mechanism
    - The data will be sent in the correct order also
  - When a packet reaches a port, any application that is listening to the port will know that this package is theirs and it will take the data packet.
  - If a packet is dropped, the machine that requested it will request it again.
- *DNS (domain name service): map readable names to ip addresses.
- if a domain is requested:
  - check DNS cache for ip address.":
  - If not there, query DNS server.
    - a- get ip address.
    - b- cant find it, so search DNS authority.
    - c- cant find it, error: DNS not found.

- basic method (FUNCTION) calls
  - Addressinfo:
    - int getaddrinfo(const char **node*, const char **service*,
      const struct addrinfo **hints*,
      struct addrinfo ***res*);
      - Given *node* and *service*, which identify an Internet host and a service, getaddrinfo() returns one or more *addrinfo* structures, each of which

contains an Internet address that can be specified in a call to bind or connect

- If no host was found, it returns NULL in res

- socket
  - *On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.*
- *connect*
  - ***int connect(int** sockfd**, const struct sockaddr** *addr**, socklen_t** addrlen**);***
    - *connects the socket referred to by the file descriptor sockfd to the address specified by addr.*
  - *If the connection or binding succeeds, zero is returned. On error, -1 is returned, and errno is set appropriately.*
- *bind*
  - ***int bind(int** sockfd**, const struct sockaddr** *addr**, socklen_t** addrlen**);***
    - ***bind**() assigns the address specified by addr to the socket referred to by the file descriptor sockfd.*
  - *On success, zero is returned. On error, -1 is returned, and errno is set appropriately.*
- *listen*
  - ***int listen(int** sockfd**, int** backlog**);***
    - ***listen()** 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*
  - *On success, zero is returned. On error, -1 is returned, and errno is set appropriately.*
- *accept*
  - ***int accept(int** sockfd**, struct sockaddr** *addr**, socklen_t** *addrlen**, int** flags**);***

        - *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.*
    - *On success, these system calls return a nonnegative integer that is a descriptor for the accepted socket. On error, -1 is returned, and errno is set appropriately.*
- what kinds of socket are there?
    - Server (Binding) socket: waits to accept connections from clients approaching the port
    - Client (Connecting) socket: tries to connect to a server socket at a specific port
- listening socket? what gets returned from a socket? how to read and write from it? connection? file descriptors? diff b/w client and server?
    - When socket is called, it returns a file descriptor corresponding to that specific socket for future use.
    - Future use includes:
        - Binding or connecting.
        - Accepting connections.
        - Reading (read) writing (write).
-

Distributed Systems (EC):

Did the professor say if DS will be on exam or not? Would appreciate an answer

^he said it was EC

- Not too sure how in-depth the questions will be but this link is probably helpful
- https://courses.cs.washington.edu/courses/cse490h/07wi/readings/IntroductionToDistributedSystems.pdf
- Benefits: No single point of failure, micro processors are cheaper, multiple processes is better than a single process, it's more scalable than a main frame,
- Disadvantages include, (lack of software ???), coordination and synchronization issues, security issues, no global state/vars
    - There are also issues with faults
- Fault types include:
    - Failstop - this is a processor halt that can be detected by other processors
    - Crash - the processor halts, but from the outside it isn't detectable that the processor has halted or if it is just doing work that takes a long time
    - Byzantine failures - the processor seems to be working and is giving back answers but they are just wrong

System level I/O

- **Terminal commands**
    - Ls: list all files in current directory'
    - -a shows hidden files
    - -l detailed list
    - Chmod: change mode (alter permissions) (Kevin Patel)
    - Chown: change owner of a file
    - Pwd: prints absolute path to current directory
    - Cd: change working directory
        - . : (single period) access current directory
        - .. : (two periods) access parent directory
    - Mkdir – makes directory
    - Rmdir –remove directory

Files

- Regular file contains arbitrary data
- text files - regular files that contain ASCII or Unicode characters

- binary files – everything else
- to kernel there is no difference
- absolute pathname starts with a slash and denotes path from root
- relative pathname starts with a filename and denotes a path from the current working directory

Directories

- Root directory is /
- Directory is a file consisting of an array of links where each link maps a filename to a file, which may be another directory
- Each directory contains at least two entries (.) & (..)
- Working directory – each process has a current working directory that identifies its current location in the directory hierarchy

Permissions

- r – read
- w – write
- x – execute

File abstractions in C

- Int Open – system call
  - Given a filename it returns a number representing the file's file descriptor
- Read
  - Reads at most n bytes from the current file position of descriptor fd to memory location buf. Returns -1 on erroe, 0 on EOF or number of bytes read
- Write
  - Copies at most n bytes from memory location buf to the current file position of descriptor fd
- Read and write can sometimes transfer fewer bytes than the application requests called short count
  - Encountering EOF on reads. Suppose we ask for 50 bytes but file is only 30 bytes long so only 30 bytes are read
  - Reading text lines from a terminal – if open file is associated with a terminal (keyboard and display) then each read function will transfer one text line at a time returning a short count equal to the size of the text line
  - Reading and writing network sockets – if open file corresponds to network socket then internal buffering constraints and long network delays can cause read and write to return short counts
- Int Stat
  - Takes as input a filename and fills in the members of a stat structure with file metadata
  - Stat structure
    - >St_size – file size in bytes
    - >St_mode – encodes both the file permission bits and the file type
- Int Fstat
  - Same as stat but takes file descriptor instead of a filename
- FILE * fopen

-Able to buffer (open cannot buffer). You may ask for 2 bytes but it sets asided 10 in case you need more later

- More portable because “open” system call may be called something else on a different machine

o DIR * opendir

-Takes a pathname and returns a pointer to a directory stream

-A stream is an abstraction for an ordered list of items, in this case directory entries

o Struct dirent * readdir(DIR * dirp)

- Returns pointer to next directory entry in the stream dirp, or NULL if there are no more entries

- File Descriptor numbers
  - Stdin --> 0
  - Stdout -->1
  - Stderr --> 2
  - So, all other file descriptors start from 3 onwards

- Descriptor table, file table, INode table
- Blocking vs Nonblocking IO calls