Sincronizzazione

Problem solving concorrente

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Esame del 17.02.2016

Esercizio

- Si realizzi uno schema di sincronizzazione in cui
  - Sono presenti due insiemi di Readers, denominati R1 e R2
  - Un insieme di Writers, denominati W
  - Ogni membro di R1 (R2) può accedere alla sezione critica insieme a altri membri di R1 (R2), però membri di R1 e R2, oppure R1 e W, oppure R2 e W devono accedere alla sezione critica in mutua esclusione

Soluzione

- Semafori e variabili

```
n1 = n2 = 0;
init (s1, 1);
init (s2, 1);
```

```
init (busy, 1);
init (meW, 1);
```

- Writer

```
wait (busy);
...
scrittura
...
signal (busy);
```

```
wait (meW);
wait (busy);
scrittura
signal (busy);
signal (meW);
```

Soluzione

- Reader 1

```
wait (s1);
  n1++;
  if (n1==1)
    wait (busy);
signal (s1);
...
lettura
...
wait (s1);
  n1--;
  if (n1==0)
    signal (busy);
signal (s1);
```

- Reader 2

```
wait (s2);
  n2++;
  if (n2==1)
    wait (busy);
signal (s2);
...
lettura
...
wait (s2);
  n2--;
  if (n2==0)
    signal (busy);
signal (s2);
```

Esame del 29.06.2016

Esercizio

- Si devono gestire
  - P produttori
  - Due insiemi di consumatori C1 e C2
  - Due code Q1 e Q2, di lunghezza N1 e N2
- In tale sistema
  - I P produttori producono tutti prima un oggetto sulla coda Q1 e successivamente un oggetto sulla coda Q2
  - I consumatori C1 consumano solo oggetti di Q1
  - I consumatori C2 consumano solo oggetti di Q2

Soluzione

- Semafori e variabili

```
init (full1, 0);
init (full2, 0);
init (empty1, N1);
init (empty2, N2);
init (MEp1, 1);
init (MEp2, 1);
init (MEc1, 1);
init (MEc2, 1);
```

Soluzione

- Produttore

```
P() {
   item m;
   while (1) {
      m = produce ();
      wait (empty1);
      wait (MEp1);
      enqueue1 (m);
      signal (MEp1);
      signal (full1);
      m = produce ();
      wait (empty2);
      wait (MEp2);
      enqueue2 (m);
      signal (MEp2);
      signal (full2);
   }
}
```

Soluzione

- Consumer 1

```
C1() {
   item m;
   while (1) {
      wait (full1);
      wait (MEc1);
      m = dequeue1 ();
      signal (MEc1);
      signal (empty1);
      consume (m);
   }
}
```

- Consumer 2

```
C2() {
   item m;
   while (1) {
      wait (full2);
      wait (MEc2);
      m = dequeue2 ();
      signal (MEc2);
      signal (empty2);
      consume (m);
   }
}
```

Esame del 27.02.2017

Esercizio

- Siano dati i tre processi successivi utilizzanti i semafori A-F tutti inizializzati a 1
- Esiste una sequenza temporale di esecuzione di P1, P2 e P3 tale per cui si crei una situazione di stallo ?
- Motivare la risposta

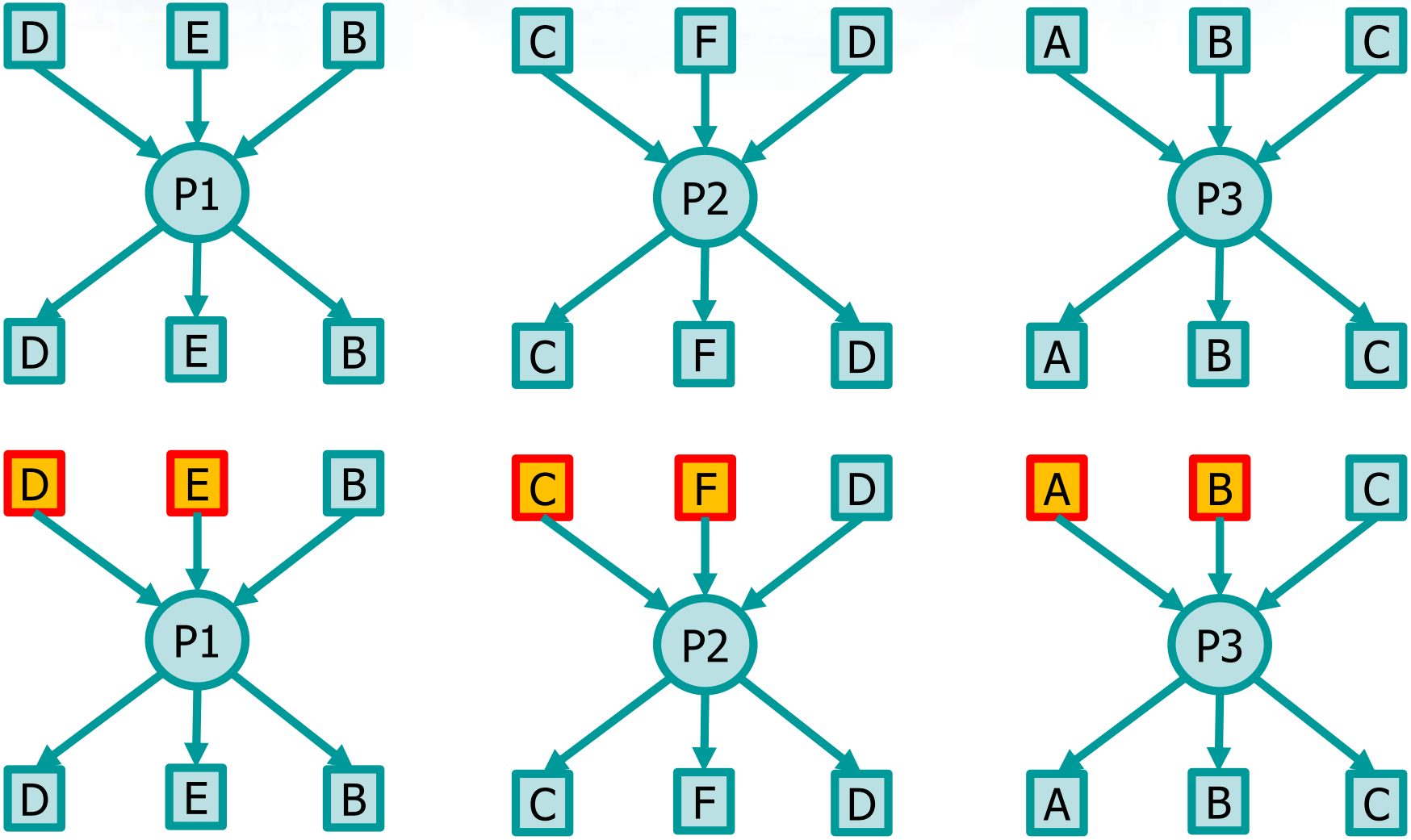
Esercizio

```
P1:
while (1) {
  wait (D);
  wait (E);
  wait (B);
  printf ("P1\n");
  signal (D);
  signal (E);
  signal (B);
}
```

```
P2:
while (1) {
  wait (C);
  wait (F);
  wait (D);
  printf ("P2\n");
  signal (C);
  signal (F);
  signal (D);
}
```

```
P3:
while (1) {
  wait (A);
  wait (B);
  wait (C);
  printf ("P3\n");
  signal (A);
  signal (B);
  signal (C);
}
```

Soluzione



Esercizio

- Un file, di lunghezza indefinita e di formato ASCII, contiene un elenco di interi
- Si scriva un programma che, una volta ricevuto un valore k (intero) e una stringa sulla riga di commando, generi k thread e li attenda
- Ciascun thread
  - Legge il file in concorrenza e effettua la somma dei valori letti
  - Una volta raggiunta la fine del file, visualizza il numero di valori interi letti e la somma calcolata
  - Termina

Esercizio

- Una volta terminati tutti i thread, il thread principale deve visualizzare il numero totale di valori letti e la loro somma totale
- Esempio

Formato file: file.txt

```
7
9
2
-4
15
0
3
```

Esempio di esecuzione

```
> pgrm 2 file.txt
Thread 1: Sum=18 #Line=4
Thread 2: Sum=14 #Line=3
Total   : Sum=32 #Line=7
```

Soluzione

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <semaphore.h>
#include <pthread.h>

#define L 100
struct threadData {
  pthread_t threadId;
  int id;
  FILE *fp;
  int line;
  int sum;
};
static void *readFile (void *);
sem_t sem;
```

Inclusioni, variabili e prototipi

Deve essere unico (ma o è globale o è passato come parametro in questa struttura)

Soluzione

Main
Parte 1

```
int main (int argc,char *argv[]) {
  int i, nT, total, line;
  struct threadData *td;
  void *retval;
  FILE *fp;

  nT = atoi (argv[1]);
  td = (struct threadData *) malloc
       (nT * sizeof (struct threadData));
  fp = fopen (argv[2], "r");
  if (td==NULL || fp==NULL) {
    fprintf (stderr, "Error ...\n");
    exit (1);
  }
  sem_init (&sem, 0, 1);
```

Not shared

Init a 1

Soluzione

Main
Parte 2

File pointer in comune
a tutti i thread

```
  for (i=0; i<nT; i++) {
    td[i].id = i;
    td[i].fp = fp;
    td[i].line = td[i].sum = 0;
    pthread_create (&(td[i].threadId),
      NULL, readFile, (void *) &td[i]);
  }
  total = line = 0;
  for (i=0; i<nT; i++) {
    pthread_join (td[i].threadId, &retval);
    total += td[i].sum;
    line += td[i].line;
  }
  fprintf (stdout, "Total: Sum=%d #Line=%d\n",
    total, line);
  sem_destroy (&sem);
  fclose (fp);
  return (1);
}
```

Soluzione

Thread function

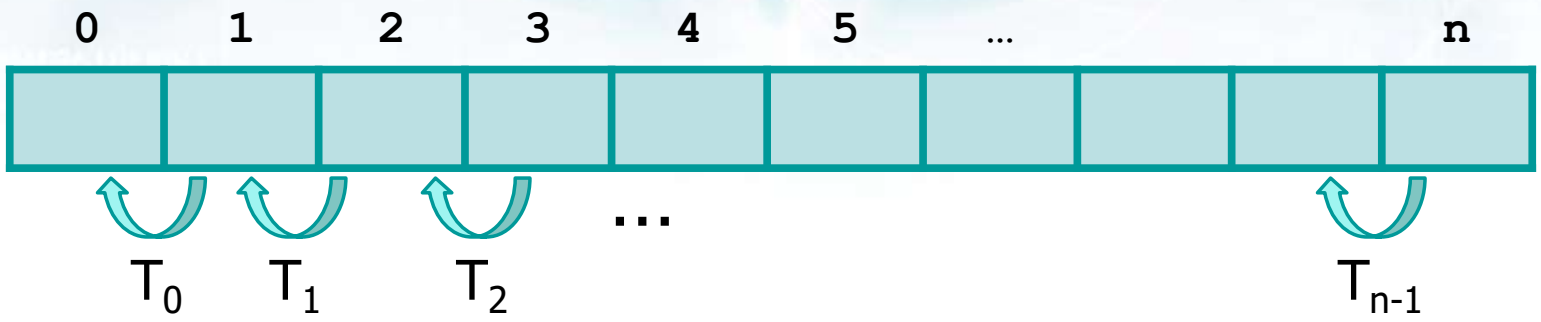
```
static void *readFile (void *arg){
  int n, retVal;
  struct threadData *td;
  td = (struct threadData *) arg;
  do {
    sem_wait (&sem);
    retVal = fscanf (td->fp, "%d", &n);
    sem_post (&sem);
    if (retVal!=EOF) {
      td->line++;
      td->sum += n;
    }
    sleep (1);  // Delay Threads
  } while (retVal!=EOF);
  fprintf (stdout, "Thread: %d Sum=%d #Line=%d\n",
    td->id, td->sum, td->line);
  pthread_exit ((void *) 1);
}
```

Protezione lettura da file

Esercizio

- Un programma concorrente deve utilizzare l'algoritmo di bubblesort (ordinamento per scambio) come segue
  - Un vettore statico include n elementi interi
  - Occorre ordinarlo eseguendo n-1 thread identici
  - Ogni thread è responsabile di due elementi adiacenti
    - Il thread 0 si occupa degli elementi 0 e 1
    - Il thread 1 degli elementi 1 e 2
    - ...
    - Il thread n-1 degli elementi n-1 e n

Esercizio



- Ciascun thread
  - Confronta i due elementi di cui si occupa e li scambia se non si trovano nell'ordine corretto
  - Una volta terminato il proprio lavoro tutti i thread si sincronizzano
  - Se
    - Tutti gli elementi sono ordinati correttamente il programma termina
    - In caso contrario tutti i thread vengono eseguiti nuovamente per effettuare una nuova serie di scambi

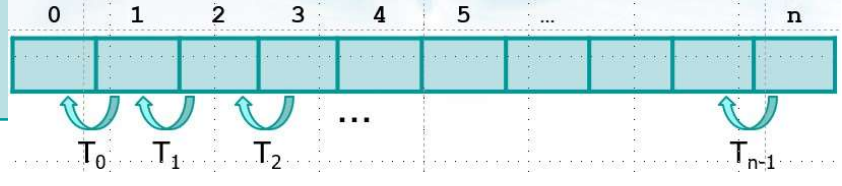
Soluzione 1

```
#include <stdio.h>

typedef enum {false, true} boolean;

int num_threads;
int vet_size;
int *vet;
boolean sorted = false;
boolean all_ok = false;
sem_t semMaster;
sem_t *semSlave;
pthread_mutex_t *me;

static int max_random (int);
void *master (void *);
void *slave (void *);
```

Tipo booleano

Variabili globali:
1 semaforo per il thread master
1 per ogni thread slave
1 mutex per ogni element del vettore

Prototipi

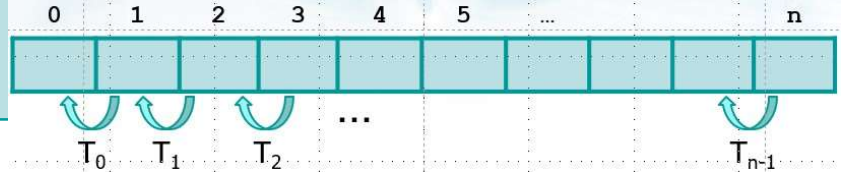


Soluzione 1

Main
Parte 1

```
int main (int argc, char **argv) {

  ... Definizioni ...

  vet_size = atoi (argv[1]);
  num_threads = vet_size - 1;

  ... Allocazioni ...

  for (i=0; i<vet_size; i++) {
    vet[i] = max_random (1000);
  }
  for (i=0; i<vet_size; i++) {
    pthread_mutex_init (&me[i], NULL);
  }
```

Crea 1 mutex per ogni elemento del vettore



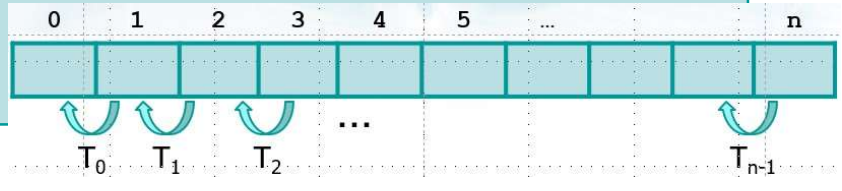
Soluzione 1

Main
Parte 2

```
sem_init (&semMaster, 0, num_threads);
pthread_create (&thMaster, NULL, master, &num_threads);

for (i=0; i<num_threads; i++) {
  id[i] = i;
  sem_init (&semSlave[i], 0, 0);
  pthread_create (&thSlave[i], NULL, slave, &id[i]);
}

for (i=0; i<num_threads; i++) {
  pthread_join (thSlave[i], NULL);
}
pthread_join (thMaster, NULL);

... Free memory and semaphores ...
```

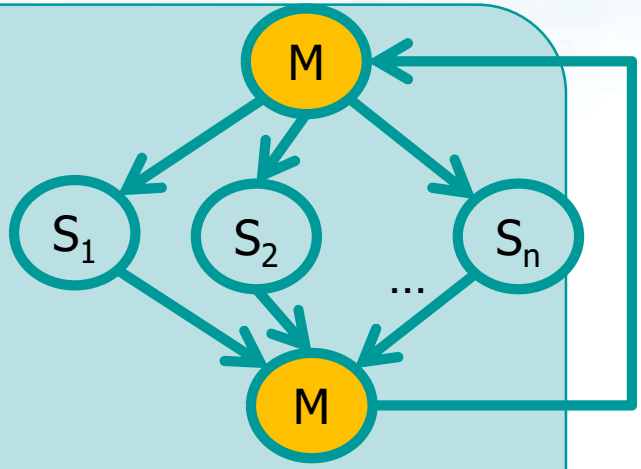
Crea 1 thread master

Crea num_threads thread slave



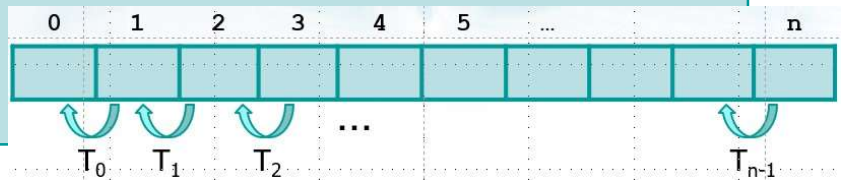
Soluzione 1

```
void *master (void *arg) {
  int *ntp, nt, i;
  ntp = (int *) arg;
  nt = *ntp;
  while (!sorted) {
    for (i=0; i<nt; i++)
      sem_wait (&semMaster);
    if (all_ok) {
      sorted = true;
    } else {
      all_ok = true;
    }
    for (i=0; i<nt; i++)
      sem_post (&semSlave[i]);
  }
  pthread_exit (0);
}
```



Attende thread slave

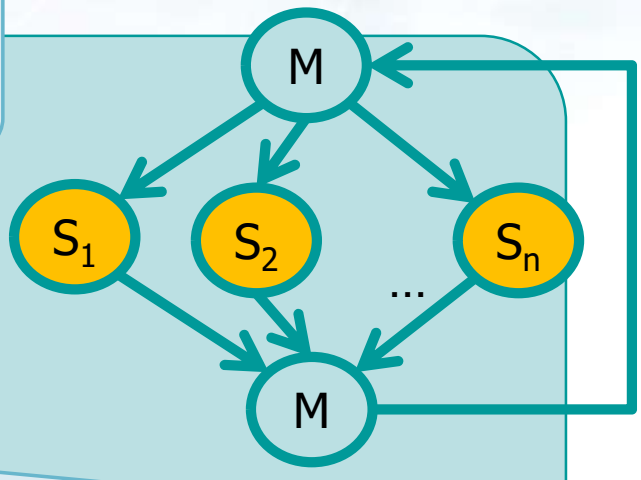
Sveglia thread slave



Soluzione 1

```
void *slave (void *arg) {
  int i = *((int *) arg);
  while (1) {
    sem_wait (&semSlave[i]);
    if (sorted) break;
    pthread_mutex_lock(&me[i]);
    pthread_mutex_lock(&me[i+1]);
    if (vet[i] > vet[i + 1]) {
      swap (vet[i], vet[i + 1]);
      all_ok = false;
    }
    pthread_mutex_unlock(&me[i+1]);
    pthread_mutex_unlock(&me[i]);
    sem_post (&semMaster);
  }
  pthread_exit (0);
}
```

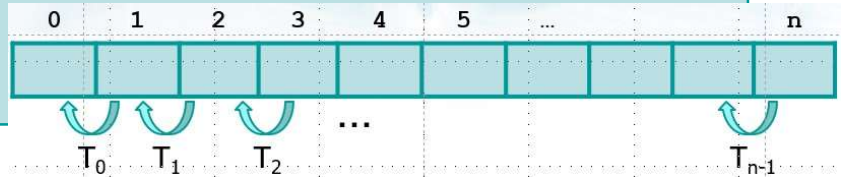
Attende thread master



Acquisisce i 2 elementi di sua competenza

Li ordina

Sveglia thread master



Soluzione 2

```
#include <stdio.h>
#include <sys/timeb.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 10

int count, vet[N];
int sorted = 0;
int all_ok = 1;
sem_t me[N];
sem_t mutex, barrier1, barrier2;
```

Evita l'utilizzo di un semaforo per ciascun slave, utilizzando un barriera

Soluzione 2

```
int main (int argc, char * argv[]) {
  ...
  count = 0;
  sem_init (&mutex, 0, 1);
  sem_init (&barrier1, 0, 0);
  sem_init (&barrier2, 0, 0);

  for (i=0; i<N; i++)
    sem_init (&me[i], 0, 1);

  for (i=0; i<N-1; i++) {
    id[i] = i;
    pthread_create (&th[i], NULL, sorter, &id[i]);
  }
  pthread_exit (0);
}
```

read_array legge o genera il vettore

Crea un mutex di protezione del contatore e 2 barriere semaforiche

Crea un semaforo per ciascun elemento del vettore

Crea N thread

Soluzione 2

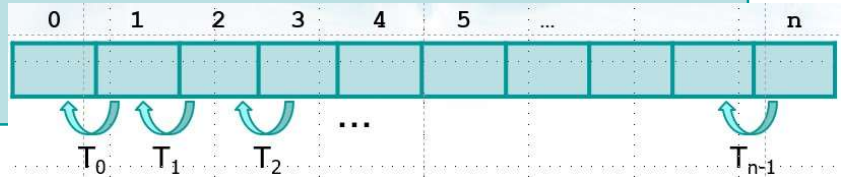
```
static void *sorter (void *arg) {
  int *a = (int *) arg;
  int i, j, tmp;

  i = *a;

  pthread_detach (pthread_self ());

  while (!sorted) {
    sem_wait (&me[i]);
    sem_wait (&me[i+1]);
    if (vet[i] > vet[i+1]) {
      swap (vet[i], vet[i + 1]);
      all_ok = 0;
    }
    sem_post (&me[i + 1]);
    sem_post (&me[i]);
```

Acquisisce i 2 elementi di sua competenza

Li riordina

all_ok rimane a 1 se nessun thread effettua uno scambio

Rilascia i valori del vettore



Soluzione 2

Barriera #1

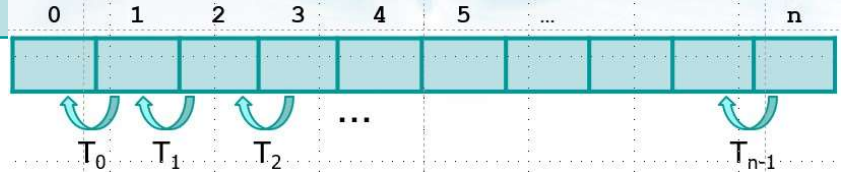
```
sem_wait (&mutex);
count++;
if (count == N-1) {
  for (j=0; j<N-1; j++)
    sem_post (&barrier1);
}
sem_post (&mutex);

sem_wait (&barrier1);
```

Prima di iterare, occorre sincronizzare tutti i thread

L'ultimo thread ad arrivare libera tutti

Tutti gli altri thread si mettono in attesa su una barriera

Mutex protegge count



Soluzione 2

```
    sem_wait (&mutex);
    count--;
    if (count == 0) {
      printf ("all_ok %d\n", all_ok);
      for (j=0; j<N; j++)
        printf ("%d ", vet[j]);
      printf ("\n");
      if (all_ok)
        sorted = 1;
      all_ok = 1;
      for (j=0; j<N-1; j++)
        sem_post (&barrier2);
    }
    sem_post (&mutex);
    sem_wait (&barrier2);
  }
  return 0;
}
```

Barriera #2

Una barriera sola non basta, perchè l'ultimo sveglia tutti e un thread veloce potrebbe iterare più volte

Quindi si realizza una seconda barriera

Riparte (se necessario)

Blocca tutto

L'ultimo thread ad arrivare libera tutti

Tutti gli altri thread si mettono in attesa sulla barriera