Assignment 4

Parikshit Gehlaut
220150009

Vemula Chandrahaas Reddy
220010062

Soumyadeep Das
220010056

October 21, 2024

1 Introduction

In our assignment we perform a series of operations to sharpen ppm images. These operations are divided into three main functions: `S1_smoothen`, `S2_find_details`, and `S3_sharpen`. Each function processes a different stage of the image transformation. We processed the input image 1000 times to increase the amount of work done.

We explore 4 different ways of doing the job-

1. part1 - without any parallelism, just using a single process.
2. part2_1 - parallelism using pipes to communicate between processes.
3. part2_2 - parallelism using shared memory to communicate between processes, synchronization is done using semaphores..
4. part2_3 - work done using 3 threads, synchronization is done using atomic operations.

2 Approach

To solve this problem, we implemented the following approach:

- **part1**
  - Input Image is read once.

- Three functions `S1_smoothen`, `S2_find_details`, and `S3_sharpen` called in order inside **for loop** for 1000 times to work on input image.
- Written sharpened_image once to output file after 1000 iterations.

- **part2_1**
  - Input Image is read once.
  - Created 3 processes using 2 fork functions, 3 processes handle independently `S1_smoothen`, `S2_find_details`, and `S3_sharpen` functions respectively.
  - Interprocess communication is done using pipes.
  - Each process work independently on input image and sends the processed image to next process for next stage for processing using respective pipes.
  - Synchronization is defaulty ensured by the way pipes function. We have used two pipes, one for communication between process1 and process 2 and other for communication between process2 and process3.
  - Data sent and read at the granularity of single pixel.
  - Integrity of images going in and out of the pipes was checked using crc32, an inbuilt $c++$ hasher that produces 32 byte hashes ensuring the hashes are small.
  - Written sharpened_image once to output file after 1000 iterations.

- **part2_2**
  - Input Image is read once.
  - Created 3 processes using 2 fork functions, 3 processes handle independently `S1_smoothen`, `S2_find_details`, and `S3_sharpen` functions respectively.
  - Interprocess communication is done using **shared memory**.
  - Each process work independently on input image and sends the processed image to next process for next stage for processing using respective shared memories.
  - Synchronization is ensured using **semaphores**. We have used two shared memories and two semaphores, one for synchronous communication between process1 and process 2 and other for synchronous communication between process2 and process3.

- Data sent and read at the granularity of whole image.
- Written sharpened_image once to output file after 1000 iterations.

- **part2_3**
  - Input image is read once.
  - Created 3 threads to run 3 functions `S1_smoothen`, `S2_find_details`, and `S3_sharpen` .
  - Using `.join()` function so that the main thread waits for all 3 threads to finish.
  - Used atomic flags for synchronisation, created 2 flags which ensure s2 doesn't make any progress till s1 is completed, and similarly s3 doesn't make any progress till s2 is completed.
  - Written sharpened_image once to output file after 1000 iterations.

3 Discussion

- **Ease of implementation**
  - **part1** was easiest to implement as we had to include only a for loop.
  - **part2_2** was relatively difficult, as it involved considerable logic and understanding of shared memory and semaphores.
- **Difficulties Faced**
  - **Memory Leaks** - which caused processes to run out of memory, which ultimately crashed the processes. Which was effectively addressed by managing memory allocation and deallocation appropriately.
- **Debugging approaches used**
  - memcheck from valgrind for checking memory leaks.
  - print statements.

4 Performance Comparison

To evaluate the performance of the different parts, we ran the programs with input image `1.ppm, 2.ppm, and 3.ppm`. The following table summarizes the time taken for all four parts:

	Time Taken (seconds)		
Part	**1.ppm**	**2.ppm**	**3.ppm**
`part1`	8.65777	34.9113	67.2437
`part2_1`	38.6699	148.388	294.637
`part2_2`	7.83946	31.8036	60.6763
`part2_3`	11.337	39.8578	74.4782

5 Conclusion

It is observed that shared memory using semaphore works much faster than memory passing using pipes and threading with atomic flags since shared memory involves less system calls as compared to memory passing, and works in parallel as compared to part1. Worst performance is observed for part2_1 that is for processes using pipes, this is due to system call on each read and write of pixel value to pipe.