PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

Guía para el laboratorio de INF239 Sistemas Operativos

Tema: Concurrencia en Golang

1. Definiciones importantes

A continuación se presenta dos conceptos importantes que suelen confundirse mucho: concurrencia y paralelismo.

"Traditionally, the word ***parallel*** is used for systems in which the executions of several programs overlap in time by running them on separate processors. The word ***concurrent*** is reserved for potential parallelism, in which the executions may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of a small number of processors, often only one. Concurrency is an extremely useful ***abstraction*** because we can better understand such a program by pretending that all processes are being executed in parallel. Conversely, even if the processes of a concurrent program are actually executed in parallel on several processors, understanding its behavior is greatly facilitated if we impose an order on the instructions that is compatible with shared execution on a single processor. Like any abstraction, concurrent programming is important because the behavior of a wide range of real systems can be modeled and studied without unnecessary detail."

Principles of Concurrent and Distributed Programming, Second Edition by M. Ben-Ari

"We define a **concurrent** program as one in which multiple streams of instructions are active at the same time. One or more of the streams is available to make progress in a single unit of time. The key to differentiating parallelism from concurrency is the fact that through time-slicing or multitasking one can give the illusion of simultaneous execution when in fact only one stream makes progress at any given time.

In systems where we have multiple processing units that can perform operations at the exact same time, we are able to have instruction streams that execute in parallel. The term parallel refers to the fact that each stream not only has an abstract timeline that executes concurrently with others, but these timelines are in reality occurring simultaneously instead of as an illusion of simultaneous execution based on interleaving within a single timeline. A concurrent system only ensures that two sequences of operations may appear to happen at the same time when examined in a coarse time scale (such as a user watching the execution) even though they may do so only as an illusion through rapid interleaving at the hardware level, while a parallel system may actually execute them at the same time in reality.

Our definition of a **parallel** program is an instance of a concurrent program that executes in the presence of multiple hardware units that will guarantee that two or more instruction streams will make progress in a single unit of time. The differentiating factor from a concurrent program executing via time-slicing is that in a parallel program, at least two streams make progress at a given time. This is most often a direct consequence of the presence of multiple hardware units that can support simultaneous execution of distinct streams."

Introduction Concurrency in Programming Languages by Matthew J. Sottile, Timothy G. Mattson, Craig E Rasmussen

“Concurrency vs. parallelism

- Concurrency is about dealing with lots of things at once.

- Parallelism is about doing lots of things at once.

- Not the same, but related.

- Concurrency is about structure, parallelism is about execution.

- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.”

Rob Pike

Le invitamos a escuchar la siguiente conferencia de Rob Pike: “***Concurrency Is Not Parallelism***”

2.- ¿ Qué problemas puede surgir cuando hay concurrencia ?

Básicamente hay dos grandes grupos en los que caen todos los problemas. El primero es protección del recurso compartido, y el segundo es sincronización.

Recurso compartido, en estos casos hay recursos tales como variables globales, archivos, impresoras, etc. que debido a la ejecución concurrentes de rutinas que hacen uso de alguno de estos estos recursos, es necesario exigir que un solo hilo acceda al recurso (exclusión mutua) para que no haya resultados inesperados. Un ejemplo clásico es la variable global, donde varios hilos incrementa la variable una cantidad fija de veces. Al final el total debe ser igual a la suma de los incrementos de cada hilo. Sin embargo esto a veces no se llega a cumplir.

Sincronización, hay situaciones donde la concurrencia puede beneficiarnos frente a una programación secuencial. Imagínese la siguiente situación: se tiene una fábrica de gaseosa donde se llena la botella y por otro lado se fabrica la tapa. Llenar 1000 botellas toma 20 minutos, fabricar 1000 tapas toma 10 minutos. Si se ejecuta de forma secuencial, todo tomaría 30 minutos. Sin embargo se podría ahorra tiempo si se ejecutan de forma concurrente la fabricación de las tapas y la del llenado de la botella. En este caso hay que considerar que si uno de ellos termina primero debe esperar al otro para lograr tener la botella con su tapa. A este proceso se le denomina sincronización.

Todos estos problemas surgen porque los hilos acaban de forma no determinista, compitiendo unos con otros. Por ese motivo a este tipo de problemas también se les denomina ***Race condition***.

En estos casos su tarea será asegurar la exclusión mutua al recurso compartido si este presenta problemas de *race condition*. O sincronizar los procesos para que cumplan un flujo específico. En ambos casos deberá conseguir concurrencia lo más que se pueda. Si anula por completo la concurrencia, significa que finalmente se están ejecutando secuencialmente. Y esto no es una solución.

3. Gorutinas y canales

En Go, cada actividad ejecutándose concurrentemente se le denomina *gorutina*. Considere un programa que tiene dos funciones, una que realiza cálculos y la otra escribe alguna salida, y asuma que ninguna función invoca a la otra. Un programa secuencial puede llamar una función y luego llamar a la otra, pero en programa concurrente con dos o más *gorutinas*, las llamadas a ambas funciones pueden estar activas al mismo tiempo.

Cuando un programa inicia, su única *gorotina* es aquella que invoca a la función `main`, así que nosotros la llamaremos *main gorutina*. Nuevas *gorutinas* son creadas por la instrucción `go`. Sintácticamente, una instrucción go es una función ordinaria o un prefijo de llamada a un método con la palabra clave go"

A continuación, un ejemplo sencillo:

```go
package main

import (
    "fmt"
    "time"
)

func routine(n int) {
    fmt.Printf("I'm goroutine %d\n", n)
}

func main() {
    for x := 0; x < 5; x++ {
        go routine(x)
    }
    time.Sleep(1 * time.Second)
}
```

Al ejecutarse en una terminal por tres veces consecutivas, se obtiene la siguiente salida:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej1
I'm goroutine 1
I'm goroutine 0
I'm goroutine 4
I'm goroutine 2
I'm goroutine 3
alejandro@abdebian:2022-1 lab 2 SO$ ./ej1
I'm goroutine 1
I'm goroutine 2
I'm goroutine 4
I'm goroutine 3
I'm goroutine 0
alejandro@abdebian:2022-1 lab 2 SO$ ./ej1
I'm goroutine 0
I'm goroutine 3
I'm goroutine 2
I'm goroutine 4
I'm goroutine 1
alejandro@abdebian:2022-1 lab 2 SO$
```

Como se puede observar las ejecuciones de las *gorutinas* son concurrentes, y el planificador en cada caso ha elegido un orden diferente de ejecución. Este resultado nos deja una lección importante: *no se puede asumir de antemano el orden en que se ejecutaran las gorutinas*.

Otro detalle importante se encuentra en la línea 16. El proceso *duerme* durante 1 segundo. En caso que se elimine la línea 16, sucederá lo siguiente: la *gorutina* principal terminará antes que las demás *gorutinas* y el programa completo acabará sin que se ejecuten el resto de *gorutinas*. En Go no hay forma de esperar a una o varias *gorutinas* salvo que se emplee un contador de tipo especial. Este tipo de contador es conocido como `sync.WaitGroup`, y el código de abajo muestra cómo usarlo:

```
 1  package main
 2
 3  import (
 4      "fmt"
 5      "sync"
 6  )
 7
 8  func routine(n int) {
 9      defer wg.Done()
10      fmt.Printf("I'm goroutine %d\n", n)
11  }
12
13  var wg sync.WaitGroup
14
15  func main() {
16      for x := 0; x < 5; x++ {
17          wg.Add(1)
18          go routine(x)
19      }
20      wg.Wait()
21  }
```

Después de compilarlo, ejecutamos el programa varias veces:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej2
I'm goroutine 4
I'm goroutine 1
I'm goroutine 3
I'm goroutine 0
I'm goroutine 2
alejandro@abdebian:2022-1 lab 2 SO$ ./ej2
I'm goroutine 3
I'm goroutine 0
I'm goroutine 4
I'm goroutine 2
I'm goroutine 1
alejandro@abdebian:2022-1 lab 2 SO$ ./ej2
I'm goroutine 0
I'm goroutine 4
I'm goroutine 2
I'm goroutine 3
I'm goroutine 1
alejandro@abdebian:2022-1 lab 2 SO$
```

Usted puede ver este tipo de variable como una barrera. Cada vez que se invoca a `Add(delta)` el contador se incrementa en `delta`. Cada vez que se llama a `Done()` le contador se decrementa en 1. La *gorutina* que invoca a `Wait()` se bloquea y se libera solo cuando el contador se encuentra en 0.

Canales

Un canal es un mecanismo de comunicación que permite a una *gorutina* enviar valores a otra *gorutina*. Cada canal es un conducto para valores de un tipo particular, llamado *tipo de elemento del canal*.

Para crear un canal, se usará la instrucción `make`, por ejemplo si sobre un canal sin búfer se desea enviar números enteros, se crea de la siguiente forma:

```
ch := make(chan int)
```

Un canal tiene dos principales operaciones, enviar y recibir. A continuación la forma sintáctica de cómo se llevan a cabo estas operaciones sobre un canal:

```
ch <- x      // se envía el valor que contiene la variable x al canal ch
x = <- ch    // se recibe del canal un valor y lo asigna a la variable x
<- ch        // se recibe del canal un valor, el mismo que es descartado.
```

Una tercera operación permitida sobre un canal es:

```
close(ch)
```

Esta tercera operación coloca internamente una bandera indicando que no se enviarán más valores sobre el canal, por tal motivo cualquier intento de seguir enviando producirá un error. Pero la *gorutina* que estaba recibiendo se desbloqueará y podrá recibir todos los valores restantes, y cuando no haya más, obtendrá el cero del tipo de canal. Por ejemplo si el canal fue de enteros y se cerró el canal de parte del que enviaba valores, el que los recibe después de drenar todos los valores, obtendrá 0.

Canales sin búfer

Una operación de envío en un canal sin búfer ocasiona que se bloquee la *gorutina* que realizó esta operación, hasta que otra *gorutina* ejecute una correspondiente operación de recepción en el mismo canal, momento en el cual se transmite un valore y ambas *gorutinas* pueden continuar. De forma inversa, si la operación de recepción se intentó primero, esta *gorutina* se bloquea hasta que se lleve a cabo una operación envío en el mismo canal.

Como consecuencia del comportamiento arriba descrito la *gorutina* que envía y la que recibe se sincronizan. Por este motivo a los canales sin búfer a veces se les denominan canales síncronos. Es importante hacer notar que cuando un valor es enviado a un canal sin búfer, la recepción del valor sucede antes que se vuelva a despertar la *gorutina* que envió.

Abajo se se muestra un ejemplo donde se crean 2 canales para comunicar 3 *gorutinas* y formar un pipeline, donde la salida de una *gorutina* sirve como entrada de la siguiente *gorutina*:

```go
package main

import "fmt"

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; x < 20; x++ {
            naturals <- x
        }
        close(naturals)
    }()

    // Squarer
    go func() {
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()

    // Printer (in main goroutine)
    for x := range squares {
        fmt.Println(x)
    }
}
```

Hay que resaltar que la instrucción `range` en el `for`, recibe los valores del canal, los asigna a la variable *x*, y termina de forma automática cuando ya no hay más valores que drenar. Esto es solo posible porque los canales se han cerrado. En caso contrario, las *gorutinas* se hubieran quedado bloqueadas.

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej3
0
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
alejandro@abdebian:2022-1 lab 2 SO$
```

Los mensajes enviados sobre los canales tienen dos importantes aspectos. Cada mensaje tiene un valor, pero a veces solo es importante el hecho de comunicarse y el momento en que esto ocurre. En Go se llama *eventos* a los mensajes cuando se desea enfatizar este aspecto. Cuando el evento no lleva información adicional, esto es, el único propósito es sincronizar, se enfatizará este hecho usando un canal cuyo elemento tipo es `struct{}`, aunque es común usar canales de `bool` o `int` para el mismo propósito, donde `done <- 1` es más corte que `done <- struct{}{}`.

Usaremos el primer ejemplo, donde sincronizamos la *gorutina* principal con la última *gorutina*, en lugar de usar la variable tipo `sync.WaitGroup`.

```
 1  package main
 2
 3  import "fmt"
 4
 5  func main() {
 6      done := make(chan struct{})
 7      for x := 0; x < 5; x++ {
 8          go func(count int) {
 9              fmt.Printf("I'm goroutine %d\n", count)
10              done <- struct{}{}
11          }(x)
12      }
13      for x := 0; x < 5; x++ {
14          <-done
15      }
16  }
```

En el ejemplo de arriba se ha empleado funciones anónimas, también llamadas funciones literales o lambda.

Canales con búfer

Un canal con búfer tiene una cola de elementos. El tamaño máximo de la cola es determinada en el momento de la creación, por el argumento de capacidad para `make`. El siguiente ejemplo crea un canal con búfer para almacenar tres valores `string`:

```
ch = make(chan string,3)
```

Una operación de envío sobre el canal con búfer inserta un elemento en la parte posterior de la cola, y una operación de recepción remueve un elemento de la cabeza de la cola. Si el canal está lleno, la operación de envío bloquea su *gorutina* hasta una *gorutina* de recepción haga espacio en el canal con búfer. De forma opuesta, si un canal está vacío, una operación de recepción se bloquea hasta que alguna *gorutina* envíe algún valor al canal.

En el ejemplo anterior, se puede enviar hasta 3 valores sin que la *gorutina* se bloquee:

```
ch <-  “A”
ch <-  “B”
ch <-  “C”
```

En este punto el canal está lleno y el envío de una cuarta cadena debería bloquearlo.

Si se recibe un valor,

```
fmt.Println(<- ch)  // “A”
```

el canal ni está vacío ni está lleno, de forma que se puede realizar una operación de envío o de recepción sin que se bloquee.

En el evento poco probable que un programa necesite saber la capacidad del búfer del canal, puede obtenerlo invocando la función cap:

```
fmt.Println(cap(ch))  // “3”
```

Después de dos operaciones más el canal se encuentra vacío nuevamente, y una cuarta debería bloquearlo:

```
fmt.Println(<-  ch)  // “B”
fmt.Println(<-  ch)  // “C”
```

En este ejemplo, todas las operaciones de envío y recepción fueron realizadas por la misma *gorutina*, pero e programas reales ellas son usualmente ejecutadas por diferentes *gorutinas*.

Multiplexando con `select`

La instrucción `select` presenta la siguiente forma general:

```
select   {
       case <- ch1:
           // ...
        case x:= <-  ch2:
           // ...
        case ch3 <-  y:
           //
        default:
           // ...
}
```

Al igual que una instrucción switch, esta tiene un número de casos y un caso opcional `default`. Cada caso especifica una comunicación (una operación de envío o de recepción sobre algún canal) y un bloque de sentencias asociado a cada caso. Una expresión de recepción puede aparecer por sí misma, como en el primer caso (del ejemplo de arriba), o con una declaración corta de una variable, como en el segundo caso; la segunda forma permite hacer referencia al valor recibido.

Un `select` espera hasta que alguna comunicación para algún caso se encuentre listo para proceder. Luego realiza esas comunicaciones y ejecuta las instrucciones asociadas al caso; las otras comunicaciones no suceden. Un `select` sin casos, `slect{}`, espera por siempre.

Si múltiples casos están listos, `slect` elige uno al azar, con lo cual cada canal tiene la misma oportunidad de ser elegido.

Algunas veces se desea tratar de enviar o recibir sobre un canal pero evitando bloquearse si el canal no está listo (una comunicación no bloqueante). Una instrucción `select` también puede hacer eso. Un `select` puede tener un `default`, la cual especifica que hacer cuando ninguna de las comunicaciones puede proceder inmediatamente.

4.- Concurrencia con variables compartidas

A continuación se muestra el clásico problema de la variable que es incrementada por 1 un millón de veces de por varios hilos, en este ejemplo 5, y cuyo valor final esperado debería ser de 5 millones, sin embargo el resultado no es el esperado.

```go
package main

import (
    "fmt"
    "sync"
)

func worker() {
    for x := 0; x < 1000000; x++ {
        count++
    }
    wg.Done()
}

var (
    count int
    wg    sync.WaitGroup
)

func main() {
    for x := 0; x < 5; x++ {
        wg.Add(1)
        go worker()
    }
    wg.Wait()
    fmt.Printf("El valor esperado de count es: 5000000 y el valor final es %d\n", count)
}
```

Al ejecutarlo varias veces, se obtiene los siguientes resultados:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej5
El valor esperado de count es: 5000000 y el valor final es 2003075
alejandro@abdebian:2022-1 lab 2 SO$ ./ej5
El valor esperado de count es: 5000000 y el valor final es 2025385
alejandro@abdebian:2022-1 lab 2 SO$ ./ej5
El valor esperado de count es: 5000000 y el valor final es 1953149
alejandro@abdebian:2022-1 lab 2 SO$ ./ej5
El valor esperado de count es: 5000000 y el valor final es 2086963
alejandro@abdebian:2022-1 lab 2 SO$ ./ej5
El valor esperado de count es: 5000000 y el valor final es 2024798
alejandro@abdebian:2022-1 lab 2 SO$
```

En este caso debemos proteger la sección crítica con variables de tipo `sync.Mutex`, estas permitirán bloquear y desbloquear (de forma atómica) las secciones críticas.

Las funciones `Lock` y `Unlock` permiten bloquear y desbloquear el código que deseamos proteger. El uso de estas funciones no deben ser indiscriminado, por el contrario deben ser usadas cuando sean estrictamente necesario y solo las líneas que lo necesiten. Caso contrario podría ocasionar que las ejecuciones se ralenticen y por tanto ocasionar un cuello de botella.

A continuación el código que permite resolver el problema:

```go
package main

import (
    "fmt"
    "sync"
)

func worker() {
    for x := 0; x < 1000000; x++ {
        mu.Lock()
        count++
        mu.Unlock()
    }
    wg.Done()
}

var (
    count int
    wg    sync.WaitGroup
    mu    sync.Mutex
)

func main() {
    for x := 0; x < 5; x++ {
        wg.Add(1)
        go worker()
    }
    wg.Wait()
    fmt.Printf("El valor esperado de count es: 5000000 y el valor final es %d\n", count)
}
```

Y las respectivas salidas:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej6
El valor esperado de count es: 5000000 y el valor final es 5000000
alejandro@abdebian:2022-1 lab 2 SO$ ./ej6
El valor esperado de count es: 5000000 y el valor final es 5000000
alejandro@abdebian:2022-1 lab 2 SO$ ./ej6
El valor esperado de count es: 5000000 y el valor final es 5000000
alejandro@abdebian:2022-1 lab 2 SO$ ./ej6
El valor esperado de count es: 5000000 y el valor final es 5000000
alejandro@abdebian:2022-1 lab 2 SO$ ./ej6
El valor esperado de count es: 5000000 y el valor final es 5000000
alejandro@abdebian:2022-1 lab 2 SO$
```

Ejercicios

Se tiene el siguiente programa en Go

```go
package main

import (
    "fmt"
    "sync"
)

var wg sync.WaitGroup

func worker1() {
    fmt.Printf("Sistemas ")
    wg.Done()
}

func worker2() {
    fmt.Printf("INF239 ")
    wg.Done()
}

func worker3() {
    fmt.Printf("Operativos ")
    wg.Done()
}

func worker4() {
    fmt.Printf("\n")
    wg.Done()
}

func main() {
    wg.Add(4)
    go worker1()
    go worker2()
    go worker3()
    go worker4()
    wg.Wait()
}
```

Al ejecutarlo repetidas veces se obtienen diferentes salidas, como se puede apreciar en la siguiente imagen:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej7
Sistemas INF239 Operativos
alejandro@abdebian:2022-1 lab 2 SO$ ./ej7
Sistemas Operativos
INF239 alejandro@abdebian:2022-1 lab 2 SO$ ./ej7

INF239 Operativos Sistemas alejandro@abdebian:2022-1 lab 2 SO$ ./ej7

INF239 Operativos Sistemas alejandro@abdebian:2022-1 lab 2 SO$ ./ej7
Sistemas INF239
Operativos alejandro@abdebian:2022-1 lab 2 SO$ ./ej7

Operativos Sistemas INF239 alejandro@abdebian:2022-1 lab 2 SO$
```

Se desea sincronizarlo para que la salida sea:

```
INF239 Sistemas Operativos
```

A continuación una solución:

```go
package main

import (
    "fmt"
    "sync"
)

var (
    wg  sync.WaitGroup
    ch1 chan struct{} = make(chan struct{}, 1)
    ch2 chan struct{} = make(chan struct{}, 1)
    ch3 chan struct{} = make(chan struct{}, 1)
)

func worker1() {
    <-ch1
    fmt.Printf("Sistemas ")
    wg.Done()
    ch2 <- struct{}{}
}

func worker2() {
    fmt.Printf("INF239 ")
    wg.Done()
    ch1 <- struct{}{}
}

func worker3() {
    <-ch2
    fmt.Printf("Operativos ")
    wg.Done()
    ch3 <- struct{}{}
}

func worker4() {
    <-ch3
    fmt.Printf("\n")
    wg.Done()
}

func main() {
    wg.Add(4)
    go worker1()
    go worker2()
    go worker3()
    go worker4()
    wg.Wait()
}
```

Se muestra el resultado de diferentes ejecuciones:

```
alejandro@abdebian:2022-1 lab 2 SO$ go build ej8.go
alejandro@abdebian:2022-1 lab 2 SO$ ./ej8
INF239 Sistemas Operativos
alejandro@abdebian:2022-1 lab 2 SO$ ./ej8
INF239 Sistemas Operativos
alejandro@abdebian:2022-1 lab 2 SO$ ./ej8
INF239 Sistemas Operativos
alejandro@abdebian:2022-1 lab 2 SO$ ./ej8
INF239 Sistemas Operativos
alejandro@abdebian:2022-1 lab 2 SO$
```

Ejercicios

Se tiene un programa en Go que tiene 5 goroutines. Cada una imprime uno de los siguientes caracteres: A,B, C, D, E. Usted debe sincronizarlos para que cumplan las siguientes exigencias. Elabore un programa por cada caso.

a) La secuencia permitida es: `ABCDEABCDEABCDEABCDEABCDEABCDE ...`
b) La secuencia permitida es: `ACDEBCDEACDEBCDEACDEBCDEACDEBCDE...`
c) La secuencia permitida es: `(A ó B)CDE(A ó B)CDE(A ó B)CDE(A ó B)CDE ...`
d) La secuencia permitida es: `(A ó B)CE(A ó B)(A ó B)DE(A ó B)CE(A ó B)(A ó B)DE …`

El siguiente programa está formado por dos goroutines, la primera de ellas (el productor) trata de llenar un búfer (arreglo de 5 enteros) y la segunda (consumidor) trata de retirar los elementos producidos.

```go
package main

import (
    "fmt"
    "sync"
)

var (
    buffer [5]int = [5]int{-1, -1, -1, -1, -1}
    index  int
    wg     sync.WaitGroup
)

func productor() {
    for n := 0; n < 20; n++ {
        item := n * n
        index = n % 5
        buffer[index] = item
        fmt.Printf("productor %d %d %v\n", index, item, buffer)

    }
    wg.Done()
}

func consumidor() {
    var item int
    for n := 0; n < 20; n++ {
        index = n % 5
        item = buffer[index]
        buffer[index] = -1
        fmt.Printf("consumidor %d %d %v\n", index, item, buffer)
    }
    wg.Done()
}

func main() {
    wg.Add(2)
    go consumidor()
    go productor()
    wg.Wait()
}
```

Debido a que no hay ningún tipo de sincronización, ni se protege la sección crítica las salida es inconsistente:

```
alejandro@abdebian:2022-1 lab 2 SO$ ./ej9
productor 0 0 [0 -1 -1 -1 -1]
productor 1 1 [-1 1 -1 -1 -1]
productor 2 4 [-1 1 4 -1 -1]
productor 3 9 [-1 1 4 9 -1]
productor 4 16 [-1 1 4 9 16]
productor 0 25 [25 1 4 9 16]
productor 1 36 [25 36 4 9 16]
productor 2 49 [25 36 49 9 16]
productor 3 64 [25 36 49 64 16]
productor 4 81 [25 36 49 64 81]
productor 0 100 [100 36 49 64 81]
productor 1 121 [100 121 49 64 81]
productor 2 144 [100 121 144 64 81]
consumidor 0 0 [-1 -1 -1 -1 -1]
consumidor 1 121 [100 -1 144 169 81]
consumidor 2 144 [100 -1 -1 169 81]
consumidor 3 169 [100 -1 -1 -1 81]
consumidor 4 81 [100 -1 -1 -1 -1]
consumidor 0 100 [-1 -1 -1 -1 -1]
consumidor 1 -1 [-1 -1 -1 -1 -1]
productor 3 169 [100 121 144 169 81]
productor 4 196 [-1 -1 -1 -1 196]
productor 0 225 [225 -1 -1 -1 196]
productor 1 256 [225 256 -1 -1 196]
productor 2 289 [225 256 289 -1 196]
productor 3 324 [225 256 289 324 196]
productor 4 361 [225 256 289 324 361]
consumidor 2 -1 [-1 -1 -1 -1 -1]
consumidor 3 324 [225 256 289 -1 361]
consumidor 4 361 [225 256 289 -1 -1]
consumidor 0 225 [-1 256 289 -1 -1]
consumidor 1 256 [-1 -1 289 -1 -1]
consumidor 2 289 [-1 -1 -1 -1 -1]
consumidor 3 -1 [-1 -1 -1 -1 -1]
consumidor 4 -1 [-1 -1 -1 -1 -1]
consumidor 0 -1 [-1 -1 -1 -1 -1]
consumidor 1 -1 [-1 -1 -1 -1 -1]
consumidor 2 -1 [-1 -1 -1 -1 -1]
consumidor 3 -1 [-1 -1 -1 -1 -1]
consumidor 4 -1 [-1 -1 -1 -1 -1]
alejandro@abdebian:2022-1 lab 2 SO$
```

Modifique el programa para que se cumpla los siguientes requisitos:

a) Se sincronice el productor de forma que una vez que se ha llegado al límite del arreglo, la *gorutina* se bloquee hasta que al menos haya una entrada libre.

b) Se sincronice el consumidor de forma que si no hay elementos en el arreglo se bloquee hasta que haya por lo menos un elemento producido.

c) Proteja la sección crítica de forma que el consumidor no consuma un valor que no se ha producido o que el productor asigne un valor a una posición que aún no se ha consumido.

Prof. Alejandro T. Bello Ruiz