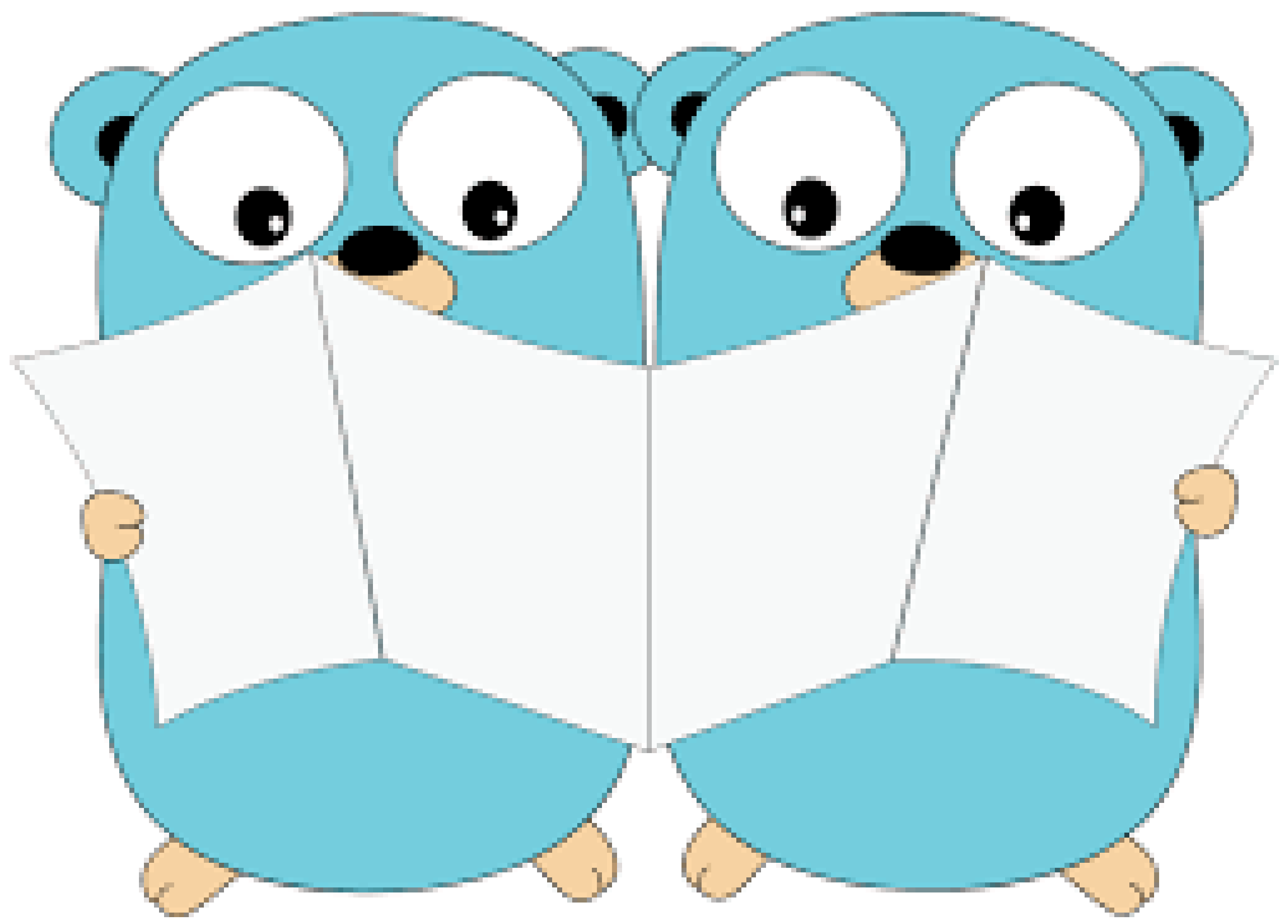
Concurrency Patterns in GO

golangbyte.com

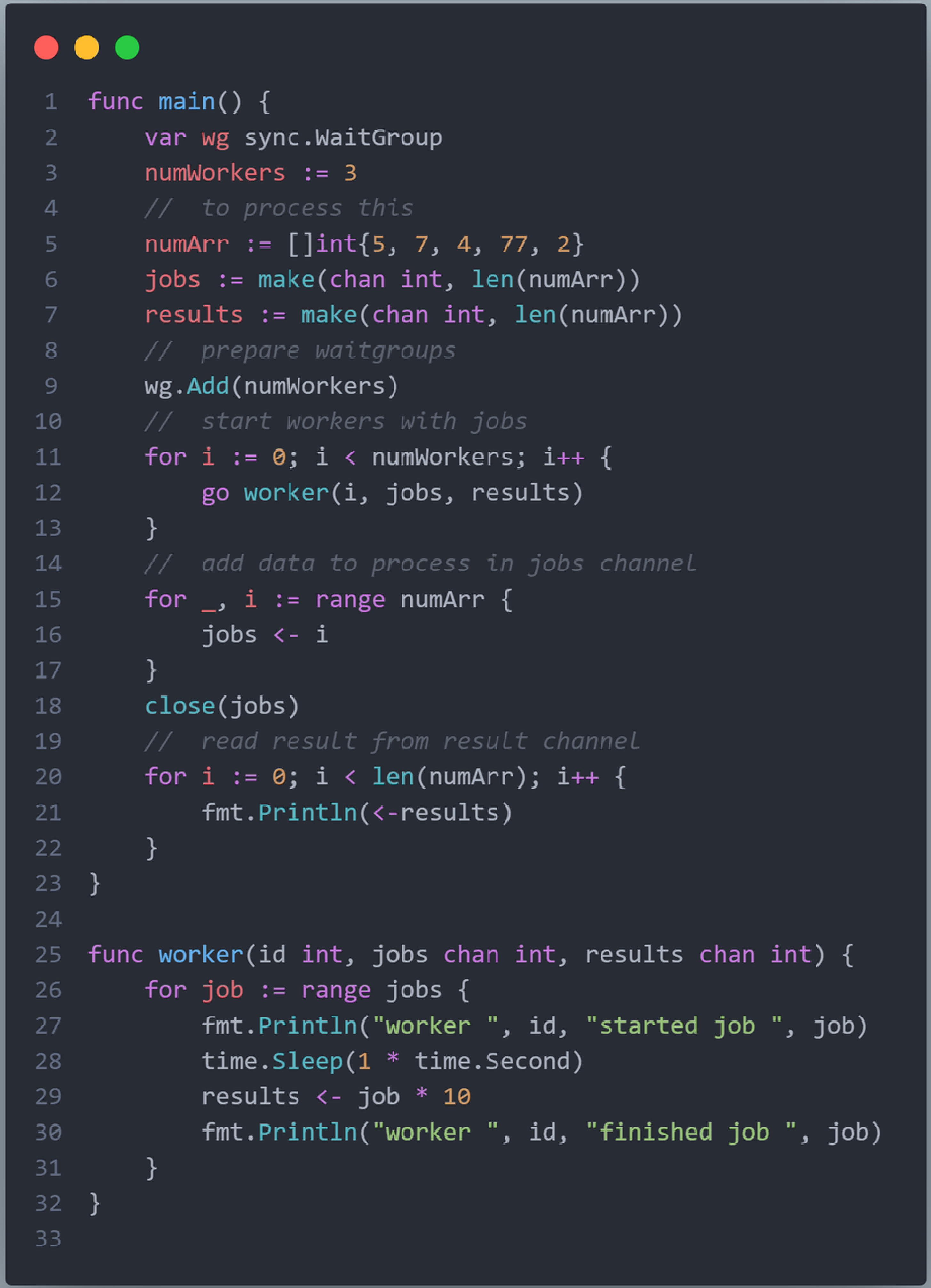
Contents



worker pool

worker pool is a concurrency pattern that aims at reusing goroutines. It initiallizes them at first then assigns them tasks. Once task is done, new task will be assigned to worker.This helps in preventing unwanted initialization of concurrent functions.

Worker pool pattern is used as a component in many complex patterns.



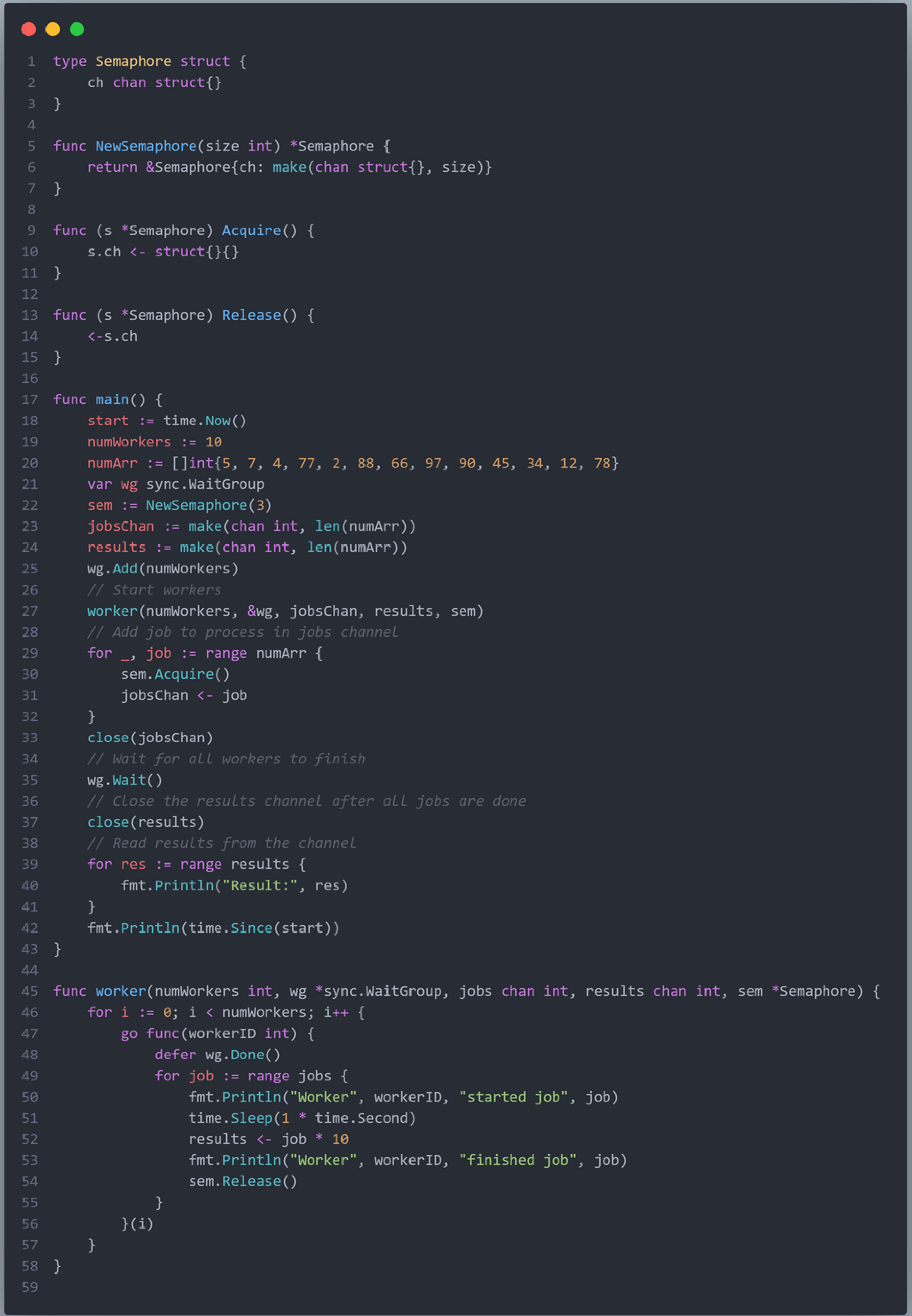
```go
func main() {
    var wg sync.WaitGroup
    numWorkers := 3
    //  to process this
    numArr := []int{5, 7, 4, 77, 2}
    jobs := make(chan int, len(numArr))
    results := make(chan int, len(numArr))
    //  prepare waitgroups
    wg.Add(numWorkers)
    //  start workers with jobs
    for i := 0; i < numWorkers; i++ {
        go worker(i, jobs, results)
    }
    //  add data to process in jobs channel
    for _, i := range numArr {
        jobs <- i
    }
    close(jobs)
    //  read result from result channel
    for i := 0; i < len(numArr); i++ {
        fmt.Println(<-results)
    }
}

func worker(id int, jobs chan int, results chan int) {
    for job := range jobs {
        fmt.Println("worker ", id, "started job ", job)
        time.Sleep(1 * time.Second)
        results <- job * 10
        fmt.Println("worker ", id, "finished job ", job)
    }
}

```

semaphore

Semaphore pattern is used to limit access control to resource in a given instant. It helps in limiting concurrent workers at a time by blocking number of workers at aa instant to the resource.

Semaphore is a struct with channel of user defined capacity. The worker acquires a token from channel and releases it after work is done for other worker.

There are standard libraries also available for semaphore patters operations.



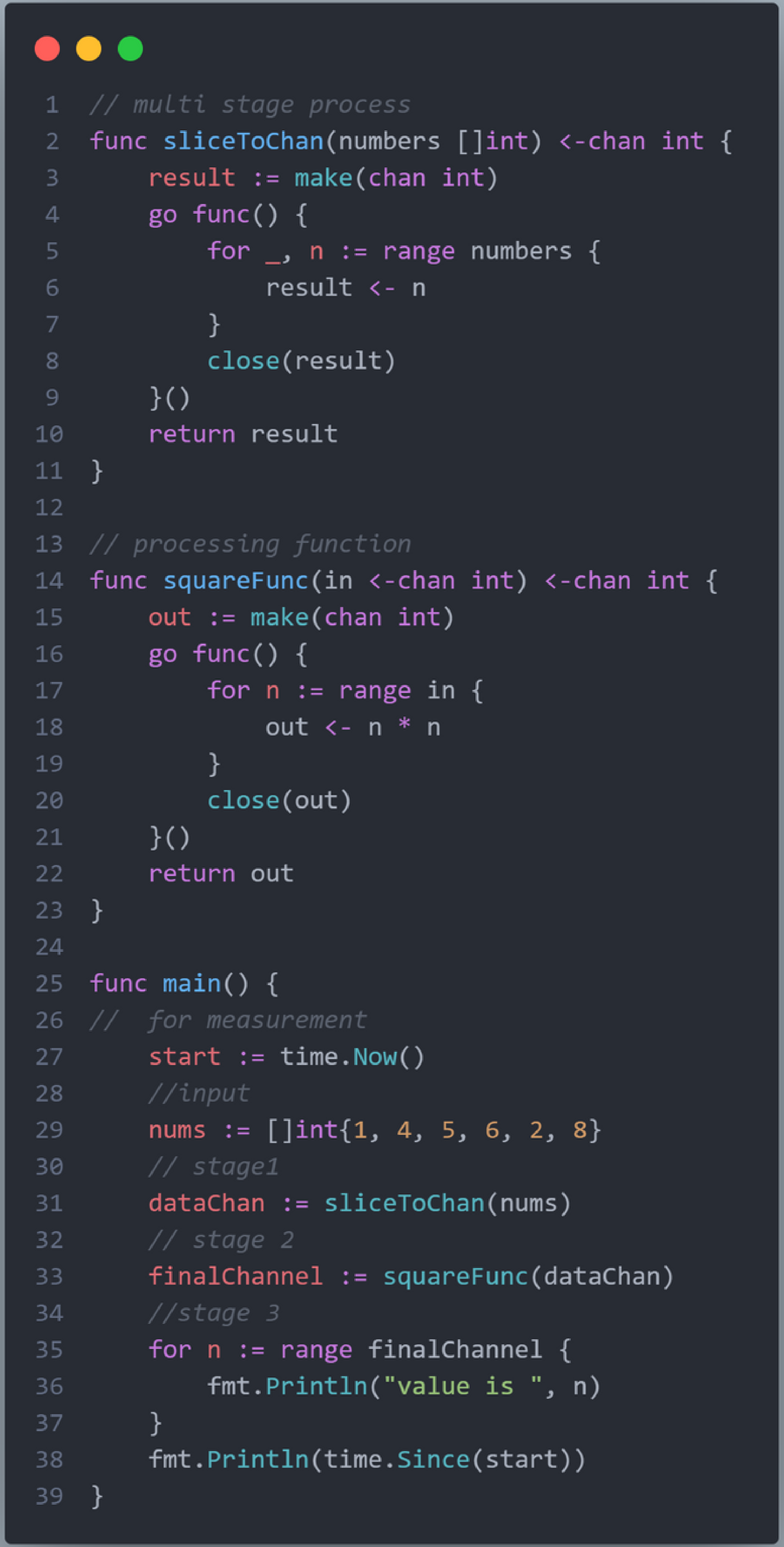
```go
type Semaphore struct {
    ch chan struct{}
}

func NewSemaphore(size int) *Semaphore {
    return &Semaphore{ch: make(chan struct{}, size)}
}

func (s *Semaphore) Acquire() {
    s.ch <- struct{}{}
}

func (s *Semaphore) Release() {
    <-s.ch
}

func main() {
    start := time.Now()
    numWorkers := 10
    numArr := []int{5, 7, 4, 77, 2, 88, 66, 97, 90, 45, 34, 12, 78}
    var wg sync.WaitGroup
    sem := NewSemaphore(3)
    jobsChan := make(chan int, len(numArr))
    results := make(chan int, len(numArr))
    wg.Add(numWorkers)
    // Start workers
    worker(numWorkers, &wg, jobsChan, results, sem)
    // Add job to process in jobs channel
    for _, job := range numArr {
        sem.Acquire()
        jobsChan <- job
    }
    close(jobsChan)
    // Wait for all workers to finish
    wg.Wait()
    // Close the results channel after all jobs are done
    close(results)
    // Read results from the channel
    for res := range results {
        fmt.Println("Result:", res)
    }
    fmt.Println(time.Since(start))
}

func worker(numWorkers int, wg *sync.WaitGroup, jobs chan int, results chan int, sem *Semaphore) {
    for i := 0; i < numWorkers; i++ {
        go func(workerID int) {
            defer wg.Done()
            for job := range jobs {
                fmt.Println("Worker", workerID, "started job", job)
                time.Sleep(1 * time.Second)
                results <- job * 10
                fmt.Println("Worker", workerID, "finished job", job)
                sem.Release()
            }
        }(i)
    }
}
```

pipeline

Pipeline pattern is a multi stage pattern for processing the task.
It breaks down task into smaller subtasks working in parallel and moving data via channels for efficient operation of task concurrently.

The functions are communicated via stage channels, each for a specific responsiblity. this completes the operation in multi steps.

This significantly improves the time and makes better use of resources.

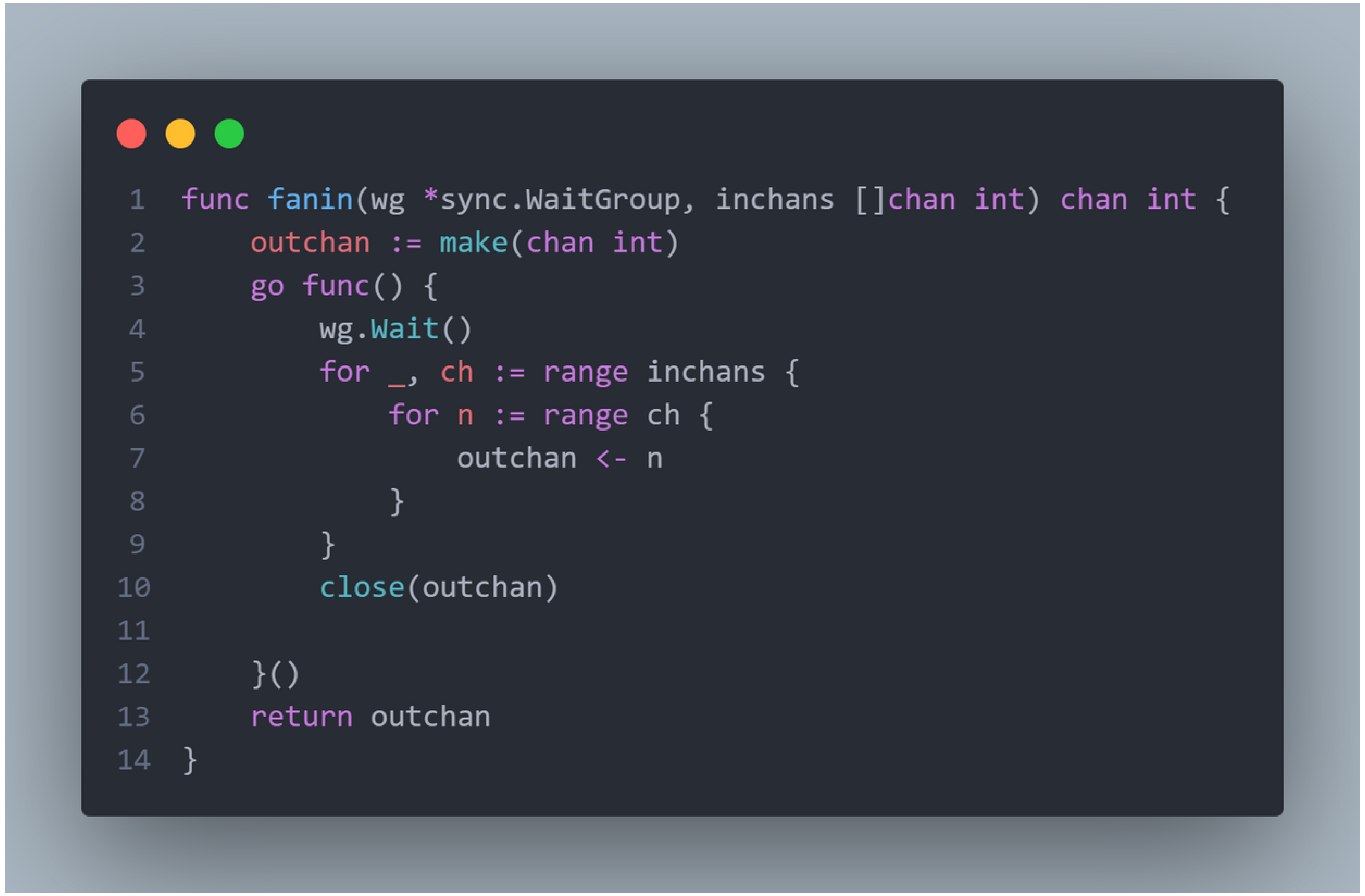


```go
// multi stage process
func sliceToChan(numbers []int) <-chan int {
    result := make(chan int)
    go func() {
        for _, n := range numbers {
            result <- n
        }
        close(result)
    }()
    return result
}

// processing function
func squareFunc(in <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}

func main() {
//  for measurement
    start := time.Now()
    //input
    nums := []int{1, 4, 5, 6, 2, 8}
    // stage1
    dataChan := sliceToChan(nums)
    // stage 2
    finalChannel := squareFunc(dataChan)
    //stage 3
    for n := range finalChannel {
        fmt.Println("value is ", n)
    }
    fmt.Println(time.Since(start))
}
```

Fan In

Fan in is a pattern that is used to get data from a lot of channels as input and combine them into a single output channel.

This is used with pipelines for more effective operations of big tasks with fanout.



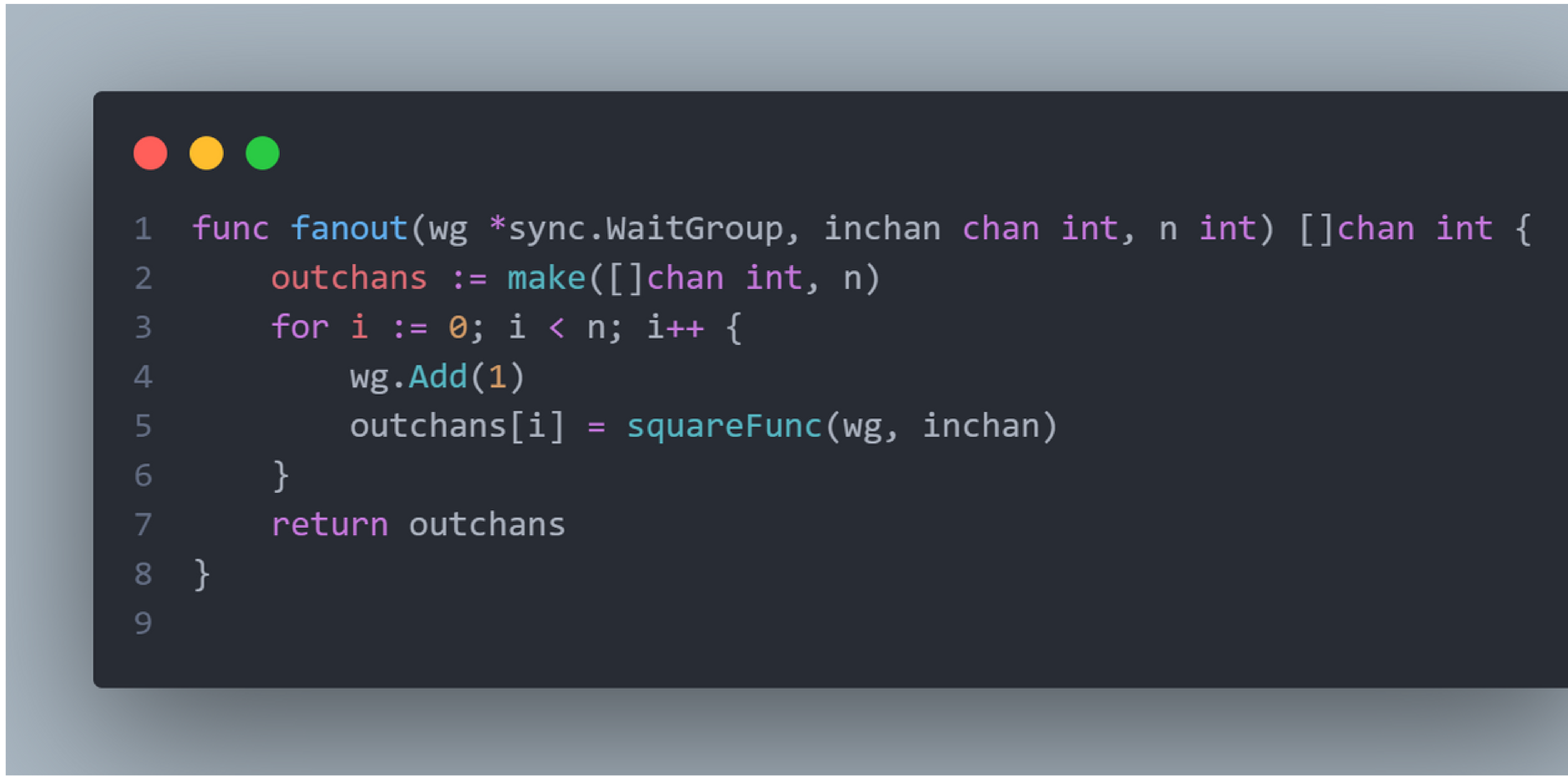
```go
func fanin(wg *sync.WaitGroup, inchans []chan int) chan int {
    outchan := make(chan int)
    go func() {
        wg.Wait()
        for _, ch := range inchans {
            for n := range ch {
                outchan <- n
            }
        }
        close(outchan)

    }()
    return outchan
}
```

Fan Out

Fanout is a pattern that takes the task and spreads it across multiple channels along with worker goroutines.

So tasks can be processed in parallel in independent channels with independent workers.

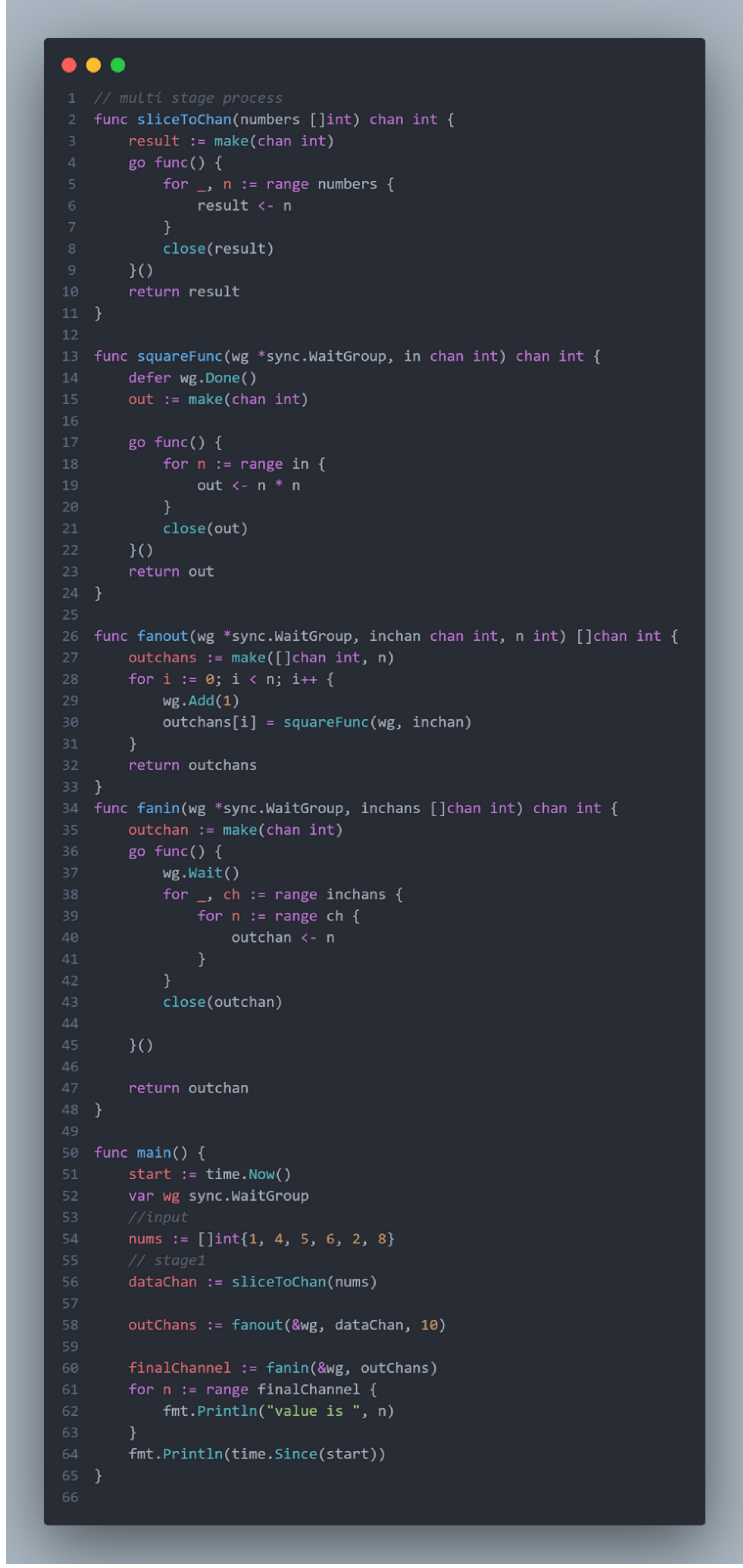
Fanin is used with fanout for processing big tasks effeciently.



```go
func fanout(wg *sync.WaitGroup, inchan chan int, n int) []chan int {
    outchans := make([]chan int, n)
    for i := 0; i < n; i++ {
        wg.Add(1)
        outchans[i] = squareFunc(wg, inchan)
    }
    return outchans
}

```

Project

A project that combines fanin fanout with pipeline pattern to process the worker task more effeciently. the worker function can be updated along with input job channel. for other functional operations if required.



```go
// multi stage process
func sliceToChan(numbers []int) chan int {
    result := make(chan int)
    go func() {
        for _, n := range numbers {
            result <- n
        }
        close(result)
    }()
    return result
}

func squareFunc(wg *sync.WaitGroup, in chan int) chan int {
    defer wg.Done()
    out := make(chan int)

    go func() {
        for n := range in {
            out <- n * n
        }
        close(out)
    }()
    return out
}

func fanout(wg *sync.WaitGroup, inchan chan int, n int) []chan int {
    outchans := make([]chan int, n)
    for i := 0; i < n; i++ {
        wg.Add(1)
        outchans[i] = squareFunc(wg, inchan)
    }
    return outchans
}
func fanin(wg *sync.WaitGroup, inchans []chan int) chan int {
    outchan := make(chan int)
    go func() {
        wg.Wait()
        for _, ch := range inchans {
            for n := range ch {
                outchan <- n
            }
        }
        close(outchan)

    }()

    return outchan
}

func main() {
    start := time.Now()
    var wg sync.WaitGroup
    //input
    nums := []int{1, 4, 5, 6, 2, 8}
    // stage1
    dataChan := sliceToChan(nums)

    outChans := fanout(&wg, dataChan, 10)

    finalChannel := fanin(&wg, outChans)
    for n := range finalChannel {
        fmt.Println("value is ", n)
    }
    fmt.Println(time.Since(start))
}
```

github.com/subpxl/golangbyte

golangbyte.com