POLITECNICO DI TORINO

Programmazione di sistema

Esame del 22/1/2024 - API Programming

Iniziato	lunedì, 22 gennaio 2024, 14:00
Terminato	lunedì, 22 gennaio 2024, 15:30
Tempo impiegato	1 ora 30 min.
Valutazione	**12,5** su un massimo di 15,0 (**83**%)

Domanda 1

Completo

Punteggio ottenuto 2,0 su 3,0

Sia data la struttura LinkedList<T> definita come:

```
pub struct LinkedList<'a, T> {
    pub val: Option<T>,
    pub next: Option<&'a Box<LinkedList<'a, T>>>,
}
```

Si definisca l'occupazione di memora di un elemento della lista e si indichi come sia possibile definire il fine lista.

L'occupazione di memoria di un elemento di tipo LinkedList è la seguente:

- il campo val è di tipo Option, quindi un enum, che occupa 1 byte di tag, x byte di T (che è un tipo generico, la cui occupazione sarà determinata dal compilatore durante la monomorfizzazione) e i restanti byte di padding;

- il campo next è a sua volta un enum di tipo Option, e la variante Some è un puntatore (Box) ad un dato memorizzato nell'heap (che sarà di tipo LinkedList). Il Box è formato dal puntatore (8 byte) al dato nell'heap e dal campo size (4 byte per rappresentare la grandezza del dato memorizzato nell'heap). Quindi si avrà 1 byte di tag, 8 byte per il puntatore all'heap, 4 byte per il campo size e 3 byte di padding.

Questa struttura è memorizzata nello stack. Nell'heap, come detto, sarà memorizzata la struct puntata dal Box contenuto nella variante Some del campo next.

Per definire il fine lista si può settare a None il campo next della struct LinkedList.

Commento:
- next è un riferimento a Box non Box (solo 8byte)
- in ogni caso la struct è sized, quindi anche Box solo 8 byte

- la seconda Option<T> utilizza la null pointer optimization per cui non occupa più memoria del puntatore

Occupazione es T=i32 e arc 64 bit:

- val 1 byte + size_of::<i32>() 4 = 5

- next: 0 byte + size_of::<&Box<LinkedList>>>() = 8

con allineamento: 16 byte

Domanda 2

Completo

Punteggio ottenuto 2,5 su 3,0

Si definisca un esempio in cui, data la necessità di creare N thread, si possano evitare *race-conditions* nel momento in cui i thread debbano accedere in scrittura alla stessa risorsa. Si distingua il caso in cui tale risorsa sia uno scalare e quella in cui sia una struttura più articolata.

In un programma multi-thread, in assenza di costrutti di sincronizzazione, si possono verificare delle race-conditions quando più thread cercano di accedere ad una stessa risorsa modificandone il valore. Nel caso di due thread, ad esempio, il valore finale della risorsa condivisa può essere quello scritto dal primo thread, dal secondo oppure un valore completamente arbitrario (questo fenomeno è noto come interferenza), e in ogni caso non è possibile prevederne il contenuto, essendo l'esecuzione non deterministica. Per questo motivo occore proteggere le risorse condivise tra più thread con opportuni costrutti di sincronizzazione. Per far ciò si può ricorrere all'uso di Mutex (mutual exclusion lock), che sono delle strutture che "incapsulano" la struttura dati condivisa permettendo l'accesso solo ad un thread alla volta: per accedere ad una risorsa occorre prima aver ottenuto il possesso del suo mutex tramite il metodo lock(). Una volta ottenuto il lock, un altro thread che cerchi di ottenerne il possesso è costretto ad aspettare finché il lock non viene liberato dal possessore attuale: in questo modo vengono evitati una serie di problemi di sincronizzazione tra più thread. E' opportuno che il thread possessore del lock lo rilasci prima della sua terminazione (per evitare situazioni di deadlock), anche in caso di terminazione con errore. Ciò in Rust è derivato dall'adozione del paradigma RAII: quando un thread esce dal suo scope sintattico (o termina con errore), la contrazione dello stack determina il rilascio di tutte le risorse, ed il Mutex (essendo di fatto uno smart pointer composto da un puntatore ad un mutex nativo del sistema operativo, un campo poison che indica se il thread sia terminato correttamente o meno, ed il dato "incapsulato" dal mutex stesso) rilascia automaticamente le risorse acquisite, evitando deadlock.
Come detto, per strutture articolate si ricorre all'utilizzo di mutex. Per risorse più "semplici" (come uno scalare), Rust, oltrei ai già citati Mutex, mette a disposizione dei particolari tipi, chiamati Atomic. Possono essercene diversi per i tipi più semplici (ad esempio AtomicBool, AtomicUsize...), e garantiscono che le operazioni che vengono effettuate su questi tipi siano atomiche (cioè indivisbili), appoggiandosi internamente ad istruzioni di tipo fence o barrier offerte dai processori, garantendo quindi sincronizzazione tra thread diversi.

Commento:
- ok la parte teorica generale, ma manca un esempio concreto in cui è necessario prevenire una race condition

Domanda 3

Completo

Punteggio ottenuto 3,0 su 3,0

Si usi un esempio concreto per dimostrare la capacità di introdurre modularità attraverso i Tratti.

Un tratto in Rust può essere visto come un'interfaccia descritta nel linguaggio Java. Un tratto cioè definisce una serie di metodi che gli oggetti che lo implementano dovranno implementare (possono anche adottare l'implementazione di default a patto che sia definita). A differenza di altri linguaggi, normalmente l'utilizzo di un tratto non comporta costi aggiuntivi: se il compilatore conosce il valore del tipo che implementa il tratto non è necessario il passaggio attraverso la vtable (che contiene tutte le implementazioni dei metodi del tratto), con conseguente costo in termini di memoria e tempo. Se invece si possiede un riferimento ad un valore che implementa un tratto, è necessario ricorrere agli oggetti-tratto, che sono formati dal puntatore al valore del tipo e da un puntatore alla vtable (in questo caso vi è quindi il costo aggiuntivo della vtable).
I tratti di per sè non possono ereditare da altri tratti (in Rust infatti non vi è un supporto specifico all'ereditarietà), ma si possono creare relazioni di "parentela" tra i diversi tratti. Ad esempio, tutti i tipi che implementano il tratto Copy (che vengono definiti quindi copiabili, come i tipi elementari: scalari, booleani...) devono per forza implementare anche il tratto Clone. Inoltre, vi possono essere dei tratti che sono vicendevolmente esclusivi: ad esempio, un tipo che implementa il tratto Copy (che quindi non rappresenta una "risorsa" che richiede ulteriori azioni al momento del rilascio) è mutuamente esclusivo con il tratto Drop, che invece definisce il comportamento di una risorsa quando viene rilasciata.
Vi sono poi dei particolari tratti, chiamati tratti marker, che non contengono metodi, ma descrivono il comportamento assunto da un tipo in particolari situazioni. Sono un esempio i tratti della concorrenza, Send e Sync. Se un tipo che implementa un tratto Send è possibile trasferirlo in maniera sicura da un thread all'altro (cioè garantisce che non siano possibili accessi contemporanei al suo valore), mentre se un tipo implementa un tratto Sync è possibile condividere riferimenti non mutabili tra thread diversi. Ci sono tipi, come Cell e Refcell, che, implementando un pattern di interior mutability, non godono del tratto Sync, pur implementando il tratto Send. I tipo che implementano il tratto Sync sono dunque un sottoinsieme di quelli che implementano il tratto Send.
E' possibile, infine, che un dato tratto "derivi" da un altro tratto (subtrait o supertrait), estendendone il comportamento o ridefinendo l'implementazione di alcuni metodi (diversi tipi, a seconda che implementino il tratto o il sotto-tratto, avranno quindi comportamenti diversi).

Commento:
Ok

Domanda 4

Completo

Punteggio ottenuto 5,0 su 6,0

La struttura MultiChannel implementa il concetto di canale con molti mittenti e molti ricevitori.

I messaggi inviati a questo tipo di canale sono composti da singoli byte che vengono recapitati a tutti i ricevitori attualmente collegati.

Riferimenti a tipi:

```
use std::result::Result;
use std::sync::mpsc::{Receiver, SendError};
```

Metodi:

```
new() -> Self
                // crea un nuovo canale senza alcun ricevitore collegato

subscribe(&self) -> Receiver<u8>
                // collega un nuovo ricevitore al canale: da quando
                // questo metodo viene invocato, gli eventuali byte
                // inviati al canale saranno recapitati al ricevitore.
                // Se il ricevitore viene eliminato, il canale
                // continuerà a funzionare inviando i propri dati
                // ai ricevitori restanti (se presenti), altrimenti
                // ritornerà un errore

send(&self, data: u8) -> Result<(), SendError<u8>>
                // invia a tutti i sottoscrittori un byte
                // se non c'è alcun sottoscrittore, notifica l'errore
                // indicando il byte che non è stato trasmesso
```

```
pub struct MultiChannel {
  senders: Arc<Mutex<Vec<Sender<u8>>>>,
}

impl MultiChannel {
  pub fn new() -> Self {
    senders: Arc::new(Mutex::new(Vec::<Sender<u8>>::new())),
  }
```

```
  pub fn subscribe(&self) -> Receiver<u8> {
    let (tx, rx): (Sender<u8>, Receiver<u8>) = channel();
    let mut guard = self.senders.lock().unwrap();
    //Proteggo il vettore dei trasmettitori con un mutex, perché non voglio che mentre ne aggiungo
uno un
    //altro thread possa operarci
    guard.push(tx);
    rx
  }

  pub fn send(&self, data: u8) -> Result<(), SendError<u8>> {
    let mut guard = self.senders.lock().unwrap();
    let no_transmitted = true;
    for s in guard.iter() {
      match s.send() {
        Ok() => { no_transmitted = false; },
        Err(_) => {},
      }
    }
    //Se no_transmitted==true significa che il dato non è stato trasmesso a nessun ricevitore, cioè
non c'è
    //più alcun sottoscrittore "iscritto" al canale, quindi ritorno un errore di tipo SendError
    if no_transmitted {
      return Err(SendError(data));
    } else {
      return Ok();
    }
  }
}
```

Commento:
Non occorre incapsulare il contenuto del campo senders all'interno di un Arc<T>: questo non ti aiuta a rendere l'oggetto condivisibile tra più possessori e introduce un'inutile accesso indiretto al Mutex.

Nel metodo send(...) non ripulisci il vettore dai canali chiusi: questo comporta un crescente costo in termini di cicli macchina e di memoria via via che canali vengono aggiunti e rimossi.

Una soluzione basata sul tuo codice è:

```
use std::sync::mpsc::{channel, Receiver, Sender, SendError};
use std::sync::Mutex;

pub struct MultiChannel {
    senders: Mutex<Vec<Sender<u8>>>,
}

impl MultiChannel {
    pub fn new() -> Self {
        Self {
            senders: Mutex::new(Vec::<Sender<u8>>::new())
        }
    }

    pub fn subscribe(&self) -> Receiver<u8> {
        let (tx, rx): (Sender<u8>, Receiver<u8>) = channel();
        let mut guard = self.senders.lock().unwrap();
        //Proteggo il vettore dei trasmettitori con un mutex, perché non voglio che mentre ne aggiungo
uno un
        //altro thread possa operarci
        guard.push(tx);
        rx
    }

    pub fn send(&self, data: u8) -> Result<(), SendError<u8>> {
        let mut guard = self.senders.lock().unwrap();
        guard.retain(|s|s.send(data).is_ok());
        //Se no_transmitted==true significa che il dato non è stato trasmesso a nessun ricevitore, cioè
non c'è
        //più alcun sottoscrittore "iscritto" al canale, quindi ritorno un errore di tipo SendError
        if guard.is_empty() {
            return Err(SendError(data));
        } else {
            return Ok(());
        }
    }
}
```