Intelligent programming

Lecture Nine

Arrays in Go language

The type [n]T is an array of n values of type **T**.

The expression

var array1 [10] int

declares a variable **array1** as an array of **ten integers**.

For loop in Go

```
for initialization; condition; post {
// statements....
}
```

For loop in Go

```
func main() {


      for i := 0; i < 4; i++ {
    fmt.Printf(“FCDS \n")
   }


}
```

```
FCDS
FCDS
FCDS
FCDS
```

For as while loop

```
for condition {
    // statement..
}
```

For as while loop

```
package main
import ("fmt")

func main() {
  number := 1

  for number <= 5 {
    fmt.Println (number)
    number++
  }
}
```

Write a for loop that gets the average of an array

```
func main() {

      array := [   ] int {1, 2, 3, 4}

       n := 4

      sum := 0

         for i := 0; i < len(array); i++ {

                sum += (array[i])
     }

         avg :=  sum / n
         fmt.Println("Sum = ", sum, "\nAverage = ", avg)
}
```

Infinite loop using for

```
package main
import "fmt"
func main(){
  number := 1
for  {
if number > 5 {
      break;
    }
    fmt.Printf("%d\n", number);
    number ++
  }
}
```

range in for loop

Syntax:

```
for i, j:= range rvariable {
   // statement..
}
```

The variable **i is initialized as 0** and is defined to **increase** at every iteration until it reaches the value of the **length of the array**.

i contains the index

j contains the value

```
import "fmt"

func main() {

	arra := [ ]int{1, 2, 3, 4}

	for index, itr := range arra {

		fmt.Print(index, " : ", itr, "\n")
	}
	  for itr := range arra {

	   fmt.Print(itr, " ")
	}
}
```

```
0:1
1:2
2:3
3:4
1234
```

```
import "fmt"

func main() {

        rvariable:= [ ]string{“Faculty", “of", “computing"}


        for i, j:= range rvariable {
       fmt.Println(i, j)
    }

}
```

Output:

0 Faculty
1 of
2 computing

```
package main

import "fmt"

func main() {
	pow := make([]int, 5)
	for i := range pow {
		pow[i] = i*i
	}
	for _, value := range pow {
		fmt.Printf("%d\n", value)
	}
}
```

```
0
1
4
9
16
```

Go maps

- In Go language, a map is a powerful, ingenious, and versatile data structure.
- Golang 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.

Go maps

- In the maps, a key must be unique and always in the type which is comparable using == operator or the type which support != operator.
- So, most of the built-in type can be used as a key like an int, float64, string.
- The values are not unique like keys and can be of any type like int, float64, rune, string, pointer, etc

Go maps

- // An Empty map
- map [Key_Type] Value_Type { }

- // Map with key-value pair
- map[Key_Type] Value_Type {key1: value1, ..., keyN: valueN}

- Example:

- var mymap map [int] string

Go maps

```
import "fmt"
func main() {
        var map_1 map[ int] int
        if map_1 == nil {
           fmt.Println("True")
    } else {
           fmt.Println("False")
    }
        map_2 := map[int] string{
               10: “car",
               20: “aeroplan",
               30: “bus",
               93: “bike",
                  }
      fmt.Println("Map-2: ", map_2)
}
```

```
True
Map-2: map[10:car 20:aeroplan 30:bus 40:bike]
```

```
import "fmt"

func main() {

	mapp := map[int]string{1: "one", 2: "two", 3: "three"}


   for i, spell := range mapp {

		fmt.Println(i , " = ", spell)
   }
}
```

```
1:one
2:two
3:three
```

i will be the key and spell is the value

For loop for maps

```
package main

import "fmt"

func main() {

        mmap := map[int]string{
        20:“Faculty",
        30:“of ",
        40:“computing",
    }
    for key, value:= range mmap {
       fmt.Println(key, value)
    }

}
```

Output:
20 Faculty
30 of
40 computing

For loop for maps

```
m := map[string]int{
        "one":   1,
        "two":   2,
        "three": 3,
}
for k, v := range m {
        fmt.Println(k, v)
}
```

```
one:1
two:2
three:3
```

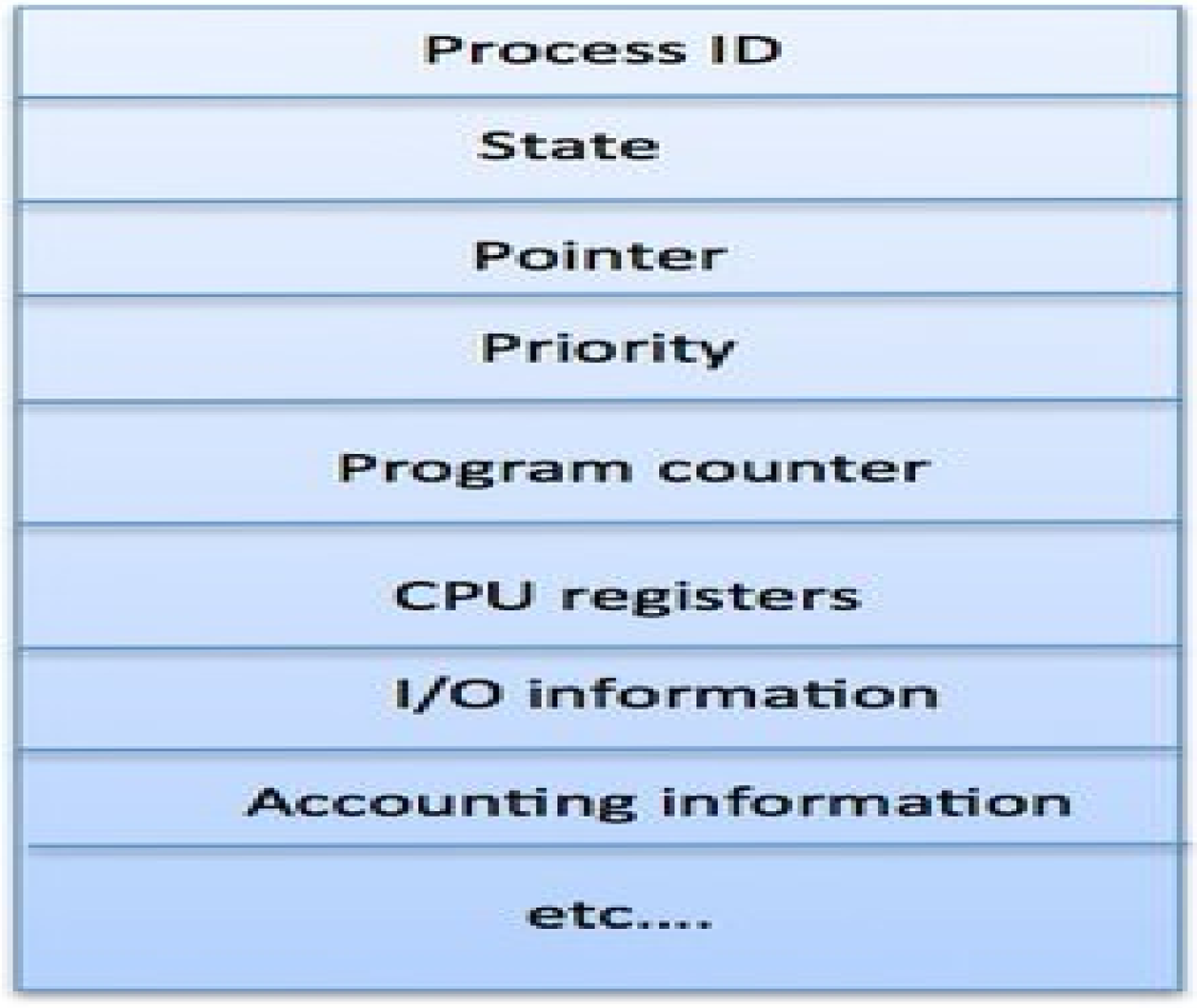
```
func main( )  {
    nums := [] int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
     for i, num := range nums {
        if num == 3 {
            fmt.Println(“ index:", i )
        }
    }
    kvs := map [string] string {"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    for k := range kvs {
        fmt.Println(k)
    }
}
```

```
sum: 9
index: 1
a -> apple
b -> banana
a
b
```

Concurrency in Go language

Process and thread

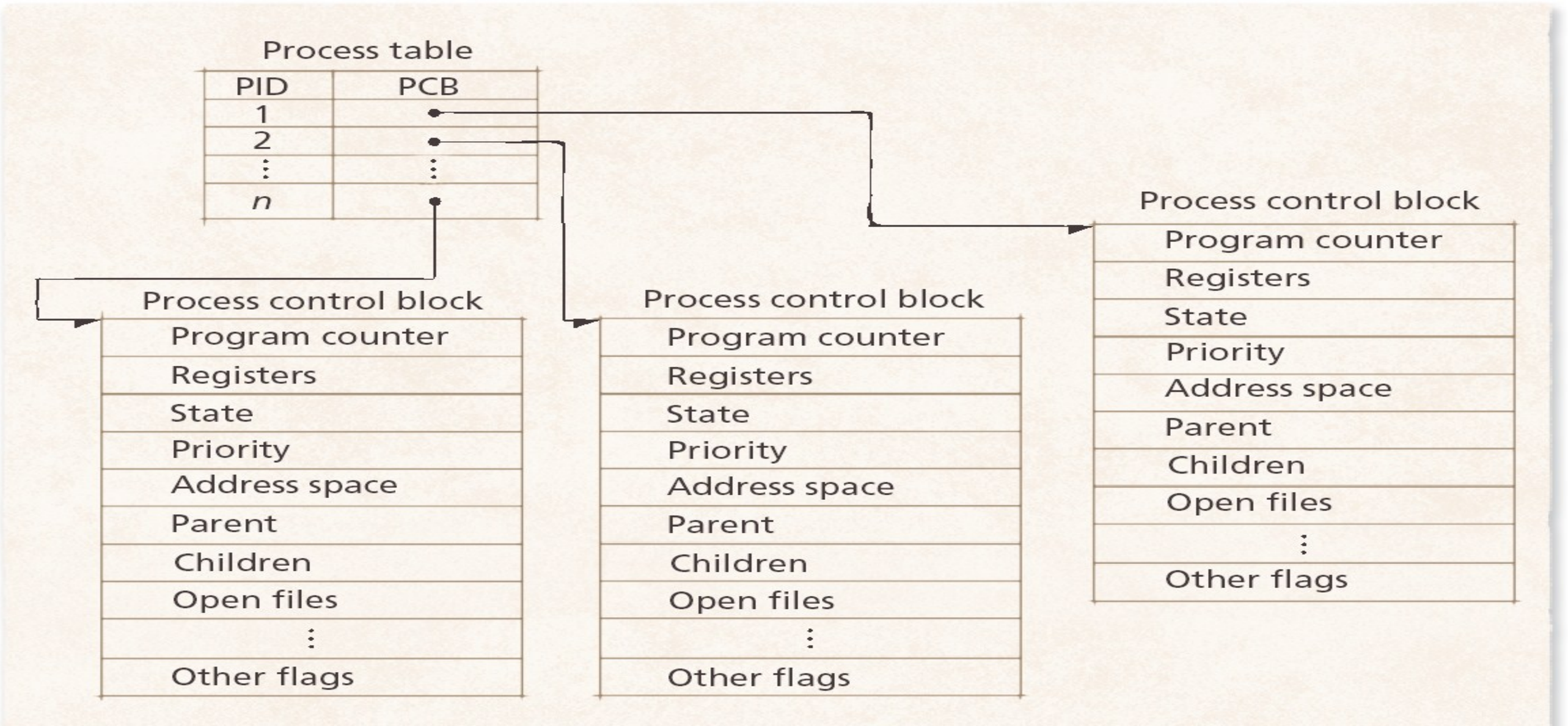
A process in a running task (program) managed by OS



Process Control Block

Process Control Blocks (PCBs)/Process Descriptors

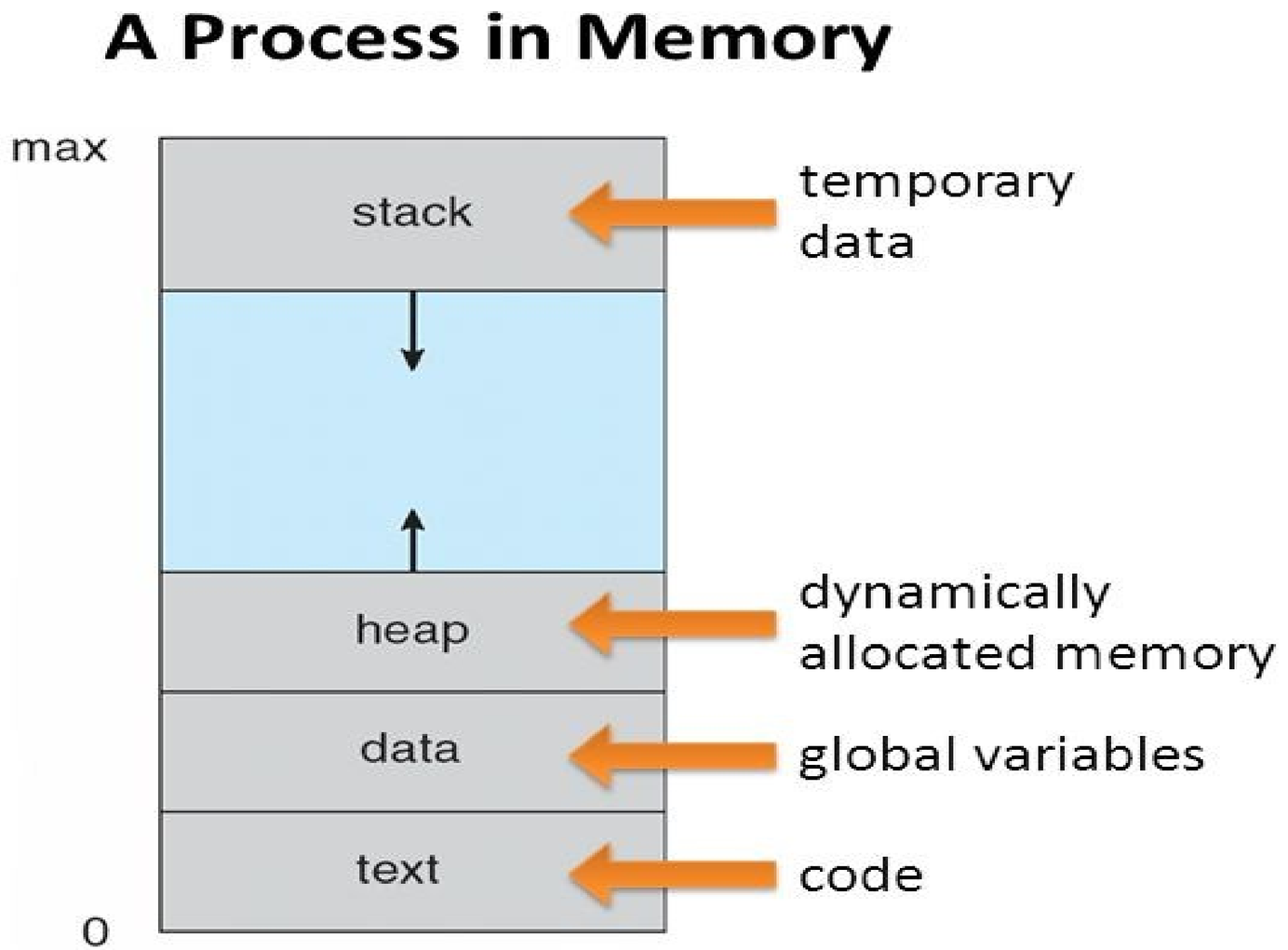
Process table and process control blocks.



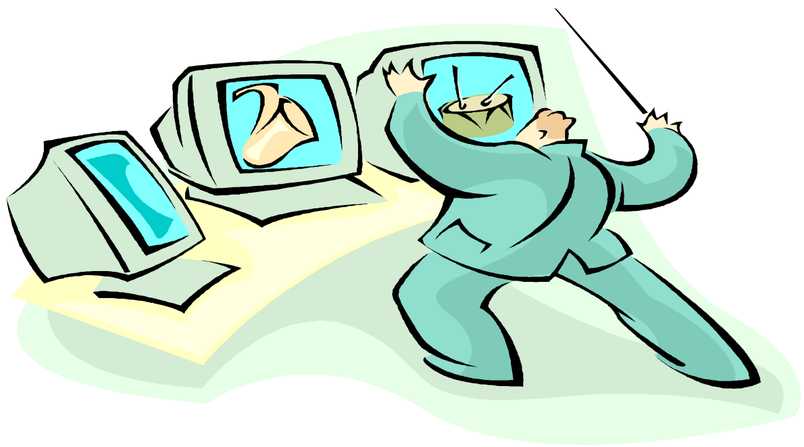
Process Concept

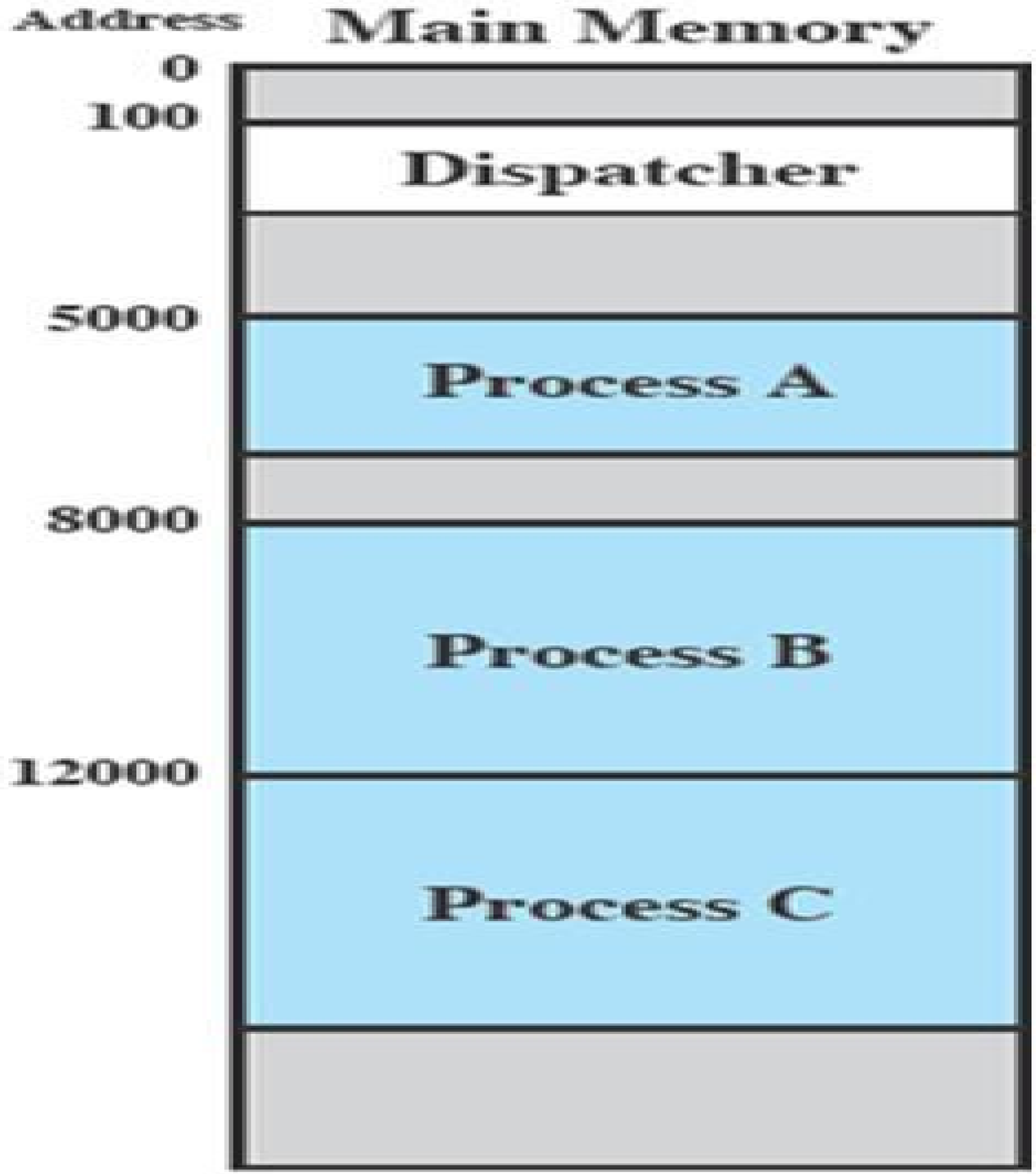
A **process** is *a program in execution.*

We use the terms *process*, *job*, and *task* interchangeably.



Process Execution





Traces of three Processes a, b, c

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Execute six instruction then time out
Dispatcher from 100- 105
Process B needs I/O in instruction 8003

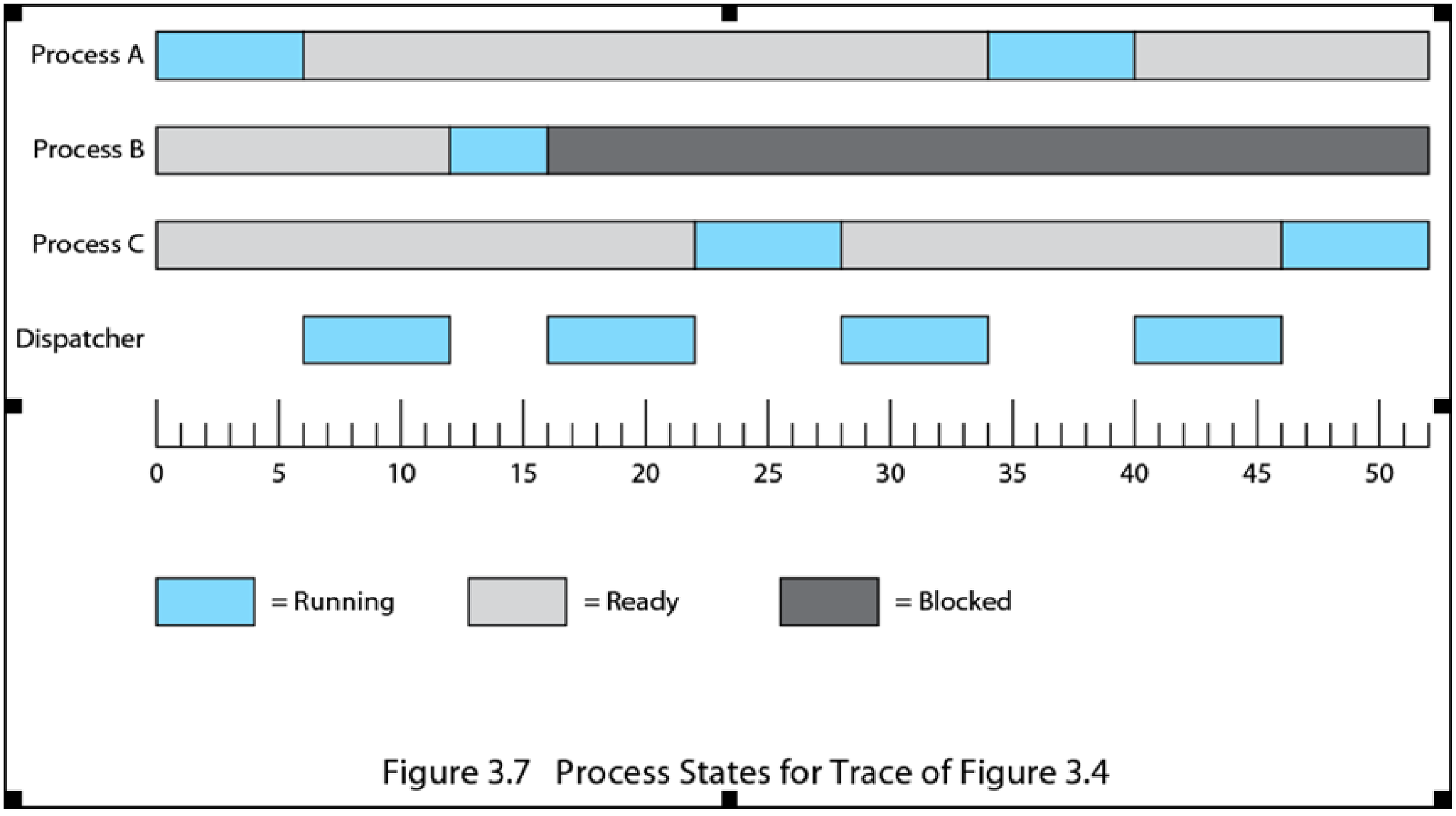
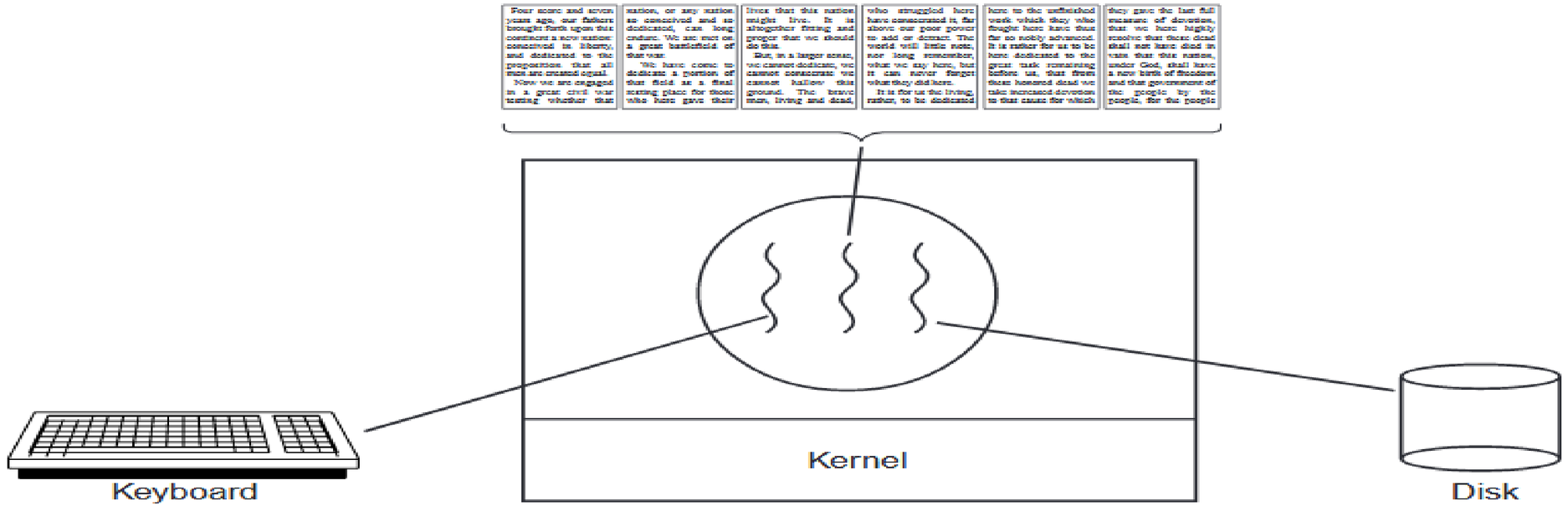


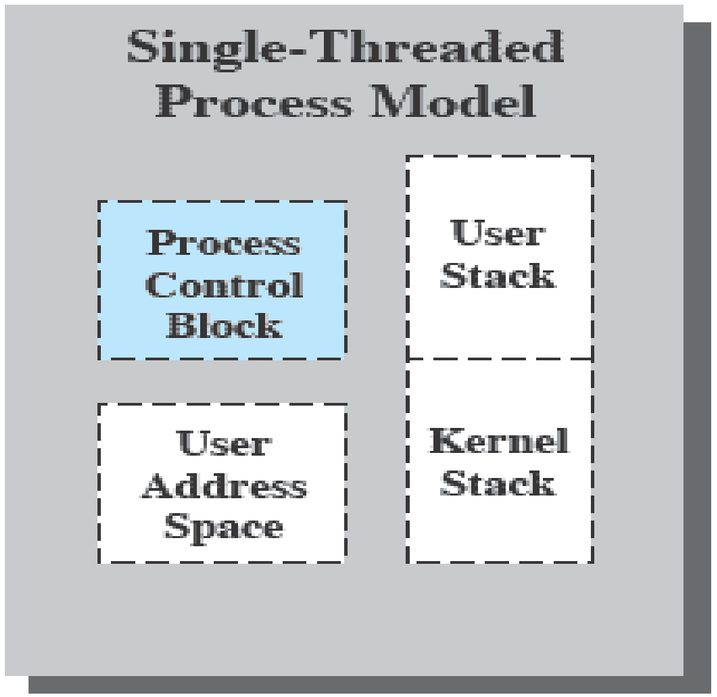
Figure 3.7 Process States for Trace of Figure 3.4

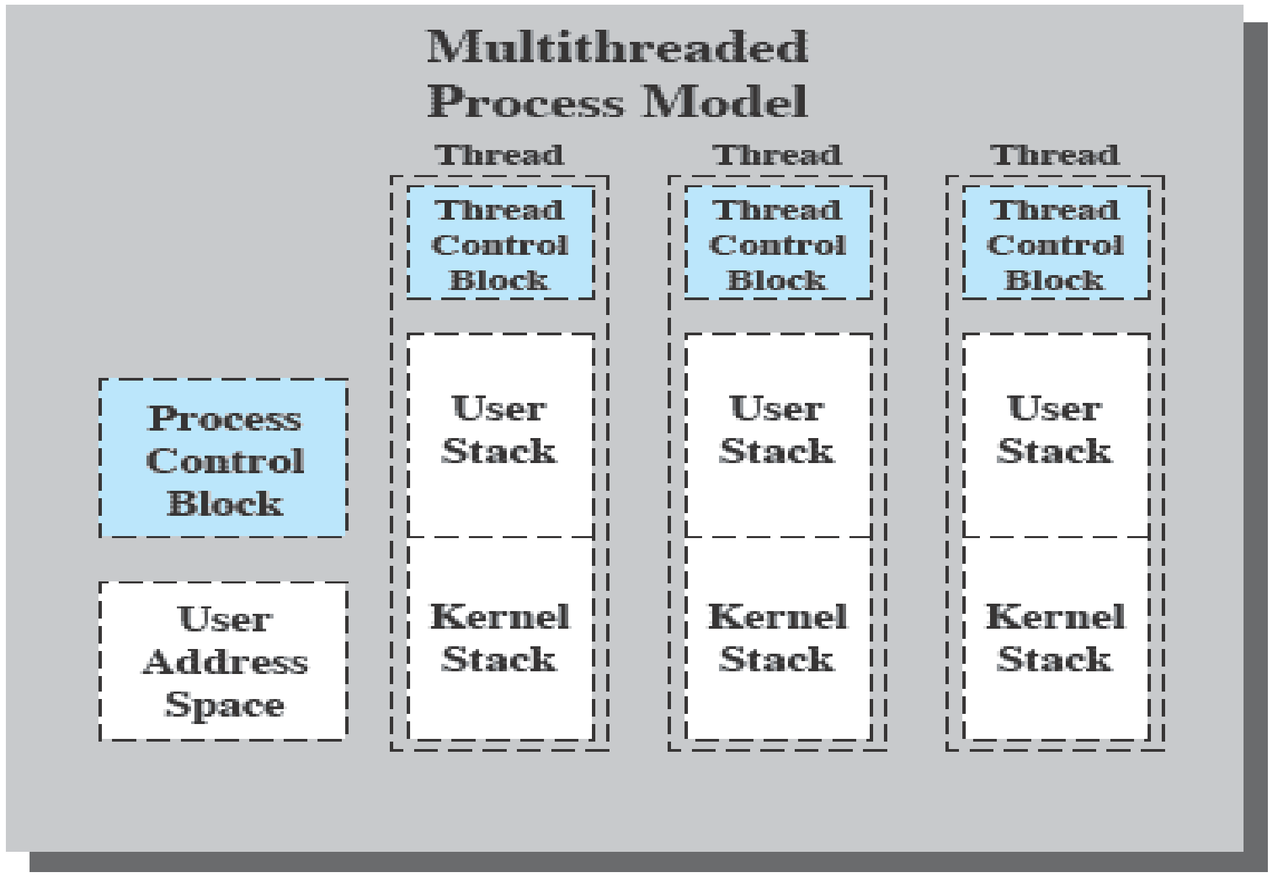
The main idea is to separate the functionality of an application (process) to smaller (threads) each thread is concerned with functional type or frequently require the same kind of resources



If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. → (low performance)

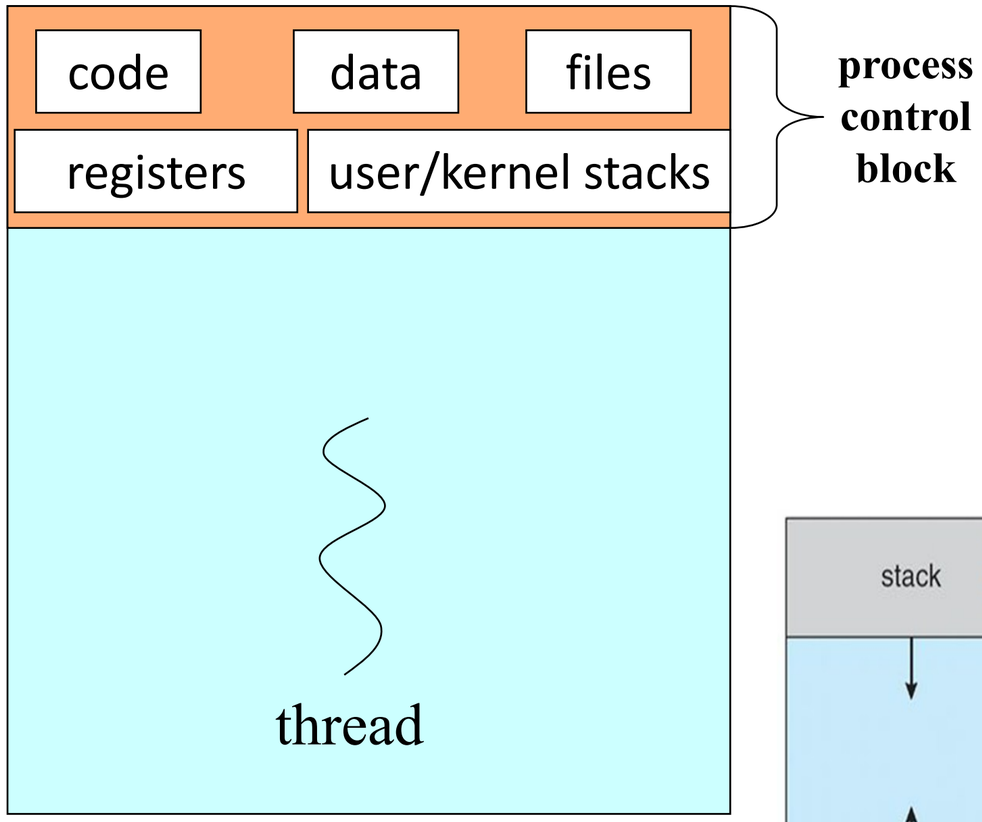
Threads vs. Processes



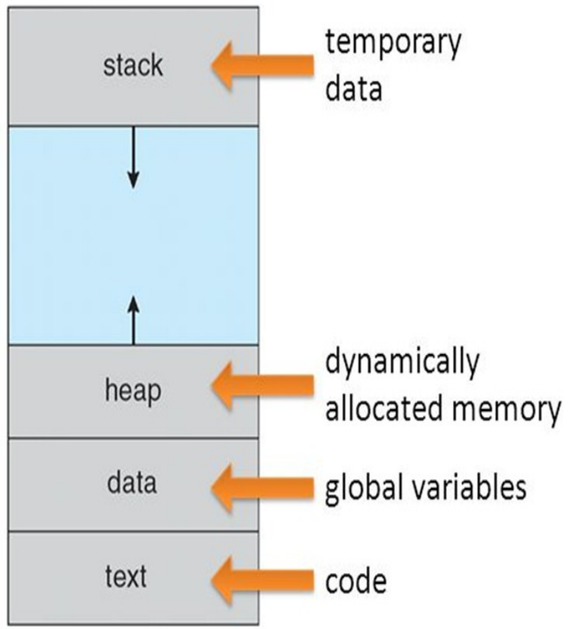


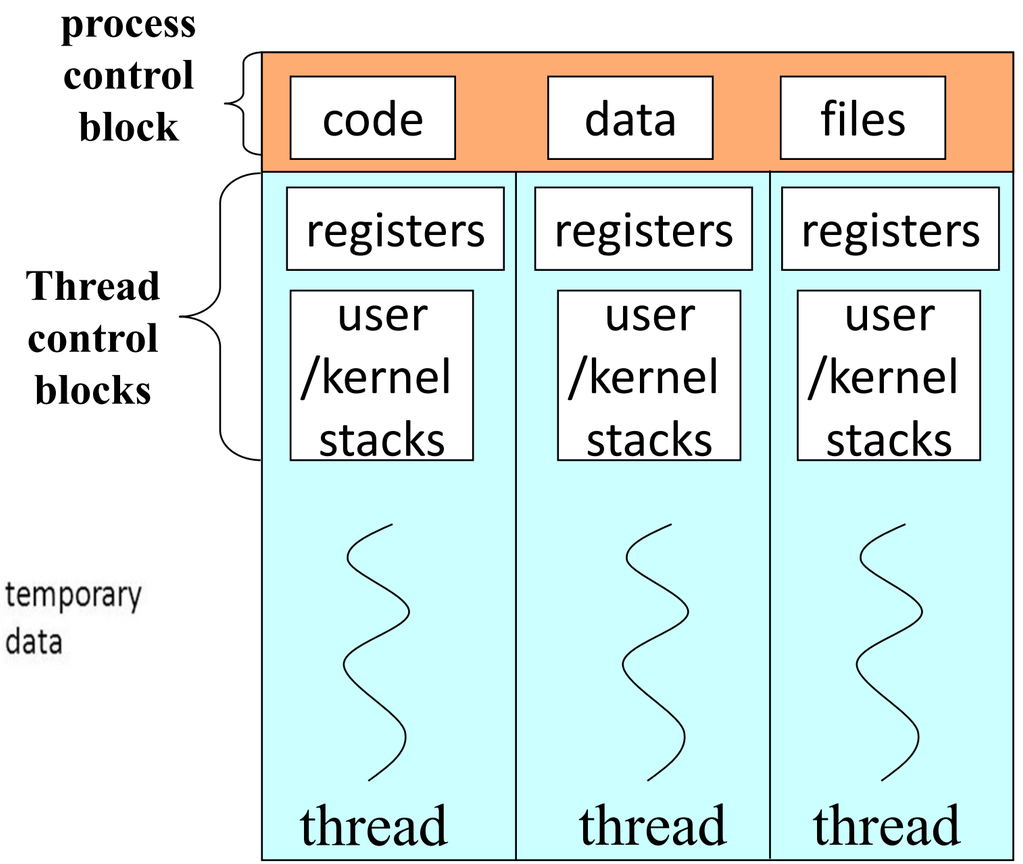
Fi

Single and multithreaded processes



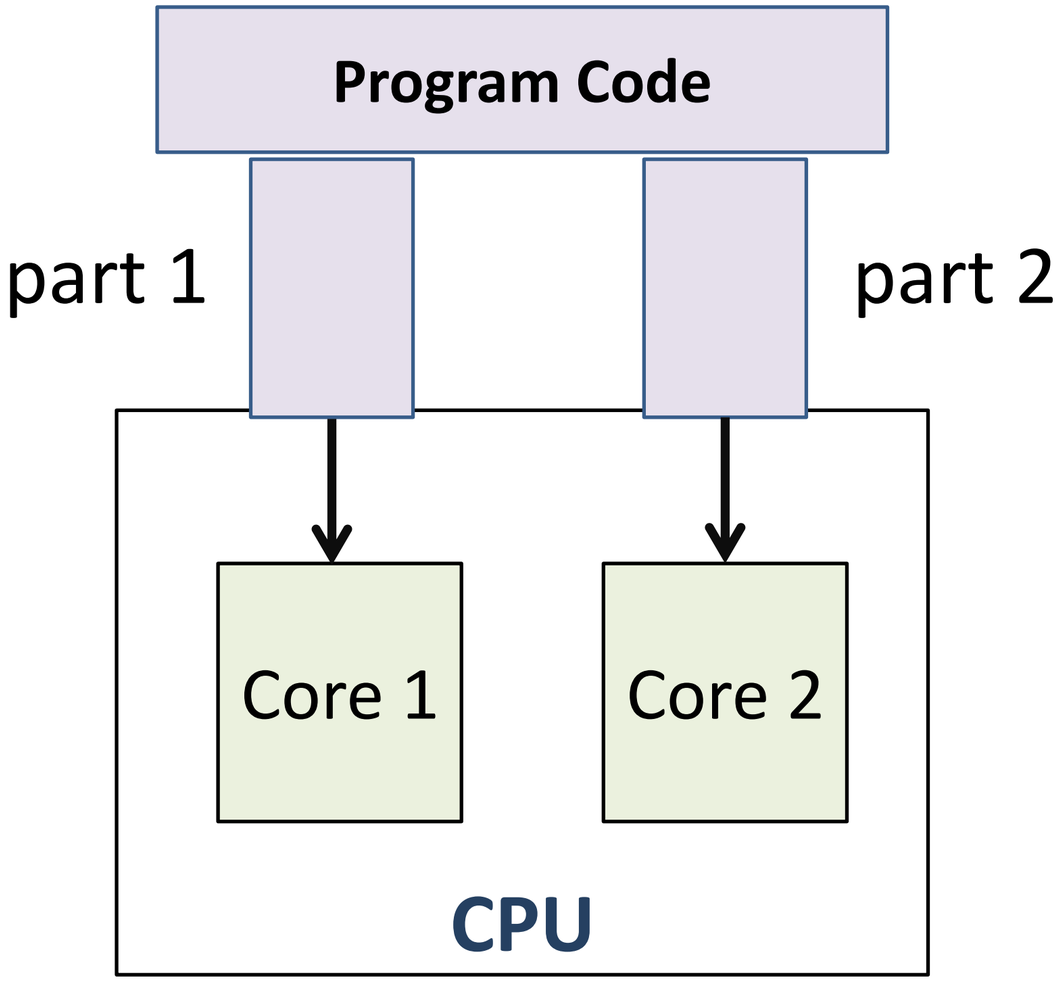
single threaded



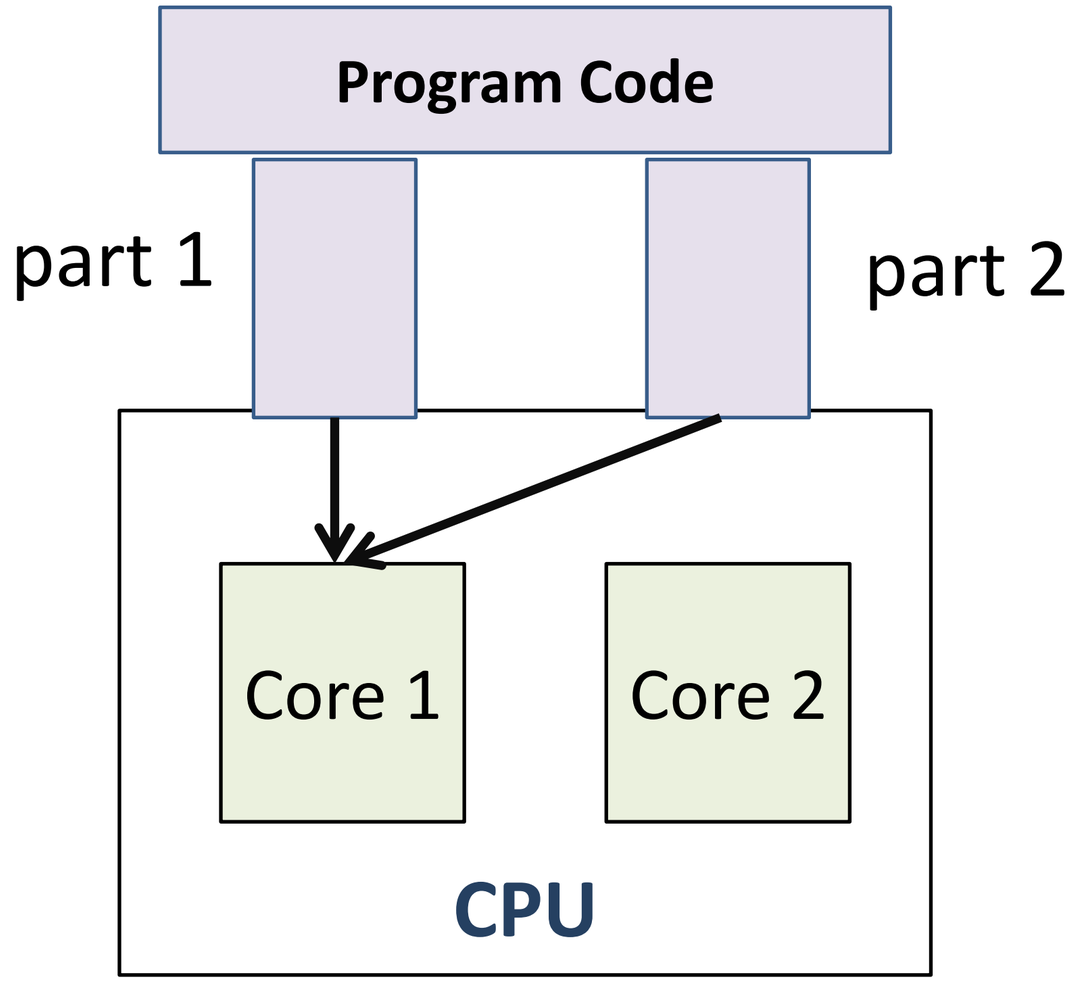


multithreaded

Concurrent programming Vs. parallel programming



Parallel programming



Concurrent programming

- Go language provides a special feature known as a Goroutines.
-
- A Goroutine is a function or method which executes independently and simultaneously in connection with any other Goroutines present in your program.

- Or in other words, every concurrently executing activity in Go language is known as a Goroutines.

- **Goroutines** are functions which executes **concurrently** along with the main program flow.

- Goroutine interacts with other goroutines using a **communication mechanism called channels** in Go.

- When a program starts, its only goroutine is the one that calls the main function, so we call it the *main goroutine*.
- New goroutines are created by the go statement:

Goroutines syntax

Syntax:

```
func name(){
// statements
}

// using go keyword as the
// prefix of your function call
go name( )
```

```
f() // call f(); wait for it to return
go f() // create a new goroutine that calls f(); don't wait
```

Implementing Go routine

```
package main
import "fmt"
func display(str string) {
    for w := 0; w < 3; w++ {
        fmt.Println(str)
    }
}
func main() {
        go display("Welcome")
      display(“FCDS")
}
```

```
FCDS
FCDS
FCDS
```

The control does not wait for Goroutine to complete their execution just like normal function.
Control always move forward to the next line after the Goroutine call and ignores the value returned by the Goroutine.

Implementing Go routine

```
package main
import (
    "fmt"
    "time"
)
func display(str string) {
    for w := 0; w < 3; w++ {
        time.Sleep(1 * time.Second)
        fmt.Println(str)
    }
}
  func main() {
          go display("Welcome")
       display(“FCDS")
}
```

```
Welcome
FCDS
Welcome
FCDS
Welcome
FCDS
```

```
import (
   "fmt"
   "time"
)
func foo (s string) {
   for i := 1; i <= 3; i++ {
      time.Sleep(100 * time.Millisecond)
      fmt.Println(s, ": ", i)
   }
}
func main() {
  fmt.Println("Main started...")
go foo("1st goroutine")
   go foo("2nd goroutine")
 time.Sleep(time.Second)
   fmt.Println("Main finished")
}
```

```
Output
Main started...")
1st goroutine : 1
 2nd goroutine : 1
2nd goroutine : 2
1st goroutine : 2
 2nd goroutine : 3
 1st goroutine : 3
 Main goroutine finished
```

Go Channels

- Channels provide a way to send messages from one goroutine to another.
- We can declare a new channel type by using the **chan** keyword along with a datatype:
- var c **chan** int
- Here, c is of type chan int – which means it's a channel through which int types are sent.
- The default value of a channel is nil, so we need to assign a value.

The channel is by default nuil use make to define it

```
package main

import "fmt"

func main() {
    var c chan int
    if c == nil {
        fmt.Println("channel c is nil, going to define it")
        c = make(chan int)
        fmt.Printf("Type of c is %T", c)
    }
}
```

Output:

channel c is nil, going to define it
Type of c is chan int

syntax to send and receive data from a channel

- a is the channel

- data := <- a // read from channel a
- a <- data   // write to channel a

- The direction of the arrow with respect to the channel specifies whether the data is sent or received.

- In the first line, the arrow points outwards from a and hence we are reading from channel a and storing the value to the variable data.

- In the second line, the arrow points towards a and hence we are writing to channel a.

```
package main
import "fmt"
func  sendValues (myIntChannel chan int) {
  for i:=0; i<5; i++ {
    myIntChannel <- i
  }
}
func main() {
  myIntChannel := make(chan int)
  go sendValues(myIntChannel) // function sending value
  for i:=0; i<5; i++ {
    fmt.Println(<-myIntChannel) //receiving value
  }
}
```

```
Output:
0
1
2
3
4
```

- In a goroutine, the function sendValues was sending values over myIntChannel by using a for-loop.

- On the other hand, myIntChannel was receiving values and the program was printing them onto the console.

- The most important point to note is that both the following statements were blocking operations myIntChannel <- i and <-myIntChannel.

- the program when blocked on myIntChannel <- i was unblocked by the <-myIntChannel statement.