Multi-threaded Data Processing System

Date: April 20, 2025

- **Term:** Spring 2025 Second Bi-term
- **Class:** Advanced Programming Languages -MSCS-632-B01
- **Instructor:** Dr. Dax Bradley
- **University of the Cumberlands**

The GitHub repository, https://github.com/baralsamrat/MSCS632_Assignment_6

REPORT : - **PDF**

Screenshots

Capture-1.PNG 1

Capture-2.PNG 2

Capture-3.PNG 3

Overview

This project implements a **multi-threaded data processing system** simulating real-world parallel work, using Go's concurrency model with goroutines and channels. The system demonstrates worker thread (goroutine) management, task queue synchronization, safe shared resource access, and error handling. Data Processing System that simulates multiple worker threads processing data in parallel. The system will consist of multiple worker threads that retrieve data from a shared queue, process it, and save results to a shared resource.

The simulated task represents **image processing** where each worker processes a file (with an artificial delay) and logs results to an output file (`output.log`).

Concurrency ensures that multiple images are processed simultaneously, reducing overall processing time compared to sequential execution.

Differences Between Go and Java Concurrency Models

Aspect	Go (Goroutines + Channels)	Java (Threads + Executors)
Lightweight Threads	Goroutines are extremely lightweight	Threads are heavier (higher memory cost)
Communication Model	Communicate via **channels**	Communicate via shared data structures
Error Handling	Explicit error returns + defer	Try-catch exception handling

Aspect	Go (Goroutines + Channels)	Java (Threads + Executors)
Synchronization	Channels inherently synchronize access	Need `synchronized` blocks or concurrent collections
Thread Management	Goroutines managed by Go runtime	Executors manage thread pools manually

Summary: Go's model favors simplicity and safety via channels and goroutines. Java provides more manual control, but requires more boilerplate for thread safety and exception handling.

How to Run

Prerequisites

- Go installed (version 1.18 or higher recommended)

Steps

1. Give execute permission to the `run.sh` script:

   ```
   chmod +x run.sh
   ```

2. Run the program:

   ```
   ./run.sh
   ```

   or specify the number of workers:

   ```
   ./run.sh 5
   ```

- Logs will be written to `output.log`.

Concurrency and Exception Handling

Go Implementation

- **Concurrency Model:**
  Go uses **goroutines** (lightweight threads) and **channels** to coordinate work among multiple workers. A **task channel** is used to safely distribute tasks without race conditions.

- **Synchronization:**
  Channels in Go inherently synchronize access. Workers block while waiting for tasks from the channel,

ensuring no two workers process the same task. A `sync.WaitGroup` tracks when all workers have completed.

- **Error Handling:**
  Go relies on **explicit error returns** from functions. This system checks errors when opening the log file using standard Go patterns (`if err != nil { ... }`). Also, `defer` ensures the log file is properly closed even if the program encounters a failure.

- **Logging:**
  Go's built-in `log` package is used, which is safe for concurrent use by multiple goroutines. Logs include the worker ID, file processed, and time taken.

Java Implementation

In Java, the system could be implemented using an `ExecutorService` (thread pool) to manage worker threads and a `BlockingQueue` to safely share tasks.

- **Concurrency Model:**
  Java uses **threads** or higher-level constructs like **Executors**. A `BlockingQueue` ensures safe task sharing across threads.

- **Synchronization:**
  Threads retrieving tasks from a `BlockingQueue` do not require explicit locking; the queue handles blocking behavior internally.

- **Exception Handling:**
  Java uses **try-catch blocks** to manage exceptions like `InterruptedException` or `IOException`. Proper exception handling ensures that threads terminate gracefully without resource leaks.

- **Logging:**
  A thread-safe logger such as Java's `Logger` or Apache Log4j would ensure atomic writes to a log file without requiring explicit synchronization.

References

- Donovan, A. A., & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley Professional.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley Professional.
- Go Documentation. (n.d.). Goroutines and Channels.
- Oracle. (n.d.). Concurrency Utilities.
- Stack Overflow. (n.d.). *Is Go's log package safe for concurrent use?* Retrieved from stackoverflow.com.