

**Pontifícia Universidade Católica do Rio Grande do Sul**  
**Sistemas Operacionais**  
**Engenharia de Software**

**Carolina Ferreira, Felipe Freitas, Luiza Heller e Mateus Caçabuena**

**@carolina.michel@edu.pucrs.br**

**@f.freitas007@edu.pucrs.br**

**@luiza.heller@edu.pucrs.br**

**@mateus.cacabuena@edu.pucrs.br**

***Relatório do Trabalho 1***

**Porto Alegre**

**2024**

# Introdução

O presente relatório aborda a implementação de soluções para dois problemas clássicos em sistemas operacionais: Produtores e Consumidores e Jantar dos Canibais. Nesta explicação, será retratado a solução desenvolvida para os problemas e como foi aplicada nos problemas anteriores.

Este trabalho foi realizado utilizando a linguagem de programação Go, o que foi uma decisão unânime entre os membros do grupo, pois é própria para situações que envolvem comunicação entre múltiplas threads e sincronização delas.

Ambos os problemas envolvem a coordenação e sincronização de múltiplas threads que acessam recursos compartilhados, exigindo técnicas eficazes de programação concorrente para garantir a integridade e consistência dos dados:

- **Produtores e Consumidores:** descreve um cenário em que threads produtoras adicionam itens a uma fila compartilhada, enquanto threads consumidores retiram esses itens. O desafio reside em garantir a exclusão mútua durante o acesso à fila, evitando condições de corrida e garantindo que a fila não se torne nem superlotada nem vazia.
- **Jantar dos Canibais:** apresenta um cenário em que um grupo de canibais se serve de porções em uma travessa compartilhada, que tem capacidade limitada. Aqui, é crucial sincronizar as ações dos canibais e do cozinheiro para evitar *deadlock* e garantir que o cozinheiro seja acionado apenas quando a travessa estiver vazia.

Para solucionar esses problemas, utilizamos semáforos implementados pelo grupo e *mutexes* personalizados baseados no algoritmo de Lamport. Essa técnica é essencial para garantir a corretude e eficiência das soluções propostas.

Este relatório apresentará as soluções desenvolvidas, destacando os principais aspectos de implementação e a aplicação prática dessas soluções nos problemas propostos. Ao final, serão discutidas as lições aprendidas durante o processo de desenvolvimento e as possíveis melhorias para trabalhos futuros nesta área.

## Produtores e Consumidores

Como estratégia para resolver esse problema, a nossa solução utiliza o conceito de **goroutines** da linguagem Go para implementar produtores e consumidores como threads leves, assim, ocorrendo a execução de forma concorrentes das operações de produção e consumo. Ainda, nós utilizamos um **mutex personalizado** para garantir a exclusão mútua durante o acesso à fila compartilhada entre as goroutines, garantindo que apenas um produtor ou consumidor tenha permissão para fazer modificações na fila em determinado momento, evitando condições de corrida.

Seguindo para a estrutura de dados, em nossa solução nós definimos duas estruturas:

- **Queue:** não apenas representa a fila compartilhada entre os produtores e consumidores, mas ainda possui métodos para adicionar e remover os itens de forma segura.
- **Mutex:** uma estrutura que implementa um mecanismo de mutex personalizado incorporando o algoritmo de **Lamport** no método **Lock**. Esse algoritmo atribui um número para cada thread, garantindo que a thread com número mais baixo tenha prioridade para acessar a seção crítica. A utilização dessa personalização garante a exclusão mútua durante o acesso à fila, e evita possíveis condições de corrida, por impedir que múltiplas goroutines modifiquem a fila ao mesmo tempo.

Figura 1: Estruturação da Queue

```
type Queue []int

func (q *Queue) Enqueue(item int) (success bool) {
    if len(*q) == QUEUE_SIZE {
        return false
    }
    *q = append(*q, item)
    return true
}

func (q *Queue) Dequeue() (head int, success bool) {
    if len(*q) == 0 {
        return -1, false
    }
    item := (*q)[0]
    *q = (*q)[1:]
    return item, true
}
```

Figura 2: Estruturação do Mutex

```
type Mutex struct {  
    number [THREAD_COUNT]int  
}  
  
func NewMutex() *Mutex {  
    return &Mutex{  
        number: [THREAD_COUNT]int{},  
    }  
}
```

Figura 3: Funções do Mutex

```
func (m *Mutex) maxNumber() int {  
    if len(m.number) == 0 {  
        return -1  
    }  
  
    max := m.number[0]  
    for i := 1; i < len(m.number); i++ {  
        if m.number[i] > max {  
            max = m.number[i]  
        }  
    }  
    return max  
} ✨  
  
func (m *Mutex) Lock(i int) {  
    m.number[i] = 1 + m.maxNumber()  
    for j := 0; j < THREAD_COUNT; j++ {  
        for m.number[j] != 0 &&  
            (m.number[j] < m.number[i] || (m.number[j] == m.number[i] && j < i)) {  
            // Wait until all threads with smaller numbers or with the same  
            // number, but with higher priority, finish their work  
        }  
    }  
}  
  
func (m *Mutex) Unlock(i int) {  
    m.number[i] = 0  
}
```

A respeito das nossas funções de produtores e consumidores:

- **Produtores:** geram elementos aleatórios, esperam para acessar a seção crítica e tentam inseri-los à fila compartilhada, sendo que se essa estiver cheia, o produtor libera a seção e espera a próxima oportunidade de acesso.
- **Consumidores:** seguem o mesmo princípio, esperam para acessar a seção crítica e tentam retirar itens dessa fila compartilhada, se ela estiver vazia ele libera a seção e espera a próxima oportunidade.

Figura 4: Função dos Produtores

```
func Producer(id int) {
    for {
        mutex.Lock(id)
        fmt.Println(YELLOW, id, "- Pronto para produzir...", RESET)
        item := GetRandom(99)
        success := queue.Enqueue(item)
        if success {
            fmt.Println(GREEN,
                "\tAdicionando: ", item, "\n",
                "\tFila Seção Crítica: ", queue, "\n",
                "\tTimeStamp: ", CurrentTimeStamp(), "\n",
                RESET)
        } else {
            fmt.Println(RED, "\tFila cheia, não foi possível adicionar\n", RESET)
        }
        mutex.Unlock(id)
        WaitRandom(PRODUCERS_COUNT)
    }
}
```

Figura 5: Função dos Consumidores

```
func Consumer(id int) {
    for {
        WaitRandom(CONSUMERS_COUNT)
        mutex.Lock(id)
        fmt.Println(YELLOW, id, "- Tentando consumir...", RESET)
        previousQueue := queue
        item, success := queue.Dequeue()
        if success {
            fmt.Println(CYAN,
                "\tAnterior: ", previousQueue, "\n",
                "\tRemovido: ", item, "\n",
                "\tNew: ", queue, "\n",
                "\tTimeStamp: ", CurrentTimeStamp(), "\n",
                RESET)
        } else {
            fmt.Println(RED, "\tFila vazia, não foi possível remover\n", RESET)
        }
        mutex.Unlock(id)
    }
}
```

Terminando nossa implementação, temos a função principal, main, onde o programa inicia um número adequado de goroutines para produtores e consumidores, com a utilização de um loop. Aqui, cada produtor e consumidor será uma goroutine separada, o que permite que as operações de produção e consumo sejam executadas de forma concorrente.

Figura 6: Função Principal

```
func main() {
    runtime.GOMAXPROCS(THREAD_COUNT)
    fmt.Println(RESET)
    fmt.Println(BOLD, "Producer Consumer with: ")
    fmt.Println(MAGENTA, "\tQueue Size:\t", QUEUE_SIZE)
    fmt.Println(GREEN, "\t", "Producers: ", "\t", PRODUCERS_COUNT)
    fmt.Println(CYAN, "\t", "Consumers: ", "\t", CONSUMERS_COUNT)
    fmt.Println(RESET)

    for i := 0; i < PRODUCERS_COUNT; i++ {
        go Producer(i)
    }

    for i := 0; i < CONSUMERS_COUNT; i++ {
        go Consumer(i + PRODUCERS_COUNT)
    }

    // Stops the program after some delay, so the user can see the output
    <- time.After(250 * time.Millisecond)
}
```

Por conta do programa possuir funções sendo executadas ao mesmo tempo, inúmeros resultados diferentes podem ocorrer. A figura a seguir mostra o exemplo de um resultado ao rodar o programa:

Figura 7: Exemplo de Resultado

```
PS C:\Users\Matheus Caçabuena\Desktop\pucrs\SistemasOperacionais\Trabalho1\ProducerConsumer> go run ProducerConsumer.go

Producer Consumer with:
  Queue Size:      3
  Producers:       2
  Consumers:       6

0 - Pronto para produzir...
  Adicionando: 66
  Fila Seção Crítica: [66]
  TimeStamp: 8118455

1 - Pronto para produzir...
  Adicionando: 9
  Fila Seção Crítica: [66 9]
  TimeStamp: 8118455

3 - Tentando consumir...
  Anterior: [66 9]
  Removido: 66
  New: [9]
  TimeStamp: 8118464
```

## Jantar dos Canibais

A solução proposta aborda o problema dos canibais e do cozinheiro através de um código que utiliza threads para modelar as entidades envolvidas. O objetivo é garantir a sincronização adequada das ações entre os canibais e o cozinheiro, evitando problemas como o *deadlock*. O código utiliza *mutex* e semáforos para controlar o acesso à travessa de comida compartilhada. Para isso, foram definidas as variáveis globais: **N** (quantidade de canibais), **M** (quantidade de travessas), **maxServingsCount** (quantidade total de travessas que o cozinheiro tem que reabastecer), **L** (quantidade de cozinheiros), **threadCount**, que é o tamanho da lista de threads que o *mutex* controla, **mutex** (semáforo que garante a exclusão mútua), **emptyPot** (semáforo que sinaliza quando a travessa está vazia) e **fullPot** (semáforo que sinaliza a travessa cheia).

Figura 8: Variáveis Globais

```
var (  
    // Savages Count  
    N uint  
    // Max Servings Count  
    maxServingsCount uint  
    // Variable Servings Count  
    M uint  
    // Cooks Count  
    L uint  
    // Threads Count  
    threadCount uint  
  
    mutex *Mutex  
    emptyPot = NewSemaphore(0)  
    fullPot = NewSemaphore(0)  
)
```

O cozinheiro é representado por uma *goroutine* (**Cook(id uint)**) que coloca  $|maxServingsCount|$  novas porções na travessa sempre que ela estiver vazia. Os canibais, representados por outra *goroutine* (**Savage(id uint)**), se servem da travessa quando há comida disponível. Se a travessa estiver vazia, o cozinheiro é acordado por algum dos canibais e, quando servidos, os canibais comem as suas refeições e dão continuidade ao ciclo.

Figura 9: Função do Cozinheiro

```
func Cook(id uint) {  
    for {  
        emptyPot.Wait()  
        putServingsInPot(id)  
        M = maxServingsCount  
        fullPot.Signal()  
        WaitRandom(DEFAULT_COOKS_COUNT)  
    }  
}
```

Figura 10: Função dos Canibais

```
func Savage(id uint) {
    for {
        WaitRandom(DEFAULT_SAVAGES_COUNT)
        mutex.Lock(id)
        if M == 0 {
            wakeUpCook(id)
            emptyPot.Signal()
            fullPot.Wait()
        }
        M--
        getServingFromPot(id)
        mutex.Unlock(id)
    }
}
```

A implementação evita que múltiplos canibais acessem a travessa ao mesmo tempo, o que garante consistência e evita condições de corrida. Ao final, a solução oferece uma abordagem eficaz para resolver o problema do jantar dos canibais de forma segura e eficiente, assegurando a ordem e a distribuição justa das porções de comida.

Por conta do programa possuir funções sendo executadas ao mesmo tempo, inúmeros resultados diferentes podem ocorrer. A figura a seguir mostra o exemplo de um resultado ao rodar o programa (junto com o breve tutorial de como inserir a quantidade desejada de canibais, cozinheiros e comidas disponíveis no início):

Figura 11: Exemplo de Resultado com o Valor Padrão

```
PS C:\Users\Matheus Caçabuena\Desktop\pucrs\SistemasOperacionais\Trabalho1\DiningSavages> go run DiningSavages.go

===== Dining Savages =====

Useful arguments missing
Usage: go run DiningSavages.go <N = number of savages> <M = number of servings> <Optional: L = number of cooks>
Example: go run DiningSavages.go 5 5 1
Using default values to fill non-provided arguments
    Savages:      5
    Servings Count: 5
    Cooks:        1

Savage 2 is serving...
Savage 2 is serving...
Savage 1 is serving...
Savage 3 is serving...
Savage 4 is serving...
Savage 3- Pot is empty, I'll wake up the cook...

Cook 0 is putting servings in pot...

Savage 3 is serving...
Savage 5 is serving...
Savage 1 is serving...
Savage 4 is serving...
Savage 3 is serving...
Savage 5- Pot is empty, I'll wake up the cook...
```

Obs.: caso não inserido o valor de canibais, cozinheiros e quantidade inicial de comida, o valor padrão será de 5 canibais, 5 pratos de comida iniciais e 1 cozinheiro.



## **Bibliografia**

G.L Peterson. Mythis About the Mutual Exclusion Problem – Volume 12, número 3, 1981.

Leslie Lamport. A new Solution of Dijkstra's Concurrent Programming Problem – Volume 17, número 8, 1974.

Filho, Sergio. Slides de aula – [moodle.pucrs.br](https://moodle.pucrs.br), 2024