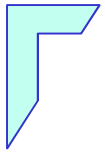
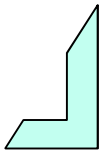


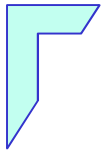
Esercitazione di Lab. di Sistemi Operativi a.a. 2012/2013

- Thread (POSIX) pthread e Mutex -

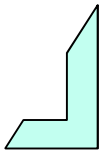


Sommario

- Thread Standard ANSI/IEEE POSIX 1003.1 (1990) pthread:
  - Cosa è?
  - Funzioni dello standard posix:
    - Creazione: funzione: **pthread_create**
    - Terminazione esplicita: funzione: **pthread_exit**
    - Attesa terminazione: funzione: **pthread_join**
- Mutex
  - mutex (semaforo binario)
    - per sezione critica
    - per struttura
  - condition variable
- Esercizi:
  - Thread
  - Mutex



- Cosa è un Thread -



- Cosa è un Thread -

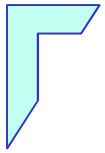
- Un "thread" è una parte di un processo, che quando viene creato in analogia ai processi, viene associato ad un pezzo di codice

- Se il processo ospite (e quindi il suo programma) è mono-thread il thread è associato all'intero programma.

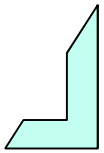
- Se il processo ospite è **multi-thread** ciascun thread di tale processo, è associato ad una **funzione** da eseguire chiamata **funzione** di avvio.

- Quando un programma viene mandato in esecuzione tramite una chiamata exec, viene creato un singolo thread detto thread principale

- Ulteriori thread vanno creati esplicitamente



- Thread: creazione -



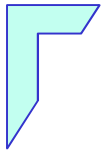
- Creazione Thread -

- Funzione "**pthread_create**"

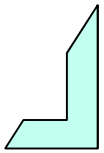
Per creare thread addizionali relativi ad uno stesso processo, Posix prevede la funzione:

```
#include <pthread.h>

int  pthread_create ( pthread_t *tid, const pthread_attr_t *attr,
                      void * ( *start_func) (void *), void *arg );
```

- se la chiamata ha successo, *tid* punta al thread ID;
- *attr* permette di specificare gli attributi del thread (se *attr* = NULL, gli attributi sono quelli di default);
- *start_func* è l'indirizzo della funzione di avvio;
- *arg* è l'indirizzo dell'argomento accettato dalla funzione di avvio;
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.



- Thread: Terminazione esplicita -



- Terminare un Thread -

Un thread può richiedere esplicitamente la propria terminazione grazie alla chiamata seguente, lasciando traccia del proprio stato di terminazione per quei thread che attendono per lui:

```
#include <pthread.h>

void  pthread_exit ( void *status);
```

•*status* punta all'oggetto che definisce lo stato di terminazione del thread. Quest'ultimo non deve essere una variabile locale al thread chiamante, pena la sua scomparsa alla terminazione del thread stesso.

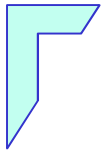
- Esempio 1: Creazione e Terminazione Thread -

```
#include <pthread.h>
void *thread(void *vargp);
int main() {
pthread_t tid;

   pthread_create(&tid, NULL, thread, NULL);
   exit(0);
}
/* thread routine */
void *thread(void *vargp) {
printf("Hello, world!\n");
pthread_exit((void*)status);
}
```

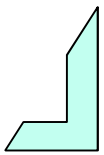
attributi Thread

argomenti Thread

Fa terminare il Thread corrente



- Thread: Attesa terminazione -



- Aspettare la terminazione di un Thread -

Un thread può attendere per la terminazione di un altro thread relativo allo stesso processo:

```
#include <pthread.h>

int  pthread_join ( pthread_t  tid, void **status );
```

- *tid* è l'ID del thread del quale si vuole attendere la terminazione;
- *status* punta al valore restituito dal thread per cui si è atteso, indicante il suo stato di terminazione (se *status* = NULL, tale stato non viene restituito);
- restituisce 0 in caso di successo, un intero positivo – secondo le convenzioni di <sys/errno.h> – in caso di errore.

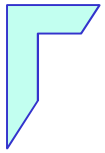
- Esempio 2 Creazione e attesa Thread -

```
#include<pthread.h>
#include<stdlib.h>
#include <unistd.h>
#include <stdio.h>
void *thread(void *vargp);
int main() {
pthread_t tid;
int i;
   pthread_create(&tid, NULL, thread, NULL);
   pthread_join(tid, (void *)&i);//attesa thread

   printf("%d\n",i);
   exit(0);
}
/* thread routine */
void *thread(void *vargp) {
printf("Hello, world!\n");
pthread_exit((void*)status);//termina il thread
}
```

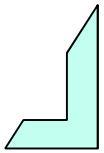
argomenti Thread

attributi Thread

Valore di ritorno



- Esercizi: Thread
Creazione e Terminazione -



Esercizio n° 1 – Creazione e Terminazione Thread –

Scrivere un programma C che crei un "**Thread**" ed attende la terminazione dello stesso. Il thread scriverà a video in 20 secondi i numeri da 0 a 19.

Compilazione:

$ gcc nomeProgramma.c -lpthread

Opzione di compilazione che serve a linkare la libreria `pthread.h` nel programma

Esecuzione

$ ./a.out

Output

Scrivo il numero: 0 1 2,……..10,……19

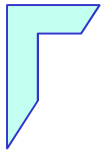
Soluzione Esercizio n° 1 - Thread -

Programma con due thread

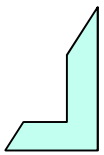
```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
void *thread_function(void *arg)
{  int i;
   void *val;
  for ( i=0; i<20; i++ )
    { printf("Scrivo il Numero: %d\n",i);
      sleep(1); //ripete ogni secondo
    }
pthread_exit(val);}
int main(void)  //thread main o principale
{  pthread_t tid;
    void *status;
if (pthread_create(&tid, NULL, thread_function, NULL) )
{  printf("error creating thread.");
    exit(1);  }
printf("Sto aspettando la terminazione del thread\n");
if ( pthread_join (tid, (void*)&status ) ) {
    printf("error joining thread.");
exit(1);  }
exit(0);}
```

Esercizio n° 1 – Creazione e Terminazione Thread –

- Il main
  - dichiara una variabile di tipo **pthread_t** definito nella libreria **pthread.h** usata per tid (id thread)
  - chiama la funzione **pthread_create** per creare un nuovo processo thread associato alla funzione **thread_function**
- Quindi abbiamo due thread:
  - quello principale (main thread)
  - ed il nuovo thread (**thread_function**) che lavorano in parallelo
- Il main thread, continua nella sua esecuzione eseguendo l'istruzione successiva **pthread_join**, mentre il nuovo thread impiega 20 secondi prima di terminare
- Il main si addormenta in attesa che il nuovo thread termini



- Esercizio Thread: condivisione di variabili -



Esercizio n° 2 – Thread – (condivisione di var.)

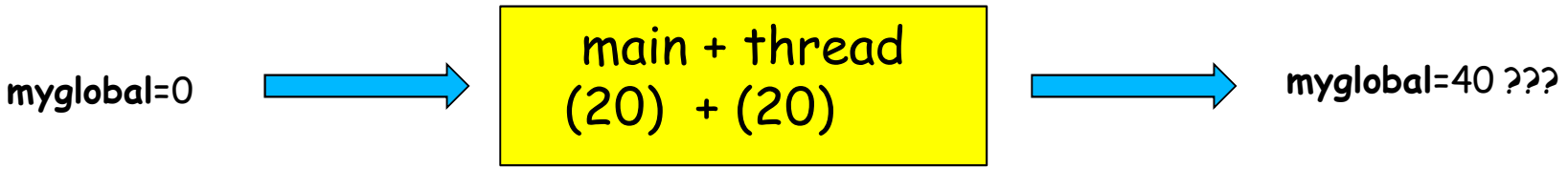
Scrivere un programma C che crei un **Thread** il quale insieme al programma **main** incrementa di venti volte una variabile globale (myglobal). Il thread chiama una funzione "**incrementa**" Supporremo che l' accesso alla variabile globale non venga regolamentato da alcun meccanismo di gestione della concorrenza.

Compilazione:

```
$ gcc nomeProgramma.c  -lpthread
```

Esecuzione

```
$ ./a.out
```



Soluzione Esercizio n° 2 - Thread -

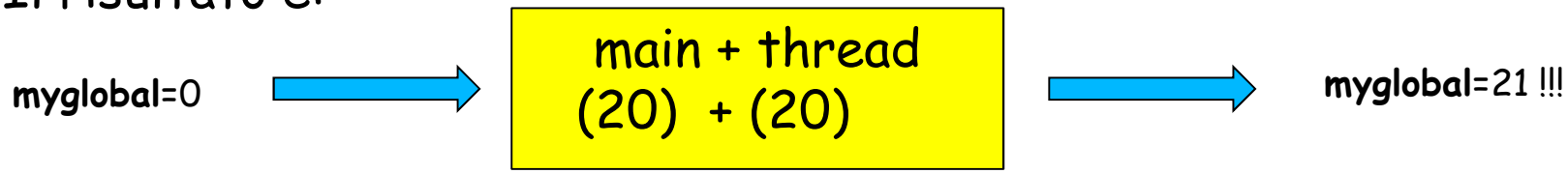
```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;//Variabile globale da incrementare

void *thread_incrementa(void *arg)
{  int i,j;
   void *val;
for ( i=0; i<20; i++ )
{
   j=myglobal; //variabile locale temporanea
   j=j+1;
   printf("thread_incrementa\n");
   fflush(stdout);
   sleep(1); //Si addormenta per 1 secondo
   myglobal=j;
}
pthread_exit(val);
}
```

Soluzione Esercizio n° 2 - Thread -

```
int main(void)
{
pthread_t mythread;
int i;
void *status;
if(pthread_create(&mythread,NULL,thread_incrementa,NULL) )
{ printf("error creating thread.");
    exit(1);
}
for ( i=0; i<20; i++) {
myglobal=myglobal+1;
printf("main\n");
fflush(stdout);
sleep(1); //Si addormenta per 1 secondo
}
if(pthread_join(mythread,(void*)&status)){
printf("error joining thread.");
exit(1);  }
printf("nmyglobal uguale %d\n",myglobal);
exit(0);
}
```

Esercizio n° 2 - Thread -

- Il risultato è:



Perché???

```
void *thread_incrementa(void *arg)
{   int i,j;
    void *val;
for ( i=0; i<20; i++ )
{
    j=myglobal; //variabile locale temporanea
    j=j+1;
    printf("thread_incrementa\n");
    fflush(stdout);
    sleep(1); //Si addormenta per 1 secondo
    myglobal=j;
}
pthread_exit(val);
}
```

Esercizio n° 2 - Thread -

- La funzione **thread_incrementa**:
  1. copia myglobal in una variabile locale j
  2. incrementa per venti volte tale variabile
  3. si addormenta per un secondo (sleep(1))
  4. copia il valore della variabile j in myglobal

- Il **main**:
  1. Incrementa per venti volte la variabile myglobal in maniera parallela al thread

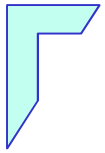
- Il punto (4) della funzione **thread_incrementa** provoca una sovrascrittura del valore presente in myglobal relativo al main (annulla ripetutamente l'incremento di myglobal da parte del main)

Esercizio n° 2 – Thread –

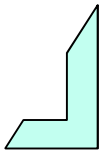
- Soluzioni al problema:

1. Evitare di modificare il valore di una variabile globale passando per una variabile locale e quindi incrementare direttamente la stessa (soluzione non corretta anche se nel nostro caso funziona)

2. Sincronizzazione di thread: i **mutex**



- Sincronizzazione -

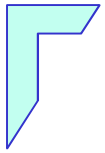


- Perché Sincronizzazione -

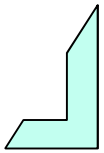
- E' necessario attuare meccanismi di sincronizzazione quando più **thread** accedono a risorse condivise come:
  - variabili e/o strutture globali (statiche e dinamiche)

- In questo caso, infatti il risultato non è prevedibile perché le operazione fatte da thread su shared memory non sono atomiche, cioè non si ha la garanzia che il cambiamento di stato di una variabile da parte di un processo vada a buon fine senza che un altro processo si intrometta.

- L'accesso a strutture condivise da parte di più processi deve essere quindi disciplinato da meccanismi di sincronizzazione:
  - mutex (semaforo binario)
    - per sezione critica e per struttura
  - variabili di condizione



- I Mutex -



- I mutex -

- Sono semafori binari di mutua esclusione che consentono di evitare **race condition** (interferenza tra processi) su risorse condivise.

- Vengono applicati a parti del codice di accesso a più thread (sezioni critiche) facendo in modo che tali sezioni critiche non vengano mai eseguite contemporaneamente.

- Quindi solo un thread alla volta può accedere ad una risorsa condivisa protetta da mutex

- Hanno due soli stati (semafori binari):
  - aperto (unlock) o chiuso (lock)

Inizializzare i mutex

- Nella libreria pthread.h un mutex è una variabile del tipo **pthrad_mutex_t**
- prima dell' uso bisogna inizializzare il mutex:
- Inizializzazione:
  - STATICA:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

  - DINAMICA:

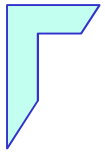
```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
```

  - Parametri della funzione **pthread_mutex_init:**
    - **mutex:** puntatore al mutex da inizializzare
    - **attr:** puntatore agli attributi del mutex (di solito NULL)

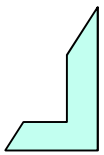
Usare i mutex

```
int pthread_mutex_lock      (pthread_mutex_t *mutex);
int pthread_mutex_trylock   (pthread_mutex_t *mutex);
int pthread_mutex_unlock    (pthread_mutex_t *mutex);
```

- acquisiscono e rilasciano il semaforo
- restituiscono 0 se OK, un codice d'errore altrimenti
- se il semaforo è occupato (locked)...
  - ...lock blocca il thread finché il semaforo si libera
  - ...trylock invece non blocca, ma restituisce subito l'errore EBUSY

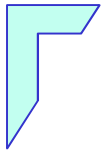


- I mutex:
per sezione critica e per struttura -

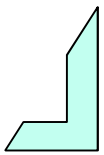


- I mutex: sezione critica e per struttura -

- I mutex possono essere:
  - Per **sezione critica**
    - **Solo** quando una struttura condivisa, viene modificata in un **unico punto** del codice (vedi esercizio N° 2)
    - In questo caso è sufficiente associare un mutex alla "sezione critica"
  - Per **struttura**
    - **Solo** quando una struttura condivisa, viene modificata in **più punti** del codice
    - Utile se più strutture devono essere condivise contemporaneamente
    - In questo caso è sufficiente associare un mutex alla "struttura"



- I mutex: per sezione critica -



Esempio 1: - mutex per sezione critica -

- Creare e inizializzare staticamente i mutex (tramite macro)

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Individuare la sezione critica: parte di codice globale

```
int myglobal;
```

- Applicare il mutex alla sezione critica (memoria condivisa da thread)

```
main
for ( i=0; i<20; i++) {
pthread_mutex_lock(&mymutex);
myglobal=myglobal+1;        sezione critica
pthread_mutex_unlock(&mymutex);
printf("main\n");
fflush(stdout);
sleep(1);
}
```

```
Nuovo Thread
for ( i=0; i<20; i++ )
{pthread_mutex_lock(&mymutex);
    j=myglobal;
    j=j+1;
    fflush(stdout);      sezione critica
    sleep(1);
    myglobal=j;
 pthread_mutex_unlock(&mymutex);
}
```

Esercizio n° 3 – Thread – (mutex per sezione critica)

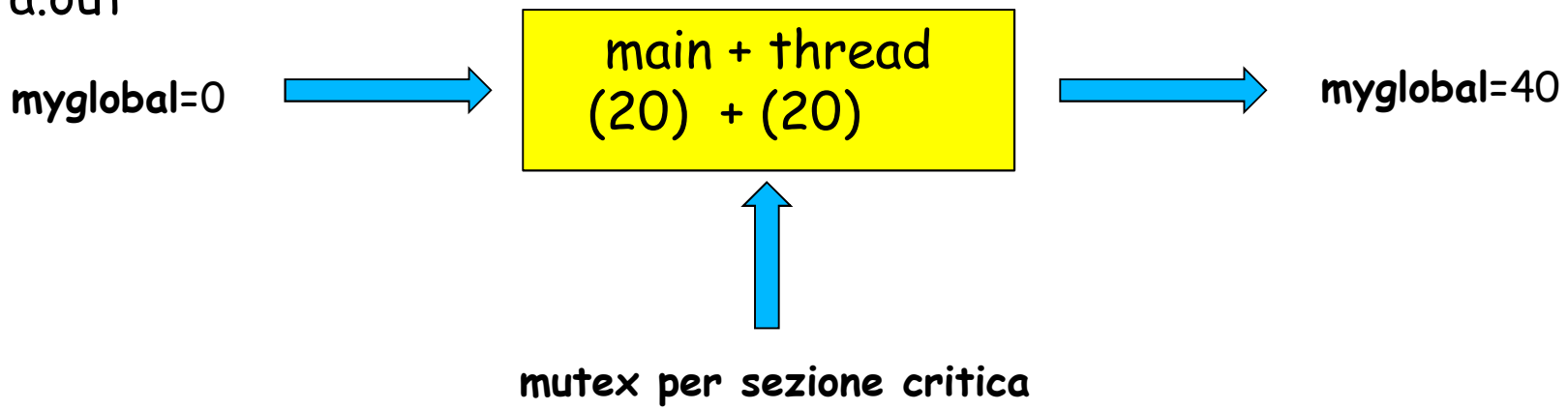
Scrivere un programma C che crei un **Thread** il quale insieme al programma **main** incrementa di venti volte una variabile globale (myglobal). Il thread chiama una funzione "**incrementa**" .
Si utilizzi come meccanismo di gestione della **concorrenza** quello della "**muta esclusione per sezione critica**", dove la struttura mutex va allocata staticamente

Compilazione:

```
$ gcc nomeProgramma.c  -lpthread
```

Esecuzione

```
$ ./a.out
```



Soluzione Esercizio n° 3 - Thread -

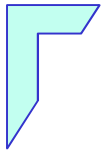
```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal; //Variabile globale
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
void *thread_incrementa(void *arg)
{  int i,j;
   void *val;
for ( i=0; i<20; i++ )
{  pthread_mutex_lock(&mymutex);
   j=myglobal; //variabile locale temporanea
   j=j+1;
   printf("thread_incrementa()\n");
   fflush(stdout);
   sleep(1); //Si addormenta per 1 secondo
   myglobal=j;
   pthread_mutex_unlock(&mymutex);
}
pthread_exit(val);}
```

sezione critica

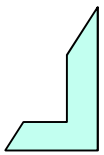
Soluzione Esercizio n° 3 - Thread -

```
int main(void)
{pthread_t mythread;
int i;
void *status;
if(pthread_create(&mythread,NULL,thread_incrementa,NULL) )
{ printf("error creating thread.");
    exit(1);
}
for ( i=0; i<20; i++) {
pthread_mutex_lock(&mymutex);
myglobal=myglobal+1;
pthread_mutex_unlock(&mymutex);
printf("main()\n");
fflush(stdout);
sleep(1); //Si addormenta per 1 secondo
}
if(pthread_join(mythread,(void*)&status)){
printf("error joining thread.");
exit(1);  }
printf("nmyglobal uguale %d\n",myglobal);
exit(0);}
```

sezione critica



- I mutex: per struttura -



Esempio 2: - mutex per struttura -

- Creare i mutex all' interno della struttura da preservare:

```
typedef struct example{
    int a;
    int b;
    pthread_mutex_t  mymutex;
} myexample
```

- Inizializzare dinamica del mutex:

```
myexample *init_struct(){
malloc(sizeof(myexample));
    fp->a=0;
    fp->b=0;
pthread_mutex_init(&fp->mymutex,NULL);
}
```

- Blocco della zona critica:

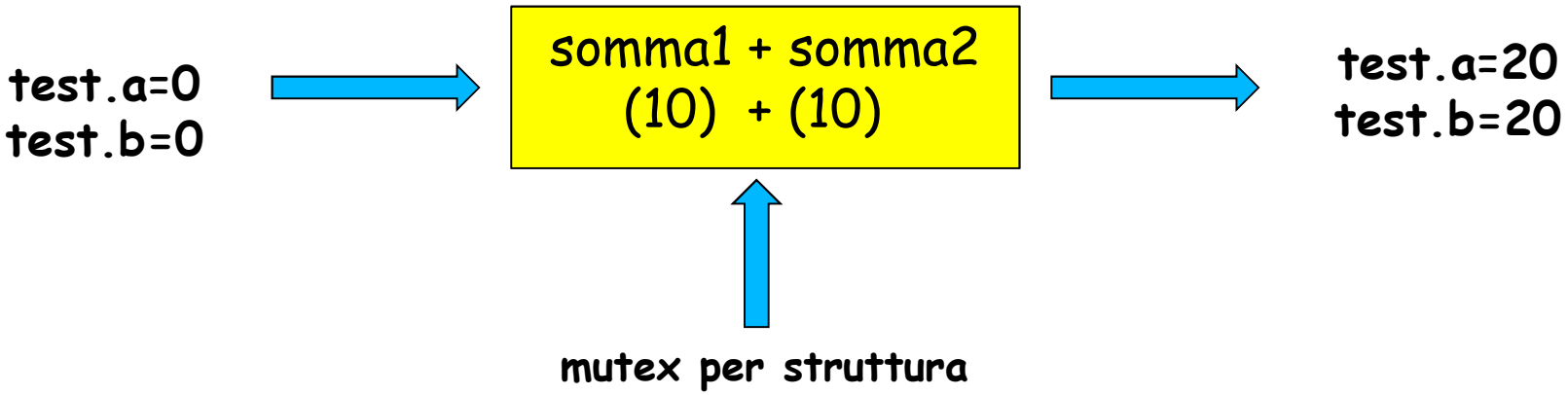
```
pthread_mutex_lock(&fp->mymutex);
    fp->a++;
    fp->b++;
pthread_mutex_unlock(&fp->mymutex);
```

Esercizio n° 4 - Thread -

- Scrivere un programma C che crea due Thread "**somma1**" e "**somma2**", entrambi accedono alle variabili **test.a** e **test.b** di una struttura dati **test** condivisa incrementandole di 1 per 10 volte, aspettano 2 secondi prima di stampare a video i valori delle due variabile. Si utilizzi come meccanismo di gestione della **concorrenza** quello della "**muta esclusione per struttura**", dove la struttura mutex va allocata **dinamicamente**.

Compilazione:

$ gcc nomeProgramma.c -lpthread



Soluzione Esercizio n° 4 - Thread -

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#define CICLI 10
typedef struct foo{    /* Struttura Condivisa fra i thread */
int a;
int b;
pthread_mutex_t  sem;       /*mutex  inserito  all'interno  della  struttura  da
controllare*/
} myfoo;

myfoo *test; // Variabile GLOBALE Puntatore alla Struttura
myfoo *init_struct(){
struct foo *fp; //Puntatore alla struttura
if((fp=malloc(sizeof(myfoo)))==NULL)/*Allocazione dinamica Struttura */
    return(NULL);
        fp->a=0; /*inizializzazione  Struttura */
        fp->b=0;
pthread_mutex_init(&fp->sem,NULL);/*inizializzazione       dinamica    del
mutex*/
return(fp);}
```

Soluzione Esercizio n° 4 - Thread -

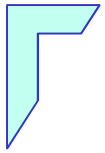
```
void *somma1(void *);
void *somma2(void *);
int main(void)
{
  pthread_t som1TID, som2TID;
  test=init_struct(); //Inizializziamo la struttura condivisa

  if (pthread_create(&som1TID, NULL, somma1, NULL) != 0) {
       printf ("Errore nella creazione del thread somma1\n");
       exit(1);}
  if (pthread_create(&som2TID, NULL, somma2, NULL) != 0) {
       printf ("Errore nella creazione del thread somma2\n");
       exit(1);}
  pthread_join(som1TID, NULL);
  pthread_join(som2TID, NULL);
  printf("valore -- a = %d \n", test->a);
  printf("valore -- b = %d \n", test->b);
  pthread_mutex_destroy(&test->sem); /*distrugge il mutex*/
  exit (0);
}
```

Soluzione Esercizio n° 4 - Thread -

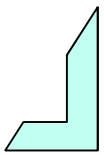
```
void *somma1(void *in)
{
  int i;
  for(i=0; i < CICLI; i++) {
    pthread_mutex_lock(&test->sem);
    test->a++;
    test->b++;
    pthread_mutex_unlock(&test->sem);
}
  pthread_exit(0);}
void *somma2(void *in)
{
  int i;
  for(i=0; i < CICLI; i++) {
    pthread_mutex_lock(&test->sem);
    test->a++;
    test->b++;
    pthread_mutex_unlock(&test->sem);
}
  pthread_exit(0);}
```

sezione critica

sezione critica



- Meccanismi di Sincronizzazione: Variabili di condizione -



- Variabili di condizione -

- Le variabili condizione (condition) sono uno strumento di **sincronizzazione** che permette 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
- **Operazioni fondamentali:**
  - inizializzazione: **pthread_cond_init**
  - sospensione: **pthread_cond_wait**
  - risveglio: **thread_cond_signal**

- Variabili di condizione: Inizializzazione -

- L'inizializzazione di una condition si realizzare:
  - In maniera **statica** utilizzando la macro **PTHREAD_COND_INIZIALIZER**
  - **Esempio d'uso**:

```
pthread_cond_t C = PTHREAD_COND_INIZIALIZER;
```

  - Oppure in maniera **dinamica** con: **pthread_cond_init**
  - **Esempio d'uso:**

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

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

- Variabili di condizione: wait -

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

- **Esempio d'uso**:

```
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

- Effetto:
  - blocca il thread chiamante sulla coda associata a **cond**, e il mutex **mux** viene sbloccato in modo da liberare la sezione critica ad un altro thread
  - Questa funzione deve essere chiamata quando il mutex è bloccato che verrà poi rilasciato durante l'attesa.

- Al successivo risveglio (provocato da una signal), il thread bloccherà il mutex **mux** automaticamente.

- Variabili di condizione: signal -

- Il risveglio di un thread sospeso su una variabile condizione può essere ottenuto mediante la funzione: **pthread_cond_signal**
  - **Esempio d'uso**:

```
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 svegliato 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 si usa: **pthread_cond_broadcast**

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

Esempio 3:- mutex con Variabile di condizione -

- Creare ed inizializzare i mutex e la variabile di condizione:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  v = PTHREAD_COND_INIZIALIZER;
```

- Associare il mutex alla sezione critica ed utilizzare la variabile di condizione per sospendere eventualmente il thread:

```
pthread_mutex_lock(&m);//blocco mutex
while ( x != y ) //se true sospensione thread
    pthread_cond_wait(&v, &m);//sospensione thread e sblocco del mutex
**********SEZIONE CRITICA*******
pthread_mutex_unlock(&m);//Sblocco del mutex
```

- Parallelamente:

```
pthread_mutex_lock(&m); //blocco mutex
    x++;
pthread_cond_signal(&v);//Risveglia il thread sospeso
pthread_mutex_unlock(&m); //Sblocco del mutex
```

Esempio 4: - mutex con Variabile di condizione -

Stop thread su condizione:

```
while ( x != y ):
1.  blocco mutex
2.  test condizione (x==y)
3.   se TRUE, sblocco  mutex e exit loop
4.   se FALSE, sospendere thread and sblocco mutex
```

Esercizio n° 5 – Thread –

Scrivere un programma C, che crei due Thread, chiamati **produttore** e **consumatore**. La risorsa condivisa, è un **buffer circolare** di dimensione data (ad esempio 20) il cui **stato** è :

- numero di elemento contenuti: **count**
- puntatore alla prima posizione libera : **writepos**
- puntatore al primo elemento occupato: **readpos**

Il **produttore**, inserisce,20 numeri interi in maniera sequenziale.
Il **consumatore** li estrae sequenzialmente per stamparli.
Il programma dovrà prevedere:

- un meccanismo di accesso controllato alla risorsa **buffer** da parte dei due Thread (mutex per il controllo della mutua esclusione nell' accesso al buffer)
- una sincronizzazione tra il **produttore** ed il **consumatore** (Thread) in caso di
  - **buffer pieno**: definizione di una condition per la sospensione del **produttore** se il buffer è pieno (notfull)
  - **buffer vuoto**: definizione di una condition per la sospensione del **consumatore** se il buffer è vuoto (notempty)

Esercizio n° 5 - Thread -

Output:

sono il thread produttore

Thread produttore scrivo 0 --->
……
……
……
……
Thread produttore scrivo 20 --->

I venti numeri scritti dal thread produttore nel buffer

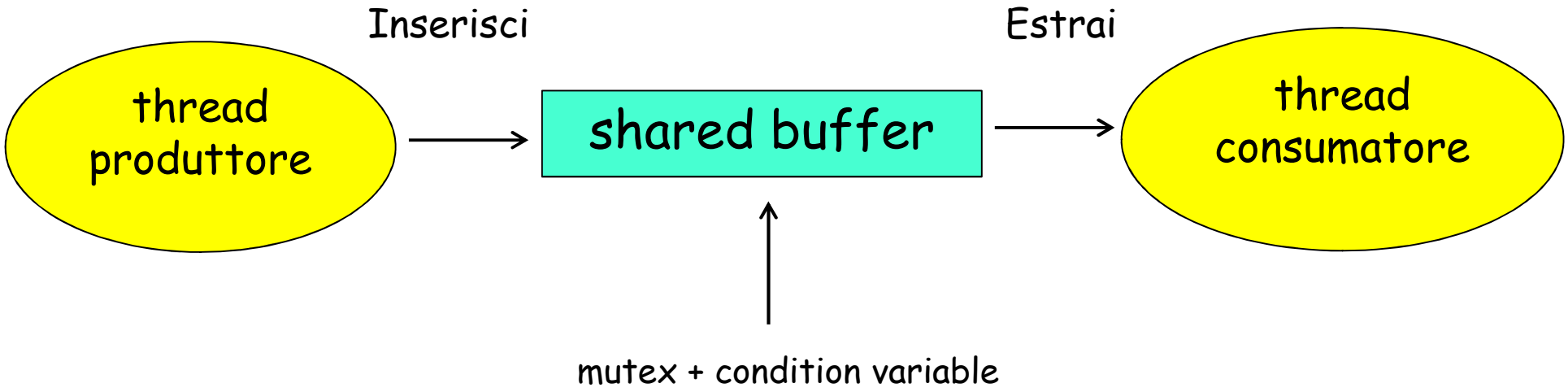
sono il thread consumatore

Thread consumatore leggo 0 --->
……
……
……
……
Thread consumatore leggo 20 --->

I venti numeri letti dal thread consumatore nel buffer

Esercizio n° 5 - Thread -

Schema produttore/consumatore



Soluzione Esercizio n° 5 – Thread -

```
#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;   /*dichiarazione delle variabili di condizione*/
        pthread_cond_t notfull;
}prodcons;

prodcons buffer;
```

Soluzione Esercizio n° 5 - Thread -

```
/* Inizializza il buffer */
void init (prodcons *b)
{
    pthread_mutex_init  (&b->lock,  NULL);    /*inizializza  il
mutex*/
     /*inizializza le condition variable*/
    pthread_cond_init (&b->notempty, NULL);
    pthread_cond_init (&b->notfull, NULL);
    b->cont=0;
    b->readpos = 0;
    b->writepos = 0;
}
```

Soluzione Esercizio n° 5 - Thread -

```
/* Inserimento elementi nel buffer*/
void inserisci (prodcons *b, int data)
{
    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);
    /* scrivi data e aggiorna lo stato del buffer */
    b->buffer[b->writepos] = data;
    b->cont++;
    b->writepos++;
    if (b->writepos >= BUFFER_SIZE)
        b->writepos = 0;
    /* risveglia eventuali thread (consumatori) sospesi */
    pthread_cond_signal (&b->notempty);
    pthread_mutex_unlock (&b->lock);//mutex sbloccato
}
```

sezione critica

Soluzione Esercizio n° 5 - Thread -

```
/*Estrazione elementi dal buffer*/
int estrai (prodcons *b)
{
    int data;
    pthread_mutex_lock (&b->lock); //mutex bloccato
    while (b->cont==0) /* il buffer e` vuoto? */
        pthread_cond_wait (&b->notempty, &b->lock);
    /* Leggi l'elemento e aggiorna lo stato del buffer*/
    data = b->buffer[b->readpos];
    b->cont--;
    b->readpos++;
    if (b->readpos >= BUFFER_SIZE)
        b->readpos = 0;
    /* Risveglia eventuali threads (produttori) sospesi*/
    pthread_cond_signal (&b->notfull);
    pthread_mutex_unlock (&b->lock); //mutex sbloccato
    return data; /*Ritorna il dato letto dal buffer*/
}
```

sezione critica

Soluzione Esercizio n° 5 – Thread -

```
void *producer (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;
}
```

Soluzione Esercizio n° 5 - Thread -

```
void *consumer (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;
}
```

Soluzione Esercizio n° 5 - Thread -

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

- Fine Esercitazione -