Harrithha, Shashank P
200010018, 200010048

CS314 Operating Systems Lab
Lab 6

---

1 Part1: Image Transformations

In this part of the assignment, two sequential transformations namely, **RGB to Grayscale** & **Increase Brightness** were made. The original image is shown below without transformations

Figure 1: **Original Image**

1.1 RGB to Grayscale

The first transformation applied was RGB to Gray-scale, i.e., it converts image from colour to black and white. To convert an image to Gray-scale, a weighted average method was used. The new pixel value is as follows:

$$NewRed = NewBlue = NewGreen = (r \times 0.2989) + (g \times 0.5870) + (b \times 0.1140)$$

Here r, g, b are the RGB values corresponding to a pixel of original image. According to this equation, red has a contribution of about 30%, green has a contribution of 59% and blue has a contribution of only about 11%.

Figure 2: **RGB to Grayscale**

1.2 Increase Brightness

To increase the brightness of an image, we multiply each r,g,b pixel value of the original image by **3** and the new pixel value would be the minimum of 255 and this new scaled value. This corresponds to a **200%** increase in brightness

$$(NewR,\ NewG,\ NewB) = min(((r,\ g,\ b)\ *\ 3),\ 255)$$

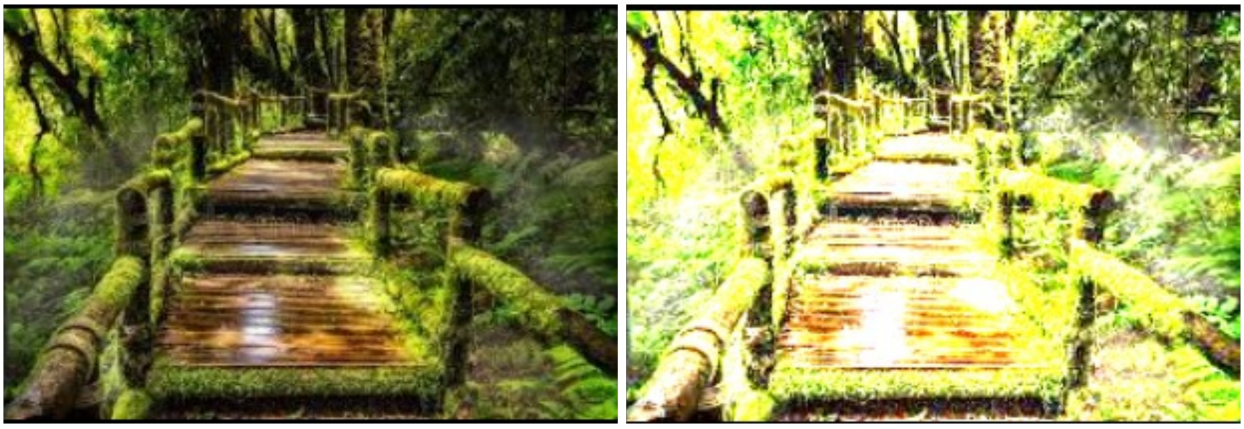

Figure 3: **Increased Brightness**

On applying both the transformations one after the other, we obtain the following image.

Figure 4: **Increased Brightness and Gray-scale**

2 Part2: Concurrency

It is given that a processor is embedded with two cores and the above operations are to be done by having the file read and T1 done on the first core, passing the transformed pixels to the other core, where T2 is performed on them, and then written to the output image file. In the following subsections, different implementations using Synchronization Primitives are explained.

2.1 Atomic Operations

T1 and T2 are performed by 2 different threads of the same process. They exchange data and communicate through the process' address space itself. To achieve mutual exclusion an array of booleans were used to keep track of whether a pixel has undergone transformation T1. To perform atomic operations on this array of booleans, we used **atomic** library in C++. The following code snippet shows atomic operations in **T1** and busy wait in **T2**.

```
// Thread performing T1
{
    int r = imgData[i][j][0];
    int g = imgData[i][j][1];
    int b = imgData[i][j][2];

    // Atomic operation to set operation completed
    transform_1_completed[i * width + j] = atomic<bool>(true);
}
```

```
// Thread performing T2
{
    // Wait for first transformation to finish on pixel i, j
    while (transform_1_completed[i * width + j] == false)
        ;
    imgData[i][j][0] = gray;
    imgData[i][j][1] = gray;
    imgData[i][j][2] = gray;

}
```

We observed the same image as the final output as in the above sequential case.

2.2 Semaphores

T1 and T2 are performed by two different threads of the same process. They communicate through the process' address space itself. Using **semaphore.h** library in C++ we can synchronize the two thread so that the second transformation thread does not overtake the first one. For this we initialized the semaphore to zero. We performed **sem_post** after first transformation and performed **sem_wait** after second transformation per iteration. In this case the second transformation will never overtake the first one as the semaphore will be less than zero. Both the transformations save the pixels in the global array of size **height * width * channels**.

```
// First Thread Transformation
{
    // Update pixel values
    imgData[i][j][0] = r;
    imgData[i][j][1] = g;
    imgData[i][j][2] = b;

    // Increment sem post variable in first transformation
    sem_post(&binarySemaphore);
}

// Second Thread Transformation
{
    // Wait for first process to call sem_post
    sem_wait(&binarySemaphore);
    r = imgData[i][j][0];
    g = imgData[i][j][1];
    b = imgData[i][j][2];
}
```

We observed the same image as the final output as in the above sequential case.

2.3 Shared Memory

T1 and T2 are performed by two different processes that communicate via shared memory. For this we used **ipc.h** and **shm.h** library in C++. The following actions were performed.

1. Create a shared memory segment of size ($height * width * channels$) in the parent program.
2. Fork the process to create a child process and execute the first transformation in the parent and second in the child process.
3. Attach the shared memory to both the parent and child transformation functions.
4. To synchronize we stored semaphores in a shared memory too. The process of synchronization is similar to **Part 1B** of the assignment.

```
// Create shared memoory segment
if ((shmid = shmget(key, imgWidth *imgHeight * 3 * sizeof(int), IPC_CREAT | 0666)) < 0)
{
    cout << "shmget error" << endl;
    perror("shmget");
    return 1;
}

// Fork the process
int pid = fork();

// Child Process second transformation
if (pid == 0)
{
    ofstream outfile;
    // string out = argv[2];
    outfile.open(argv[2], ios::trunc); // open file in append mode
    // Child process
    // Transform image to grayscale
    RBGToGrayScale(imgHeight, imgWidth, outfile);
    outfile.close(); // close file descriptor
}

// Parent process first transformation
else
{
    // Parent process
    // Increase brightness of image
    IncreaseBrightness(imgData, imgHeight, imgWidth);
    wait(NULL);
}
```

We observed the same image as the final output as in the above sequential case.

2.4 Pipes

T1 and T2 are performed by two different processes that communicate via pipes. The following actions were performed.

1. Create pipe for communication between T1 and T2.
2. Fork the process to create a child process and execute the first transformation in the parent and second in the child process.
3. In the child process, (when pid = 0) the write end of the child's copy of pipe is closed.
4. In the parent process, (when pid $\neq$ 0) the read end of the parents' copy of pipe is closed.
5. Write the data containing transformed pixels after the first transformation done by parent, into the pipe.
6. The child will then read the transformed pixels from the pipe (written by parent) and perform the next transformation.

```
1  // Create pipe for inter process communication
2
3  int fd[2];
4  if (pipe(fd) == -1)
5  {
6      cerr << "Failed to create pipe." << endl;
7      return 1;
8  }
9
10 // Fork made to create 2 processes
11 pid_t pid = fork();
12 if (pid == -1)
13 {
14     cerr << "Failed to fork process." << endl;
15     return 1;
16 }
17
18 if (pid == 0)
19 {
20     ofstream outfile;
21     outfile.open(argv[2], ios::trunc);
22     // Child process
23     close(fd[1]);
24     vector<vector<vector<int>>> input_data(imgHeight, vector<vector<int>>(imgWidth,
           vector<int>(3)));
25     // Transform image to grayscale
26     RBGToGrayScale(input_data, imgHeight, imgWidth, outfile, fd[0]);
27     outfile.close(); // close file descriptor
28     close(fd[0]);
29 }
30 else
31 {
```

```
    // Parent process
    // Increase brightness of image
    close(fd[0]);
    IncreaseBrightness(imgData, imgHeight, imgWidth, fd[1]);
    close(fd[1]);
    wait(NULL);
}
```

We observed the same image as the final output as in the above sequential case.

3 Sample Images

For analysis we have taken images of three sizes viz. sample1: **300 x 203**, sample2: **500 x 333**, sample3: **1000 x 1333**. The images and their transformations are shown below:

Figure 5: **Sample 1: Original and Transformed**

Figure 6: **Sample 2: Original and Transformed**

Figure 7: **Sample 3: Original and Transformed**

4 Pixels received as sent

To verify that Transformation 1 was performed before Transformation 2 per iteration, we setup variables called **trans1** and **trans2**. These variables kept track of the latest pixel index that was transformed for Transformation 1 and Transformation 2 respectively. In Transformation 2, after every iteration we check whether Transformation 2 has overtaken Transformation 1. The code snippet is shown below.

```
//  Tranformation 1
{
    mtx.lock();
    trans1[0] = i;
    trans1[1] = j;
    mtx.unlock();
}

//  Tranformation 2
{
    mtx.lock();
    trans2[0] = i;
    trans2[1] = j;
    if (trans1[0] < trans2[0] || (trans1[0] == trans2[0] && trans1[1] < trans2[1]))
    {
        cout << "T2 overtook T1 " << trans2[0] << " " << trans2[1] << "\n";
    }
    mtx.unlock();
}
```

After running this program we found that at no point Transformation 2 overtook Transformation 1. The output is for **Part 2: 1** but can be replicated for multiprocessors using shared memory and

shared semaphores.

Figure 8: **Sanity Check for Part 2: 1**

5 Run Time (in microseconds) and Speed Up

Speed up for each part is calculated as the ratio of the execution time for part1 (sequential transformation) to the execution time for the current part.

Image Size	**Part1 (Sequential)**	**Part2.1a (Threads - atomic operations)**	**Part2.1b (Threads – Semaphores)**	**Part 2.2 (Process – Shared memory)**	**Part 2.3 (Process – Pipe)**
300 x 203	96277	110790	142328	95696	325897
500 x 333	256100	330107	377636	260711	995871
1000 x 1333	2480445	2507716	3412118	1941125	6897502

Table 1: **Execution Time**

Image Size	**Part1 (Sequential)**	**Part2.1a (Threads - atomic operations)**	**Part2.1b (Threads – Semaphores)**	**Part 2.2 (Process – Shared memory)**	**Part 2.3 (Process – Pipe)**
300 x 203	1	0.869	0.676	1.00	0.295
500 x 333	1	0.775	0.678	0.982	0.257
1000 x 1333	1	0.989	0.726	1.277	0.359

Table 2: **Speed Up**

- We see that the sequential program's approach should have taken more time than other approaches as it does everything sequentially whereas other approaches have multiple threads and processes which have some sort of parallelization in completing the required task. But this is not observed.
- It is due to the fact that the critical section has very less computation so speed up obtained through parallelization is more than compensated by increased overhead of communication in other approaches.
- On the whole from the above table, we see that the shared memory approach takes a little less time than the sequential approach. All the other approaches take more time.

6 Ease/Difficulty of Implementing/Debugging in Approaches

The approaches using threads was relatively easy to implement as they share a data segment, making it simple to use constructs such as semaphores and atomic variables. On the other hand, approaches that involve different processes were challenging to troubleshoot because it was necessary to ensure that the correct values were passed by the processes in the right order and received properly by other processes. In particular, implementing shared memory was complex because we had to maintain the structure of the shared memory, unlike using pipes.