

I thread nel sistema operativo LINUX: *Linuxthreads*

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`)

3

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'

2

Rappresentazione dei threads

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

```
pthread_t tid;
```

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

4

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 raccoglierà 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**: può essere usato per specificare eventuali attributi da associare al thread (di solito: NULL):
 - ad esempio parametri di scheduling: *priority* 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)

5

LinuxThreads

- Un thread può terminare chiamando:

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

- Dove:

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

7

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).

6

LinuxThreads

- Un thread può 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 verrà memorizzato il valore di ritorno del thread (v. *pthread_exit*)

8

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);
}
```

9

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]$
```

11

```
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;
}
```

10

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.

12

LinuxThreads: MUTEX

- Lo standard POSIX 1003.1c (libreria `<pthread.h>`) definisce i **semafori binari** (o lock, mutex, etc.)
 - sono semafori il cui valore può 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;`

13

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` è 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.

15

MUTEX: inizializzazione

- L'inizializzazione di un mutex si può 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 può inizializzare il mutex a default con la macro:
- ```
PTHREAD_MUTEX_INITIALIZER
```
- esempio: `pthread_mutex_t mux= PTHREAD_MUTEX_INITIALIZER;`

14

## Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define MAX 10
pthread_mutex_t M; /* def.mutex condiviso tra threads */
int DATA=0; /* variabile condivisa */
int accessil=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 */
 accessil++;
 DATA++;
 k=(DATA>=MAX?0:1);
 printf("accessi di T1: %d\n", accessil);
 pthread_mutex_unlock(&M); /*epilogo */
 }
 pthread_exit (0);
}
```

16

## 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);
}
```

17

## Test

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

19

## 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);
}
```

18

## 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;
```

20

## 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.

21

## 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.

23

## Operazioni sui semafori: sem\_wait

– **wait** su un semaforo

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

dove:

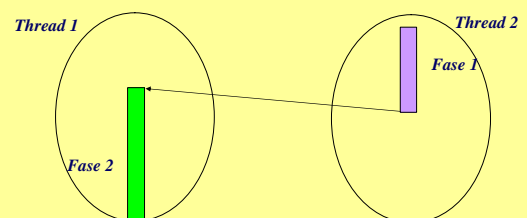
- **sem**: individua il semaforo sul quale operare.

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.

22

## Esempio: sincronizzazione



➤ Imposizione di un vincolo temporale: la FASE2 nel thread 1 va eseguita dopo la FASE1 nel thread2.

24

### 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);
}
```

25

```
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 2\n");
 exit (1);
 }

 pthread_join (th1, &ret);
 pthread_join (th2, &ret);
}
```

27

```
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);
}
```

26

### 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]$
```

28

## 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;
```

29

```
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 (V=%d)\n",++F, V);
 }
 sem_post(&m);
 sem_post(&s3);
 }
 pthread_exit (0);
}
```

31

```
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 (V=%d)\n",++F, V);
 }
 sem_post(&m);
 sem_post(&s2);
 }
 pthread_exit (0);
}
```

30

```
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 (V=%d)\n",++F, V);
 }
 sem_post(&m);
 sem_post(&s1);
 }
 pthread_exit (0);
}
```

32



```

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);
}
}

```

33

## Esecuzione:

```

[aciampolini@ccib48 threads]$ sem1
T2: 1 (V=10)
T3: 2 (V=10)
T1: 3 (V=10)
[aciampolini@ccib48 threads]$

```

35

```

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

}

```

34

## 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 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 è 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.

36

## Variabili Condizione: inizializzazione

- L'inizializzazione di una `condition` si può 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_INITIALIZER`
- esempio: `pthread_cond_t C= PTHREAD_COND_INITIALIZER;`

37

## 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 rioccupa il mutex automaticamente.

39

## Variabili condizione: wait

- Un thread può sospendersi su una variabile condizione, se la condizione non è verificata:

– ad esempio:

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

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

è necessario associare ad ogni variabile condizione un mutex :

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

38

## Variabili condizione: signal

- Il risveglio di un thread sospeso su una variabile condizione può 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.
- realizzazione "signal\_and\_continue"

**N.B.** non è prevista una funzione ("..queue") per verificare lo stato della coda associata a una condizione.

40

### Pthread Condition & Monitor

- La *condition* permette di implementare politiche di sincronizzazione mediante funzioni/procedure "entry", realizzando meccanismi di accesso alle risorse equivalenti a quelli forniti dal concetto di monitor [Hoare'74]; differenze:
    - i dati "interni" al monitor sono potenzialmente accessibili direttamente da tutti i processi;
    - la mutua esclusione delle funzioni/procedure entry deve essere garantita esplicitamente dal programmatore mediante lock/unlock su un mutex associato al "monitor".
- necessita` di autodisciplina da parte del programmatore !

41

### Esempio: uso di una risorsa a capacita` limitata

- Introduciamo la *condition* **PIENO**, sulla quale sospendere i thread che vogliono accedere alla risorsa nel caso di capacita` esaurita.
- Sia **M** il mutex associato alla condition **PIENO**.
- Introduciamo inoltre la variabile intera non negativa **N\_in** per rappresentare lo stato della risorsa, cioe` il numero di thread che stanno usando la risorsa.

```
#define MAX 100
/*variabili globali: */
int N_in=0; /* numero thread che usano la risorsa*/
pthread_cond_t PIENO;
pthread_mutex_t M; /* Mutex */
```

43

### Uso di risorsa condivisa a capacita` limitata

- Per esemplificare l'uso della *condition*, si consideri il caso di una risorsa che puo` essere usata, al massimo, da MAX thread contemporaneamente.
- Realizziamo una politica di controllo degli accessi mediante variabili condizione.

42

### Struttura dei thread

#### Tre fasi:

- 1. entrata:** viene controllato lo stato della risorsa: in caso di risorsa "piena" il thread si sospende sulla condition **PIENO**, altrimenti il numero dei thread **N\_in** viene incrementato. La fase di entrata e` una sezione critica (procedura entry).
- 2. uso della risorsa**
- 3. uscita:** terminato l'uso della risorsa, il thread deve eseguire la fase di uscita, nella quale viene decrementato il valore di **N\_in** ed eventualmente risvegliato un processo sospeso sulla condition **PIENO**. Anche la fase di uscita e` una sezione critica (procedura entry).

44

## Codice thread

```
#define MAX 100
/* "monitor": contiene le variabili interne al monitor */
typedef struct{
 int N_in; /* valore iniziale 0*/
 pthread_cond_t PIENO;
 pthread_mutex_t M;
}mon_risorse;

mon_risorse R;

void entry_Entrata(mon_risorse *m) /* Fase di Entrata: */
{
 pthread_mutex_lock (&m->M);
 /* controlla la condizione di sincronizzazione:*/
 if (m->N_in==MAX) pthread_cond_wait (&m->PIENO, &m->M);
 /* aggiorna lo stato della risorsa */
 m->N_in++;
 pthread_mutex_unlock (&m->M);
 /* fine entrata*/
}

<uso della risorsa>
```

45

## 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 è dato da:
  - numero degli elementi contenuti: cont
  - puntatore alla prima posizione libera: writepos
  - puntatore al primo elemento occupato: readpos
- il buffer è 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 è pieno (notfull)
  - buffer vuoto: definisco una condition per la sospensione dei produttori se il buffer è vuoto (notempty)

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

47

```
void entry_USCITA(mon_risorse *m) /*Fase di Uscita:*/
{
 pthread_mutex_lock (&m->M);
 /* aggiorna lo stato della risorsa */
 m->N_in--;
 pthread_cond_signal (&m->PIENO);
 pthread_mutex_unlock (&m->M);
}

void *codice_generico_thread()
{
 while(...)
 {
 entry_ENTRATA(&R);
 <uso risorsa>
 entry_USCITA(&R);
 }
 return NULL;
}

main(){...<inizializzazioni>; <creazione N thread>..}
```

46

## 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;
```

48

## Produttore e consumatore

Operazioni sulla *risorsa* `prodcons`:

- Init: inizializzazione del buffer.
- Inserisci: operazione eseguita da ogni produttore per l'inserimento di un nuovo elemento.
- Estrai: operazione eseguita da ogni consumatore per l'estrazione di un elemento dal buffer.

49

## 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;
}
```

51

## 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;
```

50

## 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);
}
```

52

### 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;
}
```

53

```
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;
}
```

55

### 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;
}
```

54

```
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;
}
```

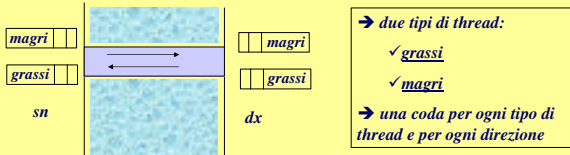
56

### 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 capacità massima MAX che esprime il numero massimo di persone che possono transitare contemporaneamente su di esso.
- Il ponte è 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 grassi nell'accesso al ponte.



57

### 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;
```

59

### Progetto della *risorsa* ponte:

- lo stato del ponte è definito da:
  - numero magri e di grassi sul ponte (per ogni direzione)
- lo stato è modificabile dalle operazioni di:
  - accesso: ingresso di un thread nel ponte
  - rilascio: uscita di un thread dal ponte
- il ponte è 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 del un tipo struct **ponte**

58

### 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.

60

### 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;
```

61

```
/*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);
}
```

63

### 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;
}
```

62

```
/*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->nmagri[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);
}
```

64



```

/* 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 priorit  */
 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);
}

```

65

```

/* 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 direzi
 */

ponte p;

void *magro (void *arg) /*codice del thread "magro" */
{
 int d;
 printf("sono un thread magro in dir. %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;
}

```

67

```

/* 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 priorit  */
 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);
}

```

66

```

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);
}

```

68

```

/* 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");

```

69

```

/* 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;
}

```

71

```

/* Attesa teminazione threads creati: */

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

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

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

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

return 0;
}

```

70

## Esercizio 1

Si consideri un castello di interesse storico.

L'accesso al castello e' consentito a **due tipi** di visitatori: **adulti** o **bambini**.

La visita al castello non puo' avvenire in modo libero, ma deve essere sempre **guidata** dal proprietario del castello. (Essendo il proprietario **unico**, e' implicito che in ogni istante ci puo' essere, al piu', una ed una sola visita in atto.)

La visita assume caratteristiche diverse a seconda che l'insieme di visitatori da accompagnare sia costituito da adulti oppure da bambini. Per questo motivo, il gruppo di persone che partecipa ad ogni visita e' quindi un insieme **omogeneo** di visitatori (cioe': o tutti adulti, oppure tutti bambini).

Per ottimizzare l'utilizzo del castello ogni visita puo' iniziare soltanto quando il numero P dei partecipanti ha raggiunto un valore prestabilito **P<sub>MAX</sub>**.

Quando il gruppo dei partecipanti e' completo, la visita puo' avere inizio attraverso l'attivazione di un opportuno processo proprietario che rappresenta la guida di ogni visita.

72

## ..continua

Al termine della visita, il processo proprietario provvede a far uscire i partecipanti, e successivamente si pone in attesa di un nuovo gruppo di visitatori da guidare nella visita successiva.

Alla fine di ogni visita, il tipo dei partecipanti della visita successiva viene stabilito in base al numero e al tipo di visitatori in attesa; in particolare, detti **AS** ed **BS** rispettivamente il numero di adulti in attesa ed il numero di bambini in attesa:

- se **AS>BS** allora la prossima visita sarà per **adulti**;
- se **BS>AS** allora la prossima visita sarà per **bambini**;
- se **BS=AS** venga data la precedenza ai bambini: la prossima visita sarà per bambini.

Definire una politica di gestione del castello che tenga conto dei vincoli indicati, e la si realizzi utilizzando la libreria **pthread**.