FreeRTOS

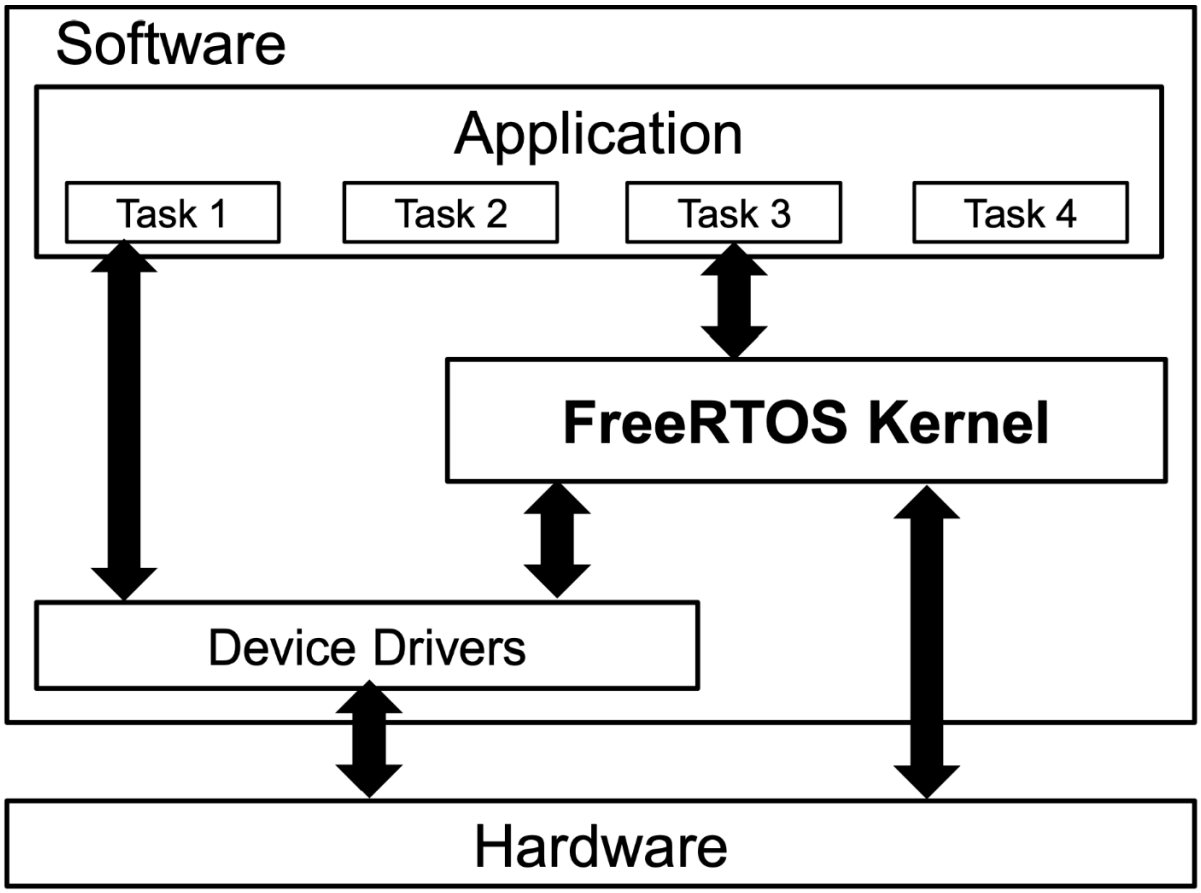
Caratteristiche

Caratteristiche generali

FreeRTOS è un sistema operativo in tempo reale (RTOS) progettato per sistemi embedded e applicazioni in tempo reale. È ottimizzato per sistemi con risorse limitate come microcontrollori e microprocessori, supportando 35 differenti architetture.

Esso è completamente open source, sviluppato da Real Time Engineers Ltd, poi passato nel 2017 sotto il controllo di amazon nel progetto AWS.

Architettura



L'architettura di FreeRTOS è basata su un kernel a thread preemptive, che fornisce un'interfaccia semplice per la creazione, l'esecuzione e la gestione dei thread (noti come "tasks" in FreeRTOS).

L'architettura di FreeRTOS è stata progettata per essere “small and simple”, facile da integrare in progetti embedded. La sua portabilità, flessibilità e affidabilità lo hanno reso una scelta popolare per una vasta gamma di applicazioni in tempo reale, come dispositivi IoT, controllori industriali, sistemi di controllo di veicoli e molti altri.

Il kernel consiste in soli tre file C:

- task.c
- list.c
- queue.c

ed altri tre opzionali:

- timers.c, per eseguire operazioni periodiche o ritardate;
- coroutine.c, per la creazione e esecuzione di task cooperativi;
- event_groups.c, per consentire la comunicazione e sincronizzazione tra i task basati su eventi.

Task

I task sono le unità fondamentali di esecuzione in FreeRTOS. Ogni task rappresenta un flusso di esecuzione indipendente, che può essere programmato per eseguire un'attività specifica. Ogni task ha una priorità associata che viene utilizzata per l'allocazione della CPU quando più task sono pronti per l'esecuzione contemporaneamente.

Ogni task ha un proprio stack e può assumere diversi stati:

- Running: task in esecuzione
- Ready: task pronto per l’esecuzione, ma in attesa della CPU
- Blocked: task ritardato o in attesa di un altro task
- Suspend: Non considerato nello scheduling

Ad ogni task è associata una priorità:

- da 0 a configMAX_PRIORITIES-1
- valori bassi denotano bassa priorità
- Il task che assume lo stato di Running è sempre quello che ha la priorità più alta
- Task con la stessa priorità condividono l’utilizzo della CPU usando un time sliced round robin scheduling scheme

Tick

Un’altra caratteristica essenziale di Free-RTOS sono i tick interrupt. Il kernel infatti misura il tempo usando una variabile di conteggio denominata tick. Quando scatta il tick interrupt, la variabile tick viene incrementata dalla TickISR, (una routine che viene eseguita in risposta all’interrupt).

Se ci sono task da eseguire con priorità maggiore del task in esecuzione, la TickISR avvia il cambio di contesto.

Confronto con OS161

Bisogna tenere conto innanzitutto che i due sistemi operativi hanno obiettivi e caratteristiche differenti. FreeRTOS è un sistema operativo in tempo reale utilizzato sia per scopi didattici che in ambienti lavorativi ottimizzato per sistemi con risorse limitate. D’altro canto OS161 è un sistema operativo

puramente didattico progettato proprio per permettere di esplorare e apprendere i concetti fondamentali dei sistemi operativi.

Architettura

Come già detto l'architettura di FreeRTOS si basa su un kernel di scheduling prioritario supportando la programmazione multitasking prioritaria offrendo funzionalità per la gestione delle code, semafori e mutex.

OS161 invece è basato su un kernel monolitico e segue un'architettura semplice e modulare, offrendo funzionalità di base come la gestione dei processi, thread, file e della memoria virtuale

Portabilità

FreeRTOS è altamente portabile e supporta una vasta gamma di architetture e microcontrollori.

OS161 è principalmente eseguito su macchine virtuali simulate.

System Calls

FreeRTOS, a differenza di OS161, è un sistema operativo in tempo reale progettato principalmente per dispositivi embedded e quindi non segue l'approccio delle syscall come in sistemi operativi come Linux.

Al loro posto FreeRTOS fornisce le proprie API per la gestione dei task, la sincronizzazione, la comunicazione tra task e altre funzionalità specifiche per i sistemi in tempo reale.

In OS161 le system call sono definite, ma non implementate e sta allo studente fornire implementazioni adeguate. Nei laboratori abbiamo fornito delle implementazioni per SYS_read, SYS_write e SYS_exit. Prendendo come esempio queste syscall FreeRTOS non le implementa tutte: SYS_read e SYS_write possono essere implementate utilizzando API fornite da Queue.h, ovvero facendo uso di una coda, che può essere utilizzata per trasmettere o ricevere dati tra task o per comunicare con un dispositivo di I/O.

Per quanto riguarda invece la SYS_exit al suo posto viene utilizzata la funzione `vTaskDelete()` all'interno del task stesso per terminare un task in FreeRTOS .
Questa funzione permette al task di terminare la sua esecuzione e liberare le risorse associate ad esso.

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
    {
        TCB_t * pxTCB;

        taskENTER_CRITICAL();
        {
            /* If null is passed in here then it is the calling task that is
             * being deleted. */
            pxTCB = prvGetTCBFromHandle( xTaskToDelete );

            /* Remove task from the ready/delayed list. */
            if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )
            {
                taskRESET_READY_PRIORITY( pxTCB->uxPriority );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Is the task waiting on an event also? */
            if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
            {
```

```
                ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Increment the uxTaskNumber also so kernel aware debuggers can
             * detect that the task lists need re-generating.  This is done before
             * portPRE_TASK_DELETE_HOOK() as in the Windows port that macro will
             * not return. */
            uxTaskNumber++;

            if( pxTCB == pxCurrentTCB )
            {
                /* A task is deleting itself.  This cannot complete within the
                 * task itself, as a context switch to another task is required.
                 * Place the task in the termination list.  The idle task will
                 * check the termination list and free up any memory allocated by
                 * the scheduler for the TCB and stack of the deleted task. */
                vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->xStateListItem ) );

                /* Increment the ucTasksDeleted variable so the idle task knows
                 * there is a task that has been deleted and that it should therefore
                 * check the xTasksWaitingTermination list. */
                ++uxDeletedTasksWaitingCleanUp;

                /* Call the delete hook before portPRE_TASK_DELETE_HOOK() as
                 * portPRE_TASK_DELETE_HOOK() does not return in the Win32 port. */
                traceTASK_DELETE( pxTCB );

                /* The pre-delete hook is primarily for the Windows simulator,
                 * in which Windows specific clean up operations are performed,
                 * after which it is not possible to yield away from this task -
                 * hence xYieldPending is used to latch that a context switch is
                 * required. */
                portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );
            }
            else
            {
                --uxCurrentNumberOfTasks;
                traceTASK_DELETE( pxTCB );

                /* Reset the next expected unblock time in case it referred to
                 * the task that has just been deleted. */
                prvResetNextTaskUnblockTime();
            }
        }
        taskEXIT_CRITICAL();

        /* If the task is not deleting itself, call prvDeleteTCB from outside of
         * critical section. If a task deletes itself, prvDeleteTCB is called
         * from prvCheckTasksWaitingTermination which is called from Idle task. */
        if( pxTCB != pxCurrentTCB )
        {
            prvDeleteTCB( pxTCB );
        }

        /* Force a reschedule if it is the currently running task that has just
         * been deleted. */
        if( xSchedulerRunning != pdFALSE )
        {
            if( pxTCB == pxCurrentTCB )
            {
                configASSERT( uxSchedulerSuspended == ( UBaseType_t ) 0U );
                portYIELD_WITHIN_API();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
```

```
        }
    }
```

xTaskToDelete: handler del task (NULL se si riferisce a se stesso)

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
    {
        TCB_t * pxTCB;

        taskENTER_CRITICAL();
        {
            /* If null is passed in here then it is the calling task that is
             * being deleted. */
            pxTCB = prvGetTCBFromHandle( xTaskToDelete );

            /* Remove task from the ready/delayed list. */
            if( uxListRemove( &( pxTCB->xStateListItem ) ) == ( UBaseType_t ) 0 )
            {
                taskRESET_READY_PRIORITY( pxTCB->uxPriority );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Is the task waiting on an event also? */
            if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
            {
                ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            /* Increment the uxTaskNumber also so kernel aware debuggers can
             * detect that the task lists need re-generating.  This is done before
             * portPRE_TASK_DELETE_HOOK() as in the Windows port that macro will
             * not return. */
            uxTaskNumber++;

            if( pxTCB == pxCurrentTCB )
            {
                vListInsertEnd( &xTasksWaitingTermination, &( pxTCB->xStateListItem ) );

                /* Increment the ucTasksDeleted variable so the idle task knows
                 * there is a task that has been deleted and that it should therefore
                 * check the xTasksWaitingTermination list. */
                ++uxDeletedTasksWaitingCleanUp;

                /* Call the delete hook before portPRE_TASK_DELETE_HOOK() as
                 * portPRE_TASK_DELETE_HOOK() does not return in the Win32 port. */
                traceTASK_DELETE( pxTCB );

                /* The pre-delete hook is primarily for the Windows simulator,
                 * in which Windows specific clean up operations are performed,
                 * after which it is not possible to yield away from this task -
                 * hence xYieldPending is used to latch that a context switch is
                 * required. */
                portPRE_TASK_DELETE_HOOK( pxTCB, &xYieldPending );
            }
            else
            {
                --uxCurrentNumberOfTasks;
                traceTASK_DELETE( pxTCB );
```

```
                /* Reset the next expected unblock time in case it referred to
                 * the task that has just been deleted. */
                prvResetNextTaskUnblockTime();
            }
        }
        taskEXIT_CRITICAL();

        /* If the task is not deleting itself, call prvDeleteTCB from outside of
         * critical section. If a task deletes itself, prvDeleteTCB is called
         * from prvCheckTasksWaitingTermination which is called from Idle task. */
        if( pxTCB != pxCurrentTCB )
        {
            prvDeleteTCB( pxTCB );
        }

        /* Force a reschedule if it is the currently running task that has just
         * been deleted. */
        if( xSchedulerRunning != pdFALSE )
        {
            if( pxTCB == pxCurrentTCB )
            {
                configASSERT( uxSchedulerSuspended == ( UBaseType_t ) 0U );
                portYIELD_WITHIN_API();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
    }
```

Meccanismi di sincronizzazione

I meccanismi di sincronizzazione in FreeRTOS sono piuttosto vari: in OS161 spinlock e semafori sono implementati, mentre manca il supporto per lock e condition variable; FreeRTOS invece offre implementazioni per i seguenti meccanismi:

- Semafori binari
- Semafori a conteggio
- Mutex
    - Ricorsivi
    - Priority Inheritance

Semafori binari

La funzione xSemaphoreCreateBinary crea un semaforo binario, ritornando un handle attraverso il quale si può far riferimento al semaforo Esso è simile ai mutex, ma non implementano il meccanismo di Priority Inheritance.

```
//semphr.h
#define xSemaphoreCreateBinary() xQueueGenericCreate( ( UBaseType_t ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH, q
ueueQUEUE_TYPE_BINARY_SEMAPHORE )

//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateBinary(void)
```

Crea un semaforo binario, ritornando un handle attraverso il quale si può far riferimento al semaforo. Simile ai mutex, ma non implementano il meccanismo di Priority Inheritance.

Si nota come, per implementare i semafori, freeRTOS adatta la struttura già esistente di `queue`, usata per la comunicazione tra processi.

L'istruzione presenta infatti una <Macro> che implementa un semaforo utilizzando il meccanismo di coda esistente: La lunghezza della coda è 1 perché si tratta di un semaforo binario, la dimensione dei dati è 0 (`semSEMAPHORE_QUEUE_ITEM_LENGTH`) poiché non vogliamo effettivamente memorizzare alcun dato, vogliamo solo sapere se la coda è vuota o piena.
Questo tipo di semaforo può essere utilizzato per una sincronizzazione pura tra task o tra un'interruzione e un task. Il semaforo non deve essere restituito una volta ottenuto, quindi un task/interruzione può continuamente "dare" il semaforo mentre un altro task/interruzione lo "prende" continuamente. Per questo motivo, questo tipo di semaforo non utilizza un meccanismo di ereditarietà delle priorità. Per un'alternativa che utilizza l'ereditarietà delle priorità, c'è `xSemaphoreCreateMutex()`.

Semafori a conteggio

```
// semphr.h
#define xSemaphoreCreateCounting( uxMaxCount, uxInitialCount )
         xQueueCreateCountingSemaphore( ( uxMaxCount ), ( uxInitialCount ) )
//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitCount)
```

Crea un semaforo di conteggio e restituisce un handle con il quale è possibile fare riferimento al semaforo appena creato.

- `uxMaxCount`: Massimo valore di conteggio
- `uxInitCount`: Valore di conteggio assegnato alla creazione

I semafori di conteggio sono tipicamente usati per due cose:

- Conteggio degli eventi.

In questo scenario di utilizzo un gestore di eventi "darà" un semaforo ogni volta che si verifica un evento (incrementando il valore del conteggio dei semafori) e un'attività del gestore "prenderà" un semaforo ogni volta che elabora un evento (decrementando il valore del conteggio dei semafori). Il valore di conteggio è quindi la differenza tra il numero di eventi che si sono verificati e il numero che sono stati elaborati. In questo caso è auspicabile che il valore di conteggio iniziale sia zero.

- Gestione delle risorse.

In questo scenario di utilizzo il valore di conteggio indica il numero di risorse disponibili. Per ottenere il controllo di una risorsa un'attività deve prima ottenere un semaforo - decrementando il valore del conteggio del semaforo. Quando il valore del conteggio raggiunge lo zero non ci sono risorse libere. Quando un'attività termina con la risorsa, "restituisce" il semaforo - incrementando il valore del conteggio del semaforo. In questo caso è auspicabile che il valore di conteggio iniziale sia uguale al valore di conteggio massimo, indicando che tutte le risorse sono libere.

Mutex

```
//semphr.h
#define xSemaphoreCreateMutex()    xQueueCreateMutex( queueQUEUE_TYPE_MUTEX )
//interfaccia effettiva
SemaphoreHandle_t xSemaphoreCreateMutex(void)
```

Crea un mutex, ritornando un handle attraverso il quale si può far riferimento al mutex
Esso è simile ai semafori binari, ma implementano il meccanismo di Priority Inheritance.

I mutex non possono essere utilizzati nelle routine di servizio interrupt.

Operazioni con i semafori

Per eliminare un semaforo, compresi i semafori di tipo mutex e i semafori ricorsivi, si utilizza la funzione vSemaphoreDelete:

```
//semphr.h
#define vSemaphoreDelete( xSemaphore )    vQueueDelete( ( QueueHandle_t ) ( xSemaphore))
//interfaccia effettiva
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore)
```

Non bisogna però eliminare un semaforo che ha attività bloccate su di esso (attività che sono nello stato Bloccato in attesa che il semaforo diventi disponibile).

Per ottenere un semaforo si utilizza la macro `xSemaphoreTake`. Il semaforo deve essere stato creato in precedenza con una chiamata a `xSemaphoreCreateBinary()`, `xSemaphoreCreateMutex()` o `xSemaphoreCreateCounting()`.

```
//semphr.h
#define xSemaphoreTake( xSemaphore, xBlockTime )    xQueueSemaphoreTake( ( xSemaphore ), ( xBlockTime ) )
//interfaccia effettiva
xSemaphoreTake(SemaphoreHandle_t xSemaphore,
TickType_t xTicksToWait)
```

- `xSemaphore`: Handle del semaforo da acquisire
- `xTicksToWait`: Tempo massimo di attesa affinché il semaforo diventi disponibile.

Per rilasciare un semaforo si utilizza invece la macro `xSemaphoreGive`. Anche in questo caso il semaforo deve essere stato creato in precedenza con una chiamata a `xSemaphoreCreateBinary()`, `xSemaphoreCreateMutex()` o `xSemaphoreCreateCounting()`.

```
//semphr.h
#define xSemaphoreGive( xSemaphore )    xQueueGenericSend( ( QueueHandle_t ) ( xSemaphore ), NULL, semGIVE
_BLOCK_TIME, queueSEND_TO_BACK )

//interfaccia effettiva
xSemaphoreGive(SemaphoreHandle_t xSemaphore)
```

- `xSemaphore`: Handle del semaforo da rilasciare

FreeRTOS offre anche la possibilità di creare un mutex ricorsivo con la funzione `xSemaphoreCreateRecursiveMutex.` Un mutex usato ricorsivamente può essere "preso" ripetutamente dal proprietario. Il mutex non diventa più disponibile fino a quando il proprietario non ha chiamato `xSemaphoreGiveRecursive()` per ogni richiesta `xSemaphoreTakeRecursive()` riuscita.

Condition Variables

Os161 offre un'interfaccia, ma non il supporto, per le condition variable. Sta quindi all'utilizzatore implementare le funzioni che garantiscono un corretto funzionamento di queste. In FreeRTOS, invece, il concetto di condition variable è del tutto assente.

Memoria virtuale e Memory Management Unit (MMU)

Siccome FreeRTOS è un sistema operativo real time, non dispone di MMU e memoria virtuale.

Mentre OS161 supporta la gestione della memoria virtuale e fornisce un'implementazione semplificata di un MMU con traduzione degli indirizzi virtuali a indirizzi fisici; FreeRTOS, essendo principalmente orientato a sistemi embedded, è senza supporto per la memoria virtuale o la gestione dell'MMU. Dunque esso non gestisce la traduzione degli indirizzi virtuali in fisici fornendo un meccanismo di allocazione contigua

Alla creazione di un task, semaforo o coda il kernel deve allocare memoria
Le standard malloc e free possono essere utilizzate ma:

- non sono sempre disponibili nei piccoli sistemi embedded
- Non sono deterministiche
- Possono soffrire di frammentazione della memoria
- La loro implementazione può essere ampia

FreeRTOS tratta l'allocazione della memoria come parte del suo portable layer

- Allocazione dinamica/statica dell'heap
- 5 differenti implementazioni disponibili per l'Heap Management
  - heap_1.c
    - Permette la sola allocazione della memoria
    - Usata quando le applicazioni non rimuovono mai risorse (task, semfori, code)
  - heap_2.c
    - Permette la deallocazione della memoria, ma non gestisce la frammentazione
    - Usata quando l'applicazione rimuove risorse continuamente
    - Da evitare quando la quantità di stack allocata ai task non è sempre la stessa
  - heap_3.c
    - Prevede dei wrapper della malloc() e free() del C per renderle thread-safe
  - heap_4.c
    - Come heap_2.c, ma prevede la gestione della frammentazione

    - heap_5.c
        - Come heap_4.c, ma permette di allocare l’heap in regioni non contigue di memoria
- Possibilità di definire la propria implementazione

Implementazione di diversi algoritmi di scheduling

Scheduling single-core (default)

OS/161 utilizza un algoritmo di scheduling a priorità fissa con una politica di esecuzione round-robin all'interno della stessa priorità. Il sistema operativo fornisce anche la possibilità di modificare la priorità dei processi durante l'esecuzione.

D’altra parte FreeRTOS utilizza un kernel di scheduling prioritario predefinito. I task vengono schedulati in base alla loro priorità, con la possibilità di gestire anche la preemption tra task con priorità diversa.

Per impostazione predefinita, FreeRTOS utilizza un criterio di scheduling preventivo a priorità fissa, con round-robin time-slicing di attività a parità di priorità:

- "Priorità fissa" significa che lo scheduler non cambierà la priorità di un'attività, anche se potrebbe aumentarla temporaneamente a causa del Priority Inheritance. Ad esempio quando un task acquisisce un mutex gli viene assegnata temporaneamente la priorità massima in quanto non deve essere swappato.
- "Preventivo” significa che lo scheduler esegue sempre l'attività RTOS con la massima priorità che è in grado di eseguire, indipendentemente da quando quest'attività è in grado di essere eseguita. Ad esempio, se una routine di servizio interrupt (ISR) modifica l'attività con la priorità più alta , lo scheduler interromperà l'attività a priorità inferiore attualmente in esecuzione e avvierà l'attività con priorità più alta, anche se si verifica all'interno di un time slice.
- "Round-robin" significa che le attività che condividono una priorità entrano a turno nello stato di esecuzione.
- "Time sliced" significa che lo scheduler passerà tra attività di uguale priorità su ogni interruzione del tick - il tempo tra gli interrupt del tick è una fetta di tempo. (Il tick interrupt è l'interrupt periodico utilizzato dall'RTOS per misurare il tempo.)

Scheduling multi-core

La politica di scheduling di freeRTOS può essere simmetrica o asimmetrica:

- La politica di pianificazione di FreeRTOS AMP
    - Il multiprocessing asimmetrico (AMP) con FreeRTOS è il luogo in cui ogni core di un dispositivo multicore esegue la propria istanza indipendente di FreeRTOS
    - I core non devono avere tutti la stessa architettura, ma devono condividere un po' di memoria se le istanze FreeRTOS devono comunicare tra loro.
    - Ogni core esegue la propria istanza di FreeRTOS, quindi l'algoritmo di pianificazione su un dato core è esattamente come descritto sopra per un sistema single-core.
    - È possibile utilizzare un flusso o un buffer di messaggi come primitiva di comunicazione inter-core in modo che le attività su un core possano entrare nello stato Bloccato per attendere che i

dati o gli eventi vengano inviati da un core diverso.

- La politica di pianificazione FreeRTOS SMP
  - Il multiprocessing simmetrico (SMP) con FreeRTOS è dove un'istanza di FreeRTOS pianifica le attività RTOS su più core del processore.
  - Poiché c'è solo un'istanza di FreeRTOS in esecuzione, è possibile utilizzare solo una porta di FreeRTOS alla volta, quindi ogni core deve avere la stessa architettura del processore e condividere lo stesso spazio di memoria.
  - Il criterio di scheduling SMP utilizza lo stesso algoritmo del criterio di scheduling single-core ma, a differenza degli scenari single-core e AMP, SMP comporta che più di un'attività sia nello stato di esecuzione in un dato momento (c'è un'attività di stato di esecuzione per core).
  - Ciò significa che l'ipotesi che un'attività a priorità inferiore verrà eseguita solo quando non ci sono attività a priorità superiore in grado di eseguire non vale più.

Caratteristiche fondamentali

FreeRTOS

- Kernel real-time
- Multitasking prioritario
- Modello di esecuzione cooperativo
- Supporto per processori a 32 bit e a 8 bit
- Gestione di attività con priorità fissa o variabile
- Supporto per i semafori binari e di conteggio
- Supporto per le code
- Supporto per i timer software e hardware
- Architettura modulare

OS161

- Architettura del kernel monolitica
- Progettato per scopi didattici
- Basato su UNIX v6 e gli standard POSIX
- Supporta l'architettura MIPS
- Implementa le chiamate di sistema e la gestione dei processi semplice
- Fornisce un supporto di base per il file system
- Architettura modulare
- Implementa la gestione della memoria virtuale utilizzando una versione semplificata della tecnica di paging a richiesta

Implementazione delle condition variables

Abbiamo creato un file condvar.h in `FreeRTOS>Source>include`

In questo file, oltre ad includere inizialmente le varie intestazioni come “FreeRTOS.h”, "semphr.h" e "queue.h”, viene definito un alias `SemaphoreHandle_t` per `QueueHandle_t`. Questo è comune in FreeRTOS, dove i semafori e le code condividono la stessa struttura dati di base:

```
typedef QueueHandle_t SemaphoreHandle_t;
```

Viene definita una struttura che rappresenta un semaforo condizionale personalizzato. Contiene due campi: mutex e coda, entrambi di tipo SemaphoreHandle_t. Il campo mutex è utilizzato per proteggere l'accesso alla coda condizionale, mentre coda rappresenta la coda condizionale stessa.

```
struct CondVarHandle_t{
SemaphoreHandle_t mutex;
SemaphoreHandle_t coda;
}
```

Inoltre definiamo anche i seguenti parametri:

```
#define semCONDVAR_SEMAPHORE_QUEUE_LENGTH    ( ( uint8_t ) 1U )
#define semSEMAPHORE_QUEUE_ITEM_LENGTH      ( ( uint8_t ) 0U )
#define semGIVE_BLOCK_TIME                  ( ( TickType_t ) 0U )
```

Rispettivamente la lunghezza massima della coda, la lunghezza massima degli elementi nella coda e il tempo di blocco per l’operazione di give.

Creazione di un semaforo condizionale

Quando il supporto alla gestione dinamica della memoria è abilitato, la funzione `xCondVarCreate()` viene utilizzata per creare un oggetto di tipo CondVarHandle_t, inizializzato tramite le funzioni `xSemaphoreCreateMutex()`, per creare il semaforo mutex, e `xSemaphoreCreateCounting()` per creare la coda condizionale

```
#if ( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
#define xCondVarCreate()
    {
        struct CondVarHandle_t xCondVar;
        (xCondVar->mutex) = xSemaphoreCreateMutex();
        (xCondVar->coda) = xSemaphoreCreateCounting(1,0);

           return xCondVar
    }
#endif
```

Se `(xCondVar) != NULL` la funzione di give viene utilizzata per dare il semaforo condizionale

Nel caso in cui è abilitato il supporto alla gestione statica della memoria, utilizziamo la funzione `xCondVarCreateBinaryStatic` così definita:

```
#if (configSUPPORT_STATIC_ALLOCATION == 1)

#define xCondVarCreateStatic(pxStaticCondVar)
{
    CondVarHandle_t cv;
    cv.mutex = xSemaphoreCreateMutexStatic(pxStaticCondVar.mutex);
    cv.coda = xSemaphoreCreateCountingStatic(1,0,pxStaticCondVar.coda);
    return cv;
}
```

La variabile `cv.mutex` viene inizializzata con un semaforo mutex statico creato dalla funzione `xSemaphoreCreateMutexStatic`, utilizzando la struttura `pxStaticCondVar.mutex` passata come argomento. Questo semaforo mutex sarà utilizzato per sincronizzare l'accesso alla coda condizionale. Invece `cv.coda` viene inizializzata con un semaforo di conteggio statico creato dalla funzione `xSemaphoreCreateCountingStatic`, utilizzando la struttura `pxStaticCondVar.coda` passata come argomento. Questo semaforo di conteggio rappresenterà la coda condizionale binaria.

La variabile `pxStaticCondVar` contiene le informazioni statiche necessarie per la creazione dei semafori statici.

Infine, la macro return restituisce la variabile `cv`, che contiene i riferimenti ai semafori mutex e di conteggio inizializzati, rappresentando così il semaforo condizionale binario.

Attendere un semaforo condizionale

```
 // Assuming you have already the possession of the mutex

#define xCondVarWaitCondition(xCondVar, xBlockTime)
{
    xSemaphoreGive(xCondVar->mutex);
    xSemaphoreTake(xCondVar->coda, xBlockTime);
    xSemaphoreTake(xCondVar->mutex, xBlockTime);
}
```

La funzione `xCondVarWaitCondition` attende una condizione specifica all'interno di un semaforo condizionale. Prende come argomenti `xCondVar`, il semaforo condizionale, e `xBlockTime`, il tempo massimo di attesa.

Questa funzione rilascia il mutex utilizzando la funziona già definita in semphr.h `xSemaphoreGive.` Questa macro quindi assume che l'attività abbia già acquisito questo mutex in precedenza.

Con la funzione `xSemaphoreTake` acquisisce il semaforo associato alla coda e riacquisisce il mutex associato al semaforo condizionale dopo che la condizione all'interno del semaforo è stata soddisfatta.

Nel caso bisogni attendere una condizione specifica all'interno di un semaforo condizionale assumendo che l'attività in esecuzione sia un servizio di interruzione (ISR) viene utilizzata la funzione `xCondVarWaitConditionFromISR` così definita:

```
#define xCondVarWaitConditionFromISR(xCondVar, pxHigherPriorityTaskWoken) \
{ \
    xSemaphoreGiveFromISR(xCondVar->mutex, pxHigherPriorityTaskWoken); \
```

```
    xSemaphoreTakeFromISR(xCondVar->coda, pxHigherPriorityTaskWoken); \
    xSemaphoreTakeFromISR(xCondVar->mutex, pxHigherPriorityTaskWoken); \
} // xQueueReceiveFromISR( ( QueueHandle_t ) ( xSemaphore ), NULL, ( pxHigherPriorityTaskWoken ) )
```

Si assume anche in questo caso che il mutex associato al semaforo condizionale sia già stato acquisito

La funzione prende come argomento il semaforo condizionale `xCondVar` e `pxHigherPriorityTaskWoken`, un puntatore alla variabile booleana che viene utilizzata per indicare se l'interruzione ha risvegliato una task di priorità superiore, utilizzato in contesti di ISR per gestire la priorità delle task.

Essa rilascia il mutex associato al semaforo con `xSemaphoreGiveFromISR`.

Poi tenta di acquisire il semaforo di conteggio associato alla coda condizionale chiamando `xSemaphoreTakeFromISR(xCondVar->coda, pxHigherPriorityTaskWoken)`. L'ISR aspetterà finché il semaforo non sarà disponibile.

Successivamente chiama nuovamente `xSemaphoreTakeFromISR(xCondVar->mutex, pxHigherPriorityTaskWoken)` per riacquisire il mutex associato al semaforo condizionale.

Rilasciare il semaforo condizionale

Per notificare o sbloccare altre attività in attesa all'interno di un semaforo condizionale viene definita la funzione `xCondVarNotify`.

```
// Assuming you have already the possession of the mutex

#define xCondVarNotify(xCondVar)
{
    xSemaphoreGive(xCondVar->coda);
    xSemaphoreGive(xCondVar->mutex);
}
```

Questa funzione prende come argomento il semaforo condizionale e rilascia il semaforo di conteggio chiamando `xSemaphoreGive(xCondVar->coda)`.
Il rilascio di questo semaforo indica che una condizione specifica all'interno del semaforo condizionale è stata soddisfatta, consentendo alle attività in attesa di procedere.

Tramite la `xSemaphoreGive(xCondVar->mutex)` viene rilasciato anche il mutex

Nel caso bisogni sbloccare un'attività all'interno di un semaforo condizionale assumendo che l'attività in esecuzione sia un servizio di interruzione (ISR) viene utilizzata la funzione `xCondVarNotifyFromISR` così definita:

```
#define xCondVarNotifyFromISR( xCondVar, pxHigherPriorityTaskWoken )
 {
    xSemaphoreGiveFromISR(xCondVar->coda);
    xSemaphoreGiveFromISR(xCondVar->mutex);
}
```

Eliminare una condition variable

Per questo scopo viene progettata la macro **`vCondVarDelete`**:

```
#define vCondVarDelete(xCondVar)
{
    vSemaphoreDelete(xCondVar->mutex);
    vSemaphoreDelete(xCondVar->coda);
}
```

Prende come argomento `xCondVar`,la struttura `CondVarHandle_t` che rappresenta il semaforo condizionale personalizzato da eliminare.

Inizia chiamando `vSemaphoreDelete(xCondVar->mutex)` che elimina il mutex associato al semaforo condizionale, successivamente, la macro chiama `vSemaphoreDelete(xCondVar->coda)` per eliminare il semaforo di conteggio associato alla coda condizionale