# P7: File Systems Part-B

**Due** Dec 11, 2019 by 11:59pm **Points** 50 **Available** until Dec 13, 2019 at 11:59pm

This assignment was locked Dec 13, 2019 at 11:59pm.

This project has two independent parts, both related to the xv6 file system. Part A involves modifying the xv6 file system to include support for symbolic links; Part B involves creating a stand-alone tool that will check the consistency of an xv6 file system image.

The two parts can be completed in either order and have no dependencies. Specifically, when you create your file system checker for Part B, you should assume the symbolic links you added to xv6 in Part B do NOT exist; your file system checker must work with the original, default xv6 file system.

Chapter 6 of the **xv6 book** **(https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** contains incredibly useful information. We always recommend reading that book while you work through the existing source code.

You may work on this project with one project partner. If you work with a project partner on one part, you must work with that project partner on the other part as well.

Note that there are two quizzes related to this project. They are both required and will be worth a portion of your Project 7 grade. The quizzes are to be done on your own and NOT with your project partner. The two quizzes must be completed by Wednesday, Dec 4 at 6:00 pm.

# Objectives

- To understand how to interpret and manipulate basic file system data structures (superblocks, inodes, directories, indirect blocks, bitmaps)
- To understand how soft/symbolic links are implemented (as opposed to hard links)
- To understand how multiple data structures in a file system are related when part of a consistent image and how to detect those inconsistencies

# Part A: Symbolic Links

The current file system in xv6 is very basic: it contains a simple superblock, inode table, bitmaps, and data blocks (which also contain indirect blocks and directories). Most of the xv6 file system data structures are even less sophisticated than that of FFS introduced in the 1980s: xv6 does not support cylinder groups, long file names, large files (beyond a single indirect block), or symbolic links. (One exception in which xv6 is more sophisticated is that xv6 does contain a simple journal/log for crash recovery, but we are ignoring the log in both parts of this project).

In this part of the project, you will update xv6 to contain one slightly modern file system feature: symbolic links.

As you know, symbolic links are different than hard links. Hard links, which are supported by xv6, are simply two different path names that refer to the same inode number; with a hard link, there are no differences across any of the names of a file. When a hard link is *created* to an existing inode, the directory entry for that new link simply contains a reference to the existing inode number and the reference count for that inode is incremented. When a file name is *unlinked* (and the directory entry is removed), the reference count for the corresponding inode is decremented; once the count goes down to zero, there are no more references to that inode and the file can be deleted (i.e., the space for the inode and the data blocks it points to are freed). Operations to *open* a file (and subsequently read/write it) are no different whether a file has just one or multiple hard links to it.

Symbolic, or soft, links, are a reference to another file (or directory, but you do not need to support soft links to directories in this project) in the form of an *absolute or relative path that affects pathname resolution*.

Thus, when a user *creates* a symbolic link from **target** to **link_name**, a directory entry is created for **link_name** to a *new* inode number. What does that new inode point to? The new inode will be of a (new) special type (i.e., T_SYM) and instead of containing direct pointers to regular data blocks for a file (or directory), it will contain a direct pointer to a (special) data block containing just the name, **target**. As the implementer of symbolic links, it is up to you to decide how to construct and interpret this data block. Remember that the target name can either be **absolute** (i.e., contain a full path name starting with the root directory) or **relative** (i.e., should be interpreted relative to the directory containing **link_name**). Note, the file system does NOT check at this time to see if the target name is valid or not.

When a user *unlinks* **link_name**, the directory entry for **link_name** is removed and the corresponding inode and its data block are freed. Remember that nothing is done to the **target** file. (Technically, in order to support hard links *to* symbolic links, the reference count for the link_name inode should be decremented and only freed if the reference count goes to zero, but we will not check this functionality).

Finally, when a user *opens* a symbolic link, the file system will check if the **target** name can be opened or not. If the **target** does not exist when the **link_name** is opened, the *open()* system call should return an error. If the **target** does exist, then future read() and write() operations should be performed on that file named **target** (so, you need to make it look the open system call is actually performed on **target**).

# Details

Specifically, you need to implement the following:

A new system call, **int symlink(char *target, char *link_name)** that behaves as described above. The system call should return an error of -1 if the **link_name** already exists or invalid strings are passed to it; symlink should return 0 when there is not an error.

The behavior of your implementation of symbolic links should generally match that of Linux, unless we specify a simplification that you may assume.

Cases that you should be able to handle:

- Target could be an absolute or a relative path name
- Target exists or does not exist (Note: this is not an error when the symbolic link is *created;* it is an error when the file is *opened*)
- Checking that **link_name** does not already exist in that directory; it is an error if it does

Cases that are **new errors**:

- Target is a directory; if it is, you should return an error when the symbolic link is opened

Cases that you do **not need to handle**:

- New simplifying assumption: The name of the target does not fit within 1 disk block (512 bytes); we will not test the case where the target name > 512 bytes
- A hard link is created to a symbolic link; we will not test this case
- Target could itself be a symbolic link; we will not test this case

## Hints

You will want to look through and understand the files **sysfile.c** and **fs.c**.

Start by just creating the new system call with functionality similar to the existing **link()** system call, but with some changes. If we can't call **symlink()**, we can't test any functionality...

Next, as part of implementing **symlink(**) figure out how to add a new directory entry for the link_name. Figure out how to allocate a new inode with a new type. Figure out how to allocate a corresponding data block and store the name of the target in that data block. Your symbolic link is now created. This is absolutely essential as well.

Next, figure out how to open a symbolic link and return the inode of the target file. If you can't open a symbolic link, there isn't much use to the file. First handle the case where the target file actually exists before worrying about error cases. Focus on absolute path names before relative path names for the target.

Finally, implement the functionality to delete the symbolic link. You could argue that your file system still sort of works even if you can't delete files.

The very last things to worry about are probably handling the error cases where the target name does not exist or is a directory.

# Part B: File System Checking

In this part of the assignment, you will be developing a working **file system checker.** A checker reads in a file system image and makes sure that it is consistent. When it isn't, the checker takes steps to repair the problems it sees; however, you won't be doing any repairs to keep your life at the end of the semester a little simpler.

For this project, you will use the xv6 file system image as the basic image that you will be reading and checking. The file **fs.h** includes the basic structures you need to understand, including the superblock, on disk inode format (struct dinode), and directory entry format (struct dirent). The tool **mkfs.c** will also be useful to look at, in order to see how an empty file-system image is created.

Much of this project will be puzzling out the exact on-disk format xv6 uses for its simple file system, and then writing checks to see if various parts of that structure are consistent. Thus, reading through mkfs and the file system code itself will help you understand how xv6 uses the bits in the image to record persistent information.

# A Basic Checker

Your checker should read through an existing xv6 file system image and determine the consistency of the following properties. When one of these does not hold, print the corresponding error message to **stderr** and exit immediately with **exit(1)**. If there are no errors, you should exit with return code 0. If there are multiple errors, you should find only the first one in the following list and then exit.

### Individual Inode Checks

1. Each inode is either unallocated or one of the valid types (T_FILE, T_DIR, T_DEV). **ERROR: bad inode.**
2. For in-use inodes, the size of the file is in a valid range given the number of valid datablocks. **ERROR: bad size in inode.**

### Directory Checks

1. Root directory exists, and it is inode number 1. **ERROR: root directory does not exist.**
2. The . entry in each directory refers to the correct inode. **ERROR: current directory mismatch.**

### Bitmap Checks

1. Each data block that is in use (pointed to by an allocated inode), is also marked in use in the bitmap. **ERROR: bitmap marks data free but data block used by inode.**
2. For data blocks marked in-use in the bitmap, actually is in-use in an inode or indirect block somewhere. **ERROR: bitmap marks data block in use but not used.**

### Multi-Structure Checks

1. For inode numbers referred to in a valid directory, actually marked in use in inode table. **ERROR: inode marked free but referred to in directory.**

2. For inodes marked used in inode table, must be referred to in at least one directory. (Note: you do not need to ensure that the directory is reachable from the root, just that the inode is in some directory.) **ERROR: inode marked in use but not found in a directory.**

## Other Specifications

Your server program must be invoked from the Linux prompt exactly as follows:

```
prompt> fscheck file_system_image
```

The image file is a file that contains the file system image. If the file system image does not exist, you should print **ERROR: image not found.** to stderr and exit with the error code of 1. If the checker detects one of the errors listed above, it should print the proper error to stderr and exit with error code 1. Otherwise, the checker should exit with return code of 0.

## Hints

The order that the errors are presented in the list above are a strong hint for the order in which you should tackle the functionality of this project. The errors listed earlier are easier to identify than later errors and usually involve fewer data structures. You often will use information you obtained in an earlier check in later checks.

We strongly recommend that you implement each check in order and verify that each is working correctly before moving on to later checks.

You will probably want to use **mmap()** for the project.

Make sure to look at **fs.img** , which is a file system image created when you make xv6 by the tool **mkfs**. The output of this tool is the file **fs.img** and it is a consistent file-system image. The tests, of course, will put inconsistencies into this image, but your tool should work over a consistent image as well. Study mkfs and its output to begin to make progress on this project.

You will probably want to use the Linux tool **hd** to take a hex dump of the fs.img file so that you can understand the on-disk structure of the file system in detail.

# Handin

Both project partners should turn in identical copies of all code for both parts of project 7.

In your ~cs537-1/handin/<login>/p7 directory, you should now see two subdirectories: PartA and PartB.

Be sure to follow all THREE steps below.

1. In your p7 directory, put a copy of **partners.txt** that has the same format as previous projects. Again, this file goes in the top-level **p7** directory, NOT in PartA or PartB. You must work with the same partner for both parts of p7. As before:

So that we can tell who worked together on this project, each person (even if you worked alone) should place a file named **partners.txt** in their handin/p7 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2.

If you worked alone, your **partners.txt** file should have only one line.

There should be no spaces within your first or last name; just use spaces to separate fields.

2. To turn in Part A, create a new directory called **xv6** within **p7/PartA.** You should copy all of your xv6 code (along with a Makefile) to this xv6 directory. Note that your Makefile can **NOT** contain rules for making new user programs. **The presence of extra user programs is known to make some of the test cases fail.**

3. To turn in Part B, copy all of the files that are needed to create the executable **fscheck.** You must **also** copy any files that you need to include from the xv6 code base to compile your fscheck (e.g., fs.h). You must copy these files even if you did not make any changes to them. Do not copy files from xv6 that you do not need. You must also include a Makefile such that we can type "make fscheck" to create your executable.