Sisteme de operare
Tema 3

Exercițiul 1

Rulați toate exercițiile prezentate și încercați să înțelegeți cum lucrează. Vedeți paginile de manual pentru funcțiile utilizate.

- Exercițiile prezentate:

```
[user@desktop-5p6viv2 destination]$ make lib
gcc -Wall -g -O   -c -o error.o error.c
ar rcs liblab3.a error.o
```

Am creat biblioteca liblab3.a prin comanda *make lib*.

```
[user@desktop-5p6viv2 destination]$ make
gcc -o seek seek.c liblab3.a
gcc -o hole hole.c liblab3.a
gcc -o mycat mycat.c liblab3.a
gcc -o fileflags fileflags.c liblab3.a
```

Am folosit comanda *make* pentru a compila exemplele.

```
[user@desktop-5p6viv2 destination]$ ./seek < /etc/motd
seek OK
[user@desktop-5p6viv2 destination]$ cat /etc/motd | ./seek
cannot seek
```

Prima linie de comandă verifică dacă putem folosi locația curentă pentru a scrie și a doua linie de comandă verifică dacă se poate citi.

```
[user@desktop-5p6viv2 destination]$ ./hole
[user@desktop-5p6viv2 destination]$ ls -l file.hole
-rwxrwxrwx. 1 root root 50 Mar 15 15:27 file.hole
[user@desktop-5p6viv2 destination]$ od -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F   G   H
0000060   I   J
0000062
```

Am creat un fișier cu o ”gaură” și după am afișat-o.

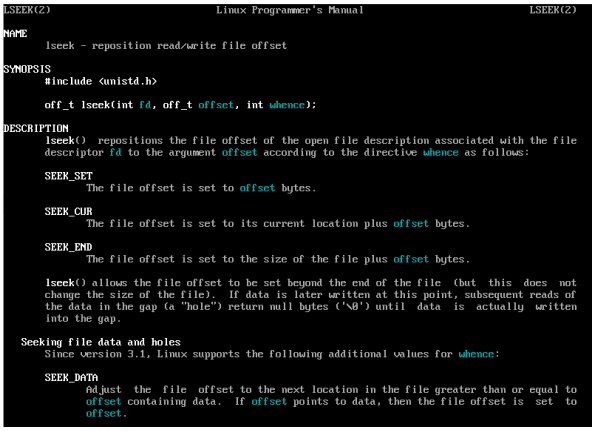
```
[user@desktop-5p6viv2 destination]$ ./mycat < file.hole
abcdefghijABCDEFGHIJ[user@desktop-5p6viv2 destination]$
```

Am copiat standardul input la standardul output.

```
[user@desktop-5p6viv2 destination]$ ./fileflags 0 < /dev/tty
read only
[user@desktop-5p6viv2 destination]$ ./fileflags 1 > temp.foo
[user@desktop-5p6viv2 destination]$ cat temp.foo
write only
[user@desktop-5p6viv2 destination]$ ./fileflags 2 2>> temp.foo
write only, append
[user@desktop-5p6viv2 destination]$ ./fileflags 5 5<> temp.foo
read write
```

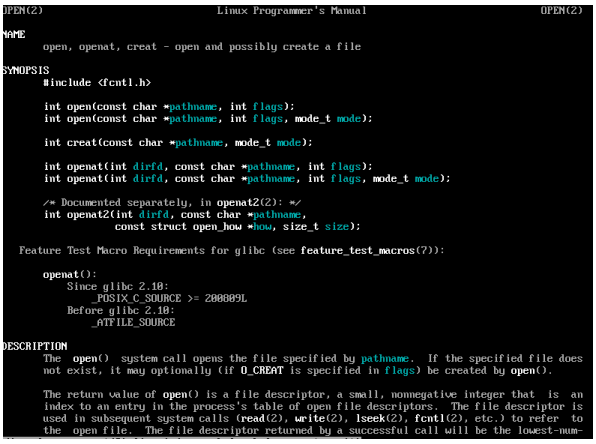
Am afișat flag-urile unui fișier asociat al unui anumit descriptor de fișier.

- ❖ Paginile de manual pentru funcțiile utilizate:



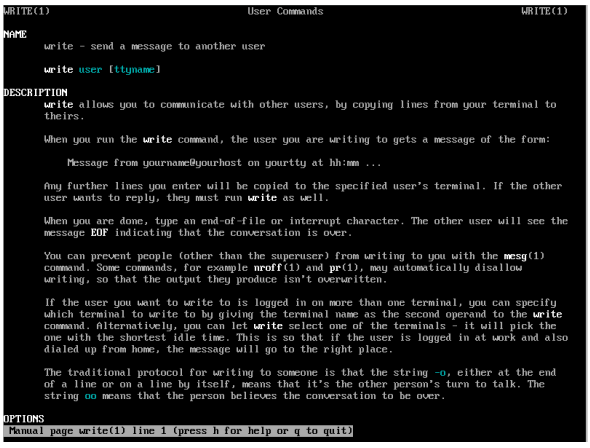
```
LSEEK(2)                        Linux Programmer's Manual                       LSEEK(2)

NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       lseek()  repositions the file offset of the open file description associated with the file
       descriptor fd to the argument offset according to the directive whence as follows:

       SEEK_SET
              The file offset is set to offset bytes.

       SEEK_CUR
              The file offset is set to its current location plus offset bytes.

       SEEK_END
              The file offset is set to the size of the file plus offset bytes.

       lseek() allows the file offset to be set beyond the end of the file  (but  this  does  not
       change the size of the file).  If data is later written at this point, subsequent reads of
       the data in the gap (a "hole") return null bytes ('\0') until  data  is  actually  written
       into the gap.

   Seeking file data and holes
       Since version 3.1, Linux supports the following additional values for whence:

       SEEK_DATA
              Adjust  the  file  offset to the next location in the file greater than or equal to
              offset containing data.  If offset points to data, then the file offset is  set  to
              offset.
```

Funcția lseek.



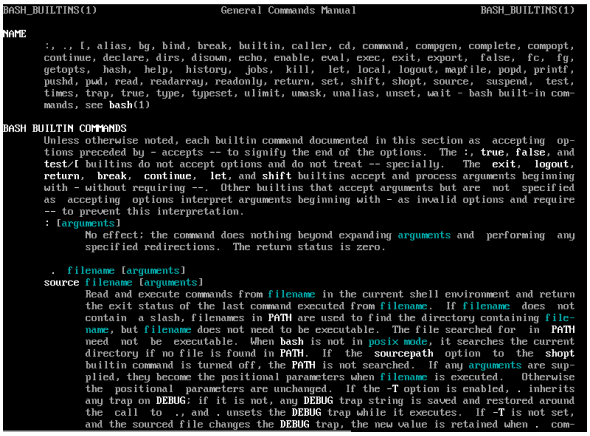
```
OPEN(2)                         Linux Programmer's Manual                       OPEN(2)

NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

       /* Documented separately, in openat2(2): */
       int openat2(int dirfd, const char *pathname,
                   const struct open_how *how, size_t size);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE

DESCRIPTION
       The  open()  system call opens the file specified by pathname.  If the specified file does
       not exist, it may optionally (if O_CREAT is specified in flags) be created by open().

       The return value of open() is a file descriptor, a small, nonnegative integer that  is  an
       index to an entry in the process's table of open file descriptors.  The file descriptor is
       used in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.) to refer  to
       the  open file.  The file descriptor returned by a successful call will be the lowest-num-
```

Funcția creat.



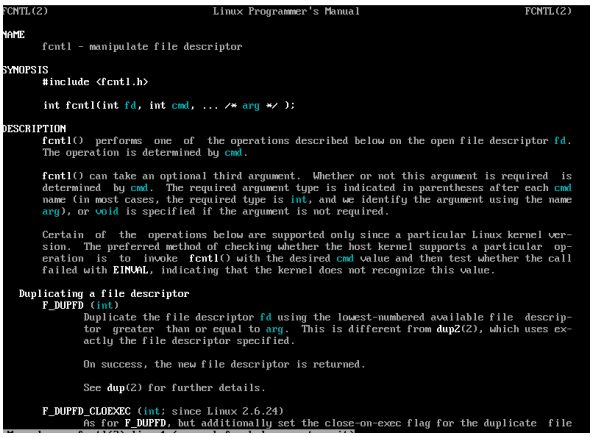
```
WRITE(1)                          User Commands                         WRITE(1)

NAME
       write - send a message to another user

       write user [ttyname]

DESCRIPTION
       write allows you to communicate with other users, by copying lines from your terminal to
       theirs.

       When you run the write command, the user you are writing to gets a message of the form:

           Message from yourname@yourhost on yourtty at hh:mm ...

       Any further lines you enter will be copied to the specified user's terminal. If the other
       user wants to reply, they must run write as well.

       When you are done, type an end-of-file or interrupt character. The other user will see the
       message EOF indicating that the conversation is over.

       You can prevent people (other than the superuser) from writing to you with the mesg(1)
       command. Some commands, for example nroff(1) and pr(1), may automatically disallow
       writing, so that the output they produce isn't overwritten.

       If the user you want to write to is logged in on more than one terminal, you can specify
       which terminal to write to by giving the terminal name as the second operand to the write
       command. Alternatively, you can let write select one of the terminals - it will pick the
       one with the shortest idle time. This is so that if the user is logged in at work and also
       dialed up from home, the message will go to the right place.

       The traditional protocol for writing to someone is that the string -o, either at the end
       of a line or on a line by itself, means that it's the other person's turn to talk. The
       string oo means that the person believes the conversation to be over.

OPTIONS
 Manual page write(1) line 1 (press h for help or q to quit)
```

Funcția write.



```
BASH_BUILTINS(1)                General Commands Manual                BASH_BUILTINS(1)

NAME
       :, ., [, alias, bg, bind, break, builtin, caller, cd, command, compgen, complete, compopt,
       continue, declare, dirs, disown, echo, enable, eval, exec, exit, export, false, fc, fg,
       getopts, hash, help, history, jobs, kill, let, local, logout, mapfile, popd, printf,
       pushd, pwd, read, readarray, readonly, return, set, shift, shopt, source, suspend, test,
       times, trap, true, type, typeset, ulimit, umask, unalias, unset, wait - bash built-in com-
       mands, see bash(1)

BASH BUILTIN COMMANDS
       Unless otherwise noted, each builtin command documented in this section as accepting op-
       tions preceded by - accepts -- to signify the end of the options. The :, true, false, and
       test/[ builtins do not accept options and do not treat -- specially. The exit, logout,
       return, break, continue, let, and shift builtins accept and process arguments beginning
       with - without requiring --. Other builtins that accept arguments but are not specified
       as accepting options interpret arguments beginning with - as invalid options and require
       -- to prevent this interpretation.
       : [arguments]
              No effect; the command does nothing beyond expanding arguments and performing any
              specified redirections. The return status is zero.

        . filename [arguments]
       source filename [arguments]
              Read and execute commands from filename in the current shell environment and return
              the exit status of the last command executed from filename. If filename does not
              contain a slash, filenames in PATH are used to find the directory containing file-
              name, but filename does not need to be executable. The file searched for in PATH
              need not be executable. When bash is not in posix mode, it searches the current
              directory if no file is found in PATH. If the sourcepath option to the shopt
              builtin command is turned off, the PATH is not searched. If any arguments are sup-
              plied, they become the positional parameters when filename is executed. Otherwise
              the positional parameters are unchanged. If the -T option is enabled, . inherits
              any trap on DEBUG; if it is not, any DEBUG trap string is saved and restored around
              the call to ., and . unsets the DEBUG trap while it executes. If -T is not set,
              and the sourced file changes the DEBUG trap, the new value is retained when . com-
```

Funcția read.



```
FCNTL(2)                        Linux Programmer's Manual                       FCNTL(2)

NAME
       fcntl - manipulate file descriptor

SYNOPSIS
       #include <fcntl.h>

       int fcntl(int fd, int cmd, ... /* arg */ );

DESCRIPTION
       fcntl() performs one of the operations described below on the open file descriptor fd.
       The operation is determined by cmd.

       fcntl() can take an optional third argument. Whether or not this argument is required is
       determined by cmd. The required argument type is indicated in parentheses after each cmd
       name (in most cases, the required type is int, and we identify the argument using the name
       arg), or void is specified if the argument is not required.

       Certain of the operations below are supported only since a particular Linux kernel ver-
       sion. The preferred method of checking whether the host kernel supports a particular op-
       eration is to invoke fcntl() with the desired cmd value and then test whether the call
       failed with EINVAL, indicating that the kernel does not recognize this value.

   Duplicating a file descriptor
       F_DUPFD (int)
              Duplicate the file descriptor fd using the lowest-numbered available file descrip-
              tor greater than or equal to arg. This is different from dup2(2), which uses ex-
              actly the file descriptor specified.

              On success, the new file descriptor is returned.

              See dup(2) for further details.

       F_DUPFD_CLOEXEC (int; since Linux 2.6.24)
              As for F_DUPFD, but additionally set the close-on-exec flag for the duplicate file
```

Funcția fcntl.

Exercițiul 2

Dacă deschidem un fișier pentru read-write cu flag-ul pentru append, se poate citi din orice poziție specificată cu `lseek`? Se poate scrie peste datele existente? Scrieți un program pentru a verifica acest lucru.

Dacă deschidem un fișier pentru read-write cu flag-ul pentru append (a), pentru orice operație de citire va începe de la începutul fișierului, indiferent de poziția specificată cu lseek.De asemenea, nu se poate scrie peste datele existente cu această metodă deoarece întotdeauna se va scrie la sfârșitul fișierului.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fnctl.h>

int main()
{
	int fisier;
	char buffer[100];
	off_t pozitie;
	fisier=open("text.txt", O_RDWR | O_APPEND);
	if(fisier<0)
	{
		perror("open");
		exit(1);
	}
	write(fisier,"abcde",5);
	pozitie=lseek(fisier,0,SEEK_SET);
	if(read(fisier,buffer,5)!=5)
	{
		perror("read");
		exit(1);
	}
	printf("A fost citit: %s\n",buffer);
	close(fisier);
	return 0;
}
```

Codul în C.

```
[user@desktop-5p6viv2 destination]$ gcc -o program exercitiu2.c
[user@desktop-5p6viv2 destination]$ ./program
A fost citit: abcde
```

Am folosit comanda *gcc -o program exercitiu2.c* pentru a creat un program care să ruleze codul și după am rulat codul cu comanda *./program*.

Exercițiul 3

Cum se comportă comanda `mycat` în cazul unui fișier cu "găuri"? Scrieți un fișier cu două astfel de "găuri" și verificați. Scrieți apoi un program `mycat2` care să elimine "găurile".

Comanda mycat în cazul unui fișier cu ”găuri” este că va afișa conținutul fișierului, inclusiv porțiunile de spațiu gol.

```
[user@desktop-5p6viv2 destination]$ touch test.txt
[user@desktop-5p6viv2 destination]$ echo "Acesta este un fisier cu doua gauri." > test.txt
```

Am creat fișierul test.txt și i-am adăugat un text.

```
[user@desktop-5p6viv2 destination]$ dd if=/dev/zero of=test.txt bs=1 count=30 conv=notrunc
30+0 records in
30+0 records out
30 bytes copied, 0.00168293 s, 17.8 kB/s
[user@desktop-5p6viv2 destination]$ printf '  ' | dd of=test.txt bs=1 seek=10 conv=notrunc
2+0 records in
2+0 records out
2 bytes copied, 0.000585918 s, 3.4 kB/s
[user@desktop-5p6viv2 destination]$ printf '  ' | dd of=test.txt bs=1 seek=20 conv=notrunc
2+0 records in
2+0 records out
2 bytes copied, 0.000607401 s, 3.3 kB/s
```

Am creat în fișierul test.txt 30 de blocuri de dimensiunea de un byte și după am creat două găuri.

```
[user@desktop-5p6viv2 destination]$ cat test.txt
    gauri.
[user@desktop-5p6viv2 destination]$ hexdump -C test.txt
00000000  00 00 00 00 00 00 00 00  00 00 20 20 00 00 00 00  |..........  ....|
00000010  00 00 00 00 20 20 00 00  00 00 00 00 00 00 67 61  |....  ........ga|
00000020  75 72 69 2e 0a                                    |uri..|
00000025
```

Am afișat fișierul după aceste modificări.

```c
#include "ourhdr.h"
#include <fcntl.h>

int main()
{
        int n;
        char buf[8192];
        int fisier1=creat("test2.txt",FILE_MODE);
        int fisier2=open("test.txt",O_RDWR);
        while((n=read(fisier2,buf,1))>0)
        {
                if(buf[0]!='\0')
                  write(fisier1,buf,n);
        }
        close(fisier2);
        remove("test.txt");
        close(fisier1);
        rename("test2.txt", "test.txt");
        return 0;
}
```

Codul în C.

```
[user@desktop-5p6viv2 destination]$ gcc exercitiu3.c -o program
[user@desktop-5p6viv2 destination]$ ./program
[user@desktop-5p6viv2 destination]$ cat test.txt
    gauri.
[user@desktop-5p6viv2 destination]$ hexdump -C test.txt
00000000  20 20 20 20 67 61 75 72  69 2e 0a                  |    gauri..|
0000000b
```

Rezultatul de după rularea codului.

Exercițiul 4

Programul `fileflags` folosește descriptorul de fișier transmis în linia de comandă. Primele două utilizări exemplificate folosesc redirectarea intrării respectiv a ieșirii. Încercați să vă dați seama ce fel de redirectare se realizează cu celelalte două redirectări exemplificate. Explicați și creați niște exemple de folosire.

Programul preia atributele descriptorului de fișier dat în linia termnalului.

./file flags 2 2>> temp.foo

Fișierului temp.foo îi este asociat descriptorul de fișier 2 (canalul de erori), iar eroarea generată de argumentul dat (5) în programul fileflags este scrisă în fișierul temp.foo la final.

./fileflags 2 1<> temp.foo

Fișierului temp.foo îi este asociat descriptorul de fișier 1 (standard output), iar rezultatul afișat în terminal este redirecționat în fișier. Rezultatul reprezintă atributele descriptorului de fișier 2 (canalul de erori), adică „read write”.