

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 processo 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 appartenenti a processi diversi.

Questo fatto e' dovuto al fatto che i thread appartenenti ad uno stesso processo condividono parte del contesto, quindi alcune parti rimarranno 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 comunicazioni fra i thread sono piu' veloci rispetto alle comunicazioni fra i processi.
- Facilita' nella gestione degli eventi asincroni
- Context Switching 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 interrotta 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:

```
#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'istruzione **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.