EA872 - Laboratório de Programação de Software Básico

Atividade 5

Sistema de Arquivos

1. Tema

Sistema de arquivos do UNIX

2. Objetivos

Familiarização com o sistema de arquivos do UNIX. Introdução a chamadas ao sistema para manipulação de arquivos.

3. Um modelo de sistemas de arquivos

Um **arquivo** (*file*) é um repositório de dados nas unidades de memória secundária (p.ex., fitas, discos e memórias flash). Por outro lado, um **sistema de arquivos** (*filesystem*) é uma estrutura de dados criada a partir de uma coleção de arquivos. Em UNIX, um sistema de arquivos pode ser de dois tipos:

- físico: compreende a disposição dos arquivos nas partições do disco. O sistema de arquivos físico é dividido pelas partições do disco, as quais determinam o número de blocos que pode ser utilizado pelos arquivos.
- lógico: árvore de diretórios composta por todos os arquivos (e partições de disco) que são acessíveis ao usuário.

O principal objetivo de um sistema de arquivos é proporcionar máxima independência entre os dispositivos físicos de armazenamento e os processos. Nesta experiência é apresentado o modelo de sistema de arquivos tradicionalmente implementado no UNIX.

No UNIX um arquivo é uma sequência de blocos contendo 512, 1024, 2048 ou algum outro múltiplo de 512 *bytes*. O tamanho de blocos é homogêneo dentro de um sistema de arquivos, mas pode variar de um sistema para outro. As informações genéricas de cada arquivo, tais como identificador do proprietário, tipo de arquivo, permissões de acesso, tamanho de arquivo e endereços físicos dos blocos que compõem o arquivo, são armazenadas numa estrutura (de 64, 128 ou 256 *bytes*) denominada **Inode** (*index-node*). No núcleo, os arquivos não são referenciados pelos seus nomes simbólicos, mas sim pelos seus números de Inode. Múltiplos nomes simbólicos podem ser utilizados para descrever o mesmo arquivo pela criação de um *link* (vínculo, conexão, atalho) entre cada nome simbólico e o Inode.

No UNIX distinguem-se três classes de arquivos:

- arquivos regulares: repositório de dados e programas, podendo conter texto em código ASCII ou dados em representação binária, incluindo programas executáveis.
- arquivos-diretório: são arquivos que contêm referências a outros arquivos, as quais são usadas para conversão dos nomes simbólicos de arquivos em seus respectivos números de Inodes. Portanto, os dados dos arquivos não ficam dentro de diretórios, os quais contêm somente o nome simbólico de cada arquivo e seu número de Inode. O número de Inode atua então como um ponteiro para o local onde o sistema pode encontrar a informação necessária sobre o arquivo.
- arquivos especiais (arquivos de dispositivo): todos os dispositivos físicos são referenciados via arquivos especiais, utilizados para estabelecer a comunicação com o *hardware*. Estes arquivos contêm informações como localização, tipo e modo de acesso aos dispositivos de *hardware*. Existem dois tipos de arquivos especiais, um para os dispositivos de bloco (fazem E/S utilizando *buffer*) e outro para os dispositivos de caractere (fazem E/S sem usar *buffer*, ou seja, a transmissão é feita caractere a caractere).

No sistema UNIX pode-se particionar uma unidade física de armazenamento (por exemplo, um disco) em vários sistemas de arquivos, denominados **dispositivos lógicos** (*logic devices*). Os dispositivos lógicos são identificáveis pelos **números de dispositivo** (*device number*) e o mapeamento dos endereços dos dispositivos lógicos em endereços físicos das unidades de disco é feito pelos *drivers* dos dispositivos. Um sistema de arquivo é composto por:

- **bloco de boot (*boot block*)**: é o primeiro bloco do sistema de arquivos, reservado para códigos de partida (*boot*). Normalmente, há apenas um bloco de boot por sistema de arquivos.
- **superbloco (*superblock*)**: é o segundo bloco. Contém informações cruciais sobre o estado do sistema de arquivo, tais como a quantidade de Inodes, a quantidade de blocos no disco e o endereço do início da lista de blocos livres no disco.

- **lista de Inodes (*Inode list*)**: fica nos blocos que seguem o superbloco. Contém informações de cada Inode definido no sistema de arquivos. O tamanho da lista é especificado pelo administrador do sistema e o núcleo faz referência aos Inodes pelos respectivos índices na lista. Um Inode dessa lista é o **Inode-raiz** (*root-Inode*) que permite estabelecer conexão lógica de um sistema de arquivos com um outro sistema de arquivos, através da chamada de sistema **`mount`**.
- **blocos de dados (*data blocks*)**: são os blocos após aqueles que contêm a lista de Inodes; contêm os dados administrativos e os dados dos arquivos. Cada bloco de dados só pode pertencer a um sistema de arquivos.

Nesta atividade são apresentadas sucintamente as estratégias adotadas pelo UNIX para gerenciar de forma eficiente o sistema de arquivos e as chamadas de sistema para acessá-lo e manipulá-lo. Quando um processo necessita acessar os dados de um arquivo, o núcleo deve carregá-los do disco à memória principal. Para melhorar o tempo de resposta do sistema operacional em relação às operações sobre o sistema de arquivos, o núcleo adota as seguintes estratégias para minimizar a frequência de acesso ao disco:

- **uso de *cache* de *buffers***: ao invés de acessar o disco em cada operação de leitura de um arquivo regular, o núcleo copia o bloco de dados desejado no *cache* de *buffers* e o utiliza para várias operações. De forma similar, pode-se melhorar o desempenho de operações de escrita, fazendo-a sempre no cache de buffers (em um bloco previamente copiado do disco) e só atualizando o disco quando o arquivo for fechado ou quando for preciso liberar área no *cache* de *buffers* para carregar outros blocos;
- **manutenção de dados auxiliares na memória**: o núcleo mantém na memória uma cópia de superblocos e dos Inodes, que contêm informações necessárias à alocação/remoção de blocos de dados.

3.1. Arquivos Regulares

No sistema UNIX os blocos de dados de um arquivo regular são organizados em lista. Assim, um arquivo não precisa ser armazenado num espaço físico contíguo. Esta estratégia de organização torna flexível a expansão e a contração de um arquivo, além de evitar desperdícios. Por outro lado, este esquema de organização pode tornar caro o gerenciamento da lista de blocos, se o arquivo for muito grande. Uma solução intermediária implementada no sistema UNIX é fixar o número de entradas na tabela de endereços em 15. As doze primeiras entradas, chamadas de *direct*, apontam diretamente ao bloco que contém os dados efetivos de um arquivo; a décima terceira, chamada de *single indirect*, referencia um bloco que contém apontadores para os blocos de dados do arquivo; a décima quarta, chamada de *double indirect*, contém os apontadores para outros blocos do tipo *single indirect* e a décima quinta, chamada de *triple indirect*, contém apontadores para outros blocos do tipo *double indirect*. Teoricamente não há restrição tanto nos níveis de encadeamento quanto na quantidade dos tipos de blocos; mas na prática essa configuração tem se mostrado há muito tempo suficiente para os tamanhos de arquivos criados. Os aumentos no tamanho dos blocos têm compensado o aumento no tamanho dos arquivos que vem ocorrendo continuamente.

O nome de um arquivo (regular ou não) em UNIX pode ter qualquer comprimento e conter qualquer caractere. No entanto, deve-se evitar o emprego de caracteres de pontuação (com exceção do hífen, *underscore* e ponto), espaços e caracteres não-imprimíveis, pois estes podem ter significado especial em uma linha de comando. O Unix faz distinção entre caracteres maiúsculos e minúsculos.

3.2. Arquivos-Diretório

O principal papel dos arquivos-diretório é a conversão do nome simbólico de um arquivo em seu número de Inode. A pesquisa por um nome simbólico começa pelo diretório corrente do processo (se o primeiro caractere do nome é diferente de /) ou pelo diretório-raiz (se o primeiro caractere do nome é /). Portanto, para iniciar a pesquisa por um nome é preciso localizar o Inode do diretório inicial de pesquisa. O inode do diretório corrente está armazenado na estrutura de usuário do processo e o Inode do diretório-raiz numa variável global do sistema. O diretório corrente de um processo e o diretório-raiz podem ser modificados através das chamadas ao sistema **`chdir`** e **`chroot`**, respectivamente. Cada usuário tem também o diretório-raiz-do-usuário (*home directory*), que é o diretório *default* quando ele acessa o sistema. É possível se referir ao *home directory* de qualquer usuário com a sequência "~nome_de_login". O diretório corrente pode ser referenciado por ".", enquanto que o "pai" do diretório corrente pode ser referenciado por "..".

Em cada diretório, o núcleo faz uma busca linear pelo próximo componente do nome. Quando o nome é localizado, o seu Inode é guardado. E assim sucessivamente até chegar ao último componente do nome, ou seja ao Inode do arquivo. Como o sistema UNIX admite atribuir mais de um nome simbólico (caminho de acesso) a um mesmo arquivo através da chamada de sistema link, os diretórios formam uma estrutura do tipo **grafo dirigido**.

Quando há mais de um nome simbólico atribuído a um arquivo, o núcleo não guarda qual é o nome original. Portanto, todos os nomes têm o mesmo tratamento. Para evitar que o Inode de um arquivo "compartilhado" seja removido antes da remoção de todos os seus caminhos de acesso, há no Inode carregado um campo para guardar o número de *links* apontados para ele. Assim, ao usar a chamada de sistema **`unlink`** para remover um nome simbólico, o núcleo só removerá efetivamente o Inode correspondente quando o número de *links* for igual a zero.

Os blocos de dados de um arquivo-diretório são organizados de forma similar aos dos arquivos regulares. Este arquivo se diferencia dos arquivos regulares em relação a:

- conteúdo dos seus blocos de dados: é uma sequência de pares (número de Inode, nome simbólico do arquivo);
- semântica de permissões de acesso: como para os arquivos regulares, as permissões de leitura e de escrita significam, respectivamente, possibilidades de leitura e escrita no diretório; mas a permissão de execução significa que o diretório pode ser "varrido", isto é, é possível usar o comando `cd` e tornar este diretório o diretório corrente, por exemplo; e
- permissões: normalmente só o super-usuário pode criar mais de uma entrada na estrutura de diretórios para os arquivos-diretório (fazer *link*). Isso procura evitar acessos cíclicos.

No sistema UNIX é permitido, no nível de processos, acessar um mesmo arquivo regular sob diferentes modos (de escrita, de leitura ou de escrita e leitura) independentemente. Esta independência é conseguida através do acréscimo de dois níveis de encadeamento entre o processo e o Inode do arquivo que ele acessa:

- **tabela de descritores de arquivos do usuário**: é uma tabela privada de cada processo. O índice de cada entrada é sempre retornado ao usuário e é denominado **descritor de arquivo** (*file descriptor*). O conteúdo de cada entrada é um apontador para a tabela de arquivos.
- **tabela de arquivos**: é uma tabela de todos os arquivos em uso pelo sistema. Cada entrada contém um apontador ao Inode, o modo de acesso e o endereço lógico do próximo acesso do arquivo correspondente.

A combinação do uso das duas tabelas permite que um mesmo Inode seja referenciado pelo mesmo descritor de arquivo em distintos processos. É importante mencionar que o valor alocado para o descritor de arquivo é **sempre** o valor mais baixo disponível, contando a partir de zero. Os descritores dos três dispositivos (canais) padrão de E/S **`stdin`**, **`stdout`** e **`stderr`** recebem os valores 0, 1 e 2, respectivamente.

Uma entrada é alocada na tabela de descritores e na de arquivos quando ocorrem chamadas ao sistema **open** (abrir ou criar um arquivo) ou **`creat`** (equivalente à chamada **open** com a opção O_WRONLY | O_CREAT | O_TRUNC). Há ainda a chamada de sistema que só aloca uma nova entrada na tabela de descritores: **dup**. Essa chamada duplica o conteúdo de uma entrada da tabela de descritores escrevendo-o na primeira entrada livre da tabela. Embora o uso dela requeira o conhecimento da manipulação interna, essa chamada provê facilidades para implementação de comandos mais sofisticados como **`pipe`** e de redirecionamento de entrada ou saída.

Dispondo do descritor de um arquivo **`<fd>`**, as seguintes chamadas de sistema podem ser utilizadas para acessá-lo ou manipulá-lo:

- **read(fd, buffer, n)**: copiar do <fd>, a partir do último endereço de acesso guardado na tabela de arquivos, <n> *bytes* para o endereço especificado em <buffer>;
- **write(fd, buffer, n)**: copiar <n> *bytes* do endereço especificado em <buffer> para o último endereço de acesso guardado na tabela de arquivos do <fd>;
- **lseek(fd, desloc, ref)**: modificar somente o último endereço de acesso da tabela de arquivos do <fd> de acordo com o número <desloc> de *bytes* de deslocamento em relação ao ponto de referência <ref> (o início, a posição corrente ou o fim do arquivo);
- **close(fd)**: a entrada do <fd> na tabela de descritores é removida e, se o número de referências ao arquivo na tabela de arquivos e na lista de Inodes for igual a um, as entradas nestas estruturas são removidas.

Para monitorar os arquivos do sistema, o UNIX dispõe das chamadas **stat(nome, buffer)** (que aceita como argumento o nome simbólico) e **fstat(fd, buffer)** (que aceita como argumento o descritor de arquivo) para obter informações de um arquivo, como estado, tipo, proprietário, tamanho, quantidade de *links*, número do Inode, permissões de acesso e horários dos últimos acessos. Para ler o conteúdo de um arquivo-diretório pode-se utilizar a chamada de sistema **`getdents`**.

Outras chamadas úteis para manipular arquivos através do seu nome simbólico <nome> ou do seu descritor <fd> são:

- **chown(nome, pId, gId)**: mudar o proprietário do arquivo para (pId, gId), onde pId e gId denotam o identificador do usuário e do grupo, respectivamente,
- **chmod(nome, modo)**: mudar os modos de acesso (permissões de escrita, leitura e/ou execução para usuário, grupo e/ou todos os usuários),
- **fcntl(fd, cmd, arg)**: executar o comando <cmd> com os argumentos <arg> sobre o arquivo <fd>.

Observe que, em geral, estas chamadas usam descritores de arquivos em lugar de apontadores para arquivos. Este procedimento aumenta muito a eficiência na transmissão de grandes quantidades de dados, mas usam operações de "baixo nível", que devem ser empregadas com maior cuidado que as funções de E/S tradicionais **`fopen`**, **`fscanf`**, **`fgets`** e **`fputs`**, por exemplo.

3.3. Arquivos Especiais

O sistema UNIX pode suportar mais de um dispositivo de E/S. Esses dispositivos devem ser instalados como arquivos especiais no diretório </dev> através da chamada de sistema **`mknod`** para que eles sejam

incorporados na estrutura de diretórios e processáveis como arquivos regulares. Assim, as chamadas de sistema **`lseek, chmod`** e **`stat`**, definidos para os arquivos regulares, aplicam-se diretamente a estes dispositivos, porque essas chamadas só precisam das informações sobre os Inodes e outros dados carregados na memória.

Os arquivos especiais são diferenciados dos outros arquivos através do tipo de arquivo especificado nos seus Inodes. Como os arquivos especiais fazem referência aos dispositivos e não aos blocos de dados, não faz sentido falar em armazenar o tamanho de arquivo e os endereços físicos dos blocos de dados nos seus Inodes. No lugar dessas informações, temos dois números de dispositivo - um maior e um menor. O maior indica a classe do dispositivo (por exemplo, terminal, disco ou fita) e o menor, a unidade de uma determinada classe. Através dessas informações o núcleo mapeia internamente as chamadas de sistema correspondentes aos arquivos regulares em rotinas de operações de E/S específicas de cada dispositivo.

Quando se trata de acessos físicos, notam-se as seguintes diferenças operacionais entre os arquivos regulares e os especiais:

- embora as chamadas de sistema **`open`**, **`read`**, **`write`** e **`close`** sejam as mesmas para os arquivos regulares e especiais, quando se trata dos arquivos especiais o sistema precisa mapear internamente através das **tabelas de chaveamento** (*block/character device switch table*) essas funções em chamadas processáveis pelo *driver* de cada dispositivo específico;
- para os arquivos especiais do tipo caractere pode-se utilizar a chamada de sistema **`ioctl`** para adequar a interface às características de *hardware* de cada dispositivo específico, como setar a velocidade do terminal;
- os arquivos especiais do tipo bloco aceitam a criação de novos sistemas de arquivos em suas partições através do comando de sistema `newfs` ou `mkfs` e esses sistemas de arquivos, por sua vez, podem ser acessados via a única estrutura de diretórios disponível no sistema por meio da chamada de sistema **`mount`**. A chamada de sistema `mount` agrega qualquer sistema de arquivos à estrutura existente através da associação de um diretório da estrutura existente com o Inode-raiz do novo sistema. O diretório utilizado para a conexão é denominado o **ponto de montagem** (*mount point*). As informações sobre essas conexões são armazenadas na **tabela de montagem** (*mount table*). Cada entrada da tabela de montagem contém os seguintes campos:
  - o número do arquivo especial que contém o sistema de arquivos agregado,
  - um apontador ao Inode-raiz do sistema de arquivos agregado,
  - um apontador ao Inode correspondente ao diretório da estrutura existente (este Inode é marcado como o ponto de montagem) e
  - um apontador ao endereço do superbloco do sistema de arquivos carregado.

Um sistema de arquivos pode ser também desconectado da estrutura de diretórios através da chamada de sistema **`umount`**.

4. Superblocos

Pelo que vimos, quando se cria um novo arquivo regular, um Inode e um conjunto de blocos de dados devem ser alocados. Uma forma de alocação seria percorrer sequencialmente a lista de Inodes e a de blocos de dados do sistema de arquivos e localizar as células livres. Essa estratégia é computacionalmente cara e requer no mínimo uma operação de E/S. Para melhorar o desempenho o sistema de arquivos mantém no superbloco (ele é carregado na memória e a versão no disco é atualizada periodicamente) uma lista de Inodes disponíveis e uma lista de blocos de dados disponíveis além das seguintes informações:

- o tamanho do sistema de arquivos;
- o número de blocos de dados livres no sistema de arquivo;
- o indicador do próximo bloco de dados livre no sistema de arquivos;
- o tamanho da lista de Inodes;
- o número de Inodes livres no sistema de arquivos;
- o indicador do próximo Inode livre na lista de Inodes livres;
- indicadores do modo de acesso às listas de blocos de dados e de Inodes livres (bloqueado ou não); e
- indicador do estado do superbloco (modificado ou não).

Para evitar uma lista muito longa de Inodes no superbloco, é estabelecido um limite máximo de número de células na lista de Inodes livres deste superbloco. Quando essa lista fica vazia, o núcleo percorre a lista de Inodes da partição e coleta novos Inodes disponíveis, procurando preencher a lista do superbloco novamente. Para otimizar a coleta, o sistema armazena sempre o último Inode visitado, a partir do qual é iniciado o próximo processo de coleta, e quando um Inode é liberado (por exemplo, um arquivo é removido do sistema) o núcleo adota a seguinte estratégia para manter essa organização consistente: compara o número do Inode liberado com o número do último Inode visitado e se o número do Inode liberado for menor, o Inode liberado será considerado como o último Inode visitado.

Quanto à lista de blocos de dados, o sistema adota uma outra política para agilizar a manipulação de blocos livres. Isso se deve ao fato de que o sistema não consegue distinguir um bloco ocupado de um bloco livre

somente pelo seu conteúdo. É necessário, portanto, manter uma lista integral de blocos disponíveis. O sistema mantém uma lista de blocos com os números de blocos livres no disco e os primeiros blocos desta lista são carregados na memória como sendo a lista de blocos de dados livres do superbloco. Quando todos os números dos blocos desta lista do superbloco forem alocados, o próximo bloco será carregado e assim sucessivamente, até esgotarem todos os blocos de dados livres do sistema de arquivos. Caso algum bloco de dado seja liberado, o seu número é inserido na lista de blocos de dados disponíveis.

Como um sistema de arquivos pode ser acessado por mais de um processo simultaneamente, pode ocorrer o problema de *race condition*. Para evitar isso é provido um indicador de estado do sistema de arquivos (bloqueado ou não) para garantir a exclusão mútua dos acessos a ele.

5. Atividades Práticas

5.1 Atividades em sala

Todas as atividades devem ter seu funcionamento documentado e explicado no relatório, de acordo com o que é solicitado no enunciado de cada uma. **Destaque o papel das chamadas de sistema relativas ao gerenciamento de arquivos** e **explique como cada programa faz para atingir seus objetivos**. Alguns programas vão criar arquivos novos e outros utilizarão arquivos já existentes, os quais receberão a denominação teste*.*. Compile um arquivo `xyz.c` com a opção `-o xyz` , de forma a preservar o nome do arquivo fonte no arquivo executável. Alguns destes arquivos estão disponíveis na página da disciplina.

Obs: para acessar as *man pages* das chamdas de sistema *write*, *stat*, etc, deve-se usar o comando `man -s 2 nome_da_chamada`, de forma a não haver confusão com os comandos `write` e `stat` do shell (a chave -s 2 indica seção 2 do manual).

(a) (1,0) Criação, abertura e fechamento de arquivos

(a.1) Após a compilação e a execução do programa `a.c`, documente a saída obtida (incluindo arquivos gerados e/ou modificados) e explique a diferença entre as chamadas `creat` e `open` .

(a.2) É possível substituir a chamada `creat` por uma chamada `open` com resultado equivalente? Como?

(a.3) Explique ainda como é possível estabelecer acesso diferenciado a um mesmo arquivo.

(a.4) Quais são as causas dos erros verificados?

```
/* a.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
int fd1, fd2, i;
char buf[30];

    if ((fd1 = creat("teste.a", 0600)) == -1)
        printf("Erro na 1a. operacao = %d\n", errno);
    if ((i = write (fd1, "Este e' o programa de teste 1", 29)) == -1)
        printf("Erro na 2a. operacao = %d\n", errno);
    else
        printf("Foram escritos %d bytes.\n", i);
    if (read(fd1, buf, sizeof(buf)) == -1)
        printf("Erro na 3a. operacao = %d\n", errno);
    if ((fd2 = open("teste.a", O_RDONLY, 0600)) == -1)
        printf("Erro na 4a. operacao = %d\n", errno);
    if ((i = read(fd2, buf, sizeof(buf))) == -1)
        printf("Erro na 5a. operacao = %d\n", errno);
    else
        printf("Foram lidos %d bytes.\n", i);
    if (write(fd2, "/0", 1) == -1)
        printf("Erro na 6a. operacao = %d\n", errno);
    close(fd1);
    close(fd2);
    exit(0);
}
```

(b) (1,0) Acesso de leitura e de escrita a um mesmo arquivo por distintos processos com execução em *background* (segundo plano)

(b.1) Após a compilação dos dois programas abaixo, execute `b1`, e documente a saída obtida (incluindo arquivos gerados e/ou modificados), justificando os resultados observados.

(b.2) Após a compilação dos dois programas abaixo, execute `b2`, e documente a saída obtida (incluindo arquivos gerados e/ou modificados), justificando os resultados observados.

(b.3) Execute o comando `b2&`, seguido imediatamente do comando `b1&` e documente a saída obtida (incluindo arquivos gerados e/ou modificados), explicando como ela foi gerada pelos programas.

(b.4) Explique como é possível estabelecer neste caso acesso "simultâneo" a um mesmo arquivo.

/* b1.c */

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

/* Processo 1 (read) */

int main()
{
int fd, i;
char buf[512];

    if((fd = open("teste.b", O_EXCL|O_CREAT|O_RDONLY, 0600)) == -1)
    {
        if((fd = open("teste.b", O_RDONLY, 0600)) == -1)
        {
            printf("Erro na abertura de arquivo\n");
            exit(errno);
        }
    }
    if((i = read(fd, buf, sizeof(buf))) == -1)
    {
        printf("Erro na leitura de arquivo\n");
        exit(errno);
    }
    printf("1a. Leitura: Foram lidos %d bytes.\n", i);
    for (i=0; i<sizeof(buf); i++)
        printf("%c", buf[i]);
    printf("\n");
    sleep(5);
    if((i = read(fd, buf, sizeof(buf))) == -1)
    {
        printf("Erro na leitura de arquivo\n");
        exit(errno);
    }
    printf("2a. Leitura: Foram lidos %d bytes.\n", i);
    for (i=0; i<sizeof(buf); i++)
        printf("%c", buf[i]);
    printf("\n");
    close(fd);
    exit(0);
}
```

/* b2.c */

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

/* Processo 2 (write) */

int main()
{
int fd, i;
char buf[512];
```

```
    if((fd = open("teste.b", O_EXCL|O_CREAT|O_WRONLY, 0600)) == -1)
    {
        if((fd = open("teste.b", O_WRONLY, 0600)) == -1)
        {
            printf("Erro na abertura de arquivo\n");
            exit(errno);
        }
    }
    for (i=0; i < sizeof(buf); i++)
        buf[i] = '*';
    if((i = write(fd, buf, sizeof(buf))) == -1)
    {
        printf("Erro na escrita em arquivo: %d\n", errno);
        exit(errno);
    }
    printf("1a. Escrita: Foram escritos %d bytes.\n", i);
    sleep(5);
    if((i = write(fd, buf, sizeof(buf))) == -1)
    {
        printf("Erro na escrita em arquivo: %d\n", errno);
        exit(errno);
    }
    printf("2a. Escrita: Foram escritos %d bytes.\n", i);
    close(fd);
    exit(0);
}
```

(c) (1,0) Trabalhando com o último endereço de acesso

(c.1) Após a compilação, execute o programa `c.c`, documente a saída e explique seu funcionamento.

(c.2) Utilize os comandos `<ls -l testec?.txt>` e `<ls -s testec?.txt>` para verificar o tamanho em *bytes* e a quantidade de blocos alocados para os arquivos `<testec1.txt>` e `<testec2.txt>`. Explique porque os parâmetros `-l` e `-s` de `ls` retornam valores distintos para um mesmo arquivo.

(c.3) Explique por que há uma diferença no número de blocos dos dois arquivos.

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

int main()
{
int fd, i;

    if((fd = open("testec1.txt", O_CREAT|O_WRONLY|O_TRUNC, 0600)) == -1)
    {
        printf("Erro na abertura de arquivo\n");
        exit(errno);
    }
    write (fd, "Este e' o arquivo de teste c1: ", 31);
    lseek (fd, 20031, SEEK_SET);  /*  consulte man !  */
    write (fd, " fim do arquivo testec1.txt\n", 28);
    close (fd);

    fd = creat("testec2.txt", 0600);
    write (fd, "Este e' o arquivo de teste c2: ", 31);
    for (i=0; i<20000; i++)
        write(fd,"$",1);
    write (fd, " fim do arquivo testec2.txt\n", 28);
    close (fd);
    exit(0);
}
```

5.2 Atividades para entrega na próxima semana

(d) (1,0) Duplicação de descritores

(d.1) Após a compilação, execute o programa `d.c` com o comando `d teste.d` (o arquivo `teste.d` é fornecido) e documente a saída.

(d.2) Explique a saída do programa a partir da função desempenhada pelas variáveis `i` e `j`. Você pode usar um programa como o `xxd` (mostra conteúdo de um arquivo em hexadecimal e em código ASCII) para visualizar de maneira mais completa o conteúdo de `teste.d`. Uso: `> xxd teste.d`

```
/* d.c */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
int i, j, k;
char buf[50];

    if((i = open(argv[1], O_RDONLY, 0600)) == -1)
    {
        printf("Erro na abertura de arquivo\n");
        exit(errno);
    }

    j = dup(i);

    read(i, buf, sizeof(buf));
    printf("1a. Leitura:\n");
    for (k=0; k<50; k++)
        printf("%c",buf[k]);
    printf("\n");

    read(j, buf, sizeof(buf));
    printf("2a. Leitura:\n");
    for (k=0; k<50; k++)
        printf("%c",buf[k]);
    printf("\n");
    close(i);

    read(j, buf, sizeof(buf));
    printf("3a. Leitura:\n");
    for (k=0; k<50; k++)
        printf("%c",buf[k]);
    printf("\n");
    close(j);
    exit(0);
}
```

Conteúdo de `teste.d`:

```
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
9876543210
```

(e) (1,0) Redireção de arquivos

(e.1) Após a compilação de `e.c`, execute o programa com o comando `e teste1.e ls -s` e documente a saída. Observe que todos os quatro termos fazem parte do mesmo comando.

(e.2) Explique o que faz o comando executado.

(e.3) Em seguida, execute o comando `ls -s > teste2.e` e documente a saída.

(e.4) Explique o que faz o comando executado.

(e.5) Compare os conteúdos dos arquivos `<teste1.e>` e `<teste2.e>` e explique suas diferenças e o funcionamento do programa.

```
/* e.c */
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
int fd;
    fd = creat(argv[1], 0600);
    dup2(fd,1);
    close(fd);
    execvp(argv[2], &argv[2]);
}
```

(f) (1,0) Uso de *link* e *unlink*

(f.1) Após a compilação de `f.c`, execute o programa com o comando `f teste.d` (o arquivo `teste.d` é fornecido) e documente a saída.

(f.2) Por que falhou a chamada `<stat>` e não a chamada `<fstat>`?

(f.3) O que acontecerá se removermos a chamada `<link>`? Justifique.

(f.4) Existe a possibilidade de uma falha grave deste programa ao executar o último `printf()`. Explique que falha é essa e como ela pode ser evitada.

```
/* f.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fd, i;
    char buf[50];
    struct stat statinfo;

    if (argc != 2)
    {
        printf("Uso: comando <nome de arquivo>\n");
        exit(-1);
    }
    link(argv[1], "tmp");

    if ((fd = open("tmp", O_RDONLY, 0600)) == -1)
    {
        printf("Falha na abertura de arquivo\n");
        exit(-1);
    }

    if (unlink("tmp") == -1)
        exit(-1);

    if (stat("tmp", &statinfo) == -1)
        printf("Falha na chamada <stat>!\n");
    else
        printf("A chamada <stat> foi bem sucedida!\n");

    if (fstat(fd, &statinfo) == -1)
        printf("Falha na chamada <fstat>!\n");
```

```
    else
        printf("A chamada <fstat> foi bem sucedida!\n");

    i = read(fd, buf, sizeof(buf));
    printf ("Leu %d bytes: %s\n", i,buf);

    exit(1);
}
```

(g) (1,0) Lista de informações a partir de um Inode

(g.1) Teste o programa `g.c` com pelo menos 3 tipos de arquivos (arquivos regulares, diretórios e arquivos especiais no diretório /dev) e documente as saídas em cada caso.

(g.2) Explique o funcionamento do programa e justifique os resultados obtidos.

```
/* g.c */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>

#define ENUF(mensagem, valor) { perror(mensagem); exit(valor); }

int main(int argc, char *argv[])
{
struct stat statbuf;  /* definicao da estrutura do Inode */
char s[127];

    if (argc != 2)
    {
        sprintf(s, "Uso: %s <nome do arquivo>", argv[0]);
        ENUF(s,1);
    }
    if (stat(argv[1], &statbuf) == -1)
    {
        sprintf(s,"Nao e' possivel obter status de %s", argv[1]);
        ENUF(s,2);
    }
    fprintf(stdout, "\n%s e' um ", argv[1]);
    switch (statbuf.st_mode & S_IFMT)
    {
        case S_IFREG : fprintf(stdout, "arquivo regular \n"); break;
        case S_IFDIR : fprintf(stdout, "arquivo-diretorio \n"); break;
        case S_IFCHR : fprintf(stdout, "arquivo especial (caracter) \n");break;
        case S_IFBLK : fprintf(stdout, "arquivo especial (bloco) \n"); break;
    }
    fprintf(stdout, "\nuser-id %d, ",    statbuf.st_uid);
    fprintf(stdout, "group-id %d, ",     statbuf.st_gid);
    fprintf(stdout, "permissao %o, ",    statbuf.st_mode);
    fprintf(stdout, "link(s) %d \n",     statbuf.st_nlink);
    fprintf(stdout, "tamanho %d (bytes)",statbuf.st_size);
    fprintf(stdout, ", %d (blocos),",    statbuf.st_blocks);
    fprintf(stdout, "Inode # %d \n\n",   statbuf.st_ino);
    fprintf(stdout, "Dados de %s foram modificados pela ultima vez em %s",
          argv[1], ctime(&statbuf.st_mtime));
    fprintf(stdout, "\nLeitura ");
    fprintf(stdout, (access(argv[1], R_OK) == 0) ? "permitida\n" : "proibida\n");
    fprintf(stdout, "Escrita ");
    fprintf(stdout, (access(argv[1], W_OK) == 0) ? "permitida\n" : "proibida\n");
    fprintf(stdout, "Execucao ");
    fprintf(stdout, (access(argv[1], X_OK) == 0) ? "permitida\n" : "proibida\n\n");
}
```

(h) (1,0) Redireção de entrada/saída

(h.1) Teste este programa `h.c` e documente as saídas (incluindo arquivos gerados).

(h.2) Explique o funcionamento do programa e justifique os resultados obtidos.

```
/* h.c
   Redirecionamento de entrada/saida de dados
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
        int fd1, fd2;

        /* fecha stdin - proxima chamada open tera' descritor igual a 0 */
        close(0);

        /* abre arquivo que contem a entrada */
        fd1 = open("a.c", O_RDONLY, 0600);
        if(fd1 == -1) {
          perror("open fd1");
          exit(0);
        }

        /* fd1 devera ter valor 0 */
        printf("fd1: %d\n", fd1); fflush(stdout);

        /* fecha stdout - proxima chamada open tera' descritor igual a 1 */
        close(1);

        /* abre arquivo que contera a saida */
        fd2 = open("./teste.h", O_WRONLY | O_CREAT, 0600);
        if(fd2 == -1) {
          perror("open fd2");
          exit(0);
        }

        /* executa grep */
        system("grep \"open\"");

        /* fecha arquivos */
        close(fd1);
        close(fd2);
}
```

(i) (2,0) Projeto Servidor WEB

Um "espaço Web" (Web space) é um diretório virtual mantido por um servidor Web e acessível para os clientes do servidor. Por exemplo, quando solicitamos ao servidor o conteúdo do URL *http://www.unicamp.br/alunos/index.html* estamos solicitando o conteúdo do recurso *index.html* no diretório *alunos* localizado na raiz (/) do espaço Web mantido pelo servidor *www.unicamp.br*. O espaço Web é mapeado no sistema de arquivos do servidor, por exemplo, no diretório */home/web/*. Isto significa que a raiz de *http://www.unicamp.br* fica em /*home/web* e dentro deste diretório encontraremos o diretório *alunos* e o arquivo *alunos/index.html*.

Podemos especificar apenas um diretório do espaço Web e omitir o nome do recurso como, por exemplo, o diretório *alunos* na forma *http://www.unicamp.br/alunos* . Neste caso, o servidor procura no diretório especificado (*alunos* neste caso) o arquivo default *index.html* ou, na falta deste, o arquivo *welcome.html*. Caso nenhum destes arquivos exista, uma mensagem de erro (404 Not Found) é enviada ao cliente (muitos servidores comerciais enviam uma página HTML com *links* para páginas de erros e/ou de alertas pré-fabricadas e armazenadas no servidor contactado).

Para o projeto do servidor http, o espaço Web que será disponibilizado para o mundo exterior poderá ser um diretório qualquer na área *home* do aluno. Portanto, a raiz do espaço Web será mapeada neste diretório.

Como atividade de projeto, implemente uma função em C que receba como argumento duas cadeias de caracteres (strings), a primeira contendo o caminho completo para o espaço Web mantido pelo servidor (e que é um diretório em algum lugar de sua área home) e a segunda, o recurso solicitado (subdiretórios e o nome do arquivo desejado).

A função deve retornar um inteiro e possuir a seguinte lógica:

1. combinar o caminho do espaço Web com o recurso solicitado;
2. acessar o recurso (arquivo ou diretório) com a chamada *stat*. Caso o recurso não exista, retornar o código de erro HTTP correspondente a "Not Found";
3. verificar se o recurso possui permissão de leitura. Caso não possua, retornar o código de erro HTTP correspondente a "Forbiden";
4. verificar se o recurso é um arquivo ou diretório. Caso seja um arquivo, abrir o arquivo com a chamada *open*, imprimir seu conteúdo na tela com a chamada *write* e retornar 0;
5. caso o recurso seja um diretório, verificar se ele possui permissão de varredura (bit x de execução setado). Caso não possua, retornar o código de erro HTTP correspondente a "Forbiden";
6. caso este diretório possua permissão de varredura, verificar se existe o arquivo *index.html* ou, na falta deste, *welcome.html*. Caso nenhum destes arquivos exista, retornar o código de erro HTTP correspondente a "Not Found". Caso um ou mais deles existam mas ambos estão sem permissão de leitura, retornar o código de erro HTTP correspondente a "Forbiden". Caso contrário (existe pelo menos um com permissão de leitura), abrir o arquivo com a chamada *open*, imprimir seu conteúdo na tela com a chamada *write* e retornar 0.

O relatório deve conter a listagem do código da função bem documentado e exemplos de teste para, pelo menos, cada uma das situações possíveis descritas acima, mostrando que todas as possibilidades foram tratadas.