POSIX Threads

E.Mumolo, DEEI
mumolo@units.it

Thread

- Processo=entità computazionale, dotato di
  - Identificatori del processo
  - Risorse e limiti (memoria, disco, tempo di CPU…),
  - Spazio di indirizzamento virtuale.
- Thread=entità computazionali che condividono
  - Lo spazio di indirizzamento
  - Le risorse
  - I limiti

… del processo

Implementazione dei thread

- Naturalmente sono forniti dalla maggioranza dei sistemi operativi
- Ma ciascuno ha una diversa interfaccia per l'uso delle funzioni associate ai thread
- IEEE Portable Operating System Interface (POSIX) 1003.1c è la parte di POSIX che riguarda i thread.
- Definisce le funzioni e le interfacce di programmazione alle applicazioni per la gestione e sincronizzazione dei thread
- POSIX è una definizione, non una realizzazione

Implementazione dei thread

- Thread in Solaris 2:
  - A livello utente: contiene identificatore, registri, pila, priorità
  - A livello del nucleo: copie dei registri del nucleo, puntatore a LWP, informazioni di scheduling e priorità, una pila
  - A livello intermedio (LWP): contiene un insieme di registri relativi ai thread che eseguono, risorse di memoria e informazioni di contabilizzazione
- Thread in Windows 2000:
  - Modello uno a uno
  - Contiene: identificativo, insieme di registri, pila utente, pila nucleo
- Thread in Linux:
  - Presente dalla versione 2.2
  - Chiamata di sistema clone: processo distinto che condivide lo spazio di indirizzi del processo chiamante
  - Il sistema non distingue tra processi e thread → task

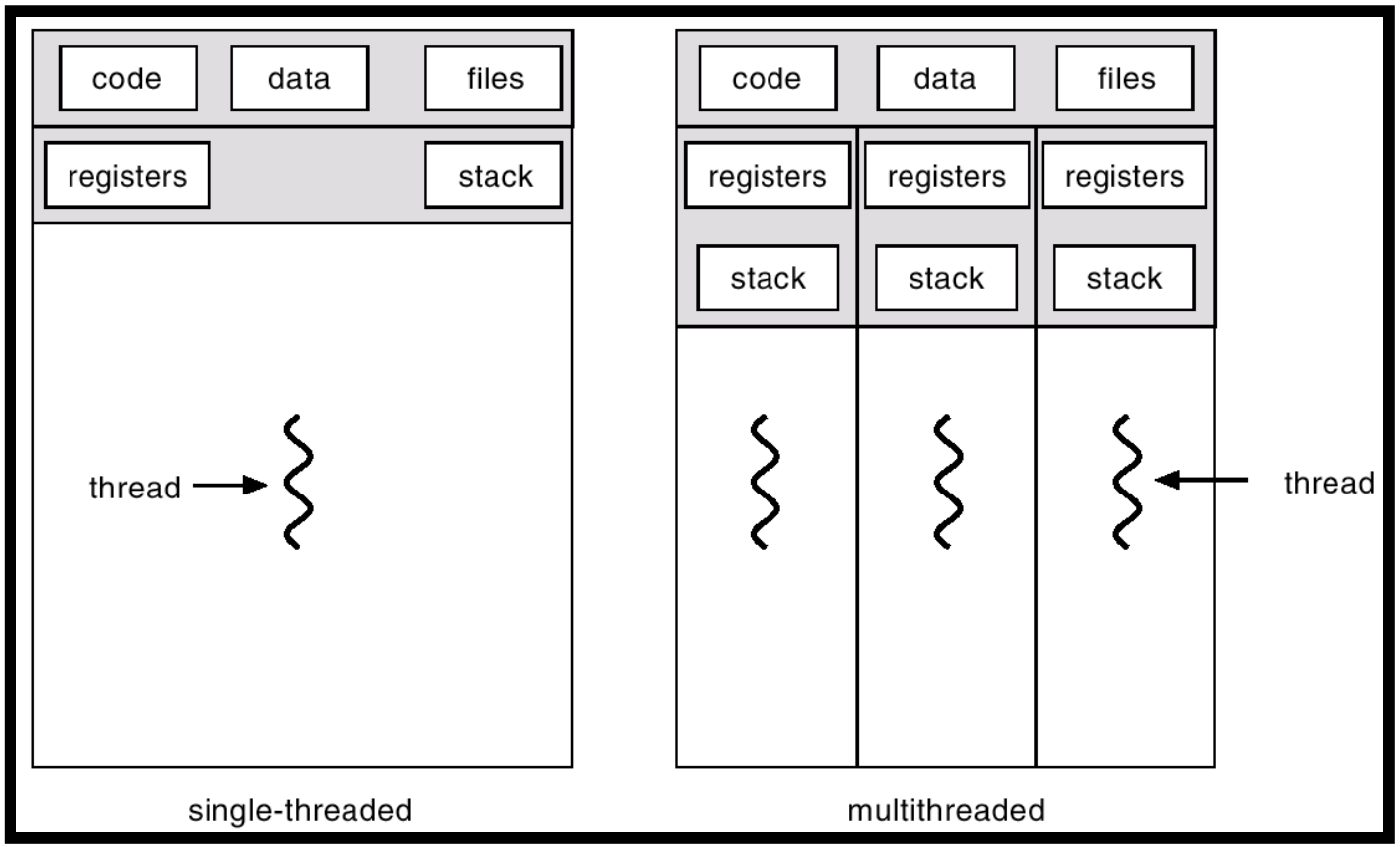
Implementazione dei pthread a livello utente

- I pthread a livello utente comprendono:
  - Uno schedulatore che esegue a livello utente, che schedula i thread in un singolo processo; ce n'è uno in ogni processo che organizzato a thread
  - Lo schedulatore del sistema operativo, che schedula ogni processo.
- Non richiedono cambiamenti a livello del kernel
  - → maggiore velocità
  - → Context switch più efficiente
- Ma:
  - Più Thread nello stesso processo non possono eseguire parallelemente in un sistema con più CPU
  - Mappatura molti a uno.

Implementazione dei pthread a livello kernel

- Mappatura uno a uno.
- Molte delle informazioni che devono essere mantenute nel kernel sono analoghe a quelle che devono essere usate per un processo singolo
- In questo caso più thread possono eseguire in parallelo su più CPU
- Ma:
  - Overhead più alto (analogo a quello dei processi)
  - Se ci sono molti thread a livello kernel, le prestazioni del sistema possono degenerare

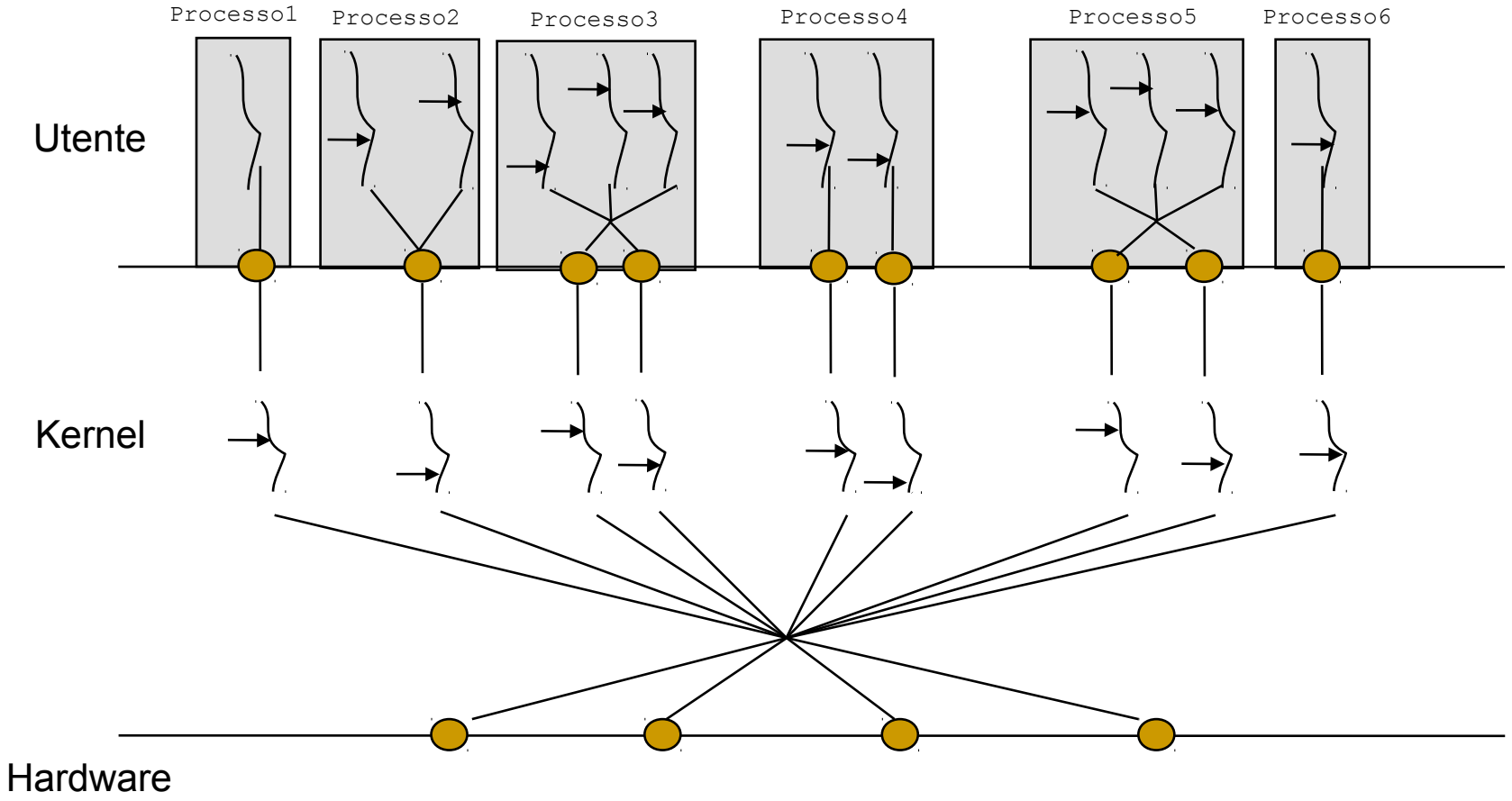
Modello implementativo dei Thread



Modelli di implementazione dei thread

- Due modelli principali:
  - Thread a livello utente – più veloci da creare, con CS più veloce dei precedenti. Svantaggio: con un kernel a singolo thread, se un thread esegue una system call bloccante, tutti gli altri thread che condividono lo stesso spazio di indirizzamento si bloccano.
  - Thread a livello kernel – il kernel controlla i singoli thread e li schedula indipendentemente. Vantaggio: anche se il kernel è a singolo thread, se un thread esegue una system call bloccante gli altri thread non si bloccano.
  - Metodi misti:
    - Da molti a uno (Solaris2)
    - Da uno a uno (Linux, WinNT, Win2000, …)
    - Da molti a molti (Solaris2, Irix, HP-UX,…)
- Nel caso da molti a molti le chiamate di sistema sono disponibili tramite “wrappers” alle system calls bloccanti: prima che si blocchino l’esecuzione è passata ad un’altro thread

Architettura



Alcune chiamate della libreria pthread

pthread_create

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

- Questa funzione viene usata per creare un nuovo thread. Argomenti:
  - Puntatore all'identificatore di tipo *pthread_t,* che è una struttura che contiene cose tipo: Dimensione - Versione - SP - Stack - Codice …
  - Puntatore alla struttura delle caratteristiche del thread (*thread attribute object*). NULL = caratteristiche di default
  - Puntatore alla funzione che comincerà l'esecuzione del thread con il nome specificato dall'argomento *start_routine*,
  - Puntatore al parametro (*arg*) passato alla funzione.
  - In caso d'errore ritorna un codice d'errore (>0). Se ha successo ritona zero

Attributi dei thread

```
typedef struct
 {
        int  detachstate;
        int  schedpolicy;
        struct   sched_param   schedparam;
        int  inheritsched;
        int  scope;
        size_t guardsize;
        int  stackaddr_set;
        void *  stackaddr;
        size_t  stacksize;
 }pthread_attr_t;
```

Caratteristiche dei pthread

- Quando viene specificato un parametro NULL in molte funzioni della libreria pthread, vengono fornite le caratteristiche di default.
- In qualche caso possono essere richieste delle caratteristiche particolari.
- In questo caso, deve essere creata una variabile particolare
- Creazione e inizializzazione delle caratteristiche:

```
pthread_attr_t my_attributes;
pthread_attr_init(&my_attributes);
```

  - Inizializza un elemento 'attributi' a default

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

  - Elimina un elemento 'attributi'

Modifica delle caratteristiche dei thread

- int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
  - PTHREAD_CREATE_DETACHED, PTHREAD_CREATE_JOINABLE
  - Libera o meno le risorse quando termina
- int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
  - Specifica la posizione dello stack: buffer di dimensione oppportuna
- int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
  - Minima stack size (in bytes) allocata quando viene creato
- int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
  - SCHED_FIFO, SCHED_RR, SCHED_OTHER
- int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
  - PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED
  - Usa i parametri di schedpolicy o meno
- int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
  - PTHREAD_SCOPE_SYSTEM, PTHREAD_SCOPE_PROCESS
  - Crea unbound o bound threads

pthread_detach

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

- Questa funzione viene usata per informare la libreria pthread che non si aspetterà la terminazione del thread creato. Cioè, la struttura dati interna usata dal thread (identificatore, stack, attributi, registri) verrà riutilizzata una volta terminato il thread.
- 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 ≠0
- un thread può anche essere creato direttamente detached specificando opportunamente il campo relativo agli attributi del thread creato nella funzione pthread_create()

pthread_equal, pthread_self

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

- Questa funzione confronta i due identificatori di thread. Se sono uguali ritorna un numero maggiore di zero, altrimenti ritorna zero

```
#include <pthread.h>
phread_t pthread_self(void);
```

- Ottiene l'identificatore del thread chiamante.

pthread_join, pthread_exit, pthread_kill

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

- Questa funzione viene usata per bloccare il thread chiamante finchè il thread specificato con “thread” termina (o è già terminato). Il puntatore “value_ptr” è usato per recuperare ogni valore d’uscita fornito dal thread terminato mediante la funzione pthread_exit.

```
int pthread_exit(void *value);
```

- Termina il thread tornando value ad ogni thread di cui si stava aspettando la terminazione con pthread_join

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

- Manda un segnale al thread specificato

Un primo esempio

```
//1o thread
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
void *thread_function1(void *arg) {
  int i;
  for ( i=0; i<5; i++ ) {
    printf("Sono il thread1!\n");
    sleep(1);
  }
  return NULL;
}
void *thread_function2(void *arg) {
  int i;
  for ( i=0; i<5; i++ ) {
    printf("Sono il thread2!\n");
    sleep(1);
  }
  return NULL;
}
```

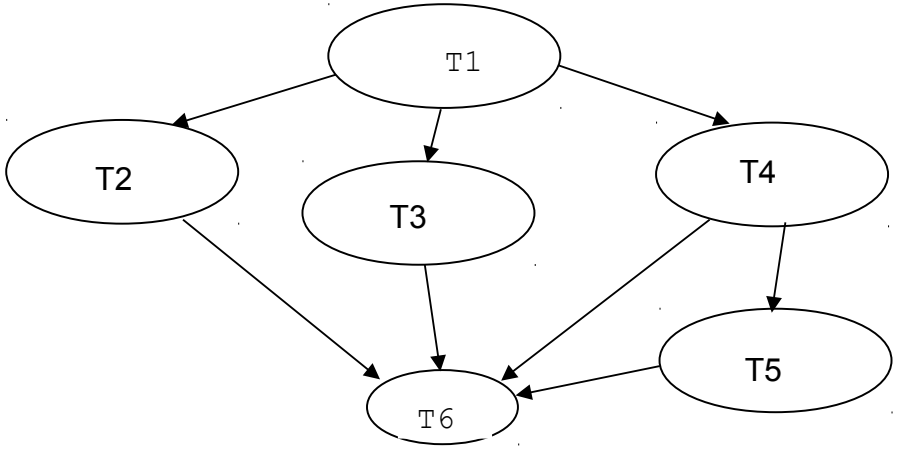
Un primo esempio (cont.)

```
int main(void) {
  pthread_t mythread1, mythread2;
  if ( pthread_create( &mythread1, NULL, thread_function1, NULL) ) {
    printf("errore di creazione thread.");
    abort();
  }
  if ( pthread_create( &mythread2, NULL, thread_function2, NULL) ) {
    printf("errore di creazione thread.");
    abort();
  }
  if ( pthread_join ( mythread1, NULL ) ) {
    printf("errore di sincronizzazione thread.");
    abort();
  }
  if ( pthread_join ( mythread2, NULL ) ) {
    printf("errore di sincronizzazione thread.");
    abort();
  }
  exit(0);
}
```

Un secondo esempio: passare parametri ai thread

```
#include <pthread.h>
#include <stdio.h>
struct char_print_parms
{
  char character; //  i caratteri da stampare
  int count;  // questo numero di volte
};
void* char_print (void* parameters) // stampa su stderr
{
  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;
}
int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args); //stampa 30000 x
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args); //stampa 20000 o
}
```

Esempio: diagrammi Data Flow




```
...
pthread_create(&th1,NULL,T1,0); pthread_join(th1, NULL);
pthread_create(&th2,NULL,T2,0); pthread_create(&th3,NULL,T3,0); pthread_create(&th4,NULL,T4,0);
pthread_join(th4, NULL); pthread_create(&th5,NULL,T5,0);
pthread_join(th2, NULL); pthread_join(th3, NULL); pthread_join(th5, NULL);
pthread_create(&th6,NULL,T6,0); pthread_join(th6, NULL);
...
```

```c
/* altro esempio di creazione di thread  */
#include <stdio.h>
#include <pthread.h>
pthread_t main_id, mythread_id;
void *body(void *arg)
{
  int p = *(int *)arg;
  printf("mythread: parameter=%d\n", p);  mythread_id = pthread_self();
  printf("mythread: main_id==mythread_id:%d\n", pthread_equal(main_id, mythread_id) );
  return (void *)5678;
}
int main()
{
  pthread_t mythread;
  pthread_attr_t myattr;
  int err,  parameter;
  void *returnvalue;
  parameter = 1234;
  /* inizializza gli attributi del thread */
  pthread_attr_init(&myattr);   puts("main: prima della creazione\n");
  main_id = pthread_self();
  /* creazione e esecuzione del nuovo thread */
  err = pthread_create(&mythread, &myattr, body, (void *)&parameter);
  puts("main: dopo la creazione del thread\n");
  /* l'oggetto degli attributi del thread non è più necessario */
  pthread_attr_destroy(&myattr);
  /* attendi la terminazione del thread appena creato */
  pthread_join(mythread, &returnvalue);
  printf("main: returnvalue is %d\n", (int)returnvalue);
  return 0;
}
```

```
/*
 *  schedulazioni dei thread
 */
#include <stdio.h>
#include <pthread.h>
#include <sched.h>

void *low(void *arg)
{
  printf("thread a BASSA priorità!!!\n");
  return NULL;
}

void *medium(void *arg)
{
  int i,j;

  for (i=0; i<300; i++) {
     for (j=0; j<1000000; j++) ;
     printf((char *)arg);
  }

  return NULL;
}
```

```
void my_create(int policy)
{
  pthread_t th1, th2, th3;
  pthread_attr_t medium_attr, low_attr;
  struct sched_param medium_policy, low_policy;

  pthread_attr_init(&medium_attr);
  pthread_attr_setschedpolicy(&medium_attr, policy);
  medium_policy.sched_priority = 2;
  pthread_attr_setschedparam(&medium_attr, &medium_policy);

  pthread_attr_init(&low_attr);
  pthread_attr_setschedpolicy(&low_attr, SCHED_FIFO);
  low_policy.sched_priority = 1;
  pthread_attr_setschedparam(&low_attr, &low_policy);

  pthread_create(&th1, &medium_attr, medium, (char *)".");
  pthread_create(&th2, &medium_attr, medium, (char *)"#");
  pthread_create(&th3, &low_attr, low, NULL);

  pthread_attr_destroy(&medium_attr);
  pthread_attr_destroy(&low_attr);

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

void *high(void *arg)       //primo esperimento: due thread a priorità media schedulati con RR
{                           //secondo esperimento: un thread a priorità bassa schedulati con FIFO
   my_create(SCHED_RR);
   my_create(SCHED_FIFO);   //primo esperimento: due thread a priorità media schedulati con FIFO
                            //secondo esperimento: un thread a priorità bassa schedulati con FIFO
   return NULL;
}
```

```
int main()
{
  pthread_t mythread;
  pthread_attr_t myattr;
  struct sched_param myparam;

  int err;
  int parameter;
  void *returnvalue;

  /* inizializza gli attributi dei thread */
  pthread_attr_init(&myattr);
  pthread_attr_setschedpolicy(&myattr, SCHED_FIFO);
  myparam.sched_priority = 3;
  pthread_attr_setschedparam(&myattr, &myparam);

  err = pthread_create(&mythread, &myattr, high, (void *)&parameter);

  if (err) {
    perror("ERRORE");
    exit(1);
  }

  pthread_attr_destroy(&myattr);

  /* attendi la fine del thread appena creato */
  pthread_join(mythread, &returnvalue);

  return 0;
}
```

Strumenti per la sincronizzazione in POSIX

- Gli strumenti principali per effettuare sincronizzazione tra threads sono:
  - Mutex
  - Pthread_join
  - Pthread_once
  - Variabili condizione
  - Semafori contatori

Mutex

- Un mutex è un semaforo binario. Cioè, un mutex viene usato per controllare una risorsa che può essere usata da al più un thread alla volta.

- `pthread_mutex_t amutex = THREAD_MUTEX_INITIALIZER;`

→ Crea un mutex

- `pthread_mutex_t amutex = THREAD_RECURSIVE_MUTEX_INITIALIZER;`

→ Crea un mutex

Caratteristiche del mutex

- Supporta solo una associazione attributo-tipo del mutex

```
#include <pthread.h>
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int kind);
int  pthread_mutexattr_gettype(const  pthread_mutexattr_t  *attr,   int
   *tipo);
```

- *Tipo* può essere:
  - PTHREAD_MUTEX_FAST_NP
  - PTHREAD_MUTEX_RECURSIZE_NP
    - Si può bloccare un mutex già bloccato
  - PTHREAD_MUTEX_ERRORCHECK_NP
    - Riporta errori: lock un mutex locked, unlock un mutex unlocked

pthread_mutex_init

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
pthread_mutex_attr *attr);
```

- Inizializza un mutex con gli attributi specificati nel mutex specificato con il parametro attr. Se attr è NULL, vengono usati gli attributi di default.

pthread_mutex_lock

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

- Usato per bloccare il mutex specificato. Se già bloccato, allora il thread chiamante si blocca finchè il mutex non si sblocca.

pthread_mutex_unlock

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

- Usato per sbloccare il mutex specificato. Per un mutex ricorsivo che è stato bloccato altre volte, solo l'ultimo sblocco rilascia il mutex per essere usato da altri thread. Se altri thread sono bloccati nell'attesa del mutex, il thread in attesa con priorità più alta viene sbloccato e diviene il proprietario del mutex.

Esempio di sincronizzazione con mutex

```
#include <pthread.h>
int a=1; b=1;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void thread1(int *arg)
{
    pthread_mutex_lock(&m);
    printf("sono il primo thread. Parametro = %d \n", *arg);
    a=a+1; b=b+1;
    pthread_mutex_unlock(&m);
}
void thread2(int *arg)
{
    Pthread_mutex_lock(&m);
    printf("sono il secondo thread. Parametro = %d \n", *arg);
    b=b*2; a=a*2;
    pthread_mutex_unlock(&m);
}
main()
{
    pthread_t th1, th2;
    int i1 = 1, i2=2;

    pthread_create(&th1, NULL, (void *)thread1, (void *)&i1);
    pthread_create(&th2, NULL, (void *)thread2, (void *)&i2);

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

    printf("Valori finali: a=%d  b=%d\n", a,b);

}
```

- pthread_mutex_t → è una struttura che contiene cose tipo:
  - Nome del mutex
  - proprietario
  - contatore
  - Struttura associata al mutex
  - … e simili

pthread_once

```
#include <pthread.h>
int pthread_once(  phread_once_t *once_block,
                   void (*init_routine) (void));
```

- Assicura che la routine di inizializzazione *init_routine* eseguirà solo una volta senza curarsi di quanti thread nel processo la chiamano. Tutti i threads chiamano la routine facendo identiche chiamate alla funzione *pthread_once* (con la stessa routine *once_block* e *init_routine*). Il thread che per primo chiama la funzione *pthread_once* può eseguirla; le chiamate seguenti non eseguono la funzione.

Esempio

```
#include <pthread.h>
pthread_once_t init=PTHREAD_ONCE_INIT;
void init_funz()
{
//varie inizializzazioni…
  printf("inizializzazione effettuata\n");
}
void funzione()
{
    (void)pthread_once(&init,init_funz);
//…istruzioni
   printf("funzione\n");
}
main()
{
    pthread_t t1, t2;
    int r;
    r=pthread_create(&t1, NULL, (void *) funzione), NULL);
    r=pthread_create(&t1, NULL, (void *) funzione), NULL);
}
```

pthread_mutex_trylock

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

- Cerca di bloccare un mutex. Se il mutex è già bloccato, il thread chiamante ritorna un errore EBUSY senza aspettare che il mutex si liberi.
- In pseudo-codice:

```
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
void thr(void *arg)
{
    while(true){
    r=pthread_mutex_trylock(&m);
    if(r!=EBUSY){
          pthread_mutex_unlock(&m); break;
    }else{
          incrementa un contatore;
    }
    }
}
```

pthread_mutex_destroy

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

- Elimina un mutex

```c
/* esempio di utilizzo dei Mutex  */
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
pthread_mutex_t mymutex;
void *body(void *arg)
{
     int i,j;
     for (j=0; j<40; j++) {
       pthread_mutex_lock(&mymutex);
        for (i=0; i<1000000; i++);
        fprintf(stderr,(char *)arg);
       pthread_mutex_unlock(&mymutex);
     }
     return NULL;
}

int main()
{
     pthread_t t1,t2,t3;
     pthread_attr_t myattr;
     int err;
     pthread_mutexattr_t mymutexattr;
     pthread_mutexattr_init(&mymutexattr);
     pthread_mutex_init(&mymutex, &mymutexattr);
     pthread_mutexattr_destroy(&mymutexattr);
     pthread_attr_init(&myattr);
     err = pthread_create(&t1, &myattr, body, (void *)".");
     err = pthread_create(&t2, &myattr, body, (void *)"#");
     err = pthread_create(&t3, &myattr, body, (void *)"o");
     pthread_attr_destroy(&myattr);  pthread_join(t1, NULL);  pthread_join(t2, NULL);  pthread_join(t3, NULL);
     printf("\n");
     return 0;
}
```

Blocco di un mutex ricorsivo

- Un thread che cerca di bloccare un mutex non ricorsivo che possiede già riceverà una indicazione di stallo e il tentativo di bloccare il mutex fallisce.

- L'utilizzo di un mutex ricorsivo evita questo problema, ma il thread deve assicurare di sbloccare il mutex il numero giusto di volte. Altrimenti nessun altro thread sarà in grado di bloccare il mutex.

Variabili condizione

- Pseudocodice dei monitor per produttore/consumatore

```
monitor prodcons
   condition pieno, vuoto;
   void Prod() {
      while(true){
         el=produci();
         if(n==N) pieno.wait; //se il buffer e' pieno, aspetta
         A[n]=el; n++;
         while(n==1) vuoto.signal; //1 elemento: sveglia il consumatore
      }
    }
    void Cons() {
         while(true){
            if(n==0) vuoto.wait; //se il buffer e' vuoto, aspetta
            el=A[n]; n--;
            while(n==N-1) pieno.signal; //1 posto libero: sveglia il prod
         }
    }
end monitor;
```

Variabili condizione

- Una variabile condizione – nei monitor – è un oggetto di sincronizzazione sul quale un processo può attendere – continuando l'esecuzione all'esterno del monitor - finchè un altro processo lo sveglia.

- Con thread POSIX, una variabile condizione viene usata assieme ad un mutex. Se l'esecuzione è all'interno del monitor, il mutex è bloccato. Se necessario un thread attende sulla variabile condizione, che sblocca il mutex, consentendo ad altri thread l'ingresso alla sezione critica.

- Più tardi, quando le condizioni lo permettono, un thread sveglia la variabile condizione, sbloccando il thread a priorità più alta che attende sulla variabile.

Inizializzazione di una variabile condizione

- Una variabile condizione è creata e inizializzata mediante un codice del tipo:

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

dove *uvc* è il nome di una variabile condizione

Attesa su una variabile condizione

- Per attendere su una variabile condizione, un thread deve aver già bloccato un mutex. Quindi esegue:

```
#include <pthread.h>
pthread_cond_wait(&a_c_v,&a_mutex);
```

- Il mutex viene sbloccato e il thread chiamante si blocca sulla variabile condizione.

- Quando si torna dalla funzione, il mutex sarà nuovamente bloccato, e sarà di proprietà del thread chiamante

→ Non utilizzare mutex ricorsivi con questa funzione

Risveglio di una variabile condizione

- Una variabile condizione può essere svegliata in due modi:
  - Mediante una chiamata alla funzione

```
pthread_cond_signal (pthread_cond_t *cond)
```

che sblocca il thread a priorità più alta che è in attesa da più tempo.

  - Oppure mediante una chiamata alla fiìunzione

```
pthread_cond_broadcast (pthread_cond_t *cond)
```

che sblocca tutti i thread in ordine di priorità, usando un ordinamento FIFO per I thread con la stessa priorità.

- Un thread può anche utilizzare la funzione `pthread_cond_timedwait`

per attendere su una variabile condizione; un valore assoluto di tempo viene fornito per sbloccare il processo se il tempo assoluto viene superato.

Primo esempio di variabili condizione

- Si considerino due variabili condivise, `x` e `y`, un mutex `m` e una variabile condizione che viene svegliata quando `x>y`

```
int x=0; int y=10;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t cond = PTHREAD_COND_INITIALIZER;


{ //...primo thread...
pthread_mutex_lock(&m);
while (x<=y) pthread_cond_wait(&cond, &m);
printf("x=%d, y=%d", x, y); //qualche utilizzo delle variabili
pthread_mutex_unlock(&m);
}


{ //...secondo thread...
 for(int i=0;i++;i<max){
    pthread_mutex_lock(&m);
    x++; y--; //qualche elaborazione delle variabili
    if (x>y) pthread_cond_signal(&cond);
    pthread_mutex_unlock(&m);
 }
}
```

Approcci da seguire dopo il risveglio

- Dopo che un thread ha svegliato la variabile condizione, ci sono due approcci che vengono generalmente seguiti:
  - Il thread esce immediatamente dal monitor (sbloccando il mutex) → questo è l'approccio di Brinch Hansen
  - Il thread aspetta finchè il thread svegliato non usa più la risorsa → approccio di Hoare.

- Generalmente si usa il primo approccio.

Quando usare i thread

- Quando c'è bisogno di alto throughput – I thread sono ideali per applicazioni server (tipicamente server web).
- Quando c'è bisogno di alte prestazioni – in macchine multiprocessore, ogni thread può essere eseguito potenzialmente in parallelo.
- Sovrapposizione di I/O e CPU – normalmente dedicando un thread al calcolo e un'altro all'I/O.
- Quando c'è una frequente creazione di processi – tipicamente in applicazioni client-server: I server creano un processo per ogni cliente.

Svantaggi...

- Maggiore complessità – algoritmi e sincronizzazione
- Difficoltà di debug e test – I debuggers che lavorano a livello dei thread sono nuovi e più primitivi di quelli che lavorano a livello processo. Inoltre: ambiente multiprocessore (problemi algoritmo e debug).
- Sincronizzazione dei dati e competizione
- Possibile sorgente di stallo
- Esistenza di librerie non rientranti: creano un ambiente non utilizzabile per I thread

Esempi

- Si supponga che i thread a livello utente aspettino frequentemente su timer, eventi o completamento di I/O.
  - Non è logico associare ciascuno di questi ad un thread di kernel, dato che vedrebbero poca attività di CPU.
  - È meglio inquesto caso associare questi thread ad un singolo thread di sistema, dando minore sovraccarico di sistema e migliori prestazioni.
- Si supponga invece che I thread di utente richiedano grosso carico computazionale. In questo caso è bene associare ogni thread ad un thread d'utente separato.
- Questa possibilità introduce una buona flessibilità nella schedulazione, selezionando uno dei thread disponibili per l'esecuzione.

In pratica...

- La maggioranza dei programmi multithread non includono solo un tipo di thread.
- Il più grande vantaggio di uno schedulatore a due livelli è la sua abilità di configurare la politica di allocazione dei thread sulla base delle caratteristiche dei thread a livello utente.

Semafori POSIX

- Ricorda: variabili protette gestite con le funzioni atomiche wait e signal. Pseudocodice:

```
wait(S):
   if(S<=0)
      aggiungi il descrittore del processo in coda su S; \
   else
      S--;

signal(S):
   if(c'e' un descrittore in attesa su S)
      esegui il processo;
   else
      S++;
```

- Principio d'uso:

```
--- P1 -------          --- P2 -------
wait(S);                wait(S);
Sezione critica         Sezione critica
signal(S);              signal(S);
```

Semafori contatori in POSIX

- Un semaforo contatore non è così efficiente nel fornire mutua esclusione come un mutex, ma è più generale.
- Le operazioni tipiche dei semafori chiamate di solito *down* e *up* o *wait* e *signal* sono chiamate in POSIX *wait* e *post*.
- Ci sono due tipi di semafori: *named* e *unnamed*.
  I primi permettono accesso da processi multipli; sono inoltre più lenti dei secondi.

*sem_t *sem_open (char *name, int oflags, mode_t creation_mode, unsigned init_val);*

- Il nome deve cominciare con una ‘/’ e non deve contenere altre occorrenze di ‘/’.
- oflags non viene usata per aprire un semaforo esistente. Per creare un semaforo usare il flag O_CREAT, possibilmente in OR con O_EXCL per ottenere la generazione di errori se il semaforo esiste già..
- **creation_mode** specifica I permessi d’accesso ad un semaforo creato. Generalmente si indica RWX per il gruppo di utenti desiderato (U, G o O).
- Il parametro **init_val** viene usato per inizializzare il valore del semaforo.
- Il valore ritornato è un puntatore al semaforo, o -1.

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

- Questa funzione crea un semaforo ‘senza nome’.
- Se il parametro **pshared** non è zero, allora il semaforo può essere condiviso tra I processi attraverso la memoria condivisa.
- Il parametro **value** è usato per inizializzare il valore del semaforo.
- Questa funzione ritorna 0 o -1 se ha successo o no.

*int sem_close (sem_t *sem);*

- Questa funzione è usata per chiudere la connessione con un semaforo con nome, aperto con **sem_open**.
- I semafori con nome sono persistenti; cioè lo stato di un semaforo persiste anche se nessuno ha aperto il semaforo.
- Utilizzando il semaforo dopo che sia chiuso ha un effetto indefinito.

*int sem_destroy (sem_t *sem);*

- Questa funzione viene usata per cancellare un semaforo senza nome dopo il suo utilizzo.
- Il semaforo che viene distrutto deve essere stato precedentemente inizializzato con **sem_init**.
- La distruzione di un semaforo sul quale altri processi sono bloccati causa il loro sblocco con un errore (**errno** = **EINVAL**).
- Usando un semaforo dopo che sia stato distrutto ha un effetto non definito.

*int sem_getvalue (sem_t *sem, int *value);*

- Questa funzione viene usata per ottenere il valore di un semaforo con o senza nome.
- Il valore tornato è positivo se la risorsa controllata dal semaforo è sbloccata.
- Se il valore tornato è 0, allora la risorsa è bloccata.
- Alcune implementazioni ritornano un numero negativo per indicare che la risorsa è bloccata.

*int sem_wait (sem_t *sem);*

- Questa funzione cerca di decrementare il valore del semaforo identificato.
- Se ha avuto successo, il processo chiamante continua.
- Se il valore del semaforo è 0, il thread chiamante è bloccato finchè non può decrementare il valore..

*int sem_post (sem_t *sem);*

- Questa funzione incrementa il valore del semaforo identificato.
- Se un processo è bloccato a causa di una chiamata **sem_wait**, il processo con priorità più alta che sta aspettando viene svegliato.
- **sem_post** è rientrante rispetto ai segnali, e può essere chiamato da un gestore di segnali.

*int sem_trywait (sem_t *sem);*

- Questa funzione cerca di decrementare il valore del semaforo identificato.
- Se ha successo, la funzione ritorna 0.
- Se non ha successo, la funzione ritorna -1 e pone il valore di errno a **EAGAIN**.

*int sem_unlink (char *name);*

- Questa funzione distrugge il semaforo con nome.
- Se altri processi hanno aperto il semaforo, possono continuare ad utilizzarlo finch' non lo chiudono con **sem_close**.

```c
/* questo programma simula un semaforo usando mutex e variabili condizione */
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int counter;
} mysem_t;

void unlock_mutex(void *m)
{
  pthread_mutex_unlock((pthread_mutex_t *)m);
}

void mysem_init(mysem_t *s, int num)
{
  pthread_mutexattr_t m;
  pthread_condattr_t c;

  s->counter = num;

  pthread_mutexattr_init(&m);
  pthread_mutex_init(&s->mutex, &m);
  pthread_mutexattr_destroy(&m);
  pthread_condattr_init(&c);
  pthread_cond_init(&s->cond, &c);
  pthread_condattr_destroy(&c);
}
```

```
void mysem_wait(mysem_t *s)
{
  pthread_mutex_lock(&s->mutex);
  while (!s->counter) {
    pthread_cleanup_push(unlock_mutex,(void *)&s->mutex);
    pthread_cond_wait(&s->cond, &s->mutex);
    pthread_cleanup_pop(0);
  }
  s->counter--;
  pthread_mutex_unlock(&s->mutex);
}
void mysem_post(mysem_t *s)
{
  pthread_mutex_lock(&s->mutex);
  if (!(s->counter++))
    pthread_cond_signal(&s->cond);
  pthread_mutex_unlock(&s->mutex);
}

mysem_t mysem;

void *body(void *arg)
{
  int i,j;
  for (j=0; j<40; j++) {
    mysem_wait(&mysem);
    for (i=0; i<1000000; i++);
    fprintf(stderr,(char *)arg);
    mysem_post(&mysem);
  }
  return NULL;
}
```

```
int main()
{
  pthread_t t1,t2,t3;
  pthread_attr_t myattr;
  int err;

  mysem_init(&mysem,1);

  pthread_attr_init(&myattr);
  err = pthread_create(&t1, &myattr, body, (void *)".");
  err = pthread_create(&t2, &myattr, body, (void *)"#");
  err = pthread_create(&t3, &myattr, body, (void *)"o");
  pthread_attr_destroy(&myattr);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);

  printf("\n");

  return 0;
}
```

```
/* programma dimostrativo per i semafori  */
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t mysem;
void *body(void *arg)
{
  int i,j;
  for (j=0; j<40; j++) {
    sem_wait(&mysem); for (i=0; i<1000000; i++);  fprintf(stderr,(char *)arg);  sem_post(&mysem);
  }
  return NULL;
}
int main()
{
  pthread_t t1,t2,t3;
  pthread_attr_t myattr;
  int err;

  sem_init(&mysem,0,1);
  pthread_attr_init(&myattr);
  err = pthread_create(&t1, &myattr, body, (void *)".");
  err = pthread_create(&t2, &myattr, body, (void *)"#");
  err = pthread_create(&t3, &myattr, body, (void *)"o");
  pthread_attr_destroy(&myattr);
  pthread_join(t1, NULL);   pthread_join(t2, NULL);   pthread_join(t3, NULL);
  printf("\n");
  return 0;
}
```

Problema dei lettori/scrittori

```
#include <stdio.h>
#include <pthread.h>
#define RIT 10000
void *scrittore();
void use_data_read(void);
void think_up_data(void);
void write_data_base(void);
void read_data_base(void);
void ritardo(int t);
pthread_t TID;
pthread_mutex_t mutex;      // Semaforo per la variabile nrlett
pthread_mutex_t arc;        // Semaforo per l'accesso al database
int nrlett;
main()
{
  printf("\n\t\tPROBLEMA DEI LETTORI SCRITTORI\n\n\n");
  pthread_create(&TID, NULL, scrittore, NULL);
  while(1){
    pthread_mutex_lock(&mutex);                  // Down su mutex
    nrlett++;
    if (nrlett == 1) pthread_mutex_lock(&arc);   // Down su arc
    pthread_mutex_unlock(&mutex);                // Up su mutex
    read_data_base();
    pthread_mutex_lock(&mutex);                  // Down su mutex
    nrlett--;
    if (nrlett == 0) pthread_mutex_unlock(&arc); // Up su arc
    pthread_mutex_unlock(&mutex);                // Up su mutex
    use_data_read();
  }
  pthread_join(TID, NULL);
}
```

```
void *scrittore()
{
  while(1){
    think_up_data();
    pthread_mutex_lock(&arc);
    write_data_base();
    pthread_mutex_unlock(&arc);
  }
}
void read_data_base(void)
{
  int i;

  printf("\t\t\t\t|  Lettore legge data base\n");
  for(i=0;i<RIT;i++);               // un pò di ritardo
 }
void think_up_data(void)
{
  int i;

  printf(" Scrittore pensa cosa scrivere  |\n");
  for(i=0;i<RIT*1.1;i++);           // un pò di ritardo
}
void write_data_base(void)
{
  int i;

  printf(" Scrittore scrive data base     |\n");
  for(i=0;i<RIT*1.2;i++);           // un pò di ritardo
}
void use_data_read(void)
{
  int i;

  printf("\t\t\t\t|  Lettore utilizza le info\n");
  for(i=0;i<RIT*1.3;i++);           // un pò di ritardo
}
```

Problema dei 5 filosofi

```
#include <stdio.h>
#include <pthread.h>
#define RIT 10000000
void *filosofo(void* arg);
void penso(int i);
void mangio(int i);
void prendo_forchette(int i);
void test(int i);
void rilascio_forchette(int i);
pthread_t TID[5];
pthread_mutex_t mutex;      // Semaforo per la variabile stato
pthread_mutex_t s[5];       // Array di semafori, uno per filosofo
char stato[5] = {'P','P','P','P','P'};

main()
{
  int i;

  printf("\n\tPROBLEMA DEI 5 FILOSOFI\n\n\n");
  for(i=0;i<5;i++){
    pthread_create(&TID[i], NULL, filosofo, (int *)i);
  }
  for(i=0;i<5;i++){
    pthread_join(TID[i], NULL);
  }
}
void *filosofo(void* arg)
{
  int i = (int)arg;

  while(1){
    penso(i);
    prendo_forchette(i);
    mangio(i);                          // Deve mangiare solo se test è OK
    rilascio_forchette(i);
  }
}
```

```
void penso(int i)
{
  int k;

  printf(" Filosofo %d PENSA\n",i+1);
  for(k=0;k<RIT+i*RIT*0.1;k++);           // Ritardo
}

void mangio(int i)
{
  int k;

  printf(" Filosofo %d MANGIA\n",i+1);
  for(k=0;k<RIT+i*RIT*0.15;k++);          // Ritardo
}

void prendo_forchette(int i)
{
  pthread_mutex_lock(&mutex);             // Down su mutex
  stato[i]='A';
  test(i);
  pthread_mutex_unlock(&mutex);           // Up su mutex
  pthread_mutex_lock(&s[i]);              // Down su s[i]
}

void rilascio_forchette(int i)
{
  int sx,dx;

  sx = (i-1)%5;
  if (sx==-1) sx = 4;
  dx = (i+1)%5;
  pthread_mutex_lock(&mutex);             // Down su mutex
  stato[i]='P';
  test(sx);                               // Controlla se i vicini sono affamati
  test(dx);
  pthread_mutex_unlock(&mutex);           // Up su mutex
}
```

```
void test(int i)
{
  int k,sx,dx;

  sx = (i-1)%5;
  if (sx==-1) sx = 4;
  dx = (i+1)%5;
  printf("\t   i = %d: ",i+1);
  for(k=0;k<5;k++){
    printf("Filos. %d = %c  ",k+1,stato[k]);
  }
  printf("\n");

  if(stato[i]=='A' && stato[sx] !='M' && stato[dx] !='M'){
    stato[i]='M';
    pthread_mutex_unlock(&s[i]);          // Up su s[i]

    printf("\tOK i = %d: ",i+1);
    for(k=0;k<5;k++){
      printf("Filos. %d = %c  ",k+1,*(stato+k));
    }
    printf("\n");

    mangio(i);
  }
}
```

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
void *task1(void *);
void *task2(void *);
class multithreaded_object
{
   pthread_t Thread1,Thread2;
public:
   multithreaded_object(void);
   int c1(void);
   int c2(void);
};
multithreaded_object::multithreaded_object(void)
{  //...
   pthread_create(&Thread1,NULL,task1,NULL);    pthread_create(&Thread2,NULL,task2,NULL);
   pthread_join(Thread1,NULL);                  pthread_join(Thread2,NULL);
}
int multithreaded_object::c1(void)
{    // ...  }
int multithreaded_object::c2(void)
{    // ...  }
multithreaded_object MObj;
void *task1(void *)
{    //...
   MObj.c1();    return(NULL);
}
void *task2(void *)
{    //...
   MObj.c2();    return(NULL);
}
```