# LabX-5, BY 24 Nov 2021

## How to do the exercises (partially repeat of text from EX-1)

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 consider processes.* 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] . Also use the summary slides in file 3.1_Linux_Bash[3]

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 should exist already from exe1

mkdir /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx5

cp yourprogram /trinity/home/LINUX_SUPERCOMPUTERS/yourname/labx5/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

[3] https://skoltech.instructure.com/files/262799/download?download_frd=1

# Further commands and Bash structure

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

## Get used to the bash script

a) Open the editor and create file mybash.sh with the following structure:

```
#!/usr/bin/bash


var="Hello, World!"       # no spaces around the = sign
prog=$0             # name of the script
echo prog: $prog

arg1=$1             # argument in 1st position
echo "arg1: $arg1"

argd=${1:-10}       # positional defults to 10 if not there
echo "arg2: $argd" # substitution
echo 'arg2: $argd' # does not substitute


newvar1=$var.new    # create new names with existing value

newvar2=${var}XX    # create new names using {}

newvar3=$(ls -l)    # run cmd in a sub shell and return result
echo $newvar3

echo
echo
newvar4=`date `     # run cmd in same shell and return result
echo $newvar4
```

Observe the different ways to initialize the variables within the bash shell.

b) In you command bash type in

myvar=”Hello, World!”

create a bash script to echo the variable: echo $myvar

Did not work? Try to initialize it with

export myvar=”Hello, World!”

and repeat the exercise. (By the way, just export myvar after the initialization should work. Try).

## 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)** Demonstrating shell behavior. The program accepts commands with its flags/parameters as arguments and executes them in an endless loop.

Here is the header and variables:

```
/*
 * demonstrate shell behaviour of running program read from stdin
 * Exercise 5
 *
 * myshell cmd args
 *
 * I.Zacharov   initial setup   Skoltech November 2021 rev.    0.1
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

#define BUFSIZE         128    // this is the size of the input buffer
#define CMDNAME          32    // max size of a command
#define NUMARGS           8    // max number of arguments
#define FULLCMD         255

int main(int argc, char *argv[])
{
        char buf[BUFSIZE];
        char cmd[BUFSIZE];
        char *cmdargs[NUMARGS] = {NULL};
        int params, totparams;
        int history = 0;
        int nparams = 0;
        int n;
        pid_t pid;
        int status;
        int cpath;
        char fullcmd[FULLCMD];
        char *PATH = getenv("PATH");
        char s1, s2;
```

The processing is an infinite loop while(1) that reads the command from the stdin fgets(), analyzes the content of the line sscanf() to get the name of the command and its arguments. The template is always:

cmd [arg1 [arg2 [….]]]

It actually tries to find the location of the cmd by looking at the PATH environment variable and checking the existence with stat().

Main part is fork() followed by execve(). Here is the processing:

```
        while ( 1 ) {
                printf("%-d> ",history++);
                if( fgets(buf,BUFSIZE,stdin) == NULL ) break;
                for(nparams = 0; nparams < NUMARGS; nparams++) {
                        free(cmdargs[nparams]);
                        cmdargs[nparams] = 0;
                }
                totparams = params = nparams = 0;
                do {
                        n = sscanf(buf+totparams,"%ms %n",&cmdargs[nparams],&params);
                        totparams += params;
                        nparams++;
                } while ( n > 0 );

                /* search for the command with standard PATH */
                cpath = 0;
                do {
                        struct stat statbuf;
                        int cmdlen = strlen(cmdargs[0]);
                        int cp2 = cpath;

                        s1 = PATH[cpath];
                        s2 = s1;
                        while ( s2 && (  PATH[cpath] != ':') ) s2 = PATH[cpath++];
                        strncpy(fullcmd,&PATH[cp2],cpath-cp2);
                        fullcmd[cpath-cp2] = '/';
                        cpath++;
                        strcpy(&fullcmd[cpath-cp2],cmdargs[0]);
                        if ( stat(fullcmd, & statbuf) == 0 ) break;
                } while( s2 );

                pid = fork();
                if ( pid < 0  ) { perror("fork"); continue; }
                if ( pid == 0 ) {       // child
#ifdef DEBUG
                        for(int i = 0; i < nparams; i++)
                                fprintf(stderr,"child cmd %s , %d parameter: %s\n",cmdargs[0], i, cmdargs[i]);
#endif
                        if ( execve( fullcmd, cmdargs, NULL) < 0) {
                                perror("execve");
                                exit(EXIT_FAILURE);
                        }
                }

                waitpid(-1, &status, 0);
                fprintf(stderr,"status execution of %s %d\n",cmdargs[0], status);
        }
        fprintf(stderr,"all done\n");    // message to the terminal
}
```

Implement the myshell command and try simple commands with it, like: ls –l

2) Demonstrate access to files from forked processes (parent – child). The preparation part until the fork() here:

```c
/*
 * demonstrate acces to files from forked processes
 * Exercise 5
 *
 * myfile filename
 *
 * I.Zacharov   initial setup   Skoltech November 2021 rev.    0.1
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>

#define BUFSIZE          128    // this is the size of the output buffer
#define NLINES            50    // number of lines to write to output

int main(int argc, char *argv[])
{
        char buf[BUFSIZE];
        char *filename;
        FILE * f1;
        int n = NLINES;
        int numchars;
        char *process;
        pid_t pid, ppid;
        useconds_t delay_microsec;
        int linepos;
        int i;

        if( argc > 1 ) filename = argv[1];
        else exit(0);

        if ( ( f1 = fopen(filename,"w") ) < 0 ) { perror(filename); exit(1); }
        fprintf(stderr,"opening file %s\n", filename);
        //setlinebuf(f1);   // alternatively: flush the buffer
        //if( setvbuf(f1, NULL, _IONBF, 0) < 0) perror("setvbuf"); //unbuffered

        pid = fork();
        if ( pid < 0  ) { perror("fork"); exit(2); }
```

The invocation is like

prog <filename>

The filename is open with “w”, thus it will be positioning stream at the beginning for writing.

After fork() the FILE *f1 will be available to both the parent and the child.

Note the commented out setting for the stream buffer size setlinebuf() will limit it to one line.

a) Note that we use same var names for parent and child. The values they are initialized with before the fork are propagated to the child, but the new values we assign are not shared. The address space if fully separated, pages are marked COPY-ON-WRITE in case variables change value.

b) in the for(..) loop we set the location of the write and do fputs() from each process (parent and child). Between the fseek() and fputs() there is a variable delay usleep() [to simulate processing].

c) to make sure the fputs() does not stay in an internal buffer we do fflush().

```c
        if ( pid > 0 ) { process = "parent"; delay_microsec = 500000; linepos = NLINES; }
        else           { process = "child";  delay_microsec = 650000; linepos = 1; }

        pid = getpid(); ppid = getppid();

        for ( i=0; i < NLINES; i++ ) {

                fprintf(stderr,"now printing %s\n",process);    // message to the terminal
                numchars = snprintf(buf,BUFSIZE,"pid=%6d ppid=%6d %6s  Here is line %3d\n",
                                        pid,ppid,process,i); // note COPY-ON-WRITE pages

                //if ( linepos )
                fseek(f1,(long )(numchars*(i+1)*linepos), SEEK_SET); //MT_SAFE

                usleep(delay_microsec);

                if ( fputs(buf,f1) < 0 ) { perror("fputs"); exit(3); }
                fflush(f1);     // flush the buffer

        }
        fprintf(stderr,"all done %s\n",process);    // message to the terminal
}
```

After implementing this with gcc and running, cat the file you wrote to. Try to understand if it is possible to safely write to same file from the two processes “at the same time”.