Facimp Wyden

PROCESSOS DE APLICAÇÕES CONCORRENTES E CONDIÇÃO DE CORRIDA

Presentation by

David Marques

Kauã Cantanhede

CONTEÚDO

Facimp Wyden

INTRODUÇÃO

INTRODUÇÃO

Facimp Wyden

Aplicações Concorrentes

- **Definição**: Sistemas operacionais permitem a execução de múltiplos processos de forma simultânea, onde diferentes processos podem interagir com o sistema ou entre si.

- **Importância**: A concorrência permite melhor utilização dos recursos do sistema.

COMPARTILHAMENTO DE RECURSOS

1

Definição
Processos concorrentes compartilham recursos do sistema, como memória e arquivos.

2

Evitando problemas

- Troca de informações entre processos.
- Sequenciamento correto de tarefas interdependentes

MECANISMOS DE SINCRONIZAÇÃO

1 Objetivo
Garantir que processos concorrentes acessem recursos de forma segura.

2 Exemplos
Semáforos e mutexes.

CONDIÇÃO DE CORRIDA

DEFINIÇÃO

Ocorre quando dois ou mais processos acessam e manipulam dados compartilhados sem controle adequado, levando a resultados imprevisíveis.

EXEMPLO

Sistema de verificação de saldo anual, onde diferentes processos atualizam o saldo ao mesmo tempo, podendo causar inconsistências.

CAUSA

A falta de sincronização entre processos ou threads que acessam o mesmo recurso simultaneamente.

EVITANDO CONDIÇÕES DE CORRIDA

1 **Exclusão Mútua**
Garantir que apenas um processo acesse um recurso compartilhado de cada vez.

2 **Região Crítica**
Parte do código onde ocorre o acesso ao recurso compartilhado.

3 **Mecanismos**
Semáforos e mutexes para controle de acesso.

EXEMPLOS EM SISTEMAS REAIS

1. Aplicações Bancárias
   Acessos simultâneos ao saldo de contas.

2. Sistemas de Reserva de Voos
   Reservas concorrentes do mesmo assento.

3. Sistemas de Compartilhamento de Arquivos
   Acesso simultâneo a arquivos.

SEMÁFOROS E SINCRONIZAÇÃO

1

Down (P)
Bloqueia o processo quando o semáforo atinge zero.

2

Up (V)
Libera o recurso e permite que outro processo entre na região crítica.

Exemplo Prática

Cenário

Facimp Wyden

Gestão de Saques em um Sistema Financeiro Multithread

- Um sistema financeiro que permite a retirada mensal de um saldo anual.
- Várias threads (um para cada mês) tentam modificar o saldo simultaneamente.
- Objetivo: Demonstrar os efeitos de condições de corrida e a importância da sincronização.

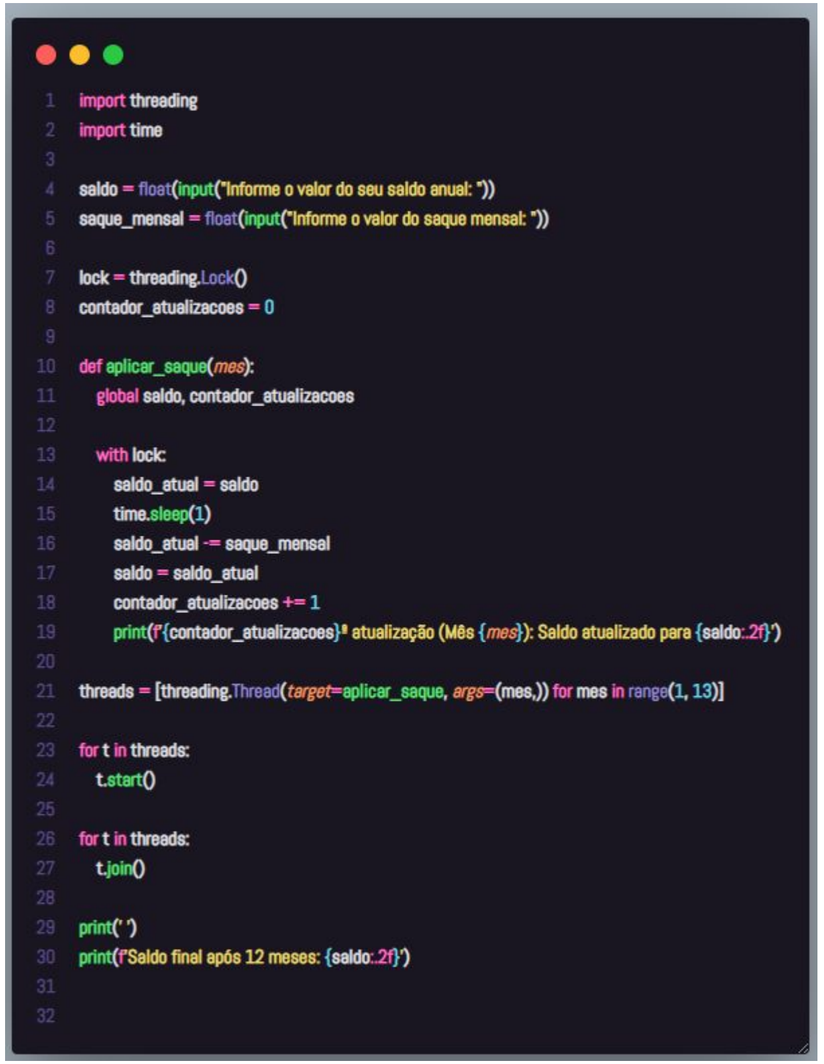
Condição de Corrida sem Sincronização

- Acesso simultâneo ao saldo anual e saque mensal.
- Linha crítica: saldo = saldo_atual.
- Uso de time.sleep() simula atraso nas threads.
- Resultado: alterações inconsistentes e aleatórias no saldo.

```python
import threading
import time

saldo = float(input("Informe o valor do seu Saldo Anual: "))
saque_mensal = float(input("Informe o valor do Saque Mensal: "))

contador_atualizacoes = 0

def aplicar_desconto(mes):
    global saldo, contador_atualizacoes

    saldo_atual = saldo

    time.sleep(1)

    saldo_atual -= saque_mensal

    saldo = saldo_atual

    contador_atualizacoes += 1
    print(f'{contador_atualizacoes}ª atualização (Mês {mes}): Saldo atualizado para {saldo:.2f}')

threads = [threading.Thread(target=aplicar_desconto, args=(mes,)) for mes in range(1, 13)]

for t in threads:
    t.start()

for t in threads:
    t.join()

print(' ')
print(f'Saldo final após 12 meses: {saldo:.2f}')

```

Condição de Corrida com Sincronização

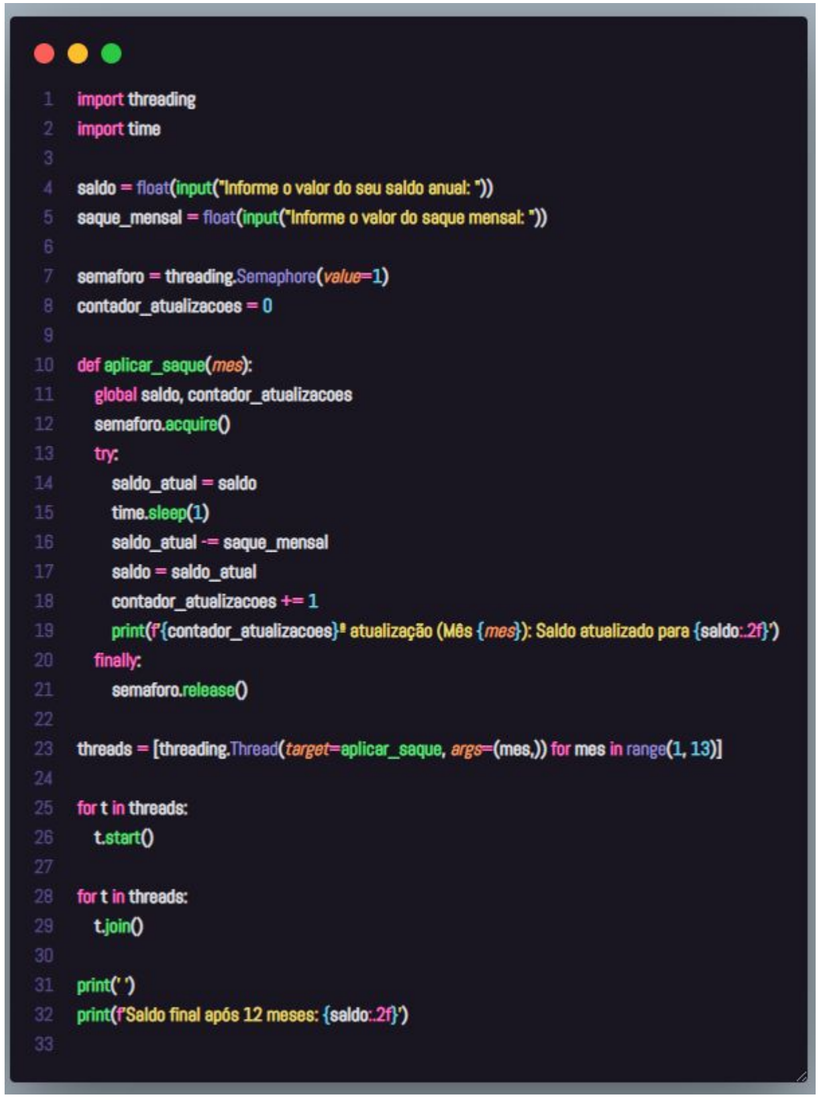
- Introdução de um mutex para controle de acesso.
- Exclusão mútua: apenas uma thread pode acessar a seção crítica por vez.
- Bloqueio das threads restantes até a conclusão da primeira.
- Resultado: alterações ordenadas e saldo atualizado corretamente



```python
import threading
import time

saldo = float(input("Informe o valor do seu saldo anual: "))
saque_mensal = float(input("Informe o valor do saque mensal: "))

lock = threading.Lock()
contador_atualizacoes = 0

def aplicar_saque(mes):
    global saldo, contador_atualizacoes

    with lock:
        saldo_atual = saldo
        time.sleep(1)
        saldo_atual -= saque_mensal
        saldo = saldo_atual
        contador_atualizacoes += 1
        print(f"{contador_atualizacoes}ª atualização (Mês {mes}): Saldo atualizado para {saldo:.2f}")

threads = [threading.Thread(target=aplicar_saque, args=(mes,)) for mes in range(1, 13)]

for t in threads:
    t.start()

for t in threads:
    t.join()

print(" ")
print(f"Saldo final após 12 meses: {saldo:.2f}")
```

Condição de Corrida Gerida pelo Próprio Sistema

- Uso de semáforo inicializado com valor 1.
- Permite que apenas uma thread altere o saldo por vez.
- Semáforo bloqueia outras threads enquanto uma está em execução.
- Scheduler e BCP gerenciam a execução e o estado das threads.
- Diferença: semáforo pode permitir múltiples threads, enquanto o mutex permite apenas uma.



```python
import threading
import time

saldo = float(input("Informe o valor do seu saldo anual: "))
saque_mensal = float(input("Informe o valor do saque mensal: "))

semaforo = threading.Semaphore(value=1)
contador_atualizacoes = 0

def aplicar_saque(mes):
    global saldo, contador_atualizacoes
    semaforo.acquire()
    try:
        saldo_atual = saldo
        time.sleep(1)
        saldo_atual -= saque_mensal
        saldo = saldo_atual
        contador_atualizacoes += 1
        print(f"{contador_atualizacoes}ª atualização (Mês {mes}): Saldo atualizado para {saldo:.2f}")
    finally:
        semaforo.release()

threads = [threading.Thread(target=aplicar_saque, args=(mes,)) for mes in range(1, 13)]

for t in threads:
    t.start()

for t in threads:
    t.join()

print(" ")
print(f"Saldo final após 12 meses: {saldo:.2f}")

```

CONCLUSÃO

A concorrência é um elemento essencial nos sistemas operacionais modernos, permitindo que múltiplos processos ou threads sejam executados simultaneamente, maximizando o uso dos recursos do sistema e aumentando o desempenho. No entanto, essa simultaneidade traz consigo desafios críticos, especialmente quando processos compartilham recursos como memória e arquivos.