CS314 : Operating Systems Lab

Lab 6

Sourabh Bhosale (200010004)
Dibyashu Kashyap (200010013)

February 19, 2023

1 Part 1

In Part-1 of the assignment, 2 sequential transformations were made namely ConvertGrayscale HorizontalBlur. The idea and implementation is explained in following subsections.

ConvertGrayscale

A weighted average method was integrated for conversion of RGB image to Grayscale type. Since red has highest wavelengths of all 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. So, the new equation that form is as follows:

- values[p][q].red = (colour_blue * 0.114) + (colour_red * 0.299) + (colour_green * 0.587);
- values[p][q].green = (colour_blue*0.114) + (colour_red*0.299) + (colour_green*0.587);
- values[p][q].blue = (colour_blue * 0.114) + (colour_red * 0.299) + (colour_green * 0.587);

where, values is the image matrix that is updated to create a new transformed Image. According to this equation, Red has a contribution of about 29.9%, Green has a contribution 58.7% which is greater in all three colours and Blue has a contribution of only about 11.4%.

HorizontalBlur

Any kind of blur is essentially making every pixel more similar to those around it. In a horizontal blur, we can average each pixel with those in a specific direction, which gives the illusion of motion. For this program, we will calculate each new pixels value as half of its original value. The other half is averaged from a fixed number of pixels to the right. This fixed number can be called the BLUR_AMOUNT. The larger the value, the more blurring that will occur. This is done independently for the red, green, and blue components of each pixel.

Algorithm

- For each row of image :
  - For each pixel in the row:
    * Set r to be half the pixels red component.
    * Set g to be half the pixels green component.
    * Set b to be half the pixels blue component.
    * For i from 1 up to BLUR_AMOUNT:
      · Increment r by R×0.5/BLUR_AMOUNT, where R is the red component of the pixel i to the right of the current pixel.
      · Increment g by G×0.5/BLUR_AMOUNT, where G is the green component of the pixel i to the right of the current pixel.
      · Increment b by B×0.5/BLUR_AMOUNT, where B is the blue component of the pixel i to the right of the current pixel.
    * Save r, g, b as the new color values for this pixel.

You must also ensure you don't access pixels off the bounds of the image. For example, the third pixel from the right should only average itself (half weight), and the next two (at a quarter weight each). For this program, set the BLUR_AMOUNT to 50.

2 Part 2

It is given that a processor is embedded with two cores. It is 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. The following subsections, different implementation using Synchronization Primitives is explained.

Part2_1a

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself. In here, Synchronization had to be done using atomic operations.

```
atomic_flag get_lock = ATOMIC_FLAG_INIT;
//get_lock set to false initially

 while (atomic_flag_test_and_set(&get_lock));
atomic_flag_clear(&get_lock);
```

After using above logic and appropriate flags, we got same image.

Part2_1b

T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself. In here, Synchronization had to be done using Semaphores.

```
sem_t s;
sem_init(&s, 0, 1);
sem_wait(&s);
sem_post(&s);
```

Using above mentioned semaphores, we implemented the Semaphore synchronization primitive.

Part2_2

T1 and T2 are performed by 2 different processes that communicate via shared memory.

```
int shmid = shmget(key, sizeof(struct pixel) * h * w, 0666 | IPC_CREAT);
sem_t *s = sem_open(SEM_NAME, O_RDWR);
sem_wait(s);
sem_post(s);

int shmid = shmget(key, sizeof(struct pixel) * h * w, 0666 | IPC_CREAT);
sem_t *s = sem_open(SEM_NAME, O_RDWR);
sem_wait(s);
sem_post(s);



key_t key = 0x1234;
int shmid = shmget(key, sizeof(struct pixel) * (h)*w, 0666 | IPC_CREAT);

values = (struct pixel *)shmat(shmid, NULL, 0);
sem_t *s = sem_open(SEM_NAME, O_CREAT | O_EXCL, SEM_PERMS, INITIAL_VALUE); // named



shmdt(values);
shmctl(shmid, IPC_RMID, NULL);
```

Using above mentioned pieces of code, they were used in appropriate while loops to implement the IPC using Shared Memory.

Part2_3

T1 and T2 are performed by 2 different processes that communicate via pipes.

```
int create_pipe[2];
int data;
data = pipe(create_pipe);

//handled errors for pipe in perror

int create_pipe2[2];
int data2;

struct file_info head[1];

head[0].h = h;
head[0].w = w;
head[0].maxAscii = maxAscii;
write(create_pipe2[1], head, sizeof(struct file_info));
struct pixel forward_data[9];
write(create_pipe[1], forward_data, sizeof(forward_data));

struct pixel get_data[9];

struct file_info head[1];
read(create_pipe2[0], head, sizeof(head));

read(create_pipe[0], get_data, sizeof(get_data));
values[p][j] = get_data[0];
```

After using such Pipe File descriptors and implementation, expected results were achieved.

3 Image results

Figure 1: Input image

Figure 2: Grayscale effect

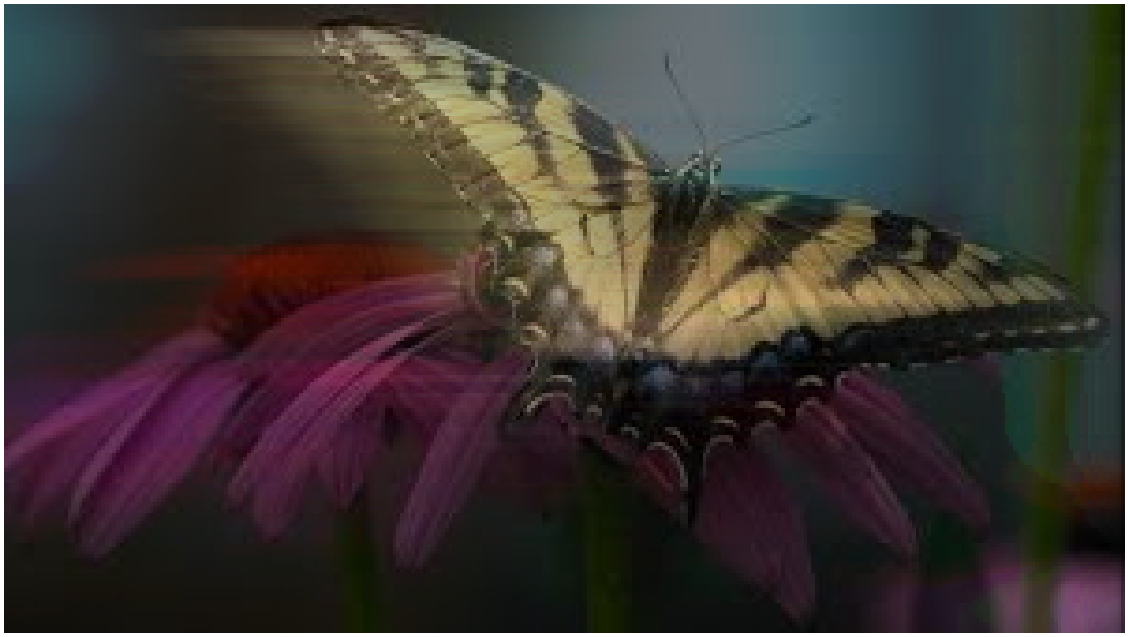

Figure 3: Horizontal blur/motion effect

Figure 4: Grayscale + Horizontal blur/motion effect

4 Proof of correctness

Method to prove correctness of ordering of Pixels:- Compare the output files directly using the diff command in Terminal. The command compares the output files, and tells about the file difference information if any i.e. command is :- diff correct_file.ppm output_file.ppm. I also printed the pixels on which each transformation was applied and it was observed that the second transformation was only applied to the pixels on which first transformation was already applied thus proving correctness of implementation.

5 Run time & Speed up comparison

Program file	Sync Primitives	Time taken (in microseconds)		
		File size : 480 KB	File size : 3.3 MB	File size : 7.4 MB
Part1.cpp	Sequentially	51155	836055	1847434
Part2_1a.cpp	Threads using Atomic Variables	99234	822463	1946293
Part2_1b.cpp	Threads using Semaphores	61094	851089	1870972
Part2_2.cpp	Threads using Shared Memory	92094	1567189	3170987
Part2_3.cpp	Processes using Pipes	92464	1284218	2976556

Figure 5: Analysis across 3 different images varying on size extremes

6 Analysis

- 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.

- Sequential approach is more expensive than Parallel Threading using atomics and semaphores.
- Shared Memory is more than Sequentially because of extra writing and reading values to and from memory.
- Pipeline Approach is most expensive because in my T2, each pixel needs next 50 pixels' T1 completed so that it can process further and also, I am sending updated values from T1 to T2 in batches using pipeline, so that's why that sending and receiving time increases total run time in this case.
- Thus, on a combined we can see that Shared Memory approach is good but it only allows shared memory up to some limitations.
- Pipeline takes a lot time to transfer and read on other end if data is large.
- Atomic Locks and Semaphore Locks perform almost same (Semaphores better a little bit).

7 Ease/Difficulty of Implementing/Debugging in Approaches

Approaches involving threads were fairly easy to implement as they shared data segment so it was easy to implement constructs like semaphores and atomic variables while approaches involving different processes were difficult to implement and debug because proper care was to be taken that correct values are passed by processes in correct order and they are properly received by other processes. Implementation through shared memory was specifically difficult because in this case structure of shared memory is to be maintained by us unlike implementation using pipes.