Lecture 18
Systems Programming in C

A C program can invoke UNIX system calls directly. A system call can be defined as a request to the operating system to do something on behalf of the program. During the execution of a **system call**, the mode is change from **user mode** to **kernel mode (or system mode)** to allow the execution of the system call. The kernel, the core of the operating system program in fact has control over everything. All OS software is trusted and executed without any further verification. All other software needs to request kernel mode using specific system calls to create new processes and manage I/O. A process is a currently executing instance of a program. All programs by default execute in the user mode. A high level programmer does not have to worry about the mode change from user-mode to kernel-mode as it is handled by a predefined library of system calls.

Unlike processes in user mode, which can be replaced by another process at any time, a process in kernel mode cannot be arbitrarily replaced by another process. A process in kernel mode can only be suspended by an interrupt or exception. A C system call software instruction generates an OS interrupt commonly called the **operating system trap**. The system call interface handles these interruptions in a special way. The C library function passes a unique number corresponding to the system call to the kernel, so kernel can determine the specific system call user is invoking. After executing the kernel command the operating system trap is released and the system returns to user mode. Unix system calls are primarily used to **manage the file system** or **control processes** or to **provide communication** between multiple processes.
A subset of the system calls include

creat( ), open( ), close( ) -- managing I/O channels
read( ), write( ) – handling input and output operations
lseek( ) – for random access of files
link( ), unlink( ) – aliasing and removing files
stat( ) – getting file status
access( ), chmod( ), chown( ) – for access control

exec( ), fork( ), wait( ), exit( ) --- for process control
getuid( ) – for process ownership
getpid( ) -- for process ID
signal( ) , kill( ), alarm( ) – for process control
chdir( ) – for changing working directory

pipe( ) – for creating inter-process communication

System calls interface often change and therefore it is advisable to place system calls in subroutines so subroutines can be adjusted in the event of a system call interface change. When a system call causes an error, it returns -1 and store the error number in a variable called "**errno**" given in a header file called **/usr/include/errno.h**. When a system call returns an error, the function **perror** can be used to print a diagnostic message. If we call **perror(**), then it displays the argument string, a colon, and then the error message, as directed by "errno", followed by a newline.

```
if (unlink("text.txt")==-1){
     perror("");
}
If the file text.txt does not exists, unlink will return -1 and that in
return will cause the program to print the message “File does not
exists”
```

Managing the File System with Sys calls

File structure related system calls to manage the file system are quite common. Using file structure related system calls, we can **create, open, close, read, write, remove** and **alias**, **set permission**, **get file information** among other things. The arguments to these functions include either the relative or absolute path of the file or a descriptor that defines the IO channel for the file. A channel provides access to the file as an unformatted stream of bytes. We will now look at some of the file related functions provided as system calls.

UNIX System I/O Calls

The high level library functions given by <stdio.h> provide most common input and output operations. These high level functions are built on the top of low-level structures and calls provided by the operating system. In this section, we will look at some low level I/O facilities that will provide insight into how low level I/O facilities are handled and therefore may provide ways to use I/O in ways that are not provided by the stdio.h. In UNIX, I/O hardware devices are represented as special files. Input/Output to files or special files (such as terminal or printers) are handled the same way. UNIX also supports “pipes” a mechanism for input/output between processes. Although pipes and files are different I/O objects, both are supported by low level I/O mechanisms.

A file can be open using the open system call as follows.

#include <sys/file.h> // can be replaced by <fcntl.h>
int open(char* filename, int access, int mode);

The above code opens the filename for **reading or writing** as specified by the **access** and returns an integer descriptor for that file. Descriptor is an integer (usually less than 16) that indicates a table entry for the file reference for the current process. File name can be given as full path name, relative path name, or simple file name. If the file does not exist, then open creates the file with the given name. Let us take a detail look at the arguments to open.

filename : A string that represents the absolute, relative or filename of the file
access : An integer code describing the access (see below for details)
mode : The file protection mode usually given by 9 bits indicating rwx permission

The access codes are given by
O_RDONLY -- opens file for read only
O_WRONLY – opens file for write only
O_RDWR – opens file for reading and writing
O_NDELAY – prevents possible blocking
O_APPEND --- opens the file for appending
O_CREAT -- creates the file if it does not exists
O_TRUNC -- truncates the size to zero
O_EXCL – produces an error if the O_CREAT bit is on and file exists

If the open call fails, a -1 is returned; otherwise a descriptor is returned. For example, to open a file for read and truncates the size to zero we could use,

open("filename", O_RDONLY | O_TRUNC, 0);

We assume that the file exists and note that zero can be used for protection mode. For opening files, the third argument can always be left at 0.

Create System Call

A file can be created using the creat function as given by the following prototype.

int creat(char* filename, mode)

The mode is specified as an octal number. For example, 0666 indicates that rw access for USER, GROUP and ALL for the file. If the file exists, the creat is ignored and prior content and rights are maintained. The library

/usr/include/sys/stat.h

provides following constants that can be used to set permissions.

S_IREAD --- read permission for the owner
S_IWRITE --- write permission for the owner
S_IEXEC --- execute/search permission for the owner
S_IRWXU --- read, write, execute permission for the user

S_IRGRP – read for group
S_IWGRP – write for group
S_IXGRP – execute for group
S_IRWXG – read, write, execute for the group

S_IROTH --- read for others
S_IWOTH – write for others
S_IXOTH -- execute for others
S_IRWXO – read , write , execute for others

For example, to create a file with read and write access only to user, we can do the following.

creat("myfile", S_IREAD | S_IWRITE);

Reading and Writing to Files

Reading and writing a file is normally sequential. For each open file, a current position points to the next byte to be read or written. The current position can be movable for an actual file, but not for stdin when connected to a keyboard.

Read System Call

```
#include       <sys/types.h> // or #include <unistd.h>
size_t  read(int fd, char *buffer , size_t bytes);
```

fd is the file descriptor, *buffer* is address of a memory area into which the data is read and *bytes* is the maximum amount of data to read from the stream. The return value is the actual amount of data read from the file. The pointer is incremented by the amount of data read. Bytes should not exceed the size of the buffer.

Write System Call

The write system call is used to write data to a file or other object identified by a file descriptor. The prototype is

```
#include       <sys/types.h>
size_t  write(int fd, char *buffer, size_t bytes);
```

fd is the file descriptor, *buffer* is the address of the area of memory that data is to be written out, *bytes* is the amount of data to copy. The return value is the actual amount of data written, if this differs from *bytes* then something may be wrong.

Example: Consider the C high level function readline(char [], int) that reads a line from stdin and store the line in a character array. This function can now be rewritten using low level read as follows

```
int readline(char s[], int size){
      char* tmp = s;
      while (--size>0 && read(0,tmp,1)!=0 && *tmp++ != '\n');
      *tmp = '\0';
      return (tmp-s);

}
```

Close System Call

The close system call is used to close files. The prototype is

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

When a process terminates, all the files associated with the process are closed. But it is always a good idea to close a file as they do consume resources and systems impose limits on the number of files a process can keep open.

lseek System Call

Whenever a read() or write() operation is performed on a file, the position in the file at which reading or writing starts is determined by the current value of the **read/write pointer**. The value of the read/write pointer is often called the **offset**. It is always measured in bytes from the start of the file. The lseek() system call allows programs to manipulate this directly by providing the facility for **direct access** to any part of the file. In other words, the lseek allows random access to any byte of the file. It has three parameters and the prototype is

```
#include       <sys/types.h>
#include       <unistd.h>
long           lseek(int fd,int offset,int origin)
```

```
origin         position
 0      beginning of the file
 1      Current position
 2      End of the file
```

```
Call                   Meaning
lseek(fd,0,0)          places the current position at the first byte
lseek(fd,0,2)          places the current position at EOF
lseek(fd,-10,1)        Backs up the current position by 10 bytes
```

Other System Level Operations

Other system level operations include creating a file using

creat(filename, mode);

The mode can be selected from the following octal bit patterns

Octal Bit Pattern	Meaning
00400	Read by owner
00200	Write by Owner
00100	Execute or search by owner
00070	Read, Write, Execute(search) by group
00007	Read, Write, Execute(search) by others

Other system calls include link a way to give alternative names to a file (aliases), and unlink a way to remove an alias to a file. The prototypes for the link and unlink are

int link (char* file1, char* file2);
int unlink(char* name);

The link( ) function creates an alias name file2 for file1, that exists in the current directory. Use of unlink with the original file name will remove the file.

Creating and removing Directories

Directories can be created and removed using mkdir and rmdir function calls. The function prototypes are

int mkdir(char* name, int mode);
int rmdir(char* name);

returns 0 or 1 for success or failure. For example, creating a new directory called "newfiles" with only the READ access for the user we can do the following.

mkdir("newfiles", 00400);

Later you can remove this directory by calling

rmdir("newfiles");

Caution: Be very careful about removing directories. These are system calls that are executed w/o further confirmation.

Accessing Directories

A UNIX directory contains a set of files that can be accessed using the sys/dir.h library. We include the library with

#include <sys/dir.h>

And the function

DIR *opendir(char* dir_name)

Opens a directory given by dir_name and provides a pointer to access files within the directory. The open DIR stream can be used to access a struct that contains the file information. The function

struct dirent *readdir(DIR* dp)

returns a pointer to the next entry in the directory. A NULL pointer is returned when the end of the directory is reached. The struct direct has the following format.

```
struct dirent {
  u-long d_ino;                        /* i-node number for the dir entry */
  u_short d_reclen;                    /* length of this record */
  u_short d_namelen ;                  /* length of the string in d_name */
  char  d_name[MAXNAMLEN+1] ;   /* directory name */
};
```

Using system libraries, we can write our own "find" function for example. The following function, **search (char* file, char* dir)** returns 0 if the file is found in the directory and returns 1 otherwise.

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

int search (char* file, char* dir){
   DIR  *dirptr=opendir(dir);
   struct dirent *entry = readdir(dirptr);
   while (entry != NULL) {
      if ( entry->d_name == strlen(file) && (strcmp(entry->d_name, file) == 0)
           {
                return 0;
           }
      entry = readdir(dirptr);
   }
   return 1;
}
```

Accessing File Status

Status of a file such as file type, protection mode, time when the file was last modified can be accessed using stat and fstat functions. The prototypes of stat and fstat functions are

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

stat and fstat functions are equivalent except that former takes the name of a file, while the latter takes a file descriptor. For example, we can get the status of a file as follows.

```
struct stat buf;  // defines a struct stat to hold file information
stat("filename", &buf) ;  // now the file information is placed in the buf
```

The status of the file that is retrieved and placed in the buf has many information about the file. For example, the stat structure contains useful information such as

st_atime --- Last access time
st_mtime --- last modify time
st_ctime --- Last status change time
st_size --- total size of file
st_uid – user ID of owner
st_mode – file status (directory or not)

Example

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

struct stat statbuf;
char dirpath[256];
getcwd(dirpath,256);
DIR *dir = opendir(dirpath);
struct dirent *dp;
for (dp=readdir(dir); dp != NULL ; dp=readdir(dir)){
      stat(dp->d_name, &statbuf);
      printf("the file name is %s \n", dp->d_name);
      printf("dir = %d\n", S_ISDIR(statbuf.st_mode));
      printf("file size is %ld in bytes \n", statbuf.st_size);
      printf("last modified time is %ld in seconds \n",
      statbuf.st_mtime);
      printf("last access time is %ld in seconds \n",
      statbuf.st_atime);
      printf("The device containing the file is %d\n", statbuf.st_dev);
      printf("File serial number is %d\n\n", statbuf.st_ino);
}
```

More can be found at
http://www.opengroup.org/onlinepubs/000095399/functions/stat.html

Working Directory

The working directory of the program can be found using getcwd. The prototype is given by

#include <unistd.h>
char* getcwd(char * dirname, int);

This copies the full path name of the current working directory into the dirname and returns a pointer to it. The description can be found

EXERCISES

1. Write a function foo(int fd, char* buf, int b_size, int n, int skip) that reads to buf from file with file descriptor fd, n blocks of size b_size each. The last argument specifies how many bytes to skip after reading each block. Return -1 if the operation is unsuccessful. Else return total number of bytes read.

2. Write a program to read all txt files (that is files that ends with .txt) in the current directory and merge them all to one txt file and returns a file descriptor for the new file.

3. Write a program that a string as an argument and return all the files that begins with that name in the current directory. For example > ./a.out foo   will return all file names that begins with foo.