Esercitazione 6

Esercizio 1: barriera ciclica con canali

Implementare la barriera utilizzano dei canali per la sincronizzazione fra i thread, senza uno stato condiviso in un mutex. L'idea per usare i canali è: se la barriera ha dimensione n immaginiamo che essa contenga n oggetti Waiter che possa distribuire agli n thread che hanno necessità di sincronizzare il proprio funzionamento. Tali oggetti contengono n-1 ingressi di canale e un singolo ricevitore. Essi offrono il metodo wait() che, quando viene invocato, invia un singolo messaggio su ciascuno degli ingressi presenti al suo interno e poi si mette in attesa di ricevere altrettanti messaggi dal proprio ricevitore.
La barriera così realizzata avrà n canali e ogni Waiter avrà un receiver dedicato e n-1 sender. Quindi anziché condividere la barriera tra i thread, converrà fornire loro un oggetto Waiter, che include il receiver dedicato al thread e gli n-1 sender verso gli altri thread.
Occorre proteggere in qualche modo la struct CyclicBarrier? Perché?

Codice di esempio di invocazione:

```
fn main() {
    let cbarrrier = cb::CyclicBarrier::new(3);

    let mut vt = Vec::new();

    for i in 0..3 {
        let waiter = cbarrrier.get_waiter();
        vt.push(std::thread::spawn(move || {
            for j in 0..10 {
                waiter.wait();
                println!("after barrier {} {}", i, j);
            }
        }));
    }
```

Esercizio 2: Threadpool

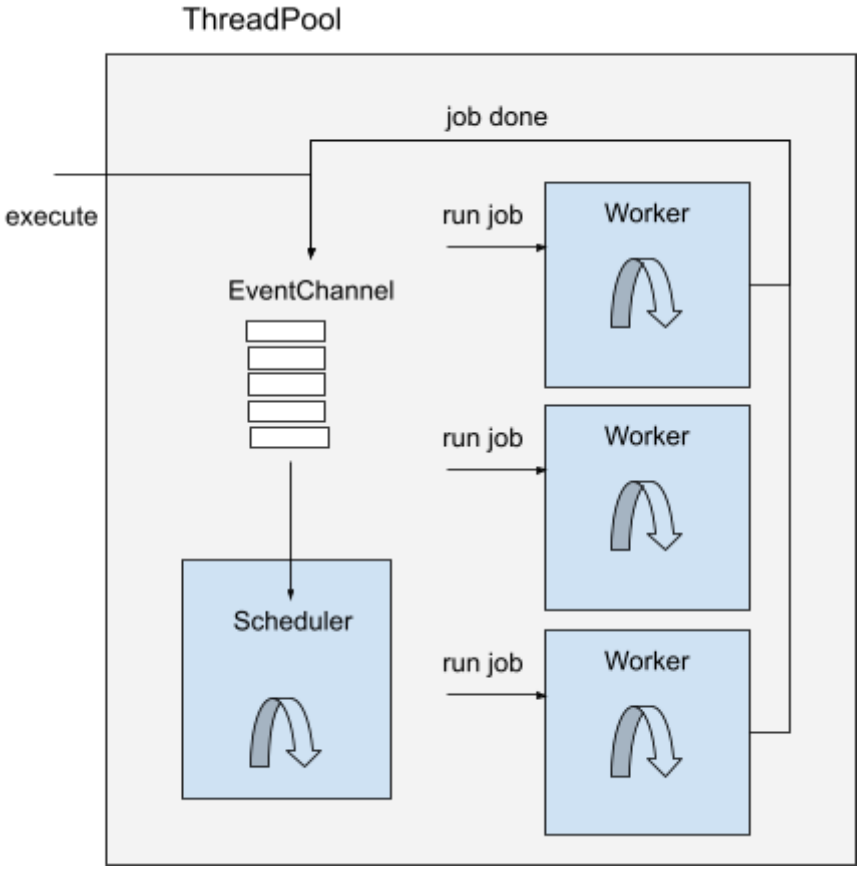
Un Thread Pool è una struttura che alloca un numero fisso di thread dentro i quali esegue dei job ricevuti dall'esterno tramite un metodo execute così definito:

```
impl ThreadPool {
    execute(&self, job: Box<dyn FnOnce()>) {}
}
```

La execute si comporta come la thread::spawn, ritornando subito ed eseguendo il job in un differente thread, con le seguenti differenze:

- non fa partire un nuovo thread, ma invia il job ad un worker con un thread già allocato e sempre in attesa di nuovi job da eseguire
- se tutti i worker sono occupati il job viene accodato in attesa di essere eseguito appena un worker si libera
- il job non restituisce nulla, ma viene eseguito in modalità *fire and forget*; nel caso in cui il job dovesse restituire un risultato è possibile inserire nel job stesso un meccanismo di ritorno (es. una barriera con valore, un canale)

Un possibile schema di realizzazione un thread pool è il seguente



La execute attraverso un canale invia un job da eseguire allo scheduler interno, che viene eseguito in un thread dedicato. Lo scheduler controlla se vi sono worker liberi, appena vi è un worker libero gli invia il primo job in coda. Ogni volta che un job viene terminato lo scheduler controlla se vi sono job in attesa e li invia ad un worker libero.

Alcuni suggerimenti:

- Inserire le funzioni in un Box, in modo che possano essere memorizzate in collezioni e inviate sui canali.
- lo scheduler deve essere svegliato da due tipi di eventi differenti, generati in posti differenti: quando arriva un nuovo job e quando un worker ha finito, per poter schedulare un nuovo job. Avere due cv porterebbe a deadlock o complicherebbe moltissimo la realizzazione: se sono in attesa su un cv per avere nuovi job, ma nel frattempo finisce un worker non mi sveglierò finché non arrivano nuovi job, ma magari ne ho altri da schedulare. Una soluzione è usare un canale che chiameremo

EventChannel: chiunque deve svegliare lo scheduler scrivere un evento sul canale e lo scheduler è sempre solo in attesa di eventi.

- gli eventi di tipo differente sullo stesso canale si possono codificare utilizzando una enum (es enum JobMessage {NewJob(Box<...>), WorkerDone(usize) } (WorkDone riceve l'id del worker libero)
- Quindi quando lo scheduler riceve un evento controlla il tipo:
    - se è un nuovo job e c'è un worker libero glielo passa, altrimenti lo memorizza in una coda
    - se è un worker che ha finito controlla se ci sono job in attesa e, nel caso, glielo passa
- scheduler e worker devono condividere: un canale per ricevere i job, l'EventChannel affinché i worker comunichino che hanno finito

Esempio di utilizzo del thread pool.

```
fn main() {
    // alloca i worker
    let threadpool = ThreadPool::new(10);

    for x in 0..100 {
        threadpool.execute(Box::new(move || {
            println!("long running task {}", x);
            thread::sleep(Duration::from_millis(1000))
        }))
    }

    // just to keep the main thread alive
    loop {thread::sleep(Duration::from_millis(1000))};

}
```

Bonus: modificare il thread pool aggiungendo il metodo stop() che ferma tutti i worker e ne attende la conclusione, eseguendo ogni eventuale job già accodato.

Esercizio 3: processi

Uno dei motivi per cui spesso conviene demandare del lavoro a dei processi figli anziché a dei thread è che sono "isolati", se escono in modo inaspettato o vanno in panic non hanno effetti collaterali sul programma chiamante. Inoltre se si sbloccano o ci sono delle operazioni troppo lunghe possono essere interrompibili.

Quindi quando si fanno operazioni di rete la cui durata è impredicibile (un sito può essere

lento, una connessione tcp può stare aperta, ma bloccata per ore) può essere conveniente creare un processo figlio per farle e interromperle se richiedono troppo tempo.

In questo esercizio voglio fare un oggetto Downloader, che riceve una url da scaricare e un timeout, oltre il quale se non ha finito deve interrompersi per liberare risorse.

Esempio di uso:

```
let dowloader = Downloader::new(“http://www.google.com”, 10);
match downloader.start() {
      Ok(data) => // save downloaded data
      Err(e) => //print error type (timeout, other)
}
```

L’oggetto Downloader nella start dovrà:

- lanciare come figlio il comando curl (scaricarlo se necessario) e passare come parametro la url da scaricare
- rimanere in attesa dell’output, da leggere con una pipe, che è la url scaricata
- far partire un thread che controlla il timeout e se passa troppo tempo chiama la kill() sul figlio
- restituire un risultato corretto.

Bonus: potete utilizzare il threadpool dell’esercizio precedente per eseguire n downloader in parallelo ed essere sicuri il threadpool non abbia mai worker bloccati in operazioni troppo lunghe.