Lab 2: Interprocess Communications with Pipes and Java Threads

Liam Geraghty - 300356748
Shane Stock - 300351190
CSI3131 - Operating Systems

June 8, 2025

1 Introduction and Objectives

The objective of this lab is to explore interprocess communication (IPC) using UNIX/Linux pipes and to learn about Java threads and thread pools. The specific goals include:

- Understanding IPC mechanisms through the use of pipes in a C program.
- Gaining experience with process management in a Linux environment.
- Learning how to implement and manage threads in Java.
- Comparing the performance of individual threads versus a thread pool.

2 Methodology

The lab was conducted in a Linux virtual machine environment. The following steps were taken to achieve the objectives:

1. **Modify C Program**: Enhanced the `mon.c` program to create `mon2.c` for monitoring processes using pipes.
2. **Compilation**: Compiled the modified `mon2.c` program using gcc.
3. **Execution**: Ran the `mon2` program with the `calcloop` argument and observed the filtered output.
4. **Signal Handling**: Experimented with sending `SIGSTOP` and `SIGCONT` signals to the `calcloop` process.
5. **Java Setup**: Compiled the provided Java files for generating the Mandelbrot set.
6. **Execution of Mandelbrot**: Executed the `MandelBrot` application with various parameters.
7. **Thread Implementation**: Modified the Java code to use threads for rendering the Mandelbrot set.
8. **Thread Pool Implementation**: Further modified the code to utilize a thread pool with Executors.

3 Presentation and Analysis of Results

3.1 Part A Interprocess Communication (C / Pipes)

mon2.c Implementation

To implement interprocess communication using pipes, we modified the original `mon.c` into `mon2.c`. The key changes included:

1. Creating a pipe to connect the output of `procmon` to the input of `filter`.
2. Forking a child process for each of the following: the target program (`calcloop`), `procmon`, and `filter`.
3. Using `dup2()` to redirect standard output/input through the pipe.
4. Implementing a cleanup routine using `kill()` and `sleep()` to terminate the processes after 20 seconds.

Relevant Code Excerpt:

Listing 1: Excerpt from mon2.c showing pipe and process setup

```c
if (pipe(fd) == -1) {
    fprintf(stderr, "Pipe failed");
    return -1;
}

procmon_pid = fork();
if (procmon_pid == 0) {
    close(fd[0]);
    dup2(fd[1], STDOUT_FILENO);
    execl("./procmon", "procmon", pid_str, NULL);
}

filter_pid = fork();
if (filter_pid == 0) {
    close(fd[1]);
    dup2(fd[0], STDIN_FILENO);
    execl("./filter", "filter", NULL);
}
```

Signal Handling

We sent `SIGSTOP` and `SIGCONT` to the `calcloop` process using the `kill` command. This allowed us to observe state transitions in the filtered output (see Figure 4).

3.2 Part B Java Threads and Thread Pools

1. Compilation of Provided Java Files

The Java Mandelbrot source files were compiled successfully from the terminal (see Figure 5).

2. Running MandelBrot with Various Parameters

We tested the Mandelbrot renderer with multiple coordinate and zoom combinations to visualize different regions of the set (see Figures 6 and 7).

3. Using Java Threads for Rendering (15 pts)

To improve rendering performance, we modified the Mandelbrot application to use Java Threads. When a rectangle is small enough (`minBoxSize`), a thread is created to fill it. Larger rectangles are divided into four quadrants, each processed in its own thread.

Listing 2: Using Java Threads to fill Mandelbrot rectangles

```java
mbp = new MBPaint(this, mg, mrect);
Thread thread = new Thread(mbp);
thread.start();
thread.join(); // Wait for completion
```

Listing 3: Recursive quadrant threading

```java
Thread thread1 = new Thread(() -> findRectangles(rect1));
...
thread1.start(); thread1.join();
```

The results show visible speedup and parallelism (see Figure 8).

4. Using a Thread Pool with Executors

We replaced raw threads with a fixed-size thread pool using Java's `Executors` API. This reduces thread creation overhead and improves performance consistency.

Listing 4: Creating the thread pool

```java
threadPool = Executors.newFixedThreadPool(mg.threadPoolSize);
```

Listing 5: Submitting tasks to the thread pool

```java
threadPool.execute(mbp);
```

Listing 6: Fallback in case of pool shutdown

```java
if (threadPool.isShutdown()) {
    mbp.run();
}
```

Thread pool rendering is more stable and efficient (see Figure 9).

5. Performance Comparison

Sequential (No Threads):

- Rendering is slow and visibly linear.
- Only one region is filled at a time.

Java Threads:

- Rendering is significantly faster.
- High thread count can lead to overhead and CPU contention.
- Threads are created and destroyed frequently.

Thread Pool:

- Best balance of performance and resource usage.
- Fewer threads reused for all tasks.
- More consistent performance across different runs.

Conclusion: Thread pools provide the most scalable and efficient rendering, especially with small `minBoxSize` values.

4 Discussion and Conclusion

This lab provided hands-on experience with both C-level interprocess communication and Java concurrency. Key takeaways include:

- The power and flexibility of UNIX pipes for IPC.
- The importance of managing threads and resources efficiently.
- Visual feedback from the Mandelbrot application made concurrency concepts more tangible.

Challenges Encountered

We expected that sending the `SIGCONT` signal would resume the process, but the output of `mon2` did not reflect this as expected.

Appendix: Team Information

- Liam Geraghty 300356748 Completed Part A (C / IPC)
- Shane Stock 300351190 Completed Part B (Java)

Appendix: Figures

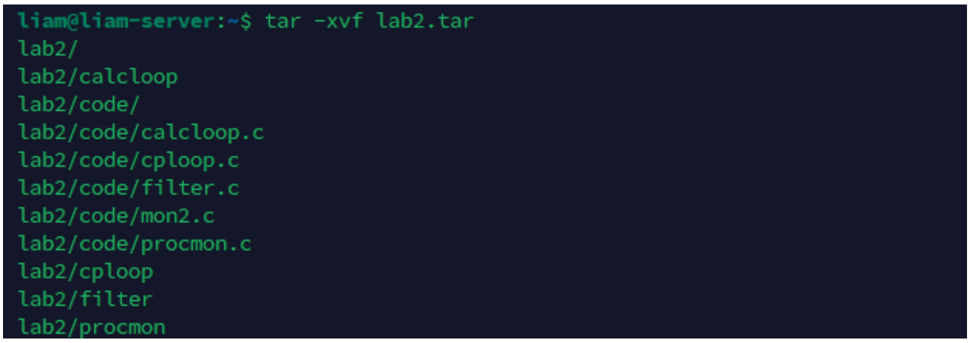

Figure 1: Extraction of `lab2a.tar`

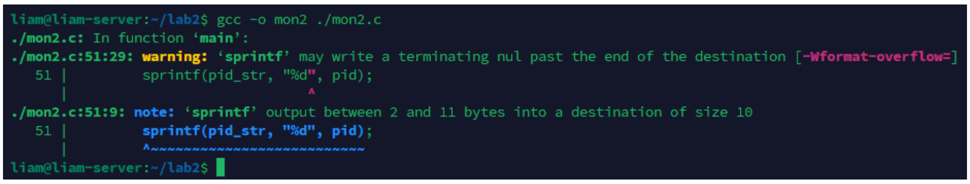

Figure 2: Compilation of `mon2.c`

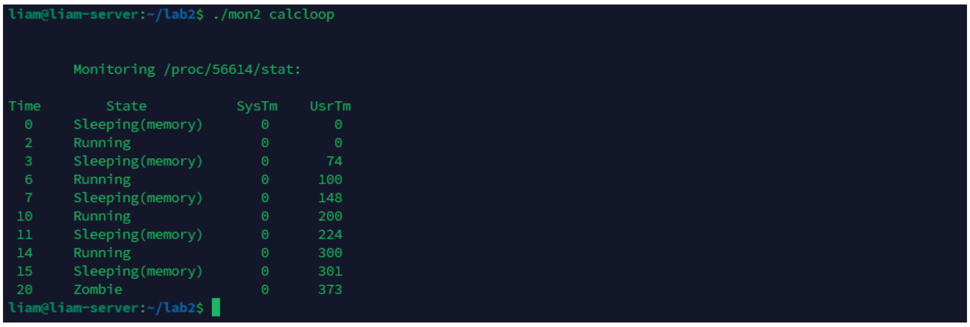

Figure 3: Execution of `mon2 calcloop` showing filtered output

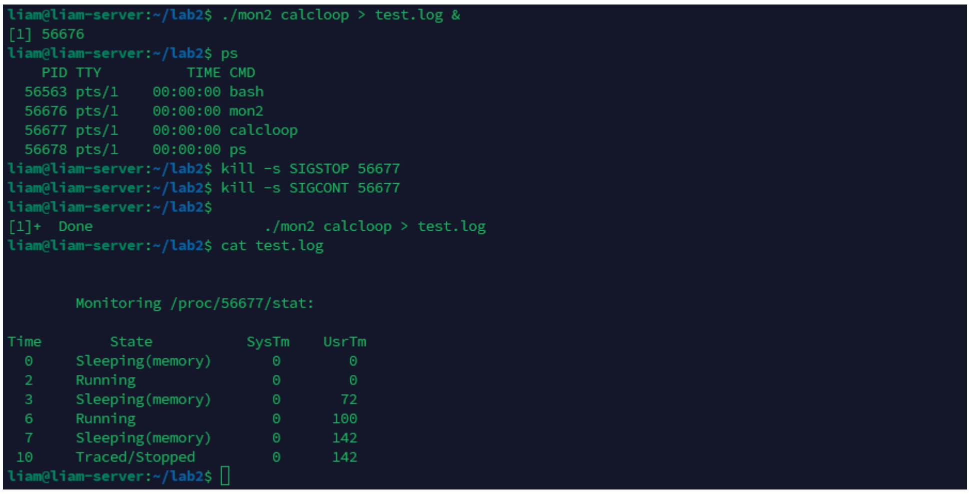

Figure 4: Sending signals to `calcloop` and observing output

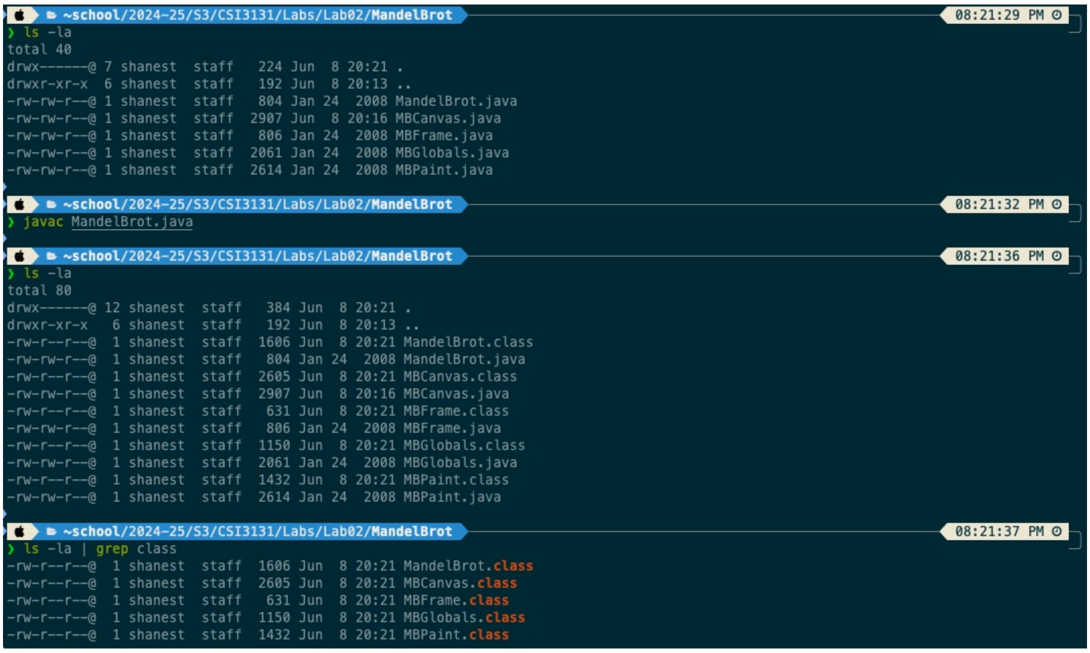

Figure 5: Successful compilation of Java Mandelbrot application

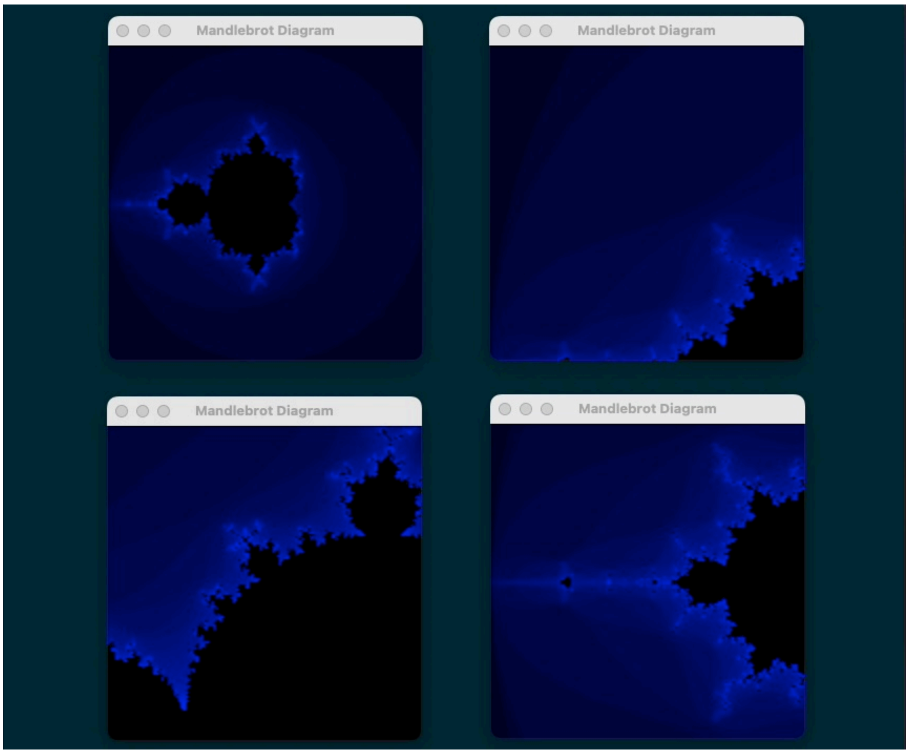

Figure 6: Four different Mandelbrot views using different parameters

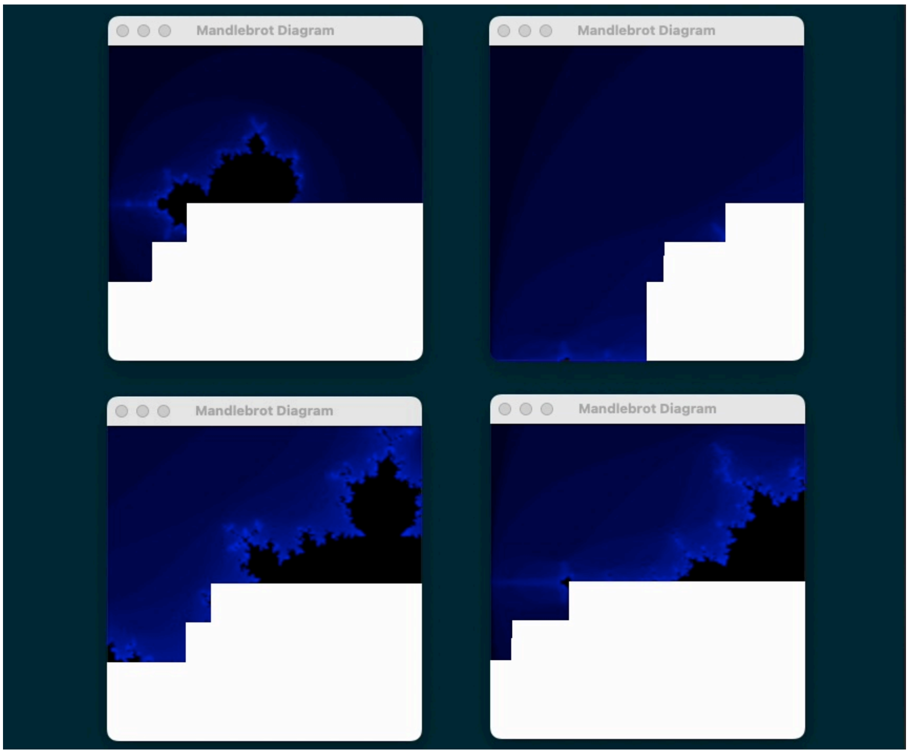

Figure 7: Initial rendering behavior without threading (sequential fill)

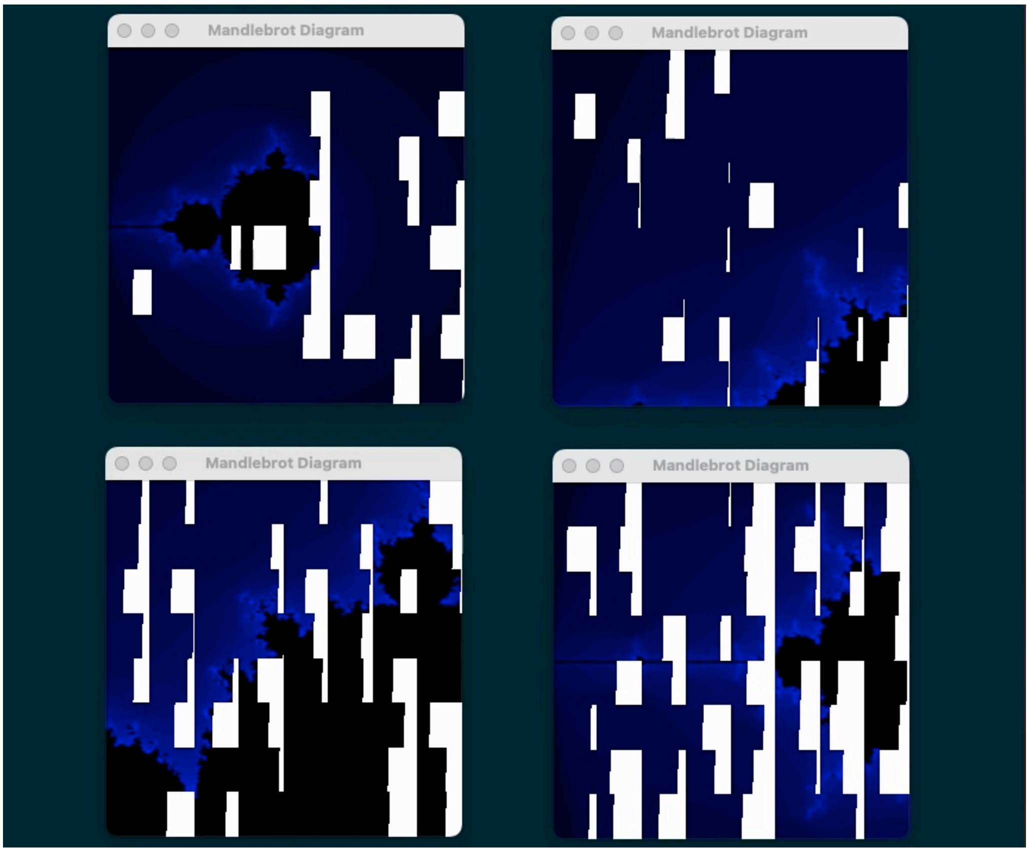

Figure 8: Rendering with Java Threads faster and more parallel

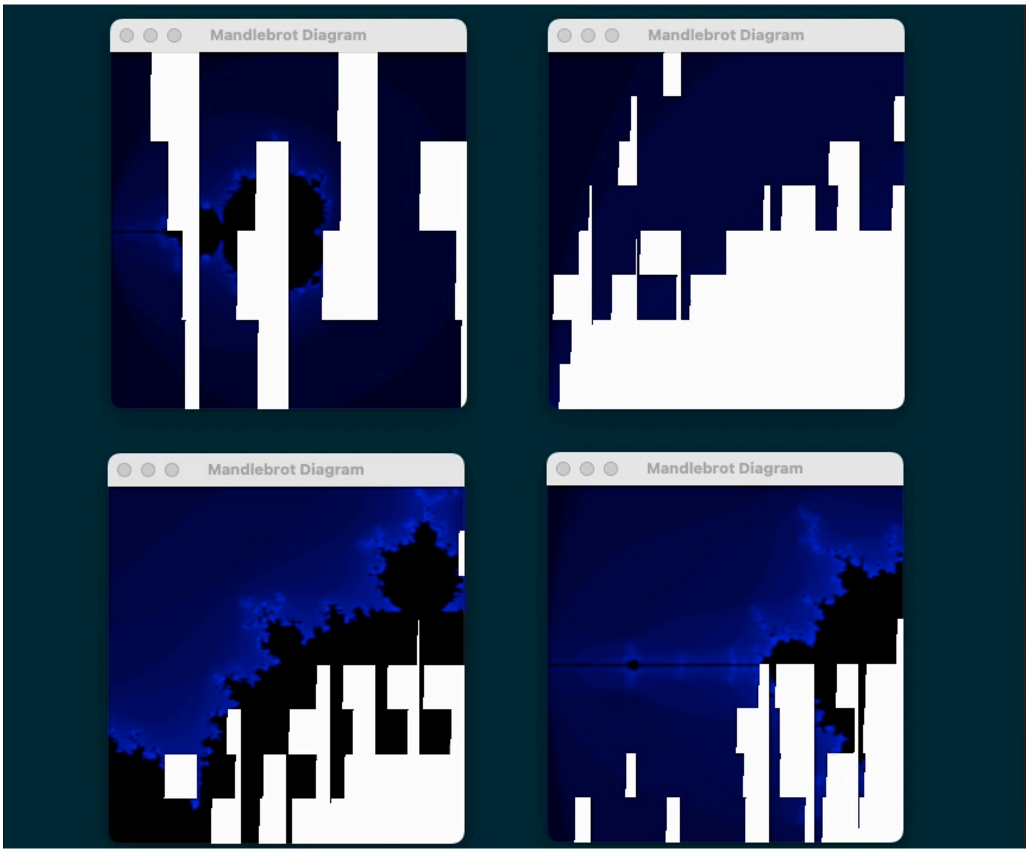

Figure 9: Rendering with Thread Pool stable and efficient