File Project

You may discuss this File Project description with others to understand the problem. You may use the Internet for point-specific questions. For example, you may search the Internet for how to call `dup` and `dup2`. You **may not** use the Internet to discover file system code that is copied or mimicked into your solution. Our OSTEP textbook explains file systems, we have discussed file systems in lectures, and this document provides insight into the problem. Use this material to design and implement your own code.

Use the file `FileProjectAnswers.docx` to answer questions in this project.

Introduction

Please read chapters 39 and 40 of OSTEP prior to beginning this project.

Persistence of the third of our OS - Three Easy Pieces. Persistence is provided as files in a file system. The goal of this project is for you to understand the basics of how an OS implements a file system.

The OS builds a filesystem on top of a block-structured device such as a spinning hard disk drive (HDD) or a solid state drive (SS). HDDs and SSDs have a sequence of blocks numbered from 0 to the last block. Each block is a specific number of bytes. Today's HDD and SSD have 4096 block sizes. Older systems built filesystems on blocks with 512 bytes. The OS provides an abstraction of directories and files to applications. Directories and files are mapped to the blocks on an HDD or SDD. Applications access directories and files using the API provided by the OS. The OS provided API allows applications to perform operations like `open`, `close`, `read`, `write`, and `fstat`.

Tiny File System API

The C file API has existed since Kernigan and Ritchie created Unix and its original file system. Tiny File System is essentially the original Unix file system, but TFS has been created in the context of a Linux C application - not in an OS. TFS supports the standard C file API; however, each of the functions are prefixed with `tfs_`. For example, instead of calling `open`, call `tfs_open`. Of course, this is needed because the TFS code uses the normal C file API to implement TFS. The include file `user.h` contains the prototypes for the `tfs_` file system API. The file `tfsfile.c` contains the code for the `tfs_` functions. Read subsequence sections to learn more on the TFS design and structure.

tfs_open

`int tfs_open(const char *pathname, int flags, mode_t mode)` - function prototype.
`int fd = tfs_open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);` - sample call.
`open` returns a file descriptor, an `int`, that is an index into the array `ofiles` that is a member of `struct proc`.

```
struct proc {
…
   struct file *ofile[NFILE];
…
}
```

As you can see each entry of `ofile` is a `struct file` that contains information about the open file.

In traditional Unix/Linux, when you call open, it returns an int that is greater than 2, because the file descriptors 0, 1, and 2 are automatically opened for a process. File descriptor 0 is for standard input (`stdin`) , file descriptor 1 is for standard output (`stdout`), and file descriptor 2 is for standard error (`stderr`). You can use `stdin`, `stdout`, and `stderr` as parameters to file system functions such as `fgets` - `fgets(buf,100,stdin)` and `fprintf` - `fprintf(stdout,"Hello \n")`, `fprintf(stderr,Error \n")`. TFS does not open `stdin`, `stdout`, and `stderr`. TFS returns file descriptor 0 for the first file opened by a process.

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

Everyone knows the fork creates an exact copy of the current process, including the open files. It is easy to envision copying the contents of one struct proc to another. When this happens the `ref` member of the `struct file` is incremented because now it is opened by two processes. The file is not closed until both processes close it (or when both exit). This is why it is important to close ends of pipes that are not used.

tfs_close

`int tfs_close(int fd)` - closes the file.

tfs_read

`int tfs_read(int fd, char *buffer, int size)` - reads up to `size` bytes from `fd` into `buffer`. Returns the number of bytes read into the `buffer`.
`s = tfs_read(fd, buffer, 19);` - sample call.

tfs_write

`int tfs_write(int fd, char *buffer, int size)` - writes up to `size` bytes from `buffer` to `fd`. Returns the number of bytes written to file.
`s = tfs_write(fd2, "HELLO TO EVERYONE IN the world! Happy New Years!", 48);`

tfs_lseek

`off_t lseek(int fd, off_t offset, int whence);` seeks to a position in a file. Subsequent reads/writes begin at the position.
`whence` values and actions

- `SEEK_SET` - the offset is set to offset bytes.
- `SEEK_CUR` - the offset is set to its current location plus offset bytes.
- `SEEK_END` - the offset is set to the size of the file plus offset bytes.

The current offset is stored in the `uint off` member of the `struct file` that is in the `filedes`.

A nice example use of lseek is in the function bread of bio.c. bread reads a specific 512-byte block from a TDD.

```
int bread(uint block, char *buf) {
    int off = lseek(fs, block*BSIZE, SEEK_SET);
    int sz = read(fs, buf, BSIZE);
    return 0;
}
```

Other tfs_ Functions

TFS supports all of the standard Linux file system function

- `tfs_fstat` - equivalent to `fstat`.
- `tfs_link` - equivalent to `link`.
- `tfs_unlink` - equivalent to `unlink`.
- `tfs_mkdir` - equivalent to `mkdir`.
- `tfs_chdir` - equivalent to `chdir`.

TFS Design Overview

The Tiny File System has been created using source code that is part of XV6. The source code has been modified to create a Linux program that creates a file system in a binary file. This allows students to study file system implementation in the context of user programs. Some of this code has been modified extensively to achieve this goal.

`tfsfile.c` contains all of the API functions for Tiny File System. It has all of the Linux file system functions, but they are prefixed with tfs. For example, `tfs_open`, `tfs_close`, `tfs_write`, `tfs_read`, `tfs_fstat`, `tfs_link`, etc. Thus the following demonstrates writing an application for Tiny File System. The application has three steps - open TFS, do application code, close TFS

Open TFS

```
// Establish Tiny File System for an application (process).
// The OS normally does these next steps; however, TFS is an application.
openfs(FSNAME); // in bio.c - opens FSNAME - see below
readfsinfo();   // in fs.c - reads superblock and inodes for file system
// We need a curr_proc which has ofiles[]
curr_proc = malloc(sizeof(struct proc)); // Files are per process
strcpy(curr_proc->name, "Gusty");
struct inode *ip = iget(T_DIR); // curr_proc needs a current working dir
curr_proc->cwd = ip;
```

Do Application Code

```
// These next steps are typical application code using TFS
int fd = tfs_open("HELLOWORLD", TO_CREATE | TO_RDWR, 0);
s = tfs_write(fd, "HELLO TO EVERYONE IN the world! Happy New Years!", 48);
s = tfs_write(fd, "Writing data to another file. 123456789abcdefgh!", 48);
tfs_close(fd);
```

Close TFS

```
writefsinfo(); // write memory resident file information to TDD
closefs(); // close the file, which is the TDD
```

TFS TDD Layout

`fs.h` contains the several defines, which provide top-level attributes of Tiny File System, which is implemented on a Tiny Disk Drive (TDD), not to be confused with Elton John's Tiny Dancer.

```
#define ROOTINO 1        // root inode number
#define BSIZE 512        // block size
#define FSNAME "tinyfs"  // File system name
#define NBLOCKS 1024     // number of blocks in file system
```

The file `tinyfs` is our TDD. It is a binary file that consists of a sequence of 1024 blocks where each block is 512 byte. The Tiny File System exists on these 512 byte blocks. `bio.c` contains functions that read and write 512 byte blocks by a block number.

`int bread(uint block, char *buf)` - reads block number block into buf
`int bwrite(uint block, char *buf)` - writes buf to block number block

The file `tinyfs` and the functions `bread` / `bwrite` serve as a simulation of a block-based device such as a hard disk drive (HDD) or a solid state drive (SSD). We call our disk drive a Tiny Disk Drive (TDD).

Changing `FSNAME` provides you with a different name for the TDD, and changing `NBLOCKS` provides you with more blocks on your TDD. These changes should work. Changing `BSIZE` requires studying all of the Tiny File System source code to make sure the data structures properly map to the new `BSIZE`.

On-disk file system format currently implemented for Tiny File System.

- Block 0 is unused.
- Block 1 is the super block.
- Block 2 inode bitmap - not used, inodes are free if `type == 0`, See function `ialloc` in file `fs.c`.
- Block 3 data block bitmap - used to find free data blocks for files. See function `balloc` in file `fs.c`.
- Blocks 4 through 7 hold inodes - The current implementation has an inode that is 64 bytes. The following code is in `fs.h`. You can add the sizes on each member of the struct. TFS allocates four 512-byte blocks for inodes. With 64 byte inodes, we can place 8 inodes on each block. Thus TFS accommodates a total of 32 files. Adding more attributes to an inode, requires thinking about the layout of TFS.

```
#define NDIRECT 8 // Files can have a total of 8 direct blocks
                  // TFS does not implement indirect blocks
struct inode {
```

```
  uint type;     // File type - dir, file
  uint nlink;    // Number of links to inode in file system
  uint size;     // Size of file (bytes)
  uint ref;      // references to inode HERE
  uint inum;     // inode number
  //uint uid;      // user id
  //uint gid;      // group id
  uint ctime;    // creation time
  uint mtime;    // modified time
  uint blocks[NDIRECT+1];   // Data block addresses
};
```

Determining the size of an `struct inode`

- A `uint` is 4 bytes.
- A `struct inode` has 7 members that are type `uint` - 28 bytes
- A `struct inode` has a `uint blocks[]` that has 9 elements - 36 bytes
- A `struct inode` is 64 bytes

There are several macros manipulating attributes of inodes.

```
// Inodes per block.
#define IPB           (BSIZE / sizeof(struct inode))
// Block containing inode i
#define IBLOCK(i)     ((i) / IPB + 2)
// Bitmap bits per block
#define BPB           (BSIZE*8)
// Block containing bit for block b
#define BBLOCK(b, ninodes) (b/BPB + (ninodes)/IPB + 3)
```

- Blocks 8 to `sb.nblocks` are data blocks, which are used to hold data of files. We have 1024 blocks on our TDD. TFS supports 32 files (see previous bullet). TFS files only support 8 direct blocks - these are pointed to by the `blocks[]` member of a struct inode (see previous bullet). This means the current implementation will at maximum file capacity only consume 256 data blocks (32 files * 8 blocks per file). The variable `sb` (at start of this bullet) is type `struct superblock`, which is defined in `fs.h`. The `createfs` function in `bio.c` fills in the superblock. See discussion below about `createfs`.

```
struct superblock {
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks - not used in tinyfs
  char name[12];     // name of file system
};
```

- TFS files are small. They can hold 4096 bytes (8 blocks * 512 bytes per block).
- TFS does not implement NINDIRECT. Indirect blocks allow a file to expand, where the last direct block points to an indirect block, which is a block devoted to more pointers. The 9th element of `uint blocks[]` in `struct inode` is for indirect blocks.

- TFS has one inode structure - discussed above. Xv6 has an ondisk inode structure and an in-memory inode. The in-memory inodes are a cache of the ondisk inodes. This allows for faster processing. All of the TFS inodes are in memory. Xv6 in-memory inodes have a `ref` member that counts the number of files referring to the in-memory inode. TFS retains the `ref` member in order for the Xv6 code to work. The function `iget` in `fs.c` expects `ref` to be non-zero to properly find an inode structure for an inode number, When TFS reads the initial inodes from disk, TFS sets `ref` to be equal to `nlink`. This allows `iget` to work properly

- A directory is a file containing a sequence of `dirent structures`, defined in fs.h. A dirent structure maps a name to an inode number. The function `iget` maps the inode number to its inode structure. TFS filenames are a maximum of 14 characters, and an `inum` is a `ushort`, which makes `struct dirent` 16 bytes - a nice number. Powers of two are always nice numbers. A directory that is allocated one 512 byte block can hold 32 files. This means a single block directory can hold all of the TFS files.

```
#define DIRSIZ 14
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

TFS Organization and Building

TFS consists of the following .c and .h files. You have used some of the files in previous labs and projects. Three pairs of files define three layers in the file system. These are discussed first.

- `tfsfile.c`, `user.h` - contain API functions and prototypes. User programs call `tfs_open` etc. These functions work with file descriptors. They map file descriptors to its `struct file`, and call the `file.c` layer.
- `file.c`, `file.h` - the layer below `tfsfile.c`. For example `tfs_read` (in `tfsfile.c`) calls `fileread` (in file.c). These functions work with `struct file`., which is defined in `file.h`. They find the `struct file`'s corresponding `struct inode`, and call the `fs.c` layer.
- `fs.c`, `fs.h` - the layer below `file.c`. For example `fileread` (in `file.c`) calls `readi` (in `fs.c`). These functions work with `struct inode`, which is defined in `fs.h`.

The remaining files are the following.

- `bio.c` - contains main and the block I/O that implements a TDD.
- `defs.h` - contains function prototypes of various functions.
- `fcntl.h` - contains definitions of TO_RDONLY, TO_WRONLY, TO_RDWR, and TO_CREATE.

- `param.h` - defines key constants of the OS, including file system constants. For example, the number of processes (`NPROC`), number of open files per process (`NFILES`), and number of inodes (`NINODE`).
- `proc.h` - defines process structures.
- `stat.h` - defines structure of information returned from `tfs_fstat`.
- `types.h` - defines common types like `uint` and `ushort`.

The makefile, program, and TDD file are the following.

- makefile - very basic makefile that creates the program tiny.
- tiny - the program created by invoking make.
- tinyfs - the TDD file. Tiny File System is placed in this file.

Running TFS

The main function is in `bio.c`. It has three paths that are selected by the argument passed to tiny.

1. `$ ./tiny create` - this step create a TFS on a TDD. After this step, the file tinyfs, which is our TDD, is formatted as a TFS. At this point, the TDD contains a superblock, inode bitmap, data block bitmap, and one inode for the root directory. This code currently writes the data block number on each data block. You only have to do this step one time. Then you can perform user operations (tfs_read, etc.). You also do this step when you want to reset TFS on a TDD. The following shows some of the blocks on TDD after this step. The highlights of the blocks shown are the following.
    a. Block 0 is not used by TFS
    b. Block 1 is the superblock. Not much in a superblock, which is shown above. See fs.h for format of struct superblock.
    c. Block 4 is the first block with inodes. The create command places the root directory (i.e. /) in inode 1. The format of an inode is shown above. Also see fs.h for format of struct inode. The inode points to the data block of the file. In this case the root directory is on block 8, the first data block. Although the 8 does not appear until after writing the first data file.

```
block: 00000:
0x00000000  426c6f63 6b203020 2d204e6f 74207573 65642e00 00000000 00000000 00000000   Block 0 - Not used.
0x00000020  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
block: 00001:
0x00000200  00040000 f8030000 20000000 00000000 74696e79 66730000 00000000 00000000                  tinyfs
0x00000220  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
block: 00004:
0x00000800  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000820  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000840  01000000 00000000 00000000 ab000000 01000000 06bc805e 00000000 00000000
0x00000860  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

2. `$ ./tiny write` - this step is a user program that writes some data to a few files. Examining the code shows this does the three steps described above - open TFS, do application code, and close TFS. Currently, the code writes small amounts of data to

three files - GUSTY, HELLOWORLD, and Another. The following shows some of the blocks on TDD after this step. The highlights of the block shown are the following.

a. Block 4 is the first block with inodes. We now have four inodes. One for the root directory (on block 8), one for GUSTY (on block 9), one for HELLOWORLD (on block 10), and one for Another (on block 11).
b. Block 8 is the data block containing the root directory. A directory is an array that maps a filename to its inode number. Root has three files - `GUSTY`, `HELLOWORLD`, and Another. The file `GUSTY` is mapped to inode 2. Looking at inode 2, you see its data is on block 9.
c. Block 9 is the data block containing file `GUSTY`'s data. The code in `main` wrote `COOPER123`, which you can see in block 9.
d. Block 10 is the data block containing file `HELLOWORLD`'s data. The code in `main` wrote the data you can see in block 10.
e. Block 11 is the data block containing file `Another`'s data. The code in `main` wrote the data you can see in block 11.

```
block: 00004:
0x00000800  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000820  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000840  01000000 00000000 30000000 02000000 01000000 06bc805e 00000000 08000000           0              ^
0x00000860  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000880  02000000 01000000 09000000 01000000 02000000 10c0805e 00000000 09000000                          ^
0x000008a0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x000008c0  02000000 01000000 60000000 01000000 03000000 10c0805e 00000000 0a000000           `              ^
0x000008e0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x00000900  02000000 01000000 30000000 02000000 04000000 10c0805e 00000000 0b000000           0              ^
0x00000920  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

block: 00008:
0x00001000  02004755 53545900 00000000 00000000 03004845 4c4c4f57 4f524c44 00000000     GUSTY           HELLOWORLD
0x00001020  0400416e 6f746865 72000000 00000000 00000000 00000000 00000000 00000000     Another

block: 00009:
0x00001200  434f4f50 45523132 33000000 00000000 00000000 00000000 00000000 00000000   COOPER123
0x00001220  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

block: 00010:
0x00001400  48454c4c 4f20544f 20455645 52594f4e 4520494e 20746865 20776f72 6c642120   HELLO TO EVERYONE IN the world!
0x00001420  48617070 79204e65 77205965 61727321 57726974 696e6720 64617461 20746f20   Happy New Years!Writing data to
0x00001440  616e6f74 68657220 66696c65 2e203132 33343536 37383961 62636465 66676821   another file. 123456789abcdefgh!

block: 00011:
0x00001600  57726974 696e6720 64617461 20746f20 616e6f74 68657220 66696c65 2e203132   Writing data to another file. 12
0x00001620  33343536 37383961 62636465 66676821 00000000 00000000 00000000 00000000   3456789abcdefgh!
```

NOTE: Known Missing Data - Directories normally contain two entries - `.` and `..`. You can notice that Block 8 - the root directory - does not have `.` and `..`. I have attempted to add `.` and `..` to root. I assume that .. in a root directory points to itself.

3. `$ ./tiny read` - this step is a user program that reads data from the file Another, which was written in step 2. The application read code is the following.

```
char buffer[512];
strcpy(buffer, "sometext");
int fd3 = tfs_open("Another", TO_RDONLY, 0);
printf("fd3: %d\n", fd3);
```

```
s = tfs_read(fd3, buffer, 19);
printf("tfs_read bytes: fd: %d, bytes read: %d value read: %s\n", fd3,
s, buffer);
```

The output shows it reads 19 bytes from `Another`. These 19 bytes are on Block 11.

```
manipulate fs file with reads.
fs : 3
fd3: 0
tfs_read bytes: fd: 0, bytes read: 19 value read: Writing data to ano
```

Utilities - hexdump

In order to complete File Project, you must be able to examine the contents of the binary TDD file, `tinyfs`. I have a hex editor on my Mac's - `hexfiend`. I suspect everyone can find a hex editor, but just in case you cannot, I created a `hexdump.c` utility program. `hexdump` reads a file in 512 byte chunks and displays the block number, followed by a sequence of 16 rows. Each row has the byte address of the row, 8 32-bit values in hex, and the corresponding printable characters (if they are printable. The blocks shown in the previous section are created by `hexdump`. `hexdump` was created for dumping `tinyfs`, but it can be used on any file.

You should not put hexdump in the same folder with Tiny File System source code. Hexdump.c has a main, and the TFS makefile does not work with two .c files with main. Instead put hexdump.c in its own folder. I have a sub-folder - `utilities`. To build `hexdump`, do the following.

```
$ gcc -o hexdump hexdump.c
```

The user's guide for hexdump is the following

`$ ./hexdump tinyfs` - dumps blocks 0 through 9
`$ ./hexdump -s15 -l5` - dumps blocks 15 through 19

The flags `-s` and `-l` specify a start block and a length, which is the number of blocks to dump.

The following shows hexdump output from block 8, which is the data block of the root directory. Blocks 0 through 7 contain 8 * 512 bytes, which is 4096 bytes - these bytes are addresses 0 through 4095. Thus the first address of a byte in block 8 is 4096, which is 0x1000 in hex. This is what you see immediately under block. The eight hex numbers to the right of 0x00001000 are the first eight 32-bit quantities in block 8. The ASCII to the right of the eight hex numbers is any printable character in the preceding hex numbers. For example, `02004755 53545900` contain the printable characters `GUSTY`.

```
block: 00008:
0x00001000  02004755 53545900 00000000 00000000 03004845 4c4c4f57 4f524c44 00000000     GUSTY           HELLOWORLD
0x00001020  0400416e 6f746865 72000000 00000000 00000000 00000000 00000000 00000000     Another
```

Assignment

Be sure your submission uses the `FileProjectAnswers.docx` to answer the questions so I can easily decipher your answers in your submission.

The assignment has several steps. The Basic Submission steps can earn 88 points. The Advanced Submission steps earn additional points depending upon your testing, which can be more than 12.

Basic Submission (earns a maximum of 88 points)

`bio.c` has several test cases. You will build and run TFS to understand the file system and how it is tested. You will use the hexdump tool on `tinyfs` to further your understanding.

1. Build and run `tiny`, using all three of the arguments - `create`, `write`, `read`. You can read above and study `main` in `bio.c` for the current content of these three arguments. Copy/paste your run log here.
2. Update `main` in `bio.c` to write data to a file you create. Also update `main` to read data from your file. For both of these updates, I suggest you retain small sizes for writes and reads for you initial update. Copy/paste your run log here.
3. Use `hexdump` to dump the file `tinyfs`, which is the TFS TDD to verify that your files are on the TDD. You must locate several aspects of them. Their names, inode numbers, and data block(s) in the root directory. Copy/paste your use of hexdump here. Annotate your copy/paste to demonstrate understanding.
4. Create an annotated diagram of TFS. Show TFS structures and how they interconnect. Include the superblock, the inode bitmap, the data bitmap, the inodes, and two files - one of which is what you added to main in step 2. Also show file descriptors, struct proc, struct file, struct inode and how a file descriptors are converted to inodes. When creating this diagram, use the hex dump from step 3 to demonstrate your understanding. Place your annotated diagram here.
5. Create a function call trace starting with `tfs_read`. Create this trace by reading the code. For each function, describe the parameters and a detailed description of what the function does. For example,
    a. `tfs_read`'s (in file `tfsfile.c`) parameters are a file descriptor, a buffer, and a size to read. `tfs_read` calls `fd_to_file` to convert the file descriptor to a `struct file`. Then `tfs_read` calls `fileread` to continue reading the file.
    b. `fd_to_file` (in file `tfsfile.c`) parameters are an `fd` and a `struct file` **. `fd_to_file` ensures the fd is within range and returns the pointer to the `struct file` in the proc's `ofile[]`.
    c. `fileread` (in `file.c`) parameters are … continue

    Place your function call trace here.

6. Update `hexdump` to include a new flag. You can select the flag. For example, you could add a `-i` flag that dumps inodes, or you can add a `-e` flag that has the end block. Copy/paste a run log of `hexdump` applied to `tinyfs` with your new flag.
7. Submit your updated `bio.c` and `hexdump.c` files.

Advanced Submission (earn 2 points per test)

Update `main` in `bio.c` to test additional `tfs_` functions. You will first have to investigate the Linux equivalent to discover what it is supposed to do. Then you can try the `tfs_` version to see if it works. When something does not work, you will be forced into deep study of the underlying code base, which will sharpen your understanding of file system implementation. You can update bio.c with explicit test calls similar to the provided tests. A beautiful update of bio.c would include a shell-like interface that allows you to open, close, link, mkdir, etc. Select from (or do all) the following tests.

8. Create a `tinyfs` file with over 512 bytes of data. This will cause a second block to be allocated. See that the inode indicates two data blocks. Copy/paste a hexdump of `tinyfs` showing your large file.
9. Repeat step 6a, but this time write to the file in steps. First write data that fits on a single data block. Then create another file with data. Then write more data to the first file to cause it to grow to two data blocks. This will show a file with two non consecutive data blocks. Copy/paste a hexdump of `tinyfs` showing your large file with data blocks separated.
10. `tfs_fstat` - equivalent to `fstat`. Copy/paste a log showing your `tfs_fstat` test.
11. `tfs_link` - equivalent to `link`. Copy/paste a log showing your `tfs_link` test.
12. `tfs_unlink` - equivalent to `unlink`. Copy/paste a log showing your `tfs_unlink` test.
13. `tfs_mkdir` - equivalent to `mkdir`. Copy/paste a log showing your `tfs_mkdir` test.
14. `tfs_chdir` - equivalent to `chdir`. Copy/paste a log showing your `tfs_chdir` test.
15. `tfs_lseek` - equivalent to `lseek`. Copy/paste a log showing your `tfs_lseek` test.
16. Submit your updated `bio.c` file that includes your test cases.