05-24.md

1. Lezione del 24/05 – Concorrenza II

Per gestire le problematiche legate alla concorrenza, sia i processori che i SO ci danno delle astrazioni di alto livello, talvolata anche indipendenti dalla piattaforma.

La prima, legata alle caratteristiche dei processori, sono le operazioni atomiche, che operano su tipi primitivi. Abbiamo poi i Mutex, oggetti possedibili da un solo thread per volta che fanno si che solo chi li possegga possa avanzare in una sezione critica, nome che associamo alle parti di codice che vogliamo possano essere eseguite da un solo thread in un dato istante.

Quando un thread rilascia un mutex, quel che accade è che non solo esso viene reso disponibile, ma il SO va a recuperare altri thread che erano in attesa che esso fosse disponibile per posizionarlo nella lista dei thread eseguibili. Se più thread sono in attesa su un Mutex, la scelta di quale sarà il prossimo non è deterministica.

Si evince subito che un thread che ottiene il possesso di un Mutex senza mai rilasciarlo blocca l'esecuzione di tutti gli altri. In Rust questo processo è difficile accada in quanto il rilascio un Mutex è un'operazione automatica.

Un ulteriore sturmento sono le Condition Variable. I Mutex infatti realizzano un tipo di attesa binaria, attendo finchè il Mutex non è libero, le Condition Variables invece permettono di attendere anche su espressioni algebriche complesse prima di continuare l'esecuzione.

Esse non vanno a sostituire interamente i Mutex, ma vanno usate in cooperazione con essi per raggiungere lo scopo, anche perché soffrono del problema dei risvegli spuri, ovvero di un thread che viene rimesso in esecuzione anche se la condizione non è verificata.

La creazione tramite librearira standard di un thread in C++ avviene mediante richiamo di thraed nativi, e fa si che il therad venga aggiunto alla lista dei thread eseguibili.

All'interno di un oggeto thread troviamo un handle, un riferimento opaco al thread che ci permette di interrogarlo sul suo stato di esecuzione. Questo handle è diverso dal thread id, che è invece solo un identificativo.

Una differenza fra Windows e Linux è ad esempio che mentre in Windows per la gestione dei thread si utilizza un handle, in Linux si utilizza un file descrpitor (????)

Bisogna stare particolarmente attenti alle ececezioni lanciate all'interno di un thread in quanto se esse non vengono gestite al loro interno, butteranno giù l'intero processo.

In Rust è possibile far ritornare dei valori ad un thread. Infatti richiamando la funzione `spawn()` per generare un nuovo thread esso ci darà una `JoinHandle<T>`, che al momento della join ci darà il valore ritornato dal thread.

I thread in Rust possono essere richiamati direttamente tramite la funzione `thread()`, che esegue la funzione ad essa passata con le varie catture del caso, oppure si puo' usare un'interfaccia costurita con Builder Pattern per definire altri valori, come il nome del thread o la dimensione del suo stack.

```
use std::thread;
let builder = thread::Builder::new()
    .name("t1".into())
    .stack_size(100_000)
let handler = builder.spawn { || println!("Hello world!")}
```

Indovinate un po'? Anche la concorrenza in Rust è raggiunta mediante i tratti. Essi tuttavia sono spesso tratti `Marker` o marcatori, ovvero tratti che non implementano realmente una funzionalità ma indicano solo che l'operazione che si vuole eseguire è lectia.

Il tratto `Send` indica che lo scambio fra thread del tipo `T` è sicuro. Questo tratto potrebbe essere implementato dal compilatore se lo ritiene necessario, ma ovviamente essendo un tratto marcatore (`std::marker::Send`) la sua implementazione è nulla, appunto la marchia solo come tale.

La comunicazione fra thread richiede che ci si scambi tipi di dato `T` tale che `T` implementi `Send`, quindi le strutture standard che le implementano non possono essere scambiate, tipo `RC` (`ARC` invece lo implementa).

Il tratto `Sync` è simile. Più nello specifico, è possibile implementare `Sync` per un tipo `T` se e solo se `&T` implementa `Send`. Questo perché il tratto `Sync` indica tipi di dato per i quali è sicuro scambiarsi le referenze fra thread, da cui il vincolo. Anche questo tratto potrebbe venir implementato implicitamente dal compilatore.

Una conseqeuenza bizzarra di tale definizione è che `&mut T` è `Sync` nel moomento in cui `T` implementa `Sync`. I tipi non `Sync` sono generamente i tipi di dato che implementano la mutabilità interiore, tipo `Cell` o `RefCell`, in quanto potendo modificare il contenuto (in maniera non atomica) anche solo con una referenza non mutabile, questo comportamento non puo' essere accettato. Per tornare all'esempio di prima, essendo `&RC` non `Send`, `RC` non puo' implementare `Sync`.

Per la comunicazione fra processi Rust supporta due modelli: strutture dati condivise fra tutti i thread, sia in lettura che scrittura, e i canali di comunicazione verso uno o più destinatari. Esistono altre possibilità offerte da crate esterni di cui non parleremo qui.

Per condividere una struttura dati, dobbiamo assicurarci che mentre un thread la stia leggendo un altro non vi scriva sopra e ovviamente che non sia possibile più di una lettura contemporaneamente. Per raggiungere questo obiettivo, si utilizzano i `Mutex`.

Un `Mutex` esegue i passaggi logici di `lock()` della risorsa, con eventuale preventiva attesa se un altro thread la stesse usando prima della chiamata, operazioni sulla risorsa successive all'acquisizione del lock, rilascio del lock in modo che altri thread possano accedervi.

Internamente le implementazioni di `lock()` e `unlock()` (in Rust `unlock()` è realizzato come `drop(LockGuard)`) sono realizzate con barriere di memoria, ovvero garantiscono che la cache venga pulita prima delle operazioni in modo da avere dati sempre aggiornati e che i miei cambiamenti siano subito disponibili in memoria principale al momento del rilascio della memoria.

Va fatto notare che i lock non sono funzionalitò offerte da un linguaggio ma astrazioni accanto alle API di sistema. In altre parole, se un SO non esponesse queste funzionalità in primo piano, queste sarebbero irrealizzabili.

Se un thread terminasse mentre è in possesso del lock, non sarebbe possibile per gli altri thread acquisire la risorsa e continuare l'esecuzione. Per questo Rust come il C++ segue il paradigma *Resource Acquisition Is Inizialization* (RAII). Solitamente per le parole così brevi si omette una lettera dall'acronimo ma sono sicuro che la SIAE si sia lamentata agli sviluppatori di Rust per violazione del marchio.

Stando alla documentazione ufficiale di Rust:

(Image: *../imgs/20220614214956.png*)

Infatti il concetto dietro questo paradigma è che una risorsa vada inizializzata nel suo metodo costruttore e vada azzerata nel suo metodo distruttore. Rust fa si che questo paradigma sia rispettato per i lock in un modo molto semplice: per design del linguaggio, una funzione che giunge al termine esegue il `Drop` di tutte le variabili locali, tra cui i lock che nel linguaggio sono delle strutture di tipo `LockGuard<T>`. Al momento del `Drop` di `LockGuard`, viene automaticamente eseguito l'`unlock()` sul `Mutex` cui corrispondono, di fatto assicurandosi che non si possa mai lasciare il `Mutex` bloccato per sempre.

Notiamo ora come il tipo di ritorno di una `lock()` non sia direttamente una `LockGuard` ma una `LockResult`. Questo accade perché in Rust un `Mutex` puo' essere avvelenato.

Per essere più specifici, un thread che chiama `panic!()` mentre mantiene una `LockGuard` NON esegue il `Drop` di tale `LockGuard`, e come tale rende inutilizzabile il `Mutex.

Se ciò accade quindi la `lock` ritorna una `LockResult` che contiene un errore, ovvero che il `Mutex` non si sbloccherà mai. Allo stesso modo chiamare la `lock` su un `Mutex` mentre già lo si possiede, causerà un `panic!()`.

`Mutex` non implementa il tratto `Clone`. Per questo motivo, il suo utilizzo tipico è quello di essere avvolto attorno a una referenza `ARC`, perché dobbiamo scambiarlo fra thread, e poi clonare e distribuire questo `ARC` tra i vari thread.