

Contents

1 Schedulabilità di algoritmi a priorità fissa	4
1.1 Istanti critici	4
1.2 Schedulabilità per priorità fissa e tempi di risposta piccoli	5
1.3 Massimo tempo di risposta	7
1.3.1 Task periodici con tempi di risposta arbitrari	7
1.4 Condizioni di schedulabilità	10
1.5 Test per sottoinsiemi di task armonici	11
2 Schedulazione di job bloccanti e job aperiodici	12
2.1 Auto-sospensione	12
2.1.1 Rallentamento dovuto all'auto-sospensione	13
2.1.2 Tempo massimo di sospensione di blocco per auto-sospensione	14
2.2 Non interrompibilità dei job	14
2.3 Cambi di contesto	15
2.4 Test di schedulabilità per job bloccanti	15
2.5 Condizioni di schedualbilità per task bloccanti +a priorità fissa .	16
2.6 Schedulazione basata su tick	17
2.6.1 Test schedulabilità per priorità fissa con tick	17
2.6.2 Condizione di schedulabilità su tick	19
2.7 Schedulazione priority-driven di job aperiodici	19
2.7.1 Schedulazione di job aperiodici soft RT in background .	20
2.7.2 Schedulazione di job aperiodici soft RT interrupt-driven .	20
2.7.3 Schedualzione di job aperiodici soft RT con slack stealing	20
2.7.4 Schedulazione di job aperiodici soft RT con polling	21
2.7.5 Server periodici	21
2.8 Algoritmi a conservazione di banda	21
2.9 Server procrastinabile	22
2.9.1 Istanti critici per server procrastinabili	23
2.9.2 Condizione di schedulabilità RM con server procrastinabile	24
2.9.3 Condizione di schedulabilità di EDF con server procras- tinabile	24
2.10 Server sporadici	25
2.10.1 Server sporadici in sistemi a priorità fissa	25
2.10.2 Server sporadico semplice	26
2.10.3 Server sporadico background	27
2.11 Constant Bandwidth server	28
2.12 Schedulabilità di job aperiodici hard real-time	30
3 Controllo d'accesso alle risorse condivise	30
3.1 Richieste e rilasci di risorse	31
3.2 Sezioni critiche	31
3.3 Controllo d'accesso alle risorse condivise	32
3.3.1 Grafi di attesa	33
3.4 Protocollo NPCS	33

3.4.1	Tempo di blocco per conflitto di risorse	33
3.5	Protocollo priority-inheritance	34
3.6	Protocollo priority-ceiling	35
3.6.1	Proprietà del protocollo priority-ceiling	37
3.6.2	Tempo di blocco per conflitto di risorse	39
3.7	Schedulabilità con priority ceiling	41
3.8	Protocollo stack-based priority-ceiling	41
3.9	Ceiling priority	42
3.10	Controllo d'accesso per job con auto-sospensione	43
3.11	Priorità dinamica	44
3.12	Accesso alle risorse di job aperiodici	44
4	Real-time su multiprocessore	45
4.1	Sistemi statici	45
4.2	Sistemi dinamici	46
4.3	Algoritmi di schedulazione multiprocessore	47
4.4	Effetto Dhall	48
4.5	Anomalie di schedulazione	48
4.6	Schedulabilità	50
4.7	Schedulazione partizionata	51
4.7.1	Allocazione dei task	51
4.7.2	RMFF	52
4.7.3	FFDU	53
4.7.4	RM-FF	53
4.7.5	EDF-FF	53
4.7.6	EDF-FFDD	54
4.8	Scheduler multiprocessore globali	54
4.8.1	EDF-US[zeta]	55
4.8.2	EDF(k)	56
4.8.3	Scheduler globali basati su priorità fissa a livello di task	57
4.8.4	Scheduler RM globale	57
4.8.5	RM-US[zeta]	57
4.9	Possibile applicabilità degli algoritmi in sistemi real-time	58
5	Sistemi operativi real-time	58
5.1	RTOS a microkernel	59
5.2	Caratteristiche chiave di un RTOS	60
5.3	Interruzioni hardware	61
5.4	Schedulazione	62
5.5	Standard per RTOS	63
5.5.1	POSIX	64
5.5.2	OSEK/VDX	64
5.5.3	ARINC 653	65
5.5.4	ITRON	65
5.6	Caratteristiche comuni dei RTOS	66
5.7	Esempi di sistemi operativi real-time	67

5.7.1	VxWorks	67
5.7.2	LynxOS	67
5.7.3	QNX Neutrino	67
5.7.4	eCos	68
5.7.5	FreeRTOS	68
5.8	Embedded	68
5.9	Zephyr	68
6	Nascita ed evoluzione dei sistemi operativi	69
6.1	Evoluzione dei CE e dei SO	69
6.1.1	Versuchsmodell-1,-2-3 e -4	69
6.1.2	Automatic Sequence Controlled Calculator	69
6.1.3	Atanasoff-Berry	70
6.1.4	Colossus	70
6.1.5	Electronic Numerical Integrator and Computer	70
6.2	Primi CE	70
6.2.1	Manchester Small-Scale Experimental Machine	71
6.2.2	EDVAC	71
6.2.3	Ferranti Mark 1/UNIVAC I	71
6.3	Uso dei calcolatori di I generazione	71
6.4	CE di II generazione	72
6.4.1	Monitor residente	72
6.4.2	I/O sovrapposto	72
6.5	III generazione	73
6.5.1	Multics	73
6.6	Unix	73
6.6.1	BSD Unix	74
6.7	IV generazione	74
6.7.1	PC IBM	75
6.7.2	Le Unix war degli anni 80'	75
7	Il software libero	76
7.1	Movimento degli hacker	76
7.2	Software libero	77
7.2.1	GNU	78
7.2.2	Open Source Initiative	79
7.3	La nascita del kernel Linux	79
7.3.1	Linux oggi	80
7.3.2	Diffusione di Linux	81
7.3.3	Chiavi del successo di Linux	82
7.3.4	La licenza GPL	82
7.3.5	Velocità di evoluzione del kernel	82
7.3.6	Linux e l'industria	83
7.3.7	Chi progetta Linux	83

8 Linux in ambito real-time	84
8.1 Limiti di Linux come hard RTOS	85
8.1.1 Gestione delle interruzioni in Linux	86
8.1.2 Schedulazione in Linux	86
8.2 Approccio mono-kernel	88
8.3 Approccio dual kernel	88
8.4 Sistemi partizionati	89
8.5 RTAI e ADEOS	90
8.6 Linux PREEMP_RT	91

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 di 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.

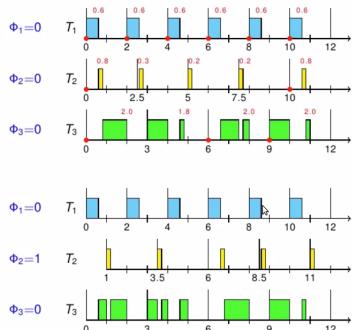
Esempio : $T_1=(2, 0.6)$, $T_2=(2.5, 0.2)$, $T_3=(3, 1.2)$.

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.

Istanti critici per $T_1=(2, 0.6)$, $T_2=(2.5, 0.2)$, $T_3=(3, 1.2)$



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, \dots, T_i e mi chiedo il tempo necessario per eseguire tutti i job dei task T_1, \dots, 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, \dots, T_i , in fase a t_0 con priorità decrescenti

con T_1, \dots, 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, \dots, 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 $k \in \{1, 2, \dots, i\}$. 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.

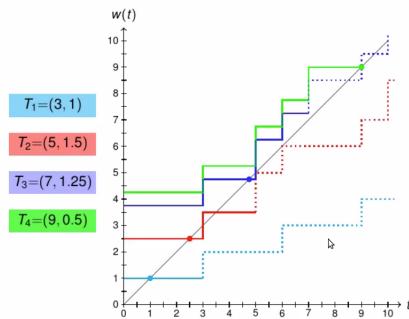
Esempio: $T_1 = (3, 1)$, $T_2 = (5, 1.5)$, $T_3 = (7, 1.25)$, $T_4 = (9, 0.5)$ e considero le funzioni di tempo necessarie:

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 $w_i(t) \leq t$ 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 : $t=w_i(t)$. Se l'equazione non ha soluzioni \leq a D_i , allora qualche job di T_i mancherà la scadenza relativa.

Uso un algoritmo:

- $t(1) = e_1$ in prima approssimazione
- Sostituisco nella funzione ed ottengo un nuovo valore $t(k+1) = w_i(t(k))$
- continuo ad iterare finché:
 - $t(k+1) = t(k)$ e $t(k) \leq D_i \Rightarrow W_i = t(k)$
 - $t(k) \geq D_i$ 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 $D_i \geq d_i p_i$
- 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 $\tau_i = T_1 \dots T_i$ 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 rilasciati 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 uguali, 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 avere il 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 $\tau_i = \{T_1, \dots, T_i\}$
- 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 $\{T_1, \dots, T_i\}$ con priorità $\pi_1 < \pi_2 \dots < \pi_i$, 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 $\tau_i = \{T_1, \dots, T_i\}$ sono in fase.
- Se il primo job di tutti i task in Tau_i termina entro il primo periodo del task \Rightarrow 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, prosegue:

$w_{2,1}(4) = 2 \cdot e_1 + e_2 = 3.25 \leq 4 \leq D_2$ quindi T_2 è schedulabile, ma ha completato oltre il periodo \Rightarrow 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.

$w_{3,2}(6) = 3 \cdot e_1 + 2 \cdot e_2 + 2 \cdot e_3 = 6 \leq p_3 + D_3 = 12$ 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: $\lim_{n \rightarrow \infty} U_{RM}(n) = \ln 2$, 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:

$T_1 = (1, 0.25)$, $T_2 = (1.25, 0.1)$, $T_3 = (1.5, 0.3)$, $T_4 = (1.75, 0.07)$, $T_5 = (2, 0.1)$.
 $U_\tau = 0.62 \leq 0.743 = U_{RM}(5) \Rightarrow$ 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 $U_\tau = 0.867 > 0.757 = U_{RM}(4)$, 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 $u_k = \frac{U_\tau}{n}$:

$$\prod_{k=1}^n \left(1 + \frac{e_k}{p_k}\right) \leq 2 \Leftrightarrow U_\tau \text{ leq } n \cdot (2^{\frac{1}{n}} - 1).$$

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, \dots, Z_{n_h} , ciascuno dei quali contiene task semplicemente periodici, allora il sistema è schedulabile con RM se:

$$\sum_{k=1}^{n_h} U_{Z_k}(n_h) \text{ oppure se } \prod_{k=1}^{n_h} (1 + U_{Z_k}) \leq 2.$$

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:

$$X_i = \log_2 p_i - \lfloor \log_2 p_i \rfloor \text{ e } \zeta = \max_{1 \leq i \leq n} X_i - \min_{1 \leq i \leq n} X_i$$

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 $U_\tau \leq a$: $U_{RM}(n, \delta) =$

- $\delta(n-1) \cdot (\frac{\delta+1}{\delta}^{\frac{1}{(n-1)}} - 1)$ per $\delta = 2, 3, \dots$
- $n(2\delta^{\frac{1}{n}} - 1) + 1 - \delta$ 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 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 sì 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

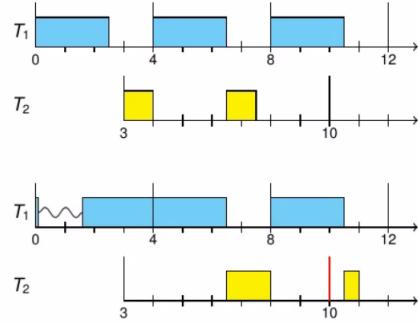
Nei SO questo tipo di operazioni sono chiamate operazioni bloccanti, nell'ambito real-time ci possono essere operazioni di auto-sospensione che però non è bloccante: in questo ambito ha senso attivo, ovvero un job ne blocca un altro. Anche in questo caso ci sono conseguenze su un altro job.

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.

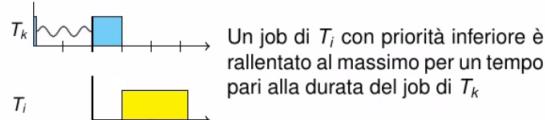
$$T_1 = (4, 2.5) \quad T_2 = (3, 7, 2, 7)$$



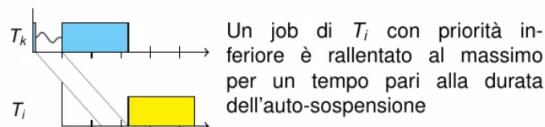
Ho impatto sui job di priorità inferiore: anche se job si auto-sospende on no lo fa, se c'è job di priorità superiore non è danneggiato, ma quelli di priorità inferiore sì.

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 mentre 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

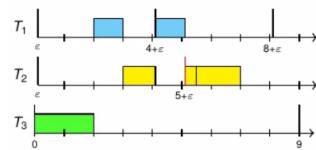
2.2 Non interrumpibilità dei job

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

- 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 interrumpibile fino alla conclusione del SO.
- costo del context switch è troppo elevato

Un job J_i è bloccato per non interrumpibilità quando è pronto per essere eseguito, ma non può perché è in esecuzione un job non interrumpibile.

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 interrumpibili. Suppongo che T_3 non sia interrumpibile, 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 interrompibilità di un job: sia

Θ_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? $b_i(np) = \max\{\Theta_k : \text{per ogni task } T_k \text{ di priorità minore a } T_i\}$: 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; $K_i = n^{\circ}$ 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; utile 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:

$$w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \cdot e_k \rceil \text{ per } 0 < t \leq \min(D_i, p_i). \text{ Ho meno tempo a disposizione per completare il job, sommo } b_i.$$

Stesso si applica al test di schedulabilità generale:

$w_{i,j}(t) = j \cdot e_i + b_i + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k$ per $(j-1) \cdot p_i < t \leq w_{i,j}(t)$. 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 schedualibilità 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 schedualabilità 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 $U_T \leq U_X(n)$, una condizione che i task non bloccino mai. Come adatto la condizione per task a priorità fissa ma che bocchino? Non posso più usare solo le condizioni di schedualabilità, 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 alla 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 approssimazione simile al precedente: task per task questo è schedualabile 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 fissa, la priorità è del job.

Teorema (Baker, 1991): in una schedulazione 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 $r < r' \Rightarrow D = d - r > d' - r' = D'$. 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 job 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.

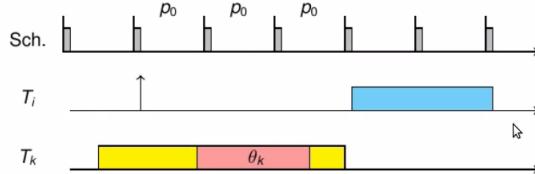
Per controllare la schedulabilità di T_i ,

- Devo aggiungere task per controllare schedulabilità di un task $T_0 = (p_0, e_0)$ a priorità massima.
- Devo modellare il fatto che quando 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 corrispondente di task $T_{0,k} = (p_k, CS_0)$ per ogni $k = i+1, \dots, n$, 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 eseguibili 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 $k = 1, 2, \dots, i$.

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 $T_0 = (1,0.05)$, $T_{0,2} = (5,0.06)$, $T_{0,3} = (20,0.06)$, $T_1 = (4,1.06)$, $b_1 = 3$.

$$w_1(t) = 1.06 + \lceil \frac{t}{1} \rceil 0.05 + \lceil \frac{t}{5} \rceil 0.06 + \lceil \frac{t}{20} \rceil 0.06.$$

$$w_1(4.06) = 4.43 \leq w_1(4.43) \leq 4.5, \text{ quindi ok.}$$

Procedo per T_2 e T_3 sempre considerando il sistema equivalente.

$$w_2(4.86) = 7.29 \leq w_2(7.29) \leq 7.5, \text{ quindi ok.}$$

$$w_3(6.06) = 12.25, w_3(12.25) = 16.53, w_3(16.53) = 19.65, w_1(19.65) = 19.8 > 19.5, \text{ quindi no.}$$

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

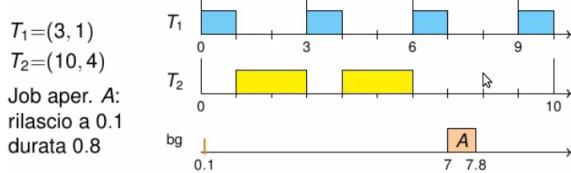
Se non ho nessuna ipotesi su tempi di arrivo ed esecuzioni non posso prendere impegni: potrà sempre arrivare qualcosa che non mi permette di rispettare le scadenze.

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

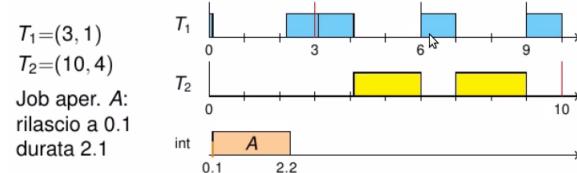
Metto in coda apposta e quando job è in background eseguo il job aperiodico in cima alla coda. Algoritmo è corretto, i task periodici non sono influenzati, ma non è ottimale: ritardo job aperiodici senza motivo.esempio:

$T_1 = (3,1)$, $T_2 = (10,4)$ job aperiodico A con rilascio 0.1 e durata 0.8.



2.7.2 Schedulazione di job aperiodici soft RT interrupt-driven

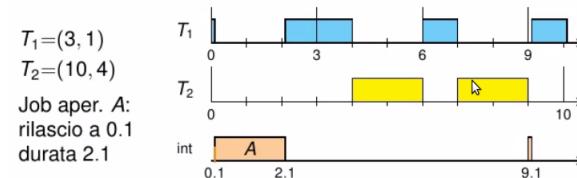
Esegui job aperiodici nel momento in cui li rilasci, algoritmo non è corretto ma è ottimo: job aperiodici finiscono nei tempi minimi, ma task periodici possono mancare le scadenze.



2.7.3 Schedulazione di job aperiodici soft RT con slack stealing

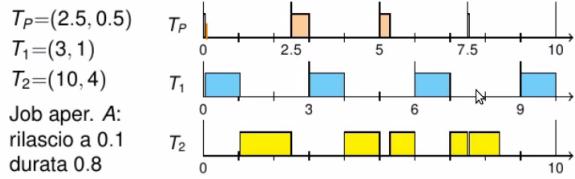
Algoritmo esegue i job aperiodici in anticipo rispetto a quelli periodici finché c'è uno slack globale positivo.

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 driven 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; la 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. VOlgo 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:

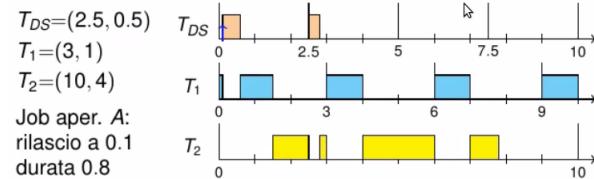
- Server procrastinabile
- Server sporadico
- Server a utilizzazione costante

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.

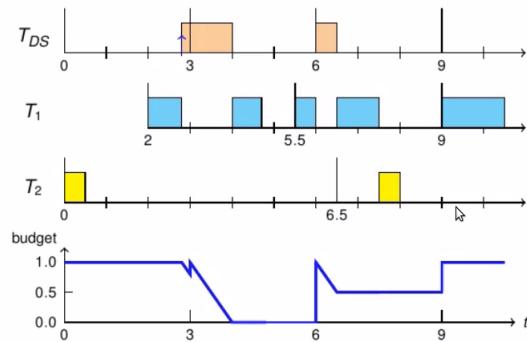
Regola di rifornimento: ad ogni istante multiplo di p_s , il budget è impostato ad e_s . Nota bene: il budget non si accumula, quello che non spendo viene perso.



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 0.8 che è il budget massimo. Questo tipo di server minimizza tempo di risposta del job aperiodico. esempio: schedulazione a priorità fissa

Sistema: $T_{DS} = (3, 1)$, $T_1 = (2.0, 3.5, 1.5, 3.5)$, $T_2 = (6.5, 0.5)$

Job aperiodico A: arrivo a 2.8, durata 1.7



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 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 $\text{budget} > 0$ può diventare eleggibile in qualunque momento. esempio: $T_{DS} = (3, 1.2)$, $T_1 = (3.5, 1.5)$, $r_{1,c} = 10$, $r_A = 10$, $e_A > 3$, $\text{budget}(10) = 1.2$, 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 azzerà 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, \dots, 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:

$$w_i(t) = e_i + b_i + e_s + \lceil \frac{t-t_0-e_s}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k \text{ per } 0 < t \leq p_i.$$

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 $t = D_i (h=0,1,\dots)$

Stesso avviene per il test di schedulabilità generale:

$$j \cdot e_i + b_i + e_s + \lceil \frac{t-t_0-e_s}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k \text{ per } 0 < t \leq p_i \text{ per } (j-1) \cdot p_i < t < w_{i,j}(t).$$

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:

$p_s < p_1 < \dots < p_n < 2p_s$ 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 $\lim_{x \rightarrow \infty} U_{RM/DS}(n) = \frac{e_s}{p_s} + \ln(\frac{e_s+2p_s}{p_s+2e_s})$.

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: $\sum_{k=1}^{i-1} \frac{e_k}{p_k} + \frac{e_s}{p_s} + \frac{e_s+b_i}{p_i} \leq U_{RM}(i+1)$.

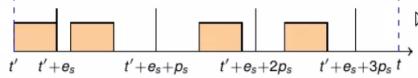
Aggiungo anche ritardo e_s in più che è il primo intervallo, il ritardo dovuto al fatto che nell'istante critico per T_1 vengo rallentato di un istante in più rispetto al tempo normale. Confronto al limite di Liu-Lyland per $i+1$ task perché includo anche il server.

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:

$$\sum_{k=1}^n \frac{e_k}{\min(D_k, p_k)} + \frac{e_s}{p_s} \cdot \left(1 + \frac{p_s - e_s}{D_i}\right) \leq 1.$$

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 manca 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 sarà il prossimo rifornimento, tipicamente sarà $t_e + \pi_s$.
- 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 $\max(t_r, \text{BEGIN})$
 - 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 eseguito 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 .

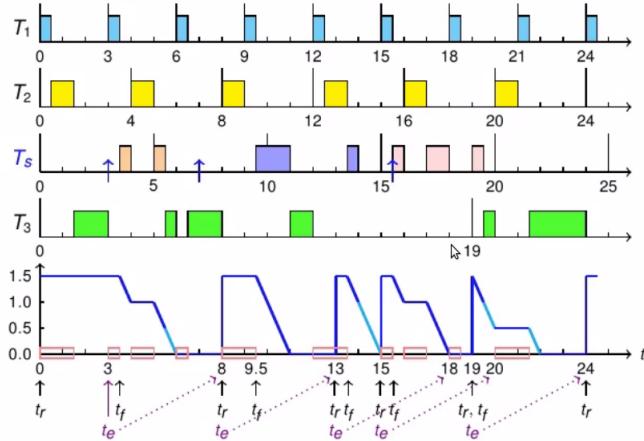
Significato di R3a: il job del server ha atteso per più di p_s unità di tempo prima di iniziare l'esecuzione, quindi il job continua nel prossimo periodo (serve il test di schedulabilità generale).

R3b: il budget è rifornito nell'istante iniziale di ogni intervallo totalmente occupato di T.

esempio di schedulazione RM con server sporadico semplice:

Sistema: $T_1=(3, 0.5)$, $T_2=(4, 1)$, $T_s=(5, 1.5)$, $T_3=(19, 4.5)$

Aperiodici: $A_1(r=3, e=1)$, $A_2(r=7, e=2)$, $A_3(r=15.5, e=2)$



Da 3.5 comincia 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

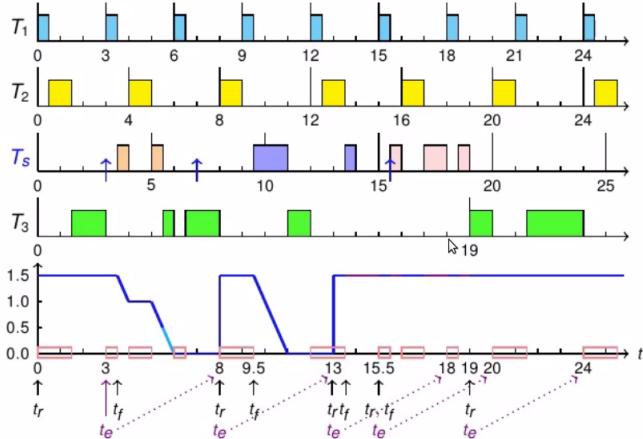
Variante del server sporadico (ne esistono di varie via via sempre più costose da implementare). Ogni volta che nessun job periodico è eseguibile, il server esegue un job aperiodico.

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 i 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:

Sistema: $T_1=(3, 0.5)$, $T_2=(4, 1)$, $T_s=(5, 1.5)$, $T_3=(19, 4.5)$
 Aperiodici: $A_1(r=3, e=1)$, $A_2(r=7, e=2)$, $A_3(r=15.5, e=2)$



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 azzerà, 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 certo 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:

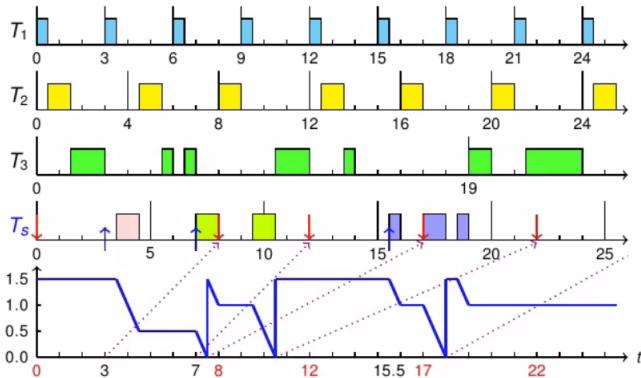
- all'inizio imposto il valore di c_s ad e_s
- c_s viene decrementato proporzionalmente all'esecuzione dei job periodici del server.
- se c_s si azzerà, c_s viene rifornito ad e_s immediatamente

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

Esempio EDF con server CBS:

Sistema: $T_s = (5, 1.5)$, $T_1 = (3, 0.5)$, $T_2 = (4, 1)$, $T_3 = (19, 4.5)$

Aperiodici: $A_1(r=3, e=1)$, $A_2(r=7, e=2)$, $A_3(r=15.5, e=2)$



intervalli	job attivi	densità
(0, 0.5]	J ₁	0.5
(0.5,1]	J ₁ J ₂	1.0
(1,2]	J ₁ J ₂ J ₃	1.5
(2,2.5]	J ₂ J 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 differente. 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, \dots, 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 job che deve acquisire un certo $n^{\circ} \eta$ 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

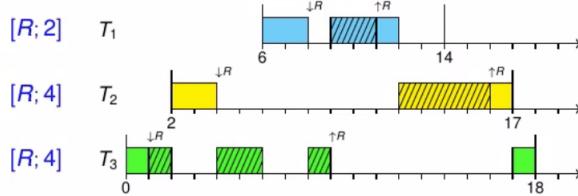
Si definisce sezione critica un segmento di esecuzione di job che inizia con $L(R_i, \eta)$ e termina con $U(R_i, \eta)$. Le richieste di risorse di un job possono essere anidate, ma assumiamo che i rilasci sono sempre LIFO.

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:

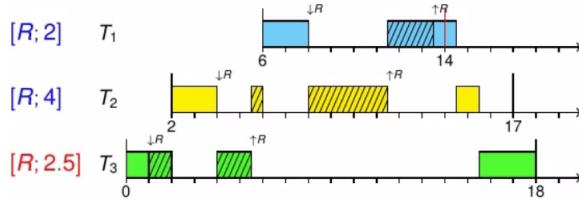
$L(R_1, \eta_1) L(R_2, \eta_2) U(R_2, \eta_2) U(R_1, \eta_1)$ rispettivamente la lunghezza di e_2 è contenuta nella la lunghezza di e_1 (ovvero nella regione critica di R_1 viene fatta la richiesta di R_2). Quando un certo job richiede una certa risorsa? Non c'è l'indicazione, ragiono sul worst case. esempio:

shedulazione con EDF con una unità di risorsa (notazione: freccia bassa è richiesta di risorsa, freccia alta è rilascio).

Task: $T_1=(6, 8, 5, 8)$, $T_2=(2, 15, 7, 15)$, $T_3=(18, 6)$
 Per T_1 e T_2 : $L(R)$ a inizio esec. +2. Per T_3 : $L(R)$ a +1



Le inversioni di priorità causata dal possedere la risorsa causa anomalie di schedulazione: se ad esempio riduco la durata della regione critica di 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.

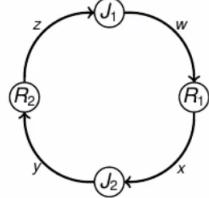
3.3 Controllo d'accesso alle risorse condivise

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:

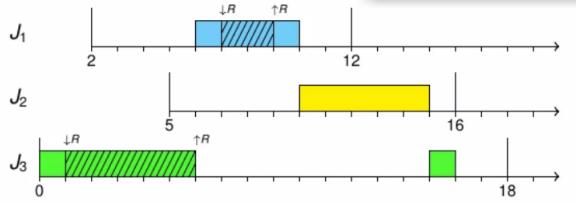


3.4 Protocollo NPCS

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 J₁ sarà bloccato da J₃ per una durata pari alla regione critica.

3.4.1 Tempo di blocco per conflitto di risorse

Sia $b_i(\text{rc})$ il tempo di blocco dovuto ad un conflitto di risorse. Per NPCS con task a priorità fissa T₁, ..., T_n:

$b_i(\text{rc}) = \max_{i+1 \leq k \leq n} (c_k)$, 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 $b_i(\text{cs})$ 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 $b_i(\text{rc}) = \max\{c_k : k \text{ tale che } D_k > D_i\}$.

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 evitare 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.

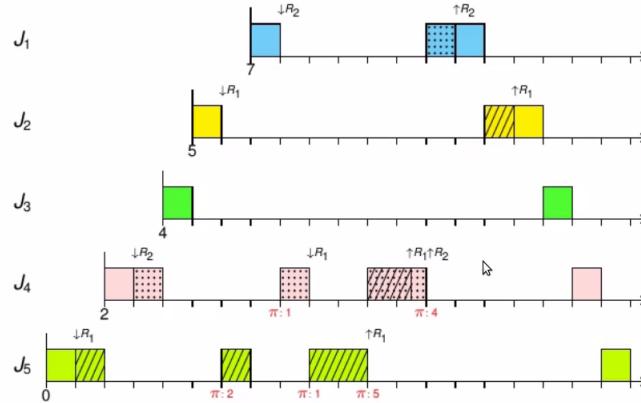
Quando un job J richiede una risorsa R al tempo t:

- Se R è disponibile, R viene 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.

	J_1	J_2	J_3	J_4	J_5	
r	7	5	4	2	0	$J_1:[R_2; 1]$
e	3	3	2	6	6	$J_4:[R_2; 4 [R_1; 1.5]]$



Limiti:

- Non evita i deadlock
- Introduce nuovi casi di blocco: un job a priorità corrente $\pi(t)$ può bloccare ogni job con priorità assegnata minore di $\pi(t)$.
- 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 sé 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 $\lceil \cdot \rceil(R)$ 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 $\lceil \cdot \rceil'(R)$ che è apri a:

- La massima priorità $\lceil \cdot \rceil(R)$ 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, $\pi(t) > \pi'(t)$ significa che $\pi(t)$ ha maggiore priorità di $\pi'(t)$; così se a valore inferiore corrisponde priorità superiore, $\pi(t) = 1$ e $\pi'(t) = 2$ implica che $\pi(t) > \pi'(t)$.

Regola di schedulazione: job schedulati in modo interrompibile secondo la loro priorità corrente.

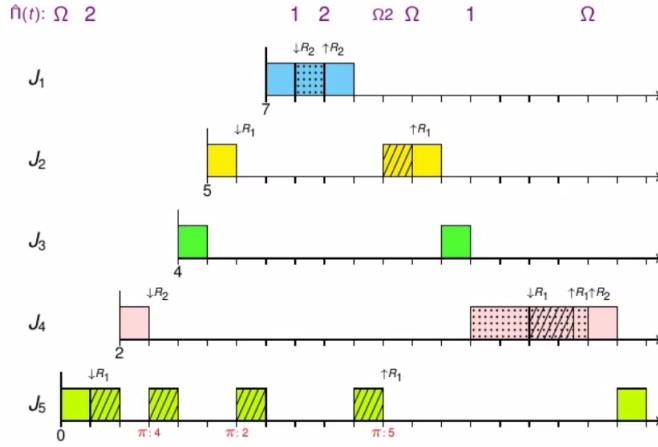
Se al tempo t un job J con una certa priorità corrente $\pi(t)$ richiede una risorsa R , R è allocata a J sole se è disponibile d inoltre:

- $\pi(t) > \lceil \rceil(t)$
- 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 $\pi(t)$ di J finché J_l non rilascia l'ultima risorsa R tale che $\lceil \rceil(R) \geq \pi(t)$; a quel punto la priorità di J_l torna ad essere la priorità $\pi_l(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R .

esempio:

	J_1	J_2	J_3	J_4	J_5		
r	7	5	4	2	0	$J_1:[R_2; 1]$	$J_2:[R_1; 1]$
e	3	3	2	6	6	$J_4:[R_2; 4 [R_1; 1.5]]$	$J_5:[R_1; 4]$



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 $\pi_l < \pi_h$:

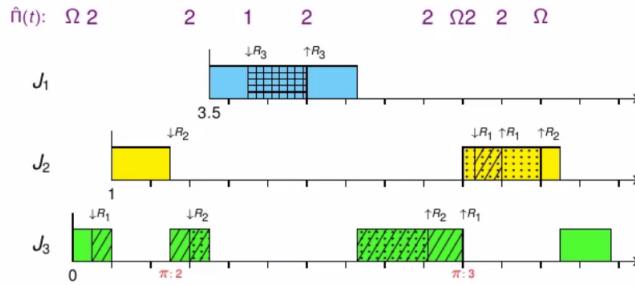
- Blocco diretto: J_h richiede una risorsa R assegnata a J_l .
- Blocco dovuto a priority-inheritance: la priorità corrente di J_l è maggiore di quella di J_h , perché J_l sta bloccando direttamente un job che ha priorità maggiore di J_h .

- 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 $\lceil\lceil(R') \geq \pi_h$.

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:

	J_1	J_2	J_3	$J_1:[R_3; 1.5]$	$J_2:[R_2; 2 [R_1; 0.7]]$	$J_3:[R_1; 4.2 [R_2; 2.3]]$
r	3.5	1	0			
e	3.8	4	6	$\Pi(R_1)=2$	$\Pi(R_2)=2$	$\Pi(R_3)=1$



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 $\lceil\lceil'(t)$ possono acquisire risorse senza 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:

$$\lceil\lceil(R_1) = \lceil\lceil(R_2) = \pi_1$$

$$\lceil\lceil(R_3) = \lceil\lceil(R_4) = \pi_2. \text{ e così via}$$

Suppongo che $\lceil\lceil'(t_0)$ 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 $\pi_J(t_0) > \lceil\lceil'(t_0)$:

- J non richiederà mai alcuna risorsa già assegnata al tempo t_0 . Quindi non avrà nessun deadlock con le risorse già assegnate
- Nessun job con priorità maggiore di $\pi_j(t_0)$ chiederà alcuna risorsa già assegnata al tempo t_0 , quindi nessun job che già possiede una risorsa al tempo t_0 potrà 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 $\lceil \pi'(t) \rceil$.
- se un job sta per ottenere una risorsa $\pi(t) > \lceil \pi'(t) \rceil$, nessun job di priorità uguale o superiore ha richiesto o richiederà le risorse già assegnate
- Se un job sta per ottenere una risorsa $\pi(t) = \lceil \pi'(t) \rceil$, il job è il possessore di tutte le risorse assegnate aventi priority ceiling uguale a $\lceil \pi'(t) \rceil$.
- i deadlock sono dunque evitati: priorità assegnate alle 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 $\pi(t_0) > \lceil \pi'(t_0) \rceil$:

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

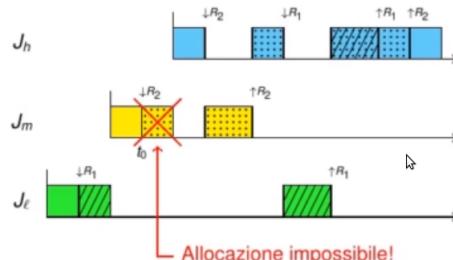
In priority-ceiling, ho 3 blocchi possibili: blocco diretto, priority-inheritance, priority-ceiling.

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:



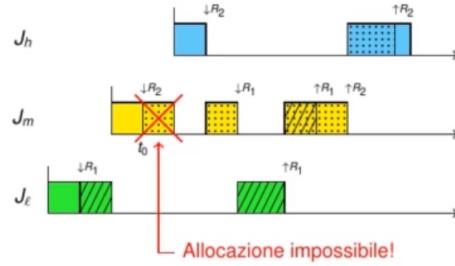
ho 3 job di priorità variabili.

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

Ora: $\lceil'(t_0) \geq \lceil(R_1) \geq \pi_h$. Il requisito per l'allocazione a t_0 deve essere $\pi_m > \lceil(R_1) \geq \pi_h$. 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 richiedere 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:

$$\pi_h > \pi_m \quad \pi_l \Rightarrow \lceil(R_1) \geq \pi_m \text{ e } \lceil(R_2) \geq \pi_h.$$

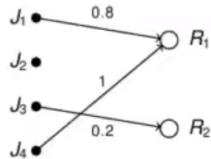
Quindi $\lceil'(t_0) \geq \lceil(R_1) \geq \lceil'(t_0)$; quindi l'allocazione non può avvenire.

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.

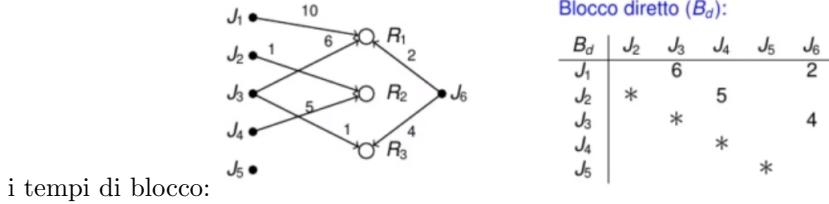
Come faccio per calcolarlo: ho 3 tipi di blocco, quindi devo calcolarlo per tutti e 3 tipi, mi il teorema mi dice che il job può essere bloccato per al massimo una sezione critica, quindi considero il massimo. esempio:

$J_1:[R_1;0.8]$, 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 $\Rightarrow b_1(\text{rc}) = 1$. J_4 può bloccare anche J_2 e J_3 , quando acquisisce $R_1 \Rightarrow b_2(\text{rc}) = b_3(\text{rc}) = 1$: 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 $b_4(\text{rc}) = 0$, perché il job di priorità più bassa.

Per esempi più complessi, conviene avere un algoritmo automatico per derivare



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'inheritance:

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 essere 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 $B_i(r,c) = \max\{B_d(j, c) : 1 \leq j \leq r-1\}$.

Se le priorità dei job sono tutte diverse, $B_c = B_i$, tranne che per i job che non utilizzano risorse.

$b_i(rc) = \max_k B_d(i, k)$, $B_c(i, k)$: considero il valore massimo per ciascuna riga, perché priority ceiling mi dice che blocca 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:

$b_i = b_i(ss) + (K_i + 1) \cdot b_i(np) + (K_i + 1) \cdot b_i(rc)$, 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: $e_i' = e_i + 2 \cdot (K_i + 1) \cdot CS + 2 \cdot (K_i + 1) \cdot CS$, 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 un 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 job richiede la stessa risorsa del job che ha interrotto: per priority ceiling deve tornare in esecuzione il job interrotto. Problema: 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 , $\lceil(R)$ 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 $\pi \leq \lceil'(t)$.

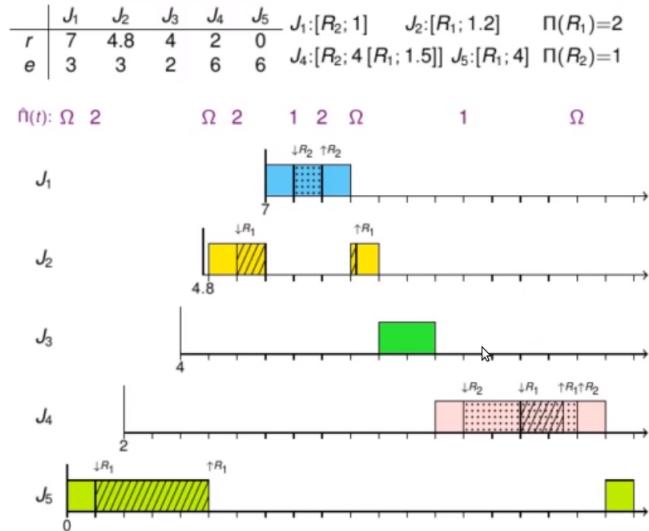
I job eseguibili sono schedulati in modo interrompibile in accordo alle priorità assegnate. Non c'è priority inheritance

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-sospenderà, 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 si 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 software 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 risorse 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 $b_i(r_c)$ 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 priorità dinamiche sono meno frequenti in priority-ceiling, che ci sono solo in caso di contesa.

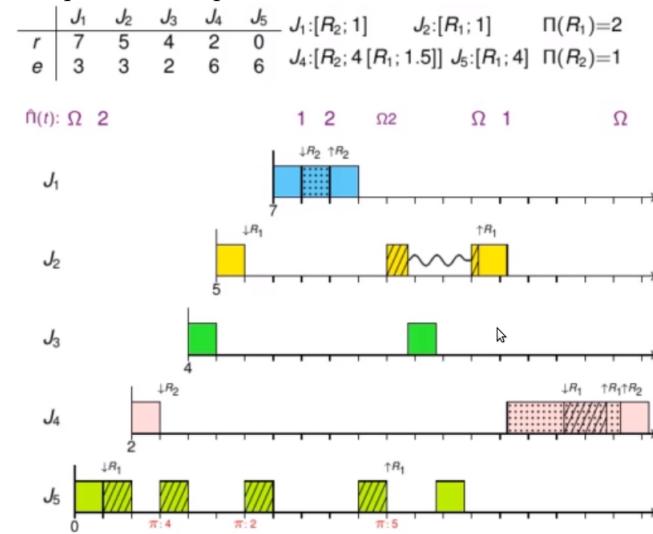
Trade off, ma spesso vincono gli ultimi due

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 $\pi(t) > \pi(t')$
- 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

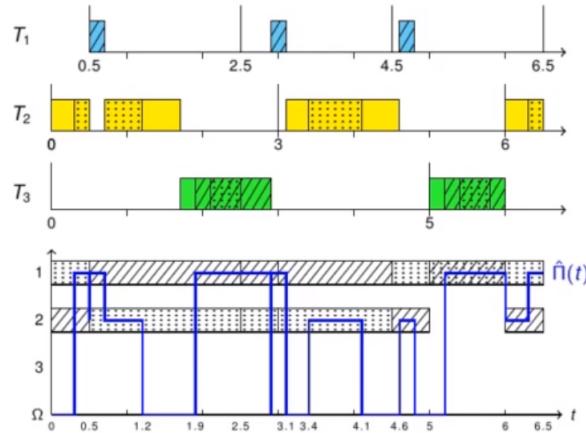
- NPCS: $b_i = b_i(\text{ss}) + (K_i + 1) \cdot \max\{b_i(\text{np}), b_i(\text{rc})\}$
- priority ceiling e ceiling priority: $b_i = b_i(\text{ss}) + (K_i + 1) \cdot (b_i(\text{np}) + b_i(\text{rc}))$. Qui i tempi di blocco $b_i(\text{rc})$ vanno calcolati anche pensando che mentre sono in sezione critica 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 priorità dovuta alla scadenza assoluta che fa cambiare i valori di priorità di tutti i job del sistema, quindi i priority 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:

$$\begin{aligned} T_1 &= (0.5, 2, 0.2, 2; [R_1; 0.2]), & T_2 &= (3, 1.5; [R_2; 0.7]), \\ T_3 &= (5, 1.2; [R_1; 1[R_2; 0.4]]) \end{aligned}$$



3.12 Accesso alle risorse di job aperiodici

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

- Esecuzione all'interno della sezione critica rende il server non interrompibile 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 multi-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 processore. Partiziono i task nel sistema tale che ciascun job o task è eseguito forzatamente da uno specifico processore: 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:

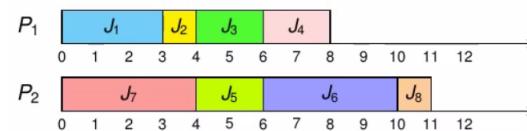
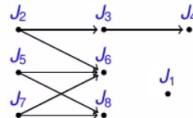
- L'assegnazione dei job è effettuata manualmente dal progettista una volta per tutte in fase di progettazione del sistema. Problema: devo conoscere tutti i parametri dei task, non posso gestire nuovi task che arrivano a runtime
- L'insieme dei task è assegnato ad uno specifico processore dal SO (o scheduler)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

Lista processore P_1 : J_1, J_2, J_3, J_4
 Lista processore P_2 : J_5, J_6, J_7, J_8

i	1	2	3	4	5	6	7	8
r_i	0	0	0	0	4	0	0	0
e_i	3	1	2	2	2	4	4	1



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 stesso 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:

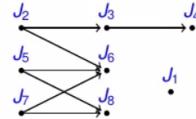
- Con job interrompibili
- Con job interrompibili e non migrabili: anche se interrotto, job è salvato in una struttura locale del processore e potrà recuperare l'esecuzione solo su quel processore
- Con job interrompibili e migrabili: job può essere migrato se interrotto, ha un costo

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

1° caso: job interrompibili e migrabili:

Lista: J_1, J_2, \dots, J_8

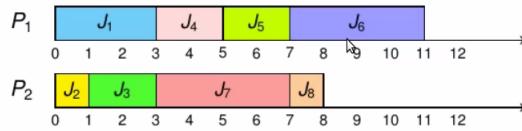
i	1	2	3	4	5	6	7	8
r_i	0	0	0	4	0	0	0	0
e_i	3	1	2	2	2	4	4	1



Job interrompibili e migrabili:



Job non interrompibili:



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 switch. 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 del 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

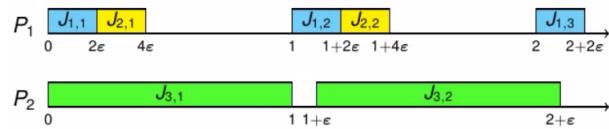
Gli algoritmi clock-driven sono in generale utilizzabili senza problemi, infatti la schedulazione avviene offline e validata una volta per tutte. Metodo però poco flessibile, ma non è immediato applicare algoritmi priority-driven, devo studiare bene come fare: devo risolvere diversi problemi

- 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 $T_1=81, 2\epsilon)$, $T_2=(1,2\epsilon, \dots, T_m=(1,2\epsilon)$, $T_{m+1}?(1+\epsilon, 1)$. Utilizzazione globale = $2\epsilon \cdot m + \frac{1}{1+\epsilon} \rightarrow 1$ se $\epsilon \rightarrow 0$. 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 sistemi 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

Comportamenti 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:

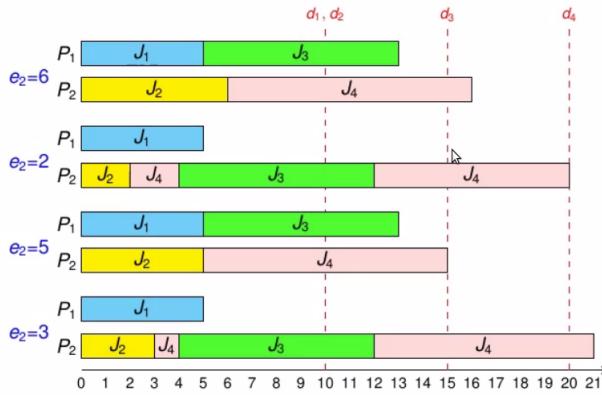
- Sistema statico, job non interrompibili: posso avere anomalie
- Sistema statico, interrompibili: non ho anomalie

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

	i	1	2	3	4
r_i		0	0	4	0
d_i		10	10	15	20
e_i		5	[2, 6]	8	10

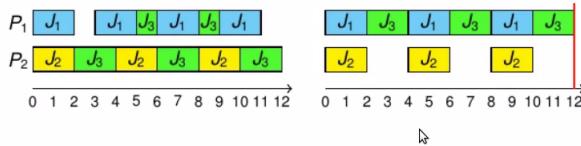
Indice minore \equiv priorità maggiore
 e_2 varia da 2 a 6



anomalie di schedulazione con job migrabili:

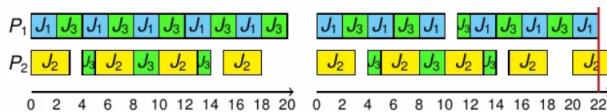
Aumento del periodo di un task di priorità alta:

$$T_1 = (3, 2), T_2 = (4, 2), \quad T_1 = (4, 2), T_2 = (4, 2), \\ T_3 = (12, 8) \quad T_3 = (12, 8)$$



Aumento del periodo di un task di priorità bassa:

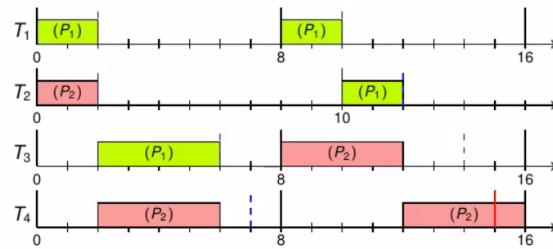
$$T_1 = (4, 2), T_2 = (5, 3), \quad T_1 = (4, 2), T_2 = (5, 3), \\ T_3 = (10, 7) \quad T_3 = (11, 7)$$



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, \dots, T_{i-1} non è necessariamente un istante critico di T_i
esempio:

$$T_1 = (8, 2, 2), T_2 = (10, 2, 2), T_3 = (8, 4, 6), T_4 = (8, 4, 7), m=2$$



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...

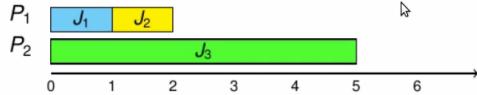
Corollario: nessun algoritmo di schedulazione globale con priorità fissa a livello di job è ottimale su multiprocessore. Non posso sfruttare al 100% tutti i processori del sistema.

esempio:

Schedul. EDF di $T_1 = (1, 1)$, $T_2 = (2, 1)$, $T_3 = (5, 5)$, $m = 2$:



Eppure una schedulazione fattibile non EDF esiste:



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 $\lceil \frac{te_i}{pi} \rceil$ o $\lfloor \frac{te_i}{pi} \rfloor$.
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 processore utilizzando un determinato algoritmo di 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 schedabilità 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 determinare
- 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 schedutabili 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 decrescenti T_1, \dots, T_n
- ordina arbitrariamente i processori: P_1, \dots
- comincia da T_1 , assegna ciasun 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.

$$URMFF = m \cdot (\sqrt[3]{2} - 1).$$

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 decrescenti T_1, \dots, T_n
- ordina arbitrariamente i processori: P_1, \dots
- 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: P_1, \dots
- 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

$$U_{RM-FF} = m \cdot (\sqrt[3]{2} - 1)$$

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: P_1, \dots
- 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

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

Fattore di approssimazione: 1.7 (meno del doppio del n° processori ottimali). È un algoritmo ottimale fra tutti gli 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 $\beta \rightarrow \infty \Rightarrow U_{EDF-FF} \rightarrow m$, per task di dimensioni infinitesime e quindi schedulabili in maniera fluida, l'algoritmo riesce a raggiungere il 100% di utilizzazione dei processori del sistema.

4.7.6 EDF-FFDD

È possibile estendere gli algoritmi partizionati basati su "first fit" anche a task sporadici con scadenze arbitrarie.

EDF-FFDD:

- Ordina arbitrariamente P_1, P_2, \dots
- Ordina i task sporadici per densità $\frac{e_i}{\min(p_i, d_i)}$ decrescenti
- 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 risulti ancora schedulabile secondo EDF

Condizione di schedulabilità:

- $\Delta_T \leq m - (m-1) \cdot \Delta_{max}$ se $\Delta_{max} \geq \frac{1}{2}$
- $\Delta_T \leq \frac{m}{2} + \Delta_{max}$ se $\Delta_{max} \leq \frac{1}{2}$

4.8 Scheduler multiprocessore globali

Ha senso considerare scheduler globali non partizionati? Assumo tutti i job indipendenti, migrabili, interrompibili e senza auto-sospensione.

Ci sono anomalie di schedulazione, ma è utile rispetto ad EDF partizionato? Questo raggiunge l'ottimo, quindi mi chiedo se ne vale la pena, problema inoltre di istanti critici ed effetto Dhall.

Anomalie di schedulazione: il tempo di esecuzione dei task è worst case, teorema del 1994: sistemi di task periodici interrompibili e migrabili su multiprocessore schedulati con algoritmi a priorità fissa (a livello di task o job) sono predicibili, ovvero non presentano anomalie di schedulazione dipendenti dal tempo di esecuzione dei job.

Risolvo la più grave anomalia, ce ne sono altre.

Utilità rispetto ad EDF partizionato: so che EDF partizionato può ottenere l'ottimo, quindi perché considera algoritmi globali.

EDF ottimale tra gli scheduler partizionati basati su priorità fissa, ma non in senso assoluto: esistono algoritmi globali a priorità dinamica che raggiungono l'ottimo m , esempio Pfair.

Teorema: un sistema di task sporadici con scadenze arbitrarie è schedulabile con m processori con algoritmo globale a priorità dinamica a livello di job se:

$$\Delta_T = \sum_i \frac{e_i}{\min(d_i, p_i)} \leq m \text{ e } \Delta_{max} = \max_i \frac{e_i}{\min(p_i, d_i)} \leq 1.$$

Corollario: se le scadenze sono implicite, allora con un sistema di m processori ed un algoritmo globale a priorità dinamica a livello di job posso schedulare se:

$$U_T = \sum_i \frac{e_i}{p_i} \leq m \text{ e } U_{max} = \max_i \frac{e_i}{p_i} \leq 1.$$

Istanti critici: il rilascio in fase di tutti i job non corrisponde necessariamente ad un instante critico. Non posso usare la funzione di tempo necessario per analizzare se un sistema di task è schedulabile su un certo n° di processori. Uso le condizioni di schedulabilità legate al fattore di utilizzazione o alla densità del

task.

Effetto Dhall: è un problema che si verifica nei sistemi multiprocessore, per cui dato un qualunque n ° di task che hanno utilizzazione totale costante rispetto ad m che non possono essere schedulati né con RM/DM né con EDF. L'effetto Dhall è condizionato dall'esistenza di un task che ha utilizzazione vicina all'unità, quindi posso cercare di correlare l'utilizzazione massima dei vari task con la schedulabilità del sistema.

Teorema: un sistema T di task sporadici con scadenze implicite, utilizzazione totale è U_T e suppongo che valore dell'utilizzazione del task a priorità massima è U_{max} , allora il sistema è schedulabile con EDF globale su un sistema con m processori se:

$U_T \leq m - (m-1) \cdot U_{max}$. Teorema va letto "all'opposto": se $U_{max} = \frac{1}{2}$ allora ho come soglia di utilizzazione $\frac{m+1}{2}$, ovvero il limite dei sistemi a priorità fissa a livello di job, partizionati. Qui, se $U_{max} \leq \frac{1}{2}$, la soglia tende ad m , quindi sistema può essere scheduabile anche se la soglia tende al numero totale di processori del sistema. Casi limite:

- $U_{max} = 1 \rightarrow U_T \leq 1$, Effetto Dhall
- $U_{max} = 0 \rightarrow U_T \leq m$

Estensione del teorema per scadenze arbitrarie, teorema: Un sistema di task T , sporadici con scadenze arbitrarie, densità totale Δ_T e densità massima dei task Δ_{max} è schedulabile con EDF globale su un sistema con m processori se:

$$\Delta_T \leq m - (m-1)\Delta_{max}.$$

Esistono diverse altre condizioni di schedulabilità per EDF globale basate sulla densità dei singoli task e/o sul calcolo del tempo di processore richiesto dai task. È possibile avere algoritmi di schedulazione a priorità fissa (a livello di task o job) migliori di EDF globale per insiemi di task generici?

Considero EDF partizionato: ha una garanzia di schedulabilità che è $\frac{m+1}{2}$, certamente migliore della garanzia di schedulabilità di EDF globale, in quanto in quest'ultimo caso la garanzia è 1. Per insieme di task grandi, EDF partizionato va meglio, ma in generale gli algoritmi globali potrebbero andare bene o anche meglio di quelli partizionati per insiemi di task piccoli. Posso trovare algoritmi ibridi che vadano meglio di quelli visti fin ora?

4.8.1 EDF-US[zeta]

EDF-US[ζ] è un algoritmo ibrido, proposto nel 2002, ζ è un parametro ≤ 1 . Una parte dei task ha priorità fissa, questi sono i task T_t tali che $\frac{e_i}{p_i} > \zeta$, quindi sono tutti i task grandi.

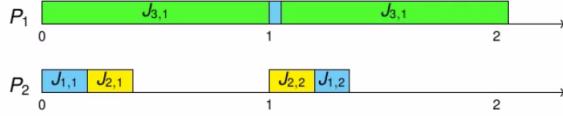
Tutti hanno identica priorità, perché l'idea è che ζ è minore del numero di processori del sistema, quindi i task potranno sempre essere seguiti in parallelo.

Gli altri task hanno priorità più piccola dei task a priorità fissa e sono schedulati mediante EDF: la priorità non è più del task, ma del job. Hanno però priorità inferiore, quindi verranno sempre interrotti da quelli a priorità fissa. Idea: task grandi prendono sempre i processori disponibili, task piccoli prendono quello che

rimane.

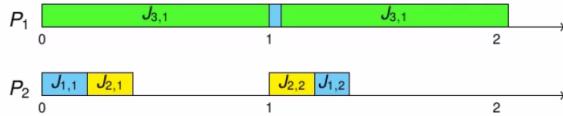
Non ho effetto Dhall: se ho task grandi, prendo processore, uso algoritmo globale per schedulare i task piccoli, quindi ebito l'effetto Dhall.
esempio:

Schedulazione con EDF-US[1/2]:



Classico effetto Dhall se schedulo con EDF globale

Schedulazione con EDF-US[1/2]:



Task T₃ ha utilizzazione $> \frac{1}{2}$, quindi ha priorità fissa ed avrà precedenza.

Prestazioni:

teorema: Un sistema di task sporadici implicite è schedulabile secondo EDF-US $[\frac{m}{(2m-2)}]$ su m processori è $U_T \leq \frac{m^2}{(2m-2)}$.

Corollario: Un sistema di task sporadici con scadenze implicite è schedulabile con EDF-US $[\frac{1}{2}]$ su m processori se $U_T \leq \frac{m+1}{2}$. Ottengo il meglio, continuano a valere le ipotesi del teorema visto in precedenza (Andersson).

Posso esistere algoritmi di schedulazione globale a priorità fissa migliori di EDF-US $[\frac{1}{2}]$: non devo confondere la garanzia di schedulazione data da questa condizione con l'effettiva capacità di schedulare sistemi di task generici

4.8.2 EDF(k)

Algoritmo ibrido, variante del precedente: k è un parametro, $k < m$. Una parte dei task ha priorità fissa, i primi $k-1$ con fattore di utilizzazione $\frac{e_i}{p_i} >$ più alti; tutti gli altri hanno priorità dinamiche e sono schedulati con EDF.

L'idea è sempre la stessa: assegno i task grandi ai processori disponibili, gli altri li schedulo con EDF. Garanzie di schedulabilità:

Teorema: Un sistema di task sporadici con scadenze implicite è schedulabile con EDF(k) su m processori se: $(k-1) + \lceil \frac{U_t - u_k}{1-u_k} \rceil \leq m$; dove u_k è il fattore di utilizzazione del k-esimo task.

Algoritmo EDF(k_{min}): Sia fissato m (in numero di processori), e sia k_{min} il minimo valore di k che soddisfa la condizione del teorema precedente. Un sistema di task sporadici con scadenze implicite è schedulabile con EDF(k_{min}) se $U_T \leq \frac{m+1}{2}$.

Vale quanto detto prima: l'insieme dei task schedulabili con EDF-US $[\frac{1}{2}]$ è un sotto-insieme proprio di EDF(k_{min}). Se guardo insieme di task con utilizzazione

maggiori di $\frac{m+1}{2}$ EDF(k_{min}) schedula più job rispetto ad EDF-US[$\frac{1}{2}$], non confondere con la garanzia di schedulabilità.

4.8.3 Scheduler globali basati su priorità fissa a livello di task

Questo tipo di algoritmi non può essere migliore di algoritmi globali o ibridi a priorità dinamica a livello di task, questo perché l'algoritmo sottostante che uso su un sistema mono-processore non può andare meglio. Ho visto che algoritmi partizionati vanno meglio in generale su insiemi di task generici (per via dell'effetto Dhall) di algoritmi globali. Il discorso della partizione in insiemi di task viene recuperato dagli algoritmi ibridi: ho un certo criterio per assegnare la priorità massima, i restanti task continuo a schedularli con un criterio differente, la priorità è legata all'algoritmo sottostante. Priorità è fissa perché è dovuta ad un condizione specifica, algoritmi globali a priorità fissa soffrono comunque dell'effetto Dhall, non possono andare meglio di algoritmi partizionati, e lo stesso vale per algoritmi ibridi: faccio assegnazione dei task a priorità massima che non dipende dall'algoritmo sottostante, quando schedulo i restanti job, schedulare con priorità fissa non può dare vantaggio rispetto a schedulare co priorità dinamica.

Questi algoritmi posso essere adottati:

- Semplici da implementare
- Presenti su tutti i SO real-time
- Sono più robusti in caso di job che superano il WCET calcolato in fase di design.

4.8.4 Scheduler RM globale

Teorema: un sistema di task periodici con scadenze implicite tale che $U_{max} \leq \frac{m}{(3m-2)}$ è schedulabile con RM su m processori se: $U_T \leq \frac{m^2}{(3m-1)}$. Tiene conto dell'effetto Dhall, limita l'utilizzazione del task più ingombrante a $\frac{m}{(3m-2)}$, se la condizione vale l'effetto Dhall è evitato e posso aumentare il carico del sistema pur di aumentare il n° di pressori, sto andando sempre in perdita.

Teorema: un sistema di task sporadici con scadenze implicite è schedulabile globalmente con RM su m processori se $U_T \leq \frac{m}{2} \cdot (1 - U_{max}) + U_{max}$.

Posso anche riformulare i teoremi in termini di densità, ma le cose si complicano parecchio.

4.8.5 RM-US[zeta]

RM-US[ζ] è un algoritmo ibrido analogo a EDF-US[ζ], la differenza è che tutti i task che hanno utilizzazione inferiore alla soglia ζ sono schedulati con RM. Stessa idea: do priorità massima ai task ingombranti. Prestazioni:

teorema: Un sistena di task peridoti con scadenze implicite è schedulabile con RM-US[$\frac{m}{(2m-2)}$] su m processori se: $U_T \leq \frac{m^2}{(3m-2)}$

Teorema (2003): Un sistema di task sporadici con scadenze implicite è schedulabile secondo RM-US[$\frac{1}{3}$] su m processori se: $U_T \leq \frac{(m+1)}{3}$. Passare da EDF a RM fa sì che $\frac{2}{3}$ dei processori rimangono inutilizzati.

Il miglior valore di soglia è $\zeta = 0.37482$, con utilizzazione massima pari a $0.37482 \cdot (m+1)$, più restrittivo del 0.3333... del risultato precedente, ma così ho utilizzazione ottima.

4.9 Possibile applicabilità degli algoritmi in sistemi real-time

Algoritmi sono applicabili, ma problema grande nei sistemi multiprocessore è che gli algoritmi sono basati su ipotesi fondamentali:

indipendenza dei task fra di loro: inserire il discorso dei vincoli di dipendenza complica molto il discorso, non ho grandi risultati già stabiliti e consolidati. Problema è ancora più serio: i vincoli di dipendenza fra task potrebbero essere nascosti. Esempio: sistema reale multiprocessore, diversi core e diversi processori, quindi sistema parallelo con multiprocessore a loro volta multicore. Dal punto di vista hw, i core condividono risorse hardware: posso avere unità ALU individuale, am non è detto che non usino una singola MMU, singola cache L2/L3, processori diversi possono avere dipendenze legate ai tempi di accesso alla RAM o al bus di sistema. WCET: tempo massimo di esecuzione quando job esegue da solo, ma se lo eseguo su sistema multiprocessore, questo verrà comunque rallentato dagli altri job su altri core perché magari c'erano ipotesi di indipendenza sbagliate. Le assunzioni di base sul worst case è falso, se presento sistema multicore a ente certificatore, non verrà mai accettato, perché è falsa assunzione di fondo del fatto che job su processori diversi non abbiano differenze. Punto cruciale della ricerca: realizzazione di sistemi single core equivalenti: sistemi che operano sui singoli core come se fossero da soli, ma in realtà sono in un sistema multi-core.

Per avere certificazione: nel sistema di task bisogna prevedere task a priorità massima che entri in esecuzione periodicamente e spenga tutti i processori nel sistema tranne 1, non voglio che più di un processore sia attivo, altrimenti no bollino certificazione real-time.

Ricerca in divenire, c'è ancora problema di applicabilità in pratica dei risultati.

5 Sistemi operativi real-time

Obiettivi di un sistema operativo:

- Fa funzionare i driver, interazione con l'hardware, usando apposite procedure detti driver
- Controllo dell'uso delle risorse hardware (massimizzazione durata batteria, salva schermo)
- Costruzione di un'astrazione dell'architettura fisica, ambiente di esecuzione delle applicazioni

- Offrire interfacce di accesso ai dispositivi hardware, utilizzo in maniera controllata
- Distribuisce le risorse tra le varie applicazioni/utenti
- Implementa servizi di comunicazione tra applicazioni locali o remote
- Nei SO multitasking (tutti praticamente ormai) è possibile eseguire contemporaneamente più processi, shcduler assegna CPU di volta in volta.
- Consente accesso a più utenti in contemporanea

Questi sono SO general purpose. In cosa differiscono da un SO real-time? Di fatto, la reale differenza sta nel tipo di applicazioni che i due SO devono eseguire: un applicazione real-time ha dei vincoli temporali da rispettare, ed anche il SO va progettato intorno all'idea che le applicazioni devono poter rispettare questi vincoli. SO real-time devono fare in modo che l'applicazione sia affidabile e predicibile.

Talvolta, le applicazioni real-time devono avere dei tempi di risposta rapidi, ma non è una proprietà caratterizzante. Questo non è però il punto cruciale dei SO real-time o delle applicazioni real-time, sono caratterizzate dal fatto che le scadenze vengono rispettate, qualunque esse siano.

So che la maggior parte dei sistemi real-time sono embedded, quindi sistemi operativi rea-time devono essere anche embedded, applicazioni devono quindi essere:

- Compatte
- Scalabili
- Consumo ridotto delle risorse, viste le esigenze dei sistemi embedded

5.1 RTOS a microkernel

Molti SO per sistemi embedded e RT sono basati su un modello a micro-kernel. SO real-time è un supporto generico ad applicazioni real-time, deve far si che funzioni correttamente e tutti gli altri compiti del SO, ma l'obbiettivo prioritario è il rispetto delle scadenze, quindi spesso molte delle feature del SO general purpose vengonoificate.

Micro-kernel è un piccolo programma che realizza pochi servizi essenziali, parte del codice del SO che esegue a livello di privilegio più alto possibile. Non tutte le parti del SO eseguono alla priorità massima, alcune potrebbero eseguire a livello di privilegio più basso. Feature del micro-kernel:

- driver dei circuiti, HW di base (tipicamente solo delle periferiche fondamentali)
- schedulazione dei processi
- comunicazione bi base fra i processi

Tutti gli altri servizi sono fatti al di fuori del kernel (driver delle periferiche, stack di rete, file system etc...) Approccio differente al kernel monolitico: a livello user mode girano librerie di sistema per supporto ad applicazioni, tutto il resto viene eseguito in kernel mode.

Vantaggi del micro-kernel ai SO real-time: è approccio elegante, ma di base inefficiente perché applicazione moderna deve fare molte cose: interazione con FS, interazione con la rete etc..., bisogna scambiare informazioni e molti dati fra processi, nell'approccio micro-kernel le applicazioni per fare una semplice cosa devono coinvolgere decine e decine di processi utente separati.

Qui la velocità non è più primaria: mentre quando progetto SO general purpose voglio utilizzo efficiente da parte del SO delle risorse, mentre nel SO real-time no, voglio rispettare le scadenze. Il micro-kernel ha poco codice, quindi è più semplice verificarne la correttezza, ma poi devo anche verificare codice es dell'applicazione di rete: ma un conto è verificare il codice alla massima priorità, un conto è partire da kernel corretto e verificare codice a livello utente: uniche ripercussioni sono relative all "gabbia" costruite intorno al processo.

Struttura del kernel monolitico: tipici esempi sono tutti i sistemi derivati da Unix, quindi Linux, FreeBSD, SunOS, Solaris.

SO a micro-kernel: QNX, GNU Hurd (Mach & Four, BeOS,...

Sistemi ibridi: partiti come micro-kernel, ma dopo avere valutato le prestazioni e aver visto che erano pessime, hanno portato componenti da user mode a kernel mode. Approccio ibrido, esempi: Windows NT, Mac OS X,...

5.2 Caratteristiche chiave di un RTOS

Un sistema operativo che opera in contesto real-time ed embedded deve offrire:

- Predicibilità: risposte agli eventi esterni predicibile, cura nella gestione degli interrupt hardware.
- Risposte efficienti ad eventi esterni, bassa latenza nei tempi di risposta (non nei tempi di risposta dei task dei job).
- Gestione affidabile e precisa di eventi temporali:
 - gestione dei timer e clock hardware con le relative interruzioni
 - gestione del tempo di sistema
 - gestione di allarmi
- Deve poter avere un modo per schedulare i task, deve essere a basso overhead e predicibile:
 - implementazione di algoritmi deterministici
 - supporto al partizionamento dei task in sistemi multiprocessore: problemi teorici legati ai sistemi globali sono ancora in analisi.
- Gestione della comunicazione e sincronizzazione dei task:

- memoria condivisa, code, segnali
- primitive di sincronizzazione come semafori, lock.

- Gestione della memoria

- memoria virtuale: ogni task nel sistema, quando usa indirizzo di memoria fa riferimento ad una cella differente da quella che usa un altro task con lo stesso indirizzo, non è detto sia necessario per sistemi con poca memoria/embedded
- protezione dello spazio di indirizzamento, discorso diverso dalla memoria virtuale. L'accesso alla memoria da parte di un task verso le celle di un altro task è proibito, possibile realizzare protezione di memoria senza che ci sia memoria virtuale. Più importante in un sistema real-time rispetto ad avere memoria virtuale

Spesso le applicazioni embedded per sistemi real-time sacrificano la memoria virtuale e spazi di indirizzi separati: il software deve passare per analisi talmente tanto strette che l'accesso a memoria che non compete sarà semplice da verificare. Problemi più importanti: se verifco che non ci sono indipendenze tra i task ,automaticamente nessuno fa ciò che non deve e quindi altre feature divengono overhead.

5.3 Interruzioni hardware

Ogni occorrenza di una interruzione hardware deve essere gestita con procedura apposita livello kernel o user mode. Deve essere rapido il modo in cui il SO riconosce l'interrupt ed agisce per gestirlo: quando dispositivo hw genera interrupt, blocca la sua attività, quindi finché CPU non riconosce l'interruzione l'hw è congelato, poi la gestione dell'interrupt può essere fatta in tempi più lunghi. Il tempo richiesto per la completa gestione dell'interrupt dipende dalla periferica hardware:

- La lettura dei dati da un sensore è breve
- Traferire di grandi blocchi di dati, ad esempio dalla scheda di rete alla memoria o dalla memoria secondaria alla RAM.

In ogni caso, le interruzioni devono essere riconosciute nel minor tempo possibile, altrimenti le periferiche HW potrebbero mal funzionare. 1 linea di interruzione fisica di interruzioni in ogni dispositivo, una sola linea verso il processore, l'interruzione arriva al PIC ed il PIC asserisce la linea verso la CPU e dopo la conferma della CPU il PIC conferma la ricezione al dispositivo, il driver del dispositivo gestirà con apposita routine la situazione.

Gestione delle interruzioni hardware in più fasi:

- Interrupt handler:
 - Priorità elevata

- da conferma della ricezione dell'interruzione al PIC
- salva e recupera il contesto di esecuzione del processore
- Interrupt service routine (IRS), specifica del dispositivo:
 - priorità più bassa (ma generalmente superiore ai processi di sistema)
 - esegue operazioni specifiche per l'interruzione ed il dispositivo che l'ha generata.

5.4 Schedulazione

Uno dei punti fondamentali, devo assicurare che le applicazioni real-time rispettino le scadenze. Bisogna usare politiche di schedulazione facili, predibili, tipicamente EDF.

SO real-time non supportano nessun tipo di analisi di schedulabilità o test di accettazione on-line per nuovi task. È il progettista che deve prevedere tutti i possibili carichi del sistema e vedere se qualcuno di questi missa le scadenze. Analisi è fatta dal progettista ed avallata dai test del sistema.

Schedulazione tipicamente preemptive, quindi job interrompibili, perché altrimenti cominciano ad avere anomalie di schedulazione. Possibile specificare che alcuni job non siano interrompibili, di solito non job collaborativi, ovvero rilasciano periodicamente il processore per far sì che venga utilizzato da altri job; il codice va analizzato dal progettista. Scheduler quasi sempre clock-driven, va considerato nell'analisi di schedulabilità. Generalmente, ogni SO offre livelli di priorità finiti: se ho più task del numero dei livelli di priorità? Perdita di schedulabilità: devo dire che due task hanno la stessa priorità e questo può comportare problemi.

Perdita di schedulabilità:

ho modellato un sistema e stabilito che mi servono Ω_n livelli di priorità, ma nel sistema ne posso avere solo $\Omega_s < \Omega_n$. Devo fare mapping fra le priorità assegnate a quelle di sistema π_1, \dots, π_s dove: $\pi_i \in \{1, 2, \dots, \Omega_n\} \forall i$. I π_i sono ordinati in maniera crescente ed indicano come mappo le priorità reali: tutte le priorità assegnate numericamente, minori o uguali a π_i sono mappate su π_i . In generale, tutte le priorità assegnate tra $\pi_{k-1} + 1$ e π_k sono mappate su $\pi_k \forall 1 < k \leq \Omega_s$. Devo tenere conto di questo mapping nell'analisi di schedulazione, devo farlo task per task:

considero task T_i , siano $T_E(i)$ e $T_H(i)$ gli insiemi dei task di priorità uguale (il primo) e superiore a T_i , rispettivamente:

$$w_i = e_i + b_i + \sum_{T_k \in T_E(i)} e_k + \sum_{T_k \in T_H(i)} \lceil \frac{t}{p_k} \rceil e_k. \text{ Aggiungo nel caso peggiore che i}$$

task di priorità uguale portano via tempo, resta il fatto che lo scheduler schedula i task in modalità FIFO: se due task hanno pari priorità, uno può essere rallentato dal fatto che arriva dopo ad un altro di pari priorità, ma quando quello ha finito tocca a lui.

$$w_{i,j}(t) = j \cdot e_i + b_i + \sum_{T_k \in T_E(i)} (\lceil \frac{(j-1)p_i}{p_k} \rceil + 1)e_k + \sum_{T_k \in T_H(i)} \lceil \frac{t}{p_k} \rceil e_k. \text{ Applico lo stesso alla funzione di tempo necessaria per tempi arbitrari, stavolta considero}$$

dal rilascio del job che sto guardando in poi. Devo calcolare quanti job con la mia stessa priorità vengono rilasciati nell'intervallo che sto considerando (+1 è rilascio iniziale).

Associazione a rapporto costante: lo scheduler potrebbe non distinguere i job di priorità più alta. Si utilizza quindi in genere un'assegnazione che riserva i livelli di priorità di sistema ai livelli di priorità assegnata più alti e mano a mano accorda le priorità assegnate di livello più basso.

In pratica, si può cercare di mantenere approssimativamente costanti i rapporti: $g_k = \frac{\pi_{k-1}+1}{\pi_k}$ ($1 < k \leq \Omega_s$), maniera di assegnare le priorità migliore.

Teorema: Per l'algoritmo di scheduling RM, con scadenze relative pari al periodo e numero n di task elevato, usando l'associazione a rapporto costante $g = \min_{1 < k < \Omega_s} g_k$, allora U_{RM} :

- $\ln(2g) + 1 - g$ se $g > \frac{1}{2}$
- g se $g \leq \frac{1}{2}$

La schedulabilità relativa indica la perdita di schedulabilità dovuta ad un numero limitato di livelli di priorità nel sistema: $\frac{U_{RM}(g)}{\ln(2)}$. Casi limite:

per $g = 1 \frac{U_{RM}(g)}{\ln(2)} = 1$, quindi nessuna perdita

$g = \frac{1}{2} \frac{U_{RM}(g)}{\ln(2)} = \frac{1}{2\ln 2}$, quindi perdita del 28%. Non dare per scontato che adattare il modello al caso reale non possa provocare problemi. Tutti i SO real-time consentono di usare algoritmi a priorità fissa, offrono delle API che consentono di impostare le priorità di task sulla base di deadline relativa. Pochi permettono di usare schedulazione dinamica a livello di task, come EDF: il reinserimento automatico dei task in coda a priorità nella posizione corretta in base alla deadline assoluta è inefficiente: servono strutture dati appropriate o gestione di pochi task. È pericoloso gestire strutture dati complesse, quindi per avere efficienza bisogna ridurre numero di task supportati.

5.5 Standard per RTOS

Standard importanti:

- Portabilità dell'applicazione
- Interoperabilità dei sistemi
- Possibile cambiare real-time OS

Diversi standard:

- POSIX, estensione real-time
- OSEK/CDX
- ARINC/APEX
- ITRON

5.5.1 POSIX

Estensione che definisce API per avere ad esempio:

- mutua esclusione con priority inheritance.
- attesa e sincronizzazione tramite variabili condizione
- memoria condivisa
- code di messaggi a priorità
- schedulazione preemptiva con priorità fissa
- server sporadici
- gestione del tempo con alta risoluzione
- possibile misurare e limitare il tempo di esecuzione dei task.

4 profili differenti dello standard:

- Minimal Real-Time System, per piccoli sistemi embedded: solo thread, non processi e I/O tramite device file nessun file system.
- Real-time COntroller, per sistemi robotici: come il precedente, ed in più un file system con cui gestire i file regolari
- Dedicated Real-time System: per sistemi avionici, grandi sistemi embedded, multi-processo con meccanismi di protezione degli accessi alle risorse
- Multi-purpose Real-time System: per sistemi general purpose con applicazioni sia real-time che non, tutti i servizi POSIX per i sistemi operativi general purpose

5.5.2 OSEK/VDX

Progetto congiunto di diverse industrie automobilistiche (BMW, Bosch, Opel) e l'università di Karlsruhe. Definisce insieme di API per un sistema real-time integrato in un sistema di gestione di rete. Orientato a sistemi di controllo con vincoli real-time hard, alata criticità, grandi volumi di produzione. Tiene in grande considerazione l'ottimizzazione del codice, riduzione dell'occupazione di memoria e miglioramento delle prestazioni del RTOS.

Propone interfacce e protocolli ad alto livello per le comunicazioni interne del veicolo ed interfacce e protocolli per l'interoperabilità dei vari sistemi embedded all'interno dell'autoveicolo:

- politiche di accesso
- meccanismi per la resistenza ai guasti
- procedure di diagnostica della rete e di comunicazione

Un tipico sistema OSEK è scalabile, può usare diversi tipi di processori, non prevede protezione di memoria. Deve produrre software portabile: tra l'applicazione ed il SO c'è uno strato di interfaccia scritto in ISO/ANSI-C. In ogni caso non vengono specificate interfacce verso i sistemi I/O.

Il progettista può usare degli strumenti di configurazione standard per definire i servizi e l'uso, viene proposto linguaggio OIL (Osek Impelmentation Lan-guage). L'allocazione è statica, ovvero tutte le strutture dati del kernel e dell'applicazione sono allocate staticamente.

Il supporto per architetture "time triggered" fornisce la specifica di un sistema operativo basato sul tempo.

Supporta architetture "time triggered", fornisce una specifica di un sistema operativo basato sul tempo che può essere completamente integrato nel framework OSEK/VDX. Ci sono le specifiche di un sistema real-time in questo stanard per sistema embedded

5.5.3 ARINC 653

Standard per sistemi avionici, progettazione e certificazione di sistemi safty-critical e real-time. Lo standard consente di definire sistemi IMA (Integrated Modular Avionics), ovvero diversi componenti software con diversi requisiti di robustezza coesistono ed utilizzano le stesse risorse hardware. Le applicazioni real-time interagiscono con i servizi offerti dalla piattaforma ARNIC usando un insieme di API chiamato APEX. Memoria fisica è suddivisa in partizioni, ciascun modulo vive nella propria partizione. Anche il processore viene partizionato con cyclic-executive che divide il tempo di processore in maniera fissa tra le varie partizioni: per ciascun sotto sistema non è possibile consumare più tempo di quanto gli viene allocato. Posso mettere qualsiasi cosa in una partizione. Possibile comunicazione tra partizioni differenti mediante scambio di messaggi:

- sampling message: hanno sempre la stessa struttura, ed ogni occorrenza di un certo messaggio sovrascrive la precedente occorrenza, le API non bloccano mai.
- queuing messages: possono avere struttura, le API possono bloccare se la coda di messaggi è piena o vuota. Inoltre, il tempo di blocco può essere definito in fase di invio o ricezione del messaggio.

Standard commerciale, non molte informazioni pubbliche su come realizzare questi standard.

5.5.4 ITRON

Progetto giapponese del 1984, progetto accademico da cui sono derivati vari standard de facto giapponese ed asiatico. Sono stati derivati diversi standart:

- ITRON, per sistemi embedded
- μ TRON: per sistemi embedded a 8/16 bit

- JTRON: variante per Java
- BTRON: per computer con interfaccia utente per desktop e tablet
- CTORN: per mainframe ed apparati di rete.

Enorme diffusione in Giappone, la caratteristica è la loose standardization:

- API specificate solo a livello sorgente, non esistono requisiti di compatibilità del codice
- Parametri delle API passati separatamente o come unico pacchetto
- Le ultime versioni dello standard profile prevede supporto a priorità dei task, code di messaggi, primitive di sincronizzazione ma non ha meccanismi per protezione di memoria.

5.6 Caratteristiche comuni dei RTOS

- Corrispondenza agli standard: generalmente le API sono proprietarie, ma con i RTOS offrono anche compatibilità allo standard RT-POSIX
- Modularità e scalabilità: il kernel ha dimensione ridotta e le sue funzionalità sono configurabili
- Dimensione del codice: spesso basati su micro-kernel
- Velocità ed efficienza: basso overhead per cambi di contesto, latenza delle interruzioni e primitive di sync.
- Porzioni di codice non interrompibile: generalmente corte e di durata predicibile
- Gestione delle interruzione separata: interrupt handler corto e predicibile, ISR lungo e di varia durata
- Gestione della memoria: possibilità di utilizzare la memoria virtuale e protezione degli spazio di indirizzi del kernel, quasi mai si usa la paginazione.

Nei principali SO real time:

- Almeno 32 livelli di priorità
- Possibilità di scelta FIFO e round robin per la gestione dei task nello stesso livello di priorità
- Possibilità di cambiare la priorità a run-time
- Generalmente non supportano politiche di scheduling a priorità dinamica o server a conservazione di banda
- Offrono meccanismi per controllo dell'inversione di priorità: tipicamente priority inheritance , alcun anche priority ceiling.

- Risoluzione nominale di un timer e orologi di un nanosecondo, ma l'accuratezza non è mai sotto i 100 ns a causa della latenza nella gestione degli interrupt.

Alcuni di questi meccanismi posso essere abilitati/disabilitati in fase di progetto. Diversi meccanismi per la comunicazione nel sistema: memoria condivisa, mutex, segnali, semafori, code di messaggi...

Standard POSIX descrive come usare questi meccanismi in ambito real-time. Per la memoria condivisa, non tutti i SO real-time supportano memoria virtuale, se supportata può anche essere disabilitata.

Le applicazioni embedded e real-time sono a memoria contenuta e processano dati di dimensioni fissa, quindi non supportano la memoria virtuale non è un problema, inoltre si risparmia overhead dovuto alla traduzione degli indirizzi. Protezione della memoria: molti SO real-time non prevedono protezione della memoria, alcuni prevedono l'esecuzione dei task in modalità kernel e questo ha come conseguenza particolare efficienza nel passare da un job all'altro. Un unico spazio di indirizzamento implica una maggiore leggerezza e semplicità nel task switching, ovviamente lo sforzo degli applicativi è maggiore.

5.7 Esempi di sistemi operativi real-time

Esistono molti sistemi operativi real-time, in particolare per sistemi embedded. Numero enorme: molti sono open source, altri commerciali ed alcuni ibridi.

Oggi si cerca di usare sistemi operativi comuni, cercando di riusare lo stesso SO: prima ognuno aveva il proprio sistema operativo nel general purpose, poi avvento di Linux; stesso è avvenuto nei SO real-time.

5.7.1 VxWorks

Basato su micro-kernel Wind, usabile anche su sistemi certificabili di alto livello. Ambiente di sviluppo basato su Eclipse, può essere adattato a sistemi embedded sia real-time, le API sono proprietarie, fornisce interfaccia di compatibilità per interfacce POSIX real-time.

5.7.2 LynxOS

Linea di sistemi operativi commerciali, derivato ed ispirato a Linux, varie versioni per i diversi scopi.

Il cuore delle versioni è un kernel è LynxOS RTOS, ovvero il kernel compatibile con l'ABI(Application Binary Interface) di Linux, ovvero è possibile eseguire compilare ed eseguire native per Linux

5.7.3 QNX Neutrino

Sistema operativo per automobili, segue standard POSIX, architettura a micro-kernel. È progettato intorno ad un meccanismo di scambio di messaggi e segnali

5.7.4 eCos

RTOS open source, progettato per sistemi embedded, costruito con licenza GPL compatibile. Fortemente modulare e configurabile. Offre API proprietarie compatibili sia con POSIX che con μ TRON. Kernel è solo uno dei package del sistema, non è indispensabile per lo sviluppo delle applicazioni in eCos. Scritto in C++ che utilizza compilatore GNU, ma tutti i dettagli legati all'architettura hardware sono in C ed Assembly.

5.7.5 FreeRTOS

Licenza GPL modificata, supporta oltre 30 architetture HW ed è molto piccolo: immagine anche di 9KB, gli bastano 256B di RAM, possibile metterlo nel microcontroller di una pennetta USB. Scritto per lo più in C, non include driver delle periferiche, più simile ad una libreria per thread che ad un SO, però è molto usato nelle sue varianti.

5.8 Embedded

Una serie di sistemi embedded della Microsoft, hanno cambiato diversi nomi, in parallelo allo sviluppo di Windows mobile, sistemi orientati ai cellulari ma molto simili a sistemi desktop. Non sono versioni leggere di Windows, sistemi operativi basati su kernel monolitici, non compatibili con lo standard POSIX, codice del kernel ha licenza shared-source. Non adatti ad architettura multi-processori e difficilmente possono essere definiti sistemi real-time: limiti architetturali molto forti, esempio non è possibile creare un task a livello di priorità alto. Lo creo a priorità base, va in esecuzione e cambia la sua priorità, ma non so quando questo avviene.

5.9 Zephyr

Progetto iniziato da Wind River per IoT. Nel 2016, abbracciato dalla Linux Foundation, sponsorizzato tra gli altri da Facebook, Google e Intel.

È un sistema completo che comprende kernel, librerie e driver. Ha un singolo spazio di indirizzamento ma con meccanismi di protezione della memoria. Ci sono scheduler a priorità fissa, ma c'è round robin opzionale. Supporta multiprocessore asimmetrico (i vari processori fanno tutte operazioni diverse) e simmetrico. È un sistema altamente modulare e configurabile. Implementa inoltre un sottoinsieme delle API POSIX, chiamato OSAL, su cui vari sistemi operativi real-time si stanno orientando.

Punto di forza: integra molti protocolli di comunicazione tra cui IPv4, IPv6, OMA, LWM2M, Bluetooth Low Energy; è uno dei SO real-time su cui convergeranno la maggior parte delle risorse dell'ICT.

6 Nascita ed evoluzione dei sistemi operativi

Information Technology moderna ruota attorno a Linux, che è uno dei principali motori economici che traina l'information technology oggi. Va di pari passo con l'evoluzione dei calcolatori elettronici.

Calcolatore elettronico: macchina prevalentemente costituita da dispositivi micro-elettronici, elabora informazioni in ingresso e produce output, usa un programma deciso dall'utente finale, immagazzinato in memoria insieme ai dati. In un calcolatore è l'utente (che lo deve programmare) a definire il comportamento della macchina, inteso come modo di elaborare le informazioni in ingresso, non il progettista.

Di conseguenza, il CE è una macchina universale, in grado di eseguire qualsiasi programma pensabile.

Sistema operativo: collezione di programmi di base per la gestione delle periferiche hardware del calcolatore per la creazione di un ambiente per l'esecuzione controllata dei programmi applicativi da parte degli utenti finali.

Crea un astrazione dei dispositivi hardware, assegna le risorse del sistema ai programmi in esecuzione, realizza un interfaccia di comunicazione tra l'utente finale ed il calcolatore.

6.1 Evoluzione dei CE e dei SO

SO è programmato in stretta correlazione con i CE, l'evoluzione avviene quindi di pari passo. Precursori:

- 1834-1871: Analytical Engine di Babbage
- 1938-1945: Versuchsmodell-1,-2-3 e -4
- ...

Primi veri calcolatori dopo la II WW

6.1.1 Versuchsmodell-1,-2-3 e -4

Progettati da k. Zuse a Berlino tra il 1936-44, motivazione per esigenze militari: una serie di calcoli ed applicazioni di leggi per progetto di caccia-bombardieri. Calcoli onerosi: Zuse costruisce calcolatori elettro-mecanici, con programma su nastro di celluloido (perforava schede, ogni foro era 0 o 1). La logica era binaria, erano capaci di operazioni in virgola mobile.

6.1.2 Automatic Sequence Controlled Calculator

Progetto Americano, industria che deve progettare sempre armi: in ogni costruzione del pezzo introduce degli errori (tolleranti), questo può provocare variazioni nel punto di caduta del proiettile anche di diverse centinaia di metri. Problema di balistica complesso, per ogni pezzo di artiglieria stampa di un libro che per ogni possibile bersaglio e condizioni risolve il problema di orientare il cannone per

sparare. Si arriva al punto per cui l'industria può produrre pezzi di artiglieria, ma non la scrittura del libro (migliaia di problemi da risolvere): arruolamento di matematiche (ragazze) per risolvere problemi di balistica come dei computer: da qui nasce il termine, lavoro giorno e notte.

Si pensa di poter risolvere i calcoli in maniera veloce e senza errori. IBM, progetto di H. H. Aiken tra il 37'-42', calcolatore elettromeccanico, con programma su nastro perforato. Donato all'università di Harvard dopo la guerra.

6.1.3 Atanasoff-Berry

Calcolatore completamente elettronico e con aritmetica binaria, ma non è una macchina universale: mancano una serie di feature, come l'esecuzione di salti condizionati

6.1.4 Colossus

Costruito da T. Flowers nel 44' con valvole elettroniche. Impiegato da Bletchley Park (UK) per decifrare i messaggi delle telescriventi e macchine cifranti tedesche (Enigma).

6.1.5 Electronic Numerical Integrator and Computer

ENIAC: ideato da J.W. Mauchly nel 41', pensa di realizzare un calcolatore con 18k valvole elettroniche: le valvole sono come lampadine elettriche, si bruciano periodicamente. Quindi, la macchina va manutenuta di continuo, ma era costruita bene per farla funzionare continuativamente per un certo periodo di tempo. Programma costituito da cavi passati tra le unità funzionali del calcolatore: ammassi di cavi, flusso dell'informazione dei cavi descrive programma eseguito sui dati.

Si dice che primo programma eseguito dalla macchina fossero una serie di calcoli per confermare progetto della atomica.

6.2 Primi CE

Solo precursori, manca ancora l'idea che il programma è incluso nella memoria con i dati.

Prima idea: Analytical Engine di C. Babbage (metà 800'), macchina estremamente impegnativa, programmabile.

Idea ripresa e reinventata nel XX sec da Turing, Zuse etc... Mai realizzata da Babbage, troppo avanti con i tempi come idea, nel 1948 entra in funzione il primo calcolatore elettronico con programma in memoria centrale: Manchester Small-Scale Experimental Machine. Nel 49' USA completa primo calcolatore vero e proprio, EDVAC mentre nel 51' primo calcolatore commerciale.

6.2.1 Manchester Small-Scale Experimental Machine

Progetto universitario, primo calcolatore con programma in memoria centrale ad entrare in funzione. Memoria da 32 registri con 32 bit, un registro accumulatore ed uno per l'istruzione corrente. Progetto basato su studi americani

6.2.2 EDVAC

Progetto degli stessi di ENIAC, riprendono idea di ricostruire macchina in cui dispositivo di memoria immagazzina dati e programmi. Architettura descritta da von Neumann del 45' ma non cita i nomi dei due progettisti. Completato nel 49', quando già diversi calcolatori erano in funzione.

6.2.3 Ferranti Mark 1/UNIVAC I

Evoluzione del SSEM di Manchester, lo rende commerciale. Da parte americana il primo progetto commerciale è UNIVAC I, funziona con clock a 2.25 Mhz, 1905 operazioni/minuto. Costo = 1.5M dollari.

Non si pensava ci sarebbe stato mercato per più di 2-3 calcolatori, IBM non si lancia nel mercato. Dal 52', venduti 46 esemplari per fare calcolo statistico.

6.3 Uso dei calcolatori di I generazione

Macchine molto grandi e costose, l'operatore era spesso anche il programmatore ed operava con console di comandi:

- Caricare il programma in memoria usando interruttori sulla console
- Caricare il programma in memoria decodificando un nastro di carte
- Impostare indirizzo iniziale del programma
- Avvia esecuzione del programma
- Controllare lo stato di esecuzione con spie luminose.
- Fermare programma
- Stampare i registri, possibilità di interfacciare CE con le stampanti.
- Stampare output del programma su nastri di carta, schede perforate etc...

Nuove periferiche inventate:

- Nastro magnetico
- Terminali: macchine per scrivere adattate per la console

Per ciascun dispositivo di I/O si doveva scrivere una procedura apposita per usarlo: driver di dispositivo. Vennero sviluppate librerie contenenti i driver di dispositivi e altre funzioni comuni che venivano copiate, secondo necessità, nel programma da eseguire. Col tempo, sviluppati anche nuovi programmi:

- Assemblatori, per facilitare la programmazione utilizzando codici simbolici ed etichette
- Compilatori per FORTRAN, LISP, ALGOL, COBOL. Opera su un certo dato ed opera per produrre risultato che sarà input dell'assemblatore.

6.4 CE di II generazione

Sdoppiamento dell'operatore e del programmatore: per diminuire i tempi morti del calcolatore, programmatore scrive i programmi su un pacco di schede perforate (job) e le passa all'operatore. L'operatore eseguiva tutti i job di tutti i programmatore e li mette insieme, separandoli con schede particolari per riconoscere quando iniziava l'altro job. Programmatore non ha il controllo della macchina: può fare debug solo sulle informazioni date dall'operatore. Conveniente: operatore metteva insieme tutte le schede in FORTRAN in un lotto, quelli COBOL in un altro lotto. Caricava il nastro nel compilatore e poi lo passava all'assemblatore, poi eseguiva.

Nascita dei primi SO: si ci rende conto che non ha senso caricare in memoria sempre gli stessi nastri, tanto vale tenerlo in memoria centrale. È anche possibile mantenere monitor che mi permetta di passare da un programma all'altro quando il precedente ha finito. I/O sovrapposto o spooling, tutto effettuato con JCL (Job COnrol Language).

6.4.1 Monitor residente

I SO nascono tentando di automatizzare il lavoro dell'operatore con sequenzializzatore automatico dei lavori di elaborazione: piccolo programma chiamato monitor residente, sempre in memoria centrale. Monitor gestiva il trasferimento automatico dell'elaborazione da un job. Interpreta schede di controllo per capire la sequenza dei job, scritte in JCL.

6.4.2 I/O sovrapposto

Il monitor residente consente di ridurre i tempi morti di elaborazione, ma collo di bottiglia diventano i dispositivi I/O. Nastri magnetici, per velocizzare l'input e l'output: le schede perforate venivano riversate su nastro, quando questo era pieno veniva installato sul calcolatore. L'output del calcolatore veniva stampato o riversato su schede.

Nastro viene poi sostituito da dischi rigidi (non come quelli odierni).

Spooling è estensione della tecnica: nasce con la nascita dei nastri magnetici, usati come memoria di transito per i dati in input ed un disco magnetico come memoria di transito per i dati in output. I dischi si interfacciano con le unità che stampano o leggono schede perforate.

Consente di collegare direttamente al calcolatore i dispositivi più lenti bufferizzando con i dischi magnetici.

6.5 III generazione

CE basati su transistor, i circuiti diventano sempre più veloci, collo di bottiglia sono sempre più le periferiche di I/O. Accanto ai mainframe si diffondono minicalcolatori, come quelli della serie DEC PDP, anni 70'. IBM era famosa nel costruire mainframe, snobba mercato dei minicalcolatori. Tra il 65'-80' lo sviluppo dei minicalcolatori porta allo sviluppo di nuovi SO:

- SO multi-tasking: passo successivo per supportare al meglio i dispositivi di I/O.
- Partizionamento del tempo macchina, time sharing
- Servizi di calcolo ad una comunità di utenti.
- Interazione tra utente e calcolatore: terminale, eventualmente distante e collegato alla linea telefonica.

Esempi di SO di 3° generazione: XDS-940, Multics, OS/360 (SO IBM costruito per i mainframe, enorme in termini di linee di codice).

6.5.1 Multics

Progetto ambizioso del MIT, General Electric e Bell Lab.

Progetto di realizzare un CE time sharing che doveva fornire servizio di calcolo ad una città come Boston, analogamente all'elettricità o all'acqua. Estremamente innovativa per l'epoca, tutti gli utenti connessi da questo servizio (predecessore di Internet). Calcolatore che lavorava con memoria virtuale, con indirizzi costituiti da numero di segmento ed offset. Segmenti suddivisi in pagine di memoria da 1K parole a 36 bit. Ogni segmento in memoria centrale era anche un file in memoria secondaria e vi si poteva accedere tramite il nome file. (ls di Linux: deriva da list segments). Il file system era una struttura ad alberi con più livelli, completamente estendibile dagli utenti. Sistema multi-processore, protezione della memoria garantita da lista di controllo degli accessi associata ad ogni file ed insieme di livelli di protezione per processi. SO scritto in linguaggi di alto livello.

6.6 Unix

Nel 69' la Bell comincia a pensare che Multics sarebbe fallito, troppo avanti per il tempo (finito qualche anno fa).

Bell voleva risparmiare sulla realizzazione del SO, ma alcuni programmati si ritrovano disoccupati: D. Ritchie, K. Thomson, McIlroy... Esperienza di Multics aveva fatto sì che ci fossero idee su come implementare un nuovo File System ed un nuovo SO. Inoltre Ritchie e Thomson avevano creato un gioco e avevano bisogno di un calcolatore su cui eseguirlo. Cominciano a produrre sw per un "piccolo" calcolatore, PDP-7.

Software in assembler per PDP-7, nucleo di Unix: ci sono alcune idee di Multics:

- File system con struttura ad albero
- La Shell di comando diventa un processo utente
- Il sistema di paginazione della memoria virtuale

Nome: Kernighan, 1970 conia il termine UNICS come presa in giro di Multics, che presto diviene Unix.

Convincono la Bell a investire su DEC-PDP-11, promettendo di sviluppare un sistema tipografico per il reparto brevetti della Bell Labs. Il primo utilizzo ufficiale di Unix comincia nel 1971, la prima applicazione ufficiale è il formattatore di test nroff.

Nel 1972 Unix è su una decina di macchine, ma il sistema è scritto in assembler ed un linguaggio interpretato detto B. Nel 71 Ritchie comincia a scrivere il C, usando strutture dati e tipi di dato di B, trasformandolo in linguaggio compilato. Unix viene riscritto quasi totalmente in C, linguaggio di alto livello: novità per il tempo, non c'era nessun SO scritto in linguaggio di alto livello. Unix adattato e portato su altre architetture, bisognava solo riscrivere le parti in assembler.

Nel 1974 Ritchie e Thomson descrivono Unix in un articolo, all'epoca era installato su 600 macchine. In tutto il mondo laboratori cominciano a chiedere codice sorgente di Unix.

Nel 58', Bell fu coinvolta in indagine anti-trust per via del monopolio sulla rete telefonica con il governo americano e gli fu proibito di entrare nel mercato dei computer.

Ogni tecnologia prodotta e non legata al mondo telefonico doveva essere fornita in licenza gratuita a chiunque. Thomson quindi non può negare src code di Unix, che si diffonde in tutti i laboratori universitari del mondo.

6.6.1 BSD Unix

Centro che contribuisce di più è il Berkeley's computer Science Research Group dell'università della California.

Prime modifiche significative da Bill Joy, cofondatore di Sun Microsystems, grazie al quale fu integrato nel 1983 in Unix lo stack dei protocolli TCP/IP di ARPANET.

Passo epocale: storia di Internet diventa la storia di Unix, diventano legati a doppio filo.

Berkeley costruì la propria costruzione Unix, chiamata BSD, con varie innovazioni nel kernel e tante applicazioni.

6.7 IV generazione

Nasce con i microcalcolatori, i primi erano dei kit che gli appassionati di elettronica compravano e assemblavano da soli. Inizialmente basati su micro-processori a 8 bit:

- Intel 8008

- Intel 8080
- MOS Technology 6502
- Zilog Z80

Poi si affermarono processori a 16 bit.

Metà anni 80', primi calcolatori pre-costruiti. Primo personal computer oggi ricordato è l'IBM PC model 5150.

6.7.1 PC IBM

Produce microcalcolatori molto costosi, fa centro con il 5150. Per commercializzarlo, serve il SO: era basato su Intel 8088, IBM si riferisce ad azienda che aveva costruito SO per cpu ad 8 bit e le chiede di avere il SO CP/M. SO nato per i microcalcolatori, macchine pensate per gli appassionati, CP/M non era SO come Unix, era molto più povero. IBM tenta di usarlo e richiede di adottarlo alla Digital.

IBM trattata male dalla Digital, quindi IBM non ha SO. C'è piccola società, fondata in un garage da Bill Gates e quell'altro. L'altro sa scrivere buoni programmatore, Microsoft ha venduto buoni prodotti all'IBM. Microsoft non ha SO, e trova QDOS, che è scritto da un'azienda di Seattle. QDOS è SO giocattolo, non ha nulla a che fare con Unix. Gates acquista per 50K i diritti, ma non dice all'azienda cosa ne farà, ovvero che lo distribuirà sui PC IBM. Molti anni dopo SCP porta in tribunale Microsoft e vince ottenendo risarcimento milionario. Microsoft cambia nome al SO e lo chiama DOS, e lo associa al BASIC (che era proprietario) e lo da all'IBM col nome di PC-DOS. Ci mette il BASIC per avere linguaggio di alto livello con cui programmare. Appaiono documenti dell'IBM in cui ci sono specifiche per costruire PC clone compatibile con quello IBM, mercato invaso da questi PC. Microsoft non può dare ai cloni lo stesso SO, quindi gli da versione MS-DOS, licenza è differente ma di fatto è quello. Questo fa la reale fortuna del SO Microsoft: cloni si diffondono in misura eccezionale. Reale innovazione non è nel SO, ma nelle interfacce grafiche GUI, inventate nei 60' da Engelbart.

Steve Jobs, visita laboratori Xerox e vede GUI, quindi cerca di costruire calcolatori che usino GUI, primo fu Lisa, poi Machintosh che ebbe enorme successo: primo calcolatore con GUI ad avere successo enorme.

Anche sistemi Unix sviluppano la loro prima GUI chiamata X Windows System. La GUI viene ripresa dalla Microsoft, come applicazione sopra l'MS-DOS. Nel 95 diventa integrata nel sistema e si parla di Microsoft Windows.

6.7.2 Le Unix war degli anni 80'

Negli anni 80' si moltiplicavano le diverse versioni di Unix, alcune basate su BSD, altre da aziende commerciali come SCO. Ci sono centinaia di versioni di Unix, inoltre nell'83 AT&T perde altra cause e Bell viene suddivisa in varie compagnie locali. Quindi AT&T decide di fare profitto sui PC, cerca di fare

profitto da Unix System V. AT&T vuole di nuovo i diritti di Unix. Nell'85, per facilitare interoperabilità dei diversi sistemi nasce lo standard POSIX, sponsorizzato da IEEE. Bisogna adattare tutti i sistemi verso questo standard, fase di caos, si litiga per licenze Unix nei tribunali, quindi tutto questo frena espansione di Unix.

Microsoft guadagna monopolio dei per i SO.

7 Il software libero

Fine anni 90': caos nel mondo dell'IT (Information Technology), calcolatori entrano nelle case di tutti. Ogni PC di oggi è diretto discendente dell'IBM, c'è monopolio di un unico SO, che è Windows: c'erano SO superiori a Windows ma nell'AT&T c'era lotta per riprenderne il monopolio dalle altre case distributrici.

7.1 Movimento degli hacker

Origine degli hacker è nel MIT, termine ha cambiato significato varie volte: inizio degli anni 50' gli hacker erano gruppi di studenti. Piena guerra fredda, termine hacker aveva significato goliardico: comunità segrete di studenti non legate allo studio.

Verso la fine degli anni 50', movimento di ribellione assume una forma più definita, si passa a movimento per reagire ad un ambiente autoritario e competitivo:

- Tunnel hacking: esplorazione delle innumerevoli gallerie sotterranee del MIT.
- Phone hacking: scherzi basati sull'abuso del sistema telefonico

Nel gruppo era bandita ogni attività che potesse essere malevola, dolorosa o dannosa.

Anni 50'-60' continuaron queste attività di hacking, che erano divertenti ed esplorative, studenti del MIT appassionati di modellismo ferroviario, c'era sottogruppo di studenti che formava comitato Signals & Power, che si occupava del circuito elettrico della ferrovia in miniatura. Circuito era collegato al sistema telefonico, quindi era possibile comandare i modellini tramite i telefoni.

Per poter riutilizzare i componenti elettrici cercava di realizzare i circuiti con il minimo di componenti ed il massimo dell'efficienza.

Verso la fine degli anni 50', i membri del comitato Signals & Power cominciano ad interessarsi di informatica.

Permesso agli studenti di usare prototipo di un calcolatore a transistor (3MLN\$), il TX0, grazie alla sponsorizzazione della Digital. MIT poi riceve in regalo il PDP-1, primo mini-computer della storia costruito dalla digital.

Studenti scrivono software per il PDP-1, sono ancora gruppi di hacker. Uno dei primi programmi: gioco, 1962 prima versione di Spacewar, tra cui Steve Russel. Anni 60'-70', movimento hacker continua: studenti si laureano, alcuni divengono professori, coinvolgono soprattutto il laboratorio di AI del MIT. Rispetto

agli hacker degli anni 50' è che interesse è focalizzato quasi tutto sui calcolatori elettronici. Inoltre, le attività non sono più segrete o ristrette, non c'è più connotazione di ribellione, sempre più studenti fanno parte della comunità.

Al contrario, vengono incoraggiati lo spirito di collaborazione tra gli studenti per risolvere problemi comuni, ideale del gruppo è avere conoscenza condivisa. C'è clima dello spirito di competizione nelle università americane: se vieni colto mentre copi, vieni espulso. Non solo le violazioni sono punite, ma gli studenti non sono incentivati a farlo: se prendo voto più alto sono sopra ad altri studenti, quindi guadagno di più, studenti non aiutano a copiare perché non è nel loro interesse.

Gruppi di hacker produssero molto software, tra cui lo stack TCP/IP di ARPANET, coniarono un proprio slang. Si formano inoltre una serie di ideali e principi diffusi nella comunità:

- L'accesso ai computer e qualunque cosa che può insegnare come funziona il mondo dovrebbe essere accessibile a chiunque
- Informazione libera
- Non fidarti delle autorità
- Hacker dovrebbero essere giudicati dalle loro capacità, no razzismo.
- Puoi creare arte e bellezza su un computer (oggi possibile creare un Rembrandt difficile da distinguere da un originale)
- I computer possono cambiare la tua vita

Hacker del MIT hanno dato il maggiore contributo al IT.

7.2 Software libero

All'acquisto di un calcolatore, assumo la proprietà della macchina: posseggo l'hardware, nel caso del software le cose sono differenti. Se acquisto il software, non assumo la proprietà del codice: il software è trattato generalmente come le altre opere dell'ingegno (opere letterarie, musicali etc...). Alla base c'è diritto di autore o copyright, in modo da garantire il giusto compenso all'autore o detentore dei diritti dell'opera.

In genere il software non viene acquistato, nel senso che non vengono trasferiti i diritti sull'opera, ossia il copyright. SI paga la licenza d'uso, ovvero la possibilità di usare il software. Brevetto: in molti paesi esiste anche un'altra possibilità per proteggere il software, ovvero il brevetto: meccanismo legale che impedisce l'utilizzo di una invenzione tecnologica non autorizzato da parte dell'inventore. In molti paesi, gli uffici brevetti sono inondati da centinaia di migliaia di richiesti di brevetti già visti o non innovativi, già solo controllare che i requisiti siano rispettati è un lavoro difficile, quindi il rilascio del brevetto avviene troppo facilmente. Amazon ha potuto brevettare la tecnologia one-click. Ogni azienda dell'IT possiede un proprio portafoglio di brevetti che usa come arma contro altre

aziende di IT (sia offensiva che difensiva). Brevetti non sono più usati come un modo per incentivare la ricerca, usate come armi per cercare di mettere fuori dal mercato un'altra azienda, aberrazione dell'uso del brevetto.

Si comincia a capire l'andazzo del mondo dell'IT all'inizio degli anni 80': l'idea di software che avevano gli hacker del MIT viene messa in crisi: una volta scritto software non c'era necessità di tenere per se il software. Cause nei tribunali per Unix, inoltre nuova legge del copyright del 76' negli USA: si comincia a capire che l'economia si basa sul valore del software e non dell'hardware.

Licenze strette sull'utilizzo del software, altra cosa che mette in crisi è che molti ricercatori del MIT lasciano il movimento per andare a lavorare in aziende di software.

7.2.1 GNU

Situazione colpisce in particolare un ricercatore del MIT, Richard Stallman (GCC, emacs etc...) fermamente convinto della libertà di uso del software. Ideatore del progetto GNU: GNU's Not Unix, Unix non era visto bene e Unix era fatto coincidere con Unix System V che AT&T voleva far usare a tutti per guadagnare. Idea è scrivere un sistema operativo libero da diritti d'autore e licenze. Rende SO compatibile con Unix, di modo da poter usare software scritto per altro sistema su GNU. Sviluppo del progetto è rapido: completato in meno di 10 anni, costruito SO partendo da 0 e senza violare copyright, si è avuto cura che si possa dimostrare in tribunale che sia così. Manca solo il nucleo del SO: kernel GNU ad oggi non ancora completato. Se usato kernel Unix BSD, poteva accadere che qualche azienda in futuro rivendicasse diritti su GNU.

La licenza BSD permette di includere il codice in prodotti commerciali "closed-source". Nuovo kernel da 0, Hurd che è prodotto open-source con architettura a micro-kernel. Progetto GNU riscuote comunque grande successo.

1985: Stallman fonda la free software foundation, che promuove la scrittura e diffusione di software libero. Alla base c'è licenza GNU GPL, la più diffusa licenza usata per software libero. Motivazioni etiche: utente deve avere accesso al source code, idea è libera. Software libero sotto licenza GPL deve garantire:

- Esecuzione per qualunque scopo
- Libertà di leggere e studiare il source code
- Ridistribuire programma originale
- Fare modifiche e ridistribuire il software così modificato

Chi modifica programma sotto licenza GPL, deve rilasciarlo sotto licenza GPL. GPL non esclude che programmatore possa essere pagato per il suo lavoro, molte aziende dell'IT fanno molti soldi usando software GPL. Inoltre, se il codice viene usato in un programma diverso, anche questo ricade sotto licenza GPL.

Tipologie di software:

- Pubblico dominio: nessuna licenza, chiunque può appropriarsene

- Copyleft: codice ha una licenza,a ma il software può essere distribuito secondo licenza GNU-GPL
- Non copyleft: permette condivisione e modifica di software ma con alcune restrizioni
- Software semi-libero: non può essere usato a scopi di lucro.
- Software proprietario: si da solo codice eseguibile, restrizioni per impedire di risalire al src code
- Freeware: software libero in formato eseguibile a titolo gratuito
- Shareware: codice eseguibile gratuito ma dopo un po' viene richiesto pagamenti

7.2.2 Open Source Initiative

Idee di Stallman sono molto radicali, altri gruppi pensano a varianti di distribuzione del software libero, 1998: OSI, fondazione che vuole presentare una variante alla FSF che non sia così radicale (nascono anche liti fra le fondazioni). Uno degli obiettivi è quello di fare avere all'utente finale il codice sorgente ma ci sono alcune possibilità come avere licenze free ma non open-source e vice versa. Nella stragrande maggioranza dei casi i progetti che sono software libero sono open-source e viceversa.

Idea è cercare di convincere le aziende che avere il src code del progetto aiuta a creare un prodotto migliore: tutti possono dare il loro contributo, si evitano situazioni di vendor lock-in e può essere commercialmente vantaggioso.

7.3 La nascita del kernel Linux

Ho GNU, ma non ho un kernel per portarlo su un calcolatore. Agosto 1991, appare mail su newnet (tipo mailing list): Linus Torvalds dice che sta sviluppando un SO libero for fun (mazza pensa se lo faceva sul serio) e vuole feedback su minix, in quanto il SO ricorda minix. Torvalds ha il kernel che manca sui PC IBM, MINIX è SO simile a Unix, creato dal professore Andrew Tanenbaum del 1987 come ausilio per un testo di SO, sviluppato per l'IBM PC e per l'IBM AT. Sorgente disponibile per scuole ed università, all'acquisto del libro del prof. c'era la licenza per il sistema. In effetti il controllo sul SO era strettamente di Tanenbaum, non accettava consigli per migliorarlo: voleva SO semplice in modo che studenti potessero capirlo.

Topo affermazione di Linux, Tanenbaum cambia la licenza di MINIX ed oggi è estremamente permissiva. Progetto era per cloni AT, per processori i386(486), SO usato era Microsoft DOS.

Linus Trovalds: studente di informatica, con esperienza di programmazione relativamente ristretta.

Progetto inizialmente focalizzato sull'architettura 808386, quindi PC IBM. Alternativa a Microsoft esisteva, era GNU a cui mancava però il kernel. Quando

Torvalds dice che sta scrivendo il kernel, questo genera grande impressione non solo per chi era interessato al mondo del PC IBM, ma anche per chi usava architetture alternative: si stava ancora aspettando scrittura del kernel GNU, che non arrivava mai.

Linus risponde che non può essere portabile, perché progetto è fortemente legato al 386 (lo stava usando anche per imparare l'architettura del 386), usava segmentazione propria del 386, ed anche paginazione.

Per favorire la portabilità del kernel si è messo mano al progetto ed è stata rimossa la segmentazione, usando solo la paginazione.

Man mano comincia ad accettare patch che arrivano da tutto il mondo, pubblica i sorgenti sui vari newsgroup: Linus continua a sviluppare ma accetta gli sviluppi degli altri utenti. Processo è volontario, idea degli hacker del MIT: offre qualcosa che ho realizzato agli altri; prima versione nel 91', prima stabile nel 94'.

Kernel assume quindi sembianze di kernel minix, legame però diventa sempre più lasco nel tempo e si cerca sempre di più di costruire sistema POSIX compilant.

7.3.1 Linux oggi

Nel 91': 76 file, 512KB, circa 8500 loc. Oggi: 70500 file, 930 MB su disco, 27 milioni di loc in C e Assembly, 42 milioni totali.

Successo estremo di Linux, trova ampio impiego nel mondo dell'IT. Quali sono stati i meriti del successo di Linux e come:

- versioni stabili: 1.0, 1.2, 2.0, 2.2... numeri pari
- versioni di sviluppo: 1.1, 1.3, 1.99, 2.1..., sviluppate in contemporanea alle versioni stabili.

Sviluppo procede in parallelo. Dal 2005, ci si rende conto che le il modello di sviluppo funziona male, c'è quindi non solo cambio di numerazione ma anche cambio di mentalità degli sviluppatori: versioni di sviluppo sono branch per fare test o proposte innovative.

Sviluppatori ora lavorano sempre sull'ultima versione, integrando cambiamenti etc... semplifica molto la gestione di come venivano effettivamente realizzate le distribuzioni di Linux, piccolo nucleo di sviluppatori dedicati alla risoluzione di bug nelle versioni stabili.

Prima del 2005 grossa variabilità nel numero di versioni di Linux prodotte ogni anno, dal 2005 in poi molte meno versioni maggiori (circa 11 versioni maggiori l'anno). Per capire evoluzione di Linux, si può anche vedere evoluzione del codice sorgente: la dimensione di per se non da indice di qualità, sicuramente però la gestione sarà impegnativa. Dimensione del source code è legata alla complessità del kernel ed alla difficoltà di gestione del progetto.

Molte metriche per valutare il kernel Linux, come numero di loc, numero di funzioni etc..., qualunque guardo mi da grafici equivalenti.

Regole strette già da subito: non esistono funzioni che sono blocchi di più pagine,

non esistono file sorgente troppo grande ... Regole imposte fin dall'inizio hanno portato ad avere le metriche equivalenti fra di loro.

Numero di loc cresce esponenzialmente di anno in anno, un programmatore esperto ha come media di scrittura 5 istruzioni/giorno.

7.3.2 Diffusione di Linux

Nei sistemi embedded:

- Linux 60%
- Windows 10%
- Altro 30%

Linux ha la maggioranza, in SE di fascia alta (in grado di supportare Linux).

Smartphones:

- Linux (Android) 49%
- Apple iOS 19%
- Blackberry 13% (oggi forse scomparsi)
- MS Windwos 11%
- Altro 8%

Oggi la statistica andrebbe aggiornata, ma di sicuro se la contendono Linux e iOS.

Mercato dei table vinto da iOS, ma Linux comunque combatte e si difende.

Tutto di rovescia nel mercato laptop e desktop:

- MS Windwos 92.2%
- Apple OS X 6.4%
- Linux 1.4%

Dovuto al predominio di Windows, ma oggi non c'è una difficoltà reale tra l'usare Windows o Linux.

Rimane problema del modello di vendita: all'acquisto del laptop/desktop c'è già installato Windows.

Server di rete:

- Linux 60%
- Windows 35%

Linux più usato per implementare server di rete .

Nell'ambito dei mainframe, per esmpio per applicazioni gestionali di grande portata:

- Linux 95%
- altri 5%

Nelle mani di IMB, che vende SO Linux.

Fascia dei super-calcolatori (per High Performance Computing): 100% dominato da Linux.

7.3.3 Chiavi del successo di Linux

Fenomeno di Linux durerà? È ormai diffuso in tutto il mondo, può accadere che nascerà un SO innovativo che soppianterà Linux, ma in 25 anni Linux è ancora al centro del mondo (25 anni so molti).

Ragioni del successo:

- Licenza di utilizzo GPL
- Flessibilità del kernel
- Responsabilità delegate dagli sviluppatori
- Personalità dei kernel hacker, programmatore più "anziani"
- Contributo dell'industria IT allo sviluppo di Linux

Tutte le ragioni sono fortemente correlate fra loro.

7.3.4 La licenza GPL

Successo di Linux è radicato nel progetto GNU di Stallman, per un SO free (nel senso della libertà di parola). Il progetto GNU ha portato allo sviluppo di molti strumenti essenziali: GCC, standard C, comandi di sistema...

Tutto il software ricade sotto licenza GPL, ma ruolo chiave è stato quello di far ricadere lo stesso kernel Linux sotto licenza GPL: chiunque può leggere e modificare i sorgenti del kernel a condizione che le modifiche siano resi disponibili con la stessa licenza GPL.

Ha impedito che qualche grande industria si impossessasse del src code di Linux e cominciasse a distribuire il prodotto col codice modificato senza fornire il src code.

7.3.5 Velocità di evoluzione del kernel

Codice viene modificato a ritmi impressionanti: una major release ogni 66 giorni. Sono paragonabili alle versioni major di un SO commerciale: in questo caso la frequenza è molto più bassa di quella del kernel Linux. Kernel che si trasforma velocemente implica che un supporto per device driver appare prima che l'hardware esca, stesso vale per il supporto di nuove CPU etc...

Quanti sviluppatori ci sono, per poter cambiare codice così velocemente: file CREDITS nel sorgente di Linux include circa 550 nomi, stima è che ci siano

migliaia di commit per ciascuna versione del kernel.

Ad oggi, si stima che ci siano più di 10000 membri nella community Linux, ma quindi come avviene la gestione di tutti questi sviluppatori: sviluppo dell'open source viola tutte le teorie di management conosciute. Il codice sorgente del kernel è modulare: ci sono componenti centrali, bus di sistema, stack di rete che sono logicamente separati ed hanno interfacce ben definite.

Kernel è monolitico, ma a livello di software il src code è organizzato in moduli ben definiti e ben separati fra loro.

Circa 1500 maintainer del kernel Linux, ciascuno segue uno specifico componente del sottosistema del kernel. Ogni responsabile coordina un certo numero di programmatori che lavorano sul componete, come regola generale le modifiche ad un componente o sottosistema debbono essere proposte a Torvalds dal responsabile interessato.

Alcuni responsabili generali hanno compiti più particolari. Tipicamente gli sviluppatori e Torvalds non analizzano a fondo le singole modifiche, bensì solo quelle alle aree più critiche. Il resto delle modifiche passa al vaglio dei responsabili e quindi dipende dalla qualità del lavoro dei programmatori.

7.3.6 Linux e l'industria

In origine, Linux è passato da uno studente ad una community di programmatori. Oggi piccole e grandi aziende supportano Linux, assumendo programmatori full time per gestire kernel Linux. Ogni anno nel kernel Linux sono integrate migliaia di modifiche per le aziende.

Vari motivi per contribuire in Linux:

- Aziende basano il loro core business in Linux
- Per garantire che i prodotti hardware funzionino a dovere con Linux. Scrittura driver serve.
- Adattamento e specializzazione del kernel in accordo ai propri requisiti.

Ciascuna azienda ha forte interesse nell'ottenere che i propri contributi siano integrati nel sorgente del kernel ufficiale, così che il codice evolva di pari passo con il resto del kernel.

Aziende devono stare al passo con le modifiche del kernel, perché altrimenti le patch diventano obsolete. Ci sono meccanismi per cui il codice prodotto dai programmatori dell'azienda deve essere revisionato prima di essere inserito nel kernel, quindi dopo varie revisioni il programmatore ha imparato a scrivere codice di qualità, c'è guadagno per tutti.

Si stima che l'80% dei contributi al kernel Linux derivano dall'industria.

7.3.7 Chi progetta Linux

Modifiche al kernel sono pilotate dall'industria, ma nessuno progetta l'evoluzione di Linux: le modifiche che vengono effettuate sono frutto della particolare necessità delle aziende, le modifiche sono dettate dal modo in cui evolve il mondo

dell'information technology.

Nessuno ormai sviluppa più sistema in proprio: per tenere al passo coi tempi un sistema bisogna pagare troppe teste. Chi decide cosa accadrà nel kernel Linux sono le aziende, non serve pianificare cosa accadrà. Questa è la ragione principale del successo di Linux, non è pilotato bensì è un qualcosa che si evolve.

Torvalds e gli altri sviluppatori sono arbitri: garantiscono che ogni modifica al kernel Linux sia tecnicamente corretta e, soprattutto, potenzialmente vantaggiosa per la comunità di Linux e non solo per la singola azienda che l'ha proposta.

Linus Torvalds ha impiego a tempo pieno presso la Linux Foundation: consorzio no profit che è finanziato da molte aziende dell'IT che ha due scopi:

- Promuovere la crescita di Linux
- Finanziare Torvalds ed altri manager in modo che siano indipendenti dalle altre aziende dell'IT.

Linux foundation sovvenzionata da molteplici aziende, tra cui anche Microsoft (che sta proprio tra le platinum, cioè quelle che cacciano 500M all'anno).

8 Linux in ambito real-time

È possibile usare SO Linux nell'ambito dei sistemi real-time: tutte le fasce di CE usano Linux, problema di fondo è che Linux non è SO real-time dalla nascita, ha una serie di scelte progettuali che ne complicano l'uso in ambito real-time.

È ottimizzato per fare in modo che hardware (in particolare i processori) siano usati per il maggior tempo possibile, e minimizzare i tempi medi di risposta.

Questi non sono gli obiettivi di un SO real-time: questo richiede la predicitività del sistema ed il rispetto delle scadenze. La proprietà deve essere inserita fin dal progetto del SO per poter garantire che le applicazioni real-time possano rispettare le scadenze. Sembra quindi che Linux non possa essere usato in ambito real-time, ma Linux ha delle caratteristiche interessanti per l'ambito real-time:

- Buona gestione dei timer software e del tempo
- Gestione separata delle interruzioni
- Schedulazione e gestione delle priorità dei task
- Supporto alle politiche di accesso alle risorse condivise
- Consente interruzione dei task anche in kernel mode, quindi è possibile sostituire il processo in esecuzione con uno a priorità maggiore
- Facile creare task in kernel mode, ovvero processi che non hanno controparte eseguita in user mode. Questo velocizza cambi di contesto e gestione della memoria quando avvengono solo tra task in kernel mode

- Supporto alla memoria virtuale e protezione della memoria per i task in user mode
- Possibilità di configurare il kernel a grana fine (in fase di compilazione è possibile decidere cosa includere/escludere)
- Kernel è monolitico, ma il codice è open e quindi modificabile, è modulare per ottenere occupazione di memoria (footprint) ridotta, posso sfruttare questa cosa nel sistema finale
 - È il SO con il maggior numero di architetture supportate
 - È il SO con il maggior numero di driver di periferiche disponibili
 - Costituito da una comunità di sviluppatori, supportato da comunità attiva

8.1 Limiti di Linux come hard RTOS

Linux già oggi non ha problemi in ambito soft real-time, grosso problema è per l'hard real-time: devo certificare che il sistema risponderà a certe caratteristiche, serve quindi predicitività temporale del kernel e delle applicazioni. Questo perché:

- Kernel non è completamente interrompibile
- Le interruzioni sono disabilitate nelle sezioni critiche (ad esempio quelle protette da mutex o semaforo) perché altrimenti potrebbero essere eseguiti flussi di codice che richiedono le stesse primitive di sincronizzazione della sezione critica, quindi si può essere esposti a deadlock.
- Le interruzioni possono essere risolte, se ad esempio riesco a scrivere sezioni che non durano più di un tot. Ma anche se posso avere un controllo su quali sono le sezioni critiche nel kernel, ma non posso dire lo stesso delle ISR (Interrupt Service Routine), che fanno parte dei driver delle periferiche. Questi sono scritti nella parte "bassa" della gerarchia dei developer, quindi qualità del codice è solitamente più bassa.
- Gestione delle interruzioni non usa meccanismi di priorità

È possibile rimediare a questi limiti, si sta cercando di porvi rimedio ed esistono alcune strade che permettono di modificare kernel Linux rendendolo predicable, così da poterlo usare in ambito hard real-time in modo da ottenere certificazione almeno dal punto di vista teorico. Kernel è monolitico, ha senso impiegarlo in ambito real.time? Sì, perché kernel è estremamente configurabile, in modo da ridurre drasticamente le sue dimensioni. Non è adatto per sistemi embedded di fascia più bassa (es firmware della chiavetta USB).

8.1.1 Gestione delle interruzioni in Linux

Molti problemi legate alla gestione delle interruzioni:

- Le interruzioni hanno priorità più elevata dei processi, possono rallentare applicazione hard real-time: non so quando periferiche hardware possono alzare un'eccezione
- Si utilizza una gestione "separata", a 3 livelli:
 - La prima parte, o top half, si compone di interrupt handler e la parte immediata dell'ISR. Interrupt handler salva i registri sullo stack e li recupera dopo, inoltre prenota una esecuzione del bottom half, quindi verrà eseguita dopo
 - La parte di bottom half può compiere operazioni lunghe e generalmente interrompibili. È costituita da una procedura da eseguire quando nessun altro top half è in esecuzione, quindi o viene eseguita immediatamente prima di tornare ad eseguire un processo oppure si esegue per mezzo di un kernel thread.
- C'è anche problema dovuto al fatto che interruzioni possono interrompersi fra di loro, che è un problema perché le interruzioni non hanno priorità. Se ad esempio ho in esecuzione una interrupt di un processo a priorità maggiore, questa può essere interrotta.. Soluzione oùò essere non permettere interruzioni, ma non è una buona soluzione, perché potrei impedire ad una interruzione di priorità più alta di essere subito eseguita (interrompendo una a priorità minore)

8.1.2 Schedulazione in Linux

Lo scheduler è sofisticato e modulare:

- È basato su classi di schedulazione, ovvero gerarchie di moduli che implementano differenti politiche di gestione dei processi
- Scheduler generale si limita ad interrogare le classi in ordine di priorità, chiedendo i processi da porre in esecuzione
- Ci sono politiche di schedulazione real-time che sono definite nella classe di schedulazione a priorità massima.
- In generale, tutte le classi di schedulazione supportano sistemi multi-processore ed hanno overherad essenzialmente indipendente dal numero di processi attivi nel sistema

Per la schedulazione real-time, la classe a priorità più alta implementa uno scheduler di default interrompibili a priorità fissa, la classe dei processi real-time è quasi completamente distinta rispetto dai processi non real-time. Due algoritmi di schedulazione: FIFO e Round-Robin. Scheduler è basato su una

coda di processi eseguibili singola, con 100 livelli di priorità (può essere problema per più di 100 processi, ma si può risolvere).

Non è problema implementare algoritmi come RM o DM per scheduler a priorità fissa. Esiste però meccanismo di sicurezza, per evitare che processo real-time che non termina mai mandi in blocco il processo: se processo real-time non rilascia mai la CPU, non può essere nemmeno killato perché bisogna interagire con shell. Il meccanismo quindi riserva una piccola percentuale di CPU ai processi in user mode (è possibile disabilitarlo accedendo ad appositi file "virtuali").

Altra classe di schedulazione importante, SCHED_DEADLINE, con priorità intermedia tra quelle real-time e SCHED_NORMAL. Introdotta da due ricercatori italiani sviluppata dal 2009 al 2014. Classe introduce una schedulazione con priorità dinamica a livello di task, quindi EDF: processi Linux non hanno concetto di priorità, idea è stata quella di utilizzare il meccanismo dei server CBS. Ad ogni task deadline associa budget, periodo e scadenza relativa. Quando vado a schedulare uso EDF, ma sulla stessa classe deadline. Scadenze assolute sono calcolate con lo stesso criterio del CBS, tempo di esecuzione è il budget e lo stato di ogni task è descritto da una deadline assoluta e dal tempo di esecuzione residuo. Non è stato implementato EDF puro perché processi Linux non hanno concetto di scadenza relativa o assoluta nativa, ma in ogni caso la cosa che più interessa è definire qual'è la frazione fissata che ciascun task userà del tempo di processore. Nell'implementazione della SCHED_DEADLINE:

- Ogni task avrà un μs ogni $p \mu s$ per la sua esecuzione
- Alla sua invocazione, lo scheduler aggiorna le scadenze e sceglie usando EDF
- Quando task termina il budget, viene sospeso fino alla deadline usata per la schedulazione

Ho quindi una schedulazione EDF con isolamento temporale dei task, non è però un'implementazione canonica di un server CBS.

Meccanismi hanno comunque dei problemi:

- Un processo real-time può comunque essere interrotto e quindi rallentato dai gestori delle interrupt
- Linux implementa meccanismi di bilanciamento del carico sui processori: processo attivo su un processore potrebbe essere migrato su un altro processore. Problema è che attualmente non si considera la priorità del processo, ma solo il carico che il processo darà a quel processore, quindi non sono considerate le priorità dei processi.

La migrazione può quindi aumentare la latenza di esecuzione ed aumenta la latenza a causa dei context switch, inoltre potrei avere delle inversioni di priorità non dovute.

In effetti è possibile risolvere problema della non predicitività del carico: c'è meccanismo dell'affinità di CPU. Posso legare un task ad una determinata CPU,

ogni processo ha una bitmap che specifica su quali processori su cui il processo può essere eseguito.

Bitmap viene ereditata dai figli, impostabile mediante syscall o tramite comando (taskset). Posso realizzare sistema statico, partizionato ed in cui non si ha mai bilanciamento del carico.

Che soluzioni posso usare per risolvere gli altri problemi:

- Approccio mono-kernel: modifica radicale del kernel per eliminare le cause di impredictibilità
- Approccio dual-kernel: cerco di fare minori modifiche possibili al kernel, ma impedisco accesso diretto ai dispositivi hardware tramite inserimento di uno stato di software intermedio tra kernel e l'hardware. C'è una parte effettivamente real-time, realizzo sistemi mixed criticality: attività sia real-time che non.

8.2 Approccio mono-kernel

In questo approccio, codice del kernel va modificato in maniera pesante:

- Kernel deve essere interrotto quasi sempre
- Consentire una gestione predibile delle interruzioni
- Protezione dei processi real-time dall'esecuzione dei gestori delle interruzioni
- Servono meccanismi di priority-inheritance per accesso a risorse condivise

Ci sono diversi prodotti:

- MontaVista Hard Hat Linux (commerciale)
- TimeSys Linux (commerciale)
- Linux con patch PREEMPT_RT (open source)

8.3 Approccio dual kernel

Meno invasivo per le modifiche effettuate, che sono minime, si cerca poi di affiancarlo ad altri programmi per realizzare RTOS. Obiettivo è quello di far coesistere sullo stesso hardware Linux ed un RTOS, chiave è quella di introdurre uno strato intermedio tra hardware e il SO, si tenta di virtualizzare il minimo e indispensabile dell'hardware, tipicamente solo le interruzioni. Alcuni approcci:

- RTAI (open)
- Xenomai (open)
- RT-Linux (open, ma un po' abbandonato oggi)

- e molti altri commerciali...

Approccio dual-kernel: due SO a priorità differente sullo stesso hardware + strato di virtualizzazione software che è IRQ manager che decide, quando arriva interruzione hardware, a quale dei due sistemi passarla. Priorità del RTOS è maggiore di Linux, quindi quest'ultimo è attivo solo quando non ci sono processi real-time in esecuzione. Strato intermedio intercetta interrupt e le consegna a Linux solo quando questo è in esecuzione. Strato intermedio sostituisce sempre la gestione iniziale delle interruzioni: Linux deve interagire con l'IRQ manager, virtualizzo l'interrupt controller. Vantaggio è che in realtà è possibile implementarlo con i meccanismi di Linux, è possibile scrivere moduli che saranno RTOS e IRQ manager che attivo dopo l'inizializzazione di Linux.

Vantaggi di mono-kernel rispetto a questo:

- Applicazioni real-time sono applicazioni Linux, quindi posso usare standard POSIX
- Può usare stack di rete etc..., sviluppo di applicazioni è facilitato
- Interazione con periferiche hardware possono usare driver già esistenti
- Le modifiche per adattare Linux nell'ambito real-time, gradualmente sono incorporate nel kernel standard. Oggi è quasi possibile includere le patch PREEMPT_RT nel kernel Linux, quindi sarà possibile dire di integrare soluzioni per real-time nel kernel standard.

Vantaggi dell'approccio dual kernel:

- Prestazioni real-time non dipendono dal kernel Linux, quindi sono drasticamente migliori, garanzie di predicitività e schedulabilità migliori
- È possibile realizzare RTOS specifico diverso da Linux, ovvero realizzare l'applicazione real-time senza supporto di alcun RTOS.
- Il processo di certificazione del sistema real-time è limitato allo stato intermedio di astrazione dell'hardware ed ai componenti software real-time
- Non è necessario effettuare modifiche pesanti al kernel Linux, quindi è facile seguire l'evoluzione del kernel.
- Molte certificazioni richiedono comunque partizionamento spaziale e temporale del sistema.

8.4 Sistemi partizionati

Molti sistemi real-time devono essere partizionati in modo spaziale, ovvero dividere le risorse del sistema tra più applicazioni o SO in modo che ciascuno di essi non possa interferire con il funzionamento di tutti gli altri.

Ad esempio, ciascuna partizione può essere associata ad una determinata porzione di RAM, generalmente questo partizionamento è realizzato da un programma

chiamato separation kernel, che deve costruire "gabbie" per le applicazioni da cui esse non possono uscire.

Partizionamento temporale avviene sul tempo, ovvero della frazione di tempo delle CPU. Tipicamente un separation kernel fa una schedulazione ciclica delle partizioni, i SO e le applicazioni in esecuzione in ciascuna partizione non possono influenzare i tempi d'esecuzione delle altre partizioni, né possono ritardare la loro schedulazione.

Differenza dalla schedulazione di un RTOS: qui la schedulazione è essenzialmente in meccanismo software influenzabile da politiche di schedulazione, priorità etc..., in una partizione di un sistema real-time può essere "contenuto" un intero SO e tutte le sue applicazioni. Schedulazione temporale è qualcosa di rigido, non flessibile.

8.5 RTAI e ADEOS

RTAI è un SO real-time ed anche separation kernel nel quale è implementato il RTOS, sviluppato dal politecnico di Milano, sistema nasce dall'esigenza di avere RTOS a basso costo che permetesse di usare l'unità in virgola mobile in kernel mode. Inizialmente basato su GNU-DOS, oggi su Linux

È costituito da due componenti:

- Separation kernel, che è versione modificata di ADEOS
- Modulo di Linux che costituisce SO

ADEOS è un getto indipendente, nano-kernel che introduce virtualizzazione tra hardware e SO. In RTAI le periferiche non sono pilotate da ADEOS, ma dai SO. ADEOS intercetta le interrupt hardware ed usa meccanismo per trasmetterle ai vari SO, interessati. Nei SO reali ormai tutte le interruzioni sono asincrone, se SO non riceve interrupt non può agire con la periferica; quindi se SO non riceve da ADEOS l'interrupt non può interagire.

C'è hardware, ADEOS: Linux ed RTAI interagiscono con hardware perché hanno driver delle periferiche a cui sono interessati, ma la gestione iniziale delle interrupt passa tramite ADEOS: questo virtualizza le interrupt e le passa ai domini interessati. Ci sono due tipi di domini: quelli parzialmente consapevoli di ADEOS (Linux), che non possono usare direttamente il controller delle interruzioni, e quelli consapevoli, come RTAI, che possono anche controllare in maniera più esplicita come le interrupt sono propagate lungo la catena. Ogni dominio ADEOS è associato ad una priorità, ogni dominio è inserito in una catena in base alla priorità. Ogni singola interrupt viene propagata lungo tutta la catena, ciascun dominio può accettare l'interruzione ed a quel punto questa viene gestita lì mediante hardware del dominio. Dominio può anche bloccare l'interrupt, quindi il procedimento lungo la catena.

Se dominio è consapevole può scegliere di ignorare l'interrupt oppure gestirla/bloccarla, mentre domini parzialmente consapevoli ricevono solo interrupt virtuali (propagate da ADEOS).

Sia ADEOS che RTAI sono moduli del kernel Linux, RTAI dominio di massima

priorità in ADEOS, Linux è di priorità minore. Occorre aggiungere un supporto per la gestione della catena delle interrupt in Linux.

RTAI usa ADEOS modificato, scorciatoie per cui alcune interruzioni possono essere gestire direttamente da RTAI o da Linux senza passare per ADEOS.

Schedulazione in RTAI:

- Scheduler è sia clock che event driven
- Meccanismo entra in esecuzione se riceve interrupt di timer, o all'uscita di una ISR o alla sospensione spontanea di un task.

In RTAI è possibile schedulare processi Linux (LXRT), che quindi vengono promossi a processi hard real-time e sottratti allo scheduler Linux. Processi non possono usare syscall di Linux perché diventano processi RTAI.

È facile far dialogare Linux ed RTAI: applicazione multi-threaded, con almeno un thread RTAI ed uno Linux. Ciascun thread verrà schedulato da Linux o da RTAI, ma condividono comunque spazio di memoria, ma questo viola principio di separazione spaziale.

Ci sono task nativi RTAI, una specie di kernel thread schedulati solo da RTAI, che quindi offrono cambi di contesto veloci. Possibile affer si che utilizzino memoria fisica specifica modificando a priori lo spazio di memoria che userà Linux da quello che userà RTAI, ma i kernel thread non potranno usare syscall kernel Linux; tipicamente si affianca a task LXTR.

RTAI supporta sistemi multi-processore, tipicamente però ogni task è forzato ad eseguire su una sola CPU, lo decide o il progettista o RTAI stesso. Supporto ad algoritmi di schedulazione FIFO, Round Robin, EDF e RM come anche alcuni meccanismi di priority inheritance.

8.6 Linux PREEMP_RT

Patch che nasce come progetto open-source, ex dipendente di MontaVista. Scopo era di rendere il kernel predicable, interrompibile in ogni momento. Patch viene modificata di pari passo al kernel Linux standard, i meccanismi vengono gradualmente introdotti nel kernel standard. Ci sono due obiettivi principali:

- ridurre le dimensioni (in termini di durata di esecuzione) delle sezioni non interrompibili del kernel
- permettere l'esecuzione dei gestori delle interruzioni in un contesto schedulabile con priorità come quelle dei processi.

Meccanismi di PREEMP_RT, idea fondamentale consiste nella sostituzione di alcuni meccanismi di sincronizzazione base del kernel, in quanto queste di base disabilitano l'interrompibilità ed iterano in un ciclo aspettando che variabile di condizione cambi stato. Meccanismo degli rt-mutex, a metà tra spinlock e semaforo: cominciano a comportarsi come spinlock, se variabile non cambia stato la primitiva diviene un semaforo (passa da attiva a passiva).

Gli rt-mutex hanno anche protocollo di priority-inheritance, inoltre l'operazione

di acquisizione di uno spinlock può essere interrotta, quindi non basta disabilitare le interruzioni, è solo idea generale.

Altro concetto fondamentale è che tutte le ISR sono gestite da kernel thread, le priorità quindi confrontate con quelle dei task real-time, kernel thread è entità schedulabile quindi assegno una priorità alle ISR delle interruzioni hardware e quindi a renderle schedulabile. Però interruzioni hardware possono ancora rallentare i processi real-time perché vanno ancora eseguite le top half, parte di gestione che fa attivare il kernel thread e quindi processo real-time può essere interrotto un n° impredicibile di volte che però sarà operazione piccola, quindi impatto minore.

Alcune interrupt, che per loro natura sono gestite in maniera "tradizionale", ma sono predibili. Questo meccanismo serve per i driver di tutte le periferiche del sistema. scritti da altri developer senza garanzie.

Nei sistemi multi-processori il metodo migliore per garantire la predicitività è partizionare, si può fare anche con PREEMT_RT. La patch PREEMPT_RT mette mano al bilanciamento del carico delle CPU:

- Non è possibile migrare processo mentre è in esecuzione
- Il bilanciamento viene fatto o prima dell'esecuzione o dopo interruzione/terminazione
- Il kernel cerca di non sovraccaricare un processore con processi di priorità elevata
- Quando processo ad alta priorità si sveglia e dovrebbe interrompere un processo corrente, si valuta un'euristica per decidere se è meglio migrarlo su un altro processore.

Patch valuta quindi questi 3 aspetti principali.