Universidade do Vale do Itajaí
Centro: Escola Politécnica
Curso: Ciência da Computação, Engenharia de Computação
Disciplina: Sistemas Operacionais

Implementação de Gerenciador de Requisições a Banco de Dados com IPC e Threads

Nome: Pedro Henrique Kons, Guilherme Thomy

14/04/2025

Resumo:

Este trabalho descreve o desenvolvimento de um sistema cliente-servidor para simular o processamento de requisições a um banco de dados, utilizando conceitos de **Sistemas Operacionais**. O objetivo foi consolidar o aprendizado em comunicação entre processos (**IPC**), concorrência e paralelismo com **threads.** Foi implementado em linguagem C, utilizando **memória compartilhada** como mecanismo de IPC e um pool de threads no servidor para processar requisições de inserção, deleção e listagem em paralelo. O acesso concorrente a um arquivo binário, simulando o banco de dados, foi sincronizado utilizando mutexes para garantir a integridade dos dados.

Introdução

Um gerenciamento eficiente de requisições em sistemas de banco de dados é crucial para garantir alta performance, evitar sobrecarga, reduzir tempos de resposta e assegurar a integridade e consistência dos dados, especialmente em ambientes com múltiplos acessos simultâneos.

A **concorrência** permite que múltiplas requisições sejam tratadas de forma intercalada, otimizando o uso dos recursos mesmo com um único processador. á o **paralelismo** divide tarefas entre múltiplos núcleos ou máquinas, executando-as simultaneamente e acelerando o processamento. Ambas aumentam a eficiência, especialmente em cenários com grande volume de dados e acessos simultâneos.

Para garantir a troca de informações entre processos distintos, como cliente e servidor, é essencial o uso de mecanismos de **IPC (Inter-Process Communication)**. Eles permitem a comunicação e sincronização entre processos independentes. Já dentro do servidor, o uso de **threads** permite a execução paralela de múltiplas tarefas no mesmo processo, aumentando a eficiência e a capacidade de atender várias requisições simultaneamente.

A **sincronização** é fundamental para evitar **condições de corrida** quando múltiplas threads acessam recursos compartilhados, como uma fila de tarefas ou um banco de dados simulado. Mecanismos como **mutexes** e **semáforos** garantem que apenas uma thread acesse esses recursos por vez, preservando a integridade dos dados e evitando comportamentos imprevisíveis no sistema.

Objetivo do Trabalho:

Desenvolver um sistema em linguagem C composto por um cliente e um servidor para simular um gerenciador de requisições a banco de dados. O sistema utiliza **memória compartilhada** como mecanismo de **IPC**, um **pool de threads Pthreads** no servidor para **processamento paralelo**, e **mutexes** para sincronizar o acesso a uma **fila de tarefas** e a um **arquivo que simula o banco de dados**, suportando operações de **INSERT**, **DELETE** e, como funcionalidade extra, **LIST**.

Desenvolvimento:

Arquitetura do Sistema:
O sistema adota um modelo **cliente-servidor**, onde o **cliente** é responsável por enviar **requisições** de operações (INSERT, DELETE, LIST) por meio de **memória compartilhada (IPC)**. O **servidor** recebe essas requisições, as insere em uma **fila de tarefas compartilhada** e as processa utilizando um **pool de threads**. Cada thread executa as operações solicitadas e atualiza, de forma sincronizada, o **estado do banco de dados simulado** em um arquivo, garantindo integridade e consistência das informações.

Comunicação Entre Processos (IPC):
Para viabilizar a comunicação entre o cliente e o servidor, foi escolhida a **Memória Compartilhada** utilizando as chamadas do sistema `shmget` e `shmat`. Essa abordagem permite que ambos os processos acessem **diretamente a mesma região de memória**, de forma eficiente e rápida, sem a necessidade de troca de mensagens ou arquivos intermediários.

A estrutura utilizada para essa comunicação é a `SharedMemory`, definida no arquivo `db.h`, composta por dois campos principais:

- `char message[256]`: onde o cliente escreve a requisição (ex: INSERT id=”22” nome=”pedro”).
- `int ready`: um flag de controle que indica o estado da memória compartilhada.

O protocolo de sincronização adotado é simples:

1. O **cliente** espera `ready == 0`, escreve a mensagem e então seta `ready = 1`.
2. O **servidor** espera `ready == 1`, lê a mensagem e então seta `ready = 0`.

Esse protocolo garante que não ocorra escrita e leitura simultânea na memória compartilhada.

Gerenciamento de Concorrência no Servidor:

O servidor utiliza um **pool fixo de threads**, definido por `THREAD_POOL_SIZE = 4`, criado com a função `pthread_create`. Essas threads trabalham em paralelo para processar requisições recebidas do cliente, otimizando o desempenho e a capacidade de resposta do sistema.

O sistema utiliza uma estrutura chamada **TaskQueue**, que atua como um **buffer intermediário** entre a thread principal do servidor (responsável por ler as requisições da memória compartilhada) e as **threads trabalhadoras** (que executam as operações solicitadas). Esse modelo segue o padrão clássico de **produtor-consumidor**:

- **Produtor**: a thread principal do servidor, que insere as tarefas na fila por meio da função `enqueue()`.

- **Consumidores**: as threads do pool, que retiram as tarefas da fila com a função `dequeue()` e as processam.

Para evitar condições de corrida e garantir a integridade da fila, a sincronização é feita com:

- **`queue_mutex`**: um **mutex** que garante **acesso exclusivo** à estrutura da fila durante as operações de inserção e remoção.

- **`queue_cond`**: uma **variável de condição** usada pelas threads consumidoras com `pthread_cond_wait()` para **esperar enquanto a fila está vazia**. Quando a thread principal adiciona uma nova tarefa, ela chama `pthread_cond_signal()` para **acordar uma thread consumidora**, permitindo o processamento contínuo das requisições.

Implementação:

Configuração da Memória Compartilhada

```c
// server.c - Criação e mapeamento no servidor
int shmid = shmget(SHM_KEY, sizeof(SharedMemory), IPC_CREAT | 0666);
if (shmid == -1) { /* Tratamento de erro */ }
shared_data = (SharedMemory*) shmat(shmid, NULL, 0);
if (shared_data == (void*) -1) { /* Tratamento de erro */ }
shared_data->ready = 0; // Sinaliza que servidor está pronto

// client.c - Acesso e mapeamento no cliente
int shmid = shmget(SHM_KEY, sizeof(SharedMemory), 0666);
if (shmid == -1) {
    perror("Erro ao acessar memória compartilhada");
    return 1;
}
SharedMemory *shared_data = (SharedMemory*) shmat(shmid, NULL, 0);
if (shared_data == (void*) -1) { /* Tratamento de erro */ }
```

O servidor cria o segmento com `IPC_CREAT`, enquanto o cliente apenas o acessa. Ambos mapeiam o segmento para seus espaços de endereço usando `shmat`.

Interação Cliente-Servidor via Memória Compartilhada

```c
// client.c - Loop de envio
while (1) {
    // ... (lê input do usuário) ...

    // Espera o servidor estar pronto (ready == 0) - Busy Waiting
    while (shared_data->ready == 1);

    // Copia comando para memória compartilhada e sinaliza
    strcpy(shared_data->message, input);
    shared_data->ready = 1;

    if (strcmp(input, "EXIT") == 0) break;
}

// server.c - Loop de recebimento na thread principal
while (1) {
    // Verifica se há nova mensagem do cliente
    if (shared_data->ready == 1) {
        if (strcmp(shared_data->message, "EXIT") == 0)
            break; // Sai do loop se for EXIT

        // Copia comando da memória compartilhada para uma Task local
        Task t;
        strcpy(t.command, shared_data->message);

        // Adiciona a task à fila para as threads processarem
        enqueue(t);

        // Sinaliza de volta ao cliente que a mensagem foi recebida
        shared_data->ready = 0;
    }
    usleep(10000); // Pequena pausa para não consumir 100% CPU
}
```

Ilustra o protocolo de handshake usando o flag `ready` para sincronizar a troca de mensagens via memória compartilhada.

Fila de Tarefas e Sincronização (Servidor)

```c
// server.c - Adiciona tarefa à fila (Produtor: thread principal)
void enqueue(Task t) {
    pthread_mutex_lock(&queue_mutex); // Adquire lock da fila
    // Espera se a fila estiver cheia (não implementado neste código, mas seria
    if (task_queue.count < QUEUE_SIZE) {
        task_queue.queue[task_queue.rear] = t;
        task_queue.rear = (task_queue.rear + 1) % QUEUE_SIZE;
        task_queue.count++;
        pthread_cond_signal(&queue_cond); // Acorda um consumidor esperando
    } else {
        // Fila cheia - poderia esperar ou descartar
    }
    pthread_mutex_unlock(&queue_mutex); // Libera lock da fila
}

// server.c - Remove tarefa da fila (Consumidor: threads trabalhadoras)
Task dequeue() {
    pthread_mutex_lock(&queue_mutex); // Adquire lock da fila
    // Espera se a fila estiver vazia
    while (task_queue.count == 0) {
        pthread_cond_wait(&queue_cond, &queue_mutex); // Espera sinal e libera mu
    }
    // Remove tarefa
    Task t = task_queue.queue[task_queue.front];
    task_queue.front = (task_queue.front + 1) % QUEUE_SIZE;
    task_queue.count--;
    pthread_mutex_unlock(&queue_mutex); // Libera lock da fila
    return t;
}
```

Mostra o uso de `pthread_mutex_lock/unlock` para acesso exclusivo e `pthread_cond_signal/wait` para coordenação entre produtor e consumidores na fila de tarefas.

Processamento de Comandos pelas Threads

```c
// server.c - Função executada por cada thread trabalhadora
void* thread_worker(void *arg) {
    while (1) { // Loop infinito da thread (idealmente teria condição de saída)
        Task task = dequeue(); // Pega tarefa da fila (pode bloquear)
        printf("[Thread %ld] Processando: %s\n", pthread_self(), task.command); //

        // Processa comando INSERT
        if (strncmp(task.command, "INSERT", 6) == 0) {
            Record r;
            // Parseia comando (sem tratamento de erro robusto)
            sscanf(task.command, "INSERT id=%d name='%[^']'", &r.id, r.name);
            insert_record(r); // Chama função que acessa o arquivo
        }
        // Processa comando DELETE
        else if (strncmp(task.command, "DELETE", 6) == 0) {
            int id;
            sscanf(task.command, "DELETE id=%d", &id);
            delete_record(id); // Chama função que acessa o arquivo
        }
        // Processa comando LIST
        else if (strncmp(task.command, "LIST", 4) == 0) {
            list_records(); // Chama função que acessa o arquivo
        }
        // Outros comandos poderiam ser adicionados aqui
    }
    return NULL;
}
```

A thread trabalhadora obtém uma tarefa via `dequeue` e, com base no comando, chama a função apropriada para interagir com o banco de dados simulado. O `printf` ajuda a visualizar qual thread está processando qual comando.

Sincronização de Acesso ao Arquivo de Banco de Dados

```c
// server.c - Exemplo com insert_record
void insert_record(Record r) {
    pthread_mutex_lock(&file_mutex); // Adquire lock ANTES de acessar o arquivo
    FILE *fp = fopen(DB_FILE, "ab"); // Abre arquivo (idealmente verificar se fp !
    if (fp) {
        fwrite(&r, sizeof(Record), 1, fp); // Escreve o registro
        fclose(fp);                        // Fecha o arquivo
    } else {
        perror("Erro ao abrir db.txt para escrita");
    }
    pthread_mutex_unlock(&file_mutex); // Libera lock DEPOIS de fechar o arquivo
}

// server.c - Exemplo com delete_record (início e fim)
void delete_record(int id) {
    pthread_mutex_lock(&file_mutex); // Adquire lock
    FILE *fp = fopen(DB_FILE, "rb");
    FILE *tmp = fopen("tmp.txt", "wb");
    // ... (lógica de leitura/escrita entre fp e tmp) ...
    fclose(fp);
    fclose(tmp);
    remove(DB_FILE);
    rename("tmp.txt", DB_FILE);
    pthread_mutex_unlock(&file_mutex); // Libera lock
}
```

Demonstra o uso crítico do `file_mutex` para garantir que as operações no arquivo `db.txt` (e `tmp.txt` no caso do delete) sejam realizadas de forma atômica em relação a outras threads, prevenindo corrupção de dados.

Exemplo de execução:

```
bd_simulator on ◿ main [!?] via C v13.3.0-gcc took 2m25s
❯ ./client
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT): INSERT id=1 name='pedro'
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT): INSERT id=13 name='guilherme'
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT): LIST
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT): DELETE id=1
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT): LIST
Digite um comando (INSERT id=1 name='Joao', LIST, DELETE id=1, EXIT):
```

```
bd_simulator on ◿ main [!?] via C v13.3.0-gcc took 2m22s
❯ ./server
[Servidor] Aguardando comandos do cliente...
[Thread 139088181065408] Processando: INSERT id=1 name='pedro'
[Thread 139088172672704] Processando: INSERT id=13 name='guilherme'
[Thread 139088164280000] Processando: LIST
ID: 1, Name: pedro
ID: 13, Name: guilherme
[Thread 139088155887296] Processando: DELETE id=1
[Thread 139088181065408] Processando: LIST
ID: 13, Name: guilherme
```

Observações da Execução:

- O cliente enviou os comandos com sucesso via memória compartilhada.
- O servidor recebeu os comandos e os enfileirou para as threads trabalhadoras.
- As mensagens "[Thread ...]" no servidor indicam que diferentes threads (identificadas por seus IDs) processaram as requisições, demonstrando o paralelismo.
- As operações `INSERT`, `LIST` e `DELETE` produziram os resultados esperados no arquivo `db.txt` e na saída do comando `LIST`.

Análise e Discussão:

Os resultados obtidos na execução demonstram que o sistema implementado atende aos requisitos funcionais básicos do projeto. A arquitetura cliente-servidor, a comunicação via memória compartilhada e o processamento paralelo com threads foram implementados com sucesso.

- **Paralelismo e Desempenho:** O uso de um pool de threads no servidor permite que múltiplas requisições sejam tratadas concorrentemente. As mensagens de log indicam que diferentes threads assumiram o processamento de comandos distintos. Embora não tenham sido realizadas medições de desempenho formais, a arquitetura tem o *potencial* de oferecer melhor throughput (vazão) e responsividade em comparação com um servidor single-threaded, especialmente sob carga elevada ou se as operações de processamento fossem mais intensivas em CPU. No entanto, é crucial notar que o acesso ao arquivo `db.txt` é serializado pelo `file_mutex`. Isso significa que, mesmo com várias threads, apenas uma pode estar efetivamente lendo ou escrevendo no arquivo por vez, tornando o I/O do disco um gargalo potencial que limita o ganho real de paralelismo para *estas operações específicas*.
- **Fila de Tarefas:** O uso de `queue_mutex` e `queue_cond` garantiu que a adição (`enqueue`) e remoção (`dequeue`) de tarefas na `TaskQueue` ocorressem de forma segura, evitando race conditions que poderiam levar à perda de tarefas ou corrupção dos índices da fila. O `pthread_cond_wait` permitiu que as threads esperassem eficientemente por novas tarefas.
- O `file_mutex` foi essencial para proteger a integridade do arquivo `db.txt`. Ao garantir exclusão mútua para todas as operações de arquivo (`insert`, `delete`, `list`), preveniu-se que múltiplas threads tentassem modificar ou ler o arquivo simultaneamente de maneira inconsistente.
- **Comunicação Interprocessos (IPC):** A memória compartilhada provou ser um mecanismo eficaz para a troca de mensagens entre cliente e servidor. A simplicidade da estrutura `SharedMemory` com o flag `ready` funcionou para a sincronização básica.

Conclusão

Este trabalho realizou com sucesso a implementação de um sistema cliente-servidor em linguagem C para simular um gerenciador de requisições a banco de dados, aplicando conceitos essenciais de Sistemas Operacionais. Foram utilizados memória compartilhada para IPC, um pool de threads Pthreads para processamento paralelo no servidor, e mutexes para garantir a sincronização segura do acesso a uma fila de tarefas e a um arquivo simulando o banco de dados.

O sistema demonstrou ser capaz de receber comandos do cliente (`INSERT`, `DELETE`, `LIST`), processá-los concorrentemente utilizando múltiplas threads e manter a integridade do banco.