SISTEMAS OPERATIVOS DE TIEMPO REAL (RTOS)

Un sistema operativo en tiempo real es más parecido a una librería de funciones que se enlazan con la aplicación, de forma que al arrancar el ordenador es la aplicación la que toma el control del ordenador e inicializa el sistema operativo, para luego pasarle el control. Esto permite eliminar las partes del sistema operativo que no se usen para ahorrar memoria, que suele estar limitada en los sistemas empotrados.

Estos no se protegen frente a errores de aplicaciones. Estos sistemas ejecutan solo una aplicación por lo tanto debemos tener más cuidado a la hora de programar, ya que si se cuelga tal aplicación todo el sistema se cuelga con ella. Por así decirlo arranca la aplicación (Programa) el cual se especifica que arranque el planificador y recién ahí arranca el sistema operativo.

El planificador

El planificador es la parte del sistema operativo en tiempo real que controla el estado de cada tarea y decide cuándo una tarea pasa al estado de ejecución. Tengo multitarea gracias al planificador ya que es el encargado de cambiar las tareas de primer plano y usa la información como su prioridad o sus límites temporales para decidir cual ejecutar, y las tareas de segundo plano las manejan las interrupciones. Existen dos tipos de planificadores: cooperativos (non preemptive) y expropiativos (preemptive).

Planificador cooperativo

En el planificador cooperativo, son las propias tareas de primer plano las encargadas de llamar al planificador cuando terminan su ejecución o bien cuando consideran que llevan demasiado tiempo usando la CPU. El planificador entonces decidirá cuál es la tarea que tiene que ejecutarse a continuación. En caso de que no existan tareas más prioritarias listas para ejecutarse, el planificador le devolverá el control a la primera tarea para que continúe su ejecución. Si por el contrario existe alguna tarea con más prioridad lista para ejecutarse, se efectuará un cambio de contexto y se cederá la CPU a la tarea más prioritaria.

El problema de este planificador es la falta de control sobre la latencia de las tareas de primer plano, ya que el planificador solo se ejecuta cuando la tarea que está usando la CPU termina. Esto hace que sea difícil de garantizar a temporización de las tareas, pudiendo ocurrir que alguna de ellas no cumpla con su límite temporal.

Planificador expropiativo

Mejora la latencia de las tareas. El planificador se ejecuta periódicamente de forma automática, lo que hace en realidad es que se ejecuta un timer en donde cada cierto tiempo genera una interrupción y esta interrupción ejecuta mi planificador. Además, si antes de cumplirse el quantum, se bloquea la tarea, se ejecuta el planificador. Por otra parte, verifica si hay una tarea más prioritaria que ejecutar y en caso afirmativo hace un cambio de contexto para ejecutarla.

Tareas

Una tarea no es más que una función en C, aunque esta función obviamente puede llamar a otras funciones. La única condición que ha de cumplir una función para convertirse en una

tarea es que no termine nunca, es decir, ha de contener un bucle infinito en el que realiza sus acciones.

La tarea se inicializa mediante una llamada al sistema operativo en tiempo real, especificándose en dicha llamada la prioridad de la tarea, la memoria que necesita, la función que la implanta (denominada punto de entrada), etc.

Estados de una tarea

- ***Ejecución (Running):*** El microprocesador la está ejecutando. Sólo puede haber una tarea en este estado.
- ***Lista (Ready):*** La tarea tiene trabajo que hacer y está esperando a que el procesador esté disponible. Puede haber un número cualquiera de tareas en este estado.
- ***Bloqueada (Blocked):*** No tiene nada que hacer en este momento. Está esperando algún suceso externo. Puede haber un número cualquiera de tareas en este estado.

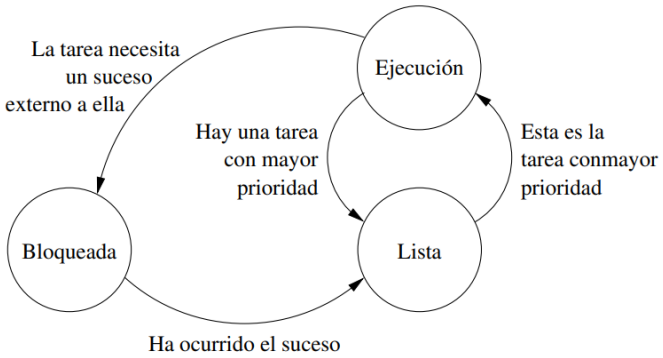

Figura 4.1: Estados de una tarea.

- Una tarea sólo puede bloquearse cuando esté ejecutándose, ya que será entonces cuando llegue a un punto en el que necesite algún dato proporcionado por otra tarea (por ejemplo, por una interrupción) o necesite esperar un determinado periodo de tiempo.

- Para que una tarea bloqueada pase a lista, otra tarea debe despertarla. Siguiendo con el ejemplo anterior, la tarea que se bloqueó esperando datos de una cola no se despertará hasta que otra tarea deposite un dato en la cola.

- Una vez que una tarea está en el estado “lista”, su paso al estado de “ejecución” depende sólo del planificador. Sólo cuando esta tarea sea la de mayor prioridad pasará a ejecutarse (solamente en expropiativo).

FreeRTOS

Planificador en FreeRTOS

El planificador no modifica las prioridades de las tares, *SIEMPRE* ejecuta la tarea de mayor prioridad que está en *READY* (no espera el quantum). Si el planificador está en modo time slicing, al cabo de un tiempo el planificador va a retirar la tarea en ejecución solo si hubiera una de igual prioridad en *READY.*

Time Slicing (segmentación de tiempo): se utiliza para compartir el tiempo de procesamiento entre tareas de igual prioridad, incluso cuando las tareas no cedan explícitamente ni entren en el estado Bloqueado. Los algoritmos del planificador que se describen como el uso de "Time Slicing" seleccionarán una nueva

tarea para ingresar al estado "En ejecución" al final de cada fracción de tiempo si hay otras tareas en estado "Listo" que tienen la misma prioridad que la tarea en ejecución. Un intervalo de tiempo (time slice) es igual al tiempo entre dos interrupciones de tic de RTOS.

Las tareas de menor prioridad, no reciben tiempo de ejecución. Este quantum se puede modificar cambiando la frecuencia (configTICK_RATE_HZ) que es la inversa. Si lo hago muy grande al tiempo tengo una latencia más grande y si lo hago muy chico va a ejecutar la mayoría de tiempo esta rutina. Las tareas también se pueden Suspender, no serán tenidas en cuenta por el planificador. Luego se pueden Reanudadas

Ventajas:

- ✓ Multitarea: simplifica la estructura del código.
- ✓ Escalabilidad: se pueden agregar tareas sin tener que modificar tanto el programa.
- ✓ Reusabilidad de código: si las tareas tienen poca dependencia es fácil incorporarlas a otras aplicaciones.

Desventajas:

- ✓ Usamos tiempo en el cambio de tareas, esta rutina que hace este cambio necesita estar en memoria entonces ocupa espacio.
- ✓ El programador NO decide cuando ejecutar cada tarea si no el planificador.
- ✓ No protege contra errores de programación.

Tareas en FreeRTOS

La unidad básica para planificar son las tareas. 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 *pvParametros)
```

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 (las tareas no deberían terminar nunca). Si una tarea deja de ser necesaria, puede eliminársela explícitamente. Si una nueva tarea es necesaria, cualquier tarea puede crearla explícitamente.

Tipos de Tareas

- ***Tareas periódicas:*** con una frecuencia determinada. La mayor parte de su tiempo están en el estado bloqueado hasta que espera un tiempo de bloqueo, en este momento pasa al estado listo. Alta prioridad para tener baja latencia
- ***Tareas aperiódicas:*** con frecuencia indeterminada. Tareas inactivas bloqueadas hasta que ocurre un evento externo.
- ***Tareas continuas****:* régimen permanente. Deben tener menor prioridad que el resto, ya que en caso contrario podrían impedir su ejecución.

Semáforos

Su función es: Restringir el acceso a una sección particular del programa (exclusión mutua), Ordenar cronológicamente eventos (serializar) y punto de encuentro.

Tiene dos primitivas, incrementar y decrementar. La diferencia entre RTS y RTOS es que en RTOS se define un tiempo máximo en que va a estar bloqueada la tarea que intenta decrementar el semáforo, esto lo hacemos para asegurar tiempos y no estar esperando por alguna mala programación.

Deadlock

El deadlock ocurre cuando dos tareas se bloquean intentando obtener el mutex de un recurso que tiene la otra. Ninguna de las dos libera el mutex ya que está bloqueada esperando obtener el otro. Finalmente, ninguna de las dos tareas recibe tiempo de ejecución.

Por suerte FreeRTOS tiene timeouts, no llegan a ser deadlock, ya que la primera tarea en pedir el semáforo, cuando termine su timeout, lo soltará y permitirá a la otra realizar su tarea.

Métodos para proteger recursos compartidos

- ***Inhabilitar las interrupciones:*** Es el método más drástico, pues afecta a los tiempos de respuesta de todas las tareas e interrupciones. Sin embargo, es el método más rápido (una o dos instrucciones de código máquina) y es la única manera de proteger datos compartidos con interrupciones. Sólo es válido cuando la zona crítica es muy pequeña para que las interrupciones no estén inhabilitadas mucho tiempo.

- ***Inhabilitar las conmutaciones de tareas:*** Es un método menos drástico que cortar las interrupciones, ya que sólo afectará a los tiempos de respuesta de las tareas de primer plano, que normalmente no son tan estrictos como los de las rutinas de interrupción. Para inhabilitar la conmutación de tareas basta con realizar una llamada al sistema operativo. Por ejemplo, en FreeRTOS basta con llamar a la función vTaskSuspendAll. Desde el momento en el que la tarea llama a esta función, ésta sólo será interrumpida por las rutinas de atención a interrupciones, pero no por otras tareas más prioritarias. Cuando la tarea termine de usar el recurso compartido ha de llamar a la función xTaskResumeAll para que el sistema operativo vuelva a conmutar tareas cuando sea necesario. Este método requiere una sobrecarga un poco mayor que la inhabilitación de interrupciones, pero menor que el uso de semáforos. Dado que afecta a los tiempos de respuesta de todas las tareas, al igual que la inhabilitación de interrupciones sólo será recomendable si la zona crítica no es muy larga.

- ***Usar semáforos***: La ventaja principal es que sólo afecta a las tareas que comparten el recurso. El inconveniente es que, al ser un mecanismo más complejo, su gestión por parte del sistema operativo presenta una mayor carga al sistema. Por otro lado, como se ha mencionado en la sección anterior, si no se usan con cuidado pueden originar errores difíciles de detectar y corregir.

Inversión de prioridad

El uso de semáforos puede producir una inversión de prioridad, que consiste en que una tarea de menor prioridad impide la ejecución de una tarea de mayor prioridad.

En un sistema hay tres tareas a las que se denomina A, B y C. La tarea A tiene mayor prioridad que la B y ésta a su vez mayor que la C. Se ha supuesto que inicialmente las tareas A y B están bloqueadas y se está ejecutando la tarea C. Durante su ejecución, la tarea C toma un semáforo que protege un recurso compartido que también usa la tarea A.

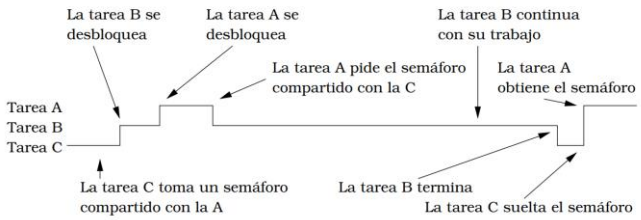

Figura 4.4: Inversión de prioridad

A continuación, se ha supuesto que se desbloquea la tarea B y, como tiene mayor prioridad que la C, el planificador realizará un cambio de contexto para pasar a ejecutar la tarea B. Seguidamente la tarea A se desbloquea también y el planificador pasará a ejecutarla. El problema se produce al pedir la tarea A el semáforo que protege el recurso que comparte con la tarea C. Como el semáforo está ocupado, la tarea A ha de bloquearse hasta que éste se libere. Como puede observar en la figura, una vez que se bloquea la tarea A, el planificador busca la tarea más prioritaria que está en estado de "lista para ejecución", la cual es la B. Por tanto, se ejecutará la tarea B y hasta que ésta no termine no se ejecutará la tarea C. Sólo cuando la tarea C se ejecute se liberará el semáforo que mantiene a la tarea A bloqueada. Por tanto, la tarea B en realidad está impidiendo que se ejecute la tarea A que tiene mayor prioridad que ella. Esto es lo que se conoce como inversión de prioridad: la tarea B se ha ejecutado, aunque la tarea A tenía que hacerlo.

La solución para esto es usar planificadores no apropiativos y si no herencia de prioridad (Mutex no soluciona, pero disminuye). En este caso lo que haría, es que cambiaría la prioridad de las tareas C y A para que cuando la tarea A pida el semáforo y se bloquee termine de ejecutar la C para poder liberarlo y poder ejecutar la de mayor prioridad A ***en vez de tener que esperar a la tarea B que termine.***

Colas para comunicar tareas

Los sistemas operativos en tiempo real también disponen de colas para permitir comunicar varias tareas entre sí. Normalmente se usan:

- Cuando se necesita un almacenamiento temporal para soportar ráfagas de datos.
- Cuando existen varios generadores de datos y un sólo consumidor y no se desea bloquear a los generadores a la espera de que el consumidor obtenga los datos

El funcionamiento de las colas es similar al estudiado para los sistemas *Foreground/Background,* sólo que ahora la gestión la realiza el sistema operativo en lugar del programador. Las principales diferencias son:

- Las colas se crean mediante una la llamada al sistema. Esta llamada se encarga tanto de crear la memoria necesaria para albergar a todos los elementos de la cola, como de crear una estructura de control, a la que denominaremos "manejador"
- El manejo de la cola, es decir, enviar y recibir datos, se realiza también exclusivamente mediante llamadas al sistema.
- En el sistema operativo FreeRTOS los elementos de la cola pueden ser de cualquier tipo, incluyendo estructuras de datos.
- Las funciones que envían y reciben datos de la cola se pueden bloquear cuando la cola esté llena o vacía, respectivamente.

Rutinas de atención a interrupción en los sistemas operativos en tiempo real

Cuando se usa un sistema operativo en tiempo real, es muy importante tener en cuenta dos precauciones a la hora de escribir rutinas de atención a las interrupciones:

- No se deben llamar funciones del sistema operativo en tiempo real que puedan bloquearse desde la rutina de atención a interrupción. Si se bloquea una interrupción, la latencia de ésta aumentará a límites intolerables y, lo que es peor, poco predecibles.
- No se deben llamar funciones del sistema operativo que puedan conmutar tareas, salvo que el sistema sepa que se está ejecutando una interrupción. Si el sistema operativo en tiempo real no sabe que se está ejecutando una interrupción, pensará que se está ejecutando la tarea que se ha interrumpido. Si la rutina de interrupción realiza una llamada que hace que el planificador pueda conmutar a una tarea de mayor prioridad (como por ejemplo escribir un mensaje en una cola), el sistema operativo realizará la conmutación, con lo que la rutina de atención a la interrupción no terminará hasta que terminen las tareas de mayor prioridad. En la figura 4.5 se ilustra el proceso gráficamente.

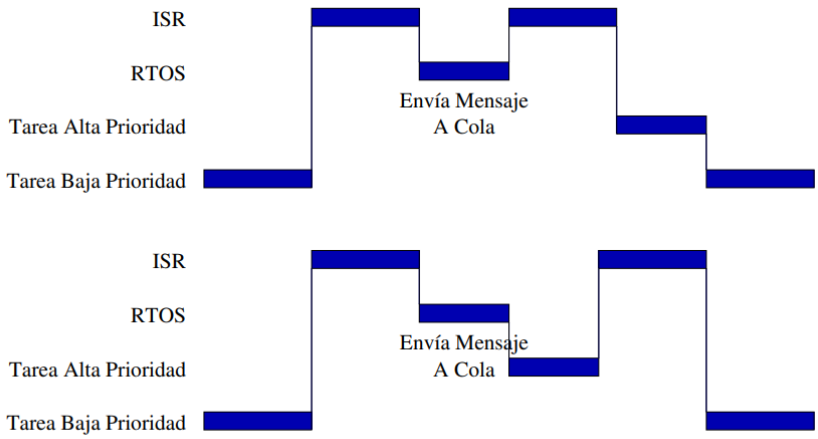

Figura 4.5: Interrupciones y sistemas operativos en tiempo real

La parte superior de la figura 4.5 muestra el comportamiento deseado: la rutina de atención a la interrupción (ISR) llama al sistema operativo en tiempo real y esta llamada despierta a la tarea de alta prioridad. No obstante, el sistema operativo espera a que termine de ejecutarse la rutina de interrupción y al finalizar ésta, en lugar de volver a la tarea de baja prioridad que estaba ejecutándose antes de producirse la interrupción, realiza un cambio de contexto y pasa a ejecutar la tarea de alta prioridad.

La parte inferior de la figura 4.5 muestra lo que ocurre si el sistema operativo no se entera de cuándo se está ejecutando una tarea y cuándo una rutina de interrupción. En este caso, al realizar la llamada al sistema desde la rutina de interrupción y despertarse la tarea de alta prioridad, el sistema operativo realiza el cambio de contexto y pasa a ejecutar la tarea de alta prioridad. Cuando ésta termina, vuelve a conmutar al contexto de la tarea de baja prioridad, con lo que continúa ejecutándose la rutina de interrupción y, cuando ésta termine, la tarea de baja prioridad. Obviamente, en este caso la rutina de interrupción tardará demasiado en ejecutarse. Este retraso puede originar que se pierdan interrupciones, lo cual es intolerable en un sistema en tiempo real.

Existen tres métodos para tratar este problema:

- Avisar al sistema operativo en tiempo real de la entrada y salida de la rutina de atención a interrupción.
- El sistema operativo en tiempo real intercepta todas las interrupciones y luego llama a la rutina de atención a interrupción proporcionada por la aplicación.
- Existen funciones especiales para su llamada desde las ISR. Este método usa la FreeRTOS.

Gestión de tiempo

En un sistema en tiempo real, son numerosas las situaciones en las que es necesario garantizar la ejecución periódica de una tarea o suspender la ejecución de una tarea durante un determinado periodo de tiempo.

La gestión de tiempo en la mayoría de los sistemas operativos se basa en usar una interrupción periódica para incrementar un contador en el que se lleva la cuenta del tiempo transcurrido desde que se arrancó el sistema. Cada incremento del contador se denomina tick de reloj.