Thread Runner – come usare i thread in C – pt.2

Eldon Tyrell: *Quale sarebbe il tuo problema?*
Roy Batty: *La morte.*
Eldon Tyrell: *La morte... beh questo temo sia un po' fuori della mia giurisdizione.*
Roy Batty: *Io voglio più vita, padre!*

Dove eravamo rimasti? Ah, si: nella prima parte di **Blade Runner** (oops… **Thread Runner**) avevamo introdotto l'argomento *thread* partendo dalla base, e cioè dai **POSIX Threads**. Ora, come promesso, tenteremo di scrivere lo stesso esempio dello scorso post usando una interfaccia alternativa, e cioè i **C11 Threads**.



…ti spiego: io sono un POSIX thread e tu un C11 thread…
E qui ci vuole una premessa che parte dal **lato oscuro della forza** (*e vabbé, il C++…*): il *committee* ISO del C++ decise di introdurre, nella versione **C++11**, i *thread* all'interno del linguaggio. Quindi niente più uso diretto dei **POSIX Threads** attraverso le funzioni della libreria *libpthread,* ma uso diretto di costrutti del linguaggio stesso. La realizzazione finale è stata (secondo me) brillante, e i **C++11 Threads** sono una delle poche cose del C++11 che uso frequentemente. Certo,

anche i *C++11 Threads* non sono esenti da difetti, e alcuni anche abbastanza gravi, eh! Come la **supercazzola** dei *future* e *promise* per controllare gli errori (**ne ho parlato qui**)… ma usandoli senza farsi troppe *seghe mentali* sono abbastanza *programmer-friendly*. Comunque già sapete cosa ne penso della brutta deriva del C++ pilotata dal *committee* ISO, ne ho parlato varie volte in passato…

Ma torniamo al nostro amato C: il *committee* ISO del C non poteva rimanere indietro, e quindi hanno pensato di fare la stessa cosa con il **C11**, ovvero far si che i *thread* siano direttamente una parte del C… e ci sono riusciti? Prima di rispondere vi anticipo una considerazione: ho una stima del *committee* ISO del C maggiore di quella che ho di quello del del C++ (*e non ci voleva molto…*), ma in questo caso devo proprio dire che non ci siamo: a seguire vedremo il perché.

Come sono stati pensati i nuovi **C11 Threads**? Allora, hanno preso tutte le funzioni e variabili che compongono i **POSIX Threads** e gli hanno cambiato il nome (e devo ammettere che quelli nuovi sono più semplici e immediati); inoltre, in alcuni casi (pochi, per fortuna), hanno cambiato i tipi dei codici di ritorno e degli argomenti delle funzioni. Punto. Geniale? Non proprio direi, e niente a che vedere con la soluzione brillante usata nel C++11. Motivi per usare questa nuova versione? Zero, direi, e non vi ho ancora esposto l problemi principali…

Comunque, per farla breve, ho riscritto l'esempio dello scorso post usando i **C11 Threads**. Vai col codice!

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <threads.h> #define NUMTHREADS 2 // numero di thread da trattare, in questo caso 2 // struttura per i dati condivisi dai thread
typedef struct { mtx_t mutex; // mutex di sincronizzazione
```

```
comune ai thread bool stop; // flag per stop thread unsigned long cnt; // counter condiviso dai thread
} Thdata; // prototipi locali
int threadFunc(void *arg); // funzione main()
int main(int argc, char* argv[])
{ char errmsg_buf[256]; // buffer per strerror_r(3) // init dei dati condivisi dai thread Thdata thdata; thdata.stop = false; thdata.cnt = 0; // init del mutex di Thdata int error; if ((error = mtx_init(&thdata.mutex, mtx_plain)) != thrd_success) { // errore! printf("%s: non posso creare il mutex (%d)\n", argv[0], error); return 1; } // loop di avvio dei thread thrd_t tid[2]; for (int i = 0; i < NUMTHREADS; i++) { // creo un thread if ((error = thrd_create(&tid[i], threadFunc, &thdata)) != thrd_success) { // errore! printf("%s: non posso creare il thread %d (%d)\n", argv[0], i, error); return 1; } } // dopo 10 secondi fermo tutti i thread sleep(10); thdata.stop = true; // loop di join dei thread for (int i = 0; i < NUMTHREADS; i++) { if ((error = thrd_join(tid[i], NULL)) != thrd_success) { // errore! printf("%s: non posso attendere il thread %d (%d)\n", argv[0], i, error); return 1; } } // cancello il mutex di sincronizzazione mtx_destroy(&thdata.mutex); // esco printf("%s: thread terminati: thdata.cnt=%lu\n", argv[0], thdata.cnt); return 0;
} // threadFunc() - funzione per i thread
int threadFunc(void *arg)
{ // ottengo i dati del thread con un cast (Thdata *) di (void *) arg Thdata *thdata = (Thdata *)arg; // loop del thread printf("thread partito\n"); unsigned long loc_cnt = 0; for (;;) { // lock del mutex mtx_lock(&thdata->mutex); // incremento il counter locale e quello condiviso loc_cnt++; thdata->cnt++; // unlock del mutex mtx_unlock(&thdata->mutex); // test dello stop flag if (thdata->stop) { // il thread esce printf("thread terminato dal main: loc_cnt=%lu\n", loc_cnt); thrd_exit(0); } // sleep del thread (uso usleep solo per comodità invece della nanosleep(2)) usleep(1000); } // il thread esce per altro motivo diverso dallo stop flag printf("thread terminato localmente: loc_cnt=%lu\n", loc_cnt); thrd_exit(0);
}
```

Come vedete il codice nuovo è praticamente identico al

vecchio, mi sono limitato a usare le nuove funzioni al posto di quelle vecchie (per esempio **thrd_create()** invece di **pthread_create(3)**) , ho usato i nuovi tipi (per esempio **mtx_t** invece di **pthread_mutex_t**) e ho leggermente modificato il test dei valori di ritorno: poche differenze, devo dire, e, in alcuni casi, in peggio: ad esempio è sparito il parametro *attr* di **pthread_create(3)**, che (per semplicità) nello scorso esempio avevo lasciato a NULL, ma che a volte può risultare utile (leggere il manuale della funzione per rendersene conto). Comunque si potrebbe dire (senza fare troppo gli schizzinosi) che la nuova interfaccia non ci offre nessun vantaggio sostanziale, ma neanche un peggioramento decisivo, quindi si potrebbe anche usare (*de gustibus*).

Ma c'è un problema: pare che i **C11 Threads** non siano stati considerati una priorità per chi scrive i compilatori e le varie **libc,** quindi per molti anni dall'introduzione è stato difficile compilare/eseguire un programma come quello che ho mostrato. Perfino il nostro amato **GCC** (che di solito è il primo a fornire supporto per le ultime novità) ci ha messo una vita a supportare i nuovi *thread* (in realtà a causa della mancata integrazione nella **glibc).** Questo è un brutto segno, per niente incoraggiante: se il mondo che gira intorno ai linguaggi e ai Sistemi Operativi non approva rapidamente vuol dire che ci sono delle perplessità.

Comunque dopo tanto tempo qualcosa è arrivato, e adesso è possibile compilare i **C11 Threads** su **Linux** (con versioni recenti di GCC e con una *glibc* dalla v.2.28 in su), sui vari sistemi **BSD**, su Windows, ma non ancora su macOS (se non mi sbaglio). Aggiungo un dettaglio illuminante sulla coppia GCC/glibc: in attesa di un supporto veramente nativo, la compilazione era basata su una implementazione che era, praticamente, un *wrapper* che simulava i **C11 Threads** usando i **POSIX Threads** (infatti bisognava *linkare* la *libpthread):* un po' assurdo, no?

E anche se ora si può usare direttamente GCC con una *glibc*

recente (la v.2.28 è del 2018), io ho deciso di seguire con la stessa soluzione che è stata disponibile quasi da subito (già nel 2014) e ho già usato in passato: ho usato la **musl libc** che è una alternativa alla *glibc,* ed è dotata di un *wrapper* per GCC (*musl-gcc*). *musl* fornisce (su Linux) il supporto completo al **C11**, ed è stata (credo) la prima in assoluto a fornire il supporto nativo ai **C11 Thread**. E come funziona tutto questo? Una volta compilato il programma si comporta correttamente, come potete vedere qui sotto:

```
aldo@Linux $ musl-gcc c11threads.c -o c11threads
aldo@Linux $ ./c11threads
thread partito
thread partito
thread terminato dal main: loc_cnt=8597
thread terminato dal main: loc_cnt=8597
./c11threads: thread terminati: thdata.cnt=17194
```

Ma il gioco vale la candela? No, per quel che mi riguarda continuerò ad usare i **POSIX Threads**, che uso da molti anni e rimangono il riferimento d'eccellenza. I **C11 Threads** sono uno strano ibrido che non rispetta la classica sintassi della *glibc* (ad esempio in caso di errore non ritorna -1 ma un codice di errore e, oltretutto, non aggiorna **errno**), e non rispetta pienamente neanche lo **standard POSIX** (i codici di errore non sono inseriti in una lista analizzabile con **strerror(3)**). E poi, come già detto sopra, le nuove funzioni sono equivalenti alle vecchie ma non abbastanza, alcune funzionalità non sono più disponibili e alcune cose semplici (come stampare il *thread Id* interno della libreria o le stringhe degli errori) sono molto più complicate… e, *dulcis in fundo*, nelle classiche pagine dei manuali **Linux** e **POSIX** non c'è ancora traccia delle nuove funzioni. Son problemi, no?

Ok, per oggi chiudo qui. Nel prossimo e ultimo episodio di **Thread Runner** vedremo un esempio pratico. Rimanete sintonizzati, eh!

Ciao e al prossimo post!

https://italiancoders.it/thread-runner-come-usare-i-thread-in-c-pt-2/