CS-314 OS LAB-6 REPORT

K.S.N MANIKANTA (210010050),
M. SURESH (210010030)

1. Transformations

1.1 Grayscale

This transformation converts a colour RGB image to a grayscale image. Among different approaches to convert a given colour image to a grayscale image, we use the Luminosity Method, in which the contribution of blue to the final value is decreased, and the contribution of green is increased because the human eye is most sensitive to green light. This produces grayscale images that are perceived as the closest match to the original colour image. After some experiments and more in-depth analysis, researchers have concluded in the equation below, that the following value is assigned to each colour component of the pixel (r, g, b):

```
grayscale = 0.299 * red + 0.587 * green + 0.114 * blue
```

1.2 Edge Detection

Edge detection includes a variety of mathematical methods that aim at identifying edges, and curves in a digital
image at which the image brightness changes sharply or, more formally, has discontinuities. Among the many methods for edge detection, we have used the First-order method. The tranformation applied to each pixel is described below:

Let pixel(i, j).red = r(i, j)
r(i, j) = sqrt((f(j − 1) − f(j + 1))^2 + (f(i − 1) − f(i + 1))^2)
where,
f(j − 1) = r(i − 1, j − 1) + 2.r(i, j − 1) + r(i + 1, j − 1)
f(j + 1) = r(i − 1, j + 1) + 2.r(i, j + 1) + r(i + 1, j + 1)
f(i − 1) = r(i − 1, j − 1) + 2.r(i − 1, j) + r(i − 1, j + 1)
f(i + 1) = r(i + 1, j − 1) + 2.r(i + 1, j) + r(i + 1, j + 1)
A similar transformation must be applied to each colour component of the pixel.

1.3 Image Blur

This transformation blurs the image. We are implementing a simple image-blurring algorithm using a 3x3 kernel. The algorithm iterates through each pixel in the image, excluding the border pixels, and calculates the average colour value of the pixel and its surrounding 8 pixels. This is achieved by summing up the RGB values of the neighbouring pixels and dividing by the total count of pixels, including the

centre one. The resulting averaged RGB values are then assigned to the corresponding pixel in a new image container, thus producing a blurred version of the original image.

```
function blurImage(img):
    width = img[0].size()
    height = img.size()
    create modifiedImg with the same dimensions as
img

    for each pixel in img:
        initialize cnt, sR, sG, sB to 0
        for each neighboring pixel around the current
pixel:
            if the neighboring pixel is within bounds:
                add its RGB values to sR, sG, and sB
                increment cnt
        calculate the average RGB values for the current
pixel
        assign the average RGB values to the
corresponding pixel in modifiedImg

    return modifiedImg
```

2. Proof of correctness

A) Part 1

In part 1, both the transformations are done by the same process sequentially, i.e., one after the other through simple function calls and not using threads or pipes or shared memory. So, the second transformation starts only after the first transformation has been completed. There is no race condition and hence the pixels will be received as sent, in the sent order.

B) Part 2-1a

In part 2_1a, we use a lock(atomic flag) which ensures synchronization. It is initially free and available to use.

```
atomic_flag lock = ATOMIC_FLAG_INIT;
```

In the code, where access to the shared image vector occurs, a "test and set" function is invoked to determine whether the lock is currently engaged or available. A value of 0 indicates that the lock is available, while 1 denotes that it is engaged. When the value is 0, the atomic flag is set to 1, granting access to the critical section where the instructions are executed. Upon completion, the atomic flag is reset to 0, and the critical section is exited. To ensure the absence of data races and confirm the integrity of data handling, a validation process is conducted by comparing the output ppm files generated by this part and part 1 using the **diff**

command. This comparison verifies that both outputs are identical, demonstrating that data was processed and written correctly without any discrepancies, i.e., the pixels were received as sent, in the sent order.

C) Part 2-1b

In part 2_1b, we use semaphores in place of an atomic flag to ensure synchronization.

```
sem_t sm;
sem_init(&sm, 0, 1);
```

We set up a semaphore with an initial value of 1, showing that it's ready to be used. When a process wants to enter a critical section, it calls sem_post(), which increases the counter to 1, letting other processes know that it's available. On the other hand, sem_wait() checks if it's okay for the current process to enter the critical section by looking at the counter. If the counter is 0, meaning the critical section is already being used, the process decreases the counter and waits until it becomes 0 again before going in. This method helps prevent conflicts between processes accessing the same data. To make sure everything was handled correctly and there were no issues with how the data was managed, we compared the output image files generated by this part and part 1 using the **diff** command. The fact that both outputs were exactly the same showed that the pixels were received as sent, in the sent order.

D) Part 2-2

In part 2_2, two processes communicate and coordinate their actions using shared memory and semaphores. First, the parent process creates a child process. The parent focuses on detecting edges in an image, while the child turns the image into a grayscale. Both processes need to share information about the image, so they use a shared memory space. They also use semaphores to control access to this shared memory, ensuring that they don't interfere with each other. Commands like ‘shmget’ and ‘shmat’ help create and connect to this shared memory.

```
int shmid_img = shmget(key1, sizeof(struct Pixel) * r * c,
IPC_CREAT | 0666);
    if (shmid_img < 0) {
        perror("shmget error"); // if shared memory is not
created
        exit(1);
    }
    struct Pixel *image = (struct Pixel *)shmat(shmid_img, NULL,
0); //2D array to hold pixel data from input

    int sid = shmget(smkey1, sizeof(sem_t), IPC_CREAT | 0666);
    if (sid < 0) {
        perror("shmget error"); // if shared memory is not
created
        exit(1);
    }
    sem_t *sm = (sem_t *)shmat(sid, NULL, 0);
```

To make sure everything works smoothly without any data conflicts, the outputs generated by this part and part 1 are compared using the **diff** command. If the outputs are the same, it means the pixels were received as sent, in the sent order.

E) Part 2-3

In part 2_3, two processes communicate using pipes to share information about pixels in an image. First, the parent process makes a copy of itself to create a child process. The child handles the task of detecting edges in the image, while the parent converts the image to grayscale. Each pixel in the image is processed one by one by the child process, which sends the processed pixel to the parent through the pipe. The parent then receives each pixel from the child, processes it further as needed, and stores it in an array. The following arrays and functions are used are used:

```
int fds[2];
pipe(fds);
int temp[3];
write(fds[1], temp, sizeof(buffer));
read(fds[0], temp, sizeof(buffer));
```

Once both the child and parent finish their tasks, the final array of pixels in the parent is saved to a file. To ensure that all the data handling was done correctly and there were no issues with how they processed the pixels, a comparison is made between the output ppm files generated by this part and part 1 using the **diff**

command. If there are no differences in the outputs, it confirms that the pixels were received as sent, in the sent order.

3. **Relative ease/difficulty of implementing/debugging each approach**

A) Part 1 was the easiest to implement because no threading or inter-process communication(IPC) is happening. There is one process only which does the image transformations sequentially. So, it was very easy to implement/debug as our focus was only on the proper execution of the image transformation.

B)Part 2_1a and Part 2_1b were relatively easier(compared to Part 2_2 and Part 2_3) to implement because Part 2_1a involved implementing a simple spinlock to ensure the atomicity of the critical section of the code and Part 2_1b involved implementing semaphores to ensure the atomicity of the critical section. Compared to Part 1, the code was slightly more involved and required slightly more debugging. Compared to Part 2_2 and Part 2_3, it was easy to implement with primitives like semaphores and atomic locks(flag) and easy to debug because we don't need any specific mechanism like IPC which is used in the case of two processes trying to communicate.

C) Part 2_2 and Part 2_3 involved two processes, communicating with each other(either through piper or shared memory) to perform the image transformations. This requires us to synchronize the transfer and ensure each pixel is read correctly. These were relatively difficult to implement and debug because proper care had to be taken so that correct values were passed by a process in the correct order and they were properly received by the other process. We needed to do some debugging to get the synchronization to work properly. It is more difficult to implement data sharing and synchronization among processes because each process has a separate virtual memory created for itself which includes heap, stack, global data variables, and code segment. In pipes, there is less maintenance to be done by us, as compared to shared memory, where the whole structure had to be set by us.

4. Run-time and Speed-up Analysis

Image size	Part 1	Part 2_1a	Part 2_1b	Part 2_2	Part 2_3
5 mb	0.73 s	0.72 s	0.72 s	0.63s	1.02 s
8 mb	1.06 s	1.02 s	1.01 s	0.93 s	1.32 s
25 mb	3.63 s	3.34 s	3.21 s	2.86 s	4.18 s
73 mb	9.43 s	8.95 s	8.76 s	7.79 s	13.37 s

A) From the above table, we can see that **Part 1, Part 2_1a and Part 2_1b** have almost similar runtime across different images of different sizes as the input. These parts don't use any IPC and the entire task is executed by a single process. The first part executes both transformations sequentially. The next two parts use threads to implement the transformations. Usually, parallel execution using the threads(part 2_1a, part 2_1b) should take less time compared to the sequential execution. But, since parallel execution using threads uses locks and when one process is holding the lock, the other is idle and waiting for the lock. Once on receiving the lock context switch happens, which is an overhead and increases the run time. Hence, these have almost similar runtime.

B) **Part 2_2** uses shared memory for IPC (inter-process communication). We see that it is significantly faster compared to the first three implementations. In shared memory, the processes are attached to their memory segment before acquiring the lock, which reduces the search time for the data. Even though there are context switches, there is a huge buffer in the shared memory. This will help us to decrease the number of context switches. **Part 2_3** uses pipes for IPC. We observe that it takes significantly more time to execute compared to the previous 4 implementations. The critical section code is quite small, to the point where the IPC becomes an overhead. We are sending updated values from one task(in child) to another(in parent) in batches and each pixel in 2nd task requires completion of the 1st task for the next few pixels to process further. As a result of the sending and receiving times, the total run time in this case is increased.