UNIX System Programming

Files and their properties

System Programming

System programming involves invoking calls to kernel routines to access or manipulate portions of the system.

Typically we subconsciously write code that assumes that all is rosy and all of our system calls will work perfectly.

But what if they don't? Why would we get errors?

- The resource we are trying to access no longer exists.
- The system we are running our code on is configured differently than our development system.
- The credentials under which the code is now running have different access rights to system resources.
- The list is almost endless...

Code Robustness

Always check the return status of your system function calls. If you don't, you have no idea why your code isn't working.

Poor coding practices are a main contributor to software vulnerabilities !

Typically, a system call will return a status of -1 on failure.

Since this isn't very descriptive of what the error might be, *NIX also sets an additional flag ***`errno`*** to represent the reason for why the call failed.

- This means a quick check of the status can simplify your code if the call was successful.
- If the call failed, you can then dig further into the issue and handle each situation in a unique way.

errno and perror()

Unix provides a globally accessible integer variable that contains an error code number

Error variable: errno – defined in errno.h

perror(" a string "): a library routine

- Uses current value of errno

```
more /usr/include/asm/*errno.h

#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#define EPERM           1   /* Operation not permitted */
#define ENOENT          2   /* No such file or directory */
#define ESRCH           3   /* No such process */
#define EINTR           4   /* Interrupted system call */
#define EIO             5   /* I/O error */
#define ENXIO           6   /* No such device or address */
```

errno and perror()

```
// file foo.c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main()
  {

  int fd;

  /* open file "data" for reading */
  if( fd = open( "nosuchfile", O_RDONLY ) == -1 )
        {
        fprintf( stderr, "Error %d\n", errno );// perror uses value of errno
        perror( "Opening my data file" );
        }
  } /* end main */
```

perror gives the reason why it failed
with the message of why it fialed after the "opening my data file"

```
{dkl:57} gcc foo.c -o foo
{dkl:58} ./foo
Error 2
Opening my data file: No such file or directory
```

Wrappers.

Wrappers are pieces of “envelope” code.

Typically they provide a consistent interface which masks the different O/S implementations of system calls.

They allows you to create a standard handling procedure for your code.

Instead of replicating line after line of code to handle failure return codes, your wrapper routine could manage all of this for you.

Wrappers: handle errors on open()

```
int my_fopen(int *fd, char *UserFile)
{
 *fd = open( UserFile , O_RDONLY ) ;   // open file "data" for reading

  if(  *fd  == -1  )                   // NOTE: Assuming read_only access for this example. Proper wrapper
    {                                  // would pass the file access mode through parameters.
     switch (errno)
       {
          case ENOENT:    // No such file
               printf("The requested file (%s) doesn't exist.
                        Please supply an existing file when you re-run this code.\n\n", UserFile);
               break;
         case EACCES:     // No access to the requested file
              printf("You do not have READ access to the file %s.\n\n", UserFile);
              break;
...
         case EROFS:      // READ ONLY FILE SYSTEM
              printf("The file you are trying to open for write access is on a read-only file system.\n\n");
              break;
         default:
              fprintf( stderr, "The open operation on %s generated an unhandled error: %d\n", UserFile, errno );
              perror( "(my_open) The Error was" );
         } // switch errno
       return  FALSE;
    }
  else
    {
       return TRUE;
    }
} // end of my_open
```

Wrappers are pieces of "envelope" code.
Typically they provide a consistent interface which masks the different O/S implementations of system calls.
They allows you to create a standard handling procedure for your code.
Instead of replicating line after line of code to handle failure return codes, your wrapper routine could manage all of this for you

```c
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE  1
#define FALSE 0

int  my_fopen(int *fd, char *UserFile);   // prototype for the function defined later.


int  main(int argc, char *argv[])
{
  int fd;
  char line[4096];                // each line in the input file can have a max length of 4096 characters.
  char UserFile[64];              // name of the control file retrieved from the command line

  strncpy(UserFile,argv[1],64);  // only take the first 64 characters (would normally check for /, . and .. )

  if( my_fopen( &fd, UserFile) )
   {
     printf("Processing file: %s\n", UserFile);

      // this is where the code to read the file should be...
      read(fd, line, 1024);      //  read a line from the file
      printf("%s\n",line);
      printf("Done...\n");
      close(fd);
   }
  else
   {
     exit(2);
   }
} /* end main */
```

Wrappers:

```
jacques@UBU64vm:~/Trent_teaching/2017_3380$ ls -lt
-rwxrwxr-x 1 jacques jacques 13296 Nov 15 12:58 open_wrapper
-rw-rw-r-- 1 jacques jacques  2891 Nov 15 12:57 open_wrapper.c
-rw-rw-r-- 1 jacques jacques   101 Nov 15 12:57 test_data.csv
-rwx------ 1 root    root        0 Nov 15 09:53 mydata.csv

jacques@UBU64vm:~/Trent_teaching/2017_3380$ ./open_wrapper test_data.csv
Processing file: test_data.csv
Number of data points:20::
livingstone:b3:
20.9:338.5:46:133
22.68:348:50.6:132.5
23.58:347:49.1:128

Done...

jacques@UBU64vm:~/Trent_teaching/2017_3380$ ./open_wrapper mydata.csv
You do not have READ access to the file mydata.csv.


jacques@UBU64vm:~/Trent_teaching/2017_3380$ ./open_wrapper fred.dat
The requested file (fred.dat) doesn't exist. Please supply an existing file when you re-run
   this code.
```

Files and Directories

Files in UNIX

A quick reminder from our Intro to UNIX:

- Files in UNIX are just a collection of bytes
- There are no “extensions” in UNIX
- A file is stored in a directory.
- In that directory a file is associated with an inode.
  - Each inode on a file system is unique
  - Files on different file systems may have the same inode number but they are DIFFERENT. Remember an inode points to data blocks on a physical device.

Frigg – File “systems”

```
[jacques@loki ~]$ df -h
Filesystem       Size  Used Avail Use% Mounted on
devtmpfs         909M     0  909M   0% /dev
tmpfs            920M     0  920M   0% /dev/shm
tmpfs            920M   98M  822M  11% /run
tmpfs            920M     0  920M   0% /sys/fs/cgroup
/dev/sda3         28G  3.1G   24G  12% /
/dev/sda1        976M  273M  637M  30% /boot
/dev/sdb1         20G  3.0G   16G  16% /var
/dev/sdd1         35G   19G   15G  57% /home
```

/home of user should be on its own drive - sd1, because if too many users and files and then that's it, done

sda, sdd - these are different physical disks

Note: /dev is a directory where *NIX stores its “devices”.

Each “entry” in that directory POINTS to a different device (or access method for a device)

Therefore /dev/sda3 and /deb/sdb1 are on different physical disks. sda2 refers to SCSI disk A, partition 2. sdb1 refers to SCSI disk B, first partition.

The root / of the file system

```
root@sherlock:/home/forensics# ls / -lt
total 92
drwxrwxrwt  13 root root  4096 Jan 27 14:09 tmp
drwxr-xr-x   3 root root  4096 Jan 27 14:06 boot
lrwxrwxrwx   1 root root    32 Jan 27 14:05 initrd.img -> boot/initrd.img-5.4.0-65-generic
lrwxrwxrwx   1 root root    29 Jan 27 14:05 vmlinuz -> boot/vmlinuz-5.4.0-65-generic
lrwxrwxrwx   1 root root    32 Jan 27 14:05 initrd.img.old -> boot/initrd.img-5.4.0-64-
    generic
lrwxrwxrwx   1 root root    29 Jan 27 14:05 vmlinuz.old -> boot/vmlinuz-5.4.0-64-generic
drwxr-xr-x 100 root root 12288 Jan 27 14:05 etc
drwxr-xr-x  19 root root  4100 Jan 27 14:03 dev
drwxr-xr-x   2 root root 12288 Jan 27 14:03 sbin
drwxr-xr-x   2 root root  4096 Jan 27 14:03 lib64
drwxr-xr-x  23 root root   660 Jan 27 14:02 run
dr-xr-xr-x  13 root root     0 Jan 21 09:36 sys
dr-xr-xr-x 212 root root     0 Jan 21 09:36 proc
drwxr-xr-x   2 root root  4096 Jan 20 21:28 bin
drwx------   5 root root  4096 Dec  9 09:59 root
drwxr-xr-x  11 root root  4096 Nov 16 11:59 usr
drwxr-xr-x  20 root root  4096 Nov 16 11:57 lib
drwxr-xr-x   5 root root  4096 Apr 13  2017 home
drwxr-xr-x  12 root root  4096 Sep 20  2016 var
drwxr-xr-x   2 root root  4096 Sep 20  2016 opt
drwxr-xr-x   4 root root  4096 Sep 20  2016 media
drwx------   2 root root 16384 Sep 20  2016 lost+found
drwxr-xr-x   2 root root  4096 Jul 19  2016 mnt
drwxr-xr-x   2 root root  4096 Jul 19  2016 srv
root@sherlock:/home/forensics#
```

function - tree -l will print our the directory structure of that folder

At the Shell Level

```
[jacques@loki ~]$ ls -lF /home/COIS
total 32
drwxr-xr-x. 4 root COIS-3380H-A  4096 Jan 23 10:00 3380/
drwxrwxr-x. 2 root jacques      20480 Jan 23 09:08 account_backups/
drwxr-xr-x. 2 root root          4096 Oct 27 11:28 quota_info/
drwxrwx---. 2 root apache        4096 Dec  5 14:30 snapshots/
```

file name

modify date

group

owner

size

number of links has a link to some inode location

each one points to its parent directory
. that file
.. parent directory

permissions

file type

Directory Listings

Adding the **–i** flag to the ls commands, allows us to see the inode numbers for each file.

1, i-include i nodes, t-sort by time

```
jacquesabeland@frigg:~/Sample_Directory> ls -lit
total 16
394081 lrwxrwxrwx 1 jacquesabeland jacquesabeland   26 Feb  2 10:41 test_warn_file ->
                                                                       /home/common/warn-20150517
394080 -rw-r--r-- 1 jacquesabeland jacquesabeland 3093 Feb  2 10:38 string_testing.c
394079 -rwxr--r-- 2 jacquesabeland jacquesabeland 1850 Feb  2 10:38 HeavyHitters
394079 -rwxr--r-- 2 jacquesabeland jacquesabeland 1850 Feb  2 10:38 topten.sh
385402 -rw------- 2 jacquesabeland jacquesabeland 1177 Sep 29  2015 my_login_script
```

these both are pointing to same inode - 2 files in folder but pointing to same i node-this is called as "LINK"

```
jacquesabeland@frigg:~/Sample_Directory> ls -lit /home/common/warn-20150517
51 -rwxr-xr-- 1 jacquesabeland jacquesabeland 4722758 Jan 20  2016 /home/common/warn-20150517
```

```
jacquesabeland@frigg:~/Sample_Directory> ls -lit $HOME/.bashrc
385402 -rw------- 2 jacquesabeland jacquesabeland 1177 Sep 29  2015 /home/jacquesabeland/.bashrc
```

Basic file I/O

- Processes keep a list of open files
- Files can be opened for reading, writing
- Each file is referenced by a *file descriptor* (integer)

- Three files are opened automatically for every process
  – fd 0: standard input - stdin
  – fd 1: standard output - stdout
  – fd 2: standard error - stderr

UNIX File access primitives

- open – open for reading, writing or to create an empty file
- creat - create an empty file
- close – close a currently opened fiel descriptor

- read - get info from file
- write - put info in file
- lseek - move to specific byte offset within file

- unlink - remove a file
- remove - remove a file
- fcntl - control attributes assoc. w/ file

open

```
#include <sys/types.h>
#include <sys/stat.h.
#include <fctl.h>
int open(const char *pathname, int flags, [mode_t mode]);
```

returns an integer - link to that file -> file descriptor

- *pathname* – is a pointer to a string that contains the pathname of the file to be opened.
- *flags* – specifies the access method (must use 1 of first 3)
  - O_RDONLY : read only
  - O_WRONLY : write only
  - O_RDWR : read and write
  - O_CREAT : create a new file (uses *mode)*
  - O_APPEND : add to the end of a file
  - O_TRUNC : removes the contents of the file

these are stored in typed.h

- *mode* : used with *O_CREAT (typically 0644)*
- returns 0 if successful and -1 if an error

open: examples (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
char *filename = “junkfile”;

int main(void)
      {
      int fd;
      if( (fd = open(filename, O_RDONLY)) == -1)
                {
                printf(“Could not open file %s\n”, filename);
                exit(1);
                }
      else
                {
                printf(“Did open file %s\n”, filename);
                exit(0);
                }
      return 0;
      }
```

open: examples (2)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
char *filename = “junkfile”;

int main(void)
    {
    int fd;
    if( (fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0644 )) == -1)
            {
            printf(“Could not open file %s\n”, filename);
            exit(1);
            }
    else
            {
            printf(“Did open file %s\n”, filename);
            exit(0);
            }
    return 0;
    }
```

this is not pipe operator; this is C, so its the OR operator

Will truncate the file contents if the file already exists

open: examples (3)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
char *filename = “junkfile”;

int main(void)
    {
    int fd;
    if( (fd = open(filename, O_RDWR | O_CREAT | O_EXCL, 0644 )) == -1)
            {
            printf(“Could not open file %s\n”, filename);
            exit(1);
            }
    else
            {
            printf(“Did open file %s\n”, filename);
            exit(0);
            }
    return 0;
    }
```

Will give an error message if the file already exists

read

give me apointer, don't care what datatype is associated with it

```
#include <unistd.h>
ssize_t read(int filedes, void *buffer, size_t n);
```

- Copies an arbitrary number of bytes (characters) from a file into a buffer (begins at the read-write pointer)
- *filedes* – the file descriptor for the open file
- *buffer* – usually an array of chars but could be a structure
- *n* – is the number of bytes to read
- Returns the actual number of bytes read if successful and -1 if an error

read: example

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

int main(void)
    {
    char *testfile = “junk”;
    char buffer[512];
    int fd;
    ssize_t nread;
    long total = 0;
    if( (fd = open(testfile, O_RDONLY)) == -1)
        {
        printf(“Could not open file\n”);
        exit(1);
        }
    while( (nread = read(fd, buffer, 512)) > 0)
        total += nread;
    printf(“The total number of characters in the file was: %ld\n”, total);
    exit(0);
    }
```

there is no datatype - "byte", so we use char for buffer

This example counts the number of characters in a file. It also demonstrates how to detect the end of file (when the amount read is not greater than 0)

write

```
#include <unistd.h>
ssize_t write(int filedes, void *buffer, size_t n);
```

- Copies an arbitrary number of bytes (characters) from a process buffer to an external file (begins at the read-write pointer)
- *filedes* – the file descriptor for the open file
- *buffer* – usually an array of chars but could be a structure
- *n* – is the number of bytes to write
- Returns the actual number of bytes written if successful and -1 if an error

read/write: example

This function copies the contents of one file to another

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#define BUFSIZE 512

int copyfile(const char *name1, const char *name2)
    {
    int infile, outfile;
    ssize_t nread;
    char buffer[BUFSIZE];
    if((infile = open(name1, O_RDONLY)) == -1)
            return -1;
    if((outfile = open(name2, O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1)
            {
            close(infile);
            return -2;
            }
    while((nread = read(infile, buffer, BUFSIZE)) > 0)
            {
            if(write(outfile, buffer, nread) < nread)
              {
              close(infile);  close(outfile);
              return -3;
              }
            }
    close(infile); close(outfile);
    if(nread == -1)
            return -4;
    else
            return 0;
    }
```

```
int main(void)
    {
    char *file1 = “testfile1”;
    char *file2 = “testfile2”;
    int result;
    /* call copyfile and test return code */
    result = copyfile(file1, file2);
    switch(result)
        {
        case 0:     printf(“Contents copied from %s to %s\n”, file1, file2);
                    break;
        case -1:    printf(“Error opening input file: %s\n”, file1);
                    break;
        case -2:    printf(“Error opening output file: %s\n”, file2);
                    break;
        case -3:    printf(“Error writing to file: %s\n”, file2);
                    break;
        case -4:    printf(“Error in last write to file: %s\n”, file2);
                    break;
        default:    printf(“Unknown error occurred\n”);
        }
    if(result != 0)
        exit(1);
    else
        exit(0);
    }
```

Main program for the copyfile function.

Why use read()/write()

- Maximal performance
  - IF you know exactly what you are doing
  - No additional hidden overhead from stdio
- Control exactly what is written/read at what times

lseek

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

unistd - uni standard has lseek

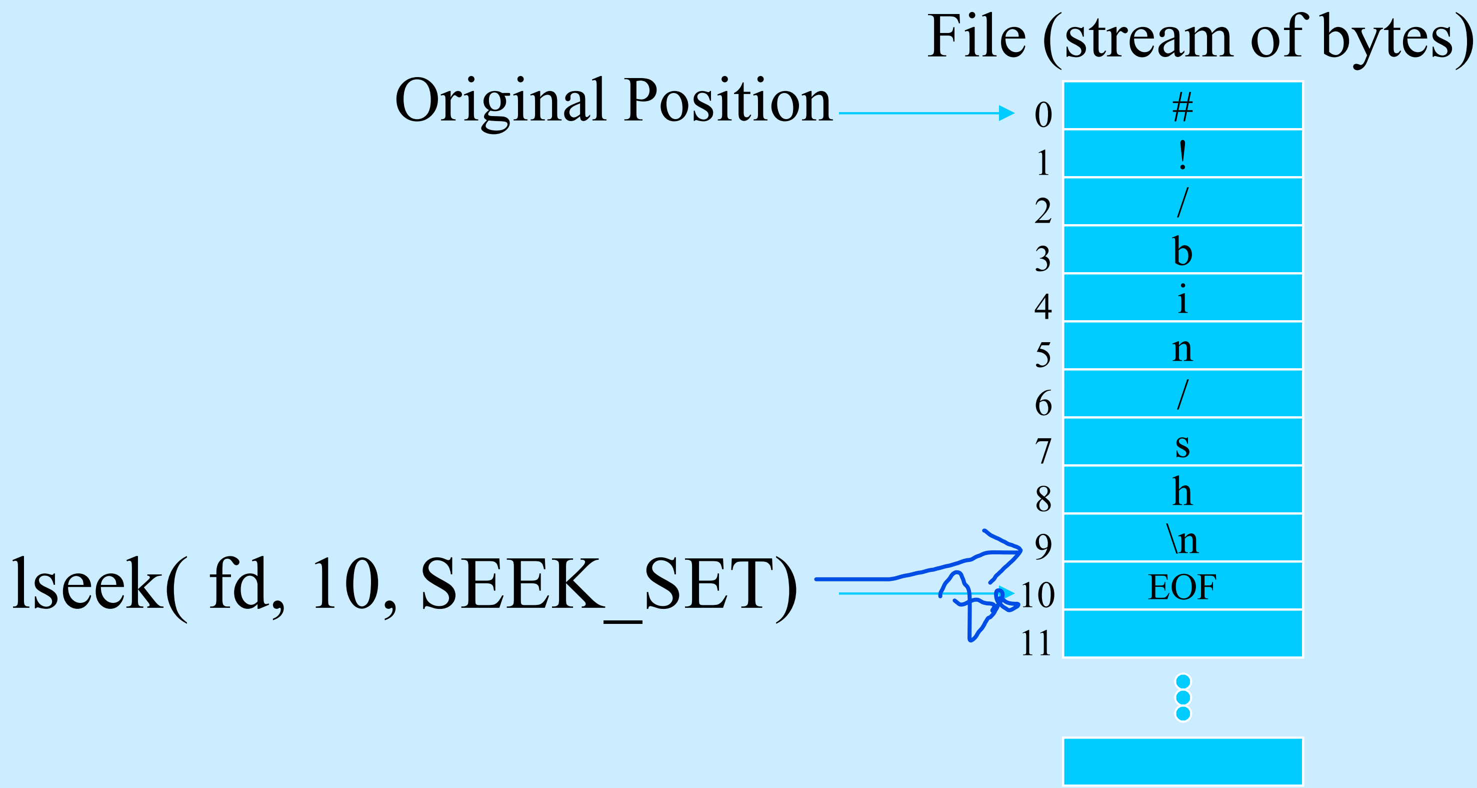
- Repositions the offset of the file descriptor *fd* to the argument offset.
- *whence*
    - SEEK_SET
        - The offset is set to offset bytes.
    - SEEK_CUR
        - The offset is set to its current location plus offset bytes.
    - SEEK_END
        - The offset is set to the size of the file plus offset bytes.

end of the file

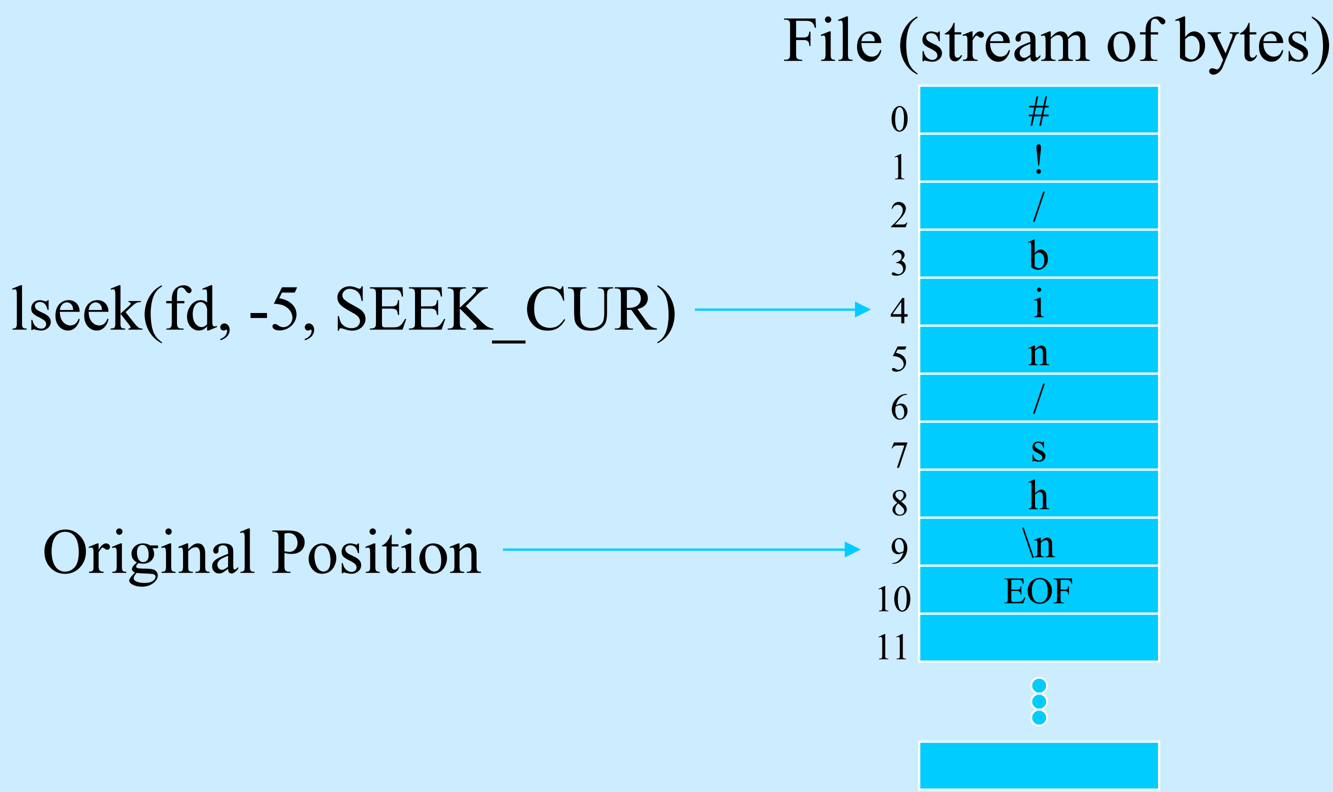
lseek: Examples

- Random access
  - Jump to any byte in a file
- Move to byte #16
  - `newpos = lseek( file_descriptor, 16, SEEK_SET );`
- Move forward 4 bytes
  - `newpos = lseek( file_descriptor, 4, SEEK_CUR );`
- Move to 8 bytes from the end
  - `newpos = lseek( file_descriptor, -8, SEEK_END );`

lseek - SEEK_SET (10)



lseek - SEEK_CUR (-5)

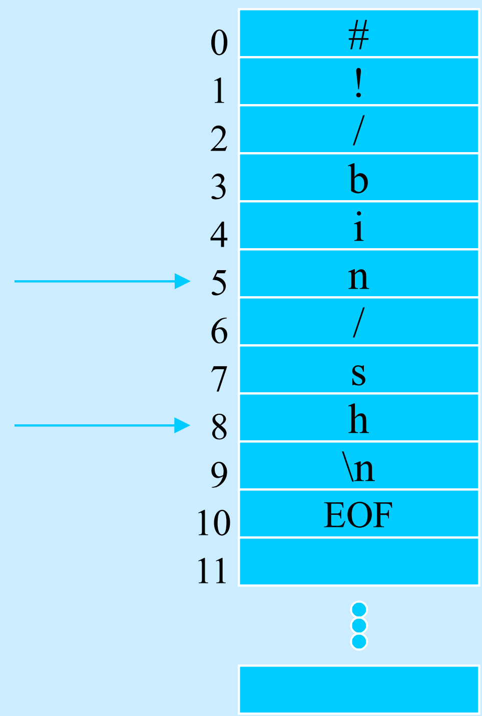


lseek - SEEK_CUR(3)

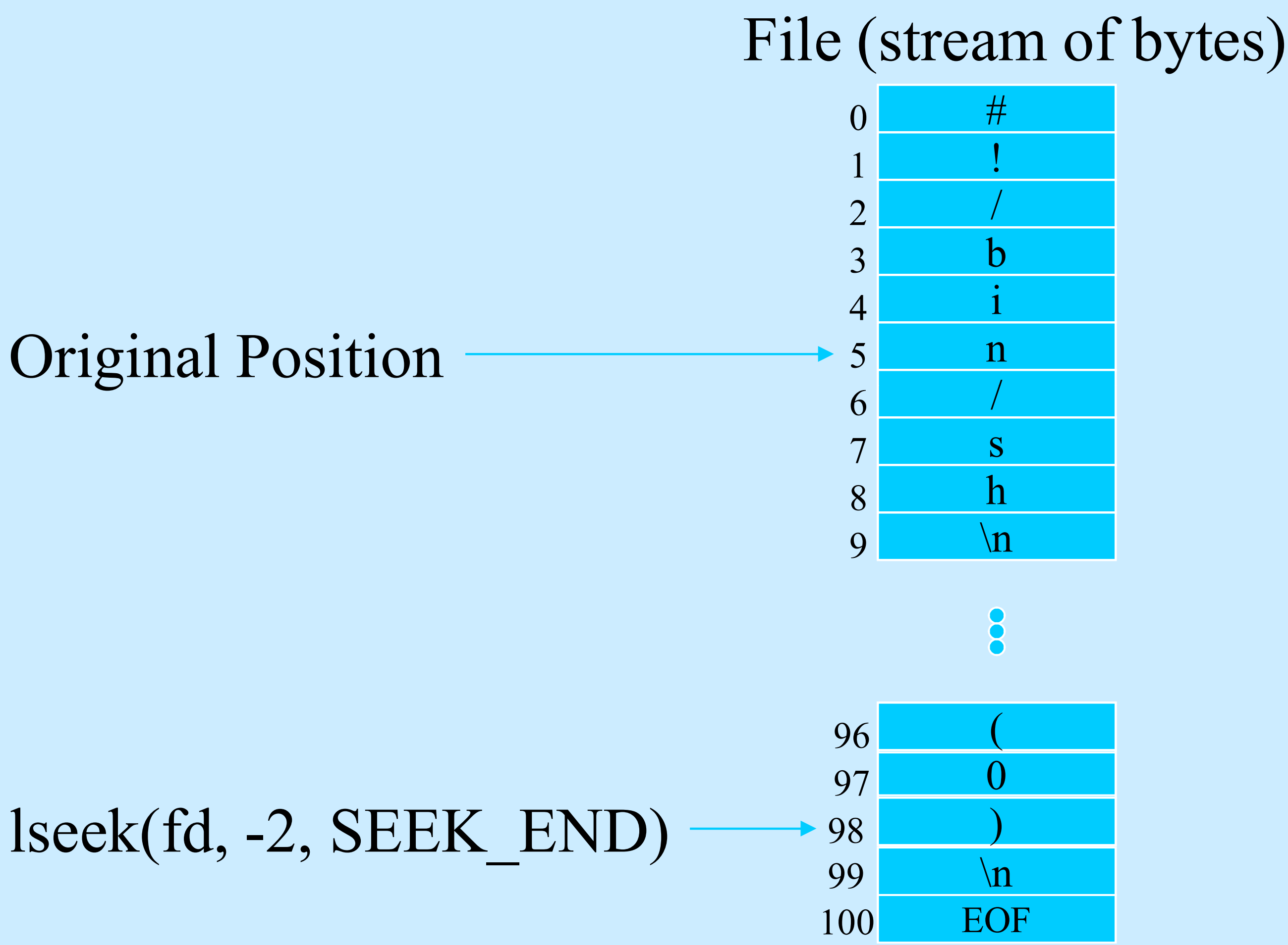
File (stream of bytes)

Original Position

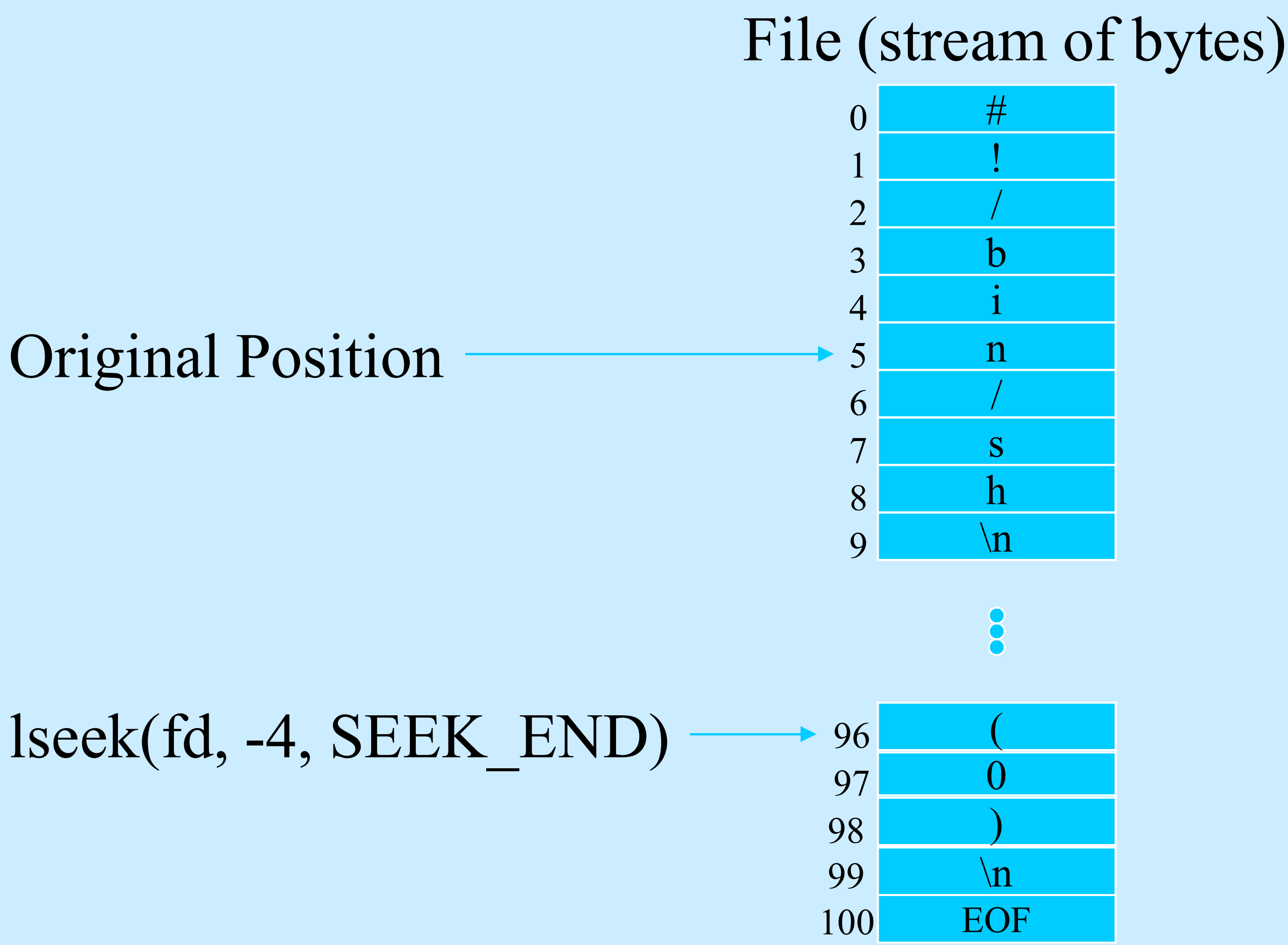
lseek(fd, 3, SEEK_CUR)



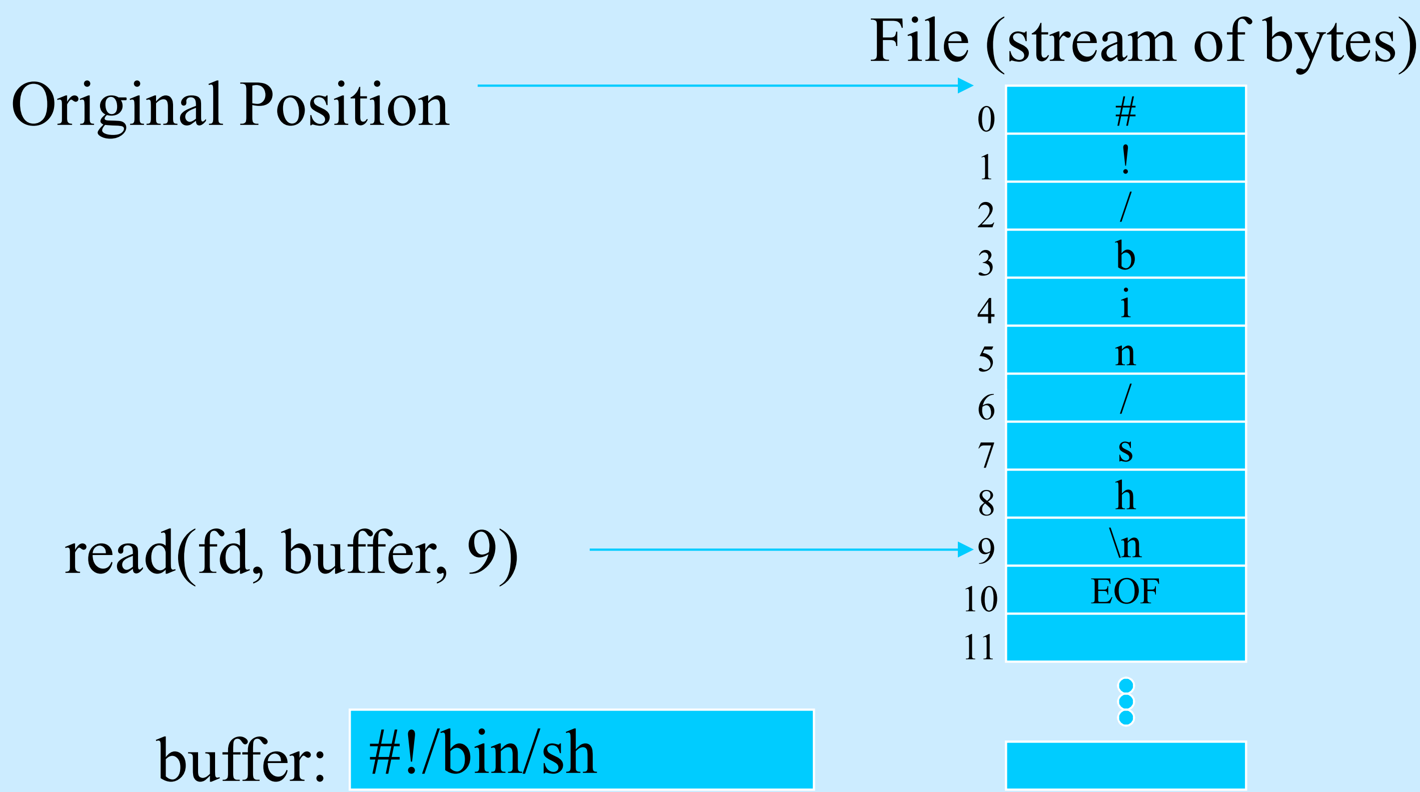
lseek - SEEK_END (-2)



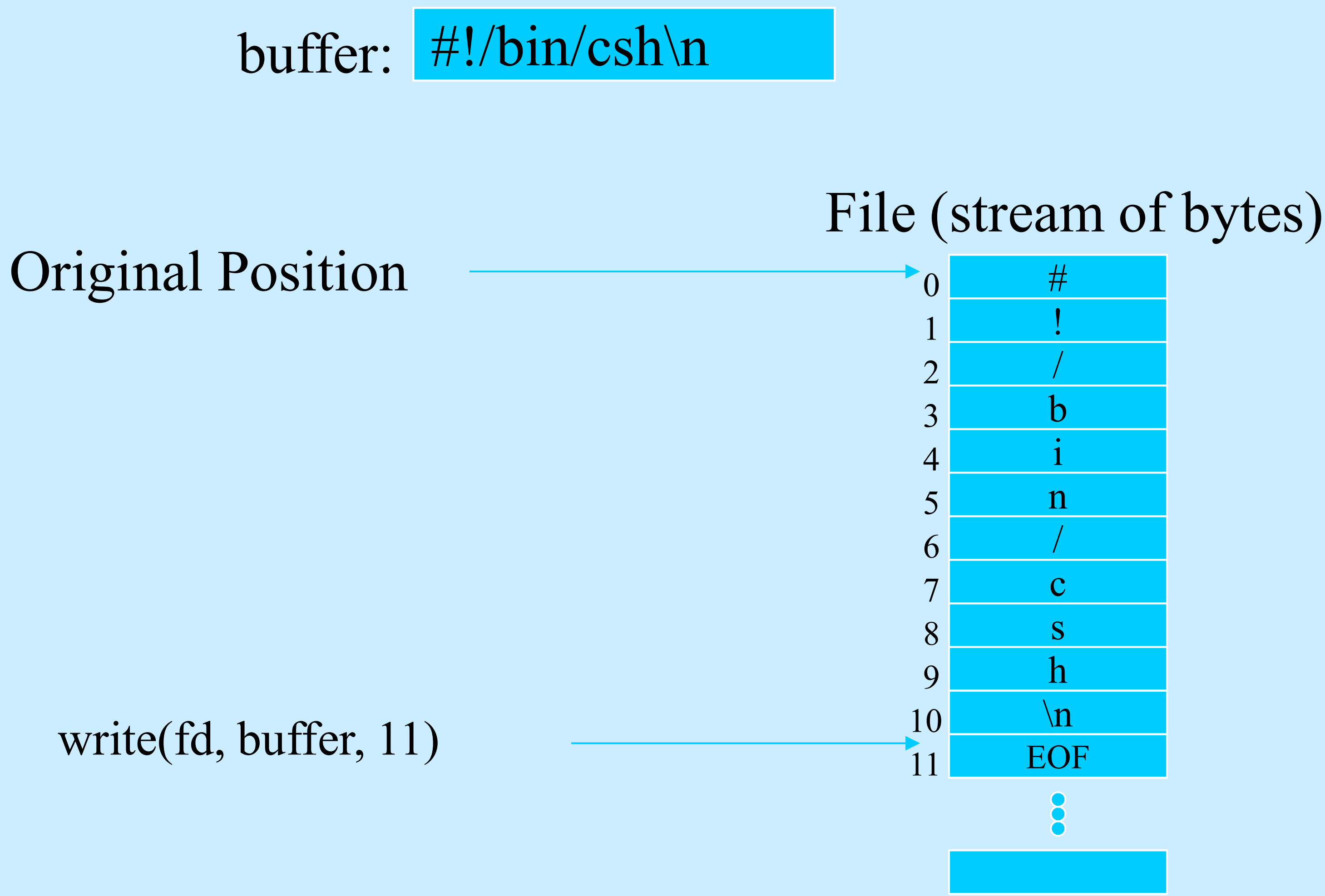
lseek - SEEK_END (-4)



Read – File Pointer



Write – File Pointer



Example #1: lseek

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

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void)
   {
   int fd;

   if( (fd = open("file.hole", O_WRONLY | O_CREAT, 0644)) < 0 )
        {
        perror("open error");
        exit(1);
        }
```

Example #1: lseek (2)

```
if( write(fd, buf1, 10) != 10 )
      {
      perror("buf1 write error");
      exit(1);
      }
/* offset now = 10 */
if( lseek(fd, 40, SEEK_SET) == -1 )
      {
      perror("lseek error");
      exit(1);
      }
/* offset now = 40 */
if( write(fd, buf2, 10) != 10  )
      {
      perror("buf2 write error");
      exit(1);
      }
/* offset now = 50 */
exit(0);
}
```

Deleting a File

```
#include <unistd.h>
int unlink(const char *pathname);

Or

#include <stdio.h>
int remove(const char *pathname);
```

- *pathname* is the file to be deleted
- Both return 0 for success or -1 for failure
- *unlink* was the original system call and *remove* was added later
- To remove empty directories, *remove*(path) is equivalent to *rmdir*(path)

File I/O using FILEs

- Most UNIX programs use higher-level I/O functions
  - `fopen()`
  - `fclose()`
  - `fread()`
  - `fwrite()`
  - `fseek()`
- These use the **`FILE`** datatype instead of file descriptors
- Need to include `stdio.h`

The Standard IO Library

If you want to use: fopen,fclose, printf, fprintf, sprintf, scanf, fscanf, getc, putc, gets, fgets, etc.

Then your code will need to have:

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

Using datatypes with file I/O

- All the functions we’ve seen so far use raw bytes for file I/O, but program data is usually stored in meaningful datatypes (int, char, float, etc.)
- `fprintf(), fputs(), fputc()` - used to write data to a file
- `fscanf(), fgets(), fgetc()` - used to read data from a file

fopen() and printf()

```
include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main( int argc, char *argv[] )
{
  FILE *file_descriptor;

  int    integer_value;
  double double_value;
  char   a_text_string[20] = "Hello mom!\0";

  file_descriptor = fopen("fprintf_test.dat","w");

  integer_value = 137;
  double_value  = 137.04;

  fprintf(file_descriptor, "%d %f %s\n",integer_value,double_value,a_text_string );

  fclose(file_descriptor);

}
```

fscanf()

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main( int argc, char *argv[] )
{
  FILE *file_descriptor;

  int    integer_value;
  double double_value;
  char   a_text_string[20] = "test test test\0";

  file_descriptor = fopen("fprintf_test.dat","r");

  integer_value = 23;
  double_value  = 2.1416;

  printf("before the read: integer_value=%d \tdouble_value=%f\ta_text_string=%s
                \n", integer_value,double_value,a_text_string);

  fscanf(file_descriptor, "%d %lf %s",&integer_value,&double_value,a_text_string );

  printf("after the read: integer_value=%d \tdouble_value=%f\ta_text_string=%s
                \n",integer_value,double_value,a_text_string);

  fclose(file_descriptor);

}
```

Obtaining File Information

For analyzing files.

- stat(), fstat(), lstat()
- Retrieve all sorts of information about a file
  - Which device it is stored on
  - Information:
    - Ownership/Permissions of that file,
    - Number of links
    - Size of the file
    - Date/Time of last modification and access
    - Ideal block size for I/O to this file

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

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

stat structure will allow OS to bring all info into the data variable

All return 0 on success, or –1 on error

(from Kerrisk p279)

^ - start with

struct stat

We will look at st_mode in detail.

```
struct stat
   {
   dev_t st_dev;     /* device num.              */
   dev_t st_rdev;     /* device # spcl files   */
   ino_t st_ino;     /* i-node num.              */
   mode_t st_mode;   /* file type,mode,perms */
   nlink_t st_nlink; /* num. of links            */
   uid_t st_uid;     /* uid of owner             */
   gid_t st_gid;     /* group-id of owner        */
   off_t st_size;    /* size in bytes            */
   time_t st_atime;  /* last access time         */
   time_t st_mtime;  /* last mod. time           */
   time_t st_ctime;  /* last stat chg time       */
   long st_blksize;  /* best I/O block size      */
   long st_blocks;   /* # of 512 blocks used */
   }
```

https://www.gnu.org/software/libc/manual/html_node/Attribute-Meanings.html

`mode_t st_mode`: File Types

1. Regular File (text/binary)
2. Directory File
3. Character Special File

   e.g. I/O peripherals, such as `/dev/ttyp0`
4. Block Special File

   e.g. cdrom, such as `/dev/mcd`
5. FIFO (named pipes)
6. Sockets
7. Symbolic Links

File Mix on a Typical System

File Type	*Count*	*Percentage*
regular file	30,369	91.7%
directory	1,901	5.7
symbolic link	416	1.3
char special	373	1.1
block special	61	0.2
socket	5	0.0
FIFO	1	0.0

File Mix on a Typical System

```
ls -ltR / | grep -v -e ^total | cut -c 1 | sort | uniq -c | sort -nr
```

- : regular file
d : directory
c : character device file
b : block device file
s : local socket file
p : named pipe
l : symbolic link

Newton

459119	-	72%
116929	l	18%
57340	d	9%
223	c	.04%
74	s	.01%
17	p	
14	b	

Loki

403547	-	75%
69877	l	13%
63284	d	12%
167	c	.03%
53	s	.01%
43	p	.01%
6	b	

Getting Mode Information

- `AND` the `st_mode` field with one of the following masks and test for non-zero:
  - `S_ISUID` set-user-id bit is set
  - `S_ISGID` set-group-id bit is set
  - `S_ISVTX` sticky bit is set

- Example:

```
if( (sbuf.st_mode & S_ISUID) != 0 )
  printf("set-user-id bit is set\n");
```

Bitmaps – Saving resources

Historically, there was never enough memory to do work. A significant amount of effort was put into storing as much information in as tiny a space as possible.

If you had a situation where you had a number of attributes that were either present, or NOT, then you could use a single BIT to represent this information.

BIT =1 attribute is TRUE, BIT=0 attribute is FALSE

Bitmaps

We can therefore take a simple integer and use it to hold a significant amount of information.

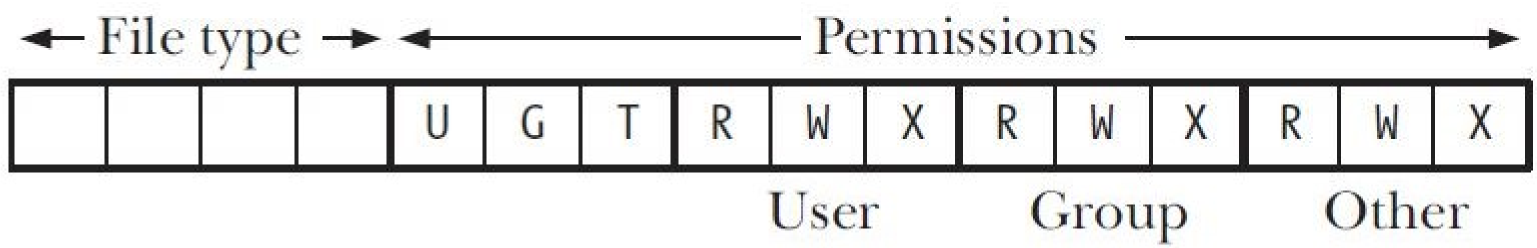


Figure 15-1: Layout of *st_mode* bit mask

The Linux Programming Interface Page 281

```
[jacques@loki 3380]$ ls -lt  file_perm_bitmap*
-rwxrwxr-x. 1 jacques jacques 8368 Feb  3 11:48 file_perm_bitmap
-rwxrwx---. 1 jacques jacques  454 Feb  3 11:48 file_perm_bitmap.c
```

Bitmaps – Testing for values

To check to see if a specific value is set, you can perform an AND operation with a known bitmap.

If the result is >0, the bit was set.

If the result is zero, then the bit was NOT set.

Some test values have multiple bits set: Check against a know value: S_IFMT pulls out the "file type" bits.

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
            printf("regular file\n");
```

Bitmaps:

You can extract/read this information from the inode information for each file using the stat() call.

We will study this later but for now, as an example of bitmaps, we can extract values:

```
[jacques@loki 3380]$ ./file_perm_bitmap
S_IRUSR: 256
S_IWUSR: 128
S_IXUSR: 64
S_IRGRP: 32
S_IWGRP: 16
S_IXGRP: 8
S_IROTH: 4
S_IWOTH: 2
S_IXOTH: 1
[jacques@loki 3380]$
```

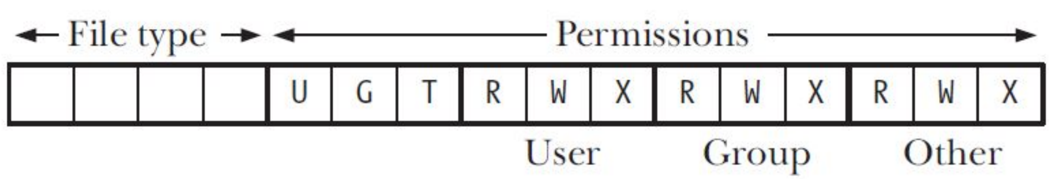


Figure 15-1: Layout of *st_mode* bit mask

```
if( (file_status.st_mode & S_IRUSR) ) { printf("r");} else { printf("-"); }
if( (file_status.st_mode & S_IWUSR) ) { printf("w");} else { printf("-"); }
if( (file_status.st_mode & S_IXUSR) ) { printf("x");} else { printf("-"); }
```

Getting Permission Info.

- `AND` the `st_mode` field with one of the following masks and test for non-zero:
  - `S_IRUSR` 0400 user read
    `S_IWUSR` 0200 user write
    `S_IXUSR` 0100 user execute
  - `S_IRGRP` 0040 group read
    `S_IWGRP` 0020 group write
    `S_IXGRP` 0010 group execute
  - `S_IROTH` 0004 other read
    `S_IWOTH` 0002 other write
    `S_IXOTH` 0001 other execute

`st_mode` Field

- This field contains type and permissions (12 lower bits) of file in bit format.

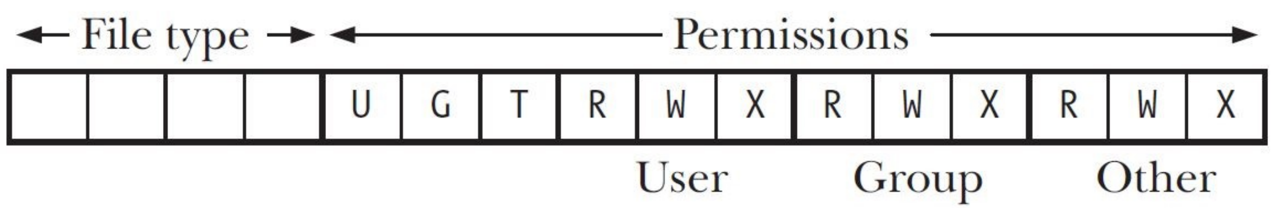


Figure 15-1: Layout of *st_mode* bit mask (from: Kerrisk page281)

- It is extracted by `AND`-ing the value stored there with various constants
  - see **`man stat`**
  - also **`<sys/stat.h>`** and **`<linux/stat.h>`**
  - some data structures are in **`<bits/stat.h>`**

Getting the Type Information

- `AND` the `st_mode` field with `S_IFMT` to get the type bits.

- Test the result against:
  - **`S_IFREG`** Regular file
  - **`S_IFDIR`** Directory
  - **`S_IFSOCK`** Socket
  - etc.

  Note: S_IFMT = 61440 $= 2^{15}+2^{14}+2^{13}+2^{12}$

  S_IFREG=2^{15} S_IFDIR=2^{14} S_IFFIFO=2^{12}

Example

```
struct stat sbuf;
    :
if( stat( file, &sbuf ) == 0 )
  if( (sbuf.st_mode & S_IFMT) == S_IFDIR )
         printf("A directory\n");
```

Type Info. Macros

- Modern UNIX systems include test macros in `<sys/stat.h>` and `<linux/stat.h>`:
  - **`S_ISREG()`** **regular file**
  - **`S_ISDIR()`** **directory file**
  - **`S_ISCHR()`** **char. special file**
  - **`S_ISBLK()`** **block special file**
  - **`S_ISFIFO()`** **pipe or FIFO**
  - **`S_ISLNK()`** **symbolic link**
  - **`S_ISSOCK()`** **socket**

Example

```
struct stat sbuf;
   :
if( stat(file, &sbuf ) == 0 )
   {
   if( S_ISREG( sbuf.st_mode ) )
         printf( "A regular file\n" );
   else if( S_ISDIR(sbuf.st_mode) )
         printf( "A directory\n" );
   else ...
   }
```

Example

```
struct stat sbuf;
      :
  printf( “Permissions: “ );
  if( (sbuf.st_mode & S_IRUSR) != 0 )
      printf( “user read, ” );
  if( (sbuf.st_mode & S_IWUSR) != 0 )
      printf( “user write, ” );
      :
```

Another Example

```
struct stat sbuf;
      :
  if( stat(file1, &sbuf ) == 0 )
      {
      /* remove group/other read/write */
      sbuf.st_mode &= 0700;
      if(chmod(file2, sbuf.st_mode) == -1)
            printf( "Error in chmod\n" );
      }
  else
      printf( "Error in stat\n" );
```

File control of open files: fcntl()

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

int fcntl( int fd, int cmd );
int fcntl( int fd, int cmd, long arg );
```

- Provides a degree of control over open files
- Provides a variety of functions
- Performs operations pertaining to *fd*, the file descriptor
- Specific operation depends on *cmd*

fcntl: cmd

- F_GETFL
  - Returns the current file status flags (as a bitmap) as set by open().
  - Access mode can be extracted from AND'ing the return value
    - return_value & O_ACCMODE
      - e.g. O_WRONLY
- F_SETFL
  - Sets the file status flags associated with fd.
  - Only O_APPEND, O_NONBLOCK and O_ASYNC may be set.
  - Other flags are unaffected

Example 1: fcntl()

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

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

     if( argc != 2 )
        {
        fprintf( stderr, "usage: myprog <descriptor#>" );
        exit(1);
        }

     if( (val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0 )
        {
        perror( "fcntl error for fd" );
        exit( 1 );
        }
   accmode = val & O_ACCMODE;
```

Masks out / extracts only the access mode bit fields!

```
if( accmode == O_RDONLY )
      printf( "read only" );
else if(accmode == O_WRONLY )
      printf( "write only" );
else if( accmode == O_RDWR )
      printf( "read write" );
else
      {
      fprintf( stderr, "unkown access mode" );
      exit(1);
      }

if( val & O_APPEND )
      printf( ", append");
if( val & O_NONBLOCK)
      printf(", nonblocking");
if( val & O_SYNC )
      printf(", synchronous writes");
putchar( '\n' );
exit(0);
}
```

Example #2: fcntl

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

/* flags are file status flags to turn on */
void set_fl( int fd, int flags )
   {
   int val;

   if( (val = fcntl( fd, F_GETFL, 0 )) < 0 )
        {
       perror( "fcntl F_GETFL error" );
       exit( 1 );
        }
   val |= flags;     /* turn on flags (Bitwise OR assignment) */
   if( fcntl( fd, F_SETFL, val ) < 0 )
        {
       perror( "fcntl F_SETFL error" );
       exit( 1 );
        }
   }
```

File Concept – An Abstract Data Type

- File Types
- File Operations
- File Attributes
- File Structure - Logical
- Internal File Structure

File Types

- Regular files
- Directory files
- Character special files
- Block special files
- FIFOs / Pipes
- Sockets
- Symbolic Links

File Operations

- Creating a file
- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file

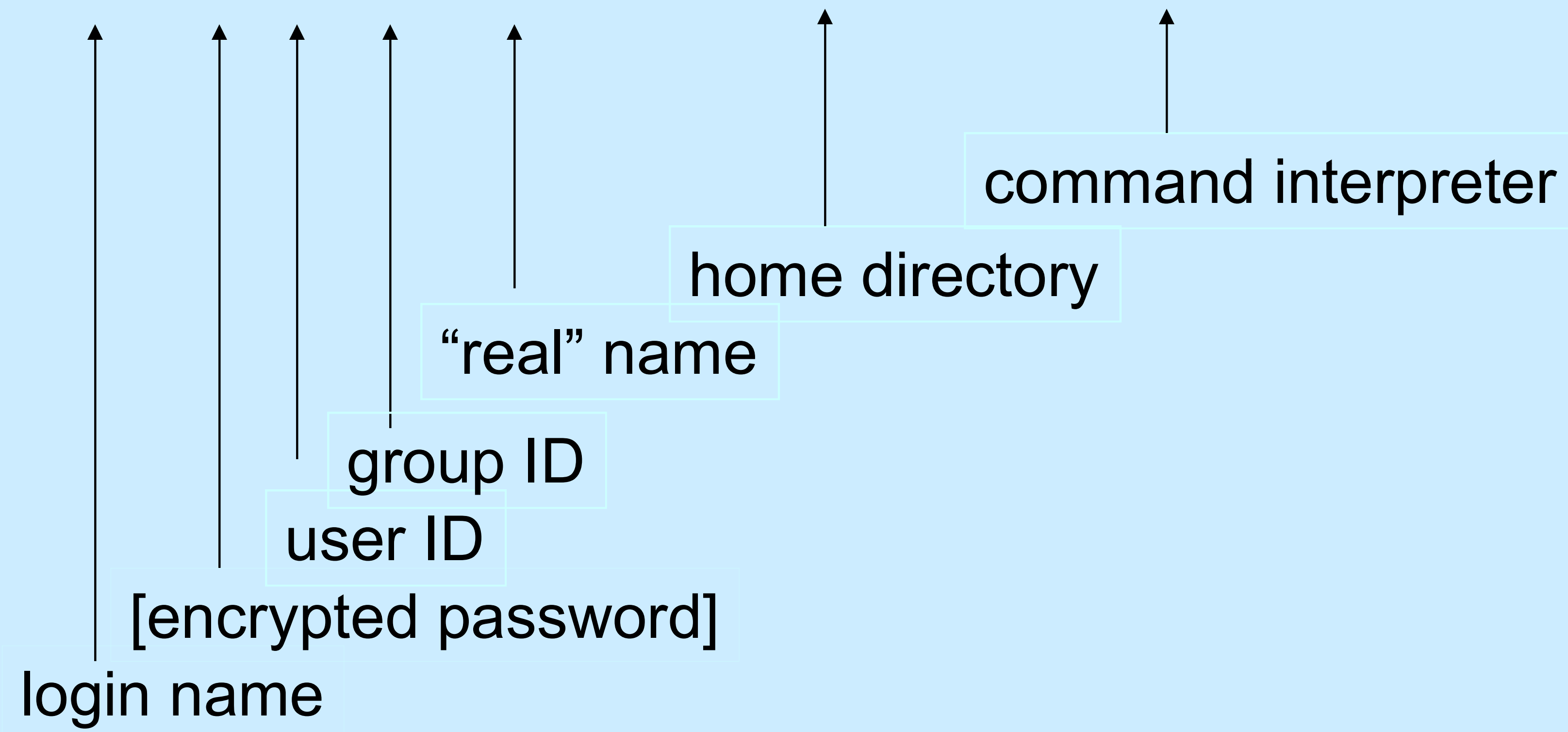
Files Attributes

- Name
- Type
- Location
- Size
- Protection
- Time,date and user identification

Users and Ownership: /etc/passwd

- Every File is owned by one of the system's users – identity is represented by the user-id (UID)
- Password file associated UID with system users.

```
gates:x:65:20:B. Gates:/home/gates:/bin/ksh
```



/etc/group

- Information about system groups

`faculty:x:23:rhurley,eileen,dkl`

list of group members

group ID

[encrypted group password]

group name

Real uids

- The uid of the user who *started* the program is used as its *real uid*.
- The real uid affects what the program can do (e.g. create, delete files).
- For example, the uid of `/usr/bin/vi` is **`root`**:
  - **`$ ls -alt /usr/bin/vi`**
    `lrwxrwxrwx 1 root root 20 Apr 13...`
- But when I use `vi`, its user id is `jacques` (not `root`), so I can only edit my files.

Effective uids

- Programs can change to use the *effective uid*
  - the uid of the program *owner*
  - e.g. the `passwd` program changes to use its effective uid (`root`) so that it can edit the `/etc/passwd` file

- This feature is used by many system tools, such as logging programs.

```
[jacques@loki tools]$ ls /home/COIS/show_http_errors  -lt
-rwsr-xr-x. 1 root root 8776 Jun 28  2019 /home/COIS/show_http_errors
```

Real and Effective Group-ids

- There are also real and effective group-ids.

- Usually a program uses the *real group-id* (i.e. the *group-id of the user*).

- Sometimes useful to use *effective group-id* (i.e. group-id of program *owner*):
  - e.g. software shared across teams

Extra File Permissions

- *Octal Value* *Meaning*

Octal Value	Meaning
`04000`	Set user-id on execution. Symbolic: `--s --- ---`
`02000`	Set group-id on execution. Symbolic: `--- --s ---`

- These specify that a program should use the effective user/group id during execution.

- For example:
  - `$ ls -alt /usr/bin/passwd`
    `-rwsr-xr-x 1 root root 25692 May 24...`

File Mode (Permission)

- S_IRUSR -- user-read
- S_IWUSR -- user-write
- S_IXUSR -- user-execute
- S_IRGRP -- group-read
- S_IWGRP -- group-write
- S_IXGRP -- group-execute
- S_IROTH -- other-read
- S_IWOTH -- other-write
- S_IXOTH -- other-execute

chmod and fchmod

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod( const char *path, mode_t mode ) ;
int fchmod( int fd, mode_t mode );
```

Change permissions of a file.

The mode of the file given by *path* or referenced by *fd* is changed.

mode is specified by OR'ing the following.

S_I{R,W,X}{USR,GRP,OTH} (basic permissions)

S_ISUID, S_ISGID, S_ISVTX (special bits)

Effective uid of the process must be zero (superuser) or must match the owner of the file.

On success, zero is returned. On error, -1 is returned.

NOTE: The sticky bit (**S_ISVTX**) on a directory means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, and by a privileged process.

Example: chmod

```
/* set absolute mode to "rw-r--r--" */
if( chmod("bar", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) < 0)
     {
     perror("chmod error for bar");
     exit(1);
     }
exit(0);
}
```

chown, fchown, lchown

```
#include <sys/types.h>
#include <unistd.h>
int  chown( const char *path, uid_t owner, gid_t group );
int fchown( int fd, uid_t owner, gid_t group );
int lchown( const char *path, uid_t owner, gid_t group );
```

- The owner of the file specified by *path* or by *fd*.
- Only the superuser may change the owner of a file.
- The owner of a file may change the group of the file to any group of which that owner is a member.

Files with Multiple Names

- Any UNIX file can be identified by more than one name
- That is, several pathnames can lead to the same physical collection of bits
- Called a *hard link*
- each file has a *link count* associated with it

```
#include <unistd.h>
int link (const char *orig_path, const char *new_path);
```

- *orig_path* is the existing path to a file
- *new_path* will be a new link to the file
- the *link count* for the file will be incremented by 1
- returns 0 if successful and -1 if it fails

unlink revisited

#include <unistd.h>
int unlink(const char *pathname);

- It removes the link pointed to by *pathname*
- It also decreases the *link count* by 1
- If the *link count* is 0, the file blocks are released to the system

#include <stdio.h>
int rename(const char *oldpath, const char *newpath);

- *oldpath* is renamed with *newpath* (link count set appropriately)
- If *newpath* exists, it is removed before being set to *oldpath*

Symbolic Links

- Two limitations to hard links are:
  1. cannot create a link to directory and
  2. cannot create a link across file systems
- A symbolic link is a file in its own right that simply holds the pathname to another file or directory (a pointer)

```
#include <unistd.h>
int symlink (const char *realname, const char *symname);
```

• upon completion, *symname* is created and points to *realname*
• does not effect *link count*
• returns 0 if successful and -1 if it fails (like symname exists)

Links – Directory entries

- A link, created with the command ln, create new entries into a directory.
- These entries POINT to the inode of the original file.
- A “hard link” relates to a link which points to an inode on the current file system
- A “soft link” is used when the file we want to point to belongs on a different file system.

Creating links using ln

- Creating a hard link: (syntax: ln real_filename link_name)
  - `ln topten.sh HeavyHitters`
  - `ln $HOME/.bashrc my_login_script`

- Creating a soft link:
  - `ln -s /home/common/warn-20150517  test_warn_file`

Sticky Bit

Octal	*Meaning*
`01000`	Save text image on execution. Symbolic: `--- --- --t`

- This specifies that the program code should stay resident in memory after termination.
  - this makes the start-up of the next execution faster
- Obsolete due to virtual memory.

The superuser

- Most sys. admin. tasks can only be done by the *superuser* (also called the *root* user)
- Superuser
  - has access to all files/directories on the system
  - can override permissions
  - owner of most system files
- Shell command: `su <username>`
  - Set current user to superuser or another user with proper password access