ELEC462003 System Programming

Directories and File Properties

Prof. Seokin Hong

Kyungpook National University

Fall 2019

Objectives

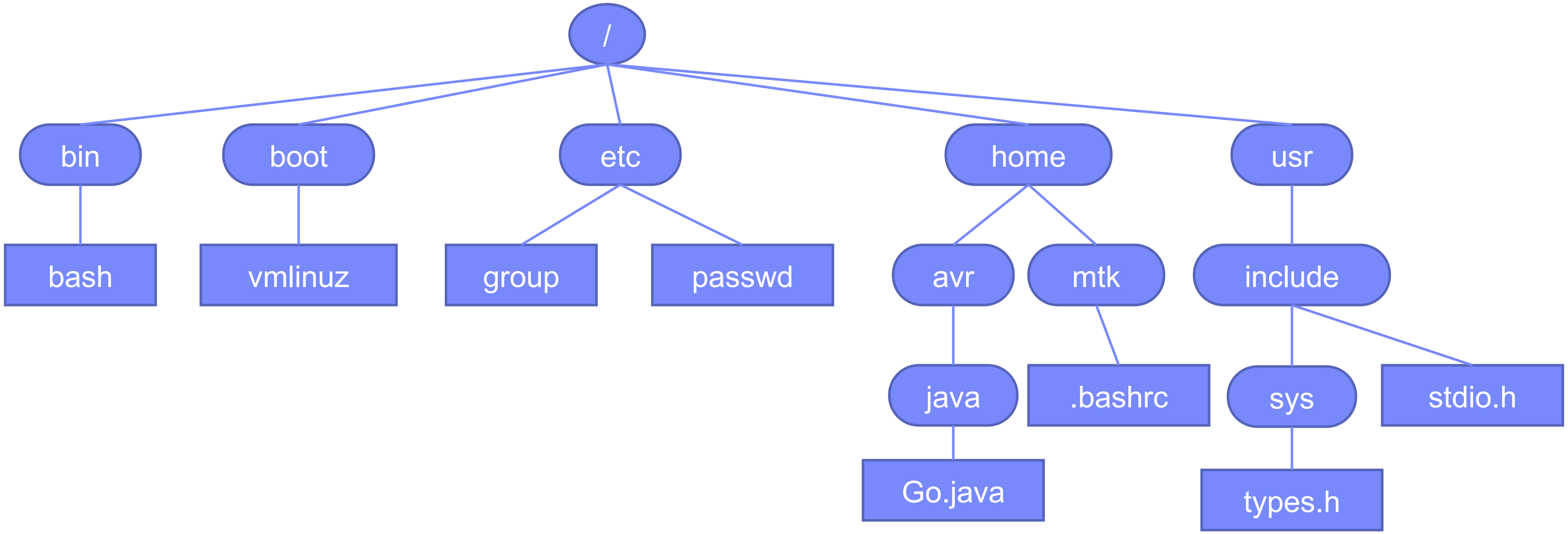
- **Ideas and Skills**
  - A directory is a list of files
  - How to read a directory
  - Types of files and how to determine the type of a file
  - Properties of files and how to determine properties of a file
  - Bit sets and bit masks
  - User and group ID numbers
- **System Calls and Functions**
  - opendir, readdir, closedir, seekdir
  - stat
  - chmod, chown, utime
  - rename
- **Commands**
  - ls

Agenda

- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

Files and Directories (Review)

- **Single hierarchical directory structure**

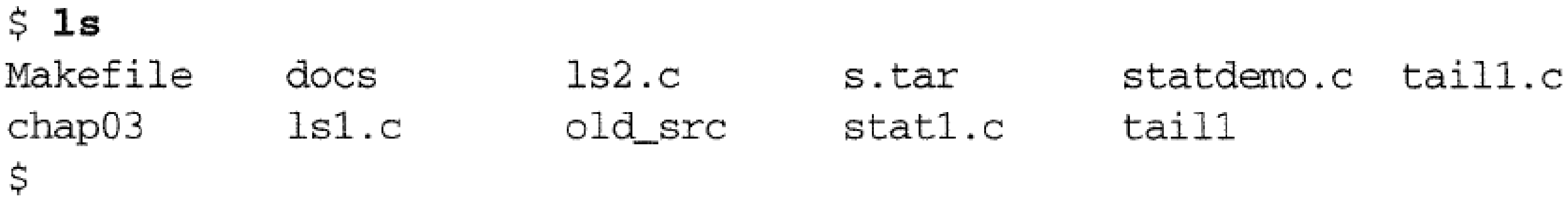


- **File types**
  - Regular(plain), device, pipe, socket, dir, sym link

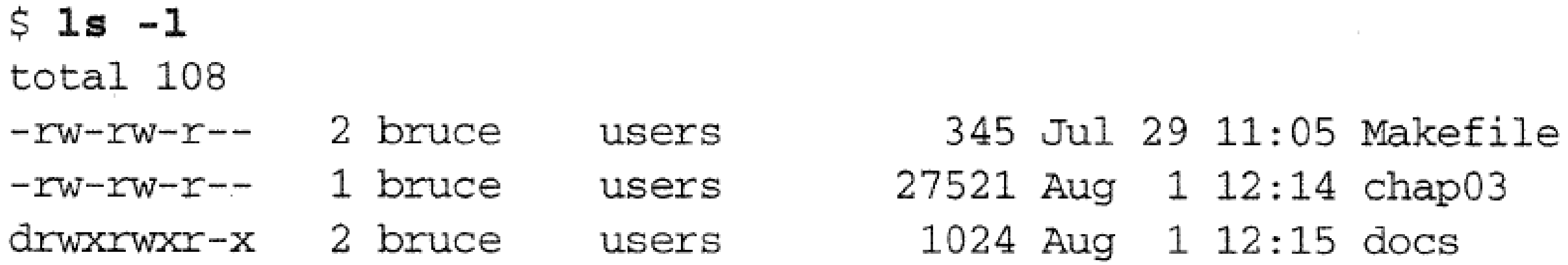
- **Directory**
  - Special file that has the table of filenames and the reference to those files

ls

- lists names of files and reports file attributes
- Example:



```
$ ls
Makefile     docs        ls2.c       s.tar       statdemo.c  tail1.c
chap03       ls1.c       old_src     stat1.c     tail1
$
```

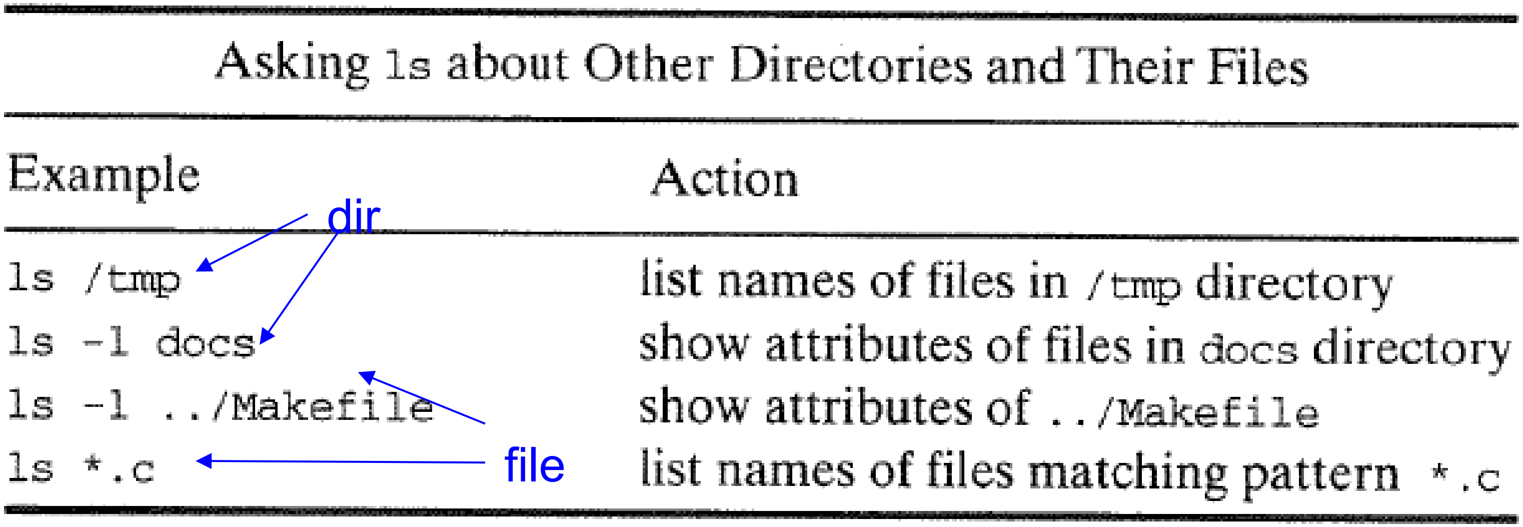


```
$ ls -l
total 108
-rw-rw-r--   2 bruce    users         345 Jul 29 11:05 Makefile
-rw-rw-r--   1 bruce    users       27521 Aug  1 12:14 chap03
drwxrwxr-x   2 bruce    users        1024 Aug  1 12:15 docs
```

Type&permission   links   owner   group   size   modified-date/time   name

ls

- Listing other directories and reporting on other files

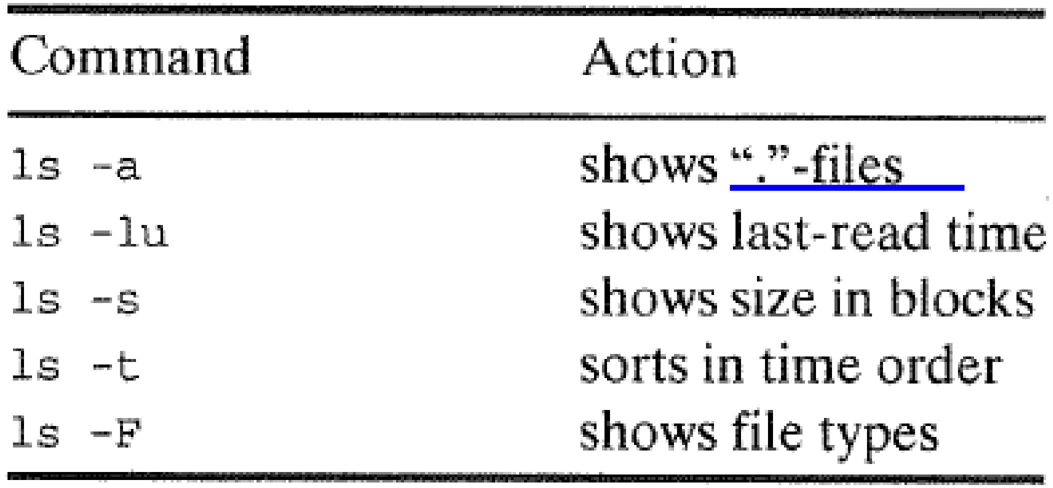


Asking `ls` about Other Directories and Their Files

Example	Action
`ls /tmp`	list names of files in `/tmp` directory
`ls -l docs`	show attributes of files in `docs` directory
`ls -l ../Makefile`	show attributes of `../Makefile`
`ls *.c`	list names of files matching pattern `*.c`

Popular Command-Line Options

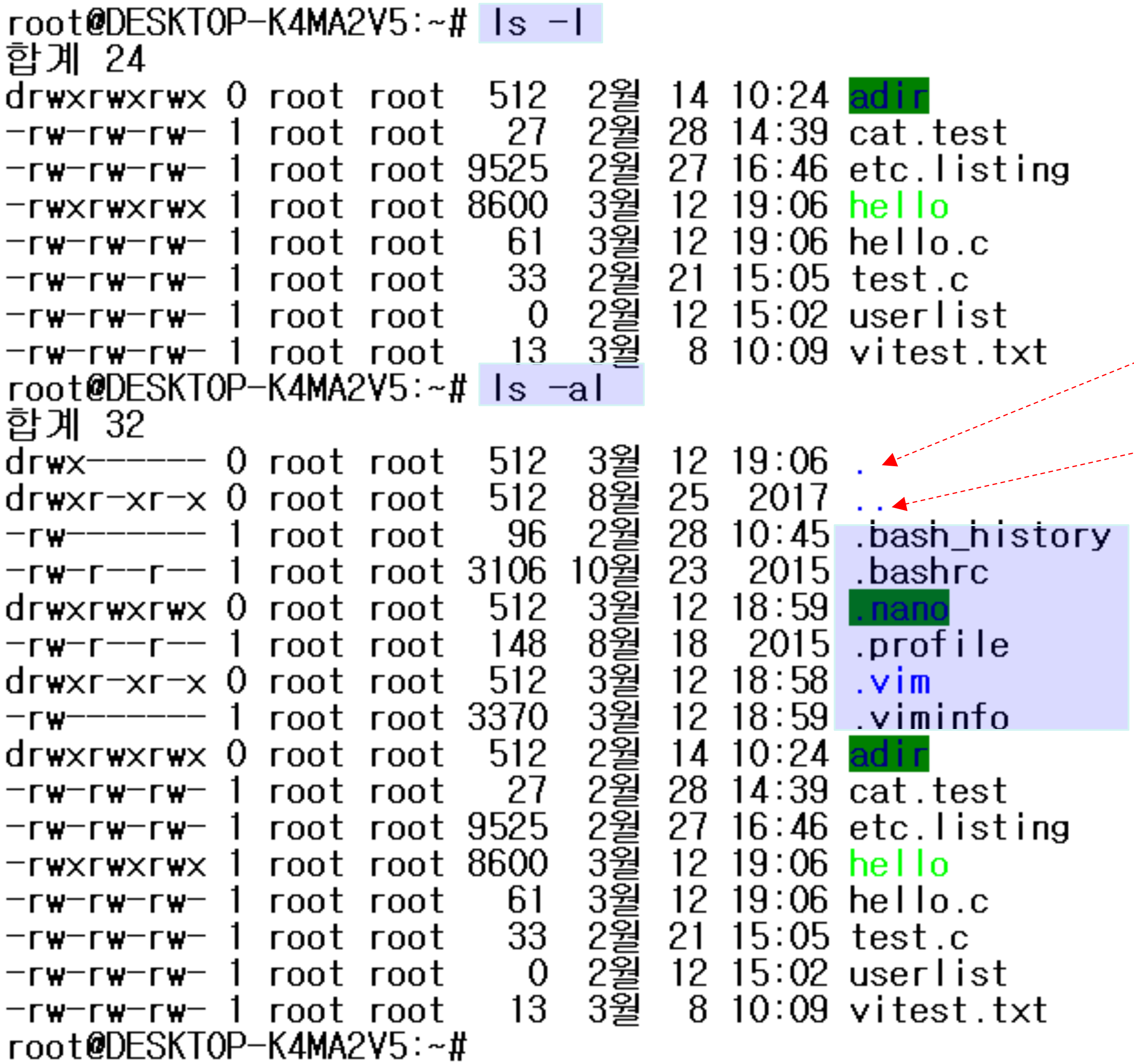
- Options:



Command	Action
`ls -a`	shows "."-files
`ls -lu`	shows last-read time
`ls -s`	shows size in blocks
`ls -t`	sorts in time order
`ls -F`	shows file types

```
ls -al
```

- A remark on Dot-Files (hidden file)
  - **ls** does not list the name of a file if the first character of the filename is a dot.
  - Some programs use dot filenames in a user's home directory to store user preferences.



```
root@DESKTOP-K4MA2V5:~# ls -l
합계 24
drwxrwxrwx 0 root root  512  2월  14 10:24 adir
-rw-rw-rw- 1 root root   27  2월  28 14:39 cat.test
-rw-rw-rw- 1 root root 9525  2월  27 16:46 etc.listing
-rwxrwxrwx 1 root root 8600  3월  12 19:06 hello
-rw-rw-rw- 1 root root   61  3월  12 19:06 hello.c
-rw-rw-rw- 1 root root   33  2월  21 15:05 test.c
-rw-rw-rw- 1 root root    0  2월  12 15:02 userlist
-rw-rw-rw- 1 root root   13  3월   8 10:09 vitest.txt
root@DESKTOP-K4MA2V5:~# ls -al
합계 32
drwx------ 0 root root  512  3월  12 19:06 .
drwxr-xr-x 0 root root  512  8월  25  2017 ..
-rw------- 1 root root   96  2월  28 10:45 .bash_history
-rw-r--r-- 1 root root 3106 10월  23  2015 .bashrc
drwxrwxrwx 0 root root  512  3월  12 18:59 .nano
-rw-r--r-- 1 root root  148  8월  18  2015 .profile
drwxr-xr-x 0 root root  512  3월  12 18:58 .vim
-rw------- 1 root root 3370  3월  12 18:59 .viminfo
drwxrwxrwx 0 root root  512  2월  14 10:24 adir
-rw-rw-rw- 1 root root   27  2월  28 14:39 cat.test
-rw-rw-rw- 1 root root 9525  2월  27 16:46 etc.listing
-rwxrwxrwx 1 root root 8600  3월  12 19:06 hello
-rw-rw-rw- 1 root root   61  3월  12 19:06 hello.c
-rw-rw-rw- 1 root root   33  2월  21 15:05 test.c
-rw-rw-rw- 1 root root    0  2월  12 15:02 userlist
-rw-rw-rw- 1 root root   13  3월   8 10:09 vitest.txt
root@DESKTOP-K4MA2V5:~#
```

Current directory

Parent directory

- Dot-file (hidden file)
- Dot-files at home directory are typically used for user preferences of programs

So, what does ls do?

- **ls** does two things
  1. Lists the contents of directories
  2. Displays information about files

- We need to learn:
  1. How to list the contents of a directory
  2. How to obtain and display properties of a file
  3. How to determine if a name refers to a file or a directory

Agenda

- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

File System Tree

- The **disk** is organized as **a tree of directories**, each of which contains files or directories.

- The commands **cd, pwd, ls** allow us to explore a file system

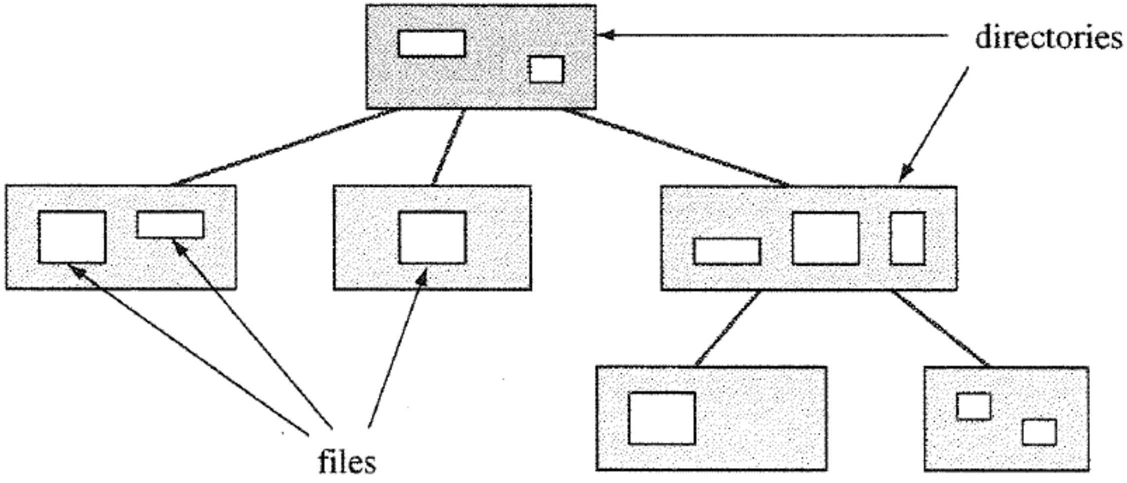
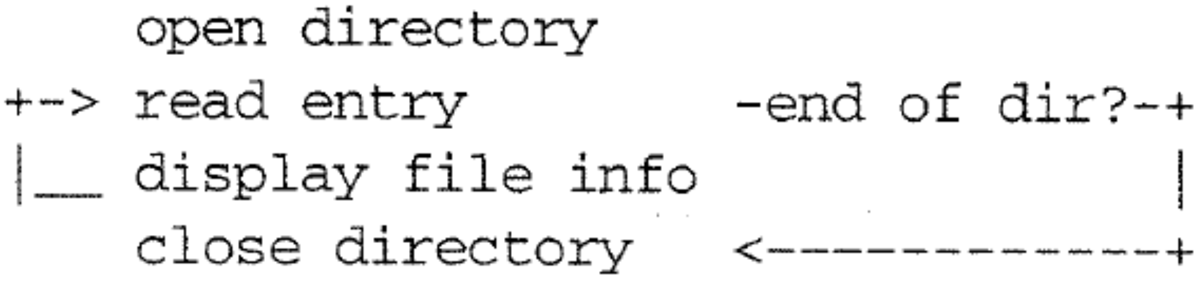


FIGURE 3.1

A tree of directories.

How Dos ls Work?

- Outline :



```
       open directory
  +--> read entry          -end of dir?-+
  |__  display file info                |
       close directory    <-------------+
```

- It looks just like the logic for who!

- Difference?
  - The main difference is that the `who` opens and reads *from a file (utmp, wtmp)*
  - **`ls`** opens and reads its data *from a directory*.

What is a Directory?

- A directory is a kind of file that contains a list of names of files and directories.

- Unlike a regular file, a directory never empty
  - Every directory contains two specific items: **. and ..**
  - **dot(.)** is the name of the current directory,
  - **dotdot(..)** is the name of the directory one level up.

Do open, read, and close work for directories?

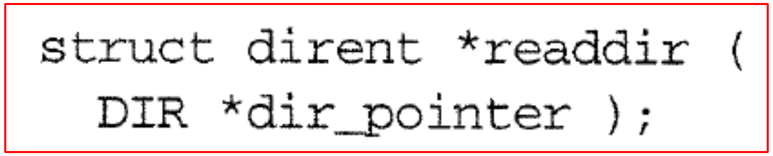
- Answer 1: on old versions of Unix, that was the only way
  - On some versions of Unix, you still can, but not for all directories

- Answer 2: It is a bad idea to use open, read and close to list contents of directory. Why?
  - Unix allows various disk formats to appear as part of a single tree.
    - It supports Mac HFS, FAT, FAT32, lots of Unix flavors;
  - Thus, using **read** to process each type would require knowing the format of the records for each type of directory

How do I read a Directory? (I)

- We read the entires by calling readdir()

- Each readdir() call returns a pointer to the next record, a variable of type struct dirent



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

How do I read a Directory? (II)

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

```
opendir(char *)
  creates a connection,
  returns a DIR *

readdir(DIR *)
  reads next record,
  returns a pointer
  to a struct dirent

closedir(DIR *)
  closes a connection
```

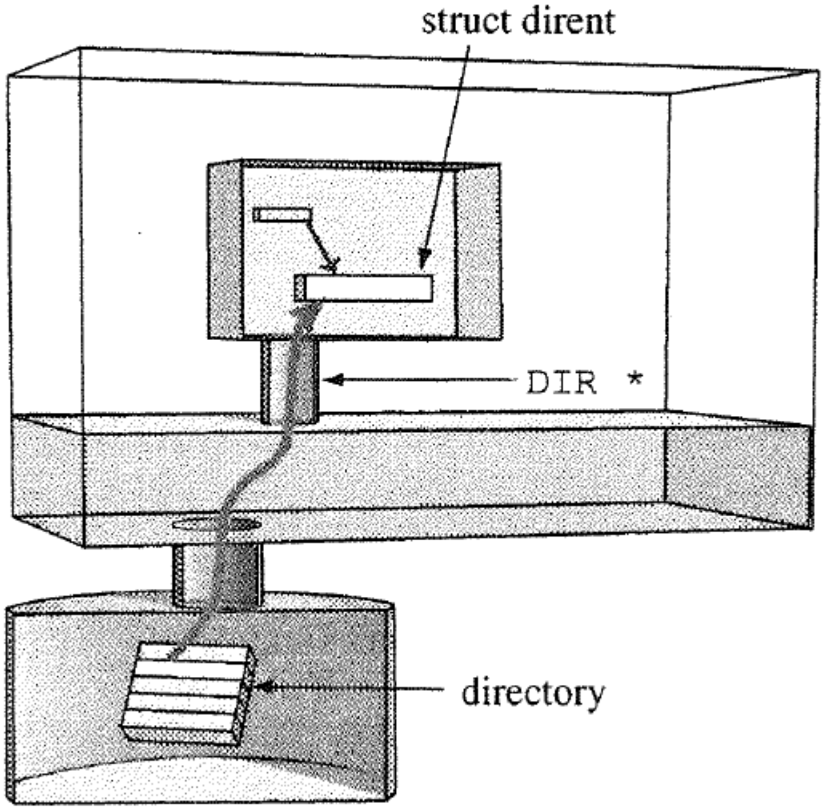
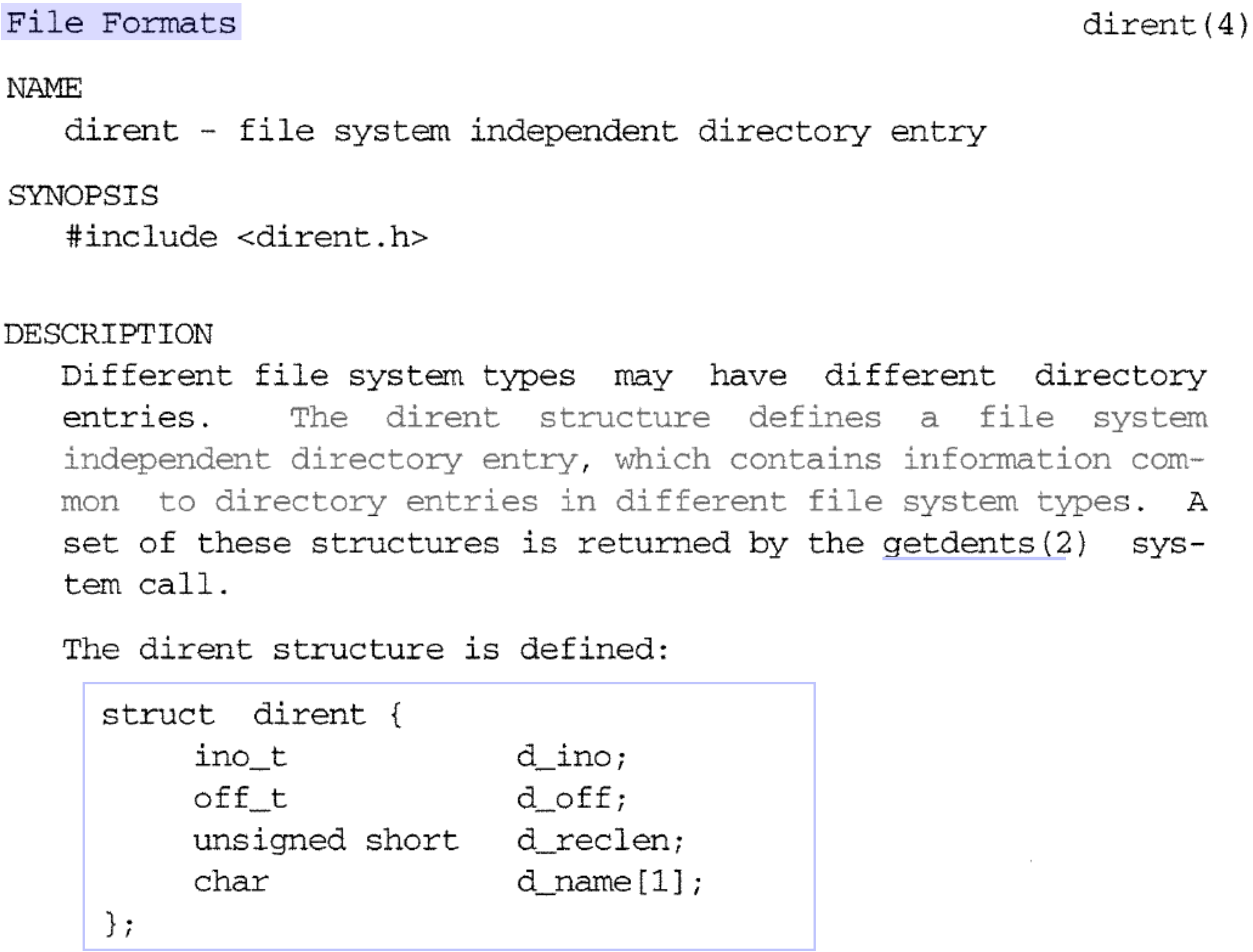


FIGURE 3.2

Reading entries from a directory.

How do I read a Directory? (III)



File Formats dirent(4)

NAME

dirent - file system independent directory entry

SYNOPSIS

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

DESCRIPTION

Different file system types may have different directory entries. The dirent structure defines a file system independent directory entry, which contains information common to directory entries in different file system types. A set of these structures is returned by the getdents(2) system call.

The dirent structure is defined:

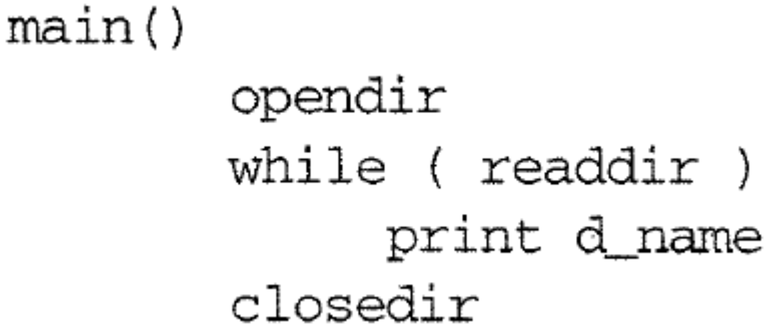
```
struct  dirent {
     ino_t              d_ino;
     off_t              d_off;
     unsigned short     d_reclen;
     char               d_name[1];
};
```

Agenda

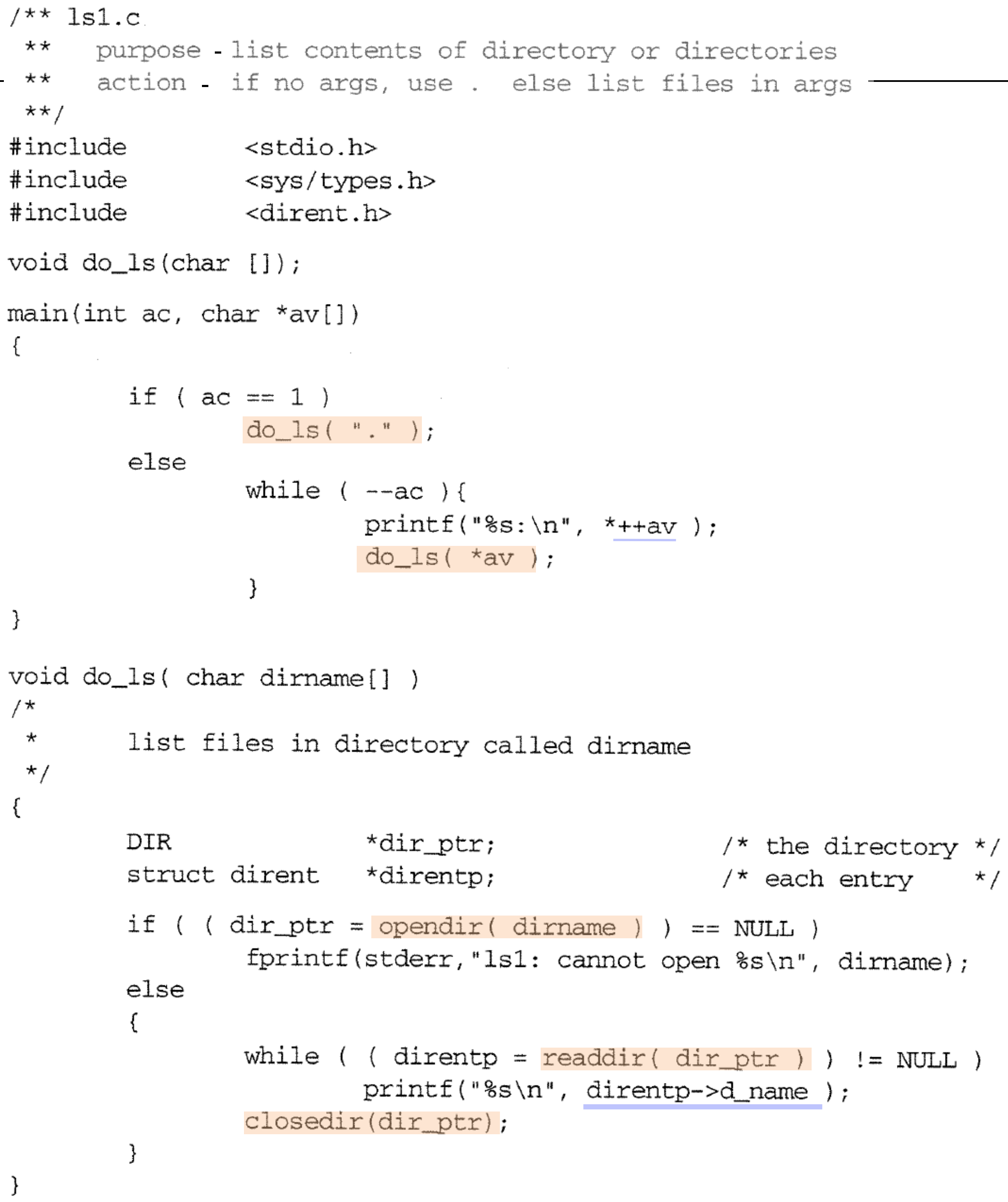
- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

Writing ls1.c

- Logic for listing a directory:

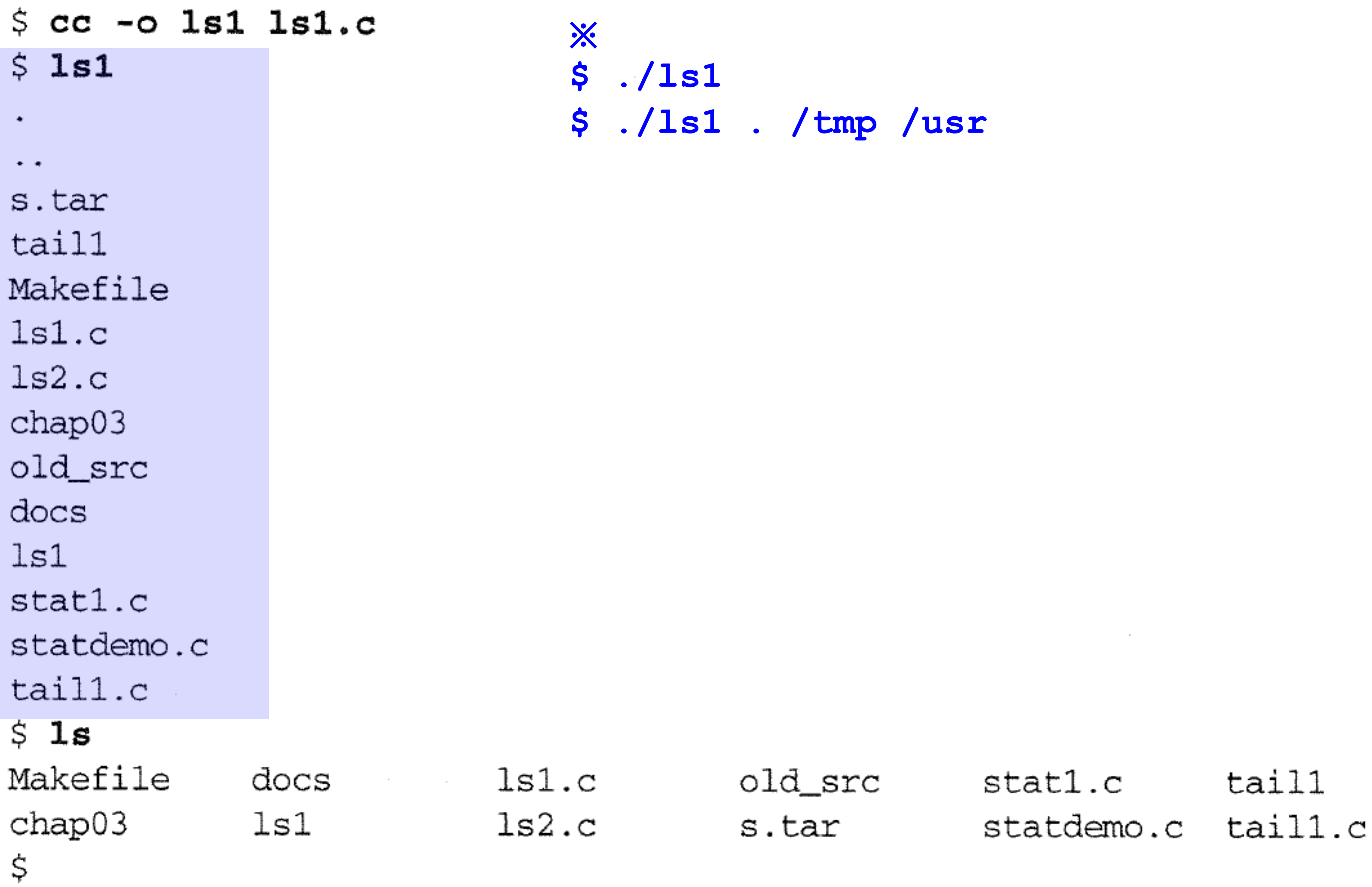


```
main()
        opendir
        while ( readdir )
             print d_name
        closedir
```



```
/** ls1.c
 **   purpose - list contents of directory or directories
 **   action - if no args, use .  else list files in args
 **/
#include        <stdio.h>
#include        <sys/types.h>
#include        <dirent.h>

void do_ls(char []);

main(int ac, char *av[])
{

        if ( ac == 1 )
                do_ls( "." );
        else
                while ( --ac ){
                        printf("%s:\n", *++av );
                        do_ls( *av );
                }
}

void do_ls( char dirname[] )
/*
 *      list files in directory called dirname
 */
{
        DIR             *dir_ptr;               /* the directory */
        struct dirent   *direntp;               /* each entry    */

        if ( ( dir_ptr = opendir( dirname ) ) == NULL )
                fprintf(stderr,"ls1: cannot open %s\n", dirname);
        else
        {
                while ( ( direntp = readdir( dir_ptr ) ) != NULL )
                        printf("%s\n", direntp->d_name );
                closedir(dir_ptr);
        }
}
```

- Compile and run it:



```
$ cc -o ls1 ls1.c
$ ls1
.
..
s.tar
tail1
Makefile
ls1.c
ls2.c
chap03
old_src
docs
ls1
stat1.c
statdemo.c
tail1.c
$ ls
Makefile    docs    ls1.c    old_src    stat1.c      tail1
chap03      ls1     ls2.c    s.tar      statdemo.c   tail1.c
$
```

※

```
$ ./ls1
$ ./ls1 . /tmp /usr
```

Agenda

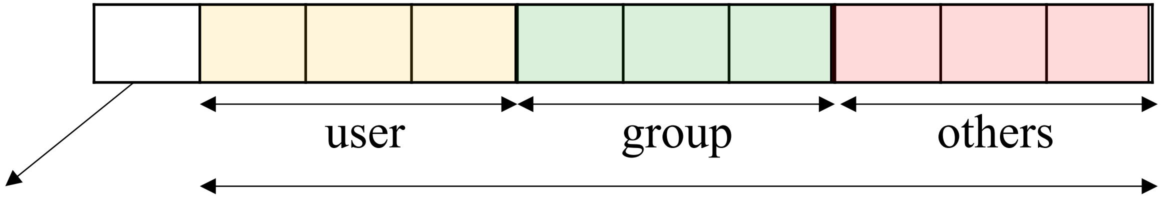
- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

What Does "`ls -l`" Do?

- **ls** does two different types of things
  - lists directories and files
  - display information about directories and files

```
$ ls -l
total 108
-rw-rw-r--   2 bruce    users          345 Jul 29 11:05 Makefile
-rw-rw-r--   1 bruce    users        27521 Aug  1 12:14 chap03
drwxrwxr-x   2 bruce    users         1024 Aug  1 12:15 docs
```

type and permission   links owner   group   size   last-modified time   name



user

group

others

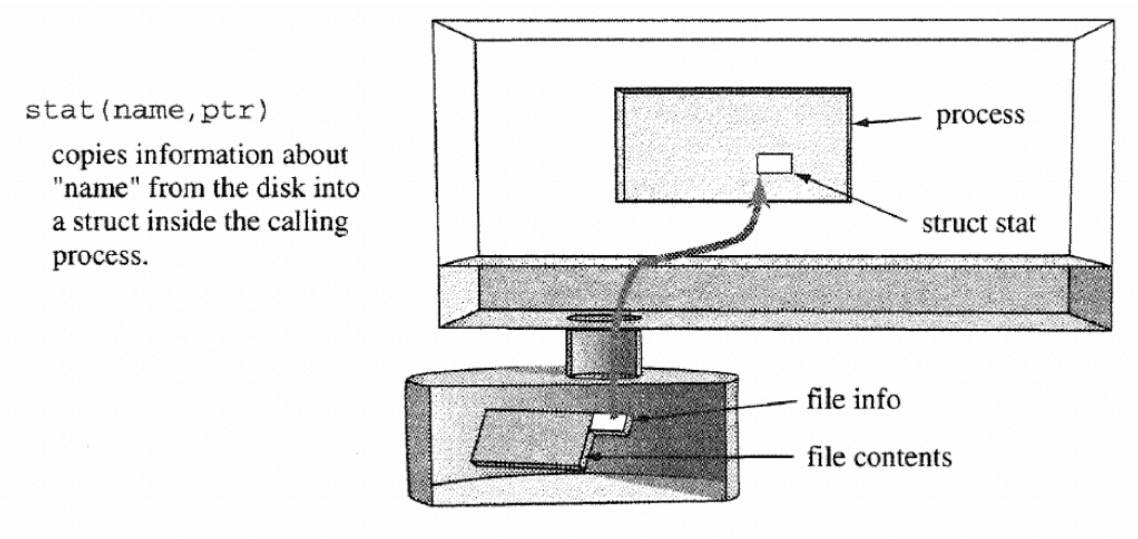
type of file

permission mode

- : regular file
d : directory

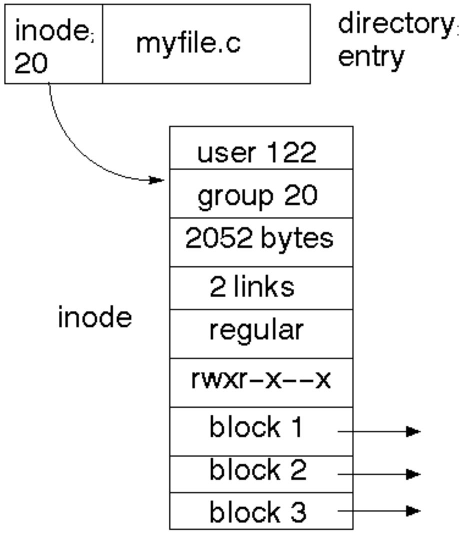
How Does “ls –l” work? (I)

- How can we get information (status/properties) about a file?
  - o **stat** system call is used to retrieve file status
- How does **stat** work:
  - o The process defines a buffer of type struct state
  - o And then asks the kernel to copy file information from the disk to the buffer



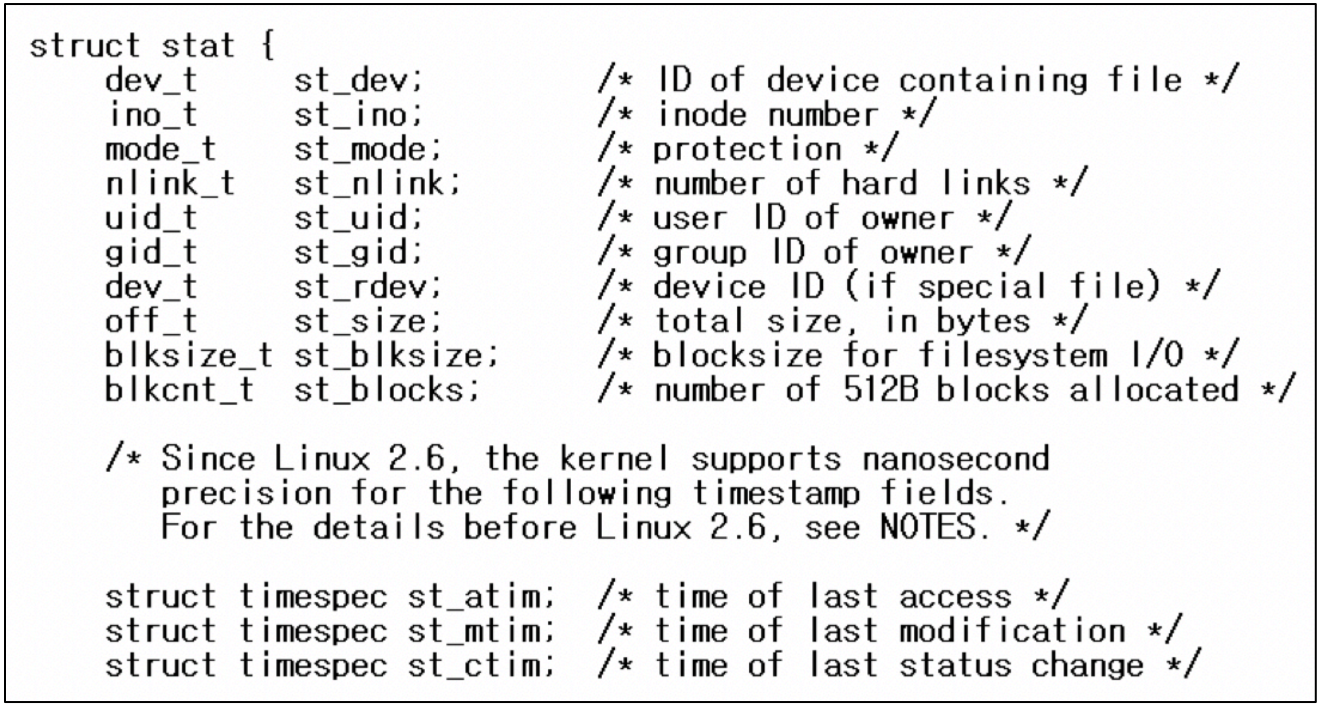
`stat(name,ptr)`

copies information about "name" from the disk into a struct inside the calling process.



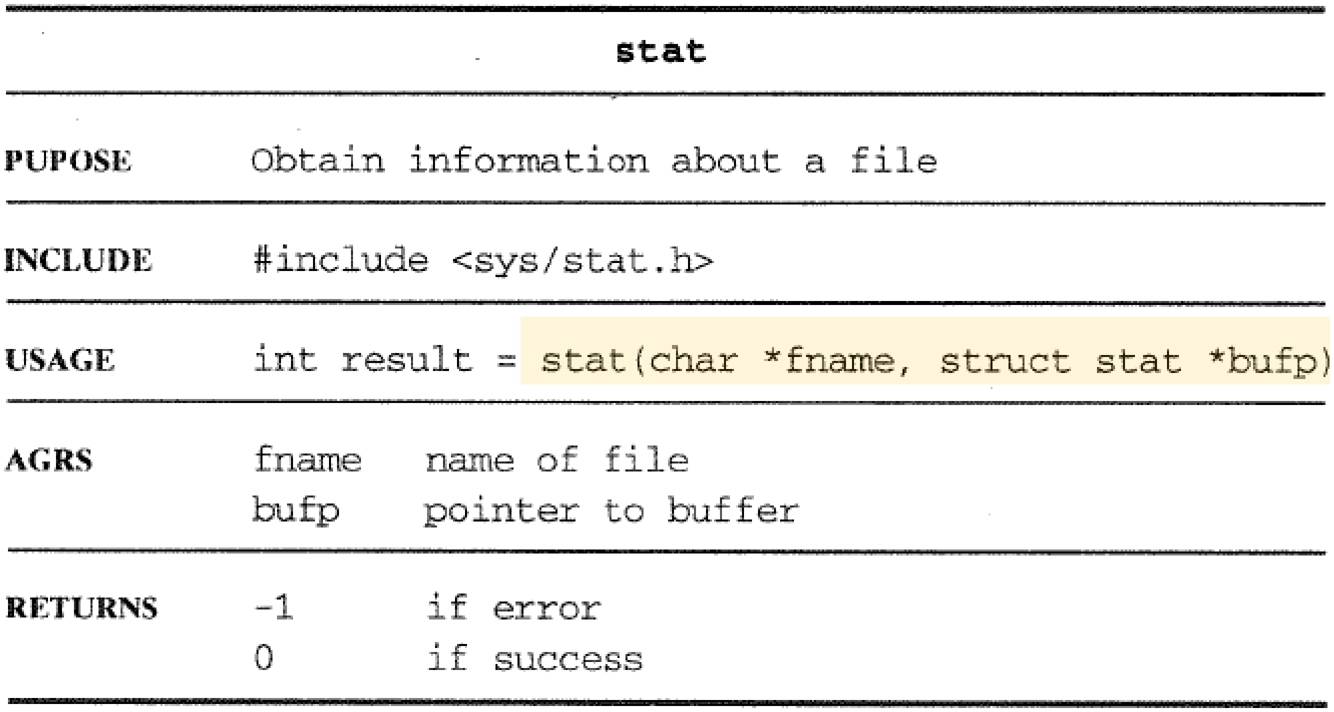
How Does “ls –l” work? (II)

- **struct stat**
  - Defined in /usr/include/sys/stat.h



```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* inode number */
    mode_t    st_mode;        /* protection */
    nlink_t   st_nlink;       /* number of hard links */
    uid_t     st_uid;         /* user ID of owner */
    gid_t     st_gid;         /* group ID of owner */
    dev_t     st_rdev;        /* device ID (if special file) */
    off_t     st_size;        /* total size, in bytes */
    blksize_t st_blksize;     /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;      /* number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* time of last access */
    struct timespec st_mtim;  /* time of last modification */
    struct timespec st_ctim;  /* time of last status change */
```

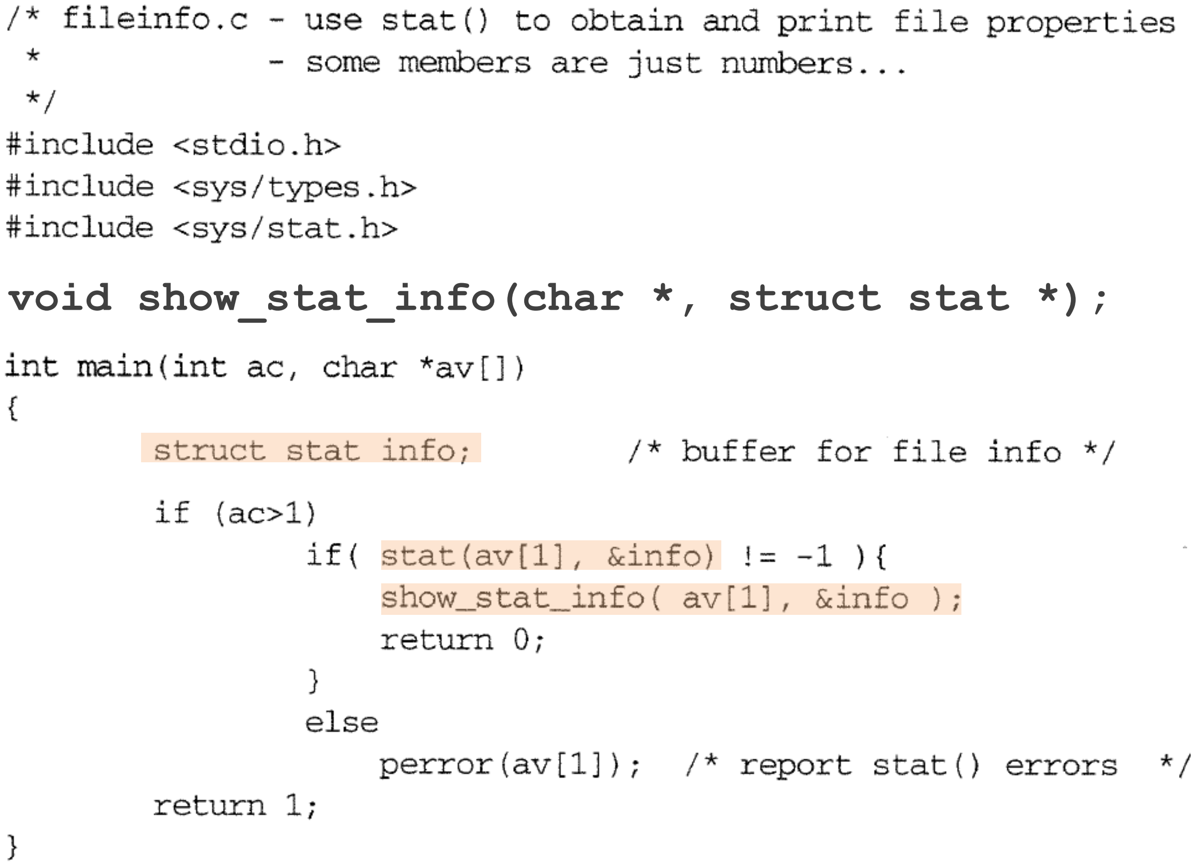
How Does “ls –l” work? (III)



	stat
PUPOSE	Obtain information about a file
INCLUDE	`#include <sys/stat.h>`
USAGE	`int result = stat(char *fname, struct stat *bufp)`
AGRS	fname name of file bufp pointer to buffer
RETURNS	-1 if error 0 if success

Writing fileinfo.c

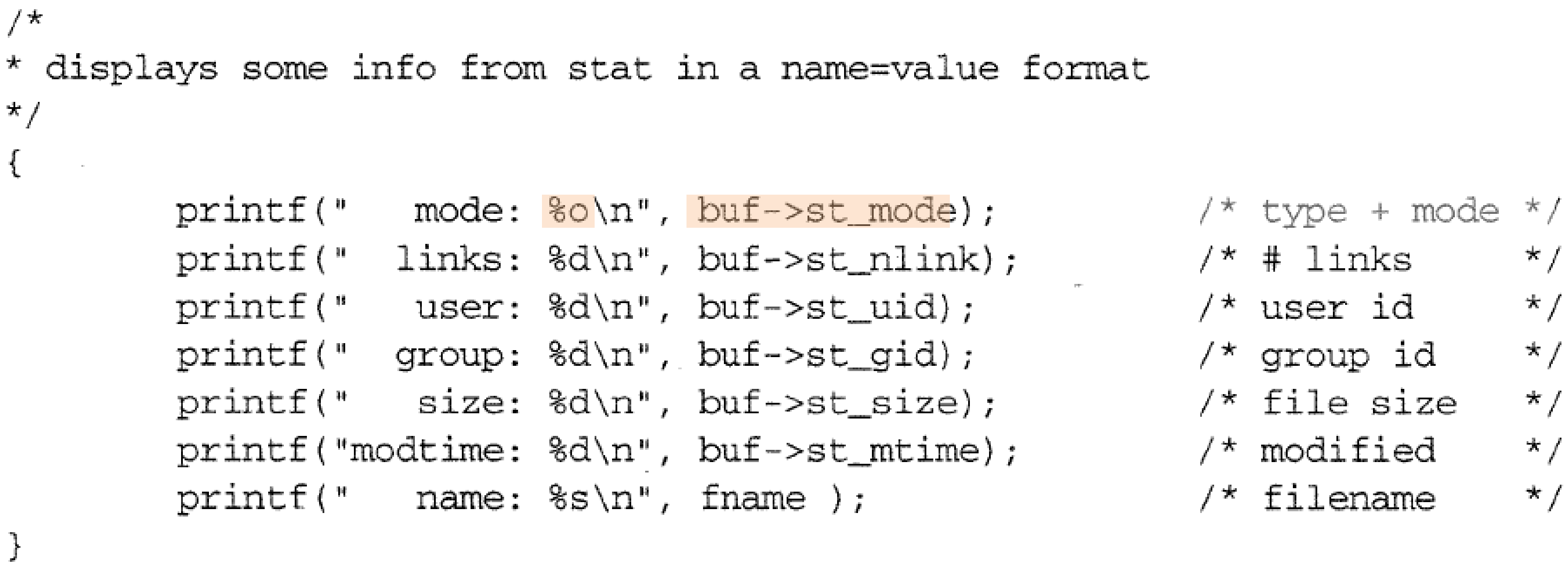
- Use stat to get file info for that name
- Display the items in the struct



```
/* fileinfo.c - use stat() to obtain and print file properties
 *             - some members are just numbers...
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

void show_stat_info(char *, struct stat *);

int main(int ac, char *av[])
{
        struct stat info;          /* buffer for file info */

        if (ac>1)
                if( stat(av[1], &info) != -1 ){
                    show_stat_info( av[1], &info );
                    return 0;
                }
                else
                    perror(av[1]);  /* report stat() errors   */
        return 1;
}
```

Writing fileinfo.c

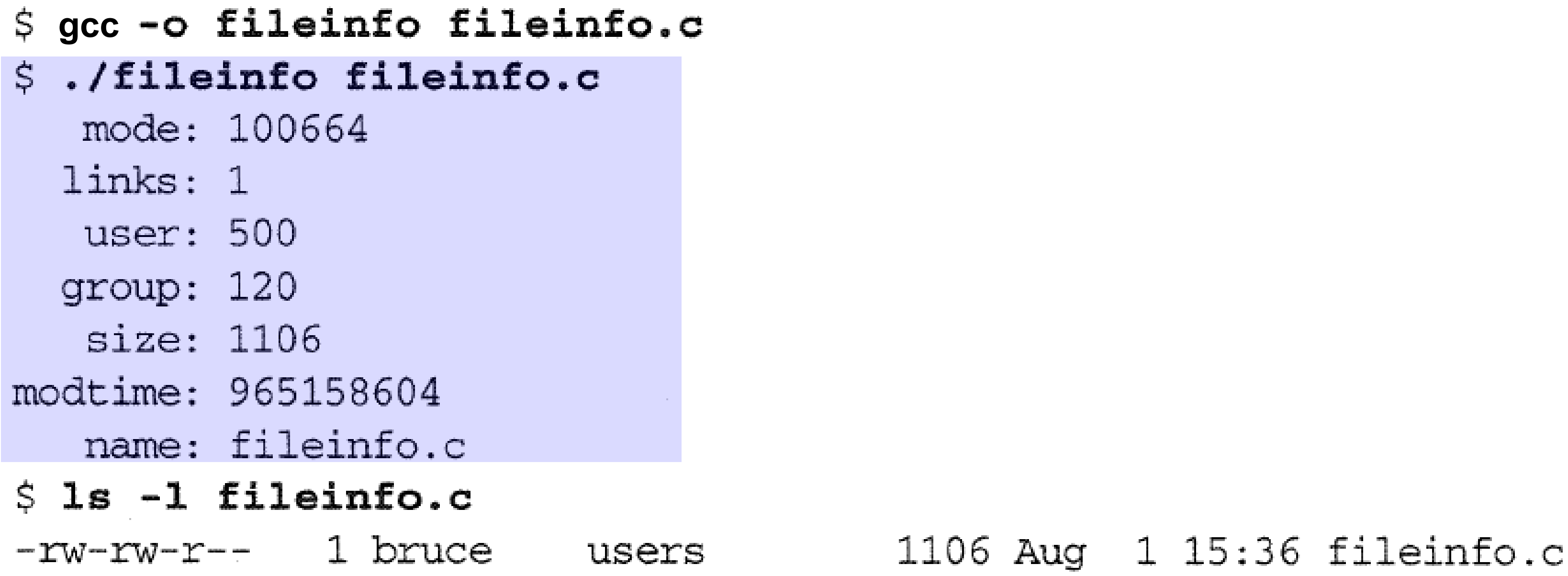
```
void show_stat_info(char *fname, struct stat *buf)
```



```
/*
* displays some info from stat in a name=value format
*/
{
        printf("   mode: %o\n", buf->st_mode);          /* type + mode */
        printf("  links: %d\n", buf->st_nlink);         /* # links     */
        printf("   user: %d\n", buf->st_uid);           /* user id     */
        printf("  group: %d\n", buf->st_gid);           /* group id    */
        printf("   size: %d\n", buf->st_size);          /* file size   */
        printf("modtime: %d\n", buf->st_mtime);         /* modified    */
        printf("   name: %s\n", fname );                /* filename    */
}
```

Writing fileinfo.c

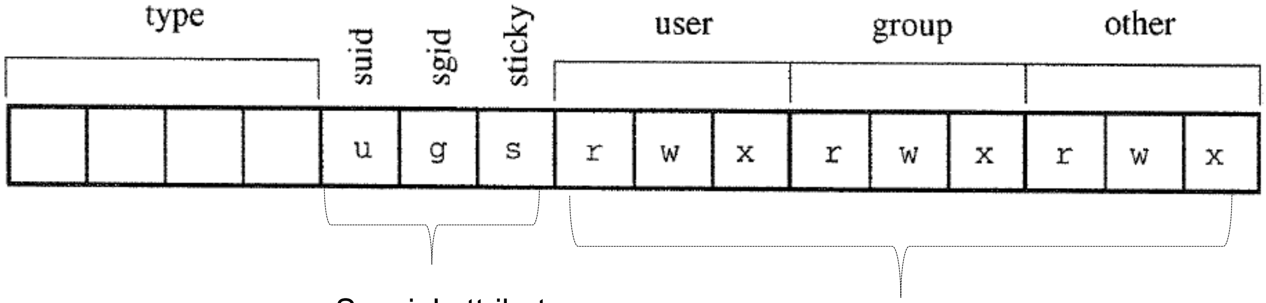
- Compile and run it :



```
$ gcc -o fileinfo fileinfo.c
$ ./fileinfo fileinfo.c
   mode: 100664
  links: 1
   user: 500
  group: 120
   size: 1106
modtime: 965158604
   name: fileinfo.c
$ ls -l fileinfo.c
-rw-rw-r--   1 bruce    users        1106 Aug  1 15:36 fileinfo.c
```

Converting file mode to a string

- File type and permission bits are stored in the **st_mode member**

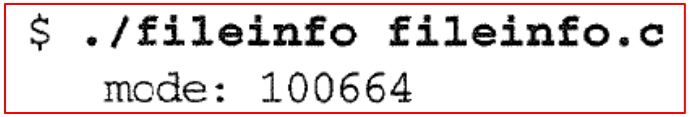


Special attribute

Permission bits

- **Type**: file types
  - 4 bits means 16 possible patterns.
  - Each pattern can correspond to a file type.

- **Permission bits** :
  - Access permission of user, group, others for the file
  - 1 indicates the permission is granted
  - 0 indicates the permission is denied

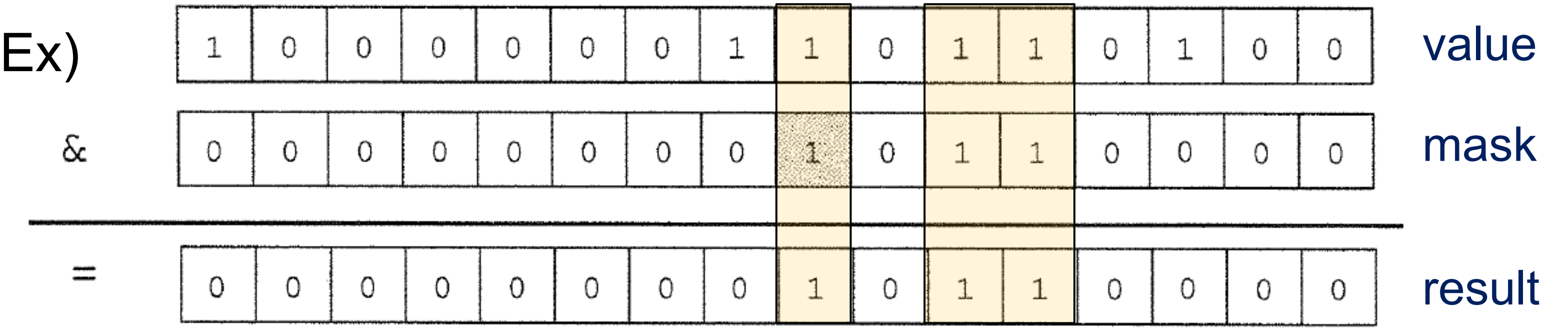


```
$ ./fileinfo fileinfo.c
    mode: 100664
```

How to read subfields: Masking

- How do we examine a bit or sub-field?
  - ex) 100664 (base 8) → -rw-rw-r--

- Use “bitwise AND (&)” to MASK
  - Ex)



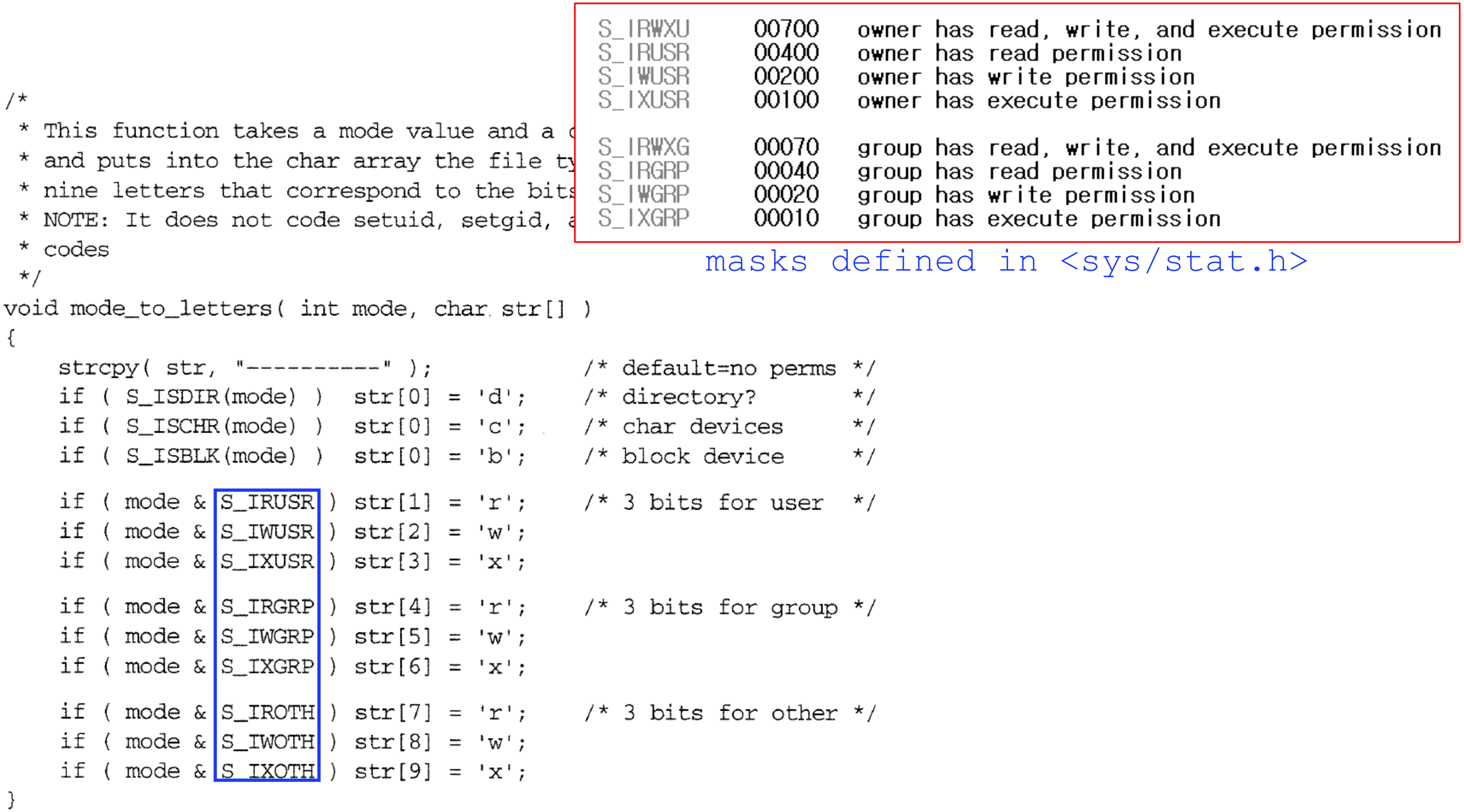
	1	0	0	0	0	0	0	1	1	0	1	1	0	1	0	0	value
&	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	mask
=	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	result

FIGURE 3.6

Applying a bitmask.

Using Masking to decode permission bits

- 100664 (base 8) → -rw-rw-r--



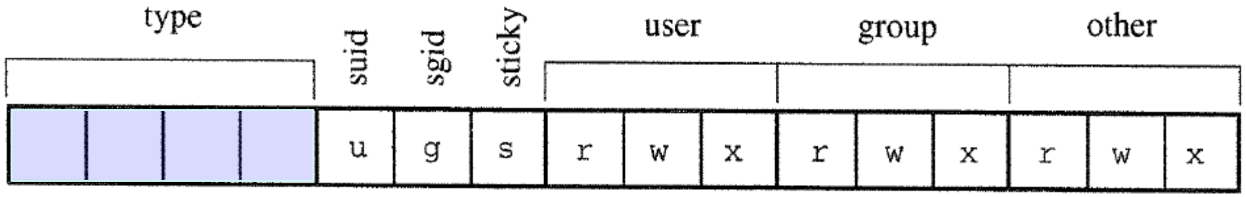
S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission

masks defined in <sys/stat.h>

```
/*
 * This function takes a mode value and a c
 * and puts into the char array the file ty
 * nine letters that correspond to the bits
 * NOTE: It does not code setuid, setgid, a
 * codes
 */
void mode_to_letters( int mode, char str[] )
{
    strcpy( str, "----------" );          /* default=no perms */
    if ( S_ISDIR(mode) )  str[0] = 'd';    /* directory?       */
    if ( S_ISCHR(mode) )  str[0] = 'c';    /* char devices     */
    if ( S_ISBLK(mode) )  str[0] = 'b';    /* block device     */

    if ( mode & S_IRUSR ) str[1] = 'r';    /* 3 bits for user  */
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';    /* 3 bits for group */
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';    /* 3 bits for other */
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}
```

Using Masking to decode file types

- Mask and file types defined in <sys/stat.h>



1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 mask

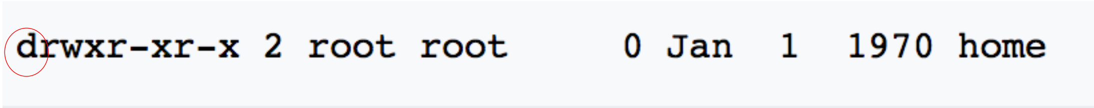
```
#define S_IFMT           0170000          /* type of file */
#define    S_IFREG       0100000          /*    regular */
#define    S_IFDIR       0040000          /*    directory */
#define    S_IFBLK       0060000          /*    block special */
#define    S_IFCHR       0020000          /*    character special */
#define    S_IFIFO       0010000          /*    fifo */
#define    S_IFLNK       0120000          /*    symbolic link */
#define    S_IFSOCK      0140000          /*    socket */
```

File types

※ octal

```
if ( (info.st_mode & 0170000) == 0040000 )
        printf("this is a directory.");
```

Using Masking to decode file types

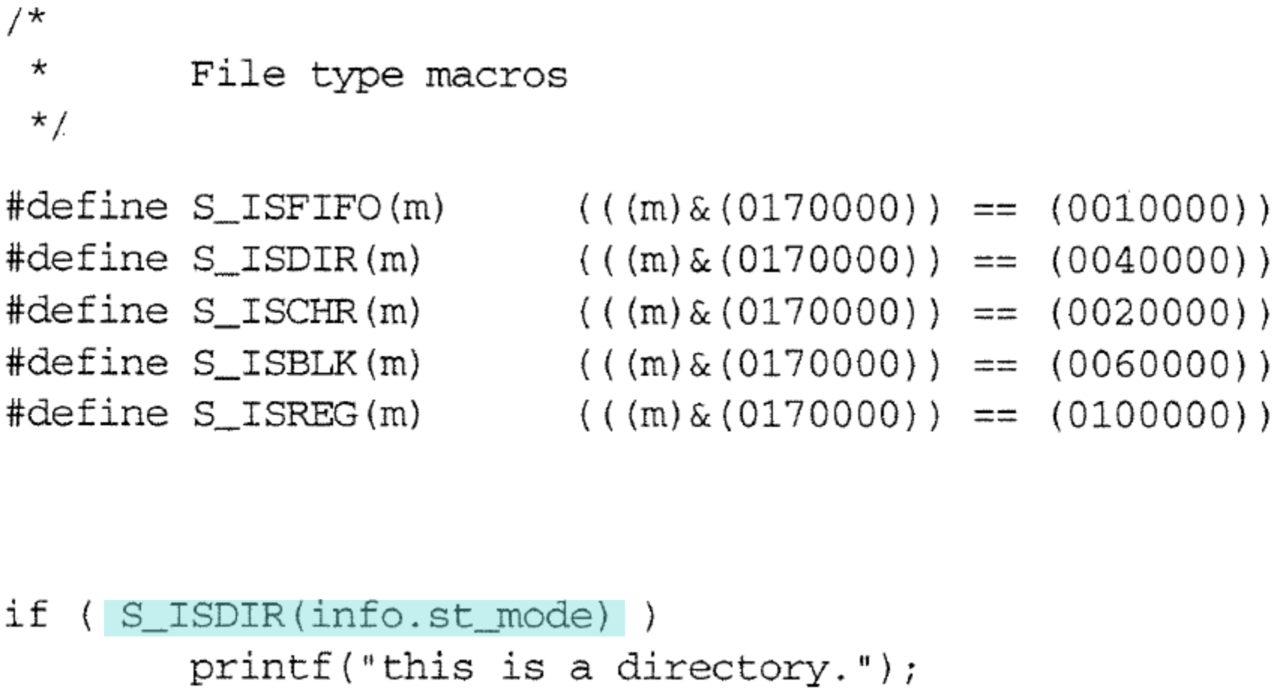


```
drwxr-xr-x 2 root root     0 Jan  1  1970 home
```

- **File types**
  - **Regular** : regular file, marked with **-**
  - **Directory** : directory. marked with **d**
  - **Symbolic link** : a reference to another file. marked with **l**
  - **Socket** : file used for inter-process communication that enable packetized-communication between two processes. communication can extend beyond localhost. marked with an **s**
  - **Block special :** interface that allows an application to interact with a hardware devices. It provides buffered access to the hardware. marked with **b**
  - **Character special :** interface that allows an application to interact with a hardware devices. It provides un-buffered, direct access to the hardware. marked with **c**
  - **FIFO (named pipe) :** file used for inter-process communication within a host. marked with **p**

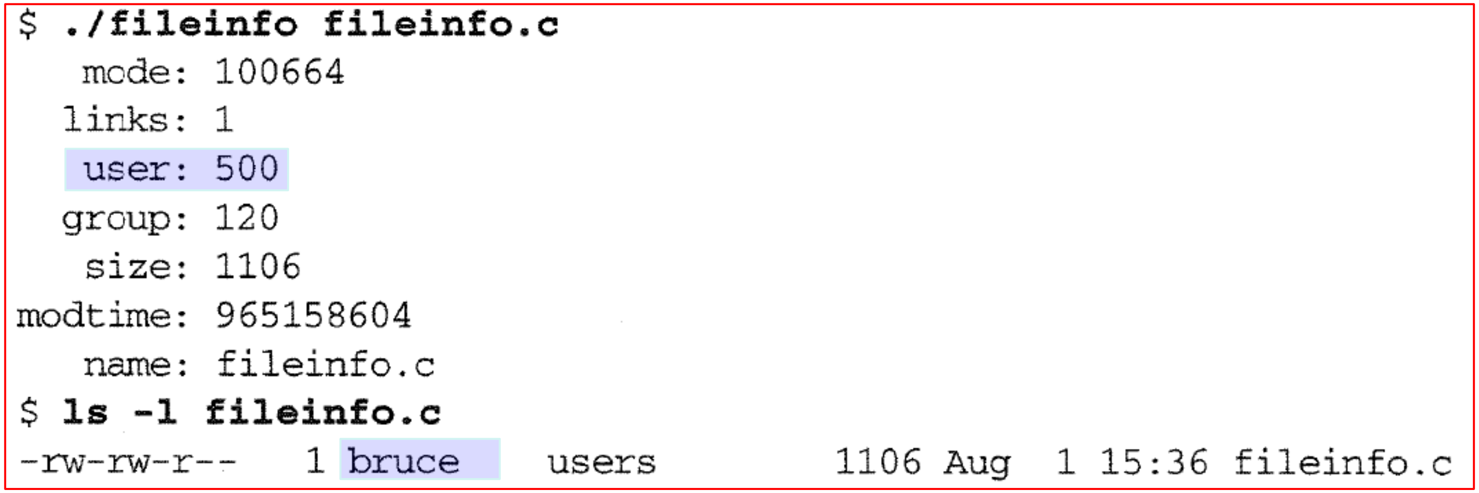
Using Macros to decode file types

- Macros defined in <sys/stat.h>



```
/*
 *      File type macros
 */

#define S_ISFIFO(m)      (((m)&(0170000)) == (0010000))
#define S_ISDIR(m)       (((m)&(0170000)) == (0040000))
#define S_ISCHR(m)       (((m)&(0170000)) == (0020000))
#define S_ISBLK(m)       (((m)&(0170000)) == (0060000))
#define S_ISREG(m)       (((m)&(0170000)) == (0100000))



if ( S_ISDIR(info.st_mode) )
        printf("this is a directory.");
```

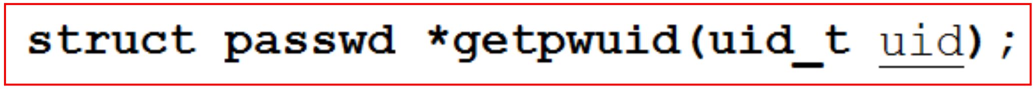
Converting User ID to Strings



```
$ ./fileinfo fileinfo.c
   mode: 100664
  links: 1
   user: 500
  group: 120
   size: 1106
modtime: 965158604
   name: fileinfo.c
$ ls -l fileinfo.c
-rw-rw-r--    1 bruce    users        1106 Aug  1 15:36 fileinfo.c
```

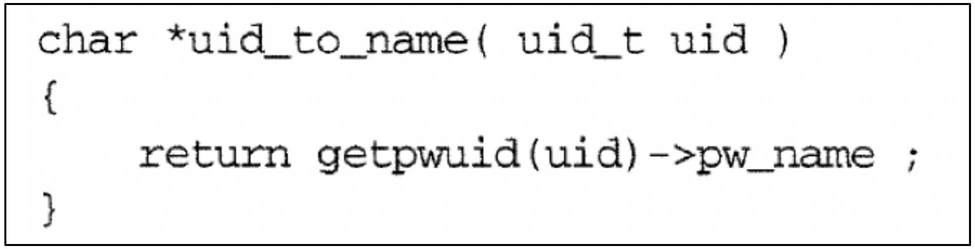
Converting User ID to Strings

- Library function getpwuid() provides access to the complete list of users
  - Defined in /usr/include/pwd.h



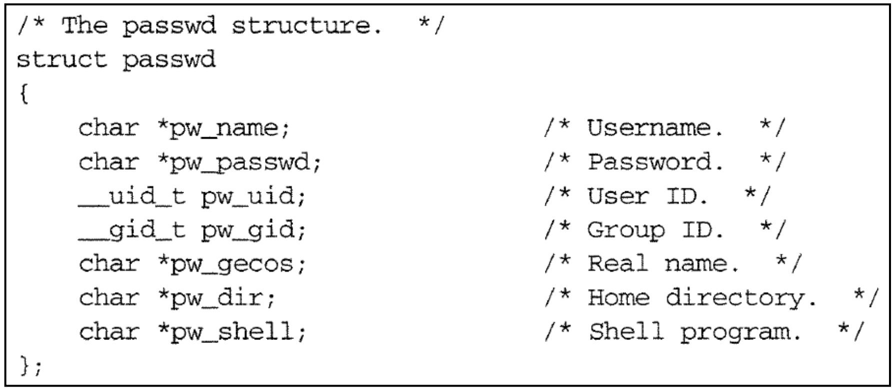
```
struct passwd *getpwuid(uid_t uid);
```

  - Example



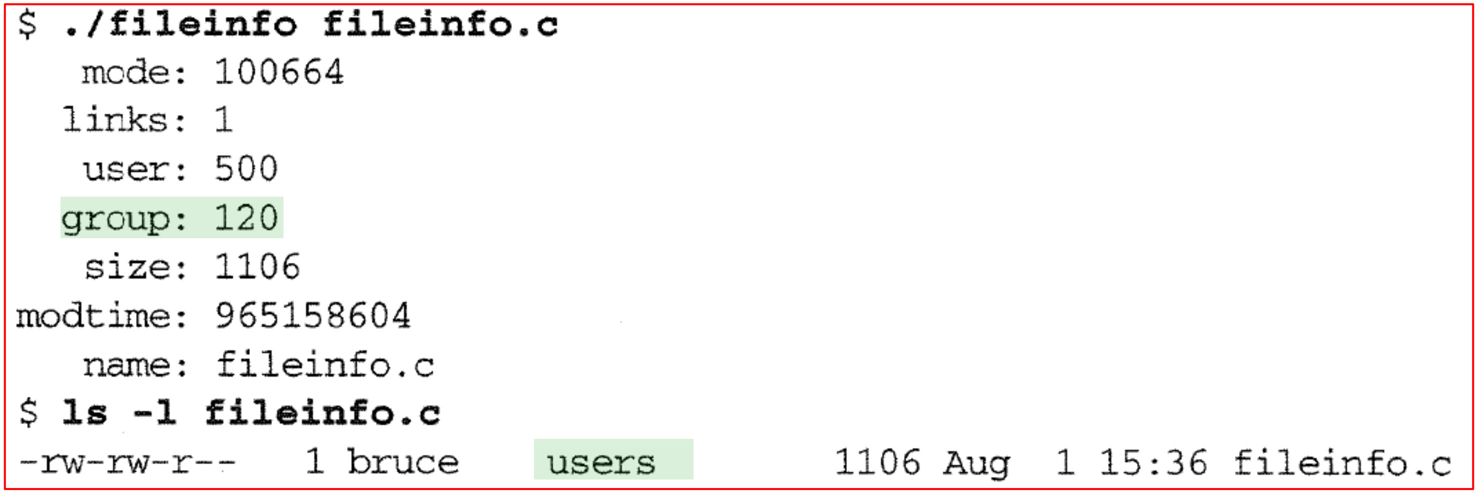
```
char *uid_to_name( uid_t uid )
{
     return getpwuid(uid)->pw_name ;
}
```

  - struct passwd



```
/* The passwd structure.  */
struct passwd
{
    char *pw_name;                  /* Username.  */
    char *pw_passwd;                /* Password.  */
    __uid_t pw_uid;                 /* User ID.  */
    __gid_t pw_gid;                 /* Group ID.  */
    char *pw_gecos;                 /* Real name.  */
    char *pw_dir;                   /* Home directory.  */
    char *pw_shell;                 /* Shell program.  */
};
```

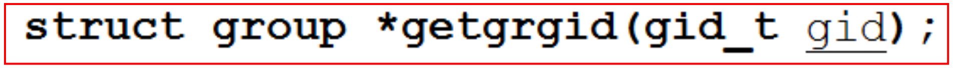
Converting Group ID to Strings



```
$ ./fileinfo fileinfo.c
   mode: 100664
  links: 1
   user: 500
  group: 120
   size: 1106
modtime: 965158604
   name: fileinfo.c
$ ls -l fileinfo.c
-rw-rw-r--   1 bruce    users        1106 Aug  1 15:36 fileinfo.c
```

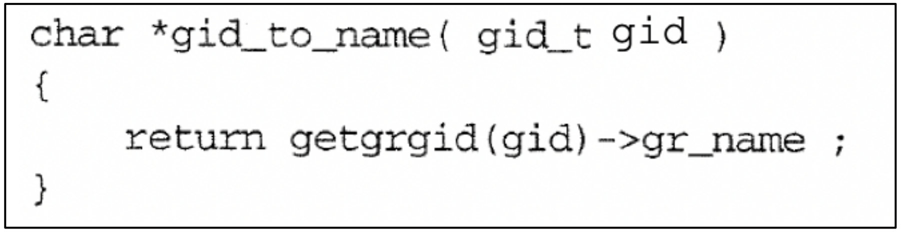
Converting Group ID to Strings

- getgrgid() provides access to the list of groups
  - Defined in /usr/include/grp.h



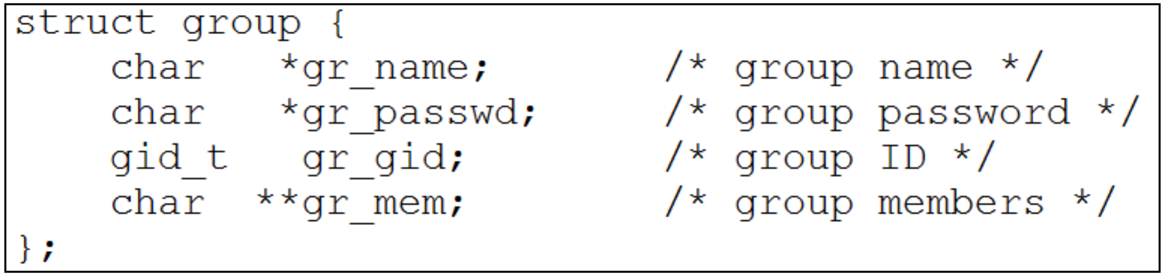
```
struct group *getgrgid(gid_t gid);
```

  - Example



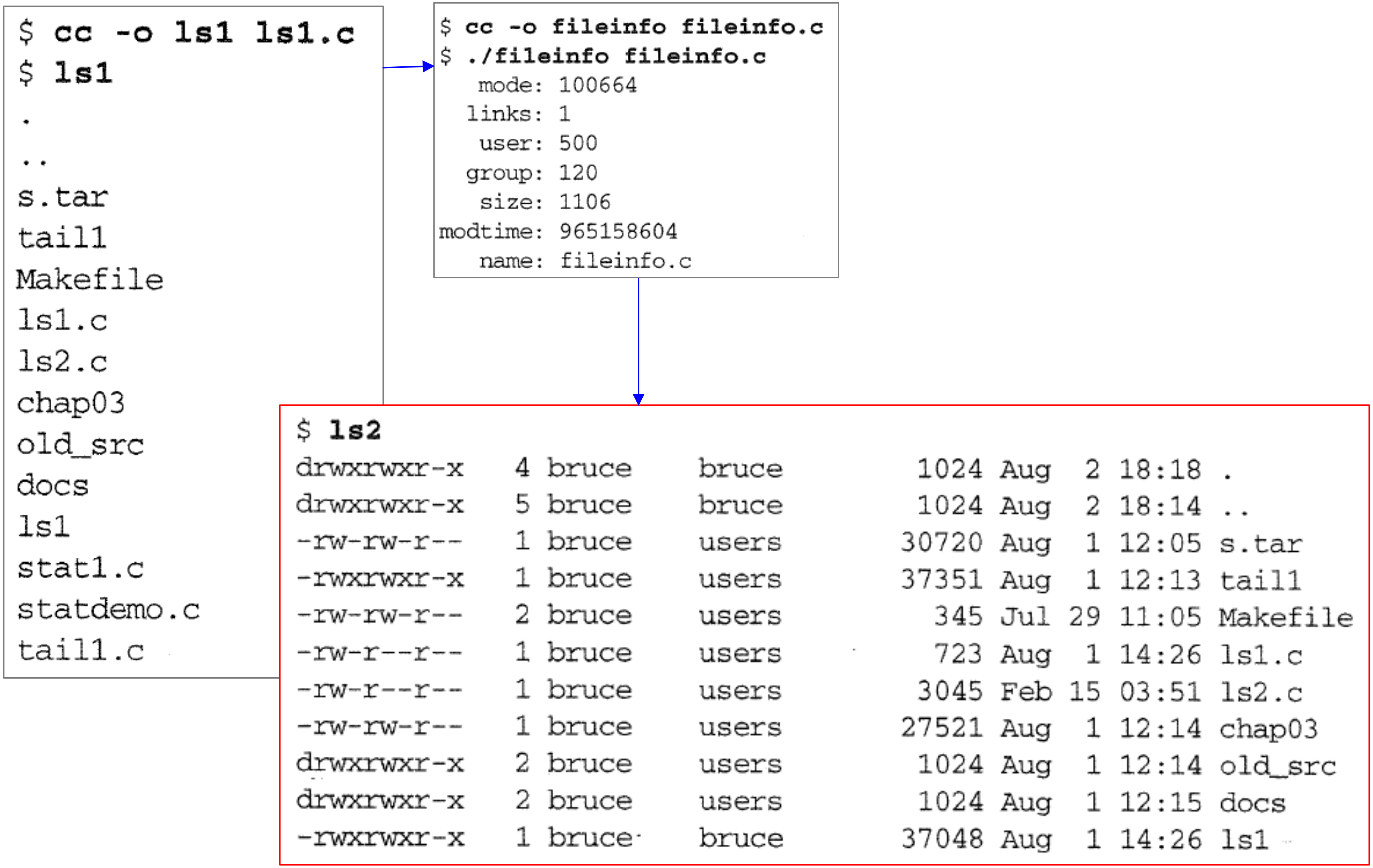
```
char *gid_to_name( gid_t gid )
{
    return getgrgid(gid)->gr_name ;
}
```

  - struct group



```
struct group {
    char    *gr_name;        /* group name */
    char    *gr_passwd;      /* group password */
    gid_t    gr_gid;         /* group ID */
    char   **gr_mem;         /* group members */
};
```

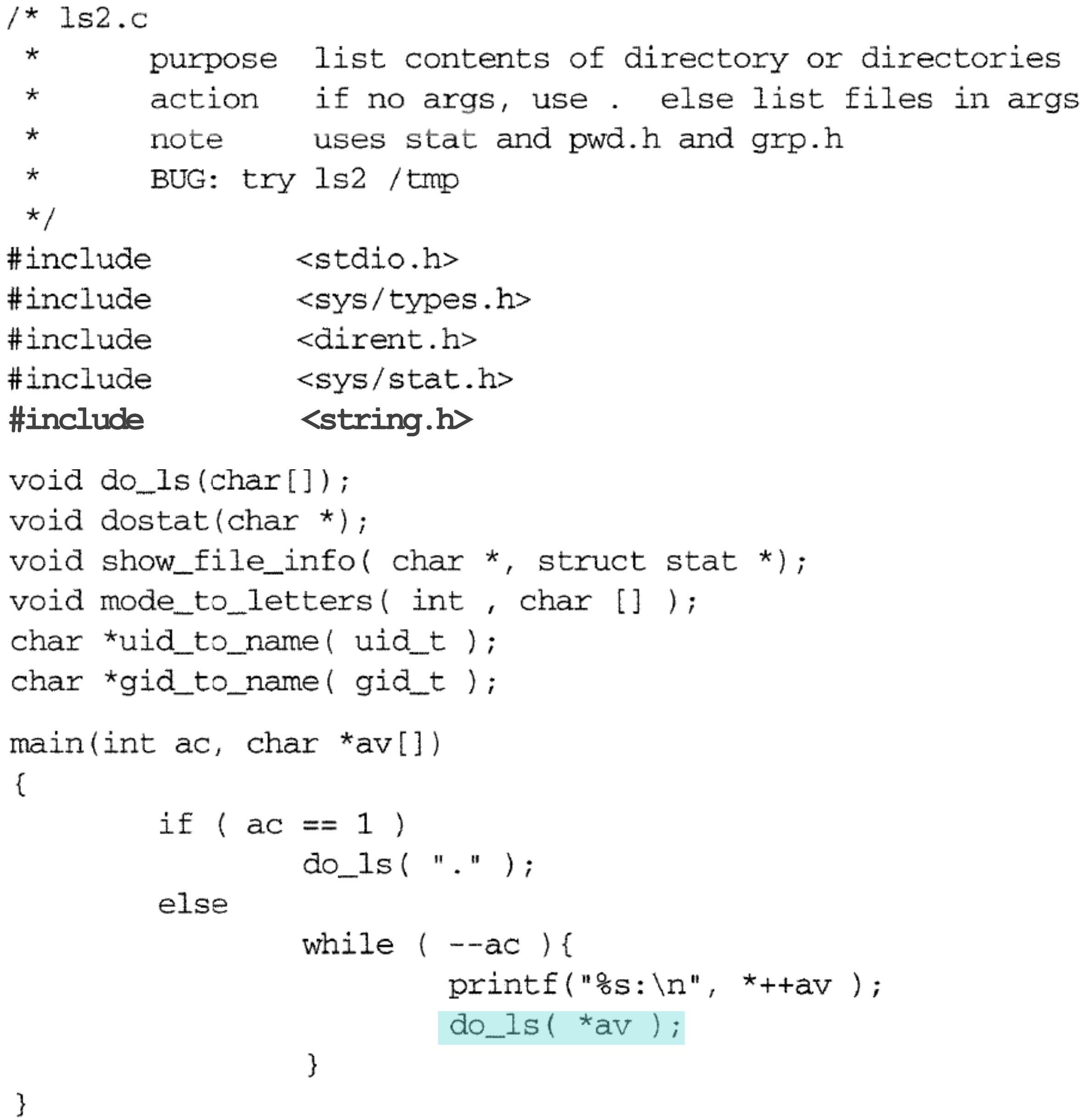
Putting It All Together: `ls2.c`



```
$ cc -o ls1 ls1.c
$ ls1
.
..
s.tar
tail1
Makefile
ls1.c
ls2.c
chap03
old_src
docs
ls1
stat1.c
statdemo.c
tail1.c
```

```
$ cc -o fileinfo fileinfo.c
$ ./fileinfo fileinfo.c
   mode: 100664
  links: 1
   user: 500
  group: 120
   size: 1106
modtime: 965158604
   name: fileinfo.c
```

```
$ ls2
drwxrwxr-x    4 bruce    bruce         1024 Aug  2 18:18 .
drwxrwxr-x    5 bruce    bruce         1024 Aug  2 18:14 ..
-rw-rw-r--    1 bruce    users        30720 Aug  1 12:05 s.tar
-rwxrwxr-x    1 bruce    users        37351 Aug  1 12:13 tail1
-rw-rw-r--    2 bruce    users          345 Jul 29 11:05 Makefile
-rw-r--r--    1 bruce    users          723 Aug  1 14:26 ls1.c
-rw-r--r--    1 bruce    users         3045 Feb 15 03:51 ls2.c
-rw-rw-r--    1 bruce    users        27521 Aug  1 12:14 chap03
drwxrwxr-x    2 bruce    users         1024 Aug  1 12:14 old_src
drwxrwxr-x    2 bruce    users         1024 Aug  1 12:15 docs
-rwxrwxr-x    1 bruce    bruce        37048 Aug  1 14:26 ls1
```

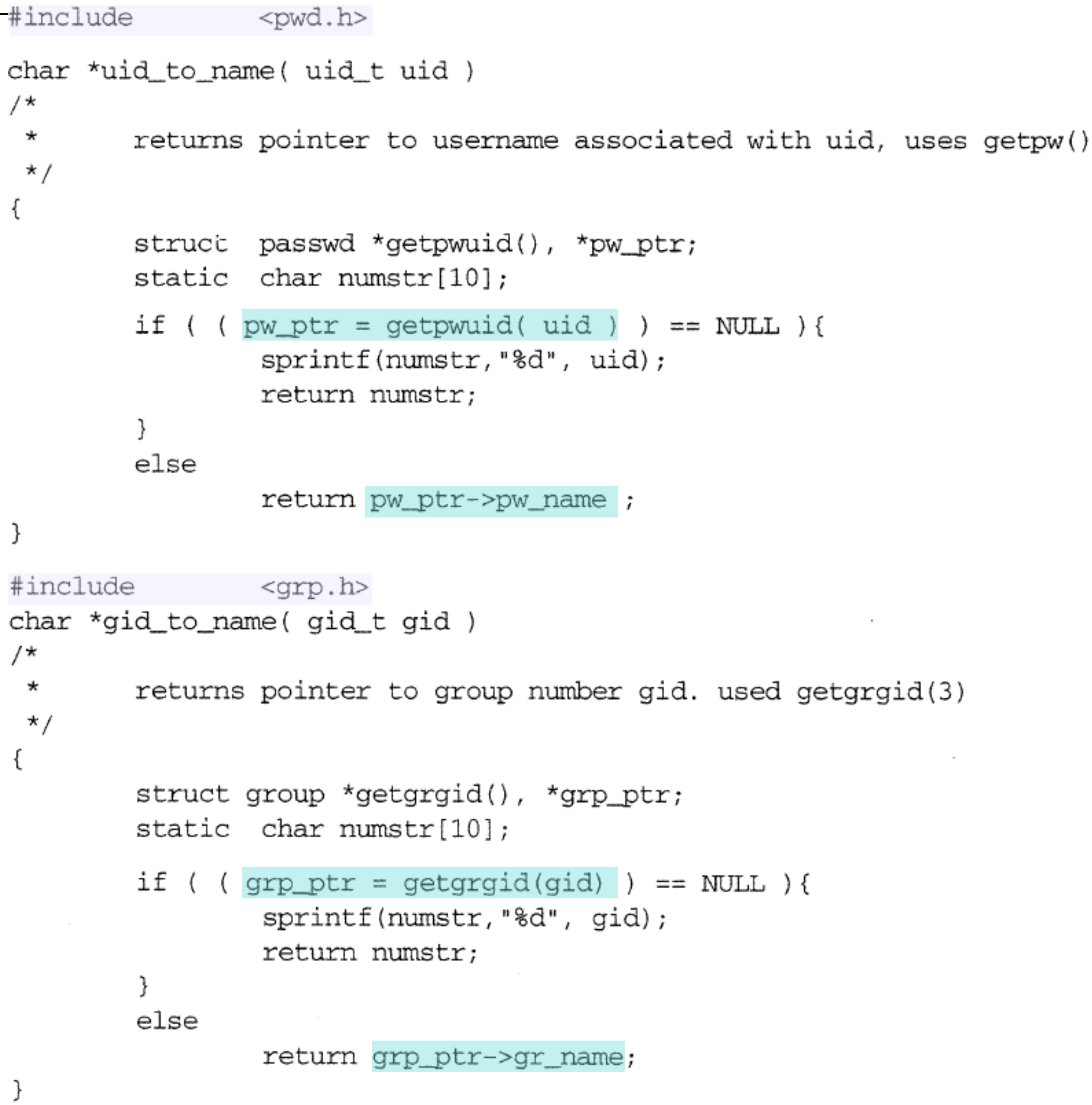


```
/* ls2.c
 *      purpose  list contents of directory or directories
 *      action   if no args, use .  else list files in args
 *      note     uses stat and pwd.h and grp.h
 *      BUG: try ls2 /tmp
 */
#include        <stdio.h>
#include        <sys/types.h>
#include        <dirent.h>
#include        <sys/stat.h>
#include        <string.h>

void do_ls(char[]);
void dostat(char *);
void show_file_info( char *, struct stat *);
void mode_to_letters( int , char [] );
char *uid_to_name( uid_t );
char *gid_to_name( gid_t );

main(int ac, char *av[])
{
        if ( ac == 1 )
                do_ls( "." );
        else
                while ( --ac ){
                        printf("%s:\n", *++av );
                        do_ls( *av );
                }
}
```

```
void do_ls( char dirname[] )
/*
 *      list files in directory called dirname
 */
{
        DIR             *dir_ptr;               /* the directory */
        struct dirent   *direntp;               /* each entry    */

        if ( ( dir_ptr = opendir( dirname ) ) == NULL )
                fprintf(stderr,"ls1: cannot open %s\n", dirname);
        else
        {
                while ( ( direntp = readdir( dir_ptr ) ) != NULL )
                        dostat( direntp->d_name );
                closedir(dir_ptr);
        }
}

void dostat( char *filename )
{
        struct stat info;

        if ( stat(filename, &info) == -1 )                      /* cannot stat */
                perror( filename );                           /* say why     */
        else                                          /* else show info      */
                show_file_info( filename, &info );
}
```

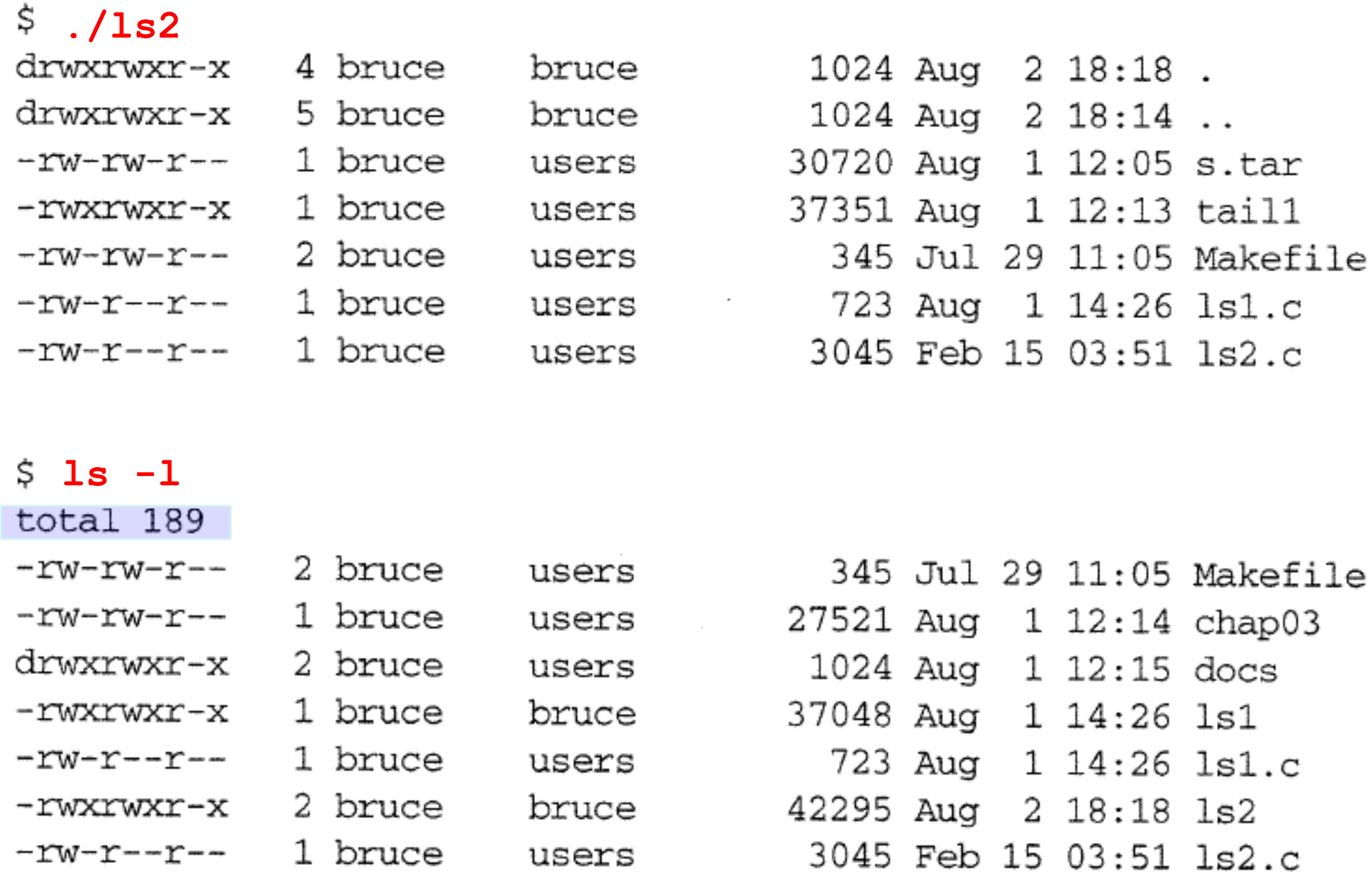
```
  void show_file_info( char *filename, struct stat *info_p )
  /*
   * display the info about 'filename'.  The info is stored in struct at
*info_p
   */
  {
          char    *uid_to_name(), *ctime(), *gid_to_name(), *filemode();
          void    mode_to_letters();
          char    modestr[11];

          mode_to_letters( info_p->st_mode, modestr );

          printf( "%s"    , modestr );
          printf( "%4d "  , (int) info_p->st_nlink);
          printf( "%-8s " , uid_to_name(info_p->st_uid) );
          printf( "%-8s " , gid_to_name(info_p->st_gid) );
          printf( "%8ld " , (long)info_p->st_size);
          printf( "%.12s ", 4+ctime(&info_p->st_mtime));
          printf( "%s\n"  , filename );

  }
```

```
/*
 * utility functions
 */

/*
 * This function takes a mode value and a char array
 * and puts into the char array the file type and the
 * nine letters that correspond to the bits in mode.
 * NOTE: It does not code setuid, setgid, and sticky
 * codes
 */
void mode_to_letters( int mode, char str[] )
{
    strcpy( str, "----------" );           /* default=no perms */

    if ( S_ISDIR(mode) )  str[0] = 'd';    /* directory?       */
    if ( S_ISCHR(mode) )  str[0] = 'c';    /* char devices     */
    if ( S_ISBLK(mode) )  str[0] = 'b';    /* block device     */

    if ( mode & S_IRUSR ) str[1] = 'r';    /* 3 bits for user  */
    if ( mode & S_IWUSR ) str[2] = 'w';
    if ( mode & S_IXUSR ) str[3] = 'x';

    if ( mode & S_IRGRP ) str[4] = 'r';    /* 3 bits for group */
    if ( mode & S_IWGRP ) str[5] = 'w';
    if ( mode & S_IXGRP ) str[6] = 'x';

    if ( mode & S_IROTH ) str[7] = 'r';    /* 3 bits for other */
    if ( mode & S_IWOTH ) str[8] = 'w';
    if ( mode & S_IXOTH ) str[9] = 'x';
}
```



```
#include        <pwd.h>

char *uid_to_name( uid_t uid )
/*
 *      returns pointer to username associated with uid, uses getpw()
 */
{
        struct  passwd *getpwuid(), *pw_ptr;
        static  char numstr[10];

        if ( ( pw_ptr = getpwuid( uid ) ) == NULL ){
                sprintf(numstr,"%d", uid);
                return numstr;
        }
        else
                return pw_ptr->pw_name ;
}

#include        <grp.h>
char *gid_to_name( gid_t gid )
/*
 *      returns pointer to group number gid. used getgrgid(3)
 */
{
        struct group *getgrgid(), *grp_ptr;
        static  char numstr[10];

        if ( ( grp_ptr = getgrgid(gid) ) == NULL ){
                sprintf(numstr,"%d", gid);
                return numstr;
        }
        else
                return grp_ptr->gr_name;
}
```

Result



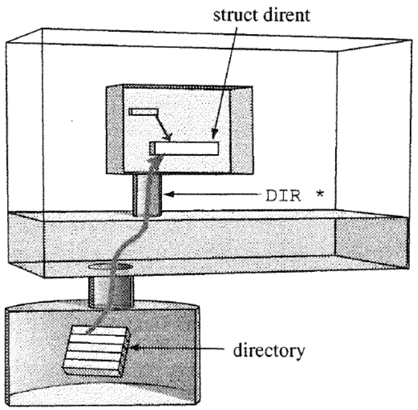
```
$ ./ls2
drwxrwxr-x   4 bruce    bruce        1024 Aug  2 18:18 .
drwxrwxr-x   5 bruce    bruce        1024 Aug  2 18:14 ..
-rw-rw-r--   1 bruce    users       30720 Aug  1 12:05 s.tar
-rwxrwxr-x   1 bruce    users       37351 Aug  1 12:13 tail1
-rw-rw-r--   2 bruce    users         345 Jul 29 11:05 Makefile
-rw-r--r--   1 bruce    users         723 Aug  1 14:26 ls1.c
-rw-r--r--   1 bruce    users        3045 Feb 15 03:51 ls2.c

$ ls -l
total 189
-rw-rw-r--   2 bruce    users         345 Jul 29 11:05 Makefile
-rw-rw-r--   1 bruce    users       27521 Aug  1 12:14 chap03
drwxrwxr-x   2 bruce    users        1024 Aug  1 12:15 docs
-rwxrwxr-x   1 bruce    bruce       37048 Aug  1 14:26 ls1
-rw-r--r--   1 bruce    users         723 Aug  1 14:26 ls1.c
-rwxrwxr-x   2 bruce    bruce       42295 Aug  2 18:18 ls2
-rw-r--r--   1 bruce    users        3045 Feb 15 03:51 ls2.c
```

sorting

Summary

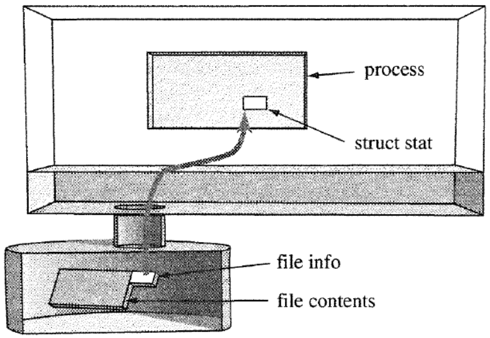
readdir()

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



stat()

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

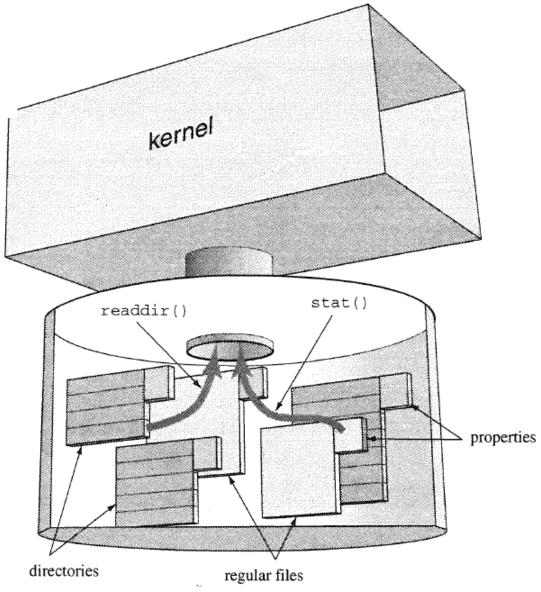


getpwuid()

```
struct passwd {
    char    *pw_name;       /* username */
    char    *pw_passwd;     /* user password */
    uid_t    pw_uid;        /* user ID */
    gid_t    pw_gid;        /* group ID */
    char    *pw_gecos;      /* user information */
    char    *pw_dir;        /* home directory */
    char    *pw_shell;      /* shell program */
};
```

getgrgid()

```
struct group {
    char    *gr_name;       /* group name */
    char    *gr_passwd;     /* group password */
    gid_t    gr_gid;        /* group ID */
    char   **gr_mem;        /* group members */
};
```

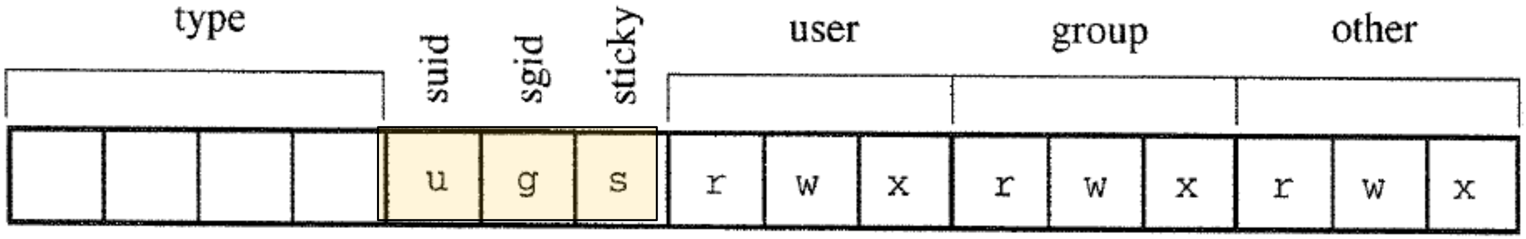


Agenda

- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

The Three Special Bits

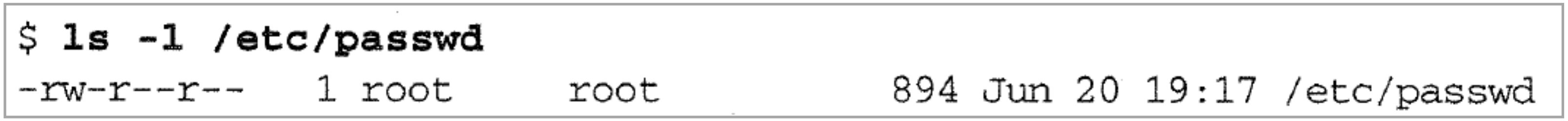
- The st_mode member of the stat structure:



- Three special bits are used to activate special properties of a file
  - suid(set-user-ID) bit
  - sgid(set-group-ID) bit
  - sticky bit

1. The Set-User-ID Bit

- How can a regular user change his or her password?
  - Use the passwd command!
  - But, how does the passwd command work?

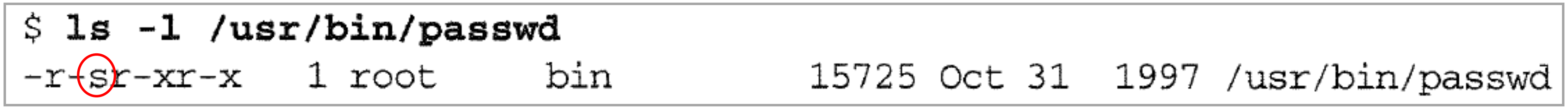


```
$ ls -l /etc/passwd
-rw-r--r--    1 root       root                894 Jun 20 19:17 /etc/passwd
```

Problem:
Changing your password means changing your record in the file `/etc/passwd`, but you do **NOT** have **permission** to write to that file.
Only the user named **`root`** has write permission.

1. The Set-User-ID Bit

- Solution: Give permission to the program, not to you.



```
$ ls -l /usr/bin/passwd
-r-sr-xr-x   1 root      bin            15725 Oct 31  1997 /usr/bin/passwd
```

  - The program you use to change your password, /usr/bin/passwd or /bin/passwd, is owned by root and has the set-user-ID (SUID) bit set.

  - That **SUID** bit tells the kernel to run the program as though it were being run by the owner of the program.

1. The Set-User-ID Bit

- Doesn't that mean I can change passwords of other users?
  - NO;
    - The **`passwd`** program knows who you are.
    - It uses the **`getuid`** system call to ask the kernel for the user ID you used when you logged in.
  - **`passwd`** has permission to rewrite the entire password file, but will **ONLY** change the **record for the user** running the program.

- Program can test whether a file has SUID bit on by using the mask defined in <sys/stat.h>

```
#define S_ISUID          0004000          /* set user id on execution */
```

2. The Set-Group-ID Bit

- The SGID bit sets the effective group ID of a program
  - If a program belongs to group g and the set-group-ID bit is set, the program runs as though it were being run by a member of group g

- This bit grants the program the access rights of members of that group

- A mask to test for the SGID bit

```
#define S_ISGID          0002000          /* set group id on execution */
```

3. The Sticky Bit

- Use for files
  - In swapping, the sticky bit told the kernel to keep the program on the swap device so that kernel can load it faster.
    - Loading program from swap device is fast because program was never fragmented on the swap device.
    - Now, **no longer necessary** due to virtual memory and paging that allow the kernel to move programs in and out of memory **in small sections**.

- **Use for directories**
  - /tmp are publicly writable, allowing any user to create and delete any files there.
  - The sticky bit overrides the publicly writable attribute for a directory. **Files in the directory may ONLY be deleted by their owners if the sticky bit is set**

The Special Bits and ls -l

- Each file has a type and 12 attribute bits, but ls uses only 9 spots to display these 12 attributes.

```
-rwsr-sr-t   1 root     root          2345 Jun 12 14:02 sample
```

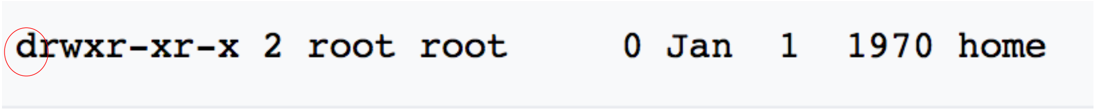
- Letter **s** indicates that the user and group-executable bits have been augmented by the set-user and set-group ID bits.

- Letter **t** at the end indicates that the sticky bits is on.

Agenda

- What Does **ls** Do?
- Brief Review of the File System Tree
- How Dos ls Work?
- Can I Write ls?
- Writing ls -l
- Three Special Bits
- Setting and Modifying the Properties of a File

Type of a File



```
drwxr-xr-x 2 root root      0 Jan  1  1970 home
```

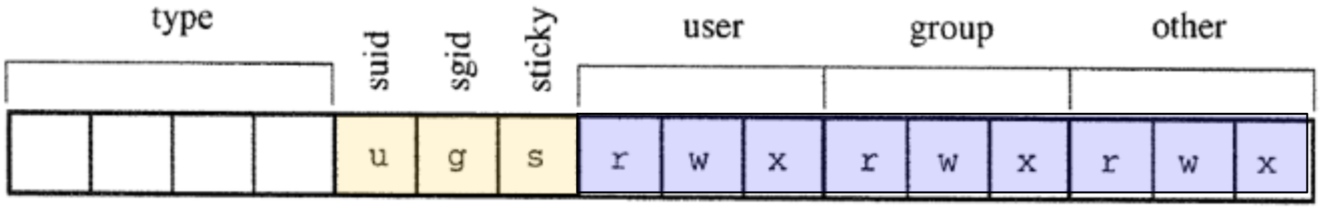
- A file has a type
  - It can be a regular file(-), a directory(d), a device file(b, c), a socket(s), a symbolic link(l), or a named pipe(p).

- The type of the file is established when the file is created.
  - The creat() system call creates a regular file.
  - Different system call are used to create directories and devices.

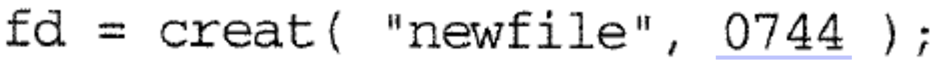
- It is not possible to change the type of a file.

Permission Bits and Special Bits

- Every file has 9 permission bits and 3 special bits.



- These bits are established when file is created and can be modified by making the chmod system call

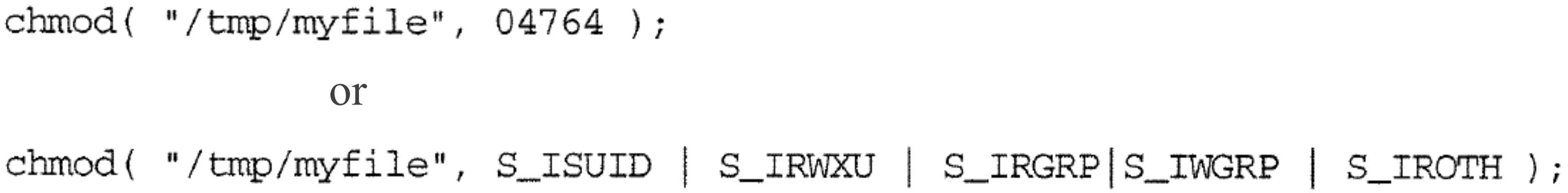


```
fd = creat( "newfile", 0744 );
```

- If you want to prevent programs from creating files that can be modified by group or others
  - umask(022);

Permission Bits and Special Bits

- Changing the mode of a file: **chmod()** system call



```
chmod( "/tmp/myfile", 04764 );
                or
chmod( "/tmp/myfile", S_ISUID | S_IRWXU | S_IRGRP|S_IWGRP | S_IROTH );
```

- A shell command to change permission and special bits

```
$ chmod 04764 test
          or
$ chmod u=rws test
$ chmod g=rw test
$ chmod o=r test
```

chmod

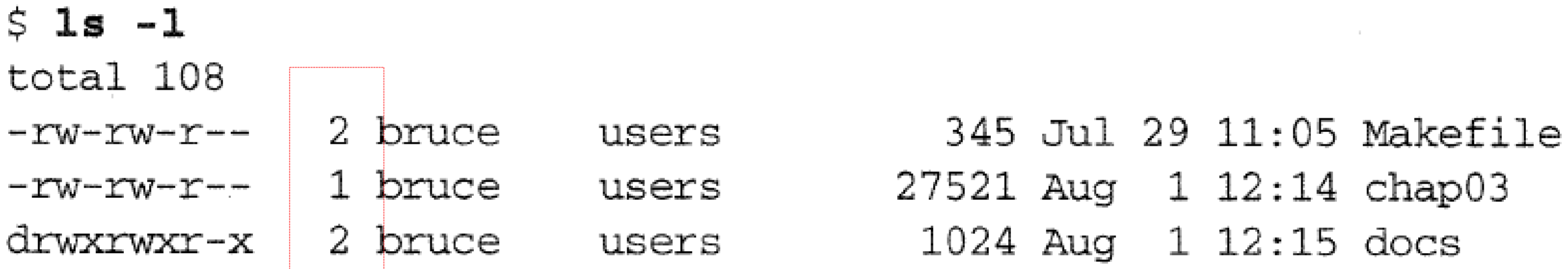
PURPOSE	Change permission and special bits for a file	

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

USAGE	int result = chmod(char *path, mode_t mode);	
ARGS	path	path to file
	mode	new value for mode
RETURNS	-1	if error
	0	if success

Number of Links to a File

- The number of links is simply the number of times the file is referenced in directories.
  - If a file appears in three places in various directories, the link count is 3. (in the next chapter)



```
$ ls -l
total 108
-rw-rw-r--   2 bruce    users            345 Jul 29 11:05 Makefile
-rw-rw-r--   1 bruce    users          27521 Aug  1 12:14 chap03
drwxrwxr-x   2 bruce    users           1024 Aug  1 12:15 docs
```

Type&permission links owner group size modified-date/time name

Owner and Group of a File

- Establishing the owner of a file:
  - The owner of file is the user who creates it
  - When kernel creates a file, it sets the owner of the file to be the effective user ID of the process that calls creat()
  - If the program has the set-user-ID bit set, though, the effective user ID is the user ID of the person who owns the program.

Owner and Group of a File

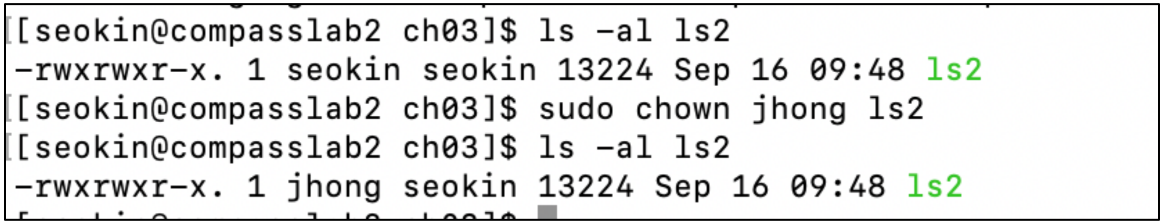
- Establishing the group of a file:
  - The group of a file is set to the effective group ID of the process that creates the file.
  - Under non-ordinary circumstances, the group ID of a file is set to the group ID of the parent directory.

Owner and Group of a File

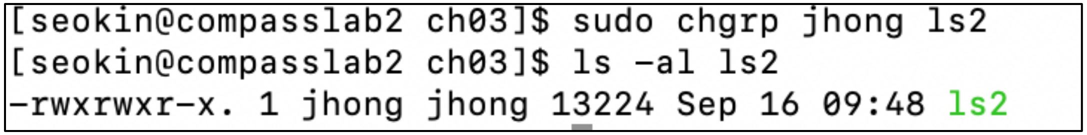
- Changing the owner and group of a File
  - **chown()** system call:
    - Normally, users do not change the owner of a file
    - Typically used to set up and manage user accounts

```
chown( "file1", 200, 40 );
```

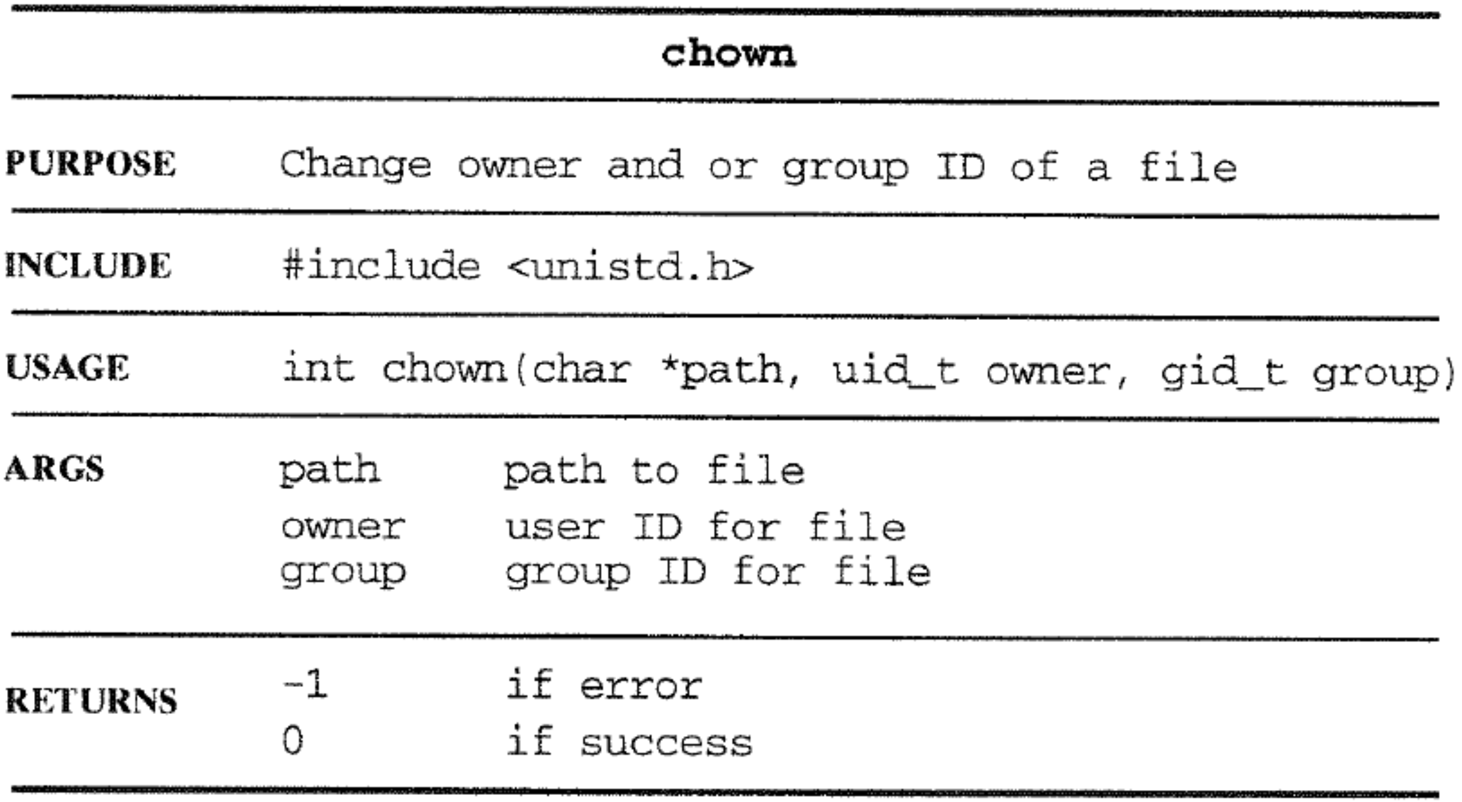
- Shell Commands to Change User and Group ID for Files: **chown, chgrp**



```
[seokin@compasslab2 ch03]$ ls -al ls2
-rwxrwxr-x. 1 seokin seokin 13224 Sep 16 09:48 ls2
[seokin@compasslab2 ch03]$ sudo chown jhong ls2
[seokin@compasslab2 ch03]$ ls -al ls2
-rwxrwxr-x. 1 jhong seokin 13224 Sep 16 09:48 ls2
```



```
[seokin@compasslab2 ch03]$ sudo chgrp jhong ls2
[seokin@compasslab2 ch03]$ ls -al ls2
-rwxrwxr-x. 1 jhong jhong 13224 Sep 16 09:48 ls2
```



	chown	
PURPOSE	Change owner and or group ID of a file	
INCLUDE	#include <unistd.h>	
USAGE	int chown(char *path, uid_t owner, gid_t group)	
ARGS	path	path to file
	owner	user ID for file
	group	group ID for file
RETURNS	-1	if error
	0	if success

Modification and Access Time

- Each file has three timestamps of
  - last modified
  - last read
  - file properties (such as owner ID or permission bits) were last changed
  - Kernel automatically updates these times as programs read and write the file

- Changing modification and access times of a file:
  - **utime()** system call

- Shell Commands: **touch**

utime

PURPOSE	Change access and modification time for files	

INCLUDE	#include <sys/time.h>
	#include <utime.h>	

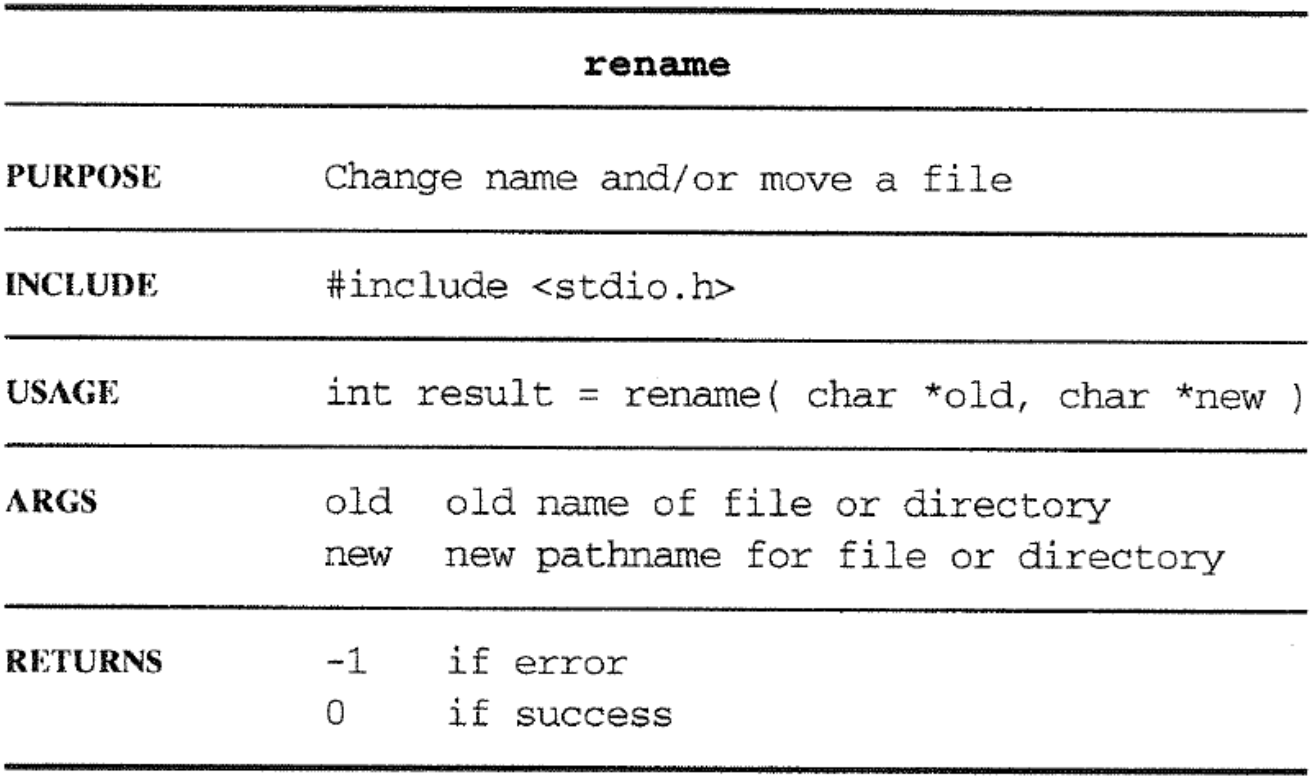
USAGE	#include <sys/types.h> int utime(char *path, struct utimbuf *newtimes)	
ARGS	path	path to file
	newtimes	pointer to a struct utimbuf see utime.h for details
RETURNS	-1	if error
	0	if success

Name of a File

- Establishing the Name of a File
  - creat() system call sets the name and the initial mode of a file.

- Changing the Name of a File:
  - **rename()** system call

- Shell Command : **mv**
  - Allows you to change the name of a file
  - Also allows you to move a file from one directory to another



	rename
PURPOSE	Change name and/or move a file
INCLUDE	#include <stdio.h>
USAGE	int result = rename(char *old, char *new)
ARGS	old old name of file or directory new new pathname for file or directory
RETURNS	-1 if error 0 if success

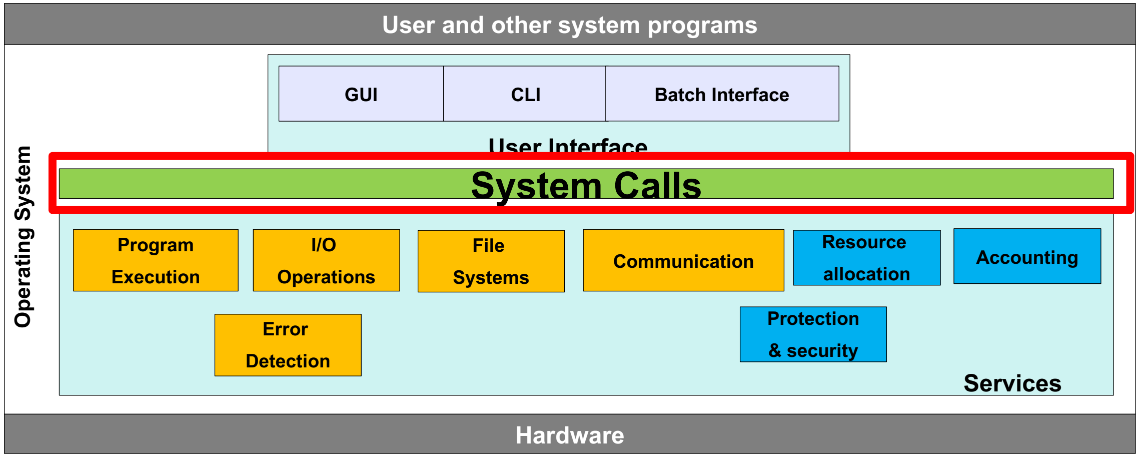
Objectives

- **Ideas and Skills**
  - A directory is a list of files
  - How to read a directory
  - Types of files and how to determine the type of a file
  - Properties of files and how to determine properties of a file
  - Bit sets and bit masks
  - User and group ID numbers
- **System Calls and Functions**
  - opendir, readdir, closedir, seekdir
  - stat
  - chmod, chown, utime
  - rename
- **Commands**
  - ls

Next?

- Ch04. File System

System Calls



- **System call** : Programmatic way in which a computer program requests a service of the operating system.
- **A function** provided to applications by the OS kernel
  - Generally to use a hardware abstraction (file, socket)
  - Or to use OS-provided software abstraction (IPC, scheduling)
- System calls are the **only entry points** into the kernel system
  - All programs needing resources must use system calls

System Calls

- Why not put these directly in the application?
  - **Protection of OS/hardware from buggy/malicious programs**
  - Applications are not allowed to directly interact with hardware, or access kernel data structures
  - OS must validate system call parameters

Interrupt

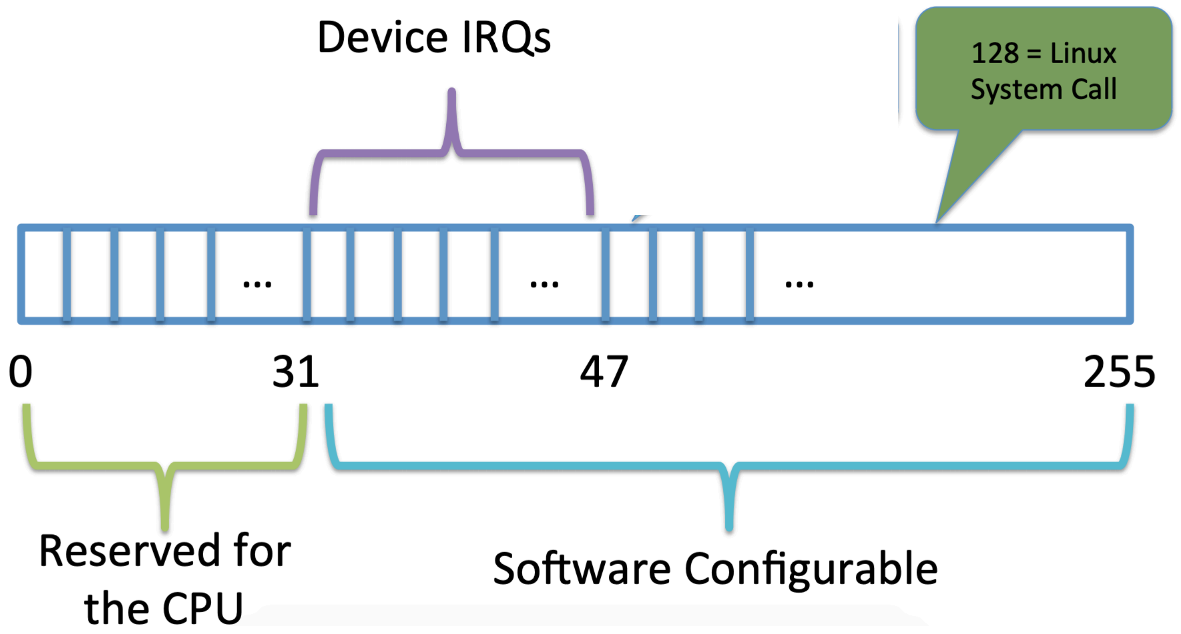
- **Interrupt view of CPU**

```
while (fetch next instruction){
    run instruction;
    if(there is an interrupt){
        save CPU context and error code if any
        find OS-provided interrupt handler
        jump to handler
        restore CPU context when handler returns
       }
}
```

Software Interrupt

- “**int** <num>” instruction
  - allows software to raise an interrupt **(software Interrupt)**
  - <num> is 0x80 in Linux

x86 interrupt table

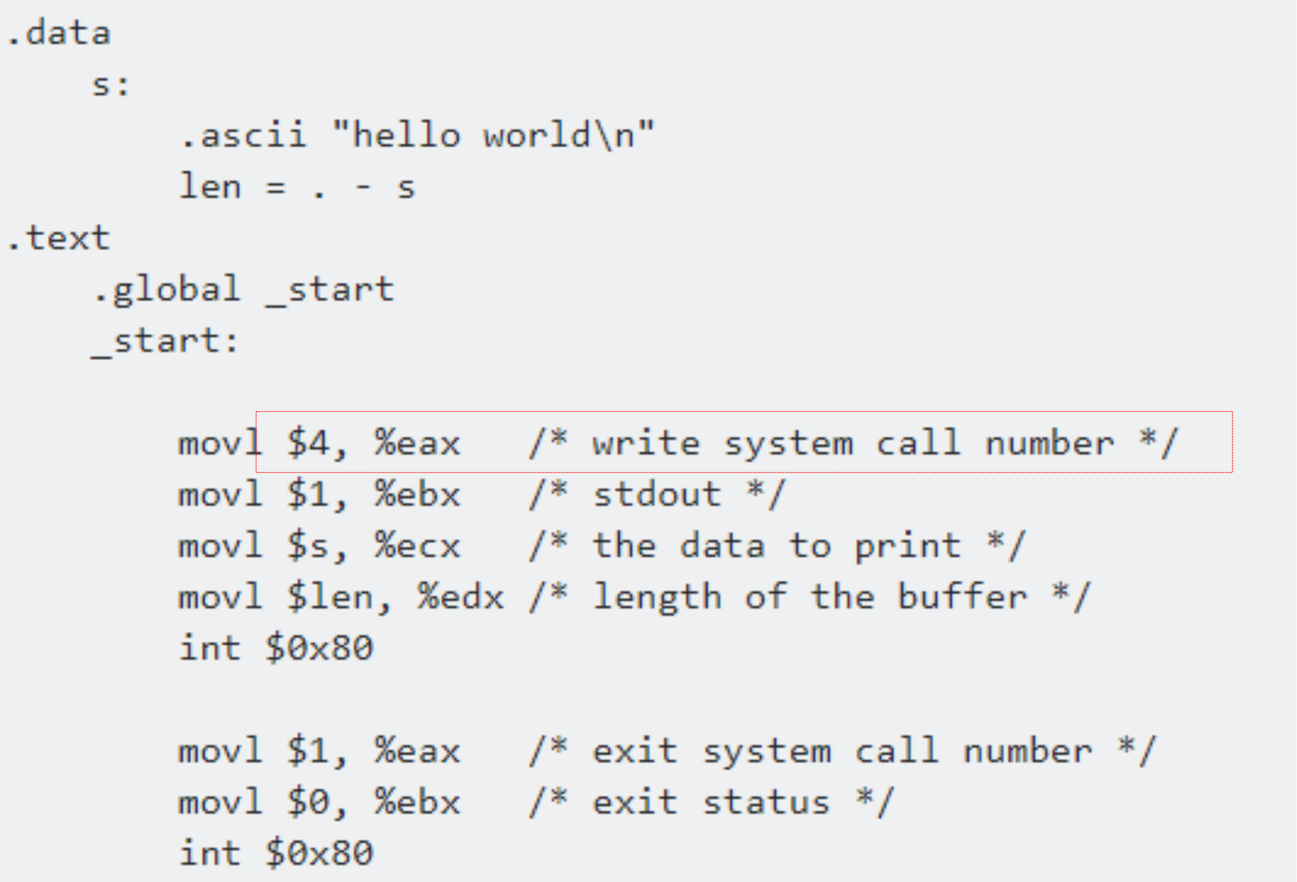


How to invoke System Calls?

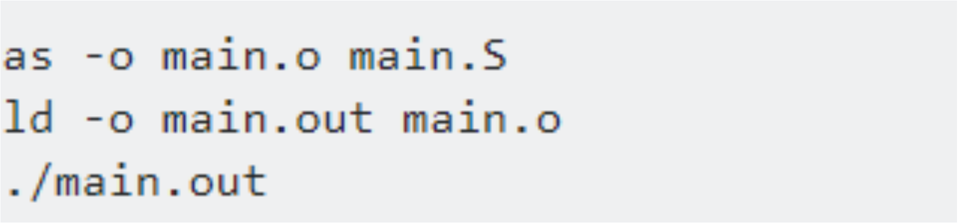
1. Kernel assigns a system call number to each system call type and initialize the system call table
2. **User process sets up system call number and arguments**
3. **User process runs int X (X is 0x80 in linux0)**
4. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)
5. Kernel looks up system call table using system call number
6. Kernel invokes the corresponding function
7. Kernel returns by running iret (interrupt return)

How to invoke System Calls?

Example



```
.data
    s:
        .ascii "hello world\n"
        len = . - s
.text
    .global _start
    _start:

        movl $4, %eax    /* write system call number */
        movl $1, %ebx    /* stdout */
        movl $s, %ecx    /* the data to print */
        movl $len, %edx /* length of the buffer */
        int $0x80

        movl $1, %eax    /* exit system call number */
        movl $0, %ebx    /* exit status */
        int $0x80
```



```
as -o main.o main.S
ld -o main.out main.o
./main.out
```

Why use interrupts to invoke system call?

- Also protection
- Forces applications to call well-defined “public functions”
- **But, using interrupt is complex and not portable!**

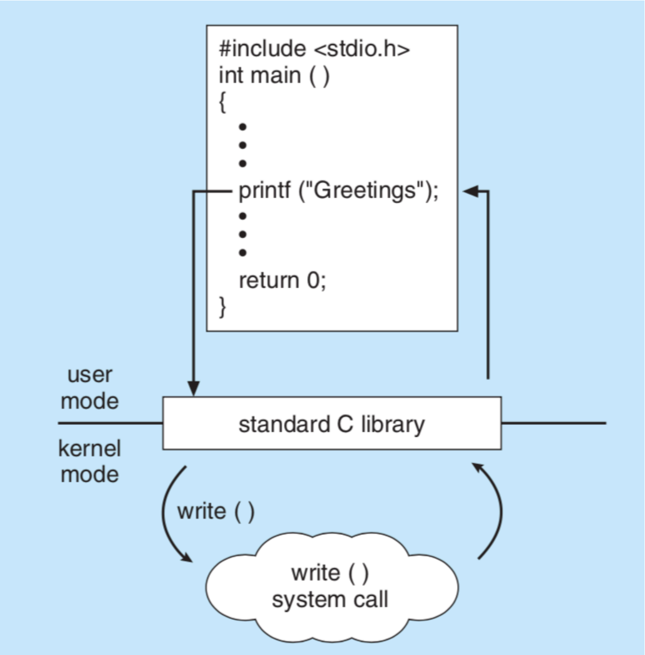
API for System Calls

- Programmers use system calls indirectly through **APIs** (Application Programming Interface: set of functions)
  - **Windows API** for Windows systems
  - **POSIX API** for POSIX-based systems (UNIX, Linux, and Mac OS X)
  - **Java API** for programs that run on the JVM
  - **A programmer accesses an API via a library**
    - Ex) libc

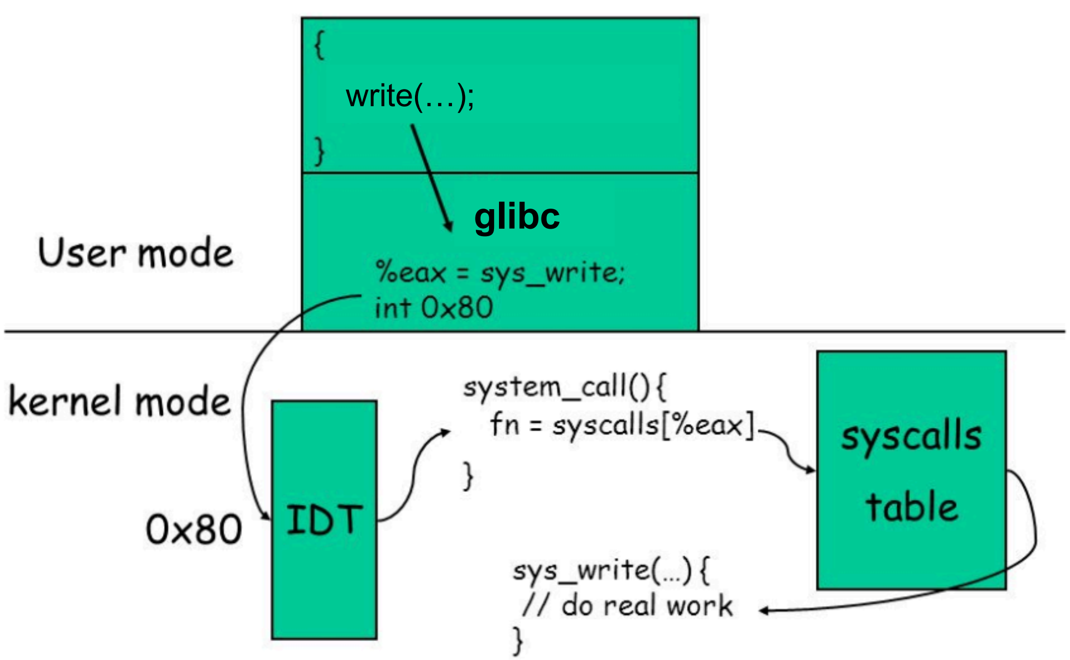
glibc	system call
write	write
read	read
printf	write
fread	read
malloc	brk
pthread_lock	futex

System Call Interface

- Set of library functions that links to the system calls



```
#include <stdio.h>
int main ( )
{
  •
  •
  •
  printf ("Greetings");
  •
  •
  •
  return 0;
}
```



```
{
  write(…);
}
```

```
%eax = sys_write;
int 0x80
```

```
system_call() {
  fn = syscalls[%eax]
}
```

```
sys_write(…) {
 // do real work
}
```

- Caller does not need to know how system call is implemented
- Caller needs to know only the interface and what it returns

```
ssize_t write(int fd, void *buf, size_t count);
```

System Call Types

- **Six categories**
  - **Process control**
    - fork, exec, exit …
  - **File manipulation**
    - create, open, close, read, write, lseek
  - **Device manipulation**
    - open, close, read, write, ioctl
  - **Information maintenance**
    - time, date, dump, pid
  - **Communications**
    - open, close, connect, accept, read, write, send, recv, pipe, mmap, sendfile …
  - **Protection**
    - chmod, umask, chown …