Concorrência TPC1: Filósofos Jantando

Grupo: José Junior Borges Monteiro e Loannis Lucas Nicou Canteiro

Introdução

Este trabalho tem como objetivo mostrar duas possíveis soluções para o problema Filósofos Jantando.

O jantar dos filósofos foi proposto por Dijkstra em 1965 como um problema de sincronização. O desafio traz um cenário onde cinco filósofos estão sentado em uma mesa redonda, onde cada filósofo tem um prato com espaguete e cada prato possui um garfo para pegar o espaguete, porém, para que um filósofo consiga comer é necessário utilizar dois garfos, causando uma competição pelo recurso garfo. Os filósofos alternam entre pensar e comer(pegando os dois garfos), o desafio está em sincronizar as ações para que um filósofo consiga pegar os dois garfos mantendo o máximo de paralelismo.

Nesta situação nos deparamos com dois problemas causados pela concorrência:

Deadlock: que irá ocorrer quando todos os filósofos pegarem o garfo esquerdo ficando impossibilitados de pegarem o garfo da direita.

Starvation: poderá ocorrer caso na solução um dos filósofos fique tentando acessar o recurso indeterminadamente sem sucesso, impossibilitando este filósofo de comer.

Soluções

A primeira solução nós optamos por utilizar canais para realizar a sincronização entre os processos, sendo possível bloquear o filósofo quando o mesmo tentar pegar um garfo que já está sendo utilizado por outro filósofo.

Com esta implementação de sincronização pode haver problema de deadlock como explicado anteriormente, para resolver esta questão nós utilizamos a solução proposta pelo Dijkstra que trata o problema de deadlock quebrando o ciclo no último filósofo. Para isso, é realizada uma modelagem onde os filósofos pegam primeiro o garfo esquerdo e depois o direito, e apenas o último filósofo pega o garfo direito e depois o esquerdo. Com isso, caso todos os filósofos façam a ação de pegar um garfo quando chegar no ultimo ele tentará pegar o garfo direito que já foi pego pelo primeiro filósofo e ficará esperando o garfo ficar disponível, permitindo que o quarto filósofo pegue o garfo direito e assim destravando os demais filósofos.

Resolvido este impasse, é necessário também prevenir contra starvation, para isso utilizamos um tempo aleatório que vai de 5 a 15 segundos na função de pensar, isso faz com que diminua a concorrência entre os processos possibilitando que todos tenham acesso ao recurso.

A segunda solução encontrada foi a do Tanenbaum, que utiliza estados para identificar se um filósofo está comendo, pensando ou faminto (pensando em pegar os garfos), também é utilizado um arranjo de semáforos, sendo um por filósofo. Ao associar um semáforo por filósofo, permitimos suspender o processo caso não consiga os dois garfos e fique esperando ser acordado por outros filósofos a fim de evitar problemas de deadlock e starvation. Também foi utilizado um semáforo para o cenário geral sendo comum a todos os processos para tornar atômico seções críticas, elas se encontram nas funções pegar garfo e largar garfo.

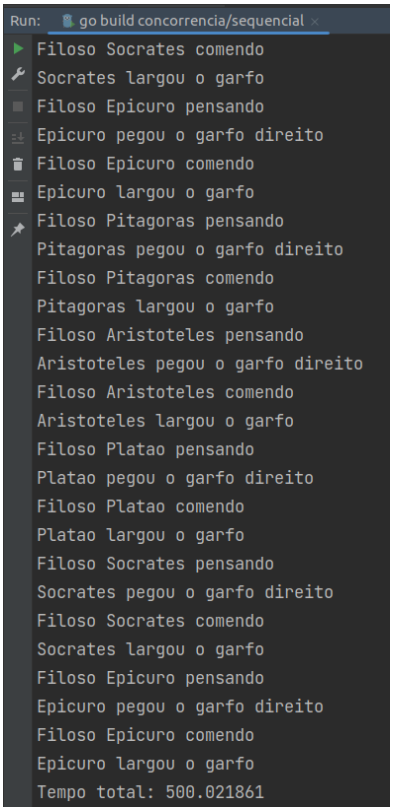
Na linguagem em que implementamos, GoLang, os semáforos são representados pela biblioteca sync.Mutex, utilizando mutex.Lock para trancar um processo e mutex.Unlock para destrancar o processo. Ademais, vale ressaltar que na solução foi utilizado semáforos binários.

Conclusão

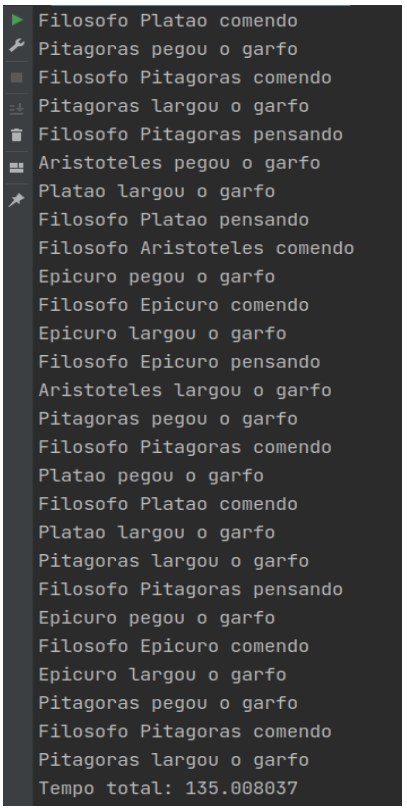
Ao realizar os testes podemos notar que o ganho em performance nas soluções concorrentes foi significativo, trazendo uma diferença de 6 minutos a menos para ambas versões concorrentes em relação a versão sequencial.

Com estes resultados, fica evidente as vantagens de se utilizar implementações concorrentes, mas também, como discutido no decorrer do artigo, fica evidente o aumento no nível de complexidade na implementação, exigindo do desenvolvedor uma atenção redobrada para as possíveis complicações que esta decisão pode trazer.

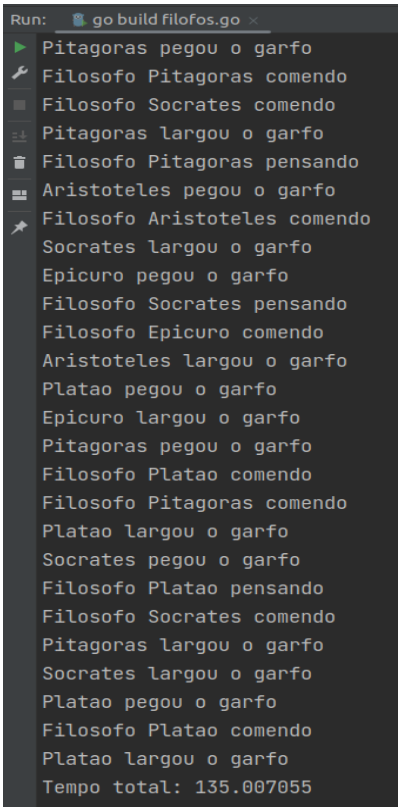
Versão sequencial



Solução Dijkstra



Solução Tanenbaum (Estados)



Versão Sequencial

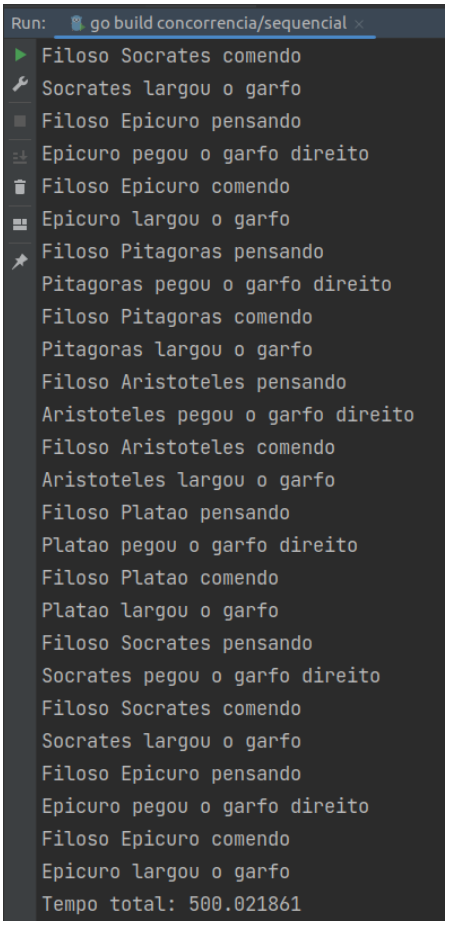
Para que fosse possível avaliar o desempenho das implementações concorrentes foi desenvolvido uma versão sequencial possibilitando realizar a comparação entre versões

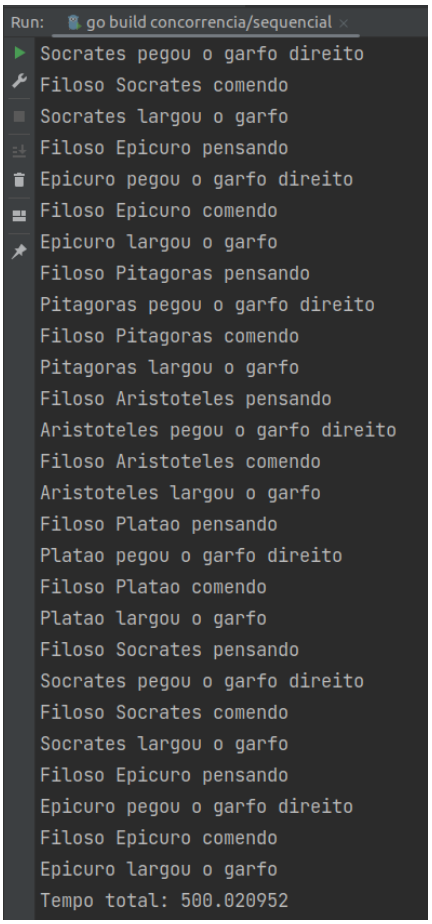
```
33      filosofos[0] = pitagoras
34      filosofos[1] = aristoteles
35      filosofos[2] = platao
36      filosofos[3] = socrates
37      filosofos[4] = epicuro
38
39      garfos[0] = Garfo{Posicao: 1}
40      garfos[1] = Garfo{Posicao: 2}
41      garfos[2] = Garfo{Posicao: 3}
42      garfos[3] = Garfo{Posicao: 4}
43      garfos[4] = Garfo{Posicao: 5}
44
45      var (
46          start time.Time
47          total time.Duration
48      )
49      start = time.Now()
50
51      for i := 0; i < 10; i++ {
52          jantar(&filosofos[0])
53          jantar(&filosofos[1])
54          jantar(&filosofos[2])
55          jantar(&filosofos[3])
56          jantar(&filosofos[4])
57      }
58      total = time.Since(start)
59      fmt.Printf( format: "Tempo total: %f", total.Seconds())
```

Abaixo estão as funções utilizadas para representar o emprego dos garfos a cada filósofo.

```
62  func jantar(filosofo *Filosofo) {
63      filosofo.pensar()
64      filosofo.pegarGarfo()
65      filosofo.comer()
66      filosofo.largarGarfo()
67  }
68
69  func (f *Filosofo) pegarGarfo() {
70      var garfoEsq = &garfos[f.Posicao-1]
71      var garfoDir = &garfos[(f.Posicao)%n]
72      f.GarfoEsq = garfoEsq
73      f.GarfoDir = garfoDir
74      fmt.Println(f.Nome + " pegou o garfo direito")
75  }
76
77  func (f *Filosofo) largarGarfo() {
78      f.GarfoEsq = nil
79      f.GarfoDir = nil
80      fmt.Println(f.Nome + " largou o garfo")
81  }
82
83  func (f *Filosofo) pensar() {
84      fmt.Println( a...: "Filoso " + f.Nome + " pensando")
85      time.Sleep(5 * time.Second)
86  }
87
88  func (f *Filosofo) comer() {
89      fmt.Println( a...: "Filoso " + f.Nome + " comendo")
90      time.Sleep(5 * time.Second)
91  }
```

Abaixo estão os resultados em milissegundos do tempo de processamento da aplicação sequencial, resultando em 500s que transformando em minutos fica próximo a 8 minutos de execução.





Versão Concorrente - Dijkstra

As estruturas utilizadas para a modelagem desta solução podem ser visualizadas na imagem abaixo. Nela podemos notar a utilização de canais necessários para realizar o controle de posse dos garfos.

```
26
27      garfos[0] = make(chan bool)
28      garfos[1] = make(chan bool)
29      garfos[2] = make(chan bool)
30      garfos[3] = make(chan bool)
31      garfos[4] = make(chan bool)
32
33      pitagoras := Filosofo{Posicao: 1, Nome: "Pitagoras"}
34      aristoteles := Filosofo{Posicao: 2, Nome: "Aristoteles"}
35      platao := Filosofo{Posicao: 3, Nome: "Platao"}
36      socrates := Filosofo{Posicao: 4, Nome: "Socrates"}
37      epicuro := Filosofo{Posicao: 5, Nome: "Epicuro"}
38
39      filosofos[0] = pitagoras
40      filosofos[1] = aristoteles
41      filosofos[2] = platao
42      filosofos[3] = socrates
43      filosofos[4] = epicuro
44
45      // aloca os canais garfos aos respectivos filosofos
46      for i := 0; i < n; i++ {
47          var garfoEsq = &garfos[i]
48          var garfoDir = &garfos[(i+1)%n]
49
50          filosofos[i].GarfoEsq = *garfoEsq
51          filosofos[i].GarfoDir = *garfoDir
52      }
53
```

Roda as threads de cada filósofo e os canais de cada garfo que realiza o controle da posse do mesmo.

```
59      start = time.Now()
60      waitGroup.Add( delta: 5)
61
62      go channelGarfos(garfos[0])
63      go channelGarfos(garfos[1])
64      go channelGarfos(garfos[2])
65      go channelGarfos(garfos[3])
66      go channelGarfos(garfos[4])
67
68      go jantar(&filosofos[0])
69      go jantar(&filosofos[1])
70      go jantar(&filosofos[2])
71      go jantar(&filosofos[3])
72      go jantar(&filosofos[4])
73
74      waitGroup.Wait()
75      total = time.Since(start)
76      fmt.Printf("Tempo total: #{total.Seconds()}")
77  }
```

Principais funções utilizadas na solução responsáveis por gerenciar a posse do garfo, na função pegarGarfo é utilizado a solução do dijkstra que alterna a pegada do primeiro garfo para o lado direito quando for o quinto filósofo resolvendo o problema de deadlock.

```go
func jantar(filosofo *Filosofo) {
    for i := 0; i < 10; i++ {
        filosofo.pensar()
        filosofo.pegarGarfo()
        filosofo.comer()
        filosofo.largarGarfo()
    }
    waitGroup.Done()
}

func (f *Filosofo) pegarGarfo() {
    // para evitar o deadlock foi adicionado uma condicao para que o ultimo filosofo pege o garfo direito primeiro
    // assim ele ficara disputando com o primeiro filosofo e o quarto filosofo poderá pegar o garfo a direita
    // quebrando o ciclo
    if f.Posicao == 5 {
        <-f.GarfoDir
        <-f.GarfoEsq
    } else {
        <-f.GarfoEsq
        <-f.GarfoDir
    }
    fmt.Println(f.Nome + " pegou o garfo ")
}

func (f *Filosofo) largarGarfo() {
    // destrava os processos lendo do canal
    f.GarfoEsq <- true
    f.GarfoDir <- true
    fmt.Println(f.Nome + " largou o garfo ")
}
```

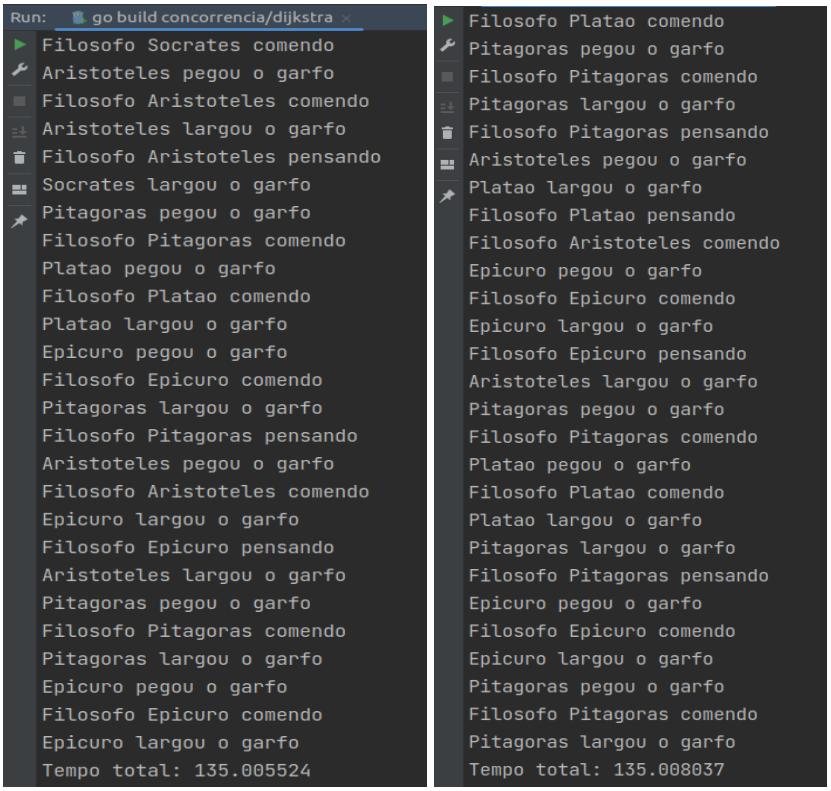
Na próxima imagem é exposto duas funções simples que imprime as ações que estão sendo realizadas pelo filósofo e aguarda por um determinado tempo. Na função comer foi utilizado um tempo aleatório que vai de 5 a 15 segundos a fim de prevenir contra o problema de starvation.
A função channelGarfos é responsável por bloquear e liberar os garfos para os filósofos.

```go
func (f *Filosofo) pensar() {
    fmt.Println( a...: "Filosofo " + f.Nome + " pensando")
    random := rand.Intn( n: 10) + 5
    time.Sleep(time.Duration(random) * time.Microsecond)
}

func (f *Filosofo) comer() {
    fmt.Println( a...: "Filosofo " + f.Nome + " comendo")
    time.Sleep(5 * time.Second)
}

func channelGarfos(c chan bool) {
    for true {
        c <- true
        <-c
    }
}
```

Abaixo estão os resultados em milissegundos do tempo de processamento da aplicação concorrente, resultando em 135s que transformando em minutos fica próximo a 2 minutos de execução.



Versão Concorrente - Tanenbaum(estados)

As estruturas utilizadas para a modelagem desta solução podem ser visualizadas na imagem abaixo. Nela podemos notar a utilização de const para representar os estados possíveis de um filósofo(Pensando, Faminto ou Comendo), o array de semáforo sendo um para cada filósofo, um semáforo para o cenário geral e outro array para guardar cada filósofo.

```
9   const n = 5
10  const (
11      PENSANDO = 0
12      FAMINTO  = 1
13      COMENDO  = 2
14  )
15
16  var (
17      waitGroup sync.WaitGroup
18      mutex     sync.Mutex
19      semaforo  [n]sync.Mutex
20      filosofos [n]Filosofo
21  )
22
23  type Filosofo struct {
24      Posicao int
25      Nome    string
26      Estado  int
27  }
```

A seguir é realizado a inicialização das variáveis necessárias

```go
func main() {
    pitagoras := Filosofo{Posicao: 0, Nome: "Pitagoras", Estado: PENSANDO}
    aristoteles := Filosofo{Posicao: 1, Nome: "Aristoteles", Estado: PENSANDO}
    platao := Filosofo{Posicao: 2, Nome: "Platao", Estado: PENSANDO}
    socrates := Filosofo{Posicao: 3, Nome: "Socrates", Estado: PENSANDO}
    epicuro := Filosofo{Posicao: 4, Nome: "Epicuro", Estado: PENSANDO}

    filosofos[0] = pitagoras
    filosofos[1] = aristoteles
    filosofos[2] = platao
    filosofos[3] = socrates
    filosofos[4] = epicuro

    for i := 0; i < n; i++ {
        semaforo[i].Lock()
    }

    var (
        start time.Time
        total time.Duration
    )
    start = time.Now()

    waitGroup.Add(n)
```

O próximo passo é iniciar os processos para cada filósofo

```go
    waitGroup.Add(n)

    go jantar(&filosofos[0])
    go jantar(&filosofos[1])
    go jantar(&filosofos[2])
    go jantar(&filosofos[3])
    go jantar(&filosofos[4])

    waitGroup.Wait()

    total = time.Since(start)
    fmt.Printf( format: "Tempo total: %f", total.Seconds())
}

func jantar(filosofo *Filosofo) {
    for i := 0; i < 10; i++ {
        filosofo.pensar()
        filosofo.pegarGarfo()
        filosofo.comer()
        filosofo.largarGarfo()
    }
    waitGroup.Done()
}
```

Na imagem abaixo é possível observar as duas principais funções da solução, nele é interessante destacar o uso de semáforo para tornar as seções críticas atômicas, além da utilização da função testa, necessário para realizar o controle da posse dos garfos.

```
76  func (f *Filosofo) pegarGarfo() {
77      mutex.Lock()
78      f.Estado = FAMINTO
79      testa(f)
80      mutex.Unlock()
81      semaforo[f.Posicao].Lock()
82  }
83
84  func (f *Filosofo) largarGarfo() {
85      var filosofoEsq = &filosofos[(f.Posicao+n-1)%n]
86      var filosofoDir = &filosofos[(f.Posicao+1)%n]
87      mutex.Lock()
88      f.Estado = PENSANDO
89      fmt.Println(f.Nome + " largou o garfo ")
90
91      testa(filosofoEsq)
92      testa(filosofoDir)
93
94      mutex.Unlock()
95  }
```

A função testa irá verificar se os filósofos a esquerda e a direita não estão comendo, assim sendo, é permitido que o filósofo pegue o garfo caso esteja faminto.

```go
func testa(f *Filosofo) {
    // busca filosofos ao lado
    var filosofoEsq = &filosofos[(f.Posicao+n-1)%n]
    var filosofoDir = &filosofos[(f.Posicao+1)%n]

    if f.Estado == FAMINTO && filosofoEsq.Estado != COMENDO && filosofoDir.Estado != COMENDO {
        f.Estado = COMENDO
        fmt.Println(f.Nome + " pegou o garfo ")
        semaforo[f.Posicao].Unlock()
    }
}

func (f *Filosofo) pensar() {
    fmt.Println( a...: "Filosofo " + f.Nome + " pensando")
    time.Sleep(5 * time.Second)
}

func (f *Filosofo) comer() {
    fmt.Println( a...: "Filosofo " + f.Nome + " comendo")
    time.Sleep(5 * time.Second)
}
```

Abaixo estão os resultados em milissegundos do tempo de processamento da segunda solução concorrente, resultando em um valor médio de 140s que transformando em minutos fica próximo a 2 minutos de execução, resultado similar a primeira solução concorrente.