Systems Programming

8.28.24

Lecture 01:

MULTIX → UNICS → UNIX → Linux] IEEE certified
→ POSIX

Unix Philosophy:

Lecture 02:

8.30.24

MULTIX → UNICS → UNIX ⇔ POSIX (IEEE)

Unix is a philosophy:
- write programs that do one thing really well
- write programs that work together
- write programs to handle text streams, as that is the universal interface

Ex: Shell scripting
* uses data pipelines

ls -l | wc -l
{ allows to direct the output of one code to the input of another

* very text-based and human readable

* use CSE-student machines 10-13
  student10.cse.nd.edu

SSH into these!
- windows - PuTTY
- can also use Visual Studio?
- must be on eduroam

Unix Commands:

-ls	files in dir
ls -a	all files
pwd	print working directory
more X	shows contents of file X
cd ~	go home
cd ..	go up one dir
which X	where is file X coming from

* tab helps you auto-complete
* code will be graded on student machines

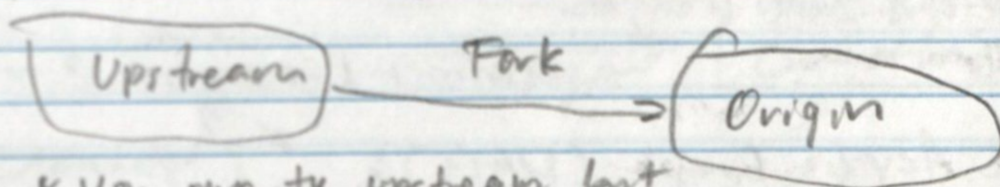
Homework 00:
- create 2 private Git repos
- enable passwordless login
  ssh-keygen to generate key
  copy public key from .ssh directory
- add/commit various files
- confirm on GitHub

* add meaningful commit messages
* commit often

- files are hidden files
- ssh is a hidden directory
  ~ view ssh keys here

Git Overview:
- free-open-source distributed version control system
- journal, time machine, shared space
'repository': inside .git directory that contains data and journals the history

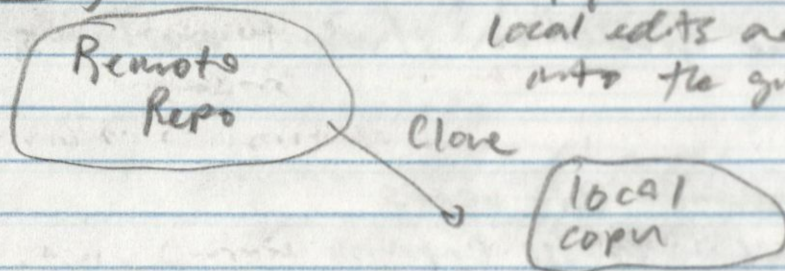
Upstream Repository: master repository



* you own the upstream but can edit the fork
* has to stay public

Cloning:

* copys entire repo and allows local edits and pushes back onto the git repository

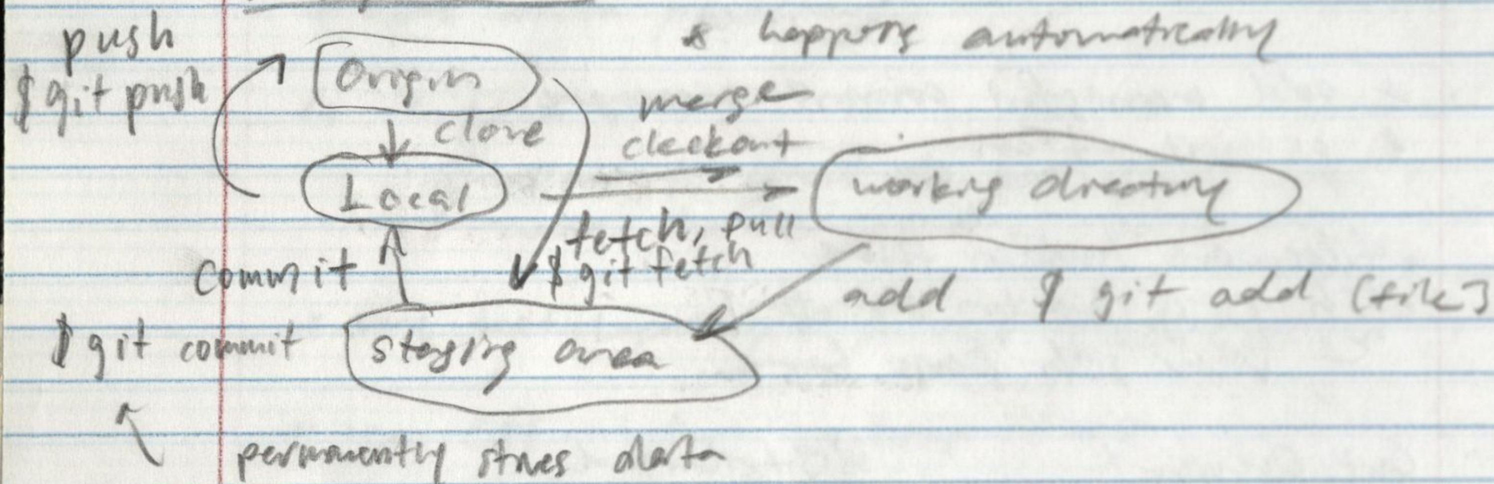


on laptop and CSE student machine NOT the same

$ clone [link]

Working Directory:

* happens automatically



permanently stores data

* pushing sends recorded data from local to remote repository (origin)

* fetching copys changes from origin to local mirror

* pull fetches and merges
$ git pull

commit 1 → commit 2 → ...

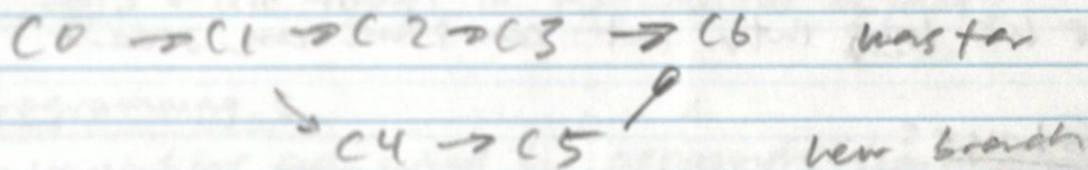
Directed Acyclic Graph (DAG):
- recorded commits stored in a DAG
- each commit has a unique hash identifyer
- last commit is the head
- default branch is the master (main)

git log - shows commit logs
git log --oneline - shorter commit logs
- shorter hashes

git checkout [hash] - returns to previous version

git switch - restores back to HEAD vers

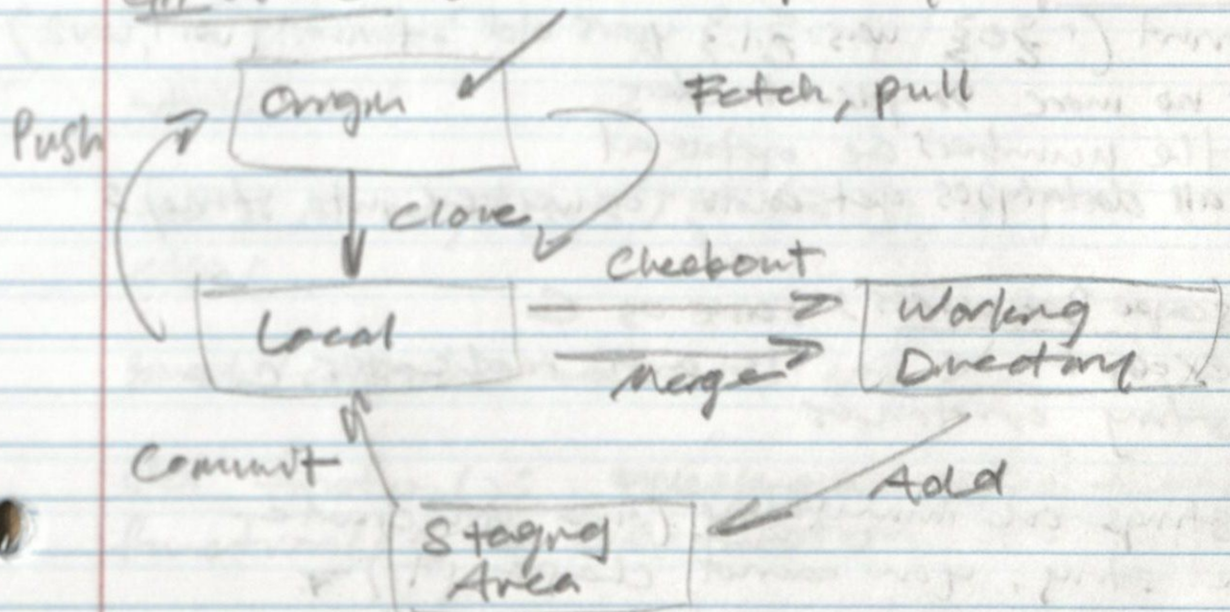
Branches: alternate line of development



working dir can only show one branch
so have to checkout wanted branch
must merge branches together again

* branches can have conflicts

Git Overview: "remote repository"



* pulling performs a fetch and a merge

Week 1 Readings:
Python Basics:
comments
print ("hello world")

Literal Constants = literal value and unchangeable
ex. 5, 1.23, 'this is a string,' "string"

ints and floats work the same
* no long int, int can store any size *

Strings:
' ' : white space preserved as-is
" " : exactly the same
'''
multi-line string
'''
* no char data type

Formatting:
n times
print ('{0} was {1} years old'.format(var1, var2))
* no more % placeholders
* the numbers are optional
* all datatypes get auto-converted into strings

Escape Sequences: same as C
except \ on its own indicates a
string continues

* strings are immutable (once you create a string, you cannot change it) *

* print() always ends with an automatic newline
print('a', end='') to override this

* identifiers, variables, and datatypes are basically the same *
- main datatypes are numbers and strings
- you can create your own datatypes with classes

objects: all things in Python are objects

Programming:
- variables are used by assigning them a value
- no declaration or data-type are needed
- semicolons are not needed
- curly braces are not needed
- indents and newlines have an effect on logic

```
if var == true:
elif var == false:
else:

while var:

for i in range(1, 5):   → [1, 4]
else:
```

break / continue work the same

```
def function():   # declare and define function
function()        # call function
```

Week 2 Readings 3 9.2.24

Python:

power ↓ (**) ; division and floor (/, //)

Operators: +, -, *, **, /, //, %

↳ <<, >>, &, |, ^, ~, <, >, <=, >=, ==

left shift, right shift, AND, OR, XOR, invert, ...

!=, not, and, or

NOT, AND OR

* includes syntactic sugars
* learn evaluation order (operator precedence)
  → check documentation

Functions w/ Parameters:

```
def print(a, b):    # declaration/definition
print(x, y)         # function call
```

* don't need datatypes
* variables inside functions are still local
  → unless declared as global

lecture 03: 9.2.24

git:

repository, working directory, staged files,

branches:

C0 → C1 → C2 → C3 → C5 master

↘ C4 → C6 br-fix1912

* since our working directory can only show one branch at a time, we must checkout the branch we want

$ git status tells us what branch we're on

Checkout: updates the files in the working directory to match the changes in the branch (kept locally)

Merge: merge these branches back to main

conflicts: if there are conflicting changes in different branches, the user must resolve before committing a merge

Pull Request: before merging a branch into main, you must make a pull request (and usually a code review) before the changes are pushed to master

Python:

Why?

- very readable and well structured
- rich ecosystem and frameworks
- free and open source, wide spread
- TensorFlow, NumPy, Pandas, Flask, ...

Python 1.0 — late 1980s
Python 2.0 — 2000
Python 3.0 — 2008

Python 3.12 — Apr 2024 * old but still maintained

which python3
python3 --version

comments: start with #
#! / location python3
shebang - how to run code

* code and variables are typically long
* write comments to explain why not how

imports: bring in code beyond basic Python

```
import sys
import os
import json
```

→ sys.argv[1:]
must use namespace to access functions / vars

```
from sense_hat import SenseHat
```

bring in specific features into global namespace

* not too much OOP in this class

varables: no datatypes needed, all automatic

* interpreted language — so no errors once no compilation errors indicated from program not working

name = something
"can love anything = anything"

int → Integer	* Everything in Python is an object
float → Decimal	• belongs to a type
string → Text	• has methods (functions)
boolean → True / False	• has attributes (data)
"" ?	

name = value
type(name) # returns type
list(name) # converts between types
int(num)

lists and dictionaries: memory safe language
- we will come back to this
- [] is how we index into something (array/list)

Control Flow / Blocks:
- no curly braces
- indentation matters

* colons inside [] allow you to access parts of a string

*arrays automatically populated
* can have any number of "d's" in an array

ex.
```
import sys
arguments = sys.argv[1:]
fh = open(arguments[0], 'r')
json_data = json.load(fh)
fh.close()

for i in range(len(json_data)):
    print ...
```

List: group of variables (objects), do not have to be of the same type
(typically is)
- index via []
- indexes start at 0
- print(myList)

```
var = myList[1:]      # start at 1 and take the
                        rest of the data
var1 = myList[-1]     # access last element of list
```

*Lists and Arrays are basically the same "pythonese"

Dictionary: name + value pair
- set of keys where each one has a value
- can blend datatypes (not smart)
- in a list the key is the index
- in a dictionary it can be anything

ex. dictionary = {'Name': 'Anda', 'Grade': 'Sophomore'}
print(dictionary) + print(dictionary.keys())

Week 2 Readings cont. 9.3.24

Data Structures:

List: dynamically allocated Python array
use [ ] to define

Tuple: immutable list
use ( ) to define

Dictionary: stores keys (immutable) associated with data (mutable). use { } to define
ex. d = { key1 : value1, key2 : value2, ... }

Sequence: special kind of List/Type with special features. use [ ] to define

Set: unordered collection of simple objects
use set([ ]) to define

Object Oriented Programming:

```
class Person:
    #code
    #methods — def hello():
p = Person  # class call
```

I/O:

Input:

```
variable = input("Enter text: ")
```

Files:

```
f = open('file.txt, 'r')  # or 'w'
f.close()
```

Lecture 04: 09.04.24

Variable arguments: handle via argparse

-- method
  • count the # methods

2 lookups

-- member
  • count # of ::

Homework 01

-- ptr
  • count # of pointers (->)

-- simplefunc
  • count # of simple funcs

-- simplefuncee
  • allow curly brace to be at the end of the line

Simple Functions

```
Function name {   like this
{
  # ONE line of code
}
```

.gitignore: tells git to ignore files

  .DS_Store
  repo-list.csv
  Enrollment-FA24-20289.csv

* Statrcipy ex of argparse *

Python:

Dictionary:

- Name + Value Pair
- can blend key types but not recommended
- key is the index
- define with {}, access with []

ex.
```
d = {'key1':'val1', ...}
print(d)      {'key1' : 'val1' , ... }
d['key1']
print(d.keys())
```

\# keys must be unique
\# reaccessing a key overwrites its value

ex.
```
if 'SSID' in theDict:
    # checks for a key ('SSID' in this ex)
```

- ERROR if trying to access key that doesn't exist

Loops:

```
for i in theList:  # iterates over the list
                   # dont change the list while iterating
for theKey in theDict:  # iterates over the keys
# can still use normal for-loops
```

Multi-Value Looping:

```
for theKey, theVal in theDict.items():
    # theKey holds the key
    # theVal holds the val
# dont change/edit/delete while looping
```

Functions:

```
def function(x):      <- parameter
    # code
```

- no return type
- no parameter datatype
- no end, just indents

Invoking:

- function(number)   <- vars have scoping

* can return 0, 1, or multiple vals
* can have 0, or any # of parameters
  no datatypes!

```
return val1, val2
val1, val2 = function(num)
```

Tuple: immutable list

```
print(type(var))   # prints datatype
```

* a dictionary can be a key to another dictionary
  but allowed

Conditional:

```
number = random.randint(0, 10)
if number % 3 == 0:
    print('Fizz')
elif number % 5 == 0:
    print('Buzz')
else:
    print(number)
```

* can use C-style printf() formatting
  %d, ...

Exceptions:

```
num = [0, 1, 2, 3]
try:
  print(num[4])
except IndexError as e:
  print('Oops!', e)
```

Loops:

```
for item in [0, 1, 2]:
    print(item)
                         1, 2, 3
                         [1, 4)
for value in range(1, 4):
    print (value)

for i, j in enumerate(range(1, 4)):
```

Lecture 05: 9-6-24

- copy .cc files into personal repo and exclude them with .gitignore

Homework01 Details:

- can have multiple flags
- print info on newlines
- arguments can be out of order

* GitHub Copilot demonstration

Open a File:

open(file, mode='r', ...)
'r': read mode (default)
'w': write mode, truncating the file first
'a': write to end of the file
: (file pointer points to the end)
* can/should only have one file writing on a file at a time
't': text mode
'b': binary mode

I/O Read File:

```
for line in open(path):    -> file path
    line = line.rstrip()   -> removes any spaces at end of string
    print(line)            -> gets file by line instead of by character
```

* look at reading Python files example sheet *

* open returns a FILE object *

I/O Writing Files:
```
with open(path, 'w') as fs:
    fs.write(data)
```

* with automatically closes the file when we leave its scope
* close + flush the file
  ↳ write it to disk

Ask the OS:
```
>> os.path.join(os.curdir, 'README.md')
./README.md
>> os.path.exists('/etc/hosts')
True
```

* os.path.method() is useful!

Run Commands Directly:

Old: os.system('ls -l')
New:
subprocess module:
subprocess.run ...

.run and .popen
(blocking) (nonblocking)

Processes next week

Week 3 Readings: 9.8.24

Regular Expressions: help search for a pattern of text
(regexes for short)

```
import re
variable = re.compile(r'\d\d\d-\d\d...')
#creates a regex object
```

Matching Regex: search() method

```
var = regex_object.search("string")
#returns None if regex pattern not found
#returns a match object if pattern exists
print(var.group()) # prints match
```

Grouping:

```
regex_obj = re.compile(r'(\d\d\d)-(\d...)')
```

access with group(1) group(2)

groups() accesses all matches

Character Classes:

- \d - numeric digits 0-9
- \D - any character NOT 0-9
- \w - any letter, numeric digit, or underscore
- \W - any character NOT a letter, numeric digit or underscore
- \s - any space, tab, or newline char
- \S - any char NOT a space, tab, or newline

Escape Chars: \., \^, \$, *, \+, \?, \{, \}, \[, \], \\, \|, \(, \)

Lecture Notes: 9.19.24

Subprocesses:

os.system("ls -l") OLD

Subprocess module:

subprocess.run(["git", "-C", str(Path('Repo')), "pull"])

subprocess.popen(... ↗

run: "wait until its done"

- forking initial process
- blocking
- once child then comes back to main

popen:

- runs in background

Unix help: XXX --help
man XXX

o popen:

- pipe in / out of information
- write to stdin, read from stdout (stderr
- allows child process to run in tandem
  - → NOT blocking

* typically we will use .run *

Process : instace of a running program.
Each process has:
- Process ID (PID)
- Parent Process ID (PPID)
- Priority
- Nice Number
- Terminal / TTY
- UID / GUID (user ID / group ID)

Process

C Library

System Call API

Kernal

Hadware

User

Kernel

Also:
Memory
Files (Open)

Kernal : OS

- Each process has an unique PID
- A process can copy itself via fork
- when a process forks, it creates a child process

!
!

View Running Processes:

UNIX / MAC:

ps

ps -a -f

top

Windows:

tasklist

Task Manager

Arguments : sys.argv

```
import sys

for argument in sys.argv[1:]:
    print(argument)
```

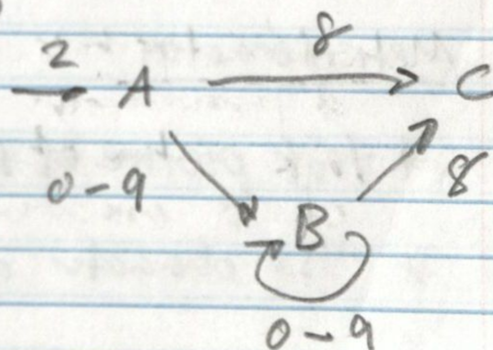
Regular Expressions : Regex

- [ ] - thing to look for
- look for a pattern to match to something
- ^ - start at line
- generally for text processing
  - CAN technically do on binary

Theory : specification of language represented by finite automaton (FA)

grep -E '2[0-9]*8'

extended grep capabilities



"look for 2, look for 0-9s, until an 8"

Unix Utilities grep, sed, and awk use Regex, supporting POSIX Basic Regular Syntax (BRE):

- Metacharacters such as ()|{}[] need to be escaped '\'

```
grep '\(user\|User\):' /etc/passwd
```

- ERE doesn't require escaping

```
grep -E '(user|User):' /etc/passwd
```

-E ↑ use ERE; (user|User): ↑ the pattern (regex); /etc/passwd ↑ where we want to find the match

In Python:

```
import re
re.search(r'[0-9]', 'a3qz')   ✓
re.search(r'[0-9]', 'pbni')   ✗
```

(r' ← raw string)

```
re.findall(r'([0,9])', 'a3qz')
```

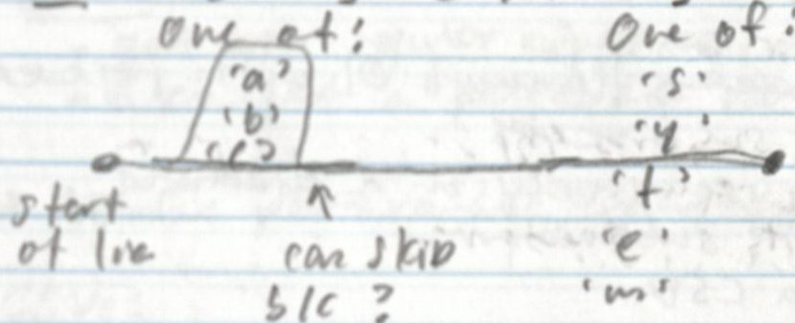
extract matches. ex. ['3']
empty set, if no matches

Metacharacters:

* took picture of slides *

* also character classes *

Ex. ^[abc]?[systems]



aaron F

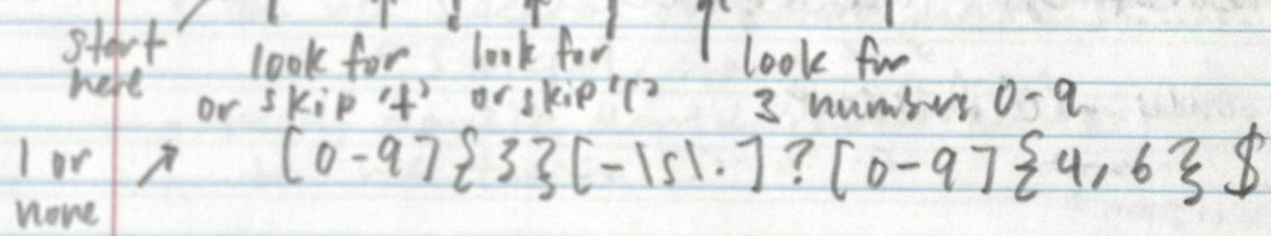
ads F

[s]triegal T abc optional

[cs]e T

[s]ystems T

Ex. ^[\+]?[(]?[0-9]{3}[)]?[-\s\.]?



+(✓

+1 X (not entirely correct for phone # lol

Lecture Notes: 9.11.24

Homework 2: enhance Homework 01 with regexes
- run tool as subprocess
- operate it recursively on a directory to get in its subdirectories
- output to a CSV
- extract various statistics

path: separate these
file: libraries do this
lines: same
include: same
local includes: only local includes
memberfuncs: use regexes
onelinefuncs: use regexes

* only .cc files *

Recursion:

```
def index_dir(path):
    index_dir(path + '___')
```

→ hw
→ hw01

-R UNIX command to
-r enable recursion

* OR use os.walk()

Exams:
- get both regular expression / metacharacter cheat sheets
- also get a page of notes

* REGEX phone number examples *

CSVs:
- each line is a record seprated by a CRLF (line break)
  - (\r or \n)
- last record may not have line break
- first line is optional header
- each line has same # of fields
- fields may be in " " or not
  → needs to be consistent
  → all in on a column or not
- can use special characters inside " "
  → "Striegel, Aaron", ...
- need double double quotes inside " "
  to use them

* Rainbow CSV VS-Code extension

CSVs and Python:
- roll your own
  → read line by line and split off of ,
- import csv package
  → csv.reader // read/parse one line at time
  in a list
  → csv.DictReader // "
- use a regex

Reading Week 4: 9.15.24

Functional Programming:

pure functions: f'n whose output value follows solely from its input values w/o any side effects

side effect: f'n that modifies its calling environment (BAD)

functional programming: program consists primarily of the evaluation of pure functions

* programming paradigm (like OOP) *
- → high level
- → transparent
- → parallelizable

first-class citizens: in python f'ns are FCCs, meaning they have the same characteristics as strings/nums
ex. you can assign a function to a variable

function composition: passing a function object as an argument
sometimes called a callback

Anonymous Functions: def a function "on the fly" without a name. use lambda
lambda <parameter_list>: <expression>

* has its own local namespace
* parameters don't conflict with identically named parameters in global namespace
* can access variables in global namespace but cannot modify them

map() : built in functions that applys a function to each element in an iterable

filter() : built-in functions that allows you to filter items from an iterable based on the evaluation of a given function

reduce() : built-in function that applys a function to the items in an iterable two-at-a-time, progressively combining them into a single result

<u>Iterable</u> : any Python object capable of returning its members one at a time, allowing it to be iterated over a for-loop

<u>Transforming Lists in Python</u> : how to create and add items in lists

<u>Using For-Loops</u> :

```
list = []   # empty list
for i in range(1,10):
    list.append(i)
```

<u>With map() objects</u> : can also do this w/ map()

<u>List comprehensions</u> : can just define list and contents at the same time

```
list = [expression for member in iterable]
```

<u>ex.</u> `squares = [n*n for n in range(1,10)]`

* can do this with conditionals:
`[char for char in sentence if char in "aeiou"]`

* can also nest these

Generator Functions: special f'n that returns a lazy iterator

lazy iterator: objects you can loop over like a list. does NOT store contents in memory

* can create these with generator functions and generator expressions

generator functions: uses yield instead of return
yield: value is sent back to caller but function is NOT exited
* state is remembered
↳ something todo w/ next()

Advanced Generator Methods: .send(), .throw(), .close()
.send() ⇒ send data to generator
.throw() → raise generator exceptions
.close() → stop generator's iteration

* can build data-pipelines with these *

How to Word Process with Python:
read pdf files → utalize regexes → export to excel/word → convert back to pdf
1. Read pdf doc with PyPDF2 or PyMuPDF packages
2. Utalize regular expressions
3. Export data to excel w/ Pandas dataframe from lists
4. Export from Python to Word w/ Python-docx package
5. Convert word to pdf w/ docx2pdf package

Lecture Notes: 9.16.24

HW2 Hints:

- Test on small/known file
- When in doubt, print it out
- Create small test directories
- Create small, nested test directories
- Use the known results from HW1

Installing Python Packages:

pip install XXX • old school

pip3 install XXX • new

sudo pip install XXX • system-wide install

Virtual Environment:

python -m venv [name]

* packages now only affect this environment

Regex Exs.

pikachu
bulbasaur
charmander
chespin
squirtle
meowth
togepi
oshawott
abra
jigglypuff

1. all of the strings → ^\w+$
2. only charmander and chespin → ^[ch]{1}\w+$
3. all words with 2 t's → ^\w*[tt]{1}\w*$
4. words that don't start with a vowel
   → ^[^aeiou]\w+$

Common Regex Tools:

Single Match [ ] Range vs. Single or Multiple Matches ( ) Group

.	^	$	?	*	+	|
any	start or not	end	0 or 1	0 or more	1 or more	or

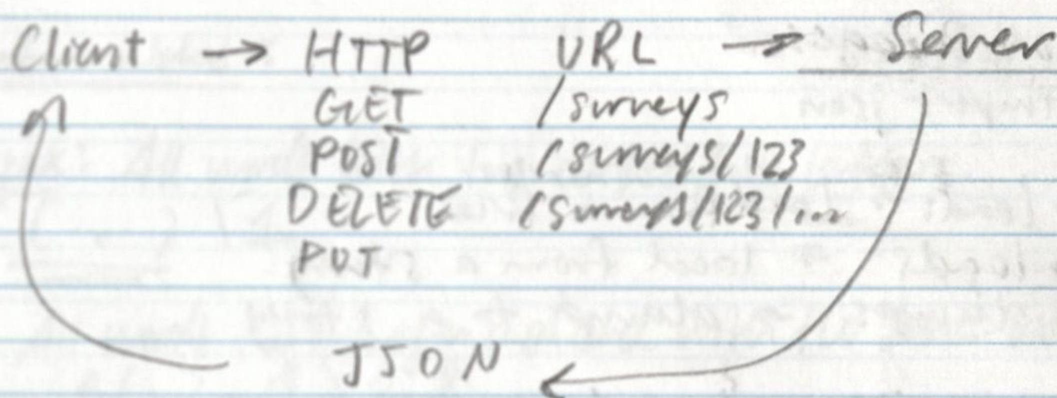
{ x } repeat x amount of times

: x : ranges / buckets of x

1. ^[:alpha:]$
2. ^(ch)[:alpha:] ← cannot use [ ] to capture 'ch' → only gets c or h
3. (tt)$
4. ^[^aeiou]

Web + JSON:

- HTTP, HTTPS
  - → Hyper Text Transfer Protocol
  - → +S adds security
  - → run over TCP/IP
  - → GET, PUT, POST
- REST
  - → REpresentational State Transfer
- Sites use API
  - → Application Programming Interface
  - → HTTP REST
  - → provide parameters, get JSON or XML



In Python:
- Fetch a web page (HTTP GET)
- Convert that to JSON

To Fetch:
- import requests package

import requests
⋮

JSON:
- JavaScript Object Notation
- Human Readable
  - → [ ] for lists
  - → { } for name/value pair

* can nest all of this
* CRLF not needed

Ex. Requests + JSON in Python in Lecture 08 slides

```
url = ...
params = { ... }
result = requests.get(url, params)
data = result.json()
...
```

JSON Package:

import json

- load → load from a file
- loads → load from a string
- dumps → dump to a string

XML - eXtensible Markup Language

- Free-form
  - → similar to HTML
- Elements
  - → <start>
  - → </start>

Lecture Notes: 9.18.24

Regex: All words with two consecutive letters
(.)\1

All words that begin and end with the same letter
^(.).*\1$

* use iHateRegex website to help debug these *

Email Crossing:
- how to fetch all CSE emails from CSE website

```
import requests
var = requests.get('https://cse.nd.edu')

var.text.find('@nd.edu')
    string.find()
```

JSON:

```
import json
f = open('file.json')
data = json.load(f)

for i in data['key']:
    print(i)

f.close()
```

load()
loads()
dumps()

XML:
- like HTML
- uses tags ad attributes

<?xml version ... ?>
<note>
<to> Love </to>
<from> Jove </from>
⋮

} Structure called a DOM "document object model"

ex, <start>, </start>

* can define any tags & cannot do this in HTML

YAML:
- YAML Ain't Markup Language
- very human readable

element:
- nest: data

import yaml
⋮

port:
- name: login
- port: 8080
⋮

* examples of the differences b/w JSON, XML, ad YAML in slides *

• best for configuration files (name + value pairs)

Ex.

```
import yaml

with open('file.yaml') as f:
    data = yaml.safe_load(f)

    print(data)
    print(data.keys())
    print(data['element'].keys())

    for key, value in data['element'].items():
        print(key, value)
```

* XML is only for older code <2010 *
  ↳ usually
Microsoft Word uses it

JSON is best for web applications

Functional Programming:

Imperative Programming:
- list of instructions
- step-by-step what to do
- sequential

Functional Programming:
- composition of functions
- done at any time in any order

FP is describing a problem to a mathematician while
IP is describing a problem to a robot

Big Ideas:
- always returns same output for given input
  - deterministic
- stateless (no side effects)
- order of evaluation undefined
- emphasizes divide and conquer

↳ will not change your input
instead give you a new object as an output

Map:
transform a list into another list

map(func, seq1, seq2, ...)

takes a function func and one or more sequences and applies func to the elements of those sequences

ex.
strings = ['3.14', '2.71', '1.0']
list(map(float, strings))
[3.14, 2.71, 1.0]

function float is applied to the sequence strings and result is stored in new list
- returns a map object
- must convert back into a list
- there should be a list for each parameter of the function

ex.

```
def addition(a, b)
    return a+b

results = map(addition, numbers1, numbers2)
print(list(results))
```

* can do this with number lists or string lists
* both lists must be the same size

Operators:

```
import operator
```

* has functions for all operations
* must use functions of these to use map()

Lambda:

short one-line function

```
map(lambda a, b: a+b, A, B)
```

(A, B → lists)

lambda arguments : expression

```
x = lambda a : a + 10
print(x(5))
15
```

} can assign them to variables

Lecture Notes 9.20.24

Lambda Ex:

sorted(People, key=lambda p: p.last_name)

* can do this with dictionaries or lists

Parse and Sort JSON:
* example in slides / terminal

Reduce: aggregate / collect items in a list into a single value

from functools import reduce
reduce(func, seq, initial)

Ex.
Sum:
reduce(lambda a,b : a+b, [1,2,3])

min:
reduce(lambda a,b: a if a<b else b, [3,5,4,1,2])

Filter: extract particular values from a list

filter(func, seq)

ex. filter evens
filter(lambda x: not x%2, [0,1,2,3,4])
[0,2,4]

List Comprehensions: another way to do things
ex. double items

```
map(lambda x: 2*x, [0,1,2,3,4])
[0, 2, 4, 6, 8]

[2*x for x in [0,1,2,3,4]]
[0, 2, 4, 6, 8]
```
← collapse vars

```
filter(lambda x: not x%2, list)

[x for x in list if not x%2]
```
← list comprehensions

Both:

```
[2*x for x in list if not x%2]
```

Syntax:

[change to apply for var in the list filter]

* map() and filter() return generators, not lists
& list comprehensions always return lists

map() and filter() use less memory

Iterators / Iterable:

```
# over List
for n in List:
    print(n)

# over Dict
for k in os.environ:
    print(k)

# over String
for c in "bill":
    print(c)

# over file
for l in open('path'):
    print(l)
```

Data Streams: sequences that allow access to next items in the stream

* cannot go back easily
* cannot index in

Common Iterators:

```
x = range(10)
r = reversed(x)
s = sorted(r)

import itertools
```

Generator: uses yield to create a generator that serves as an iterator

no return! just yield

Lecture Notes: 9.23.24

Testing:
- no test cases given on purpose
- look through files by hand
- make your own directory and test on that

HW3:
- group directory
- python virtual environments
- requests to web server
  - python requests
- JSON parsing
- sorting (lambda)
- filter
- statistics
- graph plotting (MatPlotLib)
- Word doc creating

Lazy Evaluation: w/ generators
- start function until yield
- return yield
- continue from yield when called again

Generator Expressions:
- convert list comprehension w/ generator expression by replacing [ ] w/ ( )

>>> (n**2 for n in range(4))

* List comprehension puts all values in memory
* Generator expressions only generate the one thing you need at that time
  → very lazy
  → faster, more efficient

Handout Ex:

print listComp >>> [0, 2, 4, 6]
print genExp
  >>> prints generator object
     not useful

print list(genExp) >>> [0, 2, 4, 6]
print list(genExp) >>> []

Handout Ex2:

```
def infinity():          generator function
    num = 0
    while True:
        yield num
        num += 1
```

advances generator function

```
gen = infinity()
for i in range(10):
    print('i:', str(i), 'generator', next(gen))
```

* works as expected: prints 0-9

Concurrency:
- composition of independently executing computations
- concerned about structure

Parallelism:
- simultaneous execution of (related) computations
- concerned with execution

FP: implicit concurrency
if we use FP, concurrency comes for free
→ makes it easy to parallelize

for item in stream: map(stream, item)
    compute(item)

inherently sequential   parallelizable

Concurrent Futures
concurrent.futures module
performs parallel execution

with concurrent.futures.ProcessPoolExecutor(4) ...
as executor
no argument: use every core your CPU has

Data Parallelism:
concurrent execution of the same task across
elements of a dataset
→ must be data independent

types:
- task parallelism: same data, different task
- data parallelism: same task, different data
- complex parallelism
    • different data, different tasks

- embarassingly parallel
    • scales wonderfully

Recall Processes:
- each process has its own memory block
  - → they cannot touch

Memory
- Stack
- Heap
- Code

Thread Centric View:
- shares memory block
- race conditions

Pitfalls:
- race conditions
  - → whoever runs first changes output
- deadlock
  - → multiple tasks stuck waiting for each other
  - → GIL helps protect against this
    - mutex
    - prevents 2 threads from doing python operation at the same time

I/O Bound → thread pool
*CPU Bound → Process Pool

Lecture Notes: 9.25.24

Lazy Evaluation:
1. Start evaluation of functions upon the first request for data and go until yield

x = generator() # not going to do anything yet
  # no request for data

Midterm Exam
- in class - 50 mins
- 1 sheet of notes (front/back)
  - → submit to Canvas for EC
  - → key points / recaps on Canvas
  - → HW2 / HW3 code
- content
  - → short answer
  - → MC + T/F
  - → programming - reading / writing (2)
- provided
  - → chinook sheet } Regex stuff
  - → datacamp sheet }
- practice exam on Canvas

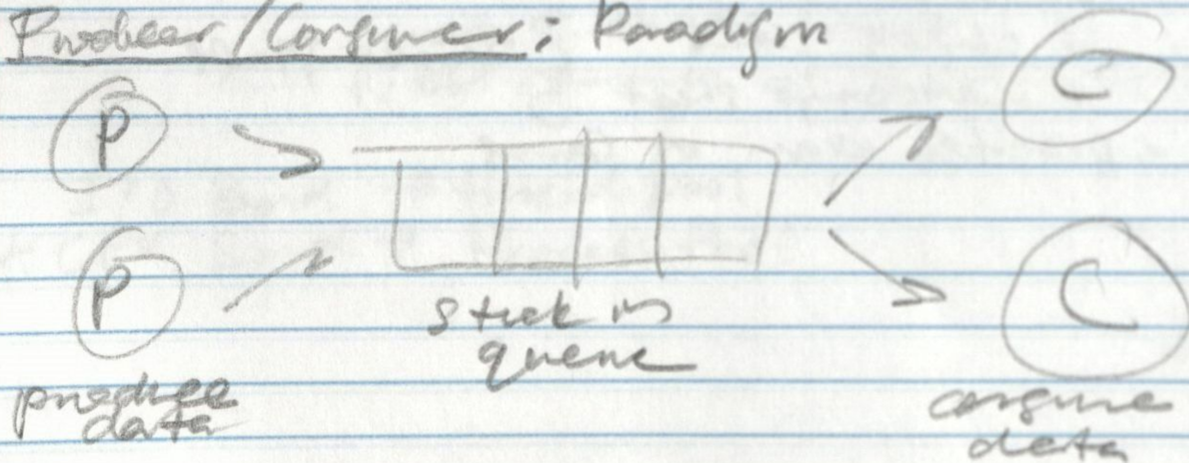
Concurrency Pitfalls:
- race conditions: multiple tasks compete for same resources non-deterministically
- deadlock: multiple tasks are stuck waiting for each other

* hard to write good multithreaded code
→ Python protects us w/ GIL

I/O Bound: use the ThreadPool
CPU Bound: use the ProcessPool
→ most of the time

Amdahl's Law:
The speedup of a program using multi-processes is limited by the time needed for the sequential portion of the problem
→ speedup limited by how much work we can divide in different processes
→ functional programming is big here

Producer/Consumer: Paradigm



Practice: Lambda

def filterData(entry, Month, Year, Interface):

filter(function, iterable)

Suppose we have list of data in theData, how to filter Month = 5, Year = 2024, and Interface eth0?

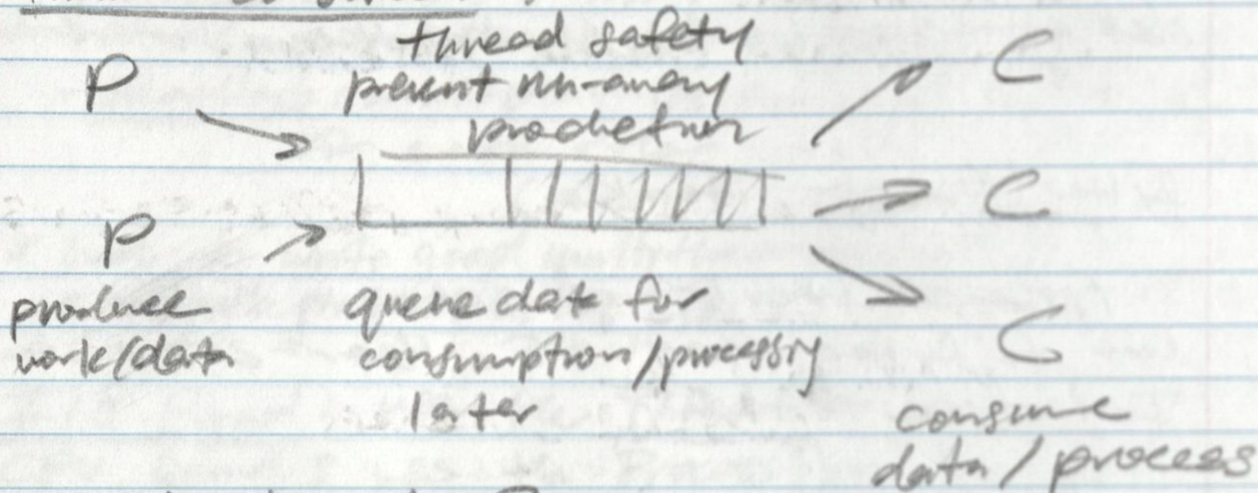
filter(lambda entry: filterData(entry, args.Month, args.Year, args.Interface), theData))

* will never have to do something like this on an exam
* helpful for HW3

Can I sort what comes from the filter?
Yes - convert to list and then sort

Lecture Notes: 9.30.24

Producer Consumer:



Why does it matter?

Cloud/Parallelization:
- Producers/Consumers/data imbalanced
- Bounded queue holds steady

Service-oriented architectures:
- heavyweight - single front end
- SaaS: Software as a Service
- go to an endpoint - do something big/interface

Microservices:
- develop the work - small/lightweight
- Embrace REST/HTTP
- decompose everything
- Local function/task optimization
- Plan/handle failures

+ Stitched together w/ REST APIs

FaaS: Function as a Service
- AWS Lambda
- Serve individual functions as needed

Serverless Computing:
- Decompose / abstract away persistent servers for devs
- Persistent state → DB
  → NoSQL / SQL
- FaaS is a subset of serverless
  → get from provider

Serverless computing still runs on servers

Unit Tests:
- testing cases of code

import unittest

Assert / Check:
- function that checks a statement

OOP: classes → mega structs w/ functions

Class + Unit Test ex. in lecture 14 slides

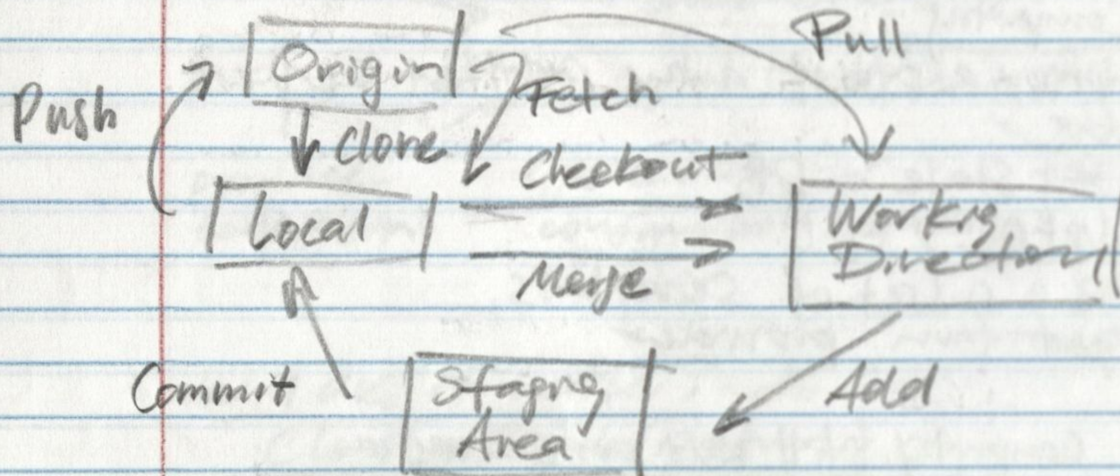
Python Unit Tests:

python3 -m unittest -v ↖ runs all of them
↳ mode

- can also have a dedicated test directory
- pyunittest
  → allows for more flexible naming
  → does not require classes

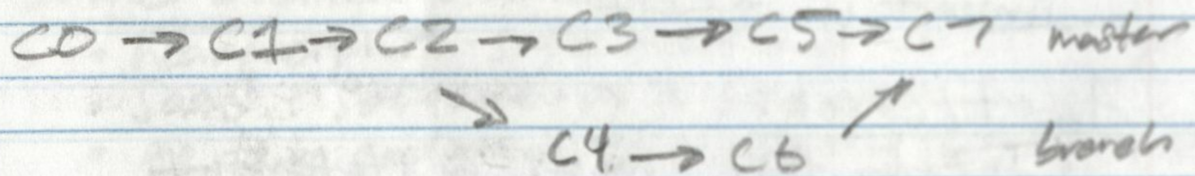
Exam 1 Review: 10.1.24

Git: free/open-source distributed version control system
- Journal, Time Machine, Shared Space



* Recorded commits in a repository stored in a DAG *

DAG: Direct Acyclic Graph



- Each commit has a unique hash
- Must checkout a hash since the working directory can only show one branch at a time
  - → this is done automatically w/ a clone
- Must eventually merge all branches back to master
  - → can raise conflicts

Pull Request: proposal to merge a set of changes from one branch to another

Python:
#comments
Importing Modules: bring in code beyond the base functionality of Python
ex. import json #import entire library
from plotdata import daily_averages
#import specific functions into global namespace

must use correct namespace

Variables = no specified datatypes
name = value

Objects: everything in Python is an object
- belongs to a type
- has methods (functions)
- has attributes (data)

Lists = dynamically allocated python array
- define with [], can be indexed
Tuple: immutable list
- define with ()
Dictionary: stores keys (immutable) associated with data (mutable)
- define with {}
- dict = {key1: value1, ... }
- Index with []
Methods = .keys(), .values()

Looping:
Lists: for x in theList:
Dictionaries: for theKey in theDict:
Multi-Value: for theKey, theVal in theDict.items():

Indexing:
x = theList[1:] #start a 1 and return rest of list
x = theList[-1] #access last element in list

Searching:
if theKey in theDict:
if value in theList:

Functions:
def function(argument=default):
| #code
| return variable #optional
- no return type, argument type, or end
  → just colon and indent
- can return multiple values
- have scope

Exceptions: programmatically caught errors
try:
| #code
except error as e:
| #error message

Range, Enumerate:
for value in range(1,4):
for index, value in enumerate(range(1,4)):

returns tuples for each index, value pair

Open a File:
f = open(file, mode):
modes: 'r', 'w', 'a'

Close a File: f.close(file)

Reading Files:
```
for line in open(path):
    line = line.rstrip()
```

Writing Files:
```
with open(path, 'w') as fp:
    fp.write(data)
```
→ deletes scope and automatically closes file when scope is left

OS Package: provides many filesystem functions
```
import os
```

System Package: reads command line input
```
import sys , sys.argv[]
```

Subprocesses:
Old: os.system()
New: subprocess module
Run: what we use
- forks a child process, runs executable, waits for result — blocking

Popen: allows for further customization — nonblocking

Process:
- Process ID (PID)
- Parent Process ID (PPID)
- Priority

- a process with a PID will fork itself and create a child process with a parent and PPID
- all parents know all their children vice versa
- parent can wait for child to complete

Regular Expressions (Refiex's):
import re
re.search (r'[regex]', '[location]')
re.findall (r'[regex]', '[location]')

*Sheet of expressions and metacharacters given on exam *

GRE: requires metacharacters be escaped
ERE: extended regular syntax
- doesn't require escaping

CSV: comma separated value
- first line may be header
- can use quotes to hold commas / special chars
- each line should have same # of fields
import csv
csv.reader (file)
csv. DictReader (file)
with open (file, 'r') as f:
reader = csv. DictReader (f)
↳ puts data into dictionary

CRLF: carriage return line feed (/n, /r, ~)

HTTP / HTTPS: HyperText Transfer Protocol
↳ adds security
REST: Representational State Transfer
API: Application Programming Interface

Client → HTTP → URL → Server

JSON ← (from Server, back to Client)

Requests = fetch data via HTTP

```
import requests
result = requests.get(url)
print(result.text)
```

JSON = JavaScript Object Notation
- [] for lists
- {} for name-value pair

* Like a list of dictionaries *

```
result = requests.get(url)
data = result.json()
```

- best for web applications

Package:

```
import json
```

- load(f) – load from a file
- loads(s) – load from a string
- dumps() – dump to a string

XML = eXtensible Markup Language
- similar to HTML
- has tags for everything
- mostly for older code – Microsoft Word

YAML: simpler JSON
- great for configuration files

Syntax:

YAML:

```
apis:
  -name: login
   port: 8080
  -name: profile
   port: 8080
```

XML:

```
<apis>
  <api>
    <name>login</name>
    <port>8080</port>
  </api>
  <api>
    <name>profile</name>
    <port>8080</port>
  </api>
</apis>
```

JSON:

```
{
 "apis": [
    {
      "name": "login",
      "port": 8080
    },
    {
      "name": "profile",
      "port": 8090
    }
  ]
}
```

Imperative vs Functional Programming:

Imperative: list of instructions, step-by-step

Functional: composition of functions, not particularly step-by-step (any time, any order)

- predictable (functions always return same output)
- stateless (no side-effects)
- undefined order of operations
- divide and conquer

Map: transform a list into another list

map (function, seq1, seq2, ...)

- returns a map object
  - → list (map (...))

Lambda: short nameless one-line function

lambda arguments: expression

sorted (people, key = lambda p: p.last_name)

Reduce: reduces list to a single value

from functools import reduce

reduce (function, seq)

Filter: extract values from list (into another list)

filter (func, seq)

List Comprehensions: syntactic sugar for map() and filter()

[change to apply for var in the list if (not) filter]

(map(): "change to apply for var in the list"; filter(): "in the list if (not) filter")

Map() and Filter() return a generator

List Comp. returns lists

Iterators/Iterable = any object we can loop over is iterable

- list, dict, string, file
- like sequences but only allow access to next item in stream
  - → next() accesses this

* not the same as lists — not subscriptable *
  cannot do iterator[0]

Common Iterators:

- x = range(10)
- r = reversed(x)
- s = sorted(r)

Generators:

- functions that yield a value instead of return
- only generate values that are called for
  - ⇒ a lot more memory efficient
- when function is called again it continues at the last yield

* special class of continuation

- can convert a list comprehension to a generator expression by replacing [ ] with ( )

Concurrency and Parallelism:

Concurrency:
- composition of independently executed computations
- concerned about structure

Parallelism:
- simultaneous execution of (possibly related) computations
- concerned with execution

Functional Programming – Implicit Concurrency
- if you use FP, you get concurrency as a side-effect

Inherently Sequential.
```
for item in stream:
    compute(item)
```

Possibly Concurrent.
```
map(compute, stream)
```

Concurrent Futures: performs parallel execution
```
import concurrent.futures
# compute pool of 4 processes
with concurrent.futures.ProcessPoolExecutor(4) as e:
    e.map(compute, stream)
```

Types of Parallelism:
- Task Parallelism: same data, different task
- Data Parallelism: same task, different data
- Complex Parallelism: different data, different task

? → * embarrassingly parallel

Process-Centric View:
- each process has its own memory block and they can never touch

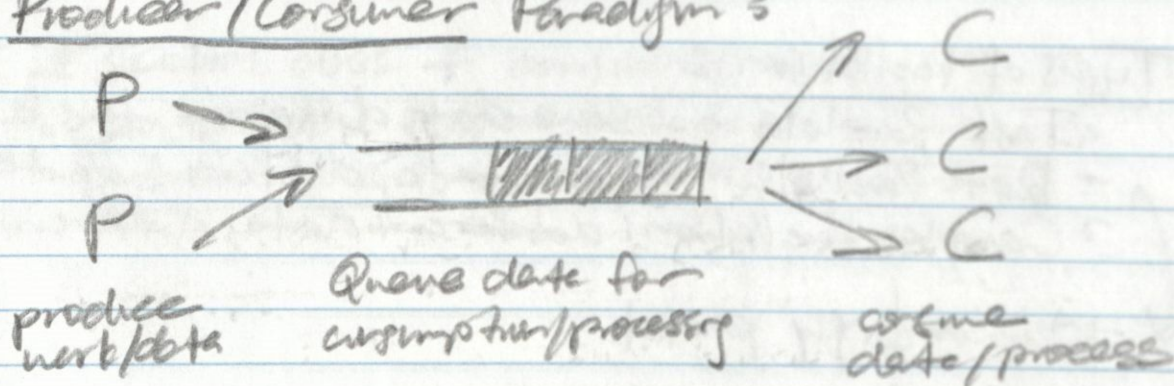
Thread-Centric View:
- all (as many as you want) processes share the same memory block

Pitfalls:
- **race conditions**: multiple tasks compete for the same resources non-deterministically
- **deadlock**: multiple processes waiting for each other

Amdahl's Law: speedup of a program running multiple processes is limited by the time needed for its sequential portion

* I/O Bound → thread pool
* CPU Bound → process pool

Producer/Consumer Paradigm:



REST API : Representational State Transfer API
- uses REST architecture to pass data / messages

Server : machine that provides a service to clients to connect remotely using the network
→ typically process(es) that listen for connections on particular network port(s)
→ URL = Uniform Resource Locator
https://nd.edu/ ...
↗ service ↖ host

Typical Ports :
SSH = 22
HTTP = 80
HTTPS = 443

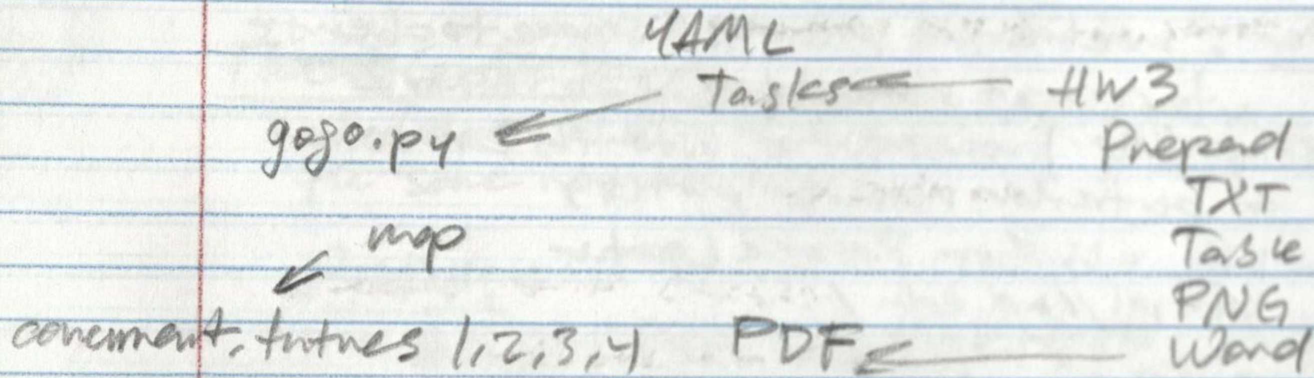
domain name server

* hostname is mapped to an IP Address via a DNS

The Cloud :
- numerous datacenters with hardware nodes
- has virtual machines hosted that customers rent

Python Servers :
requests → connect via HTTP, HTTPS
Flask : python web server
→ returns string of content (HTML, JSON)
MQTT : Publish / Subscribe to a Broker (server)
→ text based (string, JSON)
ZMQ : Zero Message Queuing
→ sends information via sockets
- within hosts (localhost), or b/w hosts
→ takes care of all threading
→ good for bindings Python and C

Lecture Notes: 10.4.24
Homework 4:



Read in a YAML File:
python yaml

Intro to Shell Scripting:
* shell scripting is just stitching together a bunch of Unix commands
  - Unix shell
  - read/understand shell scripts
  - understand Unix files / environment variables
  - write shell scripts

Where is ls Located?
# ask the shell
$ which ls — works for every command
# verify
$ ls /bin — commands usually in /bin

* can use absolute path of commands
./bin/ls

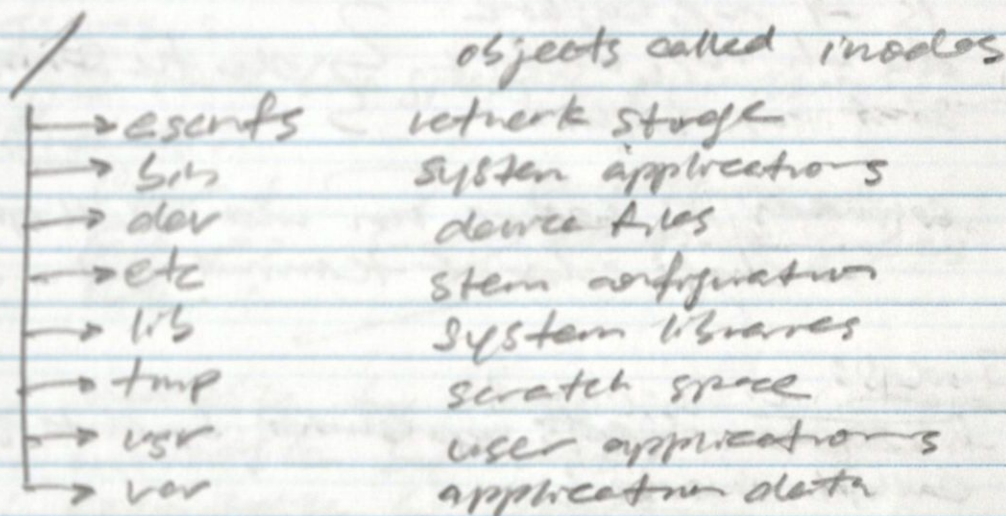
Man Pages:

man [command]

manual for linux commands

Hierarchy: Root

- files stored in a tree



* Unix is very good w/ files

Hierarchy: Home

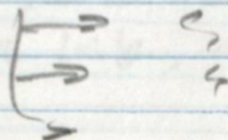
* normally, every user has a home directory
  → where user files are located

Reference with: ~/ ~username/ $HOME

hidden files: start with "." dotfiles

ls -a : list all files

escnfs
↳ home
  ↳ user



Paths:

Absolute Paths: begin with the root directory

$ /bin/ls

Relative Paths: based on the current location

$ ../../../../bin/ls

Data About Data: ~/.bashrc

ls -l ~/.bashrc
du -h ~/.bashrc
stat ~/.bashrc

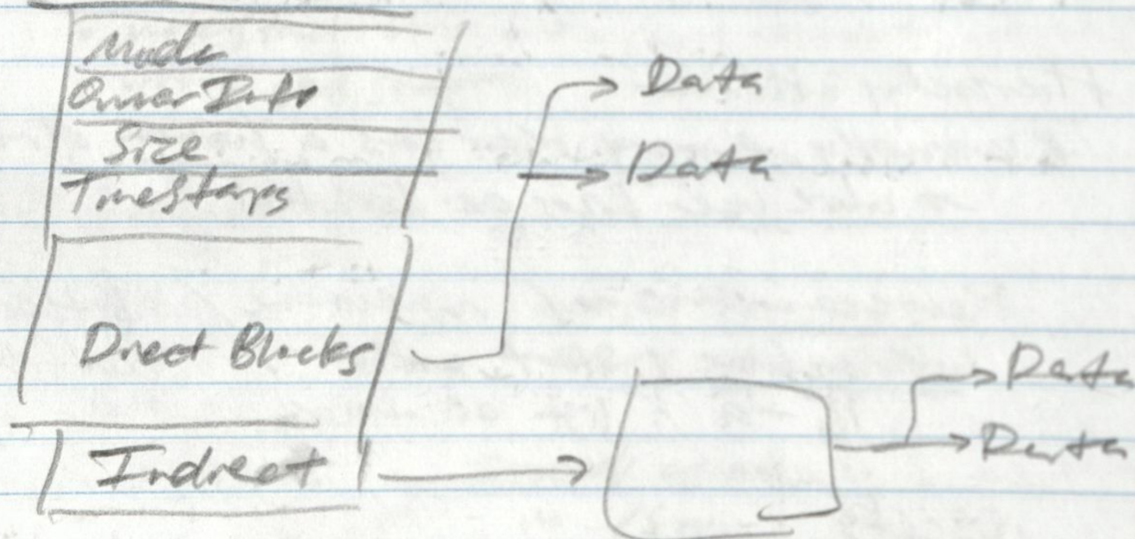
} do the same thing

commands in .bashrc run when shell opened
→ change color of terminal ect...

Inodes:

filesystem objects represented by data structure called inode:

Inode Structure:



* covered in OS

Attributes: ls -l
lists all metadata about files

metadata user group size ....

Restrict Files: chmod

Permissions:
All file mode have specific permissions
Buckets:
User:
Group: } permissions on all these sages
Other:

r: readable by class
w: writeable by class
x: executable by class

types: d, -, l, etc.
directory regular link

Set mode: use octal numbers (base 8)
numbers or symbols:
u=---, g=---, o=---
u+rwx, g+r, o+w
& can also remove access

Shortcuts/Links: creates a windows shortcuts
ln -sf [path] * file not deleted until all hard links deleted *

Hard Linking: associates filename w/ existing inode
Soft Linking: symbolic link, small file w/ pointer to another file

Lecture Notes: 10.7.24

HW4:
Flask
↘ https://student10.cse.nd.edu/hw04/....
port: 54151
↘ gogo.py

HW5:
- Branching
- UNIX Shell
- Files, Redirection

Linking:
Hard Link: associated file name with existing inode
- points to actual content
- data isn't deleted until all hard links removed
Soft Link: a small file pointing to another file
- sym link, "symbolic link"
- points to the name of content
- when data is deleted these stop working

Finder: finds files → filter: only .c files
$ find . -name '*.c'
↳ look in the current directory
↳ searches recursively?

Searching:

locate: searches database for files
↳ all files hashed on machine

$ locate ls

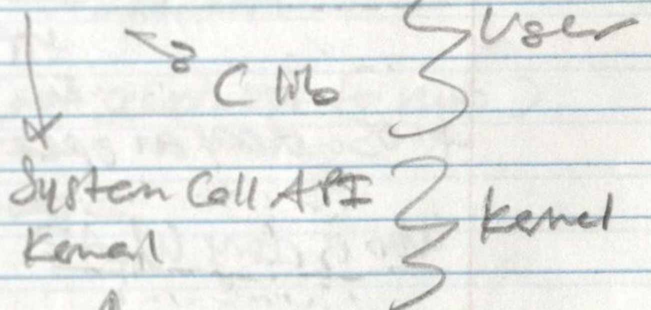
find: searches directory for files based on criteria

* locate much faster than find *

Piping: (|) sending the output of one command into the input of another command

Recall: Processes

- Process ID (PID)
- Parent PID (PPID)
- Priority
- Nice number



↳ what are you running?

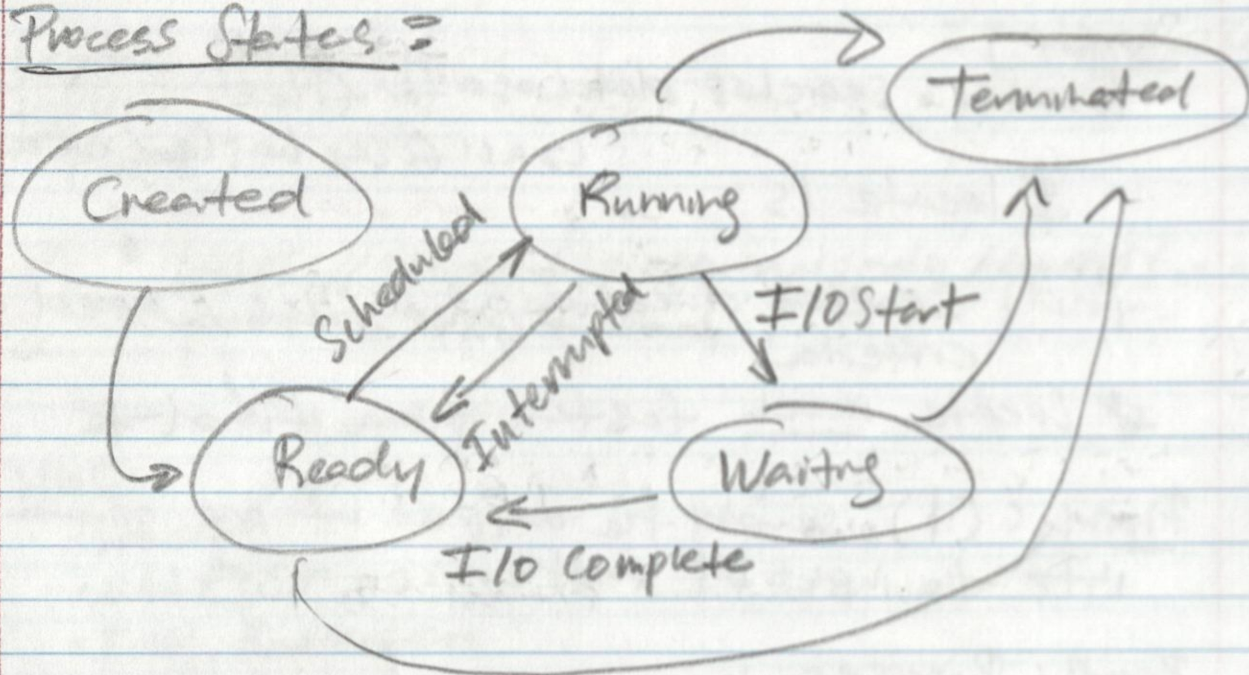
Unix: ps
ps -A -f
↳ all details
↳ all processes

top → gives all process info

Windows: task manager

Scheduling: OS picking what process to run and on what processor
→ Illusion of infinite processes
→ time sharing
↳ = number of cores

Process States:



Zombie State: Terminated process not yet cleaned up by OS

* OS does an operation every quanta/tick *

Who is doing What?

ps ux , top , ps aux
↳ user processes ↳ all processes

ps ux | grep match
↳ generalized reject processor

Niceness: higher the niceness, lower the priority
nice find /tmp
→ inverse nice → increase priority

Kill: terminate a process
cntr-C # if running the program
$ kill [PID] # from another terminal

Signals:

$ kill | -TERM | [PID] → process #
$ pkill | -HUP | watch
  ↳ process name
$ killall | -9 | watch ↗
  ↳ different signals for how to kill

Lecture Notes: 10.9.24

Background Processes: $ sleep 60 &

Jobs: $ jobs
  shows all background processes

$ fg - brings the most recent job from the background to the foreground

^Z - suspend process (not scheduled to run)

$ bg - show background process

$ kill %1 - kill job

* allows us to have multiple jobs in one shell

How to change identity?

$ su - switch identity (superuser)

$ sudo [command] - run as ...
  super user do ..

Elevated Privileges:
su - become root
sudo - run command as root

Process SUID/GUID:
allow user to execute a command as the owner or group of a file

cd /bin/cat /tmp/${user}-cat
chmod u+s /tmp/${user}-cat

* usually avoided — system vulnerabilities

* what is REST API?

I/O Redirection:
stdout/stdin : represented as #'s
→ people are lazy

Mechanism:

* STDIN (0)
* STOUT (1)
* STDERR (2)
} any of these can be redirected

* any extra created files start with 3 since 0, 1, 2 already taken

Save Result of Command:

redirect output:

$ command > output

ex. $ ls -l > out-ls.txt

* output to file is much faster than outputting to console → doesn't have to print chars

Pipe Stdout to tee:

$ command | tee output

ex. $ ls -l | tee out-tee-ls.txt | less

↓ ↗ allows output to file and input into different command

Pipe vs Redirection:

Redirection: → appends to end of file

<, >, >> reroutes command's I/O to and from a file

Pipes:

connects STDOUT of one command into STDIN of another command

use "|" - pipe symbol

* when we run command in shell, command forks itself and runs in background

Syntax:
* table of redirection/pipe commands in slides *

* Std out and error > file 2>&1
&> file
& indicates addresses like in C

Spam other terminals:
$ write [userID]
HI - waits for stdin
^d - EOF

Pipe stdout to write:
$ echo HI | write [userID]

* have to be on same machine for this to work *

Every terminal has a TTY file:
$ tty - get current command

Get all terminals:
$ ls -l /dev/pts | grep ${USER}

Write directly to terminal: → find w/ regex
$ date > /dev/pts/4
→ all devices on machine (USB devices, terminals, etc.)

How to ignore error messages?

$ [command] 2> /dev/null

trash can "black hole"

* other examples of find / direct in slides &
* redirect stdout and stderr to separate files

$ find ~ -type d > output 2> errors

0 STDIN
1 STDOUT
2 STDERR

Devices

/dev/null # black hole
/dev/zero # zeros
/dev/random

* works since "/" take to root directory
Different from "./" used elsewhere
relative path

Lecture Notes: 10.11.24

Networking:

The Internet:

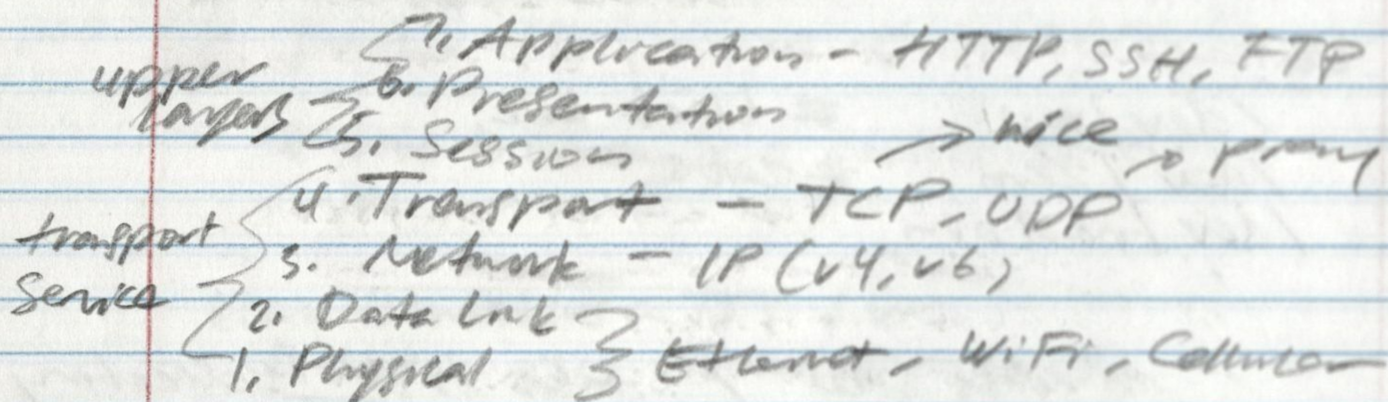
loose, unconstructed, chaotic, ad hoc collection of networks, bound by standards

- designed to handle failures
- network of networks

* connect computers to talk to each other

Rule: utility of the network: N^2

7 Layer Network Model



* 7, 4, 3, 2-1 bundle only important ones

* more info in slides *

3: handles routing through packets
- IPv4, IPv6, routers

4: exchanges messages through process-to-process channel
- UDP, TCP

What is my IP Address:
    ifconfig   # old
    ip addr   # new

one machine can have multiple IPs

NAT: network address translation
    - translates private IPs to public IPs

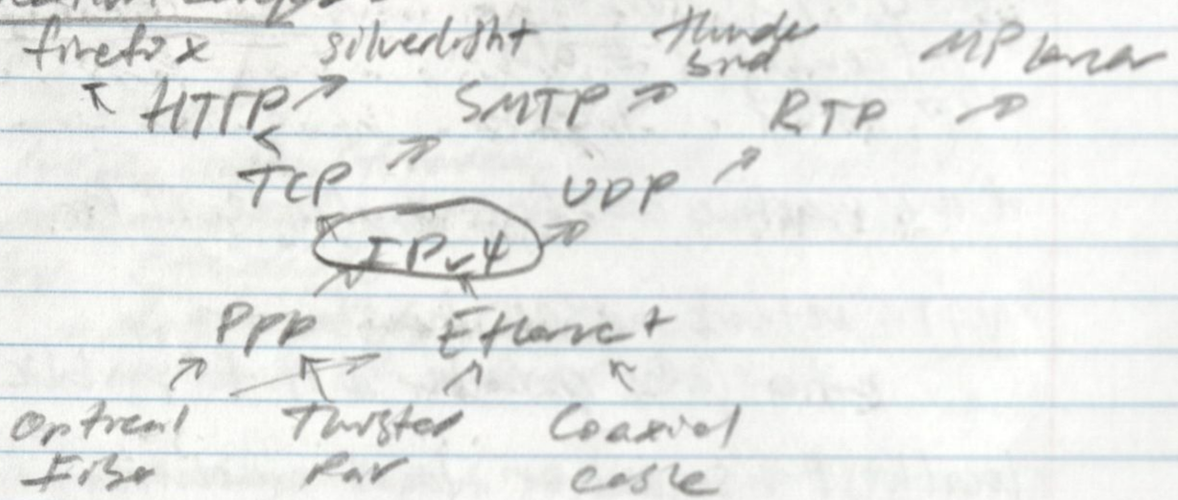
localhost: server on local machine
    - connect to yourself

Router: connects 2 different networks
Gateway: same as a network, also translates
    b/w one network system and another

TCP/IP:
    TCP: transmission control protocol
        - provides reliable two-way stream
    IP: Internet Protocol
        - provides unreliable connectionless ...

What Is Available
    netstat -tulnp   # old
    ss -tulnp   # new

Application Layer:



* hourglass ... IPv4 connects everything

what is ND's IP Address?

dig nd.edu
host nd.edu
nslookup nd.edu

↓ newer

DNS: Domain Name Systems
- done w/ ISP: Internet Service Provider

Download from the web?

curl http://...
wget ~~http~~://...

curl: dump to stdout
wget: download file

How do we measure bandwidth and latency?

Bandwidth: capacity - how fast / speed

Latency: delay - how much time to go to server and back

↳ ping, "round trip time"

wget - bandwidth

ping - latency

traceroute - trace networking route

SSH / SFTP / SCP / RSYNC:

SSH - login to remote shell

→ or run commands

SCP / RSYNC - file transfer b/w machines

Port Scanning: certain services are associated w/ particular ports

SSH - 22

SMTP (email) - 25

HTTP - 80

$ nmap -v -Pn host

scans all ports to see which ones are open

Shell Scripting: batch sets of Unix commands

* can usually be done in Python ⇒
  ⇒ but sometimes we need shell scripts

* most of the time we will read and troubleshoot ⇒

* Makefiles are shell scripts! ⇒

Bourne Shell: /bin/sh
- original Unix shell
- most primitive/portable shell
- focus for this class

```
$ cat > hello.sh
#!/bin/sh        ← she-bang: interpreter that you should run
echo "Hello World"
^D  ←  EOF
```

creates a file with that content

```
$ sh script.sh        run script with explicit interpreter
Hello, world!
```

```
$ chmod +x script.sh        make the script executable
$ ./script.sh
Hello, world!
```

→ add executable permission for everyone

* unix does NOT care about file extentions

Variables:
every process has an environment that
has name, value pairs
variables

```
env               # prints values of all environment variables
echo $EDITOR      # print value of EDITOR
                    variable

$EDITOR=eclipse       # assign eclipse to variable
$echo ${EDITOR}       # print var
$ export EDITOR       # make var visible
                        to sub processes
                        (global vars?)
```

Lecture Notes: 10.14.24

```
$ $EDITOR ~/.vimrc   # use var in command
```

vars are case sensitive → usually in ALLCAPS

Debugging:

```
echo      # print statements
```

use set:

```
set -e     # exits as soon as a command
             fails
set -x     # print the expanded command
             before it is executed
set -v     # print any input that is read
```

Capturing Output:

` ________ ` captures output

→ single "`" character by "~" in top-left
not "'" or "'"

$(____) does the same thing

ex. stat `which ls`
echo Today's date is $(date)

* no spaces b/w equal signs *
var=value

Conditionals:

```
if stat NEO > /dev/null 2>&1; then
    echo "Found him!"
elif ....; then
    echo
else
    echo
fi
```

evaluated lazily

* Exit status *
successful — exit 0
failure — exit nonzero

0 - true → opposite of C

Test / [:

can test conditionals → linked to [ ]

ex

```
if test -d /tmp; then
    echo "/tmp is a directory and exists!"
fi
```

or.

```
[ -d /tmp ] && echo "/tmp is ...  "
```

or. (-o → or flag)

```
if [ -d /tmp -o -d /var/tmp ]; then
```

Matching Patterns:

[{ * wildcard, | or }

```
case $SHELL in
  */bash|*/ksh|*/zsh)
    echo "POSIX"
    ;;
  :
  :
  *) #default statement
    echo
    ;; # break;
esac
```

Loops:

```
for var in list; do          while command; do
    body                         body
done                         done

for i in a b c; do           while :; do
    echo $i                      printf "%s\r" "$(date)"
done                             sleep 1
                             done
```

Program Arguments:

Access individual command line arguments:

```
echo $1 $2 $3 $4 ...
```

Access all arguments (in a list):

```
echo $@
```

Access name of script

```
echo $0
```

Access number of arguments

```
echo $#
```

Lecture Notes: 10.16.24

Options for testing w/ files:

- -e exists
- -d is a directory
- -f regular file
- -s empty
- -r readable
- -w writable
- -x executable

```
if [-e NETID ] then
    echo ...
fi

if test -f NETID then
    echo ...
fi
```

* more options for testing in slides *

= string is equal   -eq   numeric equal
!= not equal   -ne   numeric not equal
-n not empty
-z empty

Reading from STDIN:

read command → prompt user and read in input

```
while true; do
  read -p "Quit [Y/N]? " yesno
  case $yesno in
    [yY]) break ;;
    *) continue ;;
  esac
done
```

```
while read line
do
  echo "$line"
done < "${1:-/dev/stdin}"
```

if given argument 1 use that or default to stdin

* in POSIX the last line must be an empty line *

```
while read line || [[ -n $line ]]
  :
```

→ need 2 conditional tests here

ex.
- search in location provided as first argument
- find biggest source file (.cc, .h) in directory
- print out file name
- Hints
  - find to identify file
  - loop to go through files
  - size
    - stat
    - wc -c   bracket expansion

#!/bin/sh   *.{.cc, .h}

find location pattern

Lecture Notes: 10.18.24

HW6:   test

HW2
↳ unit tests   .good
shell   hw6 searchsrc.py   .bad
hw6 searchdir.py

Functions: curly braces

```
function () {
    test -f "$1" -o -d "$1"
}
```

no def

* No scoping - all vars are global

function /path ← call the function
{ no parentheses

Brace Expansions: Python range() equivalent

```
echo foo{c,cpp,h,hp}
echo {a..z}

for NUM in {000...005}; do

done
```

Scope: all variables are global by default
export variables?

Arithmetic: only integer arithmetic
no floating point values

```
echo $x + $y = $((x+y)) ...
```

Sub-Shell: can combine multiple shell statements into a sub-shell - acts like another process

```
(
    code
)
```

Signals:
trap: catch signals

```
trap "cleanup; exit 1" INT TERM
```

run a different exit signal
instead of a default one

Lecture Notes: 10.28.24

Python type hints:

→ hint : datatype

doesn't actually stop user from using a different datatype just gives a hint of what should be there

Pipeline: Assembly line
Unix pipeline — computational assembly line

process 1 | process 2

output fed into input

Powerful Pattern: * in shales *

Signals:
can catch signals w/ trap command

trap "function; exit 1" INT TERM
trap "function; exit 0" EXIT

3 seperate signals

INT : ^C → interrupt process
TERM : kill → terminates process
EXIT : code normally finishes ⇒ exits gracefully

kill -9 destroys any rouge processes

Grep and REGEX's:

grep -i "..."
grep -E → extended regexes

Filter: tr
translates one set of characters to another

$echo 'turn this city' | tr 'abc' 'xyz'
yurn this zity

tr 'a-z' 'A-Z'   lower to upper

tr -d '[:space:]'   removes all spaces

Filter: cut
extracts portions from each line of text

-d = delimeter

cut -d ':' 1   ← like python .split()

* sed and awk next *

Lecture Notes: 10.30.24

echo 'string literal $1' ← treated literally

echo "string $1" ← properly puts in the variable

```
if [ $# -ne 3 ]; then
fi
```
↑ if arguments not equal to 3

* do not need shebang
  - but cannot chmod -x file.sh
  - need to:
    - $ bash file.sh ← interpret it like python3
    - $ sh file.sh

Zombifortune.sh

```
#!/bin/sh

fortune | cowsay
dotrap () {
  echo "Brains"
  fortune
}

trap "do-trap" INT
               TERM

while      ; do
      read -p "FN" $fn

done
```

* DOS2Unix:

tr removes '\r' character

tr -d '\r' < dos.txt > unix.txt

Filter / Transform : sed

sed allows us to modify streams of text

$ cat /etc/passwd | sed -E 's|/var/lib|/tmp|g' ← global replace
| grep tmp

Egrep → -E; s → substitute mode; /var/lib → to find; /tmp → to replace; g → global replace

* more on this in slides *

Lecture Notes: 11/1/24

Environmental Variables : $ env

Recap : sed - allows us to modify streams of text

* lots of examples of this in slides

Filter : awk - powerful pattern matching language

* examples of this in slides

Profile : startup file when you initiate a new shell
- executed at user level
- executed at login for that session

* different shells : profile files are named differently! *

Lecture Notes:

11.4.24

#!/bin/sh

for TheFile in `find "$1" -name "*.txt" -type f`;
do

- find 1st argument
- looks for names of files w/ .txt
- run this command
- only get files

file
file
file
file
} returns a list

echo "... $TheFile" → prints each file

NewFile=$(basename $TheFile | cut -d '.' -f 1).csv

- run this command
- gives name of file
- pipe into
- delimit the file by '.'

..\...\.\test.txt → [test] . [txt]

-f 1 only returns this part

feeds it all into .csv (appends .csv)

tail -n 38 $TheFile > $1/csv/$NewFile

- take last 38 lines of this file
- magic number "just works"
- place contents into this directory (must exist)

Better way todo this:

sed 1,9d $1 | head -n 10 > badsite-nocmnt-10.csv

skip some lines — operate on this file — pipe — first 10 lines — write into this file

→ sed → stdout

(skip over first 9 lines)

1,9d : deletes lines 1-9 (files start on line 1)

Periodic Job Scheduling:

- crontab → Any user, any hour, any minute
- anacrontab → Superuser flexible start times, can be deferred

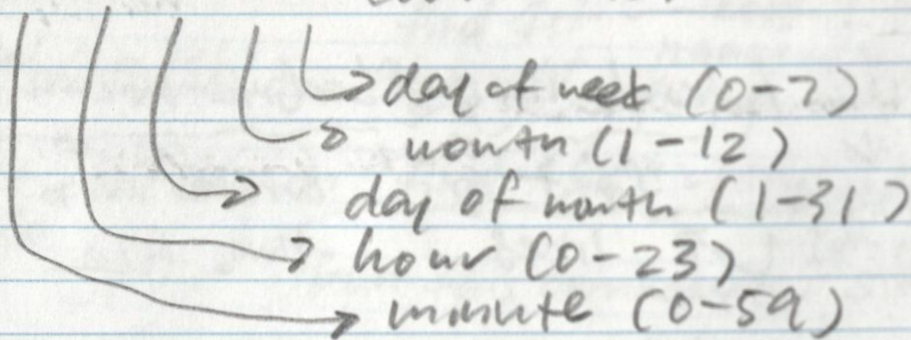
ex. clean out temp directory every day at 3AM

etc - system config files

$crontab -e edit the crontab

$crontab -l list the crontab

specific # : do at that time
list of #s : each of those times
*/# : use as an interval



cron.deny, cron.allow, etc ...

__Startup Scripts / Services:__
how to lanch processes on startup

__old__: init.d with rc.d
- rc.3: denotes what run level to start and stop at
- init.d: master script

__new__: systemd
- systemctl to start/stop/reload services

start (S), stop (K)

Lecture Notes: 11/6/24

Compiling: Program Execution
- computers can only execute low-level machine code

#include <stdio.h>

← looks to include environment variable /usr/include ...

% gcc -o hello hello.c

← creates executable in binary named "hello"

% ls -l hello

← gives information about executable rwx

% hello error: command not recognized

% ./hello tells bash where the executable is located

. look in local dir
/ look for file in there

% cat hello treats hello as a text file and tries to output it

Compiling: Assembly Language

- can use textual mnemonics to represent instructions and assemble them into machine code

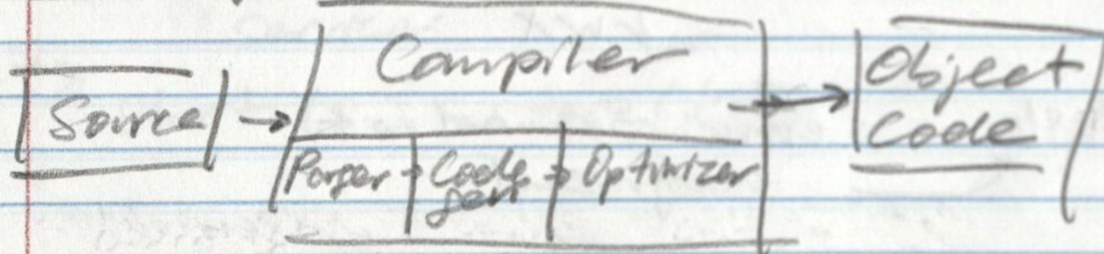
- Issues
  - Expressiveness
  - Portability

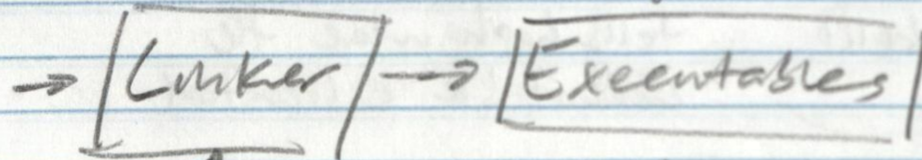
* Also, assembly is typically system dependent
  → so writing in C and compiling to assembly is useful when needed

```
% gcc -S hello.s hello.c
```

-S ← compiles to assembly

Compiling: Pipeline



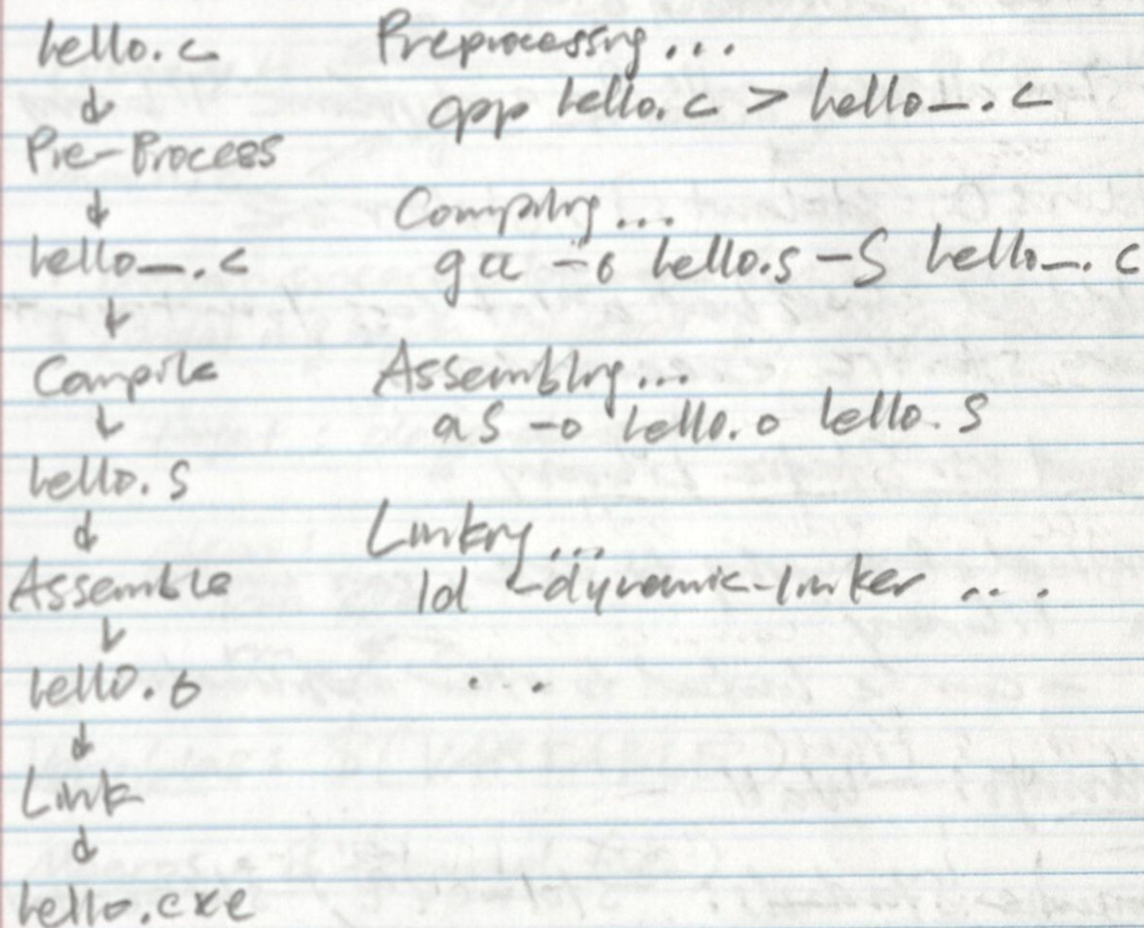


Libraries

Linker
- combines object code w/ libraries and creates .exe

Compilers:
- syntax analysis
- semantic analysis
- generates optimized object code

% gcc -v -o hello hello.c
↳ verbose



Default: GCC produces dynamic executables
- leans on system libraries

$ ldd program
↳ lists libraries used in program

-static flag: adds all libraries to executable
- larger file size

ldd: ldd fire
- prints shared object dependencies

strace: strace ./hello

- shows all system calls for a dynamic library

stdin: 0, stdout: 1, stderr: 2

* ldd and strace has a lot less footprint on static executables

Shared vs. Static Library:

instead of compiling to exe, can make a library
→ can be linked to other applications

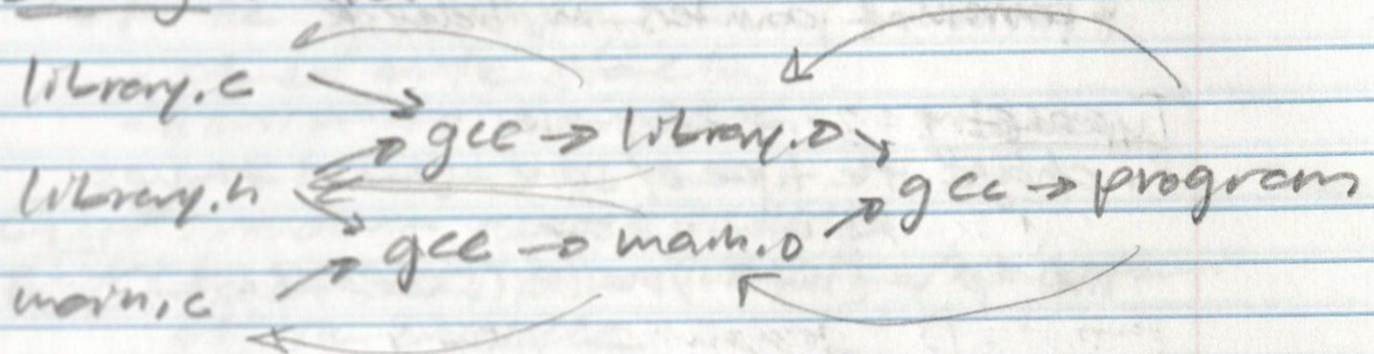
Warnings: -Wall

Language Standards: -std=c99, -std=c++11

Optimization: -Os, -O2, -O3

Header and Libraries: -Ipath, -Lpath

Building: Makefiles



- Domain specific language for DAGs
- Nodes in graph linked by dependencies

target : dependencies

clean :
    rm exec
    rm *.o

Variables: $(VARIABLE)

Macros: $ (command file)

Rules: TARGET : SOURCE
    COMMAND

Name: Makefile

ex of basic Makefile in slides

Recap .c
.o
.so
.a

Pointers:
* review of pointers in slides *

Typecasting:
· change the type of same-sized things

int *p = (int *)malloc(sizeof(int));

Lecture Notes: 11.8.24

Pointers:

pointer -> base + offset

↳ as much space as a memory address in a system
- 64 bit system -> 8 bytes

2^{10} Kb
2^{20} Mb
2^{30} Gb
2^{40} Tb
2^{50} Pb

offset: in the type the pointer points to

N * sizeof(type)

*(pVal+0) == *pVal == pVal[0]

Hex: 0x|20|00|

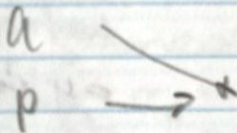
0-15 ↑ byte
0-F

sizeof(void*) == sizeof(int*)

Arrays: fixed block of contiguous memory of the same type
- stored on the stack
- similar to pointers

Good: random access, tables
Bad: fixed size, poor insertion/deletion of the middle

```
int a[] = {5, 4, 7, 0, 1}
int *p = a;
```



ADDR	VAL
0xF	1
0xE	0
0xD	7
0xC	4
0xB	5
0xA	

*can use both pointer and array syntax on a pointer

Exam 2 Review: 11.12.24

Motivations:

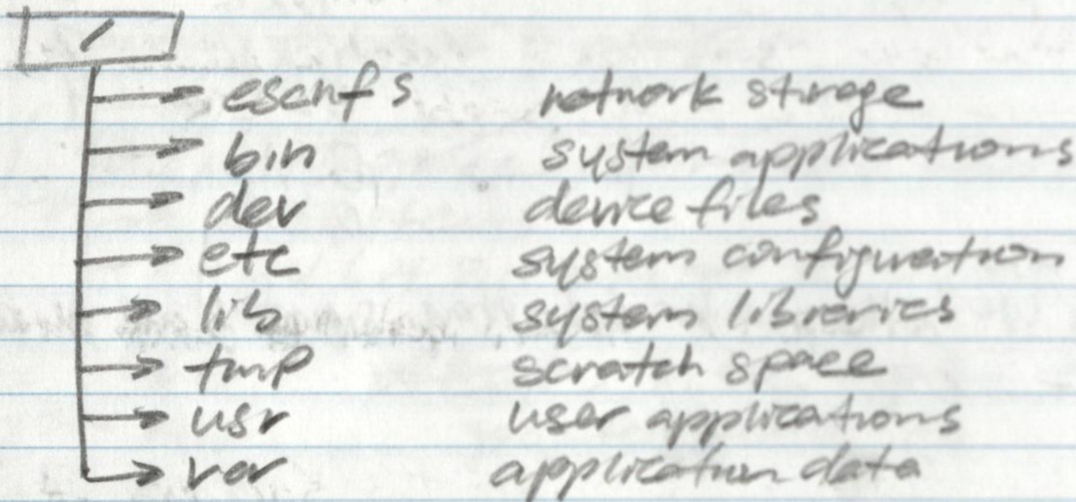
- stitch together Unix commands
- everything has a place

$ which ls #ask the shell
$ ls /bin #verify

Man Pages: $man [command]

- manual for unix commands

Hierarchy: Root



- structured as a hierarchical tree
- everything is a file!
  - directories are pointers to other files

Hierarchy: Home

escifs
└→ home
    └→ akurana2

Old files from AFS	→ AFS_Archive
Periodic Backups	→ backups
Coursework	→ esc-cources
Bash settings	→ .bashrc
Nano settings	→ .nanorc

- every user has a home directory where their personal files are stored
- directories and files w/ "." prefix are hidden

$ ~/
$ ~[username]/   } all reference home directory
$ HOME

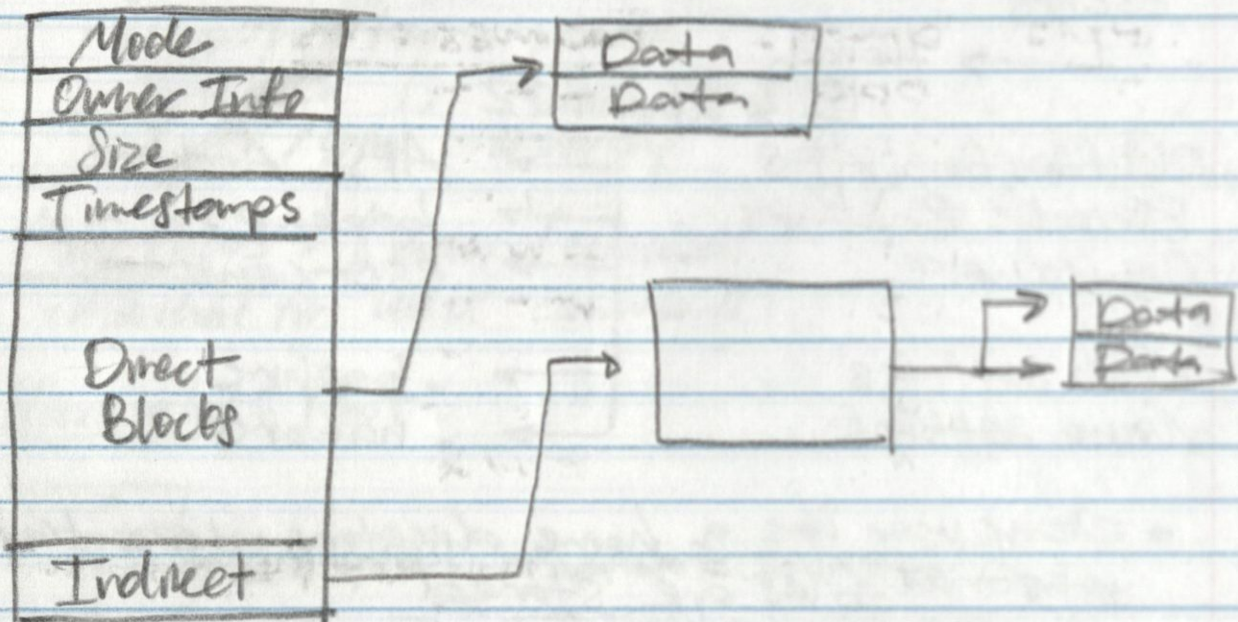
Absolute Paths: Start from the Root directory

Relative Paths: start from current location

Data about Data:

$ ls -l ~/.bashrc
$ du -h ~/.bashrc   } all do the same thing
$ stat ~/.bashrc

Inode: every filesystem object is represented by an inode data structure



Attributes: $ ls -l

1. Permissions
2. Number of Hard Links
3. Owner of File
4. Group
5. File Size
6. Last Modified Date/Time
7. File/Directory Name

Permissions: - - - - - - - - - - (10 bits)

type:
d: directory
-: regular
l: links

binary triplets for each class:

user, group, other

r: readable by class
w: writeable by class
x: executable by class

Can set permissions in Octal, Binary, and Symbolic form:

Octal:	Binary:	Permissions:
0	000	- - -
1	001	- - x
2	010	- w -
3	011	- w x
4	100	r - -
5	101	r - x
6	110	r w -
7	111	r w x

$ chmod +x # adds executable permissions to all groups

$ chmod u+rwx, g+r, o+w [file]
different uses of chmod

$ chmod 600 [file]
- can also use octal representation
- must append file to command as well

Shortcuts: $ ln -sf [path]
- Hard Links: associates file name with existing inode; points to contents
  - same data block as separate files
  - data is not deleted until all HL deleted
- Soft Links: a small file that points to another file
  - symbolic link
  - points to name of content
  - stop working when data is deleted

Finder: $ find . -name '*.c'

optional filter

→ searches in current directory

→ searches recursively

Search: $ locate ls

- faster as searches preexisting database instead of actual file hierarchy

View Processes: ps -A -f , top

→ full detail

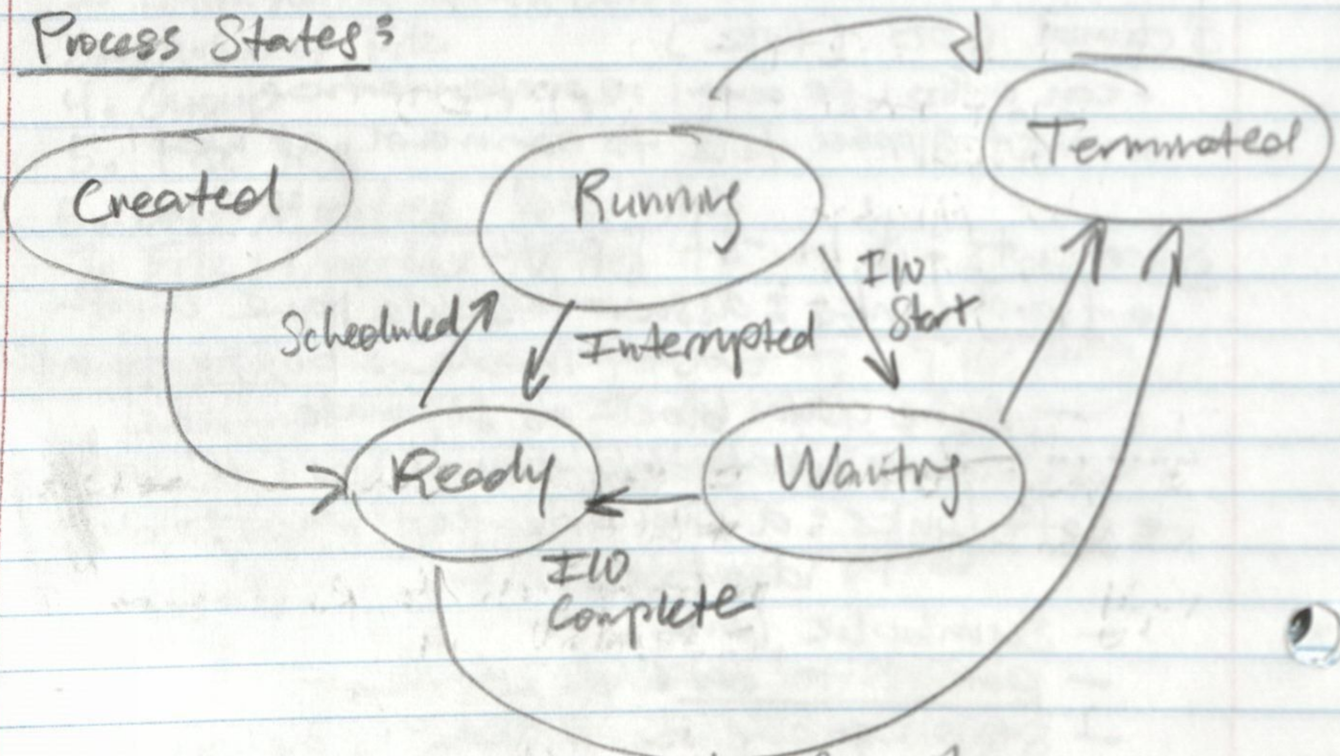
→ Show all processes on system

$ ps ux #user processes

$ ps aux #all processes

Scheduling: OS picking which processes to run at what time — illusion of infinite processing

Process States:



* higher the niceness, lower the priority

Termination:
^C of running the program
$ kill [PID] kill from another terminal

Signals:

-TERM	15	terminate process
-INT	2	interrupt process
-KILL	9	kill process
-HUP	1	hangup
-USR1	10	user-defined signal 1

$ kill [flag] [process]
↳ PID or name

$ ln original_file_path link_name *
-s means soft link, default is hard link

Processes:
- Process ID (PID)
- Parent Process ID (PPID)
- Priority
- Nice Number
- Terminal / TTY
- UID / GUID

Job Control:
$ sleep 60 # execute sleep in background
$ jobs # list jobs
$ fg # bring job to foreground
^Z # suspend job
$ bg # background job
$ kill %1 # kill job

Lecture Notes: 11.15.24

Types:

- char: 8 bits
- short: 16 bits — may be unsigned
- int: 32 bits?
- long: 32 bits

Better:

- uint8_t
- uint16_t

Typecasting: like sudo for C

- as long as sizes of types are equal
- will maps bits of one type to another

Ex.

Address	Value
0x1000	0x50
1	0x30
2	0x10
3	0x04
0x1004	0x20
5	0x05
6	0x10
7	0x02
8	0x04
9	0x09

```
char *pChar = 0x1004;
int *pInt = 0x1000;
long *pLong = 0x1006;
```

* **Effective Address:**

base + sizeof(type)·N

```
pChar = 0x1004          (pChar+1) = 0x1005
*pChar = 0x20           (pInt+1) = 0x1004
*pInt = 0x50301004      (pLong+1) = 0x100A
*pLong = 0x10020409     *(pChar+1) = 0x05
```

+1 moves sizeof(type) bits

Bus Error: tried to read in something larger than your type in a single address location

* pointer math on final exams *

Strings: NUL terminated Arrays
array of characters terminated by '\0'

- length of string not same as size because you need to account for NUL char

size_t : returns integer with the right amount of data (unsigned)

Iterating through a string: use a pointer to process one char at a time

string is over when NUL is reached

```
size_t str_len(char *s) {
    char *c;
    for (c=s; *c; c++);
    return (c-s);
}
```

keep going until 0 (NUL) reached

Functions:

```
strlen(s)
strcmp(a, b)      # compares a and b (0 means equal)
strstr(s, t)      # searches for t in s
strcpy(dst, src)  # from src to dst
strcat(dst, src)
```

strncpy() safer

Debugging: valgrind
- tool to detect memory errors
- Invalid Access: tries to write to memory it doesn't have access to
- Memory Leaks: does not properly deallocate/free heap memory

compile w/ -g flag
$valgrind --leak-check=full ./program

Debugging: gdb
- interactively trace, probe, and examine a process

- Step through code one line at a time
- Print variables
- Set breakpoints and watchpoints

$gdb ./program

* more commands for this in slides *

Lecture Notes: 11.18.24

const char *c
- points to const char type
- can change the pointer, not what it points to
- char * const is also a valid type

* examples of gdb commands in slides *

Note. must compile w/ -g flag

* know enough gdb commands to survive *

```
$ gdb ./program     # load program into gdb
(gdb) run           # run program
(gdb) bt            # display backtrace
(gdb) f 2           # examine stack frame 2
(gdb) p buffer      # print value of buffer var
```

* more commands in slides *

Memory Allocation / Processes:

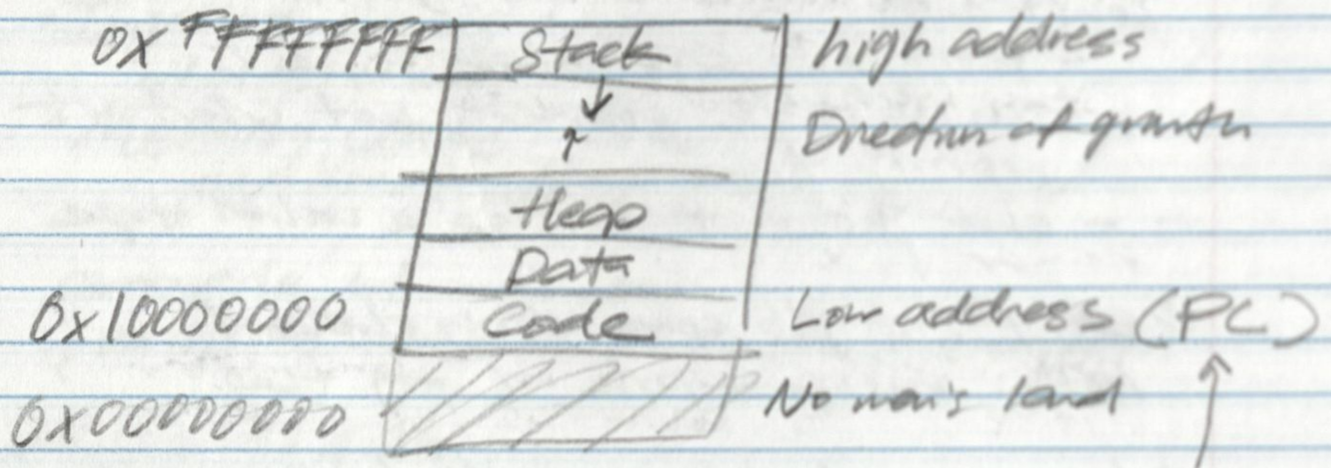
- unit of allocation (resources, privileges)

Memory Address Space: code, data, heap, stack

Kernel state: permissions, file descriptors

Execution content: program counter, stack pointer, data registers

Address Space:



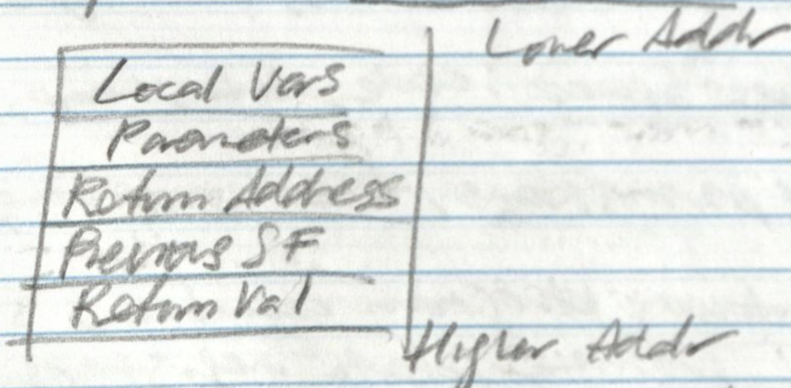
Remember:

1. Stack allocated values are high and grow down
2. Code values are low (cannot be modified)
3. Data and Heap are low and grow up
4. Stack and heap change location across execution, but code and data do not
5. variables/pointers go low to high

Stack: automatic allocation

Call Stack: region of memory to handle function calls

Composed of stack frames:



* vars are not initialized by default

Code and Data: Global, Static
- Global vars allocated in data
- Static makes vars be allocated in data
- String literals also in data

⋮

Header file

make clean } always do this after
make } editing a header file

Array too large → segfault
Too much recursion → segfault (Stack overflow)

!! Stack is limited, automatic, and not cleared w/ deallocated

Heap: Manual Allocation
- used for dynamically allocating memory
- use malloc() and free()

Memory Leak: loose track of memory
Segmentation Fault: invalid memory access

malloc(), free(), calloc(), realloc()
\# more on this in slides #

Advice:
- prefer stack over heap allocation
- avoid malloc() unless absolutely necessary
  - if you want data to persist b/w function calls
  - if data is very large
  - if data is unbounded

Lecture Notes: 11.20.24

* buffer overflow to exploit the stack ex in shellcode

Structs vs Unions:

Structs: composite data structure

```
struct point {
    int x;
    int y;
};
```

- no guiderails to prevent accessing other data
- all variables are public
- pointer to a struct will point to first byte of that struct

```
struct point p;
struct point p = {0}; // x=0, y=0
struct point p = {1, 2}; // x=1, y=2
```

Type Definition:

```
typedef struct {
    int x;
    int y;
} Point;
```

or

```
typedef struct point Point;
```

Accessing:

```
point.x   // access vars
point->x  // access and dereference
(*point).x // same thing
```

Union: polymorphic data type
views the same chunk of memory as different types → size of union is the size of the largest element in union

```
typedef union {
    char c;
    int i;
} Value;
```

kind of like typecasting!

In C/C++ members are signed by default
Can unsign them

```
int i=123;
unsigned int u=123;
```

```
#include <stdint.h>
    uint8_t s =123; // unsigned 8-bit
    uint64_t u=123; // unsigned 64-bit
```

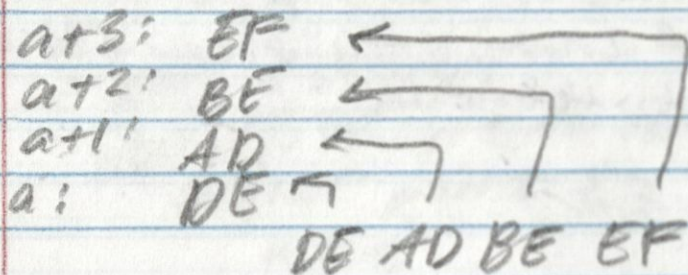
Endianness:
Big Endian: decreasing numeric significance w/ increasing mem addr

lower | MSB   LSB | higher   ARM

Little Endian: increasing significance w/ increasing mem addr

lower mem | LSB   MSB | higher mem   x86

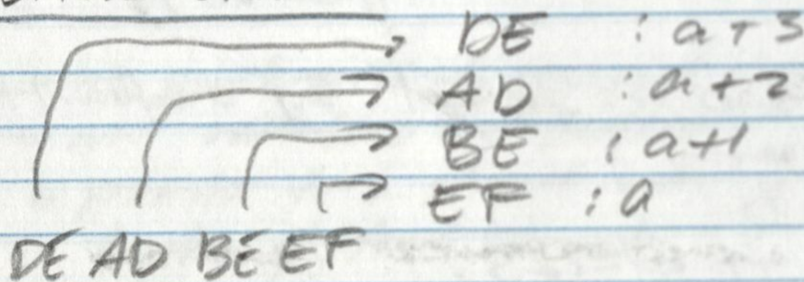
Big Endian:



order of bits do not change!

order of bytes do change

Little Endian:



Aside: Magic Numbers

numbers that just work for some reason

```
#define MAGIC_NUM 3
```

define these instead of using numbers everywhere

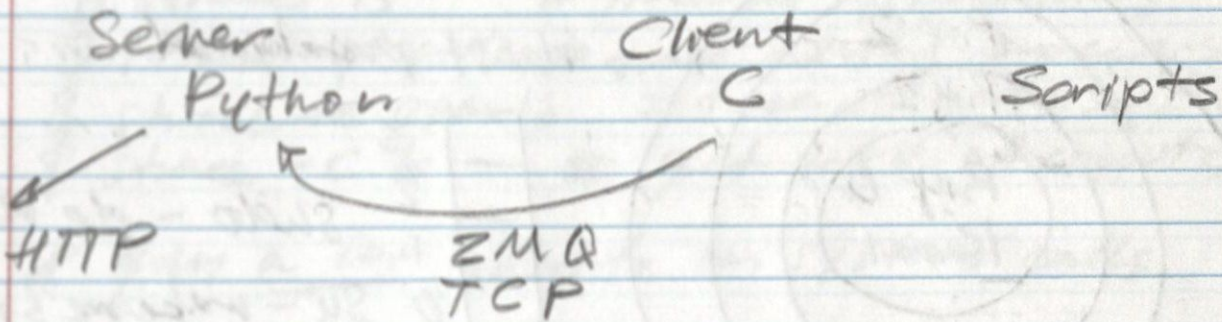
Review: Data Structures

Data Structure	C++	Python	C
Sequence	vector	list	N/A
Fixed	array	tuple	array
Associative	map	dict	N/A
Membership	set	set	N/A

* much more about C data structures in virtual lectures *
→ but mostly covered in DSA

Lecture Notes: 11.25.24

Homework 10:

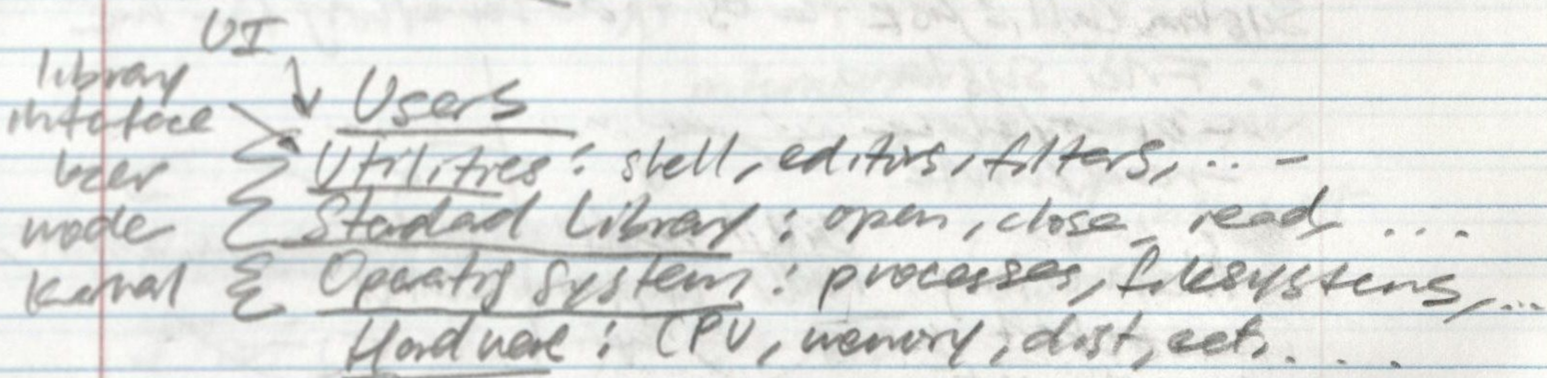


Bluetooth Beacons

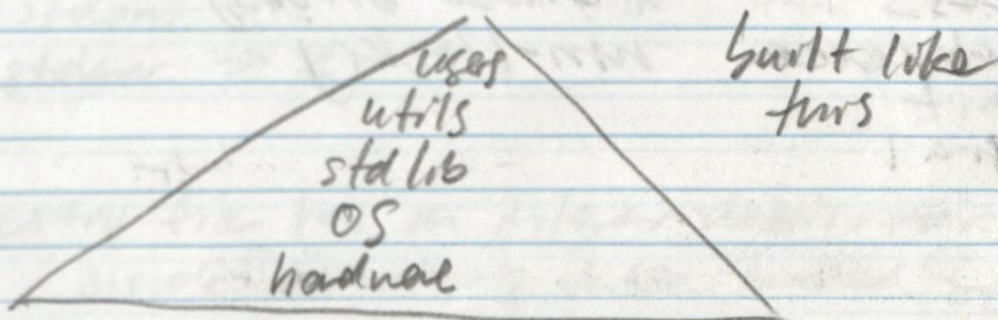
System Calls:

- makes sure shared resources (memory, processing, disk space) are safe
- all system calls managed by the OS

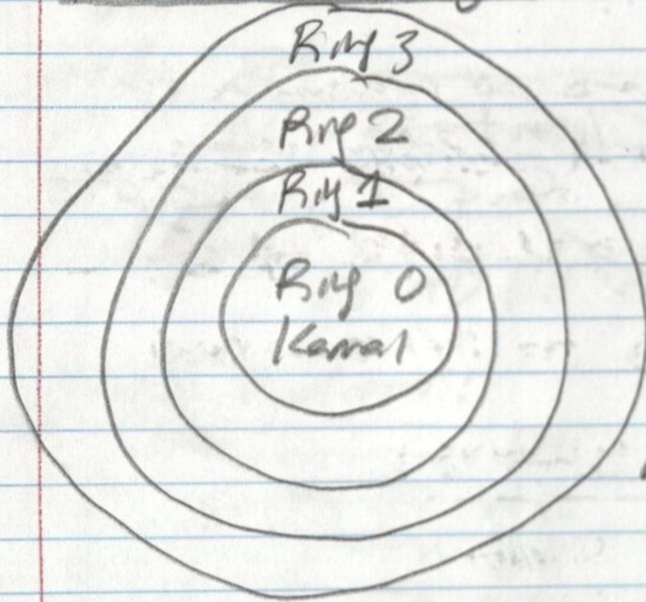
House of Cards:

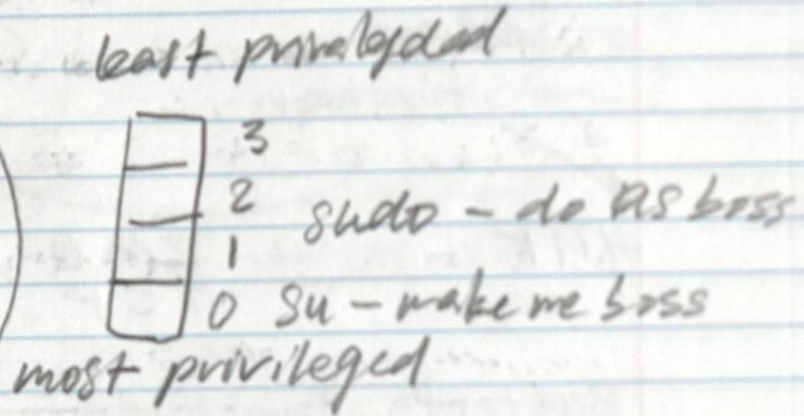


* better hierarchy in slides



Protection Rings: security model

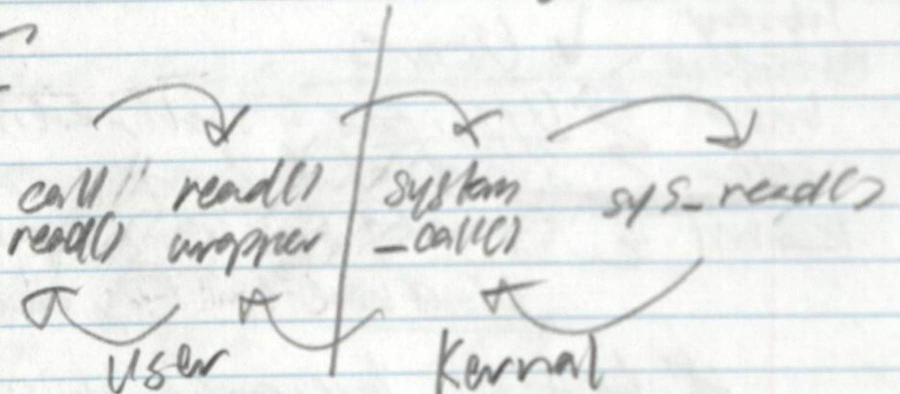




- invalid access to resources from higher rings can be handled by programs in lower rings
- most modern systems have 2 rings
- usage of VM introduces > 2 rings

System Call: ask the OS to do something for me

- File systems
  - open/close
  - read/write
  - ⋮
- Networking
  - socket
  - connect
  - ⋮
- Process
  - fork/exec
  - wait
  - signal



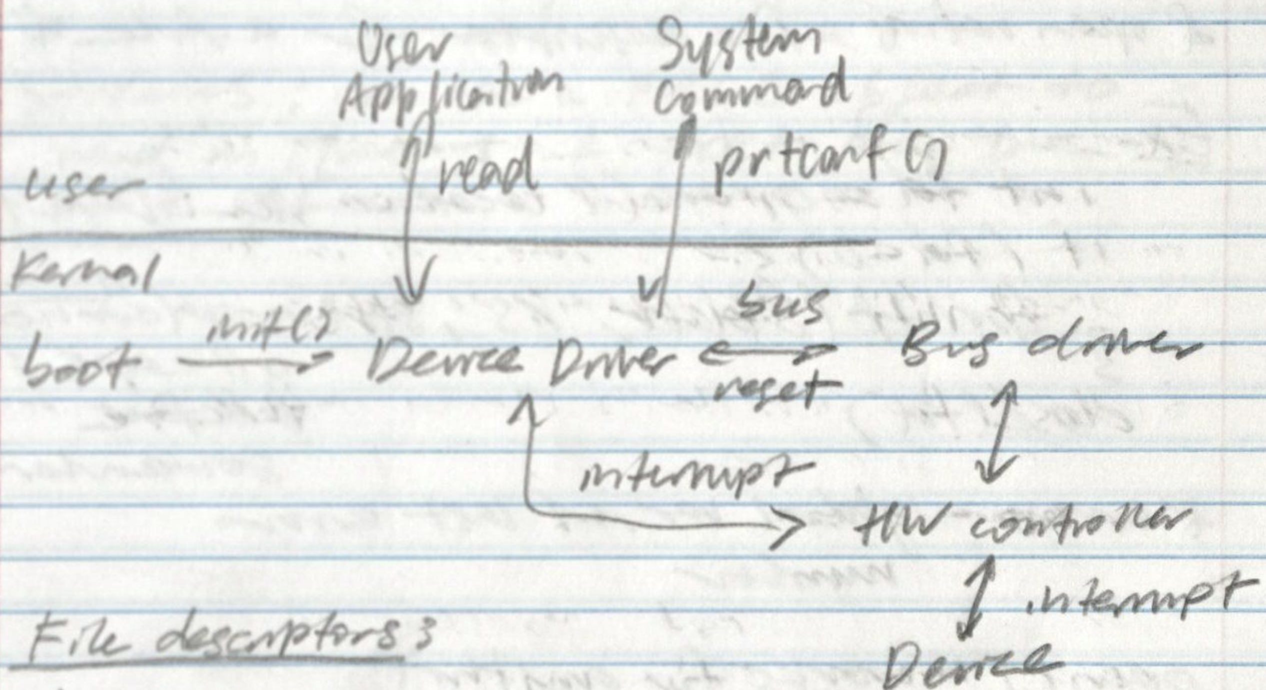
* can be blocking or non-blocking

Strace: can trace system calls

$ strace ls # trace all system calls
$ strace -p pid # trace existing process
$ strace -e open ls # trace all open sys calls
$ strace -c ls # get table of counts

& gives a lot of info on system calls

Device vs OS vs Applications:



File descriptors:

$ lsof() will list all open files

stdin -> 0
stdout -> 1
stderr -> 2
;

every file has an integer descriptor!
inside a mapping table ~~loaded~~ by the OS

I/O: Overview:

1. **Open** - create handle to stream of data
2. **Close** - destroy handle to stream of data
3. **Read** - retrieve chunk from stream of data
4. **Write** - append chunk ...
5. **Seek** - move around inside a file

* open returns a file descriptor for a file #

Ex.

```
int fd = open("location", O_RDONLY)
if (fd<0) {
    fprintf(stderr, "%s", strerror(errno));
}
close(fd)
```

all caps: #define somewhere

* errno - global var w/ last error number

open() - works for anything
fopen() - only for files

Use write to write data:

```
int fd = open("path", O_WRONLY | O_CREAT)
char* data = ....
write(fd, data, strlen(data))
```

bitwise or (bitmask)
create if doesn't exist!

Use read to read data:

```
int fd = open("path", O_RDONLY);
char buffer[BUFSIZ];
read(fd, buffer, BUFSIZ);
```

Seek: can use lseek to move in the file

```
lseek(fd, 0, SEEK_SET);
```

↳ rewind to beginning of file

Streams: can use a FILE object to read in streams of text one line at a time

Lecture Notes: 12-2-24

I/O Streams: creates a FILE object to read in files one line at a time as a stream

```
    int fd = open("file", O_RDONLY);
FILE *fs = fdopen(fd, 'r');
```

OR

```
FILE *fs = fopen("file", 'r');
```

* lowest # can get for fd is 3 *
-> since STDOUT, STDIN, and STDERR take up 0, 1, and 2
-> can be higher depending on how many libraries you load in

* don't do pointer math on FILE pointers *

Ex. rewrite cp command

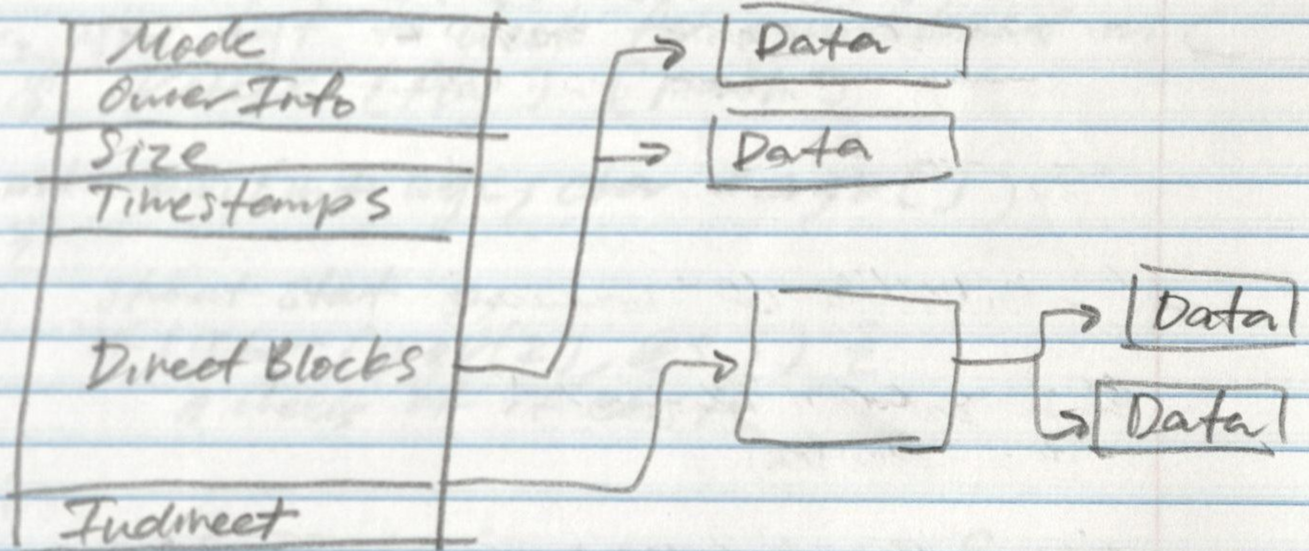
```
char buffer [BUFSIZ]
size_t nread;

while((nread = fread(buffer, 1, BUFSIZ,
    source_file)) > 0) {
  // is the # of bytes I read > 0 ?

  fwrite(buffer, 1, nread, target_file);

}
```

How big should BUFSIZ be? ~ a thousand
1024, 2048, ...

* fread / fwrite (buffer, 1, BUFSIZ, ...)
  - 1: # of blocks to read from stream
  - BUFSIZ: size of a block

* fewer system calls → faster code
  - find middle ground b/w memory and system calls

File Inodes: store administration info about files (information node)
Structure:



all in integers

- Permissions
- Owner, Group
- File size
- Links
- Modification, Access, Status times

Files: stat

can access mode info for a file w/ stat

```
struct stat s;
if (stat(path, &s) < 0 {
    fprintf(stderr, "%s", strerr(errno));
}
printf("%s size: %ld\n", path, s.st_size);
```

* all attributes of stat struct in slides *

→ can check various attributes with this

Display modes: ls -l (-i) -a

mode is the number on the left

Lecture Notes: 12/4/24

* ex Python zmq server in slides
* ex C code including Python zmq server in slides

Inodes: metadata on a file is in a different part of the disk

- blocks are pointers to where the data is actually stored
  - → direct blocks directly point to the data
  - → indirect blocks point to other blocks ---

* test equivalences chart in slides *

Ex. using test to check for equivalences in C

```
$ ./check [flag] [path]

int main(int argc, char * argv[])
{
    struct stat s;
    if(!stat(argv[2], &s)) {
        // checks if it exists
    }

    if (S_ISDIR(s.st_mode)) {
        // checks if file is directory
    }

    same for all tests in slides
```

<u>Directories</u>: just an inode (file) that references other inodes
("."): references itself
(".."): references its parent

tracks its current working directory
getcwd(buffer, BUFSIZ);

<u>Directory Walking</u>:

```
DIR *d = opendir(path);
if (!d) {
    //error
}
for (struct dirent *e = readdir(d);
        e; e = readdir(d)) {
    puts(e->d_name);
}
closedir(d);
```

// accesses files in the directory like a <u>linked list</u>

* more ex's and directory operations in the slides *

<u>Exam Q</u> - C, Python, or Shell Scripts to traverse a directory?
Python!
→ easier to code and read

Networking:

Sockets: specialized stream to send/receive network data

* cannot seek

HTTP: what makes the web work

Lecture for 12/6: 12.9.24

HTTP is text!

- Python is great w/ text
  - → use requests library
- if in shell scripts
  - → use curl command

Socket: special file descriptor that corresponds to a network endpoint

Normal File	Socket
open	socket/connect ...
read	recv
write	send
close	close/shutdown

* stream for networking

* when using TCP sockets, can convert this file descriptor into a FILE stream w/ normal read/write methods*

Client: program that dials into server and makes a request

Server: program that listens for connections and process them

Socket
Connect
Send
Receive
close

Socket
Bind
Listen
Accept
Receive
Send
close

Port: Apartment
IP: specific room address

Name Resolution: connect and bind require addresses, but users specify domain names

must map domain name to IP Address
- domain name may have multiple IP Addrs
- a single machine may have services on different ports
- each port may support different protocols

DNS: Domain Name Service
$ nslookup [domain name]
$ returns IP Addr
vice versa

Lookup Server Information:

getaddrinfo : lookup a server's address information by converting host string and port into a linked list address structure

* C code ex of this in slides *

Allocate File Descriptor:

socket : allocate file descriptor based on communication domain (ex. AF_INET, ...), socket type (ex. SOCK_STREAM), and protocol family (ex. 0).

* C ex in slides *

Establish Connections

connect : establish client connection w/ remote server

* C ex in slides *

bind : attach server socket to address and port
listen : make socket available to receive connections
accept : wait and receive an incoming connection from client

getnameinfo, fdopen, shutdown, close

* you must always close, shutdown is optional *

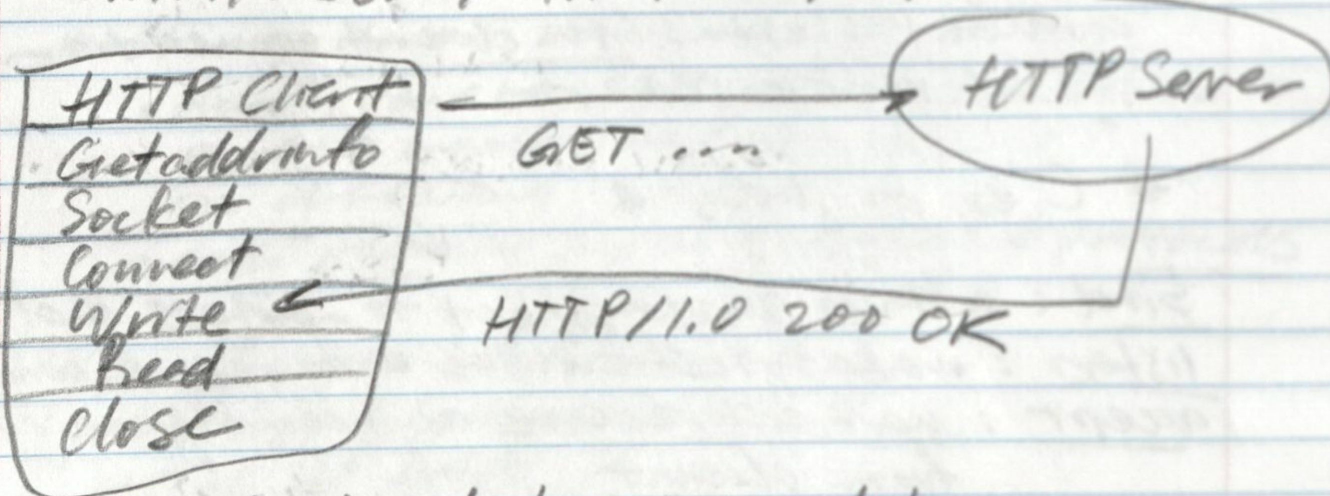
Lecture 12/9: 12.10.24

HTTP: Overview
- Hypertext trasfer protocol
- communication protocol
  1. client makes a request from a resource
  2. server returns a response to the client w/ the requested info

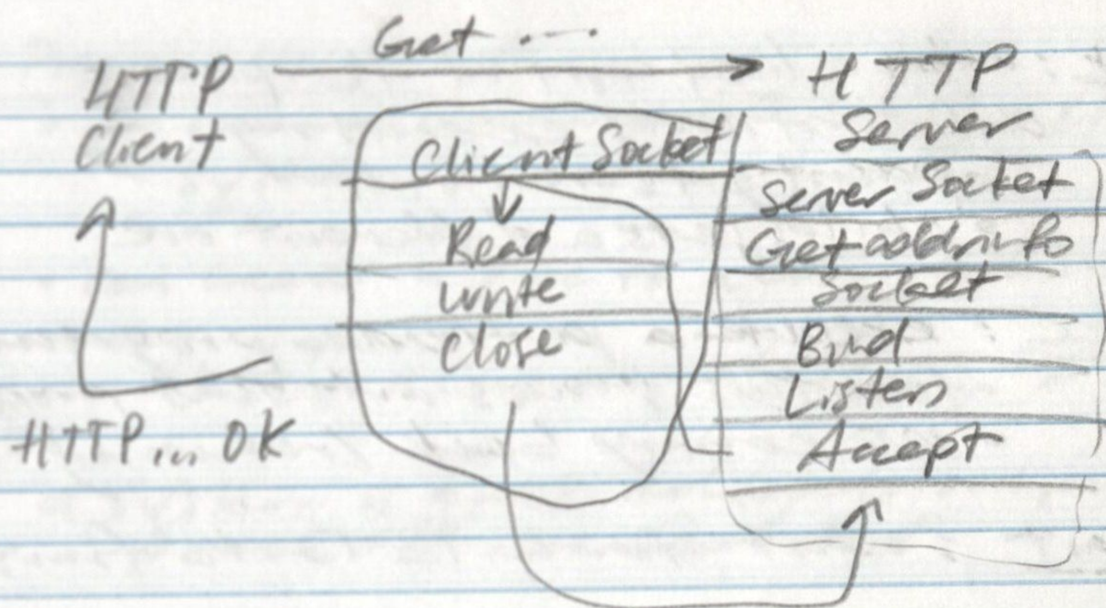
Client:
GET /index.html HTTP/1.0
Host: www.example.com
User-Agent: Mozilla/5.0

Server:
HTTP/1.0 200 OK
Content-Type: text/html;

<html><body>Hi</body></html>



* ex of client code/server in slides



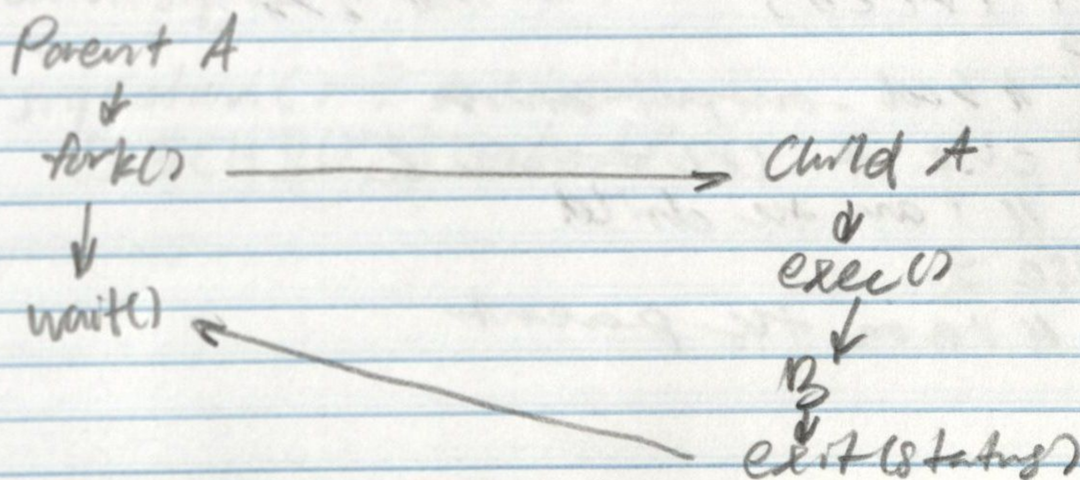
* more exs and final review in slides *

Lecture Notes: 12.4.24

Process: Life Cycle

1. Parent forks to create a new process
2. Child performs actions
   - possibly exec to run another program
3. Parent waits for child process
4. Child exits
5. Parent receives childs exit status

* flowchart of this in slides *

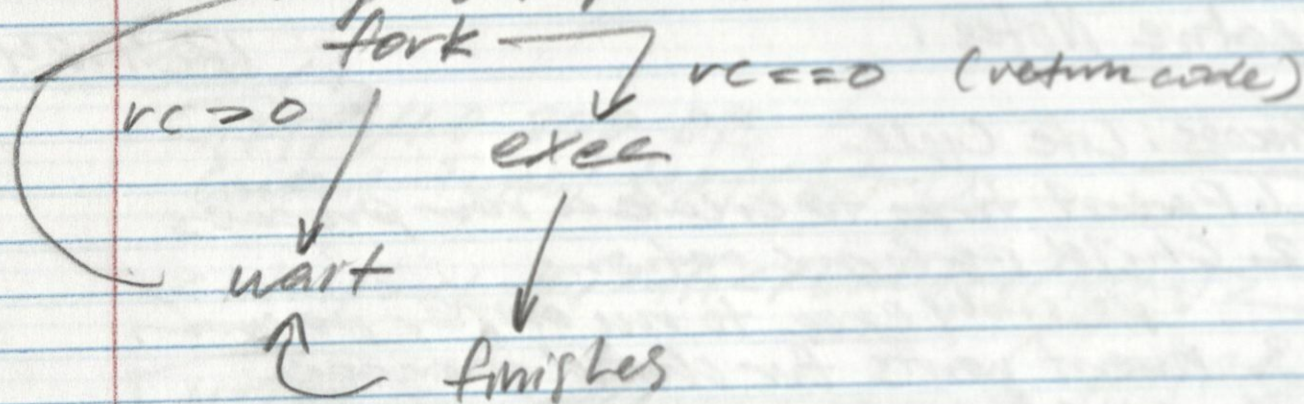


fork: makes a lazy copy of the process and continues executing
- parent gets one result
- child gets a different one

exec: execute a particular program
- turn this process into that program
- no coming back (returning)

wait: wait for a PID to finish

Shell: prompts the user
gets input from the user



* ex code of this in slides

```
int rc = fork()
if (rc<0)
{
  // bad - output error
} else if (rc==0) {
  // I am the child
else {
  // I am the parent
}
```

Process: Utilities
- can execute shell command using system
  system("ls -l");

- can create a pipe to a command sh
  using popen:
  FILE *fs = popen("ls -l", "r");

* system is quick and forgets output,
  popen has more features *

Signals: Overview
- signals notify a process about an event
  - delivers a small int

Kill: kill (pid, SIGTERM);

Signal: signal (SIGTERM, handler);

* can only send signals on the same machine)

→ like trap in shell - when handler comes
  in send the signal

* ex of signals in slides *

* sigaction()? in slides
  SIGCHILD; SIGALARM