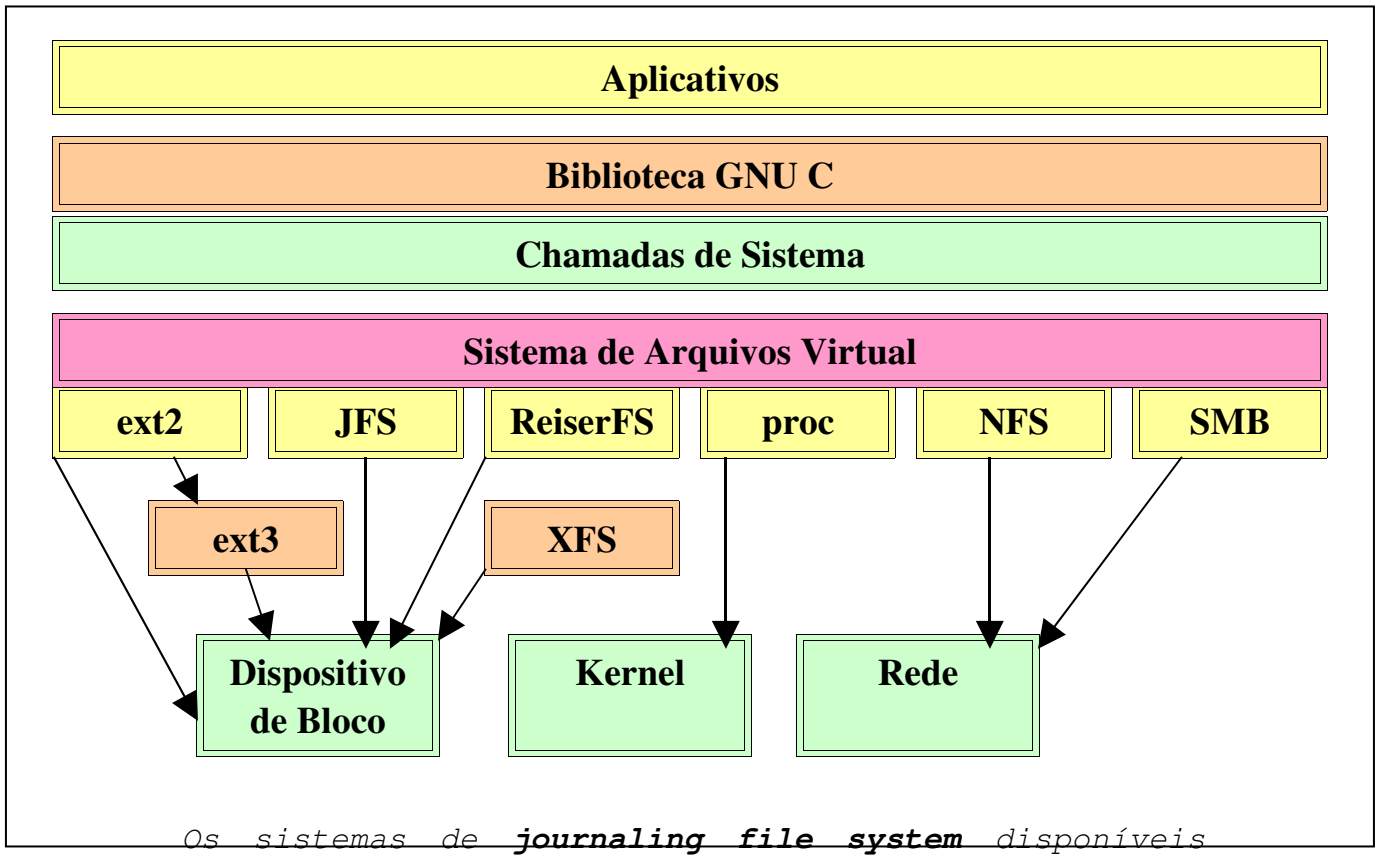
VI.2- ESTUDO DA IMPLEMENTAÇÃO

*A **necessidade de guardar informações em objetos** que pudessem ser **acessados** posteriormente de uma **forma organizada** vem de muito longe nos estudos do SOs e para isto foram criados os **sistemas de arquivos.***

*Nos sistemas de arquivos de uma forma geral, os **objetos são organizados de uma forma hierárquica** e cada um possui uma **identificação única** dentro de uma tabela. Já os **arquivos** são **entidades lógicas** que possuem **informações definidas** e são **mapeados pelo SO em dispositivos físicos.***

*O **kernel do Linux** suporta nativamente **quatro tipos** de sistemas de arquivos avançados que possibilitam o armazenamento de dados de forma transacional. Da mesma forma que um banco de dados transacional, o sistema de arquivos irá tratar as alterações dos dados como **uma simples operação atômica,** que será registrada em um **sistema de metadados.** Isto garante que todos os dados sejam **alterados com consistência** em **caso de falha,** que **nenhuma alteração seja feita,** método transacional conhecido como **journaling file system.***

ORGANIZAÇÃO DO SISTEMA DE ARQUIVOS NO SISTEMA



*Os sistemas de **journaling file system** disponíveis para Linux: **o JFS da IBM, o XFS da SGI, o RaiserFS da Namesys e o ext3** nascido das melhorias do padrão ext2 são todos de*

código aberto *e* ***organizados como pontos independentes*** *que acessam os dispositivos de bloco, permitindo que mais de um* ***sistema de arquivos seja usado ao mesmo tempo.***

A identificação dos objetos de um sistema de arquivos no Linux é conhecida como ***inode,*** *que carrega as informações de onde o objeto está* ***localizado no disco,*** *informações de* ***segurança, data e hora de criação*** *e* ***última modificação,*** *dentre outras. Quando criamos um sistema de arquivos no Linux, cada dispositivo tem um* ***número finito de inodes*** *que será diretamente proporcional ao número de arquivos que este dispositivo poderá acomodar.*

Basicamente o Linux tem dois objetos no seu sistema de arquivos: ***arquivos e diretórios.***

Os ***arquivos*** *contêm* ***informações dos usuários,*** *como* ***textos, scripts, um código binário,*** *etc.*

Os ***diretórios*** *são* ***arquivos especiais*** *mantidos pelo sistema e que implementam a* ***estrutura do sistema de arquivo*** *e contém* ***nomes de arquivos*** *e os* ***endereços*** *nos quais cada arquivo está armazenado nos* ***periféricos.***

Comparando com o Windows, *a organização dos diretórios do* ***Linux*** *é um pouco* ***mais complexa.*** *O seu sistema de arquivos é semelhante a uma* ***árvore de cabeça para baixo.*** *No topo da hierarquia do Linux existe o diretório raiz nomeado simplesmente como* ***root*** *e identificado como o sinal* ***"/".***

Toda a estrutura de diretórios do sistema é criada abaixo do root:

/	*Diretório raiz do sistema de arquivos;*
/bin	*Arquivos executáveis, especialmente comandos essenciais ao sistema;*
/boot	*Arquivos estáticos necessários à carga do sistema. É onde fica localizada a imagem do kernel;*
/dev	*Diretório onde ficam os arquivos para acesso aos dispositivos do sistema, como discos, cd-roms, disquetes, portas seriais, terminais, etc;*
/etc	*Arquivos necessários à configuração do sistema, são únicos e necessários para a carga;*
/home	*Geralmente é neste diretório onde ficam os diretórios locais dos usuários;*
/lib	*Arquivos de bibliotecas essenciais ao sistema, utilizados pelos programas em /bin e módulos do kernel;*
/mnt	*Diretório vazio utilizado como ponto de montagem de dispositivos na máquina;*
/proc	*Informações do kernel e dos processos;*
/opt	*Neste diretório ficam instalados os aplicativos que não vêm na distribuição do Linux;*
/root	*Diretório local do superusuário, que dependendo da distribuição pode estar presente ou não;*

/sbin	*Arquivos essenciais ao sistema, como aplicativos e utilitários para a administração da máquina, sendo que normalmente só o superusuário tem acesso a estes arquivos;*
/tmp	*Diretório de arquivos temporários, onde seu conteúdo é apagado a cada reboot;*
/usr	*Arquivos pertencentes aos usuários e à segunda maior hierarquia de diretórios do Linux;*
/var	*Diretórios onde são guardadas informações variáveis sobre o sistema em geral como: arquivos de log, arquivos de e-mail, entre outros;*

Numa estrutura hierárquica como esta, os nomes dos arquivos podem ser ***absolutos*** *(quando todo o caminho de acesso desde a raiz até o arquivo for conhecido. Exemplo: /usr/include/stdio.h)* ***ou relativos*** *(se o nome do arquivo não contiver a informação completa).*

Os nomes de arquivos e diretórios no Linux são ***sensíveis as letras maiúsculas e minúsculas****, assim o arquivo* ***Teste*** *é diferente de* ***teste*** *e* ***TESTE****, como as possíveis variações da palavra* ***teste****.*

Não é comum *no Linux o* ***uso de extensões****, mas são as* ***permissões que indicam se ele é executável*** *e, portanto, não necessitam de terminar com* ***.exe****, sendo que os* ***dois primeiros bytes*** *no início de cada arquivo identificam que* ***tipo*** *de arquivo ele é.*

A estrutura do sistema de arquivo do Linux é definida por um padrão de mercado chamado ***Filesystem Hierarchy Standard ou FHS****, criado pela comunidade Linux em* ***1974****, sendo as* ***distribuições não são obrigadas a seguir este padrão, mas elas entendem a importância da localização dos arquivos e diretórios padronizados****.*

O SO oferece ao usuário uma interface, sob a forma de ***procedimentos de bibliotecas****, que permitem a manipulação de arquivos. Em C, o tratamento de arquivos está declarado nos cabeçalhos* ***stdio.h, unistd.h, sys/types.h e sys/stat.h****.*

A alocação de espaço nos dispositivos físicos para armazenar os dados pode ser implementada de duas maneiras: ***alocação contígua*** *(mais simples de implementar, pois os arquivos estão gravados em blocos consecutivos, possui sérios problemas de fragmentação com áreas não utilizadas)* ***e lista encadeada*** *(grava os dados de um arquivo em uma lista encadeada de blocos, sendo que um bloco contém os dados e um apontador para o endereço do próximo bloco, assim para acessar uma informação é necessário percorrer a lista dos blocos).*

No sistema de arquivos do Linux a representação interna dos arquivos é fornecida pelos ***i-nodes****, que contem a*

descrição dos arquivos, *cada arquivo tem pelo menos um i-node e mais de um nome pode ser associado a ele.*

Todo acesso feito a um arquivo no Linux é feito através de ***um descritor de arquivo****, de forma que os processos mantêm sua própria tabela de referências aos descritores de arquivo. Os três primeiros descritores da tabela são reservados:*

0 ou stdin	*O descritor 0 é usado como descritor para a entrada padrão usado pelo processo;*
1 ou stout	*O descritor 1 é usado para saída padrão do programa;*
2 ou stderr	*O descritor 2 é usado para a saída de erros padrão do processo;*

A primitiva ***getcwd()*** *retorna o nome absoluto do diretório de trabalho associado ao processo e a conta do usuário.*

*char * getcwd (char *buffer, size_t size)*

O nome absoluto do diretório de trabalho será gravado no array de caracteres buffer, fornecido como argumento e seu tamanho, no argumento size. Um ponteiro nulo também pode ser fornecido no lugar do buffer permitindo que a função aloque um ponteiro automaticamente.

```
#include <unistd.h>
int main()
{
 char *diretorio = getcwd (NULL,0);
 printf("O diretório de trabalho é: %s.\n",diretorio);
 exit(0);
}
```

Para compilar este programa salvo como "getcwd.c":

#gcc getcwd.c –o getcwd

O resultado da execução é:

 # ./getcwd

O diretório de trabalho é: /root/sistemas.

A primitiva ***opendir()*** *abre um diretório no argumento dirname para leitura.* ***Dir * opendir (const char *dirname)***

Em caso de erro a função irá retornar um ponteiro nulo, do contrário a função irá retornar um fluxo de dados do tipo diretório que pode ser lido pela função readdir() e para ler o conteúdo de um diretório é necessário fazer referências aos símbolos declarados no cabeçalho ***dirent.h.***

A função ***readdir()*** *lê a próxima entrada de um diretório e grava em um ponteiro para uma estrutura que contém informações sobre o diretório.*

struct dirent * readdir (DIR *dirstream)*, caso não houver mais entradas em um diretório a função irá retornar um ponteiro nulo.*

*A primitiva **closedir()** fecha o descritor de um diretório aberto pela função opendir() que irá retornar 0 em caso de sucesso –1 em caso de erro.*

int closedir (DIR *dirstream)

O seguinte programa imprime o conteúdo de um diretório:

```
#include <stddef.h>
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
int
main (void)
{
 DIR *dp;
 struct dirent *ep;
 dp = opendir ("/var/log");
 if (dp != NULL)
  {
   while (ep = readdir(dp))
    puts (ep->d_name);
   (void) closedir (dp);
  }
 else
  puts ("Não foi possível abrir o diretótio.\n");
 exit(0);
}
```

Para compilar o programa salvo como “lerdiretorio.c”:

#gcc lerdiretorio.c –o lerdiretorio

O resultado da execução é:

```
# ./lerdiretorio
mail
news
suid
warn
wtmp
messages – 20040517.gz
```

Neste exemplo a função opendir() irá gravar o conteúdo do diretório no ponteiro dp do tipo DIR, já o ponteiro ep é apontado para a estrutura dirent e será carregado pela função readdir(ep).

Em sistemas POSIX, um arquivo pode ter mais de um nome, todos reais, válidos e podem estar localizados em diretórios diferentes. Um nome para um arquivo pode ser adicionado com a função link().

*A primitiva **link()** adiciona um nome à um arquivo criando um link físico.*

int link (const char *oldname, const char *newname)

Esta função recebe como parâmetro o nome antigo do arquivo e o novo nome a ser criado, irá retornar 0 em caso de

*sucesso e –1 em caso de erro, sendo possível criar um tipo de link chamado de **link simbólico**, que é outro arquivo que aponta para o arquivo original, de forma que este for apagado o link é quebrado.*

*A primitiva **readlink()** lê o nome do arquivo original apontado por um link simbólico, sendo que este valor é copia para a variável buffer, onde o argumento size define o número de caracteres que serão copiados e o argumento filename deve receber o nome do link simbólico.*

int readlin(const char *filename, char *buffer, size_t size)

Em caso de erro a função irá retornar –1.

Veja o exemplo:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main()
{
  printf("Um link simbólico será criado.\n");
  int linksimbolico = symlink("links.c","linksimbolico.c");
  if (linksimbolico==0) printf("Link simbólico criado.\n");
  printf("Um link físico será criado.\n");
  int linkfisico = link("links.c","linkfisico.c");
  if (linkfisico==0) printf("Link físico criado.\n");
  char nome[100];
  readlink("linksimbolico.c",nome,100);
   printf("O link simbólico linksimbolico.c aponta para o arquivo original:
%s\n",nome);
  exit(0);
}
```

Para compilar o programa salvo como “links.c”:

```
#gcc links.c –o links
```

O resultado da execução é:

```
# ./ links
   Um link simbólico será criado.
   Um link físico será criado.
   O link simbólico linksimbolico.c aponta para
   o arquivo original: links.c
```

Neste exemplo, um link simbólico linksimbolico será criado com a função symlink() apontando para o arquivo links.c, sendo que logo depois, um link físico será criado pela função link() e depois o programa irá ler o ponteiro linksimbolico.c para determinar o nome do arquivo original.

Liste os arquivos depois de executar este programa e observe as diferenças entre links simbólicos e físicos.

```
# ls –l
-rw-r-r-   2   root root 566  Jan 23 13:40 linksfisico.c
-rwxr-xr-x 2   root root 9086 Jan 23 13:40 links
-rw-r-r-   2   root root 566  Jan 23 13:40 links.c
```

lrwxrwxrwx 2 root root 7 Jan 23 13:39 linksimbolico.c

Observe que o link original físico linkfisico.c criado tem o mesmo tamanho e as mesmas características que o arquivo original links.c, pois apontam para a mesma região no disco. É como se um arquivo tivesse dois nomes. Já o link simbólico é um novo arquivo criado no disco que aponta para o arquivo original e se este for removido, o link não irá funcionar.

Pode-se apagar um arquivo com as funções ***unlink()*** *ou* ***remove()****. Este processo apaga somente o nome do arquivo, se tem um nome apenas, será removido, mas se outros nomes existirem, criados através dos links físicos, o arquivo permanecerá acessível pelo seu outro nome.*

A primitiva ***unlink()*** *apaga o nome de um arquivo, mas se este estiver em uso por algum processo, a remoção será adiada até que o arquivo seja fechado. Esta função está declarada no cabeçalho unistd.h.*

int unlink(const char *filename)

A primitiva remove() remove um arquivo ou diretório. Ela está declarada no cabeçalho stdio.h.

int remove(const char *filename)

Esta função retornará 0 em caso de sucesso e -1 em caso de erro.

A primitiva ***rmdir()*** *apaga um diretório vazio. Ela está declarada no cabeçalho unistd.h.*

int rmdir(const char *filename)

As primitivas unlink(), remove() e rmdir() recebem como parâmetro o nome do arquivo a ser removido filename.

A primitiva ***rename()*** *é usada para alterar o nome de um arquivo.*

int rename (const char *oldname, const char *newname)

Esta função recebe dois argumentos: ***oldname****, que indica o nome original do arquivo, e* ***newname****, o novo nome; se um arquivo já existir no sistema com o mesmo nome newname, a função irá sobrescrever o arquivo; se a operação for realizada em um diretório, o novo nome não poderá existir.*

A operação de mudança de nome é feita atomicamente e, portanto, não existe um instante entre oldname e newname; se o sistema travar durante a operação, pode acontecer de ambos os nomes existirem e esta função retornar -1 em caso de erro.

A primitiva ***mkdir()*** *cria um novo diretório especificado pelo argumento filename e com as permissões definidas em mode.*

int mkdir (const char *filename, mode_t mode)

Esta função irá retornar 0 em caso de sucesso e -1 em caso de erro.

As informações de tamanho, identificação do proprietário, data da alteração, data da criação são guardadas nos atributos de cada arquivo. Eles podem ser lidos

através das primitivas ***stat(), lstat() e fstat().*** *O resultado será carregado em uma estrutura especial chamada stat.*

O cabeçalho ***sys/stat.h*** *declara todos os atributos da estrutura stat, a seguir:*

```
struct stat {
 dev_t st_dev;                  /* dispositivo do inode */
 ino_t st_ino;                  /* número do inode */
 mode_t st_mode;                /* permissões do inode */
 nlink_t st_nlink;              /* números de links físicos */
 uid_t st_uid;                  /* ID do proprietário */
 gid_t st_gid;                  /* ID do grupo */
 dev_t st_rdev;                 /* tipo de dispositivo */
 struct timespec st_atimespec; /* data último acesso */
 struct timespec st_mtimespec; /* data última modificação */
 struct timespec st_ctimespec; /* data da mudança de status*/
 off_t st_size;                 /* tamanho arquivo em bytes */
 quad_t st_blocks;              /* número blocos alocados */
 u_long st_blksize;             /* tamanho ideal do bloco */
 u_long st_flags;               /* flags do arquivo */
 u_long st_gen;                 /* número do arquivo */
};
```

Algumas considerações sobre alguns membros desta estrutura:

mode_t st_mode	*Este membro define as permissões do arquivo, assim como seu tipo;*
ino_t st_ino	*Este membro define o número de série do arquivo, que o distingue de todo os outros arquivos de um mesmo dispositivo;*
dev_t dt_dev	*Define o dispositivo em que o arquivo se encontra;*
nlink_f st_nlink	*Define o número de links físicos que o arquivo tem. Deve haver pelo menos um link físico que é o seu nome e se este contador atingir 0, o arquivo é descartado;*
uid_t st_uid	*Define o ID da conta de usuário proprietária do arquivo;*
gid_t st_gid	*Define o ID do grupo de usuários proprietários do arquivo;*
off_t st_size	*Define o tamanho do arquivo em bytes;*
time_st st_atime	*Define a data do último acesso feito ao arquivo, modificado pela primitiva read();*
time_st st_mtime	*Define a data da última modificação feita do arquivo, modificado pela primitiva write();*
time_st st_ctime	*Define a data da última alteração dos atributos do arquivo, modificado pela primitiva write(), link() e rename();*
unsigned int st_nblocks	*Define o número de blocos que um arquivo ocupa. Nem sempre o tamanho definido em st_size é o*

	*tamanho do espaço em disco que ele ocupa, isto porque os blocos são divididos em um número de bytes e um arquivo pode ocupar um bloco e meio. Por exemplo, um arquivo de 700 bytes irá ocupar um bloco de 512 e uma parte de outro bloco. Aproximadamente, o tamanho real do arquivo é dado pelo número de blocos que ele ocupa vezes o tamanho do bloco. Ex: st_nblocks *512>st_size;*

O valor do membro st_mode é binário e pode ser lido e alterado através de algumas macros definidas no cabeçalho sys/stat.h a seguir:

símbolo	***octal***	***descrição***
S_IFIFO	*0010000*	*Verifica se o arquivo é um FIFO;*
S_IFCHR	*0020000*	*Verifica se o arquivo é um dispositivo de caracteres;*
S_IFDIR	*0040000*	*Verifica se o arquivo é um diretório;*
S_IFBLK	*0060000*	*Verifica se o arquivo é um dispositivo de blocos;*
S_IFREG	*0100000*	*Verifica se o arquivo é regular;*
S_IFLNK	*0120000*	*Verifica se o arquivo é um link simbólico;*
S_IFSOCK	*0140000*	*Verifica se o arquivo é um socket;*
S_ISUID	*0004000*	*Configurar o ID do usuário;*
S_ISGID	*0002000*	*Configurar o ID do grupo;*
S_ENFMT		*Trava o arquivo para escrita;*
S_IRWXU	*0000700*	*Configura as permissões de leitura, escrita e execução para o dono do arquivo;*
S_IRUSR	*0000400*	*Configura a permissão de leitura para o dono do arquivo;*
S_IREAD		*Configura permissões de leitura;*
S_IWUSR	*0000200*	*Configura permissão de escrita para o dono do arquivo;*
S_IWRITE		*Configura permissão de escrita;*
S_IXUSR	*0000100*	*Configura permissão de execução para dono do arquivo;*
S_IEXEC		*Configura permissão de execução;*
S_IRWXG	*0000070*	*Configura as permissões de leitura, escrita e execução para o grupo do arquivo;*
S_IRGRP	*0000040*	*Configura permissão de leitura para grupo do arquivo;*
S_IWGRP	*0000020*	*Configura permissão de escrita para grupo do arquivo;*
S_IXGRP	*0000010*	*Configura permissão de execução para grupo do arquivo;*
S_IRWXO	*0000007*	*Configura as permissões de leitura, escrita e execução para outros;*
S_IROTH	*0000004*	*Configura permissão de leitura para outros;*
S_IWOTH	*0000002*	*Configura permissão de escrita para outros;*
S_IXOTH	*0000001*	*Configura permissão de execução para outros;*

*A primitiva **stat()** retorna as informações dos atributos do arquivo indicado por filename no ponteiro buf que aponta para a estrutura stat.*

int stat (const char *filename, struct stat *buf)

Se o arquivo filename for um link simbólico, os atributos do arquivo apontado pelo link é que serão lidos.

Esta função retorna 0 em caso de sucesso e –1 em caso de erro.

*A primitiva **fstat()** funciona como a primitiva stat(), mas recebe como argumento um descritor de arquivos aberto ao invés do nome do arquivo.*

int fstat (int filedes, struct stat *buf)

O argumento filedes é um descritor de arquivos, onde a função irá retornar as informações dos atributos no ponteiro buf que aponta para estrutura stat. Esta função retorna 0 em caso de sucesso e –1 em caso de erro.

*A primitiva **lstat()** funciona como a primitiva stat(), mas não percorre um link simbólico para ler as informações do arquivo original.*

int lstat (int filedes, struct stat *buf)

Esta função retorna 0 em caso de sucesso e –1 em caso de erro.

Observe o exemplo:

```
#include <sys/types.h>
#include <sys/stat.h>
main(int argc,char *argv[])
{
     int    i,type;
     struct  stat   statinfo;
     char   *desc;
     for(i=1;i<argc;i++)
     {
          if(lstat(argv[i],&statinfo) == -1)
          {
               perror("lstat");
               exit(1);
          }
          type = statinfo.st_mode & S_IFMT;
          switch(type)
          {
               case S_IFDIR :
                         desc = "diretório";
                         break;
               case S_IFLNK :
                         desc = "link simbólico";
                         break;
               case S_IFREG :
                         desc = "arquivo normal";
                         break;
               default :
                         desc = "outro tipo";
          }
```

```
            printf("%s é um %s\n",argv[i],desc);
        }
    }
```

Para compilar o programa salvo como “listar.c”:

#gcc listar.c –o listar

O resultado da execução é:

```
# ./listar
LIVRO é um outro tipo
MYFIFO é outro tipo
assinatura é um arquivo normal
assinatura.c é um arquivo normal
links.c é um arquivo normal
linksimbolico.c é um arquivo simbólico
```

Neste exemplo os atributos do arquivo serão lidos pela função lstat() e carregados na struct statinfo do tipo stat e dependendo do valor de st_mode, o tipo de arquivo é escrito na tela e os símbolos S_IFDIR, S_IFLNK e S_IFREG são usados para fazer a identificação.

Observe outro exemplo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
int main(int argv, char* argc[])
{
 struct stat buffer;
 if(argv!=2)
  printf("Uso:\npermissoes <arquivo>\n");
 else{
  if(!stat(argc[1],&buffer))
  {
   printf("Arquivo: %s\n",argc[1]);
   printf("----------------------------------------\n");
   printf("ID do dono do arquivo : %d\n",buffer.st_uid);
   printf("ID do grupo do arquivo: %d\n",buffer.st_gid);
   printf("Permissões do arquivo : %o\n",buffer.st_mode);
  }
  else
  {
   printf("Erro ao ler atributos.\n");
  }
 }
 exit(0);
}
```

Para compilar o programa salvo como “permissoes.c”:

#gcc permissoes.c –o permissoes

O resultado da execução é:

 # ./ permissões

Arquivo: permissões.c

ID do dono do arquivo : 0
ID do grupo do arquivo: 0
Permissões do arquivo : 100644

Neste exemplo os dados do dono do arquivo e suas permissões são lidas com a função stat() e gravadas com o ponteiro buffer do tipo stat. O resultado é escrito na tela.

*A primitiva **creat()** é usada para criar um novo arquivo caso não exista, sendo que depois de criado, será aberto para escrita.*

```
#include <fildes.h>
int creat(const char *pathname, mode_t mode);
```

O argumento pathname deverá indicar o nome do arquivo a ser criado e argumento mode, as permissões de acesso do arquivo. Esta função retorna 0 em caso de sucesso e –1 em caso de erro.

*A primitiva **open()** também permite criar um arquivo novo ou abrir um arquivo existente.*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags,  mode_t mode);
```

O argumento pathname deverá indicar o nome do arquivo a ser aberto ou criado, o argumento mode é usado para determinar as permissões do arquivo a ser aberto e o argumento flags é utilizado para indicar a forma de abertura do arquivo. Esta função retorna 0 em caso de sucesso e –1 em caso de erro.

A tabela a seguir mostram os flags utilizados com a primitiva open().

octal	*Símbolo*	*Descrição*
00000	*O_RDONLY*	*Determina a abertura do arquivo somente para leitura;*
00001	*O_WRONLY*	*Determina a abertura do arquivo somente para escrita;*
00002	*O_RDWR*	*Determina a abertura arquivo para leitura e escrita;*
00004	*O_NDELAY*	*Impede o bloqueio do processo durante a leitura de um PIPE ainda não aberto para escrita;*
00010	*O_APPEND*	*Determina a escrita no fim do arquivo;*
00400	*O_CREAT*	*Determina que um arquivo seja criado caso não exista;*
01000	*O_TRUNC*	*Determina que o conteúdo arquivo seja zerado caso ele exista;*
02000	*O_EXCL*	*Determina que a função deverá retornar erro caso o arquivo exista, usado em conjunto com o flag O_CREAT;*

*A primitiva **read()** faz a leitura de count bytes de um descritor de arquivo aberto fd e grava o resultado no ponteiro buf.*

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Esta função retorna o número de bytes lidos ou 0 para indicar fim de linha e –1 em caso de erro.

A primitiva ***write()*** *faz a escrita de nbytes do ponteiro buf em um descritor de arquivos aberto fd.*

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

Esta função retorna o número de bytes escritos ou –1 em caso de erro.

A primitiva ***close()*** *fecha o descritor de arquivo fd, liberando o descritor, mas não esvaziando o buffer associado.*

#include <unistd.h>

int close(int fd);

Esta função irá retornar 0 em caso de sucesso e –1 em caso de erro.

Observe o exemplo:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <wait.h>
int main()
{
  int numbytes, fd;
  char *frase;
    printf("Um processo filho será criado para escrever algo em um arquivo.\n");
  if (fork()==0)
  {
    printf("\tUm arquivo será criado, escrito e lido.\n");
        fd = open("arquivotexte.txt", O_CREAT|O_WRONLY, S_IREAD|S_IWUSR);
    frase = "Este arquivo foi criado como exemplo escrita e leitura";
    numbytes = write(fd,frase,strlen(frase));
    printf("\t%d bytes foram gravados no arquivo.\n",numbytes);
    sleep(1);
    close(fd);
    exit(numbytes);
  }
  else
  {
    int status1;
    wait(&status1);
    char *resultado[WEXITSTATUS(status1)];
    fd = open("arquivotexte.txt", O_RDONLY, S_IREAD);
    printf("\tO arquivo será aberto para leitura.\n");
    numbytes = read(fd,resultado,WEXITSTATUS(status1));
    printf("\t%d bytes foram lidos no arquivo.\n",numbytes);
    puts((char *)&resultado);
```

```
   close(fd);
   exit(0);
  }
}
```

Para compilar o programa salvo como “arquivo.c”:

#gcc arquivo.c –o arquivo

O resultado da execução será:

 # ./arquivo

Um processo filho será criado para escrever algo em um arquivo.
Um arquivo será criado, escrito e lido.
75 bytes foram gravados no arquivo.
O arquivo será aberto para leitura.
75 bytes foram lidos no arquivo.
Este arquivo foi criado para exemplificar escrita e leitura de seu conteúdo.

Neste exemplo, o processo filho irá criar e abrir um arquivo para escrita através da função open() com os flags O_CREAT/O_WRONLY. Uma frase será gravada no arquivo com a função write() e terminará. O processo pai aguarda até que o processo filho termine. Depois o processo pai irá abrir o arquivo para leitura através da função open() com o flag O_RDONLY. O seu conteúdo será lido com a função read() e gravado na variável resultado, que será impressa na tela.