Sincronizzazione dei Processi

Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

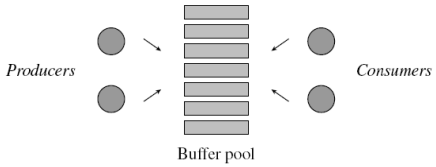
Introduzione

- Problemi di sincronizzazione di processi classici
- Approccio algoritmico per implementare le sezioni critiche

Problemi di sincronizzazione classici

- Una soluzione ad un processo di sincronizzazione dovrebbe soddisfare tre criteri importanti
  - Correttezza
  - Massima concorrenza
  - Nessuna attesa attiva
- Alcuni problemi classici
  - Produttori-Consumatori con buffer limitati
  - Lettori e scrittori
  - Filosofi a cena

Produttori-Consumatori con Buffer Limitati



- Una soluzione deve soddisfare:
  1. Un produttore non deve sovrascrivere un buffer pieno
  2. Un consumatore non deve consumare un buffer vuoto
  3. Produttori e consumatori devono accedere ai buffer in modo mutuamente esclusivo
  4. (opzionale) Le informazioni devono essere consumate nello stesso ordine in cui è messa nei buffer

Produttori-Consumatori con Buffer Limitati (cont.)

```
begin
Parbegin
   var produced : boolean;                 var consumed : boolean;
   repeat                                  repeat
      produced := false                       consumed := false;
      while produced = false                  while consumed = false
         if an empty buffer exists               if a full buffer exists
         then                                    then
            { Produce in a buffer }                 { Consume a buffer }
            produced := true;                       consumed := true;
      { Remainder of the cycle }              { Remainder of the cycle }
   forever;                                forever;
Parend;
end.
```

Producer *Consumer*

- Soffre die due problemi:
  - Poca concorrenza e attese attive

Produttori-Consumatori con Buffer Limitati (cont.)

- Come migliorare lo schema precedente?
  - Necessaria una mutua esclusione per l'accesso ai buffer, ma …
  - E' un problema di segnalazione
    - Dopo che un produttore ha inserito un elemento in un buffer deve segnalarlo al consumatore
    - Dopo che un consumatore ha estratto un elemento dal buffer deve segnalarlo al produttore
- Consideriamo una soluzione migliorata per un solo produttore e d un solo consumatore e un singolo buffer

Produttori-Consumatori con Buffer Limitati (cont.)

check_b_empty blocca il produttore se è vera
chech_b_full blocca il consumatore se vera

```
var
      buffer : . . . ;
      buffer_full : boolean;
      producer_blocked, consumer_blocked : boolean;
begin
      buffer_full := false;
      producer_blocked := false;
      consumer_blocked := false;
Parbegin
   repeat                                  repeat
      check_b_empty;                          check_b_full;
      {Produce in the buffer}                 {Consume from the buffer}
      post_b_full;                            post_b_empty;
      {Remainder of the cycle}                {Remainder of the cycle}
   forever;                                forever;
Parend;
end.
```

Producer *Consumer*

Uno schema migliorato per un sistema produttori-consumatori con singolo buffer usando la segnalazione

Produttori-Consumatori con Buffer Limitati (cont.)

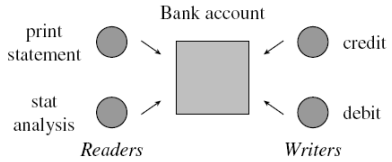
```
procedure check_b_empty                 procedure check_b_full
begin                                   begin
   if buffer_full = true                   if buffer_full = false
   then                                    then
      producer_blocked := true;               consumer_blocked := true;
      block (producer);                       block (consumer);
end;                                    end;

procedure post_b_full                   procedure post_b_empty
begin                                   begin
   buffer_full := true;                    buffer_full := false;
   if consumer_blocked = true              if producer_blocked = true
   then                                    then
      consumer_blocked := false;              producer_blocked := false;
      activate (consumer);                    activate (producer);
end;                                    end;
```

Operations of producer *Operations of consumer*

Operazioni indivisibili per il problema produttori-consumatori

Lettori e Scrittori

- Una soluzione deve soddisfare:
  1. Molti lettori possono leggere concorrentemente
  2. La lettura è proibita mentre uno scrittore sta scrivendo
  3. Solo uno scrittore può eseguire la scrittura in un dato momento
  4. (opzionale) Un lettore ha una priorità non prelazionabile sugli scrittori
     - Nota come *sistema lettori-scrittori con preferenza ai lettori*



Lettori e scrittori in un sistema bancario

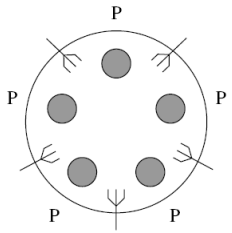
Lettori e Scrittori (cont.)

```
Parbegin
  repeat                                   repeat
     If a writer is writing                   If reader(s) are reading, or a
     then                                        writer is writing
        { wait };                             then
     { read }                                    { wait };
     If no other readers reading              { write }
     then                                     If reader(s) or writer(s) waiting
        if writer(s) waiting                  then
        then                                     activate either one waiting
           activate one waiting writer;          writer or all waiting readers;
  forever;                                 forever;
Parend;
end.

        Reader(s)                                Writer(s)
```

Uno schema del sistema lettori-scrittori

I Filosofi a Cena

- Ogni processo rappresenta un filosofo in modo che ciascuno di essi può mangiare quando ha fame e nessuno muoia di inedia
  - La soluzione non deve incorrere in deadlock o livelock



I Filosofi a Cena (cont.)

Un filosofo preleva la forchetta una alla volta (es. prima sx e poi dx)

Schema di un processo filosofo P_i

```
repeat
   if left fork is not available
   then
      block (Pi);
   lift left fork;
   if right fork is not available
   then
      block (Pi);
   lift right fork;
   { eat }
   put down both forks
   if left neighbor is waiting for his right fork
   then
      activate (left neighbor);
   if right neighbor is waiting for his left fork
   then
      activate (right neighbor);
   { think }
forever
```

- Potenziali deadlock o race condition, a meno che:
  - Se la forchetta di destra non è disponibile, rilascia la forchetta di sinistra, riprova più tardi
    - Soffre di livelock

I Filosofi a Cena (cont.)

Un filosofo preleva ambo le forchette in SC

Uno schema migliorato di processo filosofo

```
var      successful : boolean;
repeat
    successful := false;
    while (not successful)
        if both forks are available then
            lift the forks one at a time;
            successful := true;
        if successful = false
        then
            block (P_i);
    { eat }
    put down both forks;
    if left neighbor is waiting for his right fork
    then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
    then
        activate (right neighbor);
    { think }
forever
```

- Problema: il loop causa una condizione di attesa attiva

Approcci Algoritmici per le Sezioni Critiche

- Algoritmi a due processi
- Algoritmo a n-processi

Approcci Algoritmici per le SC

- Gli approcci algoritmici per implementare le SC non impiegano
  - I servizi del kernel per il blocco e l'attivazione dei processi
    - Per ritardare un processo
  - Istruzioni indivisibili HW
    - Per evitare le race condition
- Indipendenti dal SO e dallo HW, tuttavia …
  - Usano il busy waiting
  - Complesse organizzazioni logiche per evitare le race condition
    - Dimostrazioni di correttezza complicate!

Algoritmi a due processi

turn indica il prossimo processo che in SC

Prima soluzione

```
var      turn : 1 .. 2;
begin
        turn := 1;
Parbegin
    repeat
        while turn = 2
            do { nothing };
        { Critical Section }
        turn := 2;
        { Remainder of the cycle }
    forever;
Parend;
end.
```

Process P_1

```
repeat
    while turn = 1
        do { nothing };
    { Critical Section }
    turn := 1;
    { Remainder of the cycle }
forever;
```

Process P_2

- Viola la condizione del progresso

Algoritmi a due processi (cont.)

Seconda soluzione

c_i variabili di stato.
c_i indica quando P_i è in SC (0) e quando è fuori dalla SC (1)

```
var     c1, c2 : 0 .. 1;
begin
        c1 := 1;
        c2 := 1;
Parbegin
    repeat                                  repeat
        while c2 = 0                            while c1 = 0
            do { nothing };                         do { nothing };
        c1 := 0;                                c2 := 0;
        { Critical Section }                    { Critical Section }
        c1 := 1;                                c2 := 1;
        { Remainder of the cycle }              { Remainder of the cycle }
    forever;                                forever;
Parend;
end.
```

Process P_1 *Process P_2*

- Viola la condizione di mutua esclusione
- Può portare al deadlock

Algoritmo di Dekker

- Combina le soluzioni dei primi due algoritmi
- Se P_1 e P_2 provano ad entrare contemporaneamente in SC, **turn** indica a quale dei due è consentito
  - Se non c'è competizione per entrare, **turn** non ha effetto
- Se entrambi i processi tentano di accedere nelle rispettive SC, **turn** forza una dei due a favorire l'atro

Algoritmo di Dekker

turn è efficace solo quando ambo i processi cercano di entrare nella SC nello stesso tempo

```
var     turn : 1 .. 2;
        c1, c2 : 0 .. 1;
begin
        c1 := 1;
        c2 := 1;
        turn := 1;
Parbegin
    repeat                                  repeat
        c1 := 0;                                c2 := 0;
        while c2 = 0 do                         while c1 = 0 do
            if turn = 2 then                        if turn = 1 then
            begin                                   begin
                c1 := 1;                                c2 := 1;
                while turn = 2                          while turn = 1
                    do { nothing };                         do { nothing };
                c1 := 0;                                c2 := 0;
            end;                                    end;
        { Critical Section }                    { Critical Section }
        turn := 2;                              turn := 1;
        c1 := 1;                                c2 := 1;
        { Remainder of the cycle }              { Remainder of the cycle }
    forever;                                forever;
Parend;
end.
```

Process P_1 *Process P_2*

Algoritmo di Peterson

- Usa un array booleano, flag (un flag per processo)
  - Flag equivalenti alle variabili di stato c1 e c2 in Dekker
- Un processo imposta il flag a true quando intende entrare in SC e lo imposta a false quando ne esce
- Turn è usata per evitare i livelock
- Si suppone che i due processi siano P_0 e P_1 e gli id (0 e 1) sono usati per accedere ai **flag** di stato

Algoritmo di Peterson

```
var     flag : array [0 .. 1] of boolean;
        turn : 0 .. 1;
begin
        flag[0] := false;
        flag[1] := false;
Parbegin
   repeat                                  repeat
      flag[0] := true;                        flag[1] := true;
      turn := 1;                              turn := 0;
      while flag[1] and turn = 1              while flag[0] and turn = 0
          do {nothing};                           do {nothing};
      { Critical Section }                    { Critical Section }
      flag[0] :=false;                        flag[1] :=false;
      { Remainder of the cycle }              { Remainder of the cycle }
      forever;                                forever;
   Parend;
   end.
            Process P0                              Process P1
```


Soluzioni con n processi

- E' necessario conoscere il numero di processi che entrano in SC
  - Dimensione array di stato
  - Controlli per verificare se altri processi desiderano entrare in SC
  - Meccanismo con cui un processo favorisce l'altro
- Con un problema a due processi
  - Ogni processo controlla lo stato di **un solo** processo
- Con un problema a n processi
  - Ogni processo controlla lo stato di **altri n-1** processi
- Algoritmi per n processi più complessi!

Algoritmo del Panettiere (Lamport, 1974)

- Idea
  - Ogni processo prende un numero. Il processo che ha il numero più piccolo è servito
  - **«servire»** significa entrare in SC
- Si usano due array
  - ***choosing[0..n-1]***, dove **choosing[i]** indica se P_i è impegnato nella scelta
  - **number [0..n-1]**, dove **number[i]** contiene il numero scelto da P_i
    - number[i]=0 indica che P_i non ha scelto il numero
- E' servito il processo che ha la coppia **(number[i],i)** minore, dove:

```
(number[j],j) < number[i],i) se
     number[j] < number[i], oppure
     number[j] = number[i] and j < i
```


Algoritmo del Panettiere (cont.)

```
const  n = . . . ;
var    choosing : array [0 .. n − 1] of boolean;
       number : array [0 .. n − 1] of integer;
begin
       for j := 0 to n − 1 do
          choosing[j] := false;
          number[j] := 0;
Parbegin
   process Pi :
       repeat
          choosing[i] := true;
          number[i] := max (number[0], .. ,number[n − 1])+1;
          choosing[i] := false;
          for j := 0 to n − 1 do
          begin
             while choosing[j] do { nothing };
                while number[j] ≠ 0 and (number[j], j) < (number[i],i)
                    do { nothing };
          end;
          { Critical Section }
          number[i] := 0;
          { Remainder of the cycle }
       forever;
   process Pj : . . .
Parend;
end.
```

```
cosa accade
se non usiamo
l'array choosing ?
```