

# Carrera de Especialización en Sistemas Embebidos

## Sistemas Operativos en Tiempo Real

### Clase 3: Sincronización de tareas (2da parte)

# Problema: Acceso a recursos globales.

- Necesito utilizar un recurso global
  - ¿ Cómo se puede evitar el acceso concurrente al mismo?
- Ej: Una proceso envía bytes por un puerto serie, mientras que otro recibe comandos por USB una nueva configuración de ese puerto serial (baudrate, por ejemplo). Ambos no pueden "tocar" el periférico a la vez.
- Acceso concurrente significa acceder a un mismo recurso (RAM global, periférico, etc) por dos o más tareas "a la vez".
- Se necesita un mecanismo necesario para asegurar que un proceso acceda a un cierto recurso global en forma exclusiva al proceso.
- Pero... ¿ Que significa "a la vez" ?

# Caso de estudio

```
int balance = 200; /* var compartida*/


bool QuitarFondos(int monto)
{
    if( balance >= monto )
    {
        balance -= monto;
        return true;
    }

    return false;
}
```

```
void Tarea1()
{
    /* codigo */
    int valor = RecibirPorUART(); //100
    if( QuitarFondos( valor ) )
    {
        /* procedimiento de operación
        exitosa */
    }
    /* codigo */
}
```

```
void Tarea2()
{
    /* codigo */
    int valor = RecibirPorWeb(); //150
    if( QuitarFondos( valor ) )
    {
        /* procedimiento de operación
        exitosa */
    }
    /* codigo */
}
```

Línea de tiempo (arranca tarea 1: Running):



```
if( balance >= monto )      Cambio de contexto
if( balance >= monto )      Cambio de contexto
balance -= monto;           //balance = 100
balance -= monto;           Cambio de contexto
return true;                //balance = -50
return true;                Cambio de contexto
```

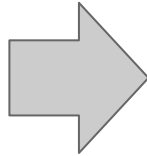
## Condición de carrera:

Es una situación generada cuando el resultado de la ejecución de un grupo de tareas depende del orden en que éstas accedan a los recursos compartidos.

# Ejemplo: Ocurre a nivel de assembler

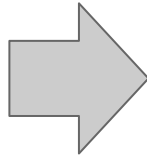
```
uint32_t contador;
```

```
static void IncrementarCantidad()  
{  
    contador++;  
}
```



```
static void IncrementarCantidad()  
{  
1a000398:    b480                push {r7}  
1a00039a:    af00                add  r7, sp, #0  
                    contador++;  
1a00039c:    4b04                ldr  r3, [pc, #16]  
1a00039e:    681b                ldr  r3, [r3, #0]  
1a0003a0:    3301                adds r3, #1  
1a0003a2:    4a03                ldr  r2, [pc, #12]  
1a0003a4:    6013                str  r3, [r2, #0]  
}  
1a0003a6:    46bd                mov  sp, r7  
1a0003a8:    f85d 7b04           ldr.w r7, [sp], #4  
1a0003ac:    4770                bx   lr
```

```
static void DecrementarCantidad()  
{  
    contador--;  
}
```



```
static void DecrementarCantidad()  
{  
1a0003b4:    b480                push  {r7}  
1a0003b6:    af00                add   r7, sp, #0  
                    contador--;  
1a0003b8:    4b04                ldr   r3, [pc, #16]  
1a0003ba:    681b                ldr   r3, [r3, #0]  
1a0003bc:    3b01                subs r3, #1  
1a0003be:    4a03                ldr   r2, [pc, #12]  
1a0003c0:    6013                str   r3, [r2, #0]  
}  
1a0003c2:    46bd                mov   sp, r7  
1a0003c4:    f85d 7b04           ldr.w r7, [sp], #4  
1a0003c8:    4770                bx    lr
```

# Concurrencia

---

- Es una propiedad fundada en la dinámica de un sistema intentando, desde diferentes fuentes, acceder al mismo tiempo a un cierto recurso.
- Las tareas en un sistema operativo apropiativo son:
  - Asíncronas
  - Independientes
  - Las instrucciones de las mismas se intercalan en cualquier orden a causa del cambio de contexto.

# Sección Crítica

- La sección crítica es una porción de código que, de ejecutarse concurrentemente por varias tareas, se realice de manera serializado (y no intercalado como en los ejemplos anteriores).
- Esto significa que la ejecución de secciones críticas deben ser de exclusión mutua.

```
while( 1 )
{
    ENTRAR_EN_SECCION_CRITICA();
    /*
    Seccion critica
    */
    SALIR_DE_SECCION_CRITICA();
    /*
    Seccion NO critica
    */
}
```

# Reglas para una Sección Crítica

- Solo un proceso al mismo tiempo puede entrar en una sección crítica.
- Un proceso que está ejecutando código fuera de una sección crítica, no puede prevenir que otro entre.
- Un proceso que quiera entrar en una sección crítica, no debe ser retrasado de forma indefinida.
- Cuando ningún proceso está dentro de la sección crítica, cualquier otro que quiera entrar, lo podrá hacer sin retraso.
- El proceso dentro de la sección crítica debe durar un tiempo reducido.

# ¿ Como resolver el problema? #1



- Deshabilitar Interrupciones GLOBALMENTE durante la sección crítica.
  - Es buena cuando se implementan rutinas del Kernel del OS o cuando se está diseñando alguna librería o driver.

- Esta acción deshabilita CUALQUIER TIPO de cambio de contexto

```
void FuncionCritica()
{
    taskENTER_CRITICAL();
    /*
    Seccion critica
    */
    taskEXIT_CRITICAL();
    /*
    Seccion NO critica
    */
}
```

Línea de tiempo:

```
taskENTER_CRITICAL();
/*
Seccion critica
*/
taskEXIT_CRITICAL();

taskENTER_CRITICAL();
/*
Seccion critica
*/
taskEXIT_CRITICAL();
```

Cambio de contexto



- Desventajas
  - Esta acción podría atentar contra el tiempo de respuesta ante eventos críticos (si la SC tarda mucho en ejecutarse).



# ¿ Como resolver el problema? #2



- Deshabilitar cambio de contexto de tareas

```
void FuncionCritica()  
{  
    vTaskSuspendAll();  
    /*  
    Seccion critica  
    */  
    xTaskResumeAll();  
    /*  
    Seccion NO critica  
    */  
}
```



Línea de tiempo:

```
vTaskSuspendAll();  
/*  
Seccion critica  
*/  
xTaskResumeAll();  
  
vTaskSuspendAll();  
/*  
Seccion critica  
*/  
xTaskResumeAll();
```

Cambio de contexto

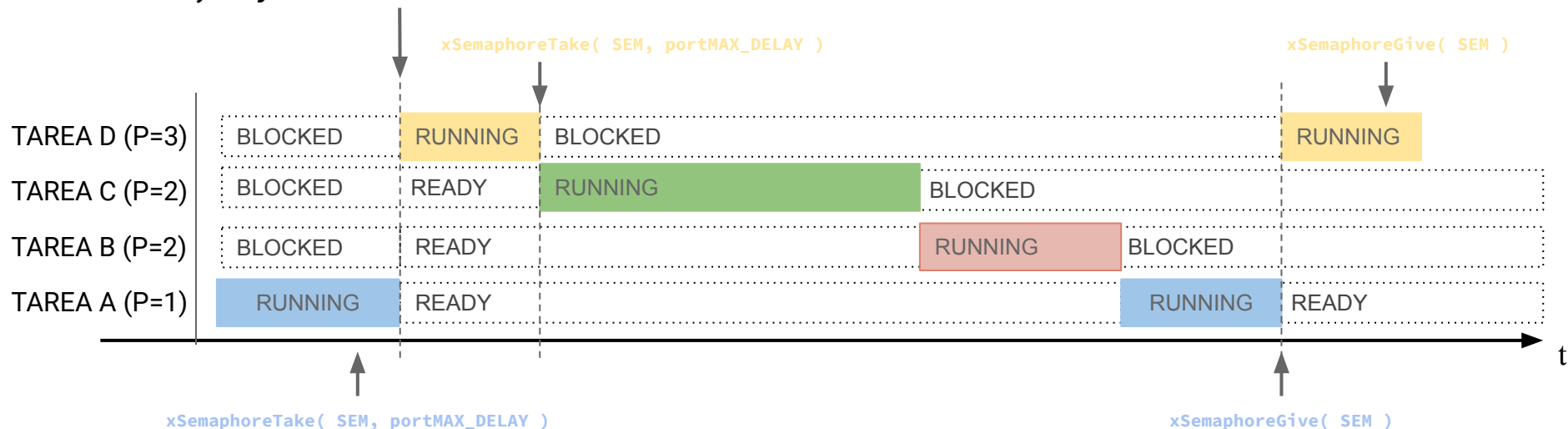
- Desventajas
  - No previene que otras interrupciones accedan a los recursos globales.

# ¿ Como resolver (**MAL**) el problema?



- Se puede "abrir" y "cerrar" la sección crítica con xSemaphoreTake y xSemaphoreGive respectivamente.
  - Esto hace que el segundo proceso que quiera "entrar" en la sección crítica, quede bloqueado hasta que el primero que entró, libere al semáforo.

**Tareas B, C y D cambian a estado READY**



- Ocurre cuando dos tareas de diferente prioridad comparten un recurso. El recurso es accedido por la tarea de más baja prioridad y sin haberlo liberado, la tarea de alta prioridad comienza a ejecutarse. La tarea de alta prioridad va a querer usar el recurso, y como lo tiene "tomado" la otra, se bloquea, cediendo el CPU a la tarea de baja prioridad.
- Para minimizar este efecto, FreeRTOS incluye un tipo especial de semáforo binario llamado MUTEX que solo sirve para proteger (por exclusión mutua) a una sección crítica, minimizando el efecto de la inversión de prioridades.

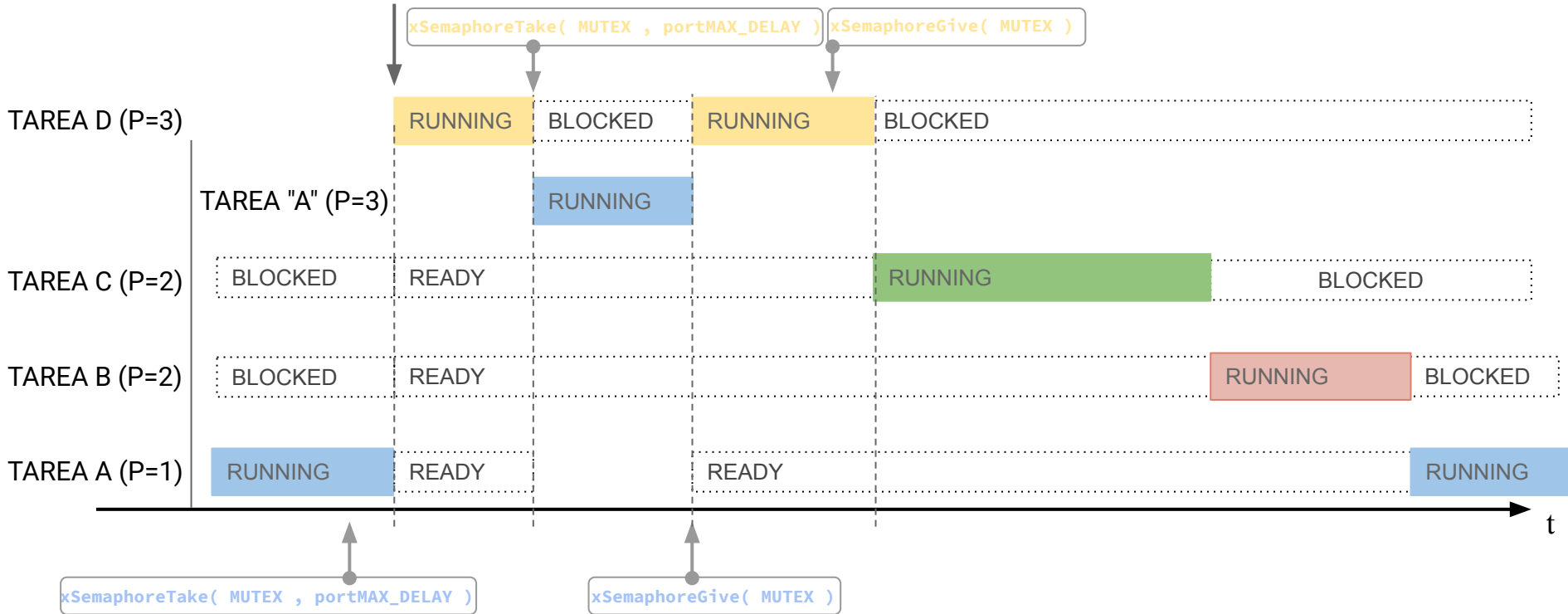
El algoritmo usado es el de "Priority Inheritance"

- El MUTEX es equivalente a un Semáforo Binario, pero con la diferencia que incluye el algoritmo de "priority inheritance" para minimizar el efecto la inversión de prioridades (no se puede evitar al 100%)
- Algoritmo:
  - La herencia de prioridad se basa en el siguiente mecanismo:  
Si una tarea de cierta prioridad (HPT) desea tomar un recurso, ya tomado por una tarea de prioridad menor (LPT), entonces eleva la prioridad de LPT, para que al bloquear a HPT, el cambio de contexto se realice a LPT, para ésta libere el recurso lo mas rapido posible. Al liberarlo, la LPT obtiene su prioridad original.
- Usa la misma API de semáforos, salvo la función para crearlo:

SemaphoreHandle\_t **xSemaphoreCreateMutex**( void )

# Resolución con Mutex

Tareas B, C y D cambian a estado READY



# ¿ Como resolver el problema? #3



- Utilizando mutex

- Tiene sentido cuando un recurso se utiliza durante un tiempo considerable desde varias tareas, de distintas prioridades.
- Permite los cambios de contexto

```
SemaphoreHandle_t mut = xSemaphoreCreateMutex();

void FuncionCritica()
{
    xSemaphoreTake( mut , portMAX_DELAY );
    /*
    Seccion critica
    */
    xSemaphoreGive( mut );
    /*
    Seccion NO critica
    */
}
```

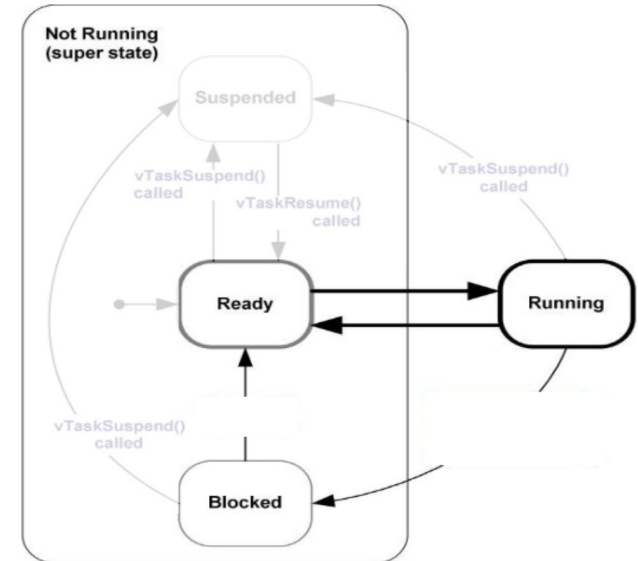
Línea de tiempo:

```
xSemaphoreTake( SEM, portMAX_DELAY );
/*
Seccion critica
*/
xSemaphoreGive( SEM );
Cambio de contexto
xSemaphoreTake( SEM, portMAX_DELAY );
/*
Seccion critica
*/
xSemaphoreGive( SEM );
```



# Problemas en el uso de bloqueos

- La utilización de recursos del RTOS que dejen a tareas a la espera de eventos asincrónicos, hace al sistema elegible para que presente problemáticas en su ejecución.
- Los problemas que se mencionan aquí están asociados a una falencia por parte del programador.
- En general están relacionados con:
  - Mala asignación de prioridades
  - Indebido uso de semáforos u otros elementos de sincronización entre tareas.
  - Eventos externos.



# Deadlock

- Ocurre cuando dos (o más) tareas toman recursos utilizados por el otro hilo. Esto puede provocar que ambos hilos esperen la liberación del recurso tomado por el otro y se bloqueen indefinidamente.

**Tarea 1**

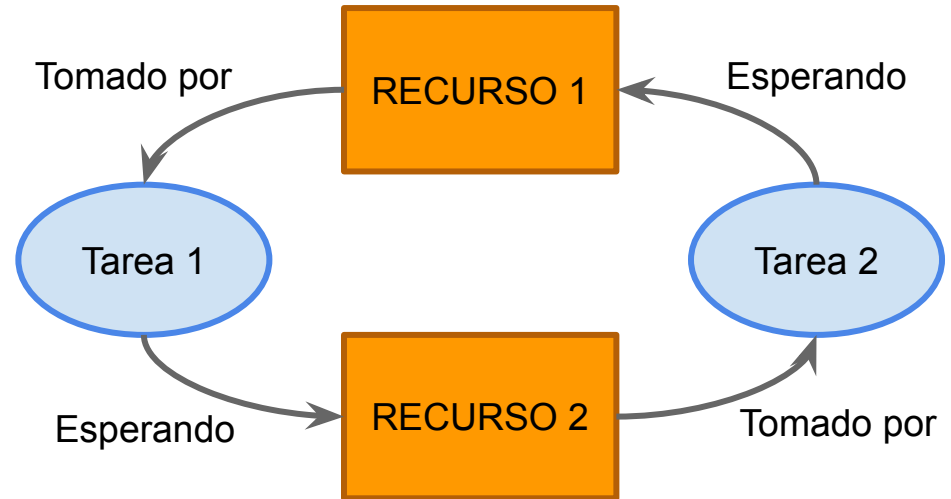
```
Tomar ( S1 );  
Tomar ( S2 );
```

```
Liberar( S2 );  
Liberar( S1 );
```

**Tarea 2**

```
Tomar( S2 );  
Tomar( S1 );
```

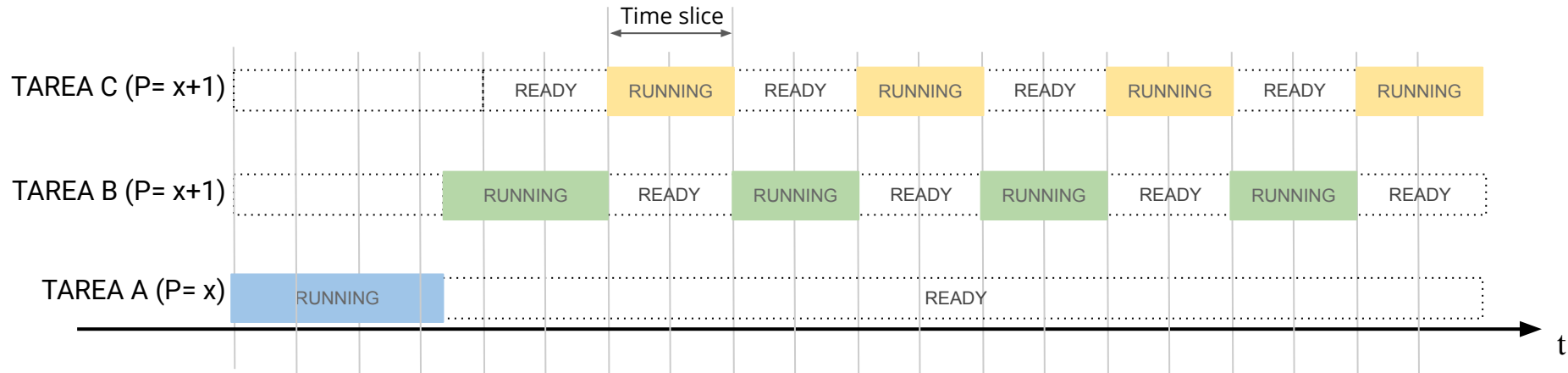
```
Liberar( S1 );  
Liberar( S2 );
```





# Starvation

- Ocurre cuando un proceso permanentemente está negando un recurso a otro para que funcione.
  - ej : 2 tareas de igual prioridad se planifican según un algoritmo de round robin, y otra, de menor prioridad, no obtiene tiempo de CPU nunca.



# Bibliografía

---

- [The FreeRTOS™ Kernel](#)
- [FreeRTOS Kernel Documentation](#)
- Introducción a los Sistemas operativos de Tiempo Real, Alejandro Celery - 2014
- Concurrencia - Sincronización, CAPSE, Franco Bucafusco, 2017
- FreeRTOS - Temporización, Cusos INET, Franco Bucafusco, 2017
- [Condición de carrera](#), Wikipedia, Consultado 19/05/17

# Licencia

---



"Sincronización de Tareas en FreeRTOS (parte 2)"

Por Mg. Ing. Franco Bucafusco, se distribuye bajo una [licencia de Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-sa/4.0/)