CS 314 – Operating Systems Lab
Lab–6 Report

Student 1: Kavali Sri Vyshnavi Devi
Roll-No: 200010023

Student 2: Meghana Sripalle
Roll-No: 200010028

1 Part 1

In Part-1, 2 sequential image transformations, RGB to Grayscale and Thresholding have been implemented. The different transformations have been elaborated on below:

1.1 RGB To Grayscale

For the conversion of an RGB image to a Grayscale image, we used the weighted method or luminosity method. In the other method called average method, an average of all the 3 colors is taken but we end up getting a black image rather than a gray one.
It is noticed that the red color has more wavelength compared to the three colors, and green is the color that has not only less wavelength than red color but also green is the color that gives more soothing effect to the eyes. It means that we have to decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. The resultant equation is the following:

$$pixelData = (redPixelValue * 0.2989) + (greenPixelValue * 0.5870) + (bluePixelValue * 0.1140)$$

where `pixelData` represents the pixel being converted to gray-scale. Red,green and blue values are converted to the value in `pixelData`.

1.2 Thresholding

Thresholding is a common immage processing technique used to segment an image into different regions based on pixel intensity values. In thresholding, we convert an image from colour or grayscale into a binary image, i.e., one that is simply black and white. The threshold value set here is 128, If the red, green or blue constituent of a pixel is greater than 128, its value is changed to 255 (white) while it is changed to 0 (black) if the original value is lesser than or equal to 128. This way, it is ensured that a pixel is either white or black.

$$pixelData = pixelData < threshold ? 0 : 255$$

2 Part 2

It is given that a processor is embedded with two cores. It is to be done by reading the input image file data followed by T1 being run on the first core, then passing the transformed pixels to the other core, where T2 is run on them, and then written to the output image file. This has been implemented in different ways using processes, threads and different types of synchronizations.

2.1 Synchronization using atomic operations

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space. We performed synchronization using atomic operations. We noticed that we got the image with thresholding performed on it after using the atomic flag and done_till_now array.

```
// Initializing the atomic flag called 'locked'
atomic_flag locked = ATOMIC_FLAG_INIT;
// This array 'done_till_now' is initialized in order to keep track
of the pixel being transformed
int done_till_now[2] = {0,0}

 while (atomic_flag_test_and_set(&locked));
float temp = 0;
for (int k = 0; k < 3; k++)
{
    temp += (float)data_arg[i][j][k] * weights[k] * 1.0;
}
data_arg[i][j][0] = (int)temp;
data_arg[i][j][1] = (int)temp;
data_arg[i][j][2] = (int)temp;
done_till_now[0] = i + 1;
done_till_now[1] = j + 1;
atomic_flag_clear(&locked);
```

2.2 Synchronization using semaphores

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space. We performed synchronization using semaphores.

```
sem_wait(&locked);
float temp = 0;
for (int k = 0; k < 3; k++)
{
    temp += (float)data_arg[i][j][k] * weights[k] * 1.0;
}
data_arg[i][j][0] = (int)temp;
```

```
data_arg[i][j][1] = (int)temp;
data_arg[i][j][2] = (int)temp;
done_till_now[0] = i + 1;
done_till_now[1] = j + 1;
sem_post(&locked);
```

2.3 Communication through shared memory and synchronization using semaphores

T1 and T2 are performed by 2 different processes that communicate via shared memory. The following code was written to implement shared memory between the 2 processes.

Usage of the semaphore "BINARY_SEM"

```
sem_init(&BINARY_SEM, 0, 1);
sem_wait(&BINARY_SEM);
//transformation being performed on pixel
sem_post(&BINARY_SEM);
```

Implementing shared memory interface

```
//initialising shared memory for pixels array
int shmID = shmget(sharedKey, 8, 0666 | IPC_CREAT);
//attaching pixels array
pixelData = (int *)shmat(shmID, NULL, 0);
```

2.4 Communication through pipe

T1 and T2 are performed by 2 different processes that communicate via pipes. The following code was written to implement pipe between the processes.

Initialising pipe:

```
int pipeReturn;
pipeReturn = pipe(pipeFileDescriptor);
if (pipeReturn == -1)
{
    perror("Failed pipe detected.\n");
}
```

Writing to the pipe:

```
for (int k = 0; k < 3; k++)
{
    data_arg[i][j][k] = (int)temp;
    transfer[k] = data_arg[i][j][k];
}
write(pipeFileDescriptor[1], transfer, BUFFER_SIZE);
```

Reading from the pipe:

```
read(pipeFileDescriptor[0], transfer, BUFFER_SIZE);
for (int k = 0; k < 3; k++)
{
    data_arg[i][j][k] = transfer[k] < THRESHOLD ? 0 : 255;
}
```

3 Verification

Here, we need to ensure that the image data transformed by RGB_TO_GRAY function is ultimately passed to the THRESHOLDING function or else the image transformation wouldn't be correct as the pixel value will be changed in the RGB_TO_GRAY function but not in the THRESHOLDING function which culminates in the formation of erroneous output images. So, to avoid these kind of changes we used an array named done_till_now which contains the index values(rows and columns) from data array till which the first function has been transformed and so if second function wanted to access the other data we check the condition mentioned below:

```
if (i < done_till_now[0] && j < done_till_now[1])
{
    for (int k = 0; k < 3; k++)
    {
        data_arg[i][j][k] = data_arg[i][j][k] < threshold ? 0 : 255;
    }
    flag = true;
}
else
{
    flag = false;
}
//implemented after this
if (!flag)
{
    j -= 1;
}
//implemented after coming out of the loop with lopp variable j
if (!flag)
{
    i -= 1;
}
```

This if condition ensures that the second function does not process the image data before it has been transformed by the first function but the i, j value in the for loop gets incremented which we nullify by decreasing i and j value at the end of the loop if flag

has been changed. Here, flag indicates that the second function tried to access the data which has not yet been processed by the first function yet.

4 Results

The input image and the image created after the transformations can be seen below.

Figure 1: Input Image

Figure 2: Output Image

The time has been measured in microseconds.

Image Size	Part1 (Sequential)	Part2.1a (Threads -atomic operations)	Part2.1b (Threads – Semaphores)	Part 2.2 (Process –Shared memory)	Part 2.3 (Process – Pipe)
1MB	8202	15473	54028	19328	47295
2MB	24334	38666	140309	48782	126151
17MB	149918	254565	966026	321260	869603

5 Observations

The Part 1 approach runs sequentially while the other approaches have multiple processes and threads due to which the way they run is in a sort of parallel manner. When parallelization occurs there should be an improvement compared to sequential running but we don't see the expected behaviour.
It may be because the transformations we used are crafted in such a way that the critical section is computationally less and the improvisation in the time due to parallelization is being overshadowed by the increased communication overhead.

Comparatively, parallel threading employing atomic variables and semaphores is more expensive than a sequential approach. In addition, shared memory costs more than sequential method since it writes and reads items to and from memory more frequently hence going as far as accessing the memory every time. The method implementing pipe is the most expensive in this scenario since transmitting newPixelData(pixel variable) increases the entire execution time. We can see that the shared memory strategy is effective from a combined perspective, but it only permits shared memory within certain bounds. If the data is huge, the pipe takes a long time to transport and read on the other end. Semaphore Locks and Atomic Locks operate similarly but from the observations we can see that Atomic Locks perform comparatively better.

6 Difficulties Faced

Due to the fact that the threaded approaches shared a data segment, it was simple to create primitive components like semaphores and atomic variables. We inferred that methods involving many processes were challenging to implement and debug due to the need to take sufficient precautions to ensure that the right values are supplied by processes in the right order and that they are correctly received by other processes. As we must maintain the shared memory's structure in this situation rather than using pipes, the shared memory implementation proved particularly challenging. In shared memory approach using semaphores became a difficult task as we used sem_open method first later we found that there it is not working as expected and used sem_init to initialize the semaphore value.