CS 314: Operating Systems Laboratory
Lab 5
Group: 18

Nampally Pranav 190010026
T Satwik 190030043

---

1 Image Transformations:

1.1 Color Inversion (T1):

Each pixel's color is defined by the r, g, b values that can each go from 0-255. In color inversion the algorithm will change each color to its complementary colors, and it is achieved by setting each of the r, g, b values to 255-r, 255-g, 255-b respectively. The following is the code that achieves this inversion.

```
int i,j;
for(i=0;i<height;i++)
{
    vector<rgbpix> row;
    for(j=0;j<width;j++)
    {
        int r,g,b;
        struct rgbpix pix;
        r=input_data[i][j].r;
        g=input_data[i][j].g;
        b=input_data[i][j].b;
        pix.r=max-r;
        pix.g=max-g;
        pix.b=max-b;
        row.push_back(pix);
    }
    output_data.push_back(row);
}
```

1.2 Grayscale (T2):

This transformation converts the image to black and white. The pixels, that have same equal r, g and b values, have gray, white and black shades. Hence to convert the image to grayscale, we can simply take average of r, g and b values and assign this average to each. The following is the code that achieves this transformation.

```
int i,j;
for(i=0;i<height;i++)
{
    vector<rgbpix> row;
    for(j=0;j<width;j++)
    {
        int r,g,b;
        struct rgbpix pix;
        r=input_data[i][j].r;
        g=input_data[i][j].g;
        b=input_data[i][j].b;
        int avg=(r+g+b)/3;
        pix.r=avg;
        pix.g=avg;
        pix.b=avg;
        row.push_back(pix);
    }
    output_data.push_back(row);
}
```

1.3 Sample Images:

The original image used is in Figure 1.
The image after applying inversion transformation(T1) can be seen in Figure 2
The image after applying grayscale transformation(T2) can be seen in Figure 3
The image after applying T1 and T2 one after the other can be seen in Figure 4

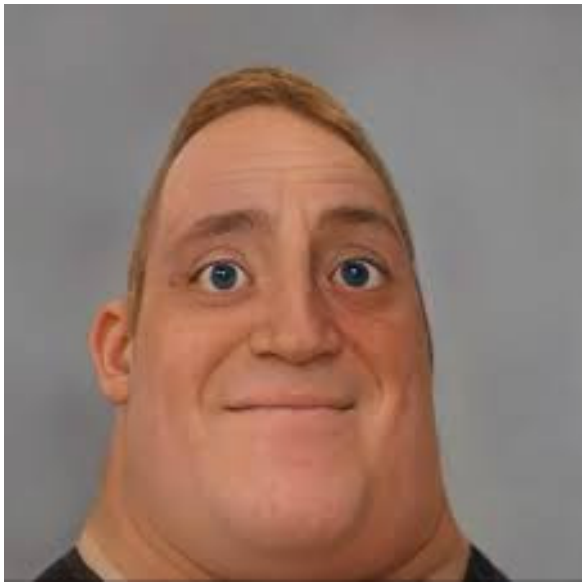

Figure 1: Original Image

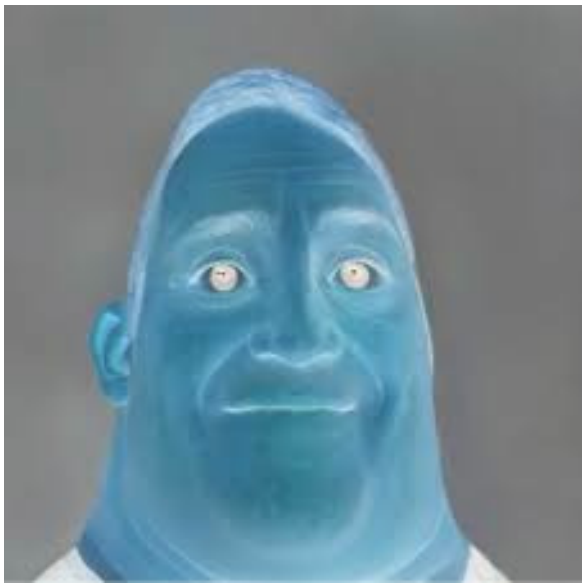

Figure 2: Inversion Transformation

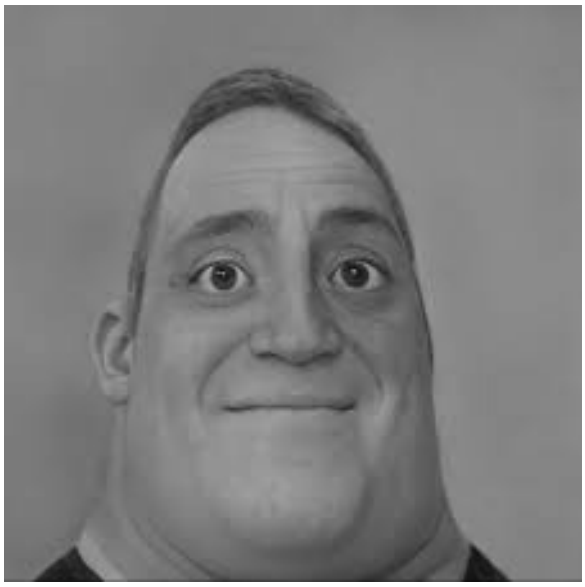

Figure 3: Grayscale Transformation

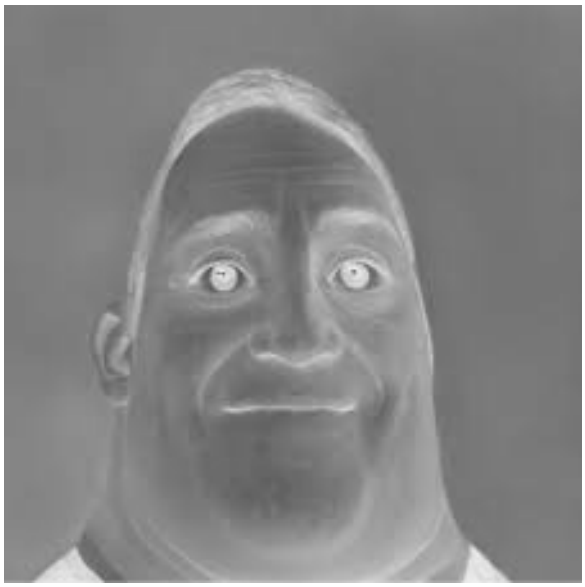

Figure 4: Final Output after applying T1 then T2

2 Proof that pixels are sent and received in correct order:

2.1 Part1:

Here we just had 2 functions that apply the transformations on the given vector, hence by calling these 2 functions one after the other, we ensure that the second transformation runs on the modified values of the first transformation.

2.2 Part2:

2.2.1 1a

The 2 transformations happen by 2 different threads, and synchronization between these threads can be achieved by making the whole transformation process atomic, by using mutex locks. The main waits(join) after creating thread1, and then creates thread 2 and hence T2 is created and run only after T1 is completely executed.

2.2.2 1b

This is similar to the above method, but the instead of mutex locks, we used semaphores that is initialized to 1, which basically acts as a lock and hence overall order doesn't change.

2.2.3 2

In this case, the child process first transforms the data and writes the whole transformed set of pixels in the shared memory, and by using semaphores between process, the parent process shall wait until all the pixels data is written into the shared memory. By using semaphores we are ensuring that the child process runs first and writes all the transformed pixels to shared memory, and parent reads

from the shared memory only after all the data is written to it by the child, and hence the overall correctness is ensured.

2.2.4 3

In this case, the child process first applies the transformation T1, and writes the modified pixels to the pipe, along with the coordinates. The parent process is scheduled in between, and it reads the data from pipe whenever possible, and by using the coordinates it form a vector of the pixel data, and it continue to read until all pixels are read from the pipe. Once the parent forms the complete vector, it applies the transformation T2 and gives the final output, and hence in this way the correctness is ensured.

3 Analysis of Run times:

The following table shows the time taken for each method.

Image No.	File Size	part1	part2_1a	part2_1b	part2_2	part2_3
1	387.5KB	0.039902	0.040460	0.041581	0.071444	0.072230
2	580.2KB	0.040341	0.040529	0.041226	0.071352	0.074933
3	509.8KB	0.039593	0.040464	0.041883	0.069968	0.073534

Table 1: Run Times for each method

3.1 Observations:

1. The time taken in both cases of threads(part2_1a and part2_1b) is almost the same(in most of the cases), hence using a mutex lock and semaphores for synchronization has same effect on the run time.

2. The time taken in case of threads is slightly more than that compared to part 1, running the functions one after the other. Here we are applying the transformations one after the other and hence concurrent execution has no real benefit, and the overhead from threads causes this slight increase in runtimes.

3. The time taken in case of 2 different process for transformations(part2_2 and part2_3) is significantly more compared to the other methods. This is because process creation and context switching etc has huge overheads and hence it takes more time.

4. The time taken in both communication using shared memory and communication using pipes is more or less comparable, but shared memory is slightly faster compared to pipes.

4 Analysis of relative ease of implementing and debugging:

The methods in increasing order of ease of execution are as follows:
Shared Memory < Pipes < Threads < Direct Execution(Part 1)

Errors Faced:

1. In shared memory, there were some bugs introduced due to the failing of shmat and shmget

which were resolved by manual deletion of the shared address spaces, and also synchronization using semaphores across processes was hard to implement.
2. In case of pipes, we tried some other approaches which lead to blocking of child process due to the pipe overflow.

5 Instructions to run:

Run the following commands
$ make part1
$ make part2_1a
$ make part2_1b
$ make part2_2
$ make part2_3
To run the corresponding parts.

In each of this execution, we assume that the input file is sample.ppm is present in the **pwd** and the output is written to the corresponding output files.

If there is an error in make part2_2, then it is likely that the shared memory is corrupt in which case, use command
$ ipcrm -m <shmid> to delete the shared memory with the id, and run the part2_2 again. You can get the shmid, by using the command, $ ipcs