I Thread POSIX

Un thread è una unità di esecuzione all'interno di un processo. Un processo può avere più thread in esecuzione, che tipicamente condividono le risorse del processo e, in particolare, la memoria. Lo standard POSIX definisce un'insieme di funzioni per la creazione e la sincronizzazione di thread. Vediamo le principali:

- `pthread_create(pthread_t *thead, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)` prende 4 argomenti:
  1. `thread`: un puntatore a `pthread_t`, l'analogo di `pid_t`. Attenzione che non necessariamente è implementato come un intero (nelle ultime versioni di Linux è un `unsigned long int`, vedi `pthreadtypes.h` in `/usr/include/...`);
  2. `attr`: attributi del nuovo thread. Se non si vogliono modificare gli attributi è sufficiente passare NULL (vedere `pthread_attr_init` per maggiori dettagli);
  3. `start_routine` il codice da eseguire. È un puntatore a funzione che prende un puntatore a void e restituisce un puntatore a void. Ricordarsi che in C il nome di una funzione è un puntatore alla funzione;
  4. `arg` eventuali argomenti da passare, NULL se non si intende passare parametri.

- `pthread_exit(void *retval)` termina l'esecuzione di un thread restituendo `retval`. Si noti che quando il processo termina (`exit`) tutti i suoi thread vengono terminati. Per far terminare un singolo thread si deve usare `pthread_exit`;
- `pthread_join(pthread_t th, void **thread_return)` attende la terminazione del thread `th`. Se ha successo, ritorna 0 e un puntatore al valore ritornato dal thread. Se non si vuole ricevere il valore di ritorno è sufficiente passare NULL come secondo parametro (vedi esempio sotto).
- `pthread_detach(pthread_t th)` se non si vuole attendere la terminazione di un thread allora si deve eseguire questa funzione che pone `th` in stato detached: nessun altro thread potrà attendere la sua terminazione con `pthread_join` e quando terminerà le sue risorse verranno automaticamente rilasciate (evita che diventino thread "zombie"). Si noti che `pthread_detach` non fa si che il thread rimanga attivo quando il processo termina con `exit`.
- `pthread_t pthread_self()` ritorna il proprio thread id.
  ATTENZIONE: questo ID dipende dall'implementazione ed è l'ID della libreria phread e non l'ID di sistema. Per visualizzare l'ID di sistema (quello che si osserva con il comando `ps -L`, dove L sta per Lightweight process, ovvero thread) si può usare una syscall specifica di Linux `syscall(SYS_gettid)`.

Esempio

Vediamo un semplice esempio in cui vengono creati 2 thread che stampano il proprio id (sia quello di libreria che quello di sistema) e vanno in sleep(1) prima di terminare. Il thread principale attende la terminazione dei due thread e poi stampa un messaggio e termina.

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

// stampa gli errori ed esce (non si puo' usare perror)
die(char * s, int e) {
    printf("%s [%i]\n",s,e);
    exit(1);
}

// codice dei thread. Notare che e' una funzione che prende
// un puntatore e ritorna un puntatore (a void)
void * codice_thread(void * a) {
    pthread_t tid;
    int ptid;

    tid=pthread_self();         // library tid
    ptid = syscall(SYS_gettid); // tid assegnato dal SO

    printf("Sono il thread %lu (%i) del processo %i\n",tid,ptid,getpid());
    sleep(1);
    pthread_exit(NULL);
}

main() {
    pthread_t tid[2];
    int i,err;

    // crea i thread
    // - gli attributi sono quelli di default (il secondo parametro e' NULL)
    // - codice_thread e' il nome della funzione da eseguire
    // - non vegnono passati parametri (quarto parametro e' NULL)
    for (i=0;i<2;i++)
        if (err=pthread_create(&tid[i],NULL,codice_thread,NULL))
            die("errore create",err);

    // attende i thread. Non si legge il valore di ritorno (secondo parametro NULL)
    for (i=0;i<2;i++)
        if (err=pthread_join(tid[i],NULL))
            die("errore join",err);

    printf("I thread hanno terminato l'esecuzione correttamente\n");
}
```

Il programma va compilato con l'opzione `-lphread` oppure `-pthread` per linkare la libreria POSIX threads.

```
> gcc test1.c -lpthread -o test1                        <=== opzione -l !!
> ./test1
Sono il thread 140072330872576 (14335) del processo 13793
Sono il thread 140072322479872 (14336) del processo 13793
I thread hanno terminato l'esecuzione correttamente
```

Si possono notare gli ID di libreria (unsigned long) e tra parentesi quelli di sistema visualizzabili con `ps -L`. Provare da un altro terminale mentre i thread sono in esecuzione:

```
$ps -AL | grep test1
14334 14334 pts/0    00:00:00 test1
14334 14335 pts/0    00:00:00 test1
14334 14336 pts/0    00:00:00 test1
$
```

Esercizi

1. Provare a "distaccare" uno dei thread e osservare l'errore restituito dalla join.
   ATTENZIONE: essendo una libreria esterna, gli errori non possono essere visualizzati con `perror` (che stampa gli errori di sistema). Consultare il manuale delle chiamate a libreria per vedere i possibili errori restituiti.

2. Provare a inviare a 2 thread 2 interi letti dalla linea di comando. I due thread calcolano il quadrato del numero intero e il thread principale, infine, stampa la somma dei due valori ottenuti. Fare attenzione: la memoria è condivisa quindi si deve passare ai 2 thread l'indirizzo di una zona di memoria "riservata" in modo da evitare interferenze.
   A tale scopo è consigliabile definire una struct che contiene 2 campi, uno di input e uno di output.

```
// struct usata per passare i valori e ricevere i risultati
struct data {
    int v; // input data
    int r; // results
};
```

   Il programma principale creerà quindi un'array di due elementi di tale struct e passerà ai singoli thread il rispettivo elemento dell'array. In questo modo i due thread possono leggere il proprio input e scrivere il risultato dell'operazione in variabili distinte.

3. Creare 2 thread che aggiornano ripetutamente (in un ciclo for) una variabile condivisa `count` per un numero elevato di volte (ad esempio 1000000). Stampare il valore finale per osservare eventuali incrementi perduti.

   NOTA: il compilatore potrebbe ottimizzare il codice e fare, ad esempio, un unico incremento di 1000000 sulla variablie `count`. In questo caso non si osserveranno interferenze. Per evitare ottimizzazioni dare a gcc l'opzione -O0 (che dovrebbe essere di default). Per sperimentare ottimizzazioni provare con -O oppure -O3. Per visualizzare l'assembly risultante si può usare il comando seguente:

```
        objdump -M intel -d test3 | grep -A 10 codicethread
```

   che fa il disassemlaggio (`-d`) in stile mnemonico intel (`-M intel`) dell'eseguibile `a.out`. Il filtro grep cerca nel codice assembly la stringa `codicethread` (nome della funzione che ci interessa) e stampa 10 righe (opzione `-A 10`, after).
   Dal codice assembly è abbastanza evidente come viene compilato il ciclo for.

Soluzioni

1. [Ex. 1]

```
...
        // distacca il thread th[1]
        if (err=pthread_detach(tid[1])) die("errore detach",err);
...
```

   Outuput

```
> a.out
Sono il thread 139687303882496 (13925) del processo 13924
Sono il thread 139687295489792 (13926) del processo 13924
errore join [22]
```

   L'errore 22 indica appunto che il thread non è 'joinable'. Guardando il manuale di `pthread_join` scopriamo che l'errore si chiama EINVAL, che genericamente indica un 'invalid argument'. Questi codici di errore sono specificati in errno.h. Una volta incluso possiamo confrontare il codice con le costanti relative e stampare messaggi di errore più descrittivi:

```
die(char * s, int e) {
    if (e == EINVAL)
        printf("%s [EINVAL: Invalid argument]\n",s);
    else
        printf("%s [%i]\n",s,e);
    exit(1);
}
```

   Output:

```
$./test1
Sono il thread 3075738480 (14558) del processo 14557
Sono il thread 3067345776 (14559) del processo 14557
errore join [EINVAL: Invalid argument]
$
```

2. [Ex. 2]

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>

// struct usata per passare i valori e ricevere i risultati
struct data {
    int v; // input data
    int r; // results
};

// NOTA: non funziona perror perche' non sono syscall ma phtread library calls
void die(char * s, int i) {
    printf("--> %s [%i]\n",s,i);
    exit(1);
    }

void * codicethread(void * i) {
    pthread_t tid;
    struct data *j=(struct data *) i;

    printf("Sono il thread %lu del processo %d! \n",pthread_self(),getpid());
    j->r=j->v*j->v; // calcola il quadrato di j->v e lo pone in j->r

    pthread_exit(NULL);
}

main(int argc, char * argv[]) {
    pthread_t th[2];
    struct data io[2];
    int ret,i;

    // vogliamo almeno 2 parametri da linea di comando
    if (argc < 3) {
        printf("Usage %s val1 val2\n",argv[0]);
        exit(1);
    }

    // inizializza la struct per i thread copiando i valori interi in input
    for(i=0;i<2;i++)
        io[i].v = atoi(argv[i+1]);

    // crea i due thread e passa il puntatore a io[i] come parametro
    // notare che a ogni thread viene passata una istanza della struct diversa e quindi
    // non ci sono interferenze nonostante la memoria sia condivisa
    for (i=0;i<2;i++) {
        if(ret=pthread_create(&th[i],NULL,codicethread,&io[i]))
            die("errore create",ret);
        printf("Creato il thread %lu, inviato il valore %i\n",th[i],io[i].v);
    }

    // attende i due thread. Notare che non si legge il valore di ritorno del thread
    // i thread, infatti, scrivono tale valore direttamente nella struct
    for (i=0;i<2;i++)
        if(ret=pthread_join(th[i], NULL))
            die("errore join",ret);

    printf("ECCOMI QUI, somma risultati: %i \n",(int) ( io[0].r + io[1].r ) );
}
```

Output

```
> ./a.out 3 4
Creato il thread 140378904565504, inviato il valore 3
Sono il thread 140378904565504 del processo 14415!
Creato il thread 140378896172800, inviato il valore 4
Sono il thread 140378896172800 del processo 14415!
ECCOMI QUI, somma risultati: 25
```

3. [Ex. 3]

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/syscall.h>

#define MAX 1000000

// contatore globale
int count=0;

// non funziona perror perche' non sono syscall ma phtread library calls
void die(char * s, int i) {
    printf("--> %s [%i]\n",s,i);
    exit(1);
    }

void * codicethread(void * i) {
    pthread_t tid;
    int j,l;

    // aggiorna una variabile globale: POSSIBILE INTERFERENZE!
    for (j=0;j<MAX;j++)
        count++;
    pthread_exit(NULL);
}

main() {
    pthread_t th[2];
    int ret,i;

    // crea i due thread
    for (i=0;i<2;i++) {
        if(ret=pthread_create(&th[i],NULL,codicethread,NULL))
            die("errore create",ret);
        printf("Creato il thread %lu\n",th[i]);
    }

    // attende i due thread
    for (i=0;i<2;i++)
        if(ret=pthread_join(th[i],NULL)) die("errore join",ret);

    printf("ECCOMI QUI, contatore = %i \n", count);
}
```

Output con interferenze (notare che la somma è sembre diversa da quella attesa di 2000000)

```
> ./a.out
Creato il thread 4149652336
Creato il thread 4141259632
ECCOMI QUI, contatore = 772422
> ./a.out
Creato il thread 4149914480
Creato il thread 4141521776
ECCOMI QUI, contatore = 1228523
> ....
```

Se osserviamo l'assembly notiamo come viene compilato il ciclo for:

```
$objdump -M intel -d test3 | grep -A 10 codicethread
....
 804855e:       eb 11                   jmp    8048571 <codicethread+0x20> # salta alla cmp
 8048560:       a1 2c a0 04 08          mov    eax,ds:0x804a02c            # legge count in eax
 8048565:       83 c0 01                add    eax,0x1                     # somma 1 a eax
 8048568:       a3 2c a0 04 08          mov    ds:0x804a02c,eax            # salva eax in count
 804856d:       83 45 f4 01             add    DWORD PTR [ebp-0xc],0x1     # aggiunge 1 a j
 8048571:       81 7d f4 3f 42 0f 00    cmp    DWORD PTR [ebp-0xc],0xf423f # controlla se j <= 999999
 8048578:       7e e6                   jle    8048560 <codicethread+0xf>  # nel caso salta alla prima mov
....
$
```

Come vediamo l'incremento di default viene proprio fatto caricando il valore in un registro, incrementando il registro e salvando il valore di nuovo in memoria. Il programma quindi è suscettibile a interferenze sul valore di count (come abbiamo visto dall'output).

Proviamo a compilare il programma con l'opzione -O3. In questo caso l'esecuzione non dà interferenze:

```
$
$ ./test3
Creato il thread 3076332400
Creato il thread 3067849584
ECCOMI QUI, contatore = 2000000
$ ./test3
Creato il thread 3076426608
Creato il thread 3067943792
ECCOMI QUI, contatore = 2000000
$
```

In effetti l'ottimizzazione fa una somma di 1000000 direttamente sulla variabile:

```
$ objdump -M intel -d test3 | grep -A 10 codicethread
...
 8048653:       c7 04 24 00 00 00 00    mov    DWORD PTR [esp],0x0
 804865a:       81 05 2c a0 04 08 40    add    DWORD PTR ds:0x804a02c,0xf4240 # somma 1000000 direttamente in count
 8048661:       42 0f 00
 8048664:       e8 e7 fd ff ff          call   8048450 <pthread_exit@plt>
...
```