Reading Material for File Operations

As Part of

Operating Systems [CS F372] Course

Semester I, 2023 – 2024

Prepared by

Harsh Gujarathi [2020A7PS1712G]

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI –K K BIRLA GOA CAMPUS

INDEX

Command Line Arguments -

Command-line arguments are the values given after the name of the program in the command-line shell of Operating Systems. Command-line arguments are handled by the main() function of a C program.

Syntax -

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

- **argc** (ARGument Count) is an integer variable that stores the number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, the 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 an 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**.

Example:

```
./hello.out check me out 3
```

Here,

`./hello.out` is the name of the program and can be accessed in the program using argv[0].

`check`, `me`, `out`, `3` will be accessible using argv[1], argv[2], argv[3] and argv[4] respectively.

File Operations:

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as fopen(), fwrite(), fread(), fseek(), fprintf(), etc. to perform input, output, and many different C file operations in our program.

1. **fopen()**

Syntax - FILE *fopen (const char *pathname, const char *mode);
The fopen() function opens the file whose name is the string pointed to by pathname and associates a stream with it.

2. **fclose()**

Syntax - int fclose (FILE *stream);
The fclose() function flushes the stream pointed to by stream and closes the underlying file descriptor.

3. **fprintf()**

Syntax - int fprintf (FILE *fptr, const char *str, …);
fprintf is used to print content in file instead of stdout console.
Example -

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    FILE* demo;
    demo = fopen("os_demo.txt", "w+");
    fprintf(demo, "%s %s %s", "Welcome", "to", "Operating
Systems");
    fclose(demo);
    return 0;
}
```

4. **fscanf()**

Syntax - int fscanf (FILE *ptr, const char *format, …)
fscanf reads from a file pointed by the FILE pointer (ptr), instead of reading from the input stream.

Example 2 - Sample C code which reads data from file using fscanf.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *demo;
    FILE *copiedDemo;
    demo = fopen("os_demo.txt", "r");
    copiedDemo = fopen("copied_demo.txt", "w");
    if (demo == NULL || copiedDemo == NULL) {
        perror("Error opening files");
        return 1;
    }
    char buffer[100];
    while (fscanf(demo, "%s", buffer) == 1) {
        fprintf(copiedDemo, "%s ", buffer);
    }
    fclose(demo);
    fclose(copiedDemo);
    return 0;
}
```

5. **fseek()**

Syntax - int fseek (FILE *pointer, long int offset, int position);
fseek() is used to move the file pointer associated with a given file to a specific position.

6. **fread()**

Syntax - size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
The function fread() reads nmemb items of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

7. **fwrite()**

Syntax - size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
The function fwrite() writes nmemb items of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

8. **ftell()**

Syntax - long ftell(FILE *stream);

The ftell() function obtains the current value of the file position indicator for the stream pointed to by stream.

Example 3 - C Program to copy the contents of cone file to another file using fread, fwrite, ftell, and fseek.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *demo;
    FILE *copiedDemo;
    demo = fopen("os_demo.txt", "r");
    copiedDemo = fopen("copied_demo.txt", "w");
    if (demo == NULL || copiedDemo == NULL) {
        perror("Error opening files");
        return 1;
    }
    fseek(demo, 0, SEEK_END);
    long fileSize = ftell(demo);
    fseek(demo, 0, SEEK_SET);
    char *content = (char *)malloc(fileSize + 1);
    if (content == NULL) {
        perror("Memory allocation error");
        return 1;
    }
    fread(content, 1, fileSize, demo);
    content[fileSize] = '\0';
    fwrite(content, 1, fileSize, copiedDemo);
    fclose(demo);
    fclose(copiedDemo);
    free(content);
    return 0;
}
```

9. **fgets()**

Syntax - char *fgets (char *str, int n, FILE *stream);
The fgets() reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.

10. **fgetpos()**

Syntax - int fgetpos(FILE *stream, fpos_t *pos);
The C function fgetpos gets the current file position of the stream and writes it to pos.

11. **fsetpos()**

Syntax - int fsetpos(FILE *stream, const fpos_t *pos);
The fsetpos() function moves the file position indicator to the location specified by the object pointed to by position. When fsetpos() is executed, the end-of-file indicator is reset.

Example 4 - C code that demonstrates the use of fgets(), fgetpos(), and fsetpos() to read lines from a text file, get and set file positions

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *file;
    file = fopen("sample.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1; // Exit with an error code
    }
    char buffer[100];
    fpos_t position;
    // Read and print the first three lines
    for (int i = 0; i < 3; i++) {
        if (fgets(buffer, sizeof(buffer), file) != NULL) {
            printf("Line %d: %s", i + 1, buffer);
        }
        else {
            perror("Error reading file");
            break;
        }
    }
    // Get the current file position
    if (fgetpos(file, &position) != 0) {
        perror("Error getting file position");
        return 1;
    }
    // Move back to the beginning of the file
    rewind(file);
    // Skip the first three lines
    for (int i = 0; i < 3; i++) {
        if (fgets(buffer, sizeof(buffer), file) == NULL) {
            perror("Error reading file");
            break;
        }
    }
    // Set the file position back to the previously saved
position
    if (fsetpos(file, &position) != 0) {
        perror("Error setting file position");
        return 1;
    }
    // Read and print lines from the saved position
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("Restored Position: %s", buffer);
    }
    fclose(file);
    return 0;
}
```

Note - Please refer to the man pages for more detailed description and usage.

Input Output Redirections -

In Linux, whenever an individual runs a command, it can take input, give output, or do both. Redirection helps us redirect these input and output functionalities to the files or folders we want, and we can use special commands or characters to do so.

Types of Redirection -

1. **Overwrite Redirection**

Overwrite redirection is useful when you want to store/save the output of a command to a file and replace all the existing content of that file.

Types -

“>” standard output

So whatever you will write after running this command, will be redirected and copied to the “file.txt”. This is standard output redirection.

For Example -

```
cat > sample.txt
I love Operating Systems.
```

This will **overwrite** the existing contents of the file sample.txt and write “I love Operating Systems.” to it.

To verify use,

```
cat sample.txt
I love Operating Systems.
```

To close the command press Ctrl + D.

“<” standard input

For Example -

```
cat < sample.txt
I love Operating Systems.
```

Now, this is standard input redirection, cat command will take the input from “sample.txt” and print it to the terminal screen.

2. **Append Redirection:**

With the help of this Redirection, you can append the output to the file without compromising the existing data of the file.

“>>” standard output

For Example -

```
cat >> sample.txt
I love C Programming.
```

This will **append** to the existing contents of the file sample.txt and write “I love C Programming”. to it.

To verify use,

```
cat sample.txt
```

```
I love Operating Systems.
I love C Programming.
```

“<<” standard input
Example -

```
cat << EndOfText
This is an example of a document here.
You can input multiple lines of text here.
EndOfText

This is an example of a document here.
You can input multiple lines of text here.
```

3. **Merge Redirection:**

This allows you to redirect the output of a command or a program to a specific file descriptor instead of standard output.
The syntax for using this is “>&” operator followed by the file descriptor number.

- “p >& q” Merges output from stream p with stream q
- “p <& q” Merges input from stream p with stream q

Some Useful commands -

- **man -** man command in Linux is used to display the user manual of any command that we can run on the terminal.
  Syntax - $ man [COMMAND NAME]
  Example - $ man fprintf

- **apropos -** apropos command helps the user when they don't remember the exact command but knows a few keywords related to the command that define its uses or functionality. It searches the Linux man page with the help of the keyword provided by the user to find the command and its functions.
  Syntax - apropos [OPTION..] KEYWORD…
  Example - $ apropos print first lines