FAST NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, PESHAWAR

DEPARTMENT OF COMPUTER SCIENCE

CL220 - OPERATING SYSTEMS LAB

LAB MANUAL # 09

INSTRUCTOR: MUHAMMAD ABDULLAH

SEMESTER SPRING 2021

Contents

Input and Output
(File Descriptors)

Input and Output File Descriptors

Probably the first thing you covered in I/O when you were doing your C/C++ programming courses was **cin, cout or cerr**. Of these, cout displays something on the standard output (display screen), whereas cin is used for obtaining input from keyboard input device.

In linux, there are three types of files that are open all the time for input and output purposes for each process. These are:

1. Standard Input Stream

2. Standard Output Stream

3. Standard Error Stream

Each of these streams are represented by a unique integer number called a ***File Descriptor***. In this case, standard input is **0**, standard output is **1**, and standard error is **2**. Other files that are in use by a process will be assigned file descriptor numbers of **3**, **4**,…These files descriptors refer to each and every instance of an open file for a process. So, if we want to open a file, close a file, read from a file, or write to a file, it has to be done through its corresponding file descriptor.

To visualize how this works, we are going to perform a simple exercise. For this, we would require two shells.

- Type the following command and note down the current bash shell PID. Let this be **X**.

  ps

- Press CTRL+ALT+T to open a new terminal window and login with your details.
- The echo command is used to display a line of text.

echo "Hello"

- We are going to use the echo command to write a string Hello to a file hello.txt using the output redirection method that has been covered in previous Sections. You can view the contents of this file to ensure that the string has indeed been written to it.

echo "Hello" > hello.txt

nano hello.txt

- In previous bullet, hello.txt was a file. We are going to use the same mechanism but a little differently. Replace X with the PID value that you noted down earlier. You will see what /proc directory is used for in later sections. We are specifying that we want to write to file descriptor= 1 of process ID marked by X.

  echo "Hello" > /proc/X/fd/1

- Now go back to the previous terminal by pressing CTRL+ALT+T. Surprise! The string Hello has been written there.

Command line arguments in C/C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems. To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

- **argc (ARGument Count)** is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)
- The value of argc should be non negative.
- **argv(ARGument Vector)** is array of character pointers listing all the arguments.
- If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.
- Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

For better understanding run this code on your linux machine.

mainfunction.cpp

```
#include <iostream>
using namespace std;
int main(int argc, char** argv)
{
        cout << "You have entered " << argc
                << " arguments:" << "\n";
        for (int i = 0; i < argc; i++)
                cout << argv[i] << "\n";
        return 0;
}
```

Input:

```
$ g++ mainfunction.cpp -o main
$ ./mainfunction test test1 test2
```

Output:

```
You have entered 4 arguments:
./mainfunction
test
test1
test2
```

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

Note : You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes “” or single quotes ”.

mainfunction.c

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main(int argc, char* argv[])   //char* argv[] is an array of Strings
{
    //argc will be the number of strings pointed to by argv
    printf("Total Number of strings passed to char* argv[]: %d",argc);
    printf("\n");
  //argc will be the number of strings pointed to by argv
    printf("Total Number of strings passed to char* argv[]: %d",argc);
    printf("\n");
    /*
    for (int i = 0; i < argc; i++)
    {
      printf("%s\n", argv[i]);
    }*/
    /*
    char* name[]= {"Ali", "Hassan", "Asim"};
    for(int i=0; i<4; i++)
    {
            printf("%s\n", name[i]);
    }
    */
    printf("First Argument is: %s",argv[0]);
    printf("\n");
    printf("Second Argument is: %s",argv[1]);
    printf("\n");
    printf("Third Argument is: %s",argv[2]);
    printf("\n");
    printf("4th Argument is: %s",argv[3]);
    printf("\n");
    return 0;
}
```

Input:

```
$ gcc mainfunction.c -o mainfunction
$ ./mainfunction test test1 test2
```

Output:

```
Total Number of strings passed to char* argv[]: 4
First Argument is: ./mainfunction
Second Argument is: test
Third Argument is: test1
4th Argument is: test2
```

1. Open a File

We use the open() call to open a file. Open() can also be used for creating a new file. It's syntax is as such:

int open(pathname, flags, modes);

This would work by including the *sys/types.h*, *sys/stat.h* and *fcntl.h* C libraries. As parameters, it takes the following:

1. Path name of the file to open.

2. Flag specifying how to open it (i.e., for read only, write only, etc.)

3. Access permissions for a file (Provided the file is newly created).

Some of the flags are listed below:

- O_RDONLY for marking a file as read only
- O_WRONLY for marking a file as write only
- O_RDWR for marking a file as read and write
- O_CREAT for creating the new fie
- O_EXCL for giving an error when creating a new file and that file already exists.

As an example, run the following code for creating a new file:

openfile.c

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main(int argc, char* argv[])
{
   char *path = argv[1];
   //printf("Total Arguments: %d",argc);
   //printf("\n");
   //int fd= open ("myfile33.txt", O_WRONLY | O_EXCL | O_CREAT);
   int fd= open (path, O_WRONLY | O_EXCL | O_CREAT);
```

```
        if(fd==-1)
        {
         printf("Error: File not created\n");
         return 1;
        }
        //printf("First Argument is: %s",argv[0]);
        //printf("\n");
        //printf("Second Argument is: %s",argv[1]);
        //printf("\n");
        //printf("Third Argument is: %s",argv[2]);
        //printf("\n");
        return 0;
    }
```

Compile this, but when running specify the command as such:

```
gcc openfile.c -o openfile
./ openfile createThisFile
```

Then run ls and check if the file has been created.

```
ls -l
```

Give the same command again and check out the output.

Question What is the size of the file? Why is it this size?

2. Close a File

So we saw that the open() call returns an integer number (the file descriptor) which has been stored in fd variable. Once we are finished with what we are doing with the file, the file should be closed. When a process terminates, linux by default closes all file descriptors so you may not close a file by yourself if you don't want to.

But if you do, then read on.

Linux uses a total of 1,024 file descriptors per process by default. So, it's a good idea that you close your file descriptor's if they are not used. To close a file descriptor, just use the following in your code:

```
close(fd);
```

Look at the following code and see what it is the output?

closefile.c

```
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
int main(int argc, char* argv[])
{
   if(argc!=2)
   {
        printf("Error: Run like this:\n");
        printf("%s name-of-new-file\n", argv[0]);
        return 1;
   }
   char *path = argv[1];
   int i=0;
   while(i<4)
   {
        int fd= open (path, O_WRONLY | O_CREAT);
        printf("Created! Descripter is %d\n", fd);
        close(fd);
        i++;
   }
```

Comment out the line *close(fd);* and then compile and run again. What is the output this time? Why do you think you are getting different values?

3. Writing to a file

Writing to a file is done using the write call. To write, we should obviously open a file first. The syntax of the write() call is as such:

write(fd, buffer, size);

This would require the unistd.h C library. The following are the parameters used by the call:

- The file descriptor (must be open ... otherwise how are you going to write to a not open file?)
- Buffer or place where data is located
- Length to write

So, let's see the write call in action. Type, compile and run the following code. Then answer the questions at the end.

writefile.c

```
#include<stdio.h>

#include<fcntl.h>

#include<sys/stat.h>

#include<sys/types.h>

#include<unistd.h>

#include<string.h>

#include<time.h>

char* get_timeStamp()

{

   time_t now=time(NULL);

   return asctime(localtime(&now));

}

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

{

   char *filename = argv[1];

   char *timeStamp=get_timeStamp();

  int fd= open(filename,O_WRONLY |O_APPEND | O_CREAT,0666);
```

```
    //int fd= open(filename,O_WRONLY |O_CREAT,0666);
    size_t lenght = strlen(timeStamp);
    write(fd, timeStamp, lenght);
    close(fd);
    return 0;
}
```

Q1: What is 0666 that is specified in the open() call? What does it mean?

Q2: What is O_APPEND doing in the same call? Run the program again and check it's output.

Q3: Modify the following line in the code and then compile and run the program and check its output. What has happened?

From:

```
size_t length = strlen(timeStamp);
```

To:

```
size_t length = strlen(timeStamp)-5;
```

4. Reading from a file

Read() system call is going to be used for this purpose. Its syntax is as such:

```
read(fd, buffer, size);
```

This would require the `unistd.h` C library. The parameters for read() are somewhat the same

as that for write(). i.e.,

- The file descriptor (must be open ... otherwise how can you read from a not-open file?)
- Buffer or place where data is to be saved after reading
- Length to read

So, let's see the read in action. Type, compile, and run the following code:

readfile.c

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<sys/types.h>
#include<unistd.h>
#include<string.h>
#include<time.h>
int main(int argc, char* argv[])
{
   if(argc!=2)
   {
        printf("Error: Run like this:\n");
        printf("%6s name-of-new-file\n", argv[0]);
        return 1;
   }
   char *path = argv[1];
   int fd= open(path,O_RDONLY);
   if(argc==-1)
   {
        printf("File Does not Exist\n");
        return 1;
   }
   char buffer[200];
   read(fd, buffer, sizeof(buffer)-1);
   printf("Contents of file are:\n");
   printf("%s\n", buffer);
   close(fd);
```

```
    return 0;
}
```

References

https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/

https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/