LabX-3, BY 17 Nov 2021

How to do the exercises (partially repeat, *for changes see italics*)

We expect that after doing the exercises your skills in handling Linux OS and programming in C will increase. What it takes, is memorizing commands by typing the commands on the Linux terminal and observing the result. This is what the first part is about. This part is for the self-learning.

The second part is about writing a C program. At the end of the course you should be able to translate your research task into a C program based on libraries of different functions found in the Linux environment.

In this assignment we look closer into the I/O functions in the file system. To accomplish this task you will use a function from standard library. Part of the exercise is to study the manual pages related to the suggested functions.

You write the program yourself based on the guidance we give in the assignment. You can use the C book of Kernighan & Ritchie (see Pages/Textbooks on canvas[1]). In canvas Files you have a summary of C language in slides file 3.1_Linux-C[2]

You compile and run the program as in the instructions in this assignments and solving error messages from the compiler. After you convince yourself that the program is doing what it supposed to do you copy it to directory for us to check (see below).

You pass successfully if the program is running and produces the desired result. We will give you feedback on your programming style, as this is important going forward. In program text, we ask you to comment the statements in the programs. Also, the header in form of a comment should contain your name and date of creation and a summary of the intended program behavior.

On the delivery of your results: When you are ready with your program assignment, please create a directory with your name and copy the files there:

```
mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/   #it may exist already from exe1
mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx3
cp yourprogram /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx3/yourprogram
```

After you have finished copying and ready to report, please send us e-mail.If there are any problems or questions, please email me or Andrey:

"Igor Zacharov" I.Zacharov@skoltech.ru

"Andrey Kardashin" Andrey.Kardashin@skoltech.ru

[1] http://cslabcms.nju.edu.cn/problem_solving/images/c/cc/The_C_Programming_Language_(2nd_Edition_Ritchie_Kernighan).pdf

[2] https://skoltech.instructure.com/files/257760/download?download_frd=1

Further commands

```
cp
scp
mv
ln
grep
wc
tr
find
```

Login to your sandbox account: ssh –p 2200 yourname@sandbox.zhores.net

Get used to frequent commands in the file system

a) Copy files: `cp hello.c hello2.c`

Also, there is secure remote copy:

```
scp -P 2200 localfile user@sandbox.zhores.net:
```

Will copy the localfile to the sandbox. User is your username on sandbox. Note the `:` (colon) at the end. Without it you will just copy to local directory with that strange name. It may ask you for your password on the remote machine, but if you have setup a transparent access it will not.

b) Move files: `mv hello2.c newprog.c`

The difference between cp and mv, that with mv the original name disappears. It is a way to rename files. Be careful, mv (move) looks similar to rm (remove), never confuse these two.

c) Link to a file. There are two types, the symbolic link (with –s) and hard link (by default)

Try
```
touch file_on_disk;
   ln -s file_on_disk   link_to_file
   ln    file_on_disk   hink_to_file
```

```
[i.zacharov@an LS_1]$ ls -li *file*
1099662281152 -rw-rw-r-- 2 i.zacharov i.zacharov  4 Nov  8 16:55 file_on_disk
1099662281152 -rw-rw-r-- 2 i.zacharov i.zacharov  4 Nov  8 16:55 hink_to_file
1099662281063 lrwxrwxrwx 1 i.zacharov i.zacharov 12 Nov  8 16:28 link_to_file -> file_on_disk
```

Inode; Link; number of hard links (also, number 2 for directory)

The `-i` switch on the ls command will display the i-node number. Observe: hard link is same i-node.

If removing the file a soft link is pointing to, the data disappears. If you make the file again, the link will point to it again (test it). Removing file with a hard link – the data is still there until all hard links are removed.

Try `rm file_on_disk; ls –li *file*`

Number of hard links on the hink_to_file is reduced. The soft link is left dangling.

Try this with file containing some data, remove the file cat the hard link and see contents preserved.

d) The `grep` command. Try `grep stdio *.c` in the directory with your C programs. This utility searches for pattern in files and prints the lines where the pattern is present. The pattern may be a wild card. Put it in ‘ ‘ to protect from bash expansion.

Try `grep '[X-Z]' *.c` in the directory where you have your C programs. This pattern says: all lines containing capital letters in the range from X to Z (thus, it should discover W from the World).

Watch the video Regex (8m): https://www.youtube.com/watch?v=jCAyQ7C71m4 I suggested as HW. I also find the explanation in https://www.youtube.com/watch?v=FqrYjWN0TZ0 good for * and ? special characters.

Get the file http://textfiles.com/science/blackhol.txt and get all lines with capital letters:

Try `grep '[A-Z]' blackhol.txt` In the output the the lines with pattern matching the requested pattern are displayed. Notice how text is highlighted for the found pattern.

The –v reverses the pattern. Try `grep –v '[A-Z]' blackhol.txt` Now lines with only small letters are printed. Discover if there are any lines with numbers: `grep '[0-9]' blackhol.txt`

Note the status $? of the command execution: 0 (true) if found, 1 (false) if not found. Test this as it is useful in bash scripts in the future.

e) the wc command can count lines, words and characters. Try `wc blackhol.txt`

There are 324 lines, 2476 words and 15872 characters.

```
[i.zacharov@an LS_1]$ wc blackhol.txt
 324  2476 15872 blackhol.txt
```

Try `grep '[0-9]' blackhol.txt | wc –l` will count only lines (21 lines – test it)

What is the number of words in lines with numbers in them.

f) How to count only the numbers themselves: use the `tr` command to remove characters:

```
grep '[0-9]'blackhol.txt |tr -s '[:alpha:]' ' ' |tr –d '[:punct:]' |wc –w
```

To see the effect of each command go step by step in extending the pipe and observe each output. At the end it should print 21. See `man tr` for the pattern classes :xxx: and other flags (-s, -d).

c) The find command has a different syntax than other commands, eg.

```
find /usr/include –name stdio.h
```

First argument is the upper level of the directory hierarchy from which it should start searching. The argument after `–name` is the name it searches for.

Try `find /usr/include –name '*.h'`
The wild cards should be in '' to prevent expansion by bash.

Try `find /usr/include -name *.h –print –exec ls –l {} \;`

The –exec will apply the command to all files found. Note the strange syntax, (see man find)
The –print flag makes sure the output is presented properly for the ls command

Editor vi (vim) – A reminder if you did not get the habit yet (this is same as EX-1)

Create an empty file with name xx (or your choice) or just type in: vi xx

```
vi fname
```

Editor vi commands to enter the insert mode: i

Exit insert mode: key **Esc** on the keyboard

To write out the changed file: type in : (colon), then type in w (letter w)

To exit the editor: ZZ (Capitals) or via the command line: type :q

Watch (7m) https://www.youtube.com/watch?v=pU2k776i2Zw to learn vi.

Writing C programs

1) Taking the previous program and make a standard structure:

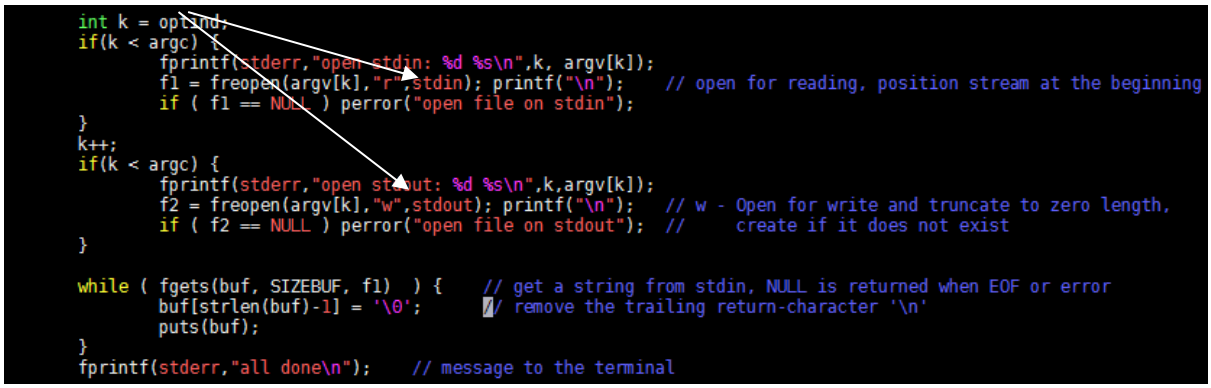
```c
/*
 * demonstrate reopen of std I/O streams
 * Exercise 2
 * Program will copy file in first argument to second argument, or std streams
 * prog -s <long> in out
 * prog -s <long> <in >out
 * prog -s <long> in         should work to dump file in to terminal
 * prog -s <long> in 2>log   should work to dump file in to terminal and error stream to log
 * some messages are printed to stderr and can be redirected to stdout with &>
 * (note the arbitrary order of lines from stderr in redirected output)
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

/* program definitions
 * put here all explicit numbers and definitions
 */
#define SIZEBUF 128     // this is the size of the input buffer
#define OPTFLAGS "s:"   // option flags as compiler symbolic string

/* global variables
 */
extern int optind;
const char optflags[] = OPTFLAGS ;   //option flags

void usage(int argc, char *argv[])
{
        fprintf(stderr,"usage: %s [-%c <int>] in out\n",argv[0],optflags[0]);
        exit(0);
}
long getparams(int argc, char *argv[])
{
        const char *flags = optflags;
        char *param = NULL;
        int opt;
        int flag = 0;
        extern char * optarg;
        long parsize = 0;

        while ( (opt = getopt(argc, argv, flags)) != -1 ) {
                if ( opt == optflags[0] ) {     // Note: switch needs const in case
                                param = optarg;
                                parsize = atoll(param);
                                //break;   should NOT break the while loop to collect all flags/parameters
                }
                else usage(argc,argv);
        }
        return parsize;
}

int main(int argc, char *argv[])
{

        FILE *f1 = stdin;
        FILE *f2 = stdout;
        char buf[SIZEBUF];      // use preprocessor defined constants
        long parsize = getparams(argc, argv);
        int i;

        fprintf(stderr,"option arguments: %ld\n",parsize);

        int k = optind;
        if(k < argc) {
                fprintf(stderr,"open stdin: %d %s\n",k, argv[k]);
                f1 = freopen(argv[k],"r",stdin); printf("\n");    // open for reading, position stream at the beginning
                if ( f1 == NULL ) perror("open file on stdin");
        }
        k++;
```

Note the “standard” structure of C programs with the following sections:

a) header with comment describing the program and creation
b) includes
c) defines
d) typedefs
e) global variables
f) functions
Before the main() function.

This is just a restructuring your previous program and this structure will be used to go forward with the next assignment, which is also visible in the screenshot above.

2) Get the arguments of the program (after the flags) and open them by re-using the standard streams (**stdin**, **stdout**) with the freopen library calls (see man freopen):



```
int k = optind;
if(k < argc) {
        fprintf(stderr,"open stdin: %d %s\n",k, argv[k]);
        f1 = freopen(argv[k],"r",stdin); printf("\n");    // open for reading, position stream at the beginning
        if ( f1 == NULL ) perror("open file on stdin");
}
k++;
if(k < argc) {
        fprintf(stderr,"open stdout: %d %s\n",k,argv[k]);
        f2 = freopen(argv[k],"w",stdout); printf("\n");   // w - Open for write and truncate to zero length,
        if ( f2 == NULL ) perror("open file on stdout");  //     create if it does not exist
}

while ( fgets(buf, SIZEBUF, f1)  ) {    // get a string from stdin, NULL is returned when EOF or error
        buf[strlen(buf)-1] = '\0';      // remove the trailing return-character '\n'
        puts(buf);
}
fprintf(stderr,"all done\n");    // message to the terminal
```

The program will copy standard input to standard output or use the files given as arguments. See that in order to suppress superficial blank lines you have to remove the \n from the input line.

Read the description of the fgets() function (`man fgets`). This is a safe function to use. To contrast, note the description of the gets and read the text market BUGS. Usage of this function may lead to successful hacker attacks by buffer overran.

Read the description of the puts() function (`man puts`). This function is writing output to stdout. What have we done to be able to use the stdout in this way? Comment the answer in the program.

Verify that this program uses similar semantics to many other standard Linux utilities with standard in and out way of handling streams of data. Does it also work as a filter (pipe with | symbol)?

What can you do to send the stderr output (i.e. the “all done” message) to a log file?

3) Get the text file http://textfiles.com/science/blackhol.txt

Based on the program above (2), write a program to read this file and write lines containing only capitals (e.g. `grep '[A-Z]' blackhol.txt | grep -v '[a-z]' >onlycapitals`). Compare your output with what you get using grep. They should be same.

Hint. You may want to use the isalnum and islower function calls (see `man isspace`) to discover the composition of the input line (in buf, see program text above). You can loop over each character with

```
for(i=0;  i < strlen(buf);  i++) {
      if ( isalnum( buf[i] )  ) flag_chars_present = 1;
      if ( islower( buf[i] )  ) flag_lower_present = 1;
}
```

4) This is a more advanced exercise using system calls for read and write.

Changing from character based stream to binary input/output. Demonstrating possible holes in the file when seeking (positioning) the write after EOF. The FILE* declarations are replaced by file descriptors needed for open:

```
int f1 = fileno(stdin); // file descriptor for the standard streams
int f2 = fileno(stdout);
int readflg = O_RDONLY;
int writflg = O_CREAT | O_WRONLY | O_TRUNC;
off_t fpos;
ssize_t nread, nwrit;
char buf[SIZEBUF];      // use preprocessor defined constants
long parsize = getparams(argc, argv);
```

The stream functions are replaced by read and write system calls. There is a special semantics to be observed:

```
while ( (nread = read(f1, buf, SIZEBUF))  ) {          // read bytes, NULL is returned when EOF or error
        nwrit = 0;
        do {
            nread -= nwrit;
            nwrit = write(f2,buf+nwrit,nread);    // if write is interrupted, it may return less bytes.
        } while ( nwrit < nread );                // in that case the call is restarted until all bytes done
}
```

```
int k = optind;
if(k < argc) {
        fprintf(stderr,"open stdin: %d %s\n",k, argv[k]);
        f1 = open(argv[k],readflg);                    //open for reading, position stream at the beginning
        if ( f1 < 0 ) perror("open input file");
}
k++;
if(k < argc) {
        fprintf(stderr,"open stdout: %d %s\n",k,argv[k]);
        f2 = open(argv[k],writflg);                    // Open for write and truncate
        if ( f2 < 0 ) perror("open output file");

}
if ( (fpos = lseek(f2, (off_t)0, SEEK_END)) < 0 ) perror("lseek" );
else fprintf(stderr,"write position in file %s is: %ld Bytes\n",argv[k], fpos);

if ( (fpos = lseek(f2, parsize, SEEK_END)) < 0 ) perror("lseek" );
else fprintf(stderr,"moving byound EOF by %ld Bytes, current position %ld Bytes\n",parsize, fpos);


while ( (nread = read(f1, buf, SIZEBUF))  ) {    // get a string from stdin, NULL is returned when EOF or error
         nwrit = 0;
         do {
                nread -= nwrit;
                nwrit = write(f2,buf+nwrit,nread);      // if write is interrupted, it may return less bytes.
         } while ( nwrit < nread );                     // in that case the call is restarted until all bytes are written
}

if ( fstat(f2, & finfo) < 0 ) perror("fstat"); // fileno extracts file descriptior from FILE *
else {
        fprintf(stderr,"file size: %ld\n",finfo.st_size);
        fprintf(stderr,"blocks allocated: %ld size on disk: %ld Bytes\n",
                        finfo.st_blocks, finfo.st_blocks * 512);
}
```

The program can be run with `./progio -s 1000000000000 small /tmp/junk1`

The `ls -l` is showing a very big file, while du (disk usage check, `man du`) shows only 4 KB allocated:

```
[i.zacharov@an LS_1]$ ls -l /tmp/junk1
---x-----t 1 i.zacharov i.zacharov 1000000000093 Nov  8 20:57 /tmp/junk1
[i.zacharov@an LS_1]$ du -ks /tmp/junk1
4       /tmp/junk1
```

Note the usage of fstat to request details of the file from the OS (see man fstat).