Comprehensive Guide to I/O Operations in Linux

Table of Contents

1. Introduction to I/O in Linux

In Linux, there are three types of files that are always open for input and output purposes for each process:

1. Standard Input Stream (File Descriptor: 0)
2. Standard Output Stream (File Descriptor: 1)
3. Standard Error Stream (File Descriptor: 2)

These streams are represented by unique integer numbers called File Descriptors.

Exercise: Visualizing File Descriptors

Follow these steps to visualize how file descriptors work:

1. Open a terminal and type `ps` to note down the current bash shell PID (let's call it X).
2. Press CTRL+ALT+F2 to open a new terminal window and login.
3. Type `echo "Hello"` to display a line of text.
4. Create a file with `echo "Hello" > hello.txt`
5. View the contents with `nano hello.txt`
6. Write to the file descriptor of the first terminal: `echo "Hello" > /proc/X/fd/1`
7. Switch back to the first terminal (CTRL+ALT+F1) to see the result.

2. File Descriptors

File descriptors are unique integer numbers assigned to open files in a process. They start from 0 (standard input) and go up to 1023 by default.

3. Opening a File

To open a file, use the `open()` system call:

```
int open(pathname, flags, modes);
```

Required Headers:

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

Parameters:

1. `pathname`: Path to the file
2. `flags`: Specifies how to open the file
3. `modes`: Access permissions (for newly created files)

Common Flags:

- `O_RDONLY`: Read-only
- `O_WRONLY`: Write-only
- `O_RDWR`: Read and write
- `O_CREAT`: Create a new file
- `O_EXCL`: Give an error if the file already exists (used with O_CREAT)

Example: Creating a New File

```
#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_CREAT);
    if (fd == -1)
    {
        printf("Error: File not Created\n");
        return 1;
    }
    return 0;
}
```

Compile and run:

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

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

Answer: The file size will be 0 bytes. This is because we only created the file but didn't write any data to it.

4. Closing a File

To close a file, use the `close()` system call:

```
close(fd);
```

Example: Opening and Closing Files

```
#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;
}
```

Question: What happens when you comment out the `close(fd);` line?

Answer: When you comment out the `close(fd);` line, you'll see different file descriptor numbers being returned. This is because without closing the file, the system keeps allocating new file descriptors for each `open()` call.

5. Writing to a File

To write to a file, use the `write()` system call:

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

Required Header:

```
#include <unistd.h>
```

Parameters:

1. `fd`: File descriptor
2. `buffer`: Data to be written
3. `size`: Length of data to write

Example: Writing a Timestamp to a File

```
#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;
}
```

Questions and Answers:

Q1: What is 0666 that is specified in the open() call? What does it mean? A1: 0666 is the file permission mode in octal notation. It means the file will be created with read and write permissions for the owner, group, and others.

Q2: What is O_APPEND doing in the same call? Run the program again and check its output. A2: O_APPEND flag ensures that the data is appended to

the end of the file. If you run the program multiple times, you'll see multiple timestamps in the file, one after another.

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;` A3: By reducing the length by 5, you're truncating the last 5 characters of the timestamp. This will likely cut off the year and newline character from the timestamp in the file.

6. Reading from a File

To read from a file, use the `read()` system call:

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

Required Header:

```
#include <unistd.h>
```

Parameters:

1. `fd`: File descriptor
2. `buffer`: Buffer to store read data
3. `size`: Maximum number of bytes to read

Example: Reading from a File

```
#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;
}
```

7. Practice Exercises

1. Create a program that opens a file, writes some text to it, closes the file, then opens it again and reads the content.
2. Modify the timestamp writing program to allow the user to input a message, then write both the message and the timestamp to the file.
3. Create a program that copies the contents of one file to another using `read()` and `write()` system calls.

Remember to compile your C programs using `gcc` and run them with any necessary command-line arguments.

This guide covers the basics of file I/O operations in Linux using C programming. Practice these concepts to become proficient in handling file operations in Linux environments.