Computer Systems Learning Resources
A Recommendation List

Zhenbang You

January 11, 2022

Last Updated: March 7, 2024

The latest version can always be found at https://www.overleaf.com/read/txqjnjxyxqqx

1 Preliminary

Disclaimer

- This list focuses on *breadth* rather than *depth*.
- This text mainly focuses on *software systems*, with less concentration on topics such as *VLSI*, *storage systems*, and *embedded systems*, although these are mentioned in some of the following resources.

Directions

- The most fundamental components of computer systems are:
  1. Computer Architecture
  2. Operating Systems (OS)
  3. Computer Networks
  4. Compilers
  5. Programming Languages (PL)

  Master them before moving on to other parts.

  Section 2-10 is the core of this list.
- The resources in each section are list in an order *suitable for learning*.
  - If time is limited, you may just read the first (or the first few) books, and move on to the next section.
  - To understand some advanced books, knowledge of subsequent sections may be required.
- The references help you identify the *authors* rather than *latest versions*.
- Given the rapid development of computer science, always read the latest version.
  - Tip: search *Amazon* for the latest version. The version number in this text may be outdated with new publications.

How to Read a Book?
Three passes.

1. Fast: get the main idea and leave out all the details.

2. Exhaustive: understanding the logic and virtually all the details; few really difficult points can be left out.

3. Deep: raise questions, and focus on interesting or difficult points.

Here is a famous paper called "How to Read a Paper" which discusses a similar issue:
https://web.stanford.edu/class/cs245/readings/how-to-read-a-paper.pdf

Why do you recommend so many resources about application?
Learn how to use it before learning how to build it.

Prerequisites

- CSAPP (3rd Edition) [16]
- C/C++
- Data Structures and Algorithms
  - Introduction to Algorithms (4th Edition) [19].
- Probability
  - Elements of *stochastic processes*, especially some basic conclusions and corollaries, are helpful.
- Logic, Automata and Complexity

Special Note about Prerequisites:
The motivation of specifying prerequisites is to help you figure out good learning roadmap. Even if you do not satisfy prerequisites, you can still learn the topic; however, in this case, if you have difficulties during learning, you may refer to prerequisites to see whether you need learn something else first.

What if I want more?

1. Search for more courses offered by top universities.

   - **How to find them?**
     (a) For a given university, search for its course list (note that the keyword you are searching for may not appear in the course names).
     Example:
     - Stanford: https://cs.stanford.edu/academicz/courses
     - UC Berkeley: https://www2.eecs.berkeley.edu/Courses/CS/
     - MIT: http://catalog.mit.edu/subjects/6/

     (b) To save time by scanning the entire course list, you can find the curriculum or program sheet of the Bachelor's/Master's degree.
     Example:
     - Stanford Computer Science Master's Program Sheets:
       https://cs.stanford.edu/academicz/current-masters/masters-program-sheets/programsheets
     - UC Berkeley CS Major Degree Requirements (Undergraduate):
       i. Lower Division:
       https://eecs.berkeley.edu/resources/undergrads/cs/degree-reqs-lowerdiv
       ii. Upper Division:
       https://eecs.berkeley.edu/resources/undergrads/cs/degree-reqs-upperdiv

     (c) The naming convention of course numbers helps you save time.

     Besides, courses provide *practice* opportunities, which is key to learning computer systems.

2. Find cross-cutting areas like machine learning systems. You can always find cutting-edge technologies and research hotspots here. Besides, computer systems are extremely powerful when different modules work together.

- Again, searching for advanced courses helps you discover these areas!

How to Avoid Forgetting?

- Repetition (multiple passes)
- Practice
- Questioning
- Discussion

2 Computer Architecture

1. Computer Organization and Design: The Hardware-Software Interface
   (5th Edition [44]/RISC-V Edition [43]/ARM Edition [45])
   - The original version is based on MIPS. RISC-V version is recommended; after reading this version, you may proceed on ARM version which is a good book for learning ARM.
2. Digital Design and Computer Architecture
   (2nd Edition [27]/RISC-V Edition [28]/ARM Edition [26])
   - You may just read Chap 1-5.
   - The original version is based on MIPS.
3. The RISC-V Reader An Open Architecture Atlas [42]
4. Computer Architecture: A Quantitative Approach (6th Edition) [29]
   - You may leave out appendices the first time you read this book.
   - Difficult as it may be, this book is just "the second book for novices". If you want to have a deep understanding of a specific topic, do go to read official tutorials/documentations such as those of NVIDIA.
   - When having some experiences on *parallel computing*, read corresponding chapters of this book again, you will surely gain some new understanding.
   - Based on my personal experiences, multiple passes are needed to gain thorough comprehension of this masterpiece.
5. RISC-V Privileged Architecture (slides)
   https://riscv.org/wp-content/uploads/2018/05/riscv-privileged-BCN.v7-2.pdf
   - A wonderful slide on RISC-V privileged architecture, as well as the core problem "what is the privileged architecture".
   - The video of this lecture can be found at
     https://www.youtube.com/watch?v=fxLXvrLN5jA
   - Most of the books on computer architecture discuss little about *privileged architecture*, resulting in great difficulties understanding the OS kernel. Always keep in mind that *ISA* consists of both the unprivileged architecture and the privileged architecture.
6. RISC-V specifications: https://riscv.org/technical/specifications/
   - Elaborate specifications as they may be, they are indeed excellent books for both neophytes and specialists!
   - Appendix A: RVWMO Explanatory Material of "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA" is a brilliant tutorial for **Memory Consistency**!
7. A New Golden Age for Computer Architecture (a Turing Lecture with full text)
   https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext
   - The famous Turing Lecture by Hennessy and Patterson. What wonderful insights of masters!

8. Prerequisites:
   - PL
     - Java.
9. To practice you knowledge of computer architecture, there are basically two ways:
   - VLSI,
   - Parallel computing.

3 Operating Systems (OS)

3.1 Principles of Operating Systems

1. Books
   (a) Operating Systems: Three Easy Pieces: https://pages.cs.wisc.edu/ remzi/OSTEP/
   - Due to the research interest of the authors, this book puts much emphasis on File Systems. You may leave out some chapters of this part the first time you read it.
   - This is just an introductory-level book, read more after finish this!

   (b) Operating Systems Principles & Practice (2nd Edition)
   - Four volumes:
     - Volume I: Kernels and Processes [7]
     - Volume II: Concurrency [4]
     - Volume III: Memory Management [5]
     - Volume IV: Persistent Storage [6]

   (c) Operating Systems Concepts (10th Edition) [46]
   - Very up-to-date. Can be an alternative to the previous one. You do not need to read both.

   (d) xv6 source and text: https://pdos.csail.mit.edu/6.828/2021/xv6.html
   - Hands-on experiences with a real OS is indispensable, and **xv6** is an excellent starting point!

   (e) Supplemental books
   i. Linux Kernel Development (4rd Edition) [51]
   ii. Understanding the Linux Kernel (3rd Edition) [14]
   iii. Linux Device Drivers (3rd Edition) [52]
   iv. Understanding Linux network internals (1st Edition) [12]
2. Courses
   - Lab-based
     (a) Stanford CS 140E Operating Systems Design and Implementation
     - **Rust** version: https://cs140e.sergio.bz/
     - **C** version: https://github.com/dddrrreee/cs140e-22win

     Uniqueness:
     i. "This course differs from most OS courses in that it uses **real hardware** instead of a fake simulator, and almost all of the code will be written by you."

     (b) Stanford CS 240LX Advanced Systems Laboratory, Accelerated:
     https://github.com/dddrrreee/cs140e-22win
     Uniqueness:
     i. "Our code will run "bare-metal" (without an operating system) on the widely-used ARM-based raspberry pi."
   - Paper-based

(a) Stanford CS 240 Advanced Topics in Operating Systems: http://web.stanford.edu/class/cs240/

3. Prerequisites:

   - Compulsory:
     (a) Computer Architecture (in particular, privileged architecture)
   - PL
     - **Java**: JVM, GC, Thread, Monitor.
     - **Go**: Goroutine, Channel, CSP (Communication Sequential Process), Asynchrony.

4. **Suggestions**:

   - four passes to learn *OS*
     (a) How to use: user/programmer's perspective, top-down. This pass is assisted by CSAPP and resources about *Linux Programming*.
     (b) How to build (build a usable one): builder's perspective, bottom-up.
     (c) How to design (build a good one if there is no compatibility issues): both perspectives. This pass can only be down with knowledge of all the major parts of computer systems.
     (d) Advanced and cross-cutting issues: *distributed systems*, *cloud computing*,...
   - Always think about the interaction and cooperation between
     - OS & hardware,
     - OS & computer networks,
     - OS & PL,
     - OS & DB.

     Also think about whether their boundary can be and should be redefined.

     DO NOT overfit *Linux*! Not everything of it is reasonable and well-suited to the current need!

3.2 Linux Programming

1. Linux man pages

   - "man" command in Linux shell, like "man fork" or "man 2 fork" where "2" specifies the volume.
   - There are multiple sources to find the man pages, and according to my experiences, I use *man7* most of the time, but the one provided by *Arch Linux* seems to be more up-to-date.

2. The Linux Programming Interface: https://man7.org/tlpi/

3. System Interfaces: Table of Contents
   https://pubs.opengroup.org/onlinepubs/009604499/functions/contents.html

   - Sometimes the man page is not enough, e.g., options of sockets.

4. Advanced programming in the UNIX environment (3rd Edition) [57]

4 Computer Networks

1. Books

   (a) Computer Networking: A Top Down Approach (8th Edition) [32]
   (b) Supplementary (by W. Richard Stevens)
   - UNIX Network Programming
     - Volume 1, 3rd Edition: The Sockets Networking API [56]

  - Volume 2, 2nd Edition: Interprocess Communications [49]
- TCP/IP Illustrated
  - Volume 1: The Protocols (2nd Edition) [21]
  - Volume 2: The Implementation [58]
  - Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols [59]

2. Courses

   (a) Stanford CS 144 Introduction to Computer Networking: https://cs144.github.io/

   (b) Stanford CS 244 Advanced Topics in Networking: https://2022-cs244.github.io/schedule/

   (c) Stanford CS 249I The Modern Internet: https://cs249i.stanford.edu/

   (d) Stanford CS 344 Topics in Computer Networks (a.k.a. Build an Internet Router): https://cs344-stanford.github.io/

3. Prerequisites:

   - Compulsory:

     (a) Operating Systems

   - PL
     - **Java**.
     - **Python**: Similar but much simpler socket interface than POSIX.
     - **Go**: RPC.

5 Compilers

The position of compilers in computer systems:

1. *Compilers* and *programming languages* jointly construct the foundation of *software*.

2. Techniques in several areas are heavily exploited in compilers:

   (a) Computer Architecture

   (b) Programming Languages

   (c) Theoretical Computer Science (automata, algorithms, data structures)

   (d) Discrete Mathematics

   (e) Algebra

3. Compilers play important roles in hardware-software codesign, since they are the bridge between hardware and programming languages.

4. Never regard compilers merely as translation tools from programming languages to assembly code. In a broad sense, any automatic technique can be viewed as part of compiler techniques (and automation is based on formalization).

1. Books

   (a) Compilers: Principles, Techniques and Tools (2nd Edition) [3]
   - "Dragon Book".
   - Well-known but a little obsolete; still wonderful for new-comers.

   (b) Modern Compiler Implementation in C [9]/Java (2nd Edition) [10]/ML [8]
   - "Tiger Book".

   (c) Advanced Compiler Design Implementation [39]
   - "Whale Book".

(d) Engineering a Compiler (2nd Edition) [18]

(e) Essentials of Compilation: An Incremental Approach
https://jeapostrophe.github.io/courses/2018/spring/406/notes/book.pdf

(f) Implementing Funtional Languages: a tutorial
https://www.microsoft.com/en-us/research/wp-content/uploads/1992/01/student.pdf

2. Courses

(a) Stanford CS 143 Compilers: https://web.stanford.edu/class/cs143/

(b) Stanford CS 243 Program Analysis and Optimization:
https://suif.stanford.edu/ courses/cs243/

(c) Stanford CS 343D Domain-Specific Programming Models and Compilers:
https://cs343d.github.io/

3. Programming suggestions:

(a) Try functional languages!
Pattern matching, *combinators*, *monadic design*, and powerful *type systems* (in particular, *algebraic data types*) are all your good friends!

(b) There are two kinds of (domain-specific) languages: internal and external, and the latter means a language embedded in another host language. It can be quite fun to write a compiler for an internal (domain-specific) language. Good choices for host languages: Scala, Haskell, C++.

4. Parser Generators

(a) C/C++: Lex & Yacc [34]

(b) OCaml: OCamllex & Menhir
https://dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html

(c) Haskell: Parser Combinators
There are various of them in Haskell.
Some tutorials:

i. Real World Haskell:
https://book.realworldhaskell.org/read/using-parsec.html

ii. Wikibook:
https://wiki.haskell.org/Parsing_a_simple_imperative_language

(d) F#: FsLex & FsYacc
Tutorial from Wikibook:
https://en.wikibooks.org/wiki/F_Sharp_Programming/Lexing_and_Parsing

(e) Scala: Scala Parser Combinators
https://github.com/scala/scala-parser-combinators

(f) ANTLR
https://www.antlr.org/about.html

These tools are convenient, but writing lexers and parsers manually is also a lot of fun!

(a) A tutorial about writing parser generators manually in Scala: Chapter 9, Functional Programming in Scala [17]

5. LLVM

(a) Kaleidoscope Tutorial
https://llvm.org/docs/tutorial/index.html

(b) LLVM Documentation Home
https://llvm.org/docs/

(c) LLVM for Graduate Student
https://www.cs.cornell.edu/ asampson/blog/llvm.html

Special note: renowned as LLVM is, it is indeed not very convenient. Besides, its APIs lack back compatibility.

Apart from C/C++, LLVM also has bindings for other languages like OCaml, Haskell, Rust, Go, C#, F#, Node.js, Go, Python.

6. Some interesting open-source compiler-related projects:
   (a) Halide/TVM
   (b) TACO
   (c) LLVM/MLIR
   (d) XLA
   (e) QEMU
7. Prerequisites:
   - Compulsory:
     (a) Computer Architecture: ILP (Instruction-Level Parallelism), Memory Hierarchy
         i. This is compulsory only when you want to learn compiler optimizations.
   - Recommended:
     (a) TCS (Theoretical Computer Science)
         i. Introduction to the Theory of Computation (3rd Edition) [55]
            – Chapter 1 and 2 are enough. For compilers, understanding how to use these automata is enough. No need to worry about proofs.
     (b) Computer Architecture: DLP (Data-Level Parallelism), TLP (Thread-Level Parallelism)
     (c) Operating Systems: Thread, Context Switch
   - PL
     – **Java**.

6 Programming Languages (PL)

When talking about *PL*, we are actually dealing with two things:

- programming,
- verification.

Abbreviations:

- OOP: Object-Oriented Programming
- FP: Functional Programming

6.1 Language List

6.1.1 Programming Language List

Suggestions for Learning Order

1. Start with functional languages, since they do not need to take *memory* into account, which is always confusing and error-prone for beginners. Besides, functional programming lets you pay more attention to the description of computation rather than its implementation; this is really a blessing.
2. Which functional language to choose?
   (a) *Lisp* family is too stale and outdated.

(b) Purely functional languages like *Haskell* is also not appropriate. Think about the effort needed to understand *monad*!

(c) *Scala* is not good too, due to its complex syntax, although being fairly modern.

(d) The only remaining choice seems to be *ML* family, of which *OCaml* and *F#* are both great. *F#* is more modern and easier to learn, and the OOP in F# is more mature.

(e) However, starting from *ML* family sounds a little horrendous.

(f) Consider learning *Kotlin* first, for its simplicity and modernity, and being functional enough except the lack of full support for pattern matching.

3. With the foundations of *Kotlin* and *OCaml/F#*, it is easy to learn virtually all the mainstream languages like *Java*, *TypeScript*, and even *Scala* (although this one obviously requires more effort than the previous two); besides, you can also advance towards *Rust* (time-consuming!), and after that, the legendary *C++*. If *Rust* is too hard for you, try *Go* first.

Tips

1. **General tips for choosing an IDE**:

   (a) For languages running on *JVM*, choose *JetBrains*.

   (b) If *JetBrains* has a specific software for the language your target language, prefer it.

   (c) Otherwise, *VS Code* is always a nice choice.

   Only IDEs I have used will be listed here, so there may be other wonderful IDEs.

2. **General tips for choosing platforms for *Windows* users**:

   (a) For languages on *.NET*, *Windows* is obviously the best.

   (b) Otherwise, *WSL2* is at least not worse than *Windows*.

   (c) For old AOT (ahead-of-time) compiled languages like *C/C++*, *OCaml*, *Windows* may be a disaster (lots of compatibility issues).

   (d) For JIT (just-in-time) compiled languages and modern AOT compiled languages such as *Go* and *Rust*, it does not matter which one you choose (luckily, *Haskell* is also good on *Windows*).

   (e) *Python* is a special case where the compatibility of *Windows* is not good enough sometimes for some packages.

3. **Development toolchains matter.**

   You need not only *compilers*, *libraries* and *IDE*, but also *build tools*, *debuggers*, *test tools*, *linters*, and even *sanitizers*, *profilers*, *formatters*, etc.

4. **Frameworks** are key components of the ecosystem of a PL. Success of some PL highly relies on the its frameworks.

5. Sometimes **searching within a tutorial/documentation** gives better answer than *Google*.

6. Unless specified explicitly, always try the latest version.

- **C/C++** (Do learn the latest version of C++, or at least C++17)

  If possible, learn *Rust* first.

  - Tutorials
    1. The C Programming Language (2nd Edition) [50]
       * "**K & R**"
    2. The C++ Programming Language (4th Edition) [60]
  - Documentations

1. https://cppreference.com/
2. Boost C++ Libraries: https://www.boost.org/
    * Sometime *standard C++ libraries* are not enough. In this case, *Boost* may provide additional useful functionality. Besides, some functionality of *Boost* may be incorporated into *standard C++ libraries* in the future, as the past shows.

- Programming guidelines
    * Scott Meyers "Effective C++" book series
        · Effective C++ (3rd Edition) [36]
        · Effective Modern C++ (1st Edition) [37]
        · Effective STL (1st Edition) [35]
    * C++ Core Guidelines:
      https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- Recommended IDE: VSCode on Linux (especially WSL).
- Recommended compilers: *latest* GCC and Clang.
    * Currently, **OpenMP** support of **Clang 13** needs to be installed separately, which is not the case for **Clang 12**.
    * **GCC** can be built from source.
        · Source code address: https://github.com/gcc-mirror/gcc/tags
    * **Clang** can by downloaded directly from https://releases.llvm.org/download.html
- Build tool:
    * CMake (also works for CUDA C++).
        · **CMake** Tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/
    * GNU make: https://www.gnu.org/software/make/manual/make.html
- **GoogleTest** User's Guide: https://google.github.io/googletest/
- Linter: **Clang-Tidy**: https://clang.llvm.org/extra/clang-tidy/
    * The coding style of C/C++ is much more important than that of other languages, due to the legacy, the flexibility and the safety issues of this language.
- IDE setup guidelines:
    1. VSCode. For Windows user, choose WSL.
    2. Latest G++/Clang++.
    3. VSCode extension: Clangd.
    4. Two files: .clang-format, .clang-tidy.
    5. compiledb (to ensure the intellisense works).
    6. Makefile (can be generated by CMake).

- **Java** (Java 17 is recommended, or at least, choose an LTS version)

  Consider learning Kotlin first.

  Prerequisite languages: none.
  Pay attention to the comparison with C++, as well as JVM and JIT.

  - Tutorials
    1. Oracle online tutorial: https://docs.oracle.com/javase/tutorial/
    2. Core Java (10th Edition). Two volumes:
        * Volume I: Fundamentals [24]
        * Volume II: Advanced Features [61]
  - Documentations
    1. https://docs.oracle.com/javase/specs/
  - Programming guidelines
    * Effective Java (3rd Edition) [13]

  - GraalVM: https://www.graalvm.org/
  - Recommended IDE: IntelliJ Idea
  - Build tool (applying to all languages on JVM) : **Maven** or **Gradle**.
    * Maven in 5 Minutes:
      https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
    * All documentations can be found at:
      https://maven.apache.org/
    * Try **Maven** with **IntelliJ Idea**.
  - **Spring**: https://spring.io/quickstart

- **C#**

  For the sake of learning, C# is a (much) better alternative Java, and it is also the key to .NET ecosystem.

  1. Microsoft C# documentation: https://learn.microsoft.com/en-us/dotnet/csharp/

- **Scala** (*Scala 3* is out!)

  Scala borrows more from *Haskell* than *OCaml*.

  Prerequisite languages: none (if you find *Scala* too difficult to learn, try *Kotlin* first).

  Noteworthy features: FP.

  Multi-paradigmatic languages like *Scala*, *OCaml* and *F#* are great starting points to learn *functional programming*, since your programming style can transition gradually from *imperative programming* to *functional programming*.

  - Tutorials
    1. Tour of Scala: https://docs.scala-lang.org/tour/tour-of-scala.html
    2. Scala Book: https://docs.scala-lang.org/overviews/scala-book/introduction.html
  - Documentations
    * Guides and Overviews: https://docs.scala-lang.org/overviews/index.html
    * All documentations: https://docs.scala-lang.org/
      · *Scala 3* documentations can also be found here!
  - Recommended IDE & build tool: same as *Java*
  - For PL study, try *Scala 3*. For the sake of ecosystem, currently *Scala 2* is preferred.

- **Kotlin**

  In my mind, it is the best language for entry-level programmers.

  Prerequisite languages: none.

  Try *Scala* and *Kotlin* together.

  Noteworthy features: null safety, modest FP.

  - Tutorials
    1. Get started with Kotlin: https://kotlinlang.org/docs/getting-started.html
    2. Kotlin Coroutines:
       https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md
       * Wonderful article about the design of stackless coroutines!
  - Documentations
    1. https://kotlinlang.org/docs/
  - Recommended IDE & build tool: same as *Java*

- **Go** (*Go 1.18* with *generics* is out!)
  Prerequisite languages: none.
  Noteworthy features: concurrent programming (goroutine + channel + asynchrony, GMP model), modest OOP (compare the OOP of Java and the OOP of Go).

- Turorials
  1. A Tour of Go: https://go.dev/tour/welcome/1
  2. Effective Go: https://go.dev/doc/effective_go
  3. Official tutorials (including those for generics, fuzzing and accessing a relational databases: https://go.dev/doc/tutorial/
- Documentations
  1. All the documentations can be found at: https://go.dev/doc/
- Blogs
  * The Go Blog: https://go.dev/blog/
    · Lots of wonderful articles about the design of Go!
    For example, "Getting to Go: The Journey of Go's Garbage Collector":
    https://go.dev/blog/ismmkeynote
  * Russ Cox's Blog: https://research.swtch.com/
    · There are not only lots of brilliant blogs about go, but other interesting topics!
- Recommended IDE: GoLand, VSCode
- Build tool: https://go.dev/doc/modules/gomod-ref

- **Rust**

  A extremely high-performance, memory-safe language without garbage collection

  Most of the features of Rust can be found in either *C++* or *OCaml*; therefore, solid foundations of these two languages does help you understand the design of Rust.

  Which one to learn first, C++ or Rust? Due to the popularity of C++, most people were exposed to it before hearing of Rust. However, the design of Rust is more consistent and cleaner, less prone to be misused, also without the horrendous legacy of C++. Thus, I believe Rust is worth learning before C++.

  - Tutorials
    1. The Rust Programming Language: https://doc.rust-lang.org/book/
       * Also a good book about the design of programming languages!
    2. Rust by Example: https://doc.rust-lang.org/stable/rust-by-example/
    3. Asynchronous Programming in Rust: https://rust-lang.github.io/async-book/
    4. The Rustonomicon: https://doc.rust-lang.org/nomicon/
       * This book is about "Unsafe Rust".
  - Courses
    1. CS 110L Safety in Systems Programming:
    https://web.stanford.edu/class/cs110l/
  - Documentations
    1. The Rust Reference: https://doc.rust-lang.org/reference/
    2. https://doc.rust-lang.org/beta/
  - Lots of excellent books can be found at:
    https://www.rust-lang.org/learn
  - Recommended IDE: VSCode, IntelliJ Idea
    * Personnaly, I prefer *VSCode*, in that its analyzer runs faster and does better in type inference.
  - Some nice crates (Rust's name for packages):
    1. Thread-level parallelism: rayon.
    2. Asynchronous programming: futures + tokio/async-std/smol.
  - Build Tool: Cargo.
    * The Cargo Book: https://doc.rust-lang.org/cargo/
    * Its tutorial can also be found in "The Rust Programming Language".

- **OCaml** (*OCaml* 5.0 with *multicore support* has been released!)

  ML, the ancestor of *OCaml*, is a programming language that won his creator a Turing Award, for it is "the first language to include polymorphic type inference together with a type-safe exception-handling mechanism".

  Prerequisite languages: none.
  Familiarity with *Scala* is very helpful.

  Comparison with *Scala*:

  1. *Scala* is more like a hybrid of *imperative programming* and *functional programming*. However, *functional programming* and *imperative programming* take the primary and secondary positions in *OCaml* respectively.
  2. *Type inference* of *OCaml* is even more powerful than that of *Scala*, e.g., in *Scala*, you need to specify the types of function arguments and the return types of recursive functions.
  3. *Scala* allows *implicit casting*, which is absolutely impossible in *OCaml*. The type system of *OCaml* is rather strict and rigor.

  If you have learned either one of them, I strongly recommend you learn the other.

  - Tutorials:
    1. Real World OCaml: https://dev.realworldocaml.org/
       * *OCaml* is one of few PL that attach great importance to **performance**, which is especially rare in *functional language*. Fortunately, this book also gives introduction to the implementation of *OCaml*, in particular, temporal and spatial cost. These contents are really instructive, in that functional programmers often know little about these.
       * This book is written by people at Jane Street. Therefore, the book turns out to be practical rather than theoretical or mathematical as you may expect books about FP will do.
    2. The OCaml Manual: https://v2.ocaml.org/manual/index.html
    3. The official tutorial and documentation: https://ocaml.org/docs
  - Courses:
    * Cornell CS 3110 Functional Programming in OCaml: https://cs3110.github.io/textbook/cover.html
    * Harvard CS 51 Abstraction and Design in Computation: https://cs51.io/
  - Build tool: *dune*.
    * https://dune.readthedocs.io/en/stable/
  - Package manager and version management tool: *opam*.
  - Recommended IDE: VSCode
  - Notes:
    1. If you are using Windows, please consider WSL2 for *OCaml*.

- **F#**

  F# is quite similar to *OCaml*, and it runs on the *.NET* platform. Due to their similarity, once you have mastered either of them, it will be easy to learn the other.

  Which one to learn first? OCaml or F#?

  Either can be a good choice. For beginners, they are both good alternatives to each other. F# is more modern, thus handier and free from the stale syntax of OCaml.

  - Tutorials
    1. Beginning F# (Video Series): https://www.youtube.com/playlist?list=PLdo4fOcmZ0oUFghYOp89baYFBTGxUkC7Z

2. F# for Fun and Profit:
   https://fsharpforfunandprofit.com/site-contents/
3. F# Language Guide: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/

– Other learning materials

1. F# Software Foundation:
   https://fsharp.org/learn/
2. Microsoft:
   https://dotnet.microsoft.com/en-us/learn/fsharp

– Documentations

1. F# documentation:
   https://docs.microsoft.com/en-us/dotnet/fsharp/

– F# has nice support for asynchronous and concurrent programming:

* *async*:
  https://learn.microsoft.com/en-us/dotnet/fsharp/tutorials/async
* *task*:
  https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/task-expressions
* *MailboxProcessor* (actor model):
  https://fsharpforfunandprofit.com/posts/concurrency-actor-model/

– Recommended IDE: VSCode

- **Haskell**
  GHC recommended.

  Purely functional language.

  Notable features:

  1. purity & laziness,
  2. type class & type system,
  3. monad & algebraic design.

  Knowledge of *programming language theory* does help. Personally, I could not understand some of the most elegant design until taking Stanford CS 242.

  Elementary knowledge of *abstract algebra* helps a lot for learning algebraic design in Haskell.

  Prerequisite languages: none.
  Recommended learning order: *OCaml/F#* $\rightarrow$ *Scala* $\rightarrow$ *Haskell*.

  Key differences between *OCaml* and *Haskell*:

  1. Purity,
  2. Laziness.
  3. Haskell has a far more complex and advanced type system, while OCaml has a more complex module system.

  Haskell is known for its steep learning curve, an appropriate learning order provides you sufficient foundations to conquer it.

  – Tutorials

  1. Learn You a Haskell for Great Good!: http://learnyouahaskell.com/chapters
     * For beginners, this one is better than Real World Haskell.
  2. Real World Haskell: http://book.realworldhaskell.org/read/
  3. *Wikibooks* has really nice passages about Haskell, including tutorials for beginners and advanced topics for experienced Haskell programmers:
     https://en.wikibooks.org/wiki/Haskell

4. *HaskellWiki* is really helpful.

- All the books, courses, tutorials, documentations and various kinds of resources can be found at: https://www.haskell.org/documentation/
- Frequently used websites:
  - Hackage
  - HaskellWiki
  - Wikibook
  - Hoogle
- GHC User's Guide:
  https://downloads.haskell.org/ghc/latest/docs/users_guide
  - GHC has many important extensions that can be found here.
- Toolchain management tool: *GHCup*
  - Do not change the version of any tool outside ghcup!
  - To make *VSCode* work for us, the version of *GHC* specified in *stack* must be compatible with the *GHC* versions supported by the *Haskell Language Server*, and this specific version of *GHC* should be installed within *GHCup*, though it is not necessary to set it as default. Besides, the latency of initialization when opening a project is substantial, thus patience required.
- Build tools:
  - Stack (**preferred**): https://docs.haskellstack.org/en/stable/
    - Most of the time you just need to deal with stack.
  - Cabal (**not recommended**): https://www.haskell.org/cabal/
- The type system of Haskell is a nice example to study *Curry-Howard Isomorphism*. Here is a great article about this:
  https://en.wikibooks.org/wiki/Haskell/The_Curry%E2%80%93Howard_isomorphism : :text= The%20Curry%E2%80%93Howard%20isomorphism%2C%20hereafter,value %20that%20has%20that%20type.
- A History of Haskell:
  https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf
- Recommended IDE: VSCode
- Notes:
  1. If you are using Windows, please consider WSL2 for *OCaml*.

- **JavaScript/TypeScript/AssemblyScript**
  Prerequisite languages: none.
  The improvement of *TypeScript* over *JavaScript* is well worth studying.
  Pay attention to its *asynchronous programming* techniques:
  1. callbacks,
  2. promises (async), await.

  - Tutorials
    1. W3Schools JavaScript tutorial: https://www.w3schools.com/js/
    2. JavaScript at the Mozilla Web Docs:
       https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide
    3. W3Schools TypeScript tutorial: https://www.w3schools.com/typescript/
    4. The TypeScript Handbook:
       https://www.typescriptlang.org/docs/handbook/intro.html
    5. Node.js official tutorial: https://nodejs.dev/learn
    6. The AssemblyScript Book:
       https://www.assemblyscript.org/introduction.html
  - Books

    1. JavaScript: The Good Parts [20]
  - Documentations
    * JavaScript: MDN Web Docs
    * TypeScript Documentation: https://www.typescriptlang.org/docs/
  - Recommended IDE: VSCode
  - Resources related to *WASM* can be found *here*18.

- **Swift**

  1. Official documentations: https://www.swift.org/documentation/

- **Julia**

  Easy to learn, easy to write, with high performance.

  A nice language for *scientific computing* and *fast prototyping*, with native support for *GPU programming and distributed programming*, as well as nice support for *just-in-time (JIT)* compilation and *gradual typing*.

  Do not **overuse** it, e.g., I don't think it is suitable for large projects.

  1. Official documentation: https://docs.julialang.org/en/v1/

6.1.2 Theorem Prover List

1. **Lean**

   Mainly a theorem prover, also a programming language (though not so handy as languages designed to be a programming language).

   (a) Lean Documentation: https://leanprover.github.io/about/

   (b) Recommended IDE: VSCode

   (c) Tutorial of Lean3:
   https://leanprover.github.io/theorem$_p$*roving*$_i$*n*$_l$*ean*/

   (d) Version management tool: *elan*

   (e) Recommended IDE: VSCode

2. **Coq**

   Based on *OCaml*.

   - https://coq.inria.fr/

3. **Agda**

   Based on *Haskell*.

   - https://wiki.portal.chalmers.se/agda/pmwiki.php

6.1.3 Other Languages

There is no clear boundary between PLs and libraries. Therefore, many libraries can be considered domain-specific languages (DSLs) and thus recommended below.

There are also some languages worth noticing or even learning. There belong to various programming paradigm, and have significant impact in the history of programming languages.

Array Programming

- APL, K

  - Numpy can be seen as their modern descendent. Some DSLs derived from Numpy like PyTorch and JAX are also good choices.

Logic Programming

- Prolog, Datalog, Epilog

Tensor Algebra

- Dense Tensor Algebra: Halide, TVM
- Sparse Tensor Algebra: TACO (http://tensor-compiler.org/)

6.1.4 Old Functional Languages

- Lisp dialects: Racket

Satifiability Modulo Theories (SMT) solver

- Z3, CVC5

Languages for Mathematicians

- Mathematica

6.2 Functional Programming

- Books
  - Recommended for green hands

    These books teach not only the syntax of FP, but (and more importantly) how to design software in the functional style, as well as the background and the origin of those functional features.

    1. Structure and Interpretation of Computer Programs (2nd Edition [1]/JavaScript Edition [2])
       - The original version is based on **Scheme**, an Lisp dialect. However, Lisp and its dialects are obsolete now. But the book is still well worth recommending.
       - The significance of the book goes well beyond teaching us functional programming.
    2. Functional Programming in Scala [17]
       - The 1st version is based on *Scala 2*, while the second version is based on *Scala 3*. Except for this, these two versions are basically the same.
  - Purely Functional Data Structures [40]
  - Other books
    1. Java 8 in Action: Lambdas, Streams, and Functional-style Programming (1st Edition) [62]
    2. All books about *Haskell* and *OCaml*.
- Suggested learning order in terms of languages
  1. Start from *OCaml/F#*. This is the cleanest one. Personally, I think *F#* is better.
  2. Move on to *Scala*. It has a lot more features and programming paradigms than *OCaml/F#*, thus harder to learn and prone to get confused. However, **trait** (**type class** in *Haskell*) is very helpful for learning functional design, which is missing in *OCaml* but present in *F#* (although it can be emulated by modules in *OCaml*). Besides, since *Scala* derives from *Haskell*, it is helpful for you to learn the latter.
  3. At this point, you may try some languages with lots of functional features like *Rust*.
  4. Then you can start challenge yourself with the legendary *Haskell*. If your foundation of the previous stuffs is solid enough, it will not be so scary for you to learn this language. From the perspective of the programming language theory, it does possess some nice properties like *termination* (if any evaluation order leads to termination, then natural order, adopted by *Haskell*, will terminate).

6.3 Programming Languages Theory

It is not only about *programming*, but also about *verification*.

- Courses
  - Stanford CS 242 Programming Languages:
    - Will Crichton: https://stanford-cs242.github.io/f19/ (more practical, more open-source materials like lecture notes and assignments)
      - Languages used: OCaml, WASM, Rust
    - Alex Aiken: https://web.stanford.edu/class/cs242/materials.html (more theoretical, better breadth and depth)
      - Languages suggested being learned before this course: OCaml, Haskell, Rust
      - Languages used:
        1. C++, Java: mainstream OOP languages
        2. OCaml, Haskell: statically typed functional languages; monad in Haskell
        3. Python, JavaScript: mainstream dynamically typed languages
        4. Rust: ownership (to implement type states)
        5. Racket: call/cc to make use of continuations
        6. Lean: theorem prover
        7. NumPy: array programming with combinators
      - Other languages useful for understanding the background:
        1. Go: channel
        2. TypeScript: gradual typing
        3. Kotlin: coroutine
        4. Scala: a complicated type system supporting most OOP and FP features
      - Prerequisites:
      - Compulsory:
        1. Propositional Logic
        2. Functional Programming
      - Recommended:
        1. Compilers
        2. Programming Language Implementation
      - Weakly Recommended:
        1. Operating Systems
        2. Computer Architecture
        3. Theoretical Computer Science
      - The background of this course is extremely profound, and thus the comprehensive knowledge of computer science, in particular, programming languages of different paradigms, does make a difference.
  - Stanford CS 151 Logic Programming: http://logicprogramming.stanford.edu/stanford/lessons.php
- Books
  1. Foundations for Programming Languages [38]
  2. Types and Programming Languages (1st Edition) [47]
     - Based on *OCaml*.
  3. Advanced Topics in Types and Programming Languages (1st Edition) [48]
  4. Practical foundations for programming languages (2nd Edition) [25]
- Prerequisites:

  - Compulsory:
    1. One OOP language (*Java* is OK, but *C++/Python* is not enough), one FP language. Basic knowledge suffices.
       * Personally, I recommend *Scala/OCaml/F#* (personally I think F# is the best) + *Rust*. They cover almost all the programming paradigms in modern and frequently used languages.
  - Recommended
    1. Computer Architecture
    2. Compilers
    3. Operating systems

Suggestions

- No need to master an entire PL in one shot; instead, study part of it when needed.
- PLs develop rapidly. To catch up with the latest development, refer to online documentations/tutorials/blogs besides books.
- Mastered at least one *modern* language in each of the following paradigms:
  - Procedural
  - Object-Oriented
  - Functional

  Note that modern languages like *Go*, *Scala*, *Kotlin* and *Rust* can be greatly different than old ones like *Java* and *Python*. As a special case, although modern *C++* (C++11 and later) is really modern, but there are inevitably a great number of legacies, so C++ can be considered as a mixture.
- PL can be viewed from at least three perspectives:
  - Computer systems
  - Software engineering
  - Software theory
    * e.g., relationship between programming paradigms and Church-Turing thesis.

  Therefore, you can always find lots of cross-cutting issues in PL.
- A complete list of *programming paradigms*:
  https://en.wikipedia.org/wiki/Programming_paradigm
- Pay attention to three kinds of *safety*
  - Type safety
  - Memory safety
  - Thread safety
- Pay attention to the *memory consistency model* (always called *memory model* in this context) of each language, although this virtually has no impact on *application-level programming*.
- *Concrete examples* always help a lot for understanding *abstract concepts*, and this is also one of the reasons why you should master several languages.
- Pay attention to the interoperation between a certain language with C/C++ (and languages on JVM with Java).

7 Parallel Computing

Parallel computing consists of three components:

- Architecture (system)
- Programming (application)
- Algorithms (theory)

Parallel computing is an natural extension to *computer architecture*.
Special Notes

- Do read *Computer Architecture: A Quantitative Approach (CAAQA)* before diving into this, and revisit that masterpiece after having some hands-on experience of parallel computing!
- *Parallel Computing* without *Memory Optimization* is **ridiculous**!
- Do learn how to analyze performance bottleneck, which is key to the success of parallel programs. Knowledge like that of *computer architecture* and *operating systems* can be extremely helpful.
- *Parallel computing* should include *distributed computing*; however, the latter is not involved in this section.

7.1 Platforms

- **CUDA** (Category: data parallel (logically), GPU)
  - CUDA by Example (1st Edition) [53]
    - A little obsolete, but the ideas are still well-presented. If your foundation is good enough, go to the following two documentations directly, and these two are highly recommended.
  - CUDA C++ Programming Guide:
    https://docs.nvidia.com/cuda/cuda-c-programming-guide/
  - CUDA C++ Best Practices Guide:
    https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
  - Thrust: https://docs.nvidia.com/cuda/thrust/index.html
    - STL in CUDA, providing some high-level abstractions.
  - All documentations can be found at: https://docs.nvidia.com/cuda/

  CUDA is also fantastic for *Asynchronous Programming* and *Heterogeneous Programming*! You can also have a taste of *Compute Hierarchy* with CUDA!
- Intel ISPC (Category: data parallel (logically), CPU)
  - Intel ISPC User's Guide: https://ispc.github.io/ispc.html
- **CPU intrinsics** (Category: data parallel (logically), CPU)
  - **x86 intrinsics** (Category: DLP, multimedia SIMD instruction set extensions)
    https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
  - **ARM SVE2 intrinsics** (Category: DLP, vector architecture)
    https://developer.arm.com/documentation/102340/0001/Program-with-SVE2
  - **ARM Neon intrinsics** (Category: DLP, multimedia SIMD instruction set extensions)
    https://developer.arm.com/documentation/102467/0100/Why-Neon-Intrinsics-

- **MPI** (Category: task parallel (logically), message passing)
  - MPI Tutorial: https://mpitutorial.com/tutorials/
  - YouTube video
    - MPI Basics: https://www.youtube.com/watch?v=c0C9mQaxsD4
    - MPI Advanced: https://www.youtube.com/watch?v=q9OfXis50Rg
- **OpenMP** (Catrgory: data parallel (logically), shared memory)
  - Tim Mattson's (Intel) "Introduction to OpenMP" (2013) on YouTube
    - Video: https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG
    - Slides: https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

  For those familiar with *pthread*, it is quite easy to learn **OpenMP**.
- High-level DSLs
  1. PyTorch
     - PyTorch 2.0 has been released!
     - The official site (https://pytorch.org/) provides enough learning resources of high quality
  2. JAX: https://jax.readthedocs.io/en/latest/

7.2 Categories

The essence of *parallel computing* is *the lack of dependencies*.

- ILP (Instruction-Level Parallelism)
  Mathematical model of ILP – DAG (Directed Acyclic Graph):
  - Node: a stage of a instruction.
  - Edge: dependency (data dependency, control dependency, name dependency) between a pair of nodes.

  Goal: eliminate dependencies (edges), and exploit the lack of dependencies between nodes.
- DLP (Data-Level Parallelism)
  The essence of DLP programming: *Vectorization*.
  - To get some hands-on experiences with this, you may start with *PyTorch*, since this relieves you from some low-level details like remaining elements and memory hierarchy.
    - Official tutorial: https://pytorch.org/tutorials/
    - Also, **PyTorch** is a convenient tool to exploit GPU for parallel computing.
- TLP (Thread-Level Parallelism)
  The essence of TLP programming: **async** (concurrent control flow), **await** (synchronization).
  - Concurrency mechanisms
    1. Thread: nearly all mainstrean languages support this, e.g., C++ std::thread/jthread, Java, pthread

2. Future/Promise: most of mainstrean languages support this, e.g., Scala, C++, Java
3. Stackful Coroutine: Go, Haskell
4. Stackless Coroutine: Kotlin, Rust (async/await, Tokio), F# (async/task), Haskell

Aside:

1. Here is a good summary of *Asynchronous programming techniques* provided in the tutorial of *Kotlin*: https://kotlinlang.org/docs/async-programming.html
2. The essence of *Asynchronous Programming* is *async/await*, as well as *suspension points*.
   * For *Threads* and *Stackful Coroutines*, every point is a suspension point, while for *Stackless Coroutines*, only a fraction of points can be suspension points and they are declared explicitly.
3. For implementations, figure out what is "**continuation**" and how it varies in *Processes*, *Threads*, *Stackful Coroutines*, and *Stackless Coroutines*. Also think about the relation between *continuation* and *suspension points*.

– Synchronization mechanisms

* Shared memory
  1. Mutex, Condition Variable: most languages.
  2. Monitor: Java. For C++, this can be readily emulated by *RAII*.
  3. Atomic variables/operations: most languages.
  4. Barrier: most languages, especially popular in data parallel programming like CUDA.
  5. Read Write Lock: most languages.
  6. Software Transactional Memory: Haskell.

  Many widely used mechanisms are not listed here.
* Message passing
  1. Channel + Select: Go, Kotlin.
  2. Actor Model:
     · Akka (with Java/Scala interface): https://doc.akka.io/docs/akka/current/typed/guide/introduction.html
     · Kotlin: https://kotlinlang.org/docs/shared-mutable-state-and-concurrency.html#actors
     · F# MailboxProcessor: https://fsharpforfunandprofit.com/posts/concurrency-actor-model/
* High-level encapsulations
  1. Thread-safe collections
     · Java: java.util.concurrent: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/package-summary.html

– Aside – comparison among *concurrency*, *parallelism*, *asynchrony*:

1. Concurrency: when multiple computations execute in overlapping time periods.
2. Parallelism: when multiple computations or several parts of a single computation run at exactly the same time.
3. Asynchrony: when one or more computations can execute separately from the main program flow.

(Source: https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/async)

7.3 Principles of Parallel Computing

- Stanford CS 149 Parallel Computing: https://gfxcourses.stanford.edu/cs149/fall21/
  - Programming languages and platforms used:
    1. C++ and std::thread
    2. ISPC
    3. CUDA
    4. OpenMP
    5. AVX intrinsics
- Stanford CS315B Parallel Programming: https://web.stanford.edu/class/cs315b/
- Stanford CME 323 Distributed Algorithms and Optimization: https://stanford.edu/ rezab/classes/cme323/S20/

Prerequisites

- Compulsory:
  1. Computer Architecture
- Recommended:
  1. **Functional Programming** (strongly recommended)
  2. Operating Systems
  3. Compilers

7.4 Memory Model

It is also called "Memory Consistency Model" or "Memory Order" sometimes.
In my opinion, it is the *trickiest* part in *concurrent programming*.
This is a cross-cutting issue, involving the following fields:

1. computer architecture: memory consistency;
2. concurrent programming: lock-free programming;
3. compilers: reordering (including code scheduling, speculation, partial redundancy elimination, common subexpression elimination, copy propagation, redundant read/write elimination, etc), register allocation.

This issue occurs in the following scenarios:

1. lock-free programming,
2. compiler development,
3. development of concurrency related libraries
4. OS development.

As an example, for C++, the following things are involved:

1. memory order (served as argument to the functions/methods of the following),
2. atomic,
3. atomic thread fence,
4. atomic signal fence (compiler fence).

Memory models of mainstream PLs can be divided into several categories:

1. C++, Rust, Go;
2. Java;
3. OCaml.

Concrete descriptions can be found in their respective official documentation. They are worth reading, but too inaccessible for green hands. There are some more newbie-friendly resources:

1. Preshing's blog posts (for C++):
   https://preshing.com/20120913/acquire-and-release-semantics/
   - There are many related articles in his blogs and I highly recommend them!
2. Rust Atomics and Locks: https://marabos.nl/atomics/preface.html
   - Chapter 2, 3, 7.
3. The Java Memory Model:
   http://www.cs.umd.edu/ pugh/java/memoryModel/

There are also some paper worth reading:

1. Boehm's papers for the C++ memory model:
   (a) Threads Cannot be Implemented as a Library:
   https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf
   (b) Foundations of the C++ Concurrency Memory Model:
   https://www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf
2. The Java Memory Model:
   https://rsim.cs.uiuc.edu/Pubs/popl05.pdf
3. Bounding Data Races in Space and Time (for OCaml memory model):
   https://dl.acm.org/doi/pdf/10.1145/3192366.3192421

Tips:

1. Compiler Explorer (https://godbolt.org/) will be your good friend.
2. Try different PLs and ISAs!
3. Sometimes you need to refer to ISA specifications.

8 Asynchronous Programming

Notes
While parallel computing tackles the performance engineering of compute-bound programs, asynchronous programming deals with the performance engineering of I/O-bound programs, and the latter is generally harder than the former.

8.1 Async Support in Programming Languages

8.1.1 Async Framework

They come in two flavors:

1. **Stackful Coroutines**

   They are also called user threads or green threads. Their programming model is the same as kernel threads.

   List of famous PL that supports this:

(a) Java 21 virtual threads.

(b) Go goroutines.

2. **Stackless Coroutines**

   This is sometimes just called asynchronous programming in the manual/tutorials of PLs.

   Typically this kind of programming paradigm involves two keywords: async, await

   List of famous PL that supports this:

   (a) Rust asynchronous programming.

   (b) Kotlin coroutines.

   (c) C#/F# asynchronous programming.

   (d) Python asyncio

   (e) JavaScript

   (f) Swift concurrency.

   (g) C++ 20 coroutines (not mature yet).

8.1.2 Support for Asynchronous Operations

Support for asynchronous operations (e.g., asynchronous I/O operations, asynchronous synchronization mechanisms) is also essential. I consider this as part of the ecosystem of asynchronous programming of a PL.

C#/F#, Rust, JavaScript do well here, but Kotlin (due to the lack of asynchrony of JVM), Swift currently don't. Worse still, C++ doesn't even has a first-party networking library...

8.2 Async Support in System Calls Level

8.2.1 Non-blocking Syscalls

- Sockets have nice support for this, while regular files generally don't.
- Linux man page is a good resource.

8.2.2 I/O Multiplexing

1. Reactive: select (deprecated), poll (deprecated), epoll

   - epoll has two modes, level-triggered and edge-triggered; both are worth learning;
   - these are mainly for network programming, and works poorly for file I/O;
   - Linux man page is a good resource for these syscalls.

2. Proactive: io_uring

   - Lord of io_uring: https://unixism.net/loti/
   - Efficient IO with io_uring: https://kernel.dk/io_uring.pdf
   - Though powerful (and arguably the future of Linux), io_uring is still in an early stage – incomplete and sometimes poor docs, dramatic improvement over versions, lots of vulnerabilities.
   - Manually installed liburing may not work with your OS.
   - io_uring provides true asynchrony of file I/O (aio is implemented by a thread pool).
   - Personally, I think io_uring has profound implications. Asking for services from OS no longer requires syscalls.
   - Two key idea of io_uring: Reduce kernel crossing and memory copy.

8.3 Kernel Bypassing Techniques

These require support from hardware.

1. RDMA
2. DPDK
3. SPDK

I don't know much about these. Mark them here and hopefully one day I will come back.

9 Databases (DB)

Special Notes

- Think about the relationship (cooperation and conflicts) between DB and OS. Also, when facing the same issues (there are lots of them) like concurrency control, how do they handle them respectively?
- *Parallel and distributed databases* and *NoSQL* are the hotspots nowadays. Do place emphasis on them!
- View DB from a full-stack perspective, that is, from systems, to applications, to theory (relation algebra). Lots of knowledge you have learned elsewhere plays an important role here. Also, you can find plenty of cross-cutting issues here.
- Pay attention to the relationship between DB and big data/data mining.

1. Books

   (a) Databases System Concepts (7th Edition) [54]

   - Partition:
     - Application: Chapter 2-11, 25, 26, 28-32
     - Systems: Chapter 12-24, 32
     - Theory: Chapter 2, 27

     As a beginner, you may read chapter 1-7, 12-19 first.
   - For Chapter 20-23, basic knowledge of *distributed systems* can be quite helpful. See "Prerequisites" for more advice.
   - Reflect on Chapter 10-11 after gaining some experiences on big data/cloud computing (e.g., programming with Spark).
   - Although Chapter 3-5 does teach a lot of SQL and these are really helpful, you should refer to the following manuals if you want to learing more specifications, especially those specific to a certain implementation of SQL.
   - CMU 15-445 and Stanford CS 145, 245 are mostly covered by this book.
   - Recommended online chapters
     i. Chapter 27: Formal-Relational Query Languages. Coverage:
        A. The tuple relational calculus
        B. The domain relational calculus
        C. *Datalog* (a nonprocedural query language based on the *logic-programming* language *Prolog*)
     ii. Chapter 32: PostgreSQL. A good case study.

   (b) Supplementary

   i. Databases Systems: The Complete Book [23]

2. Courses

(a) DB applications

- Stanford CS 145 Data Management and Data Systems: https://cs145-fa20.github.io/

(b) DB systems

- CMU 15-445/645 Intro to Databases Systems: https://15445.courses.cs.cmu.edu/fall2019/schedule.html
- Stanford CS 245 Principles of Data-Intensive Systems: https://web.stanford.edu/class/cs245/

(c) Advanced DB systems

- CMU 15-721 Advanced Databases Systems: https://15721.courses.cs.cmu.edu/spring2020/schedule.html

3. Query languages and platforms

- SQL
  - PostgreSQL
    (a) Databases System Concepts chapter 32: https://db-book.com/online-chapters-dir/32.pdf
    (b) Tutorial: https://www.postgresqltutorial.com/
    (c) Documentation & Manuals: https://www.postgresql.org/docs/
  - MySQL
    (a) Tutorial: https://www.mysqltutorial.org/
    (b) Reference manual: https://dev.mysql.com/doc/refman/8.0/en/
  - Which one should I choose?
    (a) They are two most popular open source databases.
    (b) Generally speaking, MySQL is still more popular for historic reasons, while PostgreSQL is more advanced and far more elegant.
    (c) *MySQL*'s syntax is relatively weird and the error and its error reporting is relatively vague and even misleading.
  - Tips of using specific databases: https://db-book.com/university-lab-dir/db-tips.html
- NoSQL
  - MongoDB
    (a) Get Started: https://www.mongodb.com/docs/guides/
    (b) Documentations: https://www.mongodb.com/docs/
- Redis
  (a) https://redis.io/
- Spark
  - Quick Start: https://spark.apache.org/docs/latest/quick-start.html
- .NET LINQ
  (a) https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/
- DB-Engines Ranking: https://db-engines.com/en/ranking
  - Popular databases engines and lots of relevant information can be found here.
- Advice:

(a) When studying query languages, reason about *declarative programming* (e.g., *SQL*) and *logic programming* (e.g., *Datalog*). (Aside: *logic programming* is a subset of *declarative programming*.)

4. Prerequisites:

   - Compulsory:

     (a) Data structures and algorithms: those operating data on secondary storage are particularly important

     (b) Operating systems

   - Recommended:

     (a) Computer architecture
     - Lots of similar ideas in *databases* can be found in *computer architecture*)

     (b) Computer networks
     - required for learning *parallel and distributed databases*)

     (c) Programming languages
     - Query languages help deepening our understanding of differences among programming paradigms (e.g., *imperative* vs *declarative*), as well as some feasible but rarely used programming paradigms (e.g., *logic programming*).

   - A few words about *distributed systems*
     - It is quite hard to determine which one of the following should be learned first: *parallel and distributed databases* and *distributed systems*. My personal feeling is, study of these two fields can be interleaved, since you often need knowledge of the other field when studying either one field. As a always reasonable method, study them multiple times so that you will not miss important prerequisite knowledge, if the optimal order of learning is hard to figure out.

   - PL
     - Java: JDBC.

10 Distributed Computing/Distributed Systems

1. MIT 6.824 Distributed Systems: https://pdos.csail.mit.edu/6.824/schedule.html
2. Supplementary

   (a) Princeton COS 418 Distributed Systems: https://www.cs.princeton.edu/courses/archive/fall19/cos418/

   (b) Stanford CS 244B Distributed Systems: https://www.scs.stanford.edu/22sp-cs244b/

   (c) Stanford CS 251 Cryptocurrencies and Blockchain Technologies: https://cs251.stanford.edu/

3. Prerequisites:

   - Compulsory:

     (a) Operating Systems

     (b) Computer Networks

     (c) Databases

   - Recommended:

     (a) Computer Architecture

     (b) Programming Languages
     - A little experiences with *functional programming* are quite helpful for learning *distributed computing*.

11 Cloud Computing

1. CMU 15-719 Advanced Cloud Computing:
   https://www.cs.cmu.edu/ 15719/old/spring2019/syllabus.html
2. Stanford CS349D Cloud Computing Technology:
   http://web.stanford.edu/class/cs349d/
3. Containers and container management systems
   (a) Docker
   (b) Kubernetes (K8s)
4. Prerequisites:
   - Compulsory:
     (a) Operating Systems
     (b) Computer Networks
     (c) Distributed Systems
     (d) Databases
5. **Suggestions**
   (a) There are plenty of *cross-cutting issues*. Make sure the prerequisites are met before moving on.
   (b) When reading cutting-edge papers concerning *OS*, *computer network* or *computer architecture*, you may encounter issues of *cloud computing*. At that moment, you had better have some basic knowledge about it before reading those papers.

12 Graphics

1. Stanford CS 248A Computer Graphics: Rendering, Geometry, and Image Manipulation
   https://cs248a.stanford.edu
2. Stanford CS 348K Visual Computing Systems
   https://cs348k.stanford.edu
3. Only systems-related courses are listed. For other graphics-related courses, please search CS *48 (* stands for one digit) with any suffix at
   https://explorecourses.stanford.edu/

13 Big Data

1. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (1st Edition) [31]
2. Prerequisites:
   - Compulsory
     (a) Databases
     (b) Distributed systems

14 Software Engineering

- The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition) [15]
- Design Patterns: Elements of Reusable Object-Oriented Software [22]
  - The authors are often referred to as the **Gang of Four (GoF)**.
- A Philosophy of Software Design (2nd Edition) [41]
- Write Great Code, Volume 3: Engineering Software [30]
- Materials discussing *best practice* are also valuable, which can be found at the **PL** section, as well as the **CUDA** subsection.

15 Program Analysis

Can be regarded as part of *compilers techniques* in a broad sense.

1. Software Foundations: https://softwarefoundations.cis.upenn.edu/
   - It consists of 6 volumes.
   - Volume 1 also teaches *Coq*.
2. Prerequisites:
   - Compulsory:
     (a) Compilers
     (b) Programming languages

16 Computer Security & Cryptography

Security issues are everywhere. New security issues and arise with new computer technologies invented.

Threat models are likely to differ with field varying (e.g., hardware vs OS), leading to distinct technologies.

Basic knowledge of *memory safety* of programming languages helps.

1. Textbooks
   (a) A Graduate Course in Applied Cryptography
   https://toc.cryptobook.us/
2. Courses
   (a) UC Berkeley CS 161 Computer Security: https://sp22.cs161.org/
   (b) Stanford CS 155 Computer and Network Security: https://cs155.stanford.edu/
   (c) Stanford CS 253 Web Security: https://web.stanford.edu/class/cs253/
   (d) Stanford CS 255 Introduction to Cryptography: https://crypto.stanford.edu/ dabo/cs255/
   (e) Stanford CS 350 Secure Compilation:
   https://theory.stanford.edu/ mp/mp/CS350-2019.html
   (f) Stanford CS 355 Advanced Topics in Cryptography:
   https://cs355.stanford.edu/
   (g) Stanford CS 356 Topics in Computer and Network Security: https://cs356.stanford.edu/

3. Meltdown and Spectre: https://meltdownattack.com/
4. Prerequisites:
   - Compulsory:
     (a) Operating systems
     (b) Computer networks
     (c) (Basic) Probability (for cryptography)
     (d) (Basic) Number Theory (for cryptography)
   - Recommended:
     (a) Computer architecture
     (b) Programming languages
     (c) Compilers
     (d) Databases

     Indeed, you can learn a lot about *computer security* in all of these fields. Pay special attention to *security* issues when diving in these fields.

17 Web

These are mostly *application* stuff. But these contents are frequently involved in the modern computer world, therefore, I add this section in case we may need them sometimes.

1. Stanford CS 142 Web Applications:
   https://web.stanford.edu/class/cs142/index.html
2. Stanford CS 192X Web Programming Fundamentals:
   https://web.stanford.edu/class/archive/cs/cs193x/cs193x.1176/lectures/
3. Find more tutorials and references at *W3Schools*: https://www.w3schools.com/
4. Prerequisites:
   - Recommended:
     (a) Operating systems
     (b) Computer networks
     (c) Databases

18 Instruction Set Architecture (ISA)

All of the following ISA can be learned by CSAPP or "Computer Organization and Design: The Hardware/Software Interface". However, the following resources help you go further.

- **x86**
  - Intel® 64 and IA-32 Architectures Software Developer Manuals:
    https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
  - AMD Developer Guides, Manuals & ISA Documents:
    https://developer.amd.com/resources/developer-guides-manuals/

  Master both the "Intel" format and the "AT&T" format.
- **RISC-V**
  - RISC-V Specifications: https://riscv.org/technical/specifications/

    - Wonderful books on computer architecture!

- **ARM**
  - ARM® CPU Architecture Key Documents: https://developer.arm.com/architectures/cpu-architecture
  - Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile: https://developer.arm.com/documentation/ddi0608/latest

- **PTX**
  - PTX ISA :: CUDA Toolkit Documentation: https://docs.nvidia.com/cuda/parallel-thread-execution/

- **Java Bytecode**
  - Java Language and Virtual Machine Specifications: https://docs.oracle.com/javase/specs/
    - CLick "The Java Virtual Machine Specification"
  - List of Java bytecode instructions (Wikipedia): https://en.wikipedia.org/wiki/List_of_Java_bytecode_instructions

- **WebAssembly**
  Strictly speaking, it is not an ISA, but they share a lot in common, and it is well worth studying, so I put it here.
  - Official website: https://webassembly.org/
  - Specification: https://webassembly.github.io/spec/core/index.html
  - Bringing the Web up to Speed with WebAssembly: https://people.mpi-sws.org/ rossberg/papers/Haas,%20Rossberg,%20Schuff,%20Titzer,%20Gohman,%20Wagner,%20Zakai,%20Bastien,%20Holman%20-%20Bringing%20the%20Web%20up%20to%20Speed%20with%20WebAssembly.pdf

19 Linking and Loading

1. 程序员的自我修养: 链接, 装载与库 [11]
2. Linkers and Loaders (1st Edition) [33]
3. **Suggestions**
   (a) Do not just *read books* or *play with GCC/Clang*. Do both together.
   (b) The first book plays a primary role, while the second is supplementary (however, I still recommend you read it).
   (c) These two books are relatively **obsolete** (sorry I did not find any relevant modern book). Besides, there are lots of academic errors in the first book.
   (d) These two books cover lots of knowledge. Pick the chapters you need. There is no need to read every chapter.

20 Tools

1. Git
    - Git Magic: http://www-cs-students.stanford.edu/ blynn/gitmagic/book.pdf
    - GitHub Get started: https://docs.github.com/en/get-started/quickstart/hello-world
    - GitHub Documentations: https://docs.github.com
    - Official documentations: https://git-scm.com/doc
    - *GitHub Desktop* is recommended.
2. UNIX Makefile
    - Makefile Tutorial By Example: https://makefiletutorial.com/
3. Docker
    - Official tutorials and documentations: https://docs.docker.com/
4. Shell Script
    - Shell Scripting Tutorial: https://www.shellscript.sh/
    - The following two tutorials are basically the same and are written by the same author, with the former being more newbie-friendly:
        - Bash Scripting Tutorial for Beginners: https://linuxconfig.org/bash-scripting-tutorial-for-beginners
        - Bash Scripting Tutorial: https://linuxconfig.org/bash-scripting-tutorial
    - GNU Bash manual: https://www.gnu.org/software/bash/manual/

21 Coding Style

- Google Style Guides: https://google.github.io/styleguide/
- Clang-Format Style Options: https://clang.llvm.org/docs/ClangFormatStyleOptions.html
    - *VS Code* can automatically format your code with a ".clang-format" file without installing anything!

22 Miscellany

1. Intel® Product Specifications: https://ark.intel.com
2. GCC online documentation: https://gcc.gnu.org/onlinedocs/
    - Optimize Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
    - Option Summary: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
    - Instrumentation Summary (including sanitizers and profilers): https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
3. GDB documentation: https://www.sourceware.org/gdb/documentation/
4. GNU Manuals Online: https://www.gnu.org/manual/
5. Clang 12 documentation: https://releases.llvm.org/12.0.0/tools/clang/docs/index.html
    - Change the version number in the URL to get the documentations of other versions.
6. LLVM documentation: https://llvm.org/
    - Lots of remarkable projects can be found there!

References

[1] H. Abelson and G. J. Sussman. *Structure and interpretation of computer programs*. The MIT Press, 1996.

[2] H. Abelson, G. J. Sussman, M. Henz, and T. Wrigstad. *Structure and Interpretation of Computer Programs: JavaScript Edition*. MIT Press, 2022.

[3] A. Aho, M. Lam, R. Sethi, J. Ullman, K. Cooper, L. Torczon, and S. Muchnick. Compilers: Principles, techniques and tools. 2007.

[4] T. Anderson and M. Dahlin. Operating systems principles & practice volume ii: Concurrency.

[5] T. Anderson and M. Dahlin. Operating systems principles & practice volume iii: Memory management.

[6] T. Anderson and M. Dahlin. Operating systems principles & practice volume iv: Persistent storage.

[7] T. Anderson and M. Dahlin. *Operating Systems Principles & Practice Volume I: Kernels and Processes*. Recursive books, 2014.

[8] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[9] A. W. Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[10] A. W. Appel and J. Palsberg. Modern compiler implementation in java, 2003.

[11] 俞甲子, 石凡. 程序员的自我修养: 链接, 装载与库. 电子工业出版社, 2009.

[12] C. Benvenuti. *Understanding Linux network internals*. " O'Reilly Media, Inc.", 2006.

[13] J. Bloch. *Effective java (the java series)*. Prentice Hall PTR, 2008.

[14] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[15] F. P. Brooks Jr. *The mythical man-month: essays on software engineering*. Pearson Education, 1995.

[16] R. E. Bryant and D. R. O'Hallaron. Computer systems: A programmer's perspective, 2015.

[17] P. Chiusano and R. Bjarnason. *Functional programming in Scala*. Simon and Schuster, 2014.

[18] K. D. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2011.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[20] D. Crockford. *JavaScript: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[21] K. R. Fall and W. R. Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[22] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[23] H. Garcia-Molina. *Database systems: the complete book*. Pearson Education India, 2008.

[24] I. Gvero. Core java volume i: Fundamentals, by cay s. horstmann and gary cornell. *ACM Sigsoft Software Engineering Notes*, 38(3):33–33, 2013.

[25] R. Harper. *Practical foundations for programming languages*. Cambridge University Press, 2016.

[26] S. Harris and D. Harris. Digital design and computer architecture: Arm edition, 2015.

[27] S. L. Harris and D. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[28] S. L. Harris and D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 2021.

[29] J. L. Hennessy and D. A. Patterson. Computer architecture: a quantitative approach, 2018.

[30] R. Hyde. *Write Great Code, Volume 3: Engineering Software*. No Starch Press, 2020.

[31] M. Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[32] J. Kurose, K. Ross, and J. Addison-Wesley. Computer networking: A top down approach.

[33] J. Levine. Linkers and loaders,. 1999.

[34] J. R. Levine, J. Mason, J. R. Levine, T. Mason, D. Brown, and P. Levine. *Lex & yacc*. " O'Reilly Media, Inc.", 1992.

[35] S. Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education, 2001.

[36] S. Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education, 2005.

[37] S. Meyers. *Effective modern C++: 42 specific ways to improve your use of C++ 11 and C++ 14*. " O'Reilly Media, Inc.", 2014.

[38] J. C. Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996.

[39] S. Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[40] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[41] J. K. Ousterhout. *A philosophy of software design*, volume 98. Yaknyam Press Palo Alto, 2018.

[42] D. Patterson and A. Waterman. *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[43] D. A. Patterson and J. L. Hennessy. Computer organization and design risc-v edition: The hardware software interface (the morgan kaufmann).

[44] D. A. Patterson and J. L. Hennessy. Computer organization and design: The hardware/software interface, 2013.

[45] D. A. Patterson and J. L. Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[46] J. L. Peterson and A. Silberschatz. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985.

[47] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[48] B. C. Pierce. *Advanced topics in types and programming languages*. MIT press, 2004.

[49] W. Richard. Unix network programming, volume 2: Interprocess communications, 1999.

[50] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[51] L. Robert. *Linux kernel development*. Pearson Education India, 2018.

[52] A. Rubini and J. Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[53] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[54] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 5.

[55] M. Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

[56] W. R. Stevens, B. Fenner, and A. M. Rudoff. *UNIX Network Programming Volume 1*. SMIT-SMU, 2018.

[57] W. R. Stevens, S. A. Rago, and D. M. Ritchie. *Advanced programming in the UNIX environment*, volume 4. Addison-Wesley New York., 1992.

[58] W. R. Stevens and G. R. Wright. *TCP/IP Illustrated: volume 2*. Addison-wesley, 1996.

[59] W. R. Stevens and G. R. Wright. *TCP for transactions, HTTP, NNTP, and the UNIX domain protocols*. Addison-Wesley, 2000.

[60] B. Stroustrup. The c++ programming language (hardcover), 2013.

[61] A. B. Tarımcı. Core java® volume ii: advanced features by cay s. horstmann and gary cornell. *ACM SIGSOFT Software Engineering Notes*, 39(3):24–25, 2014.

[62] R.-G. Urma, M. Fusco, and A. Mycroft. *Java 8 in action*. Manning publications, 2014.

A Key Programming Paradigms

(Tip: refer to *Wikipedia* for more resources.)

A.1 Serial

A.1.1 Non-structured

E.g., machine languages, assembly languages, early high-level languages with *goto*.

A.1.2 Structured

1. Imperative (Turing languages)
   (a) Procedural
   (b) Object-oriented
2. Declarative
   (a) Functional (Church languages)
   (Development: untyped → typed)
      i. Impure
      ii. Pure
   (b) Logic (Church languages)
   (c) Databases

Multi-paradigmatic languages are more and more popular.

A.2 Parallel

A.2.1 Data Parallel

1. High level: vectorization
2. Low level: SIMT
3. Synchronization mechanisms: barriers

A.2.2 Task Parallel

1. Concurrency mechanism
   (a) High level: async/await (a.k.a., future monad)
   (b) Low level:
      i. multiprocessing
      ii. multithreading
      iii. coroutines
         A. stackful
         B. stackless
      iv. futures, promises, and others
      v. callbacks
      vi. reactive extensions
2. Synchronization mechanisms
   (a) Shared memory
   (b) Message passing
   (c) Lock-free programming (e.g., using atomic operations)

A.2.3 Distributed Computing

1. Map-Reduce
2. Transforms-Actions (Spark)

B List of Interesting and Useful Libraries and Tools

B.1 Protocol Buffer & gRPC

Protobuf: https://protobuf.dev/
gRPC: https://grpc.io/

B.2 Google Test & Google Mock

https://google.github.io/googletest/

B.3 OpenTelemetry

https://opentelemetry.io/
Some observability tools like DataDog are really nice, but they are not free. Free ones like Jaeger are also good alternatives.

B.4 AWS SDK

B.5 JAX

https://jax.readthedocs.io/en/latest/

B.6 Clang Toolchains (for C/C++ Development)

1. clang-tidy
2. clang-format
3. clangd (works best with VSCode)

B.7 eBPF

B.8 Inline Assembly

1. C/C++: https://gcc.gnu.org/onlinedocs/gcc/extensions-to-the-c-language-family/how-to-use-inline-assembly-language-in-c-code.html
2. Rust: https://doc.rust-lang.org/reference/inline-assembly.html

B.9 Rust Crates

1. Networking:
   (a) HTTP client: reqwest
   (b) HTTP server: axum, actix
   (c) Low-level HTTP libraries: hyper
   (d) gRPC: tonic
   (e) Async I/O: tokio
2. Async: tokio, futures
3. serde
4. Data parallel: rayon