06-06.md

1. Lezione del 06/06 – Comunicazione fra Thread

Esistono varie tecniche per comunicare fra thread, inteso come il vero e proprio scambio di informazioni o messaggi a differenza di informazioni di sincronizzazione.

Il principio è quello di implementare un pattern Produttore-Consumatore, dove un thread produce dei dati e li conserva ed un altro li riprende e li consuma.

Questo approccio porta due grosse problematiche. La prima è l'ordinamento, ovvero garantire che i messaggi vengano elaborati nello stesso ordine con cui siano stati prodotti (o con un ordine predefinito). Questo proprietà non è del tutto garantita, perché come vedremo i canali implementano una politica di estrazione dei messaggi a coda, quindi è sicuro che avremo un'estrazione dei messaggi nell'ordine di arrivo ma non una loro elaborazione nell'ordine di arrivo.

L'altro problema è il capire che la comunicazione è terminata e che quindi eventuali ascoltatori possono smettere di attendere nuovi messaggi. Dobbiamo interrogarci anche del caso opposto, quello in cui la produzione è continua e veloce mentre il consumo dei messaggi non riesce ad essere altrettanto rapida.

Rust mette a disposizione due strumenti per questi casi. Il primo sono i canali, composti da un trasmettitore e un ricevitore. Il trasmettitore implementa il tratto `Clone`, vale a dire che puo' essere clonato e distribuito fra thread in modo che il creatore del canale possa ricevere messaggi da più fonti, mentre il ricevitore no.

Il ricevitore puo' ricevere un messaggio, sotto forma di `Result', il cui campo`Error`indica che il trasmettitore si è disconnesso. Nel caso tipico, dove il trasmettitore viene clonato e distribuito, che tutti i trasmettitori sono stati disconnessi ( ricordate di fare il`drop()` del trasmettitore nel main thread…).

Dall'altro lato la situazione è più o meno simile. Anche la `send` operata sul trasmettitore ritorna una `Result`, ma in questo caso si ottiene errore solo se il nostro ricevitore si è disconnesso. Una sottigliezza sta ad indicare che una `Result` di tipo `Ok` non indica che il dato è stato ricevuto oppure che verrà ricevuto, ma solo che è stato inviato. Dal lato opposto, un `Error` indica che il dato non verrà mai ricevuto in quanto il ricevitore si è disconnesso.

Vediamone un esempio:

```
use std::sync::mpsc;
let (tx, rx) : (i32, i32) = mpsc::channel();

for _ in 0..5 {
    let tx_clone = tx.clone();
    std::thread::spawn( move || {
        tx_clone.send(1).unwrap();
        // drop(tx_clone); automatico
    });
}

drop(tx);

while let Ok(x) = rx.recv() {
    println!("{}", x);
}
println!("All txs are done");
```

Notiamo che eseguiamo manualmente il `drop` del `tx` originale. Questo perchè se così non facessimo, `rx.recv()` rimarrebbe in ascolto perenne, in quanto non tutti i `tx` hanno eseguito il `drop`.

Un alternativa a questi canali, detti asincroni, sono i `sync_channel`. Si differenziano dai canali appena visti in quanto garantisce che i messaggi vengano estratti nello stesso ordine con cui sono stati inviati (ma qual è l'ordine con cui sono stati inviati?) ed accetta in fase di instanziazione un parametro `bound`. Tale parametro è un `usize` che limita il numero di messaggi in coda, il che significa che una `send` quando in coda vi sono `bound` messaggi già in coda causerà l'attesa fino a quando non si sarà liberato un posto.

Due casistiche particolari sono rappresentate dai valori di `0` e di `1` per il parametro `bound`. Il primo infatti rende il canale un canale di rendez-vous, dove la chiamata a `send` non esegue ritorno fin tanto che sul messaggio inviato non viene eseguita una `recv`, mentre il parametro `1` indica che ad ogni scrittura debba necessariamente alternarsi una lettura.

Passiamo ora dal parlare di thread a parlare di processi. Come forse abbiamo già visto in altri corsi, la cominicazione fra processi è nettamente più difficile della comunicazione fra thread. Lo stesso vale per la coordinazione.

Alcuni problemi non sono possibili da risolvere solo con thread, anzi l'approccio tipico alla parallelizzazione è proprio di aver un processo padre che genera più processi figlio che a loro volta andranno a generare più thread, al fine di sfruttare tutte le capacità hardware del computer.

In ogni SO è presente un Process Control Block, che mantiene informazioni utili al processo non solo in termini di scheduling ma anche di isolamento. Questo perchè in un cambio di contesto è necessario ad esempio salvare i registri della MMU per la protezione degli spazi di indirizzamneto.

Per far comincare due processi tutti i SO mettono a disposizione dei meccanismi di Inter Process Communication, IPC. Questo richiede che in qualche modo i processi possano condividere fra loro alcune zone di memoria.

Dal punto di vista della parallelizzazione non è detto che un problema debba per forza esser scomposto in thread. Una pratica comune nei web-servers ad esempio è creare un processo per gestire la singola richiesta.

L'API per la creazione di processi in Windows ha molti parametri. La funzione `CreateProcess` non disitngue tra la creazione di un processo interno a un processo o l'avvio di un processo per un programma. Al contrario la `fork()` in Linux non accetta alcun parametro e ha come valore di ritrno un solo identificativo numerico.

Un altro modo per creare processi in Linux è la famiglia di funzioni `exec` che esiste in varie versioni a seconda che il tipo di parametri che gli si voglia passare.

Linux all'atto della duplicazione, nel tentativo di risparmiare memoria, attua meccanismi di Copy on Write. A seconda del processo cui ci si trova, il valore di ritorno cambia. Infatti il padre avrò come valore di ritorno il PID del figlio e il figlio avrà come valore di ritorno 0. Qui Savino ha fatto una battuta? Ragionamento? Sul fatto che a chiedere la duplicazione dei processi in un processo figlio sono i processi genitori, "al limite del filosofico" ha detto. Vabbè.

Anche per terminare un processo le alternative sono molteplici. Sia Linux che Windows espongono delle primitive per terminare un processo, rispettivamente `_exit()` e `ExitProcess()`, che accettano compe parametro un univo valore numerico che sarà il valore di ritorno del processo. Questo identificatore serve a notificare altri processi sulla correttezza o meno dell'esecuzione del processo.

Rust permette la creazione di processi con una certa distizinione del SO. Per lanciare un nuovo processo si puo' utilizzare la liberia `Command`, usando sia un builder pattern che un unica chiamata acon un vettore di elementi che sono il nostro comando.

```
let output = Command::new("ls")
    .arg("-l")
    .arg("-a")
    .output()
    .expect("failed to execute process");
```

```
let output = Command::new(["ls", "-l", "-a"])
    .output()
    .expect("failed to execute process");
```

Queste due chiamate sono equivalenti e, a seconda del grado di funzionalità che voglio raggiungere, posso anche abilitare una IPC basata sul piping (operatore | in Linux) in modo che ogni processo legga da standard input e legga su standard output, con l'input di un processo e l'output di un altro che sono in realtà la stessa risorsa.

Potrei anche in alcuni contesti due processi figli, il secondo senza controllare che la generazione del primo sia già avvenuta. Questo torna utile nel momento in cui voglia avere due processi indipendenti, come nel caso in cui abbia due algoritmi, uno preciso ma più lento ed uno più veloce ma approssimato. In quel caso se dopo un certo tempo predefinito l'algoritmo preciso non è terminato, accetterò per buona la soluzione approssimata e chiderò l'interruzione del primo figlio mediante `kill <pid>`.

Per quanto riguarda il valore di ritorno, le varie funzioni di `exit` accettano come parametro un identificativo numerico che indica le modalità con cui il processo è terminato. Non siamo obbligati a fare ciò in quanto altre versioni di `exit`, nominalmente le `abort`, non richiedono tale parametro.

Naturalmente sono posisbili anche le `wait`, che sono disponibili in due modalità. La prima è la `wait` casuale, dove non specifico il pid su cui attendere e quindi mi basta che uno degli N processi che ho generato termini per continuare l'esecuzione, mentre la seconda, detta anche `waitpid`, attende che il processo identificato dal `pid` termini.

Infatti la `wait` generica in Linux ci fa gestire due parametri. Il primo, che è il valore di ritorno, è il `pid` del processo che ha terminato l'esecuzione, il secondo, passato come riferimento in memoria, è l'exit code del processo che ha terminato.

Sempre su Linux, il valore di ritorno è un intero a 16 bit, poichè composto da due valori a 8 bit. Gli 8 bit meno significativi indicano la modalità di uscita, quindi se tramite `exit` o tramite `abort`, mentre i bit più significativi indicano l'exit code. Se gli 8 bit meno significativi non sono tutti 0, vuol dire che il processo è terminato con un errore.

La gestione dell'errore dipende dal SO. Su una piattaforma Windows ad esempio se il padre termina prima del figlio a quest'ultimo viene dato come nuovo padre il processo di init di sistema. Questo comporatmamento viene anche sfruttato in maniera voluta in modo che un processo termini solo allo spegnimento del computer.

Quando accade il contrario invece, il processo figlio diventa uno zombie, il che significa che la sua esecuzione è stata arrestata e le sue informazioni nella tabella dei prcoessi verranno pulite solo quando il processo padre andrà a raccogliere il suo valore di uscita.

Nelle comunicazioni fra processo la scelta tipica è quella di una cominicazione strettamente binaria e orientata al byte. Una prima per fare ciò è scegliere un numero fisso di byte, e indicare come messaggio una sequenza lunga tale numero di byte, quindi andrò a scrivere e leggere N byte per volta.

Questo va bene per tipi di dato semplice ma non per tipi di dato complesso, come esistono in linguaggi di alto livello e in particolar modo in Rust. In questo caso le soluzioni più frequenti si basano sull'utilizzo di standard noti, come XML o JSON. In questi casi il dato viene serializzato e si indica nei primi N byte del messaggio la lunghezza del dato seriallizzato, che è quello che andrà letto.

Le pipe sono flussi di dato unidirezionali, quindi per una comunicazione bidirezionale bisogna avere due pipe tra due processi, e questo crea problemi nella coordinazione tra problemi.

A prescindere dal SO, il concetto di pipe è molto simile. Essi sono funzionalmente simili a dei file, e quindi come tali vanno aperti e chiusi correttamente. È necessario attuare la chiusura manuale di una pipe in quanto altrimenti avremmo la certezza della chiusura del file ma non lo svuotamento del buffer ad essa associata. Infine, teniamo a mente che sono file binari.

Rust permette l'accesso alle pipe in modo leggermente più semplice se paragonato agli standard del C. Nello spefifico possiamo definire lo standard input e lo standard output del processo come delle pipe in modo da poter leggere e scrivere su di essi.