Università degli Studi di Roma “Tor Vergata”
Facoltà di Ingegneria

L’API Pthreads

Corso di Sistemi Distribuiti

Valeria Cardellini
Anno accademico 2009/10

Materiale didattico su thread POSIX

- POSIX Threads Programming
  https://computing.llnl.gov/tutorials/pthreads/

- Per approfondimenti
  - K.A. Robbins, S. Robbins, “UNIX Systems Programming: Communication, Concurrency and Threads”, Prentice Hall, 2003.
  - D. Butenhof, “Programming With POSIX Threads”, Addison Wesley, 1997.
  - W.R. Stevens, S.A. Rago, “Advanced Programming in the UNIX(R) Environment, 2nd Edition”, Addison-Wesley, 2005.

Un esempio per iniziare

- Architettura sw di un Web server
- Server iterativo

```
while (1) {
  accept_new_connection();
  read_command_from_connection();
  handle_connection(); /* leggi comando,
ritorna risultato sulla connessione*/
  close_connection();
}
```

– Bassa QoS

– Possibile attacco DoS (un client invia comandi lentamente o si ferma a metà)

- Server multi-process

```
while (1) {
  accept_new_connection();
  read_command_from_connection();
  if (fork() == 0) {
    handle_connection();
    close_connection();
    exit(0);
  }
}
```

– Crash del sistema per numero eccessivo di processi

– Degrado delle prestazioni a causa del numero elevato di processi

– Processi isolati (occorre usare IPC)

Avremmo bisogno di…

- Generare “processi” più velocemente e con un minor overhead sulle risorse del sistema
- Usare meno memoria
- Condividere dati tra flussi di esecuzione

Quali soluzioni?

- Migliorare l’architettura sw del server (preforking, …)
  - Soluzione già esaminata
- Usare i thread
  - Come? Obiettivo delle prossime lezioni

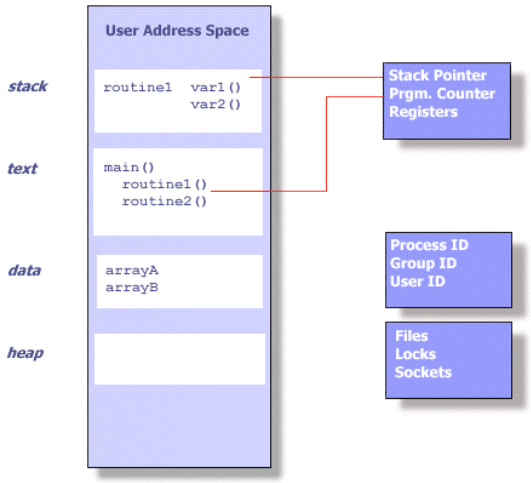
Processi in Unix

- Un processo in Unix è creato dal SO (con un certo overhead) e contiene informazioni sulle risorse del programma e sullo stato di esecuzione
  - Process ID, user ID, group ID
  - Variabili d'ambiente
  - Working directory
  - Codice del programma
  - Registri
  - Stack
  - Heap
  - Descrittori di file
  - Maschera dei segnali e disposizioni
  - Librerie condivise
  - Meccanismi per la comunicazione tra processi
- Un processo in Unix ha un unico thread di controllo

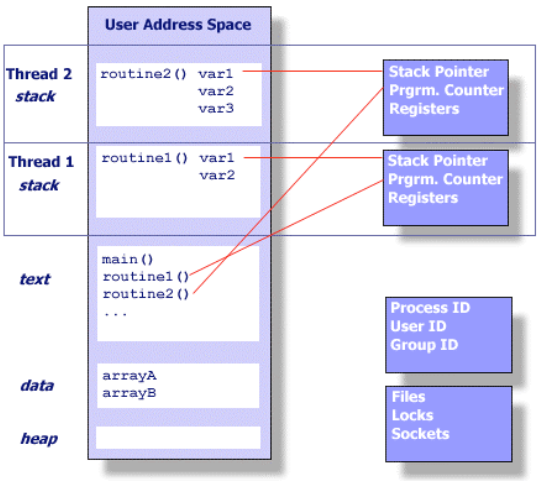
Relazione tra thread e processi

- Un thread è un flusso indipendente di istruzioni che viene schedulato ed eseguito come tale dal SO
  - Un thread ha un flusso di esecuzione autonomo, mantenendo come stato:
    - Thread ID
    - Stack
    - Registri
    - Proprietà di scheduling (priorità e politica)
    - Maschera dei segnali
    - Dati specifici del thread
    - Variabile `errno`
  - In ambiente Unix un thread:
    - Esiste all'interno di un processo ed usa le sue risorse
    - Vive come flusso indipendente finché non muore il suo processo
    - Può condividere risorse con altri thread dello stesso processo
    - E' leggero rispetto ad un processo
  - Vivendo nello stesso processo
    - I thread condividono lo stesso spazio di indirizzamento
    - Cambiamenti fatti da un thread sono visibili agli altri thread
    - Letture e scritture nella stessa memoria richiedono sincronizzazione

Processi e thread in Unix



Processo Unix



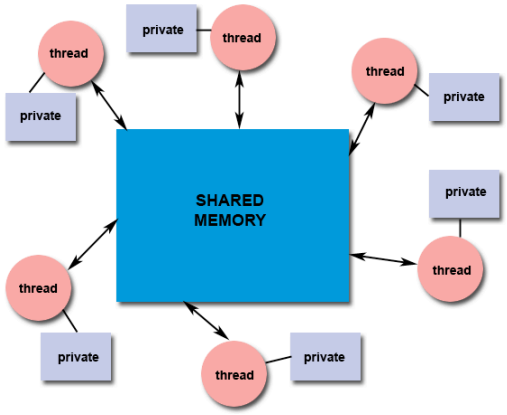
Thread in un processo Unix

Modelli per programmazione multithreaded

- Principali modelli per la programmazione multithreaded:
  - Manager/worker
    - Un unico thread manager che assegna il lavoro ai thread worker
    - Pool di thread worker statico o dinamico
  - Pipeline
    - Il lavoro da compiere è suddiviso in una serie di operazioni, ciascuna delle quali è gestita in modo concorrente da un thread differente
  - Peer
    - Simile a manager/worker, ma anche il manager partecipa al lavoro

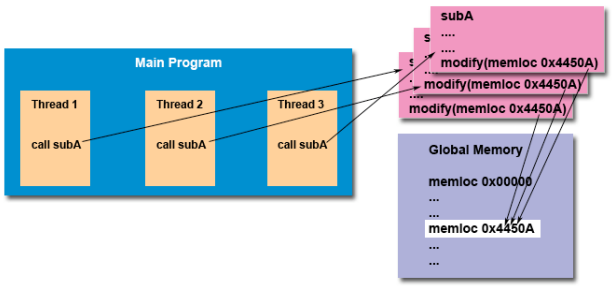
Modello a memoria condivisa

- Tutti i thread dello stesso processo hanno accedono alla stessa memoria globale condivisa
- I thread hanno anche i loro dati privati
- Occorre sincronizzare l'accesso ai dati globali condivisi



Thread safeness

- Thread-safeness: la capacità di eseguire diversi thread senza sporcare i dati condivisi
- Esempio: un'applicazione crea 3 thread che chiamano una funzione di libreria
  - Questa funzione accede/modifica una struttura globale in memoria
  - Può accadere che i thread cerchino di modificare allo stesso tempo l'area condivisa
  - Una libreria *thread-safe* sincronizza l'accesso di ciascun thread alla struttura condivisa



POSIX threads

- L'interfaccia di programmazione dei thread in Unix è stata standardizzata da IEEE nel 1995, con lo standard POSIX 1003.1
- Tutte le implementazioni fedeli a questo standard si chiamano POSIX threads o pthreads
- La libreria definisce un set di tipi C e funzioni, esportate dall'header file pthread.h e implementate tramite la libreria libpthread
- Lo standard POSIX è definito solo per il linguaggio C
- Convenzione per i nomi: tutte le funzioni della libreria pthread iniziano con pthread_
- Esempio di compilazione in Linux

  gcc -pthread main.c -o main

L'insieme di funzioni di pthread

- Tre classi di funzioni
  - Più di 60 funzioni, analizzeremo le principali
  - La maggior parte delle funzioni restituisce 0 in caso positivo, altrimenti un valore diverso da 0 che rappresenta il codice di errore
- Gestione dei thread: funzioni per creare, distruggere, aspettare un pthread
- Mutex: costrutti e funzioni di mutua esclusione (*mutex*) per garantire che un solo thread possa eseguire ad un certo istante un blocco di codice
- Condition variable: costrutti e funzioni per la comunicazione tra thread che condividono un mutex, per aspettare o segnalare il verificarsi di condizioni

Identificazione di un thread

- Come un processo, anche un thread ha un proprio ID (*tid*)
  - A differenza dell'ID di un processo (pid), il tid ha senso solo all'interno del processo a cui il thread appartiene
  - Tipo di dato `thread_id`, che può essere una struttura
- `pthread_self` per conoscere il proprio tid

```
pthread_t pthread_self(void);
```

- `pthread_equal` per confrontare due tid

```
pthread_t pthread_equal(pthread_t tid1, pthread_t tid2);
```


Creazione di un thread

```
int  pthread_create(pthread_t  *tidp,
 pthread_attr_t *attr, void  *(*start_rtn)(void *),
 void *arg);
```

- `pthread_create()` crea un nuovo thread
- Parametri della funzione
  - `tipd`: tid del thread creato
  - `attr`: per impostare gli attributi del thread (NULL di default)
  - `start_rtn`: funzione eseguita dal thread creato
  - `arg`: argomento di ingresso per start_rtn
- Una volta creato, un thread può creare altri thread; non c'è gerarchia o dipendenza implicita tra thread
- Una volta creato il thread, il programmatore sa quando il thread verrà schedulato per l'esecuzione dal SO?

Esempio: stampare l'ID del thread

```
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>

pthread_t ntid;

void printids(const char *s) {
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
           (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}
```


Esempio: stampare l'ID del thread (2)

```
int main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0) {
        printf("Error; can't create thread: %s\n", strerror(err));
        exit(1);
    }
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Output su Linux

```
main thread: pid 32579 tid 3086538432 (0xb7f8d6c0)
new thread:  pid 32579 tid 3086535600 (0xb7f8cbb0)
```

- Osservazioni:
  - `sleep()` in main() per evitare che il thread principale termini prima del nuovo thread
  - `pthread_self()` nel nuovo thread anziché uso della variabile globale ntid
  - Quali sono le soluzioni alternative (migliori)?

Terminazione di un thread

- Un thread può terminare perché:
  - L'intero processo viene terminato con una chiamata alle funzioni `exit()` o `exec()`
  - Il thread termina l'esecuzione della sua routine
  - Il thread chiama `pthread_exit()`

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

    - Usata per far terminare esplicitamente un thread
    - Non causa la chiusura di file aperti dal thread che la chiama
    - Attenzione alla terminazione di main(): usare `pthread_exit()` in main() se si vuole evitare che il processo (e tutti gli altri thread) siano terminati
  - Il thread viene cancellato da un altro thread tramite `pthread_cancel()`

Esempio: creazione e terminazione di thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS   5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

Esempio: creazione e terminazione di thread (2)

```
int main(int argc, char *argv[ ])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;

    for (t=0; t<NUM_THREADS; t++) {
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc) {
            printf("Error; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Output su Linux

```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #4!
Hello World! It's me, thread #3!
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
```


Terminazione di un thread (2)

- Il valore di ritorno `rval_ptr` di `pthread_exit()` è visibile agli altri thread nel processo tramite `pthread_join()`

```
int  pthread_join(pthread_t tid, void **rval_ptr);
```

  - Il thread che chiama `pthread_join()` rimane bloccato finché il thread specificato non termina

Passaggio di argomenti ai thread

- Come passare più di un argomento in input ad un thread creato con `pthread_create()` oppure ritornare più di un argomento in output da un thread che termina con `pthread_exit()`?
  - Passando un puntatore ad una struttura contenente tutti gli argomenti di input o output
  - Attenzione: l'area di memoria puntata deve essere ancora valida quando il thread chiamante termina
    - Ad esempio, non deve essere allocata nello stack del thread che termina (vedi esempio badexit.c)

Esempio: passaggio di parametri

- Come passare un valore intero ad ogni thread creato

```
…
long *taskids[NUM_THREADS];
for (t=0; t<NUM_THREADS; t++) {
    taskids[t] = (long *) malloc(sizeof(long));
    *taskids[t] = t;
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)taskids[t]);
    ...
}
…
```

- L'esempio è corretto?

Esempio: passaggio di parametri (2)

- Come passare ad ogni thread creato più argomenti di input tramite una struttura

```
…
struct thread_data {
    int thread_id;
    int sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg) {
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *)threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    ...
}
```


Esempio: passaggio di parametri (3)

- Segue da lucido precedente

```
int main (int argc, char *argv[ ]) {
    ...
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    rc = pthread_create(&threads[t], NULL, PrintHello,
                                    (void *)&thread_data_array[t]);
    ...
}
```

- L'esempio è corretto?

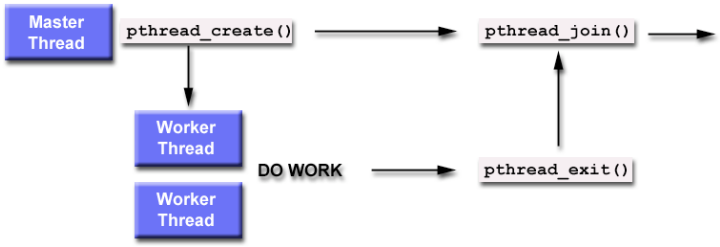
Esempio: passaggio di parametri (4)

```
int rc;
long t;
for (t=0; t<NUM_THREADS; t++) {
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
    ...
}
```

- L'esempio è sbagliato: perché?

Thread joinable o detached

- Già analizzato `pthread_join()`
  - Fornisce un primo meccanismo per sincronizzare i thread




```
int  pthread_detach(pthread_t tid);
```

- Se un thread è detached, quando il thread termina il SO può reclamare le sue risorse

Thread joinable o detached (2)

- Quando si crea un thread, uno dei suoi attributi definisce se il thread è joinable o detached
  - Attenzione: alcune implementazioni della libreria Pthread non creano un thread nello stato joinable
  - No `pthread_join()` su un thread detached

- Come creare un thread joinable (o detached)?
  - Dichiarare una variabile attributo di tipo `pthread_attr_t`
  - Inizializzare la variabile attributo usando la funzione `pthread_attr_init()`
  - Impostare lo stato joinable o detached usando la funzione `pthread_attr_setdetachstate()`
  - Liberare le risorse usate per l'attributo usando la funzione `pthread_attr_destroy()`

Esempio: creazione di thread joinable

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

#define NUM_THREADS   4
void *BusyWork(void *t)
{
        int i;
        long tid;
        double result=0.0;
        tid = (long)t;
        printf("Thread %ld starting...\n",tid);
        for (i=0; i<1000000; i++)
        {
          result = result + sin(i) * tan(i);
        }
        printf("Thread %ld done. Result = %e\n", tid, result);
        pthread_exit((void*) t);
}
```

Esempio: creazione di thread joinable (2)

```
int main (int argc, char *argv[])
{
        pthread_t thread[NUM_THREADS];
        pthread_attr_t attr;
        int rc;
        long t;
        void *status;

        /* Initialize and set thread detached attribute */
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

        for(t=0; t<NUM_THREADS; t++) {
          printf("Main: creating thread %ld\n", t);
          rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
          if (rc) {
            printf("Error; return code from pthread_create() is %d\n", rc);
            exit(-1);
            }
          }
```


Esempio: creazione di thread joinable (3)

```
        /* Free attribute and wait for the other threads */
        pthread_attr_destroy(&attr);
        for (t=0; t<NUM_THREADS; t++) {
          rc = pthread_join(thread[t], &status);
          if (rc) {
            printf("Error; return code from pthread_join() is %d\n", rc);
            exit(-1);
          }
          printf("Main: completed join with thread %ld having a status of
                    %ld\n", t,(long)status);
      }

      printf("Main: program completed. Exiting.\n");
      pthread_exit(NULL);
}
```

Gestione dello stack

- Lo standard POSIX non definisce la dimensione dello stack di un thread
  - Terminazione del programma o corruzione di dati se si supera la dimensione (di default) dello stack
  - Funzioni `pthread_attr_setstacksize()` per impostare e `pthread_attr_getstacksize()` per conoscere la dimensione dello stack
  - Funzioni `pthread_attr_setstackaddr()` per impostare e `pthread_attr_getstackaddr()` per conoscere l'indirizzo di memoria dello stack

Esempio: gestione dello stack

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
    double A[N][N];
    int i,j;
    long tid;
    size_t mystacksize;
    tid = (long)threadid;
    pthread_attr_getstacksize (&attr, &mystacksize);
    printf("Thread %ld: stack size = %li bytes \n", tid, mystacksize);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i][j] = ((i*j)/3.452) + (N-i);
    pthread_exit(NULL);
}
```

Esempio: gestione dello stack (2)

```
int main(int argc, char *argv[])
{
    pthread_t threads[NTHREADS];
    size_t stacksize;
    int rc;
    long t;
    pthread_attr_init(&attr);
    pthread_attr_getstacksize (&attr, &stacksize);
    printf("Default stack size = %li\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li\n",stacksize);
    pthread_attr_setstacksize (&attr, stacksize);
    printf("Creating threads with stack size = %li bytes\n",stacksize);
    for(t=0; t<NTHREADS; t++) {
        rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
        if (rc) {
            printf("Error; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    printf("Created %ld threads.\n", t);
    pthread_exit(NULL);
}
```


Sincronizzazione tra thread

- L'operazione di join è una forma elementare di sincronizzazione
- Sincronizzazione attraverso variabili globali
  - Condivise tra thread
  - Meccanismi di protezione
- Meccanismi forniti dalla libreria Pthread
  - Semafori di mutua esclusione
  - Lock lettore/scrittore
  - Variabili condition
- Compito del programmatore
  - Corretto utilizzo delle funzioni di sincronizzazione

Semafori di mutua esclusione (mutex)

- Protezione delle sezioni/regioni critiche
  - Variabile condivisa modificata da più thread
  - Obiettivo: fare in modo che le sezioni critiche di due thread non vengano mai eseguite contemporaneamente (mutua esclusione)
  - Solo un thread alla volta può accedere ad una risorsa condivisa protetta da un **mutex**
  - Il mutex è un semaforo binario
    - Due soli stati del mutex: aperto (*unlocked*) o chiuso (*locked*)
    - Solo un thread alla volta può possedere il mutex (ottenerne il lock)

- Interfaccia
  - Lock per bloccare una risorsa condivisa
  - Unlock per liberare una risorsa condivisa

Evitare interferenze

- I mutex permettono di prevenire una race condition (*interferenza*)
- Esempio di race condition

Thread 1	Thread 2	Saldo
Legge saldo: € 1000		€ 1000
	Legge saldo: € 1000	€ 1000
	Deposita € 200	€ 1000
Deposita € 200		€ 1000
Aggiorna saldo € 1000 + € 200		€ 1200
	Aggiorna saldo € 1000 + € 200	€ 1200

Uso dei mutex

- Sequenza tipica di uso dei mutex
  - Creazione e inizializzazione di una variabile mutex
  - Più thread tentano di accedere alla risorsa condivisa invocando l'operazione di lock
  - Un solo thread riesce ad acquisire il mutex mentre gli altri si bloccano
    - Variante di lock non bloccante: trylock
  - Il thread che ha acquisito il mutex manipola la risorsa condivisa
  - Lo stesso thread la rilascia invocando la unlock
  - Un altro thread acquisisce il mutex e così via

Mutex: tipo e inizializzazione

- Nella libreria Pthread un mutex è una variabile di tipo `pthread_mutex_t`
- Prima di essere usato, occorre inizializzare il mutex
- Inizializzazione
  - Statica: contestuale alla dichiarazione (più efficiente)

```
pthread_mutex_t mymutex =
  PTHREAD_MUTEX_INITIALIZER;
```

  - Dinamica: attraverso la funzione `pthread_mutex_init()`

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

- Parametri della funzione `pthread_mutex_init()`:
  - `mutex`: puntatore al mutex da inizializzare
  - `attr`: puntatore agli attributi del mutex
    - Estensioni per sistemi real time; se NULL usa valori di default

Mutex: operazioni lock e trylock

- Due varianti per il locking di un mutex
  - bloccante (standard)
  - non bloccante (utile per evitare deadlock)

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

- Parametro
  - `mutex`: puntatore al mutex da bloccare
- Valore di ritorno
  - 0 in caso di successo, diverso da 0 altrimenti
  - `trylock()` restituisce `EBUSY` se il mutex è occupato

Mutex: unlock e distruzione

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Parametro
  - `mutex`: puntatore al mutex da sbloccare/distruggere
- Valore di ritorno
  - 0 in caso di successo, diverso da 0 altrimenti
- Se il mutex è stato allocato dinamicamente, `pthread_mutex_destroy()` per liberare la memoria occupata
- Cosa succede se un thread fa riferimento ad un mutex dopo che questo è stato distrutto? Cosa succede se un thread invoca pthread_mutex_destroy ed un altro thread ha il mutex bloccato?
  - Non definito

Esempio: protezione di struttura dati

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

struct foo {
    int                 f_count;
    pthread_mutex_t  f_lock;
    /* ... more stuff here ... */
};
struct foo *foo_alloc(void)             /* allocate the object */
{
    struct foo *fp;
    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}
```


Esempio: protezione di struttura dati (2)

```
void foo_hold(struct foo *fp)           /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}
void foo_rele(struct foo *fp)           /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
    pthread_mutex_unlock(&fp->f_lock);
    pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    }
    else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Esempio: proteggere una funzione di libreria thread-unsafe

- Un generatore di numeri casuali protetto da un mutex
  - Restituisce un numero tra 0 e 1

```
#include <pthread.h>
#include <stdlib.h>
int randsafe(double *ranp)
{
    static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    int error;
    if (error = pthread_mutex_lock(&lock))
            return error;
    *ranp = (rand() + 0.5)/(RAND_MAX + 1.0);
    return pthread_mutex_unlock(&lock);
}
```


Esempio: flag di sincronizzazione

```
#include <pthread.h>
static int doneflag = 0;
static pthread_mutex_t donelock = PTHREAD_MUTEX_INITIALIZER;

int getdone(int *flag) { /* get the flag */
    int error;
    if (error = pthread_mutex_lock(&donelock))
            return error;
    *flag = doneflag;
    return pthread_mutex_unlock(&donelock);
}

int setdone(void) { /* set the flag */
    int error;
    if (error = pthread_mutex_lock(&donelock))
            return error;
    doneflag = 1;
    return pthread_mutex_unlock(&donelock);
}
```

Un flag di sincronizzazione che è pari ad 1 se setdone è stata invocata almeno una volta, 0 altrimenti

Esempio: flag di sincronizzazione (2)

- Come esempio, usiamo il flag di sincronizzazione per decidere se eseguire un altro comando in un'applicazione multithreaded

```
…
void docommand(void);
…
int error = 0;
int done = 0;
while (!done && !error) {
    docommand();
    error = getdone(&done);
}
```

vedere dotprod_mutex.c

Lock lettore/scrittore

- Simili ai mutex, ma consentono un maggior grado di parallelismo
- Mutex
  - 2 stati: locked, unlocked
  - Solo un thread alla volta può acquisire il mutex
- Lock lettore/scrittore
  - 3 stati: locked in lettura, locked in scrittura, unlocked
  - Solo un thread alla volta può acquisire il lock lettore/scrittore in modalità di scrittura
  - Molteplici thread possono acquisire contemporaneamente il lock lettore/scrittore in modalità di lettura
  - Se un lock lettore/scrittore è write-locked, i thread che cercano di acquisire il lock si bloccano
  - Se un lock lettore/scrittore è read-locked, i thread che cercano di acquisire il lock in lettura riescono, mentre i thread che cercano di acquisire il lock in scrittura si bloccano

  → Problema lettori/scrittori con preferenza ai lettori
- Adatti per gestire strutture date che sono più frequentemente lette piuttosto che modificate

Lock lettore/scrittore (2)

- E' una variabile di tipo `pthread_rwlock_t`
- Come il mutex, prima di essere usato il lock lettore/scrittore deve essere inizializzato (dinamicamente)
  - Funzione `pthread_rwlock_init()`
- Per bloccare il lock lettore/scrittore in modalità di lettura
  - Funzione `pthread_rwlock_rdlock()`
- Per bloccare il lock lettore/scrittore in modalità di scrittura
  - Funzione `pthread_rwlock_wrlock()`
- Per sbloccare il lock lettore/scrittore
  - Funzione `pthread_rwlock_unlock()`
- Per distruggere il lock lettore/scrittore
  - Funzione `pthread_rwlock_destroy()`

vedere rwlock.c

Variabili condition

- Oggetti di sincronizzazione su cui un thread si può bloccare in attesa
  - Associati ad una condizione logica arbitraria (predicato)
  - Evitano il polling (busy-waiting) fino al verificarsi di una condizione
    - Invece di while (x != y);
    1. Lock su un mutex
    2. Valuta la condizione x==y
    3. Se vera, unlock (esplicito) sul mutex ed uscita dal loop
    4. Se falsa, thread sospeso ed unlock (implicito) sul mutex
- Tipo di dato `pthread_cond_t`
- Attributi variabili condizione di tipo `pthread_condattr_t`

Variabili condition: inizializzazione

- Per inizializzare/distruggere variabili di condizione
- Inizializzazione
  - Statica: contestuale alla dichiarazione

    ```
    pthread_cond_t mycond = PTHREAD_COND_INITIALIZER;
    ```
  - Dinamica: attraverso la funzione `pthread_cond_init()`

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

int pthread_cond_destroy(pthread_cond_t *cond);
```


Variabili condition: sincronizzazione

- Una variabile condition è sempre associata ad un mutex che protegge la variabile
  - Un thread ottiene il mutex associato e valuta il predicato
  - Se il predicato è verificato, il thread esegue le sue operazioni e rilascia il mutex associato
  - Se il predicato non è verificato, in modo automatico e atomico
    - Il mutex associato viene rilasciato (implicitamente)
    - Il thread si blocca sulla variabile condition
  - Un thread bloccato riacquisisce in modo automatico e atomico il mutex associato nel momento in cui viene svegliato da un altro thread; il mutex deve essere esplicitamente rilasciato dal thread che sveglia
  - Un thread deve bloccare il mutex associato prima di cambiare lo stato della condizione; il mutex deve essere esplicitamente rilasciato dopo aver risvegliato uno o più thread

Variabili condition: wait

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

- Parametri della funzione
  - `cond`: puntatore all'oggetto condizione su cui bloccarsi
  - `mutex`: puntatore all'oggetto mutex da sbloccare (il mutex protegge la condizione)
- Valore di ritorno
  - 0 in caso di successo, diverso da 0 altrimenti
- Anche versione temporizzata
  - Specificare tempo assoluto (t_{now}+x) anziché relativo (x)

```
int pthread_cond_timedwait(pthread_cond_t *cond,
  pthread_mutex_t mutex, struct timespec *timeout);
```

```
while (x != y);
```

```
pthread_mutex_lock(&m);
while (x != y)
        pthread_cond_wait(&v, &m);
/* modify x or y if necessary */
pthread_mutex_unlock(&m);
```


Variabili condition: signal

- Due varianti
  - standard (sblocca un solo thread bloccato quando viene segnalata la condizione specificata)
  - broadcast (sblocca tutti i thread bloccati quando viene segnalata la condizione specificata)

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

- Parametro delle funzioni
  - `cond`: puntatore all'oggetto condizione
- Valore di ritorno delle funzioni
  - 0 in caso di successo, diverso da 0 altrimenti

Esempio: produttore-consumatore

```
#include <pthread.h>
struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}
```


Esempio: produttore-consumatore (2)

```
void enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_cond_signal(&qready);
    pthread_mutex_unlock(&qlock);
}
```

Esempi di server TCP multi-threaded

- Analizziamo 3 versioni di un server TCP multi-threaded
  A. Un thread per client, accept nel main thread
  B. Pre-threading (pool statico), accept nei worker thread con mutex
  C. Pre-threading (pool statico), accept nel main thread con condition variable e relativo mutex

- Per il codice completo, vedere il sito del corso
  - Applicazione client multiprocess (codice in client.c) per generare richieste di dimensione specificata in input
  - Applicazione server con tempi di esecuzione
    - Gestione del segnale SIGINT per catturare la terminazione del server e stampare il tempo di CPU (system e user) del server

Esempio server A

```
#include <signal.h>
#include <pthread.h>
#include "basic.h"

int main(int argc, char **argv)
{
    int                 listensd, connsd, n;
    void                sig_int(int);
    void                *doit(void *);
    pthread_t           tid;
    struct sockaddr_in  servaddr;

    if (argc != 1) {
        fprintf(stderr, "usage: serverthr_basic\n");
        exit(1);   }
    if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        fprintf(stderr, "socket error");
        exit(1);   }
    memset((void *)&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);
```

Esempio server A (2)

```
    if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
            fprintf(stderr, "bind error");
            exit(1); }
    if (listen(listensd, BACKLOG) < 0 ) {
            fprintf(stderr, "listen error");
            exit(1); }
    if (signal(SIGINT, sig_int) == SIG_ERR) {
            fprintf(stderr, "signal error");
            exit(1); }
    for ( ; ; ) {
            if ((connsd = accept(listensd, NULL, NULL)) < 0) {
                    fprintf(stderr, "accept error");
                    exit(1); }
            if ((n = pthread_create(&tid, NULL, &doit, (void *)connsd)) != 0) {
                    errno = n;
                    fprintf(stderr, "pthread_create error");
                    exit(1); }
    }
}
```


Esempio server A (3)

```
void *doit(void *arg)
{
    void  web_child(int);
    int    n;

    if ((n = pthread_detach(pthread_self())) != 0) {
            errno = 0;
            fprintf(stderr, "pthread_detach error");
            exit(1); }
    web_child((int) arg);      /* process client request */
    if (close((int) arg) == -1) {
            fprintf(stderr, "close error");
            exit(1);   }
    return(NULL);
}
void sig_int(int signo)
{
    void  pr_cpu_time(void);
    pr_cpu_time();
    exit(0);
}
```

Esempio server B

```
typedef struct {
    pthread_t       thread_tid;         /* thread ID */
    long            thread_count;       /* # connections handled */
} Thread;
Thread   *tptr;     /* array of Thread structures; calloc'ed */
int                 listensd, nthreads;
socklen_t           addrlen;
pthread_mutex_t     mlock;
```


Esempio server B (2)

```
#include <signal.h>
#include <pthread.h>
#include "basic.h“
#include "serverthr_pre_1.h“

pthread_mutex_t  mlock = PTHREAD_MUTEX_INITIALIZER;

int main(int argc, char **argv)
{
   int      i;
   void    sig_int(int), thread_make(int);
   struct  sockaddr_in    servaddr;

   if (argc != 2) {
      fprintf(stderr, "usage: serverthr_pre_1 <#threads>\n");
      exit(1);  }
   nthreads = atoi(argv[1]);
   if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
            fprintf(stderr, "socket error");
            exit(1); }
```

Esempio server B (3)

```
    memset((void *)&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

    if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
            fprintf(stderr, "bind error");
            exit(1);    }
    if (listen(listensd, BACKLOG) < 0 ) {
            fprintf(stderr, "listen error");
            exit(1);    }
    tptr = (Thread *)calloc(nthreads, sizeof(Thread));
    if (tptr == NULL) {
           fprintf(stderr, "calloc error");
           exit(1);     }
    for (i = 0; i < nthreads; i++)
            thread_make(i);                         /* only main thread returns */
    if (signal(SIGINT, sig_int) == SIG_ERR) {
            fprintf(stderr, "signal error");
            exit(1);    }

    for ( ; ; )
            pause(); /* everything done by threads */
}
```


Esempio server B (4)

```
void sig_int(int signo)
{
    int                 i;
    void   pr_cpu_time(void);

    pr_cpu_time();

    for (i = 0; i < nthreads; i++)
            printf("thread %d, %ld connections\n", i, tptr[i].thread_count);
    exit(0);
}
```

Esempio server B (5)

```
#include <pthread.h>
#include "basic.h“
#include "serverthr_pre_1.h“
void thread_make(int i)
{
    void  *thread_main(void *);
    int    n;
    if ( (n = pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i)) != 0)
    {
        errno = n;
        fprintf(stderr, "pthread_create error");
        exit(1);   }
    return;                    /* main thread returns */
}
```


Esempio server B (6)

```
void *thread_main(void *arg)
{
    int    connsd, n;
    void  web_child(int);

    printf("thread %d starting\n", (int) arg);
    for ( ; ; ) {
        if ((n = pthread_mutex_lock(&mlock)) != 0) {
            fprintf(stderr, "pthread_mutex_lock error");
            exit(1);       }
        if ((connsd = accept(listensd, NULL, NULL)) < 0 ) {
            fprintf(stderr, "accept error");
            exit(1);   }
        if ((n = pthread_mutex_unlock(&mlock)) != 0) {
            fprintf(stderr, "pthread_mutex_unlock error");
            exit(1);       }
        tptr[(int) arg].thread_count++;
        web_child(connsd);                    /* process client request */
        if (close(connsd) == -1) {
            fprintf(stderr, "close error");
            exit(1);   }
    }
}
```

Esempio server C

```
typedef struct {
    pthread_t       thread_tid;         /* thread ID */
    long            thread_count;       /* # connections handled */
} Thread;
Thread  *tptr;              /* array of Thread structures; calloc'ed */
#define  MAXNCLI        32
int                 clisd[MAXNCLI], iget, iput;
pthread_mutex_t clisd_mutex;
pthread_cond_t  clisd_cond;
```


Esempio server C (2)

```
#include <pthread.h>
#include            <signal.h>
#include "basic.h“
#include "serverthr_pre_2.h“
static int nthreads;
pthread_mutex_t clisd_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t    clisd_cond = PTHREAD_COND_INITIALIZER;
int main(int argc, char **argv)
{
    int     i, listensd, connsd;
    void  sig_int(int), thread_make(int);
    socklen_t       addrlen, clilen;
    struct sockaddr *cliaddr;
    struct  sockaddr_in    servaddr;
    if (argc != 2) {
            fprintf(stderr, "usage: serverthr_pre_2 <#threads>\n");
            exit(1);    }
    nthreads = atoi(argv[1]);
    if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    fprintf(stderr, "socket error");
            exit(1);    }
```

Esempio server C (3)

```
memset((void *)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
if ((bind(listensd, (struct sockaddr *) &servaddr, sizeof(servaddr))) < 0) {
        fprintf(stderr, "bind error");
        exit(1);    }
if (listen(listensd, BACKLOG) < 0 ) {
        fprintf(stderr, "listen error");
        exit(1);    }
cliaddr = malloc(addrlen);
tptr = (Thread *)calloc(nthreads, sizeof(Thread));
if (tptr == NULL) {
        fprintf(stderr, "calloc error");
        exit(1);    }
iget = iput = 0;              /* create all the threads */
for (i = 0; i < nthreads; i++)
        thread_make(i);               /* only main thread returns */
if (signal(SIGINT, sig_int) == SIG_ERR) {
        fprintf(stderr, "signal error");
        exit(1);    }
```


Esempio server C (4)

```
    for ( ; ; ) {
        clilen = addrlen;
        connsd = accept(listensd, cliaddr, &clilen);
        pthread_mutex_lock(&clisd_mutex);
        clisd[iput] = connsd;
        if (++iput == MAXNCLI)
                iput = 0;
        if (iput == iget) {
                printf("iput = iget = %d", iput);
                exit(1);        }
        pthread_cond_signal(&clisd_cond);
        pthread_mutex_unlock(&clisd_mutex);
    }
}
void sig_int(int signo)
{
    int     i;
    void  pr_cpu_time(void);
    pr_cpu_time();
    for (i = 0; i < nthreads; i++)
        printf("thread %d, %ld connections\n", i, tptr[i].thread_count);
    exit(0);            }
```

Esempio server C (5)

```
#include <pthread.h>
#include "basic.h“
#include "serverthr_pre_2.h“
void thread_make(int i)
{
    int     n;
    void   *thread_main(void *);
    if ((n = pthread_create(&tptr[i].thread_tid, NULL, &thread_main, (void *) i)) != 0) {
            errno = n;
            fprintf(stderr, "pthread_create error");
            exit(1);   }
    return;                         /* main thread returns */
}
void *thread_main(void *arg){
    int     connsd, n;
    void   web_child(int);
    printf("thread %d starting\n", (int) arg);
```


Esempio server C (6)

```
    for ( ; ; ) {
            if ((n = pthread_mutex_lock(&clisd_mutex)) != 0) {
                    fprintf(stderr, "pthread_mutex_lock error");
                    exit(1);        }
            while (iget == iput)
                    pthread_cond_wait(&clisd_cond, &clisd_mutex);
            connsd = clisd[iget];           /* connected socket to service */
            if (++iget == MAXNCLI)
                    iget = 0;
            if ((n = pthread_mutex_unlock(&clisd_mutex)) != 0) {
                    fprintf(stderr, "pthread_mutex_unlock error");
                    exit(1);        }
            tptr[(int) arg].thread_count++;
            web_child(connsd);                      /* process client request */
            if (close(connsd) == -1) {
                    fprintf(stderr, "close error");
                    exit(1);   }
            }
    }
```