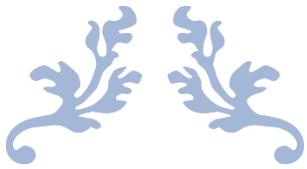
REESHABH CHOUDHARY

CONCURRENCY & PARALLELISM

EFFICIENT RESOURCE UTILIZATION IN PROGRAMMING



CONCURRENCY & PARALLELISM

Efficient Resource Utilization in Programming

REESHABH CHOUDHARY

Concurrency & Parallelism © 2024 by Reeshabh Choudhary , all rights reserved.

या कुन्देन्दुतुषारहारधवला या शुभ्रवस्त्रावृता

या वीणावरदण्डमण्डितकरा या श्वेतपद्मासना।

या ब्रह्माच्युत शंकरप्रभृतिभिर्देवैः सदा वन्दिता

सा मां पातु सरस्वती भगवती निःशेषजाड्यापहा॥

(Salutations to Devi Saraswati) Who is pure white like jasmine, with the coolness of the moon,

that shines like a garland of snow and pearls; And who is covered with pure white robes,

Whose hands are adorned with veenas and boons; And who sits on a pure white lotus,

One who is always the adoration of Bhagwan Brahma, Bhagwan Vishnu, Bhagwan Shankar and other deities,

O Devi Saraswati, please protect me and remove my ignorance completely.

Acknowledgements

I bow down to my deities *Maa Saraswati, Ganesh ji and Hanuman ji* for guiding me to pursue knowledge and giving me strength to deal with challenges on the way.

The journey from my first book Objects, Data & AI to the one you are reading currently has been nothing sort of ordinary. Having completed my first book around July 2023, I was not ready to take up the task of compiling another book for a considerable period of time. In the meantime, me and Richa have been blessed with a baby girl on 5th September 2023. And it was her journey which inspired me to take up writing again.

On the second day after her birth, she was detected with *NEC (Necrotizing enterocolitis)*, which affected her large intestine. NEC is usually detected in pre-matured babies, however, our baby being full term somehow got infected by it. She had to be operated a week later, and required another procedure, which was done successfully in December 2023. By the grace of Supreme Being, and her spirit to survive, she is now completely normal and lights up every moment of our life. The journey of this 4-month duration had been nothing sort of a rollercoaster ride though. Sometimes our own words are enough to describe what we go through, but the humankind has been blessed with writers who can capture the essence of emotion through magic of words. Which lines to select out of many written, perhaps these from Charles Dickens resonate more:

"Oh! the suspense, the fearful, acute suspense, of standing idly by while the life of one we dearly love, is trembling in the balance! Oh! the racking thoughts that crowd upon the mind, and make the heart beat violently, and the breath come thick, by the force of the images they conjure up before it; the desperate anxiety *to be doing something* to relieve the pain, or lessen the danger, which we have no power to alleviate; the sinking of soul and spirit, which the sad remembrance of our helplessness produces; what tortures can equal these; what reflections or endeavours can, in the full tide and fever of the time, allay them! "

- Oliver Twist, Chapter XXXIII

And it was in this period, to keep myself composed and clear, free from distractions and thoughts of uncertain future over which I hold no control whatsoever, I decided to research into the topic of Concurrency and Parallelism.

As they say there is light at the end of the tunnel, darker days indeed passed and sun smiled over us, and in January 2024, I started compiling my learnings in a structured format which is being presented before you.

The immortal words of Fyodor Dostoevsky summarize it better:

"The soul's journey is often fraught with trials and tribulations. Yet, it is precisely through our struggles that we uncover the depths of our character and the breadth of our capacity for love and compassion."

I cannot thank enough my dear friends *Mr. Ashish Agarwal, Mr. Gurpreet Singh, and Mr. Saurabh Singh*, who firmly stood by us throughout the journey. In the process of writing the book, *Mr. Saurabh Singh* kept pushing me to go deeper and shared some invaluable learning resources which were helpful in adding context to certain topics.

As always, I had the blessing of my mentor, *Mr. Rupam Das*, who consciously encouraged me to research into a list of topics, which would later help to enrich the content of the material which you are about to read.

I drew a lot of inspiration from the writings of *Mr. Robert Greene*, as it helped to me have faith on my abilities, chose the path less travelled by and trust my instincts. His writings acted as my inner voice which I intended to listen time and again.

This compilation would have never been possible without the strength of my beloved wife, *Richa*, who trusted on my decisions blindly and made sure I had enough space to work on this project.

And *Maithili*, you have fought against the odds and thrived. May you continue to do so, as you are the legacy which I intend to leave behind. Your smile heals me every single day. *I love you both!*

Reeshabh Choudhary

For Maithili!

Preface

Year 2023 saw the rise of Large Language models (GPT, BARD, etc.) and smart coding assistants (GitHub Copilot). Since then, a fear has been gripping in the mind of software developers that this is the end of programming as we know it. After all, these models can generate hundreds of lines of code on the context of few lines of prompt. But programming is more than just writing lines of code. This is an art, and it requires you to think through the system, and this is the missing link in the artificial intelligence. It can always be trained on past data and based on that it can predict or generate content, however, it cannot be held accountable or be taken granted for writing the most efficient program which suits your need.

Say, I have a task which requires me to compare a document against a set of documents on various parameters. I ask an AI model to write a program for this task. However, this is most likely going to be a sequential program. Now, if you want to utilize available cores of your CPU and complete the task in a shorter span of time, you might think of parallelizing the computations in your program. This requires identifying the pieces of your program which can be parallelized and do not result in synchronization overhead. Once, you are able to do that, you may ask the AI model to modify your program in a certain way. The output generated can be then integrated as per your requirements.

The point which I am trying to make here is that to write a good program, a developer does not just need to have an understanding of programming languages but must understand the capabilities of the system where program is going to run. The AI model can no doubt quickly generate lines of code, but it still requires direction of a human mind.

Machines can be trained to perform certain tasks, but this is not the guarantee that the tasks are being performed in the most efficient manner. It is called *artificial* intelligence for a reason. To think through the systems, adapt and improvise, to be held accountable, human intelligence is required and that should be the case in the future unless we are ruled by humanoids.

Reeshabh Choudhary

What this book is about?

This book is about developing an intuition about what happens underneath a program, so that we can first focus on utilizing the available resource at hand before thinking of scaling the solution. This is the very art of engineering, making use of resources in limited budget and getting things done. Once a developer understands what is going on beneath the surface, the programming languages are nothing but just abstractions via which we talk to our computers. And this is one of the reasons why this book has used minimal programming language. Rather than focussing on programming language, once a software developer starts to think with respect to the device at hand, then the real journey of programming starts.

This book just barely tries to scratch the surface of the vast world of programming and starts from the very basic concepts of Operating Systems and then moving on to application development and in the second part interaction with the databases is covered.

Right Approach to read this book.

I will request the readers of this material to have a research centric approach while reading it. At times, there are lots of concepts which interleave and for the sake of brevity, author has skipped explaining some of them to maintain the flow of the discussion. User is encouraged to leverage online search to look for concepts which require elaborate explanation.

Every reader has the ability to connect the dots in a unique way based upon the understanding and experience developed over the years. And for the same reason, the discussion presented in this book may seem quite open ended. There is no fixed approach to programming however underlying resources like threads or processes remain the same. This is why different programming languages or different frameworks within a programming language coexist.

I have humbly tried to make this point to my precious readers that every decision, whether in real life or technical design, comes with its pros and cons. There is no fixed answer, and we must stop looking for them. What works for one may not work for other use case. We must practice the ability of scrutinizing our choice and be fluid in selecting appropriate approaches to programming, rather than being biased towards an approach or language or framework. As Robert Greene aptly puts it:

"The need for certainty is the greatest disease the mind faces." – Mastery, Robert Greene.

The book tries to present different case studies of frameworks and application development to present the readers with different ideas. Rather than preaching a way of programming, book tries to weigh pros and cons of different concepts and also explores how the concepts intermingles with each other. The onus of implementing these concepts in day-to-day programming lies with the reader.

A book is always about presenting a story to its reader, whether fictional or non-fictional. The material you are reading right now is focussed on presenting the story of evolution of programming and the way forward!

NOTE: I am merely a compiler of the knowledge and information already available in the public domain. I do not intend to take any credit for the work done by amazing programmers over the years. This book is all about appreciating the programmers who think of utilizing the system resources at maximum and deliver an efficient yet robust product.

References

Throughout human history, books have been the biggest source of knowledge. A good book inspires countless other books. A student reads different books on same topic to understand different perspectives. I am no different. Compilation of this book would have been incomplete without numerous books and online source materials present. Although, it would not be possible to mention each and every source, from where I have gathered information, I would definitely like to mention some key books which have served as the axis of this book.

- The Art of Multiprocessor Programming - Maurice Herlihy, Nir Shavit
- Concurrency Control and Recovery in Database Systems - Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman
- The Little Book of Semaphores, The Ins and Outs of Concurrency Control - Allen B. Downey
- Fundamentals of Database Systems - Ramez Elmasri, Shamkant B. Navathe
- Database system concepts - Henry F. Korth, Abraham Silberschatz, S. Sudarshan
- Principles of distributed database systems - Ozsu M.T., Valduriez P
- Programming Massively Parallel Processors, A Hands-on Approach - David B. Kirk, Wen-mei W. Hwu
- The Art of Concurrency, A Thread Monkey's Guide to Writing Parallel Applications - Clay Breshears
- ACTORS, A Model of Concurrent Computation in Distributed Systems - Gul Agha
- Actors in Scala - Philipp Haller, Frank Sommers
- Is Parallel Programming Hard, And, If So, What Can You Do About It? - Paul E. McKenney
- Operating System Concepts - Abraham Silberschatz, Greg Gagne, Peter B. Galvin

Apart from these books, I gathered lot of reading material via help of stackoverflow.com and perplexity.ai . I would humbly like to thank the software engineers who have kept sharing their wisdom over the years via blogs to help fellow engineers across the globe. In my personal opinion, apart from sports, programming community is definitely the one which brings people together across boundaries.

Contents

Reeshabh Choudhary

Part 1: OS & Application Development

Reeshabh Choudhary

Introduction

The word **concurrency** is derived from the Medieval Latin word "*concurrentia*", meaning coming together or simultaneous occurrence, which itself has evolved from the Latin word "*concurrens*", meaning running together. The word concurrency has been used since the last 15th century, and we can assume that the concept of coming together at the same time would have existed much before that.

The word **parallelism** comes from the word 'parallel', which is derived from the Greek word '*parallēlos*' (*para* = alongside, *allēlos* = one another). It has been recorded in use since 16th century.

The point being, we are not new to these terms or their usage.

Imagine a huge crowd at the ticket counter of a newly released movie or customers lining up in the bank branches at the end of the month to withdraw salaries. If a resource is valuable, scarce and in demand, it is meant to face concurrent scenarios. And to serve these demands, a theatre or bank may arrange for operation of parallel counters to increase the efficiency and speed of service.

What we do in the computer world (virtual world), is merely replicate the entities and their functioning in the real world. The idea and inspiration behind the solutions implemented in software systems are always derived from the observations and situations happening in real life.

Imagine a chef working in a famous restaurant and dealing with the surge of customers on a weekend. He would adopt different strategies to deal with situations. A recipe has multiple sub-tasks to be performed, which can be either a result of one another or some steps can be performed parallelly. A chef can delegate parallel steps to other workers in the kitchen and once the task is completed, he would combine the outputs and produce the desired cuisine. Or he can create parallel workstations, where orders are prepared end to end, and he can oversee their preparations. There can be other possible approaches to deal with the situation.

In real life or virtual, there is no one stop solution. Every solution comes with its own after-effects, and it is up to us to adapt what seems the best suited for our situation. The solutions we are using today have evolved over time, after learning different outcomes in different situations.

Dealing with concurrency in the computer world is no different. Three decades back, the resources inside a computer were not compute heavy, had limited compute capability and processing power. In fact, early computers had just one processor, but evolved over time to perform multiple tasks simultaneously. The sector of computer chip design also went through a significant transformation.

In 1965, Gordon Moore, the co-founder of Intel, published a paper in which he observed that the number of transistors on an integrated circuit at minimum cost had increased by a factor of two between 1960 and 1965. Based on this observation, he further predicted that the number of transistors on an integrated circuit will double every two years with minimal rise in cost.

His observation was not scientific but based on intuitive understanding of this field, and over time his observation has stood the test of time. And the microprocessors of Intel Pentium family further cemented his legacy by standing true to his observations. These microprocessors based on a single central processing unit (CPU), drove rapid performance increases and cost reductions in computer applications for more than two decades, which allowed application software to provide more functionality, have better user interfaces, and generate more useful results. However, customer demands are rarely satisfied, and expectation increases at each turn of the wheel. The ever-increasing demand for performance improvement and cutthroat competition in this field, led to a positive cycle for computer industry.

Software developers started relying on the underlying hardware to increase the speed of the application. However, since 2003, due to energy consumption and heat-dissipation issues, processor's clock speed started hitting the upper limit. Vendors started switching towards models where multiple processing units, *processor cores*, are used in each chip to increase the processing power. And this changed the course of programming constructs since then.

Traditionally, software developers approached programming in sequential fashion, as if programs are executed by a human sequentially stepping through the code. However, a sequential program will only run on one of the processor cores, which has practically hit the ceiling in current times. But the demand for

improvement in performance in software application has been constant. Hence, in modern approach to programming, software developers have started to explore the concept of parallel programming, which basically means to increase the processing power of a computer by leveraging CPU cores in parallel.

The emphasis now is on parallelism – rather than using transistors to increase clock speeds, they are being allocated more cores and low-latency memory onto the chip. With parallelism, large computational tasks are divided into smaller tasks that can be executed simultaneously on different cores. This allows for better utilization of available resources and improved overall performance, although with some downsides, which are going to be the topic of discussion throughout the book.

Often the part where computers perform multiple tasks simultaneously creates a blur line of concurrency and parallelism. Hence, we try to address this confusion before starting any serious in-depth discussion.

1.1 Concurrency Vs Parallelism

To understand this subtle difference between concurrent and parallel operations in computer, we must understand their evolution over time. Early age computers had just one processor, hence, to perform multiple operations at the same time, it had to adopt some smart strategies like switching between tasks as they come (***concurrent***) and then resuming the operations from where it has left earlier. However, with time computers started getting multiple processing power (multiple CPUs) and they had access to more resources to handle tasks at hand. Hence, the computers could ***parallelly*** perform sub tasks of a major task through available CPUs or two independent tasks could be performed parallelly.

To be precise, ***Concurrency*** refers to the ability of different tasks to progress in overlapping time periods. In a concurrent system, multiple tasks are initiated, executed, and completed, but they may not necessarily be executed simultaneously. Concurrency is often associated with systems where tasks are interleaved, sharing the same resources or time slices. Multitasking on a single-core processor is an example of concurrency, where the CPU rapidly switches between different tasks, giving the illusion of parallel execution.

On the other hand, ***Parallel execution*** implies that tasks are literally running at the same time. In a parallel system, multiple processors, cores, or execution units work simultaneously to execute different parts of a task. This can lead to a significant increase in processing speed and throughput. Examples of parallelism

include executing multiple threads simultaneously on a multicore processor or distributing computation across multiple machines in a cluster.

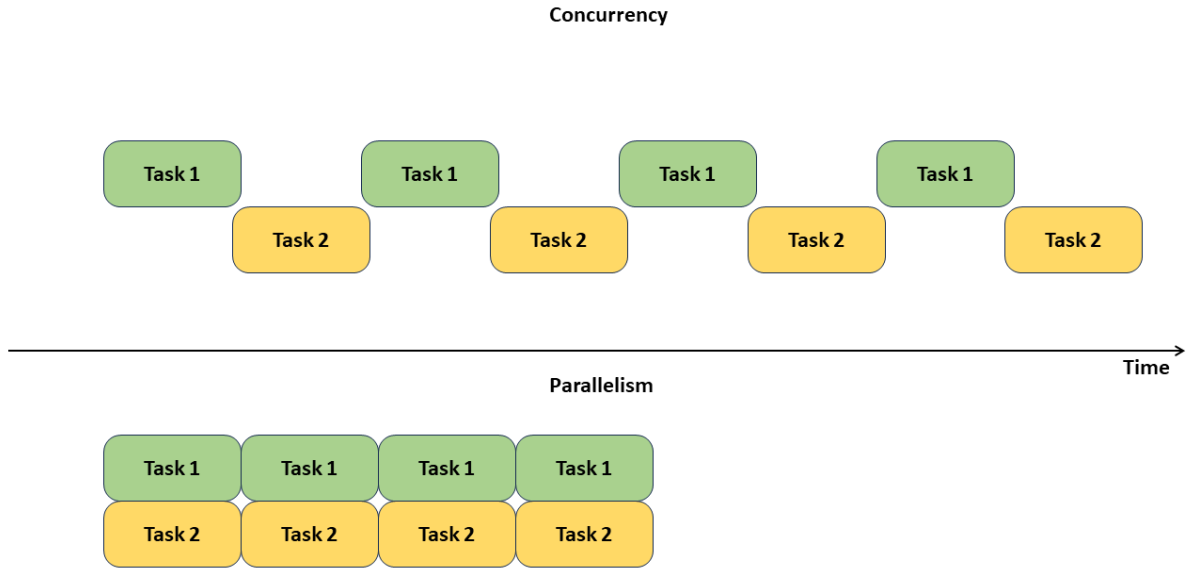


Figure 1-1 Concurrency versus parallelism

Let us go back to our kitchen example. Imagine a restaurant kitchen where different team members are working on various tasks concurrently. One person is toasting bread, another is frying eggs, and yet another is preparing coffee. Each team member is focused on their specific task, and their activities are interleaved. While one person is waiting for the toast, another is actively working on the eggs or coffee. The kitchen team efficiently manages their tasks, and the overall breakfast preparation progresses in overlapping time periods.

With more demand coming to the kitchen, they decided to upgrade the kitchen workstation. The upgraded kitchen setup has multiple cooking stations, each equipped with its own stove, toaster, and coffee maker. In this setup, different team members can work simultaneously on their specific tasks without waiting for shared resources. One person is toasting bread at Station 1, another is frying eggs at Station 2, and a third team member is brewing coffee at Station 3. The tasks are running in parallel, and the overall breakfast preparation is completed more quickly because multiple activities are happening at the same time.

Of course, what we are discussing here is vague and we shall be covering the detailed explanation in the upcoming chapters about how computers leverage resources to produce desired results, but we get an idea of concurrent and

parallel usage of a resource. Often, we approach a solution with resource cost in mind.

Parallelism often involves multiple resources, such as additional processors, cores, or servers. These resources come with associated costs, both in terms of hardware and potentially increased operational expenses. Understanding the resource requirements and costs associated with parallelism is essential to make informed decisions about the project budget and resource allocation.

We try to maximize the utilization of available resources and still if we require more helping hands, we deploy more resources (scaling). Our discussion throughout the book will be mainly focused on efficient utilization of available resources and their coordination in completing an activity over time during concurrent situations. Scalability is a beast on its own and requires a separate discussion. In this book we mostly try to look at challenges from the perspective of a single computer system.

While parallelism can enhance scalability, it needs to be planned thoughtfully. Blindly adding resources without understanding the scalability characteristics of the application can lead to suboptimal performance and wasted resources. Workload demands can vary over time, and concurrency management plays a key role in adapting to these variations.

Parallelism is about leveraging the available computational power of the processing cores, if sitting idle. This is where these two concepts interleave. Concurrency is about managing and utilizing resources efficiently. Understanding how concurrency works at various levels, from operating systems to application frameworks, helps in designing systems that make the most effective use of available hardware. This includes optimizing algorithms, minimizing contention for shared resources, and reducing unnecessary overhead. It allows developers to design systems that can dynamically scale resources based on demand, preventing unnecessary resource allocation during periods of lower demand, and ensuring responsiveness during peak periods. Concurrency presents a more strategic approach to scalability planning and ensures that resources are added in a way that aligns with the actual needs of the application.

Say, we break down a large computational task into independent subtasks which are then being parallelly processed at different cores. But, to process them efficiently, we may end up utilizing the knowledge of concurrency in OS, as the subtask can itself require a process to create multiple threads for processing the

instructions. We discuss about ‘process’ and ‘threads’ in upcoming chapter, but we can draw the kitchen analogy just discussed here.

Although modern programming languages, libraries, and frameworks provide high-level abstractions for concurrent and parallel programming, a deeper understanding of the underlying concepts is crucial for building compute-intensive applications with efficient resource management. The abstraction provided by the programming languages does not expose all the nuances of performance optimization. To achieve optimal performance and resource utilization, it is crucial to understand how concurrent and parallel constructs are implemented at the lower levels of the system. This includes knowledge of how threads are scheduled, how processes share memory, and how system resources like CPU, memory, and I/O are managed. This understanding helps developers write code that makes efficient use of available resources.

It is important to note here that the term ‘concurrency’ in computer programing is not just about how the processes or threads work internally, but also about the concurrent requests an application or database serves at one point of time. Throughout the discussion, this context does get overlapped, so please keep an eye out.

The aim of the discussions presented in this book is to set the ground for maximum resource utilization in a shared memory computer and understand the conflicting scenarios which occur in the process. In the upcoming chapter, we shall be starting our discussion from the basic computer architecture and components and how they interleave to perform tasks, and in subsequent chapters we build upon these concepts.

Reeshabh Choudhary

Beneath the surface: OS

When a user interacts with a computer, he/she does seem to interact with an entity, which itself is an abstraction over multiple components functioning together. Instructions are provided via external devices (hardware) such as keyboard, mouse, etc., in the form of application programs (text editors, spreadsheets, calculator, etc.), which are meant to perform the tasks intended by the user.

From computer's perspective, it is the **Operating System** which abstracts away the functioning of computer hardware underneath and presents to user an interface to collect instruction and get them executed using the resources of computer.

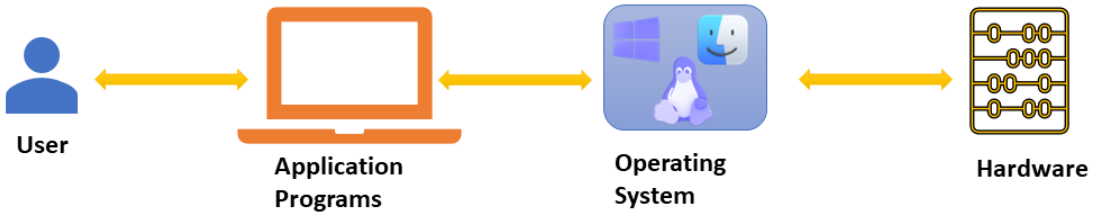


Figure 2-1 User interaction with computer

The hardware of a computer encapsulates the CPU (central processing unit), the memory and the input/output devices, and provides resources for computational ability. **Operating System (OS)** provides the necessary environment for controlling and allocating these resources to perform computational tasks provided by the user. The main aim of computer systems is to execute programs and to help users solve their problems easily. **Operating System** can be viewed as a program (**kernel**) that runs on the computer all the time, running alongside a middleware framework that ease application development and provide features, and system programs that aid in managing the system while it is running.

Computers use the binary system, which is a number system based on two symbols: 0 and 1. All information processed by a computer, including data, instructions, and even the characters we see on the screen, is ultimately represented in binary form. This is the reason why digital electronic circuits, such

as those in a CPU, operate using binary signals (on and off). The **CPU (central processing unit)** is the brain behind the operations. CPUs are made up of several parts, including the control unit (CU), the arithmetic logic unit (ALU), and the registers, which work together to execute instructions and perform tasks.

At the foundational level, CPUs are built from basic components like transistors that act as switches, toggling between on and off states based on the presence or absence of an input signal. These switches are abstracted into logic gates, such as AND, OR, and NOT gates, which form the building blocks for basic Boolean logic operations within a CPU. These logic gates are constructed by arranging transistors in specific configurations to implement the desired logical function. For example, an AND gate outputs a high signal (1) only when all of its input signals are high (1). By combining different arrangements of transistors, various types of logic gates can be created to perform different logical operations.

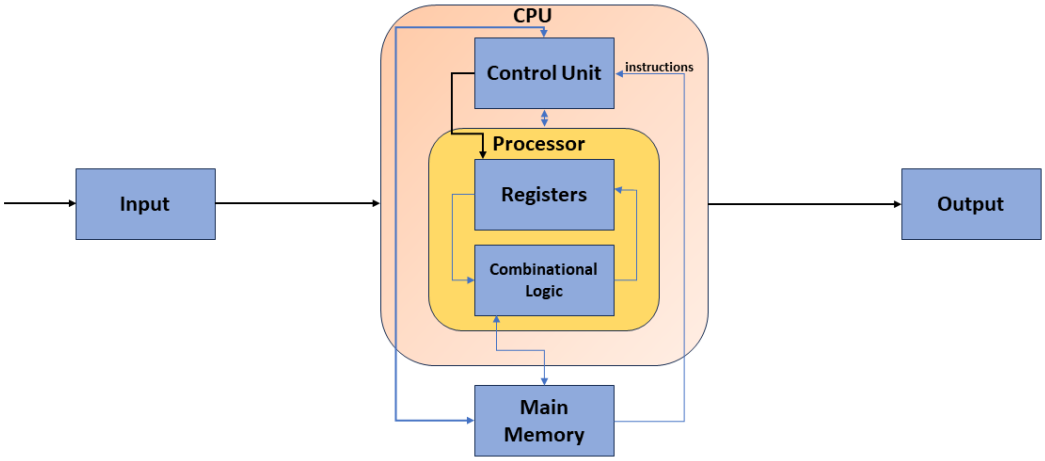


Figure 2-2 CPU Architecture

Logic gates serve as the building blocks for more complex logic circuits. These circuits can implement functions beyond basic Boolean operations, such as arithmetic operations, memory storage, and data processing. Components like adders, multiplexers, flip-flops, and registers are constructed using combinations of logic gates. By combining these and other functional blocks, CPU designers can create custom execution units tailored to specific computational tasks. One of the most critical execution units in a CPU is the Arithmetic Logic Unit (ALU), which performs arithmetic and logical operations on binary data.

The Arithmetic Logical Unit is a digital electronic circuit that performs arithmetic and logical operations on integer binary numbers. Following is the symbolic representation of an ALU:

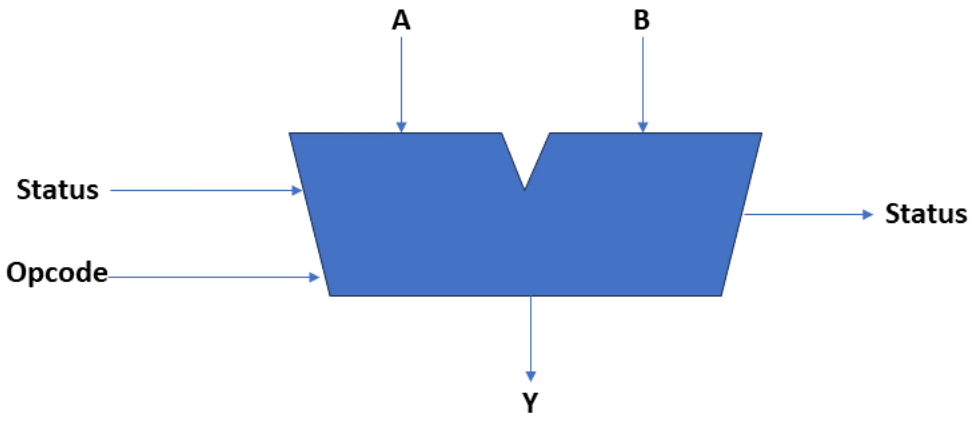


Figure 2-3 ALU

In the given diagram, A, B and Y are three parallel data buses. Each data bus is a group of signals that conveys one binary integer number. A and B takes input and depending on opcode i.e., addition, multiplication etc. Y gets the output. Opcode is, a machine level language, represented by particular combinations of bits. Opcodes are hardwired onto the CPU known as instruction set.

Software developers write code in high-level programming languages such as C, Python, or Java, which is then compiled into machine code by a compiler. The compiler translates the high-level code into a sequence of instructions conforming to the Instruction Set Architecture of the target CPU such as x86 (based on CISC). The Instruction Set Architecture (ISA) serves as an interface between the software and hardware layers of a computing system, like a language that both the software and hardware components of a computer can understand. It specifies the set of instructions that a CPU can execute and how those instructions are encoded in binary format. This provides a standard way for software developers to write programs without needing to understand the intricacies of the underlying hardware and paves the path for software portability across different hardware implementations as long as they support the same ISA.

When the CPU fetches an instruction from memory, it reads the instruction set and uses it to determine which operation to perform. This process is known as opcode decoding. Once the opcode is decoded, the CPU executes the corresponding operation, either directly or by invoking the necessary microoperations within the CPU's control and execution units. The ALU forms the core of the CPU's computational capabilities, executing instructions and performing calculations as directed by the CPU's control unit.

The Control unit directs the operation of other units within the CPU by providing timing and control signals. It manages the flow of data between the CPU and other devices, such as memory, input/output devices, and the ALU. It directs the computer's memory, arithmetic logic unit and input and output devices how to respond to the instructions that have been sent to the processor.

Registers are the fastest form of memory in a computer system. They can be accessed in a single clock cycle, which is significantly faster than accessing data from main memory. They are located directly within the CPU, which makes them immediately accessible to the processor without having to access external memory devices. However, registers have a limited capacity compared to other forms of memory. Modern CPUs typically have a small number of general-purpose registers available for storing data and instructions. Also, they are quite costly than other forms of memory.

2.1 Evolution of Computer Architecture

Primitive computers were very large in size, slow and fragile. They used vacuum tube technology, which was state-of-the-art at the time but made the machine large, power-hungry, and prone to frequent failures due to the unreliability of vacuum tubes. Programming these huge devices was a painstaking process. Operators had to physically rewire the machine and set switches for each new calculation or task, which was a time-consuming and error-prone process. Forget about reprogramming!

In 1940s, Von Neumann's proposed architectural changes laid the foundation for modern computing architectures. He introduced the concept of stored program computers, that can be programmed to perform multiple tasks and have a memory unit attached to it. Most of the modern CPU architecture what we see today is based on the design proposed by him. With advancement in semiconductor technology, transistors replaced the bulky and less reliable

vacuum tubes used in early computer designs, which resulted in creation of smaller, faster, and more reliable computers.

Modern CPU architectures are implemented on integrated circuits, commonly known as microprocessor chips. These chips typically consist of one or two metal-oxide-semiconductor chips, which contain the various components of the CPU, including the control unit, arithmetic logic unit (ALU), registers, and cache memory, among other components. In recent years, there has been a trend towards designing microprocessor chips with multiple CPU cores on a single chip. These chips are referred to as multi-core processors. Each CPU core within a multi-core processor operates independently and can execute its own set of instructions. By integrating multiple CPU cores onto a single chip, multi-core processors can significantly increase computational performance and efficiency, particularly for parallelizable tasks such as multitasking, multimedia processing, and scientific simulations.

In embedded systems, commonly used in in consumer electronics, automotive systems, industrial automation, a microprocessor chip contains a CPU core, memory (both RAM and ROM or Flash), input/output ports, timers, and other peripheral interfaces, and is known as microcontroller. They are designed to be low-cost, low-power, and highly integrated solutions for controlling and managing various tasks within an embedded system. We also have System on a Chip (SoC) integrates not only the CPU core but also additional components such as graphics processing units (GPUs), memory controllers, peripheral interfaces (e.g., USB, Ethernet, HDMI), and sometimes even wireless communication modules (e.g., Wi-Fi, Bluetooth) onto a single chip. SoCs are commonly used in smartphones, tablets, smart TVs, gaming consoles, and other computing devices where space, power efficiency, and integration are crucial.

As computer systems have evolved over time, their internal architecture has changed dramatically over time. And the changes have been aimed at improving performance of the system. In the 1980s, microprocessors typically followed a more sequential execution model compared to modern processors. The process of fetching, decoding, and executing an instruction was typically done one at a time and depending on the complexity of the instruction and the architecture of the processor, it could take several clock cycles to complete each stage before moving on to the next instruction.

Modern systems evolved to execute multiple instructions at a time using techniques such as pipelining, superscalar execution, out of order execution,

speculative execution and hyperthreading. These advancements have enabled modern CPUs to execute instructions much more quickly and efficiently than their predecessors. As a result, the number of instructions executed per unit time, or instructions per cycle (IPC), has increased dramatically. However, in contrast to CPU performance, improvements in memory latency have been relatively modest. While memory capacities have grown substantially in accordance with Moore's Law, the speed at which data can be accessed from memory has not increased at the same rate. This is primarily due to physical limitations in memory technology (*heard of speed of light delay?*), such as the speed of DRAM (Dynamic Random Access Memory) cells and the limitations of memory bus bandwidth.

Speed of a computer is about how fast it can move information from one place to another, which basically means how fast a computer can move electrons within itself. So, the physical limit of an electron moving through the matter is definitely a determinant in the speed limits of a computer system. And speed of electrons cannot surpass speed of the light. What this means that the when the CPU needs to access data from memory, memory cells closer to the CPU will inherently have lower latency compared to those farther away.

In computer systems, memory modules are typically located at varying distances from the CPU. Memory cells closer to the CPU are physically nearer, while those farther away are physically more distant.

Modern processors utilize on-chip cache memories to alleviate memory bottlenecks. Caches help reduce the number of variables that need to be accessed from the main memory (DRAM) by storing frequently accessed data and instructions closer to the processor cores, which reduces the need to access data from slower off-chip memory. This improves the overall system performance by exploiting the principle of locality, where programs tend to access the same set of memory locations repeatedly within a short period of time.

To balance memory size and access speed, modern processors employ a hierarchical structure of caches. This hierarchy consists of multiple cache levels, each with varying sizes, latencies, and access speeds. The numbering convention for cache levels reflects their proximity to the processor. Caches closer to the processor core have lower latency and higher bandwidth but are smaller in size compared to caches farther away.

For instance, L1 Cache is the cache directly attached to a processor core. It operates at a speed close to that of the processor, offering low latency and high bandwidth. However, due to its proximity, it is small in size, typically ranging from 16 to 64 KB. L1 caches are dedicated to each processor core and store frequently accessed data and instructions. L2 caches are larger than L1 caches and are often shared among multiple processor cores or streaming multiprocessors in GPU architectures. They typically range in size from a few hundred kilobytes to a few megabytes. Although larger in size, L2 caches have higher latency compared to L1 caches. Some high-end processors feature an additional level of cache known as L3 cache. L3 caches are even larger than L2 caches, potentially ranging from hundreds of megabytes to a few gigabytes in size. They provide further caching for data shared among multiple processor cores or streaming multiprocessors, which helps in reducing memory access latency and improving overall system performance.

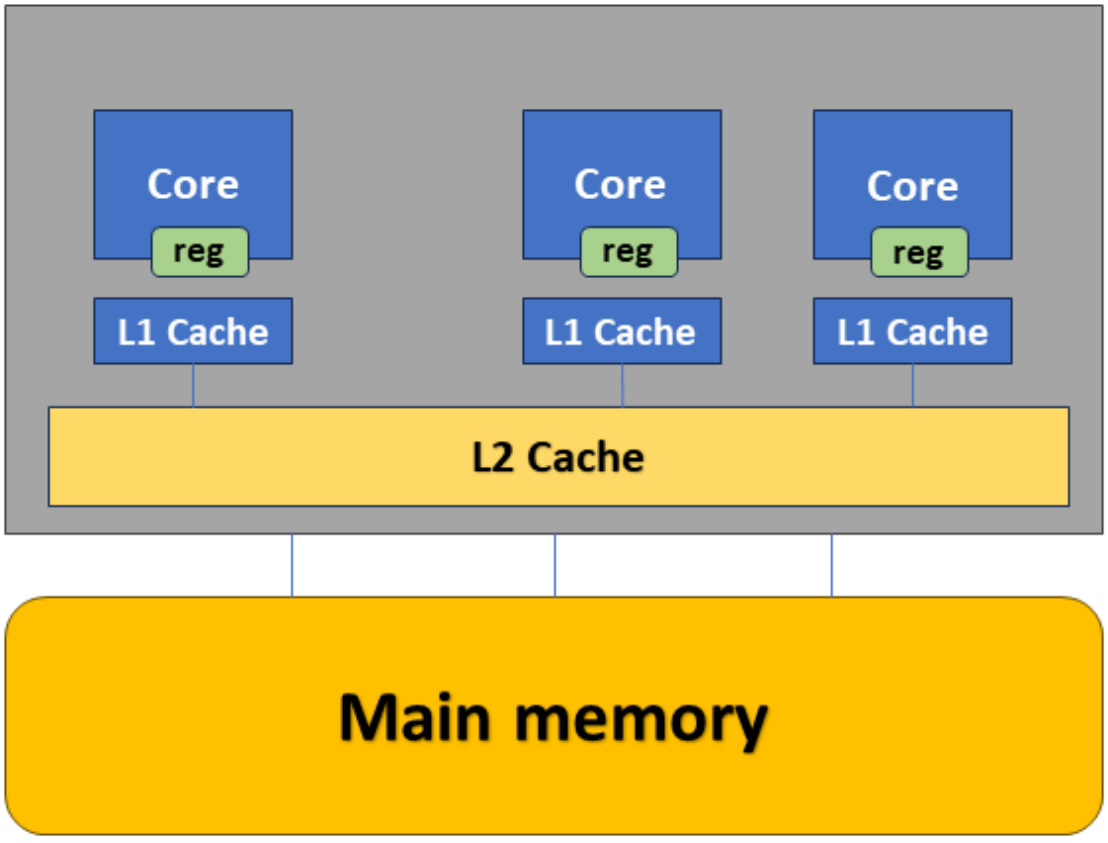


Figure 2-4 Cache Level Hierarchy

Caches exploit the principle of temporal and spatial locality, ensuring that data accessed recently, or data located near recently accessed data is readily available

to the CPU. This reduces the frequency of memory accesses and helps to bridge the performance gap between CPU and memory.

However, even with cache memory, there can still be delays when accessing data from main memory (RAM) or other components of the system, especially if the data needs to travel over relatively long distances on the motherboard or between different components. This delay is called "***Speed of Light delays***".

Let us inspect this further.

The speed of light in a vacuum is indeed approximately 299,792,458 meters per second. It is incredibly fast speed; however, it is still limited. In computer systems, we measure speed in terms of clock frequency, that is how many clock cycles occur per second. For example, a system running at 1.8 GHz has a clock frequency of 1.8 billion cycles per second. Light can only manage about an 8-centimeter round trip during a 1.8 GHz clock period. The fastest CPUs these days have a clock speed of about 5 GHz. This round-trip distance drops further to about 3 centimetres for a 5 GHz clock. How do we explain this?

To calculate the round-trip distance that light can travel during the duration of a clock period, we need to consider both the propagation speed of light and the duration of a clock cycle. Since light travels at a finite speed, it can only cover a certain distance within a given time frame. As the clock frequency increases, the duration of each clock cycle decreases. Consequently, light has less time to travel during each clock cycle, resulting in a shorter round-trip distance. This is why the round-trip distance decreases as the clock frequency increases.

Now to make matter worse, electrons don't flow in vacuum inside computer systems but through silicon. While light travels quickly in a vacuum, electric waves in silicon move much more slowly, typically three to thirty times slower than light in a vacuum. This slower movement of electric waves within silicon further constrains the speed at which signals can propagate within a computer system. For example, a memory reference may need to wait for a local cache lookup to complete before the request may be passed on to the rest of the system. Additionally, relatively low-speed and high-power drivers are required to move electrical signals from one silicon die to another, such as to communicate between a CPU and main memory.

What is implied here that CPU clock frequencies can't go higher infinitely. Now you can relate why choosing the right location for your data center is important.

Even with faster processors and improved memory architectures, the memory hierarchy — ranging from registers to caches to RAM to storage — will continue to exist. Caches serve as a critical component of this hierarchy by providing faster access to frequently accessed data. While CPUs are getting faster, memory access times are not improving at the same rate. Caches help mitigate the latency gap between the CPU and main memory by storing frequently accessed data closer to the CPU, reducing the need to access slower main memory frequently.

However, when it comes to common operations such as traversing a linked list, computer systems have to deal with unpredictable memory access patterns. Linked lists, especially those with nodes scattered across memory, exhibit poor spatial locality since accessing one node does not imply proximity to other nodes. Hence, traversing the list may cause cache thrashing, where cache lines are continuously replaced as new nodes are accessed. Each cache miss incurred during traversal may result in loading new cache lines into the cache, only to evict them shortly afterward due to subsequent accesses to different memory locations. This constant churn of cache lines leads to poor cache utilization and can negate the benefits of caching. Additionally, linked list traversal may not exploit parallelism effectively, as each node's traversal typically depends on the completion of the previous node's traversal. This sequential dependence limits the ability of modern CPUs to execute instructions in parallel and fully utilize their multiple execution units.

2.2 Interaction with a CPU Core

A **CPU core** is the physical processing unit of a central processing unit (CPU). It is the component responsible for executing instructions and performing calculations. Back in the day, computer CPUs used to have single CPU core. In modern computer systems, a CPU can have multiple cores and several device controllers connected through a common bus that provides access between components and shared memory. The more cores a CPU has, the more independent tasks the processor can run simultaneously.

To effectively understand how OS works in conjunction with CPU, let us observe how the users interact with a system from the perspective of a single core CPU.

Typically, there is a device driver for each device controller, which provides a uniform interface (abstraction) to the rest of the operating system for the corresponding external device. The OS relies on device drivers to interact with

peripheral devices. Each device controller manages a specific type of device e.g., a disk drive, audio device, or graphics display.

Any I/O operation (transfer of data to or from a computer system) is initiated by making a request to a specific device driver, which loads the appropriate registers in the device controller with the necessary information such type of operation to be performed, memory addresses, etc. The device controller, a hardware component responsible for managing the specific I/O device, examines the contents of the registers to determine the action it needs to perform (e.g., reading a character from the keyboard). It starts the transfer of data between the I/O device and its local buffer. This is often a time-consuming process, especially while dealing with slower peripheral devices like disk drives.

Once the transfer of data is complete or a certain condition is met, the device controller signals the CPU through via the system bus, which is the primary communication path between major components in a computer system, to notify about events that require immediate attention. These signals are termed as **interrupts**. Interrupts can also be generated internally due to exceptions in computation or trying to access restricted memory.

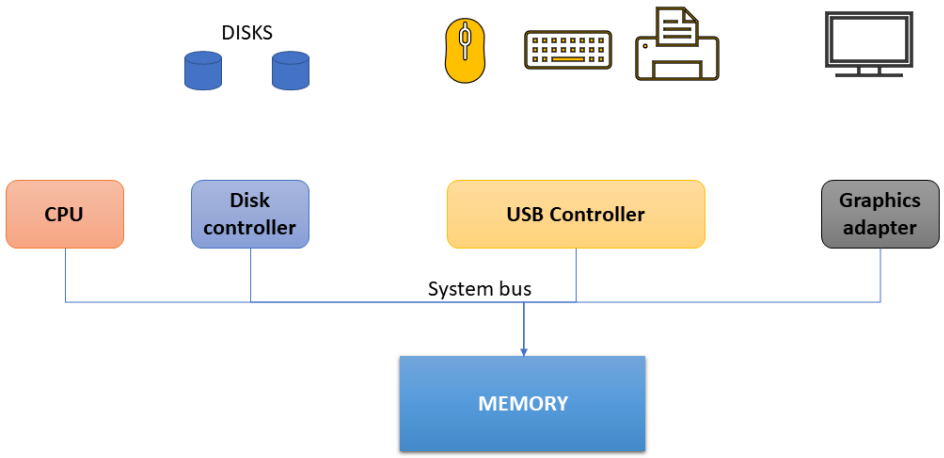


Figure 2-5 Computer Organization

When CPU receives an interruption, it stops its current task, saves its state (*more on this in upcoming section*) and transfers control to fixed location in memory known as the **Interrupt Vector Table (IVT)**. The IVT contains a list of addresses corresponding to different types of interrupts. Each entry in the IVT points to the

starting address of the ISR associated with a specific interrupt type. The CPU retrieves the address of the Interrupt Service Routine (ISR) associated with the received interrupt from the IVT and jumps to this address to start executing the code within the ISR.

ISR is part of the device drivers responsible for managing communication between hardware devices and the operating system. and performs actions related to the interrupt, such as processing data, handling I/O operations, or responding to specific events. When an interrupt occurs, indicating an event that requires immediate attention, the CPU transfers control to the ISR associated with the device that triggered the interrupt. The ISR then performs tasks specific to the device, such as reading data from or writing data to the device, acknowledging the interrupt, or initiating further processing. It executes the assigned tasks in a timely manner to ensure that the interrupt is handled efficiently, and the system's responsiveness is not compromised.

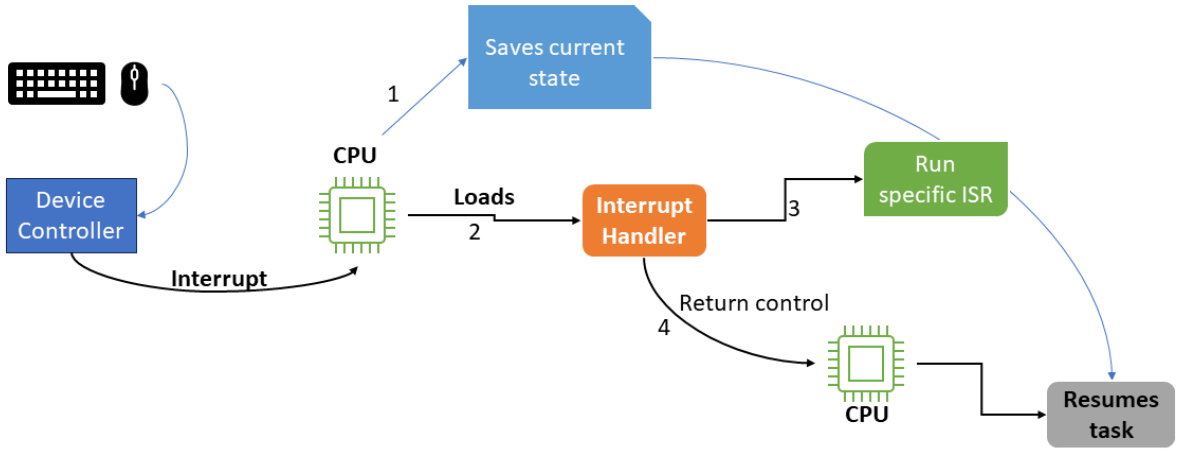


Figure 2-6 Interrupt Handling mechanism

Once the ISR completes its execution, it returns control back to the CPU, which then restores its saved state, including the program counter and register values and continues execution of the interrupted task from where it left off.

---

*The above process where CPU saves its context to switch over to perform a different task is called **context switching.** This form of communication between them works well for transfer of small amounts of data, however for modern systems, there is a huge overhead, as the system is doing no useful work while context switching. The time taken in context switch varies from machine*

to machine and is dependent on memory speed, number of registers to be copies, etc.

Especially while transferring bulk data between computer's main memory and Non-volatile storage devices, there will be interrupts generated for each byte of data transferred. As a result, each interrupt triggers a **context switch**, where the CPU switches between the normal execution of the program and the **ISR** associated with the I/O operation. Also, CPU is actively involved in managing and overseeing each byte transfer. This involvement includes updating pointers, handling interrupts, and ensuring data integrity. To overcome this overhead, **Direct Memory Access (DMA)** is commonly employed, as it allows for the efficient transfer of entire data blocks between the NVS device and main memory with minimal CPU intervention, reducing the CPU overhead and improving I/O transfer rates.

Before initiating data transfers, the CPU sets up buffers, pointers, and counters for the I/O device in coordination with the DMA controller. The DMA controller is a specialized hardware component that has its own set of registers to control the transfer parameters and operates independently of the CPU once configured. Once the setup is complete, the device controller, under the control of the DMA controller, transfers an entire block of data directly between the I/O device and main memory. Unlike interrupt driven I/O, where each byte transfer might trigger an interrupt, DMA generates only one interrupt per data block. While the DMA controller is handling the data transfer, the CPU is freed from involvement in the process and can perform other tasks concurrently, enhancing the overall system efficiency and throughput.

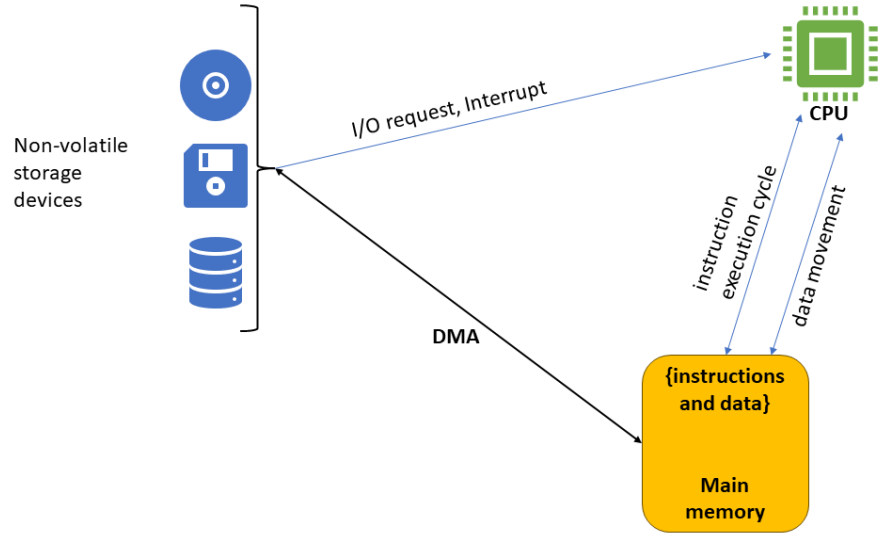


Figure 2-7 Dynamic Memory Allocation for bulk data transfer

It must be clear from our discussion that a computer with single CPU core can perform tasks concurrently by context switching or via DMA. ***Concurrency means all processes(tasks) progressing together but at one time only one task being executed by the CPU core.*** Tasks compete for CPU time. Traditional systems evolved from using one CPU core to employing multiple CPU cores with each CPU core performing tasks in parallel, and thus allowing for more scope for multi-tasking. Each CPU core can be involved in execution of separate process in parallel. Notice the gradual evolution from ***concurrency to parallelism***. In a multi-core system, multiple processor cores operate parallel, which reduces contention for the CPU among different tasks and helps distribute the workload. Modern systems take a mixed approach of using concurrency on one core and distributing tasks across multiple cores (parallelism).

So far in our discussion, we have talked about how a computer or OS reacts to the events generated by users and processes the instructions. But what does it mean by execution of an instruction or performing a task? Let us shift our discussion briefly towards what a process is and then we shall resume our discussion related to multiple processes executing together.

2.3 Process

In the computing world, we provide the computer with a set of instructions as **a program**, to execute a specific task, which is stored in the secondary memory like a hard disk as a passive entity. When we give command to the computer to

execute a specific task, CPU loads the related instructions in the main memory and becomes an active entity, called as **process.**

An example of a program is a word processing application, a web browser, or a video game. These programs are sets of instructions that define how a computer should behave when running the corresponding applications.

Process is an instance of a program running on a computer. It is a dynamic entity that occupies system resources such as memory, CPU time, and I/O channels. A process includes the program code and data and additional information required for its management by the operating system.

Each process has an organized address space and is utilized during execution. The memory layout of a process is divided into multiple sections as shown below:

- The **text section** (code segment) of the memory layout contains the executable code of the program. During execution, the CPU fetches instructions from this section, advancing the **program counter** to execute each instruction. The **Program Counter (PC)** is a special-purpose register (a small, fast-access storage location that holds data temporarily during processing) in a computer's central processing unit (CPU) that keeps track of the address of the next instruction to be executed in a program. It is crucial for controlling the flow of the program in a sequential manner. During a context switch between different processes in a multitasking environment, the contents of the Program Counter are saved and later restored to ensure that each process resumes execution from the correct point.
- The **data section** holds initialized global and static variables. These variables retain their values throughout the program's execution. This section is crucial for storing data that persists across function calls.
- The **stack** is used to manage function calls and store local variables. Each function call results in a new stack frame (**activation record**) being pushed onto the stack. Local variables and function parameters are stored within these stack frames.
- The **heap** is utilized for dynamic memory allocation, allowing the program to request memory at runtime using functions like *malloc* or *new* (in languages like C or C++). The heap accommodates data structures that can grow or shrink dynamically during execution.

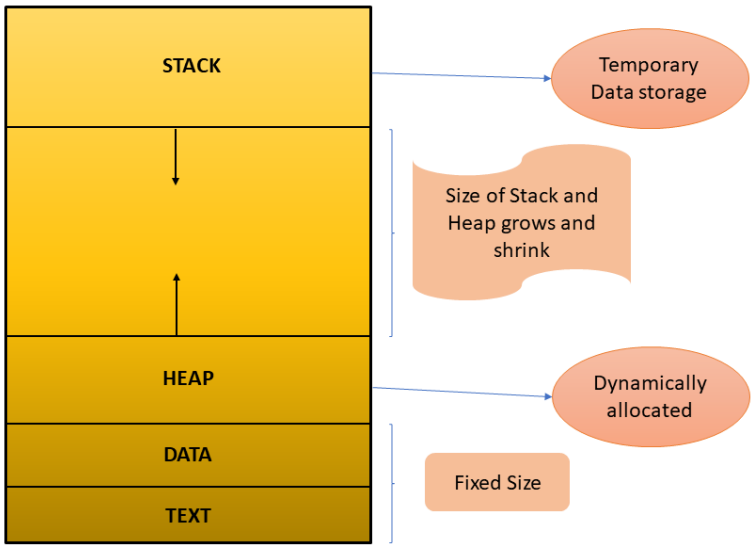


Figure 2-8 Memory layout of a process

The stack and heap sections in memory are managed by the OS to prevent them from overlapping. While it is possible for the stack and heap to overlap, the OS typically starts them far apart to avoid this. If the stack and heap were to overlap, it could lead to issues such as buffer overflow, unpredictable behavior, and memory corruption. Modern operating systems use various mechanisms, such as virtual memory guard pages, to prevent the stack from growing into the heap and vice versa. Guard pages are memory pages strategically placed between the stack and heap segments by the operating system during memory allocation. These pages are marked as inaccessible, which basically means that any attempt to access these guard pages (read/write) will trigger an error, typically a segmentation fault or a similar memory access violation. So, an effective memory barrier is created between the stack and heap. Any attempt by a program to access memory beyond the stack or heap boundaries, such as by attempting to grow the stack into the heap or vice versa, will trigger an error when it encounters the guard page.

The Information about the memory layout of a process is stored in a data structure called the **Process Control Block (PCB)**. It contains details about the process, including the program counter, register values, memory allocation, and status.

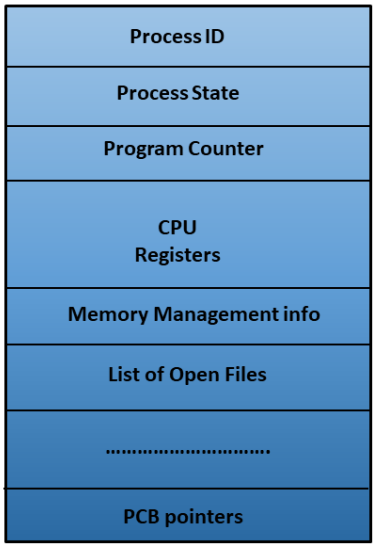


Figure 2-9 Process Control Block (PCB)

Let us look at some important sections of a PCB:

- **Process ID:** Each process is assigned a unique identifier which helps the OS distinguish it from different processes.
- **Process state:** Each process goes through various state changes like new, ready, running, waiting, etc. and the information is stored in this section.
- **Program Counter:** We discussed about Program counter above. It is a special purpose register which is used to keep track of the address of the next instruction to be executed.
- **CPU registers:** Depending upon a computer, there can be various CPU registers like general-purpose registers, status registers, and other special-purpose registers. Some common registers present across the platforms are accumulators, index registers, stack pointers, etc.
- **CPU Scheduling Information:** The information about a process's scheduling state, priority, and other attributes used by the CPU scheduler to determine the order of execution is stored in this section.
- **Memory Management Information:** It includes critical details about how a process interacts with the memory system. The value of base and limit registers, page tables, etc. are saved in this section. The base register holds the starting address of the memory block allocated to the process. The limit register contains the size of the memory block, restricting the process from accessing memory beyond this limit.

*During a **context switch**, the PCB is used to save the current state of the process, allowing the operating system to switch to and execute another process. When a process completes its execution, the PCB is typically updated to reflect its terminated state. The process's resources are then released. Information in the PCB is utilized for inter-process communication (IPC, more on this in coming section) and synchronization mechanisms, facilitating communication between processes. The PCB can be viewed as a snapshot of the process's state and serve as a reference point for the OS to make decisions regarding scheduling, resource allocation, and process execution.*

It is important to note that each process can create several new processes (child processes) during its course of execution. When a child process is created, the allocation of resources such as CPU time, memory, file access, etc. is done by the OS dynamically as it adjusts resource limits based on current system load and priorities of processes. These resources might be distributed directly to these processes or as a subset of the available resources to the parent process. The later approach is recommended, as it helps prevent system overload. It allows the OS to maintain control over resource usage and prevent any single process, whether poorly designed or malicious, from monopolizing system resources. Without such restrictions, a poorly designed or malicious process could create an excessive number of child processes, potentially overwhelming the system. Upon process termination, whether voluntarily or due to an error, all resources associated with the process are deallocated and reclaimed by the OS. If the terminated process has a parent, the operating system may notify the parent about the completion of the child process, so that the parent can perform any necessary cleanup or handle the termination event.

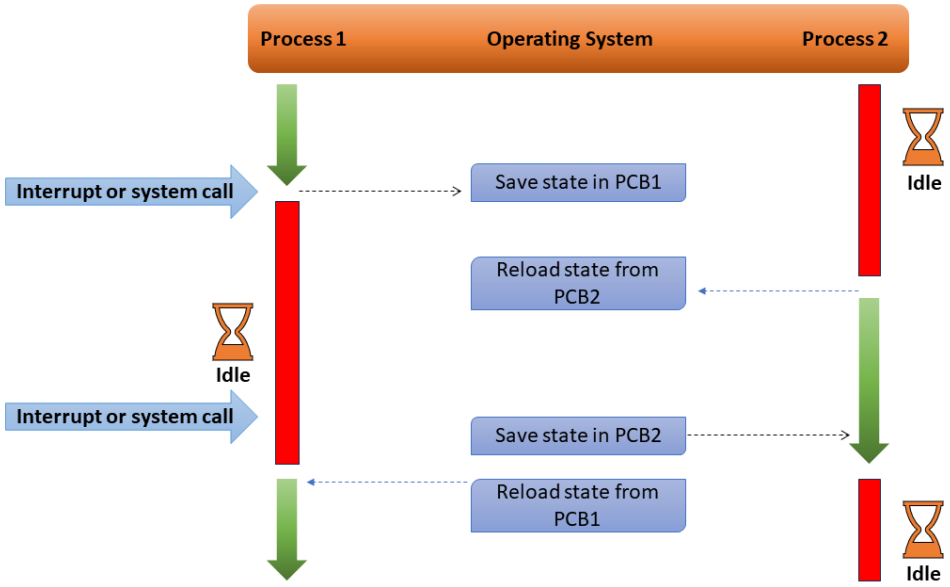


Figure 2-10 Context switch between two processes

Let us understand this by considering the multi-process architecture of Chromium browsers:

When we launch the browser, it starts a main process (parent process) that manages the user interface and browser functions. Additionally, Chrome uses a multi-process architecture where each tab, extension, and plugins run in their own separate processes (child processes). Each open tab in the Chrome browser is associated with a separate renderer process, which manages the rendering and display of web pages, ensuring that each tab runs independently. If one tab meets an issue (e.g., a crash), it doesn't affect the entire browser. Chrome may also spawn a separate GPU process to handle graphics-related tasks. There are other processes also running within Chrome browser such as Extension process and utility processes. The list of processes running under Chrome can be viewed by opening its task manager. This design enhances security and stability, as issues with one tab or process are less likely to affect others.

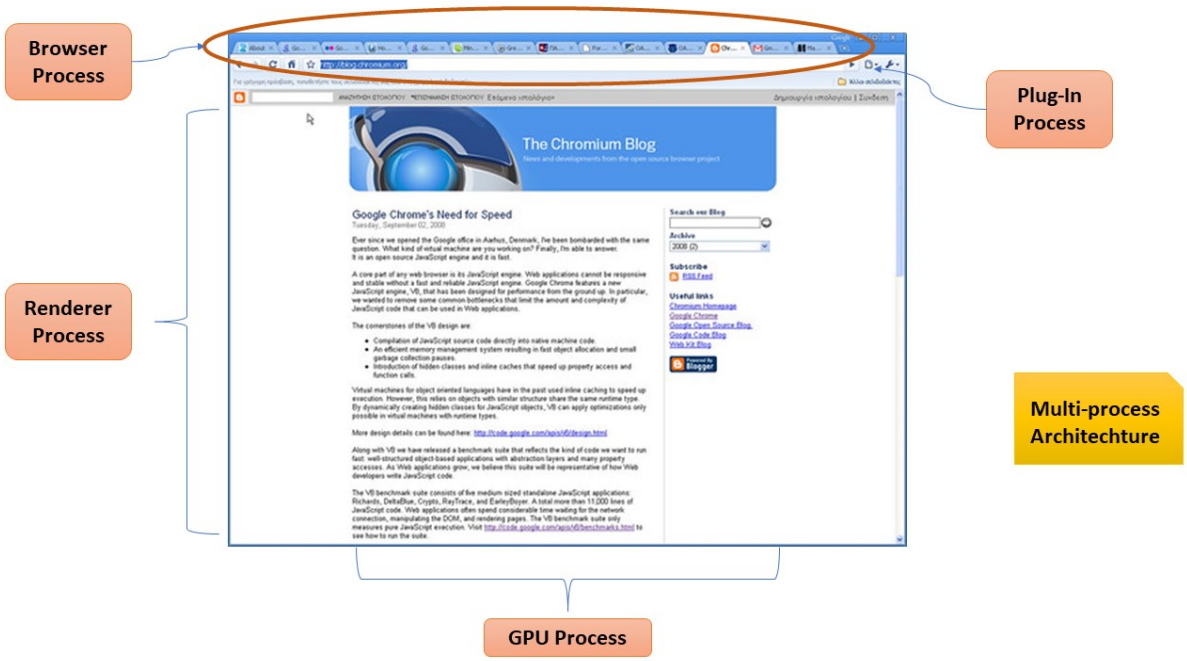


Figure 2-11 Chromium Multi-process example

2.4 Inter-process Communication (IPC)

A process which does not share data with any other process is **an independent process** and if a process shares data with other processes in system, it is a **cooperating process**. Processes not just run parallel but also communicate with each other in the form of exchanging data.

Supplying a suitable environment for inter-process communication is important for assorted reasons. Sometimes, multiple applications get interested in the same piece of information or parallelly executing child processes need to be communicating with each other to speed up the computation. In general, there are two fundamental models for inter-process communication: **shared memory** and **message passing**.

Shared memory, as the name suggests, allows multiple processes to share a common region of memory. In shared memory model, processes must agree to set up a shared-memory region, and they need to coordinate their access to this shared region. One of the processes (typically referred to as the creator or producer) starts the creation of a shared-memory segment using system calls and this segment exists in the address space of the creating process. Other processes (consumers or clients) that wish to communicate using the shared-memory segment must attach it to their own address space. Once attached, processes can read from and write to the shared-memory segment as if it were their own memory. Processes using shared memory must coordinate their