Notes on

CSE-130: Principles of Computer Systems Design

Mann Malviya
mannmalviya15@gmail.com

These notes cover the principles of computer systems design. The primary sources used to write these notes were:

- CSE 130: Principles of Computer Systems Design — UCSC, Winter 2025 taught by Prof. Kerry Veenstra
- Textbook: *Principles of Computer System Design Part I* by Jerome Saltzer and Franz Kaashoek

Please report any errors to mannmalviya15@gmail.com

Contents

Introduction **3**
- Textbooks . . . 3
- What is this Doc? . . . 3

Lecture 1 **4**
- What does class cover? . . . 4
- Topics Covered . . . 4
- Programming Assignments . . . 4
- Grade Breakdown: . . . 4
- On Using ChatGPT in this class . . . 4

Lecture 2 **6**
- What is a System . . . 6
- Complexity . . . 6
- Four Issues of Complexity . . . 6
- Signs of Complexity . . . 7
- Sources of System Complexity . . . 8
- What Increases Complexity? . . . 9
- Managing Complexity . . . 9

Lecture 3 **13**

Lecture 4 **14**
- Review of C Strings/Variables . . . 14
- UNIX File I/O . . . 14

Lecture 5 **15**

Lecture 6 **16**

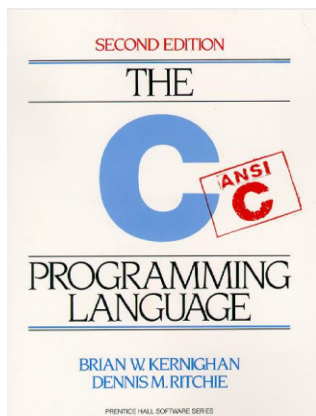
Lecture 20 **17**

Complexity Cheat Sheet **21**

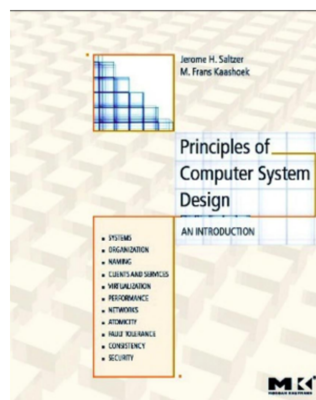
Introduction

Textbooks

1. The C Programming Language by by Brian W. Kernighan, Dennis M. Ritchie
2. Principles of Computer Systems Design Part:I by Jerome Saltzer and Franz Kaashoek (S&K)

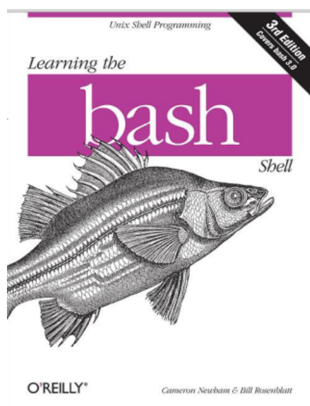


The C Programming Language
by Kernighan & Ritchie

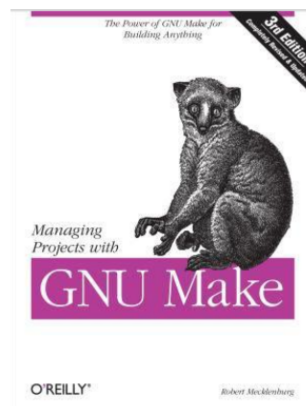


Principles of Computer Systems Design Part:I
by Saltzer and Kaashoek

3. Learning the bash Shell
4. Managing Projects with GNU Make



Learning the Bash Shell
by Newham & Rosenblatt



Managing Projects with GNU Make
by Mecklenburg

What is this Doc?

This doc will contain my "notes" from each lecture of the class CSE130 at UCSC, taught by Prof. Kerry Veenstra. I shall divide this doc based on each lecture, and under each lecture write the important concepts/ideas talked about that day and other stuff I think is interesting. I may add some additional information/explanations from other sources. This is meant to be a comprehensive study doc for this class.

Lecture 1:

What does class cover?

1. **Some Basic System Design Concepts** (Covered in S&K)
2. **Writing Multitasking C Programs**

Topics Covered

- Unix File I/O
- "Fundamental Abstractions"
- Client-Server Model
- Virtualization (also how the kernel works to virtualize: fork, exec, wait)
- Synchronization
- performance
- Security
- Using pthreads
  - → Creating/joining threads
  - → Mutexes
  - → Condition Variables
  - → Synchronization
  - → Locks

Some of the above topics are from S&K, the others are covered to enable you to do the assignments.

Programming Assignments

- **asgn0** — Practice submitting assignments
- **asgn1** — Command-line Memory
- **asgn2** — HTTP Server
- **asgn3** — Multi-Producer/Multi-Consumer Shared-Memory Queue (with updated features)
- **asgn4** — Multi-Threaded HTTP Server (with updated features)
- **asgn5** — Caching

Grade Breakdown:

- Five In-person Quizzes (**30%**)
- Six Programming Assignments (**50%**)
- Final Exam (**20%**)

On Using ChatGPT in this class

- Prof. Veenstra asked his friends/peers in industry what it is that they do, when it comes to using generative AI. He got 2 kinds of answers.
- The first answer was that the company doesn't want them putting confidential information into prompts, but they have internal systems that are trained outside of OpenAI etc. So they can use an internal generative AI system that wont leak company secrets out into other peoples prompts.
- The other answer was,
  1. Read & understand assignment PDF.
  2. Understand the **parts** of the assignment.
  3. Ask ChatGPT to show **3 different ways** to do one of the parts.
  4. Read, understand, and write that part based on your understanding.

5. If needed, repeat this process for the other parts of the assignment.

- Your default role in our society is to be a consumer. A consumer of goods, services, entertainment(where you are the product). But you can decide to perform a different role than the one given to you by default, you can choose to be a Creator, Inventor or Researcher.
- If you give ChatGPT your Assignment PDF and then try to fix the mistakes it made in the generated code you have essentially **inverted the relationship**, you are giving ChatGPT the **creative** role and you are working as its **assistant**. You, as a human being, should perform the creative role.

Lecture 2

Required Pre-Lecture Reading: S&K Chapter 1: Systems.

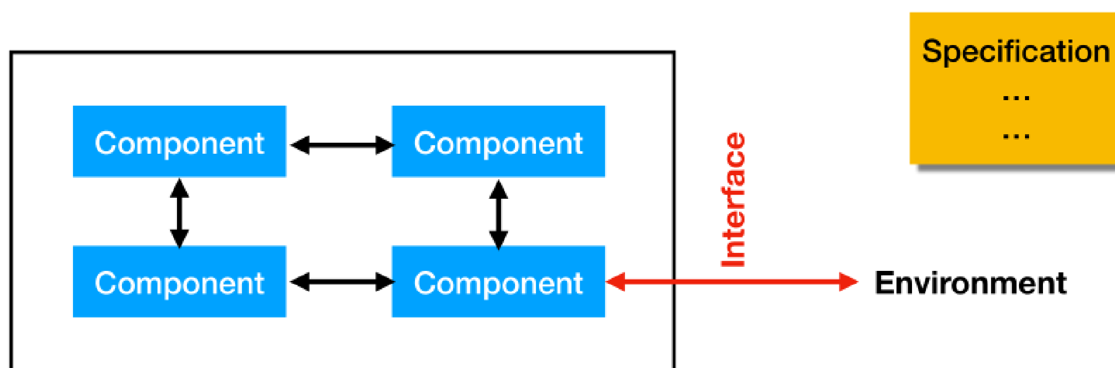
For this lecture, my notes follow the general structure in the lecture slides, but at instances I elaborate a little more by taking material from the pre-lecture reading.

Modern computer systems are large & **complex**.
This lecture covers:

- What problems arise in designing **complex** computer systems?
- How can we manage the **complexity** and the resulting problems?

What is a System:

- "A set of **interconnected components** that has an expected behavior observed at the **interface** with its **environment**."
- The underlying idea of the concept of system is to divide all the things in the world into two groups: those under discussion and those not under discussion.



System and Environment

Complexity

- Four Issues of Complexity
- Signs of Complexity
- What are sources of System Complexity?
- What increases Complexity?
- How can we manage Complexity?

Four Issues of Complexity

1. **Emergent Properties**
   - Emergent Properties are properties that are not evident in the individual components of a system but show up when combined.
   - There can always be a argument about whether or not careful enough prior analysis of the components might have allowed prediction of the emergent behavior. It is wise to avoid this argument and instead focus on the fact that, some things turn up only when a system is built.

     Some Examples:

  - The behavior of a committee(e.g. jury) can develop a way of thinking that could not have been predicted from knowledge about the individuals.
  - The Tacoma Narrows Bridge, due to a combination of closed trusses and wind vortexes gave rise to torsional oscillation(emergent property) which led to the bridge collapsing.
  - The Millennium Bridge became wobbly because of the way pedestrians synchronize their footsteps when the bridge sways, causing it to sway even more.

2. **Propagation of effects**
   - Problems in one component of a system affects other components. The relationship between the components isn't always obvious.
   - "There are no small changes in a large system."

     Some Examples:
     - A tree falling on a power line in Oregon leads to the lights going out in New Mexico, 1000 miles away.
     - Santa Cruz 911 failure of 2009. Vandals cut fiber-optic communication cables in Santa Clara county, this causes Santa Cruz 911 to fail.
     - Northeast Blackout of 1965, on a very cold winter evening, there was high power demand which caused a mis-programmed protection relay to trip under acceptable loads which caused a cascade of line overloads.

3. **Incommensurate(not proportional) Scaling**
   - Not all parts of a system scale at the same rate.
   - The mathematical description of this problem is that different parts of the system exhibit different orders of growth.

     Some Examples:
     - A mouse scaled up to the size of an elephant will collapse under its own weight, because the weight grows with volume(which is proportional to cube of linear size) while bone strength is proportional to its cross-sectional area(which is proportional to square of linear size).

4. **Tradeoffs**
   - What will I sacrifice for more of something else?
   - The general model of a trade-off begins with the observation that there is a limited amount of some form of goodness in the universe, & the design challenge is first to maximize that goodness, second to avoid wasting it, & third to allocate it to the places where it will help the most.

     Some Examples:
     - Bigger smartphones have bigger displays(positive) but use more battery(negative) and don't fit in pocket(negative).
     - In context of integrated circuits, smaller transistor dimensions causes it to be faster(positive), but leads to more power consumption(negative) and more heating(negative).
     - With Binary Classification, we often use a *proxy* to classify or detect something. As a consequence of using a proxy there will be classification errors, "where should the threshold be?". When using a Smoke detector for detecting fire, we use smoke as the proxy and make a detector for that, but we don't wan't alarm to go off for normal cooking but also don't want ti to miss a real fire. Similar argument for a Spam email filter.

Signs of Complexity

- Complex means "difficult to understand". Lack of systematic understanding is the underlying feature of complexity.
- Complexity is both a subjective and a relative concept, there is no unified measure of complexity. We instead look for signs/symptoms of it and if enough appear, argue that complexity is present.

1. **Large number of components**
2. **Large number of interconnections**
   - Even a few components may be interconnected in an unmanageably large number of ways.

     Some Examples:
     - Sun and the known planets comprise only a few components, but every one has gravitational attraction for every other, which leads to a set of equations that are unsolvable.
     - (counterexample) Certain Crystals are not Complex, even though they have many components and interconnections because they have regular/symmetric structure.
3. **Lots of irregularities(little differences)**
   - By themselves, a large number of components and interconnections may still represent a simple system, if the components are repetitive and the interconnections are regular. However, a lack of regularity, as shown by the number of exceptions or by non-repetitive interconnection arrangements, strongly suggests complexity.
   - Exceptions complicate understanding.
4. **Long description**
   - "Kolmogorov complexity" of a computational object is the length of its shortest specification. The longer the Kolmogorov complexity the more complex the system.
   - On the other hand, lack of a methodical description may also indicate that the system is complex.
5. **Large team of designers, implementers, or maintainers**
   - Several people are required to understand, construct, or maintain the system. A fundamental issue in any system is whether or not it is simple enough for a single person to understand all of it. If not, it is a complex system.
   - This is the weakest indicator of complexity.

One objection to conceiving complexity as being based on the five signs is that all systems are indefinitely, perhaps infinitely, complex because the deeper one digs the more signs of complexity turn up. Thus, even the simplest digital computer is made of gates, which are made with transistors, which are made of silicon, which is composed of protons, neutrons, and electrons, which are composed of quarks, which some physicists suggest are describable as vibrating strings, and so on. We shall address this objection in a moment by **limiting the depth of digging**, a technique known as **abstraction**. The complexity that we are interested in and worried about is the complexity that remains despite the use of abstraction.

Sources of System Complexity

1. **Interaction of Requirements**
   - A primary source of complexity is just the list of requirements for a system.
   - Each requirement, viewed by itself may even appear to add only easily tolerable complexity to an existing list of requirements.
   - The problem is that the accumulation of many requirements adds not only their individual complexities but also complexities from their interactions.

     > Design Principle:
     >
     > **Principle of Escalating Complexity**
     > Adding a requirement increases complexity out of proportion.

   - Interaction complexity arises from pressure for generality and exceptions that add complications, it's made worse by change in individual requirements over time.

- Meeting many requirements with a single design is expressed as a need for generality. There's a tension between exceptions and generality.

> Design Principle:
>
> **Avoid Excessive Generality**
> If it's good for everything, it's good for nothing.

2. **Increasing efficiency, utilization, or other measure of "goodness"**
   - The "low-hanging fruit" is easy to get, i.e., if there is an optimization that is obvious and simple to implement, it is already(or will soon be) done, but the subsequent optimizations will be harder and harder to make.
   - Additional gains in efficiency require more complexity. As a general rule, the more one tries to increase utilization of a limited resource, the greater the complexity.

> Design Principle:
>
> **The law of diminishing returns**
> The more one improves some measure of goodness, the more effort the next improvement will require.

What Increases Complexity?

(Essentially repeating each of the design principles mentioned.)

1. **Principle of Escalating Complexity**
   - Adding a requirement increases complexity out of proportion.

     Some Examples:
     - If you want to make your microprocessor go faster and you have already done all the circuit design tricks that you can think of, then the only way to increase speed is to do things like branch prediction, super-scalar, register renaming etc. all these make the system even more complicated.
2. **Principle of Excessive Generality**
   - If it's good for everything, it's good for nothing.

     Some Examples:
     - Amphicar(car+boat), ended up being a subpar boat and a subpar car. Same problem with "flying cars".
3. **Law of diminishing returns**
   - The more one measure of goodness is improved, the harder it gets to make the next improvement.

     Some Examples:
     - If you want a faster computer system, you add cache memory, if you want it to be faster still, you can go onto adding a bucket list of every more complicated optimizations like branch prediction, super-scalar etc, but the return on investment(in terms of speed) keeps getting lesser and lesser.

Managing Complexity

1. **Modularity**
   - Modularity is the technique of breaking things into smaller pieces(aka divide-and-conquer), it's essentially analyzing or designing the system as a collection of interacting subsystems, called modules.

- The power of this technique lies in being able to consider interactions among the components within a module without simultaneously thinking about the components that are inside other modules.
- Modularity allows development of a smaller unit, that reduces the number of components and interactions, that reduces complexity and reduces the number of bugs.
- Modularity makes it easy to replace an inferior module with an improved one, thus allowing incremental improvement of a system without completely rebuilding it. Modularity thus helps control the complexity caused by change.

> Design Principle:
>
> **The unyielding foundations rule**
> It is easier to change a module than to change the modularity.

Some Examples:

- Consider debugging a large program with `N` statements. Assume number of bugs in the program is proportional to its size and the bugs are randomly distributed.

$$\text{BugCount} \sim \mathtt{N}$$

Assume also that the time taken to find a bug in a program is roughly proportional to the size of the program.

$$\begin{aligned}\text{DebugTime} &\sim \mathtt{N} \times \text{BugCount} \\ &\sim \mathtt{N}^2\end{aligned}$$

Suppose the program is divided into `K` modules, each roughly equal size, so now the modules contain `N/K` statements.
The time required to debug any one module is thus reduced in two ways: the smaller module can be debugged faster, & since there are fewer bugs in smaller programs, any one module will not be needed to be debugged as many times. Thus the time required to debug the system of `K` modules becomes,

$$\begin{aligned}\text{DebugTime} &\sim \left(\frac{\mathtt{N}}{\mathtt{K}}\right)^2 \times \mathtt{K} \\ &\sim \frac{\mathtt{N}^2}{\mathtt{K}}\end{aligned}$$

Therefore we can see that modularization into `K` components reduces debugging time by a factor of `K`.

2. **Abstraction**

- When using modularity to subdivide a complex system, it should be noted that the best divisions usually follow natural or effective boundaries. They are characterized by fewer interactions among modules & cause less propagation of effects from one module to another.
- More generally, they are characterized by the ability of any module to treat all the others entirely on the basis of their external specification, without need for knowledge about what goes on inside.
- This additional requirement on modularity is called **abstraction**. Abstraction is separation of interface from internals, or specification from implementations. Essentially treating the individual components of a system as a **black box**. The black box has inputs & outputs & behaviors. You are Supposed to trust that the black box will do what it promises.
- Abstraction nearly always accompanies modularity. The term *functional modularity* is used to mean modularity with abstraction

> Design Principle:
>
> **The robustness principle**
> Be tolerant of inputs and strict on outputs.

- The effect of the robustness principle is to suppress, rather than propagate or even amplify, noise, or errors that show up in the interfaces between modules.

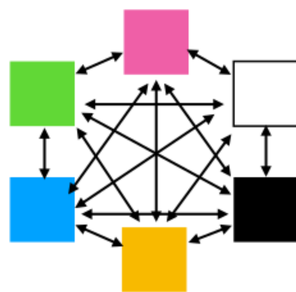
  Some Examples:

  – Programs are designed to hide details of their representation of complex data structures and details of which other programs they call. Users expect an easy-to-use user interface that abstracts incredibly complex underpinnings pf memory, processor, communication, and display management.

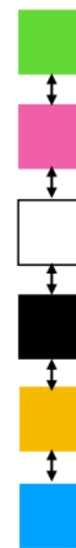
  – A market economy is characterized by modularity. Rather than having a self-supporting farm family that does everything for itself, a market economy has coopers, tinkers, blacksmiths, stables, dressmakers, and so on, each being more productive in a modular specialty, all selling things to one another using a universal interface—money.

3. **Layering**

- Systems that are designed using good abstractions tend to minimize the number of interconnections among their component modules.
- One powerful way to reduce module interconnections is to employ a particular method of module organization known as **layering**. In designing with layers, one builds on a set of mechanisms that is already complete(a lower layer) and uses them to create a different complete set of mechanisms(an upper layer).
- A layer may itself be implemented as several modules, but as a general rule, a module of a given layer interacts only with its peers in the same layer and with the modules of the next higher and next lower layers.



No Layering

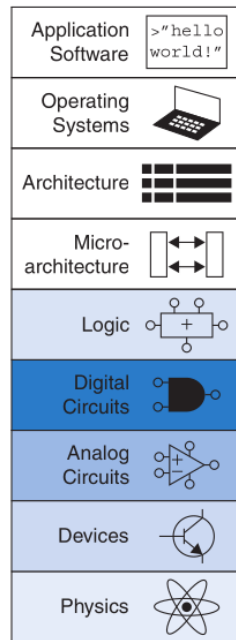


Layering

Fundamental theorem of software engineering (David Wheeler):
"Any problem in computer science can be solved with another layer of indirection, except for the problem of too many layers of indirection."

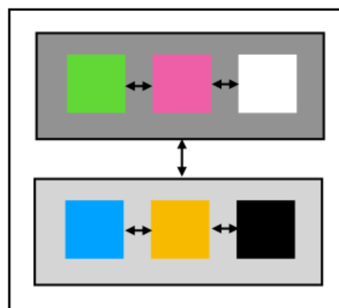
Some Examples:

– Nearly every computer system comprises several layers. The lowest layer consists of transistors, gates and memory cells, upon which is built a layer consisting of a processor and memory. On top of this layer is built an OS layer, which acts as an augmentation of the processor and memory layer. Finally an application program executes on this augmented processor and memory layer.



Levels of abstraction for an electronic computing system

4. **Hierarchy**
   - The final major technique for coping with complexity also reduces interconnections among modules.
   - Start with a small group of modules, and assemble them into a stable, self-contained subsystem that has a well-defined interface. Next assemble a small group of subsystems to produce a larger subsystem. This process continues until the final system has been constructed from a small number of relatively large subsystems.
   - The result is a tree-like structure known as a **hierarchy**.
   - Hierarchy constrains the interactions by permitting them only among the components of a subsystem. Hierarchy constrains a system of N components, which in the worst case might exhibit $N \times (N - 1)$ interactions, so that each component can interact only with members of its own subsystem, except for an interface component that also interacts with other members of the subsystem at the next higher/lower level of hierarchy.
   - Analogous to the way that modularity reduces the effort of debugging, Hierarchy reduces the number of potential interactions among modules from square-law to linear.



System broken into subsystems broken into modules

5. **Naming (We don't cover this in CSE-130)** FILL THIS UP SOME TIME

Lecture 3

- Prof. Veenstra worked through Prof. Quinn's Practicum for setting up your system for CSE 130, this included,
  - Installing & setting up Multipass VM.
  - Setting up the respective repositories(there are 2, named <your cruz-id> and resources).
  - Setting up and using the auto-grader.
  - Using the resources repo for loading local test cases.
- Had Quiz 1—On the material from Lecture 2(Complexity).

Lecture 4

Review of C Strings/Variables

UNIX File I/O

Lecture 5

Lecture 6

Lecture 20

Refresher: SI Prefixes

Prefix	Power(base 2)	Power(base 10)
Kilo(k)	2^{10}	10^3
Mega(M)	2^{20}	10^6
Giga(G)	2^{30}	10^9
Milli(m)	2^{-10}	10^{-3}
Micro(μ)	2^{-20}	10^{-6}
Nano(n)	2^{-30}	10^{-9}

Commit these to memory

Computer Systems performance

- We want computers to perform "Better":
  - Increase Throughput.
  - Decrease Latency.
  - Get more things done per unit of energy.

How is performance measured?

Capacity Metrics:

- **Capacity:** Amount of a resource thats available.
  - Eg: This machine has a **8Gb** RAM.
- **Utilization:** Percentage of Capacity being used for a given workload.
- **Overhead:** Resource that is "wasted".
  - Eg: In a layered system, layers below are Overhead.
- **Useful Work:** Resource spent on actual work for a given workload.
  - In a layered system, layers above perform useful work.

Time-based metrics:

- **Latency:** Elapsed time for a **single** action.
- **Throughput:** **Rate** of actions per unit time(also called "bandwidth").

System Latency and Throughput

- Latency and Throughput are 2 different things but there can be a relationship between them.
- If you have a single module(1 stage), the time to get the data from the input of the module to the output(the latency), assuming its not pipelined is related to the Throughput as:

$$\text{Latency} = \frac{1}{\text{Throughput}}$$

Latency = 1 / Throughput

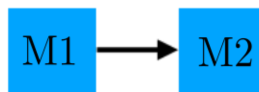
Blue Box represents a single module

In this case the module cannot process more than one thing at a time.

- Instead if we did pipelining and we have some concurrency, then the total latency and Throughput of the entire pipelined system is:

$$\text{Total Latency} = \text{sum(module Latencies)}$$

$$\text{Total Throughput} = \text{min(module Throughputs)}$$

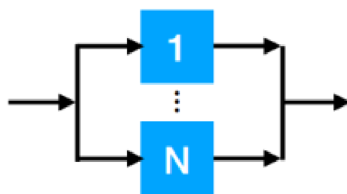


Concurrency (Pipelining)

Total Latency = sum(module latencies)
Total Throughput = min(module throughputs)

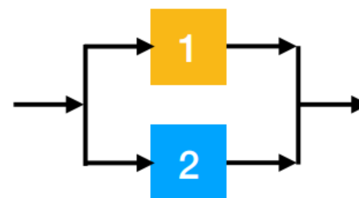
Blue Box represents a single module

- Another way to do concurrency is to have modules running parallelly. The two cases here are if all the modules are identical or if all the modules are different. In the second more general case the average latency of the entire system will be a weighted average of the Latencies of its component modules. In the second case the throughput depends on how the requests get distributed i.e. you need to know what fraction/percentage of the incoming data is going through one of the modules and what percentage is going through the other module.



Concurrency:
If N Identical Modules:

System Throughput = module Throughput × N
System Latency = module latency



Concurrency (Different Modules)

f1 = fraction of requests served by module 1
f2 = fraction of requests served by module 2
Average Latency = f1 × L1 + f2 × L2

- There can be a situation in which the the rates of the input & output on either side of the module differ, then the minimum of the two throughputs is the fastest throughput the module can have.

What if Input and Output Throughputs Differ?

Example: Data compression/decompression

Need to consider changes in throughput

Techniques for Improving System performance:

1. Pipelining
2. Concurrency
3. Fast Path
4. Limitations

1. Pipelining

- **Pipelining:** Split operation into multiple individual stages.
  - Each stage operates independently.
  - Single request goes through multiple stages.
  - Different requests may go through different stages.
- Latency: The time for a single request to go through all stages.

$$\text{latency}_{A+B} \geq \text{latency}_A + \text{latency}_B$$ [1]

- Throughput: The net throughput of the system is limited by the throughput of the slowest stage.

$$\text{throughput}_{A+B} \leq (\text{throughput}_A + \text{throughput}_B)$$

- Pipelines increase throughput by keeping every stage of the system busy.

2. Concurrency

- Concurrency can **reduce latency**, by running independent parts of stages in parallel.
- Concurrency can **increase throughput**, because each stage works on a different request.

3. Fast Path

- Reducing latency is difficult: often limited by:
  - Speed of light
  - Algorithm run time
- We can Leverage the non-uniformity: reduce latency for some requests, by creating a fast path.
- One very common approach to fast-pash is Caching.
- In caching we have a fast memory(which we call the fastpath) and we have the larger slow memory. The fast memory tends to be faster but only contains a subset of the data, the slow memory is large enough to contain all the data, if you can design your system in a way such that most of your data requests are serviced by the fast small memory, even though it only has a subset of all the data. Then it appears as if you have a large memory as fast as the faster small memory.

[1] $\geq$ because there can be overhead of the transition between the pipelined stages

Digital Circuits

This is a tcolorbox using the color palette inspired by your image.

Complexity Cheat Sheet

Complexity

1. Four Issues of Complexity
2. Signs of Complexity
3. What are Sources of System Complexity?
4. What Increases Complexity?
5. How Can We Manage Complexity?

---

1. Four Issues of Complexity

1. Emergent Properties
2. Propagation of effects
3. Incommensurate Scaling
4. Tradeoffs

2. Signs of Complexity

- Large number of components
- Large number of interconnections
- Lots of irregularities (little differences)
- Long description (high information content)
- Large team of designers (”weakest” Symptom)

3. What are Sources of System Complexity?

- Interactions of requirements
- Increasing efficiency, utilization, or other measure of ”goodness”

4. What Increases Complexity?

- Principle of escalating complexity
- Principle of excessive generality
- Law of diminishing returns

5. How Can We Manage Complexity?

- Modularity
- Abstraction
- Layering
- Hierarchy
- Naming