

# SERT - 09/10/2020 - Ottimalità di algoritmi priority-driven - R05


## Di cosa parliamo in questa lezione?

In questa lezione si discute l'ottimalità o meno degli algoritmi di schedulazione priority-driven

- 1 Il problema della validazione
- 2 Il fattore di utilizzazione
- 3 Il test di schedulabilità

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.2

Oggi continuiamo a parlare dell'ottimalità degli algoritmi priority-driven. Abbiamo già introdotto il concetto di ottimalità, e continuiamo ad esplorare questo concetto, in particolare al problema della validazione, cioè a decidere con una dimostrazione formale, se effettivamente in un certo sistema progettato in una certa maniera con un certo algoritmo di schedulazione, le scadenze saranno sempre rispettato oppure no.

## Il problema della validazione


Gli algoritmi **priority-driven** in generale:

- sono semplici da implementare
- sono flessibili
- non richiedono necessariamente di conoscere esattamente il modello di carico
- è non banale dimostrare formalmente che i vincoli temporali dei job hard real-time saranno sempre rispettati, soprattutto se i parametri temporali non sono ben precisati

### Il problema della validazione

Dati un insieme di job, processori e risorse utilizzabili dai job, e l'algoritmo di schedulazione e accesso alle risorse, determinare se tutti i job rispetteranno i vincoli temporali

Ottimalità di algoritmi priority-driven  
Marco Cesati



Schema della lezione  
Validazione  
Fattore di utilizzazione  
Test di schedulabilità

SERT'20 R5.3

In generale gli algoritmi priority-driven sono semplici da implementare. Sono facili da spiegare. Abbiamo visto come **EDF** è facile da spiegare, in quanto basta dire che la priorità è inversamente proporzionale alla scadenza assoluta. Quanto più vicina a noi è la scadenza assoluta di un job, tanto più deve avere priorità rispetto a job che hanno una scadenza più grande. Più vicina è la scadenza più urgente è completare il job.

Tutto sommato sono semplici da implementare e sono flessibili, in quanto il concetto di priorità può essere molto generale. Possiamo assegnare le priorità in modo molto diverso, ed abbiamo come risultato degli algoritmi molto differenti. **EDF** ad esempio assegna le priorità alla scadenza assoluta del job, che è molto differente da **Rate Monotonic** che assegna le priorità in base alla scadenza relativa. Abbiamo visto che sono due algoritmi completamente differenti, anche come capacità di schedulare gli insiemi di task in modo da rispettare le scadenze. Il problema è che questo tipo di algoritmi, non richiedono necessariamente di conoscere esattamente il modello di carico. Questa è una differenza sostanziale rispetto all'altra classe di algoritmi che abbiamo visto, quelli cyclic-executive / clock driven, in cui per poter costruire la schedulazione devo conoscere per filo e per segno come è fatto il carico del sistema: quali sono i task, quanti sono, che frequenza hanno, quale è il tempo di esecuzione dei job e così via. Per non parlare dei vincoli di dipendenza ecc.

Se non conosco tutto, non posso costruire una schedulazione. La schedulazione a quel punto la posso fare offline, la fa il progettista, e dopo lo scheduler si

limita solamente ad applicarla come una partitura musicale. Però ovviamente questo tipo di algoritmi sono poco flessibili, mentre gli algoritmi *priority-driven*, sono molto flessibili perché il progettista non deve sapere quali sono i task che interverranno nel sistema. Possono esserci job rilasciati, task creati, e l'unica cosa in qualche modo lo scheduler deve sapere è come calcolare le priorità e dunque le relazioni di priorità tra i job. Ma non che il progettista del sistema deve conoscere in anticipo l'intero sistema di task. Quindi effettivamente sono molto più semplici anche da progettare e da implementare.

Mentre con gli algoritmi clock-driven la schedulazione era fatta online, dunque convincere qualcuno che tutti rispettano le scadenze è facile, con questo tipo di algoritmi invece è molto più difficile. Non è banale riuscire a dimostrare che l'algoritmo riuscirà sempre a rispettare le scadenze, dato che in astratto non è detto che il progettista conosca in anticipo l'intera configurazione di task che si presenterà nel sistema. Il problema dunque è tutto qua, è quello di, dato un insieme di job, processori e risorse utilizzabili da questi job, *capire se l'algoritmo che sto utilizzando per schedulare i job e per accedere alle risorse, se saranno in grado di fare in modo che tutti i job rispetteranno sempre le scadenze*. Per questo problema dovrò dimostrare matematicamente che le cose stanno così, dovrò dare una dimostrazione certa. Questo è il problema della validazione.

### Validazione di algoritmi priority-driven

Per gli algoritmi *priority-driven* il problema della validazione è difficile da risolvere a causa delle *anomalie di schedulazione* (o *Richard's anomalies*, Graham 1976)

Sono comportamenti temporali inattesi che si verificano anche in sistemi semplici

Ad esempio, in un sistema con job *non interrompibili* il tempo di risposta dell'ultimo job che termina (*makespan*) può peggiorare se:

- Si aumenta il periodo (diminuisce la frequenza) di un job
- Si riduce il tempo di esecuzione di un job
- Si riducono le dipendenze tra i job
- Si aumenta la velocità del processore (Buttazzo 2006)


*Perché le anomalie complicano il problema della validazione?*

Se i parametri dei job di un sistema possono variare, non si può validare il sistema esaminando solo il "caso peggiore": è necessario esaminare tutte le combinazioni di parametri

Questo problema è complicato da un fenomeno paradossale, chiamato **anomalie di schedulazione**. Ci si è resi conto molto presto, già dal '76 in poi, quasi fin dalla nascita dei sistemi RT, che c'erano degli aspetti di questa teoria dei

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20 R5.4

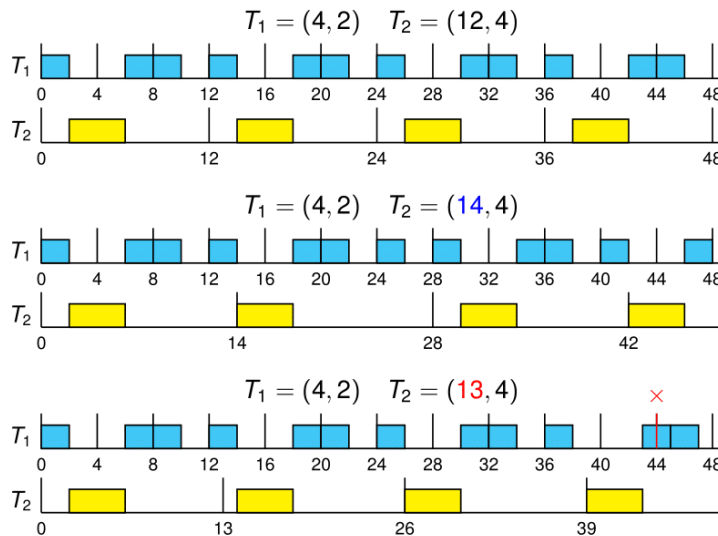
sistemi RT che era paradossale. Si avevano risultati che erano contro il senso intuitivo, contro il senso comune. Queste **anomalie di schedulazione** sono dei comportamenti che non ci attendiamo. Per esempio supponiamo che io abbia dimostrato che in un certo sistema tutte le scadenze sono rispettate, con certi parametri temporali (rilasci, scadenze e tempi di esecuzione). Poi supponiamo che per qualche job, il tempo di esecuzione si riduca, oppure che un certo task rilasci i job con una frequenza più bassa di quella con cui ho fatto la supposizione, quindi con un carico del processore più basso. Il risultato inatteso è che in queste condizioni apparentemente più favorevoli, invece di continuare ad avere il rispetto delle scadenze, queste vengono mancate.

Un esempio classico di questo fenomeno è un sistema in cui i job sono non interrompibili. Questa non interrompibilità dei job abbiamo già visto essere un problema, e qui ritorna. Per esempio in un sistema con job non interrompibili, il tempo di risposta dell'ultimo job che termina, il cosiddetto *next span*, può peggiorare se per esempio aumenta il periodo, dunque diminuisce la frequenza di un job: occupa meno il processore, ma il tempo con cui conclude l'ultimo job peggiora. Oppure un altro job riduce il suo tempo di esecuzione. Come può essere che in realtà l'ultimo job concluda più tardi rispetto a prima? Eppure è così, un comportamento inatteso ma possibile.

Oppure invece di aumentare, riduco le dipendenze che ci sono tra i job. Anche questo può portare ad un aumento dei tempi di completamento dell'ultimo job. O addirittura se aumenta la velocità del processore, daccapo, il tempo di risposta dell'ultimo job può peggiorare. Tutti fenomeni francamente inattesi, poco poco intuitivi. **Perché le anomalie complicano il problema della validazione?** La risposta è molto semplicemente che non abbiamo più un caso peggiore da esaminare. Se i parametri dei job possono variare, io in qualche modo non so dire qual'è il parametro che dà il valore peggiore per il rispetto delle scadenze, allora dovrò validare, dovrò dimostrare che il mio sistema è schedulabile per qualunque variazione del valore dei miei parametri. Questo ovviamente diventa impossibile, perché ci sono infinite variazioni dei valori dei parametri. Se ad esempio i miei job possono variare il tempo di esecuzione, perché ad esempio quello che indico è un worst case, ma poi invece il valore effettivo di esecuzione di un job può essere inferiore, se io valido il caso peggiore ma poi quel risultato non mi è valido quando il job ci mette di meno, è ovvio che quella validazione fatta nel caso peggiore non serve più a nulla. Il caso peggiore non esiste. Non è il caso peggiore per la validazione. Le anomalie di schedulazione hanno questo di problema. Ci impediscono di capire quale sia il caso peggiore, quindi ci impediscono di fare una validazione di quell'unico caso peggiore. Ci costringono quindi a fare infinite dimostrazioni, e questo ovviamente diventa impossibile.

## Esempio di anomalia di schedulazione (periodo)

Due task **non** interrompibili schedulati con **RM** su 1 processore



Vediamo qualche esempio di anomalia di schedulazione. Questo è un esempio relativo al periodo. Consideriamo due task, sempre non interrompibili, schedulati con **RM** su un processore. Abbiamo un task con periodo 4 e tempo di esecuzione 2, ed un task con periodo 12 e tempo di esecuzione 4. Qui troviamo, nel primo grafico, la schedulazione **RM**. Va tutto bene, tutti rispettano le scadenze. Adesso supponiamo di alzare il periodo del secondo task, facendolo passare da 12 a 14. Possiamo rifare la schedulazione **RM**, ed ancora una volta ci rendiamo conto che effettivamente tutti i task rispettano le scadenze. In realtà, però, se considero un periodo per il secondo task 13, allora la schedulazione **RM** fallisce. Cioè la schedulazione **RM** non riesce a far rispettare le scadenze di tutti i job. Perché questo succede?

Succede perché i job non si possono interrompere. Con un periodo 12 le cose si combinano in una maniera tale per cui tutto funziona. Per un periodo 14 c'è abbastanza tempo per i task di  $T_1$  di entrare sui task di  $T_2$  in modo che tutti rispettino le scadenze. Ma quando il periodo non è né 12 né 14, ma è 13, il job di  $T_2$  entra in un momento particolarmente sfortunato. Siccome non si può interrompere, quando arriva il task di  $T_1$  accumula un ritardo tale per cui non può completare le scadenze. Questo comportamento è inatteso, perché alzare un periodo da 12 a 13, vuol dire abbassare l'occupazione di questo task sul processore. Il processore ha meno da fare, ma nel primo caso rispetta le scadenze della schedulazione, nel secondo caso no. **Questa è propria un'anomalia.**

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

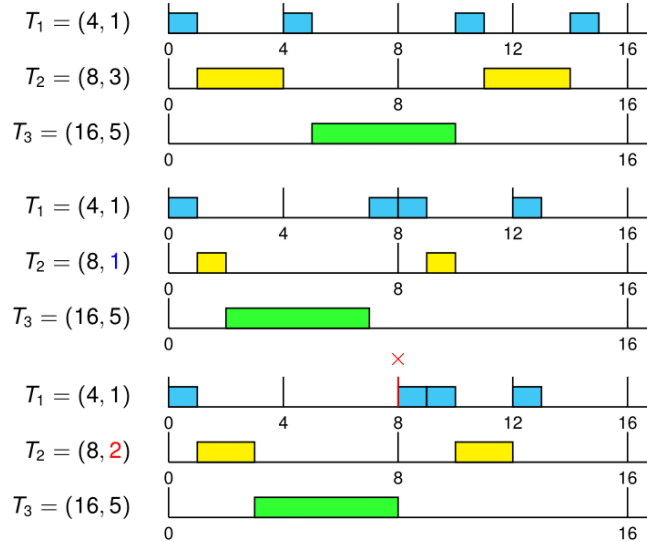
Test di schedulabilità

SERT'20

R5.5

## Esempio di anomalia di schedulazione (tempo d'esecuzione)

Tre task **non** interrompibili schedati con **RM** su 1 processore



Vediamo un altro esempio, legato non più al periodo, ma al tempo di esecuzione. Abbiamo tre task non interrompibili, schedati con **RM** su un solo processore. Tutti rispettano le scadenze. Adesso consideriamo un sistema di task in cui abbassiamo il tempo di esecuzione dei job di  $T_2$  da 3 ad 1. Dunque i job di  $T_2$  durano di meno. Ma anche in questo caso, tutti i job rispettano le scadenze. Dopo di che, consideriamo il caso in cui i job di  $T_2$  non hanno tempo di esecuzione 1, ne tre, ma tempo di esecuzione 2. In questo caso particolare, il secondo job di  $T_1$  nell'iperperiodo manca la sua scadenza.

Quale è la differenza tra questi casi? E' abbastanza evidente che nel primo caso il job di  $T_2$  è abbastanza lungo da arrivare al momento in cui il job di  $T_1$  viene rilasciato nuovamente, e quindi impedisce al job di  $T_3$  che è lungo ed ingombrante, di eseguire. In questo caso quindi, il secondo job di  $T_1$  ha il tempo di andare prima e completare. Questo è un comportamento inatteso. Abbassare i tempi di esecuzione di un job, vuol dire che il processore ha meno da fare. Quando ha il processore ha più da fare rispetto le scadenze, ma quando ha meno da fare vengono mancate. Quale è la causa di queste anomalie? Sembra essere la non interrompibilità dei job.

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

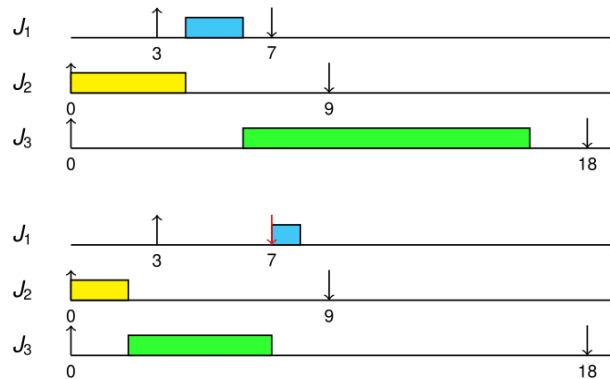
Test di schedabilità

SERT'20

R5.6

### Esempio di anomalia di schedulazione (velocità processore)

Tre job **non** interrompibili schedulati con EDF su un processore



Una anomalia analoga si verifica con due job interrompibili che accedono ad una risorsa condivisa (Buttazzo 2006)

Questa cosa la possiamo vedere anche con questo altro esempio, in cui variamo la velocità con cui il processore esegue i task. Consideriamo per esempio tre job, eseguiti tanto per cambiare tramite schedulazione **EDF** su un singolo processore. Nel momento in cui però il processore è più veloce, cosa vuol dire? Vuol dire fondamentalmente che completa i job più rapidamente. Se ad esempio, grossomodo, raddoppiamo la velocità del processore, quindi tutti questi tempi dei job si dimezzano, allora che succede? Che il job  $J_2$  completerà prima del rilascio del job di  $J_1$ . I tempi sono tempi, quelli non si riducono, e quindi entra in esecuzione il job  $J_3$ , non interrompibile, che non lascia spazio al job di  $J_1$  per completare. Un'analogia anomalia significa anche per esempio quando job sono interrompibili, ma accedono ad una risorsa condivisa. Non è soltanto la non interrompibilità dei job a creare le anomalie di schedulazione, ma ci sono anche altri casi. Tutto questo ci complica parecchio la vita.

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.7

## Esecuzione predicibile

Fissato un algoritmo, la schedulazione prodotta considerando i tempi d'esecuzione massimi (minimi) per tutti i job è detta *schedulazione massima (minima)*

L'esecuzione di un job è *predicibile* se è sempre entro i limiti temporali stabiliti dalle schedulazioni minima e massima

- siano  $\sigma^-$  e  $\epsilon^-$  gli istanti di attivazione e completamento di un job nella schedulazione minima
- siano  $\sigma^+$  e  $\epsilon^+$  gli istanti di attivazione e completamento dello stesso job nella schedulazione massima
- il job ha esecuzione *predicibile* se l'istante di attivazione è sempre in  $[\sigma^-, \sigma^+]$  e l'istante di completamento è sempre in  $[\epsilon^-, \epsilon^+]$

Fissato un algoritmo, un insieme di job è *predicibile* se lo è l'esecuzione di ciascuno dei suoi job

Però possiamo cercare di ricondurci al caso in cui i job sono effettivamente predicibili. Come? Fissato un algoritmo, consideriamo la schedulazione prodotta considerando tutti i tempi di esecuzione massimi (*worst case*). Questa schedulazione la chiamiamo **schedulazione massima**. Poi ci poniamo il caso in cui tutti i job hanno il tempo di esecuzione più piccolo possibile. Questa la chiamiamo **schedulazione minima**. Diciamo che un job, l'esecuzione di un job, è predicibile se cade sempre entro i limiti temporali stabiliti dalle schedulazioni minima e massima. Quindi, per spiegarci, consideriamo quando un job viene attivato e quando completa nella schedulazione minima,  $\sigma^-$  e  $\epsilon^-$ . Consideriamo anche  $\sigma^+$  e  $\epsilon^+$  per la schedulazione massima. Noi diciamo che l'esecuzione è predicibile per quel job se l'istante di attivazione cade sempre tra  $[\sigma^-, \sigma^+]$  e l'istante di completamento cade sempre tra  $[\epsilon^-, \epsilon^+]$ . Se un job ha un'esecuzione predicibile è predicibile. Se tutti i job di un certo insieme sono predicibili, allora tutto il sistema di task predicibile. Quindi diciamo che l'esecuzione è predicibile. La predicibilità dipende anche dall'algoritmo di schedulazione.

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.8



## Predicibilità per gli algoritmi priority-driven

### Teorema (Ha & Liu, 1993)

Un insieme di job interrompibili, indipendenti, e con istanti di rilascio fissati schedulato su un processore da un algoritmo priority-driven è **predicibile**

*Qual è il vantaggio di lavorare con insiemi predicibili?*

Il processo di validazione è facile perché possiamo verificare solo il caso della **schedulazione massima**

*Come applicare il teorema a sistemi con più processori?*

Legando l'esecuzione di ciascun job ad un singolo processore (sistema **statico**)

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.9

Questo perché è importante? Perché fondamentalmente noi possiamo trovare, daccapo, il caso peggiore, per lo meno per quanto riguarda i tempi di esecuzione. Possiamo dire: *se il tempo di completamento cade sempre entro i limiti della schedulazione minima o massima, vuol dire che il caso peggiore è la schedulazione massima. Oltre quello un job non potrà andare.* Quindi la schedulazione massima torna ad essere il caso peggiore da analizzare per la schedulazione.

Qui il punto è che il grosso problema è rispettare le scadenze, non si deve andare oltre. Ci possono anche essere delle scadenze alla rovescia, cioè posso anche avere dei vincoli che dicono che non posso completare prima di un certo tempo. Ma abbiamo visto che questi casi sono facili da gestire, perché se un job cerca di completare prima della scadenza lo posso artificialmente allungare. Il punto è che la grande difficoltà è fare in modo che i job non completino oltre la scadenza, in quanto non è un problema del job in se ma un problema comune. Siamo riusciti a dare la CPU a tutti in modo che tutti rispettino le scadenze? Questo è il grosso problema. Devo completare entro il tempo limite.

Quindi in questo senso se il sistema è predicibile, vuol dire che c'è il caso peggiore. Il caso in cui tutti i job hanno il *worst case*, cioè hanno il tempo di esecuzione peggiore possibile. Se io riesco a validare il sistema in quel caso peggiore, varrà sempre, cioè in ogni caso, anche quando i job potranno avere un tempo di esecuzione più piccolo.

Il teorema del '93 dice che un insieme di job che sono interrompibili, quindi possono essere interrotti in esecuzione e sostituiti con un altro job di priorità

superiore, indipendenti, dunque non hanno vincoli di precedenza tra i job, e con istanti di rilascio fissati, quindi in qualche modo non che arrivano ad istanti arbitrari, che è schedulato su un processore singolo con un algoritmo *priority-driven* è predicibile.

Quindi un singolo processore, job interrompibili, indipendenti e con istanti di rilascio fissati, è un sistema predicibile, c'è un caso peggiore. Quindi il vantaggio di lavorare con i sistemi predicibili è che possiamo validare il sistema, possiamo fare una certificazione. Possiamo dimostrare che tutti rispettano le scadenze, perché lo dimostro nel caso peggiore che è la schedulazione massima. Che succede se queste condizioni non valgono? I problemi cominciano a presentarsi, e vedremo come modellarli ed affrontarli. Il primo problema in effetti è cosa succede se il sistema ha più processori. Potremmo cercare di ridurre al problema con un singolo processore. Per esempio posso partizionare l'insieme di job ed assegnare ogni job ad un processore. A quel punto ci sono tanti processori nel sistema, ma ho suddiviso i job ai vari processori e fondamentalmente mi riduco al caso in cui c'è solo un processore e su quello ragiono. Si parla di **sistema statico**. In realtà il discorso che stiamo facendo è molto semplificativo e verrà ripreso più avanti.

#### Fattore di utilizzazione

- Algoritmi come **FIFO** e **LIFO** non considerano l'urgenza dei job: nei sistemi real-time hanno prestazioni pessime
- Algoritmi **fixed-priority** con priorità associate alla importanza relativa dei task hanno prestazioni cattive
- Gli algoritmi migliori sono quelli che assegnano la priorità in base a parametri temporali


Come valutare le prestazioni degli algoritmi di schedulazione basati su parametri temporali?

Fissato un algoritmo di schedulazione  $X$ , il suo **fattore di utilizzazione** (o *schedulable utilization*) è un valore  $U_X \in [0, 1]$  tale che l'algoritmo può determinare una schedulazione fattibile per qualunque insieme di task periodici su un processore se l'utilizzazione totale dei task è minore o uguale ad  $U_X$

Tanto maggiore è  $U_X$ , tanto migliore è l'algoritmo

Cerchiamo adesso, avendo chiarito quale è il problema della validazione e del perché è necessario avere un caso peggiore per poter validare il sistema, cerchiamo di capire come si fa a certificare che un sistema rispetterà le scadenze. Il concetto che ci serve è quello di **fattore di utilizzazione**, che avevamo già introdotto nelle prime lezioni quando parlavamo come erano fatti questi task e job, parlavamo di

**Ottimalità di algoritmi priority-driven**  
Marco Cesati



Schema della lezione  
Validazione  
**Fattore di utilizzazione**  
Test di schedulabilità

SERT'20 R5.10

questo parametro. Per un modello a task periodici, *l'utilizzazione di un task* era il rapporto fra il tempo di esecuzione ed il periodo con cui questo task rilasciava il job. Quindi in qualche modo era quanto pesava sul processore. Ora possiamo cercare di utilizzare questo concetto di utilizzazione per fare un confronto fra alcuni algoritmi di schedulazione. Quale è l'idea? Partiamo dall'idea che ci sono degli algoritmi che vanno molto male per i sistemi RT. Per esempio, algoritmi *LIFO* o *FIFO* vanno male, in quanto non considerano l'urgenza dei job. La priorità di questi algoritmi non è legata al concetto di scadenza, né in modo implicito e né esplicito. Quindi non abbiamo nessuna garanzia.

Ci sono poi una classe di algoritmi in cui il concetto di priorità lo posso legare in modo arbitrario, a gusto del progettista. Priorità associate all'importanza relativa dei task hanno prestazioni cattive. Che vuol dire? Vuol dire che non c'è un modo oggettivo di quantificare la bontà di questi algoritmi. In altri termini, finché le condizioni che aveva pensato il progettista valgono, in realtà gli algoritmi funzionano. Ma non si può escludere a priori che ci sia un insieme di task per i quali, fatti in una certa maniera, quell'algoritmo funzioni molto male e le scadenze vengano mancate. In realtà la teoria della schedulazione RT comincia nel momento in cui ci si rende conto che gli algoritmi per i sistemi RT devono essere basati unicamente su **parametri temporali oggettivi**: il periodo, la scadenza, la scadenza assoluta, lo slack rimanente. Ma devono essere parametri temporali *oggettivi*, cioè che non si possono mettere in discussione. Nel momento in cui questo è stabilito allora possiamo fare un discorso formale, matematico, su questi algoritmi.

Come valutare le prestazioni degli algoritmi di schedulazione che si basano su parametri temporali? Noi vogliamo che gli algoritmi *FIFO* e *LIFO* abbiano prestazioni pessime, e che sia il contrario per **EDF**. Abbiamo detto che **EDF** è ottimale. Vuol dire che se riesco a schedulare il sistema, allora **EDF** ci riesce. Allora vogliamo che questo algoritmo sia classificato come molto buono. Quello che possiamo fare è il concetto di **fattore di utilizzazione** dell'algoritmo di schedulazione, da non confondere con l'utilizzazione del task.

Si tratta di un valore  $U_X$  compreso tra  $[0, 1]$  tale che l'algoritmo riesce a determinare una schedulazione fattibile per qualunque insieme di task periodici su un singolo processore se l'utilizzazione totale dei task, cioè il rapporto di tutti i task sommati assieme, è minore o uguale a  $U_X$ . Se  $U_X = 0$ , l'algoritmo va malissimo, vuol dire che praticamente il processore è sempre libero. I task danno un carico infinitesimo al sistema, eppure non riesco a rispettare le scadenze. Se invece il fattore di utilizzazione è uguale ad 1, vuol dire che anche un insieme di task in cui il processore è occupato al 100%, perché la somma di tutti gli  $e_p$  è uguale a uno, quindi vuol dire che il processore non ha più tempo libero. Pure questo sistema di task così oneroso per il processore, con quell'algoritmo tutte le scadenze verranno rispettate. *Tanto maggiore è  $U_X$ , tanto migliore è l'algoritmo.*

## Confronto tra algoritmi di schedulazione (2)

*Qual è il fattore di utilizzazione dell'algoritmo FIFO? Zero!*

Esiste un insieme di due task con fattore di utilizzazione pari a  $\epsilon > 0$  piccolo a piacere che non è schedulabile con **FIFO**:  
 $T_1 = (10, 5\epsilon)$ ,  $T_2 = (20/\epsilon, 10)$

*Qual è il fattore di utilizzazione dell'algoritmo EDF? Uno!*

Ma non dovremmo dimostrarlo?! (Sappiamo solo che **EDF** è ottimale per job interrompibili ed indipendenti. . .)

*L'algoritmo EDF è semplice e ottimale, perché dovremmo studiare/adottare/cercare altri algoritmi?*

- Non esiste un modo efficiente per determinare quali job saranno in ritardo in caso di sovraccarico o overrun in una schedulazione a priorità dinamica quale **EDF**
- Qual è la priorità **EDF** di un job in ritardo?
- Il comportamento di un algoritmo a priorità fissa è predicibile anche in caso di sovraccarico o overrun

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

RS.11

Quale è il fattore di utilizzazione dell'algoritmo **FIFO**? Esiste un insieme di due task con utilizzazione pari con  $\epsilon > 0$  piccolo a piacere, ma che comunque non è schedulabile con **FIFO**. Quando vado a sommare le utilizzazioni di questi due task ottengo esattamente un'utilizzazione pari ad  $\epsilon$ .

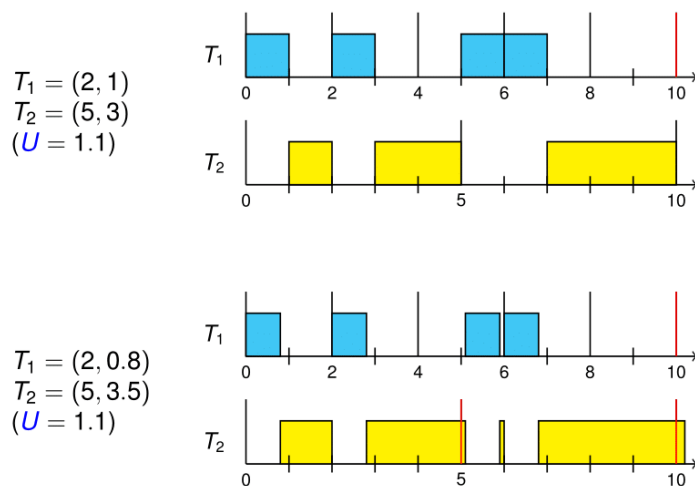
Qual è il fattore di utilizzazione dell'algoritmo **EDF**? Uno! L'algoritmo **EDF** è ottimale, ma in realtà questa conclusione non è giustificata da quello che abbiamo sentito dire fino ad adesso. Dovremmo dimostrarlo. Noi sappiamo soltanto che **EDF** è ottimale per job interrompibili, indipendenti su singolo processore. Abbiamo visto la dimostrazione, scorsa volta, che posso trasformare qualsiasi schedulazione ammissibile in una **EDF**. Ma non abbiamo mai detto che tutti i sistemi di task, anche quelli che occupano il processore al 100%, sono schedulabili, rispettano le scadenze. Questa è un'altra cosa. Per dire che **EDF** ha utilizzazione pari ad 1, vuol dire che riesce a schedulare anche gli insiemi di task che occupano il processore al 100%. Ma chi l'ha detto che questo sistema di task sia necessariamente schedulabile, anche se i job sono interrompibili, indipendenti e sul singolo processore? Dobbiamo dimostrarlo.

A questo punto, l'algoritmo **EDF** è ottimale. Allora perché spenderci e cercare e studiare altri algoritmi di schedulazione? La risposta è che **EDF** ha i suoi problemi. Uno dei problemi principali è che va benissimo se tutto va bene, ma nel momento in cui le cose cominciano ad andare male, cioè comincia a cedere l'hardware, per cui i job mancano le scadenze, rallentano, a quel punto in **EDF** è molto difficile capire che cosa succederà. Capire quali sono i job che soffriranno

da questo rallentamento diventa molto difficile. Un altro problema è: *io so che EDF da la priorità in funzione della scadenza assoluta del job. Che succede se il job supera la scadenza? La priorità diventa negativa o continua ad avere una priorità più alta di tutti?* **EDF** in se non definisce cosa succede quando il job è in ritardo. La scelta di cosa fare quando il job è in ritardo è una scelta soggettiva, con tutti i problemi che torniamo ad avere in questo caso. Potrebbe essere una scelta che porta ad un disastro, e questo è un problema.

D'altra parte, se io prendo un algoritmo a priorità fissa tipo **RM**, questo algoritmo ha un comportamento predicibile anche quando qualche job va in over-run, cioè supera le scadenze, o quando l'intero sistema è sovraccarico. Il comportamento dell'algoritmo continua ad essere predicibile, perché questo non è legato alla vicinanza della scadenza ma è legato a parametri del task che non cambiano, anche quando il sistema è sovraccarico.

### Comportamento di EDF con sovraccarico



Ottimalità di algoritmi priority-driven  
Marco Cesati

[Schema della lezione](#)  
[Validazione](#)  
[Fattore di utilizzazione](#)  
[Test di schedulabilità](#)

SERT'20 R5.12

Vediamo un esempio. In questo esempio abbiamo in sistema di 2 task. Il sistema è volutamente sovraccarico. Se io faccio l'utilizzazione totale, abbiamo 1.1, quindi è ovvio che nessun algoritmo riuscirà a rispettare le scadenze. Vediamo però che in questo caso **EDF** manca la scadenza per un job di  $T_1$ . Se però il sovraccarico si verifica per parametri lievemente differenti, per cui l'utilizzazione totale è ancora 1.1, in realtà **EDF** da un comportamento completamente diverso. Mancano le scadenze non soltanto un certo job di  $T_1$ , ma anche job di  $T_2$ . Non c'era modo di prevedere ciò. Se ho un sovraccarico, non sappiamo dire a priori che cosa succederà. Questo per molti sistemi RT può essere un problema, ed

è giustificato utilizzare un algoritmo a priorità fissa anche se ha prestazioni inferiori.

### Fattore di utilizzazione di EDF

#### Teorema (Liu & Layland, 1973)

Un sistema  $\mathcal{T}$  di task indipendenti ed interrompibili con scadenze relative uguali ai rispettivi periodi e fattore di utilizzazione  $U_{\mathcal{T}}$  ha una schedulazione fattibile su un singolo processore se e solo se  $U_{\mathcal{T}} \leq 1$

#### Corollario

L'algoritmo **EDF** ha fattore di utilizzazione  $U_{\text{EDF}} = 1$  per sistemi di task indipendenti, interrompibili e con scadenze relative uguali o maggiori dei rispettivi periodi

#### Dim. del Teorema (sketch):

- La parte "solo se" è banale
- Per la parte "se": troviamo un algoritmo che produce una schedulazione fattibile di ogni sistema  $\mathcal{T}$  con  $U_{\mathcal{T}} \leq 1$
- Candidati? **EDF**!


Teorema del 1973. Noi vogliamo dimostrare che effettivamente, se un sistema di job che sono interrompibili, indipendenti, che sono schedulati su un singolo processore, occupa il processore al più al 100%, allora necessariamente esiste una schedulazione che soddisfa le scadenze.

Corollario, dovuto al fatto che sappiamo che **EDF** è ottimale e quindi che se esiste una schedulazione che rispetta le scadenze, allora anche **EDF** lo fa, tutto questo ci porta ad enunciare il seguente corollario: l'algoritmo **EDF** ha un fattore di utilizzazione uguale ad uno, per sistemi di task indipendenti, interrompibili e con scadenze relative uguali o maggiori dei rispettivi periodi. Aggiungiamo che le scadenze possono essere maggiore dei rispettivi periodi, e continua a valere il corollario.

Non facciamo dimostrazioni formali per filo e per segno, ma facciamo degli abbozzi di dimostrazioni, cercando di spiegare perché i teoremi sono veri. Cerchiamo di spiegare perché il teorema è vero. Ovviamente la parte *solo se* è banale. Se ho un sistema di task che occupa il processore più del 100%, ovviamente qualcuno mancherà le scadenze, perché banalmente non ci sarà il tempo per completare tutti i job.

In realtà quello che cerchiamo di dimostrare è la parte del *se*. Se  $U_t \leq 1$ , allora esiste una schedulazione che rispetta le scadenze. Dobbiamo cercare un

Ottimalità di algoritmi priority-driven  
Marco Cesati



Schema della lezione  
Validazione  
Fattore di utilizzazione  
Test di schedulabilità

SERT'20 R5.13

algoritmo che produce una schedulazione fattibile di ogni sistema con  $U_t \leq 1$ . Ovviamente il candidato ideale è **EDF**. Non dobbiamo farci ingannare da questo gioco di rimandi. Dobbiamo dimostrare che **EDF** ha fattore di utilizzo uguale a 1. Sfrutto il fatto che è ottimale, cioè che esiste una schedulazione allora lui la trova. E poi dico: *perfetto, quindi se io dimostro che ogni sistema di task che occupa il processore al 100% ha una schedulazione qualunque che rispetta le scadenze, allora ho dimostrato questo teorema e quindi ho dimostrato che  $U_t$  è uguale ad 1*. Per dimostrare che ogni sistema di task che occupa il processore al 100% con queste condizioni ha schedulazione fattibile, uso *EDF*. Lo sto usando per dimostrare non per il fattore di utilizzazione direttamente, ma che EDF è in grado di trovare questa schedulazione.

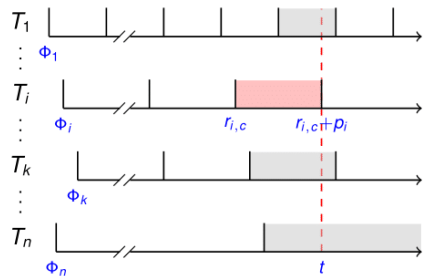
### Fattore di utilizzazione di EDF (2)

Da dimostrare: se **EDF** non trova una schedulazione fattibile, allora  $U_T > 1$

Al tempo  $t$  il (primo) job  $J_{i,c}$  non completa entro la scadenza

Assumiamo che il processore non sia mai idle prima di  $t$

1° caso: i periodi che includono  $t$  iniziano sempre dopo  $r_{i,c}$



Tutti i job nei periodi che includono  $t$  non sono eseguiti prima di  $t$  perché hanno scadenze dopo  $J_{i,c}$

$$t < \frac{(t - \phi_i) e_i}{p_i} + \sum_{k \neq i} \left\lfloor \frac{t - \phi_k}{p_k} \right\rfloor e_k \leq t \frac{e_i}{p_i} + \sum_{k \neq i} t \frac{e_k}{p_k} = t U_T \Rightarrow \boxed{U_T > 1}$$

Come si fa? Supponiamo di avere un sistema di task in cui **EDF** non riesce a trovare una schedulazione fattibile, allora posso dimostrare che il fattore di utilizzazione del sistema di task è sicuramente maggiore di 1. E' chiaro, sto cercando di dimostrare l'opposto, la negazione del teorema. Se è  $U_t \leq 1$  vuol dire che è schedulabile. Se quindi **EDF** non trova una schedulazione fattibile, vuol dire che  $U_t > 1$ , questo sto cercando di dimostrare. Supponiamo quindi di avere una schedulazione fattibile, quindi in cui effettivamente si possono rispettare le scadenze, ma tale per cui **EDF** non trova questa schedulazione. In realtà ci basta dimostrare che: consideriamo una schedulazione per cui **EDF** non trova il rispetto delle scadenze. Questo cosa significa? Che esiste un job che non rispetta la scadenza.

Ottimalità di algoritmi priority-driven  
Marco Cesati

Schema della lezione  
Validazione  
Fattore di utilizzazione  
Test di schedulabilità

SERT'20 R5.14

Supponiamo che sia il primo job che non rispetta la scadenza,  $J_{i,c}$ , il  $c$ -esimo job del task  $T_i$ . Manca la scadenza al tempo  $t$ . Le scadenze sono implicite, quindi sono uguali ai periodi, quindi  $t$  cade anche sul periodo del task  $T_i$ . Un'altra semplificazione è assumere che il processore dal tempo 0, in cui ho iniziato ad eseguire il sistema, al tempo  $t$  non sia mai rimasto idle.

Primo caso. Abbiamo  $T_i$  che è il task che contiene il primo job che manca la scadenza, e la manca al tempo  $t$ . Al tempo  $t$  cade il periodo del task  $T_i$ , qui il job che dovrebbe essere eseguito tra  $r_{i,c}$  ed il momento del rilascio del job che manca la scadenza. Da  $r_{i,c}$  ad  $r_{i,c} + p_i$  qui deve eseguire il nostro job ma non riesce a completare entro la scadenza. Il primo caso in cui ci poniamo è il caso in cui i periodi di tutti gli altri task che includono il tempo  $t$ , mostrati in grigio, iniziano sempre dopo  $r_{i,c}$ , dunque dopo il rilascio del job di  $T_i$  che manca la scadenza.

Siccome stiamo schedulando con **EDF**, vuol dire che le scadenze di tutti i job che vengono rilasciati nei periodi grigi, cioè quelli in cui cade  $t$ , hanno una priorità inferiore a quello che manca la scadenza. Perché la loro scadenza assoluta è oltre  $t$ , altrimenti sarebbe un periodo che precede, ed invece no. Questo è il periodo in cui cade  $t$ , quindi la scadenza sta dopo. Ma siccome iniziano dopo il rilascio di questo job, non soltanto hanno una priorità inferiore perché, ma anche che non hanno preso nessun tempo di processore. Siccome vengono rilasciati quando il job di  $T_i$  che manca la scadenza è stato già rilasciato, non hanno rubato nessun tempo di processore al job che manca la scadenza in quanto la loro priorità è inferiore. A questo punto facciamo il *conto della serva*. Che cosa vuol dire che manca la scadenza?

Vuol dire che il processore non ce la fa ad eseguire tutto il carico che gli è stato chiesto. Dunque il tempo che passa tra l'istante iniziale e l'istante  $t$ , in cui si manca la scadenza, questo tempo  $t$  è minore di tutto il lavoro da fare. Che cosa è il lavoro da fare? Innanzitutto c'è il lavoro da fare per il task  $T_i$ , dove c'è il job che manca la scadenza. Quanto devo fare di questo valore?

- $\phi_i$  è la fase di  $i$ , il primo rilascio.  $t - \phi_i$  è il numero di rilasci del job di  $T_i$  tra 0 e  $t$ . E' un numero multiplo di  $p_i$ , in quanto il periodo corrisponde a  $t$
- Questo numero di rilasci lo moltiplico per  $e_i$ , il tempo di esecuzione dei job del task di  $T_i$

Questo è quanto occupa il task  $T_i$  fino al tempo  $t$ . Quanto occupano tutti gli altri task? Prendiamo il generico task  $T_k$ . Quanti periodi ci sono?

- $\frac{t - \phi_k}{p_k}$ , che generalmente è un numero reale, visto che non è detto che  $t$  cada sul confine di  $T_k$ . Per risolvere questo, ne considero la parte intera inferiore, in quanto la parte che eccede coincide all'ultimo periodo *grigio* in cui il job rilasciato non può aver rubato alcun tempo al processore per il task  $T_i$ . Dunque qui ha senso prendere la parte intera inferiore. La quantità di lavoro che i task, non  $T_i$ , occupano sul processore è esattamente



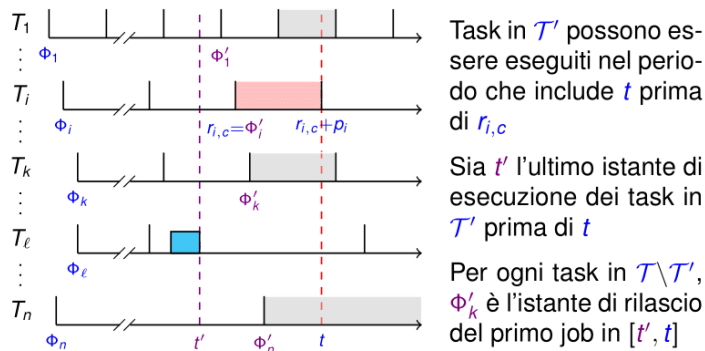
parte intera inferiore del valore, moltiplicato ovviamente per il tempo di esecuzione  $e_k$

In tutte queste dimostrazioni il gioco è capire se si tratta di un numero intero o reale. Nel secondo caso, considero la parte intera superiore o inferiore? Tutto, molto spesso, si gioca su questo meccanismo. Qui devo prendere la parte intera inferiore, in quanto i job rilasciati nell'intervallo in grigio non posso rubare tempo ai job che mancano la scadenza. Quindi, con un semplice passaggio, possiamo togliere tutte le fasi che sono valori  $\geq 0$ , e quello che ottengo è ancora  $\leq$ . Posso togliere la parte intera in quanto quello che ottengo è ancora  $\leq$ .

A questo punto, facendo queste maggiorazioni, ottengo che  $t$  è strettamente minore di  $t \cdot U_t$ , ovvero che  $U_t > 1$ . Abbiamo dunque dimostrato che in questo caso, se **EDF** non trova una schedulazione, necessariamente vuol dire che  $U_t > 1$ .

### Fattore di utilizzazione di EDF (3)

**2° caso:** l'insieme dei task  $\mathcal{T}'$  in cui il periodo che include  $t$  inizia prima di  $r_{i,c}$  è non vuoto



$$\begin{aligned} t - t' &< \frac{(t - \Phi'_i) e_i}{p_i} + \sum_{\substack{T_k \in \mathcal{T} \setminus \mathcal{T}' \\ k \neq i}} \left\lfloor \frac{t - \Phi'_k}{p_k} \right\rfloor e_k \leq (t - t') \sum_{T_k \in \mathcal{T} \setminus \mathcal{T}'} \frac{e_k}{p_k} \\ &\leq (t - t') U_{\mathcal{T}} \Rightarrow \boxed{U_{\mathcal{T}} > 1} \end{aligned}$$

Vediamo il secondo caso, quello in cui esistono un po di task in cui effettivamente il periodo che include  $t$  inizia dopo  $r_{i,c}$ , ma esiste anche un'insieme di task  $T'$  in cui il periodo che include  $t$  inizia prima, che nel nostro caso sarebbe il task  $T_l$ . Cosa possiamo fare? Possiamo dire: *però, di fatto, siccome  $t$  cade dentro questo periodo, la priorità di questo job dentro  $T_l$ , ovvero dentro tutti i task di questo insieme  $T'$ , è comunque inferiore alla priorità del job che manca la scadenza. Quindi possono essere eseguiti soltanto prima che venga rilasciato il job all'istante  $r_{i,c}$ .*

Dunque i task in  $T'$  possono essere eseguiti nel periodo che include  $t$  soltanto

prima di  $r_{i,c}$ . Non possono essere eseguiti dopo, in quanto l'algoritmo a priorità assegnerà una priorità più alta al job di  $T_i$ . Quindi posso definire un'istante  $t'$  che è l'ultimo istante di esecuzione dei task di  $T'$  prima di  $t$ . Tra tutti i task  $T'$ , che hanno il periodo che include  $t$  che inizia prima di  $r_{i,c}$ , posso definire l'ultimo momento in cui viene eseguito uno di questi task, e lo chiamo l'istante  $t'$ . Poi applico il discorso precedente all'intervallo  $[t', t]$ . Chiamo  $\phi'_k$  l'istante di rilascio del primo job tra l'intervallo  $[t', t]$ . Tutti i task in  $T'$  contribuiranno 0 all'occupazione del processore in  $[t', t]$ , perché la loro scadenza sta oltre  $t$ . Fondamentalmente li posso non contare, e conto solo quelli che occupano il processore tra  $t'$  e  $t$  e rifaccio il discorso di prima, dove al posto dei  $\phi$  ci stanno i  $\phi'$ .

Fondamentalmente ci riconduciamo al caso precedente, semplicemente considerano l'ultimo istante in cui i task  $T'$  hanno eseguito. Questo dà la dimostrazione. Dunque abbiamo dimostrato che se l'algoritmo **EDF** manca la scadenza, cioè non riesce a schedulare in modo che tutti rispettano le scadenze, nelle ipotesi di task interrompibili, indipendenti e con singolo processore, è perché la somma totale di tutte le utilizzazioni dei task è maggiore di 1. Quindi se l'utilizzazione totale dei task è minore o uguale a 1, **EDF** riesce a schedulare. Se l'utilizzazione totale di tutti i task è minore o uguale a 1 esiste un algoritmo di schedulazione che rispetta le scadenze. Quindi, **EDF** che è ottimale, rispetta le scadenze ed ha fattore di utilizzazione uguale ad 1.

### Riassumendo...

Si assumano task indipendenti, interrompibili e schedulati su un singolo processore


Abbiamo stabilito che:

- Se un sistema di task ammette una schedulazione fattibile, l'algoritmo **EDF** ottiene una schedulazione fattibile per quel sistema di task
- Un sistema di task ammette una schedulazione fattibile se e solo se  $U_T \leq 1$
- L'algoritmo **EDF** ha fattore di utilizzazione  $U_{EDF} = 1$  (almeno per scadenze uguali ai periodi)

Tra poco dimostreremo che l'algoritmo **EDF** ha fattore di utilizzazione  $U_{EDF} = 1$  anche quando le scadenze sono uguali o maggiori dei periodi

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

**Fattore di utilizzazione**

Test di schedulabilità

SERT'20 R5.16

Riassumendo, abbiamo task indipendenti, interrompibili e schedulati sul singolo

processore. Se un sistema di task ammette una schedulazione fattibile, **EDF** ottiene una schedulazione fattibile per quel sistema di task. Se un sistema di task ammette una schedulazione fattibile, allora la sua utilizzazione totale  $U_t \leq 1$ , ma è vero anche il viceversa. Quindi, il fattore di utilizzazione di **EDF** è uguale ad 1, almeno per le scadenze uguali ai periodi. Tra un'istante potremo dimostrare che l'algoritmo **EDF** ha fattore di utilizzazione uguale ad 1 anche quando le scadenze sono maggiori o uguali ai periodi.

### Densità di un sistema di task

Il teorema appena dimostrato non è valido se qualche task ha scadenza relativa **inferiore** al periodo; ad esempio:

- $T_1=(2, 0.9), T_2=(5, 2.3) \Rightarrow U=0.91 \Rightarrow$  schedulabile
- $T_1=(2, 0.9), T_2=(5, 2.3, \mathbf{3}) \Rightarrow$  non schedulabile ( $\Delta=1.22$ )

Si definisce **densità** di un task  $(\phi, p, e, D)$  il rapporto  $\frac{e}{\min(D, p)}$

#### Teorema

Un sistema  $\mathcal{T}$  di task indipendenti ed interrompibili e densità  $\Delta_{\mathcal{T}}$  ha una schedulazione fattibile su un singolo processore se  $\Delta_{\mathcal{T}} \leq 1$

- È condizione sufficiente, non necessaria
- $T_1=(2, 0.6, \mathbf{1}), T_2=(5, 2.3) \Rightarrow \Delta=1.06$  ma è schedulabile!


Che succede se non vale l'ipotesi per cui le scadenze relative sono almeno uguali ai periodi? Se qualche scadenza relativa è inferiore al periodo, tutto questo non vale. Abbiamo due task, con utilizzazione  $U = 0.91$ , quindi schedulabile. Se però dico che  $T_2$  deve avere scadenza non più entro 5 ma entro 3, allora non riesco più a rispettare le scadenze, non c'è modo.

Come possiamo fare a capire, avere una dimostrazione, che questo succede? Quello che siamo interessati a fare è una dimostrazione che anche se le **densità** sono inferiori al periodo, le scadenze potranno essere rispettate. Per fare questo introduciamo il concetto di **densità** del task. E' il rapporto  $\frac{e}{\min(D, p)}$ , dove  $D$  è la scadenza relativa del task. Se la scadenza relativa è maggiore o uguale del periodo, allora la densità coincide con l'utilizzazione. Ma se la scadenza relativa è inferiore al periodo, allora la densità non è l'utilizzazione, ma è un numero più grande.

**Teorema.** Un sistema di task indipendenti, interrompibili e densità  $\Delta_{\mathcal{T}}$  ha una schedulazione fattibile su un singolo processore se  $\Delta_{\mathcal{T}} \leq 1$ .

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20 R5.17

Questo teorema è una condizione sufficiente, non necessaria. Ad esempio, qui la densità è 1.06 e non rispetta la condizione del teorema. Eppure, se vado a vedere, esiste una schedulazione che rispetta le scadenze per questo sistema di task. Non è vero che se la densità è  $> 1$  il sistema non è schedulabile. Era vero per l'utilizzazione, quando le scadenze erano implicite o maggiori o uguali al periodo. Ma non è vero per la densità. Ci sono sistemi di task con densità maggiore di 1 che continuano ad essere schedulabili. Dobbiamo però leggere questi risultati in ottica di validazione. Se io costruisco un sistema in cui la densità dei task è minore o uguale di 1 allora sono sicuro che rispetterà le scadenze. E questo è quello che conta. Il fatto che possa costruire dei sistemi che hanno una densità maggiore di 1 e che continuano ad essere schedulabili, può essere interessante ma non è molto utile, perché fondamentalmente non ho un risultato analitico che mi dica quale di questi sistemi è schedulabile e quale no. Io sono interessato a dire che se questa condizione è rispettata, sicuramente rispetterò le scadenze, quindi rispetterò la condizione e vado sul sicuro. Quello che mi interessa di questi teoremi è la condizione sufficiente.

### Fattore di utilizzazione e densità

In un sistema  $\mathcal{T}$  di task interrompibili con fattore di utilizzazione  $U_{\mathcal{T}} = \sum_k e_k/p_k$  ed un singolo processore:

- T1) Se, per ogni task  $T_i$ ,  $D_i = p_i$ , allora esiste una schedulazione fattibile se e solo se  $U_{\mathcal{T}} \leq 1$
- C1) Se, per ogni task  $T_i$ ,  $D_i \geq p_i$ , allora esiste una schedulazione fattibile se e solo se  $U_{\mathcal{T}} \leq 1$
- C2) Il fattore di utilizzazione di EDF per task con  $D_i \geq p_i$  è  $U_{EDF} = 1$  (ossia EDF determina una schedulazione fattibile se  $U_{\mathcal{T}} \leq 1$ )
- T2) Se per qualche task  $T_i$ ,  $D_i < p_i$ , allora esiste una schedulazione fattibile se  $\Delta_{\mathcal{T}} = \sum_k e_k / \min(p_k, D_k) \leq 1$

Quando si verifica il caso  $U_{\mathcal{T}} > \Delta_{\mathcal{T}}$ ? **Mai!**


- Se  $\exists T_i$  tale che  $D_i < p_i$ , allora  $U_{\mathcal{T}} < \Delta_{\mathcal{T}}$
- Se  $\forall T_i$ ,  $D_i \geq p_i$ , allora  $U_{\mathcal{T}} = \Delta_{\mathcal{T}}$

In un sistema di task interrompibili, con fattore di utilizzazione  $U_t$  ed un singolo processore:

1. Se le scadenze sono implicite, quindi  $D_i = p_i$  allora esiste una schedulazione fattibile se e solo se  $U_t \leq 1$
2. Se per ogni task la scadenza relativa  $D_i \geq p_i$ , allora esiste una schedulazione fattibile se e solo se  $U_t \leq 1$

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

**Fattore di utilizzazione**

Test di schedulabilità

SERT'20 R5.18

1. Questo in realtà dobbiamo ancora dimostrarlo. In realtà la parte se è facile, in quanto ci sta dicendo che l'utilizzazione totale è minore o uguale ad 1, quindi tutto il sistema di task sarà schedato in un modo che tutti i job completano entro il periodo, perché valeva per le scadenze implicite. Ma se la scadenza è maggiore o uguale del periodo allora varrà a maggior ragione per il sistema di task in cui le scadenze sono oltre il periodo. Quindi la parte se è facile. Il problema è il **solo se**, che va dimostrato.
2. Vogliamo dire che il fatto che le scadenze siano oltre i periodi, non ci consente comunque di caricare un processore più del 100%. Anche se le scadenze sono sopra ai periodi. Questa cosa in realtà va dimostrata in modo formale, e lo facciamo tra un attimo
3. Il fattore di utilizzazione di **EDF**, per task con  $D_i \geq p_i$  è uguale ad 1,  $U_{EDF} = 1$ . Se il sistema di task ha un'utilizzazione totale al massimo uguale ad 1, **EDF** riesce a trovare una schedulazione fattibile.
4. Se per qualche task la scadenza relativa è strettamente inferiore al periodo,  $D_i < p_i$ , non possiamo più parlare di utilizzazione ma si parla di densità. Esiste una schedulazione fattibile se la densità totale  $\Delta_t \leq 1$ .

Quando si verifica il caso che  $U_t > \Delta_t$ ? Mai. Perché se esiste il caso in cui la scadenza relativa è minore del periodo, allora ovviamente per necessità il fattore di utilizzazione è strettamente minore della densità. Mentre se, all'opposto, per tutti i task la scadenza relativa è maggiore o uguale del periodo allora banalmente la densità e l'utilizzazione sono uguali. Dunque in nessun caso l'utilizzazione sarà maggiore della densità.

## Fattore di utilizzazione e scadenze oltre i periodi

Dimostriamo che: se per ogni task  $D_i \geq p_i$ , allora esiste una schedulazione fattibile solo se  $U_T \leq 1$

- Per un solo task (base dell'induzione):

$$e \leq D, \quad 2e \leq D + p \quad \dots \quad (k+1)e \leq D + kp \quad \dots$$

$$\text{Quindi } \forall k \geq 1, \quad \frac{e}{p} < \frac{k+1}{k} \cdot \frac{e}{p} \leq 1 + \frac{D}{pk} \implies \frac{e}{p} \leq 1$$

- Sia vero per  $n-1$  task in fase, allora per  $T_n$  e ogni  $k$  intero:


$$(k+1)e_n \leq (D_n + k p_n) \cdot \left(1 - \sum_{i=1}^{n-1} \frac{e_i}{p_i}\right)$$

$$\forall k \geq 1, \quad \frac{e_n}{p_n} < \left(\frac{D_n}{k p_n} + 1\right) \cdot \left(1 - \sum_{i=1}^{n-1} \frac{e_i}{p_i}\right) \implies \boxed{U_T \leq 1}$$

- Per task non in fase: stessa idea applicata dall'istante della fase maggiore

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

**Fattore di utilizzazione**

Test di schedulabilità

SERT'20
RS.19

Vediamo di dimostrare questa cosa che ci manca. vogliamo dimostrare che se per ogni task la scadenza relativa è maggiore o anche uguale al periodo, allora esiste una schedulazione fattibile soltanto se  $U_t \leq 1$ . In altri termini, che se  $U_t > 1$  allora necessariamente non si potrà avere una schedulazione che rispetta le scadenze. Come lo dimostriamo? Anche questa è una bozza, che in realtà non troviamo sul libro di testo, l'ha inventata il professore. La facciamo per induzione. Cominciamo a ragionare su un singolo task. Questo task ha una scadenza oltre il periodo. Ci possiamo permettere quindi che abbia anche un'utilizzazione  $\frac{e}{p}$  maggiore di 1? A prima vista uno direbbe di sì, perché tanto, nel momento in cui vado oltre il periodo, in realtà la mia scadenza è ancora oltre, quindi fondamentalmente quello che è sufficiente è che il tempo di esecuzione sia minore o uguale della scadenza relativa. Però in realtà io questo discorso devo poterlo applicare anche al secondo job di quel task, che deve anche lui terminare entro la sua scadenza assoluta. E quindi la distanza che c'è tra 0 e la scadenza assoluta del secondo job, quanto è? E'  $p$ , tempo di rilascio del secondo job, +  $D$  scadenza relativa di questo task.

$D + P$  deve essere maggiore o uguale del tempo che impiego per eseguire non più un job, ma due, il primo ed il secondo. Questa cosa la posso iterare, la debbo iterare, per tutti gli infiniti job del task. Posso scrivere che  $(k+1) - volte e$  deve essere minore o uguale di  $D$ , scadenza relativa, più  $k - volte$  il periodo per ogni  $k$ . Ma questo quindi, cosa significa? Significa che se scriviamo  $\frac{e}{p}$  e fissiamo un  $k$  qualunque, possiamo scrivere la disuguaglianza sulle slide ed alla fine otteniamo  $\frac{e}{p} \leq 1$ . Con un solo task ed era quello che volevamo dimostrare. Questa è la base

dell'induzione. In realtà, data per assunto che sia vero per  $n - 1$  task, allora quello che posso dire è che per  $T_n$  e per ogni  $k$  intero posso dire che

- $(k + 1)e_n$ 
  - Per poter eseguire  $k + 1$  job del task  $T_n$  mi occorre un tempo che deve essere minore o uguale di cosa?
- $D_n + kp_n$ 
  - Della scadenza assoluta del  $k + 1$  -esimo job di  $T_n$
- A questo punto il processore non è tutto per me, in quanto deve eseguire gli altri  $n - 1$  task precedenti che per ipotesi induttiva possiamo dire che occupano un tempo che è la sommatoria nella slide

A questo punto, stesso discorso di prima, ed arrivo al risultato per cui  $U_t \leq 1$ . Al di là dei dettagli della dimostrazione che non sono difficili, l'idea è quella di avere l'idea intuitiva. Quale è? Ho delle scadenze relative che sono oltre il periodo. Ma il problema è che se io sfrutto il fatto che le scadenze relative sono oltre il periodo, vuol dire che nei periodi successivi io comincerò a poter eseguire i job rilasciati in quel periodo un po' oltre i loro periodi, perché un po' del tempo di quel periodo me lo sono mangiato perché l'utilizzazione precedente è andata oltre, cioè andava oltre il periodo. E questa cosa, io posso avere anche una scadenza che è mille volte più grande del periodo, ma è comunque una dimensione finita. E siccome i miei task vanno avanti all'infinito, ogni volta che rilascio un task io mi mangio un pezzettino di quel vantaggio che avevo, ed alla fine arriverò ad un task che mancherà la scadenza. Questa è l'idea della dimostrazione riformulata in maniera formale, ma l'idea è che qualunque sia la quantità, andando avanti con il rilascio dei job quel vantaggio me lo sono mangiato e quindi mancherà la scadenza.

Quindi quello che possiamo dire è che il risultato sul fattore di utilizzazione  $U_t \leq 1$  vale anche, non solo per le scadenze implicite, ma anche quando queste sono maggiori o uguali al periodo. Se i task non sono in fase si può complicare la dimostrazione, applicarla dall'istante della fase maggiore ecc.

## Test di schedulabilità per EDF

*Dato un sistema  $\mathcal{T}$  completamente definito di task interrompibili, come stabilire se è schedulabile con EDF?*

- Se  $\Delta_{\mathcal{T}} \leq 1$  è schedulabile (T2 e C1)
- Altrimenti: se  $D_i \geq p_i$  per ogni  $i$  non è schedulabile (C1)
- Altrimenti: se i task sono in fase applichiamo EDF per un segmento lungo  $2H + \max p_i + \max D_i$ , ove  $H$  è l'iperperiodo (Baruah, Howell, Rosier 1993)

*E se il sistema non è completamente determinato? Ad esempio, i tempi di esecuzione o gli istanti di rilascio possono variare*

Il sistema continua ad essere schedulabile anche quando:

- i tempi di esecuzione sono più corti dei tempi di esecuzione massimi (sistema predicibile)
- i task sono sporadici, ossia gli intervalli di rilascio dei job sono maggiori dei rispettivi periodi (dalla dimostrazione del teorema)

... ma se le fasi sono sconosciute non si può simulare!

Ricapitolando. Se abbiamo un sistema di task, completamente definito, cioè conosciamo tutti i parametri temporali, ed i task sono interrompibili, un solo processore, indipendenti ecc. Come facciamo a stabilire se è schedulabile con EDF?

1. Vediamo la densità, o se sappiamo che le scadenze sono implicite, vediamo l'utilizzazione. Se  $\Delta \leq 1$  è schedulabile
2. Altrimenti: se  $D_i \geq p_i$ , in realtà sappiamo che questa densità è uguale all'utilizzazione, ma se l'utilizzazione è maggiore di 1 allora non è schedulabile
3. Altrimenti: se i task sono in fase applichiamo EDF per un segmento lungo  $2H + \max p_i + \max D_i$  ove  $H$  è l'iperperiodo. Se simulo per questo tempo ed EDF rispetta sempre le scadenze, allora posso concludere che tutti rispetteranno le scadenze

Qual'è il problema? Che il mio sistema di task potrebbe non essere completamente determinato. Potrei non avere tutte le informazioni necessarie per simulare.

- Posso avere che i tempi di esecuzione non sono fissati, o che gli istanti di rilascio non sono fissati. In quel caso che simulo? Se i tempi di esecuzione sono più corti, questo non è un problema perché il sistema in queste condizioni è **predicibile**.
- D'altra parte, se i task sono sporadici, cioè i periodi non sono esatti ma gli intervalli minimi, d'accapo posso applicare tutto quello visto prima e dimostrare ancora che vale la schedulabilità ed il risultato che trovo

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

RS.20



Ma se altri parametri come le *fasi* sono sconosciuti, a quel punto non c'è niente da fare. Non posso simulare perché dovrei simulare il sistema per ogni possibile fase, cosa impossibile. La simulazione quindi in questo caso non si può fare, se ho incertezza sulle fasi. Possiamo dire qualcosa di più se le fasi sono sconosciute? Possiamo avere un risultato analitico?

### Analisi di schedulabilità per EDF

Se alcuni task hanno scadenze relative inferiori ai periodi, la condizione di schedulabilità è sufficiente ma non necessaria

#### Teorema (Baruah & al., 1990)

Un sistema  $\mathcal{T}$  con task periodici indipendenti, interrompibili, con  $U_{\mathcal{T}} < 1$ , è schedulabile con EDF se e solo se

$$\forall L > 0, \sum_{i=1}^n \left\lfloor \frac{L + p_i - D_i}{p_i} \right\rfloor \cdot e_i \leq L$$

La formula deve essere controllata soltanto per i valori  $L$  multipli dei periodi dei task entro l'iperperiodo e tali che

$$L \leq \max \left\{ D_1, \dots, D_n, \frac{\sum_{i=1}^n (p_i - D_i) \cdot (e_i/p_i)}{1 - U_{\mathcal{T}}} \right\}$$


In realtà possiamo fare una così detta analisi di schedulabilità. Sappiamo che la condizione di schedulabilità sulla densità  $p$  è una condizione sufficiente e non necessaria. In realtà possiamo fare un'analisi più precisa, in cui il risultato ci dà una condizione necessaria e sufficiente. Si tratta del teorema del '90 e dice che se ho un sistema di task periodici, indipendenti, interrompibili, con un singolo processore e con  $U_{\mathcal{T}} < 1$ , è schedulabile con **EDF** se e solo se per ogni valore di  $L > 0$ , la sommatoria su tutti i task di  $L + \text{il periodo del task meno la scadenza relativa}$ , tutto diviso il periodo del task di cui prendo la parte intera inferiore e moltiplico per  $e_i$ , deve essere minore o uguale di  $L$ . Se questo è vero per qualunque valore di  $L$  allora è schedulabile. Se per qualunque valore di  $L$  questo non è vero, allora il sistema di task non è schedulabile con **EDF**.

Ovviamente quando applico questo teorema in pratica non devo controllare infiniti valori di  $L$ , ma ne posso controllare soltanto un numero finito. Sono tutti i valori multipli del periodo dei task che sono dentro l'iperperiodo tali comunque che è minore del valore massimo che è possibile ottenere sfruttando la formula sulle slide.

In realtà questa analisi di schedulabilità in pratica non viene molto applicata.

Ottimalità di algoritmi priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20 R5.21

Diciamo che è interessante dal punto di vista della validazione, ma diciamo che si tende a preferire la formulazione con la densità anche se è più pessimistica ed è una condizione solamente sufficiente. Perché questa obiettivamente è di applicazione molto più onerosa e quindi si tende a preferire quell'altra.

### Schedulabilità di algoritmi a priorità fissa

Gli algoritmi a **priorità fissa** sono in generale peggiori di quelli a **priorità dinamica** rispetto alla capacità di determinare schedulazioni fattibili

Ad esempio:  $T_1 = (2, 1)$  e  $T_2 = (5, 2.5)$ :

- $U = 1$ , quindi sono schedulabili (ad esempio con **EDF**)
- $J_{1,1}$  e  $J_{1,2}$  devono avere priorità maggiore di  $J_{2,1}$  ( $T_1 > T_2$ )
- $J_{2,1}$  deve avere priorità maggiore di  $J_{1,3}$  ( $T_2 > T_1$ )
- Se le **priorità** sono **fisse**, o  $T_1 > T_2$  oppure  $T_2 > T_1$

*Esistono classi di sistemi che ammettono un algoritmo a priorità fissa ottimale? **Sì!***

Esempio: sistemi di task **semplicemente periodici** (o **armonici**)

#### Task semplicemente periodici

Per ogni coppia di task  $T_i$  e  $T_k$  con  $p_i < p_k$ ,  $p_k$  è un multiplo intero di  $p_i$

Vediamo di concludere cercando di ragionare su quelli che sono gli algoritmi a priorità fissa. Gli algoritmi a priorità dinamica come **EDF** possono essere ottimali. Quelli a **priorità fissa** in generale non lo sono. Abbiamo già visto un esempio nella lezione precedente di un insieme di task che era schedulabile con EDF ma non schedulabile con **Rate Monotonic** (priorità fissa). In realtà dunque sono peggiori rispetto alla possibilità di trovare schedulazioni che rispettano le scadenze.

Per esempio, il sistema di task con periodo 2 e tempo di esecuzione 1, e periodo 5 e tempo di esecuzione 2.5 ha utilizzazione uguale ad 1 ( $U = 1$ ). Se i task sono interrompibili vuol dire che **EDF** riesce a schedularli rispettando le scadenze. Ma non riusciamo a schedularlo con nessun algoritmo a priorità fissa. Vediamo cosa succede quando scheduliamo questo sistema di task con la priorità fissa. E' abbastanza intuitivo che il primo job di  $T_1$  deve avere una priorità maggiore del primo job di  $T_2$ , altrimenti il primo job di  $T_2$  andrebbe ben oltre la scadenza del job di  $T_1$ . In realtà anche  $J_{1,2}$  deve avere priorità maggiore del primo job di  $T_2$ , altrimenti il job di  $T_2$  non lascerebbe abbastanza tempo al secondo job per completare.

In realtà le cose si rovesciano per  $J_{2,1}$ . Il primo job di  $T_2$  deve avere una priorità

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.22

maggiore del terzo job di  $T_1$ , che viene rilasciato all'istante 4. Se avesse priorità maggiore del primo job di  $T_2$ , in realtà vorrebbe dire che esegue da 4 a 5. Quindi nell'intervallo fra 0 e 5, eseguono tutti e tre i job di  $T_1$ , cioè tempo totale 3. Tra 0 e 5 resterebbero disponibili per il primo job di  $T_2$  soltanto due unità di tempo. Lui ha bisogno di 2.5 unità di tempo e quindi manca la scadenza. Perché sia schedulabile, devo soddisfare questa condizione dunque, che con una priorità fissa è impossibile. La priorità di un task o è maggiore o è minore, non può cambiare per i singoli job.

Dunque qualsiasi algoritmo a priorità fissa, questo insieme di task non lo può schedulare. La domanda diventa: *ma esistono classi di sistemi che ammettono un algoritmo a priorità fissa come ottimale?* Per alcuni classi di sistemi, gli algoritmi a priorità fissa possono essere ottimali? La domanda è tutt'altro che teorica. Abbiamo visto che gli algoritmi a priorità fissa hanno la stessa semplicità degli algoritmi a priorità dinamica, ed anche hanno delle caratteristiche, proprietà, di predicibilità superiori a quelli a priorità dinamica. E quindi potrebbe avere molto senso in pratica scegliere un algoritmo a priorità fissa per un determinato problema. E quindi sapere se esistono classi di sistemi in cui la priorità fissa può essere un algoritmo ottimale, è una domanda tutt'altro che campata per aria.

La risposta è **si**. Esistono classi di sistemi in cui un algoritmo a priorità fissa è ottimale. Un esempio sono i sistemi di task **semplicemente periodici (o armonici)**, se per ogn coppia di task  $T_i$  e  $T_k$ , con  $p_i < p_k$ , e  $p_k$  è un multiplo intero di  $p_i$ .

### Ottimalità dell'algoritmo RM

#### Teorema

Un sistema  $\mathcal{T}$  di task **semplicemente periodici**, interrompibili ed indipendenti le cui scadenze relative sono non inferiori ai rispettivi periodi ha una schedulazione **RM** fattibile su un singolo processore se e solo se  $U_{\mathcal{T}} \leq 1$

**Dim.** (sketch): supponiamo che tutti i task siano in fase, che le scadenze siano uguali ai periodi e che il processore non sia mai idle

Il task  $T_i$  manca la scadenza al tempo  $t$

Ogni task  $T_k$  con priorità maggiore di  $T_i$  ha periodo più piccolo di  $p_i$ , perciò  $t$  è un multiplo intero di tutti i  $p_k$

$$t < \sum_{k=1}^i \frac{e_k \cdot t}{p_k} = t \cdot \sum_{k=1}^i \frac{e_k}{p_k} \leq t \cdot U_{\mathcal{T}} \Rightarrow \boxed{U_{\mathcal{T}} > 1}$$

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedulabilità

SERT'20

R5.23

Il caso classico. Tutti i task hanno periodi che sono potenze di due. In questo caso banalmente tutti i task sono armonici, tutti i task hanno periodi che sono in rapporto intero tra di loro. Questo caso particolare, i task a priorità fissa possono essere ottimali. Vediamo perchè. Un sistema di task semplicemente periodici, interrompibili, indipendenti con scadenze relative non inferiori ai rispettivi periodi, ha una schedulazione **RM** fattibile su un singolo processore se e solo se  $U_t \leq 1$ .

Se c'è un modo qualunque di schedulare l'insieme di task armonici, allora anche **RM** riesce a schedulare. **RM** è ottimale per questo sotto insieme di task. Come facciamo a dimostrarlo? E' una bozza di dimostrazione. Supponiamo che i task siano in fase, diciamo che le scadenze sono uguali ai periodi e che il processore non sia mai idle da 0 fino al nostro  $t$  generico. cosa vogliamo dimostrare?

Supponiamo per assurdo che un task  $T_i$  manca la scadenza al tempo  $t$ . La conclusione a cui voglio arrivare è che allora l'utilizzazione totale del sistema è maggiore di 1. Vediamo perchè. Se manco la scadenza al tempo  $t$ , facciamo il conto della serva. Allora ogni task  $T_k$  che ha priorità maggiore di  $T_i$ , vuol dire che ha un periodo più piccolo di  $T_i$ , perchè **RM** assegna la priorità in maniera tale che periodi più piccoli significano priorità maggiori. Questo vuol dire che ne momento in cui manco la scadenza vuol dire che lì c'era un periodo del task che manca la scadenza. Ma questo  $t$  quindi deve essere un multiplo intero di tutti i task che hanno priorità superiore a quello che manca la scadenza. Perché qui è importante parlare dei task di priorità superiore? Perché tutti i task di priorità inferiore che mancano la scadenza, non possono avergli rubato tempo. Se manca la scadenza è perchè sono i task di priorità superiore che gli hanno rubato il tempo necessario. Quello di priorità inferiore non gli hanno portato via tempo quindi non li considero nel conto, ma metto soltanto tutti i task che hanno priorità superiore a quella che manca la scadenza.

Quindi il mio tempo  $t$ , l'intervallo di tempo, non è sufficiente per completare il task  $T_i$  e tutti i task di priorità superiore. Quanti sono? Sono  $t$  diviso  $p_k$ , senza parti intere, in quanto  $t$  cade sul periodo di un task  $T_i$  ed è anche un multiplo di tutti gli altri in quanto sono task armonici. Non c'è questione di parti intere da togliere. Fondamentalmente,  $U_t > 1$ . E' lo stesso schema di prima. L'idea qui è che schedulo con **RM**, quindi nel conto considero solo quelli di priorità superiore, dunque vuol dire che i task hanno periodi più grandi ma sono armonici, e sono multipli interi, dunque non servono le parti intere nel conto. Se tolgo una di queste ipotesi, questa dimostrazione non funziona più.

## Ottimalità dell'algoritmo DM

### Teorema (Leung & Whitehead, 1982)

Se per un sistema di task periodici, indipendenti ed interrompibili che sono in fase ed hanno scadenze relative minori o uguali ai rispettivi periodi esiste un algoritmo a **priorità fissa** che produce una schedulazione fattibile, allora anche l'algoritmo **DM** produrrà una schedulazione fattibile

L'algoritmo **DM** coincide con **RM** se tutte le scadenze relative sono proporzionali ai rispettivi periodi

### Corollario

L'algoritmo **RM** è ottimale tra tutti gli algoritmi a **priorità fissa** qualora le scadenze relative dei task siano non superiori e proporzionali ai rispettivi periodi

*Perché il corollario non richiede che i task siano tutti in fase?*

Perché avere i task in fase è il caso peggiore possibile!

Possiamo considerare l'algoritmo **DM**. Abbiamo visto che **DM** era un algoritmo ottimale fra tutti gli algoritmi a priorità fissa. O meglio, abbiamo visto che **DM** non poteva essere peggiore di **RM**. In realtà nell'82 si è dimostrato che **DM** è un algoritmo ottimale per tutti i sistemi di task che sono schedulabili con una priorità fissa. In altri termini, fra tutti i sistemi di task che sono schedulabili con una priorità fissa, **DM** riuscirà sempre a produrre una schedulazione fattibile. Se per un sistema di task periodici indipendenti, interrompibili che sono in fase, ed hanno scadenze relative minore o uguali ai rispettivi periodi, se esiste un algoritmo a priorità fissa che produce una schedulazione fattibile, allora anche l'algoritmo **DM** produrrà una schedulazione fattibile. Ovviamente qui non c'è l'ipotesi delle scadenze minori del periodo, perché appunto **DM** tiene conto delle scadenze minori del periodo. Considera come priorità la scadenza relativa.

**Corollario.** Il corollario è che siccome l'algoritmo **DM** coincide con **RM** quando le scadenze relative sono proporzionali ai periodi dei task, allora l'algoritmo **RM** è ottimale, fra tutti gli algoritmi a priorità fissa, qualora le scadenze relative dei task siano non superiori e proporzionali ai rispettivi periodi. Quindi **RM** è anche esso un algoritmo ottimale per una certa sottoclasse di sistemi di task. Quelli in cui le scadenze relative sono proporzionali ai periodi. Un altro caso in cui fondamentalmente utilizzarlo può essere sensato. Qui c'è una differenza, nelle ipotesi, fra il corollario ed il teorema. Il teorema prevede che i task siano in fase, mentre nel corollario abbiamo lasciato cadere questa ipotesi. L'algoritmo funziona anche quando i task non sono in fase. Perché? Perché in realtà, come andremo a dimostrare nella prossima lezione, avere i task in fase è il caso peggiore

Ottimalità di  
algoritmi  
priority-driven

Marco Cesati



Schema della lezione

Validazione

Fattore di utilizzazione

Test di schedabilità

SERT'20

RS.24

possibile. Ma di questo parleremo nella prossima lezione.