Experiment No 2

Title: Study and implement file management using low level file commands

Aim: To implement file management using low level file commands in Linux

Objective: To study and implement following

1. OPEN system call
2. CREAT system call
3. READ system call
4. WRITE system call
5. CLOSE system call

Relevance: To get knowledge about low level file commands in Linux

Theory:

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.

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- t 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.

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. 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.

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 losing 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.

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.

Examples

- **Example 1**

Write a program that creates a file with a 4K bytes free space. Such files are called files with holes.

```
#include <sys/types.h>

#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
char buf1[]="LAB ";
char buf2[]="OS Linux";
```

```
int main( void)
{
        int fd;
        if ((fd=creat("file.gol", 0666)) < 0)
         {
                perror("Creation error");
            exit (1);
        }
        if (write(fd, buf1, sizeof(buf1)) < 0)
        perror("Writing error");
        exit(2);
        }

        if (lseek(fd, 4096, SEEK_SET) < 0)
        perror("Positioning error");
        exit(3);
        }

        if (write(fd, buf2, sizeof(buf2)) < 0)
        perror("Writing error");
        exit(2);
        }
}
```

Trace the execution of the program with the help of the following commands:

```
ls -l
stat file.gol

od -c file.gol
```

- **Example 2**

Write a program that copies the contents of an existing file into another file. The names of the two file should be read as an input from the command line. You may presume that any of the commands *read* or *write* may cause errors.

```
#include <sys/types.h>

#include <sys/uio.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE 512

int main (int argc, char** argv)
{
        int from, to, nr, nw, n;
        char buf[BUFSIZE];
```

```
        if ((from=open(argv[1], O_RDONLY)) < 0) {
        perror(Error opening source file);
        exit(1);
    }

    if ((to=creat(argv[2], 0666)) < 0) {
        perror("Error creating destination file");
        exit(2);
    }
        while((nr=read(from, buf, sizeof( buf))) != 0) {
      if (nr < 0) {
         perror("Error reading source file");
              exit(3);
      }
      nw=0;
      do {
        if ((n=write(to, &buf[nw], nr-nw)) < 0) {
          perror("Error writing destination file");
          exit(4);
        }
        nw += n;
      } while (nw < nr);
    }
    close(from); close(to);
}
```