CPU Core

A core is the physical unit in a CPU that performs tasks or instructions.

1. Single-Core CPU : can only execute one instruction at a time.

2. Multi-Core CPU : multiple cores allow parallel execution of tasks.

Examples: 2-core, 4-core, 8-core CPUs, up to 64-core CPUs.

Analogy

Imagine a restaurant with one chef (single-core CPU). The chef can cook only one dish at a time.

Add more chefs (multi-core CPU), and now the restaurant can cook multiple dishes at once.

Thread

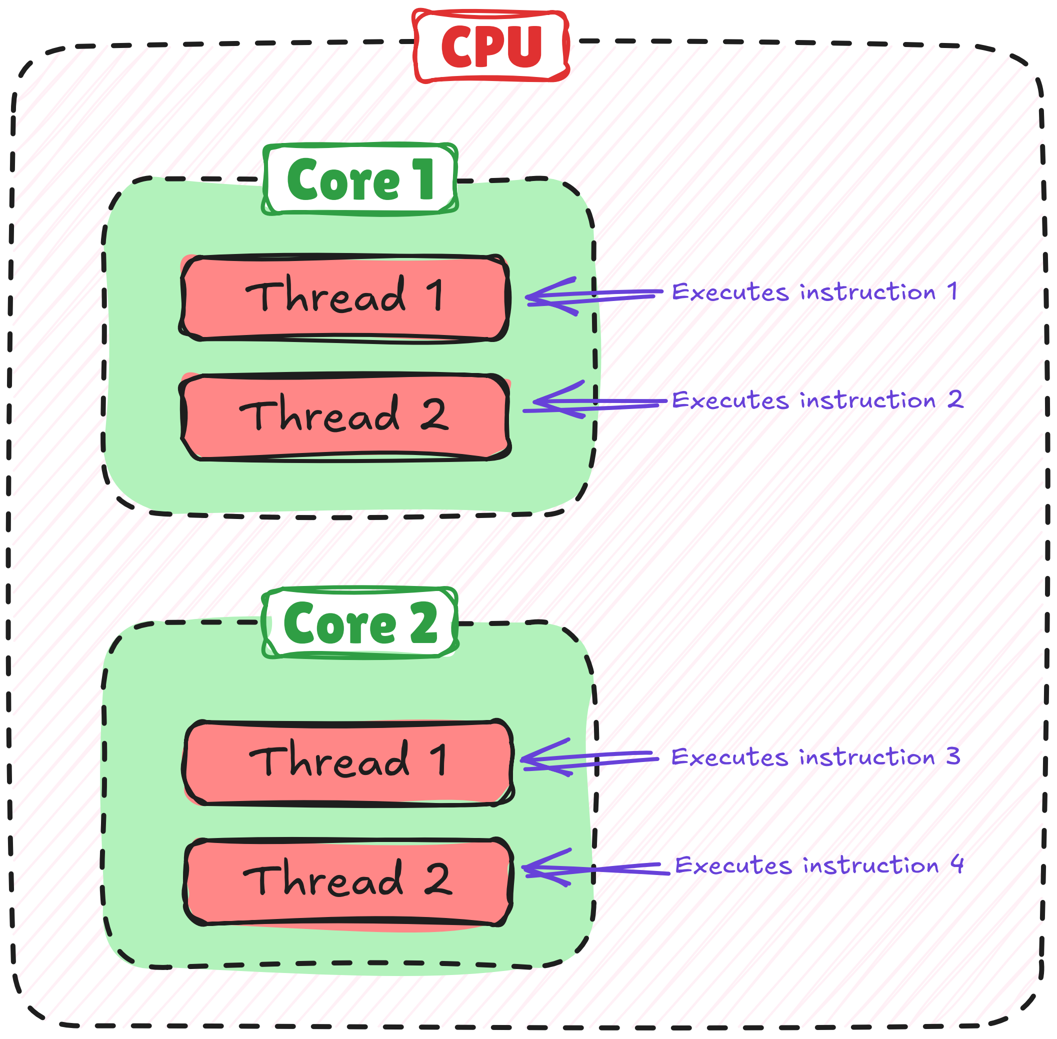
A thread is a virtual core, a software method to split a core's capacity.

Hyper-Threading (HT) or Simultaneous Multithreading (SMT) enables a core to handle multiple threads at once.

A thread is not a physical core but a way to divide a core's power to run more tasks.

Analogy

A chef (core) uses two hands (threads) to handle two tasks simultaneously, like chopping vegetables and stirring a sauce.



CPU Example:

CPU	Cores	Threads per Core	Total Threads
Intel i3-10100	4	2	8
AMD Ryzen 7 5800X	8	2	16
Apple M1	8	1	8

Process

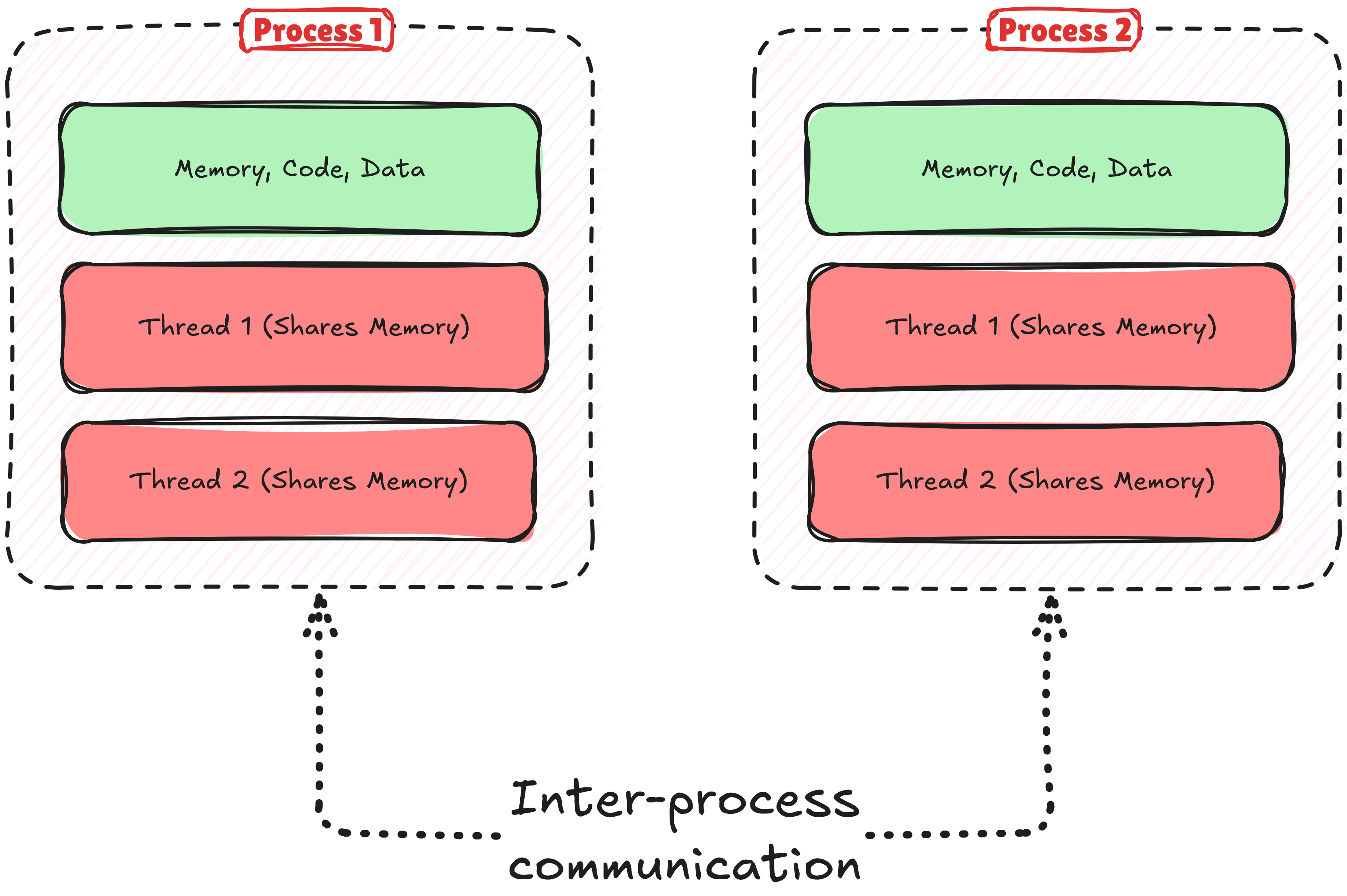
A program (app) running on your computer is a process.

A process can use multiple cores and threads to complete tasks.

Analogy

A process is like a customer order in a restaurant

A customer order might require multiple chefs (cores) to make a meal faster.
A burger, fries, and drink could each be prepared by different chefs at once.



Thread vs Process

Thread

Shares the process’s memory but can run concurrently.

Process

Independent execution context with its own memory space.

CPU Performance Factors

Clock Speed (GHz)

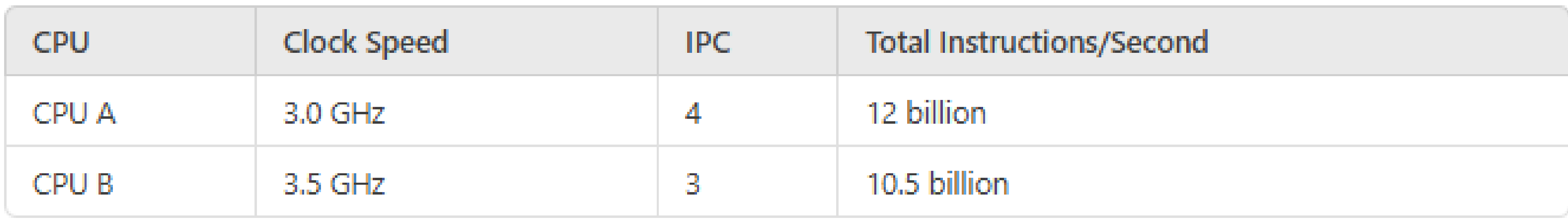
Number of cycles per second. More cycles mean faster performance.

Example: 1 GHz = 1 billion cycles per second.

IPC (Instructions per Clock):

IPC (Instructions per Clock): How many instructions the CPU can execute in one cycleperformance.

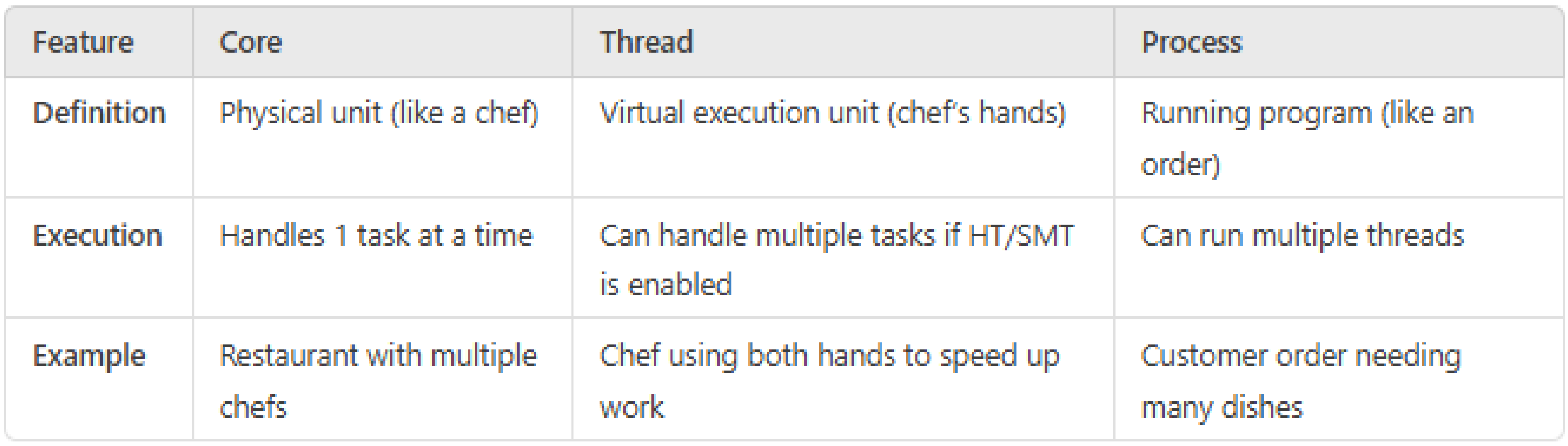
Performance = Clock Speed * IPC



CPU	Clock Speed	IPC	Total Instructions/Second
CPU A	3.0 GHz	4	12 billion
CPU B	3.5 GHz	3	10.5 billion

Conclusion: Even if **CPU B** has a higher clock speed, **CPU A** can be faster due to its higher **IPC**.

Summary

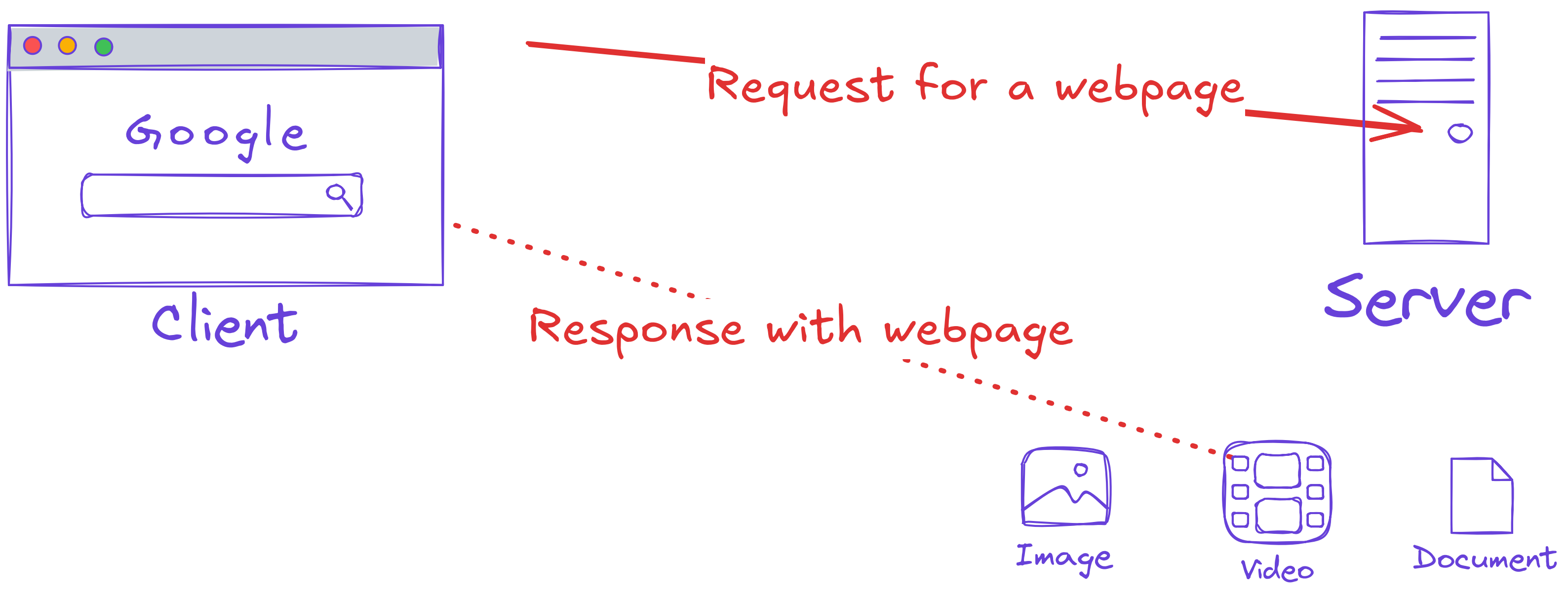


Feature	Core	Thread	Process
Definition	Physical unit (like a chef)	Virtual execution unit (chef's hands)	Running program (like an order)
Execution	Handles 1 task at a time	Can handle multiple tasks if HT/SMT is enabled	Can run multiple threads
Example	Restaurant with multiple chefs	Chef using both hands to speed up work	Customer order needing many dishes

More cores and threads allow a CPU to **handle more tasks simultaneously**, improving performance and efficiency!

Web Servers

A web server is software and hardware that uses HTTP (Hypertext Transfer Protocol) and other protocols to respond to client requests made over the World Wide Web.



APACHE HTTP SERVER

Microsoft IIS

Caddy

Popular Web Servers

Web Server serves the content over the web. This content can be either static or dynamic, depending on whether it changes based on user interactions.

Static content refers to web files that remain unchanged and are served as they are stored on the server. These files do not require any backend processing and are directly delivered to users.

Examples of Static Content:

1. HTML pages → A simple About Us page that looks the same for every visitor.
2. CSS stylesheets → The design and layout of a website.
3. JavaScript files → Frontend functionality like animations.
4. Images and videos → Logos, banners, and pre-recorded videos.

Examples of Dynamic Content:

1. User dashboards → A profile page that shows personalized information.
2. Search results → A list of products based on user queries.
3. Live data updates → Stock market prices or real-time weather forecasts.
4. User authentication → Logging in and retrieving user details from a database.

Features of Web Servers

Serving Static Files Efficiently

Web servers handle and deliver static files such as HTML, CSS, JavaScript, and images, ensuring fast and efficient content delivery.

Dynamic Content Processing

Supports server-side programming languages like Python, Node.js, and PHP to generate dynamic web pages based on user requests.

Logging & Monitoring for Insights

Maintains access logs and error logs to track server performance, troubleshoot issues, and enhance security monitoring.

Robust Security Features

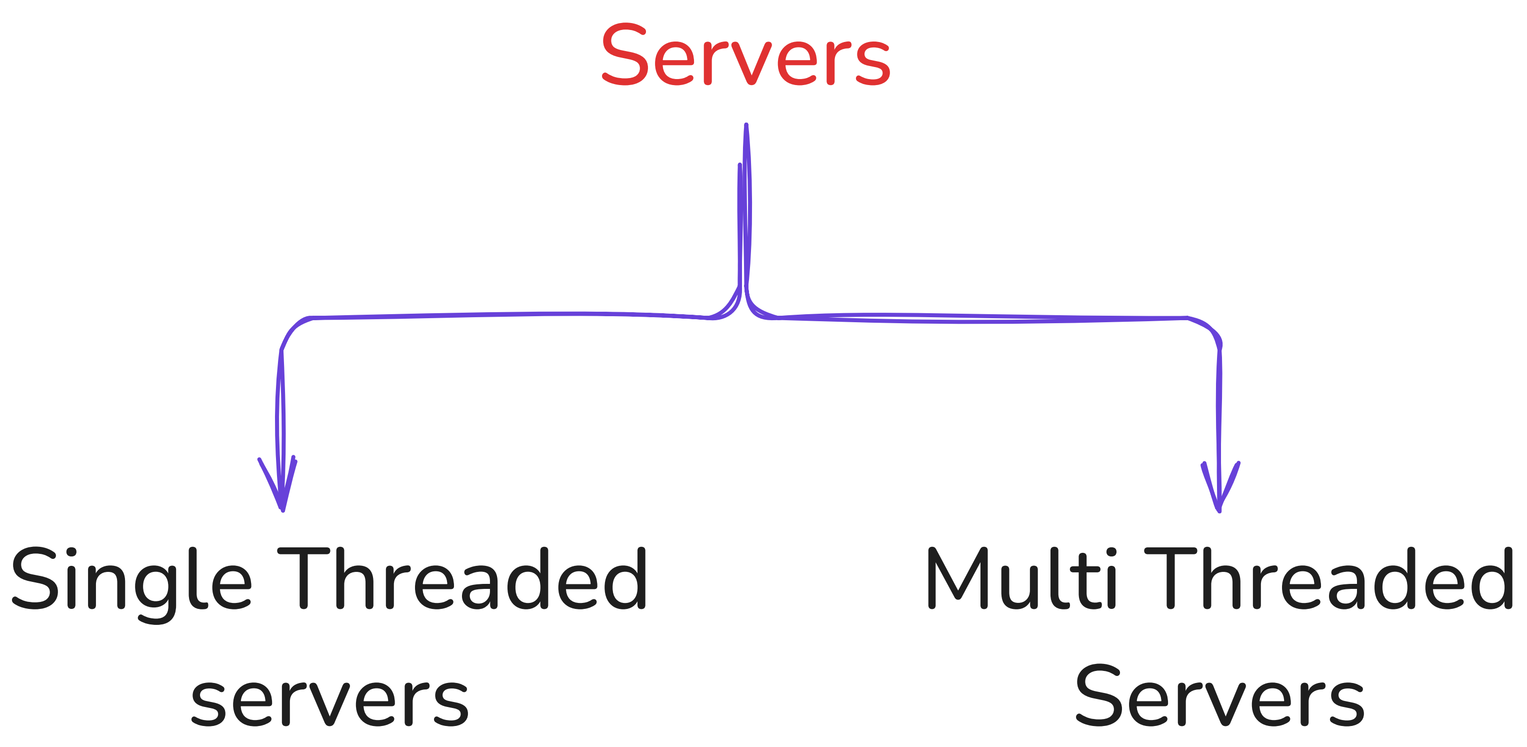
Implements SSL/TLS encryption, access controls, and firewalls to protect web applications from cyber threats and unauthorized access.

Load Balancing & Reverse Proxy

Distributes incoming traffic across multiple servers to optimize performance, prevent overload, and enhance availability.

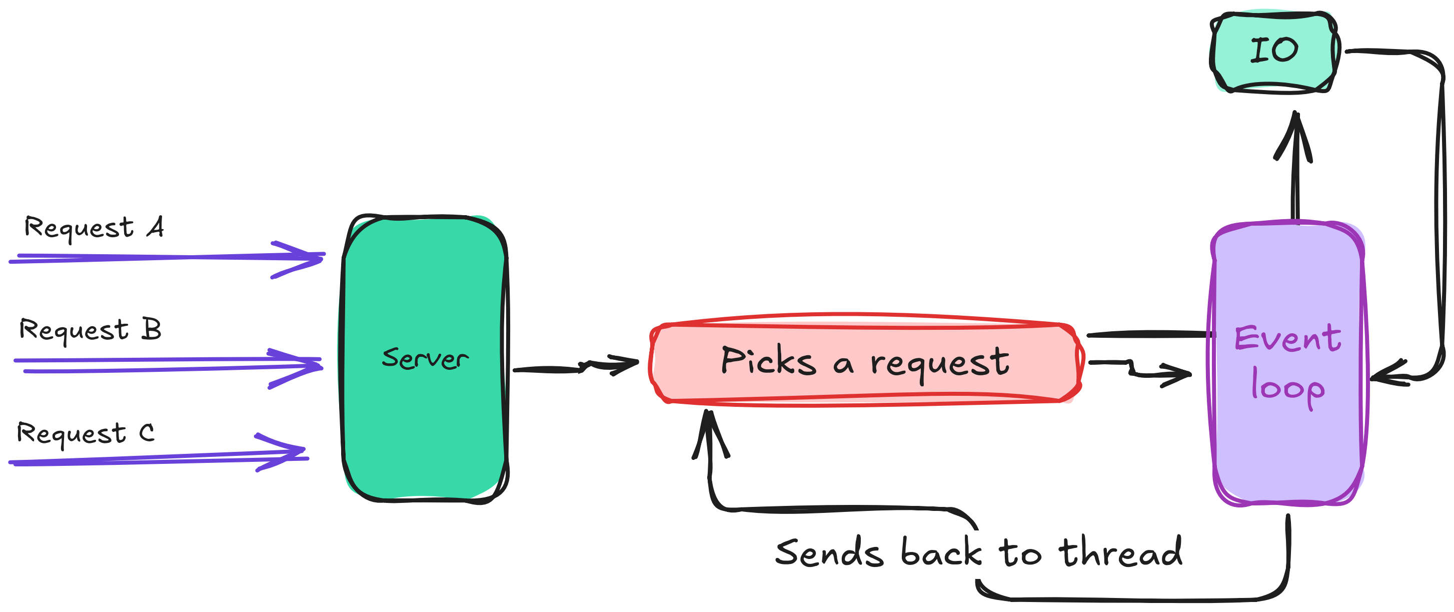
Efficient Caching Mechanisms

Stores frequently requested data to reduce server load, improve response times, and enhance the user experience.



Single-Threaded Server

A single-threaded server is a server that uses just one thread of execution to handle all requests. In other words, it processes one request at a time in a single flow, rather than creating or using multiple threads to handle requests in parallel. This can still be efficient if it uses non-blocking, event-driven operations (like Node.js), but it only operates on a single thread of the CPU.



Although there is only a single core, but multiple requests are handle concurrently due to non blocking I/O & Context Switching.

Context Switching

Context Switching allows the CPU to switch quickly between processes, making it appear like multiple tasks are running simultaneously on fewer cores/threads.

Steps

1. Task A starts running on the CPU.
2. The OS pauses Task A, saves its state.
3. The OS starts Task B by loading its state.
4. The cycle repeats, so it seems like tasks run in parallel.

Limitation of Single Threaded Servers

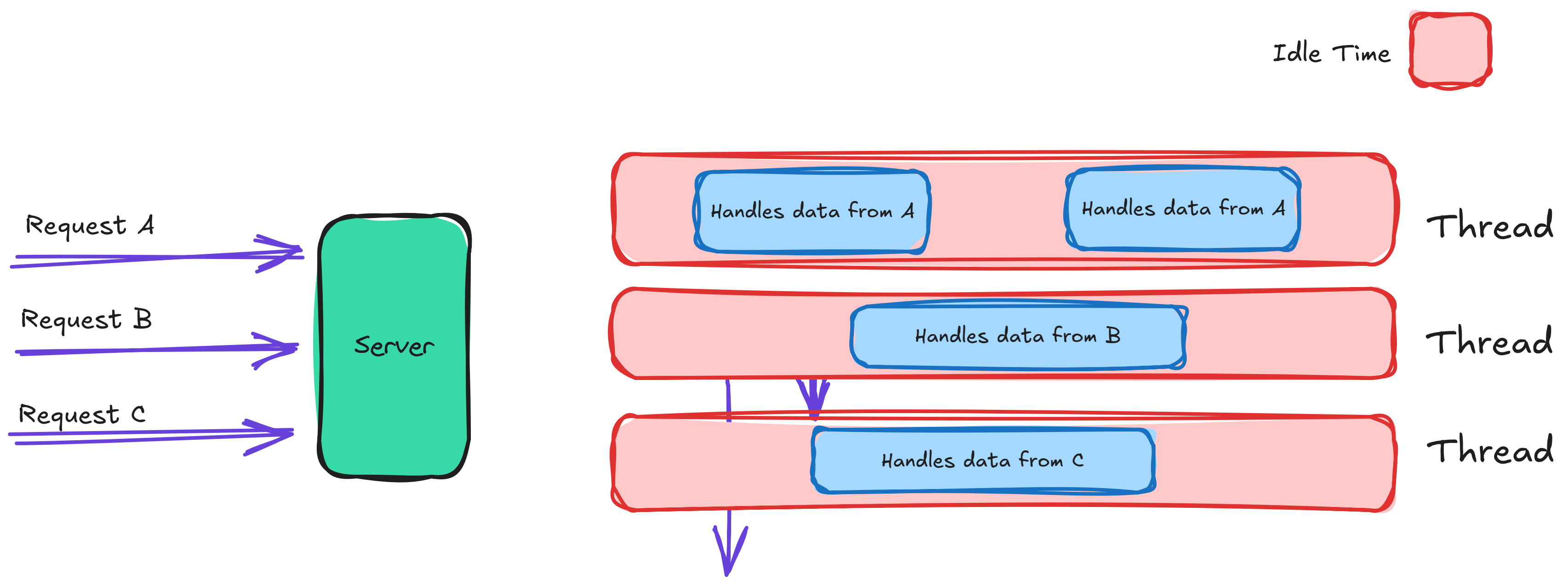
If there is a blocking I/O or CPU intensive task then the main thread will be blocked and rest of the other requests has to wait till main thread becomes available again. So for high non-blocking I/O operations single threaded servers like Node.js is preferred. But for CPU intensive tasks such as image processing or video processing we need mutli-threading.

Multi-Threaded Server

A multithreaded server is a server that uses multiple threads of execution. Each incoming request can be handed off to its own thread (or a thread from a pool), allowing the server to process multiple requests concurrently. This can improve performance on systems with multiple cores or CPUs, but also introduces added complexity (e.g., synchronization and thread management).

Multi-Threading is used for tasks like image processing or machine learning, which benefit from parallel execution.

If there is a blocking I/O operation or CPU intensive task then multi-threading is preferred.



Multi-Threaded Server

Limitation of Multi Threaded Servers

When multiple threads access and modify shared data at the same time, it can lead to inconsistent or incorrect results. For example, if two threads read the same variable and both increment it, the result might be incorrect (e.g., two increments resulting in a final value of 1 instead of 2).

To avoid race conditions and ensure thread-safe access to shared memory, synchronization techniques (such as locks, mutexes, semaphores) need to be implemented. Writing, maintaining, and debugging multithreaded code becomes more complex because of the need to manage these synchronization mechanisms.

Probability of deadlocks are high in multi-threading. Deadlocks occur when two or more threads are blocked forever because each is waiting for the other to release a resource. This typically happens when threads hold one resource while waiting for another.

Comparison

Aspect	Single-Threaded Server	Multi-Threaded Server
Concurrency Model	Uses non-blocking I/O to handle many connections in a single event loop. Can handle many connections if tasks are mostly I/O-based.	Creates or uses multiple threads (or a thread pool). Each request can be handled in a separate thread, allowing multiple truly parallel flows.
Complexity	Generally simpler to develop and debug (no shared memory concurrency, fewer race conditions).	More complex, due to potential race conditions, deadlocks, and need for synchronization across threads.
Resource Usage	Uses less memory because there's usually only one main thread and possibly a small internal thread pool (like in Node.js).	Each thread uses additional stack space and management overhead, leading to higher memory usage especially for large numbers of threads.
CPU-Bound Performance	If a task is CPU-heavy, it blocks the single thread/event loop, stopping other tasks.	Can distribute CPU-heavy tasks among different threads (on multi-core systems), increasing overall throughput.
Typical Use Cases	Good for many small, quick I/O tasks without blocking, but one large CPU task can degrade response for all.	Multiple requests/tasks can proceed in parallel, improving responsiveness for CPU-bound or mixed workloads, at cost of threading overhead.

Synchronous V/s Asynchronous Operations

Thread

Synchronous Operations: In synchronous operations, the thread waits for the task to complete before moving on to the next task. This means that the thread cannot perform any other tasks until the current task is finished.

Asynchronous Operations: In asynchronous operations, the thread does not wait for the task to complete. Instead, it can continue executing other tasks while waiting for the task (e.g., file read, network request) to complete in the background.

Blocking/Non-Blocking

Synchronous Operations: These operations are blocking. The thread is blocked and cannot proceed until the task, such as a file read or network request, is completed.

Asynchronous Operations: These are non-blocking. The thread continues executing other tasks while it is waiting for a task to complete. It can switch to other work without having to wait for the task to finish.

Efficiency

Synchronous Operations: Synchronous operations can be inefficient because the thread is idle and does not perform any work while waiting for the task (such as a file read or I/O operation) to complete.

Asynchronous Operations: Asynchronous operations are more efficient because the thread is free to perform other tasks while waiting for the I/O operation to finish. This allows for better resource utilization.

Use of Resources

Synchronous Operations: The thread becomes idle and consumes system resources without performing any meaningful work during the wait. This results in wasted resources.

Asynchronous Operations: The thread remains busy, efficiently utilizing system resources by working on other tasks while waiting for the primary task to complete.

Task Completion

Synchronous Operations: In synchronous operations, the thread is blocked until the task is completed, which means no other tasks can be performed in the meantime.

Asynchronous Operations: In asynchronous operations, the thread receives an event or callback once the task is completed, and it processes that result when notified, without blocking other tasks.

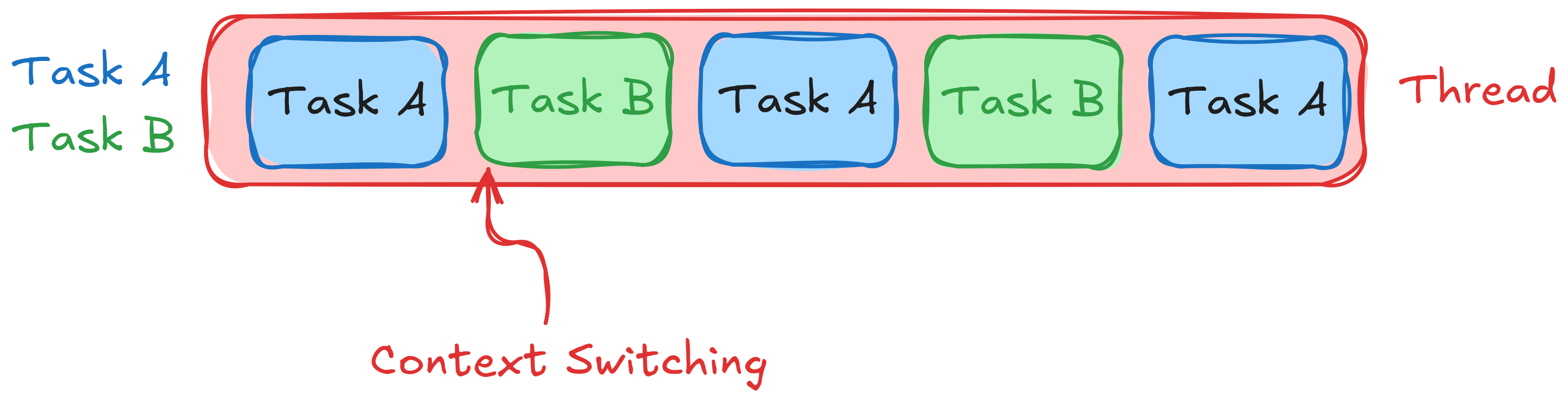
Example of Usage

Synchronous Operations: Synchronous operations are typically used in traditional, blocking environments, where tasks are executed in sequence and the thread waits for each task to finish (e.g., file read operations in a traditional system).

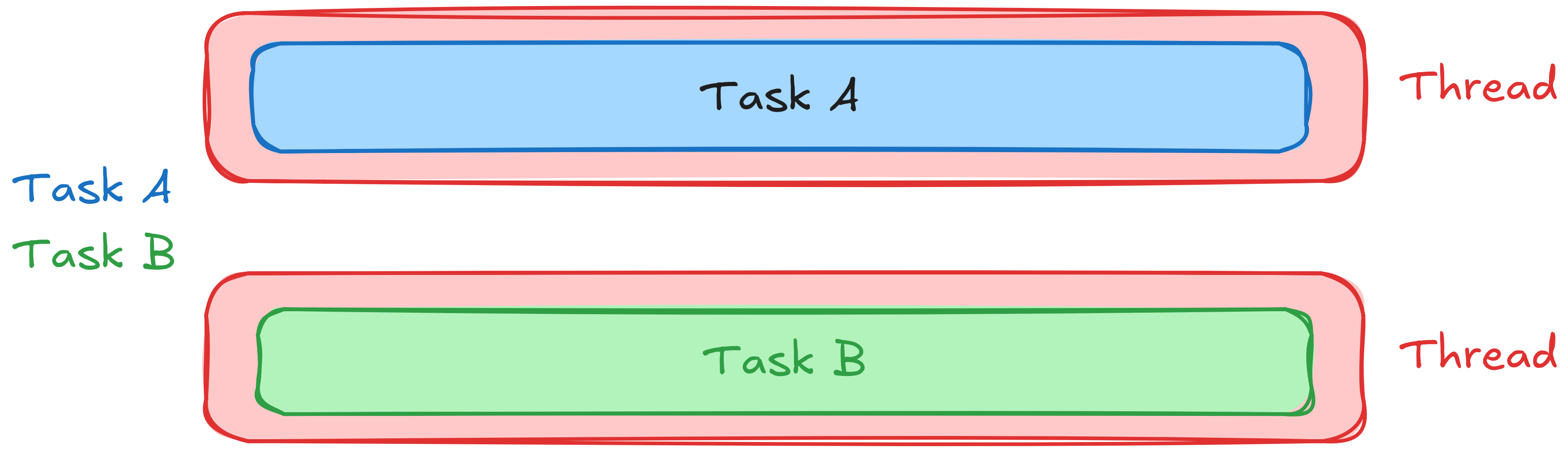
Asynchronous Operations: Non-blocking I/O operations are used in environments like Node.js, where tasks like file reading, network requests, or handling HTTP requests are executed asynchronously, allowing the thread to perform other tasks concurrently.

Concurrency vs Parallelism

Concurrency: Multiple tasks can start, run, and complete in overlapping time periods. The tasks appear to progress at the same time, but they might be rapidly switching in a single core (time-slicing) or managed by an event loop. It's about managing multiple tasks—not necessarily doing them literally at the exact same moment.



Parallelism: Multiple tasks actually run at the exact same time on different CPU cores or machines. Here, the tasks truly execute in parallel, so the system can do more total work simultaneously.



Concurrency is useful in I/O-bound applications like web servers, whereas parallelism is essential for CPU-heavy tasks like video processing or machine learning.

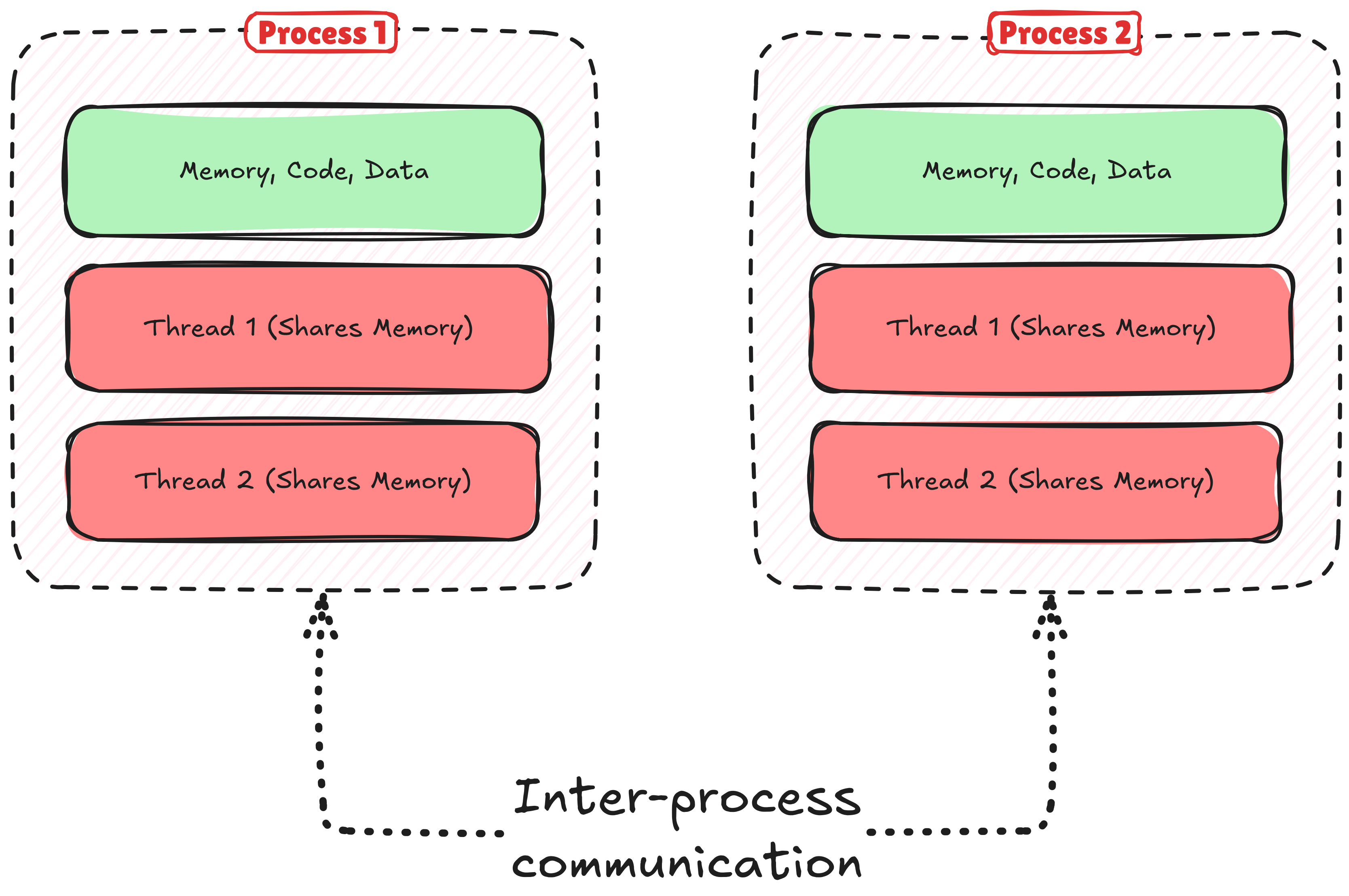
Multiprocessing

Multiprocessing allows a program to use multiple CPU cores to run tasks in parallel. This is especially useful for CPU-heavy tasks like data processing, machine learning, and video editing.

Each process runs independently, with its own memory, which helps avoid issues like race conditions (common in multithreading).

Processes communicate with each other using methods like Inter-Process Communication (IPC), which can include message passing or shared memory.

An example of multiprocessing in Node.js is Node.js Clustering, where multiple processes can handle requests in parallel, improving performance and scalability.



When Node.js uses Multi-Threading

By default, Node.js is single-threaded when it executes JavaScript code. This means that all the JavaScript operations run on a single main thread (called the event loop). For tasks like handling HTTP requests or managing event-driven callbacks, Node.js uses this single thread to process everything in a non-blocking, asynchronous mann

If you have an HTTP server in Node.js, the event loop handles incoming requests one by one without blocking the execution of other requests.

When Node.js uses multi-threading internally :

While JavaScript in Node.js runs on a single thread, Node.js internally uses multiple threads for certain tasks through its libuv thread pool. This allows Node.js to offload specific operations that would otherwise block the main thread, improving performance for I/O-bound tasks.

File System Operations (fs.readFile):

When you read or write files, Node.js uses multiple threads in the thread pool to perform these tasks asynchronously.

DNS Lookups (dns.lookup):

Resolving domain names to IP addresses can be offloaded to other threads, so the main thread remains free to handle other operations.

Cryptography (crypto.pbkdf2, crypto.randomBytes):

Complex cryptographic operations like hashing or generating random values are handled in the thread pool to avoid blocking the main event loop.

Compression (zlib.gzip):

Operations like data compression or decompression are also run in the libuv thread pool, allowing the main thread to focus on other tasks.