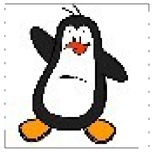


Video Lecture # 14
Design and Code of
UNIX `ls` utility

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Agenda

- Directory management
- What does **`ls`** do?
- How does **`ls`** do it?
- Coding a basic version of **`ls`** command
- Retrieving file attributes from its inode block
- Retrieving owner and group name of file
- Retrieving timestamps of file
- Determining file type and access permissions on file
- Writing version of ls that displays a long listing of directory contents
- Assignment versions
- Assignment: Design and code of **`find`** and **`grep`** utilities

Directory Management Calls

mkdir and rmdir Function

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

- **`mkdir`** function creates a new empty directory with two entries . & ..
- Permissions on the created directory are:
  `mode & ~umask & 0777`
- The new directory will be owned by the effective **`UserID`** of the process
- The **`rmdir`** function is used to delete an empty directory. If the link count of the directory becomes 0 with this call, and if no other process has the directory open, then the space occupied by the directory is freed
- If one or more processes have the directory open when the link count reaches 0 then:
  - The last link is removed
  - The . and .. entries are removed before the function return
  - No new files can be created in this directory
  - The directory is freed when the last process closes it

opendir() Function

```
DIR *opendir(const char* dirpath);
int closedir(DIR *dirp);
```

- The **`opendir()`** function opens the directory specified by `dirpath` and returns a pointer to a structure of type `DIR`, which is used to refer that directory in later calls. Upon return from `opendir()`, the directory stream is positioned at the first entry in the directory list
- The **`closedir()`** function closes the directory stream associated with `dirp`
- Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, so the write permission bits and execute permission bits for a directory determine if we can create new files in the directory and remove files from the directory

readdir() Function

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

- The **readdir()** function is passed the `DIR*` which is returned by `opendir()`. Every time it is called it returns an entry from the directory stream referred to by `dirp`. The return value is a pointer to a structure of type `dirent`, containing the following information about the entry (it may vary from OS to OS):

```
struct dirent {
   ino_t d_ino;   /* File i-node number */
   char d_name[]; /* Null-terminated name of file */
 };
```

- This structure is overwritten on each call to `readdir()`
- The filenames returned by `readdir()` are not in sorted order, but rather in the order in which they happen to occur in the directory (this depends on the order in which the file system adds files to the directory and how it fills gaps in the directory list after files are removed). (The command `ls -f` lists files in the same unsorted order that they would be retrieved by `readdir()`

readdir() Function (cont...)

- On end-of-directory or error, `readdir()` returns NULL, in the latter case setting `errno` to indicate the error. To distinguish these two cases, we can write the following:

```
 errno = 0;
  struct dirent * entry = readdir(dp);
  If ((entry == NULL) && (errno != 0))
                /* Handle error */
     else
               /* We reached end-of-directory */
  }
```

- If the contents of a directory change while a program is scanning it with `readdir()`, the program might not see the changes. SUSv3 explicitly notes that it is unspecified whether `readdir()` will return a filename that has been added to or removed from the directory since the last call to `opendir()`. All filenames that have been neither added nor removed since the last such call are guaranteed to be returned

Misc Functions

```
int chdir(const char *pathname);
void rewinddir(DIR *dirp);
off_t telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
```

- The **`chdir()`** function is used to change the current working directory of the calling process, and `pathname` is the directory where the search for all relative pathname starts
- The **`rewinddir()`** function moves the directory stream `dirp` back to the beginning, so that the next call to `readdir()` will begin again with the first file in the directory
- The **`telldir()`** function returns the current location associated with the directory stream `dirp`
- The **`seekdir()`** function sets the location in the directory stream from which the next `readdir()` call will start. The `loc` argument should be a value returned by a previous call to `telldir()`

Example: `myreaddir.c`

UNIX `ls` utility

What does `ls` do?

The default behavior of ls (w/o any arguments/options) is to list contents of present working directory after sorting them in alphabetic order and displaying them on stdout in columns (list down and then across). How much columns and rows depends on the maximum number of file names that can be managed in a single row. Important options:

- **`-t`:** Sort by modification time (newer to old)
- **`-S`:** Sort by size (bigger to smaller)
- **`-1`:** Display single entry per line
- **`-a`:** Show hidden files as well
- **`-A`:** Show hidden files less `.` and `..`
- **`-F`:** Show visual classification of files with names
- **`-i`:** Show inode numbers with names
- **`-R`:** Recursively display contents of sub directories also
- **`-l`:** Long listing, sorted in alphabetic order by file name, display `mtime`
- **`-n`:** Long listing, but show UID/GID instead of owner/group names
- **`-ld`:** Long listing of directory and not its contents
- **`-lu`:** Long listing, sorted in alphabetic order by file name, display `atime`
- **`-lc`:** Long listing, sorted in alphabetic order by file name, display `ctime`

How does `ls` do it?

1. Open the directory
2. Read entry till end of directory
3. Display entry contents
4. Go to step 2
5. Close directory

lsv0.c

Receives exactly one directory name via command line argument and display names of files and subdirectories in the order as they appear in the directory

`lsv1.c`

This version adds a feature and our `ls` command works on multiple directory names passed via command line arguments. If no argument is passed display the contents of `pwd`

lsv2.c

This version adds a feature that does not display the hidden files in the directory

`lsv3.c`

This version adds a feature that will display every file in the directory in long listing

What does `ls -l` do?

The long listing of `ls` command displays seven columns for each file:

1. File Type (`-,d,l,p,c,b,s`) and Permissions (`rwxrwxrwx`)
2. Link Count
3. User
4. Group
5. Size
6. Time
7. Name of file

These attributes of a file (less its name) are not stored in `dirent` structure

`stat()` System call

```
int stat(const *char pathname, struct stat *buff);
int lstat(const *char linkname, struct stat *buff);
```

- These functions can be used to access the file attributes stored in its inode. To stat a file no permissions are required on the file itself, however, execute (search) permission is required on all of the directories in pathname that leads to the file
- **`stat()`** stats the file pointed by `path` and fills in `buff`
- **`lstat()`** is similar to `stat`, except if path is a symbolic link, then the link itself is stated , not the file it refers to
- On success returns 0 and on error returns -1 and set `errno`
- On success, populate the `stat` structure as mentioned on next slide

stat() system call (cont ...)

Some important members of the `stat` structure that of our interest right now are shown below:

```
struct stat{
   ino_t    st_ino;   //inode number
   mode_t   st_mode;  //file type & 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
   off_t    st_size;  //total size in bytes
   time_t   st_atime; //time of last access
   time_t   st_mtime; //time of last data modification
   time_t   st_ctime; //time of last status change
   };
```

Example: `fileinfo.c`

This program receives a file name via command line argument and display its attributes on screen

getpwuid() System call

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

- The **getpwuid()** function returns a pointer to a structure containing the broken-out fields of the record in the password database (/etc/passwd) that matches the uid
- The passwd structure is defined in <pwd.h> as follows:

```
struct passwd{
   char*     pw_name;
   char*     pw_passwd;
   uid_t     pw_uid;
   gid_t     pw_gid;
   char*     pw_gecos;
   char*     pw_dir;
   char*     pw_shell;
};
```

stat() System call

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

- The **getgrgid()** function returns a pointer to a structure containing the broken-out fields of the record in the group database (`/etc/group`) that matches the `gid`
- The `group` structure is defined in `<grp.h>` as follows:

```
struct group{
   char*    gr_name;
   char*    gr_passwd;
   gid_t    gr_gid;
   char*    gr_gmem;
};
```

Examples: `uidtouname.c, gidtogname.c`

These programs receives a UID/GID via command line argument and display the corresponding user/group name

Real / Calender Time

Review OS with LinuxVideo Lec 27 on Time Management

Real Time

- Obtaining calender time is useful to programs, e.g, to update timestamps of files
- Regardless of geographic location, UNIX system represent time internally as measure of seconds since the **epoch**; i.e since midnight 00:00, Jan 1 , 1970, Universal Time (UTC, previously known as Greenwich Mean Time GMT)
- On 32-bit Linux system, **`time_t`** is a signed integer that is used to store the number of seconds passed since the UNIX epoch. It can represent dates in the range of

  `13 Dec 1901 20:45:52     to     19 Jan 2038 03:14:07`

- **2038 Problem:** A 31 bits variable can contain a maximum number of `2,147,483,646` seconds. Any number beyond this limit i.e `2,147,483,647` will wrap around & will be stored as -ve number i.e `-2,147,483,648.` Linux will interpret this as 13 Dec 1901 20:45:52
- To display number of seconds passed since UNIX epoch

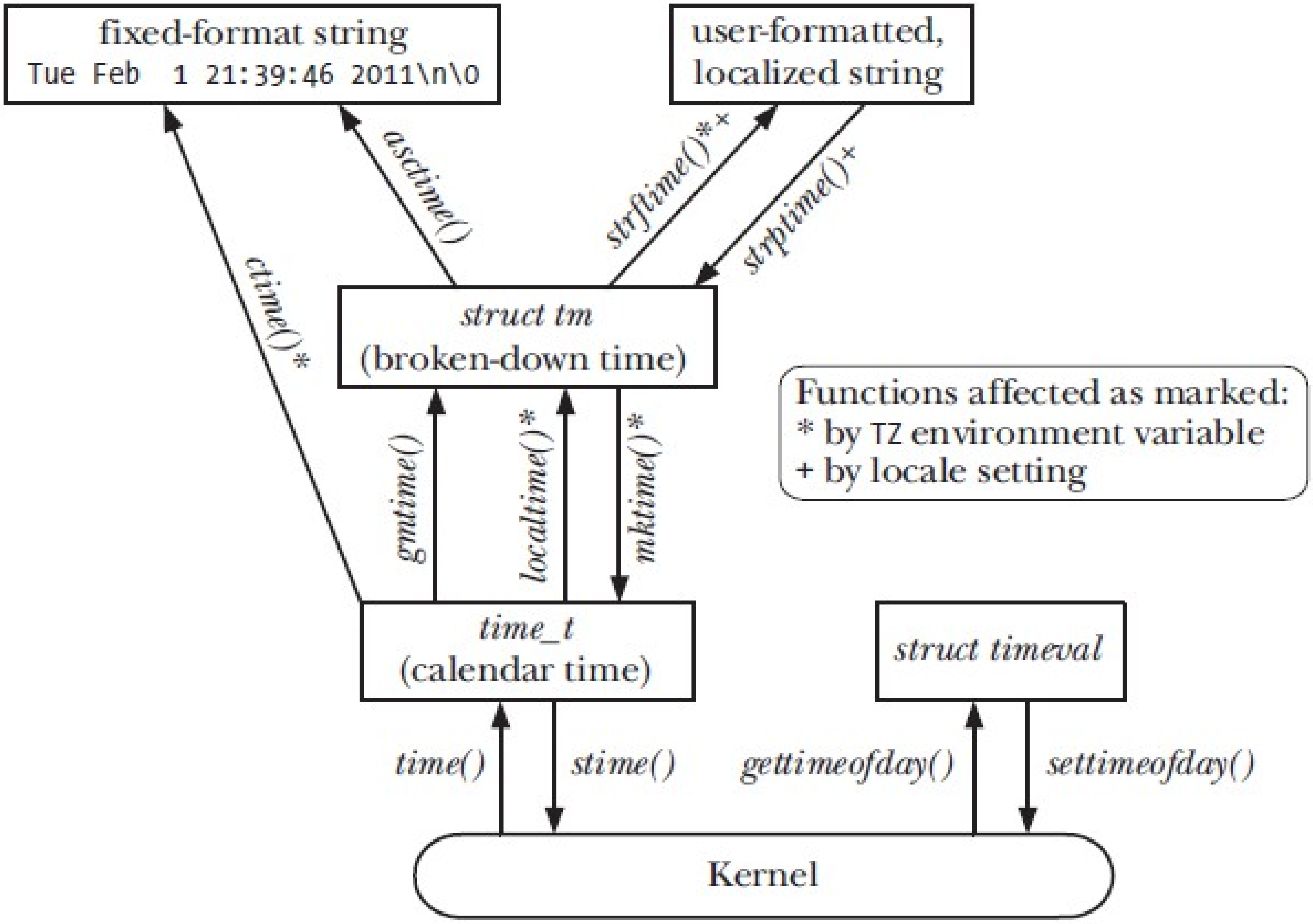
```
$date +%S
1,365,147,372
```

time() and ctime() Functions

```
time_t time(time_t *t);
char * ctime(const time_t* timep)
```

- The **`time()`** function is used to get the time since UNIX Epoch in seconds. If the argument is non-null, the return value is also stored in the memory pointed to by t
- The **`ctime()`** function takes the number of seconds and return a null terminated string of the form `Mon Mar 12 10:10:10 2018`

Relationship of Time Functions



Example:
`transformtime.c`

Determining File Type and Permissions

Review OS with LinuxVideo Lec 22 and 23

Understanding `st_mode` member of `stat()`

File Type (4)	Special Permissions (3)	User (3)	Group (3)	Others (3)
1000	000	110	100	000

- The 16 bit `st_mode` member of `struct stat` that we achieved using `stat(2)` system call contains information about file type and permissions as shown above
- The codes for the seven file types are mentioned in the table below:

Decimal	Binary	Octal	File Type
1	0001	01	p
2	0010	02	c
4	0100	04	d
6	0110	06	b
8	1000	10	-
10	1010	12	l
12	1100	14	s

Determining the File Type

File Type (4)	Special Permissions (3)	User (3)	Group (3)	Others (3)
1000	000	110	100	000

- You can determine the file type by creating a mask by making all the bits zero other than the one of your interest. So to determine the file type you set the four bits of file type and zero out the rest of the bits
- Then you perform a bitwise **&** of the `st_mode` value with the mask, and compare the output with the file codes to determine file type. A sample code snippet is shown:

```
if ((buf.st_mode &  0170000) == 0010000)
     printf("Named pipe\n");
else if ((buf.st_mode &  0170000) == 0020000)
     printf("Character Special File\n");
```

Example:
`filetype.c`

Determining the File Permissions

File Type (4)	Special Permissions (3)	User (3)	Group (3)	Others (3)
1000	000	110	100	000

- You can determine the file permissions by creating a mask by making all the bits zero other than the one of your interest. So to determine the file permissions for the owner, you set the corresponding three bits of user permissions and zero out the rest of the bits
- Then you perform a bitwise **&** of the `st_mode` value with the mask, and check if the specific bit for that permission is set or not. If it is set that means the permission is ON otherwise it is OFF. A sample code snippet is shown:

```
if((buf.st_mode & 0000400) == 0000400)
     printf("Owner has read permission\n");
if((buf.st_mode & 0000200) == 0000200)
     printf("Owner has write permission\n");
if((buf.st_mode & 0000100) == 0000100)
     printf("Owner has execute permission\n");
```

Example:
`filepermissions.c`

`lsv3.c`

This version adds a feature that will display every file in the directory in long listing

lsv4.c

This version adds a feature to display file names in columns (list down and then across). Number of columns depends on the length of the names and total number of names that come in a single line. Number of rows depends on the total number of files in that directory

`lsv5.c`

Add a feature to display the file names after a alphabetic sort. Read all the names in an array, then use `qsort(3)` to sort the array and then display on screen

lsv6.c

Add a feature that displays colorful filenames, i.e., directories in blue, executables in green, tarballs in red, softlinks in pink, character and block special files in reverse video

`lsv7.c`

Add a feature that implement the `-R` option to standard `ls`, i.e., recursively show contents of subdirectories as well

`myfinalls.c`

This is the final version will incorporate all the features with appropriate option characters as close as possible to the standard `ls` utility

myfind.c

mygrep.c

Things To Do

If you have problems visit me in counseling hours. . . .