# Process:

A process is an independent program in execution, with its own address space, code, data, and system resources. It's managed by the operating system, which assigns it a Process ID (PID) and isolates it from other processes to ensure memory safety and stability

# Thread:

A thread is the smallest unit of execution within a process. Threads share the same memory space but run independently, enabling concurrent execution, Shares memory with other threads in the same process but has its own stack memory and registers

# Multi-threading:

Multipull threads run under a single process is called as multi-threading. They share memory of Code Segment, Heap Memory, Global/Static Varibles and File Descripter and have seprate Stack Merory & Registers, execute independently and This enables efficient communication and resource sharing compared to multiprocessing

# Supporting Languages

C/C++: Provides low-level threading capabilities through libraries POSIX threads(pthreads)

Python: Offers by the threading module, It provides a higher‑level, object‑oriented interface over the lower‑level \_thread module

Java: Java treats every concurrent task as a Thread (or indirectly via Runnable/Callable),

# Use of Multithreading

Parallel Processing

Faster Execution on Multi-core Systems

Efficient Resource Utilization

Speed up I/O-bound programs (e.g., file reading, network requests)

Improve responsiveness

# Applications

Web servers (e.g., handling multiple client requests simultaneously)

Video games (e.g., separate threads for rendering, physics, and input)

Download managers (e.g., splitting files into parts to download in parallel)

Banking systems (e.g., processing concurrent transactions safely)

Machine learning/data processing (e.g., processing data batches in parallel)

# Creation of thread

To create a thread 1st insilize a thread variable,

To create thread use pthread\_create() function

syntax: pthread\_create(& thread variable name, NULL, function, function orguments);

Function syntax:

void\* fun\_name(void\* args)

-> By using void pointer it can accept any type of pointer as argument and return void pointer,

Exaple code:

#include <stdio.h>

#include <pthread.h>

int counter = 0;

void\* increment(void\* arg) {

for (int i = 0; i < 5; i++) {

counter++;

printf("Thread %d: Counter = %d\n", \*(int\*)arg, counter);

}

return NULL;

}

int main() {

pthread\_t t1;

int id1 = 1;

pthread\_create(&t1, NULL, increment, &id1);

pthread\_join(t1, NULL);

printf("Final Counter Value: %d\n", counter);

return 0;

}

# Exeguation

when a thread function call then a thread will create and exeguate indipendend and create memory at stack area. In threads if main thread's exeguation complate the remaining threads in the code will terminate even if exeguation pending of threads, because if main thread exeguation complete it says that the whole application exeguation is completed due to this the stact area will clean, thare fore threads stack area also force to clean,

To wait the main thread until the remaing threads completion, we have pthread\_join() pthread\_join() will make the main thread to

wait until the menction thread competion,

syntax: pthread\_join(pthread\_t thread, void \*\*return)

Pthread\_join() contains 2 arguments

- pthread\_t thread: The identifier of the thread to wait for.

- void \*\*retval: A pointer to a location where the exit status of the thread will be stored (can be NULL if not needed)

# Considerations Issues

1. Global data corruption: When two or more threads access and modify global data at the same time, and the access is not controlled properly (no lock, mutex, etc.), the data can get corrupted

2. Race conditions: A race condition occurs when two or more threads (or processes) access shared data at the same time, and the final result depends on the timing or order of execution.

Because the threads are "racing" to read or write the same data, the outcome becomes unpredictable and may lead to bugs, crashes, or corrupted data

3. Deadlocks: A deadlock occurs when two or more threads (or processes) are waiting for each other to release resources, and none of them can proceed.

4. LiveLock: A livelock is a situation in concurrent programming where two or more threads keep changing their state in response to each other, but fail to make any actual progress. Unlike a deadlock, where threads are stuck waiting, in a livelock, threads are actively running, but constantly yielding or retrying, which prevents them from completing their tasks.

# Synchronization and locking mechanism

1. Mutexes: A mutex, short for mutual exclusion, is a synchronization primitive used to protect shared resources in a multithreaded environment. It ensures that only one thread at a time can access a critical section of code.

When a thread locks a mutex, any other thread that tries to lock it will be blocked until the mutex is released. This prevents race conditions and data corruption when multiple threads interact with shared data

2. Semaphores: A semaphore is a synchronization tool used to control access to a shared resource by multiple threads. It works using a counter that represents the number of available resources or permits.

3. Condition Variable: A Condition Variable is a synchronization primitive that allows threads to wait for certain conditions to become true, while releasing the associated lock during the wait.

It’s commonly used when a thread needs to pause until a specific condition is met, such as waiting for a shared resource to become available or for a queue to be non-empty.

4. Spinlocks: A spinlock is a type of lock used in multithreading to protect shared resources. Unlike regular locks where a thread might go to sleep if it can't acquire the lock, a spinlock “spins” in a loop, continuously checking if the lock is available.

This means the thread stays active and consumes CPU cycles while waiting. Spinlocks are generally used in low-level systems programming or in situations where the lock is expected to be held for a very short time, because avoiding context switching can make things faster.

However, they can be inefficient on single-core systems or when the lock is held too long, as they waste CPU time by busy-waiting.

5. TryLock: A TryLock is a non-blocking version of a lock. When a thread attempts to acquire a lock using TryLock, it tries once and immediately returns with either success or failure, instead of waiting if the lock is already held by another thread.

This is useful in situations where a thread should not be blocked and instead wants to perform other tasks or retry later if the resource is not immediately available.