



Universidade do Vale do Itajaí - UNIVALI
Ciências da Computação
Disciplina de Sistemas Operacionais

SAMUEL CESÁRIO PEREIRA TRIZOTTO DE ANDRADE
DAVID SMITH SOUZA

SISTEMA MULTITHREAD PARA GERENCIAMENTO CONCORRENTE DE
REQUISIÇÕES A BANCO DE DADOS SIMULADO VIA IPCS

ITAJAÍ
2025

RESUMO

Este relatório descreve o desenvolvimento de um sistema simples de gerenciamento de banco de dados, com foco em comunicação entre processos e concorrência com threads. O sistema é composto por dois programas: um cliente que envia comandos via *pipe* e um servidor que recebe essas requisições e as processa em paralelo usando *threads* no Windows. Para garantir que os dados do banco (armazenados em vetor e salvos em um arquivo .txt) não sejam corrompidos por acessos simultâneos, foi utilizada a estrutura `CRITICAL_SECTION` para controle de exclusão mútua. Foram implementadas as operações `INSERT`, `DELETE`, `SELECT` e `UPDATE`, permitindo testar o funcionamento do sistema com múltiplas requisições simultâneas. O projeto cumpre os requisitos propostos no enunciado e demonstrou bons resultados nas simulações realizadas.

Palavras-chave: IPC, pipe, threads, exclusão mútua, banco de dados.

1 INTRODUÇÃO

Com o crescimento da complexidade em sistemas computacionais, o uso de comunicação entre processos (IPC) e controle de concorrência tornou-se essencial para garantir eficiência e integridade em aplicações que lidam com dados compartilhados. Neste contexto, este projeto propõe a implementação de um sistema simples que simula o funcionamento interno de um gerenciador de banco de dados, utilizando conceitos fundamentais da programação concorrente.

O sistema é dividido em dois programas distintos: um cliente, responsável por enviar comandos ao servidor, e um servidor, que processa essas requisições em paralelo por meio de múltiplas *threads*. A comunicação entre o cliente e o servidor é feita por *pipe* anônimo.

O banco de dados é representado por um vetor de registros e armazenado em um arquivo .txt. Para evitar problemas de concorrência no acesso ao vetor e ao arquivo, o projeto utiliza CRITICAL_SECTION da API do Windows para controle de exclusão mútua.

O sistema implementa os comandos INSERT, DELETE, SELECT e UPDATE, conforme solicitado no enunciado. O relatório apresenta o funcionamento da aplicação, os testes realizados e uma breve análise dos resultados.

2 ENUNCIADO DO PROJETO

O presente projeto tem como objetivo o desenvolvimento de um sistema que simula o funcionamento interno de um gerenciador de requisições a um banco de dados, utilizando processos distintos e comunicação via *pipe* anônimo. O sistema deve conter:

- Um processo **cliente**, responsável por enviar requisições de consulta e modificação ao banco de dados;
- Um processo **servidor**, que recebe as requisições, cria múltiplas *threads* para processá-las em paralelo e acessa uma estrutura compartilhada (vetor de registros) com controle de concorrência.

A comunicação entre os processos deve ser realizada por meio de *pipe*

ou memória compartilhada, e as operações suportadas incluem INSERT, DELETE, SELECT e UPDATE. A integridade da estrutura de dados deve ser garantida por meio de exclusão mútua, utilizando mecanismos como mutex ou semáforo.

O banco de dados simulado é composto por registros com identificador numérico e nome associado, podendo ser salvo em arquivo texto ou outra forma de armazenamento simples. O servidor deve registrar ou exibir os resultados das requisições.

Este projeto envolve a aplicação prática dos seguintes conceitos:

- Comunicação entre processos (*Interprocess Communication – IPC*);
- Criação e gerenciamento de *threads*;
- Controle de concorrência com exclusão mútua;
- Simulação de um Sistema Gerenciador de Banco de Dados (SGBD) leve;
- Leitura e escrita concorrente em estruturas de dados e arquivos.

3 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta os principais conceitos e fundamentos teóricos aplicados no desenvolvimento do sistema, evidenciando os aspectos da comunicação entre processos, execução concorrente com *threads* e controle de acesso à estrutura de dados, que constituem a base do projeto.

3.1 Comunicação entre Processos (IPC)

A comunicação entre processos (IPC - *Interprocess Communication*) é um mecanismo que permite a troca de informações entre programas em execução, sendo essencial para a construção de aplicações distribuídas e multitarefa. No presente projeto, o mecanismo de **pipe anônimo** foi utilizado para estabelecer um canal unidirecional, em que um processo (cliente) escreve os comandos em mensagens e outro processo (servidor) os lê. Essa abordagem possibilita a transmissão eficiente de dados e o desacoplamento dos processos envolvidos, já que não utilizam memória compartilhada.

3.2 Processos e Threads

Um **processo** refere-se a uma instância de um programa em execução, com seu próprio espaço de endereçamento de memória. Em contrapartida, uma **thread** representa a menor unidade de execução dentro de um processo e compartilha o mesmo espaço de memória com outras threads do mesmo processo. No sistema desenvolvido, o servidor cria uma nova thread para cada requisição recebida, permitindo o processamento paralelo e aumentando a eficiência do atendimento às operações solicitadas.

3.3 Exclusão Mútua

O acesso concorrente a recursos compartilhados pode ocasionar condições de corrida, nas quais múltiplas threads modificam uma mesma estrutura de dados sem a devida sincronização, resultando em dados inconsistentes. Para evitar essa situação, o projeto utiliza mecanismos de exclusão mútua. No ambiente Windows, foi empregada a função **Critical Section**, que atua de forma similar a um *mutex*, permitindo que apenas uma thread acesse um trecho crítico do código por vez, assegurando a integridade tanto da memória interna (vetor de registros) quanto do arquivo que simula o banco de dados.

3.4 Estrutura de Dados

A estrutura de dados escolhida para a simulação do banco de dados foi um vetor de *structs* do tipo Registro, contendo um int ID e um campo char[50]. Esta definição possibilita o armazenamento sequencial dos dados e facilita a implementação das operações de inserção, remoção, consulta e atualização. O arquivo texto (banco.txt) utilizado como meio de persistência dos dados complementa a estratégia de armazenamento, garantindo que as informações possam ser recuperadas e atualizadas de forma consistente.

3.5 Operações do Banco de Dados

As operações implementadas correspondem a comandos básicos de manipulação de dados, simulando o funcionamento de um Sistema Gerenciador de Banco de Dados (SGBD) leve. As principais operações são:

- **INSERT:** Responsável por adicionar um novo registro ao vetor e atualizá-lo no arquivo de persistência.

- **DELETE:** Remove o registro correspondente ao identificador informado.
- **SELECT:** Realiza a consulta de um registro com base no seu identificador.
- **UPDATE:** Atualiza os dados de um registro específico, modificando o valor do campo nome.

Cada uma dessas operações é executada dentro de um contexto protegido pela *Critical Section*, de modo a evitar a competição entre as threads e garantir a coerência dos dados.

4 METODOLOGIAS

4.1 Linguagem

A linguagem escolhida para o projeto foi C puro tanto para o processo servidor quanto para o cliente. O motivo foi a coerência com a linguagem usada em sala de aula, a ampla aceitação e utilização das bibliotecas em C para gerenciamento de threads e processos e os propósitos gerais da linguagem C.

4.2 Bibliotecas

A principal biblioteca do projeto é a windows.h, uma vez que é ela que permite a manipulação de threads e processos no sistema Windows, substituindo a Pthreads do Linux, uma vez que os computadores da universidade rodam apenas Windows.

4.3 Arquitetura

O projeto é separado em quatro arquivos. “cliente.c” é a interface que roda no console, onde o usuário digita os comandos. “servidor.c” é o responsável pelo gerenciamento das threads, se comunicando com cliente.c via IPC pipe. “banco.h” é um arquivo com as definições da struct e “banco.txt” é onde o programa salva o banco de dados em disco.

4.4 Implementação

4.4.1 IPC pipe

A comunicação entre o cliente e o servidor é realizada através de um *pipe* anônimo. No código do cliente, utiliza-se a função `CreatePipe` para criar os lados de leitura e escrita. Em seguida, o cliente configura o processo filho (o servidor) para redirecionar a entrada padrão para o descritor de leitura do pipe. Exemplo de trecho de código no cliente:

```
HANDLE hReadPipe, hWritePipe;
SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };

// 1. Criar um pipe anônimo para comunicação
if (!CreatePipe(&hReadPipe, &hWritePipe, &sa, 0)) {
    fprintf(stderr, "[ERRO] Falha ao criar pipe\n");
    return 1;
}
```

Após a criação do *pipe*, o cliente configura a estrutura `STARTUPINFO` para que o servidor herde o handle correspondente à entrada padrão, permitindo que as requisições enviadas pelo cliente sejam lidas diretamente pelo servidor via `stdin`.

```
// 2. Configurar o processo filho (servidor)
STARTUPINFO si = { sizeof(STARTUPINFO) };
PROCESS_INFORMATION pi;

// Redirecionar a entrada padrão (STDIN) do filho para o pipe
si.hStdInput = hReadPipe;
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
si.hStdError = GetStdHandle(STD_ERROR_HANDLE);
si.dwFlags = STARTF_USESTDHANDLES;

// 3. Criar o processo filho (servidor)
if (!CreateProcess(
    NULL,                // Nome do executável (NULL = usar linha de comando)
    "servidor.exe",      // Comando (servidor.exe deve estar no mesmo diretório)
    NULL,                // Atributos de segurança do processo
    NULL,                // Atributos de segurança da thread
    TRUE,                // Herdar handles (incluindo o pipe)
    0,                  // Flags de criação (0 = padrão)
    NULL,                // Ambiente (herdar do pai)
    NULL,                // Diretório de trabalho (herdar do pai)
    &si,                 // STARTUPINFO
    &pi                  // PROCESS_INFORMATION
)) {
    fprintf(stderr, "[ERRO] Falha ao criar processo servidor\n");
    CloseHandle(hReadPipe);
    CloseHandle(hWritePipe);
    return 1;
}

// 4. Processo pai (cliente) envia comandos pelo pipe
CloseHandle(hReadPipe); // O pai não usa a extremidade de leitura
```

4.4.2 Criação e gerenciamento de threads

No processo servidor, a cada linha recebida (representando uma requisição), o programa cria uma nova thread usando a função `CreateThread`. Cada thread trata a requisição individualmente, permitindo a execução paralela das operações.

Um trecho representativo da criação de thread é mostrado a seguir:

```
InitializeCriticalSection(&cs); // Inicializa a seção crítica (equivalente ao mutex)
carregarBanco();

char linha[MAX_REQUISICAO];
while (fgets(linha, sizeof(linha), stdin)) {
    char* req = _strdup(linha); // strdup no Windows pode exigir _strdup
    HANDLE hThread = CreateThread(
        NULL,                // Atributos de segurança padrão
        0,                   // Tamanho padrão da pilha
        tratarRequisicao,     // Função da thread
        (LPVOID)req,         // Argumento (requisição)
        0,                   // Flags de criação (0 = execução imediata)
        NULL                  // ID da thread (não necessário)
    );

    if (hThread == NULL) {
        fprintf(stderr, "[ERRO] Falha ao criar thread\n");
        free(req);
    } else {
        CloseHandle(hThread); // Fecha o handle (a thread continua executando)
    }
}

DeleteCriticalSection(&cs); // Libera a seção crítica
```

Após o lançamento da thread, o handle é fechado com `CloseHandle(hThread)` para liberar recursos, mantendo a thread em execução.

4.4.3 Controle de concorrência com Critical Section

Como múltiplas threads podem tentar acessar e modificar a estrutura de dados compartilhada (um vetor de registros) e o arquivo que simula o banco de dados, é necessário garantir a exclusão mútua. Para isso, o servidor utiliza a função `InitializeCriticalSection` para criar uma *Critical Section* que sincroniza o acesso aos recursos críticos, evitando condições de corrida. Cada operação que modifica o banco (como inserções, deleções, atualizações ou consultas) é encapsulada entre chamadas às funções `EnterCriticalSection` e `LeaveCriticalSection`.

Um exemplo dessa implementação é a criação da critical section no main e a entrada e saída em funções como salvarBanco();

```
int main() {
    InitializeCriticalSection(&cs); // Inicializa a seção crítica (equivalente ao mutex)
    carregarBanco();

    char linha[MAX_REQUISICAO];
    while (fgets(linha, sizeof(linha), stdin)) { ...

    DeleteCriticalSection(&cs); // Libera a seção crítica
    return 0;
}
```

```
void salvarBanco() {
    EnterCriticalSection(&cs);
    FILE* f = fopen("./banco.txt", "w");
    if (!f) {
        printf("[ERRO] Falha ao abrir arquivo para escrita\n");
        LeaveCriticalSection(&cs);
        return;
    }
    for (int i = 0; i < total; i++) {
        fprintf(f, "%d %s\n", banco[i].id, banco[i].nome);
    }
    fclose(f);
    LeaveCriticalSection(&cs);
}
```

4.4.4 Acesso e persistência de dados

O banco de dados simulado é representado por um vetor de estruturas definidas em "banco.h", onde cada registro contém um identificador e um nome. Para persistir os dados, o sistema salva o conteúdo do vetor em um arquivo texto (banco.txt). Quando o programa inicia, ele faz uma busca no banco.txt e transfere os dados salvos para o vetor de estruturas.

4.4.5 Comandos de Banco de dados

No processo servidor foram criados 4 casos de uso de banco de dados: INSERT, DELETE, SELECT e UPDATE. No caso do SELECT, não foram criados comandos adicionais como WHERE. Todos os comandos operam pelo ID, sendo que INSERT e UPDATE precisam do texto digitado. Esses comandos estão todos dentro da função "tratarRequisicao()", conforme o código a seguir:

```

DWORD WINAPI tratarRequisicao(LPVOID arg) {
    char* req = (char*)arg;
    char comando[10];
    int id;
    char nome[MAX_NOME];

    if (strncmp(req, "INSERT", 6) == 0) { ...
    else if (strncmp(req, "DELETE", 6) == 0) { ...
    else if (strncmp(req, "SELECT", 6) == 0) { ...
    else if (strncmp(req, "UPDATE", 6) == 0) { ...
    else {
        printf("[ERRO] Comando invalido: %s", req);
    }

    free(req);
    return 0;
}

```

```

if (strncmp(req, "INSERT", 6) == 0) {
    sscanf(req, "%s %d %s", comando, &id, nome);
    EnterCriticalSection(&cs);

    // Verificar se o ID já existe no banco
    int existe = 0;
    for (int i = 0; i < total; i++) {
        if (banco[i].id == id) {
            existe = 1;
            break;
        }
    }

    if (!existe) {
        banco[total].id = id;
        strncpy(banco[total].nome, nome, MAX_NOME);
        total++;
        salvarBanco();
        printf("[INSERIDO] id=%d nome=%s\n", id, nome);
    } else {
        printf("[ERRO] ID %d ja existe. Insercao nao realizada.\n", id);
    }

    LeaveCriticalSection(&cs);
}

```

5 Resultados obtidos

Após a implementação do sistema, foram realizados quatro testes para verificar a execução do programa como um simulador de gerenciador de banco de dados. Os testes foram feitos por meio de inserções de comandos no console do processo cliente e pela verificação manual do .txt após as operações. O banco.txt foi

manualmente resetado entre o teste 1 e o teste 2. A seguir seguem as imagens dos testes e seus resultados:

TESTE 1:

The screenshot shows a Windows command prompt window with the following text:

```

C:\Windows\system32\cmd.e: X + v
Digite comandos (INSERT, DELETE, SELECT, UPDATE):
INSERT 1 samuel
[INSERIDO] id=1 nome=samuel
INSERT 2 david
[INSERIDO] id=2 nome=david
INSERT 3 teste
[INSERIDO] id=3 nome=teste
DELETE 3
[REMOVIDO] id=3
[ERRO] ID 3 nao encontrado
UPDATE 1 samuel_andrade
[ATUALIZADO] id=1 novo_nome=samuel_andrade
SELECT 2
[SELECT] id=2 nome=david
|
  
```

To the right of the command prompt is a file named `banco.txt` with the following content:

```

output > banco.txt
1 1 samuel_andrade
2 2 david
3
  
```

O primeiro teste serviu para:

- 1- Verificar a execução correta dos 4 comandos
- 2- Verificar a escrita dos dados no banco.txt

TESTE 2:

The screenshot shows a Windows command prompt window with the following text:

```

C:\Windows\system32\cmd.e: X + v
Digite comandos (INSERT, DELETE, SELECT, UPDATE):
INSERT 1 samuel
[INSERIDO] id=1 nome=samuel
INSERT 2 david
[INSERIDO] id=2 nome=david
INSERT 3 teste
[INSERIDO] id=3 nome=teste
comando_invalido 2 teste
[ERRO] Comando invalido: comando_invalido 2 teste
DELETE 2
[REMOVIDO] id=2
UPDATE 2 teste_2
[ERRO] ID 2 nao encontrado para atualizacao
UPDATE 3 teste_2
[ATUALIZADO] id=3 novo_nome=teste_2
SELECT 3
[SELECT] id=3 nome=teste_2
  
```

To the right of the command prompt is a file named `banco.txt` with the following content:

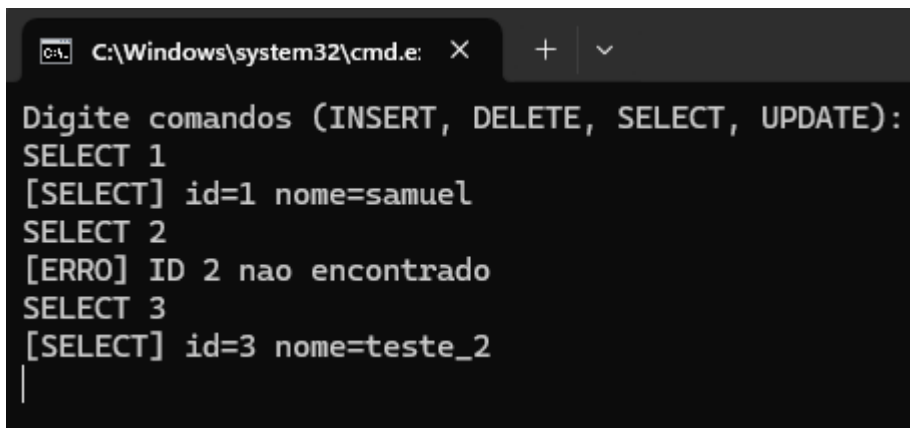
```

output > banco.txt
1 1 samuel
2 3 teste_2
3
  
```

O segundo teste serviu para:

- 1- Verificar a resposta do sistema à comandos inválidos
- 2- Verificar se a exclusão de uma tupla no banco de dados interferia no registro do ID tanto no vetor local quanto no .txt

TESTE 3:

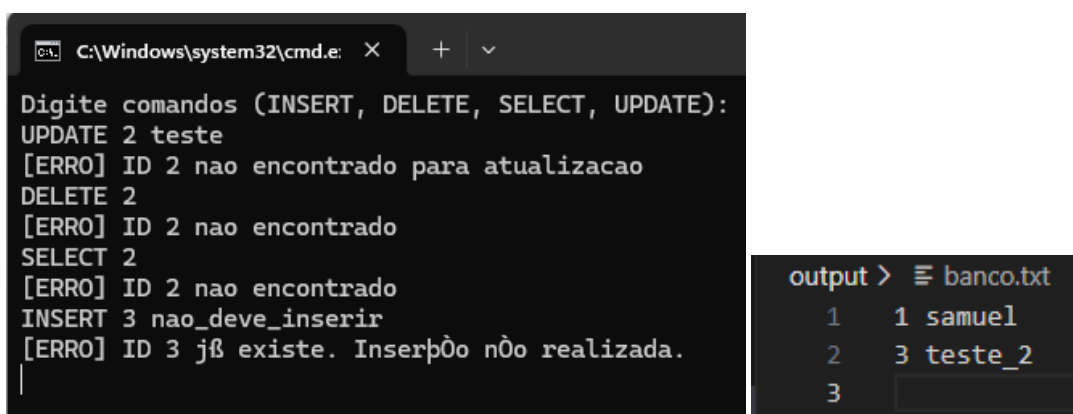


```

C:\Windows\system32\cmd.e: X + v
Digite comandos (INSERT, DELETE, SELECT, UPDATE):
SELECT 1
[SELECT] id=1 nome=samuel
SELECT 2
[ERRO] ID 2 nao encontrado
SELECT 3
[SELECT] id=3 nome=teste_2
|
  
```

O terceiro teste serviu para verificar a execução da leitura do .txt e criação do vetor local de structs.

TESTE 4:



```

C:\Windows\system32\cmd.e: X + v
Digite comandos (INSERT, DELETE, SELECT, UPDATE):
UPDATE 2 teste
[ERRO] ID 2 nao encontrado para atualizacao
DELETE 2
[ERRO] ID 2 nao encontrado
SELECT 2
[ERRO] ID 2 nao encontrado
INSERT 3 nao_deve_inserir
[ERRO] ID 3 j&eacute; existe. Inser&ccedil;o n&eacute;o realizada.
|
  
```

output > banco.txt

1	1 samuel
2	3 teste_2
3	

O quarto teste buscou:

- 1- Verificar a resposta do sistema ao tentar operar com IDs inexistentes
- 2- Verificar a possibilidade de inserçao com ID repetido

6 Análise e discussão de resultados

Após a implementação do sistema e a realização dos testes com as operações de inserção, remoção, consulta e atualização, os resultados observados confirmaram que a solução proposta atende aos requisitos do projeto. A seguir, são discutidos os principais pontos decorrentes dos testes realizados:

6.1 Validação das Operações

- **INSERT:**

Os testes com a operação INSERT demonstraram que o sistema é capaz de adicionar registros corretamente ao vetor de dados e atualizar o arquivo de persistência (banco.txt) de forma consistente. Em várias inserções consecutivas, os registros foram armazenados sem sobreposição ou perda de dados, conforme evidenciado pelas mensagens de confirmação no console. Também foi constatado que o programa não permite a inserção de IDs repetidos

- **DELETE:**

A operação DELETE foi testada removendo registros existentes e verificando a realocação dos elementos no vetor. Ao remover um registro, os registros subsequentes foram deslocados adequadamente, e a variável que armazena o total de registros foi decrementada. Foram observados também casos em que o identificador informado não estava presente, resultando na exibição de mensagens de erro apropriadas. Foi constatado um erro onde a mensagem de erro é exibida mesmo após a mensagem de confirmação de exclusão quando se exclui o último elemento do banco de dados.

- **SELECT:**

Com a operação SELECT, os testes verificaram que o sistema realiza corretamente a busca por registros pelo identificador. As consultas a registros existentes retornaram os dados corretos, enquanto requisições para identificadores inexistentes foram tratadas com mensagens de erro, demonstrando o manejo adequado das condições de busca.

- **UPDATE:**

Os testes com a operação UPDATE comprovaram que a atualização dos dados de um registro, com base no identificador, ocorre de maneira correta e

imediate. Após a atualização, o novo valor do campo foi refletido tanto na estrutura de dados quanto no arquivo persistente, garantindo a integridade dos dados modificados.

6.2 Controle de Concorrência e Sincronização

A utilização da Critical Section como mecanismo de sincronização foi essencial para a correta execução dos testes em ambiente com múltiplas threads. Durante os cenários de acesso simultâneo, observou-se que a exclusão mútua imposta pelo recurso evitou conflitos e condições de corrida, protegendo as operações críticas de acesso e modificação no vetor e no arquivo de persistência. Os resultados indicaram que, mesmo com múltiplas requisições sendo processadas em paralelo, o sistema manteve a consistência dos dados sem apresentar erros de corrupção ou inconsistências.

6.3 Desempenho e Robustez

Embora o foco do projeto tenha sido a correta implementação das operações de manipulação de dados e o controle de concorrência, os testes também apontaram uma execução eficiente do sistema. As respostas às requisições foram processadas em tempo compatível com a complexidade das operações implementadas. A robustez da solução foi confirmada pela capacidade de tratar comandos inválidos e identificar corretamente situações como buscas sem resultados ou tentativas de remover registros inexistentes.

6.4 Considerações Finais da Análise

Em resumo, os resultados dos testes demonstram que:

- Todas as operações (INSERT, DELETE, SELECT e UPDATE) foram executadas conforme esperado.
- O controle de concorrência, implementado por meio da Critical Section, foi eficaz na prevenção de condições de corrida e na manutenção da integridade dos dados.
- O uso do IPC via pipe permitiu uma comunicação fluida entre os processos, possibilitando a simulação de um ambiente distribuído básico com múltiplos componentes interagindo de forma coordenada.

Esses resultados confirmam que o sistema cumpre os objetivos propostos no projeto, servindo como uma implementação prática e funcional dos conceitos teóricos de comunicação entre processos e programação concorrente.

REFERÊNCIAS

TANENBAUM, Andrew S.; BOS, Herbert. *Sistemas operacionais modernos*. [S.l.]: Pearson, [2015]. Disponível em: <https://archive.org/details/andrew-s.-tanenbaum-herbert-bos-sistemas-operacionais-modernos-pearson/page/n7/mode/2up>. Acesso em: 15 abr. 2025.