Assignment 6: Multi-threaded Data Processing System

Nguyen The Duy Khanh

005019181

MSCS-632 - Advanced Programming Languages

Vanessa Cooper

April 6, 2025

Table of Contents

Shared Queue in Go

Share Queue in Go will use a channel to store all the messages since we are sending messages concurrently based on the worker pool implementation. The application will compile in the following orders:

- **Step 1:** Initialize all the tasks to the queue and close the channel when finishing sending all the messages to the tasks

```go
// Add tasks
for i := 1; i <= 10; i++ {
    queue.AddTask(fmt.Sprintf("Task %d", i))
}
queue.Close() // Close the channel so workers know when to stop

// Start workers
for i := 1; i <= 4; i++ {
    wg.Add(1)
    go worker(i, queue, &resultList, &mutex, &wg)
}

wg.Wait()
```

Figure 1: Initialize task in worker pool with Go

- **Step 2:** The worker will receive the task from the queue and each will be first come first serve for each worker. However, since slice is a non-concurrent safe resource, we would need to have a mutex as a lock to append the data successfully.

```go
func worker(id int, queue *SharedQueue, resultList *[]string, mutex *sync.Mutex, wg *sync.WaitGroup) {
    defer wg.Done()

    for task := range queue.queue {
        fmt.Printf("Worker %d processing task: %s\n", id, task)
        time.Sleep(1 * time.Second) // Simulate processing

        // Safely append to result list
        mutex.Lock()
        *resultList = append(*resultList, fmt.Sprintf("Result from Worker %d: %s", id, task))
        mutex.Unlock()

        fmt.Printf("Worker %d completed task: %s\n", id, task)
    }
}
```

Figure 2: Receiving task in worker pool with Go

- **Step 3:** When there is any panic in sending messages, we will return a more friendly user error without panic.

```go
// Defer to exit peacefully
defer func() {
    if r := recover(); r != nil {
        fmt.Println("Detect issues in application. Please try again")
        os.Exit(1)
    }
}()
```

Figure 3: Receiving task in worker pool with Go

When running with *go run main.go*, the application will return the following output

```
❯ go run main.go
Worker 4 processing task: Task 1
Worker 2 processing task: Task 2
Worker 3 processing task: Task 3
Worker 1 processing task: Task 4
Worker 1 completed task: Task 4
Worker 1 processing task: Task 5
Worker 4 completed task: Task 1
Worker 4 processing task: Task 6
Worker 3 completed task: Task 3
Worker 3 processing task: Task 7
Worker 2 completed task: Task 2
Worker 2 processing task: Task 8
Worker 1 completed task: Task 5
Worker 1 processing task: Task 9
Worker 2 completed task: Task 8
Worker 2 processing task: Task 10
Worker 3 completed task: Task 7
Worker 4 completed task: Task 6
Worker 1 completed task: Task 9
Worker 2 completed task: Task 10

Final Results:
Result from Worker 1: Task 4
Result from Worker 4: Task 1
Result from Worker 3: Task 3
Result from Worker 2: Task 2
Result from Worker 1: Task 5
Result from Worker 2: Task 8
Result from Worker 4: Task 6
Result from Worker 3: Task 7
Result from Worker 1: Task 9
Result from Worker 2: Task 10

MSCS632_Assignment6 on main via
```

Figure 3: Sample output with Go

Shared Queue in Java

In Java, the shared queue is implemented using a BlockingQueue to manage tasks in a thread-safe manner as they're processed concurrently by a pool of worker threads.

- **Step 1**: Begins with the initialization of all tasks using queue.addTask(). Once all real tasks are added, a special "poison pill" message is added—one per worker—to signal graceful termination.

```java
public WorkerThread(SharedQueue queue, List<String> resultList) {
    this.queue = queue;
    this.resultList = resultList;
}

@Override
public void run() {
    try {
        while (true) {
            String task = queue.getTask();

            if (task.equals(POISON_PILL)) {
                System.out.println(Thread.currentThread().getName() + " received poison pill. Exiting.");
                break;
            }

            System.out.println(Thread.currentThread().getName() + " processing task: " + task);
            Thread.sleep(millis:1000); // Simulate work

            synchronized (resultList) {
                resultList.add("Result from " + Thread.currentThread().getName() + ": " + task);
            }

            System.out.println(Thread.currentThread().getName() + " completed task: " + task);
        }
    } catch (InterruptedException e) {
        System.err.println("Error in thread " + Thread.currentThread().getName() + ": " + e.getMessage());
    }
}

public static String getPoisonPill() {
    return POISON_PILL;
}
```

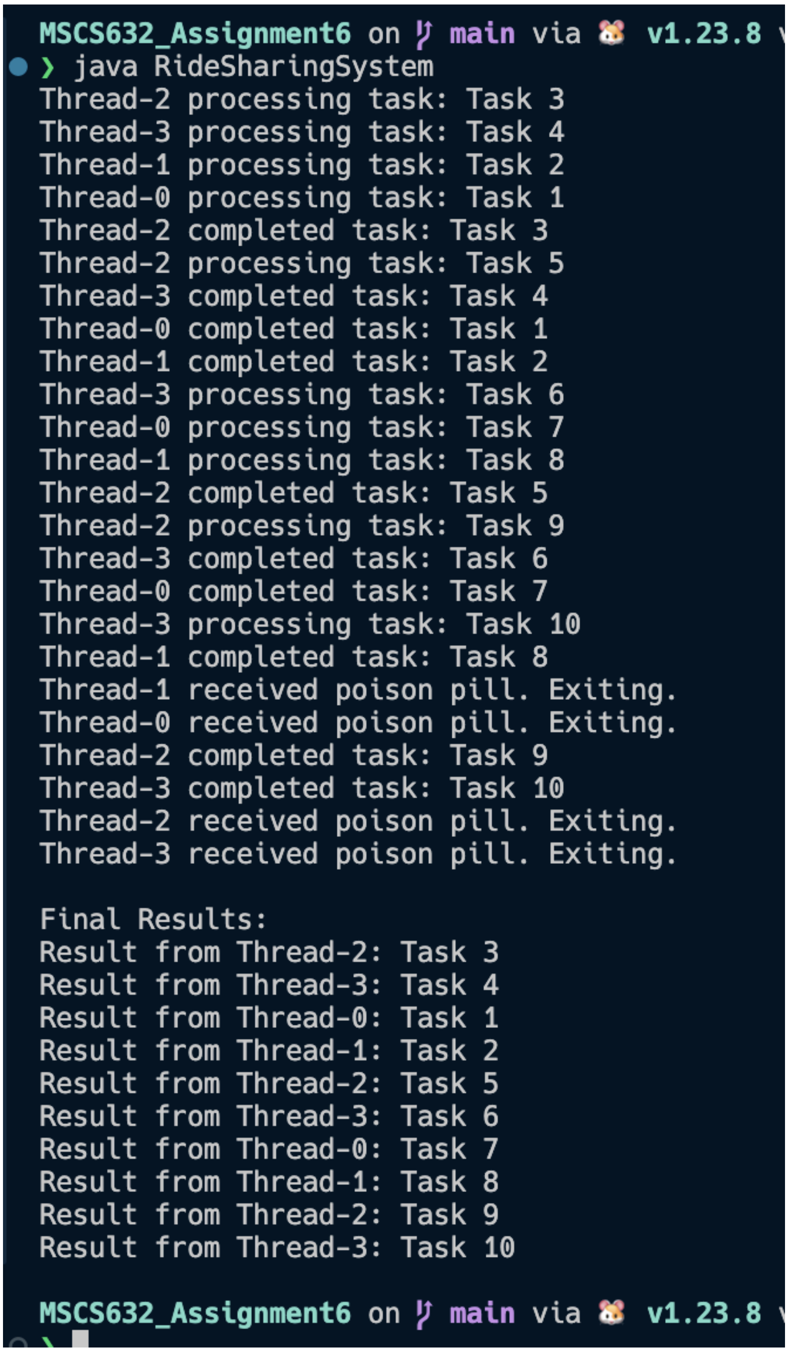
Figure 4: Initialize task in worker pool with Java

- **Step 2**: Each worker thread retrieves tasks from the queue in a first-come, first-serve fashion using queue.getTask(). After simulating the task processing, each thread synchronized access to a shared result list to safely append its output, since Java's ArrayList is not inherently thread-safe
- **Step 3:** Handles potential issues by catching InterruptedException, ensuring the system doesn't crash and provides meaningful error messages rather than raw stack traces—leading to a more user-friendly and resilient application.

```java
    }
} catch (InterruptedException e) {
    System.err.println("Error in thread " + Thread.currentThread().getName() + ": " + e.getMessage());
}
```

Figure 5: Handle exception with Java

When running with *java RideSharingSystem*, the application will return the following output



```
MSCS632_Assignment6 on main via v1.23.8
❯ java RideSharingSystem
Thread-2 processing task: Task 3
Thread-3 processing task: Task 4
Thread-1 processing task: Task 2
Thread-0 processing task: Task 1
Thread-2 completed task: Task 3
Thread-2 processing task: Task 5
Thread-3 completed task: Task 4
Thread-0 completed task: Task 1
Thread-1 completed task: Task 2
Thread-3 processing task: Task 6
Thread-0 processing task: Task 7
Thread-1 processing task: Task 8
Thread-2 completed task: Task 5
Thread-2 processing task: Task 9
Thread-3 completed task: Task 6
Thread-0 completed task: Task 7
Thread-3 processing task: Task 10
Thread-1 completed task: Task 8
Thread-1 received poison pill. Exiting.
Thread-0 received poison pill. Exiting.
Thread-2 completed task: Task 9
Thread-3 completed task: Task 10
Thread-2 received poison pill. Exiting.
Thread-3 received poison pill. Exiting.

Final Results:
Result from Thread-2: Task 3
Result from Thread-3: Task 4
Result from Thread-0: Task 1
Result from Thread-1: Task 2
Result from Thread-2: Task 5
Result from Thread-3: Task 6
Result from Thread-0: Task 7
Result from Thread-1: Task 8
Result from Thread-2: Task 9
Result from Thread-3: Task 10

MSCS632_Assignment6 on main via v1.23.8
❯
```

References

Khanh Nguyen. https://github.com/khanhntd/MSCS632_Assignment6