

# SIN 142 – SISTEMAS DISTRIBUÍDOS

## Trabalho Prático II: Controle Distribuído de Concorrência

**Nome:** Rodolfo Brizzi Junior

**Matrícula:** 6025

**Link do projeto:** <https://github.com/rodbrizzi/SIN142-Algoritmo-Exclusao-Mutua>

### 1. DESCRIÇÃO DO PROBLEMA

O problema de exclusão mútua distribuída ocorre em sistemas distribuídos, nos quais vários processos, potencialmente localizados em diferentes nós do sistema, precisam ter acesso exclusivo a um recurso compartilhado ou executar um trecho de código específico, chamado de região crítica. A principal condição a ser cumprida é que apenas um processo pode acessar essa região crítica em um determinado momento.

A natureza distribuída do sistema adiciona complexidade ao problema. Em um ambiente onde os processos não possuem um relógio comum ou memória compartilhada, coordenar esses processos para garantir a exclusão mútua se torna um desafio.

Sem mecanismos adequados de coordenação, a execução simultânea de seções críticas de código por vários processos pode resultar em condições de corrida, levando a estados indefinidos e inconsistentes no sistema. Em casos mais graves, isso pode levar à falha completa do sistema.

Portanto, é crucial que um algoritmo de exclusão mútua distribuída atenda a critérios específicos para evitar esses problemas. Esses critérios incluem segurança (garantindo que apenas um processo possa entrar na região crítica de cada vez), ausência de deadlock (um processo que requer acesso à região crítica eventualmente deve obtê-lo) e justiça (nenhum processo que requer acesso à região crítica é adiado indefinidamente).

Além disso, no problema proposto, é necessário que a quantidade "n" de processos envie solicitações para acessar a região crítica. Essas solicitações são avaliadas de acordo com um protocolo de controle de acesso. Quando um processo recebe autorização para acessar a região crítica, ele abre um arquivo chamado "resultado.txt" e registra informações relevantes. Especificamente, o processo escreve o seu PID (Identificador de Processo) e a hora atual do sistema no arquivo. Isso permite o registro e o acompanhamento do acesso de cada processo à região crítica, fornecendo uma visão detalhada do comportamento do sistema distribuído.

### 2. TECNOLOGIA UTILIZADA

Para implementar o algoritmo centralizado de exclusão mútua distribuída, usamos a linguagem de programação Python. Python é frequentemente utilizado para criar aplicativos em sistemas distribuídos porque facilita a comunicação em rede e a programação multithread.

Para a comunicação entre os processos na nossa solução, usamos o módulo de socket da biblioteca padrão do Python. Os sockets permitem a comunicação bidirecional entre processos, seja na mesma máquina ou em máquinas diferentes em uma rede. Cada processo cliente (criado pelo script "criador\_de\_processos.py") cria um socket para se comunicar com o coordenador da região crítica (implementado no script "coordenador\_de\_regiao\_critica.py"). Os processos enviam mensagens para o coordenador por meio desse socket para solicitar ou liberar acesso à região crítica. O coordenador também usa sockets para receber essas mensagens e enviar respostas.

A programação multithread é outra tecnologia importante em nossa solução. Threads são unidades de execução dentro de um processo que compartilham o mesmo espaço de memória, o que significa que podem acessar as mesmas variáveis e estruturas de dados. Em Python, usamos a biblioteca padrão "threading" para criar e gerenciar threads. Em nossa solução, cada processo é implementado como uma thread separada no script "criador\_de\_processos.py". Da mesma forma, o coordenador da região crítica usa threads separadas para aceitar conexões de entrada e processar as mensagens recebidas.

Também usamos o módulo "queue" da biblioteca padrão do Python para implementar a fila de processos que estão aguardando acesso à região crítica. Essa fila permite que o coordenador mantenha a ordem das solicitações de acesso, garantindo uma distribuição justa do acesso à região crítica.

Para gerenciar sinais, usamos o módulo "signal", que permite que o programa responda a sinais do sistema operacional, como encerrar o programa de forma controlada quando o usuário pressiona Ctrl+C.

Por fim, usamos o módulo "os" para lidar com operações dependentes do sistema operacional, como limpar a tela do terminal para a interface do usuário do coordenador.

Em resumo, o Python e seus módulos padrão forneceram todas as funcionalidades necessárias para implementar a solução do problema de exclusão mútua distribuída.

### **3. SOLUÇÃO UTILIZADA**

A solução desenvolvida tem como base dois componentes principais: o "coordenador\_de\_regiao\_critica.py" e o "criador\_de\_processos.py". Ambos scripts são escritos em Python e se utilizam de funcionalidades específicas da linguagem para lidar com a problemática de coordenação de acessos a uma região crítica em um sistema distribuído. A seguir, detalharemos cada um desses componentes.

O script "coordenador\_de\_regiao\_critica.py" implementa a funcionalidade do coordenador de região crítica. Esse componente é responsável por controlar o acesso dos processos à região crítica. Para isso, o coordenador possui um servidor de socket TCP que escuta por requisições de processos e responde de acordo com os comandos enviados. A classe Coordenador é inicializada com uma thread de processamento e uma thread de interface. A thread de processamento aguarda por conexões de entrada e para cada uma cria uma nova thread para manipular a comunicação com o processo. A thread de interface provê um menu de interação para o usuário, onde é possível verificar a fila FIFO atual de processos, e também a quantidade de Grants que cada processo recebeu.

O programa para se comunicar utiliza três tipos de mensagens:

1. Request: Mensagem enviada por um processo para requisitar acesso à região crítica.
2. Grant: Mensagem enviada pelo coordenador para conceder acesso à região crítica.
3. Release: Mensagem enviada por um processo ao sair da região crítica.

As mensagens são enviadas usando um separador que é necessário para identificar o tipo de mensagem e o PID do processo. O formato utilizado é o seguinte: X|PID|0000, onde X representa o tipo de mensagem e PID é o número de identificação do processo.

O método “handle\_client” dentro da classe Coordenador recebe mensagens dos processos e as processa de acordo com o prefixo da mensagem. Uma mensagem prefixada por 1(REQUEST) é uma requisição de acesso à região crítica e uma mensagem prefixada por 3(RELEASE) é uma liberação de acesso à região crítica.

A classe Coordenador mantém uma fila (implementada com a classe “Queue” de Python) para gerenciar as requisições de acesso à região crítica. Quando recebe uma requisição, se a fila estiver vazia, o coordenador concede imediatamente o acesso ao processo (enviando a mensagem 2-GRANT) e adiciona a requisição à fila. Se a fila não estiver vazia, o coordenador simplesmente adiciona a requisição à fila. Quando recebe uma mensagem de liberação, o coordenador remove a requisição no início da fila e concede acesso ao próximo processo na fila (se houver).

O script “criador\_de\_processos.py” cria um número especificado de processos e cada processo tenta acessar a região crítica um número especificado de vezes. Cada processo é implementado como uma thread separada, usando a classe Processo que é uma subclasse de “threading.Thread”. A região crítica é a escrita em um arquivo chamado resultado.txt.

O processo tenta acessar a região crítica enviando uma mensagem de requisição para o coordenador (REQUEST) e espera pela resposta do coordenador. Se a resposta for uma concessão de acesso (GRANT), o processo entra na região crítica e escreve no arquivo resultado.txt. Após sair da região crítica, o processo envia uma mensagem de liberação (RELEASE) para o coordenador.

Cada processo quando acessa a região crítica, registra seu PID e também a hora atual do sistema. O tempo total gasto por todos os processos é calculado no final.

Esta solução aborda a problemática de coordenação de acessos a uma região crítica em um sistema distribuído. Ela é uma implementação de um algoritmo de exclusão mútua distribuída, que garante que, em qualquer momento, apenas um processo tenha acesso à região crítica.

## **4. AVALIAÇÃO E TESTES**

Para realizar os testes, utilizamos inicialmente uma máquina com as seguintes configurações:

- Sistema operacional: Windows 11
- Processador: Ryzen 5 5500
- Memória RAM: 32 GB
- Armazenamento: 1 TB de SSD

Em seguida, para demonstrar que o programa funciona em diferentes máquinas, recorreremos ao Oracle VM VirtualBox para criar duas máquinas virtuais com o sistema operacional Ubuntu 20.04, onde o “coordenador\_da\_regiao\_critica.py” é executado em uma

máquina, e o “criador\_de\_processos.py” é executado em outra. A placa de rede das duas máquinas virtuais foi configurada no modo Bridge.

#### 4.1. Máquina Física (Windows 11)

Para testar o programa em uma máquina física, basta executar os scripts em terminais separados na seguinte sequência:

1. coordenador\_de\_regiao\_critica.py
2. criador\_de\_processos.py

Realizamos o primeiro teste para verificar o correto funcionamento com os seguintes valores de entrada:

- $n = 2$  processos;
- $r = 10$  vezes;
- $k = 1$  segundo.

A escrita no arquivo resultado.txt é apresentada na figura a seguir. É importante mencionar que a primeira linha do arquivo é um cabeçalho com informações, portanto o número total de linhas que os processos escreverão no arquivo será sempre  $(n \times r) + 1$ .

```
1  PID      | Hora Atual
2  112745   | 20:40:02.106
3  043843   | 20:40:03.106
4  112745   | 20:40:04.107
5  043843   | 20:40:05.108
6  112745   | 20:40:06.109
7  043843   | 20:40:07.110
8  112745   | 20:40:08.111
9  043843   | 20:40:09.112
10 112745   | 20:40:10.113
11 043843   | 20:40:11.114
12 112745   | 20:40:12.115
13 043843   | 20:40:13.116
14 112745   | 20:40:14.117
15 043843   | 20:40:15.118
16 112745   | 20:40:16.119
17 043843   | 20:40:17.120
18 112745   | 20:40:18.121
19 043843   | 20:40:19.122
20 112745   | 20:40:20.123
21 043843   | 20:40:21.124
22
23 Quantidade de processos(n): 2
24 Quantidade de vezes que cada processo executou a região crítica(r): 10
25 Tempo definido em segundos para cada execução na região crítica(k): 1
26
27 Tempo total para geração do arquivo 'resultado.txt': 19.018 segundos
```

Ao interagir com a interface do coordenador, conseguimos visualizar a fila de processos em exibição. A seguir, são apresentadas duas imagens que mostram diferentes momentos da fila de processos, demonstrando a troca de posições dos processos na fila.

```
Fila de Pedidos:

PID      | POSIÇÃO
854816   | 1ª
463583   | 2ª

Aperte qualquer tecla para voltar ao menu...
```

```

Fila de Pedidos:

PID | POSIÇÃO
463583 | 1º
854816 | 2º

Aperte qualquer tecla para voltar ao menu...

```

Além disso, é possível visualizar a lista de todos os processos que receberam acesso liberado à região crítica (GRANT). A imagem a seguir foi capturada no final da execução do teste, evidenciando que ambos os dois processos receberam a quantidade correta de GRANTS (10).

```

PID | QUANTIDADE DE GRANTS
854816 | 10
463583 | 10

Aperte qualquer tecla para voltar ao menu...

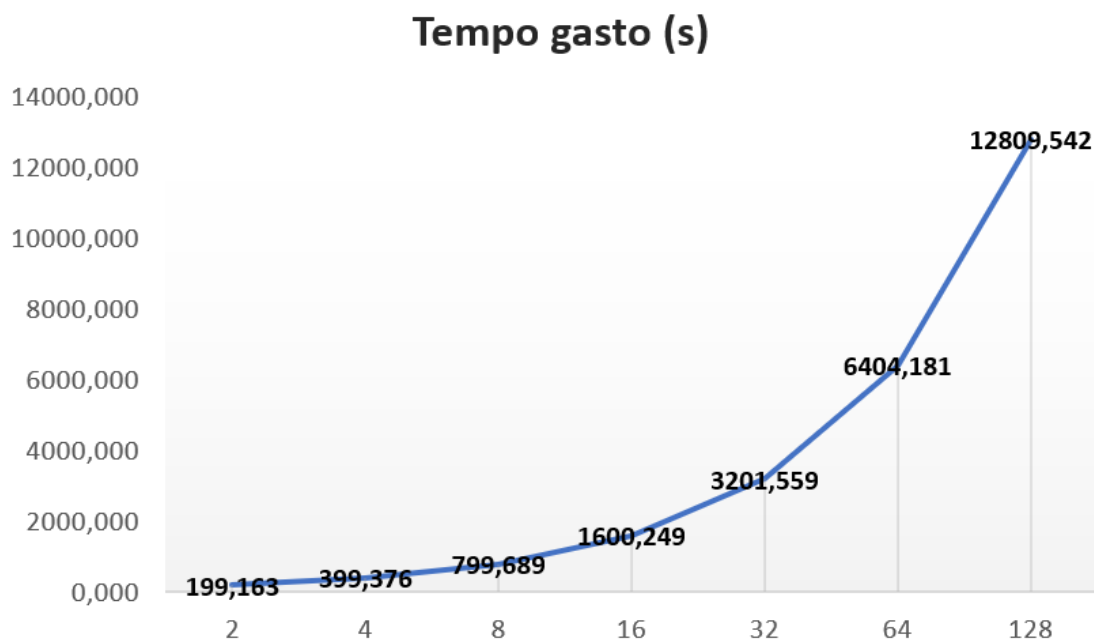
```

#### 4.1.1. Teste 1 de escalabilidade

No primeiro teste de escalabilidade, utilizamos os seguintes valores de entrada:

- n processos, onde n pertence ao conjunto {2, 4, 8, 16, 32, 64, 128};
- r = 100 vezes;
- k = 1 segundo.

Calculamos o tempo gasto para cada quantidade n de processos e registramos os dados para criar o seguinte gráfico:



A seguir é mostrada a captura de tela do teste usando 128 processos, que demonstra um bom funcionamento. O número total de acessos à região crítica do teste deve ser de 100 para cada processo, totalizando 12.800 acessos em 12.809,542 segundos.

```

12792 673471 | 10:26:53.991
12793 398832 | 10:26:54.993
12794 310408 | 10:26:55.994
12795 155128 | 10:26:56.994
12796 397296 | 10:26:57.995
12797 104141 | 10:26:58.996
12798 364754 | 10:26:59.997
12799 637085 | 10:27:00.997
12800 955931 | 10:27:01.999
12801 288614 | 10:27:02.999
12802
12803 Quantidade de processos(n): 128
12804 Quantidade de vezes que cada processo executou a região crítica(r): 100
12805 Tempo definido em segundos para cada execução na região crítica(k): 1
12806
12807 Tempo total para geração do arquivo 'resultado.txt': 12809.542 segundos

```

#### 4.1.2. Teste 2 de escalabilidade

No segundo teste de escalabilidade, utilizamos os seguintes valores de entrada:

- n processos, onde n pertence ao conjunto {2, 4, 8, 16, 32, 64, 128};
- r = 1000 vezes;
- k = 0 segundos.

Calculamos o tempo gasto para cada quantidade n de processos e registramos os dados para criar o seguinte gráfico:



A seguir é mostrada a captura de tela do teste usando 128 processos, que demonstra um bom funcionamento. O número total de acessos à região crítica do teste deve ser de 1000 para cada processo, totalizando 128.000 acessos em 31,104 segundos.

```

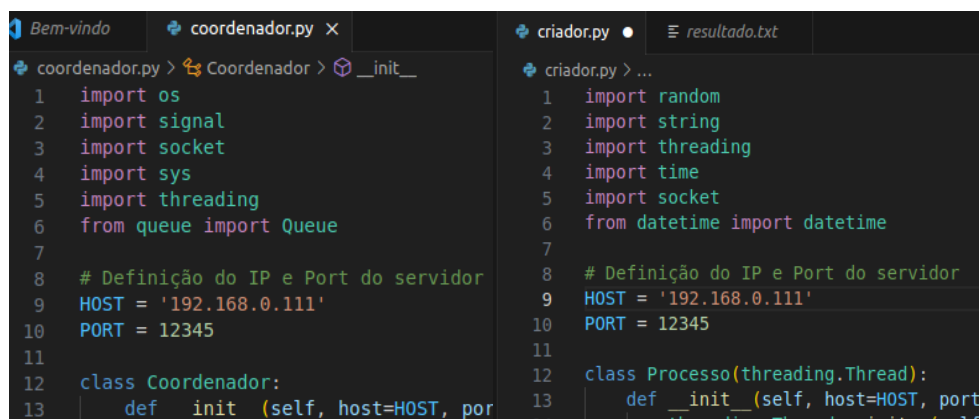
127992 259425 | 21:26:24.944
127993 716311 | 21:26:24.944
127994 891631 | 21:26:24.944
127995 669289 | 21:26:24.945
127996 925678 | 21:26:24.945
127997 789970 | 21:26:24.945
127998 281020 | 21:26:24.946
127999 765639 | 21:26:24.946
128000 947782 | 21:26:24.946
128001 463774 | 21:26:24.947
128002
128003 Quantidade de processos(n): 128
128004 Quantidade de vezes que cada processo executou a região crítica(r): 1000
128005 Tempo definido em segundos para cada execução na região crítica(k): 0
128006
128007 Tempo total para geração do arquivo 'resultado.txt': 31.104 segundos

```

## 4.2. Máquinas Virtuais (Ubuntu 20.04)

Para executar o programa em máquinas diferentes, é necessário seguir as seguintes etapas:

1. É importante fazer uma pequena modificação nos códigos, especificamente nas variáveis globais "HOST" e "PORT", para informar o endereço IP e a porta do servidor.
2. Como exemplo, suponhamos que a máquina que executará o "coordenador\_de\_regiao\_critica.py" tenha o endereço IP 192.168.0.111.
3. Portanto, nos dois códigos, é necessário inserir o endereço IP e a porta da máquina que executará o "coordenador\_de\_regiao\_critica.py" (servidor) nas variáveis globais "HOST" e "PORT".
4. Na primeira máquina, execute o arquivo "coordenador\_de\_regiao\_critica.py".
5. Em seguida, na segunda máquina, execute o arquivo "criador\_de\_processos.py".

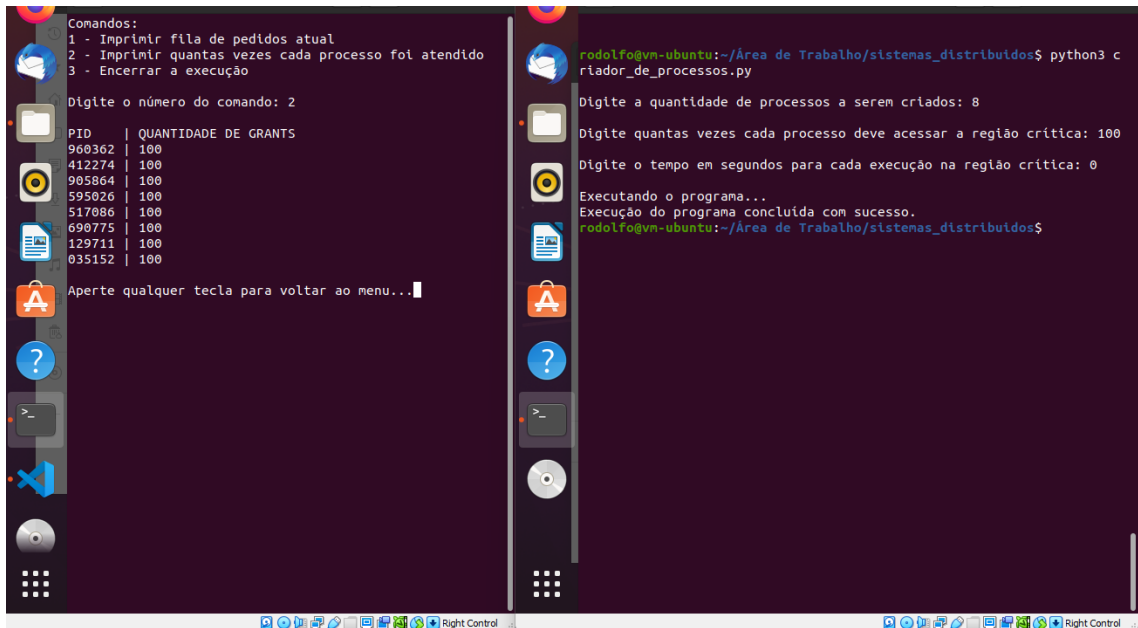


```

Bem-vindo  coordenador.py x  criador.py  resultado.txt
coordenador.py > Coordenador > __init__
1 import os
2 import signal
3 import socket
4 import sys
5 import threading
6 from queue import Queue
7
8 # Definição do IP e Port do servidor
9 HOST = '192.168.0.111'
10 PORT = 12345
11
12 class Coordenador:
13     def __init__(self, host=HOST, port=PORT):
14
criador.py > ...
1 import random
2 import string
3 import threading
4 import time
5 import socket
6 from datetime import datetime
7
8 # Definição do IP e Port do servidor
9 HOST = '192.168.0.111'
10 PORT = 12345
11
12 class Processo(threading.Thread):
13     def __init__(self, host=HOST, port=PORT):
14         super().__init__()

```

A imagem a seguir mostra as duas máquinas virtuais executando o programa. No teste mostrado foram utilizados como parâmetros 8 processos, 100 entradas na região crítica e 0 segundos.



A seguir está a imagem que mostra o estado atual do arquivo "resultado.txt" após a execução bem-sucedida do programa em diferentes máquinas virtuais.

```

Abrir  resultado.txt  Salvar
~/Área de Trabalho/sistemas_distribuidos

790 690775 | 19:18:45.252
791 517086 | 19:18:45.254
792 595026 | 19:18:45.255
793 905864 | 19:18:45.259
794 412274 | 19:18:45.272
795 960362 | 19:18:45.275
796 129711 | 19:18:45.276
797 517086 | 19:18:45.299
798 595026 | 19:18:45.301
799 960362 | 19:18:45.319
800 412274 | 19:18:45.323
801 595026 | 19:18:45.346
802
803 Quantidade de processos(n): 8
804 Quantidade de vezes que cada processo executou a região crítica(r): 100
805 Tempo definido em segundos para cada execução na região crítica(k): 0
806
807 Tempo total para geração do arquivo 'resultado.txt': 5.061 segundos

```

## 5. POSSÍVEIS ABORDAGENS ALTERNATIVAS

A solução apresentada é apenas uma das muitas possíveis abordagens para resolver este problema. Outras abordagens poderiam incluir:

- Utilizando outras bibliotecas de concorrência: Python tem outras bibliotecas para lidar com concorrência, como `asyncio` e `multiprocessing`. Essas bibliotecas poderiam ser usadas para gerir a concorrência de forma diferente.
- Implementando um protocolo de eleição: Em vez de ter um coordenador fixo, os processos poderiam eleger dinamicamente um coordenador entre eles sempre que necessário.
- Utilizando uma abordagem baseada em mensagens: Em vez de usar sockets diretamente, poderíamos usar um sistema de mensagens, como RabbitMQ ou Kafka, para gerenciar a comunicação entre os processos.



## **6. LIÇÕES APRENDIDAS**

Durante o desenvolvimento deste projeto, aprendi a importância do manejo correto de recursos do sistema, como conexões de rede e threads. O encerramento inadequado de recursos pode levar a problemas complexos e difíceis de diagnosticar. Além disso, este projeto destacou a importância de uma compreensão sólida dos conceitos de programação concorrente e distribuída.

Por último, quero expressar minha gratidão ao Prof. Dr. Rodrigo Moreira pelos ensinamentos durante o curso da disciplina de Sistemas Distribuídos. Foi uma experiência valiosa que me proporcionou conhecimento sobre o funcionamento dos sistemas que encontramos atualmente em nosso dia a dia.