Lecture_3_File_Access_and_Pipes

February 3, 2025

1 Introduction

Today, we'll dive into the core principles of file access and pipes in C:

1. Understanding how to open, read, write, and close files using the standard I/O functions.
2. How inter-process communication (IPC) with pipes enable data exchange between processes.
3. Creating unidirectional communication channels between parent and child processes.

Understanding file access and pipes in C is crucial because these concepts form the foundation of how C programs interact with data storage and communicate between processes.

---

2 Files

The `file` is the most basic and fundamental abstraction in Linux. Linux follows the **everything-is-a-file** philosophy (although not as strictly as some other systems, such as Plan 91). Consequently, much interaction occurs via reading of and writing to files, even when the object in question is not what you would consider a normal file.

In order to be accessed, a file must first be opened. Files can be opened for reading, writing, or both. An open file is referenced via a unique descriptor, a mapping from the metadata associated with the open file back to the specific file itself. Inside the Linux kernel, this descriptor is handled by an integer (of the C type `int`) called the **file descriptor**, abbreviated **fd**. File descriptors are shared with user space, and are used directly by user programs to access files. A large part of Linux system programming consists of opening, manipulating, closing, and otherwise using file descriptors.

1 Plan9, an operating system born of Bell Labs, is often called the successor to Unix. It features several innovative ideas, and is an adherent of the everything-is-a-file philosophy.

2.1 Regular files

What most of us call "files" are what Linux labels regular files. A regular file contains bytes of data, organized into a linear array called a byte stream.

Any of the bytes within a file may be read from or written to. These operations start at a specific byte, which is one's conceptual "location" within the file. This location is called the **file position** or **file offset**. The file position is an essential piece of the metadata that the kernel associates with each open file. When a file is first opened, the file position is zero. Usually, as bytes in the file are

read from or written to, byte-by-byte, the file position increases in kind. The file position may also be set manually to a given value, even a value beyond the end of the file.

The size of a file is measured in bytes and is called its **length**. The length, in other words, is simply the number of bytes in the linear array that make up the file. A file's length can be changed via an operation called truncation. A file can be truncated to a new size smaller than its original size, which results in bytes being removed from the end of the file. Confusingly, given the operation's name, a file can also be "truncated" to a new size larger than its original size. In that case, the new bytes (which are added to the end of the file) are filled with zeros. A file may be empty (that is, have a length of zero), and thus contain no valid bytes. The maximum file length, as with the maximum file position, is bounded only by limits on the sizes of the C types that the Linux kernel uses to manage files.

A single file can be opened more than once, by a different or even the same process. Each open instance of a file is given a unique file descriptor. Conversely, processes can share their file descriptors, allowing a single descriptor to be used by more than one process. The kernel does not impose any restrictions on concurrent file access. Multiple processes are free to read from and write to the same file at the same time. The results of such concurrent accesses rely on the ordering of the individual operations, and are generally unpredictable. User-space programs typically must coordinate amongst themselves to ensure that concurrent file accesses are properly synchronized.

Although files are usually accessed via **filenames**, they actually are not directly associated with such names. Instead, a file is referenced by an *inode* (originally short for *information node*), which is assigned an integer value unique to the filesystem (but not necessarily unique across the whole system). This value is called the inode number, often abbreviated as `i-number` or `ino`. An inode stores metadata associated with a file, such as its modification timestamp, owner, type, length, and the location of the file's data—but no filename! The inode is both a physical object, located on disk in Unix-style filesystems, and a conceptual entity, represented by a data structure in the Linux kernel.

3 Accessing and Writing to a File

File access in C is managed using the `standard I/O library`, which provides functions to interact with files. An external file can be opened, read from, written to, and closed in a C program. For these operations, C includes the `FILE` type for defining a file stream. The file stream keeps track of where reading and writing last occurred.

The `stdio.h` library includes file handling functions: `FILE` typedef for defining a file pointer.

`FILE` is a `structure` that stores information about a file stream, including:

- A file descriptor
- Current position in the file
- Error and EOF indicators
- Buffering information

`fopen(filename, mode)` returns a `FILE` pointer to file `filename` which is opened using `mode`. If a file cannot be opened, `NULL` is returned.

`NULL` is a **macro** in C that represents a null pointer. It is commonly used to indicate an invalid or uninitialized pointer, especially in file handling, memory allocation, and linked data structures.

The `fopen(filename, mode)` function is used to open files in different modes. Below is a table summarizing the available modes and their behavior:

Mode	Description	File Existence Requirement	File Pointer Position	Behavior if File Exists	Behavior if File Doesn't Exist
`"r"`	Open for **reading**	File must exist	Beginning of file	Opens file for reading	Returns `NULL`
`"w"`	Open for **writing**	File may or may not exist	Beginning of file	Truncates file to zero length	Creates a new file
`"a"`	Open for **appending**	File may or may not exist	End of file	Appends data to the end	Creates a new file
`"r+"`	Open for **reading and writing**	File must exist	Beginning of file	Opens file for read/write	Returns `NULL`
`"w+"`	Open for **reading and writing**	File may or may not exist	Beginning of file	Truncates file to zero length	Creates a new file
`"a+"`	Open for **reading and appending**	File may or may not exist	End of file	Appends data to the end	Creates a new file
`"rb"`	Open for **reading in binary mode**	File must exist	Beginning of file	Opens file for reading	Returns `NULL`
`"wb"`	Open for **writing in binary mode**	File may or may not exist	Beginning of file	Truncates file to zero length	Creates a new file
`"ab"`	Open for **appending in binary mode**	File may or may not exist	End of file	Appends data to the end	Creates a new file
`"r+b"`	Open for **reading and writing in binary mode**	File must exist	Beginning of file	Opens file for read/write	Returns `NULL`
`"w+b"`	Open for **reading and writing in binary mode**	File may or may not exist	Beginning of file	Truncates file to zero length	Creates a new file
`"a+b"`	Open for **reading and appending in binary mode**	File may or may not exist	End of file	Appends data to the end	Creates a new file

`fclose(fptr)` closes file opened with `FILE *fptr`, returning `0` if close was successful. `EOF` (end of file) is returned if there is an error in closing.

3.1 Writing to a File

Function	Description	Parameters	Return Value	Key Behavior
`fputc()`	Writes a single character to a file.	`int char, FILE *stream`	The character written or `EOF` on error	- Writes one character to the file.- Useful for low-level character output.

Function	Description	Parameters	Return Value	Key Behavior
`fputs()`	Writes a string to a file (without a newline).	`const char *str, FILE *stream`	Non-negative value on success, `EOF` on error	- Writes a string to the file.- Does not automatically add a newline.
`fprintf()`	Writes formatted output to a file.	`FILE *stream, const char *format, ...`	Number of characters written or negative value on error	- Writes formatted data to the file.- Similar to `printf()` but for files.

Example 1. The following program opens a file for writing, writes two strings and then closes the file:

[1]:
```
%%writefile file-open.c

#include <stdio.h>

int main() {
  FILE *fptr;

  // Open the file in write mode
  fptr = fopen("myfile.txt", "w");
  if (fptr == NULL) {
    printf("Error opening file.");
    return -1;  // Exit the program if the file cannot be opened
  }

  // Write to the file
  fprintf(fptr, "Hello, this is written to the file!\n");
  fputs("This is another line written using fputs.\n", fptr);

  // Close the file
  fclose(fptr);

  printf("Data written to the file successfully.\n");
  return 0;
}
```

```
Overwriting file-open.c
```

[2]:
```
%%script bash
gcc file-open.c -o file-open
./file-open
```

```
Data written to the file successfully.
```

In this program, if there is an error when opening the file, a `-1` error code is returned to the system. Error handling will be explained later.

Closing a file when you are done using it is a good programming practice.

Example 2. Writing numerical data to a file.

This example writes an array of floating-point numbers to a file. This could represent, for example, a time series of experimental measurements.

[3]:
```
%%writefile file-open2.c

#include <stdio.h>

int main() {
    // Sample data: time series of measurements
    double data[] = {1.2, 2.3, 3.4, 4.5, 5.6};
    int num_points = sizeof(data) / sizeof(data[0]);

    // Open a file for writing
    FILE *file = fopen("data.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return -1;
    }

    // Write data to the file
    for (int i = 0; i < num_points; i++) {
        fprintf(file, "%.6f\n", data[i]);  // Write each value with 6 decimal␣
 ↪places
    }

    // Close the file
    fclose(file);

    printf("Data written to file successfully.\n");
    return 0;
}
```

```
Overwriting file-open2.c
```

[4]:
```
%%script bash
gcc file-open2.c -o file-open2
./file-open2
```

```
Data written to file successfully.
```

4 Reading from a File

The `stdio.h` library also includes functions for reading from an open file. A file can be read one character at a time or an entire string can be read into a character buffer, which is typically a char array used for temporary storage.

To read from a file you can use functions like `fscanf()`, `fgets()`, or `fread()`. Below is an example program that reads the contents of a file (`myfile.txt`) and prints it to the console.

Function	Description	Usage Example	Key Differences
`fscanf()`	Reads formatted input from a file.	`fscanf(file, "%d %s", &num, str);`	- Reads formatted data (e.g., integers, strings).- Stops at whitespace or specified delimiter.
`fgets()`	Reads a string from a file until a newline or EOF is encountered.	`fgets(buffer, sizeof(buffer), file);`	- Reads a line of text.- Includes the newline character in the buffer.- Stops at newline or EOF.
`fread()`	Reads raw data from a file into a buffer.	`fread(buffer, sizeof(char), size, file);`	- Reads binary or raw data.- Does not interpret or format the data.- Reads a specified number of bytes.

Example 1. Reading text files line by line.

[5]:
```
%%writefile file-read.c

#include <stdio.h>

int main() {
    FILE *fptr;
    char buffer[100]; // Buffer to store the data read from the file

    // Open the file in read mode
    fptr = fopen("myfile.txt", "r");
    if (fptr == NULL) {
        printf("Error opening file.");
        return -1; // Exit the program if the file cannot be opened
    }

    // Read and print the file contents line by line
    while (fgets(buffer, sizeof(buffer), fptr) != NULL) {
        printf("%s", buffer); // Print the line to the console
    }

    // Close the file
```

```
    fclose(fptr);

    return 0;
}
```

```
Overwriting file-read.c
```

`fopen("myfile.txt", "r")`opens the file `myfile.txt` in **read mode**.

`fgets(buffer, sizeof(buffer), fptr)` reads a line from the file into the `buffer`:

- The `sizeof(buffer)` ensures that no more than `100` characters are read at a time (prevents buffer overflow).
- Returns `NULL` when the end of the file is reached.

[6]:
```
%%script bash
gcc file-read.c -o file-read
./file-read
```

```
Hello, this is written to the file!
This is another line written using fputs.
```

Example 2. Reading numerical data from a file

This example reads the data from the file stores it in an array.

[7]:
```
%%writefile file-read2.c

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

int main() {
    // Open the file for reading
    FILE *file = fopen("data.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    // Determine the number of data points
    int num_points = 0;
    double value;
    while (fscanf(file, "%lf", &value) == 1) {
        num_points++;
    }

    // Allocate memory for the data array
    double *data = (double *)malloc(num_points * sizeof(double));
    if (data == NULL) {
        printf("Memory allocation failed!\n");
```

```
        fclose(file);
        return 1;
    }

    // Rewind the file to read the data again
    rewind(file);

    // Read the data into the array
    for (int i = 0; i < num_points; i++) {
        fscanf(file, "%lf", &data[i]);
    }

    // Close the file
    fclose(file);

    // Print the data to verify
    printf("Data read from file:\n");
    for (int i = 0; i < num_points; i++) {
        printf("%.6f\n", data[i]);
    }

    // Free allocated memory
    free(data);

    return 0;
}
```

```
Overwriting file-read2.c
```

[8]:
```
%%script bash
gcc file-read2.c -o file-read2
./file-read2
```

```
Data read from file:
1.200000
2.300000
3.400000
4.500000
5.600000
```

Navigating within files efficiently is a common task in C programming, especially when dealing with large files. The functions `fseek`, `ftell`, and `rewind` are part of the C Standard Library and are used to manipulate the file position indicator within a file stream.

5 Special files

Special files are kernel objects that are represented as files. Over the years, Unix systems have supported a handful of different special files. Linux supports four: block device files, character

device files, named pipes, and Unix domain sockets. Special files are a way to let certain abstractions fit into the filesystem, continuing the everything-is-a-file paradigm. Linux provides a system call to create a special file.

Named pipes (often called *FIFOs*, short for "first in, first out") are an *interprocess communication* (IPC) mechanism that provides a communication channel over a file descriptor, accessed via a special file. Regular pipes or unnamed pipes (often simply called "pipes") are the method used to "pipe" the output of one program into the input of another; they are created in memory via a system call and do not exist on any filesystem. Named pipes act like regular pipes but are accessed via a file, called a *FIFO special file*. Unrelated processes can access this file and communicate.

6 Pipes

Pipes are used to allow one or more processes to have information "flow" between them. The most common example of this is with the shell [source].

[9]:
```
! ls | wc -l
```

```
13
```

The command `ls | wc -l` is a classic example of using pipes in a Unix-like shell (e.g., Bash).

As we've seen the std-out from the left side (`ls`) is connected to the std-in on the right side (`wc -l`). As far the each program is concerned, it is reading or writing as it normally does. Both processes are running concurrently.

The `ls` command writes its output (the list of files and directories) to the **write end** of the pipe. The `wc -l` command reads from the **read end** of the pipe and counts the number of lines. The final result is the number of files and directories in the current directory.

There are 2 types of pipes: * unnamed pipes * named pipes

Feature	Unnamed Pipes	Named Pipes (FIFOs)
Creation	Created using `pipe()`	Created using `mkfifo()` or `mknod()`
Persistence	Temporary; exist only during process lifetime	Persistent; exist until explicitly deleted
Filesystem Entry	No filesystem entry	Appears as a file in the filesystem
Communication	Unidirectional	Unidirectional (typically)
Process Relationship	Used between related processes (e.g., parent and child)	Used between unrelated processes
Blocking Behavior	Blocks if the pipe is empty (read) or full (write)	Blocks until both reader and writer are present
Bidirectional Communication	Requires two pipes	Can be used bidirectionally (with care)
Example Use Case	Communication between parent and child processes	Communication between unrelated processes or across different sessions

Feature	Unnamed Pipes	Named Pipes (FIFOs)
File Descriptors	Created with `pipe(fd)` where `fd[0]` is read end and `fd[1]` is write end	Accessed using `open()` with a pathname
Deletion	Automatically destroyed when processes terminate	Must be explicitly deleted using `unlink()`
Example Code	`int fd[2]; pipe(fd);`	`mkfifo("/tmp/myfifo", 0666);`

The examples we seen at the shell command line are unnamed. They are created, used and destroyed within the life a set of processes. Each end of the pipe has it's own file descriptor. One end is for reading and one end is for writing. When you are done with a pipe, it is closed like any other file.

Unnamed pipes (often simply called "pipes") are used for communication between related processes, typically between a parent process and its child process.

A *pipe* is a system call that creates an **unidirectional** communication link between two file descriptors. The pipe system call is called with a pointer to an array of two integers. Upon return, the first element of the array contains the file descriptor that corresponds to the output of the pipe (stuff to be read). The second element of the array contains the file descriptor that corresponds to the input of the pipe (the place where you write stuff). Whatever bytes are sent into the input of the pipe can be read from the other end of the pipe.

6.1 Example

This is a small program demonstrates the use of **unnamed pipes** for *inter-process communication* (IPC) between a **parent process** and its **child process**.

[10]:
```
%%writefile piles.c

#include <stdio.h>
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);

    if (fork() == 0) {  // Child process
        close(fd[0]);  // Close read end
        write(fd[1], "Hello, C", 9);
        close(fd[1]);
    } else {  // Parent process
        close(fd[1]);  // Close write end
        char buffer[9];
        read(fd[0], buffer, 9);
        printf("Received: %s\n", buffer);
        close(fd[0]);
    }
```

```
    return 0;
}
```

```
Overwriting piles.c
```

[11]:
```
%%script bash
gcc piles.c -o piles
./piles
```

```
Received: Hello, C
```

`<unistd.h>` provides functions like `pipe()`, `fork()`, `read()`, and `write()`.

`pipe(fd)` creates an unnamed pipe.

`fd` is an array of two integers:

- `fd[0]` is a file descriptor for the **read end** of the pipe.
- `fd[1]` is a file descriptor for the **write end** of the pipe.

`fork()` creates a child process. The return value of `fork()` is:

- `0` in the child process.
- A positive value (child's PID) in the parent process.

```
close(fd[0]);  // Close read end
write(fd[1], "Hello, C", 9);
close(fd[1]);
```

The child process closes the **read end** of the pipe (`fd[0]`) because it only needs to write. Writes the string "Hello, C" (9 bytes, including the null terminator) to the write end of the pipe (`fd[1]`) and closes the write end of the pipe after writing.

Pipes are a fundamental mechanism for inter-process communication (IPC) in Unix-like operating systems (e.g., Linux, macOS). They are important because they enable processes to communicate and share data efficiently, which is essential for building complex, modular, and scalable systems.