OS Lab5 Report

170010025, S V Praveen
170010039, Anudeep Tubati

Overview

This report showcases the notion of parallelism among processes using threads, shared memory and pipes. To achieve this, we use an example of applying 2 transformations (T1 and T2) to an image. T1 converts an RGB image into a grayscale image, whereas T2 applies edge detection filter on the output of T1.

Kernel for Edge Detection

1	2	1
2	-13	2
1	2	1

We use an image of dimensions 1877x1878

(from left) Akshat, Praveen and Anudeep trying to pose for a fake candid in Mussoorie

Method to Verify Output Image

We use the `diff` command provided in UNIX based systems to check 2 images byte by byte. If there's no output, it means that 2 images are identical, as given in the documentation for `diff`. Otherwise, the answer is wrong.

```
diff orig_output.ppm output_part1.ppm
```

Run-time

Program	Execution Time
Part 1	1.4s
Part 2 (1a)	1.26s
Part 2 (1b)	1.21s
Part 2 (2)	0.79s
Part 2 (3)	0.968s

As it is evident from the results, the sequential program in Part 1 takes the highest time. In Part 2 (1a) and (1b), the calculations have speeded up a bit. This owes to the fact that T1 will be applying its transformation while T2 uses the partial results of T1 to produce the final output.

In our case, we make T2 wait till at least 3 rows are done with T1, as T2 needs at least 3 rows and columns to work on. We ensure that T2 works on rows that are completed by T1 by keeping a count of completed rows (using an atomic variable in part (a) and semaphore in part(b)).

We can further interpret Part 2 (2) to be working really well as it almost halves the time taken by the standard sequential program. We write row by row into the shared memory which is then read by the other process. As in Part 2 (1), we process the rows for T2 once there are more than 3 rows available. This runs better than threading probably because of the overhead in the case of atomic variables and semaphores.

In Part 2 (3), we observe a little slower performance than Shared Memory as the writer has to wait when the pipe is full, unlike the shared memory case.

Ease of Implementation

Part 1 [Sequential]

The Sequential program was simple to implement as we just had to read the image and perform calculations on the input matrix.

Part 2 (1) [Threading with Atomic Variables and Semaphores]

It was tough to figure out how POSIX threads execute the `runner` function. The code to invoke the member function of a class and passing arguments to the same function through threads proved to be quite a bit of a challenge. That was until we came across the C++ library for threads. The C++ library was much simpler to use and after getting the threads, using atomic variables/semaphores to sync T1 and T2 was almost a one-liner.

Part 2 (2) [IPC using Shared Memory]

Trying to implement shared memory for IPC between two processes was an extremely arduous task. It involved debugging many segmentation faults which were finally fixed by finding a hacky way to pass integers through character arrays to solve the problem.

As we couldn't find a way to write the 2D array to shared memory, we allocated the whole 2D array in a 1D manner and simulated it as a 2D array in the program.

Part 2 (3) [IPC using Pipes]

We initially tried Implementing IPC using named FIFO pipes in Linux, however, trying to achieve error-free communication seemed to be impossible. We, therefore, switched to using simple pipes that can be used to communicate between a parent and child process. We parallelize it by feeding each row from the first function(grayScale()) to the second function(detectEdges()) as and when it is done.

Outputs

Fig. Grayscale image with Edge Detection