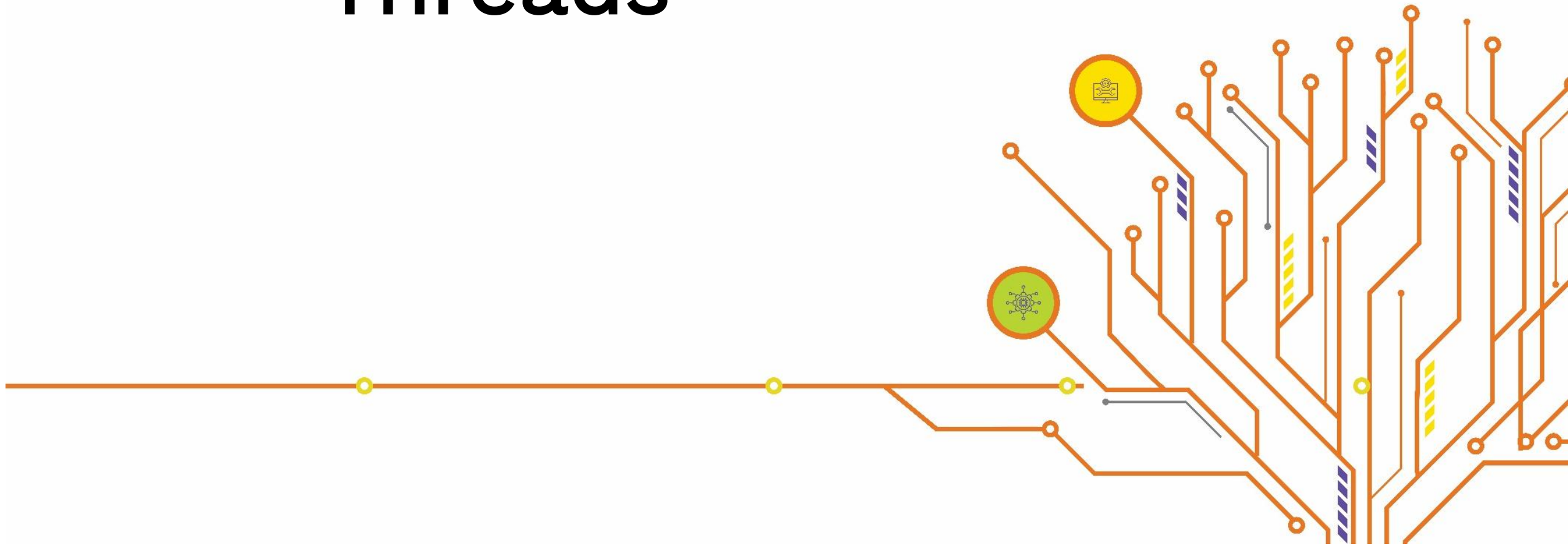
KPIT | APEX LAB

Threads



What is a Thread?

A thread is a lightweight execution unit inside a process

Multiple Threads share:

- Address space
- File descriptors
- Heap
- Global

Threads have their own:

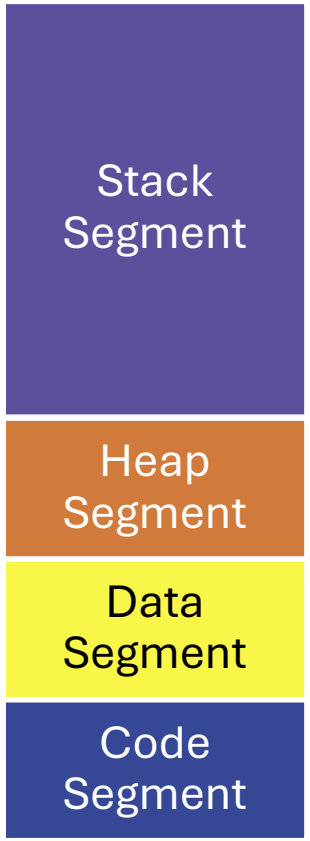
- Stack
- Registers
- thread ID

When to Use Threads

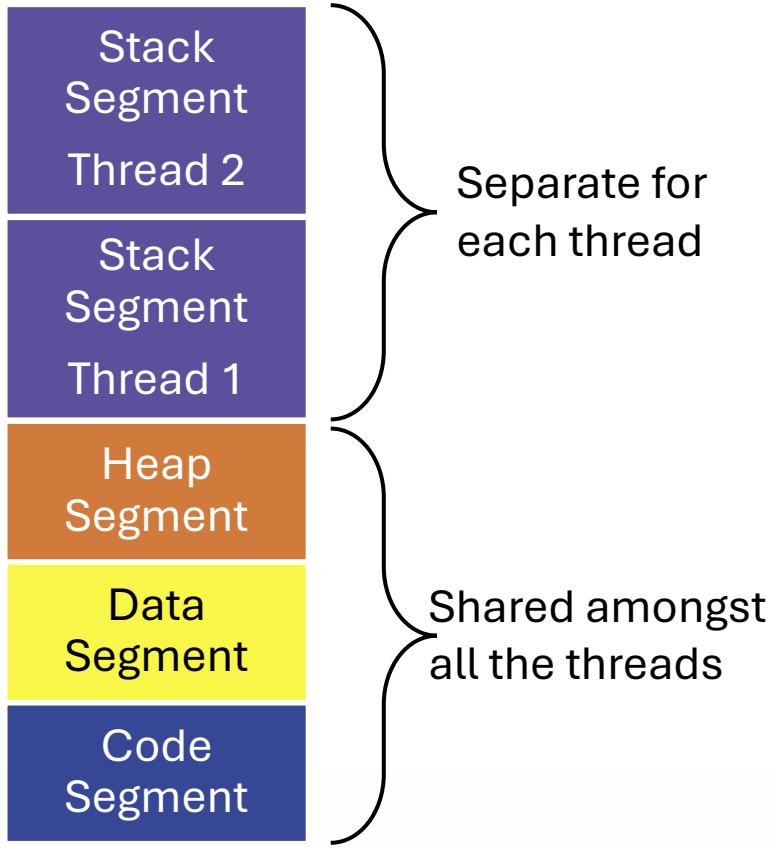
- Concurrent I/O (e.g., sensors, logging)
- Background tasks (telemetry, watchdog pings)
- Producer/consumer pipelines (ring buffers)
- Real-time task separation (with priorities & policies)

Process – Thread Memory

Each process has a separate memory allocated.



A process can have multiple threads inside it.



Core pthread APIs

Syntax	Parameters	Return & Notes
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)	thread: out param; attr: NULL for defaults; start_routine: function; arg: user data	0 success; non-zero error (e.g., EAGAIN). New thread starts at start_routine.
int pthread_join(pthread_t thread, void **retval)	thread: target; retval: receives return from pthread_exit() or function return	Blocks until thread terminates. ESRCH/EINVAL/EDEADLK possible.
void pthread_exit(void *retval)	retval: return pointer for joiner	Does not return. Cleans up thread resources properly.
int pthread_detach(pthread_t thread)	Make thread unjoinable; resources auto-released at exit	Use for fire-and-forget workers. Don’t join() after detach.

Thread ID

- Each thread within a process is uniquely identify by a thread ID.
- This thread ID is returned by the caller of pthread_create()
- A thread can obtain its thread id by using pthread_self()

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Creating A simple Threads

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

void* incrementFunction(){
for (int i = 0; i < 5; i++) {
        printf("Thread: i = %d\n", i); sleep(1);
}
return NULL; // Thread ends
}

void* decrementFunction(){
for (int j = 5; j > 0; j--) {
        printf("Thread: j = %d\n", j); sleep(1);
}
return NULL; // Thread ends
}
```

```c
int main(void){
        pthread_t thread_id1, thread_id2;

        pthread_create(&thread_id1,NULL,
        incrementFunction,NULL);
        pthread_create(&thread_id2,NULL,
        decrementFunction,NULL);

        pthread_join(thread_id1,NULL);
        pthread_join(thread_id2,NULL);

}
```

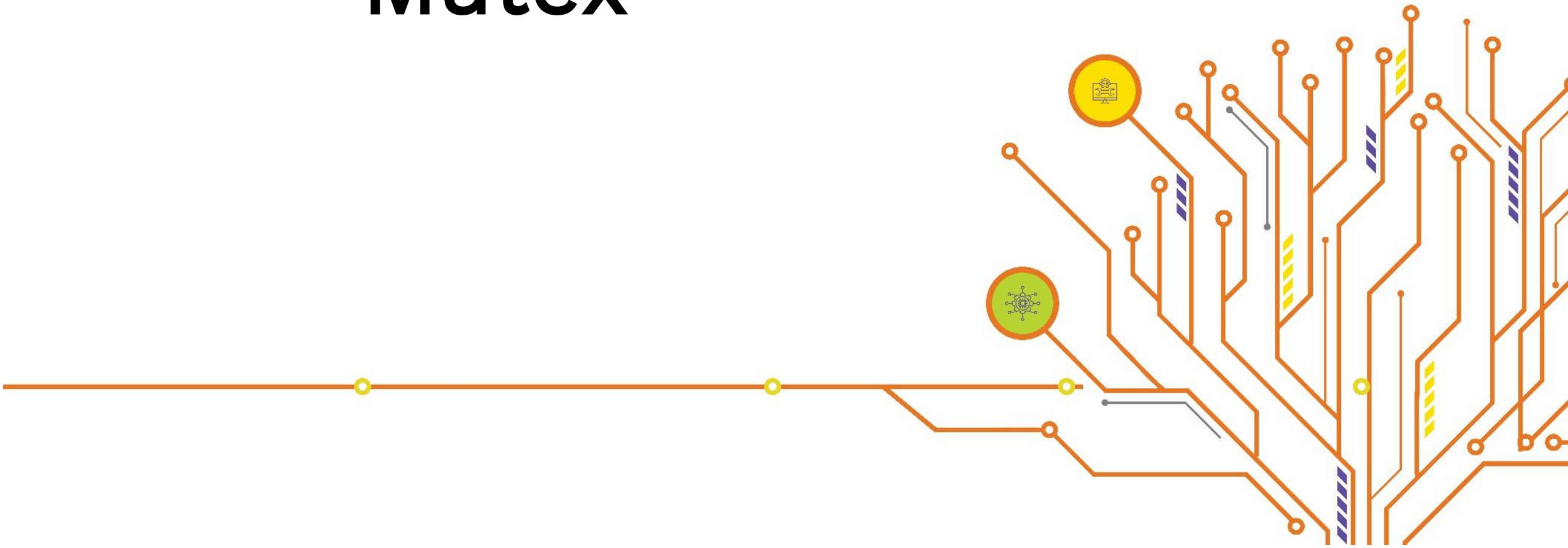
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *thread_func_T2(void *arg) {
        printf("T2: Starting work...\n");
        sleep(2);
        printf("T2: Work completed.\n");
        return (void *)"T2 done";
}

void *thread_func_T1(void *arg) {
        pthread_t *t2 = (pthread_t *)arg;
        void *retval;
        printf("T1: Waiting for T2 to finish...\n");
        pthread_join(*t2, &retval);
        printf("T1: T2 finished with message: %s\n",(char *)retval);
        printf("T1: Doing its own work...\n");
        sleep(1);
        printf("T1: Work completed.\n");
        return (void *)"T1 done";
}

void *thread_func_generic(void *arg) {
        int id = *((int *)arg);
        printf("T%d: Doing work...\n", id);
        sleep(1);
        printf("T%d: Work completed.\n", id);
        return (void *)"Generic done";
}
```

```c
int main() {
        pthread_t t1, t2, t3, t4;
        int id3 = 3, id4 = 4;
        void *retval;

        // Create T2 first
        if (pthread_create(&t2, NULL, thread_func_T2, NULL) != 0)
                {perror("Failed to create T2"); exit(EXIT_FAILURE);}

        // Create T1 and pass T2's ID to it
        if (pthread_create(&t1, NULL, thread_func_T1, &t2) != 0)
                {perror("Failed to create T1"); exit(EXIT_FAILURE);}

        // Create T3 and T4
        pthread_create(&t3, NULL, thread_func_generic, &id3);
        pthread_create(&t4, NULL, thread_func_generic, &id4);

        // Main thread joins T1, T3, and T4
        pthread_join(t1, &retval);
        printf("Main: T1 finished with message: %s\n", (char *)retval);

        pthread_join(t3, &retval);
        pthread_join(t4, &retval);

        printf("Main: All threads completed. Exiting.\n");
        return 0;
}
```

Aspect	Process	Thread
Basic Definition	A process is an independent program in execution with its own memory space.	A thread is a lightweight unit of execution within a process.
Memory Space	Has its own separate virtual memory space.	Shares the process's memory space (text, data, heap).
Stack	Each process has its own independent stack.	Each thread has its own stack within the same process.
Creation Overhead	High – memory allocation, page tables, resources must be created.	Low – shares most resources, only stack + thread control block created.
Context Switch Cost	Higher – switching memory context and kernel structures.	Lower – same memory space, lighter task switch.
Communication	Requires IPC mechanisms (pipes, shared memory, message queues).	Very fast – communicates via shared memory within the process.
Isolation	Strong isolation – one process crash does not affect others.	Weak isolation – thread crash can terminate the entire process.
Resource Sharing	Not shared by default; must use explicit IPC.	Threads share code, data, heap, file descriptors.
Scheduling Unit in Linux	The process's main thread is scheduled as a task.	Each thread is scheduled as an individual task.
Concurrency	Limited by heavier switch + isolation boundaries.	High concurrency with lightweight switching.
Security	More secure due to isolation.	Less secure because shared address space.
Suitable For	Large independent applications, services, fault isolation.	Parallel tasks within the same app, real-time workflows, worker pools.
Linux Representation	Represented by a task_struct with unique PID.	Represented by task_struct with unique TID but same thread group ID.
Creation APIs	fork(), exec()	pthread_create() or clone() with flags.
Failure Impact	Process crash usually does not affect other processes.	Thread crash can bring down the entire process.
Typical Use in Embedded Linux	Running separate services, daemons, system processes.	Real-time tasks, communication handlers, worker threads, parallel computations.

KPIT | APEX LAB

Mutex



Why Synchronization Is Required

When multiple threads run inside the same process, they share memory:

- Global variables
- Static variables
- Heap memory
- File descriptors
- Peripheral buffers (UART, CAN, SPI)

Because of this, two or more threads may try to access or modify the same shared data simultaneously.

Typical problems that occur without synchronization

Race Conditions	Two threads modifying the same variable at the same time → unpredictable output.
Data Corruption	Concurrent writes to shared buffers (e.g., ring buffers, sensor logs) can corrupt data.
Inconsistent State	Thread A updates part of a structure while Thread B reads it → B gets half-updated data.
Deadlocks & Priority Inversion	Critical in embedded and real-time systems.

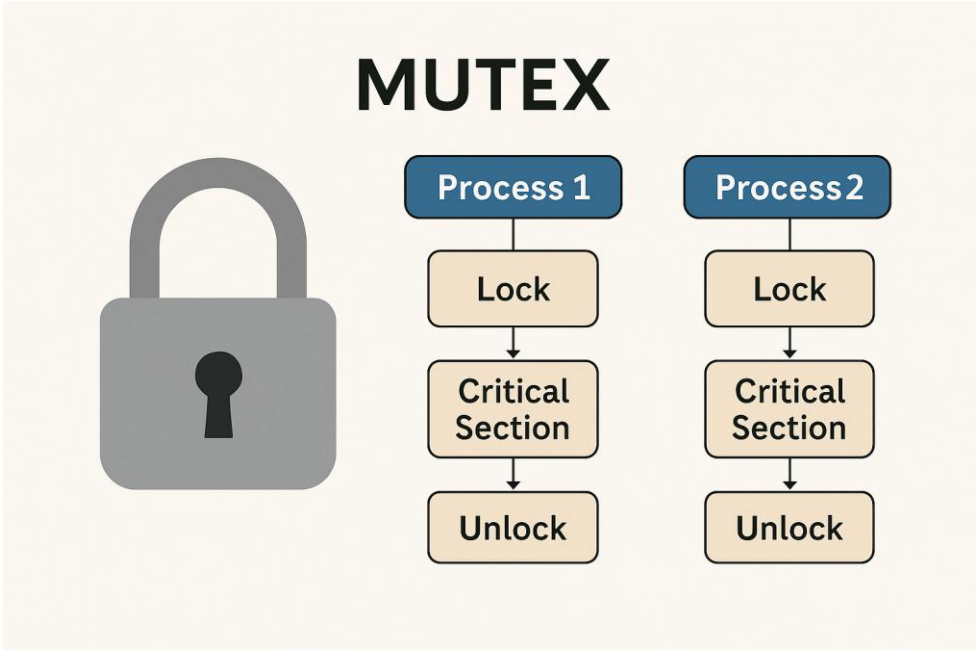
To prevent these problems, we need a mechanism that ensures only one thread at a time can access a shared resource → MUTEX

Mutex

A mutex (**MUT**ual **EX**clusion object) is a synchronization primitive that allows only one thread to own or lock a resource at a time.

It acts like:

- A key that only one thread can hold.
- Others must wait until the thread releases it.



How a Mutex Works

- State of a Mutex

Basic Operations

1.pthread_mutex_lock()

1. If unlocked → lock it and continue.
2. If locked → thread goes into **blocked state**.

2.pthread_mutex_unlock()

1. Releases mutex so another waiting thread can acquire it.

3.pthread_mutex_trylock()

1. Attempts lock without blocking.

When to Use a Mutex

Shared data must be modified	Updating a global counter Writing to shared sensor buffers Modifying linked lists, queues, ring buffers Accessing shared hardware (I2C/SPI/UART write sequences)
Only one thread must access a critical section at a time	• Logging to a file • Driving a motor control output • Updating PWM values • Shared communication stack (TCP/IP, CAN, UART Rx/Tx queues)
Protecting non-atomic operations	Even simple operations like counter++ need mutex protection if accessed by multiple threads.

Sample Example - Mutex

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

int shared_counter = 0;
pthread_mutex_t lock;

void* worker(void *arg)
{
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
        shared_counter++;    // safe update
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

```
int main()
{
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);

    pthread_create(&t1, NULL, worker, NULL);
    pthread_create(&t2, NULL, worker, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);


    printf("Final counter value: %d\n",
shared_counter);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Best Practices

Minimize time spent inside mutex	Keep critical sections small and fast.
Avoid taking multiple mutexes in different orders	Prevent deadlocks.
Use priority-inheritance mutexes for real-time tasks	Avoids priority inversion issues.
Do not lock in high interrupt frequency handlers	Better use lock-free or atomic operations.
Always unlock before returning or on error paths	Use cleanup handlers if needed.

Scenario: Robot Battery Monitor

- Imagine you are developing basic software for a home-cleaning robot. The robot continuously moves around the house and its battery slowly drains. When the battery becomes too low, the robot must automatically go to the charging dock and start charging.
- To simulate this behavior, you must write a C program that uses:
  - **One background thread** → simulates the battery draining
  - **Main thread** → checks battery level periodically
  - **fork() + exec()** → to run an external "charging program" (such as your own charger program that prints *"Robot is now charging..."*)
- **Problem Statement:** Design and implement a C program that simulates a robot's battery monitoring system. The program must meet the following requirements:

Part 1: Battery Drain Simulation (Thread)

Create a thread named **battery_thread** which:

- Starts with battery_level = 100%.
- Every 1 second:
  - Reduces the battery by a random value between **1% and 10%**.
  - Prints: [Sensor] Battery = XX%
- When battery level drops **below 20%**, it must set a shared flag: low_battery = 1.
- Use a mutex to safely update the shared variables.

Part 2: Main Thread Monitoring

In the main thread:

- Periodically (every 1 second) check the **low_battery** flag.
- When low_battery == 1, print: [Main] Low battery detected! Going to charging station...

Then trigger the charging process.

Part 3: Charging Action

When low battery is detected:

- Use fork() to create a child process.
- In the **child process**, run an external program such as:
    - **Your own program** ./charger

      which prints: Robot is now charging...
- In the **parent process**:
    - Wait for the child to finish using wait().
    - Print: [Main] Charging completed!

Part 4: Reset & Continue

After charging finishes:

- Reset battery_level back to **100%**.
- Reset the flag low_battery = 0.
- Continue monitoring forever.