

Sincronizzazione tra thread

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

Luca Iezzi Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Reti di Calcolatori A.A. 2022/23

Prof.ssa Chiara Petrioli Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma

Sincronizzazione tra thread

2

Oltre i semafori Unix, esistono altri meccanismi di sincronizzazione che sono specifici per thread:

- Mutex
- Condition variables
- Barriere

Accesso esclusivo ad una risorsa condivisa

3

Determinate risorse non permettono l'uso concorrente da parte di più thread, perciò bisogna assicurarsi che al più un thread stia eseguendo operazioni su di essa.

Un **mutex** è una variabile di tipo **pthread_mutex_t** che permette l'implementazione di questa garanzia.

➤ Inizializzazione a **PTHREAD_MUTEX_INITIALIZER**

➤ **int pthread_mutex_lock(pthread_mutex_t *mutex)**

Un thread **tenta** di acquisire un lock su mutex; se mutex è **già in possesso** di un **altro** thread, l'invocazione di questa funzione è **bloccante** fino alla rispettiva unlock; a seconda del **tipo di mutex** usato, un thread può rimanere bloccato su un mutex di cui è già in possesso

➤ **int pthread_mutex_trylock(pthread_mutex_t *mutex)**

Analogamente a pthread_mutex_lock, tuttavia non è bloccante se il mutex è già in possesso di un altro thread, restituendo un codice di errore ed impostando errno al valore **EBUSY**

➤ **int pthread_mutex_unlock(pthread_mutex_t *mutex)**

Rilascia il lock precedentemente acquisito su mutex; se ci sono diversi thread bloccati da invocazioni di pthread_mutex_lock su mutex, il thread che ne acquisirà il lock sarà determinato dalla **politica di scheduling** usata dai thread

Relazione tra mutex e scheduling

4

- SCHED_FIFO
A **parità di priorità**, i thread seguono l'**ordine** in cui sono rimasti bloccati sull'acquisizione del lock; altrimenti, thread con priorità **più alta** vengono spostati verso la **testa** della coda di attesa secondo
- SCHED_RR
A **parità di priorità**, i thread vengono eseguiti per un **determinato periodo di tempo**; altrimenti, thread con priorità **più alta** possono **superare** questa limitazione
- SCHED_SPORADIC
Nel caso di **sistemi periodici**, un thread **atipico** può essere associato ad una priorità tale da garantire l'acquisizione di un lock prima degli altri

Tipi di mutex

5

Possono esistere diversi tipi di mutex:

- **NORMAL**
Deadlock se un thread che possiede un lock tenta di **acquisirlo di nuovo**; **undefined behavior** se un thread rilascia un lock non in suo possesso
- **ERRORCHECK**
Lancia un **errore** se un thread che possiede un lock tenta di **acquisirlo di nuovo**; lancia un **errore** se un thread rilascia un lock non in suo possesso
- **RECURSIVE**
Permette ad un thread che possiede un lock di **acquisirlo di nuovo**, incrementando il **conteggio**: il mutex sarà di nuovo **disponibile** quando tale conteggio sarà **pari a 0**; lancia un **errore** se un thread rilascia un lock non in suo possesso

Inoltre, può essere specificato il livello di robustezza di un mutex:

- Non-robust
 - Robust
- } Cambia il comportamento di mutex di tipo **NORMAL**:
lancia un **errore** se un thread tenta di rilasciare un lock di cui **non** è in possesso

Attributi dei mutex

6

Analogamente alla creazione dei thread, è possibile specificare svariati comportamenti di un mutex attraverso alcuni attributi impostati tramite un'apposita funzione di inizializzazione:

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)`
Inizializzazione di mutex con gli attributi `attr`; un valore `NULL` per `attr` è equivalente all'applicazione dei valori di `default` (dipendente dall'implementazione)
- `int pthread_mutex_destroy(pthread_mutex_t *mutex)`
Distrugge mutex

Gestione degli attributi mutex:

- `int pthread_mutexattr_init(pthread_mutexattr_t *attr)`
Inizializza il valore della struttura attributi puntata da `attr` ai valori di default
- `int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`
Distrugge la struttura attributi puntata da `attr`

Attributi dei mutex

7

Gli attributi che possono essere impostati sono:

- `int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)`
Imposta il tipo del mutex: `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_ERRORCHECK`, `PTHREAD_MUTEX_RECURSIVE`; esiste anche `PTHREAD_MUTEX_DEFAULT`, ma l'implementazione potrebbe semplicemente definirlo come un alias di uno dei precedenti
- `int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr, int robust)`
Imposta il livello di robustezza del mutex: `PTHREAD_MUTEX_STALLED` (`non-robust`), `PTHREAD_MUTEX_ROBUST`
- `int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling)`
Imposta la priorità massima per cui il mutex ha efficacia: superata questa, la sezione critica viene eseguita anche senza lock. Dovrebbe quindi essere impostata ad un valore maggiore o uguale alla priorità massima dei thread che fanno uso di questo mutex
- `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)`
Imposta il metodo di utilizzo del mutex: `PTHREAD_PRIO_NONE` (nessuna variazione a priorità o scheduling), `PTHREAD_PRIO_INHERIT` (priorità thread alzata a quella più alta tra i thread in attesa su qualsiasi mutex di cui è in possesso), `PTHREAD_PRIO_PROTECT` (priorità thread alzata al ceiling di tutti i mutex di cui è in possesso; altri thread ignorati)
- `int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared)`
Imposta il livello di condivisione di un mutex: `PTHREAD_PROCESS_SHARED` (mutex può essere utilizzato anche su memoria condivisa con altri processi), `PTHREAD_PROCESS_PRIVATE`.

Esempio di data race

8

```
1 static int *counter = NULL;
```

```
1 int main(){
2     counter = malloc(sizeof(int));
3     *counter = 0;
4     pthread_t tids[5];
5     for(int i=0; i<5; i++){
6     {
7         pthread_create(&tids[i], NULL, thread_fn, NULL);
8     }
9     for(int i=0; i<5; i++){
10    {
11        pthread_join(tids[i], NULL);
12    }
13    printf("End of program: counter = %d\n", *counter);
14    free(counter);
15    return 0;
16 }
```

```
1 void operation_fn(){
2     pthread_t tid = pthread_self();
3
4     // Operazioni non atomiche
5     for(int i=0; i<100; i++){
6         *counter += 2;
7     }
8     for(int i=0; i<100; i++){
9         *counter -= 1;
10    }
11
12    printf("Thread %lu: counter = %d\n", tid, *counter);
13 }
14
15 void thread_fn(void *arg){
16     operation_fn();
17     sleep(1);
18     operation_fn();
19     sleep(1);
20     operation_fn();
21     pthread_exit(0);
22 }
```


Esempio di data race

9

```
Thread 139704237684480: counter = 100
Thread 139704078288640: counter = 400
Thread 139704212506368: counter = 500
Thread 139704229291776: counter = 200
Thread 139704220899072: counter = 300
Thread 139704237684480: counter = 600
Thread 139704212506368: counter = 800
Thread 139704078288640: counter = 700
Thread 139704229291776: counter = 900
Thread 139704220899072: counter = 1000
Thread 139704212506368: counter = 1172
Thread 139704078288640: counter = 1265
Thread 139704237684480: counter = 1165
Thread 139704220899072: counter = 1365
Thread 139704229291776: counter = 1465
End of program: counter = 1465
```

```
Thread 139983648364288: counter = 100
Thread 139983639971584: counter = 200
Thread 139983623186176: counter = 500
Thread 139983488968448: counter = 400
Thread 139983631578880: counter = 300
Thread 139983648364288: counter = 600
Thread 139983639971584: counter = 700
Thread 139983623186176: counter = 800
Thread 139983488968448: counter = 900
Thread 139983631578880: counter = 1000
Thread 139983648364288: counter = 1100
Thread 139983639971584: counter = 1200
Thread 139983488968448: counter = 1300
Thread 139983631578880: counter = 1500
Thread 139983623186176: counter = 1400
End of program: counter = 1500
```

```
Thread 140106009016064: counter = 100
Thread 140106000623360: counter = 200
Thread 140105975445248: counter = 500
Thread 140105983837952: counter = 400
Thread 140105992230656: counter = 300
Thread 140106009016064: counter = 600
Thread 140106000623360: counter = 866
Thread 140105983837952: counter = 934
Thread 140105975445248: counter = 834
Thread 140105992230656: counter = 1034
Thread 140106009016064: counter = 1134
Thread 140105983837952: counter = 1234
Thread 140106000623360: counter = 1334
Thread 140105975445248: counter = 1434
Thread 140105992230656: counter = 1534
End of program: counter = 1534
```

Ogni thread aggiunge 300 al valore del contatore, eppure su 3 esecuzioni consecutive **solamente una** effettivamente mostra il risultato corretto (non garantito in generale):

- **Data race** avvengono nella lettura e scrittura della variabile `counter`
- Un thread può leggere il valore della variabile condivisa e dopodiché potenzialmente sovrascrivere il valore assegnato da un altro thread

Esempio di mutex

10

```
1 static pthread_mutex_t counterMutex = PTHREAD_MUTEX_INITIALIZER;
2 static int *counter = NULL;
3
4 void operation_fn(){
5     pthread_t tid = pthread_self();
6
7     // Operazioni non atomiche, ma garanzia di mutex
8     for(int i=0; i<100; i++){
9         pthread_mutex_lock(&counterMutex);
10        *counter += 2;
11        pthread_mutex_unlock(&counterMutex);
12    }
13    for(int i=0; i<100; i++){
14        pthread_mutex_lock(&counterMutex);
15        *counter -= 1;
16        pthread_mutex_unlock(&counterMutex);
17    }
18
19    printf("Thread %lu: counter = %d\n", tid, *counter);
20 }
```

Mutex inizializzato tramite
`PTHREAD_MUTEX_INITIALIZER`

L'accesso alle **sezioni critiche** è controllato da
mutex: al massimo **un solo** thread per volta può
accedere a counter

P.S. Il resto del codice rimane invariato

Esempio di mutex

11

```
Thread 140515875211008: counter = 100
Thread 140515850032896: counter = 400
Thread 140515858425600: counter = 300
Thread 140515866818304: counter = 200
Thread 140515760666368: counter = 500
Thread 140515875211008: counter = 600
Thread 140515850032896: counter = 700
Thread 140515858425600: counter = 800
Thread 140515866818304: counter = 900
Thread 140515760666368: counter = 1000
Thread 140515875211008: counter = 1100
Thread 140515866818304: counter = 1595
Thread 140515850032896: counter = 1519
Thread 140515858425600: counter = 1562
Thread 140515760666368: counter = 1500
End of program: counter = 1500
```

```
Thread 140231874643712: counter = 100
Thread 140231849465600: counter = 400
Thread 140231857858304: counter = 300
Thread 140231866251008: counter = 200
Thread 140231841072896: counter = 500
Thread 140231874643712: counter = 600
Thread 140231841072896: counter = 1169
Thread 140231866251008: counter = 1093
Thread 140231857858304: counter = 1038
Thread 140231849465600: counter = 1000
Thread 140231874643712: counter = 1100
Thread 140231841072896: counter = 1200
Thread 140231866251008: counter = 1300
Thread 140231857858304: counter = 1565
Thread 140231849465600: counter = 1500
End of program: counter = 1500
```

```
Thread 140650102286080: counter = 100
Thread 140649978394368: counter = 500
Thread 140650085500672: counter = 300
Thread 140650077107968: counter = 400
Thread 140650093893376: counter = 200
Thread 140650102286080: counter = 600
Thread 140649978394368: counter = 700
Thread 140650085500672: counter = 800
Thread 140650077107968: counter = 900
Thread 140650093893376: counter = 1000
Thread 140650102286080: counter = 1100
Thread 140649978394368: counter = 1312
Thread 140650085500672: counter = 1655
Thread 140650093893376: counter = 1533
Thread 140650077107968: counter = 1500
End of program: counter = 1500
```

Ogni thread aggiunge 300 al valore del contatore:

- Nel corso delle operazioni, i **thread si alternano** secondo lo scheduling loro assegnato dal sistema operativo
- Il contatore viene quindi aggiornato da più thread, tuttavia con la **garanzia di nessuna data race**
- Al termine di tutte le esecuzioni, indipendentemente dall'ordine delle operazioni, il risultato è **corretto**

Sincronizzazione basata su notifiche

12

Molto spesso, operazioni implementate con multi-threading possono dedicare risorse esclusive a determinati thread, ma è necessario garantire un **ordine di esecuzione** dei sotto-problemi.

L'utilizzo di un mutex risulta quindi irrilevante in questi casi:

- Due thread hanno risorse **esclusive**
Non c'è necessità di condivisione della risorsa
- Un thread **dipende** dall'altro
La corretta esecuzione del proprio task necessita che l'altro thread sia giunto ad un **determinato punto** del task

Inoltre, usando i mutex, l'imposizione dell'ordine di esecuzione tramite scheduling e priorità thread **non** è una soluzione **efficiente** e **immediata**:

- **Tanti mutex** specifici quanti i punti di sincronizzazione necessari, massima attenzione alle sezioni critiche
- Uso di un **ciclo** nel thread per controllare **costantemente** un flag di sincronizzazione → **busy wait**
- Configurazione dettagliata delle **politiche di scheduling** e relativi **attributi**

Sincronizzazione basata su notifiche

13

Per questi motivi, in POSIX possono essere usate le **condition variables** per implementare un meccanismo di sincronizzazione thread basato su **notifiche**. L'uso di condition variables è **sempre** abbinato con quello dei **mutex**.

- Tipo variabile **pthread_cond_t**, valore di inizializzazione **PTHREAD_COND_INITIALIZER**
- **int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)**
Il thread viene bloccato su una condition variable **cond**, dopo aver **acquisito il lock** su mutex: questa funzione rilascerà il lock e metterà il thread in attesa di una notifica su cond. Un errore viene lanciato se il thread non possiede il lock su mutex
- **int pthread_cond_signal(pthread_cond_t *cond)**
Invia una notifica tramite la condition variable cond; se ci sono diversi thread, **almeno uno** viene sbloccato: **quale** thread viene svegliato è determinato dalla **politica di scheduling**. È necessario che il thread che invoca questa funzione sia **in possesso del lock** sul mutex **abbinato a cond**, ovvero quello con cui è stata invocata una wait da un altro thread

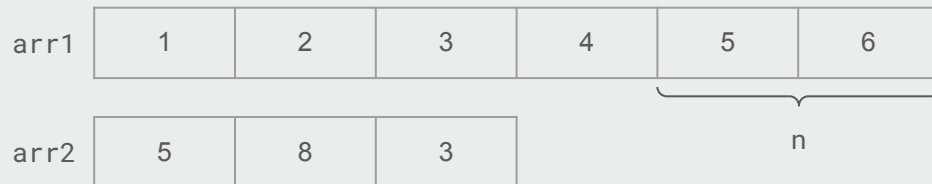
Un thread che precedentemente invoca una wait e, a seguito di una signal da un altro thread, viene risvegliato, **ottiene di nuovo il lock** su mutex automaticamente.

Questo meccanismo di notifica non va confuso con quello dei **segnali Unix**: completamente diversi!

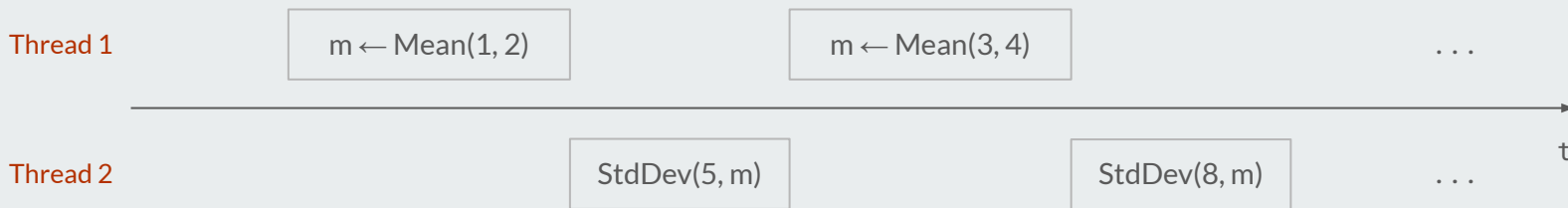
Esempio

14

Dati due array `arr1` e `arr2`, per **ogni valore** di `arr2` calcolare la **deviazione standard** rispetto alla **media** dei valori in `arr1`, **presi n alla volta**. Un thread calcola la media, l'altro thread calcola la deviazione standard.

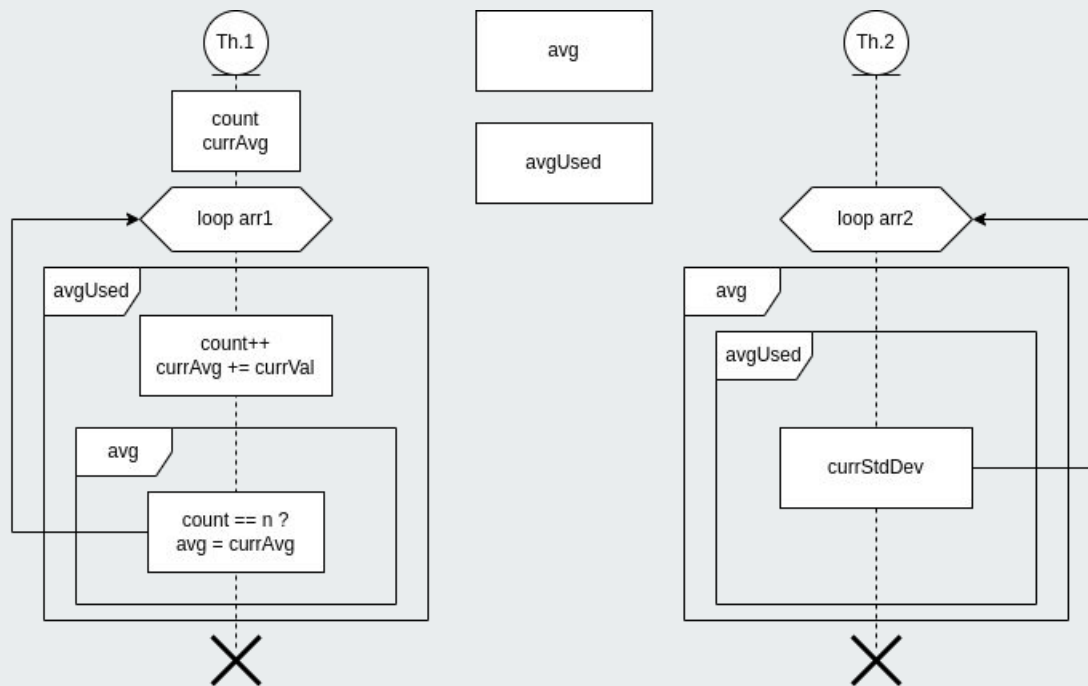


Esecuzione



Esempio

15



Esempio: solo mutex

16

- Flag avgReady e avgUsed
- Necessità di due mutex:
 - avgMutex
Accesso ad avgReady
 - usedMutex
Accesso ad avgUsed
- Bisogna accuratamente gestire la **sovrapposizione** delle sezioni critiche; pena: **deadlock!**

```
1 static int arr1[15] = {1,2,3, 4,5,6, 7,8,9, 10,11,12, 13,14,15};
2 static int arr2[5] = {5,10,2,4,6};
3
4 static float avg = 0.f;
5 static int avgReady = 0;
6 static pthread_mutex_t avgMutex = PTHREAD_MUTEX_INITIALIZER;
7
8 static int avgUsed = 1;
9 static pthread_mutex_t usedMutex = PTHREAD_MUTEX_INITIALIZER;
10
11 int main(){
12     pthread_t tid2;
13     pthread_create(&tid2, NULL, thread2_fn, NULL);
14     pthread_t tid1;
15     pthread_create(&tid1, NULL, thread1_fn, NULL);
16     pthread_join(tid2, NULL);
17     pthread_join(tid1, NULL);
18     return 0;
19 }
```


Esempio: solo mutex

17

```

1 void thread1_fn(void *arg){
2     pthread_t tid = pthread_self();
3     printf("Thread %lu: starting\n", tid);
4
5     // Calcola media di 3 valori di arr1 alla volta
6     float currAvg = 0.f;
7     int count = 0;
8     for(int i=0; i<15; i++){
9         printf("Thread %lu: array item %d\n", tid, i);
10
11         // Attendo la richiesta di aggiornare la media
12         while(1){
13             pthread_mutex_lock(&usedMutex);
14             if(avgUsed == 1){
15                 break;
16             }
17             pthread_mutex_unlock(&usedMutex);
18         }
19     }

```

} Busy wait

```

20     // Processo il valore corrente di arr1
21     count++;
22     currAvg += arr1[i];
23
24     // Aggiorno media corrente
25     pthread_mutex_lock(&avgMutex);
26     if(count == 3){
27         avg = currAvg / count;
28         avgReady = 1;
29         avgUsed = 0;
30         printf("Thread %lu: updating avg to %f\n", tid, avg);
31         count = 0;
32         currAvg = 0.f;
33     }
34     pthread_mutex_unlock(&avgMutex);
35     pthread_mutex_unlock(&usedMutex);
36 }
37
38 printf("Thread %lu: exiting\n", tid);
39 pthread_exit(0);
40 }

```

Il thread del calcolo della media attende che la media precedentemente calcolata sia stata usata, ed una volta effettuato il calcolo di nuovo renderà disponibile il valore aggiornato della media.

Esempio: solo mutex

18

```

1 void thread2_fn(void *arg){
2   pthread_t tid = pthread_self();
3   printf("Thread %lu: starting\n", tid);
4
5   // Calcola deviazione standard di ogni valore di arr2
6   // rispetto alla media corrente in arr1
7   for(int i=0; i<5; i++){
8     printf("Thread %lu: array item %d; wait for avg\n", tid, i);
9
10    // Attendo che la media sia disponibile
11    float currAvg = 0.f;
12    while(1){
13      pthread_mutex_lock(&avgMutex);
14      if(avgReady == 1){
15        break;
16      }
17      currAvg = avg;
18      pthread_mutex_unlock(&avgMutex);
19    }

```

} Busy wait

```

21   // Calcolo deviazione standard con la nuova media
22   float stdDev = sqrtf(powf(arr2[i] - currAvg, 2));
23   printf("Thread %lu: stddev: %f\n", tid, stdDev);
24
25   {
26     // Segnala che la media è stata usata e va aggiornata
27     pthread_mutex_lock(&usedMutex);
28     avgReady = 0;
29     avgUsed = 1;
30     pthread_mutex_unlock(&usedMutex);
31     pthread_mutex_unlock(&avgMutex);
32   }
33   printf("Thread %lu: exiting\n", tid);
34   pthread_exit(0);
35 }

```

Il thread del calcolo della deviazione standard attende che la media venga calcolata, ed una volta ottenuta la deviazione standard segnerà che la media attuale è stata usata, richiedendo che venga aggiornata.

Esempio: solo mutex (output)

19

```
Thread 140528340551424: starting
Thread 140528340551424: array item 0; wait for avg
Thread 140528332158720: starting
Thread 140528332158720: array item 0
Thread 140528332158720: array item 1
Thread 140528332158720: array item 2
Thread 140528332158720: updating avg to 2.000000
Thread 140528332158720: array item 3
Thread 140528340551424: stddev: 5.000000
Thread 140528340551424: array item 1; wait for avg
Thread 140528332158720: array item 4
Thread 140528332158720: array item 5
Thread 140528332158720: updating avg to 5.000000
Thread 140528332158720: array item 6
Thread 140528340551424: stddev: 8.000000
```

...

L'efficienza di questa soluzione **non è ottimale**:
entrambi i thread fanno uso di **busy wait** su avgMutex e
usedMutex.

Si potrebbe sostituire il **controllo periodico** delle rispettive
flag con delle **condition variables**, bloccando fino all'arrivo
delle notifiche.

Visualizzando la **busy wait** dei thread

```
Thread 139661557106432: starting
Thread 139661557106432: array item 0; wait for avg
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661557106432: trying for avg ready
Thread 139661548713728: starting
Thread 139661548713728: array item 0
Thread 139661548713728: trying for avg used
Thread 139661548713728: array item 1
Thread 139661548713728: trying for avg used
```

Esempio: condition variables

20

```
1 static int arr1[15] = {1,2,3, 4,5,6, 7,8,9, 10,11,12, 13,14,15};
2 static int arr2[5] = {5,10,2,4,6};
3
4 static float avg = 0.f;
5 static int avgReady = 0;
6 static pthread_mutex_t avgMutex = PTHREAD_MUTEX_INITIALIZER;
7 static pthread_cond_t avgCond = PTHREAD_COND_INITIALIZER;
8
9 static int avgUsed = 1;
10 static pthread_mutex_t usedMutex = PTHREAD_MUTEX_INITIALIZER;
11 static pthread_cond_t usedCond = PTHREAD_COND_INITIALIZER;
```

Aggiuntivamente ai mutex di ogni flag, vengono dichiarate anche le rispettive condition variables **avgCond** e **usedCond**. La definizione delle sezioni critiche risulterà più semplice grazie ad esse.

P.S. L'implementazione della funzione `main` rimane invariata.

Esempio: condition variables

21

```

1 void thread1_fn(void *arg){
2     pthread_t tid = pthread_self();
3     printf("Thread %lu: starting\n", tid);
4
5     // Calcola media di 3 valori di arr1 alla volta
6     float currAvg = 0.f;
7     int count = 0;
8     for(int i=0; i<15; i++){
9         printf("Thread %lu: array item %d\n", tid, i);
10
11         // Attendo la richiesta di aggiornare la media
12         pthread_mutex_lock(&usedMutex);
13         while(avgUsed != 1){
14             printf("Thread %lu: trying for avg used\n", tid);
15             pthread_cond_wait(&usedCond, &usedMutex);
16         }
17
18         // Processo il valore corrente di arr1
19         count++;
20         currAvg += arr1[i];

```

```

21
22     // Aggiorno media corrente
23     pthread_mutex_lock(&avgMutex);
24     if(count == 3){
25         avg = currAvg / count;
26         avgReady = 1;
27         avgUsed = 0;
28         printf("Thread %lu: updating avg to %f\n", tid, avg);
29         pthread_cond_signal(&avgCond);
30         count = 0;
31         currAvg = 0.f;
32     }
33     pthread_mutex_unlock(&avgMutex);
34     pthread_mutex_unlock(&usedMutex);
35 }
36
37 printf("Thread %lu: exiting\n", tid);
38 pthread_exit(0);
39 }

```

L'operazione di **wait** su **usedCond** viene invocata all'interno di un ciclo perché **non è garantito** che dopo una notifica su una condition variable sia questo thread ad ottenere il possesso di **usedMutex**, tra i vari altri thread in attesa.

Esempio: condition variables

22

```
1 void thread2_fn(void *arg){
2   pthread_t tid = pthread_self();
3   printf("Thread %lu: starting\n", tid);
4
5   // Calcola deviazione standard di ogni valore di arr2
6   // rispetto alla media corrente in arr1
7   for(int i=0; i<5; i++){
8     printf("Thread %lu: array item %d; wait for avg\n", tid, i);
9
10    // Attendo che la media sia disponibile
11    float currAvg = 0.f;
12    pthread_mutex_lock(&avgMutex);
13    while(avgReady != 1){
14      printf("Thread %lu: trying for avg ready\n", tid);
15      pthread_cond_wait(&avgCond, &avgMutex);
16    }
17    currAvg = avg;
```

```
19   // Calcolo deviazione standard con la nuova media
20   float stdDev = sqrtf(powf(arr2[i] - currAvg, 2));
21   printf("Thread %lu: stddev: %f\n", tid, stdDev);
22
23   // Segnala che la media è stata usata e va aggiornata
24   pthread_mutex_lock(&usedMutex);
25   avgReady = 0;
26   avgUsed = 1;
27   pthread_cond_signal(&usedCond);
28   pthread_mutex_unlock(&usedMutex);
29   pthread_mutex_unlock(&avgMutex);
30 }
31
32 printf("Thread %lu: exiting\n", tid);
33 pthread_exit(0);
34 }
```

Lo stesso approccio viene analogamente utilizzato nel thread per il calcolo della deviazione standard.

Si notano comunque le due invocazioni di **signal** dentro la rispettiva **sezione critica** per avgCond (prima) e usedCond.

Esempio: condition variables (output)

23

```
Thread 140001802856192: starting
Thread 140001802856192: array item 0; wait for avg
Thread 140001802856192: trying for avg ready
Thread 140001794463488: starting
Thread 140001794463488: array item 0
Thread 140001794463488: array item 1
Thread 140001794463488: array item 2
Thread 140001794463488: updating avg to 2.000000
Thread 140001794463488: array item 3
Thread 140001794463488: trying for avg used
Thread 140001802856192: stddev: 3.000000
Thread 140001802856192: array item 1; wait for avg
Thread 140001802856192: trying for avg ready
Thread 140001794463488: array item 4
Thread 140001794463488: array item 5
Thread 140001794463488: updating avg to 5.000000
Thread 140001794463488: array item 6
Thread 140001794463488: trying for avg used
Thread 140001802856192: stddev: 5.000000
```

```
Thread 140001802856192: array item 2; wait for avg
Thread 140001802856192: trying for avg ready
Thread 140001794463488: array item 7
Thread 140001794463488: array item 8
Thread 140001794463488: updating avg to 8.000000
Thread 140001794463488: array item 9
Thread 140001794463488: trying for avg used
Thread 140001802856192: stddev: 6.000000
Thread 140001802856192: array item 3; wait for avg
Thread 140001802856192: trying for avg ready
Thread 140001794463488: array item 10
Thread 140001794463488: array item 11
Thread 140001794463488: updating avg to 11.000000
Thread 140001794463488: array item 12
Thread 140001794463488: trying for avg used
Thread 140001802856192: stddev: 7.000000
Thread 140001802856192: array item 4; wait for avg
Thread 140001802856192: trying for avg ready
Thread 140001794463488: array item 13
Thread 140001794463488: array item 14
Thread 140001794463488: updating avg to 14.000000
Thread 140001794463488: exiting
Thread 140001802856192: stddev: 8.000000
Thread 140001802856192: exiting
```

Avendo da subito inserito i messaggi nei cicli while, è possibile notare come l'uso di **condition variables** eviti che i thread restino in **busy wait**: wait viene invocata ed il thread entra in attesa.

Inoltre, la variabile di iterazione può essere direttamente il flag protetto dal mutex, aumentando l'interpretabilità.

Condition variables: alcune insidie

24

L'operazione `wait` mette in attesa il thread che la invoca **anche a tempo indefinito**; invece, `signal` risveglia **almeno 1** thread, ovvero non necessariamente tutti quelli che sono in attesa → una ricetta per il disastro.

➤ `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`

Comportamento analogo a `pthread_cond_wait`, ma con possibilità di sfociare in **timeout**: il thread viene infatti messo in attesa per un determinato intervallo di tempo (**abstime**); un valore di ritorno pari ad **ETIMEDOUT** indica il termine dell'intervallo di attesa e non l'arrivo di una notifica su `cond`

➤ `int pthread_cond_broadcast(pthread_cond_t *cond)`

Comportamento analogo a `pthread_cond_signal`, tuttavia con la garanzia che **ogni thread in attesa** su `cond` venga risvegliato

È una buona pratica **utilizzare sempre `pthread_cond_broadcast`** al posto di `pthread_cond_signal`, data la garanzia di risveglio di tutti i thread in attesa, mentre l'utilizzo di `pthread_cond_timedwait` è situazionale.

Attributi per condition variables

25

In modo simile a quanto avvenga con i mutex, anche il comportamento delle condition variables può essere raffinato tramite l'impostazione dei loro attributi al momento della creazione.

Gli attributi sono relativi a:

- *Orologio* – particolarmente importante nell'uso di `pthread_cond_timedwait`
- Livello di condivisione della condition variable (*pshared* – uguale ai mutex)

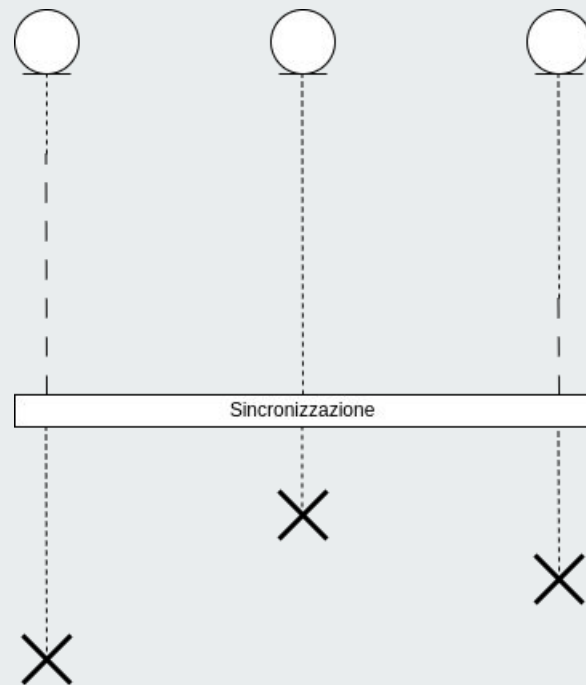
Le funzioni hanno struttura ed uso simile a quanto visto con gli attributi dei mutex.

Sincronizzazione tra *livelli* parziali di un task

26

Un'applicazione multi-threading potrebbe definire una suddivisione di sotto-problemi in modo tale che diversi thread **procedano parallelamente** finché un **punto di sincronizzazione** non venga raggiunto.

Dopo le operazioni di sincronizzazione, i thread eventualmente possono **riprendere** l'esecuzione con un **nuovo task**, sfruttando i risultati **aggregati**.



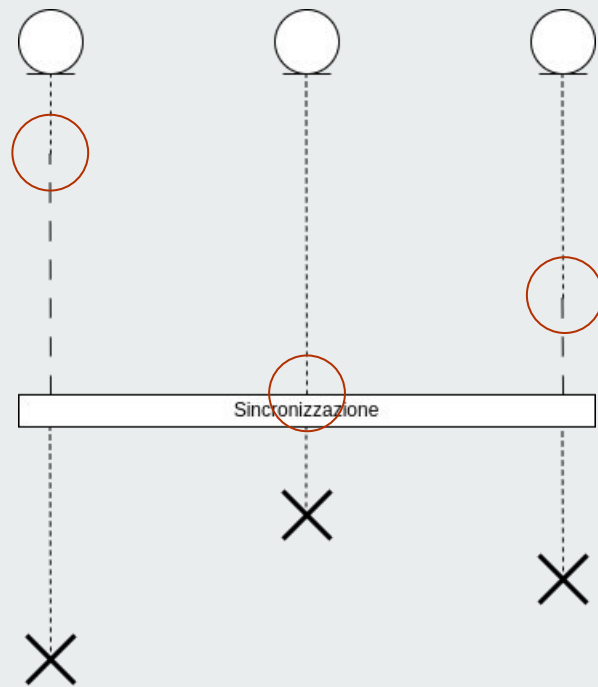
Sincronizzazione tra *livelli* parziali di un task

27

Un'applicazione multi-threading potrebbe definire una suddivisione di sotto-problemi in modo tale che diversi thread **procedano parallelamente** finché un **punto di sincronizzazione** non venga raggiunto.

Dopo le operazioni di sincronizzazione, i thread eventualmente possono **riprendere** l'esecuzione con un **nuovo task**, sfruttando i risultati **aggregati**.

- Ogni thread può quindi **completare** il task individuale **prima degli altri**, rimanendo però **in attesa** della sincronizzazione
- Non appena tutti i thread che devono sincronizzarsi raggiungono il punto di sincronizzazione, si procede con l'eventuale **aggregazione** dei risultati parziali
- Questo meccanismo può sostituire `pthread_join` sui singoli thread, ed invece riutilizzarli per task diversi dopo la sincronizzazione; viene **risparmiato il tempo di setup e distruzione** dei thread



Sincronizzazione tra *livelli* parziali di un task

28

Questa funzionalità è implementata tramite il concetto di **barriera**: rappresentata con variabili di tipo **pthread_barrier_t** in POSIX.

- Non c'è un valore di inizializzazione specifico per la variabile
- `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *attr, unsigned count)`

Inizializza una barriera `barrier` con gli attributi forniti tramite `attr` e con un **numero di thread necessari** per la sincronizzazione pari a **count**; l'unico attributo è *pshared* (analogo a mutex e condition variables)

- `int pthread_barrier_destroy(pthread_barrier_t *barrier)`

Distrugge la barriera puntata da `barrier`

- `int pthread_barrier_wait(pthread_barrier_t *barrier)`

Blocca il thread chiamante sul punto di sincronizzazione **finché tutti i count thread** non invocano questa funzione durante la loro esecuzione; una volta che tutti i thread necessari sono giunti a questo punto, la **barriera viene resettata** come se ne fosse appena stato eseguito `pthread_barrier_init`.

Il valore di ritorno è **PTHREAD_BARRIER_SERIAL_THREAD** per **uno dei thread** invocanti mentre per tutti gli altri è pari a 0; tale thread può essere quello incaricato per svolgere l'eventuale aggregazione dei risultati parziali

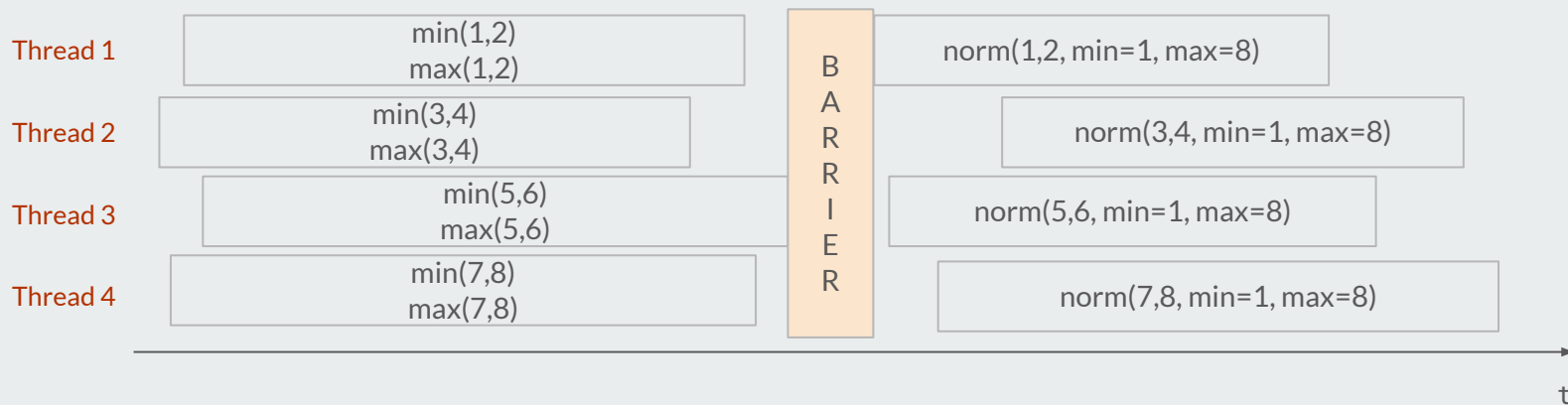
Esempio

29

Dato un array `arr`, normalizzare tutti i valori nell'intervallo $[0, 1]$ utilizzando 4 thread sia per la **ricerca** dei valori **minimo** e **massimo** che per la **normalizzazione**. Ogni thread processa n elementi dell'array.



Esecuzione



Esempio

30

```
1 static float arr[8] = {1,2, 3,4, 5,6, 7,8};  
2 const int n = 2;  
3 static float min, max;  
4 static pthread_mutex_t summaryMutex = PTHREAD_MUTEX_INITIALIZER;  
5 static pthread_barrier_t barrier;
```

- Array condiviso tra tutti i thread → accesso esclusivo a sotto-array
- Condivisione dei valori minimo e massimo dell'array → controllato tramite mutex
- Visibilità globale della barriera di sincronizzazione

Esempio

31

```
1 int main(){
2   min = arr[0];
3   max = arr[0];
4   pthread_barrier_init(&barrier, NULL, 4);
5
6   printf("Main: original array\n");
7   for(int i=0; i<8; i++){
8     printf("%d: %f\n", i, arr[i]);
9   }
10
11  pthread_t tids[4];
12  for(int i=0; i<4; i++){
13    pthread_create(&tids[i], NULL, thread_fn, (void*)i);
14  }
15  for(int i=0; i<4; i++){
16    pthread_join(tids[i], NULL);
17  }
18
19  printf("Main: normalized array\n");
20  for(int i=0; i<8; i++){
21    printf("%d: %f\n", i, arr[i]);
22  }
23  return 0;
24 }
```

Inizializzazione barriera per 4 thread, attributi di default

Ogni thread riceve come argomento un ID intero, per determinare il sotto-array a cui ha accesso esclusivo:

- Offset calcolato dal proprio ID
- Numero elementi a partire dall'offset è uguale per tutti i thread (n)

Esempio

32

```
1 void thread_fn(void *arg){
2     int thread_id = (int)arg;
3     int ix_start = n * thread_id;
4     int ix_end = ix_start + n;
5     printf("Thread #%d sub-array indices: start = %d, end = %d\n", thread_id, ix_start, ix_end);
6
7     // Ricerca min & max locali
8     float threadMin = arr[ix_start], threadMax = arr[ix_start];
9     for(int i=ix_start; i<ix_end; i++){
10         float currVal = arr[i];
11         if(currVal < threadMin){
12             threadMin = currVal;
13         }
14         if(currVal > threadMax){
15             threadMax = currVal;
16         }
17     }
18     printf("Thread #%d: min = %f, max = %f\n", thread_id, threadMin, threadMax);
}
```

Determinazione sotto-array

Accesso esclusivo,
nessuna sincronizzazione
necessaria

- Ogni thread accede a **sotto-array indipendenti** di `arr`
- La ricerca dei valori minimo e massimo quindi è scomposta negli **stessi task**, ma con un **sotto-insieme dei dati** complessivi (modello **data-parallel**)

Anche chiamato **SIMD**: Same Instruction stream, Multiple Data stream

Esempio

33

```
19 pthread_mutex_lock(&summaryMutex);
20 if(threadMin < min){
21     min = threadMin;
22 }
23 if(threadMax > max){
24     max = threadMax;
25 }
26 pthread_mutex_unlock(&summaryMutex);
27
28 // Attesa di sincronizzazione di tutti i thread
29 pthread_barrier_wait(&barrier);
30
31 // Normalizzazione in-place dell'array
32 for(int i=ix_start; i<ix_end; i++){
33     float normVal = (arr[i] - min) / (max - min);
34     arr[i] = normVal;
35 }
36
37 pthread_exit(NULL);
38 }
```

Sezione critica per aggiornare le variabili globali min e max

Garanzia che tutti i thread siano arrivati a questo punto dell'esecuzione

Accesso esclusivo di nuovo

- Una volta trovati i valori di interesse, si procede con una **semplice sincronizzazione** di min e max controllata da un mutex
- La **barriera garantisce** che i valori di **min** e **max** siano effettivamente i valori **globali** di arr
- Si procede quindi con il secondo task, **normalizzazione**, che è **di nuovo** di tipo **SIMD**

Esempio (output)

34

Soluzioni alternative

- Semaforo Unix
 - Inizializzato a 4 nel main
 - Ogni thread che completa la ricerca min e max: **decrementa di 1**
 - Ogni thread, dopo il signal, sem_op = 0: **attende che diventi 0**
 - Il resto della funzione thread rimane invariata
- Barriera, **senza** min e max globali
 - Ogni thread ha **zone dedicate** per salvare **threadMin** e **threadMax**
Devono essere comunque visibili a tutti i thread (ad es. sotto forma di **array**)
 - **Condition variable** summaryReady
 - Controllo del **valore di ritorno** di pthread_barrier_wait:
 - 0:
Operazione **wait** su summaryReady
 - PTHREAD_BARRIER_SERIAL_THREAD:
Calcola min e max **globali** usando i 4 threadMin e threadMax;
sovrascrive i valori in prima posizione con quelli globali e dopodiché
invia un **broadcast** su summaryReady
 - Il resto della funzione thread rimane invariata

```

Main: original array
0: 1.000000
1: 2.000000
2: 3.000000
3: 4.000000
4: 5.000000
5: 6.000000
6: 7.000000
7: 8.000000
Thread #1 sub-array indices: start = 2, end = 4
Thread #2 sub-array indices: start = 4, end = 6
Thread #0 sub-array indices: start = 0, end = 2
Thread #1: min = 3.000000, max = 4.000000
Thread #3 sub-array indices: start = 6, end = 8
Thread #3: min = 7.000000, max = 8.000000
Thread #2: min = 5.000000, max = 6.000000
Thread #0: min = 1.000000, max = 2.000000
Main: normalized array
0: 0.000000
1: 0.142857
2: 0.285714
3: 0.428571
4: 0.571429
5: 0.714286
6: 0.857143
7: 1.000000

```

Riferimenti

35

- [pthread_lock/trylock/unlock](#)
- [pthread_mutex_init/destroy, pthread_mutexattr_init/destroy](#)
- [pthread_mutexattr_get/settype, pthread_mutexattr_get/setrobust,](#)
[pthread_mutexattr_get/setprioceiling, pthread_mutexattr_get/setprotocol,](#)
[pthread_mutexattr_get/setpshared](#)
- [pthread_cond_wait/timedwait, pthread_cond_signal/broadcast](#)
- [pthread_barrier_init/destroy, pthread_barrier_wait](#)