Corso: Programmazione di sistema
Quiz: API Programming - 13 Gennaio 2025

PORTALE ESAMI

Iniziato	13 gennaio 2025, 13:57
Stato	Completato
Terminato	13 gennaio 2025, 15:26
Tempo impiegato	1 ora 29 min.
Valutazione	**13,0** su un massimo di 15,0 (**87**%)
Riepilogo del tentativo	1 ✓ 2 ✓ 3 ● 4 ●

Domanda 1

Completo

Punteggio ottenuto 3,0 su 3,0

Dato il seguente programma si descriva il comportamento del programma e la mappa della memoria utilizzata dalle variabili

```
fn main() {
  trait Calculate {
    fn add(&self, l: u64, r: u64) -> u64;
    fn mul(&self, l: u64, r: u64) -> u64;
  }
  struct Modulo(u64);

  impl Calculate for Modulo {
    fn add(&self, l: u64, r: u64) -> u64 {
      (l + r) %self.0
    }

    fn mul(&self, l: u64, r: u64) -> u64 {
      (l * r) % self.0
    }
  }

  let mod3 = Modulo(3);

  let tobj: &dyn Calculate = &mod3;
  let result = tobj.add(2, 2);

  println!(" Result = {:?}", result);
}
```

Il seguente programma definisce un'istanza della struct Modulo, che implementa il tratto Calculate.
Dopodichè ne acquisisce un riferimento, gestito tramite oggetto tratto, tramite la variabile tobj.
Questa viene utilizzata per chiamare il metodo add del tratto che la struct implementa ed infine
ne viene stampato il risultato.

In questo caso essendo che la chiamata viene gestita tramite oggetto tratto, implementato come puntatore doppio (puntatore alla struct e alla vtable) ne consegue una penalizzazzione in termini
di spazio e di tempo sia per la vtable che per trovare tramite essa l'indirizzo della funzione da chiamare.

L'occupazione in memoria è:
- 8 byte per mod3 (dimensione della struct Modulo, avente solamente un campo u64).
- 16 byte per la variabile tobj (8 byte puntatore alla struct modulo e 8 byte puntatore alla vtable)
- 8 byte per il risultato della chiamata del metodo add, (variabile result) essendo un u64.

Il tutto risulta essere allocato sullo stack, non essendovi vec o tantomeno smartpointer come Rc che allocano in heap.

Commento:
ok

Domanda 2
Completo
Punteggio ottenuto 3,0 su 3,0

Dato il seguente programma si motivi l'errore di compilazione e si corregga il programma in modo da stampare il valore aggiornato della variabile count dopo ogni incremento.

```
1 fn main() {
2   let mut count = 0;
3
4   let mut increment_n = |n| {
5     count += n;
6   };
7
8   increment_n(10);
9
10  println!("{}", count);
11
12  increment_n(5);
13
14  println!("{}", count);
15 }
```

L'errore di compilazione è dovuto al fatto che la closure acquisisce un riferimento mutabile a count.
Le println invece provano a stamparlo cercando di acquisire un borrow in sola lettura, che và quindi
in contrasto col borrow mutabile della chiusura.

per correggere il programma e stampare count dopo ogni incremento si potrebbe adottare una soluzione
del genere:

```
fn main() {
  let mut count = 0;

  let mut increment_n = |n| {
    count += n;
    println!("{}",count);
  };

  increment_n(10);
  increment_n(5);
}
```

in questo modo viene gestito tutto all'interno della chiusura, ed ogni volta che verrà chiamata
count verrà stampato col valore aggiornato.

Commento:
ok

Domanda 3

Completo

Punteggio ottenuto 2,0 su 3,0

Spiegare il comportamento del seguente programma.

```
1 fn call_x(mut f: impl FnMut(), x:i32 ) {
2   for _ in 0..x {
3     f();
4   }
5 }
6 fn main() {
7   let mut a = 0;
8   let count_a = move || { a += 1; print!("{} ",a); };
9   a=2;
10  call_x(count_a,2);
11  call_x(count_a,2);
12  println!("\t | a = {}" , &a);
13  call_x(count_a,4);
14  println!("\t | a = {}" , &a);
15 }
```

```
Il seguente programma è caratterizzato da una chiusura al cui interno viene mossa
(dunque ne prende possesso) la variabile a, e ad ogni esecuzione della chiusura la
variabile viene
incrementata e poi stampata (riga 7 e 8).

Diversa risulta essere quindi la variabile a che viene poi assegnata a 2 (riga 9) nel
main.
Di conseguenza le due chiamate di call_x, (riga 10 e 11), portano lo stato interno della
chiusura
ad avere valore 4, eseguendo la chiusura 4 volte e stampando 1,2,3 e 4.

Nel main a rimane 2 (sempre riga 9).

La print di riga 12 restituirà 2.
A riga 13 modifichiamo nuovamente lo stato della chiusura, portandolo a 8
(anche qui si stamperà il valore dello stato interno della chiusura incrementato, dunque
5,6,7 e 8).
La print di riga 14 restituirà 2, facendo sempre riferimento non allo stato interno della
chiusura.

Difatti lo stato interno della chiusura è stato mosso precedentemente e quindi viene
gestito
nella chiusura non tramite riferimento
(comportamento di base nel caso in cui non si usi la keywork move).
```

Commento:

I tipi che implementano il tratto Copy (come i32, il tipo della variabile "a") non vengono mossi ma copiati: all'interno della chiusura "count_a" c'è una copia della variabile "a", inizializzata a 0. Tale copia è indipendente dall'originale. Poiché la chiusura contiene solo un campo di tipo i32 che ha il tratto Copy, anche l'intera chiusura gode del tratto Copy. Ogni volta che le funzione call_x(...) viene invocata, riceve una copia della chiusura originale, con il campo "a" inizializzato a 0. La variabile "a" introdotta alla riga 7, acquisisce alla riga 9 il valore 2 e lo mantiene. Di conseguenza il programma stampa quanto segue:

1 2 1 2 | a = 2

1 2 3 4 | a = 2

Domanda 4

Completo

Punteggio ottenuto 5,0 su 6,0

La struct `DelayedExecutor` permette di eseguire funzioni in modo asincrono, dopo un certo intervallo di tempo.

Essa offre tre metodi:

- `new() -> Self` crea un nuovo DelayedExecutor
- `execute<F: FnOnce()+Send+'static>(f:F, delay: Duration) -> bool`
  se il `DelayedExecutor` è aperto, accoda la funzione f che dovrà essere eseguita non prima che sia trascorso un intervallo pari a `delay` e restituisce `true`;
  se invece il `DelayedExecutor` è chiuso, restituisce `false`.
- `close(drop_pending_tasks: bool)` chiude il `DelayedExecutor`;
  se `drop_pending_tasks` è `true`, le funzioni in attesa di essere eseguite vengono eliminate, altrimenti vengono eseguite a tempo debito.

`DelayedExecutor` è thread-safe e può essere utilizzato da più thread contemporaneamente.

I task sottomessi al `DelayedExecutor` devono essere eseguiti in ordine di scadenza.

All'atto della distruzione di un `DelayedExecutor`, tutti i task in attesa sono eliminati, ma se è in corso un'esecuzione questa viene portata a termine evitando di creare corse critiche.

Si implementi questa struct in linguaggio Rust.

```
#[derive(Eq,PartialEq)]
enum ExecutorState {
   Open,Closed
}

struct <F: FnOnce() + Send + 'static> DelayedExecutor<F> {
    tasks: Arc<Mutex<(Vec<(F,Instant)>,ExecutorState)>>,
    cv: Arc<Condvar>,
    // Option so with take method we take ownership and we can then
    // invoke join to wait on the termination
    jh: Option<JoinHandle<()>>
}

impl Drop<F: FnOnce() + Send + 'static> for DelayedExecutor<F> {
     fn drop(&mut self) {
         // when the Executor is destroyed, clear the tasks
         let mut tasks_guard = self.tasks.lock().unwrap();
         (*tasks_guard).0.clear();
         drop(tasks_guard);
         // wait on the joinhandle in case some execution is still in progress in the
wrapped thread
         self.jh.take().join().unwrap();
     }
}

impl <F: FnOnce() + Send + 'static> DelayedExecutor<F> {
    fn new() -> Self {
       // create the shared state
       let tasks = Arc::new(Mutex::new((vec![],ExecutorState::Open)));
       let cv = Arc::new(Condvar::new());
       // spawn the thread to execute the tasks
       let tasks_c = tasks.clone();
       let cv_c = cv.clone();
       let jh = thread::spawn(move || {
           loop {
                // wait until there is some tasks or the channel is closed
                let mut tasks_guard = tasks_c.lock().unwrap();
                tasks_guard = cv_c.wait_while(tasks_guard,
                      |g| {(*g).0.len() == 0 && (*g).1 == ExecutorState::Open}).unwrap();
                if (*tasks_guard).0.len() != 0 {
                    // find the task to execute (the minimum with respect to the Instant)
                    let mut min_instant = (*tasks_guard).0[0].1;
                    let mut min_index = 0;
                    for i in 0..(*tasks_guard).0.len() {
                        let instant = (*tasks_guard).0[i].1;
                        if instant < min_instant {
                           min_instant = instant;
                           min_index = i;
                        }
                    }
                    // got the minimum index, execute the corresponding task and remove
it from vec
                    let (f,inst) = (*tasks_guard).0.remove(min_index);
                    // drop the lock before actual execution, since can take a long time
                    drop(tasks_guard);
                    f();
                } else if (*tasks_guard).1 == ExecutorState::Closed {
                    // channel closed and no more tasks, exit the loop and terminate the
```

```
thread
                    break;
                }
            }
        });
        // return the instance
        Self {
            tasks,
            cv,
            jh: Some(jh)
        }
    }

    fn execute<F: FnOnce() + Send + 'static>(&self, f: F, delay: Duration) -> bool {
        // get the lock
        let mut tasks_guard = self.tasks.lock().unwrap();
        if ((*tasks_guard).1 == ExecutorState::Closed) {
            return false;
        }
        // insert the task to execute and notify the incapsulated thread
        (*tasks_guard).0.push((f,Instant::now() + delay));
        self.cv.notify_one();
        return true;
    }

    fn close(&self, drop_pending_tasks: bool) {
        // get the lock
        let mut tasks_guard = self.tasks.lock().unwrap();
        // close the channel
        (*tasks_guard).1 = ExecutorState::Closed;
        if drop_pending_tasks {
            // clear also the vector, so remove the tasks
            (*tasks_guard).0.clear();
        }
        // notify the incapsulated thread
        self.cv.notify_one();
    }
}
```

Commento:
La struct DelayedExecutor non è generica.

Nell'implementazione del tratto Drop, occorre dapprima chiudere il DelayedExecutor, poi attendere la terminazione del thread.

Piuttosto che fare una ricerca complessa del task più ravvicinato all'interno del thread, conviene - in fase di inserimento - riordinare il vettore così da avere al fondo il task più ravvicinato.

Una soluzione basata sul tuo codice è:

```
mod delayed_executor {
    use std::sync::{Arc, Condvar, Mutex};
    use std::thread;
    use std::thread::JoinHandle;
    use std::time::{Duration, Instant};

    #[derive(Eq,PartialEq)]
    enum ExecutorState {
        Open,Closed
    }



    pub struct  DelayedExecutor {
        tasks: Arc<Mutex<(Vec<(Box<dyn FnOnce()+Send+'static>,Instant)>,ExecutorState)>>,
        cv: Arc<Condvar>,
        // Option so with take method we take ownership and we can then
        // invoke join to wait on the termination
        jh: Option<JoinHandle<()>>
    }

    impl Drop for DelayedExecutor{
        fn drop(&mut self) {
            self.close(true);
            // wait on the joinhandle in case some execution is still in progress in the wrapped thread
            self.jh.take().unwrap().join().unwrap();
        }
    }

    impl  DelayedExecutor {
        pub fn new() -> Self {
            // create the shared state
            let tasks = Arc::new(Mutex::<(Vec<(Box<dyn FnOnce()+Send+'static>, Instant)>,
ExecutorState)>::new((vec![],ExecutorState::Open)));
            let cv = Arc::new(Condvar::new());
            // spawn the thread to execute the tasks
            let tasks_c = tasks.clone();
            let cv_c = cv.clone();
            let jh = thread::spawn(move || {
                loop {
                    // wait until there is some tasks or the channel is closed
                    let mut tasks_guard = tasks_c.lock().unwrap();
                    tasks_guard = cv_c.wait_while(tasks_guard,
                                                  |g| {g.0.is_empty() && g.1 ==
ExecutorState::Open}).unwrap();
                    if !tasks_guard.0.is_empty() {
                        // find the task to execute (the minimum with respect to the Instant)
                        let task = tasks_guard.0.last().unwrap();
                        let now = Instant::now();
                        if task.1<=now {
                            // execute the task
                            let item = (*tasks_guard).0.pop().unwrap();
                            // drop the lock before actual execution, since can take a long time
                            drop(tasks_guard);
                            item.0();
                            tasks_guard = tasks_c.lock().unwrap();
                        } else {
                            let d = task.1.duration_since(now);
                            // wait until the task is ready to be executed
                            tasks_guard = cv_c.wait_timeout(tasks_guard, d).unwrap().0;
                        }
                    } else {
                        // channel closed and no more tasks, exit the loop and terminate the thread
                        break;
                    }
                }
            });
            // return the instance
```

```
            Self {
                tasks,
                cv,
                jh: Some(jh)
            }
        }

        pub fn execute<F: FnOnce() + Send + 'static>(&self, f: F, delay: Duration) -> bool {
            // get the lock
            let mut tasks_guard = self.tasks.lock().unwrap();
            if (*tasks_guard).1 == ExecutorState::Closed {
                return false;
            }
            // insert the task to execute and notify the incapsulated thread
            tasks_guard.0.push((Box::new(f),Instant::now() + delay));
            tasks_guard.0.sort_by(|a,b| b.1.cmp(&a.1));
            self.cv.notify_one();
            return true;
        }

        pub fn close(&self, drop_pending_tasks: bool) {
            // get the lock
            let mut tasks_guard = self.tasks.lock().unwrap();
            // close the channel
            (*tasks_guard).1 = ExecutorState::Closed;
            if drop_pending_tasks {
                // clear also the vector, so remove the tasks
                (*tasks_guard).0.clear();
            }
            // notify the incapsulated thread
            self.cv.notify_one();
        }
    }


}
```