Intercrosses Communication

Exercise 1.1

1. Explain what the flags O_WRONLY, O_APPEND and O_CREAT do

   O_WRONLY -: Writes only
   O_APPEND -: Writes at the end of the file
   O_CREAT -: Create file if it doesn't exist

2. Explain what the modes S_IRUSR, S_IWUSR do.

   S_IRUSR -: User has read permission
   S_IWUSR -: User has write permission

Exercise 1.2

1. Write a program called mycat which reads a text file and writes the output to the standard output.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

int main()
{
 int out;
 size_t size = 100;
 size_t ret;

 out = open("sample.txt", O_RDONLY);

 if(out == -1)
 {
        fprintf(stderr,"Couldn't open the file\n");
        return 1;
 }
```

```
 char* buffer = malloc(size+1); /* size of a char is 1. This buffer holds the read
back value. */

 ret = read(out,buffer,size);

 if(ret == -1)
 {
        fprintf(stderr,"Error reading from file\n");
        return 1;
 }
 buffer[ret] = '\0'; /* we have to null terminate the string ourselves. */

 printf("%s",buffer);
 /* In case there was something already written in the file, the text read back
might not be the same as what was written */


 free(buffer);

 ret = close(out);

 if(ret == -1)
 {
        fprintf(stderr,"Error closing the file after reading.\n");
        return 1;
 }

 return 0;
}
```

2. Write a program called mycopy using open (), read (), write () and close () which takes two arguments, viz. source and target file names, and copy the content of the source file into the target file. If the target file exists, just overwrite the file.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>

int main(int argc, char *argv[])
{
 int in, out;
```

```
 size_t size = 100;
 int ret;

 in = open(argv[1], O_RDONLY);

 if(in == -1)
 {
        fprintf(stderr,"Couldn't open the file\n");
        return 1;
 }

 char* buffer = (char *) calloc(100, sizeof(char)); /* size of a char is 1. This buffer
holds the read back value. */

 ret = read(in,buffer,1000);

 if(ret == -1)
 {
        fprintf(stderr,"Error reading from file\n");
        return 1;
 }

 buffer[ret] = '\0'; /* we have to null terminate the string ourselves. */

 ret = close(in);

 if(ret == -1)
 {
        fprintf(stderr,"Error closing the file after reading.\n");
        return 1;
 }

 out = open(argv[2], O_WRONLY | O_CREAT , S_IRUSR | S_IWUSR ); /*second
parameter is a flag and third a mode. Find out what they do */

 if(out == -1)
 {
        /* the exact error could be found out by looking at the variable errno.
We do not cover it here */
        fprintf(stderr,"Couldn't open the file for writing\n");
        return 1;
 }

 ret = write(out,buffer,strlen(buffer));
```

```
    if(ret == -1)
    {
            fprintf(stderr,"Error writing to file\n");
            return 1;
    }

    free(buffer);

    ret = close(out);

    if(ret == -1)
    {
            fprintf(stderr,"Error closing the file after reading.\n");
            return 1;
    }

    return 0;
}
```

Exercise 2.1

1. What does write (STDOUT_FILENO, &buff, count); do?

   STDOUT_FILENO is the GNU/Linux file descriptor for standard. GNU/Linux sees all devices as files. When a program is started the operating system opens a path to standard out and assigns it file descriptor number 1.

   &buff is a memory address. The sequence of one-byte bit patterns starting at this address will be sent to standard out by the `write` function.

   Count is the number of bytes that will be sent (to standard out) as a result of this call to write.

2. Can you use a pipe for bidirectional communication? Why (not)?

   Can't use a pipe for bidirectional communication

   Pipes are unidirectional: write at the write end, and read from the read end.

   If you want bidirectional communication, have to use a socket, or two pipes. Can use a pipe for bidirectional communication if both processes keep both ends of the pipe open. Need to define a protocol for whose turn it is to talk. This is highly impractical.

3. Why cannot unnamed pipes be used to communicate between unrelated processes?

   Because unnamed pipes handle one-way communication. It is typically used to communicate between a parent process and a child process.

4. Now write a program where the parent reads a string from the user and send it to the child and the child capitalizes each letter and sends back the string to parent and parent displays it. You'll need two pipes to communicate both ways.

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#define READ_END 0

#define WRITE_END 1

const char banner [] = "hello there\n";

int main()

{

	int pipe_ptoc[2];

	int pipe_ctop[2];

	pid_t pid;



	if(pipe(pipe_ptoc))

	{

		perror("Pipe creation");

		return -1;

	}

	if(pipe(pipe_ctop)) {
```

```
            perror("Pipe creation");
            return -1;
}
pid = fork();
if(pid < 0)
{
            perror("Fork");
            return -1;
}
if(pid > 0)
{ /* parent */
            close(pipe_ptoc[0]);
            close(pipe_ctop[1]);

            int count;
            char buffer[100];
            printf("Enter the string: ");
            scanf("%s",buffer);

            write(pipe_ptoc[1], buffer, strlen(buffer));

            count = read(pipe_ctop[0], buffer, 100);
            buffer[count] = '\0';
```

```
            printf("%s\n",buffer);
        }
        if(pid == 0)
        { /* child */
            char buff[100];
            close(pipe_ptoc[1]);
            close(pipe_ctop[0]);
            int count;
            while(1) {
                count = read(pipe_ptoc[0], buff, 128);
                for(int i = 0; i < count; i++){
                    buff[i] = buff[i] - 32;
                }
                write(pipe_ctop[1], buff, count);
            }
        }
        return 0;
    }
```

Exercise 3.1

1. Write a program that uses fork () and exec () to create a process of ls and get the result of ls back to the parent process and print it from the parent using pipes. If you cannot do this, explain why.

   When exec () is used to replace the execution image of a forked child, all the communication means are lost since exec () replaces all the original code

Exercise 3.2

1. What does 1 in the line dup2(out,1); in the above program stands for?

   1 is new file descriptor which is used by dup2() to create a copy.

2. The following questions are based on the example3.2.c

   (i) Compare and contrast the usage of dup () and dup2(). Do you think both functions are necessary? If yes, identify use cases for each function. If not, explain why.

   The difference between dup and dup2 is that dup assigns the lowest available file descriptor number, while dup2 let us choose the file descriptor number that will be assigned and automatically closes and replaces it if its already taken.

   (ii) There's one glaring error in this code (if you find more than one, let me know!). Can you identify what that is (hint: look at the output)?

   The program doesn't terminate after displaying the search result

   (iii) Modify the code to rectify the error you have identified above.

```
#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#define INPUTFILE "fixtures"

/* function prototypes */

void die(const char*);


int main(int argc, char **argv)
```

```
{
        int pipefd[2];
        int pid;
        if (argc < 2)
        {
                printf("%s: missing operand\n", argv[0]);
                printf("Usage: %s <search_term in %s>\n",
argv[0],INPUTFILE);
                exit(EXIT_FAILURE);
        }
        char *cat_args[] = {"cat", INPUTFILE, NULL};
        char *grep_args[] = {"grep", "-i", argv[1], NULL};
        // make a pipe (fds go in pipefd[0] and pipefd[1])
        if(pipe(pipefd) == -1)
                die("pipe()");
        pid = fork();
        if(pid == (pid_t)(-1))
                die("fork()");
        if (pid == 0)
        {
                // child gets here and handles "grep <search_term>"
                // Close standard input
                close(0);
```

```
        // replace standard input with input part of pipe
        if(dup2(pipefd[0],0) == -1)
                die("dup()");
        // close unused hald of pipe
        close(pipefd[1]);
        // execute grep
        if(execvp("grep", grep_args) == -1){
                //printf("go\n");
                die("execvp()");
        }
        exit(EXIT_SUCCESS);
}
else
{
        // parent gets here and handles "cat INPUTFILE"
        // close standard output
        close(1);
        // replace standard output with output part of pipe
        if(dup2(pipefd[1],1) == -1)
                die("dup()");

        // close unused input half of pipe
        close(pipefd[0]);
```

```
                // execute cat
                if(execvp("cat", cat_args) == -1)
                        die("execvp()");
                exit(EXIT_SUCCESS);
        }
}
/* A better way to Die (exit) */
void die(const char *msg) {
        perror(msg);
        exit(EXIT_FAILURE);
}
```

3. Write a program that executes "cat fixtures | grep <search_term> | cut -b 1-9" command. A skeleton code for this is provided as exercise3.2.c_skel.c. You can use this as your starting point, if necessary.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define INPUTFILE "fixtures"

int main(int argc, char **argv)
{
```

```
int status;
int i;

char *cat_args[] = {"cat", INPUTFILE, NULL};
char *grep_args[] = {"grep", "-i", argv[1], NULL};
char *cut_args[] = {"cut", "-b", "1-9", NULL};

// make 2 pipes (cat to grep and grep to cut); each has 2 fds
int pipes[4];
pipe(pipes); // sets up 1st pipe
pipe(pipes + 2); // sets up 2nd pipe

if (fork() == 0)
 {
   // replace cat's stdout with write part of 1st pipe
   dup2(pipes[1], 1);

   // close all pipes (very important!); end we're using was safely copied
   close(pipes[0]);
   close(pipes[1]);
   close(pipes[2]);
   close(pipes[3]);

   execvp(*cat_args, cat_args);
```

```
  }
else
  {
    // fork second child (to execute grep)
    if (fork() == 0)
          {
            // replace grep's stdin with read end of 1st pipe
            dup2(pipes[0], 0);

            // replace grep's stdout with write end of 2nd pipe
            dup2(pipes[3], 1);

            // close all ends of pipes
            close(pipes[0]);
            close(pipes[1]);
            close(pipes[2]);
            close(pipes[3]);

            execvp(*grep_args, grep_args);
          }
    else
          {
            // fork third child (to execute cut)
            if (fork() == 0)
```

```
            {
              // replace cut's stdin with input read of 2nd pipe
              dup2(pipes[2], 0);

              // close all ends of pipes
              close(pipes[0]);
              close(pipes[1]);
              close(pipes[2]);
              close(pipes[3]);

              execvp(*cut_args, cut_args);
            }
          }
  }

  // only the parent gets here and waits for 3 children to finish
  close(pipes[0]);
  close(pipes[1]);
  close(pipes[2]);
  close(pipes[3]);

  for (i = 0; i < 3; i++)
    wait(&status);
}
```

Exercise 4.1

1. Comment out the line “mkfifo(fifo,0666);” in the reader and recompile the program. Test the programs by alternating which program is invoked first. Now, reset the reader to the original, comment the same line in the writer and repeat the test. What did you observe? Why do you think this happens? Explain how such an omission (i.e., leaving out mkfifo()function call in this case) can make debugging a nightmare

   When the mkfifo(fifo,0666); is commented in the reader file, while running the reader file when write file ran, the output is displayed in the reader file with some garbage values

   When the mkfifo(fifo,0666); is commented in the writer file, while running the reader file when write file ran, the output is displayed in the reader with some garbage values

   mkfifo() makes a FIFO special file with name fifo.
   Here 0666(mode) specifies the FIFO’s permissions. It is modified by the process’s umask in the usual way: the permissions of the created file are (mode & ~umask).

2. Write two programs: one, which takes a string from the user and sends it to the other process, and the other, which takes a string from the first program, capitalizes the letters and send it back to the first process. The first process should then print the line out. Use the built in command tr() to convert the string to uppercase.

   //Write

```
#include <fcntl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>
```

```
int main()
{
    char buffer[100];
    char receive[100];

    printf("Enter the Word: ");
    scanf("%s",buffer);

    int fd,fdd;
    char * fifoSimple = "/tmp/fifoSimple";
    char * fifoCapital = "/tmp/fifoCapital";

    mkfifo(fifoSimple,0666);
    mkfifo(fifoCapital,0666);

    fd = open(fifoSimple, O_WRONLY);
    write(fd, buffer,strlen(buffer));
    close(fd);

    fdd = open(fifoCapital, O_RDONLY);
    read(fdd, receive, 1024);
    printf("Capitalized Word: %s\n", receive);
    close(fdd);
```

```
    unlink(fifoSimple);
    unlink(fifoCapital);

    return 0;
}
//Read
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 1024

int main()
{
    int fd,fdd;
    char* fifoSimple = "/tmp/fifoSimple";
    char* fifoCapital = "/tmp/fifoCapital";

    mkfifo(fifoSimple,0666);
```

```
    mkfifo(fifoCapital,0666);


    fd = open(fifoSimple, O_RDONLY);

    dup2(fd, fileno(stdin));

    fd = open(fifoCapital, O_WRONLY);

    dup2(fd, fileno(stdout));


    char *tr_args[] = {"tr", "[:lower:]","[:upper:]", NULL};


    execvp(tr_args[0], tr_args);


    close(fd);

    close(fdd);


    return 0;

}
```