File Handling Solutions in C/C++ for Linux

Example Solutions and Q&A

1 File Descriptors and Streams

In Linux, there are three default file streams:

- **Standard Input** (file descriptor: 0)
- **Standard Output** (file descriptor: 1)
- **Standard Error** (file descriptor: 2)

Other files that a process opens will be assigned file descriptors starting from 3 onward.

2 Opening a File

To open a file, use the `open()` system call. Here is an example that opens a file for writing:

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

int main() {
    int fd = open("example.txt", O_WRONLY | O_CREAT, 0666);
    if (fd == -1) {
        printf("Error opening file.\n");
        return 1;
    }
    printf("File opened successfully. File descriptor: %d\n", fd);
    close(fd);
    return 0;
}
```

In this example:

- `O_WRONLY` specifies the file is open for writing.

- `O_CREAT` creates the file if it does not exist.
- `0666` sets the file permissions to allow read and write for all users.

3 Closing a File

Files are closed using the `close()` function:

```
close(fd);
```

This releases the file descriptor and makes it available for other files.

4 Writing to a File

Use the `write()` function to write data to a file. Here is an example:

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

int main() {
    int fd = open("example.txt", O_WRONLY | O_CREAT | O_APPEND, 0666);
    if (fd == -1) {
        printf("Error opening file.\n");
        return 1;
    }
    const char *text = "Hello, World!\n";
    write(fd, text, 13); // Write 13 bytes to the file
    close(fd);
    printf("Data written to file.\n");
    return 0;
}
```

Explanation:

- The text "Hello, World!" is written to the file.
- The file is opened with the `O_APPEND` flag to append data.
- `0666` sets file permissions to allow read and write for all users.

5 Reading from a File

Here is an example to read data from a file using the `read()` system call:

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

```c
int main() {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        printf("Error opening file.\n");
        return 1;
    }
    char buffer[100];
    int bytesRead = read(fd, buffer, sizeof(buffer)-1);
    if (bytesRead == -1) {
        printf("Error reading file.\n");
        close(fd);
        return 1;
    }
    buffer[bytesRead] = '\0'; // Null-terminate the string
    printf("Contents of file:\n%s\n", buffer);
    close(fd);
    return 0;
}
```

Explanation:

- The file is opened in read-only mode using `O_RDONLY`.
- Data is read into a buffer, and `buffer[bytesRead] = '\0'` ensures it is null-terminated.
- The program prints the file contents to the standard output.

6 Q&A

Q1: What is `0666` that is specified in the `open()` call? What does it mean?

The value `0666` specifies the file permissions when a file is created using the `open()` system call. In Unix-like operating systems, file permissions are set using a three-digit octal number, where each digit represents different permission sets:

- The first digit (0) represents special permissions (not used here).
- The second digit represents the owner's permissions.
- The third digit represents the group's permissions.
- The fourth digit represents permissions for others.

The permissions are set as follows:

- `6` (read and write): The owner can read and write the file.

- `6` (read and write): Members of the file's group can read and write the file.
- `6` (read and write): All other users can read and write the file.

In this case, `0666` allows read and write permissions for all users, but no execute permissions.

Q2: What is `O_APPEND` doing in the same call? Run the program again and check its output.

The `O_APPEND` flag used in the `open()` system call specifies that all writes to the file should be appended to the end of the file, rather than overwriting existing content. When this flag is set, the file pointer is moved to the end of the file before each write operation, ensuring that new data is added without altering the existing content.

Running the program with `O_APPEND` will append the new data to the end of the file, so if the program is executed multiple times, the file's contents will include repeated entries of the written data.

Q3: Modify the following line in the code and then compile and run the program. What has happened?

From:

```
size_t length = strlen(timeStamp);
```

To:

```
size_t length = strlen(timeStamp) - 5;
```

By changing `strlen(timeStamp)` to `strlen(timeStamp) - 5`, the number of bytes written to the file is reduced by 5. This means that the last 5 characters of the `timeStamp` string will not be written to the file. If the original `timeStamp` includes a newline character at the end (as is common with time-related strings), this modification may result in a truncated or incomplete line of output in the file.