Corso di Programmazione Concorrente

POSIX Thread

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Implementazioni thread piattaforme UNIX. POSIX
  - LinuxThreads – NPTL
- Creazione e terminazione di thread, *demoni*
- Passare parametri e raccogliere risultati dai thread
- Corse critiche
- Mutex vs semafori di Dijkstra
- Mutex
  - Creazione e distruzione
  - **`lock()`**, **`unlock()`**
  - tipologie ed attributi dei mutex
- Semafori per thread vs semafori di Dijkstra
- Variabili Condizioni
  - **`signal()`** and **`wait()`**
- TSD

POSIX Thread sotto Linux

- Standard POSIX molto ampio, recepito dai S.O. UNIX
- Include anche una sezione specifica per i thread
  - POSIX thread o Pthread
- Sotto linux, due diverse implementazioni:
  - LinuxThreads (ormai obsoleta) – NPTL
- Nessuna è perfettamente aderente allo standard
  - LinuxThreads: parziale implementazione user-level ormai obsoleta che risultò l'unica disponibile sino ai kernel 2.2
  - Non richiese cambiamenti al kernel linux
  - Largamente basata sulla chiamata di sistema **`clone()`**

`man 2 clone`

- NPTL: evoluzione kernel-level che ambisce ad implementare lo standard POSIX completamente
  - Anche grazie a significativi cambiamenti nel kernel Linux

Thread per Linux

- Disponibili da tempo in varie librerie
- Una, “LinuxThreads”, è presente in tutte le distr. precedenti il kernel 2.4
  - `libpthread`
- Sfruttano la chiamata di sistema `clone()` che crea un nuovo processo senza cambiare spazio di indirizzamento
- Questo ha permesso di implementare i thread relativamente facilmente, però…

Thread per Linux

- Disponibili da tempo in varie librerie
- Una, “LinuxThreads”, è presente in tutte le distr. precedenti il kernel 2.4
  - `libpthread`
- Sfruttano la chiamata di sistema `clone()` che crea un nuovo processo senza cambiare spazio di indirizzamento
- Questo ha permesso di implementare i thread relativamente facilmente, però…

Linux e lo Standard POSIX

- I thread di Linux secondo LinuxThreads sono processi che condividono lo spazio di indirizzamento
- Le raccomandazioni **POSIX** aiutano a garantire un certo grado di uniformità nella gestione dei thread di vari S.O. (in particolare *dialetti* UNIX)
- LinuxThreads non è completamente **POSIX** compliant, in particolare per quanto riguarda
  - la gestione dei segnali
  - Process Identifier / Thread Identifier

  ed in generale per tutti gli aspetti in cui la vera natura dei thread di LinuxThreads non può essere nascosta

Evidenti Discrepanze tra LinuxThreads e Standard POSIX: `thread-pid.c`

```
…include omessi…

void* thread_function (void* arg) {
  fprintf(stderr, "child thread pid is %d\n", (int)getpid());
  while (1); /* Spin forever.  */
  return NULL;
}

int main() {
  pthread_t thread;
  fprintf(stderr, "main thread pid is %d\n", (int)getpid());
  pthread_create(&thread, NULL, &thread_function, NULL);
  while (1); /* Spin forever.  */
  return 0;
}
```

thread-pid.c: Esempio di Esecuzione

```
$ ./thread-pid
main thread pid is 9154
child thread pid is 9156
$ ps x
…
 9154 pts/0    R      0:01 ./thread-pid
 9155 pts/0    S      0:00 ./thread-pid
 9156 pts/0    R      0:01 ./thread-pid
 9157 pts/0    R      0:00 ps x
…
```

I Nuovi Thread per Linux: NPTL

- Grazie all'interesse di IBM e Red Hat, si è poi sviluppato un supporto "nativo" ai thread per Linux
- NPTL: Native POSIX Thread Linux
- E' distribuita con tutte le versioni più recenti dal kernel 2.6 in poi
  - Inizialmente non tutte le distribuzioni la attivavano NPTL automaticamente
- Risultano indiscutibili
  - i vantaggi in termini di prestazioni
  - la maggiore aderenza allo standard POSIX

Compilazione di Programmi Multithread

`#include <pthread.h>`

- Prototipi delle funzioni e include giusti per le chiamate di sistema thread safe

`gcc -lpthread …`

La libreria dei thread

- POSIX richiede che tutte le chiamate dello standard ANSI C e quasi tutte le chiamate di POSIX.1 siano thread-safe
- Attenzione: dalle più recenti versioni di `gcc` è significativo l'ordine dei parametri della linea di comando

C: Allocazione Memoria e Thread

- Per evitare corse critiche dei thread sulle aree di memoria, è bene ricordare che:
  - Variabili Globali
    - allocate nel segmento dati
    - visibili da tutti i thread del processo iniziale
  - Variabili Automatiche
    - allocate sullo stack all'inizio di un blocco
    - quindi "locali" al thread
    - sono deallocate alla fine del blocco
      - esiste il pericolo di puntatori pendenti ←
  - Memoria allocata con la `malloc()`
    - allocata sullo *heap* e condivisa da tutti i thread che ne conoscono l'indirizzo
    - va deallocata esplicitamente chiamando `free()`

Pthread: Creazione e Terminazione

- `pthread_create()`
  - crea un thread che esegue la funzione specificata
- `pthread_exit()`
  - termina il thread corrente
- `pthread_join()`
  - sospende il thread corrente sino al termine del thread specificato

Creazione di Thread (1)

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

- `pthread_create` crea un nuovo thread
- il nuovo thread esegue `start_routine` a cui passa `arg`
- il nuovo thread può terminare esplicitamente (`pthread_exit()`), od implicitamente quando finisce `start_routine()`

Creazione di Thread (2)

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

- `attr` specifica gli attributi del thread
  - `NULL` = valori di default;
  - esistono diversi attributi; consultare

    ```
    man 3 pthread_attr_init
    ```

  ad es. i thread possono o meno essere “joinable”
- in caso di successo `tid` conterrà l’identificatore del thread creato e la funzione ritorna `0`; in caso di fallimento viene restituito un codice di errore $\neq 0$

Terminazione di Thread

```
#include <pthread.h>
void pthread_exit(void *retval)
```

- `retval` specifica un valore di ritorno consultabile da altri thread con `pthread_join`
- la funzione non ritorna!
- causa l'esecuzione di alcune operazioni di cleanup secondo quanto predisposto dal thread chiamante

Join con un Thread

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

- sospende l’esecuzione del thread corrente fino alla fine del thread `th`
- se `thread_return` non è `NULL`, è l'indirizzo di una area di memoria che al ritorno conterrà il valore di tipo `void *` che il thread terminante ha restituito tramite `pthread_exit()`
- la fine di un thread può essere attesa al più da un solo altro thread

Supporto alla Join

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

- Il supporto alle chiamate corrispondenti alla *join() astratta* è oneroso: analogamente al caso della `wait()` per processi, per supportare questa chiamata, quando un thread termina, non tutte le risorse allocate per la sua gestione (ad es. il descrittore di thread) possono essere immediatamente rimosse
- Per questo motivo è lasciata la possibilità di creare thread *detached* o *demoni* su cui non è possibile effettuare la `join()`
  - In questo modo non è necessario mantenere allocate delle risorse alla loro terminazione >>

Supporto all' *Hand-Off*

- Le segnature delle chiamate
  - `pthread_create()`
  - `pthread_exit()`
  - `pthread_join()`
- Sono state sensibilmente complicate dalla gestione di un *hand-off* bidirezionale:
  - Un `void *` viene passato da un thread che invoca `pthread_create()` al nuovo thread
  - Un `void *` viene restituito da un thread che termina al thread che effettua la `pthread_join()`
- Soprattutto quando questo meccanismo rimane inutilizzato, le segnature possono essere percepite come eccessivamente complicate
- Tuttavia, in molti casi consentono di evitare la creazione di strutture utili solo all'*hand-off*

Esempio

```
…
void mainline (...) {
  struct phonebookentry *pbe;
  pthread_t helper;

  /* crea thread dedicato per un'operazione lunga */
  pthread_create(&helper, NULL, fetch, &pbe);
  …
  /* eventualmente qui si possono fare altre operazioni */
  …
  pthread_join(helper, NULL); // attende la fine del-
  //l'operazione lunga prima di utilizzarne i risultati
}

void *fetch(struct phonebookentry *arg) {
  struct phonebookentry *npbe;
  npbe = search(name); // preleva il valore da un db
  if (npbe != NULL) *arg = *npbe;
  pthread_exit(0);
}
```

firstthread.c

```
#include <stdlib.h>
#include <unistd.h>
void *thread_function(void *arg) {
  int i;
  for ( i=0; i<20; i++ ) {
    printf("Thread says hi!\n");
    sleep(1);
  }
  return NULL;
}
int main(void) {
  pthread_t mythread;
  if (pthread_create(&mythread,NULL,thread_function,NULL)) {
    printf("error creating thread."); abort();
  }
  if (pthread_join(mythread,NULL)) {
    printf("error joining thread.");  abort();
  }
  exit(0);
}
```

firstthread.c:
Esempio di Compilazione ed Esecuzione

```
$ gcc -lpthread firstthread.c -o firstthread
$ ./firstthread
Thread says hi!
Thread says hi!
Thread says hi!
Thread says hi!
Thread says hi!
…
Thread says hi!
Thread says hi!
Thread says hi!
Thread says hi!
Thread says hi!
Thread says hi!
$
```

Passare Parametri ai Thread (1)

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

/* Parameters to print_function.  */
struct char_print_parms {
  char character; // char to print
  int count;      // number of times to print it.
};
/* Prints a number of characters to stderr, as given by
   PARAMETERS, which is a pointer to a
   struct char_print_parms.  */
void* char_print (void* parameters) {
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p =
       (struct char_print_parms*)parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
…
```

Passare Parametri ai Thread (2)

```
…
/* The main program.  */
int main() {
  pthread_t thread1_id, thread2_id;
  struct char_print_parms thread1_args, thread2_args;

  /* Create a new thread to print 30000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create(&thread1_id,NULL,&char_print,&thread1_args);

  /* Create a new thread to print 20000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create(&thread2_id,NULL,&char_print,&thread2_args);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);
  return 0; /* Now we can safely return.  */
}
```

primes.c: Ricevere Risultati dai Thread (1)

```
/* Compute successive prime numbers (very inefficiently).  Return the
   Nth prime number, where N is the value pointed to by *arg.  */

void* compute_prime (void* arg) {
  int candidate = 2;
  int n = *((int*) arg);

  while (1) {
    int factor, is_prime = 1;
    /* Test primality by successive division.  */
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
        is_prime = 0;
        break;
      }
    /* Is this the prime number we're looking for?  */
    if (is_prime) {
      if (--n == 0)
    /* Return the desired prime number as the thread return value.  */
        return (void*) candidate;
    }
    ++candidate;
  }
  return NULL;
}…
```

primes.c: Ricevere Risultati dai Thread (2)

```
…
int main() {
  pthread_t thread;
  int which_prime = 5000;
  int prime;
  /* Start the computing thread,
     up to the 5000th prime number.  */
  pthread_create(&thread,NULL,&compute_prime,&which_prime);
  /* Do some other work here...  */
  /* Wait for the prime number thread to complete,
     and get the result.  */
  pthread_join(thread,(void*)&prime);
  /* Print the largest prime it computed.  */
  printf("The %dth prime number is %d.\n",which_prime,prime);
  return 0;
}
```

Detaching di un Thread: *Demoni*

```
#include <pthread.h>
int pthread_detach(pthread_t th);
```

- il thread specificato viene messo nello stato di *detached*
- al suo termine, le risorse consumate saranno immediatamente rilasciate anche senza il join di altri thread
- in caso di successo ritorna `0`; in caso di fallimento un codice di errore $\neq$`0`
- un thread può anche essere creato direttamente detached specificando opportunamente il campo relativo agli attributi del thread creato nella funzione `pthread_create()`

detached.c

```
#include <pthread.h>

void* thread_function (void* thread_arg) {
  /* Do work here...  */
  return NULL;
}
int main() {
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  pthread_create(&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);
  /* Do work here...  */

  /* No need to join the second thread.  */
  return 0;
}
```

Attributi nello Standard POSIX

- Questo è solo un primo esempio di *attributo,* uno strumento usato dallo standard POSIX per specificare le caratteristiche degli oggetti creati
- Gli attributi sono
  - creati con una funzione `X_init`
  - deallocati con una funzione `X_destroy`
  - modificati con una funzione `X_set`…
- Nello standard POSIX è possibile associare attributi che definiscano le caratteristiche di
  - mutex
  - variabili condizioni
  - thread

Thread in C: Competizione

- I principali strumenti per la risoluzione dei problemi di competizione sono i mutex
- In sostanza, semafori binari in stile competitivo

- Esiste anche una versione di semafori per thread
- Tuttavia in assenza di motivazioni per fare altrimenti, è opportuno usare sempre i mutex
- ✔ Chiarisce che si sta cercando di risolvere un problema di competizione

race.c (1): Un Esempio di Interferenza

```
…include omessi…
int myglobal; // variabile globale
int main(void) {
  pthread_t mythread;
  int i;
  if (pthread_create(&mythread,NULL,thread_function,NULL)) {
    printf("creazione del thread fallita."); exit(1); }
  for ( i=0; i<20; i++) {
    myglobal=myglobal+1;
    printf("o");
    fflush(stdout);
    sleep(1);
  }
  if (pthread_join(mythread,NULL)) {
    printf("errore nel join dei thread."); exit(2); }
  printf("\nmyglobal è uguale a %d\n",myglobal);
  exit(0);
}…
```

race.c (2): Un Esempio di Interferenza

```
…
void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++ ) {
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);
    myglobal=j;
  }
  return NULL;
}
```

`race.c` (3): Esempio di Esecuzione

Compilazione

```
$ gcc race.c -o race -lpthread
```

Esecuzione:

```
$ ./race
```

Possibile Output:

```
..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
myglobal è uguale a 21
```

Pthread - Mutex

- `pthread_mutex_init()`
  - inizializza il mutex specificato
- `pthread_mutex_destroy()`
  - dealloca tutte le risorse allocate per gestire il mutex specificato
- `pthread_mutex_lock()`
  - blocca il mutex
- `pthread_mutex_unlock()`
  - sblocca il mutex

Mutex vs Semafori di Dijkstra

- Rispetto alla versione presentata astrattamente, i mutex POSIX
  - corrispondono ai semafori binari
    - sono destinati a risolvere problemi di competizione
    - per la cooperazione esistono le variabili condizione
  - ne esistono di tre tipologie diverse
    - fast
    - recursive
    - error-checking

Pthread: Tre Tipologie di Mutex

- fast: semantica classica, pertanto un thread che esegue due `mutex_lock()` consecutivi sullo stesso mutex causa uno stallo
- recursive: conta il numero di volte che un thread blocca il mutex e lo rilascia solo se esegue un pari numero di `mutex_unlock()`
- error-checking: controlla che il thread che rilascia il mutex sia lo stesso thread che lo possiede

Inizializzazione di Mutex

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

- i mutex sono di tipo `pthread_mutex_t`
- inizializzazione di un mutex
  - statica, macro per inizializzare un mutex:

```
fastmutex   = PTHREAD_MUTEX_INITIALIZER;
recmutex    = PTHREAD_RECURSIVE_MUTEX_INITIALIZER;
errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER;
```

  - dinamica, chiamata di libreria:

```
int pthread_mutex_init(pthread_mutex_t *mp,
                       const pthread_mutexattr_t *mattr);
```

    - `mp` è una mutex precedentemente allocato
    - `mattr` sono gli attributi del mutex: `NULL` per il default
    - restituisce sempre 0

Creare/Deallocare Attributi di Mutex

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

- crea in `attr` un attributo di mutex come quelli richiesti dalla `pthread_mutex_init()`

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

- dealloca l'attributo di mutex in `attr;`

- Entrambe restituiscono sempre 0

Attributi di Mutex: Tipo di Mutex

- Si può scegliere il tipo usando queste macro:
  - fast: `PTHREAD_MUTEX_NORMAL`
  - recursive: `PTHREAD_MUTEX_RECURSIVE`
  - error checking: `PTHREAD_MUTEX_ERRORCHECK`
    `PTHREAD_MUTEX_DEFAULT`

  e le funzioni per fissare/conoscere il tipo:

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

- restituisce 0 oppure un intero $\neq 0$ in caso di errore

```
int pthread_mutexattr_gettype(const  pthread_mutexattr_t *attr,
                              int *kind);
```

- restituisce sempre 0

Deallocazione di Mutex

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

- Dealloca tutte le risorse allocate per un certo mutex
- Controlla che il mutex sia sbloccato prima di eseguire effettivamente la deallocazione

lock()

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

- `lock()`: blocca un mutex
  - se era sbloccato il thread chiamante ne prende possesso bloccandolo immediatamente e la funzione ritorna subito
  - se era bloccato da un altro thread il thread chiamante viene sospeso sino a quando il possessore non lo rilascia
  - se era bloccato dallo stesso thread chiamante dipende dal tipo mutex
    - fast: stallo, perché il chiamante stesso, che possiede il mutex, viene sospeso in attesa di un rilascio che non avverrà mai
    - error checking: la chiamata fallisce
    - recursive: la chiamata ha successo, ritorna subito, incrementa il contatore del numero di lock eseguiti dal thread chiamante

unlock()

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

- `unlock()`: sblocca un mutex che si assume fosse bloccato. Ad ogni modo la semantica esatta dipende dal tipo di mutex
  - normal: il mutex viene lasciato sbloccato e la chiamata ha sempre successo
  - recursive: si decrementa il contatore del numero di lock eseguiti dal thread chiamante sul mutex, e lo si sblocca solamente se tale contatore si azzera
  - error checking: sblocca il mutex solo se al momento della chiamata era bloccato e posseduto dal thread chiamante, in tutti gli altri casi la chiamata fallisce senza alcun effetto sul mutex

Mutex: norace.c (1)

```
#include…
int myglobal;
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;

int main(void) {
  pthread_t mythread;
  int i;

  if (pthread_create(&mythread,NULL,thread_function,NULL)) {
    printf("creazione del thread fallita."); exit(1);  }
  for (i=0; i<20; i++) {
    pthread_mutex_lock(&mymutex);
    myglobal = myglobal+1;
    pthread_mutex_unlock(&mymutex);
    printf("o");
    fflush(stdout);
    sleep(1);
  }
  if (pthread_join (mythread,NULL)) {
    printf("errore nel join con il thread."); exit(2); }
  printf("\nmyglobal è uguale a %d\n",myglobal);
  exit(0); }…
```

Mutex: norace.c (2)

```
…
void *thread_function(void *arg) {
  int i,j;
  for ( i=0; i<20; i++ ) {
    pthread_mutex_lock(&mymutex);
    j=myglobal;
    j=j+1;
    printf(".");
    fflush(stdout);
    sleep(1);
    myglobal=j;
    pthread_mutex_unlock(&mymutex);
  }
  return NULL;
}
```

Accesso a Variabili Condivise con Mutex

Compilazione

```
$ gcc race.c -o race -lpthread
```

Esecuzione:

```
$ ./race
```

Possibile Output:

```
$ ./race
o..o..o.o..o..o.o.o.o.o..o..o..o.ooooooo
myglobal è uguale a 40
```

pthread_mutex_trylock()

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

- Versione non bloccante della `lock()`
- Si comporta esattamente come una `pthread_mutex_lock()`, tranne che nel caso di mutex già bloccato:
  - non blocca il thread chiamante
  - ritorna immediatamente con un errore EBUSY

Esercizi

Esercizio: un thread vuole comunicare una sequenza di interi ad un altro thread. Possono comunicare unicamente con un buffer condiviso capace di ospitare un solo intero. Sincronizzare gli accessi alla variabile di modo che la comunicazione avvenga evitando attese attive e fenomeni di interferenza (doppie letture, sovrascritture ...).

Quali difficoltà nascono utilizzando solo mutex per questo problema di cooperazione?

Pthread - Semafori

- `sem_init()`
  - inizializza un semaforo al valore dato
- `sem_destroy()`
  - dealloca tutte le risorse allocate per gestire il semaforo specificato
- `sem_wait()`
  - attende in attesa passiva che un semaforo assuma un valore non nullo; quindi lo decrementa
- `sem_post()`
  - incrementa il valore del semaforo specificato
- `sem_trywait()`
  - versione non bloccante della `sem_wait()`, nel caso che il semaforo sia rosso ritorna con un errore
- `sem_getvalue()`
  - restituisce il valore di un semaforo

Pthread - Semafori vs Semafori di Dijkstra

- I semafori POSIX corrispondono quasi esattamente alla versione dei semafori di Dijkstra precedentemente presentata
- Tuttavia si possono usare solo con i thread e non con i processi
- Lo standard prevede semafori condivisibili a due livelli
  - tra tutti i thread di un processo
  - tra diversi processi
    - (LinuxThreads supporta solo il primo tipo)

Esercizi

Esercizio: risolvere nuovamente l'esercizio del thread che vuole comunicare una sequenza di interi ad un altro thread comunicando tramite un buffer condiviso capace di ospitare un solo intero. Questa volta utilizzare i semafori per i thread POSIX per evitare fenomeni di interferenza.

Esercizio: scrivere un programma per calcolare il prodotto di due matrici di modo che ogni elemento del risultato sia calcolato da un thread distinto.

Esercizio: misurare lo speed-up del codice di cui sopra; quindi migliorarlo creando pochi thread (quanti?) che calcolano il risultato per interi sotto-blocchi della matrice risultante.

Thread in C: Cooperazione

- I principali strumenti per la risoluzione dei problemi di cooperazione sono le variabili condizione
- In sostanza, si tratta della realizzazione del concetto di variabile condizione ipotizzata da Hoare, anche se fuori dal contesto dei monitor

- Si possono anche utilizzare semafori per thread in stile cooperativo ma, in assenza di motivazioni per fare altrimenti, usare le variabili condizione chiarisce la natura della sincronizzazione che si sta cercando di gestire

Limiti dei Mutex come Strumento di Cooperazione (1)

- Se un thread attende il verificarsi di una condizione su una risorsa condivisa con altri thread
- Con i soli mutex sarebbe necessario un ciclo del tipo:

```
while(1) {
   lock(mutex);
   if (<condizione sulla risorsa condivisa>)
      break;
   unlock(mutex);
   …
}
<sezione critica>;
unlock(mutex);
```

Limiti dei Mutex come Strumento di Cooperazione (2)

- I mutex sono come strumento di cooperazione risultano:
  - inefficienti
  - ineleganti

  per risolvere elegantemente problemi di cooperazione servono altri strumenti
- Le variabili condizione sono un'implementazione delle variabili condizione teorizzate da Hoare
- Anche se fuori dal contesto dei Monitor, molte problematiche si ripresentano quasi immutate

Variabili Condizione

- strumenti di sincronizzazione tra thread che consentono di:
  - attendere passivamente il verificarsi di una condizione su una risorsa condivisa
  - segnalare il verificarsi di tale condizione
- la condizione interessa sempre e comunque una risorsa condivisa
- pertanto le variabili condizioni possono sempre associarsi al mutex della stessa per evitare corse critiche sul loro utilizzo

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

Variabili Condizione vs Primitive di Hoare

- Le variabili condizione POSIX si differenziano da quelle teorizzate da Hoare perché non inserite nel contesto dei monitor
  - bisogna allocare e gestire esplicitamente il mutex che garantisce contro eventuali corse critiche sulla risorsa interessata dalla condizione. Altrimenti:
    - un thread crea una var. cond. per una risorsa condivisa e si prepara ad attendere
    - un altro thread aggiorna lo stato della risorsa ed effettua la segnalazione sulla var. cond.
    - il primo thread ha perso la segnalazione prima ancora che riesca ad mettersi in attesa sulla condizione

- N.B. Si possono usare solo con i thread e non coi processi

Pthread - Variabili Condizione

- `pthread_cond_init()`
  - inizializza la variabile condizione specificata
- `pthread_cond_signal()`
  - riattiva uno dei thread in attesa sulla variabile condizione scelto non deterministicamente
- `pthread_cond_broadcast()`
  - riattiva tutti thread in attesa sulla variabile condizione
- `pthread_cond_wait()`
  - blocca il thread corrente che resta in attesa passiva di una signal sulla variabile condizione specificata
- `pthread_cond_destroy()`
  - dealloca tutte le risorse allocate per gestire la variabile condizione specificata

Inizializzazione Variabili Condizione

- Analogamente ai mutex:
  - inizializzazione statica

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

  - inizializzazione dinamica

```
#include <pthread.h>
int pthread_cond_init( pthread_cond_t  *cond,
                       pthread_condattr_t *cond_attr);
```

  - `cond` è la variabile condizione da inizializzare
  - `cond_attr` sono attributi della variabile condizione
    `NULL` per le impostazioni di default
- Esiste un solo tipo di variabile condizione

```
man 3 pthread_cond_init
```

Inizializzazione: varcond.c (1)

```
#include <pthread.h>
extern void do_work ();

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag() {
   // Inizializza mutex associato a variabile condizione
  pthread_mutex_init(&thread_flag_mutex,NULL);
  // Inizializza variabile condizione associata a flag
  pthread_cond_init(&thread_flag_cv,NULL);
  // Inizializza flag
  thread_flag = 0;
}…
```

Attesa: `pthread_cond_wait()`

```
#include <pthread.h>
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex);
```

- `cond` è una variabile condizione
- `mutex` è un mutex associato alla variabile

1. al momento della chiamata il mutex deve essere bloccato
2. rilascia il mutex, il thread chiamante rimane in attesa passiva di una segnalazione sulla variabile condizione
3. nel momento di una segnalazione, la chiamata restituisce il controllo al thread chiamante, e questo rientra in competizione per acquisire il mutex

- restituisce `0` in caso di successo oppure un codice d'errore $\neq 0$

Attesa: varcond.c (2)

```
…
/* Chiama do_work() mentre flag è settato, altrimenti si
   blocca in attesa che venga segnalato un cambiamento nel suo
   valore */

void* thread_function (void* thread_arg) {
  while (1) {
    // Attende segnale sulla variabile condizione
    pthread_mutex_lock(&thread_flag_mutex);
    while (!thread_flag)
      pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);
    pthread_mutex_unlock(&thread_flag_mutex);
    do_work ();          /* Fa qualcosa */
  }
  return NULL;
}…
```

Segnalazione: `pthread_cond_signal()`

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- uno dei thread che sono in attesa sulla variabile condizione `cond` viene risvegliato
  - se più thread sono in attesa, ne viene scelto uno ed uno solo effettuando una scelta non deterministica
  - se non ci sono thread in attesa, non accade nulla

Segnalazione: varcond.c (3)

```
…// Setta flag a FLAG_VALUE
void set_thread_flag (int flag_value) {
  // Lock del mutex su flag
  pthread_mutex_lock (&thread_flag_mutex);
  // cambia il valore del flag
  thread_flag = FLAG_VALUE;
  // segnala a chi è in attesa che
  // il valore di flag è cambiato
  pthread_cond_signal (&thread_flag_cv);
  // unlock del mutex
  pthread_mutex_unlock (&thread_flag_mutex);
}
```

pthread_cond_timedwait()

```
#include <pthread.h>
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            const struct timespec abstime);
```

- `cond` è una variabile condizione
- `mutex` è un mutex associato alla variabile
- `abstime` è la specifica di un tempo assoluto

- `pthread_cond_timedwait()` permette di restare in attesa fino all'istante specificato restituendo il codice di errore `ETIMEDOUT` al suo scadere
- restituisce `0` in caso di successo oppure un codice d'errore $\neq 0$

pthread_cond_broadcast()

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- causa la riparterza di tutti i thread che sono in attesa sulla variabile condizione `cond`
  - se non ci sono thread in attesa, non succede niente
- restituiscono `0` in caso di successo oppure un codice d'errore $\neq$`0`

pthread_cond_destroy()

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

- libera le risorse allocate per la variabile condizione specificata
- non devono esistere thread in attesa della condizione
- restituisce `0` in caso di successo oppure un codice d'errore $\neq 0$

Attributi delle Variabili Condizioni

- Analoghi agli attributi dei mutex
- LinuxThreads non supporta alcun tipo di attributo ed il secondo parametro di `pthread_cond_init()` è in effetti ignorato
- Fanno parte dello standard POSIX

```
#include <pthread.h>
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

queste funzioni non fanno nulla in LinuxThreads

Esercizi (Hand-Off)

Esercizio *(hand-off):* un thread vuole comunicare una sequenza di interi ad un altro thread. Possono comunicare unicamente con una singola variabile condivisa capace di ospitare un solo intero. Sincronizzare gli accessi alla variabile di modo che la comunicazione avvenga evitando attese attive e fenomeni di interferenza.

Esercizio *(testing hand-off):* utilizzando Cunit (*http://cunit.sourceforge.net/*), scrivere una serie di test di unità per fornire garanzie sulla corretta della soluzione dell'esercizio precendente

Esercizi

Esercizio: risolvere con i thread e le primitive wait/signal il problema di **`P`** produttori e **`C`** consumatori che si scambiano messaggi utilizzando un buffer circolare capace di contenere **`N`** messaggi.

Esercizi

Esercizio: riscrivere l'esercizio precedente prevedendo di generare in maniera casuale nuovi produttori. Questi producono per un intervallo di tempo casuale prima di terminare. Per controbilanciare l'aleatorietà della produzione, ed evitare che il buffer circolare si esaurisca, il sistema varia il numero di consumatori tra `C_MIN` e `C_MAX` in funzione del grado di occupazione del buffer. In particolare quando il buffer è vicino all'esaurimento si allocheranno nuovi consumatori, mentre quando è vicino allo svuotamento si termineranno alcuni di quelli esistenti.

Esercizio: risolvere il problema dei cinque filosofi mangiatori utilizzando i thread e le variabili condizioni.

Dati Privati di un Thread

- I thread condividono il segmento dati
- Complementarietà rispetto ai processi
  - thread
    - semplice scambiare dati con altri thread
    - appositi meccanismi per disporre di dati privati
      - TSD
  - processi
    - semplice disporre di dati privati del processo
    - appositi meccanismi per dialogare con altri processi
      - IPC

Thread Specific Data (TSD)

- Può servire disporre di dati che siano globalmente visibili all'interno di un singolo thread ma distinti da thread a thread
- Ogni thread possiede un'area di memoria privata, la TSD area, indicizzata da chiavi
- La TSD area contiene associazioni tra le chiavi ed un valore di tipo `void *`
  - diversi thread possono usare le stesse chiavi ma i valori associati variano di thread in thread
  - inizialmente tutte le chiavi sono associate a `NULL`

Funzioni per Thread Specific Data

- `int pthread_key_create()`
  - per creare una chiave TSD
- `int pthread_key_delete()`
  - per deallocare una chiave TSD
- `int pthread_setspecific()`
  - per associare un certo valore ad una chiave TSD
- `void *pthread_getspecific()`
  - per ottenere il valore associato ad una chiave TSD

TSD: pthread_key_create()

```
#include <pthread.h>
int pthread_key_create( pthread_key_t *key,
                        void (*destr_function) (void *));
```

- Alloca una nuova chiave TSD e la restituisce tramite `key`
- `destr_function`, se ≠ `NULL`, è il puntatore ad una funzione “distruttore” da chiamare quando il thread esegue `pthread_exit()`
- restituisce `0` in caso di successo oppure un codice d’errore ≠`0`

pthread_setspecific() & pthread_getspecific()

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key,
                        const void *pointer);
```

- Cambia il puntatore TSD associato ad una data `key` per il thread chiamante
- restituisce `0` in caso di successo oppure un codice d'errore $\neq 0$

```
void * pthread_getspecific(pthread_key_t key);
```

- Restituisce il puntatore TSD associato ad una data `key` per il thread chiamante oppure `NULL` in caso di errore

tsd.c (1)

```
#include …

static pthread_key_t thread_log_key; /* tsd key per thread */

void write_to_thread_log (const char* message); //Scrive log
void close_thread_log (void* thread_log); //Chiude file log
void* thread_function (void* args);       //Eseguita dai thread

int main() {
  // Crea una chiave da associare al log Thread-Specific
  // Crea 5 worker thread per il lavoro
  // Aspetta che tutti finiscano
  return 0;
}
…
```

tsd.c (2)

```
…
int main() {
  int i;
  pthread_t threads[5];
  // Crea una chiave da associare al puntatore TSD al log file
  pthread_key_create(&thread_log_key, close_thread_log);

  for (i = 0; i < 5; ++i) // worker thread
    pthread_create(&(threads[i]), NULL, thread_function, NULL);

  for (i = 0; i < 5; ++i) // Aspetta che tutti finiscano
    pthread_join (threads[i], NULL);
  return 0;
}
…
```

tsd.c (3)

```
…
void write_to_thread_log (const char* message) {
  FILE* thread_log = (FILE*)pthread_getspecific(thread_log_key);
  fprintf (thread_log, "%s\n", message);
}
void close_thread_log (void* thread_log) {
  fclose ((FILE*) thread_log);
}
void* thread_function (void* args) {
  char thread_log_filename[20];
  FILE* thread_log;
  sprintf(thread_log_filename,"thread%d.log",(int)pthread_self ());

  thread_log = fopen (thread_log_filename, "w");
  /* Associa la struttura FILE TSD a thread_log_key.  */
  pthread_setspecific (thread_log_key, thread_log);

  write_to_thread_log ("Thread starting.");
  /* Fai altro lavoro qui... */  return NULL;
}
```

Attributi di Thread

- Nello standard POSIX gli attributi di thread sono di tipo `pthread_attr_t`
- Creazione
  - `pthread_attr_init()`
- Deallocazione
  - `pthread_attr_destroy()`
- Impostazione, con funzioni di nome:
  - `pthread_attr_set`*attrname*
- Lettura, con funzioni di nome:
  - `pthread_attr_get`*attrname*
- Riguardano molti aspetti della vita di un thread: Scheduling, Contention, Detached state, Stack, Priorità
- Diversi non sono supportati da LinuxThreads

Creare / Deallocare Attributi di Thread

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

- restituisce tramite `attr` un attributo di thread inizializzato con valori di default

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

- dealloca l'attributo di thread in `attr`

- gli attributi di thread vengono di fatto utilizzati solo durante la creazione del thread
- cambiarli in seguito non comporta alcun effetto sui thread già creati

Impostare/Leggere Attributi di Thread

Consultare le man page

```
#include <pthread.h>
int pthread_attr_setdetachstate( pthread_attr_t *attr,
   int detachstate);
int  pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                 int *detachstate);

int pthread_attr_setschedpolicy( pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy( const pthread_attr_t *attr,
                                 int *policy);

int  pthread_attr_setschedparam( pthread_attr_t *attr,
                                 const struct sched_param *param);
int  pthread_attr_getschedparam( const pthread_attr_t *attr,
                                 struct sched_param *param);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inherit);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int  *inherit);

int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope)
```

Esercizi

Esercizio: risolvere il problema Produttori / Consumatori con buffer circolare di dimensione N utilizzando i thread in C sotto Linux. Prevedere un file di log distinto per ogni thread.