Esercitazione n.1
23 Ottobre 2017

Obiettivi:

- Introduzione alla programmazione **`pthreads`**:
  - Gestione thread posix:
    - creazione: pthread_create
    - terminazione: pthread_exit
    - join: pthread_join
  - Sincronizzazione thread posix:
    - mutua esclusione: pthread_mutex
    - semafori

Richiami sui thread

Processi & thread

Il concetto di processo e` basato su **due aspetti indipendenti:**

- **Possesso delle risorse** contenute nel suo spazio di indirizzamento.
- **Esecuzione**. Flusso di esecuzione, associato a un programma, che compete per l'uso della PU con altri flussi, possiede uno stato e viene messo in esecuzione sulla base della politica di scheduling.

- I due aspetti sono indipendenti e possono essere gestiti separatamente dal S.O.:
  - processo **leggero** (***thread***): elemento cui viene assegnata la CPU
  - processo **pesante** (***processo o task***): elemento che possiede le risorse

Un ***thread*** rappresenta un ***flusso di esecuzione*** all'interno di un *processo pesante*.

- **Multithreading**: molteplicità di flussi di esecuzione all'interno di un processo pesante.

- Tutti i thread definiti in un processo ***condividono*** le risorse del processo, risiedono nello **stesso spazio di indirizzamento** ed hanno **accesso a dati comuni**.

Ogni thread ha:

- uno **stato** di esecuzione (running, ready, blocked)
- un **contesto** che è salvato quando il thread non è in esecuzione
- uno **stack** di esecuzione -> uno spazio di memoria privato per le **variabili locali**
- accesso alla memoria e alle risorse del task **condiviso** con gli altri thread.

Vantaggi

- **maggiore efficienza**: le operazioni di *context switch*, creazione etc., sono mediamente più leggere rispetto al caso dei processi.
- maggiori possibilità di sfruttamento del parallelismo in architetture **multiprocessore**.

La libreria pthread per lo sviluppo di applicazioni concorrenti multithreaded

Uso dei thread in Linux: system call native vs. pthread

- Processi leggeri **realizzati a livello kernel**
- System call **clone**:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
     flags, void *arg)
```

➡ **E` specifica di Linux: scarsa portabilita`!**

Libreria pthread

offre funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):

- *Creazione/terminazione threads*
- *Sincronizzazione threads:lock, [semafori], variabili condizione*
- *Etc.*

➢ **Portabilita`**

LinuxThreads

Linuxthreads è l'implementazione di pthread in ambiente GNU/linux.

Caratteristiche thread:

- Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione)
- l' esecuzione di un programma determina la creazione di un thread iniziale che esegue il codice del main (a differenza di POSIX: non c' è *task*).
- Il thread iniziale può creare altri thread: si crea una gerarchia di thread che condividono uno spazio di indirizzi.
- I thread vengono creati all'interno di un processo per eseguire una funzione
- ogni thread ha il suo PID (distinzione tra *task* e *threads*)
- Gestione dei segnali non conforme a POSIX (Linuxthread):
  - Non c'e` la possibilita` di inviare un segnale a un task.
  - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads e quindi non sono piu` disponibili.

Rappresentazione dei threads

Il thread e` l'unita` di scheduling, ed e` univocamente individuato da un indentificatore (TID, long unsigned):

```
pthread_t tid;
```

Il tipo **`pthread_t`** e`dichiarato nell'**header file:**

```
<pthread.h>
```

Creazione di thread: `pthread_create`

Creazione di thread:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

Dove:

- **`thread`**: e` il puntatore alla variabile che raccogliera` il thread_ID (TID)
- **`start_routine`**: e` il puntatore alla funzione che contiene il codice del nuovo thread
- **`arg`**: e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire
- **`attr`**: puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):
  - ad esempio parametri di scheduling: priorita` etc. (solo per superuser)
  - Legame con gli altri threads (ad esempio: *detached* o no)

Restituisce : 0 in caso di successo, altrimenti un codice di errore (!=0)

Creazione di threads

Ad esempio:

```
int A, B; /* variabili comuni ai thread che verranno creati*/
void * codice(void *){ /*definizione del codice del thread */ …}
main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

➔ **Vengono creati due thread (di tid `t1` e `t2`) che eseguono le istruzioni contenute nella funzione `codice`:**

- **I due thread appartengono allo stesso *task* (processo) e condividono le variabili globali del programma che li ha generati (ad esempio A e B).**

Tid: pthread_self

```
pthread_t pthread_self (void);
```

restituisce l'identificatore del thread che la chiama.

Esempio:

```
pthread_t self_id;
self_id=pthread_self();
printf("Sono il thread %lu del processo %d!\n", self_id,getpid());
```

Terminazione di thread: `pthread_exit`

Terminazione di thread:

```
void pthread_exit(void *retval);
```

Dove **`retval`** e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. **`pthread_join`**).

E` una chiamata senza ritorno.

Alternativa: **`return();`**

pthread_join

Un thread puo` sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void **thread_return);
```

Dove:

- `th`: **e` il pid del particolare thread da attendere**
- `thread_return`: **se** `thread_return` **non è NULL, in** `*thread_return` **viene memorizzato il valore di ritorno del thread (v. parametro** `pthread_exit`**)**

Il valore restituto dalla pthread_join indica l'esito della chiamata: se diverso da zero significa che la pthread_join e` fallita (ad es. non vi sono thread figli).

Terminazione di threads

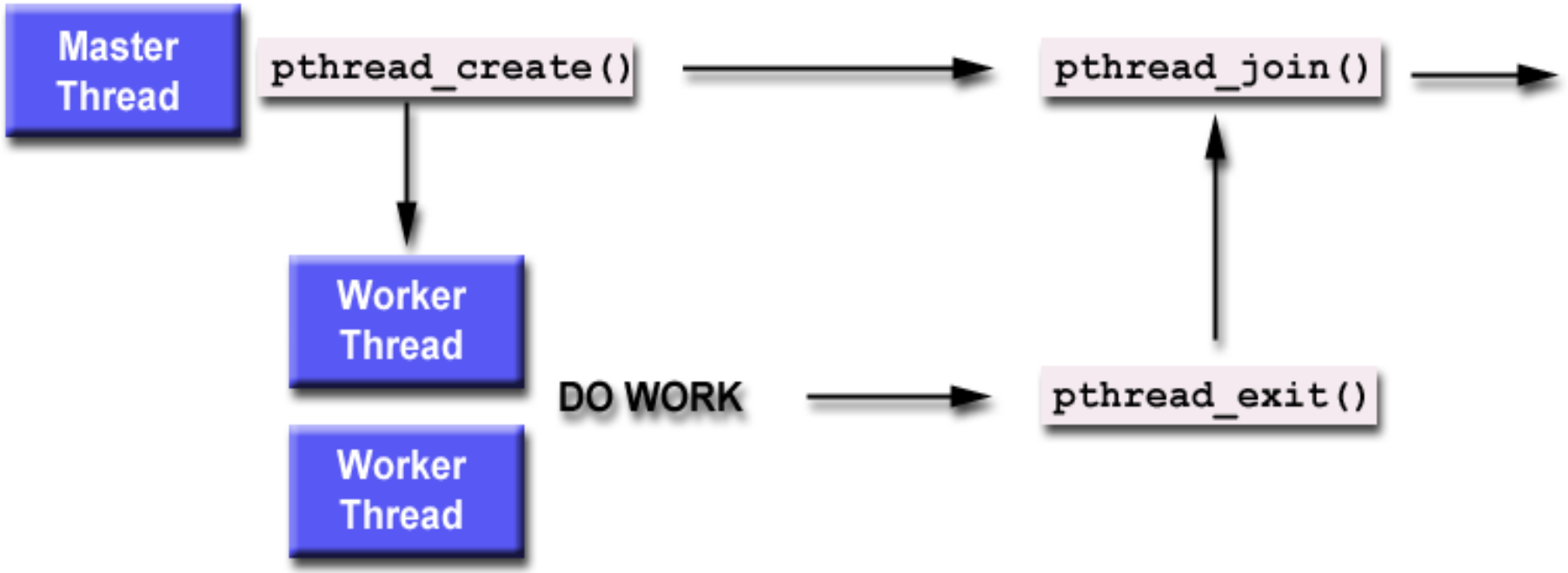
- Normalmente e` necessario che il thread “padre” esegua la **`pthread_join`** per ogni thread figlio che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

- In alternativa si puo` “staccare” il thread dagli altri con:

```
int pthread_detach(pthread_t th);
```

  - La join viene eseguita automaticamente dal sistema: il thread rilascia automaticamente le risorse assegnatagli quando termina.

Modello master/workers



Il modello si presta alla soluzione di problemi secondo lo schema *divide-and-conquer*.

Esempio: 4 workers

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define NUM_THREADS 4
void *Calcolo(void *t) // codice worker
{  int i;
   long tid, result=0;
   tid = (int)t;
   printf("Thread %ld è partito...\n",tid);
   for (i=0; i<100; i++)
      result = result + tid; //elaborazione...
  printf("Thread %ld ha finito. Ris= %ld\n",tid, result);
   pthread_exit((void*) result);
}
```

```
int main (int argc, char *argv[])
{  pthread_t thread[NUM_THREADS];
   int rc;
    long t;
    long status;
    for(t=0; t<NUM_THREADS; t++) {
         printf("Main: creazione thread %ld\n", t);
         rc=pthread_create(&thread[t], NULL, Calcolo, (void *)t);
         if (rc) {
             printf("ERRORE: %d\n", rc);
             exit(-1);   }
     }

   for(t=0; t<NUM_THREADS; t++) {
         rc = pthread_join(thread[t], (void *)&status);
         if (rc)
             printf("ERRORE join thread %ld codice %d\n", t, rc);
        else
             printf("Finito thread %ld con ris. %ld\n",t,status);
     }
}
```

Compilazione

Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c  -lpthread
```

- `D_REENTRANT` **-> esecuzione "thread-safe":**
  - **Gestione corretta del buffer di I/O condiviso (mutua esclusione tra thread)**
  - **Gestione corretta di variabili di sistema**
  - **Gestione corretta degli errori (1 errno/thread)**

Esercizio 1 - calcolo "parallelo" del massimo di un insieme di interi

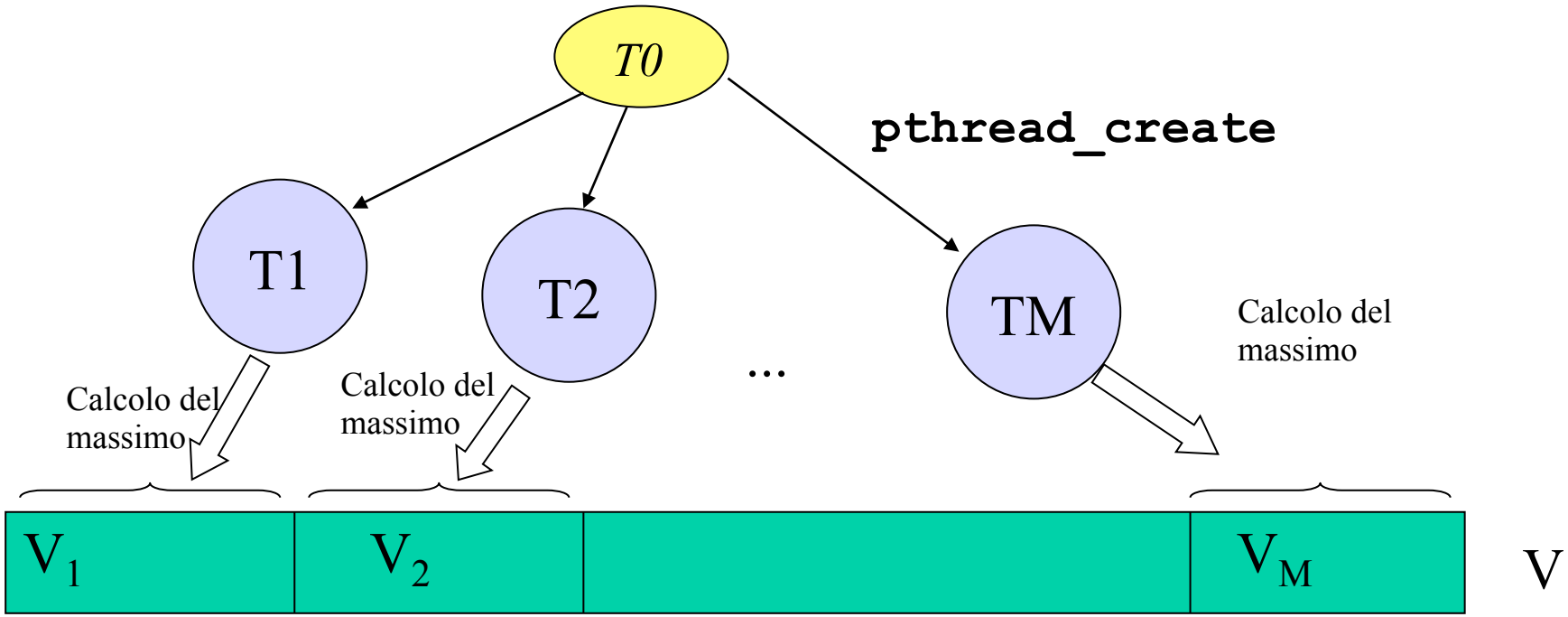
Si **calcoli il massimo in un insieme** di valori interi di N elementi, memorizzati in un vettore V.

Si vuole affidare la ricerca del massimo a un insieme di **M thread concorrenti**, ognuno dei quali si dovrà occupare della ricerca del massimo in una porzione del vettore di dimensione K data (M=N/K).

Il thread iniziale dovrà quindi:

- Inizializzare il vettore V con N valori (casuali o letti da stdin);
- Creare gli M thread concorrenti;
- Ricercare il massimo tra gli M risultati ottenuti dai thread e stamparne il valore.

Impostazione



Strumenti di sincronizzazione nella libreria LinuxThread

I semafori nelle librerie `pthread` e `LinuxThreads`

- La libreria **`pthread`** definisce soltanto il ***semaforo di mutua esclusione*** (mutex)
- La Libreria Linuxthread, implementa comunque il semaforo esternamente alla libreria pthread, in modo conforme allo standard POSIX 1003.1b (la prima versione dello standard POSIX non prevedeva il semaforo)

pthread: MUTEX

- Lo libreria pthread <**pthread.h**>) definisce i **mutex:**
  - equivalgono a semafori il cui valore puo` essere 0 oppure 1 (semafori *binari*);
  - vengono utilizzati tipicamente per risolvere problemi di **mutua esclusione** (ma non solo)
  - **operazioni fondamentali:**
    - **inizializzazione:** `pthread_mutex_init`
    - **Locking (v. operazione p):** `pthread_mutex_lock`
    - **Unlocking (v. operazione v):** `pthread_mutex_unlock`
  - **Per operare sui mutex:**

    `pthread_mutex_t` : tipo di dato associato al mutex; esempio:

    ```
    pthread_mutex_t mux;
    ```

MUTEX: inizializzazione

L'inizializzazione di un **mutex** si puo`realizzare con:

```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr)
```

attribuisce un valore iniziale all'intero associato al semaforo (default: libero):

- **mutex : individua il mutex da inizializzare**
- **attr : punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero* (default).**

– in alternativa , si puo` inizializzare il mutex a default con la macro:

PTHREAD_MUTEX_INIZIALIZER

esempio: `pthread_mutex_t mux= PTHREAD_MUTEX_INIZIALIZER;`

MUTEX: lock/unlock

- **Locking/unlocking si realizzano con:**

```
int pthread_mutex_lock(pthread_mutex_t* mux)
int pthread_mutex_unlock(pthread_mutex_t* mux)
```

  - lock: se il mutex **mux** e` occupato,il thread chiamante si sospende; altrimenti occupa il mutex.
  - unlock: se vi sono processi in attesa del mutex, ne risveglia uno; altrimenti libera il mutex.

Esempio: mutua esclusione

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso tra threads */
int DATA=0; /* variabile condivisa */
int accessi1=0; /*num. di accessi del thread 1 alla sez crit. */
int accessi2=0; /*num. di accessi del thread 2 alla sez crit. */

void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {
        pthread_mutex_lock(&M); /*prologo */
        accessi1++;
        DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T1: %d\n", accessi1);
        pthread_mutex_unlock(&M); /*epilogo */
    }
    pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{   int k=1;
   while(k)
    {
        pthread_mutex_lock(&M); /*prologo sez. critica */
        accessi2++;
        DATA++;
        k=(DATA>=MAX?0:1);
        printf("accessi di T2: %d\n", accessi2);
        pthread_mutex_unlock(&M); /*epilogo */
    }
    pthread_exit (0);
}
```

```
main()
{ pthread_t th1, th2;
  /* il mutex e` inizialmente libero: */
  pthread_mutex_init (&M, NULL);
   if (pthread_create(&th1, NULL, thread1_process, NULL) < 0)
         { fprintf (stderr, "create error for thread 1\n");
           exit (1);
         }
  if (pthread_create(&th2, NULL,thread2_process,NULL) < 0)
  { fprintf (stderr, "create error for thread 2\n");
    exit (1);
  }
   pthread_join (th1, NULL);
   pthread_join (th2, NULL);
}
```

LinuxThreads: Semafori

Memoria condivisa: uso dei semafori (POSIX.1003.1b)

- Semafori: libreria <semaphore.h>
  - **`sem_init`**: inizializzazione di un semaforo
  - **`sem_wait`**: implementazione di ***P***
  - **`sem_post`**: implementazione di ***V***

- **`sem_t`** : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

Operazioni sui semafori

- **`sem_init`: inizializzazione di un semaforo**

```
int sem_init(sem_t *sem, int pshared,
                unsigned int value);
```

attribuisce un valore iniziale all'intero associato al semaforo:

- **`sem`: individua il semaforo da inizializzare**
- **`pshared` : 0, se il semaforo non e` condiviso tra task, oppure non zero (sempre zero).**
- **`value` : e` il valore iniziale da assegnare al semaforo.**

- **`sem_t`** : tipo di dato associato al semaforo; esempio:

```
static sem_t my_sem;
```

➢ ritorna sempre 0.

Operazioni sui semafori: sem_wait

- **P su un semaforo**

```
int sem_wait(sem_t *sem);
```

dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *P* di Dijkstra:

- se il valore del semaforo e` uguale a zero, sospende il thread chiamante nella coda associata al semaforo; altrimenti ne decrementa il valore.

Operazioni sui semafori: sem_post

- ***V*** **su un semaforo:**

```
int sem_post(sem_t *sem);
```

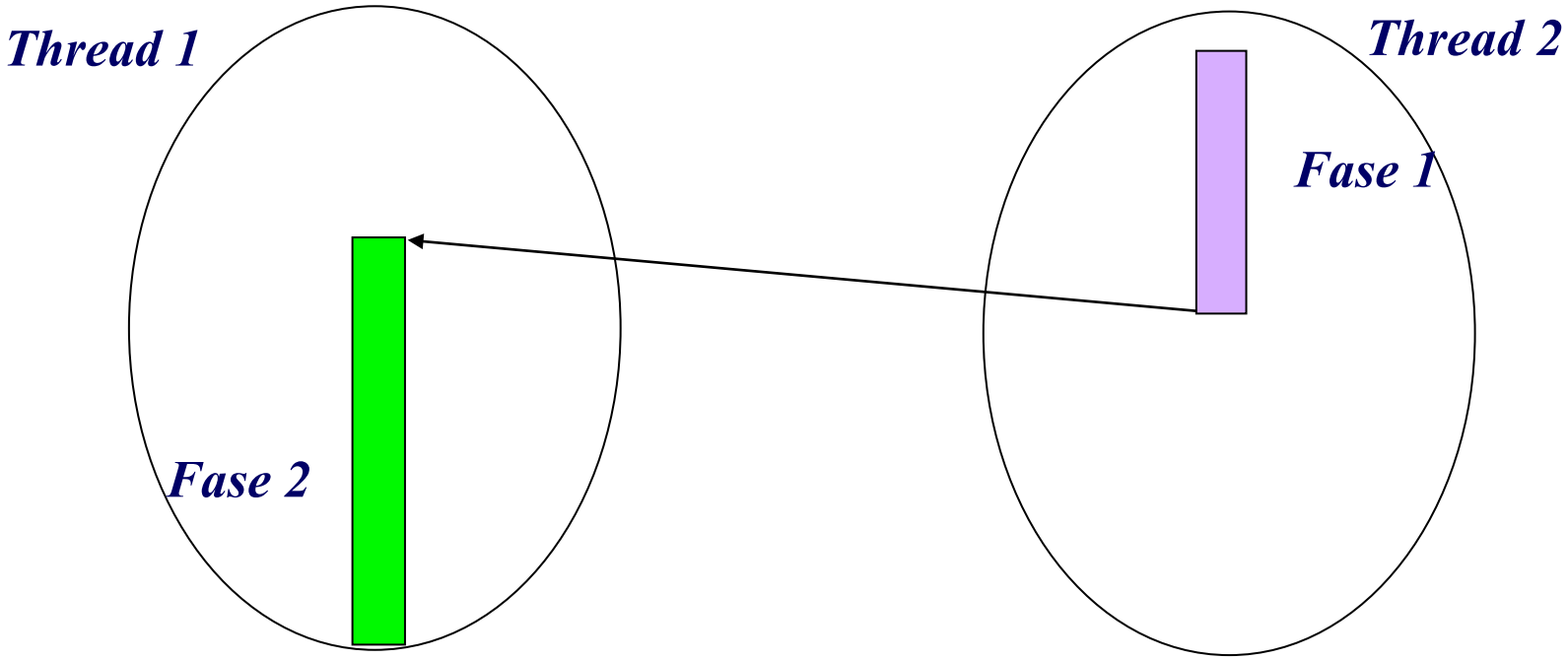
dove:

- `sem`: **individua il semaforo sul quale operare.**

e` la *V* di Dijkstra:

- se c'e` almeno un thread sospeso nella coda associata al semaforo sem, viene risvegliato; altrimenti il valore del semaforo viene incrementato.

Esempio: Semaforo Evento



- **Imposizione di un vincolo temporale: la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread2.**

Esempio: sincronizzazione

```
/* la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread 2*/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

sem_t my_sem;
int V=0;

void *thread1_process (void * arg)
{
  printf ("Thread 1: partito!...\n");
   /* inizio Fase 2: */
  sem_wait (&my_sem);
  printf ("FASE2: Thread 1:  V=%d\n", V);
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{  int i;

    V=99;
     printf ("Thread 2: partito!...\n);
     /* inizio fase 1: */
     printf (“FASE1: Thread 2:  V=%d\n", V);
    /* …
     termine Fase 1: sblocco il thread 1*/
     sem_post (&my_sem);
     sleep (1);
  pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2;
  void *ret;
  sem_init (&my_sem, 0, 0); /* semaforo a 0 */

if (pthread_create (&th1, NULL, thread1_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 1\n");
     exit (1);
  }

  if (pthread_create(&th2,NULL, thread2_process, NULL) < 0)
   {fprintf (stderr, "pthread_create error for thread \n");
     exit (1);
  }

  pthread_join (th1, &ret);
  pthread_join (th2, &ret);
}
```

Esempio:

- `gcc -D_REENTRANT -o sem sem.c -lpthread`

- **Esecuzione**:

```
[aciampolini@ccib48 threads]$ sem
Thread 1: partito!...
Thread 2: partito!...
FASE1: Thread 2:  V=99
FASE2: Thread 1:  V=99
[aciampolini@ccib48 threads]$
```

Esercizio 2 - Mutua esclusione

Una rete televisiva vuole realizzare un sondaggio di opinione su un campione di N persone riguardante il gradimento di K film.

Il sondaggio prevede che ogni persona interpellata risponda a K domande, ognuna relativa ad un diverso film.

In particolare, ad ogni domanda l'utente deve fornire una risposta (un valore intero appartenente al dominio [1,..10]) che esprime il voto assegnato dall'utente al film in questione.

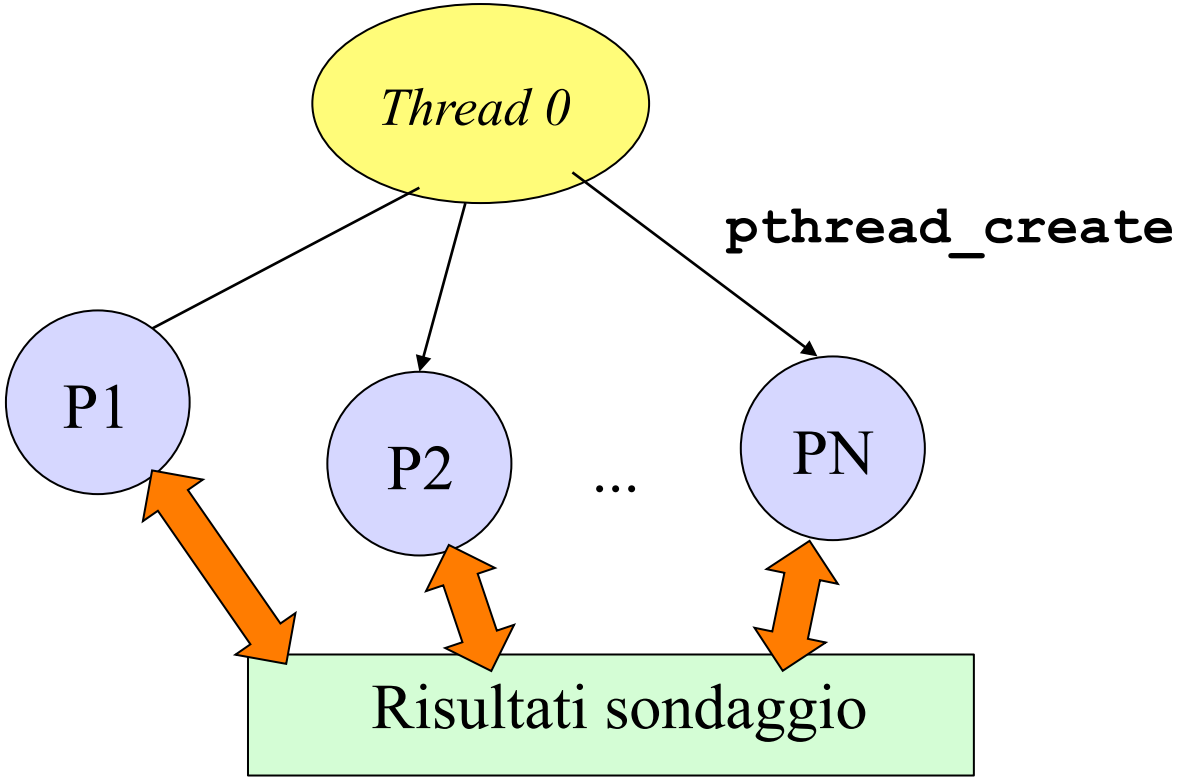
La raccolta delle risposte avviene in modo tale che, al termine della compilazione di ogni questionario, vengano presentati i risultati parziali del sondaggio, e cioè: per ognuna delle k domande, venga stampato il voto medio ottenuto dal film ad essa associato.

Al termine del sondaggio devono essere stampati i risultati definitivi, cioè il voto medio ottenuto da ciascun film ed il nome del film con il massimo punteggio.

Si realizzi un'applicazione concorrente che, facendo uso della libreria pthread e rappresentando ogni singola persona del campione come un thread concorrente, realizzi il sondaggio rispettando le specifiche date.

Spunti & suggerimenti (1)

- Persona del campione= thread
- Risultati del sondaggio: struttura dati **condivisa** composta da K elementi (1 per ogni domanda/film)



MUTUA ESCLUSIONE

- I thread spettatori dovranno accedere in modo mutuamente esclusivo alla variabile che rappresenta i risultati del sondaggio.
- Quale strumenti utilizzare?

  `pthread_mutex` o **`semaphore`**

Esercizio 3 – sincronizzazione a barriera

Si riconsideri il sondaggio di cui all'esercizio 2.

La rete televisiva vuole utilizzare i risultati del sondaggio per stabilire **quale dei K film interessati dalle domande del questionario mandare in onda**, secondo le seguenti modalità.

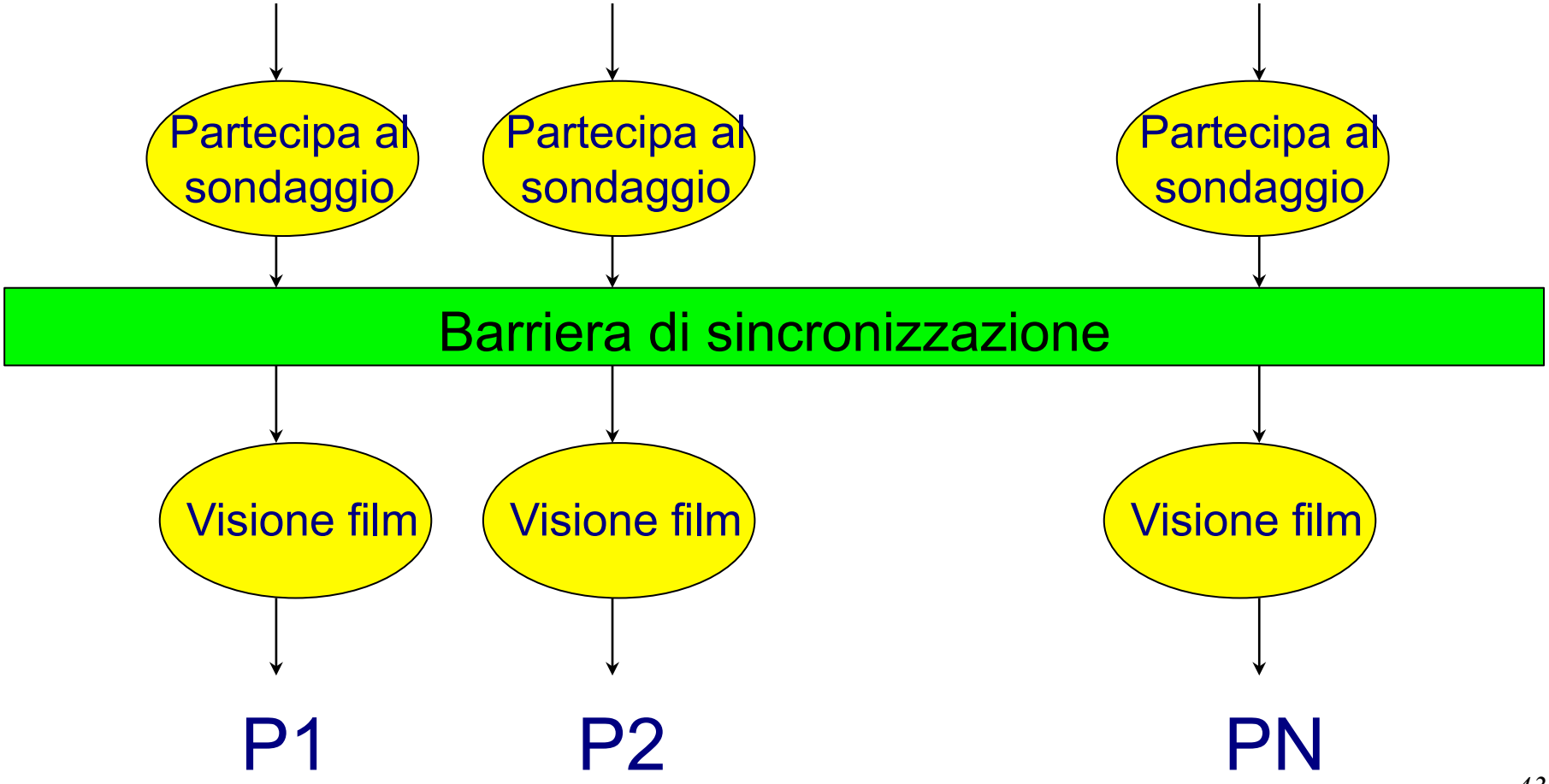
Ognuno degli N utenti ha un comportamento strutturato in **due fasi consecutive**:

1. Nella prima fase partecipa al **sondaggio**
2. Nella seconda fase **vede il film** risultato **vincitore** nel sondaggio (quello, cioè, con la valutazione massima).

Si realizzi un'applicazione concorrente nella quale ogni thread rappresenti un diverso utente, che tenga conto dei vincoli dati e, in particolare, che **ogni utente non possa eseguire la seconda fase** (visione del film vincitore) se prima non si è conclusa la fase precedente (compilazione del questionario) **per tutti gli utenti.**

Spunti & suggerimenti

Rispetto all'esercizio 1 è richiesta l'aggiunta di una **barriera di sincronizzazione** per tutti i thread concorrenti:



Barriera: possibile soluzione (pseudicodice)

- **Variabili condivise:**

```
semaphore mutex=1:
semaphore barriera=0;
int completati=0;
```

Struttura del thread i-simo Pi:

```
<operazione 1 di Pi>

p(mutex);
completati++;
if (completati==N)
   v(barriera);
v(mutex);
p(barriera);
v(barriera);

<operazione 2 di Pi>
```