0S LAB 6 REPORT

Karthik.k 200010024
Chintu Nunavath 200010033

PART 1:

In this part of the assignment, we have performed two transformations sequentially to the input ppm image, one transformation is GREY_SCALE and the other is CONTRAST.

For the **GRAY_SCALE** transformation, we have taken 58.7% contribution from the green color and 29.9% contribution from the red color and 11.4% contribution from the blue color in a pixel.

*new_pixel_variable = old_pixel.red * 0.299 + old_pixel.green * 0.587 + old_pixel * 0.114*

For the **CONTRAST** transformation, we have taken a factor F into consideration. We have added a bu er value 50 to the RGB values and multiplied each pixel's RGB values by the factor F. If any RGB value exceeds the MAXCOLOR value, we have equated the RGB values with the MAXCOLOR.
*new_pixel = ((old_pix.r+50) * F, (old_pix.g+50) * F, (old_pix.b+50) * F)*

We then wrote the resultant pixel matrix to a new ppm file.

PART 2:

In this part of the assignment, we have a processor with two cores. We need to use the first core for the file reading and Transformation 1 (T1) and the second core for the Transformation 2 (T2). In this part we have three subparts.

1. T1 and T2 are performed by 2 different threads of the same process. They communicate through the process' address space itself.
   a. Synchronization using atomic operations
   b. Synchronization using semaphores

2. T1 and T2 are performed by 2 different processes that communicate via shared memory. Synchronization using semaphores.
3. T1 and T2 are performed by 2 different processes that communicate via pipes.

We have chosen GRAY_SCALE and CONTRAST as the two transformations. These two transformations are described in part1.

For the **subpart 2.1(a)**, T1 and T2 are performed by 2 different threads of the same process. Here the communication between the threads is done by the address space itself. We have used mutex for the synchronization. *pthread_mutex_t lock;*

pthread_mutex_init(&lock, NULL); pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock); pthread_mutex_destroy(&lock);

For the **subpart 2.1(b)**, T1 and T2 are performed by 2 di erent threads of the same process. Here the communication between the threads is done by the address space itself. We have used the semaphores for the synchronization. *sem_t lock;*

sem_init(&lock, 0, 1); sem_wait(&lock);
sem_post(&lock);

For the **subpart 2.2**, T1 and T2 are performed by 2 different processes that communicate via shared memory. Synchronization is achieved using semaphores.

*int shmid = shmget(0x1234, sizeof(struct pixel) * height * width, IPC_CREAT | 0666);*
*struct pixel *data = (struct pixel *)shmat(shmid, NULL, 0); struct pixel*
**image = (struct pixel *)shmat(shmid, NULL, 0); sem_t *lock;*
*char *SEM_NAME = "shared_semaphore";*
lock = sem_open(SEM_NAME, O_CREAT, 0644, 0);
sem_unlink(SEM_NAME);

For the **subpart 2.3**, T1 and T2 are performed by 2 different processes that communicate via shared pipes.

int fd[2];

if(pipe(fd)){
fprintf(stderr, "Pipe failed.\n");
return EXIT_FAILURE; }

write(fd[1], data, sizeof(data));

read(fd[0], image, sizeof(image));
close(fd[0]); close(fd[1]);

Run-time and Speed-up of each of the approaches:

Image size(in Bytes)	Part1(Sequential) (Time taken)	Part2.1a (Threads - atomic operations) (Time taken)	Part2.1b (Threads – Semaphores) (Time taken)	Part 2.2 (Process – Shared memory) (Time taken)	Part 2.3 (Process – Pipe) (Time taken)
9075	22.35	17.87	24.78	17.71	18.32
20569	8.18	7.00	5.02	7.09	7.65
7905	34.00	35.83	32.16	35.63	37.36

Normally a sequential approach should take more run time than the other approaches due to parallelism as we are using two different threads or processes for the two transformations, however it is not the case here. The speed up of parallelism in the other approaches is compensated by the increased overhead of the communication.

Method to prove in each case that the pixels were received as sent:

- In part2_1a, the pixels storing variable is global. Since global variables are accessible by threads, the pixels were received as sent, in the sent order.
- In part2_1b, also the pixels storing variable is global. Since global variables are accessible by threads, the pixels were received as sent, in the sent order.
- In part2_2, pixels are stored in a shared memory, so the pixels were received as sent, in the sent order. We can prove this by running this file after commenting the

transformations, then we can check whether the input file and output file are the same or not.

- In part2_3, pixels are shared via pipes among the processes. the pixels were received as sent, in the sent order. We can prove this by running this file after commenting the transformations, then we can check whether the input file and output file are the same or not.

The relative ease/ difficulty of implementing/ debugging each approach:

The approaches which involved threads are relatively easier to implement and debug than those approaches that involved different processes.

It was easy to implement the synchronization primitives like semaphores and atomic operations.

It was relatively di cult to implement pipes. And debugging in pipes is relatively di cult.

Image Results:

We got the same image for all the parts. So, we can confirm that all the locks are working and there is mutual exclusion among the threads.

Before and After Transformation