日曜研究室

技術的な観点から日常を綴ります

[xv6 #59] Chapter 5 – File system – Code: logging

テキストの68〜69ページ

本文

システムコールにおけるログの典型的な使い方は次のような感じである。

```
1 begin_trans();
2 …
3 bp = bread(…);
4 bp->data[…] = …;
5 log_write(bp);
6 …
7 commit_trans();
```

begin_trans関数は、ログへ独占的に書き込むためのロックを獲得できるまで待ち、獲得できたら呼び出し元へ返る。
log_write関数は、内部でbwrite関数を使い、書きこもうとしてるブロックの内容をログに追加し、セクタ番号を記録する。
log_write関数は、渡されたバッファを解放しないので、同じトランザクション中に続けてそのブロックを読み込んでも同じ内容が読み取れる。
log_writeは、一つのトランザクション中に特定のブロックへの書き込みが複数ある場合に注意し、ログに対するそのブロックの直前の書き込みを上書きする。

log.c

```
001 #include "types.h"
002 #include "defs.h"
003 #include "param.h"
004 #include "spinlock.h"
005 #include "fs.h"
006 #include "buf.h"
007
008 // Simple logging. Each system call that might write the file system
009 // should be surrounded with begin_trans() and commit_trans() calls.
010 //
```

```
011 // The log holds at most one transaction at a time. Commit forces
012 // the log (with commit record) to disk, then installs the affected
013 // blocks to disk, then erases the log. begin_trans() ensures that
014 // only one system call can be in a transaction; others must wait.
015 //
016 // Allowing only one transaction at a time means that the file
017 // system code doesn't have to worry about the possibility of
018 // one transaction reading a block that another one has modified,
019 // for example an i-node block.
020 //
021 // Read-only system calls don't need to use transactions, though
022 // this means that they may observe uncommitted data. I-node and
023 // buffer locks prevent read-only calls from seeing inconsistent
    data.
024 //
025 // The log is a physical re-do log containing disk blocks.
026 // The on-disk log format:
027 //   header block, containing sector #s for block A, B, C, ...
028 //   block A
029 //   block B
030 //   block C
031 //   ...
032 // Log appends are synchronous.
033
034 // Contents of the header block, used for both the on-disk header
    block
035 // and to keep track in memory of logged sector #s before commit.
036 struct logheader {
037   int n;
038   int sector[LOGSIZE];
039 };
040
041 struct log {
042   struct spinlock lock;
043   int start;
044   int size;
045   int intrans;
046   int dev;
047   struct logheader lh;
048 };
049 struct log log;
050
051 static void recover_from_log(void);
052
053 void
054 initlog(void)
055 {
056   if (sizeof(struct logheader) >= BSIZE)
057     panic("initlog: too big logheader");
058
059   struct superblock sb;
060   initlock(&log.lock, "log");
061   readsb(ROOTDEV, &sb);
062   log.start = sb.size - sb.nlog;
063   log.size = sb.nlog;
064   log.dev = ROOTDEV;
065   recover_from_log();
066 }
067
068 // Copy committed blocks from log to their home location
069 static void
070 install_trans(void)
071 {
072   int tail;
```

```
073
074   for (tail = 0; tail < log.lh.n; tail++) {
075     struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log
    block
076     struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read
    dst
077     memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
078     bwrite(dbuf);  // flush dst to disk
079     brelse(lbuf);
080     brelse(dbuf);
081   }
082 }
083
084 // Read the log header from disk into the in-memory log header
085 static void
086 read_head(void)
087 {
088   struct buf *buf = bread(log.dev, log.start);
089   struct logheader *lh = (struct logheader *) (buf->data);
090   int i;
091   log.lh.n = lh->n;
092   for (i = 0; i < log.lh.n; i++) {
093     log.lh.sector[i] = lh->sector[i];
094   }
095   brelse(buf);
096 }
097
098 // Write in-memory log header to disk, committing log entries till
    head
099 static void
100 write_head(void)
101 {
102   struct buf *buf = bread(log.dev, log.start);
103   struct logheader *hb = (struct logheader *) (buf->data);
104   int i;
105   hb->n = log.lh.n;
106   for (i = 0; i < log.lh.n; i++) {
107     hb->sector[i] = log.lh.sector[i];
108   }
109   bwrite(buf);
110   brelse(buf);
111 }
112
113 static void
114 recover_from_log(void)
115 {
116   read_head();
117   install_trans(); // if committed, copy from log to disk
118   log.lh.n = 0;
119   write_head(); // clear the log
120 }
121
122 void
123 begin_trans(void)
124 {
125   acquire(&log.lock);
126   while (log.intrans) {
127     sleep(&log, &log.lock);
128   }
129   log.intrans = 1;
130   release(&log.lock);
131 }
132
133 void
```

```
134 commit_trans(void)
135 {
136   if (log.lh.n > 0) {
137     write_head();    // Causes all blocks till log.head to be
    commited
138     install_trans(); // Install all the transactions till head
139     log.lh.n = 0;
140     write_head();    // Reclaim log
141   }
142
143   acquire(&log.lock);
144   log.intrans = 0;
145   wakeup(&log);
146   release(&log.lock);
147 }
148
149 // Caller has modified b->data and is done with the buffer.
150 // Append the block to the log and record the block number,
151 // but don't write the log header (which would commit the write).
152 // log_write() replaces bwrite(); a typical use is:
153 //   bp = bread(...)
154 //   modify bp->data[]
155 //   log_write(bp)
156 //   brelse(bp)
157 void
158 log_write(struct buf *b)
159 {
160   int i;
161
162   if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
163     panic("too big a transaction");
164   if (!log.intrans)
165     panic("write outside of trans");
166
167   for (i = 0; i < log.lh.n; i++) {
168     if (log.lh.sector[i] == b->sector)   // log absorbtion?
169       break;
170   }
171   log.lh.sector[i] = b->sector;
172   struct buf *lbuf = bread(b->dev, log.start+i+1);
173   memmove(lbuf->data, b->data, BSIZE);
174   bwrite(lbuf);
175   brelse(lbuf);
176   if (i == log.lh.n)
177     log.lh.n++;
178 }
179
180 //PAGEBREAK!
181 // Blank page.
```

commit_trans関数は、まずディスクへログのヘッダブロックを書き込むので、この時点の直後におけるクラッシュは、リカバリ処理の実行（ログに記録されているブロックを再書き込みする）を引き起こすだろう。
commit_trans関数は、それからinstall_trans関数を呼び、ログに記録されているそれぞれのブロックを読み込み、ファイルシステム上のあるべき場所それを書き込む。
最後に、commit_transはログヘッダにゼロを書きこむので、次のトランザクションが開始した後にクラッシュしても、リカバリのコードによってそのログは無視される。

recover_from_log関数は、起動中、最初のユーザプロセスが開始される前に実行されるinitlog関数か

ら呼ばれる。
recover_from_log関数は、ログヘッダを読み込み、そのヘッダがコミット済みのログがあることを示していたら、commit_transと同じような処理を行う。

filewrite関数に、ログの使い方の例がある。
そのトランザクションはこのような感じである。

```
1 begin_trans();
2 ilock(f->ip);
3 r = writei(f->ip, …);
4 iunlock(f->ip);
5 commit_trans();
```

このコードは、ログ溢れを回避するため、巨大な書き込みを一度に少しのセクタだけに限定し、個別のトランザクションに分割するループ内で実行される。
writei関数の呼び出しは、このトランザクションの一部として、多数のブロック（ファイルのinode, 一つ以上のビットマップブロック、いくつかのデータブロック）を書き込む。
デッドロックを避ける戦略の一環として、begin_transの後にilockを呼び出す。
各トランザクションの周りに効果的なロックがあるので（？）、デッドロックにならないロック順は、トランザクションのロック→inodeのロックである。

file.cのfilewrite関数

```
01 // Write to file f.  Addr is kernel address.
02 int
03 filewrite(struct file *f, char *addr, int n)
04 {
05   int r;
06
07   if(f->writable == 0)
08     return -1;
09   if(f->type == FD_PIPE)
10     return pipewrite(f->pipe, addr, n);
11   if(f->type == FD_INODE){
12     // write a few blocks at a time to avoid exceeding
13     // the maximum log transaction size, including
14     // i-node, indirect block, allocation blocks,
15     // and 2 blocks of slop for non-aligned writes.
16     // this really belongs lower down, since writei()
17     // might be writing a device like the console.
18     int max = ((LOGSIZE-1-1-2) / 2) * 512;
19     int i = 0;
20     while(i < n){
21       int n1 = n - i;
22       if(n1 > max)
23         n1 = max;
24
25       begin_trans();
26       ilock(f->ip);
27       if ((r = writei(f->ip, addr + i, f->off, n1)) > 0)
28         f->off += r;
29       iunlock(f->ip);
30       commit_trans();
31
```

```
32        if(r < 0)
33          break;
34        if(r != n1)
35          panic("short filewrite");
36        i += r;
37      }
38      return i == n ? n : -1;
39    }
40    panic("filewrite");
41  }
```

感想

ログのコードの説明です。

「log_write関数は、渡されたバッファを解放しないので、同じトランザクション中に続けてそのブロックを読み込んでも同じ内容が読み取れる。」の部分の補足。
アプリレベルの処理として、ブロックの内容を読み込む→その内容を変更する→ディスクに反映、という流れを考えると、最後の「ディスクに反映」する処理の最初の方でlog_writeは呼ばれることになります。
log_writeが渡されたバッファを解放しないということは、メモリ上のバッファキャッシュにある、とあるブロックの変更された内容（未だディスクには反映されていない）がそのままメモリ上に残ることになるので、同じトランザクション中にそのブロックの内容を読み取っても、ちゃんと変更された内容（未だディスクには反映されてない。ログには反映されているけど）を読み取れるという事になります。
もしlog_writeが渡されたバッファを解放した場合、同じトランザクション中の同じブロックへの読み込みであっても、ディスクに変更前のデータを取りに行ってしまい、多分おかしなことになります。

前回、xv6にはビットマップブロックが一つしかないんじゃないか、的なことを書きましたが、今回本文で「一つ以上のビットマップブロック」と書かれてる部分があるので、予想が外れたかもしれません。
そのあたりは、以降でだんだん上の層に登っていくにつれて明らかになるんじゃないかと思います。

「デッドロックにならないロック順は、トランザクションのロック→inodeのロックである。」の部分について。
正直なぜなのかはよく分かりません。
begin_tansの構造としては、ロックを獲得→他のトランザクションが終わるまで待つ→トランザクション開始フラグを立てる→ロックを解放、という流れになっていて、ilockの方も粒度は違うけど同じような構造になっています。
粒度が小さいロック→粒度が大きいロック、よりは、粒度が大きいロック→粒度が小さいロック、の方がより安全であるのは何となくわかる気がします。
ただ、個別の順番ではなくて全体で順番を統一されてるかどうかに影響される気はしますが。
本文でもたいして説明されてないので、もしかしたらこの後の節で説明があるのかもしれません。

カテゴリー: 技術 | タグ: xv6 | 投稿日: 2012/4/3 火曜日 [http://peta.okechan.net/blog/archives/1616] |
