ls command

LINUXLAB

ESGI

Table of Contents

- History of `ls` ?
- Some useful options and tips
- How does `ls` work ?
- Our own `ls` program

History of `ls`

- One of the oldest and most important commands for Unix-like environments
- Compatible Time Sharing System (CTSS, MIT) 1961 => `listf`
- Multics (AT&T) 1964 => `list`
- Unix (AT&T) 1969 => `ls`

Nowadays the `ls` command we use comes from the GNU foundation.

Some useful options and tips

`ls` with these arguments :

Arguments	Description
`-l`	More information in list format.
`-a`	Hidden files.
`-R`	Recursive, list files in sub-directories.
`-lSh`	Sort by size with human readable.
`-i`	Inodes.
`-n`	Numeric UID and GID.
`-lt`	Sort by date (Last time modified). Add `-r` for reverse

How does `ls` work ?

The shell

The shell is the first program that the user will encounter.

With that, user can communicate with other programs, indirectly with the Kernel and finally the Hardware part :

User > Shell > Other Programs > Kernel > Hardware

It is therefore via the shell that we can type the `ls` program to call and execute it afterwards.

Checks (Aliases and PATH)

First thing done by the shell after we typed `ls` :

- Check if an alias exists

If the alias `ls` exists => replace this value with what we typed.
If the alias does not exist => look for the `ls` command among the paths contained in the `$PATH` variable.

To know all the paths specified in the path we can execute the following command:

```
echo $PATH
```

To find out where the command in question is located we can run these commands:

Command	Description
`which ls`	Gives the path where the command comes from.
`whereis ls`	Search for the command path, source code and manual.
`type -a ls`	Search all possible paths of the command.

Creation of the process

Once the `ls` command is found. The shell will be able to execute it.

To do this, three steps are necessary:
fork, exec and wait

Fork

The shell executes a fork via the `fork()` function.

Our shell will ask the kernel to create a child process from the shell which will be the parent process. So our `ls` command will have its own execution space in some way and can be identified by a PID.

Shell > Fork > Child with PID for `ls`

Exec

Now the shell will call the `exec**()` function which will load the ls program to this new process created to be executed.

Wait

Before execution, the parent process is blocked by the `wait()` function so that the child process can finish before the parent.

The parent will then wait for the end of the child's execution.

We will be sure that the child finishes its execution to avoid it becoming a zombie process.

Our own `ls` program

Execution of the ls command

Now comes our famous ls program.

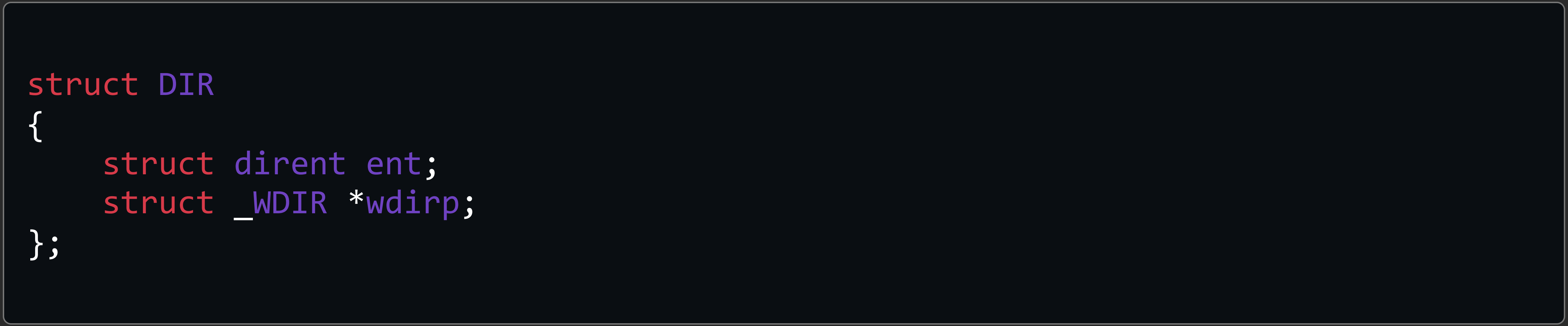
It will mainly read files and folders through kernel space functions.

Main functions for reading files in a directory :

`opendir()`

`opendir()` -> which uses `getdents()` behind.

`opendir()` returns a *DIR* struct :



```c
struct DIR
{
    struct dirent ent;
    struct _WDIR *wdirp;
};
```

`getdents()` returns a *linux_dirent* :

```
struct linux_dirent {
               unsigned long  d_ino;     /* Inode number */
               unsigned long  d_off;     /* Offset to next linux_dirent */
               unsigned short d_reclen;  /* Length of this linux_dirent */
               char           d_name[];  /* Filename (null-terminated) */
           }
```

`opendir()` opens a stream in the directory, and returns a pointer to that stream. The feed is positioned on the first entry of the directory.

If an error occurs it returns null and returns an error code contained in errno. The input stream of the directory usually contains the name and number of the inode.

`readdir()`

It returns a pointer to a dirent structure representing the next entry in the directory stream pointed to by dir. It returns NULL at the end of the directory, or on error.

`readdir()` returns a *dirent* struct :

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

`stat()`

It reads the inode table of the file system.

An inode table contains information about this directory but also the starting location of other files (the inodes of the files contained in this directory).

It finally retrieves the state of the pointed file (inode number, uid, gid, creation time, odification, etc) `stat()` returns a *stat* struct :

```
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 file system 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 */
};
```

Then according to the different arguments `ls` will loop or not in the other directories.

Finally it will be able to display the files and references found in the standard output to the shell.

Now let's code !

Thank you !