

CIRCUITOS DIGITALES Y MICROCONTROLADORES 2022

Facultad de Ingeniería
UNLP

Planificación y Ejecución de Tareas en
Sistemas Embebidos

Ing. José Juárez

Planificación de tareas en los Sistemas Embebidos

- La arquitectura de planificación más simple para un MCU es el Super-Loop:

```
/*-----*-  
Main.C  
-----  
Architecture of a simple Super Loop application  
-*-----*/  
#include "X.h"  
/*-----*/  
void main(void)  
{  
    X_Init(); // Preparar la tarea X y las condiciones iniciales  
    while(1) // Super Loop  
    {  
        X(); // Ejecutar la tarea X.  
    }  
}
```

Notar que X() es un nombre genérico de función que representa una tarea específica.

Por lo tanto, por lo tanto ejecutar la tarea significa ejecutar la función hasta completarla.

Planificación de tareas en los Sistemas Embebidos

- Planificación de varias tareas:

```
/*-----*_  
Main.C  
-----  
Architecture of a simple Super Loop application  
_*-----*/  
#include "X.h"  
#include "y.h"  
#include "z.h"  
/*-----*/  
void main(void)  
{  
    X_Init(); // Preparar la tarea X y las condiciones iniciales  
    Y_Init(); // Preparar la tarea Y y las condiciones iniciales  
    Z_Init(); // Preparar la tarea Z y las condiciones iniciales  
  
    while(1) // (Super Loop)  
    {  
        X(); // Ejecutar la tarea X  
        Y(); // Ejecutar la tarea Y  
        Z(); // Ejecutar la tarea Z  
    }  
}
```

*Ejecutivo cíclico
Round Robin*

*¿Cuál es la temporización
de cada tarea?
¿Cuál es el tiempo de respuesta
a determinados eventos
asociados a dichas tareas?*

Planificación de tareas en los Sistemas Embebidos

Agregamos interrupciones al super-loop

```
void main(void) {
    X_Init(); //
    Y_Init(); //
    Z_Init(); //
    sei();    // Globally enable interrupts

    while(1) {
        if (Flag_Z) {
            Z(); //if event Z occurred->process event Z
            Flag_Z = 0; }
        if (Flag_Y) {
            Y(); // if event Y occurred->process event Z
            Flag_Y = 0; }
        if (Flag_X) {
            X(); // if event X occurred->process event X
            Flag_X = 0; }
    }
}
```

Tareas en Background

```
/* ----- */
---ISR Event X
/* ----- */
ISR ( Event_X )
{
    Flag_X = 1; //event X has occurred
}

/* ----- */
---ISR Event Y
/* ----- */
ISR ( Event_Y )
{
    Flag_Y = 1; //event Y has occurred
}

/* ----- */
---ISR Event Z
/* ----- */
ISR ( Event_Z )
{
    Flag_Z = 1; //event Z has occurred
}

}
```

Tareas en Foreground

Cada interrupción corresponde a un **evento** asociado a una tarea específica (múltiples interrupciones). A esta arquitectura de software se la denomina **“Foreground/Background** o también **“arquitectura controlada x eventos (event-driven)”** ya que la ejecución de las tareas depende de que el evento ocurra.

Planificación de tareas en los Sistemas Embebidos

- Las tareas que se ejecutan en el super-loop se denominan **tareas de background** y se ejecutan en función de los eventos asociados a las interrupciones.
- Las ISR para manejar los eventos asincrónicos se denominan **tareas en foreground** (también se pueden pensar como hilos de ejecución).
- Las ISR deben ser de **corta duración**, generalmente activan flags o cambian variables de estado de las tareas, para determinar qué procesamiento debe hacerse para ese evento.
- El comportamiento **No es determinístico**. La ISR cambia el flag pero la tarea se procesa cuando le toque el turno en el loop y si no hay otras interrupciones pendientes.
- Podría darse el caso (es probable) de tener interrupciones simultáneas con lo cual alguna deberá “esperar” a ser atendida, según la prioridad.
- La modificación de una tarea de background afecta la temporización de las demás (poca flexibilidad y escalabilidad). Se deben usar funciones “no bloqueantes”.
- La comunicación entre el lazo principal (Background task) y las ISR (foreground Task) debe realizarse por medio de variables globales y estas variables de comunicación se convierten en **Recursos Compartidos**.
- Si dos o más tareas acceden simultáneamente a un recurso compartido, el código de acceso al mismo se convierte en una **sección crítica** de código y deben tomarse medidas por ejemplo deshabilitar las interrupciones.
- Administrar los modos de bajo consumo en esta arquitectura es complejo ya que debe garantizarse que la CPU ha realizado todas las operaciones de Background y foreground antes de entrar en modo sleep.

Planificación de tareas en los Sistemas Embebidos

- Vamos por otro camino: tareas periódicas

Por ejemplo: ejecutar X() cada 200ms:

```
void main(void)
{
    X_Init();
    while(1)
    {
        X();           // Ejecutar la tarea ( tarda aprox. 10 ms)
        _delay_ms(190); // espera de 190 ms
    }
}
```

- Debemos asumir que conocemos cuanto tarda la ejecución de X() y que es siempre la misma o por lo menos WCET (Worst Case Execution Time) \cong BCET (Best Case Execution Time)
- Si la tarea requiere interacción con periféricos ,el programador podría perder el control de la temporización (posibles bloqueos)
- El delay por software es bloqueante (caso visto en TP1)

Planificación de tareas en los Sistemas Embebidos

- Temporización de una tarea con interrupción periódica de Timer :

```
/*-----*/
Main.c
/*-----*/
#include "io.h"
#include "tick.h"

volatile uint8 Flag_X=0;
/*-----*/
void main(void) {

    X_Init();
    Tick_Init(200);    // configurar un Timer
    Sei();             // enable interrupts

    while(1) {        // Super Loop
        if(Flag_X ==1){
            X();        //se ejecuta cada 200ms
            Flag_X =0;
        }
    }
}
```

```
/*-----*/
--- tick.C -----
/*-----*/
#include "io.h"

extern volatile uint8 Flag_X;

void Tick_Init(void)
{
    // configurar el timer para que genere
    una interrupción periódica cada 200ms
}

/*-----*/
---ISR TIMER: ocurre cada 200 ms
/*-----*/
ISR (Timer_Comp_X_vect)
{
    Flag_X =1; //Tarea programada x Timer
}

/*-----*/
```

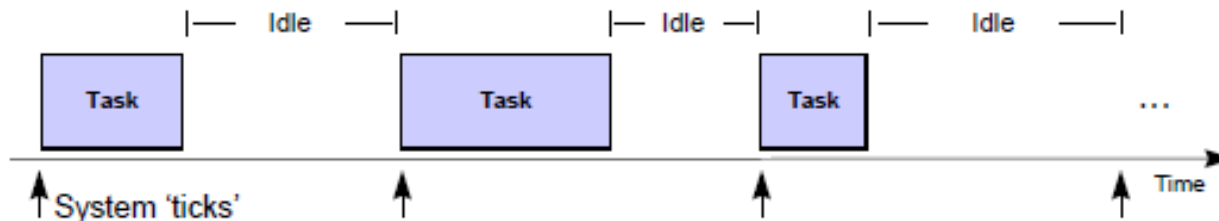
¿Y cómo hago para ejecutar varias tareas con diferentes temporizaciones?

Planificación de tareas en los Sistemas Embebidos

La respuesta:

Arquitecturas de planificación disparadas por tiempo (Time Triggered):

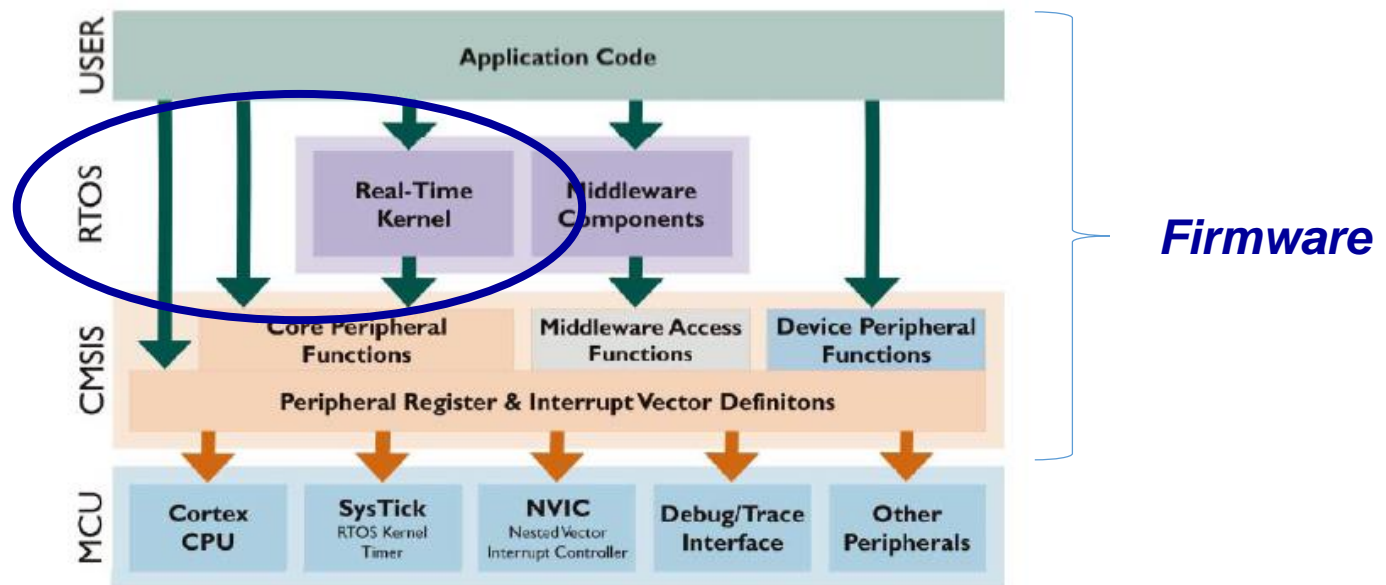
- Tareas planificadas por **una única interrupción** periódica de Timer comúnmente llamada RTI (Real Time Interrupt)
- Una **RTI** es la única **“base de tiempo”** del sistema para temporizar una o más tareas.
- El manejo de los eventos asincrónicos de los periféricos se realiza exclusivamente por encuesta periódica (Polling periódico).
- Cada vez que ocurre la interrupción es como una marca de tiempo o **Tick** del sistema, que permite planificar que tarea corresponde ejecutar.



- Cuando la CPU no tenga que ejecutar tareas (zona IDLE) podemos poner el MCU en bajo consumo (SLEEP) hasta el próximo tick y ahorrar energía.

Introducción a los RTOS

- Vimos que la arquitectura de software de un sistema embebido se puede descomponer en “capas” independientes entre si y que se comunican por medio de interfaces bien definidas.
- Aquí el RTOS es quien administra los recursos de hardware según las aplicaciones que requiera el usuario. El kernel planifica y ejecuta las múltiples tareas que el usuario requiere y además las tareas del propio kernel.
- Claramente un RTOS para los Sistemas Embebidos no es el mismo que un Sistema Operativo tradicional (Win, Linux, android) pero comparten muchos conceptos en común.



Introducción a los RTOS

¿Qué es un RTOS?

- Un **RTOS** (Real Time Operating System) es un *Sistema Operativo de Tiempo Real*.
- Esto significa que es un sistema operativo que provee respuestas a determinados eventos con un “tiempo de respuesta acotado”
- Un sistema de tiempo real “blando” (soft RTOS) acepta que ocasionalmente no se cumplan las cotas temporales (Best effort) Ej: redes de telefonía , servicios en red, interfaz de usuario.
- Un sistema de tiempo real “estricto” (hard RTOS) requiere de la garantía del cumplimiento de los parámetros temporales para evitar efectos potencialmente catastróficos. Ej: control de tráfico aéreo, sistemas de control de un avión o una planta nuclear.

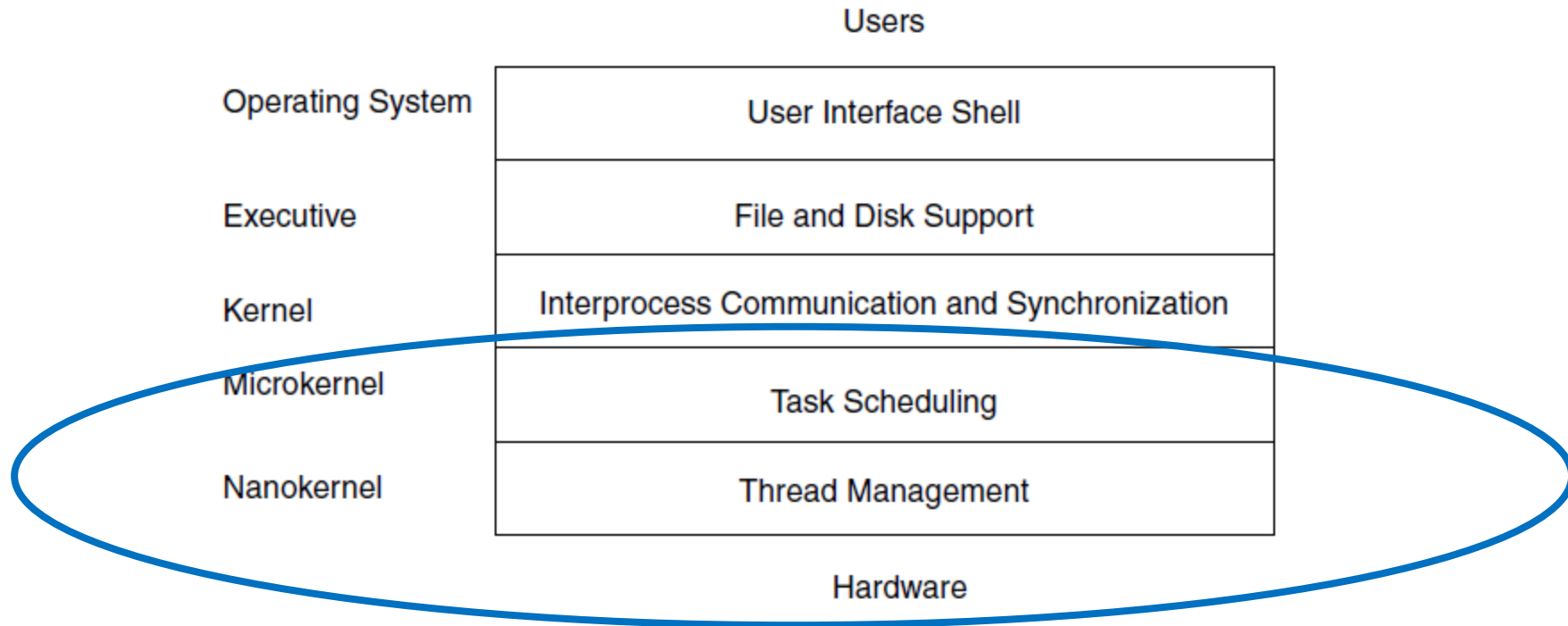
Introducción a los RTOS

- Típicamente, las tareas tienen plazos (deadlines) que son valores de tiempo físico en los cuales se debe completar la tarea.
- Más generalmente, los programas en tiempo real pueden tener todo tipo de restricciones de tiempo, no solo deadlines.
 - Por ejemplo, puede requerirse que una tarea se ejecute no antes de un momento determinado;
 - o puede requerirse que se ejecute no más de una cantidad de tiempo después de que se ejecute otra tarea
 - o se le puede solicitar que se ejecute periódicamente con un período específico.
- Las tareas pueden ser dependientes unas de otras y pueden actuar cooperativamente o pueden ser independientes (excepto que todas comparten los recursos del MCU).
- En un contexto multitareas donde hay más tareas que CPU o tareas que deben ejecutarse en un tiempo preciso es necesaria la planificador de tareas (Task scheduling).

Introducción a los RTOS

- ¿Cómo es un RTOS?

Clasificación de un RTOS según su funcionalidad*



Planificación de Tareas

* *Real-Time Systems Design and Analysis, P. Laplante*

Planificación de tareas

- Un planificador (**Scheduler**) decide cual es la siguiente tarea a ejecutar en el instante de tiempo que la CPU se libera.
- Un planificador puede ser estático (se decide el orden y tiempo de ejecución en el diseño) o dinámico (se decide que tarea ejecutar en tiempo de ejecución)
- Un planificador no-apropiativo (**non-preemptive**) permite a las tareas ejecutarse hasta terminar (run to completion) antes de asignar tiempo de CPU a otra tarea.
- Un planificador apropiativo (**preemptive**) puede tomar la decisión de detener la ejecución de una tarea y comenzar la ejecución de otra, aún cuando la anterior no haya finalizado.
 - *Por ejemplo: una interrupción se apropia del tiempo CPU del programa principal.*
- El planificador puede utilizar la prioridad de una tarea para decidir cuando corresponde su ejecución. Las tareas pueden tener prioridades fijas o pueden alterarse durante la ejecución de programa.
- Un **preemptive priority-based scheduler** siempre ejecuta la tarea habilitada de mayor prioridad mientras que un **non-preemptive priority-based scheduler** usa la prioridad para decidir que tarea corresponde ejecutar luego de que la tarea actual finalice su ejecución y nunca interrumpe la ejecución de una tarea por otra.

Modelo de tarea

Release time: o también tiempo de despacho es el tiempo a partir del cual la tarea está habilitada para ejecutarse

Start time: inicio de la ejecución

Preemption time: la tarea fue suspendida para ejecutar otra

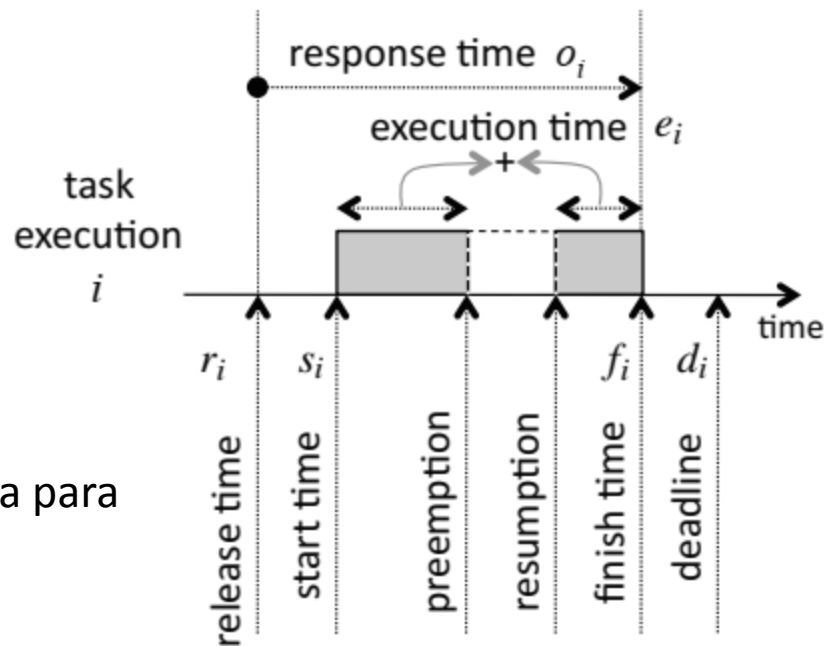
Resumption time: la tarea fue reanudada

Finish time: la tarea finalizó su ejecución

Deadline: es la restricción de tiempo en el cual la tarea debe completarse. Muchas veces esta limitación proviene de las restricciones físicas impuestas por la aplicación y su no cumplimiento puede ser considerado una falla (hard) o una degradación de performance (soft)

Response time: es el tiempo de respuesta y se mide desde la habilitación de la tarea hasta la finalización de la misma

Execution time: es el tiempo que la tarea ha usado la CPU (no tiene en cuenta el tiempo apropiado). Se puede asumir conocido y fijo o con su cota más pesimista **WCET (Worst Case Execution time)**



Arquitectura Time-Triggered

- ¿Cómo hacemos un **RTOS Embebido Simple** entonces?
 - Un **planificador de tareas (scheduler)** que permite decidir que tarea corresponde ejecutar en base a la temporización basado en una **RTI**.
 - Más un **despachador de tareas (dispatcher)** que permita ejecutar las tareas planificadas con distintas prioridades.
 - El RTOS y las tareas de aplicación del usuario son parte del mismo proyecto. NO es una aplicación independiente, pero sí es un módulo portable.
 - Con un poco de trabajo se puede lograr un RTOS independiente, portable y escalable (ver bibliografía M. Pont).

sEOS: Planificador o Scheduler

```
volatile unsigned char Flag_X=0,  
volatile unsigned char Flag_Y=0;  
volatile unsigned char Flag_Z=0;
```

```
/* ----- */  
Variables públicas del planificador  
/* ----- */
```

```
static unsigned char contX=199,  
static unsigned char contY=49;  
static unsigned char contZ=9;
```

```
/* ----- */  
Variables privadas del planificador  
/* ----- */
```

```
/* ----- */  
---ISR TIMER : ocurre cada T ms  
/* ----- */
```

ISR (Timer_OVF)

```
{  
    SEOS_SCH_Tasks();  
}
```

Si suponemos $T=1ms$

Z() se ejecutará cada 10ms

Y() cada 50ms

X() cada 200ms

void SEOS_SCH_Tasks (void)


```
{  
    if (++contX==200) {  
        Flag_X=1; //Tarea programada cada 200 ms  
        contX=0;  
    }  
    if (++contY==50) {  
        Flag_Y=1; //Tarea programada cada 50 ms  
        contY=0;  
    }  
    if (++contZ==10) {  
        Flag_Z=1; //Tarea programada cada 10 ms  
        contZ=0;  
    }  
}
```


sEOS: Despachador o Dispatcher

```
/*-----  
Main.c  
-----*/  
#include "io.h"  
#include "seos.h"  
  
/*-----*/  
void main(void)  
{  
    // Inicializar MCU, RTC, y tareas  
    while(1) {  
        SEOS_Dispatch_Tasks();  
        SEOS_Go_To_Sleep();  
    }  
} /*-----*/
```

P: prioridad, está dada por el orden en la verificación y ejecución (prioridad estática)

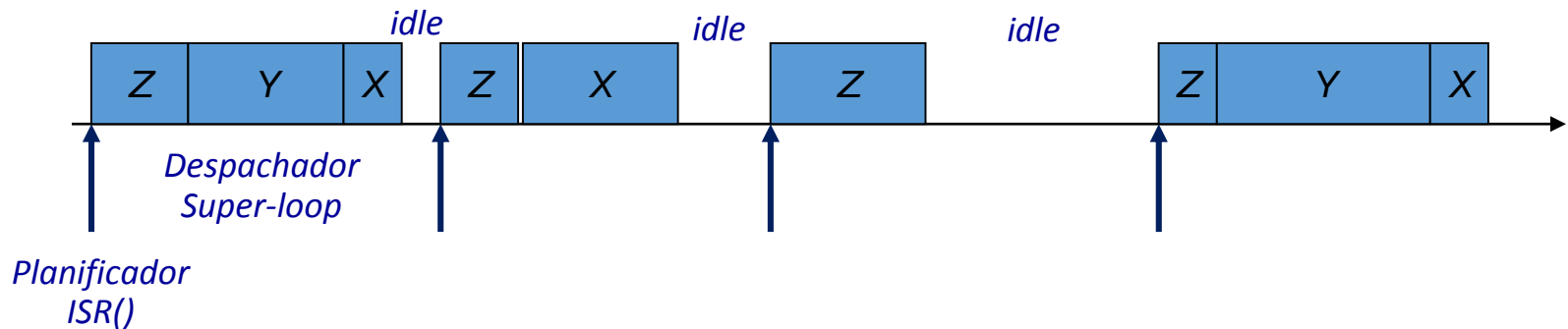
```
void SEOS_Dispatch_Tasks (void) {  
    if (Flag_Z) {  
        Z(); //Tarea programada cada 10 ms  
        Flag_Z =0;  
    }  
    if (Flag_Y) {  
        Y(); //Tarea programada cada 50 ms  
        Flag_Y =0;  
    }  
    if (Flag_X) {  
        X(); //Tarea programada cada 200 ms  
        Flag_X =0;  
    }  
}
```



Tenemos la base de un RTOS **non-preemptive priority-based scheduler** al que llamaremos de aquí en más **sEOS** (simple Embedded Operating System)

sEOS: Funcionamiento

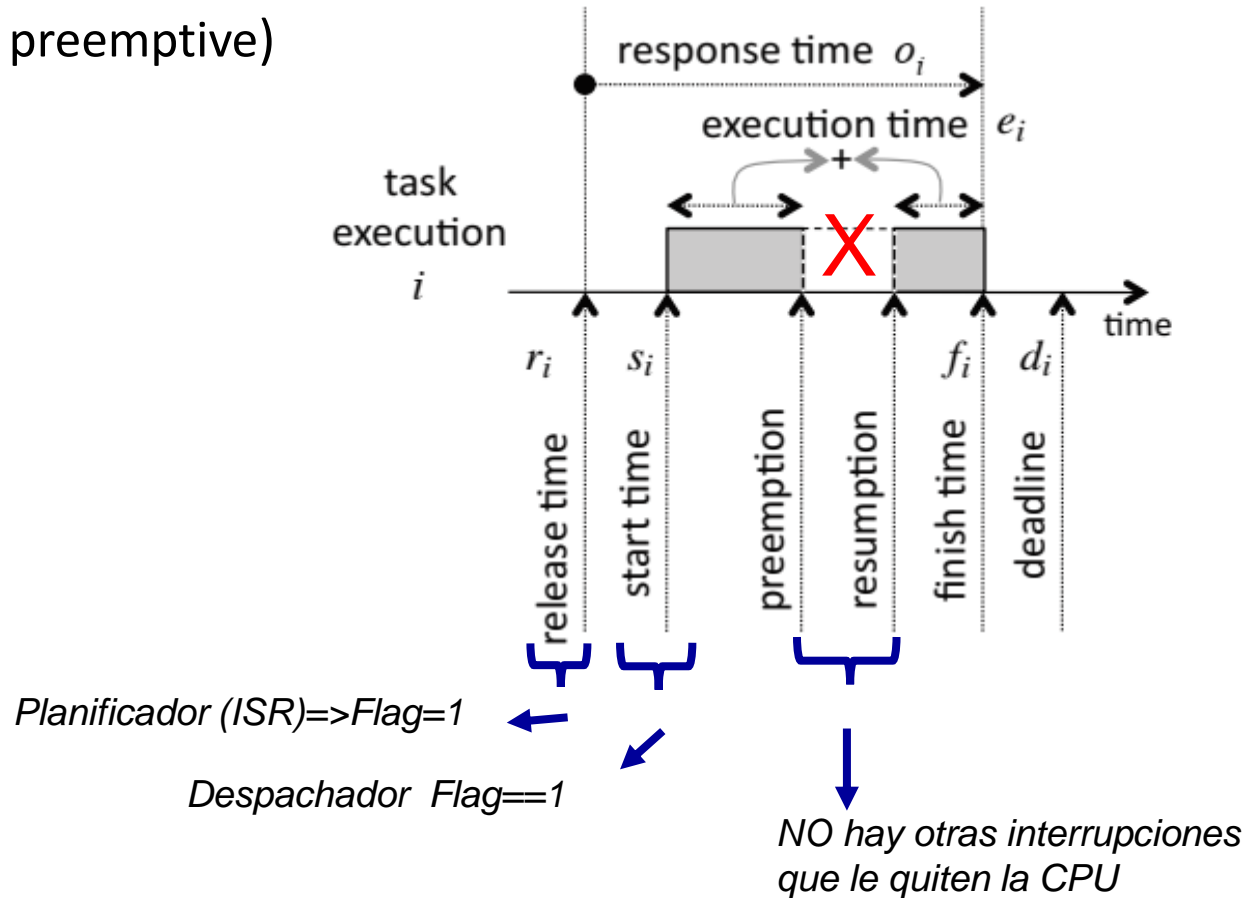
- Cooperativo (non- preemptive)



- Una tarea no se detiene hasta que haya terminado (*Run to Completion*) caso más simple y más confiable que la arquitectura Event Driven.
- Pero esto impone restricciones al tiempo de ejecución de la siguiente tarea.
- Cada tarea puede ser una *MEF* de manera de ejecutar subtareas y así pasar el control a otra tarea rápidamente (cooperativismo).
- Requiere un análisis estricto de la planificación pero *es determinístico*.
- *No hay* conflictos de recursos compartidos o secciones críticas

sEOS: Funcionamiento

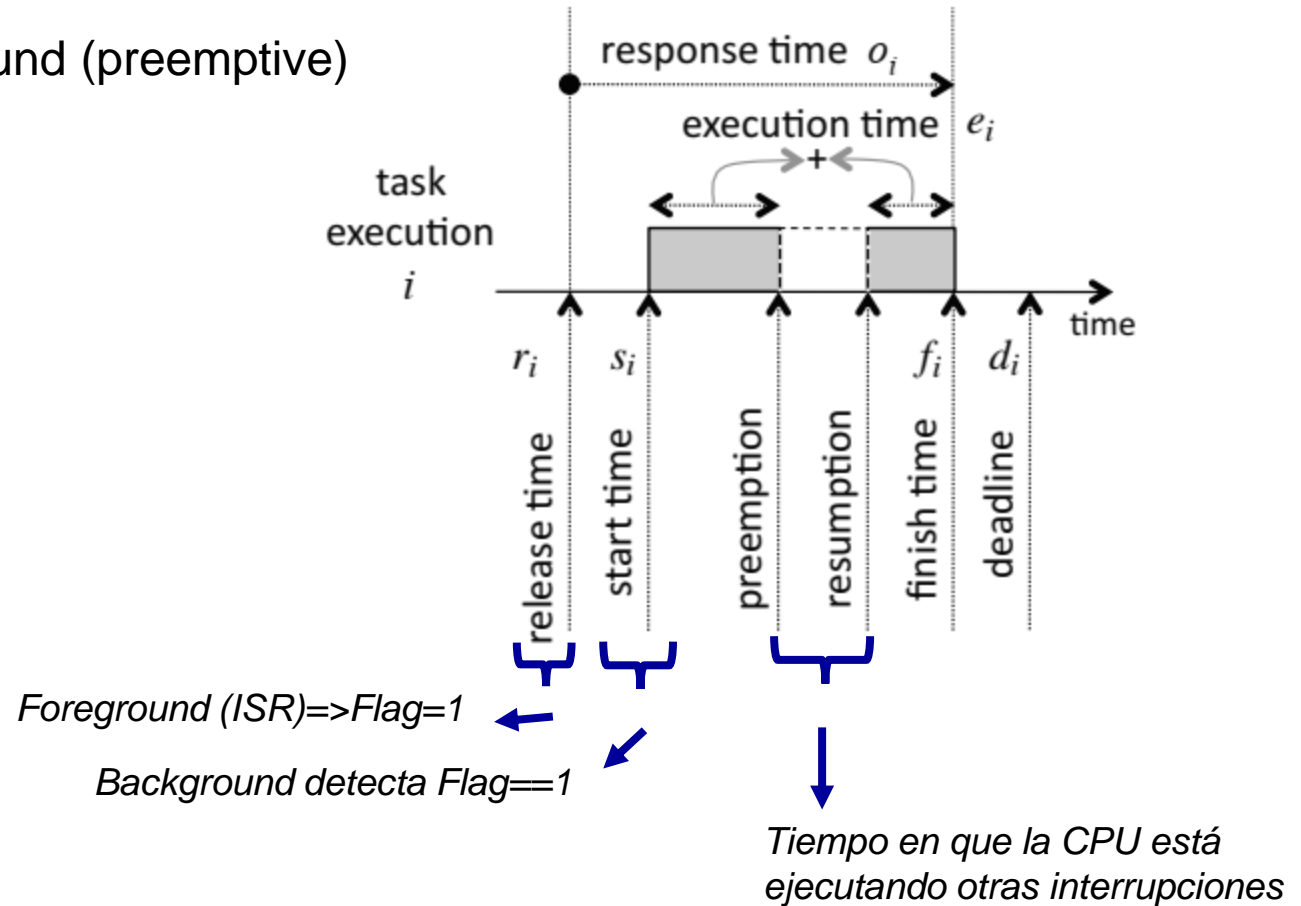
- Cooperativo (non- preemptive)



- El tiempo de respuesta es fácil de calcular (y conozco el WCET)
- La diferencia de tiempo $s_i - r_i$ depende del tiempo de ejecución de las otras tareas
- El tiempo en suspensión es 0 si diseñamos correctamente la planificación.

sEOS: Comparación con otra arquitectura

Foreground / Background (preemptive)

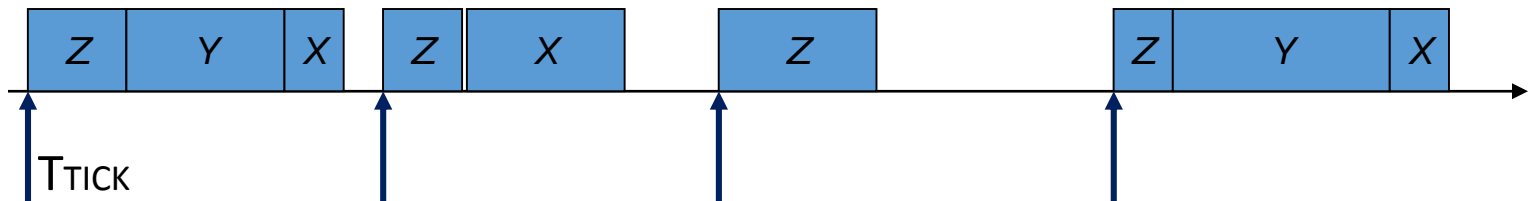


- El tiempo de respuesta es complejo de calcular (aunque conozca el WCET)
- La diferencia de tiempo $s_i - r_i$ depende del tiempo de ejecución de otras tareas de background
- El tiempo de suspensión depende del tiempo de ejecución de las tareas de foreground

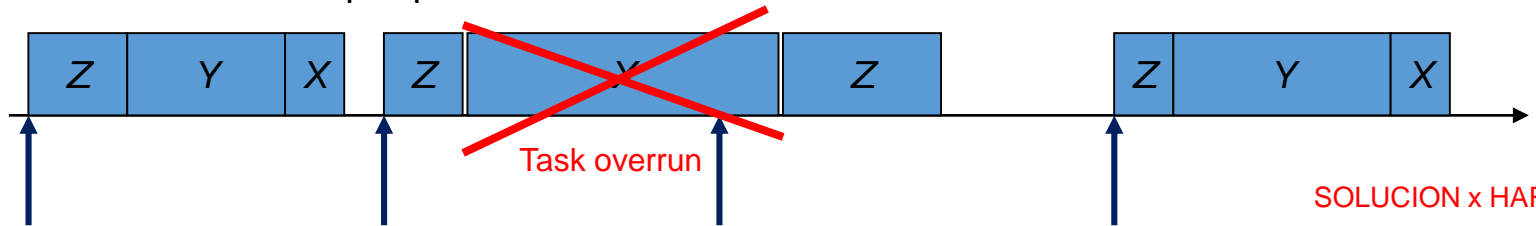
sEOS: Diseño

- 1: Requiere que bajo cualquier circunstancia:

$$\sum WCET_{tarefas} < T_{TICK}$$



- Debe diseñarse para que no ocurran **bloqueos** dentro de las tareas que hagan que la tarea tarde más tiempo que el tick del sistema.



SOLUCION x HARD:

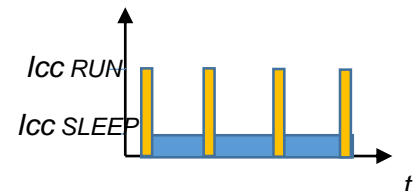
- AUMENTAR EL CLK CPU
- CAMBIAR DE MCU

- El porcentaje de uso de la CPU en cada Tick puede calcularse como:

$$CPU_{LOAD}[\%] = (\sum WCET_{tarefas} + ISR_{Tick}) * 100 / T_{TICK}$$

- El consumo promedio por Tick puede calcularse como:

$$ICPU[mA] = (CPU_{RUN} * I_{DDRUN} + CPU_{IDLE} * I_{DDSLEEP}) / T_{TICK}$$



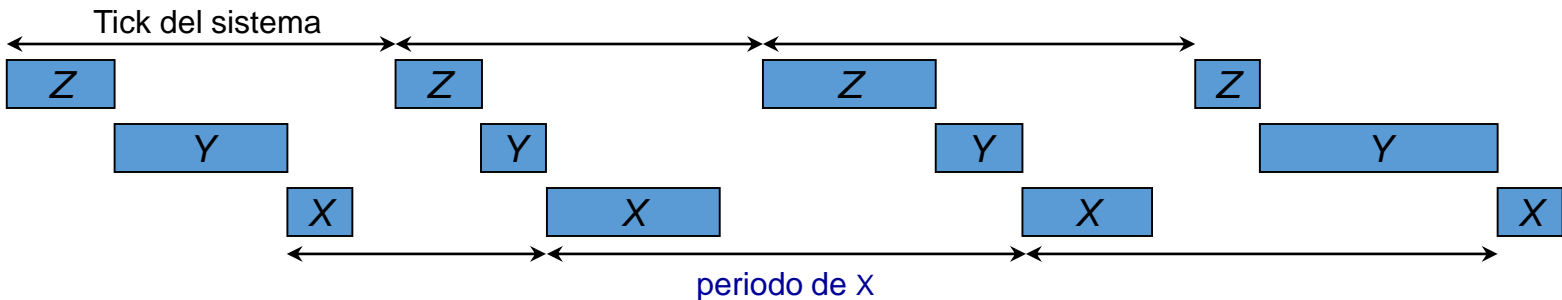
sEOS: Diseño

- 2: Esquema fijo de prioridades

Asignación de prioridades por criterio RMS ([Rate Monotonic Scheduling](#)): a mayor frecuencia de ejecución (mayor tasa) mayor prioridad de la tarea

- 3: ¿Solo la tarea más prioritaria es estrictamente periódica?

El start time de X e Y es aleatorio (jitter) y X e Y no son estrictamente periódicas



- 4: Si el resto de las tareas (menos prioritarias) tiene una temporización mucho más lenta el sistema funciona bien (el jitter es despreciable).
 - Ej: Z() se ejecuta cada 10ms, Y() cada 100ms y x() cada 1s

sEOS: Diseño

- 5: **Determinación del periodo de la interrupción de Timer (Tick del sistema)**

- Supongamos que tenemos las tareas X, Y, Z con la siguiente temporización:
 - X: se ejecuta cada 10ms
 - Y: cada 30ms
 - Z: cada 25ms
- El Factor Común Máximo entre las tres tareas es: 5 ms, entonces elegimos Tick=5ms:
 - X: se ejecuta cada 2 ticks
 - Y: cada 6 ticks
 - Z: cada 5 ticks

- Vemos que se ejecuta en cada Tick:

Tick 0: X , Y , Z

Tick 1: ninguna

Tick 2: solo X

Tick 3: ninguna

...

Tick 6: X , Y

Tick 10: X , Z

Tick 12: X , Y

Tick 18: X , Y

y así siguiendo...

Varias tareas en 1 tick
¿se puede evitar?

- La secuencia se repite cada 30 Ticks (Mínimo Común Múltiplo) es decir cada 150ms. Esto se denomina Hiperperiodo.

sEOS: Diseño

- Determinación del Intervalo Tick – continua ejemplo

- Podemos subdividir el tick anterior en un número entero de slots (o intervalos de tiempo) y obtener un nuevo tick

podemos dividir x 5 el anterior => nuevo Tick=1ms

- Luego podemos usar un slot para cada tarea :
 - X: se ejecuta cada 10ms empezando en slot 0
 - Y: cada 30ms empezando en slot 1
 - Z: cada 25ms empezando en slot 2

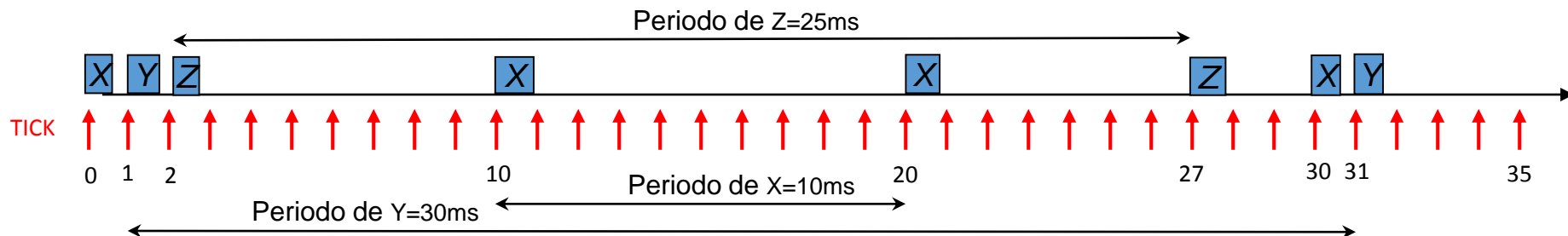
$$WCET_x < T_{TICK}$$

$$WCET_y < T_{TICK}$$

$$WCET_z < T_{TICK}$$

- Por lo tanto:
 - X: se ejecuta cada 10 ticks
 - Y: cada 30 ticks empezando un tick después
 - Z: cada 25 ticks empezando dos ticks después

static unsigned char contX=9,
static unsigned char contY=28;
static unsigned char contZ=23;



sEOS: Planificador o Scheduler

```
volatile unsigned char Flag_X=0,  
volatile unsigned char Flag_Y=0;  
volatile unsigned char Flag_Z=0;
```

```
/* ----- */  
Variables públicas del planificador  
/* ----- */
```

```
static unsigned char contX=9,  
static unsigned char contY=28;  
static unsigned char contZ=23;
```

```
/* ----- */  
Variables privadas del planificador  
/* ----- */
```

```
/* ----- */  
---ISR TIMER : ocurre cada 1 ms  
/* ----- */
```

```
ISR (Timer_CompA_vect)
```

```
{  
    SEOS_SCH_Tasks();  
}
```

```
void SEOS_SCH_Tasks (void)
```

```
{  
    if (++contX==10) {  
        Flag_X=1; //Tarea programada cada 10 ms  
        contX=0;  
    }  
    if (++contY==30) {  
        Flag_Y=1; //Tarea programada cada 30 ms  
        contY=0;  
    }  
    if (++contZ==25) {  
        Flag_Z=1; //Tarea programada cada 25 ms  
        contZ=0;  
    }  
}
```

sEOS: Diseño

- Se debe asegurar que las tareas se ejecuten en el plazo de un TICK. Esto significa, que hay que diseñarlas pensando cooperativamente...
- ¿Cómo?
 - a- **Uso de Timeout por soft o Hard**
Si la tarea no termina a tiempo, => TimeOut (falló el diseño)
 - b- **Uso de Tareas Multi-etapas:**
 - Una tarea de larga duración se divide en tareas más cortas
 - Ejemplo: Comunicación serie, transmite y recibe 1 byte a la vez
 - Ejemplo: Un LCD, actualiza un caracter a la vez
 - c- **Uso de Tareas Multi-Estados**
 - Modelizar las tareas como MEF temporizadas

sEOS: Diseño

a- Uso de Timeout por soft o Hard

- Existen porciones de código BLOQUEANTES que deben evitarse

Ej: ~~`// Leyendo un switch
while (Switch_pin == 0)
; //esperar`~~

Ej: `// Esperando fin de conversión AD
while(! Flag)
; //esperar`



Uso de pooling periodico:

```
if (Switch_pin == 0) //depende del usuario  
{  
    return 0; // No se presiono tecla  
}  
else  
{  
    return 1; // Se presiono tecla  
}
```

- Esta estructura dentro de una función puede hacer que la tarea se bloquee esperando una condición que podría “exceder” al tiempo del tick
- Evitar funciones “bloqueantes” usando funciones de sondeo, esto es equivalente a reemplazar while por if-else o usando contadores de Timeout.

sEOS: Diseño

a- Uso de Timeout por software o Hardware

- Timeout por soft

```
long Timeout_loop = 0;
while ( (! Flag) && (++Timeout_loop != 10000)) ; //espera que se active el flag
if(Timeout_loop==10000) {
    System_State=ERROR;      //el flag no se activó en el plazo planificado
    return;
}
```

- Timeout por hard (usando timer)

```
Init_timer (timeout);
while ( (!Flag) && (! TIMER_OVF) )
if(TIMER_OVF) {
    System_State=ERROR;
    return;
}
```

*Otra opción al problemas de los bucles
Bloqueantes es el Perro Guardián*

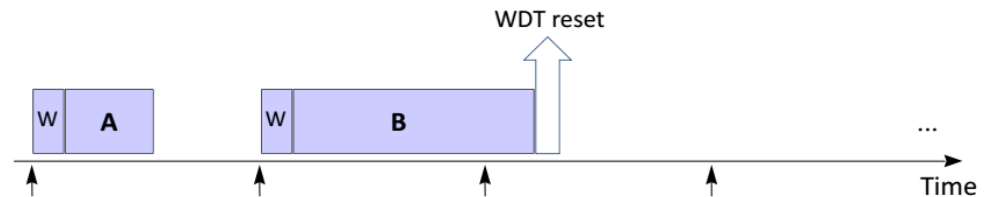


Figure 19: A WDT reset caused by a task overrun.

sEOS: Diseño

Se debe asegurar que las tareas se ejecuten en el plazo de un TICK. Esto significa, que hay que diseñarlas pensando cooperativamente...

- ¿Cómo?
 - a- Uso de Timeout por soft o Hard
Si la tarea no termina a tiempo, un TimeOut la detiene
 - b- Uso de Tareas Multi-etapas:
 - Una tarea de larga duración se divide en tareas más cortas
 - Ejemplo: Comunicación serie, transmite y recibe 1 byte a la vez
 - Ejemplo: Un LCD, actualiza un caracter a la vez
 - c- Uso de Tareas Multi-Estados
 - Modelizar las tareas con MEF temporizadas
- Veremos el punto b) en las próximas clases ...

sEOS: Diseño

Se debe asegurar que las tareas se ejecuten en el plazo de un TICK. Esto significa, que hay que diseñarlas pensando cooperativamente...

- ¿Cómo?
 - a- Uso de Timeout por soft o Hard
 - Si la tarea no termina a tiempo, un TimeOut la detiene
 - b- Uso de Tareas Multi-etapas:
 - Una tarea de larga duración se divide en tareas más cortas
 - Ejemplo: Comunicación serie, transmite y recibe 1 byte a la vez
 - Ejemplo: Un LCD, actualiza un caracter a la vez
 - c- Uso de Tareas Multi-Estados
 - Modelizar las tareas con MEF temporizadas
- Ya vimos implementación de MEF en el TP2

sEOS: Limitaciones

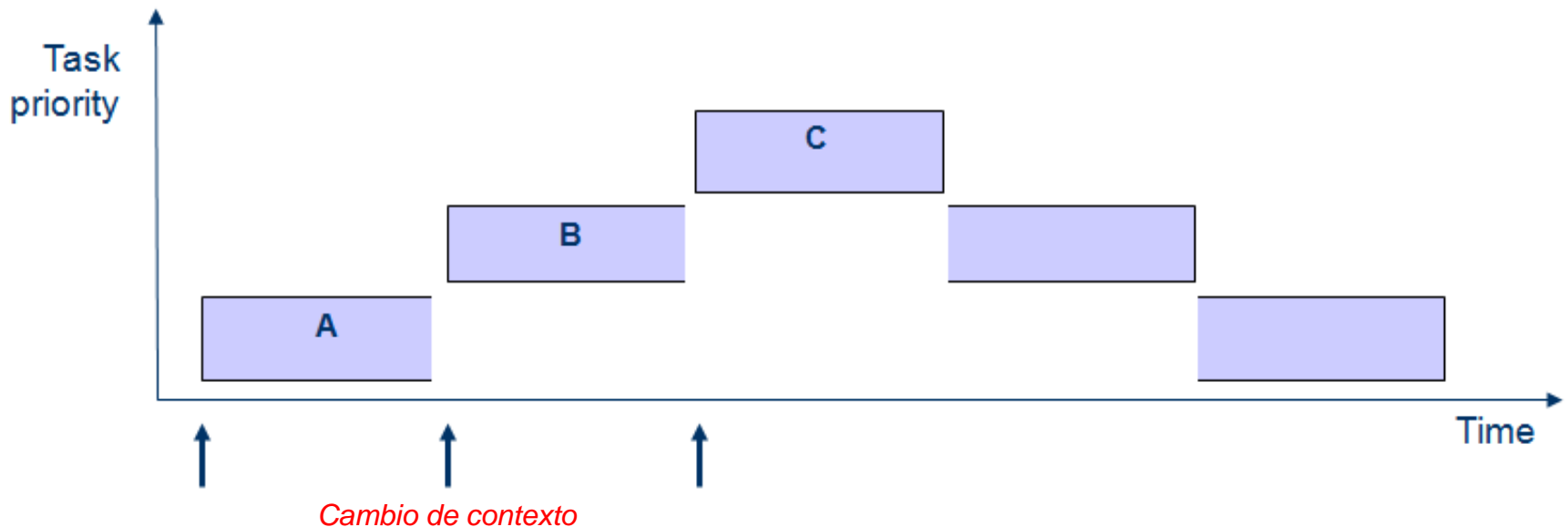
- *Escalabilidad:*
 - Sistemas reactivos complejos, con muchos estados o muchos requerimientos temporales disímiles son propensos a implementar con RTOS tradicionales.
- *Imposibilidad de manejar requerimientos heterogéneos. Por ejemplo:*
 - 1 Tarea de larga duración 100ms a repetir cada 1s (long task)
 - 1 Tarea de corta duración 0.1ms a repetir cada 1ms (frequent task)
 - No se cumple : $WCET < \text{Tick Interval}$

¿Cómo hacemos?

RTOS apropiativo

Respuesta: Sistemas apropiativos (preemptive):

Una tarea se ejecuta durante un tiempo determinado (time slice)
vencido el plazo, se pausa (haya terminado o no) y se pasa a ejecutar otra tarea.
La tarea pausada debe “esperar” el turno para continuar su ejecución.



RTOS apropiativo

- Modelo de tareas concurrentes :
- El nombre del juego es: Bloquear !!!!

Nuevos Problemas de:

- Sincronización
- race condition
- Recursos compartidos
- Inversión de prioridades
- Dead locks
- starvation

```
void thread_alarm() {           // RTOS thread routine
    pinMode(SW_PIN, INPUT);     // setup: set the Switch pin as input
    while (1) {                 // endless loop
        if (digitalRead(SW_PIN) == HIGH) { // is the switch depressed?
            digitalWrite(ALARM_PIN, HIGH); // start the alarm
        }
        else {
            digitalWrite(ALARM_PIN, LOW); // stop the alarm
        }
        RTOS_delay(100);
    }
}
```

```
void thread_blink() {          // RTOS thread routine
    pinMode(LED_PIN, OUTPUT);  // setup: set pin as output
    while (1) {                // endless loop
        digitalWrite(LED_PIN, HIGH); // turn the LED on
        RTOS_delay(1000);          // wait for 1000ms
        digitalWrite(LED_PIN, LOW);  // turn the LED off
        RTOS_delay(1000);          // wait for 1000ms
    }
}
```

- FreeRTOS (**preemptive priority-based scheduler**) es el RTOS más utilizado hoy en día para microcontroladores

<https://www.freertos.org/>

Resumen RTOS

- Las arquitecturas de planificación de tareas que vimos para Sistemas Embebidos son:
 - **Arquitectura Super Loop (Round Robin o cíclica):**
 - Simple.
 - Control x polling.
 - Sin interrupciones.
 - Temporización por retardos bloqueantes
 - **Arquitectura Background / Foreground (interrupt - Driven):**
 - Las tareas se ejecutan en respuestas a eventos asincrónicos (múltiples interrupciones)
 - Difícil de predecir para todas las circunstancias.
 - Problemas de recursos compartidos (secciones críticas).
 - **Arquitectura Time-Triggered:**
 - Única interrupción RTI
 - Los periféricos se utilizan por polling periódico
 - Cooperativo,
 - Predecible => más confiable para aplicaciones de tiempo real y críticas
- A su vez pueden clasificarse en:
 - **No apropiativas (non-preemptive):**
 - Las tareas se ejecutan hasta completarlas. No hay intercambio de contexto de las tareas.
 - **Apropiativas (preemptive):**
 - Las tareas tienen asignado un intervalo de tiempo fijo, luego del cual son detenidas. Hay intercambio de contexto. Aumentan los problemas de recursos compartidos y sincronización.

Bibliografía

- *“Patterns for Time-Triggered Embedded Systems”*, Michael J. Pont. 2014, Published by SafeTTY Systems. Disponible en <https://www.safetty.net/publications/pttes>
- *“Real-Time Systems Design and Analysis”*. P. Laplante 3rd Ed. John Wiley & Sons 2004 (en biblioteca).
- *“Embedded Microcomputer System, Real-Time Interfacing”*. J. Valvano 2nd Ed. Thomson 2007 (en biblioteca).
- *“Practical UML Statecharts in C/C++, 2nd Edition: Event-Driven Programming for Embedded Systems”*, Miro Samek, Newnes 2008, pdf descarga gratis en la web: <https://www.state-machine.com/>