Input-output system calls in C | Create, Open, Close, Read, Write

System calls are the calls that a program makes to the system kernel to provide the services to which the program does not have direct access. For example, providing access to input and output devices such as monitors and keyboards. We can use various functions provided in the C Programming language for input/output system calls such as create, open, read, write, etc.

Before we move on to the I/O System Calls, we need to know about a few important terms.

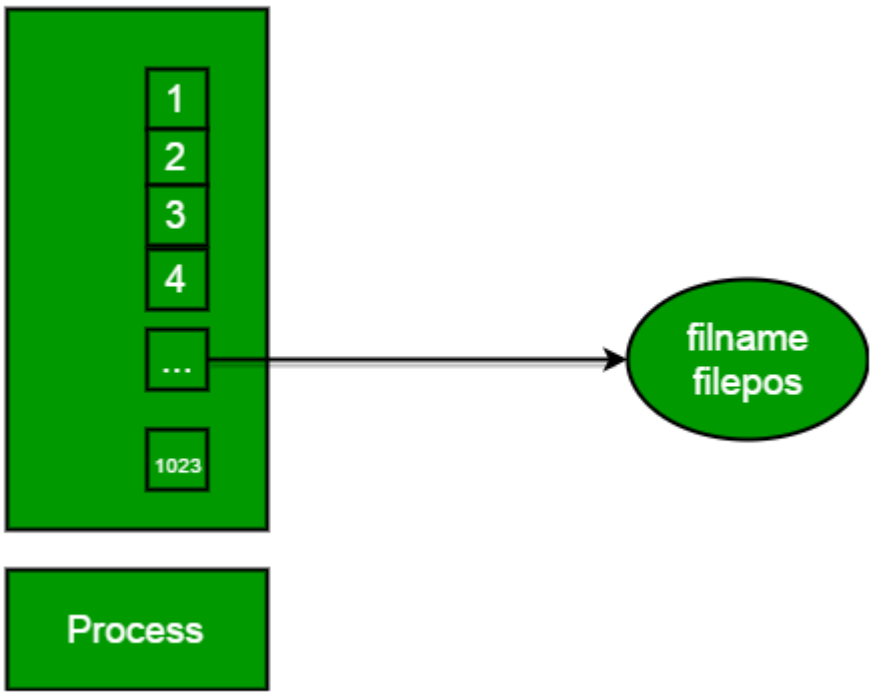
Important Terminology

What is the File Descriptor?

The file descriptor is an integer that uniquely identifies an open file of the process.

File Descriptor table: A file descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in the operating system for each process.

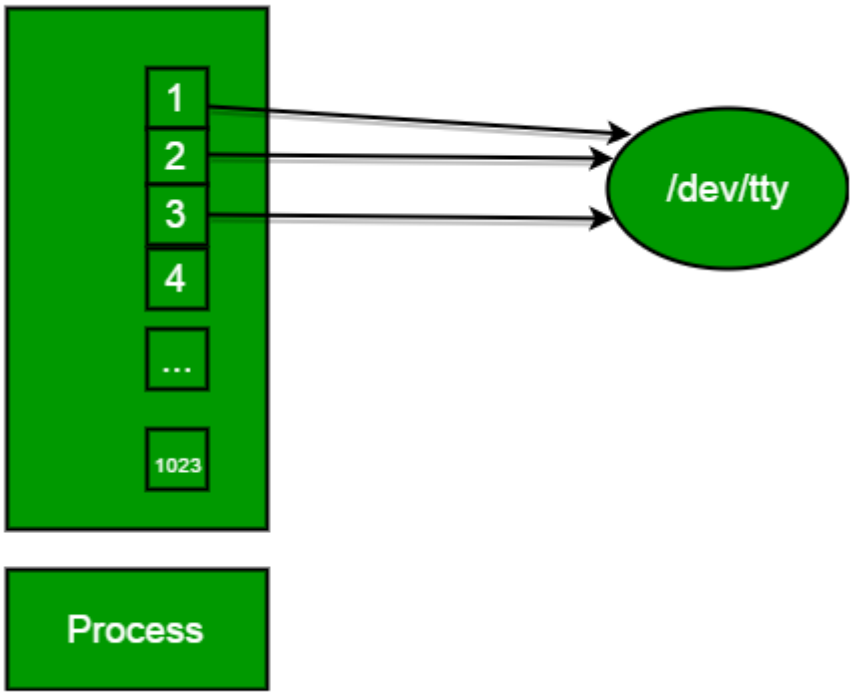
File Table Entry: File table entries are a structure In-memory surrogate for an open file, which is created when processing a request to open the file and these entries maintain file position.



Standard File Descriptors: When any process starts, then that process file descriptors table's fd(file descriptor) 0, 1, 2 open automatically, (By default) each of these 3 fd references file table entry for a file named **/dev/tty**

/dev/tty: In-memory surrogate for the terminal.

Terminal: Combination keyboard/video screen.



Read from stdin => read from fd 0: Whenever we write any character from the keyboard, it reads from stdin through fd 0 and saves to a file named /dev/tty.
Write to stdout => write to fd 1: Whenever we see any output to the video screen, it's from the file named /dev/tty and written to stdout in screen through fd 1.
Write to stderr => write to fd 2: We see any error to the video screen, it is also from that file write to stderr in screen through fd 2.

Input/Output System Calls

Basically, there are total 5 types of I/O system calls:

1. C create

The create() function is used to create a new empty file in C. We can specify the permission and the name of the file which we want to create using the create() function. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of create() in C

```
int create(char *filename, mode_t mode);
```

Parameter

- **filename:** name of the file which you want to create
- **mode:** indicates permissions of the new file.

Return Value

- return first unused file descriptor (generally 3 when first creating use in the process because 0, 1, 2 fd are reserved)
- return -1 when an error

How C create() works in OS

- Create a new empty file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

2. C open

The open() function in C is used to open the file for reading, writing, or both. It is also capable of creating the file if it does not exist. It is defined inside **<unistd.h>** header file and the flags that are passed as arguments are defined inside **<fcntl.h>** header file.

Syntax of open() in C

```
int open (const char* Path, int flags);
```

Parameters

- **Path:** Path to the file which we want to open.
    - Use the **absolute path** beginning with "/" when you are **not working in the same directory** as the C source file.
    - Use **relative path** which is only the file name with extension, when you are **working in the same directory** as the C source file.
- **flags:** It is used to specify how you want to open the file. We can use the following flags.

Flags	Description
O_RDONLY	Opens the file in read-only mode.
O_WRONLY	Opens the file in write-only mode.
O_RDWR	Opens the file in read and write mode.
O_CREAT	Create a file if it doesn't exist.
O_EXCL	Prevent creation if it already exists.
O_ APPEND	Opens the file and places the cursor at the end of the contents.
O_ASYNC	Enable input and output control by signal.
O_CLOEXEC	Enable close-on-exec mode on the open file.
O_NONBLOCK	Disables blocking of the file opened.
O_TMPFILE	Create an unnamed temporary file at the specified path.

How C open() works in OS

- Find the existing file on the disk.
- Create file table entry.
- Set the first unused file descriptor to point to the file table entry.
- Return file descriptor used, -1 upon failure.

Example of C open()

```
// C program to illustrate
// open system call
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

extern int errno;

int main()
{
    // if file does not have in directory
    // then file foo.txt is created.
    int fd = open("foo.txt", O_RDONLY | O_CREAT);

    printf("fd = %d\n", fd);

    if (fd == -1) {
        // print which type of error have in a code
        printf("Error Number % d\n", errno);

        // print program detail "Success or failure"
        perror("Program");
    }
```

```
    return 0;
}
```

Output

```
fd = 3
```

3. C close

The close() function in C tells the operating system that you are done with a file descriptor and closes the file pointed by the file descriptor. It is defined inside **<unistd.h>** header file.

Syntax of close() in C

```
int close(int fd);
```

Parameter

- **fd: F**ile descriptor of the file that you want to close.

Return Value

- **0** on success.
- **-1** on error.

How C close() works in the OS

- Destroy file table entry referenced by element fd of the file descriptor table – As long as no other process is pointing to it!
- Set element fd of file descriptor table to **NULL**

Example 1: close() in C

```
// C program to illustrate close system Call
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int fd1 = open("foo.txt", O_RDONLY);
    if (fd1 < 0) {
        perror("c1");
        exit(1);
    }
    printf("opened the fd = % d\n", fd1);

    // Using close system Call
    if (close(fd1) < 0) {
        perror("c1");
```

```
        exit(1);
    }
    printf("closed the fd.\n");
}
```

Output

```
opened the fd = 3
closed the fd.
```

Example 2:

```
// C program to illustrate close system Call
#include<stdio.h>
#include<fcntl.h>
int main()
{
    // assume that foo.txt is already created
    int fd1 = open("foo.txt", O_RDONLY, 0);
    close(fd1);

    // assume that baz.tzt is already created
    int fd2 = open("baz.txt", O_RDONLY, 0);

    printf("fd2 = % d\n", fd2);
    exit(0);
}
```

Output

```
fd2 = 3
```

Here, In this code first open() returns **3** because when the main process is created, then fd **0, 1, 2** are already taken by **stdin**, **stdout,** and **stderr**. So the first unused file descriptor is **3** in the file descriptor table. After that in close() system call is free it these **3** file descriptors and then set **3** file descriptors as **null**. So when we called the second open(), then the first unused fd is also **3**. So, the output of this program is **3**.

4. C read

From the file indicated by the file descriptor fd, the read() function reads the specified amount of bytes **cnt** of input into the memory area indicated by **buf**. A successful read() updates the access time for the file. The read() function is also defined inside the <unistd.h> header file.

Syntax of read() in C

```
size_t read (int fd, void* buf, size_t cnt);
```

Parameters

- **fd:** file descriptor of the file from which data is to be read.

- **buf:** buffer to read data from
- **cnt:** length of the buffer

Return Value

- return Number of bytes read on success
- return 0 on reaching the end of file
- return -1 on error
- return -1 on signal interrupt

Important Points

- **buf** needs to point to a valid memory location with a length not smaller than the specified size because of overflow.
- **fd** should be a valid file descriptor returned from open() to perform the read operation because if fd is NULL then the read should generate an error.
- **cnt** is the requested number of bytes read, while the return value is the actual number of bytes read. Also, some times read system call should read fewer bytes than cnt.

Example of read() in C

```
// C program to illustrate
// read system Call
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int fd, sz;
    char* c = (char*)calloc(100, sizeof(char));

    fd = open("foo.txt", O_RDONLY);
    if (fd < 0) {
        perror("r1");
        exit(1);
    }

    sz = read(fd, c, 10);
    printf("called read(% d, c, 10). returned that"
           " %d bytes were read.\n",
           fd, sz);
    c[sz] = '\0';
    printf("Those bytes are as follows: % s\n", c);

    return 0;
}
```

Output

```
called read(3, c, 10).  returned that 10 bytes  were read.
Those bytes are as follows: 0 0 0 foo.
```

Suppose that foobar.txt consists of the 6 ASCII characters "foobar". Then what is the output of the following program?

```
// C program to illustrate
// read system Call
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    char c;
    int fd1 = open("sample.txt", O_RDONLY, 0);
    int fd2 = open("sample.txt", O_RDONLY, 0);
    read(fd1, &c, 1);
    read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Output

```
c = f
```

The descriptors ***fd1*** and ***fd2*** each have their own open file table entry, so each descriptor has its own file position for ***foobar.txt***. Thus, the read from ***fd2*** reads the first byte of ***foobar.txt***, and the output is **c = f**, not **c = o**.

5. C write

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

The write() is also defined inside **<unistd.h>** header file.

Syntax of write() in C

```
size_t write (int fd, void* buf, size_t cnt);
```

Parameters

- **fd:** file descriptor
- **buf:** buffer to write data from.
- **cnt:** length of the buffer.

Return Value

- returns the number of bytes written on success.
- return 0 on reaching the End of File.
- return -1 on error.
- return -1 on signal interrupts.

Important Points about C write

- The file needs to be opened for write operations
- **buf** needs to be at least as long as specified by cnt because if buf size is less than the cnt then buf will lead to the overflow condition.
- **cnt** is the requested number of bytes to write, while the return value is the actual number of bytes written. This happens when **fd** has a less number of bytes to write than cnt.
- If write() is interrupted by a signal, the effect is one of the following:
    - If write() has not written any data yet, it returns -1 and sets errno to EINTR.
    - If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Example of write() in C

```
// C program to illustrate
// write system Call
#include<stdio.h>
#include <fcntl.h>
main()
{
int sz;

int fd = open("foo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd < 0)
{
    perror("r1");
    exit(1);
}

sz = write(fd, "hello geeks\n", strlen("hello geeks\n"));

printf("called write(% d, \"hello geeks\\n\", %d)."
    " It returned %d\n", fd, strlen("hello geeks\n"), sz);

close(fd);
}
```

Output

```
called write(3, "hello geeks\n", 12).  it returned 11
```

Here, when you see in the file foo.txt after running the code, you get a "*hello geeks*". If foo.txt file already has some content in it then the write a system calls overwrite the content and all previous content is ***deleted*** and only "*hello geeks*" content will have in the file.

Example: Print "hello world" from the program without using printf function.

```
// C program to illustrate
// I/O system Calls
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(void)
{
    int fd[2];
    char buf1[12] = "hello world";
    char buf2[12];

    // assume foobar.txt is already created
    fd[0] = open("foobar.txt", O_RDWR);
    fd[1] = open("foobar.txt", O_RDWR);

    write(fd[0], buf1, strlen(buf1));
    write(1, buf2, read(fd[1], buf2, 12));

    close(fd[0]);
    close(fd[1]);

    return 0;
}
```

Output

```
hello world
```

In this code, buf1 array's string ***"hello world"*** is first written into stdin fd[0] then after that this string write into stdin to buf2 array. After that write into buf2 array to the stdout and print output "***hello world***"