Azzolini Riccardo 2019-10-31

Sincronizzazione

1 Altro esempio di race condition

Una situazione in cui si possono verificare delle race condition è la gestione di una pila condivisa. Si suppone che due processi accedano alla stessa pila, implementata mediante una array `stack` e un indice `top` che punta all'elemento in cima:

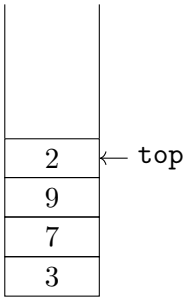
- uno dei processi fa una push, eseguendo il codice:

```
   ...
a  top = top + 1;
b  stack[top] = x;
   ...
```

- l'altro processo fa una pop, con il codice:

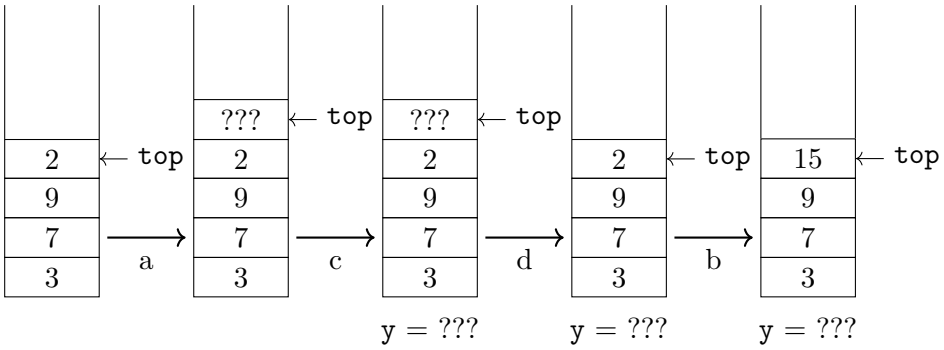
```
   ...
c  y = stack[top];
d  top = top - 1;
   ...
```

Se, per esempio, lo stato iniziale della pila è



e le istruzioni vengono eseguite nell'ordine *a*, *c*, *d*, *b*:

- `y` assume il valore di qualcosa che non è nella pila;
- il valore di `x` (ad esempio, 15) viene inserito al posto del 2, che viene sovrascritto, e quindi cancellato, senza essere mai stato estratto dalla pila.

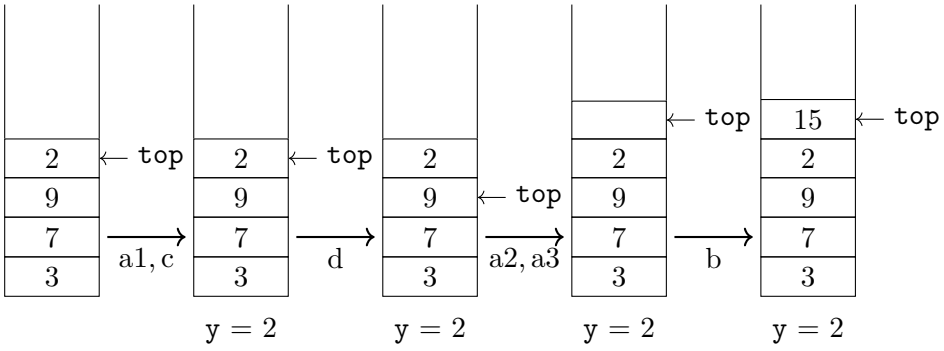


Invece, supponendo che l'istruzione ad alto livello *a* sia implementata con le istruzioni macchina

```
a1: load  top, R0
a2: add   R0,  1
a3: store R0,  top
```

se l'ordine di esecuzione è *a1*, *c*, *d*, *a2*, *a3*, *b*:

- il valore 2 viene estratto e assegnato a `y`;
- il valore di `x` (15) viene inserito sopra al 2, invece di sovrascriverlo, quindi il 2 potrà essere estratto una seconda volta (dopo l'estrazione del 15).



2 Conseguenze delle race condition

Le race condition hanno due conseguenze:

- il comportamento dei processi/thread coinvolti non è corretto;
- i dati su cui ha luogo la race condition diventano *inconsistenti*, cioè assumono uno stato non previsto.

Inoltre, quando si eseguono più volte questi programmi, le race condition si verificano solo alcune delle volte, quindi non è facile verificare la correttezza di tali programmi e/o farne il debugging.

3 Sezioni critiche

Le race condition su un dato condiviso d possono avvenire perché i processi accedono a d concorrentemente. Per evitarle, è quindi necessario impedire gli accessi concorrenti.

Definizione: Una **sezione critica** (**critical section**, **CS**) per un dato condiviso d è una porzione di codice che viene certamente eseguita *non* concorrentemente con se stessa (cioè con altri processi/thread che eseguono lo stesso codice) o con altre sezioni critiche per d.

Si parla anche di **mutua esclusione**, perché le CS su un dato d sono mutualmente esclusive: quando un processo accede a una di esse, impedisce a qualunque altro processo di accedere a tutte le CS sullo stesso dato d.

Se i processi modificano il dato d solo nelle CS per d, allora non si possono avere race condition su tale dato. Inoltre, se i processi accedono il lettura a d solo nelle CS per d, essi vedono solo stati consistenti del dato. Rimane però la difficoltà di *come implementare le CS*.

Nell'esempio delle prenotazioni aeree,

```
S1 if (next_seat <= max) {
S2     booked = next_seat;
S3     next_seat++;
   } else {
S4     printf("sorry, sold out");
   }
S5 ...
```

le race condition sono evitate se le istruzioni S1, S2 e S3 formano una CS.

4 Implementazioni delle sezioni critiche

Le implementazioni delle CS devono soddisfare alcune proprietà:

- **correttezza**: le CS per un dato d non possono essere eseguite concorrentemente, cioè deve essere garantita la mutua esclusione;

- **progresso**: se nessun processo sta eseguendo una CS per d, e alcuni processi manifestano la volontà di eseguire CS per d, allora uno di essi deve poter eseguire la propria CS per d;
- **attesa limitata** (*bounded wait*): dopo che un processo P_i manifesta la volontà di accedere a una CS per d, il numero di accessi alle CS per d da parte di un qualsiasi altro processo P_j che precedono l'accesso di P_i deve essere minore o uguale a un dato intero k (cioè, informalmente, se tanti processi vogliono eseguire CS per d, prima o poi ci riescono tutti).

Le proprietà di progresso e attesa limitata prevengono la **starvation**, cioè l'attesa infinita da parte dei processi.

5 Tentativo errato: disabilitare gli interrupt

Questa soluzione prevede di disabilitare gli interrupt prima della sezione critica, e riabilitarli al termine di quest'ultima:

```
...
disable_interrupt;
{CS}
enable_interrupt;
...
```

(dove la notazione `{CS}` indica una generica CS). Essa:

- è corretta, ma *solo* su sistemi uniprocessore;
- richiede che le istruzioni per disabilitare gli interrupt siano eseguibili in user mode, ma così un programma potrebbe usarle per monopolizzare la CPU.

6 Tentativo errato: uso di variabili lock condivise

Sia `lock` una variabile condivisa inizializzata a 0. Un processo che vuole entrare in una CS controlla prima il valore di `lock`:

- se è 0 ("aperto"), lo imposta a 1, esegue la CS, e infine lo reimposta a 0;
- se è 1 ("chiuso"), significa che la CS è già "occupata", quindi il processo aspetta finché il valore non diventa 0.

```
...
while (lock != 0) {}
lock = 1;
{CS}
lock = 0;
...
```

Questa soluzione non è corretta, perché si potrebbero avere race condition sulla variabile `lock`: più processi potrebbero trovare `lock == 0` (se ciascuno di essi perde la CPU prima di riuscire a porre `lock = 1`), e quindi entrare concorrentemente nelle CS.

Osservazione: Questa soluzione è un tentativo di usare un dato condiviso per risolvere un problema sui dati condivisi!

7 Tentativo errato: variabili di turno condivise

Ciascun processo dà il turno all'altro quando ha finito di eseguire la CS:

- P_0 esegue il codice:

  ```
  ...
  while (turn != 0) {}
  {CS}
  turn = 1;
  ...
  ```

- P_1 esegue il codice:

  ```
  ...
  while (turn != 1) {}
  {CS}
  turn = 0;
  ...
  ```

Questa soluzione:

- si applica a due processi, ma potrebbe essere generalizzata a un numero superiore (diventando però più complicata);
- è corretta, perché in qualsiasi momento il valore di `turn` impedisce a uno dei due processi di accedere alle CS;
- soddisfa la bounded wait, perché i processi si alternano, accedendo (appunto) a turni alle proprie CS;

- *non soddisfa* la proprietà del *progresso*: se il processo che ha il turno non esegue mai la propria CS, non passa neanche il turno all'altro processo, e quest'ultimo rimane quindi bloccato all'infinito (*starvation*).