POLITECNICO DI TORINO

Programmazione di sistema

Esame del 26/10/2022 (API Programming) (in presenza)

	GIOVANNI DE FLORIO 292457
Iniziato	mercoledì, 26 ottobre 2022, 19:00
Terminato	mercoledì, 26 ottobre 2022, 20:30
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

Si illustrino le differenze tra stack e heap. Insieme alle differenze, indicare per i seguenti costrutti Rust, in modo dettagliato, dove si trovano i dati che li compongono: Box<[T]>, RefCell<T> e &[T].

Lo stack serve per allocare variabili e nello steck oltre alle variabili sono contenuti i valori per i passaggi a funzione e i valori di ritorno a funzione, non va bene per le allocazioni a runtime ma solo a compile time (ogni thread ha un suo stack ma l'heap di base è comune al processo) al ritorno da funzioni lo stack si contrae.

La differenza tra lo stack e lo heap è che lo heap serve per allocare dati dinamici a runtime.

Box<[T]>: stack: puntatore al primo elemento del vettore, campo len che indica la lunghezza del vettore in heap heap: vettore di elementi di tipo T

RefCell<T>: stack: campo borrow, dato T

&[T]: stack: puntatore al primo elemento del vettore in memoria, campo len con lunghezza del vettore di T

Commento:
85%
- lo stack non è allocato a compile time, solo la lunghezza dei dati è nota a compile time
- altra differenza la durata del dato
- [T] è uno slice, per la precisione, ok la memoria (il dato può essere su stack o heap, indifferentemente)

Domanda 2

Completo

Punteggio ottenuto 2,5 su 3,0

Un sistema concorrente può essere implementato creando più thread nello stesso processo, creando più processi basati su un singolo thread o basati su più thread. Si mettano a confronto i corrispettivi modelli di esecuzione e si evidenzino pregi e difetti in termini di robustezza, prestazioni, scalabilità e semplicità di sviluppo di ciascuno di essi.

La concorrenza di fatto prevede due possibilità: il meccanismo di concorrenza in single core, dove fondamentalmente ci si contende la risorsa core con lo scheduler che interrompe e fa partire thread, poi c'è la concorrenza in multicore che permette un parallelismo più spinto ossia un parallelismo vero e proprio.

Dunque la concorrenza nasce con l'idea di parallelizzare i compiti. Un processo con più thread plausibilmente permetterà di parallelizzare i compiti (se si dispone di cpu multicore e la sincronizzazione è correttamente gestita es. non ci sono thread globabi che serializzino i compiti) e dunque svolgerli in più breve tempo rispetto ad un approccio mono-thread; va detto che agli inizi non esisteva neanche il concetto di thread (ogni processo era monothread) e si utilizzavano diversi processi e si facevano comunicare tra di loro, il problema però è che i thread condividono lo spazio di indirizzamento (nel contesto di un processo) mentre i processi non condividendo lo spazio di indirizzamento e avendo un isolamento parziale (spazio di indirizzamento non condiviso, ma condivisi file system, periferiche e porte di rete a meno che il processo non sia su un container tipo docker in cui si condivide il kernel ma non tutto il resto). Dunque il problema di questo approccio multiprocesso ma monothread crea una difficoltà di comunicazione(i ptr non hanno senso in un altro spazio di indirizzamento) più costosa perchè ci si deve preoccupare di tradurre i dati con il marshalling.

Va detto che il problema della concorrenza (immaginiamo un contesto multithread), pone delle problematiche come quelle di ordinamento, visibilità e atomicità, che sono dovute fondamentalmente al modello gerarchico delle cache dei processori multicore.

Dunque occorrono delle istruzioni a livello hardware test-and-set, compare-and-swap e barriere di memoria che poi in linguaggi più ad altro livello vengono utilizzate tramite costrutti detti primitive di sincronizzazione (Atomic, Mutex e Condition Variable) e anche qui se non si fa attenzione a utilizzarle adeguatamente si possono creare starvation o addirittura deadlock. In questo Rust ci aiuta poichè ad esempio mette in stretta correlazione la risorsa mutex al valore da proteggere permettendo di fatto al compilatore di segnalarci eventuali problemi (esempio se non stiamo proteggendo adeguatamente una variabile) già in fase di compilazione (fearless concurrency).

In un contesto in cui si utilizzano più processi che comunicano ed ogni processo è multi-thread si utilizzera un approccio di scambio di messaggi o condivisione dello stato all'interno dello stesso processo mentre ci si dovrà limitare allo scambio di messaggi tra processi, per il fatto dell'isolamento (ad esempio si potrebbero usare channel tra diversi thread dello stesso processo e pipe o code di messaggi tra diversi processi): realizzazione più complessa in termini di sviluppo.

Commento:
- robustezza e scalabilità?

Domanda 3

Completo

Punteggio ottenuto 3,0 su 3,0

In riferimento a programmi multi-thread, si indichi se la natura della loro esecuzione sia determinisitca o meno a priori. Si produca un esempio che dimostri tale affermazione.

In programmi multithread la natura (senza sincronizzazione opportuna) non è deterministica si possono ottenere comportamenti imprevedibili ed è la ragione per cui è necessario utilizzare adeguatamente costrutti di sincronizzazione (adeguatamente altrimenti si richia il deadlock). Un esempio che dimostra l'imprevedibilità di un approccio multi-thread è quello dell'interferenza, possiamo immaginare ad esempio una istruzione apparentemente innocua come può essere quella di un increment, a++.

Sembra una istruzione sola (atomica) di fatto questa si traduce in due istruzioni almeno

temp=a;

a=temp+1;

il problema di questo, che porterà poi alla conclusione che non si possono lasciare operazioni in stati intermedi e che non si può fare accesso in lettura/scrittura da più thread contemporaneamente (race condition) è ricollegabile di fatto al modello gerarchico delle memorie dove essendoci cache a diversi livelli es fino al livello 2 non comunicano tra i cori, ci saranno dei tempi di propagazione e quindi di visibilità tra diversi thread di un valore in memoria.

Quindi quello che può succedere è che un thread salvi il valore di a in temp in una sua locazione di memoria quindi, e fintanto che questo cambiamento sia visibile da un altro thread che vuole operare su quel valore quello che può succedere è che lo scheduler freeze questo primo thread e lo lasci dormiente intanto sveglia un altro thread che fa la stessa operazione salva in un indirizzo di memoria a e poi fa gli increment, questo magari lo fa un pò di volte se sono chiamati più increment. Ma quando viene svegliato il primo thread di fatto riprende la sua esecuzione e mette in a il valore del temp che lui aveva incrementato di 1; quindi stampando il valore sembra che sia tornato indietro. Dunque l'interferenza è uno degli esempi di non determinismo che si possono avere senza i costrutti di sincronizzazione. Un esempio in Rust (che non permette a++) è quello di Rc che avendo i contatori non atomici dei riferimenti (Strong e Weak) non è ne Send ne Sync, infatti per questo è stato creato Arc con i contatori atomici che permettono increment e decrement atomici (operazioni più costose ma necessarie nel multithread)

Commento:
Ok

Domanda 4

Completo

Punteggio ottenuto 5,0 su 6,0

Un componente con funzionalità di cache permette di ottimizzare il comportamento di un sistema riducendo il numero di volte in cui una funzione è invocata, tenendo traccia dei risultati da essa restituiti a fronte di un particolare dato in ingresso.
Per generalità, si assuma che la funzione accetti un dato di tipo generico K e restituisca un valore di tipo generico V.

Il componente offre un unico metodo get(...) che prende in ingresso due parametri, il valore k (di tipo K, clonabile) del parametro e la funzione f (di tipo K -> V) responsabile della sua trasformazione, e restituisce uno smart pointer clonabile al relativo valore.

Se, per una determinata chiave k, non è ancora stato calcolato il valore corrispondente, la funzione viene invocata e ne viene restituito il risultato; altrimenti viene restituito il risultato già trovato.

Il componente cache deve essere thread-safe perché due o più thread possono richiedere contemporaneamente il valore di una data chiave: quando questo avviene e il dato non è ancora presente, la chiamata alla funzione dovrà essere eseguita nel contesto di UN SOLO thread, mentre gli altri dovranno aspettare il risultato in corso di elaborazione, SENZA CONSUMARE cicli macchina.

Si implementi tale componente a scelta nei linguaggi C++ o Rust.

```
pub struct Cache<K:Clone,V>{
 map:Mutex<HashMap<K,V>>
}

impl <K:Clone,V> Cache<K,V>{
    pub fn new()-> Self{
    Cache{Mutex::new(HashMap::new())}
    }

    pub fn get<K:Clone, V>(&self,k:K,f:fn(K)->V)-> Arc<V>{
     let map=self.map.lock().unwrap();

     let b:bool=map.found_value(k);     //setto un flag per vedere se trovo una corrispondenza chiave/valore

     if (b!=true)                    //se non trovo vuol dire che non ho ancora chiamato la funzione su k
     let res=f(k)
     let r=Arc::new(res);
     map.push(k,res);
     return r;
```

```
    }else{                      //se trovo il valore lo restituisco dentro Arc
     let res=map.value(k);
     let r=Arc::new(res);
     return r;
    }

   }

}
```

Commento:
La mappa possiede i dati contenuti al suo interno: non si può derivare un Arc<V> da tali dati , perché si creerebbero due possessori per lo stesso valore. Occorre che la mappa contenga valori di tipo Arc<V>, così che possano essere clonati.

Con la tua strategia, la struttura resta bloccata per tutta la durata del metodo get(...) anche durante l'eventuale esecuzione della funzione (che può essere lenta), impedendo di fatto ad altri thread di accedere ai dati contenuti.

Una possibile soluzione all'esercizio è

```
pub mod cache {
use std::collections::hash_map::Entry::{Occupied, Vacant};
use std::collections::HashMap;
use std::hash::Hash;
use std::sync::{Arc, Condvar, Mutex};
use crate::cache::EntryState::{Pending, Present};

#[derive(Eq, PartialEq)]
    enum EntryState<V> {
Pending,
Present(Arc<V>)
    }

pub struct Cache<K: Eq + Hash + Clone, V> {
m: Mutex<HashMap<K, EntryState<V>>>,
cv: Condvar,
    }


impl<K: Eq + Hash + Clone,V> Cache<K,V> {
pub fn new() -> Self {
            Cache {
                m: Mutex::new(HashMap::new()),
                cv: Condvar::new(),
            }
        }
```

```
pub fn get(&self, k:K, f: impl FnOnce(&K) -> V +'static) -> Arc<V
> {
let mut map = self.m.lock().unwrap();
loop {
match map.entry(k.clone()) {
Occupied(e) => {
match e.get() {
Pending => {
                            map = self.cv.wait_while(
                                map,
                                |map| {
if let Occupied(e) = map.entry(k.clone()) {
if let Present { .. } = e.get() { return false }
                                    }
true
                                }).unwrap();
                        }
Present(v) => {
return v.clone();
                        }
                    }
                }
Vacant(e) => {
                    e.insert(Pending);
                    drop(map);
let v = Arc::new(f(&k));
                    map = self.m.lock().unwrap();
                    map.entry(k.clone())
                        .and_modify(|e| {*e= Present(v.clone());});
self.cv.notify_all();
return v;
                }
            }
        }
    }
}
```