Sistemas operativos en tiempo real. Conceptos

Juan Alarcón. jalarcon@frba.utn.edu.ar

Tiempo real.

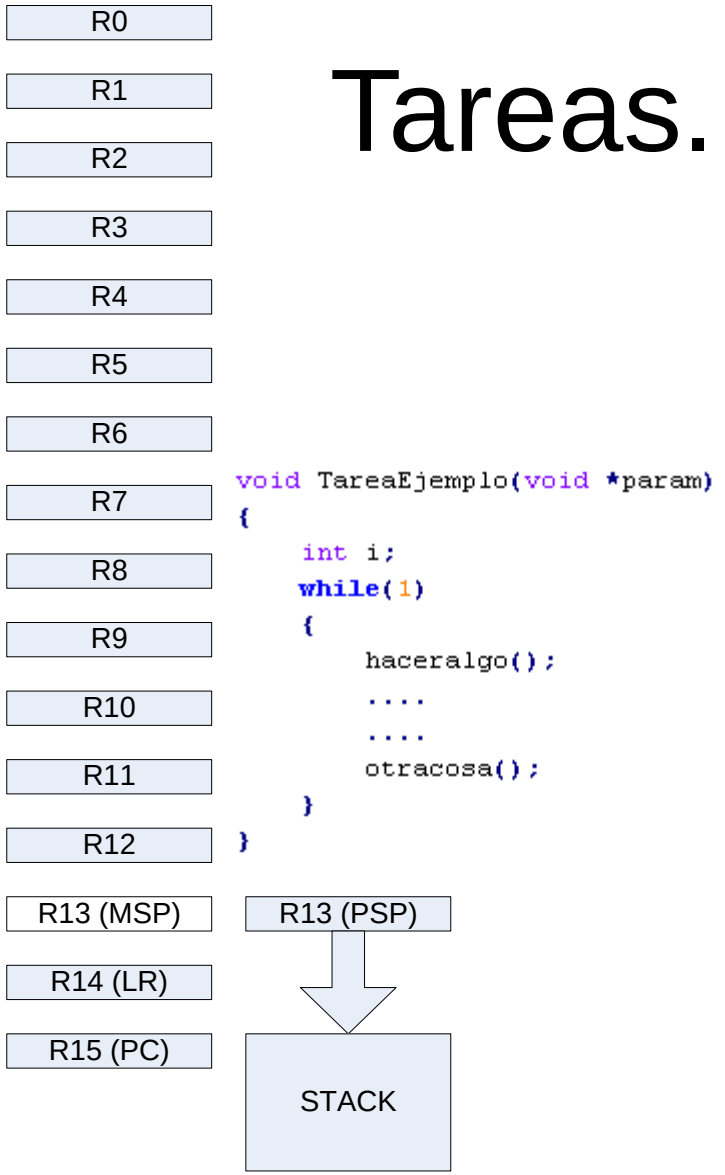
- Un sistema va a ser un sistema de tiempo real, cuando se define que una operación es correcta no solamente porque es lógicamente correcta, también debe ser realizada antes de un tiempo determinado.

- Se definen tres tipos de sistemas de tiempo real:
  - ***Estrictos (hard real time)***. Si no se alcanza la respuesta en el tiempo determinado se considera que el sistema ***falla.***
  - ***Flexible (soft real time).*** Pueden no alcanzar el tiempo de generación de la salida ocasionalmente. El valor de la respuesta decrece con el tiempo (adquisición de datos).
  - ***Firme (firm real time).*** Se puede no alcanzar el tiempo de generación de salida ocasionalmente. Una respuesta tardía no tiene valor (ej: sistema multimedia)

Procesos, Tareas.

- Un concepto fundamental de los sistemas operativos es el concepto de ***proceso***.
- Un proceso es básicamente un tramo o un programa en ejecución.
- Es decir, un proceso no sólo está compuesto por el código que ejecuta, también por su ***contexto***
- A lo largo del curso los términos proceso y tarea se van a utilizar como sinónimos (sólo válido para TD2).

Tareas. Contexto.

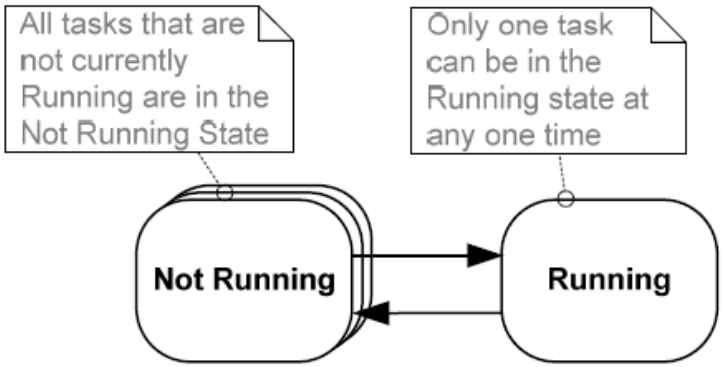



```
void TareaEjemplo(void *param)
{
    int i;
    while(1)
    {
        haceralgo();
        ....
        ....
        otracosa();
    }
}
```

- El contexto va a estar definido por el contenido de los registros, la pila del proceso y el código de la tarea, su estado y prioridad.
- La tarea va a estar compuesta por el programa y la “foto” del procesador en ese momento.

Planificador (scheduler).

- Una de las patas de los RTOS son las tareas. La otra pata es el planificador.
- El planificador es un programa que se va a encargar (con algún criterio determinado) de ejecutar, salvar, recuperar y dejar de ejecutar las diferentes tareas de la aplicación.
- Las tareas van a tener por lo pronto dos estados:
  - Ejecutando
  - En espera



Tipos de planificadores

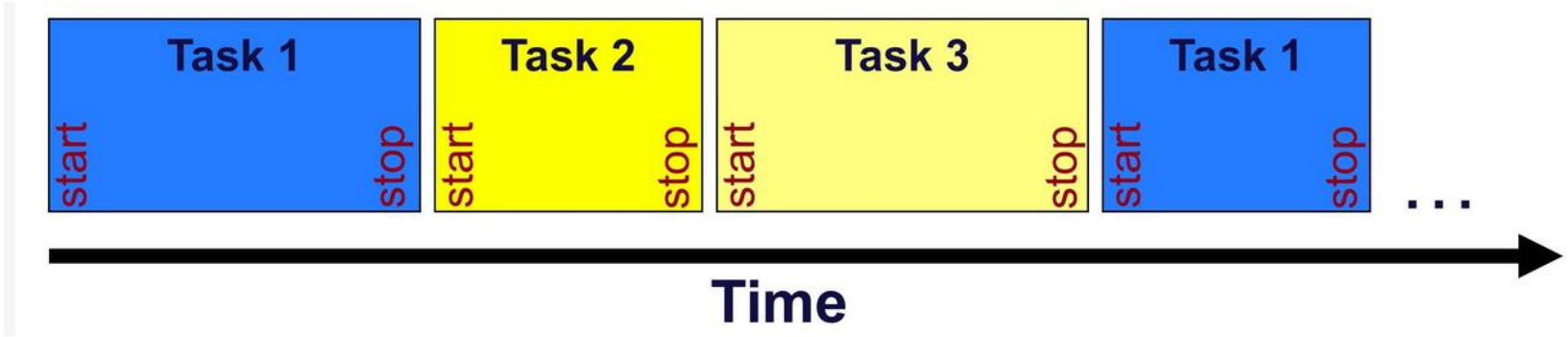
- ***Cooperativo (RTC).*** Cada tarea se ejecuta hasta que termina. La prioridad siempre la tiene la tarea en ejecución. El planificador es una lista de tareas a ejecutar.

- ***Round-Robin (Time Slice).*** El planificador tiene la capacidad de "poner y sacar" tareas en ejecución y a cada tarea se le asigna una cantidad de "ticks" de ejecución. Todas las tareas tienen la misma prioridad.

- ***Apropiativo (preemptive).*** En este tipo de planificación cada tarea tiene una prioridad determinada. El planificador va a ejecutar la tarea de mayor prioridad que esté en condiciones de ejecutarse. Este tipo de planificador es el que va a minimizar la latencia, ya que los eventos suelen modelarse con tareas de prioridad elevada.

- ***Compuestos***. Son los que combinan el comportamiento de los anteriores.

Planificador Cooperativo.

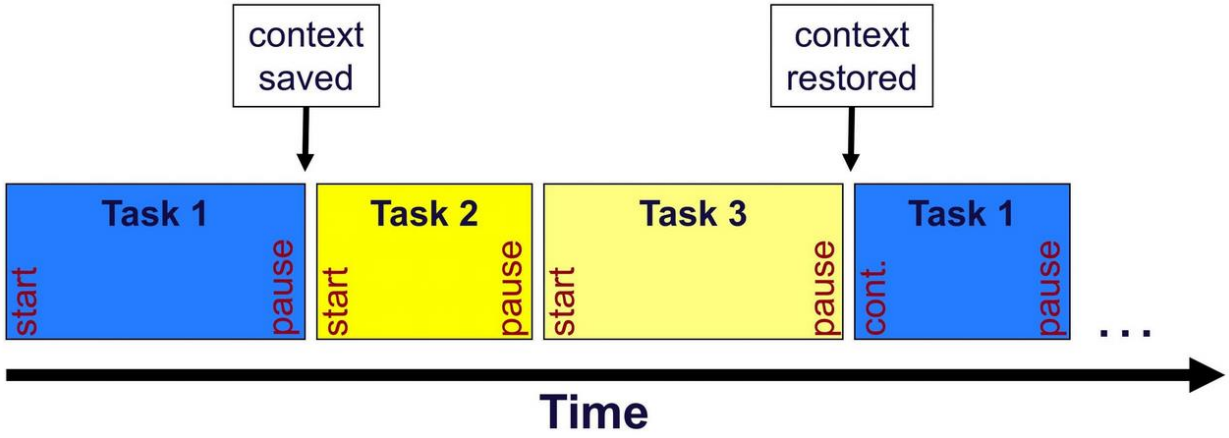


- El planificador cooperativo o RTC (run to completion scheduler) es el más sencillo de todos. Básicamente es una lista de tareas/funciones que se va a ejecutar en alguna secuencia (lista de secuencias) definida por el diseñador del sistema.

Planificador Cooperativo (2)

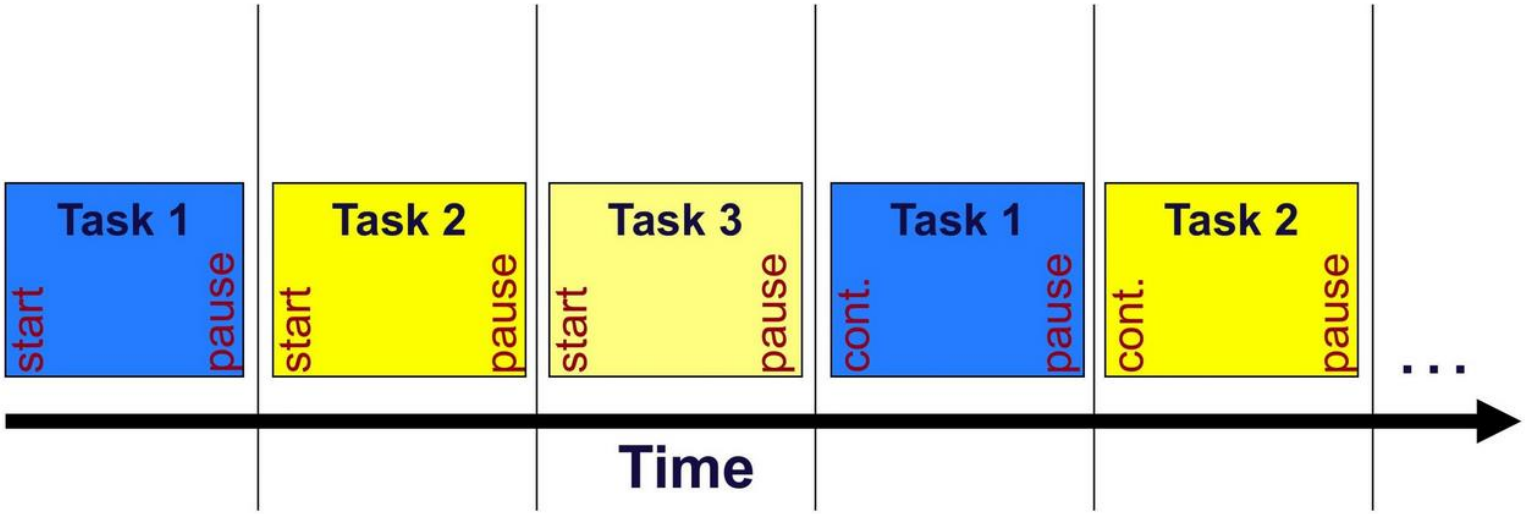
- Es muy sencillo y usa pocos recursos, por lo que suele ser tomado como primera opción, si la aplicación lo permite. ☺
- Suele ser muy portable porque en general no requiere de partes en assembler. ☺
- La comunicación entre tareas es muy sencilla y el fácil predecir su comportamiento. ☺
- Si una tarea se cuelga / pierde el sistema se pierde completo. ☹
- Si bien se puede estimar la latencia a un evento, la tarea que está corriendo tiene prioridad sobre todas las demás hasta su fin y después según el orden fijado. ☹

Planificador Round Robin (RR)



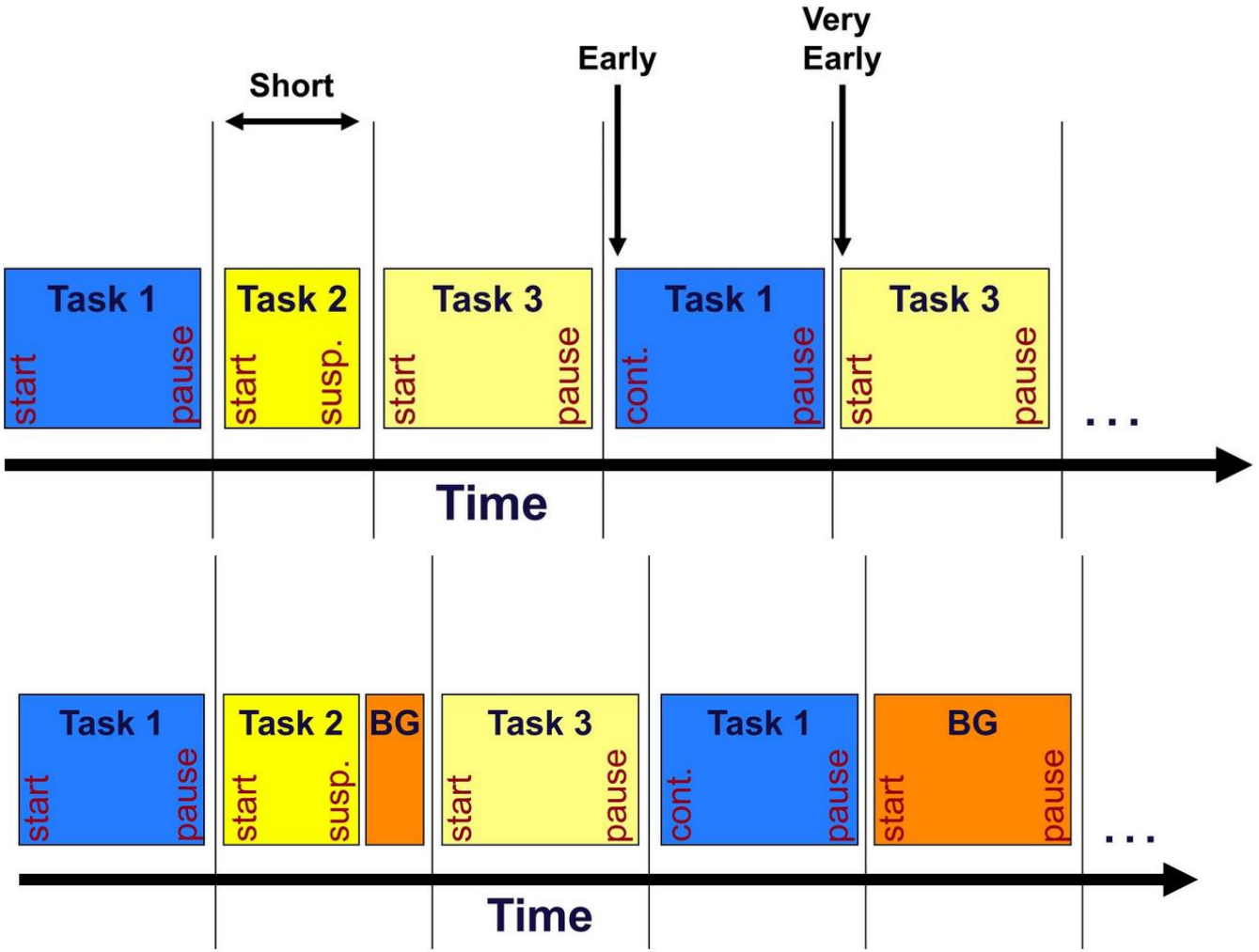
- El planificador RR no necesita que las tareas terminen, sino que el planificador por algún mecanismo (excepción) salva el contexto y ejecuta la tarea siguiente a pedido de la tarea.
- Es más flexible y complejo que el planificador cooperativo, la gran ventaja es que la planificación es independiente del código de las tareas.
- Su desventaja, la comunicación entre tareas es más compleja y la carga del planificador es mayor.
- Se necesita mantener una pila por cada tarea.

Planificador por división de tiempo



- El planificador por división de tiempo (time slice) es el paso siguiente al round robin. El planificador le asigna a cada tarea una cantidad de ticks en la que se ejecuta y luego cambia a la siguiente.
- Es simple de entender y predecible.
- La desventaja que puede presentar este sistema es que si una tarea termina o entrega el control antes de tiempo el sistema pierde la temporización.

Planificador por división de tiempo (2)

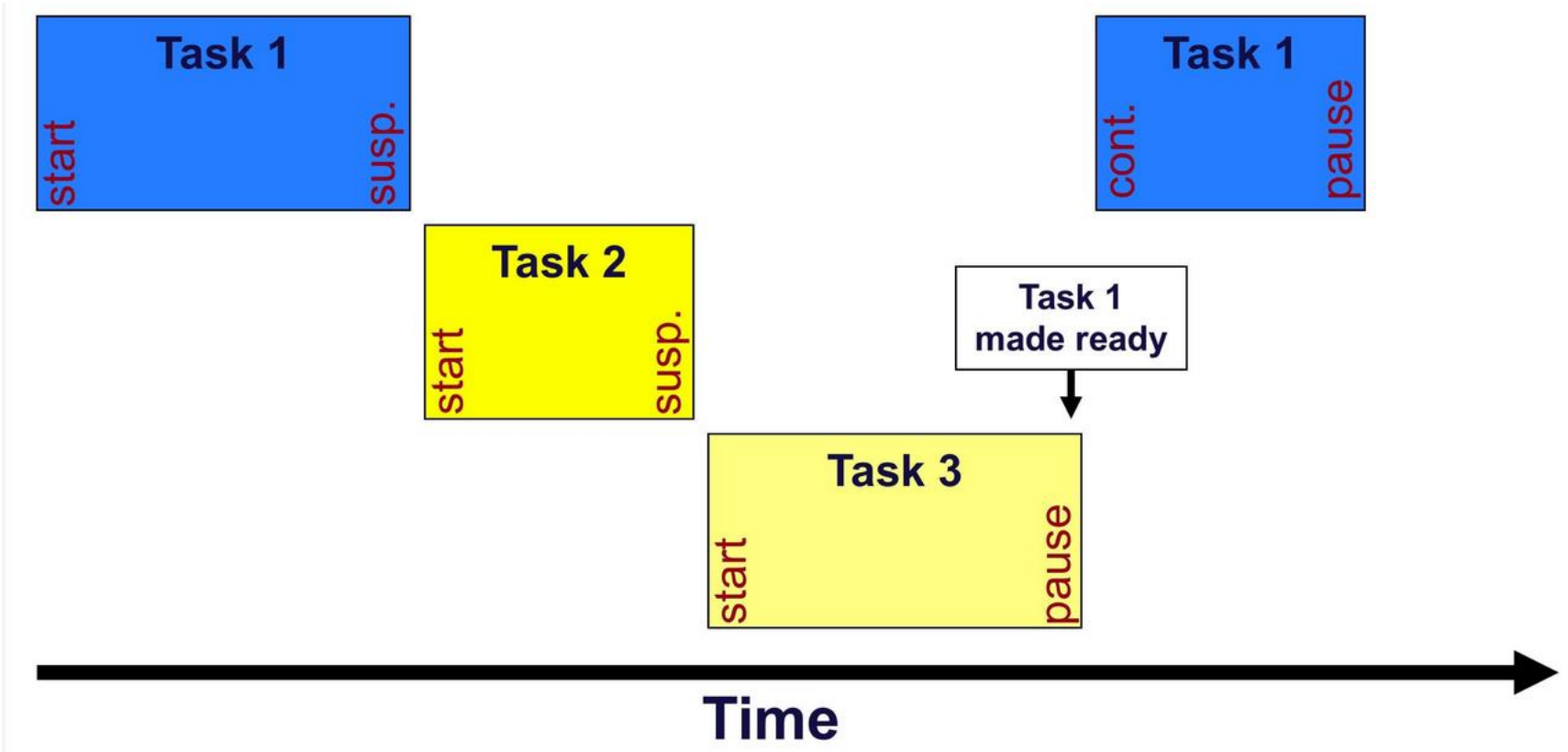


Planificador por división de tiempo (3)

- El planificador TS puede agregar una tarea que no hace nada (background - idle), sólo consumir tiempo, para evitar que el sistema pierda la temporización.

- Este tipo de sistemas no es muy apto para atender eventos que requieran poca latencia ya que en el peor caso hay que esperar los ticks de todas las tareas.

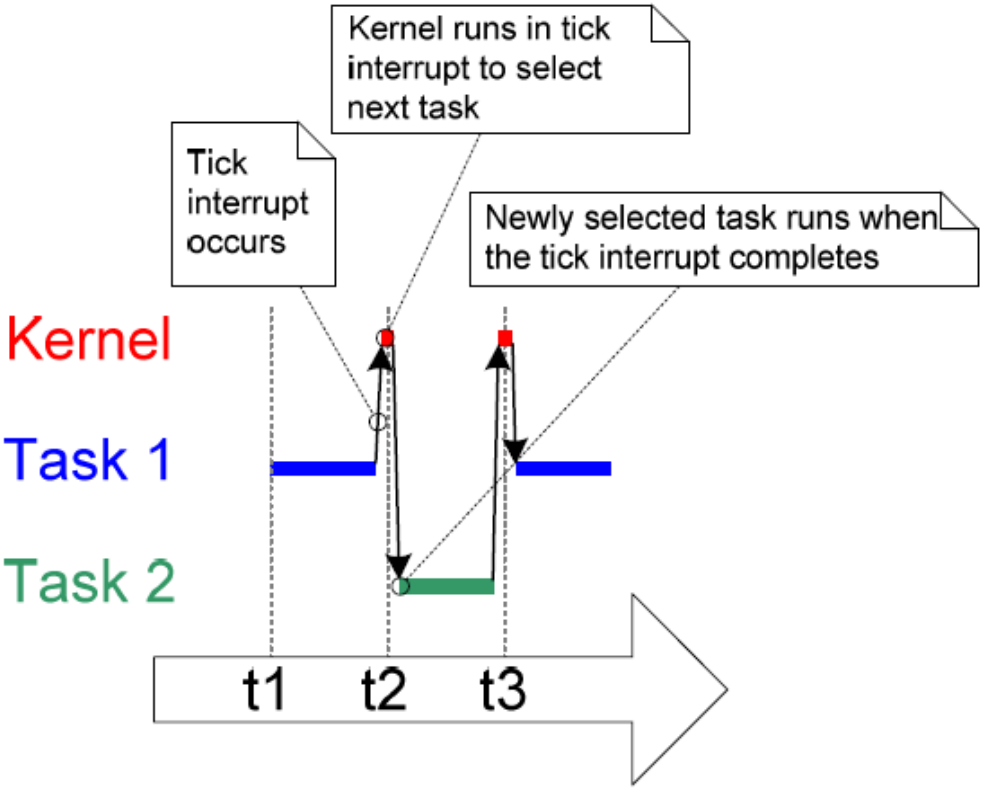
Planificador apropiativo



Planificador de tipo apropiativo (preemptive – priority scheduler).

- Es un planificador que cambia de tarea ***en función de eventos.***
- En este tipo de planificación cada tarea tiene una prioridad determinada. El planificador va a ejecutar la tarea de mayor prioridad que esté en condiciones de ejecutarse.
- Este tipo de planificador es el que va a minimizar la latencia, ya que los eventos suelen modelarse con tareas de prioridad elevada.

Planificador Apropiativo.



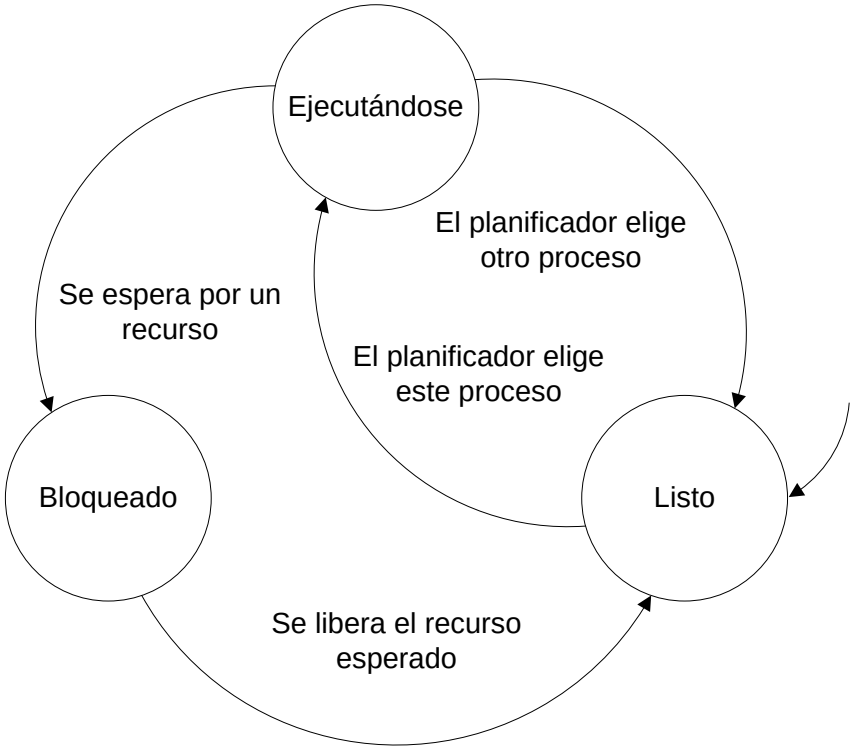
Planificadores compuestos

- Los modelos de planificación no son excluyentes entre sí.
- La mayoría de los planificadores toman enfoques compuestos de los modelos planteados, por ejemplo, FreeRTOS es apropiativo, pero dentro de la misma prioridad de tareas usa una planificación TS.

¿Quién escribe el Planificador?

- El planificador va a ser provisto por el proveedor del RTOS. Usualmente se obtiene el planificador y código para el manejo de hardware de uso común.
- Algunos RTOS:
  - FreeRTOS.
  - uC/OS-II
  - uCLinux
  - QNX
  - VxWorks
  - LynxOS

Tareas. Tarea Bloqueada.



- Anteriormente se mencionó que una tarea puede estar en dos posibles estados. Ejecutándose o no.
- Una tarea en ejecución es la que está usando el procesador.
- Las tareas que no están ejecutando se van a dividir en dos:
  - Listas. Son las que están en condiciones de ejecutarse.
  - Bloqueadas. Están esperando algún recurso y no serán planificadas hasta que el recurso se libere.

¿Si están todas las tareas bloqueadas, que pasa?

- Puede darse el caso de que todas las tareas se encuentren bloqueadas en un determinado punto del sistema.

- Como los RTOS deben estar ejecutando tareas todo el tiempo, el RTOS define una tarea de prioridad mínima (la tarea ociosa o “idle task”) que se va a ejecutar cuando todas las otras tareas estén bloqueadas.

Espera Activa.

```
void EsperaAct(void *param)
{
    int i=100;
    while(1)
    {
        haceralgo();
        while(--i);
    }
}

void EsperaAPI(void *param)
{
    int ticks = 100;
    while(1)
    {
        haceralgo();
        Waitfor(ticks);
    }
}
```

- Para que una tarea se bloqueé es necesario utilizar alguna de las funciones del RTOS para provocar que no sea planificada.
- En este ejemplo se muestra una espera activa (while) donde el RTOS “no se entera” que la tarea debe dejar de planificarse, mientras que en el ejemplo de abajo se llama a una API (de un RTOS ficticio) que genera el bloqueo de la tarea.
- Deben evitarse las esperas activas.

Ventajas y Desventajas.

- El RTOS provee abstracciones para el manejo de puertos a través de la API del RTOS.
- Permite un mejor manejo de la complejidad al tener un nivel de abstracción mayor que los sistemas FB.
- Facilita la portabilidad de código. Ya que el manejo de tiempo corre por parte del RTOS y el manejo de hardware puede abstraerse con el modelado de drivers.
- Parte del tiempo del procesador lo va a usar el planificador para el cambio de tareas.
- Se necesita bastante hardware del procesador para correr el RTOS. (excepciones por software, interrupciones, modos privilegiados y timers).

FreeRTOS

- FreeRTOS es un sistema operativo en tiempo real de código abierto (http://www.freertos.org/)
- Está portado a muchas arquitecturas.
- No es necesario pagar licencias por equipo.
- En genera poco consumo de memoria de programa (en STM32F103C8 16 Kb de memoria de programa HAL+FreeRTOS) y unos 8Kb de RAM (que es bastante para un micro de 20Kb de RAM)
- Está portado a muchos Cortex-M (LPC1769, STM32F103, STM32F401, etc.).

Definiendo tareas en FreeRTOS.

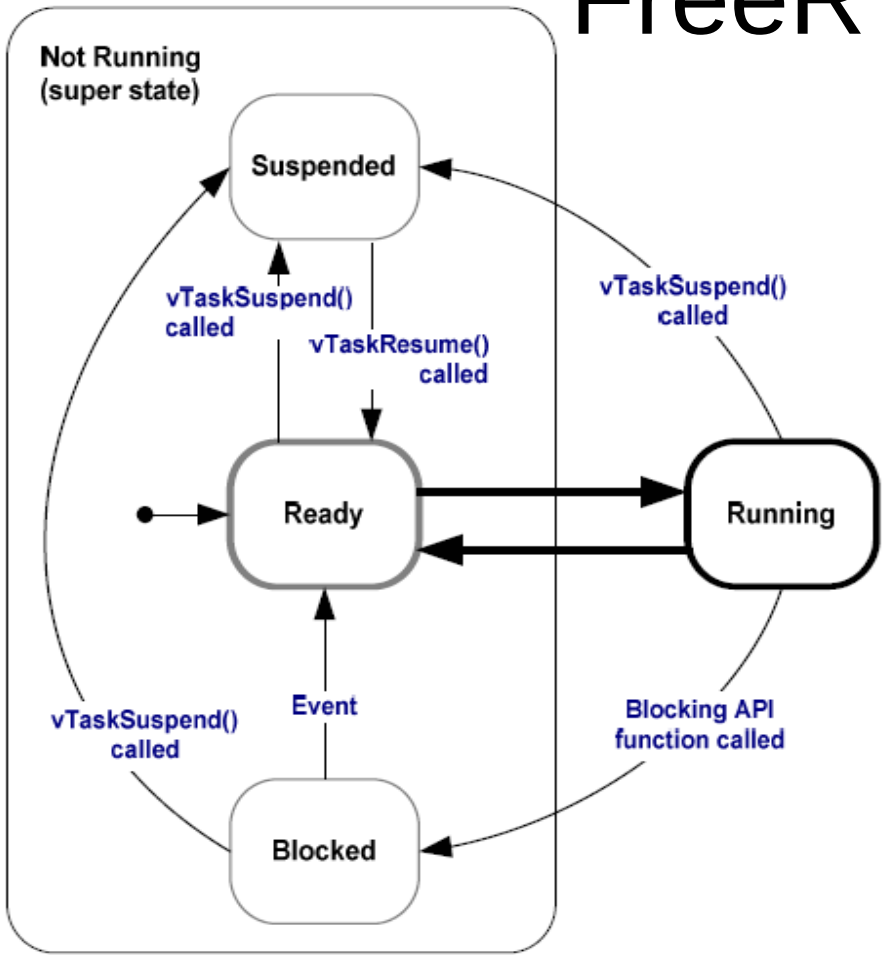
- FreeRTOS define a cada proceso como tarea.
- El código de una Tarea en FreeRTOS se define como una función en C como la de la figura.
- Una Tarea de FreeRTOS ***nunca*** debe retornar.

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance
of a task created using this function will have its own copy of the
iVariableExample variable.  This would not be true if the variable was
declared static - in which case only one copy of the variable would exist
and this copy would be shared by each created instance of the task. */
int iVariableExample = 0;

    /* A task will normally be implemented as in infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop
    then the task must be deleted before reaching the end of this function.
    The NULL parameter passed to the vTaskDelete() function indicates that
    the task to be deleted is the calling (this) task. */
    vTaskDelete( NULL );
}
```

Estados de las tareas en FreeRTOS.



- El planificador de RTOS es un planificador del tipo apropiativo (preemptive).
- ***El planificador va a ejecutar la tarea de mayor prioridad que se encuentre en condiciones de ejecutarse.***

Crear tareas en FreeRTOS.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                         );
```

Crear Tareas FreeRTOS. Ejemplo

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate debugging
                             only. */
                    1000,    /* Stack depth - most small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}
```

Cambiando prioridades.

```
void vTask1( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* This task will always run before Task2 as it is created with the higher
    priority.  Neither Task1 nor Task2 ever block so both will always be in either
    the Running or the Ready state.

    Query the priority at which this task is running - passing in NULL means
    "return my priority". */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\r\n" );

        /* Setting the Task2 priority above the Task1 priority will cause
        Task2 to immediately start running (as then Task2 will have the higher
        priority of the two created tasks).  Note the use of the handle to task
        2 (xTask2Handle) in the call to vTaskPrioritySet().  Listing 24 shows how
        the handle was obtained. */
        vPrintString( "About to raise the Task2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task1 will only run when it has a priority higher than Task2.
        Therefore, for this task to reach this point Task2 must already have
        executed and set its priority back down to below the priority of this
        task. */
    }
}
```

Demoras.

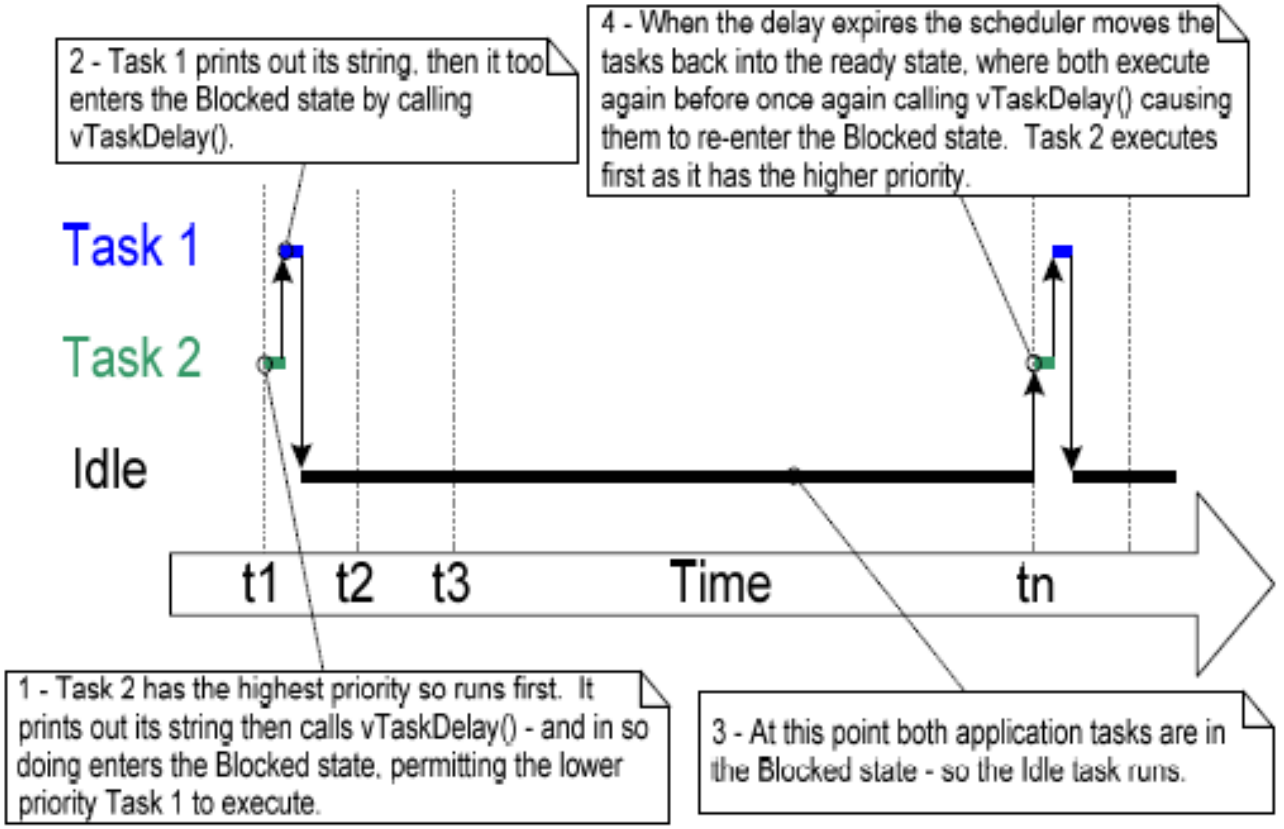
```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which
        places the task into the Blocked state until the delay period has expired.
        The delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );   aca se bloquea por un tiempo especificado
    }
}
```

Demoras. Carga del procesador



Demoras.

```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 10 milliseconds exactly. */
        vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
    }
}
```

Poniendo todo Junto. Led Parpadeante.

```
void tarea_led(void *p)
{
    for(;;)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
        vTaskDelay(LED_TICKS/portTICK_RATE_MS);
    }
}
```

```
xTaskCreate(tarea_led,                  //función con código de la tarea
            "led",                      //Nombre
            configMINIMAL_STACK_SIZE,   //STACK utilizado por la tarea
            NULL,                       //el parámetro.
            tskIDLE_PRIORITY+1,         //Prioridad (+1 de la tarea IDLE)
            NULL);                      //Puntero a la tarea
```

¿Qué pasa cuando se crea una tarea?

- Se busca en memoria un bloque que contenga el TCB (Task Control Block) y la pila.
- Se calculan los punteros a la pila.
- Se inicializa la pila
- Se inicializa el TCB, con su nombre y prioridad.
- Si es la primera de todas las tareas, se inicializan las listas de prioridades.
- Se incluye la tarea en la lista de prioridad correspondiente.
- Se marca la tarea como lista para correr

Inicializando el sistema.

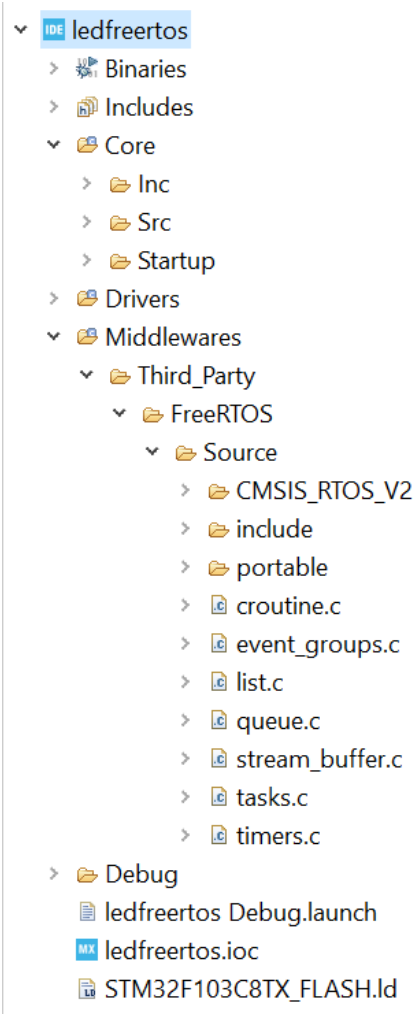
- Al llamar a la función vTaskStartScheduler(), para el código generado por el stm32cube dentro de osKernelStart(), van a suceder las siguientes cosas:
  - Se va a generar la tarea IDLE.
  - Se ponen los ticks de FreeRTOS a cero.
  - Se inicializan las interrupciones y el SysTick.
  - Se llama a la SVC 0 para cargar la primer tarea.

¿Cúanto ocupa en memoria (FreeRTOS + HAL)?

```
arm-none-eabi-objdump -h -S  ledfreertos.elf  > "ledfreertos.list"
arm-none-eabi-objcopy  -O binary  ledfreertos.elf  "ledfreertos.bin"
arm-none-eabi-size   ledfreertos.elf
   text    data     bss     dec     hex filename
  16184      24    7056   23264    5ae0 ledfreertos.elf
Finished building: default.size.stdout
```

- text representa la cantidad de memoria de programa (16184 bytes en este caso).
- data indica cantidad de memoria de lectura escritura no inicializada.
- bss indica la cantidad de memoria de lectura, escritura inicializada en cero (este programa utiliza un total de 7080 bytes de RAM).

Archivos de FreeRTOS



- La mayoría del código del FreeRTOS va a estar contenido en tasks.c, queue.c y list.c
- Las funciones dependientes de la arquitectura van a estar en port.c (dentro de portable)
- El archivo de configuración de FreeRTOS es FreeRTOSConfig.h (dentro de Core -> Inc)

Configurando FreeRTOS

```
#define configUSE_PREEMPTION                    1
#define configSUPPORT_STATIC_ALLOCATION         1
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      ( SystemCoreClock )
#define configTICK_RATE_HZ                      ((TickType_t)1000)
#define configMAX_PRIORITIES                    ( 3 )
#define configMINIMAL_STACK_SIZE                ((uint16_t)128)
#define configTOTAL_HEAP_SIZE                   ((size_t)3072)
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  0
#define configUSE_MUTEXES                       1
#define configQUEUE_REGISTRY_SIZE               8
#define configUSE_RECURSIVE_MUTEXES             1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
```

Parámetros de FreeRTOS.

- **`configUSE_PREEMPTION`**: Sí esta constante vale 1 el sistema operativo se va a comportar de manera apropiativa con tareas de diferente prioridad y round robin con tareas de igual prioridad. Sí vale 0 el sistema se va a comportar de manera cooperativa. **`taskYIELD()`**;
- **`configUSE_IDLE_HOOK`**. Sí vale 1 cuando el sistema está ocioso llama a la función **`void vApplicationIdleHook(void)`**. Sí vale 0 no genera esta llamada.

Llamada a la Función ociosa (Idle Hook).

- Usos más comunes:
  - Medir el uso del procesador.
  - Ejecutar procesos de muy baja prioridad de manera continua.
  - Poner el procesador en bajo consumo.
- Limitaciones:
  - Nunca debe bloquearse o suspenderse. Ya que no hay otra tarea a la que recurrir si la idle se bloquea.
  - Si el sistema usa **`vTaskDelete()`** debe retornar de manera lo antes posible ya que la idle task debe liberar los recursos del sistema

```
void vApplicationIdleHook(void)
{
     haciendonada++;
}
```

Parámetros de FreeRTOS

- **`configUSE_TICK_HOOK`** . Sí vale 1 se llama a la función **`void vApplicationTickHook(void)`** el contenido es esta función debe ser muy breve usar muy poca pila y no llamar funciones del FreeRTOS que no terminen en “FromISR” o “FROM_ISR”.
- **`configCPU_CLOCK_HZ`**. Frecuencia del core del chip, es necesaria para que el FreeRTOS lleve el tiempo de sistema de manera correcta.
- **`configTICK_RATE_HZ`**. Configura cada cuanto interrumpe el sistema operativo, mientras mayor sea esta frecuencia, las tareas que se ejecuten en round robin conmutaran entre sí más rápidamente, pero el SO usará más tiempo del procesador.

Parámetros del FreeRTOS

- **configMINIMAL_STACK_SIZE**. Define el tamaño de la pila para la tarea ociosa. Depende fuertemente de la arquitectura.
- **configTOTAL_HEAP_SIZE**. Define la cantidad de memoria en bytes que va a estar disponible para el sistema operativo.
- **configMAX_TASK_NAME_LEN**. Cuando se crea una tarea uno de los parámetros es el nombre de la tarea, esta constante define el largo máximo de estos nombres.
- **configUSE_TRACE_FACILITY**. Si está en 1 se activan todos las estructuras y funciones de ayuda a la visualización y depuración.

Parámetros del FreeRTOS

- **`configUSE_16_BIT_TICKS`**. Sí está en 1 la variable que cuenta los ticks del sistema se define de 16 bits. Sí está en 0 se define como una variable de 32 bits. Hay que tener en cuenta que (considerando una frecuencia de tick de 250Hz) en 16 bits se puede bloquear 262 segundos mientras que en 32 bits puede bloquear 17179869 segundos.

Otros parámetros.

```
/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       0
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_uxTaskGetStackHighWaterMark 1
```

Bibliografía.

- Sistemas Operativos. Diseño e Implementación. Andrew S. Tanenbaum.
- Using the FreeRTOS Real Time Kernel. NXP LPC17xx Edition. Richard Barry.
- VisualDSP++ 5.0 VDK (Kernel) User's Guide http://www.analog.com/static/imported-files/software_manuals/50_vdk_mn_rev_3.5.pdf
- Sistemas Empotrados en Tiempo Real. José Daniel Muñoz Frías. http://www.lulu.com/items/volume_67/1349000/1349482/8/print/ApuntesTR.pdf
- FreeRTOS http://www.freertos.org/
- Sistemas de Tiempo Real y Lenguajes de Programación. Alan Burns, Andy Wellings. Tercera Edición. Addison Wesley.