



**UFC – UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS DE SOBRAL**

**CURSO DE ENGENHARIA DA COMPUTAÇÃO**

**SISTEMAS OPERACIONAIS**

**PROFESSOR: JONIEL BASTOS BARRETO**

## **RELATÓRIO - LISTA II**

<b>ALUNOS</b>	<b>MATRÍCULA</b>
Yann Lucca Miranda Martins Barros	497746
Marcos Vinicius Andrade de Sousa	496788

## SUMÁRIO

1. THREADS.....	3
2. ESCALONAMENTO.....	9
3. SISTEMAS_DE_ARQUIVOS.....	18

## 1. THREADS

**1.1.** O programa em C usa a biblioteca de threads do linux para criar um thread pai que irá gerar 100 threads filhas, e na criação de cada thread irá ser dito um “olá” referente à thread criada no momento, e na finalização de cada thread será dito um “tchau” referente à thread que terminou o seu processo. Dessa forma, foi criada uma thread pai com o comando `pthread_create`, este comando retorna um valor 0 para uma criação bem sucedida de uma thread, e com isso foi feita um `if` para dizer que a thread foi ou não criada com sucesso. Seguindo o código, com a criação bem sucedida do thread pai, a função `pthread_join` aguarda o término do thread para que os processos de cada thread saiam em ordem. Portanto, usando a mesma lógica para a criação do thread pai, é criado um vetor de `threads[N]`, em que `N` é 100, no topo do código para que dentro da função chamada pelo thread pai, seja executado um laço `for` com `pthread_create` para cada uma das `N` threads do vetor, e finalmente cada thread filha irá dizer um olá na sua criação e um tchau na sua finalização.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#define N 100
pthread_t filho[N];

void *threadFilha(void *num)
{
    int i = *((int *)num);
    printf("Alo Filho[%d]\n", i);
}

void *threadPai()
{
    for (int i = 0; i < N; i++)
    {
        int *num = malloc(sizeof(int));
        *num = i;
        if (pthread_create(&(filho[i]), NULL, threadFilha, num) !=
0)
        {
            printf("\nErro na criação da thread filha[%d]\n", i);
        }
        else
        {
            pthread_join(filho[i], NULL);
        }
    }
}
```

```

    }
    printf("Tchau filho[%d]\n", i);
}
printf("Tchau Pai\n");
}
void main()
{
    pthread_t pai;
    if (pthread_create(&pai, NULL, threadPai, NULL) != 0)
    {
        printf("\nErro na criação da thread PAI\n");
    }
    else
    {
        printf("Alo pai\n");
        pthread_join(pai, NULL);
    }
}

```

```

marcos@marcos-VirtualBox: ~/Área de Trabalho/SO - LISTA 2/parte 1
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$ ./aloPaiFilho
Alo pai
Alo Filho[0]
Tchau filho[0]
Alo Filho[1]
Tchau filho[1]
Alo Filho[2]
Tchau filho[2]
Alo Filho[3]
Tchau filho[3]
Alo Filho[4]
Tchau filho[4]
Alo Filho[5]
Tchau filho[5]
Alo Filho[6]
Tchau filho[6]
Alo Filho[7]
Tchau filho[7]
Alo Filho[8]
Tchau filho[8]
Alo Filho[9]
Tchau filho[9]
Alo Filho[10]
Tchau filho[10]
Alo Filho[11]
Tchau filho[11]
Alo Filho[12]
Tchau filho[12]
Alo Filho[13]
Tchau filho[13]
Alo Filho[14]
Tchau filho[14]
Alo Filho[15]
Tchau filho[15]
Alo Filho[16]
Tchau filho[16]
Alo Filho[17]
Tchau filho[17]

```

**1.2.** O algoritmo calcula o seno de  $x$  através da série de taylor usando a biblioteca de thread do linux. Este programa conta com quatro funções, uma que calcula o fatorial do

denominador, outras duas funções que calculam o somatório de números positivos e negativos, e por fim a função main, que é a função principal, e lá é onde foram criadas as threads. Ademais, a função “somador” é chamada pela thread “somadora” e a função “subtrator” é chamada pela thread subtratora. Assim como a função “factorial” é chamada em cada uma das funções somadoras. Dessa forma é criado duas variáveis globais que armazenam os valores dos somatórios das duas principais funções, e por fim os valores são somados na função main e é exibido para o usuário.

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

#define N 20

double somResult = 0, subResult = 0;

// Fatorial
double factorial(int n)
{
    double fat;
    for (fat = 1; n > 1; n = n - 1)
    {
        fat = fat * n;
    }
    return fat;
}

// Termos positivos da série taylor
void *somador(void *y)
{
    int *x = (int *)y;

    double term, numerator, denominator;

    for (int i = 0; i < N; i += 2)
    {
        numerator = pow(-1, i) * pow((*x), (2 * i + 1));
        denominator = factorial(2 * i + 1);
        term = numerator / denominator;

        somResult += term;
    }
    return NULL;
}
```

```

}

// Termos negativos da série taylor
void *subtrator(void *y)
{
    int *x = (int *)y;

    double term, numerator, denominator;

    for (int i = 1; i < N; i += 2)
    {
        numerator = pow(-1, i) * pow((*x), (2 * i + 1));
        denominator = factorial(2 * i + 1);
        term = numerator / denominator;

        subResult += term;
    }
    return NULL;
}

void main()
{
    int x;
    pthread_t somadora, subtrator; // thread somadora e thread
    subtrator

    printf("Digite o valor de x: ");
    scanf("%d", &x);
    /*Criação da thread somadora*/
    if (pthread_create(&somadora, NULL, &somador, &x) != 0)
    {
        printf("\nErro na criação da thread somadora\n");
    }
    else
    {
        pthread_join(somadora, NULL);
    }
    /*Criação da thread subtrator*/
    if (pthread_create(&subtrator, NULL, &subtrator, &x) != 0)
    {
        printf("\nErro na criação da thread subtrator\n");
    }
    else

```

```

{
    pthread_join(subtratora, NULL);
}

printf("Serie Taylor: %f\n", (somResult + subResult));
}

```

```

marcos@marcos-VirtualBox: ~/Área de Trabalho/SO - LISTA 2/parte 1
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$ dir
aloPaiFilho aloPaiFilho.c entrePrimos entrePrimos.c serieTaylor serieTaylor.c
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$ ./serieTaylor
Digite o valor de x: 2
Serie Taylor: 0.909297
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$

```

**1.3.** A última parte da seção de threads se trata de um algoritmo em C que usa a biblioteca de threads do linux para dividir entre duas threads a tarefa de calcular o número de números primos entre zero e um número digitado pelo usuário. Portanto, para isso foi implementado no código quatro funções, uma main que irá criar as threads, obter a entrada do número fornecido pelo usuário e mostrar a quantidade de números primos, duas funções “counters” que irão conter um laço de repetição para percorrer os valores de 0 até  $N/2$  e de  $((N/2)+1)$  até  $N$ , assim como irão chamar a função “ifPrimo” que irá dizer se o número informado é ou não primo e irá imprimir aquele valor na tela.

```

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

int nPrimos = 0;

// Função que diz se é primo
int ifPrimo(int n)
{
    int counter = 0;

    for (int i = 1; i <= n; i++)
    {
        if (n % i == 0)
        {
            counter++;
        }
    }

    if (counter == 2)

```

```

    {
        printf("[%d] ", n);
        return 1;
    }
    else
    {
        return 0;
    }
}

// Função da thread 1 que conta os primos de 0 até N/2
void *counter1(void *num)
{
    int *n = (int *)num;
    int counterPrimo;
    printf("\n=====NUMEROS_PRIMOS=====\\n");
    for (int i = 2; i <= (*n / 2); i++)
    {
        if (ifPrimo(i) != 0)
        {
            counterPrimo++;
        }
    }
    nPrimos = nPrimos + counterPrimo;
}

// Função da thread 2 que conta os primos de (N/2)+1 até N
void *counter2(void *num)
{
    int *n = (int *)num;
    int counterPrimo;
    printf("\\n");
    for (int i = (*n / 2) + 1; i <= *n; i++)
    {
        if (ifPrimo(i) != 0)
        {
            counterPrimo++;
        }
    }
    printf("\\n=====\\n");
    nPrimos = nPrimos + counterPrimo;
}

```



```

void main()
{

    int n;
    pthread_t thread1, thread2;
    printf("Digite o numero N:");
    scanf("%d", &n);

    /*Criando as threads*/
    if (pthread_create(&thread1, NULL, counter1, &n) != 0)
    {
        printf("\nErro na criação da thread 1\n");
    }
    else
    {
        pthread_join(thread1, NULL);
    }
    /*Criação da thread subtratora*/
    if (pthread_create(&thread2, NULL, counter2, &n) != 0)
    {
        printf("\nErro na criação da thread 2\n");
    }
    else
    {
        pthread_join(thread2, NULL);
    }
    printf("\nQuantidade total de numeros primos = %d\n", nPrimos);
    return;
}

```

```

marcos@marcos-VirtualBox: ~/Área de Trabalho/SO - LISTA 2/parte 1
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$ dir
aloPaiFilho aloPaiFilho.c entrePrimos entrePrimos.c serieTaylor serieTaylor.c
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$ ./entrePrimos
Digite o numero N:6

=====NUMEROS_PRIMOS=====
[2] [3]
[5]
=====

Quantidade total de numeros primos = 3
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 1$

```

## 2. ESCALONAMENTO

O algoritmo em C de escalonamento contém 11 funções, incluindo a main, ele conta com uma função para criar novos processos vazios, uma função para limpar os processos através do comando free, outra para inserir os processos na lista de processos, outra função para inserir os processos em ordem no escalonamento SJF, para tornar uma lista linear em uma lista circular no escalonamento “round robin”, a função FIFO que irá iniciar os contadores dos processos em ordem de chegada, a função SJF que vai iniciar os contadores dos processos por ordem de tempo de execução, a função RR que irá iniciar os contadores dos processos por ordem de chegada com um determinado quantum(período máximo de tempo que um processo vai poder ficar executando por vez), uma função que irá imprimir todas as informações dos processos digitadas pelo usuário e o seu respectivo ID, uma função que irá fazer o switch dos escalonamentos desejados, e por último a função main que irá chamar as funções anteriores para realizar um fluxo de projeto para o usuário. Ademais, os processos foram definidos por uma struct com os campos id, tempo de execução total, ordem de chegada e o ponteiro para o próximo processo, assim como foi criada uma variável global que irá armazenar o número total de processos desejados pelos usuários.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <unistd.h>

int numProcessGlobal;

/*Estrutura de um processo */
struct process
{
    int id;      /* ID do processo */
    int time;    /* tempo de execucao*/
    int order;  /* ordem do processo */
    struct process *next;
};

typedef struct process Process;

/*FUNÇÃO QUE INICIALIZA O PROCESSO*/
Process *newProcess(void)
{
    return NULL;
}

/*FUNÇÃO QUE LIBERA O ESPAÇO ALOCADO PELO PROCESSO*/
void freeProcess(Process *process)
{

```

```

Process *processAux = process;
while (processAux != 0)
{
    Process *temp = processAux->next;
    free(processAux);
    processAux = temp;
}
}

/*FUNÇÃO QUE INSERE O PROCESSO NA LISTA*/
Process *insertProcess(Process *process, int id, int time, int order)
{
    Process *processPrev = NULL;
    Process *processAux = process;

    while (processAux != NULL && processAux->order < order)
    {
        processPrev = processAux;
        processAux = processAux->next;
    }

    /*Aloca um tamanho de espaço de memória para um novo processo*/
    Process *processNew = (Process *)malloc(sizeof(Process));
    if (processNew == NULL)
    {
        printf("ERRO!!\n");
        exit(1);
    }

    processNew->id = id;
    processNew->time = time;
    processNew->order = order;

    if (processPrev == NULL)
    {
        processNew->next = process;
        process = processNew;
    }
    else
    {
        processNew->next = processPrev->next;
        processPrev->next = processNew;
    }
}

```

```

    return process;
}
/*INSERE O PROCESSO EM ORDEM*/
Process *insertShortestProcessFirst(Process *process, int id, int time,
int order)
{
    Process *processPrev = NULL;
    Process *processAux = process;

    while (processAux != NULL && processAux->time < time)
    {
        processPrev = processAux;
        processAux = processAux->next;
    }

    /*Aloca um tamanho de espaço de memória para um novo processo*/
    Process *processNew = (Process *)malloc(sizeof(Process));
    if (processNew == NULL)
    {
        printf("Erro na alocação de memória.\n");
        exit(1);
    }

    processNew->id = id;
    processNew->time = time;
    processNew->order = order;

    if (processPrev == NULL)
    {
        processNew->next = process;
        process = processNew;
    }
    else
    {
        processNew->next = processPrev->next;
        processPrev->next = processNew;
    }

    return process;
}
/*FUNÇÃO QUE TORNA A LISTA CIRCULAR*/

```

```

Process *turnRound(Process *process)
{

    Process *processAux = process;
    Process *processAux2 = process;

    while (processAux2->next != NULL)
        processAux2 = processAux2->next;

    processAux2->next = processAux;

    return processAux;
}

/*FUNÇÃO QUE FAZ O ESCALONAMENTO FIFO*/
void FIFO(Process *process)
{

    Process *processAux = process;

    printf("\n===FIRST_IN_FIRST_OUT(FIFO)===\n");
    while (processAux != NULL)
    {
        printf("Processo %d executando [%d ut]\n", processAux->id,
processAux->time);
        sleep(processAux->time);
        processAux->time = 0;

        processAux = processAux->next;
    }
}

/*FUNÇÃO QUE FAZ O ESCALONAMENTO SJF*/
void SJF(Process *process)
{

    Process *processAux = process;          /*CRIA UM PROCESSO AUXILIAR A
PARTIR DO PARAMETRO RECEBIDO*/
    Process *shortests = newProcess(); /*CRIA UM PROCESSO VAZIO*/

    while (processAux != NULL)
    {

        shortests = insertShortestProcessFirst(
            shortests,
            processAux->id,

```

```

        processAux->time,
        processAux->order);

    processAux = processAux->next;
}

printf("===SHORTEST_JOB_FIRST(SJF)===\n");

for (Process *temp = shortestests; temp != NULL; temp = temp->next)
{

    printf("Processo %d executando... [%d ut]\n", temp->id,
temp->time);

    sleep(temp->time);
}

freeProcess(shortests);
freeProcess(process);
}

/*FUNÇÃO QUE FAZ O ESCALONAMENTO RR*/
void RR(Process *process, int quantum)
{
    int cont = 0;

    Process *processAux = turnRound(process);

    printf("\n===ROUND_ROBIN(RR)===\n");
    while (processAux != NULL)
    {

        if ((processAux->time != 0) && (processAux->time <= quantum))
        {
            cont = 0;

            printf("Processo %d executando [%d ut] final\n",
processAux->id, processAux->time);
            sleep(processAux->time);
            processAux->time = 0;
        }
        else if (processAux->time > quantum)
        {
            cont = 0;

            printf("Processo %d executando %dut. Tamanho: [%d ut]\n",
processAux->id, quantum, processAux->time);

```

```

        sleep(quantum);
        processAux->time -= quantum;
    }
    else
    {

        for (Process *i = process; process->time == 0; process =
process->next)
        {
            cont++;
            if (cont >= numProcessGlobal)
                exit(1);
        }

        processAux = processAux->next;
    }
}

/*FUNÇÃO QUE IMPRIME OS PROCESSOS*/
void printProcess(Process *process)
{
    printf("Processos digitados:\n");
    for (Process *temp = process; temp != NULL; temp = temp->next)
    {
        printf("=====\n");
        printf("ID:%d", temp->id);
        printf("\n");
        printf("TEMPO DE EXECUAÇÃO:%d", temp->time);
        printf("\n");
        printf("ORDEM:%d", temp->order);
        printf("\n");
        printf("=====");
        printf("\n");
    }
}

/*FUNÇÃO QUE ESCOLHE QUAL ESCALONAMENTO EXECUTAR*/
int switchOption(int option, Process *process)
{
    int quantum;
    switch (option)
    {
        case 1:
            FIFO(process);
            break;

```

```

    case 2:
        SJF(process);
        break;

    case 3:
        printf("Digite o quantum: ");
        scanf("%d", &quantum);
        RR(process, quantum);
        break;

    case 4:
        printf("APLICAÇÃO CANCELADA");
        break;

    default:
        printf("ERRO, OPÇÃO NÃO ENCONTRADA, DIGITE NOVAMENTE!!!!!!\n");
        printf("FIFO.....1\n");
        printf("SJF.....2\n");
        printf("RR.....3\n");
        printf("Cancelar_aplicação...4\n");
        printf("Digite o número da opção desejada: ");
        scanf("%d", &option);
        break;
}
return option;
}
/*FUNÇÃO PRINCIPAL*/
void main()
{
    setlocale(LC_ALL, "Portuguese");
    Process *process = newProcess(); // cria um novo processo
    int option;
    int numProcess, option2, time, order;
    printf("Quantidade de processos na fila: ");
    scanf("%d", &numProcess); // armazena a quantidade de procesos na fila
    printf("\n");
    numProcessGlobal = numProcess;
    for (int id = 1; id <= numProcess; id++)
    {
        printf("Ordem de chegada do processo %d: ", id);
        scanf("%d", &order);
        printf("Digite o tempo de execucao do processo %d: ", id);
        scanf("%d", &time);
        process = insertProcess(process, id, time, order);
    }
}

```



```

        printf("\n");
    }
    printProcess(process); // imprime os processos informados pelo usuário
    printf("\nFIFO.....1\n");
    printf("SJF.....2\n");
    printf("RR.....3\n");
    printf("Cancelar_aplicação...4\n");
    printf("Digite o número da opção desejada: ");
    scanf("%d", &option);
    if (option == 4)
    {
        return;
    }
    else
    {
        option = switchOption(option, process);
    }
    /*laço de repetição para o fluxo do programa*/
    option2 = 1;
    while (option2 == 1)
    {
        printf("\nSIM...1\n");
        printf("NÃO...0\n");
        printf("Deseja repetir a aplicação ?(1/0)\n");
        scanf("%d", &option2);
        /*Repetição do código da função main*/
        if (option2 == 1)
        {
            printf("Quantidade de processos na fila: ");
            scanf("%d", &numProcess); // armazena a quantidade de procesos
na fila

            printf("\n");
            numProcessGlobal = numProcess;
            for (int id = 1; id <= numProcess; id++)
            {
                printf("Ordem de chegada do processo %d: ", id);
                scanf("%d", &order);
                printf("Digite o tempo de execucao do processo %d: ", id);
                scanf("%d", &time);
                process = insertProcess(process, id, time, order);
                printf("\n");
            }
            printProcess(process);
            printf("\nFIFO.....1\n");

```

```

        printf("SJF.....2\n");
        printf("RR.....3\n");
        printf("Cancelar_aplicação...4\n");
        printf("Digite o número da opção desejada: ");
        scanf("%d", &option);
        if (option == 4)
        {
            return;
        }
        else
        {
            option = switchOption(option, process);
        }
    }
    else
    {
        option2 = 0;
    }
}

freeProcess(process); // limpa os processos da memória
return;
}

```

```

marcos@marcos-VirtualBox: ~/Área de Trabalho/SO - LISTA 2/parte 2
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 2$ dir
escalonamento_de_processos  escalonamento_de_processos.c
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 2$ ./escalonamento_de_processos
Quantidade de processos na fila: 2

Ordem de chegada do processo 1: 1
Digite o tempo de execucao do processo 1: 4

Ordem de chegada do processo 2: 2
Digite o tempo de execucao do processo 2: 6

Processos digitados:
=====
ID:1
TEMPO DE EXECUAO:4
ORDEM:1
=====
ID:2
TEMPO DE EXECUAO:6
ORDEM:2
=====

FIFO.....1
SJF.....2
RR.....3
Cancelar_aplicação...4
Digite o número da opção desejada: 1

===FIRST_IN_FIRST_OUT(FIFO)===
Processo 1 executando [4 ut]
Processo 2 executando [6 ut]

SIM...1
NÃO...0
Deseja repetir a aplicação ?(1/0)
0
marcos@marcos-VirtualBox:~/Área de Trabalho/SO - LISTA 2/parte 2$

```

### 3. SISTEMAS DE ARQUIVOS

#### 3.4 -

No item 4º, foi pedido a escrita de um programa que comece em um determinado diretório e percorra a árvore de arquivos a partir daquele ponto registrando os tamanhos de todos os arquivos que encontrar. Quando finalizada essa tarefa, era necessário imprimir o resultado da árvore na forma de um histograma. Para isso o código abaixo foi feito, tendo duas funções além da função Main, sendo elas `print_entrada` e `print_arvore_diretorios`.

A maior parte do código abaixo está em `print_entrada`, sendo que sua tarefa é imprimir (`print`) cada entrada de diretório. Já em `print_arvore_diretorios`, usa-se `nftw()` para chamá-lo para cada entrada de diretório que ele vê. Basicamente `prin_entrada` percorre os diretórios e imprime seus dados enquanto o `print_arvore_diretorios` monta a árvore de diretórios e ajuda a seguir o percurso por meio da função `nftw()`. O código é executado dentro da função `main`.

O POSIX.1-2008 padronizou a função `nftw()`, basicamente ela percorre a árvore de diretórios localizada sob o diretório `dirpath` e chama `fn()` uma vez para cada entrada na árvore. Por padrão, os diretórios são tratados antes dos arquivos e subdiretórios que eles contêm (percurso de pré-ordem).

```
#define _XOPEN_SOURCE 700
#define _LARGEFILE64_SOURCE
#define _FILE_OFFSET_BITS 64

#include <stdlib.h>
#include <unistd.h>
#include <ftw.h>
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#ifndef USE_FDS
#define USE_FDS 15
#endif

int print_entrada(const char *filepath, const struct stat *info,
                 const int typeflag, struct FTW *pathinfo)
{

    const double bytes = (double)info->st_size;

    if (bytes >= 1099511627776.0)
        printf(" %9.3f TiB", bytes / 1099511627776.0);
    else if (bytes >= 1073741824.0)
```

```

    printf(" %9.3f GiB", bytes / 1073741824.0);
else if (bytes >= 1048576.0)
    printf(" %9.3f MiB", bytes / 1048576.0);
else if (bytes >= 1024.0)
    printf(" %9.3f KiB", bytes / 1024.0);
else
    printf(" %9.0f B ", bytes);

if (typeflag == FTW_SL)
{
    char *target;
    size_t maxlen = 1023;
    ssize_t len;

    while (1)
    {

        target = malloc(maxlen + 1);
        if (target == NULL)
            return ENOMEM;

        len = readlink(filepath, target, maxlen);
        if (len == (ssize_t)-1)
        {
            const int saved_errno = errno;
            free(target);
            return saved_errno;
        }
        if (len >= (ssize_t)maxlen)
        {
            free(target);
            maxlen += 1024;
            continue;
        }

        target[len] = '\0';
        break;
    }

    printf(" %s -> %s\n", filepath, target);
    free(target);
}

else if (typeflag == FTW_SLN)
    printf(" %s (dangling symlink)\n", filepath);

```

```

    else if (typeflag == FTW_F)
        printf(" %s\n", filepath);
    else if (typeflag == FTW_D || typeflag == FTW_DP)
        printf(" %s/\n", filepath);
    else if (typeflag == FTW_DNR)
        printf(" %s/ (unreadable)\n", filepath);
    else
        printf(" %s (unknown)\n", filepath);

    return 0;
}

int print_arvore_diretorio(const char *const dirpath)
{
    int result;

    if (dirpath == NULL || *dirpath == '\\0')
        return errno = EINVAL;

    result = nftw(dirpath, print_entrada, USE_FDS, FTW_PHYS);
    if (result >= 0)
        errno = result;

    return errno;
}

int main(int argc, char *argv[])
{
    int arg;

    if (argc < 2)
    {
        if (print_arvore_diretorio("."))
        {
            fprintf(stderr, "%s.\n", strerror(errno));
            return EXIT_FAILURE;
        }
    }
    else
    {
        for (arg = 1; arg < argc; arg++)
        {

```

```

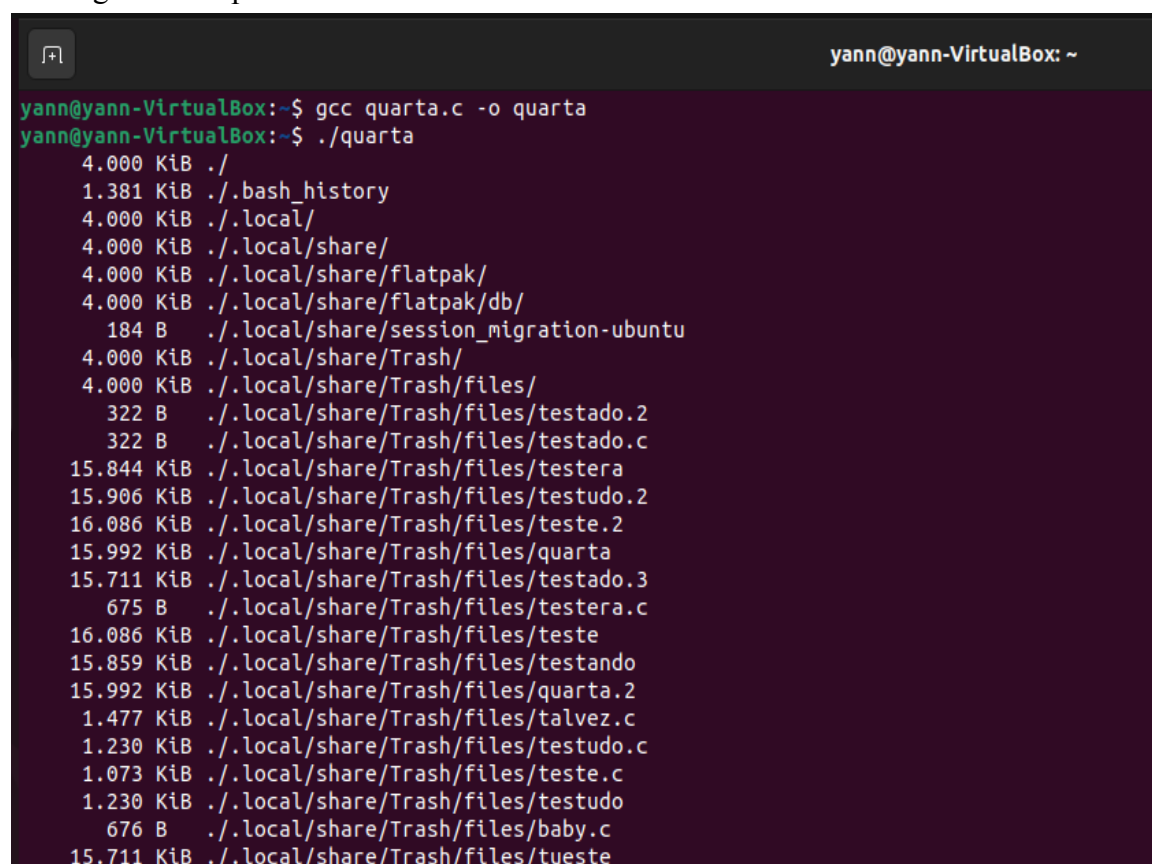
        if (print_arvore_diretorio(argv[arg]))
        {
            fprintf(stderr, "%s.\n", strerror(errno));
            return EXIT_FAILURE;
        }
    }

    return EXIT_SUCCESS;
}

```

**Executando no terminal:**

O código foi compilado e executado:



```

yann@yann-VirtualBox: ~
yann@yann-VirtualBox:~$ gcc quarta.c -o quarta
yann@yann-VirtualBox:~$ ./quarta
4.000 KiB ./
1.381 KiB ./bash_history
4.000 KiB ./local/
4.000 KiB ./local/share/
4.000 KiB ./local/share/flatpak/
4.000 KiB ./local/share/flatpak/db/
184 B ./local/share/session_migration-ubuntu
4.000 KiB ./local/share/Trash/
4.000 KiB ./local/share/Trash/files/
322 B ./local/share/Trash/files/testado.2
322 B ./local/share/Trash/files/testado.c
15.844 KiB ./local/share/Trash/files/testera
15.906 KiB ./local/share/Trash/files/testudo.2
16.086 KiB ./local/share/Trash/files/teste.2
15.992 KiB ./local/share/Trash/files/quarta
15.711 KiB ./local/share/Trash/files/testado.3
675 B ./local/share/Trash/files/testera.c
16.086 KiB ./local/share/Trash/files/teste
15.859 KiB ./local/share/Trash/files/testando
15.992 KiB ./local/share/Trash/files/quarta.2
1.477 KiB ./local/share/Trash/files/talvez.c
1.230 KiB ./local/share/Trash/files/testudo.c
1.073 KiB ./local/share/Trash/files/teste.c
1.230 KiB ./local/share/Trash/files/testudo
676 B ./local/share/Trash/files/baby.c
15.711 KiB ./local/share/Trash/files/tueste

```

### 3.5 -

O item 5 solicitou uma reconstrução do comando `ls` do linux/unix a partir da linguagem C; o Posix também foi utilizado.

O comando `ls` lista arquivos e diretórios no sistema de arquivos e mostra informações detalhadas sobre eles. É uma parte do pacote de utilitários do núcleo GNU que é instalado em todas as distribuições Linux.

No código abaixo, a função do comando `ls` foi mimetizada para a exibição de um ou dois diretórios. O código é bem simples, dividido em duas funções sendo elas a função `buscarDiretorios` e a função `main`, a função `buscarDiretorios` usa um ponteiro que

recebe o comando `opendir()` que é usado para abrir um diretório, em seguida há uma condicional para verificar se o diretório existe, caso ele não exista há um aviso de erro, caso ele exista a função entra num loop com `while` que percorre os diretórios e que opera o comando `readdir()` que lê os diretórios sugeridos pelo comando de input e também imprime esses diretórios. Ao final há um `closedir()`, comando que fecha o diretório. A função `main`, por sua vez, chama a função `buscarDiretorios` por meio de um loop.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

void buscarDiretorio(const char *path)
{
    DIR *d = opendir(path);

    struct dirent *dir;

    char nomeDoArquivo[256];

    if (d == NULL)
    {
        printf("ERRO!!! Nao foi possivel abrir o(s) diretorio(s)
escolhido(s) %s\n", path);
        return;
    }
    printf("Conteudo do Diretorio %s\n", path);

    while ((dir = readdir(d)) != NULL)
    {
        printf(">> %s\n", dir->d_name);
        if (nomeDoArquivo[0] == '.')
            continue;
        printf("%s\n", nomeDoArquivo);
    }
    closedir(d);
}

int main(int argc, char *argv[])
{
    int i;
    for (i = 1; i < argc; i++)
```

```
{  
    buscarDiretorio(argv[i]);  
}  
return 0;  
}
```

### Executando no terminal:

\*\*\*O comando deve seguir o padrão de chamada do comando ls como no exemplo abaixo:

O código foi compilado e executado no terminal do Ubuntu:

#### Foram abertos os diretórios home e trabalho

```
yann@yann-VirtualBox:~$ gcc buscar.c -o buscar  
yann@yann-VirtualBox:~$ ./buscar /home/yann/ Trabalho  
Conteudo do Diretorio /home/yann/  
>> .bash_history  
  
>> .local  
  
>> quarta  
  
>> Documentos  
  
>> Música  
  
>> buscar  
  
>> .bash_logout  
  
>> .  
  
>> snap  
  
>> Trabalho  
  
>> .config  
  
>> Vídeos  
  
>> .bashrc  
  
>> .cache  
  
>> .sudo_as_admin_successful  
  
>> Área de Trabalho  
  
>> quarta.c  
  
>> Downloads  
  
>> .mozilla  
  
>> .profile  
  
>> Público
```



```
>> ..  
>> .thunderbird  
>> buscar.c  
>> Modelos  
>> Imagens  
Conteudo do Diretorio Trabalho  
>> copia.c  
>> gigante.c  
>> .  
>> testudo.c  
>> testudo  
>> copia  
>> ..  
yann@yann-VirtualBox:~$
```

LINK DO REPOSITÓRIO NO GITHUB:

<https://github.com/MarcosVini9999/threadsEscalonamentoSistemaDeArquivos>