Lab7a File System (Part A)

Lab7a File System (Part A)

- Buffer Cache Layer
  - Binit
  - Bread and Bget
  - Bwrite and Brelease
  - Atomicity
- Logging Layer
  - Begin_op and End_op
  - Commit
    - writelog
    - writehead
    - installtrans
  - Initlog
    - readsb
    - recoverfromlog
  - Logwrite
- Inode Layer
  - bzero
  - balloc
  - bfree
- Problems

In this lab, we just build two layers out of the seven layers of file system upon basis of lab 6 (the bottom layer, disk driver) and embed superblock (binary file) into kernel to initialize file system.

Buffer Cache Layer

Binit

In this function, we should initialize **NBUF** buffers, which ensure that all the access to the driver go through these buffer caches. Here we should keep atomicity of two layers, first for the access to buffer (I mean for all the buffer caches), second for the access to one buffer. Thus, we need initialize **bcache.lock** for the first kind of atomicity and **bcache.buf[i].lock** for the second one. Further, we need to initialize a doubly linked list of all the buffers, for some reason we'll explore below. What's worth mentioning is that one buffer has two state bits associated with it, **B_VALID** indicating that the buffer contains a copy of the block and **B_DIRTY** indicates that the buffer content has been modified and needs to be written to the disk. These two bits will be set to inform the below layer, disk driver, to read or write a block of data (I just set its size to 512 bytes) to the disk.

Bread and Bget

Bread will return a buffer of given sector with its data ready (Valid). Bread calls Bget to get a buffer for the given sector. If the buffer needs to be read from disk (Invalid), Bread just calls iderw before returning the buffer.

Bget should return a buffer of given sector. First we should acquire **bcache.lock** to ensure the atomicity of the access to all the buffer. Then we just iterate the doubly linked list from beginning to end to find if there is a buffer of the given sector, if found, just increment its refcnt (reference count), release **bcache.lock**, acquire **bcache.buf[i].lock** and then return the buffer.

Lock for each buffer is a sleeplock, which just sleeps when failing to acquire the lock instead of busywaiting, since each interaction with disk will take a long time, and busywaiting will waste the precious operation time of CPU. I've made some changes to sleeplock, organize all the acquirements into a list, and only wakeup the first process waiting for the lock instead of all when releasing the lock.

If we fail to find a buffer of the given sector in the first iteration, we then need to reuse (which means replacing) a buffer that holds a different sector. We choose to scan the buffer list a second time but in the reverse direction. We choose to reuse a buffer satisfying:

- Refcnt = 0.
- Not dirty.

If found, then edit the buffer metadata to record the new device and sector number, flush the flags to inform the disk driver to read from disk, and then just do as what we do when we find the buffer for given sector.

The reason why we scan the buffer list in two different directions in two loops is that we refer the idea of hot-cold page in Linux's memory management. We keep buffers we recently access around the head of buffer list by placing the buffer at the head when releasing the buffer, thus ensuring that recently-accessed buffer will be easier to find in the list from beginning to end, and buffers around the end have not been accessed recently.

If all the buffers are busy, then too many processes are simultaneously executing file system calls, Bget just panics. It may be more graceful to sleep until a buffer becomes available, but would cause a possibility of deadlock.

Bwrite and Brelease

Once Bread has read the disk (if needed) and returned the buffer with data ready 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 backup these modifications on disk before calling Brelease to release the buffer.

Bwrite first check if the caller has the lock of the buffer, if not, panic. And then set the **B_DIRTY** bit, to inform disk driver to write the buffer to disk for the sake of persistence and then calls iderw.

As stated above, consistency between sectors on disk and buffer must be achieved before invoking Brelse. Then since Brelse need modify the metadata of the buffer cache, we should acquire **bcache.lock**, and then decrement refcnt, place the buffer at the head of the doubly linked list due to the principle of locality. Thus checking the most recently used buffers first will reduce scan time when there is good locality of reference. Also the scan to pick a buffer to reuse picks the least recently used buffer by scanning backward.

Atomicity

We should keep the specification that at most one cached buffer per disk sector, to ensure that readers see writes. Changes made to the metadata of buffer cache should be atomic, especially for refcnt. Otherwise, we may reuse a buffer which is actually referred by other processes.

Logging Layer

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 and End_op

Since Begin_op will modify the metadata of log (outstanding, the number of system calls that are not backuped on the disk), we first acquire **log.lock**, and sleep until the logging system is not currently committing, and there is enough free log space to hold the writes from this call and all currently executing system calls and then increment **log.outstanding**. We conservatively assumes that each system call might write up to **MAXOPBLOCKS** distinct blocks.

End_op also will modify the metadata of log, thus need acquire **log.lock**. Then decrement **log.outstanding**, check if log system is committing. If it is, panic, since we've ensured that no new calls can enter the log system when the log system is committing and log system can not commit until all there is no call in the system.

Set **do_commit** to True and **log.committing** to 1 to begin committing when **log.outstanding** equals to 0, meaning there are no calls in the system; otherwise just wakeup sleeping calls.

If need commit, first call **commit** and then set **log.committing** to zero and wakeup sleeping calls atomically.

Commit

When there are system calls which need committing, we go as below.

```
write_log();
write_head();
install_trans();
log.lh.n = 0;
write_head();
```

Due to the requirement of persistence, we need first write all the changes to the log sections on disk and then write the log header into disk, and finally migrate changes from log sections on disk to corresponding blocks on disk and set the number of inconsistent blocks between log sections and data sections to zero on disk (what the last **writehead** does).

writelog

In this function, we should copy modified blocks from buffer cache to log sections, **log.lh.block** records all the blocks which need to be copied. What we should do is first to (get the buffer cache of a log sector and corresponding data sector, then) copy data from data buffer to log buffer, call Bwrite to make the change

persistent on disk and then release these two buffer caches.

writehead

In this function, we write in-memory log header to disk. A crash after the write will result in recovery replaying the transactions' writes from the log, which is ensured by the order that we first write log into disk and then update the log header on disk. And second call to **write_head** in one **commit** is to update the number of inconsistent data sectors (Since all the data buffer has been persistent on data sections on disk, so clear the bit).

installtrans

In this function, we copy committed blocks from log sectors to their corresponding location on data sectors.

Initlog

Every time when we boot the kernel, we should call this function to initialize the log system, which will call **recover_from_log** to check if a crash has occurred, if it is, then need copy the backup from log sections to corresponding data sections on disk.

readsb

In this function, we need to read the superblock from the image to inform the kernel of the place on which the log sectors lie, and its size.

recoverfromlog

As stated above, we need to copy uncommitted blocks from log system to data sections on disk. So first get the log header, the first block of log system, to get information about the uncommitted blocks and then copy uncommitted blocks from log system to data sections according to the log header, and finally update log header on disk.

Logwrite

Since we take group commit, committing several transactions together, to amortize the fixed cost of a commit over multiple operations, every time when we want to modify the data on disk, we just write them to buffer. In this way, we can avoid multiple reads or writes of the same block from disk by committing only the final version of that block.

Inode Layer

bzero

Zero a block and just write changes to buffer.

balloc

In this function, we need allocate a zeroed disk block which involves finding a free block, marking it in use in bitmap on disk, and bzero the block.

bfree

In this function, we just need mark the block unused in the bitmap on disk.

Problems

This lab took me more than a month, because of the final and many little bugs which accumulate in the previous lab and boom now.

First we need to insert a binary file into the kernel image as superblock which is essential to the file system. In the beginning, I just place the superblock at the second block (seek=1) and make kernel begin at the third one. Then when initializing the log, we need to read the log header, boom!!! Then I just copy the design of file system of xv6-public, leave 1000 blocks for the file system and make the kernel start from 1003rd block. Also code in `bootmain.c` need to be updated to read the kernel from disk.

Second, time to call `initlog` also worth mentioning. Since during the function, we will acquire the sleeplock which need point out the owner of the lock and will call `thiscpu`, so `boot_aps` must precede `initlog`, I just copy what xv6-public do to defer it until the first call of `forkret`.

Third, I find that I leave a bug in my mcslock in lab 4 (lock) by adding some checkers of xv6-public, specification `thiscpu->ncli == 1` in sched, since we should only hold the **ptable.lock** during this function. It teaches me how important checkers are. After that I find my kernel can not support multi-CPU since there is no interrupt made by disk when reading or writing buffer into disk is finished. Then I find that we assign the responsibility of responding to the IDE interrupt to the last CPU, but during lab 4 we just make APs busy wait instead of entering the scheduling after booting them. After the fix, it still can not work when I turn on multi-CPU, and finally I find that my mcslock can not support multi-CPU. So I just turn back to spinlock, and it eventually works.