GO

BOOTCAMP

Clase en vivo

//Go Bases

IT BOARDING

BOOTCAMP

Objetivos de esta clase

- Conocer los paquetes fmt, os e io
- Entender qué es concurrencia y qué es paralelismo.
- Aprender qué es una go routine

Índice

IT BOARDING BOOTCAMP

1 | Repaso

IT BOARDING

BOOTCAMP

Paquete fmt

Verbos de impresión más utilizados

%v	valor en formato estándar
%T	tipo de dato del valor a imprimir
%t	bool
%s	string
%f	punto flotante
%d	entero decimal
%**b**	un entero binario
%o	octal
%c	imprime caracteres
%p	dirección de memoria

.Print( )

En este ejemplo definimos e inicializamos las variables “**nombre**” y “**edad**”. Ya que la función Print( ) recibe n número de parámetros y de cualquier tipo, podemos pasarle estas variables, y concatenarlas con cadenas definidas dentro de la función.

“**\n**” es un carácter de escape que indica un salto de línea.

```
{}
nombre, edad := "Kim", 22
fmt.Print(nombre, " tiene ", edad, " años de edad.\n")
```

.Sprint( )

La función **Sprint(a ...interface{ }) string** toma como parámetro n variables de cualquier tipo y las formatea como un string de manera estándar según su tipo.
El valor de retorno es la concatenación de estas variables condensadas en un string.

En este ejemplo usamos la función **Sprint( )** para guardar en una variable la concatenación de todas las variables pasadas por parámetro y luego usamos la función **Print( )** para imprimir la variable generada.

```
{}
nombre, edad := "Kim", 22
res := fmt.Sprint(nombre, " tiene ", edad, " años de edad.\n")
fmt.Print(res)
```

Paquete os

.ReadFile( )

La función **ReadFile(filename string)** recibe como parámetro la dirección y nombre del archivo en formato texto y nos devuelve el contenido del archivo en bytes o un error en caso que lo haya.

```
{}  data, err := os.ReadFile("./myFile.txt")
```

.Exit( )

La función **Exit(code int)** hace que el programa termine inmediatamente con el código de estado dado. Convencionalmente, el código cero indica éxito, el no cero es un **error**. Las funciones diferidas no se ejecutan.

```
{}    os.Exit(1)
```

Paquete io

ReadAll

ReadAll(r Reader) lee desde r hasta un error o EOF y devuelve los datos que leyó y un error si lo hubiera.

```
{}
r := strings.NewReader("some io.Reader stream to be read")

b, err := io.ReadAll(r);
```

Copy

Copy(dst Writer, src Reader) tiene un comportamiento similar a **ReadAll** con la particularidad de que ya incluye el destino(**dst**).

```
{}
r := strings.NewReader("some io.Reader stream to be read")

_, err := io.Copy(os.Stdout, r)
```

WriteString

La función **WriteString(w Writer, s string)** que toma una cadena de texto y un Writer, escribe el contenido de s a w, que acepta una slice de bytes.

```
{}    io.WriteString(os.Stdout, "Hello world!")
```

2 | Puesta en común

IT BOARDING

BOOTCAMP

Puesta en común

3 Concurrencia y Paralelismo

IT BOARDING

BOOTCAMP

// Algunas aclaraciones antes de empezar

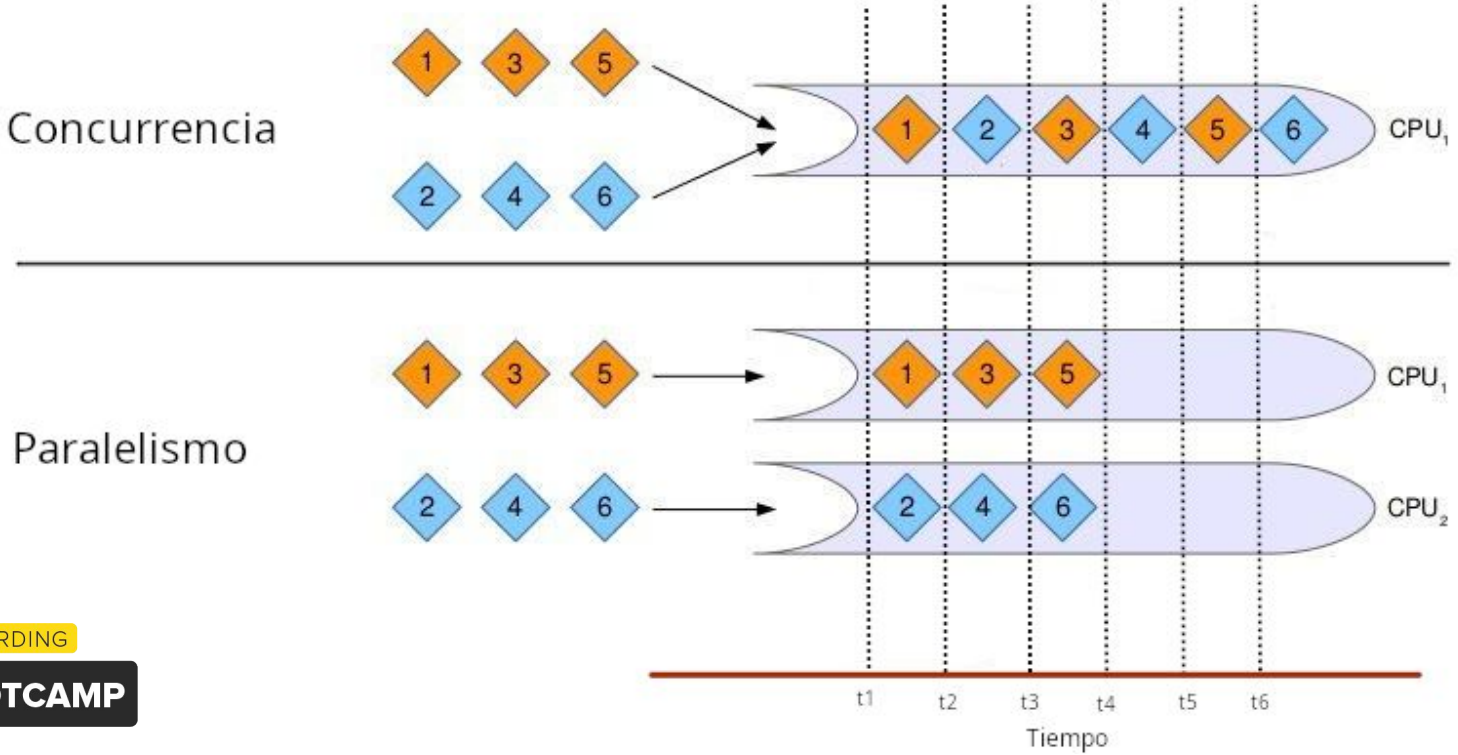
A continuación vamos a aprender las bases de concurrencia, paralelismo, Go Routines y canales. Estos conceptos nos van a servir al momento que estemos desarrollando nuestros programas.

Durante la clase de hoy abordaremos las **bases de los temas**. Es importante aclarar que durante el Bootcamp no se profundizará sobre los mismos debido a su nivel de complejidad.

// ¿Qué son la concurrencia y el paralelismo?

- Concurrencia es el acto de iniciar, ejecutar y completar dos o más tareas en **períodos de tiempo superpuestos.** No significa necesariamente que ambos se estén ejecutando en el mismo instante.
- Paralelismo implica que dos o más tareas se ejecuten **exactamente al mismo tiempo**.

// Concurrencia vs. Paralelismo



IT BOARDING

BOOTCAMP

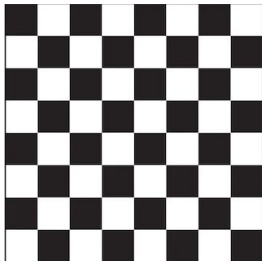
// Paralelismo #1

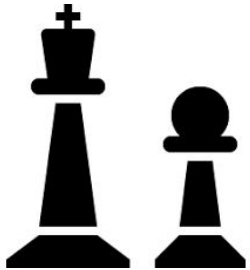
Imaginemos un torneo de ajedrez en el que entran 100 personas a participar pero tiene como límite de tiempo un día.

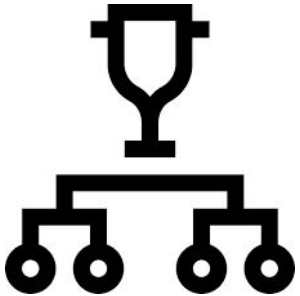
Para lograr terminar a tiempo se tienen que jugar varias partidas simultáneamente. Aunque una partida no afecta el resultado de otra que se esté jugando al mismo tiempo, esas partidas sí afectan partidas futuras si el torneo se pensó en varias etapas.

// Paralelismo #2

La cantidad de partidas simultáneas que se puedan jugar dependerá tanto del número de jugadores como de los recursos disponibles: lugares para jugar, tableros, conjuntos de piezas, etc, y del diseño del torneo.





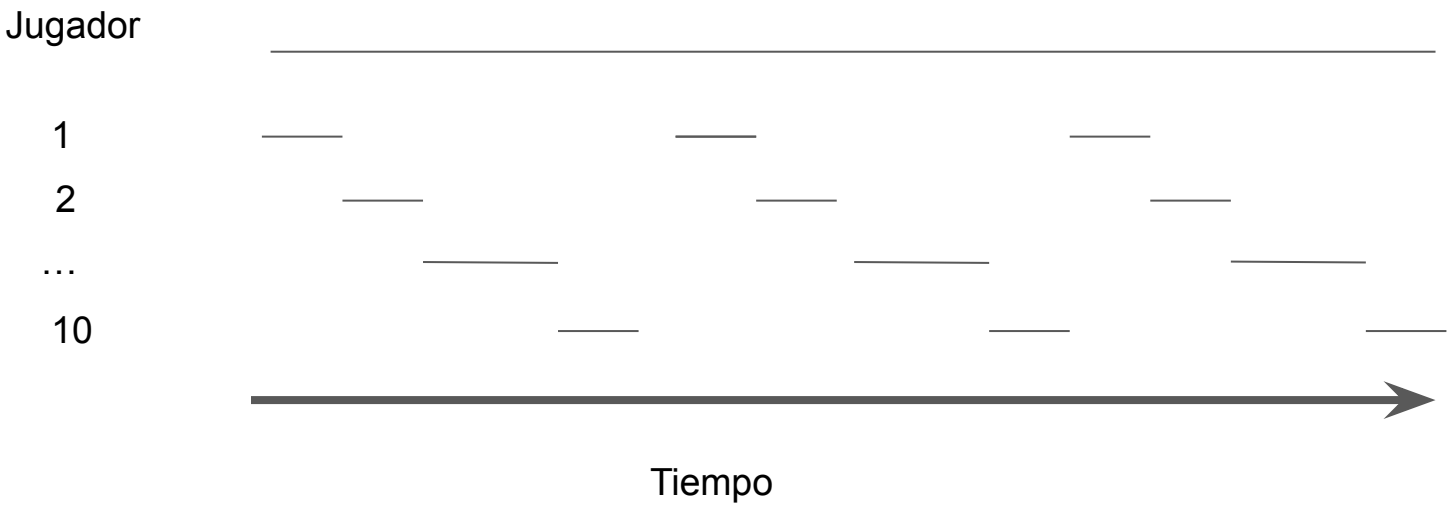


// Concurrencia #1

Ahora imaginemos que en ese torneo hay un gran maestro del ajedrez al que retan a enfrentarse con 10 personas a ciegas al mismo tiempo. El ajedrecista acepta y juega contra los 10 formados en círculo.

Para lograr esto (además de jugar muy bien ajedrez), el gran maestro debe establecer un orden para cada oponente y, por lo menos en su mente, una manera de retomar cada juego cuando sea el turno.

// Concurrencia #2



IT BOARDING
BOOTCAMP

// Concurrencia #3

Las tareas (los juegos), vistos desde el lado del GM están relacionadas en el sentido de que cada una suma dificultad al encuentro, cada uno hace más difícil ganar al siguiente. Este proceso no puede paralelizarse porque perdería todo el sentido.

4 | Go Routines

IT BOARDING

BOOTCAMP

// ¿Qué son las Goroutines?

- Son la solución que nos ofrece Go para implementar concurrencia.
- **¡NO SON THREADS!** Las Goroutines son gestionadas por el **Go Runtime** y su scheduler, **no por el sistema operativo.**
- Cuando ejecutamos una función como Goroutine, su ejecución no bloqueará la continuación de aquella función que la invocó, porque correrá de forma concurrente con esta.

Función a procesar

En este caso tenemos una función que se encargará de hacer una determinada tarea. Para ello agregaremos una pausa de un segundo en él, para simular una funcionalidad que accede a una Base de Datos o envía información a una API.

```
{}
func procesar(i int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
}
```

Procesar en hilo principal

Ejecutaremos nuestra función 10 veces, veremos que para que inicialice el 2do procesamiento el 1ero debería haber terminado.

```
{}
func main() {
    for i := 0; i < 10; i++{
        procesar(i)
    }
}
```

Pausar fin ejecución

Agregaremos una pausa de 5 segundos antes de terminar el programa, ya veremos por qué...

```
{}
func main() {
    for i := 0; i < 10; i++{
        procesar(i)
    }
    time.Sleep(5000 * time.Millisecond)
    fmt.Println("Termino el programa")
}
```

// ¿Cómo podríamos ejecutar la función en simultáneo?

Aquí entran en acción las Goroutines.

Al ser tareas completamente independientes entre sí, en lugar de ejecutar secuencialmente **procesar()**, podríamos tratar a esas llamadas como **goroutines** y cada nueva invocación no debería esperar a que la anterior termine.

Go Routine

Para indicar que queremos ejecutar la función en una Go Routine lo hacemos con la palabra reservada **go**.
Vemos cómo se ejecuta de forma simultánea y no de forma lineal:

```
{}
```

```go
func main() {
    for i := 0; i < 10; i++{
        go procesar(i)
    }
    time.Sleep(5000 * time.Millisecond)
    fmt.Println("Termino el programa")
}
```

Go Routine

Si quitamos la pausa de 5 segundos que agregamos al final se terminará el programa antes que terminen las Go Routines.
¿Cómo podríamos hacer que el programa espere hasta que terminen todas las rutinas?

```
{}
func main() {
    for i := 0; i < 10; i++{
        go procesar(i)
    }
}
```

Una alternativa para ello es usar **Canales**.

A codear...

5 Canales

IT BOARDING

BOOTCAMP

Un canal nos va a permitir enviar valores a las Go Routines y esperar hasta recibir dicho valor.

Sintaxis

Para definir un canal de tipo entero lo hacemos de la siguiente manera: utilizando la palabra reservada **chan.**

```
{}  c := make(chan int)
```

Solucionando el problema planteado

Anteriormente propusimos usar canales como solución al intercambio de datos entre rutinas concurrentes.

¿Cómo lo implementamos?

Recibir canal como parámetro

Debemos recibir como parámetro el canal en nuestra función, para ello lo definimos como **chan int** al ser un canal de enteros:

```
{}
func procesar(i int, c chan int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
}
```

Enviar valor a canal

Una vez que terminamos de procesar debemos enviarle el valor al canal, en este caso le enviaremos el valor de **i**:

{}

```
func procesar(i int, c chan int) {
    fmt.Println(i, "-Inicia")
    time.Sleep(1000 * time.Millisecond)
    fmt.Println(i, "-Termina")
    c <- i
}
```

Ejecutar ejemplo

Vamos a correr el programa ejecutando solo una función.
Vemos que sigue terminando el programa antes que termine la rutina.
¿Qué pasó? ¿Qué nos está faltando?

{}

```
func main() {
    c := make(chan int)
    go procesar(1, c)
    fmt.Println("Termino el programa")
}
```

Recibir valor

Nos está faltando recibir el valor del canal, para que el programa quede esperando hasta recibir ese valor.
Para indicarle al programa que estamos esperando para recibir el valor del canal de la variable **c** lo hacemos con **<-c**

```
<-c // recibimos el valor del canal
variable := <-c // recibimos y lo asignamos a una variable
fmt.Println("Termino el programa en ", <-c) // recibimos y lo imprimimos
```

Ejecutar ejemplo

De esta manera, el programa terminará una vez que se le asigne el valor en el canal:

{}

```
func main() {
    c := make(chan int)
    go procesar(1, c)
    fmt.Println("Termino el programa")
    <-c
}
```

Ejecutar ejemplo con for

```
{}
```

```go
func main() {
    c := make(chan int)

    for i := 0; i < 10; i++{
        go procesar(i, c)
    }

    for i := 0; i < 10; i++{
        fmt.Println("Termino el programa en ", <-c)
    }
}
```

A codear...

Conclusiones

En esta clase trabajamos sobre los conceptos de concurrencia y paralelismo en los sistemas.

Además aprendimos el uso las go routines para nutrir de asincronismo a nuestros programas y también de cómo utilizar los canales para comunicarlas.

No te comuniques compartiendo la memoria, comparte la memoria comunicándote.

Gracias.

IT BOARDING

BOOTCAMP