LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

Nota sugli “snippet” di codice

Alcuni esempi di codice possono essere semplificati, ad esempio omettendo il blocco principale con la funzione `main` *(che andrebbe aggiunto) oppure elencando alcune o tutte le librerie da includere tutte su una riga o insieme (per cui invece occorre trascrivere correttamente le direttive* `#include` *secondo la sintassi corretta) o altre semplificazioni analoghe. In questi casi occorre sistemare il codice perché possa essere correttamente compilato e poi eseguito.*

Threads

Threads

I thread sono singole sequenze di esecuzione all'interno di un processo, aventi alcune delle proprietà dei processi. I threads non sono indipendenti tra loro e condividono il codice, i dati e le risorse del sistema assegnate al processo di appartenenza. Come ogni singolo processo, i threads hanno alcuni elementi indipendenti, come lo stack, il PC ed i registri del sistema.

La creazione di threads consente un parallelismo delle operazioni in maniera rapida e semplificata. Context switch tra threads è rapido, così come la loro creazione e terminazione. Inoltre, la comunicazione tra threads è molto veloce.

Per la compilazione è necessario aggiungere il flag `-pthread`, ad esempio:

```
gcc -o program main.c -pthread
```

Creazione

In C i thread corrispondono a delle funzioni eseguite in parallelo al codice principale. Ogni thread è identificato da un ID e può essere gestito come un processo figlio, con funzioni che attendono la sua terminazione.

```
int pthread_create(
    pthread_t *restrict thread, /* Thread ID */
    const pthread_attr_t *restrict attr, /* Attributes */
    void *(*start_routine)(void *), /* Function to be executed */
    void *restrict arg /* Parameters for the above function */
);
```

Esempio creazione

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadCreate.c

void * my_fun(void * param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
}
```

Nel kernel...

```
$ ./threadList.out & ps -eLf
```

Light-**W**eight **P**rocess:
LWP: identificativo Thread
NLWP: numero di Threads nel processo

```c
#include <pthread.h> //threadList.c

void * my_fun(void * param){
    while(1);
}
void main(){
    pthread_t t_id;
    pthread_create(&t_id, NULL, my_fun, NULL);
    while(1);
}
```

Terminazione

Un nuovo thread termina in uno dei seguenti modi:

- Chiamando la funzione `noreturn void pthread_exit(void * retval);` specificando un puntatore di ritorno.
- Ritorna dalla funziona associata al thread specificando un valore di ritorno.
- Viene cancellato con `int pthread_cancel(pthread_t thread)`.
- Qualche thread chiama `exit()`, o il thread che esegue `main()` ritorna dallo stesso, terminando così tutti i threads.

Cancellazione di un thread

```
int pthread_cancel(pthread_t thread);
```

Invia una **richiesta** di cancellazione al thread specificato, il quale reagirà (come e quando) a seconda di due suoi attributi: **state** e **type**.

State può essere *enabled* (default) o *disabled*: se *disabled* la richiesta rimarrà in attesa fino a che **state** diventa *enabled*, se *enabled* la cancellazione avverrà a seconda di **type**.

Type può essere *deferred* (default) o *asynchronous*: il primo attende la chiamata di un *cancellation point*, il secondo termina in qualsiasi momento. I *cancellation points* sono funzioni definite nella libreria pthread.h (lista).

Cancellazione di un thread

State e **type** di un thread possono essere modificati solo dal thread stesso con le seguenti funzioni:

```
int pthread_setcancelstate(int state, int *oldstate);
```

con state = `PTHREAD_CANCEL_DISABLE` o `PTHREAD_CANCEL_ENABLE`

```
int pthread_setcanceltype(int type, int *oldtype);
```

Con type = `PTHREAD_CANCEL_DEFERRED` o `PTHREAD_CANCEL_ASYNCHRONOUS`

Esempio cancellazione 1

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thCancel.c
int i = 1;
void * my_fun(void * param){
    if(i--)
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,NULL); //Change mode
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    printf("Thread %ld finished\n",*(pthread_t *)param);
}
int main(void){
    pthread_t t_id1, t_id2;
    pthread_create(&t_id1, NULL, my_fun, (void *)&t_id1); sleep(1); //Create
    pthread_cancel(t_id1); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id1);
    pthread_create(&t_id2, NULL, my_fun, (void *)&t_id2); sleep(1); //Create
    pthread_cancel(t_id2); //Cancel
    printf("Sent cancellation request for thread %ld\n",t_id2);
    sleep(5); printf("Terminating program\n");
}
```

Esempio cancellazione 2

```c
#include <stdio.h> <pthread.h> <unistd.h>  <string.h>          //thCancel2.c
int tmp = 0;
void * my_fun(void * param){
    pthread_setcanceltype(*(int *)param,NULL); // Change type
    for (long unsigned i = 0; i < (0x9FFF0000); i++); //just wait
    tmp++;
    open("/tmp/tmp",O_RDONLY);  //Cancellation point!
}
int main(int argc, char ** argv){ //call program with 'async' or 'defer'
    pthread_t t_id1; int arg;
    if(!strcmp(argv[1],"async")) arg = PTHREAD_CANCEL_ASYNCHRONOUS;
    else if(!strcmp(argv[1],"defer")) arg = PTHREAD_CANCEL_DEFERRED;
    pthread_create(&t_id1, NULL, my_fun, (void *)&arg); sleep(1); //Create
    pthread_cancel(t_id1); sleep(5); //Cancel
    printf("Tmp %d\n",tmp);
}
```

Aspettare un thread

Un processo (thread) che avvia un nuovo thread può aspettare la sua terminazione mediante la funzione:

```
int pthread_join(pthread_t thread, void ** retval);
```

Questa funzione ritorna quando il thread identificato da `thread` termina, o subito se il thread è già terminato. Se il valore di ritorno del thread non è nullo (parametro di `pthread_exit` o di `return`), esso viene salvato nella variabile puntata da `retval`. Se il thread era stato cancellato, `retval` è riempito con `PTHREAD_CANCELED`.

Solo se il thread è joinable può essere aspettato! Un thread può essere aspettato da al massimo un thread!

Esempio join I

```c
#include <stdio.h> <pthread.h> <unistd.h>                    //thJoin.c
void * my_fun(void * param){
    printf("Thread %ld started\n",*(pthread_t *)param); sleep(3);
    char * str = "Returned string";
    pthread_exit((void *)str); //or 'return (void *) str;'
}
void main(){
    pthread_t t_id;
    void * retFromThread; //This must be a pointer to void!
    pthread_create(&t_id, NULL, my_fun, (void *)&t_id); //Create
    pthread_join(t_id,&retFromThread); // wait thread
    // We must cast the returned value!
    printf("Thread %ld returned '%s'\n",t_id,(char *)retFromThread);
}
```

Esempio join II

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadJoin.c
void * my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);
    return (void *)3;
}
void main(){
    pthread_t t_id;
    int arg=10, retval;
    pthread_create(&t_id, NULL, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    sleep(3);
    pthread_join(t_id, (void **)&retval); //A pointer to a void pointer
    printf("retval=%d\n", retval);
}
```

Attributi di un thread

Ogni thread viene creato con degli attributi specificati nella struttura `pthread_attr_t`. Questa struttura è usata solo alla creazione di un thread, ed è poi indipendente dallo stesso (se cambia, gli attributi del thread non cambiano). La struttura va inizializzata e, a fine utilizzo, distrutta. Una volta inizializzati, i vari attributi della struct possono, e devono, essere modificati singolarmente delle funzioni

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_setxxxx(pthread_attr_t *attr, params);
int pthread_attr_getxxxx(const pthread_attr_t *attr, params);
```

Attributi di un thread

- `…detachstate(pthread_attr_t *attr, int detachstate)`
  - `PTHREAD_CREATE_DETACHED` → non può essere aspettato
  - `PTHREAD_CREATE_JOINABLE` → default, può essere aspettato
  - Può essere cambiato durante l'esecuzione con
    `int pthread_detach(pthread_t thread);`
- `...sigmask_np(pthread_attr_t *attr,const sigset_t *sigmask);`
- `...affinity_np(...)`
- `...setguardsize(...)`
- `...inheritsched(...)`
- `...schedparam(...)`
- `...schedpolicy(...)`
- altri

Detached e joinable threads

I threads vengono creati di default nello stato joinable, il che consente ad un altro thread di attendere la loro terminazione attraverso il comando `pthread_join`. I thread joinable rilasciano le proprie risorse non alla terminazione ma quando un thread fa il join con loro (salvando lo stato di uscita) (similmente ai sottoprocessi), oppure alla terminazione del processo. Contrariamente, i thread in stato detached liberano le loro risorse immediatamente una volta terminati, ma non consentono ad altri processi di fare il “join”.

NB: un thread detached non può diventare joinable durante la sua esecuzione, mentre il contrario è possibile.

Esempio attributi

```c
#include <stdio.h> <pthread.h> <unistd.h> //threadAttr.c
void *my_fun(void *param){
    printf("This is a thread that received %d\n", *(int *)param);return (void*)3;
}
void main(){
    pthread_t t_id; pthread_attr_t attr;
    int arg=10, detachSTate;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); //Set detached
    pthread_attr_getdetachstate(&attr,&detachSTate); //Get detach state
    if(detachSTate == PTHREAD_CREATE_DETACHED) printf("Detached\n");
    pthread_create(&t_id, &attr, my_fun, (void *)&arg);
    printf("Executed thread with id %ld\n",t_id);
    pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE); //Inneffective
    sleep(3); pthread_attr_destroy(&attr);
    int esito = pthread_join(t_id, (void **)&detachSTate);
    printf("Esito '%d' is different 0\n", esito);
}
```

Mutex

Il problema della sincronizzazione

Quando eseguiamo un programma con più thread essi condividono alcune risorse, tra le quali le variabili globali. Se entrambi i thread accedono ad una sezione di codice condivisa ed hanno la necessità di accedervi in maniera esclusiva allora dobbiamo instaurare una sincronizzazione. I risultati, altrimenti, potrebbero essere inaspettati.

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>      //syncProblem.c
pthread_t tid[2];
int counter = 0;
void *thr1(void *arg){
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 1 expects 2 and has: %d\n", counter);
}
void *thr2(void *arg){
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++); //wait some cycles
    counter += 1;
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
...
```

Esempio

```
...
void main(void){
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

Una possibile soluzione: mutex

I mutex sono dei semafori imposti ai thread. Essi possono proteggere una determinata sezione di codice, consentendo ad un thread di accedervi in maniera esclusiva fino allo sblocco del semaforo. Ogni thread the vorrà accedere alla stessa sezione di codice dovrà aspettare che il semaforo sia sbloccato, andando in sleep fino alla sua prossima schedulazione.

I mutex vanno inizializzati e poi assegnati ad una determinata sezione di codice.

Creare e distruggere un mutex

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

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

```c
#include <pthread.h>                                    //createMutex.c
pthread_mutex_t lock;
int main(void){
    pthread_mutex_init(&lock, NULL); // Create mutex with default attrs
    pthread_mutex_destroy(&lock); // Destroy mutex (it can be re-init)
}
```

Bloccare e sbloccare un mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex)

int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

```
#include <pthread.h>
pthread_mutex_t lock;
void * thread(void *){
    pthread_mutex_lock(&lock); /* Waits for mutex to be unlocked, then
                                  locks it becoming the owner */
    pthread_mutex_unlock(&lock); //Unlock mutex allowing others to lock it
}...
```

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>      //mutex.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
}...
```

Esempio

```
...
void* thr2(void* arg){
    pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
int main(void){
    pthread_mutex_init(&lock, NULL);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

Tipi di mutex

- `PTHREAD_MUTEX_NORMAL`: no deadlock detection
  - Ribloccare quando bloccato → deadlock
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined
- `PTHREAD_MUTEX_ERRORCHECK`: error checking
  - Ribloccare quando bloccato → error
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- `PTHREAD_MUTEX_RECURSIVE`: multiple locks
  - Ribloccare quando bloccato → increase lock count → richiede stesso numero di sbloccaggi
  - Sbloccare quando bloccato da altri → error
  - Sbloccare quando sbloccato → error
- `PTHREAD_MUTEX_DEFAULT`:
  - Ribloccare quando bloccato → undefined
  - Sbloccare quando bloccato da altri → undefined
  - Sbloccare quando sbloccato → undefined

Esempio

```
#include <pthread.h> <stdlib.h> <unistd.h><stdio.h>     //recursive.c

pthread_mutex_t lock;
pthread_t tid[2];
int counter = 0;

void* thr1(void* arg){
    pthread_mutex_lock(&lock);
    pthread_mutex_lock(&lock);
    counter = 1;
    printf("Thread 1 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0x00FF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);
    printf("Thread 1 expects 2 and has: %d\n", counter);
    pthread_mutex_unlock(&lock);
}...
```

Esempio

```
...
void* thr2(void* arg){
    pthread_mutex_lock(&lock); pthread_mutex_lock(&lock);
    counter = 10;
    printf("Thread 2 has started with counter %d\n",counter);
    for (long unsigned i = 0; i < (0xFFF0000); i++);
    counter += 1;
    pthread_mutex_unlock(&lock);  pthread_mutex_unlock(&lock);
    printf("Thread 2 expects 11 and has: %d\n", counter);
}
void main(){
    pthread_mutexattr_t attr;
    pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&lock, &attr);
    pthread_create(&(tid[0]), NULL, thr1,NULL);
    pthread_create(&(tid[1]), NULL, thr2,NULL);
    pthread_join(tid[0], NULL); pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
}
```

CONCLUSIONI

I “thread” sono una sorta di “processi leggeri” che permettono di eseguire funzioni “in concorrenza” in modo più semplice rispetto alla generazioni di processi veri e propri (forking).

I “MUTEX” sono un metodo semplice ma efficace per eseguire sezioni critiche in processi multithread.

È importante limitare al massimo la sezione critica utilizzando lock/unlock per la porzione di codice più piccola possibile, e solo se assolutamente necessario (per esempio quando possono capitare accessi concorrenti).