Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 05 REPORT

AHMED SEMİH ÖZMEKİK
171044039

1 INTRODUCTION

1.1 Problem Definition

We have a digital image and we want to insert it's every pixels into queues and then extract them according to the different priority schemes. In the explanation of the problem, three different priority schemes are defined. And if we examine that; we have, lots of data to process and different processes required on that data.
As it actually requires, our problem is being more specific with the requirement of four different threads on the data. In the absence of this problem informations, let's try to define and analysis our problem a little bit more.

We have raw image and we are expected to represent this image in two dimensional array being formed with pixels. Hence, we will have at least something to go on with. But the real problem starts just here. We need to insert this data to queues. And what distinguish the queues among them is the comparison method. We have three comparison method defined: Lexicographical comparison (LEX), Euclidean norm based comparison (EUC), Bitmix comparison (BMX). According to that we should construct 3 queues for each comparison method. Insertion and removal operations on each queue must occur concurrently. This makes our problem a lot like producer-consumer problem; yet we do not only have one consumer but we have three of them. Concurrency and consequences of producer-consumer creates the core of our problem.

For example, since some threads will share some data, synchronization of shared content's problem arises here. Or, if the queue is empty for some consumer thread, it must not check if the queue has item with brute force; rather instead the consumer thread must wait without keeping the CPU busy. As we did state, those are the problems of concurrency and the consequences of producer-consumer problem that we shall need to deal with.

1.2 System Requirements

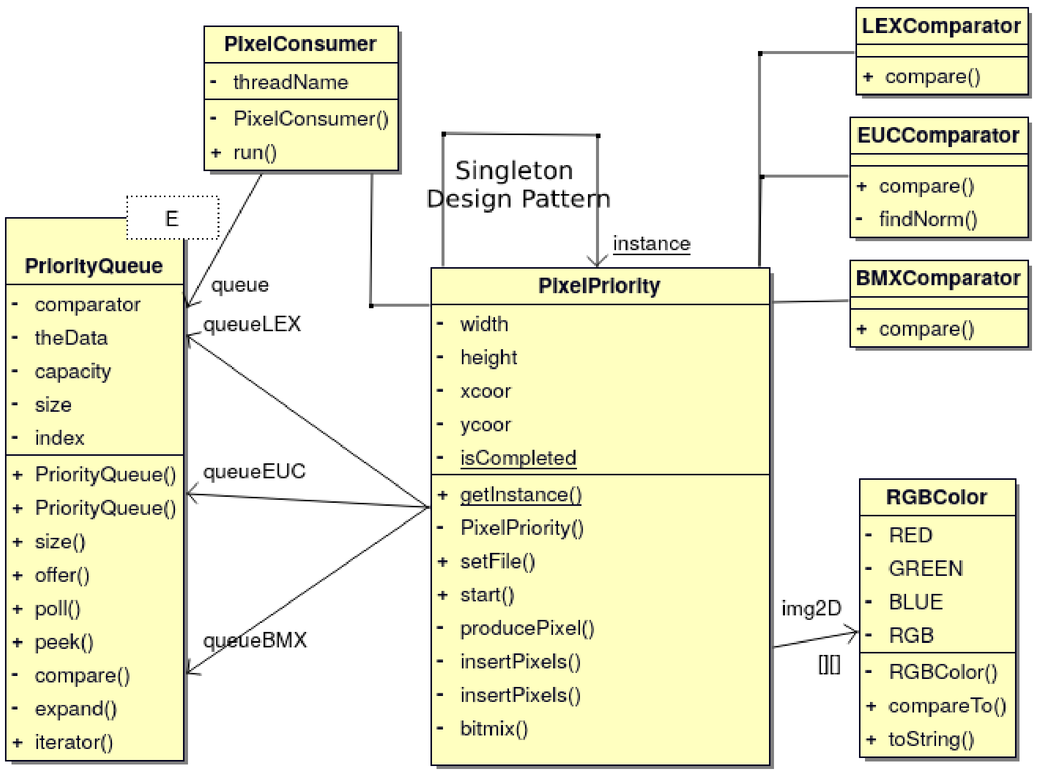
First of all, we need to get something useful from the raw image. Since we are going to represent each pixel as RGB vector, to be able represent the two dimensional array of pixels, we require a RGB vector class of our own to prevent overhead and a simple framework to get the pixels from raw image. After satisfying this requirements, we will come across to bigger problems.

Primarily, we need a Priority Queue implementation which should appropriate for different priority schemes. In implementation wise, we require a Strategy Design Pattern, which will makes our Priority Queue, comparison algorithm generic, just so we can create different queues for different priority schemes. To be more specific, we need to use “Comparator” interface to satisfy this conditions.

Hence, we can say our queues are ready to operations. After that, as our problem states, we require four different threads for four different operations. Thread-1, just after inserting hundreds of pixels to each queue, must create and start three other threads. Those 3 other threads, will have the same job which will be the removal operation from their own queues. Thread-2 will remove from queue having the scheme of lexicographical comparison; Thread-3 is the same job with the scheme of Euclidean norm based comparison and for the last Thread-4 with Bitmix comparison scheme. After satisfying this problem, we require some solution for producer-consumer problem and for the concurrency with shared data issues. We will talk about this requirements more as we mention our problem solution approach.

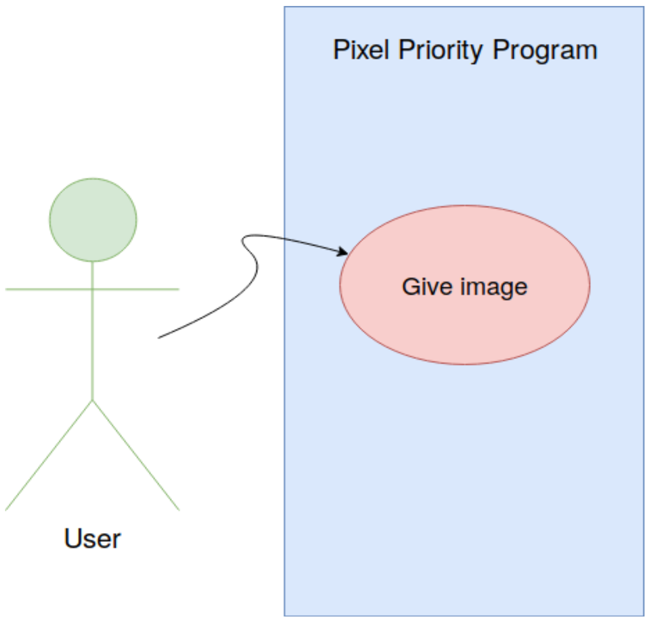
2 METHOD

2.1 Class Diagrams



Note: Usage of Singleton Design Pattern is explained in detail in problem solution approach. Straight Lines shows that only the existence of association.

2.2 UseCase Diagrams



2.3 Problem Solution Approach

Primarly, to get to pixels from raw image, I have used the “BufferedImage” class inside the “java.awt.image” package. In the implementation of the Max Priority Queue, I have used the Strategy Design Pattern with the usage of “Comparator” object. I have designed the “PixelPriority” class as Controller. All the inner details of our concrete program have been handled in the “PixelPriority” class. With the creation of three different “Comparator” class, I have satisfied, the three differrent queues for different priority scheme condition. I have used Singleton Design Pattern in “PixelPriority” class. Because, after the execution of processes (removal and insertion operations in the queues), since every queue will be discharged, our queues will become empty. Meaning that, after we are done with one operation in one context, we don’t have to create another object for totally different case in different context. We can use the same object for different calls on different methods, all we have to change is that the image itself, while queues and other fields remain the same. Of course, we can achieve this by using Singleton Design Pattern, which makes our class only have one instance at all. With this desing pattern, I believe we achieved to avoid so much overhead with many usages of “PriorityPixel”.

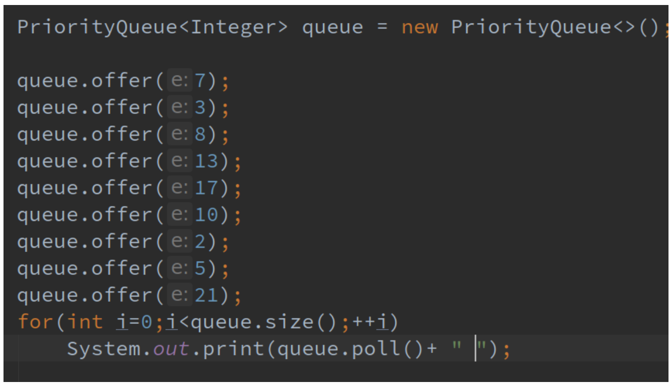
For to solve producer-consumer problem, I have used “wait()” and “notify()” methods of objects. Whenever the queue is empty for a consumer thread, it goes to state of “wait”, until the producer thread notifies to continue. And for the synchronization of shared data problem, I have used each queue as intrinsic locks. To be more specific, there is one producer thread needs to be synchronized with 3 other consumer threads which doesn’t necessarily need to be synchronized with each other. Usage of unique queues as instrict locks achieves this. And for this purpose, I have designed a “PixelConsumer” class which extends to “Thread” class. As the name suggest, this class represents the consumer threads. And for to create a thread, a name and queue must be given, just so it can use the queue as “lock” and solve the problems of concurrency.

3 RESULT

3.1 Test Cases

Case 1:

In this case, I have tested the Max Priority Queue and made sure that it is working as it supposed to do:

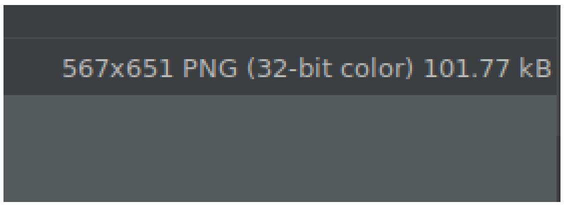


```
PriorityQueue<Integer> queue = new PriorityQueue<>();

queue.offer( e: 7);
queue.offer( e: 3);
queue.offer( e: 8);
queue.offer( e: 13);
queue.offer( e: 17);
queue.offer( e: 10);
queue.offer( e: 2);
queue.offer( e: 5);
queue.offer( e: 21);
for(int i=0;i<queue.size();++i)
    System.out.print(queue.poll()+ " ");
```

Case 2:

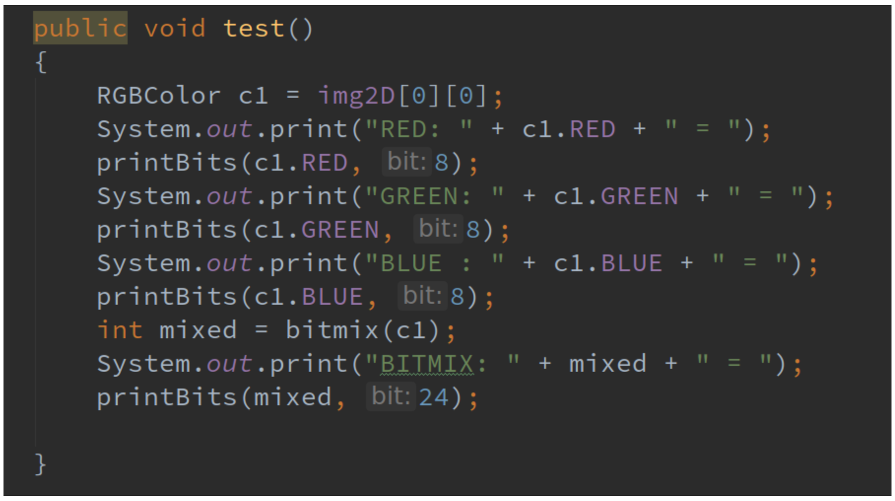
In this case, I have tested if all of the queues print out exactly WxH pixels. “example.png” has “567 x 651 = 369117” pixels.



567x651 PNG (32-bit color) 101.77 kB

Case 3:

For the last, I have tested the bitmix comparison scheme which could gone completely wrong and seemed to be the hardest one among the others.
Both Lexicographical comparison and Euclidean norm based comparison use the static compare methods of wrapper objects of primivites, which makes them well tested and ready for us. So we should only deal with testing of bitmix generator method we defined from scratch:

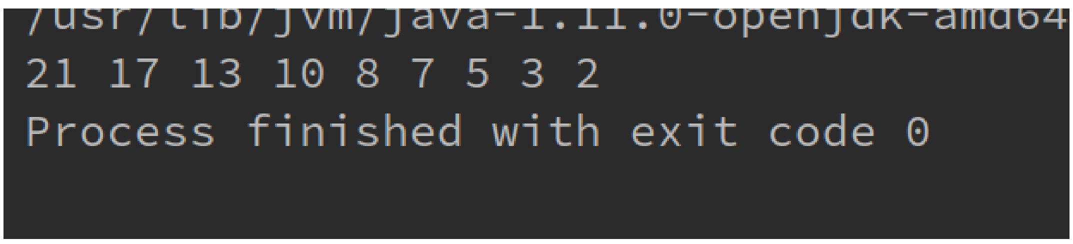


```
public void test()
{
    RGBColor c1 = img2D[0][0];
    System.out.print("RED: " + c1.RED + " = ");
    printBits(c1.RED,  bit: 8);
    System.out.print("GREEN: " + c1.GREEN + " = ");
    printBits(c1.GREEN,  bit: 8);
    System.out.print("BLUE : " + c1.BLUE + " = ");
    printBits(c1.BLUE,  bit: 8);
    int mixed = bitmix(c1);
    System.out.print("BITMIX: " + mixed + " = ");
    printBits(mixed,  bit: 24);

}
```

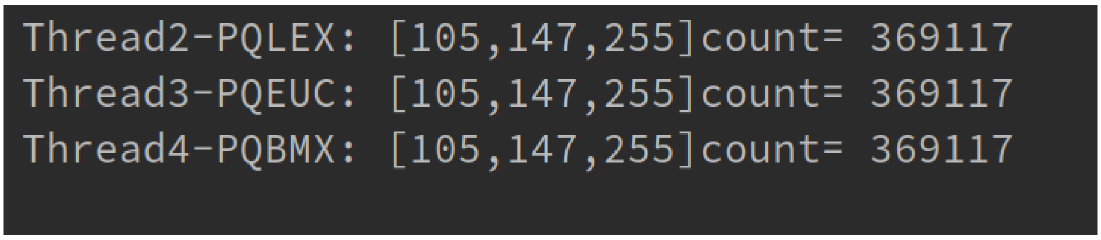
3.2 Running Results

Result Case 1:



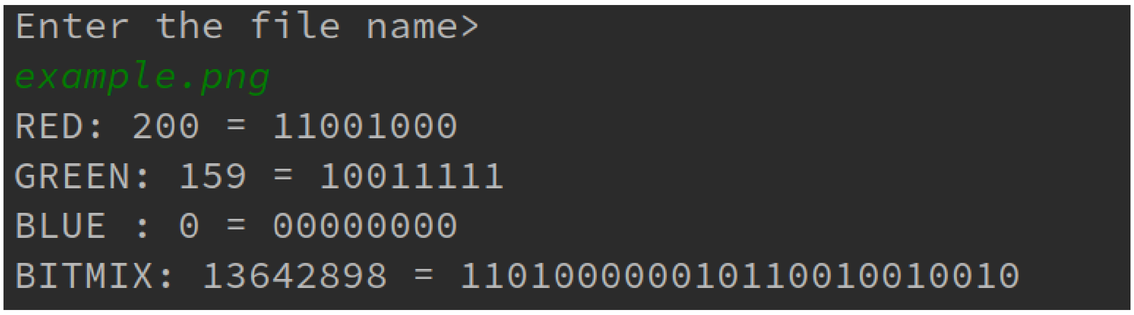
```
/usr/lib/jvm/java-1.11.0-openjdk-amd64
21 17 13 10 8 7 5 3 2
Process finished with exit code 0
```

Result Case 2:



```
Thread2-PQLEX: [105,147,255]count= 369117
Thread3-PQEUC: [105,147,255]count= 369117
Thread4-PQBMX: [105,147,255]count= 369117
```

Result Case 3:



```
Enter the file name>
example.png
RED: 200 = 11001000
GREEN: 159 = 10011111
BLUE : 0 = 00000000
BITMIX: 13642898 = 110100000010110010010010
```