Chapter 4

Input/Output

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 file 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 it's corresponding file descriptor.

To visualise 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+F2 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 Section-2.4.4. 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+F1. Surprise! The string Hello has been written there.

4.1 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 le (Provided the le is newly created)

Some of the ags are listed below:

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

As an example, run the following code for cre ating a new le:

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

int main(int argc, char* argv[])
{
  char *path = argv[1];
  int fd = open(path, O_WRONLY | O_EXCL |
                                          O_CREA
                                          T);
  if (fd == -1)
  {
    printf("Error: File not Created\n"); return 1;
  }
  return 0;
}
```

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

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

Then run ls and check if the le has been cre ated.

```
ls
```

Give the same command again and check out the output.

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

4.2 Close a File

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

Linux uses a total of 1,024 le descriptors per process by default. So it's a good idea that you close your le descriptor's if they are not used.

To close a le descriptor, just use the follow ing in your code:

```
close(fd);
```

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

```
#include <fcntl.h>
#include <stdio.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: ");
    printf("%6s name-of-new-file\n", argv[0]); return
    1;
  }
  char *path = argv[1];
  int i = 0;
  while(i<2)
  {
    int fd = open(path, O_WRONLY | O_CREAT);
    printf("Created! Descriptor is %d\n", fd); close(fd);
    i++;
  }
  return 0;
}
```

Comment out the line close(fd); and then com pile and run again. What is the output this time? Why do you think you are getting dif ferent values?

4.3 Writing to a File

Writing to a le is done using the write call. To write, we should obviously open a le rst. 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 paremeters used by

the call:

- The le descriptor (must be open ... oth erwise how are you going to write to a not open le?)
- Bu er or place where data is located
- Length to write

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

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.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);
 size_t length = strlen(timeStamp);
 write(fd, timeStamp, length);
 close(fd);
 return 0;
}
```

Q1 What is 0666 that is speci ed 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 it's output. What has happened?

From:

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

To:

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

4.4 Reading from a File

Read() system call is going to be used for this purpose. It's 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 le descriptor (must be open ... oth erwise how can you read from a not-open le?)
- Bu er 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:

```
#include <fcntl.h>
#include <stdio.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: ");
  printf("%6s name-of-existing-file\n", argv[0]);
  return 1;
 }
 char *path = argv[1];
 int fd = open(path, O_RDONLY);
 if (fd == -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;
}
```