**ICS 431 Operating Systems**

**Lab 8: File-Related System calls**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* **Objective:**

The purpose of this lab is to give a brief idea about file-related system calls, how they work and what is their use. There are many system calls that are used to manage files and directories, processes, memory, interrupts or signals. In this lab we will be practicing simple programs using some simple file structure related system calls to get an overview of system calls.

**File Related Major System Calls:**

The following table lists about 40 of the more important system calls including file-related system calls:

|  |  |  |
| --- | --- | --- |
| **GENERAL CLASS** | **SPECIFIC CLASS** | **SYSTEM CALL** |
| **File Structure** **Related Calls** | **Creating a Channel** | **creat( ), open( ), close( )** |
| **Input/Output** | **read( ), write( )** |
| **Random Access** | **lseek( )** |
| **Channel Duplication** | **dup( )** |
| **Aliasing and Removing Files** | **link( ), unlink( )** |
| **File Status** | **stat( ), fstat( )** |
| **Directory Information** | **opendir( ), readdir( )** |
|  | **Access Control** | **access( ), chmod( ), chown( ), umask(**  **)** |
|  | **Device Control** | **ioctl( )** |
| **Process Related** **Calls** | **Process Creation and Termination** | **exec( ), fork( ), wait( ), exit( )** |
| **Process Owner and Group** | **getuid( ), geteuid( ), getgid( ), getegid( )** |
| **Process Identity** | **getpid( ), getppid( )** |
| **Process Control** | **signal( ), kill( ), alarm( )** |
| **Change Working Directory** | **chdir( )** |
| **Inter Process** **Communication** | **Pipelines** | **pipe( )** |
| **Messages** | **msgget( ), msgsnd( ), msgrcv( ), msgctl( )** |
| **Semaphores** | **semget( ), semop( )** |
| **Shared Memory** | **shmget( ), shmat( ), shmdt( )** |

**File Structure Related System Calls :**

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. The kernel presents and accepts data from the channel as a process reads and writes that channel. To a process then, all input and output operations are synchronous and unbuffered.

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. Recall that file descriptors 0, 1, and 2 refer to standard input, standard output, and standard error files respectively, and that file descriptor 0 is a channel to your terminal's keyboard and file descriptors 1 and 2 are channels to your terminal's display screen.

# creat( )

The prototype for the **creat( )** system call is:

**int creat(file\_name, mode) char \*file\_name; int mode;**

where file\_name is pointer to a null terminated character string that names the file and mode defines the file's access permissions. The mode is usually specified as an octal number such as 0666 that would mean read/write permission for owner, group, and others or the mode may also be entered using manifest constants defined in the "**/usr/include/sys/stat.h**" file. If the file named by file\_name does not exist, the UNIX system creates it with the specified mode permissions. However, if the file does exist, its contents are discarded and the mode value is ignored. The permissions of the existing file are retained. Following is an example of how to use **creat( )**:

/\* This program explains use of creat( ) system call\*/

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.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);

}

# open( )

The file management system calls allow you to manipulate the full collection of regular, directory, and special files.

In most cases, **open** ( ) is used to initially access or create a file. If the file system call succeeds, it returns a small integer called a file descriptor that is used in subsequent I/O operations on that file. If **open** ( ) fails, it returns **–1**. When a process no longer needs to access an open file, it should close it using the **close** ( ) system call. All of a process’s open files are automatically closed when the process terminates. Although this means that you may often omit an explicit call to **close** ( ), it’s better programming practice to explicitly close your files.

**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** ( ).

**A single file may be opened several times and thus have several file descriptors associated with it.** Each file descriptor has its own private set of properties that have nothing to do with the file.

**When a file descriptor is created, its file pointer is positioned at offset 0 in the file (the first character) by default.** **As the process reads and/or writes, the file pointer is updated accordingly.** For example, if a process opened a file and then read **10** bytes from the file, the file pointer would end up positioned at offset **10**. If the process then wrote **20** bytes, the bytes at offset **10..29** in the file would be overwritten and the file pointer would end up positioned at offset **30**.

**int open (char \**filename*, int *mode* [, int *permissions*])**

**open ( )** allows you to open or create a file for reading and/or writing. ***filename*** is an absolute or relative pathname and ***mode*** is a ORing of a read/write flag together with zero or more miscellaneous flags. ***permissions*** is a number that encodes the value of the file’s permission flags, and **should only be supplied when a file is being created**. The values of the predefined read/write and miscellaneous flags are stored in “usr/include/sys/file.h”. The read/write flags are as follows:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **FLAG** |  | **MEANING** |
| **O\_RDONLY** |  | Open for **read-only**. |
| **O\_WRONLY** |  | Open for **write-only**. |
| **O\_RDWR** |  | Open for **read** and **write**. |

The additional miscellaneous flags are:

**O\_APPEND** Position the file pointer at the **end of the file** before each write ( ).

**O\_CREAT** If the **file does not exist**, **create** the file.

**O\_TRUNC** If the **file exists**, it is **truncated** to length **zero**.

If **successful** **open** ( ) returns a **non-negative** file descriptor, otherwise it returns **–1**.

**#include <fcntl.h>**

**int open(file\_name, option\_flags [, mode])** **char \*file\_name;** **int option\_flags, mode;**

where **file\_name** is a pointer to the character string that names the file, **option\_flags** represent the type of channel, and **mode** defines the file's *access permissions* if the file is being created.

The allowable option\_flags as defined in "**/usr/include/fcntl.h**" are:

#define O\_RDONLY 0 /\* Open the file for reading only \*/

#define O\_WRONLY 1 /\* Open the file for writing only \*/

#define O\_RDWR 2 /\* Open the file for both reading and writing\*/

#define O\_APPEND 010 /\* append (writes guaranteed at the end) \*/

#define O\_CREAT 00400 /\*open with file create (uses third open arg) \*/

#define O\_TRUNC 01000 /\* open with truncation \*/

#define O\_EXCL 02000 /\* exclusive open \*/

Multiple values are combined using the | operator (i.e. bitwise OR). Note: some combinations are mutually exclusive such as: **O\_RDONLY | O\_WRONLY** and will cause **open( )** to fail. If the **O\_CREAT** flag is used, then a mode argument is required. The mode argument may be specified in the same manner as in the **creat( )** system call. **close( )** **int close (int *fd*)**

**close ( )** frees the file descriptor ***fd***. If ***fd*** is the last file descriptor associated with a particular open file, the resources associated with that file are **deallocated**.

If **successful** **close** ( ) returns **zero**, otherwise it returns **–1**.

To close a channel, use the **close( )** system call. The prototype for the **close( )** system call is:

**int close(filedescriptor)**

**int filedescriptor;**

where **file\_descriptor** identifies a currently open channel. **close( )** fails if **file\_descriptor** does not identify a currently open channel.

**read( )**  **int read (int *fd*, char \**buf*, int *count*)**

**read ( )** copies ***count*** bytes from the file referenced by the file descriptor ***fd*** into the buffer (character array) ***buf***. **The bytes are read from the current file position, which is then updated accordingly**. **read** **( ) copies as many bytes from the file as it can, up to the number specified by *count*, and returns the number of bytes actually copied.** If a **read** **( )** is attempted after the **last byte has already been read**, it returns **0**, which indicates **end-of-file**. If **successful**, **read** **( )** returns the **number of bytes that it read**; otherwise, it returns **–1**.

**write( )** **int write (int *fd*, char \**buf*, int *count*)**

**write** **( )** copies ***count*** bytes from a buffer (character array) ***buf*** to the file referenced by the file descriptor ***fd***. **The bytes are written at the current file position, which is then updated accordingly**. If the **O\_APPEND** flag was set for ***fd***, the file position is set to the **end of the file** before each write. **write** **( )** **copies as many bytes from the buffer as it can, up to the number specified by *count*, and returns the number of bytes actually copied**. If the returned value is not ***count***, then the disk probably filled up and no space was left.

If **successful**, **write** **( )** returns the **number of bytes that it wrote**; otherwise, it returns **–1**.

The **read( )** system call does all input and the **write( )** system call does all output. When used together, they provide all the tools necessary to do input and output sequentially.

Both **read( )** and **write( )** take three arguments. Their prototypes are:

**int read(file\_descriptor, buffer\_pointer, transfer\_size)**

**int file\_descriptor;**

**char \*buffer\_pointer;**

**unsigned transfer\_size;**

**int write(file\_descriptor, buffer\_pointer, transfer\_size)**

**int file\_descriptor;**

**char \*buffer\_pointer;**

**unsigned transfer\_size;**

where **file\_descriptor** identifies the I/O channel, **buffer\_pointer** points to the area in memory where the data is stored for a **read( )** or where the data is taken for a **write( )**, and **transfer\_size** defines the maximum number of characters transferred between the file and the buffer. **read( )** and **write( )** return the number of bytes transferred.

There is no limit on **transfer\_size**, but you must make sure it's safe to copy **transfer\_size** bytes to or from the memory pointed to by **buffer\_pointer**. A **transfer\_size** of **1** is used to transfer a byte at a time for so-called "unbuffered" input/output. The most efficient value for **transfer\_size** is the size of the largest physical record the I/O channel is likely to have to handle. Therefore, 1K bytes -- the disk block size -- is the most efficient general-purpose buffer size for a standard file. However, if you are writing to a terminal, the transfer is best handled in lines ending with a newline.

Following is an example of how to use **open( ), read( ), write(** **)**:

/\* This program explains that how to use open(), read(), write() System calls \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <fcntl.h> /\* defines options flags \*/

#include <sys/types.h> /\* defines types used by sys/stat.h \*/

#include <sys/stat.h> /\* defines S\_IREAD & S\_IWRITE \*/

static char message[] = "Hello, world";

int main( )

{

    int fd, fd2;

    char buffer[80];

    /\*

    open datafile.dat for read/write access (O\_RDWR)

    create datafile.dat if it does not exist (O\_CREAT)

    return error if datafile already exists (O\_EXCL)

    permit read/write access to file (S\_IWRITE | S\_IREAD)

    \*/

    fd = open("datafile2.dat", O\_RDWR|O\_CREAT| O\_EXCL, S\_IREAD|S\_IWRITE);

    if (fd != -1)

    {

        printf("datafile2.dat opened for read/write access\n");

        write(fd, message, sizeof(message));

        close(fd);

        fd2 = open("datafile2.dat", O\_RDONLY, S\_IREAD);

        if(read(fd2, buffer, sizeof(message)))

            printf("%s: was written to datafile2.dat\n", buffer);

        else

            printf("\*\*\* error reading datafile2.dat \*\*\*\n");

        close(fd2);

    }

    else

        printf("\*\*\* datafile2.dat already exists \*\*\*\n");

    exit(0);

}

**long lseek (int *fd*, long *offset*, int *mode*)**

**lseek** **( )** allows you to **change a descriptor’s current file position**. ***fd*** is the file descriptor, ***offset*** is a long integer, and ***mode*** describes how ***offset*** should be interpreted. The three possible values of ***mode*** are defined in “/usr/include/sys/file.h”, and have the following meaning:

**VALUE** **MEANING**

**SEEK\_SET** offset is relative to the **start** of the file.

**SEEK\_CUR** offset is relative to the **current** file position. **SEEK\_END** offset is relative to the **end** of the file.

**lseek ( )** **fails if you try to move before the start of the file**. If **successful**, **lseek ( )** returns the **current file position**; otherwise, it returns **–1**.

**Assignments:**

**Problem#1 :**

Execute and understand the working of the example programs given above.

**Problem#2 :**

Implement your own version of acommand 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**

that displays the file called **file\_name** and reports with an appropriate message if the **file\_name** does not exist or can’t be opened. You can use open( ), read( ), write( ) and close( ) system calls.

**Problem#3 :**

Implement your own version of acommand to copy a file that should be almost same as the UNIX **cp** command that makes a copy of the file in the given name. Name your command as **copyfile** that should have the following syntax:

**copyfile source\_file\_name destination\_file\_name**

that copies **source\_file\_name** into **destination\_file\_name** and reports with an appropriate message if the **source\_file\_name** does not exist or can’t be opened. You can use open( ), read( ), write( ) and close( ) system calls.

**Problem#4 :**

Modify the above program to implement your own version of UNIX command called **textlines** similar to UNIX commands **head** or **tail**. Your command **textlines** should print from **m**th line to **n**th line from the given file. It should have the following syntax:

**textlines m n file\_name**

The command reports with an appropriate message if the **file\_name** does not exist or can’t be opened. You can use open( ), read( ), write( ), close( ) and lseek( ) system calls.