CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

Unidade Leopoldina

Engenharia da Computação

Problema de sincronização usando threads

Parte 1

LABORATÓRIO DE SISTEMAS DE TEMPO REAL

GABRIEL RIBEIRO PASSOS

IURI SOUSA WERNECK PEREIRA

VICTOR DE SOUZA VILELA DA SILVA

Leopoldina, MG, Brasil

novembro de 2023

Relatório do Projeto - Problema de sincronização usando threads - Parte 1

Resumo do Projeto

O projeto tem como objetivo desenvolver um sistema para a única casa lotérica da cidade fictícia de Leopoldinópolis, onde apenas um operador de caixa está disponível. O sistema é projetado para gerenciar o atendimento aos clientes, considerando regras de prioridade e evitando problemas como inanição e deadlock.A estrutura do sistema inclui definições de dados para representar informações individuais dos clientes e uma fila de atendimento eficiente. As regras de prioridade estabelecem que grávidas têm prioridade sobre idosos, idosos têm prioridade sobre pessoas com deficiência, e estas, por sua vez, têm prioridade sobre pessoas comuns. As pessoas comuns são atendidas por ordem de chegada.Para evitar inanição, o sistema introduz o conceito de "frustração", permitindo o aumento gradual da prioridade de um cliente após duas experiências consecutivas de ser ultrapassado na fila. O caixa é liberado para o próximo cliente com prioridade mais alta, exceto em casos específicos que visam prevenir inanição e deadlock.No que diz respeito aos casais, eles compartilham a mesma prioridade com base no cônjuge associado. Se ambos os membros do casal estiverem presentes, o atendimento segue a ordem de chegada.A implementação é realizada exclusivamente em C, sem o uso de recursos de outras linguagens. São utilizados pthreads e mutexes para garantir a sincronização adequada e acesso seguro aos recursos compartilhados.O sistema simula o ambiente de atendimento em uma casa lotérica, seguindo regras específicas de prioridade e implementando estratégias para evitar problemas potenciais durante o atendimento aos clientes.

Estrutura Geral do Código

1. **Estruturas de Dados:**
    a. **Pessoa:**
        i. Armazena informações sobre cada pessoa, como nome, prioridade, quantidade de uso do caixa, índice, tipo (1: grávida, 2: idoso, 3: PCD, 4: pessoa comum) e frustração.
        ii. Inclui um ponteiro para a próxima pessoa na fila.
    b. **FilaCircular:**
        i. Representa uma fila circular para gerenciar as pessoas.

    ii. Mantém ponteiros para o início (frente) e o final (tras) da fila, além de controlar o tamanho.

2. **Funções Relacionadas à Fila:**
    a. **criarFila()**: Inicializa uma fila vazia.
    b. **estaVazia(fila):** Verifica se a fila está vazia.
    c. **filaCheia(fila)**: Verifica se a fila está cheia.
    d. **tamanhoFila(fila)**: Retorna ao tamanho atual da fila.
    e. **estaNaFila(fila, pessoa)**: Verifica se uma pessoa está na fila.
    f. **enfileira(fila, pessoa)**: Adiciona uma pessoa à fila, considerando prioridades, como também é realizado o controle da frustração.
    g. **primeira_da_fila(fila)**: Retorna a pessoa no início da fila.
    h. **desenfileira(fila)**: Remove e retorna a pessoa no início da fila.
    i. **printFila(fila)**: Exibe o conteúdo da fila no console.
    j. **destruirFila(fila)**: Libera a memória alocada para a fila.
3. **Main Function:**
    a. Inicialização de variáveis, incluindo mutexes.
    b. Geração de um conjunto de pessoas com diferentes características.
    c. Embaralhamento aleatório da ordem das pessoas que não são gerentes.
    d. Criação de threads para a gerente e outras pessoas.
    e. Utilização de mutexes para garantir acesso seguro à fila e recursos compartilhados.
    f. Espera pela conclusão de todas as threads.
4. **Lógica de Atendimento:**
    a. Utilização de mutexes para controlar o acesso à fila e às operações relacionadas.
    b. A gerente atende as pessoas com base na prioridade e outras regras de atendimento.
    c. As outras pessoas solicitam atendimento, entram na fila e aguardam.
    d. A inibição ocorre na função de enfileirar onde caso alguém passe na frente de uma pessoas 2 vezes a prioridade dela é aumentada e quando alguém sai da fila a prioridade da pessoa é resetada.
5. **Considerações Adicionais:**
    a. Uso de sleep para simular o tempo de atendimento, liberação, etc.
    b. Atualização dinâmica das prioridades e frustrações das pessoas na fila.

Decisões Relevantes

- **Utilização de Fila Circular:**
    - Decidiu-se implementar uma fila circular para gerenciar a fila de atendimento. Essa escolha foi motivada pela eficiência na manipulação de filas, permitindo adição e remoção de elementos de forma rápida e sem a necessidade de realocação de memória.
- **Uso de pthread_mutex_t cliente[8];**:
    - Optou-se por atribuir um mutex para cada cliente, representados por **pthread_mutex_t cliente[8];**. Cada mutex é associado a um cliente específico, permitindo um controle mais granular do acesso ao caixa. Isso evita que clientes compartilhem o mesmo mutex e, consequentemente, reduz conflitos de acesso ao recurso compartilhado.
- **Ausência de pthread_cond_wait e pthread_cond_signal:**
    - A escolha de não utilizar **pthread_cond_wait** e **pthread_cond_signal** foi tomada devido a problemas identificados durante o desenvolvimento. A implementação inicial com essas funções resultou em complicações relacionadas à sincronização e, portanto, optou-se por soluções baseadas em mutexes para garantir a exclusão mútua e evitar condições de corrida.

Bugs Conhecidos ou Problemas

Durante a criação do sistema, enfrentamos desafios significativos ao tentar implementar **pthread_cond_wait** e **pthread_cond_signal**. Essas funções, originalmente projetadas para lidar com condições de espera e sinais entre threads, revelaram-se problemáticas para a lógica específica do nosso projeto, resultando em: Complexidade Adicional e Condições de Corrida e Sincronização Incorreta.

Conclusão

Diante dessas complicações, decidimos abandonar o uso dessas funções em favor de uma abordagem mais direta, utilizando mutexes. Essa mudança resultou em uma melhoria notável na estabilidade e eficiência do sistema. A exclusão de **pthread_cond_wait** e **pthread_cond_signal** simplificou o código, tornando-o mais compreensível e permitindo uma sincronização mais eficaz entre as threads.

Essa alteração representou um marco significativo no desenvolvimento, resultando em um código mais robusto e funcional. A experiência ressaltou a importância de

escolher abordagens que se alinhem de forma mais eficaz com as necessidades específicas do projeto, evitando complicações desnecessárias e garantindo um desenvolvimento mais suave.