Lecture 16: Principles of System Design

Principles of Computer Systems

Winter 2021

Stanford University

Computer Science Department

Instructors: Chris Gregg and Nick Troccoli

PDF of this presentation

Lecture 16: Principles of System Design

- Let's take a step back and look at the big picture.
  - This class is about implementation-driven lectures. The code teaches the material.
  - However, you also need to walk away from this course with an understanding of the basic principles guiding the design and implementation of large systems, be they file systems, multithreaded RSS feed aggregators, HTTP proxies, or MapReduce frameworks.
  - An understanding of and appreciation for these principles will help you make better design and implementation decisions should you take our more advanced systems courses.
    - CS140: Operating Systems, which has you **design** and **implement** processes, threads, virtual memory, and a much more robust filesystem than what you were charged with implementing for Assignment 2.
      - CS110 emphasizes the client use of processes, threads, concurrency directives, and so forth. CS140 is all about implementing them.
    - CS143: Compiler Construction, which has you implement a pipeline of components that ultimately translates Java-like programs into an equivalent stream of assembly code instructions.
    - CS144: Computer Networking, where you study how computer networks (the Internet in particular) are designed and implemented.
      - CS110 emphasizes the client use of sockets and the socket API functions as a vehicle for building networked applications. CS144 is all about understanding and, in some cases, implementing the various network layers that allow for those functions to work and work well.

Lecture 16: Principles of System Design

- Principles of System Design: CS110 touches on seven such principles
  - Abstraction
  - Modularity and Layering
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - **Abstraction**
    - Separating behavior from implementation (e.g. **sort** has one interface, but many different implementations).
    - Defining a clean interface that makes a library much easier to use.
    - Examples of abstractions we've taken for granted (or will soon take for granted) this quarter in CS110:
      - **filesystems** (you've dealt with C `FILE *`s and C++ `iostream`s for a while now, and knew little of how they might work until we studied them this quarter). We did learn about file descriptors this quarter, and we leverage that abstraction to make other data sources (e.g. networked servers) look and behave like files.
      - **processes** (you know how to fork off new processes now, even though you have no idea how `fork` and `execvp` work).
      - **signals** (you know they're used by the kernel to message a process that something significant occurred, but you don't know how they're implemented).
      - **threads** (you know how to create C++ `thread`s, but you don't really know how they're implemented).
      - **HTTP** (you're just now learning the protocol used to exchange text documents, images, audio files, etc.).
  - Modularity and Layering
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - **Modularity and Layering**
    - Subdivision of a larger system into a collection of smaller subsystems, which themselves may be further subdivided into even smaller sub-subsystems.
    - Example: **filesystems**, which use a form of modularity called **layering**, which is the organization of several modules that interact in some hierarchical manner, where each layer typically only opens its interface to the module above it. Recall the layering scheme we subscribed to for Assignment 2:
      - symbolic link layer
      - absolute path name layer
      - path name layer
      - file name layer
      - inode number layer
      - file layer
      - block layer
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - **Modularity and Layering**
    - Subdivision of a larger system into a collection of smaller subsystems, which themselves may be further subdivided into even smaller sub-subsystems.
    - Example: **g++**, which chains together a series of components in a pipeline (which is another form of layering).
      - the **preprocessor**, which manages **`#include`**s, **`#define`**s, and other preprocessor directives to build a translation unit that is fed to...
      - the **lexer**, which reduces the translation unit down to a stream of tokens fed in sequence to...
      - the **parser**, which groups tokens into syntactically valid constructs then semantically verified by...
      - the **semantic analyzer**, which confirms that the syntactically valid constructs make sense and respect C++'s type system, so that x86 instructions can be emitted by...
      - the **code generato**r, which translate your C++ code into equivalent machine code.
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - **Modularity and Layering**
    - Subdivision of a larger system into a collection of smaller subsystems, which themselves may be further subdivided into even smaller sub-subsystems.
    - Example: **computer networks**, which rely on a programming model known as TCP/IP, so named because its two most important protocols (TCP for Transmission Control Protocol, IP for Internet Protocol) were the first to be included in the standard.
      - TCP/IP specifies how data should be packaged, transmitted, routed, and received.
      - The network stack implementation is distributed down through four different layers:
        - application layer (the highest layer in the stack)
        - transport layer
        - network layer
        - link layer (the lowest layer in the stack)
        - (see here for more information about the layers)
      - We learned the application-layer API calls (**`socket`**, **`bind`**, etc.) needed to build networked applications.
      - CS144 teaches all four layers in detail and how each layer interacts with the one beneath it.
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - Modularity and Layering
  - **Naming and Name Resolution**
    - Names provide a way to refer to system resources, and name resolution is a means for converting between human-readable names and machine-friendly ones.
    - We've already seen two examples:
      - Humans prefer absolute and relative pathnames to identify files, and computers work better with inode and block numbers. You spent a good amount of energy with Assignment 2 managing the discovery of inode numbers and file block contents given a file's name.
      - Humans prefer domain names like www.google.com, but computers prefer IP addresses like 74.125.239.51.
      - We spent time in lecture understanding exactly how the domain name is converted to an IP address.
    - Other examples: the **URL** (a human-readable form of a resource location), the **process ID** (a computer friendly way of referring to a process), and the **file descriptor** (a computer-friendly way of referring to a file, or something that behaves like a file—yay virtualization and abstraction!).
  - Caching
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - Modularity and Layering
  - Naming and Name Resolution
  - **Caching**
    - Simply stated, a cache is a hardware or software component that remembers recently generated results so that future requests for the same data can be handled more quickly.
    - Examples of basic address-based caches (as taught in CS107 and CS107E)
      - L1-level instruction and data caches that serve as a staging area for CPU registers.
      - L2-level caches that serve as a staging area for L1 caches.
      - A portion of main memory—the portion not backing the virtual address spaces of active processes—used as a disk cache to store pages of files.
    - Examples of caches to store results of repeated (often expensive) calculations:
      - Web browsers that cache recently fetched documents when the server says the documents can be cached.
      - Web proxies that cache static resources so other clients requesting that data can be served more quickly.
      - DNS caches, which hold a mapping of recently resolved domain names to their IP addresses.
      - memcached, which maintains a dictionary of objects frequently used to generate web content
  - Virtualization
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - Modularity and Layering
  - Naming and Name Resolution
  - Caching
  - **Virtualization**
    - Virtualization is an abstraction mechanism used to make many resources look like one. Examples include:
      - RAID, which aggregates many typically inexpensive storage devices to behave as a single hard drive.
      - the Andrew File System which grafts many independent, networked file systems into one rooted at **/afs**.
      - a web server load balancer, where hundreds, thousands, or even tens of thousands of servers are fronted by a smaller set of machines in place to intercept all requests and forward them to the least loaded server.
    - Virtualization is an abstraction mechanism used to make a one resource look like many. Examples include:
      - virtual-to-physical memory mappings, which allows each process to believe it owns all of memory.
      - threads, where a process's stack segment is subdivided into many stack frames so that multiple threads of execution can be rotated through in much the same way a scheduler rotates through multiple processes.
      - virtual machines, which are software implementations designed to execute programs as a physical machine would. VMs can do something as small as provide a runtime for Java executable, or they can do as much as run several different operating systems on an architecture that otherwise couldn't support them.
  - Concurrency
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - Modularity and Layering
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - **Concurrency**
    - We have a good amount of experience with concurrency already:
      - Multiple processes running on a single processor, seemingly at the same time.
      - Multiple threads running within a single process, seemingly at the same time.
    - When multiple processors and/or multiple cores are available, processes can truly run in parallel, and threads within a single process can run in parallel.
    - Signal and interrupt handlers are also technically concurrent. Program execution occasionally needs to be halted to receive information from an external source (the OS, the file system, the keyboard, or the Internet).
    - Some programming languages—Erlang comes to mind—are so inherently concurrent that they adopt a programming model making race conditions virtually impossible. Other languages—JavaScript comes to mind—take the stance that concurrency, or at least threading, was too complicated and error-prone to support until very, very recently.
  - Client-server request-and-response

Lecture 16: Principles of System Design

- Principles of System Design
  - Abstraction
  - Modularity and Layering
  - Naming and Name Resolution
  - Caching
  - Virtualization
  - Concurrency
  - **Client-server request-and-response**
    - Request/response is a way to organize functionality into modules that have a clear set of responsibilities.
    - We've already had some experience with the request-and-response aspect of this.
      - system calls (**`open`**, **`write`**, **`fork`**, **`sleep`**, **`bind`**, etc. are userland wrappers around a special type of function call into the kernel. User space and kernel space are two separate modules with a hard boundary separating them).
      - HTTP, IMAP, DNS
      - NFS, AFS