Topics

Systems are complex, and when you’re designing a system you’re grappling with its full complexity. Given this, there are many topics you should be familiar with, such as:

Concurrency. Do you understand threads, deadlock, and starvation? Do you know how to parallelize algorithms? Do you understand consistency and coherence?

Networking. Do you roughly understand [IPC](https://secure.wikimedia.org/wikipedia/en/wiki/Inter-process_communication) and [TCP/IP](https://secure.wikimedia.org/wikipedia/en/wiki/Internet_Protocol_Suite)? Do you know the difference between throughput and latency, and when each is the relevant factor?

Abstraction. You should understand the systems you’re building upon. Do you know roughly how an OS, file system, and database work? Do you know about the various levels of caching in a modern OS?

Real-World Performance. You should be familiar with the [speed of everything](http://everythingisdata.wordpress.com/2009/10/17/numbers-everyone-should-know/) your computer can do, including the relative performance of RAM, disk, SSD and your network.

Estimation. Estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few prototypes or micro-benchmarks to write.

Availability and Reliability. Are you thinking about how things can fail, especially in a [distributed environment](https://secure.wikimedia.org/wikipedia/en/wiki/Fallacies_of_Distributed_Computing)? Do know how to design a system to cope with network failures? Do you understand durability?

Remember, we’re not looking for mastery of all these topics. We’re looking for *familiarity*. We just want to make sure you have a good lay of the land, so you know which questions to ask and when to consult an expert.

How to Prepare

How do you get better at something? If your answer isn’t along the lines of “practice” or “hard work,” then I have a bridge to sell you. Just like you have to write a lot of code to get better at coding and do a lot of drills to get really good at basketball, you’ll need practice to get better at design. Here are some activities that can help:

Do mock design sessions. Grab an empty room and a fellow engineer, and ask them to give you a design problem, preferably related to something they’ve worked on. Don’t think of it as an interview—just try to come up with the best solution you can. Design interviews are similar to actual design sessions, so getting better at one will make you better at the other.

Work on an actual system. Contribute to OSS or build something with a friend. Treat your class projects as more than just academic exercises—actually focus on the architecture and the tradeoffs behind each decision. As with most things, the best way to learn is by doing.

Do back-of-the-envelope calculations for something you’re building and then write micro-benchmarks to verify them. If your micro-benchmarks don’t match your back-of-the-envelope numbers, some part of your mental model will have to give, and you’ll learn something in the process.

Dig into the performance characteristics of an open source system. For example, take a look at [LevelDB](https://code.google.com/p/leveldb/). It’s new and clean and small and well-documented. Read about the implementation to understand how it stores its data on disk and how it compacts the data into levels. Ask yourself questions about tradeoffs: which kinds of data and sizes are optimal, and which degrade read/write performance? *(Hint: think about random vs. sequential writes.)*

Learn how databases and operating systems work under the hood. These technologies are not only tools in your belt, but also a great source of design inspiration. If you can think like a DB or an OS and understand how each solves the problems it was designed to solve, you’ll be able to apply that mindset to other systems.