CS 314 – Operating Systems Laboratory
Assignment–6 Report

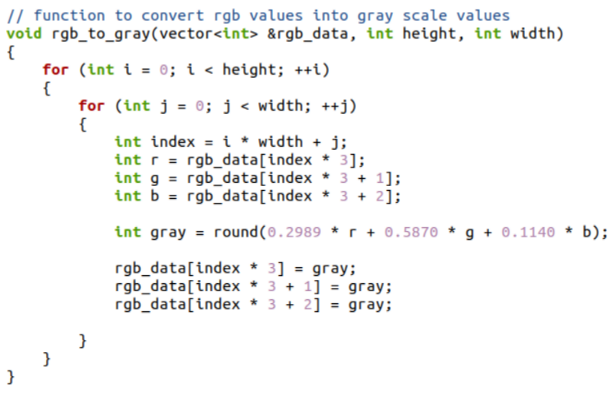
Student 1: Manche Pavanitha
Roll-No: 200010027

Student 2: Sahaja Nandyala
Roll-No: 200010032

1 Image Transformations

1.1 RGB to Gray scale

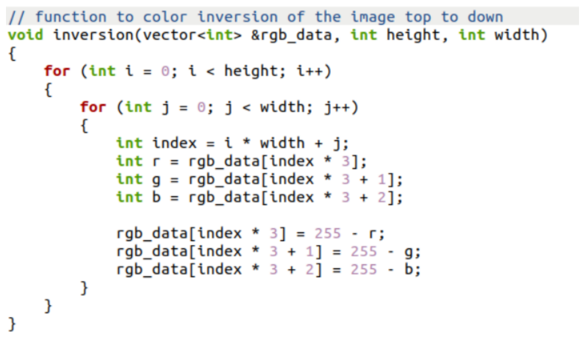
In this image transformation we are converting the image from RGB scale to Gray scale. The arguments to function are pointer to matrix of image data of image which is declared global and the function make changes to this matrix, height of image, width of image. Following is the code for the rgb to gray scale image transformation.



```
// function to convert rgb values into gray scale values
void rgb_to_gray(vector<int> &rgb_data, int height, int width)
{
    for (int i = 0; i < height; ++i)
    {
        for (int j = 0; j < width; ++j)
        {
            int index = i * width + j;
            int r = rgb_data[index * 3];
            int g = rgb_data[index * 3 + 1];
            int b = rgb_data[index * 3 + 2];

            int gray = round(0.2989 * r + 0.5870 * g + 0.1140 * b);

            rgb_data[index * 3] = gray;
            rgb_data[index * 3 + 1] = gray;
            rgb_data[index * 3 + 2] = gray;

        }
    }
}
```

1.2 Inversion

In this image transformation we are inverting the color of image. The arguments to function are pointer to matrix of image data of image which is declared global and the function make changes to this matrix, height of image, width of image. Following is the code for the inversion transformation.



```
// function to color inversion of the image top to down
void inversion(vector<int> &rgb_data, int height, int width)
{
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            int index = i * width + j;
            int r = rgb_data[index * 3];
            int g = rgb_data[index * 3 + 1];
            int b = rgb_data[index * 3 + 2];

            rgb_data[index * 3] = 255 - r;
            rgb_data[index * 3 + 1] = 255 - g;
            rgb_data[index * 3 + 2] = 255 - b;
        }
    }
}
```

2 Part2

2.1a Synchronization using atomic operations

In this case, T1 and T2 are two different threads in the same process and the synchronization of these two is done through **atomic operations.**

```
atomic_flag l = ATOMIC_FLAG_INIT; // Initializes the atomicFlag to FALSE

atomic_flag_test_and_set(&l) //Spin lock insertion

atomic_flag_clear(&l); // Signal Flag Done
```

2.1b Synchronization using semaphores

In this case, T1 and T2 are two different threads in the same process and the synchronization of these two is done through **semaphores.**

```
sem_t s; // Declaring a semaphore s

sem_init(&s, 0, 1); //initializes a semaphore and sets its initial value to 0

sem_wait(&s); //decrements the value of the semaphore

sem_post(&s);//Increments the value of the semaphore
```

2.2 Communication via Shared memory, Synchronization using semaphores

In this part, T1 and T2 are two different processes and they the interprocess communication is done via shared memory.

```
key_t k = 0x1256;
int shmid = shmget(k, sizeof(int) *height * width * 3, 0666 | IPC_CREAT);
// Gives shared memory identifier associated with key k
if(shmid<0){
      perror("shmget");
      exit(1);
} // checks if created shmid is created correctly
int *rgb_data = (int *)shmat(shmid,NULL,0); // Attaches the shared memory segment
```

2.3 Communication via pipes

In this part, T1 and T2 are two different processes and they the interprocess communication is done via pipes.

```
int pipefd[2];
if (pipe(pipefd) == -1){
    perror("pipe");
    exit(EXIT_FAILURE);
} // Initialization of pipe
write(pipefd[1], &num, sizeof(num)); // writing a pixel to the pipe
read(pipefd[0], &temp, sizeof(int));  // Readin a pixel from the pipe
```

3 Results

Following are the input image and respective output image after performing gray scale and flip transformations.

Input image

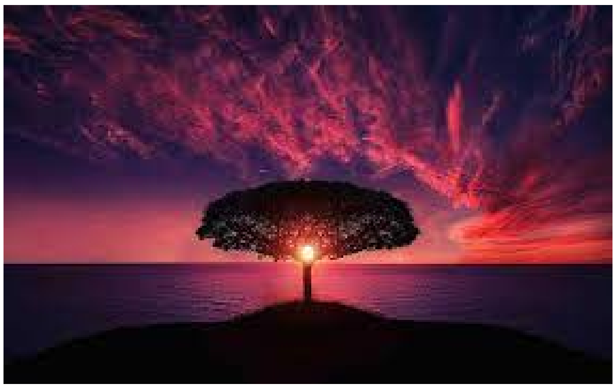


Image after applying RGB to gray scale transformation

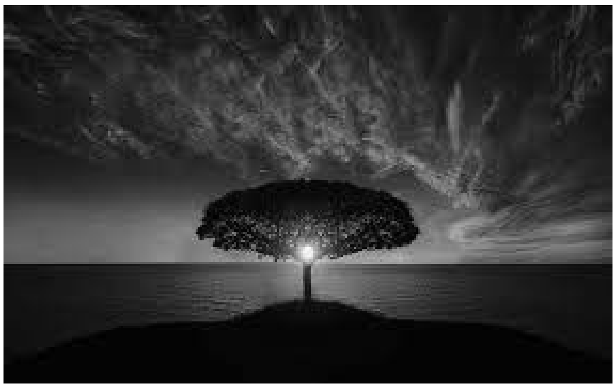


Figure 1: Rgb to gray scale transformation

Final image after applying both RGB to Gray scale transformation and inversion transformation

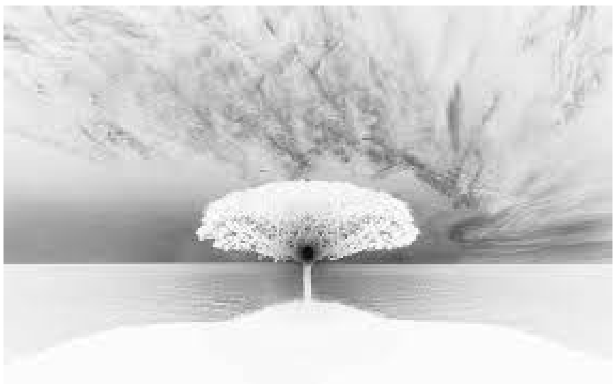


Figure 2: Gray scale to color inversion transformation

Following table shows the time taken by each method for different image sizes.

Image size	450KB	2MB	17MB
Part1 (Sequential))	0.001258	0.006718	0.048160
Part2.1a (Threads atomic operations)	0.007956	0.043753	0.210210
Part2.1a (Threads semaphores)	0.007867	0.092707	0.620952
Part2.2 (Process shared memory)	0.009444	0.030762	0.149271
Part2.3 (Process pipe)	0.077141	0.140503	0.736762

4 Analysis

Generally, we expect sequential execution to take more time, but that is not the case here. Here we can see that program where transformations are implemented sequentially is taking relatively less time than the programs with threads and processes where they communicate with each other. This may be because of the time reduced by the parallel execution of this processes is compensated by the time for waiting these to get into critical section.

Here from the results, it is clearly evident that pipes and semaphores are taking huge amount of time when compared with other implementations. In pipes, reading and writing operations take a lot of time as read operation in second transformation can start only after writing of that particular pixel into pipe is done by first transformation, thereby increasing execution time. Shared memory implementation takes less execution time as this implementation avoids reading data from the disk over and over again because the data is already in shared memory due to previous operations. So, the reduction in disk I/O reduces execution for this.

5 Proof of correctness

To check in each case, the pixels were received as sent, in the sent order, there are two files named t1.txt and t2.txt which stores the index values of the pixel in each transformation. Uisng the diff command one can ensure that the order of pixels after one function transformation is same as the received order of pixels in other transformation.

6 Ease/Difficulty of Implementing/Debugging each Approach

The methods involving threads were simple to develop because they shared a data segment, making it simple to use primitive constructions like semaphores and atomic variables. While methods combining several processes were challenging to execute and debug because it was necessary to take sufficient precautions to ensure that the right values are passed by processes in the right order and that they are correctly received by other processes. Implementation using shared memory was particularly challenging because the shared memory's structure needs to be maintained by us and implementation using pipes was also comparatively difficult beacause of writing and reading operations by two different processes i.e., parent and child.