

[Schema della lezione](#)

[Algoritmi round-robin](#)

[Algoritmi priority-driven](#)

[Alg. EDF, LRT, LST](#)

[Algoritmi RM e DM](#)

[Ottimalità di EDF](#)

SERT'20

R4.1

Lezione R4

Algoritmi priority-driven

Sistemi embedded e real-time

8 ottobre 2020

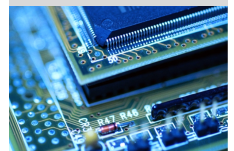
Marco Cesati

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

Di cosa parliamo in questa lezione?

In questa lezione descriviamo i tipi di algoritmi più utilizzati per realizzare gli schedulatori dei sistemi real-time che fanno uso di priorità dei job

- 1 Algoritmi di tipo round-robin
- 2 Algoritmi di tipo priority-driven
- 3 Gli algoritmi EDF, LRT, LST
- 4 Gli algoritmi RM e DM



[Schema della lezione](#)

[Algoritmi round-robin](#)

[Algoritmi priority-driven](#)

[Alg. EDF, LRT, LST](#)

[Algoritmi RM e DM](#)

[Ottimalità di EDF](#)

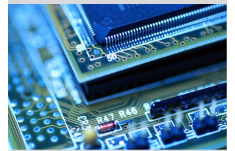
SERT'20

R4.2

Algoritmi round-robin

Un algoritmo di schedulazione è detto essere *round-robin* quando i job sono gestiti tramite code FIFO (First-In, First-Out)

- Un job è inserito in fondo ad una coda d'esecuzione quando esso diviene pronto per l'esecuzione (istante di rilascio)
- Dovendo scegliere un job tra quelli in attesa in una coda FIFO, lo scheduler seleziona quello in testa, ossia il primo inserito in ordine di tempo, e lo rimuove dalla coda
- Ogni job esegue al massimo per un intervallo di tempo predefinito chiamato *time slice* o *quantum*, poi se necessario viene interrotto ed inserito nuovamente in fondo alla coda
- Se nella coda vi sono n job, un gruppo di n *time slice* è chiamato *round*: ciascun job ottiene un *time slice* ogni round



Algoritmi round-robin (2)

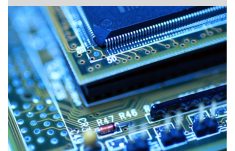
Un algoritmo di schedulazione *round-robin* è detto essere *pesato* (o *weighted*) se a job differenti possono essere assegnati pesi differenti che influiscono sulle quote di tempo di processore

- Un *round* è definito come il numero di *time slice* pari alla somma dei pesi di tutti i job in una coda
- Un job con peso w ottiene w *time slice* in ogni round
- I job con peso maggiore ottengono più tempo di processore di quelli con peso minore

Quali sono i vantaggi degli scheduler round-robin?

- Il sistema assegna il processore in maniera "equa" (ad es.: *time sharing* dei SO general-purpose), *No starvation*
- Lo scheduler utilizza semplici code FIFO, quindi è molto veloce (ha basso overhead)

↳ ruba poco tempo macchina



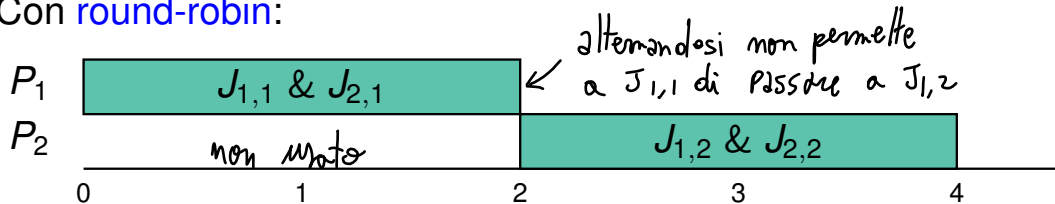
Svantaggi degli algoritmi round-robin

Quali sono i più evidenti svantaggi degli scheduler round-robin?

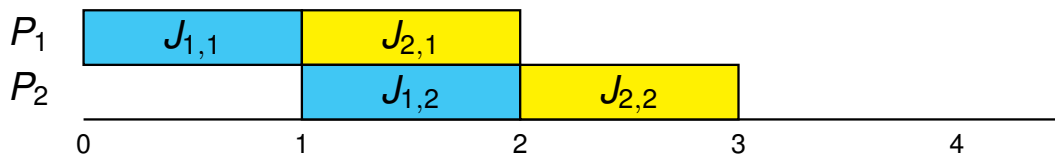
- 1) Non considerano eventuali scadenze dei job
- 2) Non gestiscono bene job con vincoli di precedenza (in fase)

Esempio: quattro job con istante di rilascio 0 e tempo d'esecuzione 1: $J_{1,1} \prec J_{1,2}$, $J_{2,1} \prec J_{2,2}$. $J_{1,1}$ e $J_{2,1}$ eseguono sul processore P_1 , $J_{1,2}$ e $J_{2,2}$ eseguono sul processore P_2

Con round-robin:



Senza round-robin:



Svantaggi degli algoritmi round-robin (2)

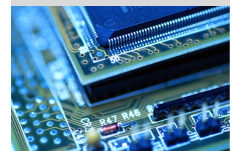
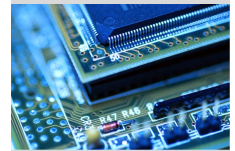
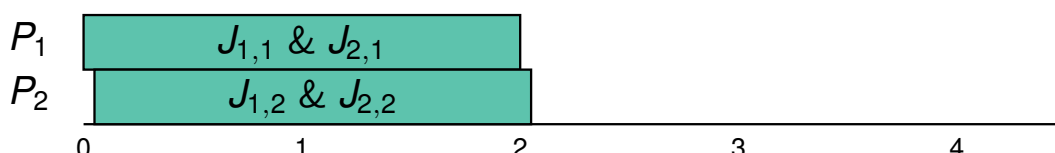
Se i vincoli di precedenza sono un problema, perché gli algoritmi di tipo round-robin sono utilizzati soprattutto nei sistemi Unix in cui i processi sono spesso collegati tramite "pipe"?

↳ canale di comunicazione logica tra processi

La dipendenza tra job di una "pipe" non è un vincolo di precedenza!

- In un vincolo di precedenza $J_a \rightarrow J_b$, J_b non può iniziare prima del completamento di J_a
- In una pipe $J_a | J_b$, J_b consuma i dati via via prodotti da J_a

Nell'esempio precedente, se al posto dei vincoli di precedenza si introducono due pipe $J_{1,1} | J_{1,2}$ e $J_{2,1} | J_{2,2}$:



Algoritmi priority-driven

Un algoritmo di schedulazione è detto *priority-driven* se ha la caratteristica di non lasciare mai intenzionalmente inutilizzato un processore o un'altra risorsa

Se P_i libero e processore può farlo, con priority driven deve usarlo!

Alcune caratterizzazioni equivalenti degli algoritmi *priority-driven*:

- Una risorsa attiva (processore) o passiva è inutilizzata solo quando non esistono job che richiedono la risorsa pronti per l'esecuzione (→ algoritmi *work conserving*)
- Le decisioni dello scheduler vengono effettuate all'occorrenza di specifici eventi, ad es. un job diviene pronto per l'esecuzione (→ algoritmi *event driven*)
- Gli algoritmi di schedulazione prendono decisioni ottimali a livello locale, ossia del singolo processore o risorsa (→ algoritmi *greedy scheduling*)

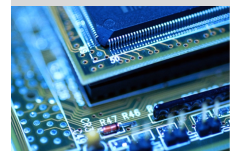
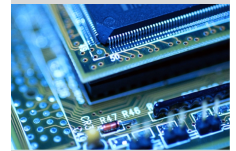
Modello di riferimento *per studiarli*

Studieremo alcuni algoritmi priority-driven utilizzando:

- Modello a task periodici (o sporadici)
- Numero di task prefissato *approfondito in futuro*
- Un singolo processore (od un sistema statico)
- Task indipendenti: nessun vincolo di precedenza e
- nessuna risorsa condivisa
- Nessun job aperiodico

Nel modello a task sporadici ciascun task è caratterizzato da un *periodo* corrispondente al *minimo* intervallo tra gli istanti di rilascio dei job

Più avanti alcuni di questi vincoli saranno rimossi !!



dall'algoritmo di schedulazione secondo criteri oggettivi, non dal sistemista!

Priorità

Ogni algoritmo **priority-driven** può essere realizzato assegnando dinamicamente valori numerici detti **priorità** ai job

La schedulazione dipende, oltre che dal modello del carico e del sistema, dalla **lista dei job ordinata per priorità**; quindi gli algoritmi sono anche chiamati **list scheduling**

Molti scheduler non real-time sono **priority-driven**: *(molto ampi, anche alg. non idonei al nostro scopo)*

- algoritmi **FIFO** (First In First Out): priorità inversamente proporzionale all'istante di rilascio
- algoritmi **LIFO** (Last In First Out): priorità direttamente proporzionale all'istante di rilascio
- algoritmi **SETF** (Shortest Execution Time First): priorità inversamente proporzionale al tempo d'esecuzione
- algoritmi **LETf** (Longest Execution Time First): priorità direttamente proporzionale al tempo d'esecuzione

Gli algoritmi round-robin sono priority-driven? Sì! Round Robin ha coda FIFO

Possiamo pensare che la **priorità** di ogni job varia dinamicamente in funzione della sua posizione nella coda FIFO
ma non adatti a contesti RT.

Tipi di priorità degli algoritmi

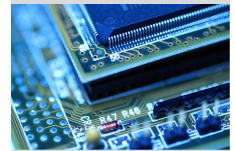
Gli algoritmi **priority-driven** possono essere:

- a **priorità fissa** (**fixed-priority**): la priorità di tutti i job di un task è identica; di conseguenza, la priorità è in effetti assegnata a ciascun task e non cambia
- a **priorità dinamica** (**dynamic-priority**): la priorità dei job di uno stesso task può cambiare

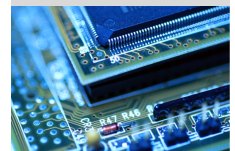
A propria volta, gli algoritmi a **priorità dinamica** possono essere di due sottotipi:

- **dinamici a livello di task** e **statici a livello di job** (**task-level dynamic-priority**): la priorità di un job già rilasciato e pronto per l'esecuzione non cambia fino al suo termine
- **dinamici a livello di job** (**job-level dynamic-priority**): la priorità di un job può cambiare dopo il suo rilascio

(es: Round Robin quando va in fondo alla coda)



*qui non parlo
mai di
scadenze*



Algoritmi priority-driven fondamentali

I tipi fondamentali di algoritmi priority-driven:

- **FIFO**: priorità invers. proporzionale all'istante di rilascio
- **LIFO**: priorità proporzionale all'istante di rilascio
- **EDF**: priorità invers. proporzionale alla scadenza assoluta
- **LST**: priorità invers. proporzionale allo slack dei job
- **RM**: priorità invers. proporzionale al periodo di riposo dei job
- **DM**: priorità invers. proporzionale alla scadenza relativa

Di che tipo sono questi algoritmi?

- **FIFO**, **LIFO**: priorità dinamica a livello di task
- **EDF**: priorità dinamica a livello di task
- **LST**: priorità dinamica a livello di job
- **RM**, **DM**: priorità statica

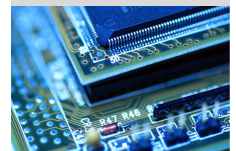
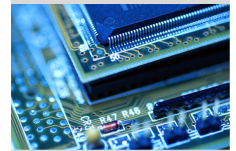
Algoritmi ottimali su singolo processore

- **EDF** (Earliest Deadline First): la priorità dei job è direttamente proporzionale alla vicinanza della scadenza
- **LRT** (Latest Release Time): è l'inverso di **EDF**, poiché inverte i ruoli degli istanti di rilascio e delle scadenze e schedula i job iniziando dall'ultima scadenza fino al presente
- **LST** (Least Slack Time First) o **MLF** (Minimum Laxity First): la priorità è inversamente proporzionale alla *slack* dei job, ossia il valore ottenuto sottraendo dalla scadenza del job il tempo presente ed il tempo richiesto per completare il job

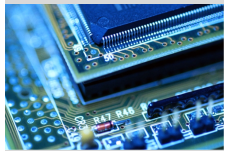
Teorema (Dertouzos 1974, Mok 1983)

Avendo un solo processore, job interrompibili (con preemption), e nessuna contesa sulle risorse condivise, gli algoritmi **EDF**, **LRT** e **LST** producono una **schedulazione fattibile** di un insieme di job con vincoli temporali arbitrari se e solo se tale insieme di job è schedulabile

→ rispetto scadenze



Esempio di schedulazione di EDF



Job interrompibili J_1, \dots, J_4 :

EDF non li "chiede in anticipo".

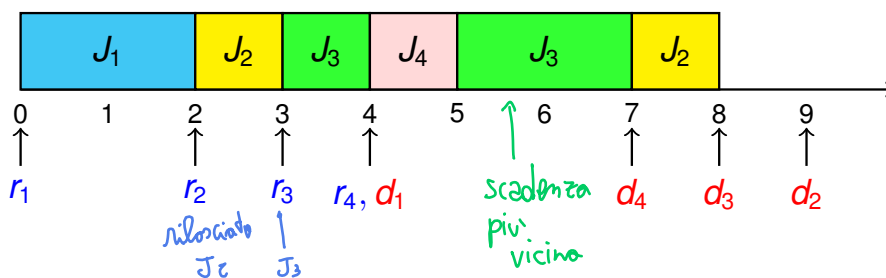
i	1	2	3	4
r_i	0	2	3	4
d_i	4	9	8	7
e_i	2	2	3	1

rilascio
scadenze assolute
tempi esecuzioni

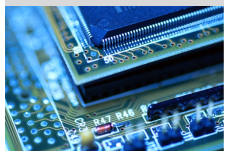
Lavoro da
compiere:

\bar{e}	0	1	2	3	4	5	6	7	8	9
J_1	2	1	0	0	0	0	0	0	0	0
J_2	2	2	2	1	1	1	1	1	0	0
J_3	3	3	3	3	2	2	1	0	0	0
J_4	1	1	1	1	1	0	0	0	0	0

non ancora rilasciati → deve scegliere chi ha scadenza più vicina



Algoritmo LRT



- L'algoritmo di schedulazione **LRT** (Latest Release Time) inverte il ruolo degli istanti di rilascio e delle scadenze, e schedula i job partendo dall'ultima scadenza fino al presente
- L'algoritmo è anche noto come **EDF inverso**
- La priorità assegnata ad un job è proporzionale al suo istante di rilascio: più avanti è l'istante di rilascio, maggiore è la priorità

Qual è il vantaggio di LRT rispetto a EDF? è più predicibile

LRT cerca di completare i job in corrispondenza della loro scadenza, quindi è meno aggressivo nell'uso del processore e riduce l'incertezza sull'istante di completamento dei job

È un algoritmo di tipo priority-driven? **No!** NON È WORK-CONSERVATIVE! ⇒

NO
PRIORITY
DRIVEN

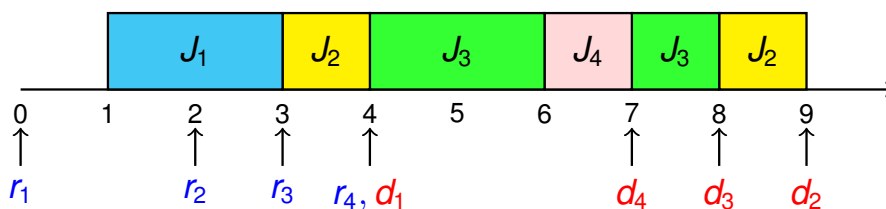
L'algoritmo potrebbe lasciare un processore inutilizzato anche in presenza di job pronti per l'esecuzione

Esempio di schedulazione di LRT

Job interrompibili J_1, \dots, J_4 :

i	1	2	3	4
r_i	0	2	3	4
d_i	4	9	8	7
e_i	2	2	3	1

	ξ	0	1	2	3	4	5	6	7	8	9
Lavoro compiuto:	J_1	2	2	1	0	0	0	0	0	0	0
	J_2	2	2	2	2	1	1	1	1	1	0
	J_3	3	3	3	3	3	2	1	1	0	0
	J_4	1	1	1	1	1	1	1	0	0	0



è EDF al contrario, traslata verso le scadenze. Non è alg. online, devo sapere cosa avverrà in **Future**. (EDF non lo richiede)

Algoritmo LST

- Il valore di **slack** di un job è la differenza tra l'istante di scadenza e la somma del tempo attuale e del tempo ancora occorrente per completare l'esecuzione
- L'algoritmo **LST** assegna priorità più alta ai job aventi valore di **slack** minore (più vicini a mancare la scadenza, se job ha $slack=0$ ha max. priorità e va eseguito tutto e subito).

Qual è la logica di questo algoritmo?

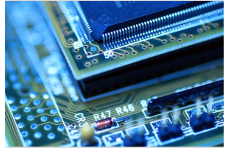
Lo **slack** è una misura di quanto tempo un job può permettersi di non essere eseguito senza mancare la propria scadenza

Qual è lo svantaggio di LST rispetto a EDF e LRT?

LST richiede di conoscere in anticipo i tempi di esecuzione (massimi) di tutti i job

Questa condizione è spesso molto difficile da rispettare

Esempio di schedulazione di LST



Schema della lezione

Algoritmi round-robin

Algoritmi priority-driven

Alg. EDF, LRT, LST

Algoritmi RM e DM

Ottimalità di EDF

SERT'20

R4.17

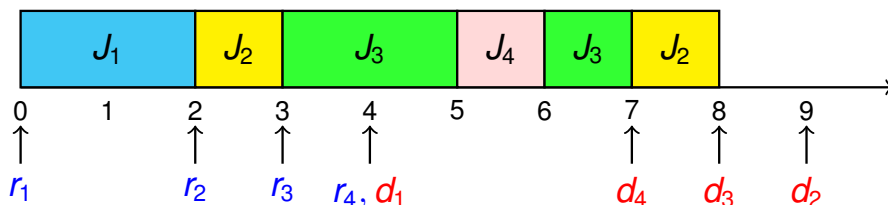
Job interrompibili J_1, \dots, J_4 :

i	1	2	3	4
r_i	0	2	3	4
d_i	4	9	8	7
e_i	2	2	3	1

Lavoro da compiere & slack:

slack (non cambia per job in esecuzione solo per quelli che non lavorano)

$\bar{e}; s$	0	1	2	3	4	5	6	7	8	9
J_1	2;2	1;2	0;2	0;1	0;0	0;-1	0;-2	0;-3	0;-4	0;-5
J_2	2;7	2;6	2;5	1;5	1;4	1;3	1;2	1;1	0;1	0;0
J_3	3;5	3;4	3;3	3;2	2;2	1;2	1;1	0;1	0;0	0;-1
J_4	1;6	1;5	1;4	1;3	1;2	1;1	0;1	0;0	0;-1	0;-2



Varianti della schedulazione LST

ottimale \Leftrightarrow scheduler scambia i job nei tempi "conetti".

Nell'algoritmo **LST** (Least Slack Time) la priorità di un job è inversamente proporzionale al valore di **slack** $d - t - x$ (d = scadenza assoluta, t = tempo corrente, x = tempo d'esecuzione rimanente)

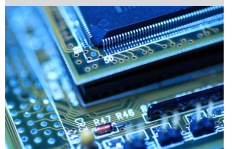
In realtà ne esistono due varianti:

- **Nonstrict LST**: lo scheduler è invocato e le priorità dei job sono cambiate solo come conseguenza del rilascio o della conclusione di un job, o di un tick periodico (NON OTTIMALE)
- **Strict LST**: le priorità sono modificate continuamente, e lo scheduler è invocato ogni volta che un job acquisisce una priorità maggiore di quella del job in esecuzione

L'algoritmo **strict LST** è utilizzato raramente perché

- è più complesso
- ha un overhead maggiore dovuto all'aggiornamento dei valori di slack e ai context switch

Ho J_1 e J_2 stesso slack, parte J_1 , diminuisce slack J_2 e va J_2 , poi riparte a J_1 , poi J_2 ... tempi piccoli tale che \approx algoritmo continuo.



Schema della lezione

Algoritmi round-robin

Algoritmi priority-driven

Alg. EDF, LRT, LST

Algoritmi RM e DM

Ottimalità di EDF

SERT'20

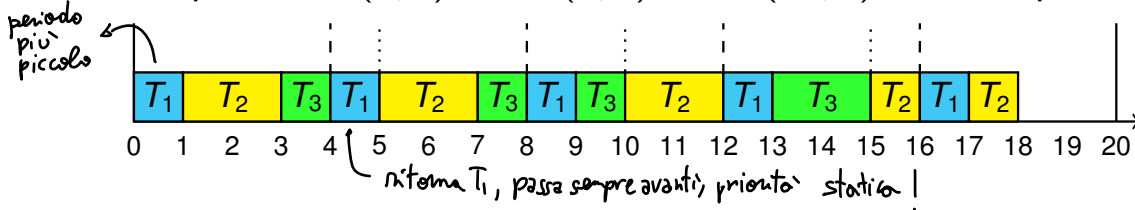
R4.18

Algoritmo Rate Monotonic (Liu, Layland 1973)

L'algoritmo **Rate Monotonic (RM)** assegna la priorità di un task in modo proporzionale alla sua **frequenza (rate)**, definita come l'inverso del suo **periodo** (priorità fissa a livello di task)

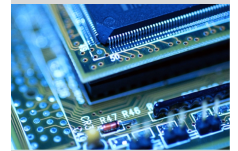
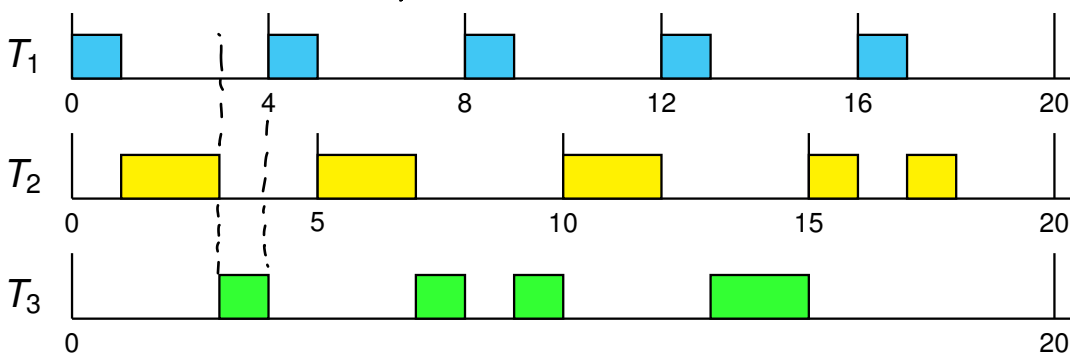
$\phi = 0$, $scad = \text{periodo}$

Esempio: $T_1 = (4, 1)$, $T_2 = (5, 2)$, $T_3 = (20, 5)$, interrompibili



Rappresentazione alternativa:

MAI SOVRAPPosti!



Noi
useremo
questa
notazione

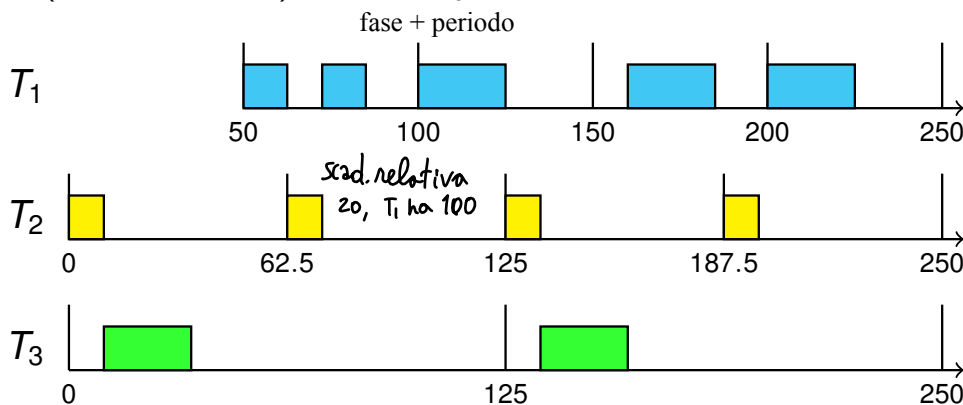
Algoritmo Deadline Monotonic (Leung, Whitehead 1982)

Nell'algoritmo **Deadline Monotonic (DM)** la priorità di un task è inversamente proporzionale alla sua scadenza relativa

Se le scadenze sono implicite, va come RM

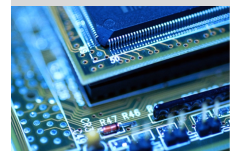
fase | periodo | tempo esecuzione | scadenza relativa

Esempio: $T_1 = (50, 50, 25, 100)$, $T_2 = (0, 62.5, 10, 20)$,
 $T_3 = (0, 125, 25, 50)$, interrompibili



In quale caso gli algoritmi RM e DM coincidono?

Nel caso: deadline relativa proporzionale al periodo (es: scadenza = $\frac{\text{periodo}}{2}$)
quando sono uguali e un sottocaso!

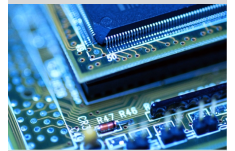
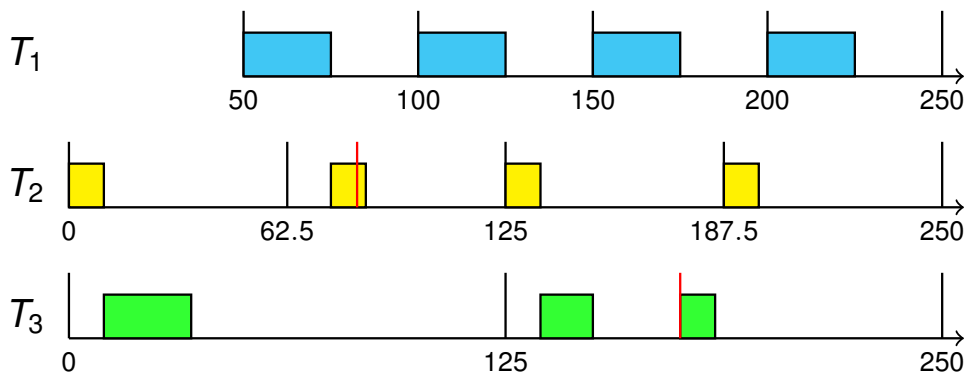


Algoritmo DM migliore di RM

In generale, l'algoritmo **DM** è migliore di **RM**:

- Se la schedulazione **DM** non è fattibile, anche la schedulazione **RM** non è fattibile
- Esistono esempi in cui la schedulazione **DM** è fattibile mentre la schedulazione **RM** non è fattibile

Nell'esempio precedente, la schedulazione **RM** non è fattibile:



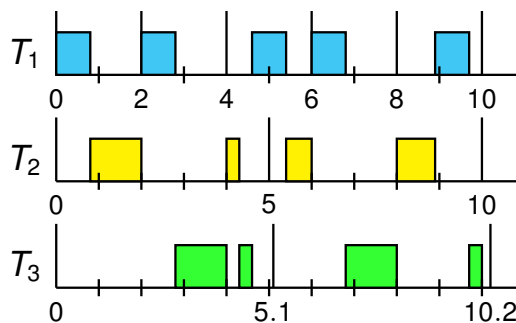
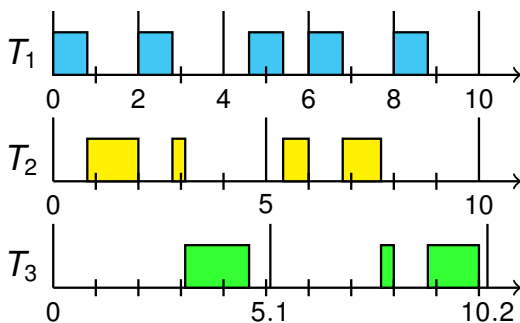
Confronto tra algoritmi di schedulazione

RIFARE A CASA

Siano $T_1 = (2, 0.8)$, $T_2 = (5, 1.5)$, $T_3 = (5.1, 1.5)$ interrompibili

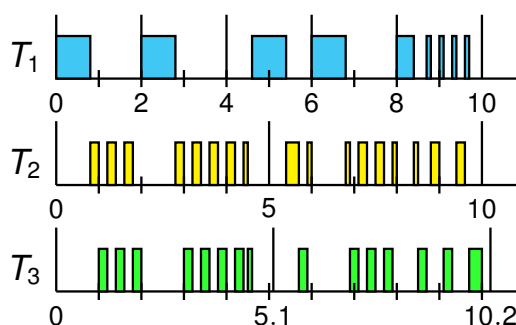
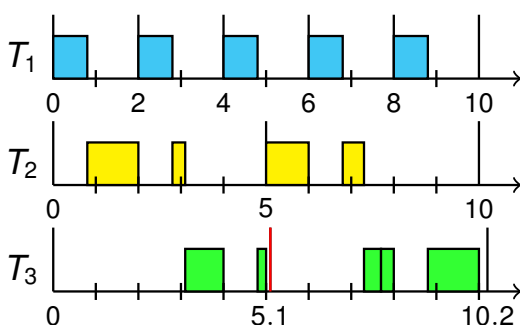
facile e
ottimale **EDF** (deterministico)

Nonstrict LST



RM/DM

Strict LST



manca scadenza,
NO OTTIMALE! usati perché più semplici di EDF, anche per studiarli!



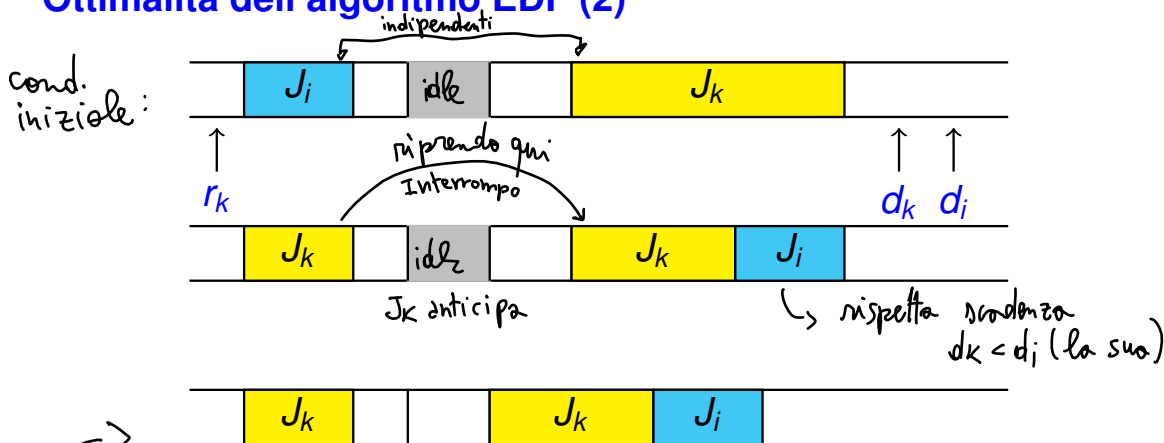
Ottimalità dell'algoritmo EDF, perché?

L'algoritmo EDF è ottimale nel senso che riesce sempre a trovare una schedulazione fattibile di un insieme di job interrompibili e indipendenti su un singolo processore, ovviamente a condizione che tale schedulazione esista

Sketch della dimostrazione (Dertouzos, 1974):

- Ogni schedulazione fattibile di un insieme di job arbitrari può essere trasformata in una schedulazione prodotta dall'algoritmo EDF (dammi schedulazione fattibile, la faccio con EDF)
- Sia J_i schedulato prima di J_k con $d_i > d_k$ (J_k priorità $> J_i$, controllo di come fa EDF, lo viola)
- Se r_k è oltre l'intervallo in cui è schedulato J_i , i due job rispettano l'algoritmo EDF; assumiamo che r_k è prima dell'intervallo in cui è schedulato J_i (se dopo, non era violazione)
- Scambiamo tra loro J_i e J_k (se necessario utilizzando l'interrompibilità dei job per tenere conto di lunghezze diverse degli intervalli di tempo)
- Ora J_i e J_k rispettano le priorità EDF

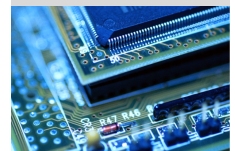
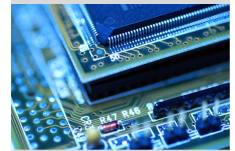
Ottimalità dell'algoritmo EDF (2)



Scambiare sistematicamente di posto tutti i job che non rispettano le priorità EDF è sufficiente a trasformare la schedulazione in quella prodotta dall'algoritmo EDF? **No!**

Potrebbero rimanere intervalli di tempo in cui il processore è inutilizzato pur essendoci job pronti per l'esecuzione ma schedulati dopo EDF e priority driven

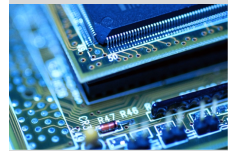
È sempre possibile anticipare l'esecuzione di uno o più job in modo da eliminare questi casi



Algoritmi priority-driven e job non interrompibili

Algoritmi
priority-driven

Marco Cesati



Schema della lezione

Algoritmi round-robin

Algoritmi priority-driven

Alg. EDF, LRT, LST

Algoritmi RM e DM

Ottimalità di EDF

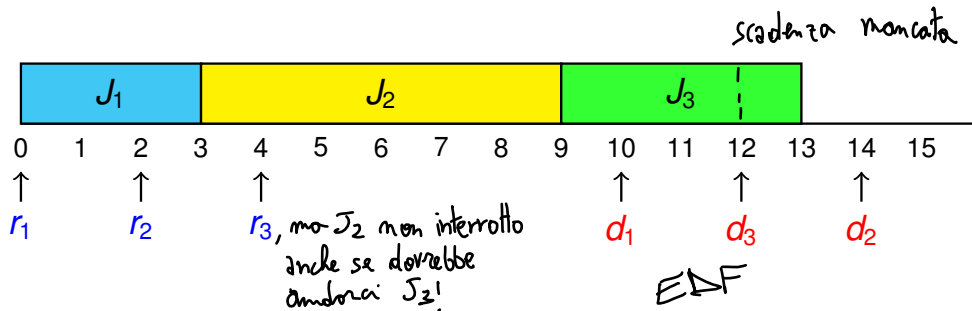
SERT'20

R4.25

Gli algoritmi **priority-driven** (ossia **work-conserving**), così come **LRT**, **non** sono ottimali se i job sono **non interrompibili**
EDF

Scheduliamo J_1 , J_2 e J_3 non interromp.:

i	1	2	3
r_i	0	2	4
d_i	10	14	12
e_i	3	6	4



Eppure una **schedulazione fattibile non priority-driven** esiste:

