Tema 7:

Micro-kernels de tiempo real

Programación de sistemas y dispositivos

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

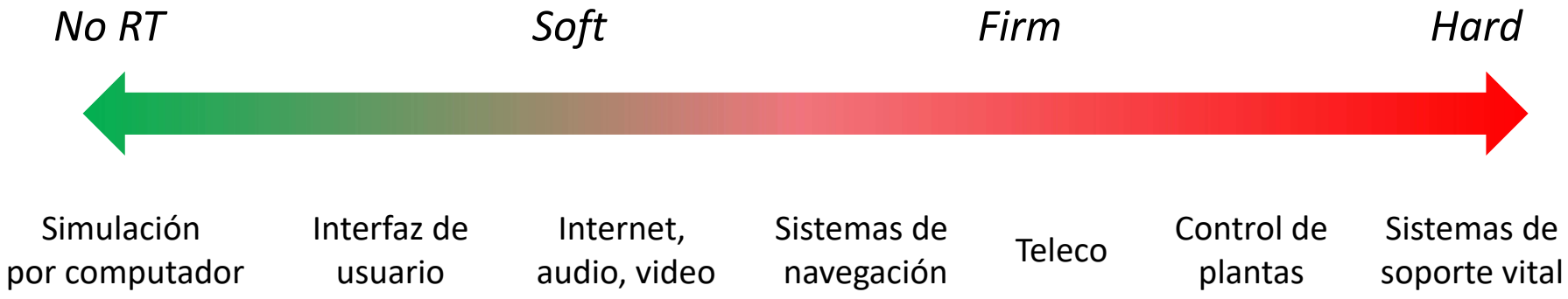
Contenidos

- ✓ Sistemas de tiempo real.
- ✓ Micro-kernels de tiempo real.
- ✓ µC/OS-II.
- ✓ Gestión del kernel.
- ✓ Gestión de tareas.
- ✓ Gestión del tiempo.
- ✓ Comunicación y sincronización.
- ✓ Gestión de memoria.
- ✓ Portado de µC/OS-II
- ✓ Configuración de µC/OS-II.

Sistemas de tiempo real

- Un sistema de tiempo real es un sistema que debe reaccionar ante cambios de su entorno en un plazo máximo (deadline)
  - Su corrección no solo depende de que realice correctamente su función sino también del momento en que produzca sus resultados.
  - Toda tarea de tiempo real tiene asociado un plazo máximo de ejecución.

- Según las consecuencias que provoque la violación de un deadline, las tareas en un sistema de tiempo real se pueden clasificar en:
  - Hard: la no obtención a tiempo de resultado provoca consecuencias catastróficas.
  - Firm: el resultado obtenido es inútil para el sistema, pero no causa daños.
  - Soft: el resultado obtenido es útil para el sistema, pero el rendimiento se degrada.



Sistemas de tiempo real

- Según la periodicidad a la que deban ejecutarse, las tareas en un sistema de tiempo real se pueden clasificar en:
  - Periódicas: se ejecutan frecuentemente a intervalos de tiempo fijos (i.e. muestreado)
  - Aperiódicas: se ejecutan frecuentemente pero a intervalos de tiempo variables que no pueden ser predichos (i.e. interactividad)
  - Esporádicas: se ejecutan raramente o quizás nunca pero tienen gran importancia (i.e. errores hardware, alertas por calentamiento, fallo de suministro eléctrico, etc.)

- El orden en que se ejecutan las tareas lo determina una planificación
  - Un conjunto de tareas es planificable si es posible satisfacer sus requisitos de tiempo
- Los planificadores en tiempo real se clasifican en:
  - Estáticos: siguen una planificación fija precalculada (i.e. Cyclic Executive)
  - Basados en prioridades estáticas: el planificador solo tiene en cuenta las prioridades fijas de las tareas (i.e Rate-Monotonic Scheduling)
  - Basados en prioridades dinámicas: las prioridades van cambiando en función de las características de las tareas y de la evolución de su ejecución (Least Completition Time, Earliest Deadline First, Last Slack Time...)

Sistemas de tiempo real

mejorando la predictibilidad

- Para garantizar el cumplimiento de los deadlines es necesario que:
  - El tiempo de ejecución de las tareas sea conocido al desarrollar la aplicación.
  - Por ello, los sistemas de tiempo real intentan eliminar todas las causas que puedan introducir retardos impredecibles en la ejecución de las tareas (incertidumbre).
- Como norma general para en sistemas de tiempo real se evita:
  - El uso de DMA por robo de ciclo: el uso compartido del bus provocar incertidumbre
    - Es preferible usar time-slice: aunque la ejecución de las tareas sea más lenta su retardo es determinista, ya que DMA y CPU se reparten el tiempo de acceso al bus.
  - El uso de cache: los fallos de cache provocan incertidumbre
    - Además en sistemas expropiativos los cambios de contexto destruyen la localidad
  - El uso de interrupciones para atender dispositivos aperiódicos: pueden interrumpir en instantes no predecibles generando incertidumbre
    - Como alternativa se usa el polling periódico de estos periféricos (por las propias tareas, o por una tarea común del kernel cada una de las tareas)
  - El uso de MMU: el tratamiento de fallos de página provoca incertidumbre.
  - El uso de estructuras de datos dinámicas: su tiempo de acceso es variable
    - Como alternativa se usan estructuras de datos estáticas con tiempo de acceso constante.
  - Incluso el uso de procesadores segmentados o superescalares.

Micro-kernels de tiempo real

- Un micro-kernel de tiempo real es la mínima cantidad de software que permite planificar eficazmente tareas de tiempo real.
  - Orientado a sobrecargar mínimamente al sistema
  - Típicamente está constituido por un planificador expropiativo basado en prioridades y un conjunto de servicios para la gestión y comunicación de tareas.
  - No facilita drivers de dispositivos, sistemas de ficheros, protocolos de red...
    - Si son necesarios se implementan y encapsulan aparte (Board Support Packages) como servicios adicionales.

- Los RTOS usados en sistemas empotrados se construyen añadiendo capas software sobre un micro-kernel
  - Propietario: VxWorks, QNX, RTEMS, ThreadX, OSE, LynxOS, µC/OS-II, µC/OS-III
  - Open Source: FreeRTOS, µClinux

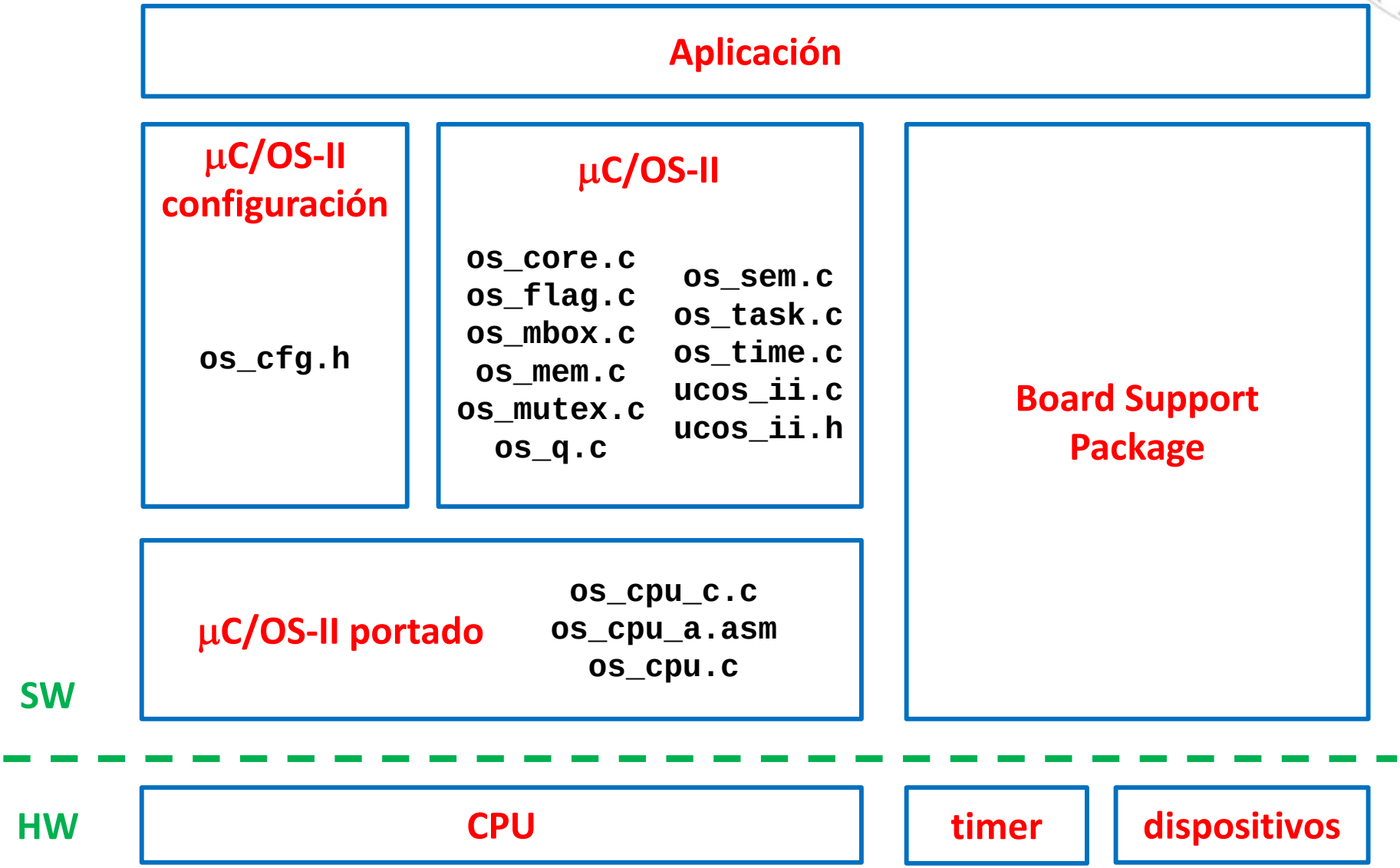
µC/OS-II

- µC/OS-II es un micro kernel de tiempo real
  - Desarrollado en 1992 por Jean J. Labrosse en ANSI C, mantenido por Micrium Inc.
  - RTOS multitarea expropiativo basado en prioridades.
- Características
  - Portable: la mayor parte del código es independiente de plataforma y está en ANSI C
    - El código dependiente de máquina está encapsulado y debe ser escrito en C/ensamblador.
    - Puede portarse a microcontroladores, microprocesadores y DSPs de 8 a 64 bits.
    - Todo el código es abierto pero su uso requiere licencia.
  - ROMable: de reducido footprint (5-24 KB) y puede arrancar desde ROM.
  - Escalable: por compilación condicional permite incluir solo las funcionalidades necesarias.
  - Determinista: el tiempo de ejecución de la mayoría de los servicios es constante, conocido e independiente del número de tareas.
  - Robusto y fiable: certificado para uso comercial en sistemas críticos en seguridad (aviónica, medicina, transporte, etc.).
- Estudiaremos en detalle la versión 2.52
  - **Referencia**: *MicroC/OS-II. The Real-Time Kernel de Jean J. Labrosse*

µC/OS-II

- µC/OS-II ofrece servicios para:
  - Gestionar el kernel de tiempo real
  - Gestionar tareas
  - Gestionar el tiempo
  - Comunicar y sincronizar tareas/RTI
    - Semáforos generales / binarios
    - Grupos de flags de eventos
    - Buzones de mensajes
    - Colas de mensajes
  - Gestionar memoria dinámica
- Todos los servicios se ofrecen como API
  - La aplicación puede usarlos haciendo llamadas a las funciones públicas de µC/OS-II.
  - El número y tipo de servicios disponible es configurable por el programador.
- µC/OS-II se compila junto al resto del software de aplicación/sistema, por lo que deberá completarse con:
  - Un bootloader.
  - Una colección de drivers de dispositivos (Board Support Package).
  - Las tareas y las RTI que forman la aplicación.

µC/OS-II



J.M. Mendías 2016

Gestión del kernel

servicios

- Para gestionar el kernel, µC/OS-II ofrece los siguientes servicios:
  - **`OSInit()`**
    - Inicializa las estructuras de datos internas del kernel y debe ser llamada antes de crear cualquier objeto de µC/OS-II y antes de llamar **`OSStart()`**.
  - **`OSStart()`**
    - Inicia la multitarea en el sistema. Previamente debe haberse creado al menos una tarea.
  - **`OSIntEnter()`**
    - Notifica al kernel que se va ha ejecutar una RTI.
  - **`OSIntExit()`**
    - Notifica al kernel que se ha finalizado la ejecución de una RTI. **`OSIntEnter()`** y **`OSIntExit()`** deben invocarse en pares y permiten que el sistema lleve la cuenta del número de interrupciones anidadas para que cuando finalicen todas pueda invocarse al planificador.
  - **`OSSchedLock()`**
    - Deshabilita el planificador para evitar temporalmente los cambios de contexto.
  - **`OSSchedUnlock()`**
    - Habilita el planificador. **`OSSchedLock()`** y **`OSSchedUnlock()`** deben invocarse en pares.
  - **`OSTimeTick()`**
    - Notifica al kernel la ocurrencia de un tick. Debe ser llamada por la RTI de un temporizador.
  - **`OSVersion()`**
    - Devuelve el número de versión del sistema (multiplicado por 100).
  - **`OSStatInit()`**
    - Inicializa la tarea stat. Debe invocarse desde la primera y única tarea creada antes de **`OSStart()`**. Con posterioridad pueden crearse las restantes tareas.
- El código fuente está disponible en el fichero os_core.c

J.M. Mendías 2016

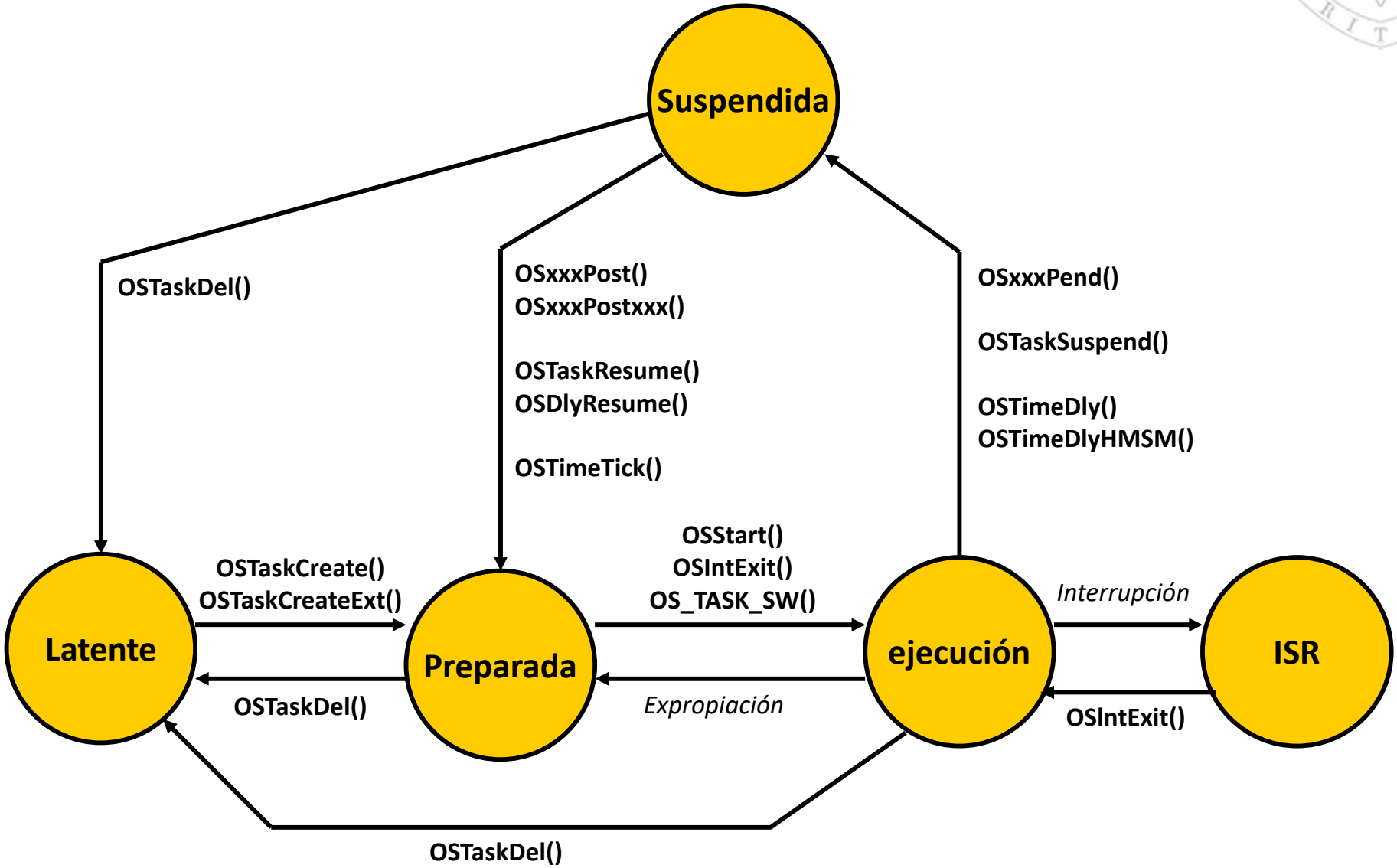
Gestión de tareas

- En µC/OS-II una tarea es la mínima unidad de planificación y tiene asociada una prioridad única
  - Es utilizada como identificador de tarea.
  - Existen como máximo 64 prioridades (64 tareas como máximo)
    - A menor valor numérico, mayor prioridad
    - 8 reservadas por el sistema: las 4 más altas y las 4 más bajas (56 tareas de aplicación)
    - Las prioridades pueden cambiarse de manera explícita en tiempo de ejecución.
  - El sistema siempre está ejecutando la tarea preparada de más alta prioridad
  - El sistema crea 2 tareas:
    - Idle (prioridad mínima: **`OS_LOWEST_PRIO`**) : pasa a ejecución cuando no hay ninguna otra tarea preparada. No puede ser eliminada.
    - Stat (prioridad **`OS_LOWEST_PRIO`**-1) : tarea opcional encargada de recopilar datos estadísticos de uso de la CPU.
  - Cada tarea tiene su propia pila gestionada por el sistema
    - El tamaño de la pila es fijo pero puede ser distinto para cada tarea.
    - El sistema ofrece servicios para conocer el tamaño requerido por una tarea.
    - El contexto de cada tarea se almacena en su pila.
  - El estado de cada tarea está almacenado en TCB (task control block)
    - Estado, puntero a cima de pila, prioridad...

J.M. Mendías
2016

Gestión de tareas

estado de una tarea



Gestión de tareas

servicios

- Para gestionar tareas, µC/OS-II ofrece los siguientes servicios:
  - `OSTaskCreate() / OSTaskCreateExt()`
    - Crea una tarea. Las tareas pueden crearse antes de arrancar la multitarea o por una tarea en ejecución. Una tarea no puede ser creada por una RTI.
  - `OSTaskDel()`
    - Borra una tarea. Una tarea puede borrarse a sí misma. La tarea borrada pasa a estado latente (dormand) y puede ser reactivada creándola de nuevo.
  - `OSTaskDelReq()`
    - Solicita a una tarea que se borre a sí misma previa liberación de los recursos que tiene asignados (bloques de memoria, semáforos, buzones, queues etc.).
  - `OSTaskChangePrio()`
    - Cambia dinámicamente la prioridad de una tarea. La nueva prioridad debe estar disponible.
  - `OSTaskSuspend() / OSTaskResume()`
    - Suspende/reanuda una tarea.
  - `OSTaskStkChk()`
    - Permite conocer la cantidad de memoria disponible en la pila de una tarea.
  - `OSTaskQuery()`
    - Permite conocer información de una tarea obteniendo una copia de su TCB.
- El código fuente está disponible en el fichero os_task.c

J.M. Mendías 2016

Gestión de tareas

programación

- Desde el punto de vista del programador de aplicaciones, una tarea es una función void de 1 argumento en C que:
  - Contiene un bucle infinito
  - Se auto elimina antes de finalizar

```
void task( void *pdata )
{
  while( 1 )
  {
    ... codigo...
  }
}
```

```
void task( void *pdata )
{
  ... codigo...
   OSTaskDel( PRIO_SELF );
}
```

- El argumento se pasa durante la creación de la tarea
  - Al ser un puntero void, en la práctica, puede pasarse a la tarea cualquier tipo de información.
- Las variables locales de la tarea conservan su valor durante todo el tiempo en que la tarea esté viva.
  - No es necesario declararlas como static, porque la tarea nunca finaliza.

J.M. Mendías
2016

Gestión de tareas

arranque de la multitarea

```
OS_STK TaskStartStk[TS_STK_SIZE];
OS_STK Task1Stk[T1_STK_SIZE];
...
void TaskStart( void *pdata );
void Task1( void *pdata );
...

extern void OSTickISR( void );

void main( void )
{
  sys_init();
  device_init();

  OSInit();
  OSTaskCreate( TaskStart, NULL, &TaskStartStk[TS_STK_SIZE - 1], 0 );
  OSStart();
}
void TaskStart( void *pdata )
{
  OS_ENTER_CRITICAL();
  timer0_open_tick( OSTickISR, 10 );
  OS_EXIT_CRITICAL();

  OSStatInit();
  OSTaskCreate( Task1, NULL, &Task1Stk[T1_STK_SIZE - 1], TASK1_PRIO );
  ...
  OSTaskDel(OS_PRIO_SELF);
}
```

Declara las pilas de las tareas (cada una con su tamaño)

Declara prototipos de las tareas

Declara el prototipo de la RTI escrita en ensamblador

Inicializa µC/OS-II

Arranca multitarea

Crea la tarea inicial

Instala OSTickISR como ISR de un temporizador y arranca la generación de ticks cada 10 ms, siempre después de OStart()

Opcionalmente, inicia la tarea stat, siempre antes de crear las tareas de la aplicación

Crea el resto de tareas

La tarea inicial se auto elimina

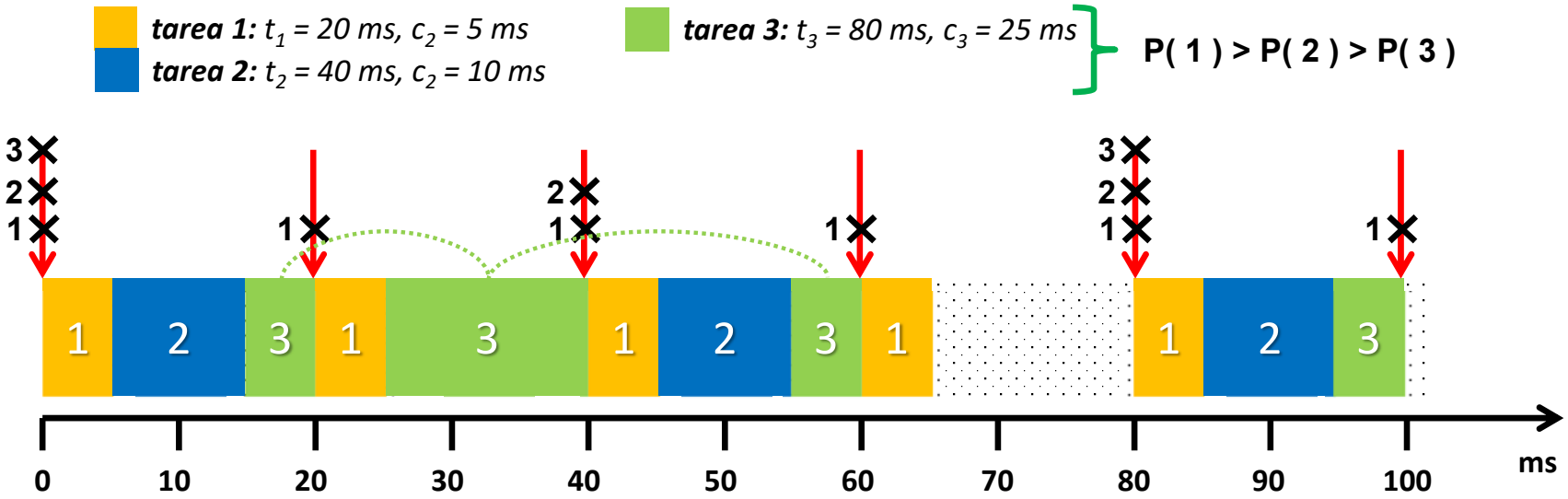
J.M. Mendías 2016

Gestión de tareas

asignación de prioridades

- La asignación de prioridades es un tarea no trivial, en general:
  - Tareas más críticas deben tener prioridades mas altas.

- Cuando las tareas son periódicas, un punto interesante de partida es basarse en la planificación Rate Monotonic.
  - Cuanto más frecuentemente deba ejecutarse una tarea, mayor prioridad debe tener.
  - Al ser expropiativa, siempre ejecuta aquella tarea de las preparadas para la que dispone de menor plazo para su ejecución.



Gestión de tareas

planificación Rate Monotonic

- Bajo ciertas simplificaciones RM es una planificación estática óptima.
  - Todas las tareas son periódicas.
  - Son independientes, no comparten recursos, ni se sincronizan, ni se excluyen mutuamente.
  - Su deadline es igual a su periodo.

- Un sistema RM será planificable si:

$$\sum_{i=1}^{n} \frac{c_i}{t_i} \leq n(2^{1/n} - 1)$$

- Donde:
  - t_i: periodo de activación (cada cuanto tiempo debe ejecutarse)
  - c_i: tiempo máximo de cómputo
  - Se asume que toda tarea dispone de un plazo de ejecución equivalente a su periodo, es decir, que entre 2 activaciones puede ejecutarse en cualquier momento.

Gestión del tiempo

- µC/OS-II dispone de un contador de ticks de 32 bits.
  - Es actualizado por la función **`OSTickISR()`** que debe ser instalada como RTI de una interrupción periódica de frecuencia entre 10 y 100 Hz.
  - Para que el sistema pueda correlacionar el número de ticks con el tiempo físico hay que definir adecuadamente la macro **`OS_TICKS_PER_SEC`**

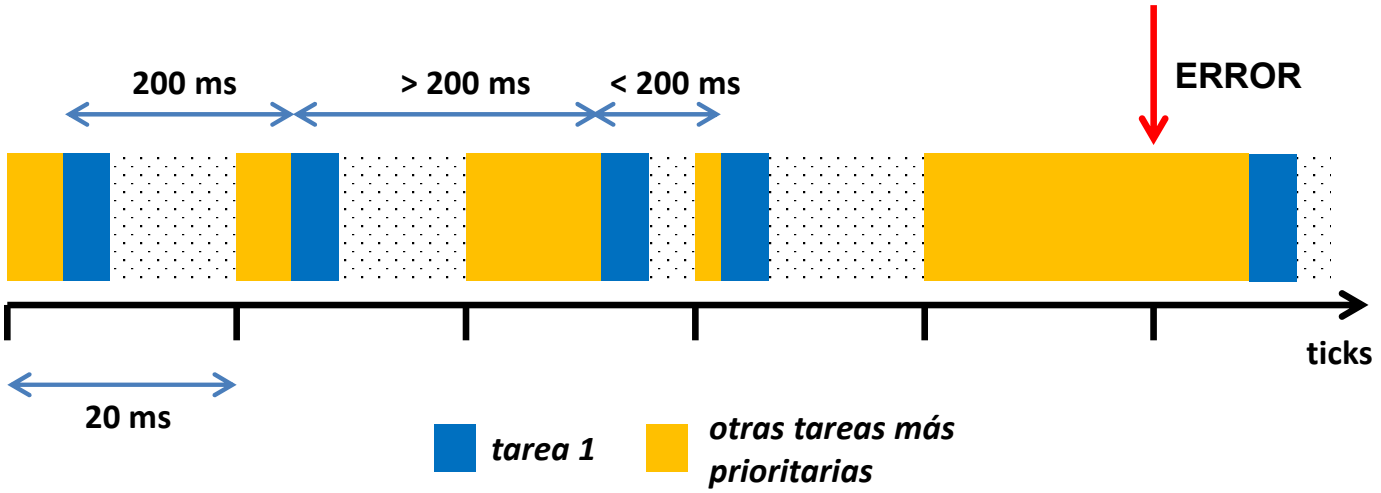
- Para gestionar el tiempo, µC/OS-II ofrece los siguientes servicios:
  - **`OSTimeDly()`**
    - Suspende la ejecución de la tarea invocante un número de ticks de reloj.
  - **`OSTimeDlyHMSM()`**
    - Suspende la ejecución de la tarea invocante un tiempo determinado.
  - **`OSTimeDlyResume()`**
    - Reanuda la ejecución de una tarea previamente suspendida por las funciones **`OSTimeDly()/OSTimeDlyHMSM()`**. No puede ser usada para reanudar tareas a la espera de un evento o timeout.
  - **`OSTimeGet()`**
    - Devuelve el valor contenido en el contador de ticks.
  - **`OSTimeSet()`**
    - Almacena un valor en el contador de ticks.
- El código fuente está disponible en el fichero os_time.c

Gestión del tiempo

programación

- No olvidar incluir **`OSTimeDly()`**en tareas que sean bucles infinitos
  - Asegura que las tareas de menor prioridad tengan posibilidad de ejecutarse
- Recordar que la llamada a la función **`OSTimeDly(n)`** no asegura que la tarea necesariamente vuelva a ejecutarse pasado un tiempo fijo.
  - Solo asegura que pasará a preparada transcurridos n ticks y se ejecutará según la carga del sistema.
    - Si el retardo en peor caso de todas las tareas de mayor prioridad es menor que el periodo de tick, la tarea se ejecutará con periodicidad fija pero con slack variable.
    - Si el retardo en peor caso de todas las tareas puede ser mayor que el periodo de ticks, el sistema fallará.

```
void Task1( void )
{
  while( 1 )
  {
     ...
     OSTimeDly( 1 );
  }
}
```



Comunicación y sincronización

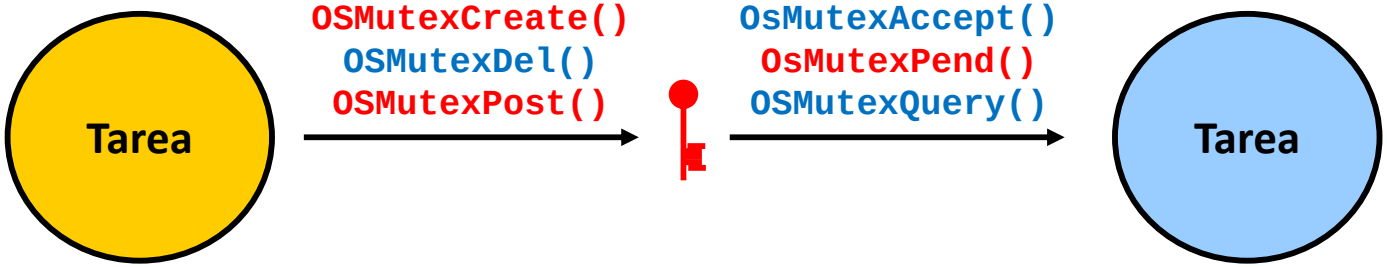
- En µC/OS-II la comunicación y sincronización entre tareas/RTI se realiza a través de distintos tipos de mecanismos:
  - Semáforos mutex
  - Semáforos generales
  - Flags de eventos
  - Buzones de mensajes
  - Colas de mensajes

- Para todos ellos µC/OS-II ofrece una colección homogénea de servicios
  - **`Create`**: crea el canal de comunicación/sincronización
  - **`Delete`**: elimina el canal de comunicación/sincronización
  - **`Pend / Accept`**: espera (wait) por un evento con posibilidad de timeout en versiones bloqueante y no.
  - **`Post`**: señaliza (signal) un evento
  - **`Query`**: obtiene información relativa al evento

- El sistema gestiona los eventos a través del ECB (Event Control Block)

Gestión de semáforos mutex

- Un semáforo mutex en μC/OS-II es:
  - Un semáforo binario que implementa un mecanismo de herencia de prioridad para prevenir el problema de la inversión de prioridad.
  - Accesible a través de 6 servicios (3 de ellos opcionales) ofrecidos por el sistema.
- Se usa para:
  - Acceder de manera mutuamente exclusiva a un recurso compartido

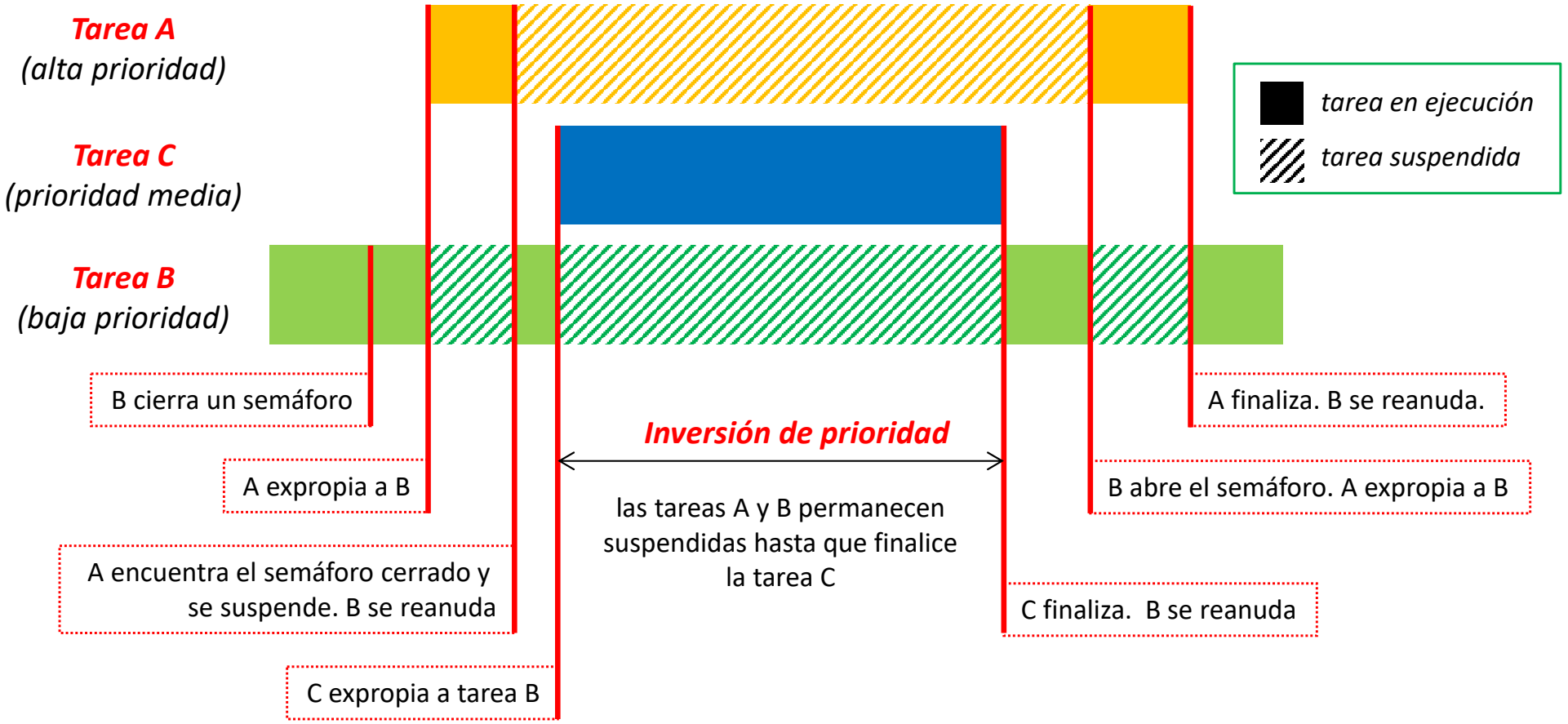


servicio siempre disponible / servicio disponible opcionalmente

Gestión de semáforos mutex

el problema de la inversión de prioridad (i)

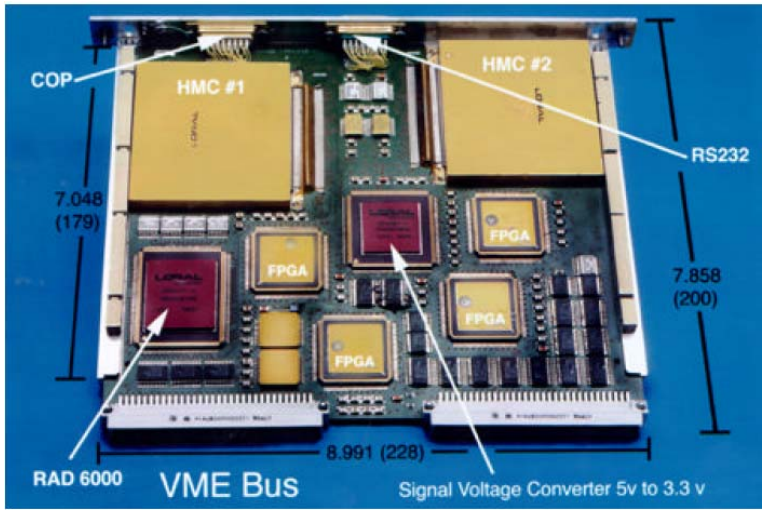
- Se dice que hay una inversión de prioridad cuando:
  - Una tarea de alta prioridad (A) queda a la espera de un recurso tomado por una tarea de baja prioridad (B) que a su vez es expropiada por una tercera más prioritaria (C).

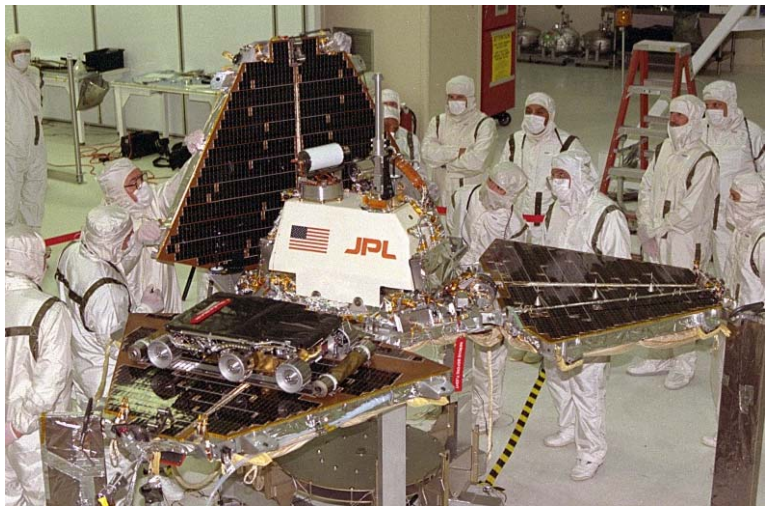


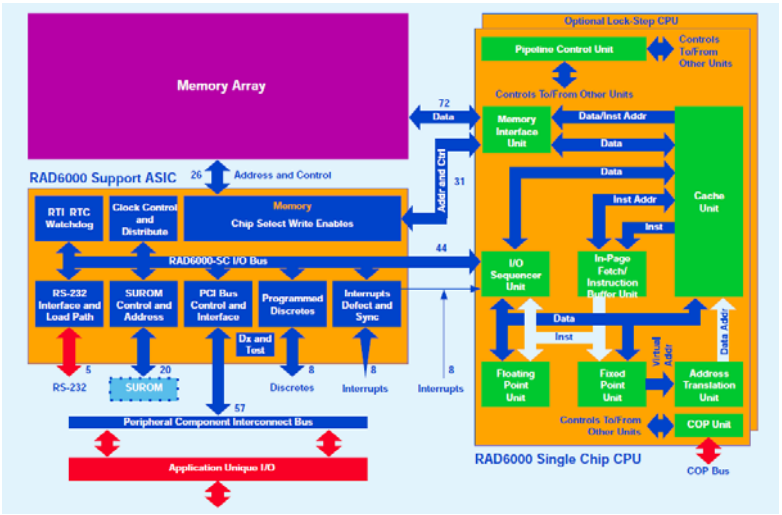
Gestión de semáforos mutex

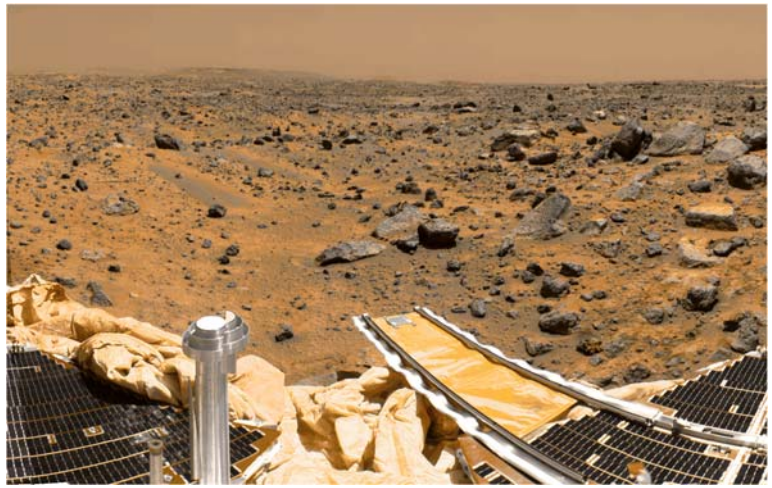
el problema de la inversión de prioridad (ii)

- El caso del Mars pathfinder que aterrizó en Marte el 4 de julio de 1997:
  - Disponía de un single board computer: RAD6000 (20 MHz), 2 MB ROM, 128 DRAM.









J.M. Mendías 2016

Gestión de semáforos mutex

el problema de la inversión de prioridad (iii)

- La arquitectura SW (que usaba el RTOS VxWorks) disponía:
  - Un "information bus"
    - Una región de memoria compartida protegida con un semáforo mutex
    - Usada para pasar información entre diferentes componentes del sistema.
  - Tarea de gestión del "information bus"
    - Movía datos del "information bus" y se ejecutaba con mucha frecuencia y alta prioridad.
  - Tarea de recopilación de datos meteorológicos
    - Publicaba datos en el "information bus" y se ejecutaba con baja frecuencia y baja prioridad.
  - Tarea de comunicación
    - Tenía prioridad media y una larga duración.

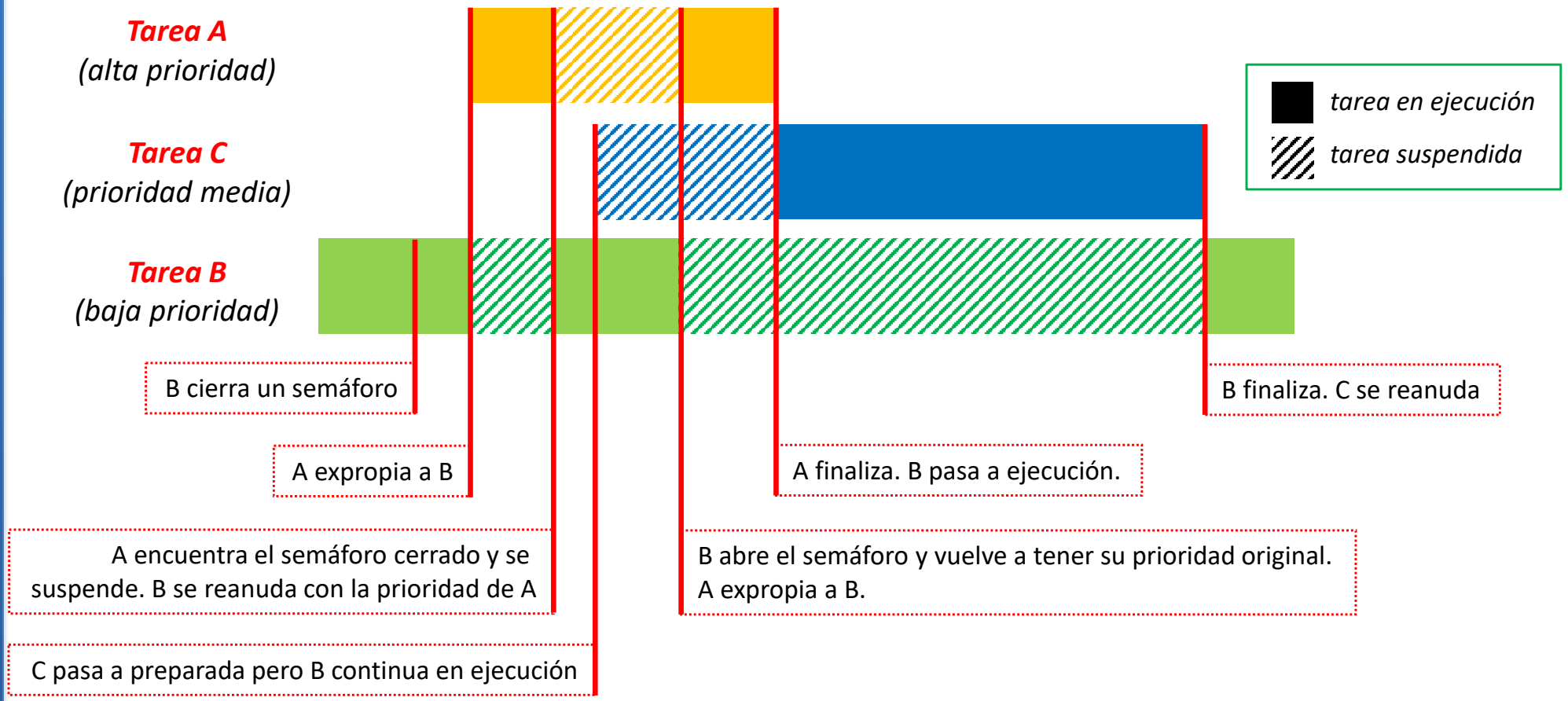
- A los pocos días de empezar a recolectar datos meteorológicos, se detectó que el sistema se reseteaba periódicamente:
  - La tarea de recopilación tomaba el mutex, era expropiada por la de gestión que al encontrar el mutex cerrado se bloqueaba y se planificaba la de comunicación.
  - Cuando el watchdog finalizaba la cuenta, se detectaba que la tarea de gestión no había sido ejecutada hace tiempo y reseteaba el sistema.

J.M. Mendías 2016

Gestión de semáforos mutex

protocolo de herencia de prioridad

- Para solventarlo puede usarse un protocolo de herencia de prioridad:
  - Mientas que una tarea dispone de un recurso hereda la máxima de las prioridades de aquellas tareas (de mayor prioridad) que estén a la espera de dicho recurso.



J.M. Mendías
2016

Gestión de semáforos mutex

servicios

- Para gestionar semáforos mutex, µC/OS-II ofrece los siguientes servicios:
  - `OSMutexCreate()`
    - Crea un semáforo inicialmente abierto (inicializado a 1).
  - `OSMutexDel()`
    - Elimina un semáforo pasando a preparadas todas las tareas que estén a la espera del mismo. Típicamente, deberían eliminarse antes todas las tareas que lo usan.
  - `OSMutexPend()`
    - (Wait) Si el semáforo está abierto, lo cierra; si el semáforo está cerrado, suspende a la tarea invocante.
  - `OSMutexPost()`
    - (Signal) Si hay tareas a la espera del semáforo, pasa a preparada la más prioritaria (pudiendo expropiar a la invocante); en caso contrario abre el semáforo.
  - `OSMutexAccept()`
    - (Wait no bloqueante) Cierra el semáforo si estuviera abierto.
  - `OSMutexQuery()`
    - Permite conocer información de un semáforo.
- El código fuente está disponible en el fichero os_mutex.c

J.M. Mendías 2016

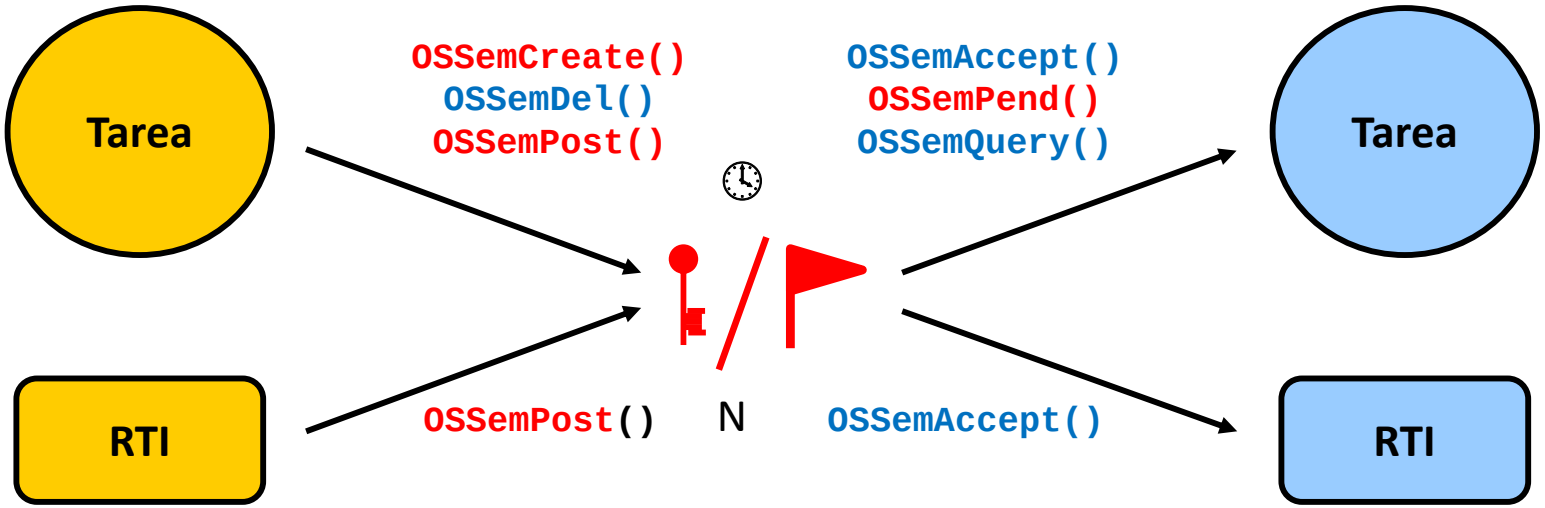
Gestión de semáforos mutex

acceso a secciones críticas

- En µC/OS-II existen 3 métodos para que las tares protejan sus secciones críticas:
  - Deshabilitando temporalmente interrupciones
    - **`OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL()`**
    - Impide que la ejecución de una porción de código pueda ser expropiada por una ISR o una tarea. Este mecanismo es el usado por el kernel.
    - Su uso es peligroso: si se deshabilitan antes de **`OSTimeDly()`** o antes de una operación PEND el sistema se bloquea.
  - Deshabilitando temporalmente el planificador
    - **`OSSchedLock()`/`OSSchedUnlock()`**
    - A diferencia del anterior solo evita que una porción de código pueda ser expropiada por una tarea. Las ISR podrán seguir interrumpiendo.
    - Se usa para cuando una tarea de bajo nivel tiene que mandar POST y no quiere ser interrumpida.
    - Su uso es peligroso: si se deshabilita el planificador antes de una operación PEND o una suspensión, el sistema se bloquea.
  - Utilizando un semáforo mutex
    - **`OSMutexPend() / OSMutexPost()`**
    - Solo afecta a las ISR o tareas que utilicen el semáforo.

Gestión de semáforos

- Un semáforo (general) en µC/OS-II es:
  - Un contador de 16b y una lista de tareas a la espera que el contador valga mas de 0.
  - Accesible a través de 6 servicios (3 de ellos opcionales) ofrecidos por el sistema.
- Se usa para:
  - Sincronizar el acceso compartido a N recursos.
  - Señalizar la ocurrencia de N eventos.



servicio siempre disponible / servicio disponible opcionalmente

Gestión de semáforos

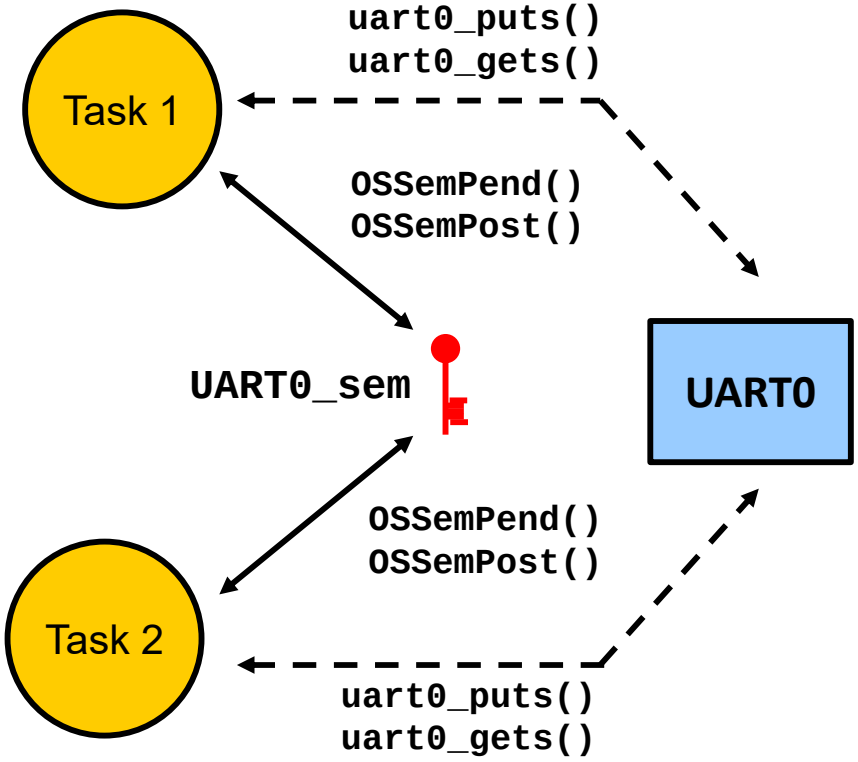
servicios

- Para gestionar semáforos, µC/OS-II ofrece los siguientes servicios:
  - **`OSSemCreate()`**
    - Crea un semáforo indicando su valor inicial.
  - **`OSSemDel()`**
    - Elimina un semáforo pasando a preparadas todas las tareas que estén a la espera del mismo. Típicamente, deberían eliminarse antes todas las tareas que lo usan.
  - **`OSSemPend()`**
    - (Wait) Si el semáforo vale 0, suspende la tarea invocante; en caso contrario, decrementa su valor.
  - **`OSSemPost()`**
    - (Signal) Si hay tareas a la espera del semáforo, pasa a preparada la más prioritaria (pudiendo expropiar a la invocante); en caso contrario incrementa su valor.
  - **`OSSemAccept()`**
    - (Wait no bloqueante) Decrementa el valor del semáforo si vale más de 0.
  - **`OSSemQuery()`**
    - Permite conocer información de un semáforo.
- El código fuente está disponible en el fichero os_sem.c

Gestión de semáforos

acceso a recursos compartidos (i)

- Cualquier acceso a un recurso compartido por varias tareas debe estar protegido por un semáforo creado inicialmente abierto.

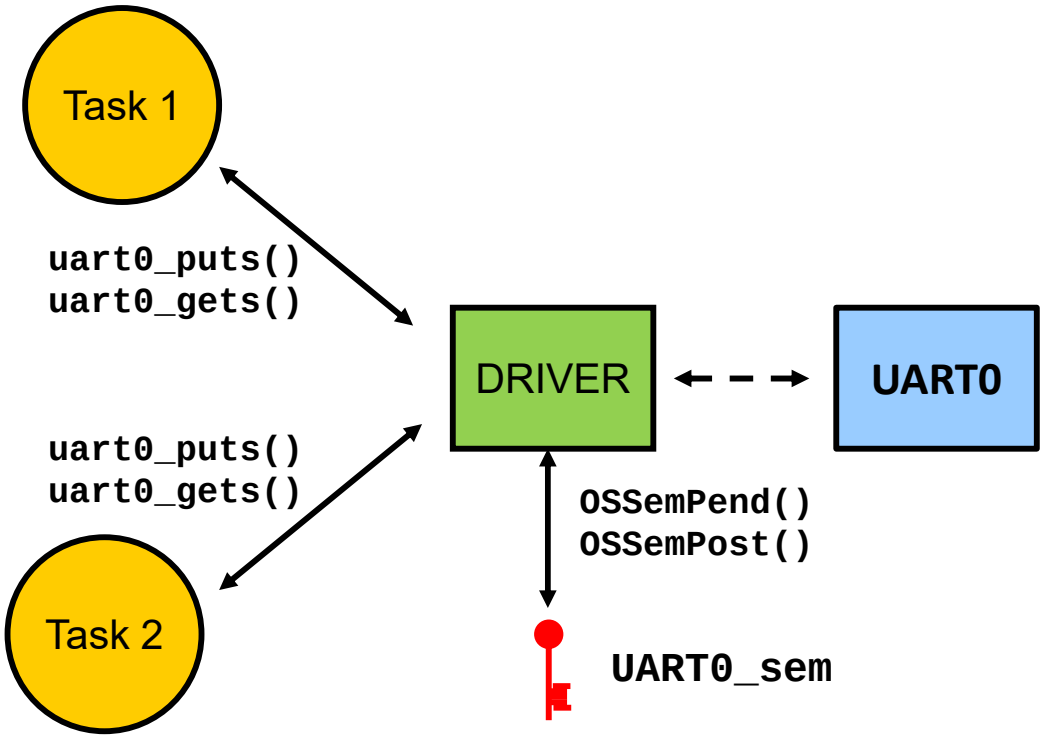



```
OS_EVENT *uart0Ssem;
...
void main( void )
{
  ...
  OSInit();
  uart0Sem = OSSemCreate( 1 );
  ...
  OSStart();
}
void Task1( void )
{
  ...
  OSSemPend( uart0Sem, 0, &err );
  uart0_puts( "Mensaje de la Tarea1\n" );
  OSSemPost( uart0Sem );
  ...
}
void Task2( void )
{
  ...
  OSSemPend( uart0Sem, 0, &err );
  uart0_puts( "Mensaje de la Tarea2\n" );
  OSSemPost( uart0Sem );
  ...
}
```

Gestión de semáforos

acceso a recursos compartidos (ii)

- No obstante, para evitar que el programador de aplicaciones tenga que ocuparse de la gestión del semáforo, mejor encapsularlo
  - Según el servicio usado del SO, el recurso dejará de poderse usar en RTI




```
static OS_EVENT *UART0_sem;
...
void uart0_init( void )
{
  ...
  UART0_sem = OSSemCreate( 1 );
  ...
}


void uart0_puts( char *s )
{
  INT8U err;

  OSSemPend( UART0_sem, 0, &err );
  while( *s )
    uart0_putchar( *s++ );
  OSSemPost( UART0_sem );

}
```

J.M. Mendias
2016

Gestión de semáforos

acceso a recursos compartidos (ii)

- Incluso pueden desarrollarse en el BSP nuevas primitivas para la toma/devolución de recursos completos.

```
void Task1( void )
{
  ...
  uart0_acquire();
  uart0_puts( "Mensaje de la Tarea1\n" );
  uart0_release();
  ...
}
void Task2( void )
{
  ...
  uart0_acquire();
  uart0_puts( "Mensaje de la Tarea2\n" );
  uart0_release();
  ...
}
```

```
static OS_EVENT *UART0_sem;
...
void uart0_init( void )
{
  ...
  UART0_sem = OSSemCreate( 1 );
  ...
}

void uart0_acquire( void )
{
  INT8U err;

  OSSemPend( UART0_sem, 0, &err );
}

void uart0_release( void )
{
  OSSemPost( UART0_sem );
}
```

Gestión de semáforos

deadlock

- Siempre que hay acceso mutuamente exclusivo a recursos compartidos hay riesgo de interbloqueo
  - µC/OS-II no facilita ningún mecanismo de evitación/recuperación de deadlocks

```
void task( void *pdata )
{
  ... codigo...
   OSSemPend( sem1, 0, &err )
   OSSemPend( sem2, 0, &err )
}
```

```
void task( void *pdata )
{
  ... codigo...
   OSSemPend( sem2, 0, &err )
   OSSemPend( sem1, 0, &err )
}
```

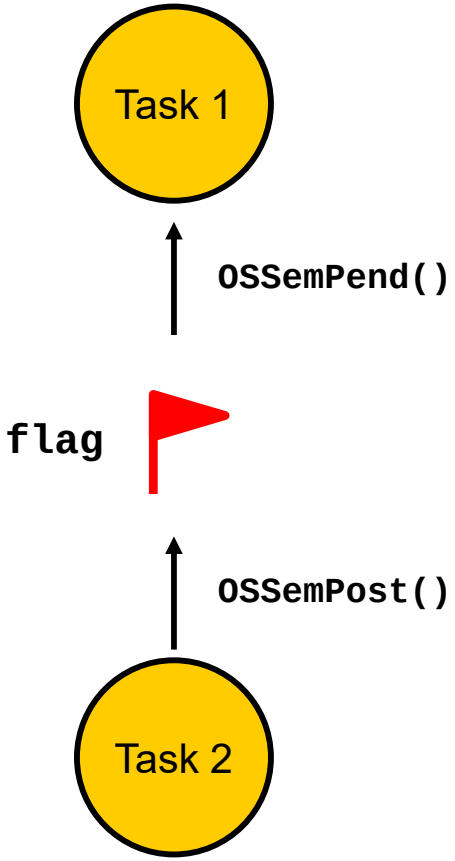
- La aplicación deberá evitarlos explícitamente:
  - Adquiriendo todos los recursos antes de proceder
  - Adquiriendo todos los recursos siempre en el mismo orden
  - Devolviendo siempre los recursos en el orden inverso a como se tomaron
  - Adicionalmente es buena práctica poner un timeout distinto de 0 en todas las PEND y hacer gestión de errores

J.M. Mendías 2016

Gestión de semáforos

sincronización (i)

- Los flags pueden implementarse con semáforos inicializados a 0
  - No olvidar que cuando una RTI necesita obtener un semáforo debe usar siempre el servicio no bloqueante **`OSSemAccept()`**




```
OS_EVENT *flag;
...
void main( void )
{
  ...
  OSInit();
  flag = OSSemCreate( 0 );
  ...
  OSStart();
}
void Task1( void )
{
  ...                              ...
  OSSemPend( flag, 0, &err );      if( OSSemAccept( flag ) > 0 )
  ... procesa ...                    ... procesa ...
}
void Task2( void )
{
  ...
  OSSemPost( flag );
  ...
}
```

chequea

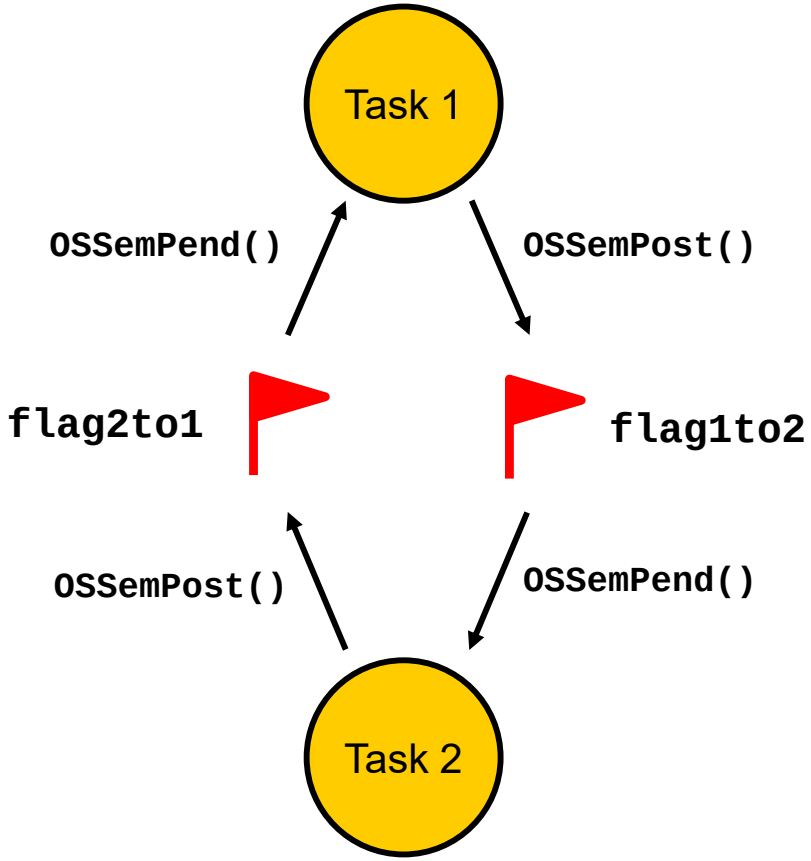
señaliza

J.M. Mendías
2016

Gestión de semáforos

sincronización (ii)

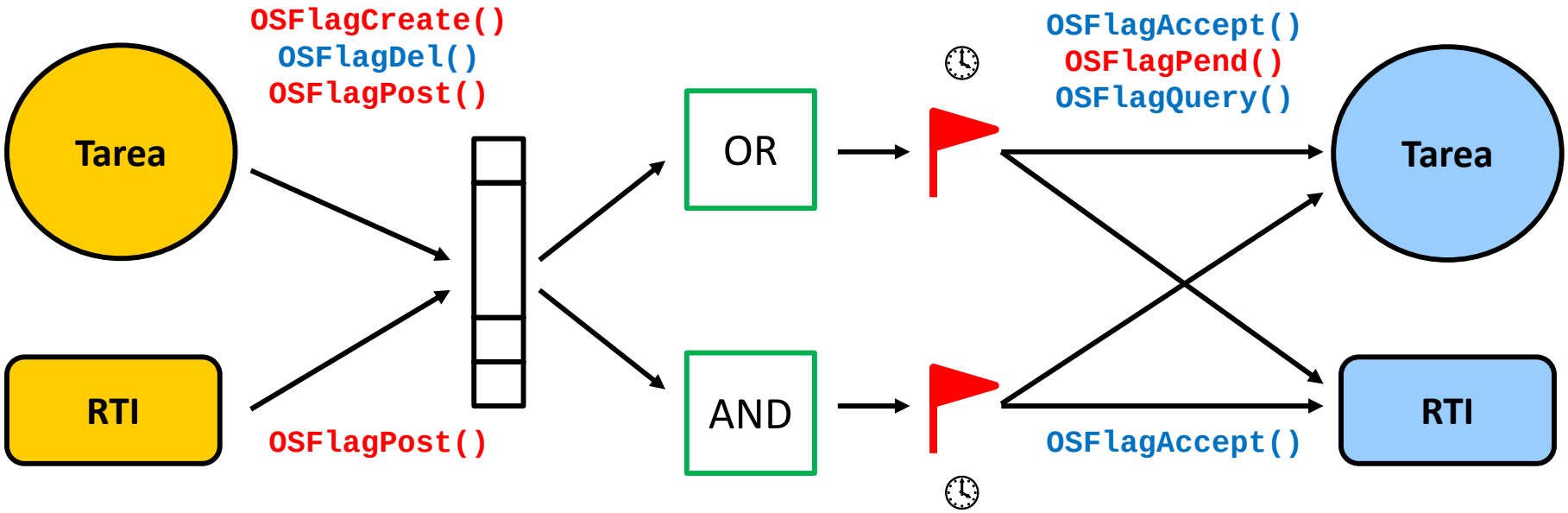
- Parejas de flags pueden usarse para implementar un rendevouz entre 2 tareas.




```
OS_EVENT *flag1to2;
OS_EVENT *flag2to1;
...
void main( void )
{
  OSInit();
  flag1to2 = OSSemCreate( 0 );
  flag2to1 = OSSemCreate( 0 );
  ...
  OSStart();
}
void Task1( void )
{
  ...
  OSSemPost( flag1to2 );
  OSSemPend( flag2to1, 0, &err );
  ...
}
void Task2( void )
{
  ...
  OSSemPost( flag2to1 );
  OSSemPend( flag1to2, 0, &err );
  ...
}
```

Gestión de flags de eventos

- Un grupo de flags de eventos en µC/OS-II es:
  - Un array de bits (8, 16 o 32) que almacenan el estado de los flags y una lista de tareas a la espera de combinaciones (todos/alguno) de valores (0/1) de los flags.
  - Accesible a través de 6 servicios (3 de ellos opcionales) ofrecidos por el sistema.
- Se usa para sincronizar tareas con la ocurrencia de múltiples eventos:
  - Sincronización disyuntiva: espera la ocurrencia de cualquiera de ellos (OR).
  - Sincronización conjuntiva: espera la ocurrencia de todos ellos (AND).



J.M. Mendías 2016

Gestión de flags de eventos

servicios

- Para gestionar flags de eventos, µC/OS-II ofrece los siguientes servicios:
  - `OSFlagCreate()`
    - Crea un grupo de flags indicado su estado inicial.
  - `OSFlagDel()`
    - Elimina un grupo de flags pasando a preparadas todas las tareas que estén a la espera del mismo. Típicamente, deberían eliminarse antes todas las tareas que lo usan.
  - `OSFlagPend()`
    - Espera (wait) por una combinación dada (todos/alguno) de valores (set/clear) de los flags del grupo. Si la combinación no se satisface, se suspende la tarea invocante.
  - `OSFlagPost()`
    - Activa o desactiva algunos flags del grupo. Pasa a preparadas todas las tareas esperando la combinación resultante (pudiendo expropiar a la invocante).
  - `OSFlagAccept()`
    - Espera (wait no bloqueante) por una combinación dada de valores de los flags del grupo sin suspender la tarea invocante en caso de que la condición no se satisfaga.
  - `OSFlagQuery()`
    - Permite conocer el valor de un grupo de flags.
- El código fuente está disponible en el fichero os_flag.c

J.M. Mendías
2016

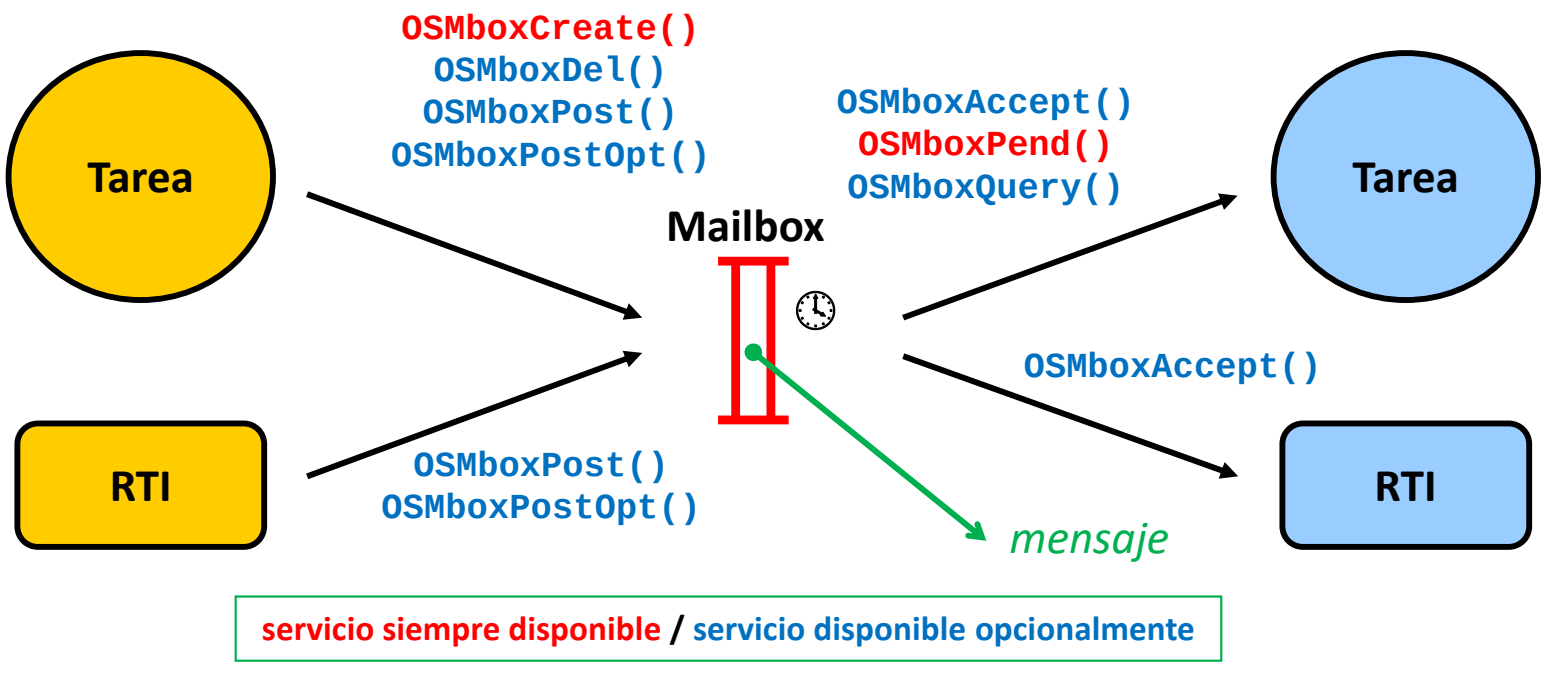
Gestión de flags de eventos

programación

```
OS_FLAG_GRP *flagSet;                                   Declara el grupo de flags
...
void main( void )
{
  OSInit();
  flagSet = OSFlagCreate( 0, &err );                    Crea el grupo de flags de eventos
  ...
  OSStart();
}
void task1( void )
{
  ...
  OSFlagPost( flagSet, bits, OS_FLAG_SET, &err );       Activa los bits indicados
  ...
}
void task2( void )
{
  ...
  OSFlagPend( flagSet, bits, OS_FLAG_WAIT_SET_ALL, 0, &err );   Espera a que todos los bits
  ...                                                           indicados se activen
}
```

Gestión de buzones de mensajes

- Un buzón de mensajes en µC/OS-II es:
  - Un puntero al mensaje y una lista de tareas a la espera de recibir un mensaje.
  - Accesible a través de 7 servicios (3 de ellos opcionales) ofrecidos por el sistema.
- Se usa para:
  - Comunicar varias tareas/RTI entre sí



J.M. Mendías 2016

Gestión de buzones de mensajes

servicios

- Para gestionar buzones, µC/OS-II ofrece los siguientes servicios:
  - `OSMboxCreate()`
    - Crea un buzón de mensajes.
  - `OSMboxDel()`
    - Elimina un buzón de mensajes. Típicamente, deberían eliminarse antes todas las tareas que lo usan.
  - `OSMboxPend()`
    - (Receive) Si hay un mensaje en el buzón, lo toma; si el buzón está vacío, suspende a la tarea invocante.
  - `OSMboxPost()`
    - (Post) Si hay tareas a la espera del mensaje, lo recibe la más prioritaria y pasa a preparada (pudiendo expropiar a la invocante); en caso contrario pone el mensaje en el buzón.
  - `OSMboxPostOpt()`
    - (Broadcast post) Si hay tareas a la espera del mensaje, lo reciben todas y pasan a preparadas (pudiendo expropiar a la invocante); en caso contrario pone el mensaje en el buzón.
  - `OSMboxAccept()`
    - (Receive no bloqueante) Si hay un mensaje en el buzón, lo toma.
  - `OSMboxQuery()`
    - Permite conocer información de un buzón.
- El código fuente está disponible en el fichero os_mbox.c

Gestión de buzones de mensajes

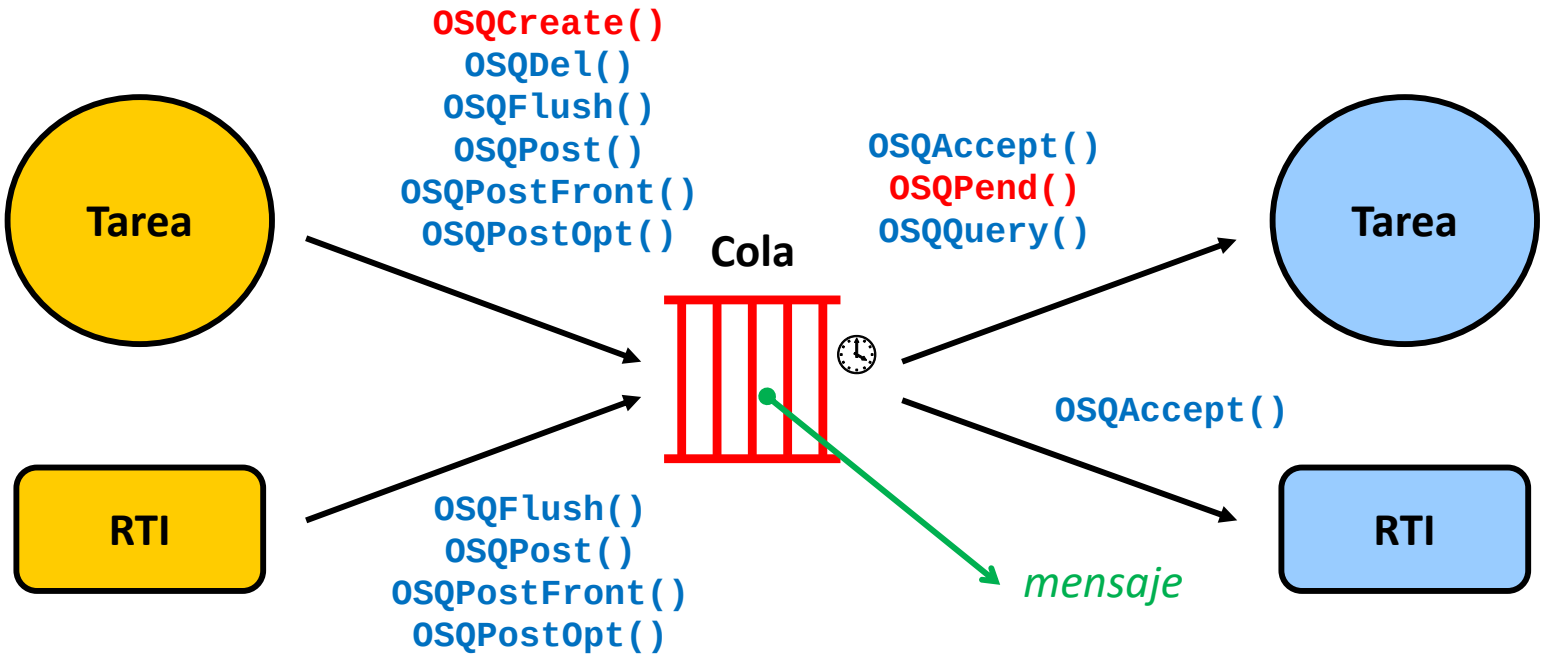
programación

- µC/OS-II solo gestiona el acceso a un puntero al mensaje
  - La aplicación debe decidir el tipo del mensaje y reservar espacio para él.

```
OS_EVENT *mailbox; ..... Declara mailbox (puntero a la ECB del buzón)
...
void main( void )
{
  OSInit();
  mailbox = OSMboxCreate( (void *) 0 ); ..... Crea mailbox
  ...
  OSStart();
}
void task1( void )
{
  char msg; ..... Declara el mensaje
  ...
  OSMboxPost( mailbox, (void *) &msg ); ..... Envía el puntero al mensaje
  ...
}
void task2( void )
{
  char *msg; ..... Declara un puntero al mensaje
  ...
  msg = (char *) OSMboxPend( mailbox, 0, &err ); ..... Recibe el puntero al mensaje
  ...
}
```

Gestión de colas de mensajes

- Una cola de mensajes en µC/OS-II es:
  - Un buffer circular de punteros a mensajes gestionados como FIFO/LIFO y una lista de tareas a la espera de recibir un mensaje.
  - Accesible a través de 9 servicios (7 de ellos opcionales) ofrecidos por el sistema.
- Se usa para:
  - Comunicar varias tareas/RTI entre sí



servicio siempre disponible / servicio disponible opcionalmente

Gestión de colas de mensajes

servicios

- Para gestionar colas, µC/OS-II ofrece los siguientes servicios:
  - `OSQCreate()`
    - Crea una cola de mensajes.
  - `OSQDel()`
    - Elimina una cola de mensajes. Típicamente, deberían eliminarse antes todas las tareas que la usan.
  - `OSQPend()`
    - (Receive) Si hay mensajes en la cola, lo toma; si la cola está vacía, suspende a la tarea invocante.
  - `OSQPost()/ OSQPostFront()`
    - (Post) Si hay tareas a la espera del mensaje, lo recibe la más prioritaria y pasa a preparada (pudiendo expropiar a la invocante); en caso contrario encola el mensaje en orden FIFO / LIFO.
  - `OSQPostOpt()`
    - (Broadcast post) Si hay tareas a la espera del mensaje, lo reciben todas y pasan a preparadas (pudiendo expropiar a la invocante); en caso contrario encola el mensaje en orden FIFO o LIFO.
  - `OSQAccept()`
    - (Receive no bloqueante) Si hay un mensaje en la cola, lo toma.
  - `OSQFlush()`
    - Elimina todos los mensajes de una cola.
  - `OSQQuery()`
    - Permite conocer información de una cola.
- El código fuente está disponible en el fichero os_q.c

J.M. Mendías 2016

Gestión de colas de mensajes

programación

- µC/OS-II solo gestiona el acceso ordenado a los n punteros a mensajes
  - La aplicación debe decidir el tipo del mensajes y reservar espacio para el array de punteros y para los propios mensajes.

```
OS_EVENT *msgQueue;..................... Declara cola (puntero a la ECB de la cola)
void     *msgQueueTable[20];............ Reserva espacio para 20 punteros a mensajes
...
void main( void )
{
  OSInit();
  msgQueue = OSQCreate( &msgQueueTable[0], 20 );..... Crea cola de 20 mensajes
  ...
  OSStart();
}
void task1( void )
{
  char msg[20] = "hola"; ..... Declara el mensaje
  ...
  OSQPost( msgQueue, (void * )&msg[0] );..... Envía el puntero al mensaje
  ...
}
void task2( void )
{
  char *msg; ..... Declara un puntero al mensaje
  ...
  msg = (char *) OSQPend( msgQueue, 0, &err );..... Recibe el puntero al mensaje
  ...
}
```

Gestión de memoria

- Una partición de memoria en μC/OS-II es una colección de bloques de memoria de tamaño fijo ocupando una región contigua de memoria.
  - Pueden crearse varias particiones con distinto número y tamaño de bloque cada una.
  - La asignación/liberación de bloques se hace en tiempo contante y conocido.
  - Esta asignación/liberación de bloques permite reutilizar porciones de memoria.

- Para gestionar memoria, μC/OS-II ofrece los siguientes servicios:
  - `OSMemCreate()`
    - Crea e inicializa una partición de memoria de un número dado de bloques del tamaño indicado.
  - `OSMemGet()`
    - Obtiene un bloque de memoria de una partición dada.
  - `OSMemPut()`
    - Devuelve un bloque de memoria de una partición dada.
  - `OSMemQuery()`
    - Permite conocer información de una partición de memoria.
- El código fuente está disponible en el fichero os_mem.c

Gestión de memoria

programación

- µC/OS-II solo gestiona el asignación/liberación de bloques
  - La aplicación debe reservar espacio para cada una de las particiones a gestionar

```
OS_MEM *memPartitionPtr; ..... Declara partición
INT8 memPartition[100][32]; ..... Reserva espacio para 100 bloques de 32 bytes cada uno
...
void main( void )
{
  OSInit();
  memPartitionPtr = OSMemCreate( &memPartition[0][0], 100, 32, &err );..... Crea partición
  ...
  OSStart();
}
...
void task( void )
{
  INT8 *pblock; ..... Declara puntero al bloque
  ...
  pblock = OSMemGet( PartitionPtr, &err );..... Obtiene bloque
  ...
  OsMemPut( PartitionPtr, pblock ); ..... Libera bloque
  ...
}
```

Flujo de desarrollo con µC/OS-II

- El desarrollo de una aplicación basada en µC/OS-II implica:
  - Portado de µC/OS-II al microprocesador/controlador usado:
    - Solo se realiza 1 vez por familia de procesador.
    - El portado puede reutilizarse para las distintas plataformas que usen dicho procesador.
  - Desarrollo/adaptación del firmware
    - Solo se realiza 1 vez por plataforma.
    - Puede reutilizarse para las distintas aplicaciones que puedan correr sobre ella.
    - Si se prevé que distintas tareas hagan uso compartido de los dispositivos disponibles deberán usarse los servicios de protección facilitados por el SO
    - Si el firmware atiende periféricos por interrupción, las ISR deberán tener completarse con un wrapper y usar los servicios de sincronización facilitados por el SO
  - Configuración de µC/OS-II
    - Se realiza 1 vez por aplicación.
    - No puede reutilizarse.
  - Desarrollo de la aplicación empotrada
    - Programación de las tareas e ISR que la forman

Portado de µC/OS-II

- Nativamente µC/OS-II es independiente de plataforma, pero:
  - Para que corra sobre un procesador concreto debe adaptarse (portar) la pequeña porción del kernel que es dependiente del procesador.
  - Esta porción se localiza en 3 ficheros: **`os_cpu.h, os_cpu.c, os_asm.h`**
- Portar µC/OS-II a un procesador dado supone:
  - Definir los tipos usados por el kernel
  - Indicar al kernel el sentido de crecimiento de la pila.
  - Implementar 2 funciones (en ensamblador) para deshabilitar/habilitar interrupciones
    - De manera que usándolas el kernel pueda proteger sus secciones críticas.
  - Indicar al kernel el nombre de dichas funciones y el tipo de método implementado.
  - Implementar 5 funciones (1 en C y 4 en ensamblador) que el kernel llamará para:
    - Crear en la pila de una tarea su contexto inicial
    - Conmutar a una tarea desde otra tarea que ha sido expropiada por el kernel
    - Conmutar a una tarea desde una RTI
    - Despachar inicialmente la tarea de más alta prioridad
    - Hacer el tratamiento de las interrupciones de un temporizador
  - Indicar al kernel el nombre de la función que debe usar para conmutar entre tareas.
  - Opcionalmente, implementar 9 funciones en C (hooks) para ampliar la funcionalidad del sistema incluyendo código en ciertos lugares estratégicos predefinidos del kernel.

J.M. Mendías
2016

Portado de µC/OS-II

tipos y macros

- En el fichero os_cpu.h los tipos a definir son:
  - **`BOOLEAN, INT8U, INT8S, INT16U, INT16S, INT32U, INT32S, FP32, FP64`**
    - Todos ellos tipos numéricos
  - **`OS_STK`**
    - Tipo de una entrada de la pila.
  - **`OS_CPU_SR`**
    - Tipo del registro de estado de la CPU (CPSR).
- Las macros a definir son:
  - **`OS_CRITICAL_METHOD`**
    - Indica al kernel el tipo de método implementado para des/habilitar las interrupciones:
      (1) Sin salvado del registro de estado
      (2) Con salvado/restauración del registro de estado en la pila
      (3) Con salvado/restauración del registro de estado en una variable local
  - **`OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL()`**
    - Indica al kernel el nombre de las funciones para des/habilitar las interrupciones.
  - **`OS_STK_GROWTH`**
    - Indica al kernel el sentido de crecimiento de la pila (1 de direcciones altas a bajas).
  - **`OS_TASK_SW()`**
    - Indica al kernel el nombre de la función para realizar cambios de contexto entre tareas.

Portado de µC/OS-II

funciones básicas

- Las 7 funciones a implementar se reparten en 2 ficheros, en el fichero os_cpu.c la función a implementar en C:
  - **`OSTaskStkInit()`**
    - Debe crear en la pila de una tarea su contexto inicial. Se llama cada vez que se crea una tarea.
- En el fichero os_cpu.asm las funciones a implementar en ensamblador:
  - **`OSCtxSw()`**
    - Conmuta (tras una expropiación) de la tarea actual a la tarea preparada con mayor prioridad.
  - **`OSIntCtxSw()`**
    - Conmuta (tras una interrupción) de la tarea actual a la tarea preparada con mayor prioridad. Su funcionalidad es equivalente a **`OSCtxSw()`** pero sin guardar el contexto de la tarea actual (lo hizo la correspondiente RTI).
  - **`OSStartHighRdy()`**
    - Comienza la multitarea arrancando la tarea preparada con mayor prioridad. Esta función es llamada por **`OSStart()`**.
  - **`OSTickISR()`**
    - Rutina de tratamiento de la interrupción generada por el temporizador usado por el sistema para hacer el seguimiento de retardos y timeouts.
  - Las funciones para des/habilitar interrupciones pueden tener el nombre que se desee
    - Pueden implementarse en C como ensamblador en línea
    - Su nombre se indica al definir las macros **`OS_ENTER_CRITICAL() / OS_EXIT_CRITICAL()`**

Portado de µC/OS-II

hooks

- Los 9 hooks se implementan también en el fichero os_cpu.c
  - **`OSInitHookBegin()`**
    - Es llamada por OSInit() a su comienzo.
  - **`OSInitHookEnd()`**
    - Es llamada por OSInit() a su final.
  - **`OSTaskCreateHook()`**
    - Es llamada cada vez que se crea una tarea.
  - **`OSTaskDelHook()`**
    - Es llamada cada vez que se elimina una tarea.
  - **`OSTaskIdleHook()`**
    - Es llamada por la tarea idle (i.e. para dormir al procesador).
  - **`OSTaskStatHook()`**
    - Es llamada cada segundo por la tarea de recopilación de estadísticas.
  - **`OSTaskSwHook()`**
    - Es llamada cada vez que se conmuta una tarea.
  - **`OSTCBInitHook()`**
    - Es llamada cada vez que se crea una tarea tras la creación e inicialización de su TCB
  - **`OSTimeTickHook()`**
    - Es llamada cada vez que hay un tick

Portado de µC/OS-II

sobre programación de RTI

- En µC/OS-II tras servir a una interrupción no tiene por qué volverse a la misma tarea que fue interrumpida.
  - En general, la tarea interrumpida se suspende y debe pasar a ejecución la tarea preparada más prioritaria.
  - El contexto de la tarea expropiada/despachada se debe guardar/restaurar de su propia pila y no de la pila IRQ/IFQ
- Por esa razón, en µC/OS-II:
  - El mecanismo ofrecido por el compilador (atributo **`interrupt`**) no se usa.
  - Todas las RTI deben programarse en ensamblador.
- No obstante, en el portado puede incluirse una macro que permita poner un wrapper a una RTI :
  - Realice las conmutaciones de contexto requerida por µC/OS-II
  - Informe al sistema de cuando comienza y finaliza la RTI
  - Llame a la RTI desarrollada en C por el programador, que será la encargada de:
    - Borrar el flag de interrupción pendiente que corresponda.
    - Opcionalmente habilitar interrupciones para permitir su anidamiento.
    - Atender al dispositivo.

Portado de µC/OS-II

portado al ARM7TDMI

- Los portados a los procesadores más comunes los facilita el proveedor
  - Si no está disponible o se desea particularizarlo puede programarse siguiendo las indicaciones del desarrollador del SO

```
#include <common_types.h>
#include <system.h>

typedef boolean BOOLEAN;
typedef uint8   INT8U;
typedef int8    INT8S;
typedef uint16  INT16U;
typedef int16   INT16S;
typedef uint32  INT32U;
typedef int32   INT32S;
typedef float   FP32;
typedef double  FP64;

typedef uint32  OS_STK;
typedef uint32  OS_CPU_SR;

#define OS_CRITICAL_METHOD  2

#define OS_ENTER_CRITICAL() INT_DISABLE
#define OS_EXIT_CRITICAL()  INT_ENABLE

#define OS_STK_GROWTH       1
#define OS_TASK_SW()        OSCtxSw()
```

Tipos definidos en common_types.h

Macros definidas en system.h

os_cpu.h

Portado de µC/OS-II

portado al ARM7TDMI

```
OS_STK *OSTaskStkInit( void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt )
{
  *(ptos)   = (INT32U) task;      ····· Apila el punto de entrada a la tarea
  *(--ptos) = (INT32U) 0;         ····· Apila el valor inicial de LR
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;
  *(--ptos) = (INT32U) 0;         ····· Apila el valor inicial de R12-R1
  *(--ptos) = (INT32U) pdata;     ····· Apila el valor inicial R0 : argumento de la tarea
  *(--ptos) = (INT32U) (0x13|0x0);····· Apila el valor inicial CPSR : modo SVS, IRQ/FIQ habilitadas

  return (ptos);                  ····· Devuelve la cima de la pila actualizada
}

...
void OSTaskIdleHook( void )
{
    sleep();                      ····· Pone a la CPU en estado IDLE, sale por interrupción
}
```

os_cpu.c

J.M. Mendías
2016

Portado de µC/OS-II

portado al ARM7TDMI

os_cpu.asm

```
OSStartHighRdy:

  bl    OSTaskSwHook ·····  (1) Llama a la función OSTaskSwHook()

  ldr   r4, =OSRunning
  mov   r5, #1
  strb  r5, [r4]            (2) OSRunning = TRUE

  ldr   r4, =OSTCBHighRdy
  ldr   r4, [r4]
  ldr   sp, [r4]            (3) SP = OSTCBHighRdy->OSTCBStkPtr (recupera el SP del TCB de la tarea
                                preparada con mayor prioridad)

  ldmfd sp!,  {r4}
  msr   spsr, r4            (4) Restaura de la pila el contexto de la tarea preparada con mayor prioridad y ...
  ldmfd sp!,  {r0-r12, lr, pc}^   (5) ... retorna a dicha tarea
```

Portado de µC/OS-II

portado al ARM7TDMI

os_cpu.asm

```
OSCtxSw:

  stmfd sp!, {lr}
  stmfd sp!, {r0-r12, lr}
  mrs   r4,  cpsr
  stmfd sp!, {r4}
```

(1) Guarda en la pila el contexto de la tarea actual

```
  ldr r4, =OSTCBCur
  ldr r5, [r4]
  str sp, [r5]
```

(2) OSTCBCur->OSTCBStkPtr = SP (guarda SP en el TCB de la tarea actual)

```
OSIntCtxSw:

  bl OSTaskSwHook
```

(3) Llama a la función OSTaskSwHook()

```
  ldr r4, =OSTCBCur
  ldr r5, =OSTCBHighRdy
  ldr r5, [r5]
  str r5, [r4]
```

(4) OSTCBCur = OSTCBHighRdy

```
  ldr  r6, =OSPrioHighRdy
  ldr  r7, =OSPrioCur
  ldrb r6, [r6]
  strb r6, [r7]
```

(5) OSPrioCur = OSPrioHighRdy

```
  ldr sp, [r5]
```

(6) SP = OSTCBHighRdy->OSTCBStkPtr

```
  ldmfd sp!,  {r4}
  msr   spsr, r4
  ldmfd sp!,  {r0-r12, lr, pc}^
```

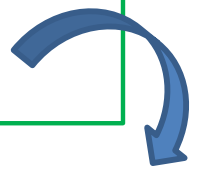
(7) Restaura de la pila el contexto de la tarea preparada con mayor prioridad y ...
(8) ... retorna a dicha tarea

Portado de µC/OS-II

portado al ARM7TDMI

os_cpu.asm

```
OSTickISR:

  stmfd sp!, {r1-r3}        ..... Apila los registros de trabajo en la pila IRQ

  mov r1, sp
  add sp, sp, #12
  sub r2, lr, #4
  mrs r3, spsr              ] Copia en R1, R2 y R3 respectivamente:
                            ] el puntero de pila, la dirección de retorno y el CPSR de la tarea interrumpida

  msr cpsr_c, #(NOINT|SVCMODE)  ..... Cambia a modo SVC

  stmfd sp!, {r2}
  stmfd sp!, {lr}
  stmfd sp!, {r4-r12}
  ldmfd r1!, {r4-r6}
  stmfd sp!, {r4-r6}
  stmfd sp!, {r0}
  stmfd sp!, {r3}           ] (1) Guarda en la pila SVC (la pila que estaba siendo usada por la tarea
                            ] interrumpida) el contexto de la tarea actual

  bl  OSIntEnter            ..... (2) Llama a la función OSIntEnter()

  ldr  r0, =OSIntNesting
  ldrb r0, [r0]
  cmp  r0, #1
  bne  OSTickISRcont
  ldr  r0, =OSTCBCur
  ldr  r1, [r0]
  str  sp, [r1]             ] (3) Si OSIntNesting == 1 entonces OSTCBCur->OSTCBStkPtr = SP
```



Portado de µC/OS-II

portado al ARM7TDMI

```
OSTickISRcont:
  msr cpsr_c, #(NOINT|IRQMODE)      ····· Vuelve a modo IRQ

  ldr r1, =I_ISPC                   }
  mov r2, #BIT_TIMER0               }  (4) Borra interrupción pendiente por timer0
  str r2, [r1]                      }

  bl  OSTimeTick ····················  (6) Llama a la función OSTimeTick ()

  msr cpsr_c, #(NOINT|SVCMODE)      ····· Cambia a modo SVC

  bl  OSIntExit ····················  (7) Llama a la función OSIntExit() que si no hay más ISR anidadas despachará
                                        la tarea preparada (si la hay) más prioritaria

  ldmfd sp!,  {r4}                  }  Si vuelve (porque no hay una tarea preparada más prioritaria que la actual):
  msr   spsr, r4                    }  (8) Restaura de la pila el contexto de la tarea actual ...
  ldmfd sp!,  {r0-r12, lr, pc}^     }  (9) ... retorna a dicha tarea
```

Este es el procesamiento asociado a una interrupción de timer0, el resto es requerido por el kernel para poder expropiar a la tarea interrumpida.

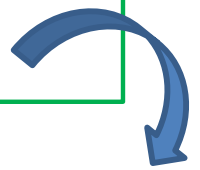
(5) *Aquí opcionalmente podrían habilitarse (y posteriormente deshabilitarse) la anidación de interrupciones*

Portado de µC/OS-II

portado al ARM7TDMI

os_cpu_isr_wrapper.asm

```
.macro OS_CPU_ISR_WRAPPER isr

  stmfd sp!, {r1-r3}          ····· Apila los registros de trabajo en la pila IRQ

  mov r1, sp
  add sp, sp, #12
  sub r2, lr, #4              ····· Copia en R1, R2 y R3 respectivamente:
  mrs r3, spsr                      el puntero de pila, la dirección de retorno y el CPSR de la tarea interrumpida

  msr cpsr_c, #(NOINT|SVCMODE) ····· Cambia a modo SVC

  stmfd   sp!, {r2}
  stmfd   sp!, {lr}
  stmfd   sp!, {r4-r12}
  ldmfd   r1!, {r4-r6}        ····· (1) Guarda en la pila SVC el contexto de la tarea actual
  stmfd   sp!, {r4-r6}
  stmfd   sp!, {r0}
  stmfd   sp!, {r3}

  bl  OSIntEnter              ····· (2) Llama a la función OSIntEnter()

  ldr  r0, =OSIntNesting
  ldrb r0, [r0]
  cmp  r0, #1
  bne  label\isr              ····· (3) Si OSIntNesting == 1 entonces OSTCBCur->OSTCBStkPtr = SP
  ldr  r0, =OSTCBCur
  ldr  r1, [r0]
  str  sp, [r1]
```



Portado de µC/OS-II

portado al ARM7TDMI

```
label\isr:
  msr cpsr_c, #(NOINT|IRQMODE) ····· Vuelve a modo IRQ
  bl  \isr ····· (6) Llama a la función pasada como argumento a la macro
  msr cpsr_c, #(NOINT|SVCMODE) ····· Cambia a modo SVC
  bl  OSIntExit ····· (7) Llama a la función OSIntExit() que si no hay más ISR anidadas despachará
                          la tarea preparada (si la hay) más prioritaria
  ldmfd sp!,  {r4}
  msr   spsr, r4
  ldmfd sp!,  {r0-r12, lr, pc}^
                      Si vuelve (porque no hay una tarea preparada más prioritaria que la actual):
                      (8) Restaura de la pila el contexto de la tarea actual ...
                      (9) ... retorna a dicha tarea
.endm
```

- La aplicación únicamente deberá tener un fichero en ensamblador que ponga un wrapper a cada una de las RTI

app.asm

```
...

OS_CPU_isr_pb:
  OS_CPU_ISR_WRAPPER isr_pb

OS_CPU_isr_keypad:
  OS_CPU_ISR_WRAPPER isr_keypad
...
```

Portado de µC/OS-II

RTI

- Una RTI es una función void sin argumentos que puede programarse en C si el portado facilita el wrapper
  - No debe atribuirse como **`interrupt`**
  - Únicamente debe incluir el código propio de atención al dispositivo
  - La versión "wrappeada" debe instalarse en la tabla de vectores de interrupción

```
..
void isr_keypad( void );
extern void OS_CPU_isr_keypad( void );

...
void main( void )
{
  ...
  pbs_open( OS_CPU_isr_keypad );
  ...
}

...
void isr_keypad( void )
{
  ... codigo ...
  I_ISPC = BIT_KEYPAD;
}
```

app.c

```
.include ...

  .extern isr_pb
  .global OS_CPU_isr_keypad

.section .text

OS_CPU_isr_keypad:
  OS_CPU_ISR_WRAPPER isr_keypad

.end
```

app.asm

Configuración de µC/OS-II

- µC/OS-II debe configurarse para que ofrezca únicamente los servicios que la aplicación vaya a usar.
  - Con esto se puede reducir el footprint del kernel al mínimo necesario.
  - Es posible porque el código del kernel integra bloques compilables condicionalmente

```
#if OS_ARG_CHK_EN > 0
    if (pgrp == (OS_FLAG_GRP *)0) {
        *err = OS_FLAG_INVALID_PGRP;
        return ((OS_FLAGS)0);
    }
#endif
```

  - El programador mediante la asignación de valores a macros puede activar/desactivar la compilación de dichos bloques.
  - Todas las macros se encuentran en el fichero: **os_cfg.h**

- Configurar µC/OS-II para una aplicación dada supone:
  - Definir el valor de 51 macros
  - Definir 1 tipo de datos

Configuración de µC/OS-II

- El kernel se configura definiendo:
  - **OS_MAX_EVENTS**
    - Núm. máximo conjutno de semáforos+ buzones +colas (mayor que 0).
  - **OS_MAX_FLAGS / OS_MAX_MEM_PART / OS_MAX_QS**
    - Núm. máximo de grupos de flags de eventos / particiones de memoria /colas (mayor que 0).
  - **OS_MAX_TASKS**
    - Núm. máximo de tareas (mayor o igual a 2 y menor o igual a 62).
  - **OS_LOWEST_PRIO**
    - Mínima prioridad asignable (menor o igual a 63)
  - **S_TASK_IDLE_STK_SIZE**
    - Número de entradas de la pila de la tarea idle.
  - **OS_TASK_STAT_EN**
    - Habilita/desahibilita la tarea de recopilación de estadísicas.
  - **OS_TASK_STAT_STK_SIZE**
    - Número de entradas de la pila de la tarea stat.
  - **OS_ARG_CHK_EN**
    - Des/habilita que las funciones de µC/OS-II chequen los argumentos.
  - **OS_CPU_HOOKS_EN**
    - Des/habilita la inclusión de las funciones hooks establecidas en el portado.
  - **OS_SCHED_LOCK_EN**
    - Des/habilita la inclusión de las funciones **`OSSchedLock()`** y **`OSSchedUnlock()`**
  - **OS_TICKS_PER_SEC**
    - Indica el número de ticks que se generarán por segundo
  - **`typedef ... OS_FLAGS`**
    - Define el tamaño de los grupos de flags de eventos (8, 16 o 32 bits)

J.M. Mendías
2016

Configuración de µC/OS-II

- La gestión de tareas se configura definiendo:
  - **OS_TASK_CHANGE_PRIO_EN**
    - Des/habilita la inclusión de la función **OSTaskChangePrio()**
  - **OS_TASK_CREATE_EN / OS_TASK_CREATE_EXT_EN**
    - Des/habilita la inclusión de la función **OSTaskCreate() /OSTaskCreateExt()**
  - **OS_TASK_DEL_EN**
    - Des/habilita la inclusión de la función **OSTaskDel()**
  - **OS_TASK_SUSPEND_EN**
    - Des/habilita la inclusión de la función **OSTaskSuspend()** y **OSTaskResume()**
  - **OS_TASK_QUERY_EN**
    - Des/habilita la inclusión de la función **OSTaskQuery()**
- La gestión del tiempo se configura definiendo:
  - **OS_TIME_DLY_HMSM_EN**
    - Des/habilita la inclusión de la función **OSTimeDlyHMSM()**
  - **OS_TIME_DLY_RESUME_EN**
    - Des/habilita la inclusión de la función **OSTimeDlyResume()**
  - **OS_TIME_GET_SET_EN**
    - Des/habilita la inclusión de las funciones **OSTimeGet()** y **OSTimeSet()**

Configuración de µC/OS-II

- La gestión de semáforos se configura definiendo:
  - **OS_SEM_EN**
    - Des/habilita la inclusión de código para la gestión de semáforos
  - **OS_SEM_ACCEPT_EN**
    - Des/habilita la inclusión de la función **OSSemAccept()**
  - **OS_SEM_DEL_EN**
    - Des/habilita la inclusión de la función **OSSemDel()**
  - **OS_SEM_QUERY_EN**
    - Des/habilita la inclusión de la función **OSSemQuery()**
- La gestión de grupos de flags de eventos se configura definiendo:
  - **OS_FLAG_EN**
    - Des/habilita la inclusión de código para la gestión de flags de eventos
  - **OS_FLAG_WAIT_CLR_EN**
    - Des/habilita la inclusión de código para la espera por la desactivación de flags
  - **OS_FLAG_ACCEPT_EN**
    - Des/habilita la inclusión de la función **OSFlagAccept()**
  - **OS_FLAG_DEL_EN**
    - Des/habilita la inclusión de la función **OSFlagAccept()**
  - **OS_FLAG_QUERY_EN**
    - Des/habilita la inclusión de la función **OSFlagQuery()**

Configuración de µC/OS-II

- La gestión de semáforos mutex se configura definiendo:
  - **OS_MUTEX_EN**
    - Des/habilita la inclusión de código para la gestión de semáforos mutex
  - **OS_MUTEX_ACCEPT_EN**
    - Des/habilita la inclusión de la función **OSMutexAccept()**
  - **OS_MUTEX_DEL_EN**
    - Des/habilita la inclusión de la función **OSMutexDel()**
  - **OS_MUTEX_QUERY_EN**
    - Des/habilita la inclusión de la función **OSMutexQuery()**
- La gestión de buzones se configura definiendo:
  - **OS_MBOX_EN**
    - Des/habilita la inclusión de código para la gestión de buzones
  - **OS_MBOX_ACCEPT_EN**
    - Des/habilita la inclusión de la función **OSMboxAccept()**
  - **OS_MBOX_DEL_EN**
    - Des/habilita la inclusión de la función **OSMboxDel()**
  - **OS_MBOX_POST_EN / OS_MBOX_POST_OPT_EN**
    - Des/habilita la inclusión de la función **OSMboxPost() / OSMboxPostOpt()**
  - **OS_MBOX_QUERY_EN**
    - Des/habilita la inclusión de la función **OSMboxQuery()**

Configuración de µC/OS-II

- La gestión de colas de mensajes se configura definiendo:
  - **OS_Q_EN**
    - Des/habilita la inclusión de código para la gestión de colas de mensajes
  - **OS_Q_ACCEPT_EN**
    - Des/habilita la inclusión de la función **OSQAccept()**
  - **OS_Q_DEL_EN**
    - Des/habilita la inclusión de la función **OSQDel()**
  - **OS_Q_FLUSH_EN**
    - Des/habilita la inclusión de la función **OSQFlush()**
  - **OS_Q_POST_EN / OS_Q_POST_FRONT_EN / OS_Q_POST_OPT_EN**
    - Des/habilita la inclusión de la función **OSQPost() / OSQPostFront() / OSQPostOpt()**
  - **OS_Q_QUERY_EN**
    - Des/habilita la inclusión de la función **OSQQuery()**
- La gestión de memoria se configura definiendo:
  - **OS_MEM_EN**
    - Des/habilita la inclusión de código para la gestión de memoria
  - **OS_MEM_QUERY_EN**
    - Des/habilita la inclusión de la función **OSMemQuery()**

Acerca de *Creative Commons*

- Licencia CC (Creative Commons)

  - Ofrece algunos derechos a terceras personas bajo ciertas condiciones. Este documento tiene establecidas las siguientes:

Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Más información: https://creativecommons.org/licenses/by-nc-sa/4.0/