Corso di Sistemi Operativi e Reti

Corso di Sistemi Operativi

Prova scritta 28 Giugno 2022

ERRORI GRAVI PIU' COMUNI

PER L'ESERCIZIO DI PROGRAMMAZIONE MULTITHREAD

ESERCIZIO 1 - PROGRAMMAZIONE MULTITHREADED

Punto 1
Si modifichi il codice di prova introducendo un secondo valore condiviso, chiamato `dc2`*, di tipo* `ReadWriteLockEvoluto`*. Si lancino più istanze di thread di tipo* `Copiatore`*. Un thread* `Copiatore` *periodicamente sceglie a caso un valore tra* `dc` *e* `dc2` *e lo copia sull'altro elemento non sorteggiato. Ad esempio, se viene scelto* `dc`*, allora bisogna effettuare l'operazione* `dc2.dato=dc.dato`*. Altrimenti bisognerà fare l'operazione* `dc.dato = dc2.dato`*.*

ERRORE GRAVE:

Non accorgersi che una cattiva implementazione di Copiatore può provocare il deadlock, come nel seguente spezzone di codice:

```
self.dc2.acquireWriteLock()
self.dc.acquireWriteLock()
self.dc2.setDato(self.dc.getDato())
self.dc.releaseWriteLock()
self.dc2.releaseWriteLock()
```

Se un altro Copiatore prende prima il lock su dc e poi su dc2 quasi in contemporanea, il deadlock è praticamente inevitabile.

ERRORE GRAVISSIMO

Implementare il Copiatore con una palese race condition, come nel seguente spezzone di codice:

```
self.dc.acquireReadLock()
val = self.dc.getDato()
self.dc.releaseReadLock()
self.dc2.acquireWriteLock()
self.dc2.setDato(val)
self.dc2.releaseWriteLock()
```

E’ evidente che il valore di val potrebbe non riflettere più il valore di dc.getDato() nel momento in cui si esegue self.dc2.setDato(val)

ALTRI PASTICCI

-Modificare ReadWriteLockEvoluto in maniera tale da avere self.dato2, getDato2() e setDato2(). Una lettura attenta del Punto 1 dovrebbe chiarire che non era questo che si chiedeva.
-Introdurre un nuovo metodo acquireCopiatore(dc2) implementato *senza prendere nessun lock su dc2*, dunque non disciplinando l’accesso contemporaneo di Lettori e Scrittori che usino dc2 in contemporanea ai Copiatori.

Punto 2

Modificare `getDato` *e* `setDato` *in maniera tale da provocare l'eccezione* `NoLockAcquired` *allorquando questi metodi vengono invocati senza che si possieda il lock corretto. Si noti che il possesso del write lock deve consentire di invocare sia* `getDato` *che* `setDato`*, mentre il possesso del read lock deve consentire di invocare solamente* `getDato`*.*

ERRORE GRAVE

Il punto 2 ci chiede di accertarsi che *il thread corrente* sia in possesso del write lock (per setDato) oppure del read lock o del write lock (nel caso di getDato).

Questa modifica a getDato è scorretta:

```
def getDato(self):
     if self.numLettori == 0 and not self.ceUnoScrittore:
         print("IL DATO NON PUO' ESSERE MODIFICATO O LETTO SENZA POSSEDERE IL LOCK CORRETTO")
         raise NoLockAcquired
     return self.dato
```

Poichè verifica che il lock sia preso *da qualcuno, altrimenti invoca NoLockAcquired*. ***Cosa succede se questo qualcuno non è il thread che correntemente esegue il codice di getDato?***

ERRORE GRAVISSIMO

Ho insistito molto sul fatto che l'interfaccia di ReadWriteLockEvoluto non si potesse cambiare perché questo compromette il funzionamento di tutto il codice pre-esistente che faccia uso di ReadWriteLockEvoluto. Fare modifiche all'interfaccia di ReadWriteLockEvoluto era inoltre *inutile*. Questo spezzone di codice:

```
    def acquireReadLock(self, id):
        with self.lockAusiliario:
            while self.ceUnoScrittore or self.numLettori >= self.max_readers:
                self.conditionAusiliaria.wait()
            self.numLettori += 1
            self.lettori.append(id)
```

E' equivalente a

```
    def acquireReadLock(self):
        with self.lockAusiliario:
            while self.ceUnoScrittore or self.numLettori >= self.max_readers:
                self.conditionAusiliaria.wait()
            self.numLettori += 1
            self.lettori.append(getThreadId())
```

ma non impone di stravolgere tutto il codice per gestire il parametro aggiuntivo che si è introdotto. E' importante tenere presente che `current_thread().ident` (e quindi anche la funzione `getThreadId()` usata nel codice) restituisce il Thread Identifier (TID) del thread che si sta eseguendo correntemente **indipendentemente** da quale sia il punto del codice in cui ci si trova. Il thread corrente può trovarsi dentro run(), o dentro *qualsiasi altra porzione di codice, inclusa l'implementazione di ReadWriteLock*.

RICORDATE INOLTRE CHE ReadWriteLockEvoluto E' UNA STRUTTURA DATI COMPOSTA DI CODICE E CAMPI INTERNI, NON UN THREAD, E DUNQUE NON POSSIEDE UN TID!

ALTRI PASTICCI

-Non avere considerato che gli scrittori *possono* eseguire anche getDato();
-Avere lanciato NoLockAcquired dentro getDato/setDato solo nel caso in cui il lock è totalmente libero;

-Avere pensato che per poter eseguire setDato basta che il thread corrente non sia nella lista dei lettori (!) senza verificare di essere lo scrittore attuale;

-Avere sbagliato l'espressione booleana che definisce quando lanciare NoLockAcquired. Esempio:

```
    def getDato(self):
        with self.lockAusiliario:
            if getThreadId() not in self.readers OR getThreadId() != self.writer:
                raise NoLockAcquired

        return self.dato
```

è sbagliata. L'espressione corretta prevede una congiunzione (AND).

-Non avere sincronizzato su lockAusiliario l'accesso alle liste dei lettori in possesso dei lock, e l'accesso al thread ID del lettore corrente. Queste variabili sono aggiornate e lette da più thread in contemporanea e necessitano di una qualche forma di sincronizzazione.

Punto 3

Si noti che se lo stesso thread T invoca per due volte consecutive acquireWriteLock, T si blocca *in attesa di sè stesso*. Lo stesso problema si verifica se uno stesso thread invoca tante volte acquireReadLock, fino a saturare il numero di lettori disponibili, oppure quando uno scrittore, già in possesso del lock in scrittura, prova ad acquisire il lock in lettura.
Si modifichi il `readwritelockevoluto` in maniera tale da ignorare eventuali invocazioni consecutive di *acquireReadLock o acquireWriteLock*, così rendendo il `readwritelockevoluto` *rientrante*.

Esempio:

```
1.dc.acquireWriteLock()
```

```
2.dc.acquireWriteLock()
3.prints("Che voglia di stampare che ho")
```

Con il sorgente fornito, un thread si bloccherebbe per sempre sul rigo 2. Questo non deve succedere, la seconda chiamata ad acquireWriteLock deve terminare immediatamente senza attese, poiché il thread corrente già possiede lo stesso lock, che è stato acquisito sul rigo 1.

Alcuni tra voi che hanno implementato il punto 3 si sono resi conto che getThreadId() poteva essere utilissimo. Tuttavia l'errore più comune è stato l'aver ritenuto che si chiedesse di IMPEDIRE le acquire consecutive. Si chiedeva invece di gestire le acquire consecutive in maniera tale da fare diventare il lock completamente rientrante.