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 Introdução

"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 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 de 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)
```

```
  canal_assincrono = make (chan int, 10) // buffer de tamanho 10
// função que utiliza um canal apenas para leitura
  func so_leio (canal_leitura <-chan int) { ... }
// função que utiliza um canal apenas para escrita
  func so_escrevo (canal_escrita chan<- int) { ... }
// função que utiliza um canal bidirecional
  func leio_e_escrevo (canal chan int) { ... }
// tipos de leitura:
  v := <-canal_sincrono // sempre leitura sincrona
  v := <-canal_assincrono // leitura assincrona quando houver espaço no buffer, síncrona caso contrário
  v, ok := <-canal // sempre leitura assíncrona, ok é setado para true ou false de acordo com o sucesso
// tipos de escrita:
  canal_sincrono <- v // sempre escrita síncrona
  canal_assincrono <- v // escrita assíncrona quando houver espaço no buffer, síncrona caso contrário
  ok := canal <- v // sempre escrita assíncrona, ok é setado para true ou false de acordo com o sucesso
// axiomas:
  // 0 <= len(canal) <= cap(canal), para qualquer canal
  // cap(make(chan int)) = 0
  // 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:

```
...
func faz_algo () { ... }
...
func main () {
  ...
  // inicia a execução concorrente da função faz_algo ...
  go faz_algo ()
  // ... e continua executando ...
  ...
}
```

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 deve 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

```
1  /* hello */
2
3  package main // executável principal sempre deve pertencer a este pacote
4
5  import . "fmt" // importa funções de saída formatada (Printf)
6
7  // função principal (como em C)
8  func main() {
9     //sem o . no import teríamos que utilizar fmt.Printf
10    Printf("hello, world\n")
11 }
```

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
}
```

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

// envia o inteiro i através do canal channel
func send (channel chan int, i int) {
  channel <- i
}

// recebe um inteiro através do canal channel e o retorna
func recv (channel chan int) int {
  return <-channel
}

// envia um valor qualquer para o canal channel
func signal (channel chan int) {
  // garante que a chamada só seja síncrona caso channel seja síncrono, ou seja,
  // caso channel seja assíncrono e esteja com o buffer cheio não envia nada
  if len(channel) < cap(channel) || cap(channel) == 0 {
    send(channel, 0) // poderia ser qualquer valor
  }
}

// recebe um valor qualquer do canal channel
func wait (channel chan int) {
  recv(channel) // ignora o valor recebido
}

// envia assincronamente um valor qualquer para o canal channel
func trysignal (channel chan int) {
  _ = channel <- 0
}

// tenta simular a atomicidade necessária para o bom funcionamento da variável de condição
func wait_cond (channel chan int, mutex *Mutex) {
  c := make(chan int)     // canal síncrono
  go waiter(channel, c) // tentativa de simular a atomicidade - não funciona!!!
  mutex.Unlock()
  wait(c)                 // em vez de wait(channel)
  mutex.Lock()
}

// espera pelo canal in e então sinaliza no canal out
func waiter (in, out chan int) {
  wait(in)
  signal(out)
}
```

4.4 Constantes

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

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 :

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

Uso :

```
mutex.Lock()
  // seção crítica
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
84  }
```

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)
    wait(mutex)
      v := buffer[front]
      front = (front + 1) % BUFFER_SIZE
      Println("\t\t\tconsuming:", v)
    signal(mutex)
    signal(empty)
    random_sleep()
  }
}

func main() {
  buffer = make([]int, BUFFER_SIZE)
  empty = make(chan int, BUFFER_SIZE);
  full = make(chan int, BUFFER_SIZE)
  for i := 0; i < BUFFER_SIZE; i++ { // inicializa empty com o valor BUFFER_SIZE
    signal(empty)
  }
  signal(mutex) // inicializa mutex com o valor 1
  go producer()
```

```
78    go consumer()
79    wait(make(chan int)) // espera indefinidamente
80  }
```

4.8 Monitor

Criação :

```
type Monitor struct {
  mutex *Mutex
  // outros campos ...
}
```

Inicialização :

```
func newMonitor () *Monitor {
  return &Monitor {
    new(Mutex),
    // inicializar outros campos ...
  }
}
var monitor *Monitor = newMonitor()
```

Uso :

```
func (m *Monitor) metodo () {
  m.mutex.Lock()
    // fazer o que tem que fazer ...
  m.mutex.Unlock()
}
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), // semáforo
```

```
50      make(chan int, BUFFER_SIZE), // semáforo
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 { // buffer cheio?
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 { // buffer vazio?
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  }
91
92  func consumer () {
93    for { // laço infinito
94      monitor.consume()
95      random_sleep()
96    }
```

```
97 }
98
99 func main () {
100   monitor = newMonitor()
101   go producer()
102   go consumer()
103   wait(make(chan int)) // espera indefinidamente
104 }
```

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.

4.9 Variável de Condição

Criação :

```
// canal sem buffer
var cond chan int = make(chan int)
```

Uso :

```
wait_cond(cond, mutex)
trysignal(cond)
```

Arquivo: pc_monitor_cond.go

Implementação (parte principal) :

```
57 type Monitor struct {
58   mutex *Mutex
59   empty, full chan int
60   buffer []int
61   front, rear, count int
62 }
63
64 var monitor *Monitor
65
66 func newMonitor () *Monitor {
67   return &Monitor {
68     new(Mutex),
```

```
69      make(chan int), // variável de condição
70      make(chan int), // variável de condição
71      make([]int, BUFFER_SIZE),
72      0, 0, 0 }
73  }
74
75  func (m *Monitor) produce (i int) {
76    m.mutex.Lock()
77      for m.count == BUFFER_SIZE { // buffer cheio?
78        wait_cond(m.empty, m.mutex)
79      }
80      Println("producing:", i)
81      m.buffer[m.rear] = i
82      m.rear = (m.rear + 1) % BUFFER_SIZE
83      m.count++
84      trysignal(m.full)
85    m.mutex.Unlock()
86  }
87
88  func (m *Monitor) consume () {
89    m.mutex.Lock()
90      for m.count == 0 { // buffer vazio?
91        wait_cond(m.full, m.mutex)
92      }
93      v := m.buffer[m.front]
94      m.front = (m.front + 1) % BUFFER_SIZE
95      m.count--
96      Println("\t\t\tconsuming:", v)
97      trysignal(m.empty)
98    m.mutex.Unlock()
99  }
100
101 func producer () {
102   for i := 0; ; i++ { // laço infinito
103     monitor.produce(i)
104     random_sleep()
105   }
106 }
107
108 func consumer () {
109   for { // laço infinito
110     monitor.consume()
111     random_sleep()
112   }
113 }
114
115 func main () {
```

```
116   monitor = newMonitor()
117   go producer()
118   go consumer()
119   wait(make(chan int)) // espera indefinidamente
120 }
```

Discussão : a semântica das variáveis de condição diz que o destrancamento do mutex tem que acontecer de forma atômica com o início da espera pela variável (ver 4.8). A melhor tentativa de simular esta atomicidade foi criar uma outra gorotina (waiter) e deixá-la esperando pelo variável de condição antes de destrancar o mutex, e então esperar pela resposta desta gorotina. Porém, mesmo esta implementação está incorreta, pois não há como garantir que a gorotina criada já esteja esperando pela variável de condição antes de destrancarmos o mutex. Portanto, não há como implementar variáveis de condição corretamente na linguagem, podemos apenas simular seu comportamente com semáforos, mas são coisas diferentes. Para se ter uma idéia da chance de ocorrência do deadlock, no meu computador pessoal o programa acima entrou em deadlock em média após 40000 iterações.

4.10 The Go Way

Os canais de go são exatamente o que precisamos para comunicação entre as gorotinas. Veja como a solução é bem mais elegante:

Arquivo : pc_channel.go

Implementação (parte principal) :

```
52 var pipe = make(chan int, BUFFER_SIZE)
53
54 func producer () {
55   for i := 0; ; i++ { // laço infinito
56     Println("producing:", i)
57     send(pipe, i)
58     random_sleep()
59   }
60 }
61
62 func consumer () {
63   for { // laço infinito
64     v := recv(pipe)
65     Println("\t\t\tconsuming:", v)
66     random_sleep()
67   }
68 }
69
70 func main () {
71   go producer()
72   go consumer()
```

```
73    wait(make(chan int)) // espera indefinidamente
74  }
```

4.11 Múltiplos Produtores e Consumidores

Arquivo : pc_channel_multi.go

Execução :

```
$ ./pc_channel_multi.go -s BUFFER_SIZE -t MAX_SLEEP_TIME -m MAX_PRODUCTIONS -p
    N_PRODUCERS -c N_CONSUMERS
```

Implementação (parte principal) :

```
40  var (
41    BUFFER_SIZE = flag.Int("s", 5, "BUFFER_SIZE") // flag -s
42    MAX_SLEEP_TIME = flag.Int64("t", 5, "MAX_SLEEP_TIME") // flag -t
43    MAX_PRODUCTIONS = flag.Int("m", 0, "MAX_PRODUCTIONS, 0 for infinity") // flag -m
44    N_PRODUCERS = flag.Int("p", 3, "N_PRODUCERS") // flag -p
45    N_CONSUMERS = flag.Int("c", 3, "N_CONSUMERS") // flag -c
46    pipe = make(chan int, *BUFFER_SIZE)
47    quit  = make(chan int)
48  )
49
50  func producer (id int) {
51    for i := 0; i < *MAX_PRODUCTIONS || *MAX_PRODUCTIONS == 0; i++ {
52      Println(id, "producing:", i + id * 1e5)
53      send(pipe, i + id * 1e5)
54      random_sleep()
55    }
56    signal(quit)
57  }
58
59  func consumer (id int) {
60    for {
61      v := recv(pipe)
62      if closed(pipe) { // canal fechado?
63        break
64      }
65      Println("\t\t\t", id, "consuming:", v)
66      random_sleep()
67    }
68    signal(quit)
69  }
70
71  func main () {
72    flag.Parse() // processa flags
```

```
73    Println("BUFFER_SIZE =", *BUFFER_SIZE)
74    Println("MAX_SLEEP_TIME =", *MAX_SLEEP_TIME)
75    Println("MAX_PRODUCTIONS =", *MAX_PRODUCTIONS)
76    Println("N_PRODUCERS =", *N_PRODUCERS)
77    Println("N_CONSUMERS =", *N_CONSUMERS)
78    for i := 0; i < *N_PRODUCERS; i++ { // cria produtores
79      go producer(i + 1)
80    }
81    for i := 0; i < *N_CONSUMERS; i++ { // cria consumidores
82      go consumer(i + 1)
83    }
84    for i := 0; i < *N_PRODUCERS; i++ { // espera o término dos produtores
85      wait(quit)
86    }
87    close(pipe) // fecha comunicação
88    for i := 0; i < *N_CONSUMERS; i++ { // espera o término dos consumidores
89      wait(quit)
90    }
91  }
```

5 Conclusão

Go é uma linguagem que tenta aliar um certo nível de abstração (coletor de lixo, canais inspirados num modelo formal amplamente aceito) com eficiência de compilação e execução. A linguagem não apresenta nativamente algumas funcionalidades comuns de concorrência, mas sua implementação é relativamente fácil, pois a linguagem possui o tipo canal que é tão poderoso quanto as funcionalidades a que estamos acostumados.

A disseminação de computadores multiprocessados e de sistemas distribuídos, aliada às dificuldades que temos atualmente para programar corretamente sistemas concorrentes, certamente fará com que linguagens como essa apresentem grande crescimento e importância, podendo começar a substituir gradualmente linguagens mais antigas (como C), pois a ineficiência de execução de suas abstrações individualmente vai ser compensada pela eficiência de sua execução como um todo, por aproveitarem melhor os recursos de hardware disponíveis.

Referências Bibliográficas

[1] *The Go Programming Language*, http://golang.org/. Acessado em: 04/04/2010.

[2] Hoare, C. A. R., *Communicating Sequential Processes*, Communications of the ACM 21 (8): 666–677.