CSCI 4061: Introduction to Operating Systems, Fall 2023

Project #3: Parallel image processing

Instructor: Abhishek Chandra

Intermediate submission due: **11:59pm (CDT), Nov. 8, 2023**

Final submission due: **11:59pm (CDT), Novt. 15, 2023**

1. Background

Image rotation is the process of changing the orientation of an image by turning it around a central point. It involves altering the position of pixels in the image so that the entire content is shifted by a certain angle, typically in a clockwise or counterclockwise direction. Image rotation is used in many Photo Editing applications like Photoshop, Google Photos. It is also extensively used in the Animation industry. In this project, we will use multithreading to speed up the process of rotating multiple images. In multi-threaded programming, threads can perform the same or different roles. In some multithreading scenarios like processing-worker thread, processing and worker threads have different functionality. In other multithreading scenarios like many parallel algorithms, threads have almost the same functionality. In this programming assignment we will be using the processing-worker scenarios.

The purpose of this programming assignment is to get you started with thread programming and synchronization. You need to be familiar with POSIX threads, mutex locks and condition variables.

2. Project Overview

Your project will be composed of two types of threads: **Processing thread** and **Worker threads**.The purpose of the processing thread is to traverse the given directory for image files and place the directory contents into a shared queue (request queue). The purpose of the Worker threads is to monitor the request queue, retrieve requests files and read the requests files bytes, apply image rotation, and serve the rotated image back to the user. The queue is a bounded buffer and will need to be properly synchronized.

2.2 Thread Pool

Your program should create a fixed pool of worker threads when the program starts. The Worker thread pool size should be ***num_Worker*** (You can assume that the number of worker threads will be less than the number of requests) and processing thread should be a ***single thread***.

2.3 Request Queue Structure:

- **Request Queue Structure:** Each request inside the queue will contain a file name and rotation angle (180 or 270). You may use a struct to store those data before adding them to the queue. The queue structure is up to you. You can implement it as a queue of structs or a linked list of structs, or any other data structure you find suitable.

2.4 Processing Thread

The processing thread is responsible for traversing a specified directory, placing the content of this directory into a shared queue, and signaling to the worker threads once no more data will be placed in the queue. Here's a breakdown of its functionality:

- **Parameters:** The Processing thread will take the directory path, number of worker threads and an angle of rotation (180 - left to right flip) or 270 - upside down flip).
- **Directory Traversal:** The input directory is traversed to identify all image files with extension ".png".
- **Queue Management:** The identified image file paths are added to a request queue along with a desired angle of rotation(180/270). This queue is shared with the workers.
- **Signaling the Done condition:** Once all the image files are added to the request queue, the processing thread will signal to all the workers threads that no more new requests will be added to the queue.
- **Termination:** The processing thread will wait for an acknowledgement from all the worker threads once they have completed all the image rotations. The processing thread will verify if the number of image files passed into the queue is equal to the total number of images processed by the workers. A termination signal is broadcast to all workers and the processing thread will terminate.

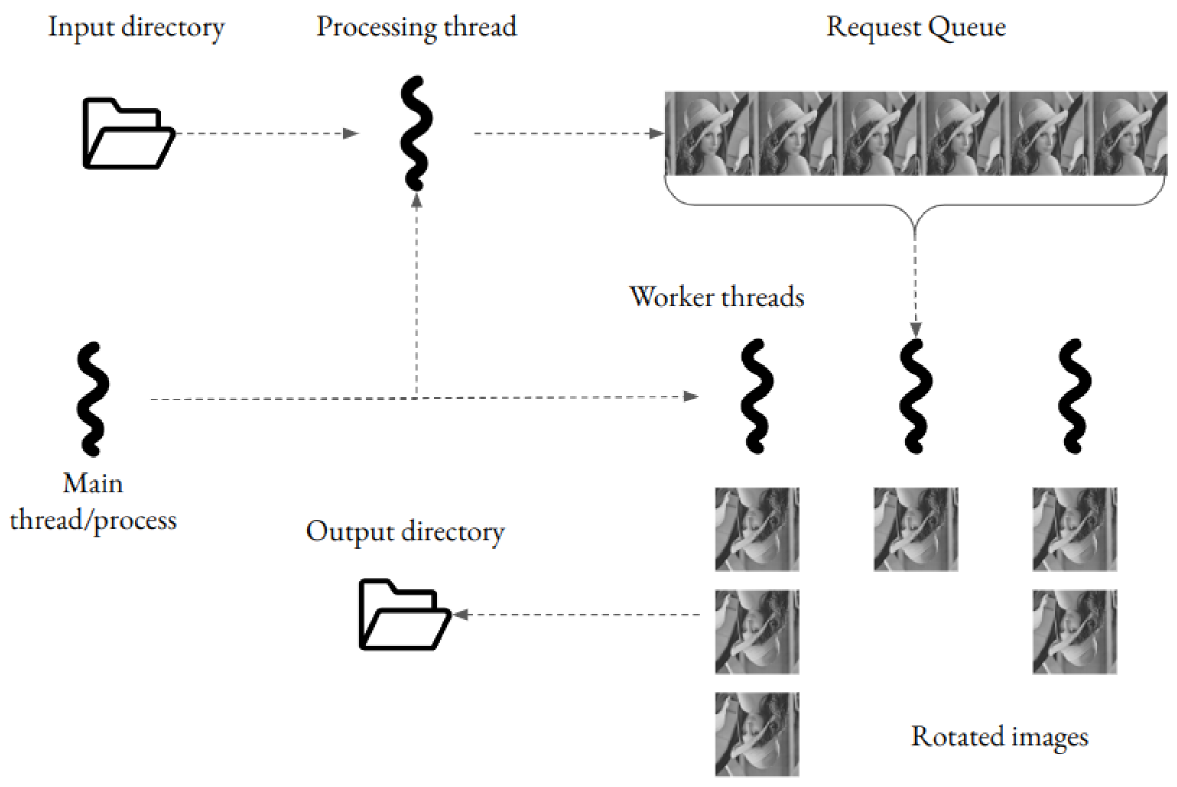
2.5 Worker Threads

The worker threads are responsible for monitoring the request queue, retrieving requests, reading the request files' bytes, applying image rotation, and serving the rotated images back to the user. Here's a breakdown of its functionality:

- **Parameters:** The worker thread will take the **threadID** as a parameter which will later be used for logging. You can assign the threads an ID in the order you created them. Note that this thread ID is different from the pthread_id assigned to the thread by the pthread_create() function.
- **Queue Monitoring:** Worker threads continuously monitor the shared request queue. When a new request arrives from the processing thread, one of the worker threads retrieves it for further processing.

- **File Preparation:** Once a request is obtained, a worker thread will read the image data in linear form using **stbi_load**. The linear form is then converted to 2D for image rotation.
- **Image Rotation:**. The 2D array is then flipped according to the provided angle. This involves calling one of the two provided functions **flip_left_to_right** or **flip_upside_down** and store its return value correctly.
- **User Response:** After rotating the image, the worker thread prepares the rotated image to be served back to the user by saving it into a new file by calling **stbi_write_png.**
- **Synchronization:** Proper synchronization mechanisms such as mutexes and condition variables are used to ensure that multiple worker threads can safely access and modify shared data structures (queues) and other global variables without race conditions or deadlocks.

Figure 1 Project Overview:



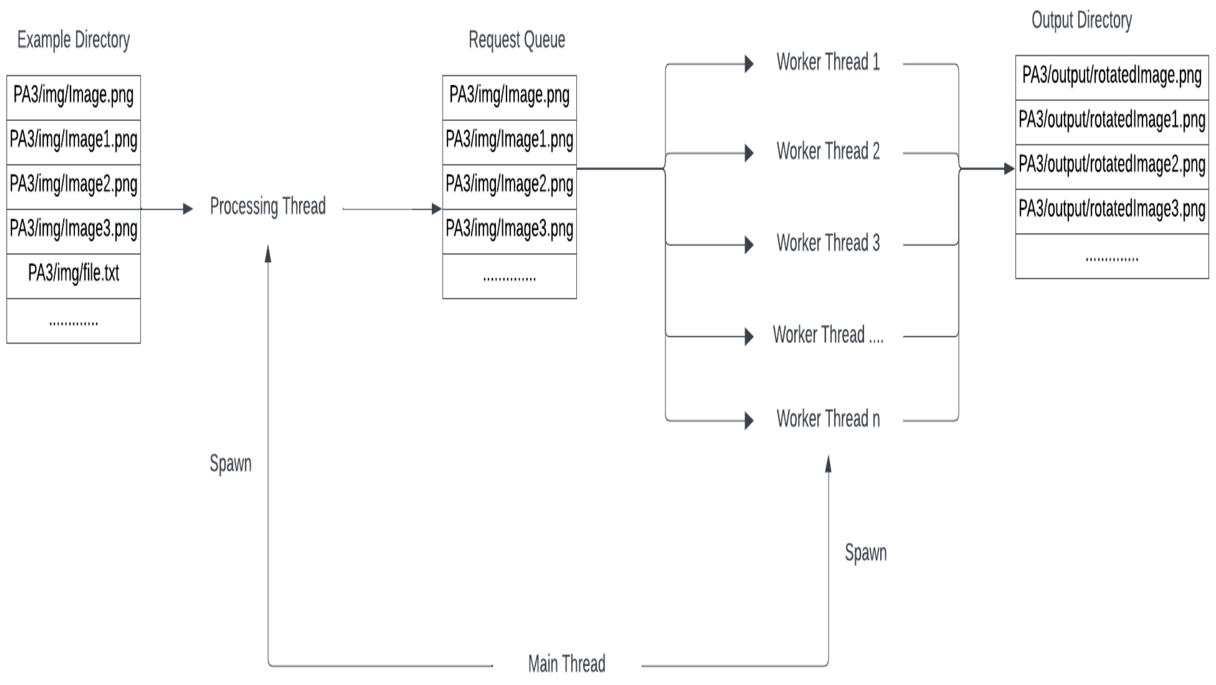


Figure 1: Shows the structure of the project. The main thread spawns a set of worker threads and the processing thread and joins them until they are done. Refer to section 2.3 and 2.4 for the functionality of each thread.

2.5 Request Logging

The worker threads must carefully log each request (normal or error-related) to a file called "request_log" and also to the terminal (stdout) in the format below. The log file should be created in the same directory where the final executable "image_rotation" exists. You must also protect the log file from race conditions. The format is:

[threadID][reqNum][Request string]

- **threadID** is an integer from 0 to num_workers -1 indicating the thread index of request handling worker. (Note: this is **not** the pthread_t returned by pthread_create).
- **reqNum** is the total number of requests a specific worker thread has handled so far, including the current request.
- **Request string** is the path of the image file that the worker thread is handling

The log (in the "request_log" file and in the terminal) should look something like the example below.

```
[8][5][./img/30.png]
[9][5][./img/30.png]
[2][5][./img/282.png]
```

2.6 Handling Worker Termination with a Condition Variable(Section 16.6 of book):

Worker threads will use a condition variable to efficiently handle termination when the request queue becomes empty. Here's how this can be accomplished:

- **Completion Tracking:** Worker threads continue to track the number of requests they have successfully processed using a thread-specific counter(used for request logging).
- **Worker Termination Condition:** Each worker thread will have access to a condition variable and a mutex associated with it. Worker threads periodically check the status of the request queue while holding the mutex. The processing thread will wait for an acknowledgement from all the worker threads once they have completed all the image rotations. The processing thread will verify if the number of image files passed into the queue is equal to the total number of images processed by the workers. A termination signal is broadcast to all workers and the processing thread will terminate.
- **Main Thread:** The main thread waits for the termination of worker and processing threads using ***pthread_join***.
- **Termination:** Once all worker threads and the processing thread have terminated, the main thread can proceed with any necessary memory cleanup.

2.7 Program Analysis (Extra Credit):

Students are encouraged to experiment with different numbers of threads in the multi-threaded image rotation program. They are tasked with evaluating how varying the number of threads affects program performance. Specifically, students will investigate whether increasing or decreasing the number of threads leads to performance improvements or deteriorations.

Experimental Setup:

- **Thread Count Variation:** Students will modify the number of worker threads in the program, allowing for a range of thread counts to be tested. They can experiment with different thread counts, such as increasing or decreasing the number of worker threads, while keeping other parameters constant.
- **Measurement Metrics:** Students will evaluate program performance using relevant metrics, such as execution time, CPU utilization, and memory usage, CPU time, etc…. These metrics will serve as objective measures of program efficiency.

Documentation and Reporting:

Students will be expected to document their experiments, results, and analyses in a report. The report should include performance data, graphs, and clear explanations of the observed outcomes. Students should draw conclusions about the impact of thread count on program performance and provide practical recommendations for optimizing the multi-threaded image rotation program in different scenarios.

4. Compilation Instructions

You can create all of the necessary executable files with

Command Line

```
$ make
```

Running the program with various directories can be accomplished with

Command Line

```
$ ./image_roatation input_dir output_dir number_threads
Rotation_angle
```

5. Project Folder Structure

Please strictly conform to the folder structure that is provided to you. **Your conformance will be graded.**

Project structure	**Contents (initial/required contents[1])**
include/	.h header files (utils.h, image_rotation.h)
lib/	.o library files (utils.o)
src/	.c source files (image_rotation.c)
img	Contain multiple Directories that contain different images
expected/	expected output(only 180 rotation angle)
Makefile	file containing build information and used for testing/compiling
README.md	file containing info outlined in 8. Submission Details

[1] This content is required at minimum, but adding additional content is OK as long as it doesn't break the existing code.

6. Assumptions / Notes

1. You can assume that the number of worker threads will be less than the number of image files/requests.

7. Submission Details

There will be two submission periods. The intermediate submission is due 1 week before the final submission deadline. The first submission is mainly intended to make sure you are on pace to finish the project on time. The final submission is due ~2 weeks after the project is released.

7.1 Intermediate Submission

For the Intermediate submission, your task is to implement directory traversal and add the directory content into a request queue. You will also create N number of worker threads and join on them in the main thread, witheach thread printing its thread ID and exiting. **Note that you do not need to implement any synchronization for the intermediate submission.**

One student from each group should upload a **.zip file** to Gradescope containing all of your project files. We'll be primarily focusing on *.c and your README, which should contain the following information:

- Project group number
- Group member names and x500s
- The name of the CSELabs computer that you tested your code on
  - e.g. csel-kh1250-01.cselabs.umn.edu
- Any changes you made to the Makefile or existing files that would affect grading
- Plan outlining individual contributions for each member of your group
- Plan on how you are going to construct the worker threads and how you will make use of mutex locks and unlock and Conditions variables.

The member of the group who uploads the .zip file to Gradescope should add the other members to their group after submitting. Only one member in a group should upload.

7.2 Final Submission

One student from each group should upload a **.zip file** to Gradescope containing all of the project files. The README should include the following details:

- Project group number
- Group member names and x500s
- The name of the CSELabs computer that you tested your code on

    - e.g. csel-kh1250-01.cselabs.umn.edu
- Members' individual contributions
- Any changes you made to the Makefile or existing files that would affect grading
- Any assumptions that you made that weren't outlined in section 7
- How you designed your program for Parallel image processing (again, high-level pseudocode would be acceptable/preferred for this part)
    - If your original design (intermediate submission) was different, explain how it changed
- Any code that you used AI helper tools to write with a clear justification and explanation of the code (Please see below for the AI tools acceptable use policy)
- Program Analysis Document(**optional**)

The member of the group who uploads the .zip file to Gradescope should add the other members to their group after submitting. Only one member in a group should upload.

Your project folder should include all of the folders that were in the original template. You can add additional files to those folders and edit the Makefile, **but make sure everything still works**. Before submitting your final project, run "make clean" to remove any existing output/ data and manually remove any erroneous files.

8. Reiteration of AI Tool Policy

Artificial intelligence (AI) language models, such as ChatGPT, may be used in a **limited manner for programming assignments** with appropriate attribution and citation. For programming assignments, you may use AI tools to help with some basic helper function code (similar to library functions). You must not use these tools to design your solution, or to generate a significant part of your assignment code. You must design and implement the code yourself. You must clearly identify parts of the code that you used AI tools for, providing a justification for doing so. You must understand such code and be prepared to explain its functionality. Note that the goal of these assignments is to provide you with a deeper and hands-on understanding of the concepts you learn in the class, and not merely to produce "something that works".

If you are in doubt as to whether you are using AI language models appropriately in this course, I encourage you to discuss your situation with me. Examples of citing AI language models are available at: libguides.umn.edu/chatgpt. You are responsible for fact checking statements and correctness of code composed by AI language models.

For this assignment: The use of AI tools for generating code related to the primary concepts being applied in the assignment, such as thread management and synchronization, **is prohibited**.

9. Rubric (tentative)

- [10%] README
- [10%] Intermediate submission
- [20%] Coding style: indentations, readability, comments where appropriate
- [30%] Test cases
- [20%] Correct use of pthread_create(), pthread_join(), pthread_mutex_t , pthread_cond_t, pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), pthread_cond_signal()
- [10%] Error handling — should handle system call errors and terminate gracefully
- [10%] Extra credit - Program Analysis

Additional notes:

- We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and runs on CSELabs.
- A list of CSELabs machines can be found at https://cse.umn.edu/cseit/classrooms-labs
  - Try to stick with the Keller Hall computers since those are what we'll use to test your code
- Helpful GDB manual. From Huang: GDB Tutorial From Kauffman: Quick Guide to gdb