Esercitazione 7

28/30 Maggio 2025

Esercizio 1: simulazione esame

Questo esercizio è da risolvere con l'istanza di Visual Studio Code che si può far partire dal proprio profilo di Moodle.

- Visualizzare la domanda con il testo del problema e premere su "Apri o accedi a CrownLabs"
- VSCode si aprirà in un nuovo tab del browser con un progetto Rust vuoto, in cui svolgere l'esercizio
- Una volta aperta la finestra di VSCode nel browser si potrà sospendere, chiudere e riaprire.
- Finito l'esercizio premere su "termina il tentativo" e poi su "Invia e termina"

Per comodità riportiamo il testo anche qui di seguito.

Un countdown latch è un oggetto che permette di sincronizzare thread con le seguenti caratteristiche:

- l'oggetto contiene un contatore che viene inizializzato con valore N
- il latch può essere condiviso su più thread
- quando un thread chiama il metodo wait_zero() del latch si blocca finché il contatore non va a zero
- quando viene chiamato count_down() il contatore viene decrementato di 1
- quando il contatore va a zero tutti i thread bloccati su wait_zero() vengono sbloccati

Implementare la struct CountDownLatch con la seguente interfaccia:

```
impl CountDownLatch {
	pub fn new(n: usize) -> Self {}
	// wait zero aspetta al massimo timeout ms
	// se esce per timeout ritorna Err altrimenti Ok
	pub fn wait_zero(&self, timeout: Option<std::time::Duration>) ->
Result<(),()>
	pub fn count_down(&self) {}
}
```

Implementato il CountDownLatch, utilizzarlo per completare l'esempio seguente, in cui dei thread hanno bisogno di un driver per eseguire del lavoro; il driver viene preparato e rilasciato dal thread principale:

- i thread non possono eseguire (2) finché il driver non è pronto
- il driver è pronto dopo (1)
- il driver deve essere rilasciato appena i thread non ne hanno più bisogno, quindi (4) deve essere chiamato il prima possibile dopo (2) senza rallentare (3) e senza aspettare la fine dei thread

- doSomeWork() è una funzione che simula lavoro con una sleep()

```
pub fn demo_latch() {
    let mut handles = vec![];
    for _ in 0..10 {
        let h = thread::spawn(||{
            doSomeWork("(2) lavoro che necessita driver");
            doSomeWork("(3) altro lavoro che non necessita driver");
        });
        handles.push(h);
    }
    doSomeWork("(1) prepapara il driver");
    doSomeWork("(4) rilascia il driver");
    for h in handles {
        let _ = h.join();
    }
}
```

Esercizio 2: Implementare una barriera ciclica

Una barriera è un pattern di sincronizzazione simile al latch che permette a *n* thread di attendere che tutti arrivino a un punto comune prima di andare avanti.

La barriera è inizialmente chiusa, quindi ogni thread si blocca in attesa degli altri; quando tutti i thread sono arrivati viene aperta e quando esce l'ultimo viene nuovamente chiusa per bloccare i thread al prossimo loop.

La barriera viene inizializzata con il numero di thread attesi (n) e i thread chiamano `barrier.wait()` quando arrivano al punto di sincronizzazione; ogni thread si ferma, finché l'ultimo non chiama `barrier.wait()`, momento in cui tutti vengono rilasciati.

Si dice ***ciclica*** una barriera che può essere riusata ciclicamente: quando l'ultimo thread chiama wait(), tutti vengono liberati e, al primo wait() successivo, i thread si bloccano nuovamente in attesa dell'arrivo di tutti.

Implementare una struct **CyclicBarrier**, con le caratteristiche descritte:
- deve essere inizializzata con il numero di thread da sincronizzare
- deve avere un metodo wait che permette di attende

Un esempio di uso sono dei thread costituiti da un loop e che devono procedere in parallelo alla stessa velocità; pertanto ad ogni step del loop chiamano `barrier.wait()` per aspettare che tutti abbiano finito il passo:

```
fn main() {
    let abarrier = Arc::new(CyclicBarrier::new(3));

    let mut vt = Vec::new();
```

```
        for i in 0..3 {
            let cbarrier = abarrrier.clone();

            vt.push(std::thread::spawn(move || {
                for j in 0..10 {
                    cbarrier.wait();
                    println!("after barrier {} {}", i, j);
                }
            }));
        }

        for t in vt {
            t.join().unwrap();
        }
    }
```

In questo esempio i thread avanzano con i rispettivi indici “j” sincronizzati, nessun thread avanza più velocemente degli altri. Provate a vedere la differenza commentando wait().

Attenzione!!! Il riuso implica che occorre gestire il caso di eventuali thread troppo veloci, che, una volta usciti, arrivano di nuovo all’ingresso della barriera e chiamano wait() mentre altri devono ancora uscire. Se trovassero la barriera aperta potrebbero eseguire lo step j+1 prima che il più lenti abbiano fatto lo step j.

(questa è una situazione difficilmente debuggabile, ma va tenuta in conto durante la progettazione della barriera; per provarlo si possono provare introdurre ritardi casuali all’uscita dalla wait)

Suggerimenti per l’implementazione
Per evitare la corsa critica in cui un thread richiama subito la wait() e trova la barriera aperta, si può pensare alla porta doppia dell’ingresso di un banca:

- All’inizio la porta esterna è aperta, mentre l’interna è chiusa.
- Quando i thread chiamano wait(), trovano la porta esterna aperta, procedono nella zona di attesa tra le due porte e attendono.
- Quando ci sono n thread fermi in attesa, si chiude la porta esterna e si apre l’interna facendo passare i thread.
- Quando esce l’ultimo thread, si chiude la porta interna e riapre l’esterna
- In questo modo se dei thread sono stati molti veloci e si sono ripresentati all’ingresso, sono stati bloccati in attesa alla porta esterna finché il thread più lento non ha lasciato la barriera e riaperto l’ingresso.

Questo implica che non basta un semplice contatore di quanti thread sono in attesa, ma occorre anche salvarsi uno stato che indica se si può entrare o meno e segnalare agli altri thread quando le porte della barriera cambiano stato.

Esercizio 3: barriera ciclica con canali

Implementare la barriera utilizzano dei canali per la sincronizzazione fra i thread, senza uno stato condiviso in un mutex e condition variable.
L'idea per sfruttare i canali è: se ci sono n thread creiamo n canali, ogni thread possiederà il lato receiver di un canale e gli altri thread n-1 cloni del sender.
Quando il thread arriva al punto di sincronizzazione:

- invia un messaggio su ciascuno degli n-1 canali verso gli altri thread
- si mette in ricezione sul proprio canale di n-1 messaggi
- quando ha ricevuto n-1 messaggi può procedere

La struttura CyclicBarrier da implementare può agevolare la creazione e la condivisione dei canali creando un oggetto Waiter che li incapsula e permette di eseguire tutto lo scambio mediante una singola chiamata a waiter.wait().

Occorre proteggere in qualche modo la struct CyclicBarrier? Perché?

Codice di esempio di invocazione:

```
fn main() {
    let cbarrrier = 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 4: canali e priorità

Realizzare un sistema di monitoraggio di sensori, costituito da un thread che riceve messaggi da varie sorgenti su due differenti canali.
I due canali sono:

- SensorData: che riceve tuple del tipo (String, i32), che contengono l'id del sensore che ha generato la misura e la misura stessa
- Commands: con messaggi di tipo String, che rappresentano un comando inviato da una console utente

Ogni sorgente (un sensore o la console), invia messaggi con una cadenza casuale, distanziati di alcuni secondi uno dall'altro (potete simularli con n thread che fanno delle sleep).

Quando arriva una misura di sensore il sistema deve memorizzare in un vettore la misura. Quando si riceve un comando il thread di monitoraggio deve *immediatamente* eseguire il comando (ad esempio “print sensor_id” stampa gli ultimi n valori ricevuti dal sensore indicato) . Questo vuol dire che il thread non può rimanere in attesa solo su uno dei canali, perché mentre è bloccato su uno, non può leggere messaggi urgenti su un altro.

Per risolvere il problema utilizzare la libreria **crossbeam** e la macro **select!**, che permette di rimanere in attesa su n canali e sbloccarsi non appena uno ha dei dati disponibili da leggere.

Curiosità: questo pattern in cui ci sono n sorgenti che inviano degli eventi ad un unico thread che li processa sequenzialmente si chiama “*event loop*”. E’ molto utilizzato nelle interfacce utente, dove ogni controllo, ogni finestra, ogni dispositivo di input può generare eventi (es: click, key down, resize, mouse move ecc). La struttura di una UI quindi può essere facilmente realizzata così:

1. ogni sorgente emette eventi in modo completamente asincrono e vengono messi in una coda di messaggi (o più code se hanno differente priorità)
2. un unico thread, l’event loop, li raccoglie e li processa sequenzialmente
3. le modifiche all’interfaccia e le azioni conseguenti vengono eseguite all'interno dell’event loop (direttamente o mediante callback registrate per gestire l’evento)
4. la nuova versione aggiornata della interfaccia viene visualizzata sul monitor (l’event loop può decidere se aggiornare ad ogni evento o ogni tot ms accorpando più modifiche)