OPERATING SYSTEMS LAB 6 REPORT

Suyash Gaurav (210010054), Vivek Gaikwad (210010059)

Part 1

In this part, we have written C++ code to read a PPM (Portable PixelMap) image file and then perform two image transformations. We also wrote the transformed image to a new file.

These transformations have been performed sequentially, one after the other. The two transformations are given below...

1.1 RGB to Grayscale

Converts each RGB pixel to grayscale using the luminosity method.

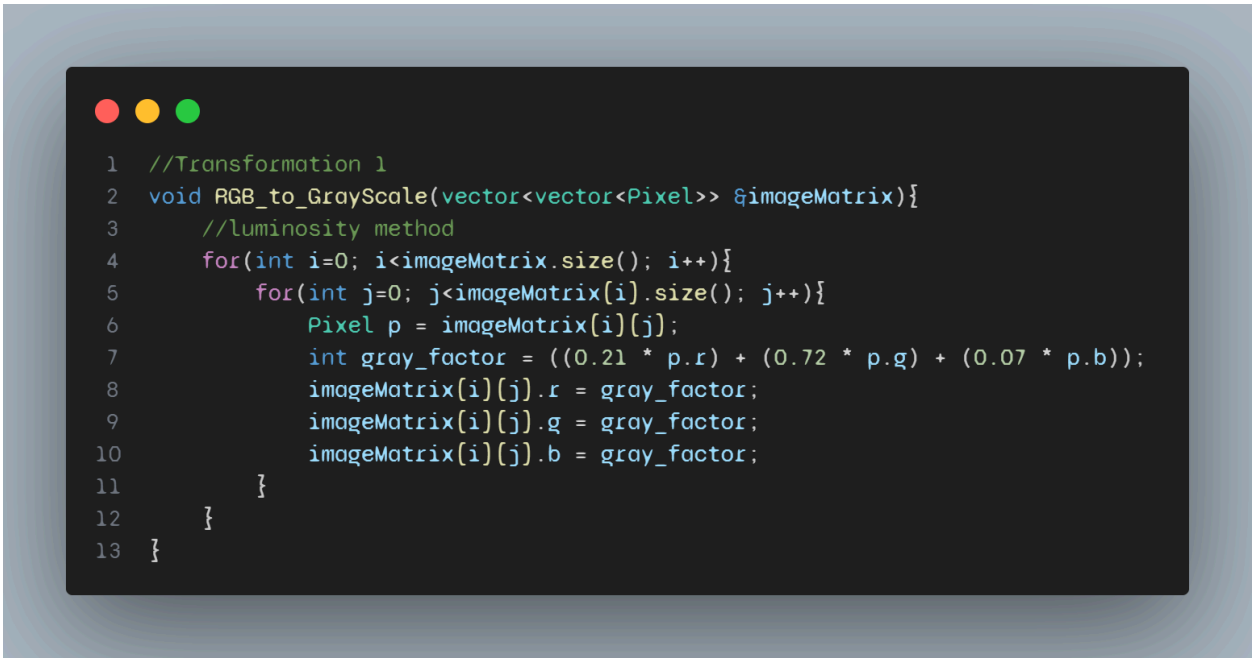
Luminosity Formula:

This method calculates the grayscale intensity (*'gray_factor'*) for a pixel using the following formula.

$$grayfactor = 0.21 \times R + 0.72 \times G + 0.07 \times B$$

R, G and B represent the pixel's red, green and blue components, respectively.

- For each pixel in the RGB image, the luminosity method calculates the gray factor.
- The resulting intensity is then assigned to each of the red, green and blue channels of pixels.
- The RGB values are effectively replaced with a single grayscale value.



```cpp
//Transformation 1
void RGB_to_GrayScale(vector<vector<Pixel>> &imageMatrix){
    //luminosity method
    for(int i=0; i<imageMatrix.size(); i++){
        for(int j=0; j<imageMatrix[i].size(); j++){
            Pixel p = imageMatrix[i][j];
            int gray_factor = ((0.21 * p.r) + (0.72 * p.g) + (0.07 * p.b));
            imageMatrix[i][j].r = gray_factor;
            imageMatrix[i][j].g = gray_factor;
            imageMatrix[i][j].b = gray_factor;
        }
    }
}
```

Code for RGB_to_GrayScale

1.2 Edge Detection

Identify and highlight edges or boundaries in an image.

Soble Operator:

It is an operator used for edge detection, and it consists of two 3*3 matrices `sobel_x` (horizontal) and `sobel_y` (vertical).

$$\text{Sobel\_x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\text{Sobel\_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- The Sobel operator is applied to the image by each pixel neighbourhood with the corresponding matrix.
- Gx and Gy are calculated for each pixel.
- Then, the magnitude of gradients ($\sqrt{(Gx\ +\ (Gy}$) is calculated to determine the strength of the edge.
- Implementation:
    - The code iterates through the image, applying the Sobel operator to calculate `Gx` and `Gy` at each pixel.
    - The magnitude is computed, and the maximum magnitude is tracked for normalisation.
    - The resultant magnitude is assigned to each channel (R, G, B) of the pixel.

```
// Transformation 2
void grayscale_to_edgeDetection(vector<vector<Pixel>> &imageMatrix){
    vector<vector<Pixel>> tempMatrix = imageMatrix;

    int sobel_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    int sobel_y[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};

    for (int i=1; i<imageMatrix.size()-1; i++) {
        for (int j=1; j<imageMatrix[i].size()-1; j++) {
            int gx = 0, gy = 0;

            for (int x=-1; x<=1; x++) {
                for (int y=-1; y<=1; y++) {
                    gx += tempMatrix[i+x][j+y].r * sobel_x[x+1][y+1];
                    gy += tempMatrix[i+x][j+y].r * sobel_y[x+1][y+1];
                }
            }

            int resultant = sqrt(gx * gx + gy * gy);
            resultant = min(255, resultant);
            imageMatrix[i][j].r = resultant;
            imageMatrix[i][j].g = resultant;
            imageMatrix[i][j].b = resultant;
        }
    }
}
```

Image Transformation:

Original

After grayscale

Final: grayscale followed by edge detection

Part 2

T1 and T2 are performed by two different threads of the same process. They communicate through the process' address space itself.

a. Synchronization using atomic operations
b. Synchronisation using semaphores

2.1a

Atomic operations controls the synchronisation between the two threads.

Here, we have used `atomic_flag_test_and_set` and `atomic_flag_clear` to make the stamps performed by transformation seem atomic.

The `imageMatrix` is a shared data structure modified by both threads concurrently.

The `while (atomic_flag_test_and_set(&flag))` construct is used to enter a critical section, ensuring that only one thread can execute the RGB to Grayscale conversion at a time.

The `atomic_flag flag` is a basic mutex, preventing simultaneous access to the critical section and avoiding data races.

Using `t1_finished` (an atomic variable) in the second transformation ensures that it waits for the completion of the first transformation before proceeding. This helps maintain the correct order of execution.

2.1b

Semaphores are used to control the synchronisation between the two threads.

Two semaphores are used `sem_t s` and `sem_t T1_done`.
`Sem_t s` is initialised to 1, providing a simple lock/unlock mechanism.
`Sem_t T1_done` is initialised to 0.

The `RGB_to_GrayScale` function is protected by the `semaphore s`, ensuring that only one thread at a time can enter the critical section. After completing the RGB to Grayscale conversion, it signals the completion using `sem_post(&T1_done)` and releases the semaphore.

The `grayscale_to_edgeDetection` function waits for the completion signal from `RGB_to_GrayScale` using `sem_wait(&T1_done)`. It then enters the critical section protected by `s`. After completing the edge detection, it releases the semaphore.

Semaphores provide a more structured synchronisation approach than atomic flags, offering features like waiting and signalling.

2.2

T1 and T2 are performed by two processes communicating via shared memory.
Synchronisation using semaphores. (Single source file)

We have created a shared memory segment using the `shmget` function. It stores the pixel values in a 2D vector called `imageMatrix`.

Since we use shared memory to store the image's pixel values, the two child processes can access the pixel values simultaneously, improving the program's performance.

Also, we are using semaphores to synchronise the access to the shared memory. This ensures that only one process can access the shared memory simultaneously, preventing race conditions.

2.3

T1 and T2 are performed by two processes communicating via pipes. (Single source file)

It utilises fork() to create child processes, where each child process handles one of the transformations. The parent process waits for the child processes to complete before writing the final result to an output image file.

based on the process ID (pid), the parent or child process executes a specific transformation function. The parent process waits for the child process to complete using the wait system call.

Data is written to and read from the pipe to communicate between parent and child processes.

The pipe system call creates a unidirectional communication channel between processes. The file descriptors `fd[0]` and `fd[1]` represent the read and write ends of the pipe, respectively.

Result and Discussion

Image size	Part1(Sequential) (in msec)	Part2.1a (Threads-atomic operations) (in msec)	Part2.1b (Threads–Semaphores) (in msec)	Part 2.2 (Process–Shared memory) (in msec)	Part 2.3 (Process–Pipe) (in msec)
~73mb	16329	16289	16125	30238	20152
~25mb	5361	5521	5482	10469	6404
~8mb	1777	1745	1728	3168	2009
~5mb	1182	1410	1240	2320	1430

- The execution times consistently increase with larger image sizes, indicating a direct correlation between image size and processing time.
- Overall, Threads-semaphores in Part2.1_b take the lowest time in more prominent image size, which means that in other cases, time was compensated by synchronisation and overhead of communication.
- Using processes and pipes for communication introduces inter-process communication overhead. The timings are generally higher compared to thread-based approaches and shared memory. Communication between processes using pipes adds latency, impacting overall performance.

- High shared memory usage due to delays caused by threads or processes waiting to access shared data structures. It could take longer for a process to attach or remove shared memory areas. Overhead is introduced by synchronisation techniques, such as locks and semaphores.

- Comparison
    - Sequential: No parallelism.
    - Threads-Atomic: Faster than sequential but atomic overhead.
    - Threads-Semaphores: Good balance of speed and safety.
    - Process-Shared Memory: Slowest, inter-process overhead.
    - Process-Pipe: Slower due to significant overhead.

Ease/Difficulty of Implementation

- Sequential Approach:
    - Ease of Implementation: Simple to implement since it follows a straightforward approach.
    - Debugging Difficulty: Debugging was relatively straightforward, as issues can be traced sequentially. However, it lacks parallelism, so performance improvements are limited.

- Threads with Atomic Operations:
    - Ease of Implementation: Requires creating two threads within the same process. Not too complex.
    - Debugging Difficulty: Debugging was challenging due to potential race conditions. Also, Ensuring atomicity requires careful handling.

- Threads with Semaphores:
    - Ease of Implementation: Similar to atomic operations but uses semaphores for synchronisation.
    - Debugging Difficulty: Easier than atomic operations. Semaphores provide better control over thread execution. Debugging focuses on semaphore usage.

- Processes with Shared Memory:
    - Ease of Implementation: Involves separate processes communicating via shared memory.
    - Debugging Difficulty: Involved a lot of segmentation faults due to not detaching shared memory. Improper pointer use, race conditions, and memory leaks in shared memory lead to segmentation faults.

- Processes with Pipes communication:
    - Ease of Implementation: Separate processes are required to communicate via pipes.

- Debugging was challenging due to the significant overhead in pipe communication. Pipes have limited capacity, writing by the child and reading by the parent process, leading to frequent context switches between the processes.

Input Image and its transformation: