Operating Systems Lab 5 - CS 314

210010019, 210010032

**Part I**

Write a simple image processing application in C/C++.

- The input to the application should be a ppm image file.
- Your application must read this file and store the pixel information in a matrix.
- Then, it must perform two transformations to the image, one after the other. Choose some simple transformations such as “RGB to grayscale”, “edge detection”, “image blur”, etc. Let us call the two transformations T1 and T2.
- Write the resultant pixel matrix to a new ppm file.
- Usage: time ./a.out <path-to-original-image> <path-to-transformed-image>

**Soluton:**

Image Transformations Used:

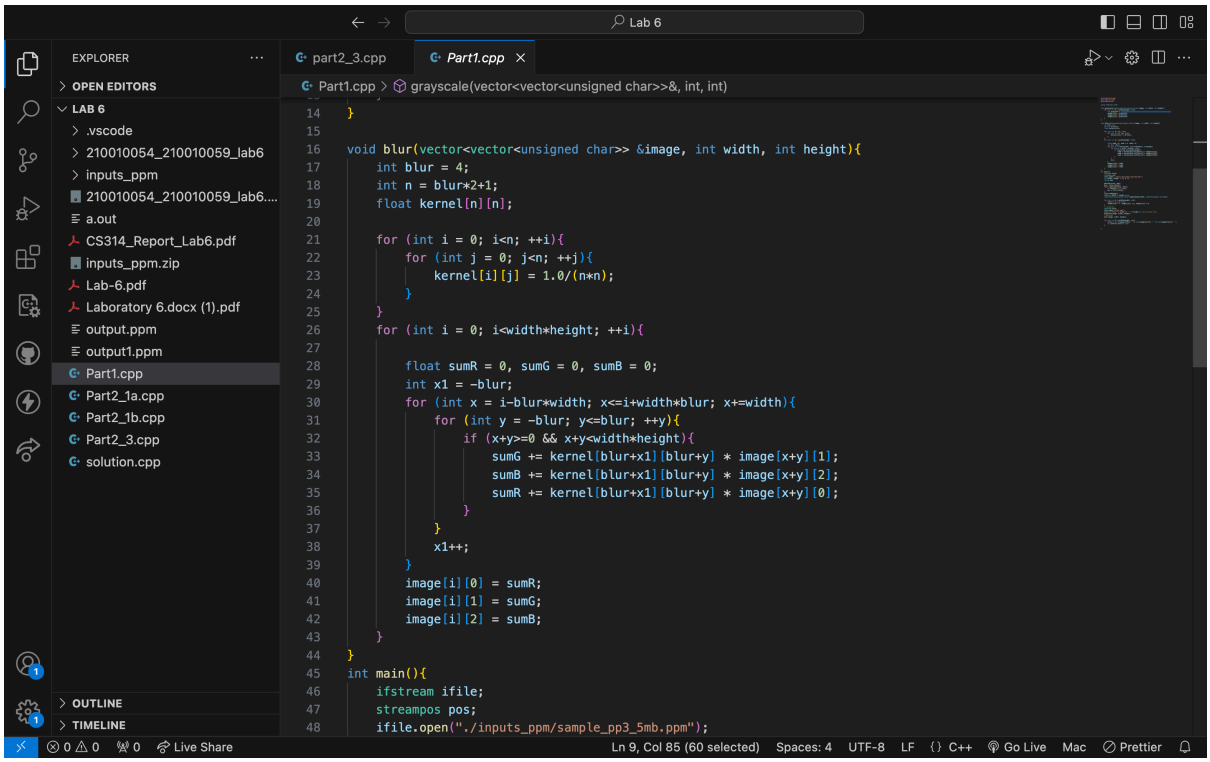
(1) RGB to Grayscale:

Grayscale value = 0.299*R+ 0.587 *G+ 0.144 *B;

```
7   void grayscale(vector<vector<unsigned char>> &image, int width, int height){
8       for (int i = 0; i<width*height; ++i){
9           int grayscale = 0.299*image[i][0] + 0.587 *image[i][1] + 0.144 *image[i][2];
10          image[i][0] = grayscale;
11          image[i][1] = grayscale;
12          image[i][2] = grayscale;
13      }
14  }
```

(2) Image Blur

Code:




Original Image:

GrayScale:


Image Blur:

**Part II**

Now suppose you have a processor with two cores. You want your application to finish faster.You can do this 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. Do this in the following ways:

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:

We have used atomic_flag_test_and_set and atomic_flag_clear to make operations atomic.

```
atomic_flag flag = ATOMIC_FLAG_INIT;
```

```
11  void grayscale(vector<vector<unsigned char>> &image, int width, int height) {
12      for (int i1 = 0; i1 < width * height; ++i1){
13          while (atomic_flag_test_and_set(&flag));
14          int grayscale = 0.299 * image[i1][0] + 0.587 * image[i1][1] + 0.144 * image[i1][2];
15          image[i1][0] = grayscale;
16          image[i1][1] = grayscale;
17          image[i1][2] = grayscale;
18          atomic_flag_clear(&flag);
19      }
20  }
```

```
22  void blur(vector<vector<unsigned char>> &image, int width, int height){
23      int blur = 4;
24      int n = blur * 2 + 1;
25      float kernel[n][n];
26      for (int i = 0; i < n; ++i){
27          for (int j = 0; j < n; ++j){
28              kernel[i][j] = 1.0 / (n * n);
29          }
30      }
31      for (int i2 = 0; i2 < width * height; ++i2) {
32          while (atomic_flag_test_and_set(&flag));
33          float sumR = 0, sumG = 0, sumB = 0;
34          int x1 = -blur;
35          for (int x = i2 - blur * width; x <= i2 + width * blur; x += width){
36              for (int y = -blur; y <= blur; ++y){
37                  if (x + y >= 0 && x + y < width * height) {
38                      sumG += kernel[blur + x1][blur + y] * image[x + y][1];
39                      sumB += kernel[blur + x1][blur + y] * image[x + y][2];
40                      sumR += kernel[blur + x1][blur + y] * image[x + y][0];
41                  }
42              }
43              x1++;
44          }
45          image[i2][0] = sumR;
46          image[i2][1] = sumG;
47          image[i2][2] = sumB;
48          atomic_flag_clear(&flag);
49      }
50  }
```

b. Synchronisation using semaphores

The sem _t variable sem is used as the semaphore.

sem wait() and sem post() functions are used to control access to the shared resource.

```
sem_t sem;
```

```
10  void grayscale(vector<vector<unsigned char>> &image, int width, int height){
11      for (int i = 0; i<width*height; ++i){
12          sem_wait(&sem);
13          int grayscale = 0.299*image[i][0] + 0.587 *image[i][1] + 0.144 *image[i][2];
14          image[i][0] = grayscale;
15          image[i][1] = grayscale;
16          image[i][2] = grayscale;
17          sem_post(&sem);
18      }
19  }
```

```
21  void blur(vector<vector<unsigned char>> &image, int width, int height){
22      int blur = 4;
23      int n = blur*2+1;
24      float kernel[n][n];
25
26      for (int i = 0; i<n; ++i){
27          for (int j = 0; j<n; ++j){
28              kernel[i][j] = 1.0/(n*n);
29          }
30      }
31      for (int i = 0; i<width*height; ++i){
32
33          sem_wait(&sem);
34          float sumR = 0, sumG = 0, sumB = 0;
35          int x1 = -blur;
36          for (int x = i-blur*width; x<=i+width*blur; x+=width){
37              for (int y = -blur; y<=blur; ++y){
38                  if (x+y>=0 && x+y<width*height){
39                      sumG += kernel[blur+x1][blur+y] * image[x+y][1];
40                      sumB += kernel[blur+x1][blur+y] * image[x+y][2];
41                      sumR += kernel[blur+x1][blur+y] * image[x+y][0];
42                  }
43              }
44              x1++;
45          }
46          image[i][0] = sumR;
47          image[i][1] = sumG;
48          image[i][2] = sumB;
49          sem_post(&sem);
50      }
51  }
```

2. T1 and T2 are performed by 2 different processes that communicate via shared memory. Synchronization using semaphores.(Single source file)

Solution:

The procedure employs shared memory to transfer picture data between threads T1 and T3, with synchronised access via a semaphore. Before accessing shared memory, the process waits for the semaphore. After updating shared memory, the process releases the semaphore. (More information about runtime is provided later.)

```
76      char P_3[3];
77      FILE *inputFile = fopen(argv[1], "r");
78      fscanf(inputFile, "%s%d%d%d", P_3, &width, &height, &maxValue);
79
80      key_t key = 0x1234;
81      int shmid = shmget(key, sizeof( pixel) *(height) * width, 0666 | IPC_CREAT);
82
83      pixel *values;
84      values = ( pixel *)shmat(shmid, NULL, 0);
```

3. T1 and T2 are performed by 2 different processes that communicate via pipes. (Single source file).

In this section, pipes are used to transfer picture data across threads. One thread writes to the pipe while another reads from it.

```
21  void blur(vector<vector<unsigned char>> &image, int width, int height, int pipefds[2]){
22      close(pipefds[1]);
23      int blur = 4;
24      int n = blur * 2 + 1;
25      float kernel[n][n];
26      for (int i = 0; i < n; ++i){
27          for (int j = 0; j < n; ++j){
28              kernel[i][j] = 1.0 / (n * n);
29          }
30      }
31      vector<vector<unsigned char>> newImage(height * width, vector<unsigned char>(3));
32      for (int i = 0; i < width * height; ++i){
33          int val;
34          read(pipefds[0], &val, sizeof(val));
35          newImage[i][0] = val, newImage[i][1] = val;
36          newImage[i][2] = val;
37      }
38      for (int i = 0; i < width * height; ++i){
39          float sumR = 0, sumG = 0, sumB = 0;
40          int x1 = -blur;
41          for (int x = i - blur * width; x <= i + width * blur; x += width){
42              for (int y = -blur; y <= blur; ++y){
43                  if (x + y >= 0 && x + y < width * height){
44                      sumG += kernel[blur + x1][blur + y] * newImage[x + y][1];
45                      sumB += kernel[blur + x1][blur + y] * newImage[x + y][2];
46                      sumR += kernel[blur + x1][blur + y] * newImage[x + y][0];
47                  }
48              }x1++;
49          }
50          newImage[i][0] = sumR;
51          newImage[i][1] = sumG;
52          newImage[i][2] = sumB;
53      }
54      image = newImage;
55      close(pipefds[0]);
```

**Result and Discussion :**

| Image size | Part1(Sequential) | Part2.1a (Threads - atomic operations) | Part2.1b (Threads – Semaphores) | Part 2.2 (Process – Shared memory) | Part 2.3 (Process – Pipe) |
|---|---|---|---|---|---|
| ~75mb | 19.392 | 15.628 | 21.692 | 10.183 | 18.809 |
| ~25mb | 5.275 | 4.622 | 6.804 | 4.044 | 6.231 |
| ~10mb | 2.287 | 1.538 | 2.014 | 1.618 | 2.047 |
| ~5mb | 1.258 | 1.086 | 1.392 | 1.249 | 1.308 |

- Because the sequential programme executes each step sequentially, it should have taken longer than other methods to achieve the task at hand.
- The sequential approach, which uses atomics and semaphores, is more expensive than parallel threading.
- Shared memory outperforms sequential memory because it allows for more writing and reading of values to and from memory.
- If the data is large, the pipeline takes a long time to transfer and read from the other end.
- Atomic and Semaphore locks behave roughly identically.

**Discuss the relative ease/ difficulty of implementing/ debugging each approach:**

Part 1:It was relatively simple to implement and debug because others relied on code from this component. The most difficult aspect of Part1 might be the ability to read and analyse ppm files, as well as implementation of T1 and T2.

Part 2. 1(a),(b): Since it mostly required reusing part 1's code along with a implementing atomic operations and semaphores, it was also rather simple to implement.

Part 2. 2: This one was most difficult to implement and debug because it required the knowledge of using shared memory, which we found difficult to implement and debug.

Part 2. 3: It was a bit challenging as well, but we managed to work it out.

**Initial Image and its transformation:**

Original Image:

Transformed Image: