Esame di Programmazione di Sistema

Parte di Programmazione RUST

25 giugno 2024

Cognome	Nome	Matricola

1 Teoria #1 (3 pt)

i consideri il programma seguente che riporta la numerazione delle linee di codice.

```
1 fn main() {
2    let numbers = vec![1,2,3,4,5,8];
3
4    let res = numbers
5       .iter()
6        .filter(|&x| x % 2 == 0)
7        .zip('a'..'z');
8
9    let last = res
10        .clone()
11        .map(|(a,b)|{format!("{b}{a}")})
12       .last();
13
14   println!("last: {:?}", last);
15   println!("res: {:?}", res.count());
16 }
```

Che cosa stampa questo codice?
Che cosa fanno le istruzioni alle righe 5,6,7?
Che cosa capita se si omette la riga 10? Perché?

2 Teoria #2 (3 pt)

Si descriva il comportamento di questo programma.

Se presenta delle problematiche, come può essere modificato?

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;
use std::time::Duration;

fn main() {
    let pair = Arc::new((Mutex::new(false), Condvar::new()));
    let pair2 = Arc::clone(&pair);

    thread::spawn(move || {
        let (lock, cvar) = &*pair2;
        let mut started = lock.lock().unwrap();
        *started = true;
        cvar.notify_one();

    });
    let (lock, cvar) = &*pair;

    println!("Waiting ...");
    thread::sleep(Duration::from_secs(1));

    let mut started = lock.lock().unwrap();
    started = cvar.wait(started).unwrap();

    println!("End!");
}
```

3 Teoria #3 (3 pt)

Il codice seguente genera un errore di compilazione: spiegare perché e indicare come modificare la **struct S** (attraverso l'aggiunta di tratti) per renderlo compilabile ed eseguibile.

```
#[derive(Debug)]
struct S {
    i: i32,
}

impl From<i32> for S { fn from(value: i32) -> Self { S { i: value } } }

fn main() {
    let mut v = Vec::<S>::new();
    let s = 42.into();
    for i in 0..3 {
        v.push(s);
    }
    println!("{:?}",v);
}
```

4 Programmazione (6 pt)

Si realizzi l'implementazione della struttura dati **Exchanger<T: Send>** (e dei metodi e delle funzioni necessarie) utile per realizzare una comunicazione bidirezionale.

Ciascun lato della comunicazione dispone di un'istanza della struttura **Exchanger<T: Send>**.

La comunicazione avviene invocando il metodo

fn exchange(&self, t:T) -> Option<T>

che, una volta invocato, si blocca fino a quando non viene invocato il metodo corrispettivo sulla struttura corrispondente al lato opposto della comunicazione, dopodiché restituisce il valore che è stato passato come argomento al metodo corrispondente al lato opposto (che farà altrettanto), sotto forma di **Some(t)**.

Lo scambio può essere ripetuto un numero arbitrario di volte.

Se una delle due strutture formanti la coppia viene distrutta, un'eventuale chiamata, bloccata sul metodo della struttura restante, terminerà restituendo il valore **None**.

Si implementi tale struttura in linguaggio Rust avendo cura che la sua implementazione sia *thread-safe*.