FreeRTOS synchronization methods

Name

Universidad Panamericana

Presentation September 22, 2024

UNIVERSIDAD
PANAMERICANA

Contents

Inter process sytem communication

Interprocess Communication (IPC)

Interprocess Communication (IPC) refers to the mechanisms provided by the operating system that allow processes to exchange data, synchronize their actions, and notify events between processes.

- **Data Exchange:** Enables processes to share information.
- **Synchronization:** Coordinates processes to avoid race conditions.
- **Event Notification:** Allows a process to signal another when an event occurs.

Semaphores

FreeRTOS Semaphores

Semaphores in FreeRTOS are synchronization tools used to signal between tasks or between tasks and interrupts.

- **Binary Semaphore:** Used for task synchronization.
  - It can only have two states: "give" or "taken".
  - Used to notify tasks that an event has occurred, like an interrupt.
- **Counting Semaphore:** Extends the concept of binary semaphores.
  - It can hold a count, allowing multiple resources or events to be tracked.
  - Useful when you want to track multiple identical resources.

Example Use Cases:

- Synchronizing tasks with an interrupt.
- Protecting access to shared resources.

FreeRTOS Semaphore API Overview

FreeRTOS Semaphore:

- Semaphores are used for task synchronization or resource management in FreeRTOS.
- Two key operations:
  - **xSemaphoreGive**: Releases the semaphore (signals that a resource is available).
  - **xSemaphoreTake**: Acquires the semaphore (blocks a task until the semaphore is available).

Typical Use Case:

- Task synchronization: One task gives the semaphore, and another task waits to take it before proceeding.

xSemaphoreTake API

xSemaphoreTake API:

xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait)

- **xSemaphore**: Handle to the semaphore.
- **xTicksToWait**: The maximum time (in ticks) to block waiting for the semaphore.
- **Return Value**:
  - ‘pdTRUE‘ if the semaphore was successfully taken.
  - ‘pdFALSE‘ if the semaphore was not taken within the timeout period.

Purpose:

- Allows a task to block until the semaphore is available, ensuring that the task does not proceed until another task signals completion (or a resource becomes available).

xSemaphoreGive API

xSemaphoreGive API:

xSemaphoreGive(SemaphoreHandle_t xSemaphore)

- **xSemaphore**: Handle to the semaphore.
- **Return Value**:
  - ‘pdTRUE‘ if the semaphore was successfully given.
  - ‘pdFALSE‘ if the semaphore could not be given (e.g., invalid semaphore).

Purpose:

- Signals that a resource is available or that a task has completed its work, allowing other tasks waiting on the semaphore to proceed.

Example: Task Synchronization with Semaphore

Scenario:

Task A waits for Task B to complete some work. Task B signals Task A by giving the semaphore, and Task A continues its execution after taking the semaphore.

Code Outline:

- Task A calls ‘xSemaphoreTake‘ to block until Task B gives the semaphore.
- Task B completes its work and calls ‘xSemaphoreGive‘ to signal Task A.

Semaphore Workflow Diagram

Semaphore Usage Workflow:

- Task A blocks on ‘xSemaphoreTake‘.
- Task B performs some work.
- Task B calls ‘xSemaphoreGive‘ after completing work.
- Task A unblocks and continues once it successfully takes the semaphore.

Diagram (optional): Show how Task A waits for Task B to give the semaphore.

FreeRTOS Semaphore Example

```
// Create binary semaphore
SemaphoreHandle_t xBinarySemaphore;

void TaskA(void *pvParameters) {
    for(;;) {
        // Wait until Task B gives the semaphore
        if (xSemaphoreTake(xBinarySemaphore,
            portMAX_DELAY) == pdTRUE) {
            // Perform some task after Task B
                signals completion
            // ...
        }
    }
}
```

FreeRTOS Semaphore Example

```
void TaskB(void *pvParameters) {
    for(;;) {
        // Perform some work
        // ...

        // Give the semaphore to signal Task A
        xSemaphoreGive(xBinarySemaphore);

        // Delay before doing the work again
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

FreeRTOS Semaphore Example

Semaphore Example: Task Synchronization

- **Scenario:** Task A waits for an interrupt (ISR) to give a semaphore, allowing Task A to proceed.

```
int main(void) {
    // Create the binary semaphore
    xBinarySemaphore = xSemaphoreCreateBinary();

    // Create the tasks
    xTaskCreate(TaskA, "TaskA", 1000, NULL, 1, NULL)
        ;
    xTaskCreate(TaskB, "TaskB", 1000, NULL, 1, NULL)
        ;

    // Start the scheduler
    vTaskStartScheduler();

    // Will never reach here
    for(;;);
}
```

Mutexes

FreeRTOS Mutexes

A **Mutex** (Mutual Exclusion) in FreeRTOS is a specialized type of semaphore used to protect shared resources.

- **Ownership:** A task that successfully “takes” a mutex becomes its owner and is the only one allowed to “give” it back.
- **Priority Inheritance:** FreeRTOS mutexes include a priority inheritance mechanism to prevent priority inversion.
- **Recursive Mutexes:** A task can take the same mutex multiple times, but it must give it the same number of times before it is released for others.

Example Use Cases:

- Protecting critical sections of code.
- Managing access to hardware resources like peripherals.

FreeRTOS Mutex API Overview

Mutex in FreeRTOS:

- A mutex (mutual exclusion) is used to protect shared resources between tasks.
- Unlike a binary semaphore, a mutex provides **priority inheritance** to avoid priority inversion.

Key Mutex API Functions: API works the same as semaphore.

- **xSemaphoreCreateMutex**: Creates a mutex.
- **xSemaphoreTake**: Locks the mutex (blocks if already locked by another task).
- **xSemaphoreGive**: Unlocks the mutex (releases it for other tasks to use).

xSemaphoreCreateMutex API

xSemaphoreCreateMutex API:

```
// Create a mutex
SemaphoreHandle_t xMutex;

void vInit(void) {
    xMutex = xSemaphoreCreateMutex();
    if (xMutex == NULL) {
        // Mutex creation failed
    }
}
```

Queues

FreeRTOS Queues

Queues are the primary method of inter-task communication in FreeRTOS, allowing tasks to send and receive data.

- **Task Communication:** Tasks can send data to each other via queues, allowing safe communication and data exchange.
- **FIFO Structure:** Queues operate on a First-In-First-Out (FIFO) basis.
- **Multiple Readers/Writers:** Multiple tasks can write to and read from a queue safely.
- **Interrupt-Safe:** FreeRTOS queues can be used to send data from an ISR (Interrupt Service Routine) to a task.

Queue API Functions

Common functions provided by the Queue API:

- `xQueueCreate()`: Creates a queue with a specified number of elements.
- `xQueueSend()`: Sends data to the queue (from producer).
- `xQueueReceive()`: Receives data from the queue (by consumer).
- `xQueuePeek()`: Reads the data without removing it from the queue.
- `xQueueReset()`: Resets the queue, discarding all data.

Example Creating a Queue

Example of how to create a queue that can hold 10 integers:

```
QueueHandle_t xQueue;
xQueue = xQueueCreate(10, sizeof(int));
if (xQueue == NULL) {
    // Queue was not created successfully
}
```

Explanation:

- `xQueueCreate(10, sizeof(int))` creates a queue to store 10 integers.

Example Sending Data to a Queue

Example of how to send data (integer) to a queue:

```c
int valueToSend = 42;
if (xQueueSend(xQueue, &valueToSend, 0) != pdPASS) {
    // Failed to send data to queue
}
```

Explanation:

- `xQueueSend()` places `valueToSend` in the queue.
- The third parameter is the wait time (0 means don't wait).

Example - Receiving Data from a Queue

Example of how to receive data from the queue:

- `xQueueReceive()` retrieves data from the queue.
- `portMAX_DELAY` means it will wait indefinitely until data is available.

```
int receivedValue;
if (xQueueReceive(xQueue, &receivedValue, portMAX_DELAY)
    ) {
    // Successfully received data from queue
}
```

Example - Queue with Multiple Tasks

Listing 1: Queue with Multiple Tasks

```
// Task 1: Producer
void vTaskProducer(void *pvParameters) {
    int valueToSend = 1;
    while (1) {
        xQueueSend(xQueue, &valueToSend, portMAX_DELAY);
        vTaskDelay(1000); // Send data every second
    }
}

// Task 2: Consumer
void vTaskConsumer(void *pvParameters) {
    int receivedValue;
    while (1) {
        if (xQueueReceive(xQueue, &receivedValue,
            portMAX_DELAY)) {
             // Process received value
        }
    }
}
```

ISR special API

Interrupt Context vs. Task Context:

- Regular FreeRTOS API calls like xSemaphoreGive, xQueueSend, or xSemaphoreTake involve task management and may block, which is not allowed in ISR context. Interrupts should execute quickly and not invoke code that could block or wait.
- Special ISR APIs allow the ISR to inform the scheduler that a context switch is necessary after the ISR completes, by setting a flag (xHigherPriorityTaskWoken).
- Semaphores and queues can be safely used in ISRs with special ISR-safe APIs (xSemaphoreGiveFromISR, xQueueSendFromISR) to avoid blocking and manage context switching efficiently.
- **Mutexes should not be used in ISRs**, and there is no ISR-safe API for them in FreeRTOS.