UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO

AMIN RICHAM SAMIR SOBH
202110033611

MARCO ANDRÉ SANTOS DA COSTA JÚNIOR
201820323611

THIAGO OLIVEIRA DA SILVA
202210032311

DAVI NUNES SANTOS
201610054111

SISTEMAS OPERACIONAIS 1
TRABALHO FINAL: Projeto Arena de Robôs

2025.1

SUMÁRIO

Propósito

O propósito deste trabalho é a criação de um projeto que simula um jogo de luta entre robôs. Aplicaremos nosso aprendizado em Sistemas Operacionais 1 com respeito a threads, processos, compartilhamento de memória entre outros conceitos importantes.

Resumo instrucional - Arena dos Processos - Batalha dos Robôs Autônomos

- Criar um jogo em modo texto, totalmente distribuído, usando processos e threads locais.
- Tabuleiro (40x20), barreiras, baterias e metadados dos robôs ficam em memória compartilhada.
- Cada robô sendo um processo que lê/escreve diretamente na memória compartilhada, usando mecanismos de sincronização.
- Duelos entre robôs adjacentes devem ser resolvidos dentro de uma região crítica protegida.
- Implementação obrigatória de no mínimo 4 robôs independentes (um sendo o "robô do jogador").
- Cada robô deve ter duas threads: `sense_act` (decide a ação) e `housekeeping` (atualiza energia, log, locks).
- A memória compartilhada deve conter o `GRID[40][20]`, um array de robôs (`robots[]`) e flags auxiliares.
- Locks obrigatórios incluem `grid_mutex`, `robots_mutex` e `battery_mutex_k` (um por bateria). A ordem de aquisição deve ser documentada.
- Atributos dos robôs: *Força (1-10), Energia (10-100, consome 1 E por movimento, +20 E ao coletar, máx. 100), Velocidade (1-5, nº de células a tentar mover)*.
- Duelos: *Poder = 2F+E*. O robô com maior poder vence, o perdedor morre. Em caso de empate, ambos são destruídos. A negociação do duelo deve ser dentro do `grid_mutex`.

- Ciclo de vida do robô: Inicialização (primeiro processo gera o cenário), Loop principal (sense_act: snapshot, decide ação, adquire locks, executa ação, libera locks), e housekeeping (reduz energia, grava log, checa vitória).
- Componente de visualização que renderiza o GRID em tempo real a cada 50-200ms e termina quando o jogo acaba.
- Criação de deadlock real e implementar prevenção, detecção ou recuperação no relatório.

Atribuição de tarefas dos integrantes

A seguir, apresentamos como foi realizada a divisão de tarefas para a realização desse trabalho:

Parte 1 - Infraestrutura Base: Memória Compartilhada + Locks

AMIN RICHAM SAMIR SOBH

Responsabilidades:

- Criar e configurar o segmento de memória compartilhada com:
- GRID[40][20]
- robots[] com atributos (ID, F, E, V, posição, status)
- Flags auxiliares (init_done, vencedor, etc.)
- Implementar os mutexes:
- grid_mutex
- robots_mutex
- battery_mutex_k ou campo de “dono”
- Garantir ordem de aquisição e evitar corrupção/dados inconsistentes

Commits: criação da infraestrutura, testes de leitura/escrita, integração com os robôs e viewer.

Parte 2 – Robôs e Threads: Lógica dos Robôs

THIAGO OLIVEIRA DA SILVA

Responsabilidades:

- Criar ≥4 processos robôs, com um sendo o “robô do jogador”
- Criar duas threads por robô:
- sense_act: decide ações com base em snapshot
- housekeeping: reduz energia, registra log, verifica fim de jogo
- Implementar movimentação, coleta de bateria e duelos
- Garantir uso correto dos locks e acesso seguro ao grid

Commit inicial: Estrutura dos processos e threads, movimentação, lógica de duelos e integração com memória.

Parte 3 - Visualizador (viewer)

MARCO ANDRÉ SANTOS DA COSTA JÚNIOR

Responsabilidades:

- Criar processo ou thread passiva que:
- Tira snapshot do GRID a cada 50-200ms
- Renderiza o tabuleiro ASCII com ANSI ou ncurses
- Finaliza quando game_over for sinalizado
- Não pode modificar o estado compartilhado

Commit inicial: Visualizador lendo o GRID e implementação do loop de exibição dos caracteres.

Parte 4 - Deadlock + Relatório

DAVI NUNES SANTOS

Responsabilidades:

- Criar situação real de deadlock entre dois robôs
- Demonstrar com logs
- Implementar prevenção, detecção ou recuperação
- Escrever e organizar o relatório final explicando:
- Arquitetura geral
- Estratégia contra deadlock
- Resultados e prints

Commits: cenário de deadlock, solução, relatório e README.

Modularização do programa

Estrutura

Nessa parte vamos estabelecer a base do jogo, criando a memória compartilhada para o `GRID[40][20]`, os atributos dos robôs (`robots[]`) e as flags auxiliares. Além disso, vamos implementar os mutexes (`grid_mutex`, `robots_mutex`, `battery_mutex_k`) e garantir a ordem de aquisição para evitar corrupção e inconsistência de dados.

Códigos:

main.py

```
from multiprocessing import Lock, Array, Process
from robots import processo_robo
import shared_struct as ss
from shared_struct import RoboShared
import gridFunctions
import flagsFunctions

def main():
    grid = Array(typecode_or_type='u',
size_or_initializer=ss.GRID_SIZE, lock=True)
    grid_mutex = Lock()

    robots = Array(RoboShared, ss.QTD_ROBOS, lock=True)  # CORRETO
    robots_mutex = Lock()

    flags = Array(typecode_or_type= "i",
size_or_initializer=(ss.QTD_FLAGS), lock=True)
    flags_mutex = Lock()

    battery_mutex = [Lock() for _ in range (ss.QTD_BATERIAS)]

    gridFunctions.iniciaGrid(grid)
    gridFunctions.adicionaElementos(grid)
    flagsFunctions.initFlags(flags)

    processos = []
    for i in range(ss.QTD_ROBOS):
        processo = Process(
            target=processo_robo,
            args=(i, grid, flags, robots, robots_mutex, grid_mutex)
        )
        processo.start()
        processos.append(processo)
```

```
    for p in processos:
        p.join()


if __name__ == '__main__':
    main()
```

---

shared_struct.py

```
import ctypes

WIDTH = 40
HEIGHT = 20
GRID_SIZE = WIDTH * HEIGHT
TOTAL_SIZE = GRID_SIZE + 4  # 4 bytes para o inteiro game_over
QTD_ROBOS = 2
QTD_FLAGS = 3
QTD_BATERIAS = 15

class RoboShared(ctypes.Structure):
    _fields_ = [
        ("ID", ctypes.c_int),
        ("forca", ctypes.c_int),
        ("energia", ctypes.c_int),
        ("velocidade", ctypes.c_int),
        ("posicao_x", ctypes.c_int),
        ("posicao_y", ctypes.c_int),
        ("status", ctypes.c_char),  # 'V', 'M'
    ]
```

Lógica

Aqui desenvolvemos a inteligência e o comportamento dos robôs. Os 4 processos robôs, incluindo o "robô do jogador". Para cada robô, implementamos as threads `sense_act` (para decidir ações baseadas em um snapshot do grid) e `housekeeping` (para gerenciar energia, logs e checar a condição de vitória). Também organizamos a mecânica de movimentação (que consome 1 E por movimento), coleta de bateria (que adiciona +20 E, até o máximo de 100) e os duelos.

Códigos:

robots.py

```
import random
import multiprocessing as mp
import threading
import shared_struct as ss
import gridFunctions
from flagsFunctions import getFlagGameOver
```

```
import time


# Criação dos locks mutex
grid_mutex = mp.Lock()


class Robot:
    def __init__(self, ID, F, E, V, posicao_x, posicao_y, status, grid,
flags, robots_shared, robots_mutex, grid_mutex):
        self.ID = ID
        self.forca = F
        self.energia = E
        self.velocidade = V
        self.posicao_x = posicao_x
        self.posicao_y = posicao_y
        self.status = status # Vivo ou morto
        self.log = [] # Log de ações do robô
        self.grid = grid
        self.flags = flags
        self.robots = robots_shared  # Lista de robôs compartilhada
        self.robots_mutex = robots_mutex
        self.grid_mutex = grid_mutex  # Mutex para proteger o grid

        # Adiciona o robô na grid se a posição estiver vazia
        with self.grid_mutex: # Protegendo o grid
            self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))

    def get_index(self, x, y):
        """Calcula o índice do grid baseado na posição x e y."""
        return y * ss.WIDTH + x

    def get_grid(self, x, y):
        """Obtém o valor do grid na posição (x, y)."""
        return self.grid[self.get_index(x, y)]

    def set_grid(self, x, y, value):
        """Define o valor do grid na posição (x, y)."""
        index = self.get_index(x, y)
        self.grid[index] = value

    def mover(self):
        if self.energia <= 0 or self.status != b"V":
            print(f"Robô {self.ID} sem energia ou morto.")
            return

        # Buscar bateria mais proxima
        bateria_mais_proxima = None
```

```
        menor_distancia = float('inf')
        for y in range(ss.HEIGHT):
            for x in range(ss.WIDTH):
                if self.get_grid(x, y) == "B":
                    distancia = abs(self.posicao_x - x) +
abs(self.posicao_y - y)
                    if distancia < menor_distancia:
                        menor_distancia = distancia
                        bateria_mais_proxima = (x, y)

        # Se encontrou uma bateria, move-se em direção a ela
        if bateria_mais_proxima:
            bateria_x, bateria_y = bateria_mais_proxima
            dx = 0
            dy = 0
            # Verifica a direção para se mover em direção à bateria
            # Se o x da bateria for maior que o x do robô, move para a
direita (dx = 1)
            if self.posicao_x < bateria_x:
                dx = 1
            # Se o x da bateria for menor que o x do robô, move para a
esquerda (dx = -1)
            elif self.posicao_x > bateria_x:
                dx = -1
            # se for 0, quer dizer que o robô está na mesma linha da
bateria
            if dx == 0:
                if self.posicao_y < bateria_y:
                    dy = 1
                elif self.posicao_y > bateria_y:
                    dy = -1
        else:
            # Movimento aleatório se não houver bateria próxima
            dx, dy = random.choice([(0, 1), (1, 0), (0, -1), (-1, 0)])

        nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x + dx))
        nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y +
dy))

        # Movimento aleatório do robo
        #dx, dy = random.choice([(0, 1), (1, 0), (0, -1), (-1, 0)])  #
Cima, baixo, esquerda, direita
        # Definindo nova posição
        #nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x +
dx)) # Verificação para não sair do grid e chegar no ultimo bloco
        #nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y +
dy))
```

```
        with self.grid_mutex:
            # Para onde o robo quer se mover
            destino = self.get_grid(nova_posicao_x, nova_posicao_y)
            if destino == "-":
                # Atualiza a posição do robô na grid
                self.set_grid(self.posicao_x, self.posicao_y, "-") #
Limpa a posição antiga
                self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y # Define a nova posição do robô
                self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID)) # Atualiza a grid com a nova posição do robô (0, 5, id) -
(x, y, id)
                self.energia -= 1 # 1 de energia por movimento
                self.log.append(f"Robo {self.ID} se moveu para
({self.posicao_x}, {self.posicao_y}). Energia restante:
{self.energia}.")

            elif destino == "B": # RAIO ⚡
                self.set_grid(self.posicao_x, self.posicao_y, "-") #
Limpa a posição antiga
                self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y # Define a nova posição do robô
                self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID)) # Atualiza a grid com a nova posição do robô
                self.energia = min(100, self.energia + 20) # Recarrega
a energia garantindo que não ultrapasse 100
                self.log.append(f"Robo {self.ID} encontrou uma bateria
e recarregou. Energia atual: {self.energia}.")


            elif destino == "#": # BARREIRA #
                self.log.append(f"Robo {self.ID} encontrou uma barreira
em ({nova_posicao_x}, {nova_posicao_y}) e não pôde se mover.")

            elif destino.isdigit() and destino != str(self.ID): # Se a
posição já estiver ocupada por outro robô
                self.duelar(int(destino))

    def duelar(self, outro_robo_id):
        outro_robo = self.robots[outro_robo_id]

        # Confirmar que ainda estão vivos
        if self.status != b"V" or outro_robo.status != b"V":
            return

        # Lógica de duelo
        if self.forca > outro_robo.forca:
```

```
            self.log.append(f"Robo {self.ID} venceu o duelo contra Robo
{outro_robo_id}.")
            with self.robots_mutex:
                outro_robo.status = b"M"
            with self.grid_mutex:
                self.set_grid(outro_robo.posicao_x,
outro_robo.posicao_y, "-")

        elif self.forca < outro_robo.forca:
            self.log.append(f"Robo {self.ID} perdeu o duelo contra Robo
{outro_robo_id}.")
            self.status = b"M"
            with self.robots_mutex:
                self.robots[self.ID].status = b"M"
            with self.grid_mutex:
                self.set_grid(self.posicao_x, self.posicao_y, "-")

        else:
            self.log.append(f"Robo {self.ID} e Robo {outro_robo_id}
empataram no duelo.")
            self.status = b"M"
            with self.robots_mutex:
                outro_robo.status = b"M"
                self.robots[self.ID].status = b"M"
            with self.grid_mutex:
                self.set_grid(self.posicao_x, self.posicao_y, "-")
                self.set_grid(outro_robo.posicao_x,
outro_robo.posicao_y, "-")

    def sense_act(self):
        """Método para o robô sentir o ambiente e agir."""
        while self.status == b"V" and getFlagGameOver(self.flags) == 0:
            self.mover()
            tempo_espera = self.velocidade * 0.2
            time.sleep(tempo_espera)  # Simula o tempo de espera
baseado na velocidade do robô

    def housekeeping(self):
        """Método para o robô realizar tarefas de manutenção."""
        while self.status == b"V" and getFlagGameOver(self.flags) == 0:
            self.energia -= 1
            if self.energia <= 0:
                self.status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                self.log.append(f"Robo {self.ID} ficou sem energia e
foi desligado.")
            time.sleep(1)
```

```
    def iniciar(self):
        t1 = threading.Thread(target=self.sense_act)
        t2 = threading.Thread(target=self.housekeeping)
        t1.start()
        t2.start()
        t1.join()
        t2.join()

        self.log.append(f"Robo {self.ID} finalizou suas atividades.")
        for linha in self.log:
            print(linha)

def processo_robo(ID, grid, flags, robots_shared, robots_mutex,
grid_mutex):
    """Função para iniciar o processo do robô."""
    # Inicializa o robô com valores aleatórios
    F = random.randint(1, 10)  # Força
    E = random.randint(50, 100)  # Energia
    V = random.randint(1, 5)  # Velocidade
    posicao_x = random.randint(0, ss.WIDTH - 1)
    posicao_y = random.randint(0, ss.HEIGHT - 1)

    robo = Robot(ID, F, E, V, posicao_x, posicao_y, b"V", grid, flags,
robots_shared, robots_mutex, grid_mutex)
    robo.iniciar()
```

---

flagsFunctions.py

```
"""
FLAGS
init_done - indica se terminou o processo de inicio do jogo
vencedor - indidca o vencendor
Game over - indica se acabou
"""
def initFlags(flags):
    flags[0] = 0
    flags[1] = -1
    flags[2] = 0

def getFlagInitDone(flags):
    return flags[0]

def setFlagInitDone(flags, data):
    flags[0] = data

def getFlagVencedor(flags):
```

```
    return flags[1]

def setFlagVencedor(flags, idWinner):
    flags[1] = idWinner

def getFlagGameOver(flags):
    return flags[2]

def setFlagGameOver(flags, data):
    flags[2] = data
```

Visualização

Aqui temos a criação do viewer, o componente de visualização passivo onde tiramos os snapshots e renderizamos o tabuleiro.

Códigos:

viewer.py

```
import time
import os
from multiprocessing import shared_memory
from shared_struct import WIDTH, HEIGHT, GRID_SIZE, TOTAL_SIZE

def render(grid_bytes):
    os.system("cls" if os.name == "nt" else "clear")
    print("=== ARENA DOS ROBÔS ===")
    for i in range(HEIGHT):
        linha = grid_bytes[i*WIDTH:(i+1)*WIDTH].decode('utf-8')
        print(linha)
    print()

def main():
    try:
        shm = shared_memory.SharedMemory(name="arena_dos_robos")
    except FileNotFoundError:
        print("Memória compartilhada não encontrada.")
        return
```

```
    while True:
        buffer = shm.buf[:TOTAL_SIZE]
        grid_bytes = bytes(buffer[:GRID_SIZE])
        game_over_flag = int.from_bytes(buffer[GRID_SIZE:GRID_SIZE+4],
"little")

        render(grid_bytes)

        if game_over_flag == 1:
            print("Fim de jogo detectado.")
            break

        time.sleep(0.2)

    # Libera referências antes de fechar
    del grid_bytes
    del buffer
    shm.close()

if __name__ == "__main__":
    main()
```

---

gridFunctions.py

```
import shared_struct as ss
from random import randint
from multiprocessing import Array

def iniciaGrid(grid):
    for i in range(len(grid)):
        grid[i] = "-"

def printGrid(grid):
    for i in range (len(grid)):
        print(grid[i], end = " ")
        if ((i+1) % 40) == 0:
            print()

def adicionaBaterias(grid):
    for _ in range (ss.QTD_BATERIAS):
        posicao = randint(0, 799)
        grid[posicao] = "B"

def adicionaBarreiras(grid):
    qtdBarreiras = randint(0,15)
#    print(qtdBarreiras)
```

```
    for _ in range (qtdBarreiras):
        posicao = randint(0, 799)
        grid[posicao] = "#"

def adicionaElementos(grid):
    adicionaBarreiras(grid)
    adicionaBaterias(grid)

def escreveNoArq(arq, qtdBaterias):
    with open(arq, "a") as arquivo:
        arquivo.write(f"QTD_BATERIAS = {qtdBaterias}")
```

Conceitos abordados de Sistemas Operacionais

Memória Compartilhada: Entendimento de como a memória é alocada e acessada por múltiplos processos.

Criação e Gerenciamento de Threads: Criação e controle de múltiplas threads dentro de um processo (`threading.Thread`).

Comunicação entre Threads/Processos: Conhecimento de como as threads e processos acessam a memória compartilhada e utilizam os locks para isso

Sincronização de Processos/Threads: Domínio do uso de mutexes (ex: `grid_mutex`, `robots_mutex`, `battery_mutex_k`), semáforos e outras primitivas para evitar condições de corrida, corrupção de dados e deadlocks.

Regiões Críticas: Compreensão do que são e como proteger o acesso a elas.

Ordenação e Aquisição de Locks: Conhecer a importância de uma ordem definida para adquirir locks para prevenir deadlocks.

Deadlock

Um deadlock trata-se de um “travamento” no programa causado por um grande impasse na aquisição de recursos. Quando há uma encruzilhada fechada entre demanda e detenção de recursos envolvendo os processos.

Teoria

Sabemos que 4 condições caracterizam os deadlocks

1. **Exclusão mútua:** Recursos que não podem ser compartilhados ao mesmo tempo, regiões exclusivas que requerem um processo por vez.
2. **Posse e espera:** Quando um processo detém o recurso sem abrir mão deste, impedindo que outros acessem
3. **Não existência de preempção:** Pois, sem preempção, não há troca de contexto involuntária na qual o sistema possa administrar e gerenciar devidamente.
4. **Espera circular:** Quando existe uma cadeia fechada de dois ou mais processos, onde cada processo detém algo que o outro deseja.

Nosso contexto

Se o robô decide mover, ele adquire os locks necessários (por exemplo, grid_mutex para modificar a célula do GRID). Ele então altera a célula de onde ele estava para vazio e a célula para onde ele se moveu para seu ID. Se coletar uma bateria, ele pegaria o battery_mutex_k da célula da bateria e alteraria o GRID. Se for duelar, a lógica de duelo ocorre dentro do grid_mutex

Dentro dos métodos da classe Robot (como mover, duelar), teremos self.shared_grid_mutex.acquire() e self.shared_grid_mutex.release(), onde self.shared_grid_mutex é a instância do lock que foi passada no seu construtor.

Cenário

- **Robô A:** Trava `battery_mutex_k` → Tenta `grid_mutex`.
- **Robô B:** Trava `grid_mutex` → Tenta `battery_mutex_k`.

Dois robôs (Robô A e Robô B) estando adjacentes a uma bateria e ambos com intenção de usufruí-la.
Precisaremos mudar a forma de aquisição de locks para que, em um momento específico, eles tentem adquirir os locks na ordem errada, simulando o cenário acima. Podemos fazer isso programando a situação destes dois robôs para este teste ou alterando a ordem de aquisição de locks no método `mover` e `duelar`

Códigos:

robots_deadlock.py

```
import random
import multiprocessing as mp
import threading
import shared_struct as ss
import gridFunctions
from flagsFunctions import getFlagGameOver, setFlagGameOver,
setFlagVencedor
import time
from teclado import ler_tecla
from multiprocessing import shared_memory
import os

class Robot:
    def __init__(self, ID, F, E, V, posicao_x, posicao_y, status, grid,
flags, robots_shared, robots_mutex, grid_mutex, flags_mutex):
        self.ID = ID
        self.forca = F
        self.energia = E
        self.velocidade = V
        self.posicao_x = posicao_x
        self.posicao_y = posicao_y
        self.status = status
        self.log = []
        self.grid = grid
        self.flags = flags
        self.robots = robots_shared
        self.robots_mutex = robots_mutex
        self.grid_mutex = grid_mutex
        self.flags_mutex = flags_mutex


        with self.robots_mutex:
            self.robots[self.ID].ID = self.ID
            self.robots[self.ID].forca = self.forca
            self.robots[self.ID].energia = self.energia
            self.robots[self.ID].velocidade = self.velocidade
            self.robots[self.ID].posicao_x = self.posicao_x
            self.robots[self.ID].posicao_y = self.posicao_y
            self.robots[self.ID].status = self.status

        with self.grid_mutex:
            self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))

    def get_index(self, x, y):
        return y * ss.WIDTH + x
```

```
    def get_grid(self, x, y):
        byte_value = self.grid[self.get_index(x, y)]
        return chr(byte_value)

    def set_grid(self, x, y, value):
        index = self.get_index(x, y)
        if isinstance(value, str):
            self.grid[index] = ord(value[0])
        else:
            self.grid[index] = value

    def _check_game_over(self):
        with self.flags_mutex:
            if getFlagGameOver(self.flags) == 1:
                return

            vivos = 0
            vencedor_id = -1
            with self.robots_mutex:
                for i in range(ss.QTD_ROBOS):
                    if self.robots[i].status == b'V':
                        vivos += 1
                        vencedor_id = self.robots[i].ID

            if vivos <= 1:
                setFlagVencedor(self.flags, vencedor_id)
                setFlagGameOver(self.flags, 1, self.grid)
                if vencedor_id != -1:
                    print(f"\nO Robô {vencedor_id} é o vencedor!")
                else:
                    print("\nTodos os robôs foram destruídos. Empate!")


    def mover(self):

        bateria_mais_proxima = None
        menor_distancia = float('inf')
        for y in range(ss.HEIGHT):
            for x in range(ss.WIDTH):
                if self.get_grid(x, y) == "B":
                    distancia = abs(self.posicao_x - x) +
abs(self.posicao_y - y)
                    if distancia < menor_distancia:
                        menor_distancia = distancia
                        bateria_mais_proxima = (x, y)

        dx, dy = 0, 0
        if self.energia < 50 and bateria_mais_proxima:
```

```
            self.log.append(f"Robo {self.ID} com energia baixa
({self.energia}). Buscando bateria.")
            bateria_x, bateria_y = bateria_mais_proxima
            if self.posicao_x < bateria_x: dx = 1
            elif self.posicao_x > bateria_x: dx = -1
            elif self.posicao_y < bateria_y: dy = 1
            elif self.posicao_y > bateria_y: dy = -1
        else:
            dx, dy = random.choice([(0, 1), (1, 0), (0, -1), (-1, 0)])

        self.mover_para(dx, dy)

    def mover_para(self, dx, dy):
        if self.status != b'V':
            return

        nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x + dx))
        nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y +
dy))

        #Segurando o grid durante toda movimentação - forçando o erro
para deadlock
        with self.grid_mutex:
            destino = self.get_grid(nova_posicao_x, nova_posicao_y)
            if destino == "-":
                self.set_grid(self.posicao_x, self.posicao_y, "-")
                self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y
                self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID))
                self.energia -= 1
            elif destino == "B":
                self.set_grid(self.posicao_x, self.posicao_y, "-")
                self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y
                self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID))
                self.energia = min(100, self.energia + 20)
            elif destino == "#":
                pass
            elif destino.isdigit() and destino != str(self.ID):
                self.duelar(int(destino), nova_posicao_x,
nova_posicao_y)

        if self.ID == 0: # Limpa a tela e mostra o grid para o jogador
            os.system("cls" if os.name == "nt" else "clear")
            print(f"Jogador {self.ID} | Energia: {self.energia} |
Força: {self.forca}")
```

```
            gridFunctions.printGrid(self.grid)


    def duelar(self, outro_robo_id, x_duelo, y_duelo):
        with self.robots_mutex:
            outro_robo_forca = self.robots[outro_robo_id].forca
            outro_robo_status = self.robots[outro_robo_id].status

            if self.status != b"V" or outro_robo_status != b"V":
                return

            print(f"\nDUELO: Robô {self.ID} (F:{self.forca}) vs Robô
{outro_robo_id} (F:{outro_robo_forca})")

            if self.forca > outro_robo_forca:
                print(f"Robô {self.ID} venceu!")
                self.log.append(f"Venceu duelo contra
{outro_robo_id}.")
                self.robots[outro_robo_id].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.set_grid(x_duelo, y_duelo, str(self.ID))
                self.posicao_x, self.posicao_y = x_duelo, y_duelo

            elif self.forca < outro_robo_forca:
                print(f"Robô {outro_robo_id} venceu!")
                self.log.append(f"Perdeu duelo para {outro_robo_id}.")
                self.status = b"M" # Atualiza o status local para sair
dos loops
                self.robots[self.ID].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")

            else: # Empate
                print("Empate! Ambos foram destruídos.")
                self.log.append(f"Empatou duelo com {outro_robo_id}.")
                self.status = b"M" # Atualiza o status local
                self.robots[self.ID].status = b"M"
                self.robots[outro_robo_id].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.set_grid(x_duelo, y_duelo, "-")

        time.sleep(0.5)
        self._check_game_over()

    def sense_act(self):
        while True:
```

```
            # CORREÇÃO: Verifica o status na memória compartilhada a
cada ciclo
            with self.robots_mutex:
                if self.robots[self.ID].status != b'V':
                    break

            if getFlagGameOver(self.flags) == 1:
                break

            self.mover()
            time.sleep(self.velocidade * 0.2)

    def housekeeping(self):
        while True:
            # CORREÇÃO: Verifica o status na memória compartilhada a
cada ciclo
            with self.robots_mutex:
                if self.robots[self.ID].status != b'V':
                    break

            if getFlagGameOver(self.flags) == 1:
                break

            time.sleep(1)
            self.energia -= 1
            if self.energia <= 0:
                self.status = b"M" # Atualiza status local
                with self.robots_mutex:
                    self.robots[self.ID].status = b"M" # Atualiza
status compartilhado
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                self.log.append(f"Robo {self.ID} ficou sem energia e
foi desligado.")
                self._check_game_over()
                break


    def iniciar(self):
        t_sense_act = threading.Thread(target=self.sense_act)
        t_housekeeping = threading.Thread(target=self.housekeeping)
        t_sense_act.start()
        t_housekeeping.start()
        t_sense_act.join()
        t_housekeeping.join()

        # CORREÇÃO: Escrita correta do log
```

```
        with open(f"robo_{self.ID}_log.txt", "w", encoding="utf-8") as
f:
            f.write(f"--- LOG Robô {self.ID} ---\n")
            for linha in self.log:
                f.write(linha + "\n")
            f.write("--- FIM LOG ---\n")


def processo_robo(ID, flags, robots_shared, robots_mutex, grid_mutex,
flags_mutex, shm_name):
    shm = shared_memory.SharedMemory(name=shm_name)
    grid = shm.buf

    F = random.randint(1, 10)
    E = random.randint(50, 100)
    V = random.randint(1, 5)

    with grid_mutex:
        while True:
            posicao_x = random.randint(0, ss.WIDTH - 1)
            posicao_y = random.randint(0, ss.HEIGHT - 1)
            if chr(grid[posicao_y * ss.WIDTH + posicao_x]) == '-':
                break

    robo = Robot(ID, F, E, V, posicao_x, posicao_y, b"V", grid, flags,
robots_shared, robots_mutex, grid_mutex, flags_mutex)
    robo.iniciar()
    shm.close()

def processo_jogador(ID, flags, robots_shared, robots_mutex,
grid_mutex, flags_mutex, shm_name):
    shm = shared_memory.SharedMemory(name=shm_name)
    grid = shm.buf

    F = 100
    E = 100
    V = 1

    with grid_mutex:
        while True:
            posicao_x = random.randint(0, ss.WIDTH - 1)
            posicao_y = random.randint(0, ss.HEIGHT - 1)
            if chr(grid[posicao_y * ss.WIDTH + posicao_x]) == '-':
                break

    robo = Robot(ID, F, E, V, posicao_x, posicao_y, b"V", grid, flags,
robots_shared, robots_mutex, grid_mutex, flags_mutex)
```

```
    controles = {'w': (0, -1), 's': (0, 1), 'a': (-1, 0), 'd': (1, 0)}

    t_housekeeping = threading.Thread(target=robo.housekeeping)
    t_housekeeping.start()

    os.system("cls" if os.name == "nt" else "clear")
    print("Controle ativado: use W, A, S, D para se movimentar.
Pressione Q para sair.")
    print(f"Jogador {robo.ID} | Energia: {robo.energia} | Força:
{robo.forca}")
    gridFunctions.printGrid(robo.grid)

    while True:
        # CORREÇÃO: Verifica o status na memória compartilhada a cada
ciclo
        with robots_mutex:
            if robots_shared[robo.ID].status != b'V':
                print("\nVocê foi derrotado!")
                break

        if getFlagGameOver(flags) == 1:
            break

        tecla = ler_tecla().lower()
        if tecla in controles:
            dx, dy = controles[tecla]
            robo.mover_para(dx, dy)
        elif tecla == 'q':
            print("Você saiu do jogo.")
            robo.status = b'M' # Marca como morto ao sair
            with robots_mutex:
                robots_shared[robo.ID].status = b'M'
            break

    robo._check_game_over()
    t_housekeeping.join()
    shm.close()
```

---

robots.py

```
import random
import threading
import shared_struct as ss
import gridFunctions
```

```
from flagsFunctions import getFlagGameOver, setFlagGameOver,
setFlagVencedor
import time
from teclado import ler_tecla
from multiprocessing import shared_memory
import os

class Robot:
    def __init__(self, ID, F, E, V, posicao_x, posicao_y, status, grid,
flags, robots_shared, robots_mutex, grid_mutex, flags_mutex):
        self.ID = ID
        self.forca = F
        self.energia = E
        self.velocidade = V
        self.posicao_x = posicao_x
        self.posicao_y = posicao_y
        self.status = status
        self.log = []
        self.grid = grid
        self.flags = flags
        self.robots = robots_shared
        self.robots_mutex = robots_mutex
        self.grid_mutex = grid_mutex
        self.flags_mutex = flags_mutex


        with self.robots_mutex:
            self.robots[self.ID].ID = self.ID
            self.robots[self.ID].forca = self.forca
            self.robots[self.ID].energia = self.energia
            self.robots[self.ID].velocidade = self.velocidade
            self.robots[self.ID].posicao_x = self.posicao_x
            self.robots[self.ID].posicao_y = self.posicao_y
            self.robots[self.ID].status = self.status

        with self.grid_mutex:
            self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))

    def get_index(self, x, y):
        return y * ss.WIDTH + x

    def get_grid(self, x, y):
        byte_value = self.grid[self.get_index(x, y)]
        return chr(byte_value)

    def set_grid(self, x, y, value):
        index = self.get_index(x, y)
        if isinstance(value, str):
```

```
            self.grid[index] = ord(value[0])
        else:
            self.grid[index] = value

    def _check_game_over(self):
        with self.flags_mutex:
            if getFlagGameOver(self.flags) == 1:
                return

            vivos = 0
            vencedor_id = -1
            with self.robots_mutex:
                for i in range(ss.QTD_ROBOS):
                    if self.robots[i].status == b'V':
                        vivos += 1
                        vencedor_id = self.robots[i].ID

            if vivos <= 1:
                setFlagVencedor(self.flags, vencedor_id)
                # O setFlagGameOver agora só aceita 2 argumentos na sua
versão
                setFlagGameOver(self.flags, 1)

    def mover(self):
        # Lógica de movimento sem a IA de caça
        bateria_mais_proxima = None
        menor_distancia = float('inf')
        for y in range(ss.HEIGHT):
            for x in range(ss.WIDTH):
                if self.get_grid(x, y) == "B":
                    distancia = abs(self.posicao_x - x) +
abs(self.posicao_y - y)
                    if distancia < menor_distancia:
                        menor_distancia = distancia
                        bateria_mais_proxima = (x, y)

        dx, dy = 0, 0
        # Se a energia for baixa, busca a bateria
        if self.energia < 50 and bateria_mais_proxima:
            self.log.append(f"Robo {self.ID} com energia baixa
({self.energia}). Buscando bateria.")
            bateria_x, bateria_y = bateria_mais_proxima
            if self.posicao_x < bateria_x: dx = 1
            elif self.posicao_x > bateria_x: dx = -1
            elif self.posicao_y < bateria_y: dy = 1
            elif self.posicao_y > bateria_y: dy = -1
        # Caso contrário, move-se aleatoriamente
        else:
```

```
            dx, dy = random.choice([(0, 1), (1, 0), (0, -1), (-1, 0)])

        self.mover_para(dx, dy)

    def mover_para(self, dx, dy):
        # Esta função contém a correção de deadlock e deve ser mantida
        if self.status != b'V' or (dx==0 and dy==0):
            return

        nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x + dx))
        nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y +
dy))

        # Adquire o lock do grid apenas para ler o destino, e o libera
em seguida
        with self.grid_mutex:
            destino = self.get_grid(nova_posicao_x, nova_posicao_y)

        # Age com base no destino, sem segurar o lock do grid
        if destino.isdigit() and destino != str(self.ID):
            self.duelar(int(destino), nova_posicao_x, nova_posicao_y)
        elif destino == "-":
            with self.grid_mutex: # Re-adquire o lock para uma operação
rápida
                if self.get_grid(nova_posicao_x, nova_posicao_y) ==
"-":
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y
                    self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID))
                    self.energia -= 1
        elif destino == "B":
            with self.grid_mutex: # Re-adquire o lock para uma operação
rápida
                if self.get_grid(nova_posicao_x, nova_posicao_y) ==
"B":
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.posicao_x, self.posicao_y = nova_posicao_x,
nova_posicao_y
                    self.set_grid(self.posicao_x, self.posicao_y,
str(self.ID))
                    self.energia = min(100, self.energia + 20)

        if self.ID == 0:
            os.system("cls" if os.name == "nt" else "clear")
            print(f"Jogador {self.ID} | Energia: {self.energia} |
Força: {self.forca}")
```

```
            gridFunctions.printGrid(self.grid)

    def duelar(self, outro_robo_id, x_duelo, y_duelo):
        # Ordem de lock consistente para evitar deadlock: robots_mutex
-> grid_mutex
        with self.robots_mutex:
            outro_robo_forca = self.robots[outro_robo_id].forca
            outro_robo_status = self.robots[outro_robo_id].status

            if self.status != b"V" or outro_robo_status != b"V":
                return

            print(f"\nDUELO: Robô {self.ID} (F:{self.forca}) vs Robô
{outro_robo_id} (F:{outro_robo_forca})")

            if self.forca > outro_robo_forca:
                self.robots[outro_robo_id].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.set_grid(x_duelo, y_duelo, str(self.ID))
                self.posicao_x, self.posicao_y = x_duelo, y_duelo

            elif self.forca < outro_robo_forca:
                self.status = b"M"
                self.robots[self.ID].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
            else:
                self.status = b"M"
                self.robots[self.ID].status = b"M"
                self.robots[outro_robo_id].status = b"M"
                with self.grid_mutex:
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.set_grid(x_duelo, y_duelo, "-")

        time.sleep(0.5)
        self._check_game_over()

    def sense_act(self):
        while True:
            with self.robots_mutex:
                if self.robots[self.ID].status != b'V': break
            if getFlagGameOver(self.flags) == 1: break
            self.mover()
            time.sleep(self.velocidade * 0.2)

    def housekeeping(self):
        while True:
```

```
            with self.robots_mutex:
                if self.robots[self.ID].status != b'V': break
            if getFlagGameOver(self.flags) == 1: break

            time.sleep(1)
            self.energia -= 1
            if self.energia <= 0:
                self.status = b"M"
                # Ordem de lock consistente para evitar deadlock
                with self.robots_mutex:
                    self.robots[self.ID].status = b"M"
                    with self.grid_mutex:
                        self.set_grid(self.posicao_x, self.posicao_y,
"-")
                self._check_game_over()
                break

    def iniciar(self):
        t_sense_act = threading.Thread(target=self.sense_act)
        t_housekeeping = threading.Thread(target=self.housekeeping)
        t_sense_act.start()
        t_housekeeping.start()
        t_sense_act.join()
        t_housekeeping.join()

def processo_robo(ID, flags, robots_shared, robots_mutex, grid_mutex,
flags_mutex, shm_name):
    shm = shared_memory.SharedMemory(name=shm_name)
    grid = shm.buf

    F = random.randint(1, 10)
    E = random.randint(50, 100)
    V = random.randint(1, 5)

    with grid_mutex:
        while True:
            posicao_x = random.randint(0, ss.WIDTH - 1)
            posicao_y = random.randint(0, ss.HEIGHT - 1)
            if chr(grid[posicao_y * ss.WIDTH + posicao_x]) == '-':
                break

    robo = Robot(ID, F, E, V, posicao_x, posicao_y, b"V", grid, flags,
robots_shared, robots_mutex, grid_mutex, flags_mutex)
    robo.iniciar()
    shm.close()

def processo_jogador(ID, flags, robots_shared, robots_mutex,
grid_mutex, flags_mutex, shm_name):
```

```
    shm = shared_memory.SharedMemory(name=shm_name)
    grid = shm.buf

    F = 100
    E = 100
    V = 1

    with grid_mutex:
        while True:
            posicao_x = random.randint(0, ss.WIDTH - 1)
            posicao_y = random.randint(0, ss.HEIGHT - 1)
            if chr(grid[posicao_y * ss.WIDTH + posicao_x]) == '-':
                break

    robo = Robot(ID, F, E, V, posicao_x, posicao_y, b"V", grid, flags,
robots_shared, robots_mutex, grid_mutex, flags_mutex)

    controles = {'w': (0, -1), 's': (0, 1), 'a': (-1, 0), 'd': (1, 0)}
    t_housekeeping = threading.Thread(target=robo.housekeeping)
    t_housekeeping.start()

    os.system("cls" if os.name == "nt" else "clear")
    print("Controle ativado: use W, A, S, D para se movimentar.
Pressione Q para sair.")
    gridFunctions.printGrid(robo.grid)

    while True:
        with robots_mutex:
            if robots_shared[robo.ID].status != b'V':
                print("\nVocê foi derrotado!")
                break
        if getFlagGameOver(flags) == 1:
            break
        tecla = ler_tecla().lower()
        if tecla in controles:
            dx, dy = controles[tecla]
            robo.mover_para(dx, dy)
        elif tecla == 'q':
            robo.status = b'M'
            with robots_mutex:
                robots_shared[robo.ID].status = b'M'
            break

    robo._check_game_over()
    t_housekeeping.join()
    shm.close()
```

Caso de deadlock

Forçamos o erro segurando o lock de grid para fazer todos os movimentos. Assim, quando fosse pegar o lock do duelo, teríamos preso, travando o jogo.
Notemos

```python
def mover_para(self, dx, dy):
    if self.status != b'V':
        return

    nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x + dx))
    nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y + dy))

    with self.grid_mutex:
        destino = self.get_grid(nova_posicao_x, nova_posicao_y)
        if destino == "-":
            self.set_grid(self.posicao_x, self.posicao_y, "-")
            self.posicao_x, self.posicao_y = nova_posicao_x, nova_posicao_y
            self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))
            self.energia -= 1
        elif destino == "B":
            self.set_grid(self.posicao_x, self.posicao_y, "-")
            self.posicao_x, self.posicao_y = nova_posicao_x, nova_posicao_y
            self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))
            self.energia = min(100, self.energia + 20)
        elif destino == "#":
            pass
        elif destino.isdigit() and destino != str(self.ID):
            self.duelar(int(destino), nova_posicao_x, nova_posicao_y)
```

```
Jogador 0 | Energia: 81 | Força: 100
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - # - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - # - - - - - - - - B - - - - - -
- - - - - - - - - - - - B - - - - - - - - - - # - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - B - - - - - - - - - - - - - - - - - - - - - -
- - - - - B - B - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- # - B - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# - - - - - - B - - - - - - - - - - - - - - - - - # - - - - - - - - - - - - - -
- - - 3 - - - - - - - - - - - - - - - - - - - - - - - - # - - - - 1 - - - - - -
- - - - - - - B - - - - - B - - - - - - - - - - - - - - - - - - - 0 - - - - - -
- - - - - # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - # - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - # - - - - - - - 2 - - -
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

DUELO: Robô 0 (F:100) vs Robô 1 (F:5)
Robô 0 venceu!
```

A chave da correção foi fazer esse mutex intermitente nos movimentos

```python
    def mover_para(self, dx, dy):
        # Esta função contém a correção de deadlock e deve ser mantida
        if self.status != b'V' or (dx==0 and dy==0):
            return

        nova_posicao_x = max(0, min(ss.WIDTH - 1, self.posicao_x + dx))
        nova_posicao_y = max(0, min(ss.HEIGHT - 1, self.posicao_y + dy))

        # Adquire o lock do grid apenas para ler o destino, e o libera em seguida
        with self.grid_mutex:
            destino = self.get_grid(nova_posicao_x, nova_posicao_y)

        # Age com base no destino, sem segurar o lock do grid
        if destino.isdigit() and destino != str(self.ID):
            self.duelar(int(destino), nova_posicao_x, nova_posicao_y)
        elif destino == "-":
            with self.grid_mutex: # Re-adquire o lock para uma operação rápida
                if self.get_grid(nova_posicao_x, nova_posicao_y) == "-":
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.posicao_x, self.posicao_y = nova_posicao_x, nova_posicao_y
                    self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))
                    self.energia -= 1
        elif destino == "B":
            with self.grid_mutex: # Re-adquire o lock para uma operação rápida
                if self.get_grid(nova_posicao_x, nova_posicao_y) == "B":
                    self.set_grid(self.posicao_x, self.posicao_y, "-")
                    self.posicao_x, self.posicao_y = nova_posicao_x, nova_posicao_y
                    self.set_grid(self.posicao_x, self.posicao_y, str(self.ID))
                    self.energia = min(100, self.energia + 20)

        if self.ID == 0:
            os.system("cls" if os.name == "nt" else "clear")
            print(f"Jogador {self.ID} | Energia: {self.energia} | Força: {self.forca}")
            gridFunctions.printGrid(self.grid)
```

Análise do Deadlock em robots_deadlock.py

O cenário de deadlock é criado pelas seguintes interações e ordem de aquisição de locks:

1. **Robô X (em mover_para):**
    - Adquire o **grid_mutex** (with self.grid_mutex:). Este lock é mantido durante toda a movimentação.
    - Durante a movimentação, ele detecta que o destino.isdigit(), significando que ele encontrou outro robô.
    - Chama self.duelar(int(destino), nova_posicao_x, nova_posicao_y).
2. **Dentro de duelar (ainda pelo Robô X):**
    - Tenta adquirir o **robots_mutex** (with self.robots_mutex:).
    - Se conseguir, ele acessa os dados do outro robô (outro_robo_id).
    - No caso de vitória do Robô X (self.forca > outro_robo_forca):
        - Ele atualiza o status do robô derrotado (self.robots[outro_robo_id].status = b"M").
        - **ELE TENTA ADQUIRIR NOVAMENTE O grid_mutex** (with self.grid_mutex:).

Onde está o Deadlock?

O problema acontece quando dois robôs (digamos **Robô A** e **Robô B**) tentam duelar, ou um robô tenta duelar enquanto outro está se movendo e já segura um lock, na seguinte sequência:

1. **Robô A:** Executa mover_para() e **adquire grid_mutex**.
    - Detecta o Robô B e chama duelar().
    - Dentro de duelar(), Robô A **tenta adquirir robots_mutex**.
2. **Robô B:** (Pode estar em mover_para() ou duelar())
    - Se Robô B também chamou mover_para() e está prestes a duelar, ele **precisa do grid_mutex** para avançar, mas o Robô A o está segurando.
    - Se Robô B já está em duelar() e também tentou adquirir robots_mutex e grid_mutex, a ordem invertida pode levar ao bloqueio.

Entendendo a Correção do Deadlock

A principal causa do deadlock no código anterior era a **violação da ordem de aquisição de locks**, especificamente entre o grid_mutex e o robots_mutex. Em cenários de concorrência, quando diferentes threads ou processos tentam adquirir múltiplos recursos em ordens variadas, cria-se a condição de "espera circular" que leva ao deadlock.

No código corrigido, a solução reside em garantir uma **ordem consistente de aquisição de recursos** para as operações que envolvem tanto o grid quanto os dados dos robôs.

Vamos analisar as mudanças mais importantes:

1. Função mover_para

No código anterior, a função mover_para mantinha o grid_mutex travado durante toda a sua execução, incluindo a chamada para duelar. Isso impedia que outro robô, ao tentar duelar, adquirisse o grid_mutex se ele já estivesse travado pelo primeiro robô que estava movendo-se e duelando.

No código corrigido:

- A função mover_para agora **adquire o grid_mutex apenas para ler o destino** (destino = self.get_grid(nova_posicao_x, nova_posicao_y)).
- **Ele libera o grid_mutex imediatamente** após essa leitura.
- A lógica de ação (mover para espaço vazio, pegar bateria ou duelar) é feita **FORA** do grid_mutex inicial.
- Se o destino for um robô (destino.isdigit()), a função duelar é chamada sem o grid_mutex já estar em posse de mover_para.
- Para as operações de movimentação para um espaço vazio ou bateria, o grid_mutex é re-adquirido em um bloco with **curto e específico** para a escrita no grid. Isso minimiza o tempo que o grid_mutex fica segurado.

Essa mudança é crucial porque **remove a dependência prolongada** do grid_mutex dentro de mover_para antes de potencialmente chamar duelar.

2. Função duelar

No código original, a função duelar tentava adquirir o robots_mutex e, dentro dele, em certas condições (vitória, empate), tentava re-adquirir o grid_mutex. Essa ordem robots_mutex -> grid_mutex podia colidir com a ordem grid_mutex -> robots_mutex de outros caminhos de execução.

No código corrigido:

- A ordem de aquisição de locks na função duelar foi estabelecida como: **robots_mutex** e, **dentro dele**, o **grid_mutex**.
- with self.robots_mutex:: Primeiro, garante-se acesso exclusivo aos dados dos robôs.
- with self.grid_mutex:: Depois, dentro do escopo do robots_mutex, adquire-se o grid_mutex para atualizar o estado do grid (movimentar o vencedor, limpar posições de robôs mortos/empatados).

Essa ordem (robots_mutex antes de grid_mutex dentro de duelar) combinada com a liberação rápida do grid_mutex em mover_para garante que as condições para o deadlock (retenção e espera, e espera circular) sejam quebradas.

Logs de desafios pessoais

Foi desafiador fazer parte deste trabalho, algo que nos tirou da zona de conforto. Tive minha primeira experiência com o GitHub, então foi um passo a mais terde aprender a utilizar a ferramenta.
O projeto é todo integrado, então, mesmo cada um fazendo uma parte, precisamos ter de ter consciência geral de tudo. Por exemplo, para fazer o estudo de deadlock e mutex, precisamos rever a forma como estava esboçada nossa estrutura inicial. Os mutex não estavam exatamente compartilhados, mas, sim, com instâncias locais, tivemos de rever isso para funcionar adequadamente.
— Davi

A maior dificuldade que tive foi implementar a lógica de como fazer para o robô identificar e ir atrás das baterias considerando sua posição atual e a posição da bateria. Tive que fazer um estudo de cálculos de distâncias eficientes que desse prioridade a movimentação mais curtas. também tive um pouco de dificuldade no entendimento das memórias compartilhadas entre os processos como o uso do SharedMemory.
— Thiago

Uma das principais dificuldades foi fazer o viewer funcionar como um processo separado, conforme solicitado. Para isso, precisei entender e aplicar o uso de shared_memory e locks para que o viewer acessasse o grid em tempo real, sem interferir nos demais processos. Também foi necessário adaptar os outros arquivos do projeto para garantir que o acesso à memória fosse seguro e sincronizado. Como meu código estava isolado do main.py, houve um certo desafio para integrá-lo corretamente com o restante do sistema.

— Marco

Foi complicado entender o funcionamento e as diferenças entre do shared_memory e do Array e como/onde usar cada um, tomar as decisões relacionadas a isso foi algo difícil. Também foi complicado(mesmo que menos do que esperava) de mudar algo que já estava feito em uma para outra.
— Amin