Esercitazione 5

7-9 giugno 2023

Esercizio 1 - Threadpool

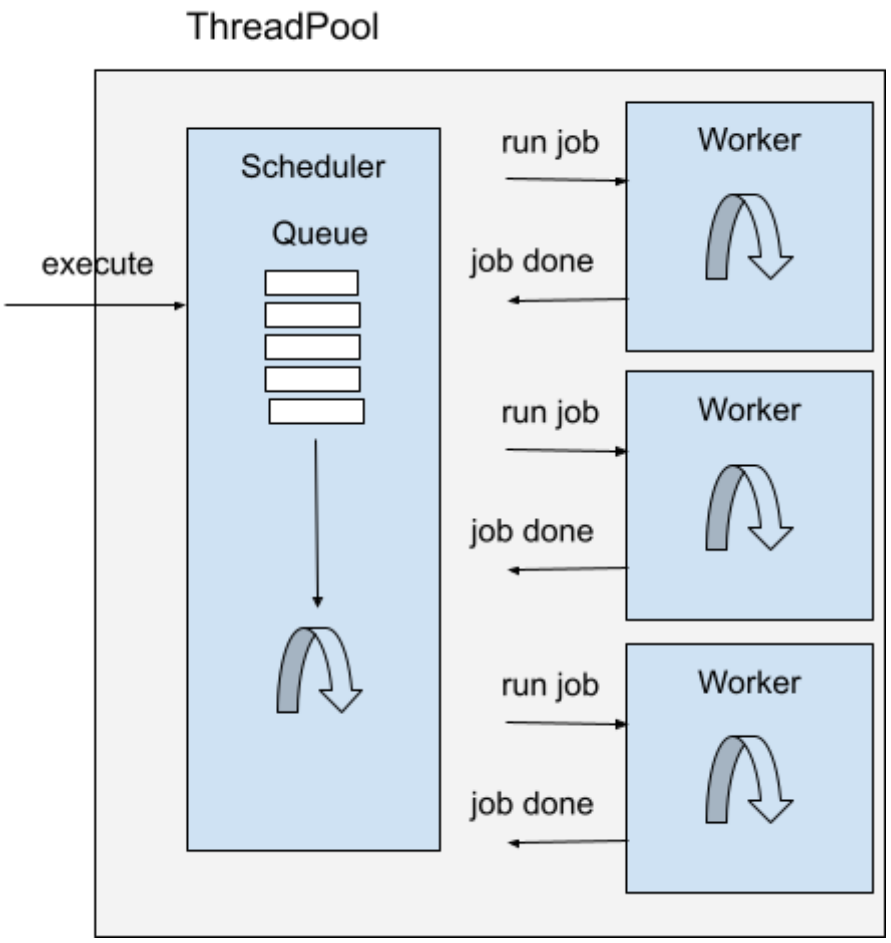
Un Thread Pool è una struttura che alloca un numero fisso di thread dentro i quali esegue dei job ricevuti dall'esterno tramite un metodo execute così definito:

```
impl<F: FnOnce()->() + Send + 'static> ThreadPool<F> {
        execute<F>(&self, job: F) {}
}
```

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 ob 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:

- lambda che implementano i trait FnOnce() -> () + Send + ‘static possono essere inviate su canali e memorizzate in collezioni
- lo scheduler deve essere svegliato da due tipi di eventi differenti: quando arriva un nuovo job e quando un worker ha finito. Si possono utilizzare canali o una condition variable; se si usano più canali per questi due eventi guardare la funzione **select** di crossbeam, che permette di fare una receive su più canali contemporaneamente; altrimenti è possibile pensare di inviare eventi di tipo differente sullo stesso canale utilizzando una enum (es enum JobMessage<F:...> {NewJob(F), JobDone }
- Per controllare se un thread è libero lo scheduler può tenere traccia al suo interno dei worker occupati, alternativamente può condividere con ciascun worker un AtomicBoolean che indica se il worker è libero o meno.

Esempio di utilizzo del thread pool.

fn main() {

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

    for x in 0..100 {
        threadpool.execute(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))};

}
```

Dopo aver realizzato il thread pool provare ad utilizzarlo con un problema in cui è necessario raccogliere i risultati, ad esempio per il risolvere il gioco delle permutazioni della esercitazione 3.

Esercizio 2

Realizzare un semplice interprete comandi con il seguente funzionamento:

- in un loop infinito il programma stampa un prompt (“>”) e rimane in attesa di leggere da standard input un comando (es “ls /”)
- esegue questo comando in un processo figlio, raccogliendo l’output del comando su una pipe e stampandolo sullo standard output del terminale
- mentre il comando è in esecuzione se utente scrive qualcosa sullo standard input i caratteri letti vanno inviati allo standard input del comando stesso (alcuni comandi come “cat” rimangono in attesa di input e non terminano finché non viene chiuso lo standard input)
- quando il comando termina il controllo viene di nuovo ceduto al terminale e ristampa il prompt in attesa di un nuovo comando

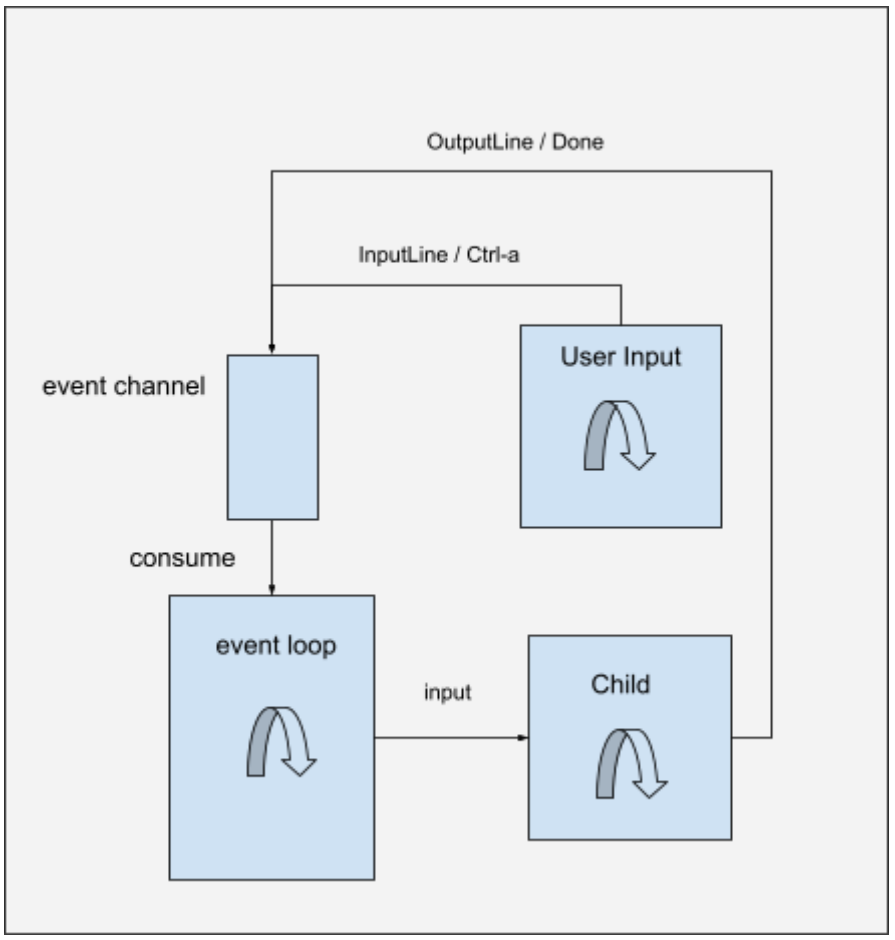
L’interprete quindi è sempre in una di queste due modalità:

- attesa comando: in questo caso lo standard input da tastiera viene utilizzato direttamente per leggere il comando da eseguire
- esecuzione comando: in questo caso l’input dell’utente viene girato all’input del comando

Il processo figlio termina in due casi:

- quando finisce il processo figlio, che chiude lo standard output
- quando l’utente preme la sequenza **ctrl-a** (byte con codice “1”), che una volta letta causerà la chiamata di child.kill() (alcuni comandi come “cat” infatti non terminano finché non li termina l’utente)

Per evitare deadlock con il processo figlio in esecuzione è meglio leggere l'output del comando e scrivere sull'input in due thread diversi. Altrimenti potrebbe accadere che mentre il processo figlio sia fermo in attesa di input, il padre sia fermo in attesa di output del figlio. Inoltre è preferibile che anche l'input dell'utente sia raccolto su un thread dedicato, in modo da non interferire con la comunicazione con il processo figlio.
Un possibile approccio per gestire in modo ordinato la comunicazione fra thread scorrelati è realizzare un loop di eventi come nella figura.



- Il main del programma è un loop che consuma eventi da un canale
- gli eventi sono dei messaggi con i dati prodotti dagli altri thread; nel caso specifico noi vogliamo essere notificati quando l'utente scrive dell'input, preme ctrl-a, il figlio produce dell'output o termina
- i thread che raccolgono gli eventi (user input e child output/status) mandano eventi sul canale
- per ogni evento ricevuto il loop esegue l'opportuna azione, es: lancia un comando e il thread per raccogliere l'output, passa l'input al figlio, termina il figlio, scrive l'output ricevuto dal figlio, ritorna in attesa di un nuovo comando

Suggerimenti:

- usare una enum per definire tutti gli eventi e i dati ad essi collegati
- per passare ad un thread differente stdin/stdout del figlio usare **child.stdin.take()** -> stdin è di tipo Option e take() toglie stdin dall'oggetto child, lasciandolo in uno stato "pulito", ovvero scrivendo None al posto di stdin. Ciò permette di muovere separatamente child / stdin / stdout, altrimenti si avrebbero più riferimenti mutabili a child.
- se nella vostra implementazione fosse necessario memorizzare un oggetto Child che può essere o non essere inizializzato, usare Option<Child>
- anche se per alcuni comandi vi possono essere problemi, potete assumere che input/output siano sempre leggibili per linea con un BufferedReader

Esempio di interazione su linux (in grassetto l'input utente):

> **ls**
Cargo.toml src …
> **cat**
1234
1234
crtl-a
> **cat Cargo.toml**
[.... contenuto file]
> **ctrl-a**
exit