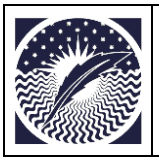

Sistemas Operativos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas de Prácticas y Desarrollo
PRÁCTICA 9: System Calls

Objectives

1. Introduce the concept of system call
2. Use system calls for working with the file system

Concepts

1. System Calls

The first thing we need to understand is the concept of system call. S*ystem calls* represent the main interface between the operating system applications and the OS kernel. That is, system calls are request sent to the OS kernel for carrying out some specific functions, for example controlling a physical device or execute some specific action.
Let us consider what has been for most of us the very first C program:

```
//example.c
#include <stdio.h>
#include <stdlib.h>

int main() {
      printf("Hello World!\n");
       return 0;
}
```

In order to print the "Hello World" message, we have used the standard C function printf(). Even if simple, this function is implemented as a series of system calls, that do something like this:

1. Open the STDOUT file (standard output) for writing.
2. Analyse the string that have to be written to STDOUT.
3. Write the previous string to STDOUT.

Thus, as we have seen, printf() rises various system calls: for opening a file, for writing in a file, etc.. This holds also for other C functions.

Knowing the system calls and how to use them, will help us to get a deeper knowledge of how the OS works. Among the basic, but at the same time very important, system calls, we can find those for handling files and directories. In Linux almost everything is a file. This makes that system calls for managing files very important in this operating system.

2. File System.

In C, there exist special libraries and functions that can be used in order to directly interact with the OS kernel. By using such functions, we will acquire a more direct control of our actions and we'll be able to program independently from the commands that the shell provides.
Some of these functions are related with the file system. In order to use them in our programs, we have to include the following libraries:

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

2.1 Creating, Opening and Closing a File

Let us start our study from how to create a file. There are two functions for this purpose:

```
int creat( const char *pathname, mode_t mode )
int open( const char *pathname, int flags, mode_t mode )
```

Originally open() could only be used for opening an existing file, so files that had been previously created with the function create(). Nowadays, a new parameter has been added to the prototype of open(), allowing in this way the creation of files

Parameters:

*const char *pathname*: path of the files that we want to create/open.

mode_t mode: this parameter is responsible to define the permission of the file. The list of the possible values that can be used is the following:

Value Meaning

S_IROTH: read permission, all users.
S_IWOTH: write permission, all users.
S_IXOTH: execute permission, all users.
S_IRGRP: read permission, group
S_IRGRP: write permission, group
S_IRGRP: execute permission, group
S_IRUSR: read permission, owner
S_IWUSR: write permission, owner
S_IXUSR0: execute permission, owner
S_IRWXU: read, write, execute/search by owner
S_IRWXG: read, write, execute/search by group.
S_IRWXO: read, write, execute/search by all users.

int flags: values for *flags* are constructed by a bitwise-inclusive OR of flags from the following list, defined in <*fcntl.h*>. Applications shall specify exactly one of the first three values (file access modes) below in the value of *flags*:

Value Meaning

O_RDONLY**:** Open for reading only.
O_WRONLY: Open for writing only.
O_RDWR: Open for reading and writing. The result is undefined if this flag is applied to a FIFO.
O_CREAT: If file does not exist, then it will be created.
O_TRUNC: If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged
O_APPEND: The file offset shall be set to the end of the file prior to each write.

These values can be combined in order to use more than one, for example:

open(pathname, O_WRONLY | O_TRUNC | O_APPEND, S_IROTH)

Output:

Both open() or creat() return an integer, which will be -1 in case of errors, otherwise it will be a positive number. This number corresponds to a file descriptor, which will be used in successive calls made on the file. Each process has a file descriptor table, that allows the process to handle files in a simple way. Initially entries 0, 1 y 2 are reserved for files STDIN, STDOUT and STDERR, respectively. These descriptors correspond to the standard input, standard output and the standard error output. Successive entries will be assigned to files that will be opened or created.

In order to open a file that already exists, you can use another version of open():

```
int open( const char *pathname, int flags)
```

Parameters:

*const char *pathname*: file to open
int flags: type of access that will be used.

Output:

same as above

When we are done using the file, we have to close it. In order to do so, we can use the following system call:

```
int close(int fd)
```

Parameters:
int fd: file descriptor.

Output:
0, if the file was successfully close, -1 otherwise

We know that in Linux some files are links to other fiels. We can create a link with the following function:

```
int link(const char *oldpath, const char *newpath)
```

Parameters:

*const char *oldpath*: original file path
*const char *newpath:* link path

Output:
0, if the link was successfully close, -1 otherwise

In order to remove a link, we can simply use the following function:

```
int unlink(const char *pathname)
```

Parameters:
*const char *pathname:* link to be removed

Output:
0, if the link was successfully removed, -1 otherwise

2.2 Using Files

The next logical step is to read or write the files that we have opened/created. To this aim, we will use two very similar system calls: read() y write(). These are their prototypes:

```
ssize_t read( int fd, void *buf, size_t count )
ssize_t write( int fd, void *buf, size_t count )
```

Parameters:

int fd: descriptor of the file (value returned by the open or creat).
*void *buf*: buffer which will be used for storing the data read from the file or the data we want to write in the file.
size_t count: number of bytes that we want to read or number of bytes of the buffer to be written in the file.

Output:
Number of bytes that were actually read or written, -1 in case of errors.

Types used for counting the number of bytes may seem strange, but they are integer. They are used in order to maintain the compatibility with older version of the functions. The same holds for the rest of the calls we will see in this class.

2.3 Moving within Files

Often, what we want, is to start writing or reading a file from a specific position, not necessarily from the beginning. The lseek() allows us to do exactly this. In fact it is a system call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms. The prototype of the function is the following:

```
off_t lseek(int fd, off_t offset, int whence)
```

In order to use this function we have to include the unistd.h library.

Parameters:

int fd: The file descriptor of the pointer that is going to be moved.
off_t offset: The offset of the pointer (measured in bytes). We'll use it as a long integer.
int whence: The method in which the offset is to be interpreted (relative, absolute, etc.). Legal values for this variable are :

SEEK_SET Set the position within the file to "offset" bytes after the beginning.
SEEK_CUR Set the position within the file to "offset" bytes from the actual position.
SEEK_END Set the position within the file to "offset" bytes from the end.

Output: -1 in cae of errors. Otherwise, the function returns the new location, in bytes, in the file, taking as origin the beginning of the file.

Example:
lseek(fd, 300, SEEK_END);

With the previous instruction, we change the pointer to 300 bytes from the end of the file whose descriptor is fd. Notice that in this case a negative value for the offset should have been used, otherwise the current position is set after the end of the file.

2.4 File Properties.

As we have seen in previous classes, a set of attributes is associated to files. For example, the date of last motification, who's the owner, access permission, etc. We can access these attributes with the stat() function. This function is a bit different from the other function we have seen so far. In fact, it uses a data structure with all the possible features of a file, and when called, such a structure has to be passed as a reference. At the end of the call, the structure will contain all the information about the file. The structures that has to be used is the following:

```
struct stat {
      dev_t st_dev; /* device */
      ino_t st_ino; /* inode number*/
      mode_t st_mode; /* file mode */
      nlink_t st_nlink; /* hard links number*/
      uid_t st_uid; /* owner's UID*/
      gid_t st_gid; /* owner's GID*/
      dev_t st_rdev; /* device type */
      off_t st_size; /* size, in bytes */
      blksize_t st_blksize; /* preferred block size */
      blkcnt_t st_blocks; /* blocks assined */
      time_t st_atime; /* time of last access*/
      time_t st_mtime; /* time of last modification */
      time_t st_ctime; /* time of last modification in an inode*/
};
```

Don't worry, you do not have to define this structure. It is already defined in the libraries that we have to include in order to use system calls. There are two version of the function:

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Parameters:
*const char *path*: name of the file.
*struct stat *buf*: structure where the data will be stored.
int fd: descriptor of the file.

Output: 0 if successful, -1 otherwise

These two functions basically have the same effect. The only difference is that the first function receives the name of the file, while the second its descriptor.

In addition to check the properties of a file, we can also change some of them.

Function **chmod()** has the same effect as the command: change the access permission of a file. Even if we are using the system calls from a C program, we are still restricted by the protection mechanisms implemented by the file system. This means that only the owner (or the system administrator) can change the access permission of a file.

```
int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

Parameters:
*const char *path*: file name.
mode_t mode: new access permission. We should use three numbers as we have seen in the first class (0777, 0,566,etc.)
int fd: file descriptor.

Output: 0 if successful, -1 otherwise.

Also in this case the two funcitons have the same purpose, the only difference lies in the first parameter.

Example:
chmod("/home/informatica/prueba", 0666);

Finally, we can also change the owner of a file:

```
int chown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Parameters:
*const char *path*: file name.
uid_t owner: integer number that identify the user, as it appears in file /etc/passwd.
gid_t group: integer file that identify the group, as in /etc/group.
int fd: file descriptor.

If we use a -1 for "owner" or "group", it means that the original value will be kept.

Output: 0 if successful, -1 otherwise.

Example:
gid_t group = 100; /* 100 is GID of group users */
chown("/home/informatica/prueba", -1, group);

3. Managing Directories

In the previous sections, we have seen the most basic system calls for managing files. However, often such calls are not enough when we are working with the file system. It is also common to work with directories, and not only with files, for example it may be necessary to know the current directory, how to create or delete directories, how to change directory, etc. In this section we will look at some system calls that allow to do exactly this.
Let us start from the simplest call, that answer the question of: where am I? i.e., what's the current working directory (CWD)? The function that provides this information is **getcwd** and has the following prototype:

```
char *getcwd(char *buf, size_t size);
```

Parameters:
*char *buf*: buffer where the pathname of current working directory will be stored
size_ti size: size in bytes of the character array pointed to by the *buf* argument

Output: Upon successful completion, *getcwd*() shall return the *buf* argument. Otherwise, *getcwd*() shall return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by *buf* are then undefined.

If we want to change the current working directory, we can use one of these functions:

```
int chdir(const char *path);
int fchdir(int fd);
```

Parameters:
*const char *path*: path of the new working directory
int fd: descriptor of the file corresponding to the new working directory

Output: 0 if successful, -1 otherwise

In order to create or delete directories, we can use the following functions:

```
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

Parameters:
*const char *path*: path of the directory we can to create/remove.
mode_t mode: number indicating the permission of the directory to be created (e.g., 0777);

Salida: 0 if successful, -1 otherwise

Finally, let us take a look at how we can check the content of a diretory. For this, we will use the *dirent* structure and the following functions:

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

Parameters:
*const char *name:* name of the directory.
*DIR *dir:* pointer to the directory

The *opendir*() function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. Upon successful completion, *opendir*() shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned.

Once we have the DIR pointer, we can pass it to the readdir function, which will return a pointer to a dirent structure. Each element contained in the directory will be represented with such a structure, which is defined as follows:

```
struct dirent {
      ino_t d_ino; // entry i-node number
      off_t d_off; // offset of the directory entry in the actual
file system directory.
      wchar_t d_reclen; // record length of this entry
      char d_name[MAX_LONG_NAME+1] // name of the entry
}
```

This structure seems complex, but it's actually not. Moreover, we are interested only in the last field, d_name.
In order to list all the entries of a directory, we first open the directory with opendir(), then we read each entry with readdir() until this function returns NULL (meaning all the entries have been read). Finally we will close the directory with closedir().

Bibliografia

1. Tanenbaum. Sistemas Operativos Modernos, 2ª edición. Pág. 379-399.
2. http://www.die.net/doc/linux/man/man2/syscalls.2.html
3. http://www.digilife.be/quickreferences/QRC/LINUX%20System%20Call%20Quick%20Reference.pdf

Experimentos

E1 (10 min) Compilar y hacer varias ejecuciones del siguiente programa. Describir su funcionamiento.

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

int main( int argc, char *argv[] )
{
   int fd;
   if( (fd = open( argv[1], O_RDWR )) == -1 )
    {
      perror( "open" );
      exit( -1 );
     }
```

```
     printf( "El archivo abierto tiene el descriptor %d.\n", fd );
     close( fd );
return 0;
}
```

E2 (10 min) Ejecutar este programa usando como parámetro en la llamada un archivo anteriormente creado que solo contenga la palabra "Hola". ¿Que es lo que aparece en pantalla?, ¿Como se solucionaría el problema?

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

#define STDOUT  1
#define SIZE  512

int main( int argc, char *argv[] )
 {
       int fd,r;
       char buffer[SIZE];
       if( (fd = open( argv[1], O_RDWR )) == -1 )
           {
                 perror( "open" );
                 exit( -1 );
           }
         r= read( fd, buffer, SIZE );
         write( STDOUT, buffer, r );
        close( fd );
  return 0;
}
```

E3 (10 min) Ejecute y compruebe los efectos del siguiente programa:

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

int main( int argc, char *argv[] ){
    char buffer[512];
    printf( "El directorio actual es: %s\n",getcwd( buffer, 512) );
    chdir( ".." );
    mkdir( "./directorio1", 0755 );
    mkdir( "./directorio2", 0755 );
    rmdir( "./directorio1" );
return 0;
}
```

E4 (10 min) Compile y ejecute el siguiente código para poder recorrer el contenido de un determinado directorio, el cual será parámetro del procedimiento:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
int main( int argc, char *argv[] )
{
      DIR *dir;
      struct dirent *mi_dirent;
      if( argc != 2 )
      {
```

```
            printf( "%s: %s directorio\n", argv[0], argv[0] );
            exit( -1 );
      }
      if( (dir = opendir( argv[1] )) == NULL )
      {
            perror( "opendir" );
            exit( -1 );
      }
      while( (mi_dirent = readdir( dir )) != NULL )
            printf( "%s\n", mi_dirent->d_name );
      closedir( dir );
      return 0;
}
```

E5 Analice el siguiente código para buscar los elementos pasado por linea de comandos en el directorio actual.

```
//    The following sample program searches the current directory for each of the
arguments supplied on the command line.

    #include <dirent.h>
    #include <errno.h>
    #include <stdio.h>
    #include <string.h>


    static void lookup(const char *arg)
    {
        DIR *dirp;
        struct dirent *dp;


        if ((dirp = opendir(".")) == NULL) {
            perror("couldn't open '.'");
            return;
        }

        do {
            errno = 0;
            if ((dp = readdir(dirp)) != NULL) {
                if (strcmp(dp->d_name, arg) == 0){
                  printf("found %s\n", arg);
                  closedir(dirp);
                  return;
                }
            }
        } while (dp != NULL);

        if (errno != 0)
            perror("error reading directory");
        else
            printf("failed to find %s\n", arg);
            closedir(dirp);
        return;
    }

    int main(int argc, char *argv[])
    {
        int i;
        for (i = 1; i < argc; i++)
```

```
            lookup(argv[i]);
        return (0);
    }
```

Problemas

P1) (40 min) Implemente un programa en C que al pasarle como argumento un fichero, nos diga sus características. Use la estructura *stat.*

P2) (40 min) Escribir un programa para "mover" ficheros. Usar dos argumentos, uno para indicar el fichero origen y otro para indicar el nuevo destino. Usar las llamadas link y unlink.

Ampliación de Problemas

AP1) (90 min) Implemente un programa en C que permita ejecutar operaciones en el sistema de archivos. El programa tiene que permitir copiar archivos de un lugar a otro, cambiar los nombres de archivos y directorios, crear y borrar directorios, y obtener un listado del contenido de un directorio. Por ejemplo, el programa resultante podría proporcionar el siguiente menú al usuario:

MENU
_ _ _ _

1.- Copiar archivo
2.- Cambiar nombre archivo
3.- Cambiar nombre directorio
4.- Mover archivo
5.- Listado directorio actual

Elija una opción:

De esta forma el usuario puede interactuar con el programa. Para la opción 1, el programa necesita fichero origen y path destino en donde copiarlo. Para la opción 2 se necesitan el nombre del fichero a renombrar y el nuevo nombre. Para la opción 3, necesita el nombre del directorio a renombrar y el nuevo nombre. Para la opción 4, necesita el nombre del archivo a mover y su destino. Cuando el usuario seleccione una opción, el propio programa le irá pidiendo los datos pertinentes.