Operating Systems and Concurrency

Assignment 1 (Formative)

Digital Image Processor

Deadline: 16:00, 09/02/2024

Submission instructions

Submit a single zip file contains your source code (do not submit your .class files). You should not submit anything other than .java files in one zip file.

The package declaration at the top of all of your Java files must be:

```
package com.kcl.osc.imageprocessor;
```

Introduction

For this assignment you will (a) implement a thread pool and (b) parallelise an existing application.

The application that you will parallelise is a digital image processor. You are provided with a single-threaded application that can perform various image processing functions. These functions are described below. Your task is to parallelise this application.

In addition to carefully reading this document, also read the `javadoc` comments in the provided source files.

Important note 1: you must not use existing Java 'thread pool' classes for your thread pool but must code your own from scratch. For example, you are not allowed to use `ExecutorService`.

Important note 2: the provided image processor can process only square images in which width=height.

Important note 3: the application relies on the use of some JavaFX classes so you must set up your project with JavaFX jdk in your classpath. The application does not have a GUI.

Digital images

This section describes digital images. You may be familiar with digital images on a technical level but this section is still worth reading. Note that the functionality below is **already implemented** in the provided application.

A digital image is essentially a 2D array of color values. Each color element of the 2D array is called a 'pixel' (short for 'picture element'). Each pixel has a color. This color can be represented in a number of

ways. One of the most common ways is to represent each pixel color as a combination of the colors red (R), green (G) and blue (B). This is the RGB color model. Using this model, each pixel usually has three bytes of data associated with it: one for red, one for green and one for blue. (Another byte of data called the 'transparency' is also usually associated with each pixel but we aren't going to worry about that value).

If using an integer type, we can see that one byte of data per color per pixel leads to a range of 0 to 255 different values. If the value of that color (red, green or blue) is set to zero it means that none of that color is represented in that pixel. If it is set to 255 it means that the maximum level of that color is represented in that pixel. Since each pixel has a value for each of the RGB values, this means that we can mix R,G and B together to get the colors we want.

If we set *all three* RGB values to zero we obtain the color black. If we set them all to 255 we get the color white. If we set them all to the *same value*, somewhere between 0 and 255, we get gray. If we set them to *different values* we then get other colors. There are many online tools for playing around with colors in this way and I recommend doing that if you are unfamiliar with this topic.

Image filters

Image processing involves amending the color values of the pixels in an image in some way, or creating a new image from an old one having processed the old one in some way. One way is to apply a *filter* to the entire image[1]. A filter is a 2D array of values. The filter is usually a small array, for example 3 rows by 3 columns (3x3). The values in the filter are used to modify each pixel in the source image in a specific way, to be described below. Usually, every pixel in the source image is modified by the *entire* filter[2].

The color value of each pixel to be filtered is computed as follows.

Consider a filter matrix f of dimension 3x3:

$$\begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}$$

Consider a digital image img, of dimension 8x8[3], represented as a matrix of colors:

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} & \dots & c_{07} \\ c_{10} & c_{11} & c_{12} & \dots & c_{17} \\ c_{20} & c_{21} & c_{22} & \dots & c_{27} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{70} & c_{71} & c_{72} \dots & \dots & c_{77} \end{bmatrix}$$

As an example, consider pixel c_{22}. The *new* color of pixel c_{22} (in the output image) is computed as follows:

[1] Also called *kernals* but nothing to do with operating system kernels.

[2] Although smaller areas of the source image might be filtered if, for example, blurring out only faces in the image.

[3] This is just an example. This would be an incredibly small image.

$$
\begin{aligned}
c_{22}^{new} = \quad & x_{00} \times c_{11} + x_{01} \times c_{12} + x_{02} \times c_{13} \\
& + x_{10} \times c_{21} + x_{11} \times c_{22} + x_{12} \times c_{23} \\
& + x_{20} \times c_{31} + x_{21} \times c_{32} + x_{22} \times c_{33}
\end{aligned} \tag{1}
$$

Since we are using the RGB color model, this equation applies to each of those *channels*. Thus, we perform the above computation for the red channel, the green channel *and* the blue channel.

Note that this is not a regular matrix multiplication. It is a matrix *convolution*.

Essentially, we consider the filter as being placed centrally on the pixel for which we want to compute the new value. Thus, the element [1,1] in the filter is positioned 'over' the pixel to be computed.

This computation is applied to all of the pixels in the image. This means that the filter is 'moved' across the image from the top-left corner, pixel by pixel.

Consider the identity filter. The identity filter leaves the source image unchanged. The 3x3 identity filter is shown below:

$$
\begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{bmatrix}
$$

If you consider this filter, and the equation for the computation of the new pixel value shown above, you will see that this matrix will leave the color of the pixel unchanged. Only the element [1,1] will contribute to the result (since all other terms will be zero), and element [1,1] is positioned over the original pixel value. The identity filter is very useful to check that your computations of the pixel values are working as expected.

Edge cases

How do we compute the value of pixel [0,0]? If we position the filter over this pixel, it means that parts of the filter will **not** be over the image. To compute the value of pixel [0,0] we need to place the filter element [1,1] over that pixel. So, which pixel would filter element [0,0] be positioned over? It wouldn't. It would be outside the range of the image.

This is literally an 'edge case'. What should we do? Well, you can read up about the various techniques that are used to overcome this issue, but the solution used for the application you are working on is a very simple one. In this implementation, the `Color` pixel array gets augmented with a single-pixel border containing only a grey color. This means that the filter can be applied to pixel [0,0] (in the original image) without worrying that the filter will 'fall off' the edge of the image, because there is now a border there. This does also mean that the very edge of the output image might look somewhat strange if you zoom in on it (because those pixel values have been computed using the false border that was added).

The `Color` class

This application makes extensive use of the `Color` class from JavaFX. Although we described the RGB values above using the range [0,255], the `Color` class actually uses `double` values in the range [0,1]. These

are exactly equivalent. The value 0 maps to 0 and 1 maps to 255.

To create a `Color` value you must compute the red, green and blue components. Note that this means that equation (1) must be used three times: once for each component to create a new red, green *and* blue value.

To create a new `Color` value from three `double` values called `red`, `green` and `blue` you do the following:

```
Color color = new Color(red,green,blue,1.0);
```

Note that the final parameter is the transparency value and you should always set that to 1.0.

One other thing to consider: when you compute the new red, green and blue values that you want for a pixel, you must ensure that the values are still in the range [0,1] otherwise an exception will be thrown. This is dealt with in the provided application by the `clampRGB()` method.

The provided application

As stated above, the image processor is fully implemented for you. You can run the provided application on your own **square** images. Some examples are available on Keats. To change the filtering operation you can change the value of the constant `filter` in the `ImageProcessorApplicationST` class to one of the following values:

{IDENTITY, BLUR, SHARPEN, EMBOSS, EDGE, GRAY}

Most of these are self-explanatory but "EMBOSS' will create the impression that parts of the image are 'raised' and "EDGE" works like an edge detector.

"GRAY", however, works a bit differently. That option does not actually apply a filter but converts the image to greyscale. This is a far simpler operation than filtering and works on a per-pixel basis, simply averaging the color values of each pixel. You can use this same operation but with different weightings to create a sepia image. You could add that operation to the application, if you wish, but that is not a part of this assignment.

The provided application consists of two public classes. `ImageProcessorApplicationST`[4] is the actual application that finds the images and then runs the `ImageProcessorST`s on them, thus `ImageProcessorST` is the actual image processor.

Make sure you read the `javadoc` comments in the two source files.

The Exercises

For both exercises you should create new source files. Create an application class called `ImageProcessorApplicationMT` and an image processor class called `ImageProcessorMT`. These can simply copies of the original classes at first, if you prefer to work that way.

[4] ST = single threaded.

Exercise 1: The Thread Pool [50% of the marks]

For this exercise you should create your own implementation of a *thread pool*. In preparation for this exercise, you should change the `ImageProcessorMT` class so that it implements the `Runnable` interface. Your thread pool will manage threads that run your new `ImageProcessorMT`s. The thread pool class itself should also implement `Runnable`.

The thread pool should offer a constructor that receives the *size* of the thread pool as a parameter.

The thread pool has relatively simple functionality. It should be possible to **submit** a task (i.e. an `ImageProcessorMT`) to the thread pool. This does not mean that the task will actually run yet, just that the thread pool is aware of it and keeps it in a *waiting list*. The number of `ImageProcessorMT`s that are added can exceed the number of threads available in the thread pool.

It should also be possible to **start** the thread pool. This means that the thread pool will then run as many `ImageProcessorMT`s as it can up to its *size*. For example, if the thread pool size is 5, but 10 `ImageProcessorMT`s have been added, it can only run 5.

When any `ImageProcessorMT` has terminated, a new one from the waiting list should be started by the thread pool (if one exists). The thread pool should continue to do this until the waiting list is empty, when it should itself terminate.

The thread pool should offer a `join()` method that causes the calling thread to wait until the thread pool has terminated.

Note that when you change `ImageProcessorMT` to implement `Runnable`, its `run()` method works the same way as it does in `ImageProcessorST`: it processes the image and then exits (terminating the thread). You might wish to add a `boolean` and a getter to `ImageProcessorMT` that signals that the run method has terminated, hence the result is available, and the thread pool can now run another thread (if the waiting list was not empty).

Exercise 2: Parallelising `ImageProcessorMT` [50% of the marks]

For this exercise you will take the existing `ImageProcessorMT` class and parallelise the filtering and greyscale operations. Remember, the `ImageProcessorMT` class itself should already now implement `Runnable` as a result of Exercise 1.

Parallelising these operations means simply 'slicing' the image data and giving each slice to a thread to work on. If there are two threads, for example, each would perform the filtering or greyscale operation on half of the image data. This means that the pixel data produced by all of the threads must be 'reassembled' at the end to create one array of pixel data as the result.

There are some practical limits to this approach. For example, if an image has dimensions 256x256 then, at most, you can set 256 threads working on the image - one row of pixels each[5]. However, this might negate the benefits of parallelism due to the overhead introduced by the parallelisation. You should parametrise the number of threads created and experiment with the effect of changing the number of threads. To get your code working, make sure at first that it works with one thread before adding more.

[5]In fact, you could have more threads working on partial rows but this would probably minimise or negate the benefits of parallelism.

Suggestion: the best approach would be to get the parallelisation working on the greyscale operation first. This is the simpler operation and will allow you to get the basic threading arrangement working. It will then be easier to understand what you need to do to get the parallel filtering operation working.

Parameters and timing

The following is not a part of the assignment but would be useful to you to check whether your parallel application is able to process the images more quickly than the single-threaded version.

As with some of the lab exercises, it would be useful to add some timing code to your application to report on how long certain operations are taking. You could add similar code to the single-threaded version so you can compare the two.

There are also two parameters (at least) to play with. One is the size of the thread pool. What effect does a smaller or larger thread pool have? Is a thread pool of size two or more better than a thread pool of size one? Also, how many slices (hence threads) seems to be 'optimal' in the image processor itself?