Concurrency & Synchronization Patterns in Embedded Systems (Mutex & Spinlock)

1. Core Concepts of Concurrency Management

In embedded systems, concurrency occurs when multiple contexts—such as the main loop, Interrupt Service Routines (ISRs), or RTOS tasks—attempt to access a shared resource (like a global variable, a hardware peripheral, or a memory buffer) at the exact same time. Without protection, this leads to **Race Conditions**: unpredictable behavior caused by overlapping Read-Modify-Write cycles.

To build ISO 26262-compliant, thread-safe architectures, we utilize synchronization patterns. Instead of scattering RTOS-specific API calls (`xSemaphoreTake`) throughout the application layer, we encapsulate the synchronization mechanism inside the object that owns the resource.

- **The Spinlock (Multi-Core & ISRs):** A non-blocking polling mechanism. If a core wants a resource that is currently locked, it enters a tight `while()` loop (spinning) until the lock is released. Used for extremely short critical sections where putting the CPU to sleep is too costly.
- **The Mutex (RTOS Tasks):** A blocking mechanism. If a task cannot acquire the lock, the RTOS puts the task to sleep (yielding the CPU to other tasks) and wakes it up only when the resource becomes available.
- **Encapsulation:** The Application Layer never interacts with the Mutex directly. It calls an API (e.g., `Sensor_Read()`), and the driver internally handles the locking and unlocking.

---

2. Example 1: Spinlock for Multi-Core / Bare-Metal Systems

When working with multi-core microcontrollers (e.g., ARM Cortex-M dual-core or Infineon Aurix TC3xx), one core might be writing to a shared memory buffer while the other tries to read it. A hardware-backed Spinlock prevents data corruption.

2.1. The Spinlock Utility: `Spinlock.h`

```
#ifndef SPINLOCK_H
#define SPINLOCK_H
```

```c
#include <stdint.h>
#include <stdbool.h>

// A simple structure holding the lock state
typedef struct {
    volatile uint32_t lock_flag;
} Spinlock_t;

// API for initializing and using the spinlock
void Spinlock_Init(Spinlock_t *const me);
void Spinlock_Lock(Spinlock_t *const me);
void Spinlock_Unlock(Spinlock_t *const me);

#endif // SPINLOCK_H
```

2.2. The Implementation: `Spinlock.c`

This typically relies on compiler intrinsics or assembly instructions (like ARM's `LDREX` / `STREX`) to guarantee that the read-modify-write operation is **Atomic** (indivisible by hardware).

```c
#include "Spinlock.h"

void Spinlock_Init(Spinlock_t *const me) {
    me->lock_flag = 0; // 0 = Unlocked, 1 = Locked
}

void Spinlock_Lock(Spinlock_t *const me) {
    // Hardware-specific Atomic Compare-and-Swap
    // Wait in a tight loop as long as the lock is taken (1)
    while (__sync_val_compare_and_swap(&me->lock_flag, 0, 1) != 0) {
        // Spin: Insert a NOP to save power and prevent bus flooding
        __asm("nop");
    }

    // Memory Barrier: Ensure all subsequent memory operations
    // happen strictly AFTER the lock is acquired.
    __sync_synchronize();
}

void Spinlock_Unlock(Spinlock_t *const me) {
    // Memory Barrier: Ensure all memory operations
    // finish BEFORE the lock is released.
    __sync_synchronize();

    me->lock_flag = 0; // Release the lock
}
```

---

3. Example 2: RTOS Mutex Encapsulation (FreeRTOS)

If you are using FreeRTOS, scattering `#include "FreeRTOS.h"` across all your drivers ruins the portability of your code (Hardware Isolation). Instead, the Mutex should be an opaque part of the object.

3.1. The Shared Resource Driver: `SharedI2C.h`

The application does not know that FreeRTOS is being used.

```
#ifndef SHARED_I2C_H
#define SHARED_I2C_H

#include <stdint.h>
#include <stdbool.h>

// Opaque context
typedef struct SharedI2C_t SharedI2C_t;

SharedI2C_t* SharedI2C_Create(void);
bool SharedI2C_WriteData(SharedI2C_t *const me, uint8_t dev_addr, uint8_t

#endif // SHARED_I2C_H
```

3.2. The RTOS-Aware Implementation: `SharedI2C.c`

```
#include "SharedI2C.h"
#include "FreeRTOS.h"
#include "semphr.h"

// The encapsulated structure
struct SharedI2C_t {
    uint32_t i2c_base_address;
    SemaphoreHandle_t mutex; // FreeRTOS specific type hidden inside .c f
};

static SharedI2C_t i2c_instance;

SharedI2C_t* SharedI2C_Create(void) {
    i2c_instance.i2c_base_address = 0x40005400;

    // Create the Mutex during initialization
    if (i2c_instance.mutex == NULL) {
        i2c_instance.mutex = xSemaphoreCreateMutex();
    }

    return &i2c_instance;
}

// Thread-Safe API
bool SharedI2C_WriteData(SharedI2C_t *const me, uint8_t dev_addr, uint8_t
    if (me == NULL || me->mutex == NULL) return false;

    // 1. Attempt to acquire the Mutex (Block for max 100 ticks)
    if (xSemaphoreTake(me->mutex, pdMS_TO_TICKS(100)) == pdTRUE) {

        // --- CRITICAL SECTION START ---
        // No other task can enter this block. Safe to use the hardware.
        Hardware_I2C_Write(me->i2c_base_address, dev_addr, data);
        // --- CRITICAL SECTION END ---

        // 2. Release the Mutex
        xSemaphoreGive(me->mutex);
        return true;
    }

    // Failed to acquire the lock within the timeout
    return false;
}
```

---

4. Architectural Warnings and Best Practices

- **Deadlocks:** Never allow a task to attempt to lock a Mutex it already owns, and never allow a sequence where Task A waits for Task B's lock, while Task B waits for Task A's lock.
- **Priority Inversion (RTOS):** If a low-priority task holds a Mutex that a high-priority task needs, the high-priority task is blocked. Always use Mutexes that support **Priority Inheritance** (where the

RTOS temporarily elevates the low-priority task to get it out of the critical section quickly).

- **Interrupt Latency (Spinlocks):** Never use a Spinlock for a long operation (like a `printf` or a large memory copy). Spinlocks halt other cores/interrupts. Keep the critical section strictly limited to a few CPU cycles (e.g., updating a pointer or a counter flag).
- **Mutexes in ISRs: Never** attempt to take a blocking Mutex inside an Interrupt Service Routine. An ISR cannot yield the CPU. If an ISR needs to synchronize with a Task, use an RTOS Queue, a Thread-Safe Ring Buffer, or a Binary Semaphore (`FromISR` variants).