A) Most UNIX filesystems including EXT4FS use a "cylinder group" organization as opposed to the simple layout illustrated near page 5 in the lecture notes. Discuss why this improves performance for a conventional mechanical hard disk, but not necessarily with an SSD.

When creating a file system, the disk is divided into cylinder groups, which are made up disk cylinders. These cylinders are then divided into addressable blocks to control and organize the file structures, which allows for files and meta data to be in the same region of the disk, reducing fragmentation.

Mechanical hard disks care about the positions of data within the disk, as it takes a relative eternity to spin the disk and move the head to the correct sector. When data is next to each other, this eternity can be substantially shortened, which is why cylinder groups work so well on mechanical hard disks.

SSDs don’t have moving parts, and therefore it does not particularly care about where data is stored - in fact, all data in the SSD can be accessed with effectively the same speed, which makes cylinder groups essentially useless. Creating a cylinder groups require additional write operations as well, which reduce the lifespan of the SSD, as an SSD has a limited quanity of writes in its lifespan. This property of SSD is ultimately why cylinder groups won’t necessarily improve the performance of an SSD.

B) A user purchases and installs a 4TB SATA disk to hold a collection of 4K-quality videos and installs the drive on a desktop PC running Linux. Each video file is exactly 2^30 bytes long. Of course, this user follows good security practices and does not download or watch the videos as "root." The expectation was that approx. 4000+ videos would fit on this disk, but the actuality was noticeably less. *Give at least two reasons why this would be the case.*

Reason 1: File system overhead.
Every file system will use some portion of its disk for things like the superblock, inodes, the free map, journalling, and so on. As a result, while the filesystem may have 4TB as advertised, in reality, we may see less than this amount in practice.

Reason 2: Deception
While users may define a TB as 1024^4 bytes, hard drive manufacturers often define a TB as 1000^4 bytes. So while the 4,398,046,511 byte disk we thought we bought might be able to contain all 4,294,967,296+ bytes of movie we wanted, the 4,000,000,000 byte disk we actually bought will not be able to do so.

C) c) At exactly 3PM on Sep 26, 2023 (approximately when you will have started this assignment) I create a file:

```
echo "HELLO" >file123
```

Then at 3:10PM I do

```
cat file123
```

Finally, at 3:30 PM I do:

```
touch -m -t 09131515.15 file123
```

What are the values of atime, mtime and ctime (you can represent these in human-readable form instead of UNIX "seconds since 1970") and why? Assume that the filesystem is not mounted with unusual options such as `relatime`

We'd expect atime to be 3:10PM on September 26, 2023. Atime represents "access time" and is when the file was last accessed, ie, when it was read by the "cat" command at 3:10.

We'd expect mtime to be 3:00PM (at 15 seconds)on September 13, 2023. mtime represents "modification time", and gets updated when we create the file via the ">" command to be 3PM on September 26. However, we change it later via the "touch" command, directly changing the mtime to be September 13, 2023 at 3PM (at 15 seconds).

We'd expect the ctime to be 3:30PM on September 26, 2023. Ctime represents "change time", and is updated when the metadata is changed. It is set to 3PM when the file is created, and updated to 3:30PM at that time when we update the mtime later.

D) A user runs the command `mv /A/B/F1 /A/C/F2;` where `F1` is a fairly large file. This command runs very quickly. Then the user runs `mv /A/C/F2 /A/Z/F3;` but this takes quite a while to run. Why might that be the case?

This could happen for many reasons.
Case 1: Say B and C are within the same disk, but C and Z are different disks.
Moving F1 from B to C and renaming the file F1 to F2 is as easy as changing the inode of the file. The file never gets moved, only the metadata gets changed, which is a very fast (or relatively fast) operation.

Moving F2 from C to Z is as easy as physically copying over the bytes from one filesystem to another, which is to say it is not nearly as easy. As this is a larger file, writing all these bytes from the source to the destination is a longer process, which means it might take a longer amount of time.

Case 2: Say we move the file from B to C, while the disk is not busy.
Than, say we perform a full backup of the disk (or some other read / write heavy operation), and try to move the file from C to Z. As the disk is busy with other operations, such a procedure may take noticably longer than before.

Other cases may cause the disk to perform this next operation slowly, but these are two reasons that may cause this.