UC Student ID: 005035101

GitHub Link: https://github.com/vtadigotla35101/MSCS-632_Assignment6

This report examines the implementation of a parallel data processing system using Java and Go. While both languages are designed to handle multiple tasks simultaneously, they represent two fundamentally different philosophies of concurrency: Shared Memory (Java) versus Message Passing (Go).

1. Concurrency Models

Java utilizes a traditional threading model where multiple threads exist within the same memory space. To prevent race conditions where two threads attempt to modify the same data at once.

- Synchronization: In the Data Processing System, we used ReentrantLock and synchronized blocks. These act as gates allowing only one thread to enter a critical section of code like retrieving a task from the queue at a time.
- Thread Management: Rather than manually creating threads, we used the ExecutorService. This manages a Thread Pool which is more efficient as it reuses existing threads instead of constantly destroying and recreating them, reducing overhead for the Operating System.

Go: It uses Goroutines, which are much lighter than Java threads, allowing thousands to run on a single machine.

- Channels: Instead of a locked queue, Go uses Channels. A channel is like a pipe; one goroutine pushes data in, and another pulls it out. The synchronization is built-in and if a channel is empty then the worker automatically waits without needing an explicit lock.
- WaitGroups: Since Go doesn't have a built-in thread pool manager like Java's Executor, we used sync.WaitGroup to track when all goroutines have finished their work before allowing the program to exit.

2. Exception and Error Handling

Java: The Try-Catch ParadigmJava uses a formal Exception Handling hierarchy. When something goes wrong, like a file not being found then the program throws an exception that stops normal execution until it is caught.

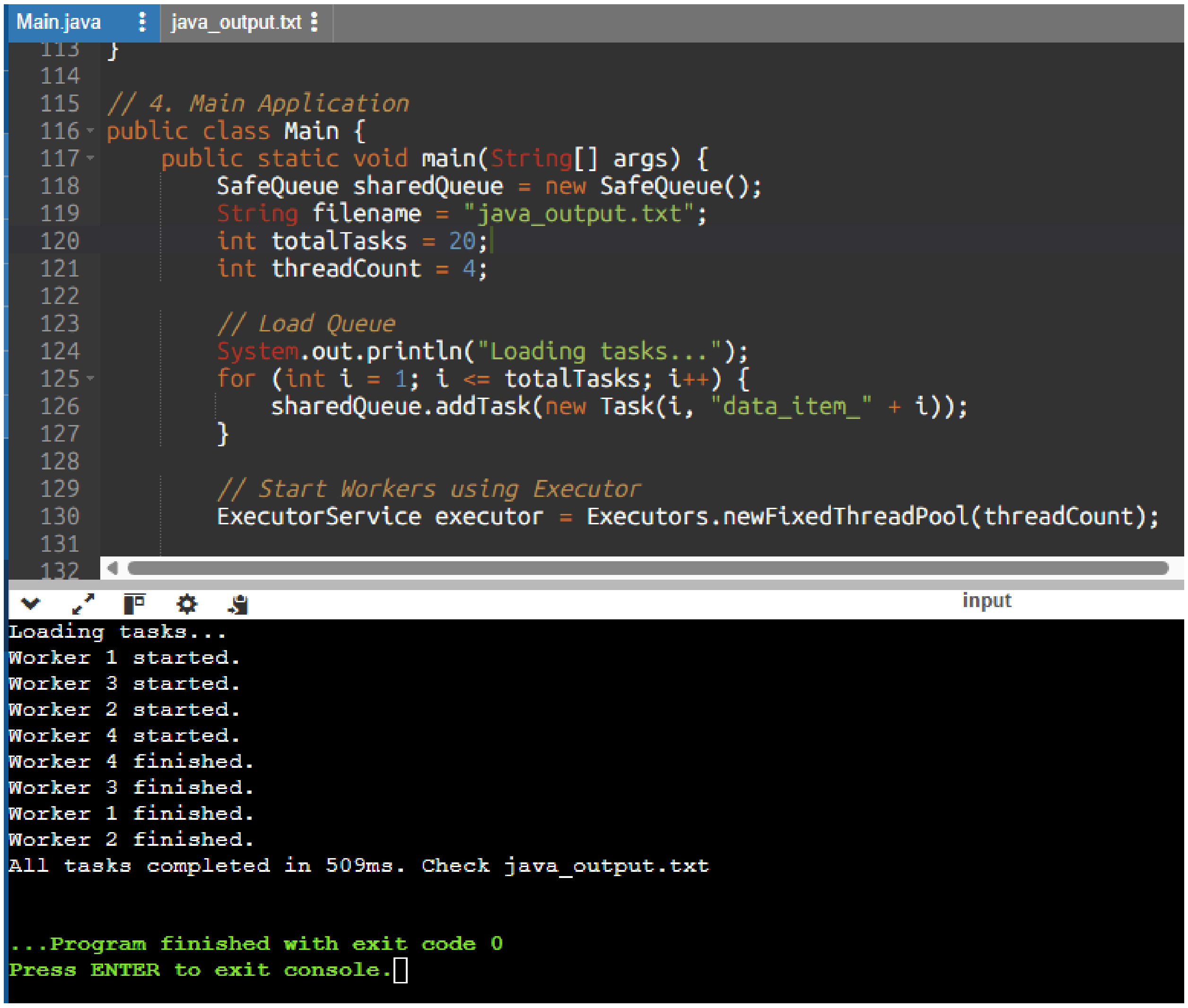
Go: Explicit Error Checking Go does not have traditional exceptions. Instead, functions that can fail return an error as their last return value.

Key Differences in Implementation

1. Queue Access:
   - Java: Uses ReentrantLock inside the SafeQueue class. Threads must explicitly lock and unlock to check the queue safely.

  - Go: Uses a chan Task. The channel *is* the queue. It is inherently thread-safe; we just push to it and pull from it.
2. Stopping the System:
    - Java: The workers pull until getTask() returns null. The ExecutorService handles the lifecycle of the threads.
    - Go: close(jobs) channel. The range loop in the workers automatically detects the closed channel and exits the loop.
3. Writing Results:
    - Java: Workers write directly to the file using a synchronized block to prevent garbled text.
    - Go: Workers send results to a results channel. A single dedicated goroutine picks them up and writes them. This avoids locking the file entirely.

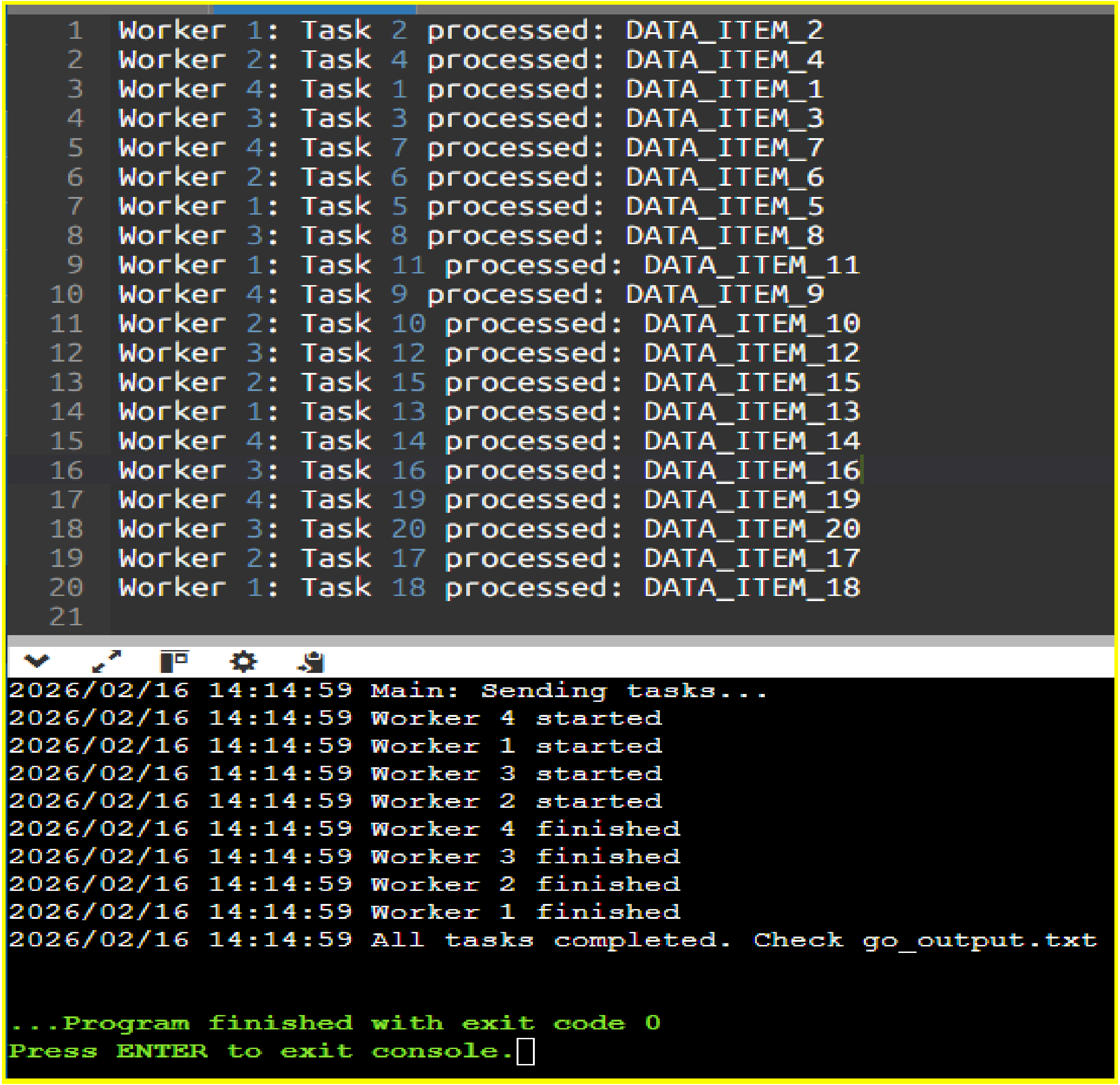
Execution Screenshot in JAVA Language:



```java
}

// 4. Main Application
public class Main {
    public static void main(String[] args) {
        SafeQueue sharedQueue = new SafeQueue();
        String filename = "java_output.txt";
        int totalTasks = 20;
        int threadCount = 4;

        // Load Queue
        System.out.println("Loading tasks...");
        for (int i = 1; i <= totalTasks; i++) {
            sharedQueue.addTask(new Task(i, "data_item_" + i));
        }

        // Start Workers using Executor
        ExecutorService executor = Executors.newFixedThreadPool(threadCount);
```

```
Loading tasks...
Worker 1 started.
Worker 3 started.
Worker 2 started.
Worker 4 started.
Worker 4 finished.
Worker 3 finished.
Worker 1 finished.
Worker 2 finished.
All tasks completed in 509ms. Check java_output.txt


...Program finished with exit code 0
Press ENTER to exit console.
```

Execution Screenshot in GO Language:



```
Worker 1: Task 2 processed: DATA_ITEM_2
Worker 2: Task 4 processed: DATA_ITEM_4
Worker 4: Task 1 processed: DATA_ITEM_1
Worker 3: Task 3 processed: DATA_ITEM_3
Worker 4: Task 7 processed: DATA_ITEM_7
Worker 2: Task 6 processed: DATA_ITEM_6
Worker 1: Task 5 processed: DATA_ITEM_5
Worker 3: Task 8 processed: DATA_ITEM_8
Worker 1: Task 11 processed: DATA_ITEM_11
Worker 4: Task 9 processed: DATA_ITEM_9
Worker 2: Task 10 processed: DATA_ITEM_10
Worker 3: Task 12 processed: DATA_ITEM_12
Worker 2: Task 15 processed: DATA_ITEM_15
Worker 1: Task 13 processed: DATA_ITEM_13
Worker 4: Task 14 processed: DATA_ITEM_14
Worker 3: Task 16 processed: DATA_ITEM_16
Worker 4: Task 19 processed: DATA_ITEM_19
Worker 3: Task 20 processed: DATA_ITEM_20
Worker 2: Task 17 processed: DATA_ITEM_17
Worker 1: Task 18 processed: DATA_ITEM_18
```

```
2026/02/16 14:14:59 Main: Sending tasks...
2026/02/16 14:14:59 Worker 4 started
2026/02/16 14:14:59 Worker 1 started
2026/02/16 14:14:59 Worker 3 started
2026/02/16 14:14:59 Worker 2 started
2026/02/16 14:14:59 Worker 4 finished
2026/02/16 14:14:59 Worker 3 finished
2026/02/16 14:14:59 Worker 2 finished
2026/02/16 14:14:59 Worker 1 finished
2026/02/16 14:14:59 All tasks completed. Check go_output.txt


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion

Java’s model is powerful for complex, object-oriented systems where fine-grained control over memory is required. However, it requires careful management to avoid deadlocks. Go’s model is often considered safer and simpler for high-volume data processing because its channel system naturally prevents many common concurrency bugs by design.