FONDAMENTI DI INFORMATICA

Prof. PIER LUCA MONTESSORO
Laureando LUCA DA RE

Facoltà di Ingegneria
Università degli Studi di Udine

I Sistemi Operativi Real-Time

Nota di Copyright

Sistemi real-time

- I dispositivi di controllo interagiscono con il mondo fisico, quindi devono necessariamente farlo in tempo reale
- Un sistema di elaborazione dati in tempo reale deve rispondere in modo certo ed entro un intervallo di tempo fissato ad eventi esterni non prevedibili

Sistemi real-time

- Real-time non è sinonimo di veloce, bensì di deterministico
- Non è importante che il tempo di tale esecuzione sia più piccolo possibile: deve essere esattamente il tempo prestabilito

Sistemi real-time

- Due tipologie di correttezza devono essere garantite:
  - Correttezza logica: i risultati/risposte forniti devono essere quelli previsti (normalmente richiesta a qualunque sistema di elaborazione)
  - Correttezza temporale: i risultati devono essere prodotti entro certi limiti temporali fissati chiamati deadlines (specifica dei sistemi real-time)

Definizioni di real-time

- *IEEE 610.12 (1990)*: il software R.T. appartiene ad un sistema o una modalità operativa la cui computazione è effettuata mentre il tempo di un processo esterno procede, affinché il calcolo possa essere usato per controllare, monitorare, o rispondere in tempi adeguati agli stimoli esterni
- Un sistema in cui la correttezza non dipende solo dai risultati logici del calcolo ma anche da quando essi sono disponibili
- Un sistema che risponde in modo predicibile (temporalmente) all'arrivo di stimoli imprevedibili
- Un sistema R.T. è un sistema interattivo che mantiene una relazione esterna con un ambiente asincrono in modalità non cooperativa

Tipi di sistemi real-time

- I sistemi real-time possono essere distinti secondo gli eventi che controllano:
  - Event-driven: reattivi od a evento, in cui si ha l'arrivo di stimoli esterni ad istanti casuali
  - Time-driven: periodici, in cui le azioni sono cadenzate dallo scadere di periodi predeterminati

Tipi di sistemi real-time

- Sono anche classificati in base alla loro reazione agli eventi:
  - Sistemi Hard real-time:
    - il non rispetto delle deadlines temporali NON è ammesso
    - porterebbe al danneggiamento del sistema (applicazioni safety critical)
  - Sistemi Soft real-time:
    - il non rispetto occasionale delle deadlines è ammissibile
    - le specifiche temporali indicano solo in modo sommario i tempi da rispettare
    - degrado delle performance accettabile

Cos'è un Sistema Operativo

- È un'insieme di elementi software che creano un'interfaccia tra l'hardware e i programmi utente
- Il Sistema Operativo si assume la gestione delle varie risorse, garantendone il corretto utilizzo
- Il calcolatore viene visto come un erogatore di servizi, dove ognuno corrisponde all'esecuzione di un proprio programma

Caratteristiche di un Sistema Operativo

- Un Sistema Operativo esegue varie operazioni di controllo:
  - Gestire l'interfaccia verso l'hardware sottostante
  - Schedulare l'esecuzione dei programmi
  - Gestire la memoria
  - Gestire l'accesso verso i dispositivi comuni:
    - tastiera, video, dispositivi di puntamento, stampanti

Caratteristiche di un Sistema Operativo Real-Time

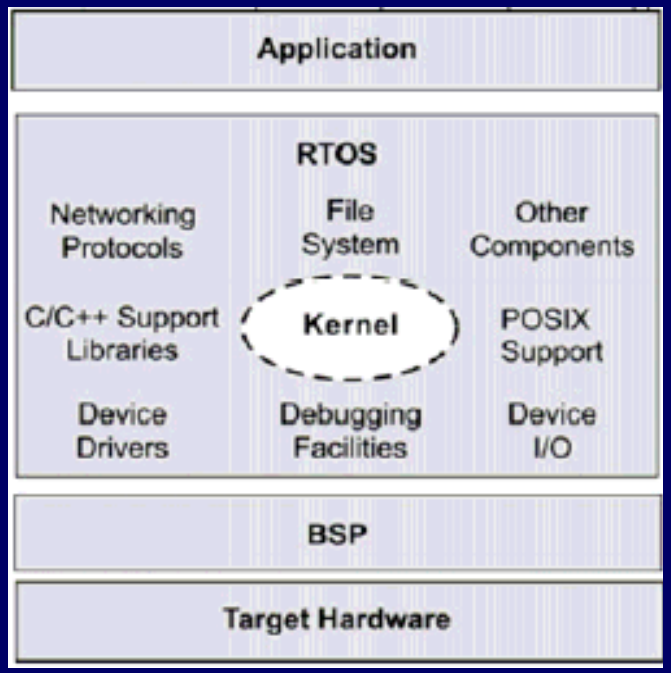
- Un Sistema Operativo Real-Time deve anche:
  - Avere una politica di schedulzione dei processi che preveda l'assegnazione di priorita'
  - Essere multitasking preemptive: deve essere prevista la possibilita' di interrompere un processo per assegnare le risorse ad un'altro a maggiore priorita' che ne ha fatto richiesta
  - Evitare situazioni di deadlock tra processi, utilizzando una politica di riassegnazione dinamica della priorita' dei processi

Caratteristiche di un Sistema Operativo Real-Time

- Realizzare un metodo di sincronizzazione e comunicazione tra processi che sia deterministico (la comunicazione deve avvenire in tempi certi)
- Devono essere noti gli intervalli di tempo necessari al S.O. per sospendere un processo, avviarne un altro, realizzare una chiamata di sistema (il *context switch*)
- Gestire gli eventuali malfunzionamenti in maniera da garantire il funzionamento (al più degradato) del sistema
- Scalabilità: deve essere possibile scegliere quali funzionalita' del S.O. utilizzare in funzione dell'applicazione di controllo da realizzare e quindi scegliere soltanto le funzionalita' necessarie, senza appesantire inutilmente il SW di controllo

Il Sistema Operativo Real-Time WindRiver VxWorks

- Fornisce gli strumenti necessari a realizzare applicazioni real time, supportando:
  - Programmazione multitasking
  - Comunicazione Inter-task
  - Adeguata schedulazione dei task
- Composizione modulare e scalabile:
  - Kernel
  - Varie librerie di supporto
- Ottimizzazione dell'occupazione di memoria:
  - E' possibile scegliere quali moduli includere nel proprio progetto



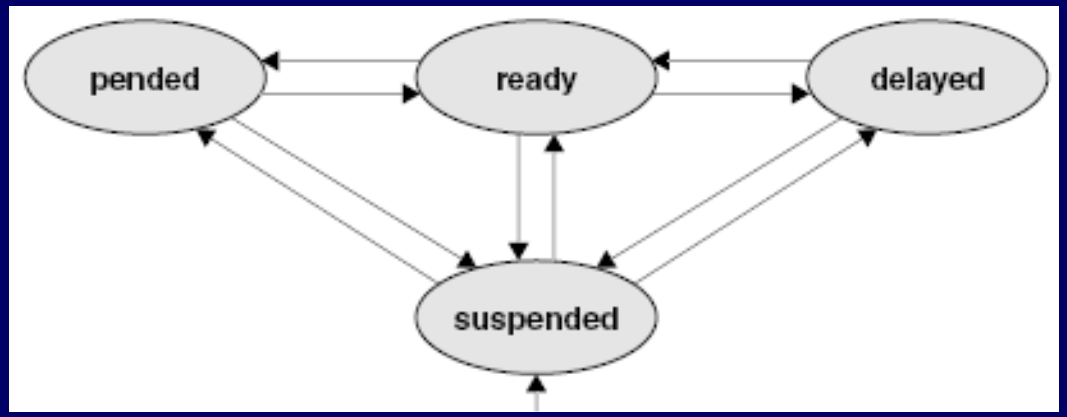
Il Kernel di VxWorks

- Il kernel di VxWorks offre al programmatore le seguenti caratteristiche:
  - multitasking con preemptive priority scheduling
  - sincronizzazione e comunicazione tra task
  - supporto per la gestione delle interruzioni e dei watchdog timers presenti sul target
  - gestione della memoria
  - supporto per la gestione del networking (TCP/IP e BSD socket)
  - download e upload dinamico di moduli oggetto e symbol tables utilizzando la rete

I task

- Il kernel di VxWorks mantiene traccia dello stato del sistema tramite un'apposita area di memoria in cui sono contenuti i Task Control Block (TCB), i quali contengono:
  - il Program Counter del task
  - i registri della CPU ed eventualmente i registri dell'unità floating point
  - uno stack per le chiamate a funzione e per le variabili dinamiche
  - i canali di I/O per lo standard input, lo standard output e lo standard error
  - un delay timer
  - un timeslice timer
  - strutture di controllo per il kernel
  - gli handlers per i segnali
  - informazioni sul debugging e sulle prestazioni

Lo stato dei task

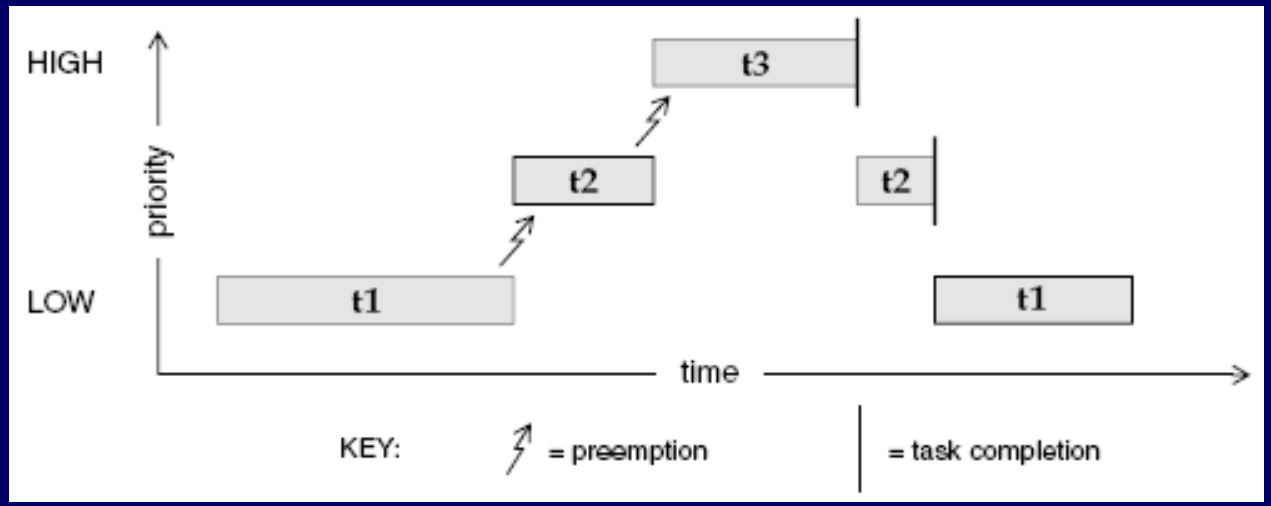


- In ogni istante ciascun task è caratterizzato da uno *stato*:
  - READY: il task è pronto per essere eseguito e la sola risorsa di cui ha bisogno è la CPU
  - PEND: il task è bloccato in attesa di una qualche risorsa
  - DELAY: il task è stato *addormentato* per qualche tempo
  - SUSPEND: il task non può essere eseguito

Lo scheduler

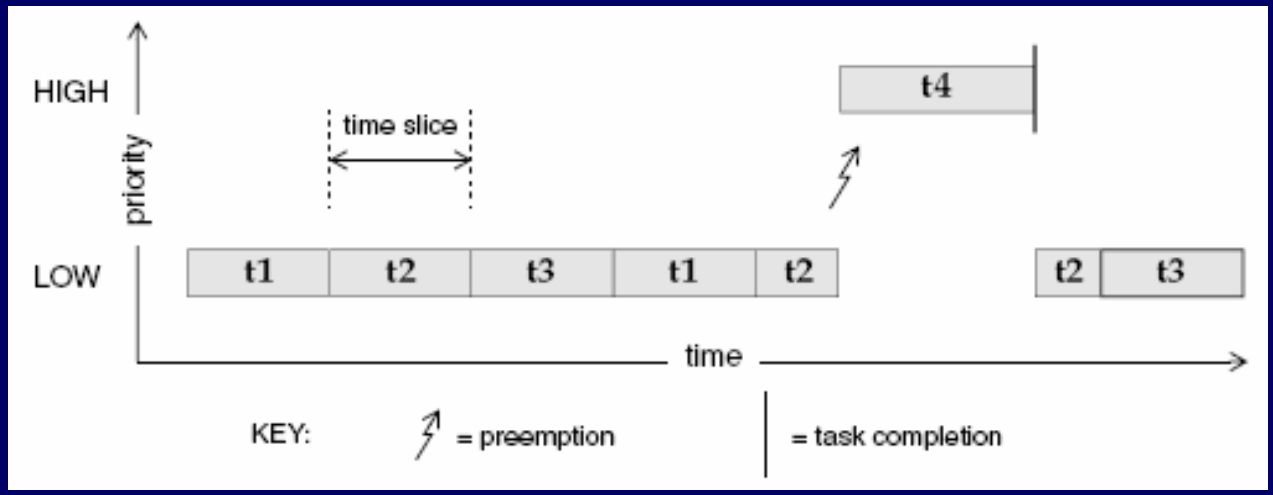
- Sceglie a quale dei task pronti allocare la CPU
- La scelta avviene in funzione della *priorità* posseduta da ciascun task
- Nel caso di VxWorks:
  - Ogni task può avere una priorità variabile fra 0 (valore massimo) e 255 (valore minimo)
  - Lo scheduling può essere di tipo *Preemptive Priority Scheduling*
  - Lo scheduling può essere di tipo *Round-Robin Scheduling*

Preemptive priority scheduling



- La CPU viene concessa al task a più alta priorità pronto per l'esecuzione
- Se due task hanno la stessa priorità uno dei due può mantenere indefinitamente la CPU

Round-robin scheduling



- Lo scheduler cerca di distribuire equamente i quanti di tempo (time slice) di CPU tra i task allo stesso livello di priorità
- Se nel frattempo arriva un task a priorità maggiore esso viene servito

Creazione di un task

- *Id*=**taskspawn**(*name, priority, options, stacksize, main, arg1. …arg10):* crea e attiva un nuovo task mettendolo in coda pronti:
  - Id: valore intero che identifica univocamente il task creato
  - Nome: stringa preferibilmente univoca per individuare il task (in debuging)
  - Priorità:
    - 100 - 255 per i task utente
    - 51 - 99 per le routine di servizio degli Interrupt
    - 0 – 50 usati dai task di servizio del S.O.
  - Nome della funzione main: è il nome della funzione del programma da usare come main del nuovo task
- Creazione e attivazione possono essere fatte in due istanti distinti:
  - *id* **= taskInit(***…***)**: crea il task (e il suo contesto) lo pone in stato pronto+sospeso
  - **taskActivate(***id***)**: mette il task in stato pronto

Eliminazione di task

- Un task può essere eliminato su sua richiesta spontanea:
  - Esplicitamente: invocando la funzione **Exit()**
  - Implicitamente: se termina la funzione indicata come main
- Un task può essere eliminato su richiesta di un altro task tramite **taskDelete(*id*)**
- Un task può prevenire la propria cancellazione da parte di altri tasks mediante **taskSafe()** e ripristinarla mediante **taskUnsafe()**:
  - Si usano prima e dopo una sezione critica
  - Il task **a** che tenta di cancellare un task **b** *sicuro* viene bloccato
  - Dopo la Unsafe, il task **a** viene rimesso in coda pronti e quando schedulato può cancellare il task **b**

Controllo di task

- Primitive per controllare il comportamento dello scheduler e lo stato dei task:
  - **kernelTimeSlice(*v*):** abilita l'uso dello scheduling round robin con quanto temporale pari a v; lo disabililta se v=0
  - **taskPrioritySet(*id,p*):** cambia la priorità del task id portandola a p
  - **taskLock():** previene la preemption: il task chiamante mantiene il possesso della CPU fino a quando non si blocca in attesa di qualche risorsa
  - **taskUnlock():** ripristina la possibilità di fare preemption sul task chiamante (che in precedenza aveva chiamato la taskLock());
  - **taskSuspend(*id*):** porta il task id in stato sospeso
  - **taskDelay(*id,t*):** porta il task id in stato delayed per il tempo t
  - **taskResume(*id*):** fa uscire il task id dallo stato sospeso

Comunicazione tra task

- Per poter essere veramente utile un S.O. multitasking deve disporre di metodi di comunicazione e sincronizzazione tra task che gli permettano di:
  - Lavorare in modo cooperativo
  - Coordinarsi nell'accesso in mutua esclusione alle risorse condivise
- VxWorks fornisce i seguenti meccanismi di comunicazione per task eseguiti su uno stesso target:
  - memoria condivisa
  - semafori
  - code di messaggi e pipe
  - socket e remote procedure call
  - segnali

Memoria condivisa

- Lo spazio di indirizzamento di VxWorks è lineare ed unico per tutti i task, quindi gli accessi ad oggetti condivisi possono essere effettuati indirizzando direttamente tali oggetti
- L'unico problema è quello di evitare accessi contemporanei allo stesso oggetto da parte di più task che potrebbero portare alla corruzione dell'oggetto stesso; si rendono quindi necessari dei meccanismi di mutua esclusione

Disabilitazione delle interruzioni

- È il metodo più sicuro per ottenere la mutua esclusione in quanto disabilitando le interruzioni si previene anche il context switch, quindi il task corrente resta in esecuzione finchè non le riabilita
- Il problema di questo metodo è che non potendo più rispondere alle interruzioni un sistema real-time è impossibilitato a reagire a stimoli esterni
- La disabilitazione avviene con una porzione di codice del tipo:

```
funcA()
{
int lock = intLock();
        /* regione critica che non può essere interrotta */
intUnlock (lock);
}
```


Disabilitazione della preemption

- La disabilitazione della preemption rimedia al problema del blocco degli interrupt, che possono ora essere tranquillamente serviti
- Blocca però anche tutti i task che non avevano bisogno della risorsa che si voleva proteggere con la mutua esclusione, anche quelli a priorità superiore a quella del task in questione, in quanto non possono più essere attivati dallo scheduler
- La regione critica viene protetta nel modo seguente:

```
funcB()
{
taskLock();
        /* regione critica che non può essere interrotta */
taskUnlock();
}
```


Semafori

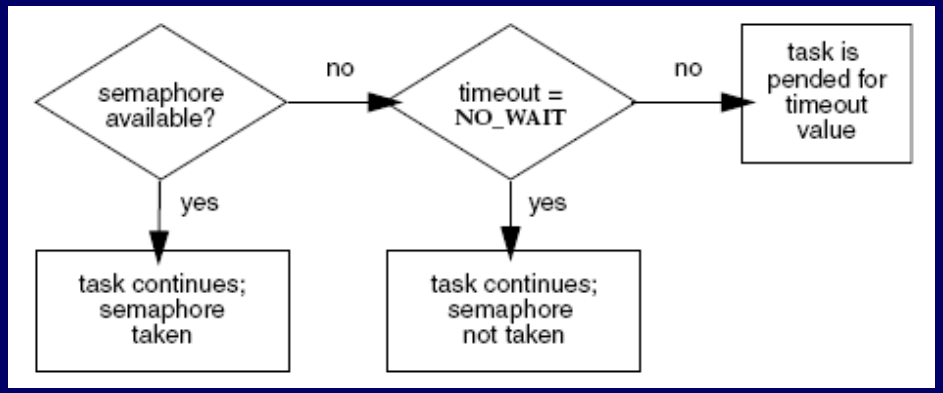
- I semafori sono il più veloce meccanismo di gestione della mutua esclusione e della sincronizzazione tra task fornito da VxWorks
- VxWorks fornisce tre diversi tipi di semafori, ognuno ottimizzato per la soluzione di una differente classe di problemi, che verranno descritti in seguito:
  - semafori binari
  - semafori di mutua esclusione
  - semafori con contatore

Semafori binari

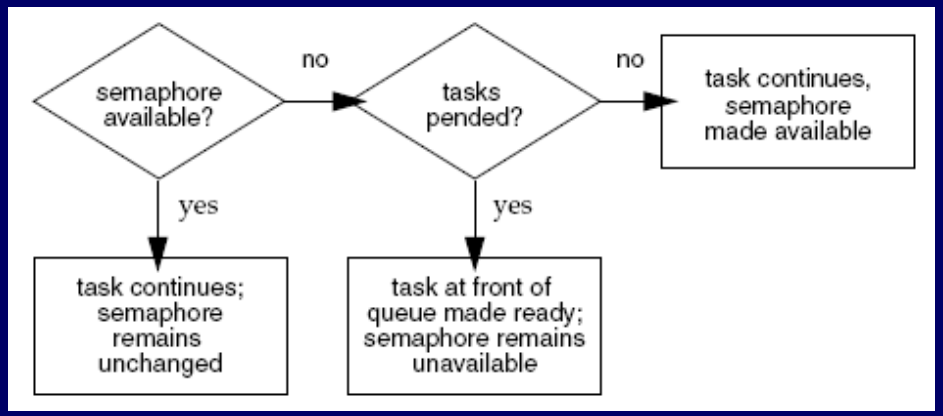
- Primitive di sistema per l'uso dei semafori binari:
  - Id **= semBCreate(***queue_options, init_value***)**: crea un semaforo binario riportandone l'id:
    - *Queue_options:* specifica se i task vengono accodati e risvegliati in base alla loro priorità o con politica FIFO
    - *init_value:* valore iniziale del semaforo (occupato o libero)
  - **semTake(***semId,timeOut***)**: testa un semaforo e se è libero lo occupa e ritorna OK; se è occupato blocca il task fino a quando il semaforo non si libera o allo scadere dell'eventuale timeOut ritornando ERROR (valori speciali per timeOut: NO_WAIT, WAIT_FOREVER)
  - **semGive(***semId***)**: libera un semaforo
  - **semDelete(***semId***)**: elimina un semaforo; tutti i task bloccati vanno in coda pronti, la semTake che li aveva bloccati ritorna ERROR

Semafori binari

- Funzionamento della semTake():



- Funzionamento della semGive():



Semafori di mutua esclusione

- semId = **semMCreate(***queue_options* | *SEM_INVERSION_SAFE* | *SEM_DELETE_SAFE***)**:
  - crea un semaforo di mutua esclusione; questo tipo di semaforo permette di proteggere porzioni di codice delimitate dalle primitive *semTake()* e *semGive()*, fornendo quindi un meccanismo di gestione della mutua esclusione più raffinato rispetto a quello ottenibile utilizzando la disabilitazione delle interruzioni o della preemption

- Il comportamento risulta essere identico a quello dei semafori binari con le seguenti eccezioni:
  - può essere utilizzato solo per la mutua esclusione
  - può essere rilasciato solo dal task che ha effettuato con successo la *semTake()*
  - non può essere rilasciato dall'interno di una routine di servizio di interruzione
  - l'operazione *semFlush()* (sblocca automaticamente tutti i task accodati sul semaforo) non è consentita

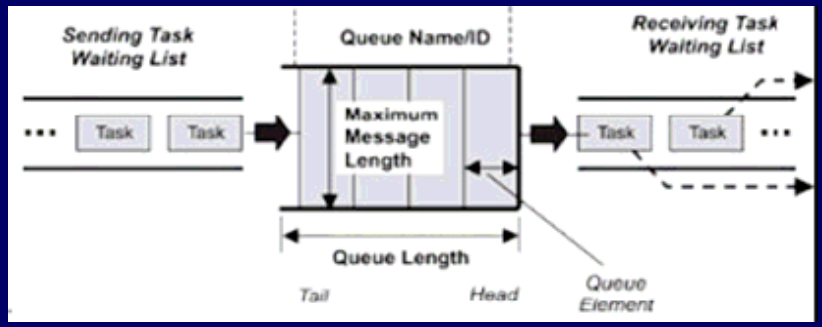
I semafori con contatore

- Sono simili ai semafori binari ma possono essere presi più volte contemporaneamente
- Nel momento in cui viene creato viene specificato un valore iniziale per il contatore
- Ogni volta che un task esegue una semTake():
  - se contatore == 0 blocca il task
  - altrimenti decrementa il valore del contatore senza bloccare il task
- Ogni volta che un task esegue una semGive():
  - se contatore == 0 e ci sono dei task bloccati, il primo della coda viene risvegliato
  - Altrimenti incrementa il valore del contatore
- Uso tipico: gestione di un pool di risorse identiche fra loro

Code

- Rappresentano un meccanismo di comunicazione inter-task di più alto livello rispetto alla coppia “memoria condivisa-semafori”:
  - Ogni coda è formata da un numero variabile di messaggi ciascuno di dimensione variabile
  - Scritture e letture possono essere fatte da qualsiasi task
  - L’invio di un messaggio è bloccante se la coda è piena
  - La ricezione di un messaggio è bloccante se la coda è vuota
  - In entrambi i casi, allo scadere del timeout il task è risvegliato ma la primitiva ritorna un valore di errore
  - L’invio di un messaggio risveglia il primo dei task eventualmente bloccati sulla coda in attesa di messaggi
  - La ricezione di un messaggio risveglia il primo dei task eventualmente bloccati sulla coda nel tentativo di inviare messaggi

Creazione ed eliminazione di code



- Creazione:

  msgQId = **msgQCreate(***maxMsgs, maxMsgLenght, Options***)**
  - Principali opzioni:
    - msg_Q_Fifo: accoda i task sulle Waiting List ordine di arrivo
    - msg_Q_Priority: accoda i task sulle Waiting List in base alla loro priorità
- Eliminazione:

  msgQDelete(*msgQid***)**
  - Tutti i task eventualmente accodati vengono sbloccati e ritornano un valore di errore

Uso di code

- Invio di un messaggio ad una coda:
  - **msgQSend (***msgQId, char * buffer, nBytes, int timeout, int priority***)**
    - *timeout:* tempo max di blocco se la coda è piena
    - *Priority:*messaggio è normale o urgente (verrà inserito in testa alla coda)
- Ricezione di un messaggio da una coda:
  - **msgQReceive(***msgQId, char * buffer, nBytes, int timeout***)**
    - *timeout:* tempo max di blocco se la coda è vuota
- Valori speciali per timeout: NO_WAIT, WAIT_FOREVER

Gli interrupts

- La gestione delle interruzioni è un meccanismo di fondamentale importanza in un sistema operativo real-time perché fornisce un modo estremamente veloce per rispondere ad eventi esterni
- L'arrivo di un'interruzione causa la sospensione del task in fase di running e l'attivazione di una routine di servizio (ISR) contenente il codice da eseguire per rispondere opportunamente all'evento che ha causato l'interrupt
- È importante notare che la sospensione del task e la contestuale attivazione dell'ISR avvengono senza la necessità di effettuare il context switch visto che la routine di servizio esegue in uno speciale contesto proprio per garantire la massima velocità possibile di risposta
- Al termine dell'ISR l'esecuzione del task interrotto viene ripristinata

Funzionamento degli interrupt

- La connessione tra l'interruzione e la routine di servizio avviene tramite la primitiva:
  - STATUS **intConnect(***VOIDFUNCPTR *vector, VOIDFUNCPTR routine, int parameter***)**
    - connette la routine specificata al vettore di interruzione
    - quando arriva l'interrupt la routine viene attivata e le viene passato come argomento il parametro specificato nella *intConnect()*
- La routine di servizio è una normale funzione C che soffre solo di alcune limitazioni: non può effettuare system call che possano in qualche modo bloccare il sistema ed eseguire operazioni di I/O

Funzionamento degli interrupt

- Prima di effettuare la connessione, la **intConnect()** aggiunge una piccola quantità di codice alla routine da connettere, facendola precedere dalle istruzioni necessarie al salvataggio dei registri del target e all'inserimento dell'argomento sullo stack
- Al termine della funzione viene inserito il codice necessario al ripristino di tali registri e dello stack, seguito dalle istruzioni per uscire dalla modalità di interruzione
- La **intConnect()** restituisce OK se è riuscita ad effettuare correttamente la connessione, ERROR in caso contrario

