Sincronización de procesos

Electrónica IV

Mg.Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

https://facetvirtual.facet.unt.edu.ar/course/view.php?id=165

Problema del Producto-Consumidor

- Una tarea genera datos que deben ser procesador por otra tarea
- El productor genera los datos y los ingresa en una cola
- El consumidor retira los datos de la cola y los procesa

Una posible implementación

```
void Productor(void) {
    while (true) {
        /* Produce un dato */
        while (ocupados == CANTIDAD_MAXIMA);
        cola[entrada] = datoProducido;
        entrada = (entrada + 1) % CANTIDAD_MAXIMA;
        ocupados++;
    }
}

void Consumidor(void) {
    while (true) {
        while (ocupados == 0);
        datoConsumido = cola[salida];
        salida = (salida + 1) % CANTIDAD_MAXIMA;
        ocupados - -;
        /* Consume un dato */
    }
}
```

Una posible implementación

- La implementación más probable

```
ocupados++                      ocupados--
registro1 = ocupados            registro2 = ocupados
registro1 = registro1 + 1       registro2 = registro2 - 1
ocupados = registro1            ocupados = registro2
```

- Una posible ejecución

```
registro1 = ocupados            registro1 = 5
registro1 = registro + 1        registro = 6
registro2 = ocupados            registro2 = 5
registro2 = registro - 1        registro2 = 4
ocupados = registro1            ocupados = 6
ocupados = registro2            ocupados = 4   ¡Error!
```

Problema de sección crítica

- Los procesos se ejecutan en forma concurrente
- El cambio de contexto puede darse en cualquier punto de la ejecución de un proceso
- El acceso concurrente a datos compartidos en forma no controlada puede derivar en problemas de coherencia de datos
- Los fragmentos de código que podrían generar estos problemas se denominan sección crítica

Problema de sección crítica

- Todos los sistemas operativos multitarea ofrecen herramientas para la sincronización de procesos
- Estas permiten asegurar el acceso en forma coordinada para evitar estos problemas
- En FreeRTOS el mecanismo más simple es el *Mutex*

Mecanismo de Exclusión Mútua

- Por cada elemento compartido de crea un “testigo” que indica si el elemento está libre o está siendo utilizado por otra tarea

- Ante de operar con el elemento compartido (ingresar a la sección crítica) la tarea pide el “testigo” correspondiente

- Si ninguna otra tarea está operando con el elemento entonces obtiene el “testigo” y puede continuar la ejecución

Mecanismo de Exclusión Mútua

- Al terminar las operaciones con el elemento compartido la tarea devuelve el “testigo”
- Si una tarea pide un “testigo” que no está disponible se bloquea hasta que el “testigo” sea devuelto por la tarea que lo está ocupando
- Como la espera se realiza en el estado bloqueado no se ocupa el procesador

Interbloqueo o abrazo mortal

- Puede ocurrir cuando dos tareas comparten dos o más recursos

- Una tarea de baja prioridad T1 comienza a ejecutar y ocupa un recurso R1

- A continuación una tarea de mayor prioridad T2 se desbloquea y ocupa un recurso R2

Interbloqueo o abrazo mortal

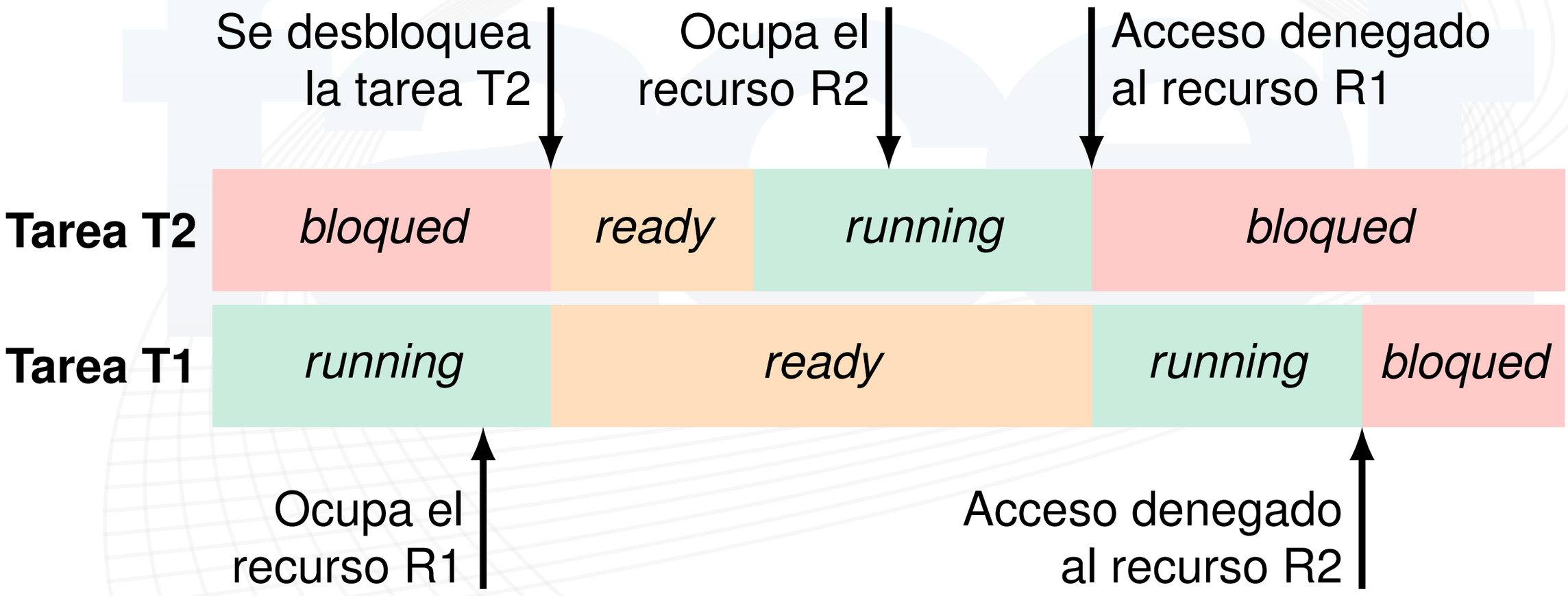
- La tarea de alta prioridad T2 continua ejecutando y, manteniendo ocupado el recurso R2, solicita tambien el recurso R1

- La tarea de alta prioridad T2 se bloquea y vuelve a ejecutar la tarea de baja prioridad T1

- La tarea de baja prioridad T1 mantiene ocupado el recurso R1 y solicita tambien el recurso R2

Interbloqueo o abrazo mortal

- Una vez que se alcanza este estado el sistema no tiene evolución posible !!!



Inversión de prioridades

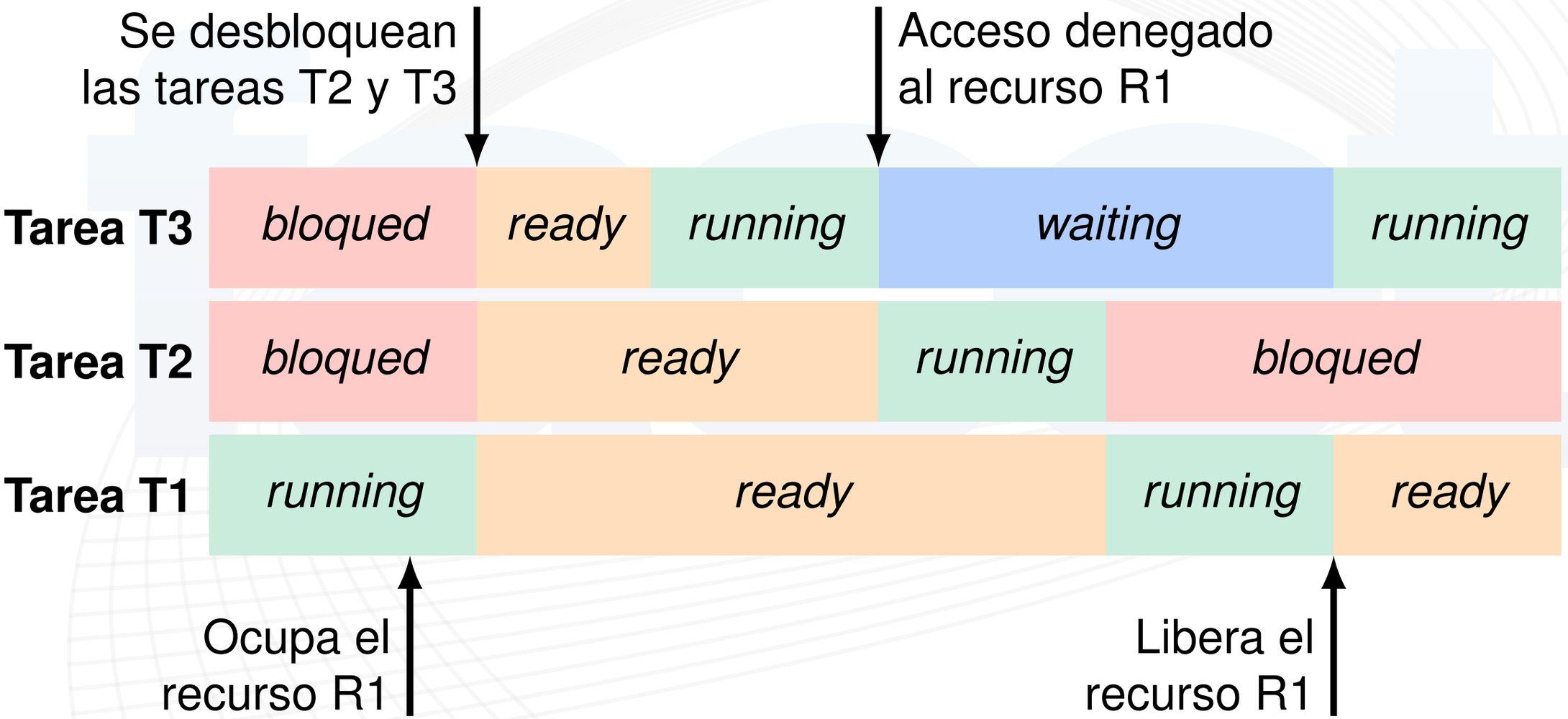
- Puede ocurrir cuando dos tareas comparten un recurso
- Una tarea de baja prioridad T1 está ejecutando y ocupa un recurso R1
- Un evento externo desbloquea dos tareas, una de alta prioridad T3 y una tarea de prioridad media T2

Inversión de prioridades

- La tarea de alta prioridad T3 comienza a ejecutarse y se bloquea al tratar de acceder al recurso R1
- La tarea de prioridad media T2 que no utiliza el recurso R1 comienza a ejecutarse
- Recién cuando termina la tarea de prioridad media T2 se continua con la tarea de baja prioridad T1
- Cuando la tarea de baja prioridad T1 libera el recurso R1 se desbloquea la tarea de alta prioridad T3

Interbloqueo o abrazo mortal

- La tarea de alta prioridad T3 tiene que esperar a la tarea de prioridad media T2



Inversión de prioridades

- Como resultado la tarea de alta prioridad T3 debió esperar la ejecución de la tarea de prioridad media
- Se debería haber ejecutado primero la tarea de baja prioridad T1 para que desocupe el recurso compartido lo antes posible

Sincronización en FreeRTOS

- En FreeRTOS el *Mutex* incluye un mecanismo de herencia de prioridad para minimizar los efectos de la inversión de prioridades
- Cuando una tarea de alta prioridad espera un *Mutex* la tarea que lo ocupa sube temporalmente la prioridad al nivel de la tarea que está lo esta esperando

Funciones para sincronización

- **`xSemaphoreCreateMutex[Static]()`**
  - Crea un nuevo mutex (con herencia de prioridad)
- **`xSemaphoreTake[FromISR]()`**
  - Solicita acceso al al mutex
- **`xSemaphoreGive[FromISR]()`**
  - Libera el acceso del mutex
- **`vSemaphoreDelete()`**
  - Elimina un semáforo o un mutex