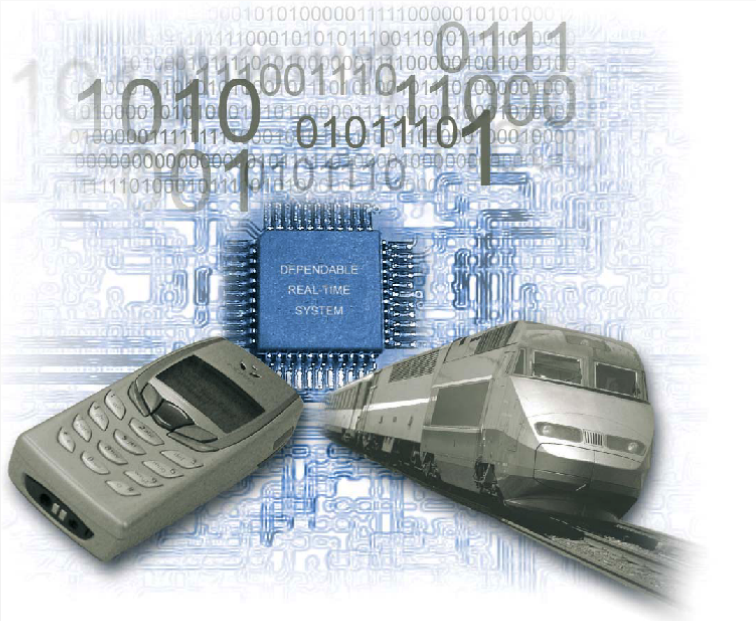
Sistemas de Tiempo Real

(*Real-Time Systems*)



Departamento de Electrónica

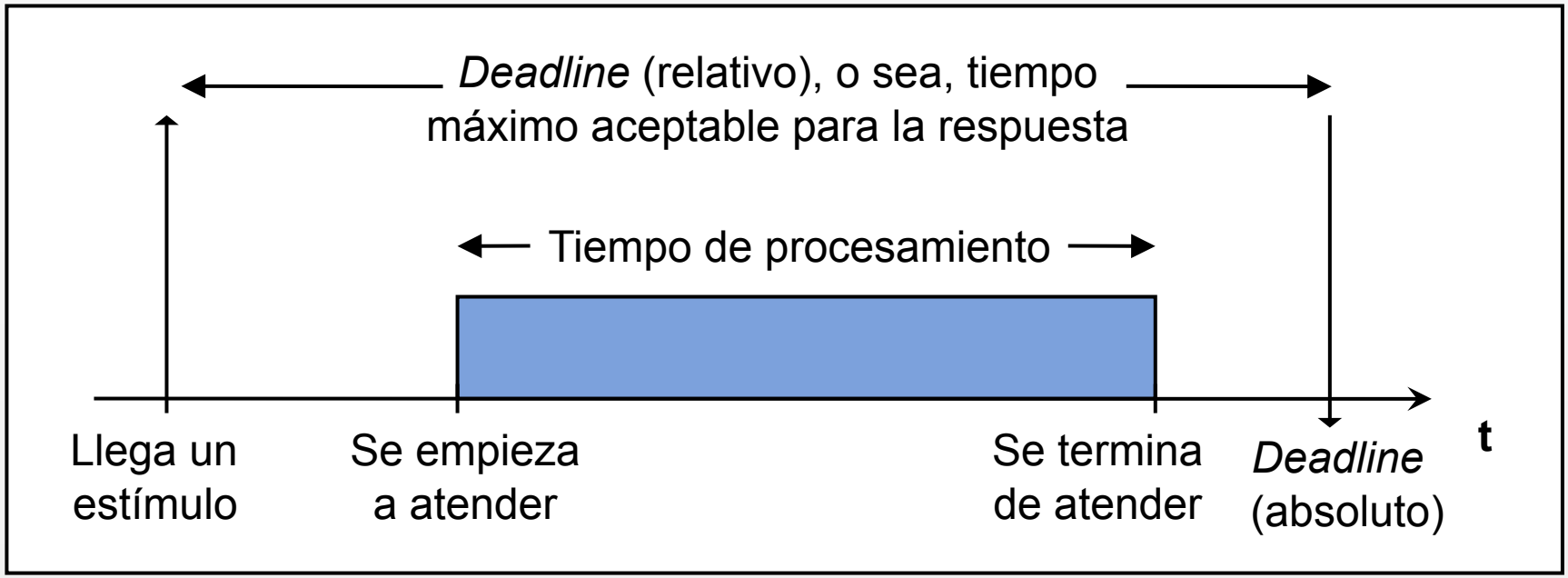
Universidad de Alcalá

Índice

- Introducción
- Requerimientos
- Diseño de un STR
- RTOS
- RTX de Keil
- Resumen
- Referencias

Introducción

- **Un sistema de tiempo real (STR o *real-time system* o RTS) es aquel que responde en un tiempo acotado.**
  - O sea que debe estar **acotado** el tiempo entre cada evento y la respuesta que provoca
    - No necesariamente tiene que ser breve, pero sí acotado



Introducción

- **STR *estricto* (*hard RTS*)**
  - Es *estricto* cuando el incumplimiento de un deadline implica un **funcionamiento incorrecto**
    - Ejemplos:
      - El sistema ABS (*anti-lock breaking system*) de un automóvil
      - Un marcapasos
- **STR *suave* (*soft RTS*)**
  - Es *suave* cuando el incumplimiento de un deadline no implica funcionamiento incorrecto pero sí una **degradación en la calidad de servicio**
    - Ejemplos:
      - Procesamiento de video
        - » Porque es aceptable que se pierda algún que otro cuadro
      - Un reproductor de DVD
      - Interfaces al usuario en general

Requerimientos

- **Un STR está definido por una lista de:**
  - Los **eventos externos** que puede atender
  - La respuesta lógica (o sea, **salida**) que debe producir cada evento
  - Los **requerimientos de temporización** de esas respuestas
    - O sea, sus **deadlines relativos**
- **Para simplificar, los STR suaves frecuentemente se diseñan como si fueran STR estrictos**
  - O sea, como si sus deadlines fueran estrictos
- **Como siempre, para especificar y refinar los requerimientos, podemos recurrir a:**
  - Diagramas de flujo
  - Statecharts
  - Otros modelos de computación y lenguajes de modelado

Requerimientos

- **A veces se recurre a métodos formales para verificar el cumplimiento de los requerimientos de temporización, pero es más frecuente testearlos mediante simulaciones y pruebas en prototipos**
  - Problemas:
    - Ante cambios menores, hay que volver a testear todo
      - Se aliviana automatizando esas pruebas, si se puede
    - El testeo nunca da garantías al 100%
  - Es muy valioso usar técnicas de programación que nos den cierta seguridad sobre el cumplimiento que los requisitos de temporización, para no depender mucho de la verificación.
- **Para cumplir esos requerimientos, a veces hay que evitar usar técnicas que implican tiempos largos y/o poco predecibles**
  - Ejemplos:
    - Programación orientada a objetos
    - *Garbage collecting* (como el de Java)
    - `malloc()` y `free()` comunes de C

Diseño de un STR

- **Un STR es un sistema reactivo con requisitos (estrictos o no) en cuanto a sus tiempos de respuesta**
- **Esos requisitos se consideran desde la etapa de elaboración de requerimientos y durante todo el proceso de diseño**
  - …a diferencia del diseño de un software transaccional común, en los cuales lo típico, como mucho, es chequear la velocidad una vez programadas sus unidades, para decidir si optimizarlas o no
- **Recordar que *reactivo* significa que responde a eventos externos, que no necesariamente tienen orden o periodicidad**

Diseño de un STR

- **Un STR puede diseñarse directamente en Assembly o lenguaje C.**
  - Ejemplo:
    - Un ciclo infinito donde se encuestan, una tras otra, las entradas correspondientes a los eventos externos, y se las atiende rápidamente
    - El evento externo que no pueda esperar, debe ir asociado a una interrupción, etc.
- **Sin embargo, en sistemas medianamente complejos, suele ser difícil asegurar los requisitos de temporización si se emplea ese enfoque**
  - Recordar que valoramos las técnicas de programación que dan cierta seguridad sobre el cumplimiento que esos requisitos
- **Por eso, a veces es conveniente utilizar un sistema operativo de tiempo real (*real-time operating system* o RTOS)**

RTOS

- **Un sistema operativo de tiempo real (RTOS) es un software de base que simplifica el diseño de software con requerimientos de tiempo real**
- **Permite que el programador estructure la aplicación como un conjunto de tareas concurrentes**
  - *concurrentes* = que se ejecutan al mismo tiempo
  - Así, el procesamiento de cada evento se asigna a una tarea, pudiéndose obtener una demora razonablemente corta entre el evento y la ejecución de la tarea, sin que haya que escribir un código muy intrincado
  - Normalmente, las tareas no son realmente concurrentes, sino que el procesador se reparte entre ellas, creando esa ilusión
- **El RTOS gestiona la ejecución de esas tareas y provee servicios que la aplicación utiliza para acceder, con tiempos razonables y predecibles, al hardware**
  - Ej., para manejar memoria y entrada/salida

¿Por qué usar un RTOS?

- **Para cumplir con compromisos temporales estrictos**
  - El RTOS ofrece funcionalidad para asegurar que una vez ocurrido un evento, la respuesta ocurra dentro de un tiempo acotado. Es importante aclarar que esto no lo hace por sí solo sino que brinda al programador herramientas para hacerlo de manera más sencilla que si no hubiera un RTOS.
    - Esto implica que una aplicación mal diseñada puede fallar en la atención de eventos aún cuando se use un RTOS.
- **Para no tener que manejar el tiempo "a mano"**
  - El RTOS absorbe el manejo de temporizadores y esperas, de modo que hace más fácil al programador el manejo del tiempo.
- **Tarea Idle**
  - Cuando ninguna de las tareas requiere del procesador, el sistema ejecuta una tarea llamada idle u ociosa. Esto Permite fácilmente contabilizar el nivel de ocupación del CPU, poner al mismo en modo de bajo consumo o correr cualquier tarea que pudiera ser de utilidad para el sistema cuando no debe atender ninguno de sus eventos.

¿Por que usar un RTOS?

- **Multitarea**
  - Simplifica sobremanera la programación de sistemas con varias tareas.
- **Escalabalidad**
  - Al tener ejecución concurrente de tareas se pueden agregar las que hagan falta, teniendo el único cuidado de insertarlas correctamente en el esquema de ejecución del sistema.
- **Mayor reutilizabilidad del código**
  - Si las tareas se diseñan bien (con pocas o ninguna dependencia) es más fácil incorporarlas a otras aplicaciones.

Inconvenientes

- **Se gasta tiempo de la CPU en determinar en todo momento qué tarea debe ejecutarse. Si el sistema debe manejar eventos que ocurren demasiado rápido tal vez no puedan atenderse.**
- **Se gasta tiempo de la CPU cada vez que debe cambiarse la tarea en ejecución.**
- **Se gasta memoria de código para implementar la funcionalidad del RTOS.**
- **Se gasta memoria de datos en mantener una pila y un TCB (bloque de control de tarea) por cada tarea**
  - El tamaño de estas pilas suele ser configurable POR TAREA, lo cual mitiga este impacto.

Inconvenientes

- Finalmente, debe hacerse un análisis de tiempos, eventos y respuestas más cuidadoso. Al usar un RTOS ya no es el programador quién decide cuándo ejecutar cada tarea, sino el scheduler. Cometer un error en el uso de las reglas de ejecución de tareas puede llevar a que los eventos se procesen fuera del tiempo especificado o que no se procesen.
- A la luz de los beneficios mencionados, en el caso de que la plataforma de hardware lo permita y el programador esté capacitado no hay razones para no usar un RTOS.

Comparación con OS comunes

- **Similitudes usuales con sistemas operativos de propósitos generales:**
  - **Multitasking**. O sea, ejecución de tareas (*tasks*) concurrentes
  - Provisión de **servicios** para las aplicaciones
  - (Algo de) **abstracción** del hardware
  - Un ***kernel***
    - O sea, un **núcleo** del sistema operativo, que está siempre en memoria gestionando la ejecución de las tareas y sirviendo de puente con el hardware
    - Sin embargo, algunos RTOS son solo librerías, o sea, *kernel-less*
- **Diferencias:**
  - En los RTOS, la gestión del multitasking está especialmente diseñada para atender requisitos de **tiempo real**
  - No suelen incluir interfaces de usuario gráficas, solo a veces incluyen gestión de archivos en disco, etc.
  - Son programas **mucho más livianos**
  - Los hay especiales para sistemas que requieren una confiabilidad excepcional
    - O sea, sistemas de *misión crítica* o *seguridad crítica*

Tipo de tareas en una aplicación de tiempo real

- **Tareas periódicas**
  - Atienden eventos que ocurren constantemente y a una frecuencia determinada. P. ej, destellar un led.
- **Tareas aperiódicas**
  - Atienden eventos que no se sabe cuándo van a darse. Estas tareas están inactivas (bloqueadas) hasta que no ocurre el evento de interés. P. ej, una parada de emergencia.
- **Tareas de procesamiento continuo**
  - Son tareas que trabajan en régimen permanente. P. ej, muestrear un buffer de recepción en espera de datos para procesar.
  - Estas tareas deben tener prioridad menor que las otras, ya que en caso contrario podrían impedir su ejecución.

Estándares

- POSIX 2008 o IEEE STD 1003.1-2008 incluye funciones para:
  - E/S síncronas y asíncronas
  - Primitivas IPC
  - Habilidad para bloquear y mantener memoria
  - Planificación por prioridades
  - Archivo de tiempo real, y
  - Cronómetro.
- TRON (The Real-time Operating-system Nucleus).
  - Consta de varios subproyectos entre ellos:
    - ITROM – especificación del RTOS para sistemas empotrados e industriales,
    - µITROM - microprocesadores y microcontroladores de bajo coste.

Algunos RTOS populares

- **VxWorks**
  - De Wind River, que es subsidiaria de Intel desde julio de 2009
  - Con soporte para multiprocesadores, IPv6 y un sistema de archivos
  - Con protección de memoria
    - Las tareas no pueden alterar la memoria de trabajo de otras tareas
  - Funciona en las plataformas de embebidos más populares
  - Se usa con un IDE para Windows/Linux, que normalmente incluye depurador, simulador y herramientas de análisis
- **QNX**
  - De QNX, subsidiaria de Research in Motion (los del Blackberry) desde mayo de 2010
  - Símil Unix, ofrece funcionalidad parecida al VxWorks
  - En 2007 fue abierto el código de su núcleo
- **RTLinux**
  - Basado en Linux
- **FreeRTOS**
  - Es gratuito

Kernel monolítico vs. µkernel

- **Existen básicamente dos maneras de organizar un OS:**
  - De **núcleo (*kernel*) monolítico**: Todas las funciones "residentes" del OS están en su núcleo
  - Con ***microkernel***: Algunas funciones del OS se implementan como **tareas** similares a las de la aplicación
- **VxWorks tiene kernel monolítico; QNX, RTLinux y FreeRTOS usan microkernel**

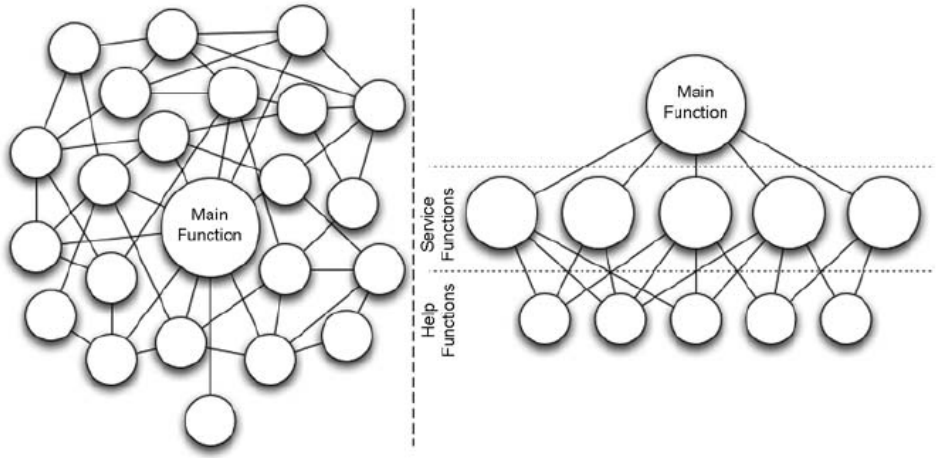


Figure 2.6. Unstructured vs. structured monolithic kernel.

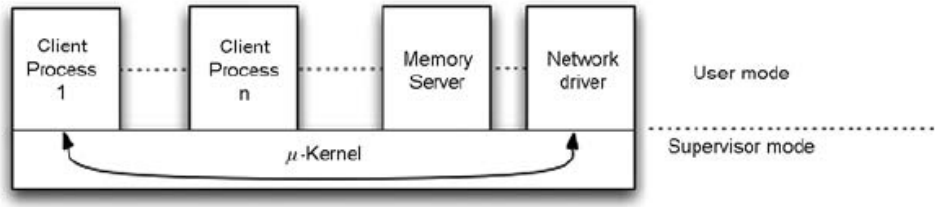
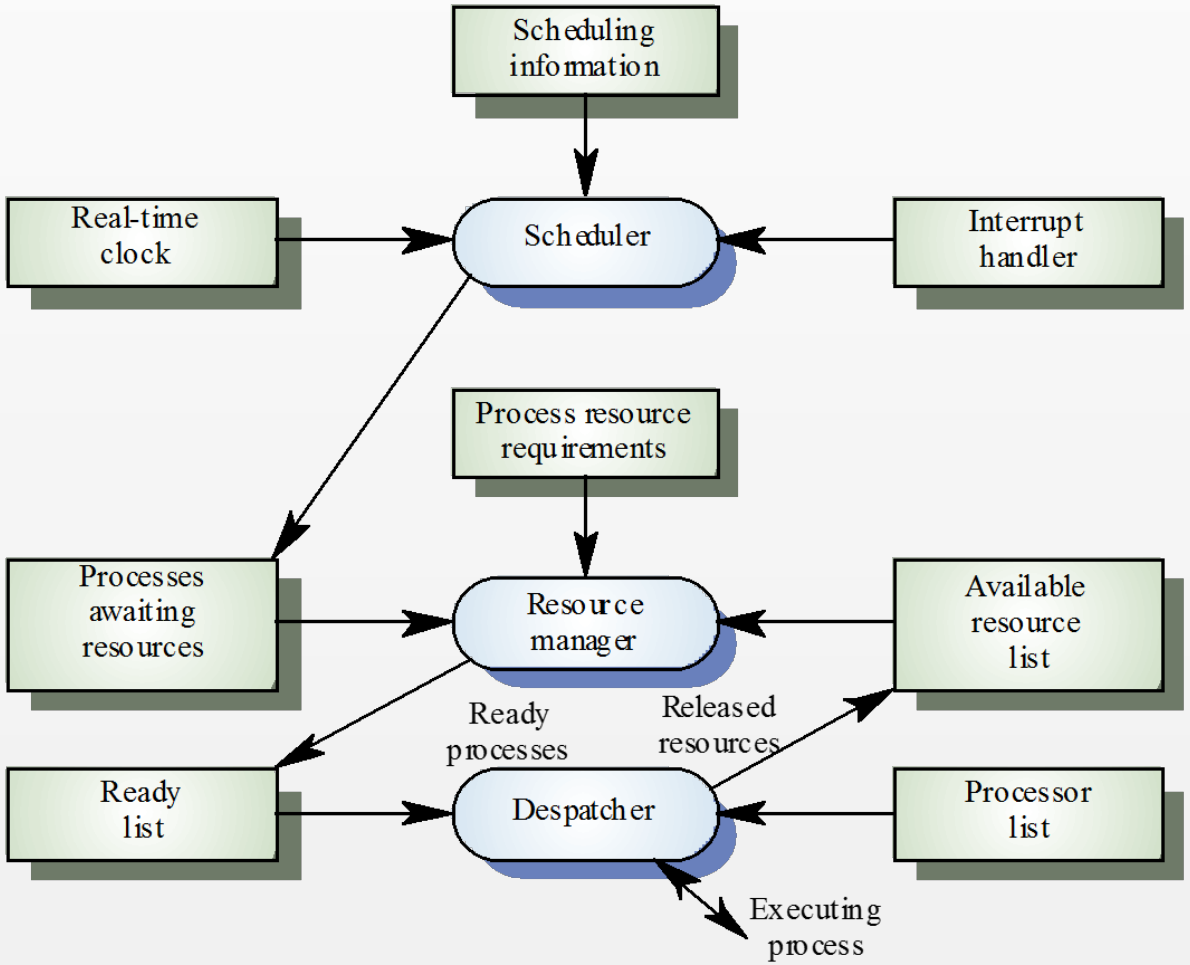


Figure 2.7. Microkernel architecture.

W.Ecker et al.; *Hardware Dependent Software, Principles and Practice*

Componentes de un RTOS



Componentes de un RTOS

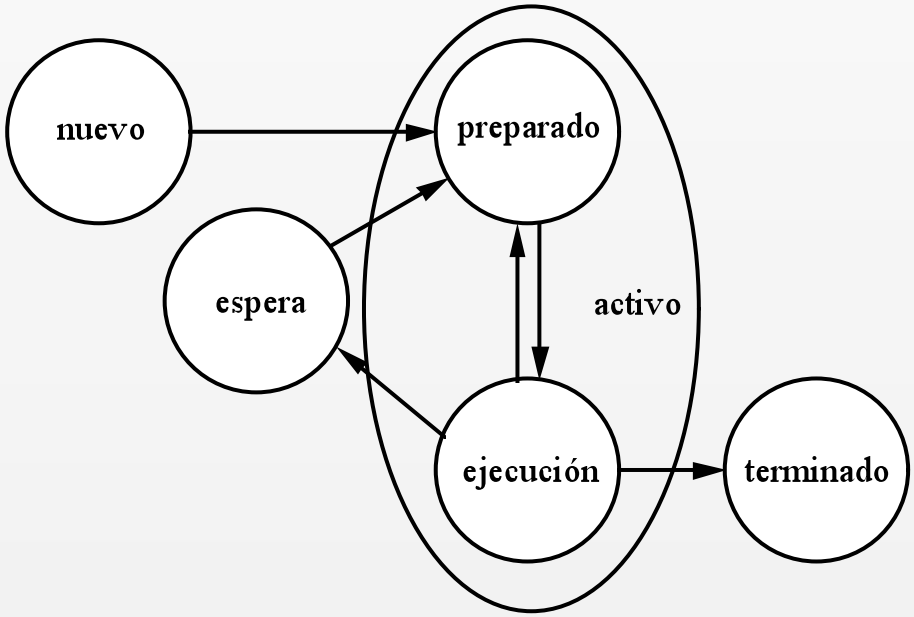
- **Programador (*scheduler*)**
  - Establece el orden de ejecución de los procesos
- **Ejecutor (*dispatcher*)**
  - Gestiona el inicio y la finalización de cada tramo de procesamiento, cambiando el contexto (stack, memoria, registros, etc.) para pasar de una tarea a otra
- **Administrador de memoria**
  - Gestiona la memoria disponible de manera dinámica
- **Servicios**
  - Drivers para acceder al hardware
  - Administrador de interrupciones de hardware o software
  - Etc.
- **Primitivas para la sincronización y comunicación entre procesos**

Otros componentes (para misión crítica)

- **Gestor de configuraciones (configuration manager)**
  - Gestiona la reconfiguración del sistema (reemplazo de componentes de hardware o actualización de software) sin interrumpir el funcionamiento del sistema
- **Gestor de fallos (fault manager)**
  - Detecta fallos de hardware y software, y toma acciones para asegurar la disponibilidad del sistema
- **Ejemplo: RadioBase de Telefonía Celular**
  - Alta disponibilidad (tolerancia a fallas mediante redundancia)
  - Actualizaciones de software y hardware sin detención
  - Reporte de estadísticas de funcionamiento y performance
  - Reconfiguración según necesidades
    - Por tráfico, variación del espectro, etc.

Estado de las tareas

- El núcleo mantiene un conjunto de listas de tareas en diferentes estados



Programación (Scheduling)

- **¿Cómo se decide el orden de ejecución de las tareas?**
  - Lo decide el **scheduler** del RTOS, en base al algoritmo de programación (o *scheduling algorithm*) definido al diseñar el sistema
- **Esas decisiones (o sea, la programación de tareas, o *scheduling*) tienen un rol crucial en el cumplimiento de los requisitos de temporización**
  - Ej., deben tener prioridad las tareas con deadline inminente
- **¿Qué hace el dispatcher mientras tanto?**
  - En aquellos momentos donde se pasó a procesar otra tarea, se ocupó de
    1. Tomar una de una **lista (ordenada) de tareas listas** que fue preparada por el scheduler
    2. Cambiar el contexto (o sea, hacer el ***context switch***)
    3. **Transferir el control** del procesador a la instrucción que corresponde, de la nueva tarea

Funcionamiento del scheduler

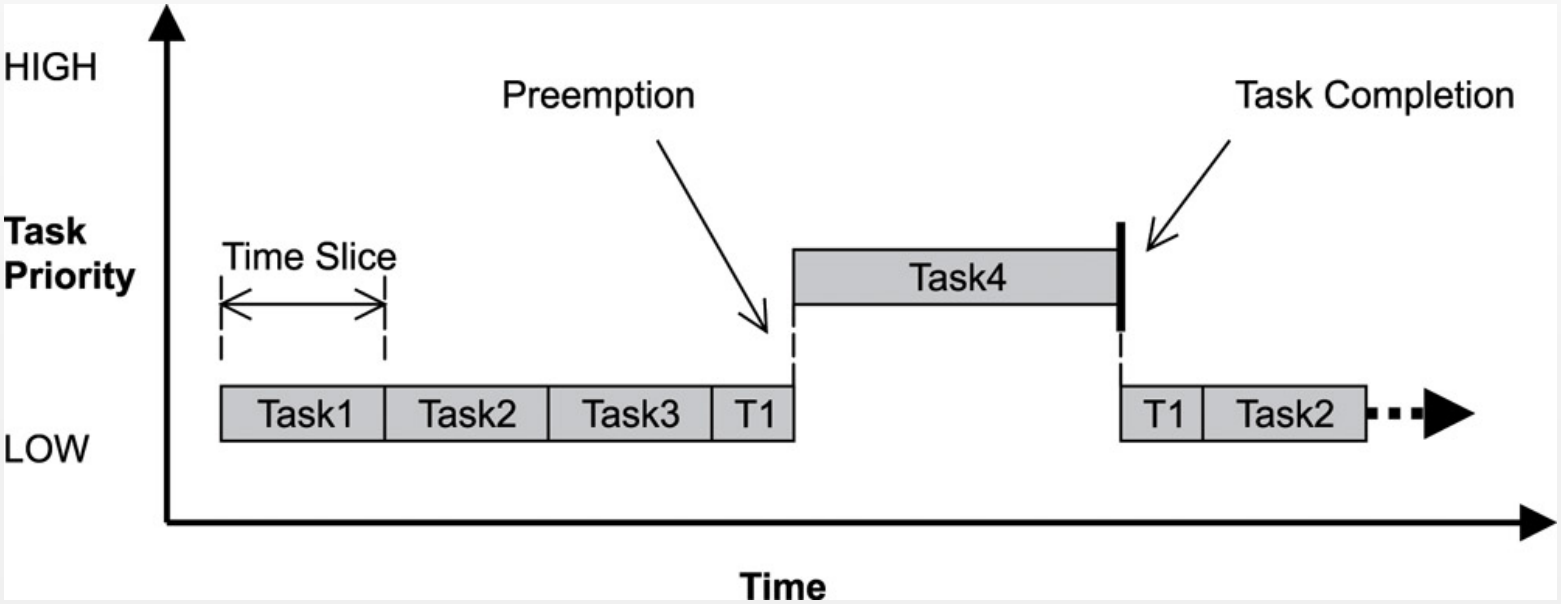
- **Típicamente, el scheduler se ejecuta a intervalos regulares**
  - Determinados por un timer o reloj de tiempo real (RTC)
- **Cuando lo hace, vuelve a elaborar la lista de tareas listas (esa que lee el dispatcher), dándoles un orden.**
- **Existen varios tipos de algoritmos para la elaboración de este orden**
  - Esos son los **algoritmos de scheduling** que ya se han estudiado

Scheduling con desalojo

- El programador asigna una prioridad a cada proceso
- Se ejecuta siempre el proceso de mayor prioridad, entre los que están listos
- Si aparece uno de mayor prioridad, se interrumpe la ejecución y se le da el control
- Las prioridades pueden ser fijas o dinámicas
  - Son dinámicas si pueden modificarse en tiempo de ejecución

Round-robin (en ronda)

- ***Round-robin*** **(en ronda)**
  - Los procesos se ejecutan siempre en la misma secuencia
  - Se usa cuando hay un grupo de tareas con las misma prioridad, asignándoles, a las tareas, fragmentos de tiempo (o *time slices*) de igual duración
  - Es útil combinarlo con el algoritmo anterior:



Sincronización y comunicación entre procesos

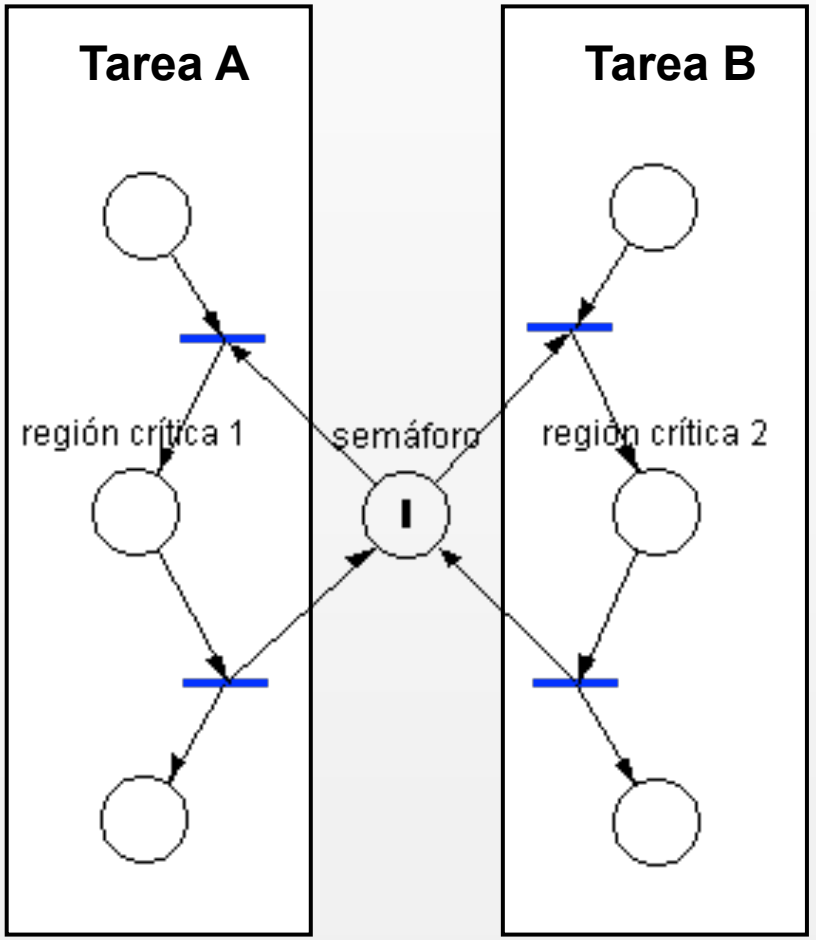
- **Imaginemos las siguientes tareas:**
  - Una **produce** ciertos datos
  - Otra **consume** esos datos
    - Mientras no reciba datos, esta tarea queda suspendida
- **Este es un problema típico**
  - Se llama *productor - consumidor*
- **Para resolverlo necesitamos medios para:**
  - Que la primera le comunique cada dato a la segunda
    - A esto se le llama ***comunicación entre procesos***
  - Que la primera señalice que le acaba de mandar un dato a la otra, para que ésta pase del estado "bloqueada" al estado "lista", y pueda leerlo
    - A esto se le llama ***sincronización de procesos***

Primitivas para la sincronización y comunicación

- **Los RTOS suelen ofrecer elementos para**
  - Comunicación
    - Ej., Memoria compartida. Colas de mensajes. Mailboxes
  - Sincronización
    - Ej., Semáforos. Mutex
- **Memoria compartida**
  - Se pueden definir partes de memoria accesibles por los dos procesos que necesitan comunicarse
  - Los datos pueden ser comunicados escribiéndolos
    - Pero cuidado, que un proceso no interprete un dato a medio escribir, como si estuviera escrito del todo
      - Es decir que se necesita un *handshaking.* Por ejemplo, una señal de sincronización
- **Colas (queues) de mensajes**
  - Son colecciones ordenadas de (estructuras de) datos
  - Se puede insertar un elemento al final, o sacar uno del principio
    - estructura *first-in, first-out* (FIFO)
  - Es una estructura de nivel más alto que la memoria compartida, que incluye sincronización

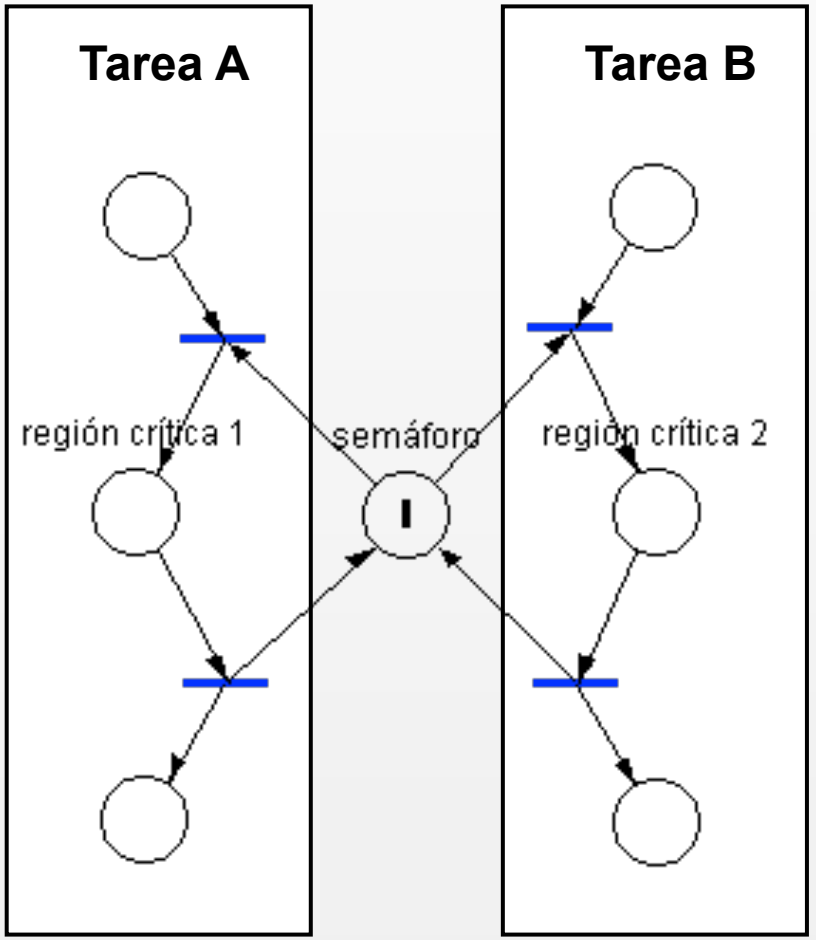
Sincronización mediante semáforos

- **Visto como red de Petri**
- **Sin la posición "semáforo" , estarían representando dos tareas (A y B) independientes**
  - Un token podría bajar por cada una de ellas, representando la ejecución independiente de los dos procesos
- **Sin embargo, el "semáforo" implica que hay un tipo de sincronización entre las dos tareas**
  - Si una está ejecutando su región crítica y la otra quiere entrar a la suya, va a tener que esperar a que la primera termine



Sincronización mediante semáforos

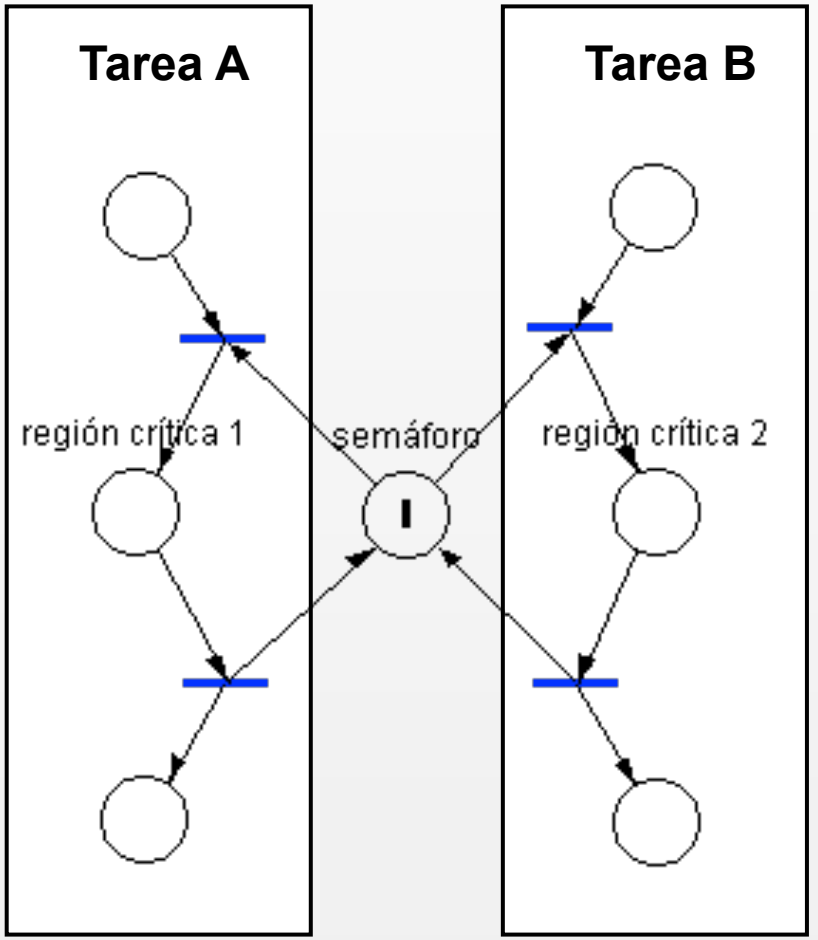
- **Esta aplicación de los semáforos se llama exclusión mutua**
  - Sirve, por ejemplo, para evitar que el proceso B esté leyendo, de una memoria compartida, algo que A está a medio escribir
  - También sirve para compartir recursos
- **Notar que los semáforos ofrecen una solución escalable**
  - O sea, podrían ser N las tareas compitiendo por el token
  - Y podrían haber M tokens, para que puedan estar, en sus regiones críticas, no más de M de ellas



Sincronización mediante semáforos

- **Los semáforos se usan mediante dos funciones:**
  - `wait()`
    - Para tomar el token
    - Son las transiciones de arriba
    - También se la llama `P()`
  - `signal()`
    - Para devolver el token
    - Son las transiciones de abajo
    - También se la llama `V()`

```
Tarea(semáforo sem) {
  pasos previos
  wait(sem)
  región crítica
  signal(sem)
  pasos posteriores
}
```



Problema productor - consumidor

Con semáforos:

```
Productor(semáforo exmu, dato) {
  while(1) {
    produce un dato
    wait(exmu)
    lo escribe en un buffer
                  compartido
    signal(exmu)
    signal(dato)
  }
}
```

```
Consumidor(semáforo exmu, dato) {
  while(1) {
    wait(dato)
    wait(exmu)
    lee un dato del
        buffer compartido
    signal(exmu)
    lo consume
  }
}
```

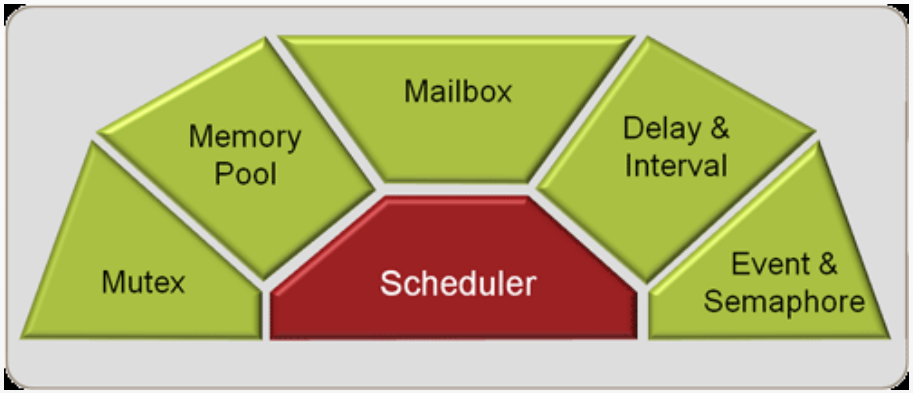
Con colas:

```
Productor(cola queue) {
  while(1) {
    produce el dato
    send(queue, dato)
  }
}
```

```
Consumidor(cola queue) {
  while(1) {
    receive(queue, dato)
    consume el dato
  }
}
```

Ejemplo de SOTR: RTX de Keil

- **SOTR gratuito de Keil**
- **Planificador:**
  - Round robin
  - Desalojo
  - Cooperativo sin desalojo



- **Comunicaciones entre tareas:**
  - Eventos
  - Semáforos
  - Mutex
  - Buzón de mensajes
- **Soporte para integración de interrupciones**
- **Gestión de temporizaciones y de memoria**

RTX-Tareas

- **Similar a una función de C.**
  - Lazo infinito
  - Tipo void.

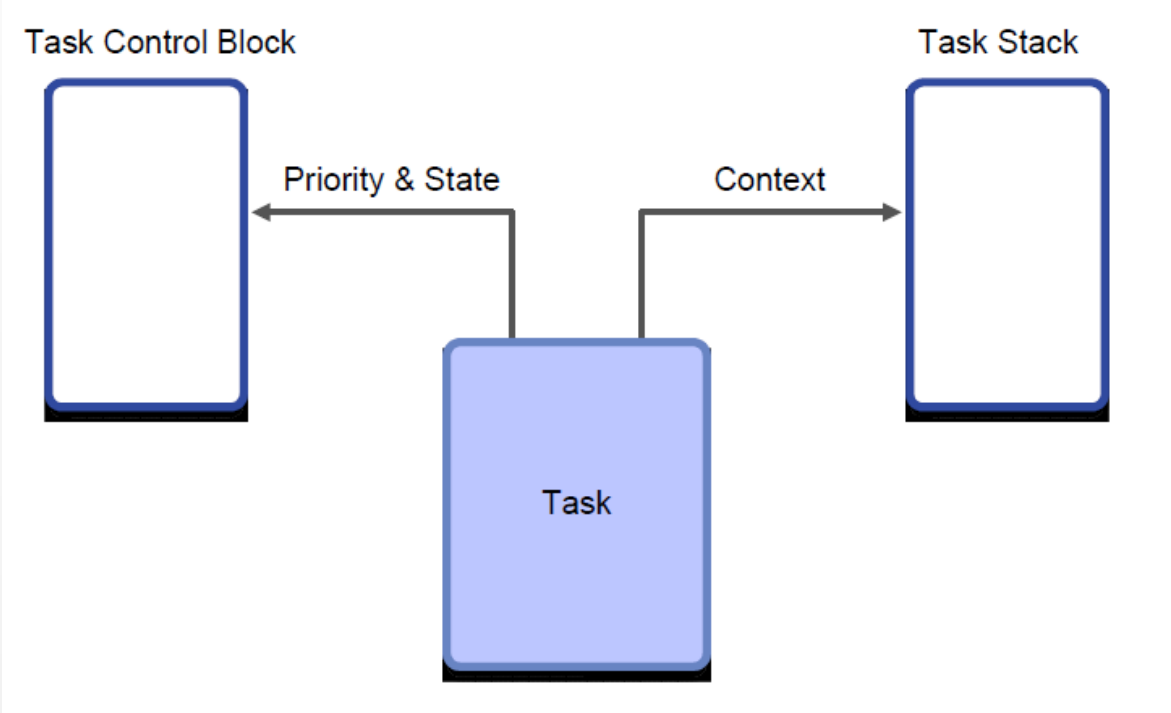
Procedure	Task

`unsigned int procedure (void) {`
`  …`
`  …`
`  return (val);`
`}`
	`__task void task (void) {`
	`  for (;;) {`
	`  …`
	`  }`
	`}`

- **Cada tarea tiene su bloque de control y su stack**
- **El planificador de tareas se activa al ritmo marcado por el timer SysTick.**

```
OS_TID id1, id2, id3;
```

RTX tareas



RTX tareas

- **Posibles estados de las tareas:**

Task	Description
RUNNING	The currently running TASK
READY	TASKS ready to run
WAIT DELAY	TASKS halted with a time DELAY
WAIT INT	TASKS scheduled to run periodically
WAIT OR	TASKS waiting an event flag to be set
WAIT AND	TASKS waiting for a group event flag to be set
WAIT SEM	TASKS waiting for a SEMAPHORE
WAIT MUT	TASKS waiting for a SEMAPHORE MUTEX
WAIT MBX	TASKS waiting for a MAILBOX MESSAGE
INACTIVE	A TASK not started or detected

RTX: estructura básica

Declaración tareas

```
__task  void task1 (void);
__task  void task2 (void);
OS_TID  tskID1,  tskID2;
```

Inicio de la primera tarea

```
void main (void)  {

  IODIR1 = 0x00FF0000;      // Do any C code you want
  os_sys_init (task1);      // Start the RTX call the first task
}
```

- Para asignar prioridad: *os_sys_init_prio()*
- Para asignar pila personalizada *os_sys_init_user()*

Creación de tareas

```
__task void task1 (void)  {
tskID2 = os_tsk_create (task2,0x10);      // Create the second task
                                          // and assign its priority.
tskID3 = os_tsk_create (task3,0x10);      // Create additional tasks
                                          // and assign priorities.
os_tsk_delete_self ();                    // End and self-delete this task
}
```

RTX: estructura básica

Opciones para crear tareas

- Normal

  os_tsk_create(task, prioridad)

- Con paso de parámetros

```
tskID3 = os_tsk_create_ex (Task3, priority, parameter);
```

- Con valores de pila personalizados

```
tskID4 = os_tsk_create_user (Task4, priority, &stk4, sizeof (stk4));
```

```
static U64 stk5 [400/8];

tskID5 = os_tsk_create_user_ex (Tsk5, prio, &stk5, sizeof (stk5), param);
```

RTX: cambios en las tareas

- **Prioridad**

```
OS_RESULT os_tsk_prio (tskID2, priority);
OS_RESULT os_tsk_prio_self (priority);
```

- **Borrado**

```
OS_RESULT = os_tsk_delete (tskID1);
            os_tsk_delete_self ();
```

- **Cesión de la CPU**

```
os_tsk_pass ();
```

- **Pueden crearse múltiples instancias de la misma tarea**

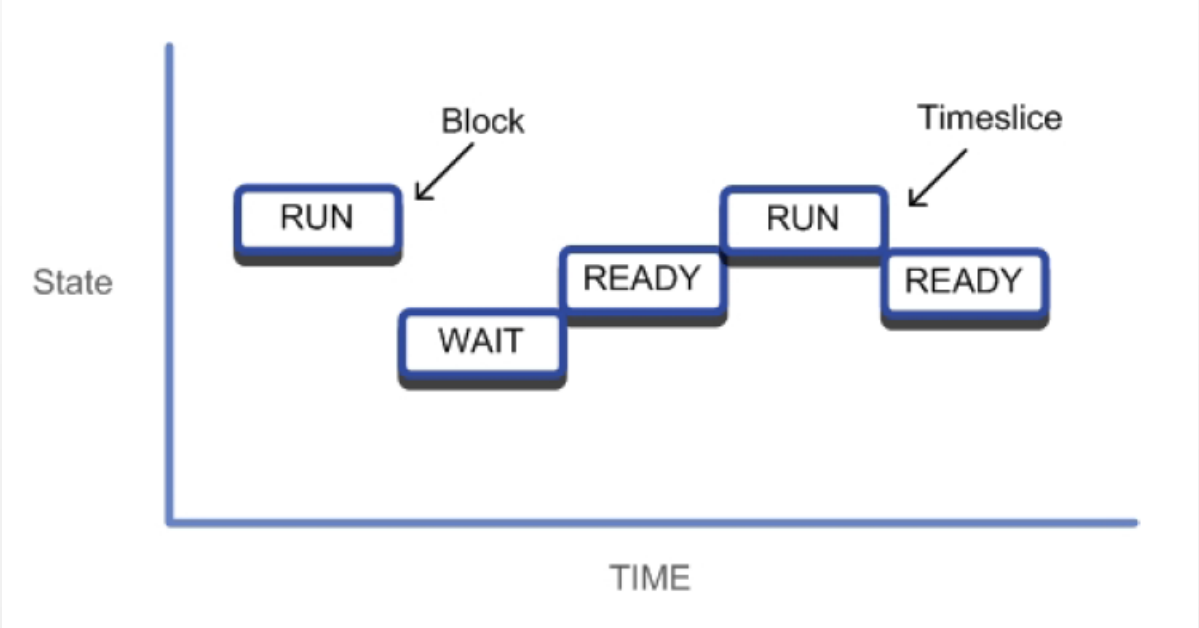
RTX: Temporizaciones

Retardos

- La tarea pasa a estado WAIT_DELAY

```
void os_dly_wait (unsigned short delay_time)
```

- Una vez finalizado el retardo, la tarea pasa a estado READY



RTX: temporizaciones

Ejecución periódica de tareas.

- Establecimiento del periodo de activación:

```
void os_itv_set (unsigned short interval_time)
```

- Paso a modo espera:

```
void os_itv_wait (void)
```

- La tarea pasa a modo WAIT_INT. Finalizado el periodo, la tarea pasa a modo READY.

RTX: temporizaciones

Temporizadores

- Creación:

```
OS_ID os_tmr_create (unsigned short tcnt, unsigned short info)
```

- Funcionan con cuenta regresiva. Al finalizar se activa la función

```
void os_tmr_call (U16 info)  {

   switch (info)  {
      case 0x01:
         …                          // user code here
      break ;
   }
}
```

Tarea idle

- Se activa si no hay otra tarea que ejecutar. Se define en RTX_Config.c

RTX: comunicaciones entre tareas

Eventos:

- Todas las tareas tienen 16 flags de eventos, almacenadas en el bloque de control de la tarea.
- Las tareas pueden quedar en espera temporizada hasta que otra tarea activa cierto número de flags.

```
OS_RESULT os_evt_wait_and (unsigned short wait_flags,
                           unsigned short timeout);

OS_RESULT os_evt_wait_or (unsigned short wait_flags,
                          unsigned short timeout);
```

- La tarea que espera para a estado WAIT_EVNT
- Cuando se activen los flags, la tarea pasa READY
- Los flags se pueden activar o borrar por otras tareas

```
void os_evt_set (unsigned short event_flags, OS_TID task);

void os_evt_clr (U16 clear_flags, OS_TID task);
```

- La tarea puede consultar sus flags:

```
which_flag = os_evt_get ();
```

RTX: Eventos e interrupciones

- **La gestión de interrupciones debe hacerse escribiendo rutinas ISR que activen (señalicen) los flags de la tarea asignada al tratamiento de la interrupción.**

```
void isr_evt_set (unsigned short event_flags, OS_TID task);
```

- **Ejemplo:**

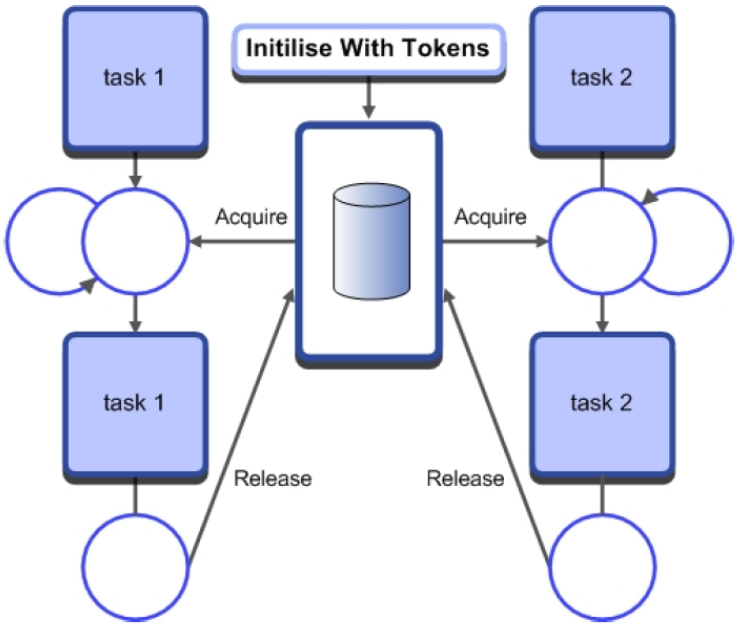
```
void Task3 (void)  {

  while (1)  {
    os_evt_wait_or (0x0001, 0xffff);  // Wait until ISR triggers an event
    …                                 // Handle the interrupt
  }                                   // Loop and go back to sleep
}
```

```
void IRQ_Handler (void) __irq  {

   isr_evt_set (0x0001, tsk3);     // Signal Task 3 with an event
   EXTINT = 0x00000002;            // Clear the peripheral interrupt flag
   VICVectAddr = 0x00000000;       // Signal end of interrupt to the VIC
}
```

RTX: Semáforos.

- **Son contenedores de marcas (Redes de Petri)**
  - Se emplean para sincronizar tareas o gestionar recursos compartidos.
  - Definición:

```
OS_SEM <semaphore>;
```



  - Inicialización en una tarea

```
void os_sem_init (OS_ID semaphore, unsigned short token_count);
```

RTX: Semáforos.

- **Las tareas pueden generar y/o consumir testigos del semáforo**
  - Si una tarea intenta consumir un testigo del semáforo y el semáforo no tiene, la tarea queda en espera temporizada.

```
OS_RESULT os_sem_wait (OS_ID semaphore, unsigned short timeout)
```

  - Si el semáforo tiene testigos, se disminuye en uno su número y la tarea sigue su ejecución.
  - Cuando una tarea ha terminado de usar el recurso asociado al semáforo, debe devolver el testigo:

```
OS_RESULT os_sem_send (OS_ID semaphore)
```

  - Los testigos también se pueden generar en las ISR:

```
void isr_sem_send (OS_ID semaphore)
```

RTX: Semáforos

- **Ejemplos de uso (‘The Little book of semaphores” by Allen B. Downy)**

```
os_sem semB;

__task void task1 (void)  {
   os_sem_init (semB, 0);
   while (1)  {
      os_sem_send (semB);
      FuncA();
   }
}
```

```
__task void task2 (void)  {

   while (1)  {
      os_sem_wait (semB, 0xFFFF);
      FuncB();
   }
}
```

  - Multiplex (limitación del número de tareas que pueden acceder a un recurso)

```
os_sem Multiplex;

void task1 (void) __task  {

   os_sem_init (Multiplex, 5);
   while (1)  {
      os_sem_wait (Multiplex, 0xFFFF);
      ProcessBuffer ();
      os_sem_send (Multiplex);
   }
}
```

RTX: Semáforos

Rendezvous:

- Dos tareas se esperan para continuar, pudiendo llegar primero cualquiera de las dos.

```
os_sem Arrived1, Arrived2;

__task void task1 (void)  {

  os_sem_init (Arrived1, 0);
  os_sem_init (Arrived2, 0);
  while (1)  {
    FuncA1 ();
    os_sem_send (Arrived1);
    os_sem_wait (Arrived2, 0xFFFF);
    FuncA2 ();
  }
}
```

```
__task void task2 (void)  {


  while (1)  {
    FuncB1 ();
    os_sem_send (Arrived2);
    os_sem_wait (Arrived1, 0xFFFF);
    FuncB2 ();
  }
}
```

Precauciones en el uso de los semáforos

- Hay que procurar no dejar a los semáforos sin testigos.

RTX: Mutex

- **Es un caso especial de semáforo, con un solo testigo.**
- **Se emplean para controlar el acceso a recursos del ucontrolador como periféricos, etc.**
- **Uso:**
  - Declaración
  - Inicialización

```
os_mut_init (OS_ID mutex);
```

  - Bloqueo

```
os_mut_wait (OS_ID mutex, U16 timeout);
```

  - Liberación

```
os_mut_release (OS_ID mutex);
```

RTX: Mailboxes

- **Permiten intercambiar datos entre tareas.**
- **Uso:**
  - Definir número de mensajes del buzón
  - Definir la estructura de datos de cada mensaje
  - Reservar un bloque de memoria para dar cabida a los mensajes
  - Formatear el bloque de memoria.
  - Enviar:
    - Para insertar un mensaje en el buzón, hay que buscar el primer sitio libre y almacenar el mensaje
    - Después se envía el mensaje.
    - Los mensajes se organizan el una cola FIFO
  - Leer mensaje:
    - Esperar el mensaje
    - Leerlo
    - Liberar el espacio ocupado por el mensaje

RTX: Mailboxes

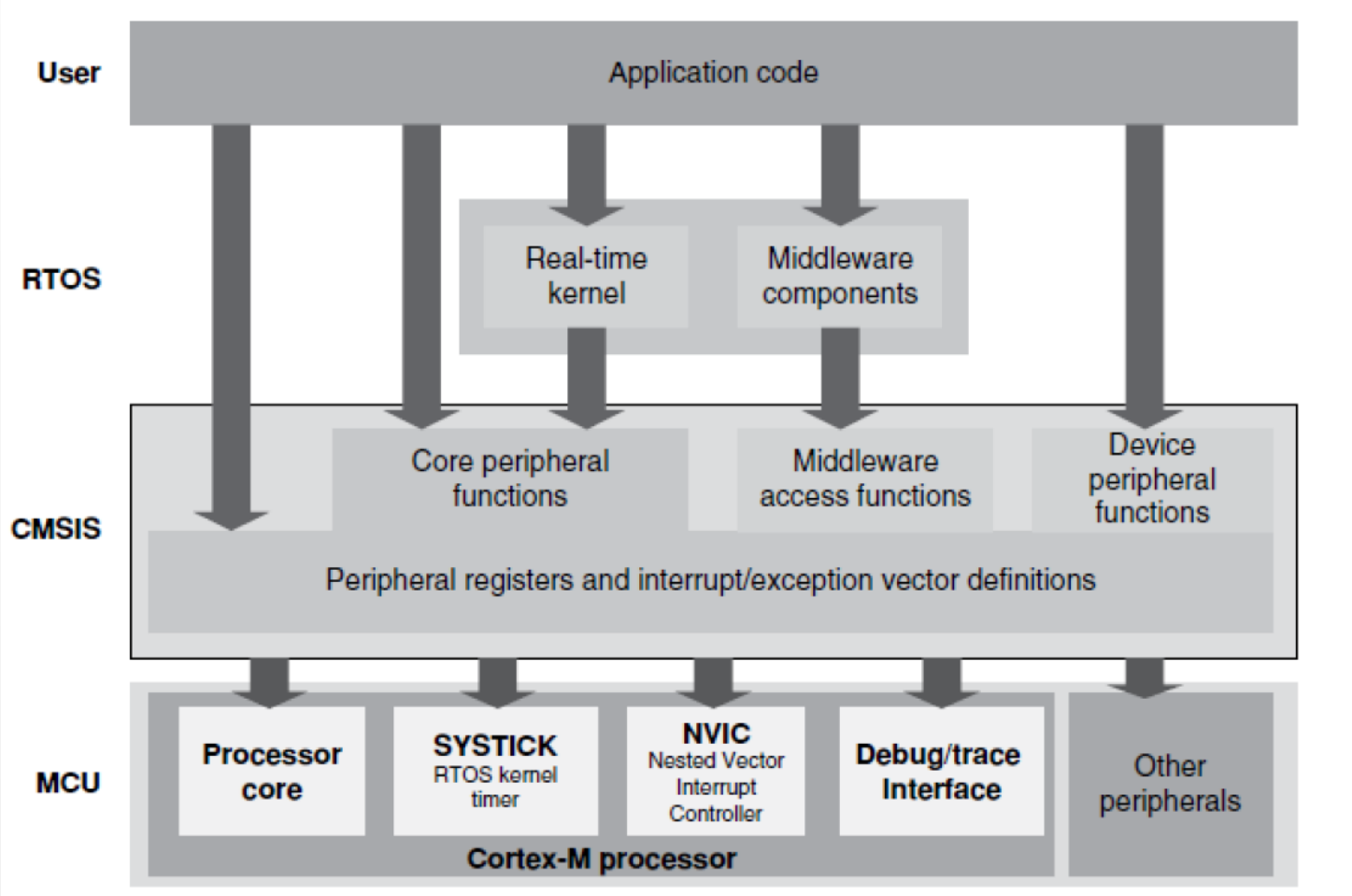
```
#include <rtl.h>

os_mbx_declare (MsgBox, 16);                /* Declare an RTX mailbox */
U32 mpool[16*(2*sizeof(U32))/4 + 3];        /* Reserve a memory for 16 messages */

__task void rec_task (void);

__task void send_task (void) {
  /* This task will send a message. */
  U32 *mptr;

  os_tsk_create (rec_task, 0);
  os_mbx_init (MsgBox, sizeof(MsgBox));
  mptr = _alloc_box (mpool);                /* Allocate a memory for the message */
  mptr[0] = 0x3215fedc;                     /* Set the message content. */
  mptr[1] = 0x00000015;
  os_mbx_send (MsgBox, mptr, 0xffff);       /* Send a message to a 'MsgBox' */
  os_tsk_delete_self ();
}

__task void rec_task (void) {
  /* This task will receive a message. */
  U32 *rptr, rec_val[2];

  os_mbx_wait (MsgBox, &rptr, 0xffff);      /* Wait for the message to arrive. */
  rec_val[0] = rptr[0];                     /* Store the content to 'rec_val' */
  rec_val[1] = rptr[1];
  _free_box (mpool, rptr);                  /* Release the memory block */
  os_tsk_delete_self ();
}

void main (void) {
  _init_box (mpool, sizeof(mpool), 2*sizeof(U32));
  os_sys_init(send_task);
}
```

RTX: Bloqueo del scheduler.

- **Si se quiere que una tarea sea ejecutada sin interrupciones, se puede bloquear la entrada del 'scheduler'**

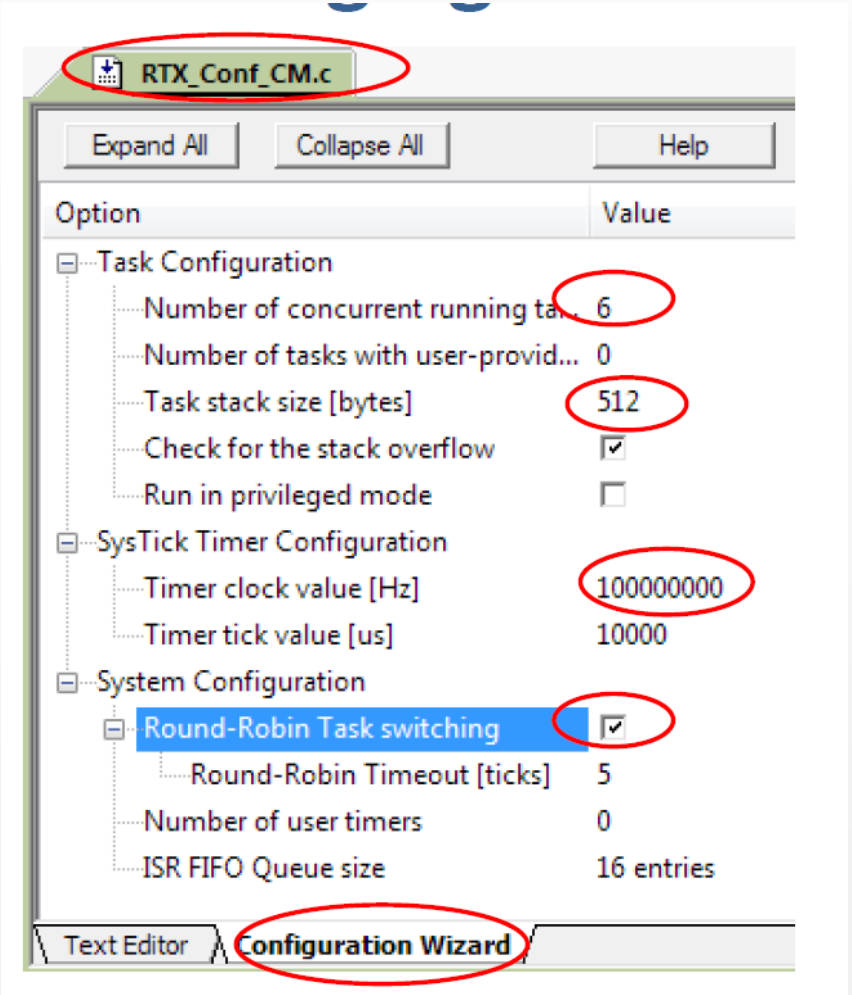
```
tsk_lock ();
   do_critical_section ();
tsk_unlock ();
```

RTX: creación aplicaciones



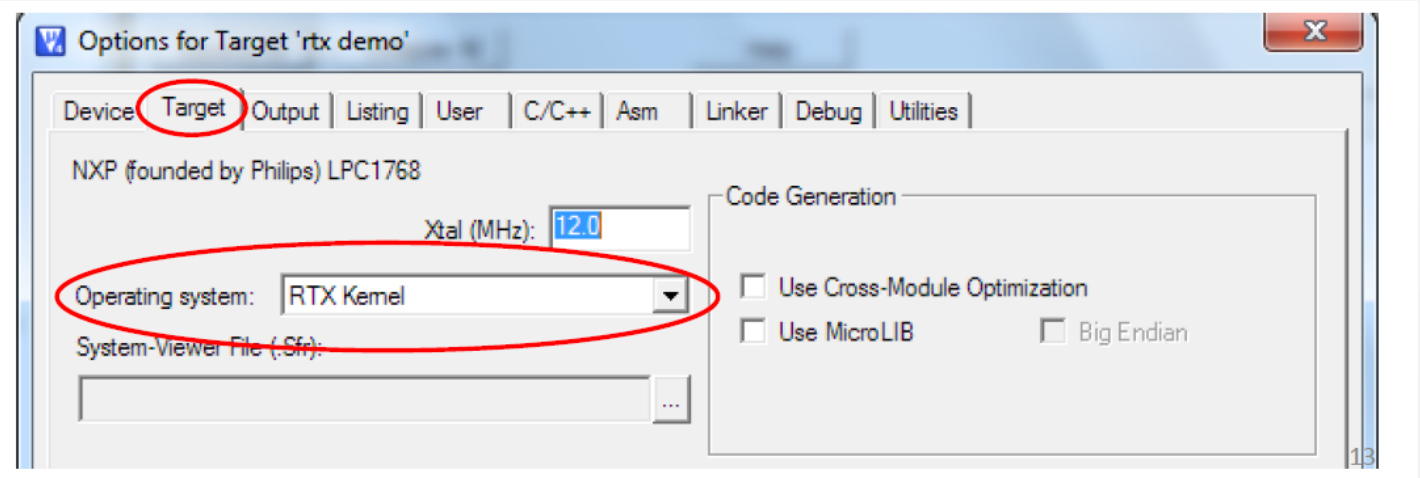
RTX: creación aplicaciones

- Crear un proyecto para LPC17xx
- Añadir en la carpeta startup el fichero RTX_Conf_CM.c
- Configurar RTX



RTX: creación aplicaciones

- **Seleccionar RTX Kernel como sistema operativo**



RTX: creación aplicaciones

- **En el fichero de código fuente:**

```
#include <RTL.h>
```

- **Para iniciar el sistema operativo (tarea de reloj) y la primera tarea se debe emplear:**

```
os_sys_init(init)
```

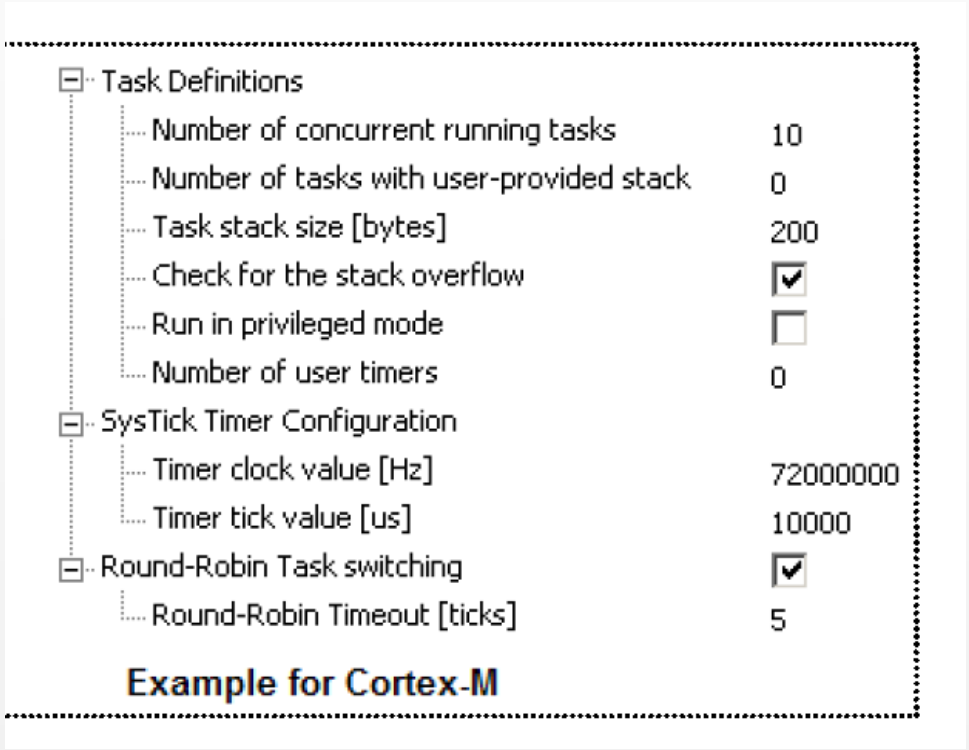
- **El código de las tareas debe comenzar con __task**

```
__task void init(void);
```

- **Si es necesario modificar alguna primitiva de RTX, el código fuente se encuentra en:**
  - …Keil\ARM\RL\RTX\SCR\CM

RTX: Configuración

- **Parámetros configurables:**



RTX: Configuración

- **Datos técnicos:**

Technical Data	RTX	
max Tasks	250	
Events/Signals	16 per task	
Semaphores, Mailboxes, Mutexes	unlimited	
min RAM	2 – 3 KBytes	
	ARM7/ARM9	**Cortex-Mx**
max Code Space	4.2 KBytes	4.0 KBytes
Hardware Needs	1 on-chip timer	SysTick timer
Task Priorities	1 – 254	1 – 254
Context Switch	< 7 µsec @ 60 MHz	< 4 µsec @ 72 MHz
Interrupt Lockout	3.1 µsec @ 60 MHz	not disabled by RTX

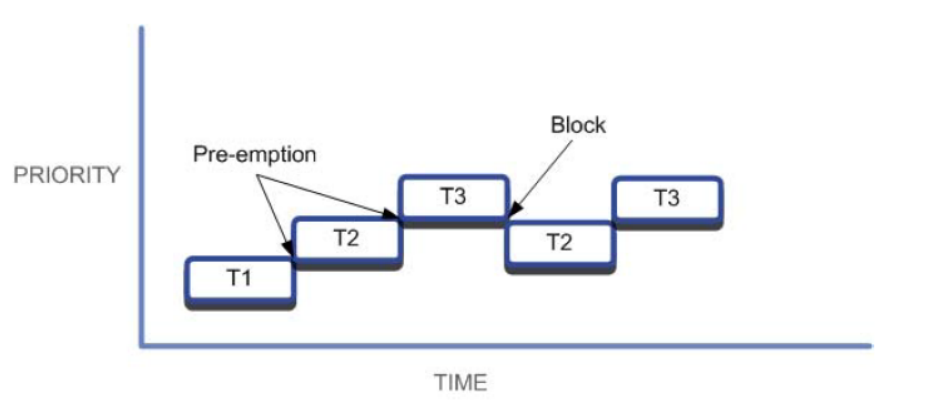
RTX: Configuración

- **Stack por defecto de las tareas.**
  - Si alguna tarea requiere más pila, se crea con la primitiva *os_task_create_usr()*
- **Número máximo de tareas en ejecución.**
- **Chequeo de saturación de pila de la tarea.**
  - Si se llena, se activa la función *os_stk_overflow()*
- **'Run in privileged mode':**
  - Permite a las tareas trabajar o no en modo privilegiado.
  - El núcleo RTX siempre trabaja en modo privilegiado.
- **Temporizador del sistema**
  - Cada vez que venza el temporizador, se activa la tarea encargada de gestionar la asignación de la CPU. Emplea el timer SysTick. Se puede configurar el tiempo entre activaciones (tick del sistema), por defecto en 10ms.

RTX: Configuración

Opciones de gestión de la CPU:

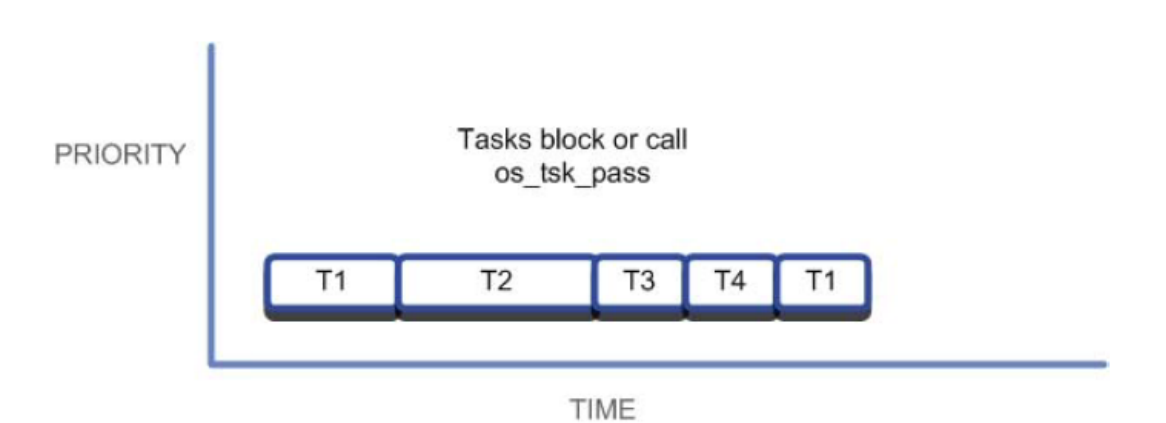
- Round Robin con desalojo:
  - Las tareas de la misma prioridad en estado READY van ocupando la CPU de manera alternativa durante un slot de tiempo (fijado en número de ticks) configurable.
  - Si aparece una tarea de más prioridad, se le asigna la CPU.
- Con desalojo:
  - A cada tarea se le asigna una prioridad distinta.
  - La tarea READY con mayor prioridad ocupa la CPU.



RTX: Configuración

Opciones de gestión de la CPU:

- Multitarea cooperativa:
  - Todas las tareas tienen la misma prioridad.
  - Las tareas se ejecutan hasta que se termina, se bloquean o hasta que ella lo decida llamando a la primitiva *os_tsk_pass()*



Inversión de prioridad:

- El núcleo RTX evita la inversión de prioridad implementando la herencia de prioridad en caso de bloqueo.

RTX: Ejemplos

Gestión Round-Robin

```
int counter0;
int counter1;

__task[1] void job0 (void)  {
  os_tsk_create (job1, 1);                    // start job 1

  while (1)  {                                 // endless loop
    counter0++;                                // Increment counter 0
  }
}

__task void job1 (void)  {
  while (1)  {                                 // Endless loop
    counter1++;                                // Increment counter 1
  }
}

main (void)  {                                 // the main function
  os_sys_init (job0);                          // starts only job 0
}
```

RTX: Ejemplos

Temporizaciones

```
int counter0;
int counter1;

__task void job0 (void)  {
  os_tsk_create (job1, 1);                    // start job 1

  while (1)  {
    counter0++;                                // Increment counter 0
    os_dly_wait (3);                           // Wait 3 timer ticks
  }
}

__task void job1 (void)  {
  while (1)  {
    counter1++;                                // Increment counter 1
    os_dly_wait (5);                           // Wait 5 timer ticks
  }
}
```

RTX: Ejemplos

Espera de activación de eventos

```
long i0, save_i0, i1;
OS_TID id1;                                  // task ID for event transmits

__task void job0 (void)  {
  id1 = os_tsk_create (job1, 1);            // start job 1

  while (1)  {
    i0++;
    if (i0 > 1000000)  {                     // when i1 reaches 1000000
      i0 = 0;                                // clear i1
      os_evt_set (1, id1);                   // set event '1' on job1
    }
  }
}

__task void job1 (void)  {
  while (1)  {
    os_evt_wait_or (1, 0xffff);              // wait for event '1'
    save_i0 = i0;                            // save value of i0
    i1;                                      // count events in i1
  }
}
```

Resumen (I)

- **Una aplicación multitarea está compuesta por:**
  - Un conjunto de tareas
  - Un núcleo
- **Las tareas compiten por**
  - Tiempo de ejecución (CPU)
  - Memoria
  - Otros recursos (timers, E/S, ...)
- **El núcleo gestiona todos estos recursos y ofrece servicios a las tareas**
  - Reparto del tiempo de procesador
  - Creación y destrucción de tareas
  - Primitivas de comunicación y sincronización entre tareas
  - Reloj y funciones basadas en el tiempo (delays, time outs)
  - Captura todas las interrupciones y da servicio a algunas
  - Gestión de memoria

Resumen (II)

- **Una tarea está compuesta por:**
  - Un código
  - Una pila propia
  - Variables globales
- **El núcleo mantiene una estructura de datos por cada tarea: el TCB ("task control block")**
  - Identificación de la tarea
  - Punto de entrada
  - Base de la pila
  - Prioridad
  - Contador de programa
  - Puntero de pila
  - Contexto
    - Registros de la CPU
    - Otras variables
- **Los núcleos de sistemas empotrados utilizan siempre la planificación basada en prioridades:**
  - Planificación expulsiva / no expulsiva
  - Estática
- **La dificultad de la programación concurrente estriba en las interacciones de los procesos:**
  - Cooperación para un fin común
  - Competencia por el uso de recursos
  - Son necesarias operaciones de comunicación y sincronización entre procesos:
    - Sincronización: cumplir restricciones en el orden en el que se ejecutan sus acciones
    - Comunicación: paso de información de un proceso a otro
  - Hay dos formas de realizarlo:
    - Datos compartidos
    - Paso de mensajes

Resumen (III)

- **Dos procesos compiten cuando comparten:**
  - un recurso
  - una variable
- **El acceso al recurso o a la variable debe ser en exclusión mutua.**
- **Sección crítica: secuencia de instrucciones que debe ejecutarse en exclusión mutua**
- **Mecanismos de sincronización ofrecidos por un núcleo**
  - Prioridades
  - Espera ocupada (busy waiting)
  - Semáforos

Resumen (IV)

- **Evitar expulsiones cuando se ejecuta una sección crítica**
  - Enmascarar interrupciones
    - No entra el núcleo, ni el reloj, ...
  - Elevar al máximo la prioridad del código
    - Posibilidad de cambiar en tiempo de ejecución la prioridad de una tarea.
- **Semáforos**
  - Ventajas
    - Mecanismo simple y eficiente
    - Permite exclusión mutua y sincronización condicional
    - Evita las esperas ocupadas (busy waiting)
  - Inconvenientes
    - Mecanismo de bajo nivel: poco estructurado
    - Su uso queda disperso por los procesos
    - Es fácil cometer errores: un solo wait o signal mal colocado puede ser fatal

Referencias

- Handbook of Real Time and Embedded Systems. Chapman And Hall. 2007
- Real-Time Concepts for Embedded Systems by Qing Li and Carolyn Yao. ISBN:1578201241 .CMP Books © 2003
- RTX: http://www.keil.com/support/man/docs/rlarm/rlarm_ar_artxarm.htm