HÉRICK VITOR VIEIRA BITTENCOURT

ANÁLISE DE Page Faults

Trabalho M2

Itajaí (SC), novembro de 2024



UNIVERSIDADE DO VALE DO ITAJAÍ

CURSO DE BACHAREL EM

CIÊNCIA DA COMPUTAÇÃO

ANÁLISE de Page faults

Trabalho M2

por

Hérick Vitor Vieira Bittencourt

Relatório da análise de page faults em aplicações Windows e Linux como parte do trabalho M2 da matéria de Sistemas Operacionais

Orientador: Felipe Viel

Itajaí (SC), novembro de 2024

1. **SOBRE O PROJETO**

Para consolidar o aprendizado sobre escalonamento e memoria, o projeto consiste na análise da geração de page faults em três programas diferentes, o primeiro sendo um alocador de memória com o tamanho de buffer predefinido durante tempo de compilação, o segundo programa analisado deve ser o sistema de comunicação entre processos e multithread desenvolvido durante o trabalho M1, por fim, deve ser feita a análise de page faults de um software multiplataforma, para avaliar a diferença dos resultados entre diferentes os sistemas operacionais Windows e Linux.

* 1. Análise do programa “Memory Cost”

O programa C++ fornecido para a execução do projeto possui como parâmetro o valor que será usado para alocar em memória e realizar as operações, estas sendo:

* Alocação de memória

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

* Escrita e leitura de valores

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

* Alocação + Escrita repetidamente

Uma imagem contendo Interface gráfica do usuário

Descrição gerada automaticamente

* Alocação + Leitura repetidamenteTexto

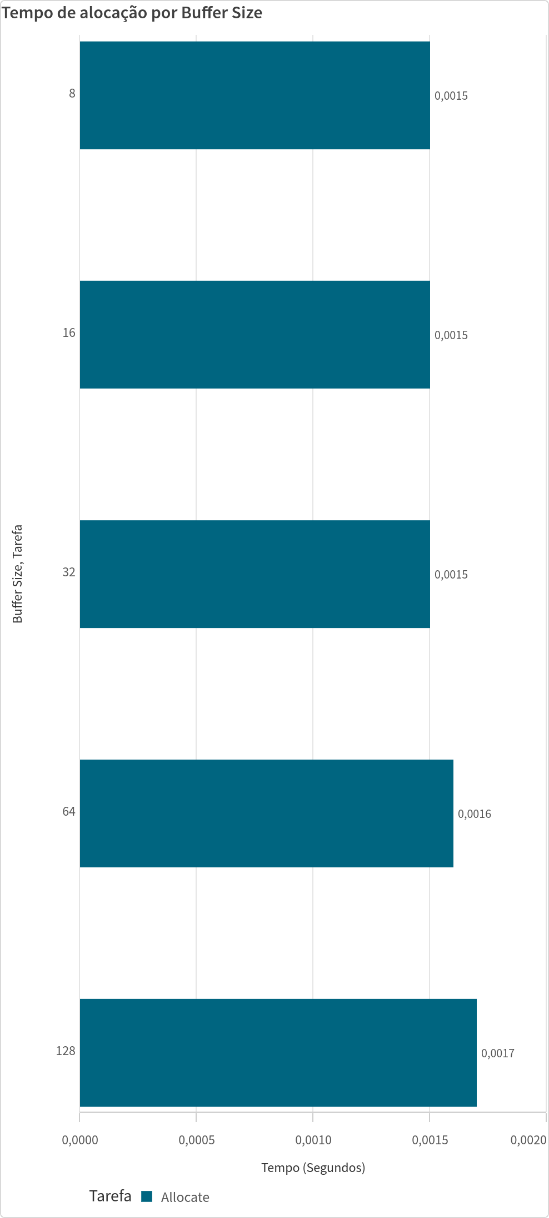
  Descrição gerada automaticamente

Os resultados foram registrados e carregados no Qlik Sense para a geração dos gráficos, facilitando a interpretação das tarefas sob diferentes tamanhos de buffer.

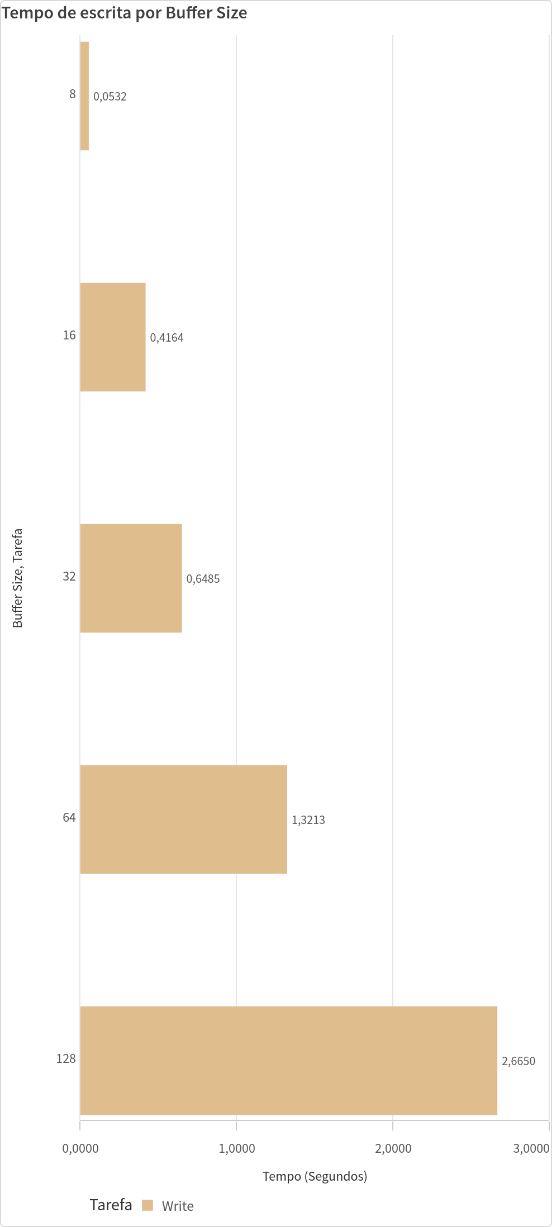
Gráfico, Gráfico de barras

Descrição gerada automaticamente

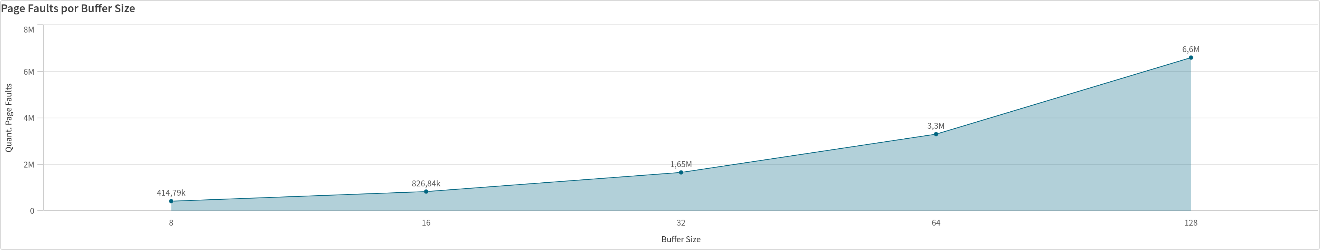
Ao comparar os resultados, torna-se aparente que a tarefa de alocar memoria é consistente entre todos os tamanhos de buffer fornecidos, no entanto, a leitura e escrita aumentam consideravelmente, dobrando de tempo para cada incremento no tamanho de buffer, isto serve como indício de que as leituras e escrita de blocos consideráveis de dados podem afetar drasticamente o desempenho, ainda que realizados na memória principal.



Gráfico, Gráfico de caixa estreita

Descrição gerada automaticamente

Ao analisar o Memory Cost no Process Explorer, é observado que o programa inicia com poucas page faults, porém rapidamente aumenta ao chegar na seção de leitura, visto que necessita acessar endereços não presentes na memória principal, porém, como o programa não possui vazamentos de memória, a quantidade de page faults tornou-se previsível entre múltiplas execuções e o crescimento esperado é que o valor dobre a cada incremento do tamanho de buffer.



* + 1. Códigos
* server.c

//Servidor pipe (testado usando WSL)

//

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/stat.h>

#include <errno.h>

#define STRING\_SOCK\_PATH "/tmp/M1SO/String"

#define NUMBER\_SOCK\_PATH "/tmp/M1SO/Number"

/\* Escolhi 4 threads pois considerando em um servidor de 8 nucleos

lógicos onde ele é o principal processo, faz sentido cada pipe

igualmente balancear a alocação, porém, se rodar em um

ambiente com mais processos ativos (Comp. Pessoais por exemplo),

pode considerar diminuir para evitar slowdown devido a

concorrencia com outros processos existentes

\*/

#define THREAD\_POOL\_SIZE 4

int socketSetup(char \*SOCK\_PATH);

void \*threadHandler(void \*arg);

typedef struct {

char \*sock\_path;

int server\_sock;

int id;

} thread\_arg\_t;

void prepareSocketFolder(){

struct stat st = {0};

// Procura por pasta

if (stat("/tmp/M1SO", &st) == -1) {

// Cria diretório

if (mkdir("/tmp/M1SO", 0700) == 0) {

printf("Pasta temporaria criada com sucesso\n");

} else {

perror("Falha ao criar o diretorio: ");

return 1;

}

} else {

printf("Pasta temporaria OK\n");

}

return 0;

}

int main()

{

// Prepara pasta do socket e thread pool

prepareSocketFolder();

pthread\_t

str\_thread\_pool[THREAD\_POOL\_SIZE],

num\_thread\_pool[THREAD\_POOL\_SIZE]

;

// Prepara o pipe de string e cria threads

int str\_sock = socketSetup(STRING\_SOCK\_PATH);

for (int i = 0; i < THREAD\_POOL\_SIZE; i++) {

thread\_arg\_t \*arg = malloc(sizeof(thread\_arg\_t));

arg->sock\_path = STRING\_SOCK\_PATH;

arg->server\_sock = str\_sock;

arg->id = i;

pthread\_create(&str\_thread\_pool[i], NULL, threadHandler, arg);

}

// Prepara o pipe de numero e cria threads

int num\_sock = socketSetup(NUMBER\_SOCK\_PATH);

for (int i = 0; i < THREAD\_POOL\_SIZE; i++) {

thread\_arg\_t \*arg = malloc(sizeof(thread\_arg\_t));

arg->sock\_path = NUMBER\_SOCK\_PATH;

arg->server\_sock = num\_sock;

arg->id = i;

pthread\_create(&num\_thread\_pool[i], NULL, threadHandler, arg);

}

// Segura o thread principal de terminar sem a pool acabar

// por mais que nunca vá acabar sem intervenção manual...

for (int i = 0; i < THREAD\_POOL\_SIZE; i++) {

pthread\_join(str\_thread\_pool[i], NULL);

pthread\_join(num\_thread\_pool[i], NULL);

}

return 0;

}

int socketSetup(char \*SOCK\_PATH){

int

server\_sock, // Server socket

len // Tamanho do socket

;

struct sockaddr\_un local; // Endereço p/ o socket

// Criação do pipe

server\_sock = socket(AF\_UNIX, SOCK\_STREAM, 0);

if (server\_sock < 0)

{

perror("Falha em criar o pipe");

return 1;

}

memset(&local, 0, sizeof(local)); // server addr. = 0

local.sun\_family = AF\_UNIX; // endereçamento UNIX

strncpy(local.sun\_path, SOCK\_PATH, sizeof(local.sun\_path) - 1); // Define server addr.

unlink(local.sun\_path); // Apaga socket pre-existente

len = strlen(local.sun\_path) + sizeof(local.sun\_family); // len = server addr. + familia

// Tentativa de colocar o socket no endereço

if (bind(server\_sock, (struct sockaddr \*)&local, len) < 0)

{

perror("Falha em capturar o socket");

close(server\_sock);

return 1;

}

// Aceitar no server\_sock até X conexões simultaneas

if (listen(server\_sock, 5) < 0)

{

perror("Falha em escutar o socket");

close(server\_sock);

return 1;

}

// Conectado!

printf("Servidor ouvindo em %s...\n", SOCK\_PATH);

return server\_sock;

}

void stringProcess(char \*buffer){

int first = 0;

int last = strlen(buffer) - 1;

char temp;

// Inverte caracteres da string

while (first < last) {

temp = buffer[first];

buffer[first] = buffer[last];

buffer[last] = temp;

first++;

last--;

}

}

void numberProcess(char \*buffer) {

// Verifica se é um numero, se não for, sobrescreve o buffer por uma mensagem de erro e retorna

for (int i = 0; i < strlen(buffer); i++) {

if (buffer[i] < '0' || buffer[i] > '9') {

snprintf(buffer, 1024, "ERRO: Nao eh um numero");

return;

}

}

// Converte string p/ numero e incrementa +10

int number = atoi(buffer);

number += 10;

// Converte numero p/ string

snprintf(buffer, 1024, "%d", number);

}

void \*threadHandler(void \*arg){

// Define o server address no escopo local e apaga o arg alocado

thread\_arg\_t \*thread\_arg = (thread\_arg\_t \*)arg;

char \*sock\_path = thread\_arg->sock\_path;

int server\_sock = thread\_arg->server\_sock;

int thread\_id = thread\_arg->id;

free(thread\_arg);

//

struct sockaddr\_un remote;

socklen\_t len = sizeof(remote);

char buffer[1024];

// Thread continuará aceitando conexões

while (1) {

printf("%s [THREAD %d]: Aguardando conexao\n", sock\_path, thread\_id);

int client\_sock = accept(server\_sock, (struct sockaddr \*)&remote, &len);

if (client\_sock < 0) {

perror("Falha em aceitar conexão");

continue;

}

printf("%s [THREAD %d]: Cliente conectado\n", sock\_path ,thread\_id);

// Lendo buffer

if (read(client\_sock, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(client\_sock);

continue;

}

printf("%s [THREAD %d] Dado recebido: %s\n", sock\_path, thread\_id, buffer);

// Processamento do buffer conforme a qual socket pertence

if (sock\_path == STRING\_SOCK\_PATH){

stringProcess(&buffer);

} else {

numberProcess(&buffer);

}

printf("%s [THREAD %d] Enviando p/ cliente: %s\n", sock\_path, thread\_id, buffer);

// Retorna para o client

if (write(client\_sock, buffer, strlen(buffer) + 1) < 0)

{

perror("Falha em escrever no socket");

close(client\_sock);

continue;

}

// Pedido encerrado, loop continua

close(client\_sock);

}

return NULL;

}

* string\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#define SOCK\_PATH "/tmp/M1SO/String"

void sendOnce(const char \*inputString);

void sendInfinitelyRandom(float interval);

void sendInfinitelySame(const char \*inputString, float interval);

void generateRandomString(char \*buffer, size\_t length);

int connectToServer();

int main()

{

char inputString[1024];

int mode;

float interval;

// Solicitar a string do usuário

printf("O servidor irá receber a string e inverter\n");

printf("Entre com o dado a ser enviado: ");

fgets(inputString, sizeof(inputString), stdin);

inputString[strcspn(inputString, "\n")] = '\0'; // Remove o caractere de nova linha

// Solicitar o modo

printf("Escolha o modo:\n1. Enviar uma vez\n2. Enviar infinitamente mensagens aleatórias\n3. Enviar infinitamente a mesma mensagem\n");

scanf("%d", &mode);

getchar(); // Consumir o caractere de nova linha deixado pelo scanf

if (mode == 1) {

sendOnce(inputString);

} else if (mode == 2 || mode == 3) {

// Solicitar o intervalo de tempo

printf("Entre com o intervalo de tempo (em segundos): ");

scanf("%f", &interval);

getchar(); // Consumir o caractere de nova linha deixado pelo scanf

if (mode == 2) {

sendInfinitelyRandom(interval);

} else if (mode == 3) {

sendInfinitelySame(inputString, interval);

}

} else {

printf("Modo inválido.\n");

}

return 0;

}

void sendOnce(const char \*inputString) {

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, inputString, strlen(inputString) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor.\n");

// Ler dados do servidor

char buffer[1024];

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

}

void sendInfinitelyRandom(float interval) {

char buffer[1024];

while (1) {

// Gerar string aleatória de 10 caracteres

generateRandomString(buffer, 10);

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, buffer, strlen(buffer) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor: %s\n", buffer);

// Ler dados do servidor

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

// Dormir por um curto período para evitar sobrecarregar o servidor

usleep((int)(interval \* 1000000)); // Converter segundos para microsegundos

}

}

void sendInfinitelySame(const char \*inputString, float interval) {

while (1) {

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, inputString, strlen(inputString) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor: %s\n", inputString);

// Ler dados do servidor

char buffer[1024];

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

// Dormir por um curto período para evitar sobrecarregar o servidor

usleep((int)(interval \* 1000000)); // Converter segundos para microsegundos

}

}

void generateRandomString(char \*buffer, size\_t length) {

const char charset[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";

if (length) {

for (size\_t i = 0; i < length; i++) {

int key = rand() % (int)(sizeof(charset) - 1);

buffer[i] = charset[key];

}

buffer[length] = '\0';

}

}

int connectToServer() {

int sockfd, len;

struct sockaddr\_un remote;

// Criar socket

sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

if (sockfd < 0)

{

perror("Falha em criar o socket");

return -1;

}

// Conectar ao servidor

memset(&remote, 0, sizeof(remote));

remote.sun\_family = AF\_UNIX;

strncpy(remote.sun\_path, SOCK\_PATH, sizeof(remote.sun\_path) - 1);

len = strlen(remote.sun\_path) + sizeof(remote.sun\_family);

if (connect(sockfd, (struct sockaddr \*)&remote, len) < 0)

{

perror("Falha em conectar no servidor");

close(sockfd);

return -1;

}

return sockfd;

}

* number\_client.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/un.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#define SOCK\_PATH "/tmp/M1SO/Number"

void sendOnce(const char \*inputNumber);

void sendInfinitelyRandom(float interval);

void sendInfinitelySame(const char \*inputNumber, float interval);

void generateRandomNumber(char \*buffer, size\_t length);

int connectToServer();

int main()

{

char inputNumber[1024];

int mode;

float interval;

// Solicitar o número do usuário

printf("O servidor irá receber um número\n");

printf("Entre com o número a ser enviado: ");

fgets(inputNumber, sizeof(inputNumber), stdin);

inputNumber[strcspn(inputNumber, "\n")] = '\0'; // Remove o caractere de nova linha

// Solicitar o modo

printf("Escolha o modo:\n1. Enviar uma vez\n2. Enviar infinitamente números aleatórios\n3. Enviar infinitamente o mesmo número\n");

scanf("%d", &mode);

getchar(); // Consumir o caractere de nova linha deixado pelo scanf

if (mode == 1) {

sendOnce(inputNumber);

} else if (mode == 2 || mode == 3) {

// Solicitar o intervalo de tempo

printf("Entre com o intervalo de tempo (em segundos): ");

scanf("%f", &interval);

getchar(); // Consumir o caractere de nova linha deixado pelo scanf

if (mode == 2) {

sendInfinitelyRandom(interval);

} else if (mode == 3) {

sendInfinitelySame(inputNumber, interval);

}

} else {

printf("Modo inválido.\n");

}

return 0;

}

void sendOnce(const char \*inputNumber) {

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, inputNumber, strlen(inputNumber) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor.\n");

// Ler dados do servidor

char buffer[1024];

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

}

void sendInfinitelyRandom(float interval) {

char buffer[1024];

while (1) {

// Gerar número aleatório de 3 digitos

generateRandomNumber(buffer, 3);

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, buffer, strlen(buffer) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor: %s\n", buffer);

// Ler dados do servidor

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

// Dormir por um curto período para evitar sobrecarregar o servidor

usleep((int)(interval \* 1000000)); // Converter segundos para microsegundos

}

}

void sendInfinitelySame(const char \*inputNumber, float interval) {

while (1) {

int sockfd = connectToServer();

if (sockfd < 0) {

return;

}

// Enviar dados para o servidor

if (write(sockfd, inputNumber, strlen(inputNumber) + 1) < 0)

{

perror("Falha em escrever no socket");

close(sockfd);

return;

}

printf("Dado enviado ao servidor: %s\n", inputNumber);

// Ler dados do servidor

char buffer[1024];

if (read(sockfd, buffer, sizeof(buffer)) < 0)

{

perror("Falha em ler do socket");

close(sockfd);

return;

}

printf("Dado recebido: %s\n", buffer);

// Fechar o socket

close(sockfd);

// Dormir por um curto período para evitar sobrecarregar o servidor

usleep((int)(interval \* 1000000)); // Converter segundos para microsegundos

}

}

void generateRandomNumber(char \*buffer, size\_t length) {

const char charset[] = "0123456789";

if (length) {

for (size\_t i = 0; i < length; i++) {

int key = rand() % (int)(sizeof(charset) - 1);

buffer[i] = charset[key];

}

buffer[length] = '\0';

}

}

int connectToServer() {

int sockfd, len;

struct sockaddr\_un remote;

// Criar socket

sockfd = socket(AF\_UNIX, SOCK\_STREAM, 0);

if (sockfd < 0)

{

perror("Falha em criar o socket");

return -1;

}

// Conectar ao servidor

memset(&remote, 0, sizeof(remote));

remote.sun\_family = AF\_UNIX;

strncpy(remote.sun\_path, SOCK\_PATH, sizeof(remote.sun\_path) - 1);

len = strlen(remote.sun\_path) + sizeof(remote.sun\_family);

if (connect(sockfd, (struct sockaddr \*)&remote, len) < 0)

{

perror("Falha em conectar no servidor");

close(sockfd);

return -1;

}

return sockfd;

}

* + 1. Resultados

Após a criação dos processos, foram realizados 3 testes com o sistema, testando cada pipe em diferentes condições, no pipe de string, o servidor devolve as strings invertidas, enquanto no pipe numérico, o valor é incrementado em 10, os resultados ocorreram com sucesso.

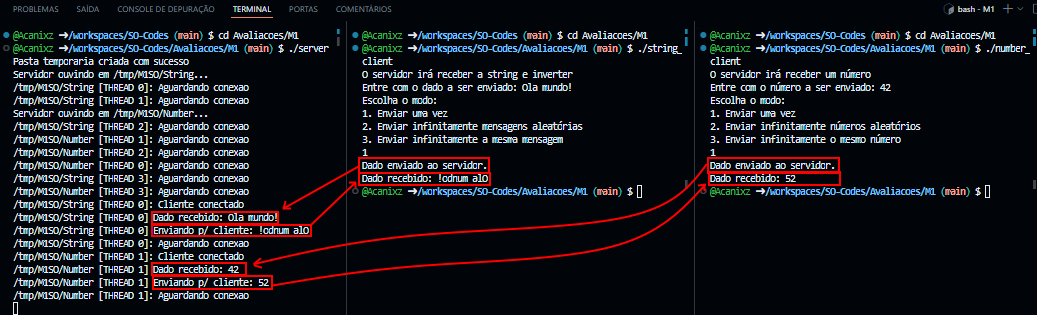


Figura 1- Demonstração dos pipes sendo executados individualmente

Com os pipes executando corretamente, torna-se necessário confirmar a presença do paralelismo no servidor, para este caso, foi desenvolvido a opção para gerar aleatoriamente um valor para que cada cliente envie em um intervalo de tempo, os resultados com 1 cliente para cada tipo ocorreram com sucesso.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Figura 2 - Teste de paralelismo, dois clientes solicitando ao mesmo tempo para o servidor informações diferentes

Por fim, um stress test foi conduzido para testar os limites do servidor, considerando que há 4 threads disponíveis por cada thread pool, utilizando os 4 threads por pool ocorreu perfeitamente, todos os clientes foram respondidos em tempo hábil.

Nota: O teste foi realizado em um uma máquina dual-core no Github Codespaces

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Figura 3- Stress Test do servidor com 8 clientes

* + 1. Justificativa

O programa do servidor permite definir quantos threads serão criados em cada thread pool, como o programa foi inicialmente criado em uma máquina de 4 núcleos, o valor durante os testes ficou definido para 8 threads no total, no entanto, vale ressaltar que a quantidade de threads utilizáveis depende diretamente de vários fatores, como a distribuição de carga entre os processos já existentes, a quantidade de threads escolhida deve ser ajustada conforme o ambiente em que a aplicação se encontra.

1. CONCLUSÕES

O modelo servidor-cliente desenvolvido demonstra de forma eficaz as capacidades de comunicação entre processos utilizando pipes. A implementação se destaca por seu tempo de resposta extremamente eficiente, alcançado através da divisão do processamento das requisições em múltiplas threads. Essa abordagem permite o paralelismo, atendendo simultaneamente a solicitações de diferentes tipos e de mesmo tipo. Com isso, o sistema não só otimiza o uso dos recursos disponíveis, mas também garante uma performance robusta e escalável, adequada para diversos cenários de aplicação