Sincronizzazione con PTHREADS

E. Mumolo

Strumenti per la sincronizzazione in POSIX

- Gli strumenti principali per effettuare sincronizzazione tra threads sono:
  - Mutex
  - Pthread_join
  - Variabili condizione
  - Semafori contatori

Mutex

- Un mutex viene usato per sincronizzare l'uso di una risorsa
- Un mutex deve essere di tipo `pthread_mutex_t`
- Le caratteristiche del mutex sono descritte nell'oggetto attributi del mu
  - Un mutex può essere inizializzato staticamente:

```
pthread_mutex_t mutex = THREAD_RECURSIVE_MUTEX_INITIALIZER;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Un mutex può essere inizializzato dinamicamente:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *at
```

- L'oggetto attributi è di tipo `pthread_mutexattr_t`
- Attributi di default vengono assegnati con

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

- La struttura attributi viene distrutta con

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Caratteristiche del mutex

- Possono essere assegnate o estratte

```
#include <pthread.h>
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int tipo);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *tipo);
```

- Tipo può essere:
  - PTHREAD_MUTEX_NORMAL
  - PTHREAD_MUTEX_RECURSIZE_NP
    - Può bloccare un mutex già bloccato.
    - Un contatore (lock count) matiene memoria dei locks (inc) e unlocks (dec)
    - Quando lock count arriva a zero, il mutex può essere acquisito
  - PTHREAD_MUTEX_ERRORCHECK_NP
    - Riporta errori: lock un mutex locked, unlock un mutex unlocked

Caratteristiche del mutex

- pthread_mutex_init inizializza un mutex con gli attributi specificati nel parametro attr. Se attr è NULL, vengono assegnati gli attributi di default

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr *attr);
```

- pthread_mutex_destroy elimina un mutex

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- pthread_mutexattr_destroy cancella la struttura attr del mutex

```
#include <pthread.h>
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Blocco/sblocco mutex

- Blocca del mutex specificato. Se già bloccato aspetta finchè il mutex non si sblocca.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t* mutex );
```

- Sblocco del mutex specificato.

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t* mutex );
```

  - Se ricorsivo, solo l'ultimo sblocco (lock count=0) rilascia il mutex.

pthread_mutex_trylock

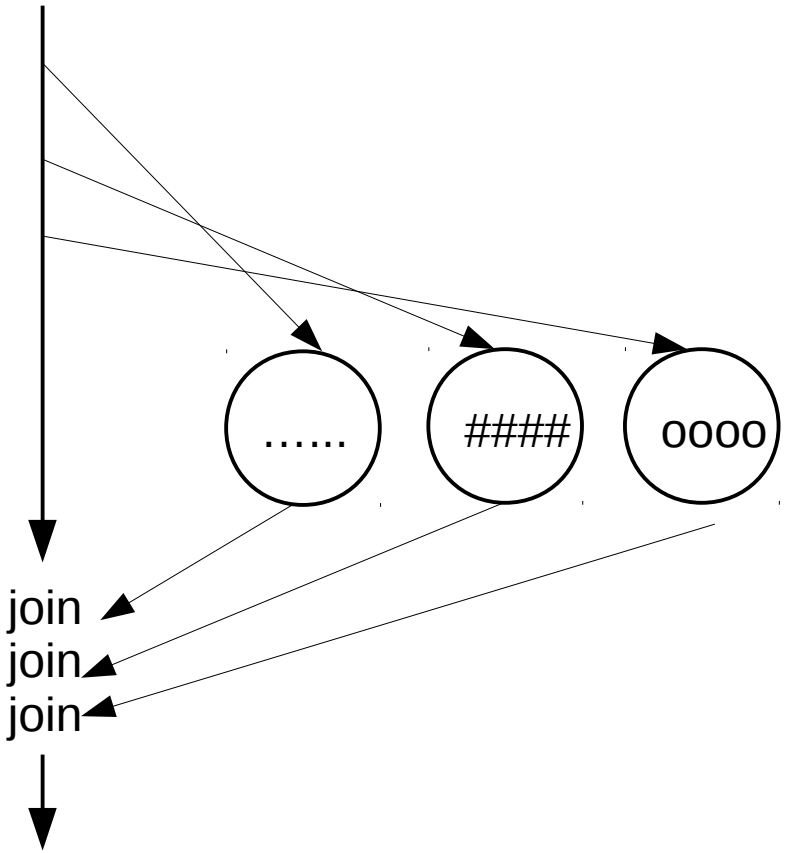
```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t* mutex );
```

- Funziona come il lock() ma per I mutex ricordivi
- Cerca di bloccare un mutex. Se il mutex è già bloccato, il thread chiamante ritorna un errore EBUSY senza aspettare che il mutex si liberi.
- Esempio (pseudo-codice):

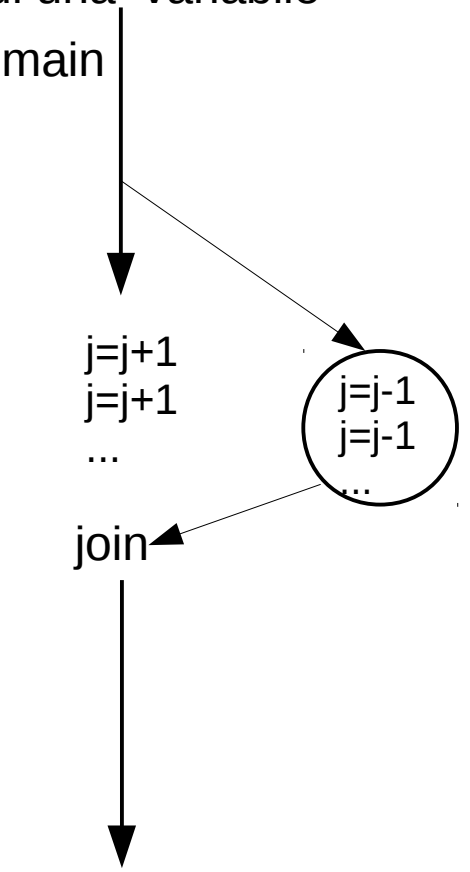
```
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void thr(void *arg){
      while(true){
         r=pthread_mutex_trylock(&m);
         if(r!=EBUSY){
             pthread_mutex_unlock(&m);
             break;
         }else{incrementa un contatore;}
      }
}
```

```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mymutex;

void *body(void *arg)
{
     int i,j;
     pthread_mutex_lock(&mymutex);
     for (j=0; j<40; j++) {
          for (i=0; i<1000000; i++);
          printf("%c",*(char *)arg);
     }
     pthread_mutex_unlock(&mymutex);
     return NULL;
}

int main()
{
     pthread_t t1,t2,t3;  int err;
     pthread_mutexattr_t mymutexattr;
     pthread_attr_t myattr;

     pthread_mutexattr_init(&mymutexattr);
     pthread_mutex_init(&mymutex, &mymutexattr);
     pthread_mutexattr_destroy(&mymutexattr);

     pthread_attr_init(&myattr);
     err = pthread_create(&t1, &myattr, body, (void *)".");
     err = pthread_create(&t2, &myattr, body, (void *)"#");
     err = pthread_create(&t3, &myattr, body, (void *)"o");

     pthread_attr_destroy(&myattr);
     pthread_join(t1, NULL); pthread_join(t2, NULL); pthread_join(t3, NULL);

     printf("\n");
     return 0;
}
```

Esempio 1



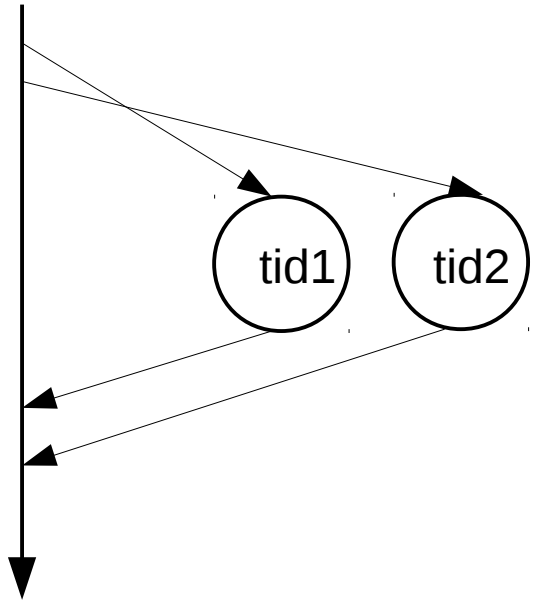
Esempio 2
Incremento e decremento concorrente di una variabile




```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal=0; //Variabile globale
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

void *thread_incrementa(void *arg)
{
     int i,j;  void *val;
     //for (i=0; i<1000000; i++);
     //pthread_mutex_lock(&mymutex);
     for ( i=0; i<20; i++ ){
           j=myglobal; //variabile locale temporanea
           j=j-1;   printf("thread_decr\n");
           myglobal=j;
     }
     //pthread_mutex_unlock(&mymutex);
     pthread_exit(val);
}

int main(void){
     pthread_t mythread;    int i;   void *status;
     if(pthread_create(&mythread,NULL,thread_incrementa,NULL) )
     {  printf("errore thread.");    exit(1);  }

     //pthread_mutex_lock(&mymutex);
     for ( i=0; i<20; i++)
     {
          myglobal=myglobal+1;
          printf("main incr\n");
     }
     //pthread_mutex_unlock(&mymutex);
     if(pthread_join(mythread,(void*)&status)){printf("errore join."); exit(1); }
     printf("\n myglobal uguale %d\n",myglobal);
     exit(0);
}
```

Esempio 3:
aggiornamento di un array

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#define check(return_val, msg) { if (return_val != 0) printf("%s",msg); }

unsigned long int condivisa[] = {0,0,0,0,0,0,0,0,0,0}; int ncondivisa = 0;/* Variabili condivise */
pthread_mutex_t Mutex = PTHREAD_MUTEX_INITIALIZER; /* lock per accesso memoria */
void *aggiorna ( int dim);  /* aggiornamento casuale */
int main(){
     pthread_t tid1,tid2;
     int retcode, k, dim=sizeof(condivisa)/sizeof(int);

     retcode=pthread_create(&tid1,NULL,(void *)aggiorna,(void *) dim);check(retcode,"errore create");
     retcode=pthread_create(&tid2,NULL,(void *)aggiorna,(void *) dim);check(retcode,"errore create");

     retcode = pthread_join(tid1,NULL);   check(retcode, "join failed");
     retcode = pthread_join(tid2,NULL);   check(retcode, "join failed");
     for(k=0; k<dim; k++)   printf("condivisa[%d]=%lu\n", k,condivisa[k]);
     exit(0);
}

void * aggiorna(int dim){
     int i;   int sl; srand(getpid());
     printf("processo=%d,thread=%lu\n",getpid(),pthread_self());
     //pthread_mutex_lock( &Mutex );
     for(i=0; i<dim/2 ; i++) {
          sl = 1 + (int) (5.0 * rand()/(RAND_MAX+1.0));
          usleep(sl*1000);
          //pthread_mutex_lock( &Mutex );
          condivisa[ncondivisa]=pthread_self();
          ncondivisa++;
          //pthread_mutex_unlock( &Mutex );
     }
     //pthread_mutex_unlock( &Mutex );
     return( (void *) NULL);
}
```



Variabili condizione

- Nei POSIX thread, variabile condizione + mutex = monitor.
- Se un thread attende sulla variabile condizione, sblocca il mutex, consentendo ad altri thread l'ingresso alla sezione critica.
- Una variabile condizione è di tipo pthread_cond_t
- La var. condizione uvc può essere inizializzata
  - Staticamente:

```
#include <pthread.h>
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
```

  - Dinamicamente:

```
#include <pthread.h>
pthread_cond_t cv;
pthread_condattr_init (attr);
pthread_cond_init (cv,attr);
```

- Cancellazione variabile: `pthread_cond_destroy (condition)`
- Cancellazione attributi: `pthread_condattr_destroy (attr)`
- Operazioni fondamentali:
  - sospensione: `pthread_cond_wait`
  - risveglio: `pthread_cond_signal`

Ricorda la definizione dei monitor...

```
monitor prodcons
condition pieno, vuoto;

entry Prod() {
	while(true){
		el=produci();
		while(n==N) pieno.wait; //se il buffer e' pieno, aspetta
		A[n]=el;
		n++;
		if(n==1) vuoto.signal; //1 elemento: sveglia il consumatore
	}
}

entry Cons() {
	while(true){
		while(n==0) vuoto.wait; //se il buffer e' vuoto, aspetta
		el=A[n];
		n--;
		if(n==N-1) pieno.signal;//1 posto libero: sveglia il produttore
	}
}
end monitor;
```

Attesa su una variabile condizione

```
#include <pthread.h>
pthread_cond_wait(&cv,&a_mutex);
```

- Il mutex viene sbloccato e il thread chiamante si blocca sulla variabile condizione.
- Quando si torna dalla funzione, il mutex sarà nuovamente bloccato, e sarà di proprietà del thread chiamante.
- Non utilizzare mutex ricorsivi con questa funzione

Risveglio di una variabile condizione

- Una variabile condizione può essere svegliata in due modi:
  - Mediante una chiamata alla funzione

    ```
    pthread_cond_signal (pthread_cond_t *cond)
    ```

    che sblocca il thread a priorità più alta che è in attesa da più tempo.
  - Oppure mediante una chiamata alla funzione

    ```
    pthread_cond_broadcast (pthread_cond_t *cond)
    ```

    che sblocca tutti i thread in ordine di priorità, usando un ordinamento FIFO per i thread con la stessa priorità.

Esempio

- Supponiamo di avere una sola variabile condivisa
- Supponiamo di avere due funzioni, put() e get(), con i seguenti pseudocodici:

```
void put(int value) {
   count = 1;
   buffer = value;
 }
```

```
int get() {
   count = 0;
   return buffer;
}
```

- Supponiamo di avere 1 consumatore e 1 produttore

```
cond_t cond;
mutex_t mutex;

void *produttore(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 1) pthread_cond_wait(&cond, &mutex);
        put(i);
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
 }

 void *consumatore(void *arg) {
    int i, tmp;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        if (count == 0) pthread_cond_wait(&cond, &mutex);
        tmp = get();
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
 }
```

Una soluzione funzionante con un produttore e un consumatore.

Attenzione: questa soluzione non funziona con piu' produttori o più consumatori!
L'IF deve diventare un while!

```
cond_t cond;
mutex_t mutex;

void *produttore(void *arg) {
   int i;
   for (i = 0; i < loops; i++) {
      pthread_mutex_lock(&mutex);
      while (count == 1) pthread_cond_wait(&cond, &mutex);
      put(i);
      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
   }
}

void *consumatore(void *arg) {
   int i, tmp;
   for (i = 0; i < loops; i++) {
      pthread_mutex_lock(&mutex);
      while (count == 0) pthread_cond_wait(&cond, &mutex);
      tmp = get();
      pthread_cond_signal(&cond);
      pthread_mutex_unlock(&mutex);
      printf("%d\n", tmp);
   }
}
```

Una soluzione funzionante con più produttori e consumatori.

Questa soluzione ha ancora un problema: usa solo UNA variabile condizione!

Soluzione con due variabili condizione e while()

```
cond_t empty, fill;
mutex_t mutex;

void *produttore(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 1) pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
 }

 void *consumatore(void *arg) {
    int i, tmp;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == 0) pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
 }
```

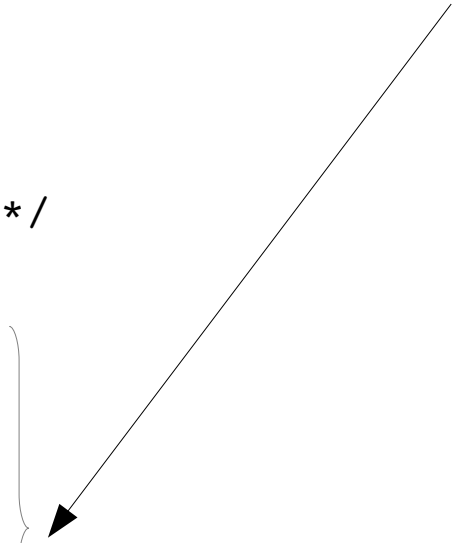
```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
   buffer[fill] = value;
   fill = (fill + 1) % MAX;
   count++;
 }

 int get() {
   int tmp = buffer[use];
   use = (use + 1) % MAX;
   count--;
   return tmp;
 }
```

Funzioni put() e get() riviste: buffer circolare, non una singola variabile!

Una soluzione finale del produttore-consumatore con buffer circolare

```
cond_t empty, fill;
mutex_t mutex;

void *produttore(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        pthread_mutex_lock(&mutex);
        while (count == MAX) pthread_cond_wait(&empty, &mutex);
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
}

void *consumatore(void *arg) {
    int i, tmp;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0) pthread_cond_wait(&fill, &mutex);
        tmp = get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
 }
```

Una variante: struttura condivisa



```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#define OVER (-1) /*Elemento di buffer vuoto*/
#define max 20 /*Numero di elementi da scrivere nel buffer*/
#define BUFFER_SIZE 20 /*Dimensione del buffer */

typedef struct /*Struttura condivisa */
{
    int buffer[BUFFER_SIZE];  /*shared memory*/
    pthread_mutex_t lock;     /*dichiarazione del mutex*/
    int readpos, writepos;
    int cont;
    pthread_cond_t notempty;  /*variabili di condizione*/
    pthread_cond_t notfull;
}prodcons;

prodcons buffer;

/* Inizializza il buffer */
void init (prodcons *b){
    pthread_mutex_init (&b->lock, NULL); /*inizializza il mutex*/

    pthread_cond_init (&b->notempty, NULL);/*inizializza le cond. var.*/
    pthread_cond_init (&b->notfull, NULL);

    b->cont=0;
    b->readpos = 0;
    b->writepos = 0;
}
```

```
void inserisci (prodcons *b, int data)  /* Inserimento: */
{
    pthread_mutex_lock (&b->lock); //mutex bloccato
    /* controlla che il buffer non sia pieno:*/
    while ( b->cont==BUFFER_SIZE) pthread_cond_wait (&b->notfull, &b->lock);

    b->buffer[b->writepos] = data;/* scrivi data e aggiorna lo stato del buffer */
    b->cont++;
    b->writepos++;
    if (b->writepos >= BUFFER_SIZE)  b->writepos = 0;

    pthread_cond_signal (&b->notempty); /* risveglia eventuali consumatori sospesi */
    pthread_mutex_unlock (&b->lock);//mutex sbloccato
}

int estrai (prodcons *b)  /*Estrazione: */
{
    int data;
    pthread_mutex_lock (&b->lock); //mutex bloccato
    while (b->cont==0) pthread_cond_wait(&b->notempty, &b->lock);/* buffer vuoto? */

    data = b->buffer[b->readpos];/* Leggi l'elemento e aggiorna lo stato del buffer*/
    b->cont--;
    b->readpos++;
    if (b->readpos >= BUFFER_SIZE) b->readpos = 0;

    pthread_cond_signal (&b->notfull);  /* Risveglia eventuali produttori sospesi*/
    pthread_mutex_unlock (&b->lock);    //mutex sbloccato
    return data;                        /*Ritorna il dato letto dal buffer*/
}
```

```
void *produttore(void *data) {  /*Thread*/
    int n;
    printf("sono il thread produttore\n\n");
    for (n = 0; n < max; n++) /*Scrittura delle elementi nel buffer*/
    {
        printf ("Thread produttore scrivo %d --->\n", n);
        inserisci (&buffer, n); /*Chiamata alla funzione inserisci*/
    }
    inserisci (&buffer, OVER); //Buffer pieno
    return NULL;
}
void *consumatore(void *data) { /*Thread*/
    int d;
    printf("sono il thread consumatore \n\n");
    while (1) /*Ciclo infinito*/
    {
        d = estrai (&buffer); /*Legge gli elementi da buffer*/
        if (d == OVER) /*Esce dal ciclo quando il buffer è vuoto*/
        break;
        printf("Thread consumatore leggo: --> %d\n", d);
    }
    return NULL;
}

main (){
    pthread_t th_a, th_b;
    init (&buffer); /*Inizializzazione struttura*/
    pthread_create (&th_a, NULL, producer, NULL);/* Creazione threads: */
    pthread_create (&th_b, NULL, consumer, NULL);
    pthread_join (th_a, NULL);/* Attesa terminazione threads creati: */
    pthread_join (th_b, NULL);
    return 0;
}
```

Semafori POSIX

- Apertura
- Inizializzazione
- Operazione Down
- Operazione Up

Creazione di un semaforo 'con nome'

```
sem_t *sem_open (char *name, int oflags,mode_t creation_mode,
                 unsigned init_val);
```

- Per creare un semaforo usare il flag O_CREAT, possibilmente in || con O_EXCL per ottenere la generazione di errori se il semaforo esiste già..
- creation_mode specifica i permessi d’accesso ad un semaforo creato.
- Generalmente si indica RWX per il gruppo di utenti desiderato (U, G o O).
  - Il parametro init_val viene usato per inizializzare il valore del semaforo.
  - Il valore ritornato è un puntatore al semaforo, o -1.

Creazione di un semaforo 'senza nome'

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

- Questa funzione crea un semaforo ‘senza nome’.
- Se il parametro pshared non è zero, allora il semaforo può essere condiviso tra i processi
- Il parametro value è usato per inizializzare il valore del semaforo.
- Questa funzione ritorna 0 o -1 se ha successo o no.

Chiusura

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

- Questa funzione è usata per chiudere la connessione con un semaforo con nome, aperto con sem_open.
- I semafori con nome sono persistenti; cioè lo stato di un semaforo persiste anche se nessuno ha aperto il semaforo.
- Utilizzando il semaforo dopo che sia chiuso ha un effetto indefinito.

Lettura del semaforo

```
int sem_getvalue (sem_t *sem, int *value);
```

- Questa funzione viene usata per ottenere il valore di un semaforo con o senza nome.
- Il valore tornato è positivo se la risorsa controllata dal semaforo è sbloccata.
- Se il valore tornato è 0, allora la risorsa è bloccata.
- Alcune implementazioni ritornano un numero negativo per indicare che la risorsa è bloccata.

Operazione down

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

- Pseudocodice:

```
wait(S):
if(S<=0)
      aggiungi il descrittore del processo in coda su S;
else
      S--;
```

- Questa funzione cerca di decrementare il valore del semaforo identificato.
- Se ha avuto successo, il processo chiamante continua.
- Se il valore del semaforo è 0, il thread chiamante è bloccato
- La chiamata sem_wait della libreria Posix usa la chiamate di sistema ~semop(-1) del kernel

Operazione up

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

- Pseudocodice:

```
signal(S):
if(c'e' un descrittore in attesa su S)
      esegui il processo;
else
S++;
```

- Questa funzione incrementa il valore del semaforo identificato.
- Se un processo è bloccato a causa di una chiamata sem_wait, il processo con priorità più alta che sta aspettando viene svegliato.
- La chiamata sem_post della libreria Posix usa la chiamate di sistema ~semop(+1) del kernel

Cancellazione di un semaforo

```
int sem_unlink (char *name);
```

- Questa funzione distrugge il semaforo con nome. Se altri processi hanno aperto il semaforo, possono continuare ad utilizzarlo finche' non lo chiudono con `sem_close`.

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

- Questa funzione viene usata per cancellare un semaforo senza nome.
- Il semaforo che viene distrutto deve essere stato precedentemente inizializzato con `sem_init`.
- La distruzione di un semaforo sul quale altri processi sono bloccati causa il loro sblocco con un errore (errno = EINVAL).
- Usando un semaforo dopo che sia stato distrutto ha un effetto non definito.

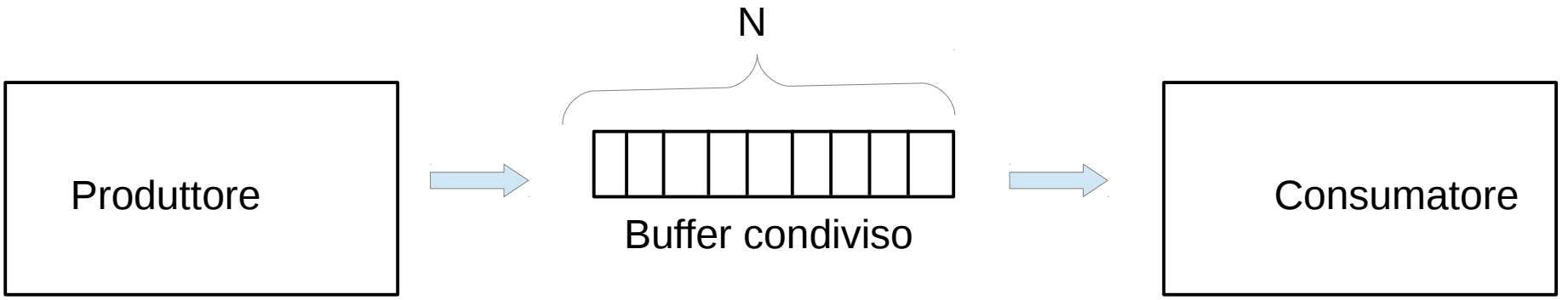
```c
/* primo esempio di utilizzo dei semafori */
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t mysem;

void *body(void *arg)
{
    int i,j;
    for (j=0; j<40; j++) {
        sem_wait(&mysem);
        for (i=0; i<1000000; i++); fprintf(stderr,(char *)arg);
        sem_post(&mysem);
    }
    return NULL;
}

int main()
{
    pthread_t t1,t2,t3;  pthread_attr_t myattr; int err;

    sem_init(&mysem,0,1);
    pthread_attr_init(&myattr);
    err = pthread_create(&t1, &myattr, body, (void *)".");
    err = pthread_create(&t2, &myattr, body, (void *)"#");
    err = pthread_create(&t3, &myattr, body, (void *)"o");
    pthread_attr_destroy(&myattr);

    pthread_join(t1, NULL); pthread_join(t2, NULL); pthread_join(t3, NULL);

    printf("\n");
    return 0;
}
```

```c
#include <stdio.h>     /* secondo esempio */
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
int x = 0; /* variabili globali */
sem_t m;
void *thread(void *arg)/* Thread function */
{
    sem_wait(&m); /* lock  m */
    x = x + 1;     /* sezione critica */
    sem_post(&m); /* unlock m */
}
void main ()
{
    pthread_t tid[10];
    int i;
    if (sem_init(&m, 0, 1) == -1) {   /* inizializza il semaforo a 1 */
        printf("non posso inizializzare il semaforo");
        exit(2);
    }
    for (i=0; i<10; i++)/* crea 10 threads */
    {
        if (pthread_create(&tid[i], NULL, thread, NULL) < 0) {
        printf("Error: thread cannot be created");
        exit(1);
    }
    for (i=0; i<10; i++) pthread_join(tid[i], NULL);
    printf("valore finale di x = %d\n", x);
    exit(0);
}
```

```
#include <stdio.h>    /* simula un semaforo usando mutex e variabili condizione */
#include <pthread.h>
#include <semaphore.h>

typedef struct {
     pthread_mutex_t mutex;
     pthread_cond_t cond;
     int counter;
} mysem_t;
mysem_t mysem;

void *body(void *arg) {
     int i,j;
     for (j=0; j<40; j++) {
          mysem_wait(&mysem);
          for (i=0; i<1000000; i++);  fprintf(stderr,(char *)arg);
          mysem_post(&mysem);
     }
     return NULL;
}

int main() {
     pthread_t t1,t2,t3;
     pthread_attr_t myattr;
     int err;

     mysem_init(&mysem,1);
     pthread_attr_init(&myattr);
     err = pthread_create(&t1, &myattr, body, (void *)".");
     err = pthread_create(&t2, &myattr, body, (void *)"#");
     err = pthread_create(&t3, &myattr, body, (void *)"o");
     pthread_attr_destroy(&myattr);

     pthread_join(t1, NULL);  pthread_join(t2, NULL);   pthread_join(t3, NULL);
     printf("\n");    return 0;
}
```

```
void unlock_mutex(void *m)
{   pthread_mutex_unlock((pthread_mutex_t *)m); }

void mysem_init(mysem_t *s, int num) {
    pthread_mutexattr_t m;
    pthread_condattr_t c;

    s->counter = num;
    pthread_mutexattr_init(&m);
    pthread_mutex_init(&s->mutex, &m);
    pthread_mutexattr_destroy(&m);
    pthread_condattr_init(&c);
    pthread_cond_init(&s->cond, &c);
    pthread_condattr_destroy(&c);
}

void mysem_wait(mysem_t *s) {
    pthread_mutex_lock(&s->mutex);
    while (!(s->counter)) pthread_cond_wait(&s->cond, &s->mutex);

    s->counter--;
    pthread_mutex_unlock(&s->mutex);
}

void mysem_post(mysem_t *s) {
    pthread_mutex_lock(&s->mutex);
    if (!(s->counter++)) pthread_cond_signal(&s->cond);

    pthread_mutex_unlock(&s->mutex);
}
```

Una soluzione semaforica al problema produttore-consumatore



Schema:
3 semafori: 1 binario, mutex
1 contatore, empty inizializzato a N
1 contatore, full inizializzato a 0

Produttore:

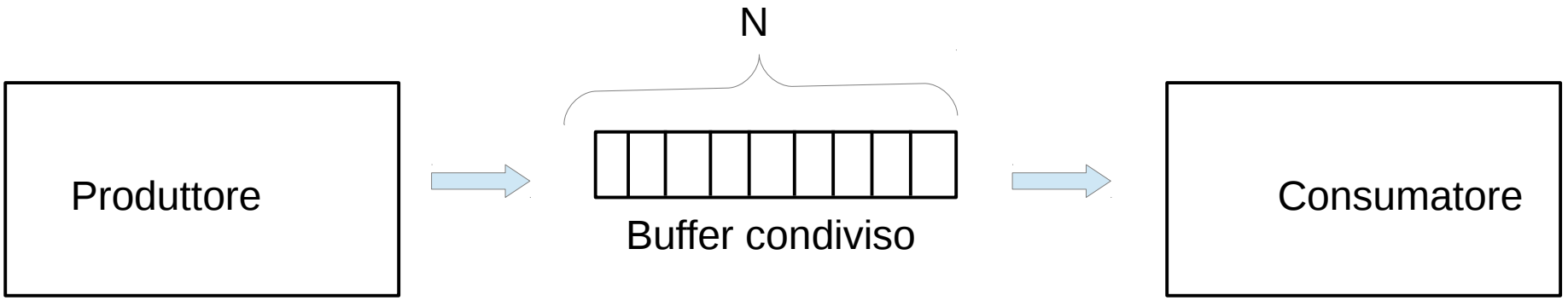
```
el=produci();
down(mutex);
down(empty) ;
Inserisci el;
up(full)    ;
up(mutex);
```

ATTENZIONE:
Questa soluzione porta allo stallo del prod e del cons

Consumatore:

```
down(mutex);
down(full);
estrai el;
up(empty);
up(mutex);
consuma(el);
```

Una soluzione semaforica al problema produttore-consumatore



Schema:
3 semafori: 1 binario, mutex
1 contatore, empty inizializzato a N
1 contatore, full inizializzato a 0

Produttore:

```
elp=produci();

down(empty) ;
down(mutex);
Inserisci elp;
up(mutex);
up(full)    ;
```

Consumatore:

```
down(full);
down(mutex);
estrai elc;
up(mutex);
up(empty);
consuma(elc);
```

Codice d'esempio:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAXITER 100
sem_t mutex,pieno,vuoto;
char arr[10]; int in,out;

int main(){
  pthread_t prod,cons;
  pthread_attr_t attr;
  pthread_attr_init(&attr);

  int err;

  sem_init(&mutex,0,1);
  sem_init(&pieno,0,0);
  sem_init(&vuoto,0,10);

  in=0; out=0;


  pthread_create(&prod, &attr, produci, NULL);

  pthread_create(&cons, &attr, consuma, NULL);

  pthread_attr_destroy(&attr);
  pthread_join(prod, NULL);
  pthread_join(cons, NULL);
  printf("Joined!\n");
  return 0;
}
```

```
void *consuma(){
  int i=0;
  while (i<MAXITER){
    i++;

    sem_wait(&pieno);
    sem_wait(&mutex);
    char c = arr[out];
    printf("Prelevo %c da %d.\n",c,out);
    out = (out+1) % 10;
    sem_post(&mutex);
    sem_post(&vuoto);
  }
}

void *produci(){
  int i=0;
  while (i<MAXITER){
    i++;
    char c =(char) (random() % 26)+97;

    sem_wait(&vuoto);
    sem_wait(&mutex);
    arr[in]=c;
    printf("Inserito %c in  %d.\n",c,in);
    in = (in+1) % 10;
    sem_post(&mutex);
    sem_post(&pieno);
  }
}
```