CS 314 – Operating System Laboratory

Assignment 6 – Report

Tejal Ladage
210010026

Ashwin Waghmare
210010060

1. Part 1

Image transformations:

1) RGB to grayscale

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

```
void rgb_to_grayscale(vector<int> &rgb_data, int height, int width)// fur
{
    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 grayscale = round(0.2989 * r + 0.5870 * g + 0.1140 * b);

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

        }
    }
}
```

2) Color inversion

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

```
void color_inversion(vector<int> &rgb_data, int height, int width) // fur
{
    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

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) // Assigns true to atomicFlag and returns before value

atomic_flag_clear(&l); // Signal Flag Done

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 lock; // Declaring a semaphore s

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

sem_wait(&lock); // Decrements the value of the semaphore

sem_post(&lock); //Increments the value of the semaphore

2) Communication via shared memory and synchronization using semaphores

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

```
key_t key = 0x6400;  // uniquely identifies the IPC resource

int shmid = shmget(key, sizeof(int) * height * width * 3, 0666 | IPC_CREAT);
// Gives shared memory identifier associated with key and returns shmid

if(shmid<0){         // checks if created shmid is created correctly
        perror("shmget");
        exit(EXIT_FAILURE);
}

int *rgb_data = (int *)shmat(shmid, NULL, 0);    // Attaches the shared memory segment to
                                                  //   process's address that has called shmat
```

3) Communication via pipes

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

```
int pipefd[2];
int x = pipe(pipefd);  // Initialization of pipe

if (x == -1){    //Checking if pipe initialization is successful
        perror("pipe");
        exit(EXIT_FAILURE);
}

write(pipefd[1], &data, sizeof(data));       // Writing a pixel to the pipe

read(pipefd[0], &data1, sizeof(data1));     // Reading a pixel from the pipe
```

3. Observation

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

Image size	Part1 (Sequential)	Part2.1a (Threads - atomic operations)	Part2.1b (Threads – Semaphores)	Part 2.2 (Process – Shared memory)	Part 2.3 (Process – Pipe)
~75mb	25.48	26.26	28.80	25.07	28.87
~25mb	6.94	7.45	8.08	7.35	8.00
~10mb	2.07	2.16	2.36	2.05	2.30
~5mb	1.43	1.53	1.58	1.43	1.64

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 time for shared memory.

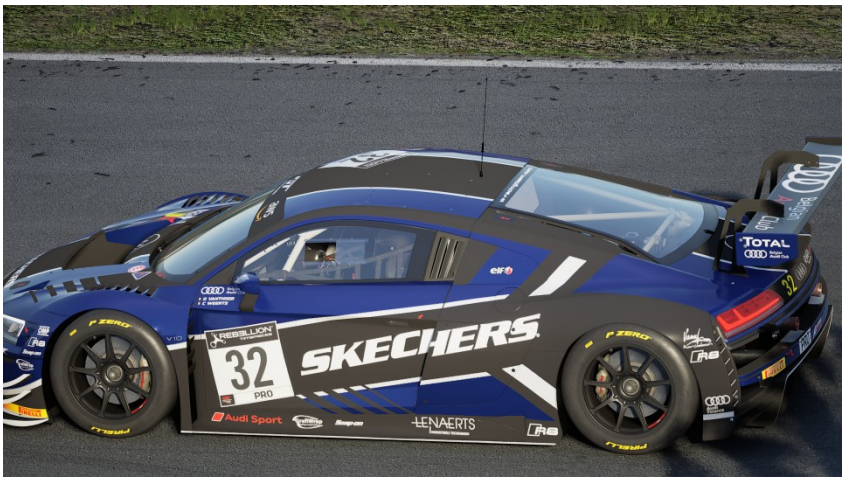
5. Proof of correctness

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. Using the diff command one can ensure that the order of pixels sent 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.

7. Results



RGB
To
Grayscale

Color
Inversion