POLITECNICO
DI TORINO

Programmazione di sistema

Esame del 8/9/2022 (API Programming) (in presenza)

Domanda 1

Completo

Punteggio ottenuto 3,00 su 3,00

Si definiscano le principali aree di memoria associate ad un eseguibile e si mostri, attraverso opportuni esempi di codice, in quale situazione ciascuna di esse viene utilizzata.

Un eseguibile contiene le seguenti aree di memoria standard:

- segmento di codice: contiene le linee di codice del programma;

- segmento delle costant: a seconda del sistema operativo, potrebbe essere accorpato a quello di codice, contiene le costanti. Nel caso di Rust sono presenti le costanti standard e tutte le variabili con lifetime 'static(let MAX: <'static i32> = 10;)

- segmento delle variabili globali: contiene le variabili globali

- stack: Lo stack è una parte di memoria che è libera di crescere verso l'alto(in maniera contigua), e contiene le variabili locali. Ogni scope sintattico(racchiuso cioè tra 2 graffe) corrisponde ad uno stackframe, che contiene le variabili locali dichiarate in quel contesto. Ogni thread possiede un suo stack.

- heap: Parte di memoria associata a memoria dinamica, non è contigua. Contiene tutto ciò di cui si può stabilire la dimensione a runtime.

ESEMPIO - stack, heap

//Alloco sullo stack

let v = [1, 2, 3]; // viene salvato nello stack, in quanto è una variabile locale

let h_v = Vec::from(v); // Alloca nello heap un vettore delle stesse dimensioni di v. h_v è solo il puntatore(in questo caso fat_pointer, cioè con alcune info in più) alla struttura dati vera e propria.

Commento:
ok

Domanda 2

Completo

Punteggio ottenuto 2,00 su 3,00

Sia dato un primo valore di tipo std::cell::Cell<T> ed un secondo valore di tipo std::cell::RefCell<T> (dove T fa riferimento alla medesima entità). Si indichino le differenze tra i due e le modalità di occupazione della memoria (quantità, zone di memoria, ecc.).

Gli smartPointer Cell<T> e RefCell<T> sono due smart pointer che rust rende disponibili che implementano un meccanismo di mutabilità interna, cioè fanno in modo che le stringenti regole del borrow checker siano rispettate solo a runtime, e non a compile time. La differenza tra i due è semplicemente il fatto che Cell<T> non permette di creare riferimenti(e dunque prestiti) al valore contenuto nello smart pointer.

In termini di memoria Cell occupa esattamente quanto il valore T, ed è allocato sullo stack, mentre RefCell contiene due campi:

- borrow: indica se il valore è stato preso in prestito

- T.

Commento:

- Per Cell<T> sono sempre a compile time
- non è ben chiaro però come funziona la mutabilità interna, cioè che fanno in pratica

Nota per tutti: Cell<T> occupa lo stesso spazio e zona di memoria di T, mentre RefCell<T> di T + assieme Cell<usize> per il borrow counter e con il wrap cambiano solo le regole di accesso; quindi possono essere interamente sullo stack. A lezione sono stati trattati assieme agli smart pointer e il concetto può non essere stato chiaro, quindi errori su questo punto non sono valutati

Domanda 3

Completo

Punteggio ottenuto 3,00 su 3,00

In un programma che utilizza una sincronizzazione basata su Condition Variable, è possibile che alcune notifiche vengano perse? Se sì, perché? In entrambi i casi si produca un esempio di codice che giustifichi la risposta.

Si, è possibile che alcune notifiche vengano perse, ad esempio nel caso in cui lo scheduling deciso faccia eseguire ad un thread la notifica prima che l'altro thread si metta effettivamente in attesa. Ad esempio t1 ad un certo punto esegue:

```
{
    let mut mutex = mutex.lock().unwrap()
    mutex = cv.wait(mutex).unwrap();
}
```

Mentre il secondo thread:

```
{
    cv.notify_one();
}
```

Questo comportamento va assolutamente evitato, in quanto porta ad errori casuali(dipendenti dalla politica attuale di scheduling, dal carico), e Rust offre un opportuno costrutto sintattico per risolvere il problema:

t1

```
{
    let mut mutex= mutex.lock().unwrap();
    while (condition(*mutex)) {
        mutex = cv.wait(mutex).unwrap()
    }
}
```

Oppure, usando un metodo delle condition variable, che riproduce lo stesso effetto:

```
{
    let mut mutex = mutex.lock().unwrap();
    mutex = cv.wait_while(mutex, |m| condition(m) ).unwrap();
}
```

t2

```
{
    cv.notify_one()
}
```

In questo modo, se l'azione che dovrebbe notificare il risveglio è stata già fatta, il thread non si mette in attesa.

Commento:
ok

Domanda 4

Completo

Punteggio ottenuto 5,50 su 6,00

In un sistema concorrente, ciascun thread può pubblicare eventi per rendere noto ad altri thread quanto sta facendo.

Per evitare un accoppiamento stretto tra mittenti e destinatari degli eventi, si utilizza un **Dispatcher**: questo è un oggetto thread-safe che offre il metodo

```
dispatch(msg: Msg)
```

mediante il quale un messaggio di tipo generico **Msg** (soggetto al vincolo di essere clonabile) viene reso disponibile a chiunque si sia sottoscritto. Un thread interessato a ricevere messaggi può invocare il metodo

```
subscribe()
```

del Dispatcher: otterrà come risultato un oggetto di tipo **Subscription** mediante il quale potrà leggere i messaggi che da ora in poi saranno pubblicati attraverso il Dispatcher. Per ogni sottoscrizione attiva, il Dispatcher mantiene internamente l'equivalente di una coda ordinata (FIFO) di messaggi non ancora letti. A fronte dell'invocazione del metodo dispatch(msg:Msg), il messaggio viene clonato ed inserito in ciascuna delle code esistenti.

L'oggetto Subscription offre il metodo bloccante

```
read() -> Option<Msg>
```

se nella coda corrispondente è presente almeno un messaggio, questo viene rimosso e restituito; se nella coda non è presente nessun messaggio e il Dispatcher esiste ancora, l'invocazione si blocca fino a che non viene inserito un nuovo messaggio; se invece il Dispatcher è stato distrutto, viene restituito il valore corrispondente all'opzione vuota.

Gli oggetti Dispatcher e Subscription sono in qualche modo collegati, ma devono poter avere cicli di vita indipendenti: la distruzione del Dispatcher non deve impedire la consumazione dei messaggi già recapitati ad una Subscription, ma non ancora letti; parimenti, la distruzione di una Subscription non deve impedire al Dispatcher di consegnare ulteriori messaggi alle eventuali altre Subscription presenti.

Si implementino le strutture dati Dispatcher e Subscription, a scelta, nel linguaggio Rust o C++11.

// Il dispathcer mantiene un certo numero di canali aperti, pari al numero di volte in cui è stata fatta la subscription
// Il vettore è protetto da un mutex, per permettere a thread diversi di accedere in sicurezza al

```
vettore.
// Ad esempio, non vogliamo che mentre si sta mandando un messaggio a 10 sender venga
aggiunto un 11 esimo.

// Msg deve implementare anche Send per essere mandato sul canale, mentre 'static è richiesto in
quanto non vogliamo che il messaggio cambi tra quando è stato inviato e quando è ricevuto
pub struct Dispatcher<Msg: Clone + Send + 'static> {
  sender: Mutex<Vec<Sender<Msg>>>
}

pub struct Subscription<Msg: Clone + Send + 'static> {
  rx : Receiver<Msg>
}

impl<Msg: Clone + Send + 'static> Dispatcher<Msg> {
  pub fn new() -> Arc<Self> {
    Arc::new(
      Mutex::new(Vec::new::<Sender<Msg>>())
    )
  }

  pub fn subscribe(&self) -> Subscription<Msg> {
    // creo il canale
    let (tx, rx) = channel();
    // Aggiungo il sender al Dispatcher e restituisco un oggetto Subscription.
    (*self.rx.lock().unwrap()).push(tx);
    Subscription{rx}
  }

  pub fn dispatch(&self, msg: Msg) {
    let senders = self.tx.lock().unwrap();
    for tx in (*senders) {
      match tx.send(msg.clone()){
        Ok(_) => { ()// il messaggio è stato inviato correttamente
        },
        Err(_) => { () // Se è qui significa che il sender associato è stato droppato. In questo caso
fa niente, ignoriamo l'errore, e proseguiamo con il prossimo Sender}
      };
    }
  }
}

impl<Msg: Clone + Send + 'static> Subscription<Msg> {
  pub fn read(&self) -> Option<Msg> {
    // La semantica del receiver ci permette di bloccarci in attesa di un messaggio, mentre se il
sender è stato droppato viene tornato errore.
    match self.rx.recv() {
```

```
        Ok(msg) => Some(msg),
        Err(_) => None
      }
    }
}
```

Commento:
Nel caso una subscription sia stata eliminata, non rimuovi il corrispondente Sender dal Dispatcher, generando un possibile leaking e consumando cicli inutili nelle successive invocazioni del metodo dispatch(...).

Questa una versione corretta basata sul tuo codice:

```
pub mod dispatcher {
    // Il dispathcer mantiene un certo numero di canali aperti, pari al numero di volte in cui è stata fatta la subscription
// Il vettore è protetto da un mutex, per permettere a thread diversi di accedere in sicurezza al vettore.
// Ad esempio, non vogliamo che mentre si sta mandando un mes saggio a 10 sender venga aggiunto un 11 esimo.

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

    // Msg deve implementare anche Send per essere mandato sul canale, mentre 'static è richiesto in quanto non vogliamo che il me ssaggio cambi tra quando è stato inviato e quando è ricevuto
    pub struct Dispatcher<Msg: Clone + Send + 'static> {
        sender: Mutex<Vec<Sender<Msg>>>
    }

    pub struct Subscription<Msg: Clone + Send + 'static> {
        rx : Receiver<Msg>
    }

    impl<Msg: Clone + Send + 'static> Dispatcher<Msg> {
        pub fn new() -> Arc<Self> {
            Arc::new(
                Dispatcher {
                    sender: Mutex::new(Vec::new())
                }
            )
        }

        pub fn subscribe(&self) -> Subscription<Msg> {
            // creo il canale
            let (tx, rx) = channel();
```

```
        // Aggiungo il sender al Dispatcher e restituisco un oggetto Subscription.
        (*self.sender.lock().unwrap()).push(tx);
        Subscription { rx }
    }

    pub fn dispatch(&self, msg: Msg) {
        let mut senders = self.sender.lock().unwrap();
        for i in (0..senders.len()).rev() {
            match senders[i].send(msg.clone()) {
                Ok(_) => {
                    ()// il messaggio è stato inviato correttamente
                },
                Err(_) => {
                    senders.remove(i); // Se è qui significa che il sender associato è stato droppato. In questo caso fa niente, ignoriamo l'errore, e proseguiamo con il prossimo Sender}
                },
            }
        }
    }
}

impl<Msg: Clone + Send + 'static> Subscription<Msg> {
    pub fn read(&self) -> Option<Msg> {
        // La semantica del receiver ci permette di bloccarci in attesa di un messaggio, mentre se il sender è stato droppato viene tornato errore.
        match self.rx.recv() {
            Ok(msg) => Some(msg),
            Err(_) => None
        }
    }
}
```