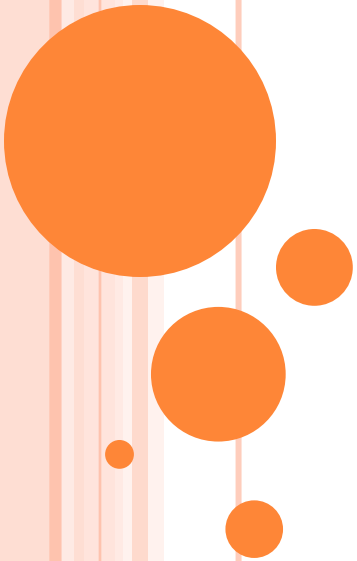


UNIX SYSTEM CALLS
File system calls

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.

File Structure Related System Calls (Cont'd)

- To a process then, all input and output operations are synchronous and unbuffered.
- 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

- Each UNIX process has 20 file descriptors at it disposal, numbered 0 through 19.
- The first three are already opened when the process begins
  - 0: The standard input
  - 1: The standard output
  - 2: The standard error output
- When the parent process forks a process, the child process inherits the file descriptors of the parent.

CREAT() SYSTEM CALL

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

```
int creat(file_name, mode)
char *file_name;
int mode;
```

CREAT() SYSTEM CALL (CONT'D)

- 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.

CREAT() SYSTEM CALL (CONT'D)

- The following is a sample of the manifest constants for the mode argument as defined in /usr/include/sys/stat.h:

```
#define S_IRWXU 0000700     /* -rwx------ */
#define S_IREAD 0000400     /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200    /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100     /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070     /* ----rwx--- */
#define S_IRGRP 0000040     /* read permission, group */
#define S_IWGRP 0000020     /* write    "         "   */
#define S_IXGRP 0000010     /* execute/search "   "   */
#define S_IRWXO 0000007     /* -------rwx */
#define S_IROTH 0000004     /* read permission, other */
#define S_IWOTH 0000002     /* write    "         "   */
#define S_IXOTH 0000001     /* execute/search "   "   */
```

OPEN() SYSTEM CALL

- The prototype for the open() system call is:

```
#include <fcntl.h>
int open(file_name, option_flags [, mode])
char *file_name;
int option_flags, mode;
```

OPEN() SYSTEM CALL (CONT'D)

- 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_NDELAY 04       /* Non-blocking I/O */
#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).

OPEN() SYSTEM CALL (CONT'D)

- The argument 'flags' defines the way in which the file needs to be opened ie in read-only, write-only or read-write mode. So corresponding to this there are three flags O_RDONLY, O_WRONLY, or O_RDWR to choose from. So, one of these three flags is mandatory. Other than this, there are some other flags that can be or'ed together and passed as part of this argument. These flags are :

  From the man page of open() ...

  O_APPEND : The file is opened in append mode.

  O_CREAT : If the file does not exist it will be created.

OPEN() SYSTEM CALL (CONT'D)

Other than the default standard input, output and error, you must explicitly open files in order to read or write them. There are two system calls for this, `open` and `creat` [sic].

`open` is rather like the `fopen`, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`. `open` returns `-1` if any error occurs.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

As with `fopen`, the `name` argument is a character string containing the filename. The second argument, `flags`, is an `int` that specifies how the file is to be opened; the main values are

`O_RDONLY`	open for reading only
`O_WRONLY`	open for writing only
`O_RDWR`	open for both reading and writing

These constants are defined in `<fcntl.h>` on System V UNIX systems, and in `<sys/file.h>` on Berkeley (BSD) versions.

To open an existing file for reading,

```
fd = open(name, O_RDONLY,0);
```

The `perms` argument is always zero for the uses of `open` that we will discuss.

OPEN() SYSTEM CALL (CONT'D)

It is an error to try to open a file that does not exist. The system call `creat` is provided to create new files, or to re-write old ones.

```
int creat(char *name, int perms);
```

```
fd = creat(name, perms);
```

returns a file descriptor if it was able to create the file, and `-1` if not. If the file already exists, `creat` will truncate it to zero length, thereby discarding its previous contents; it is not an error to `creat` a file that already exists.

If the file does not already exist, `creat` creates it with the permissions specified by the `perms` argument. In the UNIX file system, there are nine bits of permission information associated with a file that control read, write and execute access for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is convenient for specifying the permissions. For example, `0775` specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

CLOSE() SYSTEM CALL

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

```
int close(file_descriptor)
int file_descriptor;
```

OPEN() SYSTEM CALL (CONT'D)

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

int main(int argc, char *argv[])
{
    int fd;

    if(2 != argc)
    {
        printf("\n Usage :  \n");
        return 1;
    }

    errno = 0;
    fd = open(argv[1],O_RDONLY);

    if(-1 == fd)
    {
        printf("\n open() failed with error [%s]\n",strerror(errno));
        return 1;
    }
    else
    {
        printf("\n Open() Successful\n");
        /* open() succeeded, now one can do read operations
           on the file opened since we opened it in read-only
           mode. Also once done with processing, the file needs
           to be closed. Closing a file can be achieved using
           close() function. */
    }
```

READ() & WRITE() SYSTEM CALLS

- The read() system call does all input and the write() system call does all output.

```
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;
```

READ() & WRITE() SYSTEM CALLS

Required Include Files

```
#include <unistd.h>
```

Function Definition

```
size_t read(int fildes, void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to read the input. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A character array where the read content will be stored.
size_t nbytes	The number of bytes to read before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
return value	Returns the number of bytes that were read. If value is negative, then the system call returned an error.

Code Snippet

```
#include <unistd.h>

int main()
{
    char data[128];

    if(read(0, data, 128) < 0)
     write(2, "An error occurred in the read.\n", 31);
    exit(0);
}
```

READ() & WRITE() SYSTEM CALLS

Required Include Files

```
#include <unistd.h>
```

Function Definition

```
size_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A null terminated character string of the content to write.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

Code Snippet

Example using standard file descriptors:

```
#include <unistd.h>

int main(void)
{
    if (write(1, "This will be output to standard out\n", 36) != 36) {
        write(2, "There was an error writing to standard out\n", 44);
        return -1;
    }
    return 0;
}
```

READ() & WRITE() SYSTEM CALLS

Example using a file descriptor:

```
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    int filedesc = open("testfile.txt", O_WRONLY | O_APPEND);

    if (filedesc < 0) {
        return -1;
    }

    if (write(filedesc, "This will be output to testfile.txt\n", 36) != 36) {
        write(2, "There was an error writing to testfile.txt\n", 43);
        return -1;
    }

    return 0;
}
```

What is in an inode?

```
% ls -lai | tail -7
total 132
      2 drwxr-xr-x  24 root root   4096 Feb 26 13:31 .
      2 drwxr-xr-x  24 root root   4096 Feb 26 13:31 ..
2637825 drwxr-xr-x   2 root root   4096 Jan 14 19:02 bin
 196609 drwxr-xr-x   3 root root   4096 Feb 24 10:41 boot
      3 drwxr-xr-x  16 root root   4460 Mar  5 09:35 dev
 983041 drwxr-xr-x 206 root root  12288 Mar  5 07:45 etc
      2 drwxr-xr-x  14 root root   4096 Dec 29 09:24 home
```

What is in an inode?

Before I said the data blocks contain the contents of the file. The inode contains the following pieces of information

- Mode/permission (protection)
- Owner ID
- Group ID
- Size of file
- Number of hard links to the file
- Time last accessed
- Time last modified
- Time inode last modified

What is in an inode?

Notice something missing? Where is the NAME of the file. Or the Path? It's NOT in the inode. It's NOT in the data blocks. It's _in_ the directory. That's right. A "file" is really in three (or more) places on the disk.

You see, the directory is just a table that contains the filenames in the directory, and the matching inode. Think of it as a table, and the first two entries are always "." and ".." The first points to the inode of the current directory, and the second points to the inode of the parent

STAT() SYSTEM CALLS

stat is a system call that is used to determine information about a file based on its file path.

Required Include Files

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

Function Definition

```
int stat(const char *path, struct stat *buf);
```

Field	Description
const char *path	The file descriptor of the file that is being inquired.
struct stat *buf	A structure where data about the file will be stored. A detailed look at all of the fields in this structure can be found in the struct stat page.
return value	Returns a negative value on failure.

STAT() SYSTEM CALLS

An example of code that uses the stat() system call is below.

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

int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;

    struct stat fileStat;
    if(stat(argv[1],&fileStat) < 0)
        return 1;

    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n",fileStat.st_size);
    printf("Number of Links: \t%d\n",fileStat.st_nlink);
    printf("File inode: \t\t%d\n",fileStat.st_ino);

    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");

    printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");

    return 0;
}
```

STAT() SYSTEM CALLS

The output of this program is shown in the following set of commands:

Information on the current files in the directory:

```
$ ls -l
total 16
-rwxr-xr-x 1 stargazer stargazer   36 2008-05-06 20:50 testfile.sh
-rwxr-xr-x 1 stargazer stargazer 7780 2008-05-07 12:36 testProgram
-rw-r--r-- 1 stargazer stargazer 1229 2008-05-07 12:04 testProgram.c
```

Running the program with the file `testfile.sh`

```
$ ./testProgram testfile.sh

Information for testfile.sh
---------------------------
File Size:              36 bytes
Number of Links:        1
File inode:             180055
File Permissions:       -rwxr-xr-x

The file is not a symbolic link
```

Running the program with the files `testProgram.c`

```
$ ./testProgram testProgram.c

Information for testProgram.c
---------------------------
File Size:              1229 bytes
Number of Links:        1
File inode:             295487
File Permissions:       -rw-r--r--

The file is not a symbolic link
```

STAT() SYSTEM CALLS

Running the program with the directory /home/stargazer

```
$ ./testProgram /home/stargazer

Information for /home/stargazer
---------------------------
File Size:                4096 bytes
Number of Links:          61
File inode:               32706
File Permissions:         drwxr-xr-x

The file is not a symbolic link
```

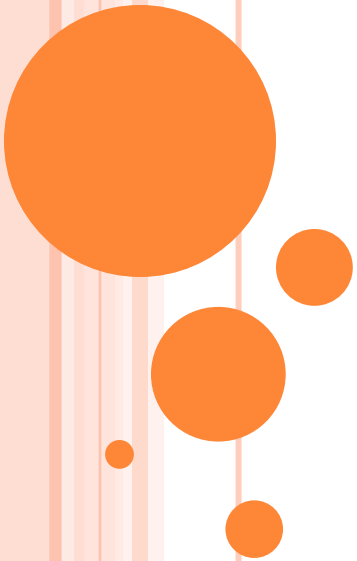
STAT() SYSTEM CALLS

`struct stat` is a system struct that is defined to store information about files. It is used in several system calls, including fstat, lstat, and stat.

Fields

The `struct stat` is simply a structure with the following fields:

Field Name	Description
st_mode	The current permissions on the file.
st_ino	The inode for the file (note that this number is unique to all files and directories on a Linux System.
st_dev	The device that the file currently resides on.
st_uid	The User ID for the file.
st_gid	The Group ID for the file.
st_atime	The most recent time that the file was accessed.
st_ctime	The most recent time that the file's permissions were changed.
st_mtime	The most recent time that the file's contents were modified.
st_nlink	The number of links that there are to this file.
st_size	



UNIX SYSTEM CALLS
Directory system calls

DIRECTORIES: MKDIR 1/2

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

int mkdir(const char *pathname, mode_t
mode);
```

- Attempts to create a directory.
- To create a directory
  - Specify directory name.
  - Specify permission. E.g. 0740;

DIRECTORIES: MKDIR 2/2

- mode_t perms = 0740; // Who can access?
- if (mkdir (“test_dir”, perms)) == -1) {
- 	perror(“failed to create directory”);
- 	return -1;
- }
- What does return value “0” mean?

DIRECTORIES: RMDIR

#include <unistd.h>

```
int rmdir(const char *pathname);
```

- Deletes a directory which must be empty.
- What errors can happen?
- You must have a permission to delete a directory.

DIRECTORIES: OPENDIR 1/2

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

DIR * opendir(const char * name);
```

- Opens a directory stream corresponding to the directory name.
- What errors can happen?

DIRECTORIES: OPENDIR 2/2

- DIR * dir; // So where is the DIR defined?
- dir = opendir (“test_dir”);
- if(dir == NULL) {
- perror (“Failed to open directory”);
- }

DIRECTORIES: READDIR 1/2

#include <dirent.h>

```
struct dirent * readdir(DIR * dirp);
```

- Returns a pointer to directory structure representing the net directory entry.
- What is `struct dirent`?

DIRECTORIES: READDIR 2/2

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all file system types */
    char           d_name[256]; /* filename */
};
```

DIRECTORIES: CLOSEDIR

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

int closedir(DIR * dirp);
```

- Closes the directory stream associated with dirp.
- What errors can happen?

Examples:

Get a list of files contained in the directory `/home/fred`:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main( void )
{
    DIR* dirp;
    struct dirent* direntp;

    dirp = opendir( "/home/fred" );
    if( dirp == NULL ) {
        perror( "can't open /home/fred" );
    } else {
        for(;;) {
            direntp = readdir( dirp );
            if( direntp == NULL ) break;

            printf( "%s\n", direntp->d_name );
        }

        closedir( dirp );
    }
```