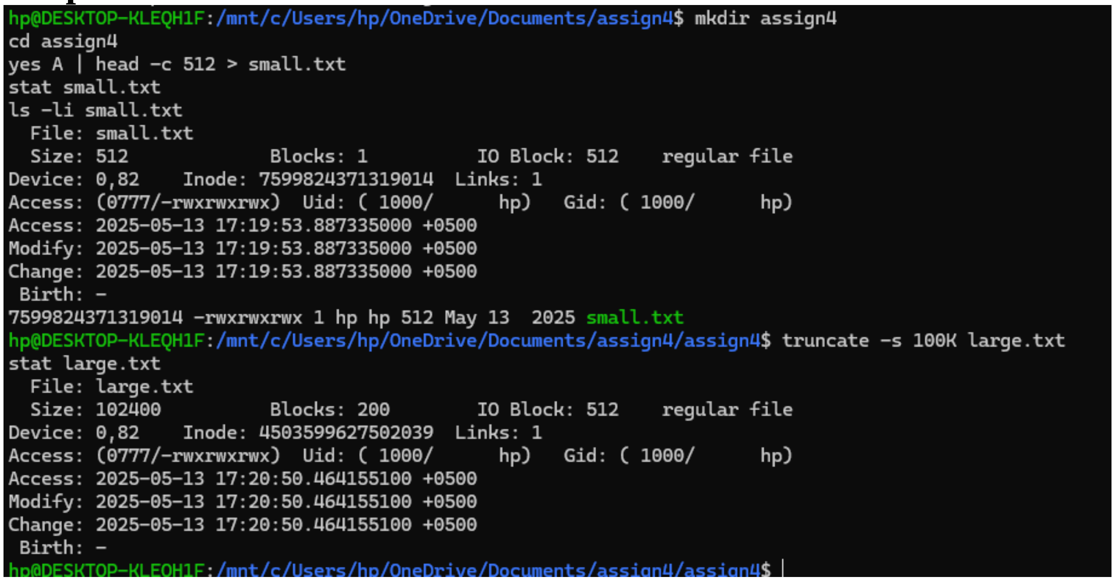
Assignment-4

Operating Systems

Zunaira Abdul Aziz
BSSE23058
Eashal Yasin
BSSE23026

Question # 1: Inodes and File Metadata

Output:



```
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4$ mkdir assign4
cd assign4
yes A | head -c 512 > small.txt
stat small.txt
ls -li small.txt
  File: small.txt
  Size: 512             Blocks: 1          IO Block: 512    regular file
Device: 0,82    Inode: 7599824371319014  Links: 1
Access: (0777/-rwxrwxrwx)  Uid: ( 1000/      hp)   Gid: ( 1000/      hp)
Access: 2025-05-13 17:19:53.887335000 +0500
Modify: 2025-05-13 17:19:53.887335000 +0500
Change: 2025-05-13 17:19:53.887335000 +0500
 Birth: -
7599824371319014 -rwxrwxrwx 1 hp hp 512 May 13  2025 small.txt
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ truncate -s 100K large.txt
stat large.txt
  File: large.txt
  Size: 102400          Blocks: 200        IO Block: 512    regular file
Device: 0,82    Inode: 4503599627502039  Links: 1
Access: (0777/-rwxrwxrwx)  Uid: ( 1000/      hp)   Gid: ( 1000/      hp)
Access: 2025-05-13 17:20:50.464155100 +0500
Modify: 2025-05-13 17:20:50.464155100 +0500
Change: 2025-05-13 17:20:50.464155100 +0500
 Birth: -
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ |
```

1a. What is small.txt's inode number, size and link count?

Based on the output of the stat small.txt and ls -li small.txt commands:

Inode number: The inode number for small.txt is 7599824371319014.

1. The stat command showed Inode: 7599824371319014.
2. The ls -li command showed 7599824371319014 as the first field in its output.
3. **Size:** The size of small.txt is 512 bytes.
4. The stat command showed Size: 512.
5. The ls -li command also reported the size as 512.

Link count: The link count for small.txt is 1.

- The stat command showed Links: 1.
- The ls -li command reported the link count as 1.

Next, the command truncate -s 100K large.txt was executed. This command creates or modifies large.txt to have a size of 100 Kilobytes (102400 bytes).

To determine the number of blocks occupied by large.txt, the stat large.txt command was run.

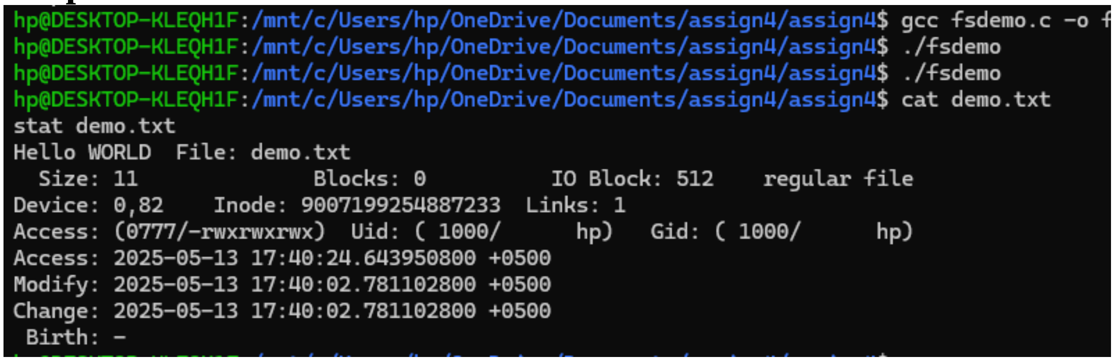
1b. How many blocks does large.txt occupy?

Based on the output of the stat large.txt command:

- The file large.txt occupies 200 blocks.
- The stat command showed Blocks: 200. The output also indicated IO Block: 512, confirming that these are 512-byte blocks, consistent with the file size (200×512 bytes=102400 bytes).

Question # 2: Key File-System System Calls

Output:



```
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ gcc fsdemo.c -o f
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ ./fsdemo
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ ./fsdemo
hp@DESKTOP-KLEQH1F:/mnt/c/Users/hp/OneDrive/Documents/assign4/assign4$ cat demo.txt
stat demo.txt
Hello WORLD  File: demo.txt
  Size: 11              Blocks: 0          IO Block: 512    regular file
Device: 0,82    Inode: 9007199254887233  Links: 1
Access: (0777/-rwxrwxrwx)  Uid: ( 1000/      hp)   Gid: ( 1000/      hp)
Access: 2025-05-13 17:40:24.643950800 +0500
Modify: 2025-05-13 17:40:02.781102800 +0500
Change: 2025-05-13 17:40:02.781102800 +0500
 Birth: -
```

2a. What is the final content of demo.txt? Look at the output of cat demo.txt.

The final content of the file is: Hello WORLD
At first, the program writes "Hello FS\n" to the file, which is 9 characters long.
Then, lseek(fd, 6, SEEK_SET) moves the file pointer to the 7th byte (index 6), which is where the S is.
After that, it writes "WORLD" starting from that position, so it overwrites the S\n part.
So, the original "Hello FS\n" becomes "Hello WORLD".

2b. What is the size of demo.txt? Use stat demo.txt to check.

The file is **11 bytes** in size. Even though the first write was 9 bytes, the second write of 5 bytes starting from position 6 overwrites part of the original content, and extends the file to 11 bytes total (since D ends up at byte position 10).

2c. What does lseek() do in this program? Describe how it affects where data is written in the file.

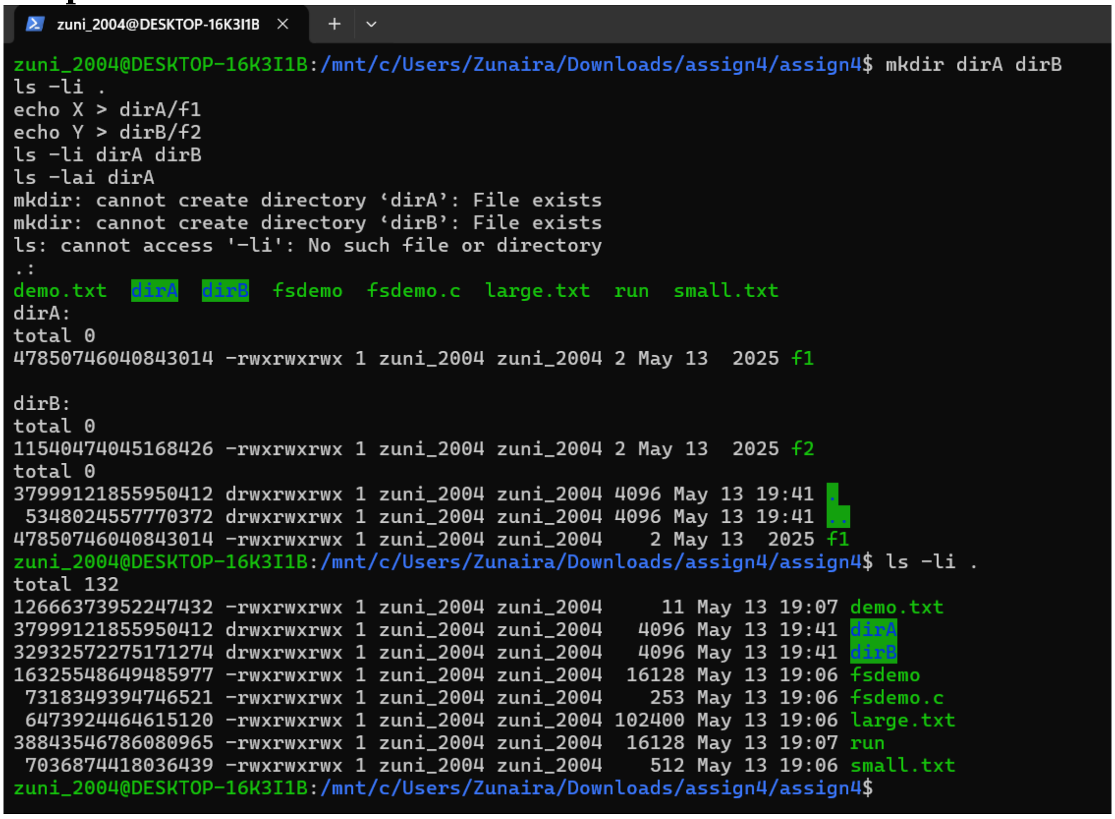
The lseek() function moves the file pointer to a specific position in the file.
In this case, lseek(fd, 6, SEEK_SET) moves it to byte 6, so the next write happens from that exact spot instead of continuing from the end of the file. Basically, it's like moving the cursor before typing new text.

2d. What is the role of fsync()? Why is it important in ensuring that the file data is written?

fsync(fd) makes sure all the data written to the file is actually saved to the disk — not just sitting in memory or cache. This is useful if the system crashes or loses power right after writing. It guarantees that the data won't be lost because it's been physically written to the disk.

Question # 3: Directory Structure & Special Entries

Output:



```
zuni_2004@DESKTOP-16K3I1B:/mnt/c/Users/Zunaira/Downloads/assign4/assign4$ mkdir dirA dirB
ls -li .
echo X > dirA/f1
echo Y > dirB/f2
ls -li dirA dirB
ls -lai dirA
mkdir: cannot create directory ‘dirA’: File exists
mkdir: cannot create directory ‘dirB’: File exists
ls: cannot access '-li': No such file or directory
.:
demo.txt  dirA  dirB  fsdemo  fsdemo.c  large.txt  run  small.txt
dirA:
total 0
47850746040843014 -rwxrwxrwx 1 zuni_2004 zuni_2004 2 May 13  2025 f1

dirB:
total 0
11540474045168426 -rwxrwxrwx 1 zuni_2004 zuni_2004 2 May 13  2025 f2
total 0
37999121855950412 drwxrwxrwx 1 zuni_2004 zuni_2004 4096 May 13 19:41 .
 5348024557770372 drwxrwxrwx 1 zuni_2004 zuni_2004 4096 May 13 19:41 ..
47850746040843014 -rwxrwxrwx 1 zuni_2004 zuni_2004    2 May 13  2025 f1
zuni_2004@DESKTOP-16K3I1B:/mnt/c/Users/Zunaira/Downloads/assign4/assign4$ ls -li .
total 132
12666373952247432 -rwxrwxrwx 1 zuni_2004 zuni_2004     11 May 13 19:07 demo.txt
37999121855950412 drwxrwxrwx 1 zuni_2004 zuni_2004   4096 May 13 19:41 dirA
32932572275171274 drwxrwxrwx 1 zuni_2004 zuni_2004   4096 May 13 19:41 dirB
16325548649485977 -rwxrwxrwx 1 zuni_2004 zuni_2004  16128 May 13 19:06 fsdemo
 7318349394746521 -rwxrwxrwx 1 zuni_2004 zuni_2004    253 May 13 19:06 fsdemo.c
 6473924464615120 -rwxrwxrwx 1 zuni_2004 zuni_2004 102400 May 13 19:06 large.txt
38843546786080965 -rwxrwxrwx 1 zuni_2004 zuni_2004  16128 May 13 19:07 run
 7036874418036439 -rwxrwxrwx 1 zuni_2004 zuni_2004    512 May 13 19:06 small.txt
zuni_2004@DESKTOP-16K3I1B:/mnt/c/Users/Zunaira/Downloads/assign4/assign4$
```

3a. What are the inode numbers of dirA, dirB, f1, and f2?

dirA : 37999121855950412

dirB : 32932572275171274

f1 (inside dirA) : 47850746040843014

f2 (inside dirB) : 11540474045168426

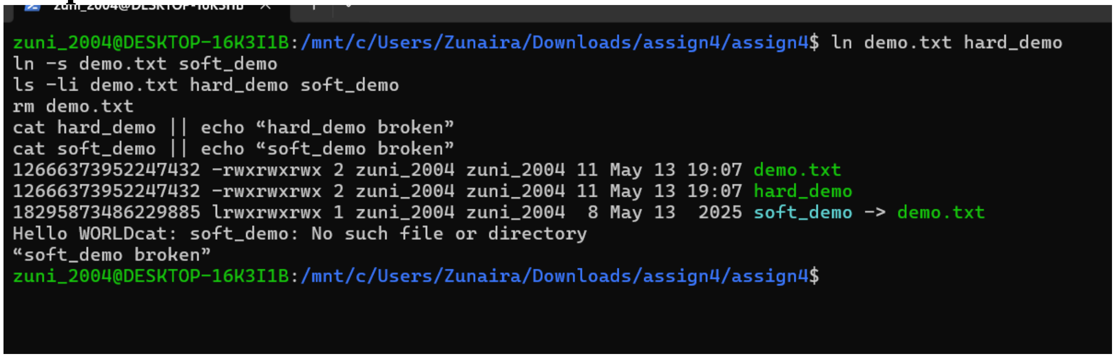
3b. In the output of ls -lai dirA, what inodes do . and .. point to?

. = Refers to dirA itself :37999121855950412

.. = Refers to parent directory (assign4): 5348024557770372

Question # 4: Hard vs. Soft Links

Output:



```
zuni_2004@DESKTOP-16K3I1B:/mnt/c/Users/Zunaira/Downloads/assign4/assign4$ ln demo.txt hard_demo
ln -s demo.txt soft_demo
ls -li demo.txt hard_demo soft_demo
rm demo.txt
cat hard_demo || echo “hard_demo broken”
cat soft_demo || echo “soft_demo broken”
12666373952247432 -rwxrwxrwx 2 zuni_2004 zuni_2004 11 May 13 19:07 demo.txt
12666373952247432 -rwxrwxrwx 2 zuni_2004 zuni_2004 11 May 13 19:07 hard_demo
18295873486229885 lrwxrwxrwx 1 zuni_2004 zuni_2004  8 May 13  2025 soft_demo -> demo.txt
Hello WORLDcat: soft_demo: No such file or directory
“soft_demo broken”
zuni_2004@DESKTOP-16K3I1B:/mnt/c/Users/Zunaira/Downloads/assign4/assign4$
```

4a. Which names still work?

After deleting the original file demo.txt, the hard link named hard_demo still works because it points directly to the same inode as demo.txt. A hard link is just another name for the same file content, so removing one name does not affect the data. However, the symbolic (soft) link soft_demo no longer works. It only stores the pathname "demo.txt" and does not point directly to the file’s inode. Once demo.txt is deleted, soft_demo becomes a dangling symbolic link, and trying to open it results in an error.

4b. What do their inode numbers and link counts show?

demo.txt and hard_demo had the same inode number 12666373952247432, which means they referred to the exact same file on disk.
The link count was 2, showing that there were two names (links) to the same file. Even after deleting demo.txt, the file content is still accessible via hard_demo because at least one link remains.
soft_demo had a different inode number 18295873486229885 and a link count of 1, since it is just a small file that stores the path to demo.txt. Once demo.txt is removed, soft_demo cannot resolve the path and thus becomes broken.