Parallel Computing

Lab (2) Notes

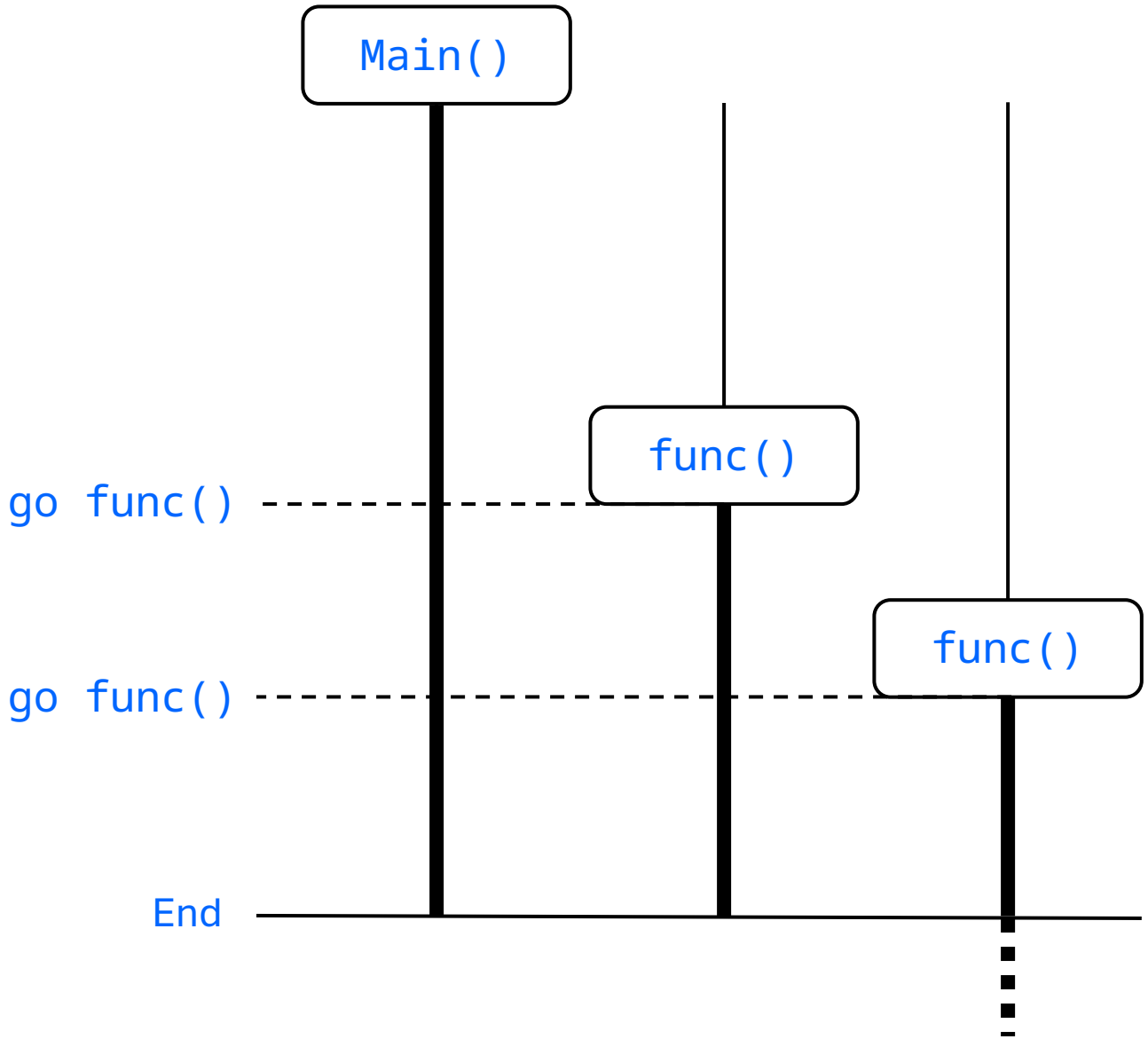
Autumn 2024-25

Review

- Concurrency manages multiple tasks simultaneously.
- Parallelism executes multiple tasks simultaneously.
- A thread is an independent execution units within a process.

Goroutines

- Goroutines are lightweight threads of execution that are managed by the Go runtime.
- Use `go` keyword to run a normal function as a goroutine.



Synchronization is needed.

Channels

- Goroutines can communicate with each other using channels.
- Channels provide a safe and efficient way to synchronize and share data between concurrent tasks.

```go
package main

import(
        "fmt"
        "time"
)

func printNumbers(num int){
        for i:=1; i <= num; i++ {
                fmt.Printf("%d\n", i)
                time.Sleep(time.Second)
        }
}

func printLetters(char rune){
        for i:='A'; i <= char; i++{
                fmt.Printf("%c\n", i)
                time.Sleep(time.Second)
        }
}


func main(){
        now := time.Now()
        defer func(){
                fmt.Println("Execution time (Concurrent): ", time.Since(now))
        }() // This function will execute at the end of the main.

        go printNumbers(5)
        printLetters('E')

        /* printNumbers(3)
        var fname string = "Ahmed"
        lname := "Mahmoud"
        fmt.Println("Hello, "+fname+" "+lname) */
}
~
~
~
~
```

```
main.go [unix] (23:42 02/10/2024)                    29,4-11 All
"main.go" [unix] 36L, 591B written
```

```
STD+qassas.ahmed@qassas MINGW64 ~/OneDrive - Mansoura University - Main/Matrouh/parallel-computing/parallel/lab_1
$ go run main.go
1
2
3
4
5
A
B
C
D
E
Execution time (Sequential):  10.0089237s

STD+qassas.ahmed@qassas MINGW64 ~/OneDrive - Mansoura University - Main/Matrouh/parallel-computing/parallel/lab_1
$ go run main.go
A
1
2
B
C
3
D
4
5
E
Execution time (Concurrent):  5.0049277s

STD+qassas.ahmed@qassas MINGW64 ~/OneDrive - Mansoura University - Main/Matrouh/parallel-computing/parallel/lab_1
$
```

```
}

func copy_simulation(num int, char rune, done chan bool){
        for i:=num; i>=0; i--{
                fmt.Printf("%c",char)
                time.Sleep(time.Second)
        }
        done<-true
}

func main(){
        now:=time.Now()
        defer func(){
                fmt.Println("\nExecution time:", time.Since(now))
        }()

        signal:= make(chan bool)

        go copy_simulation(20, '*', signal)

        var input_num int
        fmt.Printf("Enter number: ")
        fmt.Scan(&input_num)
        answer := factorial(input_num)

        fmt.Printf("\n%d! = %d\n", input_num, answer)
        <-signal

//      go printNumbers(5)
//      printLetters('E')
```

Buffered Channels

Buffered channels are commonly used in scenarios where production of data and its consumption are not always synchronized but still require a controlled flow.

```
        for i:=num; i>=0; i--{
                fmt.Printf("%c",char)
                time.Sleep(time.Second)
        }
        done<-true
}

func main(){
        now:=time.Now()
        defer func(){
                fmt.Println("\nExecution time:", time.Since(now))
        }()

        signal:= make(chan bool, 2)

        go copy_simulation(20, '*', signal)
        go copy_simulation(25, '-', signal)

        var input_num int
        fmt.Printf("Enter number: ")
        fmt.Scan(&input_num)
        answer := factorial(input_num)

        fmt.Printf("\n%d! = %d\n", input_num, answer)
        <-signal
        <-signal

//      go printNumbers(5)
//      printLetters('E')
}
```

Thank You