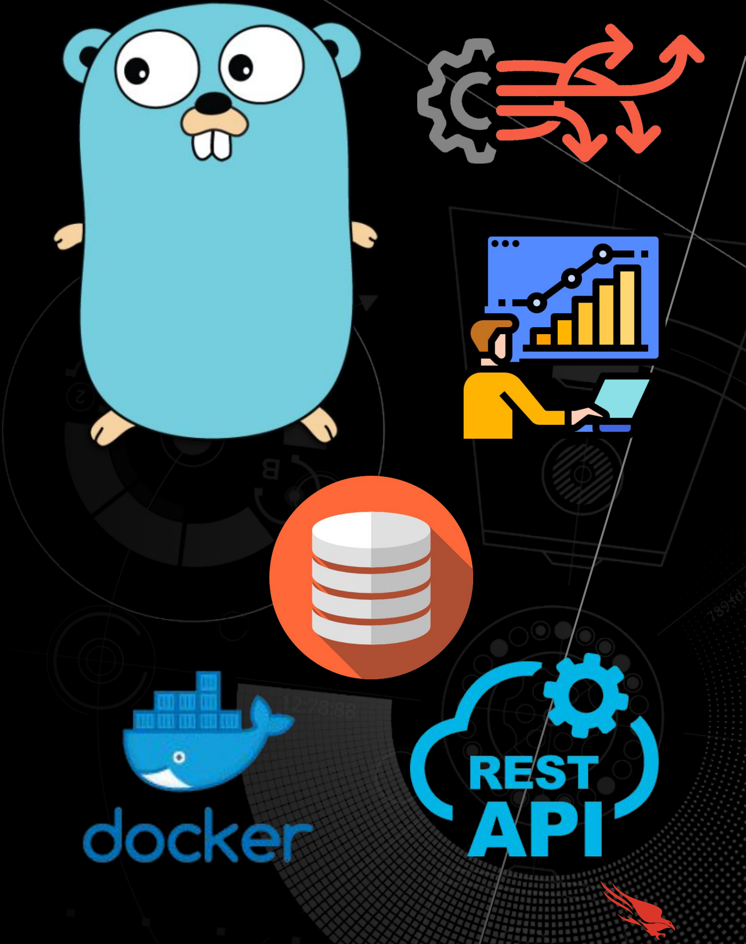
Multithreading

CrowdStrike HEROES - Cloud Workshop

CROWDSTRIKE

Date	Topic
July 25 - 16:00	Intro to golang
July 26 - 16:00	Intro to golang (continuation)
July 27 - 16:00	**Multithreading**
July 28 - 16:00	Rest API
July 29 - 16:00	Unit testing, logging and monitoring
August 1 - 16:00	Workshop and Q&A
August 2 - 16:00	Deployments/Docker
August 3 - 16:00	Databases
August 4 - 16:00	Databases extended
August 5 - 13:00	Microservices contest (4h with Awards)



Basics

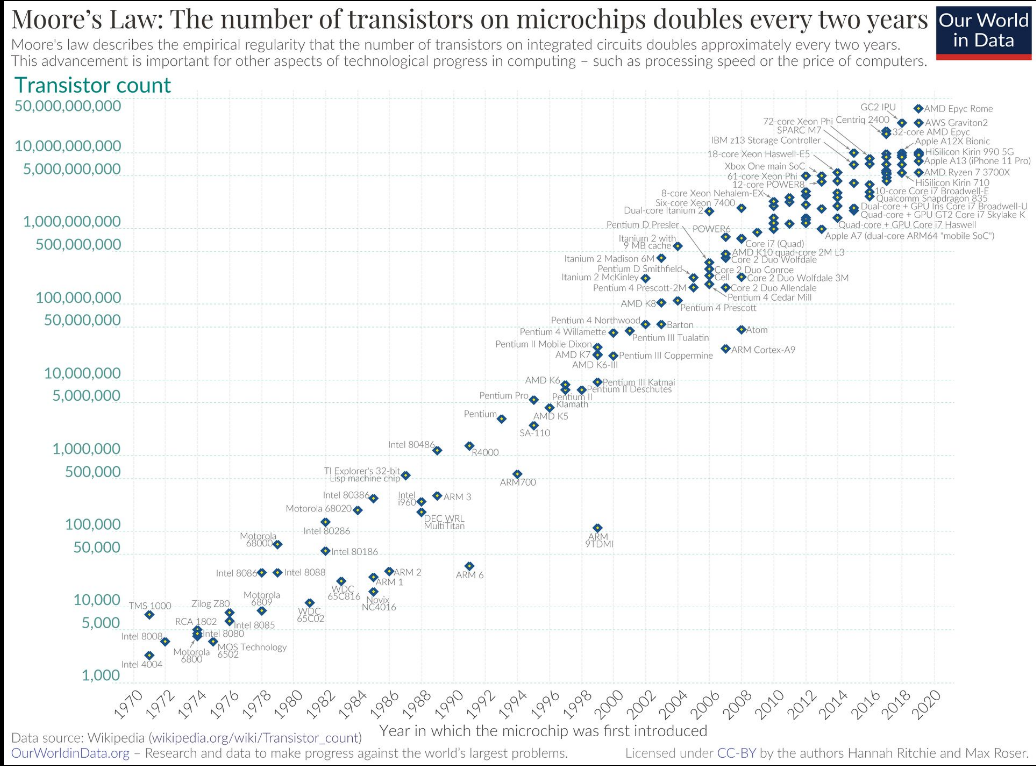
Why do we need multithreading?

What is a thread?

What is a race condition?

How to fix a race condition?

Why do we need multithreading?



- Moore's law will cease to apply at some point
- Solution: multiple processors
- Problem: working with multiple processors requires more knowledge and skill to do it correctly

Threads

A thread is a path of execution within a process.

- Threads within the same process run in a shared memory space

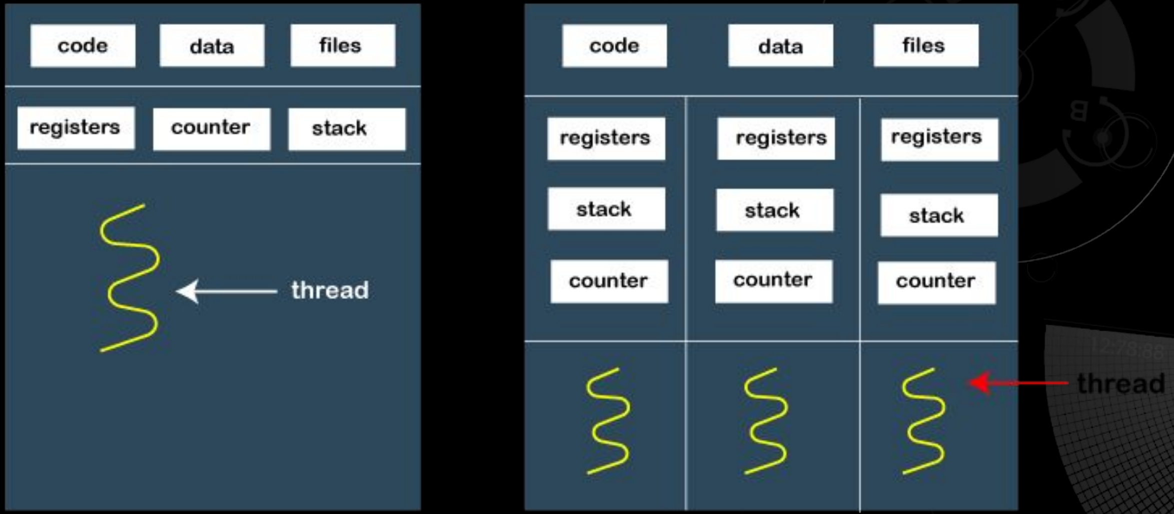


Image from https://static.javatpoint.com/difference/images/process-vs-thread3.png

More about how the OS handles threads [here]

Race conditions

Concurrent updates

Thread A	Thread B
`count = count + 1`	`count = count + 1`

What is the value of count at the end, assuming `count=0` ?

Example based on ch 1.5.2 in
https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Race conditions

Concurrent updates behind the scenes

Thread A	Thread B
`temp = count`	`temp = count`
`count = temp + 1`	`count = temp + 1`

The value of count can be either one of these:

```
count = 2
```

```
count = 1
```

Now imagine the same problem with 100 threads :D

In machine operations the increment is split in two operations.

Some computers do actually provide an increment instruction that cannot be interrupted.

An operation that cannot be interrupted is called **atomic**.

Example based on ch 1.5.2 in https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

Solution: mutex

Mutex = mutual exclusion

Usually the operations we need to perform on a mutex are:

- mutex.Lock()
- mutex.Unlock()

These operations are usually surrounding a critical section.

Be aware that any lock waits for the other threads to unlock the mutex.

Misuse of mutexes can lead to problems: see deadlocks.

Go syntax for multithreading

Goroutines

Channels

Sync package: mutex, waitGroup

Goroutines

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

  ```
  go f(x, y, z)
  ```

  starts a new goroutine running

  ```
  f(x, y, z)
  ```

- f, x, y, and z are **evaluated** in the current goroutine
- f is **executed** in the new goroutine
- Goroutines run in the same address space => access to shared memory must be synchronized

```
func say(s string) {
	for i := 0; i < 5; i++ {
		time.Sleep(100 * time.Millisecond)
		fmt.Println(s)
	}
}

func main() {
	go say("world")
	say("hello")
}
```

Function closures

- We can define anonymous functions
  - Much like lambdas in other languages

```
f := func (args) retType {
    // do stuff
}
ret := f(arg1, arg2)
```

- We can call it immediately by adding the arguments after the definition

```
res := func() int {return 0} ()
```

- As you can see we can have functions with no args or no return value

Variables in closures

Capturing outside variables

```
v := 3
f := func () int {
    return v
}
v = 4
fmt.Println(f())
> 4
```

Using values from surrounding context uses the value at the time of the call. All further changes to the objects can be seen in the function

Capturing variables by arguments

```
v := 3
res := func (v int) int {
    return v
}(v)
v = 4
fmt.Println(res)
> 3
```

Sending values as arguments takes the value at the time of the definition

Channels

- Channels help us send and receive values
- The channel operator is <-

```
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.)

- Channels must be created before use:

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

- Sends and receives block until the other side is ready

```
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)
}
```

Buffered Channels

- 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.

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

Range & 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:

```
v, ok := <-ch
```

- If `ok` is `false` there are no more values to receive and the channel is closed.
- To receive values from the channel repeatedly until it is closed:

```
for i := range c
```

```
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)
	}
}
```

Select

- The `select` statement lets a goroutine wait on multiple communication operations.
- A select blocks until one of its cases can run, then it executes that case.
- It chooses one at random if multiple are ready.

```
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
		}
	}
}
```

Default Selection

- The default case in a select is run if no other case is ready
- Use a default case to try a send or receive without blocking:

```
select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}
```

sync.Mutex

- Go's standard library provides mutual exclusion with sync.Mutex and its two methods:
  - Lock
  - Unlock
- We can define a block of code to be executed in mutual exclusion by **surrounding** it with a call to Lock and Unlock

```
func (c *SafeCounter) Inc(key string) {
	c.mu.Lock()
	// Lock so only one goroutine at a time can access the map c.v.
	c.v[key]++
	c.mu.Unlock()
}
```

Thread pools - Wait Groups

Since it's not very productive to start too many threads we can have only a few worker goroutines and send them the tasks.

To wait for multiple goroutines to finish, we can use a *wait group*.

- Wait groups are already initialized
- Add a goroutine: `wg.Add(1)`
- Mark the end of a goroutine: `wg.Done()`
- Wait for all goroutines to finish: Add a goroutine: `wg.Add(1)`

```
var wg sync.WaitGroup
for i := 1; i <= 5; i++ {
    wg.Add(1)
    i := i
    go func() {
        defer wg.Done()
        worker(i)
    }()
}
wg.Wait()
}
```

Any questions?