CPU lê a memória em "palavras", 4 bytes em sistema de 32 bits, e 8 bytes em um sistema de 64 bits. Então...?

Vamos considerar a seguinte código

```
package main
import (
  "fmt"
  "unsafe"
)
type CarlosResource struct {
  Nome         string // 16 bytes
  Sobrenome    string // 16 bytes
  HaveDog      bool   //  1 byte
  NomeDog      string // 16 bytes
  IsProgrammer bool   //  1 byte
  Endereco     string // 16 bytes
  Idade        int32  //  4 bytes
}
func main() {
  var c CarlosResource
  c.Nome = "Carlos"
  c.Sobrenome = "Anhaya"
  c.HaveDog = true
  c.NomeDog = "Capitu"
  c.IsProgrammer = true
  c.Endereco = "NDA"
  c.Idade = 29

  fmt.Println("================================================================")
  fmt.Printf("Total Memory Usage StructType:d %T => [%d]\n", c, unsafe.Sizeof(c))
  fmt.Println("================================================================")
```

```
    fmt.Printf("Nome Field StructType:d.Nome %T => [%d]\n", c.Nome, unsafe.Sizeof(c.Nome))
    fmt.Printf("Sobrenome Field StructType:d.Sobrenome %T => [%d]\n", c.Sobrenome,
unsafe.Sizeof(c.Sobrenome))
    fmt.Printf("HaveDog Field StructType:d.HaveDog %T => [%d]\n", c.HaveDog, unsafe.Sizeof(c.HaveDog))
    fmt.Printf("NomeDog Field StructType:d.NomeDog %T => [%d]\n", c.NomeDog, unsafe.Sizeof(c.NomeDog))
    fmt.Printf("IsProgrammer Field StructType:d.IsProgrammer %T => [%d]\n", c.IsProgrammer,
unsafe.Sizeof(c.IsProgrammer))
    fmt.Printf("Endereco Field StructType:d.Endereco %T => [%d]\n", c.Endereco, unsafe.Sizeof(c.Endereco))
    fmt.Printf("Idade Field StructType:d.Idade %T => [%d]\n", c.Idade, unsafe.Sizeof(c.Idade))
}
```

Quando executado, iremos perceber que o total é 88, **porém se somarmos o total de cada campo não dá 88…**

Como disse, sistemas de 32bits = blocos de 4 Bytes, sistemas de 64bits, blocos de 8 bytes. Isso significa que no nosso caso o bool que tem 1 byte e que está entre as strings, passará a ter 7 bytes "vazios", isso porque ele está entre tipos de variáveis que são maiores do que bool. O mesmo vale para o int32 que está na última posição.

Para resolver esse "problema", precisamos refatorar da seguinte maneira:

```
type CarlosResource struct {
    Nome        string // 16 bytes
    Sobrenome   string // 16 bytes
    NomeDog     string // 16 bytes
    Endereco    string // 16 bytes
    HaveDog     bool   //  1 byte
    IsProgrammer bool  //  1 byte
    Idade       int32  //  4 bytes
}
```

Neste caso, teremos o total igual 72, pois as strings estão declaradas primeiro, bool e int32 por último, o que significa que eles ocuparão uma palavra de leitura uma vez que

teremos *HaveDog; IsProgrammer e Idade* no mesmo bloco, o que resulta em 6 bytes e consequentemente 2 bytes vazios.

Ou seja, a ordem em que estão declarados os atributos dentro da struct importa, importa pois isso evitará consumo de memória e ciclos de leitura da CPU.

Eu ainda acho que um dia o Go irá implementar isso em tempo de compile, mas até lá, vale a dica =)

Referências

https://www.geeksforgeeks.org/data-structure-alignment-how-data-is-arranged-and-accessed-in-computer-memory/

https://towardsdev.com/golang-writing-memory-efficient-and-cpu-optimized-go-structs-62fcef4dbfd0