All the code can be found on: github.com/PatNei/PCPP-assignment-2

Group name: Spinning Jenny

Group members:

Tobias Skovbæk Brund	tbru
Patrick Wittendorff Abarzua Neira	paab
Anders Arvesen	aarv

Exercise 3.1

Exercises 3.1.1

1. Implement a class `BoundedBuffer<T>` as described above using *only* Java `Semaphore` for synchronization—i.e., Java `Lock` or intrinsic locks (`synchronized`) cannot be used.

```java
public class BoundedBuffer<T> implements BoundedBufferInteface<T> {

    private final LinkedList<T> buffer = new LinkedList<T>();
    private final Semaphore buffMutex = new Semaphore(1);
    private final Semaphore readSema;
    private final Semaphore writeSema;


    public BoundedBuffer(int bufferSize) {
        this.writeSema = new Semaphore(bufferSize, true);
        this.readSema  = new Semaphore(bufferSize, true);
        this.readSema.drainPermits();
    }
```

```java
public void insert(T elem) throws Exception {
    writeSema.acquire();

    buffMutex.acquire();
    buffer.add(elem);
    buffMutex.release();

    readSema.release();

    //writeSema.release();
}
```

```java
public T take() throws Exception {
    readSema.acquire();

    buffMutex.acquire();
    var elm = buffer.removeFirst();
    buffMutex.release();

    writeSema.release();
    return elm;
}
```

Exercise 3.1.2

2. Explain why your implementation of `BoundedBuffer<T>` is thread-safe. Hint: Recall our definition of thread-safe class, and the elements to identify/consider in analyzing thread-safe classes (see slides).

- **Class state =>** No class variables can be manipulated outside of get/set methods (take/insert)
    - The underlying list is encapsulated, and can only be interacted with through the BoundBufferInterface.
- **Class state =>** Constructor only use primitives
    - However the take() method allows for the generic T which could potentially be a reference to some complex object. But the incoming object doesn't belong to the class of state of our BoundedBuffer implementation. We can only guarantee freedom from data races on the fields of the class.
- **Escaping =>** We don't leak any shared state variables
    - All variables are private
- **Safe Publication =>** because the internal buffer field is final and
    - We have ensured visibility by making the buffer AND semaphore fields final.
    - By initializing the LinkedList object in the field of the class.
- **Immutability =>** We made the fields final, but they all point to complex objects so immutability is not guaranteed
- **Mutual exclusion =>** We have introduced a semaphore that acts as a lock/mutex around any access to the internal linkedlist.

Exercise 3.1.3

3. Is it possible to implement `BoundedBuffer<T>` using Barriers? Explain your answer.

Case 1 (Barrier size of buffer size)

- Producers and consumers would consume and produce in blocks of the buffer size,
    - Which would lead to starvation
    - If there aren't enough writes by the producers meaning that consumers would never get to consume anything.
- It breaks the specification of the boundedbuffer.

- Because instead of a thread being able to make an `insert` call and then move on, the thread will have to await `bufferSize` many other threads to reach the barrier.
- Could also cause the program to deadlock, if there are less writer threads than `bufferSize`.

Case 2 (Barrier size is the size of 1)

- The program would run sequentially and remove the benefits of the barrier and concurrency as a whole.

Exercises 3.1.4

Challenging

4. One of the two constructors to Semaphore has an extra parameter named `fair`. Explain what it does, and explain if it matters in this example. If it does not matter in this example, find an example where it does matter.

- SKIPPED -

Exercise 3.2

Exercise 3.2.1

Exercise 3.2 Consider a `Person` class with attributes: id (`long`), name (`String`), zip (`int`) and address (`String`). The `Person` class has the following functionality:

- It must be possible to change zip and address together.
- It is not necessary to be able to change name—but it is not forbidden.
- The id cannot be changed.
- It must be possible to get the values of all fields.
- There must be a constructor for `Person` that takes no parameters. When calling this constructor, each new instance of Person gets an id one higher than the previously created person. In case the constructor is used to create the first instance of `Person`, then the id for that object is set to 0.
- There must be a constructor for `Person` that takes as parameter the initial id value from which future ids are generated. In case the constructor is used to create the first instance of `Person`, the initial parameter must be used. For subsequent instances, the parameter must be ignored and the value of the previously created person must be used (as stated in the previous requirement).

1. Implement a thread-safe version of `Person` using Java intrinsic locks (`synchronized`). Hint: The `Person` class may include more attributes than those stated above; including `static` attributes.

PCPP-assignment-2 [SSH: devserver] - Person.java

```java
public class Person {
    private static long nextID = 0;

    private final long id;
    private String name, address;
    private int zip;

    public Person() {
        synchronized(Person.class) {
            id = nextID++;
        }
    }

    public Person(int initialID) {
        synchronized(Person.class) {
            if(nextID == 0) {
                System.out.println("Person: Setting initial nextID to: (" + initialID + ")");
                nextID = initialID;
            }
            id = nextID++;
        }
    }

    public synchronized String getName() { return name; }
    public synchronized int getZip() { return zip; }
    public synchronized String getAddress() { return address; }
    public synchronized long getID() { return id; }

    public synchronized void setZipAddress(int zip, String address) {
        this.address = address;
        this.zip = zip;
    }

    public synchronized void setName(String name) { this.name = name; }

    public synchronized String toString() {
        return "{ id = " + this.getID()
                + ", name = " + this.getName()
                + ", address = (" + this.getZip() + ", " + this.getAddress() + ")"
                + "}";
    }
}
```

Exercise 3.2.2

2. Explain why your implementation of the `Person` constructor is thread-safe, and why subsequent accesses to a created object will never refer to partially created objects.

Recall definition of **thread-safe:** *"A class is said to be thread-safe if and only if no concurrent execution of method calls or field accesses (read/write) result in data races on the fields of the class".*

Our *Person* class is *thread-safe* because:

- Class state:
  - The constructor uses synchronization (in the form of an intrinsic lock on the class object) to have the read-modify-write to the static field `nextID` be an **atomic** operation. The person id counter is then free from data races, because all read/writes to the static field are related by happens-before
  - Important to note that both `setZipAddress` and `setName` take in reference types (namely `String`) as parameters, and directly use those references to update class state. This is okay here, because `String` objects in Java are **immutable**, so even if another thread has access to the string, they cannot change it.
- Escaping:
  - The *Person* class makes use of encapsulation to hide its class state, and make sure none of it escapes.
  - `getZip` and `getID` both return value types that are passed around as copies. And `getName` + `getAddress` pass references to immutable `String` objects. => No class state escapes.
- Publication
  - The only field initialized by the *Person* constructor is the `id` field. The `id` field is marked *final,* which ensures a *happens-before* any other action that happens after the constructor finishes.
    - See the definition from [slide 31, lecture 3]: *"The initialization of final and static fields happens-before any other actions of a program (after the constructor has finished its execution)"*
  - Which is also why subsequent reads will never see a partially created object.
- Immutability: N/A.
- Mutual Exclusion
  - The shared mutable state for the *Person* class is the shared counter `nextID`. It has been put behind the intrinsic lock of the class object to avoid data races when constructing *Person* classes from multiple threads.

Why will subsequent reads never refer to a partially created object?

- Because the only instance field touched by the constructor is the `id`, and that is marked final, which ensures a *happens-before* with any action after the constructor.
- The remaining fields of *Person* are set once the object has already been created, and they are all synchronized using the intrinsic lock of the instance.

Exercise 3.2.3

3. Implement a main thread that starting several threads the create and use instances of the `Person` class.

It certainly uses them, but I don't know if it is representative of real "use". But at least visibility and the happens-before can kind of be seen in the first case where the reader got a hold of the reference to the person before the producer tried to print the *Person* object.

```
Thread[0] created: { id = 0, name = Reader here, address = (24, Strandvej)}
Thread[2] read:    { id = 0, name = Reader here, address = (24, Strandvej)}
Thread[0] created: { id = 2, name = null, address = (0, null)}
Thread[1] created: { id = 1, name = Reader here, address = (0, null)}
Thread[0] created: { id = 3, name = null, address = (0, null)}
Thread[3] read:    { id = 1, name = Reader here, address = (0, null)}
Thread[0] created: { id = 5, name = null, address = (0, null)}
Thread[1] created: { id = 4, name = null, address = (0, null)}
Thread[0] created: { id = 6, name = null, address = (0, null)}
Thread[1] created: { id = 7, name = null, address = (28, Strandvej)}
Thread[1] created: { id = 8, name = null, address = (0, null)}
Thread[1] created: { id = 9, name = null, address = (0, null)}
Thread[2] read:    { id = 2, name = Reader here, address = (0, null)}
Thread[3] read:    { id = 3, name = Reader here, address = (0, null)}
Thread[2] read:    { id = 4, name = Reader here, address = (0, null)}
Thread[3] read:    { id = 5, name = Reader here, address = (0, null)}
Thread[2] read:    { id = 6, name = Reader here, address = (0, null)}
Thread[3] read:    { id = 7, name = Reader here, address = (28, Strandvej)}
Thread[2] read:    { id = 8, name = Reader here, address = (0, null)}
Thread[3] read:    { id = 9, name = Reader here, address = (0, null)}
Main done - terminating
```

Exercise 3.2.4

4. Assuming that you did not find any errors when running 3. Is your experiment in 3 sufficient to prove that your implementation is thread-safe?
   - No, our experiment by itself is not enough. There are likely to be interleavings/timings/etc. that our experiment will have been unable to reproduce. But (as with testing) it can increase our confidence that the code works as we would like/specification says.

Exercise 3.3

Exercise 3.3.1

1. Implement a Semaphore thread-safe class using Java `Lock`. Use the description of semaphore provided in the slides.

- SKIPPED -

Exercise 3.3.2

2. Implement a (non-cyclic) Barrier thread-safe class using your implementation of Semaphore above. Use the description of Barrier provided in the slides.

- SKIPPED -