Checklist04

Scripting

- What is Unix?
  - *Unix is an operating system that treats everything as file*
  - *Unix is about small programs (filters) connected by redirected I/O to form powerful pipelines*
  - What are the **three tenants** of the **Unix Philosophy**
    - *Write programs that do one thing well*
    - *Write programs that work together*
    - *Write programs that handle text streams*
  - What are **filters**? What do they do and how are they related to the **Unix Philosophy?**
    - *A filter is a program that takes input from stdin, processes it, and outputs to stdout*
    - Examples are : `grep` , `sort` , `wc`
    - *they do one thing well, are designed to work together, and operate on text streams*
  - What happens during **I/O redirection?**
    - *default for `stdin` is keyboard, default for `stdout` is terminal*
    - *I/O redirections changes where input/output comes from/goes*
- What is **concurrency and parallelism?**
  - **Concurrency:**
    - *multiple tasks are in progress at the same time*
    - *doesn't require multiple CPUs*
    - *tasks interleave (switch back and forth)*
    - *think one CPU rapidly switching between processes*
  - **Parallelism:**
    - *multiple tasks are literally executing at the same time*
    - *requires multiple cores/CPUs*
  - **So concurrency is more about structure (how tasks are organized) and parallelism is about execution (actually happening at once)**
  - What makes **concurrency** challenging?
    - **Race conditions:** *multiple tasks compete for the same resource in an unpredictable manner*
    - **Deadlock:** *multiple tasks are stuck waiting for each other (indefinetly)*
  - What are some different types of **parallelism**?
    - **Data parallelism:** same operation applied to different pieces of data

        - **Task Parallelism:** Different tasks run in parallel
        - **Embarrassingly Parallel:** Tasks are completely **independent**, no communication or synchronization required
    - What are some common examples of these types of **parallelism?**
        - **Data parallelism:** Image processing (apply filter to many pixels at once), matrix multiplication
        - **Task parallelism:** A web server in which:
            - thread 1: handles requests
            - thread 2: logs activity
            - thread 3: database queries
        - **Embarrassingly Parallel**: running simulations with different random seeds

Shell

- How do you construct a **unix pipeline** to do the following:
    - extract fields or characters
        - `cut -d',' -f2 file.csv`, `awk '{print $1}'`
    - search based on a pattern
        - `grep "error" file.txt`
    - count the number of lines
        - `wc -l`
    - search a directory based on a pattern or file attribute
        - `find . -name "*.c"`, `find . -type f`
    - list the process on the current machine
        - `ps aux`, `top`

Python

- How do you use **map, filter**, and **lambda**?
    -
```
map(function, iterable)
filter(function, iterable)

map(int, myList)
filter(lambda x: x%2 == 0, myList)
```
- How do you use **list comprehensions?**
    - `[x for x in myList if {condition}]`
- How do you use **generator expressions?**
    - `(x for x in myList if {condition})`

- How do you **sort data?** (multifactor sorting)
  - ```
    people = [("Alice", 20), ("Bob", 20), ("Charlie", 19)]
    sorted_lst = sorted(people, key=lambda x:(x[1], x[0])) #sorts by age and then name

    #for multifacotr sort:
      # sort by the lowest priority FIRST
      # sort by the highest priority LAST
    ```
- How do you **split** and **join** strings?
  - ```
    mySTR = "Alice,Had,A,Blue,Coat"
    myLST = mySTR.split(',')
    myNewSTR = ' '.join(myLST)
    ```

C

- Why bother with C? Why not just Python or C++?
  - teaches low-level memory and systems concepts
  - produces fast, portable programs
  - foundation for understanding other languages
- Why is C considered a systems programming language?
  - Direct memory and hardware access via pointers and manual memory management
    - `malloc()`
    - `calloc()`
    - `free()`
      - efficient, minimal runtime
      - used to build OS, compilers, embedded systems
- What is C's relationship to Unix?
  - C was created to rewrite Unix portably
    - Unix originally written in assembly
      - Unix system calls and APIs are accessed through C
      - Learning C is learning how to program Unix-level systems

Pointers, Arrays, Strings

- What exactly is a pointer? array? string? How are they related?
  - a **pointer** is simply a variable that stores a memory address of another variable

    - an **array** is a contiguous block of memory holding elements of the same type
    - a **string** is a char array ending with NUL
    - An array acts like a pointer to the first element
    - strings are arrays of `char`
    - pointers can be used to iterate over arrays/strings
- What does it mean to dereference a pointer?
    - Accessing the value stored at the memory address the pointer points to
- How do we get the address of a variable?
    - use the address of operator: `&`
- What are multiple ways to access an element of an array or string?
    -
```
int arr[3] = {10,20,30};
char str[] = "abc";

arr[0] //10
str[1] // 'b'

*(arr + 2) //30
*(str + 0) // 'a'

int *p = arr;
p[1] // 20, same as *(p+1)
```

Memory Allocation

1. How do we dynamically allocate memory? How do we deallocate that memory?
    - allocate using:
        - `malloc(4 * sizeof(int))`
        - `calloc(4, sizeof(int))`
        - `realloc(int *ptr, new # of bytes)`
    - deallocate using:
        - `free(int *ptr)`
2. When should we allocate on the stack? heap? data? What are the advantages and disadvantages of utilizing each memory segment
    - stack:
        - lifetime: function scope
        - advantages: fast, no manual free, automatic cleanup
        - disadvantages: limited size, can't survive beyond function
    - heap:

    - lifetime: until `free`
    - flexible size, survives beyond function
    - slower, manual free required, fragmentation risk
  - Data:
    - lifetime: program lifetime
    - advantages: always available, easy to access
    - disadvantages: less flexible, occupies memory even if unused
    - DATA SEGMENT IS ~READONLY (i.e string literals)
3. What is a single linked list? doubly linked list?
  - SLL -> each node points to next node
  - DLL -> each node points to next and previous node
    - how would we insert an entry into a linked list?
      - ```
        #SLL prepending
        Node *insert(Node *head, int value){
          Node *new_node = malloc(1 * sizeof(Node));
          new_node -> data = value;
          new_node -> next = head
          return new_node
        }
        ```
    - how would we search for an entry in a linked list?
      - ```
        #SLL traversal

        Node *search(Node *head, int value){
          Node *curr = head;
          while (curr != NULL){
               if (curr->data == value){return curr;}
               curr = curr->next;
          }
          return NULL; //not found
        }
        ```
    - how would we remove an entry in a linked list?
      - ```
        #SLL remove (first occurance)

        Node *remove(Node *head, int value){
          Node *curr = head, *prev = NULL;
          while (curr != NULL && curr->data != value){
               prev = curr;
               curr = curr->next;
          }
          if (curr == NULL) return head; //not found
        ```
        ```

```
    if (prev == NULL) return curr->next; // remove head
    else prev->next = curr->next;
    free(curr);
    return head;

}
```

System Calls

- What are **system calls?** How are they different from normal functions?
  - A system call is the way a user program requests a service from the operating system
  - they are the bridge between user space and kernel space
  - they are different from normal functions because they:
    - enter kernel mode
    - access protected resources
    - can fail due to OS-level contraints
- What are some reasons why system calls related to files, processes, and networking would fail?
  - *System calls can fail because the OS enforces rules and limits.*
  - File Related:
    - file doesn't exist : `ENOENT`
    - No permission : `EACCES`
    - Disk Full : `ENOSPC`
    - Invalid file descriptor: `EBADF`
  - Process related failures:
    - too many processes (`EAGAIN`)
    - no memory available (`ENOMEM`)
    - Invalid arguments to `exec()`
  - Networking failures:
    - Network unreachable (`ENETUNREACH`)
    - Connection refused (`ECONNREFUSED`)
    - Timeout
    - Host not found
  - **how system calls fail?**
    - return -1 or NULL in some cases
    - it sets a global variable `errno`
  - **how do we check these failures?**

```
int fd = open("file.txt", O_RDONLY);

if (fd == -1){
  printf("error\n");
}
```

- **How do we get the error message related to these failures?**

```
printf("%s\n", strerror(errno))
```

Files

- What is the difference between open and fopen?
  - `open()` is a sys call, `fopen()` is C standard library func
    - `fopen()` uses `open()` internally
- What exactly is a file descriptor and what system calls can we do with one?
  - its a numerical descriptor that the OS uses to represent an open file or I/O resource
    - examples:
      - `0` -> stdin
      - `1` -> stdout
      - `2` -> stderr
  - system calls we can do with a `fd` :
    - `read(fd, buffer, size);`
    - `write(fd, buffer, size);`
    - `close(fd);`
    - `lseek(fd, 0, SEEK_SET);`
    - `fstat(fd, &buf);`
- How do we get the inode information for a file?

```
struct stat buf; // this struct is the "template" to hold the file metadata
stat("file.txt", &buf);
```

Processes

- What is a **process?**
  - *a process is an instance of a program in execution*
  - What attributes or components does it have?
    - *A process has:*
      - *address space:*

        - *code, data, heap, stack*
      - *kernel state:*
        - *permissions, file descriptors, etc.*
      - *execution context:*
        - *program counter, stack pointer, data registers*
  - What system calls can you do with them?
    - `fork()`
    - `exec()`
    - `wait()`
    - `exit()`
- What happens during the life cycle of a typical **process?**
  - `fork()` -> (parent + child run) -> `exec()` (optional) -> `exit()` -> `wait()`
  - After a **fork**, how does a **process** determine if they are the child or parent?
    - the `pid_t pid`
      - if `pid == 0`
        - child
      - if `pid > 0`
        - parent
      - if `pid < 0`
        - fork failed
  - After a **fork**, who executes first: parent or child?
    - undefined, no guaranteed order
    - it depends on OS scheduling
    - non-deterministic -> this is what makes concurrency so tricky
  - What happens when a process performs **exec**?
    - it replaces the entire process memory:
      - code
      - heap
      - stack
    - it loads in new code so basically changes its identity
    - keeps the same pid and file descriptor
    - **DOES NOT CREATE A NEW PROCESS, ONLY TRANSFORMS THE CURRENT ONE**
  - Why is it necessary to perform a **wait?**
    - lets the parent wait for the child to finish
    - prevents zombie processes
    - collects the child's exit status

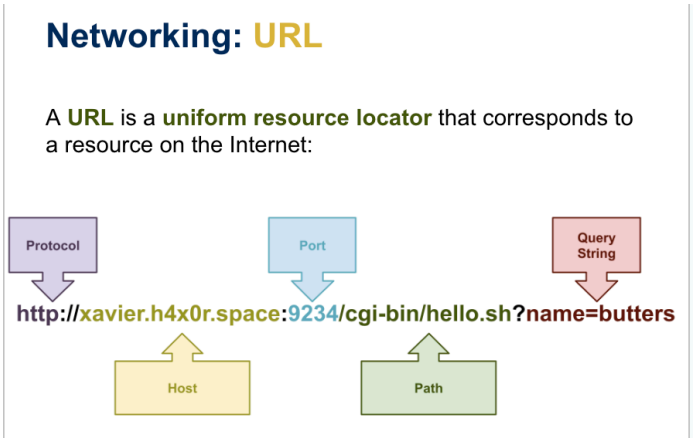
        - WITHOUT WAIT:
            - child finishes
            - parent doesn't "reap" it
            - OS keeps it as a zombie process
    - What is the purpose of **exit?**
        - terminates a process
        - returns the status code to the parent
        - frees resources (eventually cleaned by OS)
        - signals "I'm done", provides a result for `wait()` to collect
- How do we prevent a **fork bomb**?
    - A **fork bomb** is a program that repeatedly calls `fork()` exponentially until the system runs out of processes
    - How we prevent **zombies?**
        - limit the number of processes per user
        - OS enforces limits
    - Why would we want to prevent these situations?
        - Fork bomb can freeze or crash the system
            - due to the consumption of process table entries, memory, CPU scheduling slots
        - Zombies waste process table entries by not being properly reaped
- What is a **signal**
    - *a signal is a software interrupt delivered by the OS to a process to notify it that an event occurred*
    - What does a **signal** do to another **process?**
        - *basically an asynchronous notification from the OS that triggers an immediate response in that process*
    - How do we send a signal to another process?
        - *we can use `kill(pid, singal)` system call*
            - despite the name, it does NOT always mean "terminate"
            - it just means send signal
        - ```
          kill(pid, SIGINT); //interrupt (like Ctrl + C)
          kill(pid, SIGSTOP); // pause process
          kill(pid, SIGCONT); // resume process
          kill(pid, SIGKILL); // force kill (cannot be caught) i.e `kill -9`
          ```
    - How do we catch a **signal** and handle them?
          ```

- ```
  void handler(int sig){
    printf("Handled signal %d\n", sig);
  }

  signal(SIGINT, handler);
  ```

Networking

- What is **networking?**
  - *Networking is the process of connecting computers so they can communicate and exchange data over a shared medium (like the internet or a local network)*
  - What is **bandwidth** and **latency** and how do we measure them?
    - **bandwidth:** How much data can be transferred per unit time:
      - measured in bits per second
      - use `wget` or `curl` to measure them
    - **latency:** how long does it take to travel from source to destination
      - think "delay"
      - use `ping` to measure
  - What is an **IP address** and what is a **domain name**?
    - IP address is a numeric identifier for a device on a network
    - Domain name is a human readable name mapped to an IP address
      - translated to IP via **DNS** (**Domain Name System**)
  - What is a **network port?**
    - a port is a logical endpoint for communication on a machine
      - IP address = machine
      - port = specific service on that machine
      - ports allow multiple services to run on the same IP address
  - What is the basic difference between **TCP** and **UDP?**
    - **TCP (transmission control protocol):**
      - reliable
      - **connection-oriented**
      - guarantees :
        - delivery
        - order
        - no duplicates
      - use for:
        - web browsing (HTTP / HTTPS)
        - file transfers
  ```

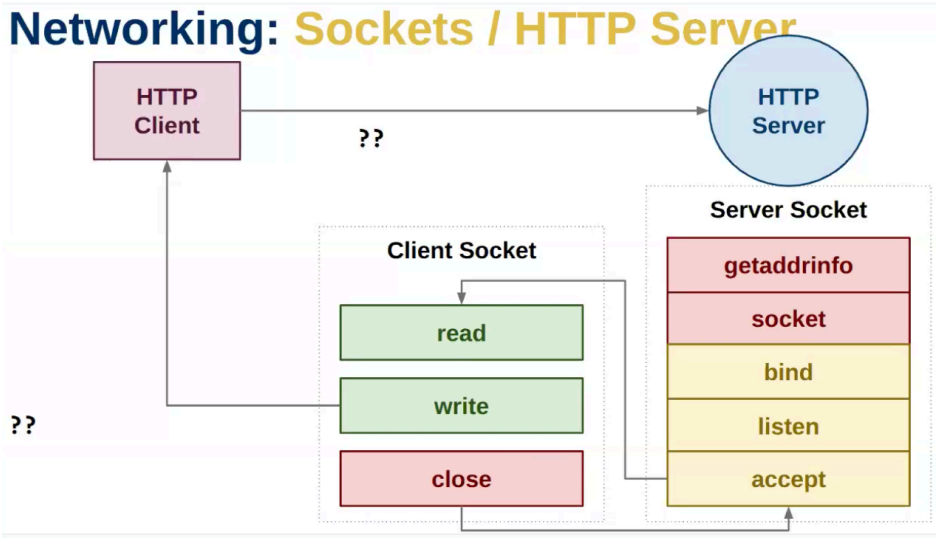
        - **UDP (User Datagram Protocol):**
            - **fast but unreliable**
            - no guarantee of delivery or order
            - **connectionless**
            - user for:
                - video streaming
                - gaming
                - DNS queries
- What is a **URL** and what are its components?
    

- How are **sockets** like **files?** How are they different?
    - **Similarities:**
        - sockets are **treated like file descriptors**
            - so you can:
                - `read()`
                - `write()`
                - `close()`
            - they behave like files in the OS abstraction layer
                - "everything is a file" -> Unix loves this
    - **Differences:**
        - sockets are a temporary stream whereas files are persistent
        - sockets have two way communication, whereas files usually I/O
        - setup socket with `socket()`, and `connect()`, you setup file with `open()`
    - A **socket** is a file descriptor representing a network communication channel instead of a disk file
    -
- What is **HTTP?**
    - *HyperText Transfer Protocol is used for communication between a client (browser) and a server*

- *it follows a request -> response model*
- What **system calls** does a typical **HTTP** client perform? What messages does it send and receive?
  - `getaddrinfo()`
  - `socket()`
  - `connect()`
  - `write()`
  - `read()`
  - `close()`
  - Client + GET request:
    - 

  - **Sends:**
    - an HTTP request message which includes:
      - method (get, post, etc.)
      - resource path (`/index.html`)
      - Headers (list Host, User-Agent)
      - Optional body (for POST/PUT)
  - **Receives:**
    - status line (200 OK, 404 Not Found, etc.)
    - Headers (content type, length, etc.)
    - Body (actual data like HTML, JSON, etc.)
- What **system calls** does a typical **HTTP** server perform? What messages does it send and receive?
  - **server socket:**
    - `getaddrinfo()`
    - `socket()`
    - `bind()`
    - `listen()`

    - `accept()`
- **client socket**
    - `read()`
    - `write()`
    - `close()`



- **Receives:**
    - an HTTP request message
    - same structure as above
- **Sends:**
    - an HTTP response message

- Key insight about Client-Server interaction:
    - HTTP is symmetric in structure, but asymmetric in role:
        - Client **always** sends requests and receives responses
        - Server **always** receives requests and sends responses