Corso di Programmazione Concorrente e Parallela - Serie 2

Maura Clerici

01.03.2019

1 Esercizio 1

Il problema di race-condition è causato dall'oggetto condiviso da tutti i threads, Autostrada. In modo particolare durante le operazioni eseguite in run() da tutti i threads, in quanto essi hanno accesso in lettura e modifica alle variabili:

```
autostrada.entrate++;
autostrada.uscite++;
autostrada.pedaggi += pedaggioTratta;
```

La prima soluzione prevede l'utilizzo di un synchronized block; le variabili a cui i thread hanno accesso in lettura e scrittura vengono raccolte in un blocco sincronizzato che "protegge" la risorsa autostrada:

```
synchronized (autostrada) {
    autostrada.entrate++;
    int pedaggioTratta = percorriAutostrada();
    autostrada.uscite++;
    autostrada.pedaggi += pedaggioTratta;
    pedaggiPagati += pedaggioTratta;
}
```

Nota: non è necessario "proteggere" pedaggiPagati in quanto si tratta di una variabile di ogni singolo thread (automobilista).

La seconda soluzione prevede l'utilizzo di metodi synchronized. È stato creato un metodo travel() all'interno della classe Autostrada che prende come parametro un pedaggio:

```
public synchronized void travel(int pedaggio) {
    this.entrate++;
    this.uscite++;
    this.pedaggi += pedaggio;
}
```

Il metodo viene richiamato nel run(). Un'altra versione sarebbe stata quella di creare tre metodi invece di uno, quindi un metodo per ogni singola operazione:

```
public synchronized void addEntrata() {
    this.entrate++;
}

public synchronized void addUscita() {
    this.uscite++;
}

public synchronized void addPedaggio(int pedaggio) {
    this.pedaggi += pedaggio;
}
```

La terza soluzione prevede l'utilizzo di explicit locks. È stato definito un lock all'interno della risorsa condivisa Autostrada. In seguito, nell'operazione di run(), il lock bloccato e sbloccato ad ogni azione eseguita da tutti i threads:

```
//Lock per incremento entrata
autostrada.lock.lock();
try {
    autostrada.entrate++;
} finally {
    autostrada.lock.unlock();
}

int pedaggioTratta = percorriAutostrada();

//Lock per incremento uscita
autostrada.lock.lock();
try {
    autostrada.uscite++;
} finally {
    autostrada.lock.unlock();
}

//Lock per somma pedaggio
autostrada.lock.lock();
try {
    autostrada.pedaggi += pedaggioTratta;
} finally {
    autostrada.lock.unlock();
}
```

2 Esercizio 2

Il problema di race-condition si crea sulla risorsa condivisa BagnoPubblico. I thread si riferiscono agli utenti che utilizzano e occupano i bagni pubblici condivisi. Nel metodo run() ci sono due variabili che subiscono un incremento,numUtilizzi e numOccupato, ma non è necessario "proteggerle" in quanto fanno parte di ogni singolo thread (utente). Al contrario occorre "proteggere" il metodo occupa(), in quanto questo è definito nella classe BagnoPubblico e all'interno vengono modificate delle variabili condivise.

La prima soluzione prevede l'utilizzo di un synchronized block, inserito all'interno del metodo interessato:

```
public boolean occupa() {
    // Verifica disponibilita bagni liberi!
    synchronized (this) {
        if (occupati < disponibili) {
            // Bagno libero! Occupa
            occupati++;
            totUtilizzi++;
        } else {
            // Tutti i bagni sono occupati!
            totOccupati++;
            return false;
        }

        // Utilizza il bagno
        utilizzaBagno();

        // Libera il bagno
        occupati--;
        return true;
    }
}
```

La seconda soluzione prevede l'utilizzo di un metodo synchronized. Dal momento che occorre unicamente "proteggere" il metodo occupa(), è sufficiente modificare la definizione del metodo:

```
public synchronized boolean occupa() {
...
}
```

La terza soluzione prevede l'utilizzo di explicit locks. È stato creato un lock nella classe BagnoPubblico, in quanto si tratta della risorsa condivisa. Viene eseguito il lock() dello stesso all'interno del metodo occupa(), in quanto si tratta dell'unica operazione da "proteggere". Alla fine del metodo viene

eseguito un unlock(). In questo modo, quando un utente occupa un bagno, è come se mettesse un "lucchetto" che toglie quando ha finito di occuparlo:

```
public boolean occupa() {
    lock.lock();
    try {
        // Verifica disponibilita bagni liberi!
        if (occupati < disponibili) {
            // Bagno libero! Occupa
            occupati++;
            totUtilizzi++;
        } else {
            // Tutti i bagni sono occupati!
            totOccupati++;
            return false;
        }

        // Utilizza il bagno
        utilizzaBagno();

        // Libera il bagno
        occupati--;
        return true;
    } finally {
        lock.unlock();
    }
}
```

3 Esercizio 3

Per svolgere il terzo esercizio è stata definita una classe per il Conto comune della banca (risorsa condivisa) e una classe per la definizione degli utenti utilizzatori.

La classe Conto dispone dei metodi incrementaSaldo, decrementaSaldo, getSaldo e empty, tutti protetti con degli explicit locks a parte incrementaSaldo, in quanto si tratta dell'unico metodo non utilizzato dagli utenti, bensì esclusivamente dal main al momento della creazione del conto.

Gli utenti hanno a disposizione dei metodi per effettuare le operazioni di prelievo ed ottenere il loro saldo personale complessivo. Durante le loro operazioni di prelievo, ogni utente verifica se sul conto è disponibile l'importo da prelevare desiderato: in caso positivo, esso viene prelevato, altrimenti l'utente preleva il residuo disponibile e riporta sia quanto è riuscito a prelevare che quanto avrebbe voluto prelevare.