RIASSUNTI RUST – PROGRAMMAZIONE DI SISTEMA

NOVARESE DANIELE

CONCORRENZA

Concorrente = programma che dispone di due flussi o più di esecuzione (più di un singolo thread)

- Un processo ha uno spazio di indirizzamento ben definito e indipendente da tutti gli altri
- Ogni thread di un processo ha il suo stack, collocato sempre nello spazio di indirizzamento del processo
- **Tutti i thread di un processo condividono:** *costanti + variabili globali + heap + codice*
    - **Lo stack è unico per ogni thread**
- *un thread è identificato da un suo id, contesto, status, spazio di indirizzamento stack.*

Thread nativi > thread gestiti dal Sistema operativo – permettono la creazione, identificazione e attesa del completamento. (no cancellazione)

Vantaggi concorrenza > vera parallelizzazione del programma – maggiore complessità possibile dovuta dalla divisione del carico computazionale – necessita però di sincronizzazione, soprattutto nel caso in cui i dati condivisi tra più thread sono mutabili e quindi possono cambiare durante l'esecuzione.

- L'esecuzione parallela e concorrente non esiste nei sistemi single-core; *è una semplice astrazione, poiché ogni thread creato verrà eseguito in modo sincrono.*

3 aspetti fondamentali:

- **Atomicità >** stabilire le operazioni che hanno effetti sincroni su tutti i thread – non possono essere interrotte. (*una eventuale modifica di un dato condiviso tra più thread deve essere fatta in modo atomico, così da non essere interrotta da nessuna operazione e così che i cambiamenti si propaghino in tutti i thread in modo uniforme e prima di qualsiasi altra istruzione)*
    - ***L'atomicità di una istruzione garantisce che funzioni di read / write / modify non saranno né osservabili né interrompibili finché non terminate..***
- **Visibilità >** stabilire quali dati sono visibili tra più thread e come sono gestiti
    - **Mutex**, dati condivisi tramite lock()
- **Ordinamento >** stabilire tramite la sincronizzazione come vengono effettuate le operazioni su dati da parte dei diversi thread
    - **Condition variable >** far sì che tutti i thread giungano ad un punto comune prima di procedere

Sincronizzazione

In un contesto multithread e concorrente è necessario utilizzare la sincronizzazione, per avere un programma che si comporta secondo un modo stabilito.

- *Evitare quindi "corse critiche" ai dati, ovvero due thread che accedono allo stesso dato in modifica contemporaneamente*
- Laddove la concorrenza permetta di salvare risorse condividendo ad esempio librerie o dati tra più thread, è **necessario garantire che due thread non modifichino la stessa variabile insieme – o che un thread acceda ad un dato mentre è ancora in stato di modifica.**

COME IMPOSTARE UN THREAD

Creazione semplice

```
                                creo il thread e ci passo una funzione
let thread_join_handle = thread::spawn(move || { //move trasferisce alla funzione
                                                 //il possesso di quanto catturato
    //computazione da eseguire

});
```

Creazione specifica dell'oggetto thread specificando parametri aggiuntivi (*nome, stack size…***)**

```
let builder = thread::Builder::new()
    .name("t1".into())
    .stack_size(100_000);

let handler = builder.spawn(|| { /* codice */}).unwrap();
```

SEND >>

I tipi di dato che implementano il tratto SEND garantiscono che il dato potrà essere condiviso in sicurezza tra più thread.

Sono passati tra thread tramite una *move* o una *copy*

SYNC >>

I tratti che implementano SEND implementano anche SYNC – questo tratto è utilizzato per permettere a più thread di accedere contemporaneamente ad un dato – può anche essere un indirizzo.

- Solo riferimenti NON mutabili

MUTEX > Per condividere un dato tra più thread e far sì che uno di questi possa modificarlo (uno per volta)

- *Mutual exclusion lock* > permette tramite l'uso di lock() / unlock() di far sì che un solo thread alla volta acceda al dato protetto dal mutex.
- **Fa busy waiting** >> processi sprecano cicli CPU e batteria mentre sono in attesa di ricevere il lock().
- **Implementa il paradigma RAII >** alla fine dello scope della funzione attuale, in cui si è ottenuto il lock() di una risorsa, questo verrà automaticamente rilasciato.

Un mutex per essere condiviso tra più thread deve usare lo smart pointer Arc<>

- Questo perché l'operazione di aggiornamento dei puntatori dell'Rc deve essere atomica
    - *Se non lo fosse, ci potremmo trovare facilmente in un contesto non-definito in cui un thread aggiorna un contatore mentre un altro prova ad accedere al puntatore*

La combinazione quindi di Arc (Atomic Reference Count) e Mutex permettono di condividere una risorsa in modo atomico e quindi sicuro tra più thread.

- Al posto di Mutex posso usare anche RwLock(), *che fa sì che tutti possano accedere alla risorsa quando non è in scrittura, e solo uno quando è necessario scrivere.*

ATOMIC >

Permette la sincronizzazione in un contesto concorrente di dati elementari, quali bool, interi, puntatori.

Si basa sull'utilizzo di tecniche quali *test-and-set* e *compare-and-swap* per far sì che le operazioni su di essi siano atomiche, e quindi non interrompibili ed eseguibili solo una per volta.

CONDITION VARIABLE >>

Permettono di introdurre un'attesa condizionata, per far sì che quindi un programma multithread possa avere un punto "comune" a cui tutti i thread devono giungere prima di poter proseguire.

Viene fatto tramite l'uso di Condvar e un Arc<Mutex>.

- Le condition variable creano un canale in cui è possibile mettere in attesa *wait()* i thread finché il lock che devono acquisire non sarà disponibile, risparmiando così cicli macchina CPU e batteria (*non fa busy waiting)*
    - Quando il lock sarà disponibile si userà *notify_one / notify_all* per svegliare uno / tutti i thread in attesa -> solo uno accederà al lock, gli altri torneranno in *wait()*

```
// Wait for the thread to start up.
let (lock, cvar) = &*pair;
let mut started = lock.lock().unwrap();
while !*started {
    started = cvar.wait(started).unwrap();
}
```

Per mettere quindi un thread in attesa, assegno al lock cvar.wait(lock)

Wait() è un metodo base, ma per evitare le notifiche spurie (notifiche false che risvegliano erroneamente il thread) si può usare wait while(….) insieme alla condizione che si vuole verificare – così anche se il thread si sveglierà, non procederà comunque e tornerà in wait.

CONDIVISIONE DI MESSAGGI – CANALI >> *mpsc::channel*

Oltre alla condivisione di dati tra più thread, è possibile creare dei canali di comunicazione basati sul paradigma *multiple producer – single consumer.*

- Channel() >>> ***sender*** *(1+) , receiver (1)*
    - Il sender deve introdurre il tratto Send
    - **Può essere clonato** (anche il dato contenuto deve introdurre il tratto Clone)

La condivisione su canali può essere sia sincrona, che senza limiti.

- La condivisione su canali sincrona ha un limite massimo di messaggi interscambiabili, che una volta raggiunto blocca il sender() in attesa che venga rimosso un messaggio dal canale.
- La condivisione su canali “normali” non ha limiti di quanti messaggi si possano scambiare.

Tre diversi paradigmi di comunicazione tra canali

- **Fan-Out / Fan-In >** *distribuisco I dati a più worker, e poi raccolgo il risultato in un singolo consumer.*
    - **Diversi thread mandano messaggi, un unico thread li riceve.**
- **Pipeline >** *il dato viene passato da un thread all’altro, ognuno dei quali lo elabora.*
    - **Inizialmente ho un unico canale > dentro ad ogni thread ne creo un altro, che invece di mandare i dati al ricevitore ricevuto come input, li manda a quello appena creato, che li manderà poi al ricevitore ricevuto come input**
- **Producer / Consumer >** *uno o più thread produttori generano dati, che vengono elaborati dal primo consumer disponibile.*

- **Simile a fan-out / fan-in > ho solo più consumatori, quindi i dati saranno elaborati più in fretta (al costo di utilizzo di più core)**

PROCESSI

Sono l'entità base di esecuzione di un programma

- Un processo è formato da uno o più thread (uno solo nel momento della creazione)
- Un processo ha un **process ID** e **uno spazio di indirizzamento indipendente da tutti gli altri processi**
- I processi possono condividere tra loro dati come librerie e handle a file (**IPC, inter process communication)**
    - Possono condividere anche dati, coordinando sempre l'accesso l'uno con l'altro.
    - Lo scambio di dati necessita che questi siano standardizzati, ovvero resi comprensibili da chi li riceve (concetto molto simile usato per device controller – driver) – si usano formati intermedi di testo (JSON, XML ,CSV) o binari (XDR, HDF)
        - **Esportazione -> MARSHALLING (*serializzazione*)**
        - **Importazione -> UNMARSHALLING (*de-serializzazione*)**

- ***Su windows*** >> processi completamente indipendenti – la creazione di un processo all'interno di un altro non lo rende "figlio"
- ***Su Linux*** >> un processo può avere dei figli – questi condividono tutti i dati del processo padre, ma secondo il paradigma COW (Copy on write)
    - *Tutti i dati sono condivisi nella loro versione originale – ne viene fatta una copia solo qualora verranno effettuate modifiche da parte di uno dei processi.*
    - La creazione di un processo figlio avviene tramite la funzione **fork()**
        - **Essendo che utilizzano il paradigma Copy-on-write, i processi figli non saranno sincronizzati con il processo padre >** *problema per concorrenza*

Terminazione di un processo >> **un processo termina quando…**

- **la funzione principale ritorna**
- **si verifica un'eccezione non gestita nel thread principale >** ExitProcess() / exit()

Entrambi restituiscono un codice di ritorno >> non ha un valore "sensato" in sé, ma è un valore attribuito secondo degli standard concordati.

IMPLEMENTAZIONE IN RUST

Struct Command >>

- **command::new(…)** crea un nuovo processo che esegue la funzione specificata come argomento
- sono possibili metodi aggiuntivi per specificare parametri:
    - arg / args >> argomenti da passare
    - env(..) >> variabili di ambiente
    - env_remove(…) >> rimuovere variabili di ambiente
    - stdin / stdout / stderr >> gestire i flussi di entrata, uscita ed errore.
- **Il processo si lancia con…**
    - Status() => Result<ExitStatus>
    - Spawn() => {} (non attende terminazione)
    - Output / wait_with_output => Result<Output>
- **Il processo si termina con…**
    - Kill() => {}
    - Exit(code: i32) => interrompe esecuzione, non richiama distruttori
    - Abort() => interruzione anomala, non richiama distruttori
    - **Panic!() => termina il thread corrente, richiamando prima tutti i distruttori.**
        - **Se il thread è quello principale, termina il programma.**

PROCESSI ORFANI >> processo padre termina prima di processi figli >> sistema assegna a tutti i processi figli come process ID del padre 1

PROCESSI ZOMBIE >> processo figlio che termina prima che il padre abbia richiamato wait(..) → *restituisce tutte le risorse al sistema operativo ma rimane in vita finché non verrà chiamato wait()*

Lo scambio di dati tra due processi in Rust può essere fatta usando le Pipe, ovvero "canali" di comunicazione (applicati ai canali stdin/ stdout dei processi coinvolti) che permettono di trasmettere messaggi di dimensioni ben stabilite.

nota - sed_child usa il comando "sed" per modificare la stringa ricevuta e correggerla secondo gli argomenti passati

```
let echo_child = Command::new("echo")
    .arg("Oh no, a tpyo!")
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start echo process");
let echo_out = echo_child.stdout.expect("Failed to open echo stdout");

let mut sed_child = Command::new("sed")
    .arg("s/tpyo/typo/")
    .stdin(Stdio::from(echo_out))
    .stdout(Stdio::piped())
    .spawn()
    .expect("Failed to start sed process");

let output = sed_child.wait_with_output().expect("Failed to wait on sed");
assert_eq!(b"Oh no, a typo!\n", output.stdout.as_slice());
```

create a process that takes as input "Oh no, a tpyo" and sends it as output

take the output from the just created process

create a new process that takes as input the previous output, does something with it, and sends it out as output

La comunicazione tra processi si basa quindi sulla gestione dei loro flussi in entrata / uscita e mandando i relativi messaggi su questi canali tramite delle Pipe().

PROGRAMMAZIONE ASINCRONA

La programmazione in parallelo permette di elaborare più istruzioni allo stesso tempo su diversi thread.

La programmazione asincrona è necessaria quando dobbiamo attendere dei dati invece che sono elaborati da altri thread.

Come implementarla su Rust (e in altri linguaggi) ?

- *Callback*
- *Async / await*

1) Callback

```
read_async(f1, vec![], |buffer: &[u8]|{
   // process buffer from file1...
});
```

- Buffer è il dato in cui sarà contenuta la risposta
- Ho quindi una funzione che, una volta finito di ricevere i dati in input, esegue la funzione passata come callback.
- **Questo metodo risulta confusionario in un contesto di molte chiamate asincrone.**
    - **And_then** + **map_error** > *simile al .then() e .catch() in JS – prendono come input (|result / error | {….})*
        - Nel caso di and_then(…) dovrò sempre controllare che il dato ricevuto non sia un errore con is_ok()

2) Async / await

Async/await permettono di definire una funzione che conterrà delle chiamate asincrone – queste risultano bloccanti nell'esecuzione in sé: *il programma non continuerà finché la chiamata asincrona non sarà completata.*

```
async fn copy(h1: FileHandle, h2: FileHandle) -> std::io::Result<()> {
    let mut buffer = vec![];
    h1.read_async(&mut buffer).await?;
    h2.write_async(&buffer).await
}
```

Async-await implementano per il tipo di dato che restituiscono il tratto *Future* – che rappresenta lo stato del dato rappresentato al suo interno tramite una enum **Poll**

- **Ready(T)**
- **Pending**

Unico metodo *poll() ->* permette in base allo stato dell'enum Poll di possibile capire se il risultato della chiamata asincrona sia pronto o meno.

Tokyo : gestione chiamate asincrone + continuare chiamata asincrona in un thread diverso da quello chiamate (implementare tratto Send)

```
#[tokio::main]
async fn main() {
    let task = tokio::spawn(async { println!("Hello, Tokio!"); });
    task.await.unwrap()
}
```

esegue la funzione non appena è disponibile il risultato

-**join!(** f1: impl Future, f2: impl Future…) > *permette di aspettare l'esecuzione di una serie di chiamate asincrone – restituisce una tupla con i risultati.*

-**select!(…) >** accetta una serie di chiamate asincrone, termina quando la prima finisce restituendo risultato

-**time::sleep(** *duration…***).await >** *sospende la chiamata per la durata attesa, e poi riprende*

-**task::spawn_blocking >** crea un thread esterno per effettuare una chiamata asincrona costosa, risulta bloccante per il thread principale (devo comunque aspettare che finisca per procedere come tutte le altre chiamate async)

Posso anche condividere lo stato di alcuni dati (usando quindi Mutex, Condvar, RwLock) **e comunicare messaggi (**canali**)**

Come?

Creando un task da eseguire come se fosse un thread, quindi con

Tokio::spawn(async move || {…}

Usando gli stessi concetti sviluppati per i thread.

Perché tokio e non std::thread?

La differenza tra thread e l'utilizzo di chiamate asincrone è che i thread sono meno efficienti rispetto all'utilizzo della libreria tokio.