CES-27 Processamento Distribuído

Golang

Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

What is Go?

- Go (or Golang) is an open source programming language that makes it easy to build simple, reliable, and efficient software.
  - https://golang.org/
- History
  - Started in 2007 by Google
  - Open and stable version in 2012
  - Go is based on CSP
  - CSP
    - Communicating Sequential Processes
    - Created by Hoare in 1978
    - A formal language for describing patterns of interaction in concurrent systems
    - Occam and Erlang are two well known languages that stem from CSP
    - One of the most successful models for providing high-level linguistic support for concurrency
    - Paradigm for expressing concurrency based on message-passing
      - Models of concurrency: message passing (with processes, messages no shared data) instead of shared memory (with threads, locks, mutexes)

Tony Hoare (1934-?)
Turing Award in 1980

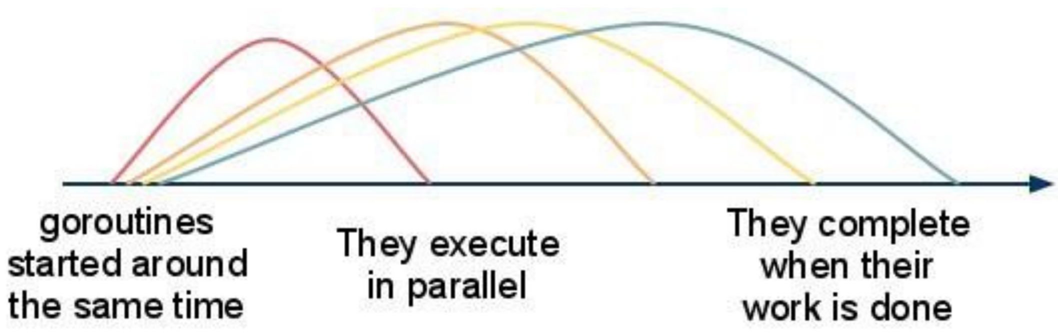
Motivation to create Go

- Started as an answer to software problems at Google:
  - multicore processors
  - networked systems
  - massive computation clusters
  - scale:
    - 10^7 lines of code
    - 10^3 programmers
    - 10^6 machines

- Goals at Google:
  - To eliminate the slowness of software development
  - To make the process more productive and scalable
  - Go was designed by and for people who develop large software systems

Go deals with concurrency!

- Go provides two important concepts:
  - **goroutine**: a thread of control within the program, with its own local variables and stack. Cheap, easy to create.
    - A goroutine consumes almost 2KB memory from the heap.
      - Note: Thread in Java consumes 1MB.
    - So, you can spin millions of goroutines at any time.
  - **channel**: carries typed messages between goroutines.



Example: Hello World

- The Go Playground
- https://play.golang.org/

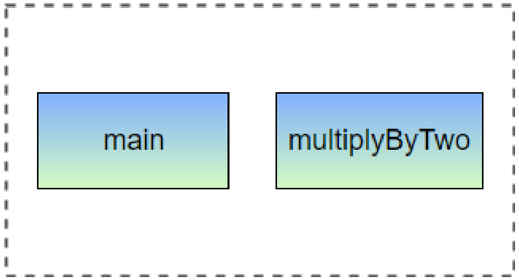
```go
package main

import (
	"fmt"
)

func main() {
	fmt.Println("Hello, playground")
}
```

Example: Visualizing Goroutines

- https://play.golang.org/p/MPV8CdrFWGi

1. What is the result?
2. Try to set zero for sleeping time
3. Try to not set 'multiplyByTwo' as goroutine

```go
package main

import (
        "fmt"
        "time"
)

func main() {
        n := 3

        // We want to run a goroutine to multiply n by 2
        go multiplyByTwo(n)

        // We pause the program so that the `multiplyByTwo` goroutine
        // can finish and print the output before the code exits
        time.Sleep(time.Second)
}

func multiplyByTwo(num int) int {
        result := num * 2
        fmt.Println(result)
        return result
}
```

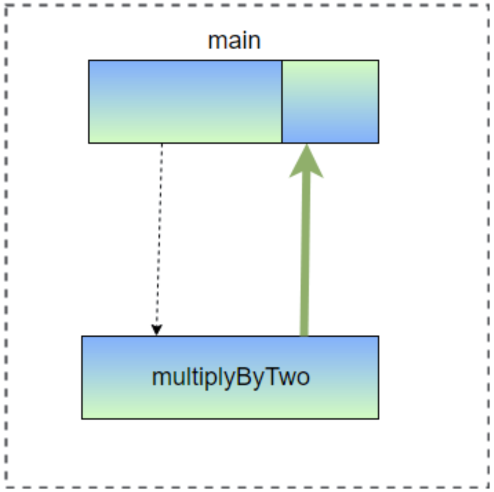


Example: Adding a channel

- https://play.golang.org/p/mIRGjGxYM3

1. What is the result? See the blocking code!
2. Try to repeat the ‘Println’ command below the current one
3. What can we do to print the result twice?

```go
package main

import (
	"fmt"
)

func main() {
	n := 3

	// This is where we "make" the channel, which can be used
	// to move the `int` datatype
	out := make(chan int)

	// We still run this function as a goroutine, but this time,
	// the channel that we made is also provided
	go multiplyByTwo(n, out)

	// Once any output is received on this channel, print it to the console and proceed
	fmt.Println(<-out)
}

// This function now accepts a channel as its second argument...
func multiplyByTwo(num int, out chan<- int) {
	result := num * 2

	//... and pipes the result into it
	out <- result
}
```

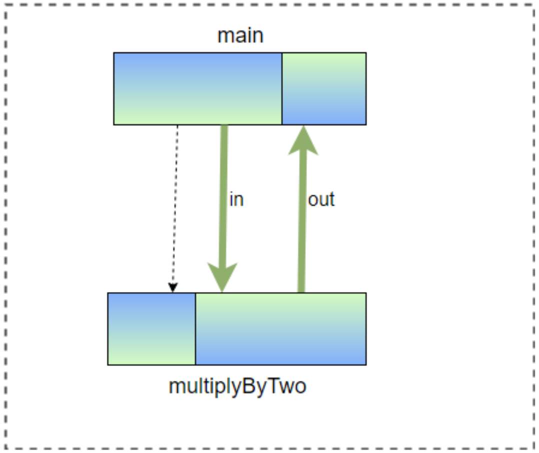


Example: Two single directional channels

- https://play.golang.org/p/aQIBDS99_d

1. What is the result?
2. See the difference in declaration of channels in 'multiplyByTwo' function.
   - Try to eliminate channels' direction here

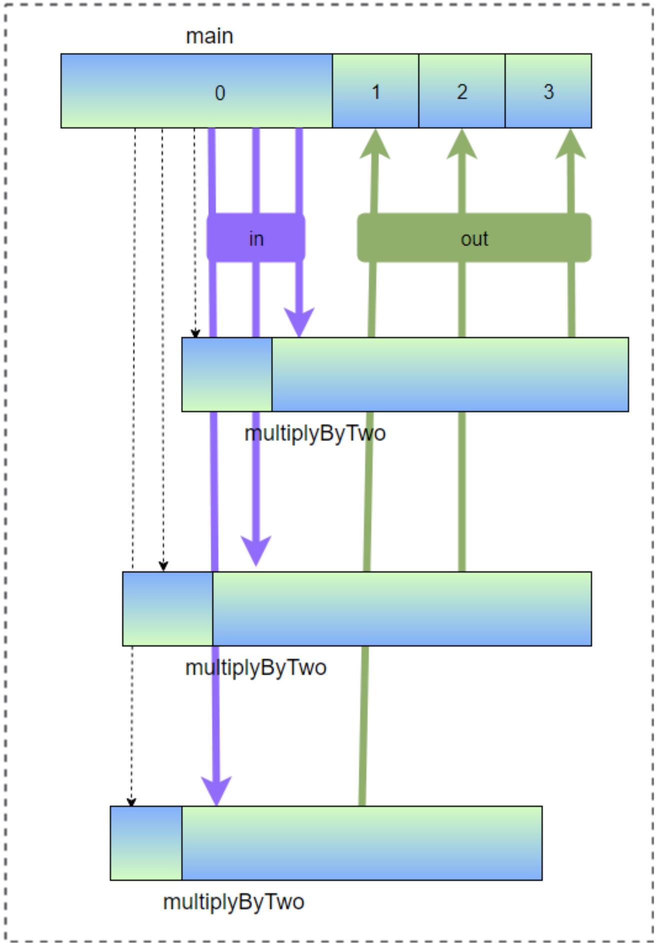
```go
package main

import (
	"fmt"
)

func main() {
	n := 3
	in := make(chan int)
	out := make(chan int)

	// We now supply 2 channels to the `multiplyByTwo` function
	// One for sending data and one for receiving
	go multiplyByTwo(in, out)

	// We then send it data through the channel and wait for the result
	in <- n
	fmt.Println(<-out)
}

func multiplyByTwo(in <-chan int, out chan<- int) {
	// This line is just to illustrate that there is code that is
	// executed before we have to wait on the `in` channel
	fmt.Println("Initializing goroutine...")

	// The goroutine does not proceed until data is received on the `in` channel
	num := <-in

	// The rest is unchanged
	result := num * 2
	out <- result
}
```



Example: Multiple concurrent goroutines

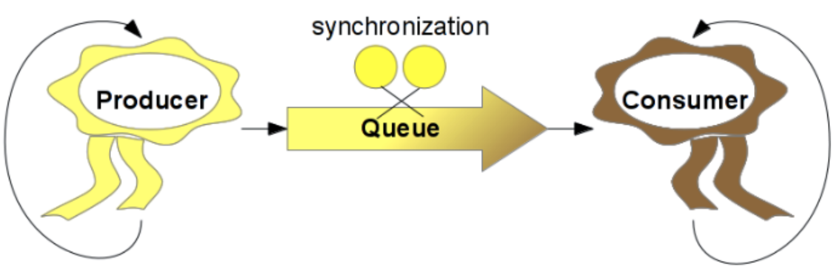
- https://play.golang.org/p/8ocrB53QS_
- There is no guarantee as to which goroutine will accept which input, or which goroutine will return an output first.
- Try to add an ID to goroutines to check the order of response
- Try to make the last goroutine sleep a bit

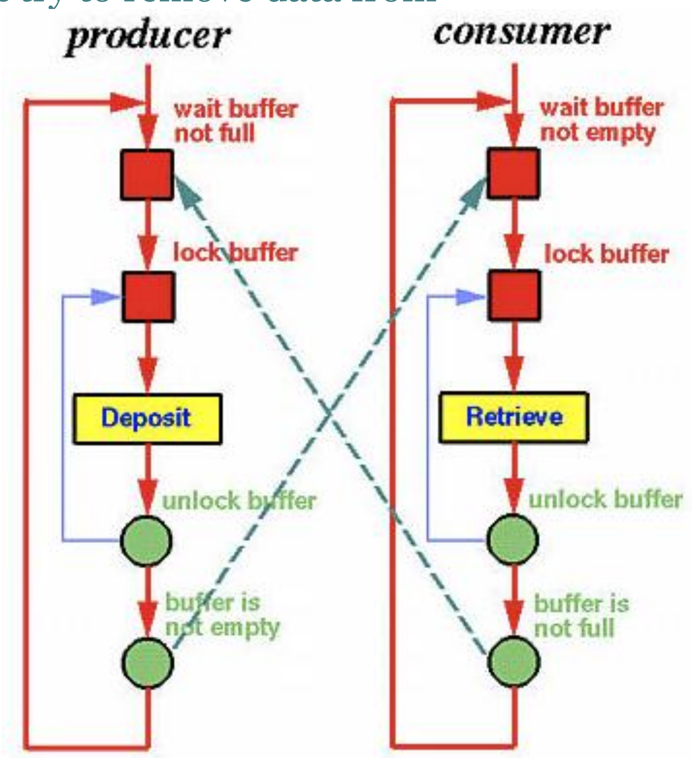
```go
package main

import (
	"fmt"
)

func main() {
	out := make(chan int)
	in := make(chan int)

	// Create 3 `multiplyByTwo` goroutines.
	go multiplyByTwo(in, out)
	go multiplyByTwo(in, out)
	go multiplyByTwo(in, out)

	// Up till this point, none of the created goroutines actually do
	// anything, since they are all waiting for the `in` channel to
	// receive some data
	in <- 1
	in <- 2
	in <- 3

	// Now we wait for each result to come in
	fmt.Println(<-out)
	fmt.Println(<-out)
	fmt.Println(<-out)
}

func multiplyByTwo(in <-chan int, out chan<- int) {
	fmt.Println("Initializing goroutine...")
	num := <-in
	result := num * 2
	out <- result
}
```



Example: Producer-Consumer

- The problem
  - Two processes (the producer and the consumer) who share a common, fixed-size buffer used as a queue.
  - The producer's job is to generate data, put it into the buffer, and start again.
  - At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.
  - The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.





Example: Producer-Consumer

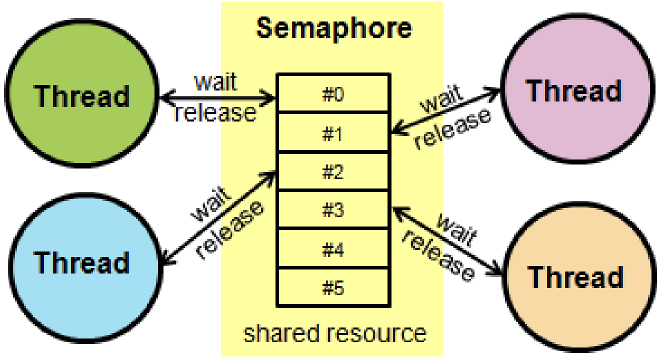
- A trivial solution
- This solution contains a race condition that can lead to a deadlock. E.g:
  - The consumer has just read the variable itemCount, noticed it's zero and is just about to move inside the if block.
  - Just before calling sleep, the consumer is interrupted and the producer is resumed.
  - The producer creates an item, puts it into the buffer, and increases itemCount.
  - Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
  - Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.
  - The producer will loop until the buffer is full, after which it will also go to sleep.

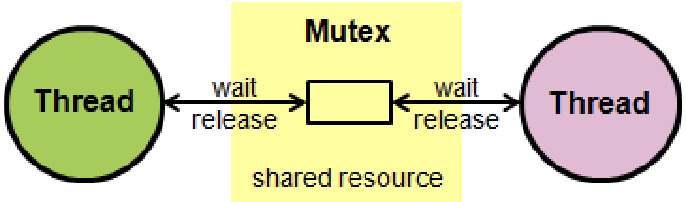
Solutions with shared memory: semaphore, mutex, and monitor

```
int itemCount = 0;

procedure producer()
{
    while (true)
    {
        item = produceItem();

        if (itemCount == BUFFER_SIZE)
        {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            wakeup(consumer);
        }
    }
}

procedure consumer()
{
    while (true)
    {

        if (itemCount == 0)
        {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

Example: Producer-Consumer

- A solution using semaphores and mutex



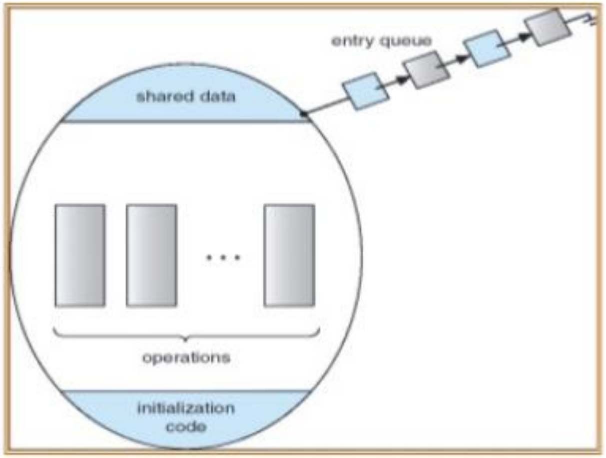


Note: mutex seems to work as a semaphore with value of 1 (binary semaphore), but there is difference in the fact that mutex has ownership concept. Ownership means that mutex can only be "incremented" back (set to 1) by the same process that "decremented" it (set to 0), and all other tasks wait until mutex is available for decrement (effectively meaning that resource is available), which ensures mutual exclusivity and avoids deadlock.

```
mutex buffer_mutex;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer()
{
    while (true)
    {
        item = produceItem();
        down(emptyCount);
        down(buffer_mutex);
        putItemIntoBuffer(item);
        up(buffer_mutex);
        up(fillCount);
    }
}

procedure consumer()
{
    while (true)
    {
        down(fillCount);
        down(buffer_mutex);
        item = removeItemFromBuffer();
        up(buffer_mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```

Example: Producer-Consumer

- A solution using monitors




```
procedure producer()
{
    while (true)
    {
        item = produceItem();
        ProducerConsumer.add(item);
    }
}

procedure consumer()
{
    while (true)
    {
        item = ProducerConsumer.remove();
        consumeItem(item);
    }
}
```

```
monitor ProducerConsumer
{
    int itemCount = 0;
    condition full;
    condition empty;

    procedure add(item)
    {
        if (itemCount == BUFFER_SIZE)
        {
            wait(full);
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            notify(empty);
        }
    }

    procedure remove()
    {
        if (itemCount == 0)
        {
            wait(empty);
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            notify(full);
        }


        return item;
    }
}
```

Note: A monitor is a shared object with operations (e.g. set data, get data), internal state, and a number of condition queues. Only one operation of a given monitor may be active at a given point in time. A process that calls a busy monitor is delayed until the monitor is free.

Example: Producer-Consumer in Go

- https://play.golang.org/p/a_HBATTBu54
- With explanation: https://gist.github.com/drio/dd2c4ad72452e3c35e7e

```
package main
/* producer-consumer problem in Go */


import ("fmt")

var done = make(chan bool)
var msgs = make(chan int)

func produce () {
    for i := 0; i < 10; i++ {
        msgs <- i
    }
    done <- true
}

func consume () {
    for {
      msg := <-msgs
      fmt.Println(msg)
    }
}

func main () {
   go produce()
   go consume()
   <- done
}
```

```
0
1
2
3
4
5
6
7
8
9

Program exited.
```

- Other way to implement consume

```
func consume () {
    for msg := range msgs {
      fmt.Println(msg)
    }
}
```

Example: Producer-Consumer in Go

- Original: https://play.golang.org/p/a_HBATTBu54
- Test with 2 producers and 1 consumer
  - Is it ok?

```go
package main
/* producer-consumer problem in Go */


import ("fmt")

var done = make(chan bool)
var msgs = make(chan int)

func produce () {
    for i := 0; i < 10; i++ {
        msgs <- i
    }
    done <- true
}

func consume () {
    for {
      msg := <-msgs
      fmt.Println(msg)
    }
}

func main () {
   go produce()
   go produce()
   go consume()
   <- done
}
```

```
0
1
2
3
4
5
6
7
8
9

Program exited.
```

  - What do we need to consume all messages?
    - https://play.golang.org/p/C7rHvuSHf1Z

Example: Producer-Consumer in Go

- Original: https://play.golang.org/p/a_HBATTBu54
- Test with 1 producer and 2 consumers
  - Add an *id* to consumers to identify them
  - Is it ok?

```go
package main
/* producer-consumer problem in Go */


import ("fmt")

var done = make(chan bool)
var msgs = make(chan int)

func produce () {
    for i := 0; i < 10; i++ {
        msgs <- i
    }
    done <- true
}

func consume (id int) {
    for {
      msg := <-msgs
      fmt.Println("Consumer", id, ":", msg)
   }
}

func main () {
   go produce()
   go consume(1)
   go consume(2)
   <- done
}
```

```
Consumer 2 : 0
Consumer 2 : 1
Consumer 2 : 2
Consumer 2 : 3
Consumer 2 : 4
Consumer 2 : 5
Consumer 2 : 6
Consumer 2 : 7
Consumer 2 : 8
Consumer 2 : 9

Program exited.
```

  - What do we need to see both consumers operating?
    - https://play.golang.org/p/N7Nu9clWKUl

Tips for non-blocking operations in Go

- Channel has size 1
  - It synchronizes the production with the consumption

- Using buffered channels
  - Consider a channel of 'Item'. E.g. int

Buffered channel of size 1 is different from former channel (with synchronization)

```go
queue := make(chan Item, 10) // queue with a capacity of 10
```

- For producer

```go
// next line will block queue is full
queue <- item
```

```go
// for a non-blocking push, do this:
var ok bool
select {
    case queue <- item:
        ok = true
    default:
        ok = false
}
// at this point, "ok" is:
//   true  => enqueued without blocking
//   false => not enqueued, would have blocked because of queue full
```

For consumer

```go
// next line will pop an item, or wait until it is possible to do so
item := <- queue
```

```go
var ok bool
select {
    case item = <- queue:
        ok = true
    default:
        ok = false
}
// at this point, "ok" is:
//   true  => item was popped off the queue (or queue was closed, see below)
//   false => not popped, would have blocked because of queue empty
```

Tips for non-blocking operations in Go

- Closing buffered channels
  - Consider a channel of 'Item'. E.g. int `queue := make(chan Item, 10) // queue with a capacity of 10`
  - For producer
    - Buffered channels are best closed by producers `close(queue)  // closes the queue`
    - The channel close event is signaled to the consumers
    - You can not write to closed channels
  - For consumer
    - If there are items yet to be popped off, the popping off happens as usual.
    - When the queue is empty *and* closed, the read will *not* block. It returns "zero-value" of the channel item type.

- For consumer

A solution: to know if the received value is **valid**

```
for {
    item, valid := <- queue
    if !valid {
        break
    }
    // process
}
// at this point, all items ever pushed into the queue
 has been processed,
// and the queue has been closed
```

OR

```
for item := range queue {
    // process
}
```

- For consumer

We can also combine the non-blocking and valid checks into one

```
var ok, valid bool
select {
    case item, valid = <- queue:
        ok = true
    default:
        ok = false
}
// at this point:
//   ok && valid  => item is good, use it
//   !ok          => channel open, but empty, try later
//   ok && !valid => channel closed, quit polling
```

Example: Producer-Consumer in Go

- Original: https://play.golang.org/p/a_HBATTBu54
- Test with buffered channel
  - 1. Add the channel dimension. E.g. 5
    - Why the output prints only until 6?
  - 2. Try to run 'consume' not as a gorountine
    - Eliminate the 'go' directive
    - Why occurs a deadlock at the end?
  - A possible answer
    - Producer close the channel after his production
    - Consumer consumes everything in buffer, but it knows when it is empty and closed
    - To keep 'consume' as goroutine, it needs to inform the 'main' when finishing (idea similar to 'done' channel)
    - https://play.golang.org/p/B0tLTKJzbhX

```go
package main
/* producer-consumer problem in Go */


import ("fmt")

var done = make(chan bool)
var msgs = make(chan int, 5)

func produce () {
    for i := 0; i < 10; i++ {
        msgs <- i
    }
    done <- true
}

func consume () {
    for {
      msg := <-msgs
      fmt.Println(msg)
   }
}

func main () {
   go produce()
   go consume()
   <- done
}
```

Other benefits of Go

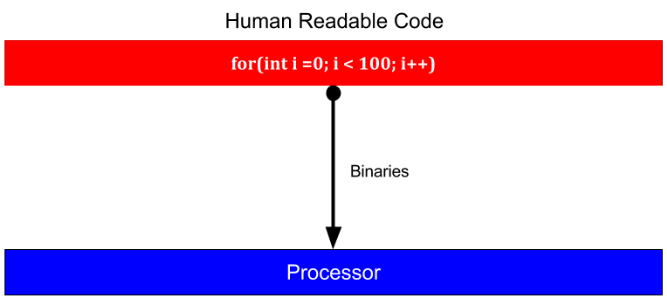
- Goroutines have growable segmented stacks
  - It means they will use more memory only when needed.
- Goroutines have a faster startup time than threads
- Goroutines come with built-in primitives to communicate safely between themselves (channels)
- Goroutines and OS threads do not have 1:1 mapping
  - A single goroutine can run on multiple OS threads
  - Goroutines are multiplexed into small number of OS threads

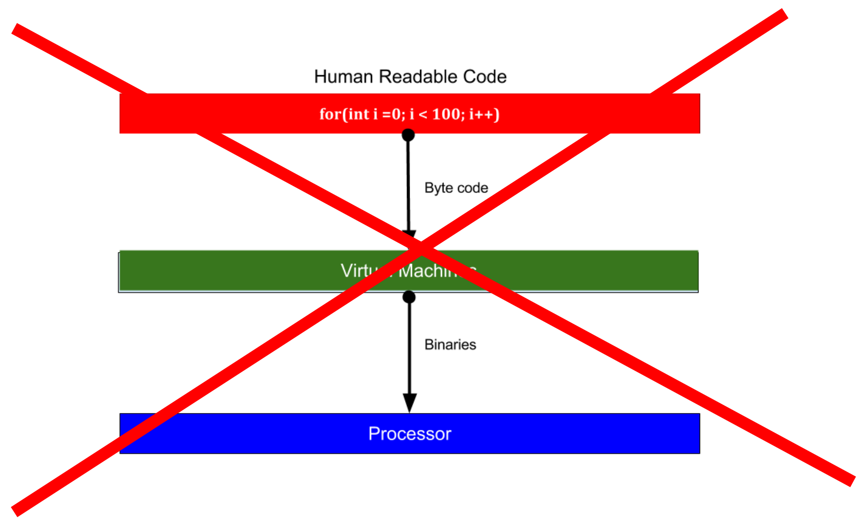
Other benefits of Go

- Go uses Static Type System
  - Type system is really important for large scale applications.

- Go uses garbage collection to allocation and removal of the object
  - So, no more malloc() and free() statements!!!

- Go is compiled language.
  - It means performance is almost nearer to lower level languages.





Install Go

- Install:
  - http://golang.org/doc/install.html

- IDEs

Editores de texto

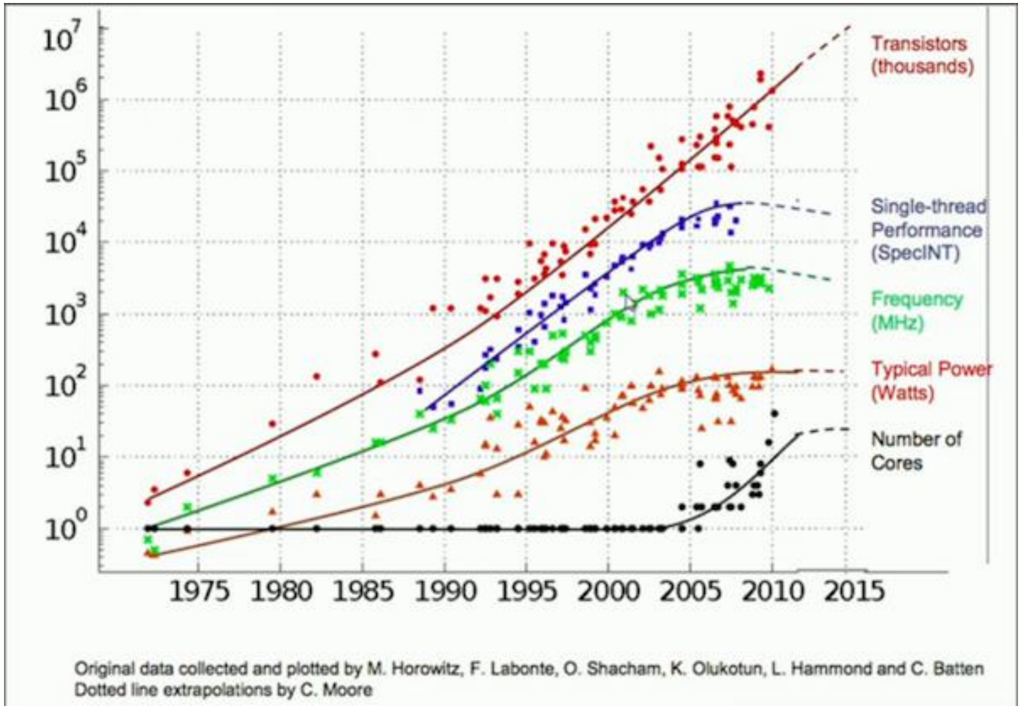
A simple, open source, cross-platform Go IDE. With debugging

IDE	LICENSE	WINDOWS	LINUX	MAC OS X	OTHER PLATFORMS
SublimeText 2	Proprietary software	Yes	Yes	Yes	-
TextMate	Proprietary software	No	No	Yes	-
IntelliJ	Apache 2.0	Yes	Yes	Yes	JVM
LiteIDE	LGPL	Yes	Yes	Yes	-
Intype	New BSD Licence	Yes	No	No	
Netbeans	Free	Yes	Yes	Yes	JVM
Eclipse	Eclipse Public License 1.0	Yes	Yes	Yes	JVM
Komodo Edit	Proprietary	Yes	Yes	Yes	-
Zeus	Proprietary	Yes	Yes (Wine)	No	-

Extras

Why do we need concurrency?

- Hardware limitations
  - Comparison of increasing the processing power with the time:



- **Single-thread performance** and the **frequency** of the processor remained steady for almost a decade.
- Adding more transistors is not the solution: cost and quantum properties (like tunneling)

Why do we need concurrency?

- Solutions for hardware limitations
  - More cores to the processor → Cost
  - More cache to the processor → Physical limits: the bigger the cache, the slower it gets
  - Hyper-threading

CONCURRENC Y

If we cannot rely on the hardware improvements, the only way to go is more efficient software to increase the performance