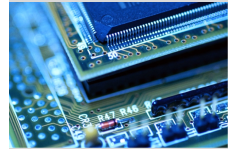


29/11/22

Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

Protocollo
priority-ceiling

SERT'20

R9.1

Lezione R9

Controllo d'accesso alle risorse condivise – I

Sistemi embedded e real-time

23 ottobre 2020

Marco Cesati

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

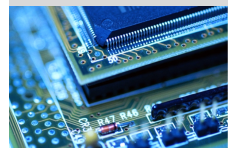
Di cosa parliamo in questa lezione?

In questa lezione parleremo di protocolli di controllo d'accesso
alle risorse condivise

- 1 Modello di sistema con risorse
- 2 Il controllo d'accesso
- 3 Il protocollo NPCS
- 4 Il protocollo priority-inheritance
- 5 Il protocollo priority-ceiling

Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

Protocollo
priority-ceiling

SERT'20

R9.2

Modello di sistema con risorse

- Singolo processore
- Risorse riciclabili seriali di tipo R_1, \dots, R_p
- Ciascun tipo di risorsa R_i ha ν_i unità di risorsa (stampa file, max 10 contemporanee) indistinguibili
- Ogni unità di risorsa è assegnabile ad un solo job alla volta
- Se R_i ha ∞ unità di risorsa non vale la pena considerarla nel modello $\Rightarrow \nu_i$ è sempre finito
- Esempi tipici: semafori, mutex, spin lock, stampanti, ...

Come modellare una risorsa R che può essere utilizzata contemporaneamente da un numero finito $n > 1$ di job?

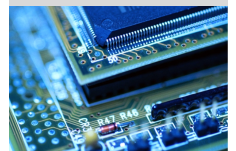
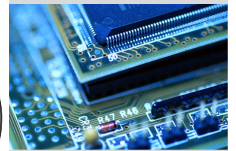
R ha $\nu = n$ unità esclusive (nessun job possiede più di 1 unità)

Come modellare una risorsa R che ha una intrinseca dimensione finita (es.: memoria)? uso le "pagine" di memoria

R ha ν unità di risorsa, e una unità rappresenta il più piccolo blocco di risorsa assegnabile

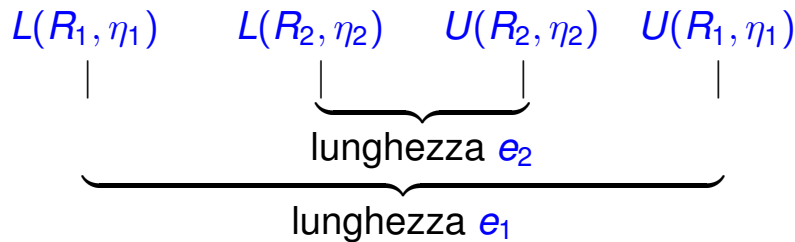
Richieste e rilasci di risorse

- Un job che deve acquisire η unità della risorsa R_i per procedere nell'esecuzione effettua una richiesta $L(R_i, \eta)$ no unità che vuole
- Se la richiesta è soddisfatta, il job continua l'esecuzione
- Altrimenti il job è bloccato (la sua esecuzione è sospesa) uniprocessore \Rightarrow attesa passiva
- Quando il job non ha più necessità della risorsa, esegue un rilascio $U(R_i, \eta)$ ($U \doteq \text{unlock}$, $L \doteq \text{lock}$)
- Spesso (ma non sempre!) il controllo di accesso alle risorse è affidato a primitive di lock/unlock (tipicamente semafori e mutex del sistema operativo)
- Spesso una risorsa R_i ha una sola unità disponibile ($\nu_i = 1$); abbreviamo $L(R_i, 1) = L(R_i)$ e $U(R_i, 1) = U(R_i)$
- Due job hanno un conflitto di risorse se entrambi richiedono una risorsa dello stesso tipo (non per forza stesso momento)
- Due job si contendono una risorsa se uno dei due richiede una unità di risorsa che è già posseduta dall'altro job

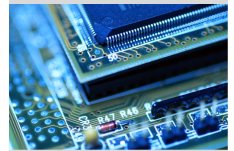


Sezioni critiche

- 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 annidate, ma **assumiamo che i rilasci sono sempre LIFO** (Se chiedo R_1 e R_2 , **RILASCIO R_2 e poi R_1**)
- 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:
chiedo η_1 unità di R_1 per tempo " e_1 ", e chiedo R_2 , **LIFO**

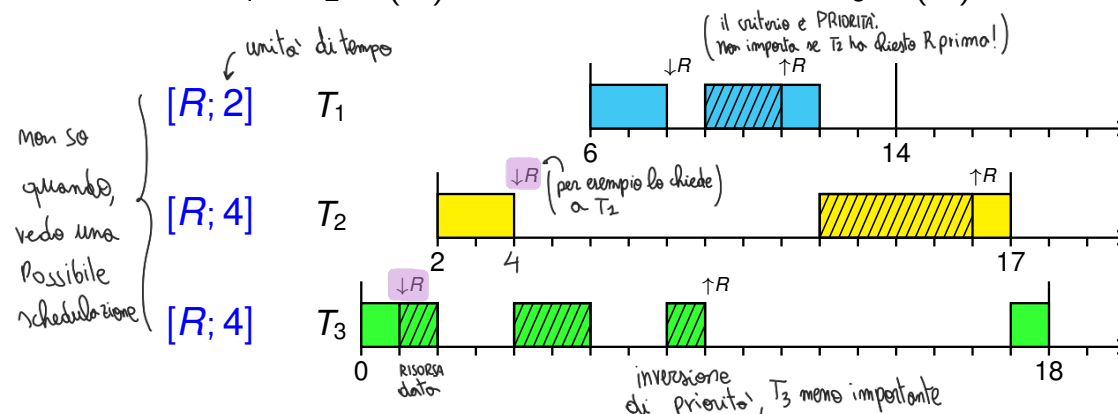


Nella notazione non sono indicati gli istanti iniziali delle sezioni critiche, ma solo il loro annidamento, assumo però casi peggiori

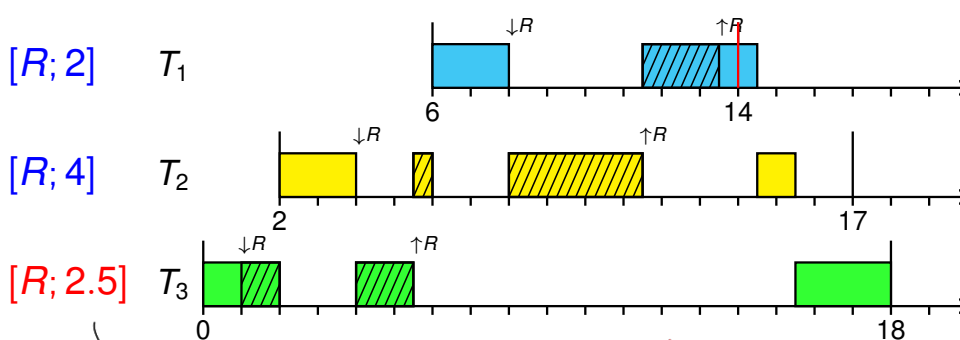


Esempi di schedulazione EDF con una unità di risorsa

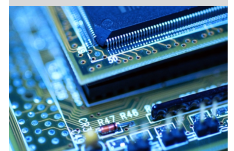
Task: $T_1=(6, 8, 5, 8)$, $T_2=(2, 15, 7, 15)$, $T_3=(18, 6)$
Per T_1 e T_2 : $L(R)$ a inizio esec. +2. Per T_3 : $L(R)$ a +1



Le inversioni di priorità causano **anomalie di schedulazione!**



↓ "lavoro da meno" ma non è meglio! Variazioni favorevoli peggiorano!

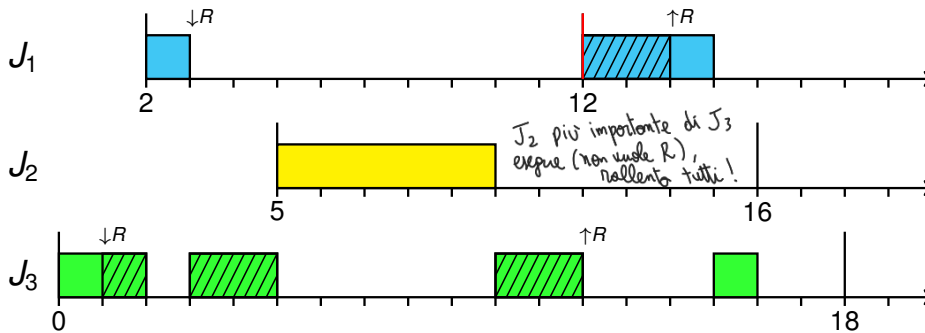


Controllo d'accesso alle risorse

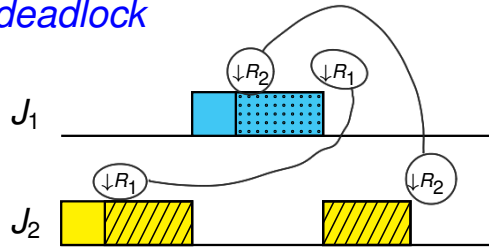
Nei sistemi real-time è sempre necessario implementare un algoritmo per il controllo d'accesso alle risorse condivise, altrimenti: devo evitare i casi di prima!

Le inversioni sono inevitabili, io voglio evitare che J_2 rallenti J_3 con risorse!

- le inversioni di priorità sono **non controllate**, cioè **arbitrariamente lunghe** (Sha, Rajkumar, Lehoczky 1990)



- sono possibili **deadlock**

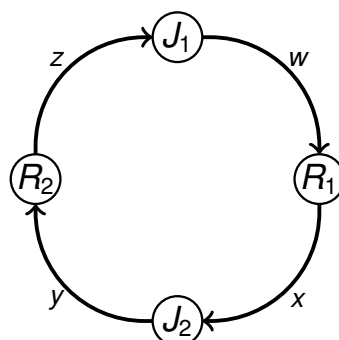


Grafi d'attesa

È possibile rappresentare la mutua relazione tra job e risorse tramite **grafi d'attesa**:

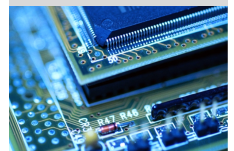
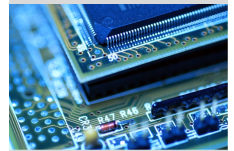
- I **nodi del grafo** sono i **job** ed i **tipi di risorse**
- Un arco orientato con etichetta x da **una risorsa ad un job** indica che **il job ha allocato x unità della risorsa**
- Un arco orientato con etichetta x da **un job ad una risorsa** indica che il job ha richiesto x unità della risorsa e la **richiesta non può essere soddisfatta**

Cosa rappresenta un ciclo nel grafo d'attesa?



Un **deadlock**!

A noi serve controllare le inversioni arbitrariamente lunghe!



Protocollo NPCS

Il più semplice protocollo di controllo d'accesso alle risorse è **NPCS** (Nonpreemptive Critical Section, Mok 1983)

Protocollo NPCS

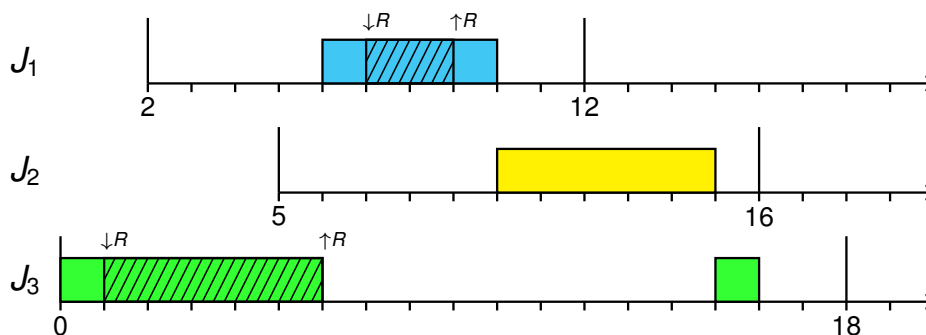
Un job avente una risorsa assegnata **non può essere interrotto**

È possibile avere deadlock utilizzando NPCS?

No, ma solo a condizione che il job **non si auto-sospenda** all'interno di una sezione critica

→ se Job si autosospende "crolla tutto".

Esempio di schedulazione con **NPCS**:



Tempo di blocco per conflitto di risorse

Sia $b_i(rc)$ il *tempo di blocco dovuto ad un conflitto di risorse*

Con task a priorità fissa T_1, \dots, T_n e **NPCS** vale

$$b_i(rc) = \max_{i+1 \leq k \leq n} (c_k) \quad \text{prende il massimo!}$$

→ task di priorità inferiori.

(c_k = tempo di esecuzione della più lunga sezione critica di T_k)

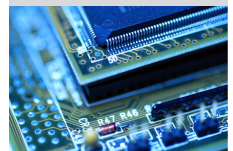
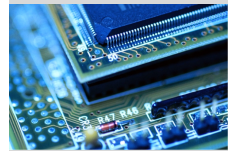
Qual è la formula per $b_i(rc)$ con schedulazione EDF?

- **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 $D_i < D_j$
- $b_i(rc) = \max\{c_k : k \text{ tale che } D_k > D_i\}$

Qual è il limite del protocollo NPCS?

Un job può essere bloccato da un job di priorità inferiore anche quando **non esiste contesa su alcuna risorsa** (non ottimale)

NPCS è comunque diffuso perché è semplice, non richiede dati sull'uso delle risorse dei job, è facile da implementare e può essere usato sia in sistemi a priorità fissa che dinamica



Protocollo priority-inheritance

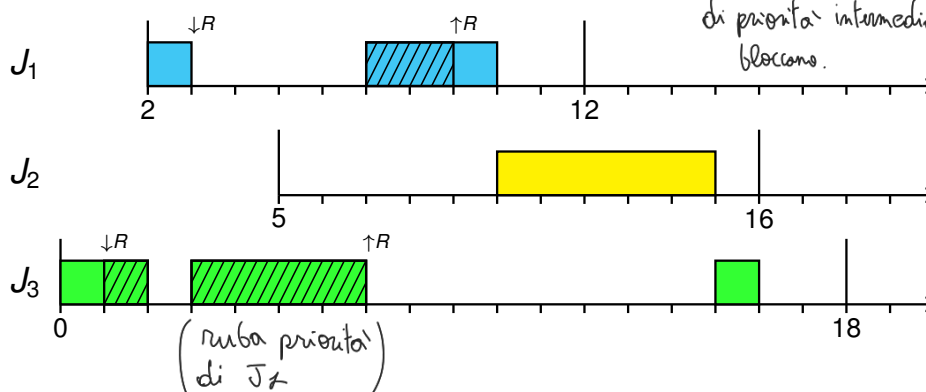
Proposto da Sha, Rajkumar, Lehoczky (1990):

- Adatto ad ogni scheduler priority-driven
- Non basato sui tempi di esecuzione dei job
- Evita il fenomeno della inversione di priorità incontrollata, *NON EVITA deadlock*

Versione base: Una sola unità per ogni tipo di risorsa

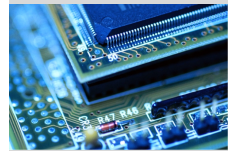
(Job bloccato perché altro ha risorsa)

Idea: cambiare le priorità se esistono **contese** sulle risorse per evitare che un job che blocca un altro job di priorità più alta sia rallentato da job di priorità intermedia tra i due, "eredita" priorità del Job superiore, con task di priorità intermedia che mi bloccano.



Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

Protocollo
priority-ceiling

SERT'20

R9.11

Protocollo priority-inheritance (2)

Regola di schedulazione

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

Regola di allocazione

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

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

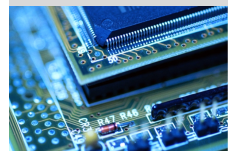
Regola di trasferimento della priorità

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 $\pi_\ell(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R

*non quella "nativa",
se T3 eredita
priorità T2, finché
è con eredita quella!*

Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

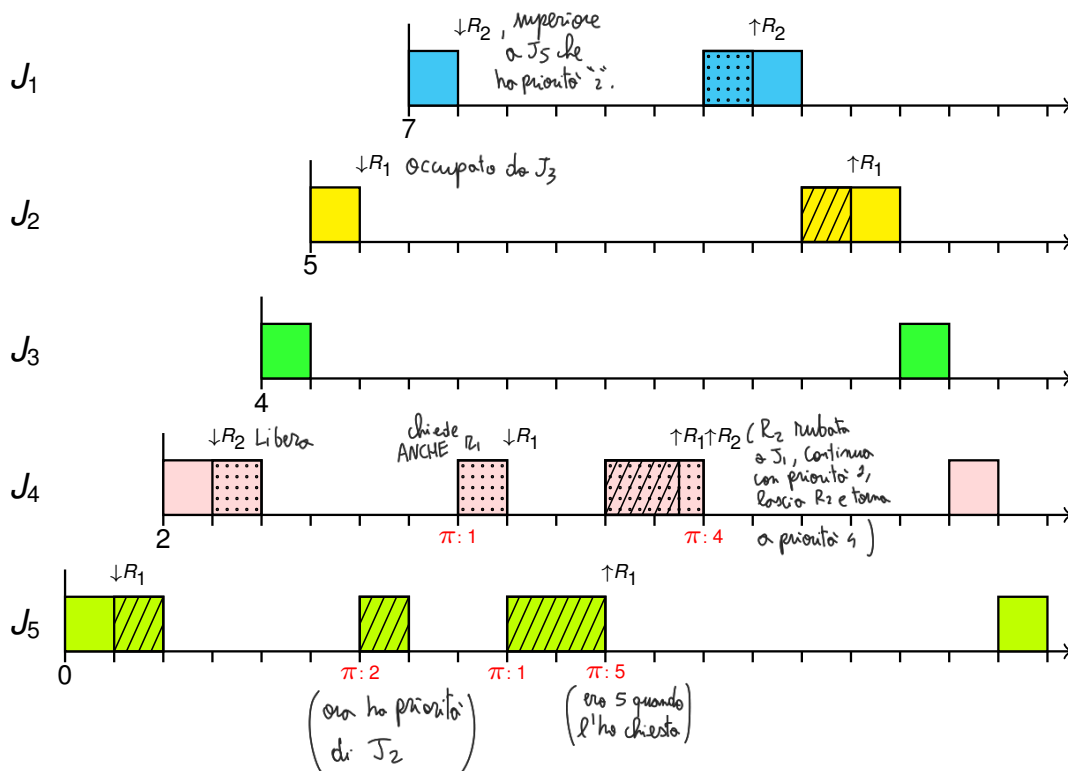
Protocollo
priority-ceiling

SERT'20

R9.12

Schedulazione a priorità fissa e priority-inheritance

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

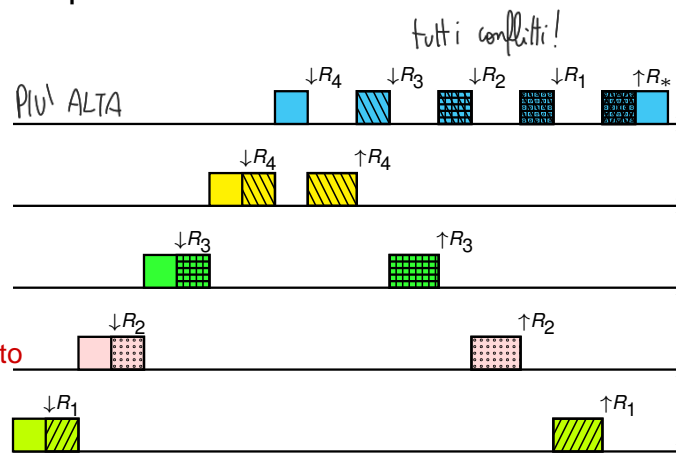


Limiti del protocollo priority-inheritance

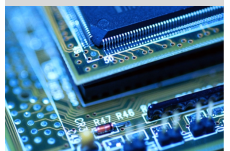
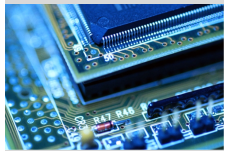
Quali sono le limitazioni del protocollo priority-inheritance?

- Non evita i deadlock
- Introduce nuovi casi di blocco: un job con priorità corrente $\pi(t)$ può bloccare ogni job con priorità assegnata minore di $\pi(t)$, ma è conseguenza delle scelte fatte.
- Non riduce i tempi di blocco dovuti ai conflitti sulle risorse al minimo teoricamente possibile (NO contese)

Un job che accede a v risorse ed ha conflitti di risorse con k job di priorità assegnata minore può ~~bloccare~~ essere bloccato $\min(v, k)$ volte



• può portare a tempi di blocco elevati.



Protocollo priority-ceiling

Proposto da Sha, Rajkumar, Lehoczky (1988, 1990):

- Adatto a scheduler con priorità fissa, NO DINAMICA.
- Basato sulle richieste di risorse dei job (prefissate)
- Evita l'inversione di priorità incontrollata e i deadlock

Versione base: Una sola unità per ogni tipo di risorsa

Idea: associare ad ogni risorsa R il valore **priority ceiling** $\Pi(R)$ ^{STATICO} pari alla massima (*) priorità dei job che fanno uso di R (numerico e più piccolo $Priorità \in 1 \dots n$)

Ad ogni istante t il valore **current priority ceiling** $\hat{\Pi}(t)$ è pari a:

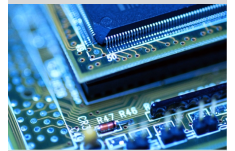
- la massima (*) priorità $\Pi(R)$ fra tutte le risorse del sistema correntemente in uso al tempo t
- al valore convenzionale Ω di priorità inferiore a quella di qualunque task se nessuna risorsa è in uso

(*)

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

Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

Protocollo
priority-ceiling

SERT'20

R9.15

Protocollo priority-ceiling (2)

Regola di schedulazione, basato su priority inheritance

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 prefissata

Regola di allocazione, diversa da inheritance

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

- (a) $\pi(t) > \hat{\Pi}(t)$, oppure (chi richiede ha priorità corrente > priority ceiling del sistema)
- (b) J possiede una risorsa il cui **priority ceiling** è uguale a $\hat{\Pi}(t)$

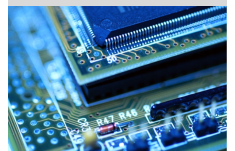
Altrimenti J è sospeso (bloccato)

Regola di trasferimento della priorità

Se J_ℓ blocca J , J_ℓ eredita la priorità corrente $\pi(t)$ di J finché J_ℓ non rilascia l'ultima risorsa R tale che $\Pi(R) \geq \pi(t)$; a quel punto la priorità corrente di J_ℓ torna ad essere la priorità $\pi_\ell(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R

Controllo d'accesso
alle risorse condivise

Marco Cesati



Schema della lezione

Modellare le risorse

Protocollo NPCS

Protocollo
priority-inheritance

Protocollo
priority-ceiling

ma la risorsa che
ha alzato il ceiling
a quel livello deve
quella richiesta io.

SERT'20

R9.16

Priority ceiling
selle risorse

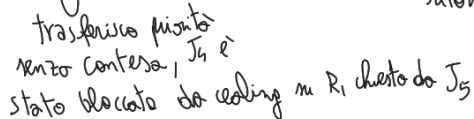
$$\begin{aligned}\Pi(R_1) &= 2 \\ \Pi(R_2) &= 1\end{aligned}$$

J_2 è risorsa più imp
che la richiede
**Controllo d'accesso
alle risorse condivise**

priority-ceiling

celing ritmo

J_1 è
più imp.
che lo
chiede



priority
bar

priorita alta

- RISORSA
OCCUPATA

priority ceiling

Se chiedo R_1 e R_2 e
altro R_2 e R_1 deadlock,
o "ordine" (od. es. SOLO R_1 e R_2)
è tutto!
L'ordine deve essere definito

SERT'20 R9.18

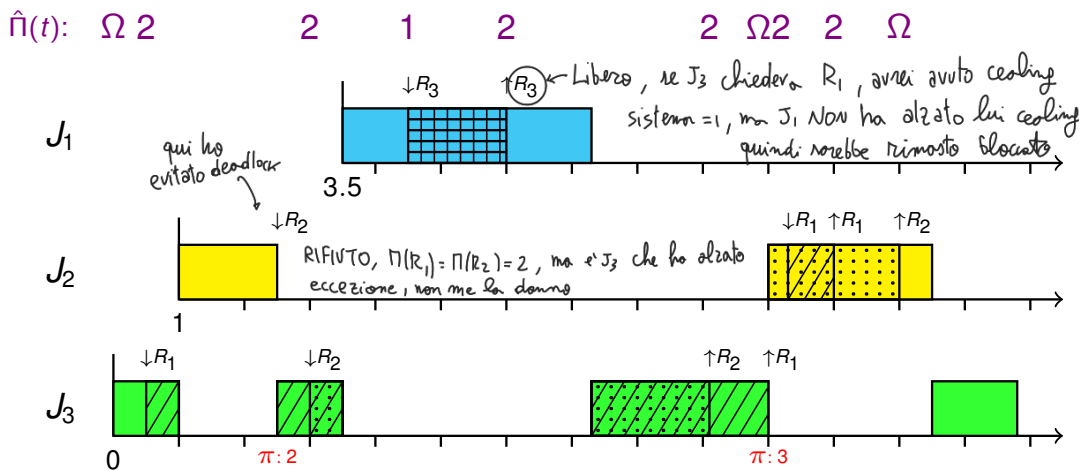
- I deadlock possono essere evitati se tutti i job acquisiscono le risorse annidate rispettando un unico ordinamento globale delle risorse (Havender, 1968)
- Il **priority ceiling** $\Pi(R)$ delle risorse e la regola di allocazione vietano le assegnazioni di risorse "fuori ordine"

forza ordinamento PARZIALE, ma SUFFICIENTE.

Forza ordinamento PARZIALE, ma SUFFICIENTE.

Come si evitano i deadlock nel protocollo priority ceiling

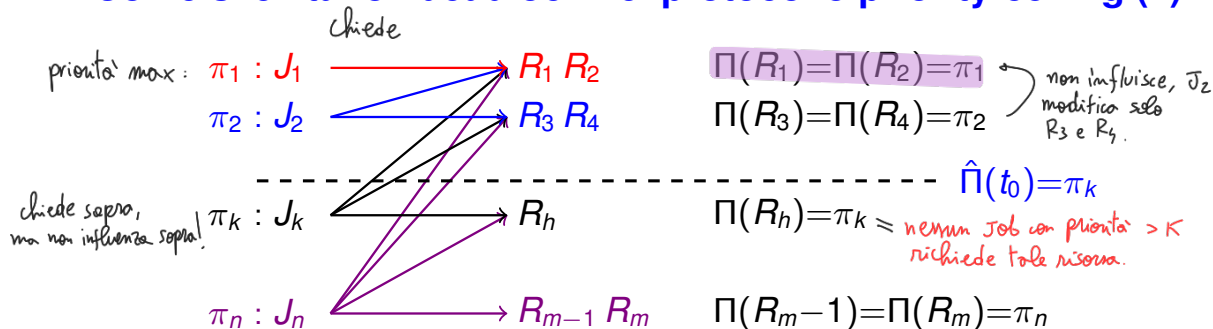
	J_1	J_2	J_3	$J_1: [R_3; 1.5]$	$J_2: [R_2; 2 [R_1; 0.7]]$	$J_3: [R_1; 4.2 [R_2; 2.3]]$
r	3.5	1	0			
e	3.8	4	6			
				$\Pi(R_1)=2$	$\Pi(R_2)=2$	$\Pi(R_3)=1$



Al tempo 2.5 J_2 richiede R_2 , ma la richiesta viene rifiutata anche se R_2 è libera \Rightarrow si evita un possibile deadlock con J_3

I job con priorità corrente maggiore di $\hat{\Pi}(t)$ possono acquisire risorse senza rischiare deadlock con le risorse già assegnate

Come si evitano i deadlock nel protocollo priority-ceiling (2)



Se al tempo t_0 un job J richiede una risorsa R e $\pi_J(t_0) > \hat{\Pi}(t_0)$:

- J non chiederà mai alcuna risorsa già assegnata al tempo t_0 , nemmeno Priority Ceiling $\equiv \pi_J$, J in futuro non chiederà mai risorse sopra!

\Rightarrow nessun deadlock con risorse già assegnate

- Nessun job con priorità maggiore di $\pi_J(t_0)$ chiederà alcuna risorsa già assegnata al tempo t_0 , nessuno dei job sopra ha chiesto risorsa già assegnata (senza R stava sopra)

\Rightarrow nessun job che già possiede una risorsa al tempo t_0 potrà interrompere J e richiedere R

\Rightarrow Il protocollo priority-ceiling evita i deadlock

evito il fuori ordine grazie a questo "spartiacque"

