Sincronizzazione di thread POSIX

Contiene lucidi tratti da: 2006-2007 Marco Faella, Clemente Galdi, Giovanni Schmid (Università di Napoli Federico II), 2004-2005 Walter Crescenzi(Università di Roma 3).

Sincronizzazione

- I thread condividono la memoria
- Rischio di race condition, se accesso di più thread a stessi dati (e.g. lettura/scrittura di stesse variabili)
- Necessari meccanismi di sincronizzazione
  - mutex (semaforo binario)
  - condition variable

Sincronizzazione

- La sincronizzazione e' necessaria quando si accede a variabili condivise
  - Variabili/Strutture dati globali (statiche e dinamiche)
- Attenzione: TUTTE le operazioni possono essere NON atomiche
  - Dipende dall'architettura (se più accessi in mem. per una operazione, un thread si può inserire durante).
- E.g. “x++” puo' diventare:
  - Carica la variabile x in accumulatore
  - Incrementa l'accumulatore
  - Memorizza l'accumulatore

Esempio

```
typedef struct foo{ int a;   int b; } myfoo;

myfoo test; // Variabile GLOBALE

void *inc(void *arg){ // incremente a e b
   test.a++;
   test.b++;
   printf("tid=%d a=%d b=%d\n" , pthread_self(), test.a, test.b);
   pthread_exit((void *)&test);
}

int main(void){
   char st[100];
   pthread_t tid;
   int i=0;
   myfoo  *b;

   while (i++<10){
      pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
   }
   sleep(1);
}
```

Esempio: race

```
tid=1077283760 a=1 b=1
tid=1089891248 a=5 b=5
tid=1079385008 a=6 b=6
tid=1081486256 a=7 b=7
tid=1083587504 a=8 b=8
tid=1085688752 a=9 b=9
tid=1087790000 a=10 b=10
tid=1096194992 a=2 b=2
tid=1094093744 a=3 b=3
tid=1091992496 a=4 b=4
```

Esempio: race

```
int myglobal;
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;}

int main(void) {
  pthread_t mythread;  int i;
  if ( pthread_create( &mythread, NULL, thread_function, NULL) ) {
    printf("error creating thread.");    abort();  }
  for ( i=0; i<20; i++) {
    myglobal=myglobal+1;    printf("o");    fflush(stdout);
    sleep(1);  }
  if ( pthread_join ( mythread, NULL ) ) {
    printf("error joining thread.");    abort();  }
  printf("¥nmyglobal equals %d\n",myglobal);
```

Esempio: race

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
```

Mutex Posix

Un mutex Posix è caratterizzato dalle seguenti proprietà:

- è una variabile di tipo pthread_mutex_t che può essere inizializzata con diversi attributi (tipo, scopo, etc.)

- può assumere solo i due stati alternativi chiuso (locked) o aperto (unlocked);

- può essere chiuso solo da un processo alla volta, ed il processo che chiude il mutex ne diviene il possessore fino alla successiva chiusura;

- può essere riaperto solo dal proprio possessore;

- deve essere condiviso tra tutti i processi che intendono sincronizzare l'accesso ad una regione critica (blocco cooperativo).

I mutex

- Un mutex è un semaforo binario (rosso o verde)
- Un mutex è mantenuto in una struttura

```
pthread_mutex_t
```

- Tale struttura va allocata e inizializzata
- Per inizializzare:
  - se la struttura è allocata staticamente:

```
pthread_mutex_t c = PTHREAD_MUTEX_INITIALIZER
```

  - se la struttura è allocata dinamicamente (e.g. se si usa malloc): chiamare `pthread_mutex_init`

Inizializzare e distruggere un mutex

```
#include <pthred.h>
int pthread_mutex_init(                pthread_mutex_t     *mutex,
                    const       pthread_mutexattr_t *attr);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- inizializza e distrugge un mutex, rispettivamente
- Quando inizializzato e' in stato aperto
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

Usare i mutex

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

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

Esempio

```
myfoo test; // Variabile GLOBALE
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;

void *inc(void *arg){ // incremente a e b
    pthread_mutex_lock(&sem);
    test.a++;
    test.b++;
    printf("tid=%d a=%d b=%d\n" , pthread_self(), test.a, test.b);
    pthread_mutex_unlock(&sem);
    pthread_exit((void *)&test);
}

int main(void){
    char st[100];
    pthread_t tid;
    int i=0;
    myfoo  *b;

    while (i++<10){
        pthread_create(&tid, NULL, inc, NULL); // Thread concorrenti
    }
```

Esempio

tid=1077283760 a=1 b=1
tid=1096194992 a=2 b=2
tid=1079385008 a=3 b=3
tid=1081486256 a=4 b=4
tid=1083587504 a=5 b=5
tid=1085688752 a=6 b=6
tid=1087790000 a=7 b=7
tid=1094093744 a=8 b=8
tid=1091992496 a=9 b=9
tid=1089891248 a=10 b=10

Esempio

```
#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);
}…
```

Esempio

```
…
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;
}
```

Esecuzione:

```
$ ./race
```

Possibile Output:

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

Sincronizzazione

- La sincronizzazione puo' essere:
  - Per sezione critica
    - SOLO quando una struttura condivisa viene modificata in UN UNICO punto nel codice (esempio precedente)
    - E' sufficiente associare un mutex alla sezione critica
  - Per “struttura”
    - Quando la struttura puo' essere modificata in piu' punti nel codice
    - Utile se piu' strutture devono essere condivise contemporaneamente
    - E' necessario associare un mutex alla “struttura”

Esempio

```
typedef struct foo{
    int a;
    int b;
    pthread_mutex_t sem;
} myfoo;


myfoo *test; // Variabile GLOBALE

myfoo *init_struct(){
    struct foo *fp;

    if ((fp=malloc(sizeof(myfoo)))==NULL)
        return(NULL);
    fp->a=0;
    fp->b=0;
    pthread_mutex_init(&fp->sem,NULL);
    return(fp);

}
```

Esempio

```
void stampa(struct foo *test){
  printf("tid=%d a=%d b=%d\n", pthread_self(),test->a, test->b);
  fflush(stdout);
}

void *inc(void *arg){ // incremente a e b
    pthread_mutex_lock(&test->sem);
    test->a=test->a+2;
    test->b++; // Variabile GLOBALE
    stampa(test);
    pthread_mutex_unlock(&test->sem);
    pthread_exit((void *)&test);
}
```

Esempio

```
int main(void){
    char st[100];
    pthread_t tid;
    int i=0;
    myfoo  *b;

    test=init_struct();
    while (i++<10){
            pthread_create(&tid, NULL, inc, NULL); // Globale
    }
    sleep(1);
    printf("Master:");
    stampa(test);
    pthread_mutex_destroy(&test->sem);
}
```

Esempio

tid=1077283760 a=2 b=1
tid=1079385008 a=4 b=2
tid=1081486256 a=6 b=3
tid=1083587504 a=8 b=4
tid=1085688752 a=10 b=5
tid=1087790000 a=12 b=6
tid=1089891248 a=14 b=7
tid=1091992496 a=16 b=8
tid=1094093744 a=18 b=9
tid=1096194992 a=20 b=10
Master:tid=1075181248 a=20 b=10

Esempio

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

struct foo {
int f_count; pthread_mutex_lock; }

strcut foo * foo_alloc(void) {
 struct foo * fp;
 if ((fp = malloc(sizeof(struct foo)))) ! = NULL)
   fp-> f_count = 1;
   if (p_thread_mutex_init(&fp->lock, NULL)!=0){
      free(fp); return (NULL);
   }} return fp;}

void foo_hold(struct foo *fp){
   pthread_mutex_lock(&fp->f_lock); fp->f_count++;
  pthread_mutex_unlock(&fp->f_lock);}

void foo_rele (struct foo *fp){
   p_thread_mutex_lock(&fp->f_lock);
   if (--fp->f_count ==0) {
      p_thread_mutex_unlock(&fp->f_lock);
      p_thread_mutex_destroy(&fp->f_lock);
      free(fp);}
    else {p_thread_mutex_unlock(&fp->f_lock);}
}
```

Non si libera la mem. finchè tutti i processi non accedono

Ogni processo aggiorna il contatore in modalità esclusiva

Ogni processo rilascia in modalità protetta, se e' ultimo libera la memoria.

Attributi 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
- Attualmente LinuxThreads supporta solo l'attributo relativo al tipo di mutex

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

Tipologie di Mutex

- inizializzazione di un mutex
  - statica, macro per inizializzare un mutex:

```
fastmutex   = PTHREAD_MUTEX_INITIALIZER;
recmutex    = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
errchkmutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

  - 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

Lock per Tipologia

- `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 per Tipologia

- `unlock()`: sblocca un mutex che si assume fosse bloccato. Ad ogni modo la semantica esatta dipende dal tipo di mutex
  - fast: 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

Attributi Mutex

- Si può scegliere il tipo usando queste macro:
  - fast: PTHREAD_MUTEX_FAST_NP
  - recursive: PTHREAD_MUTEX_RECURSIVE_NP
  - error checking: PTHREAD_MUTEX_ERRORCHECK_NP

  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 ≠0 in caso di errore

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

- restituisce sempre 0

Deadlock

- Condizione di attesa ciclica
- Soluzione base: acquisire i mutex sempre nello stesso ordine
  - Non sempre possibile!
  - Puo' essere necessario utilizzare algoritmi specifici e pthread_mutex_trylock

Limiti del MUTEX

- 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);
```

Attesa, e verifica ciclica sulla var, finchè la condizione verificata non rompe il ciclo, quindi sblocco.

Limiti MUTEX

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

  e per risolvere elegantemente problemi di cooperazione servono altri strumenti
- Le variabili condizione sono un'implementazione delle variabili condizione teorizzate da Hoare

Una operazione attendi() su una variabile condizionale sospende un processo in una coda d'attesa per quella variabile condizionale, dando così via libera ad un nuovo processo che desidera entrare. L'operazione notifica() risveglia un processo sospeso sulla variabile condizionale; questo riprende l'esecuzione appena ha via libera

Variabili di 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 di Condizione

- Servono per attendere che una condizione si verifichi, escludendo race conditions (rendezvous tra thead)
- Utilizzata con i mutex: modificata dopo lock mutex e visibile dopo acqusizione del mutex
- Una condition variable è mantenuta in una struttura pthread_cond_t
- Tale struttura va allocata ed inizializzata
- Per inizializzare:
  - se la struttura è allocata staticamente:

pthread_cond_t c = PTHREAD_COND_INITIALIZER

  - se la struttura è allocata dinamicamente: chiamare pthread_cond_init

Inizializzare e distruggere una condition variable

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

int pthread_cond_destroy(pthread_cond_t *cond);
```

- inizializza e distrugge una condition variable, rispettivamente
- Per la destroy non devono esistere thread in attesa
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

Inizializzazione

```
#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;
}
```

Usare una condition variable

- thread che aspetta una condizione

mutex_lock(m)

while (condizione falsa)

    cond_wait(c, m)

fa' qualcosa

mutex_unlock(m)

- thread che rende la condizione vera

mutex_lock(m)

rendi la condizione vera

cond_broadcast(c)

mutex_unlock(m)

Attendere una condition variable

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

- attende che cond sia segnalata come vera
- restituisce 0 se OK, un codice d'errore altrimenti
- il mutex protegge la condizione
  - deve essere acquisito prima di chiamare cond_wait
  - durante l'attesa, cond_wait rilascia il mutex
  - finita l'attesa, cond_wait riprende il mutex

Attendere un variabile di condizione

```
#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 $\neq 0$

Esempio

```
…
/* 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;
}
```

Inizializzare e distruggere una condition variable

int **pthread_cond_signal**(pthread_cond_t ***cond);**

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

- signal risveglia esattamente un thread in attesa su una condition variable
- broadcast risveglia tutti i thread in attesa su una condition variable
- restituiscono 0 se OK, un codice d'errore altrimenti
- attr può essere NULL (attributi di default)

Segnalazione

```
#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

```
// 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
```

Timed Wait & Broadcast

```
#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 ≠0

```
#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 ≠0

Esempio

```
pthread_mutex_t sem=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;

void *dec(void *arg){

   while(1){
      pthread_mutex_lock(&sem);
      while (test.a==0)
         pthread_cond_wait(&cond, &sem);

      printf("CONSUMATORE 1 a=%d \n", test.a);
      test.a--;
      pthread_mutex_unlock(&sem);
      nanosleep(&t2,NULL);
   }
}
```

Esempio

```
void *dec2(void *arg){

   while(1){
      pthread_mutex_lock(&sem);
      while (test.a==0)
         pthread_cond_wait(&cond, &sem);

      printf("CONSUMATORE 2 a=%d, b=%d \n", test.a, test.b);
      test.a--;
      test.b--;
      pthread_mutex_unlock(&sem);
      nanosleep(&t2,NULL);
   }
}
```

Esempio

```
void *inc(void *arg){ // incremente a e b

    int j=0;
    while (j++<10){
        pthread_mutex_lock(&sem);
        test.a++;
        test.b++;
        printf("PRODUTTORE tid=%d a=%d b=%d\n", pthread_self(),test.a, test.b);

        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&sem);
        nanosleep(&t2,NULL);
    }
    pthread_exit((void *)&test);
}
int main(void){
    pthread_t tid;
    pthread_create(&tid, NULL, inc, NULL);
    pthread_create(&tid, NULL, dec, NULL);
    pthread_create(&tid, NULL, dec2,NULL);
    sleep(5);
```

Esempio

PRODUTTORE tid=1077283760 a=1 b=1
CONSUMATORE 1 a=1
PRODUTTORE tid=1077283760 a=1 b=2
CONSUMATORE 1 a=1
PRODUTTORE tid=1077283760 a=1 b=3
CONSUMATORE 1 a=1
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 2 a=1, b=4
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 2 a=1, b=4
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 2 a=1, b=4
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 2 a=1, b=4
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 2 a=1, b=4
PRODUTTORE tid=1077283760 a=1 b=4
CONSUMATORE 1 a=1
PRODUTTORE tid=1077283760 a=1 b=5
CONSUMATORE 1 a=1

Attributi

- Analoghi agli attributi dei mutex
- Attualmente 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

Dati Privati di un thread

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

Dati specifici di un thread

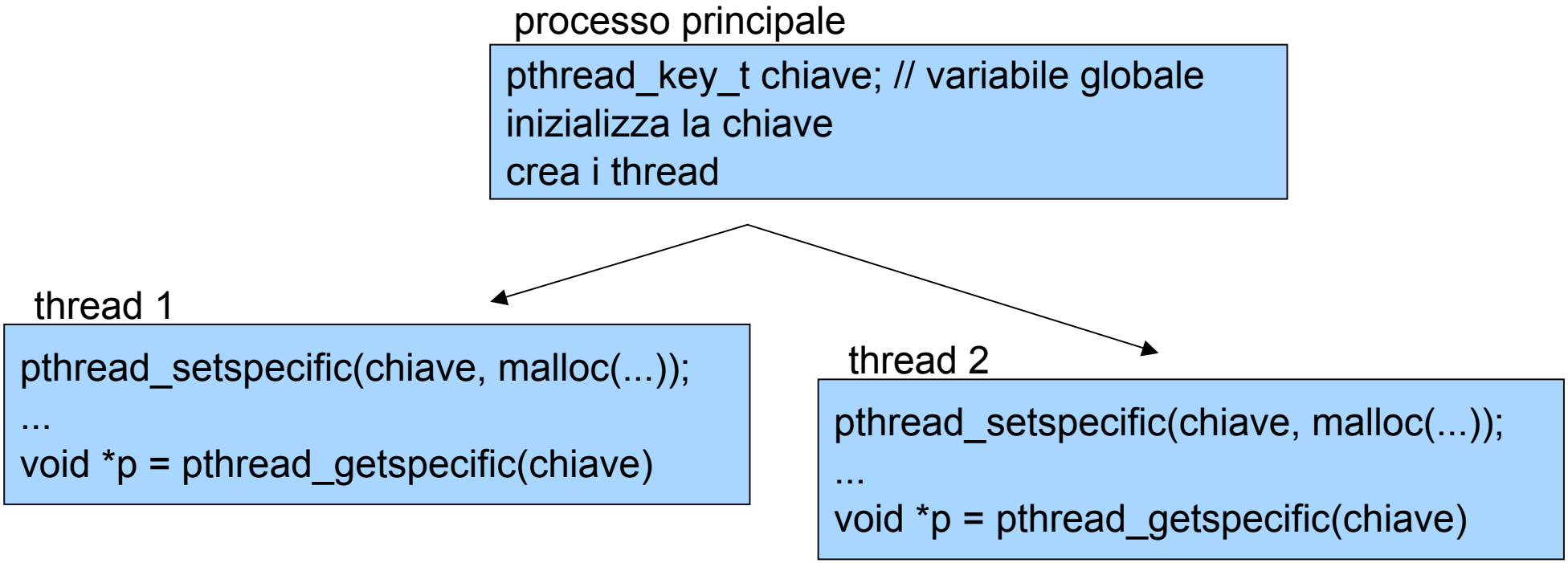
- Come fa un thread ad avere dati globali?
- Non puo' usare una variabile globale, perche' condivisa
  - ci vorrebbe una variabile globale per ogni thread: complicato
- Deve usare una variabile locale, che viene passata a tutte le funzioni chiamate dal thread
- Oppure...*thread-specific data*

Thread Specific Data e Chiavi

- 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

Thread-specific data

- associare a una stessa chiave, dati diversi per ciascun thread



Funzioni per TSD

- int pthread_key_create(…)
  - per creare una chiaveTSD
- 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

Creare una chiave

int **pthread_key_create**(pthread_key_t *key,
void (*destructor)(void *));

- crea una chiave per dati privati
- key è l'indirizzo della chiave da inizializzare
- destructor è un puntatore alla funzione distruttore che deve essere chiamata alla terminazione di un thread (pthread_exit())
- restituisce 0 se OK, un codice d'errore altrimenti

Usare una chiave

int **pthread_setspecific**(pthread_key_t ***key**,
const void* **val**);

- associa l'indirizzo val alla chiave key, per il thread chiamante
- restituisce 0 se OK, un codice d'errore altrimenti

---

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

- restituisce l'indirizzo associato alla chiave key nel thread chiamante
  - restituisce NULL se nessun indirizzo è stato associato a key

Esempio

```
#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 thread che facciano il lavoro
   // Aspetta che tutti finiscano
   return 0;
}
…
```

Esempio

```
…
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) // thread che faccia il lavoro
    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;
}
```

Esempio

```
…
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;
```

Esercizio

- realizzare un programma che accetta da riga di comando due numeri interi n ed m, e crea n *produttori* ed m *consumatori*
- produttori e consumatori condividono un array di 100 interi
- ogni produttore aspetta un numero casuale di secondi tra 1 e 10, e poi produce (cioe' inserisce nell'array) da 1 a 5 numeri casuali. Se il produttore trova l'array pieno, salta il turno
- ogni consumatore aspetta che ci sia un numero da consumare, e poi stampa a video il proprio tid e il valore consumato