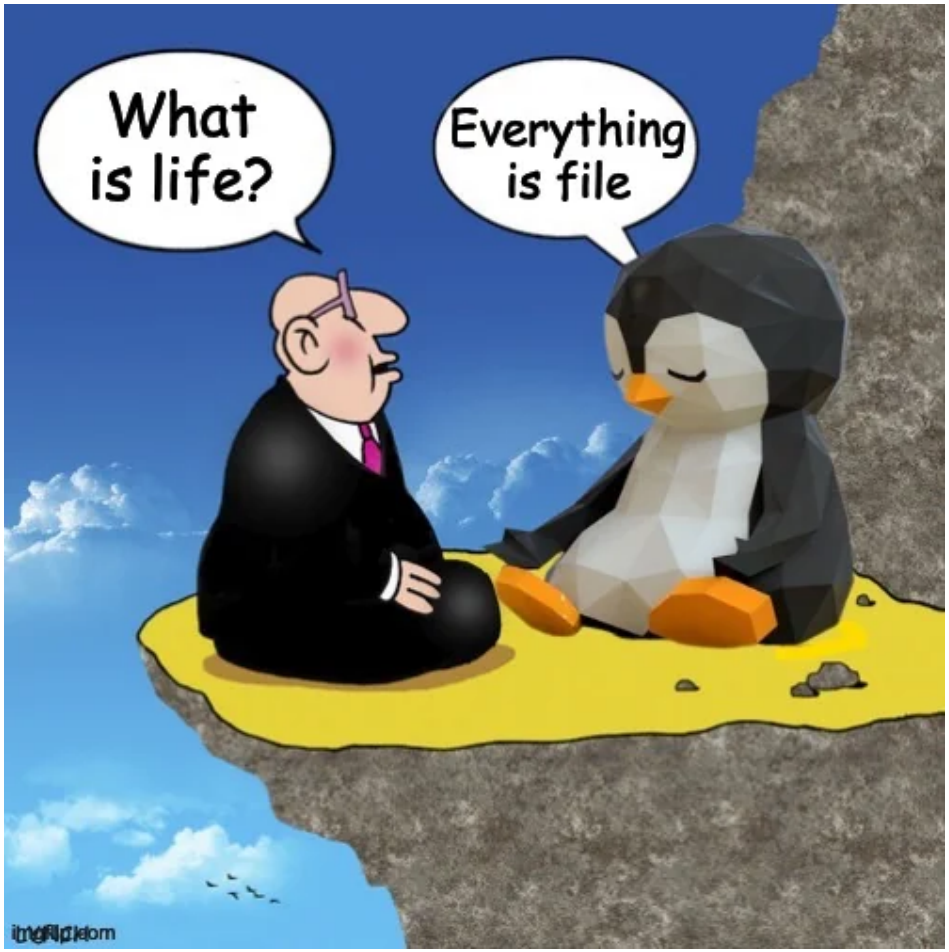
Lab 10: File Systems

Innopolis University
Course of Operating Systems

In Linux, everything is a File



Linux file system

- A simple description of the UNIX system, also applicable to Linux, is this:
- "On a UNIX system, everything is a file; if something is not a file, it is a process."
- A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files.
- Input and output devices, and generally all devices, are considered to be files, according to the system.
- Most files are just files, called regular files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.
- While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

Files

- File types in Linux file system.

Symbol	File type
-	regular file
d	directory
l	symbolic link
c	character device
b	block device
p	named pipe
s	socket

- You can check the file type using *ls -l* command as follows:

```
$ ls -l path
```

- You can get more specific info about the file type using *file* command as follows:

```
$ file path
```

Linux File system hierarchy

The file system in Linux, starts with the root directory (/). Some of the directories listed under the root directory are:

- */bin* - binaries (e.g. the most basic commands such as *ls* and *cp*).
- */boot* - kernel boot loader files.
- */dev* - device files.
- */etc* - core system configuration directory.
- */home* - personal directories for users.
- */lib* - library files that binaries can use.
- */mnt* - temporarily mounted file systems.
- */proc* - information about currently running processes.
- */root* - root user's home directory.
- */sbin* - essential system binaries, usually can only be run by root.
- */tmp* - storage for temporary files
- */usr* - user's installed software and utilities.
- */var* - variable directory, it is used for system logging, user tracking, caches, etc.

Linux file system

Example 1:

- Check the file types in the paths */dev* and */etc*. (Character and block devices will be discussed in lab 12).
- Count the number of directories in the folder **/etc** by running the command line.

```
ls -l /etc | grep ^d | wc -l
```

- Write a hello world program **ex1.c** and check the file type both before and after compilation using *file* command.
- Print “Привет, мир!” (some non-English words) instead of "Hello world!"and run the *file* command again. What is the difference?

inode

- An **inode** (index node) is a data structure that keeps track of all the files and directories within a Linux or UNIX-based filesystem. Every file and directory in a filesystem is allocated an inode, which is identified by an integer known as "inode number". These unique identifiers store metadata about each file and directory.
- All inodes within the same filesystem are unique. inodes operate on each filesystem, independent of the others. So, each filesystem mounted to your computer has its own inodes.
- Inodes store metadata such as type, size, owner ID, group ID, permissions, last access time, etc...
- You can get info about the file metadata using *stat* command.

```
$ stat path
```

- You can get the inode number of a file using *ls -i* command or *stat -c "%i"* command.

```
$ ls -i path
$ stat -c "%i" path
```

inode

Example 2:

- Check the inode of the program **ex1**. Determine the number of blocks that the file has and the size of each block. Print also the total size in bytes and permissions of the file.
- Copy the program **ex1** to **ex2** and check the number of links for the file **ex2**. Check if they have same i-node numbers. Justify your answer.
- Identify the files who have 3 links in the path */etc* by running the following command line:

```
stat -c "%h - %n" /etc/* | grep ^3
```

- What does this number (3 links) represent? Go to the next slide to know.
- **Hint:** Check the manual page (`man 1 stat`).

Linking Files – Hard Links (1/2)

- A directory may contain several filenames that all map to the same i-node number and thus to the same file in the file system.
- Unix call these names pointers or links to the file.
- Hard links are new names for the same i-node. Link count in the i-node keeps track of how many directories contain a name number mapping for that i-node
- Hard links cannot be made to a directory. This restriction means that every subdirectory has one and only one parent directory

Linking Files – Hard Links (2/2)

- Command for creating a hard link:

```
$ ln filename linkname
```

- A hard link and data it links to, must always exist in the same file system.
- Command to find i-node number of a file:

```
$ ls -i filename
```

Hard links

Example 3:

- Write a script *gen.sh* which contains only one line as follows:

```
$ for (( i=0; i<$1; i++ )); do echo $RANDOM >> $2; done
```

- Create a text file *ex1.txt* using *gen.sh* with arguments 10 and ex1.txt
- Link *ex1.txt* to *ex11.txt* and *ex12.txt*
- Compare the content of all files using *diff* or *cat* command. Is there a difference? Justify your answer.
- Check i-node numbers of all files and save the output to the file *output.txt*. Are they different i-node numbers? Justify your answer.
- Check the disk usage of *ex1.txt* file using *du* command.

Hard links

Example 3:

- Create a hard link *ex13.txt* for *ex1.txt* and move it to the folder **/tmp**.
- Trace all links to *ex1.txt* in the current path and another time in the root path (/):

```
$ find <path> -inum inodenumber
$ find <path> -samefile file_path
```

- What is the difference between searching in the current path and the root path (/)? Justify your answer.
- Check the number of hard links of *ex1.txt*.
- Remove all links from *ex1.txt*.

```
$ find <path> -inum inodenumber -exec rm {} \;
```

Linking Files – Soft Links

- A soft link or symbolic link contains **a path** to another file or directory and may point to any file or directory
- Can cross file systems
- Created by

```
$ ln -s <source> <target>
```

- **Note:** We recommend to use absolute paths for the `<source>`. You can get the absolute path of a file using *realpath* command.

Symbolic links

Example 4:

- Delete *./tmp* folder if it exists and create a symbolic link **tmp1** for the folder *./tmp*.
- Run `ls -li`
- Create the folder *./tmp*.
- Run `ls -li`. What is the difference? Justify your answer.
- Create a file *ex1.txt* using *gen.sh* and add it to *./tmp*. Check the folder *./tmp1*.
- Create another symbolic link **tmp1/tmp2** to the folder *./tmp* (symbolic link to itself). Use only absolute paths for the source.
- Create another file *ex1.txt* using the same generator *gen.sh* and add it to the folder **./tmp1/tmp2**.
- Check the content of the sub-folders. Try to access the sub-folders *./tmp1/tmp2/tmp2/tmp2/....* What did you notice?
- Delete the folder ./tmp and check the symbolic links gain.
- Delete all other symbolic links you created.

File Permissions

- Read (r): with read permission we can see the contents of the file
- Write (w): allows us to change the file such as add to a file, overwrite it etc.
- Execute (x): with execute permission we can ask the operating system to run the program

Directory Permissions

- Read (r): list the contents of the directory
- Write (w): add, rename and move files in the directory
- Execute (x): list information about the files in the directory (sometimes called search permission)

File permissions

Example 5:

- Make an empty file *ex5.txt* and try the following:
- Remove write permission for everybody
- Grant all permissions to owner and others (not group)
- Make group permissions equal to user permissions
  - What does 660 mean for *ex5.txt*?
  - What does 775 mean for *ex5.txt*?
  - What does 777 mean for *ex5.txt*?

chmod()

- The read, write and execute permissions are stored in three different places called Owner, Group and Other.
- Display permissions:

```
$ ls -l
```

- There are three sets of rwx determined by 9 bits of i-node information. Usage:

```
chmod u=rwx filename
chmod g=rwx filename
chmod o=rwx filename
chmod a=rwx filename
```

stat() system call (1/2)

The stat() function obtains information about the file pointed to by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable. Syntax (**$ man 2 stat**):

```
int stat(const char *restrict path, struct stat *restrict buf);
```

stat() system call (2/2)

Structure stat:

```
struct stat {
dev_t     st_dev;     /* device inode resides on */
ino_t     st_ino;     /* inode's number */
mode_t    st_mode;    /* inode protection mode */
nlink_t   st_nlink;   /* number of hard links to the file */
uid_t     st_uid;     /* user-id of owner */
gid_t     st_gid;     /* group-id of owner */
dev_t     st_rdev;    /* device type, for special file inode */
struct timespec st_atimespec; /* time of last access */
struct timespec st_mtimespec; /* time of last data modification */
struct timespec st_ctimespec; /* time of last file status change */
off_t     st_size;    /* file size, in bytes */
quad_t    st_blocks; /* blocks allocated for file */
_long    st_blksize;/* optimal file sys I/O ops blocksize */
u_long    st_flags;   /* user defined flags for file */
u_long    st_gen;     /* file generation number */
}
```

Source: https://man7.org/linux/man-pages/man3/stat.3type.html

opendir() function

- The opendir() function opens the directory named by filename, associates a directory stream with it and returns a pointer to be used to identify the directory stream in subsequent operations
- Syntax (**$ man opendir**):

```
DIR *opendir(const char *filename);
```

- **DIR:** a type representing a directory stream.

readdir() function

- The readdir() function returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or on error
- Syntax (**$ man readdir**):

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

- The structure **dirent** which shall include the following members:

```
ino_t  d_ino; // File serial number.
char   d_name[] Filename; // string of entry.
```

- Check this manual page from here.

opendir() + readdir() Example

```c
#include <dirent.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
void traverse(const char *fn) {
    DIR *dir;
    struct dirent *entry;
    char path[1024];
    struct stat info;
    printf("%s\n", fn);

    if ((dir = opendir(fn)) == NULL) perror("opendir() error");
    else {
        while ((entry = readdir(dir)) != NULL) {
          if (entry->d_name[0] != '.') {
            strcpy(path, fn);
            strcat(path, "/");
            strcat(path, entry->d_name);
            if (stat(path, &info) != 0)
              fprintf(stderr, "stat() error on %s\n", path);
              printf("I-node number: %lu\n", info.st_ino);
              printf("hard links: %lu\n", info.st_nlink);
          }
        }
        closedir(dir);
    }
}
int main() {
  puts("Directory structure:");
  traverse("/etc");
}
```

Monitoring filesystem events

- **inotify** utility is an effective tool to monitor and notify filesystem changes. You can specify a list of files and directories that needs to be monitored by **inotify**. This library is used by various other programs.
- The basic steps to use this utility.
  - Create the **inotify** instance by *inotify_init()*.
  - Add all the directories to be monitored to the inotify list using *inotify_add_watch()* function.
  - To determine the events occurred, do the **read()** on the inotify instance. This read will get blocked till the change event occurs.
  - The system call *read* returns list of events occurred on the monitored directories. Based on the return value of **read()**, we will know exactly what kind of changes occurred.
  - In case of removing the watch on directories/files, call *inotify_rm_watch()*.

inotify API

- You can check the manual page (`man 7 inotify`) to know more about this API. You need to include the library `sys/inotify.h`.
- `int inotify_init(void);` initializes a new inotify instance and returns a file descriptor associated with a new inotify event queue.
- `int inotify_add_watch(int fd, cont char* pathname, uint32_t mask);` adds a new watch for the file (everything is file) whose location is pathname. The argument *mask* is used to specify the inotify events to monitor. This function returns the watch descriptor for which the event is occured.
- **read** system call can be called on the inotify instance to determine the event. The obtained buffer is of type *struct inotify_ event*.

inotify API

- The definition of *inotify_event* is as follows:

```
struct inotify_event {
        int wd; // Watch descriptor
        uint32_t mask; // Mask describing event
        uint32_t cookie; // Unique cookie associating
            related events (mostly 0)
        uint32_t len; // Size of name field
        char name[]; // Optional null-terminated name
    };
```

- The *name* field is present only when an event is returned for a file inside a watched directory; it identifies the filename within the watched directory.
- The buffer size for **read** system call should be at least **sizeof(struct inotify_event) + NAME_MAX + 1** for reading one event.
- Check the manual page of inotify `(man 7 inotify)` to know which inotify events are possible to monitor.

Monitoring filesystem events

Example 6:

- Write a program **ex6.c** to monitor the changes to */tmp* directory using *inotify* API.
- The program should print a message "New file created." when the file is created and then it terminates.
- **Hint:** Check this example at https://www.thegeekstuff.com/2010/04/inotify-c-program-example/.
- **NAME_MAX** is the uniform system limit (if any) for the length of a file name component, not including the terminating null character.

Exercise 1 (1/5)

- Write a directory monitoring program **monitor.c** which watches a specific directory (*path*) and reports the changes made to the directory. The changes should be watched are applicable to the directory itself, its sub-directories and all files inside the directory. The path (`path`) of the directory should be passed as a command line argument to the program.
- The report message should identify the type (event) of change and should also print the stat info of the changed item (file/folder). For example if file (f1.txt) is modified, it should prints a message "f1.txt is modified" and the stat info for this file should also be printed to stdout.
- The **monitor.c** program should be able to print the stat info of all files and folders in the path (`path`) once on the startup, and once before termination.
- We assume that we do not have files or folders inside the sub-folders of the watched directory *(path)*.

Exercise 1 (2/5)

- The user can terminate the program **monitor.c** by sending SIGINT signal (pressing Ctrl-c), and your program needs to handle this signal SIGINT such that it prints the stat info of all entries in the directory (*path*) before termination.
- Write another program **ex1.c** which will be responsible for making some changes to the watched directory. This program gets the path (`path`) of the watched directory as a command line argument.
- In **ex1.c**, add a function `find_all_hlinks(const char* source)` to find all hard links which refers to the same i-node of the *source* in the path (`path` which is passed as a command line argument) and it should print them to stdout with their inode numbers and absolute paths (Use *realpath* function).
- In **ex1.c**, Add a function `unlink_all(const char* source)` to remove all duplicates of a hard link. This function should keep only one hard link and then print the path of the last hard link to stdout.

Exercise 1 (3/5)

- In **ex1.c**, Add a function `create_sym_link(const char* source, const char* link)` which will create a symbolic link *link* to *source* in the path (`path` which is passed as a command line argument). Use *symlink* system call.
- Write a test script **ex1_test.sh** which includes an arbitrary number of commands for creating, modifying and removing files and folders in the path (*path*). It also may include commands for creating links and accessing files and folders.
- The program **monitor.c** should monitor the following changes:
  - IN_ACCESS (File was accessed)
  - IN_CREATE (File/directory created in watched directory)
  - IN_DELETE (File/directory deleted from watched directory)
  - IN_MODIFY (File was modified)
  - IN_OPEN (File or directory was opened)
  - IN_ATTRIB (Metadata changed)
- An example of a test script is in this gist.

Exercise 1 (4/5)

- You should run the program **monitor.c** first, then the program **ex1.c**. After that, you can run the **ex1_test.sh** script.
- The main function of the program **ex1.c** should:
  - Create a file **myfile1.txt** contains the text "Hello world." and create two hard links **myfile11.txt**, and **myfile12.txt** to it and add them to the watched directory.
  - Find all hard links to **myfile1.txt** and print their i-nodes, paths, and content.
  - Move the file **myfile1.txt** to another folder **/tmp/myfile1.txt**.
  - Modify the file **myfile11.txt** (its content). Did the **monitor.c** program reported an event for **myfile11.txt**? Justify your answer and add it to **ex1.txt**.
  - Create a symbolic link **myfile13.txt** in the watched directory to **/tmp/myfile1.txt**. Modify the file **/tmp/myfile1.txt** (its content). Did the **monitor.c** program reported an event for **myfile13.txt**? Justify your answer and add it to **ex1.txt**.

Exercise 1 (5/5)

- Remove all duplicates of hard links to *myfile11.txt* only in the watched directory. After this, print the stat info of the kept hard link. Check the number of hard links and explain the difference. Add the answer to **ex1.txt**.

- **Hint:** Check the manual page of *inotify* at https://man7.org/linux/man-pages/man7/inotify.7.html.
- Submit **monitor.c**, **ex1.c**, **ex1.txt** and a script **ex1.sh** to run the program.
- **Hint:** Use **inotify** to track the changes to the directory entries.
- **Hint:** Use *link* system call to create hard links. Check this manual page.

Exercise 2 (Bonus exercise)

- Implement the example 1 in slide 6 and add the commands and answers as comments to **ex21.sh**. Submit the script **ex21.sh** only.
- Implement the example 2 in slide 8 and add the commands and answers as comments to **ex22.sh**. Submit the script **ex22.sh**.
- Implement the example 3 in slides 11 and 12 and add the commands and answers as comments to **ex23.sh**. Submit the script **ex23.sh** and **gen.sh**.
- Implement the example 4 in slide 14 and add the commands and answers as comments to **ex24.sh**. Submit the script **ex24.sh**.
- Implement the example 5 in slide 17 and add the commands and answers as comments to **ex25.sh**. Submit the script **ex25.sh**.

End of lab 10