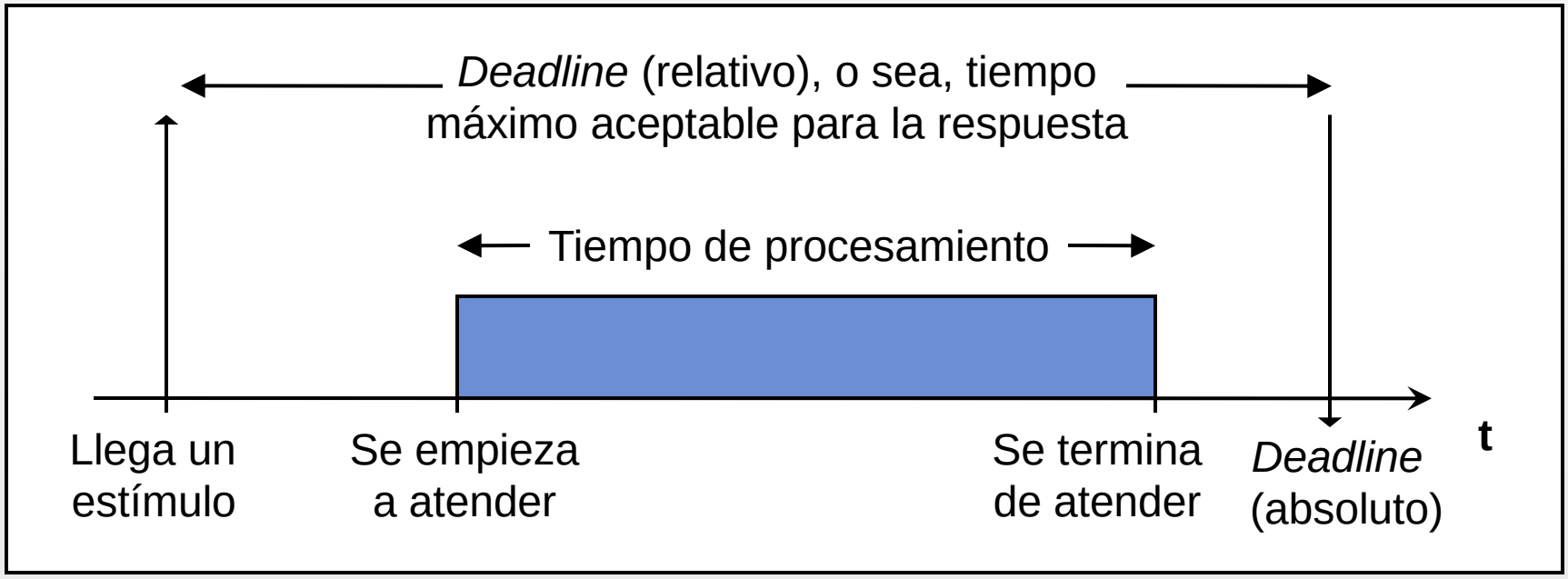
Sistemas de Tiempo Real

(*Real-Time Systems*)

Seminario de Electrónica: Sistemas Embebidos
1º cuatrimestre de 2010
Facultad de Ingeniería, UBA

Definiciones

- **Un sistema de tiempo real (STR o *real-time system* o RTS) es aquel cuyo correcto funcionamiento depende de que las salidas "lleguen a tiempo"**
  - 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



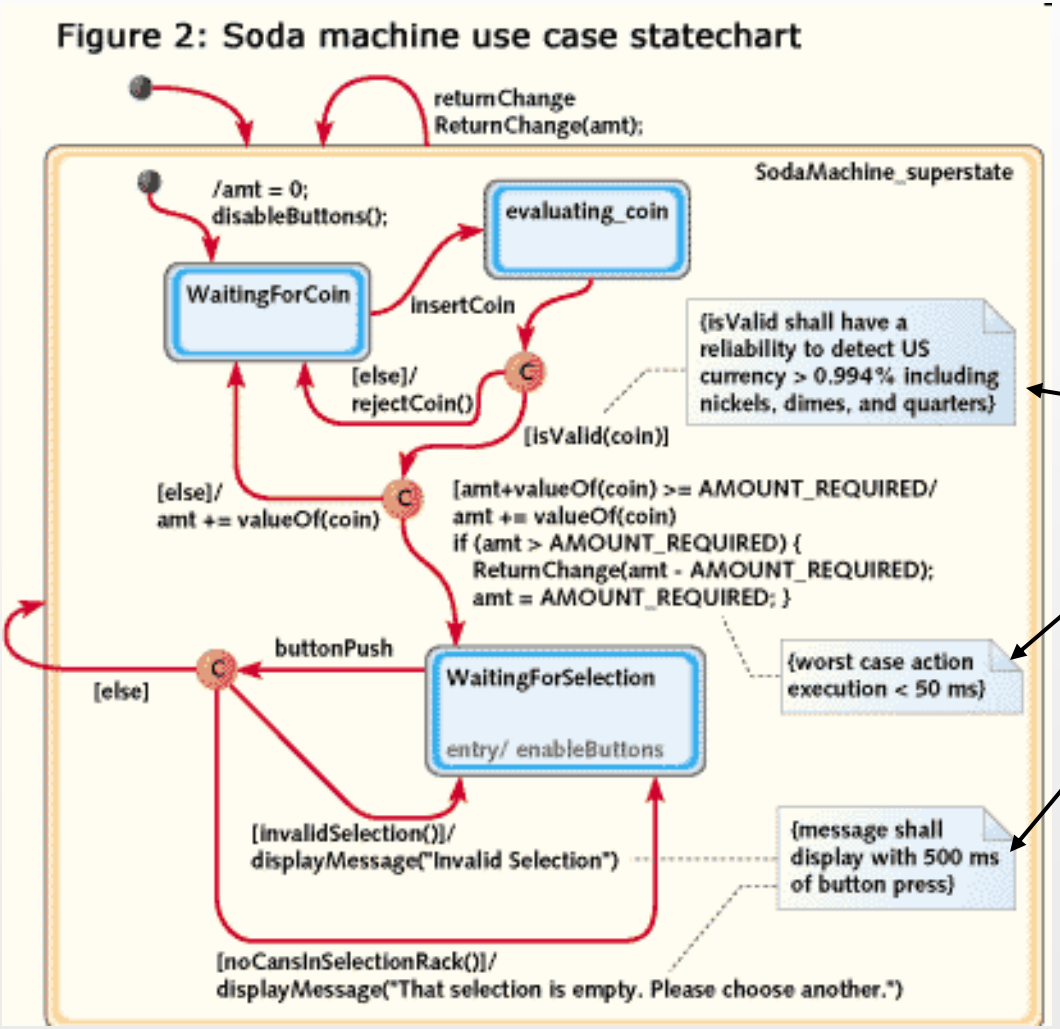
Se distinguen dos tipos:

- **STR *duro* (*hard RTS*)**
  - Es *duro* cuando el incumplimiento de un deadline implica un **funcionamiento incorrecto**
    - Ejemplos:
      - El sistema ABS (*anti-lock breaking system*) de un auto
      - 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
- **¿Pueden dar más ejemplos?**

Requerimientos de un STR

- **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 duros**
  - O sea, como si sus deadlines fueran estrictos
- **Como siempre, para especificar y refinar los requerimientos, podemos recurrir a:**
  - Diagramas de secuencia
  - Statecharts
  - Otros modelos de computación y lenguajes de modelado

Ejemplo



Notas con requermientos de temporización y de calidad de servicio (*quality of service* o QoS)

B.P.Douglass; “Capturing Real-Time Requirements “ <http://www.embedded.com/story/OEG20011016S0126>

Requerimientos de temporización

- **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

- **Como ven, 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 sin librerías ni nada**
  - 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, que vaya colgado de 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

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 al 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*

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
    - O sea que 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 y lo vamos a usar en la práctica

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**

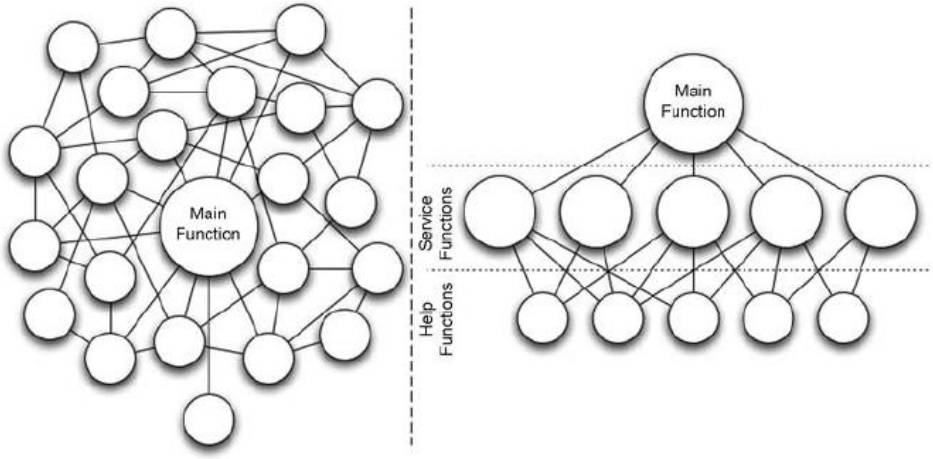


Figure 2.6. Unstructured vs. structured monolithic kernel.

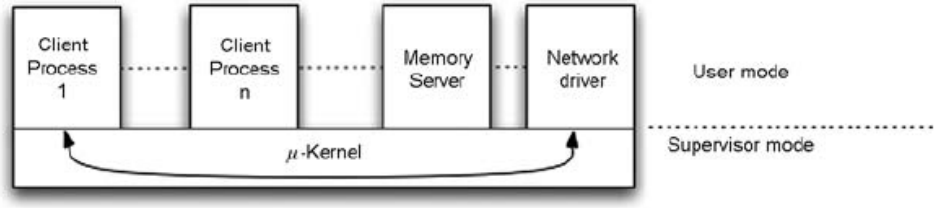


Figure 2.7. Microkernel architecture.

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

Componentes de un RTOS

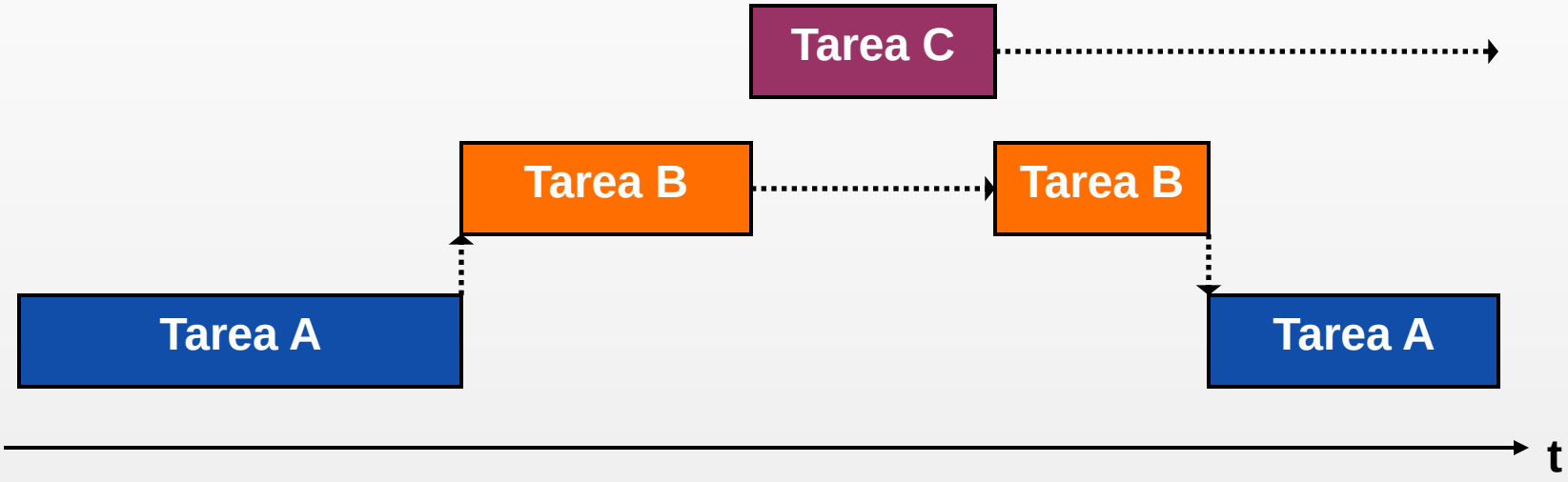
- **Programador (*scheduler*)**
  - Establece el orden de ejecución de los procesos
- **Ejecutor (*dispatcher*)**
  - Gestiona el inicio y la finalización de cada tramo de procesamiento, cambiando el contexto (stack, memoria, registros, etc.) para pasar de una tarea a otra
- **Administrador de memoria**
  - En arquitecturas con µkernel, éste suele contener al scheduler, al dispatcher y al administrador de memoria
- **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 fallas (fault manager)**
  - Detecta fallas 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.

Multitasking

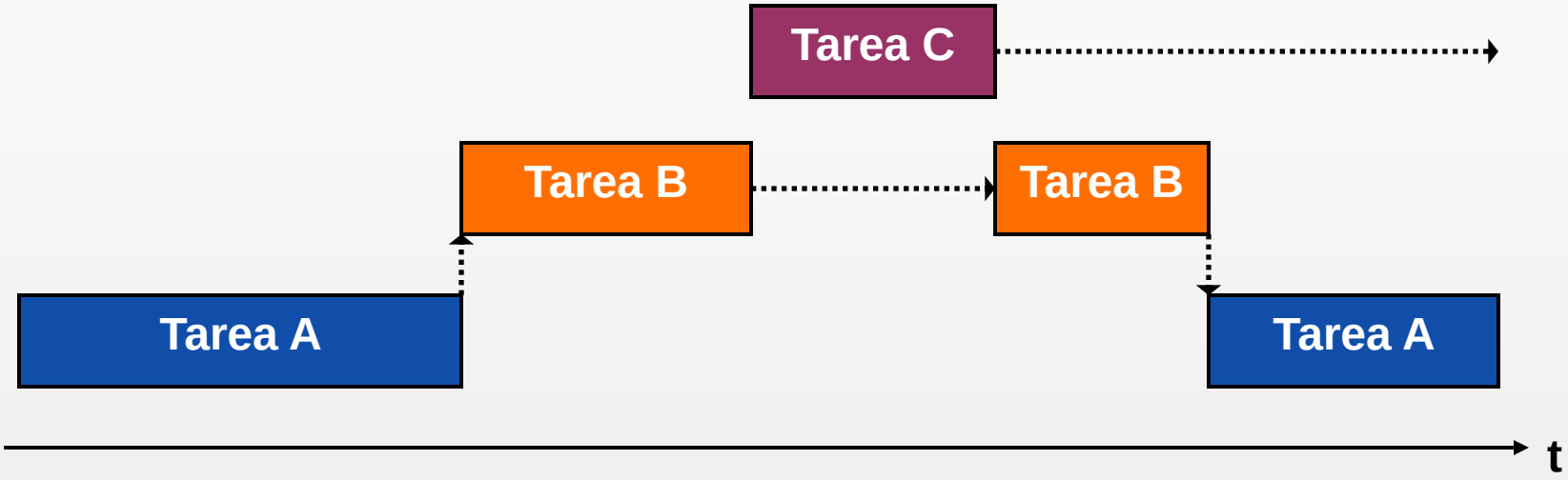
- **Como dijimos antes, el procesador (usualmente) se reparte entre distintas tareas, creando la ilusión del procesamiento concurrente**



1. En el ejemplo, el procesador primero está ejecutando la tarea A.
2. Esta le hace un pedido a la B, así que se transfiere el control del procesador a ésta última, hasta que requiere la respuesta de un periférico.
3. (Continúa)

Multitasking

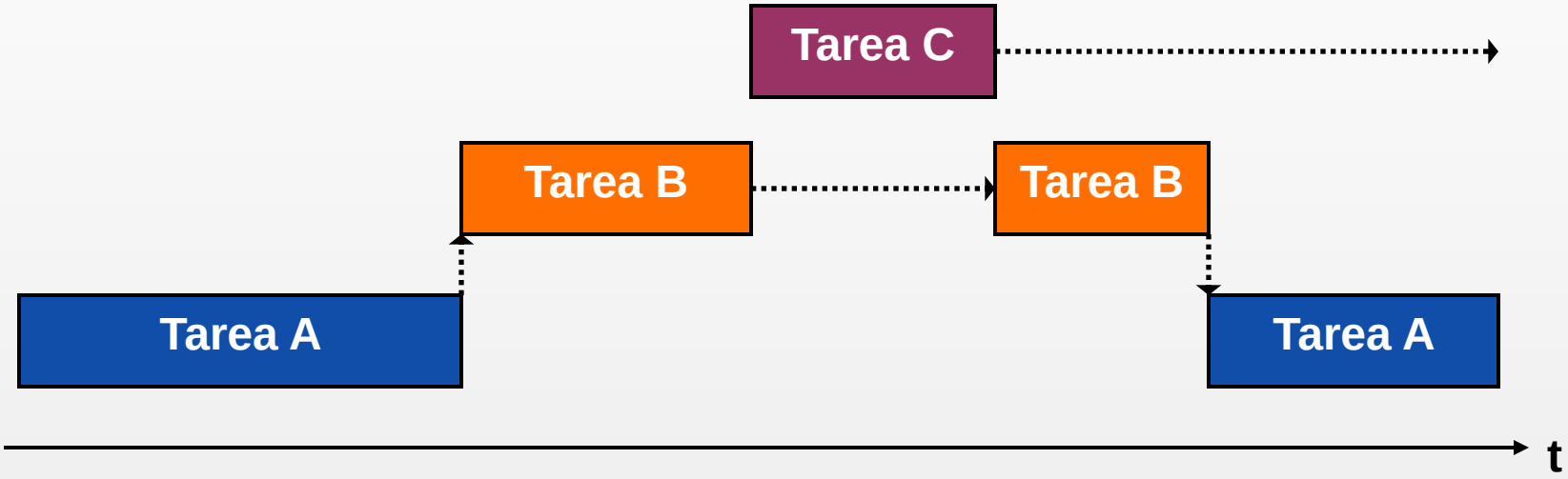
- **Como dijimos antes, el procesador (usualmente) se reparte entre distintas tareas, creando la ilusión del procesamiento concurrente**



3. Mientras espera, se le da el control a la tarea C, hasta que ésta termina lo que hacía y llama a una función especial del scheduler, mediante la cual queda en espera mientras no tenga nada que hacer
4. El procesador vuelve con B, dado que la respuesta que esperaba ya llegó.
5. B termina de realizar el pedido de A, así que se le devuelve el control a ésta.

Multitasking

- **Como dijimos antes, el procesador (usualmente) se reparte entre distintas tareas, creando la ilusión del procesamiento concurrente**



- **Como vemos, cada tarea está en uno de tres estados:**
  1. **Lista** para ser atendida por el procesador
  2. Siendo **ejecutada** por el procesador
  3. **Bloqueada** (también se le dice ***suspendida***)
     - Que es cuando está esperando una respuesta de I/O o una señal de otro proceso para pasar al primer estado

Programación (Scheduling)

- **¿Quién decidió que A esté seguido de B, si podía haber arrancado C ahí? ¿O que se haya ejecutado C en el intervalo, en lugar de alguna otra tarea (D, E, etc.) que estuviera lista?**
  - Lo decidió 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 hizo 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

Preemptive Scheduling

- **En el ejemplo anterior, el cambio de una tarea a otra se da sólo cuando el proceso en ejecución solicita un servicio del sistema**
  - Puede ser porque realmente requiere el servicio (ej., un acceso a un periférico) o porque está programado que ahí debe liberar al procesador (como lo había hecho C)
- **Pero la mayoría de los sistemas operativos pueden también interrumpir la tarea que se está ejecutando, cuando pasa demasiado tiempo sin "devolver" el procesador**
  - A esto se le dice *programación preventiva* (o ***preemptive scheduling*** o *preemptive multitasking)*
    - Para realizarlo, se necesita algún **timer** o **reloj de tiempo real** (RTC)
    - Es *preventiva* porque no se depende de que las tareas devuelvan el control a tiempo
  - A la programación de tareas que no hace estas interrupciones, se les dice programación cooperativa (o ***cooperative scheduling***)

Funcionamiento del scheduler

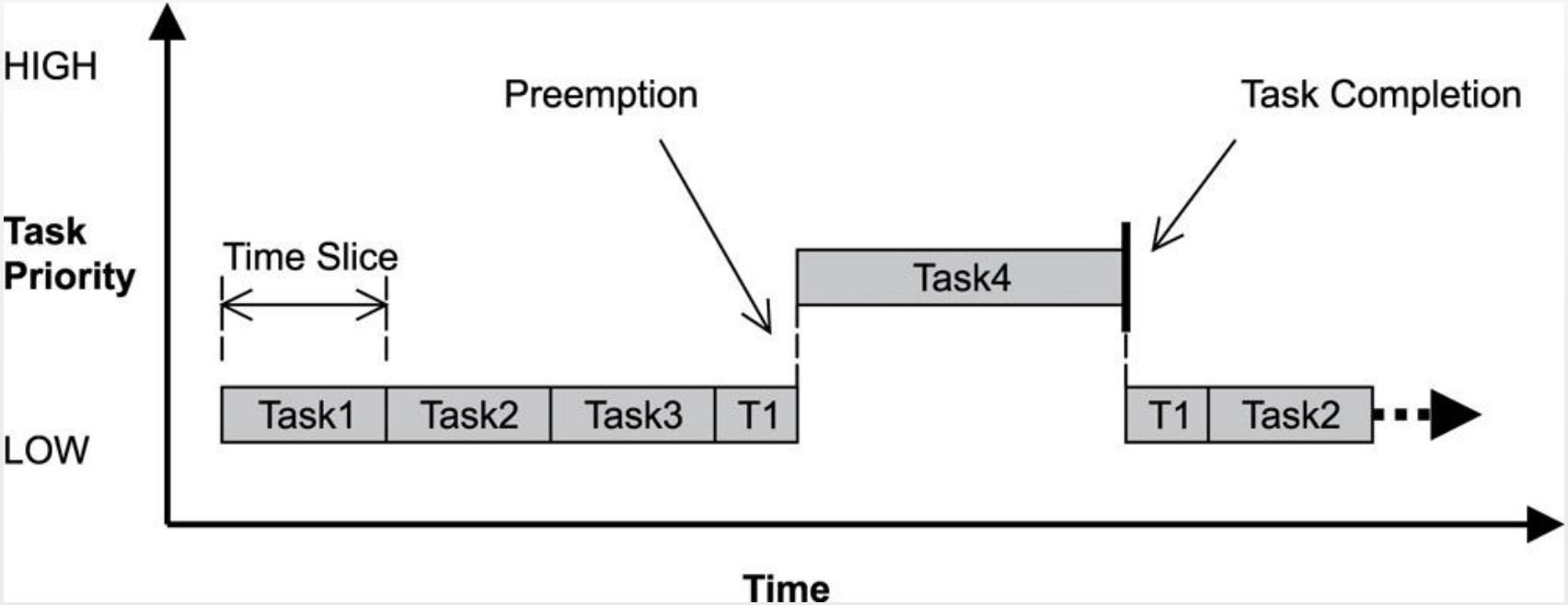
- **Típicamente, el scheduler se ejecuta a intervalos regulares**
  - Determinados por un timer o reloj de tiempo real (RTC)
- **Cuando lo hace, vuelve a elaborar la lista de tareas listas (esa que lee el dispatcher), dándoles un orden que, idealmente, depende del tiempo remanente hasta el deadline de cada una**
- **Existen varios tipos de algoritmos para la elaboración de este orden**
  - Esos son los **algoritmos de scheduling** que mencionamos antes

Scheduling preventivo, basado en prioridades

- **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
- **Se depende de que las tareas de alta prioridad devuelvan el control (simil scheduling cooperativo)**
- **Problema: por más que lo hagan, puede pasar mucho tiempo hasta que se ejecutan los de baja prioridad**
  - …si es que siempre hay una de mayor prioridad que está lista
  - A esa situación se le llama ***starvation***

Round-robin (en ronda)

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



Q.Li y C.Yao; *Real-Time Concepts for Embedded Systems*

Mejores algoritmos

- **Rate-monotonic scheduling**
  - Es un equema basado en prioridades (fijas), en donde la prioridad de una tarea es inversamente proporcional a su deadline relativo

Sincronización y comunicación entre procesos

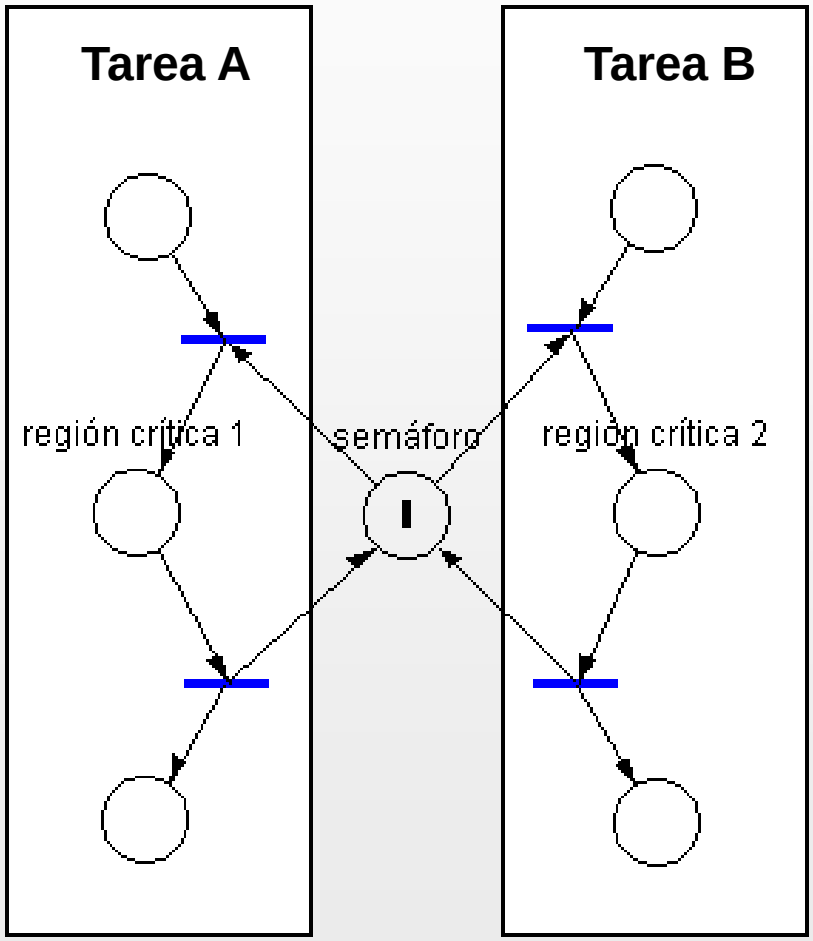
- **Imaginemos las siguientes tareas:**
  - Una **produce** ciertos datos
    - Son ingresados por un usuario, o los resultados de un cálculo, o lo que sea)
  - Otra **consume** esos datos
    - Los imprime, los usa para otros cálculos, o lo que sea
    - Mientras no reciba datos, esta tarea queda suspendida
- **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 allá
    - Pero cuidado, que un proceso no interprete un dato a medio escribir, como si estuviera escrito del todo
      - Es decir que se necesita algo como un *handshaking.* Por ejemplo, una señal de sincronización
- **Colas (queues) de mensajes**
  - Son colecciones ordenadas de (estructuras de) datos
  - Se puede ingresar un elemento al final, o sacar uno del principio
    - O sea que es una estructura *first-in, first-out* (FIFO)
  - Es una estructura de nivel más alto que la memoria compartida, que incluye sincronización

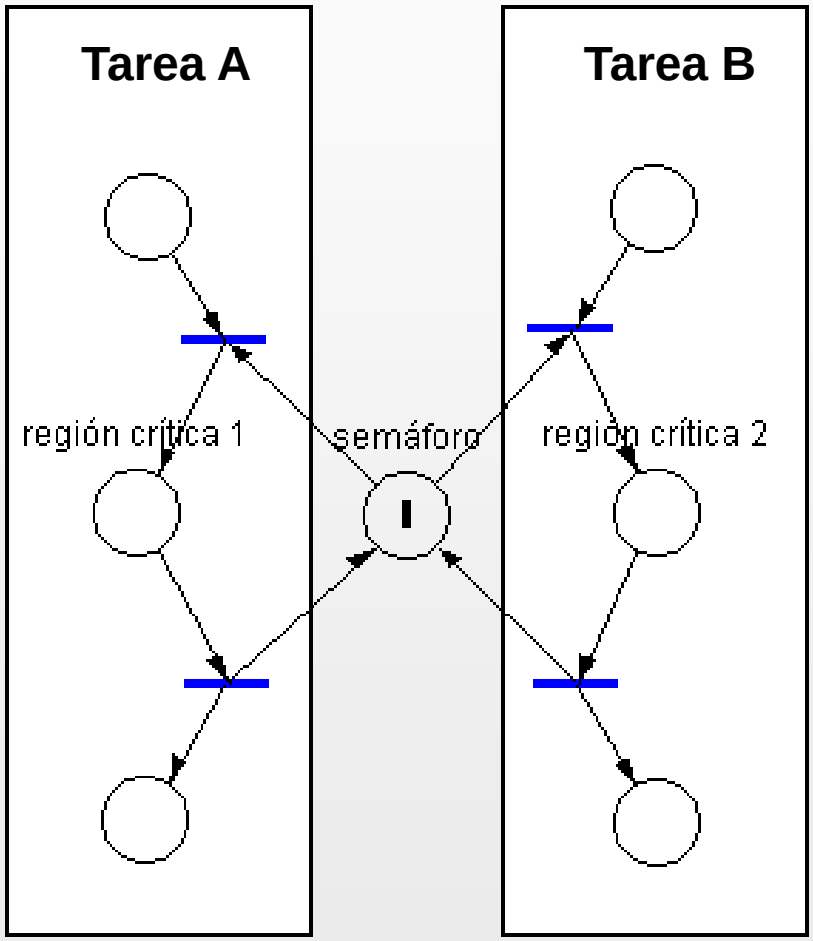
Sincronización mediante semáforos

- **Vean esta red de Petri**
- **Si no tuviéramos la posición "semáforo" ahí, las otras posiciones 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



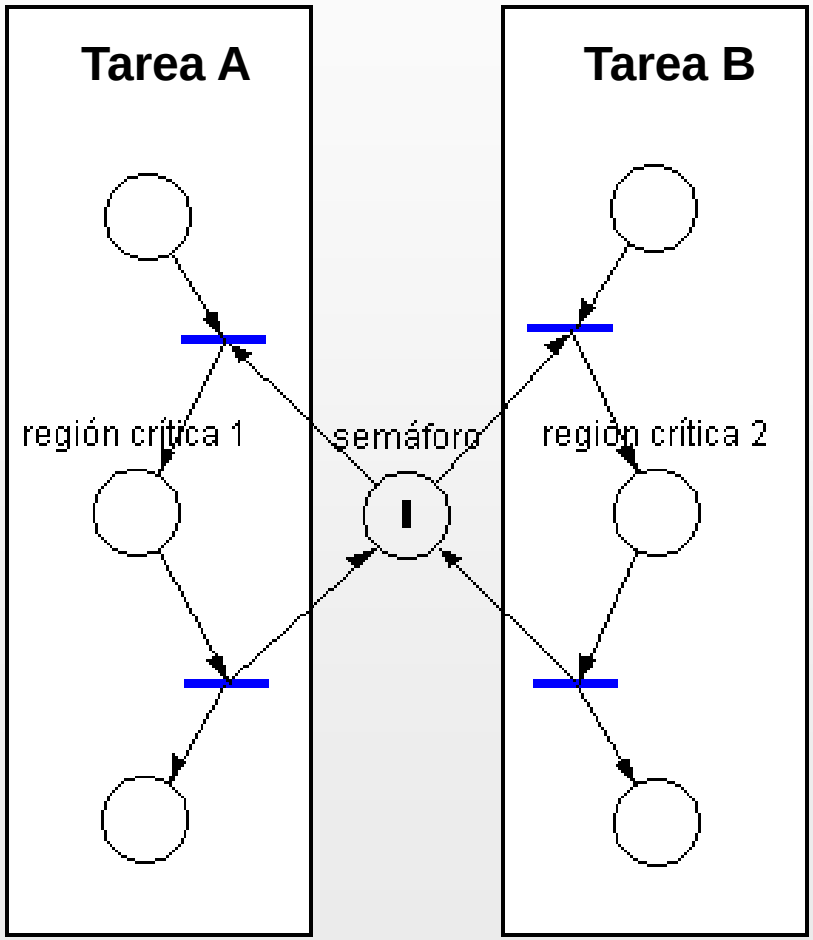
Sincronización mediante semáforos

- **Esta aplicación de los semáforos (o sea, evitar que dos procesos estén en sus regiones críticas) 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
}
```

Pseudo-código

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
  }
}
```

Problemas de la sincronización que hay que evitar

- **Deadlocks** (abrazo mortal)
- **Starvation** (problemas de los procesos de baja prioridad para ejecutarse)
- **Inversión de la prioridad** (un proceso de prioridad alta espera uno de menor prioridad)

Variantes de los semáforos clásicos

- **Semáforos binarios**
- **Mutex** (los de exclusión mutua)

Conclusiones