Paralelismo y Concurrencia

Definiciones

Concurrencia

Dictionary

Definitions from la Real Academia Española · Learn more

concurrencia

1. nombre femenino
   Acción y efecto de concurrir.

   Opposite: divergencia

2. nombre femenino
   Conjunto de personas que asisten a un acto o reunión.

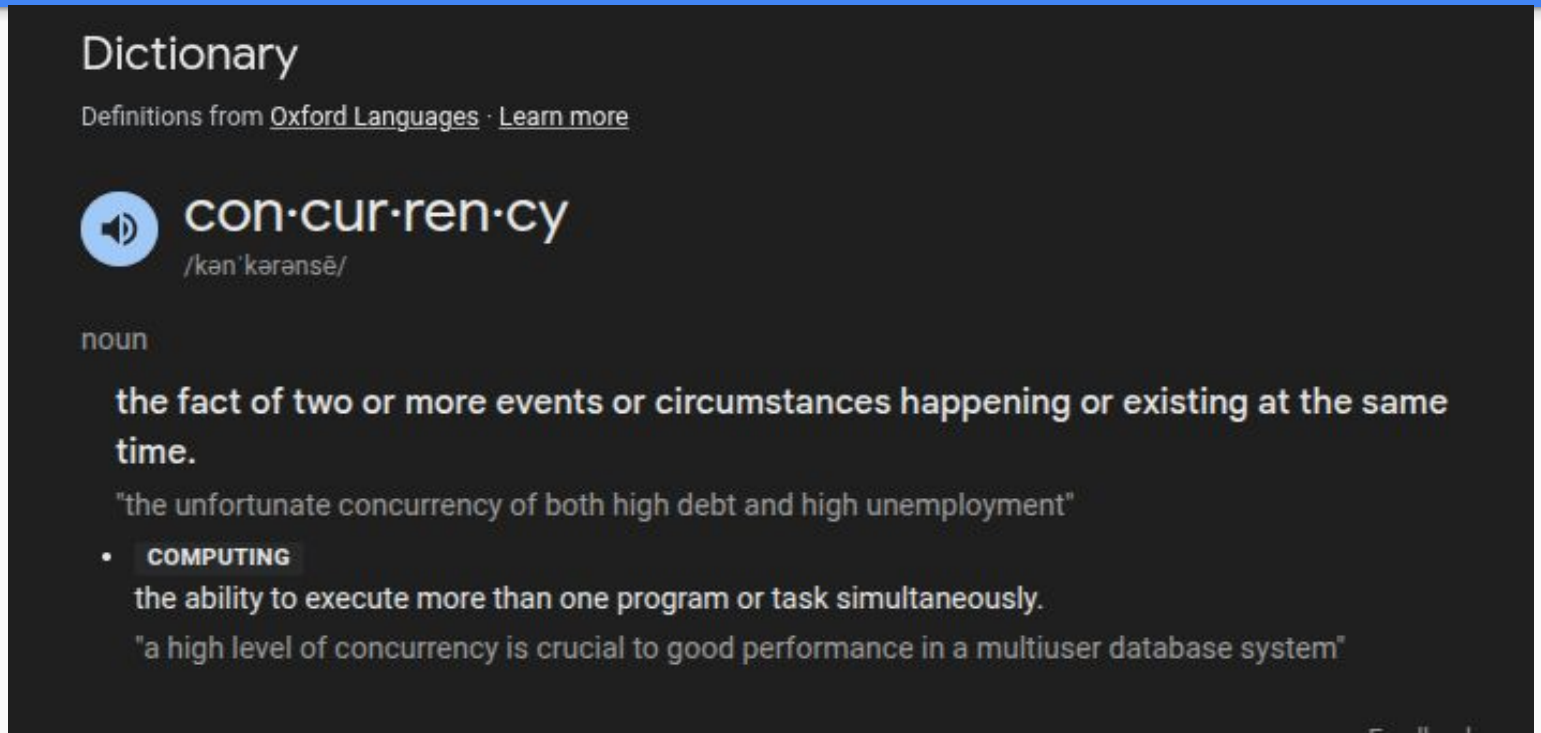
   Similar: afluencia | muchedumbre | tropel | gentío | masa | multitud

3. nombre femenino
   Coincidencia, concurso simultáneo de varias circunstancias.

   Similar: coincidencia | conjunción | convergencia | confluencia | simultaneidad

4. nombre femenino
   Asistencia, participación.

Concurrencia



Dictionary

Definitions from Oxford Languages · Learn more

con·cur·ren·cy

/kənˈkərənsē/

noun

the fact of two or more events or circumstances happening or existing at the same time.

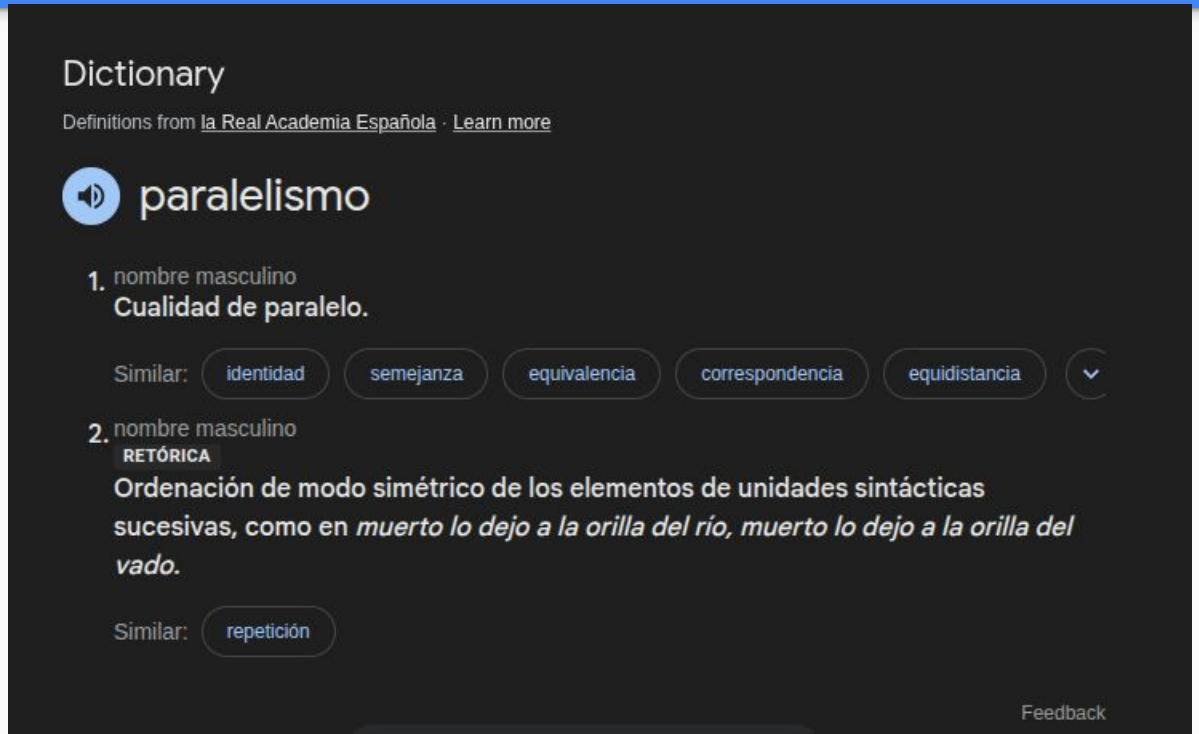
"the unfortunate concurrency of both high debt and high unemployment"

- **COMPUTING**

  the ability to execute more than one program or task simultaneously.

  "a high level of concurrency is crucial to good performance in a multiuser database system"

Paralelismo



Dictionary

Definitions from la Real Academia Española · Learn more

paralelismo

1. nombre masculino
   Cualidad de paralelo.

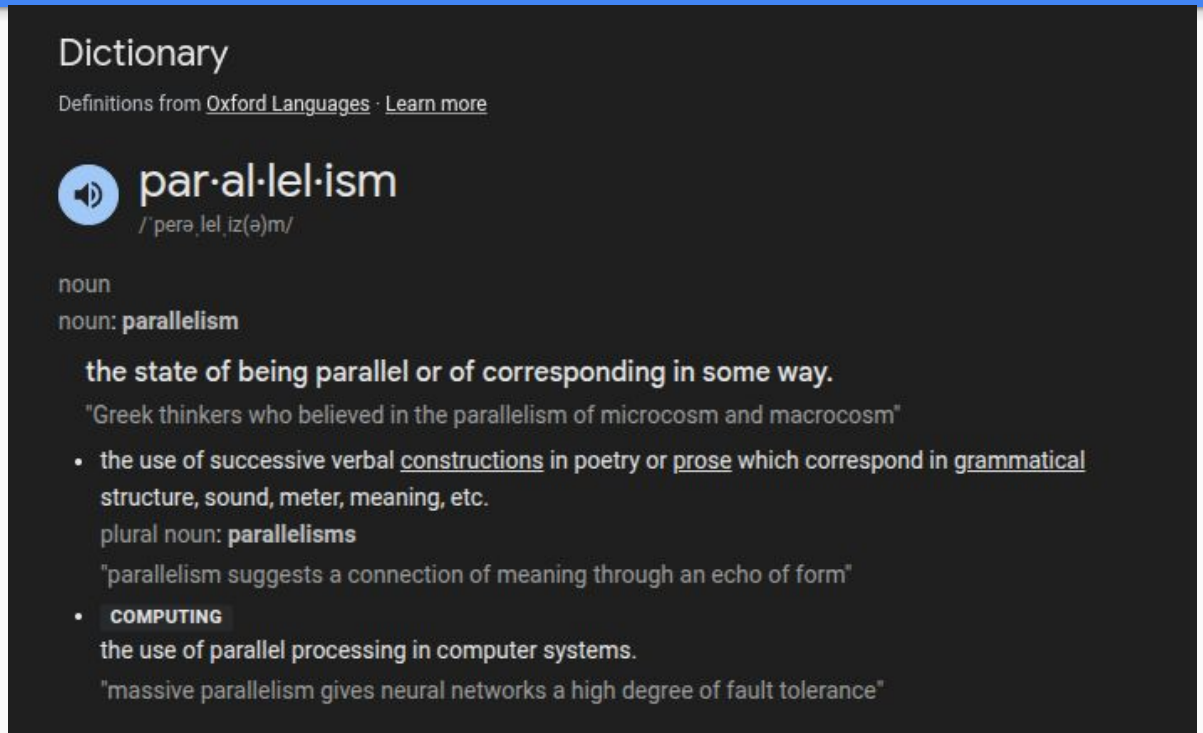
   Similar: identidad | semejanza | equivalencia | correspondencia | equidistancia

2. nombre masculino
   RETÓRICA
   Ordenación de modo simétrico de los elementos de unidades sintácticas sucesivas, como en *muerto lo dejo a la orilla del río, muerto lo dejo a la orilla del vado.*

   Similar: repetición

Feedback

Paralelismo



Dictionary

Definitions from Oxford Languages · Learn more

par·al·lel·ism

/ˈperəˌlelˌiz(ə)m/

noun

noun: **parallelism**

the state of being parallel or of corresponding in some way.

"Greek thinkers who believed in the parallelism of microcosm and macrocosm"

- **the use of successive verbal constructions in poetry or prose which correspond in grammatical structure, sound, meter, meaning, etc.**

  plural noun: **parallelisms**

  "parallelism suggests a connection of meaning through an echo of form"

- **COMPUTING**

  the use of parallel processing in computer systems.

  "massive parallelism gives neural networks a high degree of fault tolerance"

En esencia…

- Concurrencia:
  - Patrón de diseño
  - Modo de organizar un sistema tal que ciertas operaciones se puedan efectuar independientemente
- Paralelismo
  - Implementación específica de una solución concurrente
  - Cualidad de efectuar tareas independientes simultáneamente

¿Por qué es difícil la concurrencia?

Los programas concurrentes enfrentan tres desafíos principales:

1. Compartir información entre hilos.
2. Evitar interferencia al acceder a recursos compartidos.
3. Garantizar la secuencia correcta de ejecución.

Condiciones de carrera

Definición: Cuando dos o más hilos acceden al mismo recurso y el resultado depende del orden de ejecución.

Consecuencia: errores impredecibles, difíciles de detectar y depurar.

Semáforos – Control de acceso atómico

Variable entera protegida por dos operaciones atómicas: `wait` (down) y `signal` (up).

Permiten controlar la entrada a regiones críticas y sincronizar procesos.

Son una herramienta versátil, aunque de bajo nivel. Se usan ampliamente en problemas como productor-consumidor o lectores-escritores.

Monitores – Exclusión mutua automática

Agrupan datos y procedimientos.

Solo un hilo puede ejecutar código del monitor a la vez.

Usan variables de condición con `wait()` y `signal()` para sincronización.

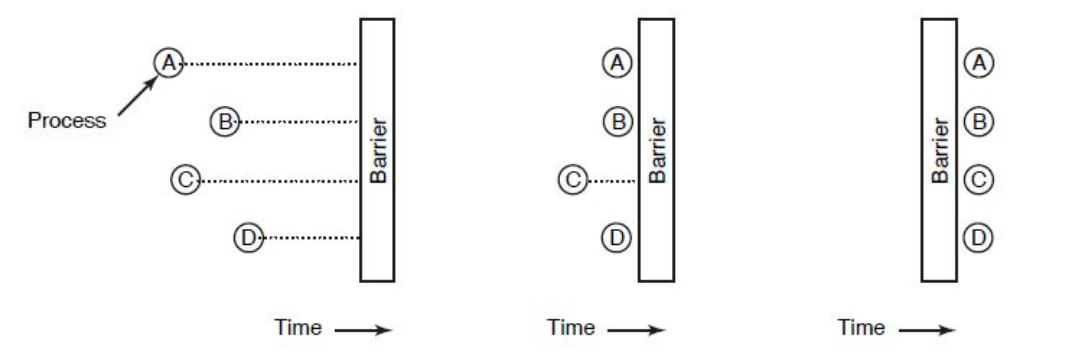
A diferencia del semáforo, el monitor evita que el programador tenga que preocuparse por el bloqueo explícito. La sincronización se integra al lenguaje.

Barriers – Sincronización por fases

Permiten que **todos los hilos esperen en un punto común**.

Solo cuando **todos han llegado**, pueden avanzar a la siguiente fase.

Útil en cálculos por etapas, como simulaciones o paralelización por bloques.



Countdown – Esperar tareas paralelas

Inicializa un contador; cada hilo que termina hace Signal().

Cuando el contador llega a 0, los que estaban en Wait() continúan.

A diferencia de Barrier, no es reutilizable ni cíclico.

Se usa cuando queremos lanzar muchas tareas, y continuar solo cuando todas terminen

Concurrencia en Go

Goroutines

- Esencialmente hilos implementados en el runtime de Go, funciones ejecutándose de forma independiente en el mismo espacio de memoria
- Sobre un mismo hilo del sistema operativo pueden correr varias goroutines
- Más ligeras que hilos del sistema operativo, es práctico tener cientos de miles y hasta millones corriendo a la vez
- La función **`main`** corre sobre la goroutine principal

Goroutines

```
func simpleConcurrency() {
»       go func () {
»       »       println("hello!")
»       }()
»
»       // no output
}
```

Goroutines

```
func simpleConcurrency() {
»       go func () {
»       »       println("hello!")
»       }()
»
»       time.Sleep(time.Second)
»
»
»       // output: hello!
}
```

Comunicación

- Los procesos concurrentes por sí solos no son muy útiles
  - Necesitamos una forma de establecer comunicación entre la goroutine principal y las demás
- Existen varias soluciones a esto
  - Por defecto en Go, built-in en el lenguaje: Canales
  - Librería sync en la librería estándar
    - Mutexes
    - WaitGroups
    - Etc.

“Don’t communicate by sharing memory, share memory by communicating”

- Rob Pike

Canales

Técnica de comunicación entre goroutines, built-in en el lenguaje

Canales

```
func usingChannels() {
	ch := make(chan int)

	go func () {
		for i := range 10 {
			ch <- i
		}

	}()

	for {
		i := <- ch
		println(i)
	}

	// fatal error: all goroutines are asleep - deadlock!
}
```

Operaciones sobre canales

- Cerrar:
  - Señala que del canal no se podrán recibir valores válidos y ya no se podrá escribir en el canal

Operaciones sobre canales

- Recibir:
  - Extrae un valor de la cola de goroutines esperando a enviar un valor, si no hay ninguna, bloquea hasta que la haya
  - Se puede recibir dos valores, el segundo indica si el canal está cerrado, el primero es el valor en sí

Operaciones sobre canales

- Enviar:
  - Entrega un valor a la primera goroutine en la cola para recibir valores, en caso de que no haya ninguna, bloquea hasta que la haya
  - Si se envía sobre un canal cerrado la goroutine entrará en pánico

Canales

```
func receivingStuff() {
»       ch := make(chan int)
»
»       go func () {
»       »       ch <- 42
»       »       close(ch)
»       }()
»
»       v, ok := <-ch
»
»       println(v) // 42
»       println(ok) // true
»
»       v, ok = <-ch
»
»       println(v) // 0
»       println(ok) // false
}
```

Canales

```
func usingChannelsRight() {
»       ch := make(chan int)
»
»       go func () {
»       »       for i := range 10 {
»       »       »       ch <- i
»       »       }
»       »       close(ch)
»       »
»       }()
»
»       for {
»       »       i, ok := <- ch
»       »       if !ok {
»       »       »       break
»       »       }
»       »       println(i)
»       }
»
»       // output: what you expect
}
```

Canales

```
func usingChannelsIdiomatically() {
»       ch := make(chan int)
»
»       go func () {
»       »       for i := range 10 {
»       »       »       ch <- i
»       »       }
»       »       close(ch)
»       »
»       }()
»
»       for i := range ch {
»       »       println(i)
»       }
»
»       // output: what you expect
}
```

Canales

```
func whatsThisAbout() {
»       ch := make(chan int)
»
»       ch <- 42
»
»       println(<-ch)
»
»       // fatal error: all goroutines are asleep - deadlock!
}
```

La goroutine que estaba corriendo la función se bloqueó pues nadie consumió el 42 que intentó enviar al canal

Canales

```
func bufferedChannel() {
»       ch := make(chan int, 1)
»
»       ch <-42
»
»       println(<-ch) // 42
}
```

Canales con búfer: Contienen una cola de valores, tal que si no está llena y una goroutine envía un valor por el canal, esta no se bloquea

Si está llena sí hay bloqueo

Canales con búfer vs. sin búfer

Los canales tienen una cola de valores, además de una cola de goroutines esperando a enviar un valor, y una cola de goroutines esperando a recibir un valor (todas son generalmente FIFO)

- Canales con búfer
  - Cola de valores con al menos un espacio: es posible colocar ahí un valor sin bloquear la goroutine que lo haga en caso de que quede espacio en la cola
- Canales sin búfer
  - Cola de valores vacía, si no existen goroutines esperando a recibir un valor, colocar un valor en el canal bloquea la goroutine

Propiedades de canales

- Len:
  - Devuelve la cantidad de valores en la cola de valores de un canal
- Cap:
  - Devuelve la capacidad máxima de valores en la cola de un canal

```
ch := make(chan bool, 2)
ch <- true
l := len(ch) // 1
c := cap(ch) // 2
```

```
ch := make(chan bool)
// ch <- true
l := len(ch) // 0
c := cap(ch) // 0
```

Carreras - Select

Select permite efectuar la primera comunicación que se pueda entre una serie de comunicaciones, ya sea enviar a o recibir de un canal.

Canales

```
func race() {
»       webService := func(computeTime time.Duration) <-chan bool {
»       »       ch := make(chan bool)
»       »       go func() {
»       »       »       time.Sleep(computeTime)
»       »       »       ch <- true
»       »       }()
»       »
»       »       return ch
»       }
»
»       ch1 := webService(time.Second)
»       ch2 := webService(time.Second * 2)
»
»       select {
»       case _ = <-ch1:
»       »       println("Webservice 1 returned")
»       case _ = <-ch2:
»       »       println("Webservice 2 returned")
»       }
}
```

Canales

```
func ioRace() {
        iCh := make(chan string)
        oCh := make(chan string)

        go func() {
                for i := 0; ; i += 1 {
                        iCh <- fmt.Sprintf("Hello %d", i)
                        r := rand.Intn(30)
                        time.Sleep(time.Millisecond*time.Duration(100*r))
                }
        }()

        go func() {
                for i := 0; ; i += 1 {
                        println(<-oCh)
                        r := rand.Intn(30)
                        time.Sleep(time.Millisecond*time.Duration(100*r))
                }
        }()

        o := "Noone wrote"

        for {
                select {
                case s := <-iCh:
                        o = s
                case oCh <- o:
                        // do nothing
                }

        }
}
```

¿Por qué canales y goroutines?

- La sintaxis facilita la legibilidad y facilidad de implementación de soluciones concurrentes
- Los canales y las operaciones que se pueden hacer sobre ellos facilitan el razonamiento sobre soluciones concurrentes
  - Incitan a la transferencia de propiedad de valores entre operaciones concurrentes, lo cual mejora la seguridad de memoria del programa y su mantenibilidad
- La implementación in-house del runtime de Go facilita chequeos como detección de deadlocks (que todas las goroutines estén esperando) e incluso de carreras, además de optimizaciones significativas

¿Por qué canales y goroutines?

Podemos pensarlo como la llegada de la programación orientada a objetos; antes de esta ya existían:

- El encapsulamiento de información (header files de C)
- La herencia de miembros de datos (simplemente interpretar un apuntador de un tipo como uno de otro con su mismo prefijo de miembros de datos)
- Polimorfismo (manejar apuntadores a funciones con la misma interfaz)

La programación orientada a objetos simplemente trajo sintaxis e implementaciones seguras para estos “hacks”, algo similar se puede decir de la sintaxis y modelación de la concurrencia en Go

Otros técnicas concurrencia en Go

En la librería estándar de go, en el paquete **sync**, se encuentran varias implementaciones de técnicas de concurrencia:

- Mutex, RWMutex
  - Permite adquirir locks sobre recursos, tal que un solo proceso pueda acceder a la vez a este, en el caso del RWMutex, permite obtener varios locks de lectura **o (exclusivo)** un lock de escritura.
- WaitGroup
  - Permite bloquear la ejecución de una goroutine hasta que cierto contador atómico llegue a 0, útil para sincronización

Otros técnicas concurrencia en Go

En la librería estándar de go, en el paquete **sync**, se encuentran varias implementaciones de técnicas de concurrencia:

- Once
  - Permite asegurar que la clausura enviada se ejecute una y solo una vez, y termine antes de que cualquier llamada a **once.Do** devuelva
- Cond
  - Permite comunicación eficiente de señales entre goroutines: una goroutine, o un conjunto de goroutines puede entrar en un estado de bloqueo hasta que otras envíen una señal, liberándolas a todas o a una por una

Aspecto	Go (Canales)	C# (Channels)	C# (Semáforos)
Modelo de concurrencia	CSP (goroutines + canales nativos)	Productor-consumidor con `Channel<T>` (`async/await`)	Memoria compartida + sincronización manual
Facilidad de implementación	Alta (conciso, idiomático)	Media (requiere `ChannelReader/Writer`, `Task`)	Baja (mucho manejo manual de `Wait` y `Release`)
Sincronización	Implícita vía canales bloqueantes	Implícita, pero necesita más estructura	Explícita, requiere cuidado en cada acceso
Deadlock y race conditions	Menor riesgo (Go detecta deadlocks globales)	Posible, mitigable con buena gestión de tasks	Alto riesgo si no se previenen cuidadosamente
Escalabilidad	Muy buena (goroutines ligeras)	Buena (optimizado para concurrencia asincrónica)	Moderada (requiere control cuidadoso)
Código compartido	Evitado (comunicación por mensaje)	Evitado (similar al modelo CSP)	Necesario (sincronización manual obligatoria)
Ventaja principal	Simple, seguro y expresivo para concurrencia	Eficiente y desacoplado para flujos de datos	Gran control sobre sincronización y recursos
Desventaja principal	Modelo menos común fuera de Go	Requiere comprensión de async y tareas	Propenso a errores, verboso, difícil de depurar