Kaue Soares da Silveira
171671

Trabalho Opcional 1: Estudo da Linguagem Golang

Disciplina INF01151 - Sistemas Operacionais II N

Professor: Alexandre Carissimi

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Sumário

1 Introducao

"Não comunique-se compartilhando memória, compartilhe memória comunicando-se."

O simples ato de comunicação garante a sincronização.

Concorrência em muitos ambientes se torna difícil pelas sutilezas envolvidas no acesso correto a variáveis compartilhadas. Go[1] encoraja uma abordagem diferente, na qual variáveis compartilhadas são passadas através de canais e nunca são de fato ativamente compartilhadas por threads de execução separadas. Apenas uma gorotina tem acesso à variável num dado momento de tempo. Condições de corrida relativas aos dados não podem ocorrer, por construção. Modelo de concorrência inspirado no CSP [2].

2 *Primeiros Passos*

2.1 Instalação

Passos que eu segui no Ubuntu 9.10:

```
cd ~
gedit .bashrc &
  # colar no final:
  export GOROOT=$HOME/go
  export GOARCH=386
  export GOOS=linux
# reiniciar o shell para que as alterações tenham efeito
sudo apt-get install bison gcc libc6-dev ed gawk make
sudo apt-get install python-setuptools python-dev build-essential gcc
sudo easy_install mercurial
hg clone -r release https://go.googlecode.com/hg/ $GOROOT
cd $GOROOT/src
mkdir $HOME/bin
./all.bash
cd $HOME/bin
sudo mv * /bin
```

2.2 Utilização

```
# criar o arquivo fonte num editor de sua preferência
gedit prog.go & # -> prog.go
# compilar
8g prog.go # -> prog.8
# ligar
8l -o prog prog.8 # -> prog
# executar
./prog
```

3 *Mecanismos de Programação Concorrente*

3.1 Gorotinas

Go tem seu próprio modelo de processo / thread / corotina, chamado gorotina (*goroutines*). Gorotinas são divididas de acordo com o necessário em threads (de usuário) e processos do sistema. Quando uma goroutine executa uma chamada de sistema bloqueane, nenhuma outra gorotina é bloqueada. Para configurar o número máximo de processos criados podemos modificar a variável de ambiente $GOMAXPROCS ou utilizar a função runtime.GOMAXPROCS (n int) durante a execução. Seu valor default é 1, ou seja, todas as gorotinas são threads de usuário pertencentes ao mesmo processo.

3.2 Channel

Representa um canal de comunicação / sincronização que pode conectar duas computações concorrentes. Na realidade são referências para um objeto que coordena a comunicação.

<- é o operador binário de comunicação (envio) <- também é o operador, desta vez unário, de recebimento. Ambos são atômicos.

Operações de leitura (respesctivamente, escrita) em canais que não têm buffer ou que estão com o buffer vazio (respectivamente, cheio) bloqueiam, e o bloqueio se mantém até que haja aconteça uma operação de escrita (respectivamente, leitura). A linguagem também permite leitura (respectivamente, escrita) não bloqueante, a qual retorna imediatamente uma flag dizendo se a operação foi realizada com sucesso ou não.

Na criação são sempre bidirecionais, mas quando são recebidos como parâmetro por uma função, além da declaração normal (chan T) podem ser declarados para apenas receber (<-chan T) e apenas enviar (chan<- T), com o objetivo garantir que serão utilizados corretamente no corpo da função.

As funções len e cap, quando aplicadas a canal, retornam, repspectivamente, a quantidade de mensagens esperando e o tamanho do buffer do canal (caso este seja assíncrono), ou zero caso contrário. São úteis para determinar se o buffer de um canal assíncrono está cheio, o que permite evitar torná-lo síncrono.

Exemplos:

```
1 // declaração:
2   var canal_sincrono chan int  // envia e recebe inteiros
3   var canal_assincrono chan int  // poderia ser qualquer outro tipo
4 // instanciação:
5   canal_sincrono = make (chan int)
```

```
6   canal_assincrono = make (chan int, 10) // buffer de tamanho 10
7 // função que utiliza um canal apenas para leitura
8   func so_leio (canal_leitura <-chan int) { ... }
9 // função que utiliza um canal apenas para escrita
10  func so_escrevo (canal_escrita chan<- int) { ... }
11 // função que utiliza um canal bidirecional
12  func leio_e_escrevo (canal chan int) { ... }
13 // tipos de leitura:
14  v := <-canal_sincrono // sempre leitura sincrona
15  v := <-canal_assincrono // leitura assincrona quando houver espaço no buffer, síncrona caso contrário
16  v, ok := <-canal // sempre leitura assíncrona, ok é setado para true ou false de acordo com o sucesso
17 // tipos de escrita:
18  canal_sincrono <- v // sempre escrita síncrona
19  canal_assincrono <- v // escrita assíncrona quando houver espaço no buffer, síncrona caso contrário
20  ok := canal <- v // sempre escrita assíncrona, ok é setado para true ou false de acordo com o sucesso
21 // axiomas:
22  // 0 <= len(canal) <= cap(canal), para qualquer canal
23  // cap(make(chan int)) = 0
24  // cap(make(chan int, n)) = n
```

3.3 go

É o operador que inicia a execução concorrente de uma gorotina, sempre no mesmo espaço de endereçamento. Prefixe uma chamada de função ou de método com a palavra-chave go para executar a chamada numa nova gorotina. Quando a chamada termina, a gorotina também termina (silenciosamente). O efeito é similar a notação & do shell Unix para rodar um comando em background.

Exemplo:

```
1 ...
2 func faz_algo () { ... }
3 ...
4 func main () {
5   ...
6   // inicia a execução concorrente da função faz_algo ...
7   go faz_algo ()
8   // ... e continua executando ...
9   ...
10 }
```

3.4 select

É um estrutura de controle análoga a um switch, mas que age sempre sobre comunicações. Escolhe qual das comunicações listadas em seus casos pode prosseguir. Se todas estão bloqueadas, ele espera até que alguma desbloqueie. Se várias podem prosseguir, ele escolhe uma aleatoriamente.

Exemplo:

```
1 // esta função envia zeros e uns aleatoriamente a cada iteração
2 func gerador_aleatorio_de_bits (canal chan<- int)
3   for { // laço infinito
4     select { // escolha não-determinística
5       case c <- 0: // nada no corpo do case, o break é implícito (diferente de c)
6       case c <- 1:
7     }
8   }
```

4 Prática

4.1 Compilando e executando os exemplos

Todos os exemplos, exceto o hello.go, implementam a solução do problema produtor-consumidor com buffer limitado e taxa de produção e consumo aleatórias, sendo que o pc_channel_multi.go implementa a versão com vários produtores e vários consumidores. Os outros implementam a versão singular apenas para manter a simplicidade, mas também são facilmente generalizáveis.

```
cd src
make
./programa_desejado
```

4.2 Hello, World!

Arquivo: hello.go

```
/* hello */

package main // executável principal sempre deve pertencer a este pacote

import . "fmt" // importa funções de saída formatada (Printf)

// função principal (como em C)
func main() {
  //sem o . no import teríamos que utilizar fmt.Printf
  Printf("hello, world\n")
}
```

4.3 Funções Auxiliares

```
// gera um número inteiro aleatório no intervalo [begin, end]
func random_between (begin, end int64) int64 {
  random_byte := make([]byte, 1)
  rand.Read(random_byte)
  return int64(random_byte[0]) % (end - begin + 1) + begin
}
```

```
19 // dorme durante um intervalo de tempo aleatório
20 func random_sleep () {
21   time.Sleep(random_between(0, *MAX_SLEEP_TIME) * 10e7)
22 }

34 // envia um valor qualquer para o canal channel
35 func signal (channel chan int) {
36   // garante que a chamada só seja síncrona caso channel seja síncrono, ou seja,
37   // caso channel seja assíncrono e esteja com o buffer cheio não envia nada
38   if len(channel) < cap(channel) || cap(channel) == 0 {
39     channel <- 0
40   }
41 }

43 // recebe um valor qualquer do canal channel
44 func wait (channel chan int) {
45   <-channel
46 }
```

4.4 Constantes

```
32 const (
33   MAX_SLEEP_TIME = 5  // tempo máximo de sleep
34   BUFFER_SIZE = 5     // tamanho do buffer
35 )
```

4.5 Destruição (de variáveis)

Como a linguagem tem coletor de lixo, não é necessário (nem possível) se preocupar com a destruição das variáveis.

4.6 Mutex

Criação :

```
1 import . "sync"
2 var mutex *Mutex = new(Mutex)
```

Uso :

```
1 mutex.Lock()
2   // seção crítica
3 mutex.Unlock()
```

Arquivo: pc_mutex.go

Implementação (parte principal) :

```
37 var (
38   mutex = new(Mutex)
39   buffer []int
40   count, front, rear = 0, 0, 0
41 )
42
43 func producer () {
44   for i := 0; ; i++ { // laço infinito
45     for count == BUFFER_SIZE { // buffer cheio?
46       random_sleep() // (almost) busy wait
47     }
48     mutex.Lock()
49       if count < BUFFER_SIZE  {
50         Println("producing:", i)
51         buffer[rear] = i
52         rear = (rear + 1) % BUFFER_SIZE
53         count++
54       } else {
55         i--
56       }
57     mutex.Unlock()
58     random_sleep()
59   }
60 }
61
62 func consumer () {
63   for { // laço infinito
64     for count == 0 { // buffer vazio?
65       random_sleep() // (almost) busy wait
66     }
67     mutex.Lock()
68       if count > 0 {
69         v := buffer[front]
70         front = (front + 1) % BUFFER_SIZE
71         count--
72         Println("\t\t\tconsuming:", v)
73       }
74     mutex.Unlock()
75     random_sleep()
76   }
77 }
78
79 func main () {
80   buffer = make([]int, BUFFER_SIZE)
81   go producer()
82   go consumer()
83   wait(make(chan int)) // espera indefinidamente
```

```
}
```

4.7 Semáforo

Criação :

```
var semaforo chan int = make(chan int, MAX_VAL) // o semáforo sempre começa em 0
```

Inicialização :

```
// atribui o valor inicial n para o semáforo
// com n = 1 simulamos um mutex
for i := 0; i < n; i++ {
  signal(semaforo)
}
```

Uso :

```
wait(semaforo)
  // seção crítica
signal(semaforo)
```

Arquivo: pc_sem.go

Implementação (parte principal) :

```
var (
  mutex = make(chan int, 1);
  empty, full chan int;
  buffer []int;
  front, rear = 0, 0
)

func producer () {
  for i := 0; ; i++ { // laço infinito
    wait(empty)
    wait(mutex)
      Println("producing:", i)
      buffer[rear] = i
      rear = (rear + 1) % BUFFER_SIZE
    signal(mutex)
    signal(full)
    random_sleep()
  }
}

func consumer () {
  for { // laço infinito
    wait(full)
```

```
59      wait(mutex)
60        v := buffer[front]
61        front = (front + 1) % BUFFER_SIZE
62        Println("\t\t\tconsuming:", v)
63      signal(mutex)
64      signal(empty)
65      random_sleep()
66    }
67  }
68
69  func main () {
70    buffer = make([]int, BUFFER_SIZE)
71    empty = make(chan int, BUFFER_SIZE);
72    full = make(chan int, BUFFER_SIZE)
73    for i := 0; i < BUFFER_SIZE; i++ {  // inicializa empty com o valor BUFFER_SIZE
74      signal(empty)
75    }
76    signal(mutex)  // inicializa mutex com o valor 1
77    go producer()
78    go consumer()
79    wait(make(chan int))  // espera indefinidamente
80  }
```

4.8 Monitor

Criação :

```
1  type Monitor struct {
2    mutex *Mutex
3    // outros campos ...
4  }
```

Inicialização :

```
1  func newMonitor () *Monitor {
2    return &Monitor {
3      new(Mutex),
4      // inicializar outros campos ...
5    }
6  }
7  var monitor *Monitor = newMonitor()
```

Uso :

```
1  func (m *Monitor) metodo () {
2    m.mutex.Lock()
3      // fazer o que tem que fazer ...
4    m.mutex.Unlock()
```

```
5 }
6 monitor.metodo()
```

Arquivo: pc_monitor.go

Implementação (parte principal) :

```
37 type Monitor struct {
38   mutex *Mutex
39   empty, full chan int
40   buffer []int
41   front, rear, count int
42 }
43
44 var monitor *Monitor
45
46 func newMonitor() *Monitor {
47   return &Monitor {
48     new(Mutex),
49     make(chan int, BUFFER_SIZE),
50     make(chan int, BUFFER_SIZE),
51     make([]int, BUFFER_SIZE),
52     0, 0, 0 }
53 }
54
55 func (m *Monitor) produce(i int) {
56   m.mutex.Lock()
57     for m.count == BUFFER_SIZE {
58       m.mutex.Unlock()
59       wait(m.empty)       // (quase) uma variável de condição
60       m.mutex.Lock()
61     }
62     Println("producing:", i)
63     m.buffer[m.rear] = i
64     m.rear = (m.rear + 1) % BUFFER_SIZE
65     m.count++
66     signal(m.full)
67   m.mutex.Unlock()
68 }
69
70 func (m *Monitor) consume() {
71   m.mutex.Lock()
72     for m.count == 0 {
73       m.mutex.Unlock()
74       wait(m.full)       // (quase) uma variável de condição
75       m.mutex.Lock()
76     }
77     v := m.buffer[m.front]
```

```
78      m.front = (m.front + 1) % BUFFER_SIZE
79      m.count--
80      Println("\t\t\tconsuming:", v)
81      signal(m.empty)
82    m.mutex.Unlock()
83  }
84
85  func producer () {
86    for i := 0; ; i++ {  // laço infinito
87      monitor.produce(i)
88      random_sleep()
89    }
90  }
```

Discussão : semáforos são utilizados para simular as variáveis de condição. As diferenças entre ambos são:

Memória: semáforos têm memória, no sentido de que sinalização nunca são perdidas, pois permanecem armazenadas no semáforo. Variáveis de condição não tem memória, caso não haja ninguém esperando quando ocorre um sinal, o sinal se perde.

Atomicidade: variáveis de condição destravam o mutex e entram em estado de espera de forma atômica. Isso é necessário pois, se houvesse uma troca de contexto entre estas duas ações, um outra thread poderia entrar no monitor e gerar um sinal que seria perdido, podendo causar um deadlock. Os semáforos, ao contrário, não realizam estas duas operações de forma atômica, mas isso não é um problema, já que os sinais não se perdem.

Variáveis de Condição:

Referências Bibliográficas

[1] Completar...

[2] Completar...