**Multi-threaded Data Processing System**

This report discusses the development of a multi-threaded Data Processing System implemented in Java and Go, simulating worker threads that retrieve and process tasks from a shared queue. The objective is to understand concurrency control, thread synchronization, error handling, and efficient data management using parallel processing. Both implementations were tested, and outputs were logged to verify correctness.

**Concurrency and Exception Handling in Java**

Java uses a thread-based concurrency model that operates on native threads managed by the Java Virtual Machine (JVM). For this system, we used the following Java concurrency constructs:

* ExecutorService was used to create a fixed thread pool (Executors.newFixedThreadPool()), allowing for efficient thread reuse.
* The shared task queue was implemented using a BlockingQueue<String>, specifically LinkedBlockingQueue, which provides built-in thread safety.
* The results list was protected using Collections.synchronizedList() to prevent concurrent modification exceptions.

Exception handling in Java was implemented using traditional try-catch blocks. Key errors managed include:

* InterruptedException — when a thread is interrupted while waiting.
* IOException — during file writing using PrintWriter.

These blocks ensured robust error management, thread safety, and clean termination of all threads upon task completion.

**Concurrency and Error Handling in Go**

Go employs a goroutine-based concurrency model, which is lightweight and optimized for concurrent execution. The Go version of this system utilized:

* Goroutines to simulate worker threads. These are managed by Go’s runtime scheduler and are more lightweight than Java threads.
* Channels served as a concurrency-safe task queue. A buffered channel allowed sending multiple tasks before workers consumed them.
* A sync.Mutex protected access to the shared results slice to avoid data races during concurrent writes.

Error handling in Go is explicit. Rather than exceptions, Go returns errors from functions. Key strategies included:

* Using defer to ensure that file resources are properly closed.
* Checking err after every file or I/O operation and gracefully logging issues.

This design encouraged safer concurrency through message-passing rather than memory sharing and helped reduce synchronization complexity.

**Comparison: Java vs. Go Concurrency**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Java** | **Go** |
| Concurrency Model | Thread-based | Goroutine-based |
| Thread Management | ExecutorService | Go runtime scheduler |
| Shared Queue | BlockingQueue (thread-safe) | Buffered channels |
| Result Storage | Synchronized List | Shared slice with sync.Mutex |
| Error Handling | try-catch blocks | Error values (if err != nil) |
| Resource Management | try-finally or try-with-resources | defer statements |
| Performance (Lightweight) | Moderate (heavier threads) | High (lightweight goroutines) |
| Use Case Suitability | Enterprise-grade systems with heavy processing | Lightweight, scalable concurrent applications |

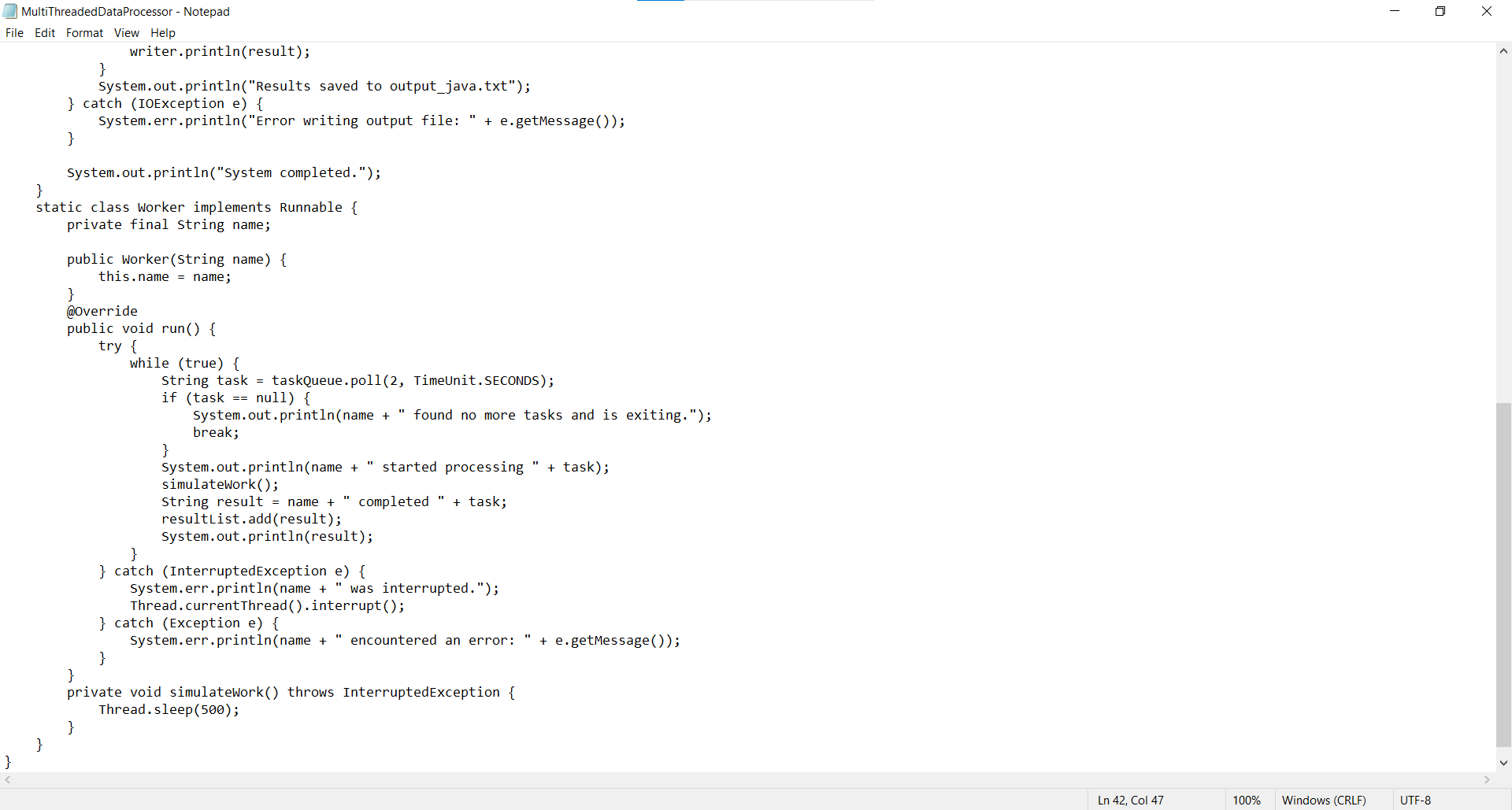
Java offers precise thread control and robust libraries but requires more boilerplate for concurrency. Go favors simplicity and performance through its message-passing model.

**Code and Output Screenshots**

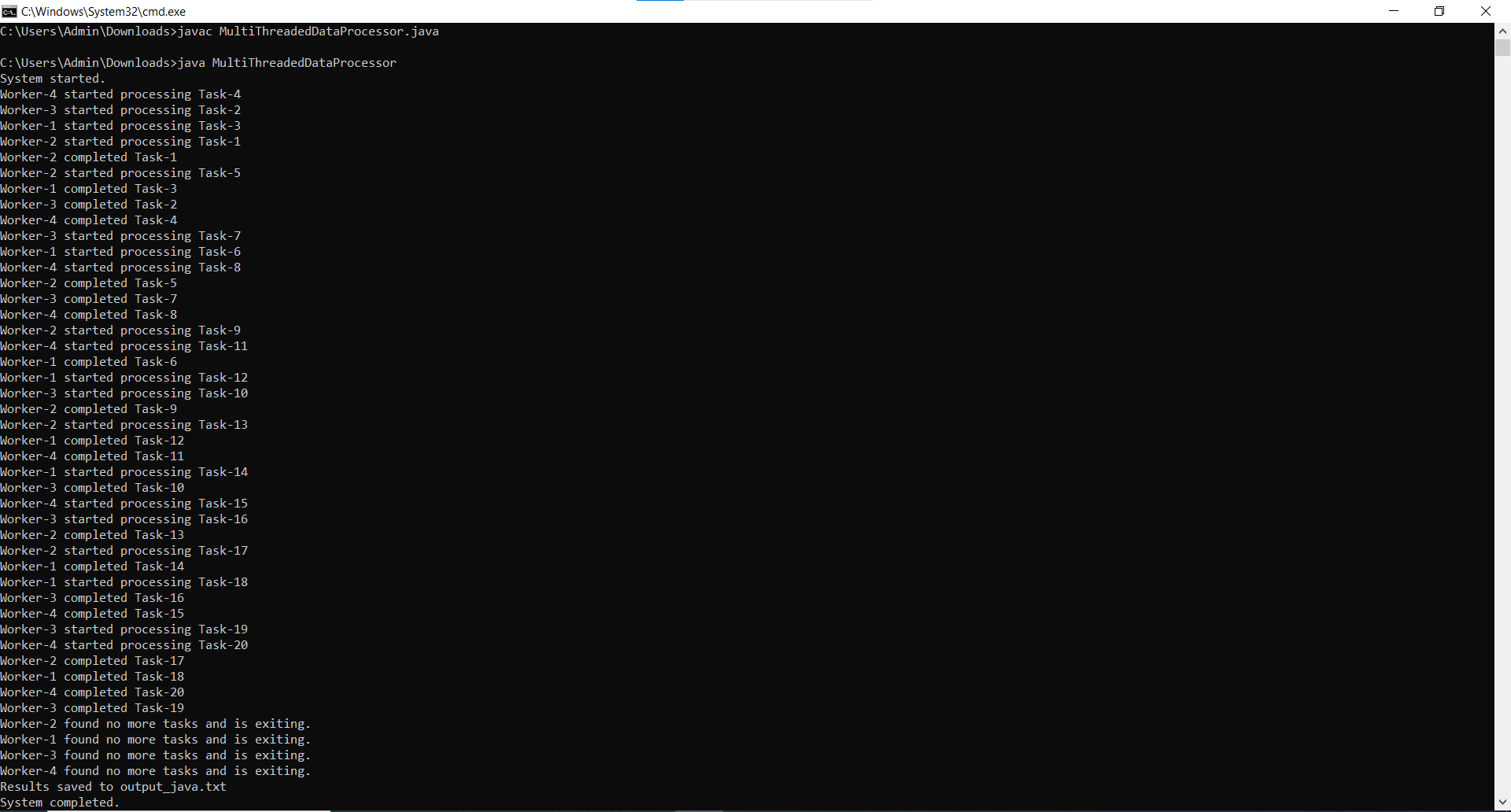
**Java Implementation:**

File name: MultiThreadedDataProcessor.java





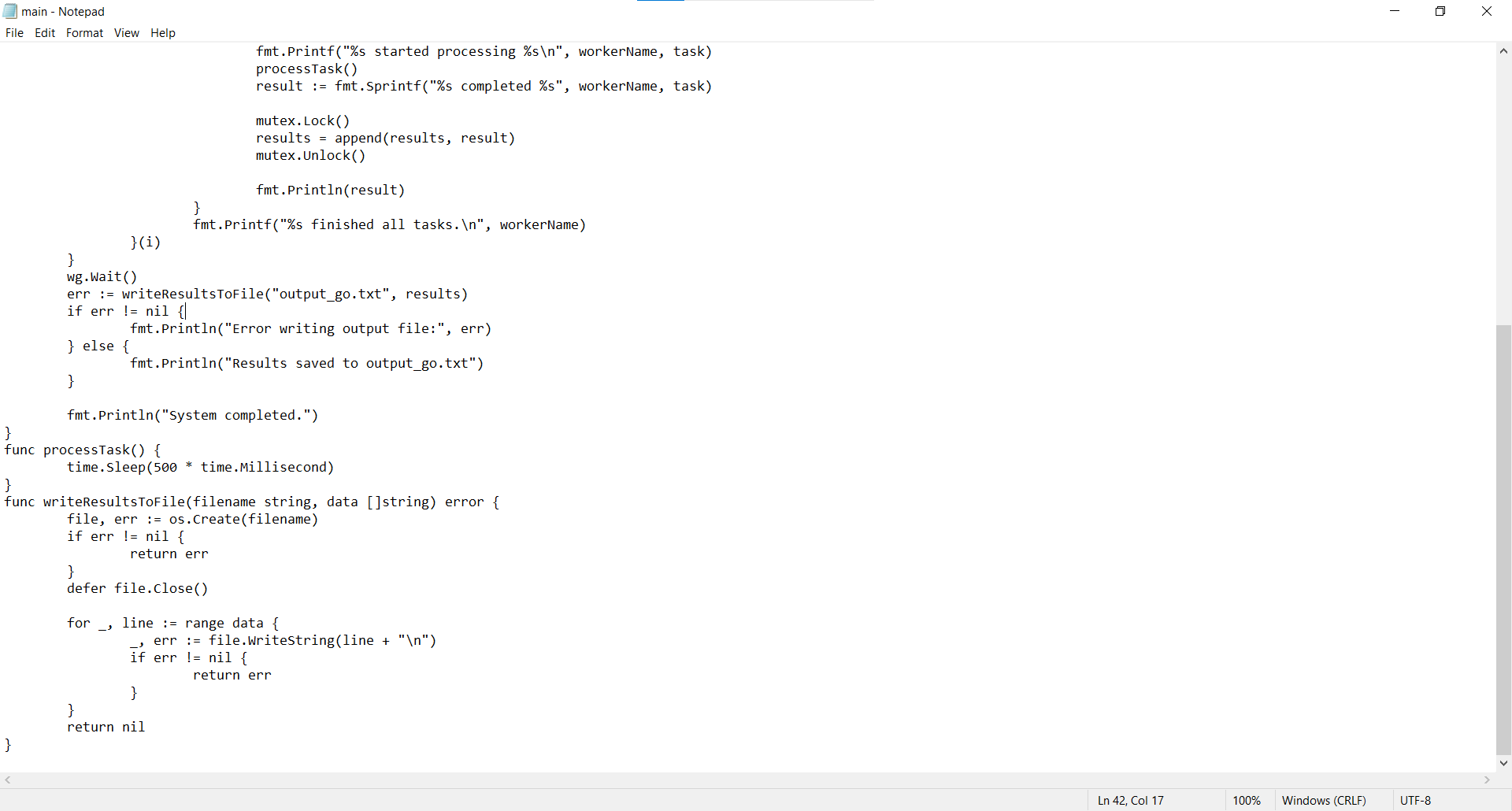
Output:



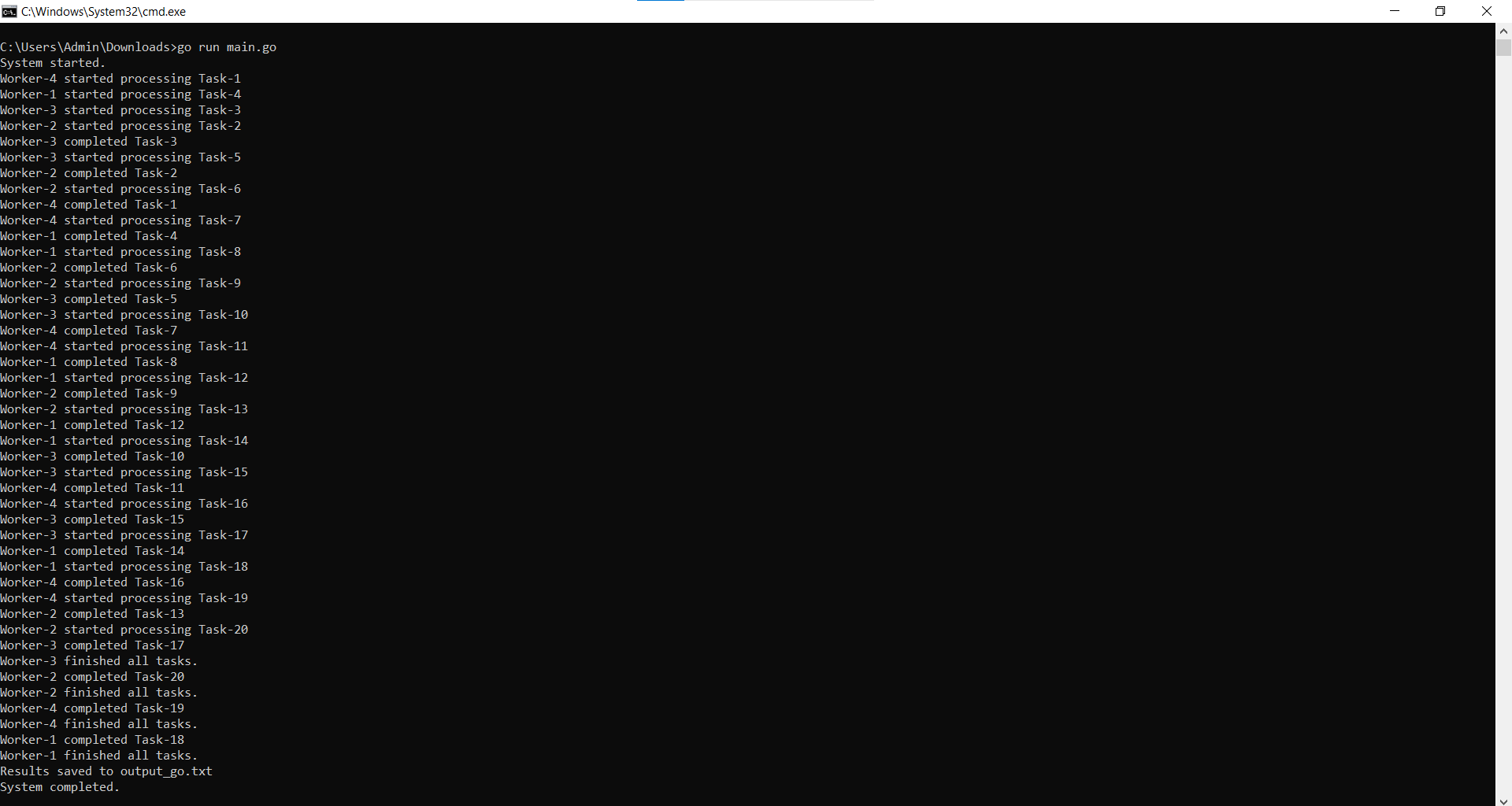
**Go Implementation:**

**File name:** main.go

****

****

**Output:**

****