Sistemi Operativi: IPC

Amos Brocco, Ricercatore, DTI / ISIN

Basato su:
[STA09] “Operating Systems: Internals and Design Principles”, 6/E, William Stallings, Prentice Hall, 2009
[TAN01] “Modern Operating Systems”, 2/E, Andrew S. Tanenbaum, Prentice Hall, 2001
[TAN09] “Modern Operating Systems”, 3/E, Andrew S. Tanenbaum, Prentice Hall, 2009

Risorse condivise: esempio di un conto bancario

int saldo_conto = 100;

Thread 1

```
int portamonete = 0;
if (saldo_conto >= 100) {
    /* Preleva dal conto */
    saldo_conto = saldo_conto – 100;
    portamonete = 100;
} else {
    printf(“Non posso prelevare!\n”);
}
if (portamonete == 100) {
     /* Fai gli acquisti */
}
/* Versamento di 50 */
saldo_conto = saldo_conto + 50;
```

Thread 2

```
/* Versamento di 50 */
saldo_conto = saldo_conto + 50;

/* ... Altre attività ... */

if (saldo_conto >= 100) {
    /* Preleva dal conto */
    saldo_conto = saldo_conto – 100;
    /* Paga le fatture */
    paga_fatture(100);
} else {
    printf(“Non posso prelevare!\n”);
}
```

Race conditions

- In un programma multithread con risorse condivise, quando la correttezza del risultato di un programma dipende dall'ordine di esecuzione dei singoli thread si ha una situazione di **race condition**
  - Il problema deriva dall'accesso concorrente a una risorsa condivisa (es. variabile) da parte di più thread
    - Per risolvere il problema bisogna innanzitutto determinare le parti del programma “a rischio”...

Sezioni critiche

- Una **sezione** o **regione critica** relativa è una parte del codice di un programma in cui si accede a una risorsa condivisa

Thread 1

```
int portamonete = 0;
if (saldo_conto >= 100) {
    /* Preleva dal conto */
    saldo_conto = saldo_conto – 100;
    portamonete = 100;
} else {
    printf("Non posso prelevare!\n");
}
if (portamonete == 100) {
     /* Fai gli acquisti */
}
/* Versamento di 50 */
saldo_conto = saldo_conto + 50;
```

int saldo_conto = 100;

Thread 2

```
/* Versamento di 50 */
saldo_conto = saldo_conto + 50;

/* ... Altre attività ... */

if (saldo_conto >= 100) {
    /* Preleva dal conto */
    saldo_conto = saldo_conto – 100;
    /* Paga le fatture */
    paga_fatture(100);
} else {
    printf("Non posso prelevare!\n");
}
```

Condizioni per evitare una race condition

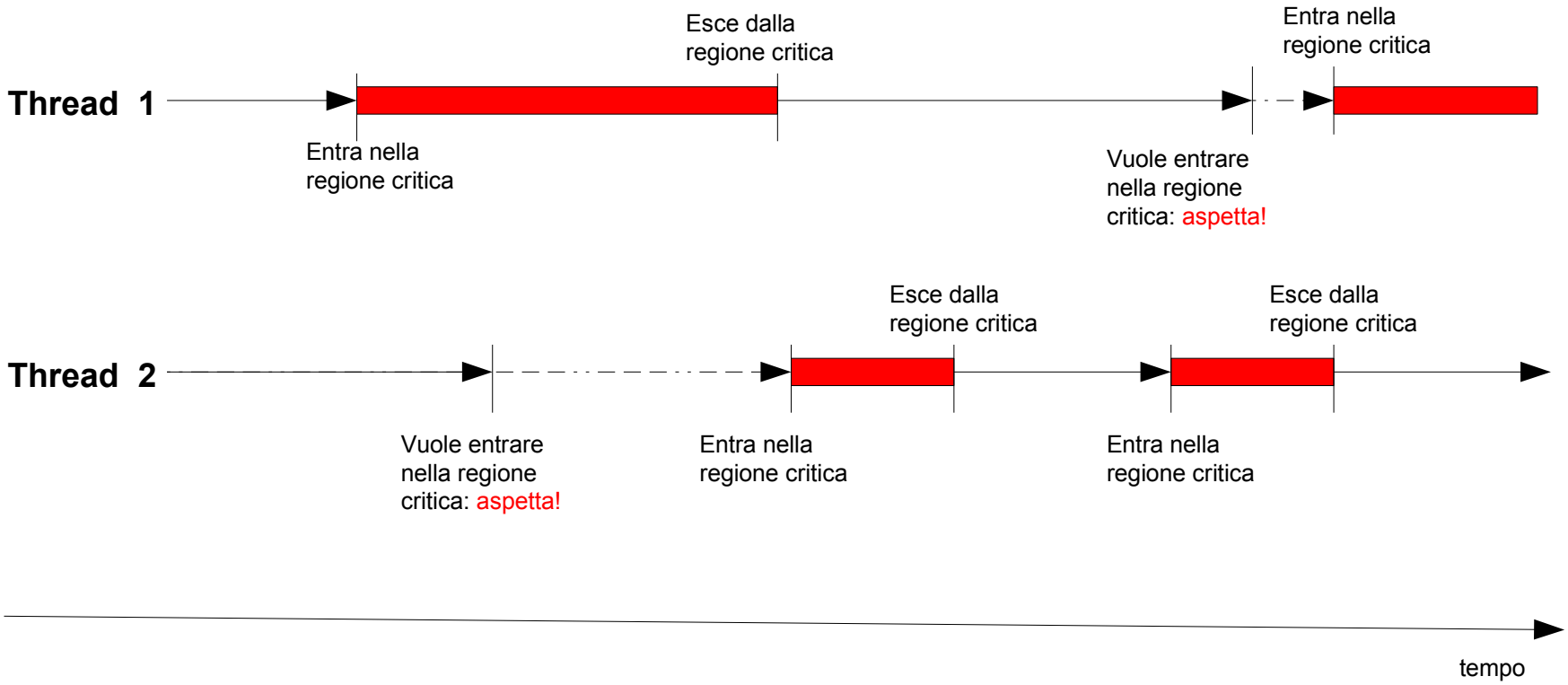
(1) Due thread non possono essere simultaneamente all'interno delle regioni critiche relative alla stessa risorsa

(2) Non si devono fare ipotesi sulla velocità di esecuzione e sul numero di thread realmente eseguiti in parallelo

(3) Fuori dalle regioni critiche nessun thread può bloccare un altro thread

(4) Nessun thread deve aspettare indefinitamente per entrare in una regione critica

Esempio


Thread 1
Entra nella regione critica
Esce dalla regione critica
Vuole entrare nella regione critica: aspetta!
Entra nella regione critica
Thread 2
Vuole entrare nella regione critica: aspetta!
Entra nella regione critica
Esce dalla regione critica
Entra nella regione critica
Esce dalla regione critica
tempo

Atomicità

- Alcuni dei meccanismi che andremo ad analizzare dipendono dal concetto di **atomicità** e da istruzioni dette atomiche:
  - Un'istruzione atomica non può essere interrotta
  - L'esecuzione di un'istruzione atomica è unica: due thread o due processi non possono ottenere l'accesso a una sezione critica contemporaneamente

Mutua esclusione: qualche soluzione e non-soluzione

- **Esecuzione sequenziale**
  - Se i thread sono eseguiti sequenzialmente (uno alla volta, dall'inizio alla fine) anziché in (pseudo-) parallelismo, la mutua esclusione è garantita.
- **Variabili di lock**
  - Definiamo una variabile condivisa 'lock', inizialmente uguale a 0. Quando un processo vuole entrare, guarda il valore di lock: se la variabile è 0, il processo la cambia in 1 e entra nella regione critica. Se la variabile è 1, il processo aspetta...
    - ***Non funziona correttamente... perché?***
- ***Alternanza stretta***

Consuma CPU! Non ottimale!

Thread 0

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

Thread 1

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

A. Tanenbaum, Modern Operating Systems, 2nd ed

Mutua esclusione: qualche soluzione e non-soluzione

- **Disabilitare gli interrupt**
  - Sistemi con una sola CPU/ un solo core:
    - Senza interrupt il context switch non è possibile
  - Sistemi multiprocessore o multicore
    - Non funziona (a meno di disabilitare gli interrupt su tutte le CPU)

- ***Soluzione di Peterson***

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* number of processes */

int turn;                           /* whose turn is it? */
int interested[N];                  /* all values initially 0 (FALSE) */
```

Da A. Tanenbaum, Modern Operating Systems, 2nd ed

```
void enter_region(int process);     /* process is 0 or 1 */
{
    int other;                      /* number of the other process */

    other = 1 – process;            /* the opposite of process */
    interested[process] = TRUE;     /* show that you are interested */
    turn = process;                 /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)      /* process: who is leaving */
{
    interested[process] = FALSE;    /* indicate departure from critical region */
}
```

Mutua esclusione... attesa attiva: qualche soluzione e non-soluzione

- **Scambiare il valore di due registri (XCHG)**
  - Lo scambio avviene in modo atomico!

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Da A. Tanenbaum, Modern Operating Systems, 2nd ed

Attesa attiva (busy waiting)

- L'alternanza stretta, la soluzione di Peterson, e i meccanismi che utilizzano le istruzioni TSL o XCHG utilizzano l'**attesa attiva** (busy waiting) per aspettare di entrare nella sezione critica, ovvero un ciclo in cui il processo o il thread riprovano continuamente ad accedere (*spinlock*)

- Problemi del busy waiting:
  - Utilizzano la CPU senza far nulla
  - Può causare il problema dell'**inversione di priorità**

Test and Set Lock (TSL)

- Nei processori esiste un'istruzione (TSL Test and Set Lock) che permette di verificare il valore di un'indirizzo di memoria *lock* e eventualmente di cambiarlo garantendo l'atomicità
  - **TSL Registro,Lock**
- Per gestire le regioni critiche, le funzioni di entrata e uscita possono essere implementate come segue:

```
entra:
  TSL Registro,Lock   ; copia lock nel registro, e imposta lock a 1
  CMP Registro,#0     ; test per vedere se lock era uguale a 0
  JNE entra           ; se non era zero, ripeti il testo (cicla)
  RET                 ; altrimenti permetti l'entrata in sez. critica
esci:
  MOVE Lock,#0        ; imposta lock a 0 (libera sezione critica)
  RET                 ; ritorna
```

A. Tanenbaum, Modern Operating Systems, 2nd ed

Attesa attiva (busy waiting)

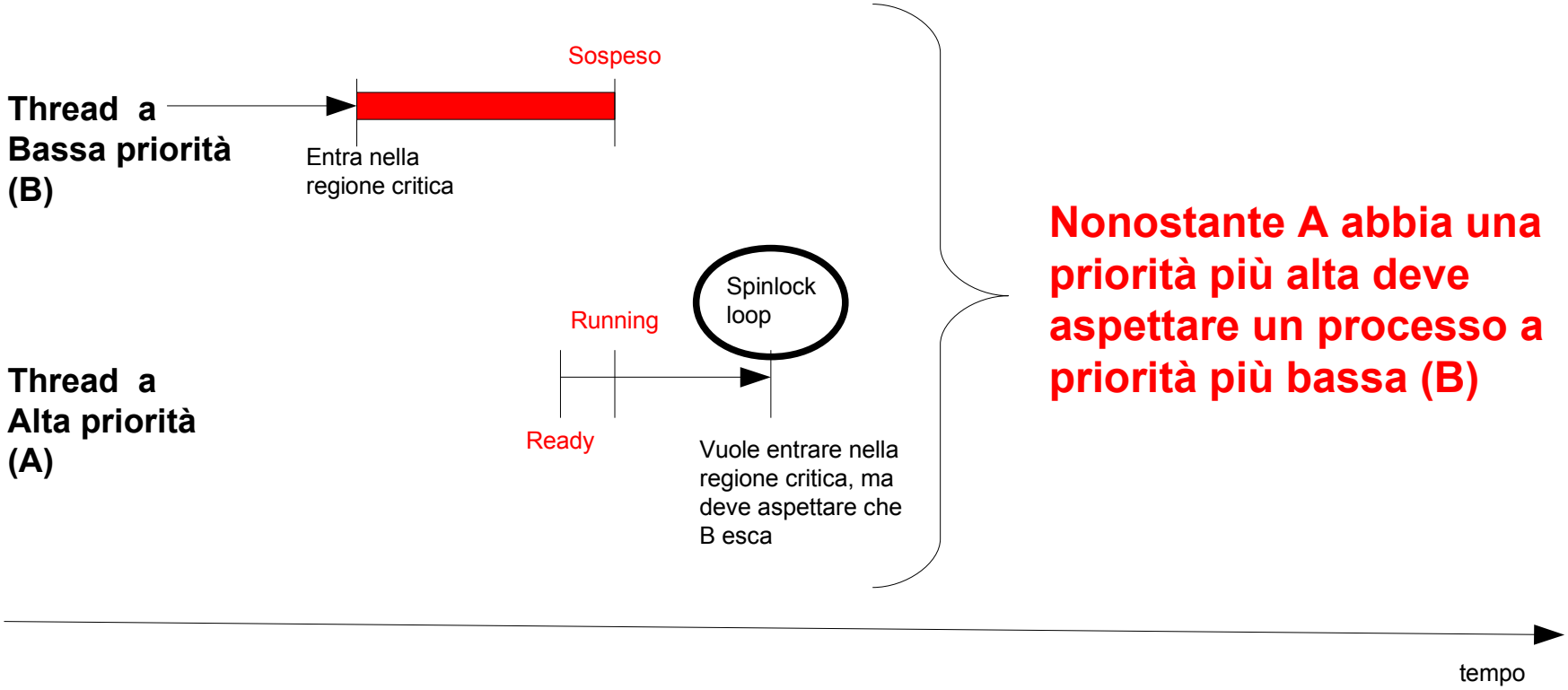
- L'alternanza stretta e il meccanismo che utilizza l'istruzione TSL utilizzano l'**attesa attiva** (busy waiting) per aspettare di entrare nella sezione critica, ovvero un ciclo in cui il processo o il thread riprovano continuamente ad accedere (*spinlock*)

- Problemi del busy waiting:
  - Utilizzano la CPU senza far nulla
  - Può causare il problema dell'**inversione di priorità**

Inversione della priorità

- Supponiamo di avere un processo ad alta priorità A, e un processo a bassa priorità B
- Solo un processo alla volta può essere in esecuzione
- Per scegliere quale processo eseguire viene utilizzata la regola seguente:
  - Quando un processo a alta priorità è pronto per essere eseguito (stato **Ready**), nessun processo con più bassa priorità può essere eseguito

Inversione della priorità



Sincronizzazione

- Per evitare le race conditions i sistemi operativi, con l'aiuto di istruzioni specifiche della CPU, mettono a disposizione dei **meccanismi di sincronizzazione** e mutua esclusione per **regolare l'accesso alle regioni critiche** senza ricorrere alle attese attive:
  - Mutex
  - Semafori
  - Barriere
  - Variabili di condizione

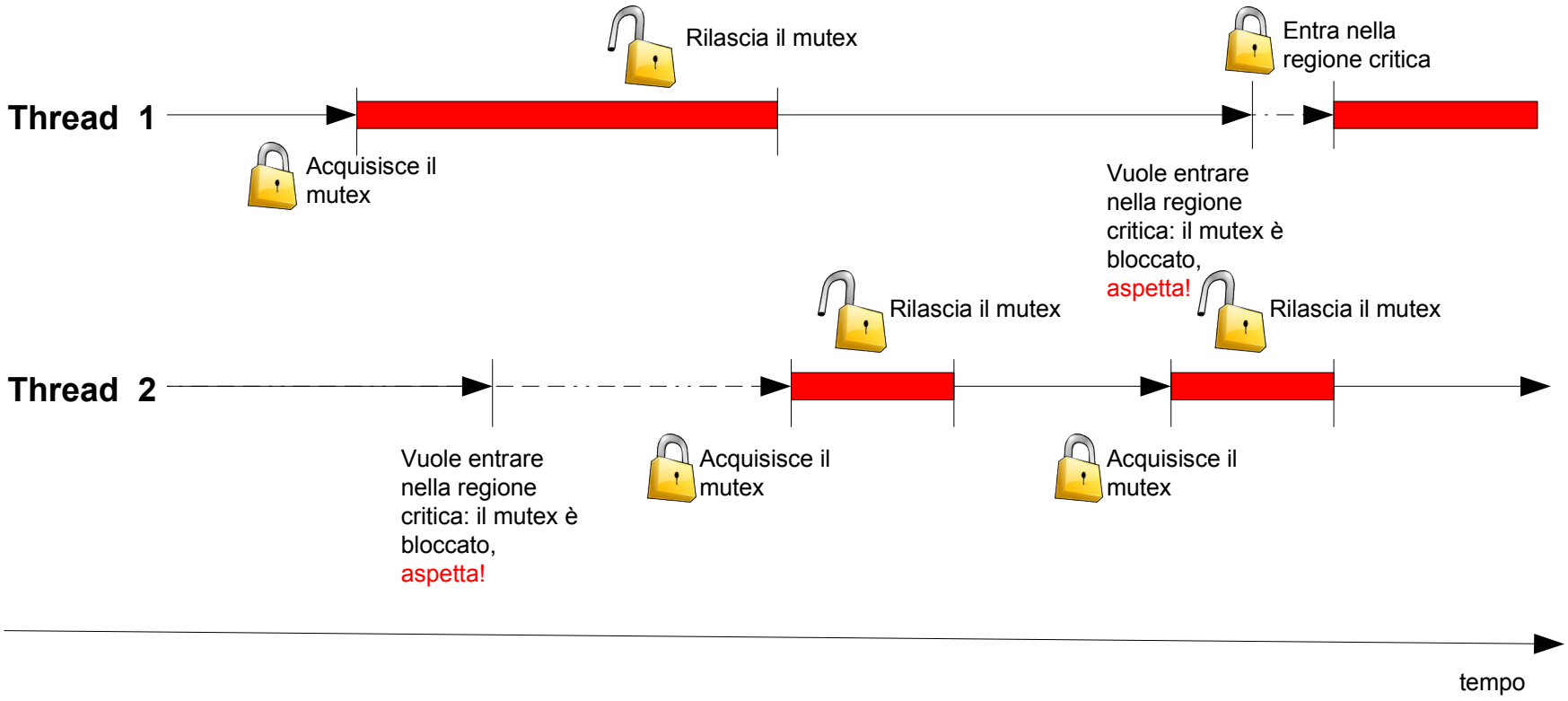
Mutex

- Un mutex è un oggetto binario con due stati:
  - **Bloccato** 
  - **Sbloccato** 
- Il passaggio tra questi stati avviene in modo atomico
- Un mutex è solitamente associato a una sezione critica, e permette di garantire la mutua esclusione

Mutex

- Per entrare nella sezione critica il thread cerca di acquisire il mutex (**lock**)
  - Solo un thread alla volta può essere in possesso del mutex
  - Se il mutex è sbloccato, un thread lo può acquisire
    - Da questo momento, solo il thread in possesso del lock sul mutex potrà rilasciarlo (**unlock**)
  - Se un mutex è bloccato, i thread che cercano di acquisirlo devono aspettare (vanno nello stato *Blocked*)

Esempio


Thread 1
Thread 2
Rilascia il mutex
Entra nella regione critica
Acquisisce il mutex
Vuole entrare nella regione critica: il mutex è bloccato, aspetta!
Rilascia il mutex
Rilascia il mutex
Vuole entrare nella regione critica: il mutex è bloccato, aspetta!
Acquisisce il mutex
Acquisisce il mutex
tempo

Pthread e mutex

- Creare e inizializzare un mutex

```
#include <pthread.h>

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

- **`pthread_mutex_t`** serve a identificare il mutex
- **`pthread_mutexattr_t`** definisce la tipologia del thread.

Tipi di mutex

```
pthread_mutexattr_settype(&attr,
PTHREAD_MUTEX_DEFAULT);
```

RAPIDO

- Definito anche con **PTHREAD_MUTEX_NORMAL**
- Può essere bloccato una sola volta: se il thread che ha bloccato il mutex tenta un secondo bloccaggio ho un deadlock
- Questo tipo di mutex è il più performante

```
pthread_mutexattr_settype(&attr,
PTHREAD_MUTEX_ERRORCHECK);
```

NON RICORSIVO

- Ritorna un valore negativo se un thread che ha bloccato il mutex tenda un secondo bloccaggio

```
pthread_mutexattr_settype(&attr,
PTHREAD_MUTEX_RECURSIVE);
```

RICORSIVO

- Può essere bloccato più volte dallo stesso thread
- Per sbloccarlo bisogna chiamare **pthread_mutex_unlock()** un numero di volte pari a quello delle chiamate **pthread_mutex_lock()** effettuate

Distruggere un mutex

- Per distruggere un mutex

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

Non è possibile distruggere un mutex se c'è un thread bloccato! (ritorna l'errore EBUSY)

Bloccare e sbloccare un mutex

- Per bloccare un mutex

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- Per sbloccare un mutex

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Esempio conto bancario: senza mutex

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

int saldo_conto = 100;

void paga_fatture(int amount)
{
  printf("Pago le fatture: %d\n", amount);
}

void controlla_saldo()
{
  if (saldo_conto < 0) {
    printf("ERRORE! Saldo negativo: %d!\n", saldo_conto);
  }
}
```

Esempio conto bancario: senza mutex

```
void* thread1() {
  int portamonete = 0;
  if (saldo_conto >= 100) {
    /* Preleva dal conto */
    printf("T1: Prelevo 100!\n");
    saldo_conto = saldo_conto - 100;
    portamonete = 100;
    controlla_saldo();
  } else {
    printf("T1: Non posso prelevare!\n");
  }
  if (portamonete == 100) {
     /* Fai gli acquisti */
  }
  /* Versamento di 50 */
  printf("T1: verso 50!\n");
  saldo_conto = saldo_conto + 50;
  printf("T1: Il saldo è %d\n", saldo_conto);
}
```

```
void* thread2()
{
  /* Versamento di 50 */
  printf("T2: verso 50!\n");
  saldo_conto = saldo_conto + 50;
  printf("T2: Il saldo è %d\n", saldo_conto);

  /* ... Altre attività ... */

  if (saldo_conto >= 100) {
    /* Preleva dal conto */
    printf("T2: Prelevo 100!\n");
    saldo_conto = saldo_conto - 100;
    controlla_saldo();
    /* Paga le fatture */
    paga_fatture(100);
  } else {
    printf("T2: Non posso prelevare!\n");
  }
}
```

Esempio conto bancario: senza mutex

```
void main()
{
  pthread_t t1, t2;

  pthread_create(&t1, NULL, &thread1, NULL);
  pthread_create(&t2, NULL, &thread2, NULL);

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

  printf("Il saldo è %d\n", saldo_conto);
}
```

Esempio conto bancario: senza mutex

```
attila@dusty:~$ ./banca
T1: Prelevo 100!
T1: verso 50!
T2: verso 50!
T1: Il saldo è 50
T2: Il saldo è 100
T2: Prelevo 100!
Pago le fatture: 100
Il saldo è 0
```

... a volte non sembrano esserci problemi...

```
attila@dusty:~$ ./banca
T2: verso 50!
T2: Il saldo è 150
T2: Prelevo 100!
Pago le fatture: 100
T1: Prelevo 100!
ERRORE! Saldo negativo: -50!
T1: verso 50!
T1: Il saldo è 0
Il saldo è 0
```

...ma poi...

Esempio conto bancario: con mutex

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

int saldo_conto = 100;
pthread_mutex_t mutex;

void paga_fatture(int amount)
{
  printf("Pago le fatture: %d\n", amount);
}

void controlla_saldo()
{
  if (saldo_conto < 0) {
    printf("ERRORE! Saldo negativo: %d!\n", saldo_conto);
  }
}
```

Esempio conto bancario: con mutex

```c
void* thread1() {
  int portamonete = 0;
  pthread_mutex_lock(&mutex);
  if (saldo_conto >= 100) {
    /* Preleva dal conto */
    printf("T1: Prelevo 100!\n");
    saldo_conto = saldo_conto - 100;
    portamonete = 100;
    controlla_saldo();
    pthread_mutex_unlock(&mutex);
  } else {
    pthread_mutex_unlock(&mutex);
    printf("T1: Non posso prelevare!\n");
  }
  if (portamonete == 100) {
     /* Fai gli acquisti */
  }
  /* Versamento di 50 */
  printf("T1: verso 50!\n");
  pthread_mutex_lock(&mutex);
  saldo_conto = saldo_conto + 50;
  pthread_mutex_unlock(&mutex);
  printf("T1: Il saldo è %d\n", saldo_conto);
}
```

```c
void* thread2()
{
  /* Versamento di 50 */
  printf("T2: verso 50!\n");
  pthread_mutex_lock(&mutex);
  saldo_conto = saldo_conto + 50;
  pthread_mutex_unlock(&mutex);
  printf("T2: Il saldo è %d\n", saldo_conto);

  /* ... Altre attività ... */
  pthread_mutex_lock(&mutex);
  if (saldo_conto >= 100) {
    /* Preleva dal conto */
    printf("T2: Prelevo 100!\n");
    saldo_conto = saldo_conto - 100;
    controlla_saldo();
    pthread_mutex_unlock(&mutex);
    /* Paga le fatture */
    paga_fatture(100);
  } else {
    pthread_mutex_unlock(&mutex);
    printf("T2: Non posso prelevare!\n");
  }
}
```

Esempio conto bancario: con mutex

```
void main()
{
  pthread_t t1, t2;

  pthread_mutex_init (&mutex, NULL);

  pthread_create(&t1, NULL, &thread1, NULL);
  pthread_create(&t2, NULL, &thread2, NULL);

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

  printf("Il saldo è %d\n", saldo_conto);

  pthread_mutex_destroy (&mutex);
}
```

Esempio conto bancario: con mutex

```
attila@dusty:~$ ./banca_mutex
T2: verso 50!
T1: Prelevo 100!
T1: verso 50!
T1: Il saldo è 50
T2: Il saldo è 100
T2: Prelevo 100!
Pago le fatture: 100
Il saldo è 0
```

```
attila@dusty:~$ ./banca_mutex
T1: Prelevo 100!
T1: verso 50!
T2: verso 50!
T2: Il saldo è 100
T2: Prelevo 100!
Pago le fatture: 100
T1: Il saldo è 50
Il saldo è 0
```

Anche se l'ordine delle operazioni è diverso (dipende dallo scheduling dei thread) in nessun caso il saldo diventa negativo

mutextrace

- Per analizzare le sequenze di lock e unlock possiamo utilizzare il tool mutextrace

```
attila@dusty:~$ mutextrace ./banca_mutex
init #1
thread create #1
lock #1, free -> thread 1
thread create #2
T1: Prelevo 100!
T2: verso 50!
unlock #1
T1: verso 50!
lock #1, free -> thread 2
unlock #1
lock #1, free -> thread 1
T2: Il saldo è 50
unlock #1
T1: Il saldo è 100
lock #1, free -> thread 2
thread exit #1
T2: Prelevo 100!
unlock #1
Pago le fatture: 100
thread exit #2
Il saldo è 0
destroy #1
```

Sorgente e pacchetti:
http://packages.debian.org/squeeze/mutextrace

Produttore e consumatore

- Il problema del produttore e del consumatore è un classico della programmazione concorrente

- Ho due thread, un **produttore** e un **consumatore**, che hanno accesso a una risorsa condivisa, il **magazzino** (di dimensione limitata)
  - Il produttore riempie il magazzino
    - Se il magazzino è pieno il produttore aspetta
  - Il consumatore svuota il magazzino
    - Se il magazzino è vuoto il consumatore aspetta fino a quando il magazzino torna a riempirsi
  - I processi di riempimento e svuotamento sono mutualmente esclusivi

Un po' di codice

```
#define N 10
int magazzino = 0;

void* produttore() {
  for(;;) {
    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
    }
  }
}
```

```
void* consumatore() {
  for(;;) {
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
    }
  }
}
```

... manca la mutua esclusione!

Un po' di codice

```
#define N 10
int magazzino = 0;
pthread_mutex_t mutex;

void* produttore() {
  for(;;) {
    pthread_mutex_lock(&mutex);
    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
    }
    pthread_mutex_unlock(&mutex);
}}
```

```
void* consumatore() {
  for(;;) {
    pthread_mutex_lock(&mutex);
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
    }
    pthread_mutex_unlock(&mutex);
}}
```

...già meglio
...ma il produttore non aspetta se il magazzino è pieno!
...e il consumatore non aspetta se il magazzino è vuoto!

Un po' di codice

```
#define N 10
int magazzino = 0;
pthread_mutex_t mutex;
pthread mutex_t elemento;

void* produttore() {
  for(;;) {
    pthread_mutex_lock(&mutex);
    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
    }
    pthread_mutex_unlock(&mutex);
    pthread_mutex_unlock(&elemento);
}}
```

Indica se c'è qualcosa in magazzino

```
void* consumatore() {
  for(;;) {
    pthread_mutex_lock(&elemento);
    pthread_mutex_lock(&mutex);
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
    }
    pthread_mutex_unlock(&mutex);
  }
}
```

... non funziona perché non posso sbloccare un mutex che è stato bloccato da un altro thread

Limiti dei mutex

- Un mutex rappresenta un valore binario
- Un mutex può essere sbloccato solo dal thread che lo ha bloccato
- Spesso ci si trova nella situazione in cui si vuole che più threads si trovino contemporaneamente all'interno della sezione critica
  - Un mutex non va bene, perché permette ad un solo thread alla volta di entrare

Semafori

- **Semaforo**
  - Un semaforo rappresenta un numero intero
    - Quando viene inizializzato è possibile specificare il valore del semaforo; successivamente sarà possibile solo incrementare (+1) o decrementare (-1) questo valore.
    - Quando un thread decrementa il semaforo, se il risultato è negativo, il thread si blocca e deve aspettare fintanto che un altro thread non incrementa il semaforo
    - Quando un thread incrementa il semaforo e ci sono altri thread in attesa, uno di questi viene sbloccato

Inizializzare un semaforo

- Per inizializzare un semaforo e assegnargli un valore iniziale

```
#include <semaphore.h>

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

Indica come può essere condiviso il semaforo:
- diverso da 0 se il semaforo è condiviso tra processi
- uguale a 0 se il semaforo è condiviso solo tra thread dello stesso processo

Valore iniziale del semaforo

Incrementare un semaforo

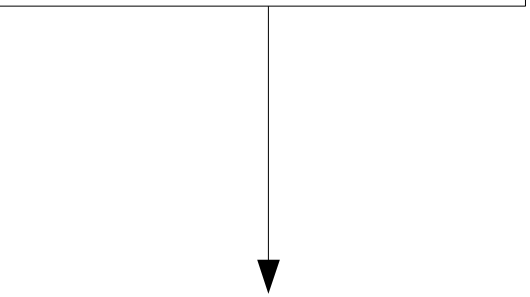
- Per incrementare un semaforo

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Decrementare un semaforo

- Per leggere il valore di un semaforo

```
#include <semaphore.h>

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



Distruggere un semaforo

- Per distruggere un semaforo

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

ATTENZIONE! Il comportamento del programma se si distrugge un semaforo in cui ci sono dei processi in attesa non è definito!

Decrementare un semaforo

- Per decrementare un semaforo

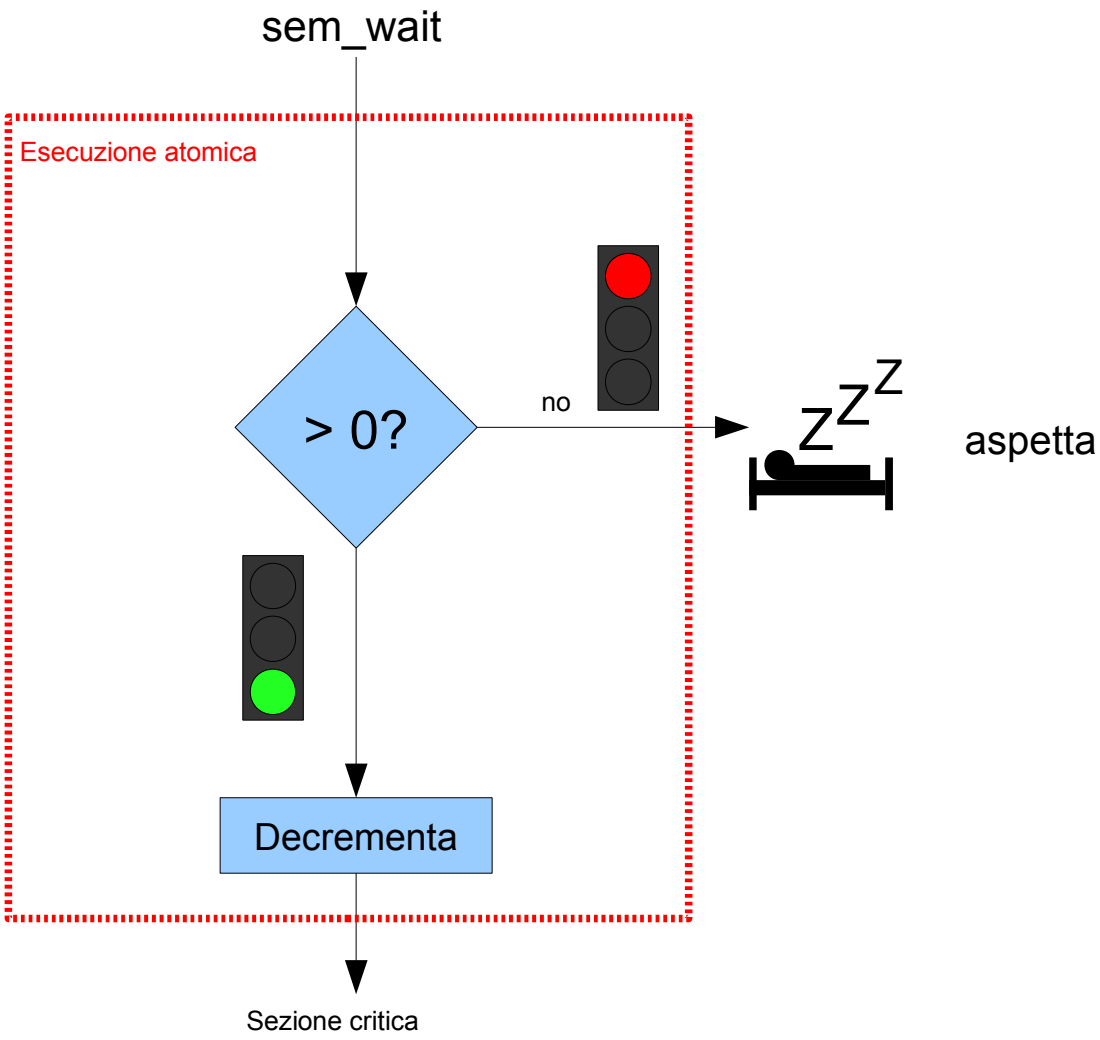
```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

  - attende che il semaforo assuma un valore positivo
  - decrementa il valore e, se non lo era già, fa riprendere l'esecuzione del thread

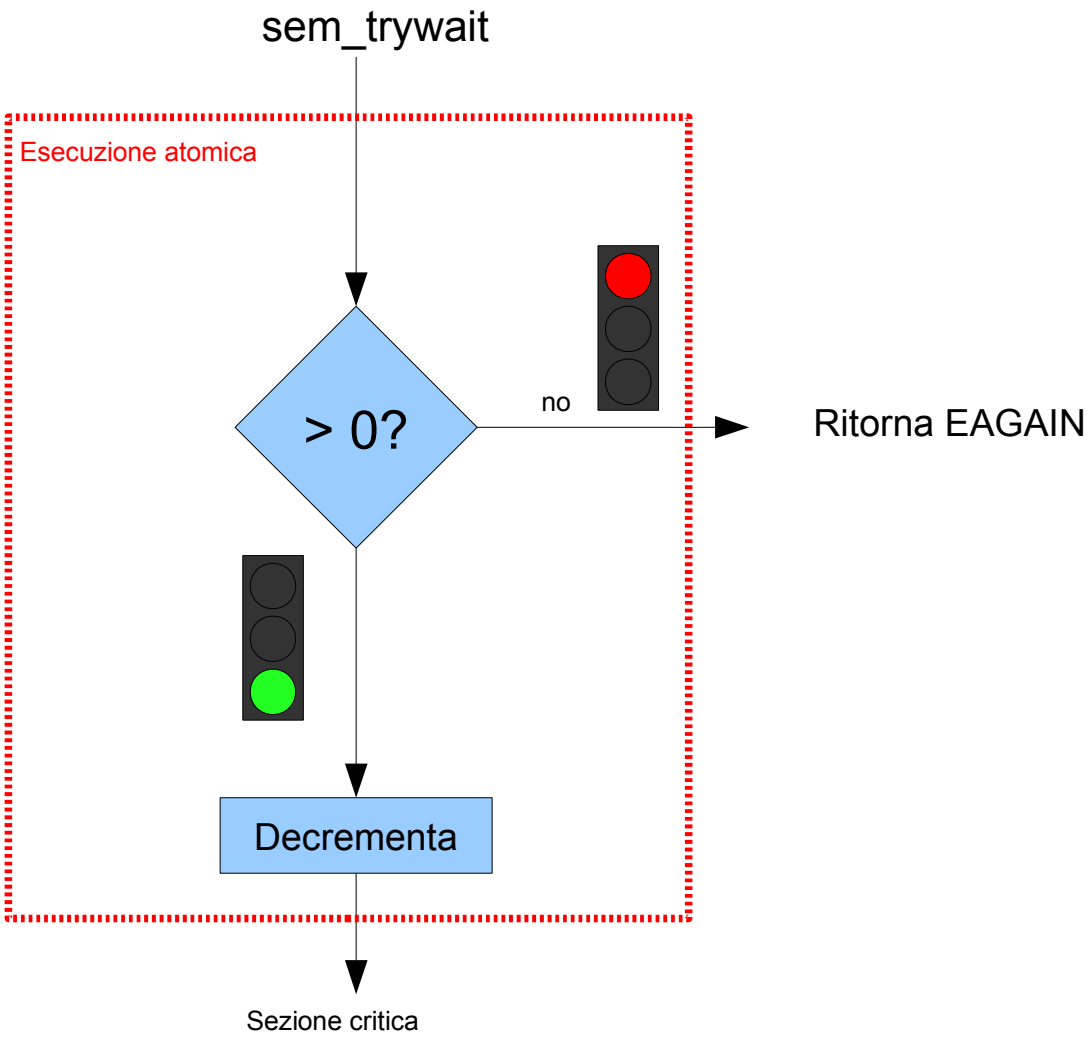
```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

  - come **`sem_wait()`** se il valore è positivo, altrimenti restituisce un errore (**`EAGAIN`**) senza bloccare il thread

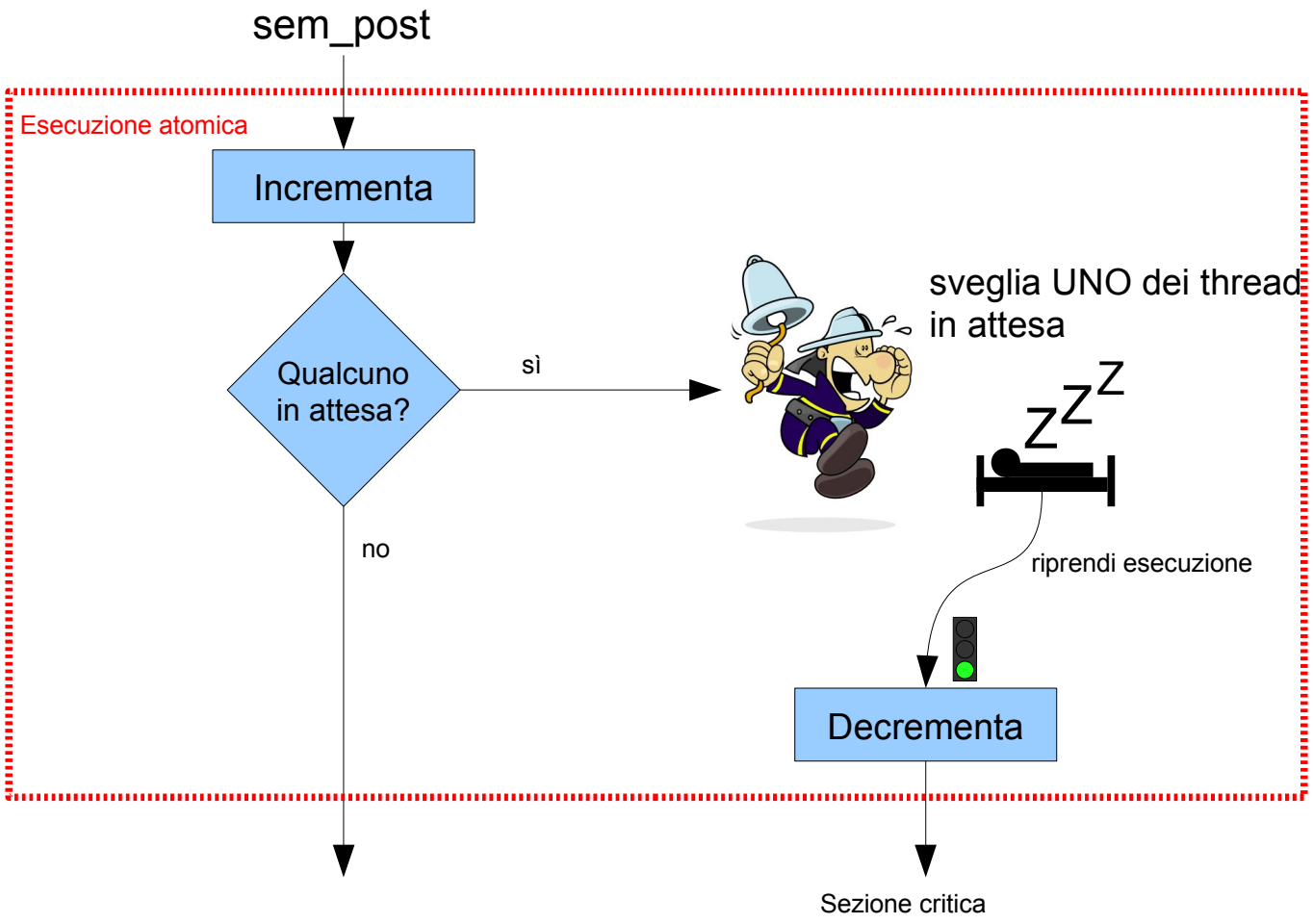
Semaforo


sem_wait
Esecuzione atomica
> 0?
no
aspetta
Decrementa
Sezione critica

Semaforo


sem_trywait
Esecuzione atomica
> 0?
no
Ritorna EAGAIN
Decrementa
Sezione critica

Semaforo


sem_post
Esecuzione atomica
Incrementa
Qualcuno in attesa?
sì
no
sveglia UNO dei thread in attesa
riprendi esecuzione
Decrementa
Sezione critica

Esempio: Produttore consumatore con un semaforo

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

#define N 10 /* Dimensione del magazzino */

int magazzino = 0;
pthread_mutex_t mutex;
sem_t elementi_in_magazzino;
```

Esempio: Produttore consumatore con un semaforo

```
void* produttore() {
  for(;;) {
    pthread_mutex_lock(&mutex);
    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
      printf("Produco (ce ne sono
              %d)\n", magazzino);
    } else {
      printf("Non ce ne stanno!\n");
    }
    pthread_mutex_unlock(&mutex);
    sem_post(&elementi_in_magazzino);
  }
}
```

```
void* consumatore() {
  for(;;) {
    sem_wait(&elementi_in_magazzino);
    pthread_mutex_lock(&mutex);
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
      printf("Consumo (ne restano
              %d)\n", magazzino);
    } else {
      printf("Non c'è niente! Mi sono
svegliato inutilmente!\n");
    }
    pthread_mutex_unlock(&mutex);
  }
}
```

Esempio: Produttore consumatore con un semaforo

```
void main()
{
  pthread_t t1, t2;

  pthread_mutex_init (&mutex, NULL);

  sem_init(&elementi_in_magazzino, 0, 0);

  pthread_create(&t1, NULL, &produttore, NULL);
  pthread_create(&t2, NULL, &consumatore, NULL);

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

  pthread_mutex_destroy (&mutex);
  sem_destroy(&elementi_in_magazzino);
}
```

Manca ancora qualcosa...

```
Non ce ne stanno!
Non ce ne stanno!
Non ce ne stanno!
Non ce ne stanno!
Non ce ne stanno!
Consumo (ne restano 9)
Consumo (ne restano 8)
Consumo (ne restano 7)
Consumo (ne restano 6)
Consumo (ne restano 5)
Consumo (ne restano 4)
Consumo (ne restano 3)
Consumo (ne restano 2)
Consumo (ne restano 1)
Consumo (ne restano 0)
Non c'è niente! Mi sono svegliato inutilmente!
Non c'è niente! Mi sono svegliato inutilmente!
Non c'è niente! Mi sono svegliato inutilmente!
Non c'è niente! Mi sono svegliato inutilmente!
N
```

... non ottimale perché produttore e consumatore non si mettono in attesa!

Esempio: Produttore consumatore con due semafori

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

#define N 10 /* Dimensione del magazzino */
int magazzino = 0;
pthread_mutex_t mutex;
sem_t spazio_in_magazzino; /* Inizializzato a N */
sem_t elementi_in_magazzino; /* Inizializzato a 0 */
```

Esempio: Produttore consumatore con due semafori

```
void* produttore() {
  for(;;) {
    sem_wait(&spazio_in_magazzino);
    pthread_mutex_lock(&mutex);
    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
      printf("Produco (ce ne sono
              %d)\n", magazzino);
    } else {
      printf("Non ce ne stanno!\n");
    }
    pthread_mutex_unlock(&mutex);
    sem_post(&elementi_in_magazzino);
  }
}
```

```
void* consumatore() {
  for(;;) {
    sem_wait(&elementi_in_magazzino);
    pthread_mutex_lock(&mutex);
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
      printf("Consumo (ne restano
              %d)\n", magazzino);
    } else {
      printf("Non c'è niente! Mi sono
svegliato inutilmente!\n");
    }
    pthread_mutex_unlock(&mutex);
    sem_post(&spazio_in_magazzino);
  }
}
```

Esempio: Produttore consumatore con due semafori

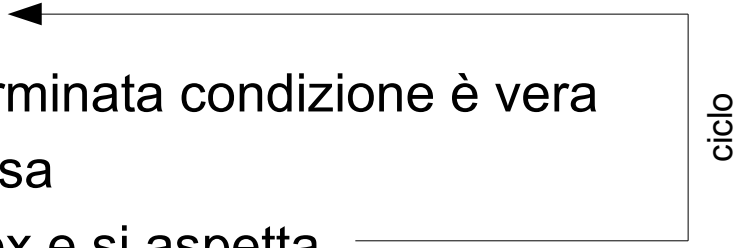
```
void main()
{
  pthread_t t1, t2;
  pthread_mutex_init (&mutex, NULL);

  sem_init(&elementi_in_magazzino, 0, 0);
  sem_init(&spazio_in_magazzino, 0, N);

  pthread_create(&t1, NULL, &produttore, NULL);
  pthread_create(&t2, NULL, &consumatore, NULL);

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

  pthread_mutex_destroy (&mutex);
  sem_destroy(&elementi_in_magazzino);
  sem_destroy(&spazio_in_magazzino);
}
```

Adesso funziona correttamente

```
Produco (ce ne sono 1)
Produco (ce ne sono 2)
Produco (ce ne sono 3)
Produco (ce ne sono 4)
Produco (ce ne sono 5)
Produco (ce ne sono 6)
Produco (ce ne sono 7)
Produco (ce ne sono 8)
Produco (ce ne sono 9)
Produco (ce ne sono 10)
Consumo (ne restano 9)
Consumo (ne restano 8)
Consumo (ne restano 7)
Consumo (ne restano 6)
Consumo (ne restano 5)
Consumo (ne restano 4)
Consumo (ne restano 3)
Consumo (ne restano 2)
Consumo (ne restano 1)
Consumo (ne restano 0)
```

Variabile di condizione

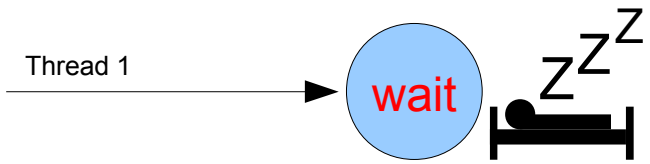
- Nella programmazione concorrente ci si trova spesso nella situazione seguente:
  - 1. Si acquisisce un mutex
  - 2. Si controlla se una determinata condizione è vera
    - se sì, si esegue qualcosa
    - se no, si rilascia il mutex e si aspetta



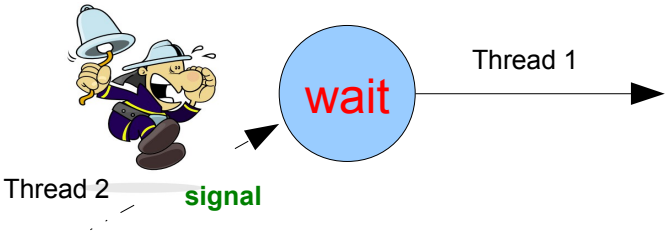
- Questa è una forma di busy waiting → non è efficiente!

Variabile di condizione

- È un'alternativa ai semafori
- Una variabile di condizione (*condition variable*) ha associate due primitive
  - **wait** (aspetta)
  - **signal** (segnala, risveglia)



Il thread 1 aspetta sulla variabile di condizione



Il thread 2 risveglia la variabile di condizione,
il thread 1 riprende l'esecuzione

Variabile di condizione

- Combina l'attesa del cambiamento di stato di una variabile con la mutua esclusione
  - inizializzo una variabile di condizione con **`pthread_cond_init`**
  - attendo su una variabile di condizione con **`pthread_cond_wait`**
  - sveglio i processi in attesa con **`pthread_cond_signal`** o **`pthread_cond_broadcast`**

Variabile di condizione

- **`p_thread _cond_wait()`** sblocca il mutex (e permette quindi ad altri thread di accedere) e poi aspetta finché non si realizzi la condizione specificata, segnalata da un evento
  - durante l'attesa il mutex rimane sbloccato, e il thread è sospeso (non ho un'attesa attiva)

Variabile di condizione

- Inizializzare una variabile di condizione

```
#include <semaphore.h>

int pthread_cond_init(pthread_cond_t *cond,
            const pthread_condattr_t *attr);
```

Attesa

- Per attendere su una variabile di condizione

```
#include <semaphore.h>

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

Durante l'attesa il mutex viene rilasciato.

Segnale

- Per attendere su una variabile di condizione

```
#include <semaphore.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

- Con questa chiamata si sveglia **almeno un thread** in attesa
  - se non c'è nessun thread in attesa non succede nulla
- Se più threads sono in attesa della stessa variabile tutti vengono svegliati e poi si contendono il mutex come se ognuno facesse `pthread_mutex_lock()` → **in tutti i casi solo un thread alla volta potrà riprendere il possesso del thread!**

Segnale

- Per attendere su una variabile di condizione

```
#include <semaphore.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Con questa chiamata si sveglia **tutti i thread** in attesa
  - se non c'è nessun thread in attesa non succede nulla
- Se più threads sono in attesa della stessa variabile tutti vengono svegliati e poi si contendono il mutex come se ognuno facesse `pthread_mutex_lock()` → **in tutti i casi solo un thread alla volta potrà riprendere il possesso del thread!**

Esempio: produttore consumatore con variabile di condizione

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

#define N 10
int magazzino = 0;
pthread_mutex_t mutex;
pthread_cond_t spazio_in_magazzino;
pthread_cond_t elementi_in_magazzino;
```

Esempio: produttore consumatore con variabile di condizione

```
void* produttore() {
  for(;;) {
    pthread_mutex_lock(&mutex);

    if (magazzino == N) {
      pthread_cond_wait
       (&spazio_in_magazzino, &mutex);
    }

    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
      printf("Produco (ce ne sono
       %d)\n", magazzino);
    } else {
      printf("Non ce ne stanno!\n");
    }
    pthread_cond_signal(&elementi_in_magazzino);
    pthread_mutex_unlock(&mutex);
  }
}
```

```
void* consumatore() {
  for(;;) {
    pthread_mutex_lock(&mutex);

    if (magazzino == 0) {
      pthread_cond_wait(&elementi_in_magazzino,
          &mutex);
    }
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
      printf("Consumo (ne restano %d)\n",
            magazzino);
    } else {
      printf("Non c'è niente! Mi sono svegliato
            inutilmente!\n");
    }

    pthread_cond_signal(&spazio_in_magazzino);
    pthread_mutex_unlock(&mutex);
  }
}
```

Esempio: produttore consumatore con variabile di condizione

```
void main()
{
  pthread_t t1, t2;

  pthread_mutex_init (&mutex, NULL);

  pthread_cond_init(&elementi_in_magazzino, NULL);
  pthread_cond_init(&spazio_in_magazzino, NULL);

  pthread_create(&t1, NULL, &produttore, NULL);
  pthread_create(&t2, NULL, &consumatore, NULL);

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

  pthread_mutex_destroy (&mutex);
  pthread_cond_destroy(&elementi_in_magazzino);
  pthread_cond_destroy(&spazio_in_magazzino);
}
```

Cosa succede se abbiamo più di un consumatore?

1) Il primo consumatore, Aldo, trova il magazzino vuoto e aspetta

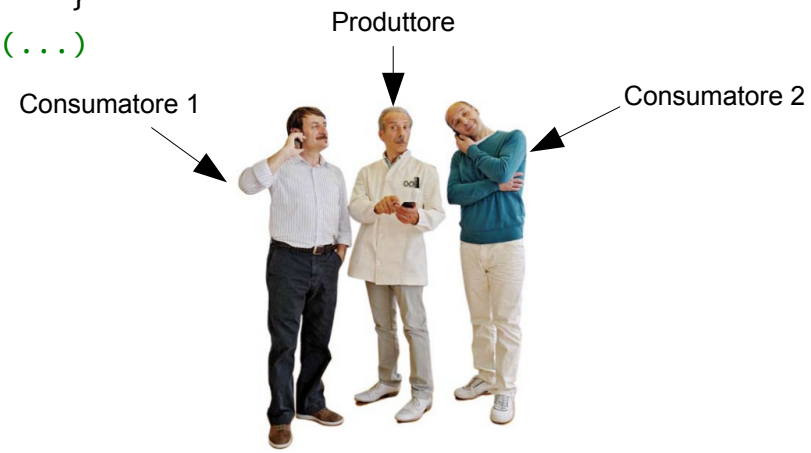
2) Il produttore, Giovanni, produce un nuovo elemento, e poi sveglia Aldo

3) Giovanni termina, e il sistema operativo decide di eseguire il secondo consumatore, Giacomo

4) Giacomo vede che c'è un elemento, quindi lo consuma e esce dalla sezione critica

5) Il sistema operativo esegue Aldo, che si è svegliato grazie al segnale inviato da Giovanni. **Aldo riacquisisce il mutex ma poi si ritrova senza elementi! Errore!**

```
void* consumatore() {
  for(;;) {
    pthread_mutex_lock(&mutex);

    if (magazzino == 0) {
      pthread_cond_wait(&elementi_in_magazzino,
          &mutex);
    }
    if (magazzino > 0) {
      (...)
    } else {
      printf("Non c'è niente! Mi sono svegliato
            inutilmente!\n");
    }
(...)
```



Esempio: produttore consumatore con variabile di condizione (versione 2)

```
void* produttore() {
  for(;;) {
    pthread_mutex_lock(&mutex);

    while (magazzino == N) {
      pthread_cond_wait
       (&spazio_in_magazzino, &mutex);
    }

    if (magazzino < N) {
      /* Produce un elemento  */
      magazzino = magazzino + 1;
      printf("Produco (ce ne sono
       %d)\n", magazzino);
    } else {
      printf("Non ce ne stanno!\n");
    }
    pthread_cond_signal(&elementi_in_magazzino);
    pthread_mutex_unlock(&mutex);
  }
}
```

Quando mi sveglio, ricontrollo prima di continuare

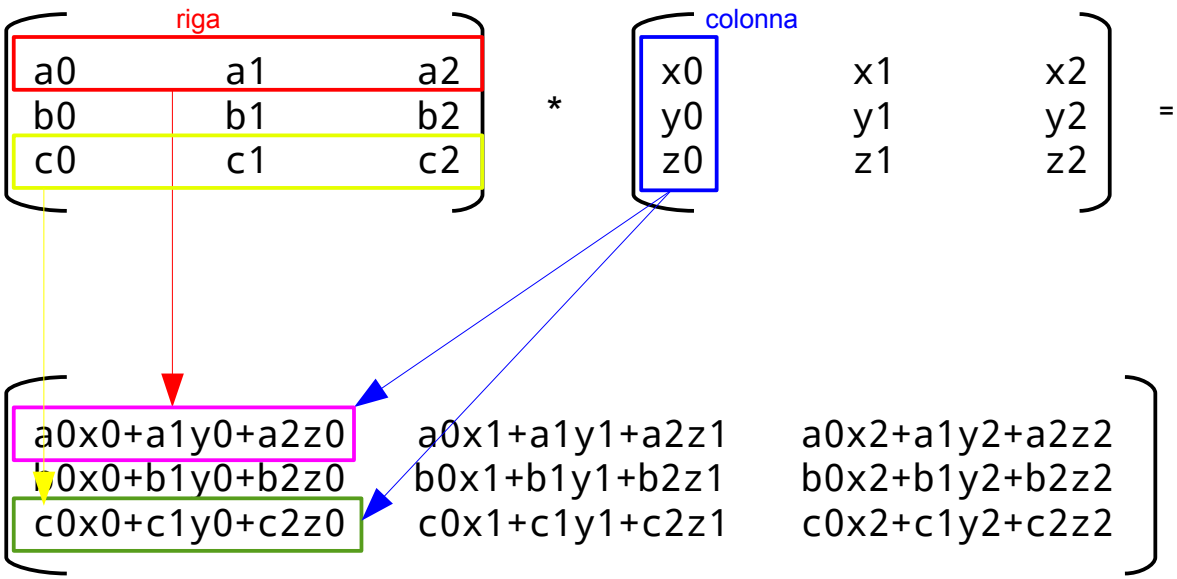
```
void* consumatore() {
  for(;;) {
    pthread_mutex_lock(&mutex);

    while (magazzino == 0) {
      pthread_cond_wait(&elementi_in_magazzino,
          &mutex);
    }
    if (magazzino > 0) {
      /* Consuma un elemento  */
      magazzino = magazzino - 1;
      printf("Consumo (ne restano %d)\n",
            magazzino);
    } else {
      printf("Non c'è niente! Mi sono svegliato
            inutilmente!\n");
    }

    pthread_cond_signal(&spazio_in_magazzino);
    pthread_mutex_unlock(&mutex);
  }
}
```

Spurious wakeup

- Può succedere che un thread si svegli da `pthread_cond_wait` anche se nessun altro thread ha fatto un segnale o un broadcast (soprattutto nei sistemi multi-processore): si parla di **spurious wakeup**.
  - Per questo motivo è sempre utile controllare la condizione prima di riprendere l'esecuzione.

Un nuovo problema...

- Consideriamo il problema della moltiplicazione di due matrici:



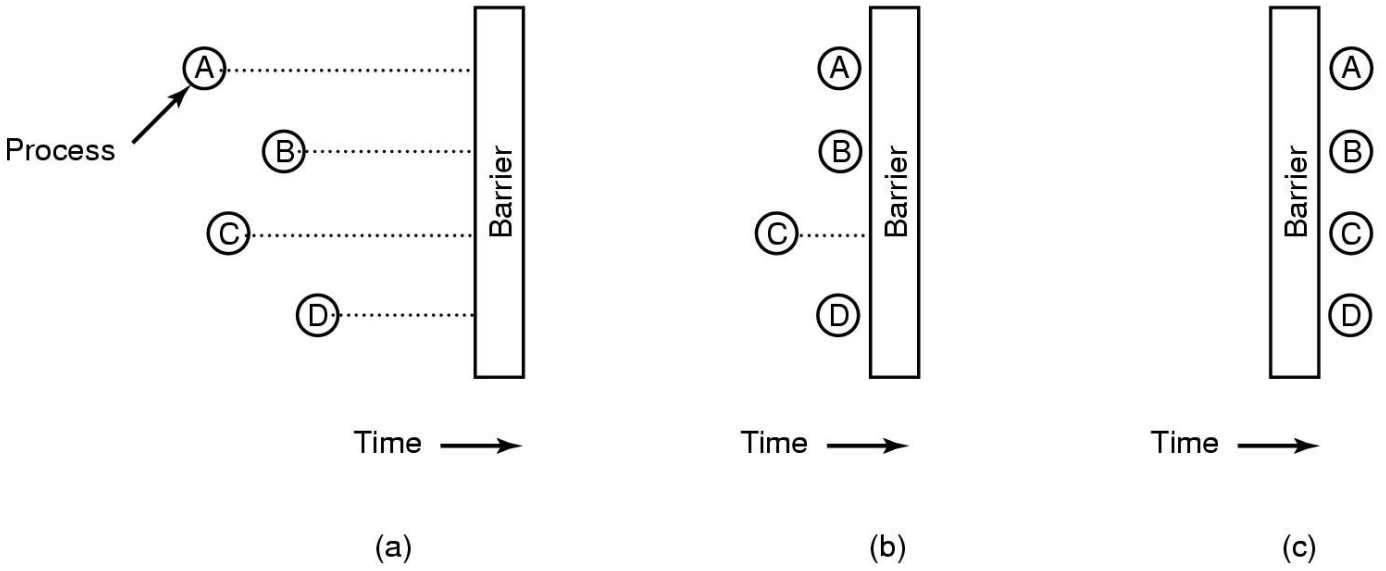
Moltiplicazione di due matrici

- Il problema può essere suddiviso in più parti, che possono essere eseguite in parallelo
  - un thread per ogni elemento della matrice risultato

- Data una matrice A, vogliamo calcolare A^3
  - in questo caso il processo richiede due passi:
    1. moltiplicazione di A * A = B
    2. moltiplicazione di B * A = C
       - il secondo passo deve attendere la conclusione del primo!

Applicazione della barriera

- Il meccanismo della barriera è utile per sincronizzare l'avanzamento di un algoritmo parallelo, dove il passo successivo dipende dall'ottenimento del risultato del passo corrente



A. Tanenbaum, Modern Operating Systems, 2nd Edition

Barriera

- Inizializzazione di una barriera

```
#include <pthread.h>

int pthread_barrier_init
            (pthread_barrier_t *restrict barrier,
      const pthread_barrierattr_t *restrict attr,
            unsigned count);
```

Numero di thread che devono sincronizzarsi

Proprietà (tipicamente NULL, per avere le proprietà di default, ovvero una barriera condivisa solo tra thread e non tra processi)

Distruggere una barriera

- Per distruggere una barriera dopo l'utilizzo:

```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t
                                    *barrier);
```

Raggiungere la barriera e aspettare

- Un thread può mettersi in attesa su una barriera con:

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t
                         *barrier);
```

- Quando la barriera si sblocca ogni thread riceve **PTHREAD_BARRIER_SERIAL_THREAD**

Una matrice al cubo

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

double A[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9} };
double B[3][3];
double C[3][3];

pthread_barrier_t barriera;

void moltiplica(double X[3][3], double Y[3][3], double Z[3][3],
                int riga, int colonna) {
  int i;
  Z[riga][colonna] = 0;
  for(i=0; i<3; i++) {
    Z[riga][colonna] += X[riga][i] * Y[i][colonna];
  }
}
```

Una matrice al cubo

```
void* thread_moltiplica (void* args)
{
  int result;
  int riga = (int) args / 3;
  int colonna = (int) args % 3;

  moltiplica(A, A, B, riga, colonna);

  result = pthread_barrier_wait(&barriera);

  if (result != 0 && result != PTHREAD_BARRIER_SERIAL_THREAD)
{
    perror("Errore nell'attesa sulla barriera!\n");
    exit(-1);
  }
```

Una matrice al cubo

```
void main()
{
  /* Un thread per ogni elemento della matrice */
  pthread_t threads[9];
  int t, r, c;
  if(!pthread_barrier_init(&barriera, NULL, 9)) {
    for (t=0; t<9; t++) {
      pthread_create(&threads[t], NULL, &thread_moltiplica, (void*) t);
    }
    for (t=0; t<9; t++) {
      pthread_join(threads[t], NULL);
    }
    printf("La matrice elevata al cubo è:\n");
    for (r=0; r<3; r++) {
      for (c=0; c<3; c++) {
        printf(" %3.2f ", C[r][c]);
      }
      printf("\n");
    }
    pthread_barrier_destroy(&barriera);
  }
}
```

Monitor

- Solo un thread alla volta può ritrovarsi in esecuzione all'interno del monitor

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Metodi **synchronized** in Java

Da A. Tanenbaum, Modern Operating Systems, 2nd ed

Monitor

Magazzino = risorsa condivisa

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Sincronizzazione con scambio di messaggi

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item();                /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);                /* get message containing item */
          item = extract_item(&m);              /* extract item from message */
          send(producer, &m);                   /* send back empty reply */
          consume_item(item);                   /* do something with the item */
     }
}
```

Chiamata bloccante, aspetta finché arriva un messaggio

Chiamata bloccante, aspetta finché arriva un messaggio

Da A. Tanenbaum, Modern Operating Systems, 2nd ed