Go Track

PowerUp! SG Tech Traineeship – Software Engineering

Learning Objectives – Mod 12

At the end of the course, participants should be able to:

- Define the components in error handling syntax.
- Examine the different panic styles of Go.
- Demonstrate the different styles of syntax in error handling for Go.

Errors vs Panics

- Go distinguishes between errors & panics – bad things that can happen during program execution.
- An *error* indicates that a particular task couldn't be completed successfully, something happened in the task that violate expectations about what should have happened.
- A *panic* indicates that a severe event occurred, such that system/subsystem cannot function, or signals abnormal conditions that may threaten the security of a program.
- Idiomatic Go emphasizes on keeping errors at the forefront of developers' minds. It gets developers used to idea of coding defensively, despite how good they think they are.

Error Handling

- Go does not make use of throw and catch exception mechanism used by other languages such as Python, Java and C#.
- Others like C, makes use of the return value for error handling.

Error Handling

- Go exploits the ability to return multiple values.
- Idiomatic Go technique for issuing errors, is to return the error always as the last value of a return.

- Because errors are always the last value returned, error handling in Go follows a certain pattern.
- Functions that return error, is wrapped in an if-else statement that checks whether the error is something other than nil, and handles it if so.

Error Handling

```
package main

import (
        "errors"
        "fmt"
        "os"
        "strings"
)
//Concat concatenates several strings, separated by commas.
//It returns an empty string and an error if no strings were passed
in.
func Concat(parts ...string) (string, error) {
        if len(parts) == 0 {
                return "", errors.New("No strings supplied")
        }
        return strings.Join(parts, " "), nil
}
```

Error Handling

optional assignment statement, that stays in if else scope.

```
func main() {
        args := os.Args[1:]

        if result, err := Concat(args...); err != nil {
                fmt.Printf("Error: %s\n", err)
        } else {
                fmt.Printf("Concatenated string: '%s'\n", result)
        }

}
```

Error Handling

- Sometimes the presence or absence of the error does not impact the task at hand, so can avoid doing error if-else check.
- In this case, developers could ignore the error using blank identifiers and work with result.

```
func main() {
        args := os.Args[1:]

        result, _ := Concat(args...)
        fmt.Printf("Concatenated string: '%s'\n", result)

}
```

Error Handling

<u>Return of result with error</u>

- Nils traditionally frequent cases of bugs, so programmers often need to check values to protect against nils.
- Some developers also treat nils as placeholders when they do not wish to return value.

- In Go however, nils indicate something more specific.
- When error return value is nil, it indicates that there are no errors when the function is executed.
- And so returning nil results with errors unnecessarily is not recommended.

Error Handling

- So rule of thumb: when a function can return a useful result, it should return one.
- Under a failure condition, if no useful data can be constructed, then return nil result with an error.

Comments describe error condition

- Notice also in most Go programs, Go emphasizes writing of concise and but useful comments atop every shared function.
- The comments should indicate what happens under normal operations as well as what happens under an error condition.

Error Handling

- When creating errors, Go has 2 useful functions:
    - *errors.New()* from errors package – great for creating simple new errors.
    - *fmt.Errorf()* gives the option of using formatting string on the error message.
- In Go, most errors are often returned of type error, creating specific error types is rare.
- Comes from notion that most errors have no special attributes that are better conveyed by specific error type=> returning a generic error simplest way to handle things!

Go usually favours convention over language =>code is simpler to read and write!

Error Handling

- Go's error type is an interface that has following listing:

```
type error interface {
        Error() string
}
```

- Anything that has an Error() function returning a string satisfies this contact.
- Sometimes, you want the errors to contain more than a simple string =>custom error types

Error Handling

- Custom error types – create a type that implements the error interface, but provide additional functionality

```
type ParseError struct {
        Message string //error message
        Line, Char int //location
}

func (p *parseError) Error() string{
        format := "%s oln Line %d, Char %d"

        return fmt.Sprintf(format, p.Message, p.Line, p.Char)
}
```

NGEE ANN SCHOOL OF INFOCOMM TECHNOLOGY

Error Handling

- What if we need one function that does return different kinds of errors?
- Using previous technique a little heavy-handed => create package-scoped error variables e.g. io.EOF and io.ErrNoProgress in io package.
- Advantage: same error variables only instantiate once, and used repeatedly.

Error Handling

```go
var ErrTimeout = errors.New("The request timed out")
var ErrRejected = errors.New("The request was rejected")

var random = rand.New(rand.NewSource(35))

func main() {
	response, err := SendRequest("Hello")
	for err == ErrTimeout {
		fmt.Println("Timeout. Retrying.")
		response, err = SendRequest("Hello")
	}
	if err != nil {
		fmt.Println(err)
	} else {
		fmt.Println(response)
	}
}
```

NGEE ANN
SCHOOL OF INFOCOMM TECHNOLOGY

Error Handling

```
func SendRequest(req string) (string, error) {

       switch random.Int() % 3 {
       case 0:
              return "Success", nil
       case 1:
              return "", ErrRejected
       default:
              return "", ErrTimeout
       }
}
```

Panic System

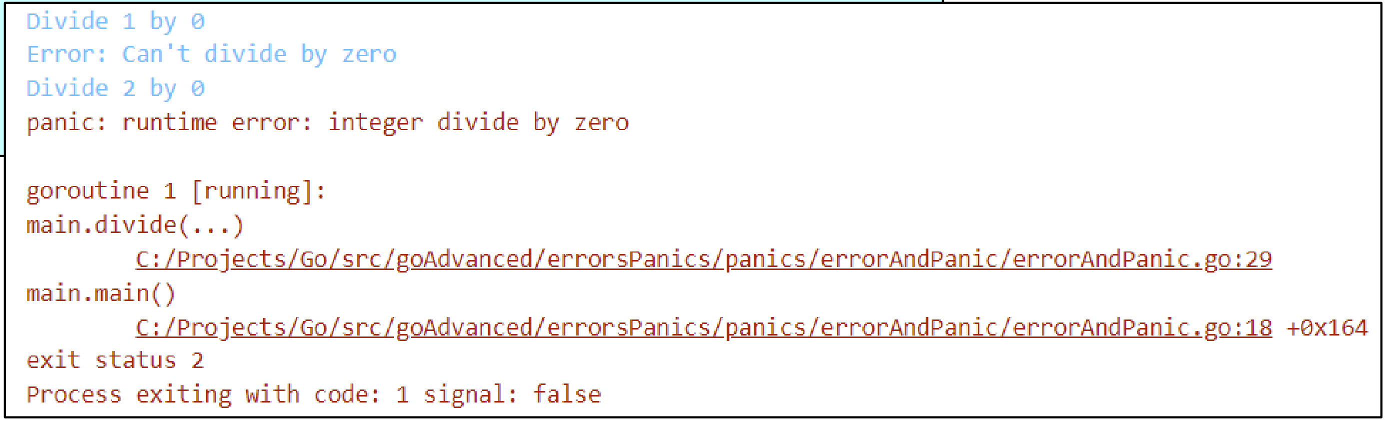
- Go provides a second way of indicating something is wrong: the ***panic*** system
- Panic system should be used sparingly and wisely.
- Need to distinguish between error and panic:
  - An *error* indicates and event occurred that violate expectations of what would have happened.
  - A *panic* indicates something that have gone wrong, in a way the system cannot continue to function.
- Go assumes that errors will be handled by programmers.
- When panic occurs, Go unwinds the stack, look for handlers for the panic. When no handler is found, Go unwinds all the way to top of function stack and stops the program.
- An unhandled panic, thus will kill the application.

Panic System

```
var ErrDivideByZero = errors.New("Can't divide by zero")

func main() {
        fmt.Println("Divide 1 by 0")
        _, err := precheckDivide(1, 0)
        if err != nil {
                fmt.Printf("Error: %s\n", err)
        }

        fmt.Println("Divide 2 by 0")
        divide(2, 0)
}
```

Panic System

```
func precheckDivide(a, b int) (int, error) {
	if b == 0 {
		return 0, ErrDivideByZero
	}
	return divide(a, b), nil
}

func divide(a, b int) int {
	return a / b
}
```



```
Divide 1 by 0
Error: Can't divide by zero
Divide 2 by 0
panic: runtime error: integer divide by zero

goroutine 1 [running]:
main.divide(...)
	C:/Projects/Go/src/goAdvanced/errorsPanics/panics/errorAndPanic/errorAndPanic.go:29
main.main()
	C:/Projects/Go/src/goAdvanced/errorsPanics/panics/errorAndPanic/errorAndPanic.go:18 +0x164
exit status 2
Process exiting with code: 1 signal: false
```

Panic System

- Therefore, errors could be said to be something that developers ought to expect to go wrong.
- Errors are documented in code, which specifies error conditions that are handled.

- Panics could be said unexpected or due to unforeseen circumstances. Occur when a constraint or limit is surpassed.

General rule of thumb: Do not panic, unless no clear way of handling condition. Return errors wherever possible.

Panic System

Working with Panics

- Definition of Go's panic function can be expressed like panic(interface{}). (In fact, almost anything can be passed in as argument)

```go
package main

func main() {
        panic(nil)
}
```

```
panic: nil

goroutine 1 [running]:
main.main()
        C:/Projects/Go/src/goAdvanced/errorsPanics/panics/panicWithNil.go:4 +0x30
exit status 2
Process exiting with code: 1 signal: false
```

Panic System

- For something more useful, can pass in a string:

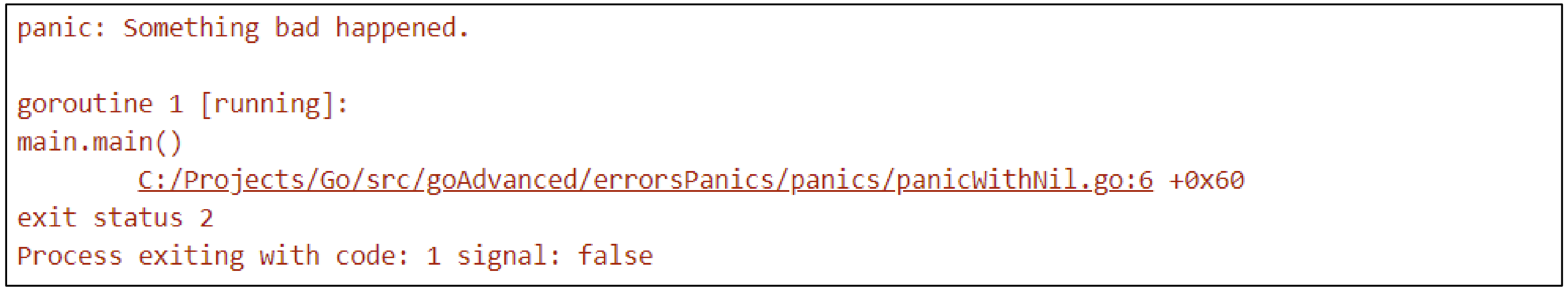
```go
package main

func main() {
	panic("Oops something happened...")
}
```

```
panic: Oops something happened...

goroutine 1 [running]:
main.main()
	C:/Projects/Go/src/goAdvanced/errorsPanics/panics/panicWithNil.go:4 +0x40
exit status 2
Process exiting with code: 1 signal: false
```

Panic System

- However, the idiomatic way is to pass in an error to a panic. Using error type also makes it easier for recovery later.

```go
package main

import "errors"

func main() {
        panic(errors.New("Something bad happened."))
}
```



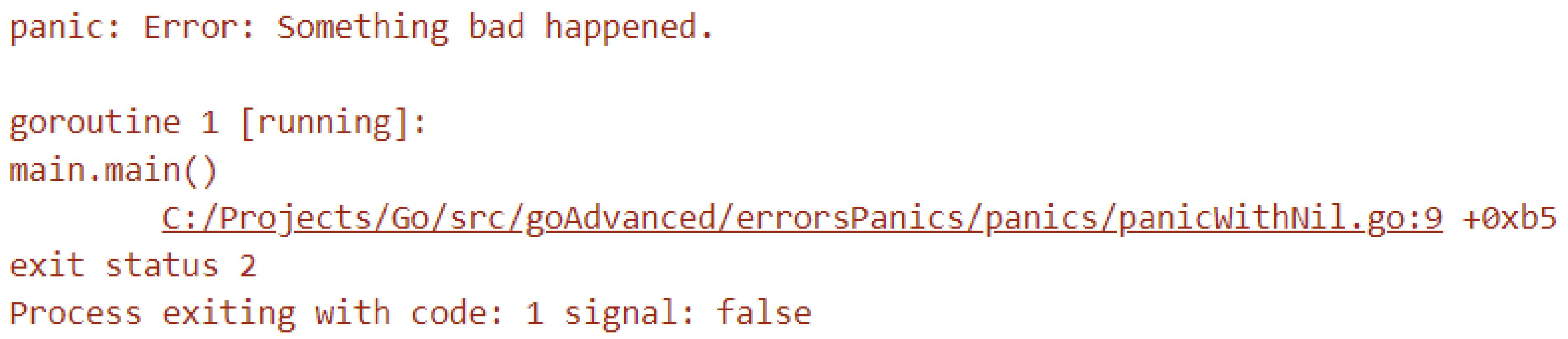
```
panic: Something bad happened.

goroutine 1 [running]:
main.main()
        C:/Projects/Go/src/goAdvanced/errorsPanics/panics/panicWithNil.go:6 +0x60
exit status 2
Process exiting with code: 1 signal: false
```

Panic System

- Passing in error also allows for formatting the panic message with print formatters:

```
package main

import (
        "fmt"
)

func main() {
        thePanic := "Something bad happened."
        panic(fmt.Errorf("Error: %s", thePanic))
}
```

Panic System



```
panic: Error: Something bad happened.

goroutine 1 [running]:
main.main()
        C:/Projects/Go/src/goAdvanced/errorsPanics/panics/panicWithNil.go:9 +0xb5
exit status 2
Process exiting with code: 1 signal: false
```

- However, do note that panics do not require error types. The panic system is meant to be flexible.
- Go panic is a way to unwind the call stack.

Panic System

Recovering from panics

- Panic Recovery in Go depends on deferred functions.
- *Deferred functions* – the execution of such function is guaranteed at the moment its parent function returns (be it due to return statement, end of function block or a panic).
- defer statement is good for closing files, sockets when done using, free up resources such as database handles or handle panics.
- General form for deferred function is:

```
func() {
       /* body*/
} ()
```

Panic System

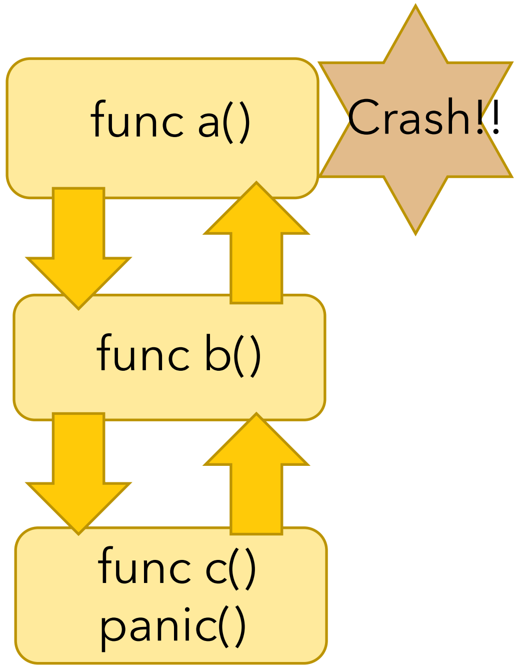
```
package main

import "fmt"

func main() {
        defer goodbye()

        fmt.Println("Hello world.")
}

func goodbye() {
        fmt.Println("Goodbye")
}
```

```
Hello world.
Goodbye
```

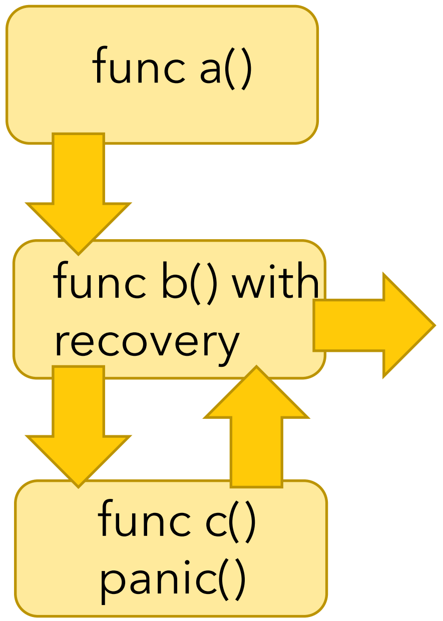
- Without defer statement, program would print “Goodbye” followed by “Hello world”. But defer keyword alters the order of execution.
- As the main function completes, the deferred goodbye function is run.

Panic System

Recovering from panics



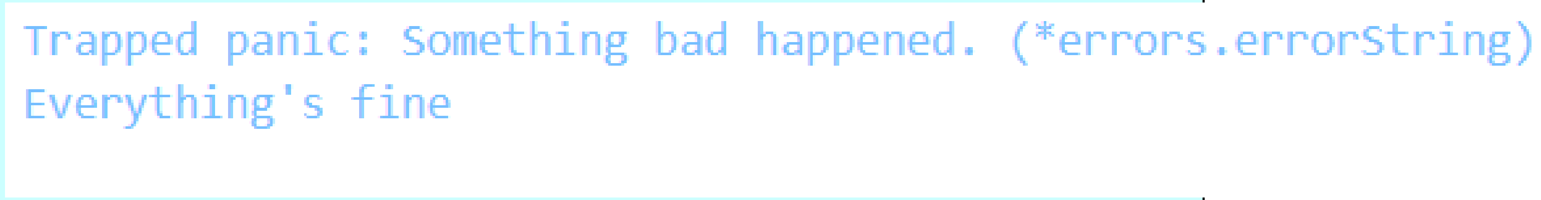
Unhandled panic



Panic with deferred recovery

Panic System

```
func main() {
        msg := "Everything's fine"
        defer func() {
                if err := recover(); err != nil {
                        fmt.Printf("Trapped panic: %s (%T)\n",
err, err)
                }
                fmt.Println(msg)
        }()

        yikes()
}

func yikes() {
        panic(errors.New("Something bad happened."))

}
```



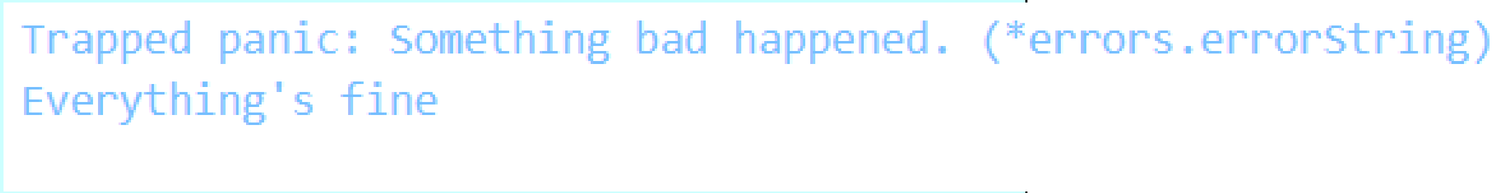
```
Trapped panic: Something bad happened. (*errors.errorString)
Everything's fine
```

Panic System

- Capturing a panic in a deferred function is standard practice in Go.
- *recover()* is called in deferred function to capture data about the panic.
- recover() returns a value (interface{}) if a panic has been raised, but in other cases returns nil.
- Value returned is what was passed into the panic.
- The value is then checked to find out what happened and determine how to handle the panic.
- Calling recover() also stops the panic from unwinding the function stack further, and allow the program to continue running.

Panic System

```
func main() {
        var msg string
        defer func() {
                if err := recover(); err != nil {
                        fmt.Printf("Trapped panic: %s
(%T)\n", err, err)
                }
                fmt.Println(msg)
        }()
        msg = "Everything's fine"

        yikes()
}

func yikes() {
        panic(errors.New("Something bad happened."))
}
```



```
Trapped panic: Something bad happened. (*errors.errorString)
Everything's fine
```

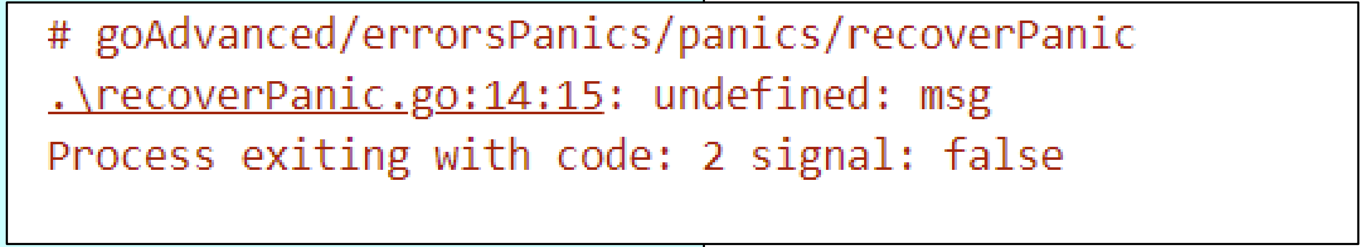
Panic System

- Closures inherit the scope of their parent. So deferred closures inherit whatever is in scope before its declaration.
- Therefore, variables defined before the closure, the closure may reference it.
- However, even though defer is executed after rest of the function, the closure does not have access to variables declared after declaration of the closure.
- Reason is, closure is evaluated upon declaration, though not executed until the function returns. At that point of declaration, those variables that are not declared yet, are undefined.

Panic System

```
func main() {
        defer func() {
                if err := recover(); err != nil {
                        fmt.Printf("Trapped panic: %s (%T)\n",
err, err)
                }
                fmt.Println(msg)
        }()

        msg := "Everything's fine"
        yikes()

}

func yikes() {
        panic(errors.New("Something bad happened."))
}
```

msg cannot be accessed, as it is declared after declaration of deferred closure!



```
# goAdvanced/errorsPanics/panics/recoverPanic
.\recoverPanic.go:14:15: undefined: msg
Process exiting with code: 2 signal: false
```

Panic System

- A more complete example that handles a panic and cleans up before returning:

```
func main() {
        var file io.ReadCloser
        file, err := OpenCSV("data.csv")
        if err != nil {
                fmt.Printf("Error: %s", err)
                return
        }
        defer file.Close()

        // Do something with file.

}
```

Panic System

```
func OpenCSV(filename string) (file *os.File, err error) {
        defer func() {
                if r := recover(); r != nil {
                        file.Close()
                        err = r.(error)
                }
        }()

        file, err = os.Open(filename)
        if err != nil {
                fmt.Printf("Failed to open file\n")
                return file, err
        }
```

NGEE ANN
SCHOOL OF INFOCOMM TECHNOLOGY

Panic System

```
        RemoveEmptyLines(file)

        return file, err
}

func RemoveEmptyLines(f *os.File) {
        panic(errors.New("Failed parse"))

}
```

```
Failed to open file
Error: open data.csv: The system cannot find the file specified.
```

Panic System

- Deferred function in example does 3 things:
  - Traps any panics
  - If a panic occurs, do all necessary things to handle panic
  - Get the error from panic, and pass it back to be handled properly by error handling mechanism.

Panic System

- Some useful guidelines for deferred functions:
  - Put deferred functions as close to top of function declaration.
  - Simple declarations placed before deferred functions.
  - More complex variables are declared before deferred functions, but only initialized until after.
  - Possible to declare multiple deferred functions inside one, but generally not recommended.
  - Recommend to close file, network connections, and other resources in defer clause. Ensures system resources freed even when errors or panics occur.

Learning Objectives – Mod 13

At the end of the course, participants should be able to:

- Define concurrency in Go.
- Examine the different events that leads up to concurrency issues in Go.
- Demonstrate the use of concurrency in Go.
- Define the use of Go Routines in concurrency.
- Examine the different concurrency properties.
- Demonstrate the use of concurrency.

Concurrency in Go

- Ability to execute several tasks concurrently can dramatically improve the performance of a system.
- Support for concurrency is built directly into Go's language and runtime.
- Concurrency in Go is ability for functions to run independent of each other.
- When a function is created as a goroutine, it is regarded as an independent unit of work, that gets scheduled, and then execute on an available logical processor.
- *Go runtime scheduler* manages all the goroutines. It sits on top of the OS, binding OS's threads to logical processors, and controls which goroutines run on which logical processor at a given time.

Concurrency in Go

- Go concurrency synchronization comes from a paradigm called, *Communicating Sequential Processes (or CSP).*
- CSP is a message-passing model, involving communicating data between go routines (instead of locking data to synchronize access)
- Key data type for synchronizing and passing messages between goroutines is called a *channel*.
- Using channels makes it easier to write concurrent programs and make them less prone to errors.

Concurrency ≠ Parallelism

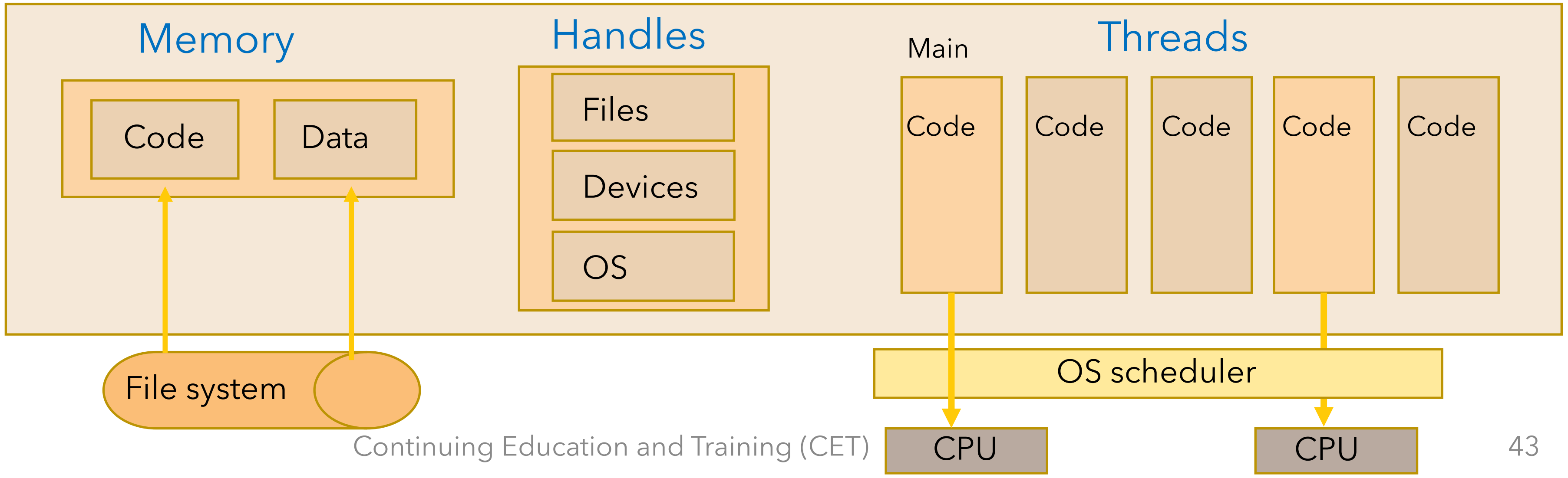
Parallelism – Doing lotsa things at once.
Concurrency – Managing lotsa things at once.

Concurrency vs Parallelism

What are OS processes and threads?

- When you run an application, OS starts a thread.
- A *process* - like a container that holds all the resources that the application uses and maintains as it runs.
- The resources include a memory address space, handles to files, devices and threads.

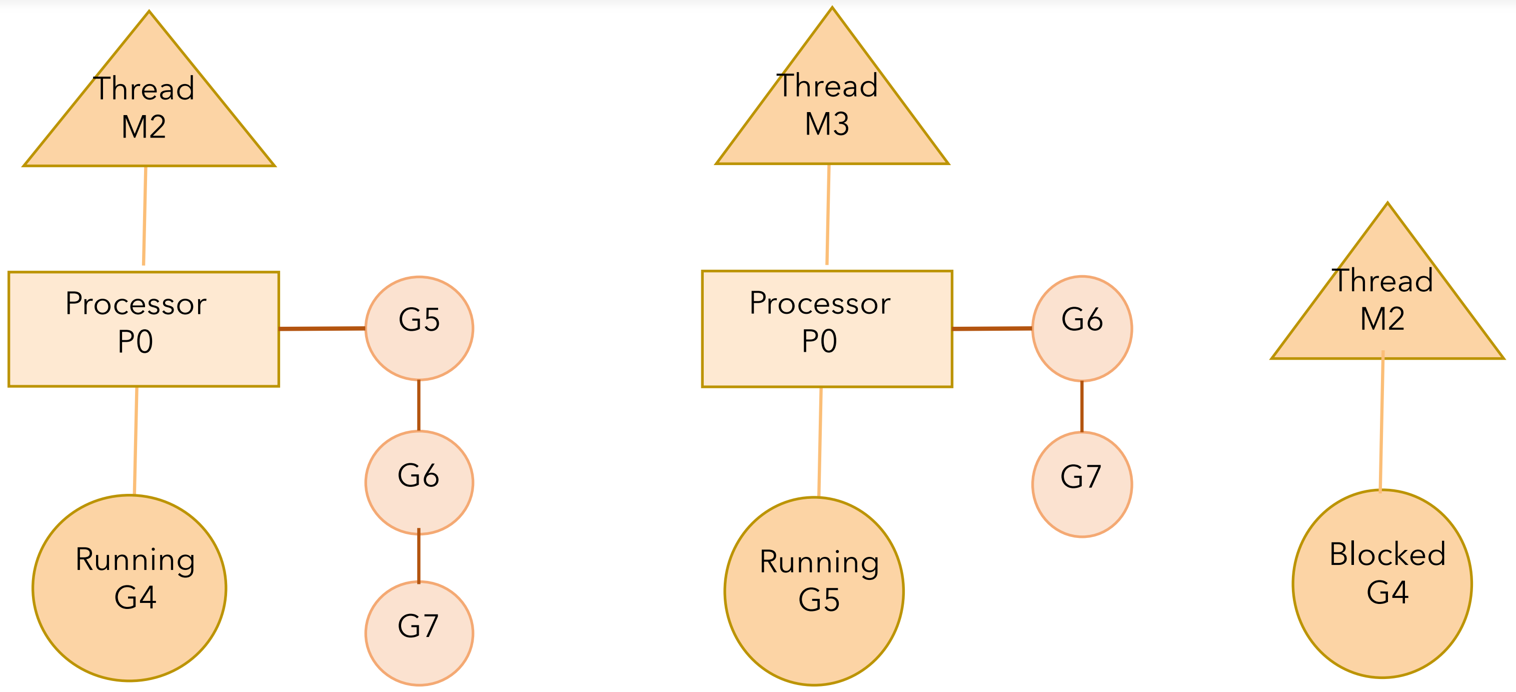
A process and its threads for a running application.



Concurrency vs Parallelism

- A *thread* – a path of execution scheduled by the OS to run the code that you write in your functions.
- Each process contains at least one thread: the main thread, the initial thread for each process. When main thread terminates, the application terminates.
- The OS schedules threads to run against physical processors, regardless the process they belong to.
- Go runtime schedules *goroutines* to run against logical processors.
- Each logical processor bound to a single OS thread. As of version 1.5, the default is to allocate a logical processor for every physical processor that is available.

Concurrency vs Parallelism



How the Go scheduler manages goroutines

Concurrency vs Parallelism

How go runtime scheduler manages goroutines:

- As goroutines are created and ready to run, they are placed in the scheduler's global run queue.
- Then, they are assigned a logical processor and placed in a local run queue for that logical processor. The goroutine waits its turn to be given logical processor for execution.
- If a running goroutine needs to perform a *blocking* syscall, the thread and the goroutine are detached from logical processor, and thread continues to block waiting for syscall to return.
- Meanwhile, scheduler creates a new thread and attaches it to the logical processor.

Concurrency vs Parallelism

- Then scheduler chooses another goroutine from local run queue for execution.
- Once syscall returns, the goroutine is placed back into the local run queue, and thread is put aside for future use.
- If goroutine needs to make a network I/O call, goroutine is detached from the logical processor and moved to runtime integrated network poller.
- Once poller indicates read/ write operation is ready, goroutine assigned back to a logical processor to handle the operation.

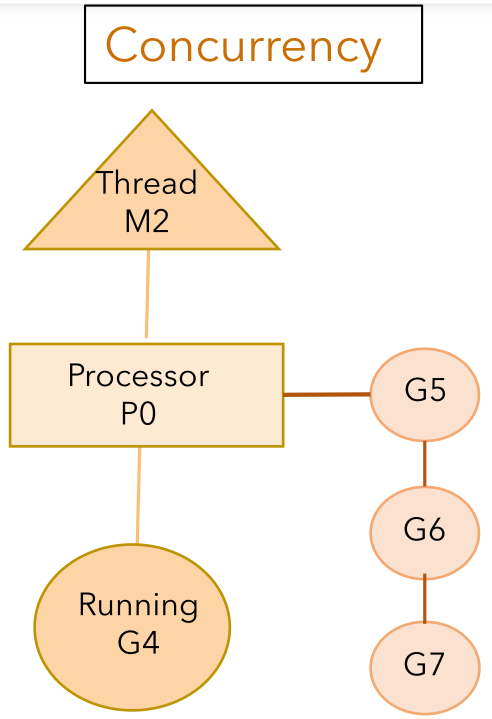
Concurrency vs Parallelism

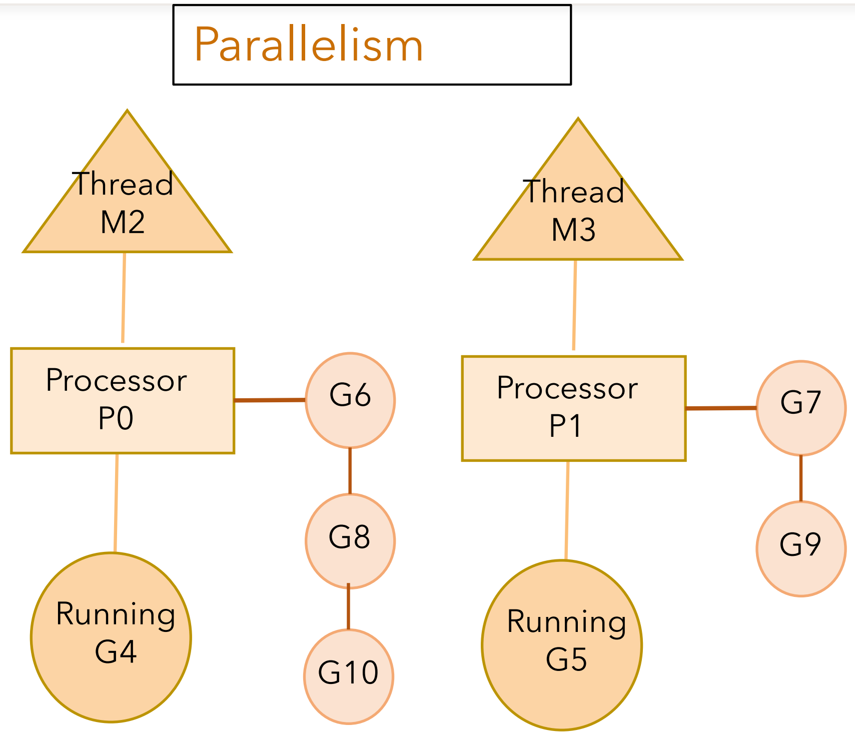
- Any limit to number of logical processors that can be created? NO!
- But by default, runtime limits each program to max to 10000 threads by default. This can be changed by calling the SetMaxThreads() function from runtime/debug package.
- The program crashes, if it attempts to use more threads.

Concurrency vs Parallelism

- Concurrency is not parallelism.
- *Parallelism* only achieved if multiple pieces of code are executing simultaneously against different physical processors.
- If wish to run goroutines in parallel, must use more than 1 logical processor.
- When there are multiple logical processors, the scheduler will evenly distribute goroutines between the logical processors. This will result in goroutines running on different threads.
- But for true parallelism, still need to run a program on a machine with different physical processors. If not, the goroutines will still run concurrently against a single physical processor, even if Go runtime is using multiple threads.

Concurrency vs Parallelism





Difference between concurrency and parallelism

Goroutines

- *Goroutine* – a function that runs independently of the function that started it. (Sometimes, some Go developers explain goroutine as a function that runs as if it were on its own thread)
- In Go, it is represented as any function that is called after the special keyword ***go***.
- Any function can be executed as a goroutine.
- Parameters can be passed into function created as a goroutine.
- Return parameters however not available when goroutine terminates.
- Most frequent use is to run a function "in the background" while the main part of the program goes on to do something else.

Goroutines

```
package main

import (
        "fmt"
        "runtime"
        "sync"
)

func main() {

        runtime.GOMAXPROCS(1)

        // waitgroup is counting semaphore to maintain record of
running goroutines.
        var wg sync.WaitGroup
        wg.Add(2)
```

Goroutine

```
fmt.Println("Start Goroutines")

// Declare an anonymous function and create a goroutine.
go func() {
        //Tell main we are done.
        defer wg.Done()

        for count := 0; count < 3; count++ {
                for char := 'a'; char < 'a'+26; char++ {
                        fmt.Printf("%c ", char)
                }
        }
}()
```

Goroutines

```
// Declare an anonymous function and create a goroutine.
go func() {
        //Tell main we are done.
        defer wg.Done()

        // Display the alphabet three times
        for count := 0; count < 3; count++ {
                for char := 'A'; char < 'A'+26; char++ {
                        fmt.Printf("%c ", char)
                }
        }
}()
```

Goroutines

```
        // Wait for the goroutines to finish.
        fmt.Println("Waiting To Finish")
        wg.Wait() //continue to block as long as value of wg is > 0

        fmt.Println("\nTerminating Program")
}
```

```
Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U V W X Y
Z a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x y z a b c d e f g h i j k l m n o p q r s t u v w x
y z
Terminating Program
```

Goroutines

- Call to the GOMAXPROCS() function from runtime package allows changing of number of logical processors to be used by the scheduler for this program. (Can also set an environment variable with same name if not making this call in the program)

- Functions are created as goroutines by using keyword ***go***.

- Once functions are created as goroutines, the code in main keeps running. So main function can return and program terminate, before goroutines complete their work, or even get the chance to run.
- Solution: use ***WaitGroup*** to wait for goroutines to complete their work.

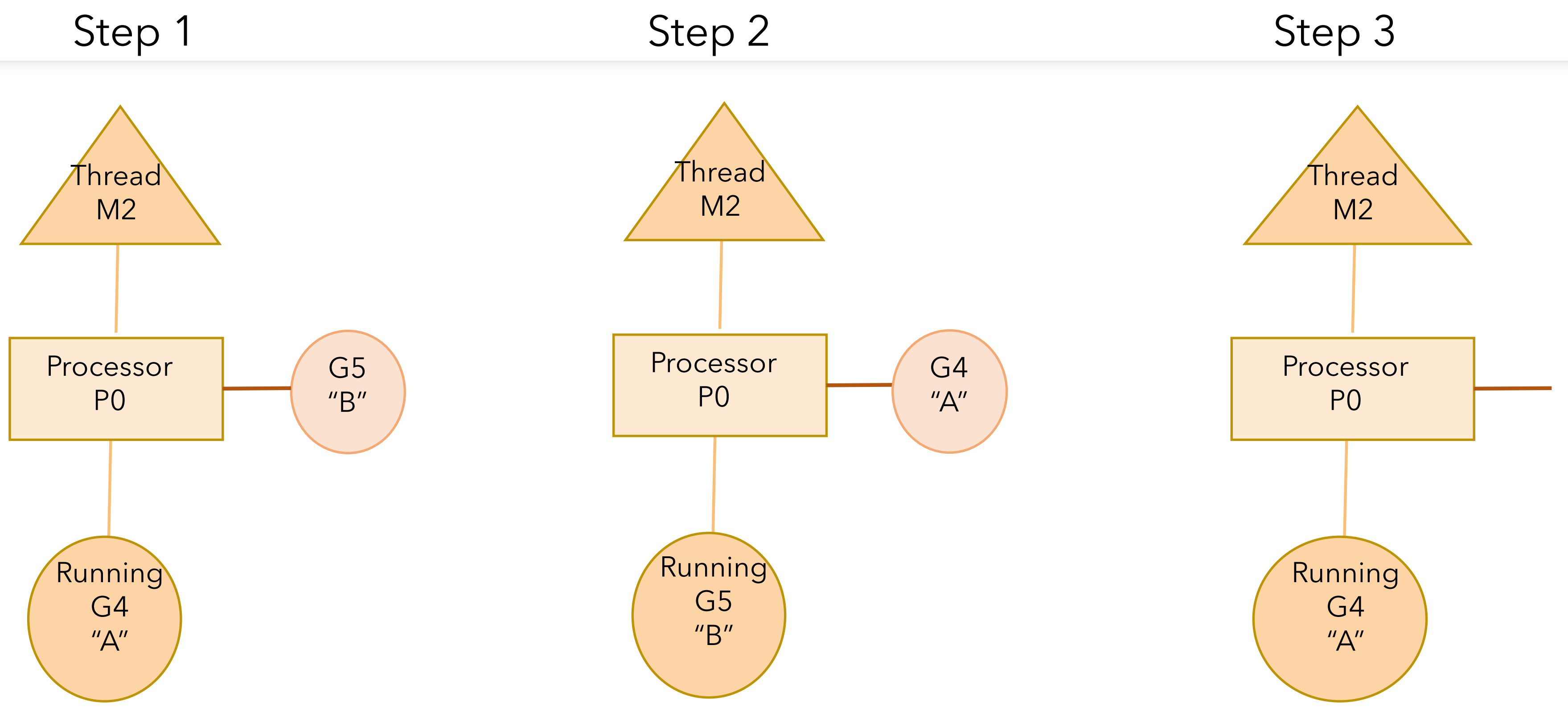
Goroutines

- ***WaitGroup*** is a ***counting semaphore*** to maintain a record of running goroutines.
- When WaitGroup > 0, Wait() method will block.
- To use it, basically tell it when it needs to wait for some task, and signal to it when task is done.
- WaitGroup only needs to know the number of tasks it is waiting for, and when each task is done.
- It can be incremented with Add(), when task is completed, signal this with Done().
- defer used to guarantee the call to Done() is made once each goroutine is done with this work and return.

Goroutines

- The scheduler usually has internal algorithm to make sure that a running goroutine can be stopped (and give another running goroutine a chance to run) and rescheduled again before it finishes its work.
- This is to prevent any single routine from holding the logical processor hostage.

Goroutines



Time-slicing, and swapping in-and-out of goroutines executing on the logical processor's thread

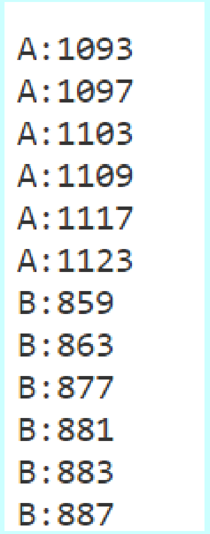
Goroutine

```
import (
        "fmt"
        "runtime"
        "sync"
)

var wg sync.WaitGroup

func main() {
        runtime.GOMAXPROCS(1)

        // Add a count of two, one for each goroutine.
        wg.Add(2)

        // Create two goroutines.
        fmt.Println("Create Goroutines")
        go printPrime("A")
        go printPrime("B")
```

Goroutines

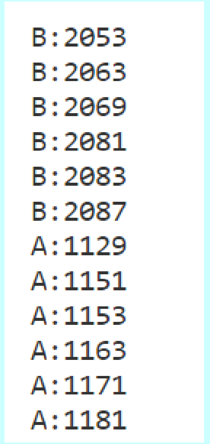
```
        // Wait for the goroutines to finish.
        fmt.Println("Waiting To Finish")
        wg.Wait()

        fmt.Println("Terminating Program")
}

// printPrime displays prime numbers for first 5000 numbers.
func printPrime(prefix string) {
        // call to Done to inform main
        defer wg.Done()

next:
        for outer := 2; outer < 5000; outer++ {
                for inner := 2; inner < outer; inner++ {
                        if outer%inner == 0 {
                                continue next
                        }
                }
                fmt.Printf("%s:%d\n", prefix, outer)
        }
        fmt.Println("Completed", prefix)
}
```

Run and see the output!



```
A:1093
A:1097
A:1103
A:1109
A:1117
A:1123
B:859
B:863
B:877
B:881
B:883
B:887
```

.
.
.



```
B:2053
B:2063
B:2069
B:2081
B:2083
B:2087
A:1129
A:1151
A:1153
A:1163
A:1171
A:1181
```

Goroutines

- So far the goroutines run concurrently on a single logical processor. What if now, we want to run our goroutines in parallel?
- Recall GOMAXPROCS from runtime package can be set, to specify the number of logical processors used.
- But what if we want to allocate a logical processor for every physical processor?

```
import (
        "fmt"
        "runtime"
        "sync"
)

Runtime.GOMAXPROCS(runtime.NumCPU())
```

Goroutines

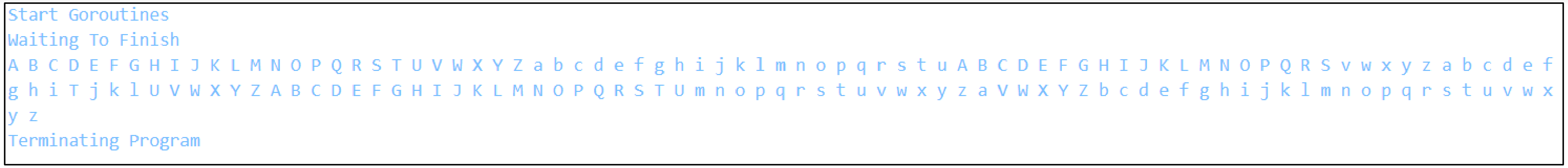
- NumCPU() returns the number of physical processor that are available.
- So the call creates a logical processor for each physical processor. (Actually more than one logical processor does not mean better performance…)

Goroutines

- Now let’s try giving our letter printing example more than 1 logical processor to use…allowing the goroutines to run in parallel…

```
package main

import (
        "fmt"
        "runtime"
        "sync"
)

func main() {

        runtime.GOMAXPROCS(2)

        // waitgroup is counting semaphore to maintain record of running goroutines.
        var wg sync.WaitGroup
        wg.Add(2)
```

Goroutines



```
Start Goroutines
Waiting To Finish
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u A B C D E F G H I J K L M N O P Q R S v w x y z a b c d e f
g h i T j k l U V W X Y Z A B C D E F G H I J K L M N O P Q R S T U m n o p q r s t u v w x y z a V W X Y Z b c d e f g h i j k l m n o p q r s t u v w x
y z
Terminating Program
```

- Notice the letters in display are now mixed! If you run a multi-core machine, the more processors used, the mixed up the letters will be!

- But note goroutines can only run in parallel when there is more than 1 logical processor and there is physical processor available to run each goroutine at same time.

Race conditions

- *Race condition* – 2 or more goroutines have unsynchronized access to a shared resource and attempt to read and write to that resource at the same time.
- Read/write operations against a shared resource must always be *atomic*, meaning done by only 1 goroutine.

Race conditions

```
package main

import (
        "fmt"
        "runtime"
        "sync"
)

var (

        counter int

        // wg is used to wait for the program to finish.
        wg sync.WaitGroup
)


func main() {
        // Add a count of two, one for each goroutine.
        wg.Add(2)
```

Race conditions

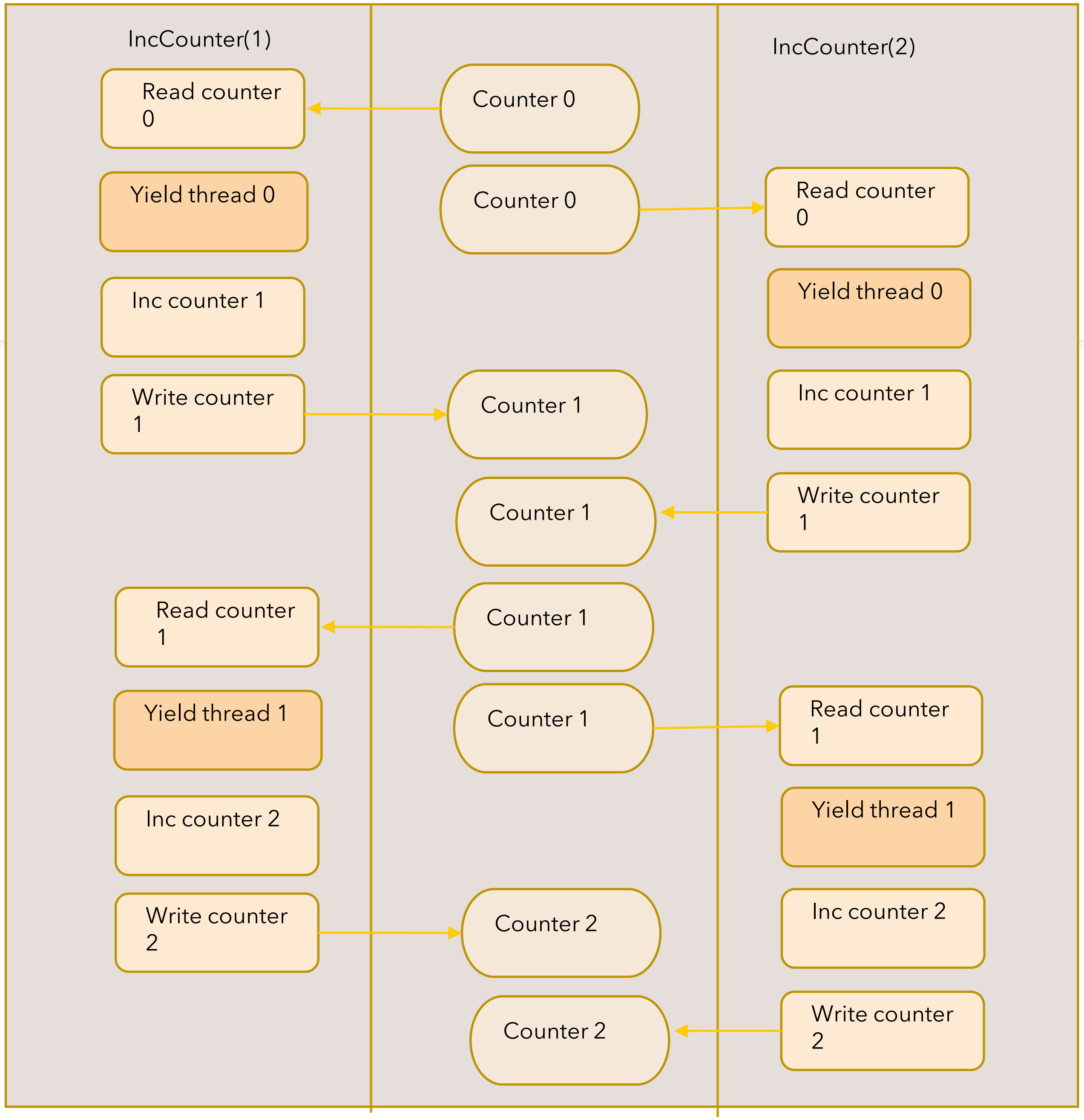
```
        // Create two goroutines.
        go incCounter(1)
        go incCounter(2)

        wg.Wait()
        fmt.Println("Final Counter:", counter)
}
```

Race conditions

```
func incCounter(id int) {
        // Schedule the call to Done to tell main we are done.
        defer wg.Done()

        for count := 0; count < 2; count++ {

                value := counter

                // Yield thread and be placed back in queue.
                runtime.Gosched()

                value++

                counter = value
        }
}
```

```
Final Counter: 2
```

Race conditions



Race conditions

- Goshed() function from runtime is to yield the thread and give other goroutines a chance to run.
- (In this case, this is called to force the scheduler to swap between 2 goroutine, to exaggerate the effects of the race condition.)

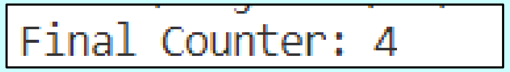
Synchronization

- To eliminate the race condition, Go has traditional support for synchronising goroutines by locking down shared resources.
- There are two possible ways:
    - *Atomic functions*
    - *Mutex*

Synchronization

- *Atomic functions* - provide low-level mechanism for synchronizing access to integers and pointers. The functions are from the atomic package

```
import (...
        ...

        "sync/atomic"
)
...
func incCounter(id int) {

        defer wg.Done()

        for count := 0; count < 2; count++ {

                // Safely Add One To Counter.
                atomic.AddInt64(&counter, 1)

                runtime.Gosched()
        }
}
```



```
Final Counter: 4
```

Synchronization

- When goroutines attempt to call any atomic function, it is automatically synchronized against the variable that is referenced.

- func AddInt64(addr *int64, delta int64) (new int64)
- AddInt64() atomically adds delta to *addr and returns the new value.
- It enforces only one goroutine can complete the add operation at a time.

- Two other functions LoadInt64() and StoreInt64(). It provides a safe way to read and write from integer.

Synchronization

- Another example…

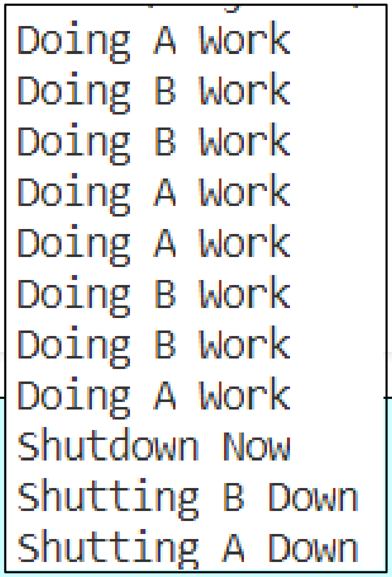
```
package main

import (
        "fmt"
        "sync"
        "sync/atomic"
        "time"
)

var (
        // shutdown is a flag to alert running goroutines to shutdown.
        shutdown int64

        wg sync.WaitGroup
)
```

Synchornization

```
func main() {

        wg.Add(2)

        go doWork("A")
        go doWork("B")

        time.Sleep(1 * time.Second)


        fmt.Println("Shutdown Now")
        atomic.StoreInt64(&shutdown, 1)

        wg.Wait()
}
```

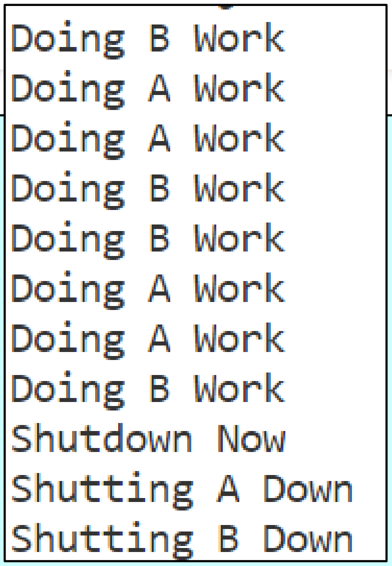
Synchronization

```
func doWork(name string) {

        defer wg.Done()

        for {
                fmt.Printf("Doing %s Work\n", name)
                time.Sleep(250 * time.Millisecond)

                // Do we need to shutdown.
                if atomic.LoadInt64(&shutdown) == 1 {
                        fmt.Printf("Shutting %s Down\n", name)
                        break
                }
        }
}
```



```
Doing A Work
Doing B Work
Doing B Work
Doing A Work
Doing A Work
Doing B Work
Doing B Work
Doing A Work
Shutdown Now
Shutting B Down
Shutting A Down
```

Different runs of the program



```
Doing B Work
Doing A Work
Doing A Work
Doing B Work
Doing B Work
Doing A Work
Doing A Work
Doing B Work
Shutdown Now
Shutting A Down
Shutting B Down
```

Synchronization

- *Mutex (Mutual exclusion)*- creates a *critical section* around code, ensuring that only one goroutine can execute that code each time.
- Operations protected within a critical section defined by Lock() and Unlock(). (the braces is just for readability, but not a must)
- Only one goroutine can enter the critical section.
- Only when a call to Unlock() is made, then another goroutine can enter the critical section.

Synchronization

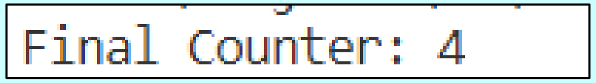
```
package main

import (
        "fmt"
        "runtime"
        "sync"
)

var (
        counter int

        wg sync.WaitGroup

        // mutex to define a critical section of code.
        mutex sync.Mutex
)
```

Synchronization

```
func main() {

        wg.Add(2)

        go incCounter(1)
        go incCounter(2)

        wg.Wait()
        fmt.Printf("Final Counter: %d\n", counter)
}
```

Synchronization

```
func incCounter(id int) {

        defer wg.Done()

        for count := 0; count < 2; count++ {

                mutex.Lock()
                {

                        value := counter

                        runtime.Gosched()

                        value++

                        counter = value
                }
                mutex.Unlock()

        }
}
```



```
Final Counter: 4
```

Channels

- Thinking of writing concurrent programs in easier manner, less error-prone or even fun way? Perhaps not atomic functions and mutexes…

- In Go, there are also *channels* that synchronize goroutines. This is done as they send and receive shared resources.

- When a resource needs to be shared between goroutines, channels act as a conduit between the goroutines. It also provides a mechanism that guarantees synchronous exchange.

Channels

- When declaring a channel, type of data shared needs to be stated.
- Values, pointers of built-in, named, struct and reference types all can be shared through a channel.
- Creating a channel needs to use built-in function make and keyword chan

```
//Unbuffered channel of integers
unbuffered := make (chan int)

//Buffered channel of 10 strings
buffered := make (chan string, 10)
```

- *Unbuffered* and *buffered channels* have different behavior. Important to know the differences to know when to use which of them.

Channels

- Sending a value or pointer into a channel requires use of <- operator.

```
//Buffered channel of strings
buffered := make (chan string, 10)

//Send a string via the channel
buffered <- “Gophercon”
```

- Receiving data from the channel using <- operator

```
//Receive a string from the channel.
<- buffered
```

- When receiving a value or pointer from a channel, notice <- operator is attached to left side of the channel variable.

Channels

- Assigning data received from channel via <- operator to a variable, using assignment operator = , or to a variable that is not declared before using short hand declaration :=

```
//Receive a string from the channel.
value := <- buffered
```

- Channels are bidirectional by default.
- However a "direction" can be specified for the channel.

```
var bidirectionalChan chan string // can read from, write to and close()
var receiveOnlyChan <-chan string // can read from, but cannot write to or close()
var sendOnlyChan chan<- string    // cannot read from, but can write to and close()
```

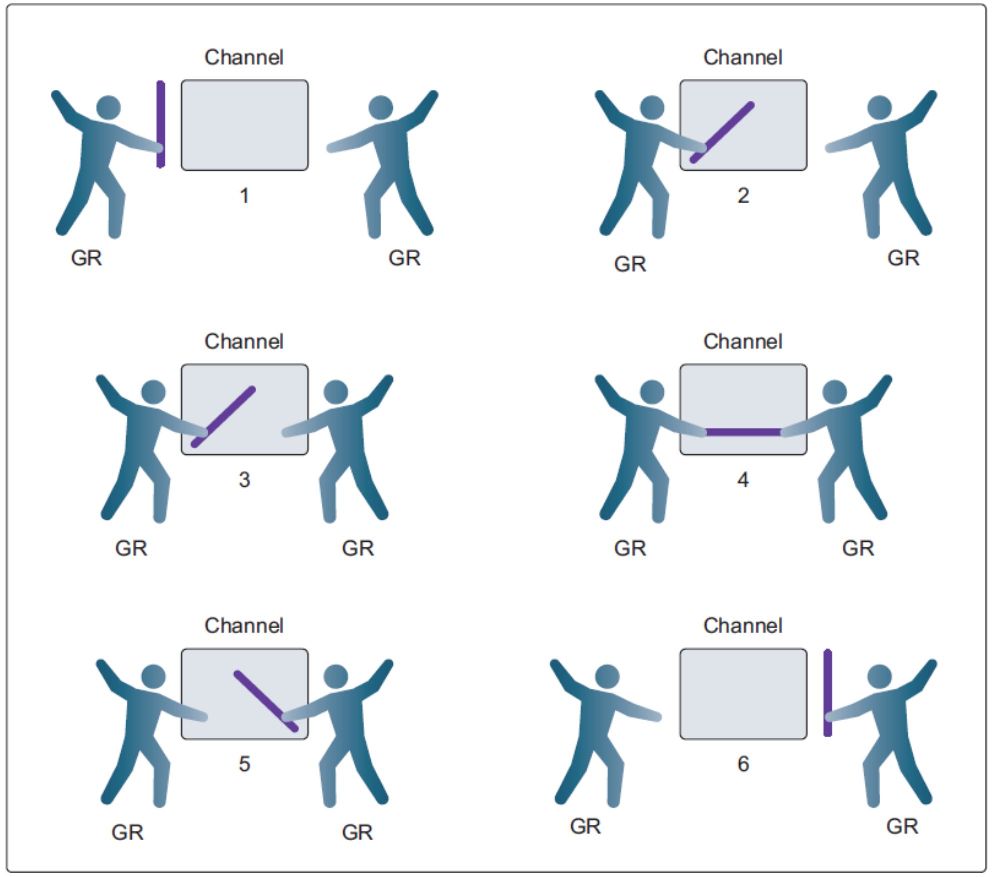
Unbuffered Channels

A channel with no capacity to hold any value before it is received.

- Requires both sending and receiving goroutine to be ready at same instance before any send or receive can complete.
- If the two goroutines are not ready at the same time, channel makes the goroutine that performs send or receive first, wait.
- Synchronization is thus built-in in this kind of channel interaction!

Unbuffered channels provide guarantee that exchange between two goroutines is performed at the time send and receive take place.

Unbuffered Channels



W. Kennedy, B. Ketelsen, E. St. Martin,,
Go in Action, 2016,

Synchronization between goroutines using unbuffered channel

Unbuffered Channels

```
package main

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

var wg sync.WaitGroup

func init() {
        rand.Seed(time.Now().UnixNano())
}
```

Unbuffered Channels

```go
func main() {
        // An unbuffered channel.
        court := make(chan int)

        wg.Add(2)

        go player("Vladmir", court)
        go player("Jaine", court)

        // Start
        court <- 1

        wg.Wait()
}
```

Unbuffered Channels

```
func player(name string, court chan int) {

        defer wg.Done()

        for {
                // Wait for ball to be hit back.
                ball, ok := <-court
                if !ok {
                        // If the channel was closed, won.
                        fmt.Printf("Player %s Won\n", name)
                        return
                }

                // Simulate missing the ball.
                n := rand.Intn(100)
                if n%13 == 0 {
                        fmt.Printf("Player %s Missed\n", name)
```

Unbuffered Channels

```
                // Close the channel to signal we lost.
                close(court)
                return
            }

            // Display and then increment the hit count by one.
            fmt.Printf("Player %s Hit %d\n", name, ball)
            ball++

            // Hit the ball back to the opposing player.
            court <- ball
        }
}
```

```
Player Jaine Hit 1
Player Vladmir Hit 2
Player Jaine Hit 3
Player Vladmir Hit 4
Player Jaine Hit 5
Player Vladmir Hit 6
Player Jaine Hit 7
Player Vladmir Hit 8
Player Jaine Hit 9
Player Vladmir Missed
Player Jaine Won
```

Unbuffered Channels

- Another example…

```
// A relay race between four goroutines.
package main

import (
        "fmt"
        "sync"
        "time"
)

var wg sync.WaitGroup

func main() {

        baton := make(chan int)

        wg.Add(1)
```

Unbuffered Channels

```
        // First runner to his mark.
        go Runner(baton)

        // Start the race.
        baton <- 1

        wg.Wait()
}

// Runner simulates a person running in the relay race.
func Runner(baton chan int) {
        var newRunner int

        // Wait to receive the baton.
        runner := <-baton

        fmt.Printf("Runner %d Running With Baton\n", runner)
```

Unbuffered Channels

```
if runner != 4 {
        newRunner = runner + 1
        fmt.Printf("Runner %d To The Line\n", newRunner)
        go Runner(baton)
}

// Running around the track.
time.Sleep(100 * time.Millisecond)

// Is the race over.
if runner == 4 {
        fmt.Printf("Runner %d Finished, Race Over\n", runner)
        wg.Done()
        return
}
```

Unbuffered Channels

```
    // Exchange the baton for the next runner.
    fmt.Printf("Runner %d Exchange With Runner %d\n",
        runner,
        newRunner)

    baton <- newRunner
}
```

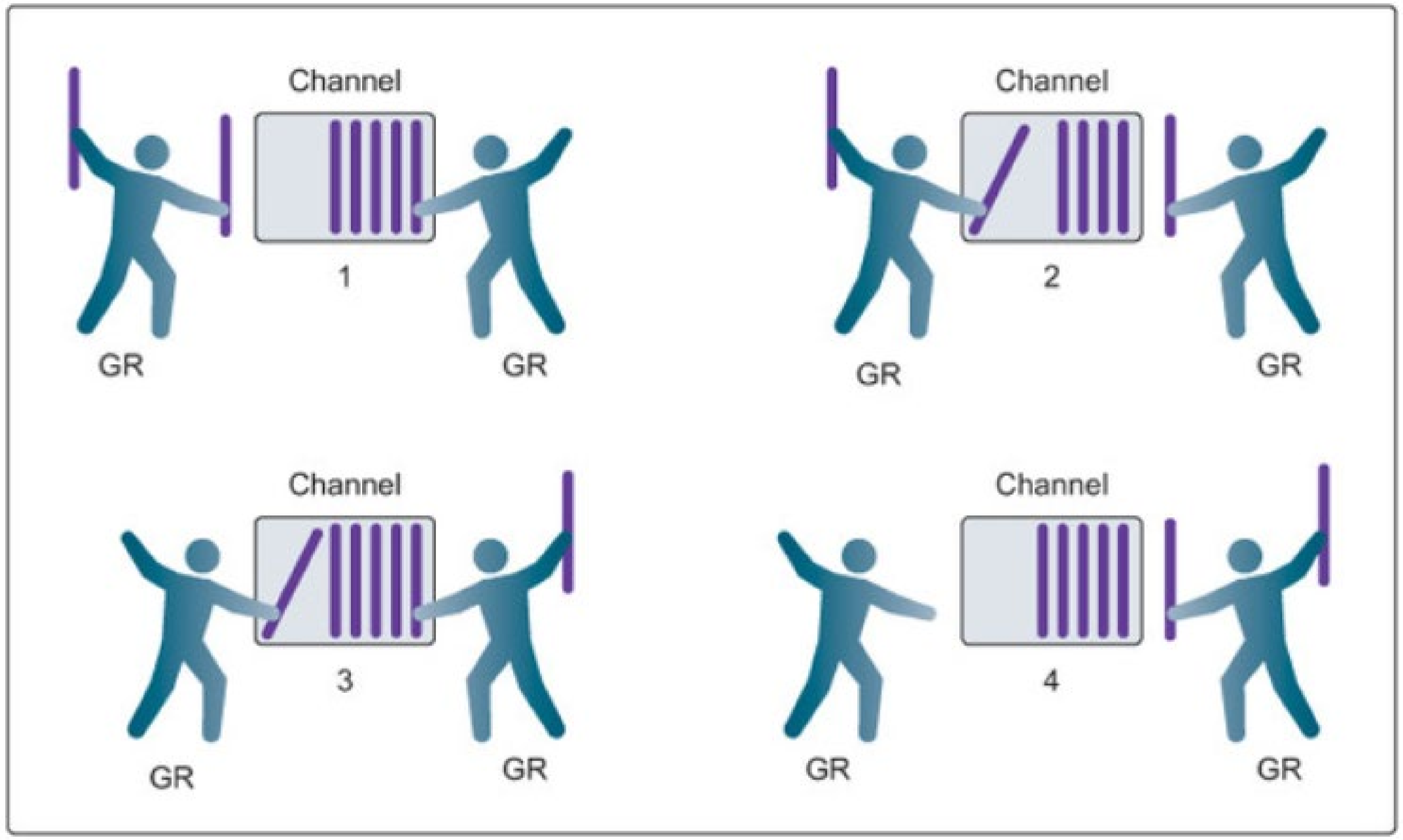
```
Runner 1 Running With Baton
Runner 2 To The Line
Runner 1 Exchange With Runner 2
Runner 2 Running With Baton
Runner 3 To The Line
Runner 2 Exchange With Runner 3
Runner 3 Running With Baton
Runner 4 To The Line
Runner 3 Exchange With Runner 4
Runner 4 Running With Baton
Runner 4 Finished, Race Over
```

Buffered Channels

- A channel with capacity to hold one or more values, before they are received.
- Do not force the goroutines to be ready at same instance to perform sends and receives.
- There are however also different conditions for when a send or receive does block.
- A receive will block only if there is no value in the channel to receive.
- A send will block only if there is no available buffer to place value that is going to be sent.

Unlike unbuffered channels, buffered channels DO NOT provide guarantee that exchange between two goroutines.

Buffered Channels



W. Kennedy, B. Ketelsen, E. St. Martin,,
Go in Action, 2016,

Synchronization between goroutines using buffered channel

Buffered Channels

```
package main

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

const (
	numberGoroutines = 4  // Number of goroutines to use.
	taskLoad         = 10 // Amount of work to process.
)

// wg is used to wait for the program to finish.
var wg sync.WaitGroup
```

Buffered Channels

```
func init() {
        rand.Seed(time.Now().Unix())
}

func main() {
        // Create a buffered channel to manage the task load.
        tasks := make(chan string, taskLoad)

        wg.Add(numberGoroutines)
        for gr := 1; gr <= numberGoroutines; gr++ {
                go worker(tasks, gr)
        }

        // Add a bunch of work to get done.
        for post := 1; post <= taskLoad; post++ {
                tasks <- fmt.Sprintf("Task : %d", post)
        }
```

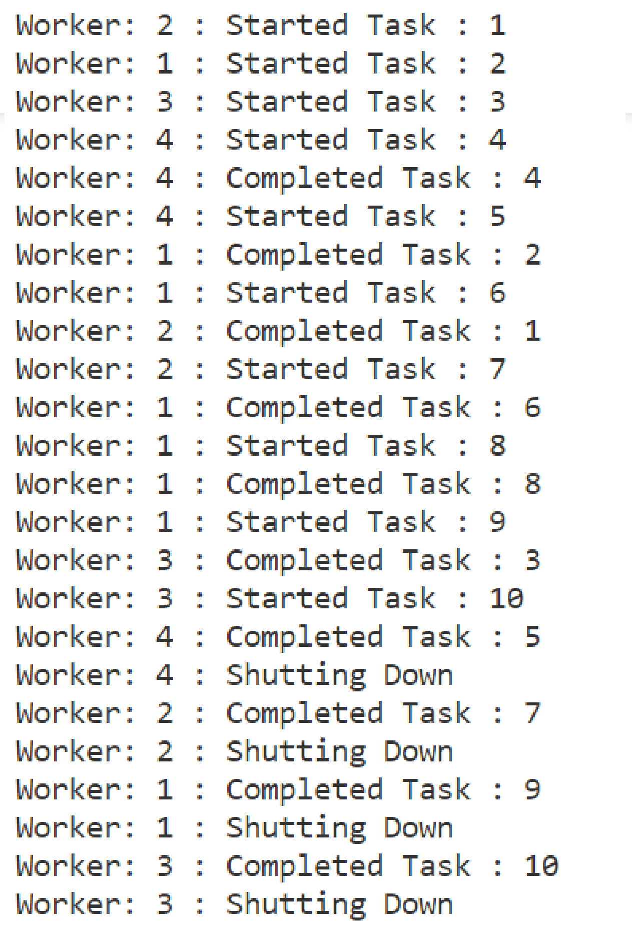
Buffered Channels

```
        // Close the channel so the goroutines will quit
        // when all the work is done.
        close(tasks)

        // Wait for all the work to get done.
        wg.Wait()
}

// worker is launched as a goroutine to process work from
// the buffered channel.
func worker(tasks chan string, worker int) {
        // Report that we just returned.
        defer wg.Done()

        for {
                // Wait for work to be assigned.
                task, ok := <-tasks
```

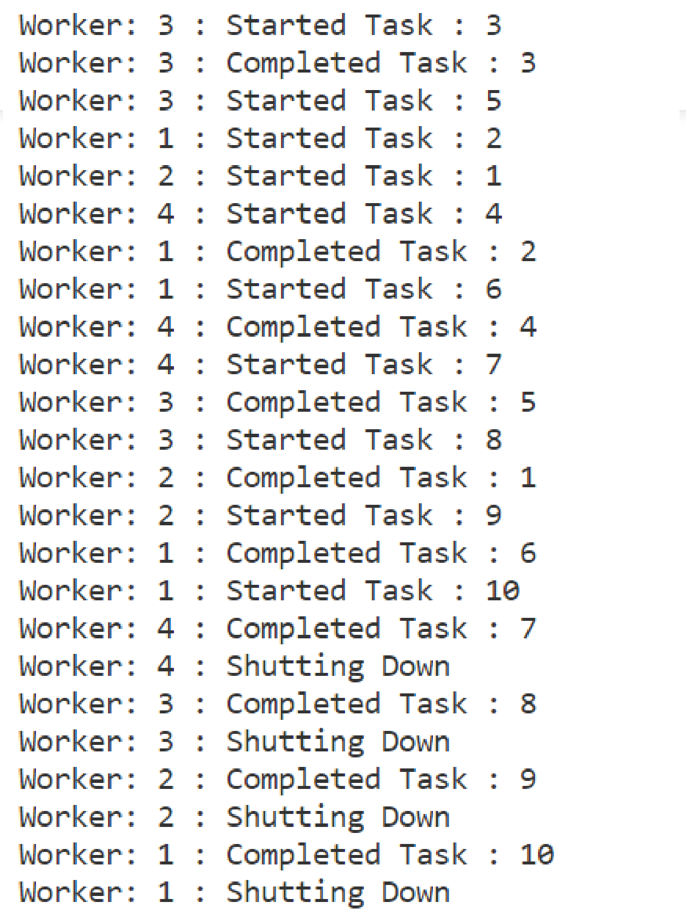
Buffered Channels

```
                    if !ok {
                            // This means the channel is empty and closed.
                            fmt.Printf("Worker: %d : Shutting Down\n", worker)
                            return
                    }

                    // Display we are starting the work.
                    fmt.Printf("Worker: %d : Started %s\n", worker, task)

                    // Randomly wait to simulate work time.
                    sleep := rand.Int63n(100)
                    time.Sleep(time.Duration(sleep) * time.Millisecond)

                    // Display we finished the work.
                    fmt.Printf("Worker: %d : Completed %s\n", worker, task)
            }
}
```

Buffered Channels



```
Worker: 2 : Started Task : 1
Worker: 1 : Started Task : 2
Worker: 3 : Started Task : 3
Worker: 4 : Started Task : 4
Worker: 4 : Completed Task : 4
Worker: 4 : Started Task : 5
Worker: 1 : Completed Task : 2
Worker: 1 : Started Task : 6
Worker: 2 : Completed Task : 1
Worker: 2 : Started Task : 7
Worker: 1 : Completed Task : 6
Worker: 1 : Started Task : 8
Worker: 1 : Completed Task : 8
Worker: 1 : Started Task : 9
Worker: 3 : Completed Task : 3
Worker: 3 : Started Task : 10
Worker: 4 : Completed Task : 5
Worker: 4 : Shutting Down
Worker: 2 : Completed Task : 7
Worker: 2 : Shutting Down
Worker: 1 : Completed Task : 9
Worker: 1 : Shutting Down
Worker: 3 : Completed Task : 10
Worker: 3 : Shutting Down
```



```
Worker: 3 : Started Task : 3
Worker: 3 : Completed Task : 3
Worker: 3 : Started Task : 5
Worker: 1 : Started Task : 2
Worker: 2 : Started Task : 1
Worker: 4 : Started Task : 4
Worker: 1 : Completed Task : 2
Worker: 1 : Started Task : 6
Worker: 4 : Completed Task : 4
Worker: 4 : Started Task : 7
Worker: 3 : Completed Task : 5
Worker: 3 : Started Task : 8
Worker: 2 : Completed Task : 1
Worker: 2 : Started Task : 9
Worker: 1 : Completed Task : 6
Worker: 1 : Started Task : 10
Worker: 4 : Completed Task : 7
Worker: 4 : Shutting Down
Worker: 3 : Completed Task : 8
Worker: 3 : Shutting Down
Worker: 2 : Completed Task : 9
Worker: 2 : Shutting Down
Worker: 1 : Completed Task : 10
Worker: 1 : Shutting Down
```

Two runs of the program. Due to random nature of the program and Go scheduler, the output for program is most likely different each time the program is run.

Locking with Buffered Channels

- We know that channels allow for communication between goroutines.
- However it is also sometimes desirable to use channels to do ***locking*** instead of using mutex, especially if the program already using channels.
- When using a channel as a lock, we want to have this behavior:
    - A function acquires a lock by sending a message on the channel
    - Function proceeds to do its sensitive operations.
    - The function releases the lock by reading message off the channel.
    - Any function that tries to acquire the lock before it is being released will be blocked when it tries to acquire the lock.

Locking with Buffered Channels

- A buffered channel does not block on send provided that buffer space still exist.
- A sender can send a message into the buffer and then move on. But buffer is full, sender will block until there is room in the buffer for it to send its message.
- And this is exactly what we want in a lock! So a channel with only 1 empty buffer space.

- So one function send a message, do its task, and then read the message off the buffer (thus unlocking it).

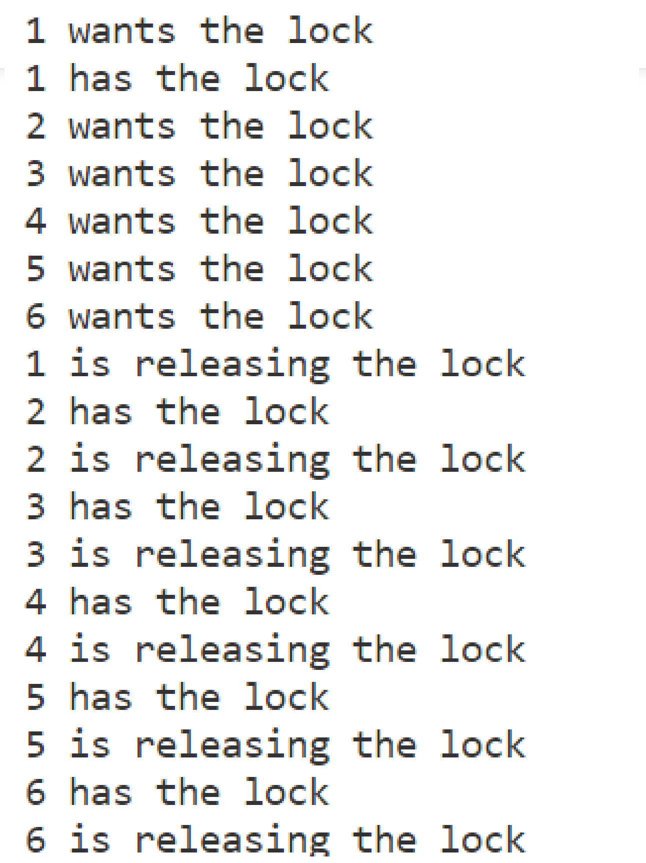
Locking with Buffered Channels

```
package main

import (
        "fmt"
        "time"
)

func main() {
        lock := make(chan bool, 1)
        for i := 1; i < 7; i++ {
                go worker(lock, i)
        }
        time.Sleep(10 * time.Second)
}
```

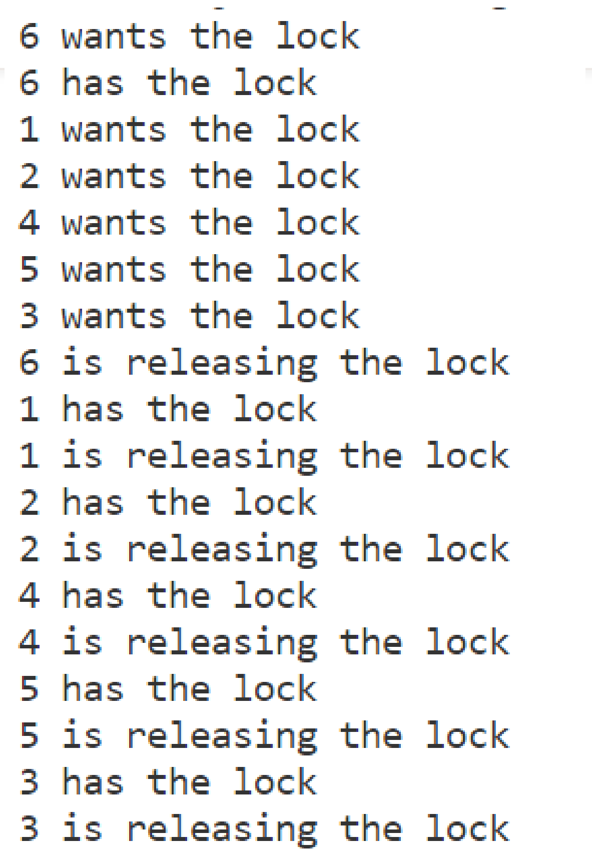
Locking with Buffered Channels

```
func worker(lock chan bool, id int) {

        fmt.Printf("%d wants the lock\n", id)

        lock <- true
        fmt.Printf("%d has the lock\n", id)

        time.Sleep(500 * time.Millisecond)

        fmt.Printf("%d is releasing the lock\n", id)
        <-lock
}
```

Locking with Buffered Channels



```
1 wants the lock
1 has the lock
2 wants the lock
3 wants the lock
4 wants the lock
5 wants the lock
6 wants the lock
1 is releasing the lock
2 has the lock
2 is releasing the lock
3 has the lock
3 is releasing the lock
4 has the lock
4 is releasing the lock
5 has the lock
5 is releasing the lock
6 has the lock
6 is releasing the lock
```



```
6 wants the lock
6 has the lock
1 wants the lock
2 wants the lock
4 wants the lock
5 wants the lock
3 wants the lock
6 is releasing the lock
1 has the lock
1 is releasing the lock
2 has the lock
2 is releasing the lock
4 has the lock
4 is releasing the lock
5 has the lock
5 is releasing the lock
3 has the lock
3 is releasing the lock
```

Two runs of the program

Closing Channels

- Closing channel is important.
- When channel is closed, goroutines can still perform receives on the channel, but can no longer send on the channel.
- This allows channel to be emptied of all its values from future receives, so nothing is lost.
- A receive on closed and empty channel always return immediately and provides the zero value for type of the channel declared with.
- Optional flag on channel receive can be requested to obtain information about state of the channel.
- A receiver should never close a receiving channel (instead, send a msg on the channel indicating that is done)
- The sender will then know upon receiving message, close the channel and return.

Deadlocks

- A deadlock happens when goroutines are waiting for each other and none of them is able to proceed.

```
func main() {
        ch := make(chan int)
        ch <- 1
        fmt.Println(<-ch)
}
```

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
    .../deadlock.go:7 +0x6c
```

- channel send operation waits forever for some other channel to read its value!

Deadlocks

- A goroutine can get stuck
  - because it is waiting for a channel or
  - because it is waiting to acquire lock in the sync package.

- Common causes are that
  - no other goroutine has access to the channel or the lock or
  - a group of goroutines are waiting for each other and none of them able to proceed.

- It is usually relatively easy to figure out what caused a deadlock for channels.
- Programs that make heavy use of mutexes can, on the other hand, can be difficult to debug.

Multiple Channels

- Channels are communication tools. They enable one goroutine to communicate information to another goroutine.
- Sometimes, the best way to solve concurrency problems is to communicate more information.
- This translates to using more channels.

- This is possible by making use of select statement.
- Select statement can watch multiple channels.
- Until something happens, it will block (or executed a default statement if there is any).
- When exactly a channel has event, the select statement will execute that event from that channel.

Multiple Channels

- If more than one event, select randomly picks one.
- If none of the channels have events, than select falls through to default.
- If no default is specified, select statement blocks until one the case can send or receive.

Multiple Channels

```
package main

import (
        "fmt"
        "os"
        "time"
)

func main() {
        done := time.After(30 * time.Second)
        echo := make(chan []byte)
        go readStdin(echo)
        for {
                select {
                case buf := <-echo:
                        os.Stdout.Write(buf)
                case <-done:
                        fmt.Println("Timed out")
                        os.Exit(0)
                }
        }
}
```

NGEE ANN
SCHOOL OF INFOCOMM TECHNOLOGY

Multiple Channels

```
func readStdin(out chan<- []byte) {
        for {
                data := make([]byte, 1024)
                l, _ := os.Stdin.Read(data)
                if l > 0 {
                        out <- data
                }
        }
}
```

- time.After() creates a channel that will receive message when 30 sec has elapsed.
- Select statement to pass data from Stdin to Stdout when received, or to shutdown when the time-out event occurs.

Channels

Do not overuse channels. Only use it when the need arises.
Channels are fantastic for communicating between goroutines. Simple to use and makes concurrent programming much easier.

But channels carry overhead and have performance impact. Also introduces complextity in program. Even more importantly, channels are the single biggest source memory management issues.

Sync package is part of Go's core, so it is well tested and maintained. So do make use of them.

Concurrency Patterns

- Reference: Google I/O 2012 - Go Concurrency Patterns by Rob Pike

Concurrency Patterns

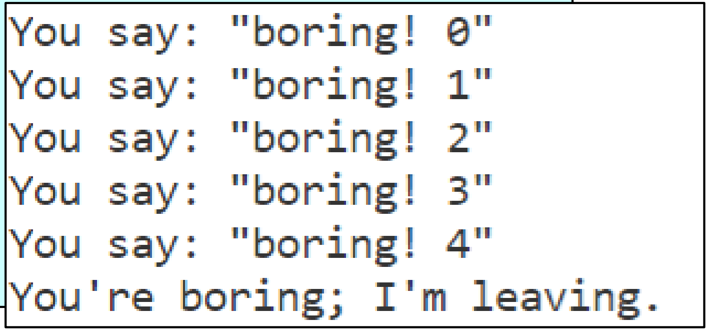
Generator Pattern

- Function that returns(thus generates) a channel
- Possible as channels are first-class values.

```go
package main

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

func main() {
	c := boring("boring!") // Function returning a channel.
	for i := 0; i < 5; i++ {
		fmt.Printf("You say: %q\n", <-c)
	}
	fmt.Println("You're boring; I'm leaving.")
}
```

Concurrency Patterns

```go
func boring(msg string) <-chan string { // Returns receive-only channel of
strings.
        c := make(chan string)
        go func() { // We launch the goroutine from inside the function.
                for i := 0; ; i++ {
                        c <- fmt.Sprintf("%s %d", msg, i)
                        time.Sleep(time.Duration(rand.Intn(1e3)) *
time.Millisecond)
                }
        }()
        return c // Return the channel to the caller.
}
```



```
You say: "boring! 0"
You say: "boring! 1"
You say: "boring! 2"
You say: "boring! 3"
You say: "boring! 4"
You're boring; I'm leaving.
```

Concurrency Patterns

Channels as a handle on a service

- function returns a channel that lets us communicate with the service it provides.
- Now can have more instances of the service.

```go
func main() {
        joe := boring("Joe")
        ann := boring("Ann")
        for i := 0; i < 5; i++ {
                fmt.Println(<-joe)
                fmt.Println(<-ann)
        }
        fmt.Println("You're both boring; I'm leaving.")
}
```

```
Joe 0
Ann 0
Joe 1
Ann 1
Joe 2
Ann 2
Joe 3
Ann 3
Joe 4
Ann 4
You're both boring; I'm leaving.
```

Concurrency Patterns

Multiplexing/Fan-In

- These programs make Joe and Ann count in lockstep.
- Now let’s use a fan-in function to allow whosoever is ready to talk.

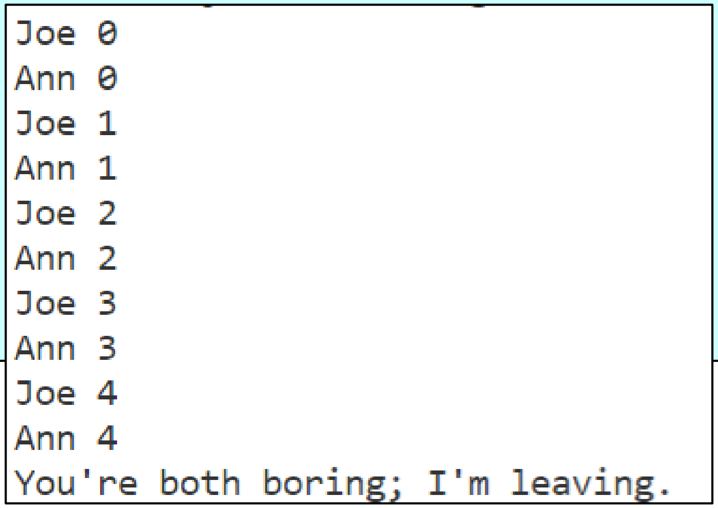
```go
package main

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

func main() {
	c := fanIn(boring("Joe"), boring("Ann"))
	for i := 0; i < 10; i++ {
		fmt.Println(<-c)
	}
	fmt.Println("You're both boring; I'm leaving.")
}
```

Concurrency Patterns

```
func fanIn(input1, input2 <-chan string) <-chan string {
        c := make(chan string)
        go func() {
                for {
                        c <- <-input1
                }
        }()
        go func() {
                for {
                        c <- <-input2
                }
        }()
        return c
}
```

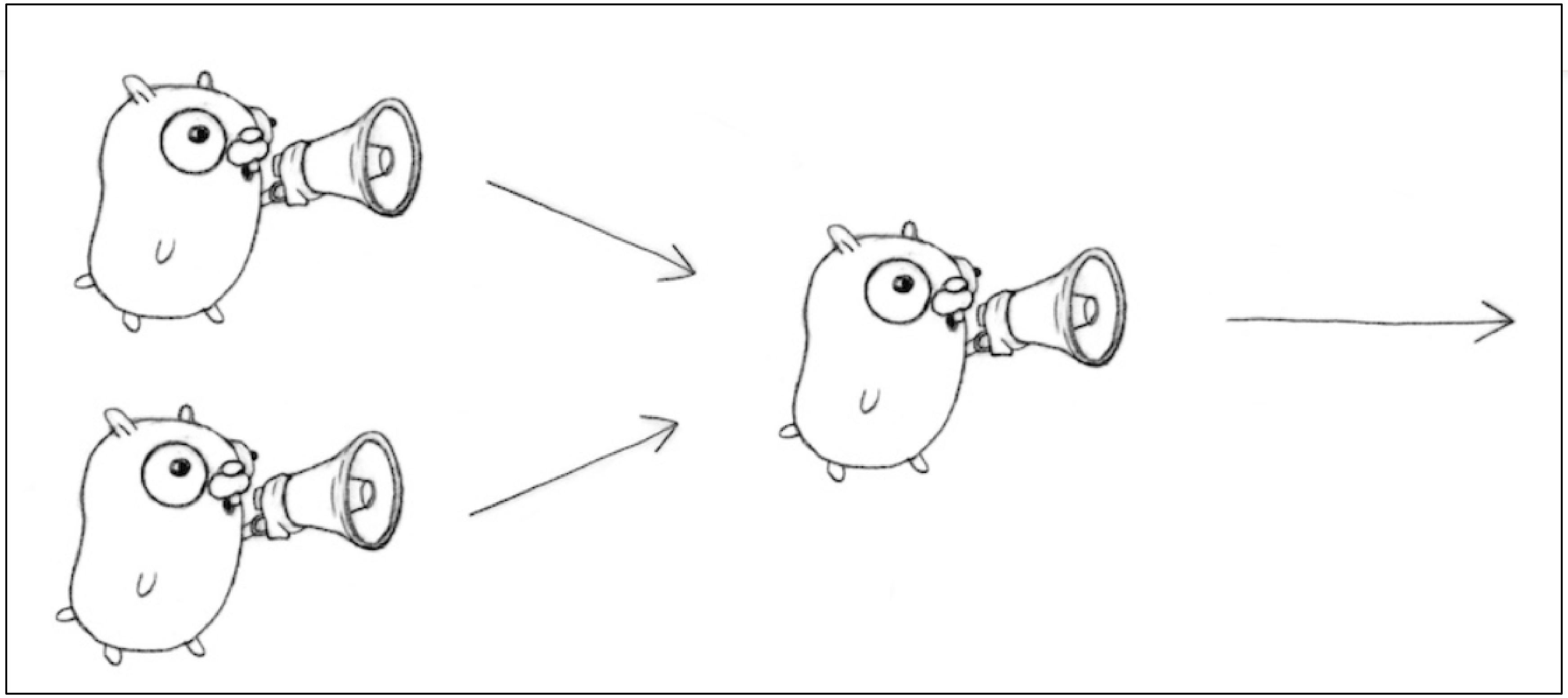
Concurrency Patterns

```
func boring(msg string) <-chan string { // Returns receive-only channel of strings.
        c := make(chan string)
        go func() { // We launch the goroutine from inside the function.
                for i := 0; ; i++ {
                        c <- fmt.Sprintf("%s %d", msg, i)
                        time.Sleep(time.Duration(rand.Intn(1e3)) * time.Millisecond)
                }
        }()
        return c // Return the channel to the caller.
}
```



```
Joe 0
Ann 0
Joe 1
Ann 1
Joe 2
Ann 2
Joe 3
Ann 3
Joe 4
Ann 4
You're both boring; I'm leaving.
```

Concurrency Patterns



Fan In

Google I/O 2012 - Go Concurrency Patterns by Rob Pike

Concurrency Patterns

Select

- A control structure unique to concurrency.
- The select statement provides another way to handle multiple channels.
- Similar to switch, but each case is a communication.
    - All channels are evaluated.
    - Selection blocks until one communication can proceed.
    - If multiple can proceed, select chooses pseudo-randomly.
    - A default clause, if present, executes immediately if no channel is ready.

Concurrency Patterns

```go
package main

func main() {
        var c1, c2, c3 chan int

        select {
        case v1 := <-c1:
                fmt.Printf("received %v from c1\n", v1)
        case v2 := <-c2:
                fmt.Printf("received %v from c2\n", v1)
        case c3 <- 23:
                fmt.Printf("sent %v to c3\n", 23)
        default:
                fmt.Printf("no one was ready to communicate\n")
        }
}
```

Concurrency Patterns

Fan In again using select

- Rewrite our original fanIn function. Only one goroutine is needed.

```
func fanIn(input1, input2 <-chan string) <-chan string {
        c := make(chan string)
        go func() {
                for {
                        select {
                        case s := <-input1:
                                c <- s
                        case s := <-input2:
                                c <- s
                        }
                }
        }()
        return c
}
```

```
Joe 0
Ann 0
Joe 1
Ann 1
Joe 2
Ann 2
Joe 3
Ann 3
Joe 4
Ann 4
You're both boring; I'm leaving.
```

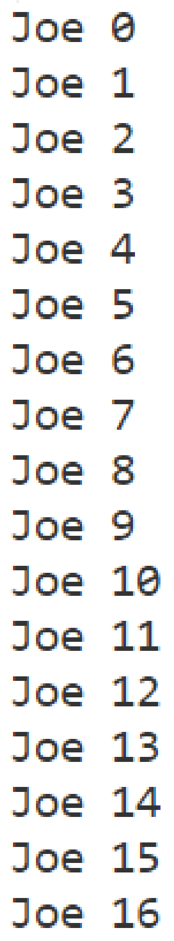
Concurrency Patterns

Time out using select

- time.After function returns a channel that blocks for the specified duration.
- After the interval, the channel delivers the current time, once.

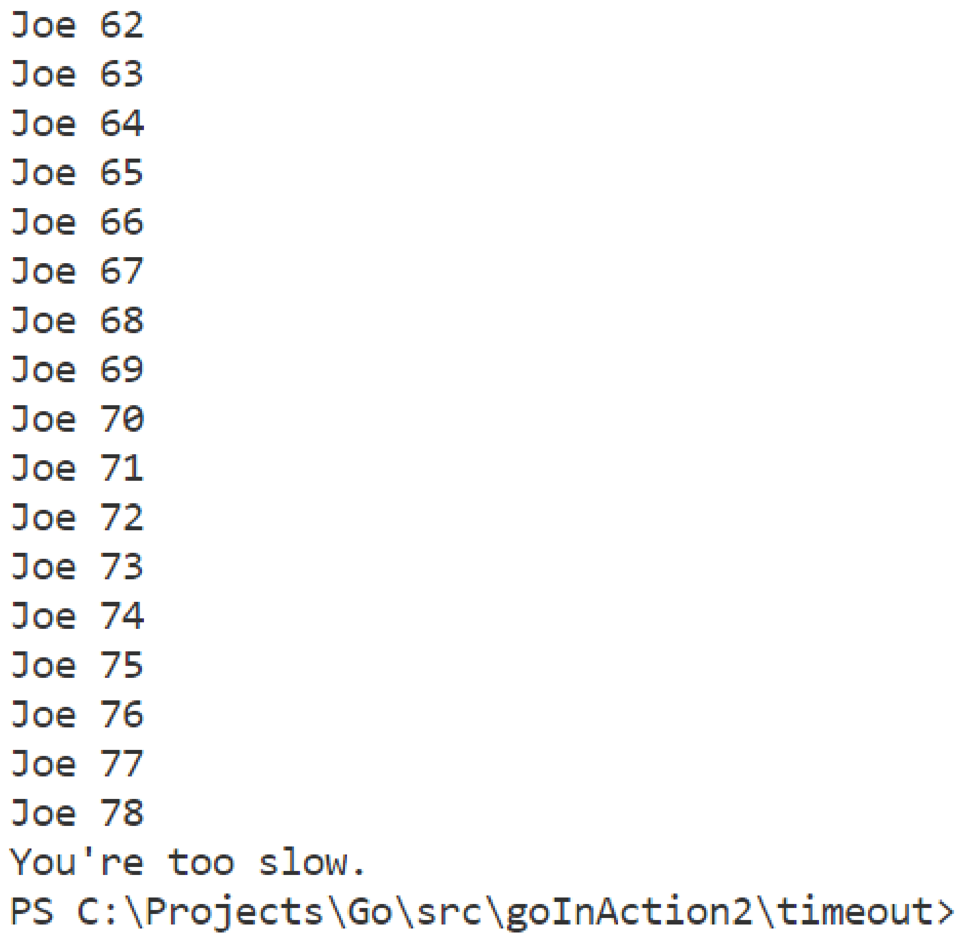
```
func main() {
        c := boring("Joe")
        for {
                select {
                case s := <-c:
                        fmt.Println(s)
                case <-time.After(1 * time.Second):
                        fmt.Println("You're too slow.")
                        return
                }
        }
}
```

Concurrency Patterns



```
Joe 0
Joe 1
Joe 2
Joe 3
Joe 4
Joe 5
Joe 6
Joe 7
Joe 8
Joe 9
Joe 10
Joe 11
Joe 12
Joe 13
Joe 14
Joe 15
Joe 16
```

...



```
Joe 62
Joe 63
Joe 64
Joe 65
Joe 66
Joe 67
Joe 68
Joe 69
Joe 70
Joe 71
Joe 72
Joe 73
Joe 74
Joe 75
Joe 76
Joe 77
Joe 78
You're too slow.
PS C:\Projects\Go\src\goInAction2\timeout>
```

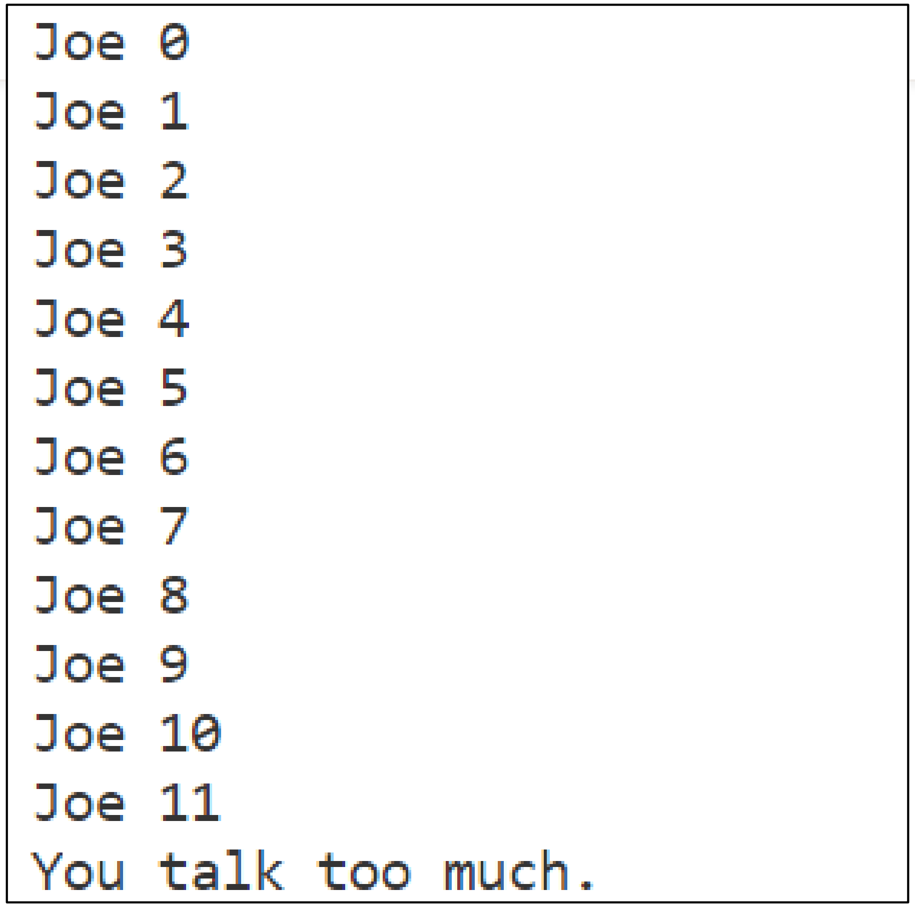
Concurrency Patterns

Time out for whole conversation using select

- Create the timer once, outside the loop, to time out the entire conversation.
- (Previously, it was timeout for each message.)

```
func main() {
        c := boring("Joe")
        timeout := time.After(5 * time.Second)
        for {
                select {
                case s := <-c:
                        fmt.Println(s)
                case <-timeout:
                        fmt.Println("You talk too much.")
                        return
                }
        }
}
```

Concurrency Patterns



```
Joe 0
Joe 1
Joe 2
Joe 3
Joe 4
Joe 5
Joe 6
Joe 7
Joe 8
Joe 9
Joe 10
Joe 11
You talk too much.
```

Concurrency Patterns

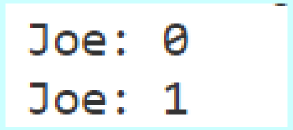
Quit Channel

- Now we can turn this around and get Joe to stop when tired of listening to him.

```
package main

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

func main() {
        quit := make(chan bool)
        c := boring("Joe", quit)
        for i := rand.Intn(10); i >= 0; i-- {
                fmt.Println(<-c)
        }
        quit <- true
}
```

Concurrency Patterns

```
func boring(msg string, quit <-chan bool) <-chan string {
        c := make(chan string)
        go func() {
                for i := 0; ; i++ {
                        time.Sleep(time.Duration(rand.Intn(1e3)) *
time.Millisecond)
                        select {
                        case c <- fmt.Sprintf("%s: %d", msg, i):
                                // do nothing
                        case <-quit:
                                return
                        }
                }
        }()
        return c
}
```



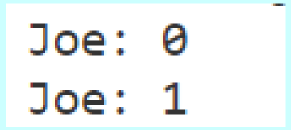
```
Joe: 0
Joe: 1
```

Concurrency Patterns

Receive on Quit Channel

- Any last messages now that we call it quits? Receive on the quit channel

```go
package main

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

func main() {
	quit := make(chan string)
	c := boring("Joe", quit)
	for i := rand.Intn(10); i >= 0; i-- {
		fmt.Println(<-c)
	}
	quit <- "Bye!"
	fmt.Printf("Joe says: %q\n", <-quit)
}
```



```
Joe: 0
Joe: 1
```

Concurrency Patterns

```
func boring(msg string, quit chan string) <-chan string {
    c := make(chan string)
    go func() {
        for i := 0; ; i++ {
            time.Sleep(time.Duration(rand.Intn(1e3)) *
time.Millisecond)
            select {
            case c <- fmt.Sprintf("%s: %d", msg, i):
                // do nothing
            case <-quit:

                quit <- "See you!"
                return
            }
        }
    }()
    return c
}
```

```
Joe: 0
Joe: 1
Joe says: "See you!"
```

Concurrency Patterns

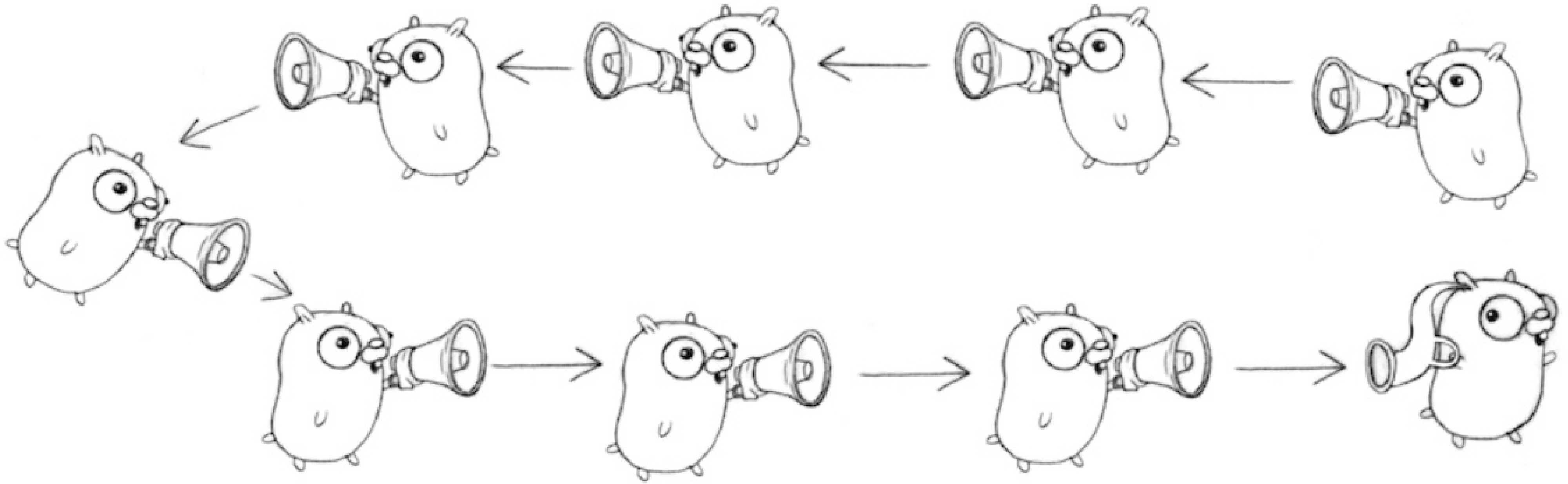
Daisy Chain

```
package main

import "fmt"

func f(left, right chan int) {
        left <- 1 + <-right
}

func main() {
        const n = 10000
        leftmost := make(chan int)
        right := leftmost
        left := leftmost
        for i := 0; i < n; i++ {
                right = make(chan int)
                go f(left, right)
                left = right
        }
        go func(c chan int) { c <- 1 }(right)
        fmt.Println(<-leftmost)
}
```

```
10001
```

NGEE ANN
SCHOOL OF INFOCOMM TECHNOLOGY

Concurrency Patterns



Daisy Chain

Google I/O 2012 - Go Concurrency Patterns by Rob Pike