Carrera de Especialización en Sistemas Embebidos

Sistemas Operativos en Tiempo Real

Clase 1: Introducción a los RTOS

ACSE
Asociación Civil para la Investigación, Promoción y Desarrollo de los Sistemas Electrónicos Embebidos
FACULTAD DE INGENIERIA
Universidad de Buenos Aires
CIM
CADIEEL

SE: 2 modelos de programación

- Bare-metal:
  - El MCU no utiliza recursos de un sistema operativo.
  - Se utilizan en general tecnicas Foreground-Background y Scan loops.
  - El comportamiento es cooperativo.
  - La tarea del programador es más ardua.

- Sistema operativo:
  - Se utiliza un sistema que gestiona la ejecución de las tareas del SE.
  - Ofrece herramientas que resuelven problemáticas comunes en el desarrollo de un SE (timers, delays, colas, etc)

¿ Qué es un OS?

- Es un conjunto de programas que ayuda al programador de aplicaciones a gestionar los recursos de hardware disponibles, entre ellos el tiempo del procesador y la memoria.
- La gestión del tiempo del procesador permite al programador de aplicaciones escribir múltiples subprogramas como si cada uno fuera el único que utiliza la CPU.
- Una parte del OS se encarga de asignar tiempo de ejecución a todos los programas que tiene cargados en base a un juego de reglas conocido de antemano. A estos subprogramas se los llama tareas.
- Con esto se logra la ilusión de que múltiples programas se ejecutan simultáneamente, aunque en realidad sólo pueden hacerlo de a uno a la vez (en sistemas con un sólo núcleo, como es el caso general de los sistemas embebidos).

¿ Cómo se administra el tiempo del CPU?

- El encargado de esta gestión es un componente del OS llamado scheduler o planificador de tareas. Su función es determinar qué tarea debe estar en ejecución a cada momento.
  - Los OS utilizan uno o más algoritmos de planificación para determinar cual tarea ejecutar en cierto momento.

- Ante la ocurrencia de ciertos eventos revisa si la tarea en ejecución debe reemplazarse por alguna otra tarea. A este reemplazo se le llama cambio de contexto de ejecución.

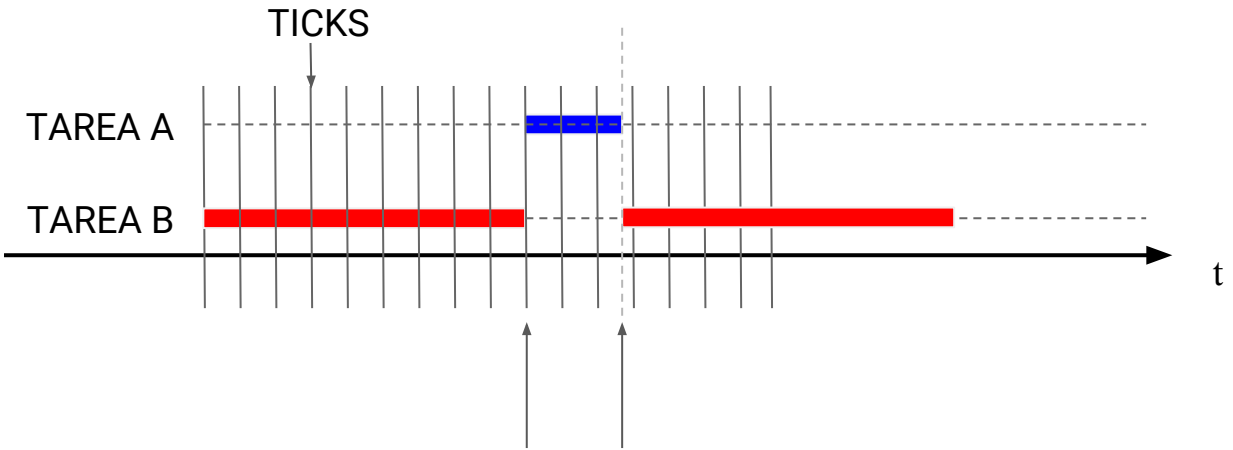
- El OS administra el tiempo de CPU utilizando una base de tiempo constante.

Contexto de Ejecución

- Se llama contexto de ejecución el mínimo conjunto de recursos utilizados por una tarea con los cuales se permita reanudar su ejecución:
  - IP (instruction pointer)
  - SP (stack pointer)
  - Registros del CPU
  - Contenido de la pila en uso
- Cuando el planificador determina que debe cambiarse el contexto de ejecución, invoca a otro componente del OS llamado dispatcher para que guarde el contexto completo de la tarea actual y lo reemplace por el de la tarea entrante.
- Por esta razón cada tarea posee su propio stack de memoria. Esto limita la cantidad de tareas simultáneas del sistema (pueden sin embargo eliminarse y crearse nuevas tareas en tiempo de ejecución).

¿ Como se realiza el cambio de contexto?

- Los cambios de contexto se realizan de forma transparente para la tarea, no agregan trabajo al programador. Cuando la tarea retoma su ejecución no muestra ningún síntoma de haberla pausado alguna vez.



En estos momentos el planificador decide (con algún criterio) que se debe producir el cambio.

Notar que los cambios pueden ocurrir alineados con el tick del sistema, o sin estar alineados.

¿ Como se realiza el cambio de contexto?

Estado del CPU

Estado: Running

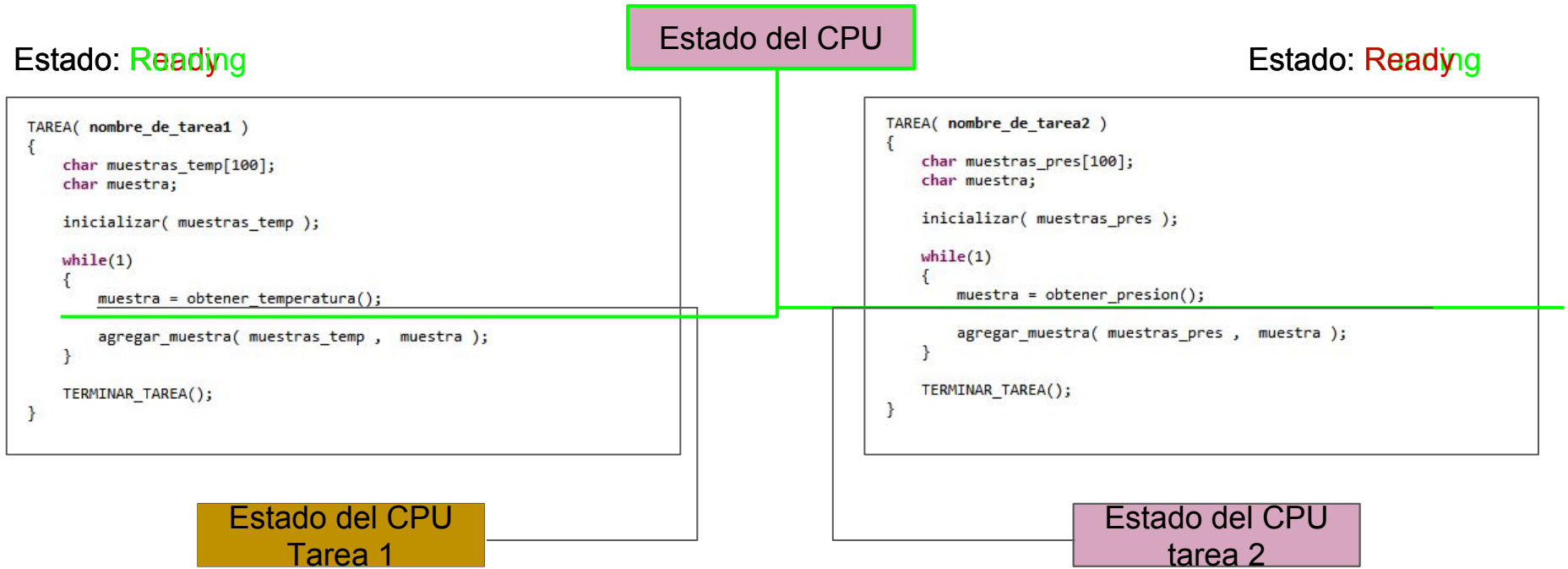
```
TAREA( nombre_de_tarea1 )
{
    char muestras_temp[100];
    char muestra;

    inicializar( muestras_temp );

    while(1)
    {
        muestra = obtener_temperatura();

        agregar_muestra( muestras_temp ,  muestra );
    }

    TERMINAR_TAREA();
}
```

Estado: Ready

```
TAREA( nombre_de_tarea2 )
{
    char muestras_pres[100];
    char muestra;

    inicializar( muestras_pres );

    while(1)
    {
        muestra = obtener_presion();

        agregar_muestra( muestras_pres ,  muestra );
    }

    TERMINAR_TAREA();
}
```

Estado del CPU tarea 2

¿ Como se realiza el cambio de contexto?



Modos de operación de un SO

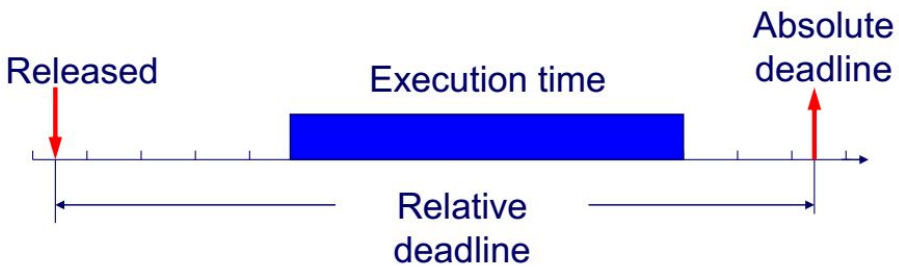
- Cooperativos:
  - Una tarea no se detiene si el planificador de tareas decide que hay otra que debiera ejecutarse.
  - Las tareas deben cooperar para que todas posean el tiempo de CPU para que permitan su ejecución sin retraso.
  - Los cambios de contexto solo ocurren cuando la tarea en ejecución relega voluntariamente el uso del CPU.
- Apropiativos:
  - Si el planificador de tareas decide que debe ejecutarse otra tarea distinta a la actual, se genera una pausa en la ejecución de la tarea actual, y se le brinda el uso de CPU a la nueva (la nueva tarea, se “apropia” del CPU). En ese momento ocurre el cambio de contexto.
    La tareas no tienen el control sobre este cambio.

¿Qué es un RTOS?

- Hace lo mismo que un OS común pero además me da herramientas para que los programas de aplicación puedan cumplir compromisos temporales definidos por el programador.
- Un RTOS es un sistema operativo de tiempo real. Se utilizan cuando tiempo de respuesta ante ciertos eventos es un parámetro crítico.
  - Respuestas demasiado tempranas o muy tardías podrían ser indeseables.
- Un RTOS se emplea cuando hay que administrar varias tareas simultáneas con plazos de tiempo estrictos.

¿Cómo se define un sistema de tiempo real?

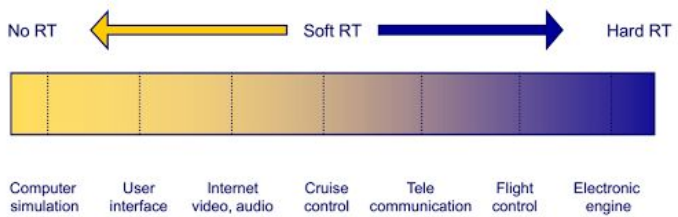
- Un STR está definido por:
  - Los eventos externos que debe atender.
  - La respuesta que debe producir ante estos eventos.
  - Los requerimientos de temporización de esas respuestas.
- Cada ciclo de operación de una tarea que tiene estrictos requerimiento temporales posee se puede modelar temporalmente por su ciclo de vida.



Atributo	Descripción
Period	Intervalo de tiempo que indica cuán seguido se deberá iniciar una tarea (para tareas periódicas)
Release Time	Momento en el tiempo en el cual la tarea se vuelve disponible para ejecución
Deadline	Momento en el tiempo en el cual la tarea debería estar completada.
Execution time	Tiempo que tarda la tarea en ejecutarse (peor caso)

Tipos de RTOS: Según tiempo de respuesta

- Duros (Hard Real Time):
  - Restricciones de tiempo estrictas
  - Consecuencias catastróficas si se pierden metas.
  - Ejemplos:
    - Controles de lazo cerrado (controles de motores, aviónica):
      - Los retrasos afectan la estabilidad
- Blandos (Soft Real Time):
  - Restricciones de tiempo menos rigurosas
  - No se considera crítico que no se cumplan los plazos.
  - Ejemplos:
    - Interfaz de teclado al usuario
    - Juegos
    - Servidor web



¿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.

- 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.

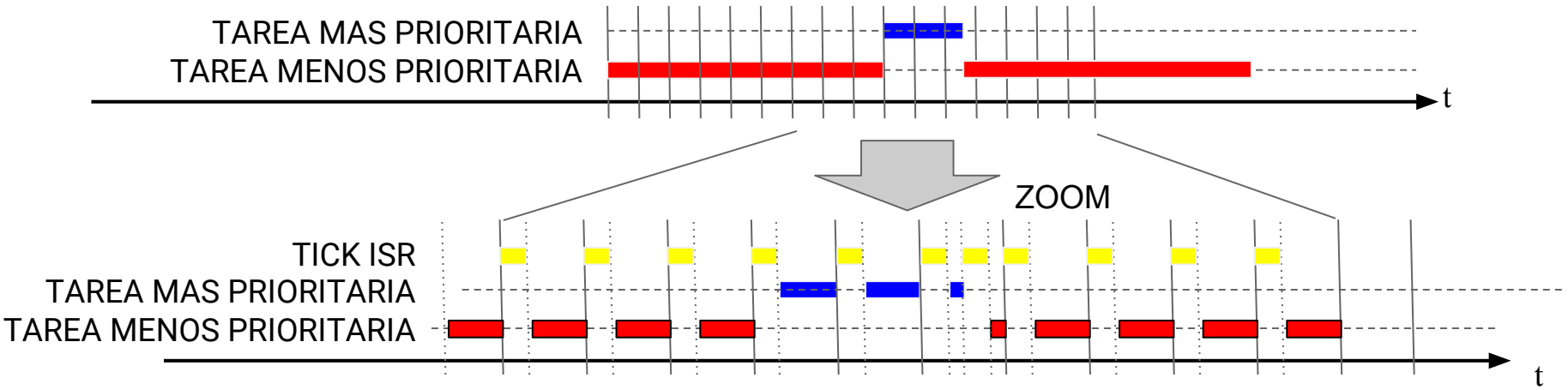
- Para contar con un framework de trabajo con muchas herramientas ya probadas y funcionales.

¿Por qué usar un RTOS?

- Multitarea
  - Simplifica sobremanera la programación de sistemas con varias tareas.
- Escalabilidad
  - Al tener ejecución concurrente de tareas se pueden agregar las que haga 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.

Letra chica 1

- Latencias temporales
  - Se gasta tiempo del CPU en determinar en todo momento qué tarea debe estar corriendo. Si el sistema debe manejar eventos que ocurren demasiado rápido tal vez no haya tiempo para esto.
  - Se gasta tiempo del CPU cada vez que debe cambiarse la tarea en ejecución.



Letra chica 2

- Memoria
  - 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.

- Para que las partes del sistema que requieren temporización estricta, debe hacerse un análisis de tiempos, eventos y respuestas muy cuidadoso. Una aplicación mal diseñada puede fallar en la atención de eventos aún cuando se use un RTOS.

Tipos de tareas.

- Tareas periódicas
  - Atienden eventos que ocurren constantemente y a una frecuencia determinada. P. ej, destellar un led, muestrear una señal, corregir un lazo de control, etc.
- Tareas aperiódicas
  - Atienden eventos que no se sabe cuándo van a ocurrir. 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.

¿Porque FreeRTOS?

- Es de código abierto
  - Licencia MIT
  - El código está ampliamente comentado, es muy sencillo verificar cómo hace su trabajo.
  - Escrito en C, salvando algunas partes específicas de cada plataforma.
- Facil de Usar
  - Mucha documentación disponible.
  - Nutrida comunidad de Usuarios
- Opciones
  - Hay opcion comercial (OpenRTOS)
  - Hay una opcion certificada (SafeRTOS)

https://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.shtml

https://www.freertos.org/a00114.html#commercial

Algunas características

- Es un mini kernel de tiempo real que puede trabajar en modos cooperativo, preemptive o mixto.
- Es configurable por el usuario:
  - FreeRTOSConfig.h permite (a través de una serie de macros)
    - Habilitar o deshabilitar características para que la compilación resulte en una utilización de memoria de programa óptima.
    - Escalar los recursos de memoria para cada tarea.
- Gran cantidad de herramientas que le hacen al programador la vida más sencilla.
- Las los objetos que gestiona el OS pueden crearse:
  - Dinámicamente: Usa malloc para su creación, y se realiza en tiempo de ejecución.
  - Estáticamente: Los espacios de memoria se asignan en tiempo de compilación.

¿Cómo es una tarea en FreeRTOS?

- El usuario las define a través de simples funciones de C, con el siguiente prototipo: `void vTareaEjemplo (void *parametros)`
- Las tareas en general poseen una sección de inicialización, y un bucle infinito.

```
void vTareaEjemplo( void *parametros )
{
    /* inicialización */
    for( ;; )
    {
        -- código de la aplicación --
    }
    /* Si es necesario finalizar con una tarea,
       deberá quitarse del planificador */
    vTaskDelete( NULL );
}
```

- Las tareas tienen una prioridad de ejecución. 0 es la menor prioridad.
  - Se recomienda usar referencias a tskIDLE_PRIORITY (+1, +2, etc)
  - No hay límites a la cantidad de prioridades del sistema, pero se gasta RAM por cada una. Usar con cuidado.
- Se pueden crear múltiples instancias de una misma función de tarea
  - Estas instancias pueden recibir un parámetro que las caracterice.

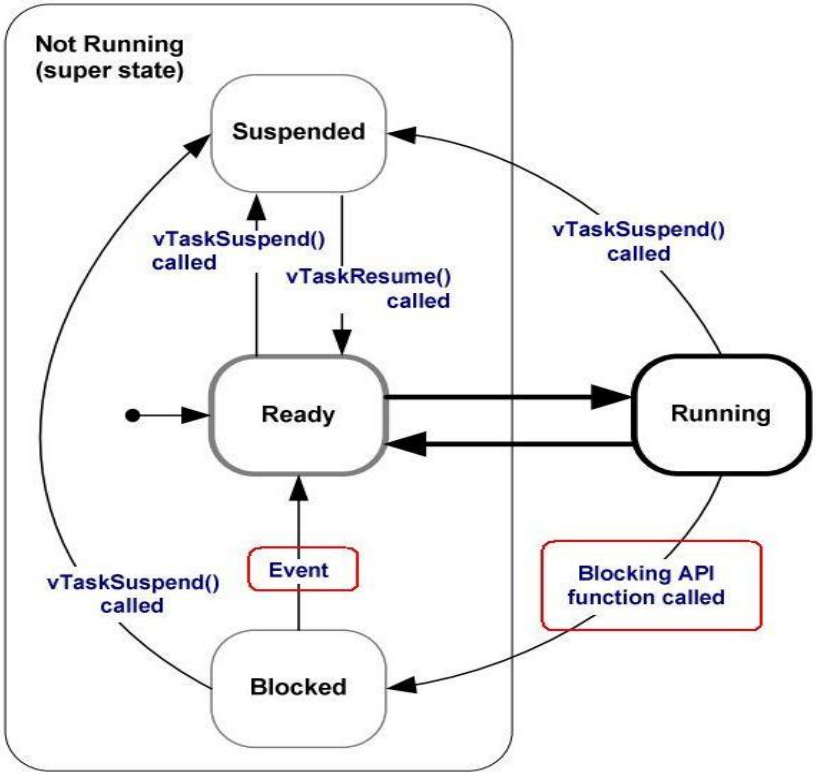
¿Cómo es una tarea en FreeRTOS?

- Al ser funciones de C, se aplican las mismas reglas de visibilidad de variables.
  - Pueden usar variables globales.

- Si una tarea se diseña para que nunca sea destruida, sus variables locales tendrán datos válidos. Pero en cuanto es destruida (por sí misma o por otra tarea) esos datos locales dejarán de tener validez.

```
int global_variable;

void taskFunction(void *pvParameters)
{
	int local_variable;

	/* sentencias de inicialización */

	while ( /* condición de lazo */ )
	{
		/* sentencias del lazo */
	}

	vTaskDelete(0); /* nunca return; !! */
}
```

Estados de las tareas en FreeRTOS

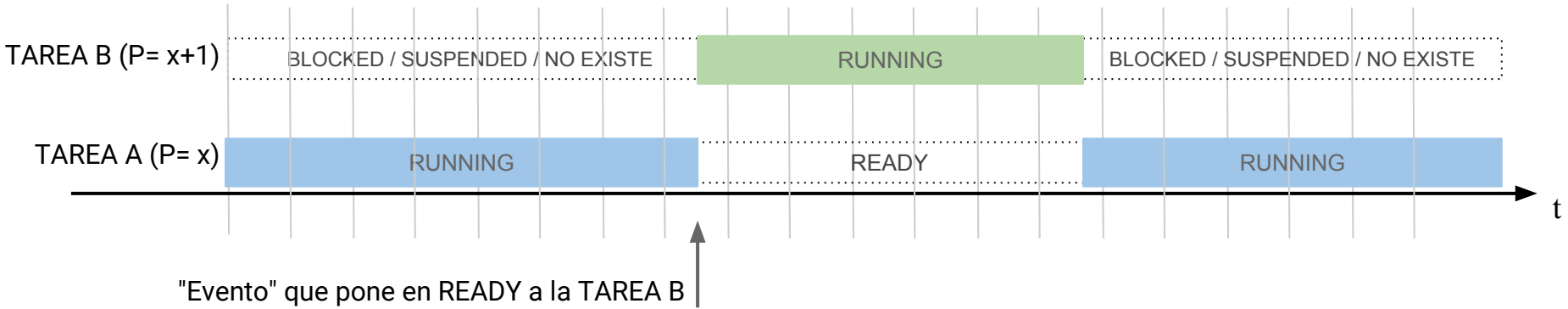


- Blocked: La tarea está esperando un evento (temporal o asincrónico).
- Ready: La tarea está lista para ser ejecutada.
- Running: La tarea está ejecutándose
- Suspended: No están dentro del planificador del OS. Es como si no se hubiera creado, pero con la diferencia que la memoria que consume la tarea queda reservada.

Algoritmo de Planificación de Tareas

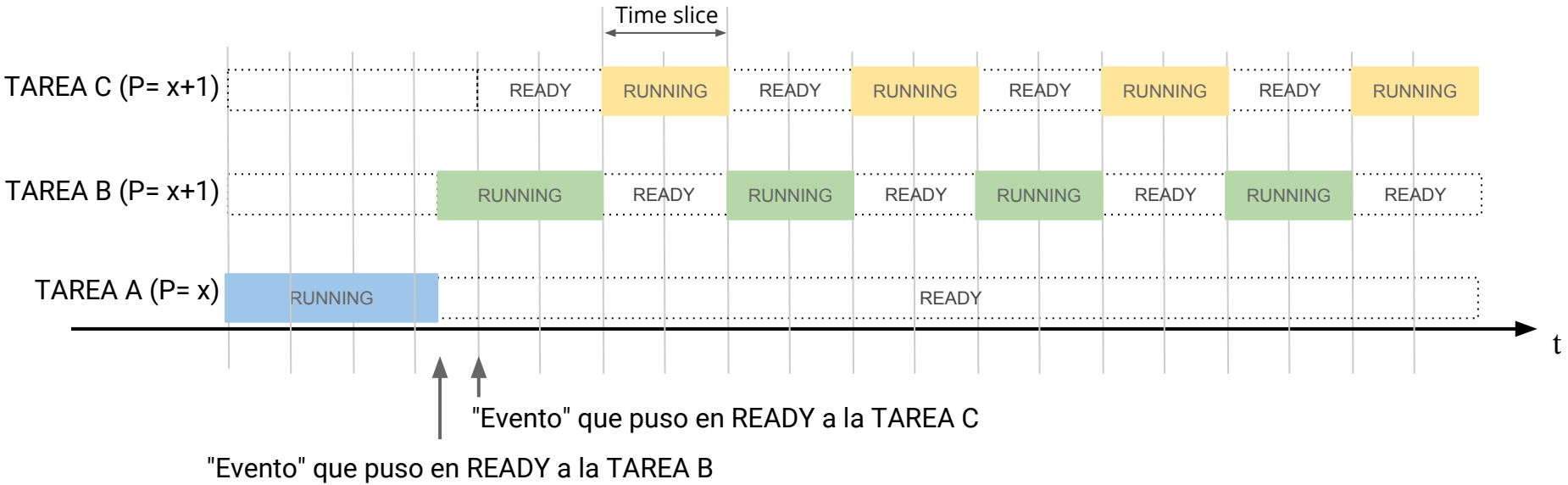
- Fixed priority preemptive scheduling
  - Fixed priority significa que el kernel NO MODIFICA las prioridades de las tareas (salvo en un caso particular que se verá más adelante).
  - Prehemptive: Significa que es apropiativo (se puede desactivar)

- Este algoritmo permite que, de un conjunto de tareas listas para ser ejecutadas (READY) SIEMPRE ejecute la tarea de mayor prioridad.



Algoritmo de Planificación de Tareas

- Round Robin Scheduling
  - Para tareas de la misma prioridad, listas para ser ejecutadas, el planificador divide el tiempo de CPU y lo reparte entre ellas.
  - Esta opción puede desactivarse.



Resolución temporal del sistema (tick)

- La resolución temporal del sistema está definida en la macro TICK_RATE.

- El valor de TICK_RATE debe asignarse para garantizar que todos los plazos temporales de cubran a la perfección.

- Un valor muy alto de TICK_RATE hace que el scheduler trabaje más seguido, ocupando más tiempo del CPU.
  - Este tiempo debiera ser despreciable respecto del que consume la aplicación.

API: **Arranque del OS**

- `void vTaskStartScheduler( void );`
  - Arranca el planificador del OS.
  - Crea la tarea IDLE.
  - Corre por primera vez el planificador, para evaluar cual tarea hay que ejecutar.

```
void tareaA(void *pvParameters)
{
      /* código de la tarea */
}

int main(void)
{
      prvSetupHardware();
      xTaskCreate(tareaA, (signed char *) "descripcion",
            configMINIMAL_STACK_SIZE, NULL,
            (tskIDLE_PRIORITY + 1UL),
            (xTaskHandle *) NULL);

      vTaskStartScheduler();

      return 1;
}
```

API: **Creación de tareas**

- BaseType_t **xTaskCreate**(...):
  - Crea e inicializa los espacios de memoria para la ejecución de una tarea. La tarea inicia en estado READY.
- Se la puede llamar en cualquier momento.
- Parámetros:
  - pvTaskCode: referencia a la función de C que describe el comportamiento.
  - pcName: Nombre descriptivo.
  - usStackDepth: Tamaño del stack en words. (valor mínimo configMINIMAL_STACK_SIZE)
  - pvParameters: Parámetro opcional a la tarea (permite varias instancias de la misma tarea, con comportamientos distintos)
  - uxPriority : prioridad de la tarea. Debe ser mayor a tskIDLE_PRIORITY
  - pxCreatedTask: Referencia a un handler, si es requerido.

API: **Eliminación de tareas**

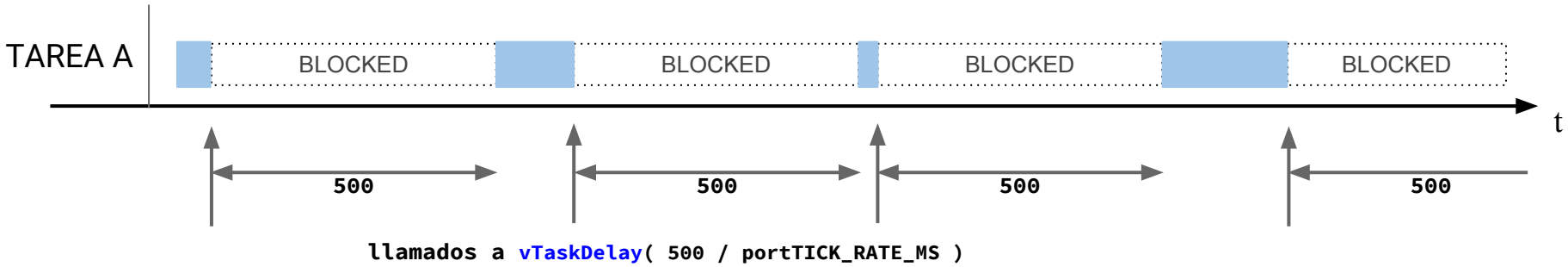
- `void` **`vTaskDelete`**`( TaskHandle_t xTask ):`
  - Quita a la tarea de la gestión del OS, liberando espacios de memoria asociados.
  - Pasándole como parámetro NULL elimina la tarea quien llamó a la función

```
void vOtherFunction( void )
{
     TaskHandle_t xHandle = NULL;

     // Se crea una tarea y almacena su handler.
     xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

     // Si la tarea anterior fue creada con éxito, se destruye.
     if( xHandle != NULL )
     {
         vTaskDelete( xHandle );
     }
}
```

API: **Delay**

- `void` **`vTaskDelay`**`( const TickType_t` **`xTicksToDelay`** `)`
  - Produce una demora en la tarea que la llama. Cede el control del CPU mientras este tiempo no expira.

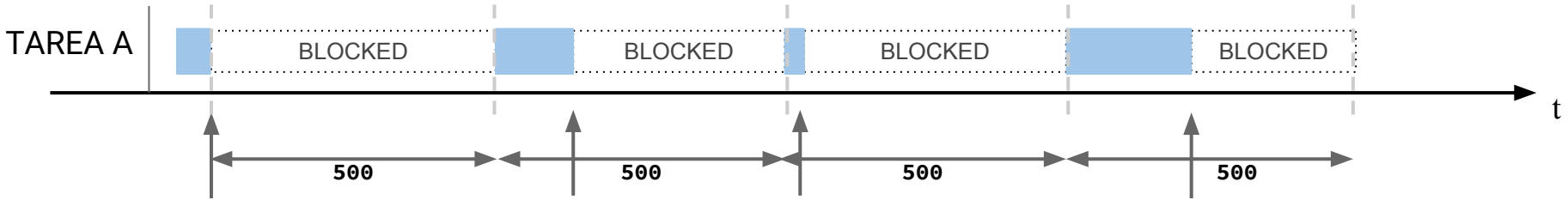
```
void TareaA( void* params )
{
	while(1)
	{
		/*
		CODIGO DE PROCESO "PERIODICO" QUE TARDA EN EJECUTARSE (SIN INTERRUPCIONES ENTRE 1 Y 200 TICKS)
		*/
		vTaskDelay( 500 / portTICK_RATE_MS );
	}
}
```



API: **vTaskDelayUntil**

- `void` **`vTaskDelayUntil`**`( TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement );`
  - Asegura un delay entre cada uno de los llamados a esta función. Cede el control del CPU mientras este tiempo no expira.

```
void TareaA( void* params )
{
        TickType_t tiempo_inicio_ciclo;
        Tiempo_inicio_ciclo = xTaskGetTickCount();
        while(1)
        {
                /*
                CODIGO DE PROCESO PERIODICO QUE TARDA EN EJECUTARSE (SIN INTERRUPCIONES ENTRE 1 Y 200 TICKS)
                */
                vTaskDelayUntil( &tiempo_inicio_ciclo, 500 / portTICK_RATE_MS );

        }
}
```



llamados a `vTaskDelayUntil( &tiempo_inicio_ciclo, 500 / portTICK_RATE_MS );`

Bibliografia

- https://www.freertos.org
- https://docs.aws.amazon.com/freertos-kernel/latest/dg/about.html
- Introducción a los Sistemas operativos de Tiempo Real, Alejandro Celery - 2014
- Introducción a los Sistemas Operativos de Tiempo Real, Pablo Ridolfi, UTN, 2015.
- Introducción a Planificación de Tareas, CAPSE, Franco Bucafusco, 2017
- FreeRTOS - Temporización, Cusos INET, Franco Bucafusco, 2017

Bibliografia

- https://www.freertos.org
- https://docs.aws.amazon.com/freertos-kernel/latest/dg/about.html
- Introducción a los Sistemas operativos de Tiempo Real, Alejandro Celery - 2014
- Introducción a los Sistemas Operativos de Tiempo Real, Pablo Ridolfi, UTN, 2015.
- Introducción a Planificación de Tareas, CAPSE, Franco Bucafusco, 2017
- Introducción a Sistemas cooperativos, CAPSE, Franco Bucafusco, 2017
- FreeRTOS - Temporización, Cusos INET, Franco Bucafusco, 2017