Escola Politécnica

NÚCLEO DE ELETIVAS INTERESCOLAS DA ESCOLA POLITÉCNICA

Disciplina Engenharia de Software

Profa. Adriana Gomes Alves, Claudia Neli de Souza Zambon

SISTEMA DE PROCESSAMENTO PARALELO DE REQUISIÇÕES A UM BANCO DE DADOS EM C

Acadêmicos:

Eduardo Sartori
Gabriel Passos Francisco
Gabriel Pavan Marquevicz

Introdução

O presente trabalho tem como proposta a implementação de um sistema que simula o comportamento interno de um gerenciador de banco de dados, utilizando a linguagem de programação C. O sistema contempla dois processos distintos que são cliente e o servidor que se comunicam por meio de um pipe nomeado (named pipe). O processo cliente é responsável por enviar comandos de requisições ao banco de dados, enquanto o servidor é responsável por interpretar e processar essas requisições, utilizando *threads* para garantir execução paralela e eficiente.

1. Enunciado do projeto

Este trabalho tem o objetivo de desenvolver um projeto de realização de um simulador de um gerenciador de requisições a um banco de dados, usando a linguagem de programação C, baseado nos conceitos de processamento paralelo, comunicação entre processos e controle de concorrência utilizando threads. A proposta é considerar dois processos principais: um cliente, que emite requisições em comandos SQL simplificados (INSERT, DELETE, SELECT, UPDATE) a ser enviada a um servidor, ou seja, quem as receberá.

Além disso, o sistema é projetado para ser manipulado bem acima de um número limitado de solicitações simultâneas e um único servidor usando apenas uma thread de serviço não conseguisse interpretar e processar essas solicitações. Para resolver esse problema, o servidor possui um pool de threads, o que possibilita ao sistema interpretar e processar as solicitações do cliente de maneira simultânea. As threads acessam a estrutura de dados compartilhada simulando um banco de dados, que requer mecanismos de exclusão mútua, como pthread_mutex_t, para manter a integridade dos dados durante operações concorrentes. O sistema não apenas trabalha com os dados em memória, mas também garante que o arquivo externo banco.json salve os dados. Isso torna possível simular uma base de dados com registros por um id e um nome.

2. Explicação e contexto da aplicação para compreensão do problema tratado pela solução.

O objetivo deste projeto é simular uma solução desenvolvendo um sistema simplificado capaz de imitar um gerenciador de banco de dados por meio de dois processos diferentes: um cliente e um servidor. Enquanto o primeiro é encarregado de enviar requisições – que são enviados em formato textual, como INSERT, DELETE, SELECT e UPDATE – o segundo recebe as requisições e as processa simultaneamente utilizando-se de threads para garantir a execução segura das operações.

A fim de simular um banco de dados real, foi utilizado uma estrutura de vetor em memória representando registros com campos de id e nome, bem como persistência em arquivo. Controle de acesso concorrente a essa estrutura é realizado com uso de mutex – exclusão mútua, impedindo conflitos durante modificações simultâneas.

3. Códigos importantes da implementação.

3.1 Comunicação entre Processos (IPC) via Pipe Nomeado

O sistema usa um pipe nomeado, denominado pipe_bd, como meio de comunicação entre o processo cliente e o processo servidor. Esse é um mecanismo de IPC que permite ao cliente enviar em tempo real comandos que são recebidos mais tarde para o servidor. A criação do pipe é realizada com a função mkfifo, enquanto as operações de leitura e escrita são realizadas com as funções open, fopen e fgets.

```
#define PIPE_NAME "pipe_bd"
```

```
// Criar pipe se não existir
if (access(PIPE_NAME, F_OK) == -1)
{
    if (mkfifo(PIPE_NAME, 0666) != 0)
    {
        perror("Erro ao criar pipe");
        return 1;
    }
}
```

3.2 Processamento Paralelo com Threads

Toda requisição enviada ao servidor é processada por uma thread distinta, realizada de forma dinâmica pelo pthread_create. Dessa forma, o sistema opera sobre várias requisições ao mesmo tempo, explorando a concorrência e emulando o comportamento do SQL do banco de dados real. Adicionalmente, todas as threads são finalizadas através do pthread_detach, provendo que o memory management seja feito automaticamente após o final do processamento de cada requisição.

```
while (fgets(buffer, sizeof(buffer), pipe))
{
    pthread_t thread;
    char *requisicao = strdup(buffer);
    pthread_create(&thread, NULL, processar_requisicao, requisicao);
    pthread_detach(thread);
}
```

3.3 Controle de Concorrência com Mutex

A fim de evitar problemas de concorrência, como condições de corrida, foi utilizado um mecanismo de exclusão mútua pthread_mutex_t. Todas as operações que operam na estrutura de dados compartilhada, isto é, inserção, exclusão ou atualização de registros, foram protegidas pthread_mutex_lock e pthread_mutex_unlock.

```c
pthread_mutex_lock(&mutex_banco);

if (strcmp(operacao, "INSERT") == 0)
{
    inserir_registro(id, nome);
    printf("Registro inserido: %d - %s\n", id, nome);
}
else if (strcmp(operacao, "DELETE") == 0)
{
    deletar_registro(id);
    printf("Registro deletado com id: %d\n", id);
}
else if (strcmp(operacao, "SELECT") == 0)
{
    Registro *r = buscar_registro(id);
    if (r)
    {
```

4. Resultados obtidos com a implementação

Após a conclusão da implementação do sistema de gerenciamento de banco de dados com suporte a múltiplos processos e threads, foram executados testes práticos a fim de verificar o seu funcionamento e aprovar suas funcionalidades principais. Como resultado, várias operações de inserção, consulta, atualização e exclusão de registros foram simuladas por meio do processo cliente e processadas pelo servidor.

```
pipe_bd
1   INSERT 1 JOAO
2   INSERT 2 ANA
3   INSERT 3 MARIA
4   SELECT 1
5   SELECT 2
6   SELECT 3
7   UPDATE 2 ANA CLARA
8   DELETE 1
```

```
Registro inserido: 1 - JOAO
Registro inserido: 2 - ANA
Registro inserido: 3 - MARIA
Registro encontrado: 1 - JOAO
Registro encontrado: 2 - ANA
Registro encontrado: 3 - MARIA
Registro atualizado: 2 - ANA CLARA
Registro deletado com id: 1
```

Ao final das operações, o conteúdo persistido no arquivo banco.json foi atualizado corretamente, refletindo o estado final da base de dados:

```
[
    {"id":3,"nome":"MARIA"},
    {"id":2,"nome":"ANA CLARA"}
]
```

Esses resultados demonstram que o sistema se comportou conforme o esperado, garantindo a integridade dos dados mesmo com a execução paralela das requisições. O uso de *threads* permitiu que o servidor respondesse rapidamente às requisições, enquanto o mecanismo de mutex garantiu que não ocorressem acessos simultâneos não controlados à estrutura de dados compartilhada.

5. Análise e discussão sobre os resultados finais.

Com a implementação do sistema proposto, foi possível nutrir na prática diversos conceitos teóricos sobre comunicação entre processos, programação concorrente e controle de acesso a recursos compartilhados. Como pôde ser observado a partir dos testes, o sistema é de fato funcional e capaz de reproduzir com exatidão um cenário onde múltiplas requisições são subtraídas a um processamento por um gerenciador de banco de dados simplificado.

E durante a execução, foi levado ao conhecimento que, de fato, cada comando emitido pelo cliente, INSERT, DELETE, SELECT e UPDATE e, posteriormente, era interpretado pelo servidor, e as operações sobre o vetor de registros, bem como o arquivo banco.json, ocorreram de maneira sincronizada. Utilizando o pthread_mutex_t, portanto, pudemos proteger os acessos de leitura e gravação simultâneos, bem como a adição e exclusão de um item do vetor ao mesmo tempo, o que assegurava que a consistência dos dados fosse preservada independentemente.