Sistemi Operativi per LT Informatica
A.A. 2017-2018

Sincronizzazione tra i processi

Docente: Salvatore Sorce

Sezione 3

3. Sezioni critiche

"Non-interferenza"

- **Problema**
  - Se le sequenze di istruzioni non vengono eseguite in modo atomico, come possiamo garantire la non-interferenza?
- **Idea generale**
  - Dobbiamo trovare il modo di specificare che certe parti dei programmi sono "speciali", ovvero devono essere eseguite in modo atomico (senza interruzioni)

Sezioni critiche

- **Definizione**
  - La parte di un programma che utilizza una o più risorse condivise viene detta *sezione critica (critical section, o CS)*
- **Esempio**

```
process P1
   a1 = rand();
   totale = totale + a1;
}
```

```
process P2 {
   a2 = rand();
   totale = totale + a2;
}
```

- **Spiegazione:**
  - La parte evidenziata è una sezione critica, in quanto accede alla risorsa condivisa **`totale`**; mentre **`a1`** e **`a2`** non sono condivise

Sezioni condivise

- **Obiettivi**
  - Vogliamo garantire che le sezioni critiche siano eseguite in modo mutualmente esclusivo (atomico)
  - Vogliamo evitare situazioni di blocco, sia dovute a deadlock sia dovute a starvation

Sezioni critiche

- **Sintassi:**
  - `[enter cs]` indica il punto di inizio di una sezione critica
  - `[exit cs]` indica il punto di fine di una sezione critica
- **Esempio:**

```
x:=0
cobegin
   [enter cs]; x = x+1; [exit cs];
//
   [enter cs]; x = x-1; [exit cs];
coend
```

Sezioni critiche

- **Esempio:**

```
cobegin
   val = rand();
   a = a + val;
   b = b + val
//
   val = rand();
   a = a * val;
   b = b * val;
coend
```

- **Perchè abbiamo bisogno di costrutti specifici?**
  - Perchè il S.O. non può capire da solo cosa è una sezione critica e cosa non lo è
- **In questo programma:**
  - Vorremmo garantire che **a** sia sempre uguale a **b** (*invariante*)
- **Soluzione 1:**
  - Lasciamo fare al sistema operativo...
  - Ma il S.O. non conosce il vincolo dell'invarianza
  - L'unica soluzione possibile per il S.O. è non eseguire le due parti in parallelo
  - Ma così perdiamo i vantaggi!

Sezioni critiche

- **Esempio:**

```
cobegin
   val = rand();
   [enter cs]
   a = a + val;
   b = b + val
   [exit cs]
//
   val = rand();
   [enter cs]
   a = a * val;
   b = b * val;
   [exit cs]
coend
```

- **In questo programma:**
  - Vorremmo garantire che **a** sia sempre uguale a **b** (*invariante*)
- **Soluzione 2:**
  - Indichiamo al S.O. cosa può essere eseguito in parallelo
  - Indichiamo al S.O. cosa deve essere eseguito in modo atomico, altrimenti non avremo consistenza

Sezioni critiche

- **Problema della CS**
  - Si tratta di realizzare **N** processi della forma

```
process Pi {   /* i=1...N */
  while (true) {
    [enter cs]
    critical section
    [exit cs]
    non-critical section
  }
}
```

in modo che valgano le seguenti proprietà:

Sezioni critiche

- **Requisiti per le CS**

  1) *Mutua esclusione*
     - Solo un processo alla volta deve essere all'interno della CS, fra tutti quelli che hanno una CS per la stessa risorsa condivisa

  2) *Assenza di deadlock*
     - Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile

  3) *Assenza di delay non necessari*
     - Un processo fuori dalla CS non deve ritardare l'ingresso della CS da parte di un altro processo

  4) *Eventual entry (assenza di starvation)*
     - Ogni processo che lo richiede, prima o poi entra nella CS

Sezioni critiche

- **Perché il problema delle CS è espresso in questa forma?**
  - Perché descrive in modo generale un insieme di processi, ognuno dei quali può ripetutamente entrare e uscire da una sezione critica
- **Dobbiamo fare un'assunzione:**
  - Se un processo entra in una critical section, prima o poi ne uscirà
  - Ovvero, un processo può terminare solo fuori dalla sua sezione critica

Sezioni critiche - Possibili approcci

- **Approcci software**
  - la responsabilità cade sui processi che vogliono accedere ad un oggetto distribuito
  - problemi
    - soggetto ad errori!
    - vedremo che è costoso in termini di esecuzione (busy waiting)
  - interessante dal punto di vista didattico
- **Approcci hardware**
  - Uso di istruzioni speciali del linguaggio macchina, progettate apposta
  - efficienti
  - problemi
    - non sono adatte come soluzioni general-purpose

Sezioni critiche - Possibili approcci

- **Approcci basati su supporto nel S.O. o nel linguaggio**
  - la responsabilità di garantire la mutua esclusione ricade sul S.O. o sul linguaggio (e.g. Java)
- **Esempi**
  - Semafori
  - Monitor
  - Message passing

Algoritmo di Dekker

- **Dijkstra (1965)**
  - Riporta un algoritmo per la mutua esclusione
  - Progettato dal matematico olandese *Dekker*
  - Nell'articolo, la soluzione viene sviluppata in fasi
  - Seguiremo anche noi questo approccio

Tentativo 1

```
shared int turn = P; cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    while (turn == Q)
      ; /* do nothing */
    critical section
    turn = Q;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    while (turn == P)
      ; /* do nothing */
    critical section
    turn = P;
    non-critical section
  }
}
```

- **Note**
  - la variabile **turn** è condivisa
  - può essere acceduta solo da un processo alla volta (in lettura o scrittura)
  - il controllo iterativo su una condizione di accesso viene detto *busy waiting*

Tentativo 1

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 3: assenza di delay non necessari
    - "Un processo fuori dalla CS non deve ritardare l'ingresso nella CS da parte di un altro processo"

Tentativo 1 - Problema

- **Si consideri questa esecuzione:**
  - **P** entra nella sezione critica
  - **P** esce dalla sezione critica
  - **P** cerca di entrare nella sezione critica
  - **Q** è molto lento; fino a quando **Q** non entra/esce dalla CS, **P** non può entrare

Tentativo 2

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {                                process Q {
  while (true) {                             while (true) {
    /* entry protocol */                       /* entry protocol */
    while (inq)                                while (inp)
      ; /* do nothing */                          ; /* do nothing */
    inp = true;                                inq = true;
    critical section                           critical section
    inp = false;                               inq = false;
    non-critical section                       non-critical section
  }                                          }
}                                          }
```

- **Note**
  - ogni processo è associato ad un flag
  - ogni processo può esaminare il flag dell'altro, ma non può modificarlo

Tentativo 2

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 1: *mutua esclusione*
    - " solo un processo alla volta deve essere all'interno della CS "

Tentativo 2 - Problema

- **Si consideri questa esecuzione:**
  - **P** attende fino a quando **`inq=false`**; vero dall'inizio, passa
  - **Q** attende fino a quando **`inp=false`**; vero dall'inizio, passa
  - **P** **`inp = true`**;
  - **P** entra nella critical section
  - **Q** **`inq = true`**;
  - **Q** entra nella critical section

Tentativo 3

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    inp = true;
    while (inq)
      ; /* do nothing */
    critical section
    inp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    inq = true;
    while (inp)
      ; /* do nothing */
    critical section
    inq = false;
    non-critical section
  }
}
```

- **Note**
  - Nel tentativo precedente, il problema stava nel fatto che era possibile che un context switch occorresse tra il controllo sul flag dell'altro processo e la modifica del proprio. Abbiamo trovato una soluzione?

Tentativo 3

- **La soluzione proposta è corretta?**
- **Problema:**
  - Non rispetta il requisito 2: *assenza di deadlock*
    - "Uno scenario in cui tutti i processi restano bloccati definitivamente non è ammissibile”

Tentativo 3 - Problema

- **Si consideri questa esecuzione:**
  - **P** `inp = true;`
  - **Q** `inq = true;`
  - **P** attende fino a quando `inq=false`; bloccato
  - **Q** attende fino a quando `inq=false`; bloccato

Tentativo 4

```
shared boolean inp = false; shared boolean inq = false;
cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    inp = true;
    while (inq) {
      inp = false;
      /* delay */
      inp = true;
    }
    critical section
    inp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    inq = true;
    while (inp) {
      inq = false;
      /* delay */
      inq = true;
    }
    critical section
    inq = false;
    non-critical section
  }
}
```

Tentativo 4

- **Che sia la volta buona?**
- **Problema 1**
  - Non rispetta il requisito 4: *eventual entry*
    - " ogni processo che lo richiede, prima o poi entra nella CS "

Tentativo 4 - Problema

- **Si consideri questa esecuzione:**
  - **P** `inp = true;`
  - **Q** `inq = true;`
  - **P** verifica `inq`
  - **Q** verifica `inp`
  - **P** `inp = false;`
  - **Q** `inq = false;`
- **Note**
  - questa situazione viene detta "*livelock*", o situazione di "*mutua cortesia*"
  - difficilmente viene sostenuta a lungo, però è da evitare...
  - ... anche per l'uso dell'attesa come meccanismo di sincronizzazione

Riassumendo - una galleria di {e|o}rrori

- **Tentativo 1**
  - L'uso dei turni permette di evitare problemi di deadlock e mutua esclusione, ma non va bene in generale
- **Tentativo 2**
  - "verifica di una variabile + aggiornamento di un altra" non sono operazioni eseguite in modo atomico
- **Tentativo 3**
  - il deadlock è causato dal fatto che entrambi i processi insistono nella loro richiesta di entrare nella CS - *in modo simmetrico*
- **Tentativo 4**
  - il livelock è causato dal fatto che entrambi i processi lasciano il passo all'altro processo - *in modo simmetrico*

Riassumendo

- **Quali caratteristiche per una soluzione?**
  - il meccanismo dei turni del tentativo 1 è ideale per "rompere la simmetria" dei tentativi 3 e 4
  - il meccanismo di "prendere l'iniziativa" del tentativo 3 è ideale per superare la stretta alternanza dei turni del tentativo 1
  - il meccanismo di "lasciare il passo" del tentativo 4 è ideale per evitare situazioni di deadlock del tentativo 2

Algoritmo di Dekker

```
shared int turn = P;
shared boolean needp = false; shared boolean needq = false;
cobegin P // Q coend

process P {
  while (true) {
    /* entry protocol */
    needp = true;
    while (needq)
      if (turn == Q) {
        needp = false;
        while (turn == Q)
          ; /* do nothing */
        needp = true;
      }
    critical section
    needp = false; turn = Q;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    needq = true;
    while (needp)
      if (turn == P) {
        needq = false;
        while (turn == P)
          ; /* do nothing */
        needq = true;
      }
    critical section
    needq = false; turn = P;
    non-critical section
  }
}
```

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (mutua esclusione)**
  - per assurdo:
    - supponiamo che **P** e **Q** siano in CS contemporaneamente
  - poiché:
    - gli accessi in memoria sono esclusivi
    - per entrare devono almeno aggiornare / valutare entrambe le variabili **`needp`** e **`needq`**
  - → uno dei due entra per primo; diciamo sia **Q**
  - → **`needq`** sarà **`true`** fino a quando **Q** non uscirà dal ciclo
  - → poiché **P** entra nella CS mentre **Q** è nella CS, significa che esiste un istante temporale in cui **`needq = false`** e **Q** è in CS
  - → ASSURDO!

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (assenza di deadlock)**
  - per assurdo
    - supponiamo che né **P** ne **Q** possano entrare in CS
  - → **P** e **Q** devono essere bloccati nel primo **`while`**
  - → esiste un istante **t** dopo di che **`needp`** e **`needq`** sono sempre **`true`**
  - supponiamo (senza perdita di gen.) che all'istante **t**, **`turn`** = **Q**
  - l'unica modifica a **`turn`** può avvenire solo quando **Q** entra in CS
  - → dopo **t**, **`turn`** resterà sempre uguale a **Q**
  - **P** entra nel primo ciclo, e mette **`needp`** = **`false`**
  - ASSURDO!

Algoritmo di Dekker - Dimostrazione

- **Dimostrazione (assenza di ritardi non necessari)**
  - se **Q** sta eseguendo codice non critico, allora **`needq = false`**
  - allora **P** può entrare nella CS
- **Dimostrazione (assenza di starvation)**
  - se **Q** richiede di accedere alla CS
    - **`needq = true`**
  - se **P** sta eseguendo codice non critico:
    - **Q** entra
  - se **P** sta eseguendo il resto del codice (CS, entrata, uscita)
    - prima o poi ne uscirà e metterà il turno a **Q**
    - **Q** potrà quindi entrare

Algoritmo di Peterson

- **Peterson (1981)**
  - più semplice e lineare di quello di Dijkstra / Dekker
  - più facilmente generalizzabile al caso di processi multipli

Algoritmo di Peterson

```
shared boolean needp = false;
shared boolean needq = false;
shared int turn;

cobegin P // Q coend
process P {
  while (true) {
    /* entry protocol */
    needp = true;
    turn = Q;
    while (needq && turn != P)
      ; /* do nothing */
    critical section
    needp = false;
    non-critical section
  }
}
```

```
process Q {
  while (true) {
    /* entry protocol */
    needq = true;
    turn = P;
    while (needp && turn != Q)
      ; /* do nothing */
    critical section
    needq = false;
    non-critical section
  }
}
```

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (mutua esclusione)**
  - supponiamo che P sia entrato nella sezione critica
  - vogliamo provare che Q non può entrare
  - sappiamo che needP == true
  - Q entra solo se turn = Q quando esegue il while
  - si consideri lo stato al momento in cui P entra nella critical section
    - due possibilità: needq == false or turn == P
    - se needq == false, Q doveva ancora eseguire needq == true, e quindi lo eseguirà dopo l'ingresso di P e porrà turn=P, precludendosi la possibilità di entrare
    - se turn==P, come sopra;

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (assenza di deadlock)**
  - supponiamo che per assurdo che **P** voglia entrare nella CS e sia bloccato nel suo ciclo **`while`**
  - questo significa che:
    - **`needp`** = **`true`**, **`needq`** = **`true`**, **`turn`** = **`Q`** per sempre
  - possono darsi tre casi:
    - **Q** non vuole entrare in CS
      - impossibile, visto che **`needq`** = **`true`**
    - **Q** è bloccato nel suo ciclo while
      - impossibile, visto che **`turn`** = **`Q`**
    - **Q** è nella sua CS e ne esce (prima o poi)
      - impossibile, visto che prima o poi **`needq`** assumerebbe il valore **`false`**

Algoritmo di Peterson - Dimostrazione

- **Dimostrazione (assenza di ritardi non necessari)**
  - se **Q** sta eseguendo codice non critico, allora `needq = false`
  - allora **P** può entrare nella CS
- **Dimostrazione (assenza di starvation)**
  - simile alla dimostrazione di assenza di deadlock
  - aggiungiamo un caso in fondo:
    - **Q** continua ad entrare ed uscire dalla sua CS, prevenendo l'ingresso di **P**
    - impossibile poiché
      - quando **Q** prova ad entrare nella CS pone `turn = P`
      - poiché `needp = true`
      - quindi **Q** deve attendere che **P** entri nella CS

Riassumendo...

- **Le soluzioni software**
  - permettono di risolvere il problema delle critical section
- **Problemi**
  - sono tutte basate su busy waiting
  - busy waiting spreca il tempo del processore
  - è una tecnica che non dovrebbe essere utilizzata!

Soluzioni Hardware

- **E se modificassimo l'hardware?**
  - le soluzioni di Dekker e Peterson prevedono come uniche istruzioni atomiche le operazioni di Load e Store
  - si può pensare di fornire alcune istruzioni hardware speciali per semplificare la realizzazione di sezioni critiche

Disabilitazione degli interrupt

- **Idea**
  - nei sistemi uniprocessore, i processi concorrenti vengono "alternati" tramite il meccanismo degli interrupt
  - allora facciamone a meno!
- **Esempio:**

```
process P {
  while (true) {
    disable interrupt
    critical section
    enable interrupt
    non-critical section
  }
}
```

Disabilitazione degli interrupt

- **Problemi**
  - il S.O. deve lasciare ai processi la responsabilità di riattivare gli interrupt
    - altamente pericoloso!
  - riduce il grado di parallelismo ottenibile dal processore
- **Inoltre:**
  - non funziona su sistemi multiprocessore

Test & Set

- **Istruzioni speciali**
  - istruzioni che realizzano due azioni in modo atomico
  - esempi
    - lettura e scrittura
    - test e scrittura
- **Test & Set**
  - `TS(x,y) := < y = x ; x = 1 >`
  - spiegazione
    - ritorna in `y` il valore precedente di `x`
    - assegna 1 ad `x`

Test & Set

```
shared lock=0; cobegin P // Q coend
process P {
  int vp;
  while (true) {
    do {
      TS(lock, vp);
    } while (vp);
    critical section
    lock=0;
    non-critical section
  }
}
process Q {
  int vp;
  while (true) {
    do {
      TS(lock, vp);
    } while (vp);
    critical section
    lock=0;
    non-critical section
  }
}
```

- **Mutua esclusione**
  - entra solo chi riesce a settare per primo il lock
- **No deadlock**
  - il primo che esegue TS entra senza problemi
- **No unnecessary delay**
  - un processo fuori dalla CS non blocca gli altri
- **No starvation**
  - No, se non assumiamo qualcosa di più

Altre istruzioni possibili

- **test&set non è l'unica istruzione speciale**
- **altri esempi:**
  - fetch&set
  - compare&swap
  - etc.

Riassumendo...

- **Vantaggi delle istruzioni speciali hardware**
  - sono applicabili a qualsiasi numero di processi, sia su sistemi monoprocessore che in sistemi multiprocessori
  - semplice e facile da verificare
  - può essere utilizzato per supportare sezioni critiche multiple; ogni sezione critica può essere definita dalla propria variabile
- **Svantaggi**
  - si utilizza ancora busy-waiting
  - i problemi di starvation non sono eliminati
  - sono comunque complesse da programmare

Riassumendo...

- **Vorremmo dei paradigmi**
  - che siano implementabili facilmente
  - consentano di scrivere programmi concorrenti in modo non troppo complesso