Concurrency

Franck van Breugel

March 29, 2018

1 Readers-Writers: synchronized

Modify the code below so that it uses **`synchronized`** blocks.

```
public class Database {
  private boolean writing;
  private int readers;

  /**
   * Initializes this database.
   */
  public Database() {
    this.writing = false;
    this.readers = 0;
  }

  public void read() {
    this.beginRead();
    // read
    assert !this.writing;
    this.endRead();
  }

  private synchronized void beginRead() {
    while (this.writing) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    this.readers++;
```

```
  }

  private synchronized void endRead() {
    this.readers--;
    if (this.readers == 0) {
      this.notifyAll();
    }
  }
```

2 Readers-Writers: start all at the same time

The class **`CyclicBarrier`** contains the constructor

`CyclicBarrier(int parties)`

Initializes this **`CyclicBarrier`** that will trip when the given number of **`parties`** (threads) are waiting upon it.

`public int await()`

Waits until all parties have invoked **`await`** on this barrier. Returns the arrival index of the current thread (the last thread to arrive has index zero).

Modify the code below so that all readers and writers start at the same time.

```
public class ReadersAndWriters {
  public static void main(String[] args) {
    final int READERS = 2;
    final int WRITERS = 2;

    Database database = new Database();

    Reader[] reader = new Reader[READERS];
    for (int i = 0; i < READERS; i++) {
      reader[i] = new Reader(database);
    }
    Writer[] writer = new Writer[WRITERS];
    for (int i = 0; i < WRITERS; i++) {
      writer[i] = new Writer(database);
    }

    for (int i = 0; i < READERS; i++) {
      reader[i].start();
    }
    for (int i = 0; i < WRITERS; i++) {
```

```
      writer[i].start();
    }
  }
}

public class Reader extends Thread {
  private Database database;

  public Reader(Database database) {
    super();
    this.database = database;
  }

  public void run() {
    this.database.read();
  }
}
```

3 Race conditions and data races

A *race condition* is a flaw that occurs when the timing or ordering of events affects a program's correctness. Generally speaking, some kind of external timing or ordering non-determinism is needed to produce a race condition.

A *data race* happens when there are two memory accesses in a program where both

- target the same location,
- are performed concurrently by two threads,
- are not reads (at least is a write),
- are not synchronization operations.

Give an example that has both a data race and a race condition.

```
public class
```

}

Give an example that has a race condition but does not have a data race.

public class

}

Give an example that has a data race but does not have a race condition.

public class

```
}
```

4 Concurrent stack

Objects of type **AtomicReference<V>** contain a value of type V that may be updated atomically.

The class contains the method

```
public final boolean compareAndSet(V expect, V update)
```

It atomically sets the value to **update** if the current value of the object == **expect**. It returns true if the update is successful, and false otherwise.

```
public class Node<T> {
  private final T data;
  private Node<T> next;

  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }
  ...
}
```

Implement a Stack by means of **AtomicReference<V>**.

```
public class Stack<T> {
```

```
}
```

The class **AtomicReferenceFieldUpdater<T,V>** contains the method

```
public static <U,W> AtomicReferenceFieldUpdater<U,W>
  newUpdater(Class<U> tclass, Class<W> vclass, String fieldName)
```

It returns an object that can be used to atomically update the field with the given **fieldName**.

The class contains the method

```
public abstract boolean compareAndSet(T object, V expect, V update)
```

It atomically sets the field of the given **object** managed by this updater to the given **update** value if the current value === **expect**.

This method is guaranteed to be atomic with respect to other calls to **compareAndSet**, but not necessarily with respect to other changes in the field.

Implement a Stack by means of **AtomicReferenceFieldUpdater<T,V>**.

```
public class Stack<T> {









}
```