Printing all subdirectories and files in current directory

Ameet Deshpande (CS15B001)

October 23, 2017

1 Exploring Stat Structure

- *stat* is a system call in C that can be used to get information about files using their file names. It gets its information from a structure which also contains the INode number. Once the Inode Number is obtained, all information about the file can be accessed. The following is the structure of 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 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 */
            };
```

As can be seen, the *st_ino* field gives the inode number of the file. The following code is written to illustrate the use of the *stat* structure.

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

int main(){}
    struct stat s;            // Structure to store file information
        stat("os.txt", &s);   // Store the information about file
        // Print details about the file
        printf("INode Number of file: %ld\n", (long) s.st_ino);
        printf("Size of the file is: %ld KB\n", (long) s.st_size);
        printf("Last accessed %s", ctime(&s.st_atime));
}
```

Note the usage of st_ino to print the inode number and *st_size* to print the size of the file names os.txt. (Code is attached)

The following output is produced when the file is run.

```
INode Number of file: 58448
Size of the file is: 3 KB
```

2 Lising the Directories

- The standard library files in *dirent.h* allow us to read directories. It provides functions *readdir*, *opendir* and *closedir* which can be used as utility functions to read, open and close files. One important catch when we are recursively printing the files and directories inside the current directory is to not include . and ... This is because it will cause a stack overflow as the current directory calls the current directory and the parent directory. There is a statement which checks if the directory name are any of those to avoid this. The program that is turned in basically does a walk over all the directories and files in the current directory and prints the inode number along with it. The walk is done using a *DIR* pointer which stores the opened file's information. After getting the DIR pointer, *DT_DIR* flag is used to check if it is a directory. If it is a directory, a recursive call is made. If it is a file, the information about the file is printed along with the inode. Note that this is a very general procedure, as once the stat structure is obtained for a file, the inode number can be obtained. The inode number contains all the information/metadata with respect to the file. The code is attached below.

```
#include <unistd.h>
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

void list_recursive (const char *name, int spaces){
    DIR *dir;
    struct dirent *entry;

    if (!(dir = opendir(name)))  // If the directory does not exist{
      return;
    }

    while((entry = readdir(dir)) != NULL){
        if(entry->d_type == DT_DIR)  // If the "file" is a directory, call recursively
        {
            char path[1024];
            if(strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)  // If parent directory or current, then dont print
            {                                                                        // Else stack overflow
                continue;
            }
            snprintf(path, sizeof(path), "%s/%s", name, entry->d_name);          // Print the path to a string
            struct stat s;
            stat(path, &s);
            printf("%ld :: ", (long) s.st_ino);
            printf("%*s[%s]\n", spaces, "", entry->d_name);
             list_recursive (path, spaces + 2);
        }
        else{
          struct stat s;
          char path[1024];
          snprintf(path, sizeof(path), "%s/%s", name, entry->d_name);
          stat(path, &s);
            printf("%ld :: ", (long) s.st_ino);
            printf("%*s- %s\n", spaces, "", entry->d_name);
        }
    }
    closedir(dir);
}

int main(void) {
     list_recursive (".", 0);     // Call the function on current directory
    return 0;
}
```

The *snprintf* function is a very helpful utility function which prints to a string what it would have otherwise printed to *stdout*.

- Sample output will look like the following.

```
58393 ::  - a.out
58448 ::  - os.txt
```

```
58408 :: − print_inode_number.c
58501 :: − print_recursive.c
58502 :: [random1]
58507 ::   − file.c
58390 ::   − file2.txt
58503 ::   [random2]
58506 ::     − file.c
58504 ::     [random3]
58505 ::       − file3.c
```

3 Exploring the inode structure

The Inode structure source code is availabele here. It is clear from the same that the stat structure has to use this behind the hood. A few lines from the structure are pasted below.

```
     loff_t              i_size;
    struct timespec      i_atime;
    struct timespec      i_mtime;
    struct timespec      i_ctime;
```

The stat structure gets its information from the above fields that are present in the inode structure.

```
    union {
        struct hlist_head  i_dentry;
        struct rcu_head        i_rcu;
    };
```

From the above, it can be seen that the first field contains the directory entry of the inode. The structure also has fields for locks and semaphores which can be used while reading from or writing to the blocks. The DIR pointers and the Stat structures can thus get all the information needed from the inode structure. The whole structure is attached below for reference. Hardlinks are used in the directory structure. It is used to associate a name with a file on the file system. Note that the name means nothing to the operating system. It is just a way to project the information to the user. Under the hood, everything is implemented using inodes as they contain all the information that is required to read all the files and sub directories.

4 Large Files and Inodes

Linux file system is based on the fact that the OS is made of a large number of files of small size. The inode thus points to direct data blocks which have enough space to store files of small size, like *4 KB*. When the file sizes become larger, the indirect blocks have to be used. The indirect blocks basically have pointers which point to other blocks on disk. There are several levels of indirect blocks. The second level indirect blocks will thus have pointers to blocks which have pointers to the actual data blocks. Thus, it can clearly be seen that in a normal unix file system, accessing large files will be cumbersome and will take a large amount of time compared to smaller files. For small files, the inode entries for indirect blocks will be empty. However as the file becomes larger, there wont be enough space to store the file in direct blocks. The indirect blocks will thus store an address of a block which stores the final data. The advantage is that the file does not have to be stored in contiguous locations and there will be lesser problems in file allocations.

5 EXT4 File System

- This file system was designed to handle larger files which will suffer in the file system described in the preceding section. How it does the same is to use the concept of extents. This basically reduces the fragmentation that the other file systems introduce. Extents can be used to map physical disk sizes of about $128MiB$, using just $4KiB$ of continuous space. The inode in ext4 can store upto 4 extents. When the number of extents required becomes more than 4, extents are indexed by a tree. This makes it more efficient to access data. The file system XFS is recommended when even larger files need to be stored. In EXT3 every-time an append operation is called, a new block allocator was called. When concurrent processes are writing to the same files, all of them are given different blocks and this causes the issue of fragmentation. This is avoided in EXT4 by delaying the allocation of blocks and thus using the same block to write. EXT4 is thus more suited for Concurrent Programming.

6 INode Struct

```
struct inode {
    umode_t          i_mode;
    unsigned short   i_opflags;
    kuid_t           i_uid;
    kgid_t           i_gid;
    unsigned int     i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl   *i_acl;
    struct posix_acl   *i_default_acl;
#endif

    const struct inode_operations *i_op;
    struct super_block   *i_sb;
    struct address_space *i_mapping;
```

```

#ifdef CONFIG_SECURITY
    void            *i_security;
#endif

    /* Stat data, not accessed from path walking */
    unsigned long    i_ino;
    /*
     * Filesystems may only read i_nlink directly.  They shall use the
     * following functions for modification:
     *
     *    (set|clear|inc|drop)_nlink
     *    inode_(inc|dec)_link_count
     */
    union {
        const unsigned int i_nlink;
        unsigned int __i_nlink;
    };
    dev_t           i_rdev;
     loff_t              i_size;
    struct timespec       i_atime;
    struct timespec       i_mtime;
    struct timespec       i_ctime;
    spinlock_t       i_lock;  /* i_blocks, i_bytes, maybe i_size */
    unsigned short          i_bytes;
    unsigned int       i_blkbits;
    enum rw_hint       i_write_hint;
    blkcnt_t       i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t       i_size_seqcount;
#endif

    /* Misc */
    unsigned long    i_state;
    struct rw_semaphore i_rwsem;

    unsigned long    dirtied_when;  /* jiffies of first dirtying */
    unsigned long    dirtied_time_when;

    struct hlist_node  i_hash;
    struct list_head    i_io_list;   /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
    struct bdi_writeback *i_wb;       /* the associated cgroup wb */

    /* foreign inode detection, see wbc_detach_inode() */
    int             i_wb_frn_winner;
    u16             i_wb_frn_avg_time;
    u16             i_wb_frn_history;
#endif
    struct list_head    i_lru;      /* inode LRU list */
    struct list_head    i_sb_list;
    struct list_head    i_wb_list;  /* backing dev writeback list */
    union {
        struct hlist_head  i_dentry;
        struct rcu_head        i_rcu;
    };
    u64             i_version;
    atomic_t        i_count;
    atomic_t        i_dio_count;
    atomic_t        i_writecount;
#ifdef CONFIG_IMA
    atomic_t        i_readcount; /* struct files open RO */
#endif
    const struct file_operations    *i_fop; /* former ->i_op->default_file_ops */
    struct file_lock_context    *i_flctx;
    struct address_space i_data;
    struct list_head    i_devices;
    union {
        struct pipe_inode_info  *i_pipe;
        struct block_device  *i_bdev;
        struct cdev     *i_cdev;
        char            *i_link;
        unsigned    i_dir_seq;
    };

    __u32           i_generation;

#ifdef CONFIG_FSNOTIFY
    __u32           i_fsnotify_mask; /* all events this inode cares about */
    struct fsnotify_mark_connector __rcu    *i_fsnotify_marks;
#endif

#if IS_ENABLED(CONFIG_FS_ENCRYPTION)
    struct fscrypt_info    *i_crypt_info;
#endif

    void            *i_private; /* fs or device private pointer */
} __randomize_layout;
```

7 References

- Wikipedia
- Stack Overflow
- GitHub Repo
- Stat Structure
- Free-Electrons