Indian Institute of Technology Jodhpur
Operating Systems Lab (CSL 3030)
Assignment 2

Dated 22nd August, 2022 Total marks: 30

This assignment is on Linux System Calls, File I/O, fork, signals, pipe, message queue etc.

This assignment has two parts. Part-I is to be completed in the lab session itself. Part - II is the takeaway component and the evaluation policy is stated at the end.

Part I

Write a C program that would do the following:

(1) reads from the first file argument, reverses lowercase and uppercase letters, that is, makes lowercase all uppercase letters and vice versa, and writes out all characters, whether or not reversed; and

(2) reads the above output, counts the number of NON-alphabetic characters (that is, those not in the a-z range nor in the A-Z range), and writes out all characters, whether counted or not, into its second file argument.

The program would be invoked from the command line as *"driver file1 file2"*
Both file arguments are required (print an error message on stderr using fprintf if either argument is missing and then exit(1)). The driver program will open the first file and create the second file, and dup them down to stdin and stdout. The driver program sets up a pipe and then forks two children.

The first child forked will dup the read end of the pipe down to stdin and then execl the program you have written, count, which will read characters from stdin, count the non-alphabetic ones (see /usr/include/ctype.h), and write all characters out to stdout. The count program uses only stdin and stdout, and only the stdio library routines (see stdio.h) for input and output (see getchar and putchar). The program will print out the final count on stderr using fprintf and then exit(0) when it hits EOF in its input.

The second child forked will dup the write end of the pipe down to stdout and then execl the program you have written, convert, which will read characters from stdin, reverse uppercase and lowercase (again, see ctype.h), and write them all out to stdout. Like count, this program uses only stdin and stdout. Also, it uses only the stdio library routines. The program will exit(0) when it hits EOF in its input.

The reason for creating the children in this order (first the one that reads from the pipe, usually called the second in the pipeline, and then second the one that writes to the pipe, usually called the first in the pipeline -- confusing!!) is that LINUX will not let a process write to a pipe if no

process has the pipe open for reading; the process trying to write would get the SIGPIPE signal. So we create first the process that reads from the pipe to avoid that possibility.

Meanwhile, the parent process, the driver, will close its pipe file descriptors and then call wait (see /usr/include/sys/wait.h) twice to reap the zombie children processes when they finish. Then the parent will exit(0).

Remember to close all unneeded file descriptors, including unneeded pipe ones, or EOF on a pipe read may not be detected correctly. Check all system calls for the error return (-1). For all error and debug messages, use ``fprintf(stderr, ...)'' or use the function perror or the external variable errno (see also /usr/include/errno.h). Declare a function to be of type void if it does not return a value, that is, if it is a subroutine rather than a true function. If you absolutely must ignore the output of a system call or function, then cast the call to void.

```
*****************************************************************************
                      Hints and  Notes

Program skeleton

  o parent opens file1:

        #include <sys/types.h>
        #include <fcntl.h>    /* defines O_RDONLY etc. */


        ...
        fd_in = open(argv[1], O_RDONLY);    /* not fopen */
            /* Check fd_in for -1!!!  If it is, then the file does not
               exist or you do not have read permission. */

  o parent creat's file2:

        fd_out = creat(argv[2], 0644);  /* mode = permissions, here rw-r--r--
*/
            /* Check fd_out for -1!!!  If it is, then the file could not
               be created */

  o parent uses close and dup so stdin is now file1 (done like children
below)

  o parent uses close and dup so stdout is now file2: if dup returns
    -1, then a problem has occurred

  o parent calls pipe system call to create pipe file descriptors

  o parent forks a child to read from pipe:

    -- this first child manipulates file descriptors
      use close, dup so stdin is read end of pipe
        (see text Figure 1-13, similar to else clause)
```

```
    -- this first child execs count
        execl("count", "count", (char *) 0)
        execl overlays the child with the binary compiled program in
        the file given by first argument and the process name becomes
        the second argument
```

- parent forks again a child to write to pipe:

```
    -- this second child manipulates file descriptors
        use close, dup so stdout is write end of pipe
        (similar to if clause in text Figure 1-13)

    -- this second child execs convert
        execl("convert", "convert", (char *) 0)
```

- parent closes both ends of pipe

- parent waits twice (see text Figure 1-10)

```
    -- use wait(&status); instead of waitpid(-1, &status, 0);
```

Before compiling and running driver, remember to:

- in count.c: fprintf(stderr, "final count = %d\n", count);

- cc -o convert convert.c, and

- cc -o count count.c

The two forks form nested if statements.

```
    pipe(...
    if (fork() != 0) {    /* parent continues here */

        if (fork() != 0) {    /* parent continues here */
            close(write end of pipe...
            wait(...
            wait(...
        } else {
            ...            /* second child, writes to pipe */
        }

    } else {
            ...            /* first child, reads from pipe */
    }
```

It is important to check all system calls (open, creat, dup, etc.) for a return value <0, particularly -1, because such a return value means an error has occurred. If you just let your program continues to execute, it will just dump core at some later time and you will not know why.

It is IMPERATIVE that the parent and the first child forked (the one to read the pipe) close their write end file descriptors to the pipe. If they do not

do this, then the first child will never get EOF reading from the pipe, even after the second child has sent everything. This is because EOF on a pipe is not indicated until ALL write file descriptors to the pipe are closed. If the parent and first child fail to close their file descriptors for the write end of the pipe, the program will hang forever (until killed). Furthermore, the parent must close its copy of the file descriptor to the write end of the pipe BEFORE the waits, not after, or deadlock will result. And the first child forked (the one to read the pipe) must close its file descriptor to the write end of the pipe before it starts reading the pipe, not after, or deadlock will result.

****

Hints and Notes on Linux Programming

Command-line arguments

A C main program is actually called with two arguments: argc and argv. argc is the number of command-line arguments the program is invoked with including the program name, so argc is always at least one. argv is a pointer to an array of character strings that contain the arguments, one per string.

Here is a program that echoes its arguments to show how to use these:

```
main (int argc, char *argv[])     /* echo arguments */
   /* argv is not a string but an array of pointers */
   /* argv[0] is command-line program name */
{
   int i;
   for (i=1; i<argc; i++)
      printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

Or more cryptically:

```
main (int argc, char *argv[])     /* echo arguments */
{
   while (--argc>0)
      printf((argc>1) ? "%s " : "%s\n", *++argv);
              /* argv[i] and *(argv+i) are same */
}
```

Also argv[i][j] or (*(argv+i))[j] is the jth character of the ith command-line argument.

Standard input and output

Remember IO is not a part of the C language, but is provided in the ``standard IO library'' accessed by putting

```
#include <stdio.h>
```

at the beginning of your program or in your include files. This will provide you with the functions getchar, putchar, printf, scanf and symbolic constants like EOF and NULL.

File access

It is also possible to access files other than stdin and stdout directly. First, make the declaration

```
FILE *fopen(), *fp;
```

declaring the open routine and the file pointer. Then, call the open routine

```
fp = fopen(name, mode);
```

where name is a character string and mode is "r" for read, "w" for write, or "a" for append.

Now the following routines can be used for IO:

```
c = getc(fp);
put(c, fp);
```

and also fprintf, fscanf which have a file argument.

The last thing to do is close the file (done automatically anyway at program termination) (fflush just to flush but keep the file open):

```
fclose(fp);
```

The three file pointers stdin, stdout, stderr are predefined and can be used anywhere fp above was. stderr is conventionally used for error messages like wrong arguments to a command:

```
if ((fp = fopen(*++argv, "r")) == NULL) {
   fprintf(stderr, "cat: can't open %s\n", *argv);
   exit(1);   /* close files and abort */
} else ...
```

Storage allocation

To get some dynamically allocated storage, use

```
calloc(n, sizeof(object))
```

which returns space for n objects of the given size and should be cast appropriately:

```
char *calloc();
int *ip;
   ...
ip = (int *) calloc(n, sizeof(int));
```

Use free(p) to return the space.

Linux system call interface

These routines allow access to the Linux system at a lower, perhaps more efficient, level than the standard library, which is usually implemented in terms of what follows.

File descriptors

Small positive integers are used to identify open files. Three file descriptors are automatically set up when a program is executed:

0 - stdin
1 - stdout
2 - stderr

which are usually connected with the terminal unless redirected by the shell or changed in the program with close and dup.

Low level I/O

Opening files:

```
int fd;
fd = open(name, rwmode);
```

where name is the character string file name and rwmode is 0, 1, 2 for read, write, read and write respectively. Better to use ``#include <fcntl.h>'' and defined constants like O_RDONLY. A negative result is returned in fd if there is an error such as trying to open a nonexistent file.

Creating files:

```
fd = creat(name, pmode);
```

which creates a new file or truncates an existing file and where the octal constant pmode specifies the chmod-like 9-bit protection mode for a new file. Again a negative returned value represents an error.

Closing files:

```
close(fd);
```

Removing files:

```
unlink(filename);
```

Reading and writing files:

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

will try to read or write n bytes from or to an opened or created file and return the number of bytes read or written in n_read, for example

```
#define BUFSIZ 1024  /* already defined by stdio.h if included earlier */
main ()   /* copy input to output */
{
   char buf[BUFSIZ];
   int n;
   while ((n = read(0, buf, BUFSIZ)) > 0) write(1, buf, n);
}
```

If read returns a count of 0 bytes read, then EOF has been reached.

Changing File Descriptors

If you have a file descriptor fd that you want to make stdin refer to, then the following two steps will do it:

```
close(0);    /* close stdin and free slot 0 in open file table */
dup(fd);    /* dup makes a copy of fd in the smallest unused
               slot in the open file table, which is now 0 */
```

Standard library

Do a ``man stdio'' to find out the standard IO library.

Do a ``man printf'', ``man getchar'', and ``man putchar'' for even more information.

Do a ``man atoi'' to find out ascii to integer conversion.

Do a ``man errno'' to find out printing error messages and the external variable errno.  Also do a ``man strerror''.

Do a ``man string'' to find out the string manipulation routines available.

```
strcat     /* concatenation */
strncat
strcmp     /* compare */
strncmp
strcpy     /* copy */
strncpy
strlen     /* length */
index      /* find c in s */
rindex
```

Do a ``man 2 intro'' to find out about system calls like open, read, dup, pipe, etc.

You can do a ``man 2 open'', ``man 2 dup'', etc. to find out more about each one (``man 3 execl'', ``man 2 execve'', ``man 2 write'', ``man 2 fork'', ``man 2 close'', ``man 3 exit'', ``man 2 wait'',

``man 2 getpid'', ``man 2 alarm'', ``man 2 kill'', ``man 2 signal'').

/usr/include and /usr/include/sys contain .h files that can be included in C programs with the ``#include <stdio.h>'' or ``#include <sys/inode.h>'' statements, for example.

Other useful ones are ctype.h, errno.h, math.h, signal.h, string.h.

Error messages

This program shows how to use perror to print error messages when system calls return an error status.

```
#include <stdio.h>
#include <sys/types.h>    /* fcntl.h needs this */
#include <fcntl.h>        /* for second argument of open */
#include <errno.h>

void exit(int);           /* gets rid of warning message */

main (int argc, char *argv[]) {
        if (open(argv[1], O_RDONLY) < 0) {
                fprintf(stderr, "errno=%d\n", errno);
                perror("open error in main");
        }
        exit(0);
}
```

How dup works

dup(fd) looks for the smallest unused slot in the process descriptor table and makes that slot point to the same file, pipe, whatever, that fd points to. For example, each process starts as

```
                        process descriptor table
                        ------------------------
                            slot #     points to
                            ------     ---------
                         0 (stdin)      keyboard
                        1 (stdout)        screen
                        2 (stderr)        screen
```

If the process does a

```
        fd_in = open("file1", ...
        fd_out = creat("file2", ...
```

then the table now looks like

```
                        process descriptor table
                        ------------------------
                        slot #         points to
```

```
                        ------          ---------
                             0           keyboard
                             1             screen
                             2             screen
                             3              file1
                             4              file2
```

and fd_in is 3 and fd_out is 4. Now suppose the program does a

```
        close(0);
        dup(fd_in);
        close(fd_in);
```

After the close(0), the table will look like

```
                        process descriptor table
                        ------------------------
                        slot #         points to
                        ------         ---------
                             0      -- unused --
                             1            screen
                             2            screen
                             3             file1
                             4             file2
```

and after the dup(fd_in), the table will look like

```
                        process descriptor table
                        ------------------------
                        slot #         points to
                        ------         ---------
                             0             file1
                             1            screen
                             2            screen
                             3             file1
                             4             file2
```

and after the close(fd_in), the table will look like

```
                        process descriptor table
                        ------------------------
                        slot #         points to
                        ------         ---------
                             0             file1
                             1            screen
                             2            screen
                             3      -- unused --
                             4             file2
```

Part-II

Implement the following client-server-based game using message queue. In this game, we have a set of k clients and one single server. The server is responsible to create and maintain the message queue. At the beginning of each round, the server accepts two integers MIN and MAX from the user, which it sends to all the clients using the message queue. Each client j randomly generates a guessed token (an integer) within the range {MIN:MAX} and sends that to the server, denoted by Rj. Now, the server also generates a token, say **G** (which is also an integer) in the range {MIN:MAX}, independently (which is not shared with the clients). On receiving Rj, from all the clients, the server computes $\delta = \frac{|Rj-G|}{\sum_{j=1}^{k}(Rj-G)^2}\ \forall j \in k.$

Among all the clients, the client scoring the smallest δ wins the round and gets a score of 5. The first client crossing the total score 50 over multiple rounds is declared as the champion, which in turn ends the game. The final result is announced by the server to all the clients before closing the message queue. The server must identify which client is sending what token as a reply to the user-suggested range at every round. After the game ends, the server must send a user-defined signal to all the clients declaring the end which the client should handle before exiting. The server should be the last one to exit the system. Also, at each round, the server should print the user-defined range, the value of the received token from the clients, the current score for each round, and the winner for the round in order to let the user know how the game is proceeding. You may assume that the maximum size of any message is 256 bytes. Implement server and client programs for this problem. You can use coin tossing

Submission Instructions:

- Submit your codes in a folder as Roll.zip (Includes driver.c, server.c and client.c files implementing part II)
- Make sure you include one text file (Readme.txt) stating the execution status, special, conditions, assumptions or list of bugs, if any for both the problems separately. [Your submission will not be accepted without the readme file]
- The submission must also include the input and output file for Problem 1.
- **Deadline: 23:59 hrs, 31st August, 2022** (Any delay will deduct marks per day basis)
- For plagiarism detected in the code, '0' mark will be awarded

Evaluation components:

- Handling input and output files correctly (Problem 1) **4 points**

- Correct implementation driver program with error handling support (Problem 1) **4 points**
- Correct implementation of pipes (Problem 1) **4 points**
- Correct implementation of server functionality (Problem 2) **3 points**
- Correct implementation of client functionality (Problem 2) **3 points**
- Correct implementation of message queue (Problem 2) **4 points**
- Correct implementation of the game logic (Problem 2) **4 points**
- Content of Readme.txt (Generic) **4 points**