



# UFRJ

Universidade Federal do Rio de Janeiro  
Engenharia de Computação e Informação

Trabalho prático 2  
Sistemas distribuídos COS 470  
Prof. Daniel Ratton Figueiredo

Lucas Máximo Dantas  
Amanda Aparecida Cordeiro Lucio

2022.1  
Rio de Janeiro

## 1. Decisões de Implementação

### 1.1. Linguagem de programação

A linguagem de programação escolhida foi **c e go**, sendo as mesmas linguagens de baixo nível, desta forma, facilitando e permitindo a interação com o hardware. A escolha da linguagem baseou-se, principalmente, na nossa proximidade com a linguagem, visto que era necessário otimizar o tempo devido às nossas outras demandas da faculdade.

### 1.2. Sistema Operacional

Para o desenvolvimento, decidimos utilizar a *SysCall* do Linux, sendo a distribuição utilizada o Ubuntu.

### 1.3. Implementação

A implementação pode ser acompanhada através do seguinte repositório:

<https://github.com/AmandaACLucio/Sistemas-Distribuidos>

## 2. Implementações

### 2.1. Spinlock

A implementação do lock, foi realizada utilizando a biblioteca atômica “sync/atomic”, na linha 91, a função `atomic.CompareAndSwap`, realizar papel similar ao `test_and_set` do C, primeiro ele compara os valores, e se for true, alterar o valor de forma atômica, evitando assim o Race Condicional

```
86  type Mylock struct {
87      held uint32
88  }
89
90  func (l Mylock) acquire() {
91      for !atomic.CompareAndSwapUint32(&l.held, 0, 1) {
92      }
93  }
94
95  func (l Mylock) release() {
96      l.held = 0
97  }
```

No trecho de código a seguir, podemos observar o momento que o lock é utilizado, para realizar a soma dos valores das threads na variável Valor Final

```
//-----Função onde as threads irão realizar a soma
func soma(numerosAleatorios *[n]int, tamanho int, i int, wg *sync.WaitGroup, mu *sync.Mutex) {
    var total int

    for o := i * tamanho; o < tamanho*(i+1); o++ {
        //fmt.Println("thread : ", i, " contando")
        total += numerosAleatorios[o]
    }

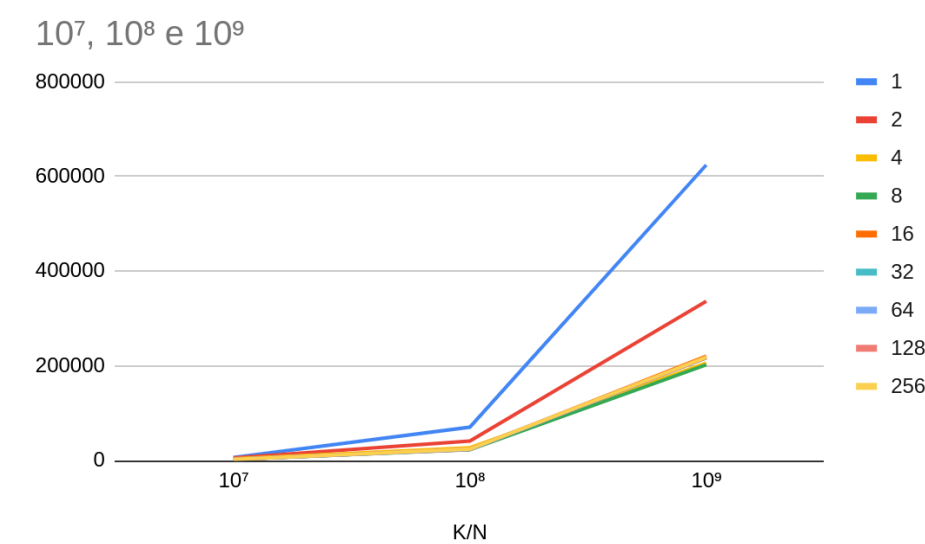
    //fmt.Println("thread : ", i, " somou : ", total)
    mu.Lock()
    *valorFinal += total
    mu.Unlock()
    defer wg.Done()
}
```

2.2. Resultados

A tabela de resultado pode ser acompanhada a seguir:

| K/N | 10 <sup>7</sup> | 10 <sup>8</sup> | 10 <sup>9</sup> |
|-----|-----------------|-----------------|-----------------|
| 1   | 6584,5          | 70596,5         | 624294,5        |
| 2   | 5671            | 41433,5         | 336416,5        |
| 4   | 3873,5          | 27330,5         | 206248          |
| 8   | 2826,5          | 23069,5         | 202277,5        |
| 16  | 2498            | 23733,5         | 220988          |
| 32  | 2446            | 24285,5         | 217108          |
| 64  | 2424,5          | 23651           | 217750,5        |
| 128 | 2477            | 24483,5         | 217755          |
| 256 | 2532            | 24352,5         | 218287,5        |


Podemos observar que a partir de 4 threads rodando em paralelo, a diferença de tempo de execução do programa pouco se alterou.



### 2.3. Alteração de hardware

Para se poder obter um melhor entendimento dos resultados e comportamento das threads, foi realizada a execução de comandos no terminal do sistema Unix, com o objetivo de desativar os núcleos do processador e rodar novamente os resultados.

O processador utilizado para a execução do código foi o modelo Ryzen 5 3600 que conta com 6 núcleos, mais informações das especificações do processador podem ser obtidas a seguir, ou no site da fabricante.

| AMD Ryzen™ 5 3600      |                                                                                                      |                                           |                                                     |
|------------------------|------------------------------------------------------------------------------------------------------|-------------------------------------------|-----------------------------------------------------|
| General Specifications | Plataforma: Boxed Processor                                                                          | Família de produto: AMD Ryzen™ Processors | Linha de produto: AMD Ryzen™ 5 Desktop Processors   |
|                        | Nº de núcleos de CPU: 6                                                                              | Nº de threads: 12                         | Clock de Max Boost: Até 4,2GHz                      |
|                        | Clock básico: 3,6GHz                                                                                 | Cachê L1 total: 384KB                     | Cachê L2 total: 3MB                                 |
|                        | Cachê L3 total: 32MB                                                                                 | TDP / TDP Padrão: 65W                     | Processor Technology for CPU Cores: TSMC 7nm FinFET |
|                        | Desbloqueado  : Sim | CPU Socket: AM4                           | Solução térmica (PIB): Wraith Stealth               |
|                        | Solução térmica (MPK): Wraith Stealth                                                                | Temps máx: 95°C                           | Data de lançamento: 7/7/2019                        |
|                        |                                                                                                      |                                           |                                                     |

O seguinte comando tem a função de desativar núcleos do processador, na imagem, está sendo desativado o núcleo 11 do processador. Para reativar o núcleo, é preciso executar o mesmo comando, porém com o parâmetro 0 alterado para 1

```
root@elementaryosdesktop712ef8a3:/sys/devices/system/cpu# echo 0 > /sys/devices/system/cpu/cpu11/online
```

E para ser possível identificar os núcleos ativos, o seguinte comando realiza essa finalidade.

```
root@elementaryosdesktop712ef8a3:/sys/devices/system/cpu# cat /sys/devices/system/cpu/online
0
```

Se quisermos observar os núcleos desativados, poderíamos alterar o caminho de “online” para “offline”.

### 2.4. Resultado 2

Com essa possibilidade de controlar os núcleos do processador, foi realizado o mesmo procedimento novamente, porém utilizando 1,2,4,8 e 12 núcleos, para cada uma das situações. O resultado para o vetor de tamanho  $10^7$  pode ser observado.

| 10 <sup>7</sup> |           |             |           |             |           |
|-----------------|-----------|-------------|-----------|-------------|-----------|
| 1 Núcleo        |           | 4 Núcleos   |           | 12 Núcleos  |           |
|                 | Média     |             | Média     |             | Média     |
| Threads 1       | 827.037,5 | Threads 1   | 685.046,0 | Threads 1   | 651.953,0 |
| Threads 2       | 833.468,0 | Threads 2   | 408.447,5 | Threads 2   | 354.903,0 |
| Threads 4       | 850.244,0 | Threads 4   | 345.605,5 | Threads 4   | 239.612,5 |
| Threads 8       | 846.387,5 | Threads 8   | 362.114,5 | Threads 8   | 236.336,0 |
| Threads 16      | 830.489,5 | Threads 16  | 313.199,5 | Threads 16  | 243.270,5 |
| Threads 32      | 831.765,0 | Threads 32  | 306.495,5 | Threads 32  | 240.391,0 |
| Threads 64      | 843.791,5 | Threads 64  | 312.753,5 | Threads 64  | 237.926,0 |
| Threads 128     | 834.435,5 | Threads 128 | 300.413,5 | Threads 128 | 239.514,5 |
| Threads 256     | 833.498,5 | Threads 256 | 303.638,5 | Threads 256 | 238.831,0 |
| 2 Núcleos       |           | 8 Núcleos   |           |             |           |
|                 | Média     |             | Média     |             |           |
| Threads 1       | 751.474,0 | Threads 1   | 649.600,5 |             |           |
| Threads 2       | 544.989,5 | Threads 2   | 360.173,0 |             |           |
| Threads 4       | 520.935,5 | Threads 4   | 255.488,5 |             |           |
| Threads 8       | 545.370,0 | Threads 8   | 252.944,0 |             |           |
| Threads 16      | 509.536,5 | Threads 16  | 242.512,5 |             |           |
| Threads 32      | 521.275,0 | Threads 32  | 243.483,0 |             |           |
| Threads 64      | 496.356,0 | Threads 64  | 243.763,0 |             |           |
| Threads 128     | 502.808,0 | Threads 128 | 243.748,0 |             |           |
| Threads 256     | 500.908,5 | Threads 256 | 244.677,5 |             |           |

E para 10<sup>8</sup>, a seguir:

| 10 <sup>8</sup> |             |             |           |             |           |
|-----------------|-------------|-------------|-----------|-------------|-----------|
| 1 Núcleo        |             | 4 Núcleos   |           | 12 Núcleos  |           |
|                 | Média       |             | Média     |             | Média     |
| Threads 1       | 1.067.153,0 | Threads 1   | 635.129,0 | Threads 1   | 639.983,0 |
| Threads 2       | 1.049.879,5 | Threads 2   | 373.078,0 | Threads 2   | 371.107,0 |
| Threads 4       | 1.101.940,5 | Threads 4   | 285.160,0 | Threads 4   | 235.972,5 |
| Threads 8       | 1.067.875,5 | Threads 8   | 267.385,0 | Threads 8   | 218.434,0 |
| Threads 16      | 1.056.284,0 | Threads 16  | 258.740,5 | Threads 16  | 228.040,0 |
| Threads 32      | 1.110.134,0 | Threads 32  | 254.121,5 | Threads 32  | 226.302,5 |
| Threads 64      | 1.137.195,0 | Threads 64  | 247.039,5 | Threads 64  | 224.491,0 |
| Threads 128     | 1.045.286,0 | Threads 128 | 242.207,0 | Threads 128 | 225.292,5 |
| Threads 256     | 1.040.250,5 | Threads 256 | 245.469,5 | Threads 256 | 224.081,5 |
| 2 Núcleos       |             | 8 Núcleos   |           |             |           |
|                 | Média       |             | Média     |             |           |
| Threads 1       | 664.211,0   | Threads 1   | 635.013,0 |             |           |
| Threads 2       | 433.852,0   | Threads 2   | 338.521,5 |             |           |
| Threads 4       | 408.589,0   | Threads 4   | 220.656,5 |             |           |
| Threads 8       | 409.363,0   | Threads 8   | 219.497,0 |             |           |
| Threads 16      | 405.271,5   | Threads 16  | 220.608,5 |             |           |
| Threads 32      | 399.696,0   | Threads 32  | 214.311,0 |             |           |
| Threads 64      | 400.164,5   | Threads 64  | 212.208,5 |             |           |
| Threads 128     | 393.140,0   | Threads 128 | 209.535,5 |             |           |
| Threads 256     | 393.382,0   | Threads 256 | 208.604,5 |             |           |

Podemos realizar algumas observações desses resultados:

- Quanto maior o tempo de execução, mais o programa se beneficia com a utilização de mais threads.
- Com a presença de um único núcleo, a utilização de várias threads perde o sentido, pois nada ocorre em paralelo, causando assim um empate técnico nesses resultados.
- Com a dobra no número de núcleos, o tempo não consegue acompanhar linearmente a proporção.
- Irá sempre existir uma possível melhor otimização do código, se considerarmos o hardware utilizado na aplicação, correlacionando o tempo de execução do programa, a quantidade de núcleos do processador e a quantidade de threads da aplicação.

## 2.5. Semáforo

### 2.5.1. Produtor

O produto irá atuar procurando um espaço vazio na lista de memória. Encontrando, o valor 0 será substituído por um outro número aleatório.

A função executará em loop enquanto o número de “produced”, quantidade produzida, for menor que o “objectiveProduction”, quantidade total que será produzida. Essa região de código que realiza alteração na matriz e em variáveis globais é protegida por semáforo.

### 2.5.2. Consumidor

Enquanto o produtor procura um espaço vazio, o consumidor procura um espaço preenchido. Ao encontrá-lo o valor será lido e transferido para a função é primo, que analisará se o mesmo é um número primo ou não. O printf printa o resultado da função, sendo o mesmo desativado para cálculo de desempenho.

A função executará em loop enquanto o número de “consumed”, quantidade consumida, for menor que o “objectiveProduction”, quantidade total que será produzida. Essa região de código que realiza alteração na matriz e em variáveis globais é protegida por semáforo.

### 2.5.3. Execução

Para facilitar a execução a mesma foi realizada por um programa em python. Todos os resultados são salvos em um csv, sendo a inscrição no arquivo feita no main do programa Semáforo.

#### 2.5.4. Resultados

A partir dos dados inseridos no CSV foi calculada uma média de tempo para o agrupamento por  $N_p$ ,  $N_c$  e  $N$ . Esses valores foram representados no gráfico abaixo:

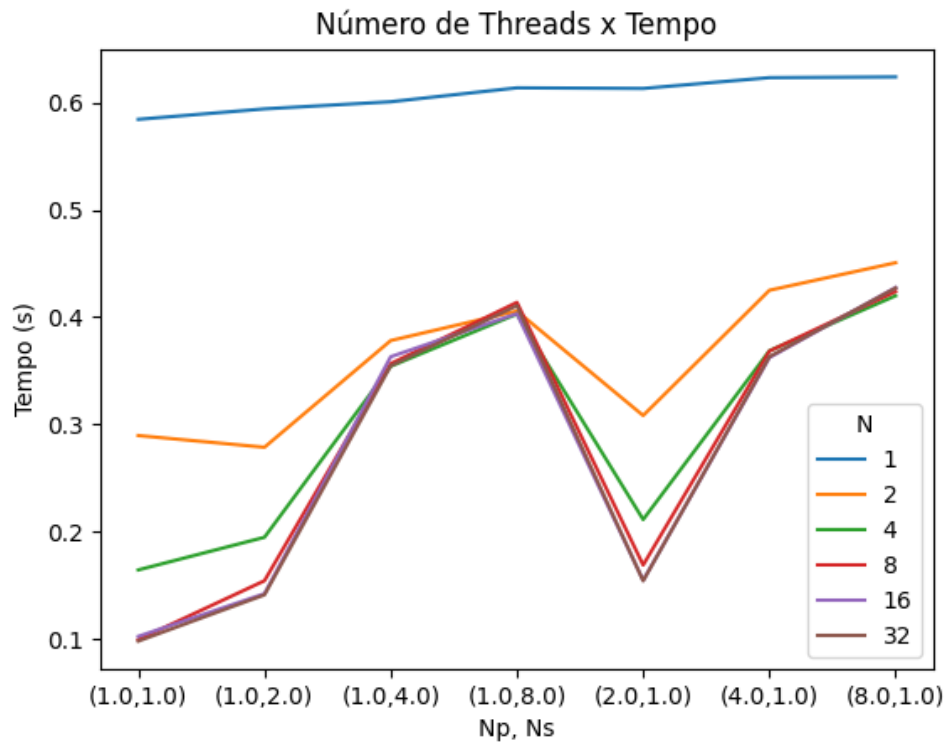


Figura 2: Gráfico de tempo x Número de Threads Produtor, Consumidor

Analisando o resultado foi percebido que os melhores resultados são para (1, 2) e (2, 1), o que não é esperado, visto que o desempenho deveria aumentar com o aumento da quantidade de consumidores. Isso pode estar ocorrendo pois a função de busca por uma posição vazia/ocupada está demandando muito tempo, sendo esse maior que o da troca de contexto, beneficiando um menor número de threads no código. A sugestão de melhoria para que isso não ocorra é a utilização de pilha, diminuindo o tempo de busca de  $O(n)$  para  $O(1)$ .