

Breno Farias da Silva
Felipe Archanjo da Cunha Mendes
Pamella Lissa Sato Tamura
Thaynara Ribeiro Falcão dos Santos

Laboratório 03: Manipulação de Threads

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Operacionais do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR
Departamento Acadêmico de Computação – DACOM
Bacharelado em Ciência da Computação – BCC

Campo Mourão
Março / 2022

Sumário

1	Introdução	3
2	Objetivos	3
3	Fundamentação	3
4	Materiais	3
5	Procedimentos e Resultados	4
	5.1 Parte 1: Análise de Threads	4
	5.2 Parte 2: Programação	7
6	Conclusões	14
7	Referências	14

1 Introdução

O relatório está dividido em duas partes, a primeira reservada para a manipulação de threads além de ser composta por perguntas e respostas. Já a segunda parte, está relacionada à programação da qual engloba o desenvolvimento de programas que criam N threads (5) para diversos objetivos.

2 Objetivos

O objetivo do trabalho consiste em, primeiramente, aprimorar os conhecimentos estudados em sala de aula sobre o funcionamento das threads. Além de construir programas que desenvolvam determinadas soluções envolvendo threads e explorar bibliotecas como pthreads.

3 Fundamentação

A definição de uma thread segundo Tanenbaum é um fluxo de execução que pode ser associado a outras threads para formar maiores processos, ou seja, são pequenas sequências de instruções que possuem um fluxo de execução independente podendo ser utilizadas dentro no núcleo dos Sistemas Operacionais (threads de núcleo).

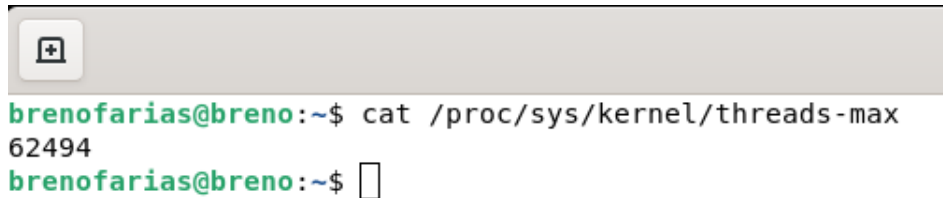
As perguntas e respostas foram fundamentadas pelas aulas e materiais fornecidos pelo próprio Prof. Dr. Rodrigo Campiolo, assim como o desenvolvimento do programa e códigos descritos na parte dois do documento. Além disso, as definições de comandos e conceitos que foram obtidos baseadas em aulas e, parcialmente, por pesquisas sequenciais por cada membro da equipe.

4 Materiais

- Debian 11 LTS (ISO)
- VirtualBox 6.1.32
- Debian GNU/Linux 11 (Bullseye)
- Ryzen 7 3800x
- 8GiB de RAM

Sendo assim, é possível constatar que o processo com maior número de threads em nossa máquina virtual estão relacionados ao `systemd`, o qual é um processo daemon que gerencia outros processos daemons.

2. Qual o número máximo de threads que o seu sistema suporta? Para verificar o número máximo de threads suportada pelo sistema operacional usado, basta executar o seguinte comando: `cat /proc/sys/kernel/threads-max` (Figura 3) Com isso, é possível ver que o limite máximo de threads suportada pela máquina virtual do Debian é de 62494 threads.



```
brenofarias@breno:~$ cat /proc/sys/kernel/threads-max
62494
brenofarias@breno:~$
```

Figura 3 – Demonstração do comando `cat /proc/sys/kernel/threads-max`

3. Verifique o tempo de execução do programa da questão 3, parte 2, considerando: 1 thread, 2 threads, 4 threads, 8 threads e 16 threads. Descreva o hardware (processador, memória e número de núcleos, tamanho da matriz usada nos testes e o tempo de execução para cada teste).

Considerando uma matriz 100 x 200, temos os seguintes resultados em uma máquina com um processador Ryzen 7 3800x de 8 núcleos e memória RAM de 8GB:

N = 1: 9.542 milissegundos

N = 2: 9.568 milissegundos

N = 4: 10.106 segundos

N = 8: 11.379 milissegundos

N = 16: 12.285 milissegundos

Conclusão: Do exposto, é possível concluir que para valores fixos de linhas (M) e colunas (N), a única variável em questão tornou-se o número de threads usadas. Seguindo o fluxo dos valores para N sendo 1, 2, 4, 8 e 16, torna-se evidente que o menor tempo de execução foi dado para N = 1 e o maior tempo de execução foi dado para N = 16. Este fato ocorreu devido a problemas com concorrência de dados, paralelismo e sincronização de threads. Há dois tipos de paralelismo, sendo eles, o paralelismo de função e paralelismo de dados.

O paralelismo de função é dado pelo uso de diferentes threads para executar diferentes funções. Neste caso, o paralelismo de função se dá pela divisão de threads diferentes para executar cálculos distintos, definidos pela mediana de cada linha e a média aritmética de cada coluna.

Já o paralelismo de dados ocorre pela execução de mesma função, porém em segmentos diferentes do mesmo conjunto de dados. No caso em questão, esse paralelismo

ocorre com a subdivisão das linhas em N subvetores e o mesmo para as colunas. Ou seja, a mesma operação é feita porém em subpartes da matriz. Ao implementar o paralelismo de função, é esperado melhorias significativas de performance, contudo nem sempre o mesmo acontece com a implementação de paralelismo de dados, pois este pode gerar problemas de sincronização os quais nem sempre irão compensar o ganho de desempenho ao adicioná-lo no algoritmo. Se a complexidade computacional da tarefa não é grande e o conjunto de dados também não então, o paralelismo pode simplesmente acarretar em um overhead (sobrecarga) adicional.

Quando criam-se threads, os SO's convencionais precisam criar uma fila para estas threads poderem usar as CPU's disponíveis. Cada thread recebe um tempo para usar a CPU (quantum/time slice) de acordo com a prioridade do processo. Quando o tempo do processo encerra, ele volta para a fila. Assim, a thread executa e para, consecutivamente, até finalizar. Isso é chamado de context switch e, tal processo, tem um custo computacional adicional. Por isso, em certos casos, criar threads demais pode piorar o tempo de processamento.

Sendo assim, a medida que o valor de N aumenta, é notável que o tempo gasto na sincronização das threads não foram suficientes para manter o ganho de performance ao adicionar o paralelismo de dados. Vale lembrar que, sem paralelismo de dados o código não se tornaria mais eficiente ao adicionar mais threads, pois threads diferentes estariam acessando/alterando o mesmo item de dados o que poderia causar problemas como "atualização perdida", pois uma escrita poderia sobrescrever a outra gerando inconsistência nos dados. Além disso, a leitura de dados no disco seria feita uma por vez, o que tiraria o paralelismo do algoritmo.

5.2 Parte 2: Programação

1. Faça um programa (Figura 4 e Figura 5) com N threads que localiza um valor em um vetor de inteiros. O espaço de busca no vetor deve ser distribuído para as N threads.

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>

// Define a quantidade de threads e o tamanho do vetor
#define N 3
#define TV 20
#define VALOR_BUSCA 26

// Estrutura para passar os parametros para as threads
struct parametros {
    int *vetor;
    int valor;
    int inicio;
    int fim;
    int id;
    int tam_vetor;
};

// Função para imprimir o vetor
void imprimir_vetor(int *vetor, int tam_vetor) {
    int id;
    printf("[ ");
    for (i = 0; i < tam_vetor; i++) {
        printf("%d ", vetor[i]);
    }
    printf("]\n");
}

// Função que busca um valor no vetor
void *busca(void *parametros) {
    struct parametros *p = (struct parametros *) parametros;
    int id;
    int *vetor = p->vetor;
    int valor = p->valor;
    int inicio = p->inicio;
    int fim = p->fim;
    int id = p->id;
    int tam_vetor = p->tam_vetor;

    printf("----- Thread %d -----\n", id);
    // Imprime o subvetor que a thread irá buscar
    imprimir_vetor(vetor + inicio, fim - inicio);

    for (i = inicio; i < fim; i++) {
        if (vetor[i] == valor) {
            printf("Thread %d: Encontrei o valor %d no vetor na posição %d\n", id, valor, i);
            return NULL;
        }
    }
    printf("Thread %d: Não encontrei o valor %d no vetor\n", id, valor);
    return NULL;
}

// Função para gerar um vetor com valores aleatórios
void gerar_vetor(int *vetor, int tam_vetor) {
    int i;
    for (i = 0; i < tam_vetor; i++) {
        vetor[i] = rand() % 100;
    }
}

// Função principal
int main() {
    printf("----- Exercício 01 ----- \n\n");

    // Alocação do vetor
    int *vetor = (int *) malloc(TV * sizeof(int));

    // Geração do vetor
    gerar_vetor(vetor, TV);

    // Impressão do vetor
    printf("Vetor: ");
    imprimir_vetor(vetor, TV);
    printf("\n");

    // Alocação do vetor de threads
    pthread_t threads[N];

    // Alocação do vetor de parametros
    struct parametros parametros[N];

    // Cálculo da parte do vetor recebida pelas threads
    int quant = TV / N;
    int qnt_aux = TV % N;

    // Alocação do vetor de threads
    int i;
    for (i = 0; i < N; i++) {
        parametros[i].vetor = vetor;
        parametros[i].valor = VALOR_BUSCA;
        parametros[i].inicio = i * (TV / N);

        // Cada thread recebe sua parte do vetor
        if (i < qnt_aux) {
            parametros[i].tam_vetor = quant + 1;
        } else {
            parametros[i].tam_vetor = quant;
        }

        parametros[i].fim = parametros[i].inicio + parametros[i].tam_vetor;
        parametros[i].id = i;
        parametros[i].tam_vetor = TV;

        // Criação da thread
        pthread_create(&threads[i], NULL, busca, (void *) &parametros[i]);
    }

    // Aguarda as threads terminarem
    for (i = 0; i < N; i++) {
        pthread_join(threads[i], NULL);
    }

    // Liberação da memória
    free(vetor);

    return 0;
}

```

Figura 4 – Demonstração do programa com N threads que localiza um valor em um vetor de inteiros

```

felipolis@DESKTOP-7JJSSPL: ~/Lab03
felipolis@DESKTOP-7JJSSPL:~/Lab03$ gcc ex01.c -o ex01 -lpthread
felipolis@DESKTOP-7JJSSPL:~/Lab03$ ./ex01
----- Exercício 01 -----

Vetor: [ 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 ]

----- Thread 0 -----
[ 83 86 77 15 93 35 86 ]
Thread 0: Não encontrei o valor 26 no vetor

----- Thread 1 -----
[ 86 92 49 21 62 27 90 ]
Thread 1: Não encontrei o valor 26 no vetor

----- Thread 2 -----
[ 90 59 63 26 40 26 ]
Thread 2: Encontrei o valor 26 no vetor na posicao 15

felipolis@DESKTOP-7JJSSPL:~/Lab03$

```

Figura 5 – Terminal do programa demonstrado acima

2. Faça um programa (Figura 6 e Figura 7) com 2 threads que calcule a frequência de elementos em um vetor que contém somente valores de 0 a 9.

```

// Exercício 01
// Objetivo: Criar 3 threads para encontrar o valor 26 em um vetor.
// Vetor: [ 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 ]
// Thread 0: Não encontrei o valor 26 no vetor
// Thread 1: Não encontrei o valor 26 no vetor
// Thread 2: Encontrei o valor 26 no vetor na posicao 15

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

// Vetor e suas dimensões
int vetor[] = {83, 86, 77, 15, 93, 35, 86, 92, 49, 21, 62, 27, 90, 59, 63, 26, 40, 26, 72, 36};
int tam_vetor = sizeof(vetor) / sizeof(int);

// Estrutura para passar os parâmetros para as threads
struct parametros {
    int thread;
    int *vetor;
    int *resposta;
    int inicio;
    int fim;
    int tam_vetor;
};

// Função para imprimir o vetor
void imprime_vetor(int *vetor, int tam_vetor) {
    int i;
    printf("Vetor: [ ");
    for (i = 0; i < tam_vetor; i++) {
        printf("%d ", vetor[i]);
    }
    printf("]\n");
}

// Função para gerar um vetor com valores aleatórios
void gerar_vetor(int *vetor, int tam_vetor) {
    int i;
    for (i = 0; i < tam_vetor; i++) {
        vetor[i] = rand() % 100;
    }
}

// Função para inicializar o vetor com zeros
void inicializar_vetor(int *vetor, int tam_vetor) {
    int i;
    for (i = 0; i < tam_vetor; i++) {
        vetor[i] = 0;
    }
}

// Função para buscar o valor no vetor
void buscar_valor(struct parametros *p) {
    int i;
    int *vetor = p->vetor;
    int *resposta = p->resposta;
    int inicio = p->inicio;
    int fim = p->fim;
    int tam_vetor = p->tam_vetor;

    printf("Thread %d: Buscando o valor 26 entre %d e %d\n", p->thread, inicio, fim);
    for (i = inicio; i < fim; i++) {
        if (vetor[i] == 26) {
            *resposta = i;
        }
    }
}

// Função para contar a frequência de 0 a 9
void contar_frequencia(int *vetor, int tam_vetor) {
    int i;
    int freq[10] = {0};
    for (i = 0; i < tam_vetor; i++) {
        freq[vetor[i]]++;
    }
}

// Função principal
int main() {
    printf("Exercício 01\n");

    // Inicialização do vetor
    inicializar_vetor(vetor, tam_vetor);
    imprime_vetor(vetor, tam_vetor);

    // Estrutura de dados para as threads
    struct parametros p0, p1, p2;

    // Atribuição de parâmetros para as threads
    p0.thread = 0;
    p0.vetor = vetor;
    p0.resposta = &vetor[0];
    p0.inicio = 0;
    p0.fim = 10;
    p0.tam_vetor = tam_vetor;

    p1.thread = 1;
    p1.vetor = vetor;
    p1.resposta = &vetor[10];
    p1.inicio = 10;
    p1.fim = 15;
    p1.tam_vetor = tam_vetor;

    p2.thread = 2;
    p2.vetor = vetor;
    p2.resposta = &vetor[15];
    p2.inicio = 15;
    p2.fim = 20;
    p2.tam_vetor = tam_vetor;

    // Criação das threads
    pthread_t t0, t1, t2;
    pthread_create(&t0, NULL, buscar_valor, &p0);
    pthread_create(&t1, NULL, buscar_valor, &p1);
    pthread_create(&t2, NULL, buscar_valor, &p2);

    // Espera as threads terminarem
    pthread_join(t0, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    // Imprime o vetor de resposta
    printf("Resposta: [ ");
    for (i = 0; i < 3; i++) {
        printf("%d ", *(p0.resposta + i));
    }
    printf("]\n");

    // Contagem da frequência
    contar_frequencia(vetor, tam_vetor);

    // Imprime a frequência
    for (i = 0; i < 10; i++) {
        printf("Frequência do valor %d: %d\n", i, freq[i]);
    }

    return 0;
}

```

Figura 6 – Demonstração do programa com 2 threads enunciado anteriormente


```

felipolis@DESKTOP-7JJSSPL: ~/Lab03
felipolis@DESKTOP-7JJSSPL:~/Lab03$ gcc ex02.c -o ex02 -lpthread
felipolis@DESKTOP-7JJSSPL:~/Lab03$ ./ex02
----- Exercício 02 -----

Vetor: [ 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 ]

----- Thread 0 -----
[ 3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 ]

----- Thread 1 -----
[ 6 0 6 2 6 1 8 7 9 2 0 2 3 7 5 ]

Vetor de resposta: [ 3 2 5 4 0 3 5 4 1 3 ]

A frequencia do numero 0 eh 3
A frequencia do numero 1 eh 2
A frequencia do numero 2 eh 5
A frequencia do numero 3 eh 4
A frequencia do numero 4 eh 0
A frequencia do numero 5 eh 3
A frequencia do numero 6 eh 5
A frequencia do numero 7 eh 4
A frequencia do numero 8 eh 1
A frequencia do numero 9 eh 3
felipolis@DESKTOP-7JJSSPL:~/Lab03$

```

Figura 7 – Execução do programa demonstrado acima

3. Implemente um programa multithread com pthreads que calcule:

a) a mediana de cada linha de uma matriz $M \times N$ e devolva o resultado em um vetor de tamanho M . (Figura 8 e Figura 9)

b) a média aritmética de cada coluna de uma matriz $M \times N$ e devolva o resultado em um vetor de tamanho N . (Figura 10) O programa deve gerar matrizes $M \times N$ com elementos aleatórios para arquivos; usar técnicas de paralelização de funções e de dados; ler matrizes $M \times N$ de arquivos no formato em anexo; gravar os resultados em um arquivo texto. (Figura 11 e Figura 12 e Figura 13)

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "matriz.h"

// Defines
#define NUM_THREADS 5

// Variaveis Globais
int** matriz;

int rows = 6;
int cols = 8;

float* medias;
int* medianas;

// Estrutura
typedef struct {
    int id;
    int quantidade;
} Dados;

// Protótipos
void* thread_media(void* param);
void selectionSort(int* vetor, int tamanho);
void* thread_mediana(void* param);

```

Figura 8 – Demonstração do programa multithread com pthreads calculando a mediana de uma matriz

```
// Função Principal
int main(){

    //Declaração de variáveis
    int op;

    pthread_t threads[NUM_THREADS];
    Dados dados[NUM_THREADS];

    // Receber a opção do usuário
    do{
        printf("Digite:\n[1] - Gerar Matriz\n[2] - Carregar Matriz\n");
        scanf("%d", &op);
    } while(op != 1 && op != 2);

    // Gerar ou carregar a matriz
    if (op == 1){

        printf("Digite o número de linhas: ");
        scanf("%d", &rows);
        printf("Digite o número de colunas: ");
        scanf("%d", &cols);

        matriz = create_matrix(rows, cols);
        srand(time(NULL));
        generate_elements(matriz, rows, cols, 99);

        printf("\nMatriz %dx%d criada com sucesso!\n", rows, cols);

    } else{

        matriz = read_matrix_from_file("data_matrix.in", &rows, &cols);

        printf("\nMatriz carregada com sucesso!\n");
    }

    // Inicio da contagem de tempo
    clock_t start = clock();

    // Output
    printf("\nNumero de Threads: %d\n", NUM_THREADS);
    printf("\nMatriz [%d x %d]\n\n", rows, cols);
```

Figura 9 – Continuação da demonstração do programa iniciado anteriormente

```
/* -----  
    TRABALHANDO COM A MÉDIA  
-----*/  
  
printf("----- MEDIA -----\\n");  
// Inicializar o vetor do resultado das médias  
float resultado_medias[cols];  
for(int i = 0; i < cols; i++){  
    resultado_medias[i] = 0;  
}  
medias = resultado_medias;  
  
// Divisao das colunas em "cols" partes  
int quant = cols / NUM_THREADS;  
int qnt_aux = cols % NUM_THREADS;  
int aux = 0;  
  
for(int i = 0; i < NUM_THREADS; i++){  
    if(i < qnt_aux){  
        dados[i].quantidade = quant + 1;  
    } else{  
        dados[i].quantidade = quant;  
    }  
    dados[i].id = i;  
  
    printf("Thread %d: %d colunas\\n", i, dados[i].quantidade);  
}  
printf("\\n");  
  
print_matrix(matriz, rows, cols);  
  
// Criar as threads para a média  
for(int i = 0; i < NUM_THREADS; i++){  
    pthread_create(&threads[i], NULL, thread_media, (void*) &dados[i]);  
}  
// Esperar as threads terminarem  
for(int i = 0; i < NUM_THREADS; i++){  
    pthread_join(threads[i], NULL);  
}
```

Figura 10 – Continuação da demonstração do programa iniciado anteriormente

```

/* -----
   TRABALHANDO COM A MEDIANA
----- */
printf("\n----- MEDIANA ----- \n");
// Divisão das linhas em "rows" partes
quant = rows / NUM_THREADS;
qnt_aux = rows % NUM_THREADS;
aux = 0;

for(int i = 0; i < NUM_THREADS; i++){
    if(i < qnt_aux){
        dados[i].quantidade = quant + 1;
    } else{
        dados[i].quantidade = quant;
    }
    dados[i].id = i;

    printf("Thread %d: %d linhas\n", i, dados[i].quantidade);
}

// Criação de uma copia da matriz principal
int matriz_copia[rows][cols];
for(int i = 0; i < rows; i++){
    for(int j = 0; j < cols; j++){
        matriz_copia[i][j] = matriz[i][j];
    }
}

// Ordenando as linhas da matriz principal
for(int i = 0; i < rows; i++){
    selectionSort(matriz[i], cols);
}
print_matrix(matriz, rows, cols);

// Inicializar o vetor do resultado das medianas
int resultado_medianas[rows];
for(int i = 0; i < rows; i++){
    resultado_medianas[i] = 0;
}
medianas = resultado_medianas;

// Criar as threads para a mediana
for(int i = 0; i < NUM_THREADS; i++){
    pthread_create(&threads[i], NULL, thread_mediana, (void*) &dados[i]);
}

// Esperar as threads terminarem
for(int i = 0; i < NUM_THREADS; i++){
    pthread_join(threads[i], NULL);
}

// Imprimir os resultados

printf("\nMediana: ");
for(int i=0; i<rows;i++){
    printf(" [%d]", resultado_medianas[i]);
}
printf("\n");

printf("\nMedia: ");
for(int i=0; i<cols;i++){
    printf(" [%2f]", medias[i]);
}
printf("\n");

// Salvar os resultados num arquivo
FILE* arq_result = fopen("results.txt", "w");

fprintf(arq_result, "Mediana: ");
for(int i=0; i<rows;i++){
    fprintf(arq_result, " [%d]", resultado_medianas[i]);
}
fprintf(arq_result, "\n");

fprintf(arq_result, "Media: ");
for(int i=0; i<cols;i++){
    fprintf(arq_result, " [%2f]", medias[i]);
}
fprintf(arq_result, "\n");

printf("\nResultados salvos em results.txt com sucesso!\n");

fclose(arq_result);

// Fim da contagem de tempo
clock_t end = clock();
double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

printf("\nTempo gasto: %f segundos\n", time_spent);

return 0;
}

```

Figura 11 – Continuação da demonstração da implementação do programa enunciado acima

```

// Função de média
void *thread_media(void *param){
    Dados *dados = (Dados*) param;

    int inicio = dados->id * dados->quantidade;
    int fim = inicio + dados->quantidade;

    for(int i = inicio; i < fim; i++){
        for(int j = 0; j < rows; j++){
            medias[i] += (float)matriz[j][i]/ (float)rows;
        }
        //printf("Coluna %d: %f\n", i, medias[i]);
    }
}

// Função de mediana
void selectionSort(int vetor[], int n){
    int i, j, min_idx;
    for (i = 0; i < n-1; i++){
        min_idx = i;
        for (j = i+1; j < n; j++){
            if (vetor[j] < vetor[min_idx]) min_idx = j;
        }
        // Swap
        vetor[min_idx] = vetor[i]; vetor[i] = vetor[min_idx];
    }
}

void *thread_mediana(void *param){
    Dados *dados = (Dados*)param;

    int inicio = dados->id * dados->quantidade;
    int fim = inicio + dados->quantidade;

    for(int i = inicio; i < fim; i++){
        medianas[i] = matriz[i][(int)ceil(cols/2)];
        //printf("Linha %d: %d\n", i, medianas[i]);
    }
}

```

Figura 12 – Continuação da demonstração da implementação do programa enunciado acima

```

felipolis@DESKTOP-7JJSSPL: ~/Lab03
felipolis@DESKTOP-7JJSSPL:~/Lab03$ gcc -g ex03.c matriz.c -o ex03 -lpthread
felipolis@DESKTOP-7JJSSPL:~/Lab03$ ./ex03
Digite:
[1] - Gerar Matriz
[2] - Carregar Matriz
2

Matriz carregada com sucesso!

Numero de Threads: 5

Matriz [6 x 8]

----- MEDIA -----
 1  2  3  4  5  6  7  8
 2  3  4  5  6  7  8  9
 3  4  5  6  7  8  9 10
 4  5  6  7  8  9 10 11
 5  6  7  8  9 10 11 12
 6  7  8  9 10 11 12 13

----- MEDIANA -----
 1  2  3  4  5  6  7  8
 2  3  4  5  6  7  8  9
 3  4  5  6  7  8  9 10
 4  5  6  7  8  9 10 11
 5  6  7  8  9 10 11 12
 6  7  8  9 10 11 12 13

Mediana:  [5] [6] [7] [8] [9] [0]

Media:  [3.50] [4.50] [5.50] [13.00] [15.00] [8.50] [0.00] [0.00]

Resultados salvos em results.txt com sucesso!

Tempo gasto: 0.000908 segundos
felipolis@DESKTOP-7JJSSPL:~/Lab03$

```

Figura 13 – Terminal da demonstração do programa multithread com pthreads e suas exigências

OBS. Segue anexo, no mesmo diretório desse documento, todos os códigos-fontes referentes aos exercícios da parte 2.

6 Conclusões

Após realizados os procedimentos solicitados na descrição da parte 1 e 2 da atividade, finalizou-se a prática do laboratório 3.

7 Referências

- <https://moodle.utfpr.edu.br/mod/resource/view.php?id=415720>
- <https://www.baeldung.com/linux/max-threads-per-process>