システムソフトウェア (CSC.T371)

第9回

月・木 7-8限,　S4-201(旧称S421)講義室

講義担当：渡部卓雄 (Takuo Watanabe) takuo@c.titech.ac.jp

本日のメニュー

- ファイルシステム(2)

xv6のファイルシステム

- Unixのファイルシステムを簡素化したもの
  - インデックスによる割り当て
    - 2レベルインデックス
    - ブロックアルゴリズム
    - ビットマップによる空きブロック管理
  - バッファキャッシュ
  - ログ機構

inodeブロック(1)

```
struct dinode {
  short type;   // ファイルの種類
  short major;  // デバイス番号
  short minor;  // デバイス番号
  short nlink;  // リンク数
  uint size;    // ファイルサイズ(バイト)
  uint addrs[NDIRECT+1]; // データブロック参照
};
```

- inodeブロックには，各ファイルのinodeが構造体dinodeのデータとして格納されている.
- 構造体dinode（大きさは64バイト）
  - shortとuintの大きさはそれぞれ2, 4バイト.
  - NDIRECT(直接参照数)は12

inodeブロック(2)

- inodeブロックの総数は以下のように計算することができる.
  - 1ブロックに格納できるinode数(IPB): 16
    - BSIZE / sizeof(struct dinode) = 1024 / 64 = 16
  - inodeブロックの総数：ninodes / IPB + 1
    - fs.imgでは13
    - 本来は「ninodes / IPB⌉とすべきだが，xv6ではさぼって ninodes / IPB + 1としている．したがってninodesがIPBで割り切れる場合に1ブロック余分にとってしまうことになる（実害はないが）.
      - ちなみに x > 1かつ y > 0 ならば「x / y⌉ は (x - 1) / y + 1 で計算できる.

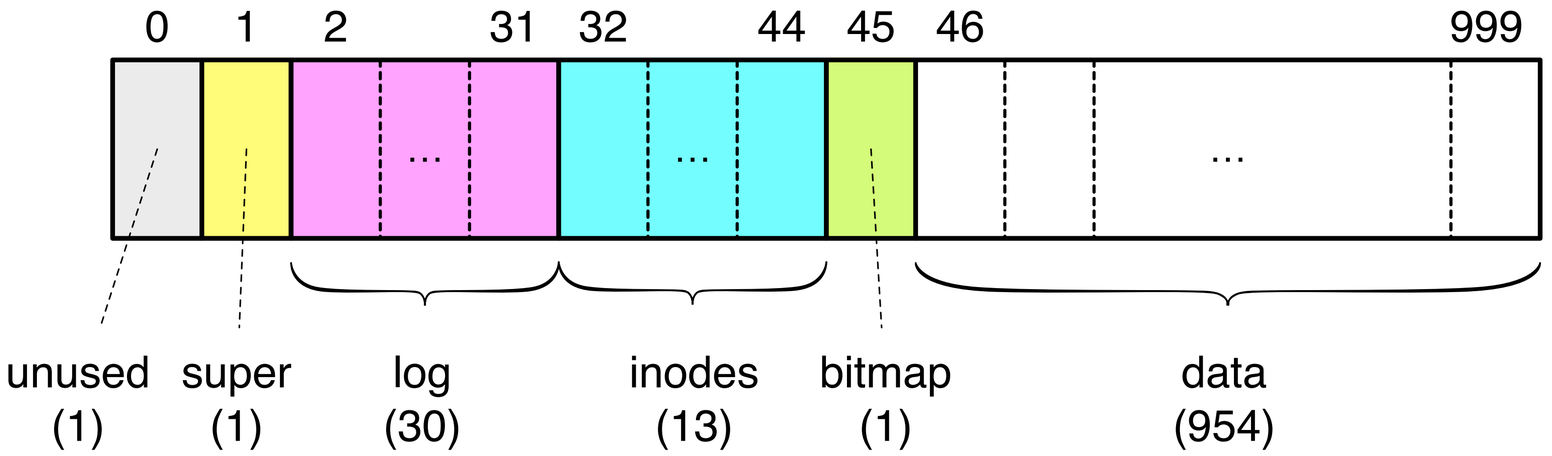
bitmapブロック

- dataブロックが使用済みか否かを管理するためのブロック．ブロック番号に対応するビットが1のときは使用済み.
- 必要なブロック数：size/(BSIZE * 8)+1
  - fs.imgでは1
  - 本来は「size / (BSIZE * 8)」であるが，xv6ではさぼってsize/(BSIZE * 8)+1と計算している.
  - 注）本来はdataブロックの使用・未使用のみを管理すればよいはずだが，ここではディスクを構成する全ブロックに対応するビットを確保している.

dataブロック

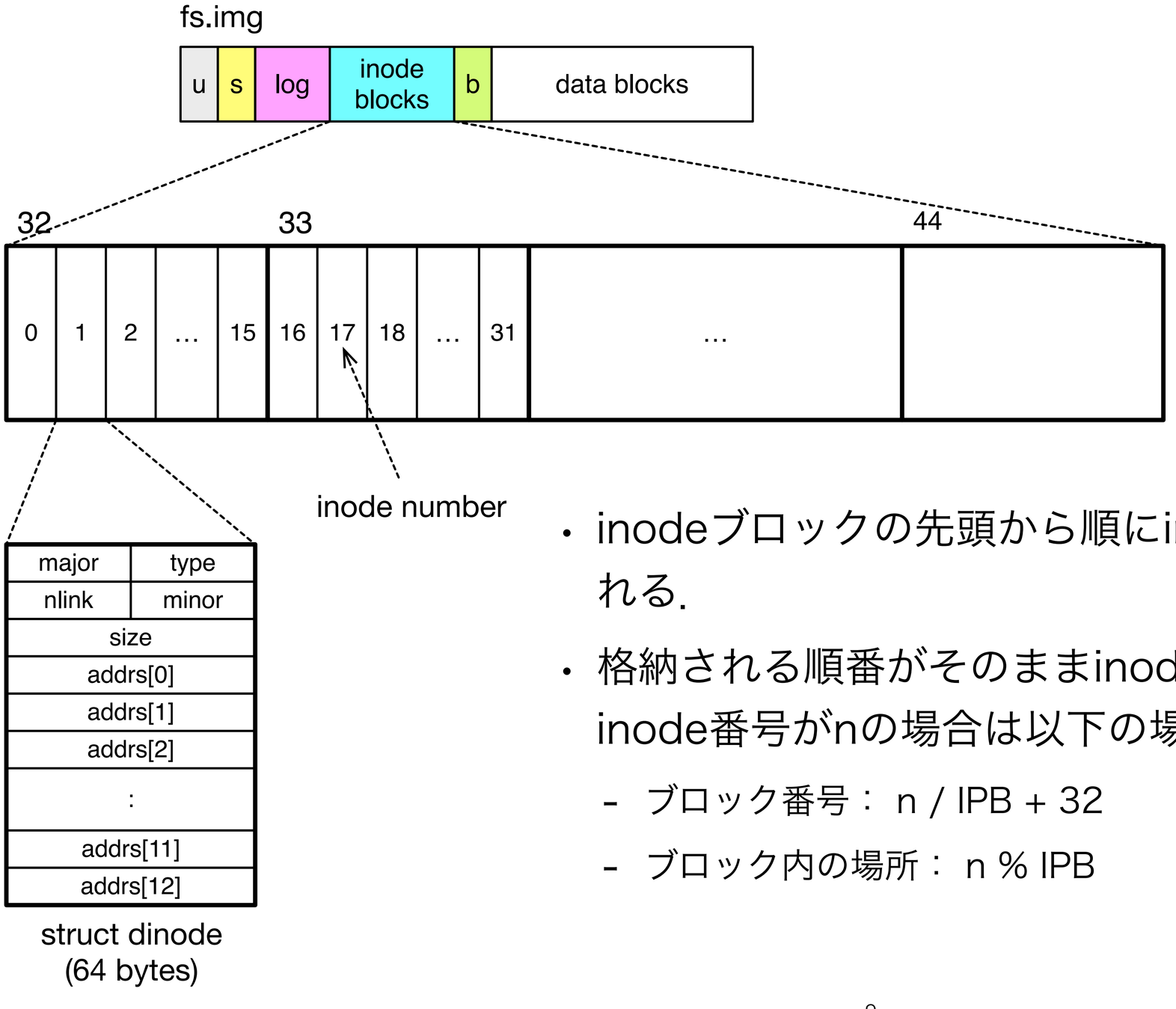
- ファイルの内容や間接参照ブロックのためのブロック.
- dataブロックの数(nblocks)
  - ディスクの総ブロック数から次の各ブロック数の合計を引いた値：ブートブロック，スーパーブロック，inodeブロック，bitmapブロック，logブロック
    - fs.imgでは954
    - 総ブロック数：1000，ブートブロック：1，スーパーブロック：1，inodeブロック：13，bitmapブロック：1，logブロック：30
      - mkfs.cでは以上の合計をnmetaとしている

fs.imgの構成



- mkfsが作成するfs.imgの構成は上の通り

inodeの構造

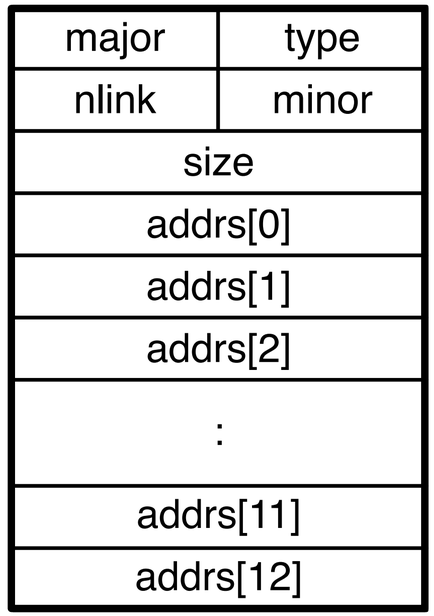


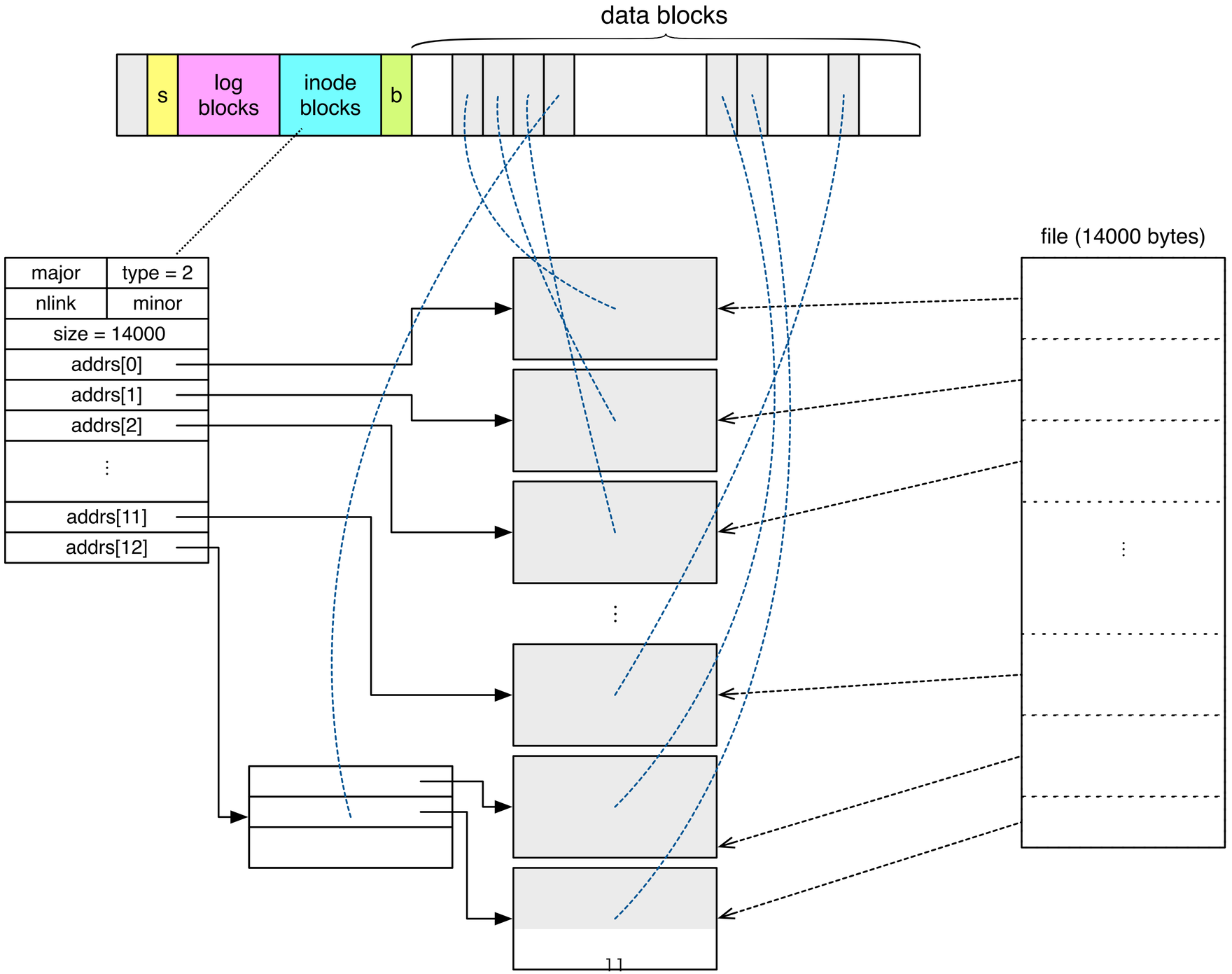
- inodeブロックの先頭から順にinodeの構造体データが格納される.
- 格納される順番がそのままinode番号になる. したがってinode番号がnの場合は以下の場所に格納されることになる.
  - ブロック番号： n / IPB + 32
  - ブロック内の場所： n % IPB

dinode構造体

- ディスク上のinodeを表す
  - type：ファイルの種類
    - T_DIR：ディレクトリ(1)
    - T_FILE：ファイル(2)
    - T_DEV：デバイス(3)
  - major, minor：デバイス番号
  - nlink：ディレクトリからリンクされている数
  - size：ファイルの大きさ（バイト）
  - addrs[0]～addrs[12]：データブロックの参照（ブロック番号）
    - addrs[0]～addrs[11]：直接参照
    - addrs[12]：間接参照（１段）

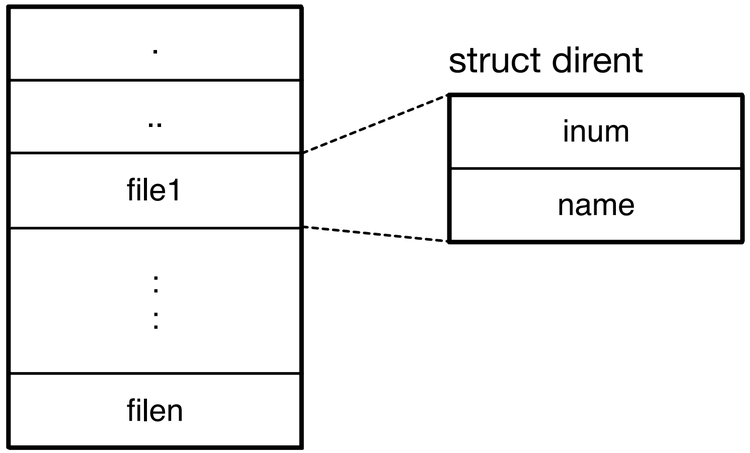
```
struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```




data blocks
s
log blocks
inode blocks
b
file (14000 bytes)
major
type = 2
nlink
minor
size = 14000
addrs[0]
addrs[1]
addrs[2]
addrs[11]
addrs[12]
11

ディレクトリの構造

- ディレクトリの内容はdirent構造体データの列
- dirent構造体
  - inum：ファイルのinode番号
  - name：ファイル名
- すべてのディレクトリには，自分自身を指す "." という名前のエントリと，自分の親ディレクトリを指す ".."という名前のエントリが存在する.
  - ただしルートディレクトリの場合，".."は自分自身となる.

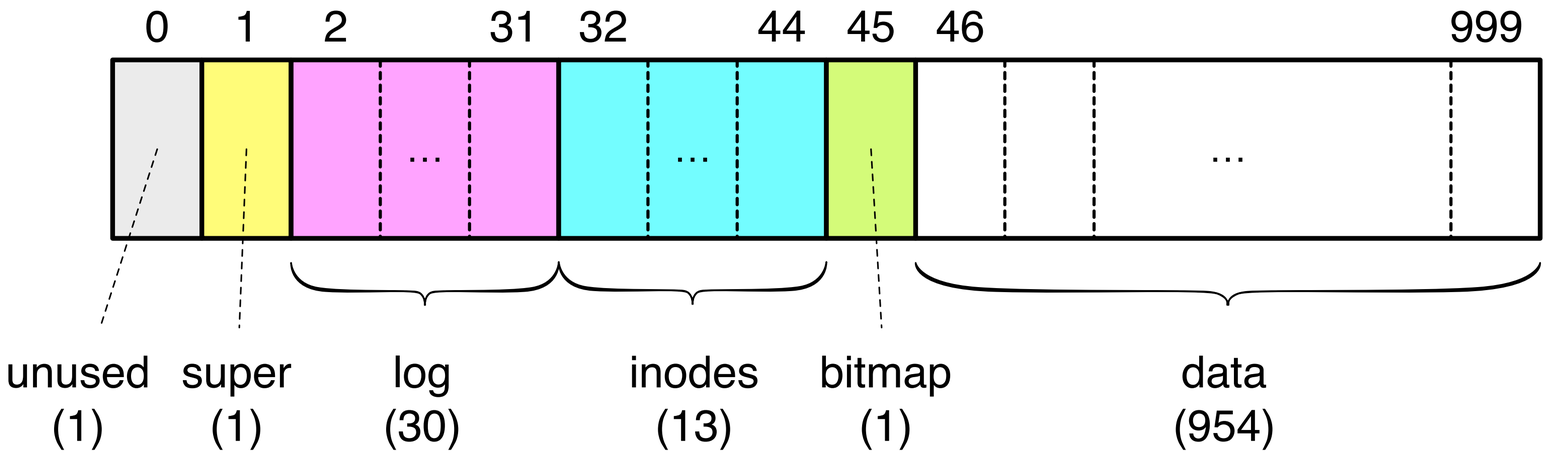


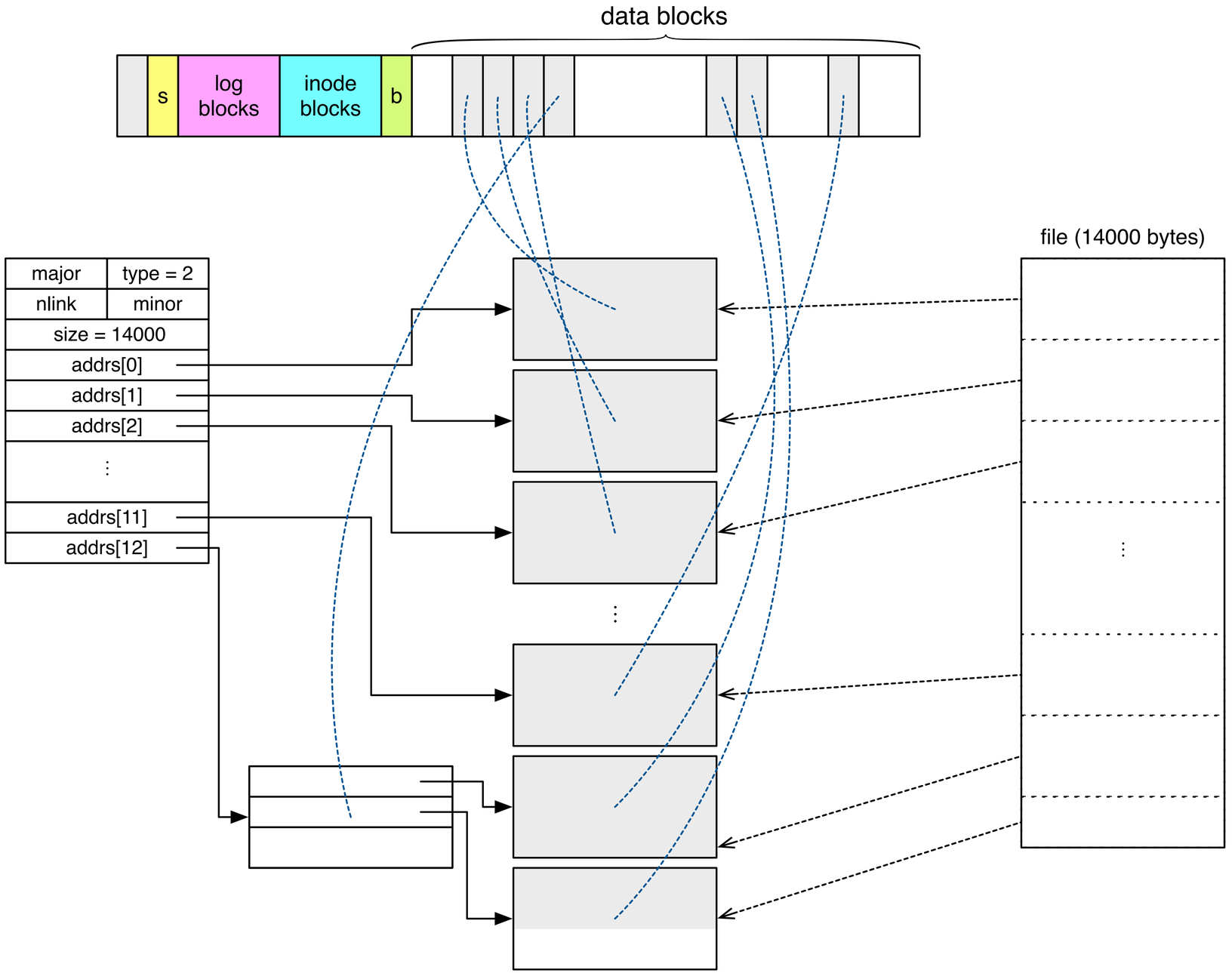

```
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

xv6のファイルシステム階層

- xv6のファイルシステムは以下のような階層構造によって実現されている
  - システムコール(sysfile.c)
  - ファイルディスクリプタ(file.c)
  - パス名(fs.c)
  - ディレクトリ(fs.c)
  - Inode (fs.c)
  - ログ(log.c)
  - バッファキャッシュ(bio.c)
  - 低レベルI/O(ide.c)

fs.imgの構成


0
1
2
31
32
44
45
46
999
...
...
...
unused
(1)
super
(1)
log
(30)
inodes
(13)
bitmap
(1)
data
(954)


data blocks
s
log blocks
inode blocks
b
file (14000 bytes)
major
type = 2
nlink
minor
size = 14000
addrs[0]
addrs[1]
addrs[2]
addrs[11]
addrs[12]

参考：opfs

xv6のディスクイメージ操作ツール（渡部作成）　https://github.com/wtakuo/opfs

（Dockerイメージには含まれている）

- opfs イメージファイル コマンド 引数1 引数2 ... 引数k
  - コマンド
    - diskinfo：ディスクイメージ全体の情報
    - info path：指定したファイルの情報
    - ls path：指定したファイル（ディレクトリ）のリストアップ
    - get path：指定したファイルの内容をホストの標準出力にコピー
    - put path：ホストの標準入力の内容を指定されたファイル名で書き込む
    - rm, cp, mv, ln, mkdir, rmdir
- 他にnewfs, modfsがある

バッファキャッシュ

- ディスク入出力の速度を向上させるために主記憶上に作られるキャッシュ
  - 全てのディスクの入出力はバッファキャッシュを介して行われる
  - キャッシュはブロック単位で割り当てられる.
  - 任意の時刻において，ディスク上の各ブロックについて対応するキャッシュは高々一つ
  - キャッシュの管理はLRU
    - LRU (Least Recently Used)：最後に使った時間が一番古いものを捨てる

buf構造体 (buf.h)

```
struct buf {
  int valid;                // 使用状況のフラグ
  int disk;                 // ディスクI/O待ちフラグ
  uint dev;                 // デバイス番号
  uint blockno;             // ブロック番号
  struct sleeplock lock     // ロック
  uint refcnt;              // 参照カウンタ
  struct buf *prev;         // LRUキュー（2重リスト）
  struct buf *next;
  struct buf *qnext;        // I/O待ちキュー
  uchar data[BSIZE];        // ブロック内容のコピー （キャッシュ）
};
```

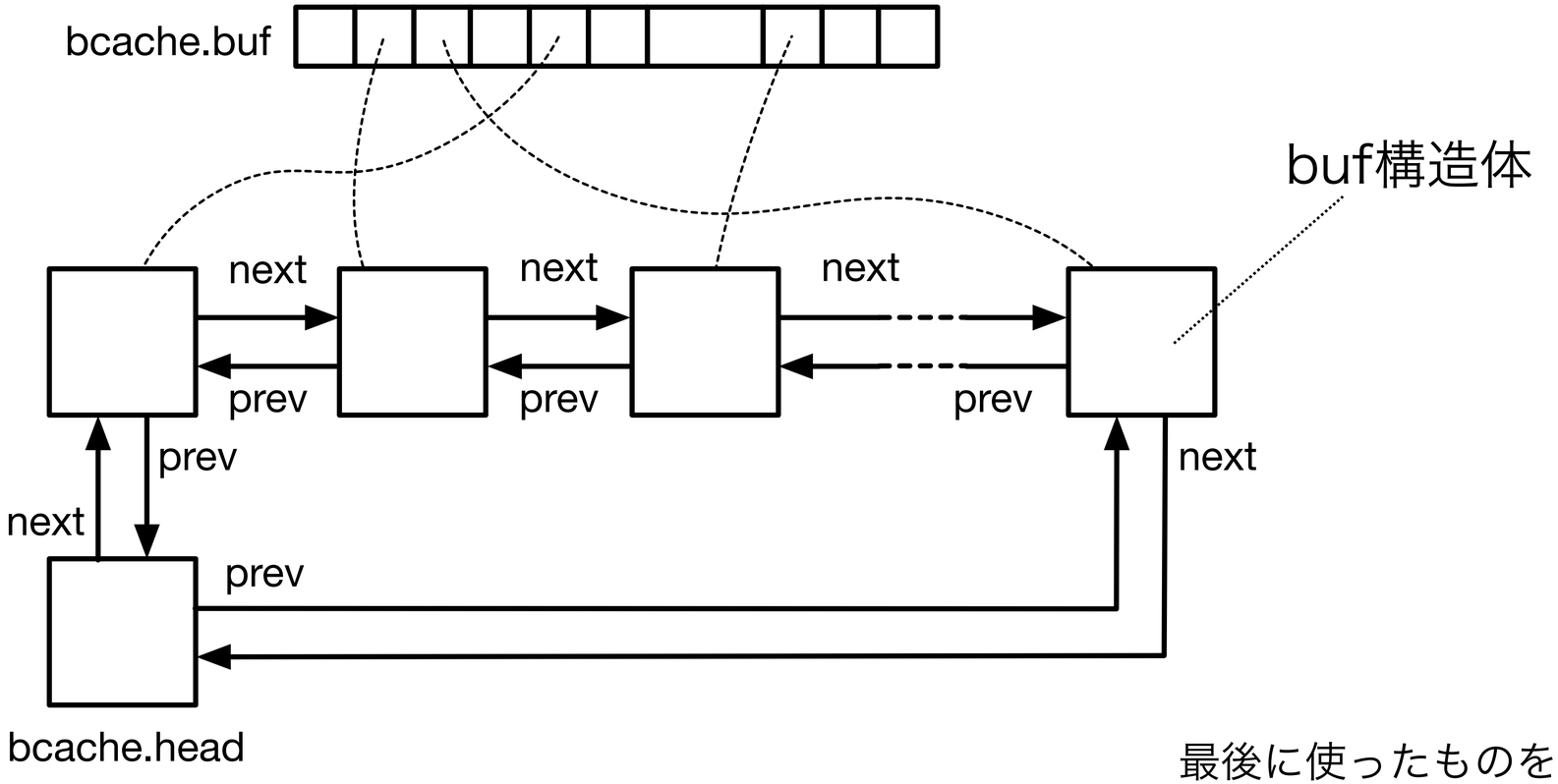
- ディスクの1ブロックのキャッシュ

bcache (bio.c)

```
struct {
  struct spinlock lock; // 相互排他のためのロック
  struct buf buf[NBUF]; // バッファキャッシュの実体
  struct buf head;      // 2重リストのためのダミー
} bcache;
```

- buf構造体の配列bufの要素を2重リストとして管理している
- headは2重リストを作るためのダミー
- head->next が最も最近に使われたキャッシュ

バッファキャッシュの構成



最後に使ったものをリストの先頭に持ってくることでLRUを実現

バッファキャッシュ関連の関数

- struct buf *bread(uint dev, uint blockno);
  - ディスクとブロック番号を指定して，そのブロックのコピーであるキャッシュを確保する.
    - 確保されたキャッシュではvalidフラグが1になっている
- struct buf *bget(uint dev, uint blockno);
  - デバイス（ディスク）とブロック番号を指定してバッファキャッシュを確保する.
- void bwrite(struct buf *b);
  - キャッシュの内容をディスクに書き出す
- void brelse(struct buf *b);
  - キャッシュのrefcntを1減らし，0になったらキャッシュをLRUリストの先頭に移動する

inode（ディスク上） (fs.h)

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

inode（メモリ上） (file.h)

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

```
struct {
  struct spinlock lock;
  struct inode inode[NINODE];
} itable;
```

inode関連の関数

- struct inode *ialloc(uint dev, short type)
  - inodeをディスク上に作成
- void iupdate(struct inode *ip)
  - メモリ上のinodeの内容をディスクに反映
- struct inode *iget(uint dev, uint inum)
  - inode（メモリ上の構造体）を取得
- void i(un)lock(struct inode *ip)
  - inodeのロック（アンロック）
- void iput(struct inode *ip)
  - メモリ上のinodeを破棄（内容はディスクに反映）
- void itrunc(struct inode *ip)
  - ファイルの内容を削る
- int readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n)
  - ファイルの内容をバッファ(dst)に読み出す
- int writei(struct inode *ip, int user_src, uint64 src, uint off, uint n)
  - srcの内容をファイルに書き出す

file構造体 (file.h / file.c)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

ファイルシステムの一貫性

- ファイルシステム全体は比較的複雑なデータ構造であり，その一貫性(consistency)を保つ必要がある
- 以下のような状態で，システムのクラッシュなどにより一貫性が失われることがある
  - ファイルシステムの変更を伴う操作の途中の状態
  - 変更されたバッファキャッシュの内容が書き出されていない状態

一貫性を保つべき箇所

- スーパーブロック
- ブロックの使用状況
- inode
- ディレクトリ

スーパーブロックに関する一貫性

- スーパーブロックに格納されている構造体superblockの各フィールドの値が正しいか
  - sizeがファイルシステムの総ブロック数になっているか
  - nblocks, ninodes, nlogがそれぞれデータブロック，inode, ログブロックの数になっているか
  - logstart, inodestart, bmapstartがそれぞれログブロック，inodeブロック，ビットマップブロックの先頭ブロックの番号になっているか

ブロックの使用状況に関する一貫性

- ビットマップブロックに格納されているビットマップが，各ブロックの使用・未使用を正しく表しているか
- 使用されている各データブロックは，ただ一つのinodeあるいは間接参照ブロックから参照されているか

inodeに関する一貫性

- inodeブロックに格納されている各inode(dinode構造体)について
  - typeが正しいファイルタイプになっているか
    - typeがT_DEVの場合，major, minorが記されているか
  - nlinkが各ディレクトリからの正しい総参照数になっているか
  - addrs（および間接参照ブロック）が使用済みデータブロックを参照しているか
  - addrsおよび間接参照ブロックが参照しているデータブロック数が⌈size/BSIZE⌉になっているか

ディレクトリに関する一貫性

- ディレクトリが参照しているのは正しい（使用済みの）inode番号か
- ルート以外のディレクトリについて，"."，".." が自分自身および親ディレクトリを指しているか
  - ルートディレクトリについては，".." も自分自身を指しているか

fsck

- Unix系のOSにおいて，ファイルシステムの一貫性を検査し，修復できるものについては修復を行うプログラム
- 以前はOSの起動時にfsckによるファイルシステムの検査と修復を行うのが一般的であった
  - ディスク容量の増加に伴い，時間のかかるfsckを起動時に行うことはしなくなった

システムコールの実行

- システムコールはファイルシステムのいろいろな箇所を変更することがある
  - 例：write
    - 空きデータブロックの確保とビットマップの更新
    - inodeの更新
    - データの書き込み
- したがって，実行を途中で中断するとファイルシステムは一貫性を失うことがある
- 中断するくらいなら全く実行しなかったことにするとよいのでは？

ログ機構（ジャーナリング）

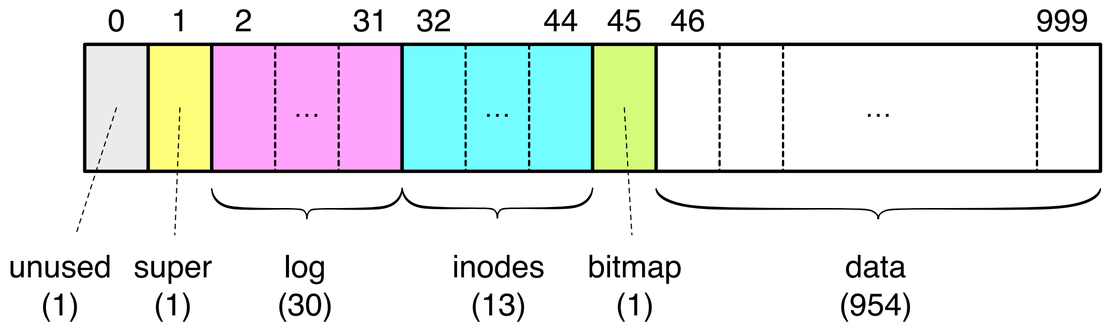
- ファイルシステムの一貫性を保つために，中断すると一貫性を失う可能性のある命令列(トランザクション)を，最後まで完全に実行するか，全く実行しなかったことにする機構
- 多くのファイルシステムで実装されている
  - 例：ext3, JFS, ReiserFS, ZFS, NTFS, HFS+

ログ機構の概要

- トランザクション実行時
  - 実行をディスクに反映する前に，ログ（ジャーナル）と呼ばれる領域に記録する
  - トランザクションが最後まで無事実行できたらその旨をマークする（コミットする）
  - コミットした変更内容をディスクに書き込み，それが無事終わったらログに記録した内容を消去する
- クラッシュからのリカバリー時
  - ログに記録されたトランザクションの実行のうち，コミットされたもののみをディスクに反映する

xv6のログ機構

- トランザクションで変更されるブロックを丸ごとログに記録する
- ログの格納場所（ログブロック）
  - スーパーブロックのlogstartで示されたブロック番号からnlog個
  - 最初のログブロックにはlogheader構造体が記録され，2番目以降に変更されたブロックのコピーが格納される



ログ機構のインターフェース (log.c)

- void begin_op();
  - トランザクションを開始する
- void end_op();
  - トランザクションを終了する
  - 他に実行中のトランザクションがなければコミットをする
- void log_write(struct buf *b);
  - 変更されたバッファをログに記録する

システムコールのアウトライン

```
begin_op();
  ...
  b1 = bread(...);
  modifies b1
  log_write(b1);
  ...
  b2 = bread(...);
  modifies b2
  log_write(b2);
  ...
  ...
  bn = bread(...);
  modifies bn
  log_write(bn);
  ...
end_op();
```

- ファイルを変更する際は，当該ブロックのバッファキャッシュを変更したのち，それを記録する.
- end_op() で変更をファイルに書き出す.

ログヘッダとログの構造 (log.c)

```
struct logheader {
    int n;                      // ログに記録されるブロック番号の数
    int block[LOGSIZE];         // ログに記録されるブロック番号の列
};

struct log {
    struct spinlock lock;
    int start;                  // ログブロックの開始ブロック
    int size;                   // ログブロックの個数
    int outstanding;            // 記録中のトランザクションの数
    int committing;             // 現在コミット中かを表す
    int dev;                    // デバイス（ディスク）
    struct logheader lh;
};
struct log log;
```

ログへの記録例

```
void iupdate(struct inode *ip) {
  ...
  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode *)bp->data + ip->inum % IPB;
  dip->type = ip->type
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  memmove(dip->addrs, ip->addrs, sizeof(ip->addrs));
  log_write(bp);
  brelse(bp);
}
```

- メモリ上のinode構造体の情報をディスク上のdinode構造体へコピーする
  - breadでdinode構造体を含むブロックのキャッシュを得たのち，当該dinode構造体へのポインタを得て，inode構造体の内容をコピーする.
  - 上記ブロックをlog_writeで記録し，brelseで解放する.

log_write

```
void
log_write(struct buf *b)
{
  int i;

  acquire(&log.lock);
  if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
    panic("too big a transaction");
  if (log.outstanding < 1)
    panic("log_write outside of trans");

  for (i = 0; i < log.lh.n; i++) {
    if (log.lh.block[i] == b->blockno)   // log absorption
      break;
  }
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n) {  // Add new block to log?
    bpin(b);
    log.lh.n++;
  }
  release(&log.lock);
}
```

write_log

```
static void
write_log(void)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start+tail+1); // log block
    struct buf *from = bread(log.dev, log.lh.block[tail]); // cache block
    memmove(to->data, from->data, BSIZE);
    bwrite(to);  // write the log
    brelse(from);
    brelse(to);
  }
}
```

- ログヘッダに記録されているブロックの内容をディスクのログブロックに書き出す.

install_trans

```
static void
install_trans(int recovering)
{
  int tail;

  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
    struct buf *dbuf = bread(log.dev, log.lh.block[tail]); // read dst
    memmove(dbuf->data, lbuf->data, BSIZE);  // copy block to dst
    bwrite(dbuf);  // write dst to disk
    if(recovering == 0)
      bunpin(dbuf);
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

コミット

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();     // Write modified blocks from cache to log
    write_head();    // Write header to disk -- the real commit
    install_trans(0); // Now install writes to home locations
    log.lh.n = 0;
    write_head();    // Erase the transaction from the log
  }
}
```

- end_op()により実行される
- 変更されたバッファキャッシュの内容を最初にディスクのログ領域に書き出し，すべて無事に書き出し終えてから本来のブロックに反映する.

クラッシュのタイミングと変更の反映

- 最初のwrite_head完了以前
  - begin_op() 以降の変更はなかったことになる
- 最初のwrite_head完了後〜2番目のwrite_head完了以前
  - ログヘッダおよびログ領域に変更済みのブロックが正しく記録されているので,
    install_transでディスク上のブロックに変更を反映できる.
  - install_trans実行中にクラッシュすると本来変更すべきブロックに未変更のものが残るが,
    ブート時にinstall_transを実行することで最後まで変更できる.
- 2番目のwrite_head完了後
  - すでに変更すべきディスク上のブロックに正しく変更が反映された後である.

期末試験

- 日時：2024年12月2日(月)，7-8限(15:25-17:05)
- 試験室：S4-201(旧称S421)
- 試験範囲：講義で扱った内容すべて
  - 過去の問題は講義webサイトに掲載
- 持ち込み
  - カンニング用紙（A4一枚・手書き）持ち込み可
  - 配布資料とノート，図書，PC，携帯端末，人間等は持ち込み不可

まとめ

- ファイルシステム(2)
  - xv6のファイルシステム