# CS314
# Lab 5

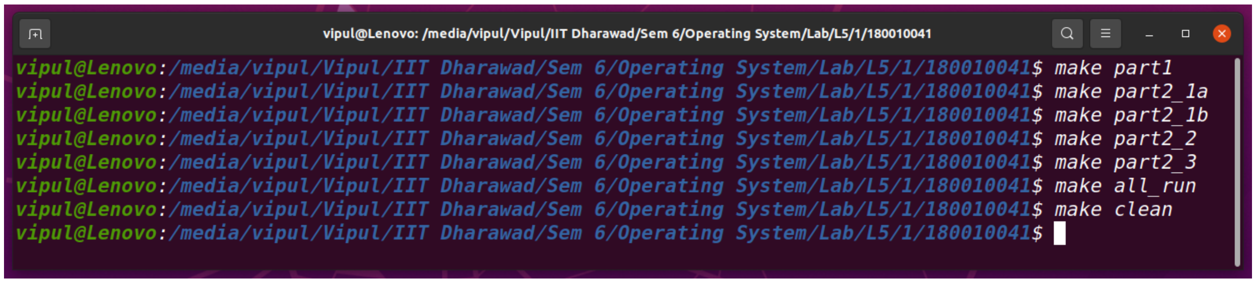
Vipul Yuvraj Nikam
180010041

-----------------------------------------------------

## How to run

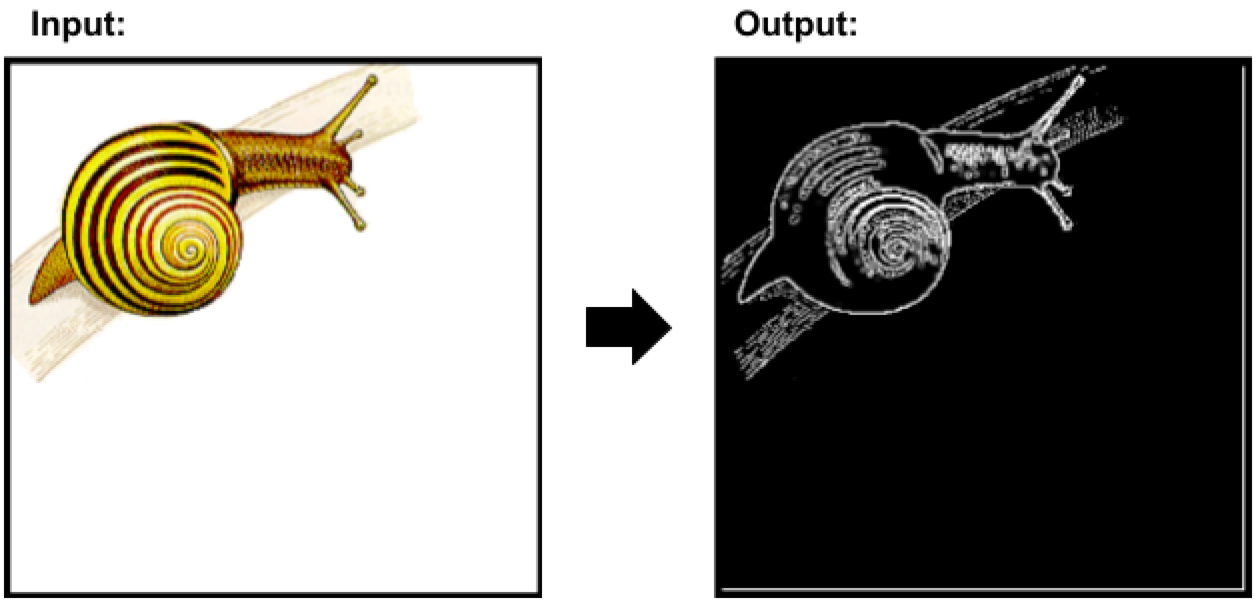
You can run all files together by just running make **all_run** command. Also you can run make clean command to delete all the previously created ppm files.

*Screenshot of terminal*



```
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make part1
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make part2_1a
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make part2_1b
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make part2_2
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make part2_3
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make all_run
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$ make clean
vipul@Lenovo:/media/vipul/Vipul/IIT Dharawad/Sem 6/Operating System/Lab/L5/1/180010041$
```

To run all 5 separately, commands have been added in makefile according to the given in question to get the following output.



## Overview

Notion of parallelism among processes using threads, shared memory and pipes is embedded here, applying 2 transformations T1 & T2. T1 is a grayscale image converter, whereas T2 is an edge detector on T1.

## Run time:

It took the time mentioned below to run each program. T2 waits until at least 3 rows are finished with T1, since T2 needs at least 3 rows and columns to work. T2 works on rows completed by T1 by keeping a count of completed rows using atomic variables in p_1a and a semaphore in part b.

| No. | Part | Time |
|---|---|---|
| 1 | P1 | 0.699s |
| 2 | P2_1a | 0.521s |
| 3 | P2_1b | 0.480s |
| 4 | P2_2 | 0.496s |
| 5 | P2_3 | 0.511s |

P2_2 works really well, since it almost halves the time it takes for the standard sequential program. write row by row in shared memory which is then read by the other process. As in P2_1, process the rows for T2 once there are more than 3 rows available. This works better than threading, probably due to overhead in the case of atomic variables and semaphores. In P2_3, I observe slightly slower performance than shared memory, as the writer has to wait when the pipe is full, unlike the case of shared memory.

## Kernel for Edge Detection:

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | -13 | 2 |
| 1 | 2 | 1 |

**Ease of Implementation:**

- P_1
    - Sequential
    - The Sequential program was simple to implement as I only had to read the image and perform calculations on the input matrix.
- P2_1
    - Threading with Atomic Variables and Semaphores
    - It was difficult to figure out how the POSIX threads execute the execution function. The code to invoke the member function of a class and pass arguments to the same function through threads proved to be quite a challenge.
    - That was until I found the C++ library for threads. The C++ library was much simpler to use and after getting the threads, using atomic semaphores to sync T1 and T2 was almost a line.
- P2_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 flaws that were eventually fixed by finding a deceptive way to pass integers through character arrays to solve the problem.
    - Since I couldn't find a way to write the 2D matrix to shared memory, I then mapped the entire 2D matrix in a 1D way and simulated it as a 2D matrix in the program.
- P2_3
    - IPC using Pipes
    - We initially tried to implement IPC using named FIFO pipes.
    - But attempting to achieve error-free communication seemed impossible. Switching to using simple conduits that can be used to communicate between a parent and child process.
    - parallelizing it by feeding each row from the first function grayScale() to the second function detectEdges() as it is done.

**XXX**