CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

Unidade Leopoldina

Engenharia da Computação

Problema de sincronização usando threads

Parte 2

LABORATÓRIO DE SISTEMAS DE TEMPO REAL

GABRIEL RIBEIRO PASSOS

IURI SOUSA WERNECK PEREIRA

VICTOR DE SOUZA VILELA DA SILVA

Leopoldina, MG, Brasil

dezembro de 2023

Resumo do Projeto

Este projeto visa desenvolver um sistema para a gestão de atendimento em uma casa lotérica em Leopoldinópolis. Utilizando threads em linguagem C, o sistema prioriza a sincronização e evita problemas como inanição e deadlock. O código incorpora estruturas de dados eficientes e estratégias para garantir uma experiência de atendimento otimizada.

Estrutura Geral do Código

O código é dividido em duas partes principais: a definição das estruturas de dados e funções relacionadas à fila, e a implementação das threads para atendimento e solicitação de atendimento.

Estruturas de Dados e Funções da Fila

- **Pessoa:** Armazena informações sobre cada cliente, incluindo nome, prioridade, quantidade de uso do caixa, índice, tipo e nível de frustração. Contém um ponteiro para a próxima pessoa na fila.
- **FilaCircular**: Representa uma fila circular para gerenciar clientes, com ponteiros para o início e o final da fila.

- **Funções Relacionadas à Fila:**
    - **criarFila():** Inicializa uma fila vazia.
    - **estaVazia(fila):** Verifica se a fila está vazia.
    - **filaCheia(fila)**: Verifica se a fila está cheia.
    - **tamanhoFila(fila):** Retorna o tamanho atual da fila.
    - **estaNaFila(fila, pessoa):** Verifica se uma pessoa está na fila.
    - **retornaTipoPessoa(fila, vet):** Retorna os tipos de pessoas presentes na fila.
    - **enfileira(fila, pessoa):** Adiciona uma pessoa à fila, considerando prioridades e controle de frustração.
    - **primeira_da_fila(fila):** Retorna a pessoa no início da fila.
    - **desenfileira(fila, prioridade):** Remove e retorna uma pessoa da fila, considerando a prioridade ou resolvendo deadlock.
    - **printFila(fila):** Exibe o conteúdo da fila no console.
    - **destruirFila(fila):** Libera a memória alocada para a fila.

Implementação das Threads

- O código principal (main) inicia mutexes, cria um conjunto de pessoas com características diferentes e embaralha aleatoriamente a ordem das não-gerentes.

- São criadas threads para a gerente e outras pessoas, utilizando mutexes para garantir acesso seguro à fila e recursos compartilhados.

- A lógica de atendimento é implementada, considerando regras de prioridade, deadlock e estratégias para evitar inanição.

Deadlock

O código enfrenta a possibilidade de deadlock quando a fila contém clientes de todos os tipos (grávidas, idosos, pessoas com deficiência e pessoas comuns). Para contornar esse cenário crítico, foi implementada uma estratégia que envolve o gerente atendendo o primeiro cliente da fila, independentemente do tipo. Essa abordagem específica visa evitar que o atendimento fique bloqueado indefinidamente, garantindo a continuidade das operações mesmo sob condições adversas na composição da fila.

Decisões Relevantes

1. Uso de Mutexes por Cliente: A escolha de atribuir um mutex para cada cliente permite um controle mais granular do acesso ao caixa, evitando conflitos e garantindo sincronização adequada.

2. Exclusão de Condicionais: A decisão de não utilizar **pthread_cond_wait** e **pthread_cond_signal** foi tomada devido a complicações identificadas durante o desenvolvimento, optando por soluções baseadas em mutexes para garantir a exclusão mútua e evitar condições de corrida.

Bugs Conhecidos ou Problemas

Durante o desenvolvimento, foram enfrentados desafios ao implementar **pthread_cond_wait** e **pthread_cond_signal**, levando a complicações e condições de corrida. Optou-se por soluções baseadas em mutexes, resultando em um código mais estável e eficiente.

Conclusão

O enfoque em deadlock no código destaca a importância de estratégias proativas para evitar situações críticas que poderiam paralisar o atendimento na casa lotérica. A implementação específica, onde a gerente atende o primeiro cliente em deadlock, representa uma solução prática e eficiente para garantir a continuidade das operações. O código resultante é robusto, funcional e demonstra a capacidade de enfrentar desafios complexos na sincronização de threads.