UNIVERSIDADE FEDERAL DE SANTA CATARINA

CAMPUS TRINDADE

INE-DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

INE5410 - PROGRAMAÇÃO CONCORRENTE

Alunos:

João Victor Cabral Machado,	Pedro Alfeu Wolff Lemos,	Tális Breda
22204113	**22200373**	**22102202**

Relatório do Trabalho 1

Florianópolis

2023

Sumário

1. Introdução

O programa em questão é um simulador de um bar que fornece rodadas grátis para seus clientes antes de fechar. O bar possui garçons que atendem um número limitado de clientes por vez, e cada cliente deve aguardar um tempo aleatório antes de fazer um novo pedido. O programa foi implementado em C, utilizando a biblioteca POSIX PThreads, com o uso de semáforos, mutexes e threads.

Antes de fazer um pedido o cliente engaja em conversa com amigos, e após ter seu pedido atendido, o cliente vai aguardar o tempo necessário para seu pedido chegar e comer. A rodada só poderá ser avançada quando todos os clientes que fizeram pedidos foram atendidos.

O programa utiliza as threads para representar os clientes e os garçons. Elas são utilizadas para simular o comportamento dos dois grupos. Elas são diretamente responsáveis por simular as conversas, os pedidos, a espera e todas as ações dos garçons.

A abordagem adotada neste trabalho mostra a importância da alocação correta dos recursos, a aplicação dos semáforos e mutexes para sincronizar as operações e o controle das etapas da simulação.

No decorrer desse relatório, iremos explicar a estrutura do código e os conceitos de programação concorrente que utilizamos para realizar o projeto.

2. Inicialização do Bar e Configuração das estruturas

O programa inicializa lendo os parâmetros da linha de comando:

- Número de clientes (N);
- Garçons(G);
- Clientes por garçom (Gn);
- Total de rodadas (R);
- Tempo máximo de conversa
- Tempo máximo de consumo.

Os argumentos são checados um por um, pois devem ser apenas números inteiros positivos para que o programa funcione corretamente. Cria-se a estrutura de dados do bar, contendo os dados obtidos dos argumentos, e aloca-se memória para os garçons e clientes. Também são inicializados os mutexes e semáforos do bar. Depois que todas as threads foram criadas, o status **inicializado** do bar é ligado, indicando para as threads que elas podem começar a rodar.

As estruturas do programa são conforme descrito a seguir:

2.1. Cliente

Cliente possui todas as informações necessárias para o cliente, que são apenas seu id, a respectiva thread e um semáforo que controla o recebimento do pedido.

2.2. Garcom

Assim como a estrutura anterior, possui as informações necessárias para a execução correta da thread, incluindo a thread e o id. Porém, o garçom precisa de outras informações para funcionar corretamente, como uma fila de pedidos e o seu tamanho, um status, e o número de clientes aguardando atendimento. Existem também três mutexes que controlam a edição dessas informações.

2.3. StatusBar

Essa estrutura tem as informações gerais do bar. Inclui os argumentos passados na execução do programa, alguns mutexes, algumas variáveis de status (fechado, inicializado) e a lista de garçons do bar.

2.4. ArgThreadCliente e ArgThreadGarcom

Foram criados como uma interface de comunicação para evitar dependência circular. ArgThreadCliente possui um ponteiro para um Cliente e outro para o StatusBar. ArgThreadGarcom possui um ponteiro para um Garcom e outro para o StatusBar.

3. Implementação das Threads

3.1. Threads

As threads são criadas no início do programa. Cada cliente e garçom são representados por uma thread separada.

Para controlar o acesso concorrente aos recursos compartilhados, como a fila de pedidos, semáforos e mutexes são usados para garantir a exclusão mútua e sincronização entre as threads. A simulação usa semáforos para notificar threads quando determinados eventos ocorrem, como quando um garçom está disponível para atender pedidos ou quando um pedido é entregue a um cliente.

3.2. clienteThread

Essa função é responsável pelo comportamento de um cliente, ela inicia o ciclo apenas com o bar aberto, dentro do ciclo: o cliente começa a conversar com os amigos, que simula uma pausa aleatória para uma conversa.

Então o cliente faz o pedido para o garçom chamando a função **fazPedido**, com o pedido feito o cliente aguarda o pedido ser entregue, chamando **esperaPedido**, uma vez que o pedido tenha chego, o cliente o consome, usando a função **consomePedido**.

Esse ciclo continua até que o bar seja fechado, momento em que o cliente termina seu comportamento e sai

3.3. garçomThread

Essa função é responsável pelo comportamento de um garçom, começando seu ciclo com o bar aberto. No início de cada rodada o garçom fica disponível para atender de clientes usando **recebeMaximoPedidos**, estando disponível, o garçom aguarda pedidos.

Quando o pedido é feito o garçom registra-os chamando **registraPedidos** e entrega os pedidos, chamando **entregaPedidos**. Esse ciclo continua até que a rodada seja concluída.

O garçom aguarda a notificação do término da rodada ou até ele não poder mais registrar pedidos, este ciclo se repete até o fechamento do bar

4. Sincronização entre Threads

Descrição: O programa deve sincronizar corretamente os diversos garçons e clientes, evitando condições de corrida e deadlocks. Os garçons devem entregar os pedidos na ordem em que foram feitos.

Solução: A sincronização entre as threads é feita usando semáforos e mutexes.

O cliente possui um semáforo, **aguardandoPedido**, que é decrementado por ele mesmo quando faz o pedido, e incrementando novamente pelo garçom quando este entrega esse pedido.

O garçom possui dois semáforos: **disponivel** é incrementado quando o garçom fica livre para atender um pedido, e diz aos clientes que eles podem chamá-lo. Ao incrementar esse semáforo, o garçom em sequência espera em outro semáforo, **pedidoRealizado**, que é incrementado quando um cliente faz um pedido.

Além dos semáforos, alguns mutexes particulares a cada garçom também são usados, principalmente para controlar o acesso à fila de pedidos do garçom, que é uma parte crítica do código. Também são usados mutexes para controlar o status e o número de clientes aguardando atendimento.

A fila é implementada de tal forma que garante que os pedidos serão entregues na ordem em que foram realizados, pois são inseridos em posições sucessivas, que são acessadas na mesma ordem na hora da entrega.

5. Simulação de Rodadas e Fechamento do Bar

Descrição: O bar deve operar por um número específico de rodadas e fechar quando todas as rodadas forem concluídas.

Solução: Sempre que um garçom entrega Gn pedidos, ele incrementa o contador de garçons finalizados do bar, e espera no semáforo **semaforo_rodada**. Quando o número de garçons finalizados for igual ao total de garçons e ainda não foram realizadas R rodadas, uma nova rodada é iniciada, e **semaforo_rodada** é incrementado G vezes. Caso todas as R rodadas tiverem sido realizadas, o status **fechado** do bar é mudado para 1, e os garçons e clientes param de executar.

6. Saída e Exibição de Resultados

Descrição: A saída do programa deve fornecer informações detalhadas sobre o funcionamento do bar em cada rodada, incluindo os pedidos dos clientes, atendimentos pelos garçons e outros eventos relevantes.

Solução: Cada ação do garçom ou do cliente é impressa no terminal. Desde quando o cliente está conversando com seus amigos, quando ele termina, quando encontra um garçom, quando verifica o status dele, entre diversas outras situações. Para o garçom, é registrado no terminal quando ele está esperando um pedido, é mostrada a fila de pedidos de cada garçom cada vez que um novo pedido é feito, além de ser mostrado quando os pedidos foram entregues e para qual cliente.

Para auxiliar com as impressões, foram criados três métodos: **printText**, **printFilaDePedidos**, e **printTerminoDeRodada**. Além deles, foi criado também um **mutex_print** global, que é utilizado para os casos onde uma etapa precisa imprimir várias linhas consecutivas. O mutex permite que não hajam outras threads concorrendo pela saída enquanto outra imprime.

6.1. printText

É um método bem simples, criado apenas para evitar a poluição visual do código por conta do uso do fflush(stdout). Ao invés de usar um printf, as threads chamam essa função

quando precisam imprimir algo, e ela já executa o que é necessário. Para gerenciamento dos argumentos foi usada a struct *va_list* da biblioteca stdarg.h.

6.2. printFilaDePedidosDo(garcom)

Esse método é chamado ao ser realizado um novo pedido a um garçom. Ele faz uso do mutex_print para imprimir todos os pedidos atualmente na fila do garçom em sequência, sem que outra thread interrompa. É garantido que, ao chamar esse método, a thread está em uma região controlada pelo **mutex_fila** desse garçom, pois os elementos da fila de pedidos são acessados.

6.3. printTerminoDeRodada

Esse método também faz uso do **mutex_print** para imprimir várias linhas em sequência. Ele é chamado cada vez que um garçom termina uma rodada, e apresenta essa informação.

7. Gerenciamento de Memória

7.1. Alocação e Desalocação de Recursos

Cada cliente e garçom é representado por uma estrutura de dados, por exemplo, uma estrutura **Cliente** e uma estrutura **Garçom**. Essas estruturas são alocadas dinamicamente na memória quando as threads de clientes e garçons são criadas usando **malloc** e desalocadas quando as threads usando **free**. Isso garante que a memória alocada seja liberada adequadamente quando não for mais necessária

7.2. Filas de Pedidos

As filas de pedidos para cada garçom foram implementadas usando estrutura de dados dinâmicas. É importante garantir que a memória seja alocada para as estruturas de dados da fila de pedidos e desalocada quando os pedidos são atendidos e removidos da fila. Isso evita vazamentos de memória

7.3. Variáveis Locais

Dentro das funções **clienteThread** e **garcomThread**, variáveis locais são usadas para armazenar dados temporários e resultados intermediários. Essas variáveis são alocadas na pilha e automaticamente desalocadas quando as funções retornam.

8. Conclusão

Em conclusão, nosso grupo adquiriu uma série de conhecimentos e experiências valiosas. Em primeiro lugar aprendemos a importância da programação concorrente e como o conceito de semáforos e mutexes podem ser aplicados para sincronizar threads de maneira eficaz. Esta experiência destacou a complexidade de situações do mundo real e como as soluções podem ser adaptadas para lidar com desafios dinâmicos.

Em segundo lugar, este projeto demonstrou a importância da organização e documentação do código. Com uma estrutura bem definida e comentários claros, fomos capazes de manter um código limpo, fácil de entender e que se mantém robusto. A ausência de vazamento de memória também reforça a importância da gestão de recursos em projetos de programação.

Por último, a simulação nos permitiu compreender a interação entre as partes de um sistema complexo, levando-se a uma apreciação mais profunda dos desafios envolvidos na tomada de decisões e na coordenação de tarefas em cenários concorrentes. No geral, este projeto foi uma oportunidade de aprendizado enriquecedora, destacando a aplicação prática dos princípios de programação, organização de código e resolução de problemas em cenários do mundo real. Esses conhecimentos adquiridos sem dúvida enriqueceram nossos futuros empreendimentos na área da computação e engenharia de software.