Lab 05: UNIX I/O

UNIX uses a uniform device interface, through file descriptors, that allows the same I/O calls to be used for terminals, disks, tapes, audio and even network communication. This chapter explores the five functions that form the basis for UNIX device independent I/O. The chapter also examines I/O from multiple sources, blocking I/O with timeouts, inheritance of file descriptors and redirection. The code carefully handles errors and interruption by signals.

UNIX provides sequential access to files and other devices through the `read` and `write` functions. The `read` function attempts to retrieve `nbyte` bytes from the file or device represented by `fildes` into the user variable `buf`. You must provide a buffer that is large enough to hold `nbyte` bytes of data. (A common mistake is to provide an uninitialized pointer, `buf`, rather than an actual buffer.)

```
SYNOPSIS

   #include <unistd.h>

   ssize_t read(int fildes, void *buf, size_t nbyte);

POSIX
```

If successful, `read` returns the number of bytes read. If unsuccessful, `read` returns –1 and sets `errno`.

A `read` operation for a regular file may return fewer bytes than requested if, for example, it reached end-of-file before completely satisfying the request. A `read` operation for a regular file returns 0 to indicate end-of-file. When special files corresponding to devices are read, the meaning of a `read` return value of 0 depends on the implementation and the particular device. A `read` operation for a pipe returns as soon as the pipe is not empty, so the number of bytes read can be less than the number of bytes requested. When reading from a terminal, `read` returns 0 when the user enters an end-of-file character. On many systems the default end-of-file character is Ctrl-D.

The `ssize_t` data type is a signed integer data type used for the number of bytes read, or –1 if an error occurs. On some systems, this type may be larger than an `int`. The `size_t` is an unsigned integer data type for the number of bytes to read.

The file descriptor, which represents a file or device that is open, can be thought of as an index into the process file descriptor table. The file descriptor table is in the process user area and provides access to the system information for the associated file or device.

When you execute a program from the shell, the program starts with three open streams associated with file descriptors `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`. `STDIN_FILENO` and `STDOUT_FILENO` are standard input and standard output, respectively. By default, these two streams usually correspond to keyboard input and screen output.

Programs should use STDERR_FILENO, the standard error device, for error messages and should never close it. In legacy code standard input, standard output and standard error are represented by 0, 1 and 2, respectively. However, you should always use their symbolic names rather than these numeric values.

The write function attempts to output nbyte bytes from the user buffer buf to the file represented by file descriptor fildes.

```
SYNOPSIS

  #include <unistd.h>

  ssize_t write(int fildes, const void *buf, size_t nbyte);

POSIX
```

If successful, write returns the number of bytes actually written. If unsuccessful, write returns –1 and sets errno.

The open function associates a file descriptor (the handle used in the program) with a file or physical device. The path parameter of open points to the pathname of the file or device, and the oflag parameter specifies status flags and access modes for the opened file. You must include a third parameter to specify access permissions if you are creating a file.

```
SYNOPSIS

   #include <fcntl.h>
   #include <sys/stat.h>

   int open(const char *path, int oflag, ...);
                                                             POSIX
```

If successful, open returns a nonnegative integer representing the open file descriptor. If unsuccessful, open returns –1 and sets errno.

Construct the oflag argument by taking the bitwise OR (|) of the desired combination of the access mode and the additional flags. The POSIX values for the access mode flags are O_RDONLY, O_WRONLY and O_RDWR. You must specify exactly one of these designating read-only, write-only or read-write access, respectively.

The additional flags include O_APPEND, O_CREAT, O_EXCL, O_NONBLOCK and O_TRUNC. The O_APPEND flag causes the file offset to be moved to the end of the file before a write, allowing you to add to an existing file. In contrast, O_TRUNC truncates the length of a regular file opened for writing to 0. The O_CREAT flag causes a file to be created if it doesn't already exist. If you include the O_CREAT flag, you must also pass a third argument to open to designate the permissions. If you want to avoid writing over an existing file, use the combination `O_CREAT | O_EXCL`. This combination returns an error if the file already exists.

The `close` function has a single parameter, `fildes`, representing the open file whose resources are to be released.

```
SYNOPSIS

   #include <unistd.h>

   int close(int fildes);
                                         POSIX
```

If successful, `close` returns 0. If unsuccessful, `close` returns –1 and sets `errno`

Tasks

Task 1: Implement the *cp* command.

NAME
cp - copy files and directories

SYNOPSIS
cp [OPTION]... SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
cp [OPTION]... --target-directory=DIRECTORY SOURCE...

DESCRIPTION
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.
Mandatory arguments to long options are mandatory for short options too.

Task 2: Implement *rm* command.

Task 3: Implement the *mov* command.

NAME
mv - move (rename) files

SYNOPSIS
mv [OPTION]... SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
mv [OPTION]... --target-directory=DIRECTORY SOURCE...

DESCRIPTION
Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.
Mandatory arguments to long options are mandatory for short options too.