Project 4

Project 4: File System

Note: The project description is long. It might be useful to print it out for easy reference.

Important Dates

Out: Monday, November 14th, 2016.

Due: Wednesday, November 23rd, 2016 (11:59 PM CDT).

Submission instructions

I have created a new set of groups for Project 4 and copied over the groups you used for Project 3. However, you can feel free to move about between groups or form new groups as you wish. The list of available Project 4 groups can be found here (https://canvas.northwestern.edu/courses/43589/groups) .

To submit, make sure your xv6 directory contains a file called **team.txt**, which contains all the netids of the members of your group on separate lines. Be sure to type **make clean** in your xv6 directory, then cd up to the directory that contains your xv6 directory. Create a .tar.gz of your xv6 directory by typing:

```
$ tar -zcvf xv6-project4.tar.gz xv6
```

You must then upload the .tar.gz on the assignment page on Canvas here: https://canvas.northwestern.edu/courses/43589/assignments/276358 (https://canvas.northwestern.edu/courses/43589/assignments/276358) (https://canvas.northwestern.edu/courses/43589/assignments/276357)

Rubric

	Points Possible
team.txt file is included and correct	5
project 4 tests	95
TOTAL	**100**
extra credit	20

(https://canvas.northwestern.edu/courses/43589/assignments/276357) (https://canvas.northwestern.edu/courses/43589/assignments/276357)

Resources

- Some useful GDB and QEMU commands: https://pdos.csail.mit.edu/6.828/2016/labguide.html (https://pdos.csail.mit.edu/6.828/2016/labguide.html)
- xv6 textbook: https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf (https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf) (note that we are using a version of the code from 2012)
- Guide to xv6 codebase: https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf (https://pdos.csail.mit.edu/6.828/2012/xv6/xv6-rev7.pdf)
- Hexadecimal converter: https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html (https://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html)

Project Overview

In this project you will implement file system tagging in xv6. This project must be completed in groups of 2-3 people.

Project (100 points)
This project requires you to create three new syscalls that will allow files in the xv6 file system to be tagged with key-value pairs. As an example, you might have an essay for your history class saved as a file in xv6. You could tag this file as "class": "History 101". In this example, the key is "class"

and the value is "History 101".

The syscalls you will write are **tagFile**, **getFileTag**, and **removeFileTag**. At a high level, they do what their names suggest. But we will describe them in detail below.

Extra credit (20 points)
For extra credit, there are new syscalls and some additional features that you can add.

Educational Objectives

This project aims to achieve several educational objectives:

- Understand how a unix-like file system is structured.
- Understand inodes
- Understand data blocks.
- Practice writing your own test cases.
- Become the computer science equivalent of a Paladin.

Getting Started

- Read Chapter 6 of the **xv6 textbook** (https://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf) to learn how the xv6 file system is structured. Follow along in the code as you read.
- Trace the code path of a couple of the file-related syscalls. For example, look at **sys_open** and **sys_write**. Understanding these syscalls will help you understand how to write the syscalls for this project..
- You'll probably write most of your code in fs.c, but as usual you will touch many other files as well, including files you haven't really worked in yet this quarter such as sysfile.c, file.c, file.h, fs.c and fs.h. Rumor has it you may also need to change the **mkfs** function in mkfs.c (although my implementation didn't require changing it).

The Code

You can start from a clean version of the **original xv6 code** (http://www.aqualab.cs.northwestern.edu/class/333-eecs343-xv6#download-xv6) or, if you wish, you can start from your code from a prior project.

Project Description and Requirements

100 points

We will explain the project by going over the syscalls that you need to create in detail. Further below, we will discuss a simplifying assumption that should make your solution easier to implement.

1) tagFile

The **tagFile** syscall tags a file with a key-value pair. For example, you may want to tag a file with "author": "Jane". In this example, "author" is the key and "Jane" is the value. Here is the function signature of **tagFile**:

```
int tagFile(int fileDescriptor, char* key, char* value, int valueLength);
```

The first argument, **fileDescriptor**, is a file descriptor such as one returned by the **open** syscall. It is just an integer, but it is used to identify a file (or a directory, or a pipe, or a device) in xv6.

The second argument, **key**, is a null-terminated string which represents the key part of the tag. The string is at least 2 characters long and at most 10 characters long, including the null termination byte (or 1 to 9 characters not counting the null termination byte).

The third argument, **value**, is a string that represents the value part of the tag. This string is not null-terminated.

The fourth argument, **length**, is the number of characters in the **value** string. This argument is necessary since **value** is not null-terminated.

The syscall should return 1 for success and -1 for failure. If the file already has a tag with the specified key, the existing value that corresponds to that key is overwritten. In other words, you want to be able to change "author": "Jane" to "author": "Mary".

Here is an example of a simple user program that makes use of the **tagFile** syscall:

```
#include "types.h"
#include "user.h"

#define O_RDONLY 0x000
#define O_WRONLY 0x001
#define O_RDWR 0x002
#define O_CREATE 0x200

int
main(int argc, char *argv[])
{
 int fd = open("ls", O_RDWR);

 char* key = "type";
 char* val = "utility";
 int len = 7;
 int res = tagFile(fd, key, val, len); // tag file as "type": "utility"
 if(res < 0){
  printf(1, "tagFile error.\n");
 }

 close(fd);

 exit();
}
```

Requirements (and hints)

Here is a list of specific requirements related to the **tagFile** syscall:

- The **tagFile** syscall must use the exact function signature that we have provided above.
- The **tagFile** syscall must tag the file specified by the file descriptor with the key-value pair that is passed in.
  - Hint: you'll have to figure out where to store these tags. One reasonable approach is to repurpose one of the inode's direct blocks for tag storage. You can read more about inodes starting on page 77 of the xv6 textbook.
- If the file already has a tag with the specified key, then the specified value will overwrite the stored value. In other words, if a file is tagged with "language": "English", you should be able to use **tagFile** to change the tag to "language": "Java".
- The **tagFile** syscall must validate the arguments passed in and return -1 as necessary to indicate an error. Some cases to consider:
  - The file descriptor must be opened in write mode in order to tag a file successfully.
  - The key must be at least 2 bytes (including the null termination byte) and at most 10 bytes (including the null termination byte).
  - You can restrict the disk space allotted to tags to 512 bytes per file. In other words, you can require that all tag information for a given file must fit within a single 512-byte disk block. If there isn't sufficient tag space for tagFile to complete, you can simply return -1.

2) removeFileTag

This syscall simply removes a tag (specified by the key) from a file. Here is the function signature:

```
int removeFileTag(int fileDescriptor, char* key);
```

The first argument, **fileDescriptor**, indicates which file will be untagged.

The second argument, **key**, indicates which tag will be removed.

Here is an example of a simple user program that makes use of the **removeFileTag** syscall.

```
#include "types.h"
#include "user.h"

#define O_RDONLY 0x000
#define O_WRONLY 0x001
#define O_RDWR 0x002
#define O_CREATE 0x200

int
main(int argc, char *argv[])
{
 int fd = open("ls", O_RDWR);

 char* key = "type";
 int res = removeFileTag(fd, key); // removes the tag with key "type"
 if(res < 0){
   printf(1, "removeFileTag error.\n");
 }

 close(fd);

 exit();
}
```

Requirements (and hints)

Here is a list of specific requirements related to the **removeFileTag** syscall:

- The **removeFileTag** syscall must use the exact function signature that we have provided above.
- The **removeFileTag** syscall must remove the specified tag from the specified file.
- The syscall should return -1 to indicate an error. Here are some cases to consider:
  - If the tag specified by **key** cannot be found or is invalid, return -1.
  - If the file descriptor is not open and writable, return -1.
- The syscall should return 1 to indicate success.

3) getFileTag

This syscall simply gets the value of a tag (specified by the key) for a given file. Here is the function signature:

```
int getFileTag(int fileDescriptor, char* key, char* buffer, int length);
```

The first argument, **fileDescriptor**, indicates which file's tags to search.

The second argument, **key**, indicates which tag to read.

The third argument, **buffer**, is a buffer. The syscall will write the tag's value to this buffer.

The fourth argument, **length**, is the length of the buffer specified by **buffer**.

The syscall returns the length of the tag value, or -1 to indicate failure.

Here is an example of a simple user program that makes use of the **getFileTag** syscall.

```
#include "types.h"
#include "user.h"
```

```
#define O_RDONLY 0x000
#define O_WRONLY 0x001
#define O_RDWR 0x002
#define O_CREATE 0x200

int
main(int argc, char *argv[])
{
 int fd = open("ls", O_RDONLY);

 char* key = "type";
 char buffer[18];
 int res = getFileTag(fd, key, buffer, 18);
 if(res <= 18){
   printf(1, "%d: %d\n", key, buffer);  // prints "type: utility" (assuming tagFile
                                        // was previously used to set the tag value as "utility"
 } else{
   printf(1, "buffer too small.\n");
 }

 close(fd);

 exit();
}
```

Requirements (and hints)

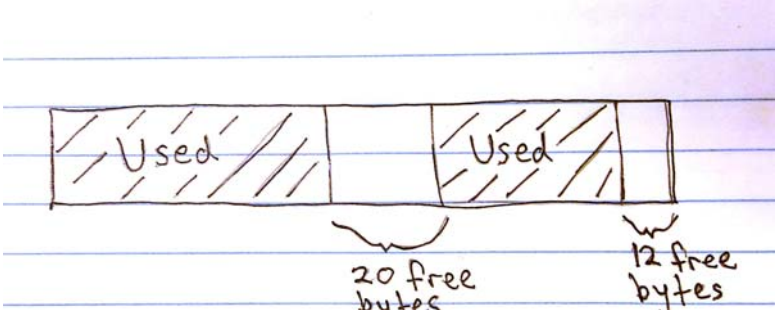
Here is a list of specific requirements related to the **getFileTag** syscall:

- The **getFileTag** syscall must use the exact function signature that we have provided above.
- The syscall should return the length of the value part of the specified tag.
- The value of the specified tag should be written to **buffer**.
- If the length of the value is longer than **length**, the syscall should return the actual length of the value. This allows the user to decide whether to allocate a larger buffer and try again.
- The syscall should return -1 to indicate failure. Here are some cases to consider:
  - If the key cannot be found or is invalid, return -1.
  - If the file descriptor is not open and readable, return -1.

Simplifying Assumption

There is a complexity that arises after adding several tags to a file, then removing some tags, then adding more tags. The problem is internal fragmentation. Here is an explanation of the problem.

Suppose we add 10 different tags to a file. Suppose these tags use up 500 bytes of the 512-byte disk block. Suppose we then remove one of these tags from somewhere in the middle of the disk block. Let's say this removed tag frees up 20 bytes in the middle of the block. So now we have 20 free bytes in the middle of the block and 12 free bytes at the end of the block, like this:



Now, if we try to add a new tag that requires 25 bytes of space, we'd like to say that there is enough room because there is a total of 32 free bytes. But the free space is fragmented, so in order to add the new tag, the free space would first have to be consolidated into a contiguous series of at least 25 bytes.

If you wish, you may sidestep this issue by enforcing a maximum length on the tag value of 18 bytes. You can then represent each tag as a 32-byte struct like this:

```
struct Tag {
 char key[10];    // at most 10 bytes for key, including NULL
 char value[18];  // at most 18 bytes for value, including NULL
                  // 4 bytes available for bookkeeping, etc, if needed
};
```

This approach allows you to treat the 512-byte disk block as an array of 16 Tag structs. This way, when adding a new tag, you just need to search the disk block for an unused tag entry.

Since we are explicitly allowing this simplified approach, we promise that the autograder won't attempt to store a tag value that is longer than 18 bytes. However, the autograder will attempt to store up to 16 different tags on a single file.

Testing

We are providing some basic tests to help you get started. The purpose of these tests is:
1) to give you a testing framework,
2) to help you understand how a user program might invoke your syscalls, and
3) to make it easy to add your own tests.

As always, the provided tests are intentionally not comprehensive and we strongly recommend that you add your own tests. If you are having trouble thinking of your own tests, see the list of Requirements. Typically, each requirement can be translated into *at least* one test. Sometimes you can write many many *many* tests for a single requirement.

As a reminder, modern software engineering principles emphasize writing your tests *before* writing your code. This is taught at Northwestern in EECS 111 as part of the **Design Recipe** (http://www.htdp.org/2001-01-18/Book/node14.htm). In industry, you may hear it described as **test-driven development** (https://en.wikipedia.org/wiki/Test-driven_development). The point is, we recommend writing tests first in order to force yourself to define the desired outcome of your code. Then after you've written your code, you can run your tests to get immediate feedback on whether your code is correct or not.

Without further ado, here is how to use the provided tests. Download the .tar.gz below, copy it to a lab machine, and extract it. Inside, you will find a README with instructions on how to run the provided tests as well as how to add your own tests. If anything is unclear or doesn't work, please post to Piazza.
provided-basic-tests.tar.gz (https://canvas.northwestern.edu/courses/43589/files/2996885/download?wrap=1)

Extra Credit Description and Requirements

20 points

For extra credit, you can extend the basic functionality above in a few ways.

1) a new syscall: **getAllTags**

Here is the function signature:

```
int getAllTags(int fileDescriptor, struct Key keys[], int maxTags);
```

The first argument, **fileDescriptor**, designates which file to get the tags of.

The second argument, **keys**, will be filled in with a list of keys from all the tags that apply to the specified file. **struct Key** is simply this:

```
struct Key {
  char key[10];  // at most 10 bytes for key
}
```

Make sure that this struct is available via either types.h or user.h so that we don't have to include any additional files when we test your code.

The third argument, **maxTags**, is the size of the **keys** array. This tells the kernel the maximum number of keys that the caller can accommodate in its buffer.

The syscall returns the actual number of tags on the specified file, or -1 for failure. Here is some example code demonstrating how this syscall may be used:

```
#include "types.h"
#include "user.h"
// make sure that struct Key is included via either types.h or user.h above

#define O_RDONLY 0x000
#define O_WRONLY 0x001
#define O_RDWR 0x002
#define O_CREATE 0x200

int
main(int argc, char *argv[])
{
 int fd = open("ls", O_RDONLY);

 struct Key keys[16];
 int numTags = getAllTags(fd, keys, 16);
 if(numTags < 0){
   exit();
 }

 if(numTags > 16){
   numTags = 16;
 }
 char buffer[18];
 int i;
 printf(1, "Here is a list of this file's tags:\n");
 for(i = 0; i < numTags; i++){
   int res = getFileTag(fd, keys[i], buffer, 18);
   if(res > 0){
     printf(1, "%d: %d\n", keys[i], buffer);
   }
 }
 close(fd);

 exit();
}
```

2) a new syscall: **getFilesByTag**

This syscall returns a list of files that are tagged with the specified key and value. Here is the function signature:

This syscall returns a list of files that are tagged with the specified key and value. Here is the function signature:

```
int getFilesByTag(char* key, char* value, int valueLength, char* results, int resultsLength);
```

The first argument, **key**, designates the key of the tag we are searching on.

The second argument, **value**, designates the value of the tag we are searching on.

The third argument, **valueLength**, designates the length of **value**.

The fourth argument, **results**, is a buffer of characters that should be filled in with the names of files that are tagged with the specified **key** and **value**. In the **results** buffer, each filename should be separated by a single NULL character.

The fifth argument, **resultsLength**, specifies the length of the **results** buffer. This is the maximum number of bytes that the kernel can write to the **results** buffer.

The syscall returns the actual number of files that have been tagged with the matching **key** and **value**, or -1 to indicate failure.

3) Additional feature: **Allow tag values to have an arbitrary length** (i.e. relax the **simplifying assumption** discussed above). Well, not quite arbitrary. Since the minimum key length is 2 bytes, and the disk block size is 512 bytes, and some bytes might be required for bookkeeping, let's say the value length can be between 0 (empty string) and 500 bytes. (If you choose to use NULL termination for the value then the empty string is actually 1 byte).

Adding this arbitrary length feature will require you to deal with the internal fragmentation discussed above. That is, when a tag is added/changed/removed, you will have to consider moving tags around on the disk block to consolidate the empty spaces. If there is a cumulative 25 bytes free in the block, then you should be able to add a 25-byte tag.

4) Additional feature: **Allow additional disk blocks to be allocated for tags**. If **tagFile** is called and the new tag won't fit on the 512-byte disk block, a new block should be allocated. To achieve this, you'll want to switch from using a direct block to an indirect block. See pages 79-80 of the xv6 textbook for more info on the difference between direct and indirect blocks. Since an indirect block can point to 128 data blocks, this feature will allow up to 64 KB of space for tags per file.

$$128\,blocks\ \times 512\,bytes\ per\ block\ = 65536\,bytes\ =\ 64\,KB$$

That's all. Good luck!

Acknowledgements

This project is based heavily upon a project from Columbia University's operating systems course.