Equipe Corvo Manco

Participantes: Alisson, Ester, Luis Felipe, Jose Carlos, Vinicius.

Questão 1

Para resolver o problema da parte 1 é possível utilizar o padrão de comunicação entre threads conhecido como Produtor-Consumidor. A arquitetura dessa solução envolve um buffer que serve como ponto intermediário entre as threads produtoras e consumidoras.

Componentes Principais

1. **Mailbox**: O buffer que armazena a mensagem.
2. **Producer**: Uma thread que produz mensagens e as coloca no Mailbox.
3. **Consumer**: Uma thread que consome mensagens do Mailbox.

Sincronização e Coordenação

- **Sincronização**: Garantir que apenas uma thread possa acessar ou modificar a mensagem no Mailbox por vez.
- **Coordenação**: Assegurar que as threads produtoras esperem até que o Mailbox esteja vazio para colocar uma nova mensagem e que as threads consumidoras esperem até que o Mailbox esteja cheio para consumir a mensagem.

Pseudocódigo

Mailbox

```
class Mailbox:
    private message: String = null
    private lockObject = new Lock()

    method storeMessage(msg: String):
        lock(lockObject):
            while message is not null:
                wait(lockObject)
            message = msg
            notifyAll(lockObject)

    method retrieveMessage() -> String:
        lock(lockObject):
```

```
            while message is null:
                wait(lockObject)
            msg = message
            message = null
            notifyAll(lockObject)
            return msg
```

Producer

```
class Producer:
    private mailbox: Mailbox
    private id: String
    private thread: Thread

    constructor(mailbox: Mailbox, id: String):
        this.mailbox = mailbox
        this.id = id
        this.thread = new Thread(run)

    method start():
        thread.start()

    private method run():
        messageCount = 0
        while True:
            message = id + " message " + messageCount
            mailbox.storeMessage(message)
            print(id + " produced: " + message)
            sleep(1000)
```

Consumer

```
class Consumer:
    private mailbox: Mailbox
    private id: String
    private thread: Thread

    constructor(mailbox: Mailbox, id: String):
```

```
        this.mailbox = mailbox
        this.id = id
        this.thread = new Thread(run)

    method start():
        thread.start()

    private method run():
        while True:
            message = mailbox.retrieveMessage()
            print(id + " consumed: " + message)
            sleep(1500)
```

Teste

```
class Test:
    method main():
        mailbox = new Mailbox()

        producer1 = new Producer(mailbox, "Producer1")
        producer2 = new Producer(mailbox, "Producer2")

        consumer1 = new Consumer(mailbox, "Consumer1")
        consumer2 = new Consumer(mailbox, "Consumer2")

        producer1.start()
        producer2.start()
        consumer1.start()
        consumer2.start()

        wait()
```

Fluxo de Execução

1. **Inicialização**: As threads produtoras e consumidoras são iniciadas.
2. **Produtor**:
    - Tenta armazenar uma mensagem no `Mailbox`.

    - Se o `Mailbox` já contém uma mensagem, a thread produtora espera (`wait`) até que o `Mailbox` esteja vazio.
    - Quando o `Mailbox` está vazio, a thread produtora armazena a mensagem e notifica todas as threads (`notifyAll`) de que o estado do `Mailbox` mudou.
3. **Consumidor**:
    - Tenta recuperar uma mensagem do `Mailbox`.
    - Se o `Mailbox` está vazio, a thread consumidora espera (`wait`) até que o `Mailbox` tenha uma mensagem.
    - Quando o `Mailbox` contém uma mensagem, a thread consumidora recupera a mensagem e notifica todas as threads (`notifyAll`) de que o estado do `Mailbox` mudou.

Exclusão Mútua e Comunicação

- **Exclusão Mútua**: O uso do `lock` (ou `Monitor` em C#) garante que apenas uma thread pode executar a seção crítica de código que manipula a mensagem do `Mailbox`.
- **Comunicação**: O uso de `wait` e `notifyAll` permite que as threads se comuniquem indiretamente, esperando e notificando sobre mudanças no estado do `Mailbox`.

Essa arquitetura garante que a produção e o consumo de mensagens sejam coordenados de maneira eficiente e segura, prevenindo condições de corrida e garantindo que as threads produtoras e consumidoras operem de forma síncrona.

Questão 2: Explicação da Figura (Exclusão Mútua)

A figura demonstra a operação de exclusão mútua entre dois processos, A e B. O gráfico de tempo mostra quando cada processo entra e sai de suas **regiões críticas** (parte do código onde o recurso compartilhado é acessado). A exclusão mútua é um conceito importante em sistemas concorrentes onde há compartilhamento de recursos por threads ou processos paralelos. A ideia é garantir que apenas um processo possa entrar na região crítica em um dado momento, para evitar inconsistências de dados.

- **Processo A**:
    - **T1**: Processo A entra na região crítica.
    - **T3**: Processo A deixa a região crítica.
- **Processo B**:
    - **T2**: Processo B tenta entrar na região crítica, mas é bloqueado porque o Processo A já está na região crítica. Isso demonstra a função da exclusão mútua, que impede o acesso simultâneo.
    - **T3**: Processo B entra na região crítica após o Processo A sair.
    - **T4**: Processo B deixa a região crítica.

Questão 3: Discussão sobre Exclusão Mútua com Espera Ocupada

Prós:

- **Simplicidade**: A solução é simples e fácil de entender.
- **Garantia de Exclusão Mútua**: Assegura eficazmente que apenas um processo entre na região crítica por vez.

Contras:

- **Espera Ocupada**: Os processos gastam ciclos de CPU verificando ativamente a condição, o que é ineficiente em termos de uso de recursos.
- **Starvation**: Existe a possibilidade de que um processo que está bloqueado aguarde indefinidamente que o outro processo saia da região crítica (possivelmente nunca entrando na região crítica).
- **Sem definição de prioridades:** não há definição nenhuma de qual thread teria prioridade para entrar na região crítica.

Outras Soluções:

- **Semaforos**: Utilizar semáforos para controlar o número de processos permitidos na região crítica ao mesmo tempo.

Questão 4: Processos e threads

Processos são programas em execução que disputam os recursos do processador, enquanto as threads são unidades de execução dentro de um processo. Cada processo possui seu próprio espaço de memória, enquanto as threads compartilham o mesmo espaço de memória.

A principal diferença é que threads são mais leves e eficientes para tarefas que exigem comunicação frequente ou compartilhamento de dados, enquanto processos oferecem maior isolamento e segurança, sendo adequados para tarefas que requerem independência entre as execuções.

Questão 5:

O conceito de Safety objetiva garantir que uma situação crítica ou indesejada nunca aconteça. Um exemplo é um sistema bancário, onde operações simultâneas, por meio de duas threads, podem resultar em um saldo negativo. Nesse caso, o objetivo é garantir que o saldo da conta nunca seja negativo, ou que o proprietário da conta não seja prejudicado pelas operações concorrentes.
Já o conceito de Liveness visa garantir que um resultado desejado eventualmente aconteça. Um exemplo seria uma fila de processos First-In First-Out. Nesse caso, o objetivo seria garantir que todos os processos, em algum momento, entrem em execução.

Questão 6:

a) Sim. A concorrência permite que a thread de download notifique a interface gráfica periodicamente sobre o progresso, permitindo atualizações em tempo real. Além disso manter o download em uma thread separada permite que a interface gráfica continue responsiva e possa processar outras interações do usuário
b) Sim. Com concorrência cada requisição pode ser tratada independentemente, reduzindo a possibilidade de uma requisição lenta ou problemática afetar outras requisições.
c) Sim. Para manter a interface gráfica responsiva.

Questão 7:

As alternativas corretas são A e B. Na concorrência tarefas ou processos simultâneos competem por recursos e precisam se coordenar para que a ultilização dos recuros seja feita de maneira sincronizada sem causar inconsistências de dados e outros probemas.

Questão 8:

a) **Overhead**: Concorrência pode introduzir a necessidade de criar e gerenciar múltiplas threads ou processos, além de coordenar o acesso a recursos compartilhados. Esse overhead pode, em alguns casos, levar a um desempenho pior do que uma implementação sequencial.

   Contention e Deadlock: Concorrência pode introduzir problemas como contenção (várias threads tentando acessar o mesmo recurso ao mesmo tempo) e deadlocks (duas ou mais threads bloqueadas esperando umas pelas outras), o que pode degradar o desempenho ou até paralisar o sistema.

   False Sharing: Em sistemas com múltiplos núcleos, o compartilhamento falso de cache (quando threads em diferentes núcleos alteram variáveis que estão próximas na memória) pode reduzir o desempenho devido ao excesso de invalidações de cache.

   Carga de Trabalho: Nem todas as cargas de trabalho são adequadas para concorrência. Tarefas que são estritamente sequenciais ou que possuem um alto grau de dependência entre etapas podem não se beneficiar significativamente da concorrência.

b) **Design e Arquitetura**: Projetar software concorrente frequentemente requer mudanças substanciais na arquitetura do programa. Isso inclui considerar a decomposição do problema em tarefas concorrentes, identificar pontos de sincronização, e evitar condições de corrida.

Mecanismos de Sincronização: Implementar concorrência requer o uso de mecanismos de sincronização como locks, semáforos, monitores, e outros, o que complica o design e implementação.

Depuração e Testes: Programas concorrentes são geralmente mais difíceis de depurar e testar devido à natureza não determinística das interações entre threads. Bugs concorrentes, como condições de corrida e deadlocks, são notoriamente difíceis de reproduzir e corrigir.

Estratégias de Particionamento: Decidir como dividir o trabalho entre múltiplas threads ou processos requer novas estratégias e pode impactar profundamente a estrutura do código.

c) **Complexidade Adicional**: A introdução de concorrência adiciona complexidade significativa ao software. Gerenciar a comunicação e sincronização entre múltiplas threads ou processos requer um cuidado especial para evitar problemas como deadlocks, starvation, e race conditions.

Corretude e Segurança: Garantir que o software concorrente funcione corretamente e seja seguro (isto é, livre de bugs de concorrência) é uma tarefa desafiadora. Pequenos erros podem levar a problemas graves e difíceis de diagnosticar.

Escalabilidade: Criar uma solução concorrente que escale bem com o número de threads ou processos e que aproveite eficientemente os recursos disponíveis (como múltiplos núcleos de CPU) requer um design cuidadoso e frequentemente envolve trade-offs complexos.

Determinismo: Concorrência introduz não-determinismo, tornando o comportamento do programa dependente do timing e da ordem de execução das threads, o que pode ser imprevisível e difícil de controlar.