https://drive.google.com/file/d/1r9QR4HRfS2cfGRMGo4MklTEDtyLPFFUA/view?pli=1

Pedro Henrique Chaves Junqueira
Matheus de Moura Rosa
Mateus Silva de Sousa
Rafael Estanislau

1- parte 1 AF2

GITHUB

1- Parte 2

Arquitetura da Solução

A solução envolve a criação de uma estrutura onde múltiplas threads (produtores e consumidores) interagem com um objeto compartilhado (Mailbox) para armazenar e recuperar mensagens. A comunicação entre as threads e a sincronização do acesso ao Mailbox são essenciais para evitar condições de corrida e garantir a consistência dos dados.

Componentes Principais

1. **Mailbox**: Armazena temporariamente a mensagem e gerencia o acesso concorrente usando métodos sincronizados.
2. **Producer**: Thread que produz mensagens e as coloca no Mailbox.
3. **Consumer**: Thread que consome mensagens do Mailbox.

Fluxo de Execução e Coordenação

1. **Produtores**:
    - Tentam armazenar uma mensagem no Mailbox.
    - Se o Mailbox já contém uma mensagem, o produtor deve esperar até que a mensagem seja consumida.
    - Após armazenar a mensagem, notificam todos que uma nova mensagem está disponível.
2. **Consumidores**:
    - Tentam recuperar uma mensagem do Mailbox.
    - Se o Mailbox está vazio, o consumidor deve esperar até que uma nova mensagem seja produzida.
    - Após consumir a mensagem, notificam todos que o Mailbox está vazio e pode aceitar uma nova mensagem.

Sincronização (Exclusão Mútua)

- O uso de blocos sincronizados (`synchronized`) garante que apenas uma thread possa executar um bloco de código crítico (armazenar ou recuperar a mensagem) no Mailbox por vez.
- Os métodos `wait()` e `notifyAll()` são utilizados para coordenar as threads:
    - `wait()`: Faz a thread liberar o bloqueio e esperar até que seja notificada.
    - `notifyAll()`: Acorda todas as threads que estão esperando no mesmo objeto, permitindo que elas tentem novamente adquirir o bloqueio.

2 - A imagem representa 2 processos A e B em exclusão mutua que garante que somente um processo ou thread tenha acesso a um recurso compartilhado em determinado momento, o processo A entra na região crítica e em certo momento o processo B tenta

entrar na região crítica porém é barrado pelo sistema, ao finalizar o processo A é liberado o acesso ao processo B que ao ser finalizado também sai da região crítica.

- processo A
    - Entra no T1.
    - Sai noT3
- Processo B
    - Entra no T2
    - Saida no T4

Região critica: Uma região crítica é uma parte do código onde dados compartilhados são acessados e onde é necessário garantir que apenas um processo ou thread por vez possa acessá-los para evitar problemas de consistência de dados.

Processo: um processo é uma instância em execução de um programa em um sistema operacional

exclusão mútua:é um conceito de programação concorrente que garante que somente um processo ou thread tenha acesso a um recurso compartilhado em determinado momento.

3 - **Prós**
Simplicidade de Implementação: É relativamente simples de implementar, exigindo apenas loops de espera ocupada e variáveis de controle.
Exclusão Mútua: Garante que apenas um processo entre na região crítica por vez, desde que tum seja corretamente inicializado.

Contra
Uso Intensivo de CPU: Os processos ficam em um loop ativo esperando que a condição `tum != X` seja satisfeita. Isso consome ciclos de CPU desnecessariamente, reduzindo a eficiência do sistema.
Falta de Progresso Garantido: Não há garantias de que um processo esperando poderá eventualmente entrar na região crítica se o outro processo não sair dela.
Risco de Starvation: Dependendo da ordem de execução dos processos, um processo pode monopolizar a região crítica, impedindo o progresso do outro.
Falta de Alternância: Não há garantia de alternância justa entre os processos na entrada da região crítica.

Soluções Alternativas

- **Semáforos:** Os semáforos são uma ferramenta poderosa para gerenciar a sincronização entre processos. Um semáforo pode ser utilizado para controlar o acesso à região crítica sem recorrer à espera ocupada.
- **Monitores:** Monitores são uma abstração de alto nível que encapsula métodos, dados compartilhados e a sincronização dos acessos a esses dados. Em linguagens que suportam monitores, como Java, podemos usar synchronized para controlar o acesso à região crítica.
- **Algoritmo de Peterson:** O algoritmo de Peterson é uma solução clássica para a exclusão mútua de dois processos que evita a espera ocupada.

- **Implementação com sleep e wakeup:** A ideia básica é usar uma variável de condição e um mutex para garantir que apenas um processo entre na região crítica de cada vez, suspendendo os processos que não podem entrar até que a região crítica esteja disponível

4) Enquanto um processo é uma instância em execução de um programa, uma thread é uma unidade menor de execução dentro de um processo que compartilha recursos com outras threads no mesmo processo. As threads permitem a execução paralela ou concorrente de tarefas dentro de um processo, enquanto os processos fornecem isolamento entre diferentes instâncias de programas em execução.

5 - Safety () (Corretude)

(safety) refere-se à garantia de que um sistema de software opera de acordo com suas especificações e requisitos sem causar danos, falhas catastróficas ou violações de, mesmo em condições adversas. Em outras palavras, um sistema seguro é aquele que não permite que eventos não autorizados ou imprevistos causem danos ou comprometam sua integridade. Caso algo de ruim aconteça, seja tratado.

Exemplos de propriedades.

- **Exclusão mútua**: Em um sistema de múltiplos threads, nenhuma duas threads podem acessar simultaneamente uma seção crítica.
- **Invariantes de integridade**: Em um banco de dados, certos invariantes, como a consistência referencial, não devem ser violados.
- **Não ultrapassar limites**: Em uma rede de tráfego, não deve haver colisão entre veículos.

Essencialmente, uma propriedade que pode ser vista como uma condição que deve sempre ser verdadeira.

Exemplo:

- **Exclusão Mútua**: Em um sistema de múltiplos threads. Imagine um banco de dados onde múltiplas threads tentam atualizar o mesmo registro ao mesmo tempo. A propriedade de exclusão mútua garante que apenas uma thread pode acessar a seção crítica (o registro) de cada vez, evitando inconsistências e garantindo que os dados não sejam corrompidos

```python
import threading

# Variável compartilhada
counter = 0
# Lock para garantir a exclusão mútua
lock = threading.Lock()

def increment():
    global counter
    for _ in range(100000):
        # Início da seção crítica
        lock.acquire()
        try:
            counter += 1
        finally:
            # Fim da seção crítica
            lock.release()

threads = []
for i in range(10):
    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print(f"Counter final: {counter}")
```

Explicação:

- `database_record` é um registro simulado que múltiplas threads tentam atualizar.
- `db_lock` garante que apenas uma thread possa acessar e modificar `database_record` por vez.
- O uso de `with db_lock` em volta da seção crítica garante a exclusão mútua, evitando que o valor do registro seja corrompido por acessos concorrentes

```
PS C:\Users\Aluno\Desktop\lap  figura>
Counter final: 1000000
PS C:\Users\Aluno\Desktop\lap  figura>
```

Liveness

Liveness refere-se à garantia de que "algo bom eventualmente acontecerá". Propriedades refere-se à propriedade de um sistema de software continuar a fazer progresso, mesmo

diante de condições adversas, como bloqueios ou deadlocks.ntinuar a fazer progresso, mesmo diante de condições adversas, como bloqueios ou deadlocks.

Exemplos de propriedades.

- **Deadlock-freedom (ausência de deadlocks)**: O sistema deve garantir que não haverá uma situação em que nenhum progresso pode ser feito.
- **Fairness (justiça)**: Todos os processos ou threads devem ter a oportunidade de progredir ou usar os recursos, evitando que algum processo fique esperando indefinidamente.
- **Eventual response (resposta eventual)**: Se uma solicitação é feita, eventualmente haverá uma resposta.

Uma propriedade que pode ser vista como uma condição que deve eventualmente ser verdadeira.

Exemplo:

- **Ausência de Deadlocks**: Considere um sistema onde múltiplas threads precisam acessar vários recursos para completar suas tarefas. A ausência de deadlocks é uma propriedade de Liveness . Por exemplo, se a Thread A possui o Recurso 1 e espera pelo Recurso 2, enquanto a Thread B possui o Recurso 2 e espera pelo Recurso 1, ambas as threads ficam bloqueadas indefinidamente. A garantia de ausência de deadlocks assegura que o sistema terá mecanismos (como um algoritmo de detecção e resolução de deadlocks) para evitar que isso aconteça, permitindo que todas as threads eventualmente completem suas tarefas.

```python
import threading

# Definindo dois recursos como locks
resource1 = threading.Lock()
resource2 = threading.Lock()

def thread1():
    with resource1:
        print("Thread 1 acquired Resource 1")
        with resource2:
            print("Thread 1 acquired Resource 2")

def thread2():
    with resource1:
        print("Thread 2 acquired Resource 1")
        with resource2:
            print("Thread 2 acquired Resource 2")

t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)

t1.start()
t2.start()

t1.join()
t2.join()

print("Threads have completed execution")

```

Explicação:

- `resource1` e `resource2` são locks representando recursos.
- `thread1` e `thread2` tentam adquirir os recursos na mesma ordem, prevenindo deadlocks.
- Ao garantir que todos os threads adquiram os recursos na mesma ordem (`resource1` antes de `resource2`), evitamos uma situação onde `thread1` possui `resource1` e espera por `resource2`, enquanto `thread2` possui `resource2` e espera por `resource1`, o que levaria a um deadlock.

```
PS C:\Users\Aluno\Desktop\lap  figura> &
Thread 1 acquired Resource 1
Thread 1 acquired Resource 2
Thread 2 acquired Resource 1
Thread 2 acquired Resource 2
Threads have completed execution
PS C:\Users\Aluno\Desktop\lap  figura>
```

6 - a) Deve ser tratada com técnicas de concorrência.

- pois o processo de download do arquivo e a atualização da tela com a porcentagem de dados baixados ocorrem simultaneamente.
- Utilizar técnicas de concorrência, como threads ou processos assíncronos, permite que o processo de download e a atualização da interface do usuário (UI) aconteçam de forma paralela, evitando que o download bloqueie a interface e permitindo que o usuário acompanhe o progresso do download em tempo real.
- Sem a concorrência, o download do arquivo poderia bloquear a interface do usuário, tornando-a não responsiva até que o download seja concluído, o que resultaria em uma experiência de usuário ruim.

b) Não deve tratar com concorrência

- Nesse caso, não é necessário concorrência para lidar com varias requisições de download, pois assim que uma começo as outras vão iniciar logo em seguida. uma após a outra.

c) Deve ser tratada com técnicas de concorrência

- Em sistemas operacionais para dispositivos móveis, é necessário realizar requisições de rede separadamente da thread principal para evitar que a interface gráfica fique bloqueada ou não responsiva enquanto a rede é acessada, garantindo uma melhor experiência do usuário.

7 - a) Ação de executar múltiplas tarefas que compartilham recursos:

- **Correta.** Concorrência envolve a execução de múltiplas tarefas que podem estar progredindo de forma independente, mas competem por recursos compartilhados. Em um sistema, onde diversas tarefas podem ser executadas de maneira intercalada, permitindo que os recursos sejam utilizados de forma eficiente.

b) **Que enquanto alguém utiliza um recurso, o outro espera a sua vez**:

- **Correta.** Este é um comportamento típico em sistemas concorrentes. Quando múltiplas tarefas competem por um mesmo recurso, uma tarefa pode precisar esperar até que o recurso esteja disponível. Isso é gerenciado por mecanismos de sincronização como locks, semáforos, etc.

ou incorreta depende da interpretação do documento. Caso seja apenas leitura sem alteração, pode ter mais de um acesso simultâneo.

c) **Uma composição de processos iguais e dependentes executando**:

- **Incorreta.** Concorrência não se refere especificamente a processos iguais e dependentes. Pode envolver a execução de tarefas diferentes, que podem ou não ser dependentes entre si. O foco está na administração do acesso aos recursos compartilhados e na coordenação das tarefas.

d) **A mesma coisa que paralelismo**:

- **Incorreta.** Concorrência e paralelismo são conceitos relacionados, mas distintos. Concorrência se refere à composição de tarefas que podem fazer progresso de forma independente, compartilhando o tempo de CPU. Paralelismo, por outro lado, envolve a execução simultânea de múltiplas tarefas em diferentes núcleos ou processadores, o que exige hardware de múltiplos núcleos.

8 - a)A afirmação está equivocada. Porque a concorrência nem sempre melhora o desempenho. Embora em muitos casos a execução simultânea de tarefas possa aumentar a eficiência, em alguns casos pode introduzir sobrecarga devido à necessidade de sincronização e coordenação entre threads ou processos. Além disso, a concorrência pode introduzir complexidade adicional ao código, o que pode levar a bugs difíceis de detectar e corrigir.

complemento durante a aula: aumento do trabalho por gerenciar múltiplas threads, não pensar que concorrência e paralelismo são a mesma coisa. Programa paralela?

b) Esta afirmação está equivocada. Ao criar programas concorrentes, o projeto de software precisa ser revisado e ajustado para lidar com as complexidades adicionais introduzidas pela concorrência. Isso pode incluir a identificação e a proteção de regiões críticas do código, a implementação de mecanismos de sincronização e a garantia de que os recursos compartilhados sejam manipulados de forma segura por múltiplas threads ou processos.

c) A afirmação está equivocada. Conseguir concorrência completa, onde todas as partes de um programa podem ser executadas simultaneamente, é uma tarefa complexa e muitas vezes desafiadora. Isso envolve lidar com condições de corrida, sincronização de acesso a recursos compartilhados, garantia de consistência de dados e resolução de problemas de concorrência, como deadlock e starvation, aumento de complexidade. Pode acontecer um caso não previsto que irá gerar inconsistência.

complemento durante a aula. que o código é 100% confiável

9 - a) Resultados possíveis

Supondo que inicialmente `lastIdUsed` tem o valor 10, 11 ou 12, as situações possíveis são:

Situação 1: `lastIdUsed` inicial é 10

1. Thread A chama `getNextId()`:
    - `lastIdUsed` é incrementado para 11

    - A retorna 11
2. Thread B chama `getNextId()`:
    - `lastIdUsed` é incrementado para 12
    - B retorna 12

No entanto, se houver interferência entre as threads (sem sincronização adequada), resultados inesperados podem ocorrer. Por exemplo:

- Se ambas as threads leem o valor de `lastIdUsed` (10) quase ao mesmo tempo e depois incrementam, ambas podem tentar retornar 11, o que resultaria em duplicação de IDs.

Situação 2: `lastIdUsed` inicial é 11

1. Thread A chama `getNextId()`:
    - `lastIdUsed` é incrementado para 12
    - A retorna 12
2. Thread B chama `getNextId()`:
    - `lastIdUsed` é incrementado para 13
    - B retorna 13

Com interferência (sem sincronização adequada):

- Ambas as threads podem retornar 12 se lerem o valor inicial ao mesmo tempo.

Situação 3: `lastIdUsed` inicial é 12

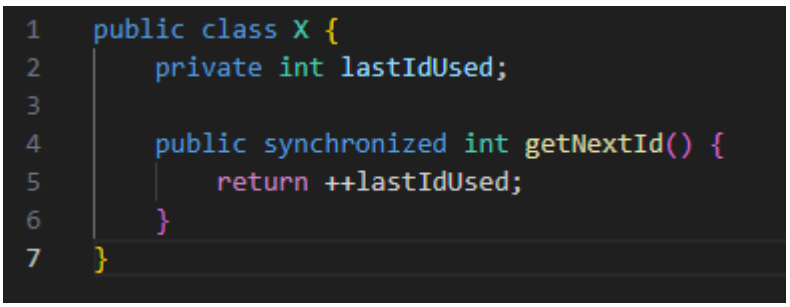
1. Thread A chama `getNextId()`:
    - `lastIdUsed` é incrementado para 13
    - A retorna 13
2. Thread B chama `getNextId()`:
    - `lastIdUsed` é incrementado para 14
    - B retorna 14

Com interferência (sem sincronização adequada):

- Ambas as threads podem retornar 13 se lerem o valor inicial ao mesmo tempo.

b) Alterações para resolver o problema

Para resolver o problema de interferência entre threads e garantir que cada chamada a `getNextId()` retorna um ID único, podemos usar sincronização para assegurar que apenas uma thread pode acessar o método `getNextId()` por vez.



```java
public class X {
    private int lastIdUsed;

    public synchronized int getNextId() {
        return ++lastIdUsed;
    }
}
```

erificação das situações propostas no item a)

1. **Situação 1: `lastIdUsed` inicial é 10**
    - Thread A chama `getNextId()` e incrementa `lastIdUsed` para 11, retornando 11.
    - Thread B chama `getNextId()` e incrementa `lastIdUsed` para 12, retornando 12.
    - Resultados garantidos: 11 e 12 (sem duplicação)
2. **Situação 2: `lastIdUsed` inicial é 11**
    - Thread A chama `getNextId()` e incrementa `lastIdUsed` para 12, retornando 12.
    - Thread B chama `getNextId()` e incrementa `lastIdUsed` para 13, retornando 13.
    - Resultados garantidos: 12 e 13 (sem duplicação)
3. **Situação 3: `lastIdUsed` inicial é 12**
    - Thread A chama `getNextId()` e incrementa `lastIdUsed` para 13, retornando 13.
    - Thread B chama `getNextId()` e incrementa `lastIdUsed` para 14, retornando 14.
    - Resultados garantidos: 13 e 14 (sem duplicação)

A sincronização garante que cada thread tenha acesso exclusivo ao método `getNextId()`, evitando interferências e garantindo IDs únicos e ordenados.