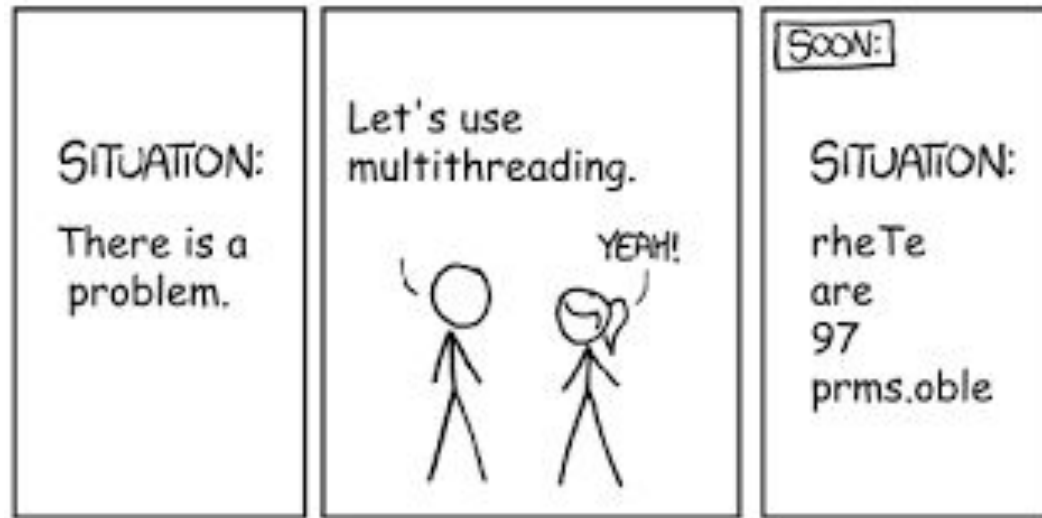


# Concurrencia y manejo de recursos compartidos



# ¿Qué es la concurrencia?

## A modo de repaso....:

- Es la ejecución de varios (dos o más) flujos de código que se ejecutan en un mismo sistema, c/u realizando acciones específicas, dando la impresión de que se ejecutan todos al mismo tiempo...a estos flujos de código los llamaremos *Tareas*
- Cada tarea se apropia temporalmente del CPU y accede a los recursos del sistema (procesamiento, memoria, periféricos, etc...)
- En algún momento la tarea cede (por un rato) el uso del CPU a otra tarea
- La tarea que cede momentáneamente el CPU a otra (voluntaria o involuntariamente), más tarde debe retomar con la ejecución de la instrucción siguiente a la última que ejecutó, como si nada hubiera sucedido. Para esto se guarda una copia de los registros del CPU (contexto)

## ¿Qué es la concurrencia?

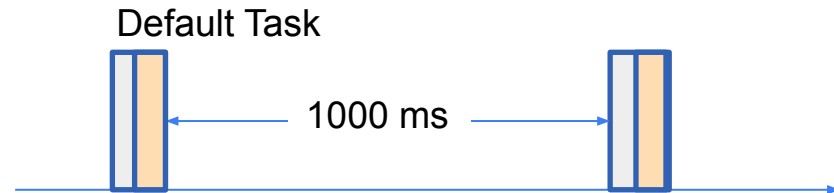
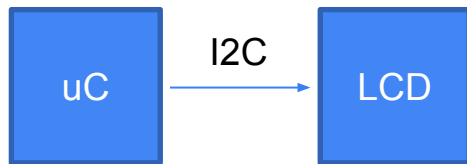
Este esquema concurrente con tareas que son interrumpidas temporalmente y que luego son reanudadas en la instrucción en que fueron interrumpidas ocurre comúnmente en dos situaciones

- **Background/Foreground (1 o más ISRs)**
- **Multithreading con RTOS (más ISRs)**

# Problemas comunes en ejecución concurrente

- Acceso concurrente al Hardware
- Accesos r-m-w **no atómicos** a variables globales
- Llamadas a funciones **no-reentrant** / **no thread-safes**

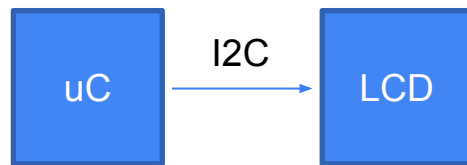
# Acceso concurrente al Hardware



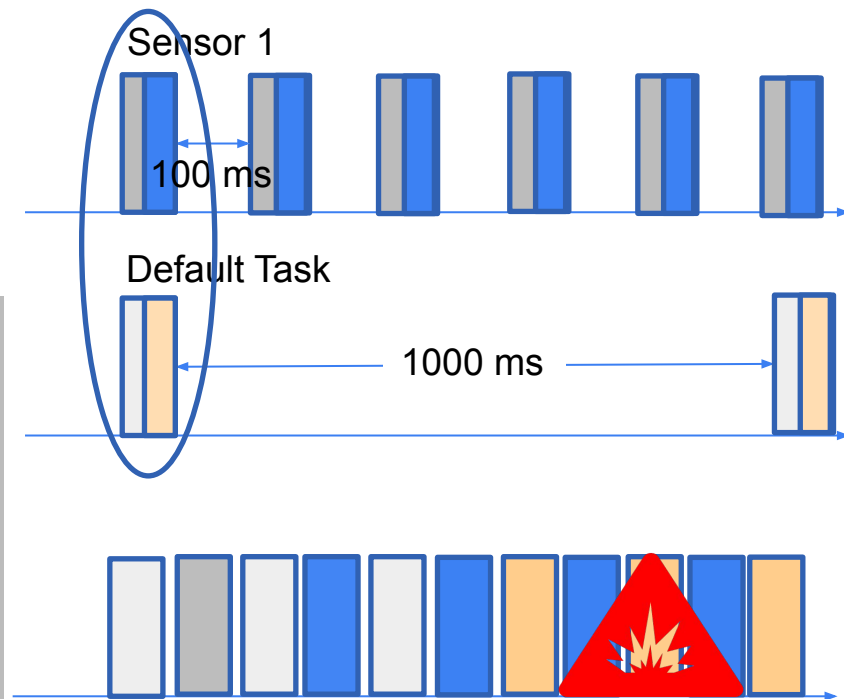
```
void ImprimirLinea(char *s, uint8_t linea) {  
    /* "borra" la línea para poder escribir en limpio */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts("          ", &Font_7x10, 1);  
  
    /* imprime la cadena */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts(s, &Font_7x10, 1);  
  
    /*actualiza pantalla*/  
    SSD1306_UpdateScreen();  
}
```

```
void StartDefaultTask(void *argument) {  
    for (;;) {  
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  
  
        if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)) {  
            ImprimirLinea("LED = OFF", 1);  
        } else {  
            ImprimirLinea("LED = ON", 1);  
        }  
  
        osDelay(1000);  
    }  
}
```

# Acceso concurrente al Hardware

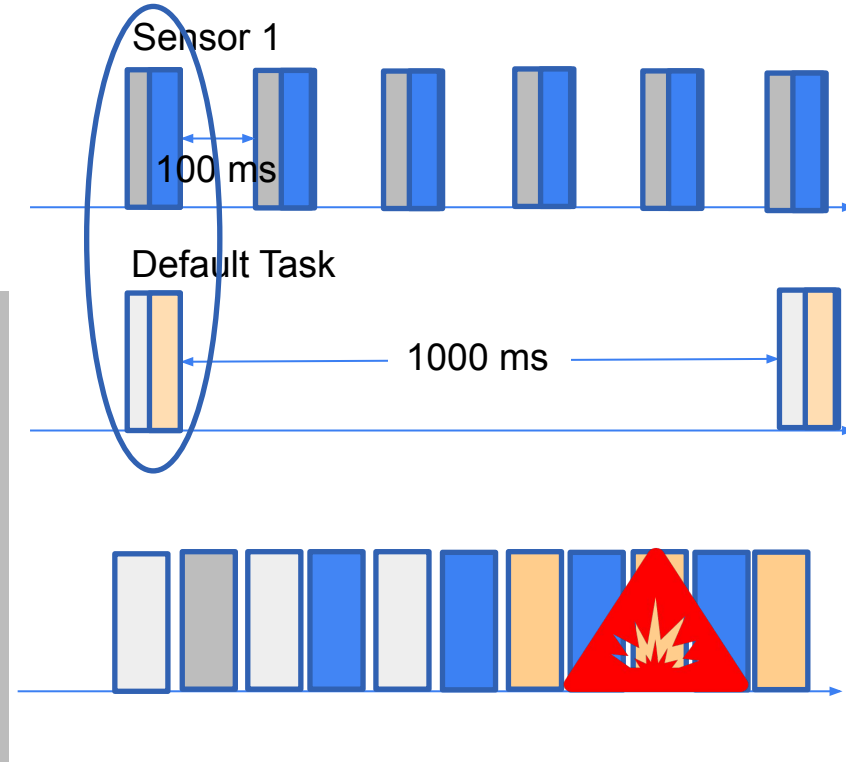
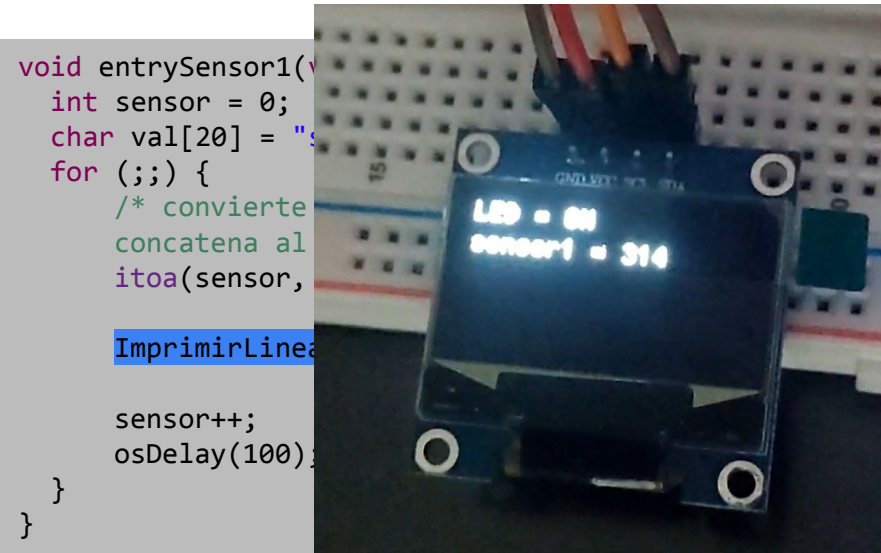
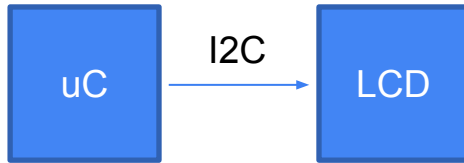


```
void entrySensor1(void *argument) {  
    int sensor = 0;  
    char val[20] = "sensor1 = ";  
    for (;;) {  
        /* convierte el valor del sensor a ascii y lo  
        concatena al final de "sensor2 = " */  
        itoa(sensor, val + 10, 10);  
  
        ImprimirLinea(val, 2);  
  
        sensor++;  
        osDelay(100);  
    }  
}
```



# Problemas comunes en ejecución concurrente

- Acceso concurrente al Hardware**



# Exclusión mutua

Una “solución” al problema de los **múltiples accesos en simultáneo al mismo recurso** es detectar la **región de código crítico** e implementar un **bloqueo o exclusión** mediante un flag global, que garantice **que sólo se accede de a una tarea por vez al recurso**

```
volatile uint8_t bloqueo = 0;

void ImprimirLinea(char *s, uint8_t linea) {
    while (bloqueo == 1);
    bloqueo = 1;
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("                ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    bloqueo = 0;
}
```



# Exclusión mutua

```
volatile uint8_t bloqueo = 0;

void ImprimirLinea(char *s, uint8_t linea) {
    while (bloqueo == 1);
    bloqueo = 1;
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("                ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    bloqueo = 0;
}
```

Inicialmente el código crítico no está bloqueado

La primera tarea que acceda a este código no ejecutará el bucle de bloqueo, pero bloqueará el código para las demás tareas

# Exclusión mutua

```
volatile uint8_t bloqueo = 0;

void ImprimirLinea(char *s, uint8_t linea) {
    while (bloqueo == 1);
    bloqueo = 1;
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("                ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    bloqueo = 0;
}
```

Si alguna otra tarea intenta ejecutar este código quedará bloqueada en el bucle, hasta que la primera la desbloquee cuando finalice la ejecución

# Accesos r-m-w no atómicos a variables globales

La anterior es una mala “solución”, ya que se accede a una variable para leerla, tomar decisiones, modificarla y guardarla de manera **no atómica**

```
volatile uint8_t bloqueo = 0;

void ImprimirLinea(char *s, uint8_t linea) {
    while (bloqueo == 1);
    bloqueo = 1;
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("          ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    bloqueo = 0;
}
```

Puede haber un cambio de tarea en cualquier momento entre la instrucción de lectura (PC=80003AC) y la de escritura (PC=80003B8)

<b>while (bloqueo == 1);</b>		
80003a8:	bf00	nop
80003aa:	4b19	ldr r3, [pc, #100];
<b>80003ac:</b>	<b>781b</b>	<b>ldrb r3, [r3, #0]</b> <b>Lectura</b>
80003ae:	b2db	uxtb r3, r3
80003b0:	2b01	cmp r3, #1
80003b2:	d0fa	beq.n 80003aa <b>Toma de decisión</b> <b>(bloquea o ejecuta)</b>
<b>bloqueo = 1;</b>		
80003b4:	4b16	ldr r3, [pc, #88]
80003b6:	2201	movs r2, #1
<b>80003b8:</b>	<b>701a</b>	<b>strb r2, [r3, #0]</b> <b>Escritura</b>

## Accesos r-m-w no atómicos a variables globales

Si se accede a una variable global de manera **no atómica** siempre existirá la posibilidad de que ocurra un cambio de contexto (ISR o expropiación por parte del RTOS) entre la lectura, la toma de decisión, la modificación y la escritura.

**Luego del cambio de contexto, en una nueva tarea (o ISR) puede volver a leerse la variable (con su valor original) y tomarse una decisión que lleve al desastre.**

Para el ejemplo anterior, más de una tarea accedería al código crítico y se comunicaría con la pantalla.

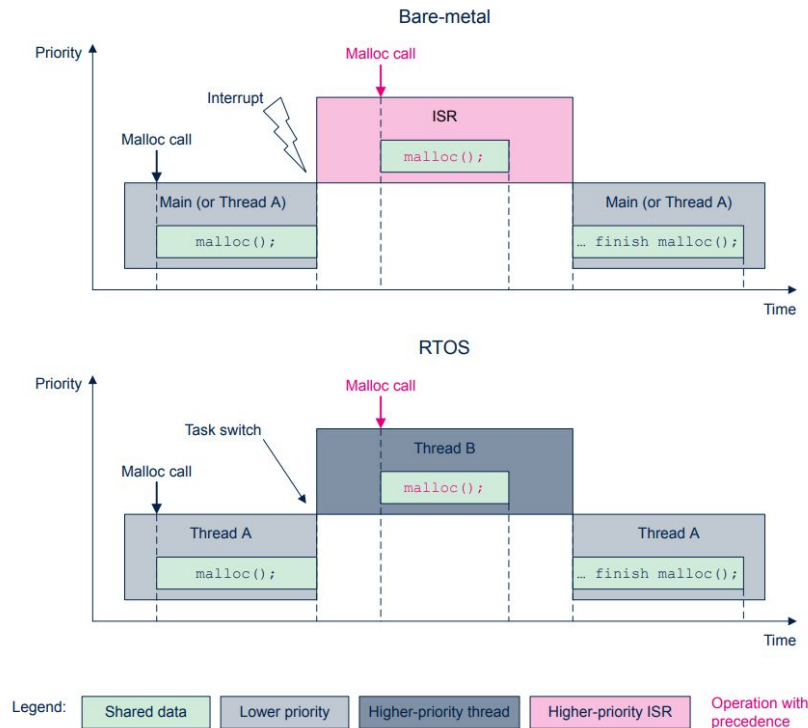
## Uso de funciones inseguras

Al usar de librerías perdemos noción de lo que hace su código interno, pero siempre deberíamos asumir que no es seguro usarla libremente desde distintas tareas sin comprometer los datos o el correcto funcionamiento (no es **thread-safe**).

Cualquier función que haga uso de una variable global o estática (que no esté almacenado en el stack de la tarea), o un recurso de HW se considerará **no-reentrante**

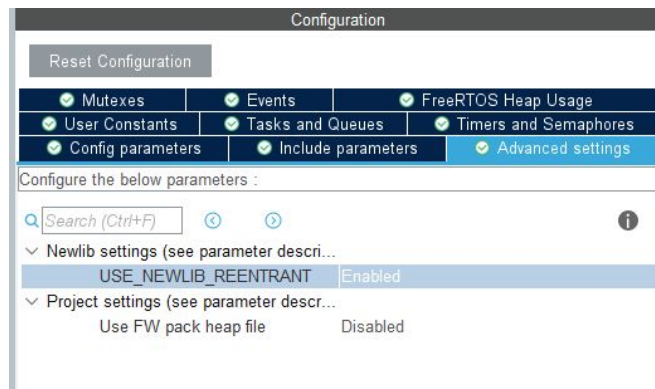
Conviene buscar librerías de terceros probadas en entornos multitareas, o escribir *wrappers* de las funciones que las conviertan en versiones seguras.

# Uso de funciones inseguras



# Uso de funciones inseguras

Si se van a utilizar las librerías estándar de C desde muchas tareas, hay que habilitar la opción `USE_NEWLIB_REENTRANT`  
Solo funciona para concurrencia entre tareas, no con ISRs.



# Uso de funciones inseguras

Para el uso seguro de la librería de C, incluyendo las ISRs, el CubeMX ofrece distintas estrategias

Figure 3. Thread-safe settings

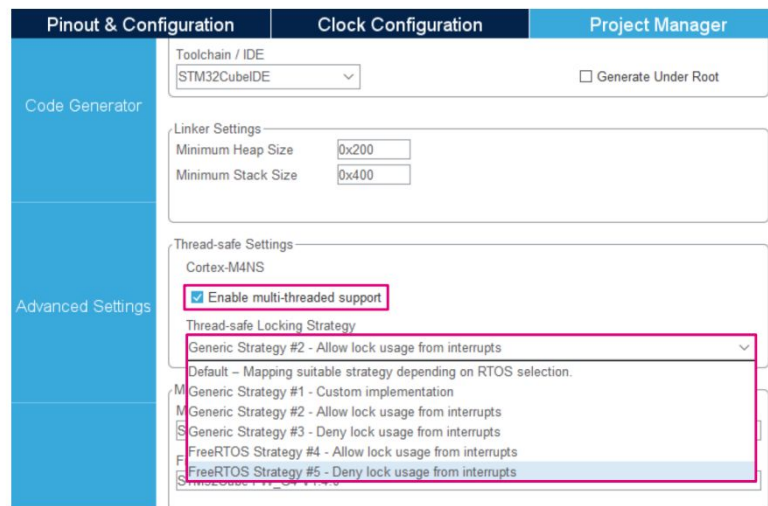
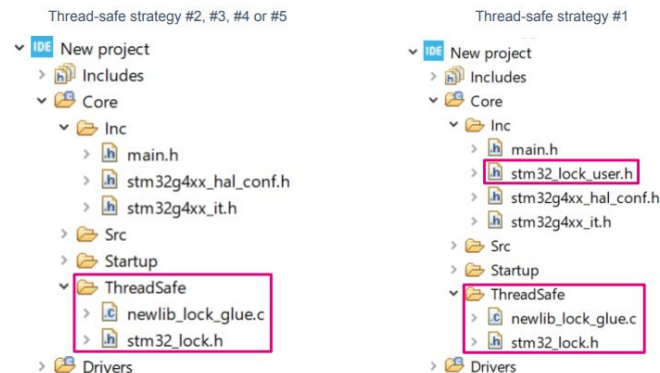


Figure 7. STM32CubeIDE single-core project configuration





# Uso de funciones inseguras

Las HAL, por otro lado, intentan implementar la exclusión mutua pero no lo hacen correctamente, ya que no hay un acceso atómico a la variable de bloqueo.

```
stm32_hal_def.h

typedef enum
{
    HAL_UNLOCKED = 0x00U,
    HAL_LOCKED   = 0x01U
} HAL_LockTypeDef;

#if (USE_RTOS == 1U)
/* Reserved for future use */
#error "USE_RTOS should be 0 in the current HAL release"
#else
#define __HAL_LOCK(__HANDLE__) \
do{ \
    if((__HANDLE__)->Lock == HAL_LOCKED) \
    { \
        return HAL_BUSY; \
    } \
    else \
    { \
        (__HANDLE__)->Lock = HAL_LOCKED; \
    } \
}while (0U)

#define __HAL_UNLOCK(__HANDLE__) \
do{ \
    \
    (__HANDLE__)->Lock = HAL_UNLOCKED; \
    \
}while (0U)

#endif /* USE_RTOS */
```

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit
(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
uint8_t *pData, uint16_t Size, uint32_t Timeout)
{
    if (hi2c->State == HAL_I2C_STATE_READY)
    {
        /* Preparativos de la comunicación*/

        /* Process Locked */
        __HAL_LOCK(hi2c);
        /****
        Comunicación insegura
        *****/
        __HAL_UNLOCK(hi2c);

        return HAL_OK;
    }
    else
    {
        return HAL_BUSY;
    }
}
```

stm32\_hal\_i2c.c

# Posibles soluciones al manejo de recursos compartidos

- **Suspensión de interrupciones**
- **Suspensión del scheduler**
- **Mutex**
- **No compartir recursos**



**Exclusión mutua**

# Suspensión de interrupciones

Cuando se identifica una sección crítica, esta puede protegerse de cambios de contexto indeseados deshabilitando interrupciones (los cambios de contexto solo pueden realizarse si ocurre una interrupción)

El mecanismo de FreeRTOS para deshabilitar y rehabilitar interrupciones en secciones de código crítico son las funciones: `taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()`

La sección crítica encerrada por estas debe ser breve y el tiempo de ejecución lo más determinístico posible, ya que se incrementa la latencia de todas las interrupciones.

# Suspensión de interrupciones

**taskENTER\_CRITICAL()** desactiva de forma **atómica** las interrupciones. Solo la tarea que logre ejecutarla tendrá acceso al código crítico.

```
void ImprimirLinea(char *s, uint8_t linea)
{
    taskENTER_CRITICAL();
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("                ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    taskEXIT_CRITICAL();
}
```

# Suspensión de interrupciones

En realidad enmascara todas las fuentes de interrupción con prioridad menor a `configMAX_SYSCALL_INTERRUPT_PRIORITY` (mayor valor numérico)

```
void ImprimirLinea(char *s, uint8_t linea)
{
    taskENTER_CRITICAL();
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(" ", &Font_7x10);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
    taskEXIT_CRITICAL();
}
```

Código en `portDISABLE_INTERRUPTS()` llamado por `taskENTER_CRITICAL()`

```
uint32_t ulNewBASEPRI;
__asm volatile(
    "mov %0, %1\n" \
    "msr basepri, %0\n" \
    "isb\n" \
    "dsb\n" \
    : "=r" (ulNewBASEPRI)
    : "i" ( configMAX_SYSCALL_INTERRUPT_PRIORITY )
    : "memory"
);
```

# Suspensión de interrupciones

```
void ImprimirLinea(char *s, uint8_t linea) {  
    taskENTER_CRITICAL();  
    /* "borra" la línea para poder escribir en limpio */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts("                ", &Font_7x10, 1);  
  
    /* imprime la cadena */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts(s, &Font_7x10, 1);  
  
    /*actualiza pantalla*/  
    SSD1306_UpdateScreen();  
    taskEXIT_CRITICAL();  
}
```

La sección crítica estará protegida hasta que se llame a **taskEXIT\_CRITICAL()** momento en el que se rehabilitan las interrupciones

# Suspensión de interrupciones - Anidamiento

En FreeRTOS es seguro anidar secciones críticas, ya que se lleva un contador de llamadas y solo se rehabilitan las interrupciones cuando el contador vuelve a cero. Sin este contador, el primer llamado a `taskEXIT_CRITICAL()` habilitaría las interrupciones antes de lo deseado.

```
void ImprimirLinea(char *s, uint8_t linea) {  
    taskENTER_CRITICAL();  
    /* "borra" la línea para poder escribir en limpio */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts("                ", &Font_7x10, 1);  
  
    /* imprime la cadena */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts(s, &Font_7x10, 1);  
  
    /*actualiza pantalla*/  
    SSD1306_UpdateScreen();  
    taskEXIT_CRITICAL();  
}
```

portDISABLE\_INTERRUPTS();  
uxCriticalNesting++;

uxCriticalNesting--;  
if( uxCriticalNesting == 0 )  
{  
 portENABLE\_INTERRUPTS();  
}

## Suspensión de interrupciones - Latencia

Como el tiempo de ejecución de una sección crítica debe ser corto, la solución propuesta no es adecuada, ya que la comunicación con la pantalla LCD demora decenas de ms. Y durante ese tiempo el sistema está bloqueado en una única tarea

```
void ImprimirLinea(char *s, uint8_t linea) {  
    taskENTER_CRITICAL();  
    /* "borra" la línea para poder escribir en limpio */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts("                ", &Font_7x10, 1);  
  
    /* imprime la cadena */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts(s, &Font_7x10, 1);  
  
    /*actualiza pantalla*/  
    SSD1306_UpdateScreen();  
    taskEXIT_CRITICAL();  
}
```



## Suspensión de interrupciones - Acceso atómico a variables

Un mejor uso de la suspensión de interrupciones es para que la lectura, decisión y escritura de variables globales compartidas se realice de forma atómica.

```
volatile uint8_t bloqueo = 0;

void ImprimirLinea(char *s, uint8_t linea) {
    uint8_t salir_del_bloqueo = 0;
    do{
        taskENTER_CRITICAL();
        if(bloqueo == 0){
            salir_del_bloqueo = 1;
            bloqueo = 1;}
        taskEXIT_CRITICAL();
    }while(salir_del_bloqueo == 0);

    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts("                                ", &Font_7x10, 1);

    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);

    /*actualiza pantalla*/
    SSD1306_UpdateScreen();

    bloqueo = 0;
}
```

## Suspensión del scheduler

Otra forma de proteger secciones críticas es suspendiendo el scheduler (esto previene cambios de contexto entre tareas pero no entre tarea/ISR).

Si el recurso compartido no es accedido desde ninguna ISR conviene esta alternativa.

En lugar de utilizar `taskENTER_CRITICAL()` / `taskEXIT_CRITICAL()` Debe usarse alguno de los siguientes pares de funciones:

- API FreeRTOS: `vTaskSuspendAll()` / `xTaskResumeAll()`
- API CMSIS-RTOS V2: `osKernelLock()` / `osKernelRestoreLock()`

# Mutex

Es un objeto provisto por el SO que permite implementar de manera sencilla la **exclusión mutua sobre un recurso, basada en la posesión de un token**.

**Solo una tarea podrá poseer el token**, acceder al recurso y ejecutar el código crítico correspondiente, mientras que el resto de las tareas pasará a un estado de bloqueo.

Cuando la tarea que posee el token lo libera, solo una de las que están en espera lo toma y es la única que puede acceder al recurso.

Es un mecanismo que solo involucra a las tareas que desean hacer uso del recurso, mientras que las otras se ejecutan libremente.

Las tareas bloqueadas a la espera del mutex no consumen ciclos de CPU.

NO DEBE USARSE DESDE ISRs

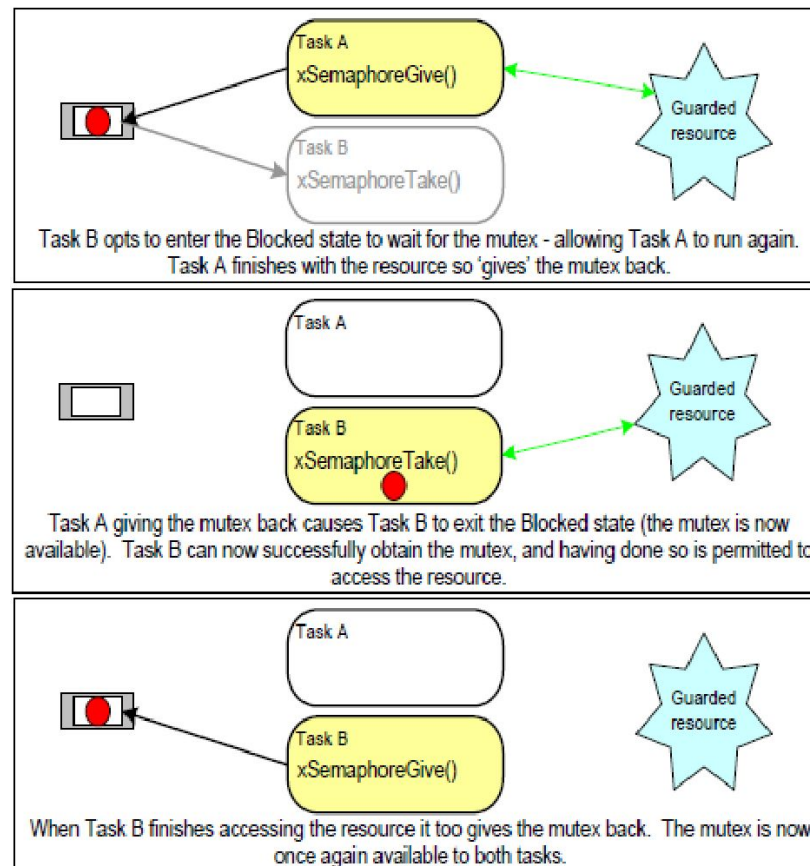
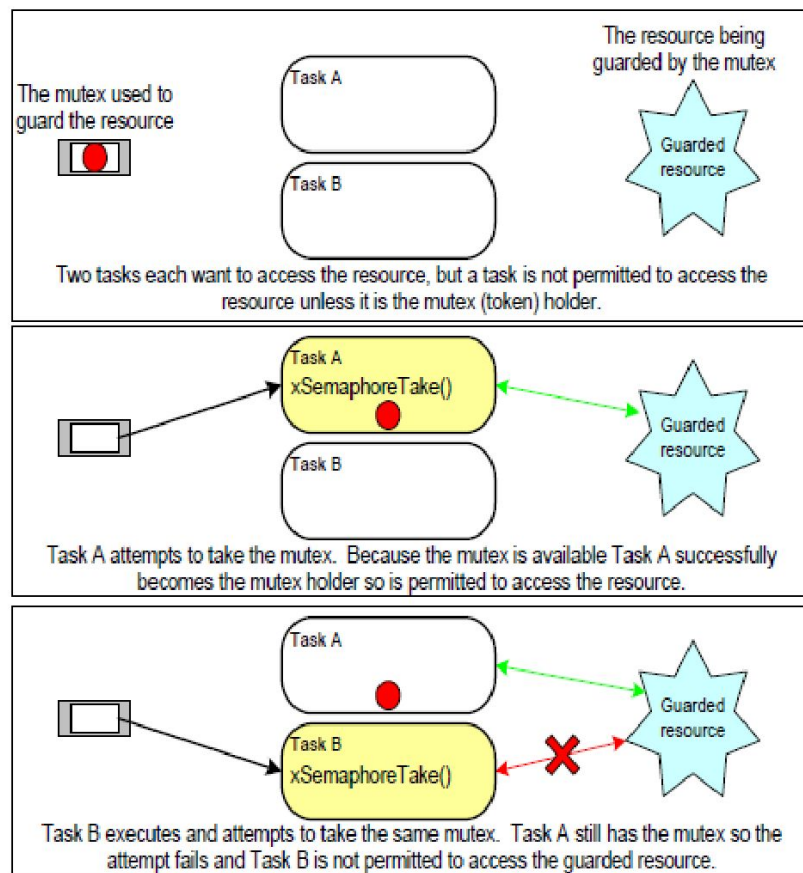


Figure 63. Mutual exclusion implemented using a mutex

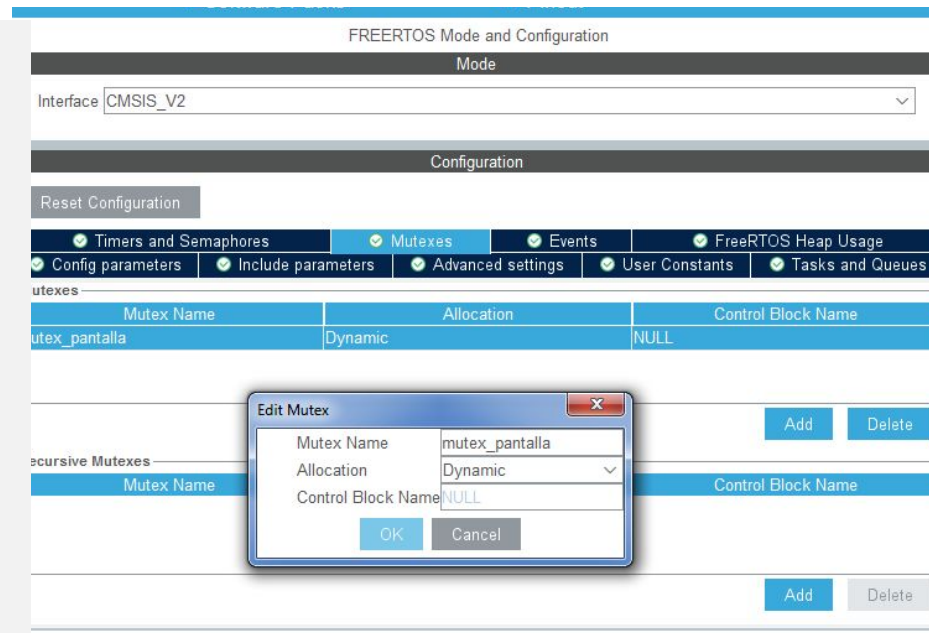
# Mutex

	API FreeRTOS	API CMSIS-RTOS V2
tipo	SemaphoreHandle_t	osMutexId_t
creación	<b>xSemaphoreCreateMutex()</b> ;	<b>osMutexNew</b> (*attr)
posesión	<b>xSemaphoreTake</b> ( xSemaphore, xBlockTime )	<b>osMutexAcquire</b> (mutex_id, timeout)
devolución	<b>xSemaphoreGive</b> ( xMutex )	<b>osMutexRelease</b> (mutex_id)

Para poder utilizarlo deberá configurarse  
configUSE\_MUTEXES en 1

# Mutex

```
void ImprimirLinea(char *s, uint8_t linea) {  
  
    osMutexAcquire(mutex_pantallaHandle, 1000);  
  
    /* "borra" la línea para poder escribir en limpio */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts("                ", &Font_7x10, 1);  
  
    /* imprime la cadena */  
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));  
    SSD1306_Puts(s, &Font_7x10, 1);  
  
    /*actualiza pantalla*/  
    SSD1306_UpdateScreen();  
  
    osMutexRelease(mutex_pantallaHandle);  
}
```

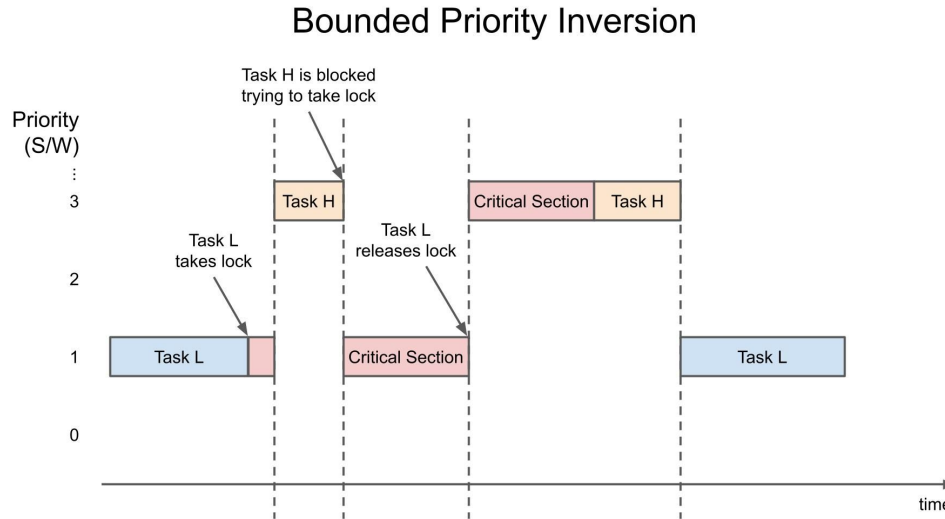


## Mutex – problemas asociados

- Inversión de prioridades
- Deadlocks
- Mutex anidados

# Mutex – problemas asociados

- Inversión de prioridades

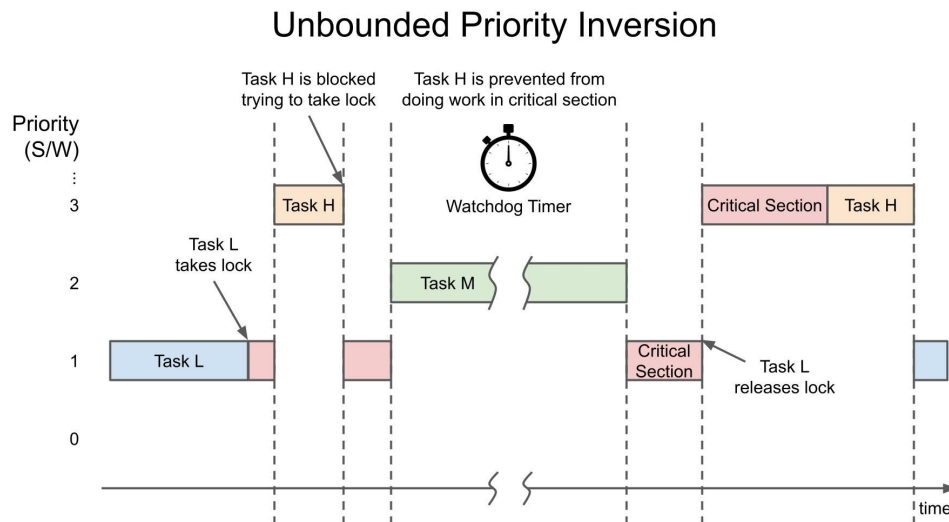


Este es el comportamiento deseado, con un bloqueo acotado por la duración de la sección crítica (que debe hacerse lo más corta posible)



# Mutex – problemas asociados

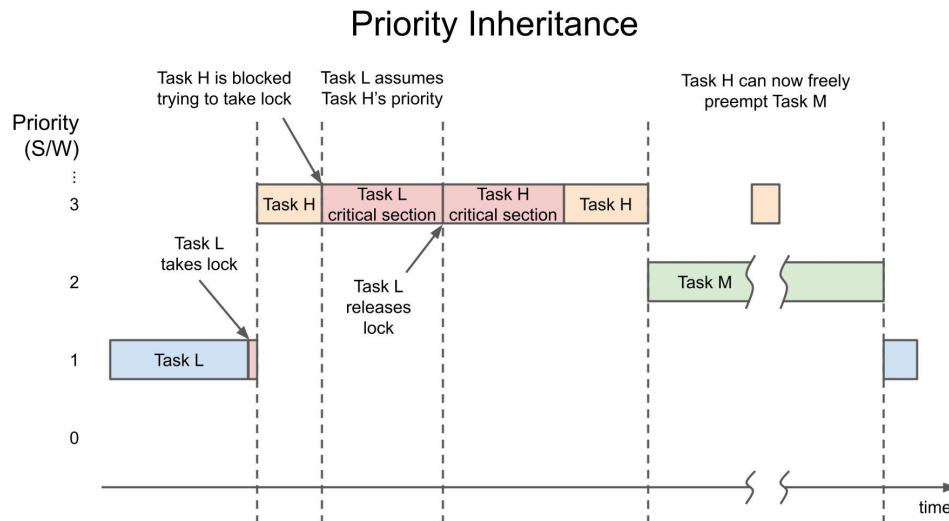
- Inversión de prioridades



Si hay otras tareas, el bloqueo deja de estar acotado, se pierde determinismo y se invierten las prioridades

# Mutex – problemas asociados

- Inversión de prioridades



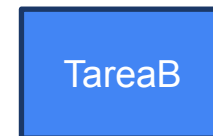
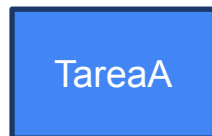
La **herencia de prioridades** soluciona el bloqueo no acotado, promoviendo la tarea propietaria del mutex a la prioridad más alta de las tareas que se encuentran bloqueadas esperando el acceso al recurso

# Mutex – problemas asociados

- **Deadlock**

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

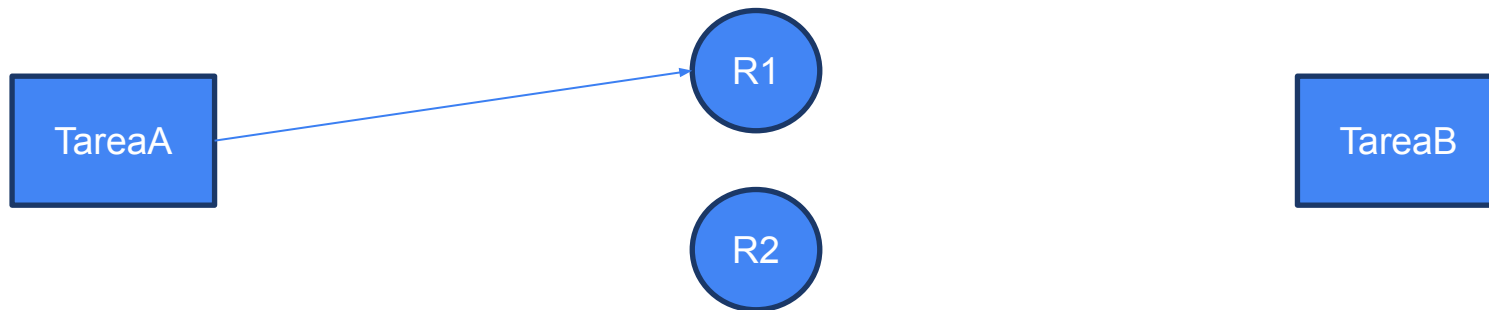


# Mutex – problemas asociados

- **Deadlock**

→ `main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}`

`main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}`

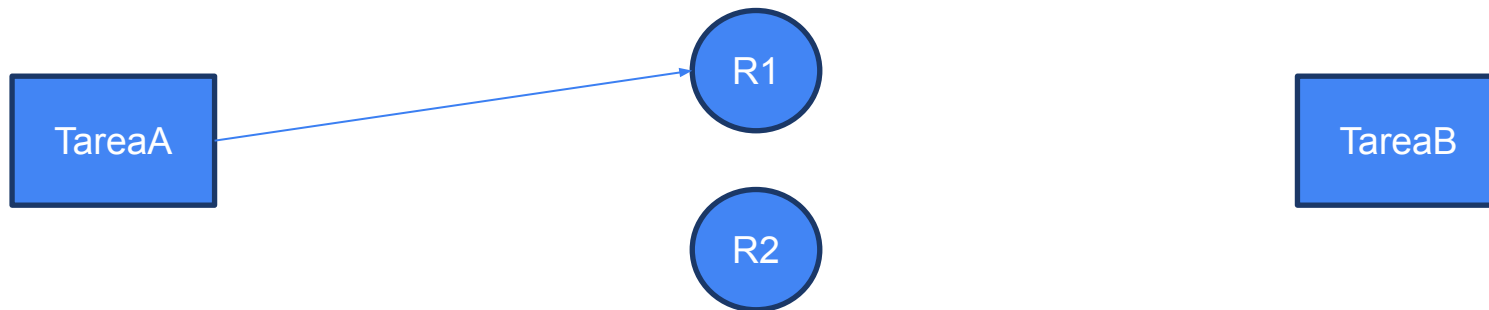


# Mutex – problemas asociados

- Deadlock

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

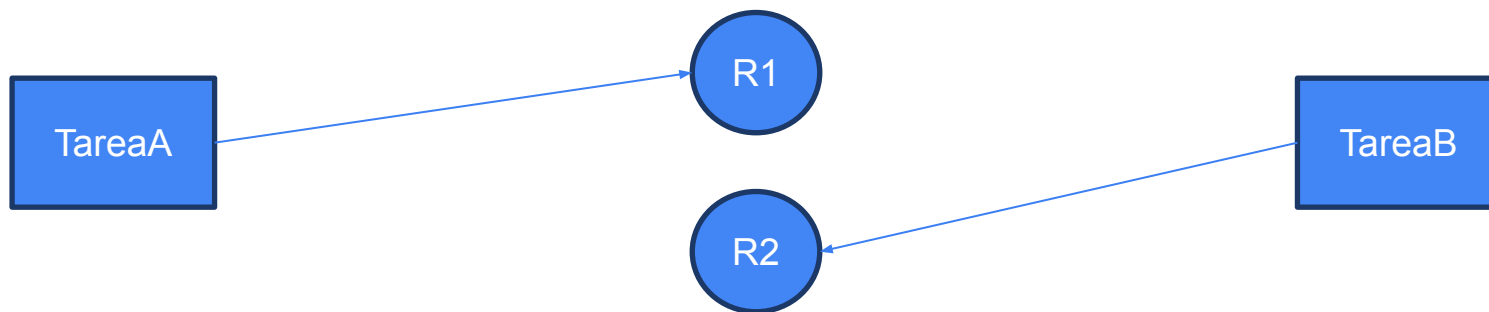


# Mutex – problemas asociados

- **Deadlock**

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

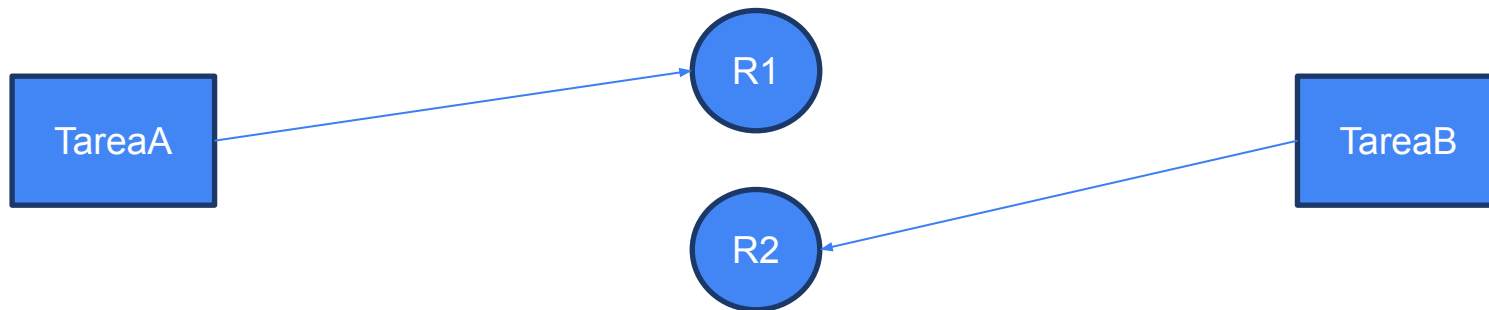


# Mutex – problemas asociados

- **Deadlock**

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

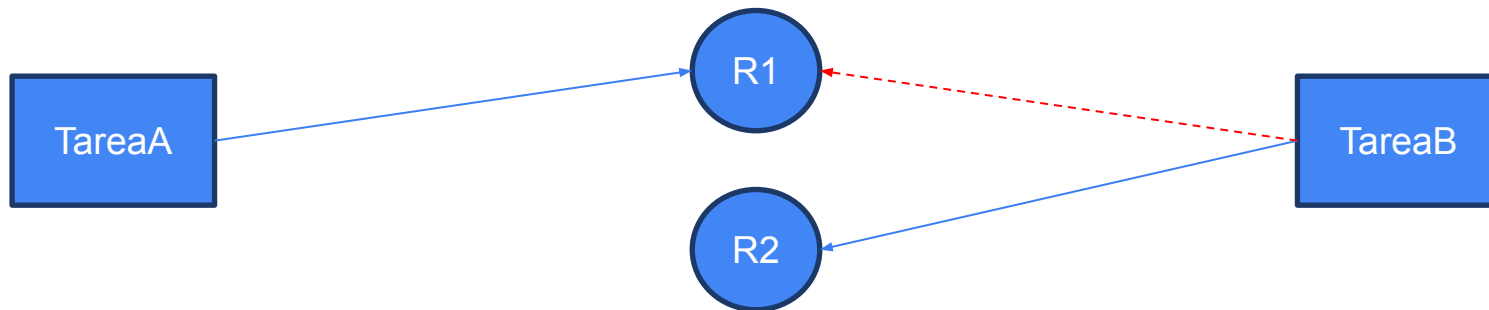


# Mutex – problemas asociados

- Deadlock

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```



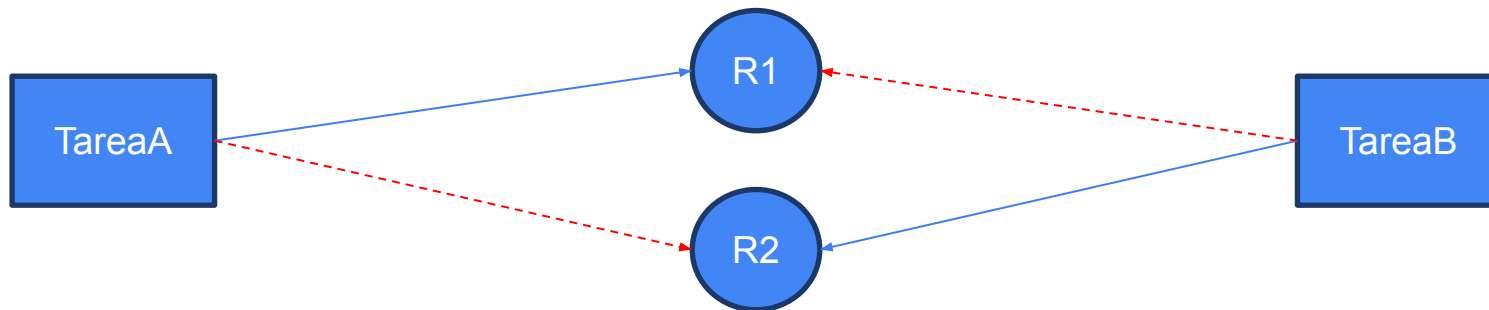


# Mutex – problemas asociados

- Deadlock

```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    ➔ xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    ➔ xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

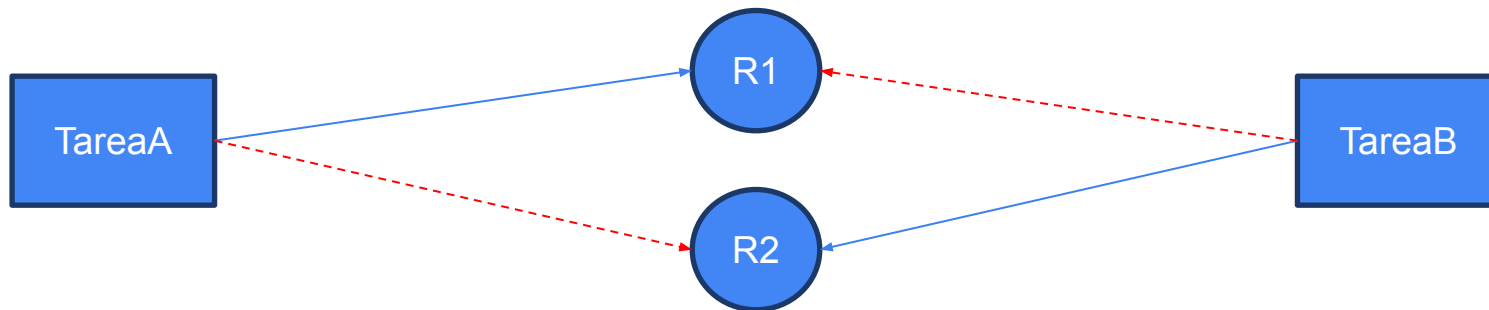


# Mutex – problemas asociados

- Deadlock

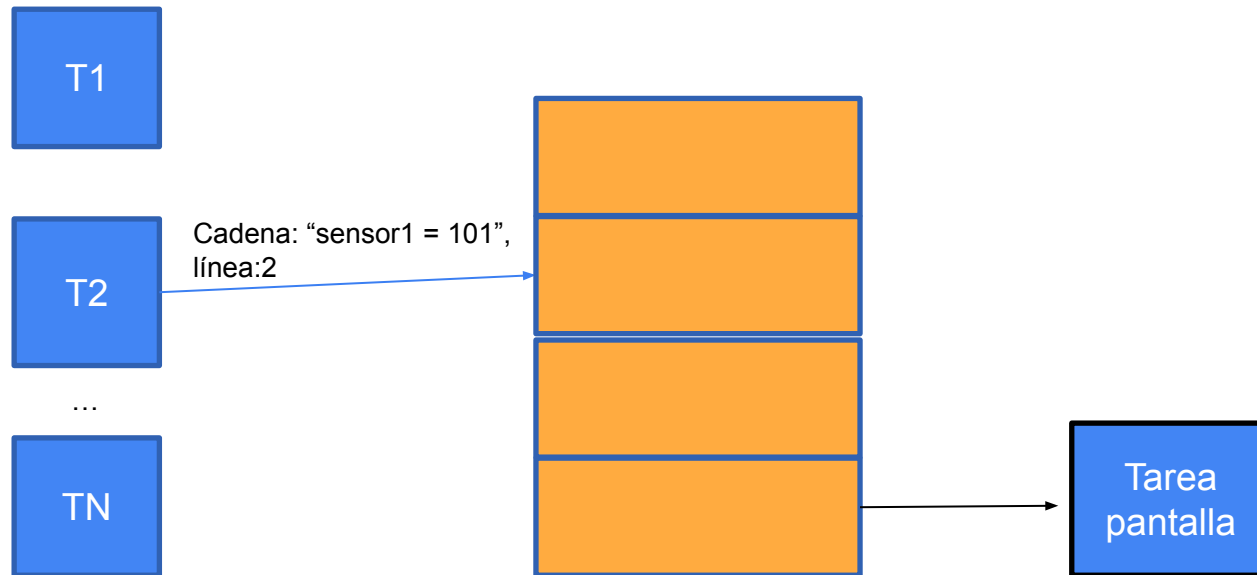
```
main_TareaA(){  
    xSemaphoreTake(Recurso1);  
    //hace algo con Recurso1  
    ➔ xSemaphoreTake(Recurso2);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```

```
main_TareaB(){  
    xSemaphoreTake(Recurso2);  
    //hace algo con Recurso2  
    ➔ xSemaphoreTake(Recurso1);  
    //hace algo con ambos Recursos  
    xSemaphoreGive(Recurso1);  
    xSemaphoreGive(Recurso2);  
}
```



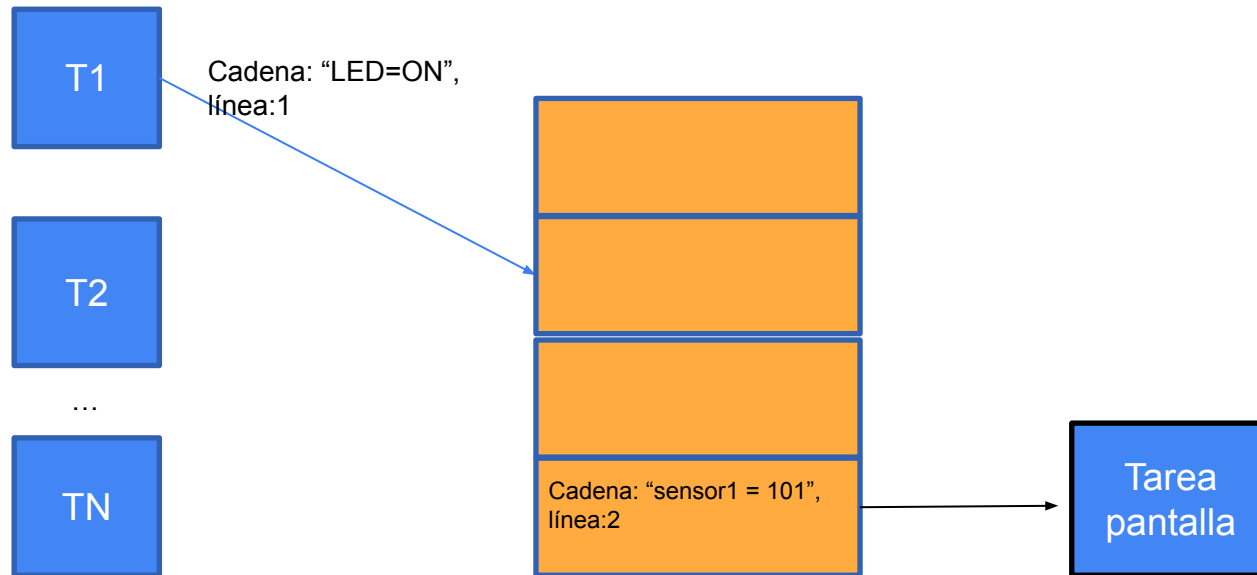
## Solución extrema ☐ No compartir recursos

Una única **tarea de servicio** gestiona el acceso al recurso desde diferentes tareas a través de colas o buffers.



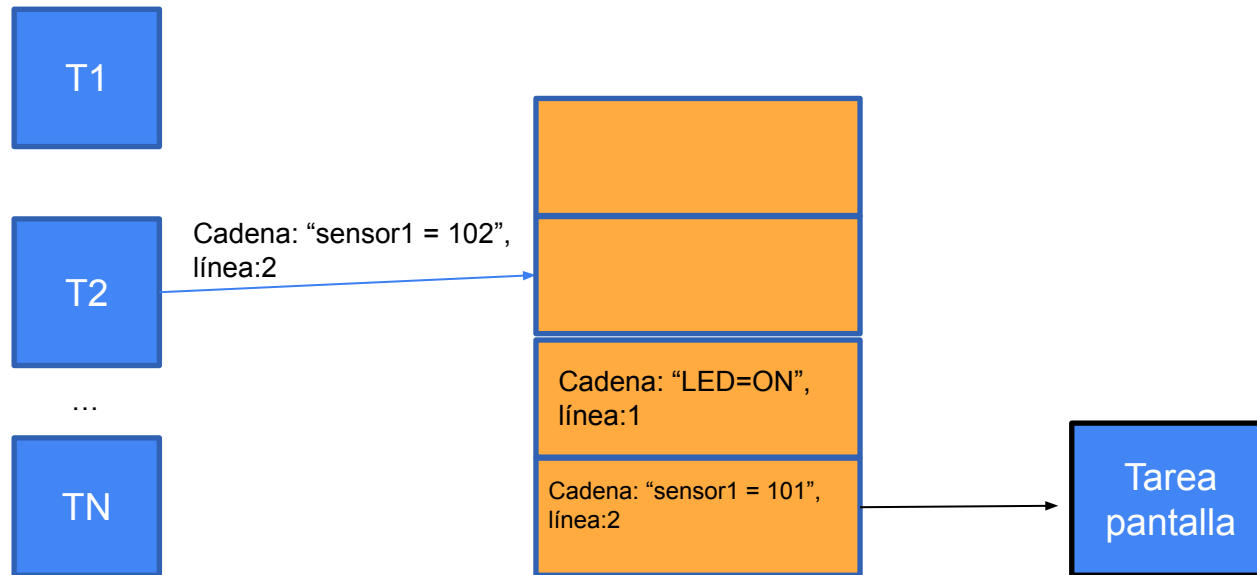
## Solución extrema ☐ No compartir recursos

Una única **tarea de servicio** gestiona el acceso al recurso desde diferentes tareas a través de colas o buffers.



## Solución extrema ☐ No compartir recursos

Una única **tarea de servicio** gestiona el acceso al recurso desde diferentes tareas a través de colas o buffers.



## Queue (cola/FIFO)

Es un elemento provisto por el SO que permite implementar y manipular de manera sencilla una cola de mensajes, datos u objetos entre tareas. Posee un tamaño fijo (cantidad de elementos), para almacenar datos de un tipo determinado (tamaño de cada elemento).

Suele utilizarse de manera que solo una tarea consume los mensajes, mientras que una o más tareas escriben en la misma.

Una tarea que desea escribir en un FIFO lleno se bloquea sin consumir ciclos de CPU, y el RTOS la desbloquea cuando se hace lugar en el FIFO (o expira un timeout).

Una tarea que desea leer de una cola vacía se bloquea sin consumir ciclos de CPU y el RTOS la desbloquea cuando se escribe algo nuevo en la cola (o expira un timeout).

El RTOS resuelve la exclusión mutua en caso de que muchas tareas estén queriendo acceder “en simultáneo” para leer/escribir en la cola.

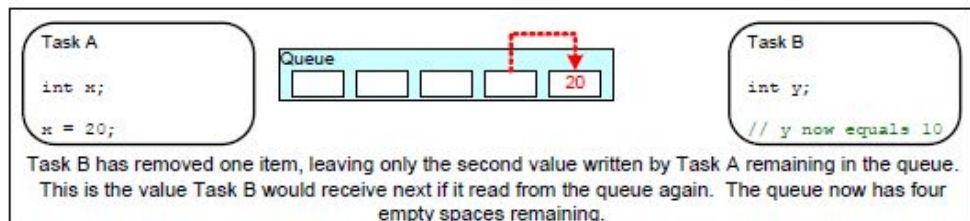
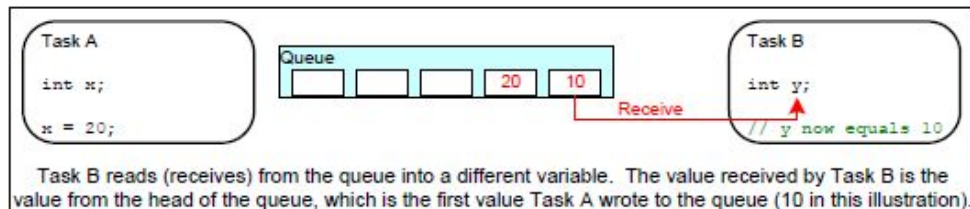
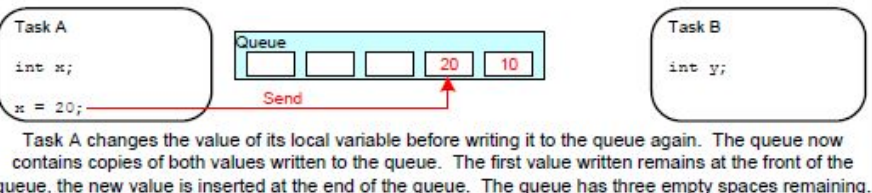
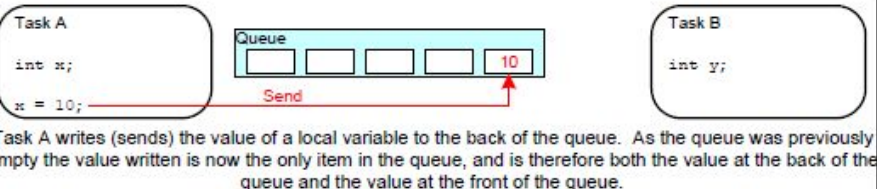
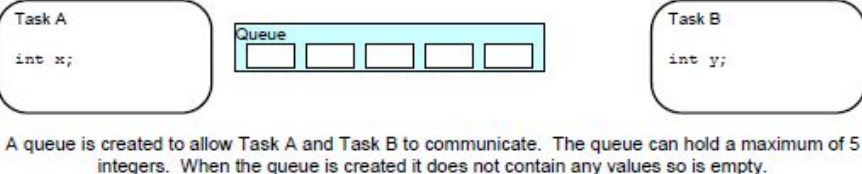





Figure 31. An example sequence of writes to, and reads from a queue

	API FreeRTOS	API CMSIS-RTOS V2
tipo	QueueHandle_t	osMessageQueueId_t
creación	<b>xQueueCreate</b> (msg_count, msg_size)	<b>osMessageQueueNew</b> (msg_count, msg_size, *attr)
escritura	BaseType_t <b>xQueueSend</b> ( QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait );  BaseType_t <b>xQueueSendToFront</b> ( QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait );	osStatus_t <b>osMessageQueuePut</b> (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout);
lectura	BaseType_t <b>xQueueReceive</b> ( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait );	osStatus_t <b>osMessageQueueGet</b> (osMessageQueueId_t mq_id, void *msg_ptr, uint8_t *msg_prio, uint32_t timeout)

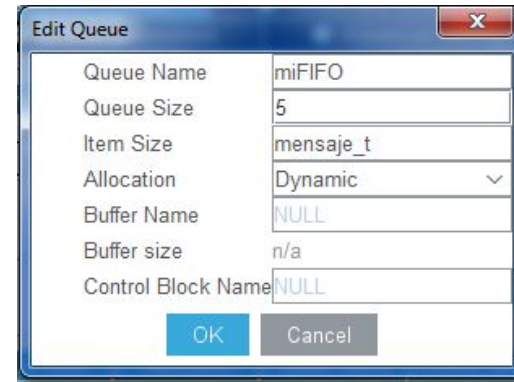


	API FreeRTOS	API CMSIS-RTOS V2
tipo	QueueHandle_t	osMessageQueueId_t
creación	<b>xQueueCreate</b> (msg_count, msg_size)	<b>osMessageQueueNew</b> (msg_count, msg_size, *attr)
escritura	BaseType_t <b>xQueueSend</b> ( QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait );  BaseType_t <b>xQueueSendToFront</b> ( QueueHandle_t xQueue, const void * pvltemToQueue, TickType_t xTicksToWait ); 	osStatus_t <b>osMessageQueuePut</b> (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout); 
lectura	BaseType_t <b>xQueueReceive</b> ( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait );	osStatus_t <b>osMessageQueueGet</b> (osMessageQueueId_t mq_id, void *msg_ptr, uint8_t *msg_prio, uint32_t timeout)

## Solución extrema ☐ No compartir recursos

```
typedef struct {char cad[20]; int linea;} mensaje_t;
```

La tarea consumidora de los datos del FIFO será la única que pueda escribir en la pantalla por lo que se puede usar nuevamente la versión insegura de ImprimirLinea()



```
void entryPantalla(void *argument)
{
    mensaje_t dato;

    for(;;)
    {
        osMessageQueueGet( miFIFOHandle, &dato, NULL, osWaitForever);
        ImprimirLinea(dato.cad, dato.linea);
    }
}
```

```
void ImprimirLinea(char *s, uint8_t linea) {
    /* "borra" la línea para poder escribir en limpio */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(" ", &Font_7x10, 1);
    /* imprime la cadena */
    SSD1306_GotoXY(0, 0 + 14 * (linea - 1));
    SSD1306_Puts(s, &Font_7x10, 1);
    /*actualiza pantalla*/
    SSD1306_UpdateScreen();
}
```

## Solución extrema ☐ No compartir recursos

Las dos tareas que antes imprimían directamente en la pantalla ahora producen datos y los encolan en el fifo

```
void StartDefaultTask(void *argument)
{
    for (;;) {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);

        if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)) {
            //ImprimirLinea("LED = OFF", 1);
            mensaje_t dato={"LED = OFF",1};
            osMessageQueuePut (miFIFOHandle, &dato, NULL, osWaitForever);
        } else {
            //ImprimirLinea("LED = ON", 1);
            mensaje_t dato={"LED = ON",1};
            osMessageQueuePut (miFIFOHandle, &dato, NULL, osWaitForever);
        }
        osDelay(500);
    }
}
```

```
void entrySensor1(void *argument)
{
    int sensor = 0;
    char val[20] = "sensor1 = ";

    for (;;) {
        /* convierte el valor del sensor a ascii y lo concatena al
           final de "sensor2 = " */
        itoa(sensor, val + 10, 10);

        //ImprimirLinea(val, 2);
        mensaje_t dato={"",2};
        strcpy(dato.cad,val);
        osMessageQueuePut (miFIFOHandle, &dato, NULL, osWaitForever);

        sensor++;
        osDelay(100);
    }
}
```

# Práctica

1. Replicar el ejemplo de las diapositivas 5-7 y verificar los problemas de concurrencia
2. En proyectos distintos resuelva el problema de concurrencia utilizando:
  - a. Suspensión de interrupciones
  - b. Suspensión del scheduler
  - c. Mutex
  - d. Una única tarea que acceda al recurso compartido y un fifo
3. Instale una nueva tarea en cada uno de los proyectos anteriores que imprima una tercera línea en la pantalla, intercalando entre su nombre y su apellido cada 200 ms.