Program Assignment 2

COMP310/ECSE 427, Fall 2013

Introduction

A file system (FS) is one of the most important components of an operating system. There is a wide variety of file system implementations in use, from traditionally disk-based ones such as NTFS, FAT32, ext2, ext3 to network-based and distributed ones (e.g. Google File System). In this assignment, you are expected to implement a Simple File System (SFS). This assignment introduces basic ideas about a file system such as disk layout, file descriptors, directories, inodes and how to implement basic file operations such as seeking, reading, writing, etc. SFS is simplified from real file systems and contains the following feature:

- Max length for file and directory names
- Single-level directory
- Opened files are both readable and writable

Specifications

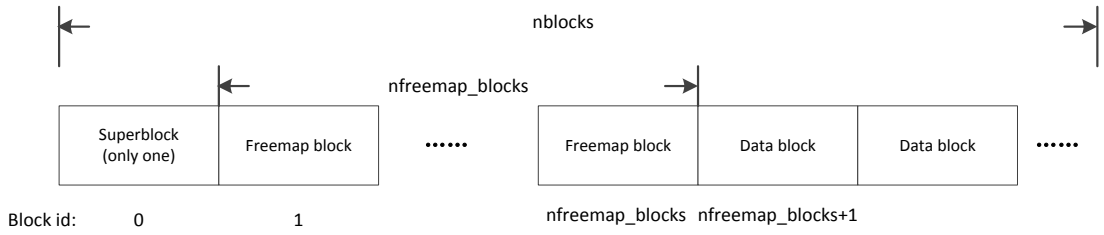
Interfaces

int mksfs();	Initialize the file system layout on the disk
void sfs_print_info();	Print the information about the SFS
int sfs_mkdir(char *dirname);	Create a directory
int sfs_rmdir(char *dirname);	Delete a directory (and all its containing files)
int sfs_lsdir();	List all directories
int sfs_open(char *name);	Open a file. Create one if not existed
int sfs_close(int fd);	Close the file
int sfs_ls();	List all the files in the file system
int sfs_write(int fd, char *buf, int len);	Write to the file
int sfs_read(int fd, char *buf, int len);	Read from the file
int sfs_remove(char *name);	Delete a file
int sfs_eof(int fd);	Check if we reach EOF (End-Of-File)

You may find these functions familiar as they provide similar headers as those found in the standard C library (fopen, fclose, fwrite, fread, feof, etc). Also you can find more detailed description of the functionality of these interfaces in “fs.c”.

Layout

SFS operates on a block disk. Each block is numbered by block id (4bytes, of type u32), starting from zero. The following graph depicts how SFS uses each block.

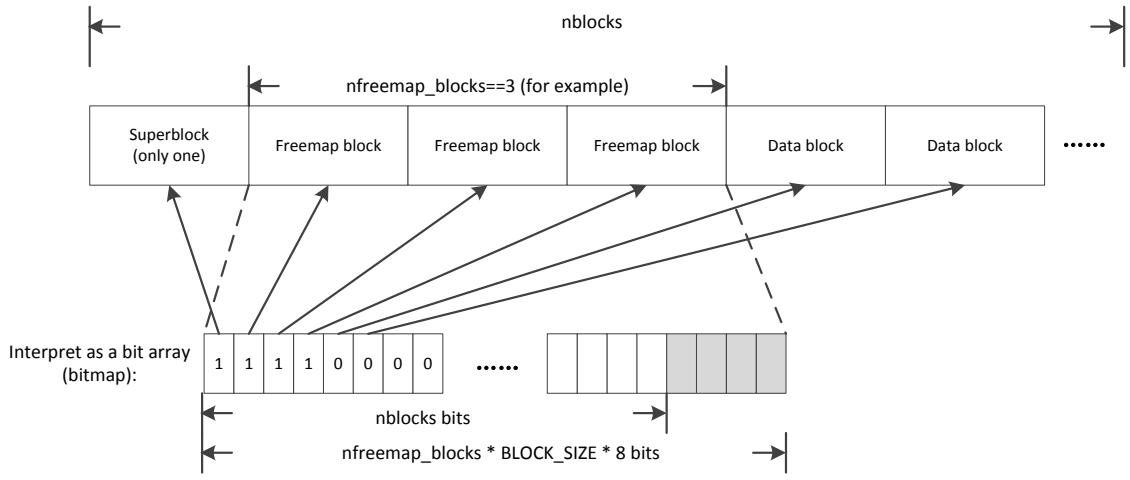


- A **super block** is the first block (with block_id=0). It contains all the basic information about SFS, including how following blocks are organized.
- **Freemap blocks** store the bitmap identifying which block on the disk is usable to SFS.
- **Data blocks** can be either an *inode* that describes a file, a *directory block* that describes a directory, a *file content block* that stores a fraction of file content, or a *frame block* that indexes content blocks.

The (only) super block gives important parameters about the SFS. *nblocks* tells the total number of blocks that can be used by SFS. *nfreemap_blocks* tells how many blocks are used as freemap blocks. For a valid SFS layout, the number of data blocks = nblocks – nfreemap_blocks – 1(superblock).

Freemap (bitmap) and block management

The SFS should know which block is used so that when the user creates a new file, SFS can allocate free blocks to accommodate its data. Freemap is the mechanism for this. This graph shows the case of three freemap blocks:

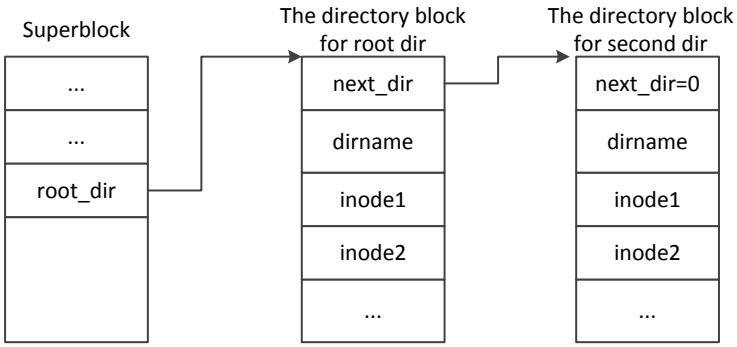


The freemap blocks can be interpreted as a bit array, where each bit indicates if its corresponding block id is used. For example, the first bit in the freemap is always 1, indicating that the block with block_id==0 (superblock) is used. In the case we have three freemap blocks, the second, third and fourth bits are also set to 1, indicating those freemap blocks themselves are used. In such sense, the freemap is self-describing. More details can be found here: http://en.wikipedia.org/wiki/Free_space_bitmap.

Note: If we use one bit for each block, we need nblocks bits to manage all disk blocks. In such case, the freemap should be large enough to hold nblocks bits, i.e., nfreemap_blocks * BLOCK_SIZE * 8 >= nblocks (each byte has 8 bits).

Directory

A directory block is a kind of data blocks that describes a directory, which contains the block ids of its containing files.



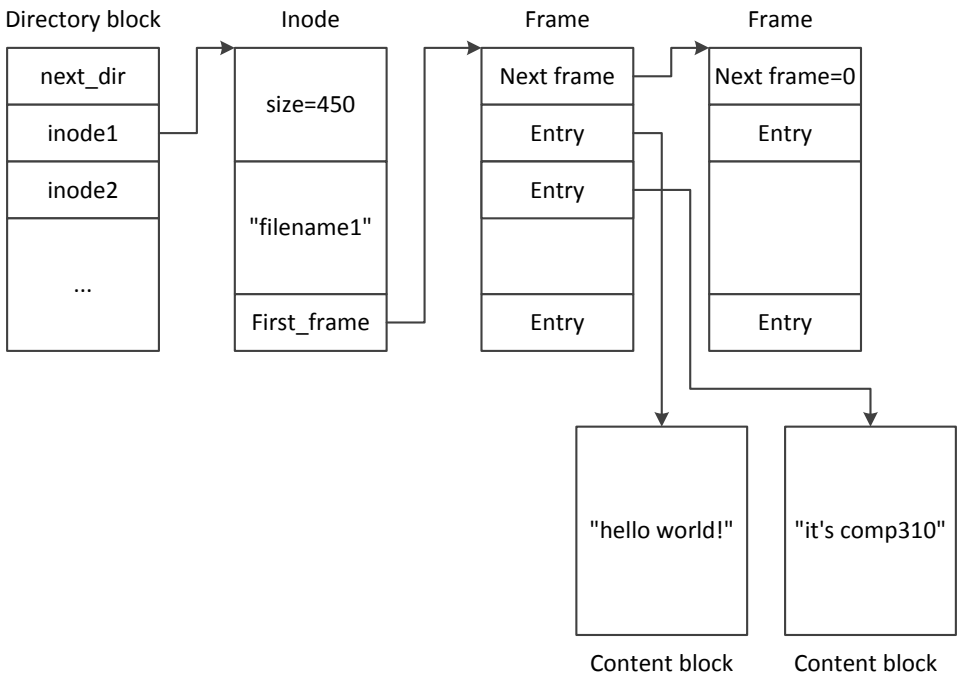
Each directory block describes one directory. Directory blocks are organized as a linked list. The block id for the first directory block is found in the superblock. The second one can be found following the next_dir field of the first one. The last directory block has a next_dir==0. The rest space of a directory stores the name of the directory and the block ids of inodes (containing files).

For simplicity, the linked list never shrinks. If a file gets deleted, the corresponding inode entry in a directory block will be changed to zero, indicating that this entry can be reused next time a file is created. When a file is created, SFS first scans through the linked list to find entry with zero value. If there is a zero entry, the entry will be used. Otherwise, SFS grows the linked list to hold more entries.

File

Inode

An Inode is the data structure that the file system uses to describe a file. Typically a file has a list of attributes (such as name, size, date, etc). An inode is essentially a data block, which contains the file attributes and provides index to frame and content blocks. The below figure demonstrates how a file is stored/indexed on the disk:



As shown, a directory block indexes inodes. An inode indexes frames. A frame indexes content blocks.

- An inode contains file information, such as size and name. And most importantly, it has the block_id for the first frame of the file.
- Frame blocks are also organized as a linked list (similar to that seen in directory blocks). The entries in a frame block contain the block ids of content blocks.
- Each content block stores a fraction of the file content. In this example, the content of the file is “hello world! It’s comp310”, which is split and stored in two content blocks.

We use frames along with content blocks to enable very large files. A file may grow in size as time goes by. So it is not practical to allocate continuous blocks for a file. To allow dynamic growing files, a frame is used to index non-continuous content blocks. In some cases, when the file is very large, one frame may not be enough. So the linked list structure helps.

To simplify the implementation, every time you need to expand the size of a file, you always allocate a “full frame”. For example, assume the block size is 12 bytes and a block id is 4 bytes. So one frame can index two content blocks. Initially the file is empty.

- First, the user writes 7 bytes to the file. SFS allocates two content blocks (C1 and C2) and a full frame indexing these two content blocks (F->C1, C2). In such case, we have two content blocks which make a capacity of 24 bytes. Next, the 7 bytes data is written in the first content block (C1).

- Second, user again writes 14 bytes. The first 5 bytes will be stored in C1 and the remaining 9 bytes go to C2. In this operation, no new frame/content blocks are allocated.
- Next time if user writes exceed the capacity, we again allocate a full frame.

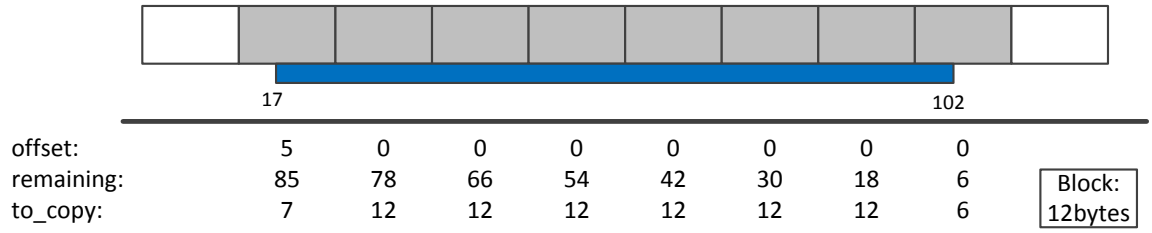
This pre-allocation mechanism explains why there are “size” and “size on disk” when you check the property of a file on Windows. “size” normally represents the actual length of the content while the “size on disk” indicates the capacity of the file.

Read/write

Read/write operation is the most important operations on a file. In a nutshell, read operations need to retrieve the content from scattered content blocks. We define a helper function called sfs_get_file_content() to get the block ids of content blocks within a given range. In the below graph, we give a detailed example.

For example, the file has 10 non-consecutive content blocks. We want to read 85 bytes starting from the 17th byte, which span 8 content blocks.

- First, sfs_get_file_content() returns the block ids of the grey content blocks in the graph. The sfs_get_file_content() hides the facts that content blocks are indexed by a linked list of frames .
- Then we iterate through these blocks and copy the content. The first iteration will copy 7 bytes starting from the 5th byte in the first grey block… Three variables are recommended to control the loop.



Opened files

When an on-disk file is opened, we use an in-memory *file descriptor (fd)* to keep the state of the file. Note that a file descriptor is created in the memory when the file is opened and freed from the memory when the file is closed. A file descriptor never stays on the disk. These are the fields in a file descriptor:

- The *cursor* identifies the position (inside the file content) of the file operations. For example, when we open a file, the cursor is zero, meaning that our read/write operations will be performed from the very beginning of the file. Next if we read five bytes, the cursor will move forward to 5. The user can also use sfs_seek() to manipulate the cursor.

- The inode id of the file. Whenever SFS needs to do read/write or retrieve the size/name of the file, it needs to go for the inode.

All file descriptors are kept in a file descriptor table. So, when a file is opened, user gets a file descriptor number (fd) indexing the file descriptor in the table.

Testing

Build with and without assignment 1

Along with this document, seven source code files are distributed: fs.c, ext.c, exta1.c, ext.h, fs.h, test.c, Makefile, a1/. You are only allow to modify fs.c. Follow the guidelines in the source code file. We provide a correct and simplified version of assignment 1 to allow you to focus on assignment 2 first. You can type "make" to build the program and run the executable "sfs" to test your implementations.

When you get everything ready, you are required to use the code from your own first assignment. Put your assignment 1 code in a1/ and type "make witha1" to build the program with assignment 1.

Test cases

We have 10 test cases. Each case examines a set of interfaces and helper functions. Test cases are arranged in the recommended implementation sequence, i.e., case 2 is dependent on case 1. For each test case, we list the functions to be implemented:

Test case	Points	Helper functions	Interfaces
1	5		sfs_mkfs(), sfs_print_info()
2	15	sfs_alloc_block(), sfs_free_block(), sfs_flush_freemap(), sfs_find_dir()	sfs_mkdir(), sfs_rmdir()
3	10		sfs_open(), sfs_close(), sfs_remove(), sfs_ls()
4	15	sfs_get_file_content()	sfs_write(), sfs_read()
5	5	sfs_resize_file()	sfs_seek()
6	5		feof()
7	5		overwrite
8	15	sfs_resize_file()	write to expand size
9	10		complicated sfs_seek()
10	15		large file (multiple frames)

Handin

If you can make PA2 work PA1, please put your PA1 files under a1/ and handin both. In this case, you will be tested by "make witha1". Otherwise, please leave a1/ empty and test with "make".

In both cases, you'll get an executable named sfs, which runs through all test cases.