ECE 612 HW #1

Ali Malik

January 31st, 2023

Descriptions

1. **Dining Philosophers:** So let's say there are five philosophers. These five philosophers only do two thing: eat and think. The philosophers are all seated around a large round table. In the middle of the table is food, for this example we'll say its cake. To eat the cake a philosopher needs to grab the utensils to both his left and right. There are a few restrictions:
   - The philosopher can only eat if both utensils are in possession.
   - The philosopher can either eat or think, not both.
   - The philosopher needs the two utensils to their left and right.
2. **Producer Consumer:** In this problem, there is exactly one producer, who produces something, and exactly one consumer, who consumes what's being produced by the producer. The Producer and the Consumer both share a buffer of a fixed size. So a sample flow would be, the producer produces a single unit of item and places it in the buffer. The consumer takes a single unit of item from the buffer. There are a few issues that arise in this classic problem:
   - The Producer can only produce an item if there is space available in the shared fixed-size buffer.
   - The Consumer can only consume an item if there is an item present in the shared fixed-size buffer.
   - Access to mutate the shared fixed-size buffer is mutually exclusive, only one entity may access the buffer at a time.
3. **Reader Writer:** The Reader/Writer problem is a problem that concerns the contention of use of an object (file or database). In this situation we have readers, parties that would only like to read the object and make no changes, and writers, parties that would like to read or possibly write to the object and make changes. Some concerns regarding this problem:
   - What if there are more than one readers? Do we allow multiple readers?
   - What if there is one reader and one writer? Who gets precedence? Is a reader allowed to read while a writer is writing to the object?
   - What if there are two writers? Are multiple writers allowed to write to the object simultaneously?

Solution

Producer Consumer

Data.java

```java
package com.amalik18.Homework1.ProducerConsumer;

public class Data {
    private String data;

    public Data(String data) {
        this.data = data;
    }

    public String getData() {
        return data;
    }
}
```

Producer.java

```java
package com.amalik18.Homework1.ProducerConsumer;

import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable {

    private final BlockingQueue<Data> queue;

    public Producer(BlockingQueue<Data> queue) {
        this.queue = queue;
    }

    /**
     * When an object implementing interface {@code Runnable} is used
     * to create a thread, starting the thread causes the object's
     * {@code run} method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method {@code run} is that it may
     * take any action whatsoever.
     *
     * @see Thread#run()
     */
    @Override
    public void run() {
        // produce message
        for (int i=0; i < 100; i++) {
            Data message = new Data("" + i);
            try {
                Thread.sleep(i);
                queue.put(message);
                System.out.println("Produce " + message.getData());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Adding final exit message
        Data message = new Data("Exit");
        try {
            queue.put(message);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

Consumer.java

```java
package com.amalik18.Homework1.ProducerConsumer;

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable {
    private final BlockingQueue<Data> queue;

    public Consumer(BlockingQueue<Data> queue) {
        this.queue = queue;
    }

    /**
     * When an object implementing interface {@code Runnable} is used
     * to create a thread, starting the thread causes the object's
     * {@code run} method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method {@code run} is that it may
     * take any action whatsoever.
     *
     * @see Thread#run()
     */
    @Override
    public void run() {
        try {
            Data message;
            while((message = queue.take()).getData() != "Exit") {
                Thread.sleep(10);
                System.out.println("Consumed: " + message.getData());
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

PCService.java

```java
package com.amalik18.Homework1.ProducerConsumer;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class PCService {
    public static void main(String[] args) {
        // Create Queue of size 15
        BlockingQueue<Data> sharedBuffer = new ArrayBlockingQueue<>(15);

        Producer producer = new Producer(sharedBuffer);
        Consumer consumer = new Consumer(sharedBuffer);

        // Start producer
        new Thread(producer).start();

        // start consumer
        new Thread(consumer).start();

        System.out.println("Producer and Consumer have started.....");
    }
}
```