PCAD
Programmazione Concorrente e Algoritmi Distribuiti

Laurea Triennale Informatica, a.a. 2021/22

Programmazione Concorrente

Ragioniamoci sopra!

Esempi di notazioni per scrivere istruzioni concorrenti

- Notazione implicita con COBEGIN/END:

$$COBEGIN\ Prog_1 \parallel Prog_2 \parallel \ldots \parallel Prog_n\ COEND$$

- Ogni sottoprogramma $Prog_1, \ldots, Prog_n$ viene eseguito in concorrenza con gli altri e rappresenta quindi un thread separato di esecuzione
- Le istruzioni che seguono il *COEND* verrano eseguite solo quando tutti i thread hanno terminato la loro parte di esecuzione

Esercizio

- Supponiamo di voler definire un programma con COBEGIN e COEND per implementare una parte della funzione Mergesort
- Abbiamo a disposizione le funzioni
  - sort(v,i,j): ordina gli elementi dell'array v dall'indice i all'indice j
  - merge(v,i,j,k): fa il merge dei due segmenti (che supponiamo già ordinati) di v che vanno rispettivamente da i a j e da j+1 a k

Schema parallelo

```
mergesort(v,l)=
{
 m= l/2;
 COBEGIN
      sort(v,1,m);       ||
      sort(v,m+1,l);      ||
      merge(v,1,m,l);
 COEND;
}
```

Schema parallelo: Non va bene!

```
mergesort(v,l)=
{
  m= l/2;
  COBEGIN
      sort(v,1,m);     ||
      sort(v,m+1,l);   ||
      merge(v,1,m,l);
  COEND;
}
```

Le tre procedure operano in parallelo sullo stesso array e quindi le chiamate a sort e merge non sono indipendenti tra loro

Soluzione corretta

```
mergesort(v,l)=
{
  m= l/2;
  COBEGIN
      sort(v,1,m);   ||   sort(v,m+1,l);
  COEND;
  merge(v,1,m,l);
}
```

Notazione esplicita con definizioni di Thread

Dichiarazioni di variabili condivise:

```
var X1=v1, X2=v2, ...
```

Definizioni di thread:

```
Thread T_1 { Prog_1 }
....
Thread T_n { Prog_n}
```

Definizione di Thread

- Il frammento di codice `Prog_i` è un programma sequenziale e può quindi contenere una sequenza con assegnamenti, if-then-else, repeat, while, ecc.
- Ogni thread viene eseguito in concorrenza con gli altri condividendo le variabili globali
- L'esecuzione in parallelo non implica l'esecuzione ripetuta dei sottoprogrammi:

  Es. se `Prog_i` non ha cicli allora verrà eseguito una sola volta in parallelo con gli altri programmi

Esempio 1

```
Risorse
var x=0

Thread P1 { x:=500; }
Thread P2 { x:=0;   }
Thread P3 { write(x); }
```

Soluzione esempio 1

- Nell'esempio 2 vi sono due possibili output (valori scritti sul monitor dall'istruzione write): 0 o 500.
- L'esecuzione di tale programma da origine ad un solo valore
- Esecuzioni diverse possono dare origine a risultati diversi
- Un programma concorrente definisce una funzione che dato un'input produce un'insieme di possibili output e non un singolo valore come nel caso sequenziale (non determinismo)

Esempio 2

```
var x=0

Thread P1 {
  while (true)    x:=500;    endwhile;
}

Thread P2 {
  while (true)    x:=0;       endwhile;
}

Thread P3 {
  while (true)  write(x); endwhile;
}
```

Soluzione esempio 2

- Nell'esempio 2 vi sono un numero infinito di possibili output
- Ognuno di essi è a sua volta una sequenza finita o infinita $v_1 v_2 v_3 \ldots$ con $v_i \in \{0, 500\}$ per $i \geq 0$

  potrei eseguire solo P1 e P2 e mai P3, eseguire una volta P1 e poi una volta P3, ecc
- Ad esempio un possibile output è 0 500 0 500 0 500 ...
- L'esecuzione di tale programma da origine ad una sola sequenza di output (una tra tutte quelle possibili)
- Esecuzioni diverse possono dare origine a risultati diversi

Esempio 3

```
Risorse
var x=100

Thread P1 {
  x:=x+1;
}

Thread P2 {
  x:=x-1;
}
```

Soluzione esempio 3

- Se le istruzioni sono eseguite in sequenza alla fine $x = 100$
- Se eseguo P1 fino alla valutazione di $x + 1$ e prima dell'assegnamento, poi eseguo P2 completamente ed infine eseguo l'assegnamento in P1 posso ottenere $x = 101$
- Se eseguo P2 fino alla valutazione di $x - 1$ e prima dell'assegnamento, poi eseguo P1 completamente ed infine eseguo l'assegnamento in P2 posso ottenere $x = 99$

Sincronizzazione

Esempio: Problema del produttore-consumatore

Tipico paradigma dei thread cooperanti

- Il thread produttore produce informazione che viene consumata da un thread consumatore
- Soluzione a memoria condivisa: tra i due thread si pone un buffer di comunicazione di dimensione fissata

Produttore-consumatore con buffer limitato

Dati condivisi tra i thread

type *item* = ... ;
var *buffer*: **array** [0..*n*-1] **of** *item*;
in, *out*: 0..*n*-1;
counter: 0..*n*;
in, *out*, *counter* := 0;

Thread Produttore

while (true) do

...

produce un item in *nextp*

...

while *counter* $=$ *n* **do skip endwhile**;

buffer[*in*] $:=$ *nextp*;

in $:=$ *in* $+ 1$ **mod** *n*;

counter $:=$ *counter* $+ 1$;

endwhile

Thread Consumatore

while (true) do

while $counter = 0$ **do skip endwhile**;

$nextc := buffer[out]$;

$out := out + 1$ **mod** n;

$counter := counter - 1$;

...

consuma l'item in $nextc$

...

end

Istruzioni Atomiche

- Le istruzioni

  $counter := counter + 1;$

  $counter := counter - 1;$
- devono essere eseguite *atomicamente*
- Se eseguite non atomicamente possono portare ad inconsistenze.

Race Condition

Race condition: più thread accedono concorrentemente agli stessi dati, e il risultato dipende dall'ordine di esecuzione di istruzioni selezionate dai diversi thread

- Le race condition si possono verificare frequentemente nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Sono estremamente pericolose: portano al malfunzionamento dei thread cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema
- Sono difficili da individuare e riprodurre: dipendono da decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, ecc

Problema della Sezione Critica: PSC

Definizione del Problema della Sezione Critica (PSC) I

- Consideriamo un numero n di thread che competono tra loro per usare dati condivisi
- Ogni thread ha la stessa struttura

while (TRUE) **do**
entry section
sezione critica
exit section
sezione non critica
endwhile;

- Nel segmento di codice detto *sezione critica* si accede ai dati condivisi.

Definizione del Problema della Sezione Critica (PSC) II

- Problema: assicurare che quando un thread esegue la sua sezione critica, nessun altro thread possa entrare nella propria sezione critica - mutua esclusione dalla sezione critica -
- Bisogna proteggere la sezione critica con apposito codice di controllo da inserire nelle sezioni *entry section* e *exit section*

Criteri per una soluzione corretta di PSC

Una buona soluzione deve soddisfare le seguenti proprietà:

1. Mutua esclusione
2. Progresso
3. Attesa limitata

Mutua esclusione

Se il thread P_i sta eseguendo la sua sezione critica,
allora nessun altro thread può eseguire la propria sezione critica.

Progresso

Se nessun thread è nella sezione critica ed esiste un thread che desidera entrare nella propria sezione critica, allora l'esecuzione di tale thread non può essere posposta indefinitamente.

Attesa limitata

Se un thread P ha richiesto di entrare nella propria sezione critica, allora il numero di volte che si concede agli altri thread di accedere alla propria sezione critica prima del thread P deve essere limitato superiormente da una costante fissata a priori.

Assunzioni

- Dobbiamo dichiarare a priori quali istruzioni sono atomiche e quali no
- Ogni thread deve rimanere nella sezione critica solo un tempo finito (altrimenti il problema perde di senso)
- Ogni thread viene eseguito ad una velocità non nulla (quindi un thread non si può fermare da solo)
- Non ci sono vincoli sulla velocità relativa dei thread (numero e tipo di CPU)

Progresso: Definizioni alternative

- Progresso: varie definizioni
  1. Se nessun thread e' nella sezione critica ed esiste un thread che desidera entrare nella propria sezione critica, allora l'esecuzione di tale thread non può essere posposta indefinitamente.
  2. Se nessun thread è in esecuzione nella sua sezione critica e qualche thread desidera entrare nella propria sezione critica, solo i thread che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del thread che può entrare per primo nella propria sezione critica; questa scelta non può essere rimandata indefinitamente
  3. Un thread al di fuori della sua sezione critica non può prevenire altri thread dall'entrare la propria; i thread che cercano simultaneamente di accedere alla sezione critica devono decidere quale thread entra.
- Nota: una situazione di deadlock (tutti i thread sono bloccati) rappresenta una possibile violazione della proprietà di progresso

Bounded waiting

- Bounded waiting
  1. Se un thread ha già richiesto l'ingresso nella sua sezione critica, esiste un limite superiore al numero di volte che si consente ad altri thread di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta al primo thread
- Nota: *starvation* (un thread non entra mai) rappresenta una violazione di bounded waiting

Algoritmi Concorrenti per PSC

Possibile soluzioni software

- Supponiamo che ci siano solo 2 thread, P_0 e P_1
- Struttura del thread P_i (l'altro sia P_j)

while (TRUE) **do**
entry section
sezione critica
exit section
sezione non critica
endwhile

- Supponiamo che i thread possano condividere alcune variabili (dette *di lock*) per sincronizzare le proprie azioni

Flag condiviso

- Variabili condivise
  - **var** *occupato*: (0..1);
    inizialmente $occupato = 0$
  - $occupato = 0 \Rightarrow$ un thread può entrare nella propria sezione critica
- Thread P_i

while (TRUE) **do**
    while ($occupato \neq 0$) **do skip endwhile**;
    $occupato := 1$;
        sezione critica
    $occupato := 0$;
        sezione non critica
endwhile

Flag condiviso

- Problema: Non funziona
  Lo scheduler può interrompere il thread subito dopo il ciclo interno e prima dell'istruzione occupato:=1.
- P1 passa il test occupato=0 (esce dal loop)
  S.O. interrupt
  P2 passa il test occupato=0 (esce dal loop)
  P2 occupato:=1
  P2 sezione critica
  S.O. interrupt
  P1 occupato:=1
  P1 sezione critica

Alternanza Stretta

- **var** *turn*: (0..1);
  inizialmente $turn = 0$
- $turn = i \Rightarrow P_i$ può entrare nella propria sezione critica
- Thread P_i

while (TRUE) **do**

while $(turn \neq i)$ **do skip endwhile**;

sezione critica

$turn := j$;

sezione non critica

endwhile

Alternanza Stretta

- Il precedente algoritmo soddisfa il requisito di mutua esclusione, ma non di progresso per thread con differenze di velocità
- P1 entra nella sezione critica
  P1 esce dalla sezione critica
  P1 cerca di entrare nella sezione critica
  P2 e' molto lento; fino a quando P2 non entra e poi esce dalla CS, P1 non puo' entrare

Busy Wait

- Negli algoritmi visti fino ad ora usiamo la nozione di attesa attiva (busy wait) di un evento da parte di un thread
  Es: aspettiamo che una variabile assuma il valore zero
  - Semplice da implementare
  - Porta a consumi inaccettabili di CPU
  - In genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)
- Un thread che attende attivamente su una variabile esegue uno *spin lock*.

Bounded waiting vs Progresso

Utilizzando alcune idee viste nei precedenti tentativi cerchiamo di dimostrare che le proprieta di progresso e bounded waiting non sono equivalenti

Controesempio 1

- Entry nella sez. critica controllata con una variabile condivisa turn: i thread entrano a turno stretto
  ($P_1, \ldots P_k, P_1, \ldots, P_k, P_1, \ldots, P_k, \ldots$).
  Cioè quando P_i esce setta $turn = i + 1$ (modulo k)
- Se P_i e' nella sezione NON critica ed esegue un loop infinito allora non vale progresso (tutti gli altri thread sono bloccati per sempre) .
- Tuttavia vale bounded waiting. Infatti i thread aspettano al piu' $k - 1$ turni.

Bounded waiting vs Progresso

Controesempio 2

- Scegliamo in modo casuale il thread che entra nella sezione critica dall'insieme dei thread in attesa di entrare.
- Vale progresso, Infatti nessun thread al di fuori della sezione critica puo' influenzare la scelta di quale thread puo' entrare.
- Non vale bounded waiting (nel caso peggiore P_i non entra mai: starvation).

Algoritmo di Peterson per 2 thread

- Algoritmo per 2 thread basato su una combinazione di *richiesta* e *accesso*
- Soddisfa tutti i requisiti e risolve il problema della sezione critica
- Si può generalizzare a N thread
- È ancora basato su spinlock

Algoritmo di Peterson

Risorse condivise
var turn:**int**;
var interested$_0$,interested$_1$: **bool**;

Process P_0:

while (TRUE) **do**
interested$_0$:=true;
turn:=0;
while (turn == 0 and interested$_1$) **do skip endwhile**;
critical section
interested$_0$:=false;
endwhile

Algoritmo di Peterson: Secondo thread

Ricordiamo che le risorse condivise sono:
var turn:**int**;
var interested$_0$,interested$_1$: **bool**;

Process P_1:

while (TRUE) **do**
interested$_1$:=true;
turn:=1;
while (turn == 1 and interested$_0$) **do skip endwhile**;
critical section
interested$_1$:=false;
endwhile

Algoritmo di Lamport per N thread

Algoritmo basato su una coda generata da biglietti di ordine crescente distribuiti ai clienti Risolve la sezione critica per n thread, generalizzando l'idea vista precedentemente.

- Prima di entrare nella sezione critica, ogni thread riceve un numero. Chi ha il numero più basso entra nella sezione critica.
- Se i thread P_i and P_j ricevono lo stesso numero: se $i < j$, allora P_i è servito per primo; altrimenti P_j è servito per primo.
- Lo schema di numerazione genera numeri in ordine crescente

Algoritmo per N thread: Possibile Schema

var number: array[1,. . . ,N] of integer;

Process $P(i)$:
while (TRUE) **do**
    $number[i] = max\{number[1], \ldots, number[N]\} + 1$;
    for $k : 1$ *to* N **do**
        while $(number[k] \neq 0$ **and** $(number[k] < number[i])$
            do skip
        endwhile;
    endfor;
    critical section
    $number[i] = 0$;
endwhile

Algoritmo di Lamport: Possibile Schema

- Il precedente algoritmo non va bene a causa di race condition
  - I thread possono prendere gli stessi biglietti
  - Servirebbe proteggere lettura e scrittura delle variabili *number*
- Come fare?
- Dobbiamo isolare degli stati attraverso i quali passano i thread

Algoritmo per N thread: Possibile Schema

var number: array[1,. . . ,N] of integer;

Process $P(i)$:
while (TRUE) **do**
  $number[i] = max\{number[1], \ldots, number[N]\} + 1$;
  for $k : 1$ *to* N **do**
    while $(number[k] \neq 0$ **and** $((number[k], k) \ll (number[i], i))$
      do skip
    endwhile;
  endfor;
  critical section
  $number[i] = 0$;
endwhile

dove $\langle a, b\rangle \ll \langle c, d\rangle$ sse $a < c$ oppure $a = c$ e $b < d$

- L'ordinamento $\ll$ serve a disambiguare i casi in cui i thread prendono lo stesso biglietto (ancora possibile) usando l'ordine degli identificatori (o priorita' o sim)
- Es. $(3, 1) \ll (3, 4)$
- Grazie a questo ordinamento formiamo una coda e quindi valgono le seguenti proprieta':
  - Mutua Esclusione
  - Progresso (un thread non interessato azzera il suo ticket)
  - Attesa limitata (i thread entrano nell'ordine della coda)

Algoritmo di Lamport per N thread

var choosing: array[1,. . . ,N] of boolean;
var number: array[1,. . . ,N] of integer;

Process $P(i)$:
while (TRUE) **do**
  $choosing[i] = TRUE$;
  $number[i] = max\{number[1], \ldots, number[N]\} + 1$;
  $choosing[i] = FALSE$;
  for $k : 1$ *to* N **do**
    while $choosing[k]$ **do skip endwhile**;
    while ($number[k] \neq 0$ **and** ($\langle number[k], k\rangle \ll \langle number[i], i\rangle$))
      do skip
    endwhile;
  endfor;
  critical section
  $number[i] = 0$;
endwhile

dove $\langle a, b\rangle \ll \langle c, d\rangle$ sse $a < c$ oppure $a = c$ e $b \leq d$

- *choosing* definisce stati per i thread
- Controlliamo le interferenze tra scelta dei ticket e confronto