P5: Filesystms
BFS - a Unix-Like FileSystem

This document describes how to implement a Unix-like filesystem. We call this filesystem “BFS”, for “Bothell File System”.

The filesystem comprises 3 layers. From top to bottom, these are:

- **fs** – user-level filesystem, with functions like fsOpen, fsRead, fsSeek, fsClose.
- **bfs** – functions internal to BFS such as bfsFindFreeBlock, bfsInitFreeList.
- **bio** - lowest level block IO functions: bioRead and bioWrite.

It is easiest to describe these 3 layers in the order, **bio**, **fs** and **bfs**, as follows:

bio (Block IO level)

A raw, unformatted BFS disk consists of a sequence of 100 blocks, each 512 bytes in size. The BFS disk is implemented on the host filesystem as a file called BFSDISK. Blocks are numbered from 0 up to 99. Block 0 starts at byte offset 0 in BFSDISK. The next block starts at byte offset 512, and so on. We can read an entire block from disk into memory. And we can write an entire block from memory onto a disk block. It’s perfectly legal to update an existing disk block by over-writing its contents.

It is not possible to read only part of a block (for example, its middle 100 bytes) from disk. It’s all or nothing. And the same for writing to a disk block: exactly 512 bytes, no more, no less.

The **bio** (for “Block Input Output”) interface lets us read and write whole blocks of data on disk, as follows:

- i32 bioRead(i32 dbn, void* buf) – read block number dbn from the disk into the memory array buf. Return 0 for success. Any failure will abort the program.
- i32 bioWrite(i32 dbn, void* buf) – write the contents of the memory array buf into block number dbn on disk. Return 0 for success. Any failure will abort the program.

i32 is a typedef for 32-bit signed int. See alias.h for details.

If the program encounters a fatal error, it prints out the file and line number where the error arose, together with a description of the error. See error.h and error.c for details.

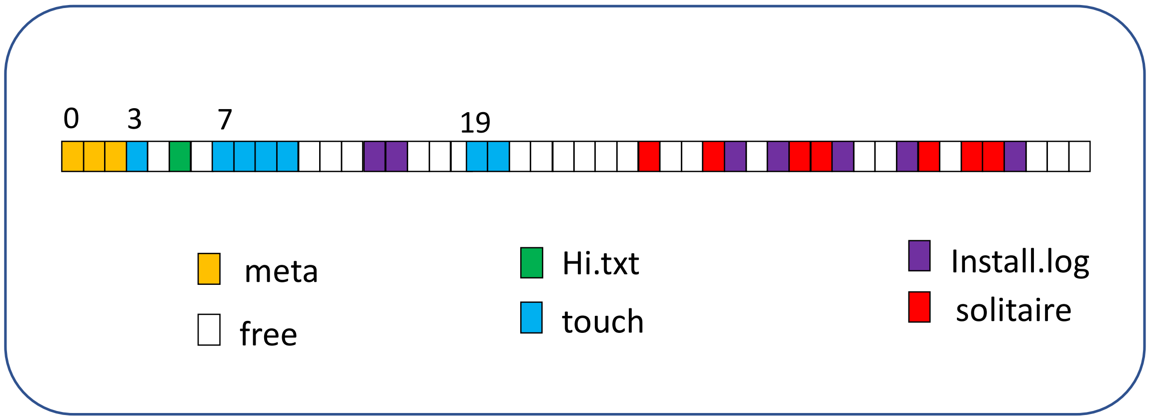
In a real filesystem, bioRead and bioWrite would be part of the Kernel, and not callable directly from user code.

fs (User-level fileystem)

An interface of *raw,* fixed-size blocks is too primitive for regular use. BFS presents a more user-friendly interface: one that lets us create files, where “file” is a collection of zero or more bytes on disk, reached by a user-supplied name. The bytes that comprise the file may be scattered over the disk's blocks, rather than having to occupy a contiguous sequence of block numbers. Files can be grown automatically as a user writes more and more data into that file

The diagram below shows a sequence of disk blocks, containing 4 files, called Hi.txt, touch, Install.log and solitaire. Hi.txt is small, and uses up just one disk block (in green). touch is larger: it uses 7 blocks, spread across the disk. Similarly for the other two files. The uncolored blocks are free – available to be allocated to future files; or to existing files, should they be extended.

BFS keeps track of those file names, and the blocks their files occupy. It also remembers where all the free blocks are on the disk.



The filesystem, or **fs**, interface comprises the following functions. Notice that BFS lets us read or write an arbitrary number of bytes (not restricted by block size), at any arbitrary byte-offset within the file.

We call the numbers used to describe blocks within a file as File Block Numbers, or FBNs. We call the numbers used to describe blocks within the BFS disk as Disk Block Numbers, or DBNs. Both FBNs and DBNs start at 0.

For example, the file called touch in the above diagram, occupies 7 blocks. So its FBNs run from 0 thru 6. But each one is stored in a DBN somewhere in the BFS disk: FBN 0 is stored in DBN 3; FBN 1 is stored in DBN 8; and so on (see the later section on Inode layout for a full explanation)

i32 fsFormat() – initialize the BFS disk. On success, return 0. On failure, abort the program. Note that this function will create a new BFSDISK. It will destroy any previous BFSDISK.

i32 fsMount() – mount an existing BFS disk. On succes, return 0. On failure, abort the program.

i32 fsOpen(str fname) – open the file called fname. On success, return a filedescriptor, or fd, which can be used to perform subsequent operations on that file. On failure, return EFNF (File Not Found). The byte-offset within the file – the position at which any subsequent fsRead or fsWrite operates – is initialized to 0. (Note that str is a typedef for char*. See alias.h for details).

i32 fsRead(i32 fd, i32 numb, void* buf) – read numb bytes from the current cursor in the file currently open on fil descriptor fd. Place the bytes into the memory buffer buf. On success, return number of bytes read (this may be less than numb if the read operation hits end-of-file). On failure, abort the program.

u32 fsWrite(i32 fd, i32 numb, void* buf) – write numb bytes from the memory buffer buf into the file currently open on filedescriptor fd, starting at the file's current cursor. Note that a write to any bytes beyond the current length of the file will automatically extend the file. On success, return 0. On failure, abort the program.

i32 fsSeek(i32 fd, i32 offset, i32 whence) – adjust the cursor to offset. whence specifies how offset is measured, with values, defined in stdio.h. SEEK_SET (0) means seek to byte-offset offset from the start of the file; SEEK_CUR (1) means add offset to the current byte-offset; SEEK_END (2) means add offset from the end of file. On success, return 0. On failure, abort the program.

i32 fsClose (i32 fd) – close the file currently open of filedescriptor fd. On success, return 0. On failure, abort the program.

i32 fsSize(i32 fd) – return the size, in bytes, of the file currently open on filedescriptor fd. On failure, abort the program.

i32 fsTell(i32 fd) – return the current cursor. On failure, abort the program.

bfs (raw IO level)

BFS uses certain blocks – which we will call metablocks – at the start of the BFS disk, to remember where all its files are stored, and to keep track of all free blocks. These metablocks are not visible to the programmer via the **fs** interface.

(As explained below, there are 3 metablocks: SuperBlock, Inodes and Directory, occupying the DBNs 0, 1 and 2 of the BFS disk)

The BFS (BFSDISK) disk is 100 blocks long. It can hold, at most, just 8 files. As will be seen later, the biggest file supported by BFS is 5+256 blocks. So a full disk would require 8 * (5 + 256) = 2,088 blocks. This is larger that the BFS disk size of 100 block (on purpose); so BFS needs to check against "disk full".

The metablocks are used to hold 3 kinds of structure, as follows:

SuperBlock

Always DBN 0 in BFSDISK. Holds 4 numbers that characterize the disk:

- numBlocks: total number of blocks in BFSDISK: always 100
- numInodes: total number of inodes in BFSDISK: always 8
- numFree: total number of free block in BFSDISK
- firstFree: DBN of the first free block in BFSDISK

The freelist is kept as a linked list: the first u16 cell in each free block holds the block number of the next free block. (u16 is a typedef for a 16-bit unsigned integer. See alias.h for details)

Directory

The Directory is a simple mapping from filename to Inode number (see later). Each filename can be up to 15 characters long (plus a trailing NUL).

Here is an example Directory, as it appears on disk. NULs ('\0') are denoted by the single glyph □.

fname
Hi.txt□
touch□
□
solitaire
□
□
□
Install.log

The Directory holds 8 names. The index of the name within the Directory, starting at 0, tells us the file's Inode number – hereafter called its Inum. For example, the file called "Install.log" has Inum 7. Unused entries in the Directory (not yet used, or used for a file that was subsequently deleted), are denoted by an empty file name.

The Directory fits within its own single metablock on disk.

Inode

There is one inode for each file. It holds the file size, plus a table of the DBNs that hold the file's data. Here is an example Inode for the file called touch above. First, notice from the diagram, that touch occupies 7 blocks with DBNs 3, 8, 7, 20, 10 and two more, specified within the indirect block. The last block holds only 100 bytes. So total file size of the touch file is (6 * 512 + 100) = 3172 Bytes.

The diagram below shows the Inode for file touch (the Inode is 16 bytes long):

Field Type	Field	Value
i32	size	3172
i16	direct[0]	3
i16	direct[1]	8
i16	direct[2]	7
i16	direct[3]	20
i16	direct[4]	10
i16	indirect	22

The direct array provides a mapping between FBN and corresponding DBN, for the first 5 FBNs of the file. So: DBNs 3, 8, 7, 20 and 10 in that order.

The indirect field holds the DBN of the "indirect block". This is a single block that holds 256 i16 values, that record the next 256 DBNs for the file. The indirect block uses the same convention as the Inode –

entries with value 0 mean there is no corresponding disk block. The following diagram shows the first few i16 entries in the indirect block at DBN 22:

Indirect Block at DBN 22
9
19
0
0
0
...

So now we see that the last two blocks for touch are stored in DBNs 9 and 19. In total, file touch occupies DBNs 3, 8, 7, 20, 10 , 9 and 19, in that order.

With the above explanations, we can see that a single inode can map a file of up to 5 FBNs. And the indirect block can map a further 256 FBNs. So the largest file that BFS supports is (5 + 256) = 261 blocks.

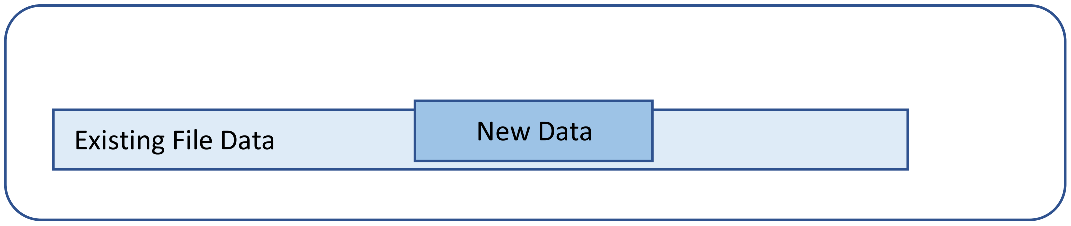
bfs Interface

The BFS interface provides all the functions required to create and maintain the filesystem. These functions are called by the **fs** interface, or by other functions within the **bfs** layer itself. There are 20+ functions in the **bfs** API. See files bfs.h and bfs.c for details.

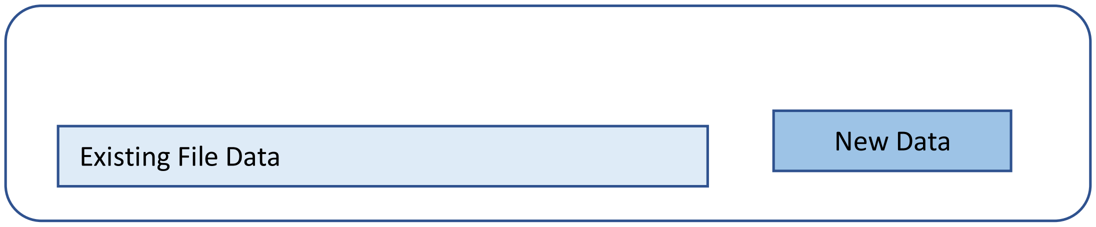
Implementation Notes

A write to a file falls into one of three cases that we describe as *inside*, *outside* and *overlap*.

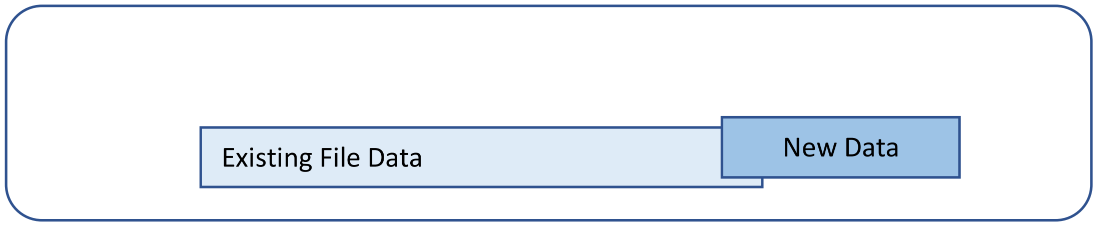
Inside means that the write fits within the existing file. There is no need to extend the existing file:



Outside means the write fits beyond the range of the existing file. This requires that new blocks are added to the file. Any region not explicitly written – the gap between existing and new – must be set to zeroes.



Overlap means the write fits covers part of the existing file, but extends beyond its current limit. This requires that new blocks are added to the file.



Notice that many **bfs** functions are written so they either succeed (return a status value of 0), or they fail – in which case, they abort the program. This behavior is by-design. It's because almost all failures in a **bfs** function reflect a bug somewhere within **bfs** itself. Having detected a fatal bug, it's unsafe to continue, since subsequent operations may make the situation worse, and/or corrupt the on-disk data. As a consequence, you can often call a **bfs** function, and not have to check whether it worked successfully: if it returned, it was successful; if it did not return, the program stopped.

Project P5

This section describes what is required in Project P5 of the CSS-430 "Operating Systems" class.

Download all the source files from Canvas at Files|Projects|P5-Download. This includes the following:

- alias.h : a short set of typedefs for u8, i16, str, etc
- bfs.h, bfs.c : support for Super block, inode, directory and Open-File-Table
- bio.h, bio.c : low-level block IO functions
- errors.h, errors.c : pretty error messages
- fs.h, fs.c : user-level filesystem API. fs.c is incomplete. Your task is to implement the two missing functions fsRead and fsWrite.
- p5test.h, p5test.c : the test program used to test your implementation of the missing functions
- main.c : the driver function

The download includes a BFSDISK that contains one file, called simply "P5". It holds 50 blocks, filled with a preset pattern: every byte of block 0 holds the value 0. Every byte in block 1 holds the value 1. And so on.

As noted above, the assignment is to implement the functions fsRead and fsWrite. You should call your program **p5** (on Linux).

Grading will consist of building your code, then running it – which will execute the 6 tests detailed in p5test.c. This includes checks for correct answers. Here are how marks will be awarded:

1. Read 100 bytes from start of file
2. Read 200 bytes, from FBN 1, at cursor = 30
3. Read 1,000 bytes from start of DBN 20 (spanning read)

4. Write 77 bytes, starting at 10 bytes into FBN 7
5. Write 900 bytes, starting at 50 bytes into DBN 10
6. Write 700 bytes, starting at FBN 49. (This write extends the file)

Note that the 6 tests do not exercise all variations. For example, no writes of an initially empty file; no *outside* writes.

A correct run of the program will produce the following output:

```
TEST 1 : GOOD
TEST 1 : GOOD
TEST 2 : GOOD
TEST 2 : GOOD
TEST 2 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
TEST 4 : GOOD
TEST 4 : GOOD
TEST 4 : GOOD
TEST 4 : GOOD
TEST 4 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 5 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
TEST 6 : GOOD
```

Here is an example of a run which held some bug in test 1:

```
TEST 1 : GOOD
TEST 1 : BAD  : buf[0] = 0 but should be 3
TEST 2 : GOOD
TEST 2 : GOOD
TEST 2 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
TEST 3 : GOOD
. . .
```

Hints

Start by downloading the files from Canvas.

Build those source files - *.c and *.h.   It will include several warnings about variables not used – these will disappear once you code fsRead and fsWrite.

Try running the project. It will fail with the following error message:

```
TEST 1 : GOOD

ERROR: File C:\Users\Jim\SkyDrive\UW\CSS-430-Hogg-Sp20\Projects\P5-Download\P5\fs.c, Line 91

ERROR: Function Note Yet Implemented

Hit any key to finish
```

As you can see, it reports that the function (fsRead in this case) is Not Yet Implemented.

In the downloaded file fs.c, functions fsRead and fsWrite are defined as follows:

```
// ============================================================================
// Read 'numb' bytes of data from the cursor in the file currently fsOpen'd on
// File Descriptor 'fd' into 'buf'.  On success, return actual number of bytes
// read (may be less than 'numb' if we hit EOF).  On failure, abort
// ============================================================================
i32 fsRead(i32 fd, i32 numb, void* buf) {

  // ++++++++++++++++++++++++
  // Insert your code here
  // ++++++++++++++++++++++++

  FATAL(ENYI);                                  // Not Yet Implemented!
  return 0;
}
```

```
// ============================================================================
// Write 'numb' bytes of data from 'buf' into the file currently fsOpen'd on
// filedescriptor 'fd'.  The write starts at the current file offset for the
// destination file.  On success, return 0.  On failure, abort
// ============================================================================
i32 fsWrite(i32 fd, i32 numb, void* buf) {

  // ++++++++++++++++++++++++
  // Insert your code here
  // ++++++++++++++++++++++++

  FATAL(ENYI);                                  // Not Yet Implemented!
  return 0;
}
```

Start with implementing fsRead. To do so, look back at slides 31-33 in Lecture-16-FS-2.pptx.

When it comes to fsWrite, you might *damage* file P5 within BFSDISK, due to bugs in your program. Indeed, you might even corrupt BFSDISK itself. So, either download a fresh copy of BFSDISK from the P5-Download folder. Or keep a copy on you PC, or Linux directory.

What to submit?

- All of the .c and .h files for your project.
- Output showing your program running correctly on p6
- Two-page design document clearly showing the key parts of the filesystem