Operating Systems Laboratory

Lab 6 Report

Pavan Kumar V Patil 200030041
Karthik J Ponarkar 200010022

February 19, 2023

1 Image tranformation techniques

Task: Write a simple image processing application in C/C++. The input to the application should be a ppm image file. The 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. Write the resultant pixel matrix to a new ppm file.

- The image transformations used for this laboratory are:
  - RGB to Grayscale (Luminosity Method)
  - Edge Detection (First-order method)

1.a Grayscale

- If we mix the three colors equally (RGB = (255, 255, 255)), we'll get white while the absence of all colors (RGB = (0, 0, 0)) means black.
- Grayscale is the simplest model since it defines colors using only one component: lightness. The amount of lightness is described using a value ranging from 0 (black) to 255 (white).
- On the one hand, grayscale images convey less information than RGB. However, they are common in image processing because using a grayscale image requires less available space and is faster, especially when we deal with complex computations.
- The contribution of blue to the final value should decrease, and the contribution of green should increase. After some experiments and more in-depth analysis, researchers have concluded in the equation below, the following value will be assigned to each color component of the pixel (r, g, b):

$$grayscale = 0.3 * R + 0.59 * G + 0.11 * B$$

1.b 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.
- The purpose of detecting sharp changes in image brightness is to capture important events and changes in properties of the world.
- There are many methods for edge detection, but most of them can be grouped into two categories, search-based and zero-crossing-based.
- We have used the First-order method which belongs to the search bases category.
- The tranformation applied to each pixel is described below: Let $pixel(i, j) \rightarrow red = r(i, j)$

$$r(i,j) = \sqrt{(f(j-1) - f(j+1))^2 + (f(i-1) - f(i+1))^2}$$

where,

$$f(j-1) = r(i-1, j-1) + 2.r(i, j-1) + r(i+1, j-1)$$
$$f(j+1) = r(i-1, j+1) + 2.r(i, j+1) + r(i+1, j+1)$$
$$f(i-1) = r(i-1, j-1) + 2.r(i-1, j) + r(i-1, j+1)$$
$$f(i+1) = r(i+1, j-1) + 2.r(i+1, j) + r(i+1, j+1)$$

- The similar transformation must be applied to each color component of the pixel.

2 Proof of correctness

2.a Part 1

- Since, in part 1 the two image transformations are done sequentially by the same process without using any threads, the second transformation will start modifying the pixels only when the first transformation is done with its execution.
- Hence, logically there won't be any data race happening with the pixels as mutual exclusiveness is always maintained in this case. Therefore, the pixels were received as sent, in the sent order.

2.b Part 2 1a

- In this case, an atomic flag is defined which is initially set to 0, which means initially the lock is free and available.

$$\textbf{atomic\_flag flag} = \textbf{ATOMIC\_FLAG\_INIT};$$

- In the part of the code, where the particular transformation accesses the critical section part (the common pixels array shared between the two transformations), the following test and set function will be called to check whether the lock is accessed or available. 0 value implies that the lock is available and 1 implies that the lock is already accessed.
- If the value is 0, it sets the atomic flag as 1, enters the critical section, executes the instructions, and finally, sets the atomic flag to 0 and comes out of the critical section.
- To justify that there was no data race and everything executed with proper data received being used and written, we used **diff** command among the two outputs ppm files and observed that there is no difference in the output.

2.c Part 2 1b

- In this case for handling the critical section problem, we have used binary semaphores. We can declare a semaphore with its initial value set as 1 (which means available) using the following command.

$$\textbf{sem\_init}(\textbf{sem}, \textbf{0}, \textbf{1});$$

- When the sem_post() method is called, it will increase the counter and set it to 1 which means the processes requesting for this critical section can access it.
- When the sem_wait() method is called, it will check whether the current process requesting for the critical can be allowed to enter or not in the following way.
- If the counter 0, then the counter will be decremented and the process will be allowed to access the critical section. If the counter was already set to 0, then the current will spin and wait until the counter becomes 0. This conceptually implies that there won't be any data race happening.
- To justify that there was no data race and everything executed with proper data received being used and written, we used **diff** command among the two outputs ppm files and observed that there is no difference in the output.

2.d Part 2 2

- In this case, the synchronization and communication between two processes are carried out using shared memory allocation for the pixels array and semaphores.
- The parent forks to create a child which the grayscale image transformation and the parent will execute the edge detection transformation. The parent will allocate shared memory for storing the pixels and common semaphores using a manually set key. And this created shared memory will use and updated by the child when it applies the grayscale transformation.
- The following commands were used for creating and attaching to the shared memory for the pixels and semaphores.

$$\textbf{int shmid\_pixel} = \textbf{shmget}(\textbf{pixels\_key}, \textbf{sizeof}(\textbf{pixel}) * \textbf{height} * \textbf{width}, \textbf{IPC\_CREAT}\|\textbf{0666});$$

$$(\textbf{pixel}*)\textbf{shmat}(\textbf{shmid\_pixel}, \textbf{NULL}, \textbf{0});$$

$$\textbf{int semid} = \textbf{shmget}(\textbf{sem\_key}, \textbf{sizeof}(\textbf{sem\_t}), \textbf{IPC\_CREAT}\|\textbf{0666});$$

$$(\textbf{sem\_t}*)\textbf{shmat}(\textbf{semid}, \textbf{NULL}, \textbf{0});$$

- To justify that there was no data race and everything executed with proper data received being used and written, we used **diff** command among the two outputs ppm files and observed that there is no difference in the output.

2.e Part 2 3

- In this case, communication between two processes are carried out using pipes for the pixels array.
- The parent forks to create a child which the edge detection image transformation and the parent will execute the grayscale image transformation. Here child processes each pixel sequentially and writes pixel by pixel to the pipe and parent reads each processed pixel from child and processes it and stores in array. After both child and parent finishes their respective task final array in parent is stored in file.
- The following functions were used for creating pipes:

$$\textbf{int fds}[\textbf{2}];$$

$$\textbf{pipe}(\textbf{fds});$$

$$\textbf{int buffer}[\textbf{3}];$$

$$\textbf{write}(\textbf{fds}[\textbf{1}], \textbf{buffer}, \textbf{sizeof}(\textbf{buffer}));$$

$$\textbf{read}(\textbf{fds}[\textbf{0}], \textbf{buffer}, \textbf{sizeof}(\textbf{buffer}));$$

- To justify that there was no data race and everything executed with proper data received being used and written, we used **diff** command among the two outputs ppm files and observed that there is no difference in the output.

3 The relative ease/ difficulty of implementing/ debugging each approach.

- `Part1` was relatively easier to implement because there is no threading or IPC happening. There is only one process for doing these image transformations sequentially. So, it was relatively easier to debug as our focus was only on the proper execution of the image transformation.
- `Part 2 1a` and `1b` use the concept of creating two threads and assigning each thread to execute an image transformation. As threads share common data space like a heap, global variable, etc. It was easier to debug because we don't need any specific mechanism like IPC which is used in the case of two processes trying to communicate. Hence, it was easy to implement constructs primitives like semaphores and atomic variables in this case.
- `Part 2 2` and `3`: While approaches involving different processes were relatively difficult to implement and debug because proper care had to be taken so that proper and correct values are passed by a process in the correct order and they are properly received by the other process. The IPC mechanisms used in this laboratory are shared memory and pips. Generally, 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. So, it naturally becomes tricky to implement the features using IPC.

4 Runtime and speed analysis

Image Size	Part 1 (in ms)	Part 2 1a (in ms)	Part 2 1b (in ms)	Part 2 2 (in ms)	Part 2 3 (in ms)
4.9 MB	152	157	155	117	257
24 MB	807	838	813	616	1370
81.1 MB	2805	2885	2857	2120	5051

Figure 1: Output of workload mix 4

- From the above table, we can verify that the `Part 1`, `Part 2 1a` and `Part 2 1b` have almost similar runtime across different images of different sizes as the input. These implements won't use any IPC and the entire task is executed by a single process. The latter two parts use threading to implement the transformations. The first part executes both transformations sequentially.
- `Part 2 2` uses shared memory for the inter-process communication. We observe it is significantly faster compared to the first three implementations.
- `Part 2 3` uses pips for the inter-process communication. We observe that it takes significantly more time to execute compared to the previous 4 implementations.

5 Sample input image execution