Convolutional Layer Using Threads: Implementing Deadlock prevention and Mutex methods

Acey Dufresne
CS 3502 Operating Systems Section: 02
cdufres1@students.kennesaw.edu

Febuary 28, 2025

Introduction:

The purpose of the project is to deal with deadlock within a program. Deadlock needs four instances to occur, those that this program focuses on is mutual exclusion, having two threads/processes accessing a resource at the same time. The other is Circular Wait, where each process is waiting on a prior process finish its tasks before it may begin.

To accomplish this, a linux environment must be set up, for this project a window's subsystem WSL was used, with Ubuntu implemented as the developement environment. Then Visual Studio Code was connected to the Ubuntu environment, allowing Visual Studio code to directly access and modify the project files from within Ubuntu.

The approach to tackle the technical requirements that were neccessary is a Convolutional Layer for a incomplete Convolutional Neural Network. Our Convolutional layer takes an inputted file path, that leads to an image already in the project directory. The image is then passed through our partial Convolutional layer and split within a grid, where it is then processed and the important edges were saved to a new image.

In a completed model, the final image, or processed image would then be given to the FC, fully connected layer, so that the program could make a prediction as to what the edges it was given were.
To complete this program in a timely manner, I used a previously created Convolutional layer that I had created within the java language, using this as my guide I was able to quickly and efficiently complete a basic flow of how the program should operate.

Working with C language offered many difficulties, whereas in my prior program in java, the program created an array and filled it in with all RGB channel values from the image, however this cannot be done in C, instead the program must rely upon the STBI libraries, which is the most efficient method to dealing with image files.

The scope of this project would multiple threading, deadlock management, and setting up and implementing a Linux environment to run the program. Furthermore, the program must contain IPC, inter process communications, within this program the processes will communicate crucial data of the image among themselves. The threads will access shared memory resources and the processes will share each seperate matrice of the created grid into one shared resource file.

Implementation

Implementation Structure of the Program: As discussed in the brief introduction, the purpose of this program is to take in an image, and find the important edges, if this program were ever to be completed, these edges would then be passed to the next layer of processes and eventually the program would provide a prediction, as shown in figure 2, the Full CNN Model.

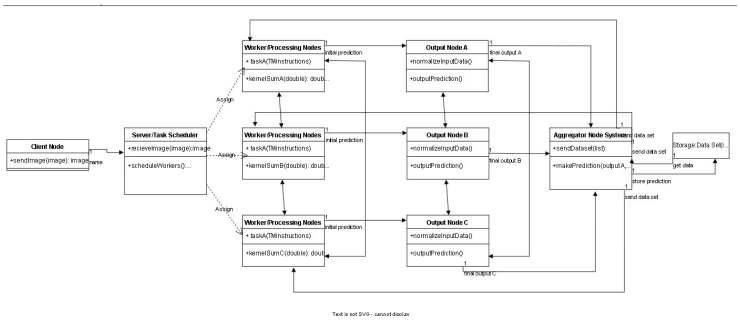

Figure 1: Fully Implemented Model.

The important processors nodes, displayed in the above figure as workers, are represented in our C program as threads, specifically p threads. To implement our threads we need a structure, to hold the valuable data and variables. These include standard information such as a reference to the memory location of our inputted image, the length and width of the image, as well as the threshold. The threshold is the important variable that determines which pixels are deemed important, for more accurate results the threshold may be manipulated to perform different outcomes. After our input image has been entered into the program, the main method passes it on to the toGrey function, within this method the program utilizes NVIDIAs formula for updating an images color values to grey scale. The program takes the three original color channels, RGB, and flattens them into one value, which is the final output from the grey scale method into the quadrant maker function.

The threadData structure also held the pos variable, for position. Within quadrantMaker, the original image, was split into a grid containing four quadrants. Each quadrant is handled by an individual thread. Each quadrant processor, or thread, scans over the single-color channel, and compares the value of the grey scale image to the provided threshold. Each thread updates the pos, variable in the threadData structure, and calls the Finished function. The finished method takes the pos variable, which is similar to an identification for each thread, and assembles the final output image by placing each thread out into the corresponding position of the final image. The final output image should contain the outline of our original inputted image.

Implementing the Threads: The threads were implemented using Cs POSIX library, using p threads, as mentioned in the structuring phase, a data structure contained the information the threads needed, and the quadrantMaker function initialized the threads and processes. The main issue encountered during the implementation of the threads was Circular Waiting.

As you can see in the diagram above, the mutex locks were implemented in such a

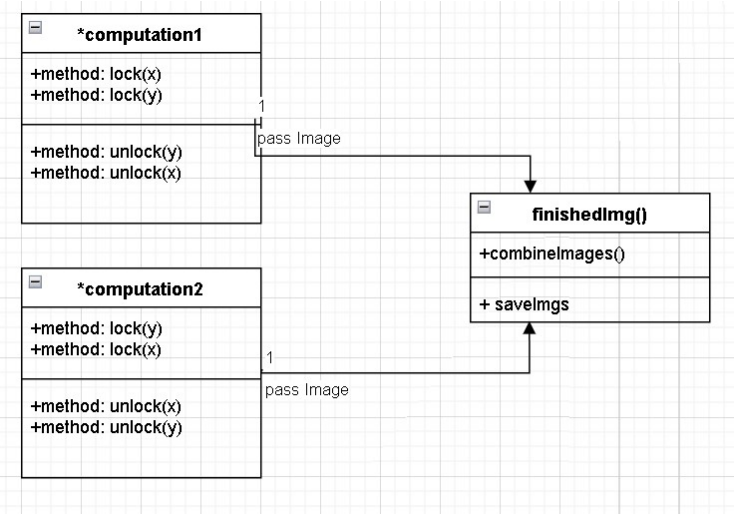

Figure 2: Illustration of the circular wait scenario.

way that they would lock and unlock in a dis-uniform manner, causing deadlock within the program. Admittedly, the program only ran into this issue a few times, and the only way to ensure deadlock would ensue, was to add a for loop that continuously locked and unlocked a threads access. To solve these issues, the mutex locks were adjusted so that they locked before a processs tasks were completed, and then unlocked at the end of the tasks, in a uniform manner. Such as, thread one lock(x) , lock(y), thread two lock(x), lock(y) ,rather than the prior implementation, thread one lock(x), lock(y), thread two lock(y), lock(x).

The only other issue encountered, which occurred when multiple threads attempted to modify the values in the threadData structure, which could result in Mutual Exclusion, as multiple processes try to access a resource at once. The solution was simple enough, by fixing the mutex locks, for the issue declared in the above paragraph, the Mutual Exclusion problem was also solved.

Another solution to these problems, which didn't work due to the guidelines of this assignment, was to remove the three additional threads, as all the threads used the same

segments of code, and create multiple instances of one thread.

IPC Implementation: IPC, inter process communication, was not needed for the program, but it was included. To implement, the program uses an un-named pipe, utilizing the pipe() function, with two file descriptors, one for writing and one for reading, this was stored in an integer array. We then use the fork() function, which creates a child process of the parent process the program initially created. This child process is an exact copy and contains both the read and write functions. The child process opens its read and write function, reads in the input integer from the user, and writes it to the array. The parent process opens read and write function, then closes the write function, as it only needs to read in the integer supplied to it.

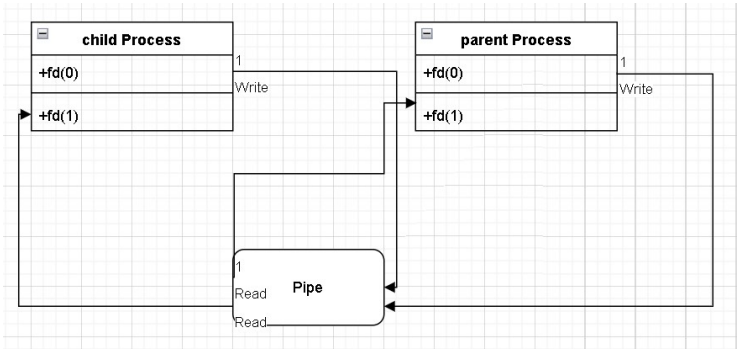

Figure 3: Illustration of the circular wait scenario.

As shown in the figure above, both the child and parent may write to the pipe, and both can access the data within the pipe.

1 Environment:

Developement Environement: For this assignment I opted for WSL, Windows Subsystem for Linux, as my environment, I decided on this because there were copious amounts of resources and tutorials on how to best utilize the system. I then implemented Ubuntu as my Linux Distribution, I chose this option because it was the first listed choice in the assignment guide, and like the latter, is open source and has plenty of resources to learn to operate it.

Both of these options had incredible ease of use, especially for a beginner. I then created a project directory within Ubuntu command terminal, and linked this to Visual Studio Code. Within Visual Studio Code, I installed the WSL extension, which allowed me to directly work within a familiar coding space, and Visual Studio Code would update the files in the directory within Ubuntu. To create the LaTeX pdf file, the LaTeX extension on Visual Studio Code was utilized. This extension makes coding a PDF document very efficient and simple, through my preperations for this assignment I found multiple templates that were premade for Report style documents. However, I opted to create my own, as this is my first time using LaTeX, I have found the process relatively straight forward, and have been writing out my segments in a word document and then transferring them into Visual Studio Code. This Visual Studio Code also provides an Output terminal, so debugging and trouble shooting is much simpler.

This co-system, allowed me to code directly in the C language, and have Ubuntu/WSL handle the Linux operations. I found a great resource was old/ancient forum posts and YouTube instructionals. As mentioned in the implementations sections, the POSIX library was the method in which the threads were implemented, and of cource C's mutex locks as well.

Another incredibly vital library was the STBI libraries, which I found on GIT, and those were fairly straight forward to getting them functional and operational. The STBI, Sean T. Barret Images, handled all the image operations, specifically loading in new images, image creation, and writing new images. The set up phase of the project did not arise any major set backs, or challenges.

Results and Outcomes: Though it took some time, I was able to cause deadlock within the program, as shown in the figure below: The image, while not very exciting show an empty terminal after, the intial user input, in contrast, observe figure five which shows the program running with DBG, where each thread is shown to have operated successfully, and the final print statement is printed into the terminal.

Another major issue was obtaining the final output, while the program did compile correctly and would run, the issue laid at the threads. Each thread would process one

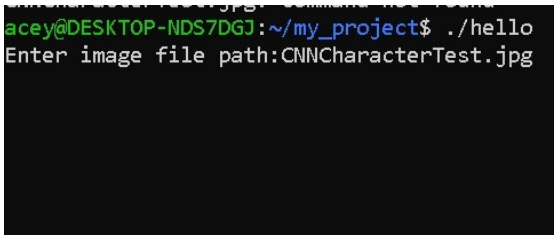

Figure 4: Illustration of the terminal when achieving Deadlock.

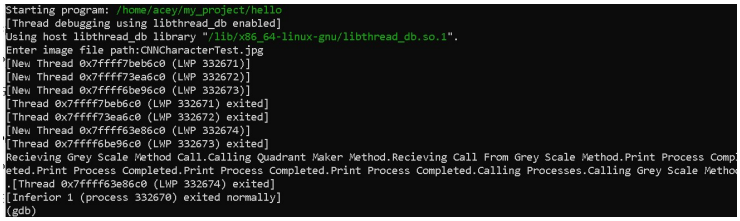

Figure 5: Illustration of how the program should compile.

quadrant, and then call the finished function. This caused multiple issues. The threads all called the finished function, meaning that when the first thread called the finished function, it would pass whatever quadrant it was working on, for example lets say the third quadrant, the finished function would then add that quadrant to the finalImage output. However, when the next thread called the finished function, passing it, another example the first quadrant, the finished function would create a new finalImage output, and the previous thread input would not be saved. This resulted in corrupted image files. As seen in figure six:

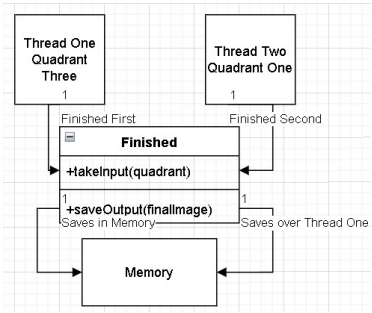

Figure 6: Illustration of how the threads worked.

To correct this issue, a new structure, final Image structure, was implemented to hold all the data of the quadrants. The structure could then be passed to the finished method, meaning that none of the threads would be directly accessing the finished

method, rather they would be updating structure at their own pace.

Results: The finished product is a program that can take a file path from a user, and access an image, that is already within the project directory. The program will take that input, create a new image object, then convert the input image to grey scale. It then passes the modified image to a matrice creator, which splits the image into four parts, or quadrants. The program then initializes four threads to deal with each quadrant. The threads process the inputted quadrant, compare each pixel to a given threshold value, and finally update the finalImage structure values and data. The finished method compiles all the segments back together, and stores the final image into the project directory, under the name: ProcessedImage.

Due to the difficulty of the project, the only rubric implemented was if the program could provide the correct output. In this case that would be a image with a white background, and all the edges that make up the outline of the object, would be printed upon this background in black. The project could use many adjustments and improvements, for example, a FC, fully connected, layer to actually classify the image, pulling from a data set for reference. Below is a complete flow of the program:

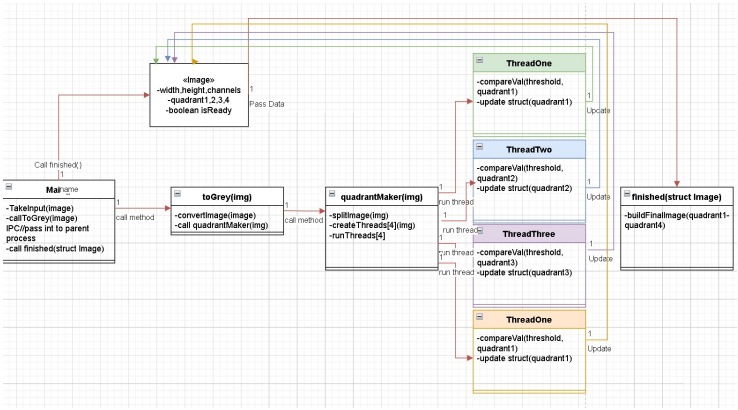

Figure 7: Illustration of the flow of the program.

2 References:

Kanich, Chris, “Everything You Should Know about Deadlock in Three Minutes or Less.” , YouTube, 2021, , , John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134

John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134
John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134

John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134 John Smith and Jane Doe, A Study on Deadlock Prevention, Journal of Operating Systems, 2025, 10, 2, 123–134