7 File System (Part A)

7.1 Introduction

In lab 7, you will write a Unix-like file system with a logging mechanism.

In part A, you will write the lowest three layer and a part of inode layer. You will finish the highest four in part B.

There are one challenge for you to choose in part A if you like:

1. write a "copy on write" mechanism instead of a logging mechanism.

7.1.1 Getting started

> git merge lab7a

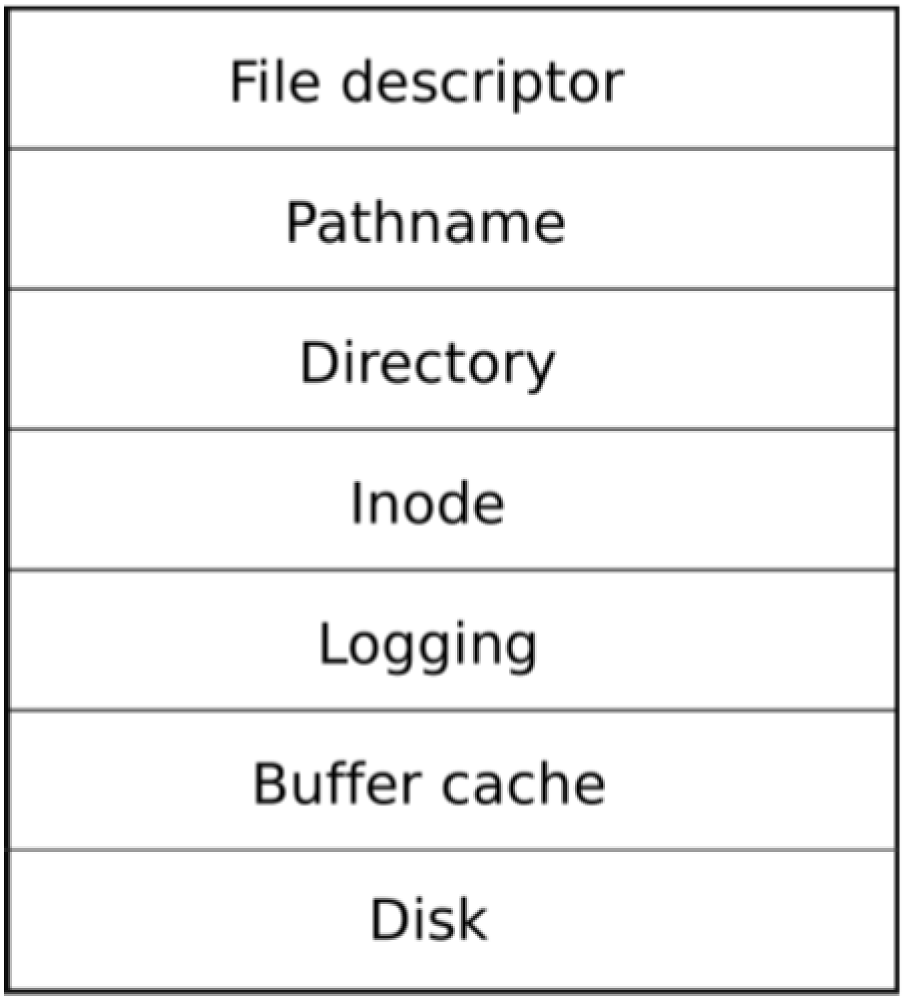
7.1.2 Lab requirements

In this part, implement a file system and answer the follow questions in a few words: Place the answers in a file called answers-lab7a.txt in the top level of your lab directory before handing in your work.

7.2 Overview

7.2.1 File System Layout

There is a file system implementation organized in seven layers:



The disk layer reads and writes blocks on an IDE hard drive, which is almost finished in lab 6.

The buffer cache layer caches disk blocks and synchronizes access to them, making sure that only one kernel process at a time can modify the data stored in any particular block.

The logging layer allows higher layers to wrap updates to several blocks in a transaction, and ensures that the blocks are updated atomically in the face of crashes (i.e., all of them are updated or none).

The inode layer provides individual files, each represented as an inode with a unique i-number and some blocks holding the file's data.

The directory layer implements each directory as a special kind of inode whose content is a sequence of directory entries, each of which contains a file's name and i-number.

The pathname layer provides hierarchical path names like /usr/local/bin/python, and resolves them with recursive lookup.

The file descriptor layer abstracts many Unix resources (e.g., pipes, devices, files, etc.) using the file system interface, simplifying the lives of application programmers.

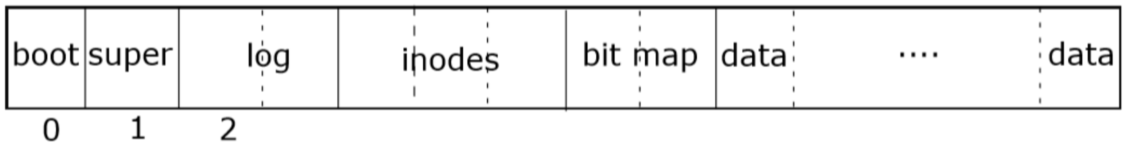
In part A, we will finish the lowest four(the lowest part is almost finished).

7.2.2 Disk Layout

The file system must have a plan for where it stores inodes and content blocks on the disk. To do so, we divides the disk into several sections (which defines in fs.h):

The file system does not use block 0 (it holds the boot sector).

Block 1 is called the superblock (The superblock is filled in by a separate program, called **mkfs**, which builds an initial file system).The superblock contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold the log. After the log are the inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use. The remaining blocks are data blocks; each is either marked free in the bitmap block, or holds content for a file or directory.



Notice that: we used to place our kernel's elf started from blocks 1, but now we should change either the arrangement of file system, or the kernel's elf file placement. We strongly recommend that you use the second solution, which is based on a very simple fact that the kernel should also be managed by the file system. As for whether you want to modify the boot loader further to read the kernel's elf, or directly specify the kernel's elf file placement, it does not matter.

7.2.1 Buffer Cache Layer

7.2.1.1 Overview

The buffer cache has two jobs:

1. synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy.
2. cache popular blocks so that they don’t need to be re-read from the slow disk.

The code you should write is in bio.c.

7.2.1.2 Binit

The function **Binit**, called by main, initializes the data structure with the NBUF buffers. Make sure that all other access to the buffer cache should refer to this data structure. A buffer should have two state bits associated with it. B_VALID indicates that the buffer contains a copy of the block. B_DIRTY indicates that the buffer content has been modified and needs to be written to the disk.

7.2.1.3 Bread and Bget

Bread will return a buffer for given sector with the data ready. Bread calls **Bget** to get a buffer for the given sector. If the buffer needs to be read from disk, **Bread** calls **iderw** to do that before returning the buffer.

Bget should return the buffer directly if there is such a buffer. If not, we recommend that you reuse a buffer in following order:

1. A buffer with refcnt=0.
2. A buffer that not locked and not dirty. (Remember to clear B_VALID flag.)
3. Waiting for a buffer released by other Bxx.(Or simply panic instead.)

> Q: xv6 simply panic here，but why?

It is **important** to make sure that there is at most one cached buffer per disk sector. To let the data structure of the buffer cache contain a lock may help you to reach this goal.

It is also **important** to make only one kernel thread at a time use that copy. Giving every buffer a lock may help you to reach this goal.

7.2.1.4 Bwrite and Brelease

Once Bread has read the disk (if needed) and returned the buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does modify the buffer, it must call **Bwrite** to write the changed data to disk before calling **Brelease** to release the buffer. Bwrite calls iderw to talk to the disk hardware.

Brelease should release the buffer, and may help the data structure of the buffer cache implement **LRU** mechanism. (Other mechanism is also allowed.)

7.2.2 Logging Layer

7.2.2.1 OverView

If the system crashs and reboots, the file system code will recover from the crash as follows, before running any processes. If the log is marked as containing a complete operation, then the recovery code copies the writes to where they belong in the on-disk file system. If the log is not marked as containing a complete operation, the recovery code ignores the log. The recovery code finishes by erasing the log.

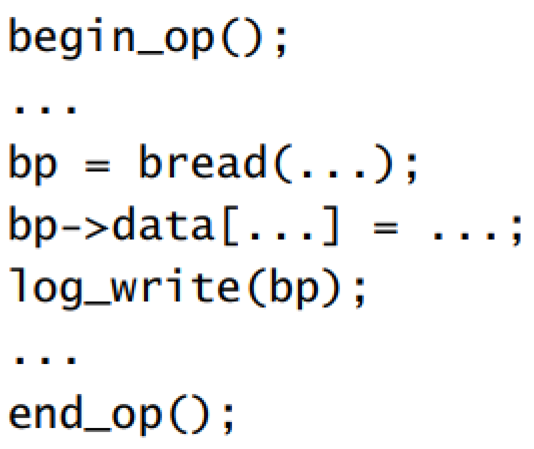
The log resides at a several blocks started with block 2. It consists of a header block followed by a sequence of updated block copies (“logged blocks”). The header block contains an array of sector numbers, one for each of the logged blocks. The header block also contains the count of logged blocks. (Xv6 writes the header block when a transaction commits, but not before, and sets the count to zero after copying the logged blocks to the file system. Thus a crash midway through a transaction will result in a count of zero in the log’s header block; a crash after a commit will result in a non-zero count.)

7.2.2.2 Group Commit

Group commit is a idea to allow concurrent execution of file system operations by different processes. It requires the logging system to accumulate the writes of multiple system calls into one transaction. Only when there are no system call still in this transaction, this transaction will be committed. Because of a fixed amount of space on disk to hold the log, it will not suffer from starvation.

7.2.2.3 Implement

A typical use of the log in a system call looks like this:



```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

begin_op waits until the logging system is not currently committing and have enough free log space. The number of system calls in this transaction and the amount of left space should be counted.

log_write acts as a proxy for bwrite. It records the block's sector number in memory, reserving it a slot in the log on disk, and marks the buffer B_DIRTY to prevent the block cache from evicting it. The block must stay in the cache until committed: until then, the cached copy is the only record of the modification; it cannot be written to its place on disk until after commit; and other reads in the same transaction must see the modifications. log_write notices when a block is written multiple times during a single transaction, and allocates that block the same slot in the log. This optimization is often called absorption. It is common that, for example, the disk block containing inodes of several files is written several times within a transaction. By absorbing several disk writes into one, the file system can save log space and can achieve better performance because only one copy of the disk block must be written to disk.

end_op first decrease the count of system calls.If it is now zero,it commits this transaction by calling **commit**. There are four stages for **commit** :

1. write_log, which write the log.
2. write_head, which write the head block to disk.
3. install_trans, which writes each blocks into the proper place.
4. set the count in head block zero.

7.2.2.4 Notification

It is REALLY IMPORTANT that any code dealing with disk after here must be called inside a transaction.

7.2.3 Inode Layer

7.2.3.1 Get ready for Inode Layer

in part A, you will only write mkfs and a block allocator, to get ready for inode layer.

7.2.3.2 Init the file system

A separate program **mkfs** sets the superblock, the bitmap block and inode block,which place the kernel's elf in the proper place.

7.2.3.3 Block allocator

The block allocator provides two functions: **balloc** allocates a new disk block, and **bfree** frees a block.

Balloc allocates a new disk block according to the bitmap. (The race that might occur if two processes try to allocate a block at the same time is prevented by the fact that the buffer cache only lets one process use any one bitmap block at a time). **Bfree** clear the right bit in bitmap.