Azzolini Riccardo 2019-10-30

Sincronizzazione

1 Vantaggi della programmazione concorrente

Su un sistema uniprocessore, programmare un'applicazione con un insieme di processi/thread concorrenti può avere diversi vantaggi:

- *semplicità* di programmazione, se l'applicazione prevede più compiti;
- *efficienza*, se si ha parallelismo tra attività di CPU e di I/O;
- possibilità di assegnare *compiti più urgenti* a processi/thread con *priorità più elevata*, se lo scheduler si basa sulle priorità.

Inoltre, se il processore è multithreading, oppure il sistema è multiprocessore, i vantaggi in termini di efficienza sono ancora più significativi.

La programmazione concorrente presenta però alcune importanti problematiche, legate all'interazione tra processi/thread diversi.

2 Memoria condivisa

Nella discussione che segue, si suppone che più processi abbiano una zona di memoria condivisa. Di norma, i processi hanno memoria separata, ma:

- le strutture dati che il kernel usa per gestire i processi sono effettivamente condivise da tutti i processi;
- molti SO mettono a disposizione delle system call che permettono ai processi di ottenere una porzione di memoria condivisa;
- le stesse problematiche analizzate in seguito si possono avere tra thread (che, per definizione, hanno memoria condivisa), oppure tra processi che usano file condivisi.

2.1 Esempio

Un esempio di memoria condivisa potrebbe essere un sistema banale di prenotazioni aeree, che usa n terminali connessi a un computer centrale. Questo sistema assegna i posti in ordine sequenziale finché non sono esauriti.

Si ha un processo per ciascuno degli n terminali, e tutti i processi condividono due variabili:

- `next_seat`: indica il prossimo posto da assegnare;
- `max`: indica il numero totale di posti sul volo.

Il programma che gli n terminali eseguono per prenotare un posto sul volo è:

```
// ...
if (next_seat <= max) {
    booked = next_seat;
    next_seat++;
} else {
    printf("sorry, flight sold out");
}
// ...
```

`booked` è una variabile privata, cioè non condivisa, nella quale ciascun processo salva il numero del posto che ha prenotato.

3 Tipi di interazioni tra processi

Esistono 4 tipi di interazioni tra processi:

data sharing: i processi condividono dati situati in memoria/file condivisi;

control synchronization: un'azione a_i di un processo P_i è abilitata solo dopo che un altro processo P_j ha svolto un'azione a_j;

message passing: un processo P_i invia un messaggio a un processo P_j, che lo riceve;

signals: un processo P_i invia un segnale (che è un messaggio vuoto) a un altro processo P_j, per segnalare una situazione particolare.

4 Processi interagenti e indipendenti

Dato un processo P_i, si dicono:

- **read_set** di P_i l'insieme R_i dei dati letti da tale processo;
- **write_set** di P_i l'insieme W_i dei dati modificati da tale processo.

Nota: Se P_i legge/scrive un dato solo in alcune circostanze (ad esempio, perché l'operazione di lettura/modifica si trova in un `if`), questo si considera comunque parte del read_set/write_set: in pratica, si considera la "possibilità" che un dato venga letto/modificato.

Due processi concorrenti P_i e P_j sono **processi interagenti** se vale *almeno una* delle seguenti proprietà:

- R_i e W_j hanno intersezione non vuota;
- R_j e W_i hanno intersezione non vuota.

$$R_i \cap W_j \neq \varnothing \quad \vee \quad R_j \cap W_i \neq \varnothing$$

Due processi concorrenti P_i e P_j si dicono invece **processi indipendenti** se non sono interagenti.

Osservazione: Queste definizioni valgono anche se P_i e P_j sono thread.

Nell'esempio delle prenotazioni aeree, ciascuno degli n processi è interagente con tutti gli altri, perché ogni processo legge e modifica la variabile `next_seat`, che quindi è sia nel read_set che nel write_set di ciascun processo.

5 Comportamento dei processi interagenti

Se due processi/thread concorrenti P_i e P_j sono *indipendenti*:

- competono per le risorse (es. CPU), rallentandosi a vicenda;
- i loro comportamenti *non dipendono* dalla loro velocità relativa, e sono *riproducibili*.

Se, invece, due processi/thread P_i e P_j sono *interagenti*:

- competono per le risorse (es. CPU), rallentandosi a vicenda;
- i loro comportamenti *dipendono* dalla loro velocità relativa, e **non sono riproducibili** (cioè esecuzioni diverse, anche con gli stessi input, possono dare risultati diversi).

Nell'esempio delle prenotazioni aeree, se P_i e P_j sono due degli n processi interagenti, le prenotazioni effettuate da uno influiscono sulle prenotazioni dell'altro. Di conseguenza, l'esecuzione di P_i non è riproducibile, in quanto dipende dalle velocità relative di P_i, P_j, e degli altri processi.

Osservazione: Il fatto che i comportamenti di processi concorrenti non siano riproducibili è inevitabile,[1] *perfettamente accettabile*, e a volte desiderabile. Ad esempio, se si tenta di prenotare contemporaneamente da due terminali l'ultimo posto disponibile su un volo, è perfettamente accettabile che la prenotazione vada a buon fine solo per uno dei due processi.

5.1 Esempio di comportamento non riproducibile

- Sia `x` una variabile `int` condivisa, inizializzata a 100.
- Sia P_i un processo che esegue il codice:

```
// ...
int y = x + 5;
printf("%d", y);
// ...
```

- Sia P_j un processo, concorrente a P_i, che esegue il codice:

```
// ...
x = 10;
// ...
```

P_i e P_j sono processi interagenti, perché la variabile `x` è nel read_set di P_i e nel write_set di P_j, quindi $R_i \cap W_j = \{\texttt{x}\} \neq \varnothing$.

Il comportamento di P_i non è riproducibile: eseguendo P_i e P_j più volte,

- a volte P_i stampa 105;
- a volte P_i stampa 15.

Il risultato dipende dall'ordine di esecuzione delle operazioni di lettura e modifica del dato condiviso `x`, cioè dalla velocità di esecuzione relativa dei due processi. Essa è imprevedibile perché dipende, a sua volta:

- dalla politica di scheduling;
- da quanti e quali altri processi sono in esecuzione;
- dalle eventuali priorità di P_i, P_j, e degli altri processi;

[1] Esistono delle tecniche per forzare esplicitamente l'ordine di esecuzione di determinate azioni, ma spesso ciò non serve (ad esempio, non ha senso fissare l'ordine in cui vengono eseguite le prenotazioni).

- ecc.

Implementando P_i e P_j come thread POSIX, il codice completo del programma è:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int x = 100; // variabile globale condivisa

void *f1(void *arg) {
    x = 10; // modifica di x
    pthread_exit(NULL);
}

void *f2(void *arg) {
    int y = x + 5; // lettura di x
    printf("y vale %d\n", y);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, f1, NULL);
    pthread_create(&t2, NULL, f2, NULL);
}
```

I due possibili risultati (*entrambi corretti*) di questo programma sono:

- `y vale 105`
- `y vale 15`

6 Race condition

Il programma seguente usa la funzione `pthread_join`, che mette il thread chiamante in waiting finché non termina un altro thread (passato come argomento):

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int x = 100; // variabile globale condivisa

void *f1(void *arg) {
```

```
    x = x + 5; // modifica di x
    pthread_exit(NULL);
}

void *f2(void *arg) {
    x = x + 10; // modifica di x
    pthread_exit(NULL);
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, f1, NULL);
    pthread_create(&t2, NULL, f2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    // I thread t1 e t2 sono terminati.
    printf("x vale %d\n", x); // lettura di x
}
```

Siccome la stampa di `x` avviene per forza dopo l'esecuzione del codice di `f1` e `f2`, con un ragionamento superficiale ci si potrebbe aspettare un solo risultato:

```
x vale 115
```

Invece, l'esecuzione di questo programma può dare tre risultati diversi:

- `x vale 115`
- `x vale 110`
- `x vale 105`

Per spiegare questo fenomeno, è necessario ragionare direttamente con le istruzioni macchina (anziché con quelle in C). Infatti, ciascuna delle operazioni `x = x + 5` e `x = x + 10` è realizzata da più istruzioni macchina, probabilmente in una sequenza *load-add-store*:

- `x = x + 5` potrebbe corrispondere a

  ```
  l1: load  X,  R0
  a1: add   5,  R0
  s1: store R0, X
  ```

- `x = x + 10` potrebbe corrispondere a

  ```
  l2: load  X,  R0
  a2: add   10, R0
  s2: store R0, X
  ```

dove `X` è l'indirizzo della variabile `x` e `R0` è un registro generale.

Allora, in base allo scheduling, si potrebbe avere ad esempio la seguente esecuzione:

1. `l1`: nel registro `R0` viene posto il valore 100, letto dalla variabile `x`;
2. il primo thread perde la CPU: il suo contesto viene salvato, e poi viene schedulato il secondo thread;
3. `l2`: nel registro `R0` viene posto il valore 100, letto da `x`;
4. `a2`: il valore di `R0` diventa 110;
5. `s2`: il valore 110 contenuto in `R0` viene salvato nella variabile `x`;
6. prima o poi viene rischedulato il primo thread, e dal suo TCB viene ripristinato il valore 100 di `R0`;
7. `a1`: il valore di `R0` diventa 105;
8. `s1`: il valore 105 contenuto in `R0` viene salvato in `x`, sovrascrivendo il valore 110 che vi aveva salvato il secondo thread.

In quest'esecuzione, l'operazione `x = x + 10` effettuata dal secondo thread si è persa. In altre esecuzioni, potrebbe invece perdersi l'aggiornamento `x = x + 5` effettuato dal primo thread. Invece, si ottiene il risultato corretto solo se ciascuna sequenza load-add-store viene eseguita per intero, senza perdere la CPU.

Un altro esempio di programma che può dare risultati errati è quello delle prenotazioni aeree.

```
S1 if (next_seat <= max) {          S1,1 load   max, R0
                                    S1,2 sub    R0, next_seat
                                    S1,3 jmpneg R0, S4,1

S2     booked = next_seat;          S2,1 move   next_seat, booked

S3     next_seat++;                 S3,1 load   next_seat, R1
   } else {                         S3,2 add    R1, 1
                                    S3,3 store  R1, next_seat
                                    S3,4 jmp    S5,1

S4     printf("sorry, sold out");   S4,1 move   "sorry", Rvideo
   }
S5 ...                              S5,1 ...
```

Siano `max = 200` e `next_seat = 200`. Ciò significa che c'è un solo posto libero (il numero 200). I processi P_1 e P_2 tentano di prenotare il volo. Tre delle possibili esecuzioni sono:

- Caso 1 – corretto:

Tempo	P_1	P_2
1	S1,1	
2	S1,2	
3	S1,3	
4	**S2,1**	
5	S3,1	
6	S3,2	
7	S3,3	
8	S3,4	
9		S1,1
10		S1,2
11		S1,3
12		S4,1

  P_1 prenota il posto (memorizzandone il numero con l'istruzione S2,1), mentre P_2 non trova più posti disponibili.

- Caso 2 – errato:

Tempo	P_1	P_2
1	S1,1	
2	S1,2	
3	S1,3	
4		S1,1
5		S1,2
6		S1,3
7		**S2,1**
8		S3,1
9		S3,2
10		S3,3
11		S3,4
12	**S2,1**	
13	S3,1	
14	S3,2	
15	S3,3	
16	S3,4	

  P_2 prenota il posto 200, P_1 prenota il posto 201 (*che non esiste*), e `next_seat` assume il valore 202 (mentre, normalmente, non dovrebbe mai superare 201).

- Caso 3 – errato:

Tempo	P_1	P_2
1	S1,1	
2	S1,2	
3	S1,3	
4	**S2,1**	
5	S3,1	
6		S1,1
7		S1,2
8		S1,3
9		**S2,1**
10		S3,1
11		S3,2
12		S3,3
13		S3,4
14	S3,2	
15	S3,3	
16	S3,4	

entrambi i processi prenotano il posto 200, e `next_seat` assume valore 201.

I risultati sbagliati ottenuti da questi due programmi sono esempi di *race condition*: intuitivamente, tali risultati non sono accettabili perché non è possibile spiegarli come l'esecuzione in sequenza, in un ordine qualsiasi, delle operazioni svolte dai due thread.

6.1 Definizione

Siano:

- d un dato condiviso dai processi (/thread) P_1 e P_2;
- o_1 e o_2 operazioni su d eseguite, rispettivamente, da P_1 e P_2;
- f_1 e f_2 funzioni tali che, se d ha valore v_d:
  - $f_1(v_d)$ è il valore assunto da d eseguendo o_1;
  - $f_2(v_d)$ è il valore assunto da d eseguendo o_2.

Le operazioni o_1 e o_2 danno luogo a una **race condition** (**corsa critica**) sul dato condiviso d se, eseguendo o_1 e o_2 con d avente un valore iniziale v_d, accade che d assume un valore v'_d diverso da $f_1(f_2(v_d))$ e da $f_2(f_1(v_d))$.

6.2 Esempi di applicazione della definizione

Nell'esempio riportato all'inizio della sezione sulle race condition:

- il dato condiviso d è la variabile `x`, e ha valore iniziale $v_d = 100$;
- l'operazione o_1 è l'istruzione `x = x + 5`;
- l'operazione o_2 è `x = x + 10`;
- la funzione f_1 è tale che $f_1(z) = z + 5$;
- la funzione f_2 è tale che $f_2(z) = z + 10$.

I due risultati accettabili sono allora

- $f_1(f_2(100)) = 115$
- $f_2(f_1(100)) = 115$

che sono uguali perché la somma è commutativa. Tutti gli altri risultati ottenuti (110 e 105) sono quindi dovuti a delle race condition.

Possono anche esserci due risultati accettabili diversi (se le due operazioni non sono commutative). Ad esempio, sostituendo l'operazione o_1 con `x = x * 5`,

```
void *f1(void *arg) {
    x = x * 5; // modifica di x
    pthread_exit(NULL);
}
```

la funzione f_1 diventerebbe $f_1(z) = 5z$, e i due risultati accettabili sarebbero:

- $f_1(f_2(100)) = 550$
- $f_2(f_1(100)) = 510$

Oltre a questi due, il programma così modificato potrebbe dare anche i risultati 110 e 500, a causa delle race condition.

Invece, l'esempio

```
int x = 100; // variabile globale condivisa

void *f1(void *arg) {
    x = 10; // modifica di x
    pthread_exit(NULL);
}

void *f2(void *arg) {
    int y = x + 5; // lettura di x
    printf("y vale %d\n", y);
```

```
    pthread_exit(NULL);
}
```

non è soggetto a race condition, perché, se v è il valore di `x` (inizialmente 100) e u è il valore di `y` (inizialmente indefinito), le due funzioni sono

- $f_1(v, u) = (v' = 10,\ u' = u)$
- $f_2(v, u) = (v' = v,\ u' = v + 5)$

e allora i risultati accettabili sono:

- $f_1(f_2(100, u)) = (v' = 10,\ u' = 105)$
- $f_2(f_1(100, u)) = (v' = 10,\ u' = 15)$

In altre parole, il valore finale di `y` (che viene stampato) può essere 105 o 15, mentre `x` deve valere 10 al termine del programma, ed entrambe queste condizioni sono sempre verificate.

Intuitivamente, infatti, l'operazione `y = x + 5` non dà problemi perché l'istruzione load che carica il valore di `x` può leggere

- o il valore originale, 100, che porta al risultato 105
- oppure il valore modificato, 10, che dà come risultato 15

mentre l'operazione dell'altro thread, `x = 10`, non dà problemi perché viene sempre eseguita (prima o poi), garantendo che il valore finale di `x` sia quello giusto. In particolare, non ci sono valori intermedi che possono essere erroneamente memorizzati e ripristinati, causando la perdita di altre operazioni.