POLITECNICO DI TORINO

Programmazione di sistema

Esame 07 Luglio 2023 - Programming

	ATTILIO
Iniziato	venerdì, 7 luglio 2023, 17:00
Terminato	venerdì, 7 luglio 2023, 18:30
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

Si spieghi il concetto di possesso in relazione agli smart pointers. Come viene gestito il ciclo delle risorse quando si utilizzano smart pointers?
Infine, si espliciti il ciclo di vita delle risorse nel seguente esempio:

```
{
    let mut i = 10;
    let bi1 = Box::new(i);
    let mut bi2 = Box::new(*bi1);
    *bi2 = 20;
    i = *bi2;
    println!("{} {:?} {:?}", i, bi1, bi2);
}
```

Gli smart pointer sono dei costrutti che aggiungono funzionalità ad un semplice puntatore. La funzionalità intrinseca degli smart pointer è di assicurare che un puntatore venga distrutto quando non è più utilizzato (usando il paradigma RAII). Quindi i smart pointer quando sono distrutti, distruggono i propri dati contenuti (incluso il puntatore). Ciò varia leggermente per quei smart pointer che supportano più owner (Rc, Arc), dove invece la struttura è deallocata quando tutti i possessori smettono di impiegarlo.

Nell'esempio usiamo lo smart pointer Box, che alloca sull'heap un dato in ingresso e mantiene il puntatore associato come descritto precedentemente.
Questo tipo di puntatore prevede un solo owner, quindi appena Box viene droppato, sarà fatto lo stesso con il puntatore e il relativo dato.
La variabile bi1 quindi punta a questa area di memoria heap con un intero 10 in tutto questo blocco di codice.
La variabile bi2 prende il valore deferenziato del Box bi1 per copia dato che i è un tipo primitivo, ma lo posiziona in una nuova area di memoria heap dove bi2 è l'owner.
Dopodichè il valore puntato da bi2 viene modificato in modo scorrelato a bi1, dato che usa una diversa sezione di memoria. Infine ad i è assegnato il valore puntato da bi2 per copia, che sarebbe 20. Lo stato finale sarà tale da far scrivere in output che i=20, bi2=Box che punta a 20, bi1 Box che punta a 10.

Commento:

Domanda 2

Completo

Punteggio ottenuto 2,00 su 3,00

Si descriva la gestione della memoria in Rust e si spieghi come vengono evitati i problemi di sicurezza comuni come le violazioni di accesso o la presenza di puntatori nulli.

Rust si fonda sul paradigma RAII, ovvero ogni struttura che agisce su delle risorse è responsabile di liberarle quando questa viene distrutta.
Si è esteso questo concetto definendo il concetto di ownership, dove una risorsa "possiede" un altra e una volta che l'owner si distrugge così verrà fatto con quello posseduto ed inoltre nessun altro può accedere o manipolare questa risorsa se non l'owner. A questo concetto si affianca quello del borrowing, dove il permesso di accedere a un dato non posseduto viene reso possibile se si rispettano delle regole controllate a tempo di compilazione:
- una risorsa può essere prestata in lettura se non ci sono altre che la possano modificare
- l'owner non può modificare la risorsa mentre quando ha un prestito attivo
- solo uno alla volta può chiedere un prestito in scrittura

Queste regole servono a garantire che ci sia sempre al massimo uno scrittore al fronte di più lettori che non danno problemi e avere questi forti controlli in fase di compilazione non rende possibile una classe di errori comuni.
Ad esempio non è possibile avere un dangling pointer (puntatore mantenuto verso una area che è stata già deallocata) o una double release (liberare più volte la stessa area di memoria, quando dalla precedenza deallocazione non dovremmo più averne "possesso")

Commento:
il borrow checker non risolve tutto... ad esempio le violazioni di accesso passano dal boundary checker? così come esistono solo i riferimenti: come impattano sui puntatori nulli?

Domanda 3

Completo

Punteggio ottenuto 3,00 su 3,00

Si illustri come sia possibile gestire correttamente le situazioni di errore in Rust, distinguendo tra Option e Result.

Rust gestisce gli errori in maniera differenza rispetto a linguaggi come il C++. In quest'ultimo si usa un approccio dove ogni funzione alloca una sezione dello stack che contiene il contesto delle eccezioni e si controlla il risultato di una funzione invocata ispezionando questo contesto mantenuto al ritorno della funzione.
In Rust non si ha questa struttura extra nello stack, perchè il ritorno stesso della funzione trasmette potenzialmente l'errore.
Ogni funzione che può ritornare un errore utilizza l'enumeratore Result o talvolta Option.
Result<R,E> è definito con due tag Err(E) e Ok(R), ognuno con una variante che contiene l'effettivo valore in caso di successo o insuccesso(Err).
Option<R> ha due tag None e Some(R), dove se la funzione ritorna un risultato valido lo segnala con Some e inserendogli l'effettivo risultato. Questo tipo è più comune per casi in cui la operazione non fallisce, ma è previsto un risultato vuoto (il cosiddetto valore null, che in Rust non è supportato dato che ogni valore è non-nullable). Questo approccio è più sicuro comunque perchè si esplicita quando un valore può essere nullo e lo si deve gestire appositamente.
Il risultato di una funzione che usa questi due tipi, possono essere valutati (e volendo propagati) dal chiamante per decidere cosa fare, subito dopo aver chiamato la funzione, costringendo alla gestione degli errori caso per caso in modo puntuale per funzione.

Commento:

Domanda 4

Completo

Punteggio ottenuto 4,00 su 6,00

Una **DelayedQueue<T:Send>** è un particolare tipo di coda non limitata che offre tre metodi principali, oltre alla funzione costruttrice:

1. **offer(&self, t:T, i: Instant)**: Inserisce un elemento che non potrà essere estratto prima dell'istante di scadenza i.
2. **take(&self) -> Option<T>**: Cerca l'elemento t con scadenza più ravvicinata: se tale scadenza è già stata oltrepassata, restituisce Some(t); se la scadenza non è ancora stata superata, attende senza consumare cicli di CPU, che tale tempo trascorra, per poi restituire Some(t); se non è presente nessun elemento in coda, restituisce None. Se, durante l'attesa, avviene un cambiamento qualsiasi al contenuto della coda, ripete il procedimento suddetto

con il nuovo elemento a scadenza più ravvicinata (ammesso che ci sia ancora).

3. **size(&self) -> usize**: restituisce il numero di elementi in coda indipendentemente dal fatto che siano scaduti o meno.

Si implementi tale struttura dati nel linguaggio Rust, avendo cura di renderne il comportamento thread-safe. Si ricordi che gli oggetti di tipo Condvar offrono un meccanismo di attesa limitata nel tempo, offerto dai metodi wait_timeout(...) e wait_timeout_while(...)).

```
// Ipotesi Instant::now() ritorna l'istante attuale (non sono sicuro al 100% che il nome del metodo sia questo)

struct DelayedQueue<T: Send> {
  mutex: Mutex<Vec<(T,Instant)>>
  cv: Condvar
}

impl<T: Send> DelayQueue<T> {
  fn new() -> Self {
    Self {
      mutex: Mutex::new(vec!()),
      cv: Condvar
    }
  }

  fn offer(&self, t: T, i: Instant) {
    let mut state = self.mutex.lock().unwrap();
    state.push((t,i));
    self.cv.notify_all();  // segnaliamo a tutti che potrebbe esserci un elemento che si potrebbe prendere
  }

  fn take(&self) -> Option<T> {
    let mut state = self.mutex.lock().unwrap();
    state = self.cv.wait_timeout_while(state, |s| {
      s.iter().any(|el| {                    //un qualunque elemento può essere estratto
        el.1 >= Instant::now()               // se l'istante dell'elemento supera quello attuale
        })
        .collect::<Vec<(T, Instant)>>()
        .length() > 0 || s.length() == 0     // l'altro possibile caso è che la coda sia vuota
    }, Duration::from_millis(500)) //decidiamo di fare aspettare 500 ms prima di riprovare

    //ok possiamo valutare cosa ritornare
    if state.length() == 0 {
     return None;
    }

    for index in 0..state.length() {
```

```
      if *state[index].1 >= Instant::now() {
        let taken_value = state.remove(index);
        self.cv.notify_all();
        return Some(taken_value);
      }
    }


  }

  fn size(&self) -> usize {
    let state = self.mutex.lock().unwrap();
    state.length()
  }

}
```

Commento:
visto che non mantiene il vettore ordinato, il for loop non ritorna il minimo, inoltre il suo wait è nella posizione sbagliata (oltre ad avere una closure con una verifica scorretta visto che la collect le da tutti gli element con Instant > Now che significa che deve bloccare solo quanto sono TUTTI così) perché aspetta una notifica anche quando non servirebbe ovvero se la take avviene dopo diverse offer.