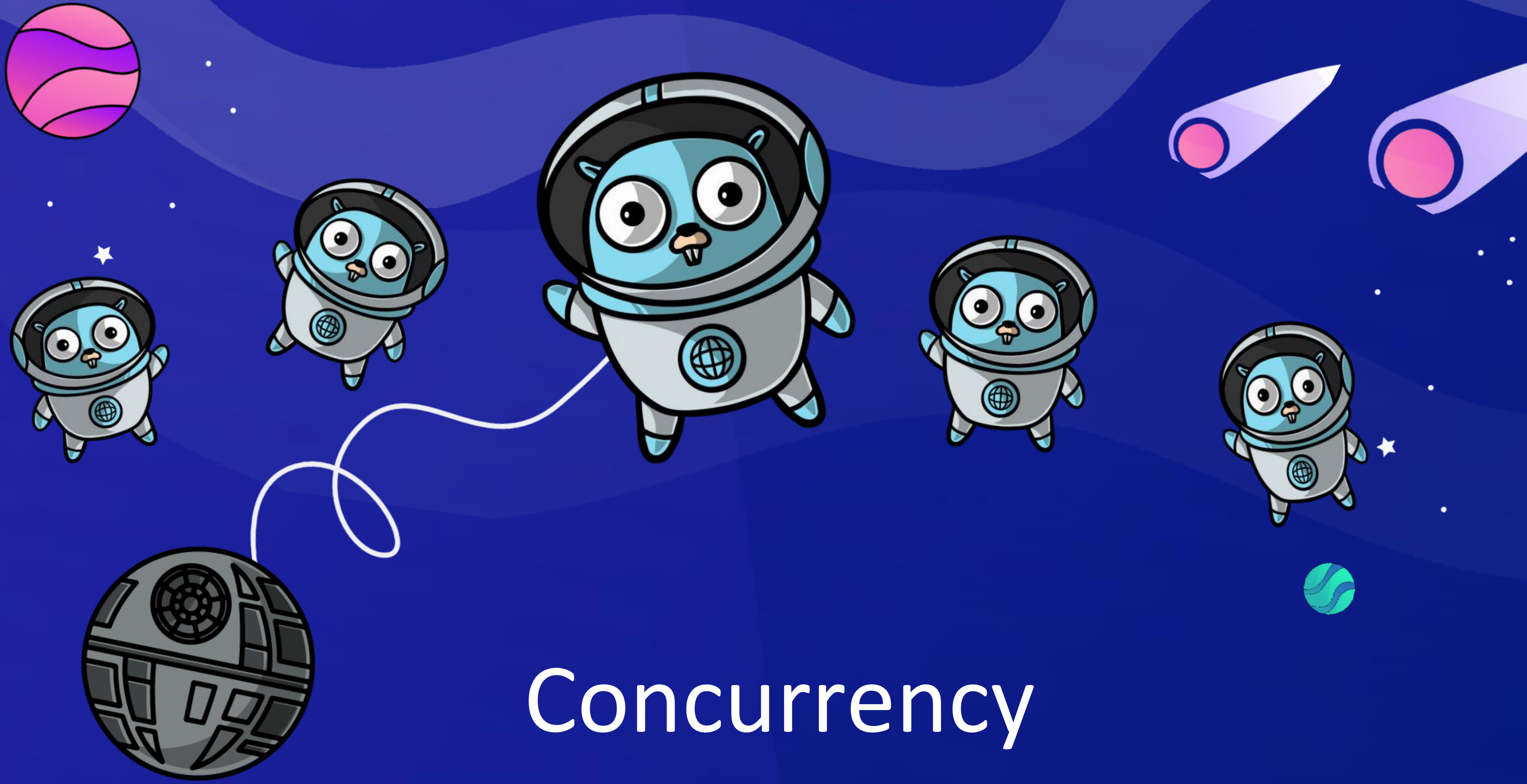
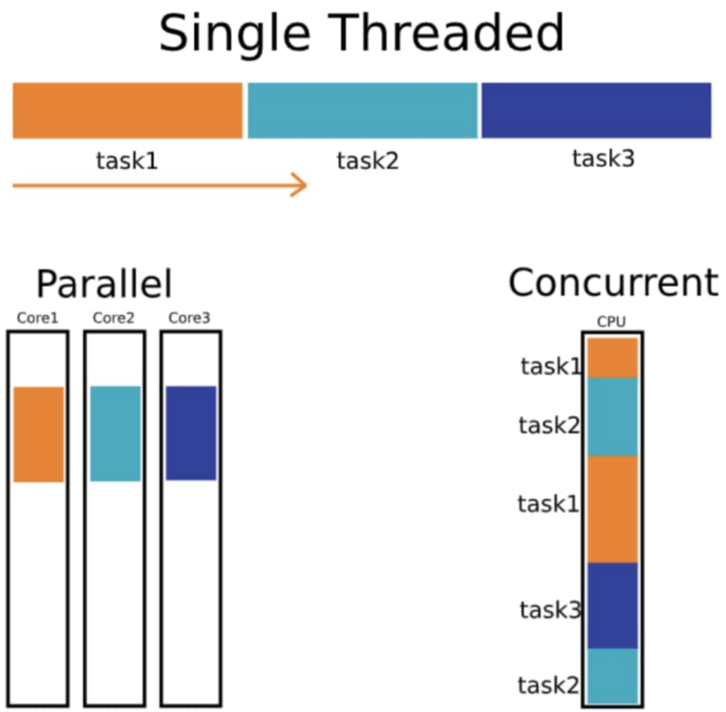


Concurrency

Agenda

- WHAT THE CONCURRENCY IS?
- DATA SYNCHRONIZATION
- DATA RACE

Concurrency vs parallelism

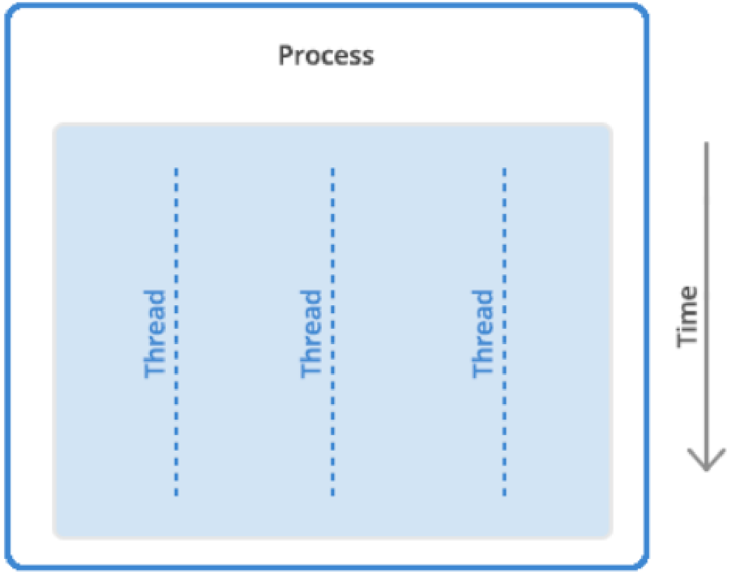


What is a PROCESS?

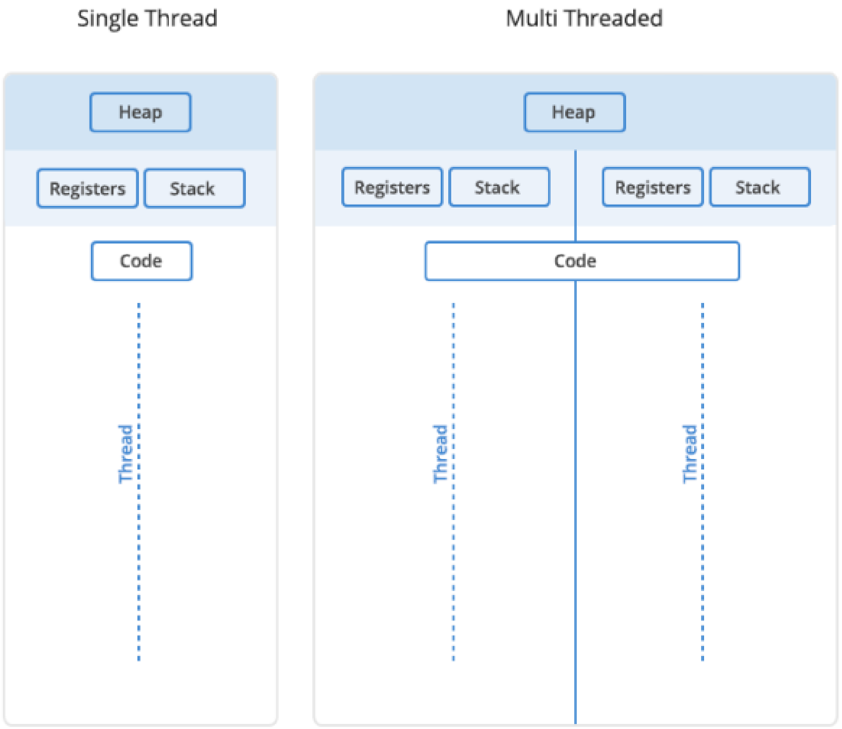
A process is the execution of a program that allows you to perform the appropriate actions specified in a program. It can be defined as an execution unit where a program runs. The OS helps you to create, schedule, and terminates the processes which is used by CPU. The other processes created by the main process are called child process.

What is a THREAD?

Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time.



Single/Multi Thread Process

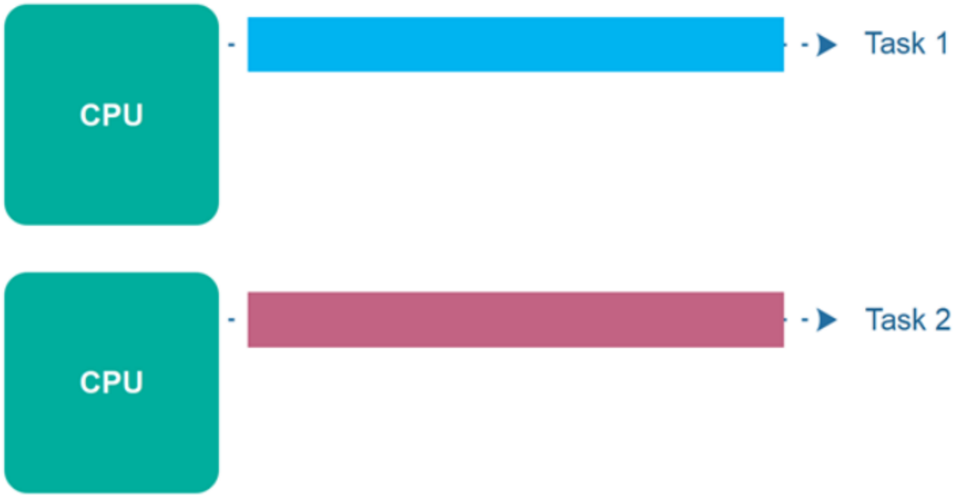


Process vs Thread

PROCESS	THREAD
Processes are heavyweight operations	Threads are lighter weight operations
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
Processes don't share memory with other processes	Threads share memory with other threads of the same process

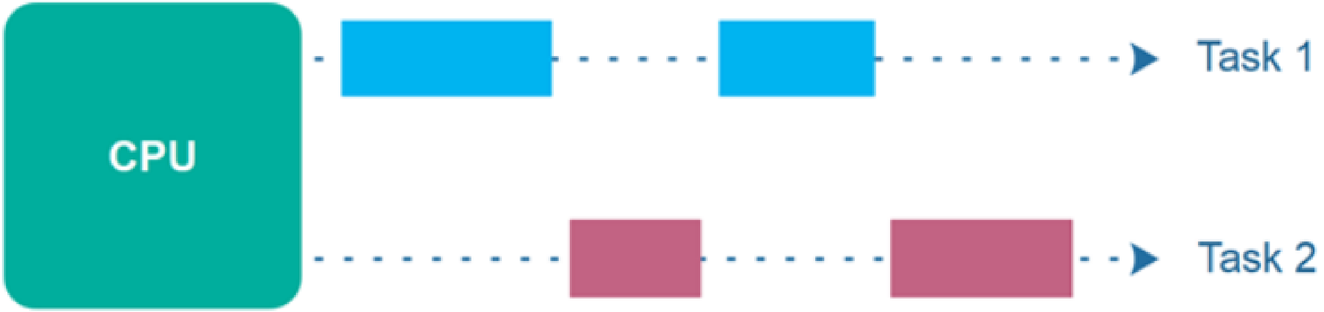
Parallel Execution

Parallel execution is when a computer has more than one CPU or CPU core, and makes progress on more than one task simultaneously. However, parallel execution is not referring to the same phenomenon as parallelism.



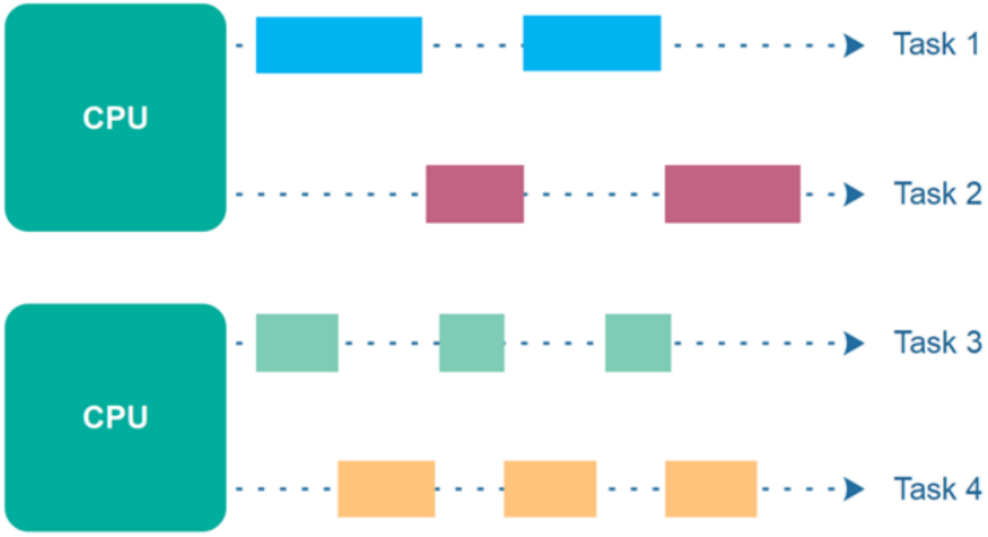
Parallel Execution

Concurrency means that an application is making progress on more than one task - at the same time or at least seemingly at the same time (concurrently). If the computer only has one CPU the application may not make progress on more than one task at exactly the same time, but more than one task is in progress at a time inside the application. To make progress on more than one task concurrently the CPU switches between the different tasks during execution.



Parallel Concurrent Execution

It is possible to have parallel concurrent execution, where threads are distributed among multiple CPUs. Thus, the threads executed on the same CPU are executed concurrently, whereas threads executed on different CPUs are executed in parallel. The diagram below illustrates parallel concurrent execution.



Goroutine

Goroutines

- What is a goroutine? It's an independently executing function, launched by a go statement.
- It has its own call stack, which grows and shrinks as required.
- It's very cheap. It's practical to have thousands, even hundreds of thousands of goroutines.
- It's not a thread.
- There might be only one thread in a program with thousands of goroutines.
- Instead, goroutines are multiplexed dynamically onto threads.

GOROUTINE

A goroutine is a lightweight thread managed by the Go runtime.

go f(x, y, z)

starts a new goroutine

f(x, y, z)

The evaluation of f, x, y, and z happens in the current goroutine and the execution of f happens in the new goroutine.

Goroutines run in the same address space, so access to shared memory must be synchronized.

GOROUTINE EXAMPLE

code

```go
package main

import (
   "fmt"
)

func main() {
   go fmt.Println("world")
   fmt.Println("hello")
}
```

output

hello

https://goplay.space/#EUdGJ0HsYcf

Channels

CHANNELS

Channels are a typed channel through which you can send and receive data with the channel operator, <-.

Like maps, channels must be created before use:

```
ch := make(chan int)

ch <- v        // Send v to channel ch.

v := <-ch      // Receive from ch, and
               // assign value to v.
```

(The data flows in the direction of the arrow.)

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without explicit locks or condition variables.

Channels (Initialization)

code

```go
func main() {
   ch := make(chan string, 1)

   go func() {
      ch <- "Hello World"
   }()

   fmt.Println(<-ch)
}
```

output

```
Hello World
```

Channel initialization:

make(chan type, n)

- type - type of the value
- n - length of the inner buffer

https://goplay.space/#CfKMQlhGFiZ

Channels (Initialization)

code

```go
func main() {
  var ch chan string

  go func() {
    ch <- "Hello World"
  }()

  fmt.Println(<-ch)
}
```

output

```
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [chan receive (nil chan)]: main.main()
/tmp/sandbox1008553871/prog.go:14 +0x4e goroutine 6
[chan send (nil chan)]: main.main.func1()
/tmp/sandbox1008553871/prog.go:11 +0x25 created by
main.main /tmp/sandbox1008553871/prog.go:10 +0x3c
```

Read and Write from channel that isn't initialized with, bring lock to the go routine.

https://goplay.space/#Gz4z9gcKZw7

CHANNELS EXAMPLE

code

```go
func main() {
   c := make(chan int)
   c <- 2
}
```

output

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

goroutine 1 [chan send]:
main.main()
        C:/ttt/main.go:5 +0x31
```

Channels (Read & Write)

code

```
func main() {
   ch := make(chan string)

   go func() {
      ch <- "Hello World"
   }()

   fmt.Println(<-ch)
}
```

output

```
Hello World
```

 # Communication between Go routines
Insert value into channel
ch <- “Hello World”

Read from channel
x := <- ch

https://goplay.space/#Dq31pd30U62

Range and Close

A sender can close a channel to indicate that no more values will be sent. Receivers can test whether a channel has been closed by assigning a second parameter to the receive expression:
After

v, ok := <-ch

ok is false if there are no more values to receive and the channel is closed.

The loop
for i := range c
receives values from the channel repeatedly until it is closed.

Note: Only the sender should close a channel, never the receiver. Sending on a closed channel will cause a panic.

Another note: Channels aren't like files; you don't usually need to close them. Closing is only necessary when the receiver must be told there are no more values coming, such as to terminate a range loop.

Channels (Finalization)

code

```go
func main() {
   ch := make(chan string)

   go func() {
      ch <- "Hello World"
      close(ch)
   }()

   for m := range ch {
      fmt.Println(m)
   }
}
```

output

```
Hello World
```

Channel finalization:

close(ch)

ch – name of the variable

https://goplay.space/#JsAiRyH6TID

Channels(Finalization)

code

```go
func main() {
   ch := make(chan string)

   close(ch)

   ch <- "Hello World"
}
```

output

```
panic: send on closed channel
goroutine 1 [running]: main.main()
/tmp/sandbox2454463867/prog.go:8
+0x45
```

Write to closed channel cause panic.

https://goplay.space/#CwM7VmJ2YLJ

Channels (Finalization)

code

```go
func main() {
  ch := make(chan string)

  go func() {
    ch <- "Hello World"
    close(ch)
  }()

  for i := 0; i < 3; i++ {
    v, ok := <-ch
    fmt.Printf("Value - %q, Ok - %t\n", v, ok)
  }
}
```

output

```
Value - "Hello World", Ok - true
Value - "", Ok - false
Value - "", Ok - false
```

Read from closed channel always return default value for the type and flag with value “false”.

https://goplay.space/#DmjdkGkmNZc

Range and Close

code

```go
func fibonacci(n int, c chan int) {
   x, y := 0, 1
   for i := 0; i < n; i++ {
      c <- x
      x, y = y, x+y
   }
   close(c)
}
func main() {
   c := make(chan int, 10)
   go fibonacci(cap(c), c)
   for i := range c {
      fmt.Println(i)
   }
}
```

output

```
0
1
1
2
3
5
8
13
21
34
```

 # range over channel
Loop that range over channel runs until channel closing.

https://goplay.space/#lNbU_AM_z_O

CHANNELS EXAMPLE

code

```go
func sum(s []int, c chan int) {
   sum := 0
   for _, v := range s {
      sum += v
   }
   c <- sum // send sum to c
}

func main() {
   s := []int{7, 2, 8, -9, 4, 0}
   c := make(chan int)
   go sum(s[:len(s)/2], c)
   go sum(s[len(s)/2:], c)
   x, y := <-c, <-c // receive from c
   fmt.Println(x, y, x+y)
}
```

output

-5 17 12

https://goplay.space/#aCVqbyM4aDq

BUFFERED CHANNELS

Channels can be buffered. Provide the buffer length as the second argument to make to initialize a buffered channel:

```
ch := make(chan int, 100)
```

Sends to a buffered channel block only when the buffer is full. Receives block when the buffer is empty.

code

```go
func main() {
   ch := make(chan int, 2)
   ch <- 1
   ch <- 2
   fmt.Println(<-ch)
   fmt.Println(<-ch)
}
```

output

```
1
2
```

BUFFERED CHANNELS

code

```
func main() {
   ch := make(chan string, 10)
   fmt.Println(len(ch), cap(ch))
   ch <- "Hello World"
   fmt.Println(len(ch), cap(ch))
   ch <- "Hello World"
   fmt.Println(len(ch), cap(ch))
}
```

output

```
0 10
1 10
2 10
```

Select

Select

code

```go
func fibonacci(c, quit chan int) {
  x, y := 0, 1
  for {
    select {
    case c <- x:
      x, y = y, x+y
    case <-quit:
      fmt.Println("quit")
      return
    }
  }
}

func main() {
  c := make(chan int)
  quit := make(chan int)
  go func() {
    for i := 0; i < 10; i++ {
      fmt.Println(<-c)
    }
    quit <- 0
  }()
  fibonacci(c, quit)
}
```

output

```
0
1
1
2
3
5
8
13
21
34
quit
```

https://goplay.space/#eJZ6RZrnwcA

Default section

code

```go
func main() {
   tick := time.Tick(100 * time.Millisecond)
   boom := time.After(500 * time.Millisecond)
   for {
      select {
      case <-tick:
         fmt.Println("tick.")
      case <-boom:
         fmt.Println("BOOM!")
         return
      default:
         fmt.Println(" .")
         time.Sleep(50 * time.Millisecond)
      }
   }
}
```

output

```
 .
 .
tick.
 .
 .
tick.
 .
tick.
 .
 .
tick.
 .
BOOM!
```

https://goplay.space/#7gu7TzopGIM

Sync

Mutex

A Mutex, or a mutual exclusion is a mechanism that allows us to prevent concurrent processes from entering a critical section of data whilst it’s already being executed by a given process.

Race condition

```go
go func() {
        i++
}()
go func() {
        i++
}()
```

```go
tmp = a+1
a = tmp
if a == 1 {
   criticalSection()
}
```

```go
tmp = a+1
a = tmp
if a == 1 {
   criticalSection()
}
```

```go
var mx sync.Mutex
// ..
mx.Lock()
tmp = a + 1
a = tmp
if a == 1 {
   criticalSection()
}
mx.Unlock()
```

Sync: WaitGroup

code

```go
func worker(id int, wg *sync.WaitGroup) {
  defer wg.Done()
  fmt.Printf("Worker %d starting\n", id)
  time.Sleep(time.Second)
  fmt.Printf("Worker %d done\n", id)
}
func main() {
  var wg sync.WaitGroup
  for i := 1; i <= 5; i++ {
    wg.Add(1)
    go worker(i, &wg)
  }
  wg.Wait()
}
```

output

```
Worker 2 starting
Worker 3 starting
Worker 4 starting
Worker 5 starting
Worker 1 starting
Worker 1 done
Worker 4 done
Worker 3 done
Worker 2 done
Worker 5 done
```

https://goplay.space/#_5jBUZmYcFz

Data Race

What is it?

Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.

code

```go
func race() {
  wait := make(chan struct{})
  n := 0
  go func() {
    n++ // read, increment, write
    close(wait)
  }()
  n++ // conflicting access
  <-wait
  fmt.Println(n) // Output: <unspecified>
}
```

How to find?

To help diagnose such bugs, Go includes a built-in data race detector. To use it, add the -race flag to the go command:

```
$ go test -race mypkg    // to test the package
$ go run -race mysrc.go  // to run the source file
$ go build -race mycmd   // to build the command
$ go install -race mypkg // to install the package
```

Typical Data Races: Race on loop counter

code

```
func main() {
  var wg sync.WaitGroup
  wg.Add(5)
  for i := 0; i < 5; i++ {
    go func() {
      fmt.Println(i)
      wg.Done()
    }()
  }
  wg.Wait()
}
```

output

```
3
5
5
5
5
```

fix

```
func main() {
  var wg sync.WaitGroup
  wg.Add(5)
  for i := 0; i < 5; i++ {
    go func(j int) {
      fmt.Println(j)
      wg.Done()
    }(i)
  }
  wg.Wait()
}
```

https://goplay.space/#_qA-EoVRpJw

Typical Data Races: Accidentally shared variable

code

```go
// ParallelWrite writes data to file1 and file2, returns the errors.
func ParallelWrite(data []byte) chan error {
  res := make(chan error, 2)
  f1, err := os.Create("file1")
  if err != nil {
    res <- err
  } else {
    go func() {
      // This err is shared with the main goroutine,
      // so the write races with the write below.
      _, err = f1.Write(data)
      res <- err
      f1.Close()
    }()
  }
  f2, err := os.Create("file2") // The second conflicting write to err.
  if err != nil {
    res <- err
  } else {
    go func() {
      _, err = f2.Write(data)
      res <- err
      f2.Close()
    }()
  }
```

fix

```go
...
_, err := f1.Write(data)
...
_, err := f2.Write(data)
...
```

Typical Data Races: Unprotected global variable

code

```go
var service map[string]net.Addr

func RegisterService(name string, addr net.Addr) {
   service[name] = addr
}

func LookupService(name string) net.Addr {
   return service[name]
}
```

fix

```go
var (
   service   map[string]net.Addr
   serviceMu sync.Mutex
)

func RegisterService(name string, addr net.Addr) {
   serviceMu.Lock()
   defer serviceMu.Unlock()
   service[name] = addr
}

func LookupService(name string) net.Addr {
   serviceMu.Lock()
   defer serviceMu.Unlock()
   return service[name]
}
```

Atomic counters

code

```go
func main() {
  var ops uint64
  var wg sync.WaitGroup
  for i := 0; i < 50; i++ {
    wg.Add(1)
    go func() {
      for c := 0; c < 1000; c++ {
        ops++
      }
      wg.Done()
    }()
  }
  wg.Wait()
  fmt.Println("ops:", ops)
}
```

output

ops: 43518

https://goplay.space/#Vy5J5SlOBMw

Atomic counters fix

code

```go
func main() {
   var ops uint64
   var wg sync.WaitGroup
   for i := 0; i < 50; i++ {
      wg.Add(1)
      go func() {
         for c := 0; c < 1000; c++ {
            atomic.AddUint64(&ops, 1)
         }
         wg.Done()
      }()
   }
   wg.Wait()
   fmt.Println("ops:", ops)
}
```

output

ops: 50000

https://goplay.space/#-v842yznjla

Mutex counter

code

```go
type SafeCounter struct {
  mu sync.Mutex
  v  map[string]int
}

func (c *SafeCounter) Inc(key string) {
  c.mu.Lock()
  c.v[key]++
  c.mu.Unlock()
}
func (c *SafeCounter) Value(key string) int {
  c.mu.Lock()
  defer c.mu.Unlock()
  return c.v[key]
}
func main() {
  c := SafeCounter{v: make(map[string]int)}
  for i := 0; i < 1000; i++ {
    go c.Inc("somekey")
  }
  time.Sleep(time.Second)
  fmt.Println(c.Value("somekey"))
}
```

output

1000

https://goplay.space/#UUh_4XFIJtT

Context

Cancel go routine with context

code

```go
func main() {
  work := make(chan func(), 10)
  ctx, cancel := context.WithCancel(context.Background())
  var wg sync.WaitGroup
  wg.Add(1)
  go func() {
    for {
      select {
      case <-ctx.Done():
        fmt.Println("Worker shutdown")
        wg.Done()
        return
      case w := <-work:
        w()
      }
    }
  }()
  for i := 0; i < 10; i++ {
    work <- func() {
      fmt.Println("Working...")
      time.Sleep(1 * time.Second)
    }
  }
  <-time.After(2 * time.Second)
  cancel()
  wg.Wait()
}
```

output

```
T+00.000000Working...
T+01.000000Working...
T+02.000000Working...
T+03.000000Working...
T+04.000000Worker shutdown
```

https://goplay.space/#TuU9qqWyhpX

Handle timeout & Deadlines

code

```go
const shortDuration = 2 * time.Millisecond

func main() {
   ctx, cancel := context.WithTimeout(context.Background(), shortDuration)
   defer cancel()

   // d := time.Now().Add(shortDuration)
   // ctx, cancel := context.WithDeadline(context.Background(), d)

   select {
   case <-time.After(1 * time.Second):
      fmt.Println("overslept")
   case <-ctx.Done():
      fmt.Println(ctx.Err()) // prints "context deadline exceeded"
   }
}
```

output

```
T+00.002000context deadline exceeded
```

https://goplay.space/#-5E6JKWdSyx

ErrGroup

Semaphore (Limit parallelism for task)

code

```go
func main() {
  const paralel = 5
  var wg sync.WaitGroup
  sem := make(chan struct{}, paralel)
  for i := 0; i < 100; i++ {
    wg.Add(1)
    sem <- struct{}{}
    go func() {
      work()
      <-sem
      wg.Done()
    }()
  }
  wg.Wait()
}
func work() {
  time.Sleep(1 * time.Second)
  fmt.Println("Work is done")
}
```

output

```
T+01.000000 Work is done
Work is done
Work is done
Work is done
Work is done
T+02.000000 Work is done
Work is done
Work is done
```

https://goplay.space/#Q8txT0BL5kJ

Semaphore (Limit parallelism for task) errgroup

code

```go
func main() {
  const parallel = 5
  errG, _ := errgroup.WithContext(context.Background())
  errG.SetLimit(parallel)
  for i := 0; i < 100; i++ {
    errG.Go(func() error {
      return work()
    })
  }
  err := errG.Wait()
}
func work() error {
  time.Sleep(1 * time.Second)
  fmt.Println("Work is done")
  return nil
}
```

output

```
T+01.000000 Work is done
Work is done
Work is done
Work is done
Work is done
T+02.000000 Work is done
Work is done
Work is done
```

https://goplay.space/#5Ci08VNbqM-

Questions

Additional materials

Links

- https://golang.org/doc/effective_go.html#concurrency
- https://www.youtube.com/watch?v=cN_DpYBzKso
- https://github.com/golang/go/wiki/LearnConcurrency
- https://habr.com/ru/company/avito/blog/466495/

Task

Homework

Concurrent batch

- Implement a function that will load users from the database

```
getBatch(n int64, pool int64) res []user
```

- The function takes two arguments - the number of users and the number of goroutines in which users will concurrently load

```
for i := 0; i < number; i++ {

      go func() {

         user := getOne...
```

- The function returns an array of received users
- It is planned to review the solution for the interview

Tips & tricks

- Don't forget the data race
- Due to autocode VM limitations, run tests locally. It is possible that autocode tests will pass with an invalid solution
- To limit concurrently running goroutines you can use:
  - Semaphore pattern
  - Worker pool
  - errgroup
  - etc

Thanks