Go Concurrency Demo

Project Overview

This project demonstrates **multithreading and concurrency** in Go using **Goroutines** and **Channels**. The program simulates a scenario where multiple workers perform tasks concurrently, and their results are collected and printed using channels.

Features Demonstrated

1. **Goroutines**: Lightweight threads that execute functions concurrently.
2. **Synchronization**: Using `sync.WaitGroup` to ensure all goroutines complete before finishing the program.
3. **Channels**: Used for communication between goroutines to pass results safely.
4. **Randomized Task Duration**: Simulates real-world variable processing times using random delays.

Program Flow

- The program starts by creating 5 concurrent workers (goroutines).
- Each worker simulates a task by sleeping for a random amount of time.
- Once a worker finishes its task, it sends its result back to the main function using a channel.
- The main function collects and prints the results from all workers in the order they complete.

Code

```
package main

import (
   "fmt"
   "math/rand"
   "sync"
   "time"
)

// simulateWork is a function that performs a "work" task, like processing data.
func simulateWork(id int, wg *sync.WaitGroup, results chan<- string) {
   defer wg.Done() // Signal completion of this goroutine
   time.Sleep(time.Duration(rand.Intn(1000)) * time.Millisecond) // Simulate variable
processing time
   result := fmt.Sprintf("Worker %d finished work", id)
```

```
    results <- result // Send result to channel
}

func main() {
    rand.Seed(time.Now().UnixNano()) // Seed random number generator for varied sleep
times
    var wg sync.WaitGroup            // WaitGroup to synchronize goroutines
    results := make(chan string, 5)  // Channel to collect results from workers

    // Start 5 goroutines
    for i := 1; i <= 5; i++ {
        wg.Add(1)
        go simulateWork(i, &wg, results)
    }

    // Close results channel when all goroutines are done
    go func() {
        wg.Wait() // Wait for all goroutines to finish
        close(results)
    }()

    // Print results as they arrive
    for result := range results {
        fmt.Println(result)
    }
}
```

Code Explanation

1. **simulateWork**: A function that represents a task performed by each worker. It sleeps for a random duration, then sends a result to the `results` channel.
2. **sync.WaitGroup**: Used to wait for all goroutines to finish their tasks before closing the program.
3. **Channel**: `results` is a buffered channel used to receive results from goroutines. The main function prints each result as they are received.
4. **Random Sleep**: The `rand.Intn` function is used to generate random sleep times, simulating varying task durations for each worker.

How to Run the Program

1. Ensure that you have **Go** installed on your system. You can download it from Go Downloads.

2. Clone or download this repository to your local machine.

Navigate to the project directory:
bash
Copy code
```
cd path/to/go_concurrency_demo
```

3.

Run the program using the command:
bash
Copy code
```
go run main.go
```

4.

Expected Output

When you run the program, you will see output similar to the following, with the order of the results varying each time:

plaintext
Copy code
```
Worker 3 finished work
Worker 1 finished work
Worker 4 finished work
Worker 5 finished work
Worker 2 finished work
```

Conclusion

This Go program demonstrates the power of concurrency with goroutines and channels. It showcases how Go makes it easy to write highly concurrent and efficient applications, making it a great choice for building scalable systems.