Anexa

Cod sursa

Main:

```
package main;

import operation.*;

import java.util.Scanner;

public class Main {
    private static Operation getOperation(String operation) throws IllegalArgumentException {
        if (operation.equalsIgnoreCase("and")) {
            return new AndOperation();
        } else if (operation.equalsIgnoreCase("or")) {
            return new OrOperation();
        } else if (operation.equalsIgnoreCase("xor")) {
            return new XorOperation();
        } else {
            throw new IllegalArgumentException("Operation must be XOR|OR|AND");
        }
    }

    public static void main(String[] args) {
        if (args.length > 2 || args.length < 1) {
            System.err.println("Invalid number of arguments");
```

```
            System.exit(1);
        }

        String firstPath = args[0];
        String secondPath = args[1];

        Scanner scanner = new Scanner(System.in);

        System.out.print("Select the operation: ");
        Operation operation = getOperation(scanner.nextLine());

        System.out.print("Select the number of threads: ");
        int threads = scanner.nextInt();

        scanner.close();

        new ImageProcessor(firstPath, secondPath, operation, threads).launchTasks();
    }
}
```

Operation:

```
package operation;

public interface Operation {
    int perform(int firstSource, int secondSource);
}
```

AndOperation:

```
public class AndOperation implements Operation {
    @Override
    public int perform(int firstSource, int secondSource) {
        return firstSource & secondSource;
    }
}
```

OrOperation:

```
package operation;

public class OrOperation implements Operation {
    @Override
    public int perform(int firstSource, int secondSource) {
        return firstSource | secondSource;
    }
}
```

XorOperation:

```
package operation;

public class XorOperation implements Operation {
    @Override
    public int perform(int firstSource, int secondSource) {
        return firstSource ^ secondSource;
    }
}
```

ProcessingTask:

```java
package operation;

import java.util.concurrent.CyclicBarrier;

public abstract class ProcessingTask implements Runnable {
    private final int start;
    private final int end;
    private final CyclicBarrier cyclicBarrier;

    public ProcessingTask(int start, int end, CyclicBarrier cyclicBarrier) {
        this.start = start;
        this.end = end;
        this.cyclicBarrier = cyclicBarrier;
    }

    public int getStart() {
        return start;
    }

    public int getEnd() {
        return end;
    }

    public CyclicBarrier getCyclicBarrier() {
        return cyclicBarrier;
    }
}
```

ApplyOperationTask:

package operation;

import java.awt.image.BufferedImage;

import java.time.Instant;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class ApplyOperationTask extends ProcessingTask {

```
    private final BufferedImage firstSource;
    private final BufferedImage secondSource;
    private final BufferedImage outputSource;

    private final Operation operation;

    public ApplyOperationTask(int start, int end, CyclicBarrier cyclicBarrier, BufferedImage
firstSource, BufferedImage secondSource, BufferedImage outputSource, Operation operation) {
        super(start, end, cyclicBarrier);
        this.firstSource = firstSource;
        this.secondSource = secondSource;
        this.outputSource = outputSource;
        this.operation = operation;
    }

    @Override
    public void run() {
        try {
            System.out.printf("[%s] is waiting - %s\n", Thread.currentThread().getName(),
Instant.now());
            getCyclicBarrier().await();
        } catch (InterruptedException | BrokenBarrierException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }

        for (int y = getStart(); y < getEnd(); y++) {
            for (int x = 0; x < outputSource.getWidth(); x++) {
                int pixel = operation.perform(firstSource.getRGB(x, y), secondSource.getRGB(x, y));
```

```
                outputSource.setRGB(x, y, pixel);
            }
        }

        try {
            getCyclicBarrier().await();
            Thread.sleep(1000);
            System.out.printf("[%s] has finished at %s\n", Thread.currentThread().getName(),
Instant.now());
        } catch (InterruptedException | BrokenBarrierException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

ImageProcessor:

```
package operation;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
```

```
public class ImageProcessor {
    private BufferedImage firstSource;
    private BufferedImage secondSource;
    private BufferedImage destination;

    private final Operation operation;
    private final int threads;

    public ImageProcessor(String firstPath, String secondPath, Operation operation, int threads) {
        this.threads = threads;
        this.operation = operation;
        try {
            firstSource = ImageIO.read(new File(firstPath));
            secondSource = ImageIO.read(new File(secondPath));

            int width = Math.min(firstSource.getWidth(), secondSource.getWidth());
            int height = Math.min(firstSource.getHeight(), secondSource.getHeight());

            destination = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
        } catch (IOException e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
    }

    public void launchTasks() {
        int size = destination.getHeight();
```

```
        ExecutorService executorService = Executors.newFixedThreadPool(threads);
        CyclicBarrier cyclicBarrier = new CyclicBarrier(threads);
        for (int i = 0; i < threads; i++) {
            int start = (int) (i * (double)size / threads);
            int end = (int) Math.min((i + 1) * (double)size / threads, size);
            executorService.submit(new ApplyOperationTask(start, end, cyclicBarrier, firstSource, secondSource, destination, operation));
        }
        executorService.shutdown();
        try {
            executorService.awaitTermination(60, TimeUnit.SECONDS);
            ImageIO.write(destination, "png", new File("output.png"));
        } catch (InterruptedException | IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```