Contents

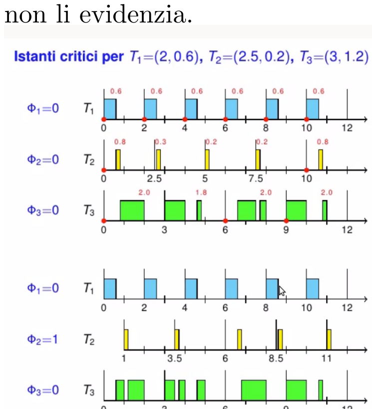
1 Schedulabilità di algoritmi a priorità fissa

Algoritmi a priorità dinamica, come EDF, sono ottimali (sotto determinate condizioni): se $\exists$ schedulazione fattibile $\Rightarrow$ anche EDF trova schedulazione.
Nessun algoritmo X a priorità fissa può avere un fatto di utilizzazione $U_X = 1$, deve per forza essere < 1.
Inoltre RM è ottimale (in senso assoluto, ovvero può raggiungere U = 1) per sistemi armonici con scadenze implicite.
In questa condizione RM è tanto buono quanto EDF.
DM è ottimale tra gli algoritmi a priorità fissa, ma non in senso assoluto: se $\exists$ algoritmo a priorità fissa che trova una schedulazione fattibile per un insieme di task, allora lo fa anche DM. Questo mi fa capire assegnare priorità fisse ai task,in modo arbitrario, non fa guadagnare nulla rispetto ad assegnarle con un parametro come la scadenza relativa. Algoritmo è altrettanto buono, se non più buono di algoritmi che fissano le scadenze in modo soggettivo, posso realizzare sistemi di task basati su parametri oggettivi e non soggettivi.
Corollario: RM è ottimale tra gli algoritmi a priorità fissa per sistemi ti task con scadenza proporzionale al periodo.
Mi pongo un problema generale: se ho sys di task generale ed un algoritmo

di schedulazione a priorità fissa, come faccio a verificare il sistema, ovvero a certificare che l'algoritmo produrrà sempre una schedulazione valida?

1.1 Istanti critici

Istanti critici: suppongo che nel sys di task tutti i job abbiano un tempo di risposta piccolo, ovvero ogni job termina prima del rilascio del job successivo del task $\Rightarrow$ ogni job viene rilasciato in un periodo e si conclude entro quel periodo (job potrebbe non rispettare la scadenza, se questa è minore del periodo).
L'istante critico è il momento in cui il rilascio del job comporta il massimo tempo di risposta possibile per quel job.
Se almeno un job T_i non rispetta la scadenza relativa, l'istante critico è un momento in cui il rilascio di un job provoca il mancato rispetto della scadenza di quel job.
Io voglio verificare che tutti i job rispettino le scadenze, sottigliezza della definizione è irrilevante dal punto di vista critico.
Teorema: Se ho un sistema di task a priorità fissa e tempi di risposta piccoli, l'istante in cui uno dei job di T_i viene rilasciato contemporaneamente ai job di tutti i task con priorità maggiore di T_i è l'istante critico di T_i.
Teorema non da condizione necessaria e sufficiente, ma solo sufficiente: se capita tale condizione $\Rightarrow$ ho un istante critico, ma potrei averne altri.
esempio : T_1=(2, 0.6) T_2=(2.5,0.2), T_3=(3,1.2).
T_1 ha la priorità massima: tutti i multipli di 2 sono istanti critici.
T_2 ha istanti critici 0 e 10, che sono anche i momenti in cui rilascio job di T_1, non c'è nessun altro momento in cui c'è rilascio contemporaneo di job di T_1 e T_2.
T_3 avrà rilasci in 0, 3, 6, 9: in 0 ho istante critico, 6 e 9 sono critici ma il thm non li evidenzia.



Stesso esempio anche se i task non sono in fase: 6 è istante critico, è descritto dal teorema.
Quando c'è rilascio in fase, siccome priorità è fissa, la schedulazione prodotta risulta identica a qualsiasi schedulazione non in fase $\Rightarrow$ mi interessa ricondurmi a quando tutti i task sono in fase.

1.2 Schedulabilità per priorità fissa e tempi di risposta piccoli

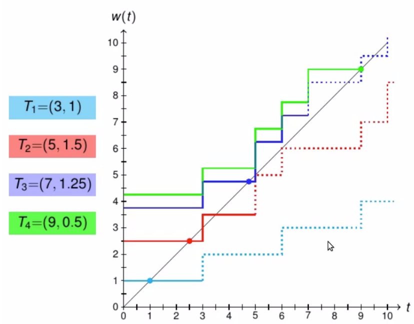
Supponiamo che in un sistema ho task a priorità fissa e tempi di risposta piccoli. Ordino i task per priorità decrescente, suppongo siano in fase all'istante t_0.
Ho i task T_1......T_i e mi chiedo il tempo necessario per eseguire tutti i job dei task T_1......T_i, nell'intervallo [t_0, t_0+t] (t $\leq p_i$:
$w_i(t) = e_i + \sum_{k=0}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k$.
Somma si estende su tutti i task di priorità superiore di T_i, devo considerarli perché portano via tempo al job di T_i. Prendo k-esimo task: a t_0 tutti i task sono in fase, quindi rilascio sicuro un job, quando ne rilascio? Prendo il ceil di $\frac{t}{p_k}$,anche job rilasciato nel periodo dopo quello considerato mi ruba tempo; moltiplico tutto per e_k, il tempo che ci metto per completare i job.
Test di schedulabilità: dati job T_1......T_i, in fase a t_0 con priorità decrescenti con T_1......T_{i-1} effettivamente schedulabili. Il task T_i può essere schedulato nell'intervallo di tempo [t_0, t_0+D] se $\exists$ t $\leq D_i$ tale che $w_i(t) \leq t$. IL mio scopo è sempre quello di verificare la schedulabilità del sistema, se ne trovo uno non schedulabile la mia analisi è finita, non ci faccio nulla col sistema di task.
Applicazione: ho T_1......T_n con priorità decrescenti.
Considero un task alla volta: $\forall$ task T_i calcolo il valore della funzione di tempo necessario $w_i(t)$ per tutti i valori t $\leq D_i$ tali per cui t è un multiplo intero di p_k per k $\in$ {1,2....,i }.Funzione $w_i(t)$ sale a gradini, devo considerare valori per cui tale funzione cambia valori.
Se per almeno uno dei valori t vale che $w_i(t) \leq$ t allora T_i è effettivamente schedulabile. Altrimenti il test fallisce, ovvero n job di T_i potrebbe mancare la scadenza, ovvero la manca sicuro se c'è un rilascio di tutti i job in fase dei task di priorità superiore e tutti quei task hanno un tempo di esecuzione pari al loro worst-case.
Possono esserci casi fortuiti, quindi in ipotesi rilassate il test non conferma schedulabilità ma scheduler riesce, però il risultato non è rilevante.
Tanto vale fermarsi e riprogettare il sistema.
esempio: T_1=(3,1), T_2=(5,1.5), T_3=(7, 1.25), T_4=(9,0,5) e considero le funzioni di tempo necessario:

Esempio: $T_1=(3,1)$, $T_2=(5,1.5)$, $T_3=(7,1.25)$, $T_4=(9,0.5)$

t	3	5	6	7	9
$w_1(t)$	1.0				
$w_2(t)$	2.5	3.5			
$w_3(t)$	3.75	4.75	6.25	7.25	
$w_4(t)$	4.25	5.25	6.75	7.75	9.0

Grafico per l'esempio precedente, ho la bisettrice del 1° quadrante, dire che $w_i(t)$ è $\leq$ t vuol dire che $w_i(t)$ sta sotto la bisettrice. La funzione è a scalini, non ha senso calcolarla, la applico nel periodo tra 0 e la fine del periodo.In T_2 la funzione sale sopra la bisettrice, ma non è importante: devo verificare che sia sotto in un certo momento, se fosse sempre sopra non sarebbe schedulabile.

Ogni volta che c'è rilascio di un task a priorità superiore $\Rightarrow$ ho gradino nella funzione di tempo necessario.



1.3 Massimo tempo di risposta

Massimo tempo di risposta W_i di T_i è il più piccolo valore prima della scadenza relativa t.c : $\mathrm{t}=\mathrm{w}_i(\mathrm{t})$. Se l'equazione non ha soluzioni $\leq$ a D_i, allora qualche job di T_i mancherà la scadenza relativa.
Uso un algoritmo:

- $t^{(}1) = \mathrm{e}_1$ in prima approssimazione
- Sostituisco nella funzione ed ottengo un nuovo valore $t^{(}k+1) = \mathrm{w}_i(t^{(}k))$
- continuo ad iterare finché:
  - $t^{(}k+1) = t^{(}k)$ e $t^{(}k) \leq \mathrm{D}_i \Rightarrow \mathrm{W}_i = t^{(}k)$
  - $t^{(}k) \geq \mathrm{D}_i$ e allora sono fuori scadenza

Ma dato che caso peggiore sono task in fase e dato che ho tutti i parametri sono noti, non sarebbe più facile provare a simulare la schedulazione? Sì, ma ci sono dei fattori che non ho considerato e che mi impediscono di simulare, esempio:

- Non è possibile determinare facilmente il worst case
- Il worst case cambia da task a task
- È difficile integrare nella simulazione altri fattori che possono essere considerati estendendo il test di schedulabilità.

In ogni caso, sia simulare il test che il test di schedulabilità stesso hanno la stessa complessità.

1.3.1 Task periodici con tempi di risposta arbitrari

Considero ora task con tempi di risposta arbitrari, che implica che:

- Un job non deve necessariamente prima che il job successivo dello stesso task sia eseguito
- è possibile che $D_i \geq$ di p_i
- Ci possono essere nello stesso istante più job di uno stesso task in attesa di essere eseguiti.
- Un job rilasciato contemporaneamente a tutti i job dei task con priorità maggiore non ha necessariamente il massimo t. di risposta possibile.

Assumo sempre che i job di uno stesso task hanno vincoli di precedenza impliciti fra di loro, ovvero sempre eseguiti FIFO.
Analizzo task per task: considero T_i (i precedenti sono schedulabili). Ho insieme task $\tau_i = T_1 .... T_i$ con priorità decrescente.
Definisco un intervallo totalmente occupato di un livello π_i un intervallo $(t_0, t_1]$ tale che:

- all'istante t_0 tutti i job di τ_i rilasciatiti prima di t_0 sono stati completati
- All'istante t_0 un job di τ_i viene rilasciato.
- L'istante t_1 è il primo istante in cui tutti i job di τ_i rilasciati a partire da t_0 sono stati completati

È possibile che in un intervallo totalmente occupato il processore sia idle o esegua task non di τ_i? No: se fosse idle, l'intervallo terminerebbe prima, non può neanche eseguire task di priorità inferiore, quindi non può eseguire task al di fuori di τ_i
esempio: T_1, T_2, T_3.
Intervalli di T_3 non sono lunghi uguale, questo perché i rilasci di T_3 non sono in concomitanza con T_1 e T_2, posso dire che l'intervallo a lunghezza massimo quando i rilasci di tutti i task sono in fase.
Test di schedulabilità generale per tempi di risposta arbitrari è ancora basato sul caso peggiore, la differenza rispetto al test per tempi piccoli è che il primo job rilasciato contemporaneamente agli altri potrebbe non avereil massimo tempo di risposta.
Idea : $\forall$ T_i analizzo tutti i suoi job eseguiti nel primo intervallo totalmente occupato di livello π_i.
Come determino l'intervallo totalmente occupato:

- Inizio determinato dal rilascio dei primi job (in fase) dei task $\tau_i = \{T_1, ..., T_i\}$
- Lunghezza massima calcolata risolvendo iterativamente t = $\sum_{k=1}^{i} \lceil \frac{t}{p_k} \rceil \cdot e_k$. Molto simile alla funzione di tempo necessario, dico che aumento t fino a

che non trovo il valore dato dalla sommatoria, ovvero il primo t per cui il lavoro necessario per compiere tutti i task permette di eseguire tutti i task rilasciati nell'intervallo [t_0, t_0+t]

Quindi si procede nel seguente modo:

- Considero i task {T_1, ..., T_i} con priorità $\pi_1 < \pi_2 .... < \pi_i$, considero un task T_i alla volta cominciando da quello con la massima priorità, ovvero T_1
- Il caso peggiore per la schedulabilità di T_i: assumere che i task τ_i = {T_1, ..., T_i} sono in fase.
- Se il primo job di tutti i task in Tau_i termina entro il primo periodo del task ⇒ decidere se T_i è schedulabile si effettua controllando se $J_{i,1}$ termina entro la scadenza tramite la funzione di tempo richiesto $w_{i,1} := w_i(t)$
- Altrimenti almeno un primo job di Tau_i termina dopo il periodo del task, calcola la lunghezza t^L dell'intervallo totalmente occupato di livello π_i che inizia da t = 0.
- Calcolo i tempi di risposta massimi di tutti i job di T_i dentro l'intervallo totalmente occupato che sono $\lceil \frac{t^L}{p_i} \rceil$; il primo l'ho già calcolato.
- Decido se questi job sono schedulabili dentro l'intervallo totalmente occupato. Uso un lemma:
  Il tempo di risposta massimo $W_{i,j}$ del j-esimo job di T_i, in un intervallo totalmente occupato di livello π_i in fase è uguale al minimo t che soddisfa l'equazione $t = w_{i,j}(t+(j-1)\cdot p_i) - (j-1)\cdot p_i$, con $w_{i,j}(t) = j\cdot e_i + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k$.
  Aggiungo un j che moltiplica e_i, devo verificare l'equazione nei punti multipli.

esercizio: $T_1 = (\phi_1,2,1,1)$, $T_2 = (\phi_2,3,1.25,4)$, $T_3 = (\phi_3,5,0.25,7)$
Parto verificando T_1:
$w_1(t) = w_{1,1}(t) = e_1 = 1 = D_1$. Quindi è sicuramente schedulabile .
T_2:
$w_{2,1}(2) = e_1 + e_2 = 2.25 > 2$, quindi non va bene. Vado avanti:
$w_{2,1}(3) = 2\cdot e_1 + e_2 = 3.25 > 3$. Non va ancora bene, proseguo:
$w_{2,1}(4) = 2\cdot e_1 + e_2 = 3.25 \leq 4 \leq D_2$ quindi T_2 è schedulabile, ma ha completato oltre il periodo ⇒ non posso più considerare tempi piccoli, devo considerare gli intervalli totalmente occupati, uso l'equazione iterativa:
$t^(1) = e_1 + e_2 = 2.25$, sostituisco nella sommatoria,ed ottengo $t^(2) = 2\cdot e_1 + e_2 = 3.25$, $t^(3) = 2\cdot e_1 + 2\cdot e_2 = 4.5$, $t^(4) = 3\cdot e_1 + 2\cdot e_2 = 5.5$, $t^(5) = 3\cdot e_1 + 3\cdot e_2 = 5.5 \Rightarrow t^(4) = t^L$, ovvero intervallo totalmente occupato di livello 2 è 5.5.
Ora calcolo quanti job di T_2 ci sono in (0, 5.5] = $\lceil \frac{t^L}{p_2} \rceil = 2$.
Veridico il secondo job di T_2:

$w_{2,2}(3) = 2 \cdot e_1 + 2 \cdot e_2 = 4.5 > 3$, no
$w_{2,2}(4) = 2 \cdot e_1 + 2 \cdot e_2 = 4.5 > 4$, ancora no.
$w_{2,2}(3) = 3 \cdot e_1 + 2 \cdot e_2 = 5.5 \leq 6 \leq p_2 + D_2 = 7$, quindi accetto il task.
Ora devo capire se posso accettare T_3, e considerare l'intervallo totalmente occupato di lvl 3:
$t^(1) = e_1 + e_2 + e_3 = 2.5$
$t^(2) = 2 \cdot e_1 + e_2 + e_3 = 3.5$
$t^(3) = 2 \cdot e_1 + 2 \cdot e_2 + e_3 = 4.75$
$t^(4) = 3 \cdot e_1 + 2 \cdot e_2 + e_3 = 5.75$
$t^(5) = 3 \cdot e_1 + 2 \cdot e_2 + 2 \cdot e_3 = 6$
$t^(6) = 3 \cdot e_1 + 2 \cdot e_2 + 2 \cdot e_3 = 6 = t^L$
job di T_3 nell'intervallo (0,6]: $\lceil \frac{t^L}{p_3} \rceil = 2$. Considero i singoli job:
$w_{3,1}(2) = e_1 + e_2 + e_3 = 2.5 > 2$, no.
$w_{3,1}(3) = 2 \cdot e_1 + e_2 + e_3 = 3.5 > 3$, no.
$w_{3,1}(4) = 2 \cdot e_1 + 2 \cdot e_2 + e_3 = 4.75 > 4$, no.
$w_{3,1}(5) = 3 \cdot e_1 + 2 \cdot e_2 + e_3 = 5.75 > 5$, no.
$w_{3,1}(6) = 3 \cdot e_1 + 2 \cdot e_2 + e_3 = 5.75 \leq 6 \leq D_3 = 7$. Posso accettare il job

$w_{3,2}(5) = 3 \cdot e_1 + 2 \cdot e_2 + 2 \cdot e_3 = 6 > 5$, no.
$w_{3,2}(6) = 3 \cdot e_1 + 2 \cdot e_2 + 2 \cdot e_3 = 6 \leq 6 \leq p_3 + D_3 = 12$ Accetto il job, e quindi il task.
Tutti i task sono schedulabili a prescindere dai loro task.

1.4 Condizioni di schedulabilità

Il test di schedulabilità generale determina se insieme di task è schedulabile o no, considerando worst case che è task in fase.
Ho dei limiti:

- Devo conoscere tutti i periodi, le scadenze ed i tempi d'esecuzione. Per validazione è necessario, ma no per implementazione di scheduler a priorità fissa. Se voglio aggiungere un task dovrei conoscere parametri che in fase di progettazione del sw non servono.
- Il risultato ottenuto non è valido se il task varia periodo, scadenza o tempo di esecuzione.
- È computazionalmente costoso, poco adatto per scheduling on-line.

Cerco di trovare delle condizioni di schedulabilità, confronto il test con la condizione, che è molto più semplice da calcolare e che può essere applicata anche se alcuni parametri non sono noti (esempio: condizione di EDF).
Mi chiedo se $\exists$ condizione di schedulabilità per algoritmi a priorità fissa:
Condizione di Liu-Layland: sistema τ di n task indipendenti ed interrompibili con scadenze relative uguali ai rispettivi periodi può essere effettivamente schedulato su un processore in accordo con RM se il suo fattore di utilizzazione

U_τ è $\leq$ a U_{RM}(n) = n$\cdot$($2^{\frac{1}{n}}$-1)
Questo è il fattore di utilizzazione di RM, se considero: $\lim_{n\to\inf}$ U_{RM}(n) = ln2, ovvero RM in generale garantisce di rispettare le scadenze pur di non caricare il processore per più del 69.3.
Ho un criterio per adottare RM negli scheduler real-time.
esempio:
T_1 = (1,0.25), T_2 = (1.25,0.1), T_3 = (1.5,0.3), T_4 = (1.75,0.07), T_5 = (2,0.1).
U_τ = 0.62 $\leq$ 0.743 = U_{RM}(5) $\Rightarrow$ è schedulabile con RM.
IL sistema T_1 = (3,1), T_2 = (5,1.5), T_3 = (7,1.25), T_4 = (9,0.5) ha fattore di utilizzazione U_τ = 0.867 > 0.757 = U_{RM}(4), forse non schedulabile.
È condizione sufficiente, difatti l'esempio 2 era quello precedente che è schedulabile se applico la funzione di tempo necessario.
L'alternativa a questo risultato è il test iperbolico: Un sistema τ di n task indipendenti ed interrompibili con scadenze relative uguali ai rispettivi periodi può essere effettivamente schedulato su un processore RM se $\prod_{k=1}^{n}(1+\frac{e_k}{p_k}) \leq 2$.
SI applica anche questo conoscendo solo fattore di utilizzazione dei task.
Correlazione con condizione di Liu-Layland: se gli n task hanno tutti lo stesso rapporto $\frac{e_k}{p_k}$ vuol dire che ciascun di questi usa una porzione uguale del processore.
Si può dimostrare che se questo è vero allora, assumendo $u_k = \frac{U_\tau}{n}$:
$\prod_{k=1}^{n}(1+\frac{e_k}{p_k}) \leq 2 \Leftrightarrow$ U_τ *leq* n$\cdot$($2^{\frac{1}{n}}$-1).
Se questo non è vero, esistono casi in cui il test iperbolico è soddisfatto, ma la condizione di Liu-Layland no; non esiste invece mai il viceversa.

1.5 Test per sottoinsiemi di task armonici

So che ,in generale RM è schedulabile se è soddisfatta condizione di Liu-Layland, ma so anche che su task armonici è ottimale. Suddivido insiemi di task in sottoinsiemi di task armonici fra loro.
Condizione di Kuo-Mok: se sistema τ di task periodici, indipendenti ed interrompibili con $p_i = D_i$ può essere partizionato in n_h sottoinsiemi disgiunti Z_1,....,Zn_h, ciascuno dei quali contiene task semplicemente periodici, allora il sistema è schedulabile con RM se:
$\sum_{k=1}^{n_h} U_{Z_k}(n_h)$ oppure se $\prod_{k=1}^{n_h}(1+U_{Z_k}) \leq 2$.
Se un sistema ha poche applicazioni molto complesse, è possibile migliorare la schedulabilità rendendo i task di ciascuna applicazione semplicemente periodici.
Esempio: 9 task con periodi 4,7 ,7 , 14, 16, 28, 32, 56, 64, fattore di utilizzazione di Liu-Layland è U_{RM} = 0.720
Considero i multipli di 2 e 7 e partizionando in due sottoinsiemi ottengo U_{Z_1} + U_{Z_2} $\leq$ U_{RM}(2) = 0.828.

Il fattore di RM è in generale U_{RM}(n), ma posso farlo diventare pari ad 1

per task semplicemente periodici.
Miglioro U_{RM}(n) considerando quanto i periodi dei task sono vicini ad essere armonici:
$X_i = \log_2 p_i - \lfloor \log_2 p_i \rfloor$ e $\zeta = \max_{1 \leq i \leq n} X_i - \min_{1 \leq i \leq n} X_i$
Considero il valore frazionario del $\log_2$ e prendo tutti i task, di cui faccio differenza tra max e min di questi scarti decimali.
Teorema: nelle ipotesi della condizione di Liu-Layland, il fattore di utilizzazione di RM dipende dal numero di task n e da ζ è: $U_{RM}(n, \zeta) =$

- $(n-1)\cdot(2^{\frac{\zeta}{(n-1)}}-1) + 2^{(1-\zeta)}-1$ se $\zeta < 1 - \frac{1}{n}$
- U_{RM}(n)

Quando si verifica il caso $\zeta = 0$? Quando $p_i = K\cdot 2^{x_i}$; non è vero il contrario
Variante: schedulabilità per scadenze arbitrarie. Se per qualche task la scadenza è più grande del periodo il limite è valido? Sì, però la formula è "pessimista": forse è possibile trovare valori di soglia superiori a U_{RM}.
Se invece per qualche task il periodo è più grande della scadenza non posso applicare Liu-Layland.
Teorema:
Un sistema τ di n task indipendenti, interrompibili e con scadenze $D_i = \delta p_i$ è schedulabile con RM se U_τ è $\leq$ a: $U_{RM}(n, \delta) =$

- $\delta(n-1)\cdot(\frac{\delta+1}{\delta}^{\frac{1}{(n-1)}} - 1)$ per $\delta = 2,3,....$.
- $n(2\delta^{\frac{1}{n}}-1) + 1 - \delta$ per $0.5 \leq \delta \leq 1$
- δ per $0 \leq \delta \leq 0.5$

2 Schedulazione di job bloccanti e job aperiodici

Avevo un modello semplice, devo rilassare qualcuna delle ipotesi dovute al fatto che i job siano sempre sempre interrompibili, o che costo di context switching sia 0. Nella pratica molti fattori rallentano l'esecuzione di un job, che possono portare a mancata scadenza. Devo tenerne conto, divido in due genti classi:

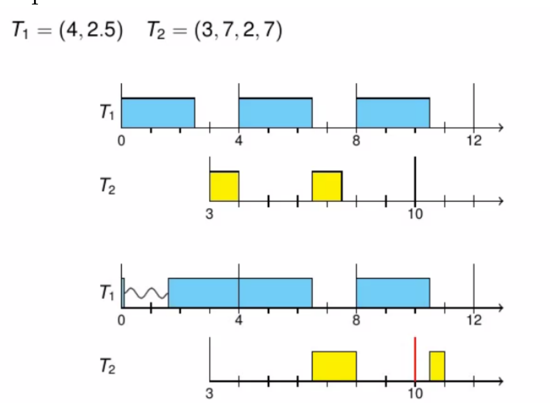
- Tempi di blocco: job non può essere eseguito nonostante il rilascio, per via di fattori esterni. Ad esempio: sul processore c'è un job non interrompibile, job rilasciato è quindi bloccato per un certo tempo. Modellati definendo b_i = tempo massimo di blocco, che tiene conto di tutti i tempi che fanno si che il job non può eseguire, va sottratto al tempo a disposizione del job.
- rallentamenti sistematici: ho calcolato il worst case di un job, ma a questo devo considerare il tempo che ci mette il job ad essere posto in esecuzione e ad essere tolto una volta completato, o anche il tempo che ci mette lo scheduler a decidere. Se questo tempo ha impatto pratico può avere senso modellarlo. Sommo al worst case del job.

2.1 Auto-sospensione

Un job rilasciato non può essere eseguito perché in attesa di eventi esterni, la cosa migliore da fare in questi casi è mettere in esecuzione un altro job. Si dice che il job si è auto-sospeso:

- job è un processo ed esegue operazione di accesso alla memoria di massa, ha senso sostituire il processo mentre questo attende i dati.
- attendo dati da rete/altri job
- attendo scadenza di un timer

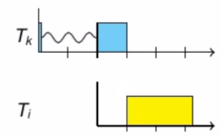
Nei SO questo tipo di operazioni sono chiamate operazioni bloccanti, nell'ambito real-time ci possono essere operazioni di auto-sospensione che però non è bloccante: in questo ambito ha senso attivo, ovvero un job ne blocca un altro. Anche in questo caso ci sono conseguenze su un altro job.
Supponiamo che ogni job di un task T_i si auto-sospende per un certo tempo x, in questo caso non appena rilasciato. Come schedulo: considero l'istante di rilascio come p_i-x, e la scadenza relativa come D_i-x.
Approccio semplificato, non funziona nel caso in cui i job si auto-sospendono solo all'inizio o per un tempo determinato, devo definire il tempo massimo di auto-sospensione $b_i(ss)$.
esempio: $T_1 = (4, 2.5)$ $T_2 = (3.7, 2, 7)$ schedulato con RM. Se primo job di T_1 si auto-sospende subito dopo il rilascio, le cose possono andare male: il primo job del task T_2 manca la scadenza, job di T_1 si risveglia in modo che per completare occupa tutto il suo periodo, quindi quando job di T_2 comincia esecuzione di porta avanti ma non riesce a finire.



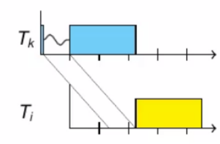
Ho impatto sui job di priorità inferiore: anche se job si auto-sospende on no lo fa, se c'è job di priorità superiore non è danneggiato, ma quelli di priorità inferiore si.

2.1.1 Rallentamento dovuto all'auto-sospensione

1° caso: il tempo di auto-sospensione di un job è maggiore della durata del job: job di T_i con priorità inferiore è rallentato al massimo per un tempo pari alla durata del job di T_k È il worst case: job T_i non riesce ad arrivare mente job di T_k è in auto-sospensione



2° caso: il tempo di auto-sospensione di un job è minore della durata del job. Un job di T_i con priorità minore è rallentato al massimo per un tempo pari alla durata dell'auto sospensione.



2.1.2 Tempo massimo di sospensione di blocco per auto-sospensione

Dato task T_k chiamo x_k il tempo massimo di sospensione di ciascun job di T_k, questo è un parametro del sistema.
Prendo task T_i con priorità minore, il rallentamento inflitto ad un job T_i da un job di T_k è minore o uguale ad x_k e minore o uguale ad e_k: $b_i(ss) = x_i + \sum_{k=1}^{i-1} min(e_k, x_k)$.
Manca qualcosa, sto assumendo che un job si auto-sospenda una volta sola, ma non c'è nessun motivo reale per cui questo sia vero: job può auto-sospendersi più volte, devo contare il numero di volte. Devo definire anche il massimo numero di volte k_i in cui un job di T_i si sospende.
Difatti:

- si può verificare un blocco da parte di un processo non interrompibile
- si ha un rallentamento dovuto allo scheduler ed al costo del context switching

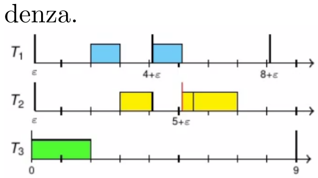
2.2 Non interrompibilità dei job

Assunzione irrealistica che i job non siano interrompibili, esistono sempre istanti in cui il job non è interrompibile:

- se sta operando su area di memoria critica
- se sta interagendo con dispositivo hardware

- job esegue syscall, e ci sono chiamate di sistema che non possono essere interrotte.Job diventa non interrompibile fino alla conclusione del SO.
- costo del context switch è troppo elevato

Un job J_i è bloccato per non interrompibilità quando è pronto per essere eseguito, ma non può perché è in esecuzione un job non interrompibile.
Quando si verifica questo fenomeno, si parla di inversione di priorità quando la priorità del job in esecuzione è minore di quella del job pronto per l'esecuzione.
esempio: $T_1 = (\epsilon, 4, 1, 4)$, $T_2 = (\epsilon, 5, 1.5, 5)$, $T_3 = (9, 2)$. Qualunque sia l'algoritmo, all'istante 0 viene messo in esecuzione job di T_3, inoltre U = 0.77 ed è schedulabile per EDF e RM, ma solo se job sono non interrompibili. Suppongo che T_3 non sia interrompibile, conclude nell'istante 2, quindi tra $[\epsilon,2]$ blocca due job con priorità maggiore. Nell'intervallo tra $[2, 5+\epsilon]$ eseguo 3 job, 2 di T_1 ed uno di T_2, ma non c'è abbastanza tempo e quindi T_2 manca la scadenza.



Come faccio a modellare che il job è non interrompibile, devo capire la durata massima di non interrombilità di un job: sia
$Theta_k$ il tempo di esecuzione massimo della più lunga sezione non interrompibile dei job di T_k. Sia b_i(np) il tempo massimo di blocco per non interrompibilità, che è tempo subito da un job a causa dei job di priorità inferiore, quando vale? $b_i(np) = \max\{\Theta_k$: per ogni task T_k di priorità minore a T_i }: suppongo che c'è job di alta priorità rilasciato, ho sul processore job di priorità inferiore T_k appena entrato nella sezione critica non interrompibile più lunga, subisco rallentamento di Θ_k, ma appena finisce la sezione lo scheduler da la priorità a me, non importa quanto solo lunghe le sezioni degli altri job: caso peggiore è che vengo rilasciato quando il job che ha il Θ_k più lungo entra in esecuzione.
Il tempo massimo di blocco totale dipende da entrambe i due tempi di blocco: $b_i = b_i(ss) + (K_i+1) \cdot b_i(np)$. Considero numero massimo di volte per cui il job J_i si sospende, il +1 è il fatto che la prima volta deve essere rilasciato sia che si auto-sospende che non.

2.3 Cambi di contesto

Come modellare rallentamenti dovuti al context switch: CS= context switch time, per ora ci metto anche tempo necessario per lo scheduler per prendere decisione.
Allungo il worst case del job: calcolato quando non c'è nulla che interferisce col job. worst case è $e'_i = e_i + 2 \cdot (K_i + 1) \cdot CS$. job deve subire almeno due cambi di contesto: quando viene messo in esecuzione e quando viene tolto

dall'esecuzione.Ma ogni volta che il job si auto-sospende c'è un altro cambio di contesto: per essere tolto e poi per essere rimesso; K_i = n° volte che il job si auto-sospende.
Alle volte non è utile modellare il context switch, però in altri casi è essenziale farlo: LST si basa sullo slack rimanente, quindi ci sono molti cambi di contesto e l'overhead è significativo ed è doveroso modellarli. Con LST è anche spesso difficile capire qual'è numero massimo di context switch di job, ma ci sono algoritmi come EDF altrettanto buoni, in un sistema real-time parametro cruciale: i job devono rispettare le scadenze; utiele vederlo in teoria ma non in pratica.

2.4 Test di schedulabilità per job bloccanti

Come faccio ad usare i parametri definiti nel processo di validazione: o uso test di schedulabilità o uso condizioni di schedulabilità.
Idea è che tempo disponibile per completare per ciascun job va diminuito del tempo massimo per cui quel job può rimanere bloccato, definisco tempo di blocco come tempo max aggiuntivo. La funzione di tempo massimo richiesto diventa:
$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \cdot e_k \rceil$ per $0 < t \leq \min(D_i, p_i)$. Ho meno tempo a disposizione per completare il job, sommo b_i.
Stesso si applica al test di schedulabilità generale:
$w_{i,j}(t) = j \cdot e_i + b_i + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \cdot e_k$ per $(j\text{-}1) \cdot p_i < t \leq w_{i,j}(t)$. non devo moltiplicare b_i per j: il 3° job di T_i è sempre $3 \cdot e_i$, ma sto cercando di capire quanto tempo rimane al 3° job, perché questo viene bloccato solo per b_i. Il blocco è qualcosa che considero soltanto quando devo studiare la schedualbilità del singolo job ed è relativa solo al singolo job. Non ha senso considerarla per tutti i task insieme, si fa sempre studio task per task.

2.5 Condizioni di schedualbilità per task bloccanti +a priorità fissa

Sia dato sistema di n task T ed un algoritmo a priorità fissa X, con fattore di utilizzazione $U_X(n)$. Sappiamo che il sistema è effettivamente schedulabile se $U_T \leq U_X(n)$, a condizione che i task non blocchino mai. Come adatto la condizione per task a priorità fissa ma che bocchino? Non posso più usare solo le condizioni di schedualbilità, perché ciascun job può bloccare con misura differente, quindi devo farlo per un task alla volta. Nel caso peggiore, ogni job di T_i impiega un tempo $e_i + b_i$ per completare l'esecuzione. Posso modellare questo tempo come tempo di esecuzione in più che il job deve subire: dato un task T_i, calcolo utilizzazione totale fino alla priorità i, dato task calcolo utilizzazione totale fino alal priorità i:
$\sum_{k=1}^{i} \frac{e_k}{p_k} + \frac{b_i}{p_i} \leq U_X(i)$. Task di priorità inferiore non possono incidere sulla priorità del task, o meglio lo faranno solo se sono non bloccanti ma lo sto già

considerando. Guardo solo ai task con priorità maggiore, considero come n° task solo fino ad i, considero solo $U_X(i)$, man mano arriverò ad $U_X(n)$. Applico anche ad EDF, considero task per task, parlo in generale di densità ed uso approccio simile al precedente: task per task questo è schedualbile se:
$\sum_{k=1}^{n} \frac{e_k}{min(D_k,p_k)} + \frac{b_i}{min(D_i,p_i)} = \Delta_\tau + \frac{b_i}{min(D_i,p_i)} \leq 1.$
Non sto parlando di task a priorità fissa, ogni job del sistema può avere priorità che precede il job in questione: di fatto, non posso applicare sommatoria solo a task a priorità superiore ma devo applicare a tutti i task del sistema, quindi arrivare alla densità del sistema. Alla densità contribuisce anche il task in questione che è $\frac{e_i}{min(D_i,p_i)}$, a cui aggiungo anche $\frac{b_i}{min(D_i,p_i)}$ poiché è come se il tempo di esecuzione del job del task è aumentato di b_i.
Problema è definire i tempi massimi di blocco se i task non hanno priorità fissata, la priorità è del job.
Teorema (Baker, 1991): in una schedualzione EDF un job con scadenza relativa D può bloccare un altro job con scadenza relativa D' solo se D > D'.
Dim: se il job con scadenza relativa D blocca quello con D', vuol dire che la sua priorità è inferiore: bloccare ha il senso che un job a priorità inferiore sta togliendo tempo ad uno a priorità superiore, quindi d > d' (scadenze assolute), per poter bloccare il processore deve averlo messo in esecuzione prima e quindi r < r' ⇒ D = d-r > d'-r' = D'. Ho una soluzione: posso ordinare i task per scadenze relative crescenti, ed applico la formula di b_i per i task con priorità fissa.
Caso dell'auto-sospensione è difficile, quindi come realizzare il teorema di Baker? Thm non è più valido: se per esempio job J' ha priorità più alta di un job J. Se J' comincia ad eseguire e si auto-sospende: prima di tornare in esecuzione comincia job di priorità più bassa. L'ipotesi che r sia < r' non è più vera, può essere dopo r' semplicemente perché il job si è autosospeso: dovrei applicare al tempo r'+tempo dopo la sospensione.
Posso applicare il ragionamento a r'+x'+e': di quanto tempo r può precedere r', sicuramente di x'+e'.Può non precedere r', ma r'+x'+e' è la massima distanza che posso avere fra r ed r'. Formulo teorema di Baker con auto-sospensione: in una schedulazione EDF, un job con scadenza relativa D può bloccare un altro job con scadenza relativa D' e tempo massimo di esecuzione x' solo se D > D'-x'-e'.
Dato che entrambi i job possono auto-sospendersi, è possibile che i due task possano bloccarsi a vicenda non ho più ordinamento totale.

2.6 Schedulazione basata su tick

Fin'ora ho visto scheduler event-driven: viene eseguito quando si verifica un evento rilevante. In pratica, è più semplice realizzare uno scheduler time-driven, ovvero che si attiva ad interruzioni periodiche: svantaggio è che tutti i tempi nel sistema avranno granularità pari alla dimensione del mio tick.
Il riconoscimento di un evento come il rilascio di un job può essere differito fino al tick successivo, è come se ci fosse inversione di priorità. Definisco job

pendenti, ovvero che sono stati rilasciati ma che lo scheduler non ha ancora preso in considerazione perché non è scattato il tick, e quelli eseguibili, ovvero quelli piazzati dallo scheduler, ho due code per le rispettivi due classi. Scheduler sposta job da coda dei job pendenti a coda dei job eseguibili.Quando jop termina, so già qual'è il prossimo da eseguire: sarà quello successivo nella coda dei job eseguibili. Se arriva job, questo viene messo nella coda dei job pendenti.

2.6.1 Test schedulabilità per priorità fissa con tick

Come posso applicare il test di schedulabilità ad uno scheduler a priorità fissa basato su tick?
Considero scheduler che si attiva con periodicità p_0, esegue in tempo e_0 il controllo della coda di job pendenti e con CS_0 trasforma un job da pendente a eseguibile.
Per controllare la schedulabilità di T_i

- Devo aggiungere task per controllare schedulabilità di un task $T_0 = (p_0, e_0)$ a priorità massima.

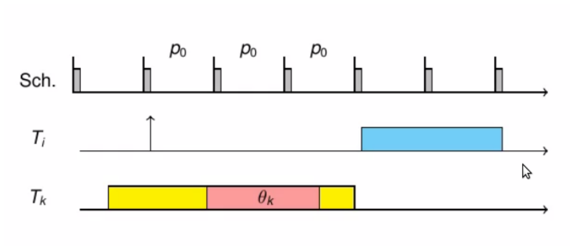
- Devo modellare il fatto che qundo arriva job, questo va prima o poi trasformato da pendente ad eseguibile.

  - per tutti i task a priorità inferiore rispetto ai job di T_i, per cui devo tenere conto del fatto che lo scheduler interverrà e trasformerà il job pendente in un job nella coda eseguibile. Oltre ai job di priorità inferiore, che vanno da i+1 a n, aggiungo un numero corrispondete di task $T_{0,k} = (p_k, CS_0)$ per ogni k = i+1,..,n, con priorità maggiore di T_1, ma che hanno periodicità CS_0, ovvero il tempo che ci mette il processore a trasformare i job in eseguibile da pendenti.

  - job a priorità superiore, aggiungo a tutti i task di priorità superiore o uguale ad $(K_k + 1) \cdot CS_0$, considero K_k perché ogni volta che mi risveglio devo essere spostato da pendente ad eseguibile. Aggiungo questi valori ad e_k per ogni k = 1,2...,i.

  Perché non considero le auto-sospensione per i task a priorità inferiore? Perché task inferiore non viene mai eseguito al posto mio, pago solo il primo rilascio, perché fin quando io sono eseguibile, quelli con meno priorità di me non hanno possibilità di essere eseguiti prima di me.

- Devo anche considerare il tempo di blocco per non-interrompibilità, anche se tutti i miei job sono sempre non interrompibili. $b_i(np) = (\lceil max_{i+1 \leq k \leq n} \frac{\Theta_k}{p_0} \rceil + 1) \cdot p_0$. max Θ_k moltiplicato il p_0 diventa il max di tutti i Θ_k di priorità inferiore, in più c'è un p_0. esempio:



ho lo shceduler che viene invocato con periodo p_0. Ad un certo istante viene rilasciato il job del task T_i, mi metto nel worst case, ovvero il job T_i arriva un infinitesimo dopo che lo scheduler ha finito di controllare la coda dei job pendenti, quindi fino al prossimo p_0 non potrò eseguire il job. In questo periodo, prima che possa intervenire lo shceduler, si continua ad eseguire un job di priorità inferiore di T_k e questo job entra nella regione interrompibile all'interno del periodo p_0 in cui è arrivato T_i, task T_k continua l'esecuzione per un numero di periodi pari a $\frac{\Theta_k}{p_0}$. Solo quando scheduler interviene si rende conto che job di T_k è diventato interrompibile e può entrare T_1, e c'è la parte intera superiore perché se il n° di periodi non è intero,anche la frazione non completata porta via tempo e devo aspettare comunque il periodo successivo; il +1 è il rilascio, almeno un periodo lo devo aspettare anche se non ho sezione critica, il job è arrivato prima.

esempio: $T_1 = (0.1, 4, 1, 4.5)$, $T_2 = (0.1, 5, 1.8, 7.5)$, $T_3 = (0, 20, 5, 19.5)$ non interrompibile in $[r_3, r_3+1.1]$. Scheduler ha $p_0 = 1$, $e_0 = 0.05$, $CS_0 = 0.06$.
Faccio analisi dei singoli task:
Verifico T_1: sistema equivalente è $T_0 = 1{,}0.05)$, $T_{0,2} = (5{,}0.06)$, $T_{0,3} = (20{,}0.06)$, $T_1 = (4{,}1.06)$, $b_1 = 3$.
$w_1(t) = 1.06 + \lceil\frac{t}{1}\rceil 0.05 + \lceil\frac{t}{5}\rceil 0.06 + \lceil\frac{t}{20}\rceil 0.06$.
$w_1(4.06) = 4.43 \leq w_1(4.43) \leq 4.5$, quindi ok.
Procedo per T_2 e T_3 sempre considerando il sistema equivalente.
$w_2(4.86) = 7.29 \leq w_2(7.29) \leq 7.5$, quindi ok.
$w_3(6.06) = 12.25$, $w_3(12.25) = 16.53$, $w_3(16.53) = 19.65$, $w_1(19.65) = 19.8 > 19.5$, quindi no.
Devo concludere che il sistema non è validato, e va ri-progettato.

2.6.2 Condizione di schedulabilità su tick

Per ciascun T_i da controllare faccio quanto segue:

- Uso il thm Baker ed ordino per scadenze relative crescenti
- aggiungo un task $T_0 = (p_0, e_0)$ di massima priorità
- Aggiungo $(K_k+1)\cdot CS_0$ al tempo i esecuzione e_k, devo farlo per tutti i task: i blocchi hanno una certa relazione ma non ho priorità fissate, quindi ogni job può portare via tempo ad un altro job nel sistema

- Tempo di blocco è $b_i(np) = (\lceil max_{i+1 \leq k \leq n} \frac{\Theta_k}{p_0} \rceil + 1) \cdot p_0$.

Nell'esempio di prima, ottengo densità totale $\Delta \simeq 0.95$., verifico T_1: prendo Δ e sommo $\frac{3}{4}$, ovvero tempo di blocco diviso periodo di T_1. Ottengo 1.69 >, quindi non schedualbile. Mi posso fermare: basta trovare un task non schedulabile.

2.7 Schedulazione priority-driven di job aperiodici

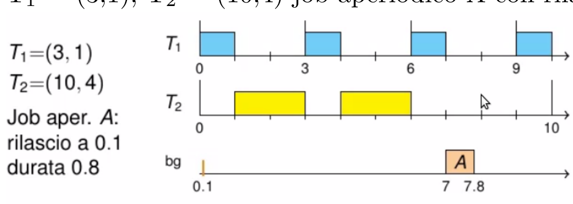
Mi pongo il problema di dover gestire job che arrivano ad istanti di tempo non predicibili:

- Job aperiodici sfot-rt: non faccio nulla, voglio però che completino nel miro tempo possibile.
- Job aperiodici hard-rt: tempi di arrivo sconosciuti, durata sconosciuta e scadenze hard.

Se non ho nessuna ipotesi su tempi di arrivo ed esecuzioni non posso prendere impegni: potrà sempre arrivare qualcosa che non mi permette di rispettare le scadenze.
Richiedono algoritmi differenti, però devono essere corretti ed ottimali: le scadenze vanno rispettate, i job aperiodici hard-rt va rifiutato se non è possibile garantirne le scadenze.Inoltre: tempi di risposta dei job soft-rt non hanno scadenze ma vanno minimizzato i tempi di risposta.

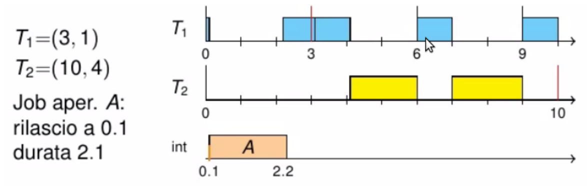
2.7.1 Schedulazione di job aperiodici soft RT in background

Metto in coda apposta e quando job è in background eseguo il job aperiodico in cima alla coda. Algoritmo è corretto, i task periodici non sono influenzati, ma non è ottimale: ritardo job aperiodici senza motivo.esempio:
$T_1 = (3,1)$, $T_2 = (10,4)$ job aperiodico A con rilascio 0.1 e durata 0.8.



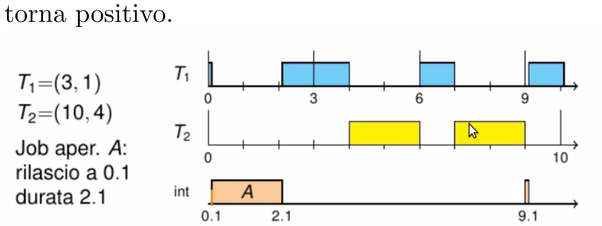
2.7.2 Schedulazione di job aperiodici soft RT interrupt-driven

Esegui job aperiodici nel mo,mento in cui li rilasci, algortimo non è corretto ma è ottimo: job aperiodici finiscono nei tempi minimi, ma task periodici possono mancare le scadenze.



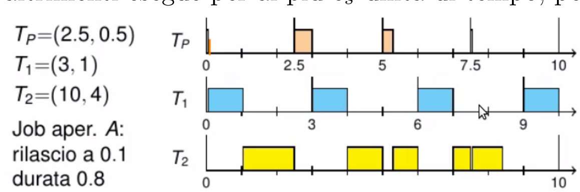
2.7.3 Schedualzione di job aperiodici soft RT con slack stealing

Algoritmo esegue i job aperiodici in anticipo rispetto a quelli periodici finché c'è uno slack globale positivo.
È corretto perché i job periodici non perdono le scadenze. È ottimale, solo per job aperiodico in cima alla coda. Svantaggio è che tenere uno slack globale in uno scheduler priority divern è difficile. Job aperiodico riprende quando lo slack torna positivo.



2.7.4 Schedulazione di job aperiodici soft RT con polling

Algoritmo basato su polling, ovvero nel sys di task periodici introduco server di polling o poller, a cui do un certo periodo p_s e tempo di esecuzione e_s e gli do priorità massima,così da ridurre i tempi di risposta dei job aperiodici. Server controlla coda job aperiodici, se vuota si auto-sospende fino a prossimo tick, altrimenti esegue per al più e_s unità di tempo, per poi auto-sospendersi.



È corretto se dimensiono il poller in modo tale che il suo fattore di utilizzazione non ecceda quello dell'algoritmo di schedulazione, è come aggiungere un task periodico che ha worst case pari ad e_s.
Non è ottimale, job aperiodico può arrivare subito dopo esecuzione del poller. Nell'esempio forse potevo anticipare l'esecuzione del job A senza far mancare le scadenze. Posso migliorare le capacità del server? Sì.

2.7.5 Server periodici

I server periodici sono una classe di task periodici aventi:

- periodo p_s
- budget e_s
- dimensione $u_s = \frac{e_s}{p_s}$

Hanno due regole:

- regola di consumo che dice come il budget viene consumato
- regola di rifornimento: come il budget viene ripristinato

Server di dice impegnato quando ha del lavoro da svolgere, ovvero la coda dei job aperiodici non è vuota, idle nel caso contrario. È eleggibile, pronto o schedulabile se è impegnato ed il suo budget è > 0. Il poller può essere descritto come server periodico, è impegnato se la coda dei job non è vuota, la regola di consumo è che sottrae il tempo impiegato ad eseguire il job aperiodico dal budget; le coda dei job aperiodici è vuota il budget viene azzerato.
Budget rifornito del suo valore massimo e_s all'inizio di ogni periodo p_s.

2.8 Algoritmi a conservazione di banda

Problema del server di polling è che se si svuota la coda, il server ha budget azzerato. Se subito dopo arriva un job che potrebbe essere subito dal server non può farlo: questo comporta ritardo di esecuzione. VOlgio un algoritmo in cui un budget non venga azzerato se la coda è vuota, in modo da migliorare tempi di risposta se arrivano job aperiodici. Esistono molti algoritmi a conservazione di banda.3 di cui parleremo:

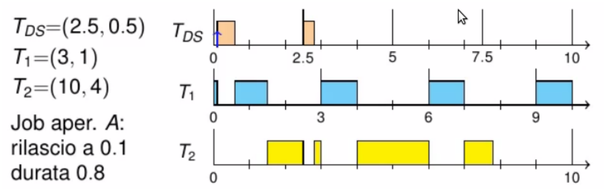
- Server procrastinabile
- Server sporadico
- Server a utilizzazione costante

Sono già molto sofisticati, usarne di più sofisticati comporterebbe overhead computazionale elevato. Questi algoritmi usati anche altrove, es gestione code di pacchetti.

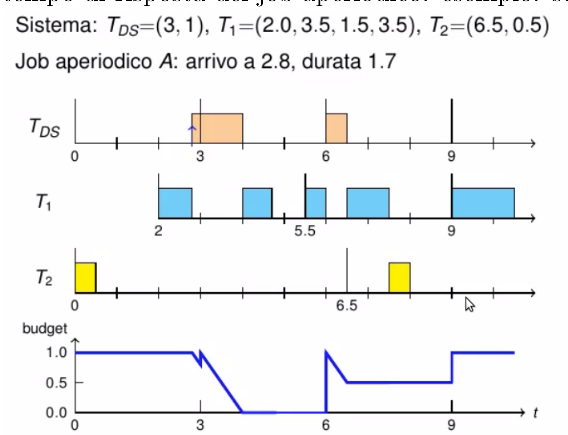
2.9 Server procrastinabile

Server più semplice possibile. Devo definire le regole di consumo e riferimento, premesso che ho consumo p_s e budget p_s. Regola di consumo: budget è decrementato di 1 per ogni unità di tempo in cui il server è in esecuzione (è proporzionale, quindi se è meno di 1 unità perdo meno di 1 unità). Non viene azzerato il budget se la coda dei job aperiodici è vuota.

Regola di rifornimento: ad ogni istante multiplo di p_s, il budget è impostato ad e_s. Nota bene: il budget non si accumula, quello che non spendo viene perso.



stesso esempio di prima, all'istante 0 coda è vuota e quindi scheduler seleziona qualcun altro, quando a 0.1 arriva job aperiodico, questo interrompe job di T_1 ed esegue per 0.5 che è il budget massimo. Questo tipo di server minimizza tempo di risposta del job aperiodico. esempio: schedulazione a priorità fissa



.

All'inizio server non ha nulla da fare, va in esecuzione job di T_2, le cose cambiano quando all'istante 2.8 arriva il job aperiodico di durata 1.7, server ha priorità superiore, budget è positivo quindi è eleggibile e quindi esegue fino a 3. All'istante 3 cade il periodo del server, quindi budget è rifornito ad 1 ed esegue fino anche non finisce.Va eseguito altro, rimetto dentro il job di T_1, dopo di che non succede nulla: processore è idle anche se avrebbe qualcosa da fare, job aperiodico non è finito, ma algoritmo continua a non schedulare job. All'istante 6 termino job aperiodico e tutto procede. È vero che l'algoritmo cerca di anticipare esecuzione dei job aperiodici, ma non è perfetto. Non è ad esempio come un slack stealing, ho dei limiti: intervallo in cui processore è idle.
Posso fare la stessa schedulazione con EDF, in questo caso server non ha la priorità più grande: la priorità è data dalla scadenza assoluta che è data implicitamente dai periodi del server.Schedulazione simile a quella di prima, ci anche sono degli istanti in cui il processore è idle.
È possibile applicare condizioni di schedulabilità a sistemi a priorità fissa con server procrastinabile? La difficoltà è che il caso peggiore non è più vero, perché il server procrastinabile non ha un comportamento simile agli altri task periodici. Se il server è eleggibile e nessun task di priorità maggiore è in esecuzione viene subito eseguito, ma qui non appena arriva job aperiodico task diviene

eleggibile, ma non so in che istante arriva il job: server con budget > 0 può diventare eleggibile in qualunque momento. esempio: T_{DS}=(3,1.2), T_1=(3.5,1.5), $r_{1,c}$=10, r_A=10, $e_A > 3$, budget(10)=1.2, fase=2.2
All'istante 10 viene rilasciato job di T_1, ma all'istante 10 sto anche eseguendo job molto lungo aperiodico, questo job viene eseguito fino allo scadere del budget. Caso peggiore vuole che a 11.2 c'è scadenza del periodo del server, quindi budget è nuovamente incrementato, quindi esegue per un altro tempo e si azzera a 12.4, ma a 12.4 non c'è più abbastanza tempo per eseguire job di T_1. Assunzione non più vera: job arriva in un qualsiasi momento, configurazione è tale per cui job continua ad eseguire per un tempo più lungo di quello che doveva ed intacca job regolare.

2.9.1 Istanti critici per server procrastinabili

Sistema di task periodici indipendenti e interrompibili, e priorità fissa con $D_i \leq p_i$ ed un server procrastinabile (p_s, e_s) con priorità massima, un istante critico di un task T_i si verifica all'istante t_0 se

- a t_0 è rilasciato un job di tutti i task T_1,...,T_i
- a t_0 il budget del server è e_s
- a t_0 è rilasciato almeno un job aperiodico che impegna il server da t_0 in avanti
- l'inizio del successivo periodo del server è $t_0 + e_s$

Nelle ipotesi del lemma, quanto tempo di processore occupa al massimo il server nell'intervallo $(t_0, t]$. Devo aggiungere questo tempo alla funzione di tempo necessario di T_i. Tempo totale: devo considerare anche il periodo troncato prima di t perché il server ha priorità massima, ho sicuro un e_s, poi ho e_s per il numero di intervalli: $e_s + \lceil \frac{t-t_0-e_s}{p_s} \rceil \cdot e_s$.
Ora posso modificare la funzione di tempo necessario per tenere conto del server procrastinabile:
$w_i(t) = e_i + b_i + e_s + \lceil \frac{t-t_0-e_s}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k$ per $0 < t \leq p_i$.
Il test controlla se $w_i(t) \leq t$ per i valori di $t \leq D_i$ tali che $t = h \cdot p_k$ oppure $t = e_s + h \cdot p_s$, oppure $t = D_i$(h=0,1,...)
Stesso avviene per il test di schedulabilità generale:
$j \cdot e_i + b_i + e_s + \lceil \frac{t-t_0-e_s}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k$ per $0 < t \leq p_i$ per $(j-1) \cdot p_i < t < w_{i,j}(t)$.
L'esempio nel caso precedente mostra che il task T_1 non è schedulabile.
Posso anche realizzare un sistema in cui server non ha priorità massima, condizione mi da risultato pessimista ma la condizione è solo sufficiente (può dare falsi negativi).

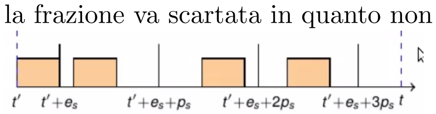
2.9.2 Condizione di schedulabilità RM con server procrastinabile

Teorema: un server procrastinabile con periodo p_s e budget e_s ed n task periodici indipendenti ed interrompibili , con $p_i = D_i$ e tali che:
$p_s < p_1 <... < p_n < 2p_s$ e $p_n > p_s + e_s$ sono schedulabili con RM se l'utilizzazione totale è:
$U_{RM/DS}(n) = \frac{e_s}{p_s} + [(\frac{e_s+2p_s}{p_s+2e_s})^{\frac{1}{n}}$ -1]. Molto simile alla formula di Liu-Layland, facile verificare che se $e_s = 0$, ottengo esattamente $U_{RM}(n)$, ma ora $\lim_{x \to inf}$ $U_{RM/DS}(n) = \frac{e_s}{p_s} + \ln(\frac{e_s+2p_s}{p_s+2e_s})$.
Mi dice quanto è il carico massimo che posso dare al sistema quando c'è server procrastinabile in modo da garantire le scadenze, valgono però le ipotesi molto forti.
Se le condizioni non si verificano, bisogna effettuare analisi task per task:

- Server non ha alcuna influenza sui task con periodo minore di p_s
- Server è schedulabile se lo è il corrispondente task periodico
- Per i task di priorità inferiore devo prevedere che il server può bloccare per un tempo e_s in più, aggiungo alla formula il tempo di blocco aggiuntivo:
  $\sum_{k=1}^{i-1} \frac{e_k}{p_k} + \frac{e_s}{p_s} + \frac{e_s+b_i}{p_i} \leq U_{RM}(i+1)$.
  Aggiungo anche ritardo e_s in più che è il primo intervallo, il ritardo dovuto al fatto che nell'istante critico per T_1 vengo rallento di un istante in più rispetto al tempo normale. Confronto al limite di Liu-Lyland per i+1 task perché includo anche il server.

2.9.3 Condizione di schedulabilità di EDF con server procrastinabile

Un task periodico T_i in un sistema di n task indipendenti ed interrompibili è schedulabile con EDF insieme ad un server procrastinabile (p_s, e_s) se:
$\sum_{k=1}^{n} \frac{e_k}{min(D_k, p_k)} + \frac{e_s}{p_s} \cdot (1 + \frac{p_s - e_s}{D_i}) \leq 1$.
Dim: per $D_k \leq p_k$. Suppongo che un job di T_i rilasciato a r_i manca la scadenza a t; t' < t è l'ultimo istante in cui il processore è idle o esegue un job a priorità inferiore. Ma allora $r_i \geq t'$, questo significa che l'istante t in cui manco la scadenza assoluta meno t' è maggiore o uguale alla scadenza relativa. Ma quindi, invertendo: $\frac{1}{t-t'} \leq \frac{1}{D_i}$. Quando tempo viene rubato dal server procrastinabile tra t' e t: $e_s + \lfloor \frac{t-t'-e_s}{p_s} \rfloor \cdot e_s$, la parte intera è inferiore perché se t cade nel mezzo vuol dire che il server procrastinabile avrà una scadenza che sarà dopo, la frazione va scartata in quanto non ruberà tempo.



Quindi t-t' < $\sum_{k=1}^{n} \frac{e_k}{p_k} \cdot (t - t') + \frac{e_s}{p_s} \cdot$ (t-t'-p_s-e_s). L'intervallo non è sufficiente a fare tutto il lavoro: il lavoro è eseguire tutti i job periodici, in più il tempo del server procrastinabile nell'intervallo t-t' ed in più +p_s - e_s, togliendo la parte intera. Ma sto dicendo che la sommatoria + il resto è > 1, ovvero se la condizione è ≤ 1 il task periodico rispetterà la scadenza.

2.10 Server sporadici

Un server procrastinabile può ritardare i task di priorità minore più di un task periodico con identici parametri. Vorrei avere un server con impatto su schedulabilità del sistema uguale a quello di un qualsiasi task periodico: server sporadico, shcedulabilità del sistema si studia semplicemente, ne esistono vari tipi: differenza sta nelle regole di consumo/rifornimento.

2.10.1 Server sporadici in sistemi a priorità fissa

Sistema di task periodici T a priorità fissa, ho un server sporadico T_s=(p_s, e_s), con priorità π_s. Definisco l'insieme T_H, che è l'insieme dei task con priorità maggiore di π_s.
Definisco l'intervallo totalmente occupato di un insieme di task:

- prima dell'intervallo tutti i job sono stati completati
- all'inizio dell'intervallo viene rilasciato almeno un job
- la fine dell'intervallo è il primo istante in cui tutti i job rilasciati entro l'intervallo sono completati

Definisco alcuni parametri e variabili:

- t_r: l'ultimo istante in cui è stato aumentato il budget, ovvero l'ultimo istante in cui è stata applicata la regola di rifornimento del server.
- t_f, che è il primo istante dopo t_r in cui il server è in esecuzione.
- t_e è invece una variabile che serve per indicare quando ci sarò il prossimo rifornimento, tipicamente sarà t_e + π_s.
- BEGIN: variabile che per ogni t, considera l'ultima sequenza di intervalli totalmente occupati contigui dei task T_H iniziata prima di t. BEGIN è esattamente l'inizio del primo intervallo totalmente questa sequenza.
- END è la fine, ma solo se la fine è precedente a t, altrimenti è ∞.

2.10.2 Server sporadico semplice

Regola di consumo: in ogni istante maggiore di t_r, il budget è decrementato di una unità per ogni unità di tempo se una delle seguenti condizioni è vera:

1. Il server è in esecuzione
2. Il server è stato in esecuzione dopo t_r ed inoltre END $<$ t.

Se le condizioni sono false, il budget si conserva. Il server consuma budget più in fretta del server procrastinabile, stiamo cercando di ridurre l'impatto che ha sui task di priorità inferiore.
Regole di rifornimento:

1. Ogni volta che faccio rifornimento, il budget è settato ad e_s, t_r viene associato all'istante corrente

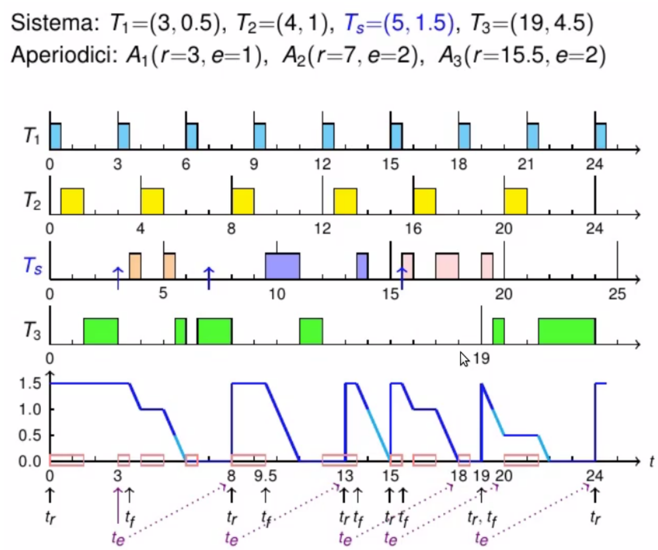
2. All'istante t_f:

   - se END = t_f, allora associa a t_e il max(t_r, BEGIN)
   - se END < t_f, associa a t_e il valore di t_f.

3. Il prossimo rifornimento sarà a t_e+p_s, ma con due eccezioni:

   - se t_e+p_s < t_f, il budget sarà rifornito non appena esaurito
   - il budget sarà rifornito ad un certo momento t_b < t_e+p_s se esiste un intervallo [t_i, t_b) in cui nessun task di T è eseguibile, ed un task di T comincia l'esecuzione a t_b

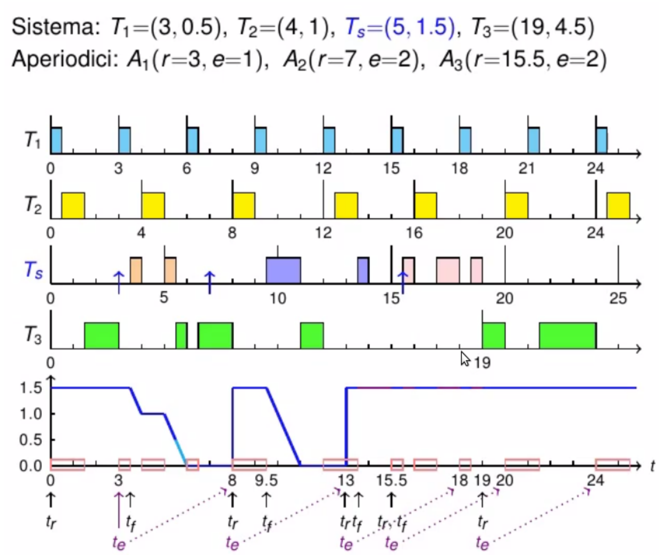
Significato della regola di consumo 1: nessun job del server esegue per un tempo maggiore di e_s in un periodo p_s
Significato di C2: il server conserva budget se un task di T_H è eseguile oppure il server non ha mai eseguito t_r; altrimenti il budget è consumato.
Significato di R2: se nell'intervallo (t_r, t_f) sono stati sempre in esecuzione task di T_H, il prossimo rifornimento sarà a t_r+p_s. Ma se questo non è vero il prossimo rifornimento sarà a t_e+p_s dove t_e è l'ultimo istante di (t_r, t_f] in cui non esegue un task di T_H.
Significato di R3a: il job del server ha atteso per più di p_s unità di tempo prima di iniziare l'esecuzione, quindi il job continua nel prossimo periodo (serve il test di schedulabilità generale).
R3b: il budget è rifornito nell'ostante iniziale di ogni intervallo totalmente occupato di T.
esempio di schedulazione RM con server sporadico semplice:



Da 3.5 comincio T$_f$ e consumo il budget, ma ora devo anche capire t$_e$, che qui è 3. Questo mi dice anche quando sarà il prossimo rifornimento, che sarà ad 8 (3+5). All'istante 5.5 termina il job aperiodico, e qui lo scheduler da il controllo al job di T$_3$, ma il budget continua ad essere consumato fino a diventare 0: non sono più nelle condizioni in cui il budget si preserva, e qui sta eseguendo un job con priorità minore del server. All'istante 7 arriva il 2° job aperiodico ma non posso eseguirlo perché il budget è 0, quindi devo aspettare 8. A 9.5 termina l'intervallo totalmente occupato, quindi posso eseguire job aperiodici.

2.10.3 Server sporadico background

Variante del server sporadico (ne esistono di verse via via sempre più costose da implementare). Ogni volta che nessun job periodico è eseguibile, il server esegue un job aperiodico.
Regola di consumo è identica a quella del server sporadico semplice, tranne che se nessun task periodico è eseguibile il budget è uguale a e$_s$.
regola di consumo è uguale tranne che per R3b: il budget è ripristinato all'inizio di ogni intervallo in cui nessun task periodico è eseguibile; t$_r$ (ed eventualmente t$_f$) è la fine dell'intervallo.
Conviene sempre implementare questo server, perché questo tende ad abbassare il tempi di risposta dei job aperiodici, l'unico caso in cui non conviene usarlo è quando si utilizzano più server sporadici per differenti tipi di job aperiodici.
esempio precedente:



Da un certo punto in poi budget rimane sempre al valore massimo, per un motivo o per un altro.

2.11 Constant Bandwidth server

Server inventato da L. Abeni e G. Buttazzo (1998). Server abbastanza recente, importante per diversi motivi:

- Server abbastanza facile da integrare in uno scheduler a priorità fissa a livello di job
- Schedulazione di job aperiodici con i vantaggi di EDF rispetto a RM/DM
- server è work conserving: non lascio mai processore idle se c'è almeno un job da eseguire.
- occupazione del processore non supera mai una frazione di tempo predefinita: permette di isolare il comportamento del server dal comportamento dei task aperiodici

Caratteristiche:

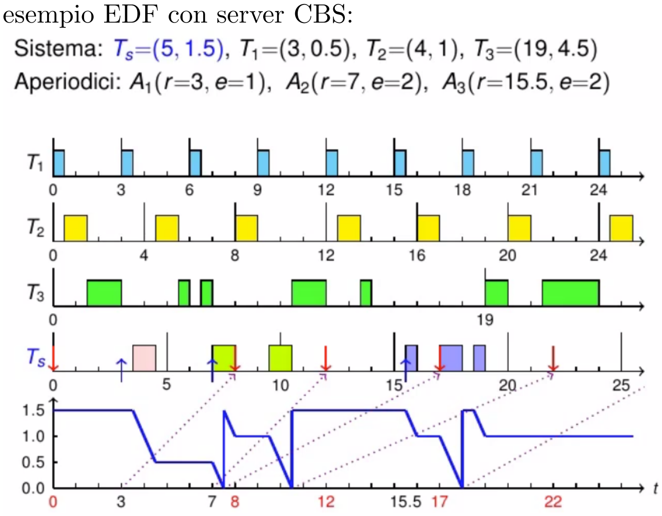
- periodo p_s
- budget massimo e_s
- budget corrente c_s
- scadenza assoluta d_s: questo perché il server va schedulato in un algoritmo di tipo EDF, devo confrontare la priorità con quella degli altri task che è basato su scadenza assoluta.

Il rapporto u$_s$ = $\frac{e_s}{p_s}$ è la bandwidth del server.
Il server CBS viene schedulato con EDF insieme agli altri task periodici considerando la scadenza assoluta corrente d$_s$.
Un sistema di task periodici ed un server CBS sono schedulabili con EDF se U$_T$ + u$_s$ $\leq$ 1.
Funzionamento del server:

- Regola di aggiornamento della scadenza:
  - inizialmente d$_s$ = 0
  - non appena budget corrente si azzera, la scadenza diviene pari a d$_s$+p$_s$, la priorità viene diminuita in modo da dare spazio agli altri task del sistema
  - Se ad un certi istante t viene rilasciato job aperiodico ed il server non è impegnato (la coda dei job aperiodici è vuota) , vado a verificare se c$_s$ $\geq$ (d$_s$-t) $\cdot$ u$_s$, perché se questo avviene rischio di prendere più tempo del processore del dovuto e quindi setto d$_s$ a t+p$_s$.
- Regola di rifornimento e di consumo:
  - all'inizio imposto il valore di c$_s$ ad e$_s$
  - c$_s$ viene decrementato proporzionalmente all'esecuzione dei job periodici del server.
  - se c$_s$ di azzera, c$_s$ viene rifornito ad e$_s$ immediatamente

  Non esiste mai un intervallo di tempo > 0 in cui il server ha un budget nullo.

esempio EDF con server CBS:



intervalli	job attivi	densità
(0, 0.5]	J_1	0.5
(0.5,1]	J_1J_2	1.0
(1,2]	$J_1J_2J_3$	1.5
(2,2.5]	J_2J_3	1.0
(2.5,3]	J_3	0.5

2.12 Schedulabilità di job aperiodici hard real-time

Concetto di densità del job aperiodico con istante di rilascio r e tempo massimo di esecuzione e e scadenza la sua densità è: $\frac{e}{d-r}$.
Vale questo teorema: Un sistema di job aperiodici, indipendenti e interrompibili è schedulabile con EDF se la densità totale di tutti i job attivi (nell'intervallo tra rilascio e scadenza) è in ogni istante ≤ 1.
In ogni istante di tempo la densità totale di tutti i job rilasciati e non ancora conclusi deve essere ≤ 1; è una condizione sufficiente, teorema permette di realizzare anche un test di accettazione. Dim: un job manca la scadenza a t, t' è l'ultimo momento in cui il processore non ha eseguito un job con scadenza $\leq$ t $\Rightarrow \sum_i e_i >$ t-t'. Partiziono l'intervallo fra (t',t] in $(t_1,t_2],(t_2,t_3]$.... dove t_k è l'istante di rilascio o scadenza per qualche job, in ciascuno dei quali l'insieme dei job attivi è differenze. Considero X_k l'insieme dei job attivi in $(t_k,t_{k+1}]$ e Δ_k la loro densità:
$\sum_i e_i = \sum_{j=1}^{l}(t_{j+1}-t_j) \cdot \sum_{J_k \in X_i} \frac{e_k}{d_k-r_k} \leq \sum_{j=1}^{l} \Delta_j(t_{j+1}-t_j) \leq$ t-t'. Vedo la somma di e_i come la densità per quell'intervallo moltiplicata per la lunghezza dell'intervallo. Ma il risultato è in contraddizione col fatto che qualcuno ha mancato la scadenza. esempio:
Considero job aperiodici J_1=(r=0, e=1,d=2), J_2=(r=0.5, e=1,d=2.5), J_3=(r=1, e=1,d=3)
Nell'intervallo (1,2] la densità totale è > 1, quindi il teorema non si applica. Schedulabilità con EDF? Sì, il teorema è solo sufficiente: metto J_1 in (0,1], J_2 in (1,2] e J_3 in (2,3] ed ottengo la mia schedulazione.

3 Controllo d'accesso alle risorse condivise

Sono partito da modello di carico nel sistema in cui tutti i job erano semplificati, task rilasciavano i job in maniera regolare e tutti i job erano indipendenti ed interrompibili. Mano a mano rilassato queste ipotesi, estendendo il modello. Continuo ad avere singolo processore, sciolgo vincolo di indipendenza dei job nel senso delle risorse condivise. Risorse condivise: accedervi significa vietare a qualunque altro job l'accesso finché il lavoro non è concluso.
Nel modello dico che esistono una serie di risorse riciclabili R_1, R_2,...., R_ρ e ciascuna risorsa R_i ha ν_i unità di risorsa indistinguibili assegnabili, non posso assegnare la stessa unità di risorsa a più job ma più job può acquisire più unità

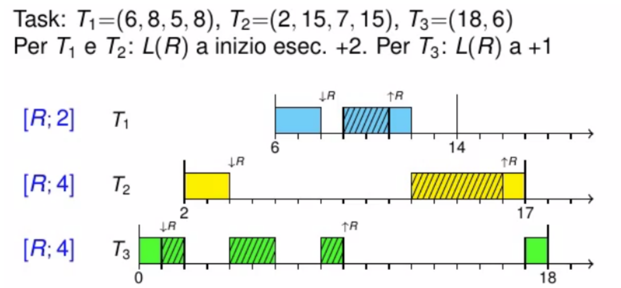
di risorsa.
Se R_i ha un numero ∞ unità di risorsa non vale la pena considerarla nel modello, considero quindi ν_i sempre finito.
esempi: semafori, mutex, spin lock, stampanti erc..., si parla di risorse passive: l'unica cosa che conta è che siano disponibili, non sono importanti le loro caratteristiche interne.
Come modello una risorsa R che può essere utilizzata da un numero finito di job $n > 1$: R ha ν unità esclusive, ovvero nessun job può ottenere più di 1 unità.
Come modello invece risorsa R che ha una intrinseca dimensione finita (es una memoria): capisco qual'è l'unità di assegnazione della memoria, ad esempio un pagina di memoria, e faccio corrispondere a ν il più piccolo blocco di risorsa assegnabile.

3.1 Richieste e rilasci di risorse

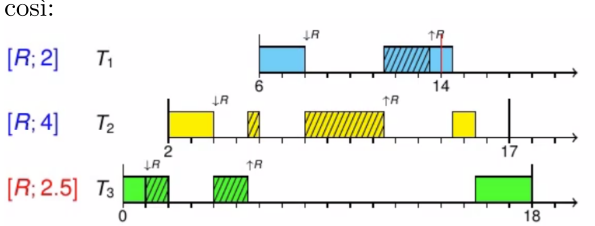
Un jbo che deve acquisire un certo n° η di unità della risorsa R_i procede ad effettuare la richiesta $L(R_i, \eta)$. La richiesta è atomica: o ottiene tutte le η unità, altrimenti il job è bloccato (la sua esecuzione è sospesa).Termine blocco è giustificato nel contesto: se non posso ottenere la risorsa, vuol dire che un job a priorità minore di me ha la risorsa. Quando job non ha più bisogno della risorsa fa un rilascio $U(R_i, \eta)$.
Spesso il controllo di accesso è affidato a primitive software di tipo lock/unlock.
Spesso la risorsa R_i ha una sola unità disponibile ($\nu_i = 1$), abbrevio quindi con $L(R_i)$ ed $U(R_i)$. È una semplificazione, ama algoritmi che studio sono facilmente adattabili ad un situazione con η variabile.
Conflitto di risorse: due job hanno un conflitto di risorse se potenzialmente possono chiedere una risorsa dello stesso tipo. Due job si contendono una risorsa se uno dei due richiede una unità di risorsa che è già posseduta dall'altro job.

3.2 Sezioni critiche

Si definisce sezione critica un segmento di esecuzione di jon che inizia con $L(R_i, \eta)$ e termina con $U(R_i, \eta)$. Le richieste di risorse di un job possono essere annidate, ma assumiamo che i rilasci sono sempre LIFO.
Una sezione critica non contenuta in alcun'altra sezione critica è detta esterna.
La notazione $[R_1, \eta_1;e_1[R_2,\eta_2;e_2]]$ corrisponde a:
$L(R_i, \eta_1)$ $L(R_2, \eta_2)$ $U(R_2, \eta_2)$ $U(R_1, \eta_1)$ rispettivamente la lunghezza di e_2 è contenuta nella la lunghezza di e_1 (ovvero nella regione critica di R_1 viene fatta la richiesta di R_2). Quando un certo job richiede una certa risorsa? Non c'è l'indicazione, ragiono sul worst case. esempio:
schedulazione con EDF con una unità di risorsa (notazione: freccia bassa è richiesta di risorsa, freccia alta è rilascio).



Le inversioni di priorità causata dal possedere la risorsa causa anomalie di schedulazione: se ad esempio riduco la durata della regione critica d T_3, quindi apparentemente i job di priorità più alta dovrebbero essere favoriti, ma non è così:



quando job di T_3 rilascia la risorsa, il job di T_1 non è ancora stato rilasciato e quindi entra job di T_2, quindi T_1 otterrà la ricorsa troppo tardi. Le inversioni di priorità possono causare anomalie di schedulazione, devo tenerne conto nell'analisi di schedulabilità.

3.3 Controllo d'accesso alle risorse condivise

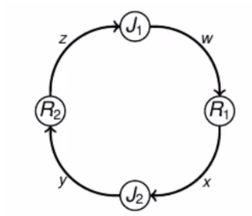
Algoritmi per il controllo di accesso sono necessari:

- Le inversioni di priorità devono essere controllate, altrimenti sarebbero arbitrariamente lunghe.esempio: J_3 acquisisce risorsa e poi viene bloccato da J_1 che vuole acquisire risorsa. Ma ora entra job di J_2 e può essere deciso dallo scheduler di metterlo nel processore, perché a priorità maggiore di J_3: J_2 rallenta sia J_3 che J_1. Il ritardo che J_3 infligge a J_1 non è solo la lunghezza della regione critica fra J_3 e J_1 va misurata nel momento in cui nessuno interrompe J_3, se ci sono processi che interrompono J_2 a priorità maggiore che prendono il posto di J_3 non so quanto sarà lungo il blocco di J_1 (posso avere molteplici job nel mezzo che rallentano). Questo fenomeno si chiama inversione di priorità non controllata.

- Deadlock: altro grave problema. J_2 chiede R_1, J_1 chiede R_2. A quel punto J_1 chiede R_1 ma è bloccato, J_2 continua ed ad un certo punto richiede R_2. Sono in una situazione di deadlock.

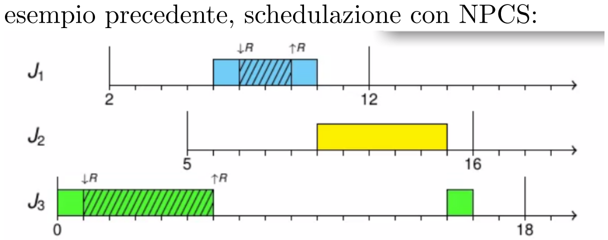
3.3.1 Grafi di attesa

Mutua relazione tra job e risorse è modellabile con grafi di attesa: i nodi dono i job, altri nodi le risorse. Un arco da un nodo di tipo risorsa ad un di tipo job indica che il job ha allocato un certo n° di unità della risorsa. Il viceversa indica che job ha richiesto un certo numero di unità della risorsa ma questa non può essere soddisfatta. Un ciclo del grafo rappresenta un deadlock:



3.4 Protocollo NPCS

Il più semplice protocollo di accesso alle risorse condivise, Nonpreemptive Critical Section: un job avente una risorsa assegnata non può essere interrotto.
Questo risolve tutti i problemi, posso avere deadlock? No, solo a condizione che il job non si auto-sospenda quando ha la risorsa: se job ottiene risorsa e non può essere interrotto non può esserci deadlock, job di priorità superiore non potrà richiedere altra risorsa perché non potrà andare in esecuzione. Se job si auto-sospende tutto il discorso cade: processore è libero e qualcuno può essere schedulato, richiedere una risorsa ed alla fine causare un deadlock.
esempio precedente, schedulazione con NPCS:



Non ci sono deadlock e non c'è inversione di priorità incontrollata: al massimo J_1 sarà bloccato da J_3 per una durata pari alla regione critica.

3.4.1 Tempo di blocco per conflitto di risorse

Sia $b_i(rc)$ il tempo di blocco dovuto ad un conflitto di risorse. Per NPCS con task a priorità fissa $T_1,\ldots,T_n$:
$b_i(rc) = \max\limits_{i+1 \le k \le n}(c_k)$, dove c_k è il tempo di esecuzione della più lunga sezione critica di T_k. Misuro il ritardo che subisce T_i, i job di priorità superiore non mi danno blocchi, se io voglio una risorsa e la trovo bloccata è per via di job a priorità minore, quelli a priorità superiore non mi fanno neanche entrare nel processore; questo in un modello a singolo processore e senza auto-sospensione. Potrei subire un ritardo perché uno dei job a priorità inferiore alla mia è dentro

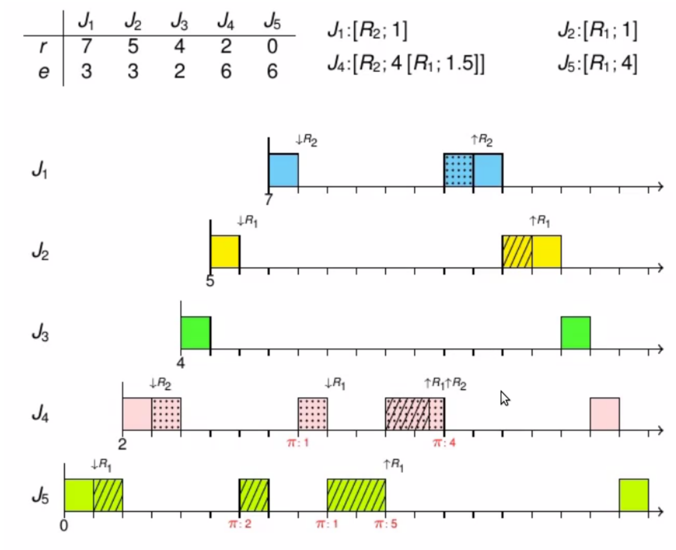
la regione critica e quindi non può essere interrotto secondo NPCS.
Blocco per conflitto di risorse in NPCS è dovuto al fatto che un job a priorità inferiore è dentro la sezione critica.
Formula per $b_i(cs)$ con schedulazione EDF, teorema di Baker: un job J_i può essere bloccato da J_j solo se $d_i < d_j$ e $r_i > r_j$, ossia $D_i < D_j$.
Quindi $b_i(rc) = \max\{c_k\text{: k tale che } D_k > D_i\}$.
Limite del protocollo NPCS: un job può essere bloccato da un job a priorità inferiore anche quanto non ci sono contese o conflitti su alcune risorse. Svantaggioso, quindi si cerca di evitar questo protocollo. D'altra parte, il protocollo è molto diffuso perché è semplice da implementare, non richiede dati sull'uso delle risorse dei job e può essere usato sia per sistemi a priorità fissa che dinamica.

3.5 Protocollo priority-inheritance

Protocollo adatto ad ogni scheduler priority-driven, non si basa sui tempi di esecuzione dei job e riesce ad evitare il fenomeno dell'inversione di priorità incontrollata.
Idea: cambiare le priorità se esistono delle contese sulle risorse per evitare che un job blocca un altro job di priorità più alta sia rallentato da job di priorità intermedi fra i due. esempio di prima: quando J_1 richiede la risorsa, poi J_3 torna normalmente in esecuzione, poi arriva J_2 che rallenterebbe J_3, ma ora il fatto che J_3 sta bloccando J_1 al sua priorità sarà innalzata fino a quella di J_1. In questo modo evito che si possano inserire job di priorità intermedia.
In pratica: i job sono schedulati in modo interrompibile secondo la loro priorità corrente. inizialmente la priorità corrente $\pi(t)$ di un job J rilasciato al tempo t è quella assegnata dall'algoritmo di schedulazione.
Quando un job J richiede una risorsa R al tempo t:

- Se R è disponibile, R vine assegnata a J
- Se R non è disponibile, J è sospeso (bloccato)

Quando un job J viene bloccato a causa di una contesa su una risorsa R, il job J_l che blocca J eredita la priorità corrente $\pi(t)$ di J finché non rilascia R; a quel punto, la priorità corrente di J_l torna ad essere la priorità $\pi_l(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R.
esempio: schedulazione a priorità fissa con priority-inheritance, qui supponiamo che la risorsa venga chiesta dopo un'unità di tempo dal rilascio.



Limiti:

- Non evita i deadlock
- Introduce nuovi casi di blocco: un job a priorità corrente π(t) può bloccare ogni job con priorità assegnata minore di π(t).
- Non riduce i tempi di blocco dovuti ai conflitti sulle risorse al minimo teorico possibile. esempio: ho un job a priorità alta: il job ha sotto di se molti job a priorità inferiore, usa molte risorse annidate. Se tutte le risorse sono assegnate: se accede ad un certo numero v di risorse ed ha conflitti con k job di priorità inferiore assegnata può bloccare per un min(k,v) volte.
  Devo dimensionare il sistema in modo molto pessimista

3.6 Protocollo priority-ceiling

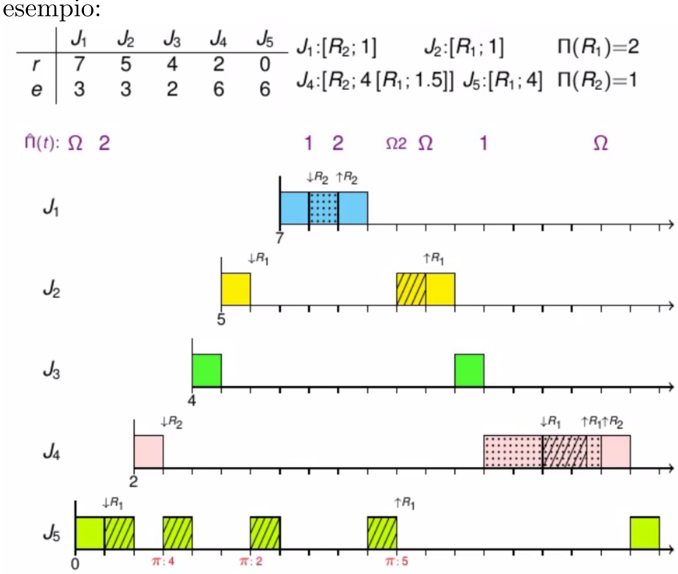
Adatto a scheduler con priorità fissa.È basato sulle richieste di risorse dei job prefissati, evita inoltre tutti e due i problemi.
Idea: associare ad ogni risorsa R il valore priority ceiling $\lceil\rceil$(R) pari alla massima priorità dei job che fanno uso di R. Dato che sa quale task userà quale risorsa, ad ogni risorsa è possibile associare il priority ceiling. Inoltre, il protocollo definisce il current priority ceiling $\lceil\rceil$'(R) che è apri a:

- La massima priorità $\lceil\rceil$(R) fra tutte le risorse del sistema correntemente in uso al tempo t
- al valore convenzionale Ω di priorità inferiore a quella di qualunque task se nessuna risorsa è in uso.

Confrontando le priorità, π(t) > π'(t) significa che π(t) ha maggiore priorità di π'(t); così se a valore inferiore corrisponde priorità superiore, π(t) = 1 e π'(t) = 2 implica che π(t) > π'(t).
Regola di schedulazione: job schedulati in modo interrompibile secondo la loro priorità corrente.
Se al tempo t un job J con una certa priorità corrente π(t) richiede una risorsa R, R è allocata a J sole se è disponibile d inoltre:

- π(t) > $\lceil\rceil$(t)
- J possiede una risorsa il cui priority ceiling è uguale a $\lceil\rceil'$(t)
- altrimenti J è bloccato.

Se J$_l$ blocca J, J$_l$ eredità la priorità corrente π(t) di J finché J$_l$ non rilascia l'ultima risorsa R tale che $\lceil\rceil$(R) $\geq$ π(t); a quel punto la priorità di J$_l$ torna ad essere la priorità π_l(t') che aveva al momento t' in cui aveva acquisito la risorsa R.
esempio:

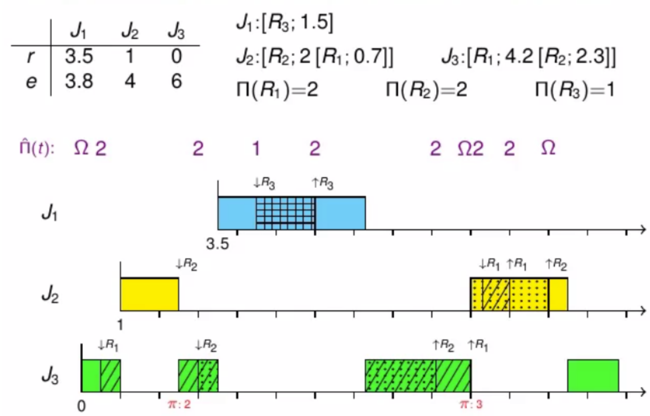


Stabilisco prima i priority ceiling delle risorse. Ho un blocco: il motivo per cui J$_4$ non può continuare perché è bloccato da J$_5$, quindi J$_5$ eredita la priorità di J$_4$.
In quanti casi diversi un job J$_l$ può bloccare un job J$_h$ con priorità $\pi l < \pi h$:

- Blocco diretto: J$_h$ richiede una risorsa R assegnata a J$_l$.
- Blocco dovuto a priority-inheritance: la priorità corrente di J$_l$ è maggiore di quella di J$_h$, perché J$_l$ sta bloccando direttamente un job che ha priorità maggiore di J$_h$.

- Blocco dovuto al priority ceiling (o avoidance blocking): J_h ha richiesto una risorsa R ma J_l possiede un'altra risorsa R' tale che $\lceil\rceil$(R') $\geq \pi_h$.

I deadlock possono essere evitati se tutti i job acquisiscono le risorse annidate rispettando un unico ordinamento globale delle risorse; metodo principale usato nei sistemi operativi.
I priority ceiling non definiscono un ordinamento globale delle risorse, bensì parziale ma che basta ad evitare i deadlock. esempio:



Sono esposto ad un deadlock, ma con priority ceiling non avverrà: al tempo 2.5 J_2 richiede R_2, ma la richiesta viene rifiutata anche se R_2 è libera, così si evita un possibile deadlock con J_3. I job con priorità corrente maggiore di $\lceil\rceil'$(t) possono acquisire risorse sezna rischiare deadlock con le risorse già assegnate.
Posso avere molti job e risorse: ho J_1, che usa R_1,R_2, poi ho J_2 che usa R_3,R_4, userà anche R_1,R_2 ma non è un problema, perché il priority ceiling di R_1,R_2 è quelli di J_1 e così via per i vari livelli:
$\lceil\rceil(R_1) = \lceil\rceil(R_2) = \pi_1$
$\lceil\rceil(R_3) = \lceil\rceil(R_4) = \pi_2$. e così via
Suppongo che $\lceil\rceil'(t_0)$ sia ad un certo livello π_k: questo vuol dire che sono assegnate nel sistema solo risorse al di sotto di questo livello. Se al tempo t_0 un job richiede un risorsa e la sua priorità $\pi_J(t_0) > \lceil\rceil'(t_0)$:

- J non richiederà mai alcuna risorsa già assegnata al tempo t_0. Quindi non avrò nessun deadlock con le risorse già assegnate

- Nessun job con priorità maggiore di $\pi_j(t_0)$ chiederà alcuna risorsa già assegnata la tempo t_0 , quindi nessun job che già possiede una risorsa al tempo t_0 portà interrompere J e richiedere R.

Il risultato è che il protocollo priority-ceiling evita i deadlock.

3.6.1 Proprietà del protocollo priority-ceiling

Come visto sopra:

- al tempo t un job possiede tutte le risorse assegnate aventi priority ceiling uguale a $\lceil\rceil'(t)$.
- se un job sta per ottenere un risorsa $\pi(t) > \lceil\rceil'(t)$, nessun job di priorità uguale o superiore ha richiesto o richiederà le risorse già assegnate
- Se un job sta per ottenere una risorsa $\pi(t) = \lceil\rceil'(t)$, il job è il possessore di tutte le risorse assegnate aventi priority ceiling uguale a $\lceil\rceil'(t)$.
- i deadlock sono dunque evitati: priorità assegnate all risorse definiscono in un certo modo un ordinamento non totale tra le risorse.

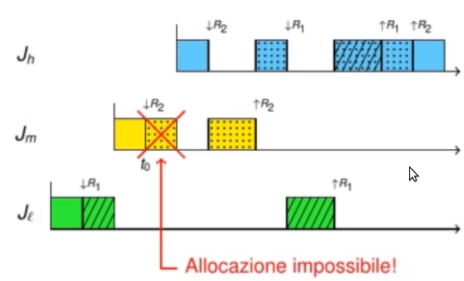
Se al tempo t_0 un job J richiede una risorsa R e $\pi(t_0) > \lceil\rceil'(t)$:

- J non richiederà mai alcuna risorsa già assegnata aò tempo t_0
- Nessun job a priorità $\geq \pi(t0)$ chiederà una risorsa già assegnata al tempo t_0.

Quindi priority ceiling evita i deadlock.
Non basta questa proprietà per giustificare la complessità di priority ceiling: basterebbe programmare bene i job nel sistema per evitare i deadlock.
In priority-ceiling, ho 3 blocchi possibili: blocco diretto, priority-inheritance, priority-ceiling.
C'è un teorema:
utilizzando il protocollo priority-ceiling un job può essere bloccato al massimo per la durata di una sezione critica.Teorema vuol dire che su un job subisce blocco a causa di una risorsa condivisa lo farà una volta sola, non subirà blocchi consecutivi. Inoltre blocco non sarà per un tempo costituito da annidamento di diverse sezioni critiche, ma per un tempo pari a solo la durata di una sezione critica.2 proprietà:

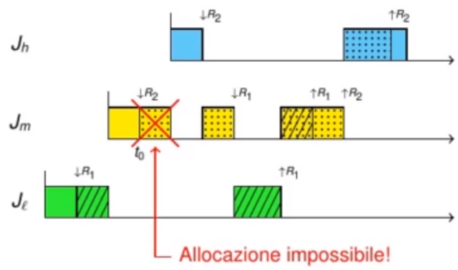
- Se un job viene bloccato, è bloccato da un solo job
- Non esiste blocco transitivo: non si verifica mai il caso J_3 blocca J_2 e J_2 blocca J_1.

Unicità del job bloccante:



ho 3 job di priorità variabili.
J_h è bloccato sia da J_m che da J_l. Perché non può avvenire in priority ceiling: $\pi_h > \pi_m > \pi_l \Rightarrow \lceil\rceil(R_1) \geq \pi_h$ e $\lceil 1rceil(R_2) \geq \pi_h$.

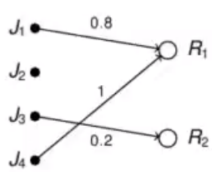
Ora: $\lceil\rceil'(t_0) \geq \lceil\rceil(R_1) \geq \pi_h$. Il requisito per l'allocazione a t_0 deve essere $\pi_m > \lceil\rceil(R_1) \geq \pi_h$. Ma questo non è verificato e quindi il priority ceiling nega l'assegnazione della risorsa a J_m.
Se J_m acquisisce una risorsa a t_0, nessun job con priorità maggiore o uguale può richiedete una risorsa già in uso a t_0.
Impossibilità del blocco transitivo:



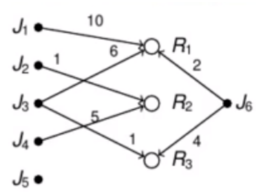
J_l sta bloccando J_m e J_m sta bloccando J_h. Perché non può verificarsi:
$\pi_h > \pi_m\ \pi_l \Rightarrow \lceil\rceil(R_1) \geq \pi_m$ e $\lceil\rceil(R_2) \geq \pi_h$.
Quindi $\lceil\rceil'(t\_0) \geq \lceil\rceil(R\_1) \geq \lceil\rceil' t_0$; quindi l'allocazione non può avvenire.

3.6.2 Tempo di blocco per conflitto di risorse

È il massimo tempo di ritardo un job del task T_i causato da un conflitto di risorse.
Come faccio per calcolarlo: ho 3 tipi di blocco, quindi devo calcolarlo per tutti e 3 tipi, mi il teorema mi dice che il job può essere bloccato per al massimo una sezione critica, quindi considero il massimo.esempio:
J_1:[R_1;0.8], J_2, J_3:[R_2;0.2], J_4:[R_1;1].



J_4 può bloccare direttamente J_1 per 1 unità di tempo $\Rightarrow b_1(rc) = 1$. J_4 può bloccare anche J_2 e J_3, quando acquisisce $R_1 \Rightarrow b_2(rc) = b_3(rc) = 1$: può bloccare per priority inheritance. Sto facendo analisi pessimista: J_4 chiede la risorsa e subito dopo la chiede J_1. Il job per J_4 è $b_4(rc) = 0$, perché il job di priorità più bassa.
Per esempi più complessi, conviene avere un algoritmo automatico per derivare



Blocco diretto (B_d):

B_d	J_2	J_3	J_4	J_5	J_6
J_1		6			2
J_2	*		5		
J_3		*			4
J_4			*		
J_5				*	

i tempi di blocco:

- Costruisco una tabella dei tempi di blocco diretti. Su entrambe le colonne avrò i nomi dei job, ciascuna componente rappresenta il tempo di blocco diretto che il job della colonna fa subire al job della riga.
  Le righe hanno i 5 job di priorità superiore mentre le colonne i 5 di priorità inferiore
- Metto asterisco sugli elementi della "diagonale", ovvero le righe e colonne con lo stesso job, so che sotto la diagonale il blocco non può avvenire, quindi avrò valore 0.
- Riempio le componenti sopra gli asterischi: vedo chi è in conflitto sulle varie risorse.

Posso derivare in maniera automatica la tabella per i blocchi dovuti all'inheritcance:

Blocco per inheritance (B_i):

B_i	J_2	J_3	J_4	J_5	J_6
J_1					
J_2	*	6			2
J_3		*	5		2
J_4			*		4
J_5				*	4

avviene in caso di contesa, quindi quando un job nel blocca un altro viene trasferita la priorità. J_3 blocca J_1 per 6 unità di tempo. Ma allora il job di priorità intermedia può essere bloccato per 6 unità di tempo, quindi il 6 scende di una posizione nella tabella. Blocco tra J_2 e J_4, questo danneggia i job di priorità intermedia, ovvero J_3, il 5 scende fino alla riga di J_3.
J_6: blocca per inheritance anche tutti i job di priorità intermedia tra lui e J_1, e quindi tutti gli altri. Voci scendono sempre fino all'asterisco. Ad un certo punto, trovo che J_6 sta bloccando direttamente anche J_3, quindi per inheritance J_3 è bloccato per 2 unità di tempo, ma poi J_4 sarebbe bloccato per 2 unità di tempo a causa della contesa con J_1 ma anche per 4 unità per via della contesa con J_3. Devo considerare il worst case: ora è 4, quindi è il 4 a propagarsi verso il basso.
Infine ho la tabella per il blocco per priority ceiling:

Blocco per ceiling (B_c):

B_c	J_2	J_3	J_4	J_5	J_6
J_1					
J_2	*	6			2
J_3		*	5		2
J_4			*		4
J_5				*	

Questo diventava simile al blocco per inheritance: J_4 può essre danneggiato da J_6 perché J_6 richiede risorsa che innalza il priority ceiling, caso peggiore è il max fra la lunghezza della regione critica fra R_1 ed R_2 (per J_6). Siccome J_5 non richiede risorse, manca il valore perché J_5 non può mai essere bloccato in quanto non richiede risorse.
A questo punto posso definire B_i(r,c) = max{B_d(j, c): 1 ≤ j ≤ r-1}.
Se le priorità dei job sono tutte diverse, $B_c = B_i$, tranne che per i job che non utilizzano risorse.
b_i(rc) = $\max_k B_d$ B_i(j, k), B_c(i, k): considero il valore massimo per ciascuna riga, perché priority ceiling mi dice che blocco al massimo 1 volta. Cosa cambiare se i job possono avere priorità identiche?

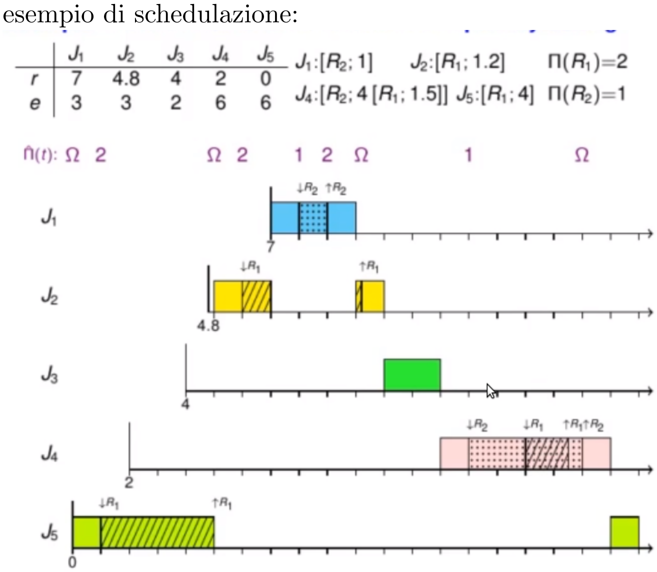
3.7 Schedulabilità con priority ceiling

Ho i tempi di blocco, li considero tra i tempi di blocco totali dei task, lo faccio task per task:
b_i = b_i(ss) + (K_i + 1)·b_i(np) + (K_i + 1)·b_i(rc), con K_i massimo numero di auto-sospensioni di un job del task T_i. Il fatto dell'unicità del job bloccante vale solo se i job non si auto-sospendono. Devo anche tenerne conto all'overhead su cambi di contesto: e_i' = e_i + 2·(K_i + 1)·CS + 2·(K_i + 1)·CS, ma solo se il job usa le risorse condivise.

3.8 Protocollo stack-based priority-ceiling

Baker, 1991. È una semplificazione del protocollo priority-ceiling, motivato da un esigenza particolare: la condivisione di un unico stack da parte dei job. Usare uno stack unico comporta problemi: stack è LIFO, ogni volta che arriva un job sopra, interrompe quello sotto. Se un job arriva e richiede una risorsa, ma poi arriva una altro job che interrompe: comincia ad usare lo stack, in una zona contigua a quel job interrotto. Quando il job si conclude, toglie dallo stack tutte le informazioni che aveva introdotto. Ma se il jbo richiede la stessa risorsa del job che ha interrotto: per priority ceiling deve tornare in esecuzione il job interrotto. Probelma: il job non può togliere le info dallo stack ed ora il job che torna si trova una parte dello stack occupato.
Questo porta al fatto che nessun job deve bloccare o auto-sospendersi.
Per ogni risorsa R, $\lceil\rceil$(R) definito come nel protocollo priority-ceiling. C'è regola di aggiornamento che è la stessa.
Regola di schedulazione: non appena rilasciato, un job J con priorità maggiore assegnata π non può essere eseguito finché non è vera la condizione $\pi \leq \lceil\rceil$'(t).

I job eseguibili sono schedulati in modo interrompibile in accordo alle priorità assegnate.Non c'è priority inheritance
Regola di allocazione: quando un job richiede una risorsa, la richiesta è soddisfatta.
Un job di priorità alta può interrompere un job di priorità bassa, e quest'ultimo non può tornare in esecuzione finché il primo non ha finito. Potrebbe farlo se il job bloccasse, ma questo non succede perché risorsa è libera o se si auto-sospendesse, ma qui questo non avviene. Quando un job comincia l'esecuzione tutte le risorse di cui ha bisogno sono libere: difatti inizia solo se la sua priorità diviene uguale o maggiore del priority ceiling del sistema.
Non ci sono mai deadlock: le risorse sono sempre libere quando le richiedo. Job non i auto-sospendono: il controllo d'accesso è effettuato solo al rilascio di un job e assume che il job non venga sospeso.
esempio di schedulazione:



3.9 Ceiling priority

Usato nel Real-time System Annex di Ada95: linguaggio molto usato negli USA. È stato definito dal governo per lo sviluppo del sw in tutte le commesse militari.
Regola di schedulazione:

- Se un job non possiede alcuna risorsa, la sua priorità è quella rassegata dallo scheduler
- Se un job possiede una risorsa, la sua priorità è uguale al massimo priority ceiling di tutte le risorsa assegnate al job.

Job con priorità identica sono schedulati in modo FIFO.
Regola di allocazione: quando un job richiede una risorsa la ottiene. Risorsa

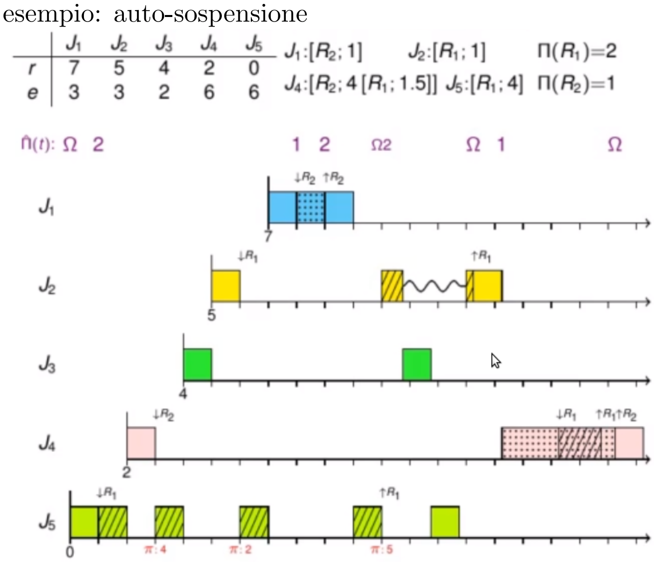
è sempre libera: se fosse occupata, il job che la sta usando avrebbe priorità almeno uguale a quello che la sta richiedendo.
Differenza fra stack-based e ceiling priority? Senza auto-sospensione le schedulazioni prodotte sono identiche. Però in ceiling-priority è possibile modificare le regole per permettere auto-sospensione.
Confronto tra i protocolli: Teorema(Baker, 1911): I tempi di blocco massimi b_i(rc) dovuti ai conflitti di risorse per priority-ceiling e per stack-based priority-ceiling sono identici. Quindi scheduler che usano stack-based o ceiling-priority sono più semplici ed efficienti, in più hanno meno context-switch. Però i cambi di prioirtà dinamiche sono meno frequenti in priority-ceiling, che ci sono solo in caso di contesa.
Trade off, ma spesso vincono gli ullimi due

3.10 Controllo d'accesso per job con auto-sospensione

I vari protocolli vanno adattati all'auto-sospensione:

- NPCS: non è possibile auto-sospendersi in una sezione critica
- Priority-inheritance: se un job J è bloccato su una risorsa posseduta da un job J' auto-sospeso, la priorità dinamica di J' è aggiornata solo se π(t) > π(t')
- Priority-ceiling: non funziona più unicità del blocco
- Stack-based: non esiste
- Ceiling-priority; se un job si auto-sospende in una sezione critica, nessun job di priorità inferiore o uguale può essere eseguito. È come se nullificasse i vantaggi dell'auto-sospensione nelle sezioni critiche

esempio: auto-sospensione

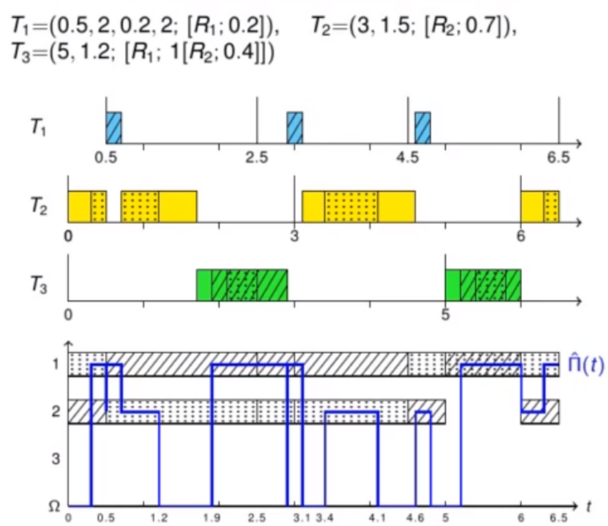


Tempi di blocco per autosospensione

- NPCS: $b_i = b_i(ss) + (K_i + 1) \cdot \max\{b_i(np), b_i(rc)\}$
- priority ceiling e ceiling priority: $b_i = b_i(ss) + (K_i + 1) \cdot (b_i(np) + b_i(rc))$. Qui i tempi di blocco $b_i(rc)$ vanno calcolati anche pensando che mentre sono in sezione critica posso auto-sospendermi, quindi debo considerare anche il tempo massimo di auto sospensione e moltiplicare per il numero di volte in cui mi auto-sospendo.

3.11 Priorità dinamica

È possibile applicare i protocolli priority-ceiling e ceiling-priority a sistemi con priorità dinamica. Il valore del priority ceiling di una risorsa non è costante, ma dipende dalla priorità dinamica che potenzialmente fanno uso della risorsa. Il priority ceiling può cambiare, ad esempio con EDF ogni volta che rilascio un nuovo job, questo ha una prioirtà dovuta alla scadenza assoluta che fa cambiare i valori di priorità di tutti ti job del sistema, quindi i, pririty ceiling delle risorse e quindi il current priority ceiling del sistema. Molto poco applicabile nella realtà. Quando schedulo con algoritmi a priorità dinamica uso o NPCS o priority inheritance o altri sistemi per evitare deadlock come allocare risorse in tempi predefiniti.
esempio di priority ceiling con EDF:



3.12 Accesso alle risorse di job aperiodici

Problema: un server procrastinabile che sta eseguendo un job aperiodico esaurisce il budeget mentre il job è in sezione critica:

- Esecuzione all'interno della sezione critica rende il server non interrombilile anche se il budgrt finisce
- Se ho accumulato ritardo, rifornisco meno budget.
- Problema di schedulabilità e ritardi aggiuntivi, devo aggiungere anche la lunghezza della sezione critica dei job aperiodici. Questo comporta difficoltà nello studio, ma è modellabile

4 Real-time su multiprocessore

Ho rimosso man mano le limitazioni complicando il modello, rilasso l'ipotesi di avere un singolo processore(ultima limitazione rimasta).
Sistema multiprocessore ha 2 o più processori, ciascuno può eseguire job in maniera indipendente.
Processori possono essere dello stesso tipo o di tipo diverso:

- diversi microprocessori mutli-core
- diverse schede di rete
- diverse schede PCI con controllori DMA

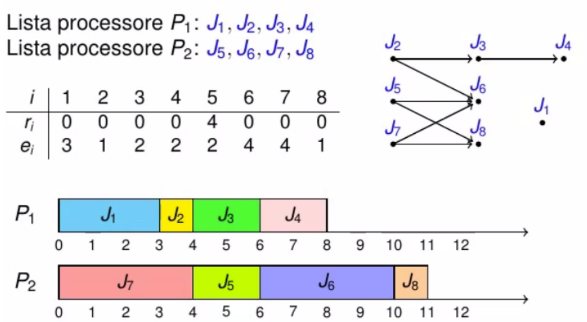
In un certo senso, per cercare di modellare il sistema avrò μ tipi di processori quanti processori m_i ci sono per i $\leq$ i $\leq \mu$, anche su quali tipi di processore si può eseguire ciascun job. Semplifico supponendo che tutti i processori sono dello stesso tipo.
Ci si è resi conto molto presto che aggiungere processori complicava le cose: molto più complesso che schedulare su singolo processore.

4.1 Sistemi statici

Un sistema real-time è statico se ciascun job è assegnato perennemente ad uno specifico precessore. Partiziono i task nel sistema tale che ciascun job o task è eseguito forzatamente da uno specifico precessore: posso effettuare la schedulazione normalmente, ciascuno ha una lista di job e mi riconduco al caso del singolo processore (non proprio così).
2 tipi di sistemi statici:

- L'assegnazione dei job è effettuata manualmente dal progettista una volta per tutte in fase di progettazione del sistema. Problema: devo conoscere tutti i parametri dei task, non posso gestire nuovi task che arrivano a run time
- L'insieme dei task è assegnato ad uno specifico processore dal SO (o scheduler) durante la fase di creazione del task. Scheduler partizionati.

In entrambi i casi, lo scheduler non decide su quale processore sarà eseguito un job appena rilasciato.
esempio di schedulazione:



I job hanno delle dipendenze: questo fa capire che il sistema non è analogo a tanti sistemi mono-processore, non ci sarebbero vincoli fra job su processori diversi. Ogni scheduler decide i job assegnati ai processori, ma i vincoli di dipendenza sono grosse complicazioni.
Vantaggi:

- Si può analizzare la schedulabilità su ciascun processore usando i risultati teorici validi per il caso mono-processore
- Se un job va in overrun (impiega più tempo del previsto) può ritardare l'esecuzione dei soli task che sono sul suo stesso processore (se job sono indipendenti tra di loro)
- Se un job è interrotto, siccome il sistema è statico riprenderà l'esecuzione sullo steso processore, evito i costi dovuti alla migrazione
- La coda di esecuzione è relativa al singolo processore ed è quindi più piccola

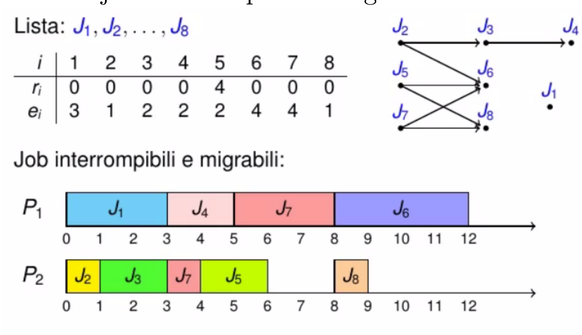
4.2 Sistemi dinamici

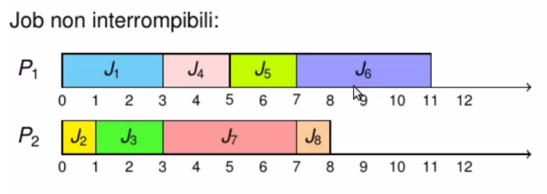
Un sistema real-time è dinamico se lo scheduler assegna dinamicamente un job ad un qualunque processore disponibile 3 varianti:

- Con job interrompibili
- Con job interrompibili e non migrabili: anche se interrotto, job è salvato in una struttura locale del processore e potrà recuperare l'esecuzione solo si quel processore
- COn job interrompibili e migrabili: job può essere migrato se interrotto, ha un costo

Una algoritmo di schedulazione per un sistema dinamico è globale perché stabilisce quale processore eseguirà quale job.
esempio:

1° caso: job interrompibili e migrabili:





2° caso, job non interrompibili:
Il momento in cui l'ultimo job completa il lavoro è uguale nel caso dei job migrabili a quello della schedulazione nel sistema statico $\Rightarrow$ anomalia di schedulazione.
Vantaggi:

- Hanno tipicamente meno cambi di contesto: quando viene rilasciato un job, in un sistema statico se c'è in esecuzione job di priorità minore devo interrompere e fare context swtich. In sistema dinamico posso avere processori free, quindi metto in esecuzione lì.
- Se un job esegue per meno tempo di quello che è il suo worst case, il tempo liberato sul processore può essere utilizzato potenzialmente da tutti i task de sistema (nel caso del sistema statico è usato solo da quelli locali al processore).
- Se un job impiega più tempo (overrun) la probabilità che questo comporti la mancanza di una op più scadenze è minore. Non è contrapposizione con 2: sistema può usare tutti i processori per cercare di porre rimedio al tempo in più per cui il job ha eseguito
- Per ogni task del sistema che è creato a run-time, assegnazione e bilanciamento del carico sono "automatici" e determinati dall'algoritmo di schedulazione globale

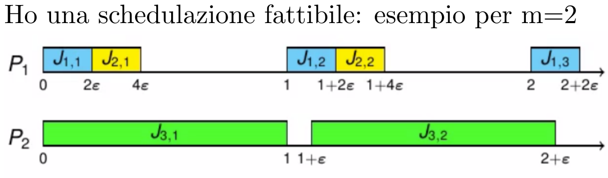
4.3 Algoritmi di schedulazione multiprocessore

Gli algoritmi clock-driven sono in generale utilizzabili senza problemi, infatti la schedulazione avviene offline e validata una volta per tutte.Metodo però poco flessibile, ma non è immediato applicare algoritmi prioirty-driven, devo studiare bene come fare: devo risolvere diversi problemi

- Efficienza degli algoritmi, effetto Dhall
- Predicibilità del sistema
- Test di schedulabilità.

4.4 Effetto Dhall

Teorema (Dhall & Liu): Per ogni numero di processori m $\geq$ 2, esistono insieme di task con utilizzazione bassa che non sono schedulabili con RM, DM o EDF.
Considero T_1=81, 2ϵ), T_2=(1,2ϵ,....,T_m=(1,2ϵ), T_{m+1}?(1+ϵ, 1). Utilizzazione globale = $2\epsilon \cdot$ m + $\frac{1}{1+\epsilon} \to 1$ se $\epsilon \to 0$. Basterebbe uno o al massimo due processori per eseguire tutti questi task.
Ho una schedulazione fattibile: esempio per m=2



Problema è che la schedulazione non è RM né DM né EDF: se schedulo con EDF vincono le scadenze assolute, quando tutti i job vengono rilasciati due job hanno priorità sugli altri, quindi quando i processori si liberano uno dei job manca la scadenza; stesso vale per RM e DM.
Questo risultato ha scoraggiato per tanti anni la ricerca su sistemi multiprocessore real-time, riprende quando siatemi multiprocessore si sono largamente diffusi da costringere a riguardare il problema: 2001, effetto Dhall esiste solo con sistemi di task in cui uno dei task ha un utilizzazione molto alta. Se task hanno utilizzazione non uguale ad 1 non si verifica l'effetto Dhall.

4.5 Anomalie di schedulazione

Comportamene di un algoritmo tale che, a fronte di variazione apparentemente vantaggiose dei parametri si hanno dei peggioramenti delle prestazioni. Esempi:

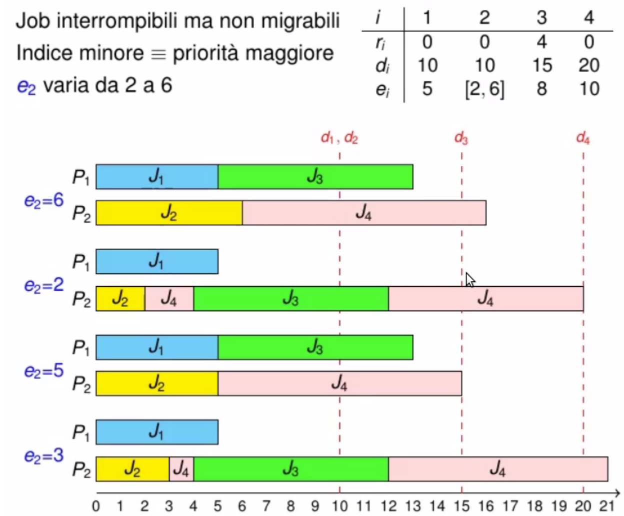
- Aumentano il periodo di un task
- Aumento n° processori
- Diminuisco il tempo di esecuzione di un task

Anomalie di schedulazione si verificano se task sono non interrompibili o indipendenti, nei sistemi uni-processore.
Nei sistemi multiprocessore? Mi metto nelle ipotesi che non ci siano vincoli di dipendenza:

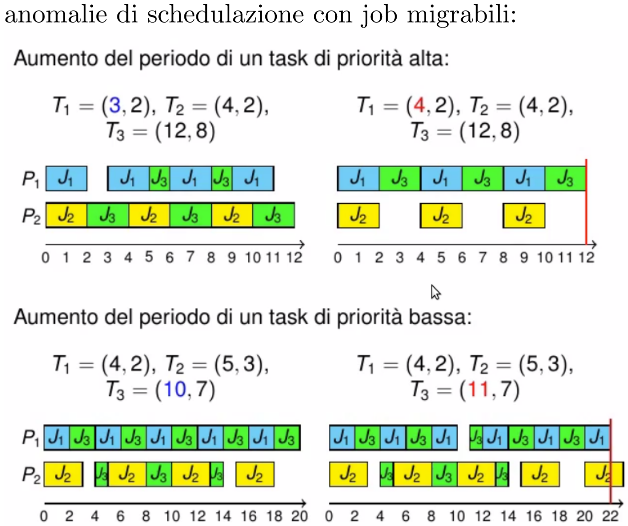
- Sistema statico, job non interrompili: posso avere anomalie
- Sistema statico, interrompibili: non ho anomalie

- Sistema dinamico, job non interrompibili: posso avere anomalie
- Sistema dinamico, job interrompibili ma non migrabili: sì
- Sistema dinamico, job interrompili e migrabili: sì.

Perché le anomalie complicano la validazione? Non esiste un worst case a cui ricondursi, bisogna cercare un modo per tenere sotto controllo le anomalie.
esempio: anomalie di schedulazione con job non migrabili

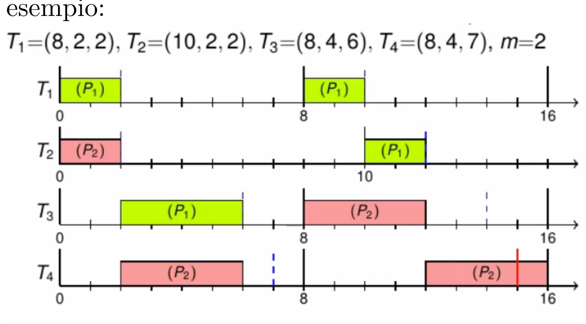


anomalie di schedulazione con job migrabili:



4.6 Schedulabilità

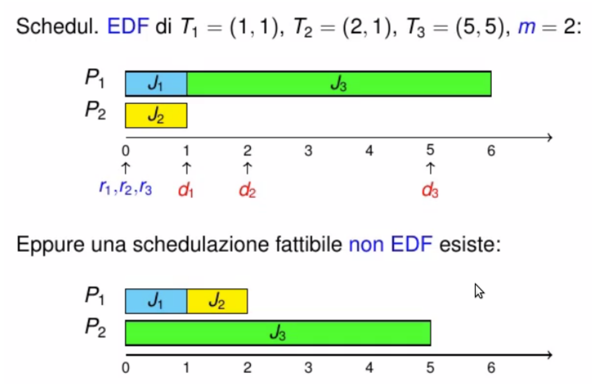
Istanti critici in schedulazioni globali: c'è un teorema che mi dice che usando uno scheduler globale a priorità fissa a livello di task, l'istante in cui un job di un task T_i è rilasciato contemporaneamente ai job di tutti i task di priorità superiore T_1,....,T_{i-1} non è necessariamente un istante critico di T_i
esempio:



No ho modo di usare il test di schedulabilità.
Fattore di utilizzazione per multiprocessore: (thm) dato un sistema di task periodici con scadenze uguali ai periodi ed m processori, se X è un qualsiasi algoritmo di schedulazione partizionato con priorità fissa a livello di task:
$U_X \leq \frac{m+1}{1+2^{\frac{1}{m+1}}}$. In pratica: se ad esempio uso RM, che ha su mono processore $U_X \simeq 0.69$, considerandolo partizionato il fattore di utilizzazione è limitato, non può in nessun modo utilizzarlo.
Teorema(2001): dato un sistema di task periodici con scadenze implicite ed m processori, sia X:

- un qualsiasi algoritmo di schedulazione partizionato
- un qualsiasi algoritmo di schedulazione globale con priorità fissa a livello di job

allora per il fattore di utilizzazione di X si ha: $U_X \leq \frac{m+1}{2}$.
Man mano che aumento il numero di processori, devo lasciarne sempre di più liberi. Sto quindi lavorando in perdita: se voglio aumentare di un processore il mio sistema, ne devo aggiungere 2 e cos' via...
Corollario: nessun algoritmo di schedulazione globale con priorità fissa a livello di job è ottimale su multiprocessore. Non posso sfruttare al 100% tutti i processori del sistema.
esempio:



Come si vede nell'esempio, schedulazione non EDF ottimale esiste.
Esistono algoritmi ottimali di schedulazione dinamica a livello di job che hanno fattore di utilizzazione pari ad m, esempio LST che però è complicato da implementare. Non esistono algoritmi on-line ottimali se gli istanti di rilascio dei job non sono esattamente prefissati.
Classe di algoritmi ottimali su multiprocessore è derivata dall'algoritmo Pfair:

- basati sul concetto di schedulazione fluida. Ogni task progredisce in modo proporzionale alla sua utilizzazione.
- tempo diviso in quanti: allo scadere di ogni quanto, lo scheduler assegna i task ai processori in modo che per ogni task T_i, il lavoro compiuto sia $\lceil \frac{te_i}{p_i} \rceil$ o $\lfloor \frac{te_i}{p_i} \rfloor$.
  Più è piccolo il quanto di tempo, più mi avvicino ad una schedulazione fluida, come in un SO in cui processi si alternano per quanti piccoli: sembra come se processi facessero progressi contemporaneamente

Gli algoritmi dinamici a livello di job sono molto costosi in termini di overhead dello scheduler. quindi non sono adottati.

4.7 Schedulazione partizionata

Nei sistemi real-time multiprocessore statici, l'algoritmo di schedulazione è partizionato. Non parlo di algoritmi in cui il progettista fissa i task ai processori, serve descrivere due componenti:

- Algoritmo che assegna i task ai processori (problema NP hard, non ammette algoritmo ottimale efficiente, tempi esponenziali nell'istanza del problema)
- Algoritmo che schedula i task su ciascun processore

4.7.1 Allocazione dei task

Dato un sistema di task periodici,partizionare i task in sottoinsiemi tali che ciascun sottoinsieme può essere schedulato in modo fattibile su un singolo precessore utilizzando un determinato algoritmo do schedulazione.

Un sistema di n task indipendenti è schedulabile con n processori (purché ciascun task abbia densità inferiore ad 1).
Non esiste algoritmo polinomiale che possa capire il più piccolo n° di processori per schedularlo. Gli algoritmi di schedulazione dei task possono solo calcolare soluzioni approssimate (non ottimali):

- Non riescono ad associare i task ai processori in modo da sfruttarli nel miglior modo possibile
- Non riescono a determinare schedulabilità fattibili per ogni possibile insieme di task schedulabile

Limiti che non sono superabili.
Metriche di bontà:

- Rapporto di approssimazione: è il massimo valore $\frac{m}{m_0}$, dove m è il numero di processori usato dall'algoritmo di allocazione ed m_0 è il minimo numero teoricamente necessario, considerando ogni possibile sistema di task. Buono se n° task è piccolo, m_0 non è facile da determinaste
- Fattore di accelerazione: quanto è necessario aumentare la velocità di esecuzione degli m_0 processori per schedulare fattibilmente ogni possibile sistema di task le assegnazioni determinate dall'algoritmo di allocazione.
- Fattore di utilizzazione: il valore di soglia per cui i sistemi di task con fattore di utilizzazione totale inferiore o uguale sono sempre schedulabilità utilizzando l'algoritmo di allocazione dei task.

4.7.2 RMFF

Il più semplice (Rate Monotonic First Fit), passi:

- ordina i task per periodi non decrescentisi T1,...,Tn
- ordina arbitrariamente i processori: P1,...
- comincia da T1, assegna ciasun task Ti al primo processore Pj tale che l'insieme dei task già assegnati a Pj insieme a Ti risulta ancora schedulabile tramite RM.

U$RMFF$ = m $\cdot$ ($\sqrt[2]{2}$ - 1).
Fattore di approssimazione: 2.23, usa un numero di processori che è 2.33 $\cdot$ numero ottimale (più del doppio). Non può essere usato come algoritmo online: siccome ordino per periodo non decrescenti, se arriva un nuovo task a run-time devo rifare tutto e quindi anche le assegnazioni, il che è impossibile. Usabile solo se conosco tutti i task in anticipo.

4.7.3 FFDU

Passi:

- ordina i task per periodi non decrescentisi T1,...,Tn
- ordina arbitrariamente i processori: P1,...
- cominciando da T_1, assegna ciascun task T_i al primo processore P_j tale che l'insieme dei task già assegnati a P_j insieme a T_i risulta ancora schedulabile tramite RM.

$U_{FFDU} = m \cdot (\sqrt[2]{2} - 1)$
Fattore di approssimazione: 1.67. Poiché richiede di ordinare i task, comunque non è on-line.

4.7.4 RM-FF

Una variante di RMFF, che non effettua l'ordinamento dei task prima dell'allocazione:

- ordina arbitrariamente i processori: P1,...
- assegna ciascun task T_i al primo processore P_j tale che l'insieme dei task già assegnati a P_j insieme a T_i risulta ancora schedulabile secondo RM

$U_{RM-FF} = m \cdot (\sqrt[2]{2} - 1)$
Fattore di approssimazione: 2.33.
A differenza di RMFF, RM-FF può essere usato on-line.

4.7.5 EDF-FF

Lo stesso algoritmo di RM-FF, ma schedulo il singolo processore con EDF:

- ordina arbitrariamente i processori: P1,...
- assegna ciascun task T_i al primo processore P_j tale che l'insieme dei task già assegnati a P_j insieme a T_i risulta ancora schedulabile secondo EDF

$U_{EDF-FF} = \frac{\beta \cdot m+1}{\beta+1}$, $\beta = \lfloor \frac{1}{\max\limits_k \frac{e_k}{p_k}} \rfloor$.

Fattore di approssimazione: 1.7 (meno del doppio del n° processori ottimali). È un algoritmo ottimale fra tutti gi algoritmi ottimali:
per $\beta = 1 \Rightarrow U_{EDF-FF} = \frac{(m+1}{2}$, ovvero nel caso peggiore ho il limite superiore nel caso in cui c'è un task di dimensione 1. Posso convivere con l'effetto Dhall, pagando processori in più. Se $\beta \to \infty \Rightarrow U_{EDF-FF} \to$ m, per task di dimensioni infinitesima e quindi schedulabili in maniera fluida, l'algoritmo riesce a raggiungere il 100% di utilizzazione dei processori del sistema.