CSS 503A Lecture #1 2019-04-01

Administria

- Anybody here for computational geometry?
  - just kidding - my sense of humor → occasionally skips out of calibration (but I will try to keep it no more than PG)
  - anybody look at assignment 1 yet? Don't panic

- CSS 503A - Systems Programming
  - AKA Operating Systems
  1) What's an OS - how are they put together
     - how do they work
  2) How to interface with OS services
     - how to work with them

- Turnkey course - follow prof Fukuda's notes
  + - my own spin (of course) - value added

- firing a lot at you concepts at you
  - repetition: it will sink in
  - bear with me

Administrivia (cont.)

classes: M-W 6 - 7:45 pm
UW1 - 041

office hours
- after class & Sunday afternoons from about 1:30 pm
- Linux lab - UW1-320
  - can arrange to meet elsewhere for privacy if requested
* office hours are generous - use them

- grading
  - 50% assignments 4 - 4th one worth 20
  - 50% exams midterm & final

- ground rules for assignments
  - README
  - BUILD (& RUN) script
  - source code
  - sample data / expected output
    - NO large data files!
  - other stuff: reports / diagrams / graphs

- compile with -Wall -Werror -pedantic
- check return codes for system calls
  - for assignments: ok to terminate on error but do detect the problem
- unless otherwise indicated:
  - input from stdin
  - output to stdout
  - meta-info (error logs messages, etc) to stderr
- practice good resource management
  - don't waste resources unnecessarily
  - ephemeral programs: program terminates releases resources

- BUILD script - fit & finish
  - can be as simple as g++ command that you type interactively saved to an executable file
  - may be more elaborate (call make, check for success)
  - not going to grade your script but will help you if not true

- each assignment has a lab session - warmup exercise
  - I will make myself available during office hours (after class & Sunday afternoons) specifically to help you on the labs

- many of you are new to Linux
  - not going to expect class time going over the basics
    → will do impromptu tutorials during office hours

- you guys have been around, so I'm not going to read you the riot act on cheating.
  a) I take it as a personal insult
  b) I have to report it to the university and I hate doing that kind of paperwork

- My office hours are generous and I want you to succeed
  - unless you cheat or are not trying

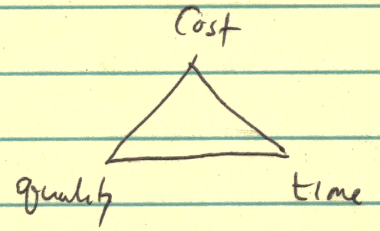
⇒ learning skills vital to your future career (whether in academia or industry)

- Don't panic!

Digression: About Engineering & Software Engineering

(I try to say something about this in every class I teach)

- engineering (in general) in one word: tradeoffs
  - time / space (algorithms)
  - cost / schedule / features / quality
  - delivery time is maintainability



* bonus: sometimes you get a solution that is "better" without sacrificing other aspect (rumpy)
  - but that's rare - mostly you're navigating around antagonistic constraints

* double jeopardy: sometimes there are constraints you cannot trade off (a woman cannot have a baby in a month)

Engineering & Software Engineering (cont.)

- Software engineering in one word: abstraction
  - key concept: Many classes
    modeling (e.g. airplane ← ATC, airline scheduling/reservation system, maintenance support system)
    information hiding
    public/private parts
    APIs
    micro services
    abstract data types
    container classes
    Magic
    decomposition
    interface/implementation
    Modules
    namespaces
    Objects/classes

Engineering & Software Engineering (cont.)

- why?
  - because, basically, we don't have the mental capacity to build & maintain complex (larger-scale (software) systems

- solution: build software systems in layers of abstractions
  - don't worry about "details" handled by "lower" layers
    - it's magic

- cost of abstractions: less-efficient use of machine resources (tradeoff)
  - not always: e.g. numpy in Python
  - usually.

- most of the time we live in a world where programmer time is more valuable than machine resources
  ⇒ consider: 10 minutes to code + 10 minutes to run
  vs 1 hr to code & 3 seconds to run
  - most of the time: grab a coffee & get your result
- often enough, it does matter: consider deployment on 10k servers vs 50 servers — 200× performance difference
  ↳ @$5k → 50 million dollars vs ¼ million dollars

Example

- don't think there are 200x performance differences?

=> Sum matrix by rows & Colums (ms)

C++	0.4	10.8	2.5x
Java1	4	154	}
Java2	3.6	138	
Python1	170.9	237,1	~ 600x
Python2	258	332	— >800x

⇒ we're going to explore a world where performance (usually) matters

- why does performance matter: (critical)
  - base layer on which everything else depends

- why study this?
  1) to understand that your lower levels of abstraction are not (really) magic
  2) to be able to use machine resources efficiently when it matters

- interestingly, performance usually matters most on the largest (datacenter) machines and on the smallest (embedded/real-time)

Operating Systems - Intro

Q: what's an operating system

A: I don't really know

- multiple definitions: different POV

history: how we got to where we are today

system architecture

- protection (one process can't scribble over another)
- resource management
  - I/O
    - async operations
  - memory management
    - each task sees logical address space
  - disk management
    - logical view: directories & files

History

- earliest days of computing (before my time):
  batch systems:
    - program + data encoded on 80-column cards
    - output to "line printer"
      - paper with sprockets
    - maybe if you're lucky: tape drive

Zavie: few years ago, cleaning up papers
found a flow chart (beautiful calligraphy!)
for a program on a 1960s-era computer
- main memory on rotating drum
- assembly code includes address of next instruction
- try to figure out location of next instruction s.t. next instruction will be under read head just as the computer is ready to read the next instruction
- worst-case: 1-rotation latency

Hashing (cont.)

- batch systems
  - you own the entire computer
  - do one thing at a time
  - "reboot" between jobs
  - each program had to include all code
    - including I/O operations

- 2 problems:
  - functions common to all programs
    - I/O operations
    - => libraries ("ordinary" code collected & reused)

  - I/O operations are much slower than CPU ops
    -> inefficient use of expensive resource
    (computers were room-fulls of equipment, outperformed by today's supercomputer-in-a-pocket standards)

History (cont.)

- Solution: run several jobs/tasks/processes concurrently
  - "time share" (not cooks)
  - Current scope: architecture that makes this possible

But:

- a computer can only do one thing at a time
- even if you could do things concurrently, two programs might need simultaneous access to common resources

fast-forward to present: these problems are basically solved (modulo ongoing improvement, more ambitious system architectures)

- eg: two programs writing concurrently to log file/console: output gets interleaved

Key OS Concerns

1) efficient use of hardware resources — device drivers
   - manage system hardware efficiently
   - provide access to hardware via well-defined interfaces
     → user-level access to hdw via OS calls (system requests)

2) process management — yield process when performing long-running I/O operation
   - create, schedule, teardown
   - interprocess communication
     → manage concurrency: fair scheduling (what is fair? - TBD)

3) file systems — manage contention for system resources

4) networking
   - complex (low-level) protocols TCP/IP

=> security / safety

convenience: API for user-land (application) programs

Working Definitions of OS

0) Traffic cop

1) resource allocator (CPU & memory, disk, ...)

2) control program (Tom: "master control program") system "traffic cop"

3) abstraction layer over hardware: "Virtual Machine"
   - not "VM" - another story "hardware abstraction layer"

* - goal of OS
   - make it easier to use computer
   - make efficient use of hardware (resources)

- Process
  - Memory management
- File systems

OS: Kernel vs Full Distro

Linux: kernel (core of OS)

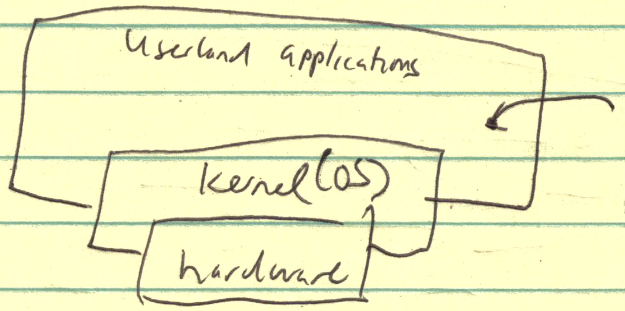
Linux distro (distribution):

- GNU/Linux
- 1000s system/user programs
- initial installation program (Anaconda)

- Shell (Bash): not operating system
  - ordinary userland application
  - can choose from several – or roll your own

- Similarly for windows
  - OS + utilities + applications
  - "Browser" is part of OS

Layered Model



System Servics (e.g print server)

applications:

- CLI (command-line interpreter) i.e. shell
- GUI
- compiler/assembler/linker
- Python/Jvm
- text editor/IDE
- spreadsheet
- word processor
- browser
- game