

## Carrera de Especialización en Sistemas Embebidos

# Sistemas Operativos en Tiempo Real

Clase 1: Introducción a los RTOS









## SE: Dos modelos de programación

#### Bare-metal:

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

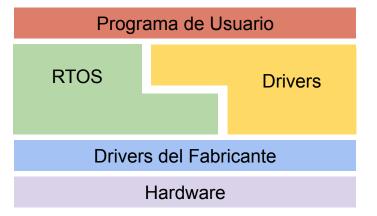
#### Sistema operativo:

- 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.



 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.



## Tipos de tareas.

#### Tareas periódicas

- Ejecutan acciones constantemente y a una frecuencia determinada.
  - Ej: Destellar un led, muestrear una señal, corregir un lazo de control, etc.

#### Tareas "disparo único" (asincronico, esporadicas, reactivas)

- Atienden eventos que no se sabe cuándo van a ocurrir. Estas tareas están inactivas hasta que ocurre el evento de interés.
  - Ej: una parada de emergencia.

#### Tareas de procesamiento continuo

- Son tareas que trabajan en régimen permanente (sin una periodicidad definida)
  - Ej: tareas de mantenimiento, en general de baja prioridad.



## Tipos de tareas

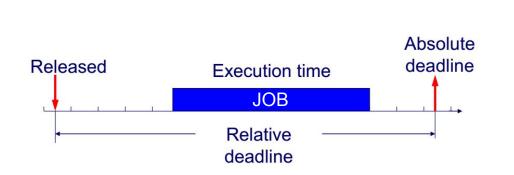
```
void tarea_periodica()
{
    /* incializacion */
    while(1)
    {
        job();
        delay( X );
    }
}
```

```
void tarea_disparo_unico()
{
    /* incializacion */
    while(1)
    {
        esperar_señal();
        job();
    }
}
```

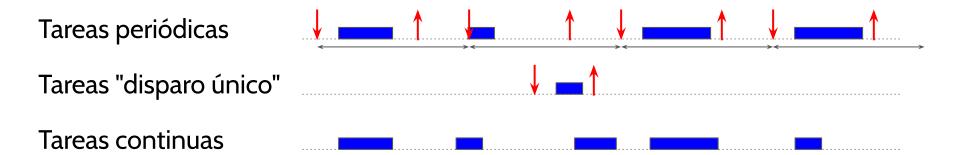
```
void tarea_continua()
{
     /* incializacion */
     while(1)
     {
          job1();
          job2();
          job3();
     }
}
```



#### Ciclo de vida de las tareas



Atributo	Descripción	
Release Time	Momento en el tiempo en el cual la tarea se vuelve disponible para ejecución	
Relative Deadline	<u>Tiempo máximo aceptable</u> para que la tarea sea completada.	
Execution time	Tiempo que tarda la tarea en ejecutarse (peor caso)	





## Scheduler, dispatcher y base de tiempo

- Scheduler (planificador): Administra el tiempo de CPU para cada tarea.
  - o Determina qué tarea debe estar en ejecución a cada momento.
  - o Utiliza uno o más algoritmos para decidir cual tarea ejecutar.

Dispatcher: Provee el control del CPU a una cierta tarea.

 <u>Base de tiempo (tick)</u>: Provee la mínima unidad de medida de tiempo para las operaciones temporizadas del OS (delays y timeouts).



## Modos de operación de un SO

#### Cooperativos:

- Una tarea no se detiene aun cuando el planificador de tareas decida que hay otra que debiera ejecutarse.
- Las tareas deben <u>cooperar</u> para que todas posean el tiempo de CPU para que permitan su ejecución.

#### 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 <u>cambio de contexto</u>.



## OS apropiativo

 Permite al programador de aplicaciones escribir múltiples tareas como si cada una fuera la única que utiliza la CPU.

 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 ).



## 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
  - o Contenido de la pila en uso
- Cuando el planificador determina que debe cambiarse el contexto de ejecución, invoca al <u>dispatcher</u> para que guarde el contexto completo de la tarea actual y lo reemplace por el de la tarea entrante.
- Cada tarea <u>posee su propio stack de memoria</u>.



#### ¿ Como se realiza el cambio de contexto?

#### 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 del CPU

```
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 de Tarea 2



#### ¿ Como se realiza el cambio de contexto?

#### Estado: Reading

```
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 del CPU de Tarea 1

#### Estado del CPU

```
Estado: Reading
```

```
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 de Tarea 2

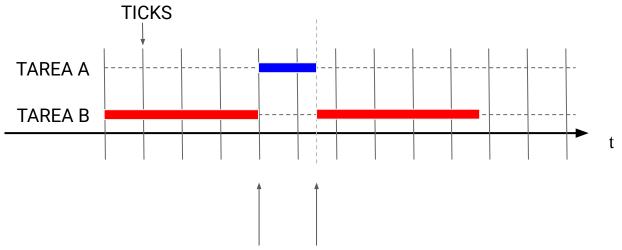


#### ¿Cuándo se realiza el cambio de contexto?

- Los cambios de contexto se realizan de forma transparente para la tarea.
   Cuando la tarea retoma su ejecución no muestra ningún síntoma de haberse pausado alguna vez.
- Ocurren cuando el <u>planificador lo determina</u>.
- Ejemplos:
  - Luego que se cumpla un cierto tiempo
  - Que el programador llame a una API que, indirectamente, invoque al scheduler.
    - Estos llamados se conocen como RESCHEDULING POINT



## ¿Cuándo se realiza el cambio de contexto?



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.



## ¿Qué es un RTOS?

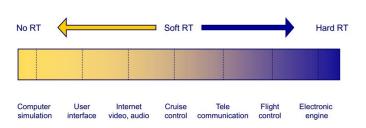
 Es un OS que está diseñado para que los programas de aplicación puedan cumplir compromisos temporales definidos por el programador.

- Se utilizan cuando <u>tiempo de respuesta</u> ante ciertos eventos es un parámetro crítico.
  - Respuestas demasiado tempranas o muy tardías podrían ser indeseables.
- Un RTOS se emplea cuando hay sistema funciona correctamente no solo con los <u>resultados lógicos</u>, sino también el <u>tiempo</u> en que se producen.



## Tipos de RTOS: Según tiempo de respuesta

- Duros (Hard Real Time):
  - Restricciones de tiempo estrictas
  - o 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

o 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 reusabilidad del código

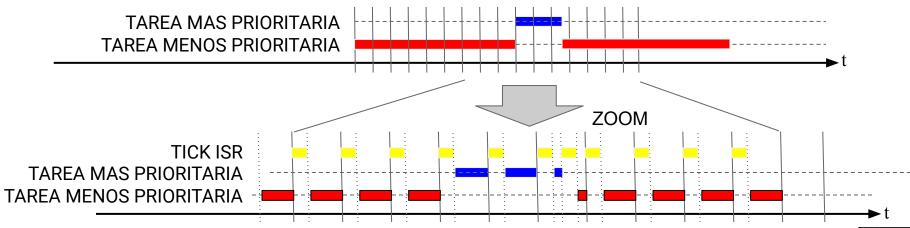
 Si las tareas se diseñan bien (con pocas o ninguna dependencia) es más fácil incorporarlas a otras aplicaciones.



#### 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
  - o 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.



## ¿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

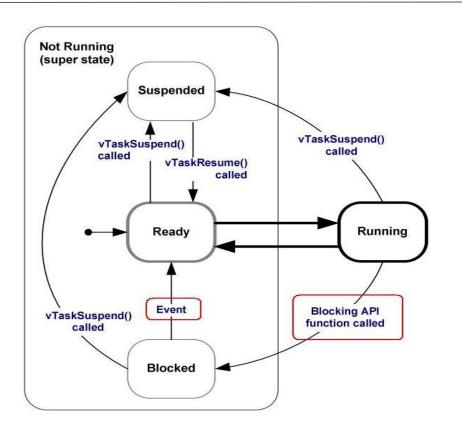
#### Opciones

- Hay una opción comercial (OpenRTOS)
- Hay una opción certificada (SafeRTOS)



#### Estados de las tareas en FreeRTOS





- <u>Blocked</u>: 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.



#### Round Robin Scheduling

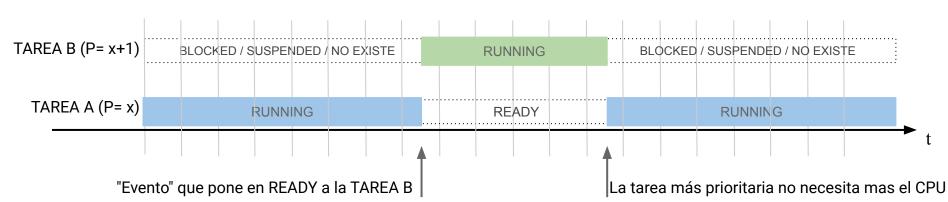
- Para tareas de la misma prioridad, listas para ser ejecutadas, el planificador divide el tiempo de CPU y lo reparte entre ellas.
- Esta opción puede desactivarse.



"Evento" que puso en READY a la TAREA B



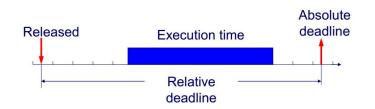
- Fixed priority preemptive scheduling
  - Fixed priority significa que el kernel NO MODIFICA las prioridades de las tareas (salvo en un caso particular que se verá más adelante).
  - o Preemptive: Significa que es apropiativo (se puede desactivar)
- Este algoritmo permite que, de un conjunto de tareas listas para ser ejecutadas (READY) SIEMPRE ejecute la tarea de mayor prioridad.



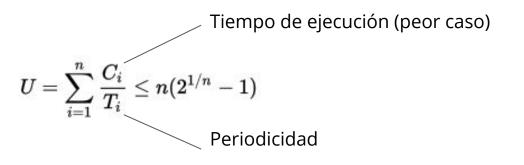




- Fixed priority preemptive scheduling
  - ¿ Cómo se asignan las prioridades ?



- Algoritmo "Rate Monotonic Scheduling" (RMS)
  - o Se asignan prioridades más altas a tareas más frecuentes.
- Utilización del CPU (modelo válido solo para tareas periódicas)



De no cumplise la igualdad, las tareas no serian planificables (no cumplirian el deadline)



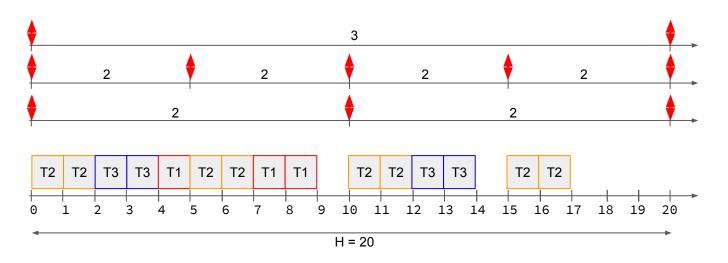


Ejemplo

Tarea	Ci	Ti
T1	3	20
T2	2	5
Т3	2	10

H = MCM(20, 5, 10)

Usando RMS ordenamos las prioridades: P(T2) > P(T3) > P(T1)



- Algoritmo "Deadline Monotonic Scheduling" (DMS)
  - Se asignan prioridades más altas a tareas deadline relativos más cortos

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \le n(2^{1/n} - 1)$$

- Dynamic priority preemptive scheduling
  - ¿ Cómo se asignan las prioridades ? No se asignan.
     Se definen deadlines relativos y el scheduler cambia las prioridades basado en esos valores.
- Algoritmo "Earliest Deadline First" (EDF)
  - El planificador le da mayor prioridad a la tarea que se encuentra más cerca del deadline

$$U=\sum_{i=1}^n rac{C_i}{T_i} \leq 1,$$



## 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.

#### 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:
  - o <u>pvTaskCode</u>: referencia a la función de C que describe el comportamiento.
  - <u>pcName</u>: Nombre descriptivo.
  - <u>usStackDepth</u>: Tamaño del stack en words. (valor mínimo

configMINIMAL\_STACK\_SIZE)

- <u>pvParameters</u>: 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
- <u>pxCreatedTask:</u> <u>Handle</u> de la tarea, 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.
  - o 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 );
    }
}
```



## Bibliografia

- https://www.freertos.org
- FreeRTOS Kernel Documentation
- 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, Cursos INET, Franco Bucafusco, 2017
- Rate-monotonic scheduling, Wikipedia Consultado 19-5-2
- Earliest Deadline First, Wikipedia Consultado 19-5-2

#### Licencia



"Introducción a los RTOS"

Por Mg. Ing. Franco Bucafusco, se distribuye bajo una <u>licencia de Creative Commons</u>

<u>Reconocimiento-Compartirlgual 4.0 Internacional</u>