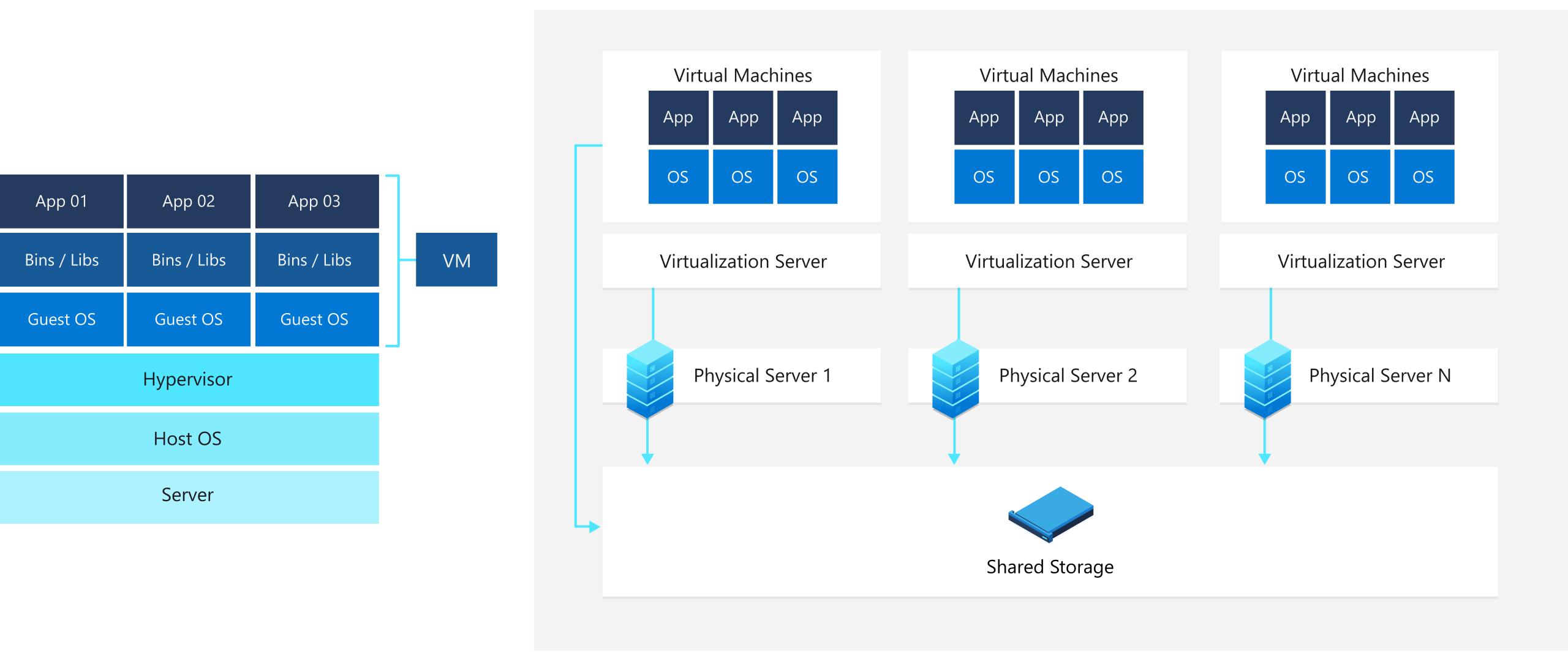
**What is Virtual Machines?**

Virtual machine, commonly shortened to just VM, is no different than any other physical computer like a laptop, smart phone, or server. It has a CPU, memory, disks to store your files, and can connect to the internet if needed.

VMs are often thought of as virtual computers or software-defined computers within physical servers, existing only as code.



**How does Virtual Machine works?**

A virtual machine is a computer file, typically called an image, that behaves like an actual computer. It can run in a window as a separate computing environment, often to run a different operating system or even to function as the user's entire computer experience as is common on many people's work computers.

The virtual machine is partitioned from the rest of the system, meaning that the software inside a VM can't interfere with the host computer's primary operating system.

--------------------------------------------------------------------------------------

Next I have Installed the Virtual Box and also the Ubuntu via virtual box

**Go Lang**

**Go Functions:**

A function is a group of statements that together perform a task. Every Go program has at least one function, which is main().

A function declaration tells the compiler about a function name, return type, and parameters. A function definition provides the actual body of the function.

**Defining a Function**:

func function\_name([parameter list])[return type]

{

Body of the function

}

**function\_name**: It is the name of the function.

**parameter-list**: It contains the name and the type of the function parameters.

**return\_type\_list**: It is optional and it contains the types of the values that function returns. If you are using return\_type in your function, then it is necessary to use a return statement in your function.

**Multiple return Values:**

In Go language, you are allowed to return multiple values from a function, using the return statement. Or in other words, in function, a single return statement can return multiple values. The type of the return values is similar to the type of the parameter defined in the parameter list.

Program:

// function return multiple values

package main

import "fmt"

func myfunc(p, q int)(int, int, int ){

return p - q, p \* q, p + q

}

func main() {

var myvar1, myvar2, myvar3 = myfunc(4, 2)

fmt.Printf("Result is: %d", myvar1 )

fmt.Printf("\nResult is: %d", myvar2)

fmt.Printf("\nResult is: %d", myvar3)

}

Output:

Result is: 2

Result is: 8

Result is: 6

--------------------------------------------------------------------------------------------

**Defer Function:**

Defer function or method call arguments evaluate instantly, but they don’t execute until the nearby functions returns. You can create a deferred method, or function, or anonymous function by using the defer keyword.

Important Points in Defer:

1.multiple defer statements are allowed in the same program and they are executed in LIFO.

2.the arguments are evaluated when the defer statement is executed, not when it is called.

Syntax:

// Function

defer func func\_name(parameter\_list Type)return\_type{

// Code

}

// Method

defer func (receiver Type) method\_name(parameter\_list){

// Code

}

**Program:**

package main

import “fmt”

func mul(a1,a2,int)int

{

res:=a1\*a2

fmtPrintln(“Result:”,res)

return 0;

}

func show()

{

fmtprintln(“Hello”);

}

Func main()

{

Mul(23,45)

defer mul(23,56)

show()

}

**Output:**

Result: 1035

Hello

Result: 1288

**Go Map**

Maps is a collection of unordered pairs of key-value. It is widely used because it provides fast lookups and values that can retrieve, update or delete with the help of keys.

In Go language, maps can create and initialize using two different ways:

1. **Simple**: In this method, you can create and initialize a map without the use of make() function:

**Syntax:**

// An Empty map:

map[Key\_Type]Value\_Type{}

// Map with key-value pair:

map[Key\_Type]Value\_Type{key1: value1, ..., keyN: valueN}

**Program:**

Note**: Initializing map using map literals**

package main

import "fmt"

func main() {

var map\_1 map[int]int

// Checking if the map is nil or not

if map\_1 == nil {

fmt.Println("True")

} else {

fmt.Println("False")

}

// Using shorthand declaration and

// using map literals

map\_2 := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

fmt.Println("Map-2: ", map\_2)

}

Output:

True

Map-2: map[90:Dog 91:Cat 92:Cow 93:Bird 94:Rabbit]

**Using Make Function**:

Syntax:

make(map[Key\_Type]Value\_Type)

Program:

package main

import "fmt"

func main() {

// Using make() function

var My\_map = make(map[float64]string)

fmt.Println(My\_map)

// As we already know that make() function

// always returns a map which is initialized

// So, we can add values in it

My\_map[1.3] = "Rohit"

My\_map[1.5] = "Sumit"

fmt.Println(My\_map)

}

Output:

map[]

map[1.3:Rohit 1.5:Sumit]

**How to iterate over a map?:** You can iterate a map using the range for loop. The value of this loop may vary because the map is an unordered collection.

Program:

package main

import “fmt”

func main()

{

var m\_a\_p:=map[int]string

{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

for id,pet:=range m\_a\_p

{

fmt.Println(id.pet)

}

}

Output:

90 Dog

91 Cat

92 Cow

93 Bird

94 Rabbit

**Adding key-value pairs in the map:**

In maps, you are allowed to add key-value pairs in the initialized map using the given syntax:

Syntax:

map\_name[key]=value

Program:

package main

import "fmt"

// Main function

func main() {

// Creating and initializing a map

m\_a\_p := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

fmt.Println("Original map: ", m\_a\_p)

// Adding new key-value pairs in the map

m\_a\_p[95] = "Parrot"

m\_a\_p[96] = "Crow"

fmt.Println("Map after adding new key-value pair:\n", m\_a\_p)

// Updating values of the map

m\_a\_p[91] = "PIG"

m\_a\_p[93] = "DONKEY"

fmt.Println("\nMap after updating values of the map:\n", m\_a\_p)

}

Output:

Original map: map[90:Dog 91:Cat 92:Cow 93:Bird 94:Rabbit]

Map after adding new key-value pair:

map[90:Dog 91:Cat 92:Cow 93:Bird 94:Rabbit 95:Parrot 96:Crow]

Map after updating values of the map:

map[90:Dog 91:PIG 92:Cow 93:DONKEY 94:Rabbit 95:Parrot 96:Crow]

**Retrieve a value related to a key in the maps:**

// Go program to illustrate how to

// retrieve the value of the key

package main

import "fmt"

// Main function

func main() {

// Creating and initializing a map

m\_a\_p := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

fmt.Println("Original map: ", m\_a\_p)

// Retrieving values with the help of keys

value\_1 := m\_a\_p[90]

value\_2 := m\_a\_p[93]

fmt.Println("Value of key[90]: ", value\_1)

fmt.Println("Value of key[93]: ", value\_2)

}

Output:

Original map: map[90:Dog 91:Cat 92:Cow 93:Bird 94:Rabbit]

Value of key[90]: Dog

Value of key[93]: Bird

**Delete key from the map:**

In the map using the delete() function. It is inbuilt function and does not return any value and does not do anything if the key does not present in the given map.

Program:

package main

import "fmt"

// Main function

func main() {

// Creating and initializing a map

m\_a\_p := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

93: "Bird",

94: "Rabbit",

}

fmt.Println("Original map: ", m\_a\_p)

// Deleting keys

// Using delete function

delete(m\_a\_p, 90)

delete(m\_a\_p, 93)

fmt.Println("Map after deletion: ", m\_a\_p)

}

Output:

Original map: map[90:Dog 91:Cat 92:Cow 93:Bird 94:Rabbit]

Map after deletion: map[91:Cat 92:Cow 94:Rabbit]

Modifying Map:

Program:

// Go program to illustrate the

// modification concept in map

package main

import "fmt"

// Main function

func main() {

m\_a\_p := map[int]string{

90: "Dog",

91: "Cat",

92: "Cow",

}

fmt.Println("Original map: ", m\_a\_p)

// Assigned the map into a new variable

new\_map := m\_a\_p

// Perform modification in new\_map

new\_map[96] = "Parrot"

new\_map[98] = "Pig"

// Display after modification

fmt.Println("New map: ", new\_map)

fmt.Println("\nModification done in old map:\n", m\_a\_p)

}

Output:

Original map: map[90:Dog 91:Cat 92:Cow ]

New map: map[90:Dog 91:Cat 92:Cow 96:Parrot 98:Pig]

Modification done in old map: map[90:Dog 91:Cat 92:Cow 93 96:Parrot 98:Pig]

--------------------------------------------------------------------------------------------

**Panic in Golang:**

Panic means an unexpected condition arises in your Go program due to which the execution of your program is terminated.

It is deliberately thrown by the programmer to handle the worst-case scenario in the Go program with the help of panic() function

**Syntax:**

func panic(v interface{})

**Program:**

package main

import "fmt"

func entry(lang \*string, aname \*string) {

if lang == nil {

panic("Error: Language cannot be nil")

}

if aname == nil {

panic("Error: Author name cannot be nil")

}

fmt.Printf("Author Language: %s \n Author Name: %s\n", \*lang, \*aname)

}

// Main function

func main() {

A\_lang := "GO Language"

entry(&A\_lang, nil)

}

**Output:**

panic: Error: Author name cannot be nil

goroutine 1 [running]:

main.entry(0x41a788, 0x0)

/tmp/sandbox108627012/prog.go:20 +0x140

main.main()

/tmp/sandbox108627012/prog.go:37 +0x40

**Using Defer while Panicking:**

Defer statement is used in it

**Program:**

package main

import (

"fmt"

)

func entry(lang \*string, aname \*string) {

defer fmt.Println("Defer statement in the entry function")

if lang == nil {

panic("Error: Language cannot be nil")

}

if aname == nil {

panic("Error: Author name cannot be nil")

}

fmt.Printf("Author Language: %s \n Author Name: %s\n", \*lang, \*aname)

}

// Main function

func main() {

A\_lang := "GO Language"

defer fmt.Println("Defer statement in the Main function")

entry(&A\_lang, nil)

}

Output:

Defer statement in the entry function

Defer statement in the Main function

panic: Error: Author name cannot be nil

goroutine 1 [running]:

main.entry(0x41a780, 0x0)

/tmp/sandbox121565297/prog.go:24 +0x220

main.main()

/tmp/sandbox121565297/prog.go:44 +0xa0

**Recover in Golang:**

Recover function is used to handle panic. It is an inbuilt function which is defined under the builtin package of the Go language.

The main use of this function is to regain control of a panicking Goroutine.

**Syntax:**

func recover() interface{}

**Program:**

package main

import ("fmt")

func handlepanic() {

if a := recover(); a != nil {

fmt.Println("RECOVER", a)

}

}

func entry(lang \*string, aname \*string) {

defer handlepanic()

if lang == nil {

panic("Error: Language cannot be nil")

}

if aname == nil {

panic("Error: Author name cannot be nil")

}

fmt.Printf("Author Language: %s \n Author Name: %s\n", \*lang, \*aname)

fmt.Printf("Return successfully from the entry function")

}

// Main function

func main() {

A\_lang := "GO Language"

entry(&A\_lang, nil)

fmt.Printf("Return successfully from the main function")

}

Output:

RECOVER Error: Author name cannot be nil

Return successfully from the main function

----------------------------------------------------------------------------------------------------