POLITECNICO DI TORINO

Programmazione di sistema

Esame del 20/6/2023 (API Programming)

[redacted]
303434

Iniziato	martedì, 20 giugno 2023, 13:31
Terminato	martedì, 20 giugno 2023, 15:01
Tempo impiegato	1 ora 30 min.
Valutazione	**14,5** su un massimo di 15,0 (**97**%)

Domanda 1

Completo

Punteggio ottenuto 2,5 su 3,0

Si definisca il problema delle referenze cicliche nell'uso degli smart pointers. Si fornisca quindi un esempio in cui tale problema sia presente.

In Rust i riferimenti classici implicano il possesso del valore referenziato, e ne dtermina il ciclo di vita: ciò significa che finché esiste un riferimento ad un valore, questo non viene distrutto. Ciò comporta un problema qualora si desideri creare delle strutture con riferimenti ciclici, ovvero una stuttura dati che è puntata da una seconda struttura di cui la prima possiede un riferimento. Questo implica che tali strutture non possano essere correttamente deallocate, in quanto la distruzione della prima è vietata dall'esistenza di un riferimento ad essa da parte della seconda, e viceversa.
Per ovviare a questo problema Rust offre lo smart pointer "Reference Conter", o Rc<T>, il quale è così strutturato:

La varibile Rc sullo stack consiste in un semplice puntatore, che punta ad una struttura sullo heap, composta da due contatori, chiamati rispettivamente Strong e Weak, e dal valore di tipo T tramitr Rc allocato. Rc permette di condividere tale valore sullo heap, in quanto la variabile Rc può essere clonata senza problemi, e la creazione di un nuovo puntatore al valore, porta all'aumento del contatore Strong. In questo modo, la distruzione di riferimenti causa un decremento del contatore Strong fino a 0, e quando giunge a 0, la struttura viene deallocata.
Il problema delle referenze cicliche viene risolto grazie al secondo contatore, Weak, che identifica il numero di Smart Pointer di tipo Weak, al dato allocato. Questi ultimi Smart Pointer si creano a partire da un Rc tramite il metodo Rc::downgrade(&Rc), e danno origine ad un puntatore alla struttura allocata, senza però implicarne l'accesso: difatti, un tramite un Weak non è possibile accedere al valore, se non tramite il metodo upgrade(), il quale può fallire se il valore referenziato è già stato allocato.

In questo modo è possibile creare una referenza ciclica, a patto che un riferimento sia di tipo Strong e uno di tipo Weak. Sta al programmatore implementare tale logica correttamente, considerando i puntatori quando un oggetto viene deallocato.

Un esempio di tale problema è qui illustrato

```
struct Node{
    val: i32,
    pointer: Option<Rc<Node>>
}

fn main(){
    let mut a = Node{val: 1, pointer: Rc::new(None)};
    let mut b = Node{val: 2, pointer: Some(Rc::downgrade(&a))};
    println!("a reads index {} from b", *(a.pointer.unwrap())); //reading 2
    println!("b reads index {} from a", b.pointer.unwrap().upgrade().unwrap()); //reading 1

    drop(a);
    println!("b reads index {} from a", b.pointer.unwrap().upgrade.unwrap());
}                                                                    //^^^ panics here cause a was deallocated from the heap
```

Commento:
ok
Attenzione: nell'esempio c'è solo un riferimento doppio a non cicliclo. poi non si può assegnare Weak a Rc (poointer)

Domanda 2

Completo

Punteggio ottenuto 3,0 su 3,0

Si identifichino i tratti fondamentali della concorrenza. Successivamente, in riferimento alla mutabilità/immutabilità delle risorse, si delinei come questa affligga la gestione della sincronizzazione a livello thread.

I tratti fondamentali della concorrenza in Rust sono Send e Sync. Send è implementato da tutti i tipi che possono essere trasferiti in un contesto multi-thread, mentre Sync è implementato da tutti i tipi T tali per cui &T implementa il tratto Send.

Il tema della mutabilità/immutabilità delle risorse in Rust è molto importante, in quanto se una risorsa è fornita ad una variabile con accesso mutabile, l'accesso ad essa è possibile solo a tale variabile, ed essa non può condiviedere il proprio lifetime con altre variabili che posseggono tale valore. Se l'accesso è dato in sola lettura invece, e ovvero il valore è condiviso in modo immutabile, possono coesistere più variabili allo stesso tempo che ne condividono l'accesso.

Tale problema diventa molto più complesso in un contesto multi-thread dove l'intersezione dei lifetime non è sempre predicibile e controllabile. Per questo motivo, tutti i metodi che rìpermettono di modificare una variabile in un contesto multi-thread implementano la interior mutability, ovvero chiedono il possesso del valore condiviso in modo immutabile, salvo poi poterlo modificare ugualmente. Tuttavia, com'è naturale, per garantire il corretto accesso a tale valore in modo corretto, e per assicurare l'impossibilità dell'insorgere di comportamenti non deterministici, tali valori condivisi, se appunto, mutabili, richiedono strutture specifiche che ne garantiscano la sicurezza dell'accesso, come i tipi Atomic e i Mutex.

Commento:
- Sync però vuol dire che può essere "condiviso" in generale

Domanda 3

Completo

Punteggio ottenuto 3,0 su 3,0

Data la struttura dati definita come

```
struct Data {
    Element: AsVector,
    next: Rc<Data>
}
enum AsVector {
    AsVector(Box::<Rc<i32>>),
    None
}
```

Indicare l’occupazione di memoria di un singolo elemento in termini di: 1. numero di byte complessivi (caso peggiore, architettura a 64 bit) e 2. posizionamento dei vari byte in memoria (stack, heap, ecc.).

Il Box è un puntatore ad un valore sullo heap, per cui la sua dimensione sullo stack è di 8 bytes.

Sullo heap il Rc allocato è un puntatore, anch'esso di 8 bytes, che però punta alla struttura dell'Rc, che possiede due contatori Strong e Weak (da 4/8 bytes, supponiamo qui 8 bytes), oltre che dalla dimensione del, valore referenziato. Essendo la struttura dell'Rc quindi 20 bytes, lo spazio totale occupato sullo heap sale a 28 bytes (20 struttura + 8 puntatore).

L'enum necessita di un byte per indicare la label, che si va a sommare alla dimensione dell'opzione di dimensioni maggiori sullo Stack, per cui la sua dimensione sarebbe in teoria di 9 bytes. In pratica bisogna considerare anche l'allineamento richiesto dal valore contenuto, ma i Box non hanno vincoli di allineamento, per cui 9 bytes è anche la dimensione reale dell'enum sullo stack.

La struct possiende sullo stack semplicemente dimensione pari alla somma delle dimensioni dei suoi campi, allineate all'allineamento più grande richiesto. il campo element richiede dunque 9 bytes, mentre il campo next è un Rc, che sullo stack è un puntatore che occupa 8 bytes, come già spiegato.

Sullo heap abbiamo i 28 bytes richiesti dall'enum, e altri 8+8+17=33 bytes richiesti dal contenuto dell'Rc.
In realtà se questa struct Data all'interno dell'RC è allocata, implica uno spazio aggiuntivo nello heap, ma possiamo fermarci qui nella considerazione, altrimenti il problema si propagherebbe ricorsivamente.

Pertanto l'intera struttura richiede 17 bytes sullo stack (dimensione della struct Data), e 33+28=61 bytes sullo heap, dati rispettivamente dai valori dentro il Box del campo "Element" e puntati dall'Rc del campo "next".

Commento:

La struct Data occupa 16 byte: 8 dedicati a element, 8 dedicati a next
Tale struttura può essere allocata sia nello stack che nello heap.
Element contiene alternativamente, un puntatore valido oppure 0 (None);
non c'è byte di tag, in quanto None non richiede alcuna informazione propria
e può essere dedotto come puntatore invalido.
Se presente, il puntatore punta ad un blocco di 8 byte, sullo heap contenente
un puntatore a 20 byte sullo heap (Rc<i32>)
next occupa 8 byte e punta ad un blocco di 32 byte posto sullo heap (Rc<Data>),
che verrebbe a contenere l'elemento successivo.
Considerando un solo elemento risultano quindi 16 byte allocabili sia sullo stack che
sullo heap per la struct Data, a cui si aggiungono 28 (8+20) byte sullo heap,
per un totale di 44 byte.

Domanda 4

Completo

Punteggio ottenuto 6,0 su 6,0

La struct MpMcChannel<E: Send> è una implementazione di un canale su cui possono scrivere molti produttori e da cui possono attingere valori molti consumatori.

Tale struttura offre i seguenti metodi:

- *new(n: usize) -> Self* //crea una istanza del canale basato su un buffer circolare di "n" elementi
- *send(e: E) -> Option<()>* //invia l'elemento "e" sul canale. Se il buffer circolare è pieno, attende
  //senza consumare CPU che si crei almeno un posto libero in cui depositare il valore
  //Ritorna:
  // - Some(()) se è stato possibile inserire il valore nel buffer circolare
  // - None se il canale è stato chiuso (Attenzione: la chiusura può avvenire anche
  // mentre si è in attesa che si liberi spazio) o se si è verificato un errore interno
- *recv() -> Option<E>* //legge il prossimo elemento presente sul canale. Se il buffer circolare è vuoto,
  //attende senza consumare CPU che venga depositato

almeno un valore
//Ritorna:
// - Some(e) se è stato possibile prelevare un valore dal buffer
// - None se il canale è stato chiuso (Attenzione: se, all'atto della chiusura sono
// già presenti valori nel buffer, questi devono essere ritornati, prima di indicare
// che il buffer è stato chiuso; se la chiusura avviene mentre si è in attesa di un valore,
// l'attesa si sblocca e viene ritornato None) o se si è verificato un errore interno.

- *shutdown() -> Option<()>* //chiude il canale, impedendo ulteriori invii di valori.
//Ritorna:
// - Some(()) per indicare la corretta chiusura
// - None in caso di errore interno all'implementazione del metodo.

Si implementi tale struttura dati in linguaggio Rust, senza utilizzare i canali forniti dalla libreria standard né da altre librerie, avendo cura di garantirne la correttezza in presenza di più thread e di non generare la condizione di panico all'interno dei suoi metodi.

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

struct MpMcChannel<E: Send>{
    c_buffer: Mutex<(ChanelState, Vec<E>)>,
    buffer_size: usize,
    cv: Condvar
}

impl<E: Send> MpMcChannel<E> {
    fn new(n: usize) -> Arc<Self> {
        return Arc::new(MpMcChannel{
            c_buffer: Mutex::new((ChannelState::Open, vec![])),
            buffer_size: n,
            cv: Condvar::new()
```

```
        })
    }

    fn send(&self, e: E) -> Option<()>{
        let mut try_lock = self.c_buffer.lock();
        if try_lock.is_err() {return None};
        let mut lock = try_lock.unwrap();
        try_lock = self.cv.wait_while(lock, | l |{ (*l).1.len == self.buffer_size && (*l).0 ==
ChannelState::Open}).unwrap();
        if try_lock.is_err() {return None}
        lock = try_lock.unwrap();
        if (*lock).0 == ChannelState::Closed {return None}
        (*lock).push(e);
        self.cv.notify_all();
        return Some(());
    }

    fn recv(&self) -> Option<E> {
        let mut try_lock = self.c_buffer.lock();
        if try_lock.is_err() {return None}
        let mut lock = try_lock.unwrap();
        try_lock = self.cv.wait_while(lock, | l | {(*l).1.len()==0 && (*l).0 == ChannelState::Open});
        if try_lock.is_err() {return None}
        lock = try_lock.unwrap();
        if (*lock).0 == ChannelState::Closed && (*lock).1.len() == 0 {return None}
        let e = (*lock).1.first().unwrap(); //non sono sicuro del nome del metodo, ma esiste un metodo
che permette di
                                            //fare pop() dalla testa del vattore se non sbaglio
        self.cv.notify_all();
        return Some(e)
    }

    fn shutdown() -> Option<()> {
        let try_lock = self.c_buffer.lock();
        if try_lock.is_err() {return None}
        let mut lock = try_lock.unwrap();
```

```
        (*lock).0 = ChannelState::Closed;
        self.cv.notify_all();
        return Some(())
    }

}
```

Commento:
Se usi un Vec<E> per implementare un buffer circolare, hai un costo O(N) all'atto dell'estrazione del messaggio più vecchio: per implementare questo tipo di dato si usa std::collections::VecDeque<E>.

Una soluzione basata sul tuo codice è:

```
use std::collections::VecDeque;
use std::sync::{Condvar, Mutex};

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

pub struct MpMcChannel<E: Send>{
    c_buffer: Mutex<(ChannelState, VecDeque<E>)>,
    buffer_size: usize,
    cv: Condvar
}

impl<E: Send> MpMcChannel<E> {
    pub fn new(n: usize) -> Self {
        return MpMcChannel{
            c_buffer: Mutex::new((ChannelState::Open, VecDeque::with_capacity(n))),
            buffer_size: n,
            cv: Condvar::new()
        }
    }
    pub fn send(&self, e: E) -> Option<()>{
        let mut try_lock = self.c_buffer.lock();
        if try_lock.is_err() {return None};
        let mut lock = try_lock.unwrap();
        try_lock = self.cv.wait_while(
            lock,
            | l |{
                l.1.len() == self.buffer_size && l.0 == ChannelState::Open
            }
        );
        if try_lock.is_err() {return None}
```

```
            lock = try_lock.unwrap();
            if lock.0 == ChannelState::Closed {return None}
            lock.1.push_back(e);
            drop(lock);
            self.cv.notify_all();
            return Some(());
        }

        pub fn recv(&self) -> Option<E> {
            let mut try_lock = self.c_buffer.lock();
            if try_lock.is_err() { return None }
            let mut lock = try_lock.unwrap();
            try_lock = self.cv.wait_while(
                lock,
                |l | {l.1.len()==0 && l.0 == ChannelState::Open
                }
            );
            if try_lock.is_err() {return None}
            lock = try_lock.unwrap();
            if lock.0 == ChannelState::Closed && lock.1.len() == 0 {return None}
            let e = lock.1.pop_front(); //non sono sicuro del nome del metodo, ma esiste un metodo che p
ermette di
            //fare pop() dalla testa del vattore se non sbaglio
            drop(lock);
            self.cv.notify_all();
            return e
        }

        pub fn shutdown(&self) -> Option<()> {
            let try_lock = self.c_buffer.lock();
            if try_lock.is_err() {return None}
            let mut lock = try_lock.unwrap();
            lock.0 = ChannelState::Closed;
            self.cv.notify_all();
            return Some(())
        }
    }
```