05-23.md

1. Lezione del 23/05 – Concorrenza I (LEZIONE TEORICA)

Quando parliamo di esecuzione intendiamo che all'interno di un processo o di un programma vi siano più flussi di esecuzione. Questi flussi di esecuzione non per forza si alternano, anzi sono pensati proprio per eseguire calcoli indipendenti in parallelo per sfruttare al massimo l'hardware ormai diffuso sui PC.

Alla creazione, un processo dispone di unico flusso di esecuzione, detto thread. Esso esegue le istruzioni contenute nell'eseguibile da cui viene lanciato, partendo dal main o dalle operazioni preliminari ad esso, e puo' richiedere al SO di attivare ulteriori flussi di esecuzione, denominati thread.

La programmazione concorrente è quindi una scelta esplicita del programmatore, in quanto essa puo' portare grossi vantaggi in termini di tempo di esecuzione del programma ma porta con sé anche tutta una serie di complicazioni accessorie.

Ogni thread esegue un suo flusso indipendente, parte da un punto di ingresso deciso dal programmatore e quando ha finito l'esecuzione, causando gli effetti collaterali desiderati o incappando in un errore, termina.

La programmazione concorrente è resa possible dalle funzionalità che il SO espone e dalle funzioni di libreria che le rendono disponibili. È quindi il classico scenario in cui il SO implementa delle primitive per creare i thread, in quando solo esso puo' effettivamente crearli, e le librerire espongono delle API per accedervi con più facilità.

Il SO tiene anche traccia dei thread in esecuzione su di una tabella, assieme ad alcune informazioni tipo lo stato di esecuzione e lo stack delle chiamate. In base ai SO puo' essere un'unica struttura dati, come accade nei sistemi UNIX-like, oppure più pezzi come accade in Windows.

Questi flussi sono solitamente tanti e spetta al SO scegliere quando e quali eseguire, e per fare ciò utilizzerà uno scheduler cui verrà associata una politica. Possiamo immaginare lo scheduler come un oggetto che manipoli due liste, quella dei thread eseguibili e quella dei thread non eseguibili. Un thread è eseguibile quando è pronto ad iniziare l'esecuzione e sta solo attendendo che il thread lo faccia partire ed è non eseguibile quando è in attesa di un evento prima di poter iniziare o continuare.

Quando un core del nostro processore è disponibile, lo sceduler pesca un thread e lo mette in esecuzione su di esso. Essi resteranno in esecuzione per un tempo definito di tempo, detto *quantum*, al termine del quale esso verrà riposto nella lista dei thread eseguibili per permettere anche ad altri thread di venir eseguiti. Più tardi verrà rimesso in esecuzione. La posizione in cui thread in esecuzione viene riposto nella lista è soggetta a vari fattori e alla politica di scheduling, quindi non viene per forza inserito in fondo alla coda.

Un thread interrotto e in seguito ripreso continua l'esecuzione esattamente da dove era stato interrotto, ma questo approccio mal si sposa quando più flussi di esecuzione cercano di cooperare al fine di ottenere un risultato comune.

L'allocazione di un thread su una CPU puo' e deve essere delegata al SO, ed in questo caso si parla di thread nativi, che vengono esposti secondo specifiche API di sistema. Si parla invece di Green Thread quando questi thread sono schedulati da una virtual machine. Questo perché le VM simulano il multi-threading senza supporto del SO e sono gestiti nello user-space anzi che nel kernel space, abilitando al multi-threading anche sistemi che non lo supporterebbero.

Il funzionamento dei thread varia a seconda del SO. Per quanto funzionalmente le differenze siano minime, alcuni accorgimenti vanno fatti nel momento in cui si scrive il codice a seconda della piattaforma per cui si sta sviluppando.

Le funzioni messe a disposizione per i thread solitamente accettano come parametro un funtore (puntatore a funzione) per sapere da dove il nuovo thread dovrà inziare l'esecuzione. Possono

opzionalmente accettare anche uno stack, in quanto ogni thread ha un suo stack di esecuzione, mentre condivide invece con tutti gli altri lo spazio di heap.

Le funzioni di creazione dei thread ritornano solitamente un handle, che ci serve per far riferimento al thread appena creato. Ogni thread è univocamente identificat tramite un ID, detto TID. Inoltre spesso sono presenti funzioni per permettere a un thread di attendere il verificarsi di una condizione senza che questi sprechino tempo in CPU ad attendere (busy-waiting).

Il passaggio da non eseguibile ad eseguibile non implica che il thread verrà eseguito subito, ed è anche interessante notare che molti SO non supportino la cancellazione di un thread, vale a dire che un thread puo' solo arrestarsi da solo.

Tramite la concorrenza possiamo sovrapporre operazioni più lente con operazioni più veloce, come un flusso di dati ed un flusso di operazioni di I/O (non per forza disco, si pensi alla rete).

Valutando l'ipotesi di impiegare i thread nel nostro programma, dobbiamo chiederci se i benefici siano tali da giustificare la complessità avanzata.

Un altro motivo per la concorrenza è, se vogliamo, di natura storica. UNIX originariamente non aveva il concetto di thread, ma solo di processo, ognuno quindi con un singolo thread. Anche operazioni comuni, come `ls | grep ... | sed ...` creavano quindi ognuna il suo processo, e questi andavano poi a comunicare in maniera testuale, il che è problematico dato che la comunicazione tra processi è molto più complessa della comunicazione fra thread.

Il primo problemo che si riscontra passando da un contesto a singolo flusso di esecuzione ad uno concorrente è che l'esecuzione del programma diventa non deterministica. A questo va aggiunto che spesso nelle architetture moderne la lettura da memoria raramente è una vera lettura da memoria ed è invece più soesso una lettura da cache, che è associata al singolo processore e quindi non visibile agli altri.

Due o più thread in esecuzione all'interno di un processo possono spesso proseguire senza interferire l'un l'altro, ma sarà quasi scontato avere almeno un punto in cui il lavoro svolto dai thread debba avere ripercussioni sugli altri. L'utilità di avere computazioni parallele è spesso proprio quella di poter scomporre il lavoro in più parti indipendenti e poi ricombinare il risultato finale.

In genere in un contesto concorrente dobbiamo tentare di garantire tre proprietà: - Attesa Limitata - Un thread messo in attesa di esecuzione non deve poter attendere per sempre - Visibilità - Le operazioni di lettura + modifica di dati comuni non devono poter essere interrotte - Ordinamento - Stabilire le garanzie minime per l'esecuzione nel corretto ordine delle operazioni

I principali errori che si commettono in programmazione concorrente sono proprio quelli di sicnronizzazione, in quando una scorretta o mancata cooperazione fra trhead potrebbe dar luovo a risultati non prevedili o attese indefinite.

Quando nel contesto di un processo si chiede di creare un thread, il kernel lo inserisce nella struttura dati e alloca la memoria per tale thread, quindi lo stack. Esso ha come vincolo che deve essere contiguo, quindi un blocco preallocato per intero anche se inizialmente usato solo in piccola parte. In un processo variabili globali, codice e heap sono condivisi, lo stack no.

(Image: *../imgs/20220523132804.png*)

Vediamo un esempio di creazione di thread in C++:

```
void run(std::string msg) {
    for ( int i = 0; i < 10; i++ )
        std::cout << msg << i << std::endl;
}

int main() {
    std::thread t1(run, "aaaa");
    std::thread t2(run, "bbbb");
    t1.join()
    t2.join()
}
```

Questo codice non ha un risultato fisso, infatti sia:

```
aaaa0
aaaa1
bbbb0
aaaa2
...
```

che:

```
bbbb0
aaaa0
bbbb1
aaaa1
```

Sono due possibili output. Anche Rust ovviamente essendo un linguaggio moderno offre un modo idiomatico per creare dei thread:

```
fn run(msg: String) {
    for i in 0..10 {
        println!("{} {}", msg, i);
    }
}

fn main() {
    let t1 = thread::spawn( || { run("aaaa") } );
    let t2 = thread::spawn( || { run("bbbb") } );
    t1.join().unwrap();
    t2.join().unwrap();
}
```

Una comodità di Rust è che ovviamente il borrow checker ci dà delle garanzie extra sulla corretteezza del codice che si va a scrivere, ma l'analisi statica del codice da sola non puo' identificare tutti i possibili errori di programmazione.

La sincronizzazione è necessaria per evitare il verificarsi delle *race condition*, ovvero comportamenti del codice che dipendono dal modo (in altre parole, dalla sequenza) con cui i thread vengono schedulati, e le si possono superare mediante le operazioni atomiche ormai da anni disponbili su qualunque processore. Il problema delle operazioni atomiche però è che sono molto limitate, spesso ad operazioni molto banali, ed infatti fungono da pietra angolare metodi di sincronizzazione avanzati.