



W • S E N S E
INTEGRATED CABLELESS SOLUTIONS



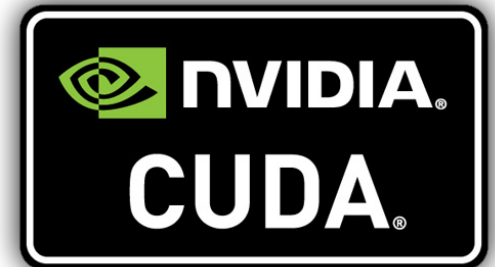
SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione di sistemi multicore Seconda Parte

Fabrizio Gattuso

Cosa
vedremo
nella
seconda
parte

amazon



NVIDIA.COM



SAPIENZA
UNIVERSITÀ DI ROMA

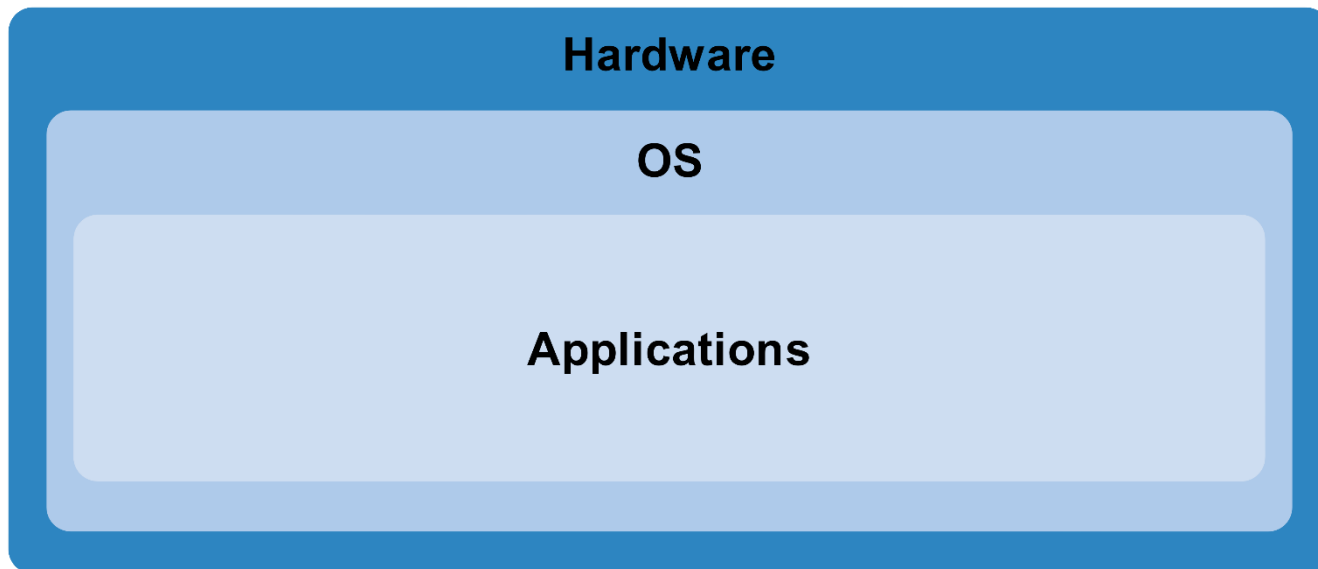
Programmazione di sistemi multicore – FreeRTOS
Fabrizio Gattuso



W • S E N S E
INTEGRATED CABLELESS SOLUTIONS

Che cos'è un OS?

Un Sistema Operativo (OS) è un software che dà accesso all'hardware sottostante e fornisce funzionalità utili allo sviluppo di applicazioni di livello superiore.



Multitasking e scheduler

La maggior parte dei sistemi operative permette l'esecuzione di più programmi in contemporanea, simulando così il concetto di parallelismo.

In realtà un solo task è eseguibile da un singolo core. Questo concetto è detto **multitasking**.

Ogni OS è basato su uno **scheduler** che implementa le politiche su come gestire i singoli task.

Multitasking vs Multithreading

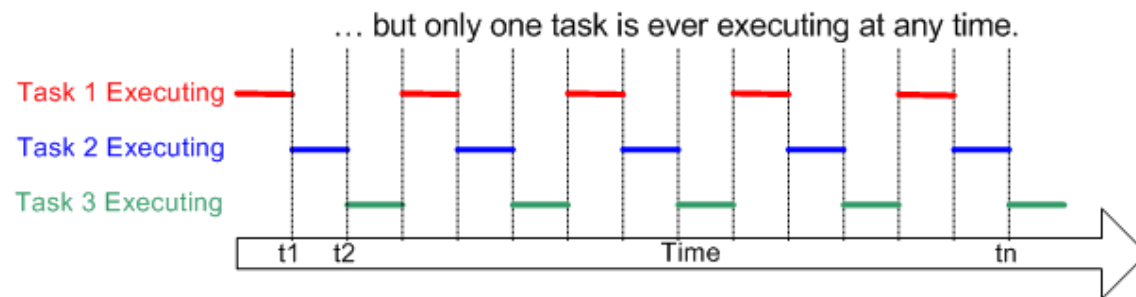
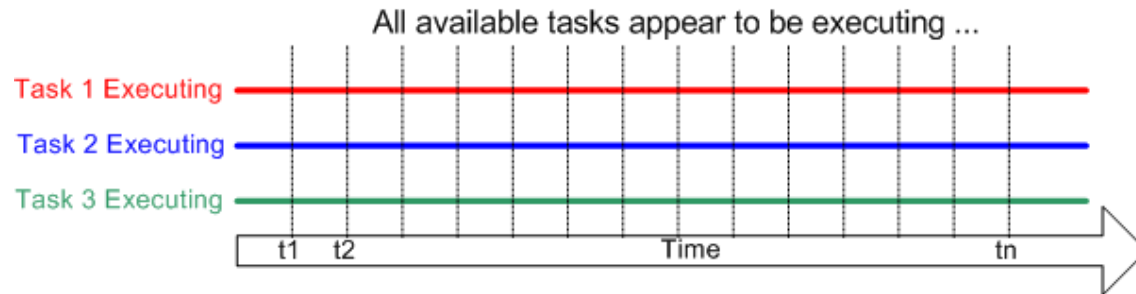
Un thread è una porzione di software che svolge una specifica funzione di un software più complesso (task).

Questo implica che il multithreading è più semplice da gestire perché richiede meno risorse (come ad esempio la memoria).

Il multithreading non è equivalente al multitasking.

RTOS

Un Real Time Operating System (RTOS) è un sistema operativo progettato per fornire un modello di esecuzione predicibile (**deterministico**).



Perché un RTOS

Un OS realtime è essenziale in tutti quegli scenari in cui il fattore tempo è cruciale (hard real time).

- **Hard real time:** è sempre necessario un tempo di esecuzione **massimo** garantito
- **Soft real time:** è necessario un tempo di esecuzione **medio**

Scheduler: problemi e soluzioni

Lo scheduler fa uso della tecnica del ***preemptive fixed priority scheduling***.

Ad ogni task è assegnato una priorità fissa e l'algoritmo cerca di dare più tempo possibile ai task con priorità più alta.

Questo potrebbe portare a problemi? Quali?

Priority inversion problem

Nel 1997 la **NASA** ha inviato il suo primo spacecraft su Marte (JPL Pathfinder).

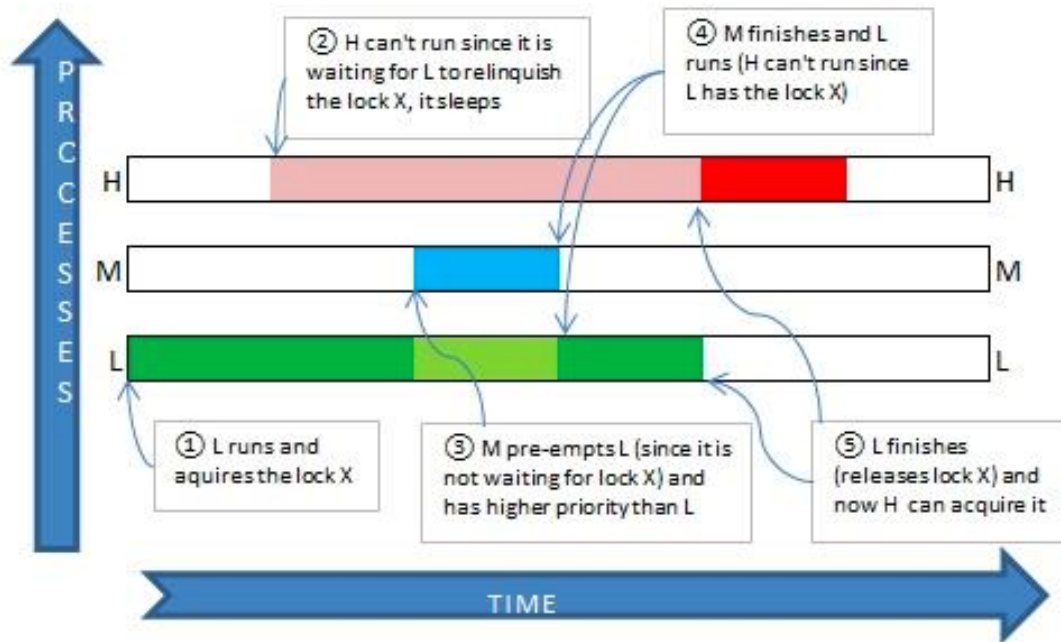
Dopo le prime ore di missione, a causa dell'intervento del watchdog, il pathfinder cominciò a riavviarsi da solo.

Il problema fu indentificato nella scorretta gestione di due task concorrenti da parte del RTOS. Il problema è noto come il **priority inversion problem**.

Priority inversion problem (2)

This is what happened !!

Priority (H) > Priority (M) > Priority (L)



Un task con priorità più bassa ha bloccato un task con priorità più alta a causa delle risorse dipendenti.



Priority inversion problem (3)

Questo problema noto è risolvibile con la tecnica della **priority inheritance**.

Ai task che acquisiscono risorse bloccati tramite mutex e semafori viene **assegnata una priorità alta** in modo da uscire il prima possibile dalla sezione critica. È essenziale che la priorità venga assegnata il **più breve tempo possibile** per non ricadare nello stesso problema.

Che cos'è FreeRTOS?

FreeRTOS è una classe di RTOS sviluppati per essere eseguiti su microcontrollori.

FreeRTOS fornisce:

1. un realtime scheduler
2. primitive di sincronizzazione e temporizzazione
3. comunicazione tra processi.

In realtà è più corretto descrivere FreeRTOS come un kernel con funzionalità di scheduling.

Perché FreeRTOS?

- Free sotto licenza MIT
- Qualità del software controllata
- Robusto e supportato
- Comunità utenti sempre più grande con molti esempi
- Portabile su molte piattaforme hardware



amazon



**ORA IL PROGETTO È
PORTATO AVANTI DA AMAZON**



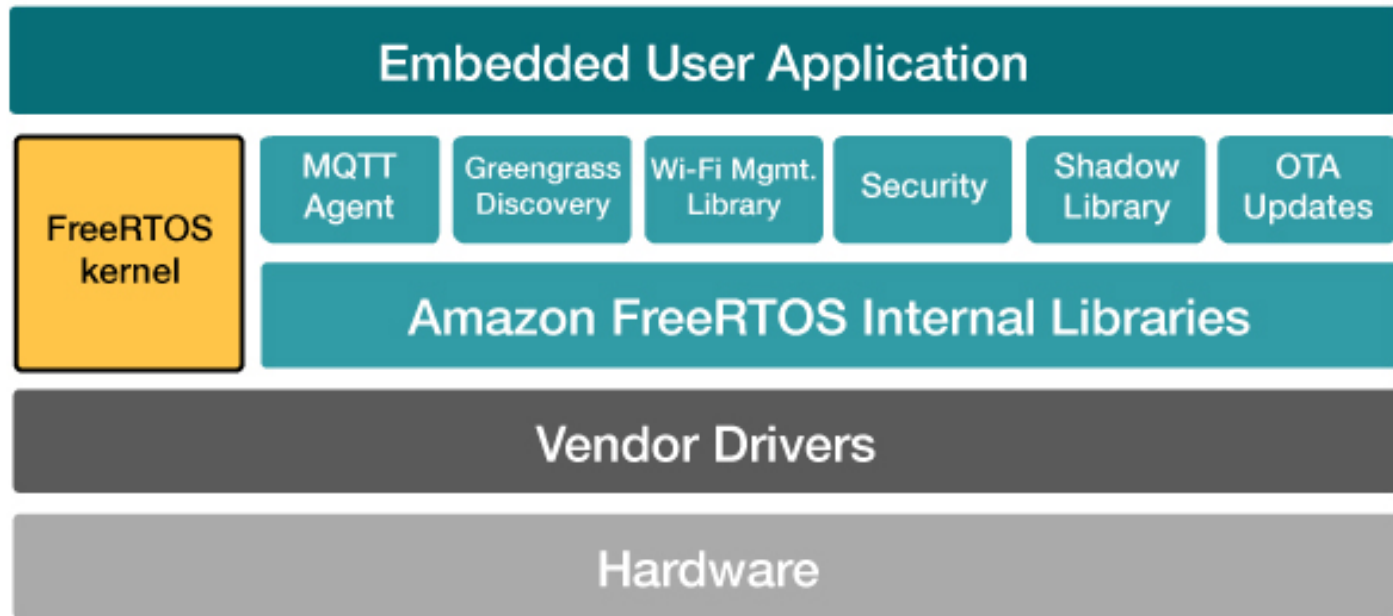
SAPIENZA
UNIVERSITÀ DI ROMA

Programmazione di sistemi multicore – FreeRTOS
Fabrizio Gattuso



W • S E N S E
INTEGRATED CABLELESS SOLUTIONS

FreeRTOS vs AMAZON FRTOS



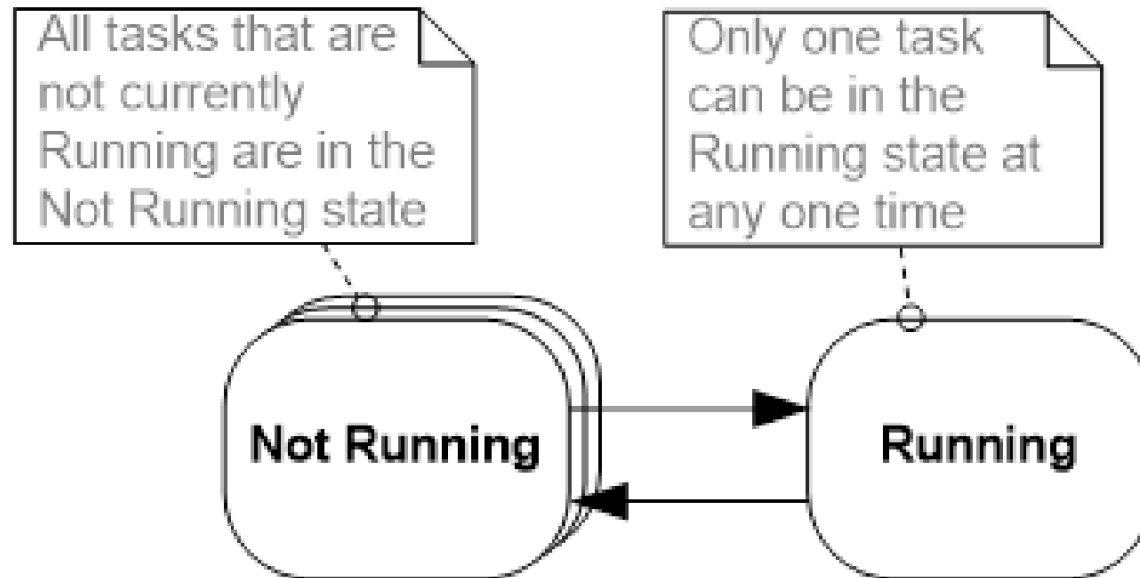
Gestione dei task

I task sono funzioni C, che ritornano void e prendere come parametro un puntatore a void:

```
void ATaskFunction( void *pvParameters );
```

Normalmente sono pesante come task infiniti e sono implementati grazie a *while(1)* o *for(;;)*

Stati di un task



STM32CUBEIDE

Per programmare i nodi avrete bisogno del software STM32CUBEIDE per Windows, Mac e Linux

<https://www.st.com/en/development-tools/stm32cubeide.html>

Allocazione della memoria

Può essere gestita in modo static o dynamic.

La memoria dinamica è il sistema più flessibile ma non verrà utilizzata in questo corso.

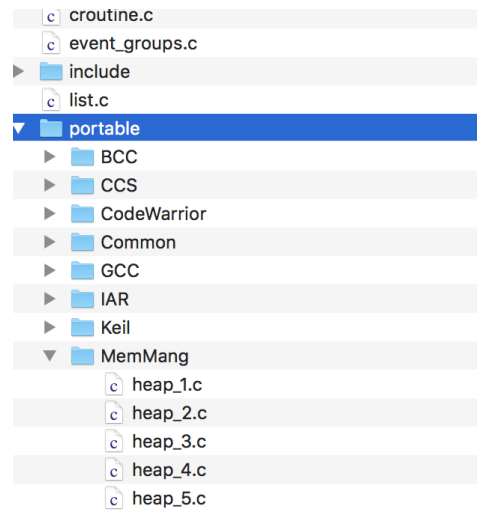
L'allocazione di memoria statica è più robusta ed utilizzata nelle applicazioni in cui livello di affidabilità è molto alto.

Heap System

Come è stato accennato useremo la memoria statica per il corso ma è bene conoscere le altre alternative.

- Heap 1
- Heap 2
- Heap 3
- Heap 4
- Heap 5

FreeRTOS/Source/portable/MemMang

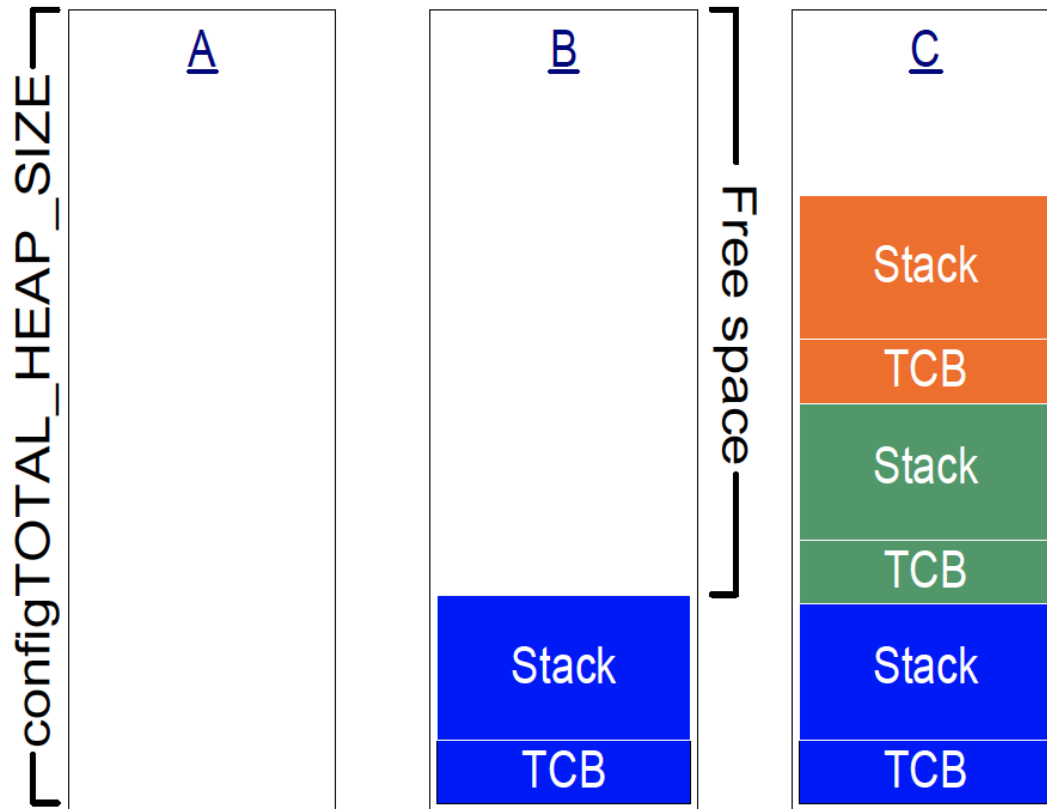


Heap 1

Lo schema più semplice. **Una volta che la memoria è stata allocata non può più essere deallocata.**

La memoria è divisa in diversi blocchi. La memoria totale è configurata tramite **configTOTAL_HEAP_SIZE** – definito in **FreeRTOSConfig.h**.

Heap 1 (2)



Heap 1 (3)

Usabile in applicazioni che:

- Non hanno bisogno di cancellare task, queue, semaphore, mutex, etc. (la maggioranza dei casi)
- Determinista e non porta alla frammentazione della memoria
- Applicazioni che non fanno un vero uso dell'allocazione dinamica della memoria

Heap 2

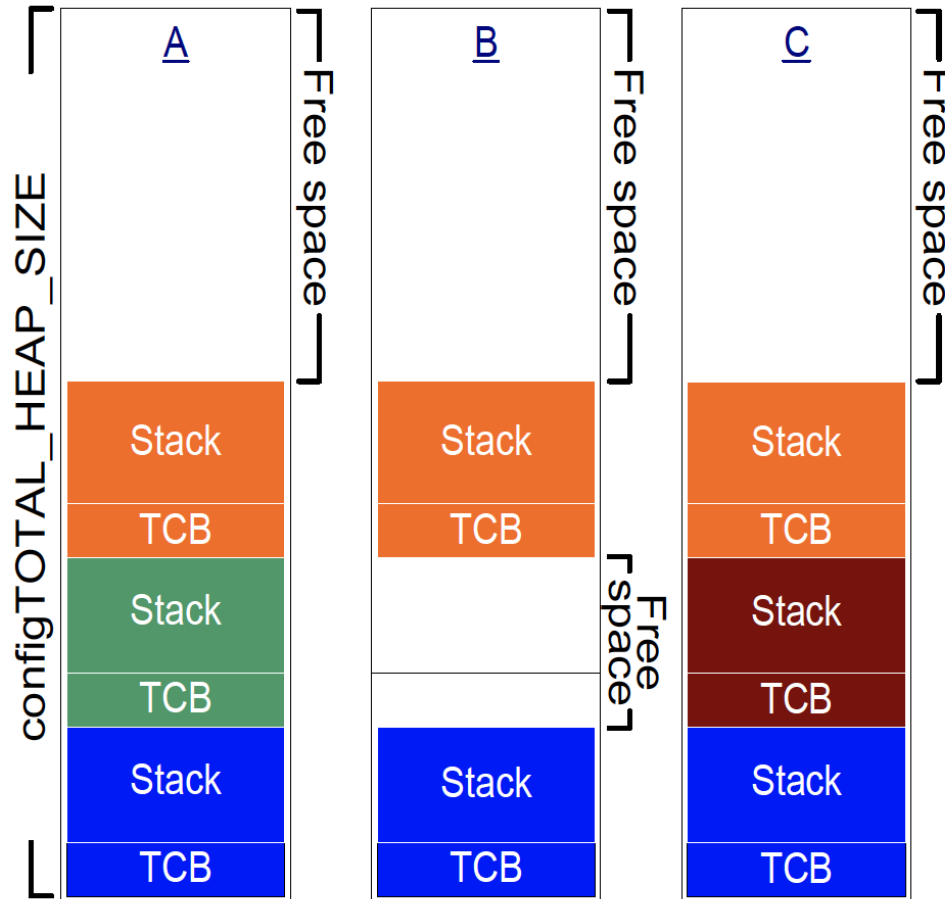
Fa uso del **best fit algorithm** e permette che la memoria sia deallocata ma non combina gli spazi di memoria libera tra loro. Potrebbe portare a frammentazione della memoria.

La memoria totale è configurata tramite **configTOTAL_HEAP_SIZE** – definito in **FreeRTOSConfig.h**.

xPortGetFreeHeapSize() ritorna la memoria libera



Heap 2 (2)



Usabile quando l'applicazione richiede l'eliminazione continua di tasks, queues, semaphores, mutexes, etc. ma attenzione alla frammentazione della memoria.

Heap 3

Un wrapper per le funzioni standard C **malloc()** e **free()** rendendole thread-safe ma non deterministiche.

Porta all'aumento della memoria utilizzata dal kernel (OS).

Non è consigliata per applicazioni real-time.

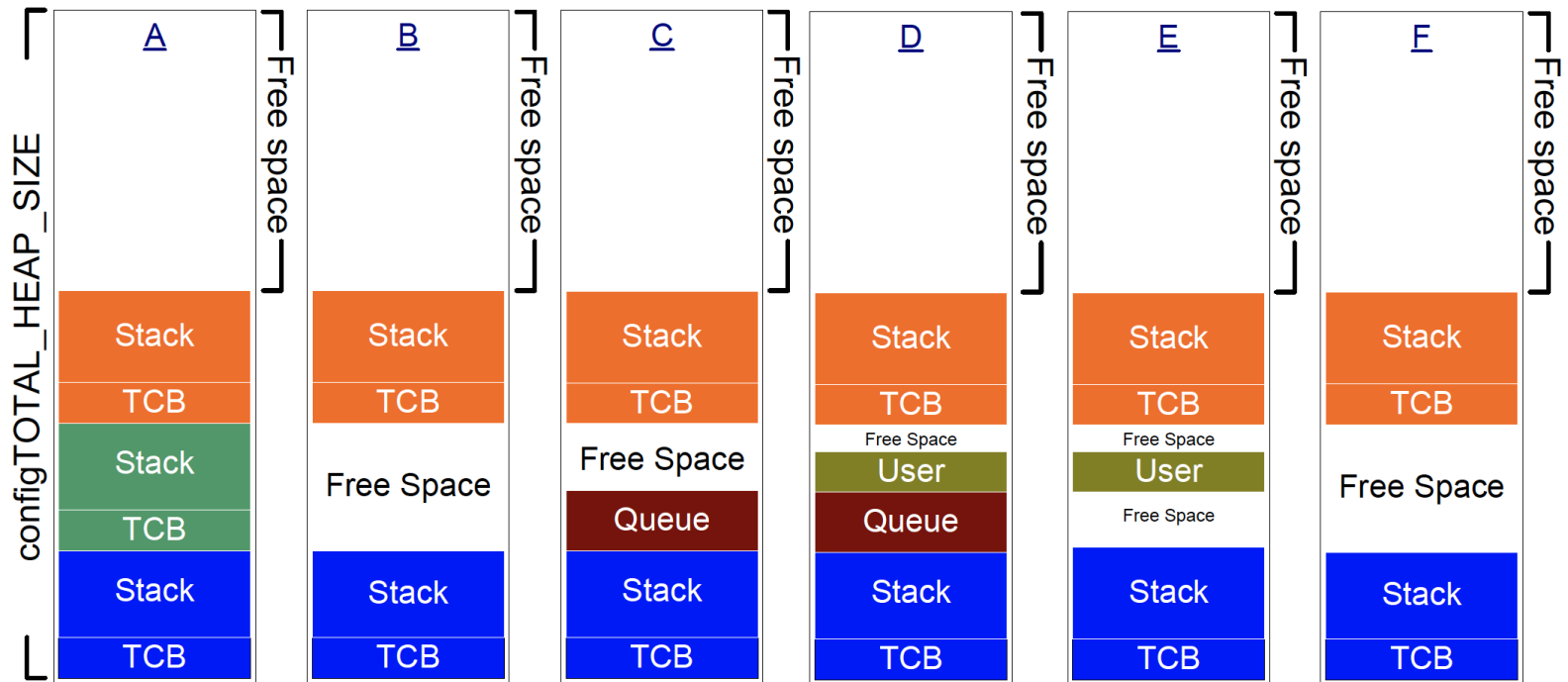
Heap 4

Utilizza il **first fit algorithm** e **combina gli spazi di memoria vuota in un unico spazio di memoria libero.**

Evita la frammentazione.

Continua a non essere determinista ma è più efficiente dell'implementazione standard C.

Heap 4 (2)



Dynamic Memory Allocation API

È possibile utilizzare le librerie C standard per allocare memoria ma:

non sono deterministiche, raramente thread-safe, non sempre disponibili su sistemi embedded ridotti

`malloc()` -> `pvPortMalloc()`

`free()` -> `vPortFree()`

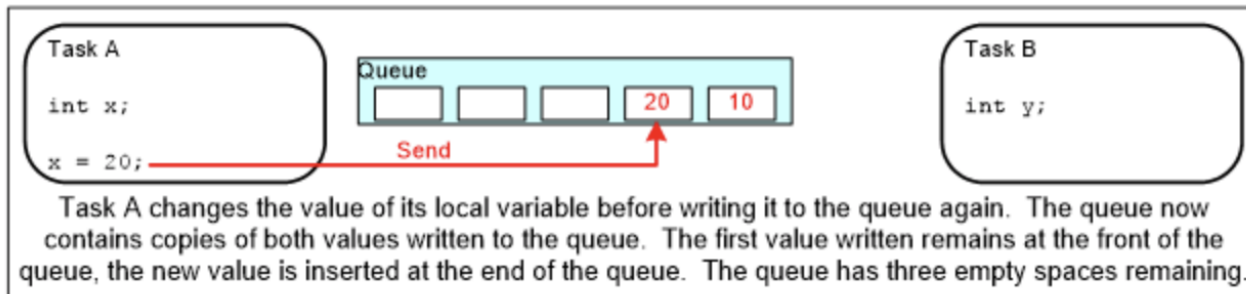
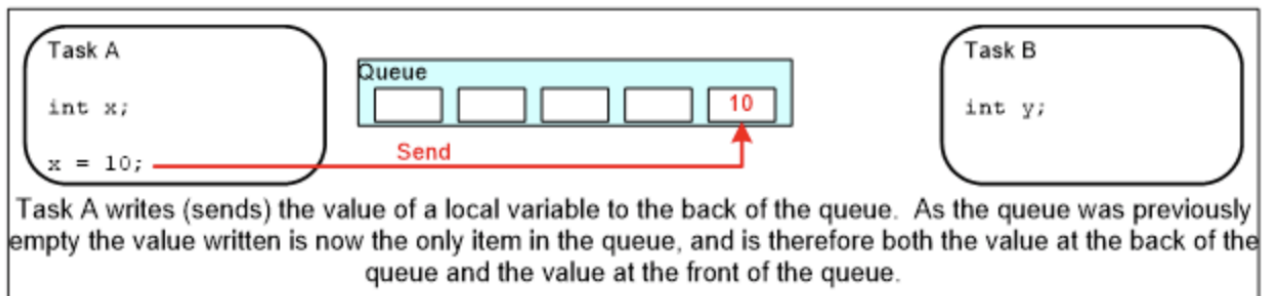
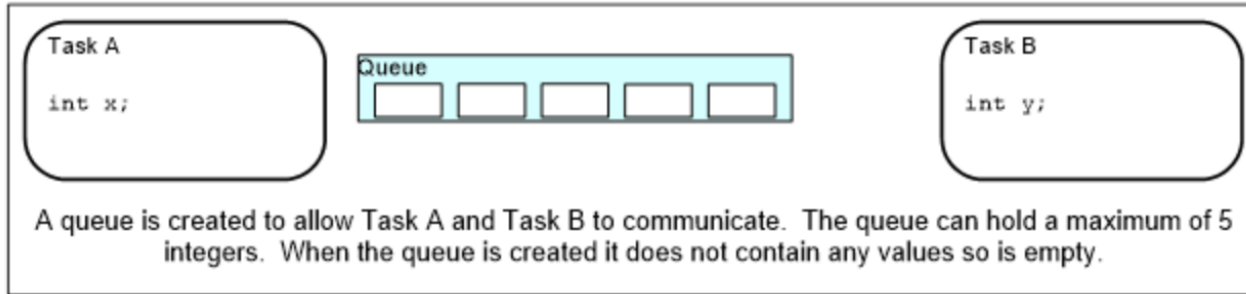
Queue

Le code sono utilizzate come mezzo di comunicazione:

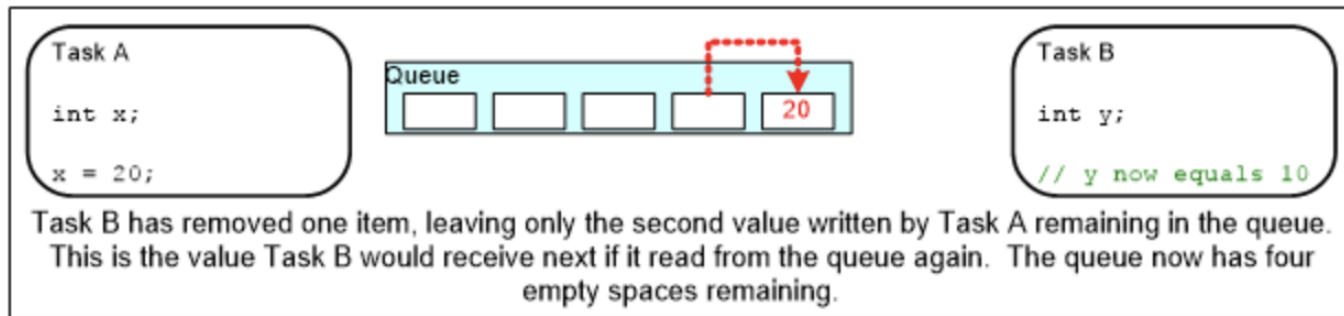
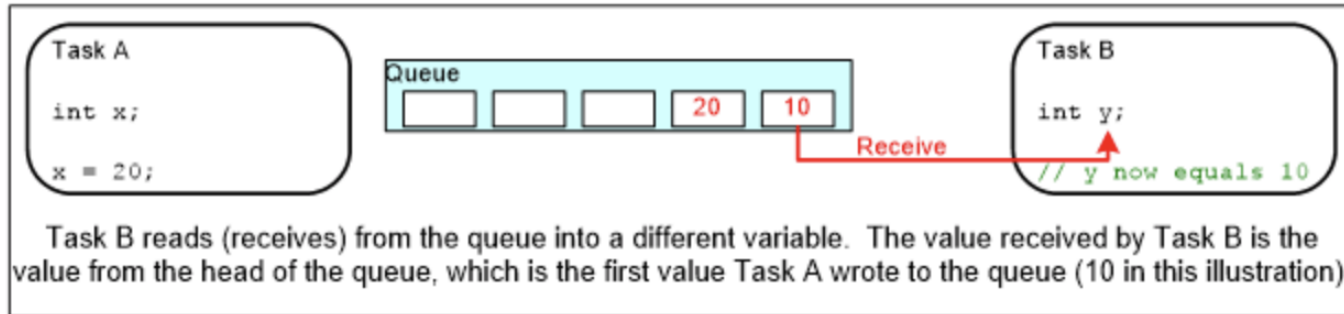
1. Tra task
2. Da task a interrupt (e viceversa)

Una coda può contenere un **numero finito e di dimensione fissa di elementi**. Il numero massimo di elementi è chiamato **length**. La lunghezza della coda e la dimensione del singolo elemento sono fissati in fase di creazione.

Queue (2)



Queue (3)



Le code funzionano copiando un elemento nella propria memoria (no referenze)

Queue (4)

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,  
                             UBaseType_t uxItemSize );
```

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,  
                               const void * pvItemToQueue, TickType_t xTicksToWait );
```

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
                              const void * pvItemToQueue, TickType_t xTicksToWait );
```

```
BaseType_t xQueueReceive( QueueHandle_t xQueue, void * const pvBuffer,  
                           TickType_t xTicksToWait );
```



Queue con puntatori

Quando la dimensione dei dati da salvare è notevole è possibile passare un puntatore alla coda. Questo introduce delle difficoltà:

- La memoria non va mai modificata simultaneamente rendendo la memoria **inconsistente**.
- Se la memoria è allocata dinamicamente allora un task deve essere responsabile della **deallocazione** di essa.
- Non bisogna mai passare puntatori a della memoria allocata all'interno del singolo task. La memoria verrà liberata se il task verrà eliminato.

Software timer

I software timer sono utilizzati per schedulare delle operazioni nel futuro.

Non bisogna confonderli con i timer hardware ma non utilizzano nessuno ciclo di clock per gestire l'attesa.

È necessario includere il file `<timer.c>` nel proprio progetto e configurare la variabile `configUSE_TIMERS = 1` nel file `FreeRTOSConfig.h`

Mai utilizzare codice bloccante all'interno di un Timer!

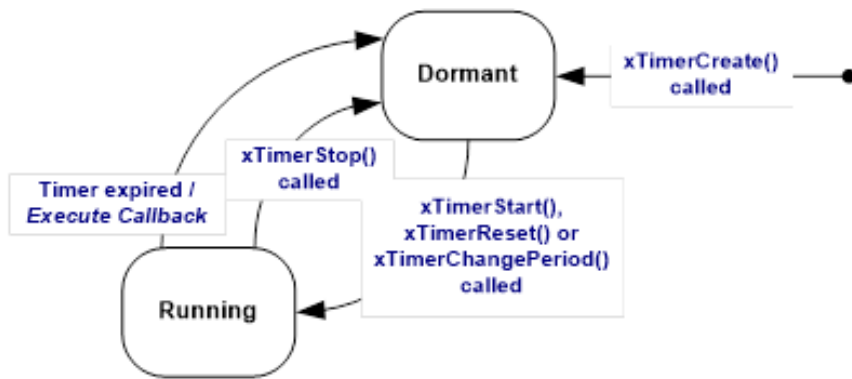
Software timer (2)

- **One shot**

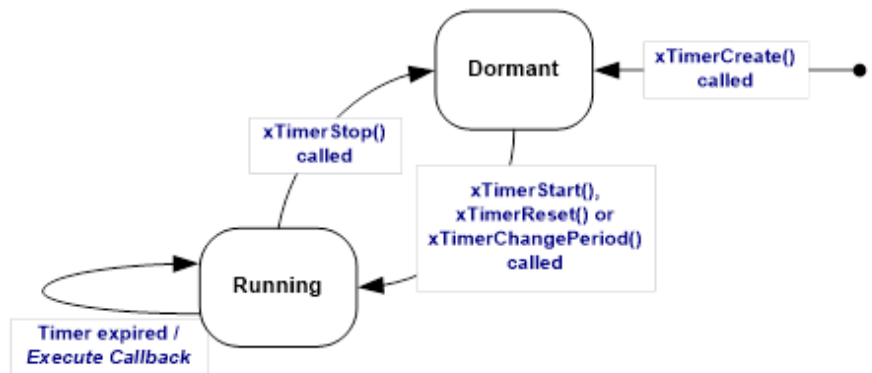
Questo tipo di timer vengono eseguiti **una sola volta**. Un one-shot timer può essere riavviato manualmente.

- **Auto-reloaded**

Un timer di questo genere viene rieseguito a cadenza prefissata chiamando la sua callback function.



One Shot



Auto-reloaded

Software timers (3)

```
TimerHandle_t xTimerCreate( const char * const pcTimerName,  
                             TickType_t xTimerPeriodInTicks,  
                             UBaseType_t uxAutoreload, void * pvTimerID,  
                             TimerCallbackFunction_t pxCallbackFunction);
```

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

```
TimerHandle_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

```
TimerHandle_t xTimerStop( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

```
TimerHandle_t xTimerChangePeriod( TimerHandle_t xTimer,  
TickType_t xNewTimerPeriodInTicks, TickType_t xTicksToWait );
```