# System Creation

TODO:

* Lesson for advanced Shell Commands?

Every program is secretly at least one text based interface. At more advanced levels, its multiple text based interfaces all interacting with each other.

1. Design: We want a basic CLI Chat application - Later make the user interface networked and persistent
2. Create an application without persistence during the data lesson (Chat application?) - Structs, data modeling

**!!! Where does JSON and CSV fall into this????**

1. Transition to persistence
2. Abstract logic out and move to database (Data layer?)

* Decouple logic for database and application

1. Transition database to be networked and deployed (Systemd unit file, daemonize)
2. Transition network application to be networked
3. Create client application: CLI oriented - Frontend backend distinction

Maybe:

1. Show idea of virtual machines
2. Show AWS and many of the abstractions they provide and simple versions
3. Create deployment script - Then show IOT like terraform
4. Create User interface

<https://xesoa.com/wp-content/uploads/2014/04/APUE-3rd.pdf> or <https://scis.uohyd.ac.in/~apcs/itw/UNIXProgrammingEnvironment.pdf> for OS/Process/IPC content

<https://kremlin.cc/rob.pdf> for some programming life skills

But the biggest section, and one I have not successfully found a book for, is case studies. I want to present many MVPs on how to create common patterns, programs and systems. Either by integrating into another system by abiding by their protocol, or creating my own from scratch.

I want a class that teaches students how to hack anything into existence (This), then we can specify by learning how to create good code / work in teams (SWE Class) and design good systems (System Design)

This should be a 12b Corequisite

This should be more functional, less theoretical, we will get through the core material quickly, and spend the majority of time on Case Studies and MVP’s of common patterns and technologies. The goal is making everything very visibly attainable.

**Part 1: Primitives (Power)**

Content

* Hardware/OS/Linux (basic)
* System Calls/Processes (basic)
* Process management syscalls (Fork, Exec, Wait, etc)
* Shell (Interface to computer, how it works)
* File System (basic)
* C

Homework

* Project: Shell in C
  + Creating basic programs and style

**Part 2: Combination**

Content:

* IPC: Files, Signals, Pipes, Sockets, etc (Spend much time here)
* Data: Data structures, Standards (parsing, marshaling, file formats, protocols)

Homework

* Project: Database in C
* Project: Website in C

*Now we have the tools for creating (almost) any program and can focus on learning how many technologies at their core work*

**Part 3: MVP’s, Abstractions and Patterns**

* *These are listed below: This is all about applying what we learned and about thinking how to make things ourselves.*
* Abstractions over IPC
* Internet and Protocols

**Things Learned from Homework:**

* Deployment
* Services vs programs
* Algorithms: (Brief)

Additional Optional Ideas:

* Creating Projects: MVP, Iteration, listing requirements, getting ideas
* Version control
* Containerization
* Optimization
* Errors: Handling, Monitoring, debugging, and reading logs
* Deployment + Creating Apps
* UI
* Code Reuse: Libraries, Frameworks, Transpilers, API’s
* Micromacroservices (Just unix philosophy rebranded)
* Distributed Systems: Concurrency, Parallelism, clocks

Abstractions: Organized from low to high IPC complexity

* Process Control :
  + Loggers
  + Databases
* Multi process programs:
  + .debugger
* Networked Systems
  + . IRC server
* Large Scale Systems

To Place

* Discord bot
* Logger + Parser + Bigger Logging Systems
* Logger parser
* Mail Middleware to auto some actions
* Daemons
* HTTP Framework
* [LSP](https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/) (App integration server to server)
* Java Bytecode (File IO)
* SSH (Networking)

Projects

3 Weeks per project, talk about the MVP during class and do some planning.

Project: Set a precedent: I expect correctness, it's not if you can get something in, it's when I accept your solution. So keep submitting and getting feedback. I EXPECT questions. I am actively concerned if you don’t ask any. This is how your projects will operate.

1. MVP
2. Additional Features
3. Additional Features + Deployment

1: [Shell in C](https://docs.google.com/document/d/1IIc5fqK1okY1o9Btz4tzQE0mOK9BpAxwYd_0bOYFNdk/edit?usp=sharing)

2: [Database in C](https://docs.google.com/document/d/10w1Zj44kkR3cMO4OOy6wDxJh3XKXVE-vs6FjAuLVass/edit?usp=sharing)

3: [Website in C](https://docs.google.com/document/d/1KbeE7Si6X-Z5uVXupAmW4vY0sumRXmirK_mwwm15I08/edit?usp=sharing)

## Current Schedule

Week 1:

1. Jan 29. [Intro and Starting Projects](https://docs.google.com/presentation/d/1GrfjkrMllRfPD8vxBH35nS4lVQl9R6VE5KklEb0yFXI/edit?usp=sharing)

Week 2:

2. Feb 4. [Hardware, OS, Processes and Linux](https://docs.google.com/presentation/d/1VpKaiOcrpD-TfYTTC8vm7TRnLwEOPb9XNUgUwWP-mjA/edit?usp=drive_link)

* HW: [Explore Linux](https://drive.google.com/file/d/1rFUtTi_btmBq0dO5dKEX3xmnyj8-zc02/view?usp=drive_link)

3. Feb 5. [Processes, Syscalls, and Learning (Via manpages)](https://docs.google.com/presentation/d/1gOgbHC1OHPl9Pf_kTIQmkuRoYO8e6HjH4aYxy0JUauk/edit?usp=sharing)

* Practice fork, exec, wait

Week 3:

4. Feb 11. [Shell and the File System](https://docs.google.com/presentation/d/1DMOKEWMyQ45EnqTQ4Ub7n3zOUIyEYmx7STVddjqdGdU/edit?usp=sharing)

* HW: [Shell Practice](https://drive.google.com/file/d/11MUvsAmklRFHqxYhOS1UxvYKKZOZewPQ/view?usp=sharing)

5. Feb 12. [C](https://docs.google.com/presentation/d/1aRSxlus5zvhcfk0oW9baV_2OCk40H0wbSmar8dl7row/edit?usp=drive_link)

* [Project 1: Shell in C](https://docs.google.com/document/d/1IIc5fqK1okY1o9Btz4tzQE0mOK9BpAxwYd_0bOYFNdk/edit?usp=sharing): Part 1

Week 4:

6. Feb 18. Break!

7. Feb 19. Break!

Week 5:

8. Feb 25. [Combination: File IO and Standards](https://docs.google.com/presentation/d/1WHgo0pfTBZjAQxGFzq9KzjrNm0r3_GkOf2Y8Ua5221E/edit?usp=drive_link) (Standards = parsing, marshaling, file formats, protocols)

9. Feb 26. [Combination: IPC and Standards](https://docs.google.com/presentation/d/1Gj4qrNP6lLFFmeE2gL7_vSNjFQQ3MVvGCa-8icXHI10/edit?usp=drive_link)

* Deadline for Shell in C part 1
* [Start Shell in C part 2](https://drive.google.com/file/d/1Orh3uf3e7mOLF2k8tAivi8zpYjRzj6R0/view?usp=sharing)

Week 6:

10. Mar 3. [Computer Networking, the final IPC](https://docs.google.com/presentation/d/1z5zlmMWcFQ2Zm47C7GHfxAtybZ1gxuk-y2m7LZZ_Eq4/edit?usp=drive_link)

* Deadline for Shell in C part 2

11. Mar 4. [Combination: Case Studies](https://docs.google.com/presentation/d/1gM01DZ7iq9ysJ6CqLxs6tmASDNJmlyihqSmZqwtnNgg/edit?usp=sharing)

* [Shell in C part 3](https://drive.google.com/file/d/1c7ncnY3VNy8ZAyYQfVKT168ZhLT1cVFx/view?usp=drive_link)

March 7th: Module 2 Starts

Week 7:

12. Mar 10. [Database: Relational Databases and Data Modeling](https://docs.google.com/presentation/d/10QTxpADG65A_2JBebA_La6fYBy2XxE_HorhXdOxTC50/edit?usp=sharing)

* Deadline for Shell in C Part 3
* Homework: [Data Modeling](https://docs.google.com/spreadsheets/d/1Pt9vvcpNQtLpzpAmmR6uLyOg_Vx_OLPIut3M7kE7Ksg/edit?usp=drive_link)

1. Mar 11. [Database: SQL](https://docs.google.com/presentation/d/1r5evWhdhfZJxVtyk3Z-1XMGcXz0MfyfAJhD8DV00Dw0/edit?usp=drive_link)

* (Optional) Homework: SQL Statements
* Project: [Start Database in C](https://drive.google.com/file/d/12SOQx2H2PDlMwbLbrQloOFID6r8_cAQF/view?usp=drive_link)
  + No persistence, but with relationships

Week 8:

2. Mar 17. File Types and Designing Specifications

3. Mar 18.[Git and Github](https://docs.google.com/presentation/d/1UWMlEMxdP-wIPp3MBvcGoJzkKGWGnbYS9s2UZ9-53lQ/edit?usp=drive_link)

* Project: Continue Database in C
  + Add persistence in binary format

Week 9:

4. Mar 24.

5. Mar 25.

* Project: Database in C Deployment

*– Rest of this is case studies and going over patterns: If students are interested in how to make something, we can review it in class.*

Week 10:

6. Mar 31.

\* Deadline for Database Project

7. Apr 1.

* Project: Web server

Week 11:

8. Apr 7.

9. Apr 8.

Actual database Deadline

Week 12

10. Apr 14.

11. Apr 15.

Week 13

12. Apr 21. Last day: Project showcase and Pizza

#######################################################################################################################################

Big ideas

Axis: Services Code oriented

* Components in a system
* Base Layer IPC/Cloud(VM’s) OS, Frameworks/Transpilers/Compilers
* Side layer IPC/API’s/Cloud Modules, Libraries, Dynamic Link Libraries

These can be created by us, or them

(Maybe 3rd axis: Created by us or them)

* Breaking our systems up Processes/Threads Modules
* Using Concurrency
  + See above (Using other people's services)
  + Performance improvements

Using IPC: Prereq: Processes and process management

* Getting used to IPC methods
* Synchronous vs Asynchronous
* Protocols intro
  + Simplex, Half Duplex, Full Duplex(?)
  + Marshaling
  + Multiplexing
  + Error correction
  + Packets
    - Sizing
* Use cases
  + Connecting with services
  + Performance improvements
* In depth computer networking

Using Frameworks

* Documentation
* Since its code, you have to install it onto your machine

Lesson Ideas:

* Command Line, git and github
* Creating Projects + Project Design Case studies: After Interfaces + Protocols (Libraries, Syscalls, Services)
  + Discord Bot, Trading Bot, VPN, submission system
* Concurrency Models and Multithreading
* System Design Intro: Considerations for Large Scale software development
* Docker and Containerization
* Git for Collaborative Development, Debugging and Developing in Large Codebases
* Business Development and DevOps, Business and Engineer considerations
* Monitoring your system (create a logger, analytics)
* Large scale development?
* Have them do a project, but DDOS it
* Performance
* DevOps
* More hardcore monitoring
* Branching
* Not too much since they will get more experience when they actually develop at a larger scale

Mini’s

* Simplicity and Learning - Write things down simply, I avoid crazy optimizations, do work when you need to. Note, this might not work for everyone, find your own way.
* OOP and the Software Crisis - Provide reading either by Barbara Liskov on ADT’s or The Humble Programmer by Edgar Dijkstra
* Quickly transferring files between remote machines and you.

Problems I’m offering

* Submissions + grading system

Next Semester

* C lesson: Main function, go down in orde rof thingsy
* Better Setup: WSL + VScode
* Mention compiling and C in L1
* Make all projects 1 week earlier and spend more time on final project
* Lesson 1: Overview of SWE, Hardware, OS, Linux, basic CLI
* Lesson 2: Processes, Process management, and System Calls TODO: Split Lesson 1 up and specialize

Don’t wave hands unless at hardware layer - if using a tool - at least have them think of how to implement it

## Projects

3 Weeks per project, talk about the MVP during class and do some planning.

1. MVP
2. Additional Features
3. Additional Features + Deployment

### 1: [Shell in C](https://docs.google.com/document/d/1IIc5fqK1okY1o9Btz4tzQE0mOK9BpAxwYd_0bOYFNdk/edit?usp=sharing)

### 2: [Database in C](https://docs.google.com/document/d/10w1Zj44kkR3cMO4OOy6wDxJh3XKXVE-vs6FjAuLVass/edit?usp=sharing)

### 3: [Website in C](https://docs.google.com/document/d/1KbeE7Si6X-Z5uVXupAmW4vY0sumRXmirK_mwwm15I08/edit?usp=sharing)

## ===========================================================================================

## ===========================================================================================

## ===========================================================================================

Building from scratch => Using Libraries

Programs => Multiprocess Programs => Distributed Programs

Start from OS/C

Projects: Shell, Database, Website with custom Protocol

Current Resources

<https://ocw.mit.edu/courses/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/pages/unit-2-signals-and-systems/signals-and-systems/>

Current Implementations

[tamid\_tech\_education](https://github.com/archerheffern/tamidtecheducation)

## Overview

### Goal

* Create self sufficient powerful developers who…
  + Can learn anything
  + Know where power comes from (Interfaces)
  + Have a fundamental understanding of how things work, and as such can guess how many technologies work under the hood
* Learn basic tools and methodologies that are helpful for creating programs and general productivity

### Ideology

* Algorithms and Data and the core of CS, but Interfaces are power.
* Create people who can convert ideas into code. Teach ideas, how to find ideas, and practice translating ideas into code.
* Engineering should be freeing, don’t worry about doing it the “correct way” yet

### Approach

Learn the takeaways from Operating Systems, without going in depth into exactly how they work, then build up in abstractions from there. A systems approach with less focus on performance and more on getting things done / Coding. Also focus on learning general software development tools and tips. Eg content in “The Practice of Programming by Brian Kernighan and Rob Pike”

Conceptual: OS IPC, Computer Networking, IRC/Websites

Hands on: Linux (CLI, Services/Daemons), Program Structure, IPC (Signals, Pipes, Sockets, File)

Primitives: Hardware, OS, Linux, System Calls

Combination: IPC, Computer Networking

Abstraction: Shell, Databases, Websites

Patterns: Fork, exec

Additional Skills:

* Learning
* Researching
* Git/Github
* CLI / Package management
* Using Libraries/Frameworks

Low level { OS => simple programs (CLI, File applications, syscalls, basic IPC, algorithms, data structures) => Systems (IPC: network, pipes, domain sockets, etc) => Using libraries (Flask, Django, React) } High Level

* Deep dive into OS and how we build on its interfaces
* Research/Coding heavy projects
* Alternating between 1 project material oriented class, and 1 developer productivity class. Dev Productivity will be fast and more focused on coding projects and asking questions.
* Homeworks are kata style and code oriented
* When teaching technologies, teach theory and basic syntax, and then provide the resources to search up additional syntax

#### SWE with Delfino vs Archer

The Delfino SWE course is focused on development in industry - devops, planning, and whatnot.

My implementation is focused on learning how to be your own developer

### Placement in CS Curriculum

After:

* COSI 12b: Teaches Abstraction, Modularity, Interfaces, and general coding
* Something I’ve noticed is that Object Oriented Programming is really a reflection of software systems and how they are designed by cobbling together interfaces of programs (Unix Philosophy)
* (Optional) COSI 21a: Data structures and algorithms

Before:

* System Design: This class will teach how to think more about the efficiency of the systems we have designed.
* Delfino’s SWE: This class will teach how to develop such programs in an industry setting

## Projects

2 Part Projects: MVP with lots of assistance, and second section with a paper to help

* Shell in C. Very simple, no piping, just fork exec. Is to teach how applications are generally written and the C language: Env variables, Arg variables, (hidden) config files (/etc/<name>, $HOME/<name>), logging with $HOME/.<name>\_history , add to /usr/local/bin and should learn the difference between /usr/bin, /bin, /usr/local/bin, create manpage.
  + Prereq: Linux: File system?, fork exec, commands to navigate file system, Shell, PATH variable, file permissions (chmod a+x <filename>) - Have a common issues section
* [~~Linux Daemon~~](https://youtu.be/9nDYYc_7sKs?si=X9uQRyorfNbvNG-C) ~~or~~ [~~C debugger~~](https://blog.tartanllama.xyz/writing-a-linux-debugger-setup/) ~~(CLI, argv, envp, syscalls (ptrace or read, write, ), signals, threads/processes)~~
* Database (File, Data structures, git, using files as data structures, researching)
* IRC server (Networking, protocols, multithreading) -> Cache -> Some extension Thinking of removing this project so we can spend more time on the other ones - Rn I don’t feel it adds too much. Perhaps can be server part 1 (custom protocols, threading), and web server part 2 (libraries, HTTP protocol, DNS, APIs (Database, Data fetching)) Little concerned of impeding on the System Design Web Proxy Project Though (Though I think it's good they learn such content earlier in the curriculum)
* Static web server (preps students for possible project of web development) -> Upgrade to Flask/Django web server + React (dependencies/libraries, github, teamwork, ML, API’s, HTTP, DNS, Flask)
  + Maybe add some data analysis and API calls?
  + Ideas: Mail Server, Trading Bot

## Contents

### Overview of SWE

### Building Basic Systems

* Overview: Program model, application model, interfaces
* Systems: PCAP: Primitives, Combination, Abstraction, Patterns

# TODO: Reorganize: This is a mess

* + Primitives
    - Physical: Hardware, OS, IPC and Syscalls, Signals, Processes, Threads, [everything is a file](https://en.wikipedia.org/wiki/Everything_is_a_file#:~:text=%22Everything%20is%20a%20file%22%20is,through%20the%20filesystem%20name%20space.),
      * syscall(2): exec, fork, wait, open, close, connect, bind, lseek,
    - Conceptual: algorithms, data structures
  + Combination
    - Interfaces: 4 operations (open, read, write, close) and Protocols
  + Abstraction
    - Modules, Libraries, Frameworks, etc, package managers
  + Patterns
    - Databases,
  + CLI Applications, Network Applications, GUI Applications, Web Development
  + Services and Code
    - Libraries, Frameworks, SDKs, etc
    - Network API’s

Takeaways

* How to translate ideas into code
* Where does power come from
* Application patterns
* How to cobble things together to create systems
* Foundational understanding of how many seemingly complex technologies work - how you can build simple versions - and how you can learn to improve upon them

### Developer productivity

* Learning, researching, reading, searching
* Debugging
* Staying up to date and tool discovery
* Learning Metas
* Simplicity/Agile
* Git/Github
* CLI/Package Managers

Takeaways

* Misc productivity methodologies and tools

## Lessons

### Lesson 1: Overview of Software Engineering

* What is Software Development?
* What career paths are available?
* SWE Levels
* What domains are there?
* What projects can I do?
* Tips for starting out
* How to make a program/system summary (Shown below)
* Simplicity, Iterative Development (MVP, User Stories)
  + *Programming is pretty simple: Know what you want, then make it. Google when you need to and try to understand deeply what's going on.*
  + *Research your domain and the problem at hand*
  + *Everyone will say something different about how to "Correctly Code". From this you can infer there isn't a correct way. So just write code, and if something actually becomes an issue, then fix it.*
  + *In general:*
  + *\* Be familiar with your language*
  + *\* Be familiar with your environment aka the available Interfaces (These are your sources of true power)*
  + *\* OS Syscalls*
  + *\* IPC (File, Stdout, network, etc)*
  + *\* Hardware Devices*
  + *\* IO / UI's*
  + *\* Others code and services (Get using package manager, Network API's)*
  + *\* Applications/layers running on (Recursive)*
  + *\* Know your dev tools (git, debugger, code editor, SDK, Command line)*
  + *Do some basic planning*
  + *Write a very basic solution, and then continuously improve upon it.*

### Lesson 2: Systems: Primitives and Combination

Slides: <https://docs.google.com/presentation/d/1VpKaiOcrpD-TfYTTC8vm7TRnLwEOPb9XNUgUwWP-mjA/edit?usp=drive_link>

Homework: <https://drive.google.com/file/d/1rFUtTi_btmBq0dO5dKEX3xmnyj8-zc02/view?usp=drive_link>

*Currently feels very disorganized, choppy, and not really a point in saying all this - I think there is something interesting hidden in here though.*

*Make it about how we need to use others interfaces to not start at ground 0 - Us creating multi process applications comes later*

*Pretty theoretical, should have a lot of practice involved*

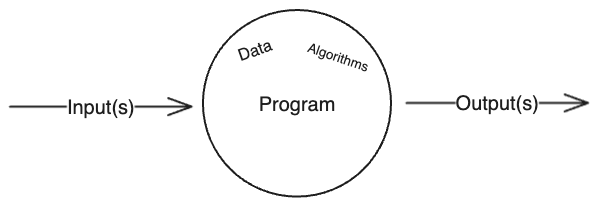
Overview:

* How do we think of programs - Data, Algorithms, IO
* Where is the power, how can we connect things together, how can we do that easily, and how can we discover new functionalities?
* Interfaces
  + Our source of power (hardware) or reducing complexity (abstraction)
  + Patterns
    - Layers - Explains why we are learning OS instead of another layer,
    - (Optional) adapters, controllers, frameworks (technically a layer), libraries (Just nice to know for conceptual reasons - A sort of classification)
    - OOP [design patterns](https://refactoring.guru/design-patterns) apply here
  + How to use any interface
* Systems as many interfaces?
* Creating Command Line Programs - Already know we need interfaces+IO, we also want configurability (argv, envp, config files)
* How does the OS Work?

Creating Systems:   
Data Structures, and Algorithms are so academic… (But still very important)

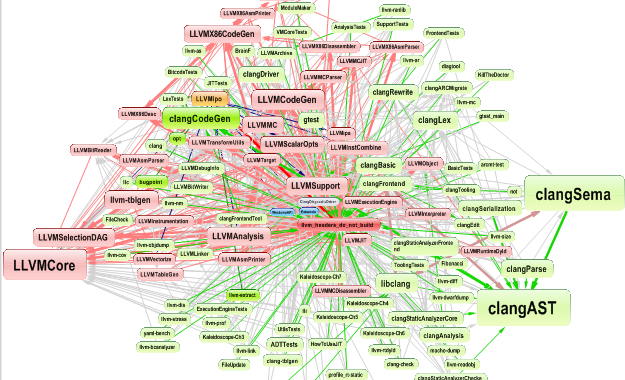
We want power, and to not have to start from scratch, so let's look at interfaces

We think of programs as such…



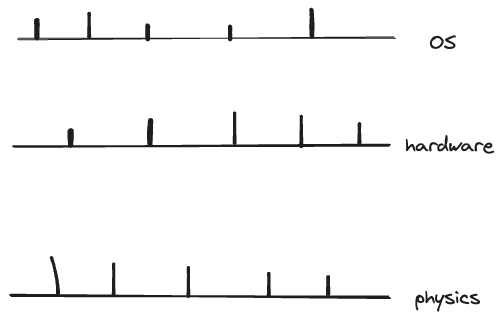
We think of the input and output aspect as the “Interface”

Notice how we can string programs together. This is how we create more interesting systems

* Some interfaces let us interact with the real world (hardware)
* Some do more conceptual, helper things
* We can visualize all of software as very large dependency graph of interconnected modules - connected by their interfaces
* 

This is madness, how can we orient ourselves? Where do we get started? If everything depends on something else, where does it start - is it turtles all the way down?

*We can think in terms of large abstraction layers (paradigm shifters). These will serve as our grounding*



*Each layer exposes its own interface (Speaker Note: Show how web dev, drivers, and network applications fall on this TODO: Distributed multi hardware applications)*

*Domains fall into this model as specific problems that use certain interfaces, algorithms, and data:* TODO: Does this make sense?

There are many patterns of how programs design their interfaces (See OOP Design patterns)

#### Interfaces in depth

So how do we actually string things together?

Interacting with every interface at the foundational level requires the following:

1. Ability to transfer data with interface (In OS this would be Connect, Read, Write, and Close operations)
2. Ability to speak the same “language”, we call this a protocol - we will deep dive into protocols later.

Class Demonstration for interfaces: How to access and just work with any interface: Do several examples of how easy it is to interact with anything you want.

#### What will we focus on?

The interfaces the Operating System - specifically Linux exposes, and later using other higher level interfaces to do a lot of the heavy work for us.

The OS will give us fine grained control over hardware, as well as a fundamental understanding of what is actually possible, as opposed to the much higher abstractions you would learn in a web dev or application focused environment.

We will talk about operating system and its interfaces next time

#### Homework Ideas

* Connect several interfaces together
* Need an example where you have to string stuff together to create a greater system

#### TODO

* Domains fall into the above model as
* PCAP?

\*\* Interfaces, protocols, and marshaling \*\*

\*\* OS syscalls, IPC, some protocol, and some marshaling example\*\*

[Protocols](https://youtu.be/kH7P1ZX44DQ?si=TdiieP15H_GV0YXR)

### Lesson 3: OS and its interfaces, Reading manpages

*Might need to split this up*

Content:

* What is the OS
* Basic Hardware devices (What they do but nothing deeper),
* Processes
  + Signals
* Syscalls
  + execve, fork, wait,
* Interacting with hardware devices

Homework: Start on project 1 - Prerequisites: Linux based system (Have Windows students install WSL)

Provide man(ual) pages as supplemental material:

* environ(5)
* signal(7)
* syscall(2)
* sigaction(2)
* pipe(2) popen(3) pclose(3)
* Do man 2 <syscall> for any of the mentioned syscalls

To read man pages, either run man <name> in your terminal or go to [man7.org](https://man7.org/linux/man-pages/)

To find out more about man command, run man man. I also recommend reading the manpages referenced at the bottom of the man manpage.

[lesson](https://github.com/ArcherHeffern/TechEducation/blob/main/04_research_learning_and_debugging.md)

Homework Ideas

Read X paper and convert its ideas to code

Read git manpages and use features I did not teach (try not to use google)

Contents:

* File System
* Config Files (/etc)
* File Descriptors
* File Patterns
* Everything is a File

Etc files example: /etc/motd

See [Login(1)](https://man7.org/linux/man-pages/man1/login.1.html) and [motd(5)](https://man7.org/linux/man-pages/man5/motd.5.html)

#### Supplementary Material

File System

* man hier
* <https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard>
* [Bytebytego File System Explanation (Video)](https://youtu.be/bbmWOjuFmgA?si=l9lXlEk-1ziLWPM3)

Syscalls

* [Syscalls 1](https://www.youtube.com/watch?v=xHu7qI1gDPA&ab_channel=BrianWill)
* [Syscalls 2](https://www.youtube.com/watch?v=2DrjQBL5FMU&t=99s&ab_channel=BrianWill)

Overview

* [Operating System Basics](https://youtu.be/9GDX-IyZ_C8?si=Blse8gRqvPfY4jum)

### Lesson X: Linux

Contents:

* What is Linux

[How to learn and troubleshoot Linux](https://youtu.be/08upAdSKR4c?si=KeYyM-iYs6RT_92E)

### Lesson X: Terminal

Prereq: Mac: homebrew, windows: WSL, knowledge of OS

Contents:

* What is the terminal, environment of terminals, overview of things you can do with terminal
* running terminal commands - ls
* Learning how to use terminal commands (manpages) - man ls
* Installing more terminal commands (apt) - apt install exa, brew install exa
* Finding more terminal commands to install
* Common commands: managing fs, networking (netcat/telnet, ssh, scp)

[Lesson](https://github.com/ArcherHeffern/TechEducation/blob/main/0x_command_line.md)

### Lesson X—X+1: Networking, Protocols, and IP

Contents:

* Review: We looked at X IPC’s
* What is internet (Routers, Information transmission, etc)
* Protocols and layering protocols
* IP
  + How works: Router recognized, hardware support with Network interface card

Homework: Read internet whitepaper + answer questions, Design a protocol (Mail Server), (Error correction, etc)

### Lesson X: Networking: Advanced Protocols and Sockets

* TCP/UDP
* DNS
* Using sockets
* If you are interested in learning more, read TCP/IP illustrated, the RFC’s, or take system design

Homework/ Ideas:

* Creating your own mail server protocol and implementing

# TODO and Notes

Conventional Program structures

* Configurable with Command line arguments, Environment variables, and config files
* Syscalls, stdout, stdin, stderr, other interfaces

Current Unsure parts

* Database (SQL)
* Frontend
* When to do personal project in addition to homework

Ideas

* Learn about IPC and Syscalls through CLI and Network application
* Theory lesson with homework, then project homework?
* Learn how to effectively sift information: new technologies, docs

TODO: Need to organize big ideas

Programs: Data and algorithms, core syscalls

Systems: Power theory - interfaces: Protocols, connection of interfaces

Kata Homeworks

* Custom Codebase: Get a large codebase from github for this (or a known beautiful codebase)
  + Getting Used to New Codebases
  + Debugging: They must use a debugger / Logs to figure out and issue
* Researching, Git: The above projects should implement this
* CLI: git manpages to do XYZ
* Adding features to large codebase

Starting disclaimer:

You don’t need to learn this to program and create something good. There is nothing stopping you from just getting started on a project and googling as you go.

I also don’t think you need to directly use C or program at a low level to be a good programmer.

But I do know that spending some time at the lower levels, learning the theory, knowing what is truly available, and being aware of what we take for granted will make you a much better programmer

**Approaches:**

Questions: How should interface theory be split up, what should be taught during

=====1====

“Starting Disclaimer”

What is an OS

Programs

* Processes (stdin, stdout, stderr)
* Command line programs

Syscalls for power

Systems + Interface theory: PCAP

* Syscalls for systems (IPC)
* Libraries

=====2====

“Starting Disclaimer”

Systems: PCAP

* Conceptualizing systems, power of interfaces

What is an Operating System

Primitives and some combination

* Processes
* Syscalls + Reading Manpages
* Interface theory contd.
* IPC
* Cli Applications + Examples

=====3=====

Goals:

* L1: Where is the power (basic systems) (syscalls, Stdin, File) - OS, processes (overview), Creating CLI Programs, manpages
* L2: Creating advanced inter process systems with IPC (Network, pipes, signals), processes in depth

=====4===== Is this it!?

L1: Programs: Where is the power, and wielding the power

* Starting disclaimer
* Program model
  + Program as algorithms, data: But this idea is very academic!
* Introduction to interfaces for power - But what level of interfaces should we focus on?
  + We need an interface that lets us run programs while also giving us the ability to impact the world
* Operating System
  + Is our source of power,
  + hardware, abstractions (File system, processes (will learn more about these later))
  + Powerful interface called syscalls (read, write, open, close, exec, fork) + manpages + [man7](https://man7.org/linux/man-pages/man2/syscalls.2.html) list
* Coding
  + Find your programming languages OS module
  + Libraries: Make it easier to use
  + Command Line Applications (argv, envp, config files)

L2: Systems: Creating advanced systems with IPC

* PCAP
* Previously we learned about the primitives of the operating system
* Why we are learning about multiprocess systems and not multi module systems and drawing parallels
* Basic Interface theory
* Processes more in depth
* IPC syscalls

Homework:

L1: Use system calls,

## Classifying OS Items:

* Primitives: Processes, Threads (Compute power, memory), syscalls to do hardware things
* Combination: syscalls to do combination things: IPC: File Descriptors, Pipes, Signals
* Abstraction: ~
* Patterns: ~

## Goals

1. Where is the power? Syscalls, Program model
2. How can we create systems? - modules and processes

## Questions

1. How do syscalls get executed? (Languages -> Asm -> Hardware supported)

## Issues

Why are we learning to create multi-process systems and not just modules?

#### Issue: Modules and Programs

* Within a program: Use libraries and linker to create a system
* Multiple programs: Use IPC to create a system.

Should I address both of these?

* + I assume you already know how to create single program systems with modules, we are now going to focus on multi process (running program) systems.

## Topics

Starting Disclaimer

Program model

Program as algorithms, data, and interfaces

Interface theory (Optional: 3 sections, explain the importance of interfaces, basic interface theory, and protocols)

Systems: Primitives, Combination, Abstraction and Patterns

Modules

Operating System (Hardware, software, Fetch decode execute)

Processes

Threads

Compute Power

Memory

Syscalls (Hardware, IPC, file system)

Libraries

CLI applications (std fds, envp, argv)

manpages

## Homework:

Either practice creating a basic system conceptually or with syscalls

Practice with understanding what is going on at the hardware level

inode, PCB, file descriptor/description

We have a CPU, this is the heart of the computer, it's what runs everything

Kind of like a little REPL but in machine code (1’s and 0’s)

It can run 1 “process” at a time. A process is a running program, think of a python script or your GUI.

*Show executing code and jumping around*

A computer has many processes - but since it can only run one thing at a time, it “schedules” which process to go next using a scheduler. If you have taken Data structures, this is usually a priority queue.

*Show scheduling*

A process is represented by a process control block. It contains all the information it needs.

*Show PCB*

Processes are managed by the OS

One of the instructions a process can run is known as a system call. System calls are very powerful functions that require hardware and OS Support to complete.

The full list of system calls is \_\_\_

Some system calls we are interested in are “Fork, Exec, and Exit”. These allow us to create a copy of the running process, execute a new program within our process, and exit our current process.

When the computer boots up, it creates the init process, it is the parent of all other processes:

When a process is created (only by another process) - it is provided some default information:

* Argument variables
* Env variables
* 3 File descriptors (references to files): Stdin, stdout, and stderr

Recall in the PCB, we had an open file section. We have a file System: Many things are files you may not expect:

Represented by an iNode, a data structure that looks a bit like …

You don’t need to know this - but you will see the term inode come up from time to time when reading documentation

* Processes are given references to files known as “File Descriptors”. They are just numbers, but the OS knows how to reference them (Just a hashmap!)

**What they want**

* To become an actual programmer and not a school coder
* learning what fields of software exist and being introduced to them at least on some level
  + During first lesson, show examples of what you can do in each domain - Starter projects, middle projects, and late game projects
* what I can do outside of classes to start gaining experience as a developer
* While I'm sure I will learn countless knowledge from Archer, the bigger picture is whether I'll be confident in learning and teaching myself.

Create

1. Discord bot

2. Trading bot

3. Vpn client