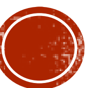
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

CORSO DI ADVANCED OPERATING SYSTEMS: MOBILE, CLOUD AND IOT

Real Time Operative Systems

RIFERIMENTI

- Giorgio Buttazzo, Sistemi in Tempo Reale, Pitagora, 2006
- Giorgio Buttazzo, Hard Real-Time Computing Systems, Springer, 2011
- Buttazzo, Lipari, Abeni, Caccamo, Soft Rea-Time Systems, Predictability vs. Efficiency, Springer, 2005

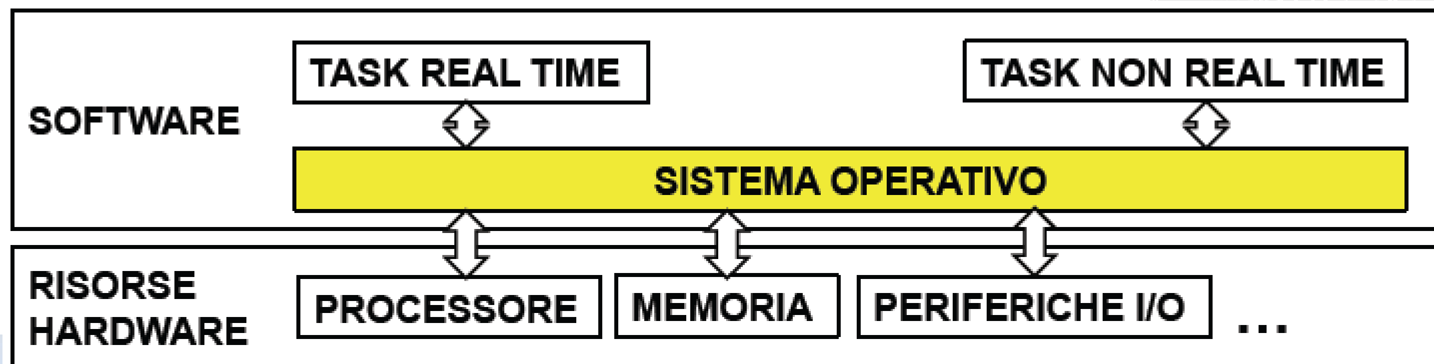


Hardware, Software, Sistema Operativo

Un qualsiasi sistema di controllo **real-time** è oggigiorno implementato via software.

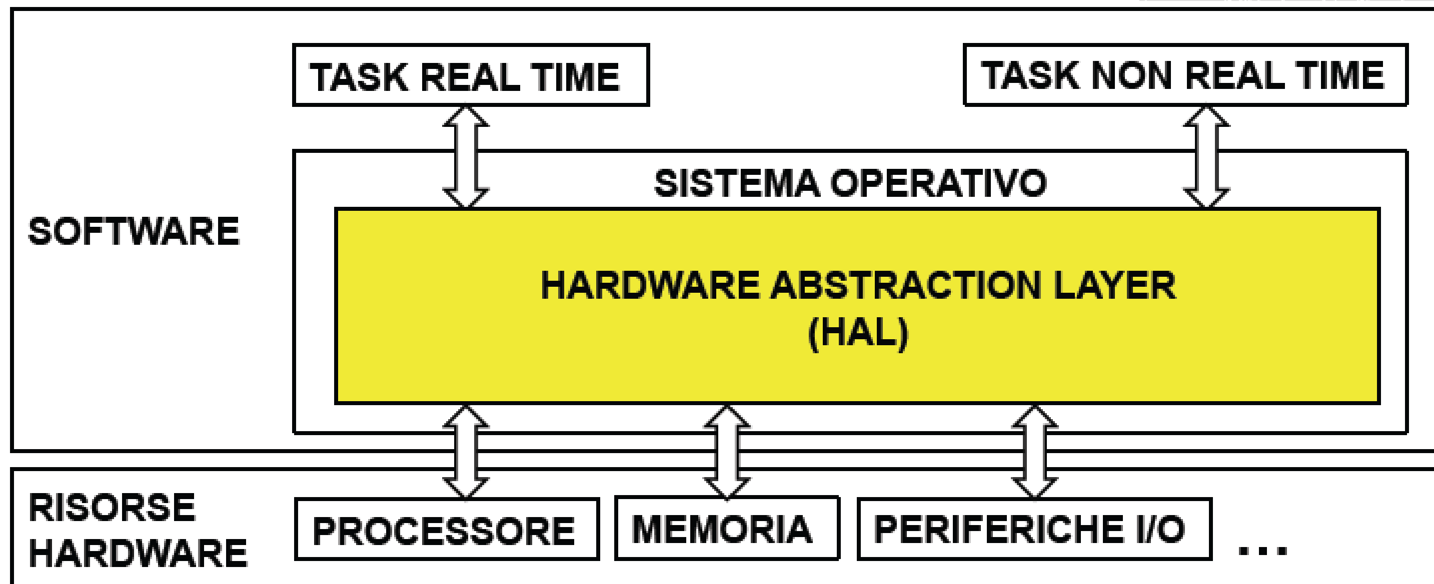
A fare da collante tra il livello **software** che implementa la logica del sistema di controllo e le risorse **hardware** controllate, vi è il **sistema operativo**.

Il sistema operativo è l'interfaccia software che permette di gestire i task real-time e non real-time che devono essere eseguiti dal processore.



Hardware Abstraction Layer (HAL)

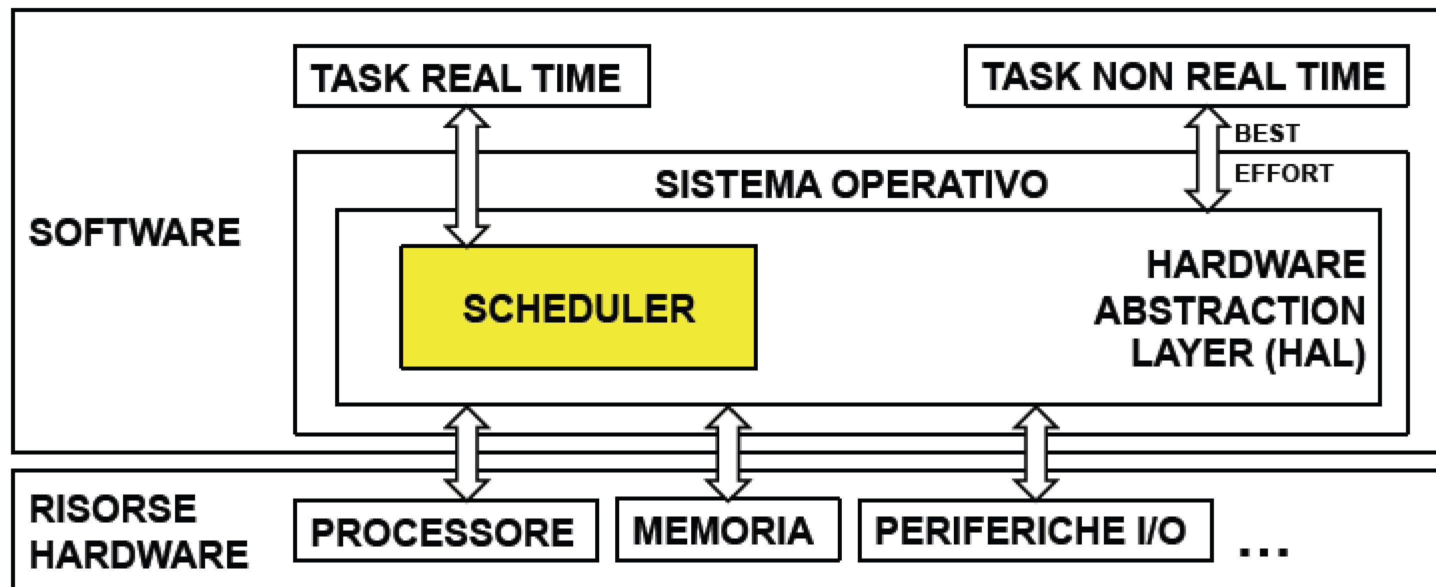
Per disaccoppiare il sistema operativo dalle infinite possibili combinazioni di risorse hardware viene introdotto un componente chiamato **HARDWARE ABSTRACTION LAYER** (HAL).



Real-Time e non Real-Time

I task **non real-time** vengono gestiti dal HAL con politica **BEST EFFORT**.

I task **real-time** vengono gestiti attraverso lo **SCHEDULER** che implementa algoritmi di scheduling specifici.



I sistemi operativi Event Driven

Un sistema operativo si dice **EVENT DRIVEN** se esso è in grado di schedulare un task (scartandolo, mettendolo in coda o mandandolo subito in esecuzione) nello stesso istante in cui si verifica l'evento che lo ha attivato.

I sistemi operativi Event Driven

VANTAGGI

• **INTUITIVO**: ogni task viene mandato allo scheduler non appena l'evento che lo attiva occorre

• **SEMPLICE DA USARE**: ad ogni task può essere associata una priorità e l'algoritmo di scheduling pensa a soddisfare anche vincoli hard real time.

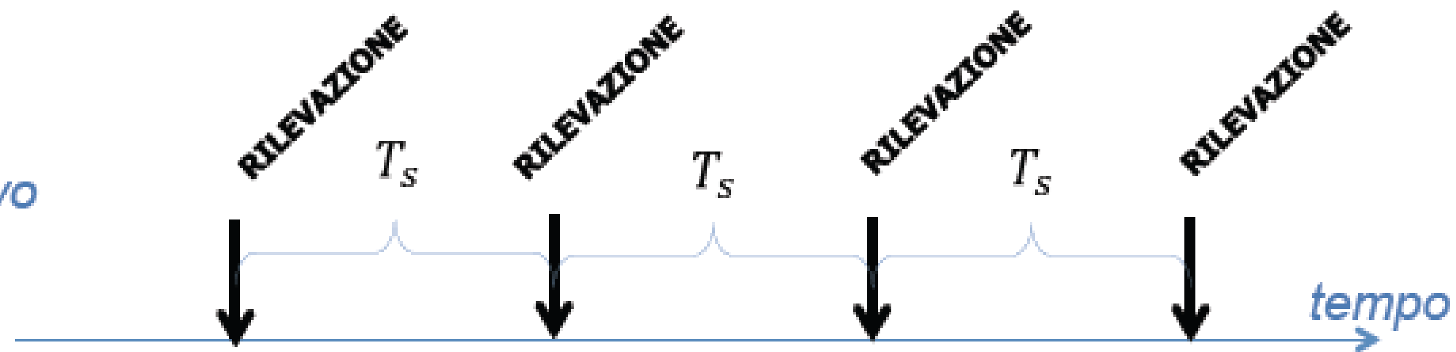
SVANTAGGI

• **COMPLESSO DA REALIZZARE**: definire un algoritmo di scheduling generale che risolva il problema della programmazione concorrente è difficile ed oneroso se non si conoscono a priori le caratteristiche dei task in ingresso.

I sistemi operativi Time Driven

Un approccio puramente event-driven è in realtà irrealizzabile se l'unità di elaborazione è di tipo digitale e quindi intrinsecamente quantizzata nel tempo.

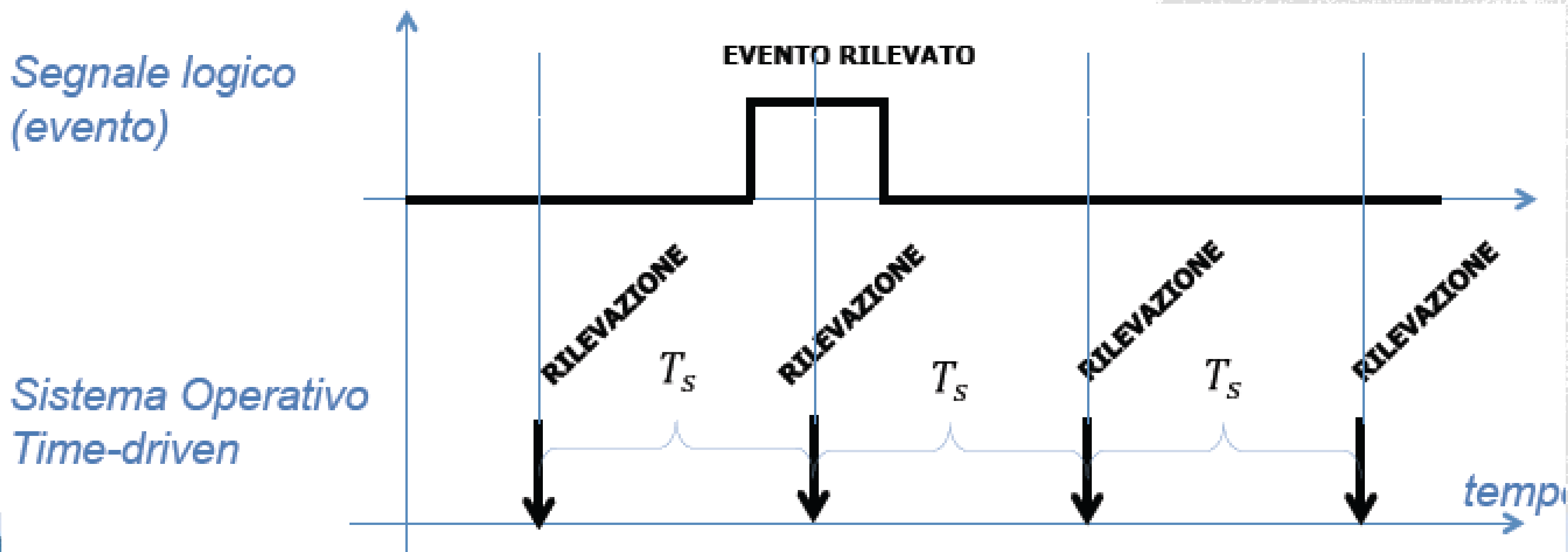
Un approccio realizzabile consiste nel rilevare periodicamente l'occorrenza di eventi e di gestire di conseguenza i relativi task. Tale approccio è detto **TIME DRIVEN**. Il periodo di tempo che intercorre tra due rilevazioni consecutive è detto **periodo di rilevazione** (o **periodo di scansione**) *Ts*.



I sistemi operativi Time Driven

Un sistema di controllo **TIME DRIVEN** deve gestire necessariamente le problematiche relative alla **gestione sincrona** di **eventi asincroni** (che attivano i task).

In un sistema di controllo **digitale**, un **evento** può essere associato al valore logico di una variabile binaria (**segnale logico**).

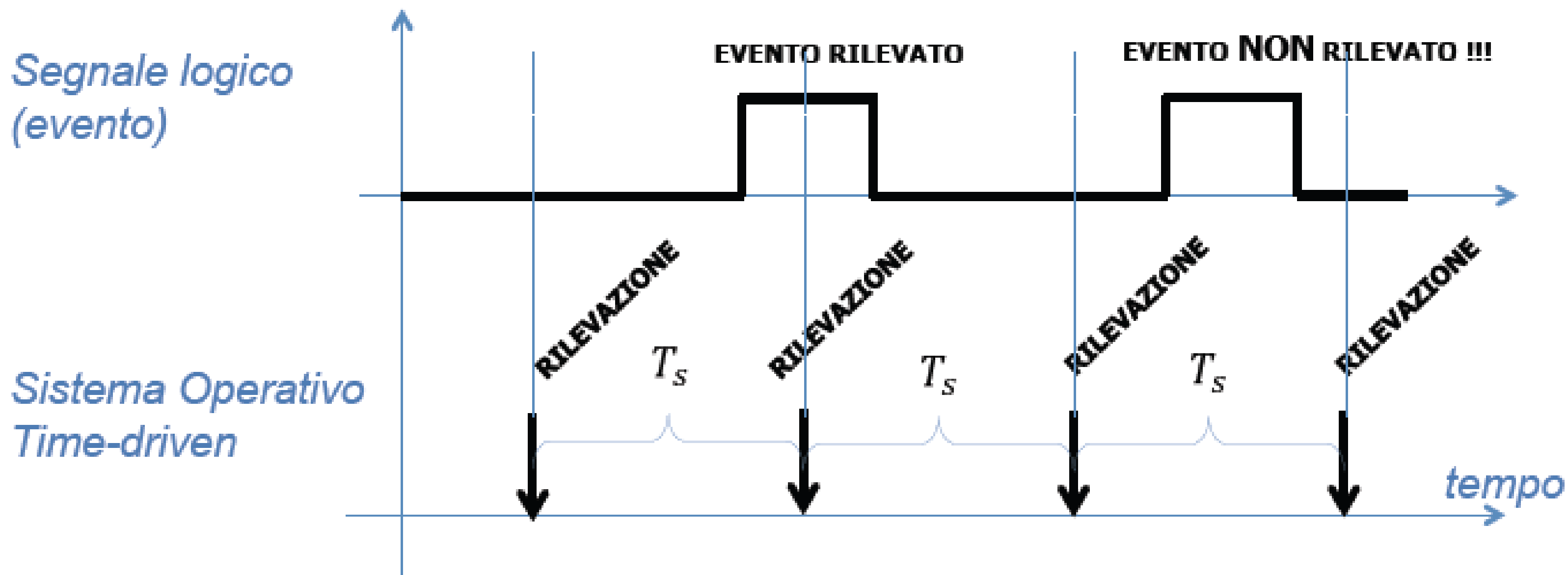


I sistemi operativi Time Driven

PROBLEMA 1 – OSSERVABILITÀ DEGLI EVENTI

In un sistema di controllo time driven un evento può NON ESSERE OSSERVABILE.

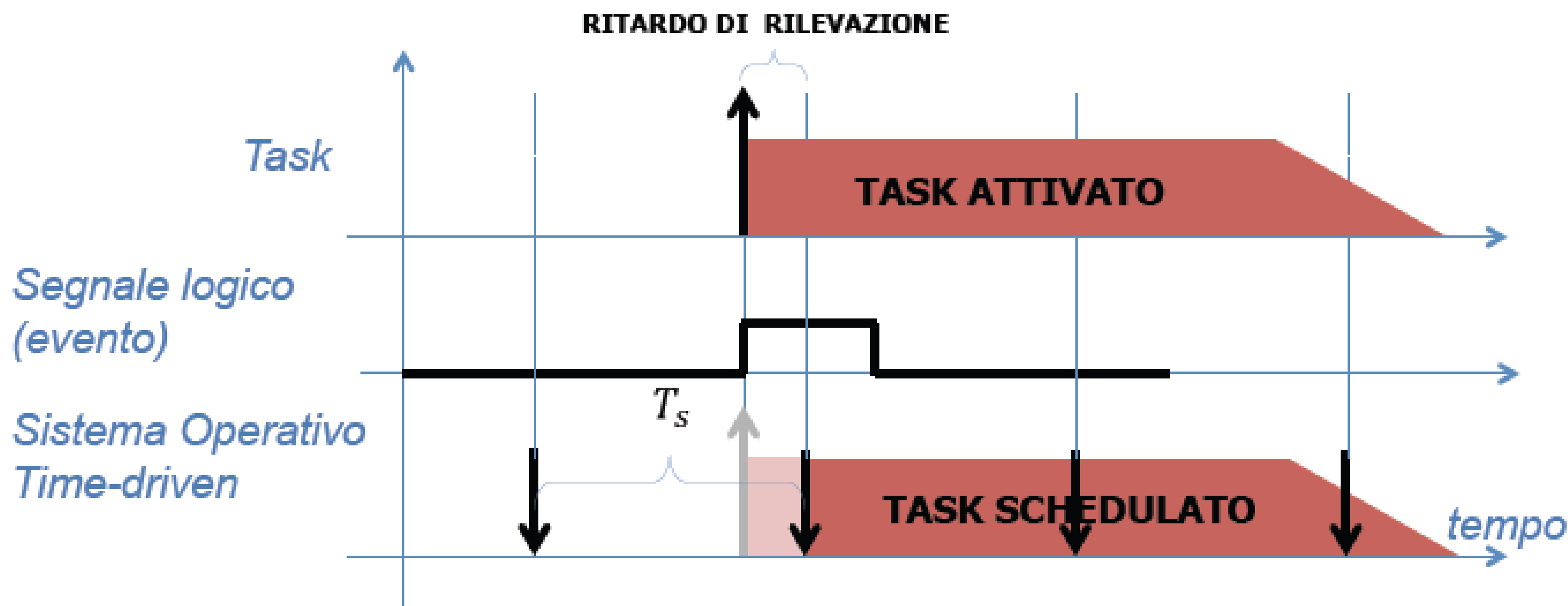
In particolare ciò può avvenire solo se il segnale logico associato all'evento rimane attivo per un tempo inferiore del periodo di rilevazione.



I sistemi operativi Time Driven

PROBLEMA 2 – RITARDO DI RILEVAZIONE

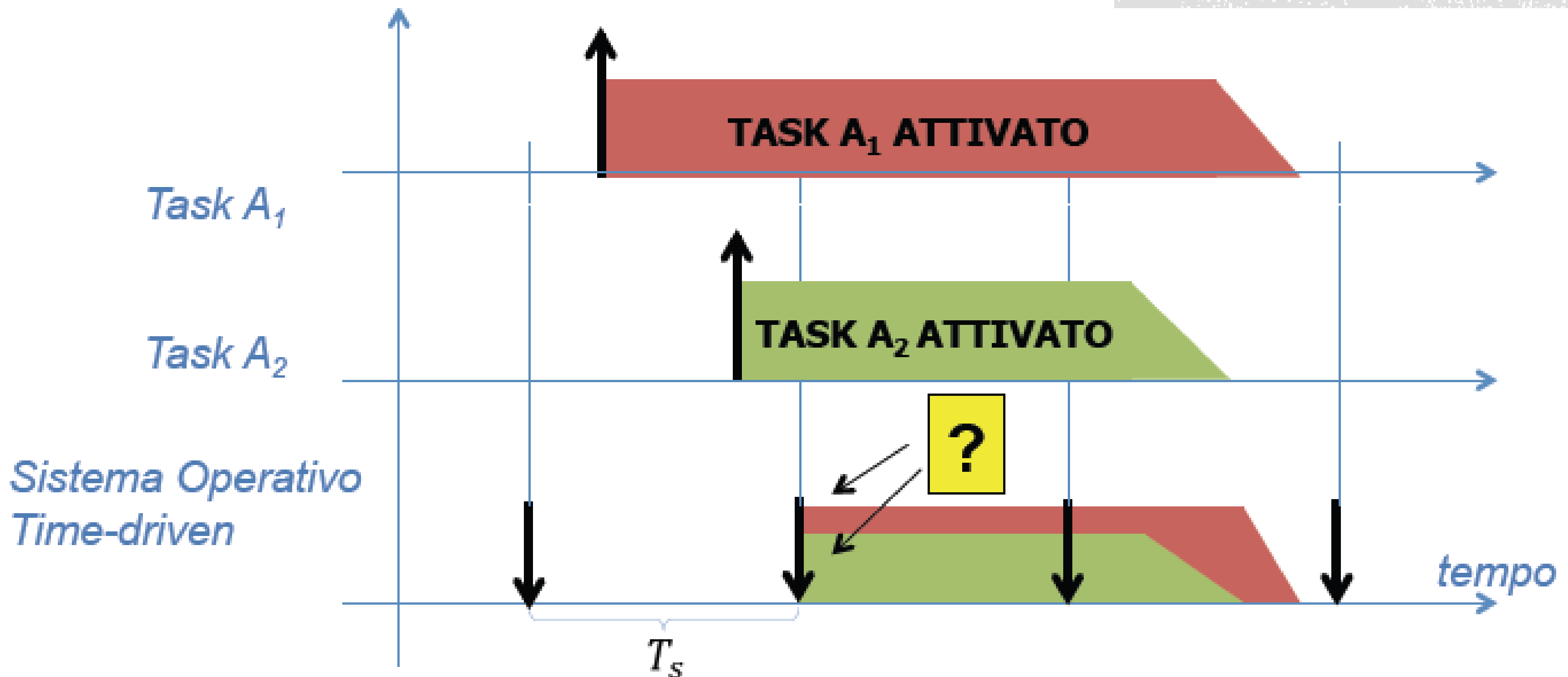
In un sistema di controllo time driven ogni occorrenza di un task è soggetta ad un ritardo di rilevazione che impatta necessariamente sullo start time del task.



I sistemi operativi Time Driven

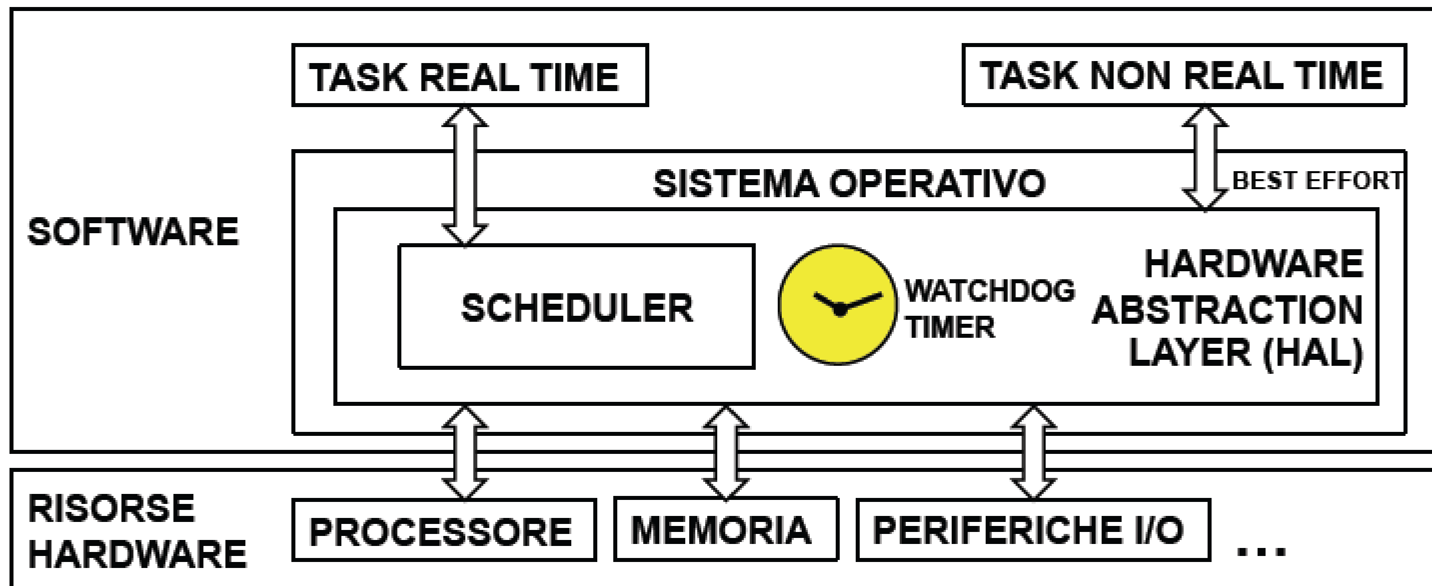
PROBLEMA 3 – ORDINE DI OCCORRENZA

In un sistema di controllo time driven, l'ordine in cui si presentano due o più eventi occorrenti tra due rilevazioni successive viene perso.



Il WatchDog

Il **sistema operativo** controlla periodicamente che le deadline dei task real-time vengano rispettate attraverso un **WATCHDOG TIMER**. Allo scadere del timer se una deadline è scaduta, l'anomalia è segnalata e viene eseguita una routine di emergenza.



Task Periodici e Aperiodici

I compiti (**task**) di un sistema real-time si distinguono in **periodici** e **aperiodici**

Definizione: I task che devono essere eseguiti ogni **p** unità di tempo sono detti periodici (di **periodo p**).

> ***Es***.: *task che eseguono un algoritmo di controllo andando a leggere periodicamente dei sensori e comandando degli attuatori.*

Tutti gli altri task sono detti aperiodici

Task che richiedono il processore in modo impredicibile sono detti **sporadici** (se c'è una separazione minima tra due invocazioni).

> ***Es***.: *task che rispondo a richieste di un utente umano.*

Le Deadline

Definizione: si indica con il termine **deadline** l'istante di tempo dopo il quale la computazione non è semplicemente in ritardo, ma è errata.

Le **deadline** sono vincoli di tempo stringenti imposti ai task dall'ambiente esterno.

> ***Es***.*: se un sistema di controllo di un processo industriale richiede che vengano controllati dei sensori ogni* ***p*** *unità di tempo, il task relativo dovrà essere in grado di portare a termine il controllo entro* ***p*** *unità di tempo, indipendentemente da quali siano gli altri task del sistema.*

Scheduling real-time

Nei **Sistemi Operativi Real-Time** (RTOS), i tempi di risposta di alcuni processi sono **fondamentali**.

I task real-time possono essere principalmente di due tipi:

- **Hard Real-Time**: devono essere eseguiti entro la loro scadenza altrimenti potrebbero causare malfunzionamenti.
- **Soft Real-Time**: hanno una scadenza associata, che è desiderabile, ma non obbligatoria.

Gli eventi legati all'esecuzione di tali task possono essere o **periodici** o **aperiodici**.

Un sistema che deve eseguire diversi task periodici è in grado di gestirli solo se i tempi da essi richiesti soddisfano determinati vincoli.

Scheduling real-time

Dati **m** task periodici con un periodo $\mathbf{P_i}$ e richiedenti $\mathbf{C_i}$ secondi di tempo CPU per essere gestiti, il rapporto $\mathbf{C_i / P_i}$ rappresenta la frazione di tempo CPU occupata da un certo task

I task possono essere soddisfatti solo se: $$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Lo scheduling real-time è ottenuto tramite algoritmi di scheduling a priorità.

Bisogna quindi trovare delle strategie di assegnazione delle priorità ai task che consentano uno scheduling ottimale

Rate Monothonic Scheduler

Il **Rate Monothonic Scheduler** (abbreviato RMS) è stato introdotto da Liu e Layland nel 1973.

Assegna le priorità ai task in base al loro periodo **T**, inteso come la quantità di tempo compresa fra l'arrivo di un istanza del task e la successiva → il task con il periodo più breve ha priorità

Il tempo di esecuzione **C** è la quantità di tempo di elaborazione richiesta da ogni occorrenza del task e in un sistema single-core il tempo di esecuzione non deve essere maggiore del periodo, ossia **C** < **T**.

Se tutti i processi vengono eseguiti completamente, si può calcolare l'utilizzo del processore come:

$$U = \frac{C}{T}$$

In un ambiente con priorità assegnate staticamente l'RMS si può considerare ottimale. Liu & Layland hanno dimostrato che per un insieme di **n** task periodici esiste una schedulazione che soddisfi tutte le scadenze entro un certo limite di tempo.

Rate Monothonic Scheduler

Il test di schedulabilità per l'RMS è una condizione sufficiente ma non necessaria con la quale viene garantito che tutti i processi vengano schedulati entro le loro scadenze, ed è data da:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1)$$

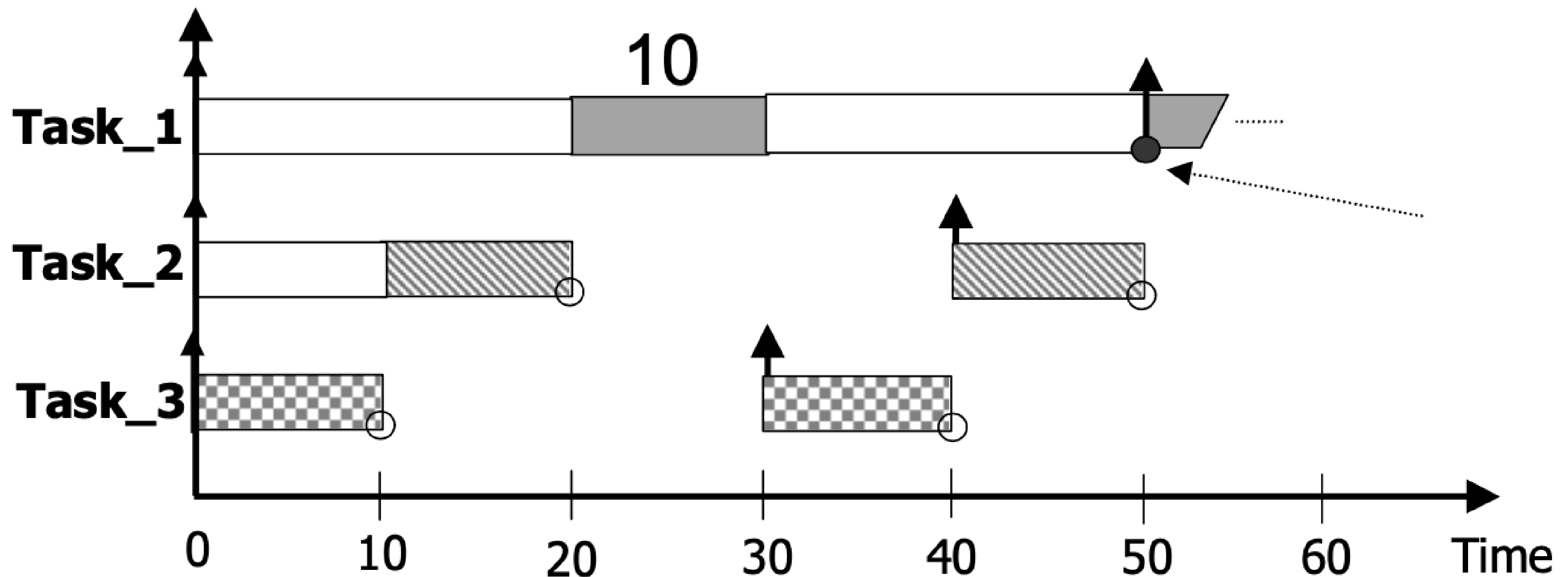
Se il numero di processi **n** tende ad infinito, questa espressione tende a:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147$$

RMS può garantire tutte le scadenze con un utilizzo del processore intorno al 69,3% e quindi il restante 30,7% della CPU può essere utilizzato per la schedulazione di processi non real-time.

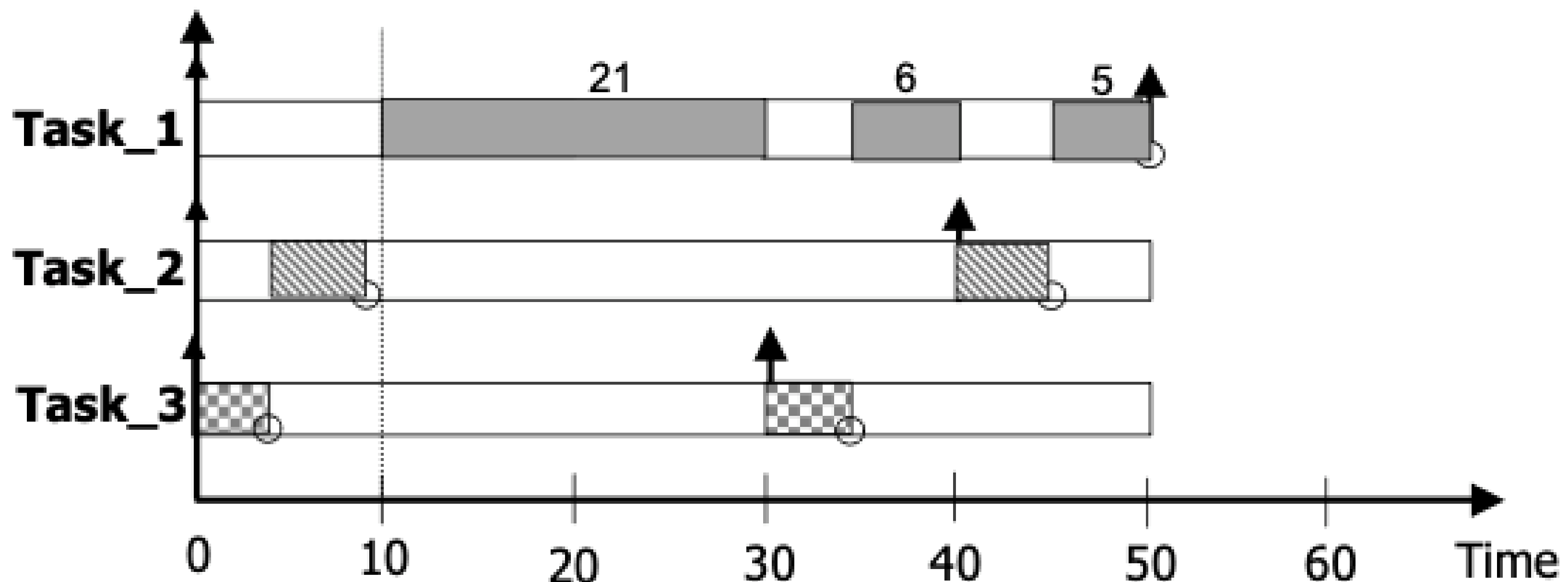
Process	Period, *T*	Computation time, *C*	Priority, *P*	Utilization, *U*
Task_1	50	12	1	0.24
Task_2	40	10	2	0.25
Task_3	30	10	3	0.33

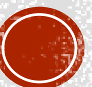
$$U = 12/50 + 10/40 + 10/30 = 0.24 + 0.25 + 0.33 = 0.82$$
$$U > U(3) = 3\,(2^{1/3} - 1) = 0.78$$



Process	Period, *T*	Computation time, *C*	Priority, *P*	Utilization, *U*
Task_1	50	32	1	0.400
Task_2	40	5	2	0.125
Task_3	30	4	3	0.250

$$U = 32/50 + 5/40 + 4/30 = 0.4 + 0.125 + 0.25 = 0.775$$
$$U < U(3) = 3\,(2^{1/3} - 1) = 0.78$$





EDF Scheduler

Questo tipo di scheduling utilizza una coda di priorità e tiene in considerazione le scadenze di ogni task.

Ogni volta che viene chiamata la funzione di schedulazione la coda viene analizzata alla ricerca del processo più vicino alla sua scadenza, il quale viene schedulato per l'esecuzione.

L'EDF è un algoritmo ottimo su sistemi single-core con pre-rilascio.

Nel caso ci siano processi periodici, anch'essi con delle scadenze (uguali al loro periodo) EDF ha una utilizzazione della CPU vicina al 100%. Il test di schedulabilità per EDF è infatti:

$$U = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

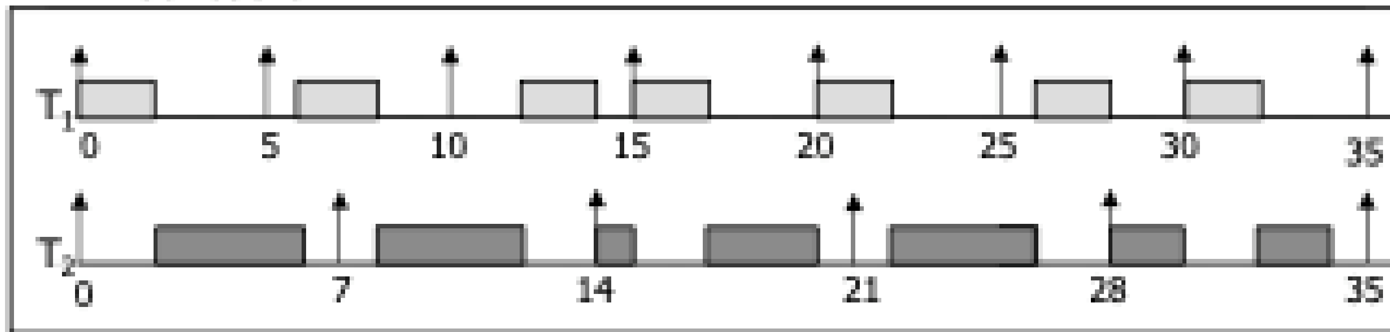
I limiti di questo algoritmo sono:

- se il sistema e sovraccarico è impossibile prevedere il numero di processi che non rispettano la scadenza.
- è un algoritmo molto difficile da implementare, per via di come dover rappresentare le scadenze.

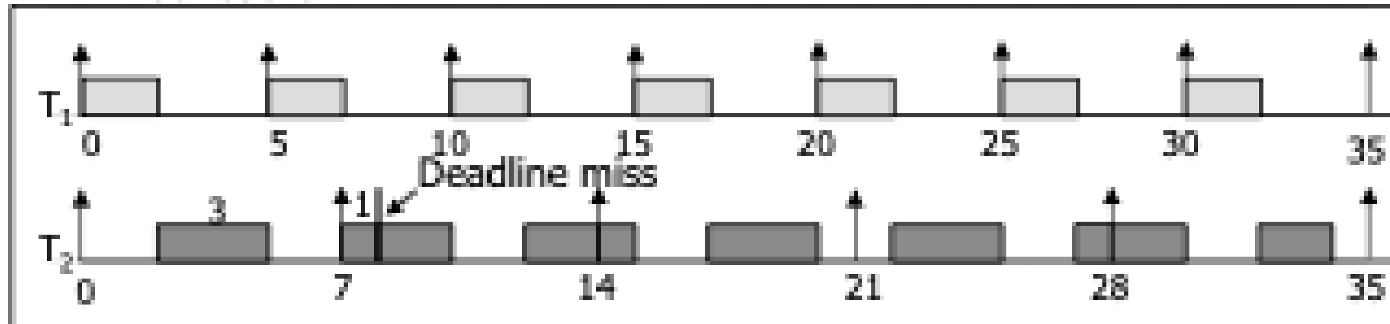
Example

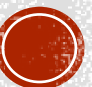
Process	Period, T	WCET, C
T_1	5	2
T_2	7	4

EDF schedule



RMA schedule





Sistemi Soft Real-time e Hard Real-time

Soft real—time:

I vincoli sui tempi di risposta non sono stringenti, perché in qualche caso si può non rispettarli

- **Es.**: i programmi di visualizzazione di un DVD devono estrarre i dati e processarli in modo da rispettare i vincoli di un frame ogni 50-esimo di secondo. Ma in casi di computazioni particolarmente complesse (ad es. compensazione di errori, o impegno della Cpu in un altro compito, se i vincoli non sono rispettati si produce solamente un peggioramento della qualità dell'immagine.

Hard real—time:

I vincoli sui tempi di risposta sono stringenti, perché se non vengono rispettati Il sistema di controllo è inutile, o addirittura pericoloso

- **Es.**: programmi di controllo delle superfici di volo di un aereo

Classificazione di task real-time

A seconda delle conseguenze che possono verificarsi a causa di una deadline mancata, un task real-time può essere distinto in tre categorie:

Hard: se mancare la scadenza può causare conseguenze catastrofiche sul sistema sotto controllo.

Firm: se produrre i risultati dopo la scadenza è inutile per il sistema, ma non causa alcun danno.

Soft: se produrre i risultati dopo la scadenza ha ancora qualche utilità per il sistema, sebbene causi un peggioramento delle prestazioni.

Gestione di task in contesti ibridi

In genere, le applicazioni del mondo reale includono hard, firm e soft task, per cui un sistema real-time deve essere progettato per gestire i diversi tipi di task implementando strategie diverse:

Hard: devono essere garantiti off-line.

Firm: devono essere garantiti on-line, interrompendoli se la loro scadenza non può essere rispettata.

Soft: devono essere gestiti con l'obiettivo di ridurre al minimo il tempo di risposta medio.

Esempi di task in contesti ibridi

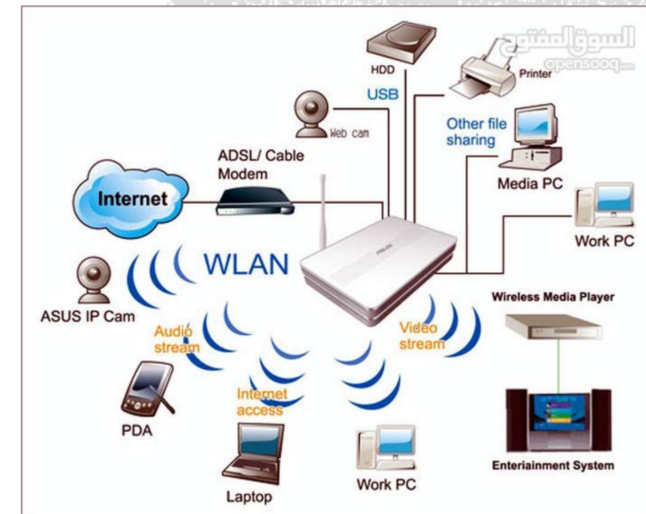
Tipici esempi di task appartenenti alle diverse categorie:

Hard:

- acquisizione di dati da sensori;
- fusione di dati e image processing;
- controllo a basso livello di sistemi

Firm:

- video playing;
- multimedia encoding/decoding;
- trasmissione dei dati.



Soft:

- Interprete dei comandi;
- I/O da periferiche.

Limiti dei comuni sistemi Real-Time

La gran parte dei sistemi real-time adottati per il controllo dei sistemi è basata su kernel, che sono riadattamenti degli analoghi utilizzati in sistemi time-sharing, con conseguenti svantaggi:

- **Multitasking**: la programmazione concorrente è gestita mediante system call, che non tengono esplicitamente considerazione del tempo, provocando ritardi imprevedibili.
- **Scheduling basato su priorità**: sono molto flessibili, ma hanno un numero limitato di livelli di priorità e non sempre i requisiti di tempo sono mappabili su tali intervalli. Inoltre, l'arrivo di un nuovo task potrebbe richiedere il ricalcolo di tutte le priorità.
- **Reattività agli interrupt esterni**: agli interrupt esterni è assegnata una priorità più alta di quella dei task da eseguire, che potrebbero rimanere sospesi per un tempo imprevedibile.

Limiti dei comuni sistemi Real-Time

La gran parte dei sistemi real-time adottati per il controllo dei sistemi è basata su kernel, che sono riadattamenti degli analoghi utilizzati in sistemi time-sharing, con conseguenti svantaggi:

IPC e sincronizzazione: i semafori binari normalmente usati provocano fenomeni quali inversione della priorità, attese a catena e deadlock, che rendono imprevedibile la durata dei task.

Kernel ridotti e context switch veloci: benché questi riducano l'overhead del sistema non garantiscono esplicitamente il rispetto delle deadline.

Assenza di un real-time clock: non offrono la possibilità di gestire in modo esplicito il timing dei task.

Desiderata per i sistemi Real-Time

Applicazioni di controllo complesse che richiedono forti vincoli temporali nell'esecuzione dei task devono essere supportati da sistemi operativi altamente **prevedibili**.

La **prevedibilità** può essere ottenuta solo introducendo cambiamenti radicali nei paradigmi di progettazione di base dei sistemi classici timesharing.

Poiché il codice di ogni task è noto a priori, il sistema è in grado di verificarne la **schedulabilità**, avendo come vincolo il rispetto delle deadline piuttosto che la riduzione del tempo medio di risposta.

Desiderata per i sistemi Real-Time

Fra i desiderata per un sistema operativo real-time si annoverano sicuramente:

Timeliness: la validità del risultato è intesa sia in termini di valore, che di tempo entro il quale esso è fornito.

Prevedibilità: affinché le performance siano accettabili, il sistema deve essere in grado di prevedere le conseguenze di qualunque scelta nello scheduling.

Efficienza: i sistemi real-time eseguono, generalmente, su hardware fortemente limitato in termini di spazio, peso, alimentazione, memoria e potenza di calcolo.

Robustezza: non devono collassare in caso di eccessi di carico.

Fault Tolerance: problemi hardware o software non devono mandare in crash l'intero sistema.

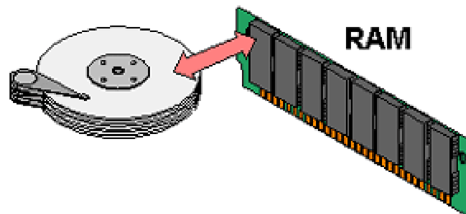
Manutenibilità: devono essere progettati secondo uno schema modulare in modo da semplificare modifiche e integrazioni.

La prevedibilità nei sistemi Real-Time

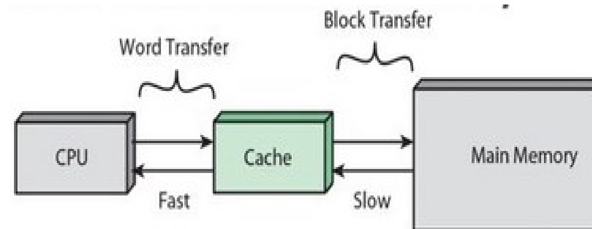
Il più importante requisito per un sistema hard real-time è la prevedibilità.

La prevedibilità di un sistema è influenzata da fattori hardware e software, tra cui:

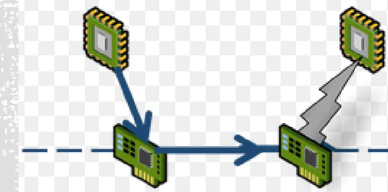
DMA



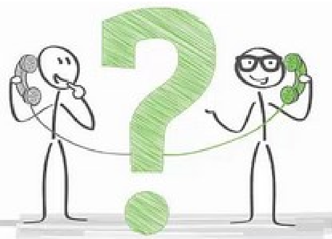
Cache



Interrupts

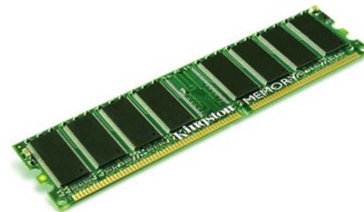


System Calls



Semaphores

Memory Management

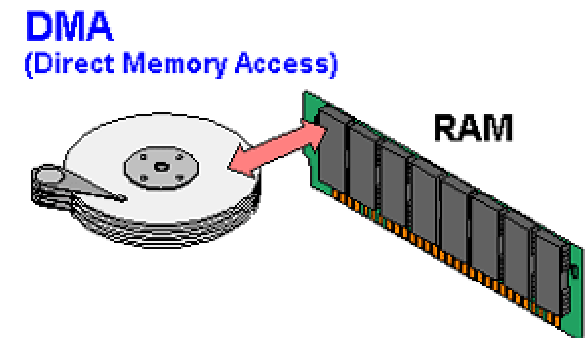


Programming Language

DMA

L'accesso diretto alla memoria (DMA) è una tecnica utilizzata da molti dispositivi periferici per trasferire i dati tra il dispositivo e la memoria principale.

Uno dei metodi più comuni è il **Cycle Stealing**, in cui il dispositivo DMA sottrae un ciclo di memoria alla CPU per eseguire un trasferimento di dati.
Il trasferimento I/O e l'esecuzione del programma vengono eseguiti in parallelo e se la CPU e il dispositivo DMA vanno in conflitto, il bus è assegnato al dispositivo DMA e la CPU attende.

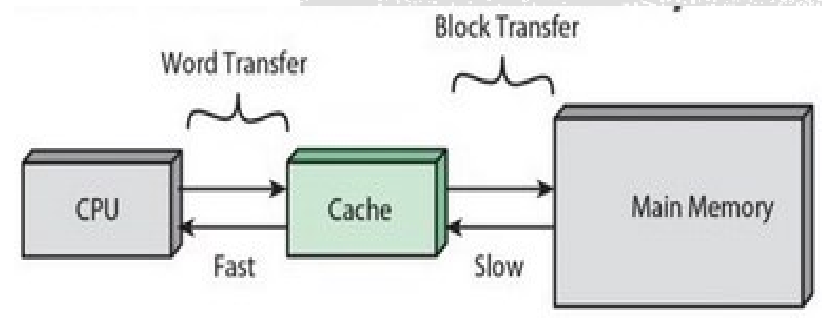


La tecnica **time-slice** divide un ciclo in due parti consecutive, allocandone una alla CPU e l'altra al dispositivo DMA, evitando il rischio che la CPU attenda un tempo indefinito

CACHE

La **cache** è una memoria veloce che viene inserita come buffer tra la CPU e la RAM per accelerare l'esecuzione dei processi.

Nei sistemi real-time la cache introduce un elevato grado di **non determinismo**, poiché la durata di un task varia a seconda del numero di cache hit e cache miss.

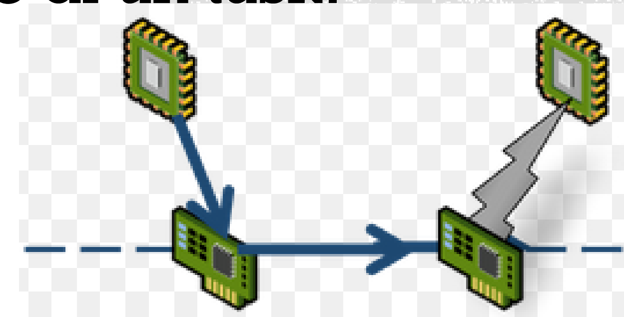


Inoltre, la cache rallenta le operazioni di scrittura in RAM, poiché è necessario garantire la **coerenza** dei dati.

Nei sistemi con **prelazione**, il comportamento della cache è fortemente influenzato dalla prelazione per via dei frequenti context switch.

Gli Interrupt

Gli interrupt generati dalle periferiche I/O rappresentano un grosso problema per la prevedibilità di un sistema in tempo reale perché, se non gestiti correttamente, possono introdurre ritardi notevoli nell'esecuzione di un task.



Approccio A: tutti gli interrupt esterni sono disabilitati a meno del timer. Le operazioni di I/O sono gestite direttamente dal processo mediante polling.

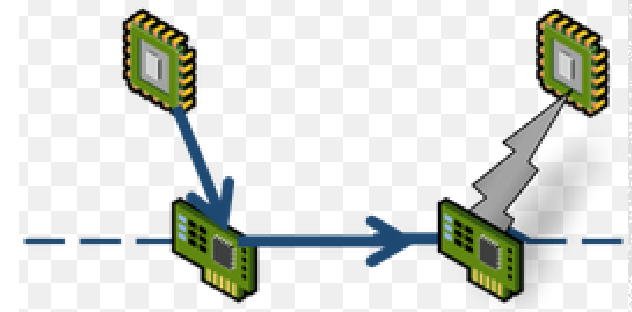
Vantaggi:
- eliminazione dei ritardi;
- calcolo preciso dei tempi di trasferimento dei dati;
- non bisogna modificare il kernel.

Svantaggi:
- minore efficienza nella gestione dell’I/O;
- l’applicazione deve conoscere i dettagli tecnici di ogni periferica a cui accede.

Gli Interrupt

Gli interrupt generati dalle periferiche I/O rappresentano un grosso problema per la prevedibilità di un sistema in tempo reale perché, se non gestiti correttamente, possono introdurre ritardi notevoli nell'esecuzione di un task.

Approccio B: tutti gli interrupt esterni sono disabilitati a meno del timer. Le operazioni di I/O sono gestite da routine del kernel attivate periodicamente dal timer.

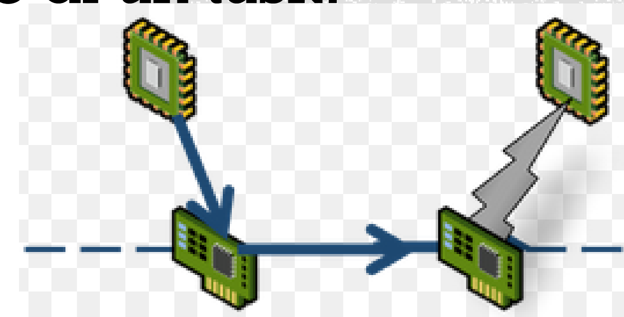


Vantaggi:
- eliminazione dei ritardi;
- In alcuni sistemi i dispositivi sono divisi in lenti e veloci;
- tutti i dettagli per la gestione dei dispositivi sono incapsulati nel kernel.

Svantaggi:
- il kernel è rallentato dai ritardi nella gestione dell'I/O;
- il kernel deve essere modificato ogniqualvolta viene aggiunto o eliminato un dispositivo.

Gli Interrupt

Gli interrupt generati dalle periferiche I/O rappresentano un grosso problema per la prevedibilità di un sistema in tempo reale perché, se non gestiti correttamente, possono introdurre ritardi notevoli nell'esecuzione di un task.



Approccio C: tutti gli interrupt esterni sono abilitati, ma i driver hanno un ruolo minimale. L'unica mansione di un driver è attivare un task, il cui scopo è gestire il dispositivo.
La priorità di questi task è inferiore rispetto a quelli con deadline stringenti.

Vantaggi:
- non riduce le prestazioni del kernel;
- Le durate dei task sono più facilmente prevedibili.

Le system call

La **prevedibilità** del sistema dipende anche da come vengono implementate le primitive del kernel. Per valutare con precisione il tempo di esecuzione nel caso peggiore di ogni attività, tutte le **system call** dovrebbero essere caratterizzate da un tempo di esecuzione limitato.

Al fine di semplificare l'analisi della **schedulabilità**, è desiderabile che ogni primitiva del kernel sia **prelazionabile**.

Qualsiasi sezione non eliminabile potrebbe ritardare l'attivazione o l'esecuzione di attività critiche, causando il mancato rispetto delle deadline.

I semafori

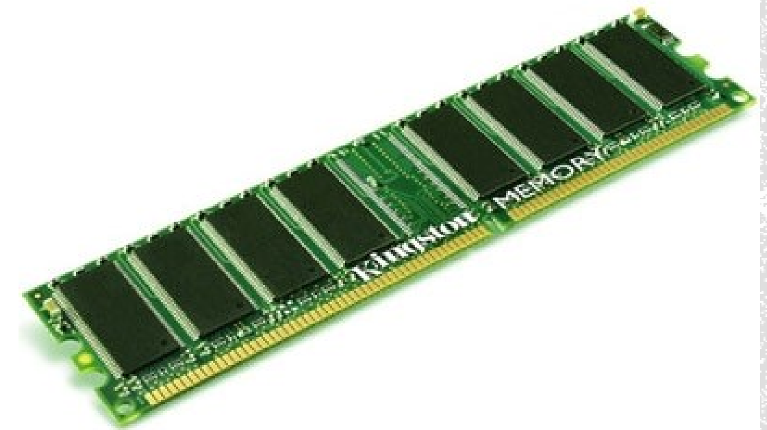
Il tipico meccanismo dei **semafori** utilizzato nei sistemi operativi tradizionali non è adatto per l'implementazione di applicazioni in tempo reale poiché è soggetto al fenomeno di **inversione di priorità**, che si verifica quando un'attività ad alta priorità viene bloccata da un'attività a bassa priorità per un intervallo di tempo illimitato.
Esistono soluzioni efficaci a questo problema, quali:

- Basic Priority Inheritance;
  https://www.geeksforgeeks.org/priority-inheritance-protocol-pip-in-synchronization/
- Priority Ceiling;
  https://www.geeksforgeeks.org/priority-ceiling-protocol/
- Stack Resource Policy.
  https://embeddedartistry.com/fieldmanual-terms/stack-resource-policy/

La gestione della memoria

Gli **schemi di paginazione** comuni non sono adatti alle applicazioni in tempo reale soggette a rigidi vincoli temporali a causa dei ritardi notevoli e imprevedibili causati da page fault e sostituzione di pagine.

Le soluzioni tipicamente adottate nella maggior parte dei sistemi in tempo reale si basano sulla **segmentazione** della memoria con uno schema di gestione della memoria prefissato.



Il partizionamento statico è particolarmente efficiente quando i programmi applicativi richiedono quantità di memoria simili.

I linguaggi di programmazione

Gli attuali **linguaggi di programmazione** non sono abbastanza espressivi da caratterizzare determinati comportamenti temporali e, quindi, non sono adatti per realizzare applicazioni prevedibili in tempo reale.

Recentemente, sono stati proposti nuovi linguaggi di alto livello per supportare lo sviluppo di applicazioni hard real-time:

-Real-Time Euclid;
- Real-Time Concurrent C.

Real-time Euclid

Il linguaggio Real-Time Euclid impone al programmatore di specificare limiti temporali ed eccezioni di timeout in tutti i loop, le attese e le istruzioni di accesso ai dispositivi. Inoltre, impone diverse restrizioni di programmazione, quali:

Assenza di strutture dati dinamiche: array dinamici, puntatori e stringhe di lunghezza arbitraria non sono ammesse poiché rendono imprevedibili i tempi di allocazione e deallocazione.

Assenza di ricorsione: la ricorsione non premette la stima del tempo di esecuzione delle chiamate ricorsive nell'analisi della schedulabilità.

Cicli temporizzati: il programmatore deve specificare il numero massimo di iterazioni in ogni ciclo al fine di rendere possibile la stima della durata di un task.

Real-Time Concurrent C

Questo linguaggio estende il concurrent C fornendo delle facility per indicare la periodicità/aperiodicità e le deadline.

Esso fornisce costrutti del tipo:

> **within deadline** *(d) statement-1*
>
> [**else** *statement-2*]

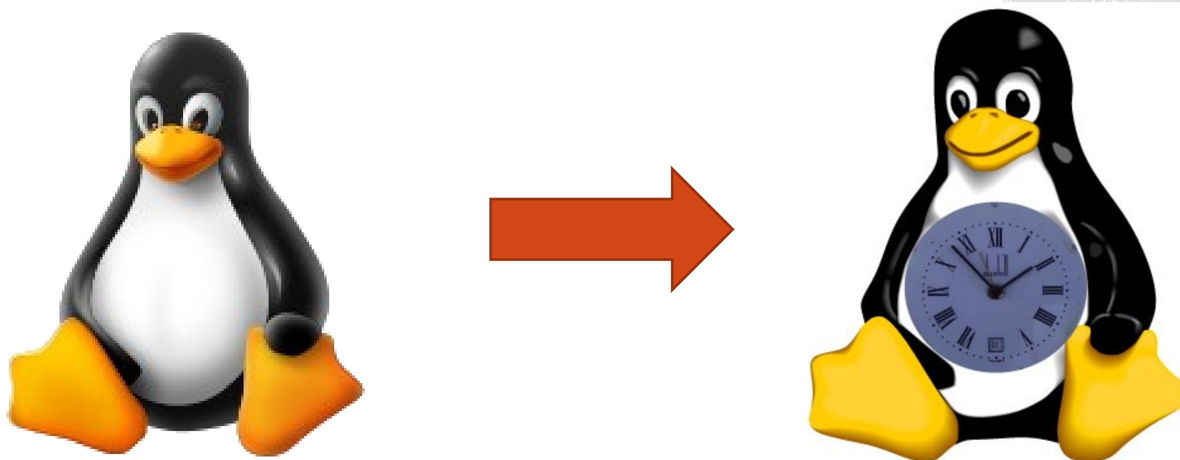
Che permettono di specificare le istruzioni da eseguire nel caso in cui la deadline sia (non sia) rispettata.

A differenza di Real-Time Euclid, che è stato progettato per supportare la verifica statica di sistemi real-time, Real-Time Concurrent C è orientato a sistemi dinamici, in cui è possibile attivare task in fase di esecuzione.

Da Linux a Real-Time Linux

È interessante valutare come è possibile migrare da una piattaforma Linux non real-time ad una piattaforma specificatamente RTOS implementata su quest'ultima.

In particolare, è importante analizzare le componenti di Linux che riducono il determinismo riducendone la prevedibilità ed il modo, in cui queste devono essere modificate, così da renderlo un sistema operativo adatto a contesti real-time.



Da Linux a Real-Time Linux

il Kernel 2.6 mostra sicuramente un'apertura più evidente verso il real-time rispetto alle precedenti versioni:

- Modifica della struttura generale dello scheduler (da un'unica coda per tutti i task del sistema ad una coda per ognuno dei 140 livelli di priorità su ogni CPU);
- Frequenza interna del clock portata da 100 Hz (kernel 2.4) a 1000 Hz;
- Parte del kernel è divenuta preemptable;
- Eliminati numerosi "Big Locks";
- Compatibilità con Preemption;
- Migliorato il sistema di gestione di I/O.

Da Linux a Real-Time Linux

Alle motivazioni strutturali, si aggiungono, inoltre, motivazioni quali:

- Linux è un sistema operativo di alto livello ed è Open Source;
- È molto ben documentato;
- Nasce nell'ambiente universitario ed è naturalmente inserito in molti contesti di ricerca;
- È compatibile con i maggiori standard relativi al software (POSIX etc…);
- Ha buone prestazioni già in versioni base (senza modifiche strutturali);
- La struttura monolitica lo rende adatto ad essere utilizzato in scenari Embedded;
- Il mondo dell'industria è interessato ad applicazioni di Linux in svariati settori: GPS (Global Positioning System), HDTV (High Definition TV), DVB-RCS, cellulari, …
- Sono attualmente disponibili numerose implementazioni di architetture software Open Source per modificare Linux dotandolo di funzionalità real-time da cui attingere spunto…

REAL TIME SCHEDULING IN LINUX

- Nelle versioni correnti di Linux è possible implementare algoritmi di scheduling real time andando ad agire sulla priorità dei processi
- I processi normali di Linux hanno priorità crescenti da zero (idle) a 39
- E' possible assegnare a processi real time priorità crescenti da 40 a 139
  - Attenzione: non c'è alcun controllo che impedisca di giungere a situazioni non volute, come deadlock o starvation
- Per modificare la priorità di un processo verso un valore real time è possible utilizzare il comando *chrt*

CHRT

- chrt [options] --pid <priority> <pid>

Get policy:

chrt [options] -p <pid>

Policy options:

-b, --batch set policy to SCHED_BATCH

-d, --deadline set policy to SCHED_DEADLINE

-f, --fifo set policy to SCHED_FIFO

-i, --idle set policy to SCHED_IDLE

-o, --other set policy to SCHED_OTHER

-r, --rr set policy to SCHED_RR (default)

Scheduling options:

-R, --reset-on-fork set SCHED_RESET_ON_FORK for FIFO or RR

-T, --sched-runtime <ns> runtime parameter for DEADLINE

-P, --sched-period <ns> period parameter for DEADLINE

-D, --sched-deadline <ns> deadline parameter for DEADLINE

Other options:

-a, --all-tasks operate on all the tasks (threads) for a given pid

-m, --max show min and max valid priorities

-p, --pid operate on existing given pid

-v, --verbose display status information

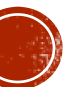
-h, --help display this help

PS

- Con il comando ps è possibile osservare le priorità dei processi e anche il loro scheduling

ps –A –l

```
F S  UID  PID  PPID C PRI  NI BIT   SZ WCHAN  TTY        TIME CMD
4 S    0    1     0 1  19   0  -  2995 ep_po+ ?      00:00:01 init
1 S    0    2     0 0  19   0  -     0 kthre+ ?      00:00:00 kthreadd
1 S    0    3     2 0  19   0  -     0 smpbo+ ?      00:00:00 ksoftirqd/0
1 S    0    4     2 0  19   0  -     0 worke+ ?      00:00:00 kworker/0:0
1 S    0   10     2 0 139   0  -     0 smpbo+ ?      00:00:00 migration/0
1 S    0   11     2 0 139   0  -     0 smpbo+ ?      00:00:00 migration/1
1 S    0   12     2 0  19   0  -     0 smpbo+ ?      00:00:00 ksoftirqd/1
```



Distribuzioni di Real-Time Linux

Negli anni sono nate diverse distribuzioni real-time commerciali di Linux:

RTLinuxPro(FSMLabs). RTCore fornisce un ambiente real-time POSIX in cui Linux esegue come task a bassa priorità. Limiti prestazionali – quelli dell' hardware. Latenza di scheduling sotto 20 microsecondi sulla maggior parte delle piattaforme.

MontaVista RTLinux. Basato su miglioramenti a MontaVista Linux. Preemptable kernel + real-time scheduler + frameworks.

FlightLinux Versione real-time di Linux adattata ad applicazioni spaziali al Goddard Space Flight Center e testata sullo Space Shuttle.

Microsistema operativo real-time (non Linux-based) che esegue Linux come processo a bassa priorità. BlueCat RT è incentrato sulle basse latenze di interrupt raggiunte grazie al core ridottissimo e altamente performante.

Distribuzioni di Real-Time Linux

Negli anni sono nate anche diverse distribuzioni real-time open source di Linux:

RTLinux. Hard Real Time mini sistema operativo che esegue Linux come processo a minima priorità rendendolo totalmente preemptable. L'ultima versione supporta user-space RT programming. La versione MiniRTL è adatta ad utilizzi embedded.

RTAI (Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano). Simile ad RTLinux ma con varie funzionalità in più tra cui LXRT Layer che permette di controllare task real-time dallo user-space memory-protected di Linux . AtomicRTAI è la versione a ridotto footprint.

RT Linux

RT-Linux è un sistema operativo nel quale un piccolo kernel real-time coesiste con il kernel di Linux (like-POSIX).

L'intenzione è quella di fare uso dei servizi sofisticati ed altamente ottimizzati (per quel che concerne il comportamento medio) di un sistema di computer standard time-shared, mentre ancora si permette alle funzioni real-time di operare in un ambiente prevedibile ed a bassa latenza.

In RT-Linux un task real-time è eseguito su un kernel non real-time, che è concepito come un task a più bassa priorità, utilizzando un livello di virtual-machine per rendere il kernel standard completamente prelazionabile.