PennState

CMPSC 311 - Introduction to Systems Programming

Systems Programming

Professors:

Sencun Zhu and Suman Saha

(Slides are mostly by *Professor Patrick McDaniel* and *Professor Abutalib Aghayev*)

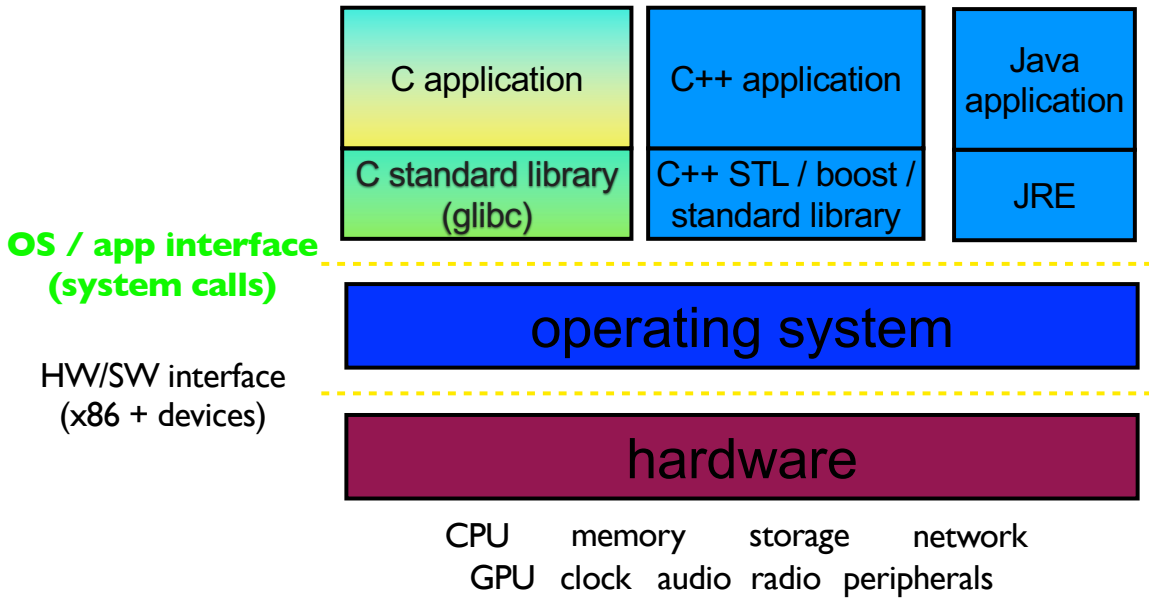
Panel: Systems Programming in 2014 and Beyond

- Bjarne Stroustrup (C++):
  - you have to deal with hardware
  - you have resource constraints
  - you need finer grained control
- Rob Pike (Go):
  - server writing
  - stuff that runs in the cloud
- Niko Matsakis (Rust):
  - you have high latency needs
- Andrei Alexandrescu (D):
  - must allow you to write your own memory allocator
  - Forge a number into a pointer since that's how hardware works

Defining properties of systems languages:

- expose the details of the underlying hardware
- give fine-grained control

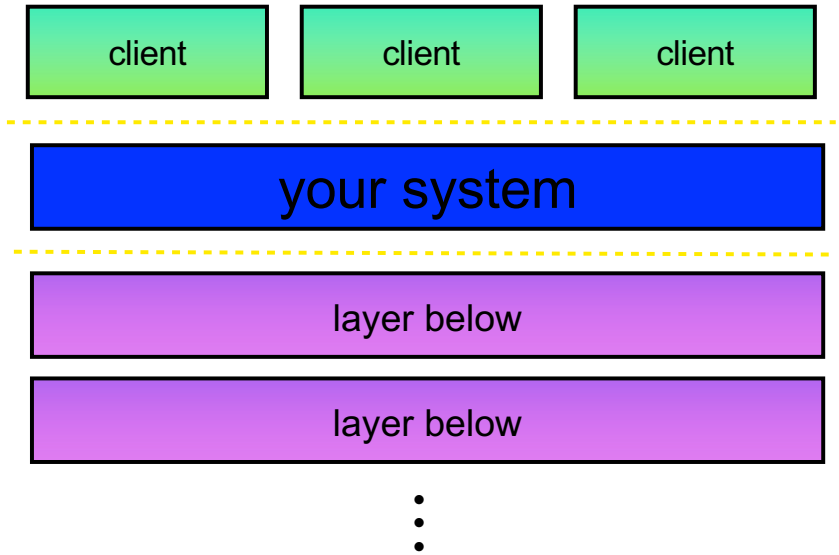
10,000-foot View of Systems



A layered view

provides service to layers above

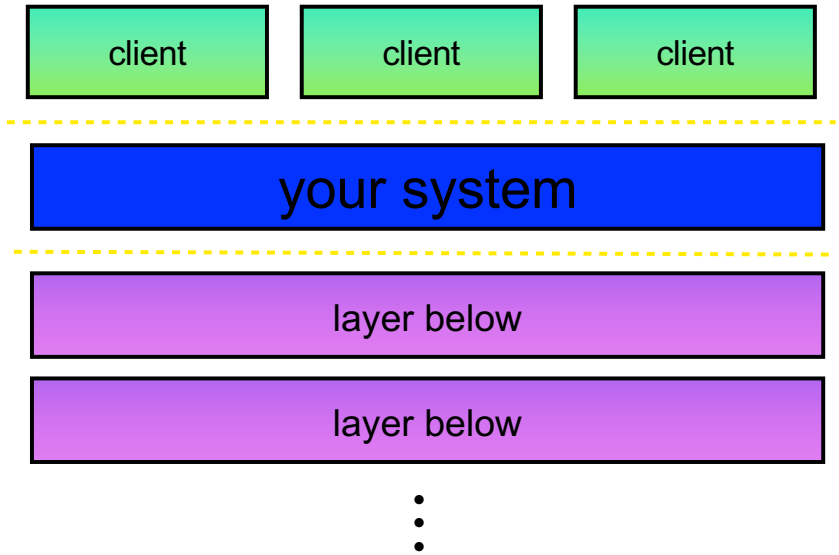
understands and relies on layers below



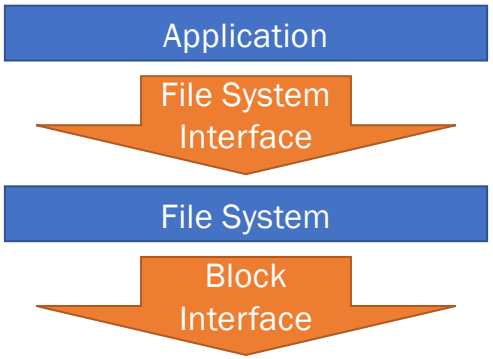
A layered view

more useful, portable, reliable abstractions

constrained by performance, footprint, behavior of the layers below



Example layered system

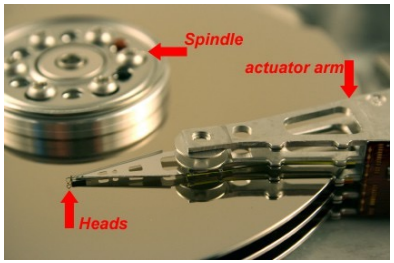


Files and directories
- create, read, write file
- create, delete directory

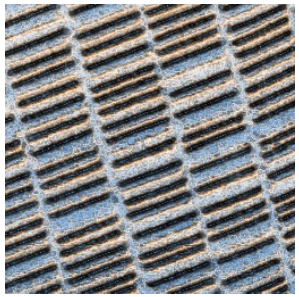
A sequence of 512-byte blocks
- read block
- write block

Software

Hardware



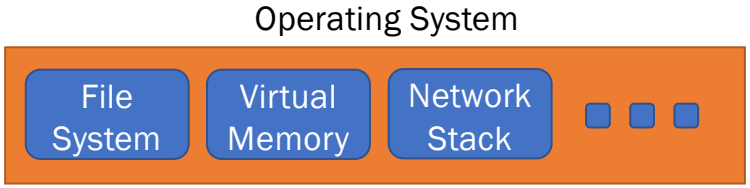
Source: dtidatarecovery.com



Source: dataclinic.it

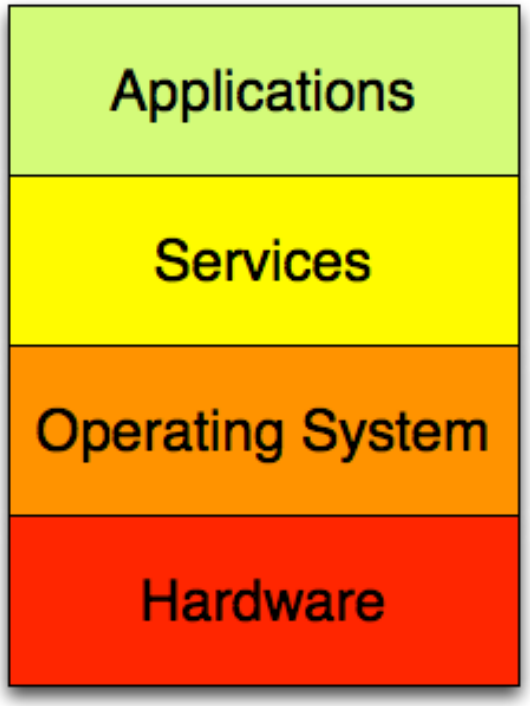
Example system: operating systems

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



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 layer
    - processes
    - files
    - virtual memory



A real world abstraction …

- What does this thing do?

What about this?

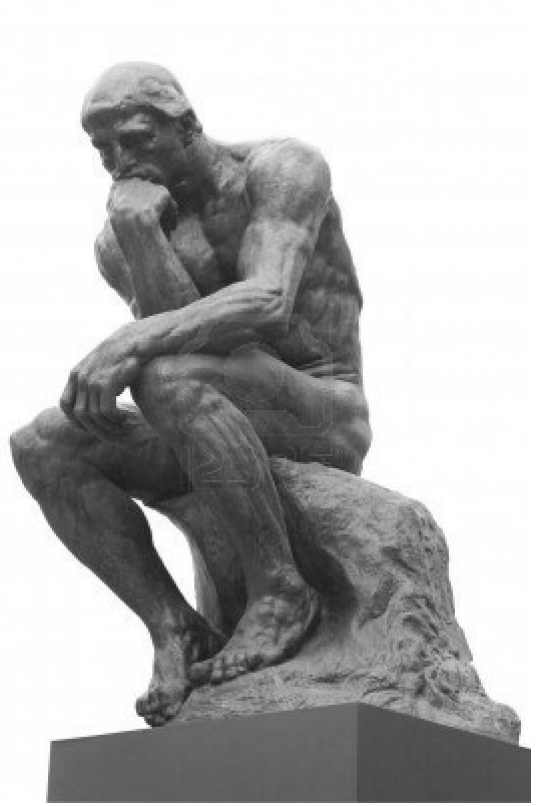
What makes a good abstraction?

- An abstraction should match "*cognitive model*" of users of the system, interface, or resources

> "Cognitive science is concerned with understanding the processes that the brain uses to accomplish complex tasks including perceiving, learning, remembering, thinking, predicting, inference, problem solving, decision making, planning, and moving around the environment."
>
> –Jerome Busemeyer

How humans think (vastly simplified)

- Our brains receive sensor data to perceive and categorize the environment (pattern matching and classification)
  - Things that are easy to assimilate (learn) are close to things we already know
  - The simpler and more generic the object, the easier (most of the time) it is to classify
- See human factors, physiology, and psychology classes ..



A good abstraction …

- Why do computers have a desktop with files, folders, trash bins, panels, switches …

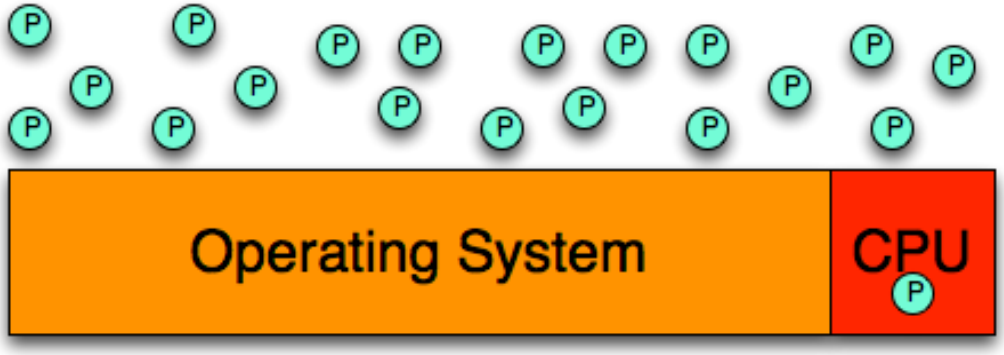
- … and why not streets with buildings, rooms, alleys, dump-trucks, levers, …

Computer system abstractions

- What are the basic abstractions that we use (and don't even think about) for modern computer systems?

Processes

- Processes are the hardware supported structures that form independent programs running concurrently within operating systems
  - The execution abstraction provides is that it has sole control of the entire computer (a single stack and execution context)



Tip: if you want to see what processes are running on your UNIX system, use the "ps" command, e.g., "ps -ax".

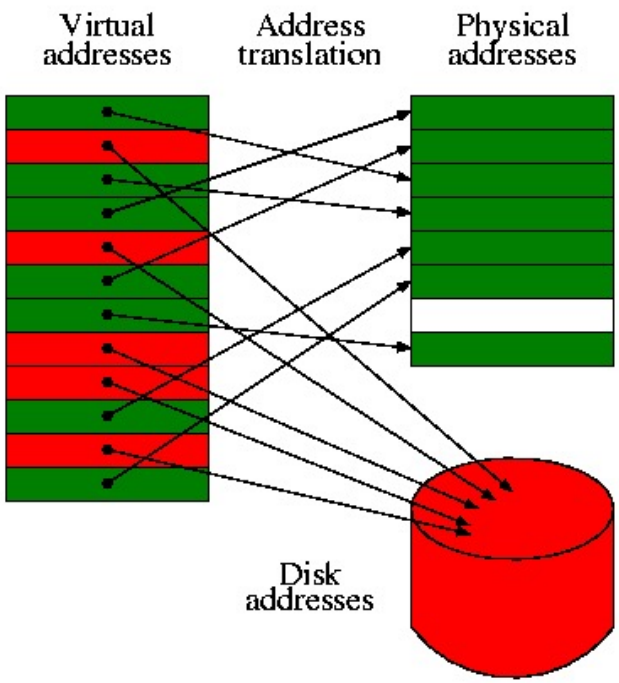
Files

- A file is an abstraction of a read only, write only, or ready/write data object.
  - A *data file* is a collection of data on some media
    - often on secondary storage (hard disk)
  - Files can be much more: in UNIX nearly everything is a file
    - Devices like printers, USB buses, disks, etc.
    - System services like sources of randomness (RNG)
    - Terminal (user input/out devices)

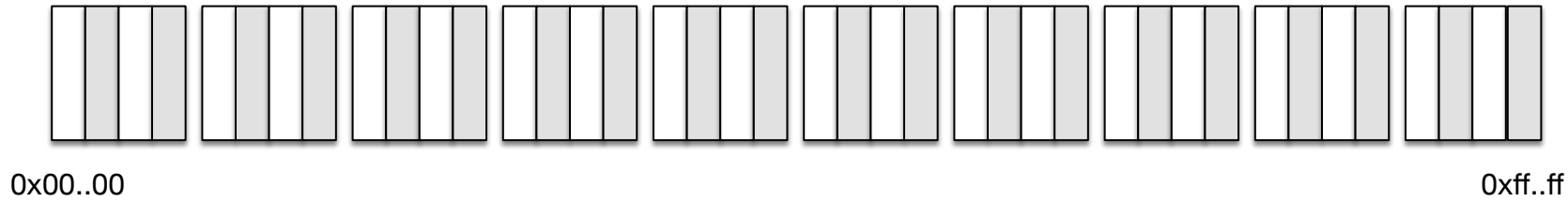
Tip: /dev directory of UNIX contains real and virtual devices, e.g., “ls /dev”.

Virtual Memory

- The *virtual memory* abstraction provides control over an imaginary address space
  - Has a virtual address space which is unique to the process
  - The OS/hardware work together to map the address on to …
    - Physical memory addresses
    - Addresses on disk (*swap space*)
  - Advantages
    - Avoids interference from other processes
    - "swap" allows more memory use than physically available



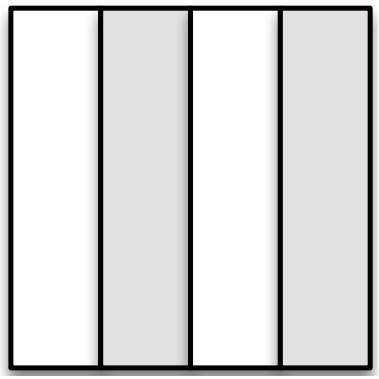
Byte-Oriented Memory Organization



- Programs Refer to Virtual Addresses
  - Conceptually very large array of bytes
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular “process”
    - Program can clobber its own data, but not that of others
- Compiler + Run-Time System Control Allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space

Machine Words

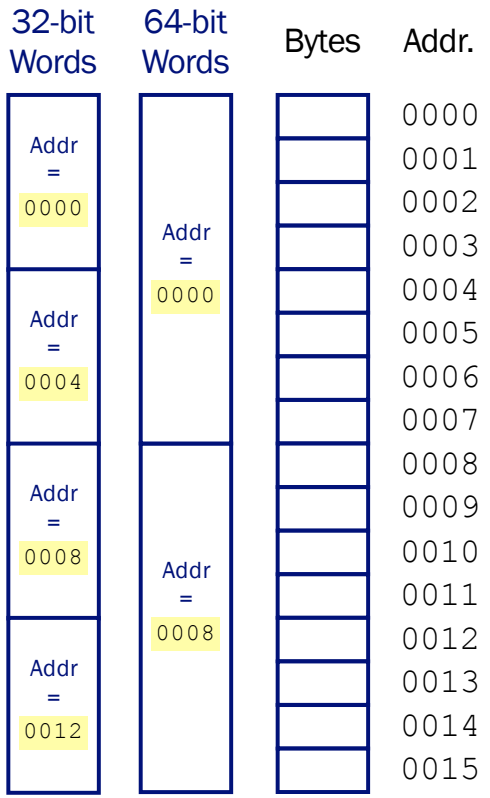
- Machine Has “Word Size”
  - Nominal size of integer-valued data Including addresses
  - Many old machines use 32 bits (4 bytes) words
    - Limits addresses to 4GB (4,294,967,296 bytes)
    - Too small for memory-intensive applications
  - Current systems use 64 bits (8 bytes) words
  - Potential address space ≈ 1.8 X 1019 bytes
  - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes



Word size

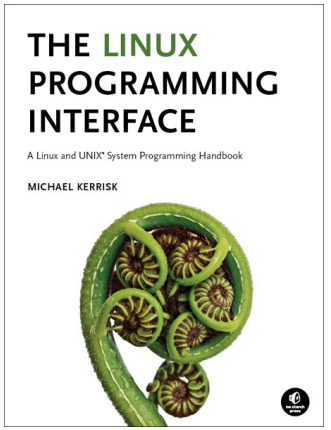
Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



APIs

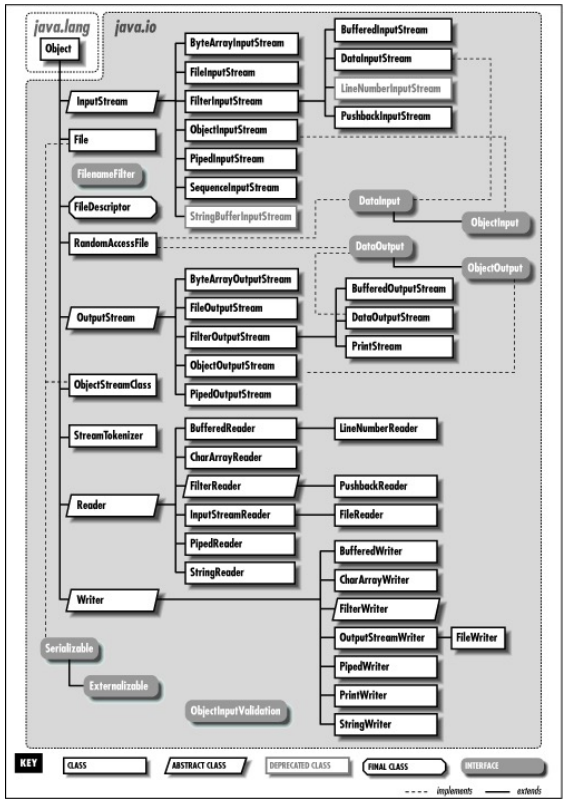
- An Applications Programmer Interface is a set of methods (functions) that is used to manipulate an abstraction
  - This is the “library” of calls to use the abstraction
  - Some are easy (e.g., `printf`)
  - Some are more complex (e.g., network sockets)
  - Mastering systems programming is the art and science of mastering the APIs and layers including:
    - How they are used?
    - What are the performance characteristics?
    - What are the resource uses?
    - What are their limitations



Example: Java Input/Output

- Set of abstractions that allow for different kinds of input and output
  - Streams …
  - Tokenizers ….
  - Readers …
  - Writers …

- Professional Java programmers know when and how to uses these to achieve their goals



Systems programming

- The programming skills, engineering discipline, and knowledge you need to build a system using these abstractions:
  - **programming**: C (the abstraction for ISA)
  - **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

- Low-level languages (C, C++)
  - hides some architectural details, is kind of portable, has a few useful abstractions, like types, arrays, procedures, objects
  - permits (forces?) 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
- High-level languages (Python, Ruby, JavaScript, …)
  - focus on productivity and usability over performance
  - powerful abstractions shield you from low-level gritty details (bounded arrays, garbage collection, rich libraries, …)
  - usually interpreted, translated, or compiled via an intermediate representation
  - slower (by 1.2 to 100+)
  - less control

Coding 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-once, read-never code