Sincronizzazione tra thread

Reti di Calcolatori A.A. 2023/24

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

Michele Mastrogiovanni - Dipartimento di Ingegneria Informatica

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione tra thread

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

- Mutex
- Condition variables
- Barriere

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Accesso esclusivo ad una risorsa condivisa

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)`
  Analoga 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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Relazione tra mutex e scheduling

- `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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Tipi di mutex

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 de 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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Attributi dei mutex

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`

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Attributi dei mutex

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` .

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio di data race


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

```c
int main(){
  counter = malloc(sizeof(int));
  *counter = 0;
  pthread_t tids[5];
  for(int i=0; i<5; i++)
  {
    pthread_create(&tids[i], NULL, thread_fn, NULL);
  }
  for(int i=0; i<5; i++)
  {
    pthread_join(tids[i], NULL);
  }
  printf("End of program: counter = %d\n", *counter);
  free(counter);
  return 0;
}
```

```c
void operation_fn(){
  pthread_t tid = pthread_self();

  // Operazioni non atomiche
  for(int i=0; i<100; i++){
    *counter += 2;
  }
  for(int i=0; i<100; i++){
    *counter -= 1;
  }

  printf("Thread %lu: counter = %d\n", tid, *counter);
}

void thread_fn(void *arg){
  operation_fn();
  sleep(1);
  operation_fn();
  sleep(1);
  operation_fn();
  pthread_exit(0);
}
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio di data race


```
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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio di mutex


```c
static pthread_mutex_t counterMutex = PTHREAD_MUTEX_INITIALIZER;
static int *counter = NULL;

void operation_fn(){
  pthread_t tid = pthread_self();

  // Operazioni non atomiche, ma garanzia di mutex
  for(int i=0; i<100; i++){
    pthread_mutex_lock(&counterMutex);
    *counter += 2;
    pthread_mutex_unlock(&counterMutex);
  }
  for(int i=0; i<100; i++){
    pthread_mutex_lock(&counterMutex);
    *counter -= 1;
    pthread_mutex_unlock(&counterMutex);
  }

  printf("Thread %lu: counter = %d\n", tid, *counter);
}
```

Mutex inizializzato tramite `PTHREAD_MUTEX_INIZIALIZER`

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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio di mutex


```
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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione basata su notifiche

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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione basata su notifiche

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

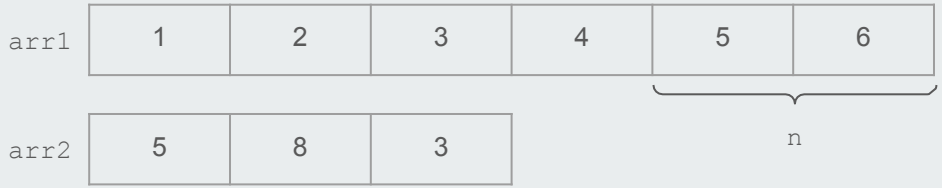
Un thread che precedentemente invoca una wait e, a seguito di una signal da un altro thread, viene risvegliato, ottiene di nuovo il lock su mutex automaticamente.

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

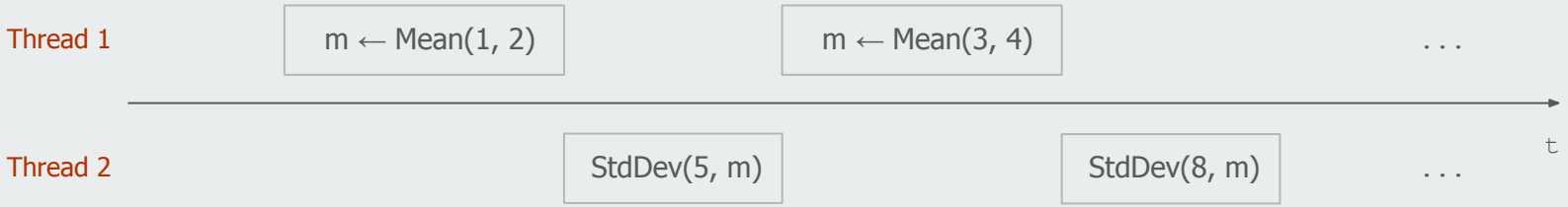
Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio

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.

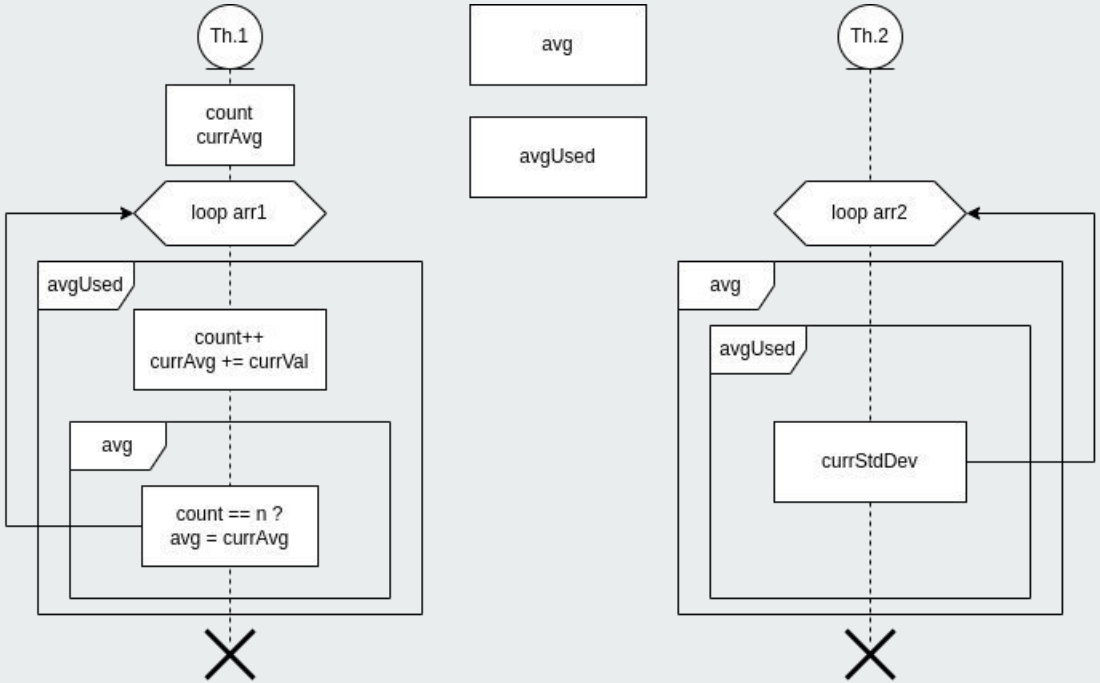


Esecuzione



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: solo mutex

- 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!

```c
static int arr1[15] = {1,2,3, 4,5,6, 7,8,9, 10,11,12, 13,14,15};
static int arr2[5] = {5,10,2,4,6};

static float avg = 0.f;
static int avgReady = 0;
static pthread_mutex_t avgMutex = PTHREAD_MUTEX_INITIALIZER;

static int avgUsed = 1;
static pthread_mutex_t usedMutex = PTHREAD_MUTEX_INITIALIZER;

int main(){
  pthread_t tid2;
  pthread_create(&tid2, NULL, thread2_fn, NULL);
  pthread_t tid2;
  pthread_create(&tid1, NULL, thread1_fn, NULL);
  pthread_join(tid2, NULL);
  pthread_join(tid1, NULL);
  return 0;
}
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: solo mutex


```
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.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: solo mutex


```c
void thread2_fn(void *arg){
  pthread_t tid = pthread_self();
  printf("Thread %lu: starting\n", tid);

  // Calcola deviazione standard di ogni valore di arr2
  // rispetto alla media corrente in arr1
  for(int i=0; i<5; i++){
    printf("Thread %lu: array item %d; wait for avg\n", tid, i);

    // Attendo che la media sia disponibile
    float currAvg = 0.f;
    while(1){
      pthread_mutex_lock(&avgMutex);
      if(avgReady == 1){
        break;
      }
      currAvg = avg;
      pthread_mutex_unlock(&avgMutex);
    }
```

Busy wait

```c
    // Calcolo deviazione standard con la nuova media
    float stdDev = sqrtf(powf(arr2[i] - currAvg, 2));
    printf("Thread %lu: stddev: %f\n", tid, stdDev);

    // Segnala che la media è stata usata e va aggiornata
    pthread_mutex_lock(&usedMutex);
    avgReady = 0;
    avgUsed = 1;
    pthread_mutex_unlock(&usedMutex);
    pthread_mutex_unlock(&avgMutex);
  }

  printf("Thread %lu: exiting\n", tid);
  pthread_exit(0);
}
```

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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: solo mutex (output)


```
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 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
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: condition variables


```
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.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: condition variables


```c
void thread1_fn(void *arg){
  pthread_t tid = pthread_self();
  printf("Thread %lu: starting\n", tid);

  // Calcola media di 3 valori di arr1 alla volta
  float currAvg = 0.f;
  int count = 0;
  for(int i=0; i<15; i++){
    printf("Thread %lu: array item %d\n", tid, i);

    // Attendo la richiesta di aggiornare la media
    pthread_mutex_lock(&usedMutex);
    while(avgUsed != 1){
      printf("Thread %lu: trying for avg used\n", tid);
      pthread_cond_wait(&usedCond, &usedMutex);
    }

    // Processo il valore corrente di arr1
    count++;
    currAvg += arr1[i];

    // Aggiorno media corrente
    pthread_mutex_lock(&avgMutex);
    if(count == 3){
      avg = currAvg / count;
      avgReady = 1;
      avgUsed = 0;
      printf("Thread %lu: updating avg to %f\n", tid, avg);
      pthread_cond_signal(&avgCond);
      count = 0;
      currAvg = 0.f;
    }
    pthread_mutex_unlock(&avgMutex);
    pthread_mutex_unlock(&usedMutex);
  }

  printf("Thread %lu: exiting\n", tid);
  pthread_exit(0);
}
```

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.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: condition variables


```c
void thread2_fn(void *arg){
  pthread_t tid = pthread_self();
  printf("Thread %lu: starting\n", tid);

  // Calcola deviazione standard di ogni valore di arr2
  // rispetto alla media corrente in arr1
  for(int i=0; i<5; i++){
    printf("Thread %lu: array item %d; wait for avg\n", tid, i);

    // Attendo che la media sia disponibile
    float currAvg = 0.f;
    pthread_mutex_lock(&avgMutex);
    while(avgReady != 1){
      printf("Thread %lu: trying for avg ready\n", tid);
      pthread_cond_wait(&avgCond, &avgMutex);
    }
    currAvg = avg;
    // Calcolo deviazione standard con la nuova media
    float stdDev = sqrtf(powf(arr2[i] - currAvg, 2));
    printf("Thread %lu: stddev: %f\n", tid, stdDev);

    // Segnala che la media è stata usata e va aggiornata
    pthread_mutex_lock(&usedMutex);
    avgReady = 0;
    avgUsed = 1;
    pthread_cond_signal(&usedCond);
    pthread_mutex_unlock(&usedMutex);
    pthread_mutex_unlock(&avgMutex);
  }

  printf("Thread %lu: exiting\n", tid);
  pthread_exit(0);
}
```

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`.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio: condition variables (output)


```
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.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Condition variables: alcune insidie

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.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Attributi per condition variables

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)

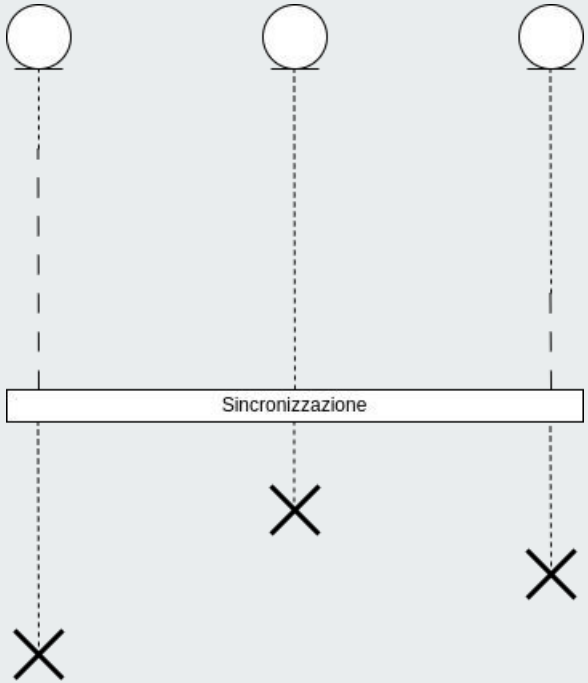
Le funzioni hanno struttura ed uso simile a quanto visto con gli attributi dei mutex.

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione tra *livelli* parziali di un task

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.



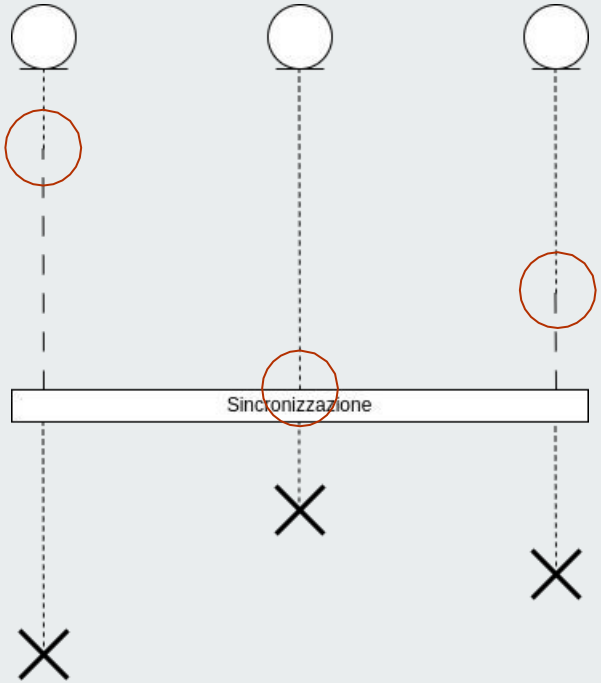
Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione tra *livelli* parziali di un task

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 procedere 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



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Sincronizzazione tra *livelli* parziali di un task

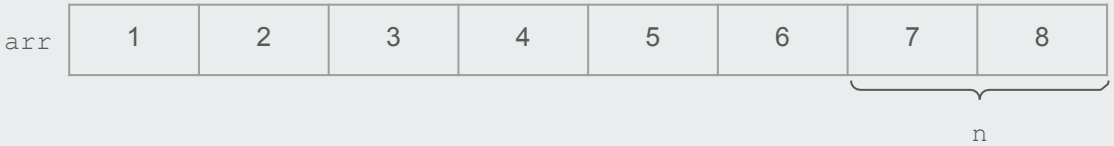
Questa funzionalità è implementata tramite il concetto di barriera: rappresentata con variabili di tipo `pthread_barrier_t` in POSIX.

- 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

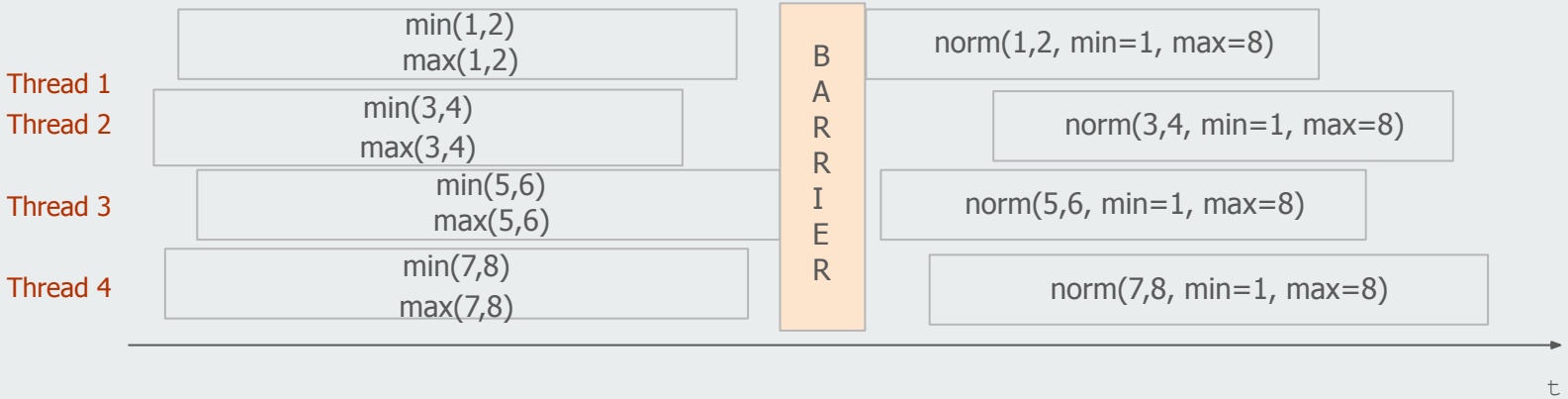
Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma
  ```

Esempio

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



Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio


```
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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio


```
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 a valori dell'array

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`)

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio


```
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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio


```
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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Esempio (output)

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
```

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma

Riferimenti

- 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

Ringraziamenti: - Emanuele Giona Dipartimento di Informatica, Sapienza Università di Roma - **Luca Iezzi** Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma