SLS: Part I - UNIX : Introduction & Preliminaries

Contents

Notes

- In general our goal is not to just learn a bunch of facts or commands.
- Rather it is to understand how thing work
    - developing a mental model
    - to this end we will often try and develop visualizations
- To this end we will start our exploration of UNIX by developing a way of visualizing how the layers of software and hardware relate and interact.

1.1. UNIX is an operating system

- But what is an operating system?

> A collection of software that makes a ***computer useful***.

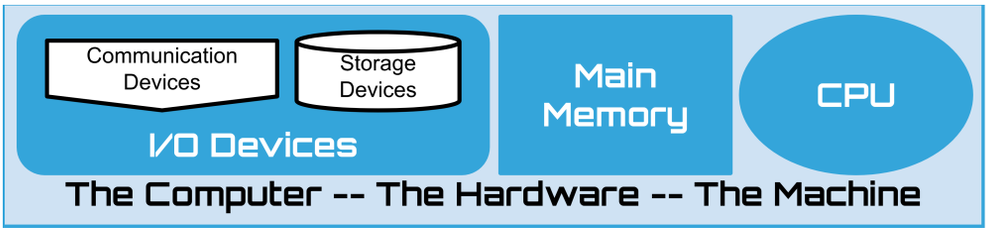
1.2. Computer?

Notes

- On the practical side computers are hardware devices that are programmable
  - we will focus on those that conform to a “Classic” model of programmable device called a von Neuman Architecture
    - today this encompasses Laptops, smart phones, tablets, smart watches, desktops, servers, supercomputers, and thousands of embedded systems around you

FIXME: Add some images

1.2.1. An Organized Collection of Hardware



Notes

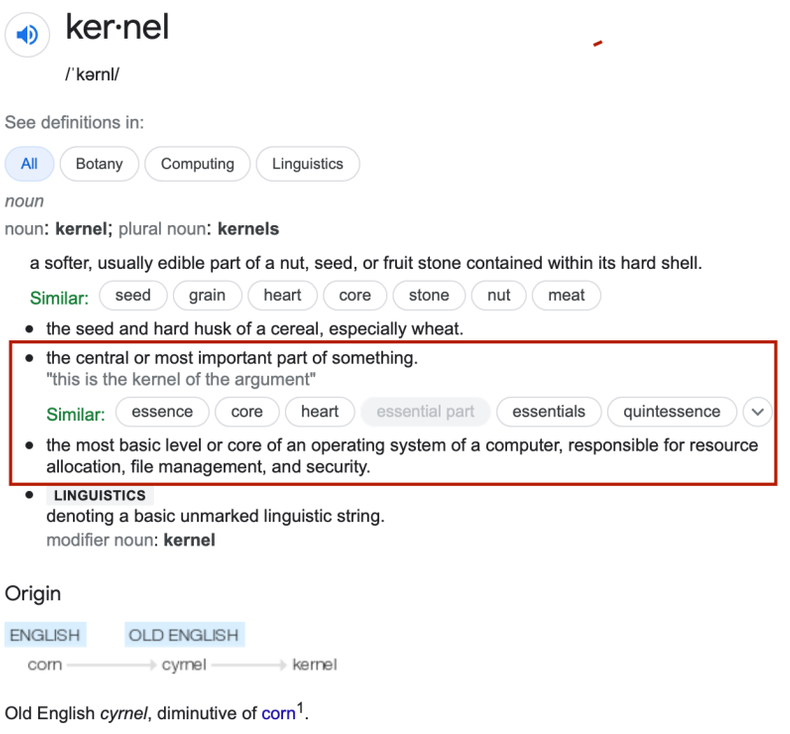
- We will learn much more later but for the moment we need a working model:
  - three core components
    - CPU
      - the smart bits that “execute” the instructions of software – performs calculations
    - Memory
      - the devices that hold the instructions and data that make up the running software
      - physically directly connected to the CPU – often referred to as RAM and ROM, main memory, etc
      - fast, power hungry. and volatile
    - I/O devices - for the moment two main categories
      - Storage devices
        - hard drives, ssd’s, flash memory, flash drives/usb sticks, etc
        - slow and large compared to main memory
        - non-volatile
      - Communication devices
        - allow connections to the outside world
        - networks, terminals, usb devices – keyboards, mice, etc.

1.3. Useful?

- Hardware is like the raw parts
  - not really useful until we add software to have it do stuff
    - aka run programs/apps
    - but what stuff
- For most people they want to run programs that allow them to use the computer to:
  - surf the web
  - read and write email
  - record, view and edit audio, pictures and video
  - compose documents
  - analyze and visualize data
  - play games
  - etc ...

- but for some of us, programmers, our primary goal is to write programs

1.3.1. The “Kernel” – Unique to Every OS



ker·nel

/ˈkərnl/

See definitions in:

All | Botany | Computing | Linguistics

noun

noun: **kernel**; plural noun: **kernels**

a softer, usually edible part of a nut, seed, or fruit stone contained within its hard shell.

Similar: seed | grain | heart | core | stone | nut | meat

- the seed and hard husk of a cereal, especially wheat.
- the central or most important part of something.
  "this is the kernel of the argument"

  Similar: essence | core | heart | essential part | essentials | quintessence
- the most basic level or core of an operating system of a computer, responsible for resource allocation, file management, and security.
- LINGUISTICS
  denoting a basic unmarked linguistic string.
  modifier noun: **kernel**

Origin

ENGLISH OLD ENGLISH

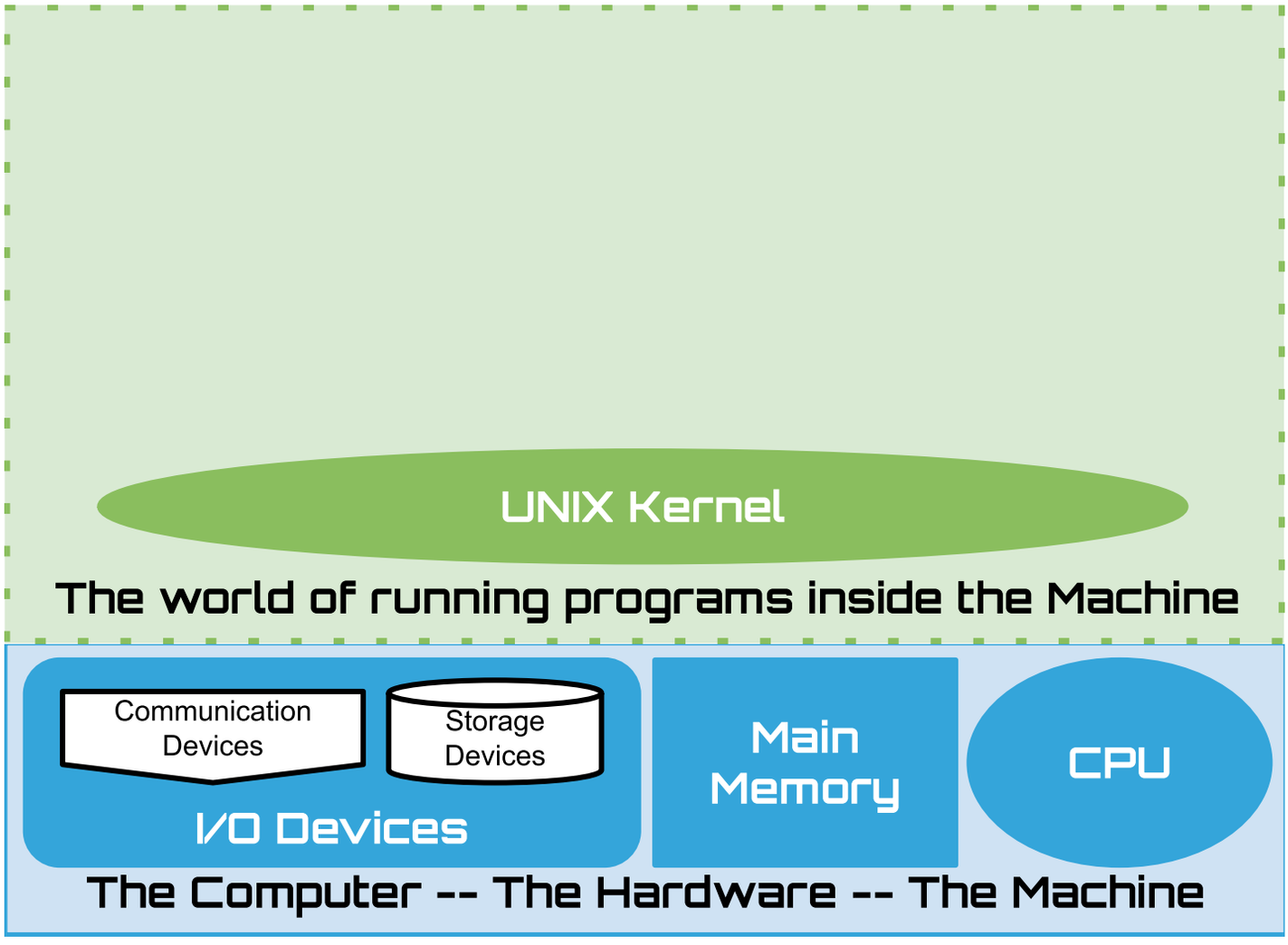
corn → cyrnel → kernel

Old English *cyrnel*, diminutive of corn[1].

1. Bootstraps the HW and has direct access to all of it
2. Bottom layer that enables other programs to run
3. A unique collection of functions that programs can invoke

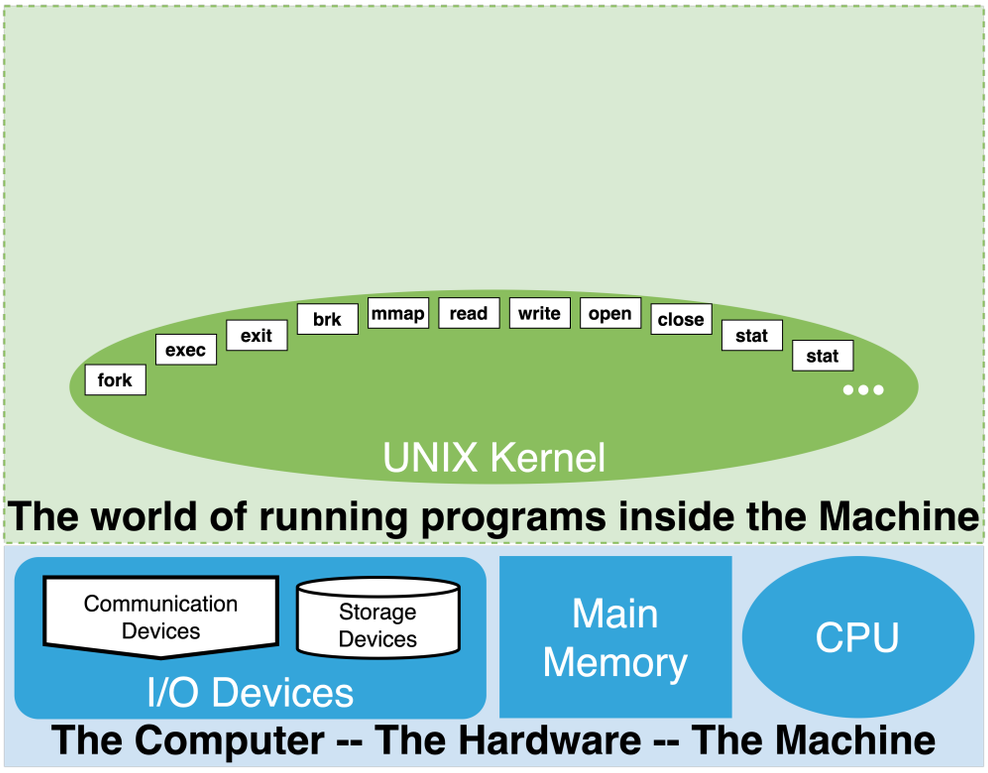
Not useful on its own only useful and accessed by running other programs.

1.4. Hardware + Kernel



- bottom layer of the software that has direct access to all the hardware
- single instance that bootstraps the hardware
- provides means for starting application/user programs
    - enables several to run at a time
    - keeps them isolated from each other
    - program can start and end
        - but kernel is always present and running
        - facilities for managing the running user programs
            - listing, pausing, terminating, etc.
- a collection of ever present functions and objects that programs/programmers can rely on
    - provides core “software” “abstraction” such as files
        - makes it easier for programs to use the hardware
        - consistent across different hardware
    - programs / programmers need not worry about the details of the hardware
- programs NOT humans interact with the kernel

1.5. System calls



1.5.1. User Programs

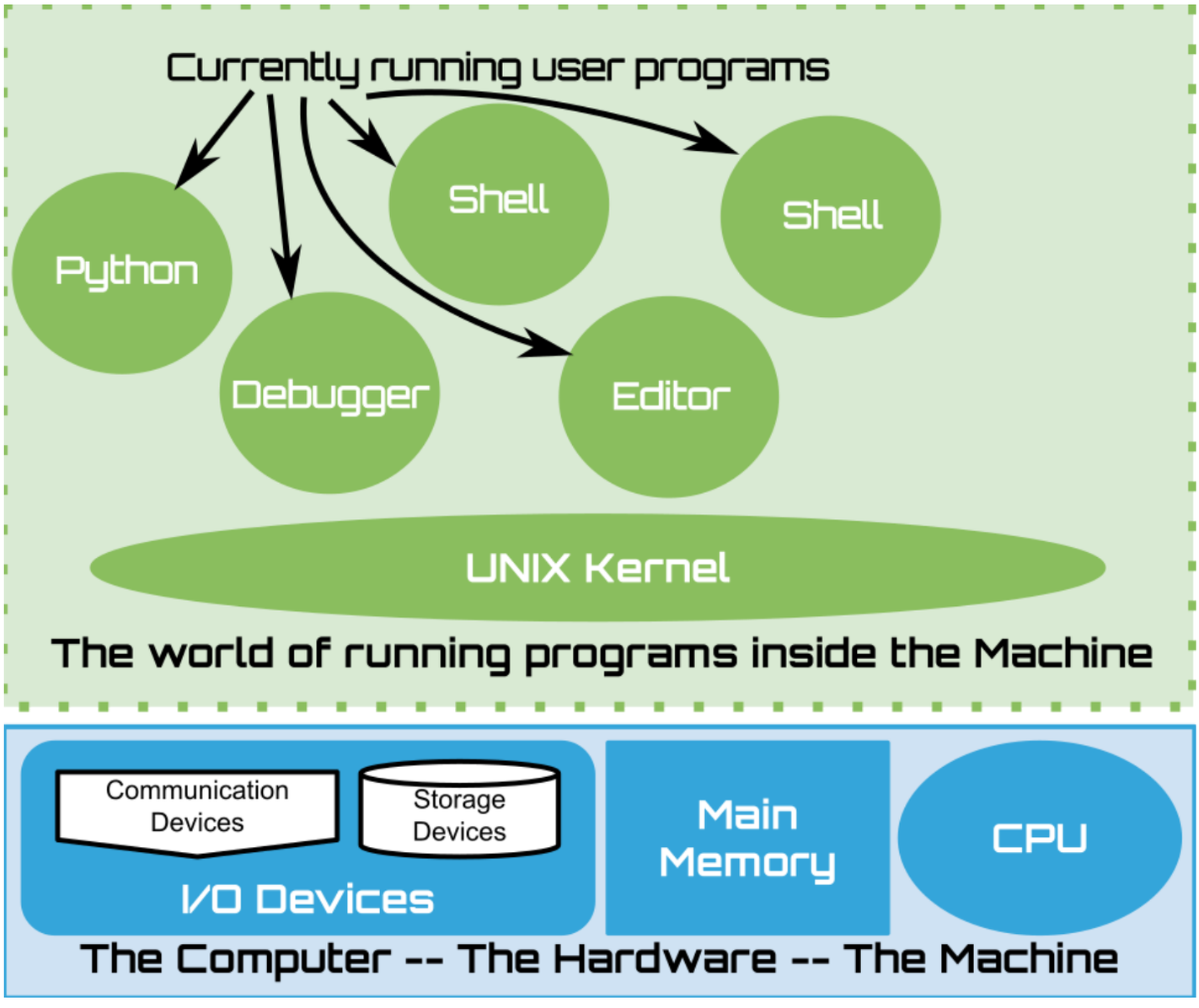
Programs that come with the Kernel – all the other stuff

1. Display / Interface Mangers
2. Finders/Explores
3. All the Other Apps/Utilities

This collection is unique to every OS and depends on its target audience.

- a collection of programs that come pre-packaged with the OS
  - one or more interface programs
    - provides a human with a way of interacting with the OS to use the other programs installed. Eg.
      - the graphical front-end of your OS aka the “desktop”, the “home” screen, the car menu system
      - text based command line
  - programs for exploring and finding information about the other programs and data installed
    - eg. The Windows File Explore, OSX Finder, etc.
  - Many other utilities depending of different categories depending on the OS
    - personal computers – media programs, web-browsers, productivity apps, etc.
    - tools for developing new programs:
      - program editors
      - programming language software: python, ruby, etc.
      - debuggers
  - The nature and feel of these programs is specific to the goals or target audience of the OS
    - Most of commodity OS’s are targeted at non-programmers
    - assumes ease of use is primary and programability is secondary

1.6. Hardware + Kernel + User Programs



1.7. Why Unix?

By programmers for programmers – teaches you to think like a programmer.

- automation is the name of the game – every aspect is programmable
- text oriented interfaces and utilities makes it easy to write programs that translate and transform information
- it is easy to access and see what other OS's purposely hide

A lot of the world around you runs on some form of Unix!

1.7.1. Benefits to understanding/learning Unix

Unix's terminal interface and program development environment became the gold standard for university CS education

- A programmer oriented model for interacting and using the computer – The Shell
- A collection of composable and extensible tools for processing ASCII documents
    - Making it easy to write new programs
        - including programs that translate ASCII documents into new programs
- While it takes some effort to learn:
    - it teaches you to think like a programmer
        - writing little re-usable programs
        - incrementally evolving those as needed
        - makes the value in decomposing and reusing existing programs obvious

- it unleashes your creativity by providing simple building blocks
  - minimizes time and effort from idea to prototype
  - provide a environment where everything can be customized and programmed
  - automation is the name of the game
- it only has a few small core ideas that you need to understand to get going
  - files, processes, I/O redirection
- Rich body of existing software for program development
- The computers of the Internet and Cloud primarily runs some form of Unix
- The embedded systems around you often run Unix
- Foundation of Open Source software is Unix based – ala Linux

---

1.8. UNIX Preliminaries

1. Files and Directories
2. ASCII
3. The Terminal and the Shell

1.8.1. Files and directories (folders)?

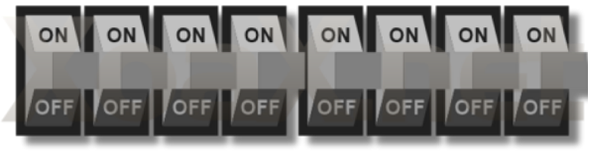
OS converts HW into an information management and processing system

- primitive for representing information: The File
- primitive for organize the information: Hierarchy of Directories

---

1.8.2. Files - But first the Byte and some notation

- The byte
  - basic unit of information the hardware can store and manipulate



- A byte: 8 switches that form a single location of memory/storage
- We measure capacity or size of memory/storage in units of bytes
- We think of each byte of memory or data as a vector of 8 bits

$[b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0]$ where each b_i is $0, 1$

ALL OFF	0	0	0	0	0	0	0	0
ALL ON	1 $[b_7$	1 b_6	1 b_5	1 b_4	1 b_3	1 b_2	1 b_1	1 $b_0]$

- A byte : can take on 256 unique values – $2^8 = 256$ possible values

---

1.9. Hexadecimal

- often times we see base 16 (hexadecimal) used to express a particular binary byte value
    - it is easier to write down 2 base hex digits than 8 binary digits

We prefix a hex value with `0x` to distinguish base 16 values (eg. `0x11`)

And we use `0b` to distinguish base 2 value (eg. `0b11`).

If we don't prefix we will assume it is obvious from context or we are assuming base 10.

---

1 $[b_7$	0 b_6	1 b_5	0 b_4	1 b_3	1 b_2	1 b_1	0 $b_0]$

0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A	1	0	1	0
B	1	0	1	1
C	1	1	0	0
D	1	1	0	1
E	1	1	1	0
F HEX	1 [b_3	1 b_2	1 b_1	1 b_0]

1 [b_7	0 b_6	1 b_5	0 b_4]

1 [b_3	1 b_2	1 b_1	0 b_0]

A

E

0b10101110 = 0xAE

In base 16 (hexadecimal) each digit has 16 possible values. The following table shows how we map all 16 possible patterns of 4 bits to a single hex digit. This allows us to easily express any byte value as two hex digits. We simply rewrite the two groups of 4 bits of a byte as two hex digits. Use chalkboard to illustrate

- `bin2Hex(0b00010000)` : 0b00010000 → 0x10
- `bin2Hex(0b10000001)` : 0b10000001 → 0x81
- `bin2Hex(0b10111001)` : 0b10111001 → 0xb9
- `bin2Hex(0b10101010)` : 0b10101010 → 0xaa
- `bin2Hex(0b01010101)` : 0b01010101 → 0x55
- `bin2Hex(0b11110111)` : 0b11110111 → 0xf7

- 0b00010000 → 0x ?
- 0b10000001 → 0x ?
- 0b10111001 → 0x ?
- 0b10101010 → 0x ?
- 0b01010101 → 0x ?
- 0b11110111 → 0x ?

0x00 (000)	0	0	0	0	0	0	0	0
0x01 (001)	0	0	0	0	0	0	0	1
0x02 (002)	0	0	0	0	0	0	1	0
0x03 (003)	0	0	0	0	0	0	1	1
0x04 (004)	0	0	0	0	0	1	0	0
0x05 (005)	0	0	0	0	0	1	0	1
0x06 (006)	0	0	0	0	0	1	1	0
0x07 (007)	0	0	0	0	0	1	1	1
0x08 (008)	0	0	0	0	1	0	0	0
0x09 (009)	0	0	0	0	1	0	0	1
0x0a (010)	0	0	0	0	1	0	1	0
0x0b (011)	0	0	0	0	1	0	1	1
0x0c (012)	0	0	0	0	1	1	0	0
0x0d (013)	0	0	0	0	1	1	0	1
0x0e (014)	0	0	0	0	1	1	1	0
0x0f (015)	0	0	0	0	1	1	1	1
0x10 (016)	0	0	0	1	0	0	0	0
0x11 (017)	0	0	0	1	0	0	0	1
0x12 (018)	0	0	0	1	0	0	1	0
0x13 (019)	0	0	0	1	0	0	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x14 (020)	0	0	0	1	0	1	0	0
0x15 (021)	0	0	0	1	0	1	0	1
0x16 (022)	0	0	0	1	0	1	1	0
0x17 (023)	0	0	0	1	0	1	1	1
0x18 (024)	0	0	0	1	1	0	0	0
0x19 (025)	0	0	0	1	1	0	0	1
0x1a (026)	0	0	0	1	1	0	1	0
0x1b (027)	0	0	0	1	1	0	1	1
0x1c (028)	0	0	0	1	1	1	0	0
0x1d (029)	0	0	0	1	1	1	0	1
0x1e (030)	0	0	0	1	1	1	1	0
0x1f (031)	0	0	0	1	1	1	1	1
0x20 (032)	0	0	1	0	0	0	0	0
0x21 (033)	0	0	1	0	0	0	0	1
0x22 (034)	0	0	1	0	0	0	1	0
0x23 (035)	0	0	1	0	0	0	1	1
0x24 (036)	0	0	1	0	0	1	0	0
0x25 (037)	0	0	1	0	0	1	0	1
0x26 (038)	0	0	1	0	0	1	1	0
0x27 (039)	0	0	1	0	0	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x28 (040)	0	0	1	0	1	0	0	0
0x29 (041)	0	0	1	0	1	0	0	1
0x2a (042)	0	0	1	0	1	0	1	0
0x2b (043)	0	0	1	0	1	0	1	1
0x2c (044)	0	0	1	0	1	1	0	0
0x2d (045)	0	0	1	0	1	1	0	1
0x2e (046)	0	0	1	0	1	1	1	0
0x2f (047)	0	0	1	0	1	1	1	1
0x30 (048)	0	0	1	1	0	0	0	0
0x31 (049)	0	0	1	1	0	0	0	1
0x32 (050)	0	0	1	1	0	0	1	0
0x33 (051)	0	0	1	1	0	0	1	1
0x34 (052)	0	0	1	1	0	1	0	0
0x35 (053)	0	0	1	1	0	1	0	1
0x36 (054)	0	0	1	1	0	1	1	0
0x37 (055)	0	0	1	1	0	1	1	1
0x38 (056)	0	0	1	1	1	0	0	0
0x39 (057)	0	0	1	1	1	0	0	1
0x3a (058)	0	0	1	1	1	0	1	0
0x3b (059)	0	0	1	1	1	0	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x3c (060)	0	0	1	1	1	1	0	0
0x3d (061)	0	0	1	1	1	1	0	1
0x3e (062)	0	0	1	1	1	1	1	0
0x3f (063)	0	0	1	1	1	1	1	1
0x40 (064)	0	1	0	0	0	0	0	0
0x41 (065)	0	1	0	0	0	0	0	1
0x42 (066)	0	1	0	0	0	0	1	0
0x43 (067)	0	1	0	0	0	0	1	1
0x44 (068)	0	1	0	0	0	1	0	0
0x45 (069)	0	1	0	0	0	1	0	1
0x46 (070)	0	1	0	0	0	1	1	0
0x47 (071)	0	1	0	0	0	1	1	1
0x48 (072)	0	1	0	0	1	0	0	0
0x49 (073)	0	1	0	0	1	0	0	1
0x4a (074)	0	1	0	0	1	0	1	0
0x4b (075)	0	1	0	0	1	0	1	1
0x4c (076)	0	1	0	0	1	1	0	0
0x4d (077)	0	1	0	0	1	1	0	1
0x4e (078)	0	1	0	0	1	1	1	0
0x4f (079)	0	1	0	0	1	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x50 (080)	0	1	0	1	0	0	0	0
0x51 (081)	0	1	0	1	0	0	0	1
0x52 (082)	0	1	0	1	0	0	1	0
0x53 (083)	0	1	0	1	0	0	1	1
0x54 (084)	0	1	0	1	0	1	0	0
0x55 (085)	0	1	0	1	0	1	0	1
0x56 (086)	0	1	0	1	0	1	1	0
0x57 (087)	0	1	0	1	0	1	1	1
0x58 (088)	0	1	0	1	1	0	0	0
0x59 (089)	0	1	0	1	1	0	0	1
0x5a (090)	0	1	0	1	1	0	1	0
0x5b (091)	0	1	0	1	1	0	1	1
0x5c (092)	0	1	0	1	1	1	0	0
0x5d (093)	0	1	0	1	1	1	0	1
0x5e (094)	0	1	0	1	1	1	1	0
0x5f (095)	0	1	0	1	1	1	1	1
0x60 (096)	0	1	1	0	0	0	0	0
0x61 (097)	0	1	1	0	0	0	0	1
0x62 (098)	0	1	1	0	0	0	1	0
0x63 (099)	0	1	1	0	0	0	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x64 (100)	0	1	1	0	0	1	0	0
0x65 (101)	0	1	1	0	0	1	0	1
0x66 (102)	0	1	1	0	0	1	1	0
0x67 (103)	0	1	1	0	0	1	1	1
0x68 (104)	0	1	1	0	1	0	0	0
0x69 (105)	0	1	1	0	1	0	0	1
0x6a (106)	0	1	1	0	1	0	1	0
0x6b (107)	0	1	1	0	1	0	1	1
0x6c (108)	0	1	1	0	1	1	0	0
0x6d (109)	0	1	1	0	1	1	0	1
0x6e (110)	0	1	1	0	1	1	1	0
0x6f (111)	0	1	1	0	1	1	1	1
0x70 (112)	0	1	1	1	0	0	0	0
0x71 (113)	0	1	1	1	0	0	0	1
0x72 (114)	0	1	1	1	0	0	1	0
0x73 (115)	0	1	1	1	0	0	1	1
0x74 (116)	0	1	1	1	0	1	0	0
0x75 (117)	0	1	1	1	0	1	0	1
0x76 (118)	0	1	1	1	0	1	1	0
0x77 (119)	0	1	1	1	0	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x78 (120)	0	1	1	1	1	0	0	0
0x79 (121)	0	1	1	1	1	0	0	1
0x7a (122)	0	1	1	1	1	0	1	0
0x7b (123)	0	1	1	1	1	0	1	1
0x7c (124)	0	1	1	1	1	1	0	0
0x7d (125)	0	1	1	1	1	1	0	1
0x7e (126)	0	1	1	1	1	1	1	0
0x7f (127)	0	1	1	1	1	1	1	1
0x80 (128)	1	0	0	0	0	0	0	0
0x81 (129)	1	0	0	0	0	0	0	1
0x82 (130)	1	0	0	0	0	0	1	0
0x83 (131)	1	0	0	0	0	0	1	1
0x84 (132)	1	0	0	0	0	1	0	0
0x85 (133)	1	0	0	0	0	1	0	1
0x86 (134)	1	0	0	0	0	1	1	0
0x87 (135)	1	0	0	0	0	1	1	1
0x88 (136)	1	0	0	0	1	0	0	0
0x89 (137)	1	0	0	0	1	0	0	1
0x8a (138)	1	0	0	0	1	0	1	0
0x8b (139)	1	0	0	0	1	0	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0x8c (140)	1	0	0	0	1	1	0	0
0x8d (141)	1	0	0	0	1	1	0	1
0x8e (142)	1	0	0	0	1	1	1	0
0x8f (143)	1	0	0	0	1	1	1	1
0x90 (144)	1	0	0	1	0	0	0	0
0x91 (145)	1	0	0	1	0	0	0	1
0x92 (146)	1	0	0	1	0	0	1	0
0x93 (147)	1	0	0	1	0	0	1	1
0x94 (148)	1	0	0	1	0	1	0	0
0x95 (149)	1	0	0	1	0	1	0	1
0x96 (150)	1	0	0	1	0	1	1	0
0x97 (151)	1	0	0	1	0	1	1	1
0x98 (152)	1	0	0	1	1	0	0	0
0x99 (153)	1	0	0	1	1	0	0	1
0x9a (154)	1	0	0	1	1	0	1	0
0x9b (155)	1	0	0	1	1	0	1	1
0x9c (156)	1	0	0	1	1	1	0	0
0x9d (157)	1	0	0	1	1	1	0	1
0x9e (158)	1	0	0	1	1	1	1	0
0x9f (159)	1	0	0	1	1	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0xa0 (160)	1	0	1	0	0	0	0	0
0xa1 (161)	1	0	1	0	0	0	0	1
0xa2 (162)	1	0	1	0	0	0	1	0
0xa3 (163)	1	0	1	0	0	0	1	1
0xa4 (164)	1	0	1	0	0	1	0	0
0xa5 (165)	1	0	1	0	0	1	0	1
0xa6 (166)	1	0	1	0	0	1	1	0
0xa7 (167)	1	0	1	0	0	1	1	1
0xa8 (168)	1	0	1	0	1	0	0	0
0xa9 (169)	1	0	1	0	1	0	0	1
0xaa (170)	1	0	1	0	1	0	1	0
0xab (171)	1	0	1	0	1	0	1	1
0xac (172)	1	0	1	0	1	1	0	0
0xad (173)	1	0	1	0	1	1	0	1
0xae (174)	1	0	1	0	1	1	1	0
0xaf (175)	1	0	1	0	1	1	1	1
0xb0 (176)	1	0	1	1	0	0	0	0
0xb1 (177)	1	0	1	1	0	0	0	1
0xb2 (178)	1	0	1	1	0	0	1	0
0xb3 (179)	1	0	1	1	0	0	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0xb4 (180)	1	0	1	1	0	1	0	0
0xb5 (181)	1	0	1	1	0	1	0	1
0xb6 (182)	1	0	1	1	0	1	1	0
0xb7 (183)	1	0	1	1	0	1	1	1
0xb8 (184)	1	0	1	1	1	0	0	0
0xb9 (185)	1	0	1	1	1	0	0	1
0xba (186)	1	0	1	1	1	0	1	0
0xbb (187)	1	0	1	1	1	0	1	1
0xbc (188)	1	0	1	1	1	1	0	0
0xbd (189)	1	0	1	1	1	1	0	1
0xbe (190)	1	0	1	1	1	1	1	0
0xbf (191)	1	0	1	1	1	1	1	1
0xc0 (192)	1	1	0	0	0	0	0	0
0xc1 (193)	1	1	0	0	0	0	0	1
0xc2 (194)	1	1	0	0	0	0	1	0
0xc3 (195)	1	1	0	0	0	0	1	1
0xc4 (196)	1	1	0	0	0	1	0	0
0xc5 (197)	1	1	0	0	0	1	0	1
0xc6 (198)	1	1	0	0	0	1	1	0
0xc7 (199)	1	1	0	0	0	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0xc8 (200)	1	1	0	0	1	0	0	0
0xc9 (201)	1	1	0	0	1	0	0	1
0xca (202)	1	1	0	0	1	0	1	0
0xcb (203)	1	1	0	0	1	0	1	1
0xcc (204)	1	1	0	0	1	1	0	0
0xcd (205)	1	1	0	0	1	1	0	1
0xce (206)	1	1	0	0	1	1	1	0
0xcf (207)	1	1	0	0	1	1	1	1
0xd0 (208)	1	1	0	1	0	0	0	0
0xd1 (209)	1	1	0	1	0	0	0	1
0xd2 (210)	1	1	0	1	0	0	1	0
0xd3 (211)	1	1	0	1	0	0	1	1
0xd4 (212)	1	1	0	1	0	1	0	0
0xd5 (213)	1	1	0	1	0	1	0	1
0xd6 (214)	1	1	0	1	0	1	1	0
0xd7 (215)	1	1	0	1	0	1	1	1
0xd8 (216)	1	1	0	1	1	0	0	0
0xd9 (217)	1	1	0	1	1	0	0	1
0xda (218)	1	1	0	1	1	0	1	0
0xdb (219)	1	1	0	1	1	0	1	1

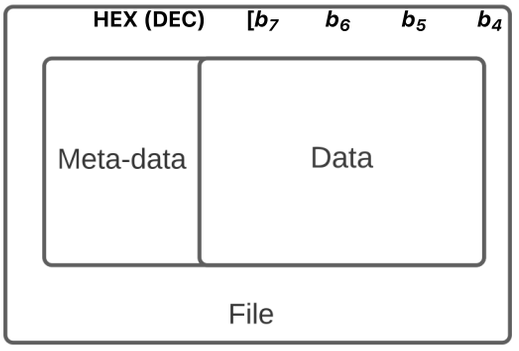
HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0xdc (220)	1	1	0	1	1	1	0	0
0xdd (221)	1	1	0	1	1	1	0	1
0xde (222)	1	1	0	1	1	1	1	0
0xdf (223)	1	1	0	1	1	1	1	1
0xe0 (224)	1	1	1	0	0	0	0	0
0xe1 (225)	1	1	1	0	0	0	0	1
0xe2 (226)	1	1	1	0	0	0	1	0
0xe3 (227)	1	1	1	0	0	0	1	1
0xe4 (228)	1	1	1	0	0	1	0	0
0xe5 (229)	1	1	1	0	0	1	0	1
0xe6 (230)	1	1	1	0	0	1	1	0
0xe7 (231)	1	1	1	0	0	1	1	1
0xe8 (232)	1	1	1	0	1	0	0	0
0xe9 (233)	1	1	1	0	1	0	0	1
0xea (234)	1	1	1	0	1	0	1	0
0xeb (235)	1	1	1	0	1	0	1	1
0xec (236)	1	1	1	0	1	1	0	0
0xed (237)	1	1	1	0	1	1	0	1
0xee (238)	1	1	1	0	1	1	1	0
0xef (239)	1	1	1	0	1	1	1	1

HEX (DEC)	[b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0]
0xf0 (240)	1	1	1	1	0	0	0	0
0xf1 (241)	1	1	1	1	0	0	0	1
0xf2 (242)	1	1	1	1	0	0	1	0
0xf3 (243)	1	1	1	1	0	0	1	1
0xf4 (244)	1	1	1	1	0	1	0	0
0xf5 (245)	1	1	1	1	0	1	0	1
0xf6 (246)	1	1	1	1	0	1	1	0
0xf7 (247)	1	1	1	1	0	1	1	1
0xf8 (248)	1	1	1	1	1	0	0	0
0xf9 (249)	1	1	1	1	1	0	0	1
0xfa (250)	1	1	1	1	1	0	1	0
0xfb (251)	1	1	1	1	1	0	1	1
0xfc (252)	1	1	1	1	1	1	0	0
0xfd (253)	1	1	1	1	1	1	0	1
0xfe (254)	1	1	1	1	1	1	1	0
0xff (255)	1	1	1	1	1	1	1	1

This table shows all 256 patterns of bits that a single byte could have. On the left is how we would refer to the value in both Hex and (decimal)

1.9.1. Files - Data

1.9.2. The information a file contains is its data



A file is an abstract object that can store Data and has associated with it additional descriptive facts we call Meta-data

Data: is a collection of bytes:

In Unix there is no explicit type that tells us what the bytes "are"

- they could be human readable program code
- they could be pixel data of an image

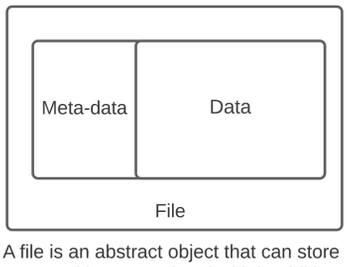
We are free to try and interpret the bytes in any way we like depending on what programs we use to process them.

UNIX assumes we know what we are doing!

There are many tools that let us just work with raw bytes so we can always use these tools with any file! (eg dump its data as hexadecimal values)

1.9.3. Files - Meta Data

1.9.4. Information describing other properties of the file.

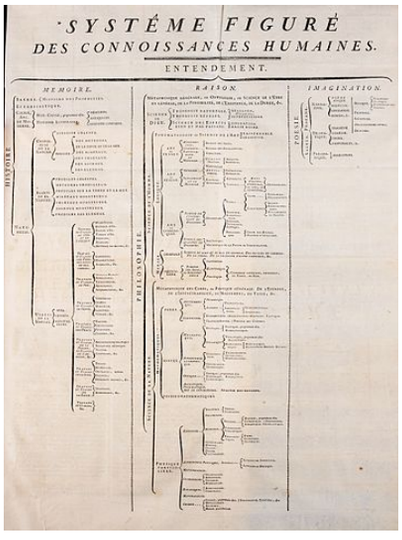


A file is an abstract object that can store Data and has associated with it additional descriptive facts we call Meta-data

Meta-data: Bytes that encode extra descriptive facts:

- who owns the file
- the length of the file (measured in bytes)
- who has permissions to read or write its contents
- the time the contents was last modified
- the time that it was last read
- the time that the descriptive facts where last changed (eg the file permissions were modified)

The meta data is key to use being able to control access to the data of a file.



1.10. Hierarchy – Trees and Directories

Files are a good start but not enough to stay organized

Must have a way for naming and finding files that flexible and can be easily understood and organized by the users

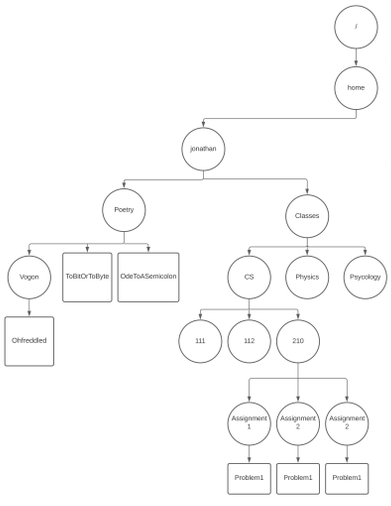
Directory: a list of names each name identifies either a single file or another directory. Entries of this are *in* the directory.

Sub-directory: contained with a parent directory

Like files directories have meta-data but the data is just its list of entries. Results in a tree and the name of any file is a path within the tree.

Users are empowered to organized their files as they see fit both in names and directory structure.

1.10.1. PATHS and the ROOT



`/ + home + jonathan + Classes + CS + 210 + Assignment 1 + Problem 1`

`/ + home + jonathan + Classes + CS + 210 + Assignment 2 + Problem 1`

`/ + home + jonathan + Classes + CS + 210 + Assignment 3 + Problem 1`

"Full" path name: join independent components with the "/" character. So the above three files as proper unix path names would be:

`/home/jonathan/Classes/CS/210/Assignment 1/Problem 1 /home/jonathan/Classes/CS/210/Assignment 2/Problem 1 /home/jonathan/Classes/CS/210/Assignment 3/Problem 1`

- Talk through the example draw as necessary to highlight idea of a path
- note the notion of root as an anchor
- later we will get to relative paths and lack of constraints on file names

---

1.10.2. ASCII - One more thing

- An important encoding of bytes that many Unix tools assume is ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

- ASCII provides a simple code that maps bytes to English letters, numbers, punctuation, and some text orient controls.
- The "base" encoding that programs use.

1.11. Terminal and the Shell - Here we go

Finally ready to get on with it

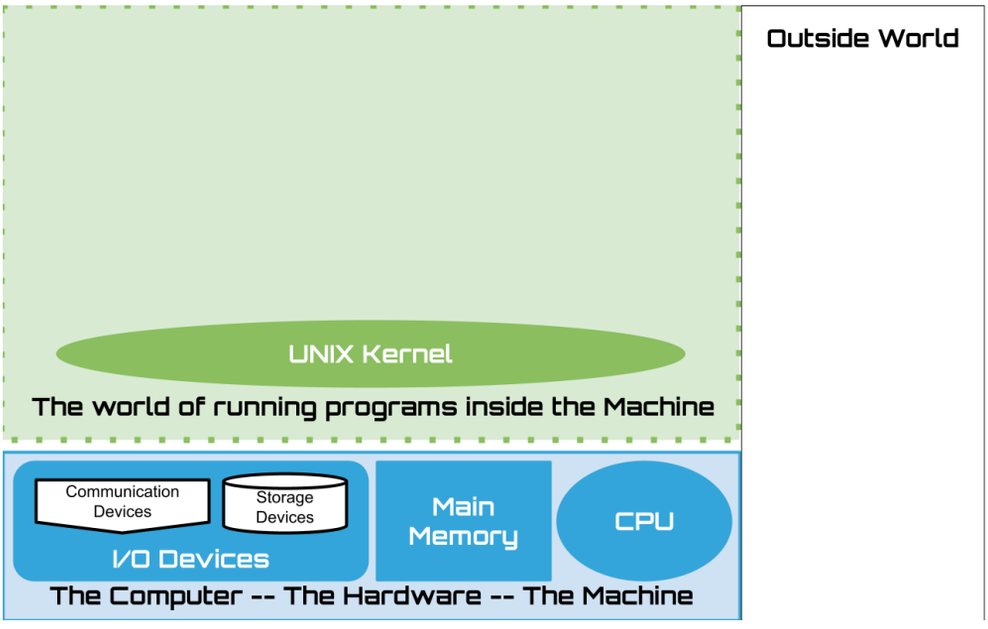
- a computer - the hardware
- with the Unix kernel booted and running on it waiting for us!

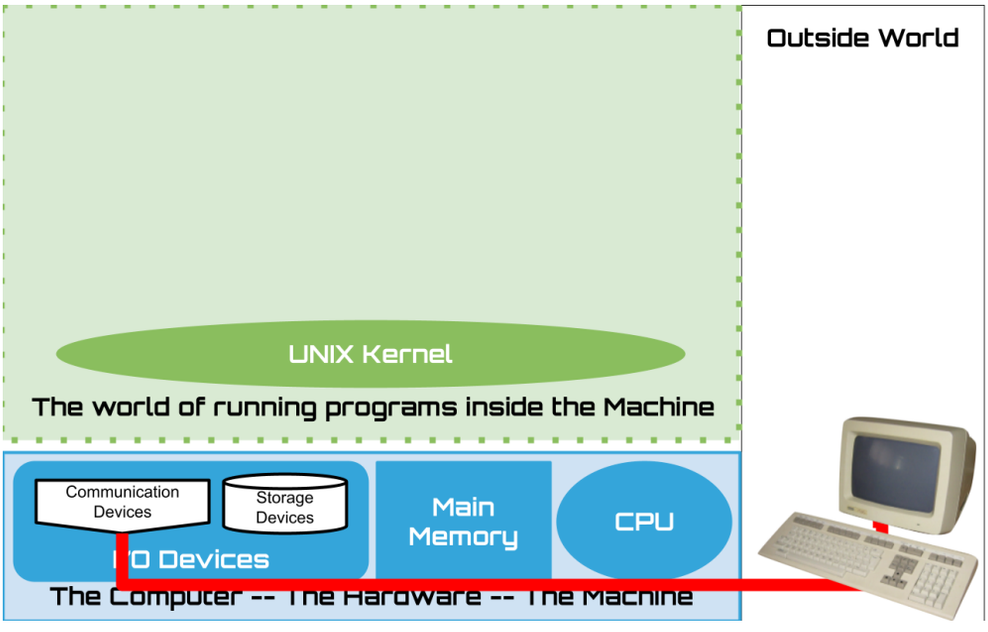
1.11.1. Shell: One program to "rule/run" them all

- Every story has to start somewhere

- every new terminal connection made to the UNIX system starts a shell and directs its input and output to the the terminal
- Shell is designed to have a text based conversation with a programmer –
  - the conversation is exchanges between the programmer and the shell - eg.

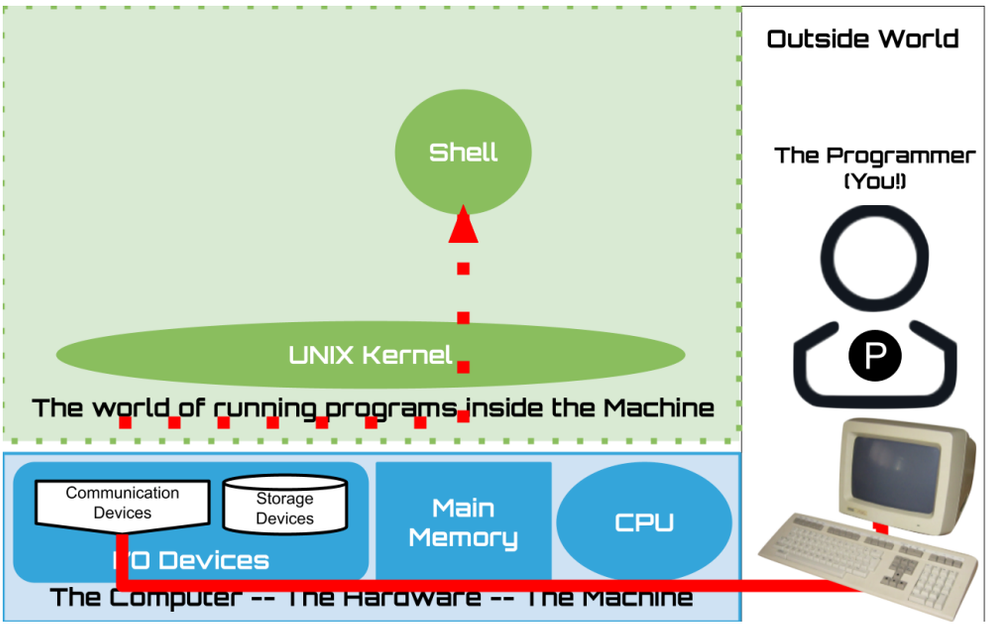
```
SHLL -> READY
PGMR -> <enter> '\n'
SHLL -> READY
SHLL -> #<enter>
SHLL -> READY
PGMR -> echo "Hello"
SHLL -> Hello
SHLL -> READY
PGMR -> for ((i=0; i<3; i++)); do echo $i; done
SHLL -> 0
SHLL -> 1
SHLL -> 2
SHLL -> READY
PGMR -> Bye
SHLL EXITS AND KERNELENDS THE TERMINAL SESSION

      - the shell begins the conversation by send a prompt "string" to the
terminal
      - the programmer sends requests to the shell in a "line oriented" way
          - A line is a set of ASCII characters terminated by a single "newline"
character
              - '10', '0x0a', '\n'
              - the terminal sends this character when the programmer presses the
"return" or "enter" key

  - Internal vs External command
```

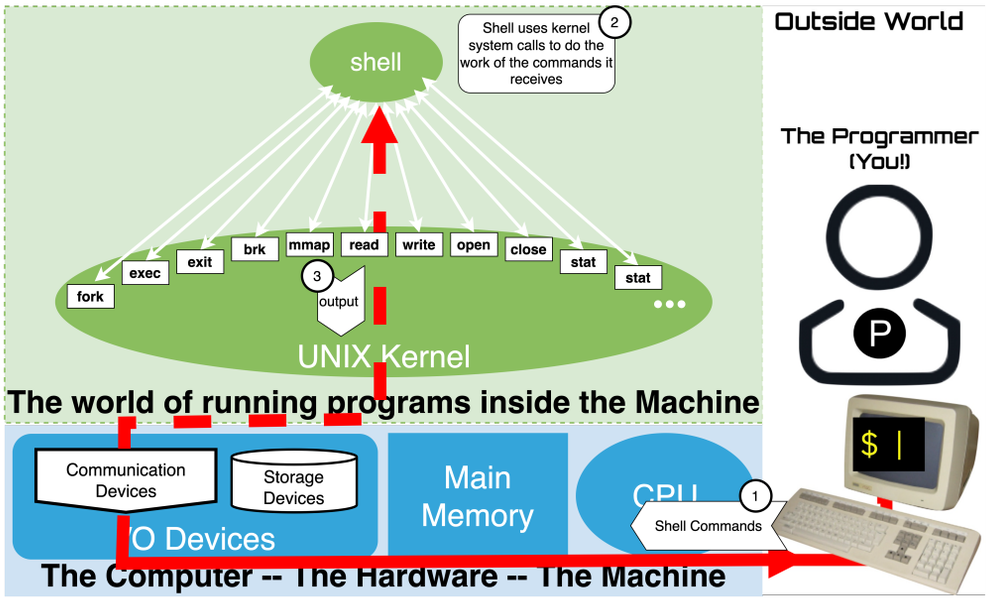




1.11.2. Connect a Terminal

- Connect a terminal so that the system can communicate with a user via ASCII data
- Unix kernel awaits for a user to login on the terminal





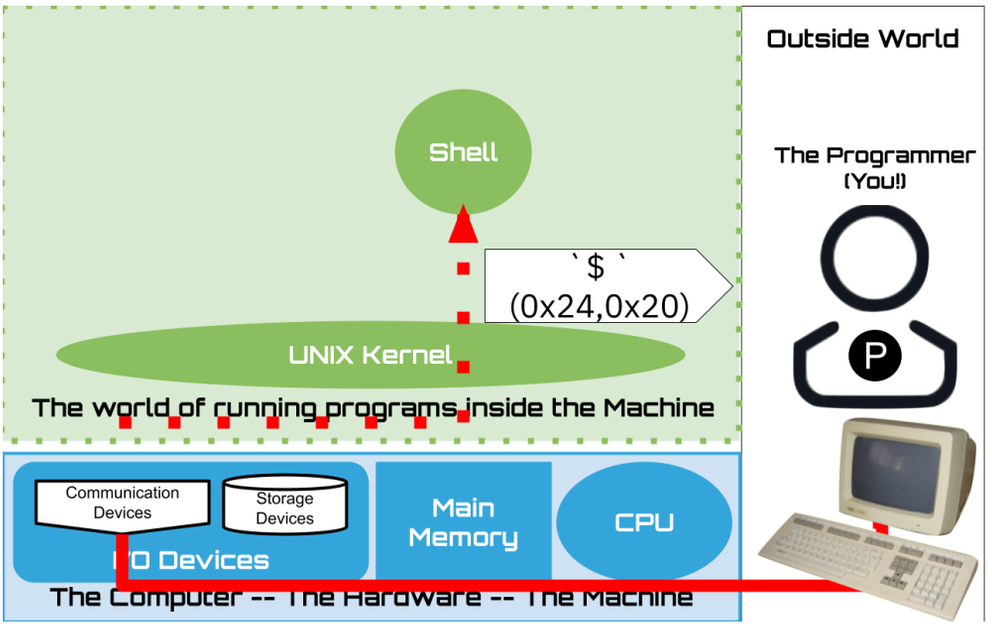
1.11.3. Start a Shell

- Terminal prints/displays ASCII sent
  - ASCII Byte → character image on screen
- User presses keys terminals sends ASCII byte
  - key press → ASCII Byte
- After Login
  - Kernel Starts users specified shell
- The Kernel will ensure that:
  1. Shell output → Terminal
  2. Terminal → Shell

- Terminal prints ASCII sent to on the screen as appropriate images of the corresponding characters
- User presses keys on the keyboard to have the Terminal send the corresponding ASCII values to the Kernel
- After Login the kernel launches an instance of the user's chosen shell program

- Default on Linux is Bash (Bourne Again Shell), however, there are others (eg. csh, ksh, zsh, sh, ...)
- The Kernel will ensure that:
    1. Shell's output will be sent to the Terminal
    2. and values sent by the Terminal will be available to the Shell as input

---



1.12. Shell Conversation

Exchanges:

1. Request – Command **line** sent by the user
2. Reply – Response sent by shell

Shell Loop:

```
while true:
    Wait for a command "line"
    Process line sending output as reply
```

Newlines `\n` – `0xa` mark end-of-line

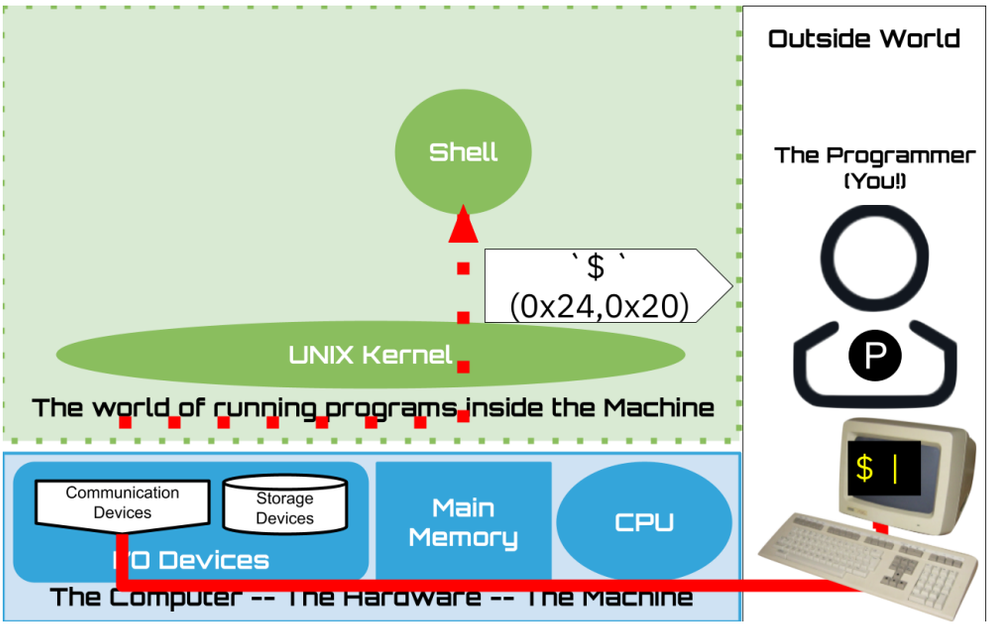
```
hello\n
```

- The shell is an ASCII line oriented interface program
- Communication is structured as a conversation between the user and the shell
    1. Request – Command **line** sent by the user
    2. Reply – Response sent by shell

```
while true:
    Wait for a command "line"
    Process line sending output as reply
```

Lines are a sequence of ASCII characters terminated by a `\n` – `0xa` eg.

```
hello\n
```

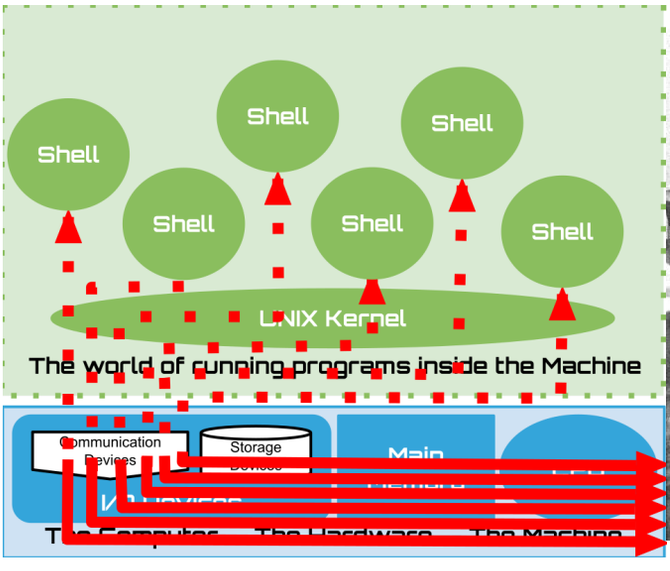


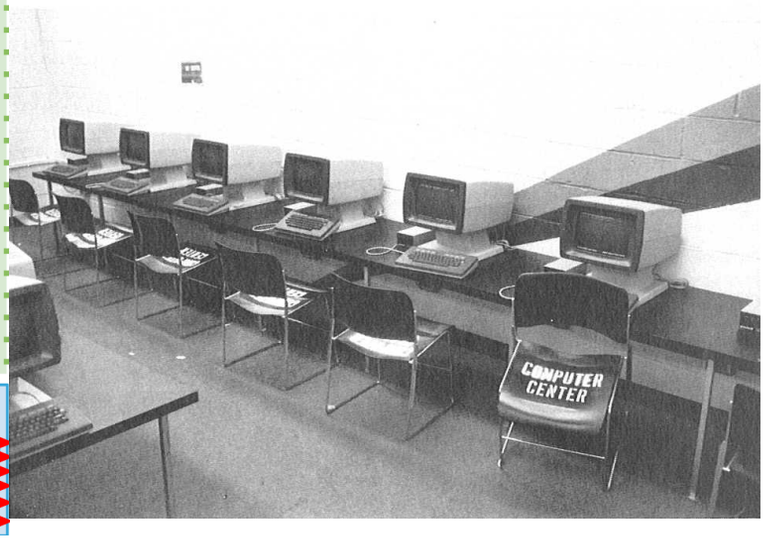
1.13. The Prompt

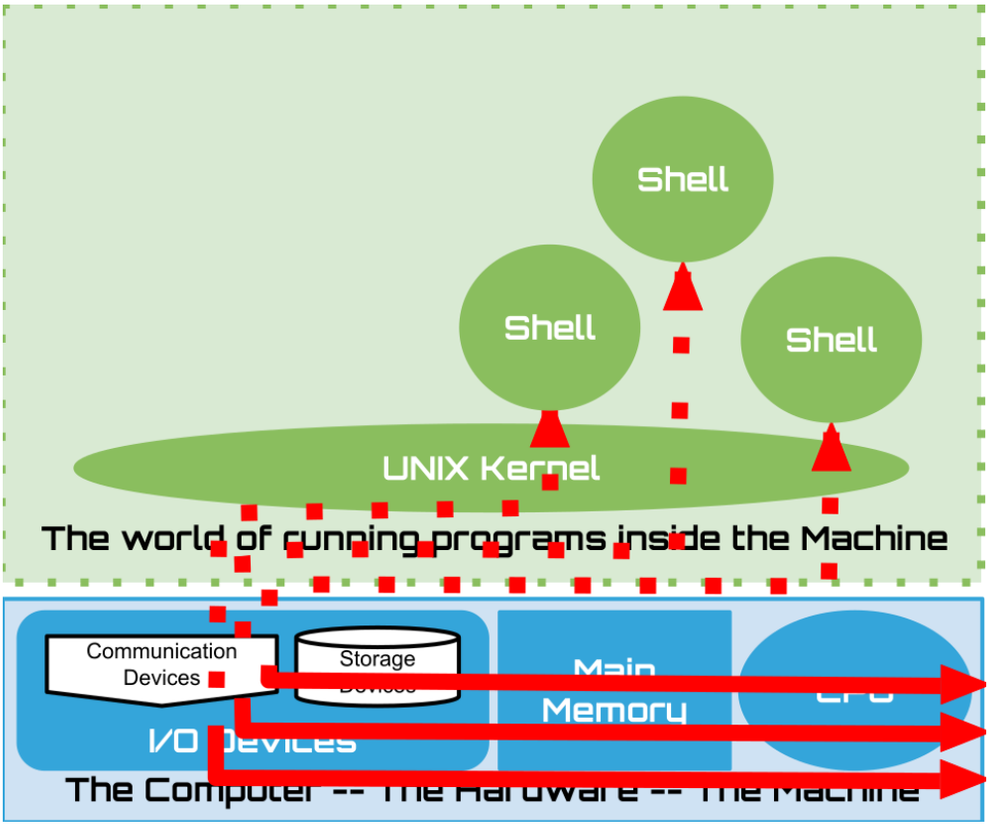
- The prompt a string of characters that the shell sends when it is ready for input
- It is the responsibility of the user to recognize that the shell is Ready for the their next request.
- Here we assume the Prompt is set to be dollar sign followed by a space `$ ` or there byte values in hex `0x24,0x20`
    - We will see later how you can customize the prompt string to your liking

---

1.14. Terminals vs Terminal Emulators





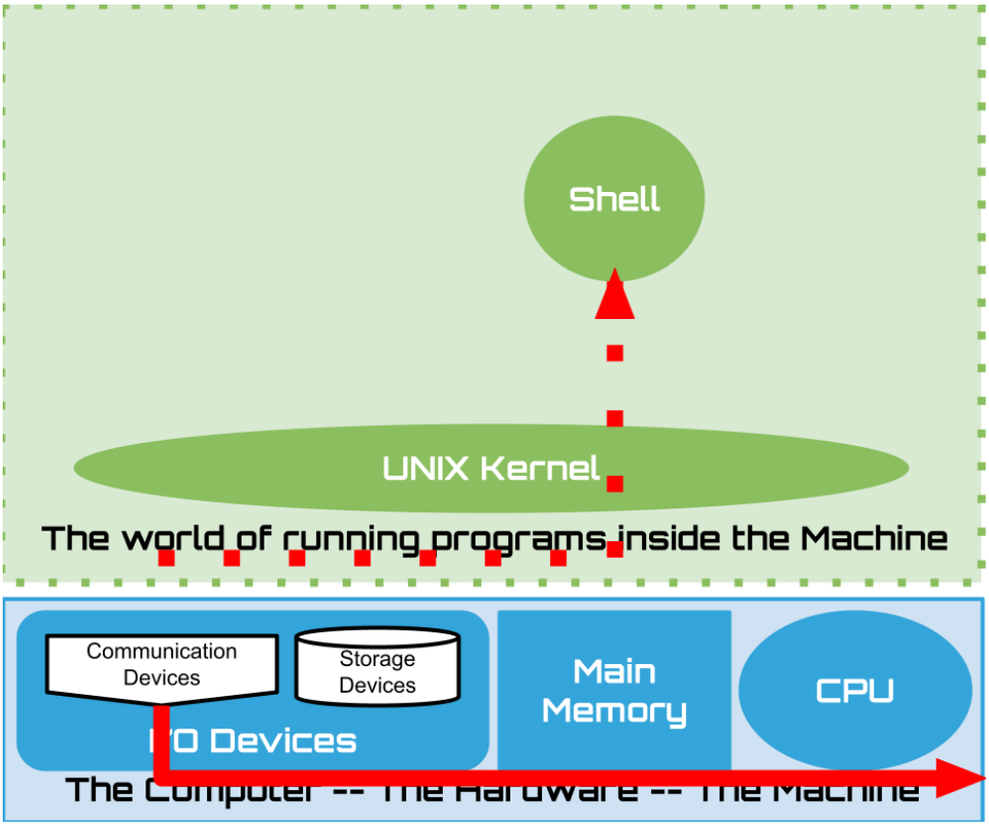


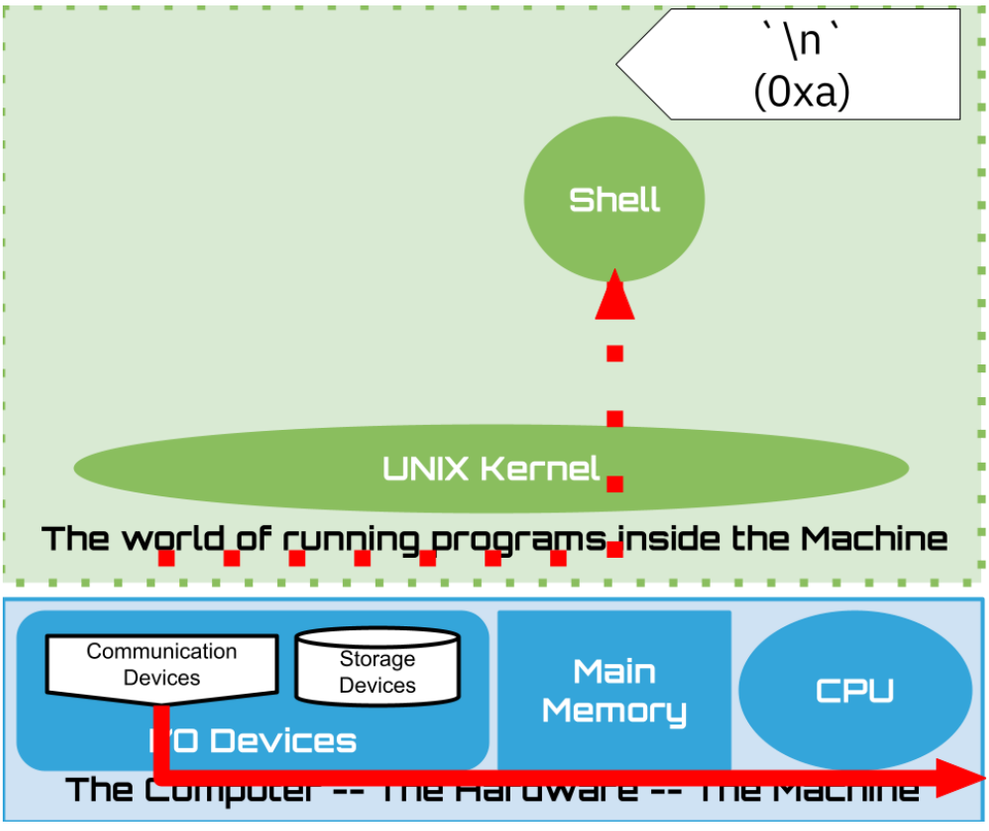
- Today a single user can create many “terminal” connections to organize their work. Eg.
    1. One to run arbitrary shell commands
    2. One to run an ascii text editor
    3. One to run an ascii email client
- A Terminal is now just a program called a Terminal Emulator:
    - You can run one to create a terminal “session/connection” on your personal computers:
        - “Terminal” App on OSX
        - “Windows Terminal” on Windows
        - “xterm” on Linux
        - Or as in our case xterm.js which lets us run terminals within a web-browser (to the right) – nice thing is then that we don’t need any extra software

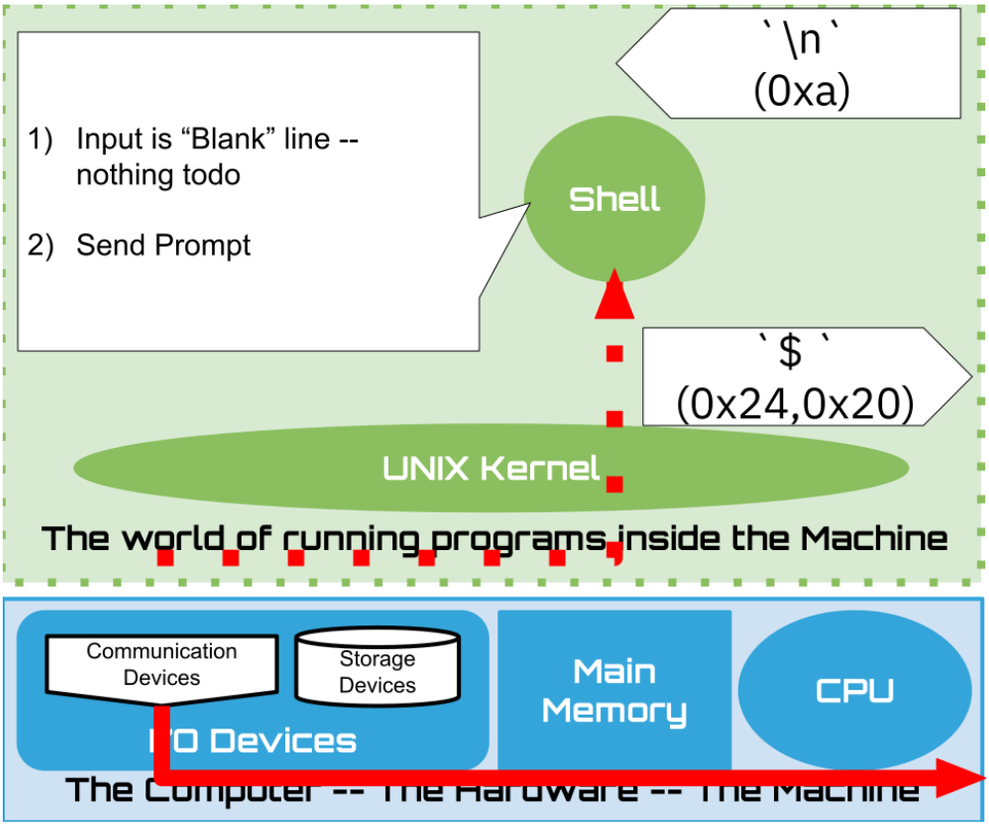
---

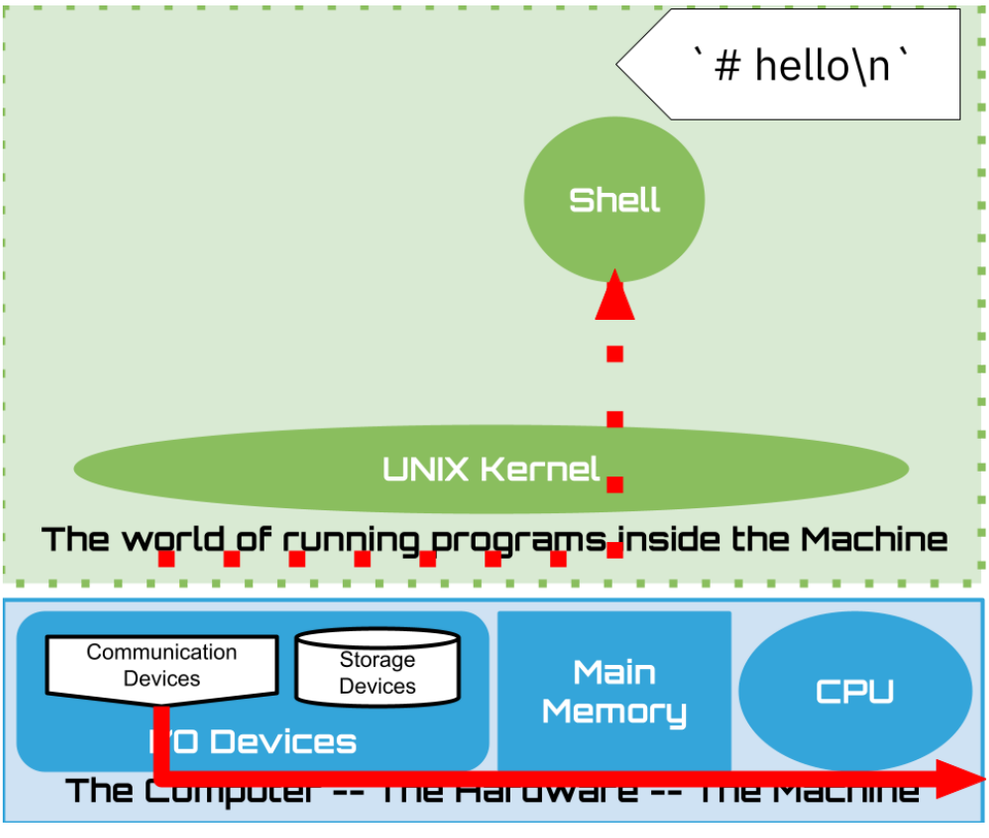
1.15. Shell: Lets have a conversation

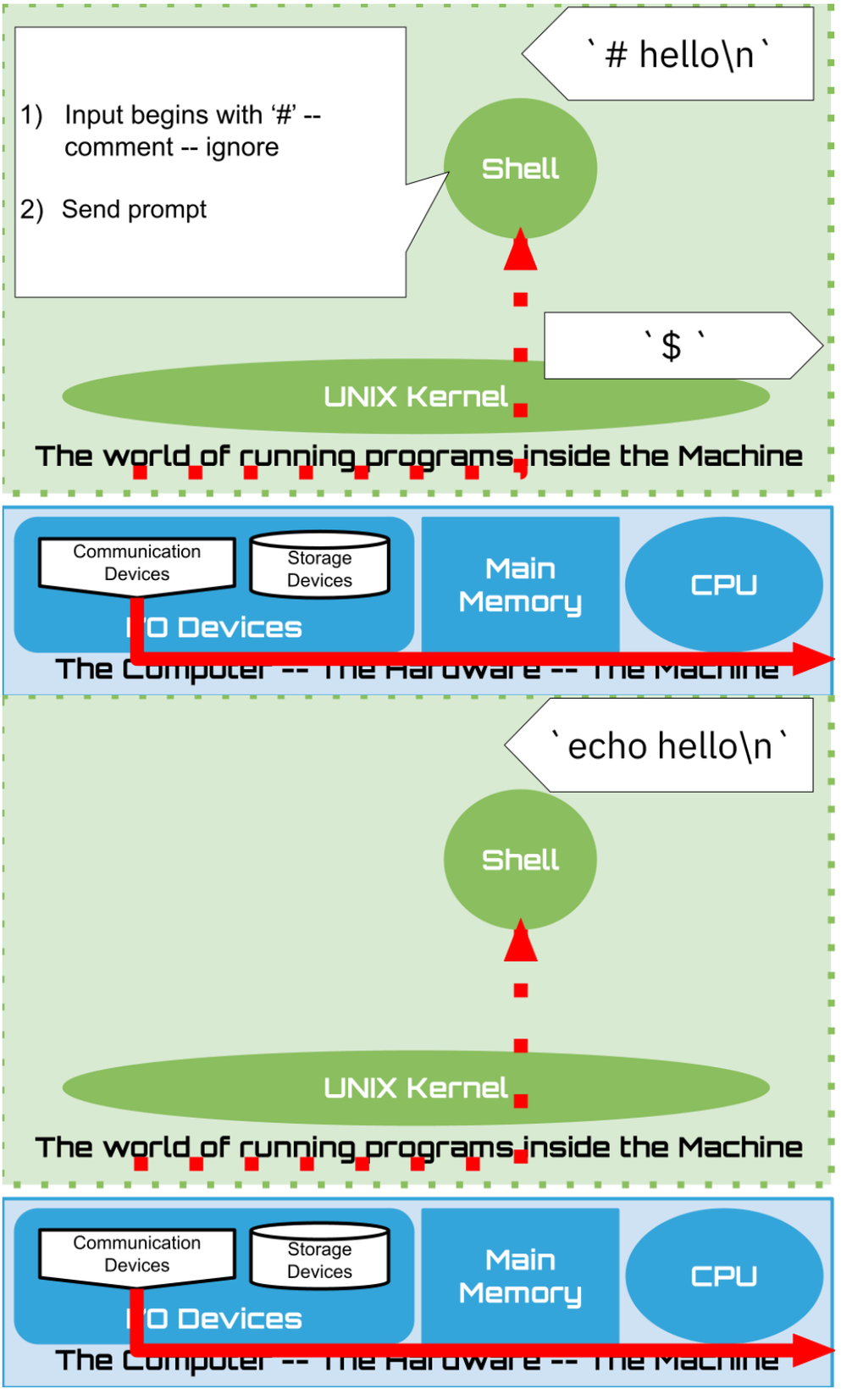
Lets build up our understanding by poking around a little

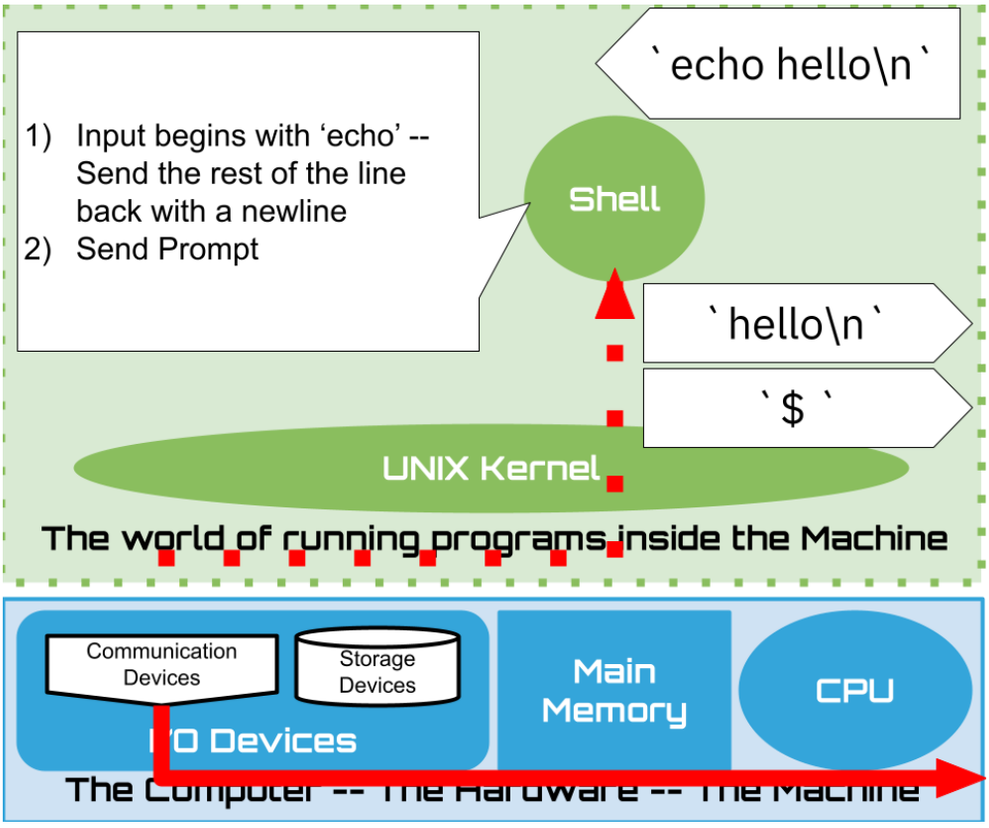

Shell
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

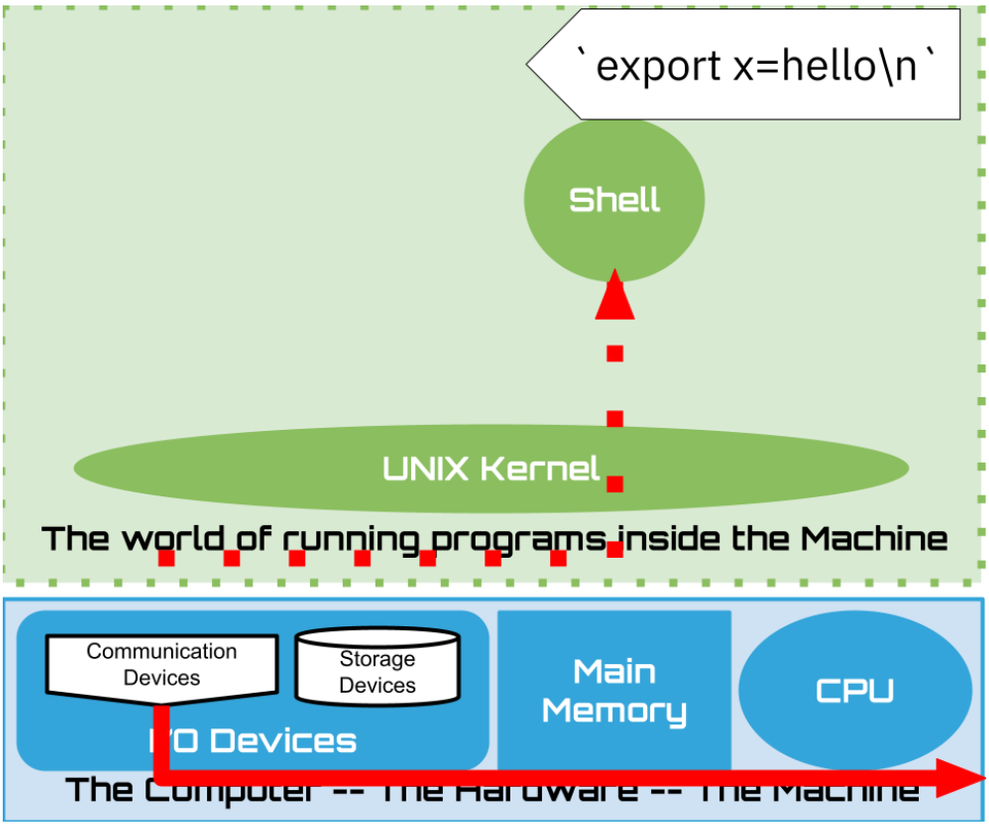

` \n `
(0xa)
Shell
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

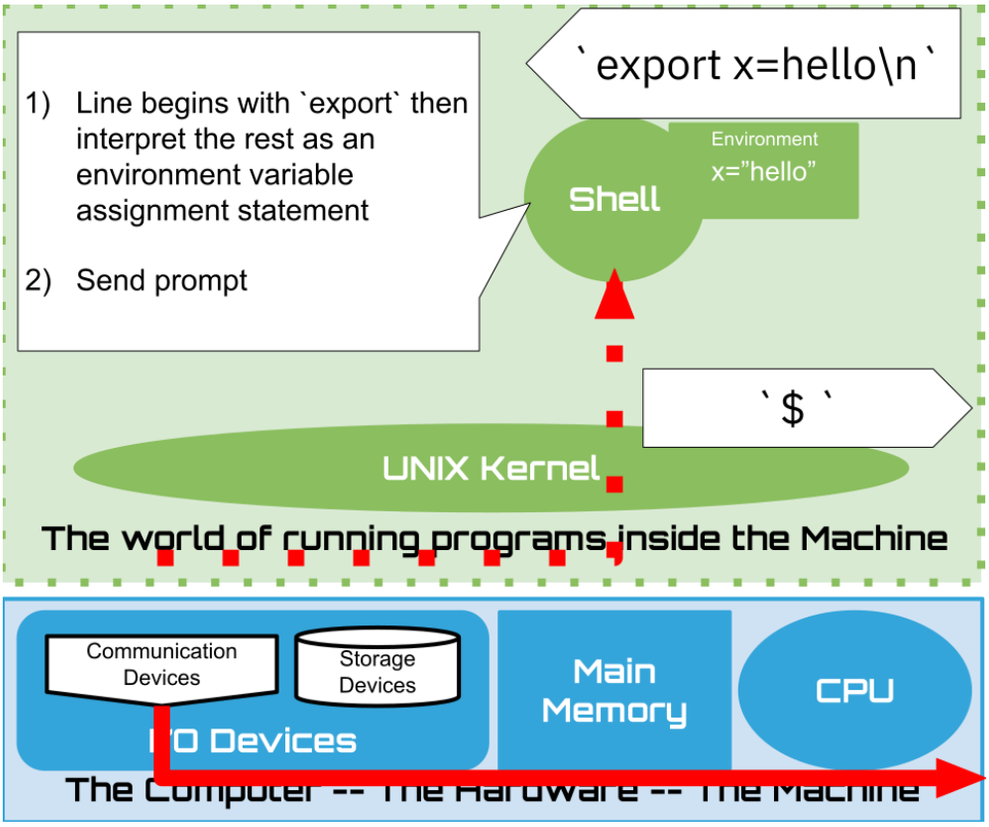

1) Input is “Blank” line -- nothing todo
2) Send Prompt
`\n`
(0xa)
Shell
`$ `
(0x24,0x20)
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine


`# hello\n`
Shell
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

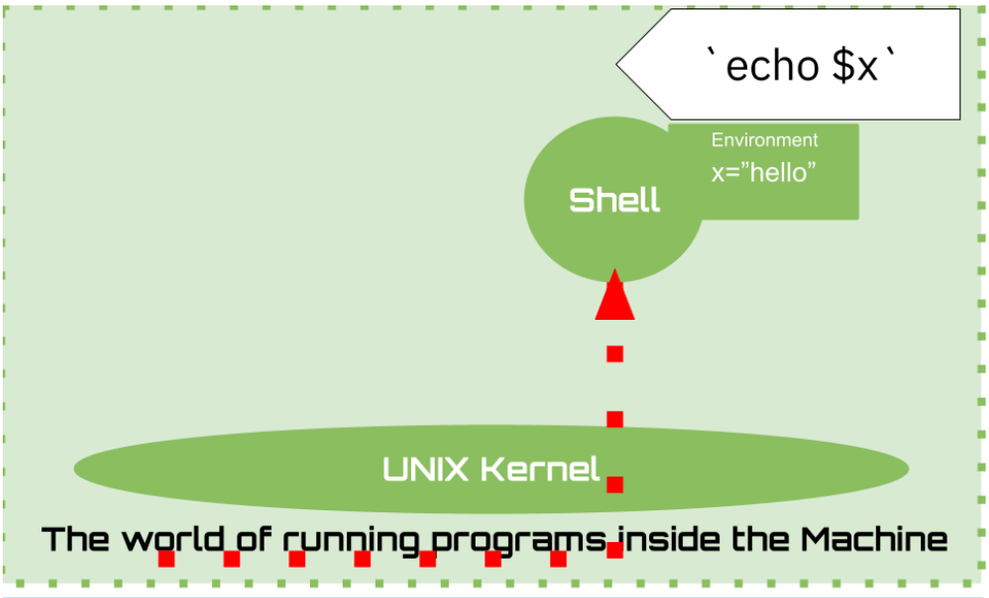

`# hello\n`
1) Input begins with '#' -- comment -- ignore
2) Send prompt
Shell
`$ `
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine
`echo hello\n`
Shell
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

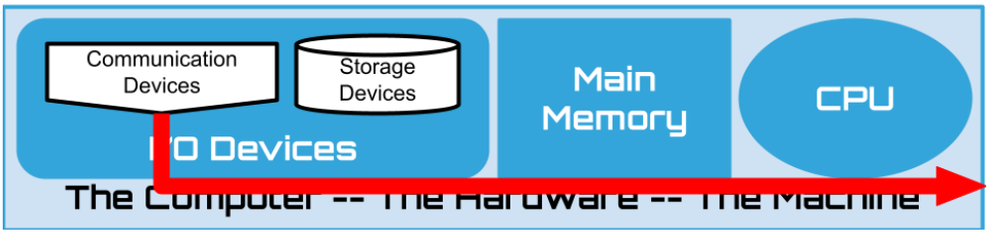

`echo hello\n`
1) Input begins with ‘echo’ -- Send the rest of the line back with a newline
2) Send Prompt
Shell
`hello\n`
`$ `
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

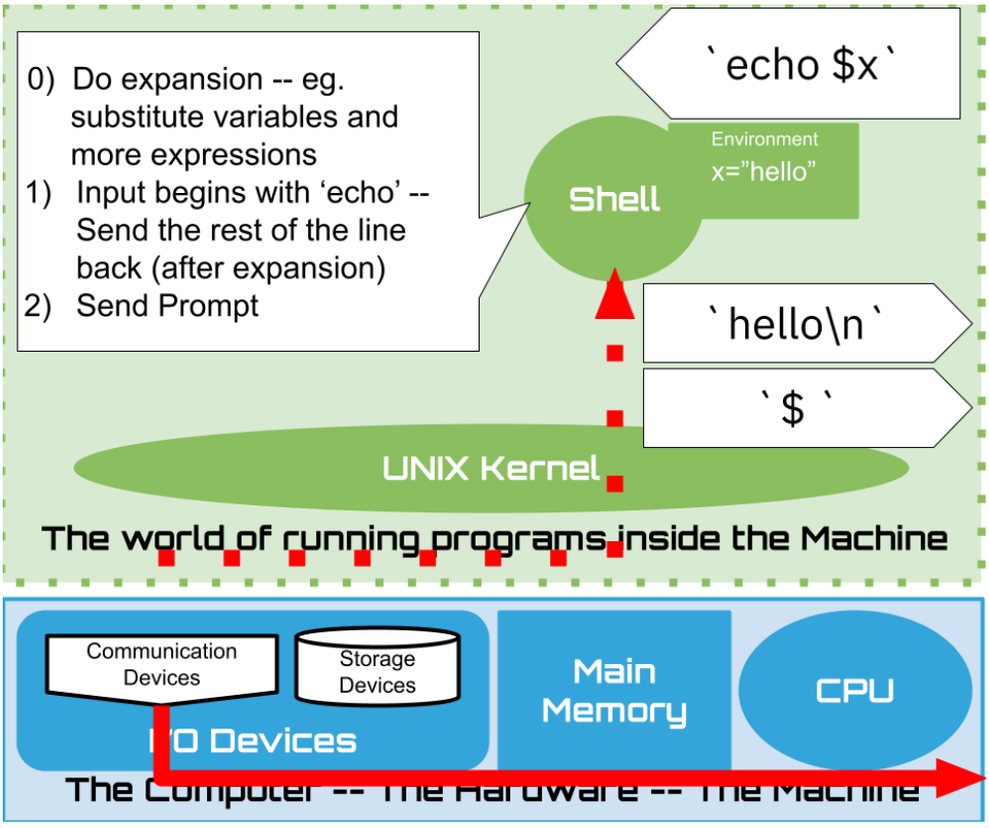

`export x=hello\n`
Shell
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine

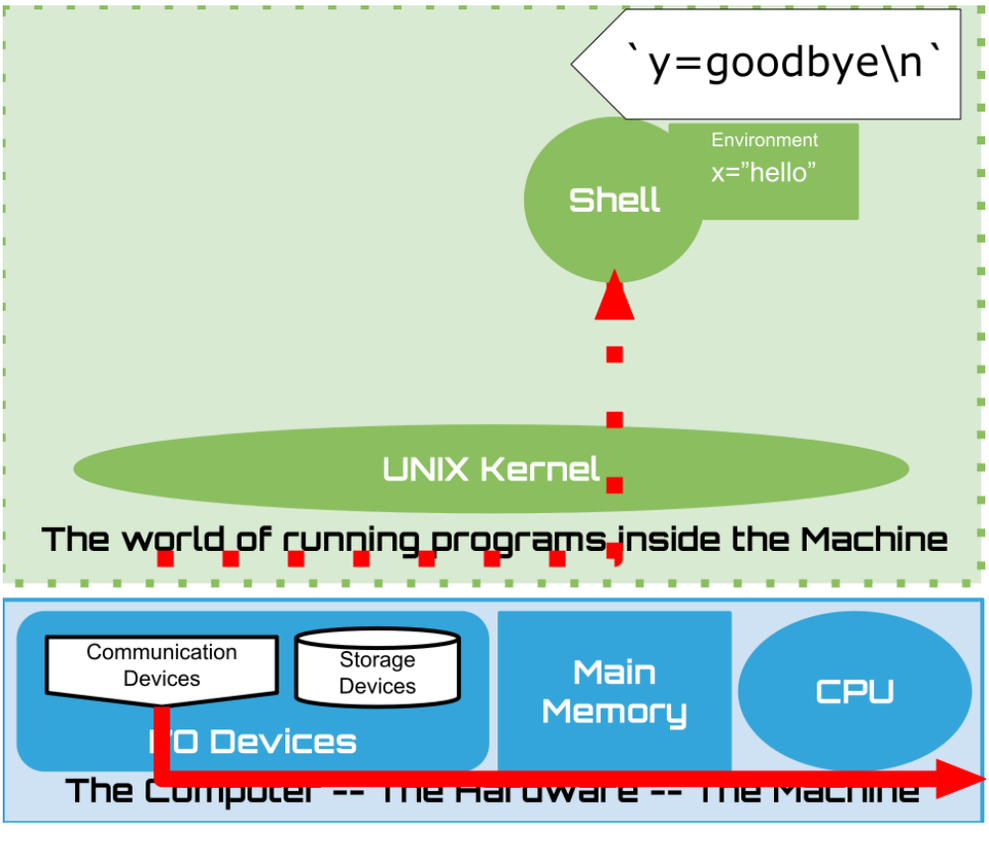

1) Line begins with `export` then interpret the rest as an environment variable assignment statement
2) Send prompt
`export x=hello\n`
Shell
Environment
x="hello"
`$ `
UNIX Kernel
The world of running programs inside the Machine

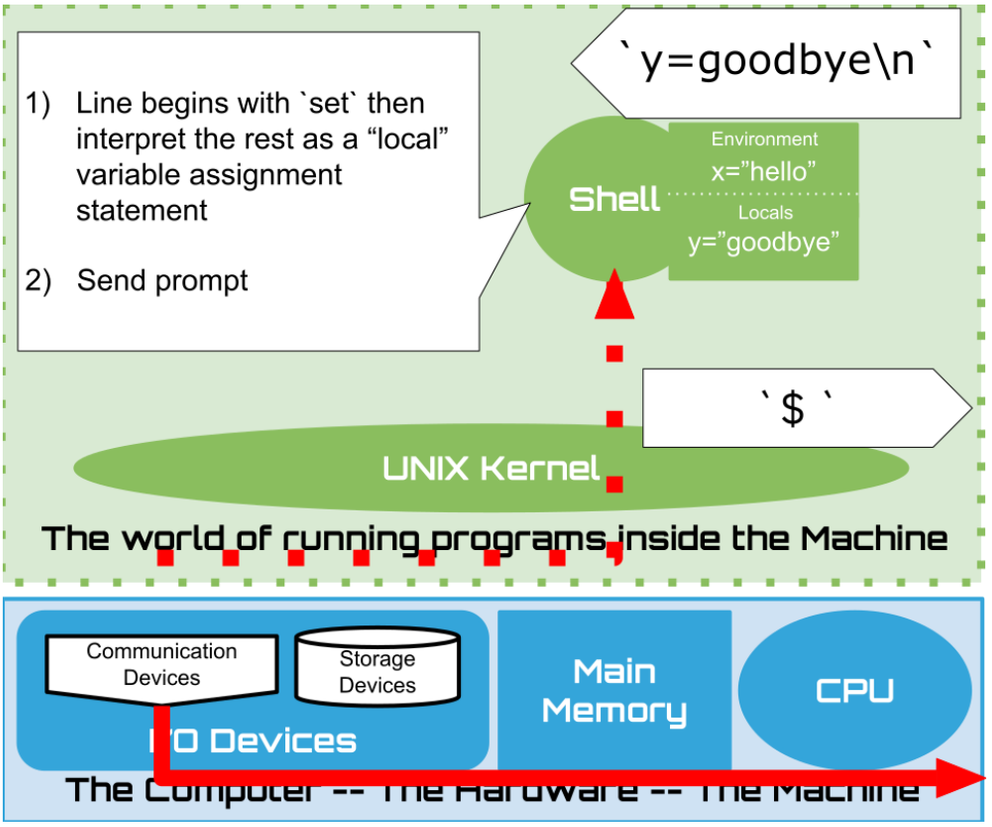
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU


`echo $x`
Shell
Environment
x="hello"
UNIX Kernel
The world of running programs inside the Machine


Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU


0) Do expansion -- eg. substitute variables and more expressions
1) Input begins with 'echo' -- Send the rest of the line back (after expansion)
2) Send Prompt
`echo $x`
Shell
Environment
x="hello"
`hello\n`
`$ `
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine


`y=goodbye\n`
Shell
Environment
x="hello"
UNIX Kernel
The world of running programs inside the Machine
Communication Devices
Storage Devices
I/O Devices
Main Memory
CPU
The Computer -- The Hardware -- The Machine



1.16. Shell: Built-ins

- So we can see that there is some “built-in” behavior
  - Actually “built-in” syntax forms an entire programming language ;-)
  - more on these commands and language later
- Lets dig deeper into the shell’s inherent loop as there is more going on than you might think

1.16.1. Shell: Processing a command line … so far

1. Splits line into blank (space or tab) separated `words`
   - first word is the **Simple** Command to execute eg. `echo foo bar`, `echo` is command
     - later we will see that simple commands can be chained in interesting ways on one line
     - optionally prefixing the command with some variable assignments
   - remaining words are treated as arguments to the command eg `echo foo bar`, arguments are `foo` and `bar`
   - see Shell Grammer for the gory details
2. Does expansions – so far we have only seen one kind
   - But there are actually 9 kinds of expansion
     - we will cover aspects as we need to
3. Execute the command
   - If the simple command is one of the known built-ins then execute it
   - But what happens if it is not

```
$ hello
bash: hello: command not found
$
```

1.16.2. This is where things get interesting

1.16.3. Its not quite as obvious as you might think

1.17. Shell: Externals

So what happens if the first `word` of what you type does NOT match the name of a built in? First off what are the built-ins?

1.17.1. Bash: Built in `help` command

Lets figure it out using the bash `help` built-in

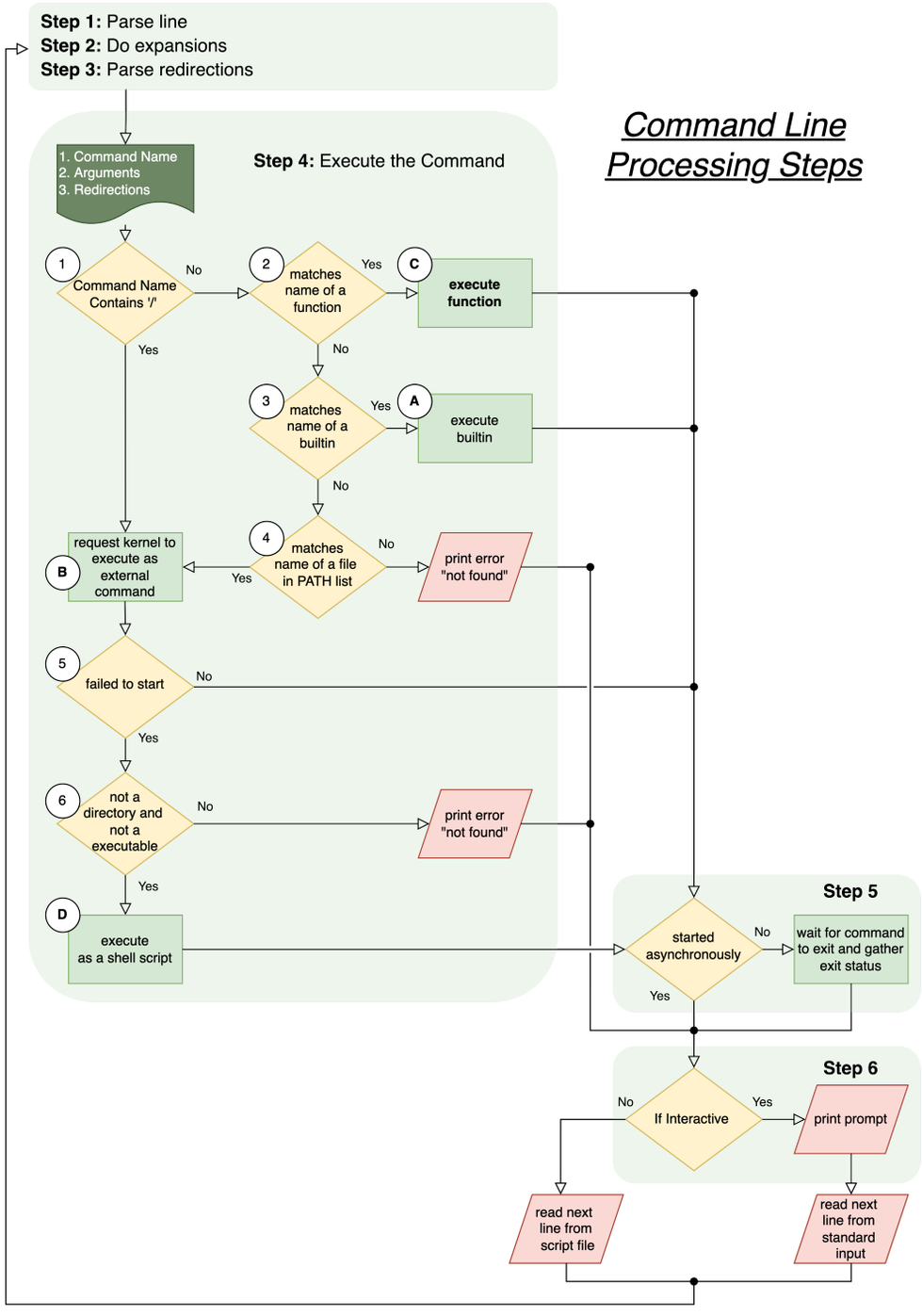
```
$ help
GNU bash, version 5.0.17(1)-release (x86_64-pc-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                            history [-c] [-d offset] [n] or hist>
 (( expression ))                        if COMMANDS; then COMMANDS; [ elif C>
 . filename [arguments]                  jobs [-lnprs] [jobspec ...] or jobs >
 :                                       kill [-s sigspec | -n signum | -sigs>
 [ arg... ]                              let arg [arg ...]
 [[ expression ]]                        local [option] name[=value] ...
 alias [-p] [name[=value] ... ]          logout [n]
 bg [job_spec ...]                       mapfile [-d delim] [-n count] [-O or>
 bind [-lpsvPSVX] [-m keymap] [-f file>  popd [-n] [+N | -N]
 break [n]                               printf [-v var] format [arguments]
 builtin [shell-builtin [arg ...]]       pushd [-n] [+N | -N | dir]
 caller [expr]                           pwd [-LP]
 case WORD in [PATTERN [| PATTERN]...)>  read [-ers] [-a array] [-d delim] [->
 cd [-L|[-P [-e]] [-@]] [dir]            readarray [-d delim] [-n count] [-O >
 command [-pVv] command [arg ...]        readonly [-aAf] [name[=value] ...] o>
 compgen [-abcdefgjksuv] [-o option] [>  return [n]
 complete [-abcdefgjksuv] [-pr] [-DEI]>  select NAME [in WORDS ... ;] do COMM>
 compopt [-o|+o option] [-DEI] [name .>  set [-abefhkmnptuvxBCHP] [-o option->
 continue [n]                            shift [n]
 coproc [NAME] command [redirections]    shopt [-pqsu] [-o] [optname ...]
 declare [-aAfFgilnrtux] [-p] [name[=v>  source filename [arguments]
 dirs [-clpv] [+N] [-N]                  suspend [-f]
 disown [-h] [-ar] [jobspec ... | pid >  test [expr]
 echo [-neE] [arg ...]                   time [-p] pipeline
 enable [-a] [-dnps] [-f filename] [na>  times
 eval [arg ...]                          trap [-lp] [[arg] signal_spec ...]
 exec [-cl] [-a name] [command [argume>  true
 exit [n]                                type [-afptP] name [name ...]
 export [-fn] [name[=value] ...] or ex>  typeset [-aAfFgilnrtux] [-p] name[=v>
 false                                   ulimit [-SHabcdefiklmnpqrstuvxPT] [l>
 fc [-e ename] [-lnr] [first] [last] o>  umask [-p] [-S] [mode]
 fg [job_spec]                           unalias [-a] name [name ...]
 for NAME [in WORDS ... ] ; do COMMAND>  unset [-f] [-v] [-n] [name ...]
 for (( exp1; exp2; exp3 )); do COMMAN>  until COMMANDS; do COMMANDS; done
 function name { COMMANDS ; } or name >  variables - Names and meanings of so>
 getopts optstring name [arg]            wait [-fn] [id ...]
 hash [-lr] [-p pathname] [-dt] [name >  while COMMANDS; do COMMANDS; done
 help [-dms] [pattern ...]               { COMMANDS ; }
$
```

No surprise we don't see a command called `hello` ... we also don't see `ls` but ...

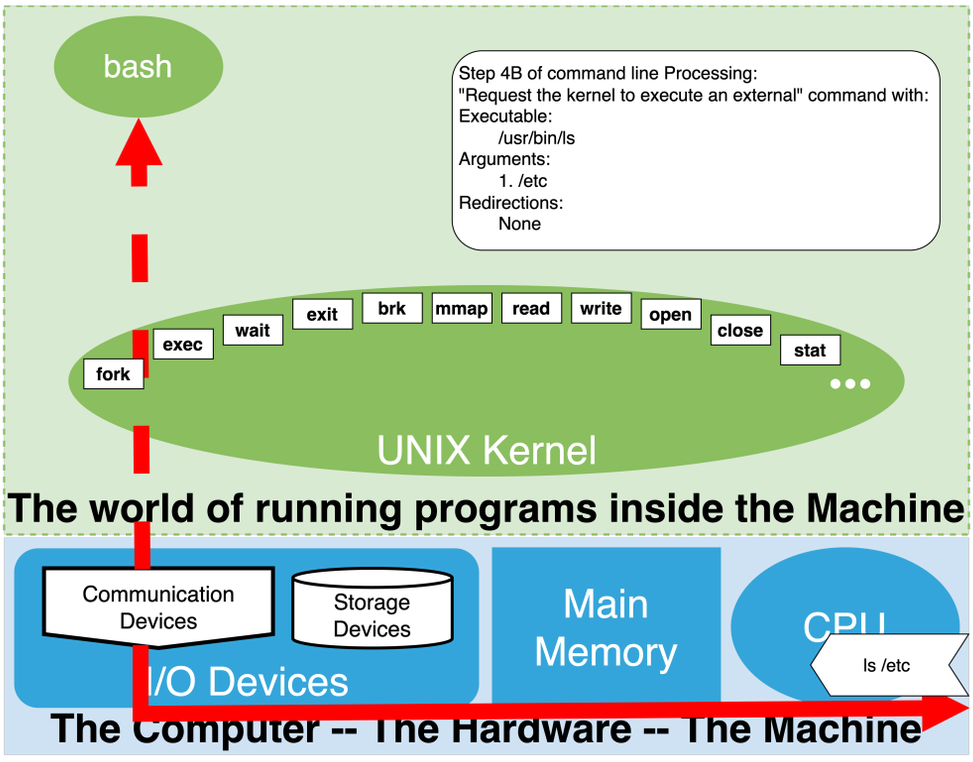
```
$ ls
anotherfile  codedata     errors  mydate  myfile  mynewdir  simpleasm  vmm
cfuncs       datastructs  mybin   mydir   myinfo  myscript  syscalls
$
```

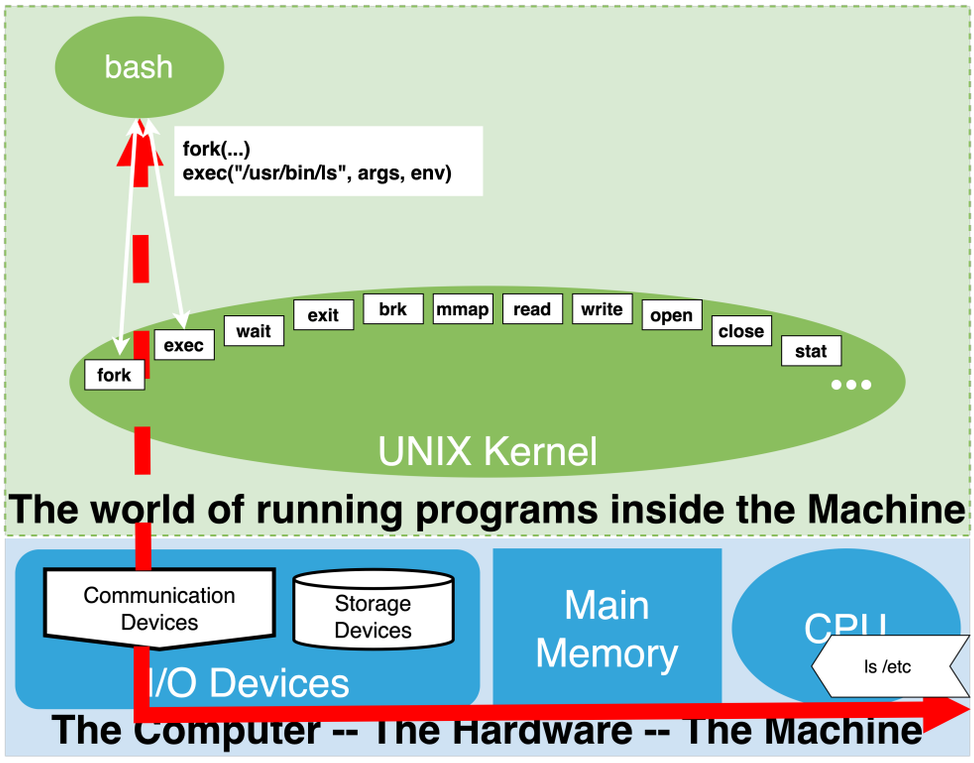
```
$ help ls
bash: help: no help topics match `ls'.  Try `help help' or `man -k ls' or `info
ls'.
$
```

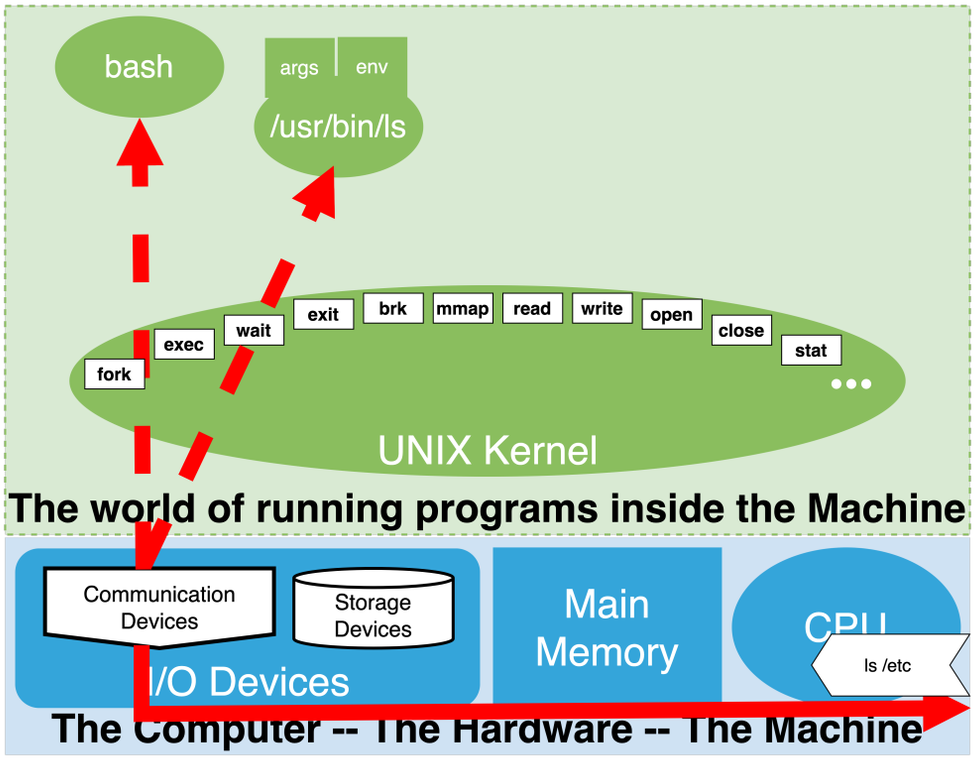
1.18. The gory details



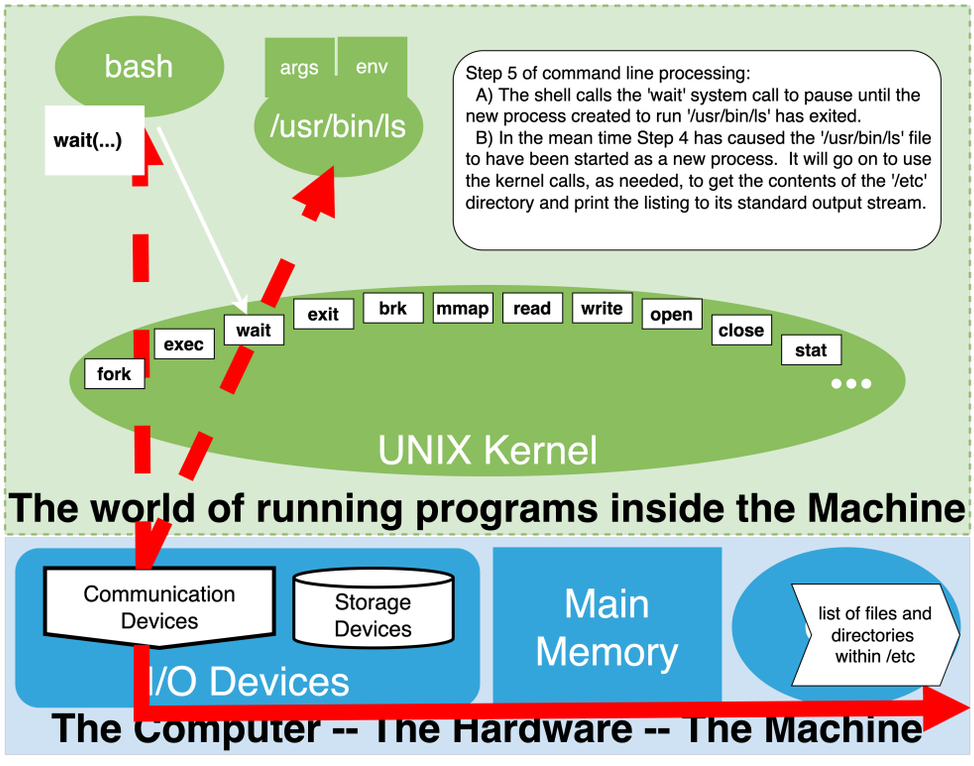
1.19. Step 4B Execute as external command

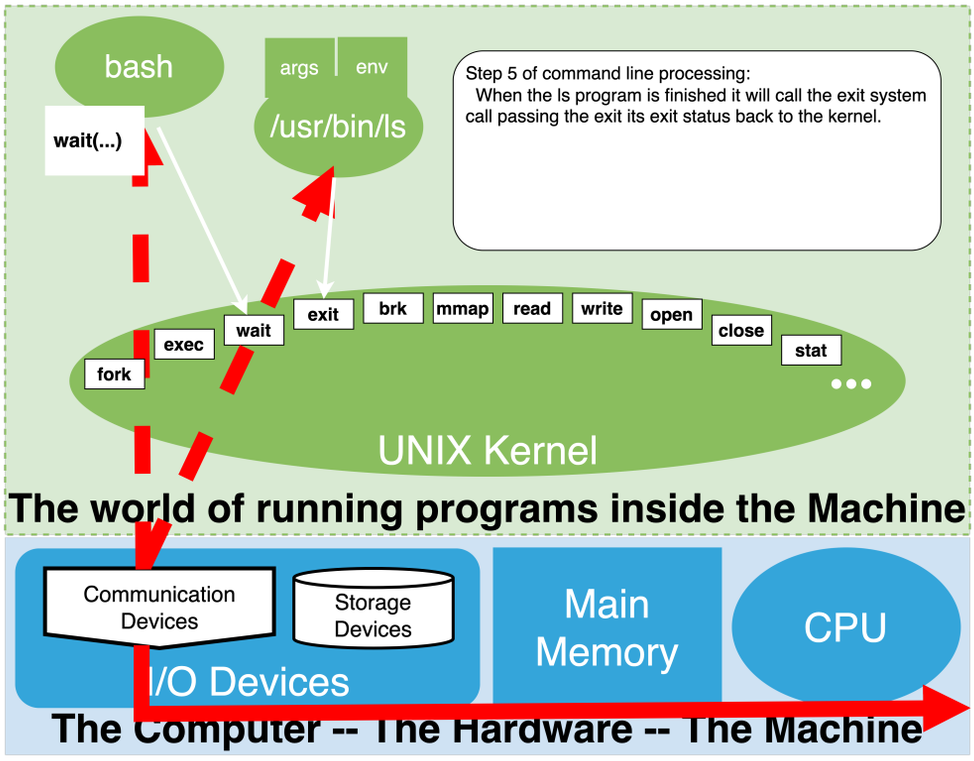






1.19.1. synchronous execution





1.20. Practical summary

1. Shell searches for Simple Command in PATH

```
$ echo $PATH
/opt/app-root/src/juphub-utils:/opt/app-root/src/juphub-
utils:/opt/conda/bin:/opt/conda/condabin:/opt/app-root/src/juphub-
utils:/opt/conda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
$
```

1. if matching name of executable “in PATH” with the help of the kernel **start program** connecting its output
    - wait until it exits

Vim :q!

mac : cntrl + C → stop

web : cntrl + d → stop

1. `/bin/ls` runs - more about `ls` later
    - output goes to the terminal

1. Exits and gets cleaned up by Kernel
    - shell wake’s up
    - shell picks up return value

1. Shell sets special variable `?` to return value
    - prints prompt

This skips several of the details.

1.20.1. PATH Variable and Alternative

- PATH environment variable is
    - colon ‘:’ separated list of directories
    - the shell search list, **in order**, for an *executable* file who’s name matches the command name entered.
        - more about what an *executable* file is later
            - and how to mark one as such
        - first match is used so order of directories matters

- Alternatively if the command name has a slash anywhere in it Eg. `/home/joyvan/bin/foobar`
    - then PATH search is skipped

- if there is an *executable* who's name matches run it

Two ways to get the shell to execute programs:

1. Modify path so shell finds it
2. Explicitly specify the full path name of the program as if it where a command

Notes:

- Current environment variables and their values are "copied into the new program"
- extra arguments on the command line are "copied into the new program command line arguments"

1.21. PAY ATTENTION

Every terminal → new shell
specific environment is only specific for one shell
environment variable is only exists in one shell
환경 변수에 만든 variable은 그 terminal에서만 존재

- This is where a lot of the magic and confusion about the shell and Unix comes in.

1. We now see how the shell is the program we use to find and run other programs!

1. Other programs, if written in the UNIX way, will feel like they are commands of the shell
    - technically externals are not part of the shell
    - but there is a whole set of standard external programs that come with UNIX that one relies on to do anything
        - including navigating the file system
            - see what files and programs exist
            - adding, remove and organizing
    - by initializing PATH these standard programs *'become'* shell commands

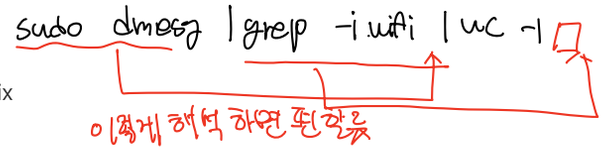
1. A user's programs can naturally extend and customize their command line experience
    - place your own programs in to directories eg. put `hello` into `$HOME/bin`
    - add these directories to the `PATH` variable eg. `export PATH=$PATH:$HOME/bin`
    - `hello` is now a 'shell command'

1.22. The UNIX Way

1.22.1. Don't *Hard code* - Decompose and put the power in the hands of programmers!

Every command has "man" → manual

- break things down into little independent units
    - building blocks eg. programs like `ls`, `grep`, and `wc` (even `bash`), multiverse
        - that compose with what exists
        - naturally extends the current environment
        - can also be easily over-ridden or customized
    - make everything programmable!



Pay attention and you will constantly see examples of the above as we explore Unix

- ls : reads the contents of one or more directories and prints them out
- grep : searches ascii data for specified target patterns
- wc: can count characters, words and lines of ascii data

chmod +x checkwifi

1.23. The Shell Way

1.23.1. lots of little programs that naturally feel like built-ins

- `echo $PATH`
    - `ls /bin`
        - see `ls` there?
        - notice `bash` ;-)

- how can we figure out where `ls` is coming from
  - bash has handy built in called `type`
    - `help type`
    - `type -a ls`
- by tradition all preinstalled programs should have a manual page
  - `man ls`
  - `man bash`
  - `man <something in /bin>`
- many programs also have a help flag
  - `ls --help`

1.23.2. extending the shell is easy

- set `PATH="$HOME/bin:$PATH`
  - putting a program or script called `hello` in `$HOME/bin`
    - now `$ hello` will feel like a built-in

1.23.3. overide what's there

- putting `ls` in `$HOME/bin` will now overide the others
- Even override a built in
  - `function echo() { builtin echo -n "myecho: "; builtin echo $@ }`
  - will talk about this one later

1.23.3.1. natural ways of composing and extending via programming

Two important ones:

1. shell scripts
2. command pipelines

1.23.3.2. shell provide natural model for composition

- put shell commands in a file : eg put this in `hello`

```
#!/bin/bash
echo "My first shell script"
echo "hello"
```

- mark as executable
- now shell will be able to run `hello`

1.23.3.3. pipeline: allow programs to be easily composed

- `ls -1` list files - one per line
- `wc -l` counts lines
- `ls -1 | wc -l` - tells us how man files in this direcotyr
- `ls -1 /bin | wc -l`
- or to get really fancy
- exploit more knowledge about shell expansion abilities
- `echo $PATH`
- `echo ${PATH//:/ }`
- `ls -1 ${PATH//:/ } | wc -l`
  - what do you think this did?

A lot of the above are just teasers we will be covering these topics in more detail now that we have a more general idea of how things work

1.24. Before we end lets reveal more of the Shell Loop

- explore aspects over the next few lectures and the rest of our lives ;-)

1.24.1. Bash Processing Loop for simple commands

1. Read input line

1. Splits line into `words` and operators
    - applies **quoting** rules as part of this step
        - we will cover as much of this as we need to
            - eg. examples of when to use `""`, `''`, `/` and no quotes or escapes

1. **Expansions**: Nine of them we will primarily focus on core aspect of these 3

- tilde
- variable
- filename

See notes for more info

```
1. Brace Expansion: useful but you can wait to learn about it
2. **Tilde Expansion** : simple and worth knowing
3. *Parameter and Variable Expansion*: really useful to know basics - rest can
wait till later
4. Command Substitutions: useful but you can wait to learn about it
5. Arithmetic Expansion: useful but you can wait to learn about it
6. Process Substitutions: subtle you can really wait on this one
7. Word Splitting on expansion results: subtle you can wait on this one
8. **Filename expansion**: basics are a must know
9. Quote removal: nothing really fancy here just removal of left over quotes
  - eg. echo "hello"
```

See https://www.gnu.org/software/bash/manual/bash.html#Command-Search-and-Execution

See https://www.gnu.org/software/bash/manual/bash.html#Shell-Operation

1. **Redirections**: how we control where input and output goes
    - very important we will cover the basics

1. Execute the command: (steps 1, 2 and 3 affect how the execution will happen)
    1. if command has no slashes '/'
    - if there is a 'shell function' who's name matches invoke it: **DONE**
    - else look for a matching built-in and invoke it: **DONE**
    - else look for matching file in path
        - if found then carry on to B
        - else print error : **DONE**
    1. External program execution: either command had a slash or was found in path
    - ask kernel to run program passing copy of arguments and environment variables
        - a command can be prefixed with some extra variable assignments
    - if kernel request fails
        - if failure was due to file not being executable (and it is not a directory)
            - then try attempt to run it as a shell script (eg. bash /tmp/hello) : **DONE**
        - else print error: **DONE**
2. Optionally waits for command to complete and get return code
    - update `$?`
3. Print prompt if Interactive

1.24.1.1. Two basic modes

1. Interactive:
    - Command lines are read from the terminal connection
2. Non-Interactive:
    1. Command lines read from a file (Shell scripts)
        - explicit:
    2. Command line passed

1.25. NEXT

1.25.1. Now that we understand some basic Terminal and Shell concepts we can get on with learning some more factual / functional information

- Syntax of the shell as a programming language:
  - What the shell does beyond simple commands:
    - loops, if, case, functions, more about variables
- More about some the expansions the shell performs
- Working with files and directories
  - current working directory
  - changing directory
  - relative vs full path
- I/O redirection.
- Processes and Process control

1.25.2. After that we will focus on some particular Programming Tools

1.26. Things worth looking at.

- `help`
- `man intro`
- `man man`
- `info info`
- `info bash`
- `man bash` This is very big and detailed. Best to skim it and see what catches you eye. Eg.
  - Definitions
  - Reserved Words
  - Shell Grammar
    - Simple Commands
    - Pipelines
    - Lists
    - Compound Commands
  - Shell Function Definitions
  - Shell Variables
  - Expansion
- `man ls`
- `man wc`
- `man grep`
- `man <cmd>`
- `info info`
- `info bash`
- Online official bash manual: https://www.gnu.org/software/bash/manual/bash.html
- `whatis` display one-line manual page descriptions
  - `man whatis`
  - eg. `whatis ls`
- `apropos` search the manual page names and descriptions
  - `man apropos`
  - eg `apropos games`