Corso di Programmazione Concorrente

Semafori di Dijkstra

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- Il Problema della Mutua Esclusione
- Semafori
  - Spin-Lock
  - Semafori di Dijkstra
  - Schema di implementazione
- Espressività dei Semafori
- Attese Attive vs Attese Passive
- Cenni agli algoritmi *non-bloccanti*

Interazione tra Flussi di Esecuzione Concorrenti

- Due flussi di esecuzione possono essere:
  - disgiunti
  - interagenti
    - *competizione*: due o più f.d.e. chiedono l'uso di una risorsa comune riusabile e di molteplicità finita
    - *cooperazione*: due o più f.d.e. cooperano per raggiungere un obiettivo comune
- Per ora ci concentreremo sulla competizione

Competizione

- La competizione occorre ogni qualvolta che c'è una risorsa riusabile condivisa e di molteplicità finita (spesso seriale)
  - incrocio stradale
  - sportellista alle poste
  - stampante dipartimentale
  - rete ethernet
- In presenza di competizione è necessario "gestire" i possibili fenomeni di interferenza

Il Problema della Mutua Esclusione

- Adotteremo quasi sempre la strategia di evitare interferenze
- Fissiamo le idee con riferimento al più semplice dei problemi di competizione

Dati due F.d.E. P e Q e una risorsa R, garantire che:

a) In ogni istante R è libera oppure risulta assegnata ad uno solo tra P e Q (mutua esclusione)

b) Ciascuno dei due flussi di esecuzione deve sempre poter ottenere l'uso della risorsa (fairness)

Commenti al Problema della Mutua Esclusione

- La prima condizione impone la serializzazione nell'uso della risorsa
- La seconda condizione impone che nessun f.d.e. rimanga indefinitivamente in attesa della risorsa (*fairness* nello scheduling della risorsa)
  - attenzione: non bastano argomenti probabilistici per affermare che tale condizione è soddisfatta. La condizione di fairness deve essere garantita dall'algoritmo di scheduling, ovvero si deve dimostrare che non esiste nemmeno una seq. di esecuzione ammissibile in cui un f.d.e. non ottiene mai l'uso della risorsa
- Ovviamente ogni f.d.e. può usare la risorsa solo per un periodo finito di tempo

Struttura delle Soluzioni al Problema della Mutua Esclusione

Ogni soluzione prevede un protocollo di utilizzo della risorsa $\mathcal{R}$ articolato in tre fasi

1) richiesta di $\mathcal{R}$

2) utilizzo di $\mathcal{R}$

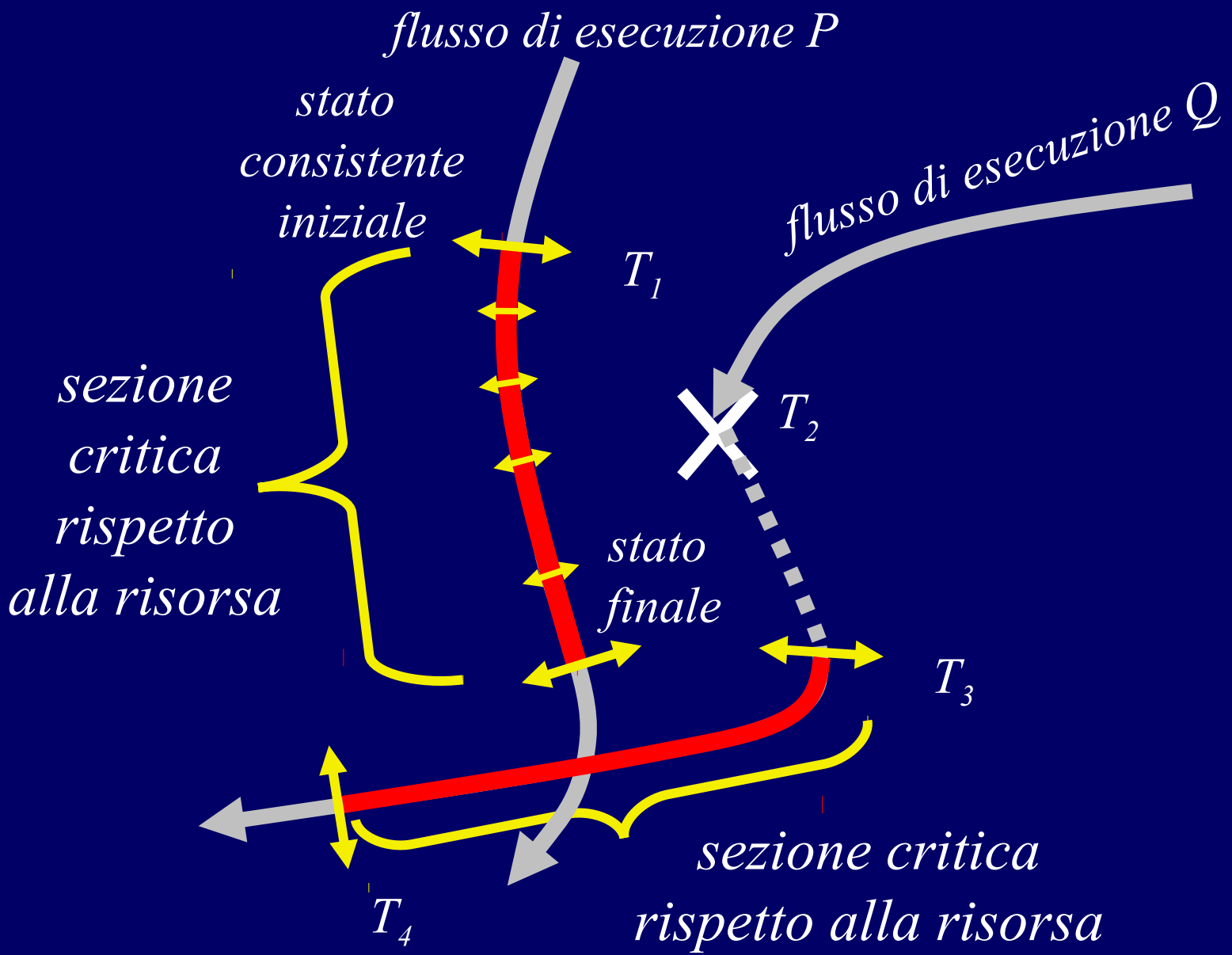
3) rilascio di $\mathcal{R}$

- La richiesta è necessaria per garantire la prima condizione
- Il rilascio è necessario per garantire la seconda condizione

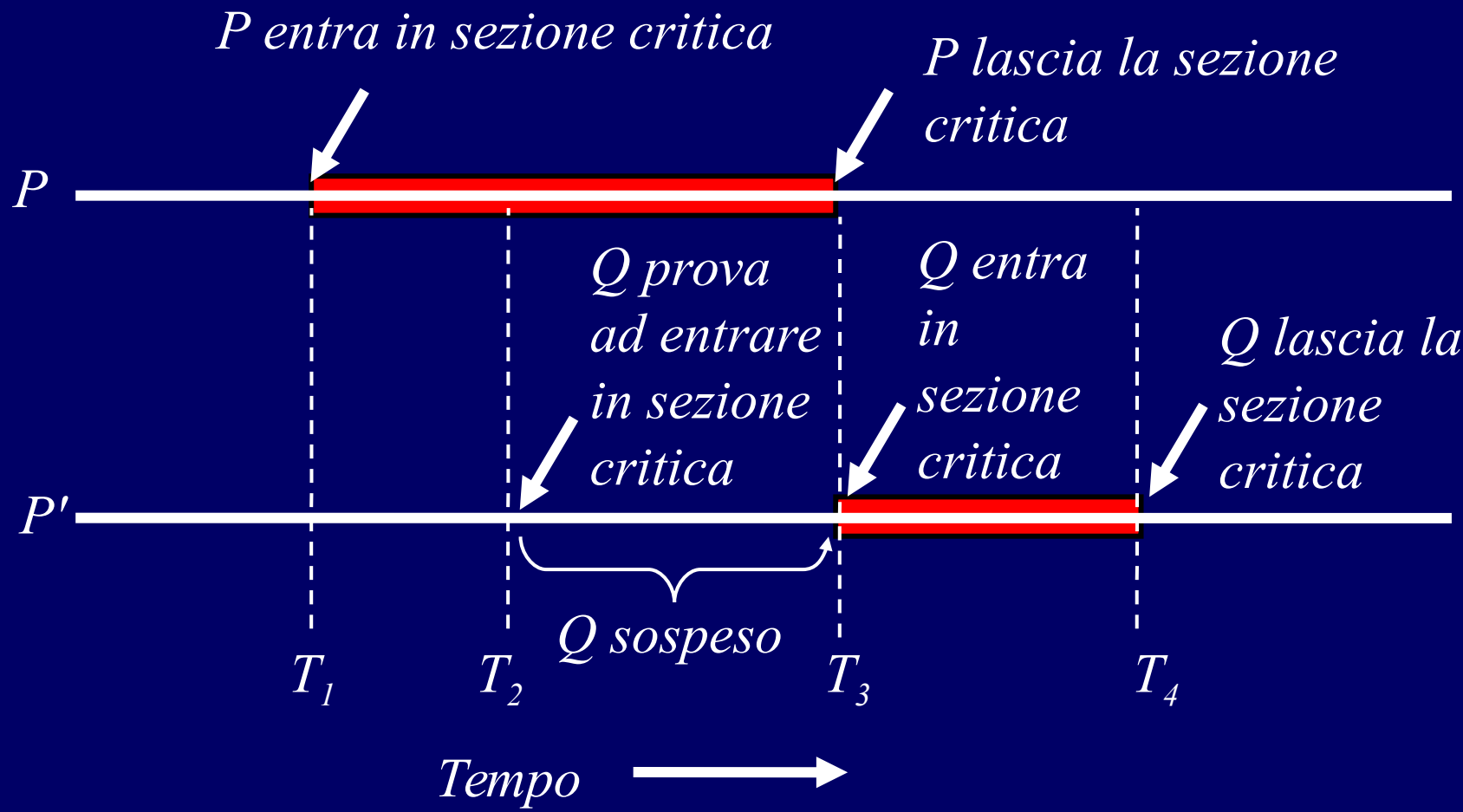
Sezioni Critiche

- Il frammento di programma che utilizza la risorsa si dice *sezione critica* o *regione critica* rispetto alla risorsa $\mathcal{R}$
- i flussi di esecuzione *in sezione critica* vogliono accedere alla risorsa con la garanzia di esserne l'unico utilizzatore
- In altre parole, si vogliono escludere quelle sequenze di esecuzione, o di interleaving, che possono violare l'accesso in mutua esclusione

Sezioni Critiche e Flussi di Esecuzione



Mutua Esclusione e Sezioni Critiche



Risoluzione del Problema di Mutua Esclusione

- Costruiremo una soluzione per raffinamenti successivi, cominciando dalla più semplice soluzione possibile
- Una variabile X associata alla risorsa $\mathcal{R}$
  - X=0: uno ed uno solo f.d.e. è nella sua sezione critica
  - X=1: nessun f.d.e. è nella sua sezione critica
- In realtà è sufficiente che X sia un bit di memoria, detto anche *semaforo di basso livello*
  - X=0 significa *rosso*
  - X=1 significa *verde*

Spin Lock (1)

- Il protocollo di utilizzo assume questa forma:
- *<richiesta>:*

```
LOCK(X):
begin
    loop begin
        if X=1 exit; // aspetta che sia verde
    end
    X ← 0; // poni il semaforo a rosso
end
```

Ciclo di attesa attiva

- *<rilascio>:*

```
UNLOCK(X):
begin
    X ← 1; // poni il semaforo a verde
end
```

Spin Lock (2)

Questa proposta presenta alcuni gravi problemi

- Attesa attiva
- Non garantisce l'accesso seriale:
  - *P* esegue la LOCK(X) e trova il semaforo verde, ma prima di riuscire a farlo diventare rosso perde l'utilizzo del processore fisico per sopraggiunto interleaving
  - allorché anche *Q* esegue la LOCK(X) e trova il semaforo ancora verde
  - Entrambi proseguono impostando il semaforo a rosso
  - I due f.d.e. *P* e *Q* sono entrambi in sezione critica
- Bisogna garantire l'indivisibilità delle due primitive

Spin Lock (3)

- *<richiesta>:* **LOCK(X): begin**
  loop begin
  <disabilita interruzioni>
  if X=1 exit; *// aspetta che sia verde*
  <abilita interruzioni>
  <ritardo>
  end
  X ← 0; *// poni il semaforo a rosso*
  <abilita interruzioni>
  end
- *<rilascio>:* **UNLOCK(X): begin**
  <disabilita interruzioni>
  X ← 1; *// poni il semaforo a verde*
  <abilita interruzioni>
  end

Osservazioni sugli Spin Lock (1)

- La soluzione presentata garantisce l'indivisibilità delle primitive di **LOCK** ed **UNLOCK**
- Tuttavia ha questi inconvenienti:
  - valida solo su sistemi uni-processor:
    - *P* esegue la LOCK(X) e trova il semaforo verde, ma prima di riuscire a farlo diventare rosso interviene un f.d.e. *Q* in *esecuzione su un altro processore*
    - anche *Q* esegue la LOCK(X) e trova il semaforo ancora verde
    - entrambi proseguono impostando il semaforo a rosso
    - i due f.d.e. *P* e *Q* sono entrambi in sezione critica
  - In pratica, alla sequenza di interleaving ipotizzata prima, si è sostituita una sequenza di esecuzione con effetti analoghi

Osservazioni sugli Spin Lock (2)

- Altri inconvenienti:
  - richiede continuamente di disabilitare le interruzioni compromettendo la reattività del sistema
- Questi inconvenienti si risolvono con un piccolo ma decisivo aiuto da parte dell'hardware:
  - l'unico punto in cui la LOCK(X) può essere interrotta è nell'intervallo di tempo che intercorre tra il test del valore di X e la sua assegnazione a 0
  - pertanto si prevede una istruzione macchina *TestAndSet* che, in maniera indivisibile, controlla il valore di un bit e lo azzera
  - analogamente si può supporre indivisibile l'istruzione macchina che implementa la UNLOCK(X)

Spin Lock con *TestAndSet*

- *<richiesta>:*

```
LOCK(X):  begin
        loop begin
            TestAndSet(X);
                if (X era 1) exit; // aspetta il verde
            <ritardo>
        end
        //qui il semaforo è già a rosso
    end
```

istruzione macchina indivisibile

- *<rilascio>:*

```
UNLOCK(X): begin
        Set(X); // poni il semaforo a verde
    end
```

istruzione macchina indivisibile

Spin Lock: Ancora Inconvenienti

- Attesa Attiva
  - il processore viene impegnato attivamente nel controllare continuamente il valore di un bit, per tutto il periodo in cui il f.d.e. attende che la risorsa si liberi
- Non è garantita la proprietà di fairness
  - non esiste garanzia esplicita contro l'evenienza che un f.d.e. attenda indefinitivamente un semaforo che pure diventa verde infinite volte
- Quest'ultimo punto esclude gli spin-lock come soluzione *fair* al problema della mutua esclusione
- N.B.: sebbene si tratta di una soluzione solo parziale, sarà utilizzata per costruirne di migliori

Fairness degli Spin-Lock

concurrent program FAIR_OR_UNFAIR?;
Var SX: *semaforo di basso livello*;
P_pid, Q_pid: *identificatori di f.d.e.;*

```
concurrent procedure P
begin      ...
    loop
       LOCK(SX);
         <sezione critica di P
          rispetto ad R>
       UNLOCK(SX);   ...
  end loop
end
```

```
concurrent procedure Q
begin      ...
    loop
       LOCK(SX);
         <sezione critica di Q
          rispetto ad R>
       UNLOCK(SX);   ...
  end loop
end
```

```
begin
   Set(SX);
   cobegin P || Q coend;
end
```

Esiste una seq. di interleaving unfair in cui un flusso (Q) non avanza mai? Quale?

Concreto vs Astratto

- Sinora ci siamo concentrati sugli aspetti realizzativi per evidenziare le difficoltà intrinseche del Problema di Mutua Esclusione
  - Semaforo di *basso* livello (o spin-lock)

- Al contrario, le primitive disponbili oggi sono state concepite partendo da uno strumento astratto inizialmente ideato astraendo dai dettagli realizzativi
  - Semaforo di *alto* livello (o di Dijkstra)

»

- Successivamente le due visioni complementari ci conduranno ad uno “schema di implementazione” completo

Come Risolvere il Problema della Mutua Esclusione

- Nel 1968 E.W. Dijkstra ha proposto due primitive che permettono la soluzione di qualsiasi problema di interazione fra f.d.e. astraendo dai dettagli realizzativi
- Le due primitive ricevono come parametro un intero non negativo S detto *semaforo*
  - N.B. Esistono definizioni alternative equivalenti

- Inizialmente discuteremo le due primitive nella versione *astratta*, ovvero indipendentemente da qualsiasi aspetto realizzativo

L’idea di Dijkstra

- I f.d.e. eseguono attese attive perché verificano autonomamente gli eventi sui quali sincronizzarsi
- Per un f.d.e. che attende un evento, esiste sempre un altro f.d.e. nella posizione “ideale” per indicargli attivamente quando si verifica
- Le attese attive possono essere trasformate in attese passive disciplinando
  - i f.d.e. *consumatori* di eventi ad esplicitare il proprio interesse ed attendere passivamente che si verifichino
  - i f.d.e. *produttori* di eventi ad informare attivamente i corrispondenti f.d.e. consumatori

Le Primitive **P** & **V** di Dijkstra

P(S):

if S=0

then <<*poni il f.d.e. in stato di attesa passiva di una* V(S) >>

else S ← S – 1;

V(S):

if <<*esiste un f.d.e. in attesa di una* V(S)>>

then <<*risveglia uno di questi f.d.e.*>>

else S ← S + 1;

Commenti alle Primitive **P** & **V** di Dijkstra

- Ad ogni semaforo S è associata una coda Q_S di f.d.e. in attesa di una V(S)
- Un f.d.e. che esegue una P(S):
  - può avanzare solo se trova S>0,
  - altrimenti deve accodarsi in Q_S per attendere passivamente una V(S)
- Un f.d.e. che esegue una V(S):
  - se non esistono f.d.e. in attesa dentro Q_S ha come unico effetto quello di incrementare S
  - altrimenti risveglia uno dei f.d.e. in Q_S
  - in ogni caso, il f.d.e. che ha eseguito la V(S) può continuare indisturbato

Requisiti delle Primitive **P** & **V**

- Le primitive **P** & **V** devono essere indivisibili
  - altrimenti sarebbe possibile una sequenza di interleaving come questa:
    1. un f.d.e. P_1 esegue una P(S) e trova S=0
    2. un f.d.e. P_2 incrementa S (quindi S>0)
    3. il f.d.e. P_1 viene posto in attesa di S>0
- La disciplina di estrazione/inserimento della coda Q_s deve garantire la proprietà di fairness
  - in genere si utilizza una disciplina FIFO

Una Soluzione al Problema della Mutua Esclusione

concurrent program PME;
var S_R: *semaforo*;

```
concurrent procedure P
begin      ...
       P(S_R);
         <sezione critica di P
          rispetto ad R>
       V(S_R);       ...
end
```

```
concurrent procedure Q
begin      ...
       P(S_R);
         <sezione critica di Q
          rispetto ad R>
       V(S_R);       ...
end
```

begin
  INIZ_SEM(S_R,1);
  cobegin P || Q **coend**;
end

Fairness dei Semafori di Dijkstra

concurrent program FAIR_OR_UNFAIR;
var S_R: *semaforo*;

```
concurrent procedure P
begin      ...
    loop
       P(S_R);
         <sezione critica di P
          rispetto ad R>
       V(S_R);        ...
    end loop
end
```

```
concurrent procedure Q
begin      ...
    loop
       P(S_R);
         <sezione critica di Q
          rispetto ad R>
       V(S_R);        ...
    end loop
end
```

```
begin
   INIZ_SEM(S_R,1);
   cobegin P || Q coend;
end
```

Esistono seq. di interleaving unfair in cui un f.d.e. (ad es. Q) non avanza mai?

Implementazione della Primitiva **P(S)**

P(S): begin

<*disabilita le interruzioni*>

LOCK(SX)

if S=0 **then** <<*sospendi il f.d.e. ed inserisci il suo descrittore nella coda* Q_s>>

else S ← S – 1;

UNLOCK(SX)

<*abilita le interruzioni*>

end

Implementazione della Primitiva **V(S)**

V(S): begin

<disabilita le interruzioni>

LOCK(SX)

if Q_s *non è vuota*

then *<<estrai un descrittore da* Q_s

e poni il corrispondente f.d.e.

in stato di pronto>>

else $S \leftarrow S + 1;$

UNLOCK(SX)

<abilita le interruzioni>

end

Implementazioni di **P** & **V**: Domande

- SX è uno spin lock utilizzato per garantire l'indivisibilità delle due primitive ed è necessario solamente nei sistemi multi-processore
  - ritorna il problema delle attese attive?

- Per i sistemi uni-processore basterebbe disabilitare le interruzioni
  - Eppure conviene disabilitarle anche per i sistemi multi-processore, perché?

Implementazioni di **P** & **V**: Risposte (1)

- *Nei sistemi multi-processore l'utilizzo degli spin lock comporta attese attive?*

  Si. Ma le risorse condivise a cui sono associati vengono rilasciate immediatamente; la risorsa condivisa in questo caso è la variabile intera S.

  Le attese attive, anche se presenti, sono normalmente tollerabili in quanto di brevissima durata.

Implementazioni di **P** & **V**: Risposte (2)

- *Perché conviene disabilitare le interruzioni anche nei sistemi multi-processore?*

  Perché altrimenti sarebbero possibili seq. di esecuzione ammissibili come questa:

  1. un f.d.e. *P* esegue una P(S) od una V(S) su un processore CPU_i e riesce a superare la LOCK(SX)
  2. prima che riesca a passare la corrispondente UNLOCK(SX), *P* viene interrotto da una interruzione allo stesso processore che attiva un altro f.d.e.
  3. a questo punto un f.d.e. *Q* esegue una P(S) o una V(S) su un processore CPU_j ($j \neq i$) e rimane quindi in attesa attiva sulla LOCK(SX) fino a quando il f.d.e. *P* non viene risvegliato ed esegue la UNLOCK(SX)

Implementazioni di **P** & **V**: Riepilogo

Sistemi	Abilitazione / Disabilitazione Interruzioni	LOCK(SX) / UNLOCK(SX)
Multi-Processore	conveniente ma non indispensabile	necessarie
Uni-Processore	necessarie	superflue

Commenti sulle Implementazioni di **P** & **V**

- Lo spin lock garantisce l'indivisibilità (ma non la *fairness*) delle primitive da f.d.e. che avanzano su altri processori
- La disabilitazione delle interruzioni garantisce l'indivisibilità delle primitive da f.d.e. che avanzano sullo stesso processore

Fairness dei Semafori di Dijkstra Implementati Utilizzando Spin-Lock

concurrent program FAIR_OR_UNFAIR;
var S_R: *semaforo (implementato usando spin-lock)*;

```
concurrent procedure P
begin      ...
    loop
       P(S_R);
         <sezione critica di P
          rispetto ad R>
       V(S_R);        ...
    end loop
end
```

```
concurrent procedure Q
begin      ...
    loop
       P(S_R);
         <sezione critica di Q
          rispetto ad R>
       V(S_R);        ...
    end loop
end
```

```
begin
   INIZ_SEM(S_R,1);
   cobegin P || Q coend;
end
```

Esistono seq. di interleaving unfair in cui un f.d.e. (ad es. Q) non avanza mai? Quali?

Espressività dei Semafori

- I semafori sono sufficienti a risolvere qualsiasi problema di sincronizzazione
- Infatti è possibile implementare la primitiva di **join** con i semafori:

Il f.d.e. P_j esegue una join sul f.d.e. P_i

- un semaforo binario S_{ij} associato a tale join
- S_{ij} inizializzato a 0
- $P(S_{ij})$ al posto della join P_i
- il f.d.e. P_i esegue una $V(S_{ij})$ prima di terminare

Semafori come Strumenti di Competizione e Cooperazione

- Due utilizzi tipici dei semafori
  - per sincronizzare due f.d.e. su un evento
  - per risolvere problemi di competizione nell'accesso ad un risorsa di molteplicità finita
- Ai due utilizzi corrispondono
  - semafori in *stile cooperativo*
    - assumono come valore 0 o 1
    - vengono inizializzati a 0
    - Il verificarsi dell'evento a cui sono associati viene segnalato con un semaforo che "passa a verde"
    - Il f.d.e. che effettua una V è diverso da quello che ha effettuato la P
  - semafori in *stile competitivo*
    - assumono come valore da 0 ad M, se M è la molteplicità della risorsa
    - vengono inizializzati al valore della molteplicità
    - il f.d.e. che effettua una V è lo stesso che ha effettuato la corrispondente P di apertura della sezione critica sulla risorsa

Esercizi

Esercizio: perché la primitiva **V** deve essere indivisibile?

Esercizio: un f.d.e **Q** deve eseguire ciclicamente una certa serie di operazioni ogni volta che un f.d.e. **P** gli dà il permesso eseguendo una **V** su un semaforo FAI inizialmente posto a 0:

```
concurrent procedure P        concurrent procedure Q
loop begin                    loop begin
        …                             P(FAI);
        V(FAI);                       …
        …                     end
end
```

Mostrare che se FAI è un semaforo binario i f.d.e. possono esibire degli errori dipendenti dal tempo

Esercizi

Esercizio: tradurre con le primitive **fork** e **P**, **V** ma senza **join** i diagrammi delle precedenze visti sinora.

Esercizio: le primitive **P** e **V** originariamente proposte da Dijkstra sono alternativamente così definite:

P(S): S ← S – 1;
if S<0 **then** *<<poni il f.d.e. in una coda di attesa Qs >>*

V(S): S ← S + 1;
if S<=0 **then** *<<risveglia, se esiste, uno dei f.d.e. di Qs>>*

Secondo questa definizione S può assumere anche valori negativi. Dimostrare che se S<0 allora –S rappresenta la lunghezza della coda Q_s. Stabilire, inoltre, se le P e V così definite siano o meno semanticamente equivalenti a quelle prima definite.

Esercizi

Esercizio: dimostrare che i semafori binari hanno la stessa espressività di quelli a conteggio.

Suggerimento: mostrare che è possibile implementare un semaforo a conteggio mediante due semafori binari, uno in stile competitivo ed uno in stile cooperativo.

Contesto di Riferimento

- In questo corso:
  - tratteremo di f.d.e. sequenziali debolmente connessi (a “grana grossa”)
  - gli scenari di riferimento sono quelli in cui le attese passive sono generalmente ritenute convenienti rispetto allo *spinning*
    - ✔ è considerato *di riferimento* lo *schema di implementazione* delle primitive di Dijkstra basato su spin-lock di basso livello ed attese passive dei f.d.e.
  - chiamiamo questo schema di implementazione *bloccante* nonostante utilizzi *brevi* attese attive per garantire l'indivisibilità delle primitive: notare che i f.d.e. possono incorrere in lunghe attese passive

Sincronizzazione *Blocking*

- Le tecniche di sincronizzazione di riferimento per questo corso sono *blocking,* ma riprenderemo le attese attive verso la fine del corso quando parleremo di sincronizzazione *non-blocking*

- Tuttavia, traendo spunto dalle primitiva ***TestAndSet()*** di cui si fa uso nello schema di implementazione delle primitive non-blocking:
  - si suggerisce uno schema di implementazione alternativa dei semafori dove al contrario i f.d.e. aspettano in attesa attiva
  - analizziamo meglio le caratteristiche dell'implementazione di riferimento

- Per trovare una motivazione, si può partire dall'osservazione che nello schema di implementazione bloccante, lo spinning all'interno delle primitive LOCK & UNLOCK viene tollerato proprio perché di brevissima durata

Sincronizzazione *Non-Blocking*

- Un nuovo schema di implementazione delle due primitive basato direttamente sugli spin-lock

- Implementazione di un semaforo binario *non-bloccante*
- Direttamente:
  - P(S) con LOCK(SX)
  - V(S) con UNLOCK(SX)

  può comportare una attesa attiva per tutta la durata della sezione critica di qualche altro f.d.e. ma risparmia il costo del context-switching

- Ha senso solo per i multi-processor!
  - Ma tanto ormai è la norma!
- E la *fairness!*?

Costo della Sincronizzazione (1)

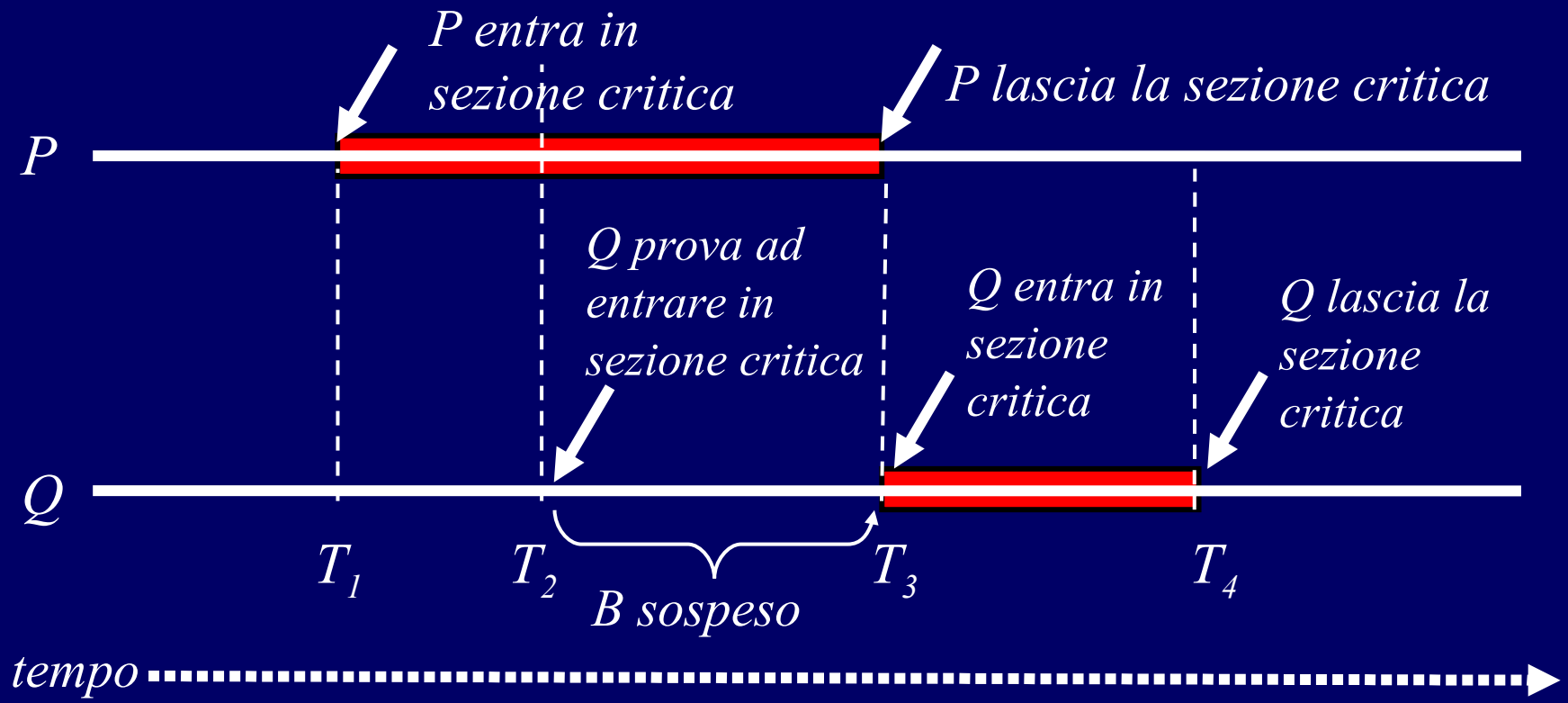
- *Overhead per singola esecuzione* di una sezione critica realizzata tramite primitive **P** e **V**

- Confrontiamo i due schemi di implementazione alternativi:
  - *Blocking*: attesa passiva e context-switch
  - *Non-blocking*: attesa attiva

    (nessun context-switch viene forzato)

- I principali costi risultano essere:
  - *Blocking*:
    - costo del context-switching
    - costi delle attese attive sullo spin-lock di basso livello [trascurabili]
  - *Non-blocking*:
    - costo delle attese attive (sullo spin-lock di *alto* livello)

Costo della Sincronizzazione (2)

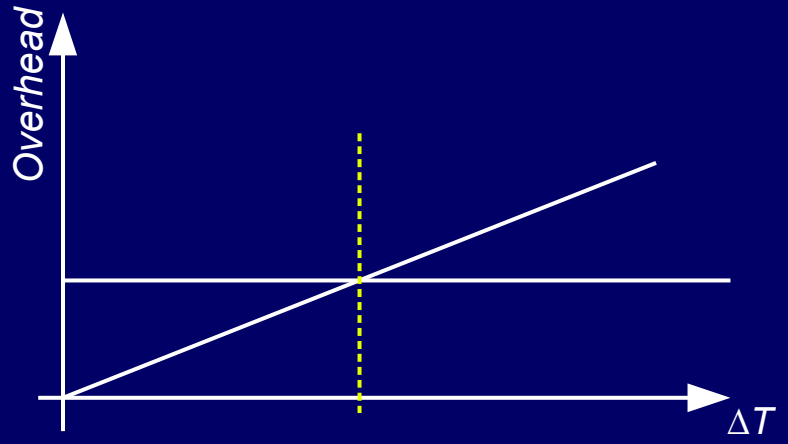


- *Non-Blocking*: spinning per l'intervallo di tempo T_3-T_2
- *Blocking*: 2 context-switch

(+spinning sul lock di basso livello trascurabile)

Costo della Sincronizzazione (3)

- Consideriamo l'andamento dei costi di sincronizzazione all'aumentare del livello di competizione e della durata delle sezioni critiche
  - aumenta per la sincronizzazione *non-blocking*
  - si può ipotizzare costante per la sincronizzazione *blocking*



- Per macchine multi-processor, deve esistere un certo *livello di competizione* al di sotto del quale, se le sezioni critiche sono di durata sufficientemente breve
- ✔ *può risultare meno oneroso una attesa attiva di breve durata che due context-switch*

Attese: *Attive* vs *Passive*

- L'implementazione *bloccante* può inquadrarsi come una gestione “pessimistica” della sincronizzazione
  - ✔ Più efficiente in presenza di forte competizione
- L'implementazione *non-bloccante* può inquadrarsi come una gestione “ottimistica” della interferenza
  - ✔ Più efficiente in presenza di scarsa competizione

Approccio Ibrido

- Alcune piattaforme multi-processor possono scegliere la migliore implementazione di un semaforo sulla base di una analisi statistica della frequenza e della durata delle sezioni critiche a tempo di esecuzione

- Spesso, più semplicemente:
  - prima, spinning di breve durata
  - quindi, context-switch ed attesa passiva

- Altre possibilità:
  - Backoff prima di ritentare lo spinning

Algoritmi *Non-Blocking* (1)

- Queste osservazioni sono spinte all'estremo da una classe di algoritmi chiamati *non-blocking*
  - ammettono solo attese attive
  - ammettono l'esistenza di stati inconsistenti (“parzialmente” costruiti)
  - utilizzano istruzioni atomiche elementari (tipo *TestAndSet*) per tutti gli aggiornamenti di singole “porzioni” dello stato
  - ciascun f.d.e. deve saper lavorare in presenza di interferenza!

Algoritmi *Non-Blocking* (2)

- Principali vantaggi:
  - efficienza e scalabilità
  - non soffrono di stallo
- Principali svantaggi:
  - molto difficili da progettare e testare
  - possono indurre a scrivere soluzioni unfair
- In effetti il loro utilizzo è motivato in contesti piuttosto particolari:
  - porzioni del nucleo di un sistema operativo
  - librerie di strutture dati thread-safe e scalabili

  Alla fine di questo corso saranno trattati tra gli argomenti avanzati