

Universidade Federal De Uberlândia
Faculdade de Computação
Sistemas Operacionais
GSI018

EULLER HENRIQUE BANDEIRA OLIVEIRA
11821BSI210

Relatório :
Atividade Prática de Fixação
(Unidade 2)

Uberlândia
2021

Computadores utilizados:

Computador utilizado na questão 1,2,3 e 4: Notebook Samsung NP300E4L-KW1BR, PROCESSADOR: i3 (sexta geração) , CPUS: 4, RAM: 4gb, HD: 1tb,SO: Linux Mint 19.3 Cinnamon

Computador utilizado na questão 5: Notebook Lenovo Ideapad S145, PROCESSADOR: i7 (oitava geração), CPUS: 8, RAM: 20gb, SSD: 256gb, HD: 1tb, PLACA DE VÍDEO: Nvidia MX110, SO1: Windows 10 Home, SO2: Ubuntu 20.04.2.0 LTS, SO UTILIZADO: SO2

1. Utilizando dois compiladores diferentes de linguagem C, produza o código assembly para o programa abaixo. Analise as diferenças de cada código assembly produzido em comparação ao código C. A plataforma de SO é de livre escolha.

Q1.C

```
#include <stdio.h>
int i=3;
int j;
int main(){
    int w;
    int z=3;
    printf("Hello World!\n");
    printf("%d%d%d%d",i,j,w,z);
}
```

GCC

```
sudo apt-get install gcc
gcc -S Q1.c
```

.file	"Q1.c"	.cfi_def_cfa_register 6
.text		subq \$16, %rsp
.globl i		movl \$3, -8(%rbp)
.data		leaq .LC0(%rip), %rdi
.align 4		call puts@PLT
.type i, @object		movl j(%rip), %edx
.size i, 4		movl i(%rip), %eax
i:		movl -8(%rbp), %esi
	.long 3	movl -4(%rbp), %ecx
	.comm j,4,4	movl %esi, %r8d
	.section .rodata	movl %eax, %esi
.LC0:	.string "Hello World!"	leaq .LC1(%rip), %rdi
.LC1:		movl \$0, %eax
	.string "%d%d%d%d"	call printf@PLT
	.text	movl \$0, %eax
	.globl main	leave
	.type main, @function	.cfi_def_cfa 7, 8
main:		ret
.LFB0:		.cfi_endproc
	.cfi_startproc	.LFE0:
	pushq %rbp	.size main, .-main
	.cfi_def_cfa_offset 16	.ident "GCC: (Ubuntu
	.cfi_offset 6, -16	7.5.0-3ubuntu1~18.04) 7.5.0"
	movq %rsp, %rbp	.section
		.note.GNU-stack,"",@progbits

CLANG/LLVM
 sudo apt-get install clang
 clang -S Q1.c

```

.text
.file "Q1.c"
.globl main # -- Begin function main
.p2align 4, 0x90
.type main,@function
main: # @main
.cfi_startproc
# %bb.0:
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset %rbp, -16
movq %rsp, %rbp
.cfi_def_cfa_register %rbp
subq $16, %rsp
movabsq $.L.str, %rdi
movl $3, -8(%rbp)
movb $0, %al
callq printf
movabsq $.L.str.1, %rdi
movl i, %esi
movl j, %edx
movl -4(%rbp), %ecx
movl -8(%rbp), %r8d
movl %eax, -12(%rbp) # 4-byte Spill
movb $0, %al
callq printf
xorl %ecx, %ecx
movl %eax, -16(%rbp) # 4-byte Spill
movl %ecx, %eax
addq $16, %rsp
popq %rbp
retq
.Lfunc_end0:
.size main, .Lfunc_end0-main
.cfi_endproc
# -- End function
.type i,@object # @i
.data
.globl i
.p2align 2
i:
.long 3 # 0x3
.size i, 4
.type .L.str,@object # @.str
.section .rodata.str1.1,"aMS",@progbits,1
.L.str:
.asciz "Hello World!\n"
.size .L.str, 14
.type .L.str.1,@object # @.str.1
.L.str.1:
.asciz "%d%d%d%d"
.size .L.str.1, 9
.type j,@object # @j
.comm j,4,4
.ident "clang version 6.0.0-1ubuntu2
(tags/RELEASE_600/final)"
.section ".note.GNU-stack","",@progbits

```

Diferenças: O código em assembly gerado pelo CLANG possui mais instruções do que o código em assembly gerado pelo GCC.

O código em assembly gerado pelo GCC possui mais instruções do que o código em c.

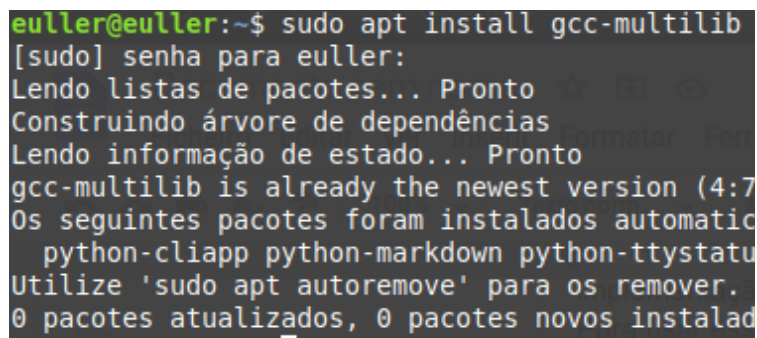
2. A partir do código-fonte abaixo, gere o programa executável, execute-o e anote a senha que será exibida na tela. Observe que a rotina `passcode()` não faz parte do programa abaixo e nem da linguagem C. Sua implementação está disponível no arquivo objeto denominado “`passcode.o`” que acompanha esta lista. Para usar essa rotina é necessário incluir o arquivo de cabeçalho “`passcode.h`” que acompanha o “`passcode.o`”. O arquivo “`passcode.o`” foi criado para funcionar apenas no sistema operacional Linux.

```
#include <stdio.h>
#include "passcode.h"
int main(){
    char code[11];
    passcode(code);
    printf("%s",code);
}
```

Passo 1: Instale o `gcc-multilib`

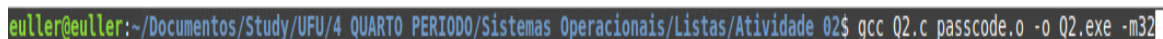
O `gcc-multilib` permite que um código `.o` que foi gerado por um computador com 32 bits seja transformado em um código `.exe` em um computador com 64 bits.

`sudo apt install gcc-multilib`



```
euller@euller:~$ sudo apt install gcc-multilib
[sudo] senha para euller:
Lendo listas de pacotes... Pronto
Construindo árvore de dependências
Lendo informação de estado... Pronto
gcc-multilib is already the newest version (4:7
Os seguintes pacotes foram instalados automaticamente:
python-cliapp python-markdown python-ttystatu
Utilize 'sudo apt autoremove' para os remover.
0 pacotes atualizados, 0 pacotes novos instalados
```

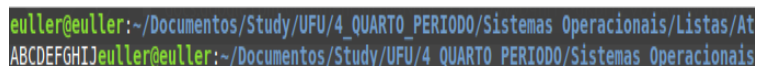
`gcc Q2.c passcode.o -o Q2.exe -m32`



```
euller@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02$ gcc Q2.c passcode.o -o Q2.exe -m32
```

`./Q2.exe`

ABCDEFGHIJ



```
euller@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02$ ./Q2.exe
ABCDEFGHIJ
```

3. O arquivo de programa “prog01.exe” (no diretório desta lista) possui três funções, main(), f1() e f2(), cujo código fonte em C está listado abaixo. Faça alterações diretamente no arquivo de programa (prog01.exe) para que ao ser executado a função main() chame primeiro f2() e depois f1(). O arquivo de programa prog01.exe foi criado para funcionar apenas no sistema operacional Linux.

```
#include <stdio.h>
```

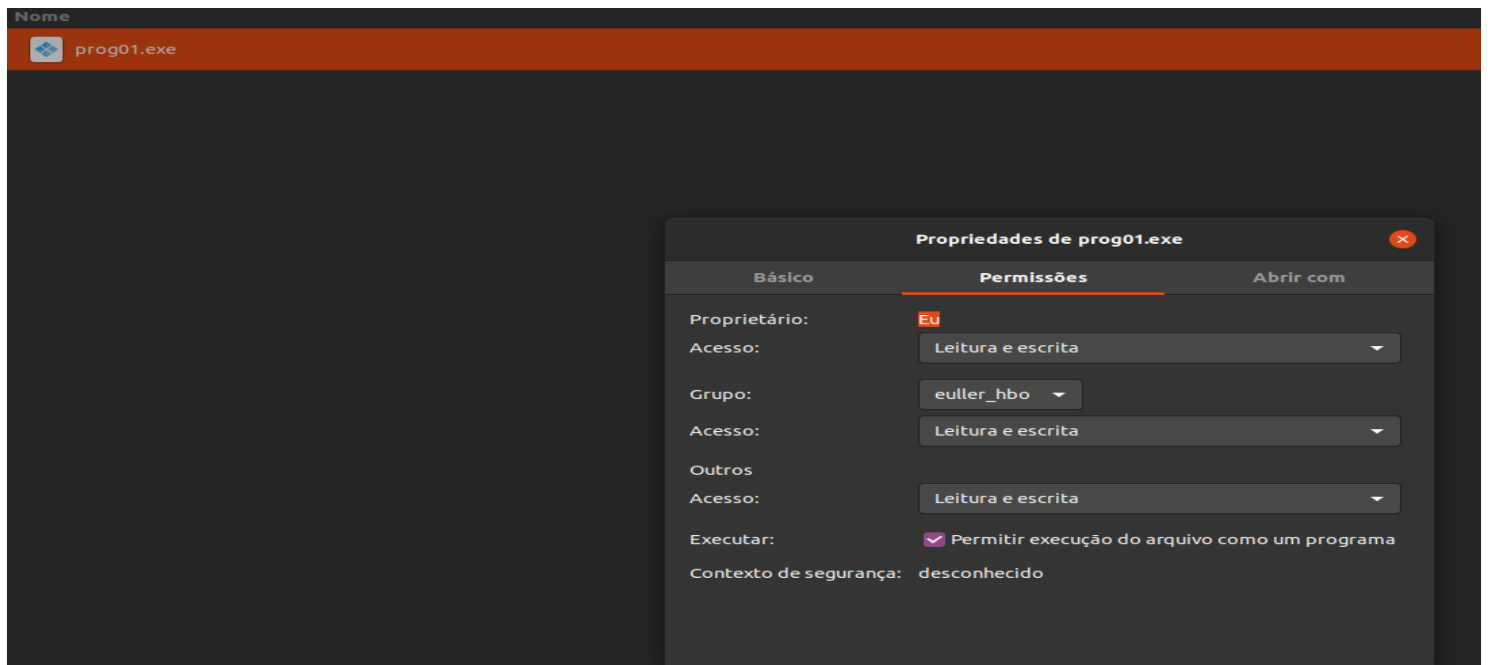
```
int f1(){  
    printf("F1");  
}
```

```
int f2(){  
    printf("F2");  
}
```

```
int main(){  
    f1();  
    f2();  
}
```

Primeiro passo:

Conceda todos as permissões para o arquivo prog01.exe



Segundo passo:

Transforme o o.exe em .s utilizando o comando objdump -S prog01.exe

```
euller@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q3$ objdump -S prog01.exe
```

Terceiro passo:

Localize a main

08048228 <main>:
8048228: 8d 4c 24 04 lea 0x4(%esp),%ecx
804822c: 83 e4 f0 and \$0xffffffff0,%esp
804822f: ff 71 fc pushl -0x4(%ecx)
8048232: 55 push %ebp
8048233: 89 e5 mov %esp,%ebp
8048235: 51 push %ecx
8048236: 83 ec 04 sub \$0x4,%esp
8048239: e8 0e 00 00 00 call 804824c <f1>
804823e: e8 21 00 00 00 call 8048264 <f2>
8048243: 83 c4 04 add \$0x4,%esp
8048246: 59 pop %ecx
8048247: 5d pop %ebp
8048248: 8d 61 fc lea -0x4(%ecx),%esp
804824b: c3 ret

Localizar

Q <main>

⌕ ⬆ ⬆

☐ Diferenciar maiúsculas/minúsculas

☐ Coincidir apenas com palavra completa

☐ Coincidir como expressão regular

☒ Voltar ao início

Quarto passo:

Na main, localize o endereço posterior à primeira função call e localize o endereço posterior à segunda função call

P call1: 804823e

08048228 <main>:
8048228: 8d 4c 24 04 lea 0x4(%esp),%ecx
804822c: 83 e4 f0 and \$0xffffffff0,%esp
804822f: ff 71 fc pushl -0x4(%ecx)
8048232: 55 push %ebp
8048233: 89 e5 mov %esp,%ebp
8048235: 51 push %ecx
8048236: 83 ec 04 sub \$0x4,%esp
8048239: e8 0e 00 00 00 call 804824c <f1>
804823e: e8 21 00 00 00 call 8048264 <f2>
8048243: 83 c4 04 add \$0x4,%esp
8048246: 59 pop %ecx
8048247: 5d pop %ebp
8048248: 8d 61 fc lea -0x4(%ecx),%esp
804824b: c3 ret

P call2: 8048243

08048228 <main>:
8048228: 8d 4c 24 04 lea 0x4(%esp),%ecx
804822c: 83 e4 f0 and \$0xffffffff0,%esp
804822f: ff 71 fc pushl -0x4(%ecx)
8048232: 55 push %ebp
8048233: 89 e5 mov %esp,%ebp
8048235: 51 push %ecx
8048236: 83 ec 04 sub \$0x4,%esp
8048239: e8 0e 00 00 00 call 804824c <f1>
804823e: e8 21 00 00 00 call 8048264 <f2>
8048243: 83 c4 04 add \$0x4,%esp
8048246: 59 pop %ecx
8048247: 5d pop %ebp
8048248: 8d 61 fc lea -0x4(%ecx),%esp
804824b: c3 ret

Quinto passo:

Faça a conversão de hexadecimal para decimal

P call1: 804823e -> 134513214

P call2: 8048243 -> 134513219

Sexto passo:

Localize o endereço da função f1 e f2

f1: 804824c -> 134513228

```
0804824c <f1>:
804824c: 55          push    %ebp
804824d: 89 e5       mov     %esp,%ebp
804824f: 83 ec 08    sub     $0x8,%esp
8048252: 83 ec 0c    sub     $0xc,%esp
8048255: 68 48 2a 0b 08 push   $0x80b2a48
804825a: e8 31 0f 00 00 call    8049190 <_IO_puts>
804825f: 83 c4 10    add     $0x10,%esp
8048262: c9         leave  %ebp
8048263: c3         ret
```

f2: 8048264 -> 134513252

```
08048264 <f2>:
8048264: 55          push    %ebp
8048265: 89 e5       mov     %esp,%ebp
8048267: 83 ec 08    sub     $0x8,%esp
804826a: 83 ec 0c    sub     $0xc,%esp
804826d: 68 4b 2a 0b 08 push   $0x80b2a4b
8048272: e8 19 0f 00 00 call    8049190 <_IO_puts>
8048277: 83 c4 10    add     $0x10,%esp
804827a: c9         leave  %ebp
804827b: c3         ret
804827c: 90         nop
804827d: 90         nop
804827e: 90         nop
804827f: 90         nop
```

Sétimo passo:

Faça a conversão de hexadecimal para decimal do endereço da função f1 e f2

f1: 804824c -> 134513228

f2: 8048264 -> 134513252

Oitavo passo:

Subtraia o endereço da função f1 pelo endereço da função posterior ao segundo call

$134513228 - 134513214 = 9$

Faça a conversão de decimal para hexadecimal

9 -> 9

Nono passo:

Subtraia o endereço da função f2 pelo endereço da função posterior ao primeiro call

$134513252 - 134513214 = 38$

Faça a conversão de decimal para hexadecimal

38 -> 26

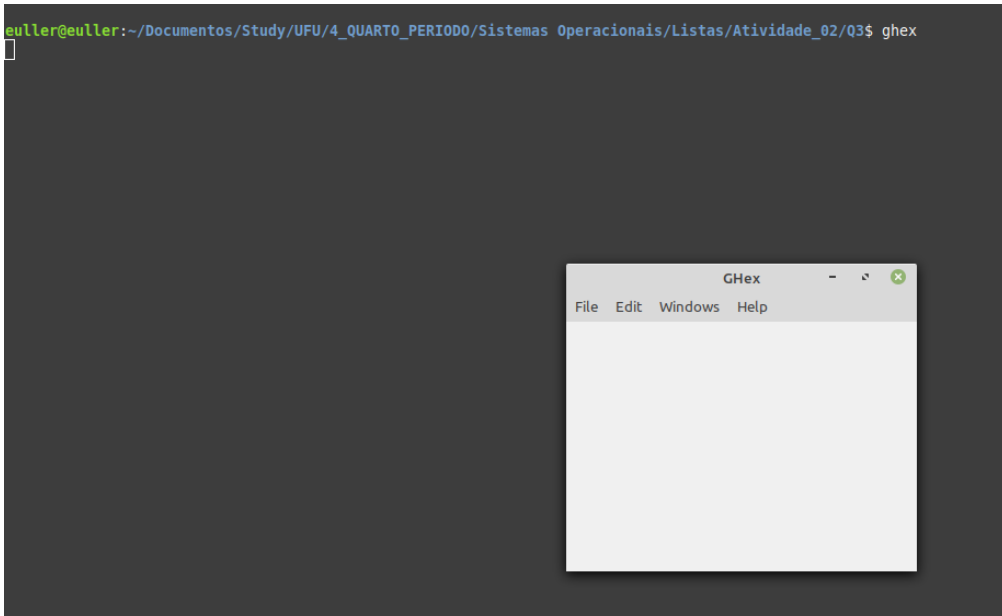
Décimo passo:

Instale o ghex por meio do comando sudo apt-get install ghex

```
euller@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q3$ sudo apt-get install ghex
[sudo] senha para euller:
Lendo listas de pacotes... Pronto
Construindo árvore de dependências
Lendo informação de estado... Pronto
ghex is already the newest version (3.18.3-3).
Os seguintes pacotes foram instalados automaticamente e já não são necessários:
 python-cliapp python-markdown python-ttstatus python-yaml
Utilize 'sudo apt autoremove' para os remover.
0 pacotes atualizados, 0 pacotes novos instalados, 0 a serem removidos e 65 não atualizados.
```

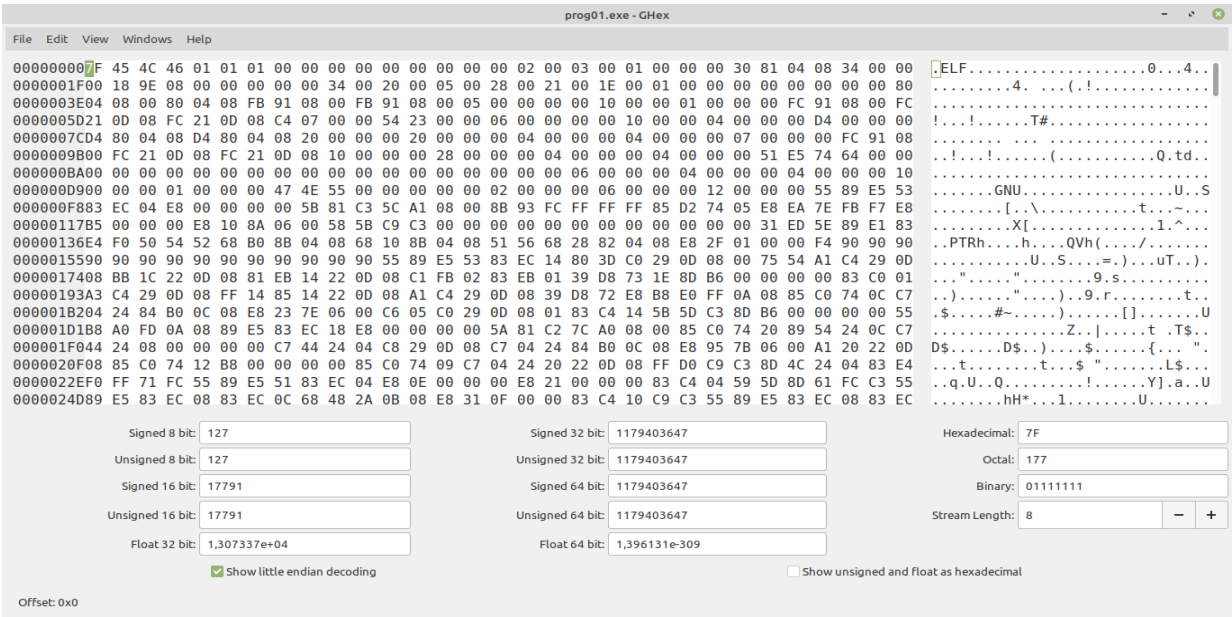
Décimo primeiro passo:

Execute o ghex



Décimo segundo passo:

Abra o arquivo prog01.exe no ghex



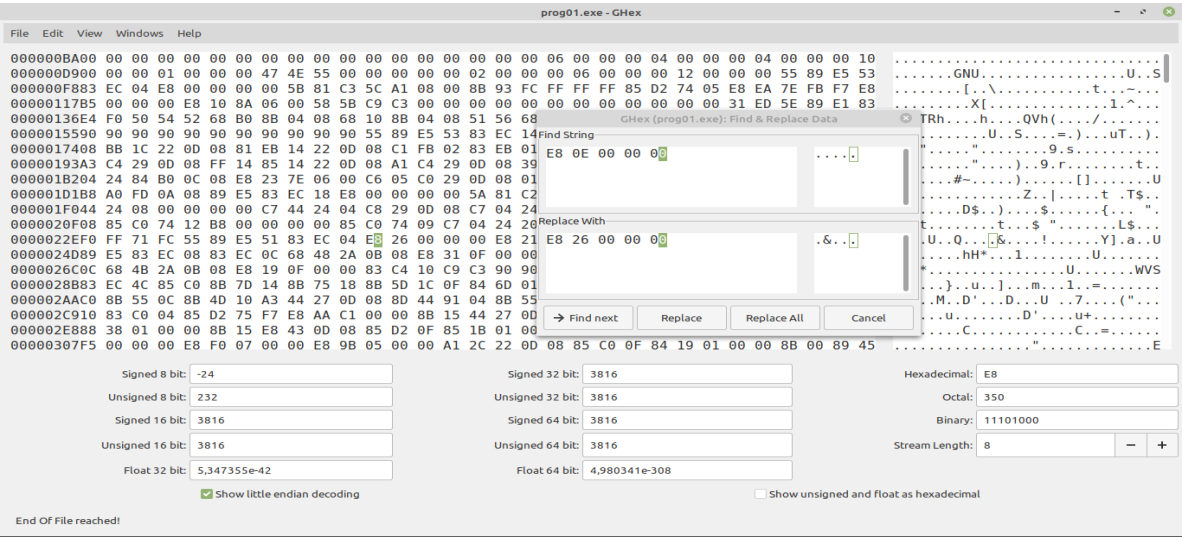
Décimo terceiro passo:

Localize o código do primeiro call

e8 0e 00 00 00

Troque o segundo campo pela subtração realizada entre o endereço da função f2 e o endereço da chamada posterior à primeira função call

e8 26 00 00 00



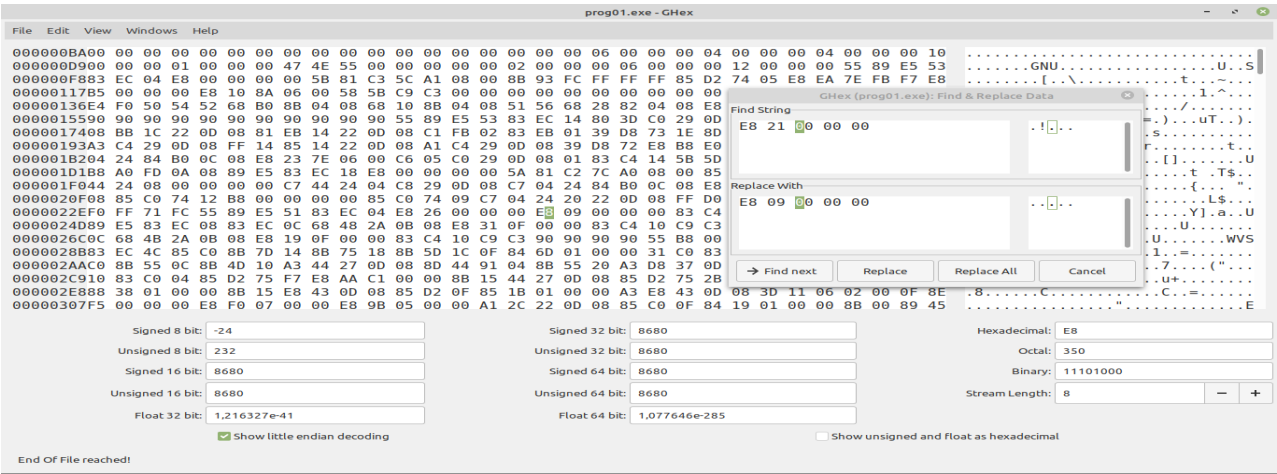
Décimo quarto passo:

Localize o código do segundo call

e8 21 00 00 00

Troque o segundo campo pela subtração realizada entre o endereço da função f1 e o endereço posterior ao primeiro call

e8 09 00 00 00



Décimo quinto passo:

Salve o arquivo

Décimo sexto passo:

Execute o prog01.exe

./prog01.exe

```
euller@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q3$ ./prog01.exe
F2
F1
```

Obs: primeiro_campo -> Código da função assembly
segundo_campo -> Deslocamento de bits

4. A partir do código-fonte abaixo, crie os arquivos de programas “prog02.exe” e “prog03.exe”. Execute cada programa comparando seus tempos de execução. Para isso, utilize o comando “time” (Linux) ou “PowerShell Measure-Command” (Windows). Abaixo exemplos de utilização.

```
/* prog02.c */
```

```
#include <stdio.h>
int main(){
    int i;
    for(i=0;i<10000;i++)
        printf("A\n");
}
```

```
/* prog03.c */
```

```
#include <stdio.h>
#include<string.h>

int main(){
    int i;

    char str[20001]="";

    for(i=0;i<10000;i+=2){
        *(str+i)='A';
        *(str+i+1)='\n';
    }

    *(str+i)='\0';
    printf("%s",str);
}
```

No linux:

```
time prog02.exe
time prog03.exe
```

Obs: Execute 11 vezes cada programa, descarte a primeira execução de cada programa, e tire a média dos demais 10 resultados de cada programa para ter um valor aproximado dos tempos de execução de cada um.

gcc prog02.c -o prog02.exe

1. time ./prog02.exe
 - a. real 0m0,078s
 - b. user 0m0,027s
 - c. sys 0m0,050s
2. time ./prog02.exe
 - a. real 0m0,113s
 - b. user 0m0,018s
 - c. sys 0m0,091s
3. time ./prog02.exe
 - a. real 0m0,090s
 - b. user 0m0,016s
 - c. sys 0m0,074s

4. time ./prog02.exe
 - a. real 0m0,105s
 - b. user 0m0,012s
 - c. sys 0m0,093s
5. time ./prog02.exe
 - a. real 0m0,062s
 - b. user 0m0,008s
 - c. sys 0m0,054s
6. time ./prog02.exe
 - a. real 0m0,111s
 - b. user 0m0,020s
 - c. sys 0m0,092s
7. time ./prog02.exe
 - a. real 0m0,113s
 - b. user 0m0,009s
 - c. sys 0m0,103s

8. time ./prog02.exe
 - a. real 0m0,109s
 - b. user 0m0,008s
 - c. sys 0m0,101s
9. time ./prog02.exe
 - a. real 0m0,095s
 - b. user 0m0,017s
 - c. sys 0m0,078s
10. time ./prog02.exe
 - a. real 0m0,103s
 - b. user 0m0,017s
 - c. sys 0m0,086s
11. time ./prog02.exe
 - a. real 0m0,110s
 - b. user 0m0,013s
 - c. sys 0m0,097s

Média

- a. real 0.08103s
- b. user 0.0138s
- c. sys 0.0869s

gcc prog03.c -o prog03.exe

1. time ./prog03.exe
 - a. real 0m0,011s
 - b. user 0m0,001s
 - c. sys 0m0,010s
2. time ./prog03.exe
 - a. real 0m0,009s
 - b. user 0m0,001s
 - c. sys 0m0,008s
3. time ./prog03.exe
 - a. real 0m0,009s
 - b. user 0m0,001s
 - c. sys 0m0,008s

4. time ./prog03.exe
 - a. real 0m0,008s
 - b. user 0m0,001s
 - c. sys 0m0,008s
5. time ./prog03.exe
 - a. real 0m0,015s
 - b. user 0m0,005s
 - c. sys 0m0,011s
6. time ./prog03.exe
 - a. real 0m0,018s
 - b. user 0m0,000s
 - c. sys 0m0,018s
7. time ./prog03.exe
 - a. real 0m0,008s
 - b. user 0m0,001s
 - c. sys 0m0,008s

8. time ./prog03.exe
 - a. real 0m0,008s
 - b. user 0m0,001s
 - c. sys 0m0,008s
9. time ./prog03.exe
 - a. real 0m0,016s
 - b. user 0m0,001s
 - c. sys 0m0,015s
10. time ./prog03.exe
 - a. real 0m0,015s
 - b. user 0m0,004s
 - c. sys 0m0,011s

11. time ./prog03.exe
 - a. real 0m0,008s
 - b. user 0m0,004s
 - c. sys 0m0,004s

Média

- a. real 0m0,0114s
- b. user 0m0,0019s
- c. sys 0m0,0099s

5. Fazer um programa, em linguagem C, para contar e imprimir o número total de arquivos armazenados em um disco rígido. Implementar e comparar o tempo de execução de três versões desse programa. A primeira versão deve ser programada como um único processo singlthreaded. A segunda versão deve ser programada como múltiplos processos singlthreaded, onde o número de processos (n) deve corresponder ao número de processadores do computador. Caso o computador tenha apenas um processador, então utilize $n = 2$. A terceira versão deve ser programada como múltiplos processos, tal como a segunda versão, contudo cada processo deve utilizar múltiplas threads (mt). O valor de mt deve ser 2. Na segunda e terceira versões, o algoritmo de busca e contagem de arquivos deve ser paralelizado; por exemplo, enquanto um processo conta os arquivos em uma parte do disco (ex. C:\ no Windows ou /dev/sda1 no Linux) o outro processo conta os arquivos em outra parte (ex. D:\ ou /dev/sda2). O mesmo aplica-se para múltiplas threads. A estratégia de paralelização do algoritmo de contagem de arquivos é de livre escolha, assim como a plataforma de SO escolhida para realizar esse exercício.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

int DIRENT_01(char caminho_pai[]){

    int qtd = 0;

    //O ponteiro dir do tipo DIR será utilizado para armazenar o endereço do
    diretório desejado
    DIR *dir;

    //Esse ponteiro será utilizado para armazenar o endereço de uma struct dirent.
    //Tal struct é utilizada para armazenar os dados de um arquivo.
    struct dirent *lsdir;

    /* A struct dirent foi definida da seguinte maneira na biblioteca dirent

    struct dirent {
        ino_t          d_ino;
        off_t          d_off;
        unsigned short d_reclen;    // Tamanho do arquivo
        unsigned char  d_type;      // Tipo do arquivo
        char           d_name[256]; // Nome do arquivo
    };

    */
```

```

// A função opendir abre o diretório determinado
// Tal função retorna o endereço do diretório
dir = opendir(caminho_pai);
if(dir == NULL){
    return 0;
}
//A função readdir é utilizada para ler arquivos do diretório determinado
//Tal função possui como parâmetro um ponteiro que receberá o endereço do
diretório que será lido.
//Tal função retorna o endereço da struct dirent do arquivo atual presente no
diretório determinado
//A cada iteração, o próximo arquivo é lido

while((lsdir = readdir(dir)) != NULL){

    char nome[256];

    //O nome do diretório é acessado por meio do campo d_name da struct dirent
    strcpy(nome, lsdir->d_name);

    if(strcmp(nome, ".") != 0 && strcmp(nome, "..") != 0){

        char caminho_filho[1000];

        strcpy(caminho_filho, caminho_pai);

        if(strcmp(caminho_filho, "/") != 0){
            strcat(caminho_filho, "/");
        }

        strcat(caminho_filho, nome);

        qtd+=1;
        //printf("Arquivo: %s\n", caminho_filho);

        //If utilizado para um atalho não ser acessado
        if(lsdir->d_type == 4){

            qtd+=DIRENT_01(caminho_filho);

        }

    }

}

closedir(dir);
return qtd;
}

```

```

/*

    Fazer um programa, em linguagem C, para contar e imprimir o número total de arquivos
    armazenados em
    um disco rígido. Implementar e comparar o tempo de execução de três versões desse
    programa.

    A primeira versão deve ser programada como um único processo singlthreaded.

*/

void CONT_01(){

    int qtd_total = DIRENT_01("/");
    printf("\nQTD_TOTAL: %d\n", qtd_total);
}

/*

    A segunda versão deve ser programada como múltiplos processos singlthreaded, onde o
    número de processos (n) deve corresponder
    ao número de processadores do computador. Caso o computador tenha apenas um
    processador, então utilize n = 2.

*/

#include <sys/wait.h>
#include <sys/sysinfo.h>
#include <unistd.h>
#include <math.h>

char diretorio_raiz[100][100];
int n_p;
int qtd_raiz;

void DIRENT_02(){

    DIR *dir;
    struct dirent *lsdir;
    dir = opendir("/");
    int i = 0;
    while((lsdir = readdir(dir)) != NULL){

        char nome[256];

        //O nome do diretório é acessado por meio do campo d_name da struct dirent
        strcpy(nome, lsdir->d_name);
    }
}

```



```

        if(strcmp(nome, ".") != 0 && strcmp(nome, "..") != 0){

            strcpy(diretorio_raiz[i], "/");
            strcat(diretorio_raiz[i], nome);

            i++;
            qtd_raiz++;
        }

    }

    closedir(dir);
}

```

```

void F_01(int n, double i, double j){

```

```

    pid_t pid;
    int status=0;
    int qtd_diretorio = 0;
    FILE *file;

    if(n > 0){

        pid = fork(); //O programa se divide e começa a ser executado ao mesmo tempo em
n_p processos

        printf("fork: %d      ->      ", pid);

        if(pid > 0){

            //Pai

            wait(&status);

            printf("PID PAI: %d      ->->      ", getpid());
            printf("f(n-1)\n");

            F_01(n-1, j, ceil(j+(double)qtd_raiz/(double)n_p));

        }else if (pid == 0){

            //Filho

            printf("PID FILHO: %d      ->->      ", getpid());
            printf("PROCESSO %d-> i: %lf, j: %lf\n\n", n, i, j);

```

```

        for(; i < j ; i++){

            printf("%s\n",diretorio_raiz[(int)i]);
            qtd_diretorio+=DIRENT_01(diretorio_raiz[(int)i]);
        }

        file = fopen("qtd.txt", "a");
        fprintf(file,"%d\n",qtd_diretorio);
        fclose(file);

        exit(status);

    }

}

}else{

    int qtd_final = 0;
    int i = 0;
    file = fopen("qtd.txt","r");
    while(feof(file) == 0){

        fscanf(file,"%d\n",&qtd_diretorio);
        printf("\nPROCESSO %d: Qtd anterior: %d + Qtd atual %d", i+1, qtd_final,
qtd_diretorio);
        qtd_final+=qtd_diretorio;
        printf(" = %d\n\n", qtd_final);
        i++;

    }

    printf("QTD_TOTAL: %d\n", qtd_final);

    file = fopen("qtd.txt","w");
    fprintf(file,"%s", "");

    fclose(file);

}

}

```

```

void CONT_02() {

    qtd_raiz = 0;

    DIRENT_02();

    //Número de processadores == Número de processos

    n_p = get_nprocs();

    F_01(n_p, 0, ceil((double)qtd_raiz/(double)n_p));

}

/*
    A terceira versão deve ser programada como múltiplos processos, tal como a segunda
    versão,
    contudo cada processo deve utilizar múltiplas threads (mt). O valor de mt deve ser 2.

    Na segunda e terceira versões, o algoritmo de busca e contagem de arquivos deve ser
    paralelizado; por exemplo,
    enquanto um processo conta os arquivos em uma parte do disco (ex. C:\ no Windows ou
    /dev/sda1 no
    Linux) o outro processo conta os arquivos em outra parte (ex. D:\ ou /dev/sda2).
    O mesmo aplica-se para múltiplas threads.

    A estratégia de paralelização do algoritmo de contagem de arquivos é de livre
    escolha,
    assim como a plataforma de SO escolhida para realizar esse exercício.

*/

#include <pthread.h>

typedef struct {
    int i1;
    int j1;

    int i2;
    int j2;
} INTERVALO;

int cont = 1;

```

```

void *THREAD_FUNCTION(void *args){

    INTERVALO *intervalo = args;
    int i = 0;
    int j = 0;

    printf("THREAD -> \n");

    if(cont == 1){

        i = intervalo->i1;
        j = intervalo->j1;
        cont++;

    }else if(cont == 2){

        i = intervalo->i2;
        j = intervalo->j2;

        cont = 0;

    }

    int qtd_diretorio = 0;

    printf("\n");

    for(; i < j; i++){

        printf(" i: %d j: %d\n",i,j);
        qtd_diretorio+=DIRENT_01(diretorio_raiz[i]);
        printf("%s\n",diretorio_raiz[i]);

    }

    printf("\n");

    FILE *file = fopen("qtd.txt", "a");
    fprintf(file,"%d\n",qtd_diretorio);
    fclose(file);

    pthread_exit(NULL);
}

```

```

void F_02(int n, double i, double j){

    pid_t pid;
    int status=0;
    int qtd_diretorio = 0;

    if(n > 0){

        pid = fork(); //O programa se divide e começa a ser executado ao mesmo tempo em n
processos

        printf("fork: %d      ->      ", pid);

        if(pid > 0){

            //Pai

            wait(&status);
            printf("PID PAI: %d      ->->      ", getpid());
            printf("f(n-1)\n");

            F_02(n-1, j, ceil(j+(double)qtd_raiz/(double)n_p));

        }else if (pid == 0){

            //Filho

            printf("PID FILHO: %d      ->->      ", getpid());
            printf("PROCESSO %d-> i: %f, j: %f\n\n", n, i, j);

            pthread_t thread_id[2];
            INTERVALO* intervalo = malloc(sizeof *intervalo);

            intervalo->i1 = i;
            intervalo->j1 = j-2;

            intervalo->i2 = j-2;
            intervalo->j2 = j;

            for(int x = 0; x < 2; x++){

                if(pthread_create(&thread_id[x], NULL, THREAD_FUNCTION, intervalo)){
                    free(intervalo);
                }

            }

        }

    }
}

```

```
printf("\n");

for(int y = 0; y < 2; y++){

    pthread_join(thread_id[y], NULL);

}

exit(status);

}

}else{

    int qtd_final = 0;
    int i = 0;
    FILE *file = fopen("qtd.txt","r");
    while(feof(file) == 0){

        fscanf(file,"%d\n",&qtd_diretorio);

        printf("\nP||T: %d -> Qtd atual: %d + Qtd anterior %d", i+1, qtd_diretorio,
qtd_final);

        qtd_final+=qtd_diretorio;
        printf(" = %d\n\n", qtd_final);
        i++;

    }

    printf("QTD_TOTAL: %d\n", qtd_final);

    file = fopen("qtd.txt","w");
    fprintf(file,"%s", "");

    fclose(file);

}

}
```

```

int CONT_03(){

    qtd_raiz = 0;

    DIRENT_02();

    //Número de processadores == Número de processos

    n_p = get_nprocs();

    F_02(n_p, 0, ceil((double) qtd_raiz/(double)n_p));

}

int main(void){

    int op;

    printf("LEITOR E CONTADOR DE ARQUIVOS\n");
    printf("Autor: Euller Henrique Bandeira Oliveira \n");
    printf("Matrícula: 11821BSI210\n");
    printf("Período: 5º\n");
    printf("Curso: Sistemas De Informação\n");
    printf("Universidade: Universidade Federal De Uberlândia\n\n");

    printf("Você deseja ler e contar os arquivos do seu computador de que maneira?\n\n");
    printf("0 - Nenhum\n");
    printf("1 - Processo singlthreaded.\n");
    printf("2 - Múltiplos processos singlthreaded\n");
    printf("3 - Múltiplos processos multithreaded\n");
    while(1){
        printf("\n>>> ");
        scanf("%d", &op);
        if(op == 0){
            return 0;
        }else if(op ==1){
            CONT_01();
        }else if(op==2){
            CONT_02();
        }else if(op==3){
            CONT_03();
        }
    }

    return 0;
}

```

Explicação

DIRENT_01:

A variável qtd do tipo int será declarada. Essa variável será utilizada para armazenar a quantidade de arquivos presentes no computador.

O ponteiro dir do tipo DIR será declarado. Esse ponteiro será utilizado para armazenar o endereço do diretório desejado. Para utilizar esse tipo foi preciso incluir a biblioteca dirent.h no código por meio do #include <dirent.h>.

O ponteiro lsdirent do tipo struct dirent será declarado. Esse ponteiro será utilizado para armazenar o endereço de uma struct dirent. Tal struct é utilizada para armazenar os dados de um arquivo.

A struct dirent foi definida da seguinte maneira na biblioteca dirent:

```
struct dirent {
    ino_t      d_ino;
    off_t      d_off;
    unsigned short d_reclen; // Tamanho do arquivo
    unsigned char d_type;    // Tipo do arquivo
    char        d_name[256]; // Nome do arquivo
};
```

Em seguida, o ponteiro dir recebe o retorno da função opendir (oriunda da biblioteca dirent.h).

Essa função possui como parâmetro um vetor de caracteres que irá receber o caminho do diretório que será aberto. Nesse caso, se deseja abrir o diretório encaminhado para a função DIRENT_01. Portanto, a função recebe caminho_pai como argumento. A função opendir retorna o endereço do diretório em questão, tal endereço é armazenado no ponteiro dir.

Logo após, um laço while com a seguinte condição será executado:

Se o ponteiro lsdirent (esse ponteiro recebe o retorno da função readdir (oriunda da biblioteca dirent.h)) for diferente de null, o laço continua.

Essa função possui como parâmetro um ponteiro do tipo DIR que irá receber o endereço do diretório que será lido. Nesse caso, o argumento é o ponteiro dir. Tal função retorna o endereço da struct dirent do arquivo atual. A cada iteração o próximo arquivo é lido, quando não existir mais arquivo a função irá retornar null..

Ao iniciar o while duas variáveis serão declaradas: A variável nome com 256 caracteres do tipo char, ou seja, um vetor de 256 caracteres. Tal variável será utilizada para armazenar o nome do arquivo.

Em seguida, a função strcpy (oriunda da biblioteca string.h) será executada. Essa função possui dois parâmetros, o primeiro parâmetro é o vetor destino e o segundo parâmetro é o vetor origem. Nesse caso, o primeiro argumento é o vetor nome e o segundo argumento é o vetor d_name da struct dirent presente no ponteiro lsdirent, isto é, o segundo argumento é lsdirent→d_name. Essa função possui a utilidade de copiar a string presente no argumento 2 para o argumento 1.

Depois, um if é executado com as seguintes condições:

Se o nome do arquivo for diferente de "." (diretório atual):

Por meio da função strcmp é testado se o vetor nome é diferente da string ".", se for, a função retorna um valor diferente de 0.

&&

Se o nome do arquivo for diferente de “..” (diretório pai:

Por meio da função strcmp é testado se o vetor nome é diferente da string “.”, se for, a função retorna um valor diferente de 0.

Se o if atender todas as condições:

A variável caminho_filho com 1000 caracteres do tipo char será declarada. Tal variável será utilizada para armazenar o caminho pai + nome do arquivo lido.

A função strcpy (oriunda da biblioteca string.h) será executada. Essa função possui dois parâmetros, o primeiro parâmetro é o vetor destino e o segundo parâmetro é o vetor origem. Nesse caso, o primeiro argumento é o vetor caminho_pai e o segundo argumento é o vetor caminho_filho. Essa função possui a utilidade de copiar a string presente no argumento 2 para o argumento 1.

Um if será executado com a seguinte condição:

Se o caminho filho for diferente de “/” (diretório raiz).

Tal verificação é realizada por meio da função strcmp, essa função verifica se o vetor caminho_filho é diferente da string “/”, se for, a função retorna um valor diferente de 0.

Se o if atender essa condição:

O nome do arquivo se concatenará com o caminho pai.

Essa concatenação é feita por meio da função strcat. O primeiro parâmetro dessa função é o vetor destino e o segundo parâmetro dessa função é um vetor origem. Nesse caso, o primeiro argumento é o caminho pai e o segundo argumento é o nome do arquivo.

A variável qtd recebe um incremento.

Um if será executado com a seguinte condição:

Se o tipo do arquivo é igual à 4, ou seja, se o arquivo é um diretório.

Tal verificação faz com que o programa não entre em loop infinito ao acessar um arquivo que é um atalho.

Se o if atender essa condição:

Uma chamada recursiva será realizada:

A função DIRENT_01 será chamada, tal função possui como parâmetro um vetor de caracteres que receberá o caminho_pai. Nesse caso, o argumento é o caminho_filho.

O seu retorno é a quantidade de arquivos presentes no diretório lido. Tal retorno será incrementado à variável qtd.

Após o while se encerrar, a função closedir será executada. Essa função possui como parâmetro um ponteiro do tipo DIR que irá receber o endereço do diretório que será fechado. Nesse caso, esse argumento é o ponteiro dir.

Por fim, a variável qtd se tornará o retorno da função.

-----END-----

A variável global `diretorio_raiz` será declarada, tal variável é uma matriz 100x100 do tipo `char`, essa matriz será utilizada para armazenar os arquivos presentes no diretório raiz do computador.

A variável global `n_p` do tipo `int` será declarada, essa variável será utilizada para armazenar quantos processadores o computador possui. Se o computador possuir `x` processadores, `x` processos serão criados. Se o computador possui somente um processador, 2 processos serão criados.

A variável global `qtd_raiz` do tipo `int` será declarada, essa variável será utilizada para armazenar quantos arquivos estão presentes no diretório raiz do computador.

CONT_01:

A variável `qtd_total` do tipo `int` será declarada, tal variável receberá o retorno da função `DIRENT_01`. Essa função possui como parâmetro um vetor de caracteres que receberá o `caminho_pai`. Nesse caso, o argumento é o caractere `"/"`, ou seja, a pasta raiz do computador.

Um `printf` exibe o valor presente na variável `qtd_total`.

DIRENT_02:

O ponteiro `dir` do tipo `DIR` será declarado. Esse ponteiro será utilizado para armazenar o endereço do diretório desejado. Para utilizar esse tipo foi preciso incluir a biblioteca `dirent.h` no código por meio do `#include <dirent.h>`.

O ponteiro `lsdir` do tipo `struct dirent` será declarado. Esse ponteiro será utilizado para armazenar o endereço de uma `struct dirent`. Tal `struct` é utilizada para armazenar os dados de um arquivo.

A variável `i` do tipo `int` será declarada. Tal variável será utilizada como índice da matriz `diretorio_raiz`.

A `struct dirent` foi definida da seguinte maneira na biblioteca `dirent`:

```
struct dirent {
    ino_t      d_ino;
    off_t      d_off;
    unsigned short d_reclen; // Tamanho do arquivo
    unsigned char d_type;    // Tipo do arquivo
    char        d_name[256]; // Nome do arquivo
};
```

Em seguida, o ponteiro `dir` recebe o retorno da função `opendir` (oriunda da biblioteca `dirent.h`).

Essa função possui como parâmetro um vetor de caracteres que irá receber o caminho do diretório que será aberto. Nesse caso, se deseja abrir o diretório raiz. Portanto, a função recebe `"/"` como argumento. A função `opendir` retorna o endereço do diretório em questão, tal endereço é armazenado no ponteiro `dir`.

Logo após, um laço `while` com a seguinte condição será executado:

Se o ponteiro `lsdir` (esse ponteiro recebe o retorno da função `readdir` (oriunda da biblioteca `dirent.h`)) for diferente de `null`, o laço continua.

Essa função possui como parâmetro um ponteiro do tipo DIR que irá receber o endereço do diretório que será lido. Nesse caso, o argumento é o ponteiro dir. Tal função retorna o endereço da struct dirent do arquivo atual. A cada iteração o próximo arquivo é lido, quando não existir mais arquivo a função irá retornar null..

Ao iniciar o while duas variáveis serão declaradas: A variável nome com 256 caracteres do tipo char, ou seja, um vetor de 256 caracteres. Tal variável será utilizada para armazenar o nome do arquivo.

Em seguida, a função strcpy (oriunda da biblioteca string.h) será executada. Essa função possui dois parâmetros, o primeiro parâmetro é o vetor destino e o segundo parâmetro é o vetor origem. Nesse caso, o primeiro argumento é o vetor nome e o segundo argumento é o vetor d_name da struct dirent presente no ponteiro lsdirent, isto é, o segundo argumento é lsdirent->d_name. Essa função possui a utilidade de copiar a string presente no argumento 2 para o argumento 1.

Depois, um if é executado com as seguintes condições:

Se o nome do arquivo for diferente de "." (diretório atual):

Por meio da função strcmp é testado se o vetor nome é diferente da string ".", se for, a função retorna um valor diferente de 0.

&&

Se o nome do arquivo for diferente de ".." (diretório pai):

Por meio da função strcmp é testado se o vetor nome é diferente da string "..", se for, a função retorna um valor diferente de 0.

Se o if atender todas as condições:

diretorio_raiz[i] receberá a string "/" por meio da função strcpy.
diretorio_raiz[i] se concatenará com a variável nome por meio da função strcat

A variável i será incrementada.

A variável global qtd_raiz será incrementada.

Após o while se encerrar, a função closedir será executada. Essa função possui como parâmetro um ponteiro do tipo DIR que irá receber o endereço do diretório que será fechado. Nesse caso, esse argumento é o ponteiro dir.

F_01:

A variável pid do tipo pid_t será declarada;

A variável status do tipo int será declarada.

A variável qtd_diretório do tipo int é declarada.

O ponteiro file do tipo FILE é declarado.

Se n for maior que 0:

A variável pid recebe o retorno da função fork. A função fork faz com que o programa se divide em dois e comece a ser executado ao mesmo tempo em n_p processos.

Um if testa se $\text{pid} > 0$, ou seja, se o processo atual é o processo pai:

A função `wait` é chamada. Nesse caso, o argumento da função `wait` é o endereço da variável `status`. Tal função possui a utilidade de fazer com que o processo pai espere que o processo filho termine de executar para sua execução continuar.

Um `printf` exibe o retorno da função `getpid`.

Um `printf` exibe a string “f(n-1)”

A função `F_01` se chama, ou seja, uma recursão é realizada. Nesse caso, o primeiro argumento é a variável `n-1`, o segundo argumento é a variável `j`, e o terceiro argumento é o retorno da função `ceil`. A função `ceil` receberá o resultado da soma da variável `j` com a divisão da variável `qtd_raiz` pela variável `n_p`.

Ideia Geral:

`qtd = 25`

`n_p = 8`

Primeiro `f_01` -> `n = 8, i = 0, j = 4`

Segundo `f_01` -> `n = 7, i = 4, j = 8`

Terceiro `f_01` -> `n = 6, i = 8, j = 12`

Quarto `f_01` -> `n = 5, i = 12, j = 16`

Quinto `f_01` -> `n = 4, i = 16, j = 20`

Sexta `f_01` -> `n = 3, i = 20, j = 24`

Sétimo `f_01` -> `n = 2, i = 24, j = 28`

Oitavo `f_01` -> `n = 1, i = 28, j = 32`

Um if testa se $\text{pid} == 0$, ou seja, se o processo atual é o processo filho:

Um `printf` exibe o retorno da função `getpid`.

Um `printf` exibe a numeração do processo atual, o índice inicial e o índice final.

Um `for` é utilizado para percorrer os valores presentes entre `i` e `j`:

Um `printf` exibe o diretório que será lido e contado.

O retorno da função `DIRENT_01` é adicionado à variável `qtd_diretorio`. Nesse caso, o `diretorio_raiz[i]` é o argumento da função `DIRENT_01`.

A variável `file` recebe o retorno da função `fopen`. Nesse caso, o argumento dessa função é a string `qtd.txt` e a string “a” (“a” -> Adiciona no final do arquivo).

A função `fprintf` é chamada. Nesse caso, o argumento dessa função é o ponteiro `file`, o tipo da variável (`%d`) e a variável que será salva (`qtd_diretorio`).

A função `fclose` é chamada. Nesse caso, o argumento dessa função é o ponteiro `file`.

A função `exit` é chamada. Nesse caso, o argumento da função é o `status`. Tal função possui a utilidade de fazer com que o processo filho notifique o processo pai de que a sua execução foi encerrada.

Se n for menor que 0:

A variável file recebe o retorno da função fopen. Nesse caso, o argumento dessa função é a string qtd.txt e a string "r" ("r" -> Modo leitura).

A variável qtd_final do tipo int é criada.

A variável i do tipo i é criada. Tal variável será utilizada para exibir a que processo o valor lido do arquivo pertence.

Um while percorre o arquivo em questão enquanto o fim do arquivo não for detectado por meio da função feof:

A função fscanf lê um número e armazena na variável qtd_diretorio

Um printf exibe o número do processo atual, a quantidade anterior (qtd_final) e a quantidade atual de arquivos lidos (qtd_diretorio).

A variável i é incrementada.

Um printf exibe a quantidade de arquivos totais presentes no computador.

Para limpar o arquivo:

A variável file recebe o retorno da função fopen. Nesse caso, o argumento dessa função é a string qtd.txt e a string "w" ("w" -> Modo sobrescrita).

A função fprintf é chamada. Nesse caso, o argumento dessa função é o ponteiro file, o tipo da variável (%s) o valor que será salvo (" ").

A função fclose é chamada.

-----END-----

CONT_02:

A variável global qtd_raiz recebe o valor 0. Como a variável é global, se essa atribuição não for feita, qtd_raiz conterà o valor obtido na execução anterior.

A função DIRENT_02 será chamada. Tal função possui como parâmetro uma matriz. Nesse caso, o argumento é a matriz diretorio_raiz.. Tal função possui o objetivo de armazenar na matriz diretorio_raiz, os arquivos presentes no diretório raiz do computador.

A variável n_p receberá o retorno da função get_nprocs(). Tal função pertence à biblioteca sys/sysinfo.h/, ela retorna quantos processadores o computador possui.

A função F_01 será chamada. Tal função possui como parâmetro o número de processos que serão criados, o índice do primeiro diretório que será lido e contado e o índice do último diretório que será lido e contado. Nesse caso, o primeiro argumento é a variável n_p, o segundo argumento é o valor 0, e o terceiro argumento é o retorno da função ceil.

A função ceil receberá o resultado da divisão da variável qtd_raiz pela variável n_p. Tal função foi utilizada para arredondar o resultado da divisão para cima, esse arredondamento é extremamente importante, pois se o resultado não fosse exato, o resultado da divisão seria arredondado para baixo e com isso um ou mais diretórios não seriam lidos. Por exemplo, se o arquivo raiz do computador possuir 25 diretórios e o computador possuir 8 processadores, o resultado da divisão será 3.125, com isso, como as variáveis utilizadas na divisão são do tipo int, o valor seria arredondado para baixo. Dessa maneira, como cada diretório leria 3 diretório, o vigésimo quinto diretório não seria lido. Ao utilizar a função ceil, o vigésimo quinto diretório será lido no sétimo processo.

-----END-----

A struct INTERVALO é declarada e 4 variáveis são declaradas dentro dessa struct (i1,j1,i2,j2).

A variável global cont do tipo int é declarada e recebe o valor 1;

***THREAD_FUNCTION:**

O ponteiro para struct intervalo do tipo INTERVALO é declarado e recebe o argumento da função atual.

A variável i do tipo int é declarada.

A variável j do tipo int é declarada.

Um printf exibe a string "THREAD".

Se cont for igual a 1:

A variável i recebe a variável i1 da struct INTERVALO.

A variável j recebe a variável j1 da struct INTERVALO.

A variável cont é incrementada.

Se cont for igual a 2:

A variável i recebe a variável i2 da struct INTERVALO.

A variável j recebe a variável j2 da struct INTERVALO.

A variável cont é zerada.

A variável qtd_diretorio do tipo int é declarada.

Um for é utilizado para percorrer os valores presentes entre i e j:

Um printf exibe o valor atual do i e do j.

O retorno da função DIRENT_01 é adicionado à variável qtd_diretorio. Nesse caso, o diretorio_raiz[i] é o argumento da função DIRENT_01.

Um printf exibe o diretório que foi lido e contado.

A variável file recebe o retorno da função fopen. Nesse caso, o argumento dessa função é a string qtd.txt e a string "a" ("a" -> Adiciona no final do arquivo).

A função fprintf é chamada. Nesse caso, o argumento dessa função é o ponteiro file, o tipo da variável (%d) e a variável que será salva (qtd_diretorio).

A função fclose é chamada. Nesse caso, o argumento dessa função é o ponteiro file.

A função pthread_exit é chamada. Nesse caso, o argumento da função é NULL. Essa função serve para encerrar a thread.

F_02:

A variável pid do tipo pid_t será declarada;

A variável status do tipo int será declarada.

A variável qtd_diretorio do tipo int é declarada.

Se n for maior que 0:

A variável pid recebe o retorno da função fork. A função fork faz com que o programa se divide em dois e comece a ser executado ao mesmo tempo em n_p processos.

Um if testa se $pid > 0$, ou seja, se o processo atual é o processo pai:

A função `wait` é chamada. Nesse caso, o argumento da função `wait` é o endereço da variável `status`. Tal função possui a utilidade de fazer com que o processo pai espere que o processo filho termine de executar para sua execução continuar.

Um `printf` exibe o retorno da função `getpid`.

Um `printf` exibe a string “f(n-1)”

A função `F_02` se chama, ou seja, uma recursão é realizada. Nesse caso, o primeiro argumento é a variável `n-1`, o segundo argumento é a variável `j`, e o terceiro argumento é o retorno da função `ceil`. A função `ceil` receberá o resultado da soma da variável `j` com a divisão da variável `qtd_raiz` pela variável `n_p`.

Ideia Geral:

`qtd = 25`

`n_p = 8`

Primeiro `f_01` -> `n = 8, i = 0, j = 4`

Segundo `f_01` -> `n = 7, i = 4, j = 8`

Terceiro `f_01` -> `n = 6, i = 8, j = 12`

Quarto `f_01` -> `n = 5, i = 12, j = 16`

Quinto `f_01` -> `n = 4, i = 16, j = 20`

Sexta `f_01` -> `n = 3, i = 20, j = 24`

Sétimo `f_01` -> `n = 2, i = 24, j = 28`

Oitavo `f_01` -> `n = 1 = 28, j = 32`

Um if testa se $pid == 0$, ou seja, se o processo atual é o processo filho:

Um `printf` exibe o retorno da função `getpid`.

Um `printf` exibe a numeração do processo atual, o índice inicial e o índice final.

O vetor `thread_id` do tipo `pthread_t` com 2 posições será criado.

O ponteiro para struct intervalo do tipo `INTERVALO` é declarado e recebe o retorno da função `malloc`. A função `malloc` aloca espaço para a struct ser armazenada na memória. Nesse caso, a função `malloc` recebe o tamanho da struct `INTERVALO`.

A variável `i1` da struct `INTERVALO` recebe a variável `i`. Ex: `i = 0`

A variável `j2` da struct `INTERVALO` recebe a variável `j-2`. Ex: `j = 4 -> 4-2=2`

A variável `i2` da struct `INTERVALO` recebe a variável `j-2`. Ex: `j = 4 -> 4-2=2`

A variável `j2` da struct `INTERVALO` recebe a variável `j`. Ex: `j = 4`

Um laço for vai de x=0 a 2 (quantidade de threads desejadas):

A função `pthread_create` é chamada (Essa função pertence à biblioteca `pthread.h`). Nesse caso, essa função recebe o endereço da variável `thread_id` na posição x, `NULL`, um ponteiro para a função que será executada pela thread (`thread_function`), um ponteiro para o argumento que a função `thread_function` receberá (ponteiro para a struct `INTERVALO`).

Um laço for vai de y=0 a 2 (quantidade de threads desejadas):

A função `pthread_join` é chamada (Essa função pertence à biblioteca `pthread.h`). Nesse caso, essa função recebe a variável `thread_id` na posição y e o valor `NULL`. Essa função possui o objetivo de fazer com que a thread chamadora seja suspensa até que a thread presente na variável `thread_id` termine.

A função `exit` é chamada. Nesse caso, o argumento da função é o status. Tal função possui a utilidade de fazer com que o processo filho notifique o processo pai de que a sua execução foi encerrada.

Se n for menor que 0:

A variável `file` recebe o retorno da função `fopen`. Nesse caso, o argumento dessa função é a string `qtd.txt` e a string `"r"` (`"r"` -> Modo leitura).

A variável `qtd_final` do tipo `int` é criada.

A variável `i` do tipo `i` é criada. Tal variável será utilizada para exibir a que processo o valor lido do arquivo pertence.

Um `while` percorre o arquivo em questão enquanto o fim do arquivo não for detectado por meio da função `feof`:

A função `fscanf` lê um número e armazena na variável `qtd_diretorio`

Um `printf` exibe o número do processo atual, a quantidade anterior (`qtd_final`) e a quantidade atual de arquivos lidos (`qtd_diretorio`).

A variável `i` é incrementada.

Um `printf` exibe a quantidade de arquivos totais presentes no computador.

Para limpar o arquivo:

O ponteiro `file` do tipo `FILE` é declarado, esse ponteiro recebe o retorno da função `fopen`. Nesse caso, o argumento dessa função é a string `qtd.txt` e a string `"w"` (`"w"` -> Modo sobrescrita).

A função `fprintf` é chamada. Nesse caso, o argumento dessa função é o ponteiro `file`, o tipo da variável (`%s`) o valor que será salvo (`" "`).

A função `fclose` é chamada.

CONT_03:

A variável global `qtd_raiz` recebe o valor 0. Como a variável é global, se essa atribuição não for feita, `qtd_raiz` conterá o valor obtido na execução anterior.

A função `DIRENT_02` será chamada. Tal função possui como parâmetro uma matriz. Nesse caso, o argumento é a matriz `diretorio_raiz`. Tal função possui o objetivo de armazenar na matriz `diretorio_raiz`, os arquivos presentes no diretório raiz do computador.

A variável `n_p` receberá o retorno da função `get_nprocs()`. Tal função pertence à biblioteca `sys/sysinfo.h`, ela retorna quantos processadores o computador possui.

A função `F_02` será chamada. Tal função possui como parâmetro o número de processos que serão criados, o índice do primeiro diretório que será lido e contado e o índice do último diretório que será lido e contado. Nesse caso, o primeiro argumento é a variável `n_p`, o segundo argumento é o valor 0, e o terceiro argumento é o retorno da função `ceil`.

A função `ceil` receberá o resultado da divisão da variável `qtd_raiz` pela variável `n_p`. Tal função foi utilizada para arredondar o resultado da divisão para cima, esse arredondamento é extremamente importante, pois se o resultado não fosse exato, o resultado da divisão seria arredondado para baixo e com isso um ou mais diretórios não seriam lidos. Por exemplo, se o arquivo raiz do computador possuir 25 diretórios e o computador possuir 8 processadores, o resultado da divisão será 3.125, com isso, como as variáveis utilizadas na divisão são do tipo `int`, o valor seria arredondado para baixo. Dessa maneira, como cada diretório leria 3 diretório, o vigésimo quinto diretório não seria lido. Ao utilizar a função `ceil`, o vigésimo quinto diretório será lido no sétimo processo.

MAIN:

A variável `op` do tipo `int` será declarada.

Um pequeno texto com as informações do programa será imprimido na tela.

Um pequeno texto com as opções será imprimido na tela;

Um laço `while` infinito será iniciado:

A string `">>> "` será impressa, tal string é utilizada somente para indicar ao usuário que ele deve digitar algum comando.

A função `scanf` lerá o valor digitado pelo usuário e armazenará tal valor na variável `op`.

Se `op` for igual a 0:

O programa se encerra

Se `op` for igual a 1:

A função `CONT_01` é chamada

Se `op` for igual a 2:

A função `CONT_02` é chamada

Se `op` for igual a 3:

A função `CONT_03` é chamada

TEMPOS DE EXECUÇÃO

CONT_01:

```
euller_hbo@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q5$ gcc Q5.c -o Q5.exe -lpthread -lm
euller_hbo@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q5$ time ./Q5.exe

QTD_TOTAL: 918452

real    0m4,693s
user    0m0,254s
sys     0m2,195s
euller_hbo@euller:~/Documentos/Study/UFU/4_QUARTO_PERIODO/Sistemas Operacionais/Listas/Atividade_02/Q5$
```

CONT_02:

```
/run
/var
/opt
fork: 3077      ->      PID PAI: 3045      ->->      f(n-1)
fork: 0        ->      PID FILHO: 3080      ->->      PROCESSO 3-> i: 20.000000, j: 24.000000

/cdrom
/lib
/swapfile
/sbin
fork: 3080      ->      PID PAI: 3045      ->->      f(n-1)
fork: 0        ->      PID FILHO: 3081      ->->      PROCESSO 2-> i: 24.000000, j: 28.000000

/mnt

fork: 3081      ->      PID PAI: 3045      ->->      f(n-1)
fork: 0        ->      PID FILHO: 3082      ->->      PROCESSO 1-> i: 28.000000, j: 32.000000

fork: 3082      ->      PID PAI: 3045      ->->      f(n-1)
PROCESSO 1: Qtd anterior: 0 + Qtd atual 272706 = 272706

PROCESSO 2: Qtd anterior: 272706 + Qtd atual 379276 = 651982

PROCESSO 3: Qtd anterior: 651982 + Qtd atual 304 = 652286

PROCESSO 4: Qtd anterior: 652286 + Qtd atual 313824 = 966110

PROCESSO 5: Qtd anterior: 966110 + Qtd atual 14696 = 980806

PROCESSO 6: Qtd anterior: 980806 + Qtd atual 39664 = 1020470

PROCESSO 7: Qtd anterior: 1020470 + Qtd atual 0 = 1020470

PROCESSO 8: Qtd anterior: 1020470 + Qtd atual 0 = 1020470

QTD_TOTAL: 1020470

real    0m4,288s
user    0m0,261s
sys     0m1,893s
```

CONT_03:

```
fork: 2729          ->          PID PAI: 2666          ->->      f(n-1)
P||T: 1 -> Qtd atual: 1538 + Qtd anterior 0 = 1538
P||T: 2 -> Qtd atual: 271189 + Qtd anterior 1538 = 272727
P||T: 3 -> Qtd atual: 2783 + Qtd anterior 272727 = 275510
P||T: 4 -> Qtd atual: 321104 + Qtd anterior 275510 = 596614
P||T: 5 -> Qtd atual: 3 + Qtd anterior 596614 = 596617
P||T: 6 -> Qtd atual: 301 + Qtd anterior 596617 = 596918
P||T: 7 -> Qtd atual: 96381 + Qtd anterior 596918 = 693299
P||T: 8 -> Qtd atual: 218035 + Qtd anterior 693299 = 911334
P||T: 9 -> Qtd atual: 1196 + Qtd anterior 911334 = 912530
P||T: 10 -> Qtd atual: 13497 + Qtd anterior 912530 = 926027
P||T: 11 -> Qtd atual: 385 + Qtd anterior 926027 = 926412
P||T: 12 -> Qtd atual: 39279 + Qtd anterior 926412 = 965691
P||T: 13 -> Qtd atual: 0 + Qtd anterior 965691 = 965691
P||T: 14 -> Qtd atual: 0 + Qtd anterior 965691 = 965691
P||T: 15 -> Qtd atual: 0 + Qtd anterior 965691 = 965691
P||T: 16 -> Qtd atual: 0 + Qtd anterior 965691 = 965691
QTD_TOTAL: 965691
real      0m6,949s
user      0m0,441s
sys       0m2,595s
```