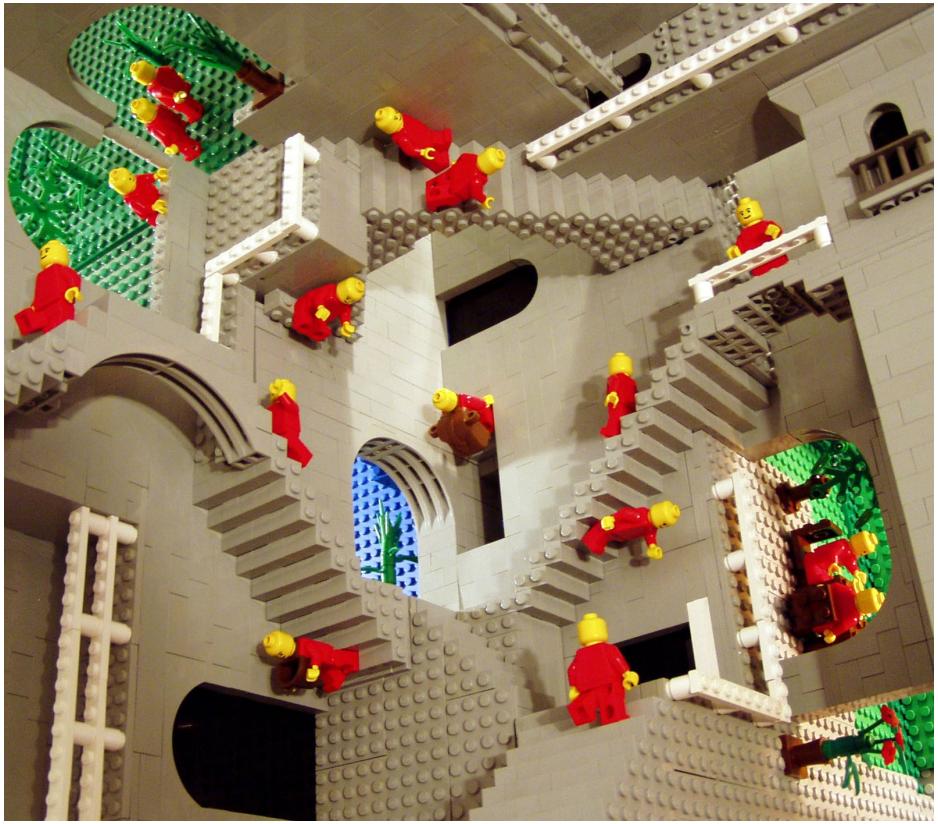
Systems Programming

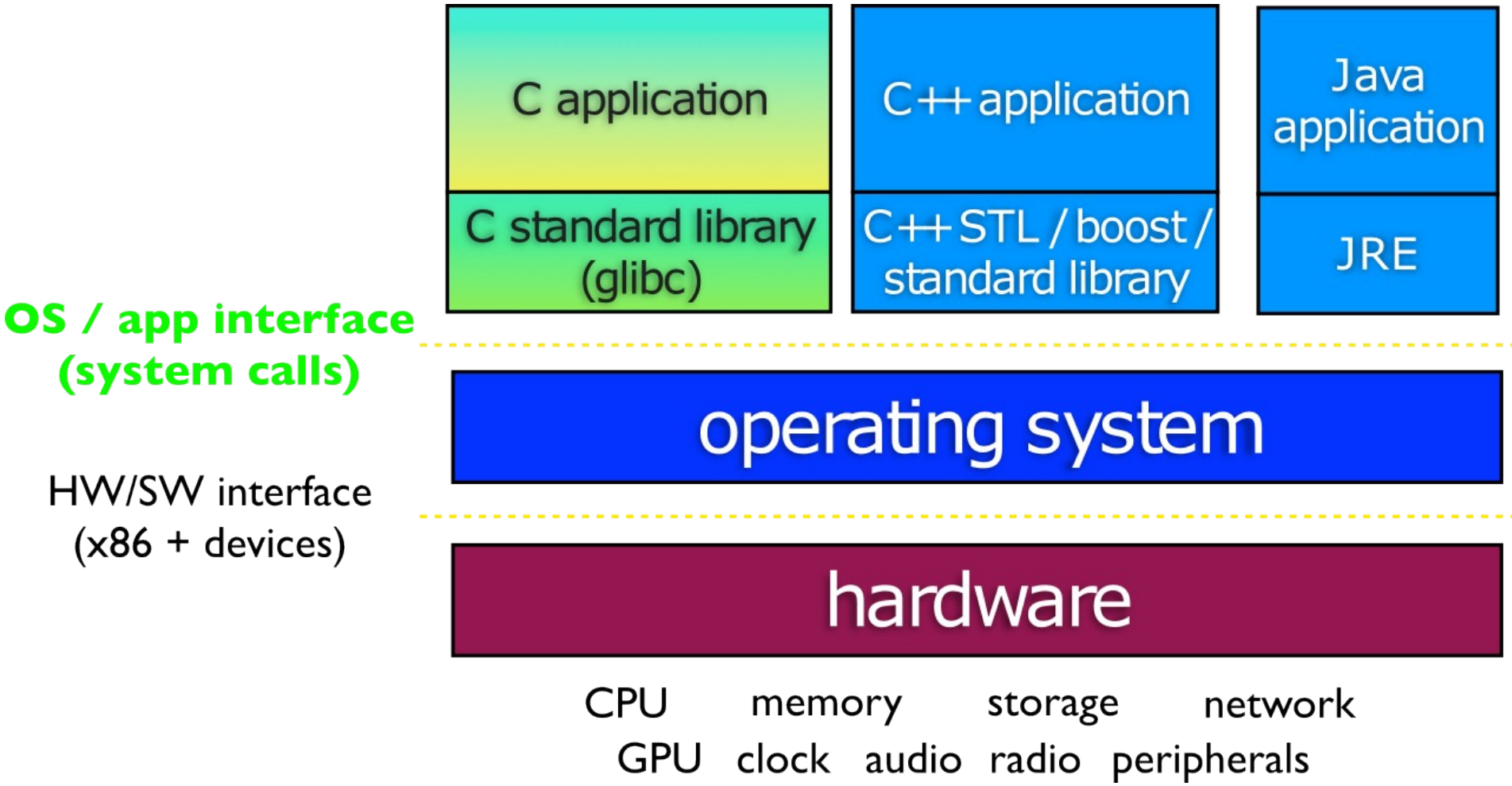
Devin J. Pohly <djpohly@cse.psu.edu>

Software system

- A platform, application, or other structure that:
  - is composed of multiple modules
    - the system's **architecture** defines the *interfaces of* and *relationships between* the modules
  - usually is complex in terms of implementation, performance, and management
  - hopefully meets some requirements
    - performance, security, fault tolerance, data consistency

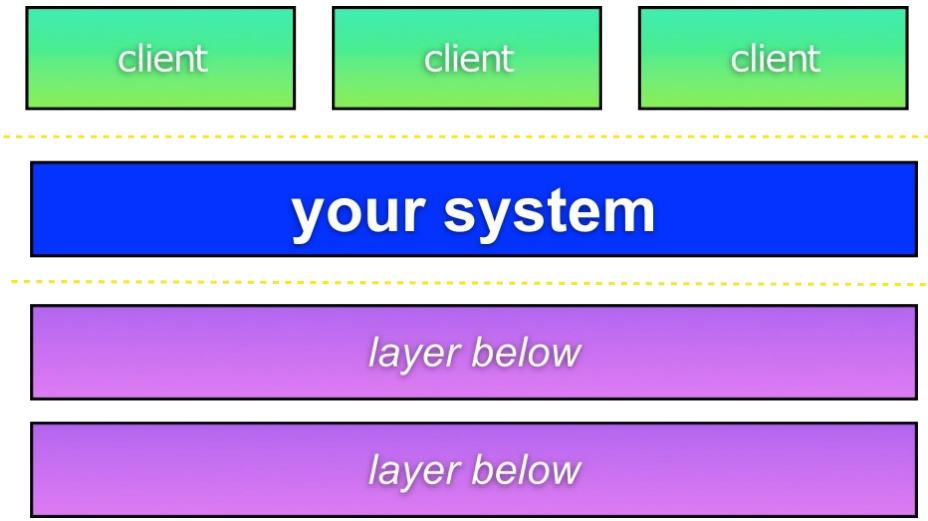


10,000-foot view



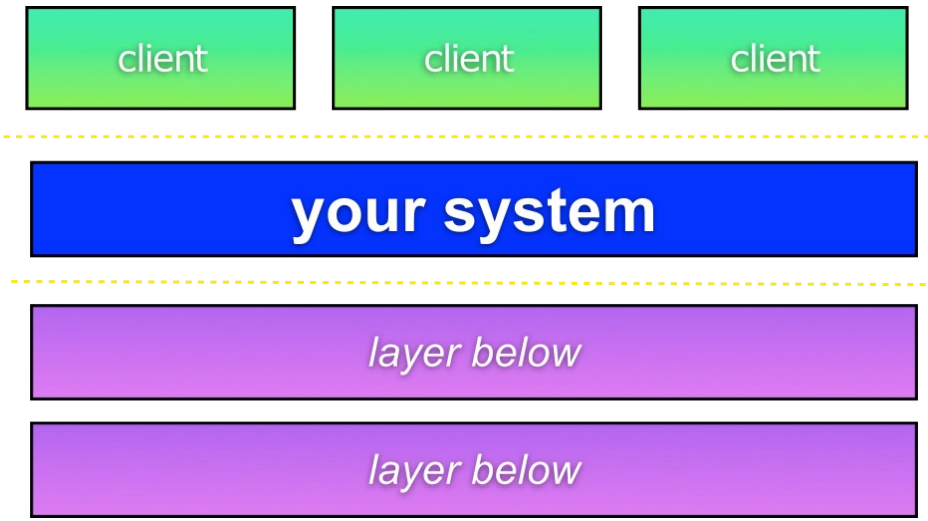
A layered view

- Each layer:
  - provides service to layers above
  - understands and relies on layers below



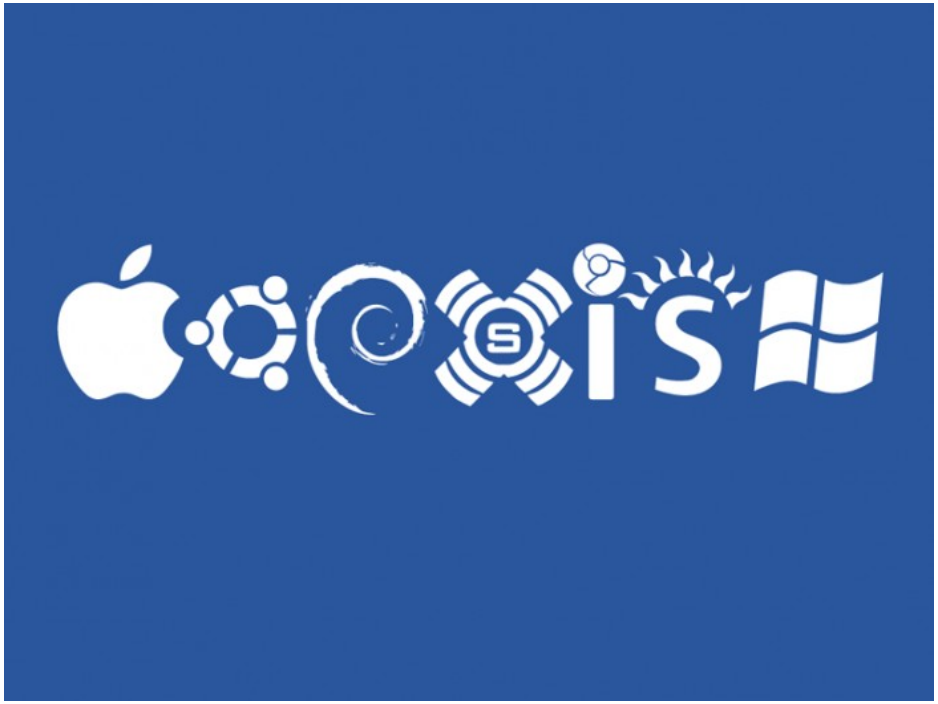
A layered view

- Higher-level layers
  - more useful, portable, reliable abstractions
- Lower-level layers
  - constrained by performance, footprint, behavior of the layers below



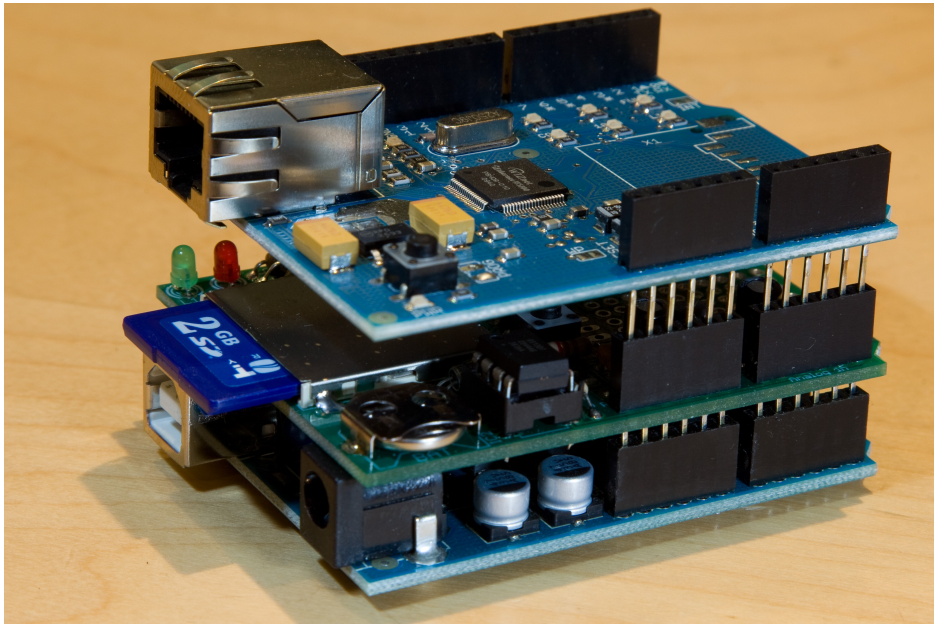
Example system

- Operating system
  - a software layer that abstracts away the messy details of hardware into a useful, portable, powerful interface
  - modules:
    - filesystem, virtual memory management, network stack, protection system, scheduler
    - each of these "subsystems" is a major system of its own!
  - design and implementation has many engineering tradeoffs
    - e.g., speed vs (portability, maintainability, simplicity)



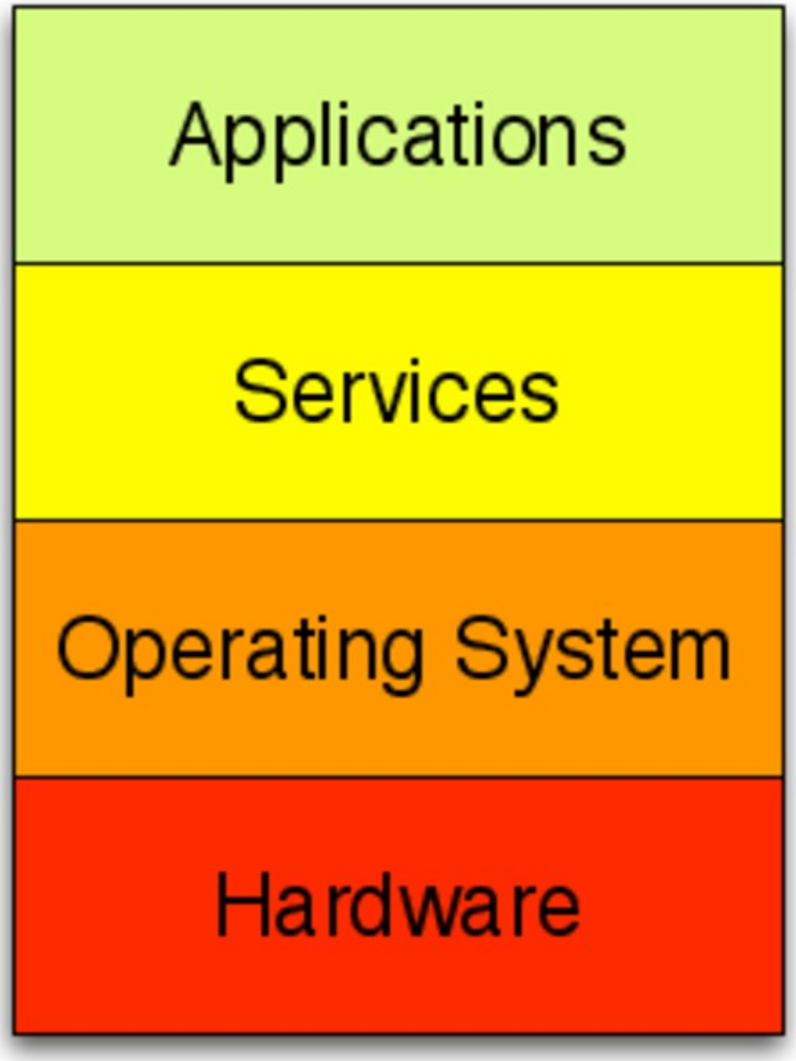
Another example system

- Web server framework
  - a software layer that abstracts away the messy details of OSes, HTTP protocols, and storage systems to simplify building powerful, scalable Web services
  - modules such as:
    - HTTP server, HTML template system, database storage, user authentication
  - also has many, many tradeoffs:
    - programmer convenience vs. performance
    - simplicity vs. extensibility
- Note: we will focus on the OS as an example system



Systems and layers

- Layers are collections of system functions that support some *abstraction* to service/app above
  - Hides the specifics of the implementation of the layer
  - Hides the specifics of the layers below
  - Abstraction may be provided by software or hardware
  - Examples from the OS: processes, virtual memory, files



A real-world abstraction

What does this do?

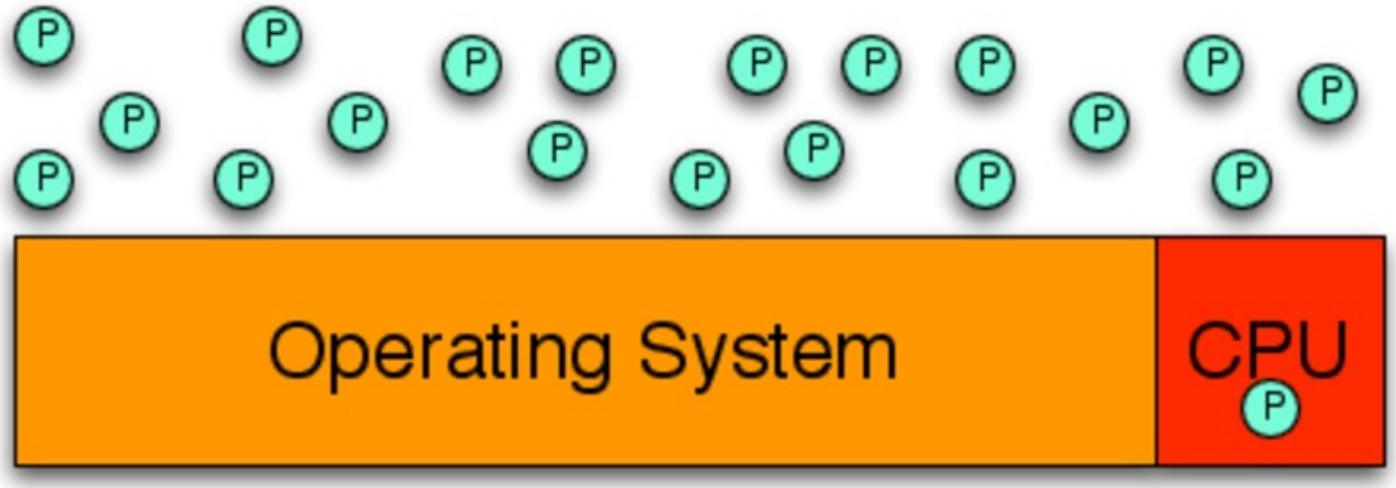
A real-world abstraction

How about this?

(Side note: “Law of Leaky Abstractions”)

Processes

- Processes are independent programs running concurrently within the operating system
  - The execution abstraction provides the illusion that each process has sole control of the entire computer (a single stack and execution context)
- PROTIP: if you want to see what processes are running on a UNIX system, use the `ps` command. Try "`ps -ax`".



Virtual memory

- The *virtual memory* abstraction provides control over an imaginary address space
  - Each process has its own virtual address space
  - The OS/hardware work together to map the address onto:
    - Physical memory addresses
    - Addresses on disk (*swap space*)

Virtual memory

- Advantages of virtual memory
  - Allows process to use entire address space
  - Avoids interference from other processes
  - Swap allows more memory use than physically available

Files

- A file is an abstraction of a read-only, write-only, or read/write data object.
- A *data file* is a collection of data on some medium
  - often on secondary storage (hard disk)
  - also called a "regular file"
- What other "objects" could fit this abstraction?

Files

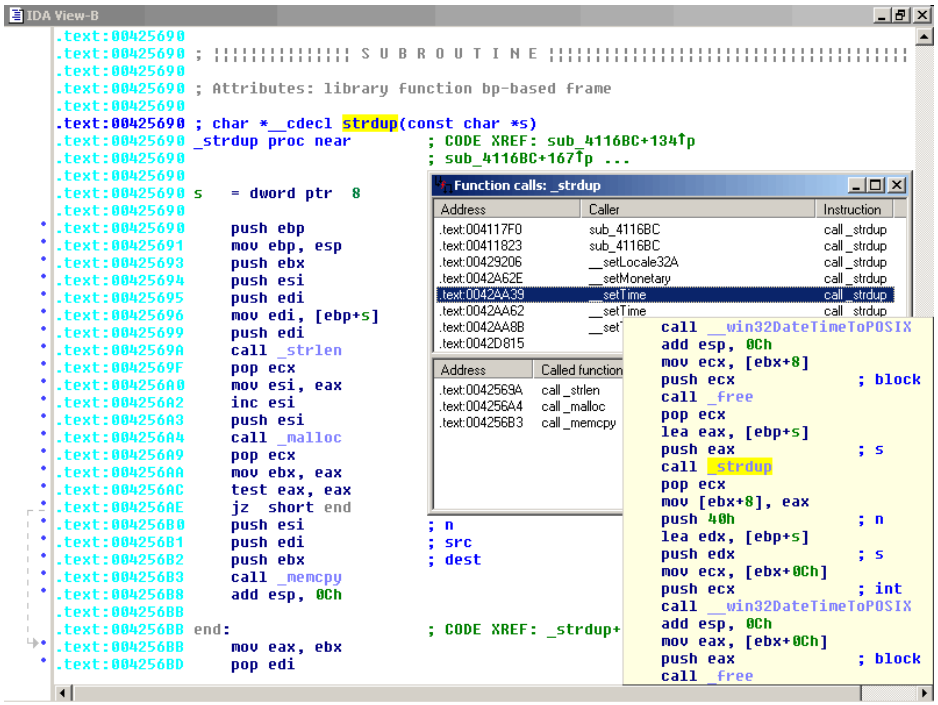
- In UNIX nearly *everything* is a file
  - Devices like printers, USB buses, disks, etc.
  - System services like sources of randomness (RNG)
  - Terminals (user I/O devices)
  - Even process information!
- PROTIP: The `/dev` directory of UNIX contains real and virtual devices. Try "`ls /dev`".

Systems programming

- The tools you need to build a system using these abstractions
  - **programming skills**: C (the abstraction for ISA)
  - **engineering discipline**: testing, debugging, performance analysis
  - **knowledge**: long list of interesting topics
    - concurrency, OS interfaces and semantics, techniques for consistent data management, algorithms, distributed systems
    - most important: deep understanding of the "layer below"

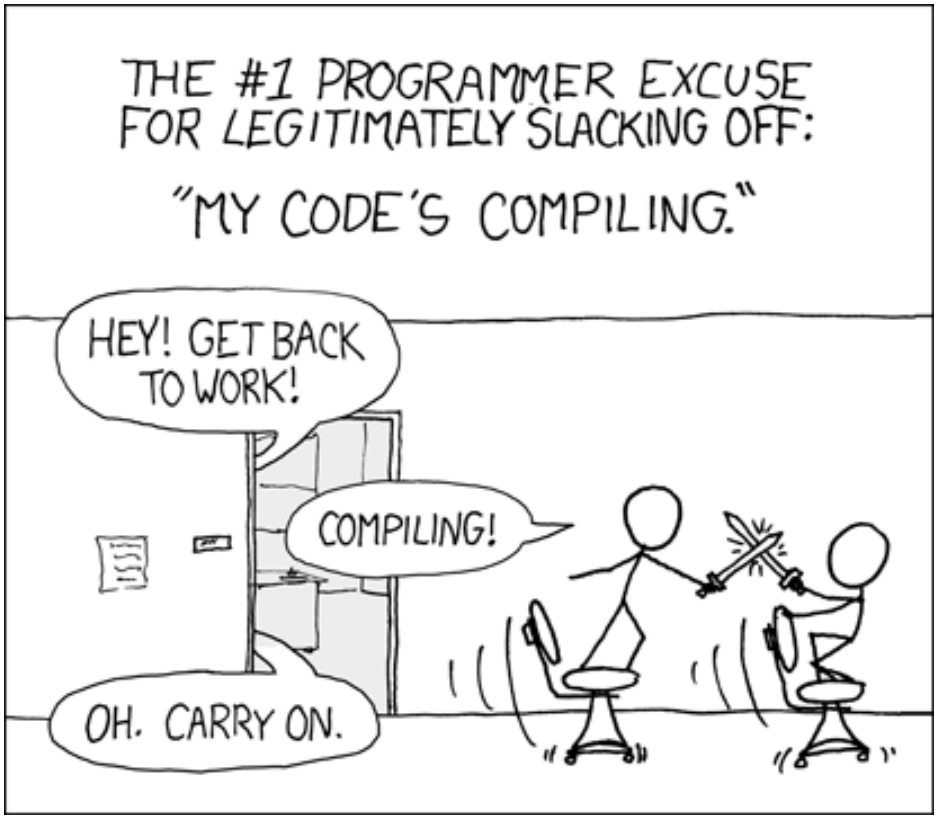
Programming languages

- *Assembly language* (ASM) and machine language
  - (*approximately*) directly executed by hardware
  - tied to a specific machine architecture, not portable
  - no notion of structure, few programmer conveniences
  - possible to write really, really fast code



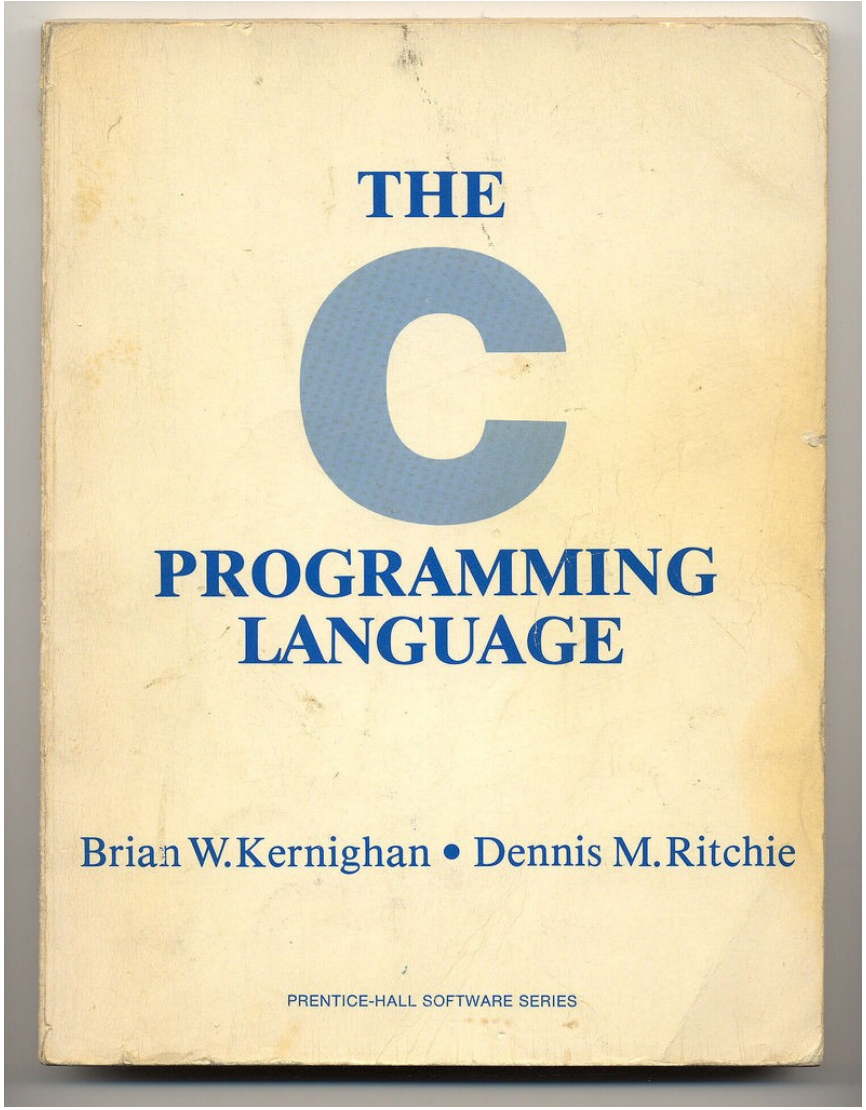
Programming languages

- *Compilation* of a programming language results in executable code to be run by hardware.
  - gcc (C compiler) produces target machine executable code (ISA)
  - javac (Java compiler) produces Java Virtual Machine executable code



Programming languages

- Structured but low-level languages (C and C++)
  - hide some architectural details
  - kind of portable
  - have a few useful abstractions like types, arrays, procedures, objects



Programming languages

- C permits (or forces?) the programmer to handle low-level details like memory management, locks, threads
- Low-level enough to be **fast** and to give the programmer **control** over resources
  - double-edged sword: low-level enough to be complex, error-prone
  - shield: engineering discipline

Programming languages

PENNSTATE 1855

- High-level languages (Python, Ruby, JavaScript, ...)
  - focus on productivity and usability over performance
  - powerful abstractions to hide the low-level gritty details (bounded arrays, garbage collection, rich libraries)
  - usually interpreted, translated, or compiled via an intermediate representation
  - slower (by 1.2x-10x), less control

Discipline

- Cultivate good habits, encourage clean code
  - coding style conventions
  - unit testing, code coverage testing, regression testing
  - documentation (code comments, design docs)
  - code reviews
- Will take you a lifetime to learn
  - but oh-so-important, especially for systems code
    - avoid "write-only" or "write-once, read-never" code

Knowledge

- Tools
  - gcc, gdb, g++, objdump, nm, gcov/lcov, valgrind, IDEs, race detectors, model checkers
- Lower-level systems
  - UNIX system call API, relational databases, map/reduce, Django
- Systems foundations
  - transactions, two-phase commit, consensus, RPC, virtualization, cache coherence, applied crypto