POLITECNICO DI TORINO

Programmazione di sistema

Esame 04 Settembre 2023 - Programming

Iniziato	lunedì, 4 settembre 2023, 13:31
Terminato	lunedì, 4 settembre 2023, 15:01
Tempo impiegato	1 ora 30 min.
Valutazione	**12,00** su un massimo di 15,00 (**80**%)

Domanda 1

Completo

Punteggio ottenuto 3,00 su 3,00

Attraverso un esempio pratico si illustri l'utilizzo dei Mutex nel linguaggio Rust. Qual è il meccanismo che consente il loro utilizzo in un contesto thread-safe?

Esempio:

```
fn main() {
  let counter = Arc::new(Mutex::new(0));

  let threads = vec![];
  let N = 10;
  for t in 0..N {
    let thread_counter = counter.clone();
    threads.push(
      thread::spawn(
        move | | {
          // il mut è necessario per aggiornare il counter
           let mut counter_guard = thread_counter.lock().unwrap();
           *counter_guard += 1;
        }  // guard droppato, lock rilasciato
      )
  }

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

Il meccanismo che permette un utilizzo sicuro dei Mutex è composto da più parti.

In primis, i mutex sono condivisi dai thread tramite Arc (Atomic Reference Counted),
che sono degli smart pointer con le seguenti proprietà:
- tengono traccia dei riferimenti (strong e weak) al dato a cui si riferiscono
- sono thread safe: gli aggiornamenti dei contatori sono effettuati con operazioni atomiche su tipi atomici
- implementano il paradigma RAII: quanto l'ultimo riferimento è droppato, la memoria viene rilasciata

Quindi, incapsulando un mutex in un arc, possiamo condividere in modo sicuro un mutex tra più thread.

Commento:

Domanda 2

Completo

Punteggio ottenuto 3,00 su 3,00

Si descriva il concetto di "ownership" in Rust e come contribuisca a prevenire errori come le race condition e i dangling pointer rispetto ad altri linguaggi come C++.

Per imporre dei meccanismi di programmazione sicura, Rust applica delle regole severe sulle operazioni eseguibili sui dati.

Il concetto alla base di queste regole è quello di ownership:
il compilatore rust assegna ad ogni valore un solo possessore (variabile),
e tale di tale valore possono esistere, in modo esclusivo, o più riferimenti non mutabili,
o un sono riferimento mutabile.

Alla base dell'implemetazione di questo meccanismo c'è la semantica di spostamento.
In rust i dati vengono copiati solamente se implementano il tratto Copy,
altrimenti ogni assegnazione è uno spostamento.

La semantica di spostamento permettte di associare ad ogni dato un solo possessore.
Il possessore è anche incaricato di liberare la memoria quando non sarà più richiesta.

I dandling pointer vengono evitati perché il borrow checker verifica che il dato puntato esista ancora
quando viene utilizzato:
se il pointer viene utilizzato fuori dal lifetime del dato puntato, il programma non verrà compilato.

Il un contesto concorrente, le race condition possono essere evitate in quanto eventuali primitive di sincronizzazione tra thread potranno essere possedute da un thread alla volta.

Commento:

Domanda 3

Completo

Punteggio ottenuto 3,00 su 3,00

Si dimostri come sia possibile implementare il polimorfismo attraverso i tratti. Si fornisca anche un esempio concreto che faccia riferimento ad almeno due strutture diverse.

I tratti in Rust sono utilizzati per indicare che un tipo supporta un certo tipo di metodi definiti dal tratto.

Es: Il tratto Iterator richiede il metodo next(&self) -> Option<Self::ItemType>
Se un tipo scritto dall'utente volesse essere marcato come Iterator dovrebbe implementare questo metodo.

Per poter implemetare il polimorfismo vero e proprio in Rust,
si dovrebbe ricorrere alla keywork dyn.
dyn NomeTrattto può essere utilizzato come placeholder al posto del tipo vero e proprio,
e sta ad indicare che l'argomento di quella funzione dovrà essere un tipo che implementi quel tratto.
I dyn sono composti sia da un puntatore al data che alla sua vtable
(puntatori alle implemetantazioni delle funzioni del tratto per quel dato)

Un esempio pratico potrebbe essere il seguente:

```
trait Printer {
  fn print(&Self, s: String)
}

struct EpsilonPrinter()

impl Print for EpsilonPrinter {
  fn print(&self, s: String) {
    println!("{} by Epsilon", s);
  }
}

struct CanonPrinter()

impl Print for CanonPrinter {
  fn print(&self, s: String) {
     println!("{} by Canon", s);
  }
}


// usage

fn get_printer<P>() -> P where P: Printer {
  // ritorna dinamicamente o un CanonPrinter o EpsilonPrinter
  // il compilatore verificherà che ci sia un'implementazione del tratto
  // per il tipo ritornato
  CanonPrinter()
  // or
  EpsilonPrinter()
```

```
}

fn print_hello_word<P>(p: P) where P: Printer {
  p.print(String::new("Hello world"));
}

fn main() {
  let p = get_printer();
  print_hello_world(p);
}
```

Commento:

Domanda 4

Completo

Punteggio ottenuto 3,00 su 6,00

Una cache è una struttura dati, generica, thread safe che consente di memorizzare coppie chiave/valore per un periodo non superiore ad una durata stabilita per ciascuna coppia. Nell'intervallo di validità associato alla coppia, richeste di lettura basate sulla chiave restituiscono il valore corrispondente, se presente. Trascorso tale periodo, eventuali richieste relative alla stessa chiave non restituiscono più il valore.

Poiché il numero di richieste in lettura può essere molto maggiore delle richieste in scrittura, è necessario fare in modo che le prime possano sovrapporsi temporalmente tra loro, mentre le seconde dovranno necessriamente essere con accesso esclusivo. Per evitare la saturazione della struttura, quando si eseguno operazioni di scrittura, si provveda ad eseguire un ciclo di pulizia, eliminando le eventuali coppie scadute.

Si implementi in Rust la struct **Cache<K: Eq+Hash, V>** dotata dei seguenti metodi:

- pub fn new() -> Self // Crea una nuova istanza
- pub fn size(&self) -> usize // Restituisce il numero di coppie presenti nella mappa
- pub fn put(&self, k: K, v: V, d: Duration) -> () // Inserisce la coppia k/v con durata pari a d
- pub fn renew(&self, k: &K, d: Duration) -> Bool // Rinnova la durata dell'elemento rappresentato dalla chiave k; restituisce true se la chiave esiste e non è scaduta, altrimenti restituisce false
- pub fn get(&self, k: &K) -> Option<Arc<V>> // Restituisce None se la chiave k è scaduta o non è presente nella cache; altrimenti restituisce Some(a), dove a è di tipo Arc<V>

Si ricordi che Duration è una struttura contenuta in std::time, che rappresenta una durata non negativa. Può essere sommato ad un valore di tipo std::time::Instant (che rappresenta un momento specifico nel tempo) per dare origine ad un nuovo Instant, collocato più avanti nel

tempo.

```
pub struct Cache<K: Eq+Hash, V> {
  // siccome i metodo sono implementati per reference non mutabili
  // l'hashmap viene incapsulata in una Cell, che permette interior mutability
  // per migliorare l'efficienza della concorrenza usiamo un RwLock
  cache: RwLock<Cell<HashMap< K,(Arc<V>, Duration, Instant) >>>
}

impl <K: Eq+Cash, V> Cache<K, V> {
  // Crea una nuova istanza
  pub fn new() -> Self {
    Self {
      cache: RwLock::new(Cell::new(HashMap::new()))
    }
  }

  pub fn size(&self) -> usize {
    let read_guard = cache.read().unwrap();
    // ipotizzando che la cell.get() ritorni un riferimento
    // non mutabile alla hashmap che contiene
    (*read_guard).get().size();

  }

  pub fn put(&self, k: K, v: V, d: Duration) -> () {
     let mut write_guard = cache.write().unwrap();
     let mut hmap = (*write_guard).get_mut();
     match hmap.get(&k) {
       Some(previously_put_value) => panic!("comportamento non specificato"),
       None => continue,
     }
     // memorizzazione anche l'instante in cui viene inserita
     hmap.entry(k).or_insert((Arc::new(v), d, Instant::now())); // hmap[k] = (Arc::new(v), d,
Instant::now()) ??
  }

  pub fn renew(&self, k: &K, d: Duration) -> bool {
     let mut write_guard = self.cache.write().unwrap();
     // ipotizzo che get_mut(&mut cell) ritorni un riferimento mutabile
     // al dato contenuto nella cella
     let mut hmap = (*write_guard).get_mut();
     let mut entry = hmap.entry(k);
     if entry.is_none() {false}
     else {
      let mut elapsed = false;
```

```
      entry.unwrap().and_modify(move |(v, d_old, i)| {
       if d_old + i > Instant::now() {elapsed= true} (v, d, Istant::now()};
      elapsed
     }
  }

  pub fn get(&self, k: &K) -> Option<Arc<V>> {
    let read_guard = self.cache.read().unwrap();
    let hmap = (*read_guard).get();
    match hmap.get(k) {
      None => None,
      Some(v, d, i) => {
        let now = Instant::now();
        if i + now > d {
           None
        } else {
        Some(v.clone())
        }
    }
  }

}
```

Commento:
la panic non serve a niente... è una cache, se cambia il dato nella put, cambia il dato...

nella renew manca la rimozione se l'entry è scaduta... ed in generale non ha previsto una sola rimozione di una coppia (come invece richiesto...)