UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA PRIMENJENE RAČUNARSKE NAUKE

Paralelni i distribuirani algoritmi i strukture podataka

ms Nebojša Horvat

Zimski semestar 2019/2020.

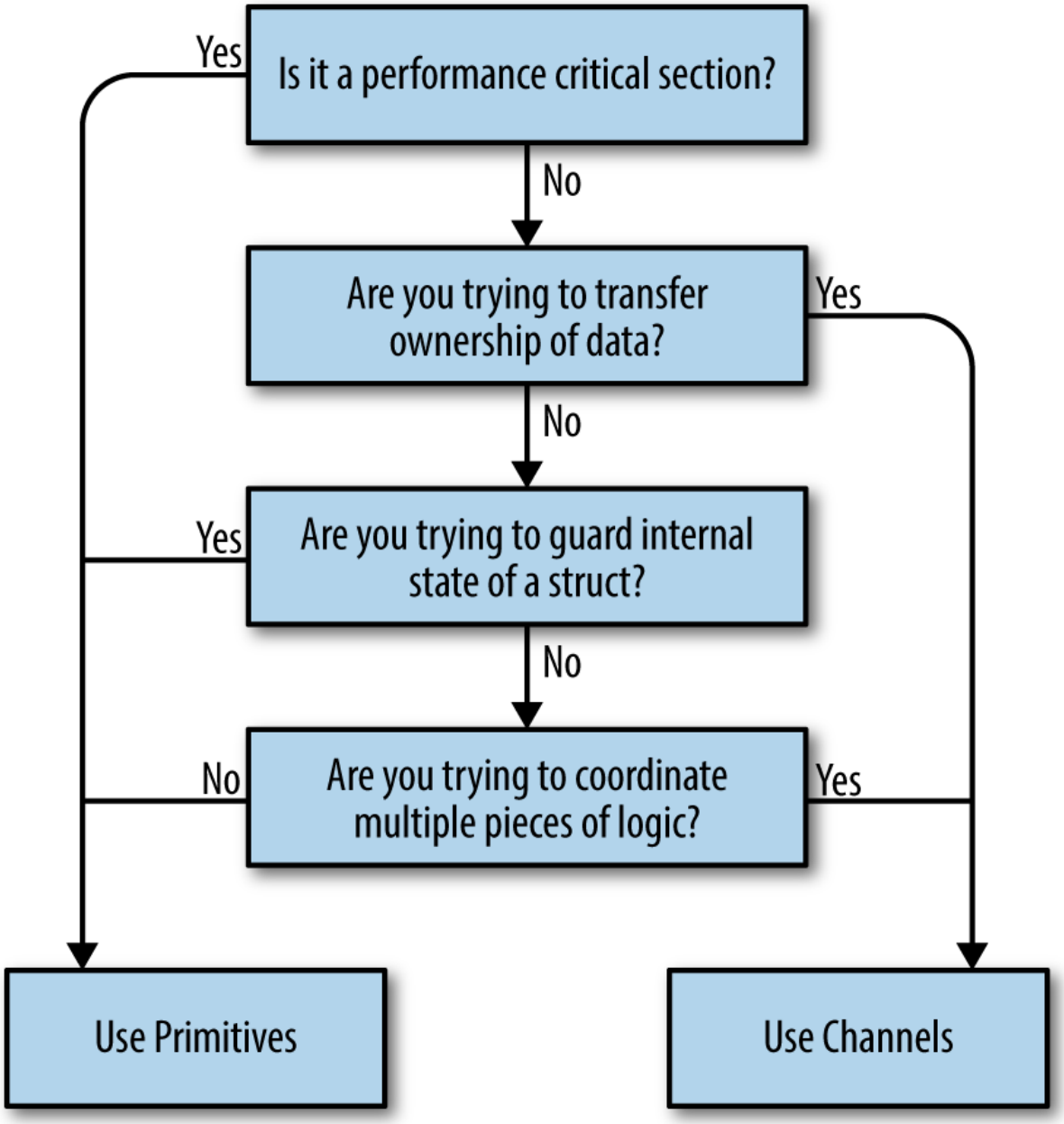
Studijski program: Računarstvo i automatika

Modul: Računarstvo visokih performansi

Go (Golang)

Filozofija go jezika

- Go je pretežno zasnovan na CSP-u (communicating sequential processes)
- Do not communicate by sharing memory. Instead, share memory by communicating.
- Go u sebi ima ugrađene i primitive za sinhronizaciju pristupa. Takve primitive su vama već poznati mutex itd.
- Iako Go ohrabruje programere da razmišljaju na drugačiji način, postoje problemi koji se mnogo efikasnije rešavaju preko primitiva te se one dodate kao osnovni deo jezika.


Is it a performance critical section?
Yes
No
Are you trying to transfer ownership of data?
Yes
No
Are you trying to guard internal state of a struct?
Yes
No
Are you trying to coordinate multiple pieces of logic?
No
Yes
Use Primitives
Use Channels

Primitive ili kanali

- **Prebacivanje vlasništva nad podacima** Kada jedan deo koda proizvodi neke podatke i deli ih sa drugim delom koda, to je zapravo prenošenje vlasništva nad podacima.

- Kanali su idealni za navedenu upotrebu. Kanali sa baferom su posebno pogodni za transfer podataka pošto oni predstavljaju **red** koji odvaja proizvođača podataka od korisnika.

Primitive ili kanali

- **Očuvanje internog stanja strukture** je odličan primer u kome je idealno koristiti primitive za sinhronizaciju, i primer na kome se ne preporučuje upotreba kanala
- Ključna reč je očuvanje **internog** stanja. Nije poželjno koristiti primitive poput mutex-a za sinhronizaciju pristupa memoriji izvan strukture.

```
type Counter struct {
    mu sync.Mutex
    value int
}
func (c *Counter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}
```

Primitive ili kanali

- **Koordinacija različitih delova programa.** Kod u kome su mutex-i razbacani na sve strane izgleda haotično. Dok je sa druge strane razbacivanje kanala po kodu poželjno i preporučeno.
- Kompleksnost konkurentnog koda, koja se javlja sa povećanjem količine koda se može umanjiti kroz upotrebu kanala i select naredbe.
- Ukoliko imate problema da razumete sami svoj konkurentni kod, a pritom koristite primitive za sinhronizaciju, onda je to verovatno dobar pokazatelj da treba da pređete na upotrebu kanala.

Primitive ili kanali

- Ukoliko su **performanse od krucijalnog značaja** preporučuje se upotreba primitiva za sinhronizaciju.
- To ne znači da ukoliko želimo da naš kod radi brzo, onda ne treba da koristimo kanale. Već ukoliko profajler kaže da je neka sekcija koda kritična i da je ključno da se ona ubrza, onda treba maksimalno koristiti primitive za sinhronizaciju i izbegavati kanale.

Sync paket

- Sync paket se koristi za rad sa primitivama za sinhronizaciju pristupa.
- Funkcije koje se nalaze u sync paketu su dobro poznate ljudima koji su radili sa drugim programskim jezicima koji pružaju sinhronizaciju pristupa memoriji.

WaitGroup

- WaitGroup je odličan način da se sačeka izvršavanje grupe operacija kada nas ne zanima rezultat operacija ili postoji drugi način da prikupimo rezultate operacija.
- Ukoliko neka od prethodno pomenutih pretpostavki nije slučaj, predlaže se upotreba kanala i select naredbe.

```
var wg sync.WaitGroup
wg.Add(1)
go func() {
    defer wg.Done()
    fmt.Println("1st goroutine sleeping...")
    time.Sleep(1)
}()
wg.Wait()
fmt.Println("All goroutines complete.")
```

Mutex

- Mutex (mutual exclusion) se koristi za osiguravanje dela koda, tako da ga u jednom trenutku može izvršavati samo jedna go rutina.

```
lock.Lock()
defer lock.Unlock()
count--
```

- Česta praksa je da se Unlock radi preko defer kako bi se osiguralo da će se otključavanje izvršiti iako dođe do panike

Zadatak

- Napraviti strukturu koja u sebi sadrži brojač. Istovremeno pokrenuti 4 go rutine koje povećavaju brojač. U trenutku kada brojač dostigne 100 ispisati njegovu vrednost i završiti program.
- Za realizaciju koristiti samo primitive za sinhronizaciju

Once

- Once samo vodi računa o tome koliko puta je **Do** pozvano. Ne koliko puta se **Do** poziva sa dragačijom funkcijom!

```
var count int
increment := func() {
        count++
}

var once sync.Once
var increments sync.WaitGroup
increments.Add(100)
for i := 0; i < 100; i++ {
    go func() {
       defer increments.Done()
       once.Do(increment)
    }()
}
increments.Wait()
fmt.Printf("Count is %d\n", count)
```

Pool

- Pool je konkuretno bezbedna implementacija objekt pool šablona.
- Gledano sa visokog nivao apstrakcije, bazen (engl. Pool) nam omogućava kreiranje i prosleđivanje fiksnog broja elemenata koji se mogu upotrebiti.
- Najčešće se koristi kako bi se ograničila proizvodnja stvari koje su “skupe” kao što su konekcija na bazu itd.

```
myPool := &sync.Pool{
   New: func() interface{} {
      fmt.Println("Creating new instance.")
      return struct{}{}
},
}
myPool.Get()
instance := myPool.Get()
myPool.Put(instance)
myPool.Get()
```

Creating new instance.
Creating new instance.

Kanali (nastavak)

Zadatak

- Naparaviti strukturu koja u sebi sadrži brojač. Istovremeno pokrenuti 4 go rutine koje povećavaju brojač. U trenutku kada brojač dostigne 100 ispisati njegovu vrednost i završiti program.
- Za sinhronizaciju niti koristiti kanale i mutex

Jednosmerni kanali

- Prošli čas smo pričali o osnovnim funkcionalnostima kanala, a sada ćemo proširiti priču
- Moguće je definisati jednosmerne kanale, tj. kanale iz kojih se može samo čitati ili pisati.
- Takvi kanali su korisni kao ulaz ili izlaz iz funkcija.
- Konverzija iz dvosmernog u jednosmerni kanala se implicitno radi.

```
dataStream := make(<-chan interface{})
dataStream := make(chan<- interface{})
```

Konverzija jednosmernih kanala

```
var receiveChan <-chan interface{}
var sendChan chan<- interface{}
dataStream := make(chan interface{})

// Valid statements:
receiveChan = dataStream
sendChan = dataStream
```

Zatvaranje kanala

- Kanale je moguće zatvoriti i na taj način signalizirati onima koji čiraju iz kanala da neće biti više upisivanja.
- Bitno je da kanal zatvara samo onaj ko u njega upisuje. Pisanje u zatvoren kanal rezultuje greškom.

```
integer, ok := <- intStream
```

- Prethodna komanda isčitava element iz kanala i vraća true ukoliko je kanal otvoren ili vraća default vrednost za taj tip i false ukoliko je kanal zatvoren

Zatvaranje kanala

- Mogućnost čitanja iz kanala uz saznanje da li je kanal zatvoren, otvara par novih šablona koji se mogu često koristiti.

```
intStream := make(chan int)
go func() {
    defer close(intStream)
    for i := 1; i <= 5; i++ {
        intStream <- i
    }
}()
for integer := range intStream {
     fmt.Printf("%v ", integer)
}
```

Zatvaranje kanala

- Ukoliko više go rutina čeka na isčitavanje iz nekog kanala mnogo je efikasnije i brže zatvoriti taj kanal i na taj način signalizirati svim rutinama da nastave sa izvršavanjem. To je bolji način od ubacivanja vrednosti u kanal onoliko puta koliko ima rutina koje čekaju.

```
begin := make(chan interface{})
var wg sync.WaitGroup
for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(i int) {
            defer wg.Done()
            <-begin
            fmt.Printf("%v has begun\n", i)
        }(i)
}
fmt.Println("Unblocking goroutines...")
close(begin)
wg.Wait()
```

Vlasništvo kanala

- Kako bi se izbegle nepoželjne situacije pri radu sa kanalima dobra praksa je dodeliti rutini vlasništvo nad kanalom. To se radi kroz prosleđivanje jednosmernih kanala rutinama. Tako da je vlasnik kanala zadužen za:
  - Pravljenje kanala
  - Pisanje u kanal ili prosleđivanje vlasništva drugom kanalu
  - Zatvaranje kanala
- Korisnik kanala u tom slučaju brine o samo dve stvari:
  - Praćenje otvorenosti kanala
  - Odgovono rukovanje blokiranjem pri čitanju iz kanala

Vlasništvo kanala

```
chanOwner := func() <-chan int {
resultStream := make(chan int, 5)
go func() {
      defer close(resultStream)
      for i := 0; i <= 5; i++ {
          resultStream <- i
      }
}()
return resultStream
}
resultStream := chanOwner()
for result := range resultStream {
      fmt.Printf("Received: %d\n", result)
}
fmt.Println("Done receiving!")
```

Select

“The select statement is the glue that binds channels together”

```
var c1, c2 <-chan interface{}
var c3 chan<- interface{}
select {
	case <- c1:
		// Do something
	case <- c2:
		// Do something
	case c3<- struct{}{}:
		// Do something
}
```

Select

- Podseća na switch
- Za razliku od switch case-a, iskazi u select bloku se ne testiraju sekvencijalno
- Ukoliko nijedan od kriterijuma nije zadovoljen ne izlazi se iz komande kao u slučaju kod switch naredbe, već se čeka da jedan od iskaza bude ispunjen
- Ukoliko više iskaza moze istovremeno da se izvrši, izvršiće se u proizvoljnom redosledu

Select

```
done := make(chan interface{})
go func() {
	time.Sleep(5*time.Second)
	close(done)
}()
workCounter := 0
loop:
for {
	select {
		case <-done:
			break loop
		 default:
	}
	// Simulate work
	workCounter++
	time.Sleep(1*time.Second)
}
fmt.Printf("Achieved %v cycles of work before signalled to stop.\n", workCounter)
```

Achieved 5 cycles of work before signalled to stop.

Zadatak select

- Ispisati N elemenata fibonačijevog niza.
- Jedna go rutina računa elemente, dok druga istovremeno ispisuje elemente niza na konzoli.
- Za realizaciju zadatka koristiti select i kanale

Šabloni za pisanje konkuretnog koda

Zatvaranje koda

- U radu sa paralelnim kodom najbolje je izbeći greške kroz:
  - Upotrebu nepromenljive (*eng. immutable*) memorije koja je kao takva bezbedna (*eng.thread safe*)
  - Zaštiti memoriju kroz zatvaranje

Zatvaranje

- Kroz zatvaranje se postiže smanjenje mentalnog napora programera i pravljenje manjih kritičnih sekcija.
- Zatvaranje je prosta i moćna tehnika koja osigurava da su informacije dostupne samo jednoj niti u jednom trenutku
- Zatvaranje se može postići:
  - ad hock
  - leksičko

ad hock zatvaranje

- Ad hock zatvaranje se postiže kroz konvenciju
- U velikim projektima poštovanje konvencije postaje značajan proble.

```
data := make([]int, 4)
loopData := func(handleData chan<- int) {
    defer close(handleData)
    for i := range data {
        handleData <- data[i]
    }
}
handleData := make(chan int)
go loopData(handleData)
for num := range handleData {
    fmt.Println(num)
}
```

Leksičko zatvaranje

- Leksičko zatvaranje koristi leksički domet koji otkriva podatke i primitive samo određenim paralelnim prosecima.

```
chanOwner := func() <-chan int {
    results := make(chan int, 5)
    go func() {
        defer close(results)
        for i := 0; i <= 5; i++ {
            results <- i
        }
    }()
    return results
}
consumer := func(results <-chan int) {
    for result := range results {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}
results := chanOwner()
consumer(results)
```

Leksičko zatvaranje

- Kako su kanali već thread safe prethodni primer možda nema najbolji
- U primer ispod podelimo podatke na dva slajsa tako da zbog leksičkih dohvata funkcija možemo biti sigurni da neće doći do konflikta i da nema potrebe za sinhronizacijom pri pristupu memoriji

```
printData := func(wg *sync.WaitGroup, data []byte) {
    defer wg.Done()
    var buff bytes.Buffer
    for _, b := range data {
        fmt.Fprintf(&buff, "%c", b)
    }
    fmt.Println(buff.String())
}
var wg sync.WaitGroup
wg.Add(2)
data := []byte("golang")
go printData(&wg, data[:3])
go printData(&wg, data[3:])
wg.Wait()
```

For-select loop

- For-select petlja je nešto što se veoma često viđa u go kodu
- Prateći kod se koristi kako bi se elementi nekog niza upisali u kanal jedan po jedan

```
for _, s := range []string{"a", "b", "c"} {
    select {
        case <-done:
            return
        case stringStream <- s:
    }
}
```

Curenje rutina

- Iako su go rutine lagane može doći do njihovog “curenja” isto kao što može doći do “curenja” memorije.
- Neke go rutine mogu da se pokrenu i da se ne završe sve dok se ne završi program tj. main rutina. Kod nekih programa to može dugo da traje!

```
doWork := func(strings <-chan string) <-chan interface{} {
    completed := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(completed)
            for s := range strings {
                // Do something interesting
                fmt.Println(s)
            }
    }()
    return completed
}
doWork(nil)
// Perhaps more work is done here
fmt.Println("Done.")
```

Curenje rutina

- Način na koji se curenje može rešiti je da roditelj rutina signalizira detetu da se ugasi.

```
doWork := func( done <-chan interface{}, strings <-chan string ) <-chan interface{} {
   terminated := make(chan interface{})
   go func() {
      defer fmt.Println("doWork exited.")
      defer close(terminated)
      for {
         select {
            case s := ←strings:
               // Do something interesting
               fmt.Println(s)
            case ←done:                          // Korišćenje for-select šablona kako bi se prekinuo rad rutine
               return
          }
       }
    }()
return terminated
}
done := make(chan interface{})
terminated := doWork(done, nil)
go func() {                                      // Pokretanje rutine koja će zaustaviti doWork rutinu
   // Cancel the operation after 1 second.
   time.Sleep(1 * time.Second)
   fmt.Println("Canceling doWork goroutine...")
   close(done)
}()
←terminated                                      // Spajanje (engl. join) main rutine I njege dete rutine woWork
fmt.Println("Done.")
```

Rukovanje greškama

- Nije poželjno pustiti go rutinu da sama brine i obrađuje greške
- Poželjno je vratiti error roditeljskoj rutini pošto ona ima širu sliku o tome šta se desava u programu
- U sledećem primeru sve greške se vraćaju glavnoj go rutini koja onda odlučuje šta će raditi sa njima
- Ukoliko je broj grešaka veći od npr. Tri, može se završiti program

Rukovanje greškama

```
type Result struct {
   Error error
   Response *http.Response
}
checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result {
   results := make(chan Result)
   go func() {
      defer close(results)
      for _, url := range urls {
         var result Result
         resp, err := http.Get(url)
         result = Result{Error: err, Response: resp}
         Select {
            case <-done:
               return
            case results <- result:
         }
      }
   }()
return results
}
done := make(chan interface{})
defer close(done)
urls := []string{"https://www.google.com", "https://badhost"}
for result := range checkStatus(done, urls...) {
   if result.Error != nil {
      fmt.Printf("error: %v", result.Error)
      continue
   }
   fmt.Printf("Response: %v\n", result.Response.Status)
}
```

Problem pet filozofa

- Filozofi obavljaju samo dve različite aktivnosti: misle ili jedu
- Na jednom okruglom stolu nalazi se pet tanjira i pet viljušaka (između svaka dva tanjira po jedan)
- Filozof prilazi stolu, uzima levu viljušku, pa desnu viljušku i nakon toga jede
- Zatim vraća viljuške na sto i nastavlja da misli

Problem pet filozofa