05-25.md

1. Lezione del 25/05 - Concorrenza III

Un'alternativa ai `Mutex` in Rust è rappresentata dagli `RWLock`, che è funzionalmente simile ma con delle differenze interessanti.

I `Mutex` in Rust rappresentano una scelta piuttosto netta che solo una thread per volta possa eseguire le istruzioni successive all'acqusizione del lock. Questo è ottimale quando vogliamo a tutti i corsi costrutire una sezione critica, ma non molto efficiente in altri contesti.

Ad sempio, una risorsa puo' esser letta da più thread contemporaneamente senza rischi di consistenza di dato, sono le letture che vogliamo eseguire in completa esclusione, eppure i `Mutex` limiterebbero anche le letture ad un singolo consumatore.

Gli `RWLock` implementano proprio questa funzonalità, garantendo che molti thread possano leggere contemporaneamente dalla risorsa ma che quando uno tenta di scrivere sarà l'unico a starvi operando sopra.

Per far ciò si utilizzano due funzioni: `read()` e `write()`. Entrambe ritornano una `LockResult` identica a quella vista per i `Mutex`, con tanto di meccanismo di avvelenamento (per questo si usa una `LockResult`) in quanto le scritture potrebbero fallire. (Non perché è insicuro usarle, solo perché praticamente tutto puo' finire con una chiamata a `panic!()`).

Passiamo infine a discutere dei tipi atomici. Essi sono disponibili solo per i tipi di base, come `i32` o `bool` e simili, e forniscono funzioni per operazioni atomiche come l'incremento/decremento atomico, test-and-test-and-set, test-and-set e test-and-swap.

Tutte queste operazioni hanno come parametro una barriera di memoria che vogliamo andare ad utilizzare le operazioni.

I tipi atomici sono thread safe ma non offrono meccanismi di condivisione esplicita. Come tutti i valori di Rust, sono soggetti alla regola del possessore unico, e quindi per essere condivise hanno bisogno di essere avvolte in un `Arc`. Per offrire queste funzionalità questi tipi, similarmente a `Cell`, implementano mutabilità interna, ed è quindi possibile richiamare tali operazioni anche variabili immutabili.

Potremmo con essi realizzare un nostro spinlock come:

```
let spinlock = Arc::new(AtomicUsize.new(1));
let spinlock_clone = Arc::clone(&spinlock);
let thread = tread::spawn( move || {
    spinlock_clone.store(0, Ordering::SeqCst);
    };
)

while spinlock.load(Ordering::SeqCst) != 0 {
    hint::spin_loop(); // è un NOOP macchina
}
thread.join().unwrap();
```

Digressione personale su Ordering::*

Dalla documentazione ufficiale:

(Image: *../imgs/20220614222601.png*)

Essenzialmente gli enum definiti in `Ordering::` spiegano come verranno riordinate, sia in fase di compilazione che di esecuzione, le istruzioni macchina del nostro codice.

La versione breve è: - `SeqCst`: le istruzioni prima e dopo un'operzione `SeqCst` avverranno esattamente prima e dopo, NO riordino ma costo di esecuzione (invalida le cache) alto - `Acquire` o `Release`:

permettono un po' di riordino, sono pensate per essere usate in fase di acquisiione o rilascio dei lock. - `Relaxed`: permette il riordino di tutto ma un'operazione dichiarata come atomica sarà comunque eseguita in modo atomico.

Fine digressione personale

Nelle situazioni in cui ci si vuole fermare fino alla fine di un thread, un `Mutex` non è ottimale, in quanto non permette di interrompere un thread in polling o in busy waiting.

Non sapendo a priori quanto dovrò attendere, per non sprecare tempo di CPU potrei pensare dopo ogni controllo di aggiungere una `sleep()` di durata fissa ma predefinita, ma questo non è un approccio ottimale. In questi casi la soluzione migliore è l'utilizzo di una Condition Variable.

Esses sono delle astrazioni dei SO implementate dai linguaggi che permettono di bloccare l'esecuzione fino al raggiungimento di una condizione.

Per mettersi in attesa su una Condition Variable si utilizzerà una funzione `wait()`. Il thread a fare tare chiamata verrà inserito in una lista di thread in attesa (in Rust denominata `parking_lot`) e quando un altro thread noterà che la condizione è esaudita risveglierà uno o più thread in tale lista tramite le funzioni `wake_one()` o `wake_all()`.

Notiamo come sia possibile svegliare più thread anche in un contesto dove vogliamo comunque ottenre una sezione critica. Per questo motivo le `wait()` sono solitamente implementate in un ciclo per verificare che la condizione per cui si è stati svegliati sia ancora valida al risveglio. Tale meccanismo è solitamente raggiunto abbinando un `Mutex` alle `Condvar`. Un altro motivo per fare ciò è che con le Condition Variables possono accadere risvegli spuri:

Supponiamo che il thread principale debba attendere che la condizione di una variabile booleana campi. A quel punto, il codice sarà:

```
let lock = Mutex::new(true);
let cvar = Condvar::new();
let lock_arc = Arc::new(lock);
let cvar_arc = Arc::new(cvar);

thread::spawn(move || {
   let mut started = lock_arc.lock().unwrap();
   *started = true;
   cvar_arc.notify_one();
})

let mut go = lock.lock().unwrap();
while !*go {
    cvar.wait(&mut go).unwrap();
}
```

Notiamo come il controllo `!*go` sia in un loop per i motivi sopra descritti.

Inoltre, non solo è importante che ad una `Condvar` sia sempre associato un `Mutex`, ma è anche importante che vi sia associato **un solo** Mutex, altrimenti avremmo problemi di sincronismo.

Se non facessimo questo tipo di controllo in modo esplicito, creeremmo dei bug software difficili da identificare. Capita raramente, ma capita.

Sia C++ che Rust offrono forme di `wait()` più articolate rispetto a queste più semplici, tipo `wait_while()` che accetta per parametro una lambda che viene eseguita ad ogni risveglio per verificare la condizione, che probabilmente è il meccanismo che il nome Condition Variable suggerisce. In Rust quando questa lambda ritorna `true` il thread torna a dormire, altrimenti viene risvegliato.

Esistono infine dei metodi per l'attesa vincolata, vale a dire che permettono di mettersi in attesa per un tempo massimo alla fine del quale il thread continuerà comunque la sua esecuzione.