Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2023/24

Approcci allo scheduling real-time

prof. Stefano Caselli
stefano.caselli@unipr.it
http://rimlab.ce.unipr.it

Approcci principali allo scheduling real-time

- Schedulatori di tipo *clock-driven* (o *time-driven*):

  Le decisioni di scheduling sono prese *in specifici istanti di tempo*, tipicamente scelti *a priori*

- Schedulatori di tipo *priority-driven*:

  Le decisioni di scheduling sono prese *quando si verificano particolari eventi* nel sistema, ad es.:
  - un job diventa pronto
  - il processore diventa libero

Schedulatori work-conserving

- Scheduling *work-conserving*:

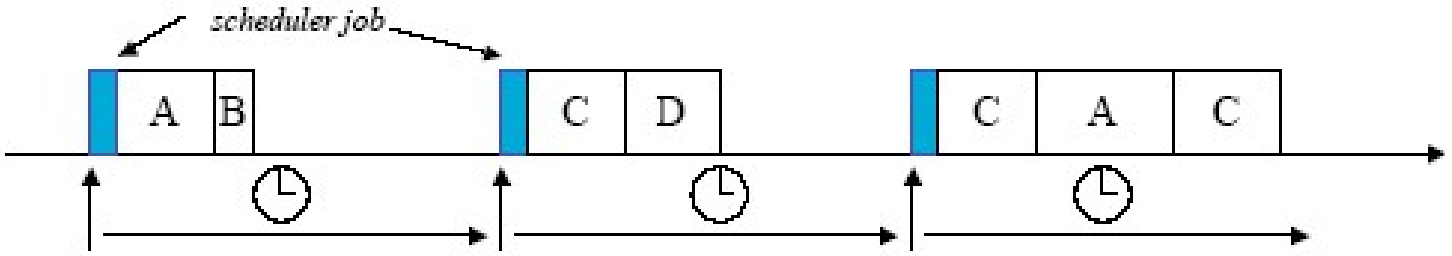
  il processore è comunque *impegnato* nell'esecuzione di un job se ci sono job da completare

- Scheduling *non work-conserving*:

  in qualche situazione il processore può essere mantenuto *libero* anche in presenza di job da completare

Approccio *clock-driven*

- *Istante di decisione di scheduling*: istante in cui lo scheduler decide quale job eseguire successivamente
- Negli scheduler clock-driven, gli istanti di decisione sono definiti *a priori*
- Ad esempio: lo scheduler viene risvegliato periodicamente e genera una porzione di schedule



Approccio *clock-driven*

- Quando i parametri dei job sono noti a priori, la schedule può essere pre-calcolata fuori linea e memorizzata in una tabella
  → schedulatori *table-driven*

- In alternativa: negli istanti predefiniti di decisione lo scheduler utilizza la priorità per decidere l'ordine dei job da mettere in esecuzione; lo scheduler resta clock-driven!

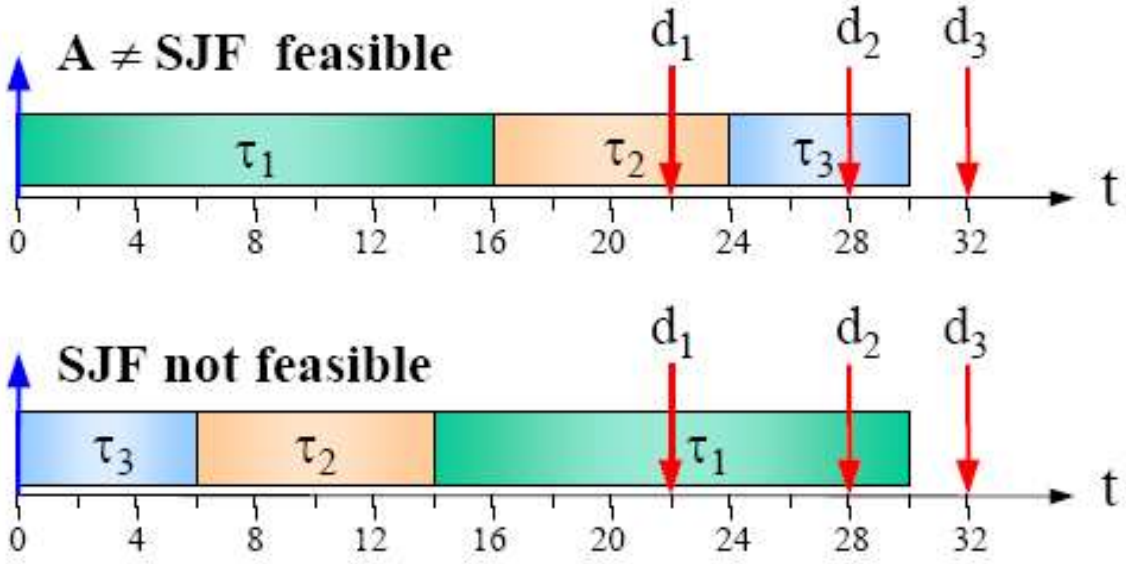
- La pianificazione statica della schedule o degli istanti di decisione può mantenere il processore *idle* anche in presenza di job pronti

Approccio *priority-driven*

- Criterio base: il processore non resta mai inattivo se c'è del lavoro da svolgere (schedulatori *work-conserving*)
- Il job viene scelto da una lista gestita con un *criterio di priorità*. Esempi: priorità statica, FIFO, LIFO, SET, LET, EDF
- La priorità attribuita a livello utente in generale non coincide con la priorità attribuita al job dal SO e su cui il SO basa le proprie decisioni di scheduling
- La priorità può influenzare le decisioni di scheduling in modo *preemptive* o *non-preemptive*

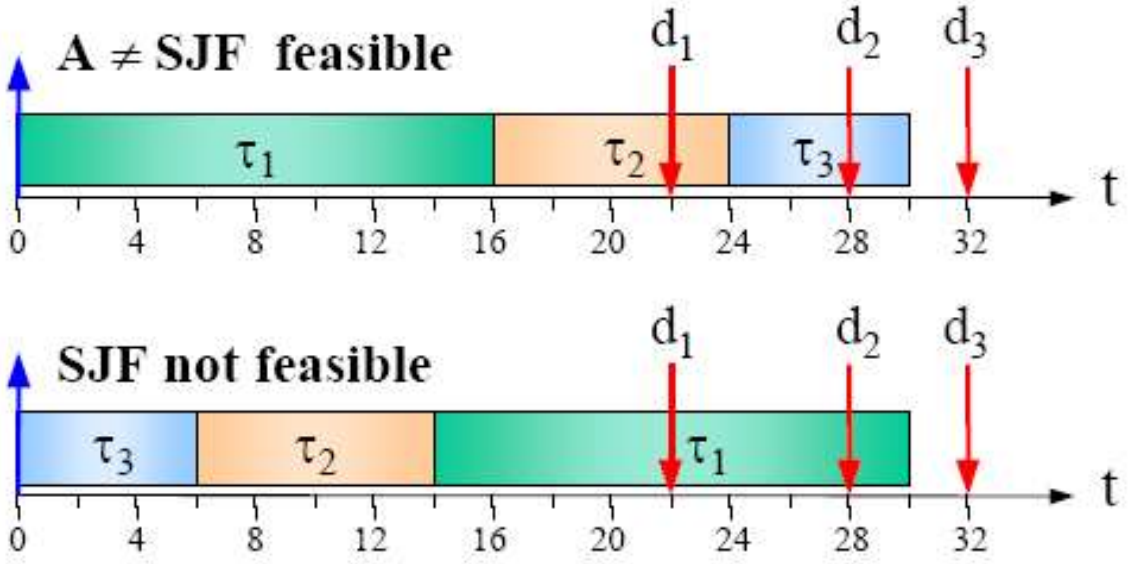
Approccio *priority-driven*

- Esempi di algoritmi di ordinamento delle priorità che considerano le caratteristiche dei job: LET, SET, EDF, LRT, LSF
- SJF (ovvero SET) è ottimo nello scheduling general-purpose ma non lo è nei sistemi RT perchè ignora le deadline:



Approccio *priority-driven*

- Esempi di algoritmi di ordinamento delle priorità che considerano le caratteristiche dei job: LET, SET, EDF, LRT, LSF
- SJF (ovvero SET) è ottimo nello scheduling general-purpose ma non lo è nei sistemi RT perchè ignora le deadline:



do urgent things first

do small things first

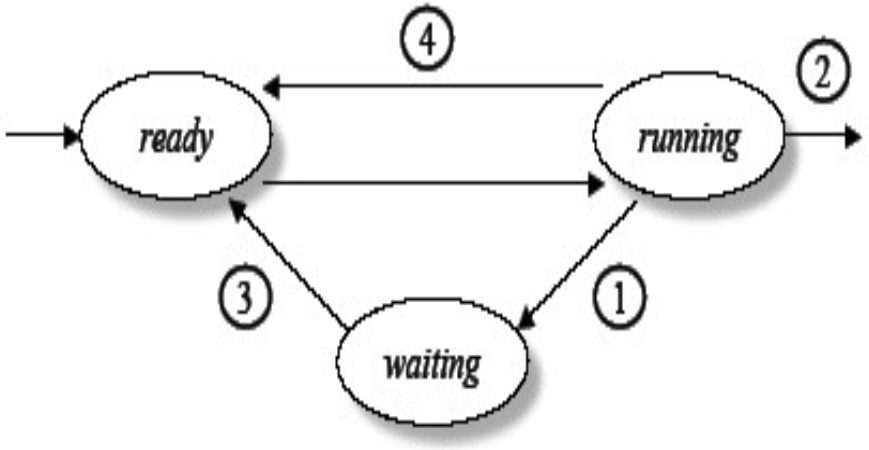
Approccio *priority-driven*

- Una possibile realizzazione di uno scheduling priority-driven di tipo *preemptive*:
  - Assegna le priorità *ai job*
  - Prendi le decisioni di scheduling quando:
    - un job diventa pronto
    - il processore diventa libero
    - le priorità dei job cambiano
  - In ciascun istante di decisione di scheduling, scegli il job a *priorità* più elevata

- La priorità dei job può essere statica o dinamica (es. slack time scheduling)

Approccio *priority-driven*

- Istanti di decisione di scheduling:

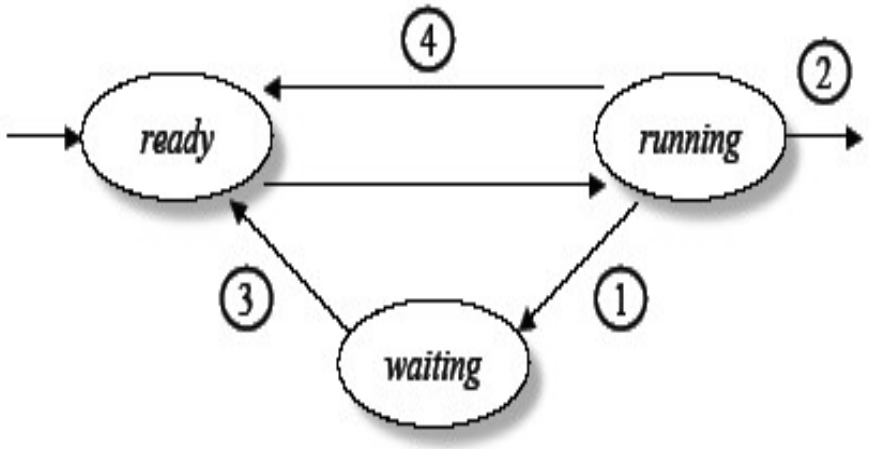


1. il job passa da esecuzione a waiting
2. il job in esecuzione termina
3. un job in attesa diviene pronto
4. il job in esecuzione torna nello stato pronto

- In genere le priorità dei job possono cambiare solo in corrispondenza alle transizioni di stato

Approccio *priority-driven*

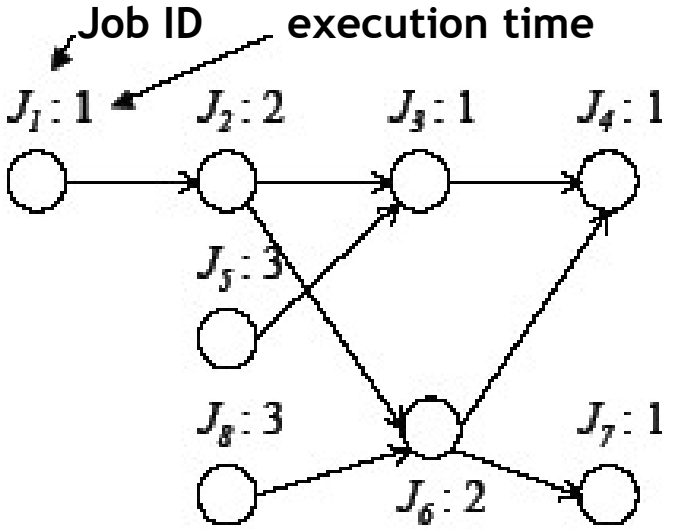
- In uno schedulatore priority-driven *non-preemptive*, le decisioni di scheduling sono prese solo quando:
  - il processore diventa libero (1), (2)
  - un job diventa pronto e il processore è libero (3 con processore libero)
  - Il job in esecuzione esegue una yield (4)



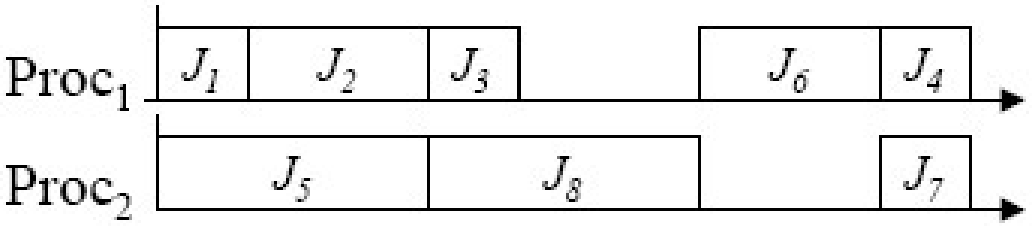
Elementi aggiuntivi del problema di scheduling

- Esistenza di vincoli di precedenza tra job
- Coerenza tra vincoli di precedenza e vincoli temporali
- Presenza o assenza di preemption
- Criterio di priorità in presenza di vincoli temporali, vincoli di precedenza, preemption o non preemption

Schedule basate su priorità in assenza di revoca



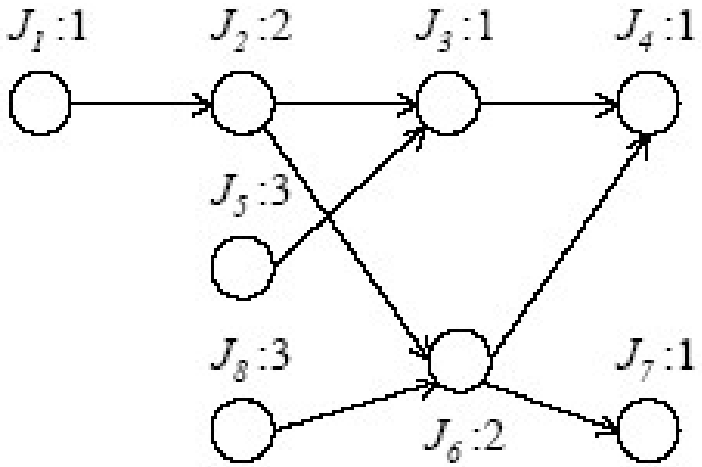
- Grafo con vincoli di precedenza
- Vincoli temporali: $r_i=0$, $d_i=t^*$ $\forall i$
- Schedule su 2 processori, con $Pri(J_i) > Pri(J_j)$ per $\forall i<j$:

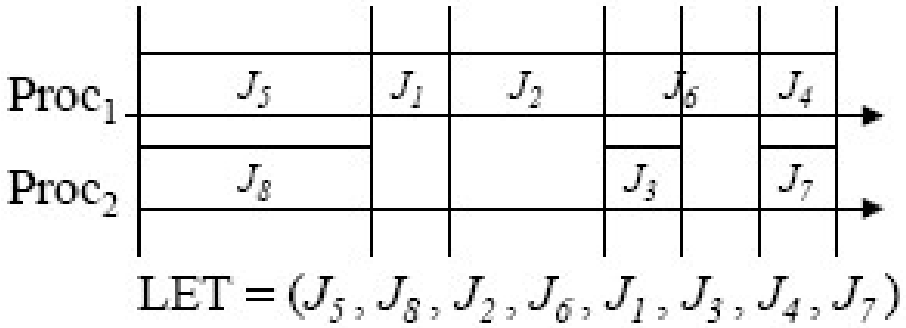


$t_c=9$

Schedule basate su priorità in assenza di revoca

- Schedule con algoritmo LET:

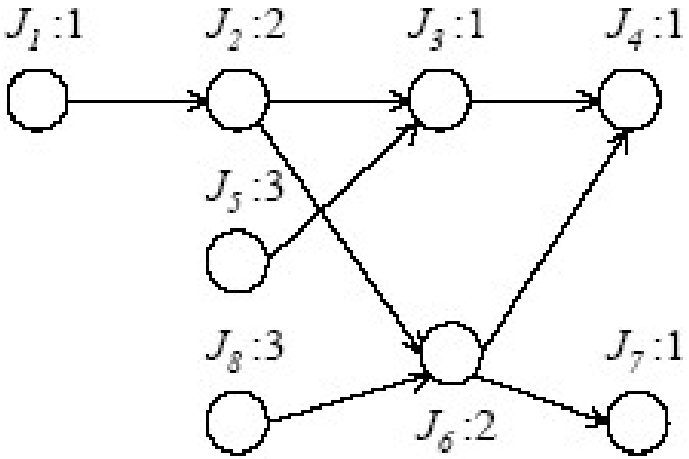


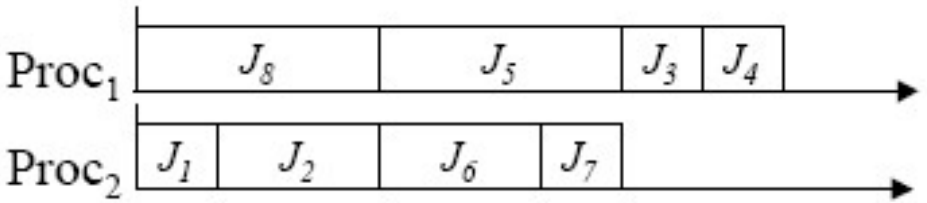


t_c=9

Schedule basate su priorità in assenza di revoca

- Schedule alternativa:

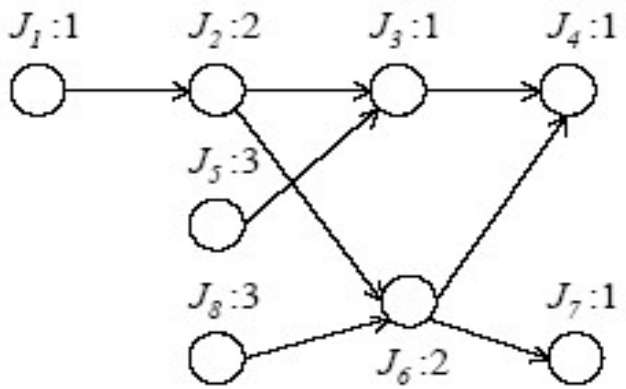


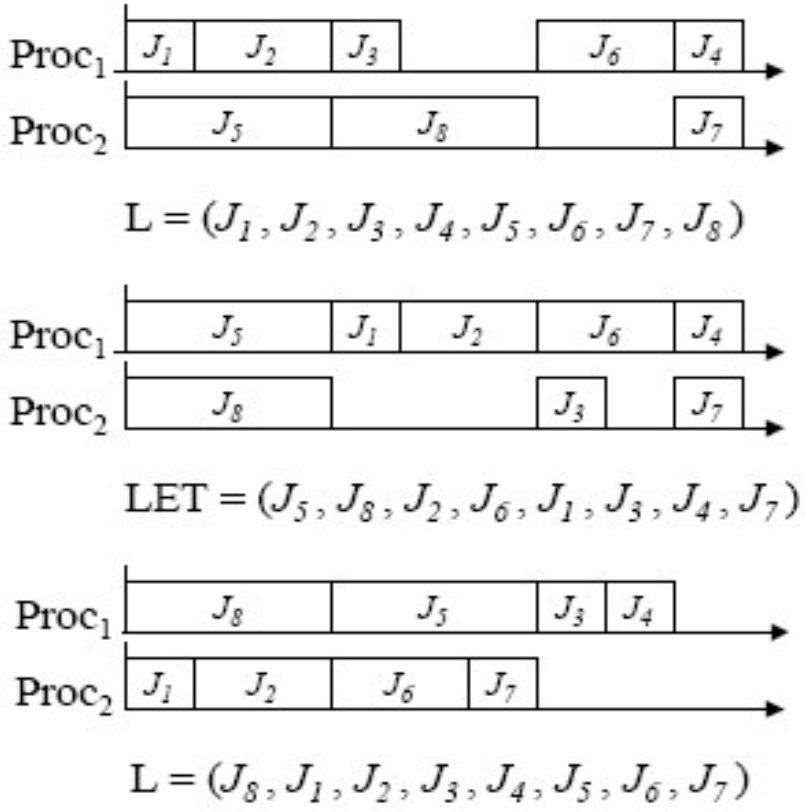


$$L = (J_8, J_1, J_2, J_3, J_4, J_5, J_6, J_7)$$

t_c=8

Schedule basate su priorità in assenza di revoca





- la priorità può essere scelta arbitrariamente ?
- determina schedule e tempi di completamento diversi

Vincoli temporali effettivi

- I vincoli temporali possono essere *non consistenti* con i vincoli di precedenza
  - Esempio: $d_1 > d_2$ mentre $J_1 \rightarrow J_2$
- Per il caso *uniprocessore* è possibile rimuovere i vincoli di precedenza considerando i *vincoli temporali effettivi*
- *Istante di rilascio effettivo*:

$$r_i^{eff} = \max \{r_i, \{ r_j^{eff} \mid J_j \rightarrow J_i \}\}$$

- *Deadline effettiva*:

$$d_i^{eff} = \min \{d_i, \{ d_j^{eff} \mid J_i \rightarrow J_j \}\}$$

Vincoli temporali effettivi

- Vincoli temporali effettivi = vincoli temporali consistenti con i vincoli di precedenza
- Job con predecessori: → istante di rilascio effettivo
- Job con successori: → deadline effettiva
- Il calcolo dei vincoli temporali effettivi ha costo $O(n^2)$

- Teorema

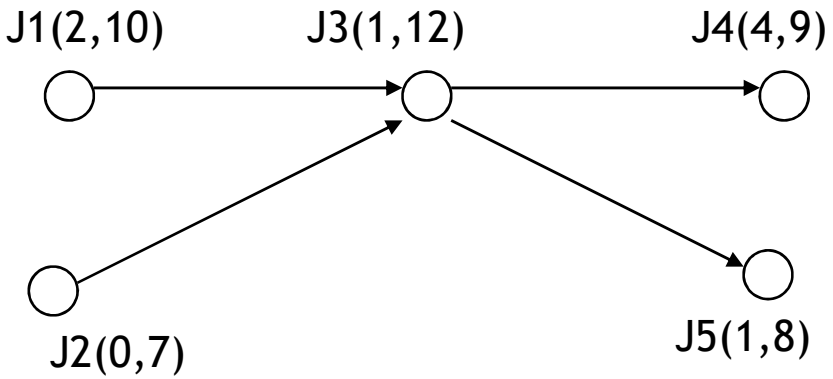
  Un insieme di job **J** è *schedulabile* su *un* processore se e solo se può essere schedulato in modo da rispettare i tempi di rilascio e le deadline *effettivi* dei job.

Vincoli temporali effettivi

- Il calcolo dei vincoli temporali effettivi prescinde dai tempi di esecuzione
- Il teorema tuttavia ci dice che esiste una schedule fattibile
- E' possibile che una schedule generata a partire dai vincoli temporali effettivi sia non corretta; tuttavia in tal caso è certamente possibile scambiare l'ordine di qualche job per produrne una corretta

Vincoli temporali effettivi

- Esempio - DAG con $J_i(r_i,d_i)$:



  - Dopo il calcolo dei vincoli effettivi:

    J1(2,8), J2(0,7), J3(2,8), J4(4,9), J5(2,8)

L'algoritmo EDF (Horn, 1974)

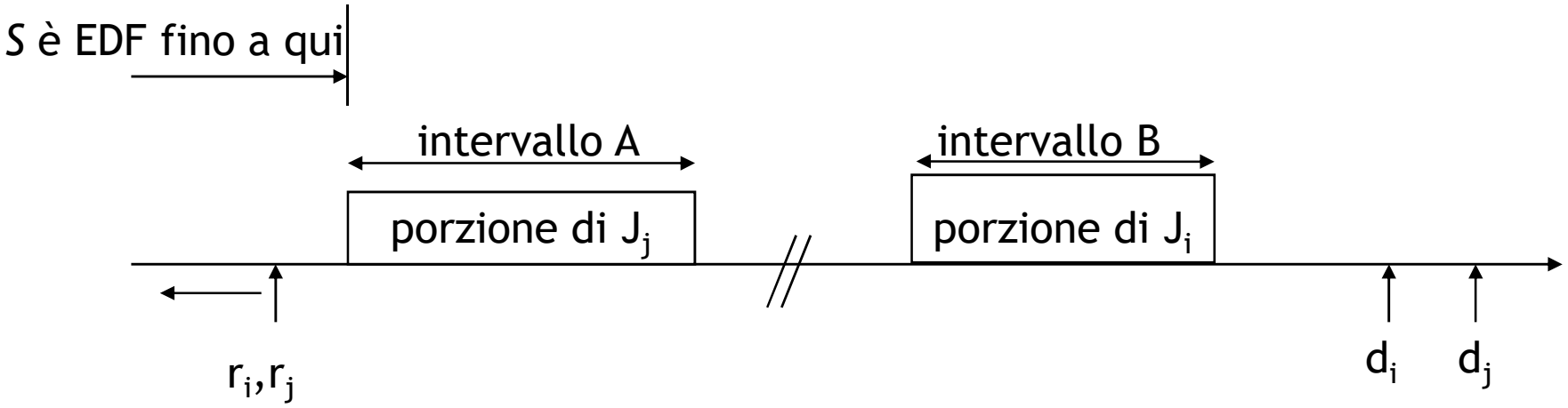
- *Algoritmo EDF (Earliest Deadline First)*:
  In ciascun istante, esegui il job *disponibile* (pronto o già in esecuzione) con la *deadline più prossima*

- Teorema *(Ottimalità di EDF*) (Dertouzos, 1974):
  In un sistema *monoprocessore con preemption*, EDF può generare una schedule fattibile per un insieme di job *J* con istanti di rilascio e deadline arbitrari se e solo se tale schedule esiste

- Dimostrazione: mediante trasformazione della schedule

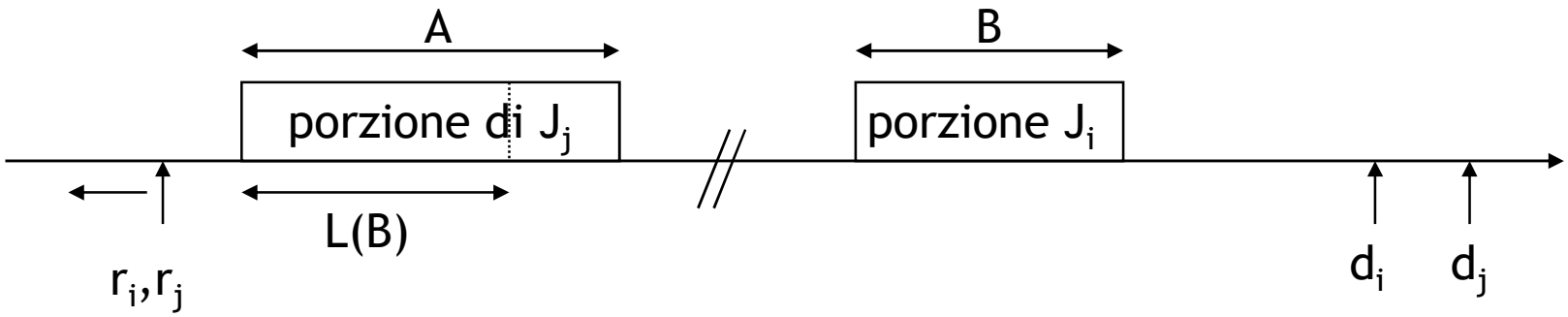
Ottimalità di EDF

- Si ipotizzi che un'arbitraria schedule *S* rispetti i vincoli temporali, mentre la schedule generata da EDF non rispetti i vincoli (ovvero: EDF non è ottimo)
- Se *S* non è una schedule EDF, si deve verificare la seguente situazione:



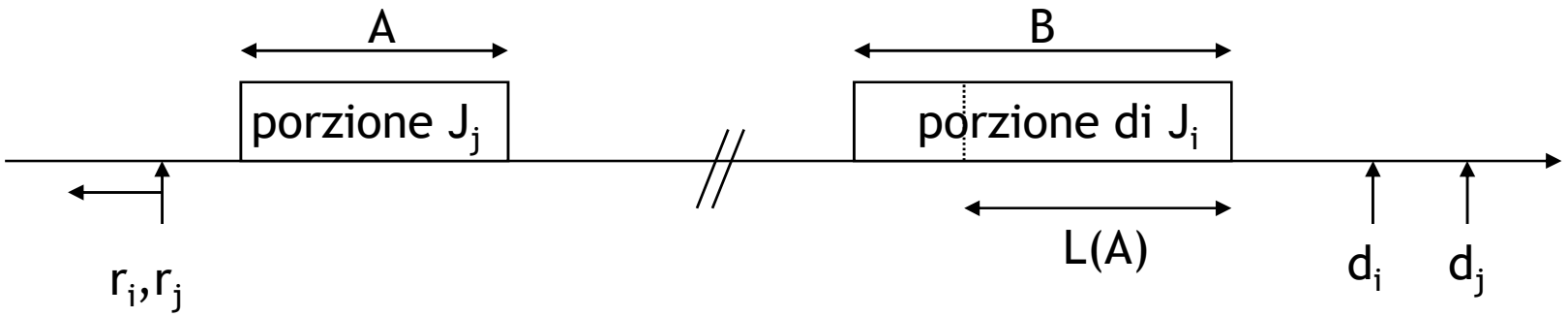
Ottimalità di EDF (2)

- Si possono avere due casi, e per entrambi è possibile una trasformazione che mantiene la fattibilità della schedule
- Caso 1: L(A) > L(B)
  - E' possibile anticipare l'esecuzione di J_i in A; J_j inizia in A e completa in B
  - Possibile perché J_j è revocabile



Ottimalità di EDF (3)

- Caso 2: L(A) ≤ L(B)
  - Ji inizia l'esecuzione in A e la completa nella prima parte di B
  - Possibile perché Ji è revocabile



- In entrambi i casi la schedule risulta riordinata in modo EDF ed è fattibile

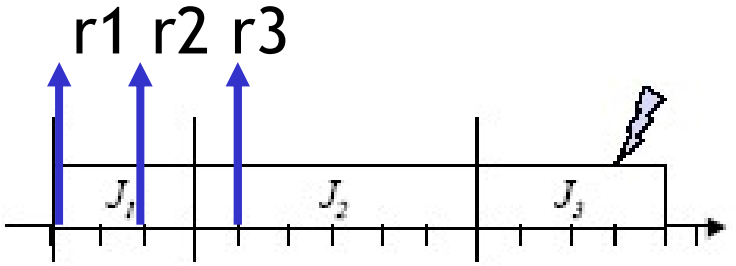
Ottimalità di EDF (4)

- Si ripete la trasformazione della schedule per tutte le coppie di job non schedulate in modo EDF
- Si eliminano eventuali intervalli *idle* (scheduling non work-conserving) anticipando l'esecuzione dei job pronti
- La schedule risultante è EDF...
- Il teorema segue per contraddizione □

EDF è ottimo solo nelle ipotesi date...

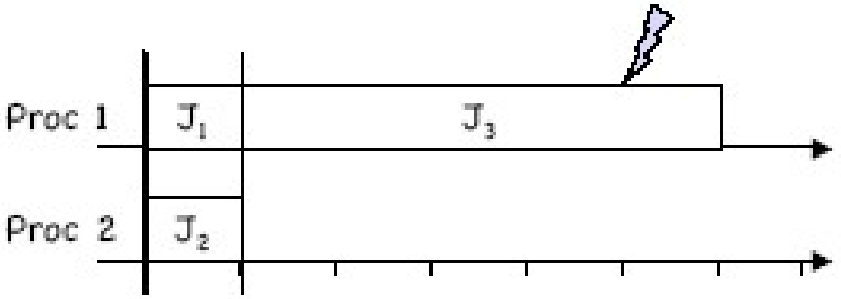
- Caso 1: Preemption non consentita:

		r_i	d_i	e_i
J_1	=	0,	10,	3
J_2	=	2,	14,	6
J_3	=	4,	12,	4



- Caso 2: Sistema multiprocessore:

		r_i	d_i	e_i
J_1	=	0,	4,	1
J_2	=	0,	4,	1
J_3	=	0,	5,	5

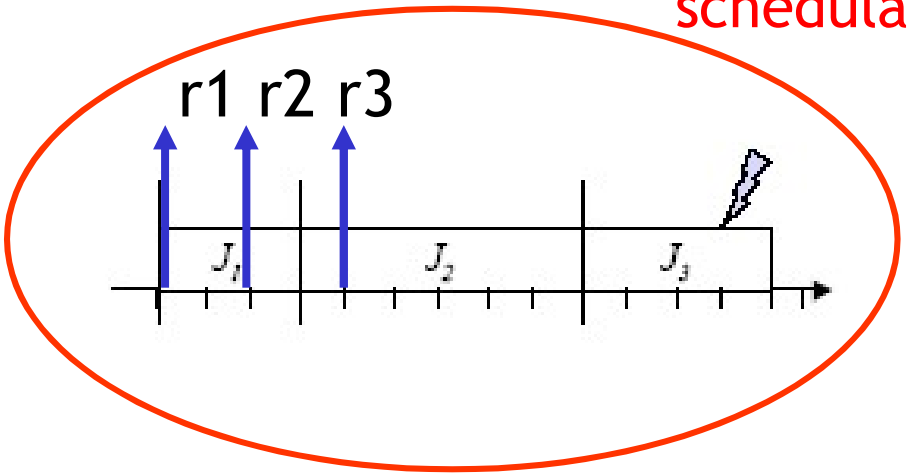


EDF è ottimo solo nelle ipotesi date...

- Caso 1: Preemption non consentita:

schedulabile?

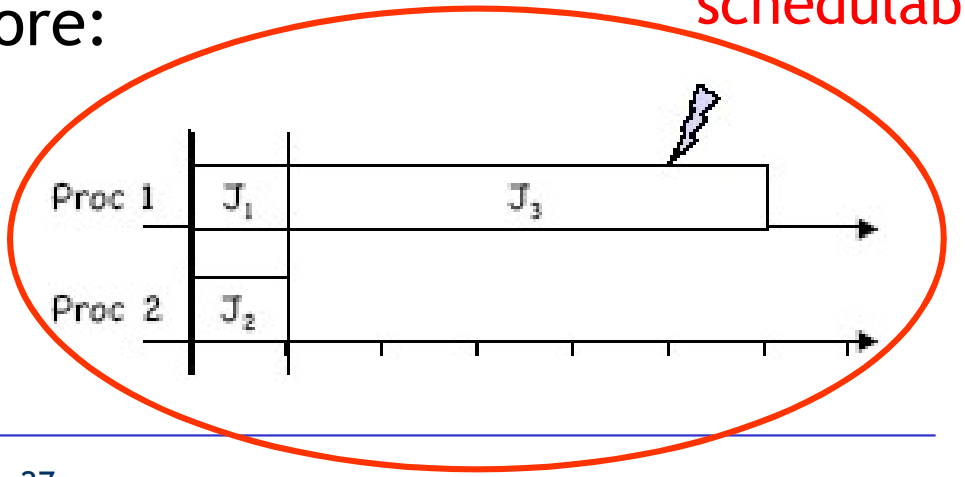
		r_i	d_i	e_i	
J_1	=	(0,	10,	3	)
J_2	=	(2,	14,	6	)
J_3	=	(4,	12,	4	)



- Caso 2: Sistema multiprocessore:

schedulabile?

		r_i	d_i	e_i	
J_1	=	(0,	4,	1	)
J_2	=	(0,	4,	1	)
J_3	=	(0,	5,	5	)



Non ottimalità di EDF

- Il caso 1 è schedulabile solo da un algoritmo *non* work-conserving
- Nessun algoritmo priority-driven è in grado di schedulare i job del caso 1
- → In assenza di preemption e con parametri temporali arbitrari, *nessun algoritmo priority-driven* (cioè work-conserving) *è ottimo*

Ottimalità e non ottimalità di EDF

- Q1: in un sistema uniprocessore, in assenza di preemption e con $r_i=0$ $\forall i$, EDF è ottimo?

- Q2: in assenza di preemption, con istanti di rilascio arbitrari ma multipli di un intervallo elementare, tempi di esecuzione unitari, EDF è ottimo? (Ad es. sistemi sincronizzati con un clock periodico)

Ottimalità e non ottimalità di EDF

- R1: Non c'è mai motivo di avere preemption: non arrivano nuovi job, e tutte le decisioni di scheduling sono prese quando il processore diventa libero -> EDF è ottimo
- L'algoritmo EDF con istanti di rilascio tutti uguali e senza preemption è stato introdotto nei problemi di *job shop scheduling* come *Earliest Due Date* (EDD) (Jackson, 1955)

- R2: Anche in questo caso la preemption non è necessaria

Altri criteri di ordinamento delle priorità

- *Latest Release Time (LRT)*: è l'algoritmo duale rispetto ad EDF

- Priorità: cresce all'aumentare dell'istante di rilascio
- Schedula i job a partire dall'ultima deadline ed inserisce via via i job con istante di rilascio maggiore

- Tende a lasciare il processore "idle" all'inizio dell'orizzonte temporale: utile per ridurre il tempo medio di risposta dei job soft o non RT
- E' un algoritmo *non work-conserving*

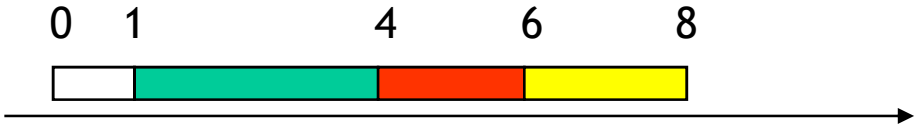
Latest Release Time

- Esempio:

	e_i	r_i	d_i
J1	3	0	6
J2	2	5	8
J3	2	2	7

  - L'ultima deadline è t=8. J2 può essere in esecuzione tra 7 ed 8
  - A t=7 J2 e J3 possono essere in esecuzione: J2 ha priorità perché r2>r3. J2 è quindi in esecuzione tra 6 e 7
  - A t=6 J1 e J3 possono essere in esecuzione: J3 ha priorità perché r3>r1 ed è schedulato tra 6 e 4
  - A t=4 J1 è schedulato tra 4 e 1

Schedule:



Ottimalità LRT

- Teorema (Ottimalità LRT):

  In un sistema *monoprocessore con preemption*, LRT può generare una schedule fattibile per un insieme di job *J* con istanti di rilascio e deadline arbitrari se e solo se tale schedule esiste.

- Segue dalla dimostrazione di ottimalità per EDF

Limiti della schedulazione LRT

- Attenzione: LRT può essere utilizzato solo se si dispone di *tutte le informazioni* sui job da schedulare per l'orizzonte temporale di interesse
- LRT non può operare dinamicamente perché è *non work conserving*
- Cosa accade, nell'esempio precedente, se all'istante 0 LRT conosce solo J1, l'unico rilasciato? Può attendere a metterlo in esecuzione fino all'istante 3 (d_1-e_1=3)? Le fasi di idle possono pregiudicare la schedulabilità successiva!

- work escaping? deferring? ...rischioso!

Least Slack Time First (LST) / Minimum Laxity First (MLF)

- In ciascun istante t vale: $slack_i = d_i - t - e_i(t)$
  in cui $e_i(t)$ è il tempo di esecuzione residuo di J_i
- L'algoritmo LST (o MLF) schedula in ciascun istante il job J_i con il minimo slack time, $slack_i$

- Teorema: Anche l'algoritmo LST/MLF è *ottimo* nelle medesime ipotesi di EDF e LRT

Priority-driven vs. clock-driven

- + Gli algoritmi priority-driven sono più *flessibili* di quelli clock-driven
- + La loro *realizzazione* (nel caso EDF) si basa su una semplice coda gestita in base alle priorità
- - Gli algoritmi priority-driven possono esibire un comportamento *non deterministico* se i parametri temporali variano in modo imprevisto
- - E' più difficile *validare* un insieme di job schedulati in modo priority-driven
- → alcune tipologie di sistemi hard real-time safety critical sono progettate secondo l'approccio clock-driven

Scheduling real-time di task aperiodici

- I vincoli sono tipicamente specificati in termini di makespan dell'insieme o di deadline dei singoli task

- L'algoritmo EDF è ottimo nell'ipotesi di task revocabili e di sistema monoprocessore, con istanti di rilascio e deadline arbitrari
- I vincoli di precedenza possono essere rimossi tramite il calcolo degli istanti di rilascio e delle deadline effettivi

- Per i casi di assenza di preemption o multiprocessore il problema di trovare una schedule fattibile è NP-hard. Gli algoritmi hanno complessità elevata e sono utilizzabili solo off-line

Algoritmo EDD (Jackson, 1955)

- Insieme di n task aperiodici $\{\tau_1,\ldots \tau_n\}$ da eseguire su un processore
- I task sono costituiti da un solo job e sono rilasciati in modo sincrono: $T=\{J_i(C_i, D_i), i=1, \ldots, n\}$
- Algoritmo *Earliest Due Date (EDD)*:

  "Quando il processore è libero seleziona il task con la deadline *relativa* minima"
- Caratteristiche:
  - Priorità statica (parametri D_i noti per il rilascio simultaneo dei task)
  - D_i coincide con d_i assumendo t=0
  - Non richiede preemption

Algoritmo EDD

- Teorema: Dato un insieme di n task indipendenti, ogni algoritmo che esegue i task *in ordine di deadline non decrescenti è ottimo* rispetto al criterio della minimizzazione della massima lateness, L_{max}
- Se $L_{max} \leq 0$ tutti i task rispettano la propria deadline
- Test di garanzia: $f_i \leq d_i \quad \forall i$

  Ordinando i task in base alle deadline D_i: $f_i = \sum_{k=1,i} C_k$

  Il test diviene: $\sum_{k=1,i} C_k \leq D_i \quad \forall i$

- Complessità: O(n log n) per l'ordinamento dei task, O(n) per garantire il task set

Algoritmo EDF (Horn, 1974)

- EDF estende l'algoritmo EDD considerando *istanti di arrivo arbitrari*
- Algoritmo *Earliest Deadline First (EDF)*:

  "In ogni istante esegui il job con la deadline *assoluta* più prossima"

- Caratteristiche:
  - Priorità dinamiche (dipendono dagli arrivi)
  - Preemption consentita ovunque
  - Algoritmo ottimo: minimizza la massima lateness L_{max}
- Complessità: O(n) per inserire un nuovo task pronto in coda, O(n) per garantire un nuovo task -- online

Algoritmo EDF

- Test di garanzia: $f_i \leq d_i \quad \forall i$
- Ipotesi: job ordinati in base alle deadline d_i

  pertanto: $f_i = t + \sum_{k=1,i} C_k(t)$ ove $C_k(t)$ è il WCET residuo

  da cui: $\sum_{k=1,i} C_k(t) \leq d_i - t \quad \forall i$

- Verifica della garanzia *online*: al rilascio di un nuovo job J_i occorre verificare se esso è garantito e se tutti i job J_j accettati in precedenza (con $j>i$, cioè $d_j>d_i$) restano garantiti

Esercizio

- Dimostrare la ottimalità/non ottimalità dell'algoritmo priority driven LET (longest execution time first) in un sistema monoprocessore con preemption ovunque consentita

Esercizio

- Dato il grafo di precedenza di un insieme di task aperiodici Ji(ri,di) con tempo di esecuzione unitario per ogni task:
  - Calcolare i vincoli temporali effettivi
  - Determinare una schedule con algoritmo EDF
  - Determinare una schedule con algoritmo LRT