POLITECNICO DI TORINO

Programmazione di sistema

Esame 26 Ottobre 2022 - Programming

	DAVIDE MANCA 288034
Iniziato	mercoledì, 26 ottobre 2022, 19:03
Terminato	mercoledì, 26 ottobre 2022, 20:30
Tempo impiegato	1 ora 26 min.
Valutazione	**10,50** su un massimo di 15,00 (**70**%)

Domanda 1

Completo

Punteggio ottenuto 3,00 su 3,00

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 i thread di per sè hanno un esecuzione non deterministica che rappresenta uno svantaggio perchè a ogni esecuzione ho un risultato diverso. Da un punto di vista architetturale garantire operazioni atomiche per cui non mi ritrovo risultati inattesi (atomicità, visibilità, ordinamento, barriere per sincronizzazione dati da cache in memoria) Inoltre il non determinismo è dovuto al fatto che lo scheduler quando prende dei thread dalla coda not runnable e li mette nella coda dei runnable (per poi essere messi nella running queue) lo fa casualmente senza nessun meccanismo di priorità nè di sincronizzazione (se non viene toccata la sincronizzazione).
Dunque affinchè rispettino un comportamento deterministico devo io programmatore implementare la parte di sincronizzazione che consiste nel far produrre dei dati a un thread che poi devono essere usati da un altro thread (oppure se un thread sta operando su un dato nessun altro deve andare a toccarlo fin quando il thread in esecuzione non ha finito) e quindi si mettono in piedi i meccanismi di mutua esclusione all'accesso alle zone di memoria condivise, e di attesa sincronizzata. Per cui un thread quando deve aspettare una condizione si mette in attesa (acquisendo mutex e rilasciandolo quando va a dormire) in modo che un altro thread possa soddisfare la condizione per cui il thread stava aspettando e notificare il medesimo che la condizione si è verificata. (meccanismo di condition variable)

```
pun fn f(){
pub struct s{
    m: mutex<bool>
   cv: Condvar
}
```

```
let s1 = Arc::new(s{Mutex::new(false), Condvar::new()});
let s2 = Arc::clone(&s1);

let state_s1 = s1.m.lock().unwrap();
while(*state_s1 != false){
    state_s1 = self.cv.wait(state_s1).unwrap();
}
let state_s2 = s2.m.lock().unwrap();
*state_s2 = true;
s2.cv.notify_one()
}
```

In questo modo avendo sincronizzato sono sicuro che l'esecuzione sarà deterministica

Commento:

Domanda 2

Completo

Punteggio ottenuto 3,00 su 3,00

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.

Un processo ha un suo spazio di indirizzamento che è condiviso con i suoi thread. Gestire un sistema concorrente in questo modo implica gestire efficaciemente i thread. Posso dividere la computazione del processo in più flussi da assegnare a ciascun thread in modo da avere una parallelizzazione di programmazione e guadagnarne in termini di velocità e di prestazioni computazionali. Dal punto di vista architetturale cio è fatto assegnando i thread ai core della cpu. E' necessario che il codice dei non abbia grandi dimensioni altrimenti perderei quel vantaggio ottenendo lo svantaggio di avere molta complessità dal punto di vista programmativom un codice molto corposo per il quale non avrebbe senso dividere il processo di base in più flussi.
Programmare con i thread significa che se non vengono gestite le cose correttamente si possono avere situazioni anomale per cui può essere gestita male la mutua esclusività a strutture dati in zone di memoria condivise, si possono avere comportamenti non deterministici se non si effettua correttamente la sincronizzazione (responsabilità del programmatore) per cui non si può neanche effettuare il debug se qualcosa va storto perchè ogni volta ho un comportamento differente.
Dunque devo gestire correttamente ed efficientemente sincronizzazione e mutua esclusione.
I thread terminano da soli perchè è meglio rispetto che a farli terminare di nostra volontà (se ad esempio uno stava scrivendo un file viene interrotta a metà la scrittura => situazione evitabile).Ritornano un handle o un errore. Se ritornano un eccezione se è nel main thread viene buttato giù tutto il processo altrimenti solo il thread secondario(in rust). In c++ qualsiasi thread

abbia questo comportamento butta giù tutto il processo.

Creare più processi basati su un singolo thread implica che il processo è gestito dal main thread per cui è come far comunicare più processi tra loro.
Ogni processo ha un pid che corrisponde ha un PCB allocato nel sistema operativo in cui mi dice se le zone di memoria sono leggibili, scrivibili, eseguibili e in più mi fornisce informazioni sull'isolamento importante per garantire spazi di indirizzamento diversi fra processi e per far si che la memoria non sia operabile se le operazioni su di essa non sono atomiche.
Gestire più processi con un singolo thread è meno efficiente che avere processi multithreaded perchè la complessità e il carico computazionale spetta a un solo thread. Dall'altra parte devo saper gestire i thread correttamente come detto poc'anzi.
La comunicazione fra processi è fatta tramite IPC. Ogni processo ha la sua rappresentazione interna che può essere tipi semplici (int, bool), tipi più strutturati (strutture, enum) e puntatori grezzi (alberi, grafi). Quest'ultimo caso è il peggiore perchè non posso trasferire un puntatore da un processo all'altro, essi hanno spazii di indirizzamento diversi.
Dunque attraverso meccanismi offerti dal sistema operativo come le pipe metto in comunicazione due o più processi dove i dati vengono scritti in binario e ne si deve conoscere la dimensione (la pipe non può trasportare grandi quantità).
Dalla rappresentazione interna si deve applicare un opportuna conversione a un rappresentazione esterna (JSON, XML, CSV) in modo da serializzare i dati e trasportarli all'altro processo che ne fa la deserializzazione (li estrae per convertirli nella sua rappresentazione interna). Il lavoro è svolto mediante opportune funzioni di conversione della serializzazione/deserializzazione (più facile in rust col crate serde.js, impossibile in c perchè non ho ne classi ne tratti).

Se più processi hanno più thread bisogna fare attenzione nel momento in cui viene creata una fork() e bisogna mantenere gli stati congruenti. Ad esempio banalmente se il padre è multithreading dopo la fork il figlio condivide col padre i riferimenti alle stesse pagine di memoria. Quando uno decide di modificare qualcosa scatta il COW (si fa una copia che possiede e che si modifica per sè) e se dei thread acquisiscono dei lock su dei mutex, lo stato viene visualizzato nel figlio (perchè i dati sono condivisi) ma avendo due spazi di indirizzamento diversi i thread non sono duplicati nel figlio e quindi non posso rilasciarli. Quindi c'è bisogno di meccanismi che garantiscono degli stati consistenti e congruenti (in c++ la funzione phtread_atfork() che ha come argomenti 3 funzioni *prepare(), *parent(), *child() che garantiscono quei meccanismi e quindi una maggiore robustezza

Commento:

Domanda 3

Completo

Punteggio ottenuto 1,50 su 3,00

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]

Box<[T]> è uno smart pointer che detiene il possesso della risorsa in maniera esclusiva. Il dato in questione
è un vettore. Non sono ammesse copie, di conseguenza l'assegnazione consiste nel movimento. Nessun altra risorsa
può possedere gli indirizzi del vettore,
Vi è un puntatore nello stack che punta al primo elemento del vettore(con elementi di tipo T) che sta nello heap.
Quindi nello stack ho un puntatore allo heap che punta al vettore

RefCell<T> a differenza di Cell<T> è usabile in contesti thread-safe. Permette di avere un contenitore immutabile
con un contenuto mutabile, in modo da passare i check che fa il borrow checker in termini di immutabilità di
contenitore e di avere allo stesso tempo un contenuto che può essere alterabile. In rust non ho altri modi di
ottenere questo comportamento. Inolte RefCell<T> offre dei metodi per poter referenziare il dato in modo condivisibile.
I metodi sono borrow() e borrow_mut() che restituiscono rispettivamente un riferimento immutabile e uno mutabile.
In termini di memoria lo smart pointer sta nello stack e ha un campo borrow e un campo T: T è il dato e il borrow è
un Cell<usize> che indica il borrow counter

&[T] indica la slice, un riferimento a un vettore di elementi di tipo T generico che si trova nello stack. Il puntatore
è un fat pointer con puntatore allo start della sequenza e dimensione della slice. E' un riferimento immutabile.
Si può scegliere un inizio arbitrario della sottosequenza senza fare operazioni di corrsipondenza col verore

Commento:
Ma le differenze tra heap e stack??

Box<[T]> deve contenere anche una dimensione.

Domanda 4

Completo

Punteggio ottenuto 3,00 su 6,00

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 mod cache{

    pub struct Componente<K:Clone, V>{
        m: Mutex<HashMap<K,V>>,
        cv: Condvar
    }

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

        pub fn get(&self, k:K, f:F) where F:Fn(k) -> V -> Arc<V>{
            let state = self.m.lock().unwrap();
            if(*state.get(k).is_none()){
                let res = f(k);
                *state.cv.wait(state).unwrap();
            }else{
```

```
                let res = *state.get(k);
                *state.cv.notify_one();
            }
            Arc::new(res)
        }

    }
}
```

Commento:
La soluzione è troppo semplicistica perché non tiene in considerazione il fatto che più thread, contemporaneamente, possano richiedere la stessa funzione...