Computer Systems Learning Resources
A Recommended List

Zhenbang You

January 11, 2022

Last Updated: May 27, 2022

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 2-9 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 The version number in this text may be outdated with new publications.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 an Abstruse 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) [18].
  - In addition to classical algorithms, four kinds of algorithms are widely used in computer systems and thus noteworthy:
    - Randomized algorithms
    - Parallel and distributed algorithms
    - Approximation algorithms
    - Online algorithms
- Probability
  - Elements of *stochastic processes*, especially some basic conclusions and corollaries, are helpful.
- Logic, Automata and Complexity

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 [38]/RISC-V Edition [37]/ARM Edition [39])
    - 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 [23]/RISC-V Edition [24]/ARM Edition [22])
    - You may just read Chap 1-5.
    - The original version is based on MIPS.
3. The RISC-V Reader An Open Architecture Atlas [36]
4. Computer Architecture: A Quantitative Approach (6th Edition) [25]
    - 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) [40]
      - 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) [45]
      ii. Understanding the Linux Kernel (3rd Edition) [14]
      iii. Linux Device Drivers (3rd Edition) [46]
      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)
     (b) Data structures and algorithms
   - 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.
   - Linux man pages online: https://man7.org/linux/man-pages/
2. Advanced programming in the UNIX environment (3rd Edition) [51]

4 Computer Networks

1. Books
   (a) Computer Networking: A Top Down Approach (8th Edition) [28]
   (b) Supplementary (by W. Richard Stevens)
   - UNIX Network Programming
     - Volume 1, 3rd Edition: The Sockets Networking API [50]

    - Volume 2, 2nd Edition: Interprocess Communications [43]
  - TCP/IP Illustrated
    - Volume 1: The Protocols (2nd Edition) [19]
    - Volume 2: The Implementation [52]
    - Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols [53]

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

Compilers should be associated with *PL* and *Computer Architecture*.

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 [34]
     - "Whale Book".
   (d) Engineering a Compiler (2nd Edition) [17]

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/

3. Prerequisites:
   - Compulsory:
     (a) TCS (Theoretical Computer Science)
       i. Introduction to the Theory of Computation (3rd Edition) [49]
     (b) Computer Architecture: ILP (Instruction-Level Parallelism), Memory Hierarchy
     (c) Data structures and algorithms
   - Recommended:
     (a) Computer Architecture: DLP (Data-Level Parallelism), TLP (Thread-Level Parallelism)
     (b) Operating Systems: Thread, Context Switch
   - PL
     - **Java**.

6 Programming Languages (PL)

Abbreviations:

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

6.1 Language List

Tips

1. You need not only *compilers*, *libraries* and *IDE*, but also *build tools*, *debuggers*, *test tools*, *linters*, and even *sanitizers*, *profilers*, *formatters*, etc. Development toolchains matter.
2. Frameworks are key components of the ecosystem of a PL. Success of some PL highly relies on the its frameworks.
3. Sometimes searching within a tutorial/documentation returns better result than searching *Google*.
4. Unless specified explicitly, always try the latest version.
5. Only IDEs I have used will be listed here, so there may be other wonderful IDEs. Generally speaking, JetBrains and VSCode are always good choices (except that the VSCode support for JVM is awful).

- **C/C++** (Do learn the latest version of C++, or at least C++17)
  - Tutorials
    1. The C Programming Language (2nd Edition) [44]
       * "**K & R**"
    2. The C++ Programming Language (4th Edition) [54]
  - 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) [31]
      · Effective Modern C++ (1st Edition) [32]
      · Effective STL (1st Edition) [30]
    * 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.
    * **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/
  - **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.

- **Java** (Java 17 is recommended, or at least, choose an LTS version)
  Prerequisite languages: C++.
  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 [21]
            * Volume II: Advanced Features [55]
        3. Java 8 in Action: Lambdas, Streams, and Functional-style Programming (1st Edition) [56]
    - Documentations
        1. https://docs.oracle.com/javase/specs/
    - Programming guidelines
        * Effective Java (3rd Edition) [13]
    - 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

- **Scala** (*Scala 3* is here!)
  Prerequisite languages: Java.
  Noteworthy features: FP.
  Multi-paradigmatic languages like *Scala* and *OCaml* is 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*
    - Currently, the ecosystem of *Scala 2* is better than that of *Scala 3*. The development *Scala 3* is still at the early stage, but it is also recommended, especially in the long run.

- **Kotlin** (*Kotlin/JVM* is recommended, with *Kotlin/Native* and *Kotlin/JS* being immature)
  Prerequisite languages: Java, Scala.
  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* is here!)
  Prerequisite languages: C++, Java.
  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 database: https://go.dev/doc/tutorial/
    4. 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
  - Documentations
    1. All the documentations can be found at: https://go.dev/doc/
  - Recommended IDE: GoLand
  - Build tool: https://go.dev/doc/modules/gomod-ref

- **Rust**
  Prerequisite languages: C++17, Go, a functional language (e.g., Scala, Haskell, OCaml).
  Noteworthy features: ownership, borrow checker, various **safety**, modest OOP.
  Do compare Rust with other languages.
  Do reflect on *C++* when learning Rust.
  - 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/
  - Recommended IDE: IntelliJ Idea
  - Build Tool: Cargo.
    * Its tutorial can be found in "The Rust Programming Language".

- **OCaml**
  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 return value of recursive functions.
3. Arguments are passed by *reference* in *Scala*, and by *value* in *OCaml*.

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

- Tutorials:
  1. Real World OCaml: https://dev.realworldocaml.org/
  2. The official tutorial and documentation: https://ocaml.org/docs

- **Haskell**
  Purely functional language.

  Recommended learning order: *Scala* → *OCaml* → *Haskell*.

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

  - Tutorials
    1. Real World Haskell: http://book.realworldhaskell.org/read/
    2. Learn You a Haskell for Great Good!: http://learnyouahaskell.com/chapters
  - All the books, courses, tutorials, documentations and various kinds of resources can be found at: https://www.haskell.org/documentation/

- **JavaScript/TypeScript**
  The improvement of *TypeScript* over *JavaScript* is well worth studying.
  Pay attention to its *asynchronous programming* techniques:

  1. callbacks,
  2. promises (async), await.

  - Tutorials
    * JavaScript tutorial: https://www.w3schools.com/js/
    * TypeScript tutorial: https://www.w3schools.com/typescript/
    * Node.js official turorial: https://nodejs.dev/learn
  - Recommended IDE: VSCode

6.2 Principles of Programming Languages

- Courses
  - Stanford CS 242 Programming Languages:
    * Autumn: https://stanford-cs242.github.io/f19/ (preferred)
    * Winter: https://web.stanford.edu/class/cs242/materials.html (supplementary)
  - Stanford CS 151 Logic Programming:
    http://logicprogramming.stanford.edu/stanford/lessons.php

- Books
  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.
  2. Foundations for Programming Languages [33]
  3. Types and Programming Languages (1st Edition) [41]

4. Advanced Topics in Types and Programming Languages (1st Edition) [42]

- Prerequisites:
  - Compulsory:
    1. Computer Architecture
    2. Compilers
    3. One OOP language (*Java* is OK, but *C++/Python* is not enough), one FP language. Some basic knowledge is sufficient.
  - Recommended:
    1. 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: DLP, GPU)
  - CUDA by Example (1st Edition) [47]
    - 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/
  - 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!
- **CPU intrinsics** (Category: DLP, 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: TLP, 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: TLP, 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**.

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: C++, Java, pthread
    2. Future/Promise: Scala
    3. Stackful Coroutine: Go
    4. Stackless Coroutine: Kotlin, Rust (async/await)

    Aside:
    1. In essence, every kind of *Concurrency* is an encapsulation of *Asynchrony.* Therefore, *Concurrent Programming* can be seen as a subset of *Asynchronous Programming.*
       * 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, as well as CUDA.
5. Read Write Lock: most languages. This is especially useful in DLP.

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

∗ 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

7.3 Principles of Parallel Computing

- Stanford CS 149 Parallel Computing: https://gfxcourses.stanford.edu/cs149/fall21/
- Stanford CME 323 Distributed Algorithms and Optimization: https://stanford.edu/ rezab/classes/cme323/S20/

Prerequisites

- Compulsory:
  1. Computer Architecture
  2. Compilers
- Recommended:
  1. Operating Systems

8 Database (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) Database System Concepts (7th Edition) [48]
      - 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. Database Systems: The Complete Book [20]

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 Database 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 Database Systems: https://15721.courses.cs.cmu.edu/spring2020/schedule.html

3. Query languages and platforms

- SQL
  - PostgreSQL
    (a) Database 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) The syntax of *MySQL* is wierd sometimes.
  - 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/

- Spark
  - Quick Start: https://spark.apache.org/docs/latest/quick-start.html
- DB-Engines Ranking: https://db-engines.com/en/ranking
  - Popular database 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.

9 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) Database

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

10 Cloud Computing

1. CMU 15-719 Advanced Cloud Computing:
   https://www.cs.cmu.edu/ 15719/old/spring2019/syllabus.html
2. Prerequisites:
   - Compulsory:
     (a) Computer Architecture
     (b) Operating Systems
     (c) Computer Networks
     (d) Distributed Systems
     (e) Database
   - Recommended:
     (a) Parallel Computing
3. **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.

11 Big Data

1. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems (1st Edition) [27]
2. Prerequisites:
   - Compulsory
     (a) Database
     (b) Distributed systems

12 Software Engineering

- The Mythical Man-Month: Essays on Software Engineering (Anniversary Edition) [15]
- A Philosophy of Software Design (2nd Edition) [35]
- Write Great Code, Volume 3: Engineering Software [26]
- Materials discussing *best practice* are also valuable, which can be found at the **PL** section, as well as the **CUDA** subsection.

13 Program Analysis

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

14 Computer Security

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.

1. Stanford CS 155 Computer and Network Security: https://cs155.stanford.edu/
2. Stanford CS 253 Web Security: https://web.stanford.edu/class/cs253/
3. Stanford CS 255 Introduction to Cryptography: https://crypto.stanford.edu/ dabo/cs255/
4. Stanford CS 350 Secure Compilation:
   https://theory.stanford.edu/ mp/mp/CS350-2019.html
5. Stanford CS 355 Advanced Topics in Cryptography:
   https://crypto.stanford.edu/cs355/22sp/
6. Stanford CS 356 Topics in Computer and Network Security: https://cs356.stanford.edu/
7. Meltdown and Spectre: https://meltdownattack.com/
8. Prerequisites:
    - Compulsory:
        (a) Computer architecture
        (b) Operating systems
        (c) Computer networks
        (d) Programming languages
    - Recommended:
        (a) Compilers
        (b) Database

      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.

15 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) Database

16 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/

17 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) [29]
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.

18 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/

19 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!

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[55] 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.

[56] 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) Database

Multi-paradigmatic languages are more and more popular.

A.2 Parallel

A.2.1 Data-Level Parallelism

1. High level: vectorization
2. Low level: SIMT

A.2.2 Thread-Level Parallelism

1. Concurrency mechanism
   (a) High level: async/await
   (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 mechanism
   (a) Shared memory
   (b) Message passing

A.2.3 Distributed Computing

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