**File system:**

Read file system chapter from the xv6 book.

The goal of a file system to provide interfaces to read/write data to persistence storage (e.g., disk). Most of the OS provides directory and file interface. Every file is identified using a unique name, which is the path of the file. A directory may contain more than one file and directory. On xv6 each pathname must begin with the root directory. The root directory may contain other directories and files and so on. For example, the pathname “/usr/share/file.txt” is interpreted as: root directory contains “usr” directory that contains “share” directory.  The “share” directory contains a file called file.txt.

The secondary storage is a disk device where these file and directory are stored. There can be other types of secondary storage, but we will discuss magnetic disk as our secondary storage. Most of the general-purpose computers contain magnetic disks. A magnetic disk provides interfaces to read/write data to sectors. A sector is a contiguous chunk of storage area. A typical sector size used by a disk device is 512 bytes. You can think of disk to be a sequential array of sectors. The number of sectors depends on the size of the disk. A file may require one or more sectors. The file data can be stored on contiguous or non-contiguous sectors, depending on how the file system decides to store them.

A file is divided into blocks. Block size can be multiple of sectors. xv6 chooses block size similar to sector size, but a different OS may choose a different block size. The advantage of having larger block size is more data of the file may get stored sequentially on the disk. We already know that the sequential access on disk is faster (because you don’t have to move the arm and rotate the disk platter multiple times). The drawback of larger block size is fragmentation.

Let us talk about xv6. On xv6, block and sector size are the same. So, an entire block can be stored on a sector. Because a block can be stored in multiple sectors, we need to store these mapping somewhere (think about page-tables). Corresponding to every file and directory xv6 also maintains some metadata that is also stored on the disk. These metadata are called inodes. inode contains the size of file, type of file, and a mapping from blocks to sectors.

What is the maximum file size in xv6?

How do we support large files in xv6?

A directory on xv6 is also a file that contains a list of files and the address of their inodes.

What is the maximum number of files and directories are possible within a directory in xv6?

A link system call takes two path names and makes them alias of each other. In other words, both the names point to same inode.

fd = open (“x/y”, O_CREATE)

link (“x/y”, “x/z”);

unlink (“x/y”);

In the above example, the link command makes “y” and “z” files alias of each other. The unlink system call, unlinks “y” from its parent directory “x”. An inode also contains the number of references from directories to it. In this example, before unlink, the number of links is two. After the unlink, the number of links is one. We cannot delete an inode until the number of links from directories is zero.

Interestingly, we cannot delete a file even if the number of links is zero. Because there might be open file descriptors (e.g., fd), that can be passed to read/write system calls to access the file. So, the file system must wait until all the open descriptors are closed.

The number of open descriptors is not stored on disk because they will go away on power off. The file system keeps the number of open descriptors in an in-memory inode. An in-memory inode caches the disk inode, but it also keeps other information, e.g., the number of open file descriptors.

The structure of the xv6 file system is discussed in the xv6 book.

Which blocks are updated on file creation?

file inode
parent directory inode
parent directory data blocks

Because a file can be read/written by concurrent processes, synchronization must be needed for the correct behavior. E.g., suppose two processes are trying to update the same directory, the concurrent access to directory data might overwrite each other contents. To prevent this, xv6 uses a per-inode lock. This lock is also stored in the in-memory copy of the inode. Before reading or writing to any file/directory, xv6 always acquires the inode lock, to prevent the corruption of data.