System Programming in C for Linux (I)

File management, part I:

I/O system calls for working with files

Cristian Vidrașcu

cristian.vidrascu@info.uaic.ro

April, 2024

Overview

Introduction

The functions that you can call in the C programs that you write, to access and process files (both regular files and directories or other file types) fall into two categories:

- The POSIX API, which provides wrapper functions for the system calls provided by the `Linux` kernel ; these functions can be called from C programs that will be compiled for the `Linux` platform and, more generally, for any `UNIX` operating system that implements the POSIX standard.
  - Advantage: the functions from this API provide access to virtually all `Linux` kernel features "exported" to user-mode.
  - Disadvantage: programs that use these functions are not portable, *e.g.* cannot be compiled for the `Windows` platform (at least not directly, but only in the WINDOWS SUBSYTEM FOR LINUX environment, introduced in `Windows 10`).
- C STANDARD LIBRARY, which offers a number of higher-level features, including for working with files; these functions can be called from C programs that will be compiled for any platform that offers a C compiler, plus an implementation of the standard C library. For example, for `Linux` platform the most used is the GCC (*the GNU Compiler Collection*) compiler and the GLIBC (the GNU libc) implementation of the standard C library.
  - Advantage: allows you to write portable programs between various platforms (*e.g.*, `Windows`, `UNIX`/`Linux`, etc.).
  - Disadvantage: contains features with limited ability to manage operating system resources (*e.g.*, files), which is why it is suitable for writing simple programs.

Outline

The main categories of I/O system calls

The UNIX / Linux file management system provides the following categories of system calls, according to the POSIX standard:

- system calls for creation of new files of various types: mknod, mkfifo, mkdir, link, symlink, creat, socket
- system calls for file deletion: rmdir (for directories), unlink (for all other file types)
- system call for file renaming, of any type: rename
- system calls for query the i-node of a file: stat / fstat / lstat, access
- system calls for manipulating the i-node of a file: chmod / fchmod, chown / fchown / lchown
- system calls for file system extensions: mount, umount
- system calls for accessing and manipulating the contents of a file, through a *working session*: open / creat, read, write, lseek, close, fcntl, etc.
- system calls for duplication, in a process, of a *working session* with a file: dup, dup2

The main categories of I/O system calls (cont.)

- system calls for consulting the "status" of some *working sessions* with files (multiplexed synchronous I/O operations): `select`, `poll`
- system call for "truncating" the contents of a file: `truncate` / `ftruncate`
- system calls for modifying attributes in a process:
  - `chdir` : changes the current working directory
  - `umask` : changes the "mask" of the default permissions when creating a file
  - `chroot` : modifies the root of the file system accessible to the process
- system calls for exclusive access to files: `flock`, `fcntl`
- system call for the "mapping" of a file in the memory of a process: `mmap`
- system call for the creation of an anonymous communication channel: `pipe`
- etc.

Remark: in case of an error, all these primitives return the value `-1`, as well as an error number that is stored in the global variable `errno` (defined in the header file `<errno.h>`), error that can be diagnosed with the function `perror()`.

6 / 26

The access system call

- *Checking access permissions to a file*: the `access` system call.
  The interface of the function `access` ([5]):
  `int access(char* pathname, int mode)`
  - *pathname* = the name of the file to check
  - *mode* = the access permission being checked, which can be a combination (*i.e.*, logical disjunction on bits) of the following symbolic constants:
    - `X_OK` (=1) : does the calling process have the right to execute the file ?
    - `W_OK` (=2) : does the calling process have the right to write the file ?
    - `R_OK` (=4) : does the calling process have the right to read the file ?

    Remarks: i) for *mode* = `F_OK` (=0) only the existence of the file is checked ; ii) the other rights, if checked, imply the existence of the file ; iii) here, by the calling process is meant the real owner of it, not the effective one .
  - the value returned is `0`, if the verified access(es) is/are allowed , respectively `-1` in case at least one of the accesses is forbidden, or of other errors.

7 / 26

The creat system call

- *Creating regular type files*: the `creat` system call.
  The interface of the function `creat` ([5]):
  `int creat(char* pathname, mode_t mode)`
  - *pathname* = the name of the file being created
  - *mode* = access permissions for the newly created file
  - the value returned is the open file descriptor, or `-1` in case of an error.

  Effect: the execution of the `creat` function creates the specified file and, moreover, the file is "open" in writing (!), the returned value having the same meaning as in the `open` system call.

  Remark: if that file already exists, it is truncated to zero, retaining the access permissions it had.

  Note: basically, a call `creat(pathname, mode);` is equivalent to the following call:
  `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

The open system call

- *"Opening" a file*, *i.e.* initializing a *working session*: the `open` system call.
  The interface of the function `open` ([5]):
  `int open(char* pathname, int flags, mode_t mode)`
  - *pathname* = the name of the file that is open
  - *mode* = file access permissions (used only if the call has the effect of creating that file)
  - *flags* = specifies the type of opening, which can be exactly one of the values `O_RDONLY` or `O_WRONLY` or `O_RDWR`, and possibly combined with a combination (*i.e.*, logical disjunction on bits) of some of the following symbolic constants: `O_APPEND`, `O_CREAT`, `O_TRUNC`, `O_CLOEXEC`, `O_NONBLOCK`, `O_EXCL`, `O_DIRECT`, `O_SYNC`, `O_ASYNC`, etc.
  - the value returned is the *open file descriptor*, or `-1` in case of an error.
    Remark: the file descriptor is the index to a new entry created in the process's table of open file descriptions, which references a newly created entry in the system-wide table of open files. For more details see `man 2 open` .

The read system call

- *Reading from a file*: the `read` system call.
  The interface of the function `read` ([5]):
  `int read(int fd, char* buffer, size_t count)`
  - *fd* = the descriptor of the file which is read
  - *buffer* = the memory address to which the read bytes are written
  - *count* = the number of bytes to read from the file
  - the value returned is the number of bytes actually read, if the reading was successful (even partially), or `-1` in case of an error.

  Remarks:

  1. At the end of the reading, the cursor will be positioned on the next byte after the last actually read byte.
  2. The number of bytes actually read may be less than specified (*e.g.*, if at the beginning of the reading the cursor in the file is too close to the end of the file); in particular, it can be even 0, if at the beginning of the reading the cursor in the file is in the `EOF` position (*i.e.*, *end-of-file*).

10 / 26

The write system call

- *Writing to a file*: the `write` system call.
  The interface of the function `write` ([5]):
  `int write(int fd, char* buffer, size_t count)`
  - *fd* = the descriptor of the file to write to
  - *buffer* = the memory address whose content is written to the file
  - *count* = the number of bytes to write to the file
  - the value returned is the number of bytes actually written, if the writing was successful (even partially), or `-1` in case of an error.

  Remarks:

  1. At the end of the writing, the cursor will be positioned on the next byte after the last actually written byte.
  2. The number of bytes actually written may be less than specified (*e.g.*, if that write would increase the space allocated to the file, and this cannot be done for various reasons – lack of free space or exceeding the *quota*).

11 / 26

The lseek system call

- *Position the cursor in a file* (*i.e.* adjusting the current location in the file): the `lseek` system call.
  The interface of the function `lseek` ([5]):
  `long lseek(int fd, off_t offset, int whence)`
  - *fd* = the descriptor of the file been (re)positioned
  - *offset* = the (re)positioning adjustment value
  - *whence* = the adjustment mode, indicated as follows:
    - `SEEK_SET` (=0) : relative to the beginning of the file
    - `SEEK_CUR` (=1) : relative to the current position in the file
    - `SEEK_END` (=2) : relative to the end of the file
  - the value returned is the new file position (always relative to the beginning of the file), or `-1` in case of an error.

The close system call

- *"Closing" a file*, *i.e.* ending a *working session*: the `close` system call.
  The interface of the function `close` ([5]):
  `int close(int fd)`
  - *fd* = the file descriptor
  - the value returned is 0 if closing succeded, respectively `-1` in case of an error.

Remark: the usual way to process a file, *i.e.* a *work session*, consists in the following operations: "opening the file", followed by a loop through it with read and/or write operations, and possibly with changes to the current position in the file, and finally "closing" it.

Example: see the two filter programs `dos2unix.c` and `unix2dos.c` ([2]).

Demo: the solved exercises [AsciiStatistics] and [MyCp] presented in lab support #6 ([3]) illustrates other examples of programs that call I/O functions from the POSIX API for file processing.

Demo: examples of *working sessions* with files

Here is a first example of a program that performs two *working sessions* with files, more precisely it performs a sequential copy of a given file:

```c
/* Basic cp file copy program. POSIX implementation. */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUF_SIZE 4096 // This is exactly the page size, for disk I/O efficiency!

int main (int argc, char *argv []) {
    int input_fd, output_fd;
    ssize_t bytes_in, bytes_out;
    char buffer[BUF_SIZE];
    if (argc != 3) {  printf("Usage: cp file-src file-dest\n");  return 1;  }
    input_fd = open(argv[1], O_RDONLY);
    if (input_fd == -1) {  perror(argv[1]);  return 2;  }
    output_fd = open(argv[2], O_WRONLY|O_CREAT|O_TRUNC, 0600);
    if (output_fd == -1) {  perror(argv[2]);  return 3;  }

    /* Process the input file a record at atime. */
    while ((bytes_in = read(input_fd, buffer, BUF_SIZE)) > 0) {
        bytes_out = write(output_fd, buffer, bytes_in);
        if (bytes_out != bytes_in) {
            fprintf(stderr,"Fatal write error!\n");  return 4;
        }
    }
    close(input_fd);  close(output_fd);  return 0;
}
```

Note: this example is available for download here: `cp_POSIX.c` ([2]).

Demo: examples of *working sessions* with files (cont.)

Here is a second example of a program that performs a *working session* with a file, using `lseek` syscall to read from a specific offset in the given file:

```c
/* Basic program using lseek for reading from a file. POSIX implementation. */
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main () {
    int input_fd;  long offset;  ssize_t bytes_in;  char buffer[6];

    input_fd = open("datafile.txt", O_RDONLY);
    if (input_fd == -1) { perror("open"); return 1; }

    offset = lseek(input_fd, 10, SEEK_SET);
    if (offset == -1) { perror("1st lseek"); return 2; }
    bytes_in = read(input_fd, buffer, 5);
    if (bytes_in == -1) { perror("1st read"); return 3; }
    if (bytes_in != 5) { fprintf(stderr,"1st read warning: insufficient info in file!");}
    buffer[bytes_in]=0; printf("First read from file: %s\n", buffer);

    lseek(input_fd, -10, SEEK_END); /* test for lseek error ... */
    bytes_in = read(input_fd, buffer, 5); /* test for read errors ... */
    buffer[bytes_in]=0; printf("Second read from file: %s\n", buffer);
    close(input_fd);  return 0;
}
```

```
UNIX> gcc -Wall 2nd_program.c ; echo -n "0123456789ABCDEabcde01234" > datafile.txt ; ./a.out
```

```
First read from file: ABCDE
Second read from file: abcde
```


Other I/O system calls

- *Obtaining information contained in the i-node of a file*: the system calls `stat`, `lstat` or `fstat`
- *Changing file access permissions*: the system call `chmod`
- *Changing the owner of a file*: the system calls `chown` and `chgrp`
- *Configuring the access permissions mask when creating a file*: the system call `umask`
- *Creating / deleting a link for a file*: the system call `link`, respectively `unlink`
- *"Duplicating" a file descriptor*: the system calls `dup` and `dup2`
- *Control of I/O operations*: the system calls `fcntl` and `ioctl`
- *Mounting / unmounting a file system*: the system call `mount`, respectively `umount`
- *Creating pipes* (*i.e.*, anonymous communication channels): the system call `pipe`
- *Creating fifo files* (*i.e.*, communication channels with names): the system call `mkfifo`
  The interface of the function `mkfifo` ([5]):
  `int mkfifo(char* pathname, mode_t mode);`
  - *pathname* = the name of the *fifo* file being created
  - *mode* = access rights for it
  - the value returned is 0 on success, or -1 in case of an error.
- etc.

The I/O functions for directories

- *Creating / deleting a directory*: the system call `mkdir`, respectively `rmdir`
  The interface of the function `mkdir` ([5]):
  `int mkdir(char* pathname, mode_t mode);`
  - *pathname* = the name of the directory being created
  - *mode* = access rights for it
  - the value returned is 0 on success, or -1 in case of an error.
- *Finding the current working directory of a process*: the system call `getcwd`
- *Changing the current working directory of a process*: the system call `chdir`
  The interface of the function `chdir` ([5]):
  `int chdir(char* pathname);`
  - *pathname* = the name of the new current working directory of the calling process
  - the value returned is 0 on success, or -1 in case of an error.
- *"Processing" files in a directory*: primitivele `opendir`, `readdir` and `closedir`. Other useful functions: `rewinddir`, `seekdir`, `telldir` and `scandir`.

  A *working session* with directories is implemented in the same way as a file session, *i.e.* it is a sequence of the form: "open directory", a loop with read operations, "close directory".

17 / 26

The template for reading a directory

The data types `DIR` and `struct dirent` are used together with the functions listed above, like this:

```
DIR           *dd;  // open directory descriptor
struct dirent *de;  // directory entry

/* opening the directory */
if( (dd = opendir(pathname)) == NULL)
{
   ...  // TODO: treat the error at opening
}

/* sequential processing of all entries in the directory */
while( (de = readdir(dd)) != NULL)
{
   ...  // TODO: processes the current entry, which has the name: de->d_name
}

/* closing the directory */
closedir(dd);
```

Demo: an example of a program that uses this template – see the solved exercise [MyFind #1] presented in lab support #6 ([3]). This example also illustrates the use of the `stat` system call to find out the properties of a file.

18 / 26

About the *file-system cache* managed by `Linux` kernel

The file system management component within the kernel of an OS, uses an internal memory area in *kernel-space* that implements a *cache* for disk operations (*i.e.*, the contents of the most recently accessed disk blocks are stored in RAM).

This cache is named **file-system cache** (or *disk cache*) in the literature ([4]), and it works according to the same general rules of caches of any kind: i) repeated readings of the same disk block, at very short intervals, will find the information directly from the cache in memory; ii) repeated writes of the same disk block, at very short intervals, will update the information directly in the cache, and the information stored on disk will be updated only once at the time of operation *cache-flushing*; iii) invalidate/update information from *cache*: . . . ; etc.

The granularity of this *cache* (*i.e.*, the **allocation unit** in *cache*) is the physical page, which has a size dependent on the hardware architecture (*e.g.*, for the x86/x64 architecture the page size is 4096 bytes). In other words, the actual I/O operations through DMA between memory and disk transfer blocks of information of this size!

This *file-system cache* is unique per system, *i.e.* there is only one instance of it, managed by OS and used simultaneously (as a " shared resource ") by all the processes running in the system.

Remark: you can read here (in romanian) more details about the implications of the existence of this *file-system cache* for programming applications using the `read` and `write` functions from the POSIX API, including about the use of `O_SYNC` and `O_DIRECT` flags to control the use of this cache.

Standard C library: functions for I/O to files

Outline

About the standard C library

- Standard C library contains functions with limited capability to manage OS resources (*e.g.*, files)
- Often adequate for simple programs
- Possible to write portable programs, between various platforms (*e.g.*, `Windows`, `UNIX/Linux`, etc.)
- Include files: `<stdlib.h>` , `<stdio.h>` and `<string.h>` ([6])
- Competitive performance
- Still constrained to synchronous I/O
- No control of file security via C library

- Call to `fopen()` specifies whether file is text or binary
- *Open files* are identified by pointers to `FILE` structures
  - `NULL` indicates invalid value
  - Pointers are “handles” to *open file* objects
- Errors are diagnosed with the functions `perror()` or `ferror()`

The I/O functions from standard C library

Standard C library contains a set of I/O functions (those from the `<stdio.h>` header file ([6])), which also allow you to process a file in the usual way:

- `fopen` = for “opening” the file
- `fread`, `fwrite` = for binary reading or writing
- `fscanf`, `fprintf` = for formatted reading or writing
- `fclose` = for “closing” the file

Remark: these are library functions (no system calls) and they work buffered, with I/O streams, and the file descriptors they use are not `int`, but `FILE *`.

Note: implementations of these library functions, however, use the system calls which are specific to each platform (*i.e.*, `Windows` vs. `Linux/UNIX`).

Remark: there are many more I/O functions in the `<stdio.h>` library; to see their list and the description of the standard I/O library, including details about those three standard I/O streams (*i.e.*, `stdin`, `stdout` and `stderr`), I recommend you consult the manual page `man 3 stdio` ([6]).

The I/O functions from standard C library (cont.)

What does it mean that these library functions work buffered ?

Answer: it means that they are using a *disk cache* implemented at the standard C library level, i.e. “above” the *file-system cache* managed at the OS kernel level, which I will talk about in my theory classes.
In other words, this is a cache of the information in the *file-system cache*, which in turn is a cache of the information on the disk.

In addition, this cache managed by the stdio library is implemented in *user-space* (as well as all library functions), which means it is *unique per process* and not per system, meaning there is not a single cache of the library to be shared by all processes using stdio library calls.

Conclusion: keep in mind that this stdio library-managed cache is not unique to the system, as in the case of the OS-managed *file-system cache*, but is “local” to the process.

The library functions for I/O format conversion

The library contains a number of functions that perform "formatted" reads/writes, that is, it performs the conversion between the two representations, *binary* vs. *text*, of each data type, based on a *format* argument describing the conversions to be made by some "format specifiers". Those functions are:

- the pair `scanf` / `printf` : read from `stdin` / write to `stdout` ;
- the pair `fscanf` / `fprintf` : reading from a file on disk / writing to a file on disk ;
- the pair `sscanf` / `sprintf` : reading from a string in memory / writing to a string in memory .

The *format* argument uses "format specifiers", in the form '%letter', to describe different types of data, and thus determines what kind of conversion will be made between the two representations, *binary* vs. *text*, of that data type.
For example, here are some format specifiers and the data type associated with each:

- %c : a character
- %s : a null-terminated string
- %d : a `int` (a signed integer), the textual representation being the one corresponding to the writing of the number in base 10
- %u : an `unsigned int` (an unsigned integer), the textual representation being the one corresponding to the writing of the number in base 10
- %o : an unsigned integer, the textual representation being the one corresponding to the writing of the number in base 8
- %x or %X : an unsigned integer, the textual representation being the one corresponding to the writing of the number in base 16
- %f : a `double` (a real number, with sign), the textual representation being the one corresponding to the writing of the number in the notation with decimal point
- %e : a `double`, the textual representation being the one corresponding to the writing of the number in the notation with mantissa E
- etc.

For more details on these function pairs and the *format* argument they use, see the documentation: `man 3 scanf` and `man 3 printf` ([6]).
In addition, you can consult the material available here (in romanian).

24 / 26

Demo: an example of a *working session* with files

Here is an example of a program that performs two *working sessions* with files, more precisely it performs a sequential copy of a given file:

```c
/* Basic cp file copy program. C library implementation. */
#include <stdio.h>
#define BUF_SIZE 4096 // This is exactly the page size, for disk I/O efficiency!

int main (int argc, char *argv []) {
    FILE *input_file, *output_file;
    ssize_t bytes_in, bytes_out;
    char buffer[BUF_SIZE];
    if (argc != 3) {  printf("Usage: cp file-src file-dest\n");  return 1;  }
    input_file = fopen(argv[1], "rb");
    if (input_file == NULL) {  perror(argv[1]);  return 2;  }
    output_file = fopen(argv[2], "wb");
    if (output_file == NULL) {  perror(argv[2]);  return 3;  }

    /* Process the input file a record at atime. */
    while ((bytes_in = fread(buffer, 1, BUF_SIZE, input_file)) > 0) {
        bytes_out = fwrite(buffer, 1, bytes_in, output_file);
        if (bytes_out != bytes_in) {
            fprintf(stderr,"Fatal write error!\n");  return 4;
        }
    }
    fclose(input_file);  fclose(output_file);
    return 0;
}
```

Note: this example is available for download here: `cp_stdio.c` ([2]).

Demo: the solved exercises [ArithmeticMean], [MyExpr] and [MyWc] presented in lab support #6 illustrates other examples of programs that call I/O functions from the standard C library.

Mandatory bibliography

[1] Chapter 3, §3.1 from the book "*Sisteme de operare – manual pentru ID*", by C. Vidrașcu, UAIC Publishing House, 2006. This is available as ebook at the address:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/books/ManualID-SO.pdf

[2] The demo programs from this presentation can be downloaded from:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/lectures/Linux/demo/files/

[3] The online support lesson associated with this presentation:

- https://edu.info.uaic.ro/sisteme-de-operare/SO/support-lessons/C/en/support_lab6.html

Additional bibliography:

[4] Chapter(s) 4, 5, 13, 15 and 18 from the book "The Linux Programming Interface : A Linux and UNIX System Programming Handbook", by M. Kerrisk, No Starch Press, 2010.

- https://edu.info.uaic.ro/sisteme-de-operare/SO/books/TLPI1.pdf

[5] POSIX API : `man 2 access` , `man 2 open` , `man 2 read` , `man 2 write` , etc.

[6] C STANDARD LIBRARY : `man 3 stdio` , `man 3 string` , `man 0p stdlib.h` .