Esercitazione 2
Sincronizzazione con semafori

31 Ottobre 2013

Strumenti di sincronizzazione nella libreria LinuxThread e nel linguaggio Java

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, conformemente allo standard POSIX 1003.1b

pthread: MUTEX

- Lo standard POSIX 1003.1c (libreria **<pthread.h>**) definisce i **semafori binari** (o lock, mutex, etc.)
  - sono semafori il cui valore puo` essere 0 oppure 1 (*occupato* o *libero*);
  - vengono utilizzati tipicamente per risolvere problemi di **mutua esclusione**
  - **operazioni fondamentali:**
    - **inizializzazione**: `pthread_mutex_init`
    - **locking**: `pthread_mutex_lock`
    - **unlocking**: `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

```
#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 critica */
int accessi2=0; /*num. di accessi del thread 2 alla sez critica */

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);
}
```

Esempio

```
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 sez. critica*/
    }
    pthread_exit (0);
}
```

Esempio:

```
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);
}
```

Test

```
$
$ gcc -D_REENTRANT -o tlock lock.c -lpthread
$ ./tlock
accessi di T2: 1
accessi di T1: 1
accessi di T2: 2
accessi di T1: 2
accessi di T1: 3
accessi di T1: 4
accessi di T1: 5
accessi di T1: 6
accessi di T1: 7
accessi di T1: 8
accessi di T2: 3
$
```

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);
```

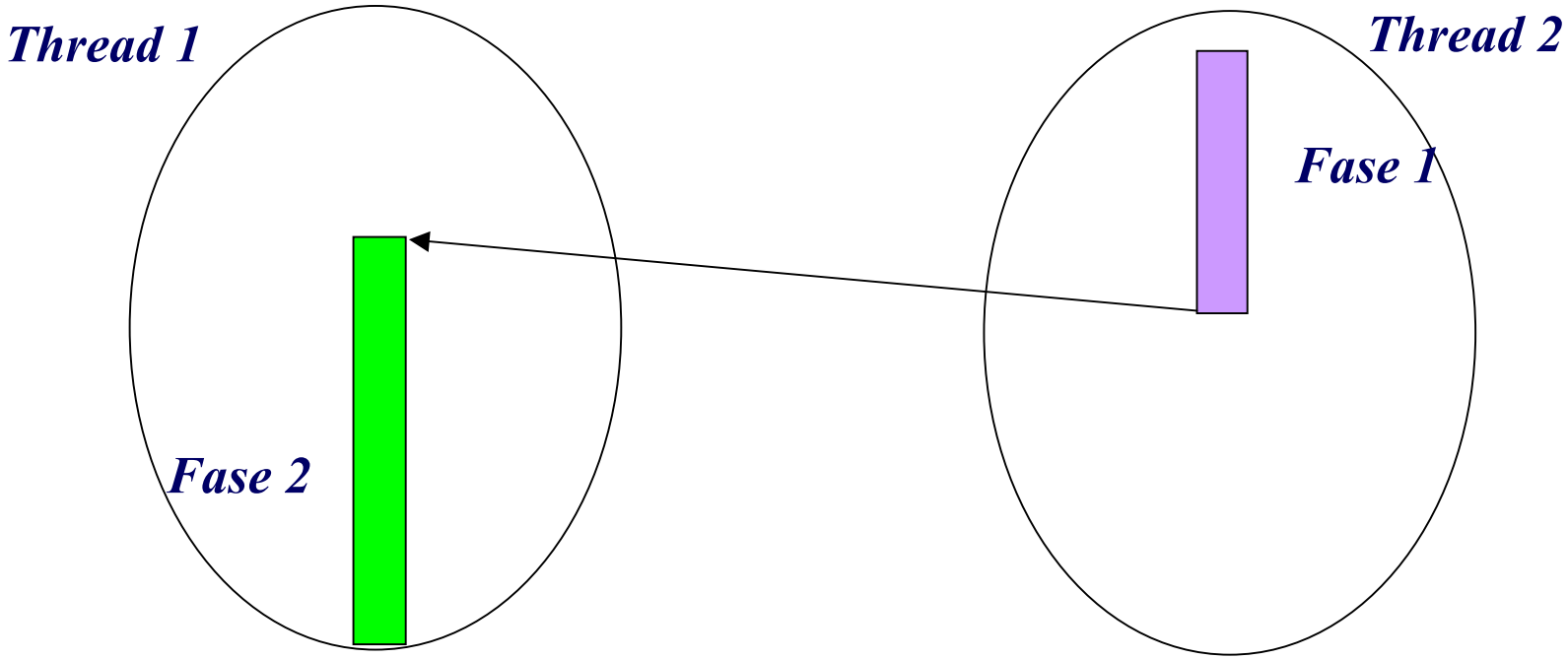
dove:

- **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]$
```

Semafori binari composti: esempio

```
/* tre processi che, ciclicamente, incrementano a
   turno (in ordine P1,P2,P3) la variabile V*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX 13
static sem_t s1,s2,s3; /* semafori per imporre
            l'ordine di accesso (P1,P2,P3) alla
            variabile V */
int V=0,F=0;
```

```
void *thread1_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s1);
       if (V<MAX)
                V++;
        else
        {       k=0;
                printf("T1: %d (V=%d)\n",++F, V);
        }
       sem_post(&s2);
    }
  pthread_exit (0);
}
```

```
void *thread2_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s2);
       if (V<MAX)
                V++;
         else
         {       k=0;
                 printf("T2: %d (V=%d)\n",++F, V);
         }
       sem_post(&s3);
    }
  pthread_exit (0);
}
```

```
void *thread3_process (void * arg)
{   int k=1;
    while(k)
    {   sem_wait (&s3);
        if (V<MAX)
                V++;
         else
        {       k=0;
                printf("T3: %d (V=%d)\n",++F, V);
        }
        sem_post(&s1);
    }
   pthread_exit (0);
}
```

```
main ()
{ pthread_t th1, th2,th3;

  sem_init(&s1,0,1);
  sem_init(&s2,0,0);
  sem_init(&s3,0,0);
  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 2\n");
     exit (1);
  }
  if (pthread_create(&th3,NULL,thread3_process, NULL) < 0)
   { fprintf (stderr, "pthread_create error for thread 3\n");
     exit (1);
  }
```

```
    pthread_join (th1, NULL);
    pthread_join (th2, NULL);
    pthread_join (th3, NULL);

}
```

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 richiede 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 (appartenente al dominio [1,..10]) che esprime il voto assegnato dall'utente al film in questione.

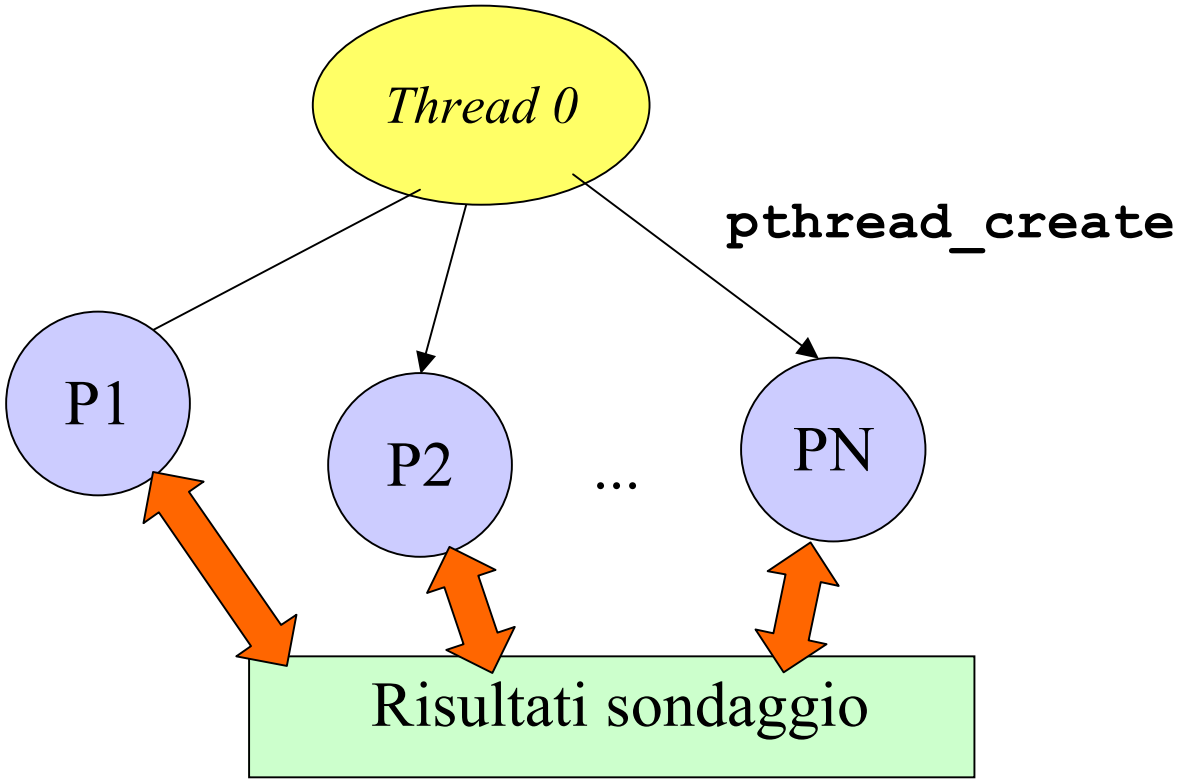
La raccolta delle risposte avviene in modo tale che, al termine della compilazione di ogni questionario, vengano presentati i risultati parziali del sondaggio, e cioè: per ognuna delle k domande, venga stampato il voto medio ottenuto dal film ad essa associato.

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`

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 mandare in onda**, secondo le seguenti modalità.

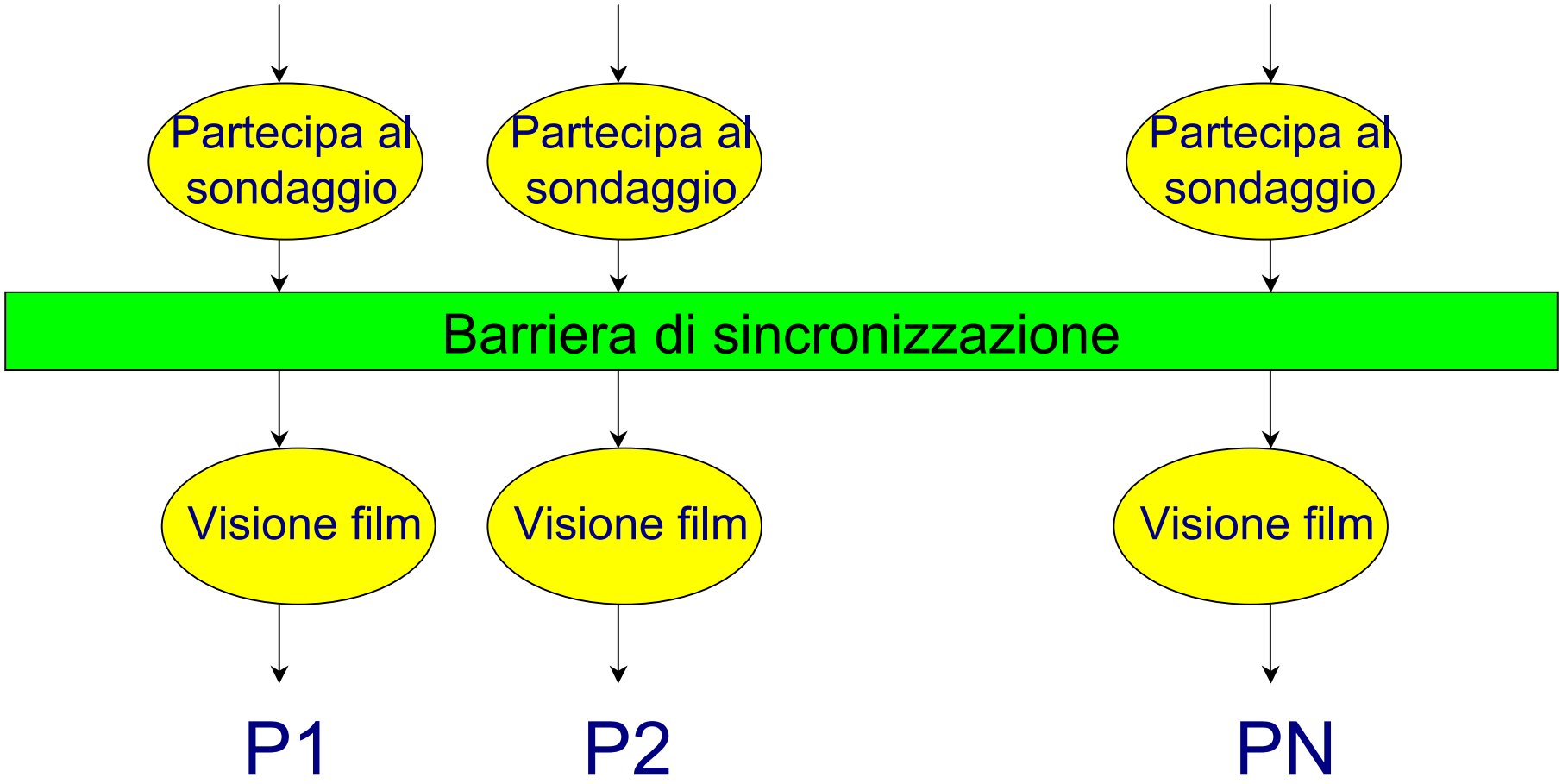
Ognuno degli N utenti ha un comportamento strutturato in **due fasi consecutive**:

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 arrivati=0;
```

Struttura del thread i-simo Pi:

```
<operazione 1 di Pi>

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

<operazione 2 di Pi>
```

Esercizio 3

Si aggiunga al problema dell'esercizio 2 i seguenti vincoli:
possono assistere alla visione del film vincitore:

- Sia le persone che hanno partecipato al sondaggio;
- Sia qualunque altro utente della rete televisiva.

In particolare, si supponga che:

- il film sia accessibile soltanto in modalità *streaming* tramite il portale web della rete
- che vi sia un limite massimo MAXC al numero degli utenti che possono connettersi per la visione del film.

Estendere la soluzione dell'esercizio 2 in modo tale da soddisfare i requisiti di cui sopra.

Spunti & suggerimenti