I thread nel sistema operativo LINUX: *Linuxthreads*

LinuxThreads: Caratteristiche

- Processi leggeri **realizzati a livello kernel**
- System call `clone`:

```
int  clone(int (*fn) (void *arg), void *child_stack, int
       flags, void *arg)
```

➔ **E` specifica di Linux: scarsa portabilita`!**

- Libreria *LinuxThreads*: funzioni di gestione dei threads, in conformita` con lo standard POSIX 1003.1c (*pthreads*):
  - *Creazione/terminazione threads*
  - *Sincronizzazione threads:lock, [semafori], variabili condizione*
  - *Etc.*

➢ Portabilita`

LinuxThreads

- Caratteristiche threads:
  - Il thread e` realizzato **a livello kernel** (e` l'unita` di schedulazione)
  - I thread vengono creati all'interno di un processo (task) per eseguire una funzione
  - Ogni thread ha il suo PID (a differenza di POSIX: distinzione tra *task* e *threads*)
  - Gestione dei segnali non conforme a POSIX:
    - Non c'e` la possibilita` di inviare un segnale a un task.
    - SIGUSR1 e SIGUSR2 vengono usati per l'implementazione dei threads e quindi non sono piu` disponibili.
- Sincronizzazione:
  - **Lock**: mutua esclusione (**`pthread_mutex_lock/unlock`**)
  - **Semafori**: esterni alla libreria **`pthread`** <semaphore.h> (POSIX 1003.1b)
  - **Variabili condizione** : (**`pthread_cond_wait,pthread_cond_signal`**)

Rappresentazione dei threads

- Il thread e` l'unita` di scheduling, ed e` univocamente individuato da un indentificatore (intero):

```
pthread_t tid;
```

  - Il tipo **`pthread_t`** **e`dichiarato nell'header file `<pthread.h>`**

LinuxThreads

- Creazione di threads:

```
#include <pthread.h>
int pthread_create(pthread_t *thread,pthread_attr_t *attr,
                   void *(*start_routine)(void *), void * arg);
```

- **Dove:**
  - **`thread`: e` il puntatore alla variabile che raccogliera` il thread_ID (PID)**
  - **`start_routine`: e` il puntatore alla funzione che contiene il codice del nuovo thread**
  - **`arg`: e` il puntatore all'eventuale vettore contenente i parametri della funzione da eseguire**
  - **`attr`: puo` essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):**
    - ad esempio parametri di scheduling: priorita` etc.(solo per superuser!)
    - Legame con gli altri threads (ad esempio: *detached* o no)
- Ritorna : 0 in caso di successo, altrimenti un codice di errore (!=0)

LinuxThreads: creazione di threads

Ad esempio:

```
int A, B;
void * codice(void *){ /*definizione del codice del thread */ …}
main()
{pthread_t t1, t2;
..
pthread_create(&t1,NULL, codice, NULL);
pthread_create(&t2,NULL, codice, NULL);
..
}
```

- **Vengono creati due thread (di tid `t1` e `t2`) che eseguono le istruzioni contenute nella funzione `codice`:**
  - **I due thread appartengono allo stesso *task* (processo) e condividono le variabili globali del programma che li ha generati (ad esempio A e B).**

LinuxThreads

- Un thread puo` terminare chiamando:

```
void pthread_exit(void *retval);
```

- Dove:
  - **`retval`: e` il puntatore alla variabile che contiene il valore di ritorno (puo` essere raccolto da altri threads, v. `pthread_join`).**
- E` una chiamata senza ritorno.
- Alternativa: **`return();`**

LinuxThreads

- Un thread puo` sospendersi in attesa della terminazione di un altro thread con:

```
int pthread_join(pthread_t th, void **thread_return);
```

- Dove:
  - **`th`: e` il pid del particolare thread da attendere**
  - **`thread_return`: e` il puntatore alla variabile dove verra` memorizzato il valore di ritorno del thread (v. `pthread_exit`)**

Esempio: creazione di thread

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}
```



```
main ()
{
  pthread_t th1, th2;
  int retcode;
  if (pthread_create(&th1,NULL,my_thread_process,"1") < 0)
  { fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  if (pthread_create(&th2,NULL,my_thread_process,"2") < 0)
  { fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  retcode = pthread_join (th1, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 1\n);

  retcode = pthread_join (th2, NULL);
  if (retcode != 0)
    fprintf (stderr, "join fallito %d\n", retcode);
  else printf("terminato il thread 2\n);
  return 0;
}
```

Compilazione

- Per compilare un programma che usa i linuxthreads:

```
gcc -D_REENTRANT -o prog prog.c   -lpthread
```

```
[aciampolini@ccib48 threads]$ prog
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
terminato il thread 1
terminato il thread 2
[aciampolini@ccib48 threads]$
```


Terminazione di threads

- Normalmente e` necessario eseguire la **`pthread_join`** per ogni thread che termina la sua esecuzione, altrimenti rimangono allocate le aree di memoria ad esso assegnate.

- In alternativa si puo` "staccare" il thread dagli altri con:

```
int pthread_detach(pthread_t th);
```

  - il thread rilascia automaticamente le risorse assegnatagli quando termina.

LinuxThreads: 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 100
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);
}
```

LinuxThreads: Semafori

- **Memoria condivisa**: uso dei semafori (POSIX.1003.1b)
  - Semafori: libreria <semaphore.h>
    - **`sem_init`: inizializzazione di un semaforo**
    - **`sem_wait`: *wait***
    - **`sem_post`: *signal***
  - **`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

- **wait su un semaforo**

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

dove:

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

e` la *wait* 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

- ***signal* su un semaforo:**

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

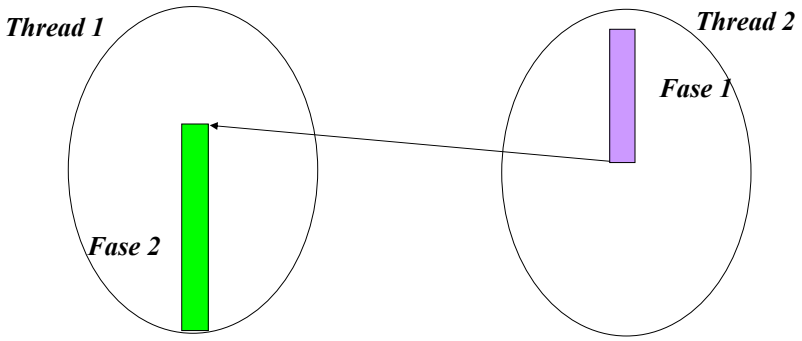
dove:

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

e` la *signal* 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: sincronizzazione



- **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: 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 m; /* semaforo per la mutua esclusione
           nell'accesso alla sezione critica */
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);
        sem_wait(&m);
        if (V<MAX)
                V++;
        else
        {       k=0;
                printf("T1: %d\n",++F);
        }
        sem_post(&m);
        sem_post(&s2);
    }
  pthread_exit (0);
}
```



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

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



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

  sem_init (&m, 0, 1);
  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);

}
```


Esecuzione:

```
[aciampolini@ccib48 threads]$ sem1
T2: 1
T3: 2
T1: 3
[aciampolini@ccib48 threads]$
```

LinuxThreads: variabili condizione

- Lo standard POSIX 1003.1c (libreria <**`pthread.h`**>) implementa le **variabili 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).
  - **operazioni fondamentali:**
    - **inizializzazione:** **`pthread_cond_init`**
    - **sospensione:** **`pthread_cond_wait`**
    - **risveglio:** **`pthread_cond_signal`**
  - **Per operare sulle variabili condizione:**
    `pthread_cond_t`: è il tipo predefinito per le variabili condizione.

Variabili Condizione: inizializzazione

- **L'inizializzazione di una `condition` 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.**

NB: linux non implementa gli attributi !

- in alternativa, una variabile condizione può essere inizializzata staticamente con la costante:
  `PTHREAD_COND_INIZIALIZER`
- **esempio**: `pthread_cond_t C= PTHREAD_COND_INIZIALIZER;`

Variabili condizione: wait

- Un thread puo` sospendersi su una variabile condizione, se la condizione non e` verificata:
  - ad esempio:

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
int bufferpieno=0;
...
if (bufferpieno) <sospensione sulla cond. C>
```

  - La verifica della condizione e` una sezione critica!
  - Necessita` di garantire la mutua esclusione:

    e` necessario associare ad ogni variabile condizione un mutex :

```
pthread_cond_t C= PTHREAD_COND_INIZIALIZER;
pthread_mutex_t M=PTHREAD_MUTEX_INITIALIZER;
int bufferpieno=0;
...
pthread_mutex_lock(&M);
if (bufferpieno) <sospensione sulla cond. C>
pthread_mutex_unlock(&M);
```


Variabili condizione: 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

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.

Variabili condizione: 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.**

N.B. non e` prevista una funzione per verificare lo stato della coda associata a una condizione.

Esempio: produttore e consumatore

Si vuole risolvere il classico problema del produttore e consumatore.

Progetto della *risorsa* (`prodcons`):

- **buffer circolare di interi, di dimensione data (ad esempio, 16) il cui stato e`dato da:**
  - **numero degli elementi contenuti: `cont`**
  - **puntatore alla prima posizione libera: `writepos`**
  - **puntatore al primo elemento occupato : `readpos`**
- **il buffer e`una risorsa da accedere in modo mutuamente esclusivo:**
  - **predispongo un mutex per il controllo della mutua esclusione nell'accesso al buffer: `lock`**
- **i thread produttori e consumatori necessitano di sincronizzazione in caso di :**
  - **buffer pieno: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notfull`)**
  - **buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer e` pieno (`notempty`)**

Incapsulo il tutto all'interno di un tipo struct associato al buffer: `prodcons`

Produttori & Consumatori: tipo di dato associato al buffer

```
typedef struct
{
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```


Produttore e consumatore

Operazioni sulla *risorsa* `prodcons`:

- **`Init`: inizializzazione del buffer.**
- **`Inserisci`: operazione eseguita dai produttori per l'inserimento di un nuovo elemento.**
- **`Estrai`: operazione eseguita dai consumatori per l'estrazione di un elemento dal buffer.**

Esempio: produttore e consumatore

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

#define BUFFER_SIZE 16


typedef struct
{
  int buffer[BUFFER_SIZE];
  pthread_mutex_t lock;
  int readpos, writepos;
  int cont;
  pthread_cond_t notempty;
  pthread_cond_t notfull;
}prodcons;
```


Esempio:Operazioni sul buffer

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

Operazioni sul buffer

```
/* Inserimento: */
void inserisci (prodcons *b, int data)
{  pthread_mutex_lock (&b->lock);
  /* 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);
}
```


Operazioni sul buffer

```
/*ESTRAZIONE:  */
int estrai (prodcons *b)
{  int data;
  pthread_mutex_lock (&b->lock);
  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)*/
  pthread_cond_signal (&b->notfull);
  pthread_mutex_unlock (&b->lock);
  return data;
}
```

Produttore/consumatore: programma di test

```
/* Programma di test: 2 thread
   - un thread inserisce sequenzialmente max interi,
   - l'altro thread li estrae sequenzialmente per stamparli */

#define OVER (-1)
#define max 20

prodcons buffer;

void *producer (void *data)
{ int n;
  printf("sono il thread produttore\n\n");
  for (n = 0; n < max; n++)
    { printf ("Thread produttore %d --->\n", n);
      inserisci (&buffer, n);
    }
  inserisci (&buffer, OVER);
  return NULL;
}
```



```
void *consumer (void *data)
{  int d;
  printf("sono il thread consumatore \n\n");

  while (1)
    {
      d = estrai (&buffer);
      if (d == OVER)
        break;
      printf("Thread consumatore: --> %d\n",  d);
    }
  return NULL;
}
```

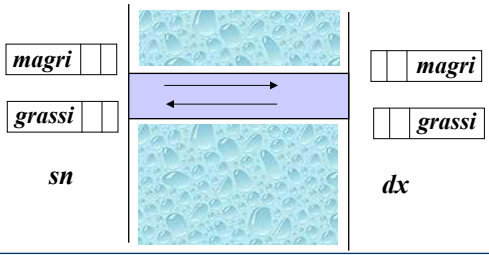
```
main ()
{
  pthread_t th_a, th_b;
  void *retval;

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


Esempio: Ponte con utenti grassi e magri

Si consideri un ponte pedonale che collega le due rive di un fiume.

- **Al ponte possono accedere due tipi di utenti: utenti magri e utenti grassi.**
- **Il ponte ha una capacita` massima MAX che esprime il numero massimo di persone che possono transitare contemporaneamente su di esso.**
- **Il ponte e` talmente stretto che il transito di un grasso in una particolare direzione d impedisce l'accesso al ponte di altri utenti (grassi e magri) in direzione opposta a d.**

Realizzare una politica di sincronizzazione delle entrate e delle uscite dal ponte che tenga conto delle specifiche date e che favorisca gli utenti magri rispetto a quelli magri nell'accesso al ponte.



➔ *due tipi di thread:*

- ✓ *grassi*
- ✓ *magri*

➔ *una coda per ogni tipo di thread e per ogni direzione*

Progetto della *risorsa* `ponte`:

- lo stato del ponte e`definito da:
  - numero magri e di grassi sul ponte (per ogni direzione)
- lo stato e` modificabile dalle operazioni di:
  - accesso: ingresso di un thread nel ponte
  - rilascio: uscita di un thread dal ponte
- il ponte e`una risorsa da acquisire e rilasciare in modo mutuamente esclusivo:
  - predispongo un mutex per il controllo della mutua esclusione nell'esecuzione delle operazioni di accesso e di rilascio: `lock`
- i thread grassi e magri si possono sospendere se le condizioni necessarie per l'accesso non sono verificate :
  - una coda per ogni tipo di thread (grasso o magro)e per ogni direzione
- per ispezionare lo stato delle code introduciamo:
  - un contatore dei thread sospesi per ogni tipo di thread (grasso o magro)e per ogni direzione

➔Incapsulo il tutto all'interno di un tipo struct associato al buffer: `ponte`

Grassi & Magri: tipo di dato associato al ponte

```
typedef struct
{
  int nmagri[2]; /* numero magri sul ponte (per ogni dir.)*/
  int ngrassi[2];/* numero grassi sul ponte (per ogni dir.)*/
  pthread_mutex_t lock;/*lock associato alla risorsa "ponte"*/
  pthread_cond_t codamagri[2]; /* var. cond. sosp. magri */
  pthread_cond_t codagrassi[2]; /* var. cond. sosp. grassi */
  int sospM[2];/* numero di processi magri sospesi*/
  int sospG[2];/* numero di processi grassi sospesi*/
}ponte;
```

Produttore e consumatore

Operazioni sulla *risorsa* `ponte`:

- **`init`: inizializzazione del ponte.**
- **`accessomagri/accessograssi`: operazione eseguita dai thread (grassi/magri) per l'ingresso nel ponte.**
- **`rilasciomagri/rilasciograssi`: operazione eseguita dai thread (grassi/magri) per l'uscita dal ponte.**

Grassi & Magri: soluzione

```
#include <stdio.h>
#include <pthread.h>
#define MAX 3 /* max capacita ponte */
#define dx 0 /*costanti di direzione*/
#define sn 1

typedef struct
{
  int nmagri[2]; /* numero magri sul ponte (per ogni dir.)*/
  int ngrassi[2];/* numero grassi sul ponte (per ogni dir.)*/
  pthread_mutex_t lock;/*lock associato al"ponte" */
  pthread_cond_t codamagri[2]; /* var. cond. sosp. magri */
  pthread_cond_t codagrassi[2]; /* var. cond. sosp. grassi */
  int sospM[2];/* numero di processi magri sospesi*/
  int sospG[2];/* numero di processi grassi sospesi*/
}ponte;
```

Grassi & Magri: soluzione

```
/* Inizializzazione del ponte */
void init (ponte *p)
{
  pthread_mutex_init (&p->lock, NULL);
  pthread_cond_init (&p->codamagri[dx], NULL);
  pthread_cond_init (&p->codamagri[sn], NULL);
  pthread_cond_init (&p->codagrassi[dx], NULL);
  pthread_cond_init (&p->codagrassi[sn], NULL);
  p->nmagri[dx]=0;
  p->nmagri[sn]=0;
  p->ngrassi[dx]=0;
  p->ngrassi[sn]=0;
  p->sospM[dx] = 0;
  p->sospM[sn] = 0;
  p->sospG[dx] = 0;
  p->sospG[sn] = 0;
  return;
}
```



```
/*operazioni di utilita`: */
int sulponte(ponte p); /* calcola il num. di persone sul ponte */
int altra_dir(int d); /* calcola la direzione opposta a d */

/* Accesso al ponte di un magro in direzione d: */
void accessomagri (ponte *p, int d)
{  pthread_mutex_lock (&p->lock);
  /* controlla le codizioni di accesso:*/
  while ( (sulponte(*p)==MAX) || /* vincolo di capacita` */
          (p->ngrassi[altra_dir(d)]>0) ) /*ci sono grassi in
                                           direzione opposta */
   {      p->sospM[d]++;
          pthread_cond_wait (&p->codamagri[d], &p->lock);
          p->sospM[d]--;
  }
  /* entrata: aggiorna lo stato del ponte */
  p->nmagri[d]++;
  /* risveglia eventuali thread "omologhi" nella stessa dir. */
  pthread_cond_signal (&p->codamagri[d]);
  pthread_mutex_unlock (&p->lock);
}
```

```
/*accessograssi: Accesso al ponte di un grasso in dir.d:  */

void accessograssi (ponte *p, int d)
{  pthread_mutex_lock (&p->lock);
  /* controlla le codizioni di accesso:*/
  while ( (sulponte(*p)==MAX) ||
          (p->ngrassi[altra_dir(d)]>0) ||
          (p->sospM[altra_dir(d)]>0)) /*priorita` ai magri: ci
                          sono magri in attesa in dir opposta */
  {       p->sospG[d]++;
          pthread_cond_wait (&p->codagrassi[d], &p->lock);
          p->sospG[d]--;
  }
  /* entrata: aggiorna lo stato del ponte */
  p->ngrassi[d]++;
  /* risveglia eventuali thread "omologhi" nella stessa dir: */
  pthread_cond_signal (&p->codagrassi[d]);
  pthread_mutex_unlock (&p->lock);
}
```



```
  /* Rilascio del ponte di un magro in direzione d: */

  void rilasciomagri (ponte *p, int d)
  {
    pthread_mutex_lock (&p->lock);

    /* uscita: aggiorna lo stato del ponte */
    p->nmagri[d]--;
    /* risveglio in ordine di priorita` */
    pthread_cond_signal (&p->codamagri[altra_dir(d)]);
    pthread_cond_signal (&p->codamagri[d]);
    pthread_cond_signal (&p->codagrassi[altra_dir(d)]);
    pthread_cond_signal (&p->codagrassi[d]);
    printf("USCITA: magro in direzione %d\n", d);
    pthread_mutex_unlock (&p->lock);
  }
```

```
/* Rilascio del ponte di un grasso in direzione d: */

void rilasciograssi (ponte *p, int d)
{
  pthread_mutex_lock (&p->lock);

  /* uscita: aggiorna lo stato del ponte */
  p->ngrassi[d]--;
  /* risveglio in ordine di priorita` */
  pthread_cond_signal (&p->codamagri[altra_dir(d)]);
  pthread_cond_signal (&p->codamagri[d]);
  pthread_cond_signal (&p->codagrassi[altra_dir(d)]);
  pthread_cond_signal (&p->codagrassi[d]);
  printf("USCITA: grasso in direzione %d\n", d);
  pthread_mutex_unlock (&p->lock);
}
```



```
/* Programma di test: genero un numero arbitrario di thread magri
   e grassi nelle due direzioni */
#define MAXT 20 /* num. max di thread per tipo e per direzione */


ponte p;

void *magro (void *arg) /*codice del thread "magro" */
{ int d;
  printf("sono un thread magro in direzione %s\n\n",(char *)arg);
  d=atoi((char *)arg); /*assegno la direzione */
  accessomagri (&p, d);
  /* ATTRAVERSAMENTO: */
   printf("Magro in dir %d: sto attraversando..\n", d);
  sleep(1);
  rilasciomagri(&p,d);
  return NULL;
}
```

```
void *grasso (void *arg) /*codice del thread "grasso" */
{ int d;
  printf("sono un thread grasso in direzione %s\n", (char *)arg);
  d=atoi((char *)arg); /*assegno la direzione */
  accessograssi (&p, d);
  sleep(1);
  printf("Grasso in dir %d: sto attraversando\n", d);
  rilasciograssi(&p,d);
  return NULL;
}


main ()
{
  pthread_t th_M[2][MAXT], th_G[2][MAXT];
  int NMD, NMS, NGD, NGS, i;
  void *retval;

  init (&p);
```



```
/* Creazione threads: */
  printf("\nquanti magri in direzione dx? ");
  scanf("%d", &NMD);
  printf("\nquanti magri in direzione sn? ");
  scanf("%d", &NMS);
  printf("\nquanti grassi in direzione dx? ");
  scanf("%d", &NGD);
  printf("\nquanti grassi in direzione sn? ");
  scanf("%d", &NGS);
  /*CREAZIONE MAGRI IN DIREZIONE DX */
  for (i=0; i<NMD; i++)
          pthread_create (&th_M[dx][i], NULL, magro, "0");
   /*CREAZIONE MAGRI IN DIREZIONE SN */
  for (i=0; i<NMS; i++)
          pthread_create (&th_M[sn][i], NULL, magro, "1");
  /*CREAZIONE GRASSI IN DIREZIONE DX */
  for (i=0; i<NGD; i++)
          pthread_create (&th_G[dx][i], NULL, grasso, "0");
   /*CREAZIONE GRASSI IN DIREZIONE SN */
  for (i=0; i<NGS; i++)
          pthread_create (&th_G[sn][i], NULL, grasso, "1");
```

```
   /* Attesa teminazione threads creati: */

    /*ATTESA MAGRI IN DIREZIONE DX */
    for (i=0; i<NMD; i++)
            pthread_join(th_M[dx][i], &retval);

     /*CREAZIONE MAGRI IN DIREZIONE SN */
    for (i=0; i<NMS; i++)
            pthread_join(th_M[sn][i], &retval);

    /*CREAZIONE GRASSI IN DIREZIONE DX */
    for (i=0; i<NGD; i++)
            pthread_join(th_G[dx][i], &retval);

     /*CREAZIONE GRASSI IN DIREZIONE SN */
    for (i=0; i<NGS; i++)
            pthread_join(th_G[sn][i], &retval);

    return 0;
  }
```



```
 /* definizione funzioni utilita`:*/
int sulponte(ponte p) /* calcola il num.di pers.sul ponte */
{
        return p.nmagri[dx]+p.nmagri[sn]+p.ngrassi[dx]+
  p.ngrassi[sn];

}

int altra_dir(int d) /* fornisce la dir. opposta a  d */
{
        if (d==sn) return dx;
        else return sn;
}
```

Segnali & Thread

- Per spedire un segnale a un **`thread`**:

```
int pthread_kill(pthread_t thread, int signo);
```

- Per sospendere un thread in attesa di un segnale:

```
int sigwait(const sigset_t *set, int *sig);
```

- Gestione del segnale associata al task:
  - **System call `signal`**
  - **Funzione `sigset` (POSIX)**