Esercitazione n.1
25 Marzo 2024

Il modello a memoria comune

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 esegue concorrentemente con altri flussi e compete con essi per l'uso della/e CPU, 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 (task).

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

Il thread è l'unità di esecuzione nel modello a memoria comune.

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 (rispetto ai processi tradizionali , o «pesanti»):

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

La libreria pthread

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.
- Il thread iniziale può creare altri thread: si crea una gerarchia di thread che condividono lo stesso spazio di indirizzi.
- I thread vengono creati all'interno di un processo per eseguire una funzione.
- ogni thread ha il suo TID.
- 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 (default: NULL):
  - ad esempio parametri di scheduling: priorita` etc. (solo per superuser)
  - Legame con gli altri threads (ad esempio: detached o no)

pthread_create

- attr==NULL (default) significa:
  - thread joinable
  - default scheduling policy

Restituisce : 0 in caso di successo, altrimenti un codice di errore (ad esempio, se è stato raggiunto il numero massimo di thread PTHREAD_THREADS_MAX)

pthread_create: passaggio di parametri ai thread

```
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

La pthread_create consente di passare alla funzione eseguita dal thread **un singolo argomento di tipo void *.**

L'argomento passato all'interno della start_routine dovrà essere convertito nel tipo effettivo mediante casting.

Ad esempio:

```
pthread_create(&id, NULL, start_f, (void *) d);
```

…

Nel corpo di **start_f**:

```
…
dato D;
D=(dato *)d;
```

Tid: `pthread_self`

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

restituisce l'identificatore del thread che la chiama.

Esempio:

```
pthread_t self_id; //solitamente: long unsigned integer
self_id=pthread_self();
printf("Sono il thread %lu del processo %d!\n",
self_id,getpid());
```

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).

Terminazione di thread: `pthread_exit`

Un thread (che esegue start_routine) puo` terminare in due modi:

- terminando la funzione start_routine
- chiamando **`pthread_exit()`**

NB: exit(), chiamato da qualsiasi thread, termina il processo, quindi tutti i 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

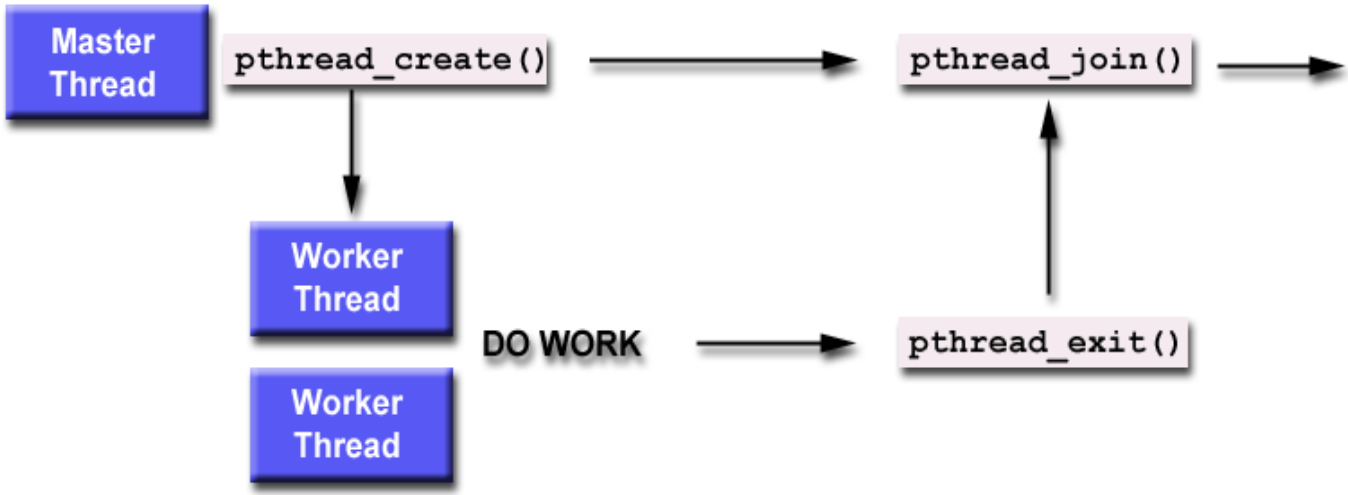
- Normalmente (thread joinable) e` necessario che il thread “padre” esegua la **`pthread_join`** per ogni thread figlio che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

- 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”:

  chiamare contemporaneamente due funzioni della libreria da due thread concorrenti potrebbe non funzionare.
  Causa più frequente: uso interno di variabili globali e simili. -> opzione

  `-D_REENTRANT`

  In questo modo:
  - Gestione corretta degli errori (1 errno/thread)
  - Gestione corretta del buffer di I/O condiviso (mutua esclusione tra thread)

Esempio 1 - calcolo “parallelo” del massimo di un insieme di interi

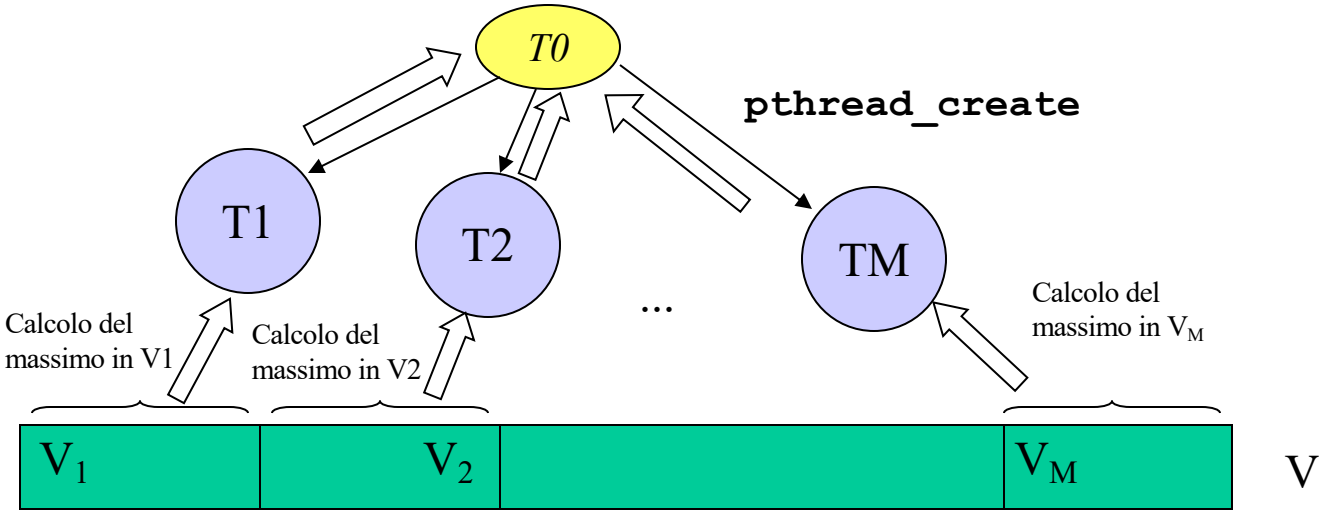
Si calcoli il **massimo** in un insieme di valori interi di N elementi, memorizzati in un vettore V.

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:

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

Impostazione



Esempio: soluzione

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#define N 20 //dimensione matrice
#define K 4 // dim singola frazione

int V[N];

void *Calcolo(void *t) // codice worker
{   int first, result=0;
    first = (int)t;
    for (int i=first; i<first+K; i++)
        if (V[i]>result)
                result=V[i];
    //printf("Worker ha calcolato il massimo locale: %d\n", result);
    pthread_exit((void*) result);
}
```

```
int main (int argc, char *argv[])
{   pthread_t thread[N/K];
    int rc;
    int M, t, first, status, max=0;

    M=N/K; // numero workers
    srand(time(0));
    printf("inizializzazione vettore V:\n");
    for(int i = 0; i < N; i++){ //inizializzazione casuale
     V[i]=1+rand()%200; //numeri casuali tra 1 e 200
     printf("%d\t", V[i]);
    }
    printf("\n");

    for(t=0; t<M; t++) {
        printf("Main: creazione thread n.%d\n", t);
        first=t*K; //indice del primo elemento da elaborare
        rc = pthread_create(&thread[t], NULL, Calcolo, (void *)first);
        if (rc) {
            printf("ERRORE: %d\n", rc);
            exit(-1);   }
    }
```

```
//.. continua T0:

  for(t=0; t<M; 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. %d\n",t,status);
                if (status>max)
                        max=status;
        }
   }
   printf("main-risultato finale: %d\n", max);

}
```

Strumenti di sincronizzazione nella libreria LinuxThread

I semafori nelle librerie `pthread` e `LinuxThreads`

Sincronizzazione tramite semafori:

- La libreria **`pthread`** definisce 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).

Sincronizzazione tramite mutex

La 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)

Definizione di mutex:

```
pthread_mutex_t;
```

Ad esempio:

```
pthread_mutex_t mux;// la var. mux è un mutex
```

Operazioni fondamentali:

- **inizializzazione:** `pthread_mutex_init`
- **Locking (v. operazione p):** `pthread_mutex_lock`
- **Unlocking(v. operazione v):** `pthread_mutex_unlock`

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***

Il tipo di dato associato al semaforo è **sem_t**.

Ad esempio:

```
static sem_t my_sem; //my_sem è un semaforo
```

Operazioni sui semafori

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

➢ ritorna sempre 0.

Operazioni sui semafori: `sem_wait`

Operazione p(sem):

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

dove:

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

e` l'operazione *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`

Operazione v(sem):

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

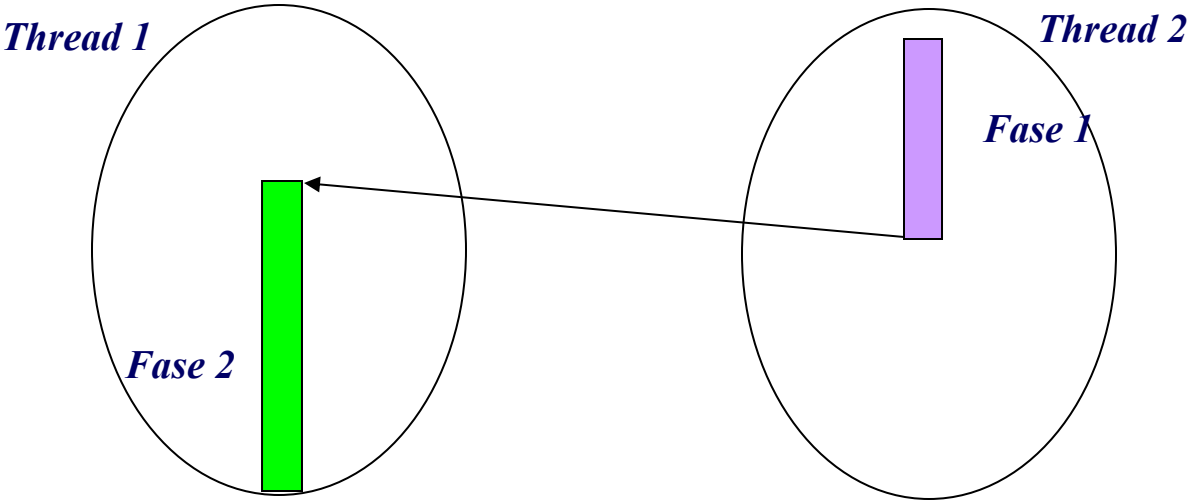
dove:

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

e` l'operazione 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: vincolo di precedenza



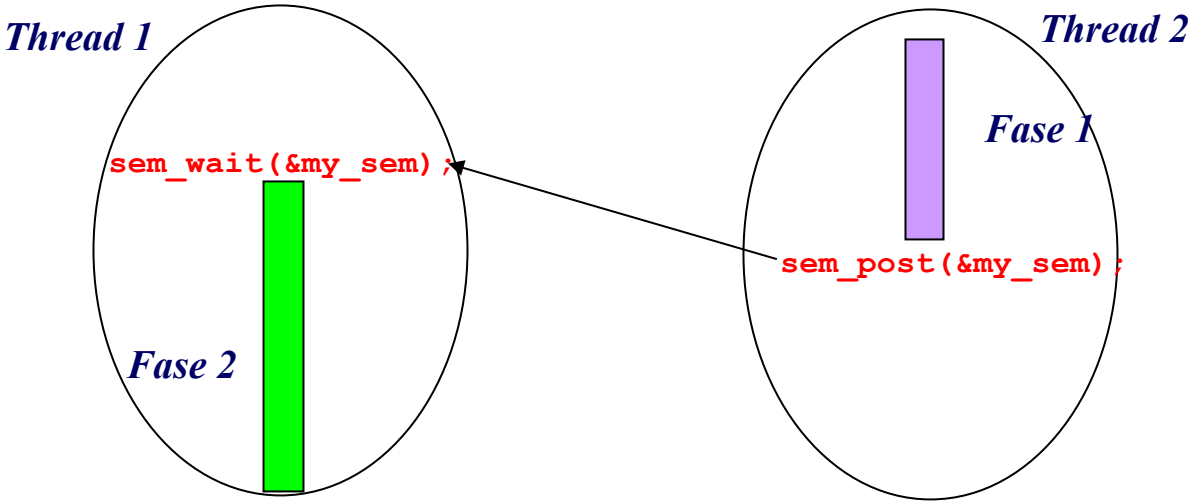
- Imposizione di un vincolo temporale: la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread2 .

soluzione: semaforo evento

Definiamo un semaforo **my_sem** inizializzato a 0:

- Thread 1 esegue **p(my_sem)** prima di fase 2
- Thread 2 esegue **v(my_sem)** dopo fase 1

```
sem_t my_sem;
..
sem_init (&my_sem, 0, 0); /* v. iniziale my_sem = 0 */
```



Semaforo evento

```
/* 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); //operazione "p"
  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); //operazione "v"

     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 1 - 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.

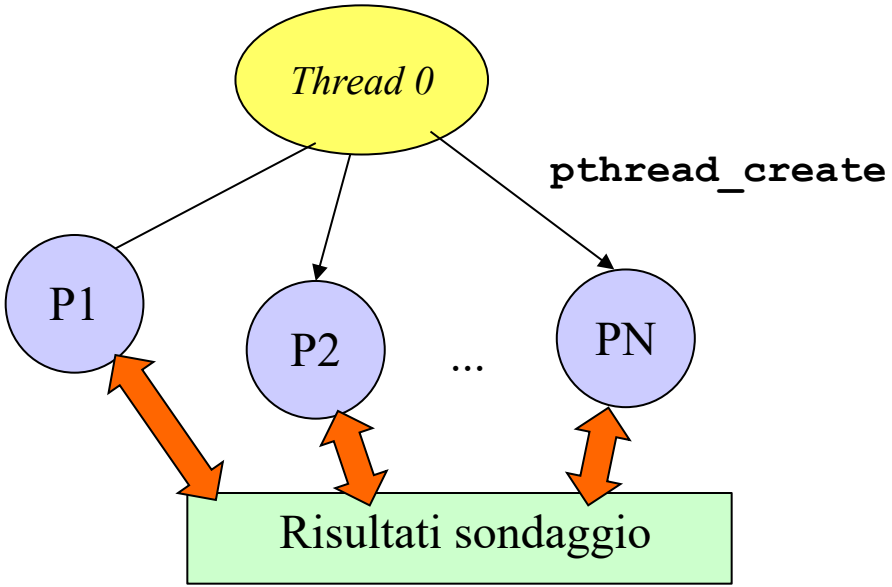
La raccolta delle risposte avviene in modo tale che, al termine della compilazione di ogni questionario, vengano presentati i risultati parziali del sondaggio, e cioè: per ognuna delle k domande, venga stampato il voto medio ottenuto dal film ad essa associato.

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
- Accumulo risultati del sondaggio: struttura dati **condivisa** composta da K elementi (1 per ogni domanda/film)



MUTUA ESCLUSIONE

I thread spettatori dovranno **accedere alla struttura dati** che rappresenta i risultati del sondaggio **in modo mutuamente esclusivo**.

Quale strumenti utilizzare?

pthread_mutex o semaphore

Rappresentazione struttura dati condivisa (esempio):

```
typedef struct{
        char film[K][40];
        int voti[K];
        int pareri;
        pthread_mutex_t m;
} sondaggio;
```

Esercizio 2 – sincronizzazione a barriera

Si riconsideri il sondaggio di cui all'esercizio 1.

La rete televisiva vuole utilizzare i risultati del sondaggio per stabilire quale dei K film interessati dalle domande del questionario rendere disponibile per il download, secondo le seguenti modalità.

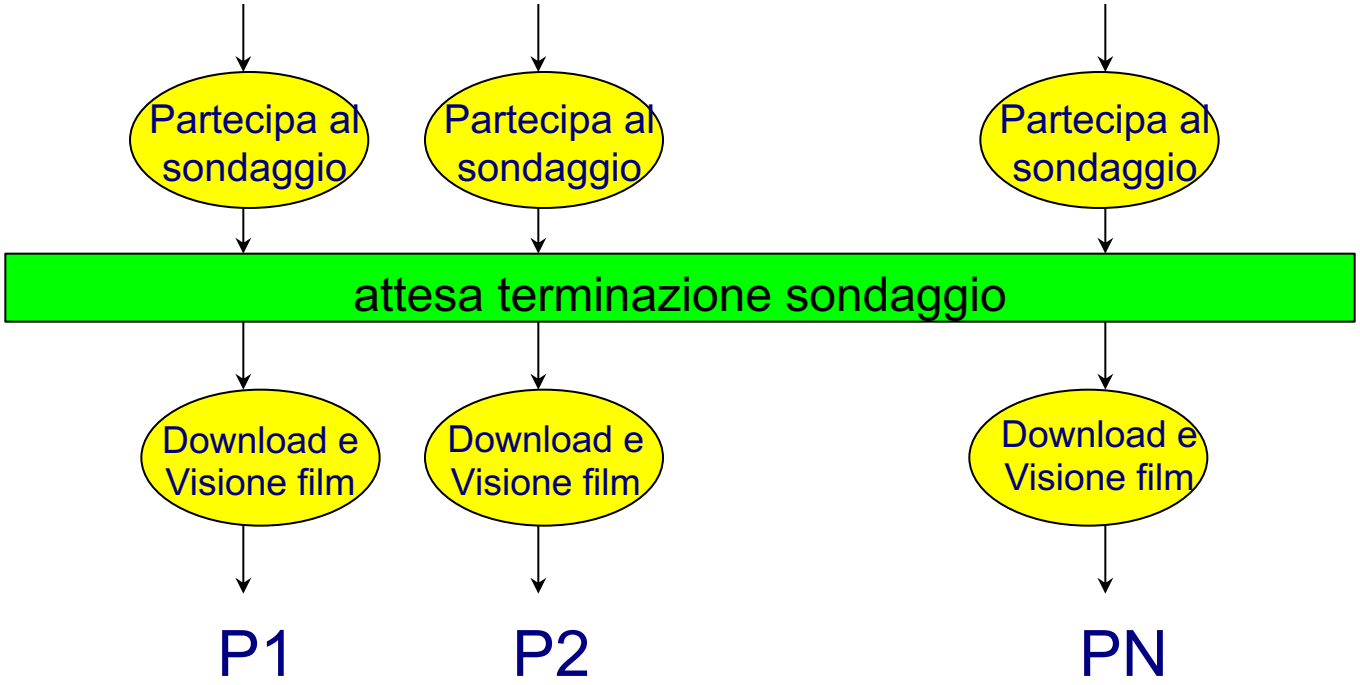
Ognuno degli N utenti ha un comportamento strutturato in due fasi consecutive:

1. Nella prima fase **partecipa al sondaggio**
2. Nella seconda fase **scarica e 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 (download e visione del film vincitore) se prima non si è conclusa la fase precedente (compilazione del questionario) per tutti gli utenti.

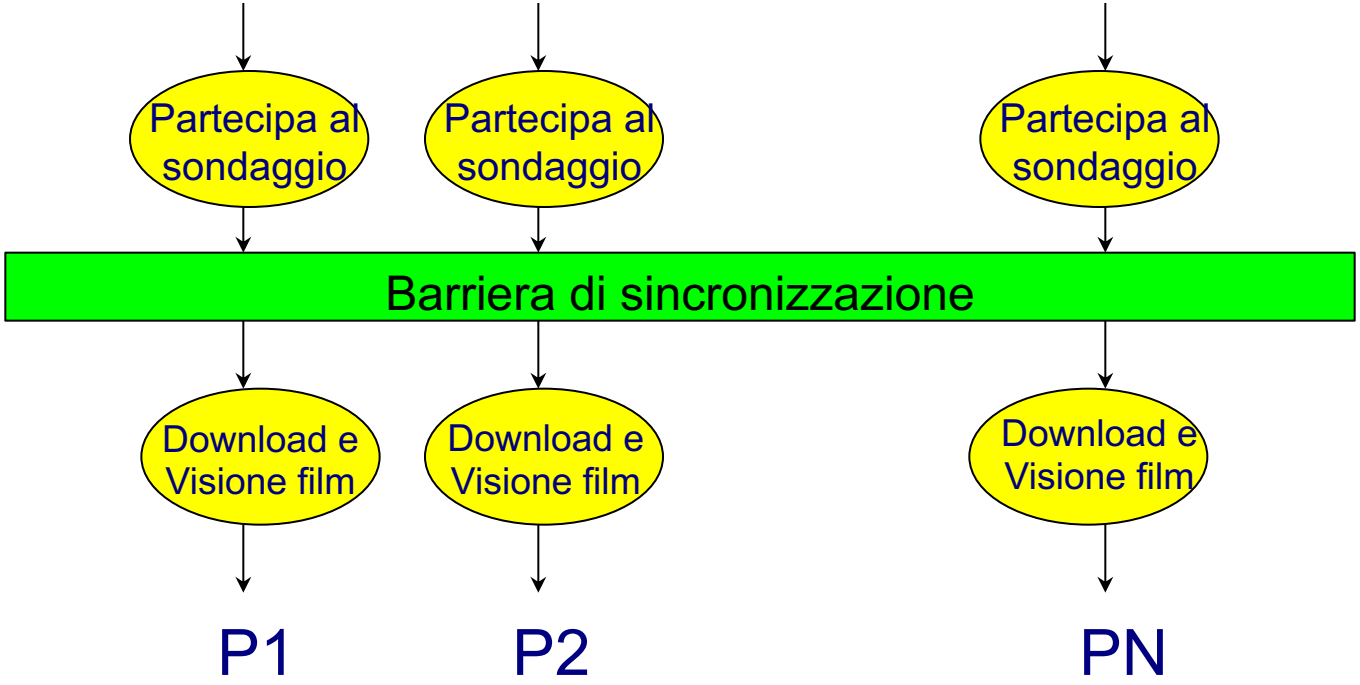
Spunti & suggerimenti

Estensione dell'esercizio 2: tutti i thread concorrenti devono attendere la terminazione del sondaggio prima di poter passare alla fase successiva.



Barriera di sincronizzazione

in ogni thread il passaggio dalla prima fase alla seconda può essere realizzato tramite l'introduzione di «**barriera di sincronizzazione**», realizzata tramite semafori.



Barriera di sincronizzazione

Struttura del thread i-simo Pi:

<partecipazione al sondaggio>

<BARRIERA>

<download e visione film>

Come implementare la barriera di sincronizzazione utilizzando i pthread?

- **Variabili condivise:**

```
sem_t mutex; //semaforo di mutua esclusione: Val.Iniz. 1
sem_t barriera; // semaforo inizializzato a 0
int completati=0; // numero dei thread che hanno completato la prima fase
```

... (v. barriera nel modello a memoria comune)..