CS 333 Operating Systems Lab
Autumn 2023
Lab10: the xv6 diaries

The goal of this lab is to understand the key components of xv6 which implement and enable the file system abstraction by solving some interesting tasks.

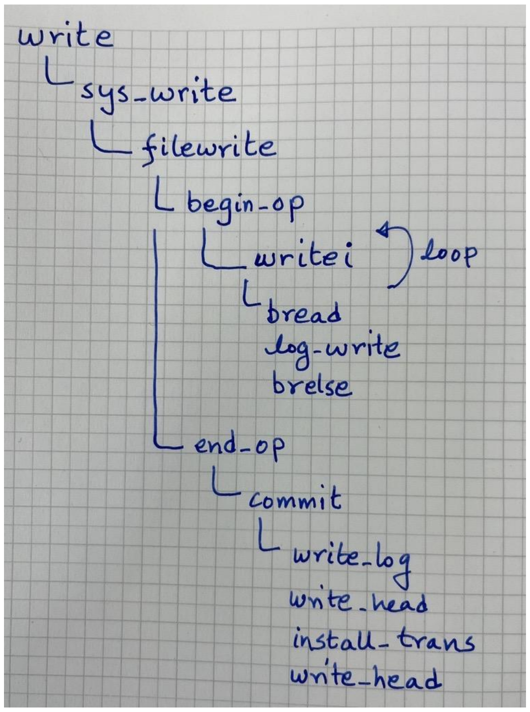
The overall design and details of the xv6 file system are described in Chapter 6 of the xv6 book. This is required background reading.
https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf

Task 1: eyeing the inode

To understand the file system related design and implementations, the following files containing system calls and helper functions —

- **sysfile.c**
  helper functions used to implement file system related system calls
- **file.h and file.c**
  definition of the in-memory inode and file variables/structures
  implementation of various file operations (seeded via the file object)
- **fs.h and fs.c**
  definition of the on disk objects — superblock, inode, root inode number, block size etc.
  file system implementation for the file operations (primary access us using inode)
  (management of blocks, logging, caching, directory management …)
- **bio.c**
  implementation of the buffer cache (block cache) for read/write of disk blocks
- **log.c**
  implementation of log-based writes to the disk (for consistency and other multi-write optimizations).
  begin_op()/end_op() operations mark a transaction and the intent of FS system calls to update the file system on-disk state. Once no FS system calls are in operation or if the log is full, a **commit** operation writes all logs to disk.
- **mkfs.c**
  the user level program that creates fs.img, the image file that emulates a disk for xv6 to use with the set of files available after boot up.
  The disk layout setup is as follows —
  [boot block | sb block | log | inode blocks | free bit map | data blocks]

In this question, we will implement three system calls, the descriptions of each system call is as follows, the actual implementation of these system calls can be done in **file.c.** To solve this part, understanding of the inode structure and file access and manipulation mechanism via data blocks is crucial.



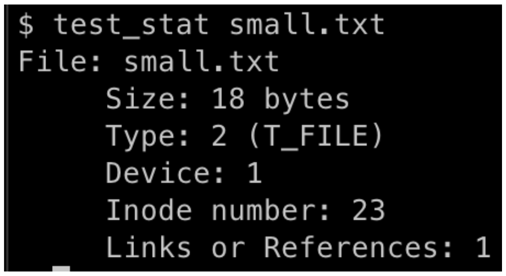
As shown in Figure 1, you can let the logging and block handling operations stay as is and do modifications to functions in **file.c** and **fs.c** and associated header files for the solutions. This lab does not go into details of managing and using the log and buffer cache.

Figure 1: xv6 file write path flow

- Use the user function **`stat`** defined in ulib.c to get metadata information of a file.
  No implementation is needed.
  `int stat(const char *n, struct stat *st);`

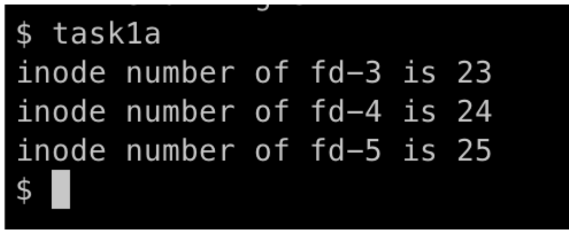
  Lookup **ulib.c** and **stat.h** to write code in a user space program **stat.c** to read information of a specified file name. **stat** uses the system call **fstat** which fetches the on-disk inode information given a filename (path).

  `usage:  test_stat <filename>`



```
$ test_stat small.txt
File: small.txt
      Size: 18 bytes
      Type: 2 (T_FILE)
      Device: 1
      Inode number: 23
      Links or References: 1
```

- **`int getinodenum(int fd)`**
  Implement a system call which, given a file descriptor of an open file, returns the inode number of the file. The inode number of a file is available in the file object.

  ```
  $ task1a
  inode number of fd-3 is 23
  inode number of fd-4 is 24
  inode number of fd-5 is 25
  $
  ```

  Above is the expected output for **task 1a** which is intended to test **getinodenum** implementation. Refer to **task1a.c** where we open 3 files and call **getinodenum** on fds. Compare the inode number reported via **test_stat.**

- **`int getdatablock(int fd, int offset)`**
  Given a file descriptor and an offset value, write a system call to return the on-disk data block number on which the corresponding file data is stored, return -1 if offset is out of range. The relevant files are **file.h, file.c**

  <u>NOTE</u>:-

  1. Data blocks corresponding to a file are stored as 12 direct blocks and 1 indirect block in an inode. Offset resolution will have to consider navigating the whole range.

  2. Check **fs.h** for declarations of variables and constants.

  3. All reads to blocks are via an in-memory buffer cache and each block in this cache is accessed after acquiring a lock. A data block will have to read to look up an indirect mapping block.
     Note that **bread** returns a pointer to a block with the lock acquired. Usually, the calls in **fs.c** (the file system layer) will coordinate block access and acquire and release of the lock. Our system call directly accesses blocks (via functions in bio.c) and hence will have to release the lock as well.
     Refer to the function **`brelse()`** and its usage, for example in **fs.c** look at **`readsb()`**.
     This point is applicable for the next task **readdatablock()** as well.

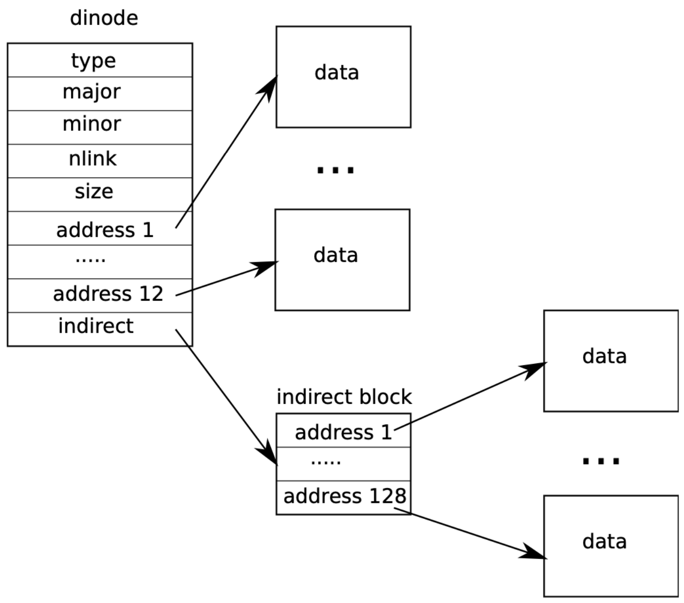
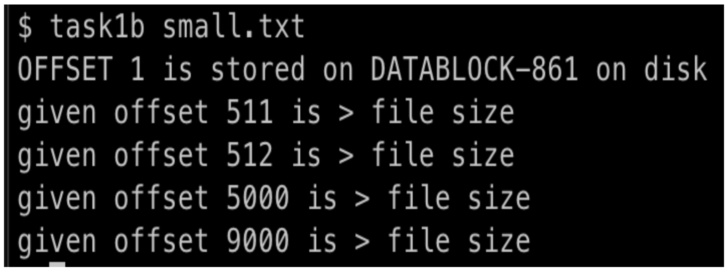
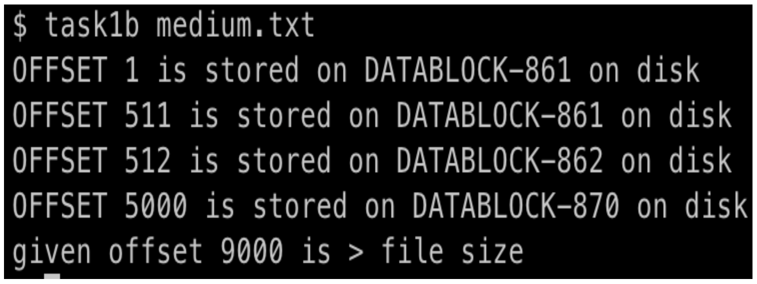


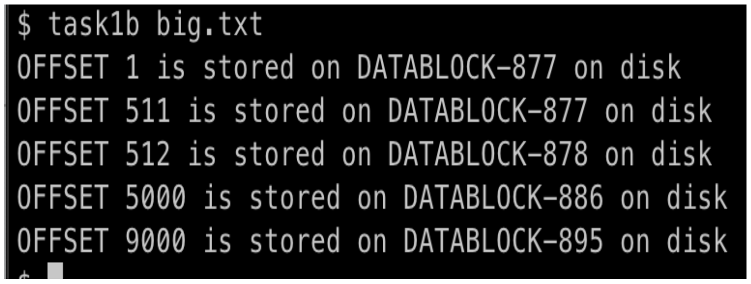
Figure 2: xv6's inode structure and data blocks mapping
image source: :https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf



```
$ task1b small.txt
OFFSET 1 is stored on DATABLOCK-861 on disk
given offset 511 is > file size
given offset 512 is > file size
given offset 5000 is > file size
given offset 9000 is > file size
```



```
$ task1b medium.txt
OFFSET 1 is stored on DATABLOCK-861 on disk
OFFSET 511 is stored on DATABLOCK-861 on disk
OFFSET 512 is stored on DATABLOCK-862 on disk
OFFSET 5000 is stored on DATABLOCK-870 on disk
given offset 9000 is > file size
```



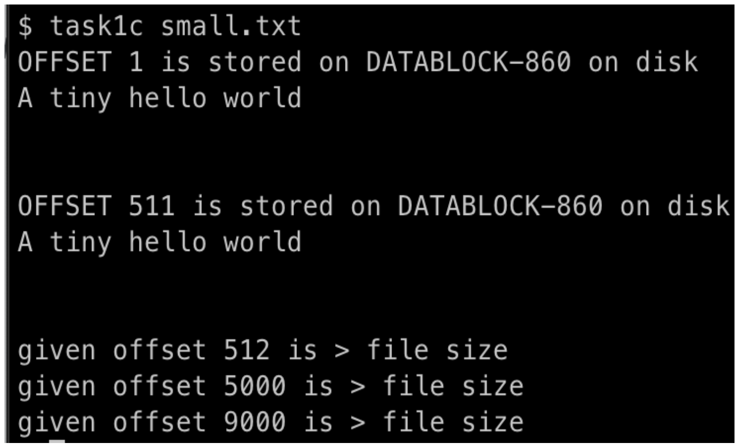
```
$ task1b big.txt
OFFSET 1 is stored on DATABLOCK-877 on disk
OFFSET 511 is stored on DATABLOCK-877 on disk
OFFSET 512 is stored on DATABLOCK-878 on disk
OFFSET 5000 is stored on DATABLOCK-886 on disk
OFFSET 9000 is stored on DATABLOCK-895 on disk
$
```

Below is the expected output for **task 1b** when called on 3 files separately. Notice how different offset values map to the same/different data blocks.

- **`int readdatablock(int device, char *buffer, int db)`**
  Given the device number and the block number, write a system call that reads the contents of the data block from the device into a buffer. Note that this call bypasses the file system to read directly a block from the device.

  NOTE: Be careful while reading the contents of a locked buffer returned by `bread()`, be sure to do `brelse()` after reading buffer contents. A sample usage is at `readsb()` in **fs.c**

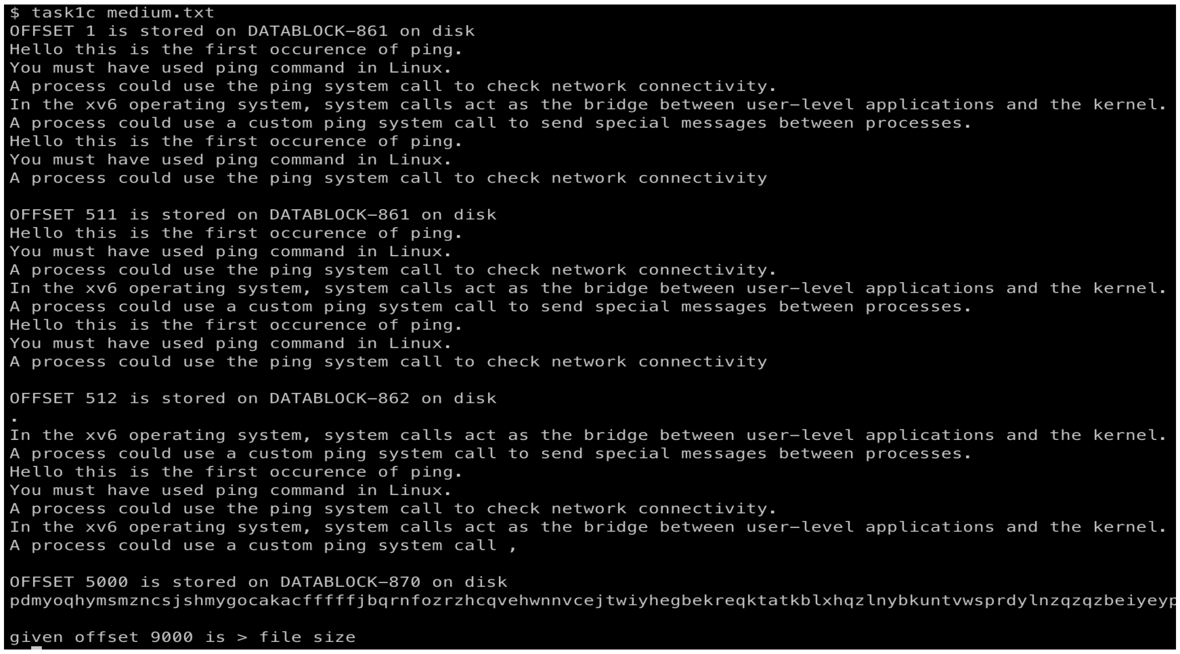
  Below images show the expected output for **task1c** when called on **small.txt, medium.txt** , there is also **big.txt** file which spans across indirect block. Refer to **task1c.c** file to test your implementation.



```
$ task1c small.txt
OFFSET 1 is stored on DATABLOCK-860 on disk
A tiny hello world


OFFSET 511 is stored on DATABLOCK-860 on disk
A tiny hello world


given offset 512 is > file size
given offset 5000 is > file size
given offset 9000 is > file size
```



```
$ task1c medium.txt
OFFSET 1 is stored on DATABLOCK-861 on disk
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity.
In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
A process could use a custom ping system call to send special messages between processes.
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity

OFFSET 511 is stored on DATABLOCK-861 on disk
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity.
In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
A process could use a custom ping system call to send special messages between processes.
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity

OFFSET 512 is stored on DATABLOCK-862 on disk
.
In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
A process could use a custom ping system call to send special messages between processes.
Hello this is the first occurence of ping.
You must have used ping command in Linux.
A process could use the ping system call to check network connectivity.
In the xv6 operating system, system calls act as the bridge between user-level applications and the kernel.
A process could use a custom ping system call ,

OFFSET 5000 is stored on DATABLOCK-870 on disk
pdmyoqhymsmzncsjshmygocakacfffffjbqrnfozrzhcqvehwnnvcejtwiyhegbekreqktatkblxhqzlnybkuntvwsprdylnzqzqzbeiyeyp

given offset 9000 is > file size
```

- **related xv6 files**
  `sysfile.c, file.h, fs.h, file.c, fs.c, bio.c, log.c, buf.h, stat.h defs.h, param.h`

- **related xv6 functions**
  `argfd, fileread, filewrite, writei, readi, bread, bwrite, log_write, brelse, commit, begin_op, end_op, bread(),memmove(), bmap()`

Task 2: does your data talk?

Task2 is to extend the file system to support encrypted data files. Processes use the same file related system calls—open, read, write, close etc., but the file system transparently stores and retrieves data via encryption and decryption of the data before storing and after reading from disk.

Steps to achieve this —

- Add a new operation mode flag **O_ENCRYPT** (refer **fcntl.h**) for usage with the **open** system call. This is the only method to create and access encrypted files.
  The **O_ENCRYPT** mode assumes all such files are encrypted and have read and write access.

  e.g., **`fd = open("filename", O_CREATE | O_ENCRYPT);`**

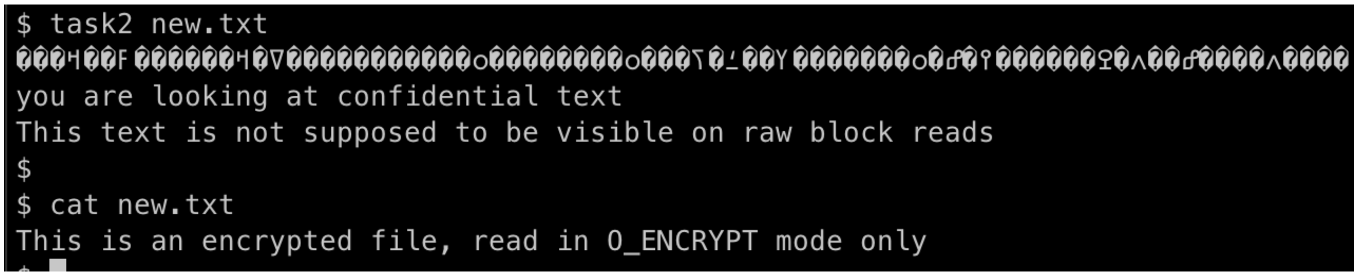
- If an encrypted file is opened in normal read/write mode, it should return an error. Check sample outputs/test cases for reporting such a condition.

  To store the mode in which a file is opened, look up sys_open and check how the type of the file object is updated with the operation mode. A similar variable called **encrypt** will have to be added to the file object/struct and store information about the O_ENCRYPT mode on open (whether it was used for open or not).

- Add encryption related code in the **`filewrite`** function and decryption code in the **`fileread`** function. Use the bitwise NOT operator (**~**) for encryption and decryption.

- Note that raw reads of data blocks of encrypted files should yield encrypted data.

Sample usage



```
$ task2 new.txt
���ᛋ��ᚩ ������ᛋ�∇�������������o���������o���ᛧ�⸍��ᛉ ��������o�ᛞ�ᛘ �������♀�∧��ᛞ�����∧����
you are looking at confidential text
This text is not supposed to be visible on raw block reads
$
$ cat new.txt
This is an encrypted file, read in O_ENCRYPT mode only
```

The program **task2** creates a new encrypted file **new.txt** and writes to it.
The first line of output is via **readdatablock** and displays encrypted data.
The next 2 lines correspond to data fetched via the read system call on the same file opened with the O_ENCRYPT flag.

Further, programs that open encrypted files without the O_ENCRYPT mode, cannot read or write contents from such files. the `cat` program opens a file in O_RDONLY mode so in the example shown cannot open and read from an encrypted file.

Refer **task2.c** for source code of the test case.

Task3: all shortcuts are bad, but some shortcuts are less bad

A ***link*** is a file system abstraction, that is a special file which acts like a shortcut or an alternate name (with possibly a different pathname) to refer to a source file.

Two type of links are possible,

I. **Hard link**
A hard link is a file (directory entry) which points to the same inode on disk.
e.g., **/home/data.txt** and **/home/today/datanow.txt** can be mapped to the same inode number 23 on the disk. This information is also reflected on the on-disk inode via a number of links count. A file delete operation has to take into account the number of links before clearing up the inode.

Hard links are already supported by xv6. Check the user program **ln.c** for implementation and usage.

e.g., execute **`ln small.txt newsmall.txt`**
check outputs of **`ls`** and **`test_stat small.txt`**

II. **Symbolic link/soft link**

A symbolic link (symlink) or soft link is a file (directory entry) that points to another file or directory. On an open or read/write access to a symlink, the operation is redirected to the source file.
Think of it as a signpost that tells you where to find the actual (source) file, allowing you to access it without having to remember its full (actual) path.
With symlinks no inodes are stored as part of the links, only the source (original) files store the actual inode number and all operations are through this file. The links are name aliases and resolve to the source file.

Read More-
https://www.linux.com/topic/desktop/understanding-linux-links/
https://www.geeksforgeeks.org/soft-hard-links-unixlinux/
https://linux.die.net/man/1/hardlink
https://en.wikipedia.org/wiki/Symbolic_link
https://www.futurelearn.com/info/courses/linux-for-bioinformatics/0/steps/201767

As part of this task, enhance functionality of the **ln.c** program to create symbolic links. Specifically, the command **`ln -s source_file target_file`** should create a symbolic link from the (new) target_file to the (original) source_file.

Implement a new system call **`int symlink(char * target, char *path)`** which will be to set up the symbolic link. **path** refers to the pathname of the source file.

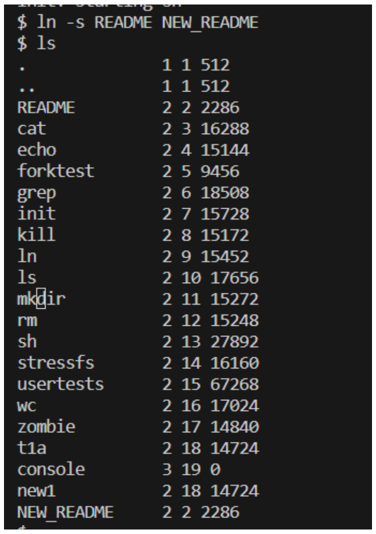
<u>Steps:</u>

- Create a symlink system call.
- Add a new file type (T_SYMLINK) to represent a symbolic link in **stat.h**
- Implement the system call **sys_symlink** in **sysfile.c**
  - Modify and use the **create** function to create a new file of type T_SYMLINK.
  - Using the inode of the new symlink file, write on to the data block path of the source file.
    format to write on the data block could be —
    4 bytes (length of path to source file)
    N bytes (next N bytes store the character sequence of the path)
  - This information can later be used using **readi** while opening the file in sys_open().
  - Carefully update the inode (if written) before unlocking it.
- Make changes to **sys_open** to handle opening symbolic links.
  An **open** on a symbolic link should be handled by looking up the source file name and then resolving it to get the inode of the source file as part of handling open.

- You need to call **begin_op()** in **sys_symlink** before creating a new inode and **end_op()** before returning.

- Use function **create()** with apt arguments (T_SYMLINK) for creating an inode for a symbolic link.
  create acquires a lock to an inode and returns the same inode with the lock.
  Don't forget to unlock before returning from **sys_symlink**.

- xv6 already has a user program implementation of hard links (in **ln.c**) which can be invoked by running the command
  `ln source_file target_files`
  We have provided a modified **ln.c** which incorporates a **-s** option that creates symlinks instead of hard links. Uncomment the call to symlink to use it.



```
$ ln -s README NEW_README
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16288
echo           2 4 15144
forktest       2 5 9456
grep           2 6 18508
init           2 7 15728
kill           2 8 15172
ln             2 9 15452
ls             2 10 17656
mkdir          2 11 15272
rm             2 12 15248
sh             2 13 27892
stressfs       2 14 16160
usertests      2 15 67268
wc             2 16 17024
zombie         2 17 14840
t1a            2 18 14724
console        3 19 0
new1           2 18 14724
NEW_README     2 2 2286
```

Sample usage

A symbolic link to README by the name NEW_README is created.
On **ls,** we can see that the size of both README and NEW_README are the same.

Further, the file type of README is T_FILE (2) and that of NEW_README is_SYMLINK (5).

Further, further, **`cat README`** and **`cat NEW_README`** should yield the same contents.

Note: In the provided output, it is evident that both NEW_README and README share the same inode number and there is an inconsistency in the type of NEW_README, as it should be 5 (T_SYMLINK), but it is incorrectly marked as 2 (T_FILE). This discrepancy arises from the fact that the **ls.c** program contains a call to **open()** that returns the inode of the linked file.

It is worth noting that while this behavior is not consistent with Linux, xv6 has its own set of limitations. The same behavior can also be observed with hard links. Ideally, a symlink should have no inode; instead, it should only be listed as a directory entry in its parent directory. However, for the time being, implementation of this approximate functionality is still of value.

Submission instructions

- Submissions are on the assignment link via Moodle.
  Name your submission as `{rollnumber_lab10}.tar.gz`
  ```
  tar -czvf {rollnumber_lab10}.tar.gz {rollnumber_lab10}
  ```
  (e.g., 200050183_lab10.tar.gz)

- The tar submission should contain all xv6 source files along with the test case files and the makefile

- Run `make clean` before making a tar file for submission

Deadline: 2nd November 2023, 5 pm