AXO

Architettura dei Calcolatori e Sistemi Operativi

programmazione concorrente

programmazione concorrente – 1

- modello di esecuzione sequenziale
  - istruzioni eseguite in ordine predeterminabile
  - in base al codice del programma e dei dati in ingresso
- modello di programmazione concorrente
  - flussi di controllo (sequenze di istruzioni) eseguiti in parallelo che interferiscono tra loro scambiandosi informazione
  - tipicamente i flussi condividono lo spazio di indirizzamento di memoria (per esempio condividono le strutture dati)

problemi di correttezza dell'esecuzione

programmazione concorrente – 2

- comportamento non-deterministico dell'esecuzione
  - impossibile prevedere la prossima istruzione eseguita
  - neppure valutando la dipendenza dai dati di ingresso
- esecuzione del singolo processo: deterministica
- esecuzione di processi indipendenti: deterministica
- esecuzione di un processo con più flussi di contollo (multi-threading): non-deterministica (quanto meno potenzialmente)

il non-determinismo dà luogo a errori di esecuzione

programmazione concorrente – esempio

```
1.  #include <pthread.h>
2.  #include <stdio.h>

3.  void * tf1 (void * arg) {
4.      printf ("x");
5.      printf ("y");
6.      printf ("z");
7.      return NULL;
8.  } /* tf1 */

9.  void * tf2 (void * arg) {
10.     printf ("a");
11.     printf ("b");
12.     printf ("c");
13.     return NULL;
14. } /* tf2 */
```

```
15. pthread_t tID1, tID2;

16. void main ( ) {
17.     pthread_create (
            &tID1, NULL,
            tf1, NULL
        );
18.     pthread_create (
            &tID2, NULL,
            tf2, NULL
        );
19.     pthread_join (tID1, NULL);
20.     pthread_join (tID2, NULL);
21.     printf ("\nfine\n");
22. } /* main */
```

programmazione concorrente – esempio

```
giuseppe@vmware-linux ~/thread_cap3 $ gcc -pthread xyzabcNOsinc.c
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbyzc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $
```

test del programma concorrente

- si verifica l'uscita del programma in funzione dell'ingresso (relazione deterministica tra ingresso e uscita)
- ma la verifica non garantisce la correttezza del programma
- questo tipo di test è uno strumento efficace per verificare la correttezza del programma nell'esecuzione sequenziale
- in un programma concorrente la relazione tra ingresso e uscita dipende
  - non solo dai dati in ingresso
  - ma anche dall'ordine esecuzione

  che come visto non è deterministico

correttezza del programma concorrente

- dimostrazione formale di correttezza
  basata su logica formale – DIFFICILE
- ragionamenti di tipo strutturato
- tecniche / costrutti di programmazione *ad hoc* per risolvere
  - sequenze critiche (mutua esclusione)
  - istruzioni atomiche (mutua esclusione)
  - deadlock o stallo (ordinamento o semaforo)
  - sincronizzazione (semaforo)

sequenza critica – 1

- sequenza = successione sequenziale di operazioni / istruzioni eseguite da un thread

  ti.*j* = istruzione *j* eseguita dal thread *i*

- relazione temporale tra istruzioni
  - dello stesso thread — *ti*.1 < *ti*.2 < … < *ti*.*n*
  - di thread diversi (p. es. due) — *t*1.1 < *t*1.2 < *t*2.1 < *t*1.3 < … < *ti*.*n*

- sequenza critica = successione di istruzioni che può essere eseguita da due o più thread in parallelo, le quali istruzioni però non devono essere mescolate (intercalate) tra loro, onde assicurare che il risultato dell'esecuzione sia sempre corretto

sequenza critica – 2

- per garantire la proprietà definita al punto precedente, è necessario eseguire in sequenza (sequenzializzare) la sequenza critica tra i vari thread, cioè:
  - se un qualsiasi thread comincia l'esecuzione della sequenza
  - tutta la sequenza deve essere eseguita da quel thread
  - prima di essere eseguita da un qualsiasi altro thread (*si riduce il livello di concorrenza*)
- la **mutua esclusione** è la proprietà che si vuole garantire per l'esecuzione concorrente delle sequenze critiche, affinché il risultato finale sia sempre corretto

sequenza critica – esempio 1a

```
contoA = contoA + importo
contoB = contoB – importo
```

1. memorizza `contoA` in una variabile locale **`cA`**
2. memorizza `contoB` in una variabile locale **`cB`**
3. aggiorna `contoA` al nuovo valore `cA` + `importo`
4. aggiorna `contoB` al nuovo valore `cB` – `importo`

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

sequenza critica – esempio 1a

		dopo							
	inizio	*t*1.1	*t*1.2	*t*1.3	*t*2.1	*t*2.2	*t*2.3	*t*2.4	*t*1.4
contoA	100			110			130		
contoB	200							180	190
cA (in *t*1)		100							
cB (in *t*1)			200						
cA (in *t*2)					110				
cB (in *t*2)						200			

t1 trasferisce 10 da contoA a contoB
t2 trasferisce 20 da contoA a contoB

t2 memorizza contoB in cB prima che contoB sia stato aggiornato a 190 da parte di t1

sequenza critica – esempio 2a

```
1.  #include <pthread.h> <stdio.h>
2.  int contoA = 100;
3.  int contoB = 200;
4.  int totale;

5.  void * trasferisci (void * arg) {
6.      int importo = *arg;
7.      int cA, cB;
8.      // inizio sequenza critica
9.      // leggi contoA in var locale
10.     cA = contoA;
11.     // leggi contoB in var locale
12.     cB = contoB;
13.     contoA = cA + importo;
14.     contoB = cB – importo;
15.     // fine sequenza critica
16.     return NULL;
17. } /* importo */
```

```
17. pthread_t tID1, tID2;

18. void main ( ) {
19.     int importo1 = 10;
20.     int importo2 = 20;
21.     pthread_create (
            &tID1, NULL,
            trasferisci, &importo1
        );
22.     pthread_create (
            &tID2, NULL,
            trasferisci, &importo2
        );
23.     pthread_join (tID1, NULL);
24.     pthread_join (tID2, NULL);
25.     totale = contoA + contoB;
26.     printf (
            “contoA = %d contoB = %d\n”,
            contoA, contoB
        );
27.     printf (“il totale è %d\n”, totale);
28. } /* main */
```

sequenza critica – esempio 2a

```
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
```

istruzione atomica – 1

- un'istruzione atomica è considerata indivisibile (non interrompibile) durante l'esecuzione
- per l'esempio, servirebbe che sia *a* sia *b* fossero atomiche
  a. `contoA = contoA + importo`
  b. `contoB = contoB – importo`
- se le istruzioni sono atomiche, qualsiasi loro ordinamento relativo ai due thread produce sempre un risultato corretto
  - $t1 < t2$ oppure $t2 < t1$ ovvio !!! ma anche
  - $t1.a < t2.a < t2.b < t1.b$ e
  - $t1.a < t2.a < t1.b < t2.b$
  - e le altre due permutazioni possibili

istruzione atomica – 2

S2		**dopo**			
	inizio	*t*1.*a*	*t*2.*a*	*t*1.*b*	*t*2.*b*
contoA	100	110	**130**		
contoB	200			**190**	**170**

S3		**dopo**			
	inizio	*t*1.*a*	*t*2.*a*	*t*2.*b*	*t*1.*b*
contoA	100	110	**130**		
contoB	200			**180**	**170**

t1 trasferisce 10 da contoA a contoB
t2 trasferisce 20 da contoA a contoB

istruzione atomica e linguaggio macchina

- in generale un singolo *statement* (o un costrutto strutturato) di un linguaggio di alto livello (p. es. C) viene tradotto dal compilatore in una sequenza di due o più istruzioni in linguaggio macchina
- la singola istruzione in **linguaggio macchina** (ma NON il singolo *statement* in linguaggio di alto livello) è sempre **atomica**
- così però il programmatore dovrebbe sapere come il linguaggio di alto livello viene tradotto in linguaggio macchina
- per esempio questo semplicissimo *statement* in linguaggio C

  `i++` cioè `i = i + 1`

  dove `i` è una variabile condivisa tra più thread, una volta tradotto in linguaggio macchina **può dare luogo a sequenza critica !**

istruzione atomica e linguaggio macchina

- per esempio lo *statement* di alto livello seguente

```
contoA = contoA + 100
```

è traducibile in linguaggio macchina in due modi

- modo 1, atomico (il processore lavora direttamente in memoria)

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

nota bene: *MIPS* non ha questa istruzione macchina (ma *IA* sì)

- modo 2, non atomico (il processore usa un registro interno *t0*):

```
lw   t0, contoA   // carica contoA in reg t0
addi t0, t0, 100  // somma cost 100 a reg t0
sw   t0, contoA   // memorizza reg t0 in contoA
```

nota bene: *MIPS* ha queste istruzioni macchina

statement concorrente serializzabile – 1

- dati due *statement* in linguaggio di alto livello appartenenti a una sequenza ed eseguibili da due thread, si può avere

  $t1.i < t2.j$ oppure $t2.j < t1.i$

- tuttavia si potrebbe anche avere la situazione seguente

  $t1.i :: t2.j$

- ciò indica un'esecuzione concorrente, dove le istruzioni macchina che realizzano (traducono) i due *statement* di alto livello si intercalano variamente (ciò implica la possibile scorrettezza dell'esecuzione)

- due *statement* C concorrenti sono serializzabili se la loro realizzazione concorrente in linguaggio macchina produce sempre risultati corretti
  - in molti casi si riesce a rendere serializzabili due *statement* C, traducendoli in linguaggio macchina in modo opportuno
  - tuttavia, in generale gli *statement* C che modificano una variabile basandosi sul valore della variabile stessa, non sono serializzabili

statement concorrente serializzabile – 2

t1.i $y = x + 2$ ***t2.j*** $y = x + 4$

- prima dell'esecuzione la variabile *x* vale 10
- l'esecuzione sequenziale *t*1.*i* < *t*2.*j* produce $y = 14$
- l'esecuzione sequenziale *t*2.*j* < *t*1.*i* produce $y = 12$

*t*1.*i*1	`lw`	`t1, X`	*t*2.*j*1	`lw`	`t0, X`
*t*1.*i*2	`addi`	`t1, t1, 2`	*t*2.*j*2	`addi`	`t0, t0, 4`
*t*1.*i*3	`sw`	`t1, Y`	*t*2.*j*3	`sw`	`t0, Y`

- qualsiasi esecuzione concorrente di queste due sequenze di istruzioni macchina (ossia in qualunque modo le due sequenze intercalino le loro istruzioni macchina) produce risultato 12 oppure 14
- cioè è equivalente all'esecuzione sequenziale
- pertanto gli *statement t*1.*i* e *t*2.*j* sono serializzabili

mutua esclusione – mutex – 1

- per garantire che l'esecuzione concorrente di una sequenza critica di istruzioni macchina o di uno *statement* di alto livello non serializzabili sia corretta, occorre realizzare la mutua esclusione sulla sequenza o sullo *statement*
- esistono costrutti specializzati per realizzare la mutua esclusione
- in *POSIX* il costrutto previsto a tale scopo si chiama mutex
  - un mutex implementa un blocco che ingloba la sequenza o lo *statement* critico, e il cui accesso è controllato da un segnale di occupato
  - quando un thread attiva un blocco, eventuali altri thread che in seguito tentassero di accedere al contenuto del blocco verrebbero messi in attesa fino al momento in cui il thread bloccante avrà liberato o rilasciato il blocco
- l'operazione (primitiva) di attivazione del blocco si chiama lock
- l'operazione (primitiva) di rilascio del blocco si chiama unlock

mutua esclusione – mutex – 2

- va dichiarata una variabile di tipo **`pthread_mutex_t`**

  `conti` nel programma di esempio

- va inizializzata tramite la primitiva **`pthread_mutex_init`**

- successivamente il mutex va bloccato e sbloccato tramite
  - **`pthread_mutex_lock`**
  - **`pthread_mutex_unlock`**

- quando un thread esegue l'operazione di lock, se il mutex è già bloccato il thread viene messo in stato di attesa

mutex – esempio

```
// trasferimento sincronizzato con mutex
1.  #include <pthread.h> <stdio.h>
2.  int contoA = 100;
3.  int contoB = 200;
4.  int totale;
5.  pthread_mutex_t conti; // dichiara mutex

6.  void * trasferisci (void * arg) {
7.     int importo = *arg;
8.     int cA, cB;
9.     pthread_mutex_lock (&conti);
10.    // inizio sequenza critica
11.    cA = contoA;
12.    // leggi contoA in var locale
13.    cB = contoB;
14.    // leggi contoB in var locale
15.    contoA = cA + importo;
16.    contoB = cB - importo;
17.    pthread_mutex_unlock (&conti);
18.    // fine sequenza critica
19.    return NULL;
20. } /* trasferisci */
```

```
21. pthread_t tID1, tID2;

22. void main ( ) {
23.    int importo1 = 10;
24.    int importo2 = 20;
25.    // inizializza mutex
26.    pthread_mutex_init (&conti, NULL);
27.    pthread_create (
          &tID1, NULL,
          trasferisci, &importo1
       );
28.    pthread_create (
          &tID2, NULL,
          trasferisci, &importo2
       );
29.    pthread_join (tID1, NULL);
30.    pthread_join (tID2, NULL);
31.    totale = contoA + contoB;
32.    printf (
          "contoA = %d  contoB = %d\n",
          contoA, contoB
       );
33.    printf ("il totale è %d\n", totale);
34. } /* main */
```

mutex – esempio

```
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 $
```

sequenza critica senza mutex

sequenza critica senza mutex – 1

- è possibile implementare la mutua esclusione (e pure altre funzioni / concetti) anche in assenza di librerie specifiche per la programmazione concorrente (come la libreria **`pthread`**)
- in questo caso però la programmazione concorrente diventa di realizzazione più complessa
- l'esercizio nel seguito consente di esplorare meglio alcuni possibili pericoli della programmazione concorrente stessa

sequenza critica senza mutex – versione 1
(mutua esclusione non garantita)

```
// sequenza critica con variabile
   intera al posto del mutex
1.  #include <pthread.h> <stdio.h>
2.  int blocca = 0;

3.  void * trasferisci (void * arg) {
4.     // busy waiting
5.     while (blocca == 1); // ciclo attesa
6.     // inizio sequenza critica
7.     blocca = 1;
8.     printf ("thread %d: entro in seq
          critica\n", (int) arg);
9.     printf ("thread %d: termino seq
          critica\n", (int) arg);
10.    blocca = 0;
11.    // fine sequenza critica
12.    return NULL;
13. } /* trasferisci */
```

```
14. pthread_t tID1, tID2;

15. void main ( ) {
16.    pthread_create (
          &tID1, NULL,
          trasferisci, (void *) 1
       );
17.    pthread_create (
          &tID2, NULL,
          trasferisci, (void *) 2
       );
18.    pthread_join (tID1, NULL);
19.    pthread_join (tID2, NULL);
20.    printf ("fine\n");
21. } /* main */
```

sequenza critica senza mutex – versione 1
(mutua esclusione non garantita)

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

sequenza critica senza mutex – versione 1
(mutua esclusione non garantita)

- il blocco di accesso alla sequenza critica è gestito tramite la variabile globale **`blocca`**
  - se = 1 sequenza già in esecuzione da parte di un thread
  - se = 0 sequenza eseguibile (da parte di un thread qualunque)
- **`mutex_lock`** è implementato così

```
while (blocca == 1) // ciclo attesa
blocca = 1
```

- **`mutex_unlock`** è implementato così

```
blocca = 0
```

- implementazione errata, la sequenza di istruzioni seguente è possibile

  ***t*1.5** (= false ⇒ ingr. in seq. critica) < ***t*2.5** (= false ⇒ ingr. in seq. critica) …

sequenza critica senza mutex – versione 2 (mutua esclusione garantita ma presenta deadlock)

- a ciascun thread X che esegue la sequenza critica si associa una variabile globale **`blocca_x`**
- essa rappresenta l'intenzione del thread X di accedere alla sequenza
  - se = 1 il thread X vuole entrare nella sequenza (si può dire, *prenota* la seq)
  - se = 0 il thread X non vuole entrare nella sequenza
- per potere effettivamente entrare nella sequenza critica, il generico thread X, dopo avere dichiarate le sue intenzioni (**`blocca_x = 1`**)
  - testa ogni altra variabile **`blocca_y`** (con **y** ≠ **x**)
  - e resta in attesa se una qualsiasi di queste vale 1
- al termine della sequenza critica, ciascun thread X libera la sua variabile blocca (**`blocca_x = 0`**)
- per gestire correttamente le variabili **`blocca ...`**, bisogna associare un indice (etichetta) diverso a ciascun thread, quando il thread è creato

sequenza critica senza mutex – versione 2
(mutua esclusione garantita ma presenta deadlock)

```
// mutua esclusione con deadlock
1.  #include <pthread.h> <stdio.h>
2.  int blocca1 = 0, blocca2 = 0;

3.  void * trasferisci (void * arg) {
4.     if ((int) arg == 1) {blocca1 = 1;
5.        while (blocca2 == 1);} /* if */
6.     if ((int) arg == 2) {blocca2 = 1;
7.        while (blocca1 == 1);} /* if */
8.     // inizio sequenza critica
9.     printf ("thread %d: entro in seq
          critica\n", (int) arg);
10.    printf ("thread %d: termino seq
          critica\n", (int) arg);
11.    if ((int) arg == 1) blocca1 = 0;
12.    if ((int) arg == 2) blocca2 = 0;
13.    // fine sequenza critica
14.    return NULL;
15. } /* trasferisci */
```

```
16. pthread_t tID1, tID2;

17. void main ( ) {
18.    pthread_create (
          &tID1, NULL,
          trasferisci,(void *) 1
       );
19.    pthread_create (
          &tID2, NULL,
          trasferisci,(void *) 2
       );
20.    pthread_join (tID1, NULL);
21.    pthread_join (tID2, NULL);
22.    printf ("fine\n");
23. } /* main */
```

sequenza critica senza mutex – versione 2
(mutua esclusione garantita ma presenta deadlock)

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

sequenza critica senza mutex – versione 2 (mutua esclusione garantita ma presenta deadlock)

- se uno dei due thread entra in sequenza critica è garantita la mutua esclusione
- ma si consideri la seguente sequenza di esecuzione di *statement*

$$t1.4 < t2.6 < t1.5 < t2.7$$

- essa mantiene indefinitamente in stato di attesa sia il thread 1 sia il thread 2, senza che nessuno dei due possa entrare nella sequenza critica liberando la propria variabile e così facendo terminare il ciclo di attesa dell'altro thread
- più in generale, questo succede quando tutti (due o più) thread impegnano virtualmente la risorsa in mutua esclusione senza usarla effettivamente, e pertanto non possono neppure rilasciarla
- quando un thread aspetta l'azione di un altro thread per proseguire e nello stesso tempo il secondo aspetta l'azione del primo, si ha deadlock (stallo)
- questo è uno dei problemi / errori più gravi in programmazione concorrente

sequenza critica senza mutex – versione 2 (mutua esclusione garantita ma presenta deadlock)

- se un thread (p. es. thread 1) entra in sequenza critica, è garantita la mutua esclusione, cioè l'altro thread (thread 2) non può essere entrato
- questa condizione è vera se ***t*1.8 < inizio di *t*2.7** (infatti se si avesse ***t*2.7 < inizio di *t*1.8** allora si avrebbe deadlock)
- se il thread 1 è in ***t*1.8** allora deve essere stato ***t*1.5 < *t*2.6**
- dunque poi può essere solo

  ***t*1.5 < *t*1.8 < *t*2.6 < *t*2.7**

  oppure

  ***t*1.5 < *t*2.6 < *t*1.8 < *t*2.7**

- pertanto thread 1 è in sequenza critica e thread 2 è in ciclo in ***t*2.7**

sequenza critica senza mutex – versione 3 (eliminazione del deadlock)

- si dichiara una nuova variabile globale – **`favorito`**
- i valori assegnabili a **`favorito`** sono quelli degli indici definiti nell'esempio precedente per gestire correttamente le variabili **`blocca_x`** che garantiscono la mutua esclusione
- ciascun thread assegna a **`favorito`** l'indice dell'altro thread (insomma prenota la sequenza, ma fa subito il *gentleman* e cede il passo all'altro thread – si può dire che assume un comportamento *fair*, cioè *equo*, *gentile*)
- così rende l'altro thread preferenziale a entrare nella sequenza critica
- l'ultimo thread che esegue l'istruzione di assegnamento a **`favorito`** è quello che resta in attesa fino al momento in cui l'esecuzione della sequenza critica da parte dell'altro è terminata
- si chiama «algoritmo di Petersen»

sequenza critica senza mutex – versione 3 (eliminazione del deadlock)

```
// mutua esclusione corretta
1.   #include <pthread.h> <stdio.h>
2.   int favorito = 1, blocca1 = 0, blocca2 = 0;

3.   void * trasferisci (void * arg) {
4.      if ((int) arg == 1) {
5.         blocca1 = 1;
6.         favorito = 2;
7.         while (blocca2 == 1 && favorito == 2);
8.      } /* if */
9.      if ((int) arg == 2) {
10.        blocca2 = 1;
11.        favorito = 1;
12.        while (blocca1 == 1 && favorito == 1);
13.     } /* if */
14.     // inizio sequenza critica
15.     printf ("thread %d: entro in seq critica\n",
           (int) arg);
16.     printf ("thread %d: termino seq 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.  } /* trasferisci */
```

```
22. pthread_t tID1, tID2;

23. void main ( ) {
24.     pthread_create (
           &tID1, NULL,
           trasferisci,(void *) 1
        );
25.     pthread_create (
           &tID2, NULL,
           trasferisci,(void *) 2
        );
26.     pthread_join (tID1, NULL);
27.     pthread_join (tID2, NULL);
28.     printf (“fine\n”);
29. } /* main */
```

sequenza critica senza mutex – versione 3
(eliminazione del deadlock)

```
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 $
```

sequenza critica senza mutex – versione 3 (eliminazione del deadlock)

- se sono stati eseguiti questi *statement* in qualsiasi ordine
  - ***t*1.5** (**`blocca1 = 1`**)
  - ***t*2.10** (**`blocca2 = 1`**)
- ma nessuno degli altri *statement* (nella versione 2 questo porta a deadlock)
- allora, se vale ***t*1.6 < *t*2.11** (assegnamento alla variabile **`favorito`**)
  - thread 2 rimarrà in attesa
  - mentre thread 1 entra in sequenza critica
- viceversa
  - thread 1 rimarrà in attesa
  - mentre thread 2 entra in sequenza critica

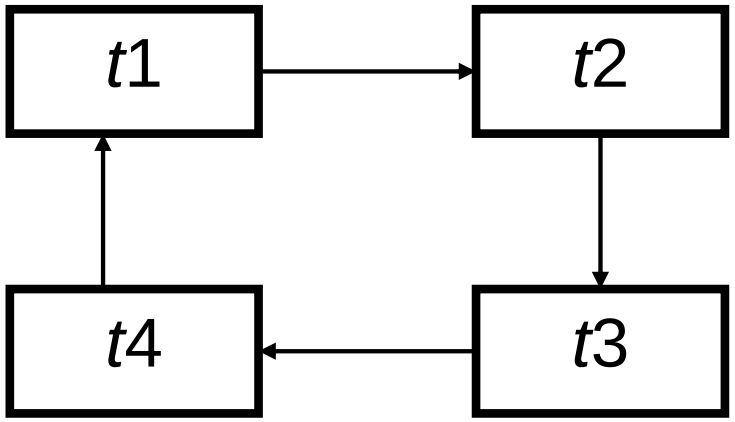
deadlock e mutex

deadlock e mutex – 1

- in generale si viene a determinare un deadlock quando:
  - due thread devono bloccare due risorse A e B
    (per accedervi in mutua esclusione)
  - ma le bloccano in ordine inverso
    (in generale non le bloccano nello stesso ordine)
- esempio:
  - thread 1 ha bloccata A e vuole bloccare B
  - thread 2 ha bloccata B e vuole bloccare A
- esempio:
  - due thread devono eseguire in ordine inverso due sequenze critiche annidate
  - allora sicuramente si ha deadlock

deadlock e mutex – 2

- più in generale, la situazione di deadlock è rappresentata dall'esistenza di un ciclo in un grafo di attesa:
  - i nodi del grafo rappresentano i thread
  - l'arco orientato dal nodo ***i*** al nodo ***j*** rappresenta il fatto che il thread ***ti*** richiede la risorsa ***x*** bloccata dal thread ***tj***



- nota bene: gli archi si devono riferire a situazioni di richiesta / blocco tutte possibili nel medesimo istante di tempo

deadlock e mutex – esempio

```
pthread_mutex_t mutexA, mutexB;

1.  void * lockApoiB (void * arg) {
2.     pthread_mutex_lock (&mutexA);
3.     // inizio sequenza critica 1
4.     printf ("thread %d: entro in seq
          critica 1\n", (int) arg);
5.     pthread_mutex_lock (&mutexB);
6.     // inizio sequenza critica 2
7.     printf ("thread %d: entro in seq
         critica 2\n", (int) arg);
8.     printf ("thread %d: termino seq
         critica 2\n", (int) arg);
9.     pthread_mutex_unlock (&mutexB);
10.    // fine sequenza critica 2
11.    printf ("thread %d: termino seq
         critica 1\n", (int) arg);
12.    pthread_mutex_unlock (&mutexA);
13.    // fine sequenza critica 1
14.    return NULL;
15. } /* lockApoiB */

16. void * lockBpoiA (void * arg) {
17.    pthread_mutex_lock (&mutexB);
18.    // inizio sequenza critica 2
19.    printf ("thread %d: entro in seq
          critica 2\n", (int) arg);
20.    pthread_mutex_lock (&mutexA);
21.    // inizio sequenza critica 1
22.    printf ("thread %d: entro in seq
          critica 1\n", (int) arg);
23.    printf ("thread %d: termino seq
          critica 1\n", (int) arg);
24.    pthread_mutex_unlock (&mutexA);
25.    // fine sequenza critica 1
26.    printf ("thread %d: termino seq
          critica 2\n", (int) arg);
27.    pthread_mutex_unlock (&mutexB);
28.    // fine sequenza critica 2
29.    return NULL;
30. } /* lockBpoiA */
```

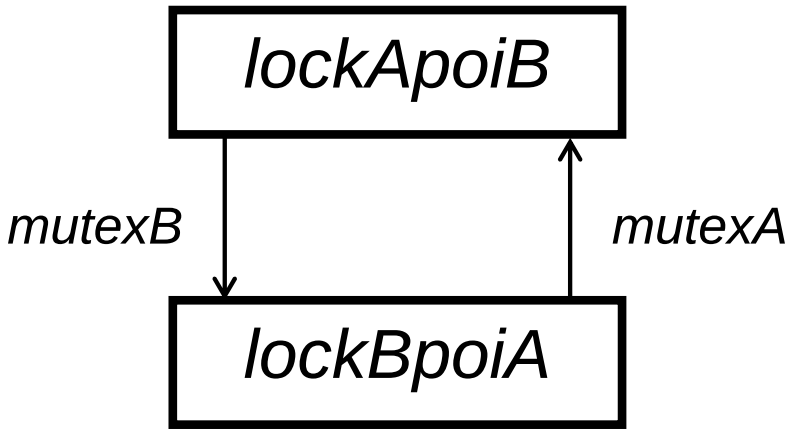
deadlock e mutex – esempio

```
1.  #include <pthread.h> <stdio.h>
2.  pthread_mutex_t mutexA, mutexB;

3.  void * lockApoiB (void * arg) {
4.     pthread_mutex_lock (&mutexA);
5.     printf ("thread %d: iniz 1\n", (int) arg);
6.     pthread_mutex_lock (&mutexB);
7.     printf ("thread %d: iniz 2\n", (int) arg);
8.     printf ("thread %d: fine 2\n", (int) arg);
9.     pthread_mutex_unlock (&mutexB);
10.    printf ("thread %d: fine 1\n", (int) arg);
11.    pthread_mutex_unlock (&mutexA);
12.    return NULL;
13. } /* lockApoiB */
14. void * lockBpoiA (void * arg) {
15.    pthread_mutex_lock (&mutexB);
16.    printf ("thread %d: iniz 2\n", (int) arg);
17.    pthread_mutex_lock (&mutexA);
18.    printf ("thread %d: iniz 1\n", (int) arg);
19.    printf ("thread %d: fine 1\n", (int) arg);
20.    pthread_mutex_unlock (&mutexA);
21.    printf ("thread %d: fine 2\n", (int) arg);
22.    pthread_mutex_unlock (&mutexB);
23.    return NULL;
24. } /* lockBpoiA */
```

```
25. pthread_t tID1, tID2;

26. void main ( ) {
27.    pthread_mutex_init (
          &mutexA, NULL
       );
28.    pthread_mutex_init (
          &mutexB, NULL
       );
29.    pthread_create (
          &tID1, NULL,
          lockApoiB, (void *) 1
       );
30.    pthread_create (
          &tID2, NULL,
          lockBpoiA, (void *) 2
       );
31.    pthread_join (tID1, NULL);
32.    pthread_join (tID2, NULL);
33.    printf ("fine\n");
34. } /* main */
```

deadlock e mutex – esempio

- grafo di attesa dell'esempio, dove le risorse richieste / bloccate etichettano gli archi (qui le risorse sono i mutex):



archi nell'istante di tempo in cui lockApoiB invoca lock (&mutexB) e lockBpoiA invoca lock (&mutexA)

- il ciclo denuncia la possibilità di avere un deadlock
- c'è anche il deadlock opposto, scambiando le risorse

deadlock e mutex – esempio

```
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
```

sincronizzazione e semafori

sincronizzazione e semaforo – 1

- sincronizzazione: relazione temporale deterministica che si vuole imporre tra due o più thread
- il costrutto specializzato, molto generale e utilizzabile anche per altri scopi, è il semaforo (*semaphore*)
- il modello di semaforo ha queste caratteristiche generali
  - il semaforo è una variabile di tipo **`sem_t`**
  - per le operazioni il semaforo è considerato molto simile a un intero, e come tale può avere valore positivo, nullo o negativo
  - il semaforo viene inizializzato a un certo valore, tipicamente ≥ 0, tramite la primitiva di inizializzazione **`sem_init (...)`**
  - un thread utilizza un semaforo tramite due sole primitive (operazioni): **`sem_wait (...)`** e **`sem_post (...)`**

sincronizzazione e semaforo – 2

- **`sem_wait (...)`** decrementa il valore del semaforo
  - se il semaforo ha valore > 0, il thread che ha invocato **`sem_wait`** decrementa di un'unità il semaforo e prosegue nell'esecuzione
  - se il semaforo ha valore ≤ 0, il thread che ha invocato **`sem_wait`** decrementa di un'unità il semaforo, ma viene bloccato in stato di attesa fino al momento in cui un altro thread invocherà **`sem_post`** sul quel semaforo, e solo allora potrà proseguire nell'esecuzione
- **`sem_post (...)`** incrementa il valore del semaforo
  - il thread che ha invocato **`sem_post`** incrementa di un'unità il semaforo, sblocca uno degli eventuali thread in stato di attesa su quel semaforo (se ce ne sono) e prosegue nell'esecuzione
- generalmente i thread in stato di attesa su un semaforo vengono sbloccati in ordine *FIFO*: primo bloccato – primo sbloccato
  - però l'ordine di sblocco dipende dal sistema operativo soggiacente

sincronizzazione e semaforo – 3

- il valore del semaforo è interpretabile come il numero di risorse disponibili – ossia associate al semaforo – per i thread che usano quel semaforo
- se il valore è > 0, rappresenta il numero di risorse disponibili e dunque rappresenta quanti thread possono decrementare il semaforo – cioè possono usare una risorsa – senza venire bloccati e messi in stato di attesa
- pertanto incrementare il semaforo significa liberare una risorsa e renderla nuovamente disponibile
- se il valore è < 0, rappresenta (in modulo) il numero (≥ 1) di thread bloccati in stato di attesa che una risorsa si liberi per così proseguire nell'elaborazione
- se il valore è $= 0$, significa sia che non ci sono thread bloccati in stato di attesa, sia che non ci sono risorse disponibili
- pertanto un thread che eseguisse **`sem_wait`** su un semaforo con valore $= 0$ si bloccherebbe in stato di attesa, e il semaforo assumerebbe valore negativo: –1
- un semaforo inizializzato a 1 e gestito come segue, è equivalente a un mutex:
  - per entrare in sequenza critica, un thread invoca wait, così marcando come occupata la sequenza critica; se poi un altro thread invocasse wait, si bloccherebbe in attesa
  - per uscire dalla sequenza critica, un thread invoca post, così marcando come libera la sequenza critica e sbloccando uno degli eventuali thread in attesa (se ce ne sono)

semaforo nello standard *POSIX* – **`pthread`**

- la realizzazione *POSIX* del semaforo è conforme al modello generale descritto prima, ma con una semplificazione

  il semaforo può soltanto avere valore positivo o nullo

- pertanto, anche quando sul semaforo ci sono uno o più thread bloccati in stato di attesa, il semaforo stesso non assume mai valore negativo, ma mantiene sempre il valore 0

- quando un thread invoca post su un semaforo di valore 0 e il semaforo ha uno o più thread bloccati in attesa, uno di essi viene sbloccato, e il semaforo mantiene il valore 0
  - se sem = 0 e non ci sono thread in attesa su sem, post mette sem = 1

- per valori > 0, il semaforo *POSIX* segue il modello generale

- tramite primitive ausiliarie della libreria **`pthread`**, è comunque possibile sapere quanti thread sono bloccati su un dato semaforo

sincronizzazione e semaforo – esempio 1

- due thread stampano le due sequenze di caratteri “*abc*” e “*xyz*”, rispettivamente
- in uscita si vuole stampare sempre la sequenza “*axbycz*”
  - la funzione tf1 stampa singolarmente i caratteri *x*, *y* e *z*
  - la funzione tf2 stampa singolarmente i caratteri *a*, *b* e *c*
- occorrono due semafori
  - **`sem1`** blocca il thread che scrive *x*, *y* e *z*
  - **`sem2`** blocca il thread che scrive *a*, *b* e *c*
- le due funzioni di thread sono diverse perché si deve iniziare la stampa con il carattere “*a*”

sincronizzazione e semaforo – esempio 1

```
1.  #include <pthread.h> <stdio.h> <semaphore.h>
2.  sem_t sem1, sem2;

3.  void * tf1 (void * arg) {
4.      sem_wait (&sem1);
5.      printf ("x");
6.      sem_post (&sem2); sem_wait (&sem1);
7.      printf ("y");
8.      sem_post (&sem2); sem_wait (&sem1);
9.      printf ("z");
10.     return NULL;
11. } /* tf1 */

12. void * tf2 (void * arg) {
13.     printf ("a");
14.     sem_post (&sem1); sem_wait (&sem2);
15.     printf ("b");
16.     sem_post (&sem1); sem_wait (&sem2);
17.     printf ("c");
18.     sem_post (&sem1);
19.     return NULL;
20. } /* tf2 */
```

```
21. pthread_t tID1, tID2;

22. void main ( ) {
23.     sem_init (&sem1, 0, 0);
24.     sem_init (&sem2, 0, 0);
25.     pthread_create (
            &tID1, NULL,
            tf1, NULL
        );
26.     pthread_create (
            &tID2, NULL,
            tf2, NULL
        );
27.     pthread_join (tID1, NULL);
28.     pthread_join (tID2, NULL);
29.     printf ("\nfine\n");
30.     return 0;
31. } /* main */
```

sincronizzazione e semaforo – esempio 1

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

sincronizzazione e semaforo – esempio 2

- rapporto produttore / consumatore: esempio tipico di utilizzo di semafori
- si vuole realizzare un programma che fa uso di un buffer (in linea di principio un array di caratteri), dove uno o più thread produttori scrivono dei caratteri e uno o più thread consumatori leggono dei caratteri
- la versione più semplice di tale sistema è quella dove
  - il buffer può contenere un solo carattere (variabile di tipo carattere)
  - un thread vi scrive i caratteri, ma uno per volta (thread produttore)
  - un thread ne legge i caratteri, ma uno per volta (thread consumatore)
- intuitivamente, occorrono due semafori che indicano lo stato del buffer
  - pieno per bloccare il consumatore, dunque vale 0 se il produttore non ha ancora inserito / scritto il carattere (all'inizio vale 0 per trattenere il consumatore)
  - vuoto per bloccare il produttore, dunque vale 0 se il consumatore non ha ancora prelevato / letto il carattere (all'inizio vale 1 per liberare il produttore)

sincronizzazione e semaforo – esempio 2

```
// produttore consumatore con semafori
// versione semplice ma con errori !

#include <pthread.h> <stdio.h> <semaphore.h>
char buffer;
int fine = 0;
sem_t pieno, vuoto;

1.  void * scrivi (void * arg) {
2.      int i;
3.      for (i = 0; i < 5; i++)  {
4.          sem_wait (&vuoto);
5.          buffer = `a' + i;
6.          sem_post (&pieno);
7.      } /* for */
8.      fine = 1;
9.      return NULL;
10. } /* scrivi */
```

```
1.  void * leggi (void * arg) {
2.      char miobuffer;
3.      int i;
4.      while (fine == 0) {
5.          sem_wait (&pieno);
6.          miobuffer = buffer;
7.          sem_post (&vuoto);
8.          printf (
                "il buffer contiene %c\n",
                miobuffer
            );
9.      } /* while */
10.     return NULL;
11. } /* leggi */
```

sincronizzazione e semaforo – esempio 2

```
1.  #include <pthread.h> <stdio.h> <semaphore.h>
2.  char buffer; int fine = 0; sem_t pieno, vuoto;

3.  void * scrivi (void * arg) {
4.     int i;
5.     for (i = 0; i < 5; i++)  {
6.        sem_wait (&vuoto);
7.        buffer = `a' + i;
8.        sem_post (&pieno);
9.     } /* for */
10.    fine = 1;
11.    return NULL;
12. } /* scrivi */

13. void * leggi (void * arg) {
14.    char miobuffer;
15.    int i;
16.    while (fine == 0) {
17.       sem_wait (&pieno);
18.       miobuffer = buffer;
19.       sem_post (&vuoto);
20.       printf ("il buffer contiene %c\n", miobuffer);
21.    } /* while */
22.    return NULL;
23. } /* leggi */
```

```
// produttore consumatore con semafori
// versione semplice ma con errori !
24.  pthread_t tID1, tID2;

25.  void main ( ) {
26.     sem_init (&pieno, 0, 0);
27.     sem_init (&vuoto, 0, 1);
28.     // buffer vuoto all'inizio
29.     pthread_create (
           &tID1, NULL,
           scrivi, NULL
        );
30.     pthread_create (
           &tID2, NULL,
           leggi, NULL
        );
31.     pthread_join (tID1, NULL);
32.     pthread_join (tID2, NULL);
33.     printf ("fine\n");
34.  } /* main */
```

sincronizzazione e semaforo – esempio 2

- analisi formale di questa soluzione semplice (e come si vedrà poi, errata)
- thread produttore: è strutturato per iterare esattamente per cinque volte, e segnala la fine del ciclo produttivo con l'assegnamento **`fine`** = 1
- thread consumatore: poiché a priori non sa quanti caratteri verranno prodotti, è strutturato per iterare un numero arbitrario di volte, ogniqualvolta la variabile condivisa **`fine`** è trovata ancora a valore 0
- i semafori pieno e vuoto proteggono i corpi dei cicli **`for`** e **`while`** del produttore e del consumatore, e hanno questi effetti sulla concorrenza
  - rendono mutuamente esclusivi i due corpi protetti (come anche un mutex potrebbe fare)
  - bloccando a turno il consumatore e il produttore, garantiscono che i corpi di **`for`** e **`while`** vengano eseguiti in ordine di alternanza, a cominciare da **`for`** (produttore) (mentre un mutex NON potrebbe garantire alcun tipo di ordinamento !)
- in apparenza tutto ciò è ben congegnato, anzi pare quasi ovvio, ma …

sincronizzazione e semaforo – esempio 2

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

- errore visibile: non viene stampato il carattere “*e*”, il quinto e ultimo prodotto
- causa: il thread produttore, uscendo dal ciclo **`for`**, può arrivare a eseguire lo *statement* **`fine = 1`** prima che il thread consumatore sia riuscito a rivalutare per la quinta volta la condizione **`fine == 0`** del ciclo **`while`**
- effetto: il thread consumatore trova la condizione già falsa e salta la quinta iterazione di **`while`**, pertanto non consuma e non stampa l’ultimo carattere
- soluzione ovvia: aggiungere dopo **`while`** una copia del corpo per la quinta iterazione, quando è saltata, cioè quando il semaforo vuoto ha ancora valore 0

sincronizzazione e semaforo – esempio 2

- inoltre, sebbene nel test non si sia manifestato, si potrebbe avere un deadlock !
- causa: il thread produttore, uscendo dal ciclo **`for`**, può arrivare a eseguire lo *statement* **`fine = 1`** dopo che il thread consumatore, avendo regolarmente eseguita la quinta (e in teoria ultima) iterazione del ciclo **`while`**, è di nuovo riuscito a rivalutare la condizione **`fine == 0`** di **`while`** (ossia per la sesta volta)
- effetto: il thread consumatore trova la condizione ancora vera ed entra nella sesta iterazione di **`while`**, dove si blocca indefinitamente in attesa di un sesto carattere che arriverà mai, poiché il thread produttore ne produce solo cinque
- questo è un caso di deadlock, seppure limitato al solo thread consumatore !
- soluzione: spostare lo *statement* **`fine = 1`** dentro la sezione protetta dai semafori nel corpo di **`for`** (dopo il riempimento del buffer), però facendo in modo che lo *statement* sia eseguito solo durante la quinta e ultima iterazione di **`for`**
- motivazione: per l'ordine di alternanza, il consumatore esegue la sua quinta iterazione dopo la quinta del produttore, pertanto a un eventuale sesto giro il consumatore troverebbe la variabile **`fine`** ormai senz'altro aggiornata a 1 e non entrerebbe più nel corpo del ciclo (e salterebbe anche il corpo aggiuntivo finale)

sincronizzazione e semaforo – esempio 2

```
// produttore consumatore con
   semafori – versione corretta
#include <pthread.h> <stdio.h>
        <semaphore.h>

1.  char buffer;
2.  int fine = 0;
3.  sem_t pieno, vuoto;

4.  void * scrivi (void * arg) {
5.    int i;
6.    for (i = 0; i < 5; i++) {
7.      sem_wait (&vuoto);
8.      buffer = `a' + i;
9.      if (i == 4) fine = 1;
10.     sem_post (&pieno);
11.   } /* for */
12.   return NULL;
13. } /* scrivi */
```

l'assegnamento a **fine** viene eseguito nella sezione protetta durante la quinta e ultima iterazione di **for**

```
14. void * leggi (void * arg) {
15.   char miobuf;
16.   int i;
17.   while (fine == 0) {
18.     sem_wait (&pieno);
19.     miobuf = buffer;
20.     sem_post (&vuoto);
21.     printf ("il buffer contiene %c\n", miobuf);
22.   } /* while */
23.   // controllo per l'ultimo carattere
24.   sem_getvalue (&vuoto, &i); // i = val. di vuoto
25.   if (i == 0) {
26.     // buffer pieno o in via di riempimento
27.     sem_wait (&pieno);
28.     miobuf = buffer;
29.     sem_post (&vuoto);
30.     printf ("il buffer contiene %c\n", miobuf);
31.   } /* if */
32.   return NULL;
33. } /* leggi */
```

ramo then: il consumatore ha iterato **while** solo per 4 volte, ma il produttore ha riempito o sta riempiendo il buffer per la quinta volta

la primitiva **sem_getvalue** è una funzione ausiliaria della libreria *NPTL*

sincronizzazione e semaforo – esempio 2

```
// produttore consumatore con semafori
1.  pthread_t tID1, tID2;

2.  void main ( ) {
3.      sem_init (&pieno, 0, 0);
4.      sem_init (&vuoto, 0, 1);
5.      // buffer vuoto all'inizio
6.      pthread_create (&tID1, NULL, scrivi, NULL);
7.      pthread_create (&tID2, NULL, leggi, NULL);
8.      pthread_join (tID1, NULL);
9.      pthread_join (tID2, NULL);
10.     // buffer vuoto alla fine
11.     printf ("fine\n");
12. } /* main */
```

sincronizzazione e semaforo – esempio 2

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

- gli errori sono scomparsi, e in effetti ora il sistema è formalmente corretto
- osservazione: dopo la terminazione coerente dei due thread secondari produttore e consumatore, cioè dopo le due *join* quando resta solo il thread principale, lo stato dei semafori è di nuovo quello iniziale pieno = 0 e vuoto = 1
- ciò è coerente con lo stato finale effettivo del buffer, che al termine della sequenza di produzione e consumo è di nuovo vuoto come all'inizializzazione
- qui non è indispensabile garantire tale coerenza stretta, ma lo diventerebbe se il sistema fosse permanente e la produzione/consumo potesse ripartire