

EA872 Laboratório de Programação de Software Básico

Atividade 5



Vinícius Esperança Mantovani

RA 247395

Entrega (limite): 06/09/2023, 13:00

OBSERVAÇÃO: existem imagens que contêm, além da resposta do terminal para o comando desejado, também sua resposta para o comando “ls” antes e depois da execução do comando de chamada do programa. Isso se dá para que se possa observar a criação de possíveis arquivos pelo programa.

Atividade d)

- d1) Compilando e executando o código d.c, obtemos a seguinte saída:

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c b1.c b2.c c.c d d.c e.c f.c g.c h.c teste.d
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./d teste.d
1a. Leitura:
9876543210
9876543210
9876543210
9876543210
98
2a. Leitura:
76543210
9876543210
9876543210
9876543210
9876
3a. Leitura:
543210
9876543210
9876543210
9876543210
987654
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c b1.c b2.c c.c d d.c e.c f.c g.c h.c teste.d
```

Figura 1: resposta do terminal para a execução do código “d.c” com argumento “teste.d”

- d2) Analisando o código do programa, nota-se que temos um buffer de 50 caracteres e três variáveis, “i”, “j” e “k”. As duas primeiras, são file descriptors, ou descritores de arquivo, que são inteiros representantes do índice de uma entrada na tabela de descritores e, o conteúdo dessas entradas é um índice para a tabela de arquivos. A variável “i” é definida como um file descriptor de “teste.d” no trecho “i = open(argv[1], O_RDONLY, 0600)” e, em seguida, temos a variável “j” sendo definida, também como tal, na linha “j = dup(i);”, na qual se gera outro file descriptor para o arquivo “teste.d” e o retorna na variável “j”. Desse modo, existem três blocos de impressão principais no código, um deles lê o conteúdo do arquivo de file descriptor “i” para o buffer e o imprime em sequência (a parte lida). Em seguida, existem dois blocos que fazem o mesmo que o anterior, porém, usam o file descriptor “j”. Esse processo faz com que sejam impressas três partes subsequentes de 50 caracteres de teste.d no output, uma vez que, como a função read lê partes subsequentes do texto e, o file descriptor “j”, apesar de não ser igual ao “i”, aponta para uma outra entrada na tabela de descritores que tem como conteúdo o mesmo arquivo. Esse comportamento pode ser observado na Figura 1, que documenta a saída do programa “d.c”.

Atividade e)

- e1)

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c b1.c b2.c c.c d d.c e e.c f.c g.c h.c teste.d
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./e teste1.e ls -s
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c b1.c b2.c c.c d d.c e e.c f.c g.c h.c teste1.e teste.d
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ cat teste1.e
total 72
 4 a.c
 4 b1.c
 4 b2.c
 4 c.c
16 d
 4 d.c
16 e
 4 e.c
 4 f.c
 4 g.c
 4 h.c
 0 teste1.e
 4 teste.d
```

Figura 2: resposta do terminal para a execução do programa “e.c” conforme indicado no enunciado

- e2) Analisando o código, percebe-se que o programa recebe como argumentos um nome de arquivo, um programa a ser executado e, os argumentos a serem passados para a execução desse programa. Isso se dá, de modo que há a definição de um file descriptor “fd” com o nome passado como argumento e, em seguida, cria-se um novo file descriptor de valor 1, ou seja, uma entrada na posição 1 da tabela de descritores (saída padrão), para o arquivo teste1.e. Por fim, o primeiro file descriptor, “fd”, é fechado e é usada a função “execvp” para iniciar o programa passado como argumento no comando, neste caso, “ls”, com os argumentos, também, passados como argumentos no comando. Desse modo, a saída de “ls -s” é escrita na saída-padrão, que é o arquivo teste1.e. Portanto, o arquivo passa a conter os tamanhos em blocos de disco dos arquivos da presente pasta, conforme vê-se na Figura 2.
- e2) Para o comando “ls -s > teste2.e”, temos a seguinte saída:

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls -s > teste2.e
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ cat teste2.e
total 76
 4 a.c
 4 b1.c
 4 b2.c
 4 c.c
16 d
 4 d.c
16 e
 4 e.c
 4 f.c
 4 g.c
 4 h.c
 4 teste1.e
 0 teste2.e
 4 teste.d
```

Figura 3: saída no terminal para o comando “ls -s > teste2.e” acompanhado de “cat teste2.e”

- e4) Conforme explicado anteriormente, o comando usado amostra os nomes dos arquivos presentes na pasta atual e seus tamanhos em blocos de disco, que dependem tanto do sistema de arquivos como das configurações do sistema.
- e5) A única diferença entre os arquivos é a existência de 4 novos blocos no teste2.e que não haviam sido contabilizados em teste1.e, uma vez que eram os 4 blocos componentes dele próprio, portanto não seriam contabilizados de fato. Sendo assim, o programa funciona conforme explicado mais a fundo em “e2)”, é criado um file descriptor que é definido como file descriptor de um arquivo de nome recebido como argumento, em seguida, a entrada no índice um da tabela de descritores passa a assumir o valor da entrada de índice dado pelo “fd” (por meio da função “dup2”), ou seja, passa a conter um apontador para a representação de “teste1.c” na tabela de arquivos. Por fim, o primeiro file descriptor é fechado e se inicia a execução do comando “ls -s” com a saída-padrão em “teste1.c”.

Atividade f)

- f1) Abaixo se encontra a resposta do terminal para a execução do comando desejado:

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c  b2.c  d      e      f      g.c  teste1.e  teste.d
b1.c  c.c    d.c    e.c    f.c    h.c  teste2.e
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./f teste.d
Falha na chamada <stat>!
A chamada <fstat> foi bem sucedida!
Leu 50 bytes: 9876543210
9876543210
9876543210
9876543210
98
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c  b2.c  d      e      f      g.c  teste1.e  teste.d
b1.c  c.c    d.c    e.c    f.c    h.c  teste2.e
```

Figura 4: resposta do terminal para a execução do programa “f.c”

- f2) Ao analisar o código, notamos que a chamada “<stat>” falha, pois a string passada para especificar o arquivo do qual se quer adquirir informações não está linkada a nenhum arquivo desde a linha “unlink(“tmp”)”, na qual “tmp” passa a não especificar mais arquivo algum. Portanto, ao usar-se “tmp” como parâmetro para a função “stat”, recorremos em um erro na execução da função, o qual resulta no retorno de “-1” e, conseqüentemente, pelo fluxo do programa, na impressão da mensagem de falha vista na Figura 4.
- f3) Caso removamos a chamada “link”, teríamos uma falha na execução da função “unlink”, pois “tmp” não estaria vinculada a arquivo nenhum. Desse modo, ocorreria que a condicional “if (unlink(“tmp”) == -1)” seria satisfeita e o comando “exit(-1);” seria executado, pois o retorno de “unlink(“tmp”)” seria -1 pelo problema de link de “tmp”. Assim, o programa teria sua execução encerrada.
- f4) Existem dois problemas principais na execução do último “printf” que podem ocorrer. O primeiro deles, seria a possibilidade de mal comportamento de “printf” por conta da possível inexistência de um caractere nulo no fim da string “buf”, o que seria resolvido pela adição deste caractere ao fim da string. Ademais, o problema mais fácil de se perceber é que, no caso de termos um erro na execução de read, teríamos como retorno que o número de bytes lidos foi igual a “-1” na saída escrita pelo “printf”.

Atividade g)

- g1) As figuras abaixo apresentam respectivamente a resposta do terminal para o uso do comando “./g” com os argumentos “/dev/disk” (arquivo de diretório), “/dev/console” (arquivo especial) e “teste.d” (arquivo regular).

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./g /dev/disk

/dev/disk e' um arquivo-diretorio

user-id 0, group-id 0, permissao 40755, link(s) 7
tamanho 140 (bytes), 0 (blocos),Inode # 233

Dados de /dev/disk foram modificados pela ultima vez em Tue Sep 5 19:08:49 2023

Leitura permitida
Escrita proibida
Execucao permitida
```

Figura 5: resposta do terminal para o uso do comando “./g /dev/disk”

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./g /dev/console

/dev/console e' um arquivo especial (caracter)

user-id 0, group-id 5, permissao 20620, link(s) 1
tamanho 0 (bytes), 0 (blocos),Inode # 12

Dados de /dev/console foram modificados pela ultima vez em Tue Sep 5 19:08:50 2023

Leitura proibida
Escrita proibida
Execucao proibida
```

Figura 6: resposta do terminal para o uso do comando “./g /dev/console”

```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./g teste.d

teste.d e' um arquivo regular

user-id 1000, group-id 1000, permissao 100644, link(s) 1
tamanho 226 (bytes), 8 (blocos),Inode # 11799242

Dados de teste.d foram modificados pela ultima vez em Tue Aug 29 20:54:48 2023

Leitura permitida
Escrita permitida
Execucao proibida
```

Figura 7: resposta do terminal para o uso do comando “./g teste.d”

- g2) O programa funciona como um apresentador de status de um arquivo, de forma que tem um objeto da estrutura stat, “statbuf”, que é utilizado para amostrar várias informações sobre o arquivo. Isso se dá de modo que é passado o nome de um arquivo como argumento no comando “./g...”, então, esse arquivo tem seus status armazenados em “statbuf” pela função “stat()” e, já em seguida, é impresso de que

tipo o arquivo se trata por meio de um “switch” entre casos de “st_mode” (usa-se a máscara IFMT para permitir que se trate apenas dos bits de st_mode que fazem referência ao tipo de arquivo): arquivo regular, caso tenhamos a parte relevante de st_mode igual a S_IFREG; arquivo-diretório, caso tenhamos a parte relevante de st_mode igual a S_IFDIR; arquivo especial (caracter), caso tenhamos a parte relevante de st_mode igual a S_IFCHR e; arquivo especial (bloco), caso tenhamos a parte relevante de st_mode igual a S_IFBLK.

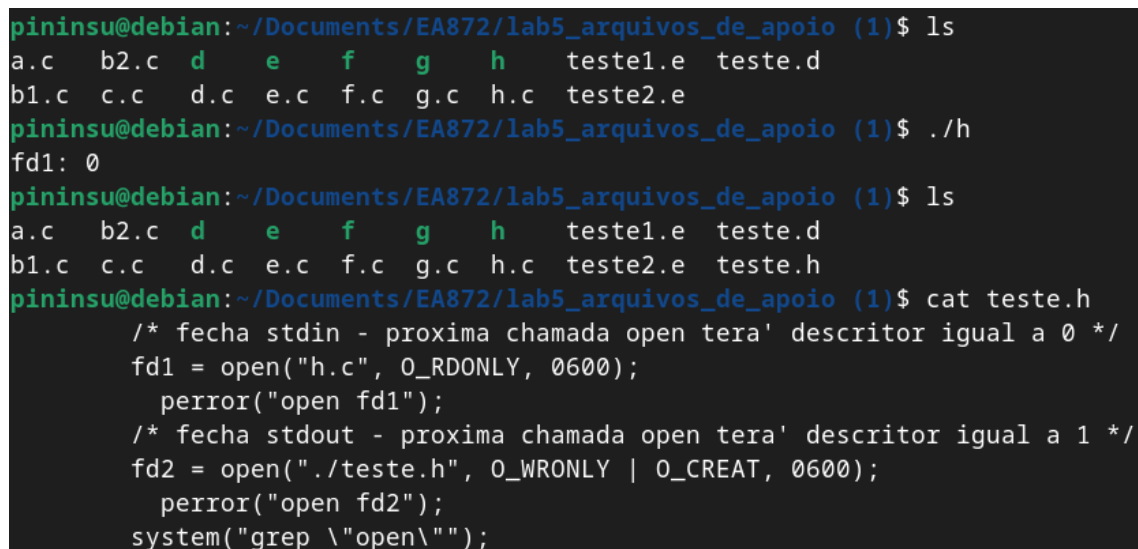
Nota-se ainda, que o processo descrito anteriormente só é executado se não houverem problemas na execução da função “stat” nem no número de argumentos passados no comando, pois esses problemas ocasionariam o encerramento da execução do programa.

Seguindo, então, com o fluxo do programa, temos uma série de dados a respeito do arquivo passado como argumento sendo impressas por meio de “fprintf” com atributos do objeto “stabuf” da estrutura “stat”, que armazenam as informações buscadas nas chamadas de “fprintf”.

Portanto, as saídas amostradas nas figuras 5, 6 e 7 são facilmente justificadas, já que, conforme descrito anteriormente a respeito do fluxo do programa, nas figuras temos, inicialmente a apresentação do tipo do arquivo cujo nome foi passado como argumento no comando, o que se segue por uma sequência de informações a respeito deste arquivo e, por fim, ainda conforme o código, são impressas três linhas amostrando as possibilidades para escrita, leitura e execução do arquivo.

Atividade h)

- h1) A seguir, está a imagem da resposta do terminal para o comando “./h” e, o conteúdo do arquivo “teste.h” gerado:



```
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c  b2.c  d      e      f      g      h      teste1.e  teste.d
b1.c  c.c    d.c    e.c    f.c    g.c    h.c    teste2.e
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ./h
fd1: 0
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ ls
a.c  b2.c  d      e      f      g      h      teste1.e  teste.d
b1.c  c.c    d.c    e.c    f.c    g.c    h.c    teste2.e  teste.h
pininsu@debian:~/Documents/EA872/lab5_arquivos_de_apoio (1)$ cat teste.h
/* fecha stdin - proxima chamada open tera' descritor igual a 0 */
fd1 = open("h.c", O_RDONLY, 0600);
perror("open fd1");
/* fecha stdout - proxima chamada open tera' descritor igual a 1 */
fd2 = open("./teste.h", O_WRONLY | O_CREAT, 0600);
perror("open fd2");
system("grep \"open\"");
```

Figura 8: resposta do terminal para o comando “./h” e o conteúdo do arquivo gerado

- h2) O programa funciona da seguinte forma: inicialmente, são definidos dois inteiros “fd1” e “fd2” e a entrada de índice 0 da tabela de descritores (stdin) é fechada. Em

sequência, “fd1” passa a ser o file descriptor de “h.c” e, é impresso por “printf” o valor de fd1 no terminal, que deve ser igual a 0, ou seja, o arquivo “h.c” passa a ser a entrada padrão. Seguindo, é fechada, também, a entrada de índice 1 da tabela de descritores, que se trata da saída padrão (stdout) e, “fd2” é definido como file descriptor de um arquivo que é criado neste momento chamado “./teste.h” com permissão apenas para escrita. Disso, segue que é feito um “grep” com o comando “grep \”open\””, responsável por procurar pela palavra “open” na entrada padrão, que neste caso, é o próprio arquivo “h.c”. Logo, o próprio comando “grep” imprime as linhas de “h.c”, em que se encontraram aparições de “open”, na saída padrão, que neste caso é o arquivo “teste.h”.

Portanto, a saída gerada pelo programa “h.c” é justificada pelo fato de, conforme a Figura 8, a saída ser, no terminal (saída padrão até o momento do fluxo de código) um “fd1: 0” e, no arquivo “teste.h”, a saída ser todas as linhas de “h.c” que contêm a palavra “open” no mínimo uma vez. Isso é explicitamente o exemplo prático do que foi descrito anteriormente a respeito do funcionamento do programa “h.c”.

Atividade i)

Abaixo encontra-se o código do programa desenvolvido para exercer tal tarefa, no entanto, todas as saídas, para qualquer entrada, estão sendo **"HTTP 404: File Not Found"**.

C/C++

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>

/*
 * Responde a requisicao do usuario por um recurso.
 *
 * @param caminho string indicadora do caminho para o recurso;
 * @param recurso string indicadora do nome do recurso;
 */
int getRes(char *caminho, char *recurso){ //a string caminho deve ser acompanhada de um / no final
    int fd1;
    char buf[1000];
    int fd2;
```

```

struct stat statind;
struct stat statwel;
struct stat statbuf;
char caminho_recurso[100] = "";

//concatena o caminho e o nome do arquivo:
strcpy(caminho_recurso, caminho);
strcat(caminho_recurso, recurso);

//guarda os status do recurso em statbuf
int st_out = stat(caminho_recurso, &statbuf);
printf("%s\n", caminho_recurso); //print de teste do caminho_recurso

//caso o arquivo nao exista
if (st_out == -1 && errno == ENOENT) {
    printf("HTTP 404: File Not Found");
    return 404;
}

//caso nao haja permissao de leitura
if((statbuf.st_mode & S_IRUSR) == 0){
    printf("HTTP 403: Forbidden");
    return 403;
}

//switch entre casos de arquivo regular ou diretorio
switch (statbuf.st_mode & S_IFMT)
{
    //caso seja arquivo regular:
    case S_IFREG :
        if((statbuf.st_mode & S_IRUSR) != 0){
            fd1 = open(caminho_recurso, O_RDONLY, 0600);
            int rBytes = read(fd1, buf, sizeof(buf));
            int wBytes = write(1, buf, rBytes);
        }
        break;

    //caso seja arquivo de diretorio
    case S_IFDIR :
        //caso seja permitida a varredura no diretorio

```



```

if((statbuf.st_mode & S_IXUSR) != 0){

    char cpy1[100]; //copia de caminho_recurso com "index.html" concatenado
    char cpy2[100]; //copia de caminho_recurso com "welcome.html" concatenado
    int rBytes; //numero de bytes lido
    int existsInd; //flag para identificar existencia ou nao do arquivo index.html
    int existsWel; //flag para identificar existencia ou nao do arquivo welcome.html

    //faz o processo de copia e concatenacao de cpy1 e cpy2
    strcpy(cpy1, caminho_recurso);
    strcpy(cpy2, caminho_recurso);
    strcat(cpy1, "/index.html");
    strcat(cpy2, "/welcome.html");

    //caso não existam index.html nem welcome.html
    if((existsInd = stat(cpy1, &statind) == -1 && errno == ENOENT) &&
        (existsWel = stat(cpy2, &statwel) == -1 && errno == ENOENT)){
        printf("HTTP 404: File Not Found");
        return 404;
    }

    //caso exista e haja permissao para leitura index.html:
    else if(existsInd != -1 && (statind.st_mode & S_IRUSR) != 0){

        fd2 = open(cpy1, O_RDONLY, 0600);

        while((rBytes = read(fd2, buf, sizeof(buf))) != 0){
            write(1, buf, rBytes);
        }

        //caso exista e haja permissao para leitura welcome.html:
    } else if(existsWel != -1 && (statwel.st_mode & S_IRUSR) != 0){

        fd2 = open(cpy2, O_RDONLY, 0600);

        while((rBytes = read(fd2, buf, sizeof(buf))) != 0){
            write(1, buf, rBytes);
        }

        //caso em que pelo menos um existe mas nenhum tem permissao para leitura:
    } else{

```

```

        printf("HTTP 403: Forbidden");
        return 403;
    }
}
break;
}
printf("EO_functionTest");
return 0;
}

int main(int argc, char *argv[]) { //main para teste da funcao!
    int k = getRes(argv[1], argv[2]);
    return 0;
}

```

Para testar o código, foi usado um programa chamado “get.c”, que recebe como parâmetros o caminho do arquivo e o nome dele, conforme o código acima (o caminho deve ter uma barra no final).

Abaixo, estão as respostas do programa aos diversos casos tratados por ele. Vale destacar que, para facilitar a compreensão dos casos, a função além de ter retornos diferentes para cada caso, imprime um descritivo do caso.

```

pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/meu-webs
pace/ teste.html ; echo
/home/pininsu/meu-webspace/teste.html
HTTP 404: File Not Found

```

Figura 9: Teste da função para um arquivo inexistente

```

pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/meu-webs
pace/dir1/ texto2.html ; echo
/home/pininsu/meu-webspace/dir1/texto2.html
HTTP 403: Forbidden

```

Figura 10: Teste da função para um arquivo sem permissão para leitura

```
pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/meu-webspace/
dir1/ texto1.html ; echo
/home/pininsu/meu-webspace/
dir1/texto1.html
<html><head>
  <title>TEST01</title>
</head>
<body>
  Aqui está o conteúdo da página. Altere-o de maneira a
  facilitar a identificação de cada página. Ajudaria dizer aqui
  qual é a página e onde ela está na estrutura de diretórios.

</body></html>E0_functionTest
```

Figura 11: Teste da função para um arquivo regular com permissão de leitura

```
pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/meu-webspace /
; echo
/home/pininsu/meu-webspace/
<!DOCTYPE html>
<html>
<head>
<title>INDEX</title>
</head>
<body>
  Aqui está o conteúdo da página. Altere-o de maneira a
  facilitar a identificação de cada página. Ajudaria dizer aqui
  qual é a página e onde ela está na estrutura de diretórios.
</body>
</html>E0_functionTest
```

Figura 12: Teste da função para um arquivo diretório com arquivo index.html dentro

```
pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/Documents/EA872/ativ5/
teste ; echo
/home/pininsu/Documents/EA872/ativ5/teste
Este é um <html></html>E0Function_Test
```

Figura 13: Teste da função para um arquivo diretório com arquivo welcome.html dentro

```
pininsu@debian:~/Documents/EA872/ativ5$ ./get /home/pininsu/Documents/EA872/ativ5/
teste ; echo
/home/pininsu/Documents/EA872/ativ5/teste
HTTP 403: Forbidden
```

Figura 14: Teste da função para um arquivo diretório com welcome.html sem permissão de leitura dentro