Thread e programmazione Multi Threading in C

Andrea Cecchini

December 6, 2022

1 Thread

Un **Thread** e' un componente di un processo che astrae il concetto di **flusso di istruzioni** che lo scheduler puo' far eseguire separatamente o **concorrentemente** con il resto del processo.
Piu' banalmente possiamo pensare ad un Thread come ad una procedura che lavora in parallelo con le altre procedure del processo.

1.1 Contesto di un thread e condivisione della memoria

Un Thread per poter svolgere il suo lavoro si munisce di proprie strutture dati. In particolare un thread ha un proprio **contesto**:

- Process ID
- Program Counter
- Stato dei registri
- **Stack di memoria**
- ...

Tuttavia i thread **condividono** alcune parti del proprio contesto, come:

- Zona Codice della memoria dedicata al processo
- Zona Data della memoria dedicata al processo nel quale e' possibile accedere a variabili globali in maniera condivisa.
- Tabella dei File Descriptors

In questa maniera, oltre a condividere la CPU, ogni thread puo' accedere a tutte le variabili globali di un processo e anche alla tabella dei file descriptors.

1.2 Cambio di contesto e Scheduler

Ruolo centrale nella gestione dei thread viene svolta dallo **Scheduler**. Quello che fa non e' tanto diverso dalla suddivisione del tempo di utilizzo della CPU con i processi, andiamo quindi ad estendere il concetto. Lo Scheduler affida per un certo Δt di tempo la CPU ad un **thread di un processo**. Successivamente lo scheduler potra' decidere di affidare la CPU ad un thread di un altro proceso o **dello stesso processo**.
Questo fatto porta a delle considerazioni da fare.
Il cambio di contesto tra threads dello stesso processo **e' molto piu' veloce** del cambio di contesto da parte di thread appartenti a processi diversi.
Questo fatto e' dovuto al fatto che i thread appartenti ad uno stesso processo condivino parte del contesto, quindi alcune parti rimaranno inalterate nella fase di switching.

1.3 Vantaggi/Svantaggio Thread vs Processi

Esploriamo i **vantaggi** nell'utilizzo di piu' thread anziche utilizzare piu' processi:

- Visibilita' dei dati globali
- Piu' flussi di esecuzione
- Comunicazioni veloci, infatti ogni thread condivide lo stesso spazio di indirizzamento, quindi le comuinicazioni fra i thread sono piu' veloci rispetto alle comuinicazioni fra i processi.
- Facilita' nella gestione degli eventi asincroni
- Context Swithing veloce

Il fatto e' che alcuni vantaggi possono essere visti come dei **svantaggi**:

- Non si parla di parallelismo ma di **concorrenza**, in quanto bisogna gestire il problema delle **mutua esclusione** dei thread verso ai dati condivisi.

ATTENZIONE!, non si parla solamente di mutua esclusione dei dati globali di un processo, la concorrenza deve essere gestita da:

- Dai programmatori che scrivono il programma
- Dal **Sistema Operativo** che deve implementare le funzioni di libreria e le system calls in maniera **thread safe**.

Per introdurre il concetto di **rientranza e thread safe call**, bisogna definire il significato di **operazione atomica**.

1.4 Atomicita' delle operazioni

In generale, un'operazione si dice **atomica** se risulta essere **indivisibile**, ovvero che un'altra operazione che utilizza dei dati condivisi con essa non puo' iniziare prima che sia finita la prima. In altre parole non si puo' utilizzare la caratteristica dell' **interleaving** che mette a disposizione la CPU. Una conseguenza di questo fatto e' che **a parita' di condizioni iniali, un'operazione atomica portera' al medesimo output**. Parlando pero' di operazioni che accedono alla memoria e che potrebbero invocare lo **swapping** e il **page fault** dobbiamo estendere il significato:
Un'operazione si dice atomica se a parita' di input otteniamo lo stesso output.
Questo comporta che un'operazione atomica puo' essere interotta per eseguirne un'altra piu' importante, per poi essere **richiamata e startata dall'inizio**, magari con un cambio di input iniziale.

1.4.1 Le istruzioni C sono atomiche?

NO!!
In generale le istruzioni ad alto livello non sono mai atomiche.
Nemmeno le minuscole operazioni di assegnamento di una variabile ad una costante sono atomiche:

```c
#define CONSTANT_VALUE 16251652
/*
 * Using a 64 unsigned integer inside a 32 bit architetture
 */
uint64_t g_var;

int main(void)
{
    G = CONSTANT_VALUE
    return (0);
}
```

Nella traduzione di questo sorgente C in assembly, vengono eseguite due istruzioni macchina dovute al fatto di stare processando una variabile a 64 bit in un'architettura da 32 bit.

```
mov DWORD PTR G, -1241209825
mov DWORD PTR G+4, 473
```

1.4.2 Le istruzioni in Assembly sono atomiche?

No, non tutte.
La maggioranza delle istruzioni di Assembly non sono altro che **Macro** nelle quali al suo interno vengono chiamate altre istruzioni Assembly. Basti pensare all'istruzioni **call** che permette di mettere sulla cima dello stack la procedura che vogliamo eseguire.
Nemmeno molte istruzioni elementari sono atomiche. Un chiarissimo esempio risulta essere l'istruzione **inc**:

- Deve trasportare sul bus il valore dalla RAM al registro.
- Deve incrementare di un BYTE il valore del registro
- Deve trasportare sul bus il valore dal registro alla RAM.

1.4.3 Cosa c'e' di atomico allora?

Allora, cosa rimane di atomico?
Alcune istruzioni di Assembly potrebbero risultare atomiche, sottostando ad alcune considerazioni. **La sola copia di un dato da registro a memoria** potrebbe essere atomica se:

- Richiede un solo accesso al bus, vale a dire che la dimensione in bit e' adatta all'architettura della macchina.
- Inoltre, si dovrebbe parlare riguardo alla memoria virtuale e riguardate il problem del **page fault**, il quale richiede di ricominciare l'operazione.

Un'istruzione molto particolare che risulta essere atomica viene fornita dal livello ISA dei processori INTEL x86: **compare-exchange**. accompagnata dall'attributo **lock**.

```
lock cpmxchg primo_argomento secondo_argomento // compare-exchange instruction
```

Il primo argomento e' una **l'indirizzo di una variabile intera**, mentre il **secondo** e' il nome del registro in cui e' contenuto il valore da assegnare alla variabile.
Vi e' un terzo argomento, implicito, che e' il contenuto del registro **EAX**, nel quale vi e' il valore con cui confrontare il primo argomento.
Atomicamente, se la variabile intera ha il valore cercato allora lo si assegna con il nuovo valore, altrimenti la variabile intera non viene modificata.

```
while(1)
{
    if(occupato = 0)
    {
        occupato = 1;
        /* Esegi poi operazioni atomiche */
        break;
    }
}
```

Questo codice non risulta essere atomico in quanto l'operazione di controllo e assegnazione di **occupato** non e' atomica. Dovremo quindi utilizzare delle **API** fornite dal sistema operativo che permettono di eseguire la **Compare-Exchange**.

1.5 System call

Le prime versioni di LINUX non implementavano il concetto di thread, ma ne simulavano il comportamento allocando invece dei processi. Lo standard POSIX si adeguo' a questo stardand creando una serie di systemcall non atomiche, che d'ora in poi chiameremo **NON RIENTRANTI** o **NON THREAD SAFE**.

```
/*
 * Questa funzione permette di ottenere la stringa codificata
 * in ASCII partendo dalla rappresentazione dell'indirizzo IP
 * in memoria, il quale e' un uint32_t.
 * Questa funzione e' NON RIENTRANTE in quanto salva il risultato
 * della traduzione in una variabile globale il cui indirizzo
 * viene restituito per ogni chiamante, quindi ai diversi thread.
 */
char* inet_ntoa()
```

Le implementazioni future derivate dallo **STANDARD POSIX** offrono un'implentazione **Thread Safe** per la maggior parte delle syscall. Per distinguere le funzioni rientranti da quelle no sono stati definiti dei nuovi simboli per le chiamate a tale funzioni:

```
/* Funzione di libreria NON RIENTRANTE */
char* strerr(int errnum);
/* Funzione di libreria RIENTRANTE -- aggiunta del postfisso _r */
int strerr_r(int errnum, char* buf, size_T buflen);
```

1.6 POSIX Threads

I thread sono stati **standardizzati**.
I thread POSIX sono noti come **PThread**.
Le **API** per i PThreads si distinguono in 3 parti:

- **Thread Managment**: funzioni per creare, eliminare attendere la fine dei pthread.
- **Mutexes**: funzioni per realizzare un tipo di sincronizzazione semplice chiamata "**mutex**" (mutua esclusione).
- **Condition variables**: funzioni a supporto per una sincronizzazione piu' complessa dipendente dal valore delle variabili.

Convenzioni sui nomi della libreria:

- **pthread** : gestione dei pthread in generale
- **pthread_attr_**: funzioni per la gestione degli attributi dei pthread
- **pthread_mutex_**: gestione mutua esclusione

Il file di intestazione che contiene tali definizioni si chiama **pthread**.

1.6.1 Compilazione dei programmi che utilizzano i PThread

1.6.2 Linking dei programmi che utilizzano i PThread