# Lezione R12

## Sistemi operativi real-time – I

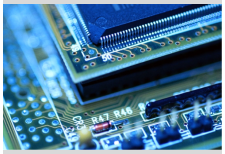
Sistemi embedded e real-time

8 gennaio 2013

Marco Cesati

Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma Tor Vergata







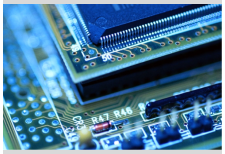


## Di cosa parliamo in questa lezione?

In questa lezione richiamiamo alcune nozioni di base sui sistemi operativi con particolare riferimento a quelli real-time

1. Sistemi operativi real-time
2. Interruzioni hardware
3. Gestione del tempo
4. Schedulazione





## Sistemi operativi general-purpose

*Quali sono gli obiettivi di un sistema operativo?*

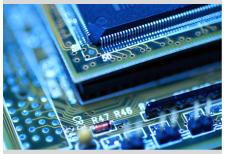
Interagire con l'hardware:

- Fa funzionare i dispositivi (programmi *driver*)
- Controlla l'uso delle risorse H/W (consumo di energia, salva-schermo, . . . )

Fornire un ambiente di esecuzione alle applicazioni:

- Costruisce una astrazione dell'architettura fisica
- Offre interfaccie di accesso ai dispositivi H/W
- Distribuisce le risorse H/W alle applicazioni
- Implementa i servizi di comunicazione tra applicazioni locali e remote
- Permette l'esecuzione contemporanea di più applicazioni (*multitasking*)
- Consente l'accesso di più utenti (*multiuser*)









## Sistemi operativi real-time

*In cosa differisce un sistema operativo real-time da un sistema operativo general purpose?* ***Le applicazioni!***

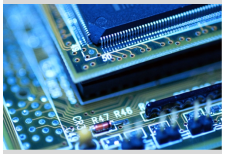
Le applicazioni real-time sono profondamente differenti rispetto a quelle di un sistema operativo general-purpose

Le caratteristiche distintive delle applicazioni real-time:

- Predicibilità
- Affidabilità

Talvolta le applicazioni real-time debbono avere *tempi di risposta rapidi*, ma questa proprietà non è caratterizzante





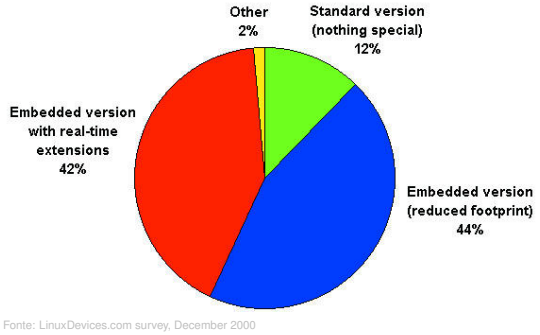
## Sistemi operativi embedded e real-time

Spesso le applicazioni real-time sono anche embedded

Devono quindi essere:

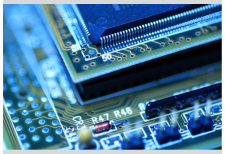
- Compatte
- Scalabili
- Con ridotto consumo di risorse

Quali necessità determinano la scelta di un sistema operativo?




Fonte: LinuxDevices.com survey, December 2000









## RTOS a microkernel

Molti SO per sistemi embedded e RT sono basati su un modello detto a *microkernel*

Il microkernel è un piccolo programma che realizza pochi servizi essenziali:

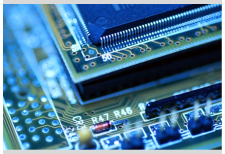
- driver dei circuiti H/W di base
- schedulazione dei processi
- comunicazione di base tra processi

Tutti gli altri servizi offerti alle applicazioni (driver delle periferiche, stack di rete, file system, . . . ) sono realizzati da altre applicazioni di sistema

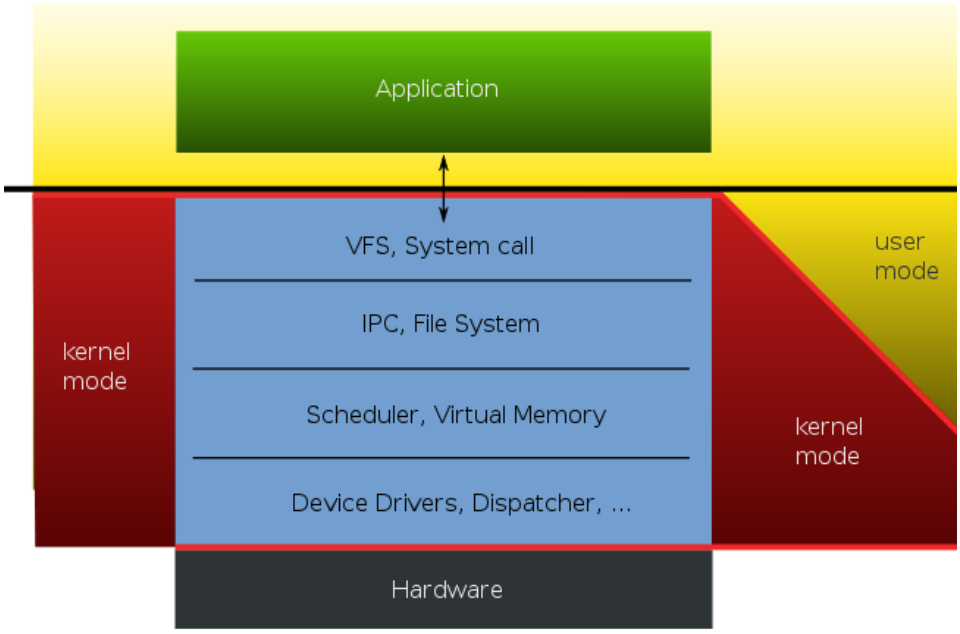
*Quali vantaggi offre l'approccio microkernel agli RTOS?*

Il microkernel è costituito da poche linee di codice, quindi è possibile verificarlo con accuratezza





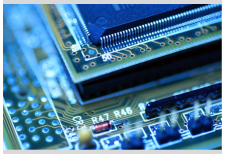
# Struttura di un SO monolitico




Fonte: Golftheman – public domain

Esempi: Linux, FreeBSD, SunOS Solaris, . . .

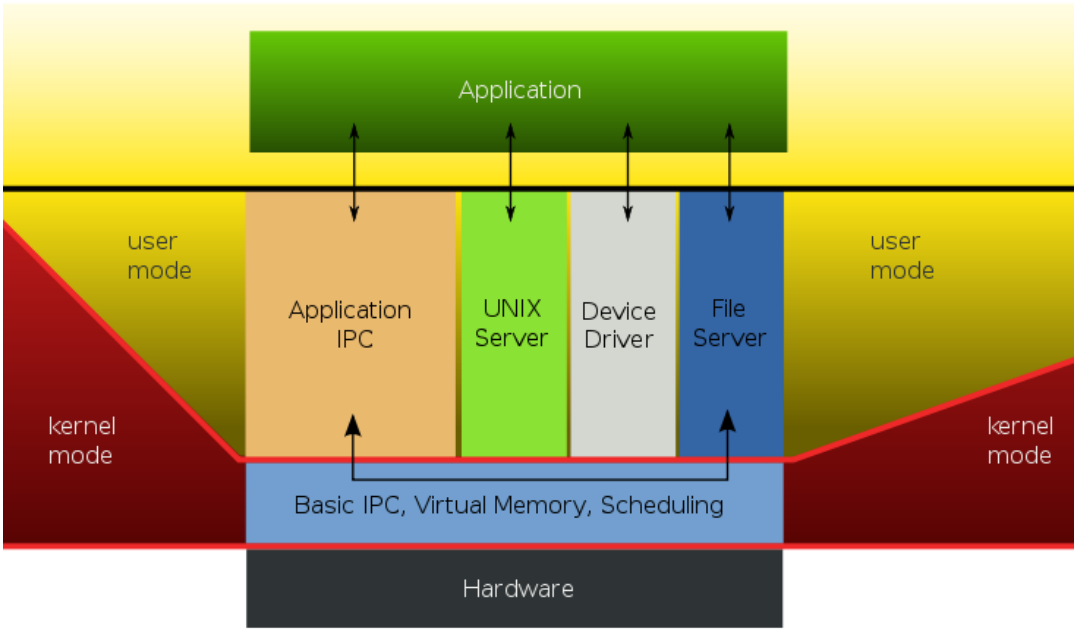








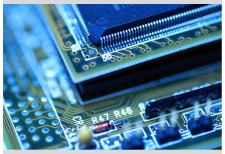
# Struttura di un SO a microkernel



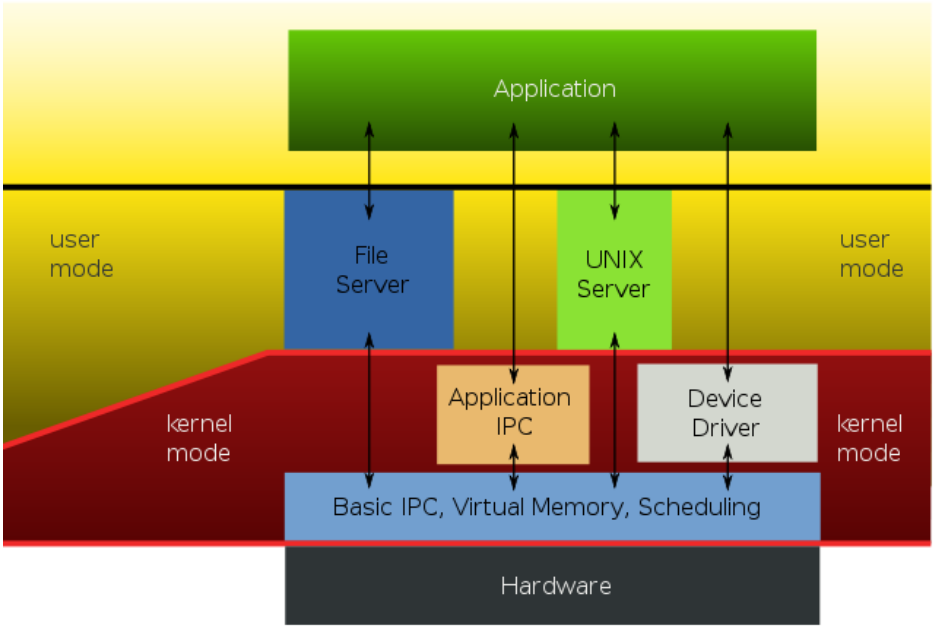

Fonte: Golftheman – public domain

Esempi: QNX, GNU Hurd (Mach & Ell Four), BeOS, . . .





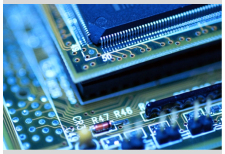
## Struttura di un SO ibrido




Fonte: Golftheman – public domain

Esempi: Windows NT, Mac OS X (XNU), . . .







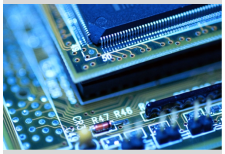


## Caratteristiche chiave di un RTOS

Un sistema operativo che opera in un contesto real-time ed embedded deve quindi offrire:

- Risposta ad eventi esterni predicibile: cura nella gestione delle interruzioni hardware
- Risposta ad eventi esterni efficiente: bassa latenza nei tempi di risposta
- Gestione affidabile e precisa di eventi temporali:
  - gestione dei timer e clock hardware con le relative interruzioni
  - gestione del tempo di sistema
  - gestione di allarmi e timer software per le applicazioni





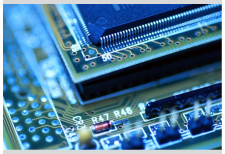
## Caratteristiche chiave di un RTOS (2)

- Schedulazione predicibile ed efficace dei task
  - implementazione di algoritmi deterministici (non euristici)
  - supporto al partizionamento dei task in sistemi multiprocessore
- Gestione della comunicazione e sincronizzazione dei task
  - memoria condivisa, code, mailbox, segnali
  - primitive di sincronizzazione
- Gestione della memoria
  - memoria virtuale
  - protezione dello spazio di indirizzamento

*In un RTOS tutte queste caratteristiche devono sempre essere presenti?* ***No!***

Ad es., in un sistema embedded le applicazioni RT potrebbero non utilizzare memoria virtuale e spazi di indirizzi separati









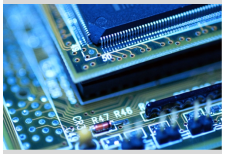
## Interruzioni hardware

Le *interruzioni hardware* sono segnali elettrici che consentono di notificare eventi esterni al SO od alle applicazioni in esecuzione

Esempi di eventi esterni:

- attività di I/O (trasferimenti disco, buffer della scheda grafica, tastiera e mouse, . . . )
- invio e ricezione di messaggi su rete, seriale . . .
- scambio di dati su bus di I/O (es. bus VME)
- disponibilità di nuovi dati su sensori
- richiesta di servizio da parte di dispositivi hardware





## Gestione delle interruzioni hardware

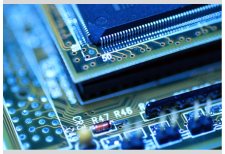
Ogni occorrenza di una interruzione hardware deve essere gestita tramite l'esecuzione di una opportuna procedura del SO o di una applicazione

Spesso la procedura deve essere attivata e conclusa rapidamente dopo la generazione dell'interruzione

Il tempo richiesto per la completa gestione di un interruzione (*latenza*) dipende comunque dalla periferica H/W:

- la lettura dei dati da un sensore è tipicamente un operazione breve
- il trasferimento di grandi blocchi di dati, ad esempio in una operazione su memoria di massa o su rete, è tipicamente una operazione lunga





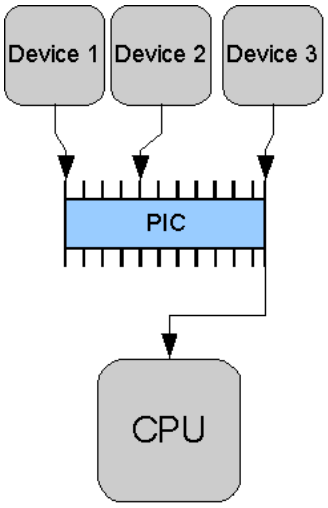




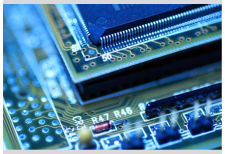
## Gestione delle interruzioni hardware (2)

In ogni caso, il SO deve dare risposte rapide alle interruzioni, altrimenti le periferiche H/W potrebbero funzionare male

- 1 linea fisica di interruzione da ogni dispositivo
- 1 sola linea verso il processore
- l'interruzione arriva al PIC (Programmable Interrupt Controller)
- il PIC asserisce la linea verso la CPU e aspetta conferma di ricezione (*acknowledge*)
- la CPU deve confermare velocemente la ricezione per poter ricevere ulteriori interruzioni dagli altri dispositivi





## Gestore delle interruzioni e ISR

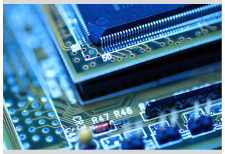
Problema: le interruzioni hardware devono essere confermate nel minor tempo possibile

Soluzione: i SO gestiscono le interruzioni hardware in più fasi distinte

Almeno due fasi implementate da due procedure distinte:

- *interrupt handler*
  - priorità elevata
  - dà conferma della ricezione dell'interruzione al PIC
  - salva e recupera il contesto di esecuzione del processore
- *interrupt service routine* (ISR):
  - priorità più bassa (ma generalmente superiore ai processi nel sistema)
  - esegue le operazioni specifiche per l'interruzione ed il dispositivo che l'ha generata









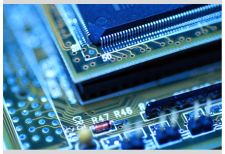
## La gestione del tempo

Una buona architettura per la gestione del tempo è fondamentale per supportare efficacemente applicazioni real-time

Il sistema operativo offre supporto relativamente a:

- dispositivi di tempo (*clock device*)
- funzioni di timer
- modalità di temporizzazione
- gestione precisa e predicibile degli eventi timer

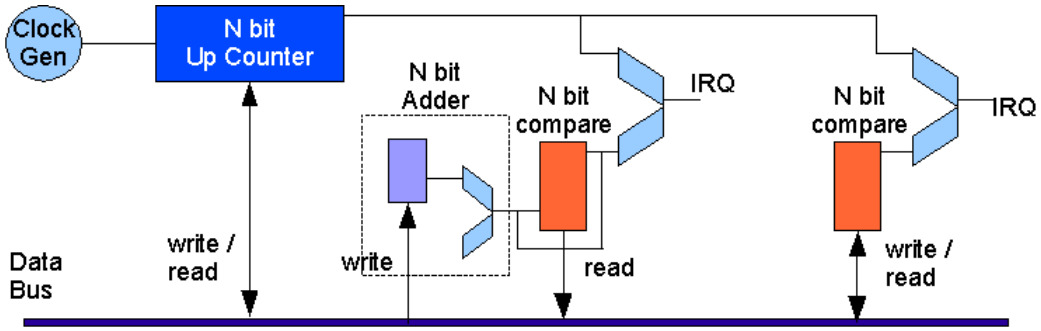




# Clock Device

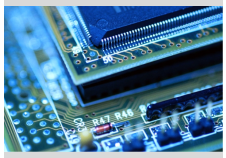
Un dispositivo di tempo consiste fondamentalmente in:

- un registro contatore incrementato ad una frequenza prefissata
- diversi comparatori hardware programmabili che generano una interruzione quando il valore del contatore coincide con quello contenuto in registro prefissato




Schema del clock device HPET









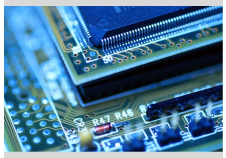
# Clock Device (2)

- Il kernel imposta il clock device per generare interruzioni periodiche (*clock tick*)
- Ad ogni *clock tick*:
  - viene aggiornato il tempo di sistema
  - si controlla la scadenza dei timer software
  - se necessario si invoca lo scheduler
- L'ampiezza dell'intervallo del tick è critica ai fini delle prestazioni
  - la risoluzione tipica di 100 Hz è talvolta insufficiente per alcuni tipi di applicazioni real-time
  - un clock tick troppo frequente introduce overhead non desiderabili

*Perché un intervallo di tick troppo lungo è un problema?*

L'invocazione dello scheduler, la granularità dei timer software e la risoluzione del tempo di sistema sono tipicamente legati alle occorrenze delle interruzioni periodiche





## Clock Device (3)

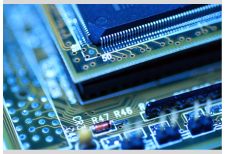
Per ottenere una buona risoluzione di clock con un overhead contenuto sono generalmente utilizzati tre approcci:

1. Aumentare la frequenza del clock interrupt, ma effettuare le operazioni che richiedono un overhead elevato solo a multipli interi di questa frequenza
2. Attivare lo scheduler non solo ad ogni tick, ma anche quando si creano o distruggono processi, si conclude una ISR, ecc.
3. Consentire alle applicazioni utente di accedere direttamente a dispositivi H/W di temporizzazione ad alta risoluzione

*Ciascuno di questi approcci ha qualche problema. Quali?*

1. La gestione delle interruzioni temporali non ha durata fissa
2. Le attivazioni dello scheduler non sono predicibili, e la risoluzione dei timer non migliora
3. Il codice delle applicazioni è fragile e non portabile







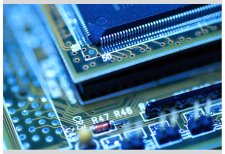


## Dispositivi di temporizzazione

Esempi di dispositivi di temporizzazione:

- *Real Time Clock (RTC)*: dispositivo a batteria, mantiene la data e l'ora anche quando il computer è spento (es. Motorola 146818 nei PC)
- *Time Stamp Counter (TSC)*: contatore monotono crescente associato con il segnale del bus (IA-32 o AMD64; l'equivalente PowerPC è il Time Base Register)
- *Programmable Interrupt Timer (PIT)*: dispositivo in grado di generare interrupt periodici o *one-shot* (es. PIT 8254)
- *CPU Local Timer (LOC)*: timer integrato nella CPU (es. APIC in IA-32 e AMD64, Decrementer in PowerPC)
- *High Precision Event Timer (HPET)*: dispositivo ad alta precisione, fornisce un contatore monotono crescente a 64 bit e numerosi timer
- *ACPI Power Management Timer (PMT)*: un timer disponibile su tutti i calcolatori basati su ACPI non influenzato dai meccanismi di risparmio dell'energia

Sistemi operativi real-time
Marco Cesati



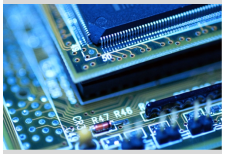
Schema della lezione
RTOS
Interruzioni H/W
Gestione del tempo
Schedulazione

# Dispositivi di temporizzazione (2)

| Disp. | Sorg. Tempo | Frequenza | IRQ | One Shot? | Risol. | Accur. |
|---|---|---|---|---|---|---|
| RTC | Interna | 2 - 8192 Hz | Sì | No | Bassa | Bassa |
| TSC | Bus | BusFreq / k | No | No | Alta | Bassa[1] |
| PIT | Interna | $\approx$ 1.2 kHz | Sì | Sì | Bassa | Buona |
| LOC | Bus | BusFreq / 16 | Sì | No | Alta | Bassa |
| HPET | Interna | $\geq$ 10 MHz | Sì | Sì | Alta | Alta |
| PMT | Interna | $\approx$ 3.58 MHz | No | No | Media | Alta |

[1] Può risentire del *CPU frequency scaling*









# Timer

Per le applicazioni, un *timer* è una "funzionalità software" che consente di eseguire una funzione o di notificare un evento in un certo istante temporale nel futuro

Una applicazione invoca una API del SO che crea nel kernel una struttura dati contenente:

- la scadenza del timer (*expire time*) assoluta o relativa
- la funzione da eseguire (*timer handler*) od il segnale da inviare alla scadenza

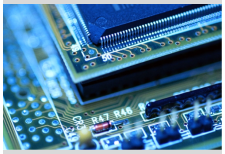
Il timer può essere

- *sincrono*: prevede la sospensione del task chiamante fino alla scadenza del timer
- *asincrono*: il task chiamante può continuare l'esecuzione

Esempi di API POSIX.1-2001:

timer sincrono: `clock_nanosleep()`
timer asincrono: `timer_settime()`

Sistemi operativi real-time
Marco Cesati



Schema della lezione
RTOS
Interruzioni H/W
Gestione del tempo
Schedulazione

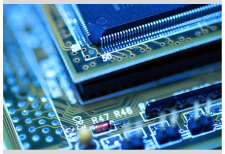
## La schedulazione

- La schedulazione di applicazioni real-time è uno dei punti fondamentali degli RTOS
- L'obiettivo dello scheduler è di determinare il successivo task da porre in esecuzione
- La scelta è effettuata utilizzando politiche di schedulazione semplici e dal comportamento facilmente predicibile

Gli RTOS oggi più adottati non supportano alcun tipo di analisi di schedulabilità o test di accettazione *on-line* per nuovi task

L'analisi di schedulabilità è generalmente realizzata durante la fase di progetto ed è normalmente affiancata da un test approfondito del comportamento del sistema









## Schedulazione a priorità fissa

- Tutti gli RTOS implementano un qualche tipo di politica di schedulazione preemptive con priorità fissa
- In molti RTOS lo scheduler è esclusivamente clock-driven

Va considerato nell'analisi di schedulabilità!

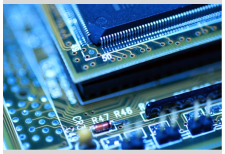
- Generalmente gli RTOS offrono un numero adeguato di livelli di priorità, ad esempio:

  Windows CE: 256 Linux: 100 VxWorks: 256

- Tuttavia l'analisi teorica relativa ai test di schedulabilità per algoritmi preemptive a priorità fissa prevede un numero infinito di livelli di priorità differenti

Va considerato anche il numero di livelli di priorità nell'analisi di schedulabilità!





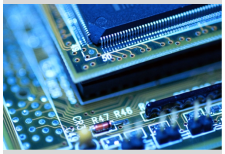
## Perdita di schedulabilità

In generale un numero finito di livelli di priorità causa una *perdita di schedulabilità*

- Siano $1, 2, \ldots, \Omega_n$ le priorità assegnate dall'algoritmo di schedulazione
- Sia $\Omega_s$ ($< \Omega_n$) il numero di livelli di priorità del sistema
- L'associazione tra le priorità assegnate e le priorità di sistema è rappresentato dai valori scelti per le priorità di sistema $\pi_1, \pi_2, \ldots, \pi_s$, ove:
  - $\pi_i \in \{1, 2, \ldots, \Omega_n\}$, per ogni $i$
  - $\pi_i < \pi_j$ se $i < j$
  - tutte le priorità assegnate numericamente minori o uguali a $\pi_1$ sono mappate su $\pi_1$
  - tutte le priorità assegnate tra $\pi_{k-1}+1$ e $\pi_k$ sono mappate su $\pi_k$, per $1 < k \leq \Omega_s$

Esempio: se $\Omega_n = 10$ e $\Omega_s = 3$, l'associazione lineare tra le priorità assegnate e quelle di sistema mappa 1, 2 e 3 su $\pi_1$; 4, 5 e 6 su $\pi_2$; 7, 8, 9 e 10 su $\pi_3$ $\Rightarrow$ $\pi_1 = 3$, $\pi_2 = 6$, $\pi_3 = 10$









## Associazione a rapporto costante

*È conveniente utilizzare una associazione lineare tra priorità assegnate e priorità di sistema?* ***In generale no!***

Lo scheduler non potrebbe distinguere i job di priorità più alta

Si utilizza invece una associazione che riserva più livelli di priorità di sistema ai livelli di priorità assegnata più alti, accorpando insieme le priorità assegnate di livello più basso

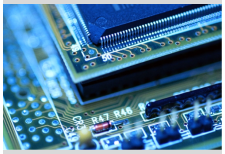
In pratica si può cercare di mantenere approssimativamente costanti i rapporti

$$g_k = \frac{\pi_{k-1} + 1}{\pi_k} \quad (1 < k \leq \Omega_s)$$

Nell'esempio precedente considerando $\pi_1 = 1$, $\pi_2 = 4$, $\pi_3 = 10$ si ottengono rapporti uguali a 1/2

Quindi 1 è mappato su $\pi_1$; 2, 3 e 4 su $\pi_2$; da 5 a 10 su $\pi_3$





## Perdita di schedulabilità (2)

**Teorema (Lechoczky & Sha, 1986)**

Per l'algoritmo di scheduling RM, con scadenze relative pari al periodo e numero $n$ di task elevato, se $g = \min_{1<k\leq\Omega_s} g_k$ allora:

$$U_{\mathrm{RM}}(g) = \begin{cases} \ln(2g) + 1 - g & \text{se } g > 1/2 \\ g & \text{se } g \leq 1/2 \end{cases}$$

La *schedulabilità relativa* indica la perdita di schedulabilità dovuta ad un numero limitato di livelli di priorità nel sistema:

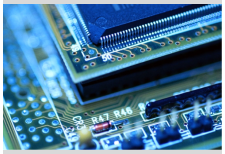
$$U_{\mathrm{RM}}(g)/\ln 2$$

Casi limite:

$g = 1$: $U_{RM}(g)/\ln 2 = 1$ $\Rightarrow$ nessuna perdita

$g = 1/2$: $U_{RM}(g)/\ln 2 = 1/(2\ln 2)$ $\Rightarrow$ perdita del 28%









## Schedulazione a priorità dinamica

- Tutti gli RTOS offrono delle API che consentono di impostare le priorità di un task direttamente sulla base della deadline relativa
- Generalmente lo scheduler implementa liste di task ordinate in base alla deadline relativa dei task
- Pochi RTOS (RTAI, RTLinux) supportano nativamente una politica di scheduling EDF
- Il reinserimento automatico dei task in una coda a priorità nella corretta posizione in base alla deadline assoluta è inefficiente
  - Si utilizzano strutture di dati dinamiche più sofisticate, ad esempi alberi di ricerca bilanciati