Concurrency

EECS 4315

www.eecs.yorku.ca/course/4315/

Course evaluation

The course evaluation can be completed here.

Readers-writers problem

```
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++;
}
```

Synchronized blocks

```
synchronized(o) {
  ...
}
```

Before executing the block of code, the lock of the object o needs to be acquired.

Readers-writers problem

Question

Implement the `read` method using `synchronized` blocks.

Readers-writers problem

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

Starting all threads "at the same time"

The class `CyclicBarrier`, which is part of the package `java.util.concurrent`, contains the method `await` that allows a set of threads to all wait for each other.

`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).

Starting all threads “at the same time”

Question

Modify the readers-writers solution so that all readers and writers start at the same time.

Starting all threads "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();
    CyclicBarrier barrier = new CyclicBarrier(READERS + WRI

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

Starting all threads “at the same time”

```
public class Reader extends Thread {
  private Database database;
  private CyclicBarrier barrier;

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

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

Race condition and data race

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.

Race condition and data race

Many race conditions are due to data races, and many data races lead to race conditions. However, we can have race conditions without data races and data races without race conditions.

Race condition and data race

Many race conditions are due to data races, and many data races lead to race conditions. However, we can have race conditions without data races and data races without race conditions.

Question

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

Hint

We have already seen such an example earlier in the course.

Race condition and data race

```
/**
* Two threads that share an account and both do
* a deposit concurrently cause a data race and
* a race condition.
*/
public class Account {
  private double balance;

  public void deposit(double amount) {
    this.balance += amount;
  }
}
```

Race condition and data race

Many race conditions are due to data races, and many data races lead to race conditions. However, we can have race conditions without data races and data races without race conditions.

Question

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

Hint

Modify the previous example.

Race condition and data race

```
/**
* Two threads that share an account and both do
* a deposit concurrently cause a race condition
* but no data race.
*/
public class Account {
  private double balance;

  public void deposit(double amount) {
    double temp;
    synchronized (this) {
      temp = this.balance;
    }
    temp += amount;
    synchronized (this) {
      this.balance = temp;
    }
  }
```

Race condition and data race

Many race conditions are due to data races, and many data races lead to race conditions. However, we can have race conditions without data races and data races without race conditions.

Question

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

Race condition and data race

```
/**
* Multiple threads searching for an element in an
* array may cause a data race but not a race condition.
*/
public class Search {
  private int[] collection;
  private boolean found;

  public Search(int[] collection) {
    super();
    this.collection = collection;
    this.found = false;
  }

  public void find(int from, int to, int element) {
    for (int i = from; i < to && !this.found; i++) {
    if (this.collection[i] == element) {
      this.found = true;
    }
  }
```

Detecting data races with JPF

```
target=<name of the class>
classpath=<path to the folder that contains the bytecode>
listener=gov.nasa.jpf.listener.PreciseRaceDetector
```

Compare-and-swap (CAS)

The operation CAS(variable, expected, new) atomically

- loads the value of variable,
- compares that value to expected,
- assigns new to variable if the comparison succeeds, and
- returns the old value of variable.

java.util.concurrent.atomic

The Java package `java.util.concurrent.atomic` contains classes that support lock-free thread-safe programming on single variables.

AtomicReference⟨V⟩

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.

Node⟨T⟩

```
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;
  }
  ...
}
```

AtomicReference⟨V⟩

Problem

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

Stack

```
public class Stack<T> {
  private final AtomicReference<Node<T>> top;

  public Stack() {
    super();
    this.top = new AtomicReference<Node<T>>();
  }
  ...
}
```

Stack

```
public T pop() throws Exception {
  Node<T> node;
  do {
    node = this.top.get();
    if (node == null) {
      throw new Exception();
    }
  }
  while (!this.top.compareAndSet(node, node.getNext()));
  return node.getData();
}
```

Stack

```
public void push(T data) {
  Node<T> node = new Node<T>(data, null);
  do {
    node.setNext(this.top.get());
  }
  while (!this.top.compareAndSet(node.getNext(), node));
}
```

AtomicReferenceFieldUpdater⟨T,V⟩

The class 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`.

AtomicReferenceFieldUpdater⟨T,V⟩

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.

AtomicReferenceFieldUpdater⟨T,V⟩

Problem

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

Stack

```
public class Stack<T> {
  private Node<T> top;
  private static final
    AtomicReferenceFieldUpdater<Stack, Node> updater =
    AtomicReferenceFieldUpdater.newUpdater(Stack.class,
      Node.class, "top");

  public Stack() {
    this.top = null;
  }
  ...
}
```

Stack

```
public T pop() throws Exception {
  Node<T> node;
  do {
    node = this.top;
    if (node == null) {
      throw new Exception();
    }
  }
  while (!updater.compareAndSet(this, node,
    node.getNext()));
  return node.getData();
}
```

Stack

```
public void push(T data) {
  Node<T> node = new Node<T>(data, null);
  do {
    node.setNext(this.top);
  }
  while (!updater.compareAndSet(this, node.getNext(),
    node));
}
```