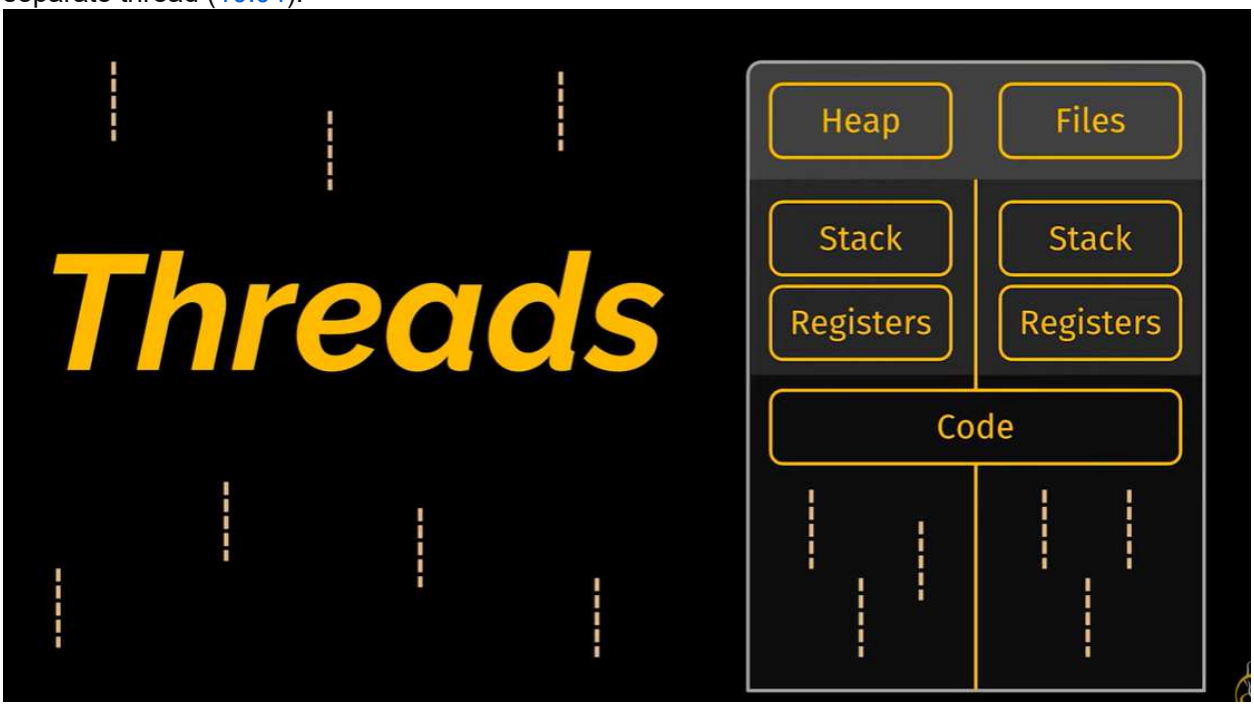
This video discusses the concept of **threading in Processing** to prevent animations from stuttering when loading data from external sources (0:10). When a program needs to fetch data that takes time (an asynchronous call), it can cause the animation to pause or "stutter" if done on the main animation thread (0:50-1:27).
The solution presented is to use **separate threads** for data loading (2:50). The main animation thread can continue running smoothly while the data is fetched in the background. The video demonstrates this by comparing a function that causes a delay in the main `draw()` loop (1:49) versus executing the same function in a separate thread using Processing's `thread()` function (5:47).

Key takeaways include:

- **The Problem:** Slow data loading in the main `draw()` loop causes animations to stutter (1:17).
- **The Solution:** Use **threading** to perform data requests in the background, allowing the main animation thread to continue uninterrupted (2:50).
- **Processing's `thread()` function:** This function allows you to execute a specified function in a separate thread (5:47).
- **Loading status indication:** You can use a global variable to check if a threaded operation is finished and display a loading bar or animation accordingly (7:30).
- **Example data source:** The video suggests `time.json-test.com` as a rapidly changing JSON feed for practicing asynchronous data loading (8:38).
- **Image loading:** For large images, Processing's `requestImage()` function can load the image in a separate thread (10:04).

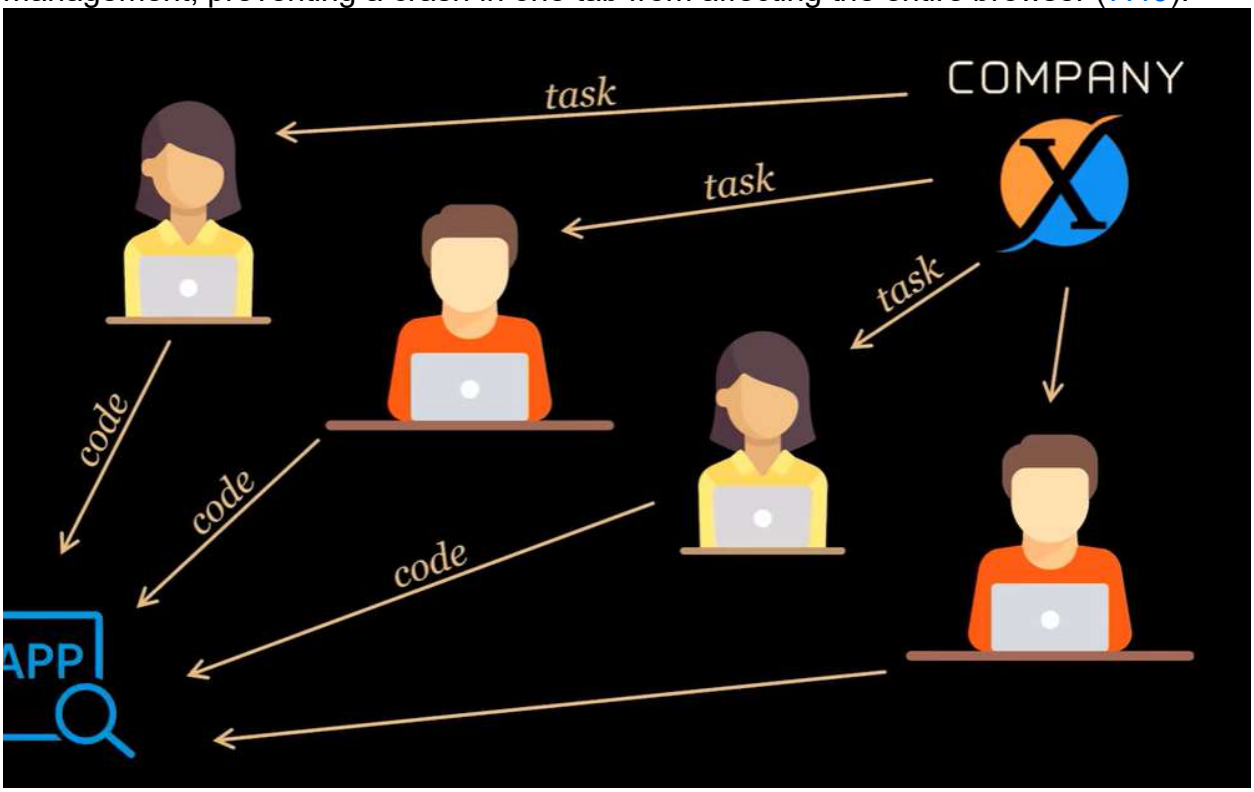



This video explains fundamental concepts in programming, including **threads**, **processes**, **programs**, **parallelism**, and **schedulers** (0:35).

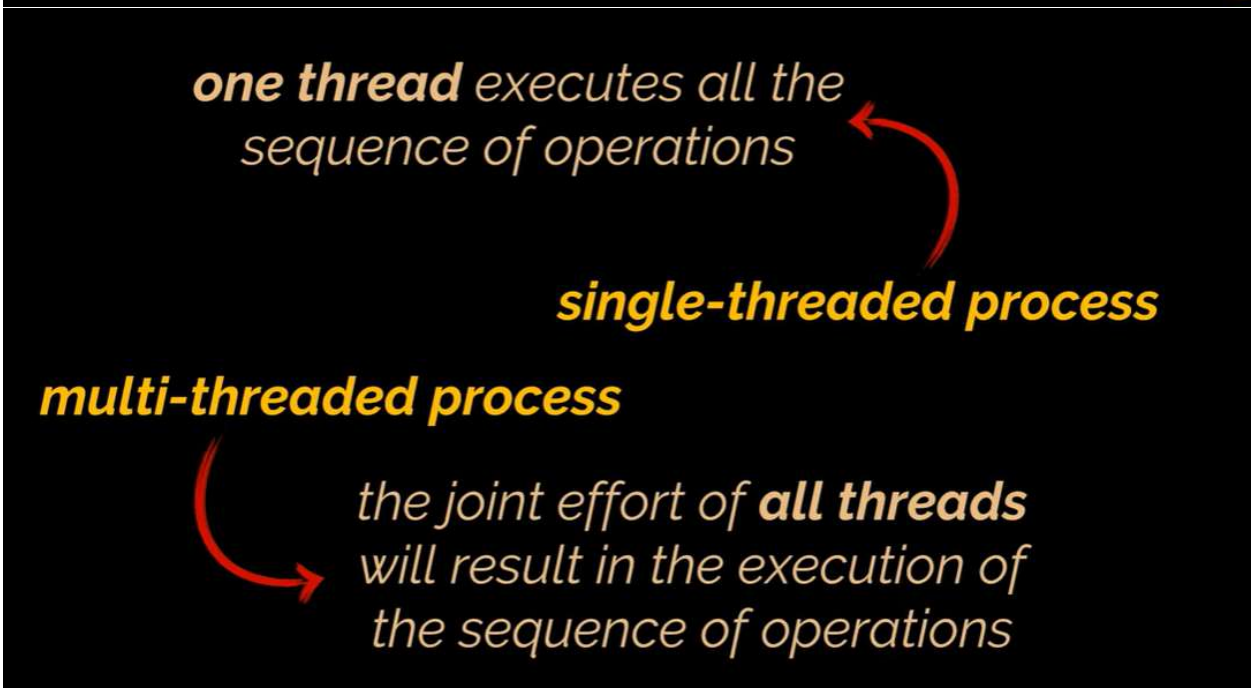
Key takeaways:

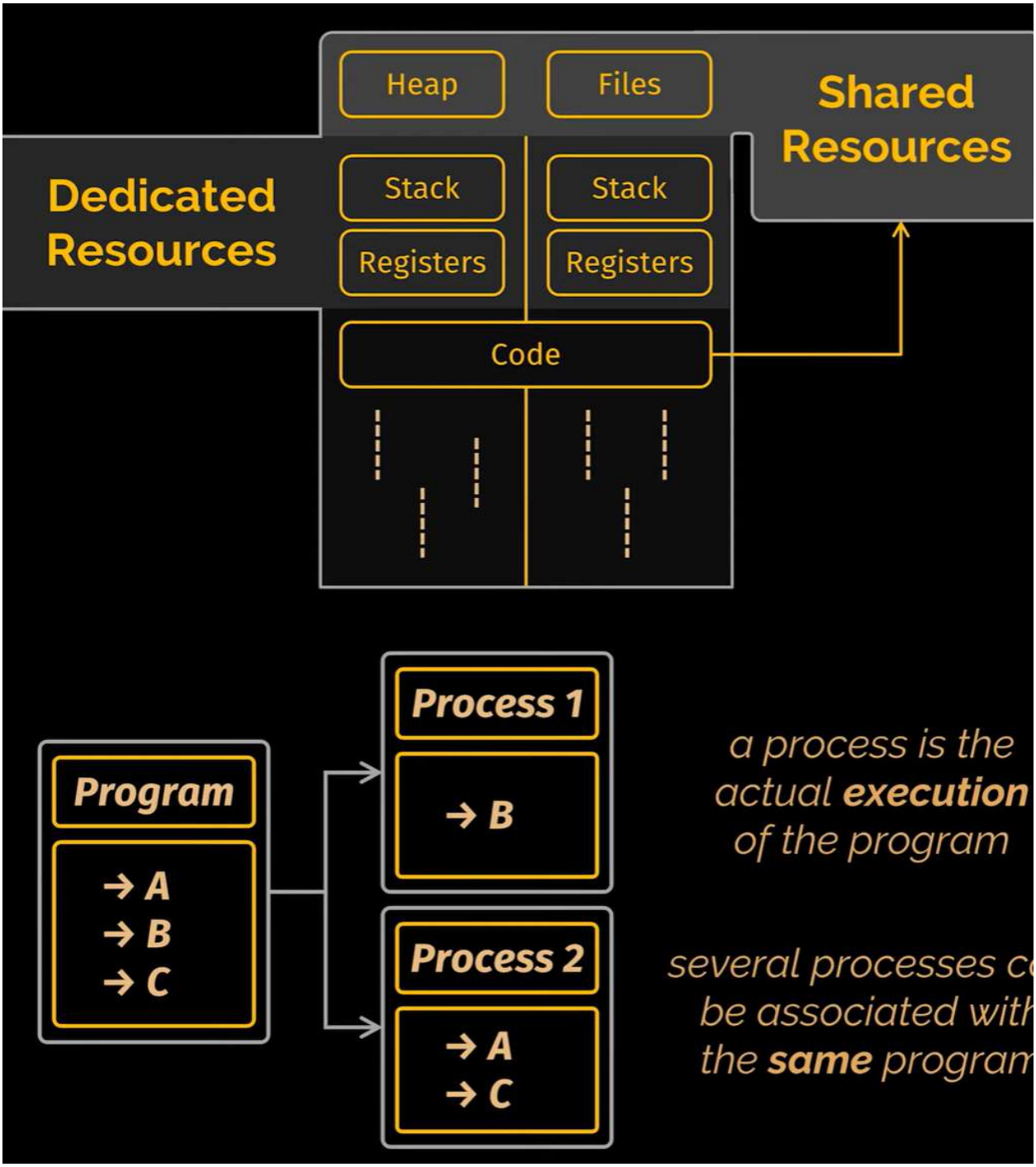
- **Program:** A program is likened to a definition or a blueprint, similar to a class in object-oriented programming. It outlines the instructions that can be performed (3:09).
- **Process:** A process is the actual execution of a program, comparable to an object created from a class. Multiple processes can stem from the same program (3:50).

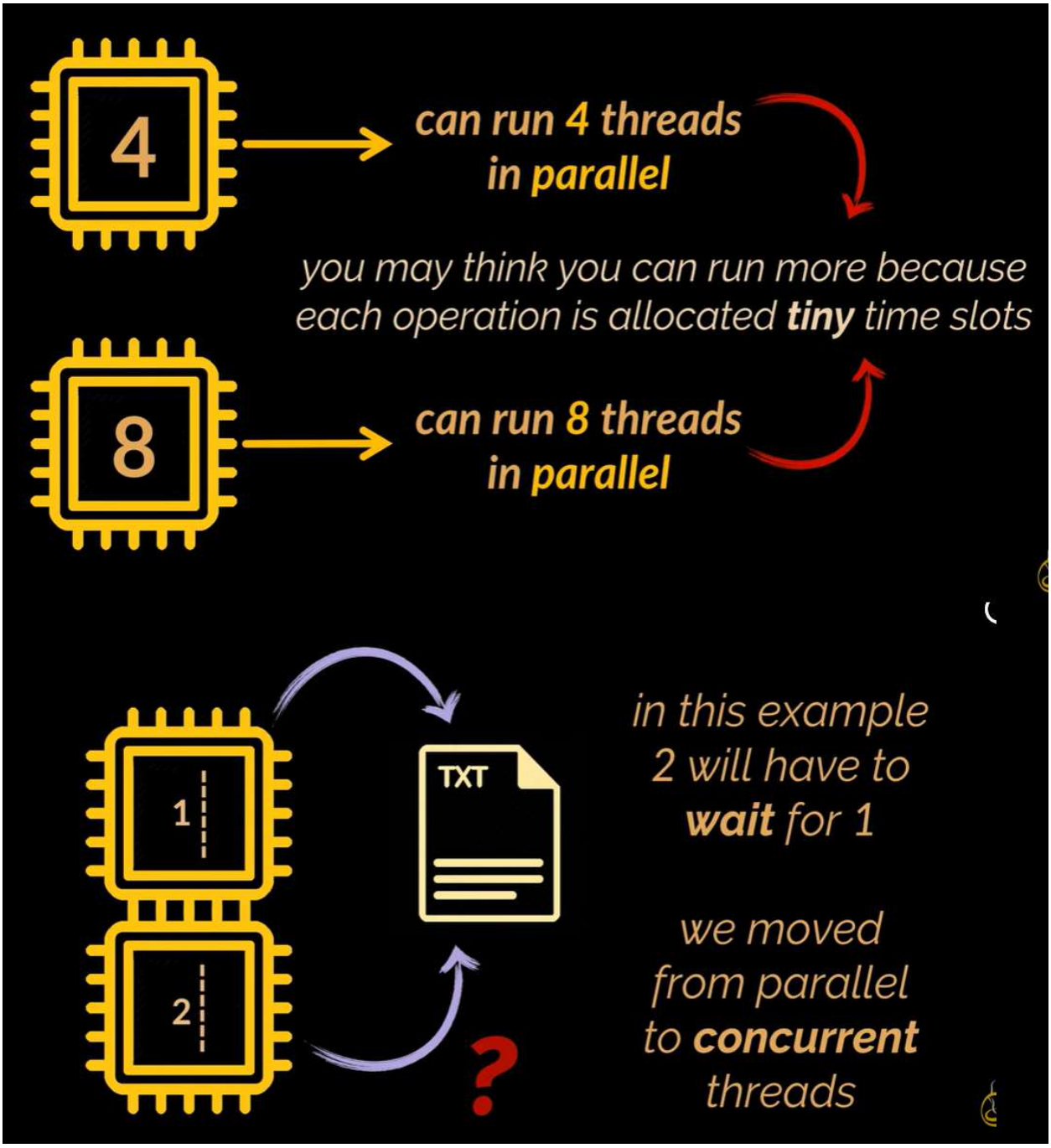
- **Thread:** A thread is the smallest independent unit of instructions that can be executed by a processor or managed by a scheduler (6:51).
- Multiple threads can exist within a single process and share resources like memory (7:04).
- A process must have at least one active thread to exist (7:13).
- **Parallelism:** This refers to the ability to execute multiple tasks simultaneously. While a CPU might have multiple cores (e.g., quad-core, octa-core) allowing for true parallel execution of a limited number of threads, an operating system can create the illusion of parallelism by rapidly switching between a larger number of threads (4:12).
- **Scheduler:** A scheduler is part of the operating system responsible for managing and distributing processor time efficiently among threads, ensuring all computer resources are utilized and tasks are prioritized (6:07).
- **Inter-Thread vs. Inter-Process Communication:**
- Communication between threads within the **same process** is generally faster because they share memory (7:27).
  Communication between **different processes** is slower as they have separate memory addresses (7:38). The video uses Google Chrome as an example of a multi-process design, where each tab runs as a separate process to enhance stability and memory management, preventing a crash in one tab from affecting the entire browser (7:49).

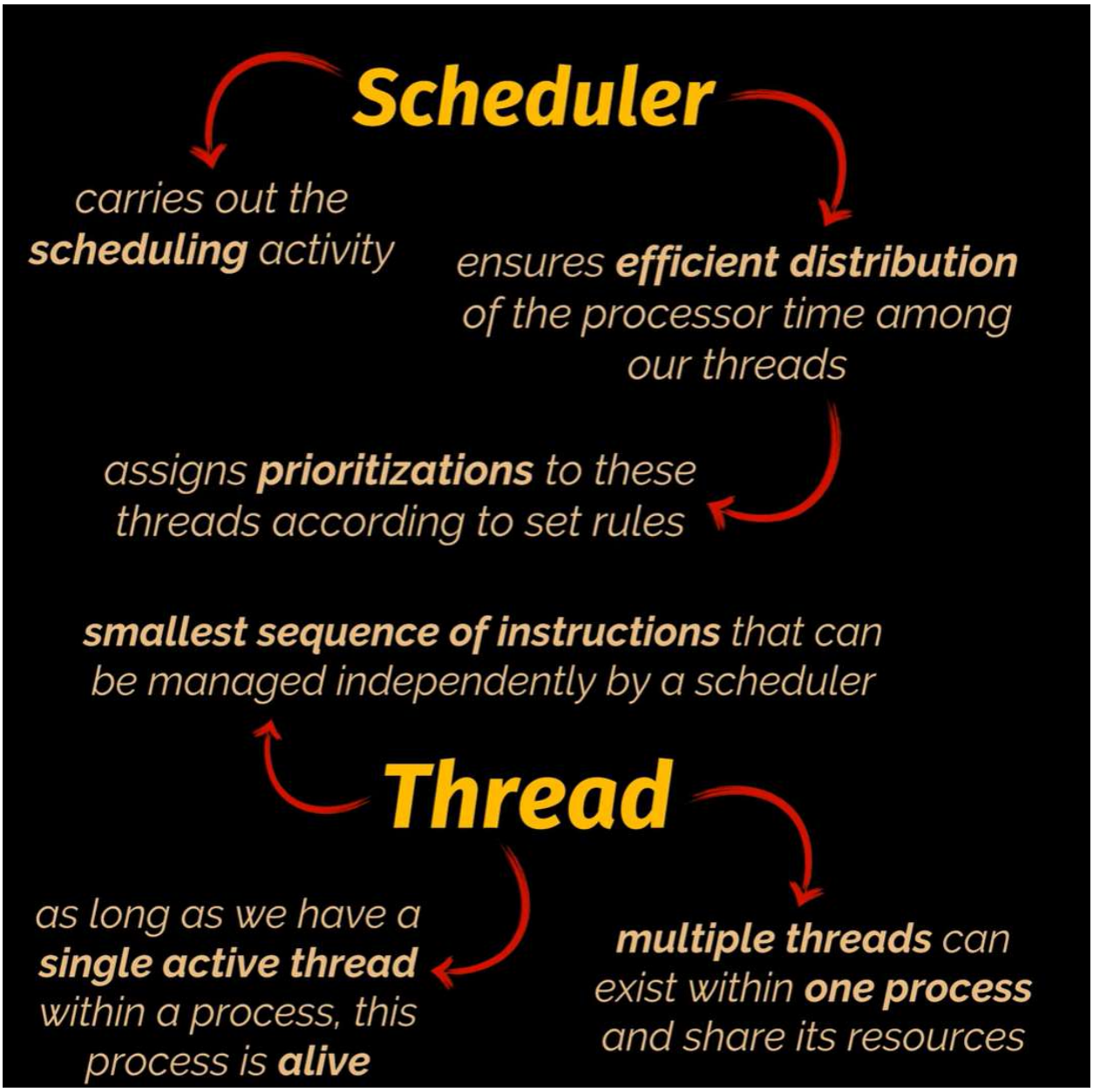


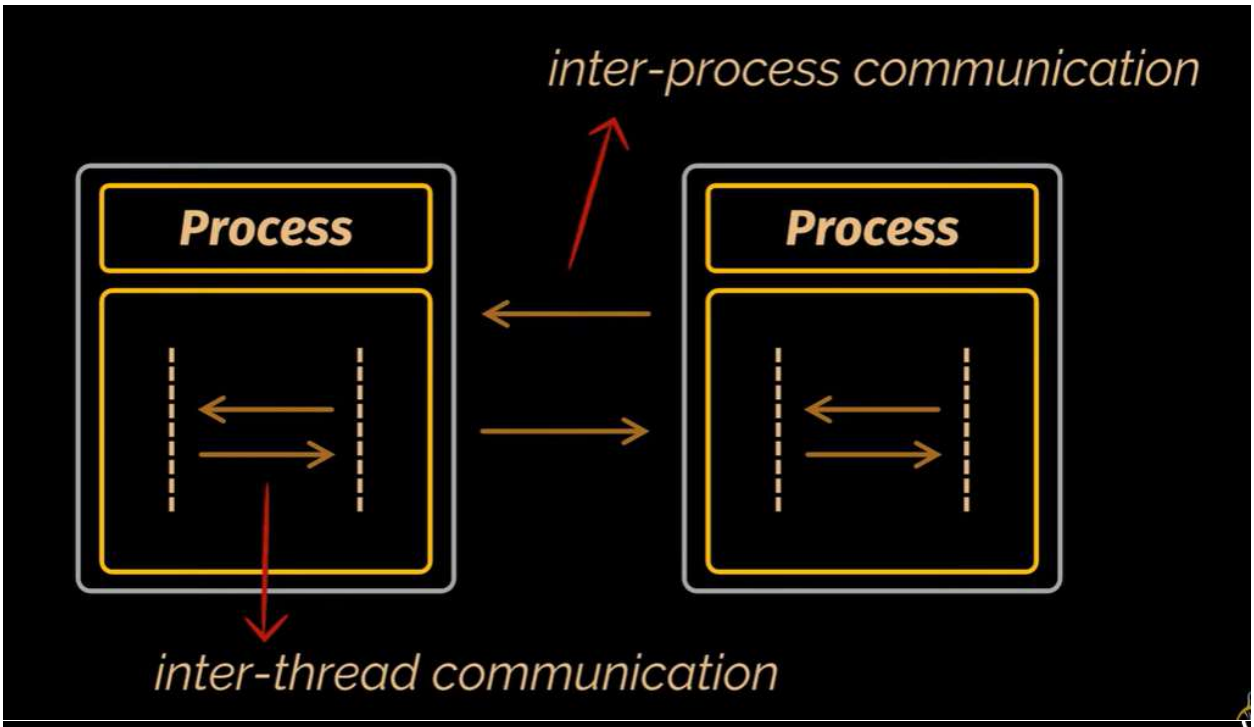

an employee to our company
is like a thread to a process
SMALLEST UNIT

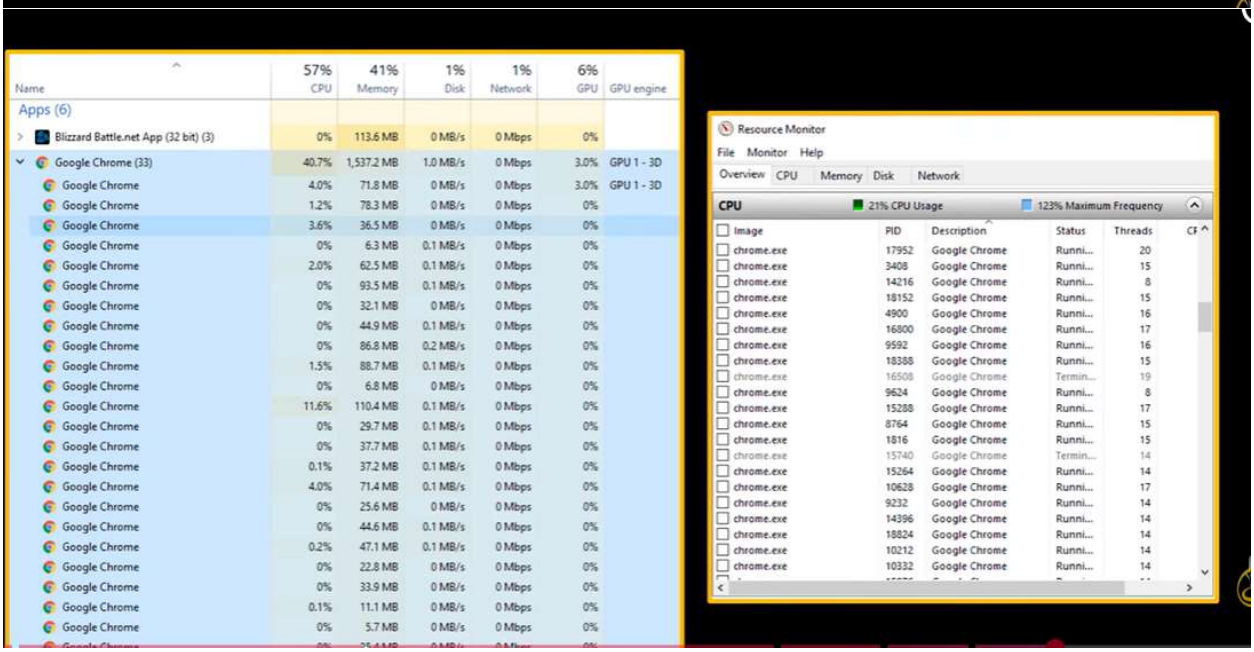

one thread executes all the
sequence of operations
single-threaded process
multi-threaded process
the joint effort of all threads
will result in the execution of
the sequence of operations

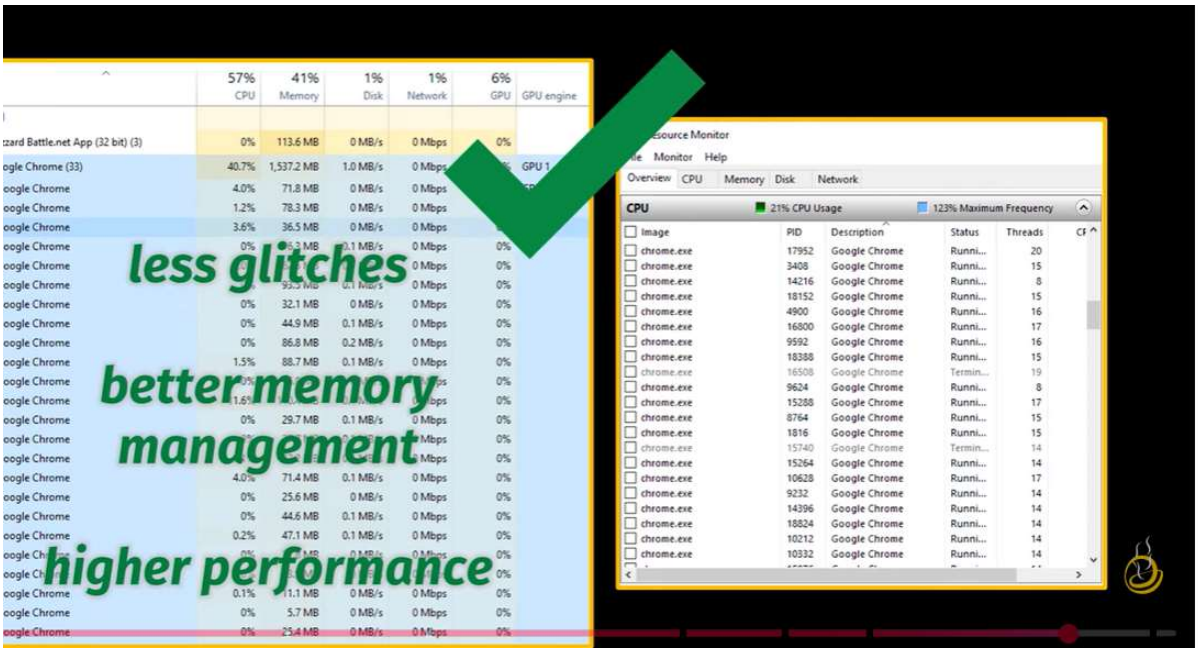

Heap
Files
Shared Resources
Dedicated Resources
Stack
Stack
Registers
Registers
Code
Process 1
→ B
Program
→ A
→ B
→ C
Process 2
→ A
→ C
a process is the actual execution of the program
several processes c
be associated with
the same program


4
can run 4 threads
in parallel
you may think you can run more because
each operation is allocated tiny time slots
8
can run 8 threads
in parallel
1
2
TXT
?
in this example
2 will have to
wait for 1
we moved
from parallel
to concurrent
threads


Scheduler
carries out the scheduling activity
ensures efficient distribution of the processor time among our threads
assigns prioritizations to these threads according to set rules
smallest sequence of instructions that can be managed independently by a scheduler
Thread
as long as we have a single active thread within a process, this process is alive
multiple threads can exist within one process and share its resources


inter-process communication
Process
Process
inter-thread communication


Name
57% CPU
41% Memory
1% Disk
1% Network
6% GPU
GPU engine
Apps (6)
Blizzard Battle.net App (32 bit) (3)
Google Chrome (33)
Resource Monitor
File Monitor Help
Overview CPU Memory Disk Network
CPU
21% CPU Usage
123% Maximum Frequency
Image
PID
Description
Status
Threads




This video discusses **cloud security threats and mitigation strategies** to safeguard data and infrastructure in cloud environments (0:00).

The key threats and their mitigation strategies include:

- **Data Breach (2:10)**: This involves unauthorized access to sensitive customer data.
- **Mitigation Strategies (3:30)**:
- **Data Encryption (3:26)**: Encrypting data at rest and in transit.
- **Access Control (3:50)**: Implementing role-based access control (RBAC) to restrict data access.
- **Regular Auditing (4:21)**: Continuously monitoring user activities and data access to detect breaches early.
- **Account Hijacking (4:47)**: This occurs due to weak passwords or phishing attacks, leading to unauthorized account access.
- **Mitigation Strategies (5:43)**:
- **Multi-factor Authentication (MFA) (5:44)**: Adding an extra layer of security for user logins.
- **Least Privilege Principle (6:02)**: Granting users only the necessary permissions based on their roles.
- **Continuous Monitoring (6:18)**: Using tools like SIEM to monitor system activities for unusual behavior.
- **Insider Threat (6:48)**: Internal employees or third-party vendors intentionally or unintentionally expose data.
- **Mitigation Strategies (7:06)**:
- **User Activity Monitoring (7:07)**: Tracking what users are accessing.
- **Strict Off-boarding Process (7:14)**: Revoking access for employees who leave the company.
- **Data Loss Prevention (DLP) Policies (7:26)**: Preventing unauthorized data transfer outside the company.
- **Insecure APIs (7:56)**: Vulnerable Application Programming Interfaces can be exploited to gain unauthorized access to cloud services.
- **Mitigation Strategies (8:26)**:
- **API Authentication (8:27)**: Ensuring proper authentication for API access.
- **Rate Limiting (8:33)**: Controlling the number of API requests a user can make.
- **Regular API Testing (8:39)**: Regularly testing APIs for security vulnerabilities.
- **Distributed Denial of Service (DDoS) Attacks (8:47)**: Overwhelming a system with traffic to make it unavailable.

- **Mitigation Strategies (9:33)**:
- **DDoS Protection Services (9:33)**: Utilizing services like AWS Shield, Cloudflare, or Azure DDoS Protection.
- **Firewalls (9:42)**: Implementing proper firewall rules.
- **Load Balancers (9:42)**: Distributing traffic to maintain service availability.
- **Auto-scaling (9:51)**: Automatically adjusting resources based on demand.
- **Lack of Cloud Security Monitoring (11:13)**: Inadequate monitoring and auditing can lead to delayed detection of security incidents.
- **Mitigation Strategies (10:54)**: Early and continuous monitoring and auditing are crucial to detect and address issues before they escalate.
- 

### 3. Insider Threats

**Real-Life Example:**

**Amazon Web Services (AWS) Employee Leak (2020)** – A former AWS employee misused customer data by exploiting internal access to cloud services.

**Mitigation Strategies:**

- **User Activity Monitoring**: Log and monitor all user actions using **CloudTrail (AWS)** or **Azure Monitor**.
- **Strict Offboarding Process**: Revoke access immediately when employees leave.
- **Data Loss Prevention (DLP)**: Implement DLP policies to prevent unauthorized data transfers.

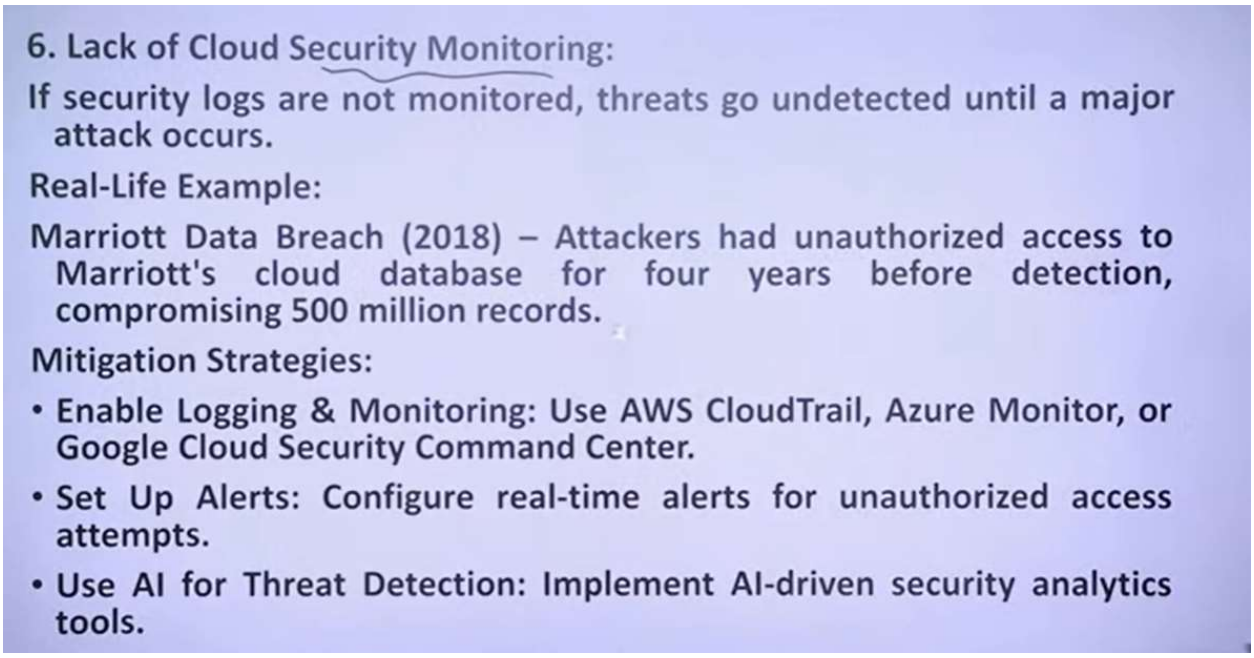
### 5. Distributed Denial of Service (DDoS) Attacks

**Real-Life Example:**

**GitHub DDoS Attack (2018) – A record-breaking 1.3 Tbps DDoS attack targeted GitHub, disrupting services for 15-20 minutes.**

**Mitigation Strategies:**

- **DDoS Protection Services: Use AWS Shield, Cloudflare, or Azure DDoS Protection.**
- **Traffic Filtering: Deploy firewalls and load balancers to block malicious traffic.**
- **Auto-Scaling: Set up auto-scaling in cloud infrastructure to handle high traffic loads.**




Here's a **complete, in-depth explanation of Threads** in the context of **Parallel and Distributed Computing**, with **examples, advantages, types, and applications**.

---

# Threads in Parallel & Distributed Computing

## 1. Definition of Thread

A **thread** is the **smallest unit of execution** within a process.
A **process** can contain **one or more threads**, and all threads within a process share the same memory and resources.

### Exam-ready one-line definition:

**A thread is a lightweight, independent path of execution within a process that allows multiple operations to run concurrently.**

---

## 2. Why Threads are Important

- **Parallel Execution:** Run multiple tasks at the same time on different CPU cores.
- **Resource Sharing:** Threads within a process share memory and data.

- **Efficiency:** Less overhead than creating multiple processes.
- **Responsiveness:** GUI apps remain responsive while background tasks run.

---

# 3. Process vs Thread

| Feature | Process | Thread |
|---|---|---|
| Memory | Separate memory space | Shared memory within process |
| Overhead | High (process creation) | Low (thread creation) |
| Communication | Inter-process communication (IPC) needed | Direct memory access |
| Execution | Independent | Can run concurrently within a process |
| Example | Chrome browser process | Each tab has multiple threads |

---

# 4. Thread Lifecycle

A thread goes through several states:

1. **New / Created:** Thread object created but not started.
2. **Runnable / Ready:** Thread ready to run, waiting for CPU.
3. **Running:** Thread is executing.
4. **Waiting / Blocked:** Waiting for resource or event.
5. **Terminated / Dead:** Thread has finished execution.

---

# 5. Types of Threads

## 5.1 User-Level Threads

- Managed by **user-level libraries**
- OS unaware of them
- Fast context switch
- Example: Java green threads (older versions)

## 5.2 Kernel-Level Threads

- Managed by **operating system**
- OS schedules threads individually
- Example: POSIX Threads (Pthreads), Windows threads

### 5.3 Hybrid Threads

- Combination of user and kernel threads
- Example: Many modern OS implementations

---

# 6. Advantages of Threads

- **Parallelism:** Run multiple tasks simultaneously
- **Efficiency:** Less CPU and memory overhead than processes
- **Responsiveness:** GUI apps remain interactive
- **Resource sharing:** Threads share process memory
- **Scalability:** Can utilize multi-core CPUs

---

# 7. Disadvantages / Challenges

- **Race conditions:** Threads accessing shared data may conflict
- **Deadlocks:** Two or more threads waiting indefinitely
- **Debugging complexity**
- **Thread synchronization required**

---

# 8. Thread Synchronization

Since threads share memory, **synchronization** is needed to avoid conflicts.

### Mechanisms

- **Mutex / Lock:** Allows only one thread to access a resource.
- **Semaphore:** Controls number of threads accessing a resource.
- **Monitor / Critical Section:** Protects shared resources.
- **Barrier / Join:** Wait for threads to finish.

---

# 9. Thread Example (Parallel Computing)

### Example in Python (Multithreading)

```
import threading
```

```python
def print_numbers():
    for i in range(5):
        print(f"Number: {i}")

def print_letters():
    for c in ['A','B','C','D','E']:
        print(f"Letter: {c}")

# Create threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)

# Start threads
t1.start()
t2.start()

# Wait for threads to finish
t1.join()
t2.join()

print("Finished execution")
```

**Explanation:**

- `t1` and `t2` run concurrently.
- CPU switches between threads.
- Output may interleave numbers and letters.

---

# 10. Thread Example (C / Pthreads)

```c
#include <stdio.h>
#include <pthread.h>

void* print_message(void* arg){
    char* msg = (char*) arg;
    printf("%s\n", msg);
    return NULL;
}

int main(){
    pthread_t thread1, thread2;

    pthread_create(&thread1, NULL, print_message, "Hello from Thread 1");
    pthread_create(&thread2, NULL, print_message, "Hello from Thread 2");

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

---

# 11. Threads in Distributed Systems

- In **distributed computing**, threads can run tasks on **local nodes**, while processes may run on **remote nodes**.
- Example:
  - Master node distributes tasks to worker threads on multiple compute nodes.
  - Threads execute tasks in parallel and return results.

---

# 12. Parallel Computing Example with Threads

- Problem: Sum of an array with 4 threads

```
Array: [1,2,3,4,5,6,7,8]

Thread 1 → sum(1,2)
Thread 2 → sum(3,4)
Thread 3 → sum(5,6)
Thread 4 → sum(7,8)

Main thread → combine all partial sums → total sum
```

- Threads run **simultaneously**, total time ≈ max(time of one thread).

---

# 13. Applications of Threads

- Web servers (handle multiple requests)
- Real-time systems
- Parallel data processing
- Games & GUI applications
- Scientific simulations

---

# 14. Threads vs Processes vs Distributed Nodes

| Feature | Thread | Process | Distributed Node |
|---|---|---|---|
| Memory | Shared | Separate | Separate machines |
| Communication | Shared memory | IPC | Network messages |
| Overhead | Low | Medium | High |

| Feature | Thread | Process | Distributed Node |
|---|---|---|---|
| Parallelism | Within process | Across processes | Across machines |
| Example | Multi-core computation | Multi-app execution | Cloud / Grid computing |

---

# 15. Advantages in Parallel & Distributed Computing

- Improves **CPU utilization**
- Reduces **response time**
- Facilitates **task decomposition**
- Works with **multi-core systems**
- Forms the base for **distributed task execution** in clusters

---

# 16. Exam-Oriented Summary

### Keywords:

- Thread, lightweight process
- Concurrency
- Synchronization (mutex, semaphore)
- Parallel execution
- Multi-core, HPC

### One-Paragraph Answer:

A thread is the smallest unit of execution within a process that allows multiple tasks to run concurrently. Threads share the same memory space of the process, enabling efficient parallelism. They are widely used in parallel computing for task decomposition, in distributed systems for executing multiple tasks on local nodes, and require synchronization mechanisms like mutexes or semaphores to prevent conflicts.

---

If you want, I can next create a **diagram showing Threads vs Processes vs Distributed Nodes** with **parallel & distributed computing examples**—very helpful for exams.

Do you want me to do that?