KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

Information and Computer Science Department

ICS 431 Operating Systems

Lab # 12

System Calls Related to File Manipulation

Objective:

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 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 <sys/types.h> /* defines types */
     #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 OR'ing 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:

Value	Meaning
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_NDELAY	Used wih communications devices
O_NONBLOCK	Used with communication devioces
O_APPEND	Open at end of file for writing
O_SYNC	If set this results in *write()* calls not returning until data has actually been written to dics.
O_CREAT	Create the file if it doesn't already exist
O_EXCL	If set and O_CREAT set will cause open() to fail if the file already exists
O_TRUNC	Truncate file size to zero if it already exists

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.

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(file_descriptor)
int file_descriptor;
```

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 <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;
   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("datafile.dat",O_RDWR | O_CREAT | O_EXCL,
S_IREAD | S_IWRITE);
        if (fd != -1)
        {
           printf("datafile.dat opened for read/write
access\n");
           write(fd, message, sizeof(message));
           close (fd);
           fd = open("datafile.dat",O_RDWR | O_CREAT | O_EXCL,
S_IREAD | S_IWRITE);
           if (read(fd, buffer, sizeof(message)) ==
sizeof(message))
              printf("\"%s\" was written to datafile.dat\n",
buffer);
           else
              printf("*** error reading datafile.dat ***\n");
           close (fd);
        }
        else
           printf("*** datafile.dat already exists ***\n");
        exit (0);
}
```

---

lseek()

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

lseek ( ) allows you to **change a descriptor's current file position**. ***fd*** is the file descriptor, ***offset*** is a long integer, and ***where*** describes how ***offset*** should be interpreted. The three possible values of ***where*** 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**.

---

stat()

int stat (char *name, struct stat *buf)

stat ( ) fills the buffer ***buf*** with **information about the file *name***. The **stat** structure is defined in "/usr/include/sys/stat.h". The **stat** structure contains the following members:

NAME	MEANING

st_dev	**the device number**
st_ino	**the inode number**
st_mode	**the permissions flag**
st_nlink	**the hard link count**
st_uid	**the user id**
st_gid	**the group id**
st_size	**the file size**
st_atime	**the last access time**
st_mtime	**the last modification time**
st_ctime	**the last status change time**

There are some predefined macros defined in "/usr/include/sys/stat.h" that takes **st_mode** as their argument and return true (1) for the following file types:

MACRO	**RETURNS TRUE FOR FILE TYPE**
S_ISDIR	**directory**
S_ISREG	**regular file**

stat returns **0** if **successful** and **–1** otherwise.

---

link()

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

link ( ) creates a new label *newpath* and **links it to the same file as the label *oldpath*. The hard link count of the associated file is incremented by one**. If ***oldpath*** and ***newpath*** resides on different physical devices, a hard link may not be made and **link ( )** fails.

link ( ) returns **–1** if **unsuccessful**, and **0** otherwise.

---

unlink()

int unlink (char **filename*)

unlink ( ) removes the hard link from the name ***filename*** to its file. If the ***filename*** is the **last link to the file**, the file's resources are **deallocated**.

If **successful**, **unlink ( )** returns **0**; otherwise, it returns **–1**.

---

getdents()

int getdents (int *fd*, struct direct *buf, int *structsize*)

getdents ( ) reads the directory file with descriptor ***fd*** from its current position and fills the structure pointed to by ***buf*** with the next entry. The structure **direct** is defined in "/usr/include/sys/dir.h", and contains the following fields:

NAME	MEANING
d_off	the **offset** of the **next directory entry**
d_fileno	the **inode number**
d_reclen	the **length of the directory entry structure**
d_namelen	the **length of the file name**
d_name	the **name** of the file

getdents ( ) returns the **length of the directory entry** when **successful**, **0** when the **last directory entry has already been read**, and **–1** in the case of an error.

Example Programs

A program that collects all information about a file like the link count, access and modification times, user and group ID, mode and size.

monitor.c

***** Before you execute the above program make sure that you have a file called 'file' in the same directory of the program. *****

A program that reads a name, which can be a filename or a directory name and if it is a directory it counts the number of files in the directory and prints the count as well as the names of all the files.

dir-count.c

***** When you execute the above program supply the name of file/directory as an argument on the command line. *****

Exercises

Note:

Lab Problems will be given during the lab based on material covered in this lab manual