AAiT ITSE 4931: Operating Systems

Laboratory 7

File Management

Objective:

The aim of this laboratory is to show you some of the aspects of the UNIX file system. This lab session discusses the system calls to manage files and directories under UNIX/LINUX operating systems. There are many system calls that are used to manage files and directories. In this lab we will be practicing simple programs using some simple file structure related system calls.

Using system calls for processing files.

The file structure related system calls available in the UNIX system let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. A channel is a connection between a process and a file that appears to the process as an unformatted stream of bytes.

When doing I/O, a process specifies the file descriptor for an I/O channel, a buffer to be filled or emptied, and the maximum size of data to be transferred. An I/O channel may allow input, output, or both. Furthermore, each channel has a read/write pointer. Each I/O operation starts where the last operation finished and advances the pointer by the number of bytes transferred. A process can access a channel's data randomly by changing the read/write pointer.

All input and output operations start by opening a file using either the "creat( )" or "open( )" system calls. These calls return a file descriptor that identifies the I/O channel. File descriptors are numbered sequentially, starting from zero. Every process may have up to 20 (the usual system limit) open files at any one time, with the file descriptor values ranging between 0 and 19.

By convention the first three file descriptor values have special meaning:

Value	Meaning
0	Standard Input (Screen)
1	Standard Output (Screen)
2	Standard Error

For example, the printf ( ) library function always sends its output using file descriptor 1, and scanf ( ) always reads its input using file descriptor 0 which is the display screen. When a reference to a file is closed, the file descriptor is freed and may be reassigned by a subsequent open ( ).

File Descriptors

To the kernel, all open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open or creat as an argument to either read or write.

The creat( ) System call

The creat() system call can be used to create new files. This call will try to create a new file and upon success it will return an integer value representing the new file's descriptor, the call didn't succeed the call will return -1 representing failure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
        int creat (const char *name, mode_t mode);
```

The following is a typical creat( ) call:

```
int fd;
fd = creat (file, 0644);
if (fd == -1)
        /* error */
else
        /*other processing on the file*/
```

The mode argument is the familiar UNIX permission bit set, such as octal 0644 (owner can read and write, everyone else can only read).

Practical 1: Implement the following program demonstrating creat()

```
#include <stdio.h>
#include <sys/types.h>/* defines types used by sys/stat.h */
#include <sys/stat.h>/* defines S_IREAD & S_IWRITE       */
  int main( ) {
     int fd;
     fd = creat("datafile.dat", S_IREAD | S_IWRITE);
     if (fd == -1)
        printf("Error in opening datafile.dat\n");
     else
        {
        printf("datafile.dat opened for read/write access\n");
        printf("datafile.dat is currently empty\n");
        }
     close(fd);
     exit (0);
  }
```

The open( ) System Call

A file is opened, and a file descriptor is obtained with the open( ) system call:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
        int open (const char *name, int flags);
        int open (const char *name, int flags, mode_t mode);
```

The open( ) system call maps the file given by the pathname name to a file descriptor, which it returns on success. The file position is set to zero, and the file is opened for access according to the flags given by flags.

Flags for open( )

The flags argument must be one of O_RDONLY, O_WRONLY, or O_RDWR. Respectively, these arguments request that the file be opened only for reading, only for writing, or for both reading and writing.

For example, the following code opens /home/student/test for reading:

```
int fd;
fd = open ("/home/student/test", O_RDONLY);
if (fd == -1)
        /* error */
else
        /*other processing on the file*/
```

Additional flags can be used with open to perform additional tasks. The flags argument can be bitwise-ORed with one or more of the following values, modifying the behaviour of the open request:

O_APPEND: The file will be opened in append mode.

O_CREAT: If the file denoted by name does not exist, the kernel will create it.

O_DIRECTORY: Is used to open directories. If name is not a directory, the call to open ( ) will fail.

O_TRUNC: If the file exists and is open for writing then the file will be truncated to zero length.

For example, the following code opens for writing the file /home/student/test. If the file already exists, it will be truncated to a length of zero. Because the O_CREAT flag is not specified, if the file does not exist, the call will fail:

```
int fd;
fd = open ("/home/student/test", O_WRONLY | O_TRUNC);
if (fd == -1)
        /* error */
else
        /*other processing on the file*/
```

Reading via read( )

Now that you know how to open a file, let’s look at how to read it. The most basic and common mechanism used for reading is the read( ) system call:

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

Each call reads up to len bytes into buf from the current file offset of the file referenced by fd. On success, the number of bytes written into buf is returned. On error, the call returns -1. The file position is advanced by the number of bytes read from fd.

It is legal for read( ) to return a positive nonzero value less than len. This can happen for a number of reasons: mostly because of less than len bytes may have been available. The possibility of a return value of 0 is another consideration when using read( ). The read( ) system call returns 0 to indicate end-of-file (EOF); in this case, of course, no bytes were read.

Size measures
The size_t and ssize_t types are mandated by POSIX. The size_t type is used for storing values used to measure size in bytes. The ssize_t type is a signed version of size_t. On 32-bit systems, the backing C types are usually unsigned int and int, respectively.

Writing with write( )
The most basic and common system call used for writing is write( ). write( ) is the counterpart of read( ):

```
#include <unistd.h>
ssize_t write (int fd, const void *buf, size_t count);
```

A call to write( ) writes up to count bytes starting at buf to the current file position of the file referenced by the file descriptor fd. On success, the number of bytes written is returned, and the file position is updated in kind. On error, -1 is returned. A call to write( ) can return 0, but this return value does not have any special meaning; it simply implies that zero bytes were written.

As with read( ), the most basic usage is simple:

```
const char *buf = "My ship is solid!";
ssize_t nr;
/* write the string in 'buf' to 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
        /* error */
```

Closing Files
After a program has finished working with a file descriptor, it can un-map the file descriptor from the associated file via the close( ) system call:

```
#include <unistd.h>
int close (int fd);
```

A call to close( ) un-maps the open file descriptor fd, and disassociates the process from the file. The given file descriptor is then no longer valid, and the kernel is free to reuse it as the return value to a subsequent open( ) or creat( ) call. A call to close( ) returns 0 on success. On error, it returns -1. Usage is simple:

```
int status = close(fd);
if (status == -1)
        /* display an error message */
else
        /* display close success message */
```

Practical 1: Creating a file and writing some data to it.

Write the following program which creates a file using the system calls. Compile and run the file to see whether the file is created and the given data is written to it.

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

```
#include <string.h>

void main(){
  int fd;
  char *file = "/home/student/filest.txt";
  const char *buf = "My ship is solid!";
  ssize_t nr;
  int status;

  fd = creat (file, 0644);
  if (fd == -1)
    printf("\n\n\tThere was an error creating the file");

  nr = write (fd, buf, strlen (buf));
  if (nr == -1)
    printf("\n\n\tThere was an error writing to the file\n");

  status = close(fd);
  if (status == -1)
    printf("\n\n\tThere was an error closing the file\n");
}
```

Exercises:

Exercise 1: Write a program that uses the read system call to read the file created above and display its Contents to the screen.

Exercise 2: Read about what lseek() function is used for. How can we apply lseek(). Implement the above program, write on the file in the middle using lseek().

Exercise 3: Append some additional text into the previously created file.

Exercise 4: Implement a simple file copying program, by reading from the first file and writing it to the second File.

Exercise 5: Implement your own version of a command to display a file that should be almost same as the UNIX cat command that displays a requested file on the screen. Name your command as displayfile that should have the following syntax:

```
displayfile  file_name
```