Laboratório 2

Este laboratório é composto de quatro exercícios que devem ser realizados usando os arquivos fornecidos. São dois exercícios obrigatórios e dois exercícios facultativos. O código deve estar identado, organizado e comentado. A entrega do laboratório deverá ser feita até o dia 29/11 às 23:59, através de um arquivo zip na tarefa do Google Classroom.

Exercício 1: Contador Seguro com Mutex (5 pontos)

Objetivo: Familiarizar-se com o gerenciamento de mutex em C++ para lidar com variáveis compartilhadas entre threads.

Desenvolvimento do Programa: Crie um arquivo `ex1.cpp` e desenvolva um programa em C++ que simule um contador compartilhado entre várias threads. O objetivo é garantir que as operações de incremento e decremento no contador sejam seguras, evitando condições de corrida.

Instruções:

- Crie uma variável inteira compartilhada chamada `contador` e inicialize-a com zero.
- Crie um `mutex` (trava) que será usado para proteger o acesso ao contador compartilhado.
- Crie duas funções diferentes: `incrementar` e `decrementar`.
  - A função `incrementar` deve ser executada por uma thread e deve incrementar o contador em uma unidade após adquirir o mutex. Após a operação, ela deve liberar o mutex.
  - A função `decrementar` também deve ser executada por outra thread e deve decrementar o contador em uma unidade após adquirir o mutex. Após a operação, ela deve liberar o mutex.
- Crie 8 threads que chamem alternadamente as funções incrementar e decrementar. Certifique-se de que essas threads compartilhem o mesmo mutex para garantir a exclusão mútua.
- Implemente uma função principal que inicie as threads e as aguarde terminarem.
- Ao final da execução do programa, verifique se o valor do contador é zero. Isso deve ser verdadeiro, pois as operações de incremento e decremento estão balanceadas.

Dicas: Use a biblioteca `thread` para criar e gerenciar threads e a biblioteca `mutex` para criar e gerenciar mutexes.

Entrega: O arquivo `ex1.cpp` deve estar em um único arquivo zip com os arquivos do exercício 2, e devem ser entregues no Google Classroom.

Exercício 2: Sistema Bancário (5 pontos)

Objetivo: Familiarizar-se com um gerenciamento mais complexo de mutex em C++, lidando com possíveis deadlocks.

Desenvolvimento do Programa: Crie um arquivo `ex2.cpp` e simule um sistema bancário com várias contas bancárias e transações entre essas contas. Cada conta bancária é protegida por um `mutex` para evitar acessos concorrentes que possam levar a inconsistências nos saldos.

Instruções:

- Crie uma classe ContaBancaria que contenha os seguintes campos:
  - Um número de conta único.
  - Um saldo.
  - Um mutex para proteger o acesso ao saldo.
- Crie um vetor ou lista de contas bancárias, por exemplo, `std::vector<ContaBancaria> contas`, e inicialize-o com várias contas, cada uma com um saldo diferente.
- Após inicializar as contas, imprima no terminal o número da conta e seu saldo.
- Crie uma função de transferência `transferir` que aceita duas contas bancárias como parâmetros e um valor a ser transferido da primeira conta para a segunda. Certifique-se de que a transferência seja feita com exclusão mútua, ou seja, apenas uma thread pode transferir dinheiro de uma conta por vez. Após finalizar a transferência, imprima uma mensagem informando quais contas estão envolvidas na transferência e o valor transferido.
- Crie uma função de simulação que realiza várias transações entre as contas. Por exemplo, você pode criar um loop que escolhe aleatoriamente duas contas e realiza uma transferência de uma conta para a outra.
- Crie várias threads que executam a função de simulação de transações concorrentemente.
- Execute as threads e observe se ocorre um `deadlock`. Um deadlock pode ocorrer se as threads adquirirem mutexes em ordens diferentes e aguardarem indefinidamente que o outro mutex seja liberado.
- Experimente trocar a ordem de aquisição dos mutexes nas threads ou use outras estratégias para evitar o deadlock, como timeouts ou detecção de deadlock.
- Ao final do programa, imprima as informações das contas novamente.
- Crie um arquivo em de texto `ex2.pdf` explicando brevemente por que o deadlock ocorre e qual estratégia você utilizou para evitá-lo.

Observações: Utilize um lock para imprimir as mensagens e evitar que elas sejam cortadas. Este exercício representa um cenário mais realista com várias contas bancárias e transações entre elas, onde a ordem de aquisição dos mutexes

é crucial para evitar deadlock em sistemas concorrentes. Certifique-se de compreender por que o deadlock ocorre neste exercício e como resolvê-lo ajustando a ordem de aquisição dos mutexes ou usando outros mecanismos de prevenção de deadlock.

Entrega: O arquivo `ex2.cpp` e `ex2.pdf` devem estar em um único arquivo zip com os arquivos do exercício 1, e devem ser entregues no Google Classroom.

Exercício 3: Simulação de Restaurante com Garçons e Clientes (Facultativo)

Objetivo: Familiarizar-se com a utilização de variáveis de condição em C++, lidando com sincronização.

Desenvolvimento do Programa: Implemente um programa em C++ que simule um restaurante com garçons e clientes usando threads, mutexes e variáveis de condição para garantir um funcionamento sincronizado.

Instruções:

- Crie um restaurante com vários garçons e clientes. Os garçons e clientes serão representados por threads.
- O restaurante possui um número limitado de mesas disponíveis. Implemente um sistema para controlar a disponibilidade das mesas usando mutex e variável de condição.
- Implementação do cliente:
  - Cada cliente é uma thread que deseja ser acomodada em uma mesa.
  - O número de clientes deve ser maior que o número de mesas.
  - Quando a thread do cliente for iniciada você deve imprimir a mensagem `Cliente <id_c> está aguardando uma mesa.`, em que `<id_c>` é o número de identificação do cliente.
  - Se todas as mesas estiverem ocupadas, o cliente deve aguardar até que uma mesa esteja disponível.
  - Após ser acomodado em uma mesa, o cliente deve imprimir a mensagem `Cliente <id_c> sentou na mesa <id_m>.`, em que `<id_m>` é o número de identificação da mesa.
  - O cliente acomodado vai aguardar ser atendido por um garçom. Após o atendimento, o cliente deve imprimir a mensagem `Cliente <id_c> recebeu seu pedido.` e liberar sua mesa.
- Implementação do garçom:
  - Cada garçom é uma thread que atende os clientes nas mesas.
  - O número de garçons deve ser menor que o número de mesas.
  - Quando a thread do garçom for iniciada você deve imprimir a mensagem `Garçom <id_g> está disponível.`, em que `<id_g>` é o número de identificação do garçom.
  - Os garçons podem atender apenas uma mesa por vez.

  - Após atender um cliente, o garçom deve imprimir a mensagem `Garçom <id_g> atendeu o cliente <id_c> na mesa <id_m>`.
- Implemente um sistema para permitir que os garçons atendam os clientes em ordem. Por exemplo, um garçom deve atender um cliente por vez e, depois de atender um cliente, ele pode escolher o próximo cliente na fila.
- Use mutexes e variáveis de condição para sincronizar o acesso ao buffer de mesas e as operações de alocação de mesas. Garanta que os clientes aguardem quando todas as mesas estiverem ocupadas e que os garçons aguardem quando não houver clientes a serem atendidos.
- Implemente uma função principal que cria as threads de garçons e clientes e aguarde a conclusão de todas as threads.
- Certifique-se de que o programa funcione de forma sincronizada e evite problemas como clientes sendo acomodados em mesas já ocupadas ou garçons atendendo várias mesas simultaneamente.

Observações: Utilize um lock para imprimir as mensagens e evitar que elas sejam cortadas. Além disso, tente imprimir em cores diferentes as mensagens dos garçons e dos clientes. Os identificadores de garçons, mesas e clientes não precisam ser diferentes entre si, por exemplo, pode ter um garçom e um cliente com identificador 0, porém dois garçons não podem ter o mesmo identificador.

Exercício 4: Implementação Paralela do Merge Sort (Facultativo)

Objetivo: Explorar as dificuldades de paralelização em um algoritmo de ordenação existente, lidando com variáveis compartilhadas e sincronização.

Desenvolvimento do Programa: Analise o arquivo `ex4.cpp` que contém a implementação serial do algoritmo de ordenação Merge Sort. Seu objetivo é modificar esse código serial para torná-lo paralelo usando threads.

Instruções:

- Você receberá o código serial do algoritmo Merge Sort, que já ordena um vetor de inteiros.
- Sua tarefa é modificar esse código para torná-lo paralelo, de forma que a ordenação possa ser acelerada usando threads.
- Você deve criar uma versão paralela do algoritmo Merge Sort que divide o vetor em partes menores e utiliza threads para ordenar essas partes de forma independente. Em seguida, as partes ordenadas devem ser mescladas de forma adequada.
- Utilize mutexes ou outras técnicas apropriadas para evitar condições de corrida e garantir que as threads cooperem corretamente durante a ordenação paralela.
- Implemente uma função principal que leia a entrada, chame a versão paralela do Merge Sort com o número especificado de threads e aguarde a conclusão de todas elas.

- Teste o programa modificado com diferentes tamanhos de entrada e números de threads para verificar o desempenho da ordenação paralela em comparação com a versão serial.
- Imprima o vetor ordenado após a conclusão da ordenação para verificar a corretude do algoritmo paralelo.

Observações: O objetivo deste exercício é aprender a paralelizar um algoritmo de ordenação existente e compreender os desafios envolvidos na programação paralela. Certifique-se de que as threads estejam cooperando corretamente e evitando problemas de concorrência durante a ordenação paralela.