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

Dated 24th August, 2021 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 and not for evaluation. Part - II is the takeaway component and the evaluation policy is stated at the end.

## Part I:

(A) Write a C program to implement pipe. Use fork() to create a child process. Now let the parent process send a set of characters taken as a command line input from the user to the child process. The child will receive the input from the parent, swap the cases of the characters received and send back the updated character stream to the parent. Understand the concept of file descriptors and how dup() and close() system calls can be used to make the re-adjustments of the pipe's input/output for data transfer between the parent and the child process.

**********

**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

1. Extend the first part to implement a distributed merge sort using the concept of Linux system call.

   Method: Randomly generate an array of 100 integers and store it in a file, separated by a whitespace. Provide the pathname of the input file as the command line argument to your program. Now create two processes each time a partition subroutine is called to handle one half of the partition array. At each level the parent process will wait till the both children return with the sorted output. The parent will sort the values received from both the children and return a sorted subarray to its parent in turn. The main process starts with the complete array and at the end kills all other processes (understand why it is needed), prints the output to another file. Your program should be able to handle the corner cases like empty input file or array already sorted.

2. 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 a string S from the user (in a question form), which it sends to all the clients using the message queue. Each client independently sends a reply to that question, denoted by Rj. Now the server converts both the S and each Rj into its equivalent integer value by performing a function H(S) as follows:

$$H(S) = (\sum_{i=1}^{n} a_i)\%n$$

Where $a_i$ is the ASCII value of the ith character in the given string S and n is the length of the S.

The server then computes the difference $\delta = |H(S) - H(R_i)| \forall i \in k$

Among all the clients, the client scoring the smallest $\delta$ 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 message as a

reply to the sent question 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 initial question, the received messages from the clients, 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 program for this problem.

Submission Instructions:

- Submit your codes in a folder as Roll.zip (Includes mergesort.c, server.c and client.c files implementing 1 and 2 component of part II respectively)
- 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, 30th August, 2021** (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) **2 points**
- Correct implementation of mergesort algorithm (Problem 1) **4 points**
- Correct implementation of IPC for sorting the array in a distributed way (Problem 1) **4 points**
- Correct implementation of sever 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) **6 points**