**Go’da Concurrency’e Giriş**

Concurrency(eşzamanlılık), çoklu işlem yapısından xyararlanarak performansı arttırmayı sağlar. Go, API desteğiyle, programcılara paralel algoritmaları verimli bir şekilde implemente etmesine yardım eder. Günümüzdeki bir çok popüler yazılım dilleri concurrency i ek bir özellik gibi sunarken Go’da bu gömülü biçimde temel özellik olarak gelmektedir. Bu makaleyle Go’da concurrency e giriş anlatılmaya çalışılmıştır.

**Go’da Eş Zamanlı Programlama**

Eşzamanlı programlama, modern bilgisayarlarda bulunan çoklu işlem çekirdeğinden (Multiple Processing Cores) tam olarak yararlanmaktadır. Kullanılan mantık aslında uzun bir süredir tek çekirdekli işlemcilerde bir kullanılmaktaydı. C++, Java ve benzeri dillerde genellikle çoklu thread kullanılması concurrency nin uygulanmasını sağlıyordu.

Thread(iş parçacığı); basitçe anlatılması gerekirse, bağımsız biçimde yürütülmek üzere programlanmış küçük bir komut setini temsil eder. Kavramsal olarak bakarsak büyük bir işteki küçük iş parçacıkları olarak canlandırılabilir. Bu yüzden, çoklu thread, birden çok komplex iş parçacığını aynı anda yürütebilir. Ancak, tek çekirdekli işlemci gibi temelinde sınırlı bir donanımın işleri yalnızca zaman paylaşımlı bir şekilde planlayarak yapabileceğini unutulmamalıdır. Birden fazla görev arasındaki bu tutarlılık, concurrency hissi verir.

Bugünün işlemcileri birden çok çekirdeği içerisinde bulunmaktadır. Bu nedenle, bu işlem kapasitesini sonuna kadar kullanan diller daha çok rağbet görmektedir. Ana akım programlama dilleri yavaş yavaş bu gerçeği fark ediyor ve concurrency fikrini “temel kabiliyetleri” içinde benimsemeye çalışıyor. Bununla durumla birlikte, Go'nun tasarımcıları, neden temel kabiliyetleri içinde eşzamanlılık konseptine dayanan bir dil oluşturmadıklarını düşünüp harekete geçtiler. Go, eşzamanlı programlar yazmak için üst düzey API'ler sunan bir dil olarak karşımıza çıktı.

**Genel MultiThread Problemleri**

Multithreading, yazımı ve bakımı zor olduğu kadar debug yapılması da zor bir yapıdır. Ayrıca, performansı arttırmak için concurrency i sağlayan Multiple Thread kullanan bir algoritmayı parçalarına ayırmak ya da modüler olarak kullanmakta zordur. Öyledir ki Multi-Thread kullanmanın da kendi maliyeti bulunmaktadır. Multi Thread yapısındaki processler arası haberleşme, Shared-Memory yönetimi gibi bir çok sorumluluk Ortam uygulmasının yönetimindedir. Burada, yazılımcı yapmak istediği işe odaklansın, kalan paralel programlamaya ait alengirli işlerin detayına karışmasın gibi bir yapı kurulmuştur.

Tüm bu sıkıntılara alternatif bir çözüm olarak, multiprocessing e ait direksiyonu işletim sistemine bırakmak olabilir. Bu durumda Processler arası haberleşme ve Shared Memory yönetimi kullanıcıya kalır. Evet performansa yönelik ayarların yapılması kolaylaşsa da, bunların getirdiği karmaşanın arasında da kolayca kaybolunabilir.

**Eşzamanlı Programlamada Go’nun Avantajları**

Go, eşzamanlı programlama için 3 farklı çözüm sunmaktadır.

* Üst düzey destek, yalnızca eşzamanlılık elde etmeyi kolaylaştırmakla kalmaz, aynı zamanda bakımını da kolaylaştırır.
* Goroutine lerin kullanımı. Goroutine ler, threadlerden çok daha az maliyetlidir.
* Go'nun otomatik GarbageCollector özelliği, geliştiricilerin müdahalesi olmadan bellek yönetiminin karmaşıklığını ele alır.

**Go’da Concurrency Sorunlarının Yönetimi**

GoRoutine ler, concurrency yapımını kolaylaştırır. Burada yürütme faaliyetine GoRoutine denir. Şöyle örneklendirebiliriz; birbirini çağırmayan iki fonksiyon düşünelim. Sıralı çalıştırmada, bir fonksiyon çalışmasını bitirdiğinde ikinci çalışma çağrılmaktadır. Ancak Go’da iki fonksiyon aynı anda koşmaktadır. Bu fonksiyonlar birbiri ile ilişki halinde değilse kolay, ancak fonksiyonlar iç içe geçtiğinde ve birbirlerinin yürütme zamanlarını paylaştığında işler zorlaşmaktadır.

Go’nun üst düzey concurrency desteğiyle bile bazen hatalara düşüldüğü olabiliyor, özellikle main fonksiyonu kendisine bağlı fonksiyonlardan önce sonlanırsa. Bu yüzden Main GoRoutine, tüm işler tamamlana kadar sonlandırılmamalıdır.

Başka bir problem ise **DeadLock** dediğimiz durumdur. Burada birden fazla GoRouitine in belirli bir kaynağı ulaşımı kilitleyip başka bir GoRoutine in bu kaynakta değişiklik yapmaya çalışmasından kaynaklanmaktadır. Bu risk, eşzamanlı programlamada genel olsa da Go’ da bu tür durumlar için kanal ve bu kanalları kilitleme çözümleri vardır. Genel anlamda, iş yapılırken çalışmanın tamamlandığını bildiren bir kanal sinyali gönderilir. Başka bir yol da Senkronize WaitGroup raporlamasını beklemektir. Ancak her iki durumda da kilitlenme meydana gelebilmesi mümkündür. En iyisi, dikkatli bir tasarımla bunların önüne geçmektir. Basitçe Go, concurrency nin doğru işleyişini planlamak için araçlar sağlar

**WaitGroup ile GoRoutine Örneği**

Herhangi bir fonksiyon çağrısının önüne Go anahtar kelimesi konularak GoRoutine ler basitçe oluşturulabilir. Daha sonra Go kelimesi eklenen bu fonsiyon thread gibi davranmaya başlayacaktır. Ayrıca ormal fonksiyonlar gibi, herhangi bir argümana, global değişkene ya da erişebileceği her şeye erişebilir haldedir

Aşağıda girilen herhangi bir Websitesinin Çalışır durumda olup olmadığını kontrol eden bir kod bulunmaktadır. Sonrasında aynı koda GoRoutine uygulanmış halini görebilirsiniz. Concurrency uygulandığında çalışma zamanının nasıl daha kısaldığını gözlemlenebilir.

package main

import (

"fmt"

"net/http"

"time"

)

func main() {

start := time.Now()

sitelist := []string{

"https://www.google.com//",

"https://www.duckduckgo.com/",

"https://www.developer.com/",

"https://www.codeguru.com/",

"https://www.nasa.gov/",

"https://golang.com/",

}

for \_, site := range sitelist {

GetSiteStatus(site)

}

fmt.Printf("\n\nTime elapsed since %v\n\n", time.Since(start))

}

func GetSiteStatus(site string) {

if \_, err := http.Get(site); err != nil {

fmt.Printf("%s is down\n", site)

} else {

fmt.Printf("%s is up\n", site)

}

}

Yukarıdaki kodu çalıştırdığımızda aşağıda ki sonuçları elde etmekteyiz:

https://www.google.com// is up

https://www.duckduckgo.com/ is up

https://www.developer.com/ is up

https://www.codeguru.com/ is up

https://www.nasa.gov/ is up

https://golang.com/ is up

Time elapsed since 6.666198944s

Şimdi, WaitGroup senkronizasyon yapısı ile yukarıdaki kodda concurrency uygulanırsa, performansın gözle görülür bir şekilde arttığı gözlemlenecektir.

package main

import (

"fmt"

"net/http"

"sync"

"time"

)

func main() {

var wg sync.WaitGroup

start := time.Now()

sitelist := []string{

"https://www.google.com//",

"https://www.duckduckgo.com/",

"https://www.developer.com/",

"https://www.codeguru.com/",

"https://www.nasa.gov/",

"https://golang.com/",

}

for \_, site := range sitelist {

go GetSiteStatus(site, &wg)

wg.Add(1)

}

wg.Wait()

fmt.Printf("\n\nTime elapsed since %v\n\n", time.Since(start))

}

func GetSiteStatus(site string, wg \*sync.WaitGroup) {

defer wg.Done()

if \_, err := http.Get(site); err != nil {

fmt.Println("%s is down\n", site)

} else {

fmt.Printf("%s is up\n", site)

}

}

Yukarıdaki kodu çalıştırılıp tekrar Go programının çıktısına bakılırsa zaman iyileşmesi aşağıdaki gibi gözlemlenecektir:

https://www.nasa.gov/ is up

https://www.google.com// is up

https://www.developer.com/ is up

https://www.duckduckgo.com/ is up

https://golang.com/ is up

https://www.codeguru.com/ is up

Time elapsed since 1.816887681s

Buradaki WaitGroup bir senkronizasyon yapısıdır. Oluşturan her GoRoutine için WaitGroup’un wg.Add(1) kullanılarak arttırıldığını ve rutin tamamlandığında wg.Done() kullanılarak azaltıldığını kodta görebiliriz. Burada dikkat edilecek bir diğer nokta ise, Wg.Wait() main GoRoutine in, diğer tüm GoRoutine görevlerinin tamamlanmasının beklemesini sağlayan bir engelleyen bir komut olduğudur.

**Go’da Mutex ile Senkronizasyon**

Go dilinde paylaşılan kaynaklar üzerinde senkronizasyonu sağlamak için WaitGroup dışında, Mutex yapısı da kullanılmaktadır. Burada sync.Mutex.lock() kullanılarak GoRoutine lerin paylaşılan bir kaynağa ulaşması engellenmektedir. Benzer şekilde, sync.Mutex.Unlock() kullanılarak bu engel kaldırılabilmektedir.

Bir önceki kod üzerinde Mutex senkronizasyonu uygulamaları aşağıda görülebilir

//...

func main() {

var wg sync.WaitGroup

var mut sync.Mutex

//...

for \_, site := range sitelist {

go GetSiteStatus(site, &wg, &mut)

wg.Add(1)

}

//...

}

func GetSiteStatus(site string, wg \*sync.WaitGroup, mut \*sync.WaitGroup) {

defer wg.Done()

if \_, err := http.Get(site); err != nil {

fmt.Println("%s is down\n", site)

} else {

mut.Lock()

defer mut.Unlock()

fmt.Printf("%s is up\n", site)

}

}

**Golang’te Kanalların Uygulanması**

Kanallar, GoRoutine fonksiyonları arasındaki bağlantıdır. Burada kanal yapısı UNIX sistemdeki gönderme alma yapısı gibi, bir GoRoutine’den gönderilen değerin diğer GoRoutine den alınmasını sağlar. Bu nedenle bir kanalın gönderme ve alma olmak üzere iki temel işlevi vardır. Kanallar oluşturulma esnasında tiplerinin tanımlanması gerekmektedir. Örneğin, eleman tipi INT olan bir kanal şu şekilde yazılabilir:

cha := make(chan int)

Buradaki eleman türü, kanaldan geçirilecek değerin türünü belirler. Interface tipinde kanal belirleyerek, herhangi bir tipteki değerin iletilmesi ve alıcı tarafta bu türün belirlenmesi sağlanabilir.

Aynı türden bir kanal eşitlik operatörü(“==”) ile ve ya boş bir kanal “nil” ile karşılaştırılabilir.

Ayrıca kanal, arabellek boyutunun ayarlanmasını destekler.

Tüm bunlar dahilinde Go'da bir kanalın nasıl uygulanacağına dair hızlı bir örnek verirsek:

package main

import "fmt"

func main() {

odd := make(chan int)

oddsquared := make(chan int)

//Tek Sayılar

go func() {

for x := 1; x < 10; x++ {

if x%2 != 0 {

odd <- x

}

}

close(odd)

}()

// Tek Sayıların karesi

go func() {

for x := range odd {

oddsquared <- x \* x

}

close(oddsquared)

}()

for x := range oddsquared {

fmt.Println(x)

}

}

**Go Concurrency Programlama Üzerine Son Düşünceler**

Go, üst düzey concurrency desteğinin yanı sıra, sync/atomic paketi ile low-level işlevsellik de sağlar. Bu özelliği, genel olarak bakıldığında normal kullanıcılar tarafından pek kullanılmasa da temel olarak thread-safe senkronizasyonunu ve veri yapısı uygulamalarını gibi gelişmiş öğeleri destekler. Concurrency için çoğunlukla GoRoutine ve Kanal gibi HiLevel araçlar kullanılmaktadır. Go’daki bu üst düzey API lerinin desteği diğer ana akım programlama dillerinden daha kapsamlıdır. Bu durum Go tasarımcılarının concurrency desteğinin doğrudan temel özelliklere dahil etmek için sağlam çaba sarf ettiğini açıkça gösteriyor.