Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte P: Programmazione di Sistema e Concorrente

cap. P3 – Programmazione Concorrente

3. PROGRAMMAZIONE CONCORRENTE

3.1 Introduzione

La programmazione usuale si basa su un modello di esecuzione sequenziale, cioè sull'ipotesi che le istruzioni di un programma vengano eseguite una dopo l'altra in un ordine predeterminabile in base al codice del programma ed eventualmente ai dati di input. Questo modello si applica a un singolo programma eseguito da un processo in assenza di thread multipli (cioè con il solo thread principale o di default).

Diversi processi possono eseguire diversi programmi in parallelo, ma finchè i diversi processi non interferiscono tra loro il programmatore non ha alcun bisogno di tenerne conto, quindi l'esistenza di diversi processi da sola non altera il paradigma della programmazione sequenziale.

Dal punto di vista del programmatore le cose si complicano quando diverse attività eseguite in parallelo interferiscono tra loro scambiandosi dei messaggi oppure accedendo a strutture dati condivise. La programmazione di questo tipo è detta programmazione concorrente e costituisce l'oggetto di questo capitolo. Lo studio di questo argomento richiede un lavoro attivo: le nozioni da apprendere sono poche, ma è necessario un esercizio di riflessione. Per questo motivo su alcuni argomenti invece di spiegare i problemi, questi sono presentati come esercizi il cui svolgimento deve avvenire contestualmente alla lettura del testo. Per gli esercizi marcati con un * è fornita una soluzione alla fine del capitolo. Anche il codice dei programmi costituisce parte integrante del testo.

3.2 Concorrenza, parallelismo e parallelismo reale

I problemi della programmazione concorrente si presentano tutte le volte che diversi flussi di controllo concorrenti interferiscono tra loro, indipendentemente dal tipo di motivazione per cui questa situazione si verifica, ad esempio multithreading, processi comunicanti tra loro, routine di interrupt che lavorano su dati comuni, ecc...

Per semplicità nel seguito useremo la terminologia e baseremo gli esempi sul multithreading.

La scrittura di programmi concorrenti, nei quali cooperano diversi processi e/o thread è più difficile della normale scrittura di programmi sequenziali, eseguiti isolatamente da un singolo processo.

Per questo motivo sono state sviluppate diverse tecniche che permettono di evitare molti degli errori ai quali questo tipo di programmazione è soggetta; le tecniche della programmazione concorrente si applicano sia ai processi che ai thread, però sono più tipicamente necessarie con i thread, perchè, come abbiamo visto, i thread sono più utilizzati per realizzare flussi di controllo paralleli ma fortemente cooperanti.

3.3 Modello di esecuzione: parallelismo e non determinismo

Il modello di esecuzione del processo è deterministico, cioè è possibile stabilire dopo l'esecuzione di un'istruzione A quale sarà la prossima istruzione che verrà eseguita (se A è un'istruzione condizionale è necessario conoscere anche il valore delle variabili del programma).

Il modello di esecuzione dell'Hardware invece è non-deterministico, perchè la prossima istruzione che verrà eseguita può dipendere anche dal verificarsi di eventi il cui esatto istante di accadimento è imprevedibile (vedremo molti esempi di eventi di questo tipo: si pensi alla terminazione di un'operazione da parte di una periferica o all'arrivo di dati sulla rete, ecc...).

Il modello di esecuzione dell'Hardware è anche caratterizzato dall'esistenza di parallelismo reale, perchè i diversi dispositivi (processore e periferiche) che costituiscono l'Hardware funzionano in parallelo; inoltre, in alcuni casi, che non saranno trattati in questo testo, il calcolatore possiede più di un processore (sistemi multiprocessore) e quindi in questi casi il parallelismo riguarda anche la stessa esecuzione delle istruzioni dei programmi.

Spesso un programma con thread multipli è non-deterministico, cioè non si può dire in base al codice del programma cosa accadrà esattamente, anche in assenza di dipendenza dai dati di input.

Ad esempio, si consideri un programma con 2 thread che stampano le sequenze di 3 lettere abc e xyz rispettivamente. Si provi a domandarsi quali dei seguenti risultati sono possibili:

R1) abcxyz

R2) xyzabc

R3) axbycz

R4) aybcxz

altri ...

I risultati R1, R2, e R3 sono tutti possibili; i primi due corrispondono all'esecuzione in sequenza di un thread dopo l'altro, che è una situazione possibile, anche se non certa, il terzo è dovuto all'alternanza tra i due thread dopo ogni stampa di un singolo carattere.

Si osservi che il risultato R4 invece NON può verificarsi, perchè i due thread sono singolarmente sequenziali anche se sono in concorrenza tra loro, e quindi nel risultato le due sequenze abc e xyz possono essere mescolate, ma il loro ordinamento parziale deve essere rispettato.

Il figura 3.1 è mostrata una realizzazione del programma appena descritto e in figura 3.2 è mostrato il risultato di una serie di esecuzioni di tale programma, che mette in evidenza il suo non-determinismo.

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.  void * tf1(void *arg)
4.  {
5.  printf("x");
6.  printf("y");
7.  printf("z");
8.  return NULL;
9.  }
10. void * tf2(void *arg)
11. {
12. printf("a");
13. printf("b");
14. printf("c");
15. return NULL;
16. }
17. int main(void)
18. {
19. pthread_t tID1;
20. pthread_t tID2;
21. pthread_create(&tID1, NULL, &tf1, NULL);
22. pthread_create(&tID2, NULL, &tf2, NULL);
23. pthread_join(tID1, NULL);
24. pthread_join(tID2, NULL);
25. printf("\nfine\n");
26. return 0;
27. }
```

Figura 3.1

AVVERTENZA

I risultati delle esecuzioni di questo e dei successivi esempi di programmi di questo capitolo sono stati ottenuti inserendo in mezzo alle istruzioni dei ritardi come i seguenti:

for (i=0; i<100000; i++); oppure sleep(n);

Tali ritardi aggiuntivi non sono mostrati nel testo, perchè essi non alterano la logica del programma ma rendono molto più probabile il verificarsi di sequenze di esecuzione che sarebbero altrimenti altamente rare.

Pertanto chi tentasse di ottenere i risultati presentati nel testo senza inserire ritardi non si stupisca se dovrà eseguire il programma un numero altissimo (migliaia, milioni?) di volte, cercando di variare le condizioni al contorno.

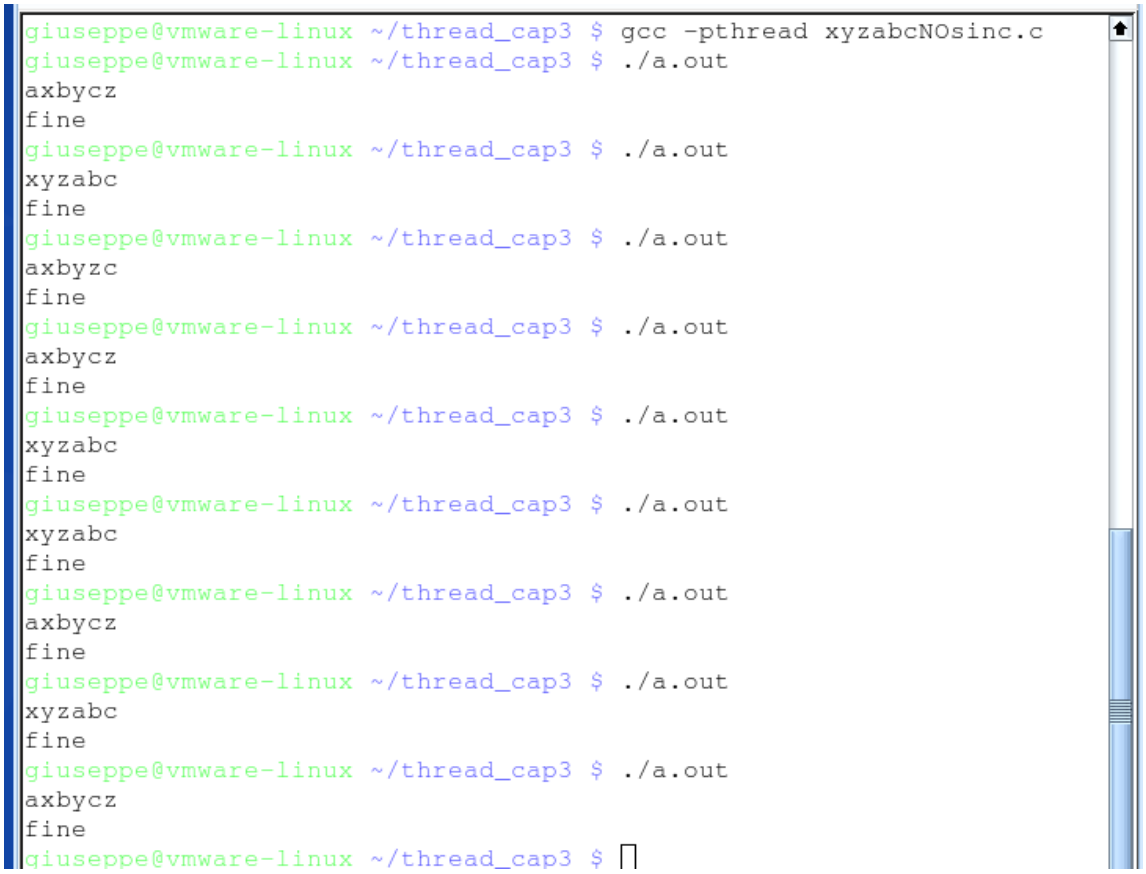


Figura 3.2

3.4 Non determinismo e testing

Abbiamo visto che se eseguissimo il programma precedente molte volte, ogni volta potremmo ottenere la stampa di una sequenza diversa di caratteri. Per questo motivo il testing di programmi concorrenti è molto difficile: supponiamo che un programmatore abbia scritto il programma dell'esempio precedente avendo come obiettivo di produrre esattamente la stampa "axbycz", cioè i caratteri delle due sequenze in alternanza, e lo abbia eseguito molte volte ottenendo sempre la sequenza desiderata: il programmatore potrebbe convincersi erroneamente che il programma produce sempre tale sequenza, mentre in realtà molte sequenze diverse potrebbero verificarsi in situazioni diverse. Non deve quindi sorprendere se talvolta capita che programmi concorrenti che hanno funzionato benissimo per migliaia di volte e sono quindi super-testati improvvisamente producono un errore, causando magari dei disastri.

Si noti la differenza rispetto al testing di un programma sequenziale: è vero che anche un programma sequenziale produce diversi risultati, ma tali risultati dipendono dai dati di input e il programmatore può esplorare i comportamenti del programma fornendogli input diversi. Per quanto il testing non dimostri la correttezza del programma, nel caso sequenziale costituisce uno strumento efficace per la verifica.

Da queste considerazioni segue una conseguenza importante: ***gli errori dei programmi concorrenti non si possono determinare ed eliminare tramite testing ma devono essere evitati tramite una programmazione particolarmente accurata.***

In particolare, è necessario cercare di "dimostrare" che il programma è corretto. Tuttavia, una dimostrazione formale della correttezza di un programma concorrente è molto difficile da realizzare e fuori dagli obiettivi di questo capitolo; noi ci limiteremo a fare dei "**ragionamenti strutturati**", cioè dei ragionamenti basati su proposizioni accuratamente analizzate.

3.5 Mutua Esclusione e Sequenze Critiche

Un tipico problema nella programmazione concorrente è legato alla necessità di garantire che alcune sequenze di istruzioni di 2 thread siano eseguite in maniera sequenziale tra loro.

Si consideri ad esempio il seguente problema: sono dati 2 conti bancari, *contoA* e *contoB*, e si vuole realizzare una funzione, che chiamiamo *trasferisci( )*, che preleva un importo dal contoB e lo deposita sul contoA.

La struttura fondamentale della funzione possiamo immaginare che sia la seguente (si osservi che contoA e contoB devono essere considerati variabili globali e persistenti, tipicamente memorizzate in un file o in un database):

1. leggi contoA in una variabile locale cA;
2. leggi contoB in una variabile locale cB;
3. scrivi in contoA il valore cA + importo;
4. scrivi in contoB il valore cB - importo;

Si osservi che dopo l'esecuzione di tale funzione la somma dei due conti dovrà essere invariata; una proprietà di questo tipo è detta **invariante** della funzione.

Supponiamo ora che tale funzione sia invocata da un main( ) che riceve richieste di trasferimento tra i conti A e B (il tipico bonifico bancario) da diversi

terminali. Per rendere più veloce il sistema, il main crea un diverso thread per ogni attivazione di funzione. I diversi thread eseguiranno quindi le 4 operazioni concorrentemente e saranno possibili molte sequenze di esecuzione diverse tra loro.

Per indicare una operazione all'interno di una sequenza di esecuzione introduciamo la seguente notazione:

ti.j indica l'operazione j svolta dal thread ti:

Con questa notazione possiamo indicare alcune delle possibili sequenza di esecuzione (ipotizziamo nei seguenti esempi che i thread siano 2):

S1) t1.1 < t1.2 < t1.3 < t1.4 < t2.1 < t2.2 < t2.3 < t2.4

S2) t2.1 < t2.2 < t2.3 < t2.4 < t1.1 < t1.2 < t1.3 < t1.4

S3) t1.1 < t1.2 < t1.3 < t2.1 < t2.2 < t2.3 < t2.4 < t1.4

…

Esercizio 1. Determinare il risultato di queste esecuzioni, supponendo che i valori inziali siano contoA=100 e contoB=200 e che i thread t1 e t2 trasferiscano rispettivamente gli importi 10 e 20.

Soluzione 1.

1) Il risultato corretto dovrebbe essere sempre contoA=130, contoB=170 e totale=300.

2) Le sequenze S1 e S2 corrispondono alle esecuzioni sequenziali dei 2 thread t1<t2 e t2 <t1, quindi danno sicuramente risultati corretti

3) Analizziamo ora la sequenza S3: si consideri l'effetto dell'esecuzione di ogni operazione svolta dai due thread, riportato nella seguente tabella (sono indicati solamente i cambiamenti di valore delle variabili)

	inizio	dopo t1.1	dopo t1.2	dopo t1.3	dopo t2.1	dopo t2.2	dopo t2.3	dopo t2.4	dopo t1.4
contoA	100			110			130		
contoB	200							180	190
cA (in t1)		100							
cB (in t1)			200						
cA (in t2)					110				
cB (in t2)						200			

Il risultato finale è contoA=130, contoB=190; inoltre l'invariante alla fine vale 130+190=320, quindi il risultato è errato■

Esercizio 2. determinare altre sequenze di esecuzione possibili e il loro risultato■

In figura 3.3 è mostrata una realizzazione in C del programma descritto sopra; per semplicità i valori iniziali e gli importi trasferiti sono assegnati come costanti.

Il main() crea 2 thread ai quali fa eseguire la funzione *trasferisci( )* passandole l'importo come argomento, poi attende la terminazione dei thread e infine stampa il valore dei due conti e il totale (l'invariante), che dovrebbe essere sempre lo stesso.

La funzione esegue le 4 operazioni indicate sopra alle righe 13, 14, 15 e 16.

Il risultato di una serie di esecuzioni del programma è mostrato in figura 3.4; si può osservare che solo due delle 6 esecuzioni hanno fornito il risultato corretto, mentre le altre sono sicuramente errate, perchè hanno modificato l'invariante (che deve essere sempre 300 in questo caso).

Esercizio 3*. Per ogni risultato di figura 3.4 si determini almeno una sequenza di esecuzione che produce tale risultato■

I due esempi precedenti mostrano un tipico problema della programmazione concorrente: per ottenere un risultato corretto alcune sequenze di istruzioni non devono essere mescolate tra loro durante l'esecuzione. Chiameremo una sequenza di istruzioni di questo genere una **sequenza critica** e chiameremo **mutua esclusione** la proprietà che vogliamo garantire a tali sequenze. Nel testo del programma di figura 3.3 l'inizio e la fine della sequenza critica sono indicati come commenti.

Per essere sicuri di ottenere un'esecuzione corretta dovremo garantire che tutte le istruzioni che costituiscono una sequenza critica siano eseguite in maniera sequenziale. Si osservi che la presenza di sequenze critiche obbliga a ridurre la concorrenza, quindi sarà necessario cercare di limitare le sequenze critiche alle istruzioni che creano effettivamente un problema nell'esecuzione concorrente.

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.  //
4.  int contoA=100, contoB=200;
5.  int totale;
6.  //
7.  void * trasferisci(void *arg)
8.  {
9.  int importo=*((int*)arg);
10. int cA,cB;
11. //inizio sequenza critica
12. cA = contoA; //leggi contoA in variabile locale
13. cB = contoB; //leggi contoB in variabile locale
14. contoA = cA + importo;
15. contoB = cB - importo;
16. //fine sequenza critica
17. return NULL;
18. }
19. //
20. int main(void)
21. {
22. pthread_t tID1;
23. pthread_t tID2;
24. int importo1=10, importo2=20;
25. pthread_create(&tID1, NULL, &trasferisci, &importo1);
26. pthread_create(&tID2, NULL, &trasferisci, &importo2);
27. pthread_join(tID1, NULL);
28. pthread_join(tID2, NULL);
29. totale=contoA + contoB;
30. printf("contoA = %d  contoB = %d\n", contoA, contoB);
31. printf("il totale è %d\n", totale);
32. return 0;
33. }
```

Figura 3.3

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 180
il totale è 310
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
```

Figura 3.4

Molto spesso una sequenza critica è costituita da un insieme di istruzioni che devono essere eseguite tutte per trasformare una struttura dati da uno stato corretto ad un altro stato corretto, mentre una loro esecuzione parziale crea uno stato non corretto. Durante l'esecuzione di una sequenza di questo tipo è possibile eseguire concorrentemente soltanto istruzioni che non utilizzino in alcun modo la struttura dati oggetto dell'aggiornamento, altrimenti queste istruzioni osserverebbero uno stato dei dati non corretto.

3.6 Modello di esecuzione e istruzioni atomiche

L'esempio precedente conduce anche ad un altro tipo di osservazione. Si potrebbe obiettare con riferimento a tale esempio che la soluzione del problema è banale: basterebbe sostituire le 4 istruzioni della funzione trasferisci con due sole istruzioni.

a. contoA = contoA + importo;
b. contoB = contoB – importo;

Esercizio4. si simulino tutte le possibili esecuzioni concorrenti di queste 2 istruzioni e si verifichi che non producono errori. Si cerchi la spiegazione di questa differenza rispetto alle 4 istruzioni utilizzate precedentemente.

Soluzione4.

Le sequenze sono 6:

S1) t1.a < t1.b < t2.a < t2.b

S2) t1.a < t2.a < t1.b < t2.b

S3) t1.a < t2.b < t2.b < t1.a

e altre 3 ottenibili per simmetria scambiando t1 con t2.

S1 corrisponde all'esecuzione sequenziale dei 2 thread, quindi fornisce un risultato corretto.

Simuliamo la S2, che risulta corretta

	inizio	dopo t1.a	dopo t2.a	dopo t1.b	dopo t2.b
contoA	100	110	130		
contoB	200			190	170

Simuliamo la S3, che risulta corretta

	inizio	dopo t1.a	dopo t2.a	dopo t2.b	dopo t1.b
contoA	100	110	130		
contoB	200			180	170

Quindi, per simmetria, tutte le sequenze possibili sono corrette■

In realtà il risultato di questa analisi è ingannevole, perchè presuppone che le operazioni a e b siano indivisibili o **atomiche**, cioè eseguite completamente oppure non eseguite affatto, e che quindi una sequenza possibile di esecuzione debba essere costituita da un ordinamento di tali operazioni. La situazione reale però è diversa: il programma eseguito dal sistema è l'eseguibile in linguaggio macchina prodotto dal compilatore, e la commutazione tra processi o thread diversi può avvenire tra due qualsiasi istruzioni della macchina, anche tra istruzioni che derivano dalla traduzione di un solo statement in linguaggio C.

E' quindi necessario, per analizzare a fondo un programma concorrente rispondere alla seguente domanda fondamentale: esistono delle operazioni del calcolatore che possiamo considerare per loro natura **atomiche**, cioè tali per cui non è possibile che altre operazioni siano svolte nel mezzo?

Ebbene, la risposta a questa domanda deve essere articolata in due parti, una positiva e una negativa:

- le istruzioni elementari del linguaggio macchina del calcolatore sono garantite essere atomiche, cioè vengono sempre eseguite interamente senza permettere che altre operazioni si inseriscano nel mezzo,
- ma (parte negativa!), il programmatore che utilizza un linguaggio diverso dal linguaggio macchina non può sapere a quante istruzioni macchina corrisponde una istruzione di tale linguaggio, perchè questa è una scelta del compilatore.

Esempio: l'istruzione in linguaggio C

```
contoA = contoA + 100;
```

potrebbe essere tradotta in assemblatore come un'unica istruzione

```
ADD  #100, contoA  //somma 100 a contoA
```

oppure come 3 istruzioni appoggiate su un registro R1

```
MOVE  contoA, R1   //copia contoA in R1
ADD    #100, R1        //somma 100 a R1
MOVE   R1, contoA   //copia R1 in contoA
```

o anche in molte altre forme diverse.

La conseguenza di queste considerazioni è che dobbiamo in generale considerare anche operazioni costituite da una unica istruzione del linguaggio di programmazione come una sequenza di operazioni; perfino l'istruzione "i++" non può essere considerata atomica.

Esercizio 5*. Mostrare che 2 thread che eseguono l'operazione i++ sulla variabile condivisa i possono produrre un risultato errato.

Traccia di soluzione 5

1) Tradurre lo statement i++ in una sequenza di almeno 2 istruzioni macchina;

2) Scrivere una sequenza di operazioni utilizzando la notazione introdotta in precedenza, nella quale le singole operazioni dei thread sono le istruzioni macchina, trovando una sequenza di operazioni che incrementa i una volta sola invece che 2.■

A questo punto sorge una domanda: come deve comportarsi il programmatore ad alto livello, che non conosce la traduzione in linguaggio macchina, per garantire il funzionamento corretto dei programmi?

In generale, se ragioniamo su un linguaggio ad alto livello come il C, i cui costrutti non sono atomici, si pone il problema delle possibili sequenze di esecuzione. Infatti, considerando due operazioni t1.i e t2.j eseguite da due thread, le relazioni tra queste due operazioni possono essere non solamente t1.i < t2.j oppure t2.j < t1.i, ma anche **t1.i :: t2.j**, dove col simbolo **::** vogliamo indicare un'esecuzione concorrente in cui le istruzioni macchina che implementano le due operazioni sono variamente alternate.

Ne consegue che le sequenze di operazioni che si realizzano fisicamente sono molte di più di quelle ottenute permutando le istruzioni a livello C. Per affrontare questa difficoltà noi ci baseremo sul fatto che nella maggior parte dei casi due istruzioni C concorrenti sono serializzabili, cioè il risultato dell'esecuzione di t1.i :: t2.j è equivalente all'esecuzione sequenziale t1.i < t2.j oppure t2.j < t1.i, e in tal caso possiamo ragionare su tali sequenze.

Esempio: supponiamo che x e y siano due variabili intere,

t1.i sia y= x+2;

t2.j sia y= x+4;

e che X valga 10 prima dell'esecuzione,

l'esecuzione sequenziale t1.i < t2.j produce y == 14, mentre

l'esecuzione sequenziale t2.j < t1.i produce y == 12.

Se immaginiamo che i due assegnamenti siano realizzati con tre istruzioni macchina come;

t1.i1	MOVE X, R1	t2.j1	MOVE X, R0
t1.i2	ADD #2, R1	t2.j2	ADD #4, R0
t1.i3	MOVE R1, Y	t2.j3	MOVE R0, Y

qualsiasi esecuzione concorrente di queste due sequenze di istruzioni produce 12 oppure 14, cioè è equivalente ad un'esecuzione sequenziale, e quindi t1.i e t2.j sono serializzabili■

In generale supporremo che non siano serializzabili modifiche della stessa variabile basate sul valore della variabile stessa, cioè del tipo x = f(x). Abbiamo visto negli esempi precedenti che operazioni di questo tipo non sono serializzabili e producono un risultato errato, e quindi dobbiamo applicare la regola che ***ogni volta che diversi thread modificano la stessa variabile in base al suo valore precedente bisogna garantire che le operazioni su tale variabile costituiscano sequenze critiche in mutua esclusione***.

3.7 Mutua esclusione – Mutex

Garantire la mutua esclusione delle sequenze critiche utilizzando solamente i normali linguaggi di programmazione sequenziali è difficile, come mostreremo più avanti. Per semplificare questo compito in alcuni ambiti sono disponibili costrutti linguistici specializzati per la programmazione concorrente.

Nel contesto dei Pthread i costrutti specializzati per gestire la mutua esclusione si chiamano Mutex. Un Mutex è un blocco, ovvero un segnale di "occupato", che può essere attivato da un solo thread alla volta. Quando un thread attiva il blocco, se un altro thread cerca di attivarlo, quest'ultimo viene posto in attesa fino a quando il primo thread non rilascia il blocco.

Nella terminologia dei Pthread l'attivazione del blocco è chiamata "lock" e il rilascio è chiamato "unlock".

In figura 3.5 è mostrato un programma ottenuto aggiungendo al programma di figura 3.3 un Mutex in modo da garantire la mutua esclusione delle sequenze critiche (che in questo caso coincidono con tutta la thread function, vanificando in un certo senso lo scopo della concorrenza).

Una variabile di tipo pthread_mutex_t (*conti* nel programma) deve essere dichiarata (riga 3) e poi inizializzata tramite la funzione pthread_mutex_init (riga 27).

Successivamente il mutex può essere bloccato e sbloccato tramite le due funzioni pthread_mutex_lock e pthread_mutex_unlock. Quando un thread esegue l'operazione di lock se il Mutex è già bloccato il thread è posto in attesa che si sblocchi.

```
1.  //Trasferimento sincronizzato con mutex
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  //
5.  int contoA=100, contoB=200;
6.  int totale;
7.  pthread_mutex_t conti;                    //dichiarazione di un mutex
8.  //
9.  void * trasferisci(void *arg)
10. {
11. int importo=*((int*)arg);
12. int cA,cB;
13. pthread_mutex_lock(&conti); //inizio sequenza critica
14. cA = contoA; //leggi contoA in variabile locale
15. cB = contoB; //leggi contoB in variabile locale
16. contoA = cA + importo;
17. contoB = cB - importo;
18. pthread_mutex_unlock(&conti); //fine sequenza critica
19. return NULL;
20. }
21. //
22. int main(void)
23. {
24. pthread_t tID1;
25. pthread_t tID2;
26. int importo1=10, importo2=20;
27. pthread_mutex_init(&conti, NULL);          //inizializza mutex
28. pthread_create(&tID1, NULL, &trasferisci, &importo1);
29. pthread_create(&tID2, NULL, &trasferisci, &importo2);
30. pthread_join(tID1, NULL);
31. pthread_join(tID2, NULL);
32. totale=contoA + contoB;
33. printf("contoA = %d  contoB = %d\n", contoA, contoB);
34. printf("il totale è %d\n", totale);
35. return 0;
```

```
}
```

Figura 3.5

In figura 3.6 è mostrata una serie di esecuzioni di questo programma, che risultano tutte corrette.

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $
```

Figura 3.6

3.8 Sequenze critiche senza Mutex (ovvero implementazione Software del Mutex)

Abbiamo visto che il programma di trasferimento fondi può essere scritto correttamente tramite l'uso di un Mutex. In questo capitolo vogliamo analizzare se e come è possibile ottenere lo stesso risultato senza utilizzare i Mutex. I diversi tentativi che faremo per ottenere questo risultato costituiscono un esercizio di programmazione concorrente e mostrano alcuni dei problemi che insorgono in questo tipo di programmazione.

Inoltre, la possibilità di scrivere un programma che realizza la mutua esclusione dimostra che un Mutex può essere realizzato da software, tramite una normale libreria.

Per semplificare la discussione invece del programma di trasferimento dei fondi utilizziamo un programma che semplicemente entra ed esce da una sequenza critica. In figura 3.7 sono mostrati il testo del programma, che utilizza i Mutex, e il risultato di una sua esecuzione, corretto in quanto un solo thread alla volta entra nella sequenza critica; ci proponiamo quindi di realizzare lo stesso programma senza utilizzare i Mutex.

```
1.  //Sequenze critiche con mutex
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  //
5.  pthread_mutex_t mutex;                  //dichiarazione di un mutex
6.  //
7.  void * trasferisci(void *arg)
8.  {
9.  pthread_mutex_lock(&mutex);             //inizio sequenza critica
10. printf("thread %d: entro in sequenza critica\n", (int)arg);
11. printf("thread %d: termino sequenza critica\n", (int)arg);
12. pthread_mutex_unlock(&mutex);          //fine sequenza critica
13. return NULL;
14. }
15. //
16. int main(void)
17. {
18. pthread_t tID1;
19. pthread_t tID2;
20. pthread_mutex_init(&mutex, NULL);          //inizializza mutex
21. pthread_create(&tID1, NULL, &trasferisci, (void *)1);
22. pthread_create(&tID2, NULL, &trasferisci, (void *)2);
23. pthread_join(tID1, NULL);
24. pthread_join(tID2, NULL);
25. printf("fine\n");
26. return 0;
27. }
```

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
thread 1: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: entro in sequenza critica
thread 2: termino sequenza critica
fine
giuseppe@vmware-linux ~/thread_cap3 $
```

Figura 3.7

3.8.a Prima variante

Per sostituire i mutex nel programma di figura 3.8 abbiamo utilizzato semplicemente una variabile intera *blocca* e un ciclo di attesa secondo la seguente logica:

mutex_lock → while (blocca==1); blocca=1;

Il ciclo while (riga 9) mantiene il thread in un'attività inutile di continua ripetizione del test della condizione fino a quando la variabile blocca non diventa 0;

quando tale variabile diventa 0 il thread esce dal ciclo e pone blocca=1 (riga 10) per entrare nella sequenza critica.

Si osservi che un ciclo while come questo è detto di "**busy waiting**" (impegnato ad attendere), perchè dal punto di vista del sistema il programma è attivo ed esegue istruzioni, anche se la sua attività è totalmente inutile.

L'uscita dalla sequenza critica si basa sulla seguente traduzione dell'operazione di unlock ed è eseguita nella riga 13

mutex_unlock → blocca = 0;

Dato che la variabile blocca assume lo stesso significato del Mutex e il ciclo while mantiene il thread che desidera entrare in sequenza critica in attesa che tale variabile sia 0 in molti casi questo programma può fornire un risultato corretto, ma esistono alcune sequenze particolari di eventi che portano i due thread a violare la mutua esclusione, come mostrato dal risultato della sua esecuzione riportato in figura, dove i due thread entrano ambedue in sequenza critica.

Esercizio 6*. Determinare una sequenza di eventi che causa tale violazione.

Traccia di soluzione: si analizzino le possibili sequenze di esecuzione delle operazioni 9 e 10 da parte dei due thread ■

```
1.  //Sequenze critiche con variabile intera al posto del mutex
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  //
5.  int blocca=0;
6.  //
7.  void * trasferisci(void *arg)
8.  {
9.  while (blocca==1);         //busy waiting
10. blocca=1;  //inizio sequenza critica
11. printf("thread %d: entro in sequenza critica\n", (int)arg);
12. printf("thread %d: termino sequenza critica\n", (int)arg);
13. blocca=0; //fine sequenza critica
14. return NULL;
15. }
16. //
17. int main(void)
18. {
19. pthread_t tID1;
20. pthread_t tID2;
21. pthread_create(&tID1, NULL, &trasferisci, (void *)1);
22. pthread_create(&tID2, NULL, &trasferisci, (void *)2);
23. pthread_join(tID1, NULL);
24. pthread_join(tID2, NULL);
25. printf("fine\n");
26. return 0;
27. }
```

```
thread 1: entro in sequenza critica
thread 2: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: termino sequenza critica
fine
```

Figura 3.8 – Mutua esclusione; versione 1 - errata

3.8.b Seconda versione

Per cercare di correggere il problema evidenziato nella versione precedente proviamo a bloccare la sequenza prima di entrare in attesa, come mostrato nel programma di figura 3.9. Il programma utilizza due variabili blocca1 e blocca2; bloccaX indica l'intenzione del thread X di entrare in sequenza critica. Per svolgere un'azione diversa in base al thread che la esegue, la funzione trasferisci riceve il numero di thread (non il tID) come argomento.

Tentiamo di dimostrare con un ragionamento strutturato che questa soluzione garantisce la mutua esclusione.

1) Se t1 entra in sequenza critica (riga 14) prima che t2 arrivi alla riga 13, allora t2 verrà bloccato nel ciclo di attesa di riga 13 fino a quando t1 non uscirà dalla sequenza critica → in queso caso la mutua esclusione è rispettata;

2) Se t1 entra in sequenza critica (riga 14), allora deve avere terminato il test della riga 11 (ciclo di attesa su blocca2) prima che t2 abbia eseguito l'assegnamento della riga 12 (blocca2=1), cioè deve essere t1.11 < t2.12;

→ t2 non può avere terminato la riga 12 prima che t1 entri in sequenza critica,

→ la condizione indicata al punto precedente (t2 non è ancora arrivato alla riga 13) è vera,

→ se t1 entra in sequenza critica t2 sicuramente non può entrare in sequenza critica

3) Per simmetria si può ripetere il ragionamento scambiando t1 con t2 → se t2 entra in sequenza critica t1 non può entrare in sequenza critica

Quindi questa soluzione garantisce la mutua esclusione.

Tuttavia questa soluzione, pur garantendo la mutua esclusione, causa in presenza di alcune sequenza di eventi un problema così grave da renderla inapplicabile; tale problema è noto con il nome di **deadlock** o stallo. ***Il deadlock tra due thread consiste in un'attesa reciproca infinita, per cui t1 attende che t2 sblocchi la risorsa critica e viceversa, ed essendo ambedue i thread bloccati, nessuno dei due potrà attuare l'azione che sblocca l'altro.***

Esercizio 7*. Determinare una sequenza di esecuzione che causa un deadlock.

Traccia. Trovare una sequenza che ponga ambedue i thread nel ciclo di attesa■

Il deadlock, una volta verificatosi, è permanente; in effetti, l'esecuzione di questa versione del programma è andata in deadlock e per riprendere il controllo del terminale è stato necessario far eliminare il processo con un comando dato al sistema operativo da un altro terminale; per questo motivo il risultato dell'esecuzione mostrato in figura è costituito solamente dalla scritta "terminated":

Il problema del deadlock verrà ripreso in consisderazione e analizzato più a fondo nel seguito.

```
1.  //Sequenze critiche: Mutua esclusione con Deadlock
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  //
5.  int blocca1=0;
6.  int blocca2=0;
7.  //
8.  void * trasferisci(void *arg)
9.  {
10. if ((int)arg==1){blocca1=1;
11. .                   while (blocca2==1);}
12. if ((int)arg==2){blocca2=1;
13. .                   while (blocca1==1);}
14. //inizio sequenza critica
15. printf("thread %d: entro in sequenza critica\n", (int)arg);
16. printf("thread %d: termino sequenza critica\n", (int)arg);
17. if ((int)arg==1){blocca1=0; }
18. if ((int)arg==2){blocca2=0; }
19. //fine sequenza critica
20. return NULL;
21. }
22. //
23. int main(void)
24. {
25. pthread_t tID1, tID2, tID3;
26. pthread_create(&tID1, NULL, &trasferisci, (void *)1);
27. pthread_create(&tID2, NULL, &trasferisci, (void *)2);
28. pthread_join(tID1, NULL);
29. pthread_join(tID2, NULL);
30. printf("fine\n");
31. return 0;
32. }
```

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
Terminated
```

Figura 3.9 - Mutua esclusione; versione 2 - Deadlock

3.8.c Terza versione: realizzazione corretta della mutua esclusione

In figura 3.10 è mostrato un programma che realizza correttamente la mutua esclusione (algoritmo di Petersen). L'idea base consiste nell'estendere la soluzione precedente, che garantiva la mutua esclusione, eliminando il rischio di deadlock. Per eliminare il rischio di deadlock questo programma contiene una nuova variabile

intera, *favorito* (dichiarata in riga 5), il cui scopo è di garantire che non possano restare nel ciclo di attesa ambedue i thread.

La condizione di attesa di t1 (riga 14) è stata estesa controllando che non solo l'altro thread abbia impostato blocca2=1, ma anche che l'altro thread sia favorito. In sostanza la condizione di attesa è stata indebolita. Proviamo a dimostrare la correttezza di questa soluzione con un ragionamento strutturato:

1) Il deadlock non può sicuramente verificarsi, perchè la variabile favorito è unica e quindi il test su tale variabile deve essere falso in uno dei due cicli di attesa delle righe 14 e 18;

2) Dobbiamo però dimostrare che questo indebolimento della condizione di attesa non porti a violare la mutua esclusione; diamo per acquisito il ragionamento fatto sul programma precedente e quindi ci limitiamo a dimostrare che se t1 esce dall'attesa perchè favorito==1 (cioè è falsa la seconda parte della condizione di riga 14), allora t2 non può essere nella sequenza critica:

2.1) se t1 arriva a riga 19 → blocca1==1;

2.2) se t1 arriva a riga 19 → blocca2==0 oppure favorito==1

2.2.1) se blocca2==0 → t2 non ha ancora raggiunto la riga 16 (e valgono gli stessi ragionamenti del caso precedente)

2.2.2) se blocca2==1 deve essere favorito==1

2.3) se t1 arriva a riga 19 e favorito==1 deve essersi verificato t1.14 < t2.17 (< t2.18)

2.4) quindi t2 è entrato nel ciclo 18 con blocca1==1 && favorito==1, impostati da t1.12 e t1.13,

2.5) quindi quando t1 entra nella sequenza critica, t2 non può avere superato il ciclo di attesa 18; per simmetria la mutua esclusione è sempre garantita.

```
1.  //mutua esclusione corretta
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  //
5.  int favorito =1;
6.  int blocca1=0;
7.  int blocca2=0;
8.  //
9.  void * trasferisci(void *arg)
10. {
11. if ((int)arg==1){
12. .                   blocca1=1;
13. .                   favorito=2;
14. .                    while (blocca2==1 && favorito ==2);}
15. if ((int)arg==2){
16. .                    blocca2=1;
17. .                    favorito=1;
18. .                     while (blocca1==1 && favorito ==1);}
19. //inizio sequenza critica
20. printf("thread %d: entro in sequenza critica\n", (int)arg);
21. printf("thread %d: termino sequenza critica\n", (int)arg);
22. if ((int)arg==1){blocca1=0; }
23. if ((int)arg==2){blocca2=0; }
24. //fine sequenza critica
25. return NULL;
26. }
27. //
28. int main(void)
29. {
30. pthread_t tID1;
31. pthread_t tID2;
32. pthread_create(&tID1, NULL, &trasferisci, (void *)1);
33. pthread_create(&tID2, NULL, &trasferisci, (void *)2);
34. pthread_join(tID1, NULL);
35. pthread_join(tID2, NULL);
36. printf("fine\n");
37. return 0;
38. }
```

```
thread 1: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: entro in sequenza critica
thread 2: termino sequenza critica
fine
giuseppe@vmware-linux ~/thread cap3 $
```

Figura 3.10 - Mutua esclusione; versione 3 - corretta

Istruzioni atomiche a livello Hardware

I processori moderni possiedono generalmente alcune istruzioni che rendono più semplice realizzare un Mutex. Si rimanda a questo proposito al capitolo 2.11 (pag. 105 e seguenti) del testo Patterson-Hennessy.

3.9 Deadlock

In un esempio del capitolo precedente abbiamo già incontrato il problema del deadlock. Il deadlock è una situazione di attesa circolare, nella quale un certo numero di attività si trovano in stato di attesa reciproca, e nessuna può proseguire.

La situazione più elementare di deadlock si crea, come visto nel paragrafo precedente, quando due thread t1 e t2 bloccano due risorse A e B e raggiungono una situazione nella quale t1 ha bloccato A e attende di poter bloccare B mentre t2 ha bloccato B e attende di poter bloccare A.

Un programma che opera in questo modo è mostrato in figura 3.11. Il blocco delle risorse A e B è rappresentato dai due mutex *mutexA* e *mutexB* dichiarati in riga 4. Il main( ) attiva 2 thread t1 e t2 i quali eseguono le due funzioni *lockApoiB( )* e *lockBpoiA( )*. Queste due funzioni eseguono dei lock progressivamente sui due mutex A e B, ma procedono in ordine inverso.

E' evidente che un deadlock può verificarsi se t1 arriva a bloccare mutexA e t2 riesce a bloccare mutexB prima che t1 blocchi mutexB.

Esercizio 8*. Determinare una sequenza che porta al deadlock■

In figura 3.12 è mostrato il risultato di una serie di esecuzioni di questo programma; come si vede, in una escuzione le cose sono andate bene, perchè t1 è riuscito a bloccare ambedue i mutex prima di t2, ma in altre esecuzioni si è arrivati a un deadlock ed è stato necessario terminare il programma dall'esterno.

```
1.  #include <pthread.h>
2.  #include <stdio.h>
3.  //
4.  pthread_mutex_t mutexA, mutexB;
5.  //
6.  void * lockApoiB (void *arg)
7.  {
8.  pthread_mutex_lock(&mutexA);                  //inizio sequenza critica 1
9.  printf("thread %d: entro in sequenza critica 1\n", (int)arg);
10. pthread_mutex_lock(&mutexB);                  //inizio sequenza critica 2
11. printf("thread %d: entro in sequenza critica 2\n", (int)arg);
12. printf("thread %d: termino sequenza critica 2\n", (int)arg);
13. pthread_mutex_unlock(&mutexB);                //fine sequenza critica 2
14. printf("thread %d: termino sequenza critica 1\n", (int)arg);
15. pthread_mutex_unlock(&mutexA);                //fine sequenza critica 1
16. return NULL;
17. }
18. void * lockBpoiA (void *arg)
19. {
20. pthread_mutex_lock(&mutexB);                  //inizio sequenza critica 2
21. printf("thread %d: entro in sequenza critica 2\n", (int)arg);
22. pthread_mutex_lock(&mutexA);                  //inizio sequenza critica 1
23. printf("thread %d: entro in sequenza critica 1\n", (int)arg);
24. printf("thread %d: termino sequenza critica 1\n", (int)arg);
25. pthread_mutex_unlock(&mutexA);                //fine sequenza critica 1
26. printf("thread %d: termino sequenza critica 2\n", (int)arg);
27. pthread_mutex_unlock(&mutexB);                //fine sequenza critica 2
28. return NULL;
29. }
30. int main(void)
31. {
32. pthread_t tID1;
33. pthread_t tID2;
34. pthread_mutex_init(&mutexA, NULL);
35. pthread_mutex_init(&mutexB, NULL);
36. pthread_create(&tID1, NULL, &lockApoiB, (void *)1);
37. pthread_create(&tID2, NULL, &lockBpoiA, (void *)2);
38. pthread_join(tID1, NULL);
39. pthread_join(tID2, NULL);
40. printf("fine\n");
41. return 0;
42. }
```

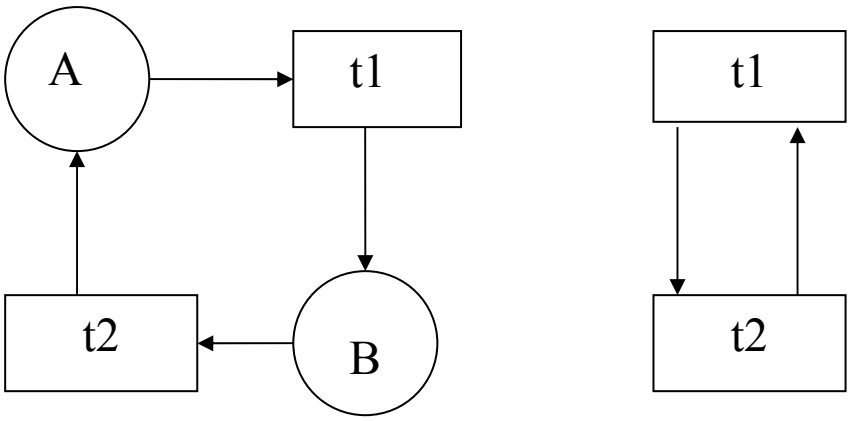
Figura 3.11 – Deadlock causato dai Mutex

```
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 1: entro in sequenza critica 2
thread 1: termino sequenza critica 2
thread 1: termino sequenza critica 1
thread 2: entro in sequenza critica 2
thread 2: entro in sequenza critica 1
thread 2: termino sequenza critica 1
thread 2: termino sequenza critica 2
fine
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ gcc -pthread
CritSecMutex12deadlock.c
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 2: entro in sequenza critica 2
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated
```

Figura 3.12

Una situazione di questo genere può essere rappresentata con un grafo di accesso alle risorse come quello di figura 3.13.a, da interpretare nel modo seguente:

- i rettangoli rappresentano le attività o thread,
- i cerchi rappresentano le risorse (ovvero i mutex o sequenze critiche),
- una freccia da un thread a una risorsa indica che il thread è in attesa di bloccare la risorsa
- una freccia da una risorsa a un thread indica che la risorsa è stata bloccata dal thread



(a) ***(b)***

Figura 3.13

Possiamo semplificare il grafo di accesso alle risorse sostituendo la sequenza "ti richiede la risorsa X bloccata da tj" con un'unica freccia che interpretiamo come "ti attende tj" e otteniamo un **grafo di attesa** come quello di figura 3.13.b.

La situazione di deadlock è rappresentata dall'esistenza di un ciclo in un grafo di attesa.

Le situazioni di deadlock possono coinvolgere più di 2 thread, come mostrato dal grafo di attesa di figura 3.14, dove esiste un ciclo che coinvolge 4 thread.

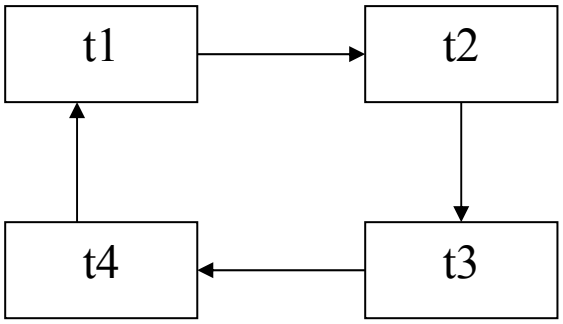


Figura 3.14

Nella scrittura di programmi concorrenti è necessario tener conto del rischio di deadlock e prevenire la posibilità che si verifichi. In base all'esempio precedente, utilizzando i mutex il rischio di deadlock esiste se:

- due o più thread bloccano più di un mutex
- l'ordine in cui i thread bloccano i mutex è diverso

Esercizio 9*. Si mostri con un ragionamento strutturato che, se due thread acquisiscono due o più mutex procedendo nello stesso ordine non può verificarsi un deadlock■

Come abbiamo visto nell'esempio di figura 3.9, si può verificare un deadlock anche senza utilizzare i mutex, semplicemente a causa di cicli di attesa su normali variabili. Nell'esempio di figura 3.10 abbiamo visto una soluzione che ha permesso di prevenire la formazione di deadlock in un programma che presentava questo rischio.

3.10 Sincronizzazione e semafori

Talvolta un thread deve attendere che un altro thread abbia raggiunto un certo punto di esecuzione prima di procedere.

Ad esempio, si consideri il primo programma trattato in questo capitolo (figura 3.1), che stampa una sequenza di lettere in un ordine non determinabile a priori e si supponga di voler produrre esattamente la sequenza "axbycz" senza modificare la struttura dei due thread; un modo per ottenere questo risultato può essere il seguente: ogni thread attende, prima di stampare una lettera, che l'altro thread abbia stampato la lettera precedente. Lo strumento per realizzare questo tipo di sincronizzazione tra due thread è costiuito dai semafori, un costrutto di supporto alla programmazione concorrente molto generale.

Un semaforo è una variabile intera sulla quale si può operare solamente nel modo seguente:

1. Un semaforo viene inizializzato con un valore e successivamente può essere solamente incrementato o decrementato di 1 (come un contatore); nella libreria pthread il tipo di un semaforo è *sem_t* e l'operazione di inizializzazione è *sem_init(...)*
2. Per decrementare il semaforo si utilizza l'operazione *sem_wait(...)*; se il valore è già zero, il thread tw che esegue l'operazione wait viene bloccato fino a quando un altro thread eseguirà un'operazione di incremento (a quel punto la wait si sblocca e il thread tw prosegue)
3. Per incrementare il semaforo si utilizza l'operazione *sem_post(...);* se ci sono altri thread in attesa sullo stesso semaforo, uno di questi viene sbloccato e può procedere (e il semaforo viene decrementato nuovamente)

Il valore di un semaforo può essere interpretato nel modo seguente:

1 se il valore è positivo, rappresenta il numero di thread che lo possono decrementare senza andare in attesa;

2 se è negativo, rappresenta il numero di thread che sono bloccati in attesa;

3 se è 0 indica che non ci sono thread in attesa, ma il prossimo thread che eseguirà una wait andrà in attesa.

L'uso dei semafori per ottenere la stampa della stringa "axbycz" è illustrato nel programma di figura 3.15. La sintassi delle operazioni sui semafori è molto semplice e autoesplicativa a parte la funzione *sem_init( )*, nella quale il secondo parametro è sempre 0 e il terzo indica il valore iniziale da assegnare al semaforo.

Nel programma i semafori sono ambedue inizializzati a 0, cioè tali da bloccare i thread che eseguono wait su di loro. Il primo thread, che esegue tf1, esegue subito una sem_wait e si blocca, mentre il secondo thread, che esegue tf2, stampa una 'a', esegue una sem_post sul semaforo 1 per sbloccare il primo thread e poi esegue una sem_wait e si blocca a sua volta..

L'unico carattere che può essere stampato dopo la 'a' è quindi la 'x', perchè il primo thread è stato sbloccato, mentre il secondo è bloccato.

L'alternanza dei due thread procede secondo questo schema.

Nella stessa figura sono mostrate due esecuzioni del programma che producono ambedue la sequenza desiderata.

```
1       #include <pthread.h>
2       #include <stdio.h>
3       #include <semaphore.h>
4       //
5       sem_t sem1;
6       sem_t sem2;
7       void * tf1(void *arg)
8       {
9       sem_wait(&sem1);  printf("x"); sem_post(&sem2);
10      sem_wait(&sem1);  printf("y");  sem_post(&sem2);
11      sem_wait(&sem1);  printf("z");
12      return NULL;
13      }
14      void * tf2(void *arg)
15      {
16      printf("a");  sem_post(&sem1);
17      sem_wait(&sem2);  printf("b");  sem_post(&sem1);
18      sem_wait(&sem2);   printf("c");  sem_post(&sem1);
19      return NULL;
20      }
21      int main(void)
22      {
23      pthread_t tID1;
24      pthread_t tID2;
25      sem_init(&sem1,0,0);
26      sem_init(&sem2,0,0);
27      pthread_create(&tID1, NULL, &tf1, NULL);
28      pthread_create(&tID2, NULL, &tf2, NULL);
29      pthread_join(tID1, NULL);
30      pthread_join(tID2, NULL);
31      printf("\nfine\n");
32      return 0;
33      }
```

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $
```

Figura 3.15

Vale la pena di osservare che un semaforo inizializzato al valore 1 può essere utilizzato esattamente come un mutex. Se un thread vuole entrare in una sequenza critica, fa un'operazione di wait sul semaforo. Se nessun altro thread è nella sequenza critica, il semaforo avrà il suo valore 1 e verrà decrementato a 0, il thread potrà

proseguire entrando nella sequenza critica mentre qualsiasi altro thread verrà bloccato al momento della wait.

Quando il thread termina la sequenza critica esegue una post e questa operazione sblocca un altro thread già in attesa oppure riporta il semaforo al suo stato iniziale, che indica che la sequenza critica è libera.

Tuttavia un semaforo è uno strumento più generale di un mutex, perchè può essere utilizzato anche con valori maggiori di 1. Ad esempio, si supponga che esista un certo numero R di risorse utilizzabili dai thread (si pensi a un insieme di buffer) e di inizializzare un semaforo S al valore R. In questo caso ogni thread eseguirà una wait su S prima di allocarsi una risorsa e una post su S quando rilascia la risorsa: i primi R thread si allocheranno le risorse ed eventuali thread aggiuntivi resteranno in attesa che le risorse vengano rilasciate.

Un esempio classico di uso dei semafori è costituito da un buffer nel quale uno o più thread produttori scrivono caratteri e uno o più thread consumatori leggono caratteri.

Una versione molto semplice di questo problema è quella affrontata dal programma di figura 3.16: il buffer contiene un solo carattere, un thread vi scrive in sequenza un certo numero di lettere dell'alfabeto e l'altro thread le legge e le stampa.

Il semaforo *pieno* blocca il thread produttore fino a quando il thread consumatore non ha prelevato il carattere; il semaforo *vuoto* blocca il consumatore fino a quando il produttore non ha scritto il carattere. Quando il primo thread ha finito pone la variabile *fine* a 1 e l'altro thread esce dal ciclo di lettura e stampa.

Il programma contiene però un errore di sincronizzazione, come si vede dalla esecuzione rappresentata nella stessa figura; infatti il thread di scrittura scrive 5 caratteri e il thread di lettura ne stampa solamente 4.

Esercizio 10*. Determinare una sequenza di eventi che conduce alla situazione di errore e una sequenza, se esiste, che conduce alla stampa di tutti i caratteri■

```
1.  //produttore consumatore con semafori –versione con errore
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  #include <semaphore.h>
5.  //
6.  char buffer;
7.  int fine = 0;
8.  sem_t pieno;
9.  sem_t vuoto;
10. //
11. void * scrivi(void *arg)
12. {
13.       int i;
14.       for (i=0; i<5; i++)
15.            { sem_wait(&vuoto);
16.              buffer= 'a' +i;
17.              sem_post(&pieno);
18.            }
19. fine = 1;
20. return NULL;
21. }
22. void * leggi(void *arg)
23. {
24. char miobuffer;
25. int i;
26. while (fine == 0)
27. {
28. sem_wait(&pieno);
29. miobuffer=buffer;
30. sem_post(&vuoto);
31. printf("il buffer contiene %.1s \n", &miobuffer);
32. }
33. return NULL;
34. }
35. int main(void)
36. {
37. pthread_t tID1;
38. pthread_t tID2;
39. sem_init(&pieno,0,0);
40. sem_init(&vuoto,0,1); //il buffer inizialmente è vuoto
41. pthread_create(&tID1, NULL, &scrivi, NULL);
42. pthread_create(&tID2, NULL, &leggi, NULL);
43. pthread_join(tID1, NULL);
44. pthread_join(tID2, NULL);
45. printf("fine\n");
46. return 0;
47. }
```

```
$ ./a.out
il buffer contiene a
il buffer contiene b
il buffer contiene c
il buffer contiene d
fine
```

Figura 3.16

L'errore è stato eliminato nel programma di figura 3.17 (esecuzione in figura 3.18), nel quale sono state aggiunte le righe 33-37 alla funzione leggi eseguita dal consumatore dopo essere uscito dal ciclo; tali istruzioni verificano se il semaforo *pieno* vale ancora 1 e in tal caso prelevano l'ultimo carattere. La funzione di libreria *sem_getvalue(...)* presente in riga 33 restituisce il valore del semaforo pieno nella variabile i.

Osservazione: Può sorprendere l'uso di due semafori nell'esempio precedente, perchè una sola variabile *pieno/vuoto* sarebbe sufficiente a indicare se il buffer è pieno o vuoto; tuttavia se si pensa che devono esistere due stati di attesa, uno del produttore e uno del consumatore, l'esigenza risulta più evidente.

```
1.  //produttore consumatore con semafori
2.  #include <pthread.h>
3.  #include <stdio.h>
4.  #include <semaphore.h>
5.  //
6.  char buffer;
7.  int fine = 0;
8.  sem_t pieno;
9.  sem_t vuoto;
10. //
11. void * scrivi(void *arg)
12. {
13. int i;
14. for (i=0; i<5; i++)
15. { sem_wait(&vuoto);
16. buffer= 'a' +i;
17. sem_post(&pieno);
18. }
19. fine = 1;
20. return NULL;
21. }
22. void * leggi(void *arg)
23. {
24. char miobuffer;
25. int i;
26. while (fine == 0)
27. {
28. sem_wait(&pieno);
29. miobuffer=buffer;
30. sem_post(&vuoto);
31. printf("il buffer contiene %.1s \n", &miobuffer);
32. }
33. sem_getvalue(&pieno, &i);                    //controllo per l'ultimo carattere
34. if (i==1)
35. {
36. miobuffer=buffer;
37. printf("il buffer contiene %.1s \n", &miobuffer);
38. }
39. return NULL;
40. }
41. int main(void)
42. {
43. pthread_t tID1;
44. pthread_t tID2;
45. sem_init(&pieno,0,0);
46. sem_init(&vuoto,0,1); //il buffer inizialmente è vuoto
47. pthread_create(&tID1, NULL, &scrivi, NULL);
48. pthread_create(&tID2, NULL, &leggi, NULL);
49. pthread_join(tID1, NULL);
50. pthread_join(tID2, NULL);
51. printf("fine\n");
52. return 0;
53. }
```

Figura 3.17

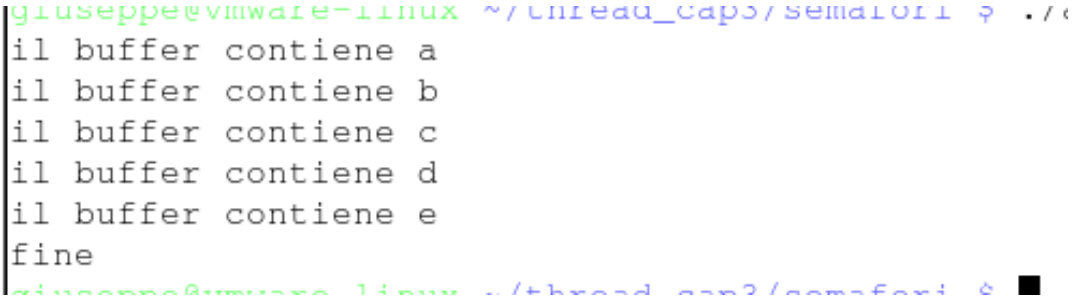

il buffer contiene a
il buffer contiene b
il buffer contiene c
il buffer contiene d
il buffer contiene e
fine

Figura 3.18

3.11 Esercizi e domande finali

Esercizio X

Si consideri il seguente problema (problema noto come rendezvous): dati 2 thread che svolgono due operazioni x e y ciascuno, si vuole garantire tramite semafori che

t1.x < t2.y e t2.x < t1.y

I due programmi seguenti vogliono risolvere il problema utilizzando due semafori A e B. Dire se sono corretti; in caso contrario dimostrare qual'è l'errore.

Programma 1, costituito da due thread t1 e t2:	
t1.1 (x)	t2.1 (x)
t1.2 (wait B)	t2.2 (wait A)
t1.3 (post A)	t2.3 (post B)
t1.4 (y)	t2.4 (y)

Programma 2, costituito da due thread t1 e t2:	
t1.1 (x)	t2.1 (x)
t1.2 (post A)	t2.2 (post B)
t1.3 (wait B)	t2.3 (wait A)
t1.4 (y)	t2.4 (y)

Esercizio Y*

Si consideri il seguente problema (problema della barriera): un numero T di thread deve sincronizzarsi in un punto, cioè ogni thread deve arrivare a un punto (punto di sincronizzazione) dove deve attendere che tutti gli altri thread siano arrivati al loro punto di sincronizzazione.

Una prima soluzione (sbagliata) consiste nell'uso di un semaforo A e un contatore, applicata da tutti i thread nel modo seguente:

programma 1 (eseguito da ogni thread)

```
1    conta++;
2    if (conta==T) post A;
3    wait A;
```

Questo programma contiene 2 errori gravi: individuarli e correggerli■

3.12 Soluzioni di alcuni esercizi

Soluzione 3.

Il primo, il quarto e il sesto risultato sono ugualli a quello prodotto dalla sequenza S3 analizzata nel testo (sostituendo i numeri di riga 12, 13, 14 e 15 alle 4 operazioni 1, 2, 3 e 4)

Il secondo e il quinto sono corretti, prodotti ad esempio dalle sequenza S1 o S2 del testo.

Il terzo risultato è prodotto da una sequenza come la seguente:

t2.12 < t2.13 < t2.14 < t1.12 < t1.13 < t1.14 < t1.15 < t2.15

Soluzione 5.

Ipotizziamo che i++ sia tradotta dal compilatore con le tre istruzioni seguenti:

1. Move i, R (poni il valore di i nel registro R)
2. Incr R (incrementa R)
3. Move R, i (poni il valore di R in i)

La seguente sequenza di esecuzione produce un solo incremento (anche se il sistema salva il valore di R quando passa dall'esecuzione di un thread all'altro)

t1.1 < t2.1 < t2.2 < t2.3 < t1.2 < t1.3

Soluzione 6.

Una sequenza possibile che causa l'errore è la seguente (ne esistono anche altre):

t1.9 (condizione falsa, quindi procedi) <

t2.9 (condizione falsa, quindi procedi) < t1.10 < t2.10■

Soluzione 7.

Una sequenza possibile che causa il deadlock è la seguente (ne esistono anche altre):

t1.10 < t2.12 < t1.11 < t2.13

Soluzione 8.

Una sequenza possibile che causa il deadlock è la seguente (ne esistono anche altre):

t1.8 (t1 blocca A) < t2.20 (t2 blocca B) < t1.10 (t1 va in attesa di B) < t2.22 (t2 va in attesa di A)

Soluzione 9.

Supponiamo che i mutex siano A e B; 2 thread eseguono le operazioni

1) ti.lockA < 2) ti.lockB

ambedue nello stesso ordine.

Supponiamo che venga eseguito per primo t1.1

A questo punto, se t2 esegue t2.1, t2 va in attesa

Prima o poi t1 eseguirà t1.2 e sarà quindi in grado di concludere. Per simmetria vale lo stesso ragionamento a parti scambiate se viene eseguito per primo t2.1.

Il ragionamento rimane valido se i lock sono più di due, purchè sia soddisfatta la condizione di acquisirli nello stesso ordine.

Soluzione 10.

Una sequenza che provoca l'errore di non stampare l'ultimo carattere è la seguente:

Consideriamo il seguente stato iniziale:

t2 sta eseguendo la stampa 31 relativa al carattere 'd' (penultimo carattere) → t2 ha già eseguito il post su vuoto relativo al carattere 'd'

t1 è all'ultimo ciclo, i==4,

A questo punto si verifica la sequenza.

t1.15 (t1 esce dal wait su vuoto) < t1.16 (t1 pone buffer='e') < t1.17 (post pieno) < t1.19 (fine=1) < t2.26 (t2 testa la condizione di fine trovandola vera) < t2 termina senza processare il carattere 'e'

Talvolta lo stesso programma può stampare tutti i caratteri; consideriamo lo stesso stato iniziale precedente, ma il seguente ordine di esecuzione:

t2.26 (t2 testa la condizione di fine trovandola falsa, quindi entra nel ciclo) < t2.28 (t2 esegue wait su pieno) < t1.15 (t1 esce dal wait su vuoto) < t1.16 (t1 pone buffer='e') < t1.17 (post pieno) < t1.19 (fine=1) < t2.29 < t2.30 < t2.31 (t2 stampa 'e')

Soluzione Y

Il primo errore consiste nella mancanza di protezione del contatore durante l'aggiornamento. Questo errore può essere corretto introducendo un mutex M.

Il secondo errore è il deadlock. Supponiamo T = 10. Dopo l'arrivo di 9 thread il valore di conta sarà 9 e quello del semaforo A sarà -9; a questo punto arriva l'ultimo thread, conta diventa 10, viene eseguito un post, un thread in attesa viene sbloccato e il semaforo viene incrementato a -8. Tutti gl ialtri thread restano bloccati.

Per correggere questi errori il programma deve diventare

```
1    mutex lock M;   conta++;    mutex_unlock M;
2    if (conta==T) post A;
3    wait A;
4    post A
```

In questo modo i primi 9 thread vanno in attesa, l'ultimo sblocca un thread, che parte ed esegue il post (riga 4), sbloccandone un altro, che esegue il post, e così via fino a sbloccare tutti i thread.