Bradfield
SCHOOL of COMPUTER SCIENCE

Software Systems: Behind the Abstractions

Mar-May 2023 with Oz Nova and Elliott Jin

> *To understand a program you must become both the machine and the program.*
>
> Alan Perlis, Epigrams

> *Unix and C are the ultimate computer viruses.*
>
> Richard Gabriel, The Rise of Worse is Better

As a field, computer programming advances from one abstraction to another. It's hard to imagine a world without high level languages, operating system services, network sockets and database query languages. However, no abstraction is perfect—some very far from it! Unfortunately, a software engineer who can't peer through, subvert, modify or replace these abstractions thus has a cap on their abilities.

This course is a first look under the hood of many important systems including the CPU itself, compilers and interpreters, operating systems and computer networks. Since we cover a large number of topics, we must unfortunately sacrifice some depth. As such we encourage you to see the course as providing a "map of the territory", in other words a way to start seeing how various software systems work under the hood, with various pointers as to how to understand each component in greater depth at a later date.

Class structure

For each class, we have 2-3 hours of pre-class work that we will assume everybody has done. Typically this involves an exercise, as well as suggested accompanying material. For most students, we suggest that you attempt the exercises first, and draw upon the supplementary material as needed. The benefit of this approach is

that otherwise "dry" material becomes much more interesting if it helps you make progress on a real problem! Some students prefer to obtain some background before attempting an exercise; this is of course fine too.

As much as possible we try to distinguish between primary goals and stretch goals for pre-class exercises. The stretch goals are intended for students who are particularly enjoying a topic or exercise; we won't assume that they've been completed prior to class.

We do not grade exercises in this course, but we are happy to provide some feedback on your work. Our preference is that you post your solutions to slack, for other students to comment on too. You're also welcome to send it to us directly. In either case, please be specific about the type of feedback you'd like.

Recommend resources

One challenge with covering a diverse array of topics is that if we were to use textbooks, we would need to ask you to purchase quite a few of them, only to use a chapter or two from each. Instead, we've endeavored to reference only freely available resources for required pre-class work.

That being said, there are also some very good textbooks that do cover a wide variety of these topics. We provide pointers to relevant chapters to a few of them in the "further resources" section for each class. Two textbooks we refer to repeatedly are *Computer Systems: A Programmer's Perspective* ("CS:APP") and *Computer Networking: A Top Down Approach* by Kurose and Ross ("K&R").

Using a systems language

Since this course aims to look behind various abstractions, we will often find it easier to use languages that are typically employed to *implement* these abstractions. Consider that Linux, PostgreSQL and Python are all implemented in C… in order to use these systems as case studies, it helps to be able to read C. Some newer systems may be written in other languages (such as C++, Rust or Golang) but these languages are more similar to one another than they are to the interpreted languages that you might be more familiar with.

If you've never worked with a systems language like the above, we

strongly suggest that you pick up some basic familiarity before the course begins. If in doubt, pick C, and work through at least the first chapter of *The C Programming Language*, including the exercises.

1. Introduction

Tuesday, 21 March 2023

Our first class aims to help you develop a high level understanding of the major components of a modern computer system—including those within the CPU itself—and how each is involved in the execution of a program.[1]

We will see that a useful model of program execution is that the CPU will repeatedly fetch an instruction from memory, decode it to determine which logic gates to utilize, then execute it and store the results. By the end of the class, you should be able to confidently describe this fetch-decode-execute process, including how the primary components of the system play a role in each step. In a later class, you will actually *simulate* this process by implementing a very simple virtual machine (to interpret bytecode).

Along the way, we will discuss how computers come to be architected this way, how they have evolved to perform this basic function at tremendous speed, and how the binary encodings of both instructions and data are tightly related to hardware components. These are themes that we will continue to explore throughout the course.

We will also discuss the basic organization of CPU caches, which has come to be an important consideration when writing high performance programs.

Pre-class Work

As your pre-class exercise, you will solve a complete a series of small exercises to begin exploring some of the limits and other "leakiness" of your computer's architecture.

Further Resources

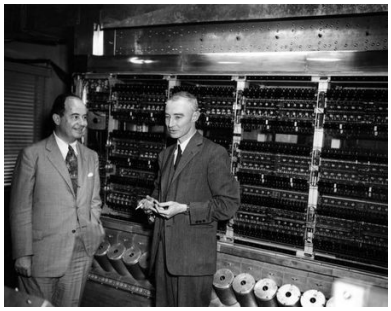
As supplementary reading, we suggest *Computer Systems: A Programmer's Perspective* chapter 1 "A Tour of Computer Systems",

[1] Broadly, the architecture of most modern computers is often but controversially called the "von Neumann architecture" after the prodigious mathematician and computer science pioneer John von Neumann. Specifically, von Neumann was a consultant on the EDVAC project and wrote a report about it that you could say went viral.

EDVAC was one of the first binary digital computers, and a successor to ENIAC which used decimal encodings but is considered by many to be the first digital computer.

Sadly, J. Presper Eckert and John Mauchly who in fact created the "von Neumann architecture" for the ENIAC are not as well known.

Below is a photo of John von Neumann standing with J. Robert Oppenheimer in front of the IAS machine, built under von Neumann's direction based on the design in the EDVAC report. IAS was built at the Institute of Advanced Studies at Princeton. Oppenheimer and von Neumann previously worked together on the Manhattan Project.



and in particular sections 1.1 to 1.4.

[2]There are many additional resources that you might find useful to better understand the execution of a computer at a very high level, as we are aiming to do with this class. One that other students have found helpful is Richard Feynman's introductory lecture (1:15 hr).

There are also several books that provide a good high-level introduction to how computers work: a popular one is *Code* by Charles Petzold, another is *But How Do It Know* by J Clark Scott.

For those looking for an introduction to computer architecture from a more traditional academic perspective, we recommend P&H chapters 1.3-1.5 and 2.4, as well as this 61C lecture from 55:51 onward.

To keep things simple, we have skipped over some important aspects of modern CPU architectures such as superscalar and out-of-order execution, pipelining, and branch prediction. For a basic introduction to a "real" CPU architecture, see this talk.

[2] Richard Feynman is of course better know for his contributions to physics, but he did spend some time thinking about computing, including the lecture series *Feynman Lectures on Computation* and some work on the Connection Machine. Like von Neumann and Oppenheimer above, Feynman too worked on the Manhattan Project—he was 24 when he joined.

2. Binary encodings of data

Friday, 24 March 2023

In this class, we'll discuss several data types and their binary representations. You may be wondering, why do we care how the computer expects a value to be encoded? Can't humans just work with the high level interpretation as a "character" or "integer" or whatever?

Unfortunately, this is yet another leaky abstraction. Consider:

- The Ariane 5 rocket explosion was largely caused by an integer overflow;
- One of the most common reasons for PostgreSQL outages is transaction ID wraparound (again, integer overflow);
- The 1991 Patriot Missile failure (which enabled a Scud to kill 28 American soldiers) was caused due to precision loss over time in a 24 bit register;
- A lot of people complain about surprising results of working with numbers in JavaScript (e.g. `typeof(NaN) === "number"`, NaN propagation, positive and negative infinity, negative zero etc) but these are all consequences of exposing IEEE

754 floating points directly to users, and make sense if you understand the goals of that design;

- The Unicode standard defines almost 150,000 characters, implying that there's demand for more than the 128 "characters" (many of which are terminal control codes!) defined by ASCII. Unfortunately there's no one correct way to encode these; and,
- Many, many other abstraction leaks.

Thankfully, it's much easier to avoid these issues once you've understood how and why we use particular binary representations of data. By the end of this class you should be able to reason from first principles about the implications of how we encode signed and unsigned integers, IEEE 754 floating point and UTF-8 characters. You will also become familiar with some building blocks like working with hexadecimal and byte ordering, and start to learn to interpret sophisticated binary formats such as network protocol formats and binary files.

Pre-class Work

In preparation for class, please complete the following short exercises. We will use the class primarily to troubleshoot any issues and clarify confusing concepts.

Further Resources

CS:APP covers binary encodings in chapter 2.

What Every Computer Scientist Should Know About Floating-Point Arithmetic is a classic and stunningly detailed paper covering the many rough edges and surprising behaviors of the standard.

Joel Spolsky on Unicode and character sets is another great resource for understanding character encoding from a programmer's perspective.

3. Programming the actual computer

Tuesday, 28 March 2023

This class introduces the x86-64 architecture,[3] and provides an

[3] This is the 64-bit version of Intel's x86 architecture, an astonishingly popular architecture first introduced with the Intel 8086, released in 1978. Modern x86 machines are largely backwards compatible with programs written for these older machines, including the Intel 386 or 486 which were the first computers that many of us used.

opportunity for us to write programs "close to the metal" in x86-64 assembly language.

Assembly language has become an abstraction layer, and a modern microprocessor may perform many microoperations to execute a single machine code instruction, but assembly language remains one of the best ways for us as software engineers to reason about the execution of our programs. In this class, we'll primarily write short programs to help us understand how our higher level programming constructs are interpreted by the machine.

By the end of the class, you should be able to read and write simple assembly routines, and to understand:

- Which high level programming statements compile to single instructions, and which to multi-step procedures;
- How a control structures like loops and conditional statements are executed, at a low level; and,
- What a calling convention is, and how a function call is made.

Pre-class Work

In preparation for class, you will write a series of short assembly programs in x86-64 assembly.

Further Resources

CS:APP covers basic x86-64 assembly programming in chapters 3.1-3.6.

One of the best options for further study is to complete some of the x86-64 exercises on Exercism. As an alternative approach, there are two *games worth playing*: SHENZHEN I/O and Human Resource Machine.[4] Both use much simpler instruction sets than x86-64, but solving the programming problems in either will help train you to think at the level of a typical machine.

[4] SHENZHEN I/O is a fantastic game by Zach Barth, where you play an American computer engineer who has moved to Shenzhen and is tasked with building small devices like e-cigarettes and garage door openers by wiring together hardware components and writing small assembly programs. Zach Barth is also responsible for the assembly programming game TIS-100, and other puzzle games including Opus Magnum and Infinifactory. You may also enjoy his talk Zachtronics: Ten Years of Terrible Games.

4. Building a virtual machine

Friday, 31 March 2023

Now that you've learned how to program the computer directly by writing x86-64 assembly and compiling it down to machine code, we

will explore an alternative approach used by many popular modern languages (such as Python, Ruby, and Java).

This class introduces the concept of a "virtual machine"[5] by asking you to implement a simplified one. We will discuss tradeoffs between executing machine code programs on a real CPU on the one hand, and executing "bytecode" programs on a virtual machine on the other.

This will also be a good opportunity for you to revisit and solidify your understanding of the "fetch-decode-execute" loop.

[5] There are two types of virtual machines: "system virtual machines", such as VirtualBox or QEMU, and "process virtual machines", such as the Java Virtual Machine or the CPython interpreter. We will be discussing process virtual machines.

See VMWare's article What is a virtual machine? or the Wikipedia page on virtual machines for more details.

Pre-class Work

In preparation for the session, please complete the exercise in the linked instructions.

Further Resources

The video Bytes in the Machine[6] gives a high-level overview of how the CPython interpreter works, and how to implement one. The chapter A Python Interpreter Written in Python from the book *The Architecture of Open Source Applications* goes into more detail.

The CPython interpreter is a "stack-based" virtual machine, but the one you implemented is register-based. The Implementation of Lua 5.0 describes another register-based machine. Robert Nystrom, the author of *Crafting Interpreters*, called this "one of my all time favorite computer science papers".

[6] At 13:35 in the video, `mod.func_code` should be `mod.__code__` instead if you're using Python3.

5. Overview of languages and compilers

Tuesday, 4 April 2023

In this class, we will investigate how a language implementation takes source code, written in a high-level language, and transforms it into low-level bytecode or machine code.

We will discuss several steps along the process, and how the representation of the program changes after each step: scanning (also known as lexing or lexical analysis), parsing, semantic analysis, optimization, and code generation.[7]

[7] Interestingly, separation into these core concerns derives from Fortran, the first successful high-level programming language. First released in 1957, Fortran was influential in many ways over subsequent languages. As just one more example: the use of "functions" for procedures derives from the objective of Fortran's designers to make the language

Scanning (also known as *lexical analysis*, or *tokenization*) is the problem of turning raw source code (a sequence of characters) into a list of meaningful tokens; for example, we might convert `100 * (2 + 5)` into the list of tokens `[INT(100), MUL, L_PAREN, INT(2) ADD, INT(5), R_PAREN]`.

Parsing (also known as *syntax analysis*) is the problem of turning a list of meaningful tokens into an abstract syntax tree, or AST. For example, we might convert `[INT(100), MUL, L_PAREN, INT(2) ADD, INT(5), R_PAREN]` into the following AST:

```
MUL(
    INT(100),
    ADD(
        INT(2),
        INT(5)))
```

Semantic analysis (also known as *static analysis*) might include traversing an AST to determine the scope of each variable, or to check for type errors (e.g. attempts to add an integer to a string).

We will explore simple examples of *optimization* and *code generation* in the pre-class exercise.

Finally, we will see some of these steps in action in the context a real language implementation such as C, Go, or Python.

Pre-class Work

In preparation for class, please begin by reading the following two sections of *Crafting Interpreters* for motivation and context:

- Why Learn This Stuff?
- A Map of the Territory

Next, please complete the challenges in the linked instructions.

Further Resources

General Resources

Crafting Interpreters by Bob Nystrom is our main recommended resource for learning about language implementations. Much of our

discussion will follow the early chapters of this book.

The Architecture of Open Source Applications has a chapter on LLVM by Chris Lattner[8]. This chapter provides more context on the compilers toolchain and project organization used by a number of modern language implementations.

[8] Chris Lattner is co-creator of the hugely influential LLVM project, a set of tools for compiler projects designed to be a more modular, reusable alternative to equivalent GNU tools (for instance Clang, an alternative to the GCC compiler). While working on LLVM at Apple, Lattner also designed the Swift programming language.

Compilers: Principles, Techniques, and Tools (aka "The Dragon book") provides a great high level introduction in chapters 1 and 2. Professor Aiken does so in the lecture segments Introduction, Structure of a Compiler and The Economy of Programming Languages.

For more or on the history of programming languages, we recommend the book *History of Programming Languages*.

Parsers in Production Languages

Parser generators vs. handwritten parsers: surveying major language implementations in 2021 shows that many languages use handwritten parsers.

Guide to the Parser describes the CPython parser, which uses a parser generator.

Go's `parser` package provides an example of a handwritten parser. The `ParseFile` function is a good entry point to start exploring.

6. Runtimes and memory management

Friday, 7 April 2023

Programmers working in higher-level languages (such as Python, Ruby, or JavaScript) are protected from low-level details of memory management. That is, until a memory leak takes down the server, or an issue with garbage collection degrades performance to an unacceptable level.

Memory management is an important part of program execution; understanding the details of memory allocation and garbage collection will help you diagnose difficult issues and write more performant programs.

This session will be a broad overview of memory management involving topics such as:

- The stack vs. the heap
- Manual memory allocation via `malloc` and `free`
- Considerations involved in designing a good storage allocator
- Automatic memory management via garbage collection[9]

[9] Broadly, there are two major approaches to automatic memory management: reference counting and tracing garbage collection.

Reference counting involves constantly incrementing and decrementing a count on the references to each object. As soon as the count reaches zero, the object is deallocated and its outbounded references are recursively decremented too. This is a very simple but relatively inefficient strategy used by very few modern language implementations, although CPython uses a version of it.

Tracing garbage collection is common enough that it is typically referred to as just "garbage collection". Instead of maintaining reference counts, a tracing garbage collector periodically performs a graph search from a set of root objects, and deallocates any objects that can't be reached. This can be a disruptive process in its naive form, but Go uses a number of strategies for mitigating this cost.

Pre-class Work

In preparation for the session, please complete the readings and exercises in the linked instructions.

Further Resources

`malloc` Design and Implementation

Sections 5.4 and 8.7 of *The C Programming Language* describe a simple implementation of `malloc` and `free`, along with some suggested exercises.

Dan Luu wrote a malloc tutorial which walks through an implementation, provides some debugging tips, and suggests some additional exercises / improvements.

The GNU Allocator gives some context for the `malloc` implementation used in the GNU C Library.

TCMalloc is Google's custom implementation of `malloc`.

- Go's memory allocator was also originally based on TCMalloc
- Cloudflare wrote a case study on the effect of switching from the standard library `malloc` to Google's TCMalloc

Garbage Collection

Ardan Labs has an extensive reading list on garbage collection.

The following resources discuss garbage collection in the context of specific languages:

- Go:
  - A Guide to the Go Garbage Collector
  - Garbage Collection in Go
- JavaScript:

  - Many of the posts from the V8 blog with the "memory" tag
- Python:
  - Garbage Collector Design
  - The Memory Management chapter of CPython Internals
- Ruby:
  - A Day in the Life of a Ruby Object
  - Ruby Garbage Collection in Under Two Hours
  - Chapter 5 of Ruby Hacking Guide

7. Introduction to operating systems

Tuesday, 11 April 2023

In our first class, we'll discuss the key features operating systems provide for their users,[10] and discuss their purposes and the concepts they introduce. This will provide a big picture view of typical operating systems, to help us dive deeper in subsequent classes.

[10] In this course we'll be studying the operating system in its role as a *software platform* for developers. That is to say—for the purposes of this course—when we say "OS user" we mean the *developer* writing code against the Operating System's API, not the "end user" typing queries into the spotlight search bar.

Pre-class Work

In preparation for this class, please read *OSTEP* chapter 2: Introduction to Operating Systems. Test your understanding by summarizing the primary responsibilities of an operating system.

Please also install VirtualBox and an Ubuntu image, and follow the instructions below to start exploring the operating system.

In-class Exercise

Our in-class exercise will involve a quick tour of the Linux built-in documentation and command line interface, followed by an interactive exploration of some of the user-facing aspects of each of the important conceptual concerns of an operating system, such as process scheduling and memory management.

Further Resources

CS:APP has a very short introductory section on operating systems, in chapter 1.7. Relevant chapters from *Linux Kernel Development* include 1: "Introduction to the Linux Kernel" and 5: "System Calls".

Relevant CS162 lectures include 1: Introduction, overview and 2: Abstractions: Services, Structures, and Threads.

For some interesting historical context, see The UNIX Timesharing System, the original 1974 Unix paper from Ritchie and Thompson and The Evolution of the Unix Time-Sharing System, a 1979 retrospective paper from Ritchie.

8. Virtual memory and file systems

Friday, 14 April 2023

We'll have already discussed many aspects of virtual memory in the previous class, but in this class we'll do a deeper dive into the various hardware and software concepts involved. We'll also examine the interface presented by the OS to manage and manipulate the memory subsystem.

In the file systems portion of this class, we'll focus on how abstractions like "files" and "directories" and "links" are actually implemented. As a developer, a strong understanding of the capabilities and limits of your operating system's file subsystem can save you potentially thousands of lines of unnecessary code.

Pre-class Work

Please prepare for the virtual memory portion of class by reading *OSTEP* chapters 13 Address Spaces and 18 Paging. Test your understanding with the following questions and exercises:

- What is the page size for your operating system's virtual memory system?
- Approximate the size (in bytes) of the page table for your operating system right now, based on the size of programs that are running.
- Write a program which consumes more and more physical memory at a predictable rate. Use `top` or a similar program to observe its execution. What happens as your memory utilization approaches total available memory? What happens when it reaches it?
- Suppose the designers of your operating systems propose quadrupling the page size. What would be the trade-offs?

In preparation for the file systems component, please read *OSTEP* chapters 39 Files and Directories and 40 File System Implementation. If time permits, test your understanding by writing a small `ls` clone.

Further Resources

OSTEP covers the virtual memory subsystem in more detail in chapters 13 to 24. *CS:APP* does so in chapter 9: "Virtual Memory".

Relevant CS162 lectures include 9: Protection: Address Spaces, Address Translation and 10: Address Translation, Caching and TLBs.

Linux Kernel Development covers Linux's virtual memory implementation primarily in chapter 15: "The Process Address Space", but also to some degree in 12: "Memory Management".

For a summary of the virtual memory implementations of a number of commercial systems, see Virtual Memory: Issues of Implementation.

Beyond the *OSTEP* readings provided above, the entire "Persistence" section is relevant to this class. In particular, we would recommend chapters 42 and 43, on journaling and log-structured file systems. In *CS:APP*, the file system abstraction is covered in chapters 10.1-10.4.

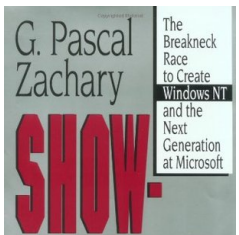
The most relevant CS162 lecture is 13: Filesystems: Files, Directories, Naming, and Caching.

Linux Kernel Development covers "The Virtual Filesystem" in chapter 13.

For some useful resources on particular file system implementations see the video ZFS: The Last Word in File Systems, Design of the FAT file system on Wikipedia, and the Apple File System Guide.

For an excellent book covering the history of developing Windows NT (including NTFS) see *Show Stopper!*.[11]

[11] For those who aren't immediately drawn to a book about the history of a version of Windows, note that Windows NT was one of the most ambitious and successful software projects ever undertaken, with a protagonist (David Cutler) who makes the book a thrilling page-turner.



9. Virtual machines and containers

Tuesday, 25 April 2023

Our final operating systems class covers two technologies with distinct purposes which tend to be conflated. Virtual machine monitors (or "hypervisors") are a decades old technology for running multiple operating systems on the same machine,[12] by providing a virtual machine interface to each of them. Containers are a higher-level, lighter weight construct for isolating groups of processes running within the same operating system.

We will discuss how both VMMs and container runtimes are typically implemented, and discuss their corresponding use cases and trade-offs.

[12] VMMs have both an ancient and a modern history. They were first invented for IBM customers to be able to run multiple concurrent operating systems on the same expensive IBM mainframe they'd just purchased. They were later re-invented by researchers at the Computer Systems Laboratory at Stanford, including Mendel Rosenblum and Diane Greene, who with colleagues spun out to start VMware in 2003. At the time of writing, Diane Greene runs Google's cloud businesses, and Professor Rosenblum continues his research at Stanford, now as an ACM fellow. They are pictured below.

Pre-class Work

Please prepare by reading the *OSTEP* Appendix on VMs and watching Brian Cantrill's VelocityConf talk on containers. Test your understanding by asking yourself: what do you expect the overhead to be of using containers? What about virtual machines?

When you're ready, follow the instructions below to implement a very basic container runtime.

Further Resources

OS:APP has a few sections related to virtual machines: there's a virtual machines "case study" in section 2.10 and section 10.2 covers the additional memory virtualization challenges introduced by virtual machine monitors.

10. Tour of relational databases

Friday, 28 April 2023

We start by considering what database management system are, what problem(s) they're intended to solve, and how they are typically structured.

Our first exercise will be a data exploration activity *without* using a DBMS, as a way to motivate our subsequent discussion. We'll then

track the lifecycle of a SQL query through PostgreSQL, to improve our understanding of the overall architecture. In particular, we will focus on the role and implementation of the query executor.

Pre-class Work

Please spend 1-2 hours skimming the excellent but long paper Architecture of a Database System. The objective here is to obtain an overall sense of the key responsibilities and components of a typical relational database management system; we don't expect you to pick up every detail.

Further Resources

During class, we'll be exploring the PostgreSQL codebase and stepping through some code. If you'd like to continue exploring, you can check out the PostgreSQL repo, build it from source[13], and import some test data[14]. Here are a few steps you might find interesting to try:

- Use Wireshark to watch network traffic between the `psql` client[15]) and the Postgres server.
- Step through query execution with a debugger[16]. `ExecutorRun` and specific nodes such as `ExecLimit` are good places to set breakpoints if you execute a simple query such as `SELECT * FROM movies LIMIT 5`.
- Inspect the on-disk representation of a table[17], and see how it changes as you execute `INSERT`, `UPDATE`, or `DELETE` queries[18].

For more on PostgreSQL architecture, you can consult:

- This backend flowchart
- This overview of Postgres internals
- PostgreSQL Internals Through Pictures, which walks through the lifecycle of a Postgres query

For more context and perspective on relational database management systems, you can consult:

- The introductory session of CS186

[13] Make sure you run `configure` with the `--enable-debug` flag so that you can step through with a debugger. You may also want to update the `configure` script to remove all occurrences of `-O2`, since optimizations can complicate debugging.

[14] The MovieLens 20M Dataset is a good choice for test data. You can import CSV files directly into Postgres.

[15] Since `psql` uses Unix Domain sockets by default, you will need to specify the `-h` and `-p` flags to use Wireshark.

[16] In `gdb` and `lldb`, you can use the `attach <pid>` command to debug a running process.

[17] You can use `SHOW data_directory` to find the base path and `SELECT pg_relation_filepath('...')` to show the relative path for a particular table.

[18] Make sure you run `CHECKPOINT` so that changes get flushed to the underlying files (and not just to the write-ahead log).

- Michael Stonebraker's retrospective on his paper "One Size Fits All - An Idea Whose Time Has Come and Gone", which provides a higher level view of the overall problem of information processing, and the various approaches that we may take for different workloads

Michael Stonebraker will be making *many* apprearances in this course. He was a pioneer of relational databases with Ingres and POSTGRES (POST inGRES, later PostgreSQL), as well as later paradigms such as stream processing, column stores and in-memory databases. He was the recipient of the 2014 Turing Award.

11. Physical storage and indexes

Tuesday, 2 May 2023

> *Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious.*
>
> Fred Brooks, The Mythical Man Month

How do databases actually store data on disk and access them efficiently? In this class we'll discuss two key Postgres file formats: heap files and B+ tree indexes. Understanding how the data is actually manifested, as bytes on physical storage, will also help us understand the steps involved in efficient access, as well as the trade-offs involved in different access methods.

Pre-class Work

Please watch the CS186 lectures on:

- File Organizations and Indexing (start from 31:30)
- Tree-Structured Indexes

Then, test your understanding by sketching out (1) the process of looking up a particular row with a full table scan, and (2) the process of looking up a row using a B+ tree index.

Further Resources

R&G covers tree-structured indexing in chapter 10, and hash-based indexing (which we won't discuss) in chapter 11.

For more of a practitioner's perspective, see the website Use the index, Luke.

This B+ Tree Visualization is worthwhile for those struggling to visualize how a B+ tree is organized.

The Evolution of Effective B-tree: Page Organization and Techniques is an interesting account of some practical improvements made to layout to data *within* a B-tree node.

12. Joins and query optimization

Friday, 5 May 2023

One of the fundamental challenges of information systems is to efficiently answer questions regarding two or more separate collections of data at the same time. In the context of database tables, this operation is called a join; in this class, we'll explore the three common algorithms for performing joins in a relational database. We will also discuss the query optimizer from a user's perspective.

Pre-class Work

Please start by watching the CS186 lecture on Join Algorithms.

Then, test your understanding by sketching out pseudocode implementations of at least two joins algorithms.

Finally, please begin the exploration exercise in the linked instructions. We will work through the exercise in class together so you don't need to complete the entire exercise before class, but please make sure you've at least read through the exercise and ran the psql script locally.

Further Resources

Join Processing in Database Systems with Large Main Memories is a 1986 paper advocating the use of hash joins over sort-merge joins at time when main memory sizes were becoming large enough to make hash joins feasible. It also presents a novel hybrid hash join algorithm.

PostgreSQL implements nested loops join, hash join and sort merge join, and as usual these implementations are highly readable.

R&G covers query optimization in chapter 15.

These topics present a great opportunity to dive into some old papers and be impressed by their continued relevance. Access Path Selection in a Relational Database Management System is Patricia Selinger's paper outlining the dynamic programming based query planner in System R that's still in wide use today (note however that unlike the paper, Postgres DOES consider "bushy" query plans), and System R: Relational Approach to Database Management is an older paper first describing System R's adoption of the relational model generally.

For some perspective around the relational model and its alternatives, it's also worth reading What Goes Around Comes Around, particularly sections 1-4.

13. Networking overview

Tuesday, 9 May 2023

Our first class explores the history of computer networks and presents the layered model of network protocols. [20] Due to the layered model, application layer protocols like DNS, HTTP, and SMTP can be built in isolation from concerns like routing and reliable delivery. Similarly, TCP can focus on transmission and IP can focus on routing without having to consider the application data.

Pre-class Work

In preparation for class, please watch the following short videos:

- The 4 Layer Internet Model
- The IP Service
- Life of a Packet
- Encapsulation
- Memory, Byte Order, and Packet Formats

Our objective with this pre-class work is to build a more complete mental model of the various pieces of computer networks, and how they all interact to facilitate something like "The Internet".

Once you have a grasp of the layered model of network protocols, please attempt the exercise described below, which involves parsing

[20] We use a simplified 5-layer version of the OSI 7-layer networking model, as does the K&R book.

The OSI model was published in 1984, approximately 10 year after the release of TCP/IP. It was intended not only to serve as a rationalized separation of concerns for network protocols, but to replace them all with alternatives designed to be cleaner and better designed than the relatively makeshift TCP/IP and friends. As is often the case with design-by-committee protocols, they never caught on.

What did stick though, due to historical use in textbooks, is the OSI 7 layer model, even if not all 7 are quite reflected in the protocols in use today. As a consequence, many people will use variants of the 7 layer model, as we do. See this Wikipedia section for a comparison of various models, and the article OSI: The Internet That Wasn't for a fascinating account of the battle between OSI and TCP/IP.

Kurose and Ross	OSI model
Five layers	*Seven layers*
"Five-layer Internet model"	OSI model
Application	Application
	Presentation
	Session
Transport	Transport
Network	Network
Data link	Data link
Physical	Physical

a file containing captured network packets, and reconstituting a downloaded image.

Further Resources

We particularly recommend the book *Networking: A Top Down Approach* by Kurose and Ross. Chapter 1 is relevant to this class.

14. DNS and HTTP

Friday, 12 May 2023

There are perhaps 100 or more application layer protocols; they are common enough that you may be involved in developing one at some point.[21] While we cannot cover them all, our explorations of DNS and HTTP in this lesson should give us a good overall idea of the key concerns and design decisions behind any other application layer protocols.

Many are surprised to learn that DNS is an application layer protocol, and that to resolve a URL into an IP address we must make a network request to a DNS server (identified by its IP address!).[22] This lesson will elucidate how this is possible, and give us a stronger understanding of one of the major services that makes the Internet user friendly.

HTTP has become a ubiquitous protocol employed well beyond its original purpose of delivering HTML pages to browsers.[23] Given the commonality and usefulness of the protocol, we will spend this class digging into some of the details of the HTTP specification as well as discussing the most important differences between HTTP/1.1 and HTTP/2.

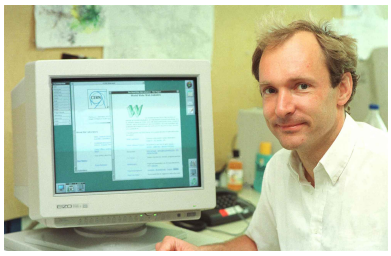
Pre-class Work

In order to send a DNS message—just like any network application message—we must know the protocol well enough to construct the message correctly. But that's not quite enough! We also require a means of transmitting the message, by having our operating system wrap it in the necessary transport, network and link layer meta data, and to actually transmit the resulting frames over the network. The standard mechanism for achieving this is the socket.[24]

[21] Consider that in April 2001, Bram Cohen quit his job and started designing what would become BitTorrent. He released the first implementation 2 months later.

[22] DNS was first designed in 1983. Astonishingly, until its deployment, the mapping between names and addresses was maintained in a simple text file HOSTS.TXT on a server at SRI International (previously Stanford Research Institute) from which every host on the network downloaded a copy! Here is the HOSTS.TXT file from early 1983.

[23] Despite the ubiquity of the World Wide Web, few non-engineers recognize that it is quite distinct from the Internet, and developed over 2 decades later. The Internet grew directly out of ARPANET, which commenced development in the late 1960s, whereas Sir Tim Berners-Lee conceived of and first prototyped the Web—including its key protocols HTTP and HTML—in 1989. Pictured below is Berners-Lee at CERN in 1994, demonstrating web pages.



[24] To be specific, it is the "BSD socket" or "Berkeley socket", a specific

As a pre-class exercise, please follow the instructions below to build a simple DNS client, both as our first introduction to socket programming, and to further improve our understanding of this important application layer protocol.

For some concrete background on how DNS works, please also watch the first 10 minutes of the video Why was Facebook down for five hours?. The remainder of the video is also worthwhile, but not specific to this class.

Since most students have some familiarity with HTTP (and since we already have a substantial amount of pre-class work for this class) there's no assigned work for the HTTP portion. If time permits however, please skim through the short online book http2 explained focusing on the major new features of HTTP/2 at a high level.

Further Resources

K&R covers both DNS and HTTP in chapter 2.

The most illustrative RFCs related to DNS are RFC 1034 and RFC 1035.

For an approachable and fascinating history of DNS, see the paper Development of the Domain Name System.

HTTP/1.0, HTTP/1.1 and HTTP/2 are specified in RFCs 1945, 2616 and 7540 respectively. If you want to really understand these protocols, you should go to the source of truth.

For a general overview of the features in HTTP/2, see the website HTTP/2 FAQs. For a great video on its performance implications, see Yesterday's perf best-practices are today's HTTP/2 anti-patterns.

15. Transport, Network and Link Layers

Tuesday, 16 May 2023

This class briefly covers a number of topics which could easily constitute entire courses of their own, including:

- IP addresses, subnets and autonomous systems
- How routers work and how routing tables are maintained
- How TCP achieves reliable delivery

- The basics of congestion control and flow control
- The very basics of link layer protocols like Ethernet and WiFi

The objective of this class is to ensure that the network layers below the application layer are not a total mystery to you! It should hopefuly pique your curiosity and provide a mental framework for further exploration of these topics.

Pre-class Work

For some background on the topics we'll cover in this class, we suggest the following videos:

- Principle: Packet Switching
- Names and Addresses: IPv4
- Longest Prefix Match
- The TCP Service Model
- The ICMP Service Model
- Flow Control
- Flow Control II
- Retransmission Strategies

Given the significant number of topics covered, the programming exercise for this class is optional (but rewarding, in our opinion). In it you will implement a simple clone of `traceroute` using raw sockets and ICMP ping requests. Traceroute is a useful little program for exploring the structure of the internet, so building a clone will help reinforce some concepts related to routing. It will also be an excuse to work directly with raw sockets, which will allow us to send ICMP messages.

Further Resources

K&R covers the transport layer in chapter 3, the network layer in chapters 4 and 5, and the link layer in chapter 6. Did we already mention that we're trying to cover a lot in this class?

For historical resources, see Vint Cerf and Bob Kahn's early paper A Protocol for Packet Network Intercommunication. The first RFCs for TCP and UDP are RFC 793 and RFC 768 respectively.

The Computer History Museum has produced a series of video-based oral histories of early pioneers of computing. Bob Kahn's interview by Vint Cerf (part 1 and part 2) is 5.5hrs in total but thoroughly fascinating.

For a quick (2min) animation of the Go-Back-N sliding window protocol, see this video.

For more information on QUIC, see the video QUIC: next generation multiplexed transport over UDP and draft protocol: QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2.

The original RFCs for IPv4 and IPv6 are RFC 791 and RFC 2460 respectively. ICMPv6 is defined in RFC 4443.

The IEEE 802.11 standard is not light reading, but may be interesting to skim through and as a resource. For ARP, the corresponding RFC is RFC 826.

For an account of the early development of Ethernet, see this oral history of Bob Metcalfe, its inventor.

16. Overview of distributed systems

Friday, 19 May 2023

In our final class we'll cover the primary challenge of distributed systems, namely how to use replication and partitioning to achieve reliability and scalability (and at what cost!)

Reliable software systems are challenging to build, and their failures are sometimes catastrophic. The Ariane 5 rocket explosion was a costly failure resulting from inappropriate code reuse, an integer overflow, inadequate redundancy and miscommunication between subsystems.

Pre-class Work

In preparation for class, please read How Complex Systems Fail, a short but compelling summary of the key behavioral characteristics of complex systems.

Further Resources

Our preferred textbook for distributed systems is *Designing Data-Intensive Applications* by Martin Kleppmann.

An exemplary case study of system failure is Nancy Leveson's on the Therac-25. For a brief description of the Ariane 5 explosion, see this

one by James Gleick. The particular Github incident discussed in class is detailed here.

For an extensive list of post mortems, see this one collated by Dan Luu and many other contributors.