Lezione R9

Controllo d'accesso alle risorse condivise – I

Sistemi embedded e real-time

4 dicembre 2012

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

Modello di sistema con risorse

- Singolo processore
- Risorse riciclabili seriali di tipo $R_1, \ldots R_\rho$
- Ciascun tipo di risorsa R_i ha ν_i unità di risorsa 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 $\Rightarrow$ ν_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) ?

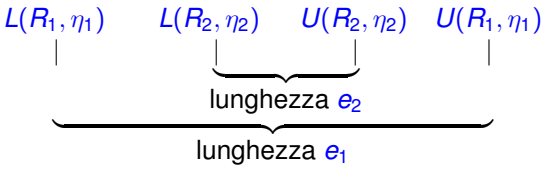
R ha ν unità di risorsa, e una unità rappresenta il più piccolo blocco di risorsa assegnabile

Richieste e rilasci di risorse

- Un job che deve acquisire η unità della risorsa R_i per procedere nell'esecuzione effettua una *richiesta* $L(R_i, \eta)$
- Se la richiesta è soddisfatta, il job continua l'esecuzione
- Altrimenti il job è bloccato (la sua esecuzione è sospesa)
- Quando il job non ha più necessità della risorsa, esegue un rilascio $U(R_i, \eta)$
- 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
- Due job *si contendono* una risorsa se uno dei due richiede una risorsa che è già posseduta dall'altro job

Sezioni critiche

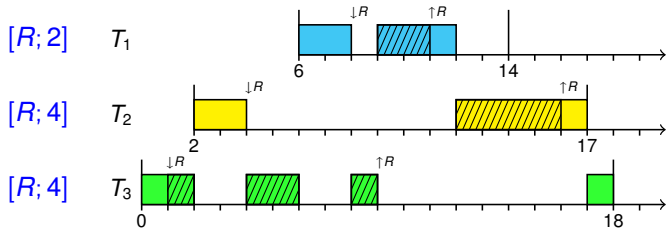
- Si definisce *sezione critica* un segmento di esecuzione di job che inizia con $L(R_i, \eta)$ e termina con $U(R_i, \eta)$
- Le richieste di risorse di un job possono essere annidate, ma assumiamo che i rilasci sono sempre LIFO
- Una sezione critica non contenuta in alcun'altra sezione critica è detta *esterna*
- La notazione $[R_1, \eta_1; e_1\ [R_2, \eta_2; e_2]]$ corrisponde a:



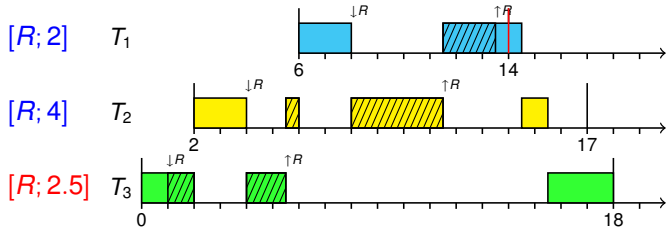
Nella notazione non sono indicati gli istanti iniziali delle sezioni critiche, ma solo il loro annidamento

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



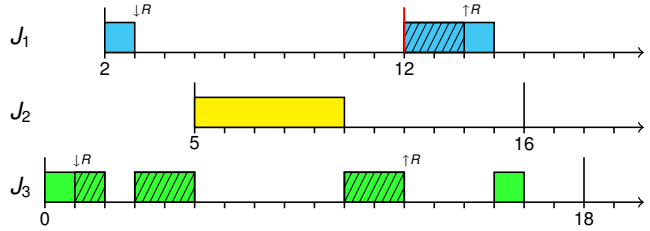
Le inversioni di priorità causano anomalie di schedulazione!



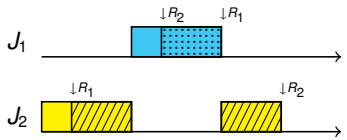
Controllo d'accesso alle risorse

Nei sistemi real-time è sempre necessario implementare un algoritmo per il controllo d'accesso alle risorse condivise, altrimenti:

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



- sono possibili *deadlock*



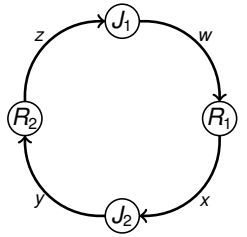
Grafi d'attesa

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!



Protocollo NPCS

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

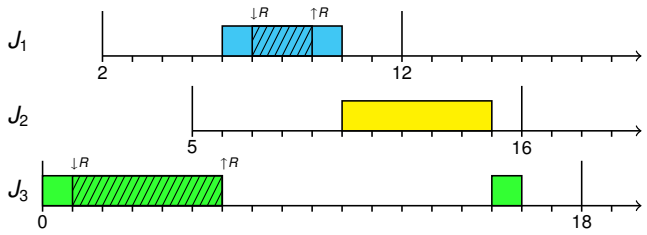
Protocollo NPCS

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

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 $T_1, \ldots T_n$ e NPCS vale

$$b_i(rc) = \max_{i+1 \le k \le n} (c_k)$$

(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

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

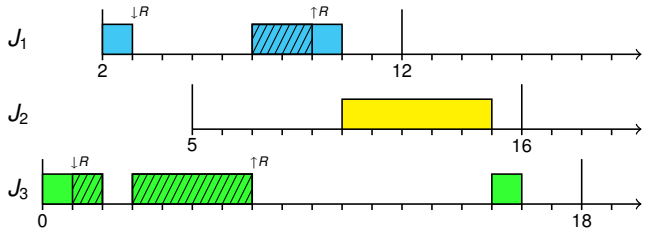
Protocollo priority-inheritance

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

Versione base: Una sola unità per ogni tipo di risorsa

Idea: cambiare le priorità se esistono conflitti 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



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:

(a) Se R è disponibile, R è assegnata a J

(b) Se R non è disponibile, J è sospeso (bloccato)

Regola di trasferimento della priorità

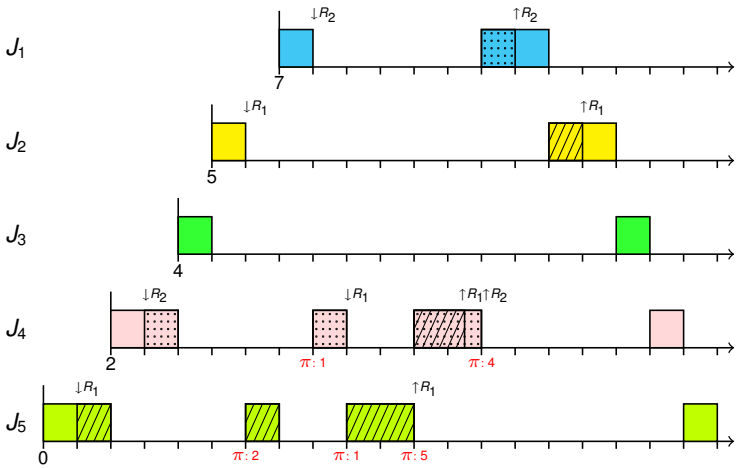
Quando un job J viene bloccato a causa di un conflitto di risorsa R, il job J_ℓ che blocca J eredita la priorità corrente $\pi(t)$ di J finché non rilascia R; a quel punto, la priorità corrente di J_ℓ torna ad essere la priorità $\pi_\ell(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R

Schedulazione a priorità fissa e priority-inheritance

	J_1	J_2	J_3	J_4	J_5
r	7	5	4	2	0
e	3	3	2	6	6

J_1:$[R_2; 1]$ J_2:$[R_1; 1]$

J_4:$[R_2; 4\,[R_1; 1.5]]$ J_5:$[R_1; 4]$

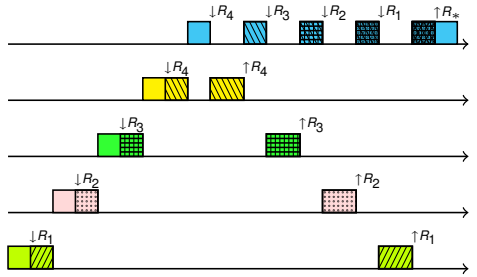


Limiti del protocollo priority-inheritance

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 $\pi(t)$
- Non riduce i tempi di blocco dovuti ai conflitti sulle risorse al minimo teoricamente possibile

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



Protocollo priority-ceiling

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

- Adatto a scheduler con priorità fissa
- 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* $\Pi(R)$ pari alla massima (*) priorità dei job che fanno uso di R

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

Protocollo priority-ceiling (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 prefissata

Regola di allocazione

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

(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

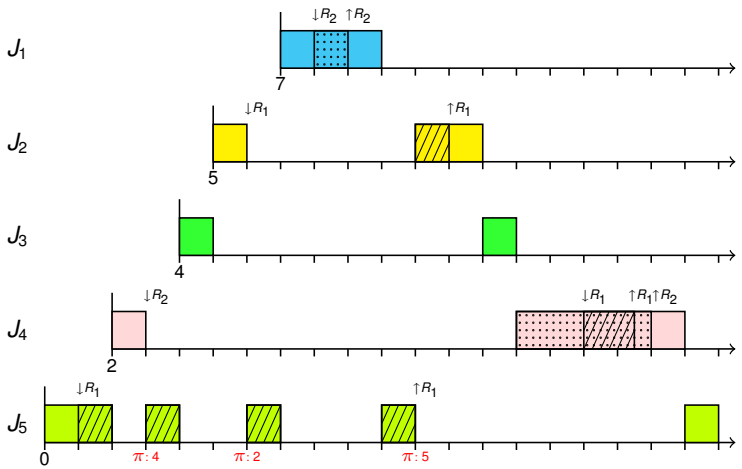
Se J_ℓ blocca J, J_ℓ eredita la priorità corrente $\pi(t)$ di J finché J_ℓ non rilascia l'ultima risorsa R tale che $\Pi(R) \geq \pi(t)$; a quel punto la priorità corrente di J_ℓ torna ad essere la priorità $\pi_\ell(t')$ che aveva al momento t' in cui aveva acquisito la risorsa R

Esempio di schedulazione con priority-ceiling

	J_1	J_2	J_3	J_4	J_5
r	7	5	4	2	0
e	3	3	2	6	6

J_1:$[R_2; 1]$ J_2:$[R_1; 1]$ $\Pi(R_1)$=2
J_4:$[R_2; 4\,[R_1; 1.5]]$ J_5:$[R_1; 4]$ $\Pi(R_2)$=1

$\hat{\Pi}(t)$: Ω 2 1 2 Ω2 Ω 1 Ω



Tipi di blocco nel protocollo priority-ceiling

Nel protocollo priority-ceiling, in quali diversi casi un job J_ℓ può bloccare un job J_h con priorità assegnate $\pi_\ell < \pi_h$?

- 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 blocca direttamente un job di priorità maggiore di J_h
- Blocco dovuto a priority-ceiling (o *avoidance blocking*): J_h ha richiesto una risorsa R ma J_ℓ possiede un'altra risorsa R' tale che $\Pi(R') = \hat{\Pi}(t)$

Perché il protocollo priority-ceiling evita i deadlock?

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

Controllo d'accesso alle risorse condivise
Marco Cesati

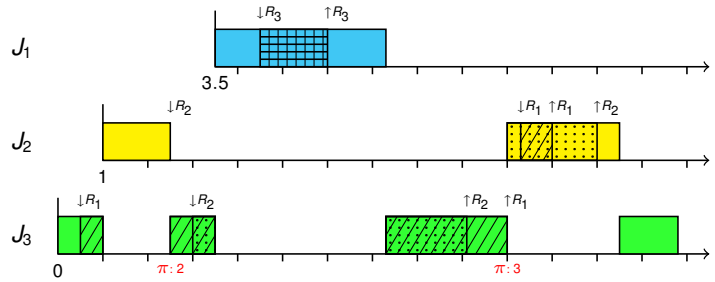
Schema della lezione
Modellare le risorse
Protocollo NPCS
Protocollo priority-inheritance
Protocollo priority-ceiling

Come si evitano i deadlock nel protocollo priority ceiling

	J_1	J_2	J_3
r	3.5	1	0
e	3.8	4	6

J_1:$[R_3; 1.5]$
J_2:$[R_2; 2\,[R_1; 0.7]]$ J_3:$[R_1; 4.2\,[R_2; 2.3]]$
$\Pi(R_1)$=2 $\Pi(R_2)$=2 $\Pi(R_3)$=1

$\hat{\Pi}(t)$: Ω 2 2 1 2 2 Ω2 2 Ω



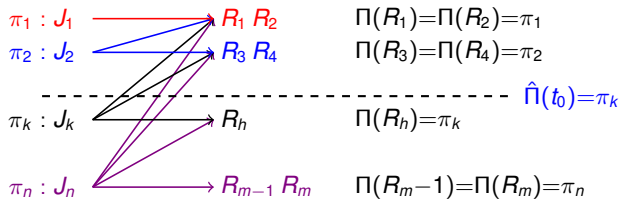
Al tempo 2.5 J_2 richiede R_2, ma la richiesta viene rifiutata anche se R_2 è libera ⇒ si evita un possibile deadlock con J_3

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

Controllo d'accesso alle risorse condivise
Marco Cesati

Schema della lezione
Modellare le risorse
Protocollo NPCS
Protocollo priority-inheritance
Protocollo priority-ceiling

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



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

- J non chiederà mai alcuna risorsa già assegnata al tempo t_0

  ⇒ nessun deadlock con risorse già assegnate
- Nessun job con priorità maggiore di $\pi(t_0)$ chiederà alcuna risorsa già assegnata al tempo t_0

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

⇒ Il protocollo priority-ceiling evita i deadlock

Controllo d'accesso alle risorse condivise
Marco Cesati

Schema della lezione
Modellare le risorse
Protocollo NPCS
Protocollo priority-inheritance
Protocollo priority-ceiling

Proprietà del protocollo priority-ceiling

- Assegna ad ogni risorsa il valore priority-ceiling che indica la massima priorità tra i job che usano la risorsa
- Mantiene aggiornato il current priority ceiling del sistema $\hat{\Pi}(t)$ che indica il massimo valore associato a tutte le risorse assegnate
- Job bloccanti ereditano la priorità dinamica dei job bloccati
- Al tempo t un solo job possiede tutte le risorse assegnate aventi priority ceiling uguale a $\hat{\Pi}(t)$
- Se un job ottiene una risorsa e $\pi(t) > \hat{\Pi}(t)$, nessun job di priorità uguale o superiore (compreso il job stesso) utilizza risorse già assegnate
- Se un job ottiene una risorsa e $\pi(t) \leq \hat{\Pi}(t)$, il job è il possessore di tutte le risorse assegnate con priority ceiling uguale a $\hat{\Pi}(t)$
- I deadlock sono evitati

Durata dei blocchi nel protocollo priority-ceiling

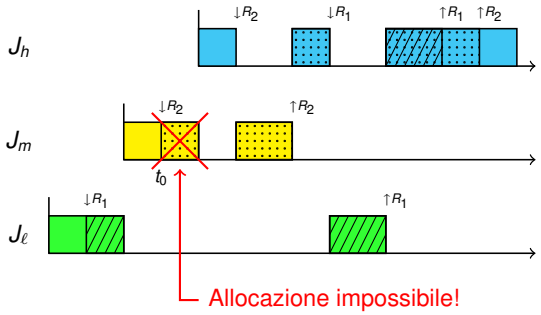
Teorema

Utilizzando il protocollo priority-ceiling un job può essere bloccato al massimo per la durata di una sezione critica

Il teorema è conseguenza di due proprietà:

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

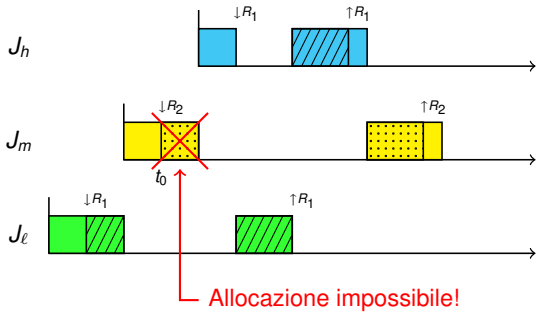
Unicità del job bloccante



- $\pi_h > \pi_m > \pi_\ell \quad \Rightarrow \quad \Pi(R_1) \geq \pi_h$ e $\Pi(R_2) \geq \pi_h$
- $\hat{\Pi}(t_0) \geq \Pi(R_1) \geq \pi_h$
- Requisito per allocazione a t_0: $\pi_m > \hat{\Pi}(t_0) \geq \pi_h$

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

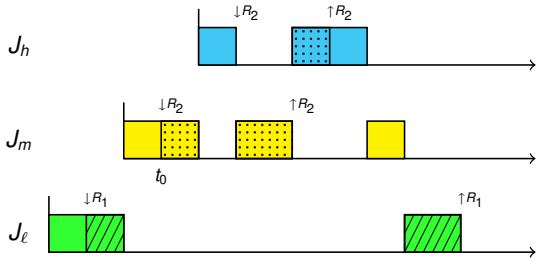
Unicità del job bloccante (2)



- $\pi_h > \pi_m > \pi_\ell \quad \Rightarrow \quad \Pi(R_1) \geq \pi_h$ e $\Pi(R_2) \geq \pi_m$
- $\hat{\Pi}(t_0) \geq \Pi(R_1) \geq \pi_h$
- Requisito per allocazione a t_0: $\pi_m > \hat{\Pi}(t_0) \geq \pi_h$

J_h non può essere bloccato da J_ℓ se J_ℓ è stato interrotto da J_m e J_m ha acquisito una risorsa

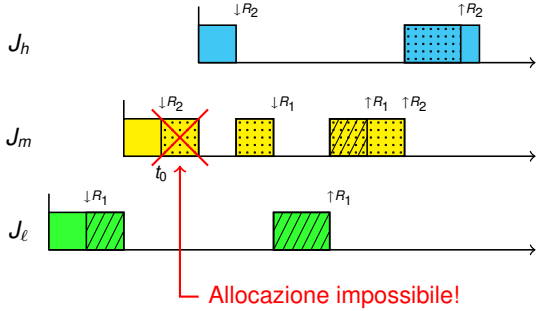
Unicità del job bloccante (3)



- $\pi_h > \pi_m > \pi_\ell \quad \Rightarrow \quad \Pi(R_1) \geq \pi_\ell$ e $\Pi(R_2) \geq \pi_h$
- $\hat{\Pi}(t_0) \geq \Pi(R_1) \geq \pi_\ell$
- Requisito per allocazione a t_0: $\pi_m > \hat{\Pi}(t_0) \geq \pi_\ell$

J_h può essere bloccato da J_m solo se J_m possiede la risorsa che ha il massimo priority ceiling tra tutte quelle in uso ($= \hat{\Pi}(t)$)

Impossibilità del blocco transitivo



- $\pi_h > \pi_m > \pi_\ell \quad \Rightarrow \quad \Pi(R_1) \geq \pi_m$ e $\Pi(R_2) \geq \pi_h$
- $\hat{\Pi}(t_0) \geq \Pi(R_1) \geq \pi_m$
- Requisito per allocazione a t_0: $\pi_m > \hat{\Pi}(t_0) \geq \pi_m$

Se J_m blocca J_h, J_m non può essere bloccato da J_ℓ

Tempo di blocco per conflitto di risorse

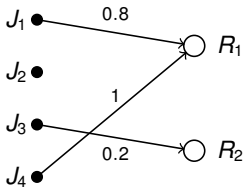
Il *tempo di blocco per conflitto di risorse* $b_i(rc)$ è il massimo ritardo di un job del task T_i causato da un conflitto di risorse

Come calcolare $b_i(rc)$ per il protocollo priority-ceiling?

Con priority ceiling esistono 3 tipi di blocco: blocco diretto, blocco per priority inheritance e blocco per priority ceiling

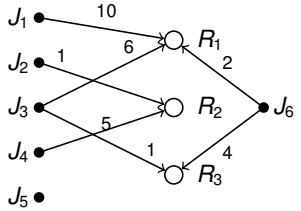
Poiché ogni job è bloccato al massimo per la durata di una sola sezione critica, è sufficiente per ciascun task determinare i valori massimi dei ritardi introdotti da ciascun tipo di blocco

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 $\quad \Rightarrow \quad b_1(rc) = 1$
- J_4 può bloccare J_2 e J_3 quando eredita $\pi_1 \quad \Rightarrow \quad b_2(rc) = b_3(rc) = 1$
- Ovviamente $b_4(rc) = 0$

Tempo di blocco per conflitto di risorse (2)



Blocco diretto:

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

Blocco per inheritance:

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

Blocco per ceiling:

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				*	

- $B_i(r,c) = \max\{B_d(j,c) : 1 \leq j \leq r-1\}$
- Se le priorità dei job sono tutte diverse, $B_c = B_i$ tranne che per i job che non utilizzano risorse (non bloccano)
- $b_i(rc) = \max_k\{B_d(i,k), B_i(i,k), B_c(i,k)\}$