POLITECNICO DI TORINO

Programmazione di sistema

Esame 20 Giugno 2022 - Programming

	RICCARDO
Iniziato	lunedì, 20 giugno 2022, 13:39
Terminato	lunedì, 20 giugno 2022, 15:08
Tempo impiegato	1 ora 29 min.
Valutazione	**14,50** su un massimo di 15,00 (**97**%)

Domanda 1

Completo

Punteggio ottenuto 3,00 su 3,00

Si definisca il concetto di Smart Pointer, quindi si fornisca un esempio (Rust o C++) che ne evidenzi il ciclo di vita.

Lo smart pointer in Rust svolge lo stesso lavoro della reference ma in aggiunta fornisce ulteriori funzioni. Possono ad esempio fornire la possibilità di contare quanti diversi riferimenti vengono fatti ad uno stesso dato (Rc) oppure introddure funzionalità per implementare la concorrenza (es.: Mutex). Un ulteriore esempio è il Box<T> che definisce una struttura nello stack che contiene il puntatore al dato che si trova nell'heap. Nel caso di tipi Sized tale struttura contiene il riferimento al dato.
Nel caso di DTS, che quindi non implementano Sized, viene salvata la reference e un usize per rappresentare la grandezza del dato (es.: slice).
Nel caso di oggetto-tratto la struttura è composta da 2 puntatori: uno punta al tipo stesso, e l'altro all'oggetto.
Questo smart pointer risulta utile quando viene gestita una grande mole di dati per cui lo spostamento, o addirittura la copia del dato stesso, diventano molto onerosi.

```
fn f(flag: bool) -> Box<i32>{
 let mut b: Box<i32>;
 if bool{ b = box::new(5)}
 else {b= Box::new(1)}
 return b
}

fn main(){
 let b_true = f(true);
 let b_false = f(false);
}
```

Quando f viene chiamata la prima volta, f crea un Box con 5. Quindi viene inizialmente salvato il puntatore al valore 5 (allocato nell'heap) nella struttura del Box b. Nel momento del ritorno viene trasferito il puntatore da b a b_true e b viene cancellato dallo stack. Lo stesso accade con b_false.

Commento:

Domanda 2

Completo

Punteggio ottenuto 3,00 su 3,00

Si illustrino le differenze nel linguaggio Rust tra **std::channel()** e **std::sync_channel()**, indicando quali tipi di sincronizzazione i due meccanismi permettono.

Entrambi servono per implementare la comunicazione tra thread attraverso messaggi. Attraverso la clone permettono di creare molteplici sender, ma può esistere un solo receiver (multiple sender single receiver).
alla costruzione di channel non viene richiesta una dimensione del buffer. questo è di tipo asincrono e non bloccante.
sync_channel invece chiede un parametro che indica la dimensione del buffer. Questo parametro può essere >=0.
Proprio a causa della dimensione del buffer, questo oggetto assume comportamenti diversi, infatti la send può diventare bloccante. Questo accade quando si cerca di inviare un messaggio ma il buffer è pieno.
dimensione del buffer 0 ==> la lettura e la scrittura devono avvenire contemporaneamente. quindi qui la send è sempre bloccante.
dimensione del buffer n ==> la send diventa bloccante solo quando il buffer è stato saturato.

Quando il sender cerca di mandare messaggi, ma il receiver ha cessato di esistere, viene restituito un Err al sender. Stesso vale quando il receiver cerca di ricevere in assenza di un sender.

Commento:

Domanda 3

Completo

Punteggio ottenuto 3,00 su 3,00

Dato il seguente frammento di codice Rust (ogni linea è preceduta dal suo indice)

```
1.  struct Point {
2.      x: i16,
3.      y: i16,
4.  }
5.
6.  enum PathCommand {
7.      Move(Point),
8.      Line(Point),
9.      Close,
10. }

11. let mut v = Vec::<PathCommand>::new();
12. v.push(PathCommand::Move(Point{x:1,y:1}));
13. v.push(PathCommand::Line(Point{x:10, y:20}));
14. v.push(PathCommand::Close);
15. let slice = &v[..];
```

Si descriva il contenuto dello stack e dello heap al termine dell'esecuzione della riga 15.

---

stack
Vec (puntatore al primo elemento nell'heap | size=3 | capacity)
slice (puntatore al primo elemento nell'heap | size = 3)

heap: 3 enum con dimensione pari a (1B + 32b) * 3
Move |tag=0|1|1|
Line |tag=1|10|20|
Close |tag=2|x|x|

Commento:

Domanda 4

Completo

Punteggio ottenuto 5,50 su 6,00

Un paradigma frequentemente usato nei sistemi reattivi è costituito dall'astrazione detta **Looper**.

Quando viene creato, un **Looper** crea una coda di oggetti generici di tipo **Message** ed un thread. Il thread attende - senza consumare cicli di CPU - che siano presenti messaggi nella coda, li estrae a uno a uno nell'ordine di arrivo, e li elabora. Il costruttore di **Looper** riceve due parametri, entrambi di tipo (puntatore a) funzione: **process**(…) e **cleanup**(). La prima è una funzione responsabile di elaborare i singoli messaggi ricevuti attraverso la coda; tale funzione accetta un unico parametro in ingresso di tipo **Message** e non ritorna nulla; La seconda è funzione priva di argomenti e valore di ritorno e verrà invocata dal thread incapsulato nel **Looper** quando esso starà per terminare.

Looper offre un unico metodo pubblico, thread safe, oltre a quelli di servizio, necessari per gestirne il ciclo di vita: **send**(msg), che accetta come parametro un oggetto generico di tipo **Message** che verrà inserito nella coda e successivamente estratto dal thread ed inoltrato alla funzione di elaborazione. Quando un oggetto **Looper** viene distrutto, occorre fare in modo che il thread contenuto al suo interno invochi la seconda funzione passata nel costruttore e poi termini.

Si implementi, utilizzando il linguaggio Rust o C++, tale astrazione tenendo conto che i suoi metodi dovranno essere *thread-safe*.

```
struct Looper{
 sender: Sender
}

impl Looper{
 fn new(process:T , cleanup : Q) where T:Fn(msg:Message)->(), Q:Fn()->() -> Self{
   let (sender, receiver) = std::channel<Message>::new();
   thread::spawn(move ||{
    loop{
     match receiver.recv(){
      Ok(msg) => process(msg);
      Err(_) => break;
     }
   }
   });
   Self{ sender }
 }

 pub fn send(&self, msg:Message){
  self.sender.send(msg).unwrap();
 }
}

impl Drop for Looper{
 drop(self.sender);
}
```

Commento:
e la cleanup?