**System calls for working with files and directories in Linux**

**1.    Theoretical Background**

The following article presents the way to use the most common system calls in order to make input-output operations on files, as well as operations to handle files and directories in the Linux operating system.

**2.    File descriptors**

The operating system assigns internally to each opened file a descriptor or an identifier (usually this is a positive integer). When opening or creating a new file the system returns a file descriptor to the process that executed the call. Each application has its own file descriptors. By convention, the first three file descriptors are opened at the beginning of each process. The 0 file descriptor identifies the standard input, 1 identifies the standard output and 2 the standard output for errors. The rest of the descriptors are used by the processes when opening an ordinary, pipe or special file, or directories. There are five system calls that generate file descriptors: *create*, *open*, *fcntl*, *dup* and *pipe*.

**3.    System calls when working with files**

**3.1.  System call OPEN**

Opening or creating a file can be done using the system call open. The syntax is:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

**int** open(**const char** \*path,

**int** flags,... /\* mode\_t mod \*/);

This function returns the file descriptor or in case of an error -1. The number of arguments that this function can have is two or three. The third argument is used only when creating a new file. When we want to open an existing file only two arguments are used. The function returns the smallest available file descriptor. This can be used in the following system calls: *read*, *write*, *lseek* and *close*. The effective UID or the effective GID of the process that executes the call has to have read/write rights, based on the value of the argument *flags*. The file pointer is places on the first byte in the file. The argument *flags* is formed by a bitwise OR operation made on the constants defined in the *fcntl.h* header.

O\_RDONLY

            Opens the file for reading.

O\_WRONLY

            Opens the file for writing.

O\_RDWR

            The file is opened for reading and writing.

O\_APPEND

            It writes successively to the end of the file.

O\_CREAT

            The file is created in case it didn\92t already exist.

O\_EXCL

            If the file exists and O\_CREAT is positioned, calling *open* will fail.

O\_NONBLOCK

In the case of pipes and special files, this causes the open system call and any other future I/O operations to never block.

O\_TRUNC

            If the file exists all of its content will be deleted.

O\_SYNC

            It forces to write on the disk with function *write*. Though it slows down all the system, it can be useful in critical situations.

The third argument, *mod*, is a bitwise OR made between a combination of two from the following list:

S\_IRUSR, S\_IWUSR, S\_IXUSR

Owner: *read*, *write*, *execute*.

S\_IRGRP, S\_IWGRP, S\_IXGRP

Group: *read*, *write*, *execute*.

S\_IROTH, S\_IWOTH, S\_IXOTH

Others: *read*, *write*, *execute.*

The above define the access rights for a file and they are defined in the *sys/stat.h* header.

**3.2.   System call CREAT**

A new file can be created by:

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

**int** creat(**const char** \*path, **mode\_t** mod);

The function returns the file descriptor or in case of an error it returns the value -1. This call is equivalent with:

     open(path, O\_WRONLY | O\_CREAT | O\_TRUNC, mod);

The argument *path* specifies the name of the file, while *mod* defines the access rights. If the created file doesn\92t exist, a new i-node is allocated and a link is made to this file from the directory it was created in. The owner of the process that executes the call - given by the effective UID and the effective GUID - must have writing permission in the directory. The open file will have the access rights that were specified in the second argument (see *umask*, too). The call returns the smallest file descriptor available. The file is opened for writing and its initial size is 0. The access time and the modification time are updated in the i-node. If the file exists (permission to search the directory is needed), it looses its contents and it will be opened for writing. The ownership and the access permissions won\92t be modified. The second argument is ignored.

**3.3.  System call READ**

When we want to read a certain number of bytes starting from the current position in a file, we use the *read* call. The syntax is:

#include <unistd.h>

**ssize\_t** read(**int** fd, **void\*** buf, **size\_t** noct);

The function returns the number of bytes read, 0 for end of file (EOF) and -1 in case an error occurred. It reads *noct* bytes from the open file referred by the *fd* descriptor and it puts it into a buffer *buf*. The pointer (current position) is incremented automatically after a reading that certain amount of bytes. The process that executes a read operation waits until the system puts the data from the disk into the buffer.

**3.4.  System call WRITE**

For writing a certain number of bytes into a file starting from the current position we use the *write* call. Its syntax is:

#include <unistd.h>

**ssize\_t**write(**int**fd,**const void\***buf, **size\_t**noct);

The function returns the number of bytes written and the value -1 in case of an error. It writes *noct* bytes from the buffer *buf* into the file that has as its descriptor *fd*. It is interesting to note that the actual writing onto the disk is delayed. This is done at the initiative of the root, without informing the user when it is done. If the process that did the call or an other process reads the data that haven\92t been written on the disk yet, the system reads all this data out from the cache buffers. The delayed writing is faster, but it has three disadvantages:

a)      a disk error or a system error may cause loosing all the data

b)      a process that had the initiative of a write operation cannot be informed in case a writing error occurred

c)      the physical order of the write operations cannot be controlled.

To eliminate these disadvantages, in some cases the O\_SYNC is used. But as this slows down the system and considering the reliability of today\92s systems it is better to use the mechanism which includes using cache buffers.

**3.5.  System call CLOSE**

For closing a file and thus eliminating the assigned descriptor we use the system call *close*.

#include <unistd.h>

**int** close(int fd);

The function returns 0 in case of success and -1 in case of an error. At the termination of a process an open file is closed anyway.

**3.6.  System call LSEEK**

To position a pointer (that points to the current position) in an absolute or relative way can be done by calling the *lseek* function. Read and write operations are done relative to the current position in the file. The syntax for *lseek* is:

#include <sys/types.h>

#include <unistd.h>

**off\_t** lseek(**int**fd, **off\_t** offset, **int**ref);

The function returns the displacement of the new current position from the beginning of the file or -1 in case of an error. There isn\92t done any I/O operation and the function doesn\92t send any commands to the disk controller. It *ref* is set to SEEK\_SET the positioning is done relative to the beginning of the file (the first byte in the file is at position 0). If *ref* is SEEK\_CUR the positioning is done relative to the current position. If *ref* is SEEK\_END then the positioning is done relative to the end of the file. The system calls *open*, *creat*, *write* and *read* execute an *lseek* by default. If a file was opened using the symbolic constant O\_APPEND then an *lseek* call is made to the end of the file before a write operation.

**3.7.  System call LINK**

To link an existing file to another directory (or to the same directory) link can be used. To make such a link in fact means to set a new name or a path to an existing file. The *link* system call creates a hard link. Creating symbolic links can be done using *symlink* system call. The syntax of link is:

#include <unistd.h>

**int** link(**const char\*** oldpath, **const char\***newpath);

**int** symlink(**const char\*** oldpath, **const char\***newpath);

The function returns 0 in case of success and -1 in case of an error. The argument *oldpath* has to be a path to an existing file. Only the root has the right to set a link to a directory.

**3.8.  System call UNLINK**

To delete a link (a path) in a directory we can use the *unlink* system call. Its syntax is:

#include <unistd.h>

**int** unlink(**const char\*** path);

The function returns 0 in case of success and -1 otherwise. The function decrements the hard link counter in the i-node and deletes the appropriate directory entry for the file whose link was deleted. If the number of links of a file becomes 0 then the space occupied by the file and its i-node will be freed. Only the root can delete a directory.

**3.9.  System calls STAT, LSTAT and FSTAT**

In order to obtain more details about a file the following system calls can be used: *stat*, *lstat* or *fstat*.

#include <sys/types.h>

#include <sys/stat.h>

**int** stat(**const char\*** path, **struct stat\*** buf);

**int** lstat(**const char\*** path, **struct stat\*** buf);

**int** fstat(**int** df, **struct stat\*** buf);

These three functions return 0 in case of success and -1 in case of an error. The first two gets as input parameter a name of a file and completes the structure of the buffer with additional information read from its i-node. The *fstat* function is similar, but it works for files that were already opened and for which the file descriptor is known. The difference between *stat* and *lstat* is that in case of a symbolic link, function *stat* returns information about the linked (refered) file, while *lstat* returns information about the symbolic link file. The *struct stat* structure is described in the *sys/stat.h* header and has the following fields:

struct stat {

mode\_t st\_mode;   /\* file type & rights \*/

ino\_t st\_ino;     /\* i-node \*/

dev\_t st\_dev;     /\* număr de dispozitiv (SF) \*/

nlink\_t st\_nlink; /\* nr of links \*/

uid\_t st\_uid;     /\* owner ID \*/

gid\_t st\_gid;     /\* group ID \*/

off\_t st\_size;    /\* ordinary file size \*/

time\_t st\_atime; /\* last time it was accessed \*/

time\_t st\_mtime; /\* last time it was modified \*/

time\_t st\_ctime; /\* last time settings were changed \*/

dev\_t st\_rdev;   /\* nr. dispozitiv \*/

                  /\* pt. fişiere speciale /

long st\_blksize; /\* optimal size of the I/O block \*/

long st\_blocks;   /\* nr of 512 byte blocks allocated \*/

};

The Linux command that the most frequently uses this function is *ls*. Type declarations for the members of this structure can be found in the *sys/stat.h* header. The type and access rights for the file are encrypted in the *st\_mode* field and can be determined using the following macros:

|  |
| --- |
| ***Table 1. Macros for obtaining the type of a file*** |
| **Macro** | **Meaning** |
| S\_ISREG(st\_mode) | Regular file.  |
| S\_ISDIR(st\_mode) | Directory file. |
| S\_ISCHR(st\_mode) | Special device of type character.  |
| S\_ISBLK(st\_mode) | Special device of type block.  |
| S\_ISFIFO(st\_mode) | Pipe file or FIFO.  |
| S\_ISLNK(st\_mode) | Symbolic link.  |

Decrypting the information contained in the *st\_mode* field can be done by testing the result of a bitwise AND made between the *st\_mode* field and one of the constants (bit mask): S\_IFIFO, S\_IFCHR, S\_IFBLK, S\_IFDIR, S\_IFREG, S\_IFLNK, S\_ISUID (*suid* bit set), S\_ISGID (*sgid* bit set), S\_ISVTX (*sticky* bit set), S\_IRUSR (read right for the owner), S\_IWUSR (write right for the owner), S\_IWUSR (execution right for the owner), etc.

**3.10.    System call ACCESS**

When opening a file with system call *open* the root verifies the access rights in function of the UID and the effective GID. There are some cases though when a process verifies these rights based upon the real UID and GID. A situation when this can be useful is when a process is executed with other access right using the *suid* or *sgid* bit. Even though a process may have root rights during execution, sometimes it is necessary to test whether the real user can or cannot access the file. For this we can use *access* which allows verifying the access rights of a file based on the real UID or GID. The syntax for this system call is:

#include <unistd.h>

**int** access(**const char\*** path, **int** mod);

The function returns 0 if the access right exists and -1 otherwise. The argument mod is a bitwise AND between R\_OK (permission to read), W\_OK (permission to write), X\_OK (execution right), F\_OK (the file exists).

**3.11.    System call UMASK**

To enhance the security in case of operations regarding the creation of files, the Linux operating system offers a default mask to reset some access rights. Encrypting this mask is made in a similar way to the encrypting of the access rights in the i-node of a file. When creating a file those bits that are set to 1 in the mask invalidate the corresponding bits in the argument that specify the access rights. The mask doesnot affect the system call *chmod*, so the processes can explicitly set their access rights independently form the *umask* mask. The syntax for the call is:

#include <sys/types.h>

#include <sys/stat.h>

**mode\_t** umask(**mode\_t** mask);

The function returns the value of the previous mask. The effect of the call is shown below:

     main() /\* test umask \*/

{

int fd;

umask(022);

if ((fd=creat("temp", 0666))==-1)

perror("creat");

system("ls -l temp");

}

The result will be of the following form:

-rw-r--r-- temp

Note that the write permission for the group and other users beside the owner was automatically reset.

**3.12.    System call CHMOD**

To modify the access rights for an existing file we use:

#include <sys/types.h>

#include <sys/stat.h>

**int** chmod(**const char\*** path, **mode\_t** mod);

The function returns 0 in case of a success and -1 otherwise. The *chmod* call modifies the access rights of the file specified by the *path* depending on the access rights specified by the *mod* argument. To be able to modify the access rights the effective UID of the process has to be identical to the owner of the file or the process must have root rights.

The *mod* argument can be specified by one of the symbolic constants defined in the *sys/stat.h* header. Their effect can be obtained by making a bitwise OR operation on them:

|  |
| --- |
| **Table 2.  Bit masks for testing the access rights of a file** |
| **Mode** | **Description** |
| S\_ISUID | Sets the suid bit. |
| S\_ISGID | Sets the sgid bit. |
| S\_ISVTX | Sets the sticky bit. |
| S\_IRWXU  | Read, write, execute rights for the owner obtained from: S\_IRUSR | S\_IWUSR | S\_IXUSR |
| S\_IRWXG  | Read, write, execute rights for the group obtained from: S\_IRGRP | S\_IWGRP | S\_IXGRP  |
| S\_IRWXO | Read, write, execute rights for others obtained from: S\_IROTH | S\_IWOTH | S\_IXOTH |

**3.13.    System call CHOWN**

This system call is used to modify the owner (UID) and the group (GID) that a certain file belongs to. The syntax of the function is:

#include <sys/types.h>

#include <unistd.h>

**int** chown(**const char\*** path, **uid\_t** owner, **gid\_t** grp);

The function returns 0 in case of success and -1 in case of an error. Calling this function will change the owner and the group of the file specified by the argument *path* to the values specified by the arguments *owner* and *grp*. None of the users can change the owner of any file (even of his/her own files), except the root user, but they can change the GID for their own files to that of any group they belong to.

##  Creation of Special Files

The system call mknod creates special files including named pipes, device files, and directories.

mknod (pathname, type and permissions, dev);

where dev specifies major and minor device numbers for block and character special files.

It also sets the file type field to indicate if its a pipe, directory or a special file. If a directory is being created, proper format for the directory is set (such as setting the "." and ".." entries).

**4.    Functions for working with directories**

A directory can be read as a file by anyone whoever has reading permissions for it. Writing a directory as a file can only be done by the kernel. The structure of the directory appears to the user as a succession of structures named directory entries. A directory entry contains, among other information, the name of the file and the i-node of this. For reading the directory entries one after the other we can use the following functions:

#include <sys/types.h>

#include <dirent.h>

**DIR\*** opendir(**const char\*** pathname);

**struct dirent\*** readdir(**DIR\*** dp);

**void** rewinddir(**DIR\***dp);

**int** closedir(**DIR\*** dp);

The *opendir* function opens a directory. It returns a valid pointer if the opening was successful and NULL otherwise.

The *readdir* function, at every call, reads another directory entry from the current directory. The first *readdir* will read the first directory entry; the second call will read the next entry and so on. In case of a successful reading the function will return a valid pointer to a structure of type *dirent* and NULL otherwise (in case it reached the end of the directory, for example).

The *rewinddir* function repositions the file pointer to the first directory entry (the beginning of the directory).

The *closedir* function closes a previously opened directory. In case of an error it returns the value -1.

The structure *dirent* is defined in the *dirent.h* file. It contains at least two elements:

struct dirent {

ino\_t d\_fileno;               // i-node nr.

char d\_name[MAXNAMLEN + 1];   // file name

}

##  Change Directory and Change Root

When process 0 is created, it sets its current directory as root. It gets the root inode (iget), saves it in its u-area as the current directory and releases the inode. When a new process is created with fork, it inherits the current directory from the parent process in its u-area and the inode reference count is incremented.

chdir (pathname);

In a way, chdir is similar to open, as both of them leave the inode allocated. In the case of chdir, the inode will be release only when the process calls chdir again with different pathname, or when the process exits.

Processes usually use the global root "/" as for pathnames starting with "/". The kernel has a global variable that points to the inode of the global root. Processes can change their current root via chroot system call. This is useful for simulations.

chroot (pathname);

The algorithm is similar to chdir. But if the old root was the global root, it is not released. When the current root is changed, searches for pathnames starting from "/" will start from the new root. And all of the child process will inherit the current root.

## Dup

The dup system call copies the given file descriptor to the first free slot in the user file descriptor table, and returns the new file descriptor.

newfd = dup (fd);

Since it duplicates the entry in the user file descriptor, it increments the reference count in the file table.

## Mounting and Unmounting File Systems

A physical disk may have many logical partitions. And if each partition has a file system on it, it means that it will have its own boot block, super block, inode list, and data blocks. The mount system call can attach such a file system on a specified location on the current file system. The unmount system call can detach it. The mount system call thus, allows data on a disk block to be read as a file system instead of sequence of a disk blocks.

mount (special pathname, directory pathname, options);

where special pathname is the name of the device file of the disk section whose file system is to be mounted. The directory pathname is the path in existing file system where the new file system will be mounted. The options indicate whether to mount in a "read-only" manner.

For example, if this system call is made:

mount ("/dev/dsk1", "/usr", 0);

the kernel attaches the file system on "/dev/disk1" on the directory "/usr". The root of the file system on "/dev/dsk1" will be accessed by "/usr". The processes can seamlessly access this newly mounted file system. Only the link system call checks if the file system is same for the files being linked.

The kernel has a mount table which has entries for each mounted file system. The contents of each entry are:

* The device number of the file system.
* Pointer to a buffer containing the super block of the mounted file system.
* Pointer to a buffer containing the root inode of the mounted file system.
* Pointer to a buffer containing the inode of the mount point on the existing file system.

### Unmounting a File System

unmount (special filename);

where special filename indicates the file system to be unmounted.

Before unmounting the file system, it is checked if any file on the file system is in use. After unmounting the file system, the buffer that hold the data of the unmounted file system do not need to be cached. So they are release from the buffer pool. They are placed in the front of the free list so that buffers with valid data will remain on the buffer pool for a long time.