Esercitazione 4

24 / 26 maggio

Esercizio 1: Barriera ciclica

Una barriera è un pattern di sincronizzazione che permette a *n* thread di attendere che tutti arrivino a un punto comune prima di andare avanti.
La barriera è “chiusa” fino a quando tutti i thread non arrivano ad un certo punto (esempio un risultato pronto), quando sono tutti all’ingresso viene “aperta” e i thread vanno avanti.

Per funzionare la barriera viene inizializzata con il numero di thread attesi (n) e, quando un thread chiama barrier.wait(), si blocca finché tutti i thread non hanno chiamato wait().

Si dice *ciclica* una barriera che può essere riusata; questo implica che occorre gestire eventuali thread troppo veloci, che chiamano subito una seconda wait() appena usciti, e quindi trovino ancora la barriera aperta, senza fermarsi.

Un esempio di come può essere usata:

```
fn main() {
  let abarrrier = Arc::new(cb::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! La barriera ha due stati di funzionamento:

- attesa: si aspetta che giungano tutti i thread
- svuotamento: lascia andare avanti i thread

Quando è aperta in uscita occorre evitare che i thread chiamino di nuovo wait rientrino. Idealmente si può pensare come le porte doppie delle banche: finché non è aperta quella verso il fuori, non viene aperta quella verso il dentro e viceversa.
Questo implica che non basta un semplice contatore di quanti thread sono in attesa, ma occorre anche salvarsi uno stato.

In questo esercizio realizzare la barriera in tre modi differenti:

- "classico", usando mutex e condition variable per gestire lo stato condiviso
- senza stato condiviso e mutex/condition variable, ma usando dei canali. L'idea da cui partire è pensare che ogni thread abbia un certo numero di "token", pari al numero di thread, all'ingresso manda un token a tutti gli altri thread e attende finché non riceve tutti i token degli altri altri (attenzione: se è utile, è possibile modificare l'interfaccia della wait() in wait(i) identificando il thread)
- con un stato gestito da un thread addizionale che agisce da coordinatore e che decide quali thread possano andare avanti; anche in questo caso utilizzare i canali per comunicare fra thread

Provare inoltre a passare un valore *T generico* alla wait e condividere fra tutti i thread il valore passato.

Esercizio 2

Risolvere il problema di exercism chiamato "React"

https://exercism.org/tracks/rust/exercises/react

L'obiettivo generale è realizzare una versione semplificata di foglio elettronico, dove alcune celle contengono valori di ingresso e altre celle sono calcolate dinamicamente da una funzione, sulla base di valori presenti in altre celle.
Un sistema che si comporta in questo modo si definisce "reattivo", vale a dire che come conseguenza del modificarsi di un qualsiasi "input", lo stato del sistema e tutti gli output collegati vengono immediatamente aggiornati.

Il codice di partenza presente in exercism mostra l'interfaccia richiesta.
In più l'esercizio richiede di poter impostare delle funzioni callback che dovranno essere invocate ogni volta che un valore derivatos cambia.

Per questo tipo di problema sono possibili due approcci:

1. "*lazy*", ovvero una volta impostati i valori di ingresso, non viene calcolato il valore di nessuna cella derivata, se non qualora si provi a leggere il contenuto di tali celle
2. "*sincrono*": quando si imposta un valore in ingresso, tutti i valori derivati da questo input vengono immediatamente aggiornati.

Entrambi i metodi hanno pro e contro che dipendono dal contesto di utilizzo (ad esempio in un foglio elettronico vorreste vedere tutte le celle derivate aggiornate all'istante, non solo quando ci cliccate). Dai test e da come vengono definiti in exercism determinare se occorre fare un aggiornamemto sincrono, lazy o è indifferente

Suggerimenti (**NB: leggerli dopo aver visto la base lib.rs di exercism o avranno poco senso**; le classi che suggerite non sono vincolanti e neppure complete).

Nelle celle derivate vi serve memorizzare un valore “calcolato” per confrontare se è cambiato, ma all’inizio può non essere settato: si può usare Option
Per memorizzare la funzione di calcolo dovete usare uno smart pointer in quanto il compilatore vi dice che la dimensione dell’oggetto con il trait Fn non può essere determinato a compile time.
Esempio

```
struct ComputeCell<T> {
    val: Option<T>,
    deps: Vec<CellId>,
    fun: Box<dyn Fn(&[T]) -> T>,
    callbacks: HashMap<CallbackId, Box<dyn FnMut(T)>>
}
```

Per evitare di ciclare su tutte le celle ad ogni aggiornamento per vedere da quali dipende ogni cella, potete memorizzare in una collezione separata (inv_deps) e tutte le dipendenze tra celle; idem per quanto riguarda le callback

```
pub struct Reactor<T> {
    inputs: Vec<T>,
    cells: Vec<ComputeCell<T>>,
    inv_deps: HashMap<CellId, HashSet<ComputeCellId>>,
                                // cellid -> dep to compute only dirty cells
    cbcells: HashMap<CallbackId, ComputeCellId>,
    cb_ids: usize
}
```