C++

#include <iostream>

#include <fcntl.h>

#include <unistd.h>

using namespace std;

int main() {

const char \*filename = "my\_file.txt";

char buffer[100];

// Writing to a file

int fd = open(filename, O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd == -1) {

cerr << "Error opening file for writing!" << endl;

return 1;

}

cout << "Enter a string to write to the file: ";

cin.getline(buffer, 100);

write(fd, buffer, strlen(buffer));

close(fd);

// Reading from the file and printing to the screen

fd = open(filename, O\_RDONLY);

if (fd == -1) {

cerr << "Error opening file for reading!" << endl;

return 1;

}

cout << "Contents of the file:\n";

int bytesRead;

while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {

write(STDOUT\_FILENO, buffer, bytesRead);

}

close(fd);

return 0;

}

Use code with caution.

Explanation:

Include necessary headers: iostream for input/output, fcntl.h for file control flags, and unistd.h for system calls.

Declare variables: filename for the file name, buffer for storing data, and fd for the file descriptor.

Open the file for writing: open with O\_WRONLY, O\_CREAT, and O\_TRUNC flags to create if not exists, truncate if exists, and open for writing.

Error handling: Check if fd is -1, indicating an error.

Get user input: Read a string from the user using cin.getline.

Write to the file: Use write to write the string to the file.

Close the file: Use close to close the file.

Open the file for reading: open with O\_RDONLY flag.

Error handling: Check for errors as before.

Read from the file and print: Use read to read data into the buffer, and write to print it to the screen (using STDOUT\_FILENO).

Close the file: Close the file.

Key Points:

System calls provide a lower-level interface to file operations.

Error handling is crucial when using system calls.

open flags control file creation and access modes.

read and write operate on file descriptors and byte counts.

STDOUT\_FILENO represents the standard output file descriptor.

This code demonstrates the use of system calls to perform file operations in C++, providing a more direct interaction with the operating system.

System API: **open**

The open system call is a fundamental function in Unix-like operating systems used to open a file or device for reading, writing, or appending. It returns a file descriptor, a non-negative integer used to reference the open file in subsequent system calls.

#include <fcntl.h>

int open(const char pathname, int flags, ... /\* mode\_t mode \*/);

Parameters

pathname: A null-terminated character string specifying the path to the file or device to be opened.

flags: An integer specifying the flags that control how the file is opened.

mode: An optional mode\_t argument used only if the file is created (when the O\_CREAT flag is set). It specifies the permissions for the new file.

Flags

The flags argument is a bitwise OR of one or more of the following flags:

O\_RDONLY: Open for reading only.

O\_WRONLY: Open for writing only.

O\_RDWR: Open for both reading and writing.

O\_CREAT: Create the file if it does not exist.

O\_EXCL: Used with O\_CREAT, fails if the file already exists.

O\_TRUNC: Truncate the file to zero length if it exists.

O\_APPEND: Append writes to the end of the file.

O\_NONBLOCK: Non-blocking open (for devices).

O\_SYNC: Synchronize file writes (slower).

O\_DSYNC: Synchronize data writes (faster than O\_SYNC).

System API: **close**

The close call closes a file descriptor, so that it no longer refers to any file and may be reused.

#include<unistd.h>

Int close(int fd);

If fd is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed; if the file descriptor was the last reference to a file which has been removed using, the file is deleted.

System API: **read**

Synopsis :

#include<unistd.h>

ssize\_t read(int fd, void buf[.count], size\_t count);

read() attempts to read up to *count* bytes from file descriptor fd into the buffer starting at buf.

On files that support seeking, the read operation commences at file offset, and the file offset is incremented by the number of bytes read. If the file offset is at or past the end of file, no bytes are read, and read() returns zero.

If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read with a count of 0 returns zero and has no other effects.

System API: **write**

Synopsis :

#include<unistd.h>

ssize\_t write(int fd, const void buf[.count], size\_t count);

write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.

writing takes place at the file offset, and the file offset is incremented by the number of bytes actually written. If the file was opened with O\_APPEND, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

System API: **socket**

Synopsis:

#include <sys/socket.h>

int socket(int *domain*, int *type*, int *protocol*);

**socket**() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

The domain argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in <sys/socket.h>. The formats currently understood by the Linux kernel include:

**AF\_UNIX** Local communication

**AF\_LOCAL** Synonym for **AF\_UNIX**

**AF\_INET** IPv4 Internet protocols

**AF\_AX25** Amateur radio AX.25 protocol

**AF\_IPX** IPX - Novell protocols

The socket has the indicated *type*, which specifies the communication semantics. Currently defined types are:

**SOCK\_STREAM**

Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanismmay be supported.

**SOCK\_DGRAM**

Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

**SOCK\_SEQPACKET**

Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each input system call.

**SOCK\_RAW**

Provides raw network protocol access.

**SOCK\_RDM**

Provides a reliable datagram layer that does not guarantee ordering.

System API: **bind**

Synopsis

#include<sys/socket.h>

Int bind (int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

When a socket is created with socket() , it exists in a name space but has no address to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, o the address structure pointed to by addr.

System API: **connect**

Synopsis:

#include<sys/socket.h>

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t *addrlen*);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd.

If the socket sockfd is of type **SOCK\_DGRAM**, then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type **SOCK\_STREAM** or **SOCK\_SEQPACKET**, this call attempts to make a connection to the socket that is bound to the address specified by addr.