Introducción a los Sistemas operativos de tiempo Real

Alejandro Celery
acelery.utn@gmail.com

Parte 1

Un poco de contexto

Sistemas embebidos

- Los sistemas embebidos son plataformas con recursos muy limitados en comparación con una PC. Es por esto que generalmente no tienen un sistema operativo completo, sino sólo el subconjunto de estos que pueden manejar ventajosamente.
- En algunos casos el OS no es un programa en sí mismo sino que es un conjunto de funciones que se ejecutan solo en momentos determinados del programa.

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 programador de tareas. Su función es determinar qué tarea debe estar en ejecución a cada 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.

Más contexto

- Se llama contexto de ejecución al conjunto de recursos que identifican el estado de ejecución de una tarea:
  - IP (instruction pointer)
  - SP (stack pointer)
  - Registros del CPU
  - Contenido de la pila en uso

Cambiemos de contexto

- Cuando el scheduler 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 debe reservarse un bloque de memoria de datos para cada tarea. Esto limita la cantidad de tareas simultáneas del sistema (pueden sin embargo eliminarse y crearse nuevas tareas en tiempo de ejecución).

Cambiemos de contexto

- Estos 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.
- Los OS trabajan en dos modos:
  - En modo cooperativo, estos cambios solo ocurren cuando la tarea en ejecución relega voluntariamente el uso del CPU.
  - En cambio en modo preemptive el scheduler tiene la facultad de remover una tarea sin el consentimiento de la misma.
    - En este caso debe preveerse que algunas operaciones no deben ser interrumpidas, a estos pasajes del programa se los llama secciones críticas. Los OS permiten inhibir de este modo los cambios de contexto cuando es necesario.

¿Qué es un RTOS?

- Un RTOS es un sistema operativo ***de tiempo real***.
- Esto significa que 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. El objetivo del mismo es diferente de un OS convencional.
- 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.*
- Los STR suaves se suelen diseñar como si fueran STR duros. Luego se da mayor prioridad a la atención de los eventos con compromisos temporales estrictos.

¿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.
    - Esto implica que una aplicación mal diseñada puede fallar en la atención de eventos aún cuando se use 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 facil al programador el manejo del tiempo.
- Tarea Idle
  - Cuando ninguna de las tareas requiere del procesador, el sistema ejecuta una tarea llamada idle u ociosa. Esto me permite fácilmente contabilizar el nivel de ocupación del CPU, poner al mismo en modo de bajo consumo o correr cualquier tarea que pudiera ser de utilidad para el sistema cuando no debe atender ninguno de sus eventos.

¿Por que usar un RTOS?

- Multitarea
  - Simplifica sobremanera la programación de sistemas con varias tareas.
- Escalabalidad
  - 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.

La letra chica 1

- 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.
- 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
  - El tamaño de estas pilas suele ser configurable POR TAREA, lo cual mitiga este impacto.

La letra chica 2

- Finalmente, debe hacerse un análisis de tiempos, eventos y respuestas más cuidadoso. Al usar un RTOS ya no es el programador quién decide cuándo ejecutar cada tarea, sino el scheduler. Cometer un error en el uso de las reglas de ejecución de tareas puede llevar a que los eventos se procesen fuera del tiempo especificado o que no se procesen en lo absoluto.
- A la luz de los beneficios mencionados, en el caso de que la plataforma de hardware lo permita y el programador esté capacitado no hay razones para no usar un RTOS.

Tipo de tareas que vamos a implementar en una aplicación de tiempo real

- Tareas periódicas
  - Atienden eventos que ocurren constantemente y a una frecuencia determinada. P. ej, destellar un led.
- Tareas aperiódicas
  - Atienden eventos que no se sabe cuándo van a darse. Estas tareas están inactivas (bloqueadas) hasta que no ocurre el evento de interés. P. ej, una parada de emergencia.
- Tareas de procesamiento contínuo
  - 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.

Caso de estudio

- Tarea de la primera práctica
  - Destellar un led a una frecuencia determinada.
  - Muestrear un pulsador y reflejar su estado en un led (tarea periódica).
  - Contar pulsaciones y destellar un led esa misma cantidad de veces (tarea aperiódica).

Parte 2

Por qué FreeRTOS

Es de código abierto

- No hay costo de implementación.
- El código está ampliamente comentado, es muy sencillo verificar cómo hace su trabajo.
- Es relativamente sencillo de portar a plataformas nuevas.

Es fácil de implementar

- Hay mucha documentación disponible.
  - Libro de texto escrito por uno de los diseñadores del sistema.
  - Incluye una demostración de sus funciones para cada plataforma en la que está soportado.
- Nutrida comunidad de usuarios (ver http://sistemasembebidos.com.ar/foro)
- Hay una opción con soporte comercial.
- Hay una opción con certificación SIL-3 para sistemas de misión critica.

Está pensado para microcontroladores

- Está escrito mayormente en C, salvo las partes específicas de cada plataforma.
- Es liviano en tamaño de código y en uso de memoria RAM.

Primera mirada a FreeRTOS

- Es un mini kernel de tiempo real que puede trabajar en modos cooperativo, preemptive o mixto.
  - Mini kernel significa que provee los servicios mínimos e indispensables
- Permite compilar solo las funciones que se vayan a usar, acotando así su impacto en la memoria de código.
- Se puede definir y acotar en tiempo de compilación el uso de memoria de datos por parte del sistema.
- Ofrece funciones de temporización, de comunicación entre tareas, de sincronización entre tareas e interrupciones, y de definición de secciones críticas

Algunas opciones de configuración

- FreeRTOSConfig.h
  - configUSE_PREEMPTION
  - configCPU_CLOCK_HZ
  - configTICK_RATE_HZ
  - configMAX_PRIORITIES
  - configMINIMAL_STACK_SIZE
  - configTOTAL_HEAP_SIZE
  - configUSE_USE_MUTEXES
  - configUSE_CO_ROUTINES
  - #define INCLUDE_vTaskDelete 1

Parte 3

Tareas en FreeRTOS

Cómo es una tarea en FreeRTOS?

- Citando a Richard Barry: "Las tareas se implementan con funciones de C. Lo único especial que tienen es su prototipo, que debe devolver void y recibir un puntero a void como parámetro".
  - void vTareaEjemplo (void *parametros);
- Cada tarea tiene su propio punto de entrada, sección de inicialización y lazo de control.
- Las funciones de tarea (en adelante, tareas) en FreeRTOS NO DEBEN RETORNAR BAJO NINGÚN CONCEPTO. No deben incluir un return ni ejecutarse hasta la llave de cierre
  - Si una tarea deja de ser necesaria, puede eliminársela explícitamente.

Cómo es una tarea en FreeRTOS?

- 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
- IMPORTANTE: Cuando una tarea deja de estar en ejecución, las referencias a variables alojadas en su pila dejan de ser válidas.
- Al ser funciones de C, se aplican las mismas reglas de visibilidad de variables => se puede compartir memoria entre las tareas usando variables globales.
  - En este caso se debe cuidar el acceso a este recurso compartido.

Ejemplo 1: Dos tareas de igual prioridad se apropian del CPU, pero son administradas por el scheduler.

```
void vTask1( void *pvParameters )
{
// Punto de entrada - Seccion de inicializacion
const char *pcTaskName = "Task 1 is running\n";
volatile unsigned long ul;

    // Cuerpo de la tarea
    for( ;; )
    {
            vPrintString( pcTaskName );  // envía el string a la consola del IDE
            for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ ) { }
    }
    // La tarea NUNCA debe pasar de este punto, si lo hiciera debe ser
    eliminada
    vTaskDelete(NULL);
}
```

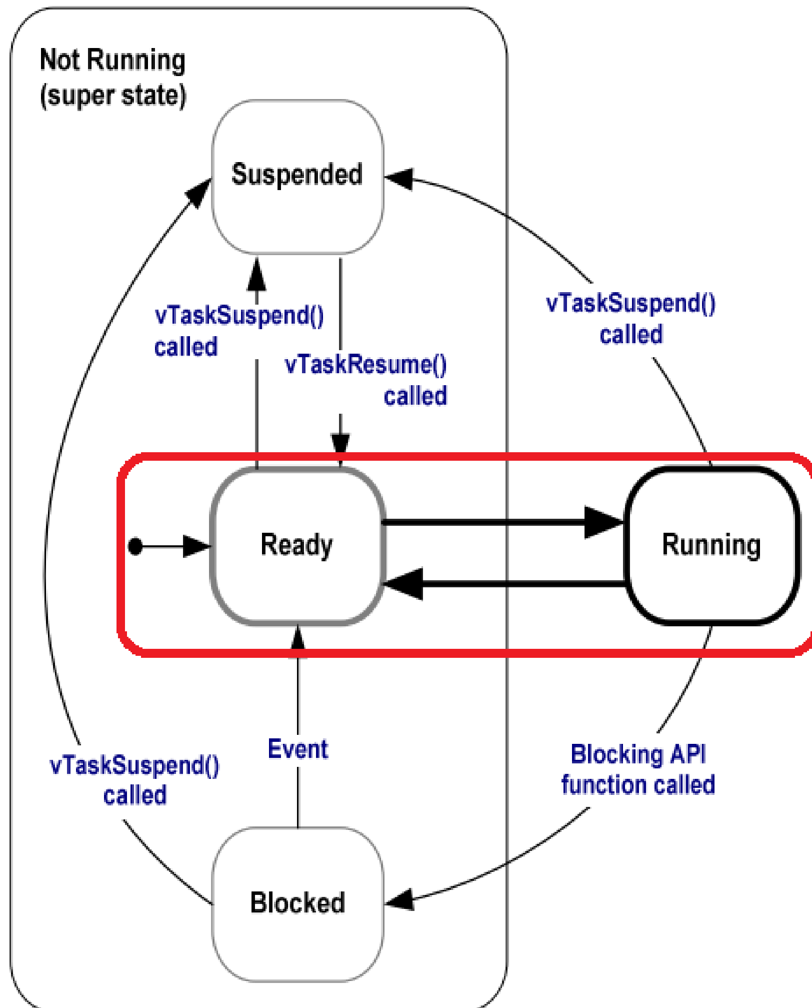
Ejemplo 1: Dos tareas de igual prioridad se apropian del CPU, pero son administradas por el scheduler.

```
void vTask2( void *pvParameters )
{
// Punto de entrada - Seccion de inicializacion
const char *pcTaskName = "Task 2 is running\n";
volatile unsigned long ul;

    // Cuerpo de la tarea
    for( ;; )
    {
            vPrintString( pcTaskName );  // envía el string a la consola del IDE
            for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ ) { }
    }
    // La tarea NUNCA debe pasar de este punto, si lo hiciera debe ser
    eliminada
    vTaskDelete(NULL);
}
```

Ejemplo 1: Dos tareas de igual prioridad se apropian del CPU, pero son administradas por el scheduler.

```
// Punto de entrada de la aplicación con FreeRTOS
int main( void )
{
    // Creo las dos tareas
    xTaskCreate( vTask1, "Task 1", 240, NULL, 1, NULL );
    xTaskCreate( vTask2, "Task 2", 240, NULL, 1, NULL );

    // Inicio el scheduler
    vTaskStartScheduler();

    // No se llega a este punto si no hubo problemas al iniciar el
    scheduler
    for( ;; );
    return 0;
}
```

Estados de las tareas en FreeRTOS

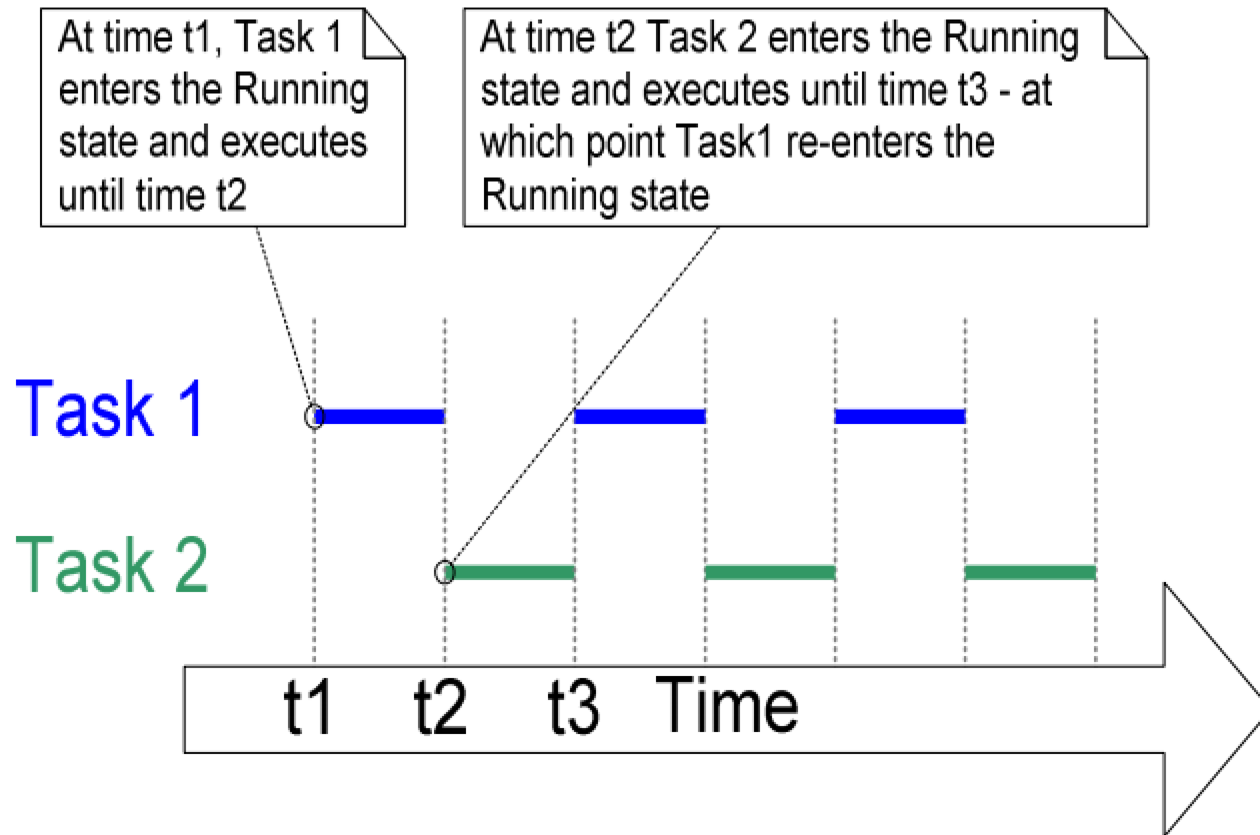


Transiciones de estados de las tareas del ejemplo 1.
Nótese que ninguna de las tareas cede el control del CPU, sin embargo el scheduler las conmuta al cabo de un tiempo.

Ejemplo 1: Demostración

- Veámoslo funcionando

Ejemplo 1: Diagrama de ejecución



Algoritmo de *scheduling*

- Como elige el scheduler la tarea a ejecutar? Usa un algoritmo llamado *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).
  - Esto simplifica el análisis de ejecución del sistema, a la vez que transfiere al programador la total responsabilidad del cumplimiento de los plazos.
- SIEMPRE ejecuta la tarea de mayor prioridad que esta en condiciones de ejecutarse.
  - Esto puede hacer que tareas de menor prioridad no reciban ningún tiempo de ejecución. A esto se le denomina “starvation” (hambreado) de la tarea.
  - Ya veremos cómo una aplicación deja de estar en condiciones de ejecutarse
- **Si la aplicación está en modo preemptive,** al cabo de un tiempo el scheduler va a retirar la tarea en ejecución **solo si hubiera una de igual prioridad en condiciones de ejecutarse.**

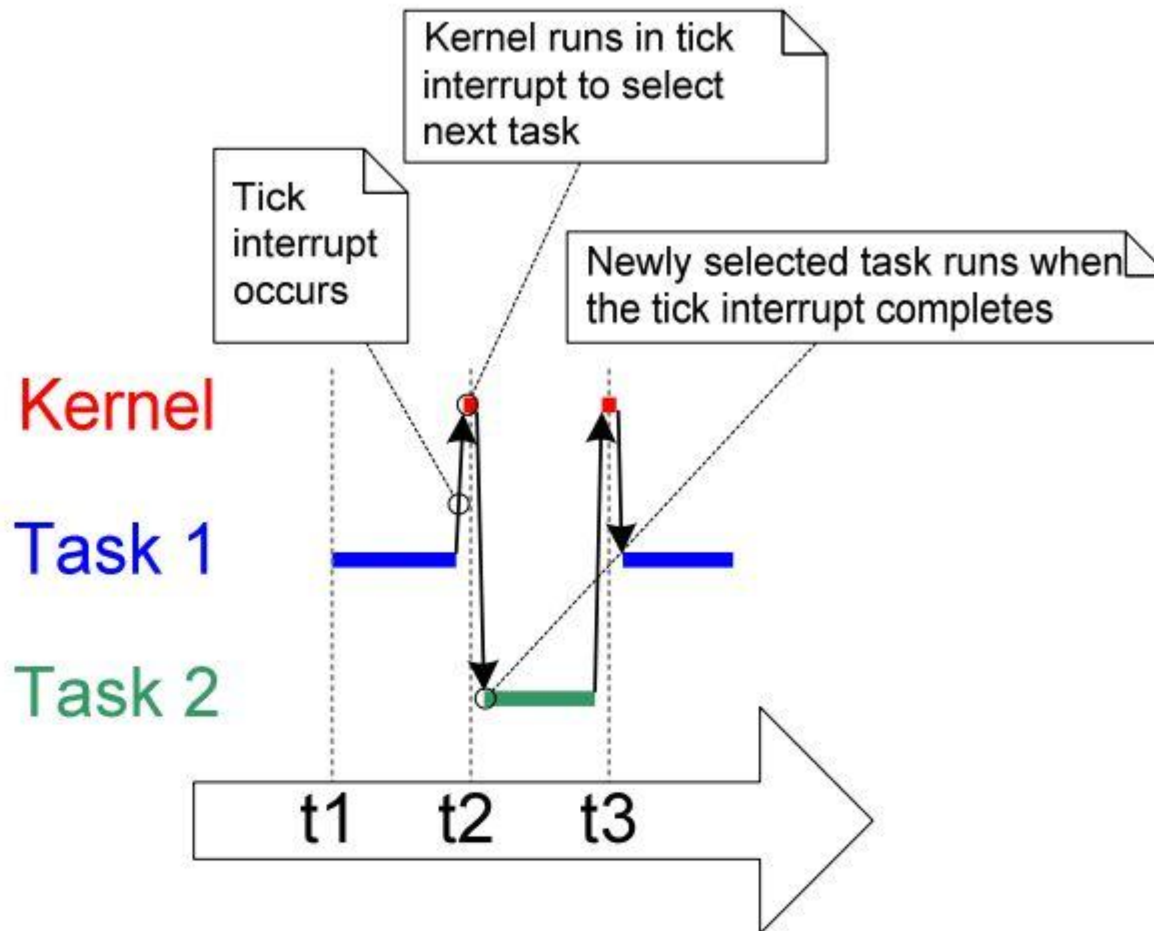
Más sobre el algoritmo de scheduling

- Recordar: "la tarea de mayor prioridad en condiciones de ejecutarse" (en estado **Ready)**
  - Si hubiere varias tareas de la misma prioridad en este estado, el kernel las ejecuta secuencialmente a todas.
    - Si hubiere alguna(s) de menor prioridad, no va(n) a recibir tiempo alguno hasta que todas las de mayor prioridad salgan del estado **Ready.**
  - Se define una frecuencia llamada configTICK_RATE_HZ. La inversa es el lapso que el scheduler asigna a cada tarea para ejecutarse antes de verificar si debe quitarla del estado **Running**.
  - La definición de esta frecuencia es un punto importante del diseño.

Resolución temporal del sistema

- Un valor muy bajo de TICK_RATE hace que el sistema sea lento para responder a los eventos temporales.
  - Si la tarea en ejecución no cede el control del CPU, la próxima tarea a ejecutarse debe esperar a que termine el lapso actual.
  - Por esto es que para mantener al sistema ágil cada tarea debe usar el CPU lo mínimo e indispensable, o más bien cederlo todas las veces que no lo necesite.
- 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.
- El TICK_RATE del sistema termina siendo un valor de compromiso entre estas dos cotas.
  - Recordar que la tarea idle ayuda a contabilizar el grado de utilización del CPU.

Detalle de la interrupción del scheduler



Consideraciones de diseño

- Siempre que varios procesos deban ocurrir en paralelo, son candidatos a ser tareas.
- Como primer criterio para asignar prioridades, se sugiere preguntarse que evento se debe atender y cuál puede esperar si ambos ocurren simultáneamente.
- Se debe evitar el hambreado de las tareas de menor prioridad.
  - Esto no es necesariamente un error, si la tarea de mayor prioridad tiene trabajo que hacer, es correcto que no le de paso a las demas.
- Es importante hacer un análisis detallado de los eventos, sus respuestas y los compromisos temporales antes de asignar las prioridades de ejecución.

Funciones para trabajar con tareas

- portBASE_TYPE xTaskCreate(...);
  - Se la puede llamar en cualquier momento.
  - Se pueden crear y destruir tareas durante la ejecución del programa.
  - Si una aplicación usa varias tareas pero no simultáneamente, puede crearlas a medida que las necesita y luego destruirlas para recuperar la memoria de datos.
  - Ver detalles en http://www.freertos.org/ -> API Reference -> Task Creation
- void vTaskStartScheduler(void);
  - Crea la tarea IDLE
  - Comienza la ejecución de las tareas
  - Ver detalles en http://www.freertos.org/ -> API Reference -> Kernel Control
- void vPrintString(const char *)
  - Envía un string a la consola del entorno LPCxPRESSO
  - La vamos a utilizar en los ejemplos.

Ejemplo 3: Mala asignación de prioridades

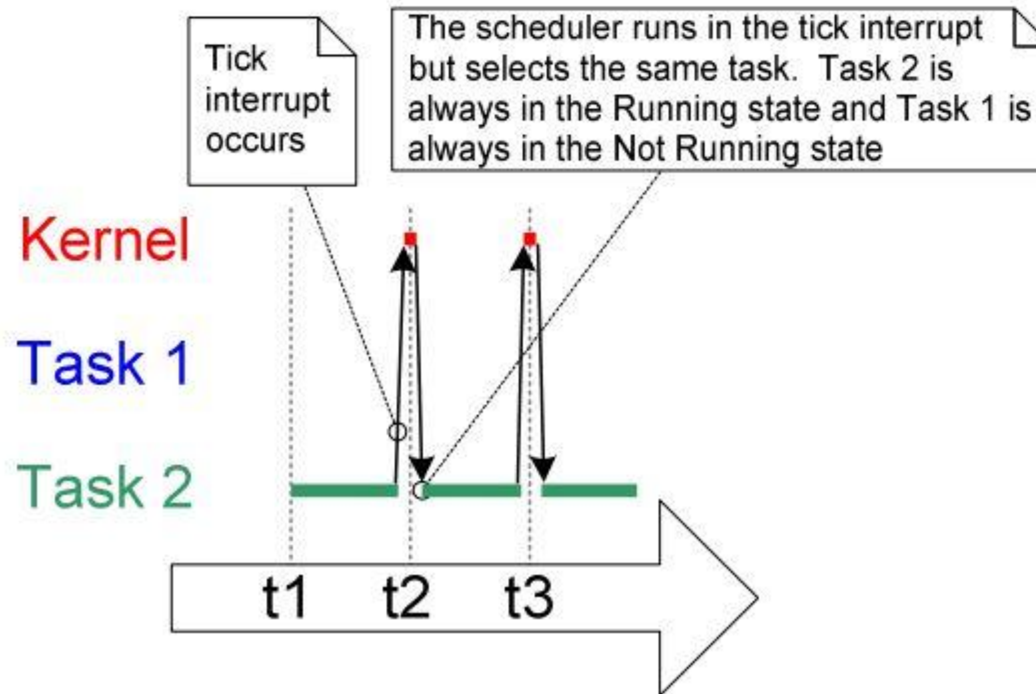
Las funciones de tarea son las mismas del ejemplo 1.

```
// Punto de entrada de la aplicación con FreeRTOS
int main( void )
{
    // Creo las dos tareas
    xTaskCreate( vTask1, "Task 1", 240, NULL, 1, NULL );
    xTaskCreate( vTask2, "Task 2", 240, NULL, 2, NULL );

    // Inicio el scheduler
    vTaskStartScheduler();

    // No se llega a este punto si no hubo problemas al iniciar el scheduler
    for( ;; );
    return 0;
}
```

Ejemplo 3: Demostración

- Veámoslo funcionando

Ejemplo 3: Diagrama de ejecución



Parte 4

Primera práctica con FreeRTOS

Tarea opcional para el hogar

- Tarea 1: Destellar un led a una frecuencia determinada
- Tarea 2: Seguir el estado de un pulsador con un led, incluyendo el anti-rebote eléctrico
- Tarea 3: Contar pulsaciones de un boton y luego de un segundo sin pulsaciones destellar un led esa misma cantidad de veces
- Se sugiere implementarlas usando el driver de GPIO provisto por NXP

Recursos de software 1

- void vTaskDelay(portTickType xTicksToDelay);
  - Produce una demora en la tarea que la llama. Cede el control del CPU mientras este tiempo no expira.
  - INCLUDE_vTaskDelay must be defined as 1 for this function to be available.
  - Uso:

```
vTaskDelay (500 / portTICK_RATE_MS);
```

Recursos de software 2

- void vTaskDelayUntil(portTickType *pxPreviousWakeTime, portTickType xTimeIncrement);
  - INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available.
  - Asegura un tiempo constante entre sucesivos llamados a esta función.
  - Cada vez que se la llama actualiza *pxPreviousWakeTime
  - Uso:

```
portTickType xLastWakeTime;
xLastWakeTime = xTaskGetTickCount();
for( ;; ) {
    vTaskDelayUntil( &xLastWakeTime, xFrequency );
}
```

Recursos de hardware

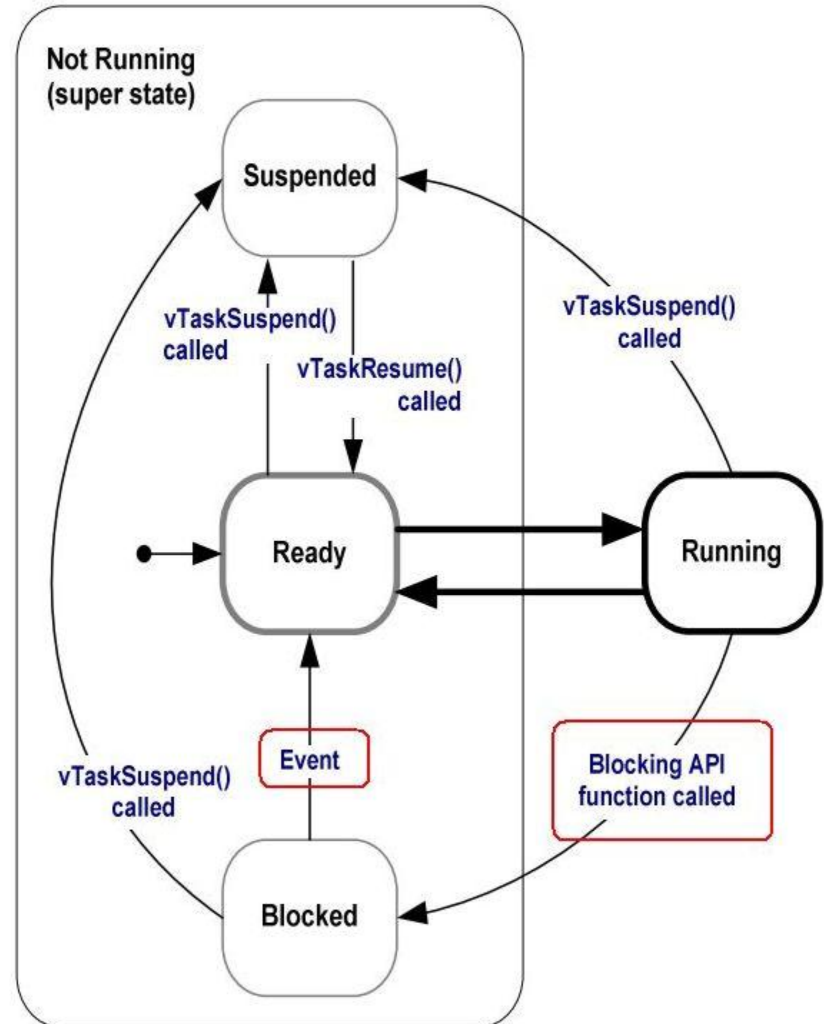
- Se pueden usar leds y pulsadores del base board
- Se pueden insertar el target board en un protoboard y agregarle pulsadores y leds
  - Cuidado, las IO del LPC manejan hasta 4mA (simétrico)
  - Se sugiere un buffer tipo ULN2003 para manejar los LEDs
- Las entradas pueden configurarse con pull-up interno, de este modo solo debe ponerse un pulsador conectado a GND
  - Usando una IO como GND y limitándose al LED2 del target board se pueden hacer las prácticas con solo esta placa y un botón soldado sobre el área de prototipado que incluye.
- El target board da una salida de 3,3V
  - Toma alimentación de un regulador de 500mA
  - Se sugiere no tomar más de 300mA de esta salida

Parte 5

Más estados de las tareas

Más estados de las tareas

- Hasta acá solo vimos tareas en los estados Running y Ready
- Para que no haya *starvation* de las tareas menos prioritarias del sistema, es necesario que las más prioritarias dejen el estado ready
- Cuando una tarea deja de estar en condiciones de ejecutar, pasó a alguno de los dos estados bloqueados: **Blocked** o **Suspended**



Funciones de temporización

- Ahora podemos ver con más detalle el efecto de llamar a vTaskDelay() y vTaskDelayUntil()
- Ponen a la tarea en el estado **Blocked** hasta que expira la cantidad de ticks indicada como parámetro
  - El kernel maneja el tiempo en ticks, es por esto que se usa siempre la constante portTICK_RATE_MS
- Cada vez que expira un período de ejecución, el scheduler revisa si alguna tarea de mayor prioridad a la actual sale del estado blocked (pasa al estado ready)

Tareas periódicas

- Ahora podemos ver como ve el kernel una tarea periódica.
  - Pasa la mayor parte de su tiempo en el estado *blocked* hasta que expira su tiempo de bloqueo, en este momento pasa al estado *ready*.
- En FreeRTOS se implementan mediante llamadas a vTaskDelayUntil().

Parte 6

Sincronización entre tareas y eventos

Tareas aperiódicas

- Acabamos de ver como una tarea puede permanecer bloqueada durante un tiempo fijo.
- Ahora nos interesa que permanezca bloqueada hasta que ocurra un evento determinado (interno o externo).
- La situación es la misma, queremos que la tarea ceda el CPU a las de menor prioridad, ya que no va a estar haciendo nada.
- Para esto existe el concepto de **semáforo**.

Qué es un semáforo?

- Un semáforo en informática representa el mismo concepto que un semáforo vial.
- Su función es restringir el acceso a una sección particular del programa (de la misma manera que uno vial no me permite cruzar la calle cuando están pasando autos).
- En ambos casos, nadie me obliga a consultar el semáforo antes de ingresar a la sección crítica. Sin embargo, los peligros de ignorar un semáforo son bien conocidos.
  - *El uso de semáforos es una convención a la que deben adherir todas las tareas involucradas.*

Cómo se usan en un RTOS?

- El uso que le vamos a dar es el de impedir que una tarea ingrese a cierta parte del código hasta que ocurra el evento esperado.
- Los semáforos tienen dos operaciones asociadas:
  - Tomar el semáforo (sería equivalente a ponerlo en rojo)
  - Dar el semáforo (sería equivalente a ponerlo en verde)
- Es responsabilidad del programador tanto tratar de tomar el semáforo antes de acceder a la sección crítica como darlo al salir de la misma

Como herramienta de sincronización

- Vamos viendo que una tarea que debe esperar un evento, lo único que debe hacer es tomar un semáforo que no esté disponible.
- Luego la tarea o interrupción que descubre o genera el evento tan solo debe liberar el semáforo
- En ambas operaciones el scheduler verifica cuál es la tarea de mayor prioridad en condiciones de ejecutar, bloqueando o desbloqueando según el estado del semáforo y las prioridades de las tareas.
- Este semáforo particular que solo puede tomarse y darse una vez, se llama ***semáforo binario.***

Como herramienta de sincronización

- Dependiendo de la tarea, el comportamiento puede ser:
  - Volver a bloquear luego de atender el evento. Para esto se vuelve a tomar el semáforo.
    - Este es el caso típico de los manejadores de eventos aperiódicos.
    - En este caso el “dado” del semáforo se puede pensar como un “vale por una única ejecución”, que luego es descartado.
  - Continuar su ejecución una vez dada la condición que se esperaba. En este caso NO se vuelve a tomar el semáforo.
    - Se da cuando hay una espera por única vez, como ser la inicialización de un periférico u otro recurso.

Acotando la espera de eventos

- Puede ocurrir que el evento que estoy esperando no llegue nunca. Según la aplicación esto puede ser normal o un error.
- Para los casos en que se quiera acotar la espera del evento a un tiempo finito, se usa un tiempo máximo de bloqueo o BlockTime.
- Todas las funciones de FreeRTOS susceptibles de bloquear a la tarea que las llama especifican un tiempo de bloqueo.
- En el caso de un bloqueo acotado, hay que verificar el código de retorno de la llamada para determinar si se produjo el evento esperado o si expiró el blockTime.

Acotando la espera de eventos

- Para esperar un tiempo finito, usar un blockTime no cero.
  - 500 / portTICK_RATE_MS
- Para esperar indefinidamente
  - portMAX_DELAY
  - (debe estar #define taskSuspend 1 en FreeRTOSConfig.h)
- Para que la operación no bloquee
  - usar 0 como blockTime
  - devuelve pdPASS / pdFALSE si el evento no ocurrió
  - Se debe chequear siempre el valor de retorno de la llamada, para verificar si ocurrió el evento o si expiró el tiempo de bloqueo.

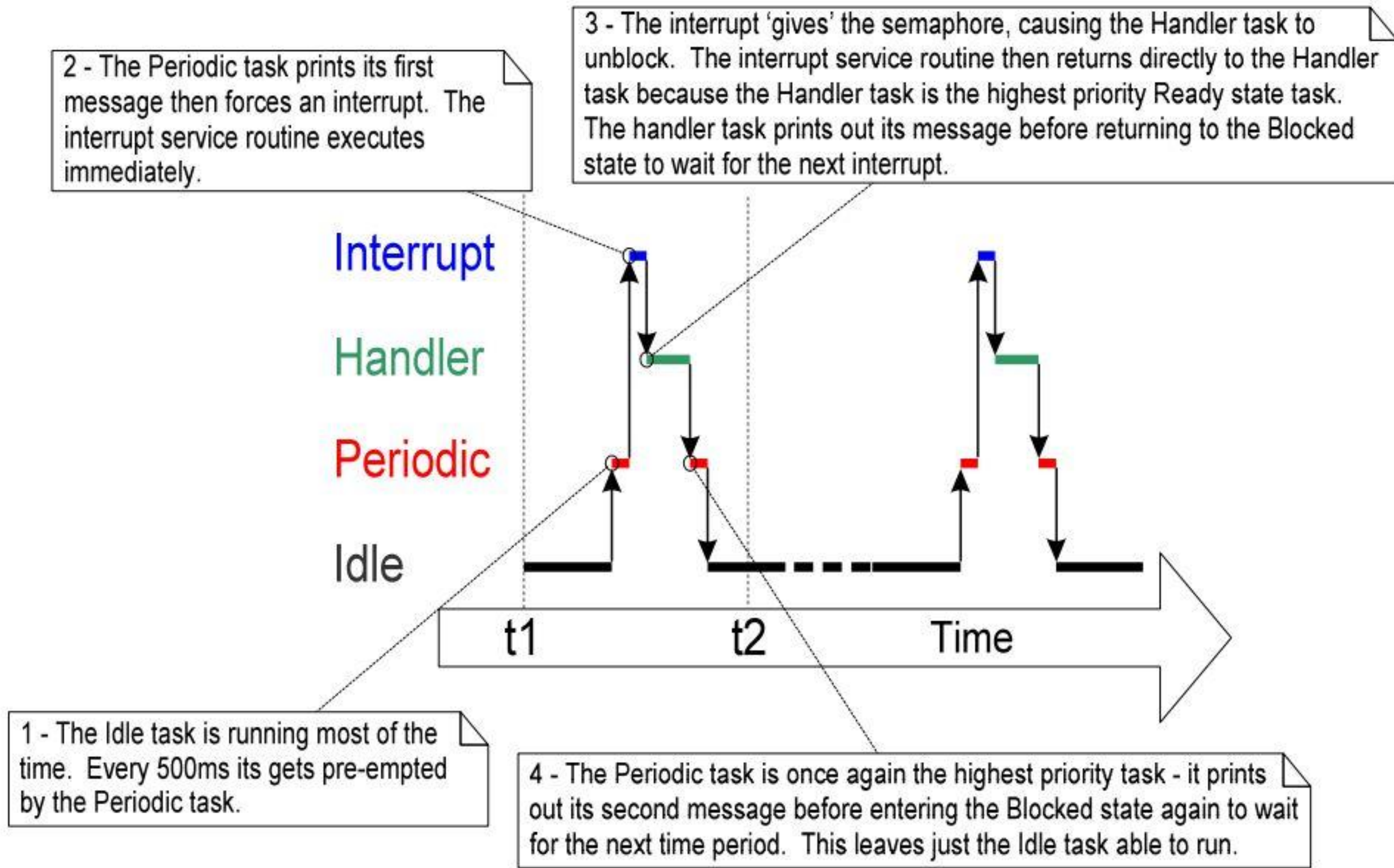
Ejemplo 12: Desbloquear una tarea mediante un semáforo

- Para simplificar el código, se usa una tarea periódica como generadora de eventos.
  - Cada vez que "de" el semáforo genera un pulso de 500ms en el LED2 del target board.
- El manejo del evento consiste en enviar un mensaje al entorno.

Ejemplo 12: Demostración

- Veámoslo funcionando

Ejemplo 12: Diagrama de ejecución



API de semáforos de FreeRTOS

- vSemaphoreCreateBinary(xSemaphoreHandle xSem)
- xSemaphoreTake(xSemaphoreHandle xSem, portTickType xBlockTime)
- xSemaphoreGive(xSemaphoreHandle xSem)

- NOTA: Están implementadas como macros, no como funciones => reciben los parámetros por valor, no por referencia (más adelante veremos por qué)

- xSemaphoreHandle es un typedef, debe usarse este tipo para trabajar con semáforos.

Cuestiones de implementación

- Los semáforos son objetos del kernel. Se toma memoria del sistema por cada uno que se crea.
- El scheduler revisa si debe cambiar el contexto de ejecución cada vez que se opera sobre un semáforo => se emplea tiempo de CPU.
- Si el manejo del evento es inmediato o diferido dependerá de las prioridades de las tareas y el estado de las mismas.
  - En el mejor de los casos, darle ejecución a una tarea bloqueada causa un cambio de contexto. Este tiempo debe tenerse en cuenta a la hora de analizar los requisitos temporales del sistema.
- Deben ser vistos por todas las tareas del sistema => se crean en forma global (y antes de iniciar el scheduler)
- Usar con racionalidad
- Los semáforos son útiles siempre que hay dependencias entre tareas y eventos o entre tareas solamente.
- Nunca se debe tomar un semáforo dentro de una interrupción!

Checkpoint!

- Los conceptos de sincronización vistos hasta ahora se repiten en los próximos capítulos.
- Preguntas hasta ahora?

Parte 7

Manejo de colas para
intercambio de datos

Intercambio de datos entre las tareas

- Vimos que las tareas son funciones de C, aplican las mismas reglas de visibilidad.
- También vimos que cuando una tarea sale de ejecución dejan de ser válidas las referencias a su pila.
- Me queda como mecanismo de intercambio el uso de variables globales, pero se presentan dos problemas
  - Cuidar el acceso al recurso
  - También se presenta el caso de tareas aperiódicas que esperan una lectura / escritura de un dato para procesarlo.
- Surge entonces la necesidad de un mecanismo de comunicación entre tareas sincronizado.

Intercambio de datos entre las tareas

- Para modularizar la implementación de un sistema, se suelen seprar los productores de datos de sus consumidores.
  - Una tarea produce datos
  - Otra tarea los consume
    - Mientras no reciba datos, esta tarea queda suspendida
- Cuando los datos se producen más rápido que lo que se los consume (momentáneamente), se necesita un almacenamiento temporario de los mismos.
  - Si la velocidad media de producción supera a la de consumo, no se puede hacer nada.
- A este esquema se lo llama patrón productor – consumidor.

Intercambio de datos entre las tareas

- Una primera aproximación al problema sería usar semáforos, aunque agregaría una cierta complejidad a la aplicación.
- Una solución más simple es usar colas (queues) del RTOS
- Son visibles por todas las tareas (deben ser creadas en forma global)
- Incorporan el mecanismo de sincronización
- De la misma manera que un semáforo, se puede bloquear al leer / escribir datos de una cola.

Cómo se las crea?

- Se crean mediante la función xQueueCreate()
- Trabajan con cualquier tipo de datos, no solo los nativos.
- Los mensajes se pasan en la cola POR COPIA
  - Se debe dimensionar la cantidad de elementos de la cola y el tamaño de los mismos cuando se la crea. Estos parámetros no pueden cambiarse luego.
- Son objetos del OS, toman su memoria del heap del sistema.
- Se la debe crear antes de iniciar el scheduler.
- Se la identifica mediante un xQueueHandle devuelto por la llamada a xQueueCreate.

Operaciones disponibles con colas

- Cualquier tarea puede leer y escribir en ellas.
- Leer datos de una cola:
  - xQueueReceive(): Quita el dato de la cola.
  - xQueuePeek(): No quita el dato de la cola.
- Escribir datos en una cola:
  - xQueueSendToBack(): La escritura normal a una cola.
  - xQueueSendToFront(): Pone el mensaje en la posición de salida. Sería como apilar el dato.
- Consultar los mensajes disponibles en las colas.
  - uxQueMessagesWaiting().

Operaciones disponibles con colas

- Las operaciones de cola pueden bloquear.
  - Intentar leer de una cola vacía
  - Intenar escribir a una cola llena
- Por esto, se les puede asignar un blockTime igual que a un semáforo. En este caso también debe chequearse el valor de retorno de la operación.
- Con este detalle se puede ver ahora que un semáforo binario no es más que una cola de un solo elemento.
  - Tomar el semaforo es equivalente a escribir en la cola.
  - Dar el semáforo es equivalente a leer de la misma.
  - Por esta razón es que está implementado con macros.

Ejemplo 10: Lectura sincronizada

- Presenta el caso típico de un driver de salida.
- Una tarea asociada a un periférico de salida espera que el sistema produzca datos para enviar al mismo.
- Mientras no los haya, se mantiene bloqueada.
  - En este caso no tiene sentido especificar un tiempo de bloqueo.
- Este caso particular da prioridad al procesamiento de los datos => la cola nunca va a tener más de un dato almacenado.
- Cabe analizar el caso de cola llena.

Ejemplo10: Tarea consumidora de datos

```
static void vReceiverTask( void *pvParameters )
{
    // Seccion de inicializacion
    long lReceivedValue;
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    // Cuerpo de la tarea
    for( ;; )
    {
            xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
            vPrintStringAndNumber( "Received = ", lReceivedValue );
    }
}
```

Ejemplo10: Tarea productora de datos

```
static void vSenderTask( void *pvParameters )
{
    // Seccion de incializacion
    long lValueToSend;
    portBASE_TYPE xStatus;
    lValueToSend = ( long ) pvParameters;

    for( ;; )
    // Cuerpo de la tarea
    {
            xQueueSendToBack( xQueue, &lValueToSend, 0 );
            taskYIELD();  // Cedo el resto de mi timeslice
    }
}
```

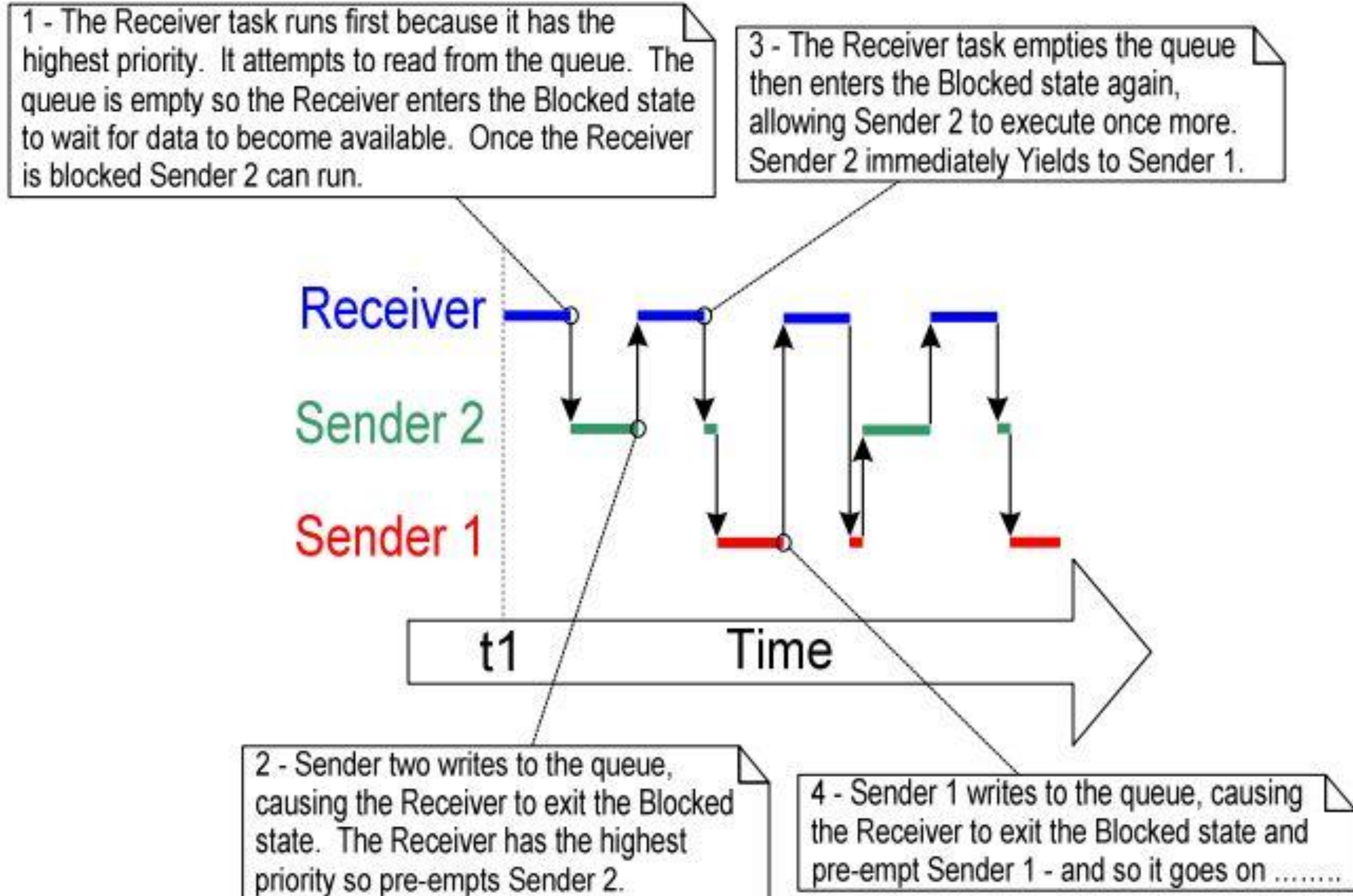
Ejemplo10: Funcion main()

```
xQueueHandle xQueue; // Variable para referenciar a la cola
int main( void )
{
    xQueue = xQueueCreate( 5, sizeof( long ) );
    xTaskCreate( vSenderTask, "Sender1", 240, ( void * ) 100, 1, NULL );
    xTaskCreate( vSenderTask, "Sender2", 240, ( void * ) 200, 1, NULL );
    xTaskCreate( vReceiverTask, "Receiver", 240, NULL, 2, NULL );
    vTaskStartScheduler();
    for( ;; );
    return 0;
}
```

Ejemplo 10: Demostración

- Veámoslo funcionando

Ejemplo 10: Diagrama de ejecución



Cuestiones de diseño

- La cantidad de elementos de la cola debe contemplar el peor caso de uso posible.
- Si los elementos son muy grandes es preferible usar la cola para enviar datos por referencia y no por copia. En este caso debe usarse memoria global. Y debe preveerse que los datos apuntados se mantengan válidos hasta que sean consumidos.
- Recordar que las operaciones de cola pueden llevar un tiempo de bloqueo, en este caso debe chequearse el valor de retorno de la llamada.
- Al analizar las prioridades de las tareas que emplearán la cola, contemplar los casos de cola llena, cola vacía y si el envío los datos encolados debe interrumpir a la generación de los mismos.

Parte 8

Gestión de recursos

Problemáticas de la concurrencia

- Cuando varios procesos ejecutan simultáneamente, puede ocurrir que quieran usar un mismo recurso al mismo tiempo. En realidad, el acceso no es simultáneo sino que ocurre un cambio de contexto entre que el primer proceso empieza y termina de modificar el recurso.
  - Dos tareas quieren escribir a una UART a la vez, los mensajes se entremezclan.
  - Un proceso modifica una variable de configuración mientras otro la lee, no se sabe en qué estado va a quedar el recurso.
  - Operaciones de read-modify-write (x 1= (1<<7));
- Estos problemas se evitan serializando el acceso al recurso.

Exclusión mutua

- A este serializado de procesos se le llama exclusión mutua (mutex).
- FreeRTOS brinda varias alternativas para implementar exclusión mutua.

Alternativa 1: Deshabilitar interrupciones

- Si se deshabilitan las interrupciones, no puede ocurrir ningún cambio de contexto preemptive.
  - No hay manejo de SysTick (cuando expira un timeslice)
  - No se ejecuta ningún código que pueda iniciar un cambio de contexto.
- De este modo, el sistema se vuelve momentáneamente cooperativo y puedo terminar la operación crítica.
- Además de estar protegido contra otras tareas, estoy protegido contra las funciones de interrupción.
- Pero en el interim el sistema pierde la capacidad de responder a eventos, debe usarse con cuidado.

Alternativa 1: Deshabilitar interrupciones

- FreeRTOS provee las macros taskENTER_CRITICAL() y taskEXIT_CRITICAL()
- Su uso debe estar apareado (un exit por cada enter).
- Se anidan correctamente. Esto significa que si se producen N taskENTER_CRITICAL(), el sistema no rehabilita las interrupciones hasta contar N taskEXIT_CRITICAL().
- Si bien es la operación más rápida des/hacer, es la que más perturba al sistema, debe reservarse para operaciones muy breves.

Alternativa 2: Supender el scheduler

- Llamando a la funcion vTaskSuspendAll() se suspenden los cambios de contexto mientras que se mantienen habilitadas las interrupciones.
  - El sistema sigue registrando los eventos externos.
  - La operación atómica se lleva a cabo sin problemas.
- Se debe poner el correspondiente xTaskResumeAll() para reanudar al scheduler.
  - Estas operaciones se anidan correctamente.
- Este recurso debe usarse con cuidado, ya que todos los eventos del sistema que se pierdan se atenderán al momento de la reanudación (semáforos, colas, etc).

Alternativa 3: Subir la prioridad de la tarea

- FreeRTOS me permite modificar la prioridad de las tareas en tiempo de ejecución.
- Se puede aprovechar esto para modificar la prioridad de la tarea que está entrando a la sección crítica.
- De este modo las interrupciones quedan habilitadas y puedo definir un umbral de tareas que no van a interrumpir a la tarea crítica.
  - unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
  - void vTaskPrioritySet(xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority);
  - xTaskHandle es un parámetro pasado a xTaskCreate, me permite hacer referencia a una tarea.

Alternativa 4: Usar mutex del sistema

- Un mutex es una estructura del sistema muy similar a un semáforo binario.
- La diferencia es conceptual. Cuando varias tareas quieren usar un recurso, se guarda al mismo con un mutex. Puede pensárselo como una llave de acceso al recurso.
- La única tarea que puede accederlo es la que tiene tomado el mutex.
- Es imprescindible que al terminar de usar el recurso devuelva el mutex.
- Recordar que el uso del mutex es una convención. El OS no impide, a ninguna tarea acceder al recurso "por las malas".

Uso de mutex

- mutex = xSemaphoreCreateMutex();
- xSemaphoreTake(mutex, portMAX_DELAY);
  - consoleprint(“Hola Mundo!”);
- xSemaphoreGive(mutex);

- Esta es una implementacion alternativa de la funcion vPrintString()
- Un mutex resulta la opción más sencilla de analizar, solo podría generar demoras a otras tareas que estuvieran compitiendo por elrecurso.

Alternativa 5: Delegar el manejo del recurso a una tarea driver

- Como se ha visto anteriormente, si varias tareas necesitan compartir el acceso a un único recurso, se puede escribir un driver de este.
- Esto traslada la complejidad del caso
  - Las tareas se vuelven más simples
  - Se agrega el trabajo de escribir el driver
- Permite un manejo flexible del recurso.
  - Un LCD alfanumérico con varias tareas enviando mensajes.
  - Una única UART compartida por varias tareas, se comportaría como un router ethernet.

Recursos limitados en cantidad

- Un servidor web puede servir múltiples páginas web concurrentemente.
- Esta concurrencia está limitada por la cantidad de tareas que el sistema puede crear.
- Existe para estos casos un tipo de semáforo llamado **counting** (contador) que puede darse y tomarse múltiples veces.
  - Se lo crea con un valor inicial igual a la cantidad de recursos disponibles.
  - Su cuenta disminuye una vez cada vez que se lo toma y aumenta una vez cuando se lo da.
  - Si la cuenta llega a 0, la tarea que no puede obtener un recurso pasa al estado Blocked.

Conteo de eventos

- Otro uso de los semáforos contadores es el de contar eventos.
- Su uso es el dual del anterior:
  - Se lo crea con un valor inicial 0
  - Se lo da una vez por cada ocurrencia del evento
  - Se lo descuenta una vez por cada vez que se lo maneja
  - El manejador del evento pasa al estado blocked cuando el semáforo vuelve a 0 (no quedan más eventos por manejar)

API de FreeRTOS para semáforos

- Los mutex y ambos tipos de semáforos se manejan con las mismas funciones give y take.
- Tienen diferentes funciones para ser creados:
  - vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)
  - xSemaphoreHandle xSemaphoreCreateCounting(uxMaxCount, uxInitialCount)
  - xSemaphoreHandle xSemaphoreCreateMutex(void)

Parte 9

Segunda práctica con FreeRTOS

Tarea opcional para el hogar

- Para todos los ejercicios se sugiere manejar los periféricos que vayan a ser usados en el proyecto.
- El sistema debe pasar en la tarea idle todo el tiempo posible.
- Ejercicio 1: Escribir una tarea que envíe un mensaje al IDE cuando otra tarea registra una pulsación de más de 500ms.
- Ejercicio 2: Contar pulsaciones de un botón y enviar el valor de cuenta a un perif. de salida mediante una cola.
- Ejercicio 3: Reimplementar la función vPrintString usando mutex.

Parte 10

Algunos problemas de los mutex

Recaudos a tomar al usar mutex

- Usar mutex como mecanismo de exclusión mutua es la alternativa más sencilla y que menos complica el análisis del sistema.
- Sin embargo no hay que perder de vista que cuando una tarea toma un mutex para acceder a un recurso se modifican las reglas de ejecución, recordar "la tarea mayor prioridad en condiciones de ejecutarse"
- Se deben preveer dos situaciones que pueden ocurrir:
  - La inversión de prioridades
  - El deadlock

Inversión de prioridades

- Pongamos un ejemplo:
  - Una tarea periódica de alta prioridad está bloqueada esperando su tiempo de ejecutar.
  - Mientras, una tarea de baja prioridad baja toma un mutex para acceder a un recurso protegido durante un tiempo prolongado.
  - La tarea de alta prioridad pasa al estado ready al expirar su tiempo de bloqueo e intenta acceder al recurso.
- En este caso, la tarea de alta prioridad debe esperar que se libere el mutex. En la práctica, está esperando a una tarea de baja prioridad, por esto se dice que se invirtieron las prioridades del sistema.

Inversión de prioridades

- Ahora imaginemos que una tercer tarea de prioridad intermedia pasa al estado Running durante otro tiempo prolongado.
- La tarea de alta prioridad está esperando a una tarea de baja prioridad que no va a liberar el mutex ya que no recibe tiempo de ejecución para terminar de usar el recurso!
- Para evitar esto los mutex de FreeRTOS incorporan un mecanismo llamado *priority inheritance*.
  - Cuando ocurre la inversión de prioridades, el scheduler eleva la prioridad de la tarea que tiene el mutex al nivel de *la tarea de mayor prioridad que está intentando tomarlo.*

Priority inheritance

- Este mecanismo está incorporado en los mutex de FreeRTOS, funciona siempre que se los usa.
- Esta es la única excepción al algoritmo *Fixed priority preemptive scheduling.*
- No soluciona la inversión de prioridades, pero acota la duración de la misma, lo que simplifica el análisis de tiempos del sistema.

Como se evita?

- Se debe tomar en cuenta al diseñar el sistema y evitar esta situación.
- Es conveniente delegar el manejo del recurso en un driver del mismo implementado con una cola, de modo que ninguna tarea se apropie del recurso.
- Para detectarlo, se manifiesta como una tarea que se ejecuta más tarde de lo previsto (pudiendo fallar su plazo) o que no ejecuta en absoluto.

Deadlock

- El deadlock ocurre cuando dos tareas están bloqueadas esperando un recurso que tiene la otra.
  - Más bien, cuando bloquean al tomar un mutex retenido por la otra tarea.
- Ninguna de las dos libera el mutex ya que está bloqueada esperando el otro.
- Finalmente, ninguna de las dos tareas recibe tiempo de ejecución.
- Un ejemplo sencillo es el típico intercambio de prisioneros en las películas, donde ninguno de los protagonistas quiere liberar su recurso hasta no recibir el del otro.

Como se evita?

- El deadlock es un problema de diseño, no de implementación.
- Se debe prever su posible ocurrencia al diseñar el sistema y tomar recaudos para que no ocurra.
- Puede darse para el caso de N tareas con una cadena de dependencias, no solo dos.
  - En http://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_filosofos se puede ver el problema en detalle y varias soluciones al mismo.
- Para detectar el deadlock, tener presente que no se bloquea el sistema entero sino solo las tareas que lo causan, el resto se ejecuta con normalidad.

Parte 11

Interrupciones y FreeRTOS

Tratamiento particular de las interrupciones

- Una ISR siempre debe ejecutar por completo.
- En el caso de un sistema multitarea, no debe forzarse un cambio de contexto dentro de una ISR.
- Para esto, todas las funciones de FreeRTOS que podrían generar un cambio de contexto tienen una variante *ISR-safe*, que es el mismo nombre de la función con el sufijo FromISR.
  - **xSemaphoreGiveFromISR**
  - **xQueueSendFromISR**
  - **xQueueReceiveFromISR**

Operaciones no permitidas en una ISR

- Las operaciones cuyo objetivo es justamente bloquear a la tarea no deben usarse en una ISR y no tienen una variante FromISR.
  - No hay xSemaphoreTakeFromISR
  - No hay xDelayFromISR
  - No hay taskYIELDFromISR

Cambios de contexto sincronizados a una ISR

- Las funciones FromISR reciben un parámetro adicional por referencia en el cual señalizan si debe hacerse un cambio de contexto.
  - xSemaphoreGiveFromISR(xSemaphoreHandle Semaphore, signed portBASE_TYPE pxHigherPriorityTaskWoken);
- El fin de la ISR se hace llamando a la macro portEND_SWITCHING_ISR() con este valor.
  - NOTA: Esta macro es la versión *ISR-safe* de la macro taskYIELD. El nombre varía según el port de FreeRTOS usado.

Cambios de contexto sincronizados a una ISR

- De este modo el scheduler sabe si debe permanecer en el mismo contexto o volver de la interrupción *a una tarea distinta.*
- En este último caso, se maneja el evento en una tarea sincronizada a la interrupción.
  - Recordar que esto agrega un tiempo extra al tiempo de respuesta al evento.
- En sistemas sin RTOS se suele hacer de la misma manera, donde las ISR tan solo modifican algún flag global y el programa principal es el encargado de manejar el evento.

Parte 12

Tercera práctica con FreeRTOS

Evaluación de proyectos finales

- Los que tienen un proyecto en mente, discutámoslo y empecemos a bosquejar una implementación.
- Para los que no, a continuación recordamos las actividades de las dos primeras clases y agregamos algunas más.
- A trabajar!

Tarea opcional para el hogar

- Tarea 1: Destellar un led a una frecuencia determinada
- Tarea 2: Seguir el estado de un pulsador con un led, incluyendo el anti-rebote eléctrico
- Tarea 3: Contar pulsaciones de un boton y luego de un segundo sin pulsaciones destellar un led esa misma cantidad de veces
- Se sugiere implementarlas usando el driver de GPIO provisto por NXP

Tarea opcional para el hogar

- Para todos los ejercicios se sugiere manejar los periféricos que vayan a ser usados en el proyecto.
- El sistema debe pasar en la tarea idle todo el tiempo posible.
- Ejercicio 1: Escribir una tarea que envíe un mensaje al IDE cuando otra tarea registra una pulsación de más de 500ms.
- Ejercicio 2: Contar pulsaciones de un botón y enviar el valor de cuenta a un perif. de salida mediante una cola.
- Ejercicio 3: Reimplementar la función vPrintString usando mutex.

Tarea opcional para el hogar

- Ejercicio 4: Atender un evento aperiódico dando un semáforo en una ISR.
- Ejercicio 5: Implementar un driver de un periférico de salida. El dato se genera en una ISR. Debe implementar las siguientes funciones
  - void InicializarPeriferico (void);
  - void Escribir (portBASE_TYPE dato);
- La ISR puede ser generada en un pulsador, un fin de conversión de un ADC, o pueden usarse las macros del ejemplo 12 desde una tarea periódica.
  - mainCLEAR_INTERRUPT();
  - mainTRIGGER_INTERRUPT();

Parte 11

Funcionalidad en el estado Idle y en el Systick handler

Funcionalidad en el estado Idle

- Cierta funcionalidad de la aplicación puede ser necesario implementarla cuando el sistema está ocioso
  - Poner al CPU en estado de bajo consumo
  - Medir el tiempo disponible del sistema para agregarle más tareas
  - Funcionalidad específica de la aplicación
- Para esto FreeRTOS llama constantemente a una misma función cuando está ocioso. Este tipo de funciones se llaman *hooks* (ganchos) ya que me permiten "enganchar" funcionalidad en este punto.
- El hook que se llama cuando la *Application* está *Idle* se llama **vApplicationIdleHook**.

Tarea Idle (representación)

```
void prvIdleTask( void *pvParameters ){
    for( ;; ) {
        prvCheckTasksWaitingTermination();  // Garbage-collector

        #if ( configUSE_PREEMPTION == 0 ) {
            taskYIELD(); // En modo cooperativo ninguna tarea "preemptea" a la
    tarea Idle
        } #endif

        #if ( configUSE_IDLE_HOOK == 1 ) {
            vApplicationIdleHook();  // Hook
        } #endif
    }
}
```

Cuestiones de implementación

- La tarea Idle NO DEBE BLOQUEAR NI SUSPENDER
  - Si lo hiciera no quedaría ninguna tarea en estado Running, y siempre debe haber una.
- Para usarla, hay que #define *configUSE_IDLE_HOOK* 1
- La función que implemente el hook debe tener exactamente el siguiente prototipo
  - void vApplicationIdleHook (void);
- Si se usa vTaskDelete para eliminar tareas, los recursos de la misma se recuperan en la tarea Idle, por esta razón no debe acapararse la tarea Idle con el *Application Hook*

Uso del Systick timer

- Del mismo modo que hay un *Application Idle* hook, hay un *Systick timer hook.*
- Se llama dentro de una ISR, aplican las reglas habituales.
  - Mantenerla breve
  - No bloquear
  - Usar funciones …FromISR
- Para usarla, hay que #define configUSE_TICK_HOOK 1
- La función que implemente el hook debe tener exactamente el siguiente prototipo
  - void vApplicationTickHook (void);
- Es útil para contar tiempo transcurrido sin dedicarle

Llamada al TickHook

```
void vTaskIncrementTick( void )
{ // Esta función se llama dentro de una sección crítica en el Systick
   Handler
   if( uxSchedulerSuspended == pdFALSE ) {
           ++xTickCount;
           prvCheckDelayedTasks(); // Chequeo de tareas en vTaskDelay
   }

   #if ( configUSE_TICK_HOOK == 1 )
   {
           vApplicationTickHook();  // Hook
   }
   #endif
}
```

Parte 12

Cómo encarar un diseño con
RTOS

Analizar el sistema a implementar

- Lo primero es tener en claro cuáles son las entradas del sistema, sus respectivas respuestas y los compromisos temporales asociados.
- Amén de las prioridades, de este análisis surgirán los tiempos de bloqueo de las operaciones de sincronismo.

Identificar áreas independientes

- Buscar áreas del programa que se puedan programar independientemente de las demás.
  - Estas áreas son candidatas a tener su propia tarea.
  - Tener presente que si una tarea está completamente anidada dentro de otra, puede ser una simple función y no necesita una tarea.
  - No todo es una tarea!
- Las tareas deben tener poca dependencia entre ellas, una relación complicada entre ellas es síntoma de un mal diseño y complica el posterior mantenimiento del software.

Considerar un patrón MVC

- MVC es un patrón de diseño usado mucho en web.
- Modelo – Vistas – Controladores.
  - El modelo es el estado del sistema.
  - Las vistas son las representaciones del mismo que ve el usuario.
  - Los controladores son funciones que modifican el estado del modelo.
- Ayuda a mantener independientes las tareas cuando hay varias entradas y salidas dependientes del contexto de la aplicación:
  - Las entradas envían mensajes al modelo,
  - Las salidas reciben mensajes del modelo.

Considerar un patrón MVC

- El modelo puede ser tan simple como una estructura en memoria compartida.
- Ayuda a evitar que muchas tareas manden mensajes a muchas otras.

Atender a las tareas reentrantes

- Algunas tareas se instancias varias veces.
- Se debe tener cuidado con el uso que hacen de recursos compartidos.
  - No se deben usar variables globales, ya que todas las tareas ven la misma copia.
  - Si usan un mismo recurso, se lo debe tratar con mutex como si fueran tareas distintas.

Asignar prioridades a las tareas

- Se debe tener en claro qué eventos tienen requisitos temporales estrictos y cuáles relajados.
- Ayuda pensar en la superposición de todos los eventos y determinar cuales deben ser atendidos primero.
- Considerar el caso de semáforos y colas
  - Cuando se libera un semáforo, es más importante la atención del evento o la tarea que lo señaliza tiene todavía trabajo por hacer?
  - Cuando un productor escribe datos en una cola, el consumidor debe pasar a ejecución en ese momento o más tarde, cuando el productor ceda el CPU?
- Cuidado con el starvation!

Buscar IPC y sincronismos entre las tareas

- Algunas tareas esperarán eventos externos, otras esperarán mensajes de otras tareas.
- De este análisis surgen los semáforos y colas a usar.
  - Atento a los valores iniciales de los semáforos binarios y contadores.

Dimensionar las colas de mensajes

- Según las prioridades de los productores y consumidores de las colas, estas pueden mantenerse más bien llenas o vacías. En base a esto debe determinar la cantidad de elementos a almacenar en ellas.

Identificar los recursos compartidos

- Las tareas tienen necesariamente algunos puntos de contacto. Estos se deben proteger con mutex.
  - P. ej: Una tarea lee una variable de configuración y otra tarea la modifica según una entrada del sistema. No se debe corromper el acceso!
- Estos recursos pueden ser periféricos del CPU o posiciones de memoria.
- Atento a la inversión de prioridades cuando hay mutex en juego y tareas de distintas prioridades!
  - El tiempo durante el que se retiene el mutex es crucial para determinar si habrá o no problemas.

Identificar operaciones atómicas

- Algunas tareas no pueden ser interrumpidas
- Considerar quién puede interrumpir la tarea para determinar la mejor estrategia de protección.
  - Si es sólo una tarea, tal vez alcanza con suspenderla o bajarle la prioridad.
  - Si cualquier cambio de contexto es problemático se puede suspender el scheduler o las interrupciones.

Diseñar los drivers que hagan falta

- Es importante tener clara la funcionalidad requerida del dispositivo en cuestión.
- Un driver puede ser algo tan sencillo como una cola de entrada y una de salida.
- Si una cola es demasiado lenta para el caso, se puede usar memoria compartida (variables globales en el caso de FreeRTOS) y semáforos para señalizar los eventos.

Implementación y verificación

- Proceder a implementar las tareas de a una, verificando que se cumplan los requisitos temporales.
- Se recomienda tener un buen modelo del sistema antes de proceder a esto.

Parte 13

Asignación dinámica de memoria

Uso de memoria dinámica

- El OS asigna memoria dinámica para gestionar sus objetos cuando los crea:
  - Tareas
  - Semaforos / Mutex
  - Colas
- Esta memoria la obtiene del heap definido en FreeRTOSConfig.h, mediante llamadas similares a Malloc y Free.
- Por esta razón durante el desarrollo debe verificarse que las llamadas que crean estos objetos no devuelvan un error.

Problema de Malloc y el determinismo

- Las aplicaciones de tiempo real no pueden permitirse el comportamiento del Malloc estándar
  - Tiempo de ejecución variable
  - Puede o no encontrar el bloque de memoria solicitado
  - Implementación muy abultada para la memoria disponible
- Por esta razón, FreeRTOS separa la asignación de memoria del resto del kernel y la ubica en la capa portable, se puede modificar el algoritmo de manejo de memoria en cada aplicación.
- FreeRTOS proporciona tres algoritmos de manejo de memoria, el programador puede elegir uno o escribir uno propio.
  - Heap_1.C, Heap_2.C. Heap_3.C
  - Debe haber uno solo de estos (o uno propio) en la carpeta *portable*
- Las llamadas a Malloc y Free se reemplazan con *pvPortMalloc* y *pvPortFree*.

Heap_1.C

- No contempla la liberación de memoria - no implementa pvPortFree()
- La memoria se asigna secuencialmente hasta que se acaba
- El total disponible para el kernel está definido en configTOTAL_HEAP_SIZE
- Es el más simple de usar cuando la memoria disponible es suficiente para todos los objetos que se van a crear
  - El tiempo de ejecución es determinista
  - La memoria no se fragmenta

Heap_2.C

- Contempla la liberación de memoria, implementa pvPortFree()
- Usa un algoritmo "best-fit".
  - Recorre todos los bloques disponibles de memoria y devuelve el que más se ajusta a lo pedido - No es determinista!
  - El sobrante del bloque asignado queda marcado como un nuevo bloque.
  - La memoria se fragmenta con las des/asignaciones.
- El total disponible para el kernel está definido en configTOTAL_HEAP_SIZE.
- Es útil cuando se crean y destruyen distintas tareas pero con el mismo tamaño de stack. De este modo la memoria liberada se vuelve a ocupar sin desperdicios.

Heap_3.C

- Usa la implementación estándar de Malloc y Free
- No es determinista
- Encierra ambas operaciones en secciones críticas para hacerlo thread-safe.
- No usa el tamaño de heap definido por configTOTAL_HEAP_SIZE, si no el área de memoria definida para esto en el linker script.

Cuál elegir?

- Si la totalidad de los objetos de la aplicación caben en la memoria disponible, Heap_1 es el más adecuado.
- Si se necesitan más tareas corriendo simultáneamente de las que se pueden albergar, se puede crear una tarea cuando se la necesita y eliminarla luego de su uso. En este caso se necesita al menos Heap_2.
  - Para optimizar el uso de memoria, las tareas a reciclar debieran tener el mismo tamaño de stack.
- Si se van a reciclar tareas con distinto tamaño de stack, se puede usar Heap_3 o escribir un handler propio.
  - Pero a estas alturas el sistema ya se vuelve complicado de analizar y predecir, sería mejor revisar el diseño para utilizar Heap_2 o definir un algoritmo a medida e implementar un manejador propio.

Cómo dimensionar el Heap?

- Si se usa Heap_1 o Heap_2, la función xPortGetFreeHeapSize (void) devuelve la memoria disponible en el heap
- Llamarla luego de que se crearon todos los objetos del sistema me permite saber en cuánto disminuir su tamaño sin hacer un análisis minucioso.

Parte 14

Verificación del uso de las pilas

Protección contra el desborde de pila

- La pila de una tarea no debe desbordarse en ningun momento
- El parámetro de medición es qué tan alto llegó el stack pointer. A este valor se le llama *high-water mark*.
- FreeRTOS provee una función para esto llamada uxTaskGetStackHighWaterMark (xTaskHandle tarea);
- Devuelve el mínimo disponible que hubo de stack desde que la tarea empezó a ejecutar
- Se la puede usar para dimensionar el stack de la tarea

Chequeos en tiempo de ejecución

- El momento más delicado para la pila es cuando la tarea sale de ejecución. Todo el contexto se vuelca en su pila.
  - Por este motivo está definida la constante configMINIMAL_STACK_SIZE, las tareas no deben tener una pila menor a este tamaño.
- Se puede configurar el kernel para que en ese momento verifique si la pila desbordó.
  - Estos chequeos aumentan el tiempo del cambio de contexto!
- Se puede implementar un hook para ayudar a depurar el problema.
  - void vApplicationStackOverflowHook (…)

Dos métodos

- Método 1: Verifica que el stack pointer esté dentro de la pila de la tarea al volcarle el contexto.
  - Ejecuta rápido
  - Si la pila desborda durante la ejecución pero vuelve a un nivel permitido antes de salir de contexto, no lo detecta.
- Método 2: Verifica que los últimos 20 bytes de la pila mantengan su valor inicial
  - Cuando se crea una pila se le da un valor conocido
  - Es más lento pero ciertamente más seguro
    - Sólo fallaría en el caso de que el desborde de la pila tenga el mismo patrón que el valor inicial.
- Se elige con #define configCHECK_FOR_STACK_OVERFLOW 1 o 2
  - Definirlo en 0 deshabilita los chequeos

Depuración

- Cuando se produce un desborde de pila el error es irrecuperable
- Lo habitual es atrapar la ejecución en un lazo infinito y verificar con el entorno de depuración qué es lo falló.
  - Se puede hacer un trace de las instrucciones ejecutadas o de las funciones llamadas, para determinar el flujo de ejecución que llevó a esa falla.

Parte 15

Las 5 cosas que no entraron en el curso

1. Otros problemas de la concurrencia

- Reentrancia
  - Si una tarea se va a instanciar varias veces, no debe guardar sus datos en variables globales, ya que todas las instancias ven la misma variable y podrían corromperla.
    - Las variables locales están en el stack de cada instancia.
  - A la condición de no usar variables globales y permitir la reentrancia se la llama “thread-safe”.
- Carreras críticas (race conditions)
  - Si varias tareas usan una misma variable (p. ej un contador global), el acceso al mismo debe ser mutex.
  - Un acceso no mutex a un contador puede hacer que el código en cuestión ejecute más o menos veces que lo que debería.

2. Coroutines

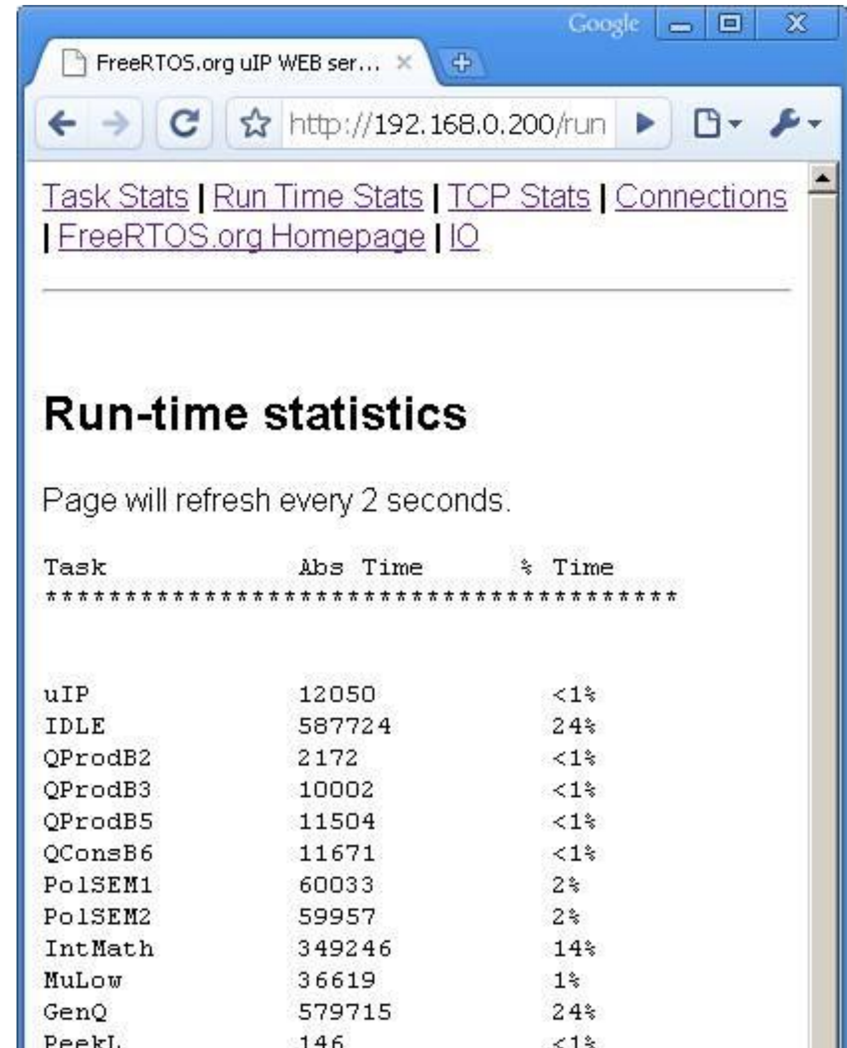
- Ver http://www.freertos.org/croutine.html
- Las CR son como los threads de un sistema tipo UNIX.
  - Una tarea puede tener varias CR
  - Tienen sus propias prioridades que aplican solo entre las CR de una misma tarea.
  - Comparten el stack de la tarea que las contiene.
- Son un recurso a tener en cuenta en procesadores con muy poca memoria de datos.
- Se puede implementar un sistema con solo CR, solo tareas o una mezcla de ambas

3. Trazado de la ejecución

- Ver http://www.freertos.org/rtos-trace-macros.html
- A lo largo del código hay numerosas macros en puntos de interés de la aplicación
  - traceQUEUE_SEND_FROM_ISR
  - traceTASK_SWITCHED_IN
  - traceTASK_SWITCHED_OUT
- Por defecto están definidas sin cuerpo, de modo que no se agrega overhead a la ejecución
- Se pueden definir con el comportamiento deseado para analizar la ejecución de la aplicación
  - Por ej. Medir el tiempo entre dar un semáforo y la atención del evento

4. Estadísticas en tiempo de ejecución

- Ver http://www.freertos.org/rtos-run-time-stats.html
- La API tiene varias funciones para generar estos datos
- Las demos para plataformas con interfaz ethernet contienen una tarea para mostrar esta información



5. Otros algoritmos de scheduling

- http://en.wikipedia.org/wiki/Rate-monotonic_scheduling
  - Es una manera de asignar las prioridades a las tareas.
  - Consiste en asignarlas de modo inversamente proporcional al tiempo de ejecución de la tarea.
  - Busca maximiza el throughput de tareas ejecutadas correctamente.
- http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling
  - Asigna mayor prioridad a las tareas más próximas a fallar su deadline
  - No es fixed-priority, el scheduler modifica la prioridad de las tareas dinámicamente para garantizar que ninguna fracase
  - Si el sistema está sobrecargado es difícil de predecir qué tareas fallarán su deadline.

Fin de la 1era clase

Se sugiere la realización de las actividades propuestas.

Preguntas?

Fin de la 2da clase

- Se sugiere la realización de las actividades propuestas.
- Preguntas?
- Las próximas dos clases analizaremos los proyectos finales.
- Se sugiere que traigan las alternativas que estén manejando para analizar que proyectos es conveniente implementar con FreeRTOS.

Fin de la 3ra clase

- Se sugiere la realización de las actividades propuestas.
- Sería bueno que vayan definiendo los proyectos!
- Preguntas?

Fin de la exposición

- Preguntas?
- A trabajar en los proyectos!