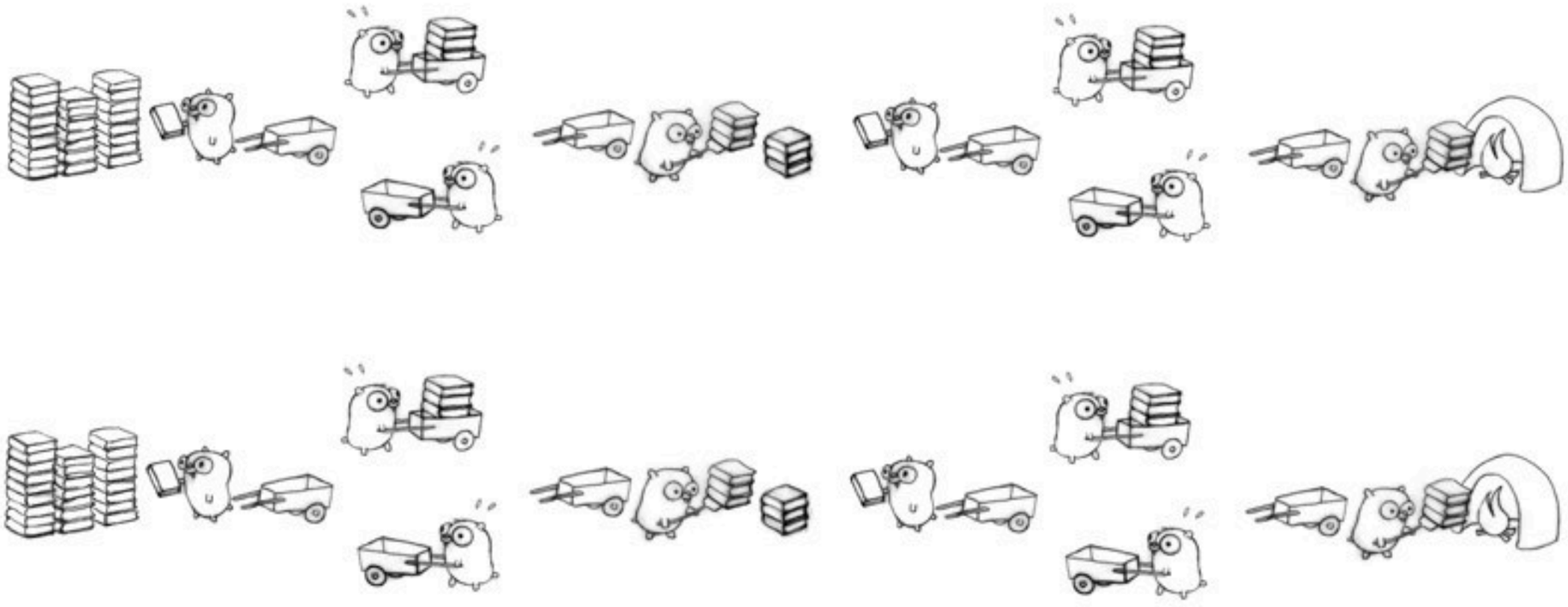


Go Routine
Concurrency and Parallelism

Jin Hyun Kim

Fall, 2019

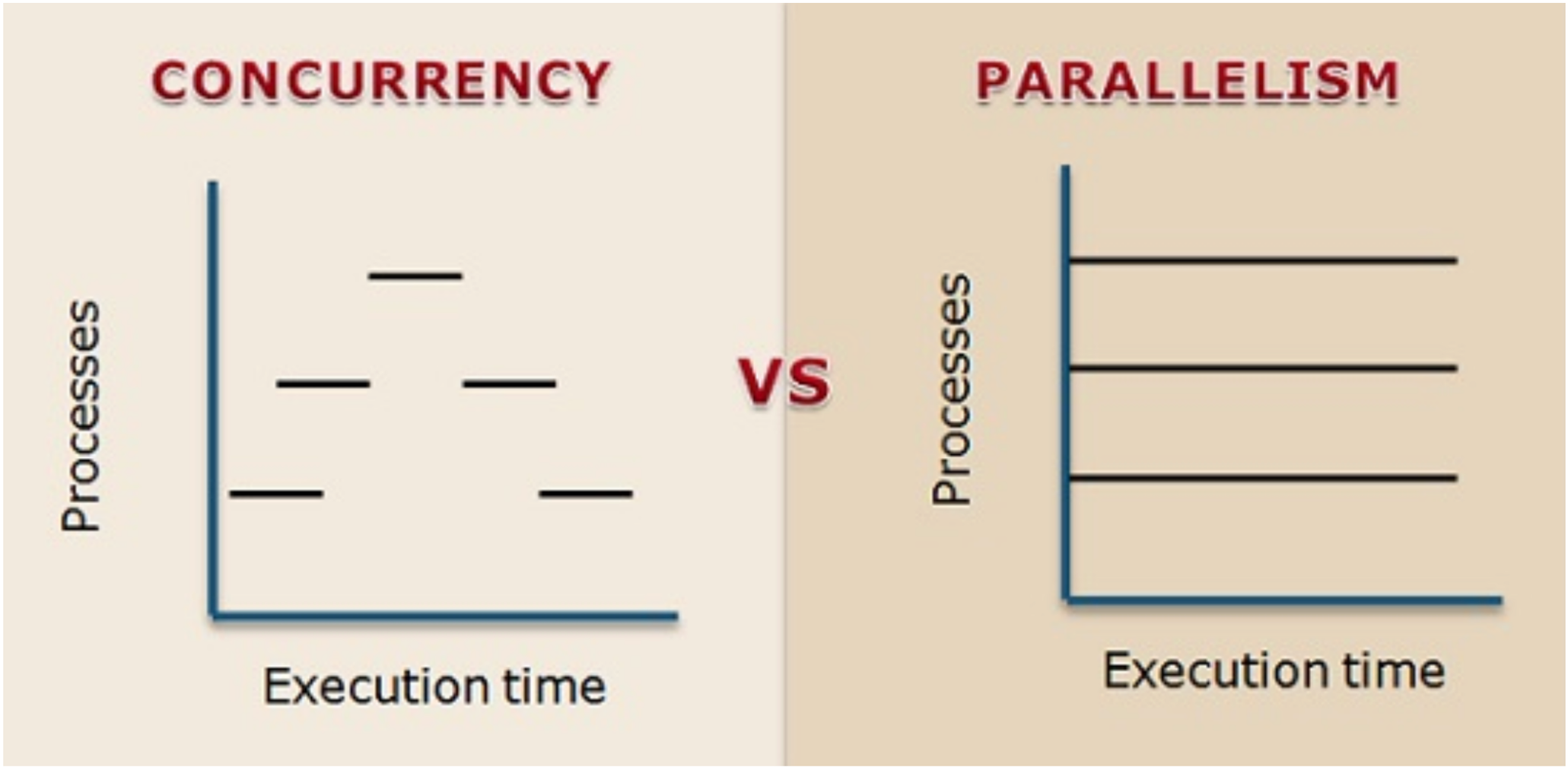
Today

- Concurrency and Parallelism
- Multi Processing
    - Process, Thread, Routine
- Golang routine and channel

Concurrency vs Parallelism

- Concurrency vs Parallelism (동시성과 병렬성)
- Parallelism 병렬성
    - 물리적인 시간 상에서 동시 (同时)에 함께 실행 Execute together at the same time
- Concurrency 동시성
    - 가능하다면 서로 독립적으로 실행할수 있도록 컴포넌트를 구성하여 Execute together using the same resources

Concurrency vs Parallelism



Concurrency vs Parallelism

BASIS FOR COMPARISON	CONCURRENCY	PARALLELISM
Basic	It is the act of managing and running multiple computations at the same time.	It is the act of running multiple computations simultaneously.
Achieved through	Interleaving Operation	Using multiple CPU's
Benefits	Increased amount of work accomplished at a time.	Improved throughput, computational speed-up
Make use of	Context switching	Multiple CPUs for operating multiple processes.
Processing units required	Probably single	Multiple
Example	Running multiple applications at the same time.	Running web crawler on a cluster.

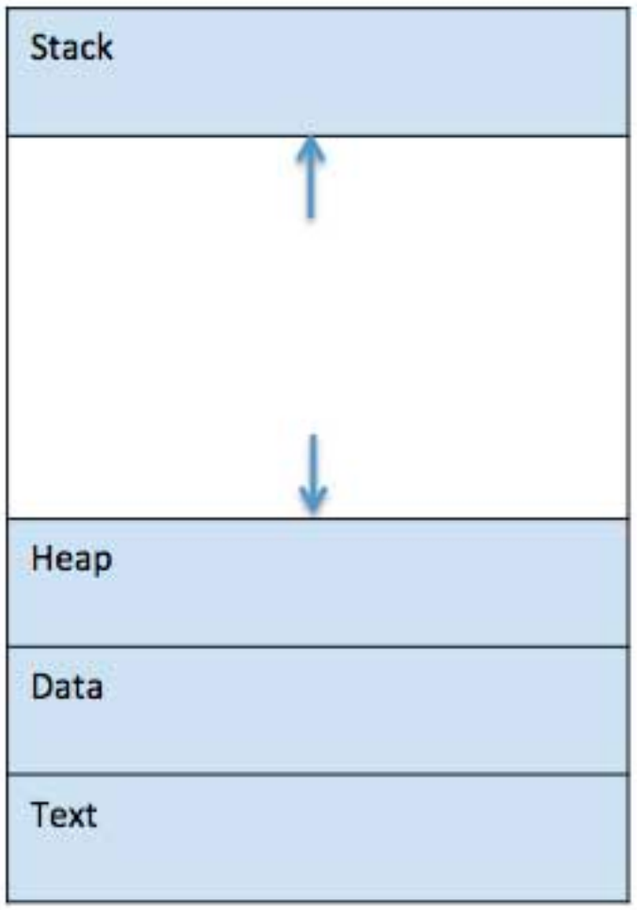
Concurrency 동시성

- Process 프로세스: CPU의 스케줄링 단위, 명령어, 사용자 데이터, 시스템 영역, 실행 과정에 수집한 다양한 종류의 리소스로 구성된 실행단위

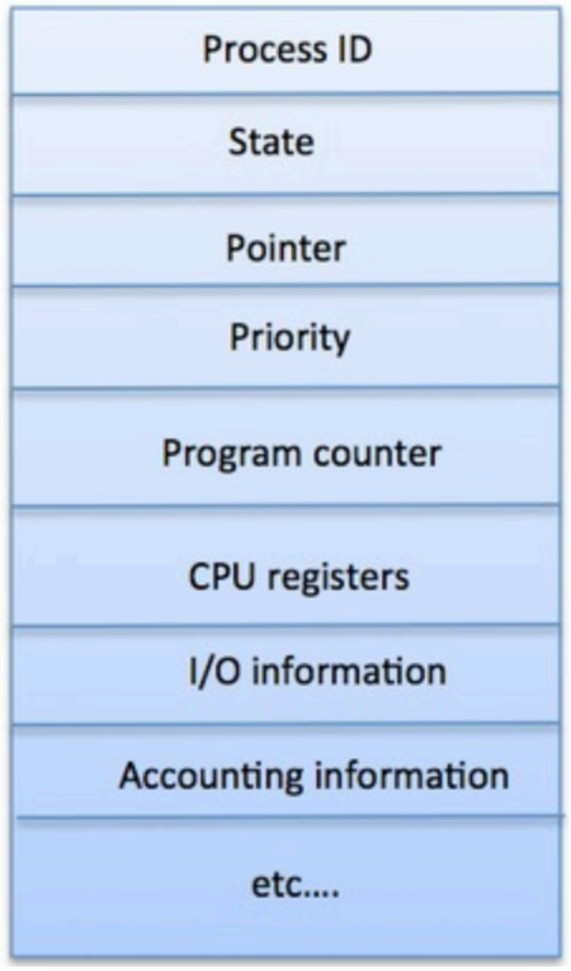
- 프로그램: 프로세스의 명령어와 데이터를 담은 파일

- Thread 쓰레드: 프로세스에 의한 보다 작은 실행단위, 독립적인 제어 흐름 및 **스택**을 갖는다.

Process

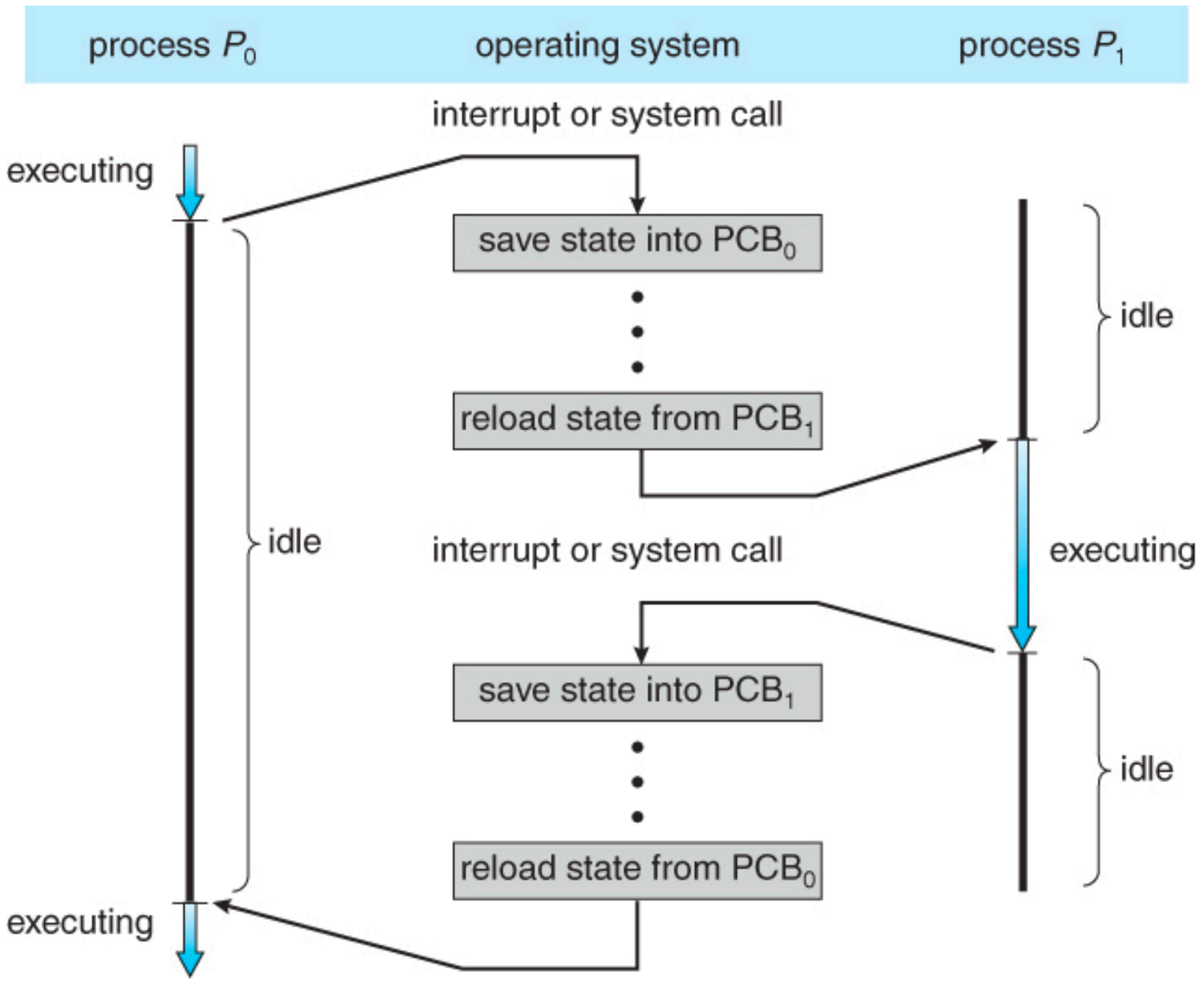


Program in Memory (Process)

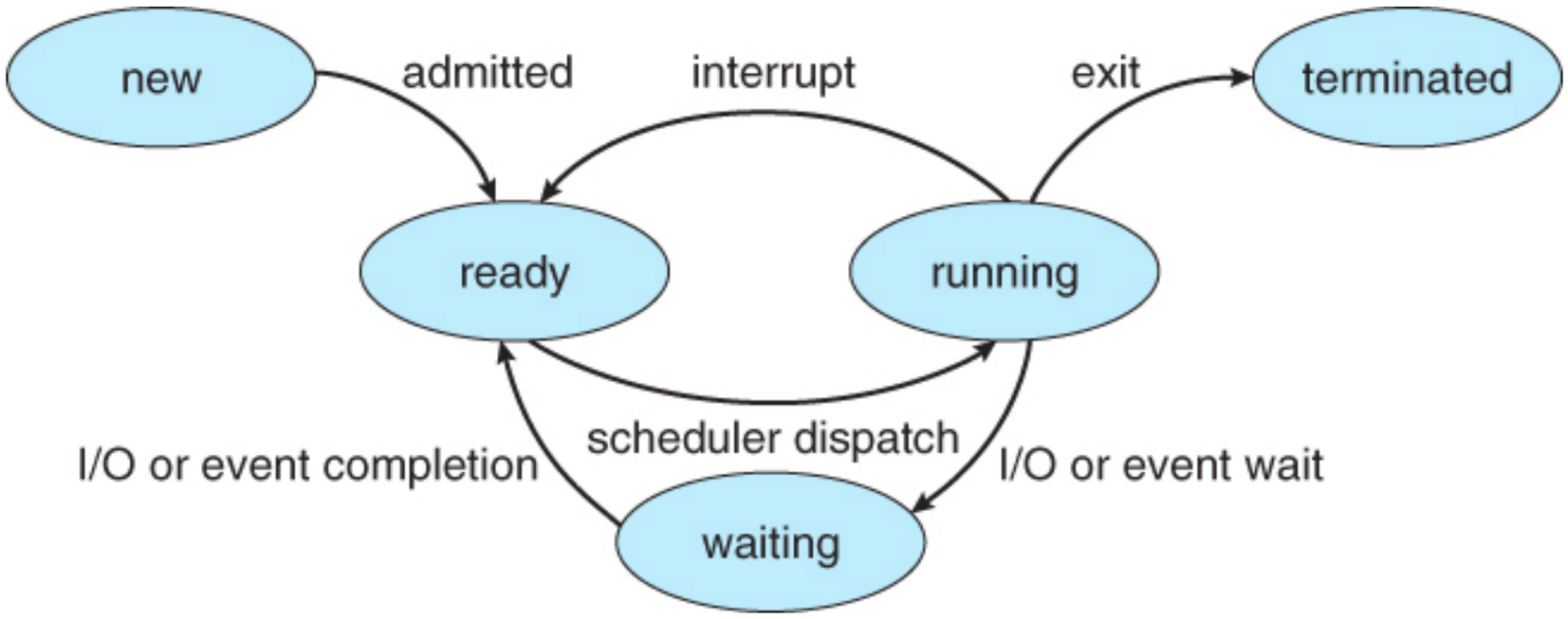


Program Control Block (PCB)

Process Context Switch
프로세스의 문맥전환

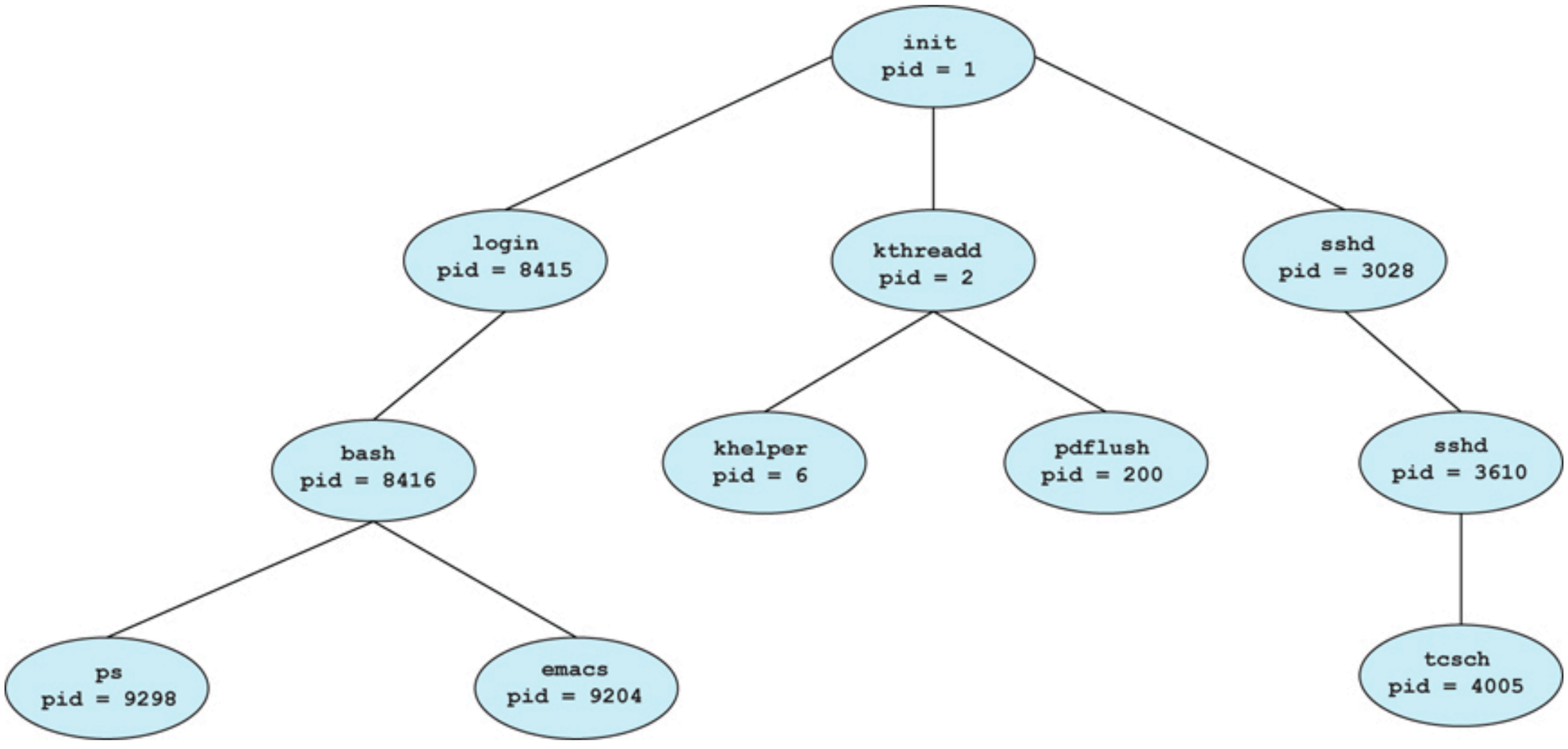


Process

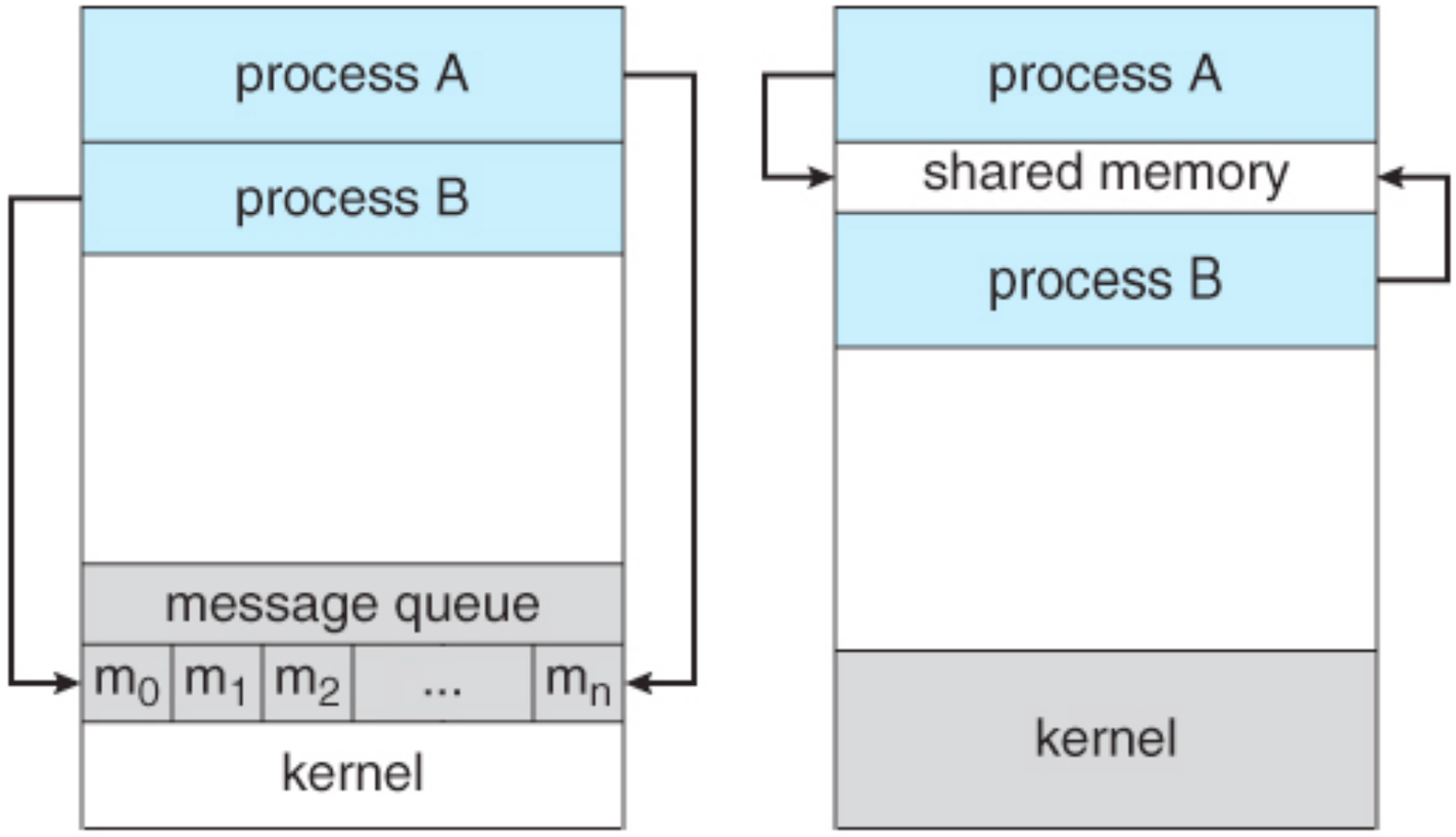


프로세스의 상태 변화

Proces in OS


init
pid = 1
login
pid = 8415
kthreadd
pid = 2
sshd
pid = 3028
bash
pid = 8416
khelper
pid = 6
pdflush
pid = 200
sshd
pid = 3610
ps
pid = 9298
emacs
pid = 9204
tcsch
pid = 4005

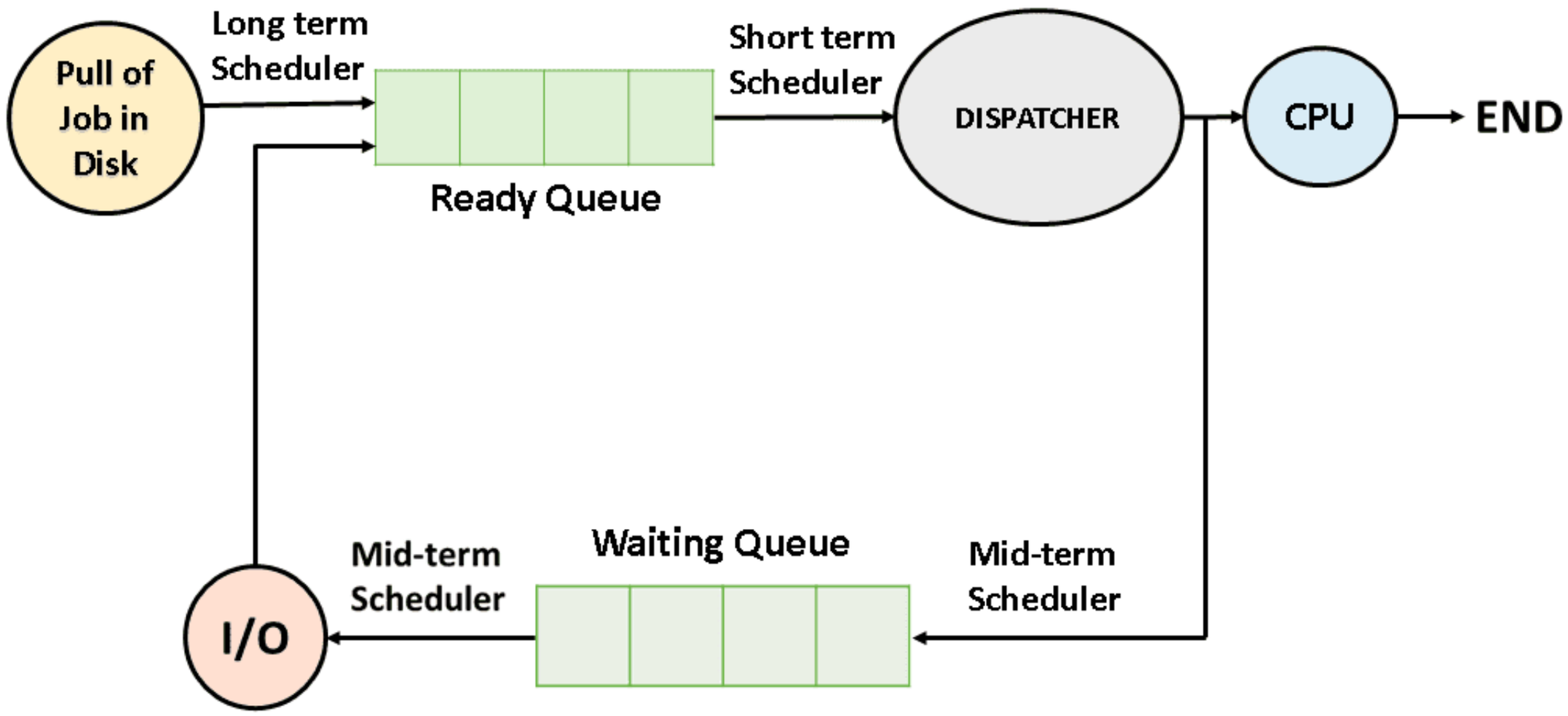
Inter-Proces Comm. (IPC)



(a) Message Passing

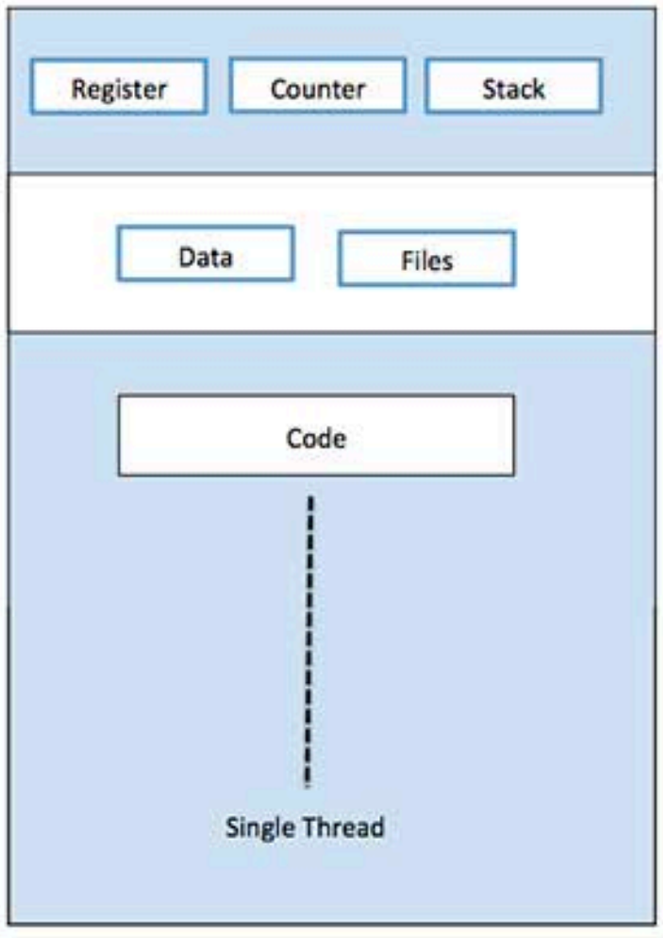
(b) Shared Memory

Process Scheduling

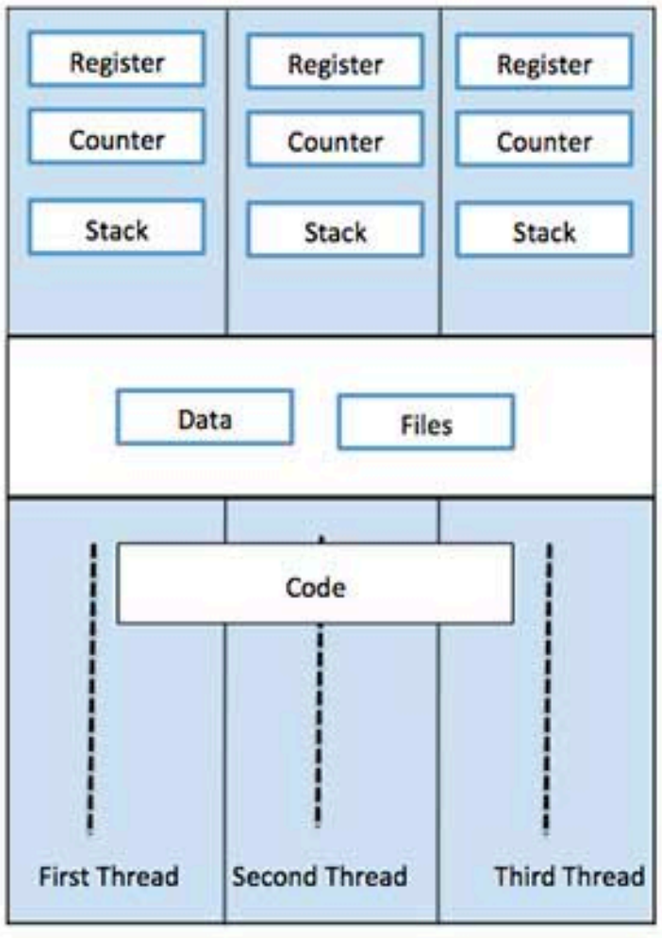


Thread

- A basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)



Single Process P with single thread



Single Process P with three threads

Thread

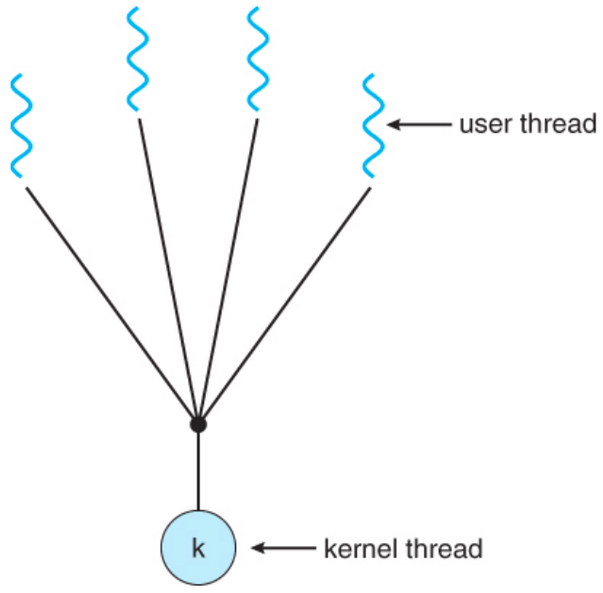
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

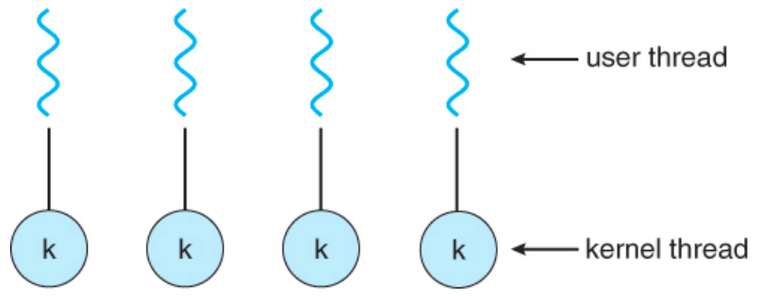
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork off separate processes for every incoming request.

Thread Execution
쓰레드의 실행

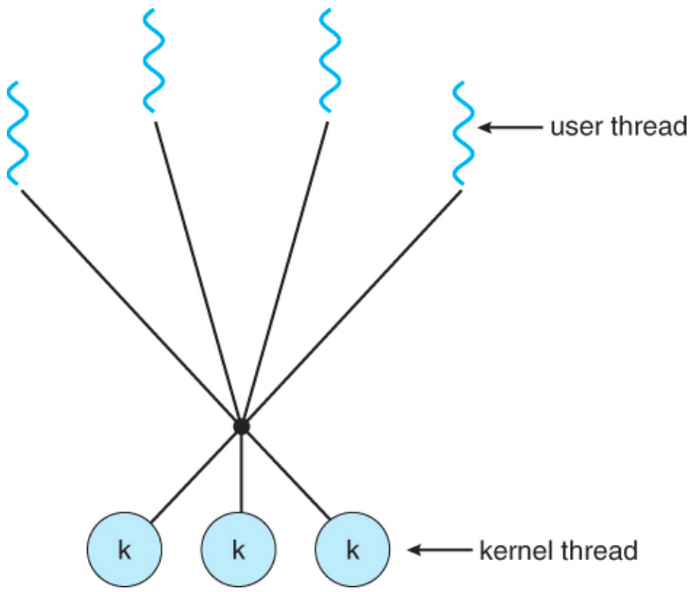
- User Level Threads – User managed threads.

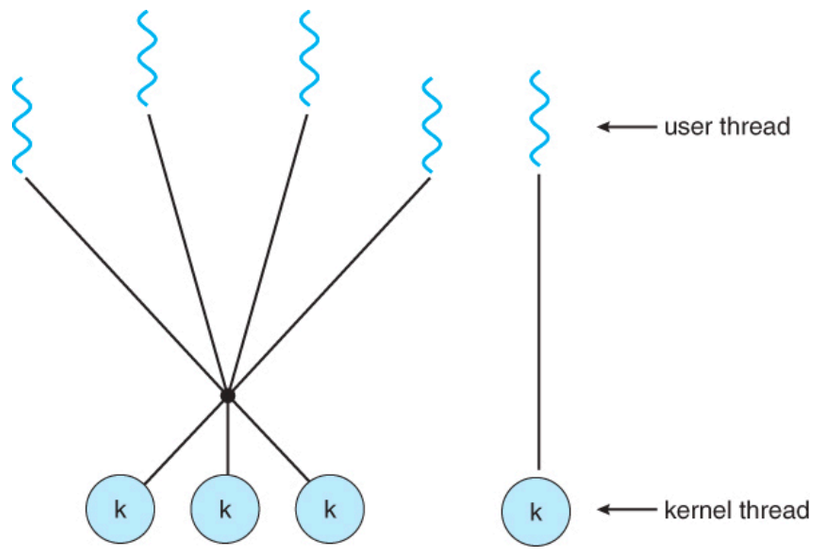
- Kernel Level Threads – Operating System managed threads acting on kernel, an operating system core.





Thread Execution
쓰레드의 실행





Process vs Thread

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Routine in Go (Goroutine)

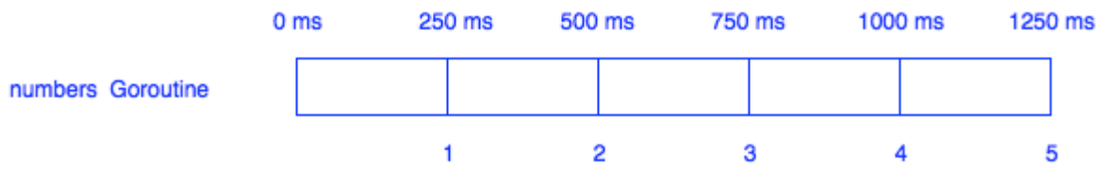
- Goroutines are functions or methods that run concurrently with other functions or methods.
- Can be thought of as light weight threads.
- The cost of creating a Goroutine is tiny when compared to a thread.

Goroutine Example

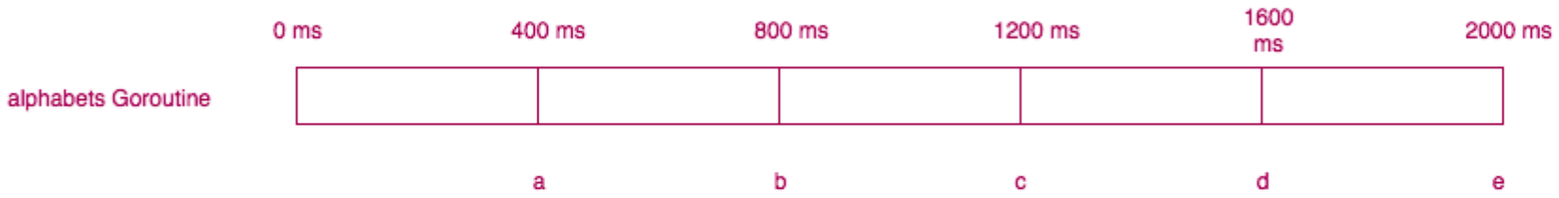
```go
package main

import (
    "fmt"
    "time"
)

func numbers() {
    for i := 1; i <= 5; i++ {
        time.Sleep(250 * time.Millisecond)
        fmt.Printf("%d ", i)
    }
}
func alphabets() {
    for i := 'a'; i <= 'e'; i++ {
        time.Sleep(400 * time.Millisecond)
        fmt.Printf("%c ", i)
    }
}
func main() {
    go numbers()
    go alphabets()
    time.Sleep(3000 * time.Millisecond)
    fmt.Println("main terminated")
}
```

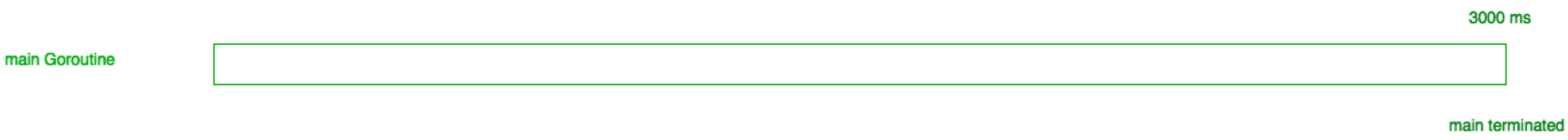
```
1 a 2 3 b 4 c 5 d e main terminated

Program exited.
```

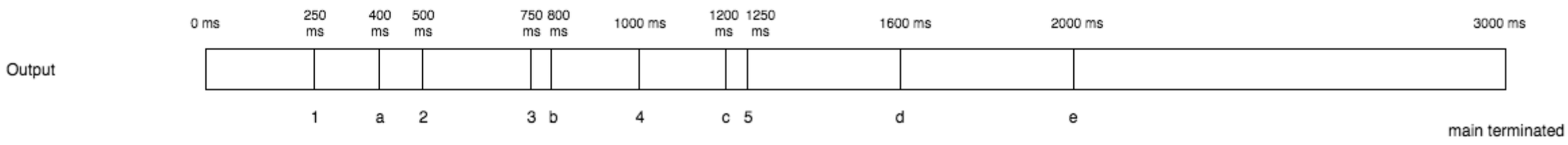
Goroutine Example




```
1 a 2 3 b 4 c 5 d e main terminated

Program exited.
```







Experiments

```go
package main

import (
    "fmt"
    "time"
)

func say(s string) {
    for i := 0; i < 10; i++ {
        fmt.Println(s, "***", i)
    }
}

func main() {
    // 함수를 동기적으로 실행
    say("Sync")

    // 함수를 비동기적으로 실행
    go say("Async1")
    go say("Async2")
    go say("Async3")

    // 3초 대기
    time.Sleep(time.Second * 3)
}
```

익명함수

```go
package main

func main() {
    sum := func(n ...int) int { //익명함수 정의
        s := 0
        for _, i := range n {
            s += i
        }
        return s
    }

    result := sum(1, 2, 3, 4, 5) //익명함수 호출
    println(result)
}
```

- 함수명을 갖지 않는 함수를 익명함수(Anonymous Function)

일급함수

- Go 프로그래밍 언어에서 함수는 일급함수로서 Go의 기본 타입과 동일하게 취급
  - 따라서 다른 함수의 파라미터로 전달하거나 다른 함수의 리턴값으로도 사용될 수 있음
  - 즉, 함수의 입력 파라미터나 리턴 파라미터로서 함수 자체가 사용될 수 있음

일급함수

```go
package main

func main() {
    //변수 add 에 익명함수 할당
    add := func(i int, j int) int {
        return i + j
    }

    // add 함수 전달
    r1 := calc(add, 10, 20)
    println(r1)

    // 직접 첫번째 파라미터에 익명함수를 정의함
    r2 := calc(func(x int, y int) int { return x - y }, 10, 20)
    println(r2)

}

func calc(f func(int, int) int, a int, b int) int {
    result := f(a, b)
    return result
}
```

```go
// 원형 정의
type calculator func(int, int) int

// calculator 원형 사용
func calc(f calculator, a int, b int) int {
    result := f(a, b)
    return result
}
```

Experiments

- Anonymous 익명함수 이용한 루틴

```go
package main

import (
    "fmt"
    "time"
)

func function() {
    for i := 0; i < 10; i++ {
        fmt.Print(i)
    }
    fmt.Println()
}

func main() {
    go function()

    go func() {
        for i := 10; i < 20; i++ {
            fmt.Print(i, " ")
        }
    }()

    time.Sleep(1 * time.Second)
}

```

Wait with sync.WaitGroup

```go
package main

import (
    "flag"
    "fmt"
    "sync"
)

func main() {
    n := flag.Int( name: "n",  value: 20,  usage: "Number of goroutines")
    flag.Parse()
    count := *n
    fmt.Printf( format: "Going to create %d goroutines.\n", count)

    var waitGroup sync.WaitGroup

    fmt.Printf( format: "%#v\n", waitGroup)
    for i := 0; i < count; i++ {
        waitGroup.Add( delta: 1)
        go func(x int) {
            defer waitGroup.Done()
            fmt.Printf( format: "%d ", x)
        }(i)
    }

    fmt.Printf( format: "%#v\n", waitGroup)

    waitGroup.Wait()

    fmt.Println( a...: "\nExiting...")
}
```

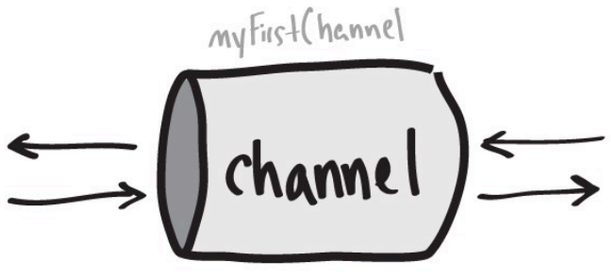
```go
var waitGroup sync.WaitGroup
```

- When launching a goroutine, waitGroup.Add(1)
- Program ends with waitGroupWait()

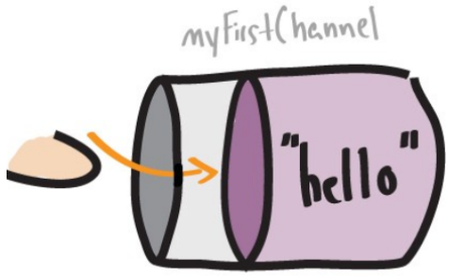
Channel

- Channels allow go routines to communicate with each other.

- A channel is like a pipe, from which go routines can send and receive information from other go routines.




```
myFirstChannel := make(chan string)
```




```
myFirstChannel <- "hello" // Send
myVariable := <- myFirstChannel // Receive
```

Channel

```go
package main

import (
    "fmt"
    "math/rand"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func main() {
    // create new channel of type int
    ch := make(chan int)

    // start new anonymous goroutine
    go func() {
        snd := random( min: 10, max: 20)
        fmt.Println( a...: "sending", snd)
        // send 42 to channel
        ch <- snd
    }()
    // read from channel
    rcv := <-ch
    fmt.Println( a...: "receiving",rcv)
}
```

https://divan.dev/posts/go_concurrency_visualize/

Great Work!

- GopherCon 2016: Ivan Danyliuk - Visualizing Concurrency in Go

- https://www.youtube.com/watch?v=KyuFeiG3Y60

- https://divan.dev/posts/go_concurrency_visualize/

Channel

```go
package main

import "time"

func timer(d time.Duration) <-chan int {
    c := make(chan int)
    go func() {
        time.Sleep(d)
        c <- 1
    }()
    return c
}

func main() {
    for i := 0; i < 24; i++ {
        c := timer(1 * time.Second)
        <-c
    }
}
```

Channel
Ping-Pong 2 Players

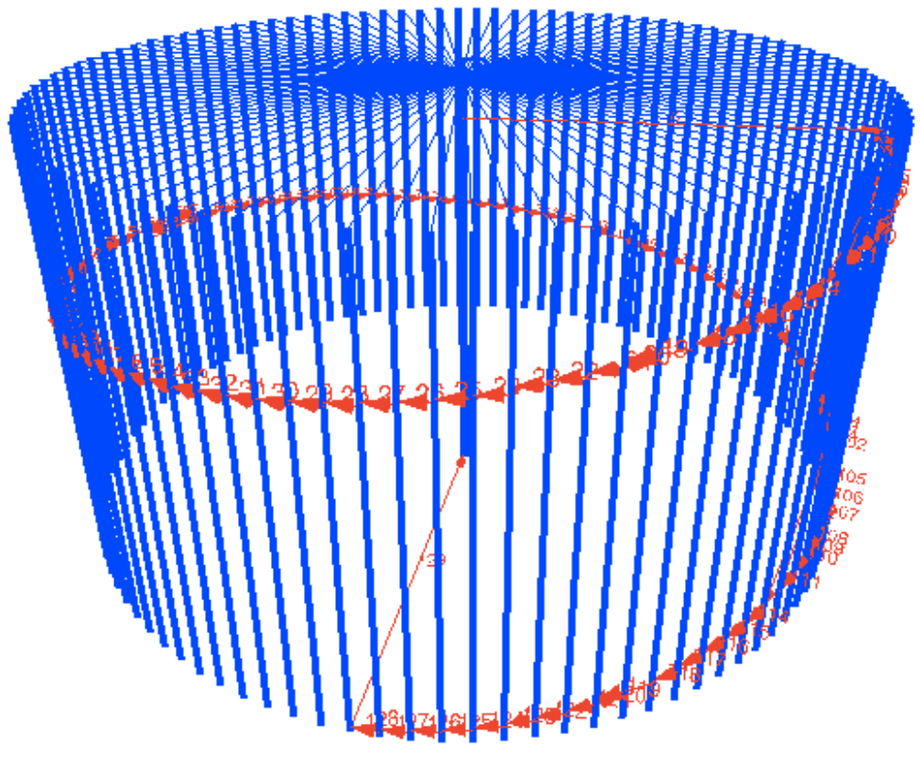
```go
package main

import "time"

func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Channel
Ping-Pong 3 Players

```go
package main

import "time"

func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

Channel
Ping-Pong 100 Players

```go
package main

import "time"

func main() {
    var Ball int
    table := make(chan int)
    go player(table)
    go player(table)

    table <- Ball
    time.Sleep(1 * time.Second)
    <-table
}

func player(table chan int) {
    for {
        ball := <-table
        ball++
        time.Sleep(100 * time.Millisecond)
        table <- ball
    }
}
```

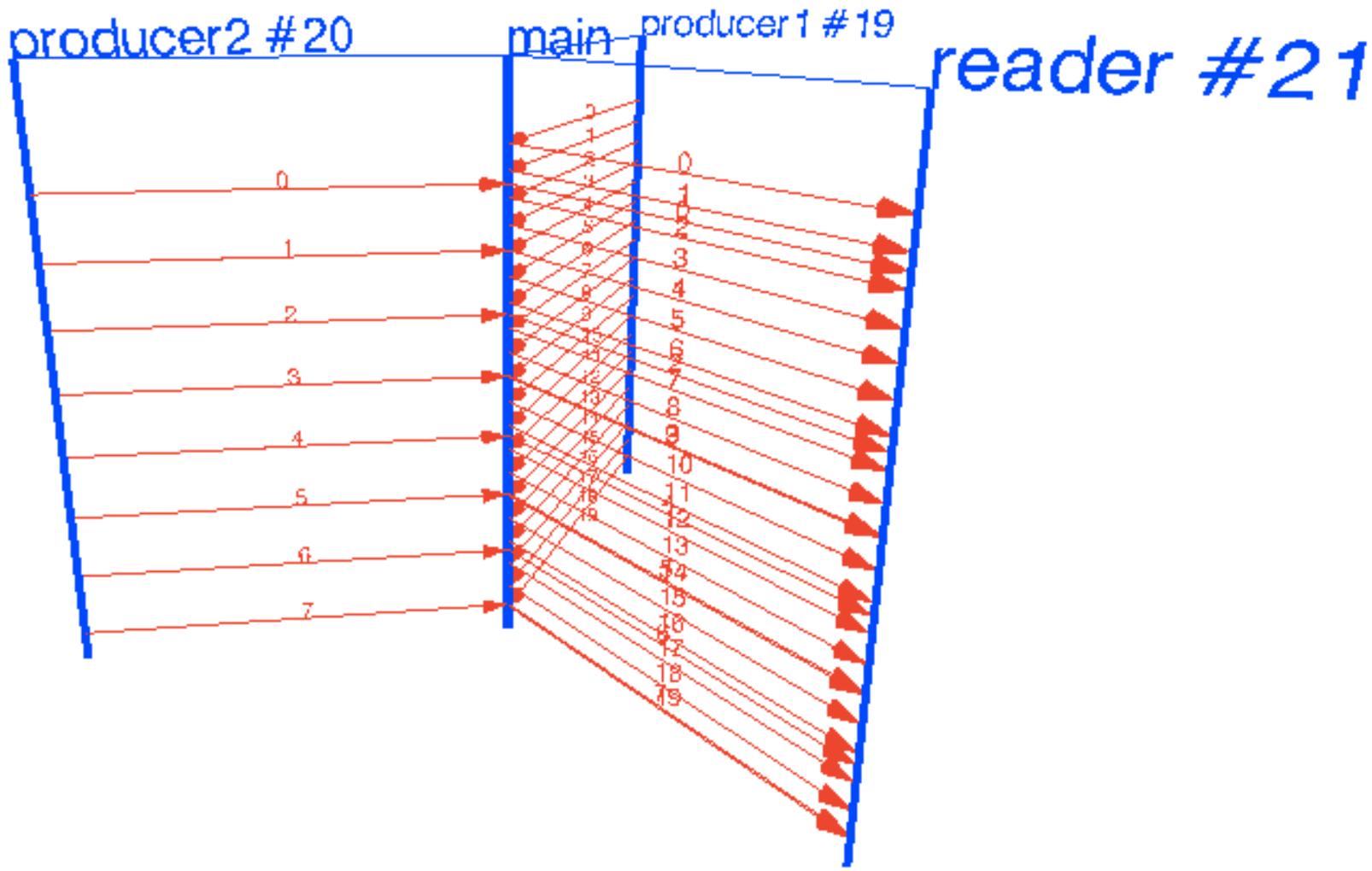


Channel: Producer-Consumer

```go
package main

import (
    "fmt"
    "time"
)

func producer(ch chan int, d time.Duration) {
    var i int
    for {
        ch <- i
        i++
        time.Sleep(d)
    }
}

func reader(out chan int) {
    for x := range out {
        fmt.Println(x)
    }
}
```

```go
func main() {
    ch := make(chan int)
    out := make(chan int)
    go producer(ch, 100*time.Millisecond)
    go producer(ch, 250*time.Millisecond)
    go reader(out)
    for i := range ch {
        out <- i
    }
}
```

Channel: Producer-Consumer



https://divan.dev/posts/go_concurrency_visualize/

In next class

- More about channel

- Go Scheduler