Concurrent programming example; Thread safe collections

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 16, 9 March 2023

An exercise in concurrent programming

- A narrow North-South bridge can accommodate traffic only in one direction at a time.
- When a car arrives at the bridge
  - Cars on the bridge going in the same direction $\Rightarrow$ can cross
  - No other car on the bridge $\Rightarrow$ can cross (implicitly sets direction)
  - Cars on the bridge going in the opposite direction $\Rightarrow$ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour.
- When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more.

An example . . .

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the bridge at one time (all going in the same direction!)
- `Bridge` has a public method `public void cross(int id, boolean d, int s)`
  - `id` is identity of car
  - `d` indicates direction
    - `true` is North
    - `false` is South
  - `s` indicates time taken to cross (milliseconds)

An example . . .

```
public void cross(int id, boolean d, int s)
```

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change
    `Car 10 going South stuck at Thu Mar 9 12:42:13 IST 2023`
  - The direction changes
    `Car 10 switches bridge direction to South at Fri Feb 25 12:42:13 IST 2023`
  - A car enters the bridge
    `Car 10 going South enters bridge at Thu Mar 9 12:42:13 IST 2023`
  - A car leaves the bridge
    `Car 10 leaves at Thu Mar 9 12:42:14 IST 2023`

Analysis

- The "data" that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on bridge — `int bcount`
  - Current direction of bridge — `boolean direction`
- The method `public void cross(int id, boolean d, int s)` changes the state of the bridge
  - Concurrent execution of `cross` can cause problems ...
- ... but making `cross` a synchronized method is too restrictive
  - Only one car on the bridge at a time
  - Problem description explicitly disallows such a solution

Analysis . . .

- Break up `cross` into a sequence of actions
  - `enter` — get on the bridge
  - `travel` — drive across the bridge
  - `leave` — get off the bridge
  - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
  - `enter` : increment number of cars, perhaps change direction
  - `leave` : decrement number of cars
- Make `enter` and `leave` synchronized
- `travel` is just a means to let time elapse — use `sleep`

Code for `cross`

```
public void cross(int id, boolean d, int s){

    // Get onto the bridge (if you can!)
    enter(id,d);

    // Takes time to cross the bridge
    try{
        Thread.sleep(s);
    }
    catch(InterruptedException e){}

    // Get off the bridge
    leave(id);
}
```

Analysis . . .

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is zero, it can reset the direction and enter
- Otherwise, `wait()` for the state of the bridge to change
- In each case, print a diagnostic message

Code for enter

```
private synchronized void enter(int id, boolean d){
    Date date;

    // While there are cars going in the wrong direction
    while (d != direction && bcount > 0){

        date = new Date();
        System.out.println("Car "+id+" going "+direction_name(d)+" stuck at "+date);

        // Wait for our turn
        try{
            wait();
        }
        catch (InterruptedException e){}
    }

    ...
```

Code for enter

```
private synchronized void enter(int id, boolean d){
    ...
    while (d != direction && bcount > 0){ ... wait() ...}
    ...

    if (d != direction){  // Switch direction, if needed
        direction = d;
        date = new Date();
        System.out.println("Car "+id+" switches bridge direction
           to "+direction_name(direction)+" at "+date);
    }

    bcount++;  // Register our presence on the bridge

    date = new Date();
    System.out.println("Car "+id+" going "+direction_name(d)+" enters bridge at "+date);
}
```

Code for `leave`

Leaving the bridge is much simpler

- Decrement the car count
- `notify()` waiting cars . . . provided car count is zero

```
private synchronized void leave(int id){
    Date date = new Date();
    System.out.println("Car "+id+" leaves at "+date);

    // "Check out"
    bcount--;

    // If everyone on the bridge has checked out, notify the
    // cars waiting on the opposite side
    if (bcount == 0){
        notifyAll();
    }
}
```

Summary

- Concurrent programming can be tricky
- Need to synchronize access to shared resources
- . . . while allowing concurrency
- This bridge crossing example is a prototype for a number of real world requirements

Concurrency and collections

- Synchronize access to bank account array to ensure consistent updates
- Noninterfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance
- Can we implement collections to allow such concurrent updates in a safe manner — make them thread safe?

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment accounts[i], neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability
- Contrast with serializability in databases, where transactions (sequences of updates) appear atomic

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                    int source,
                    int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Thread safe collections

- To implement thread safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`
  - `BlockingQueue`, `ConcurrentSkipList`, ...
  - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates
- Sequences of updates (transfer from one account to another) still need to be manually synchronized to work properly

Usings thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer–Consumer system
  - Producer threads insert items into the queue
  - Consumer threads retrieve them.
- Bank account example
  - Transfer threads insert transfer instructions into shared queue
  - Update thread processes instructions from the queue, modifies bank accounts
  - Only the update thread modifies the data structure
  - No synchronization necessary
- How does a consumer thread know when to check the queue?

Blocking queues

- Blocking queues block when ...
  - ... you try to add an element when the queue is full
  - ... you try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
  - Producers write intermediate results into the queue
  - Consumers retrieve these results and make further updates
- Blocking automatically balances the workload
  - Producers wait if consumers are slow and the queue fills up
  - Consumers wait if producers are slow to provide items to process

Summary

- When updating collections, locking the entire data structure for individual updates is wasteful
- Sufficient to protect access within a local portion of the structure
  - Ensure that two updates do not overlap
  - Region to protect depends on the type of collection
  - Implement using lower level locks of suitable granularity
- Java provides built-in thread safe collections
- One of these is a blocking queue
  - Use a blocking queue to coordinate producers and consumers
  - Ensure safe access to a shared data structure without explicit synchronization