EX1

// Write a program where two goroutines pass an integer back and forth

// ten times. Display when each goroutine receives the integer. Increment

// the integer with each pass. Once the integer equals ten, terminate

// the program cleanly.

package main

import (

"fmt"

"sync"

)

func main() {

// Create an unbuffered channel.

share := make(chan int)

// Create the WaitGroup and add a count

// of two, one for each goroutine.

var wg sync.WaitGroup

wg.Add(2)

// Launch two goroutines.

go func() {

goroutine("Bill", share)

wg.Done()

}()

go func() {

goroutine("Joan", share)

wg.Done()

}()

// Start the sharing.

share <- 1

// Wait for the program to finish.

wg.Wait()

}

// goroutine simulates sharing a value.

func goroutine(name string, share chan int) {

for {

// Wait to receive a value.

value, ok := <-share

if !ok {

// If the channel was closed, return.

fmt.Printf("Goroutine %s Down\n", name)

return

}

// Display the value.

fmt.Printf("Goroutine %s Inc %d\n", name, value)

// Terminate when the value is 10.

if value == 10 {

close(share)

fmt.Printf("Goroutine %s Down\n", name)

return

}

// Increment the value and send it

// over the channel.

share <- (value + 1)

}

}

**EX2**

// Write a program that uses a fan out pattern to generate 100 random numbers

// concurrently. Have each goroutine generate a single random number and return

// that number to the main goroutine over a buffered channel. Set the size of

// the buffer channel so no send every blocks. Don't allocate more buffers than

// you need. Have the main goroutine display each random number is receives and

// then terminate the program.

package main

import (

"fmt"

"math/rand"

"time"

)

const (

goroutines = 100

)

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

// Create the buffer channel with a buffer for

// each goroutine to be created.

values := make(chan int, goroutines)

// Iterate and launch each goroutine.

for gr := 0; gr < goroutines; gr++ {

// Create an anonymous function for each goroutine that

// generates a random number and sends it on the channel.

go func() {

values <- rand.Intn(1000)

}()

}

// Create a variable to be used to track received messages.

// Set the value to the number of goroutines created.

wait := goroutines

// Iterate receiving each value until they are all received.

// Store them in a slice of ints.

var nums []int

for wait > 0 {

nums = append(nums, <-values)

wait--

}

// Print the values in our slice.

fmt.Println(nums)

}

**EX3**

// Write a program that uses goroutines to generate up to 100 random numbers.

// Do not send values that are divisible by 2. Have the main goroutine receive

// values and add them to a slice.

package main

import (

"fmt"

"math/rand"

"sync"

"time"

)

const (

goroutines = 100

)

func init() {

rand.Seed(time.Now().UnixNano())

}

func main() {

// Create the channel for sharing results.

values := make(chan int)

// Create a sync.WaitGroup to monitor the Goroutine pool. Add the count.

var wg sync.WaitGroup

wg.Add(goroutines)

// Iterate and launch each goroutine.

for gr := 0; gr < goroutines; gr++ {

// Create an anonymous function for each goroutine.

go func() {

// Ensure the waitgroup is decremented when this function returns.

defer wg.Done()

// Generate a random number up to 1000.

n := rand.Intn(1000)

// Return early if the number is divisible by 2. n%2 == 0

if n%2 == 0 {

return

}

// Send the odd values through the channel.

values <- n

}()

}

// Create a goroutine that waits for the other goroutines to finish then

// closes the channel.

go func() {

wg.Wait()

close(values)

}()

// Receive from the channel until it is closed.

// Store values in a slice of ints.

var nums []int

for n := range values {

nums = append(nums, n)

}

// Print the values in our slice.

fmt.Printf("Result count: %d\n", len(nums))

fmt.Println(nums)

}

**EX4**

// Write a program that creates a fixed set of workers to generate random

// numbers. Discard any number divisible by 2. Continue receiving until 100

// numbers are received. Tell the workers to shut down before terminating.

package main

import (

"fmt"

"math/rand"

"runtime"

"sync"

)

func main() {

// Create the channel for sharing results.

values := make(chan int)

// Create a channel "shutdown" to tell goroutines when to terminate.

shutdown := make(chan struct{})

// Define the size of the worker pool. Use runtime.GOMAXPROCS(0) to size the pool based on number of processors.

poolSize := runtime.GOMAXPROCS(0)

// Create a sync.WaitGroup to monitor the Goroutine pool. Add the count.

var wg sync.WaitGroup

wg.Add(poolSize)

// Create a fixed size pool of goroutines to generate random numbers.

for i := 0; i < poolSize; i++ {

go func(id int) {

// Start an infinite loop.

for {

// Generate a random number up to 1000.

n := rand.Intn(1000)

// Use a select to either send the number or receive the shutdown signal.

select {

// In one case send the random number.

case values <- n:

fmt.Printf("Worker %d sent %d\n", id, n)

// In another case receive from the shutdown channel.

case <-shutdown:

fmt.Printf("Worker %d shutting down\n", id)

wg.Done()

return

}

}

}(i)

}

// Create a slice to hold the random numbers.

var nums []int

for i := range values {

// continue the loop if the value was even.

if i%2 == 0 {

fmt.Println("Discarding", i)

continue

}

// Store the odd number.

fmt.Println("Keeping", i)

nums = append(nums, i)

// break the loop once we have 100 results.

if len(nums) == 100 {

break

}

}

// Send the shutdown signal by closing the channel.

fmt.Println("Receiver sending shutdown signal")

close(shutdown)

// Wait for the Goroutines to finish.

wg.Wait()

// Print the values in our slice.

fmt.Printf("Result count: %d\n", len(nums))

fmt.Println(nums)

}