Reference: [https://www.man7.org/linux/man-pages/man2/stat.2.html (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/stat.2.html)

To retrieve information about files, we can use the stat system call.

Copy and paste the following code into "stat\_example1.c".

#include <sys/types.h>  
#include <sys/stat.h>  
#include <time.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/sysmacros.h>  
#include <unistd.h>  
  
int main(int argc, char \*argv[])  
{  
    struct stat sb;  
  
    if (argc != 2)  
    {  
 // fprintf can print to files. In this case, it's printing to stderr  
       fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);   
        exit(EXIT\_FAILURE);  
    }  
  
   if (**stat(argv[1], &sb)** == -1)  
    {  
       perror("stat"); // perror is used to print non-formatted strings to stderr  
       exit(EXIT\_FAILURE); // exit is actually a system call that will cause our program to end.  
 // EXIT\_FAILURE is a constant defined in stdlib.h  
    }  
  
    printf("I-node number:            %ld\n", (long) sb.st\_ino);  
    printf("File size:                %lld bytes\n", (long long) sb.st\_size);  
  
   exit(EXIT\_SUCCESS); // EXIT\_SUCCESS is a constant (0) defined in stdlib.h  
}

Test the program above by compiling

gcc stat\_example1.c -o stat\_example1

Then run

./stat\_example1 <filename>

**<filename>** should be replaced with the name of an actual file. Try the following

./stat\_example1 .

./stat\_example1 stat\_example1.c

./stat\_example1 stat\_example1

./stat\_example1 /dev/null

In the code, **stat(argv[1], &sb)**calls the **stat** function and passes the 1st command line argument to stat as well as **&sb**. **&sb** is the memory address of the **struct stat sb**. The **stat** function set the values in the struct at the given memory address so that they contain information about file with filename given by **argv[1]**.

2 file-io

Reference: [https://www.man7.org/linux/man-pages/man2/open.2.html (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/open.2.html)

File input/output (io) requires system calls. There are functions like fgets that make file io more convenient. Here, we will use system calls to perform file io. The system calls are

* open - opens a file,
* read - reads bytes from file,
* write - writes bytes to file, and
* close - closes the opened file.

Before running the code below, you need a test file called "infile". Use the following command to create this file.

head -c 100 /dev/random > infile

This command prints the first 100 bytes of /dev/random. /dev/random generates random bytes.

Verify that "infile" was created and contains 100 bytes.

ls -lh infile

Now that we have a test input file, we can run the following code.

Copy and paste the following code into "file\_io.c". Review, compile and run the code.

#include<stdio.h>  
#include<string.h>  
#include<unistd.h>  
#include<fcntl.h>  
  
int main ()  
{  
    int in\_fd, out\_fd;  
    int bytes;  
  
    char buffer[5];  
  
    // assume infile is already created  
    in\_fd = **open**("infile", O\_RDWR);  
  
    out\_fd = **open**("outfile", O\_RDWR | O\_CREAT, S\_IRUSR | S\_IWUSR);  
  
    bytes = **read**(in\_fd, buffer, 5); // read 5 bytes into buffer  
    printf("Bytes read: %d\n", bytes);  
  
    bytes = **write**(out\_fd, buffer, 5);  
    printf("Bytes written: %d\n", bytes);  
  
    **close**(in\_fd);  
    **close**(out\_fd);  
  
    return 0;  
}

Let's look at what is going on in the code.

2.1 File Descriptors

First, the **open** function returns an **int**. The int that is returned is called a "file descriptor". This is a number that is used to identify our open file. Here's what the man pages for open say about the file descriptor.

The return value of **open**() is a file descriptor, a small,

nonnegative integer that is used in subsequent system calls

([read(2) (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/read.2.html), [write(2) (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/write.2.html), [lseek(2) (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/lseek.2.html), [fcntl(2) (Links to an external site.)](https://www.man7.org/linux/man-pages/man2/fcntl.2.html), etc.) to refer to the

open file. The file descriptor returned by a successful call

will be the lowest-numbered file descriptor not currently open

for the process.

2.2 File Access Flags

Next, note that we pass **O\_RDWR** to the first call to the open function. **O\_RDWR** is a "flag" that specifies that we want to ***open the file with read and write access***.

Flags are a common concept in C. They are really just numbers. For example, **O\_RDWR** is **2**. It's helpful to represent them in binary to understand how they work. Here are the common flags for file access and corresponding binary representation of each.

|  |  |  |
| --- | --- | --- |
| **Flag** | **Binary Representation** | **Meaning** |
| O\_RDONLY | 000000000000 | Open file as read only. |
| O\_WRONLY | 000000000001 | Open file as write only. |
| O\_RDWR | 000000000010 | Open file with read and write access. |
| O\_APPEND | 010000000000 | Open file and append when writing. |
| O\_CREAT | 000001000000 | Open file and create the file if it does not exist. |

It is common to combine flags using the **bitwise or operator.** For example, if you want to open a file with write only access and append to the existing file if there is one, you can combine O\_WRONLY and O\_APPEND like this **O\_WRONLY | O\_APPEND.**

**O\_WRONLY | O\_APPEND** is the "bitwise or" of O\_WRONLY and O\_APPEND. It computes the "or" operator on each bit of the numbers O\_WRONLY and O\_APPEND.

So, **O\_WRONLY | O\_APPEND =** 010000000001

Note that a bit in some position of**O\_WRONLY | O\_APPEND** is 1 if the bit in the same position in **O\_WRONLY**is 1**or**the bit in the same position in**O\_APPEND** is 1. If the bit in some position in **O\_WRONLY | O\_APPEND** is 0, then the bits in the corresponding positions in **O\_WRONLY and O\_APPEND** are 0.

I.e., the bits in **O\_WRONLY | O\_APPEND**are computed using the following table for each bit.

|  |  |  |
| --- | --- | --- |
| **bit1** | **bit2** | **bit1 | bit2** |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

There's also a **bitwise and** ( **&** ) operator that is computed bitwise for two numbers using the following table.

|  |  |  |
| --- | --- | --- |
| **bit1** | **bit2** | **bit1 & bit2** |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

**Bitwise and** is good for extracting information from a number. For example, if we have the number **num = 010000000001** and know that the number corresponds to file access flags, we can check if the number encodes O\_APPEND access by doing

**num & O\_APPEND == O\_APPEND**

Computing **num & O\_APPEND** gives 010000000000, which equals **O\_APPEND.**

2.3 File Permission Flags

Note that the second call to **open** looks like **open**("outfile", O\_RDWR | O\_CREAT, S\_IRUSR | S\_IWUSR);

So, from O\_RDWR | O\_CREAT, we see that "outfile" will be opened with read and write access and will be created if it does not exist.

When a file is created, it will be created with user, group, and other permissions for read, write, and execute. To set these permissions, we can use permission flags. In the call to open above, **S\_IRUSR | S\_IWUSR** set the permissions. Here are the flags that you can use for permissions.

|  |  |  |
| --- | --- | --- |
| **Flag** | **Binary Representation** | **Meaning** |
| S\_IRUSR | 100000000 | Set read permision for the user. |
| S\_IWUSR | 010000000 | Set write permision for the user. |
| S\_IXUSR | 001000000 | Set execute permision for the user. |
| S\_IRGRP | 000100000 | Set read permision for the group. |
| S\_IWGRP | 000010000 | Set write permision for the group. |
| S\_IXGRP | 000001000 | Set execute permision for the group. |
| S\_IROTH | 000000100 | Set read permision for others. |
| S\_IWOTH | 000000010 | Set write permision for others. |
| S\_IXOTH | 000000001 | Set execute permision for others. |

We can combine permissions using bitwise or. In the example above, **S\_IRUSR | S\_IWUSR** is passed to open so that if open ends up creating a new file, the user is given read and write access to the file.

3 More file-info

Reference: [https://man7.org/linux/man-pages/man7/inode.7.html (Links to an external site.)](https://man7.org/linux/man-pages/man7/inode.7.html)

Using flags like we saw in the previous section is common in C programming. Often, C functions return a number that encodes data in the bits of the number. These bits must be extracted. The following code is an extended version of stat\_example1.c.

Paste the following code in "stat\_example2.c". Review, compile and run the code.

#include <sys/types.h>  
#include <sys/stat.h>  
#include <time.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/sysmacros.h>  
#include <unistd.h>  
  
int main(int argc, char \*argv[])  
{  
    struct stat sb;  
  
    if (argc != 2)  
    {  
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);  
        exit(EXIT\_FAILURE);  
    }  
  
  if (**stat(argv[1], &sb)** == -1)  
    {  
      perror("stat");  
        exit(EXIT\_FAILURE);  
    }  
  
    **printf("File type:                ");**  
  
**switch (sb.st\_mode & S\_IFMT)**  
**{**  
**case S\_IFBLK:  printf("block device\n");            break;**  
**case S\_IFCHR:  printf("character device\n");        break;**  
**case S\_IFDIR:  printf("directory\n");               break;**  
**case S\_IFIFO:  printf("FIFO/pipe\n");               break;**  
**case S\_IFLNK:  printf("symlink\n");                 break;**  
**case S\_IFREG:  printf("regular file\n");            break;**  
**case S\_IFSOCK: printf("socket\n");                  break;**  
**default:       printf("unknown?\n");                break;**  
**}**  
  
    printf("I-node number:            %ld\n", (long) sb.st\_ino);  
    printf("File size:                %lld bytes\n", (long long) sb.st\_size);  
  
    **printf("\nFile access: ");**  
**if (sb.st\_mode & S\_IRUSR)**  
**printf("read ");**  
**if (sb.st\_mode & S\_IWUSR)**  
**printf("write ");**  
**if (sb.st\_mode & S\_IXUSR)**  
**printf("execute");**  
**printf("\n");**  
  
  
    exit(EXIT\_SUCCESS);  
}

Note in the code above that the bitwise and ( & ) is being used to extract information from **sb.st\_mode.**For example, to check if the user has read access, **sb.st\_mode & S\_IRUSR**is used.

Note that the type of file is also being extracted from**sb.st\_mode.**This is done by checking **sb.st\_mode & S\_IFMT.**

According to the man pages

The following mask values are defined for the file type:

**S\_IFMT** 0170000 bit mask for the file type bit field

**S\_IFSOCK** 0140000 socket

**S\_IFLNK** 0120000 symbolic link

**S\_IFREG** 0100000 regular file

**S\_IFBLK** 0060000 block device

**S\_IFDIR** 0040000 directory

**S\_IFCHR** 0020000 character device

**S\_IFIFO** 0010000 FIFO

A "mask" is a number that can be used to extract particular bits from a number. Extraction is done with bitwise and ( & ). In the man pages, masks and flags are given as hexadecimal numbers for brevity.

So, **sb.st\_mode & S\_IFMT** is used to compute a number. That number corresponds to **S\_IFSOCK, S\_IFLNK,** **S\_IFREG,** **S\_IFBLK, S\_IFDIR,** **S\_IFCHR, or** **S\_IFIFO.**In the code example above, a switch statement is used to determine which file type we have.