LABORATORIO DI PROGRAMMAZIONE 2
Corso di laurea in matematica

Sistemi Operativi : sincronizzazione di processi e thread

Marco Lapegna
Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

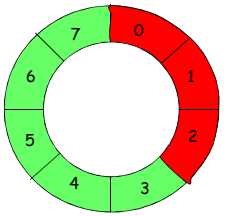
wpage.unina.it/lapegna

buffer limitato e memoria condivisa

- Dati condivisi

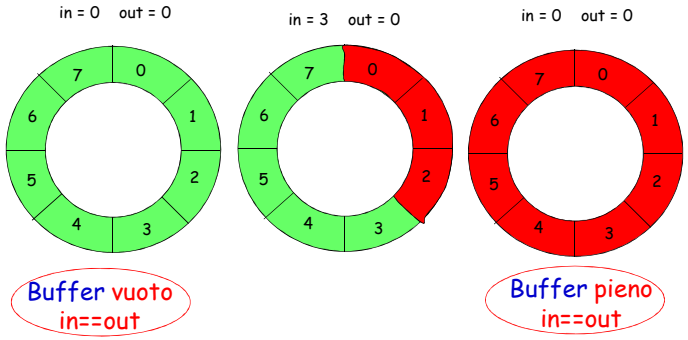
```
#define BUFFER_SIZE 8
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int  in = 0;
int  out = 0;
```

Es: BUFFER_SIZE = 8
in = 3 out = 0



- in indice della successiva posizione libera
- out indice della prima posizione occupata

Prima soluzione



osservazione

La condizione in == out equivale
sia a "buffer pieno" sia a "buffer vuoto"

Non e' possibile utilizzare tutti gli elementi del buffer
(se ne possono usare solo BUFFER_SIZE – 1)

- in == out ➔ buffer vuoto
- (in+1) %BUFFER_SIZE == out ➔ buffer pieno

Soluzione con buffer limitato e memoria condivisa

```
item nextProduced;
while (1) {
      while ( ((in + 1) % BUFFER_SIZE) == out ) ;
                /* buffer pieno: non fare niente */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
}
```

Processo Produttore

```
item nextConsumed;
while (1) {
      while (in == out) ;
               /* buffer vuoto: non fare niente  */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
}
```

Processo Consumatore

Problema

Trovare una soluzione che utilizza tutti gli elementi del buffer

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

La variabile counter (numero di elementi presenti nel buffer) permette di evitare l'ambiguita'

Buffer limitato

```
item nextProduced;
while (1) {
      while (counter == BUFFER_SIZE);
               /* buffer pieno */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter ++;
}
```

Processo produttore

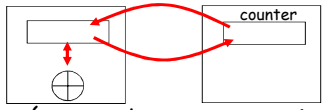
```
item nextConsumed;
while (1) {
      while (counter == 0);
               /* buffer vuoto */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter – –;
}
```

Processo consumatore

Problema

- L' istruzione di aggiornamento del contatore counter++ viene realizzata in linguaggio macchina come…

```
register1 = counter
register1 = register1 + 1
counter = register1
```

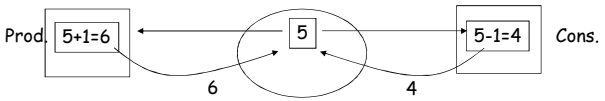


cpu (assegnata al produttore) — memoria condivisa

- Se, sia il produttore che il consumatore tentano di accedere al buffer contemporaneamente, le istruzioni in linguaggio assembler possono risultare interfogliate e il risultato dipende da come i processi accedono alla cpu

Problema con la memoria condivisa

Esempio: se counter inizialmente vale 5



Il risultato sara' 4 o 6 a secondo di chi accede per ultimo alla memoria condivisa (mentre il risultato corretto e' 5: un elemento prodotto e uno consumato)

L'istruzione di modifica di counter deve essere atomica
(non deve subire interruzioni)

Proteggere i dati nella memoria condivisa e controllare gli accessi

Race condition

- **Race condition**: Situazioni nelle quali più processi accedono in concorrenza, e modificano, dati condivisi. L'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi.
- Per evitare le corse critiche occorre che processi concorrenti vengano **sincronizzati**.
- Ciascun processo è costituito da un segmento di codice, chiamato **sezione critica**, in cui accede a dati condivisi in maniera esclusiva

Problema della sezione critica

- Problema — assicurarsi che, quando un processo esegue la sua sezione critica, a nessun altro processo sia concesso eseguire la propria.
- L'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva nel tempo.
- Soluzione — progettare un protocollo di cooperazione fra processi:
  - Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una entry section;
  - La sezione critica è seguita da una exit section; il rimanente codice è non critico.

Struttura generale dei programmi con sezione critica

```
do {
        sezione non critica
    entry_section( )
    sezione critica
    exit_section( )
        sezione non critica
} while (1);
```

L'implementazione delle funzioni che regolano l'accesso alla sezione critica puo' avvenire

- Attraverso strumenti software
- Attraverso strumenti hardware

Esempio: processo produttore

```
item nextProduced;
while (1) {
        while (counter == BUFFER_SIZE);
                /* buffer pieno */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        entry_section()
        counter ++;
        exit_section()
}
```

Sezione critica

Durante l'esecuzione del codice nella sezione critica, il processo consumatore non puo' accedere alla variabile counter

Soluzione software al problema della sezione critica

Requisiti da soddisfare

1. **Mutua esclusione**. Se un processo è in esecuzione nella sua sezione critica, nessun altro processo può eseguire la propria sezione critica.
2. **Progresso**. Un processo che non è in esecuzione nella propria sezione critica non puo' impedire ad un altro processo di accedere alla propria sezione critica.
3. **Attesa limitata**. Se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata.

- Si assume che ciascun processo sia eseguito ad una velocità diversa da zero.
- Non si fanno assunzioni sulla velocità relativa degli *n* processi.

Algoritmo 1

- Valido per 2 processi P_0 e P_1
- Variabile condivisa che determina il turno:
  - int turn;
    (inizialmente turn = 0).
  - turn = i ⇒ P_i può entrare nella propria sezione critica.

- Processo P_0 (analogamente per P_1)

```
do {
    sezione non critica
    while (turn == 1 );
    sezione critica
    turn = 1;
    sezione non critica
} while (1);
```

Aspetta che turn==0

P1 puo' entrare

Algoritmo 1 : caratteristiche

- Usa variabili globali per controllare quale processo puo' entrare nella sezione critica
- Verifica costantemente se la sezione critica e' disponibile
  - (attesa attiva)
  - Consuma cicli di cpu
- I processi eseguono la sezione critica alternandosi

- Soddisfa la **mutua esclusione**, ma non il **progresso**. Se *turn=0*, P_1 non può entrare nella propria sezione critica, anche se P_0 si trova fuori della propria sezione critica.

Algoritmo 2

- Variabili condivise:
  - boolean flag[2];
    (inizialmente flag [0] = flag [1] = false).
  - flag [i] = true ⇒ P_i vuole entrare nella propria sezione critica.

- Processo P_0

```
do {
      sezione non critica
      flag[ 0 ] = true;
      while ( flag[ 1 ] );
      sezione critica
      flag [ 0 ] = false;
      sezione non critica
} while (1);
```

P1 non puo' entrare

P1 puo' entrare

Algoritmo 2 : caratteristiche

- Introduce un flag prima della sezione critica
- Garantisce la mutua esclusione
- Introduce la possibilita' di stallo dei processi
  - Entrambi i processi possono porre *flag[i] = true*, (entrambi tentano di entrare) bloccandosi indefinitamente
- garantisce il progresso

Algoritmo 3 (di Peterson)

- Combina le variabili condivise degli algoritmi 1 e 2
  - int turn =0
  - boolean flag[0] = flag[1] = false

- Processo P_0 (analogamente per P_1)

```
do {
      sezione non critica
      flag[ 0 ] = true;
      turn = 1;
      while (flag[ 1 ] and turn == 1);
      sezione critica
      flag[ 0 ] = false;
      sezione non critica
} while (1);
```

P1 vuole entrare?

Tocca a P1?

Algoritmo di Peterson

- Usa il concetto di "processo favorito" per determinare chi deve accedere alla sezione critica (variabile turn)
  - Risolve i conflitti tra i processi

- Sono soddisfatte tutte le condizioni.
  - Mutua esclusione
  - Progresso
  - Attesa limitata

Risolve il problema della sezione critica per due processi.

Dimostr. mutua esclusione dell'alg. di Peterson

Algoritmo di P0

```
do {
   sezione non critica
   flag[ 0 ] = true;
   turn = 1;
   while (flag[ 1 ] and turn == 1);
   sezione critica
   flag[ 0 ] = false;
   sezione non critica
} while (1);
```

supponiamo (per assurdo) che P0 e P1 sono entrambi nella s.c.

⇩

flag[1]==false oppure turn==0 (in P0)
flag[0]==false oppure turn==1 (in P1)

Ma flag[0]=flag[1]=true

⇩

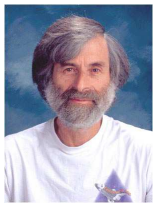
turn=0 e turn=1
(assurdo perche' e' una variabile condivisa)

Algoritmo del fornaio (Leslie Lamport)

Soluzione del problema della sezione critica per *n* processi.

- Prima di entrare nella loro sezione critica, i processi ricevono un numero. Il possessore del numero più basso entra in sezione critica.
- Se i processi P_i e P_j ricevono lo stesso numero, se $i < j$, allora P_i viene servito per primo, altrimenti tocca a P_j.
- Notazione : (# biglietto, # processo)
  - $(a,b) < (c,d)$, se $a < c$ oppure se $a = c$ e $b < d$;
- Lo schema di numerazione genera sempre numeri non decrescenti; ad esempio, 1,2,3,3,3,3,4,5...
- Variabili condivise:

```
boolean scelta[n] = false;
int number[n] = 0;
```



Leslie Lamport

Algoritmo del fornaio (proc P_i)

```
do {
        sezione non critica
   scelta[ i ] = true;
   number[  i ] = max(number[0], number[1], …, number [n – 1]) + 1;
   scelta[ i ] = false;
   for (j = 0; j < n; j++) {
        if ( i != j) {
            while (scelta[ j ]);
            while ((number[ j ] != 0)  &&  (number[  j ] , j ) < (number[ i ] , i )) ;
        }
    }
        sezione critica
   number[ i ] = 0;
        sezione non critica
}  while (1);
```

Prendo un numero

Controllo altri proc

Aspetto se Pj sta prendendo un numero

Aspetto il mio turno (ho il numero piu' grande)

number [i] = 0 indica che P_i e' uscito dalla sezione critica.

Soluzioni Hardware

In un ambiente multiprogrammato per garantire la mutua esclusione basterebbe disabilitare le interruzioni, in modo che i processi non vengano interrotti mentre eseguano la sezione critica

⇩

Sono annullati tutti i vantaggi del time sharing

Molte architetture possiedono particolari istruzioni che sono eseguite **atomicamente**

Tali istruzioni possono essere utilizzate per risolvere il problema della sezione critica in maniera facile, efficiente e sicuro

Hardware per la sincronizzazione: TestAndSet

- Istruzione TestAndSet

  boolean TestAndSet(boolean *target)

- definizione:
  - Ritorna vero se target e' vero
  - Ritorna falso se target e' falso
  - Pone sempre target vero
- L'istruzione e' eseguita atomicamente

Mutua esclusione con TestAndSet

- Dati condivisi:
  boolean lock = false;
- Soluzione per n processi
- Process P_i

```
do {
      sezione non critica
      while ( TestAndSet(&lock) ) ;
      sezione critica
      lock = false;
      sezione non critica
}
```


Hardware per la sincronizzazione: Swap

- Istruzione Swap

  void Swap(boolean &a, boolean &b)

- definizione:
  - Scambia il contenuto di a e b
- L'istruzione e' eseguita atomicamente

Mutua esclusione con Swap

- Dati condivisi
  boolean lock = false;
- soluzione per n processi
- Processo P_i

```
do {
      sezione non critica
      key = true;
      while (key == true) Swap(&lock, &key);
      sezione critica
      lock = false;
      sezione non critica
}
```

Semafori

- I meccanismi di sincronizzazione basati su dati condivisi hanno lo svantaggio dell'attesa attiva con conseguente consumo di cicli di cpu
- I semafori sono strumenti di sincronizzazione che superano il problema dell'attesa attiva
- Il semaforo contiene una variabile intera S che serve per proteggere l'accesso alle sezioni critiche

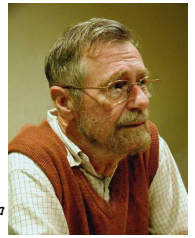
- Si può accedere al semaforo (alla variabile) solo attraverso due operazioni atomiche
  - ***wait(S):*** il processo chiede di accedere alla propria sezione critica. Esso aspetta se altri processi sono dentro la loro sezione critica
  - ***signal(S):*** il processo vuole uscire dalla propria sezione critica

definizione di wait e signal

wait (S)
- se S>0 allora S=S-1 e il processo continua l'esecuzione
- altrimenti (S=0) il processo si blocca

signal(S)
- esegue S=S+1

altri nomi di wait e signal
- wait ( ) == down ( ) == P ( )
- signal ( )== up ( ) == V ( )



Edsger W. Dijkstra

Sezione critica con *n* processi

- Variabili condivise tra gli n processi:
  semaphore mutex; // inizialmente **mutex = 1**
- Processo P_i:

```
do {
        sezione non critica
        wait( & mutex);
                sezione critica
        signal(& mutex);
        sezione non critica
} while (1);
```


Implementazione dei semafori (1)

Implementazione "ingenua"

S > 0 ➔ possibile accedere alla regione critica
S = 0 ➔ non e' possibile accedere alla regione critica

```
void wait(int * S) {
        while (S <= 0);
        S=S-1;
}
```

```
void signal (int * S) {
        S=S+1;
}
```

- La mutua esclusione alla variabile S, facilmente realizzabile con TestAndSet o Swap
- Non risolve il problema dell' "attesa attiva"

Implementazione dei semafori (2)

- Si definisce un semaforo come una struttura:

```
typedef  struct {
    int value;
    struct process *List;
} semaphore;
```

Lista di processi (PCB) in attesa al semaforo

numero di proc in attesa

- Si assume che siano disponibili due operazioni (syscall):
  - **block** sospende il processo che la invoca;
  - **wakeup(*P*)** riprende l'esecuzione di un processo sospeso ***P***.

Implementazione dei semafori

- Le operazioni sui semafori possono essere definite come…

```
void wait (semaphore * S){
        S.value– –;
        if (S.value < 0) {
                aggiungi il processo corrente a  S.List;
                block;
        }
}
```

```
void signal (semaphore * S){
        S.value++;
        if (S.value <= 0) {
                rimuovi un processo P da S.List;
                wakeup(P );
        }
}
```


problema

Come garantire che le operazioni
wait e signal vengano eseguite atomicamente?

(mettiamo un semaforo su un semaforo?)

Essendo funzioni di piccole dimensioni in questi casi si disabilitano le interruzioni durante l'esecuzione delle funzioni

Esempio di sincronizzazione con semafori

- I semafori possono anche essere usati per sincronizzare le operazioni di due o piu' processi
- Ad esempio: si vuole eseguire *B* in P_j solo dopo che *A* è stato eseguito in P_i
- Si impiega un semaforo *flag* inizializzato 0
- Codice:

P_i	P_j
⋮	⋮
A	***wait*(&*flag*)**
***signal*(&*flag*)**	***B***

Un pericolo con i semafori: Deadlock

- due o più processi sono in attesa indefinita di un evento che può essere generato solo da uno dei due processi in attesa.

- Siano *S* e *Q* due semafori inizializzati a 1:

P_0	P_1
wait(&*S*);	*wait*(&*Q*);
wait(&*Q*);	*wait*(&*S*);
⋮	⋮
signal(&*S*);	*signal*(&*Q*);
signal(&*Q*)	*signal*(&*S*);

Attesa indefinita !!

Se dopo *wait(&S)* di P_0 viene eseguita *wait(&Q)* di P_1 si ha un deadlock (P_0 non puo' accedere a *Q* e P_1 non puo' accedere a *S*)

Due tipi di semafori

- Semaforo ***contatore*** — intero che può assumere valori in un dominio non limitato.
- Semaforo ***binario*** — intero che può essere posto solo a 0 o 1; può essere implementato più semplicemente.
- Un semaforo contatore può essere realizzato mediante due semafori binari.

Problema

Esercizio del produttore/consumatore
a buffer limitato con i semafori

- Variabili condivise:
  semaphore full, empty, mutex;
  // inizialmente **full = 0, empty = n, mutex = 1**
- Il buffer ha *n* posizioni, ciascuna in grado di contenere un elemento.
- Il semaforo binario *mutex* garantisce la mutua esclusione sugli accessi al buffer.
- I semafori contatore *empty* e *full* contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer.

Produttore/consumatore a buffer limitato

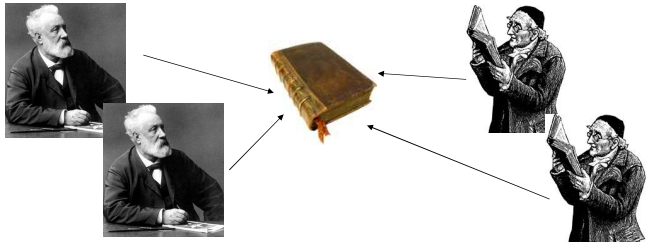
Processo produttore

```
do {
        ...
    produce un elemento in nextp
        ...
    wait(empty);
    wait(mutex);
        ...
    inserisce nextp nel buffer
        ...
    signal(mutex);
    signal(full);
} while (1);
```

Processo consumatore

```
do {
    wait(full)
    wait(mutex);
        ...
    sposta un elemento dal buffer in nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    consuma un elemento in nextc
        ...
} while (1);
```

Problema dei lettori e degli scrittori



- Un insieme di dati (ad es. un file) deve essere condiviso da più processi concorrenti che possono richiedere la sola lettura del contenuto, o anche un aggiornamento.
- Se due *lettori* accedono contemporaneamente ai dati condivisi non ha luogo alcun effetto negativo
- Se uno *scrittore* accede simultaneamente ad un altro processo si può avere incoerenza dell'informazione

2 tipi di problemi

- 1° problema dei lettori-scrittori: nessun processo lettore deve attendere, salvo che uno scrittore abbia già ottenuto l'accesso ai dati condivisi (precedenza ai lettori)

possibilità di starvation per gli scrittori

- 2° problema dei lettori-scrittori: nessun processo scrittore deve attendere, salvo che uno scrittore abbia già ottenuto l'accesso ai dati condivisi (precedenza agli scrittori)

possibilità di starvation per gli lettori.

soluzione al primo problema scrittori/lettori

- Variabili condivise:
  semaphore mutex, wrt;
  int readcounter;
  // inizialmente **mutex = 1, wrt = 1, readcounter = 0**

Processo scrittore

```
wait(wrt);
   ...
effettua la scrittura
   ...
signal(wrt);
```

Uso esclusivo ai dati condivisi

Altri possono accedere ai dati condivisi

soluzione al primo problema scrittori/lettori

Uso esclusivo a readcounter

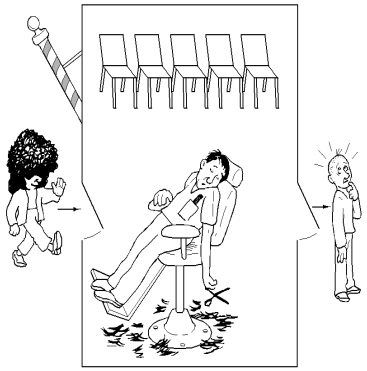
Se sono il primo lettore fermo gli scrittori

Se sono l'ultimo lettore sblocco gli scrittori

Processo lettore

```
wait(mutex);
readcounter++;
if (readcounter == 1)  wait(wrt);
signal(mutex);
      ...
   si effettua la lettura
      ...
wait(mutex);
readcounter – –;
if (readcounter == 0)  signal(wrt);
signal(mutex);
```

il barbiere sonnolento



in un negozio di barbiere ci sono
- una sedia per lavorare
- n sedie per far attendere i clienti

- il barbiere di solito dorme
- quando arriva un cliente, questi lo sveglia e si fa servire
- se nel frattempo arrivano altri clienti, si accomodano sulle sedie oppure vanno via se sono tutte occupate

1a soluzione: n=infinito

```
semaphore  customer = 0        # clienti in attesa
semaphore  barber = 0          # barbieri
```

```
void barbiere( )

while (1){
        wait(&customer)
        signal(&barber)
        cut_hair( )
}
```

```
void cliente ( )

signal(&customer)
wait(&barber)
cut_hair( )
```


2a soluzione : n=5

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
      while (TRUE) {
            wait  (&customers);     /* go to sleep if # of customers is 0 */
            wait  (&mutex);         /* acquire access to 'waiting' */
            waiting = waiting – 1;  /* decrement count of waiting customers */
        signal (&barbers);          /* one barber is now ready to cut hair */
        signal (&mutex);            /* release 'waiting' */
            cut_hair( );            /* cut hair (outside critical region) */
      }
}


void customer(void)
{
       wait  (&mutex);              /* enter critical region */
      if (waiting < CHAIRS) {       /* if there are no free chairs, leave */
            waiting = waiting + 1;  /* increment count of waiting customers */
        signal (&customers);        /* wake up barber if necessary */
        signal (&mutex);            /* release access to 'waiting' */
            wait  (&barbers);       /* go to sleep if # of free barbers is 0 */
           get_haircut( );          /* be seated and be serviced */
      } else {
        signal (&mutex);            /* shop is full; do not wait */
      }
}
```


Thread Posix e semafori

I principali meccanismi di sincronizzazione previsti dallo standard POSIX 1003.1c sono:
- **I mutex (semafori binari, per le sezioni critiche)**
- **i semafori (semafori contatore)**
- **le variabili condizione (generalizzazione)**

Mediante tali strumenti è possibile realizzare meccanismi di accesso alle risorse

I mutex

Un mutex è un semaforo binario (0 oppure 1)

Un mutex è mantenuto in una struttura
pthread_mutex_t

Tale struttura va allocata e inizializzata

Per inizializzare:
– se la struttura è allocata staticamente:
pthread_mutex_t mutex= PTHREAD_MUTEX_INITIALIZER

– se la struttura è allocata dinamicamente (e.g. se si usa malloc): chiamare pthread_mutex_init

I mutex

```
#include <pthread.h>

int pthread_mutex_lock( pthread_mutex_t   mutex)
```

mutex: semaforo binario

Implementazione della funzione wait

I mutex

```
#include <pthread.h>

int pthread_mutex_unlock( pthread_mutex_t   mutex)
```

mutex: semaforo binario

Implementazione della funzione signal

Esempio : somma con 4 thread

```
.....
pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER;
void *thread(void *) ;
int somma = 0;

int main( ){                          // inizio main
      pthread_t   tid1, tid2, tid3, tid4;
      pthread_create(&tid1, NULL, thread, NULL);
      pthread_create(&tid2, NULL, thread, NULL);
      pthread_create(&tid3, NULL, thread, NULL);
      pthread_create(&tid4, NULL, thread, NULL);
      pthread_join(tid1, NULL);
      pthread_join(tid2, NULL);
      pthread_join(tid3, NULL);
      pthread_join(tid4, NULL);
} //fine main

void *thread (void *arg){             // inizio thread function
int   i;   float sum;   double btime, etime, a=10;

     btime = get_cur_time();
      sum=0;
      for (i=0;  i<10000000  ; i++){
            sum=sum+a;
           a = -a;
      }
     pthread_mutex_lock(&m1);
      somma=somma+sum;
      pthread_mutex_unlock(&m1);

     etime = get_cur_time();
     printf("ELAPSED = %f\n", etime-btime);
}                                     // fine thread function
```

m1 e' un semaforo binario (mutex)

somma e' una variabile condivisa

Sezione critica protetta da semafori

N.B. compilare il tutto linkando la libreria libpthread.a

semafori

Semafori il cui valore può essere impostato dal programmatore

utilizzati per casi più generali di sincronizzazione
esempio: produttore consumatore

Interfaccia

- operazione wait
- operazione post (signal)

associati al tipo
sem_t

includere l'header
#include <semaphore.h>

semaforo. : inizializzazione

L'inizializzazione di una v.c. si puo` realizzare con:

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

dove
- **sem**: puntatore al semaforo
- **pshared**: mettere 0.
- **value**: valore iniziale del semaforo

ritorna:
- 0 se ok; -1 se errore

wait e signal

```
int sem_wait (sem_t *sem)

int sem_post (sem_t *sem)
```

• sem: nome del semaforo

disponibile anche la funzione

```
sem_getvalue(sem_t *sem, int *sval)
```

per leggere il valore del semaforo

Variabili di condizione

Le variabili condizione (condition) sono uno strumento di sincronizzazione che premette ai threads di sospendere la propria esecuzione in attesa che siano soddisfatte alcune condizioni su dati condivisi.

ad ogni condition viene associata una coda nella quale i threads possono sospendersi (tipicamente, se la condizione non e` verificata).

Definizione di variabili condizione:
pthread_cond_t: è il tipo predefinito per le variabili condizione

Operazioni fondamentali:
- **inizializzazione: pthread_cond_init**
- **sospensione: pthread_cond_wait**
- **risveglio: pthread_cond_signal**

Var.Cond. : inizializzazione

L'inizializzazione di una v.c. si puo` realizzare con:

```
int pthread_cond_init(pthread_cond_t* cond,
                      pthread_cond_attr_t* cond_attr)
```

dove
- **cond** : individua la condizione da inizializzare
- **attr** : punta a una struttura che contiene gli attributi della condizione; se NULL, viene inizializzata a default.

in alternativa, una variabile condizione può essere inizializzata staticamente con la costante:
PTHREAD_COND_INIZIALIZER

esempio: pthread_cond_t C= PTHREAD_COND_INIZIALIZER ;

Var.Cond. : Wait

La **sospensione** su una condizione si ottiene mediante:

```
int pthread_cond_wait(pthread_cond_t* cond,
    pthread_mutex_t* mux);
```

dove:
cond: e` la variabile condizione
mux: e` il mutex associato ad essa (la verifica della condizione e' una s.c.)

Effetto:
- il thread chiamante si sospende sulla coda associata a cond, e il mutex mux viene liberato

Al successivo risveglio (provocato da una signal), il thread rioccupera` il mutex automaticamente.

Var.Cond. : Signal

Il **risveglio** di un thread sospeso su una variabile condizione puo`essere ottenuto mediante la funzione:

```
int pthread_cond_signal(pthread_cond_t* cond);
```

dove **cond** e` la variabile condizione.

Effetto**:**
- se esistono thread sospesi nella coda associata a cond, ne viene

risvegliato uno (non viene specificato quale).
- se non vi sono thread sospesi sulla condizione, la signal non ha effetto.

- Per risvegliare tutti i thread sospesi su una variabile condizione(v.

signal_all):
int pthread_cond_broadcast(pthread_cond_t* cond);