システムソフトウェア

2020年度

第11回 (11/12)

月曜7-8限・木曜7-8限(Zoom)

講義担当：渡部卓雄 (Takuo Watanabe)

http://titech-os.github.io

e-mail: 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

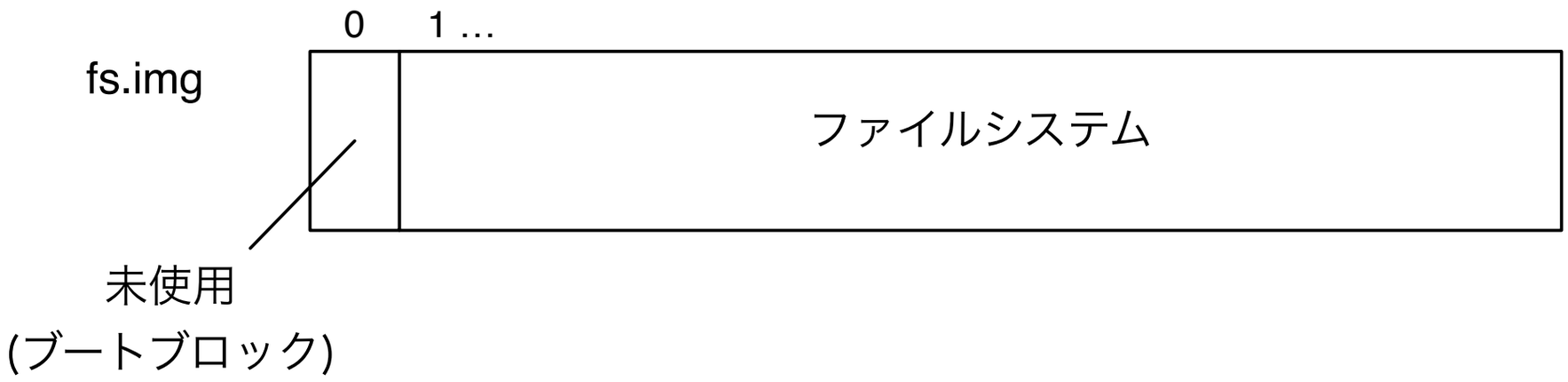
xv6のファイルシステム階層

- 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)

xv6の「ディスク」

- xv6ディレクトリで make を実行するとカーネルの実行形式が作られる
- 続けて make qemu を実行すると，fs.imgというファイルが作られ，このファイルをディスクとみなした仮想コンピュータが起動する.
- fs.img はハードディスクの内容をそのままファイルとしたもので，イメージファイルなどと呼ばれる.

fs.img

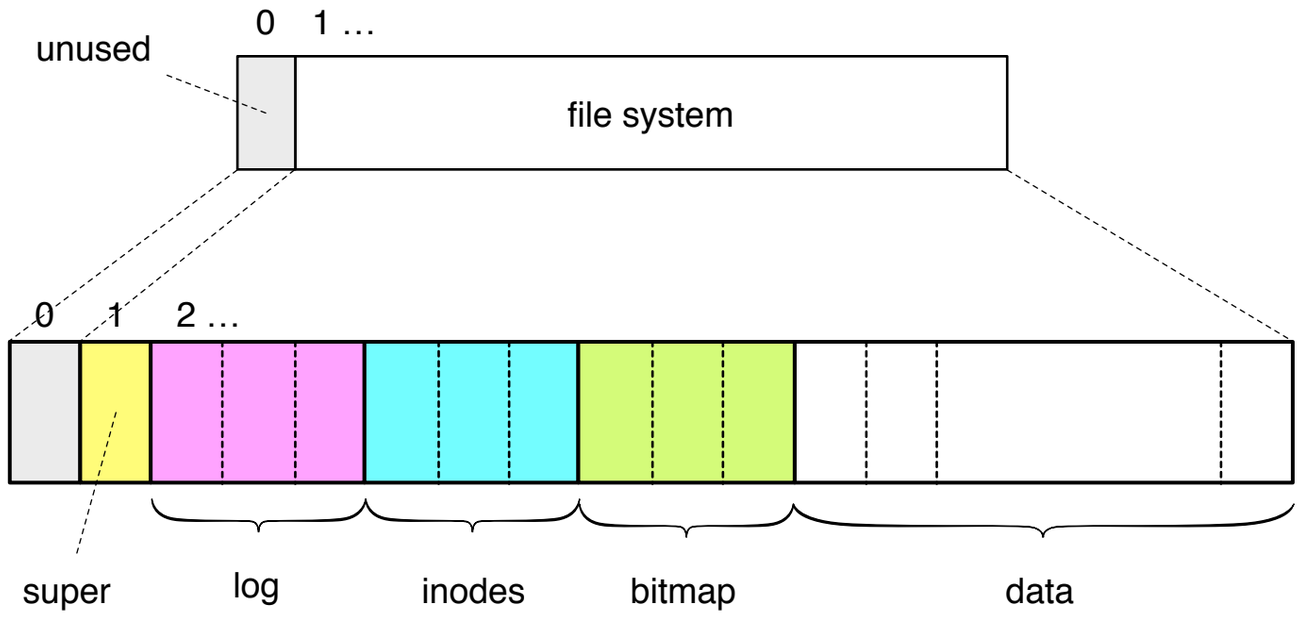


- １ブロックは1024バイト
  - ブロックには0から順に番号がついている（ブロック番号）
- fs.img（ファイルシステムディスク）
  - 第０ブロック：未使用（ブートブロック）
  - 第１ブロック以降：ファイルシステム

xv6の起動（概略）

- 起動時にカーネルのコード（実行形式）を 0x80000000 (KERNBASE)以降に読み込む.
  - QEMUの -kernel オプションで指定
- 0x1000 番地にあるブートプログラムから起動する.
  ブートプログラムは 0x80000000 番地に制御を移す.
- 各CPUコアのスタックを設定し，関数startに制御を移す(entry.S)
- タイマ割り込みの設定をおこない関数 main に制御を移す(start.c)
- 各種の初期化とプロセス init の起動を行う (main.c)

fs.imgの構造



- 先頭(第0ブロック)は未使用，以降は次のブロックからなる.
  - スーパーブロック（1個）
  - logブロック
  - inodeブロック
  - bitmapブロック
  - dataブロック

スーパーブロック (fs.h)

```
struct superblock {
  uint magic;           // マジックナンバー(FSMAGIC)
  uint size;            // ファイルシステム全体のブロック数
  uint nblocks;         // dataブロックの総数
  uint ninodes;         // inodeの総数
  uint nlog;            // logブロックの総数
  uint logstart;        // 最初のlogブロックの番号
  uint inodestart;      // 最初のinodeブロックの番号
  uint bmapstart;       // 最初のビットマップブロックの番号
};
```

- ファイルシステム全体に関する情報が構造体superblockのデータとして第1ブロック（スーパーブロックと呼ばれる）に格納されている.
- カーネルは最初にディスクからこの情報を読み込み, ファイルシステムに関する各種パラメータを得る.

fs.img

- xv6のファイルシステムディスクのイメージファイルfs.imgは，makeの実行時にプログラムmkfsによって作成される.
- その際，スーパーブロックに格納されるパラメータは以下のようになる（nblocksについては後述）.
  - size = 1000
  - ninodes = 200
  - nlog = 30
- したがってfs.imgは以下のようになる.
  - ディスク全体のブロック数は1000．つまりfs.imgの大きさは1000×1024=1024000バイト.
  - inode数は200．よって最大200個までファイルを作ることができる.

mkfs

`mkfs` *imgfile* $file_1$ $file_2$ … $file_n$

- ホストOS上で，Xv6のファイルシステムディスクのイメージファイルを作成するプログラム
  - $file_1$, $file_2$, …, $file_n$ を含むディスクのイメージファイルを*imgfile*という名前で作成する.
  - $file_i$の名前が '_'（アンダースコア）で始まる場合は，作成されるファイルシステム内でのファイル名は先頭のアンダースコアを削除したものになる.
- ソースコードmkfs.c（やや手抜きだが）を読めば，どのようにしてfs.imgが作られるかがわかる.

inodeブロック(1)

- 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

```
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(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」でよいはずだが，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

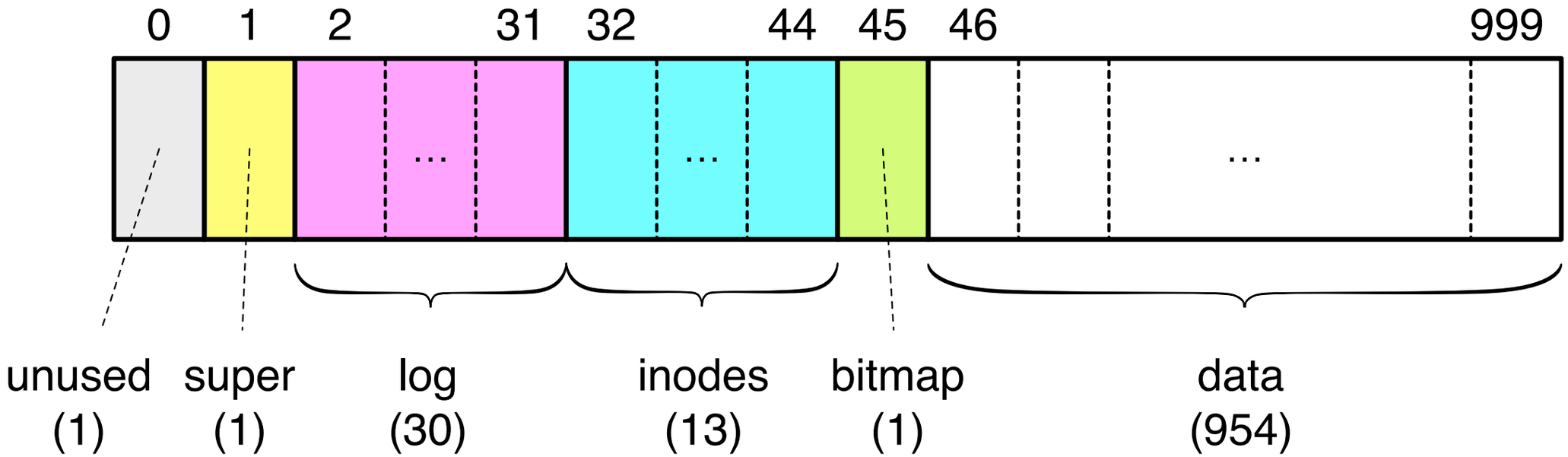
- dataブロックが使用済みか否かを管理するためのブロック．ブロック番号に対応するビットが1のときは使用済み.
- 必要なブロック数：size/(BSIZE * 8)+1
  - fs.imgでは1
  - 本来は「size / (BSIZE * 8)┐であるが，xv6ではさぼってsize/(BSIZE * 8)+1と計算している.
  - 注）本来はdataブロックの使用・未使用のみを管理すればよいはずだが，ここではディスクを構成する全ブロックに対応するビットを確保している.

dataブロック

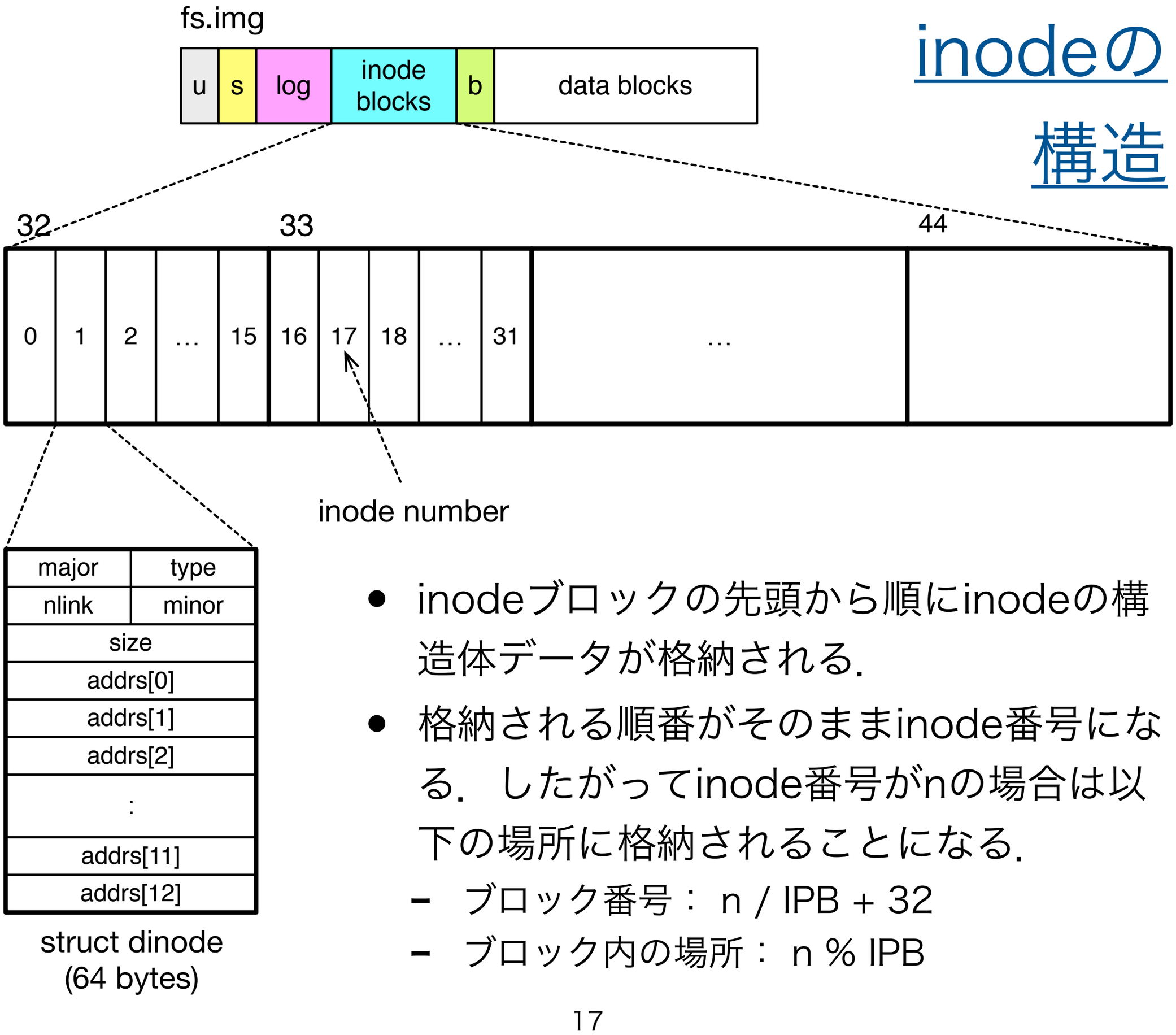
- ファイルの内容や間接参照ブロックのためのブロック.
- dataブロックの数(nblocks)
  - ディスクの総ブロック数から次の各ブロック数の合計を引いた値：ブートブロック，スーパーブロック，inodeブロック，bitmapブロック，logブロック
    - fs.imgでは954
    - 総ブロック数：1000，ブートブロック：1，スーパーブロック：1，inodeブロック：13，bitmapブロック：1，logブロック：30
      - mkfs.cでは以上の合計をnmetaとしている

fs.imgの構成

- mkfsが作成するfs.imgの構成は以下の通り



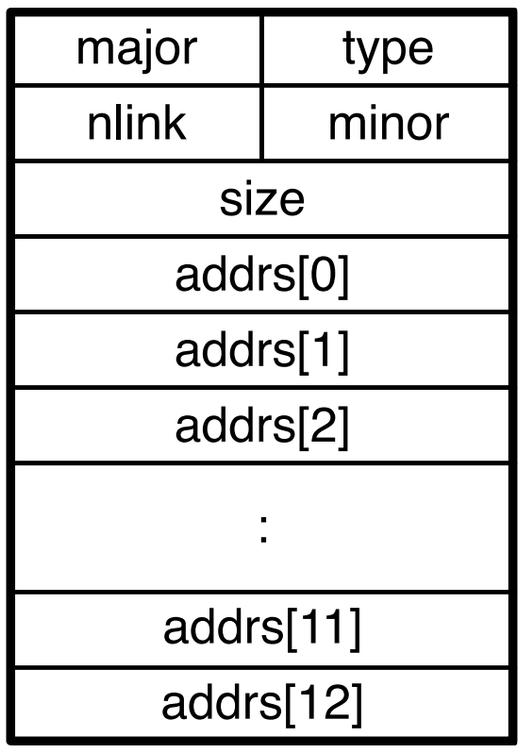
inodeの構造



- 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

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



- 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）

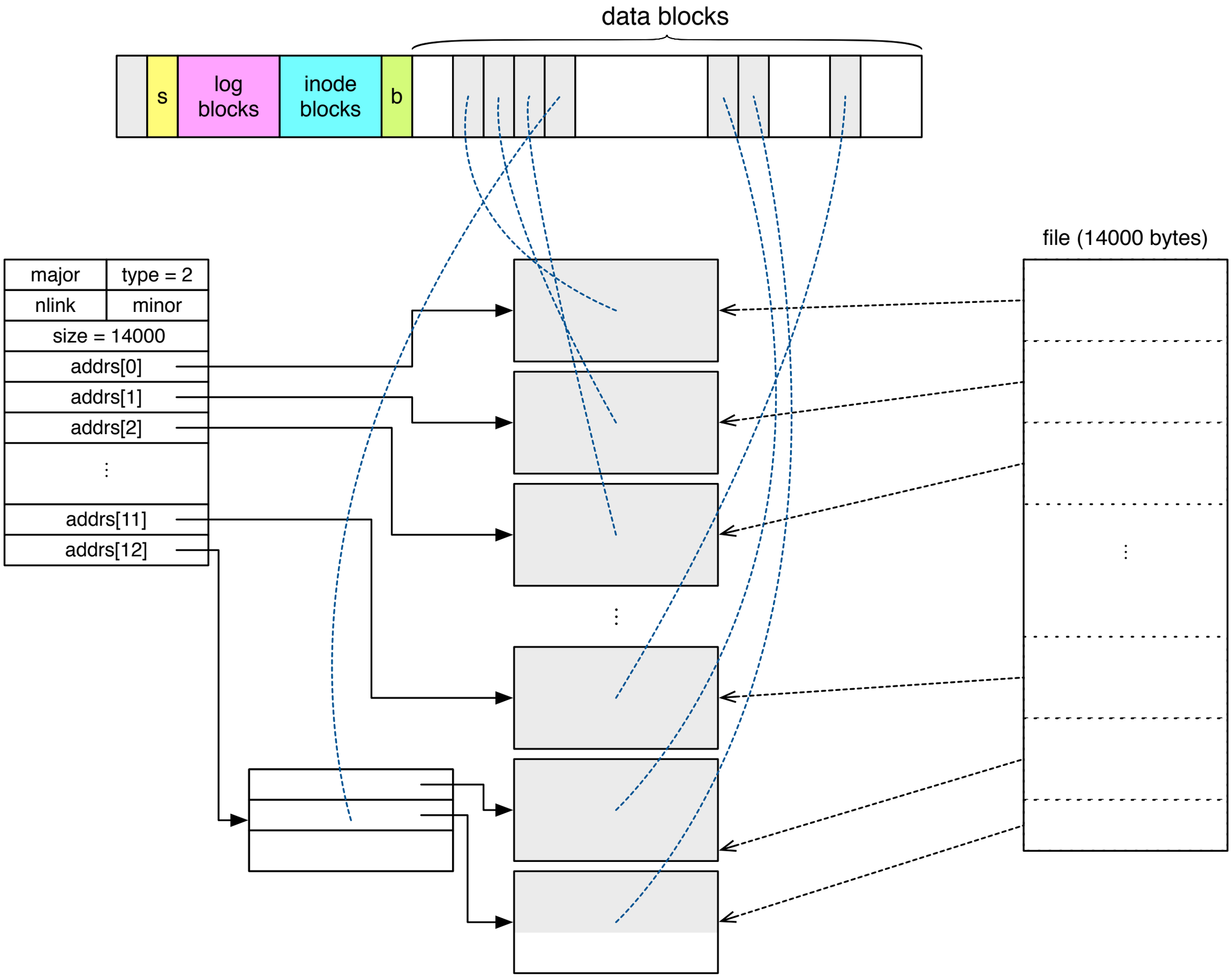
メモリ内のinode構造体

file.h

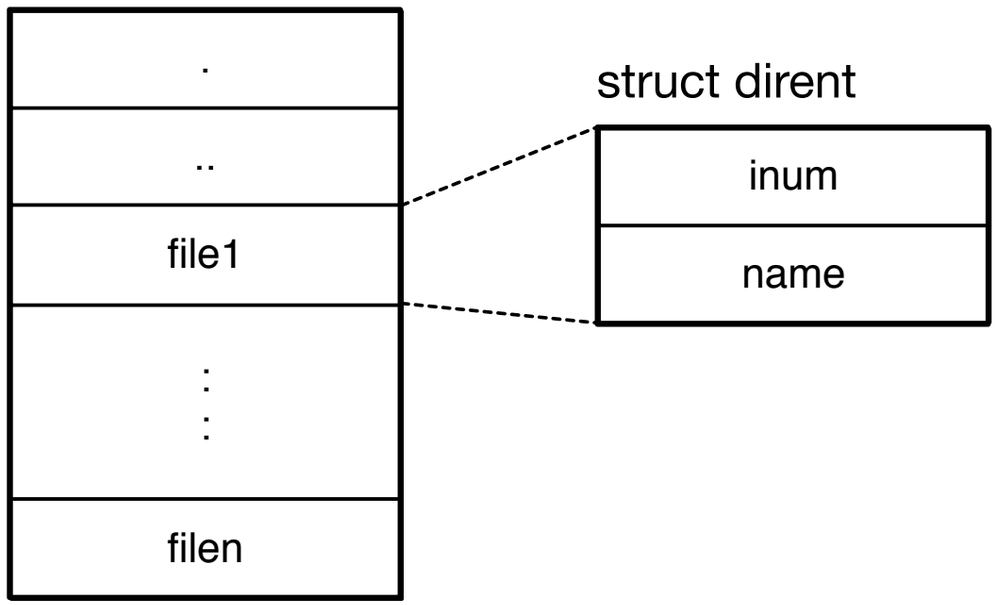
```
struct inode {
  uint dev;                // デバイス番号
  uint inum;               // inode番号
  int ref;                 // 参照数
  struct sleeplock lock; // ロック
  int valid;               // ディスクから読み込まれた正しい内容か？

  short type;              // 以下はdinodeと同じ
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

fs.c

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


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]

ディレクトリの構造




```
#define DIRSIZ 14

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

- 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.

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)

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以下が成り立つ
    - 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
    - ~~各キャッシュを使用しているカーネルスレッド（プロセス）は高々一つ~~
  - キャッシュの管理はLRUで行われる

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
};
```

- ディスクの1ブロックのキャッシュ

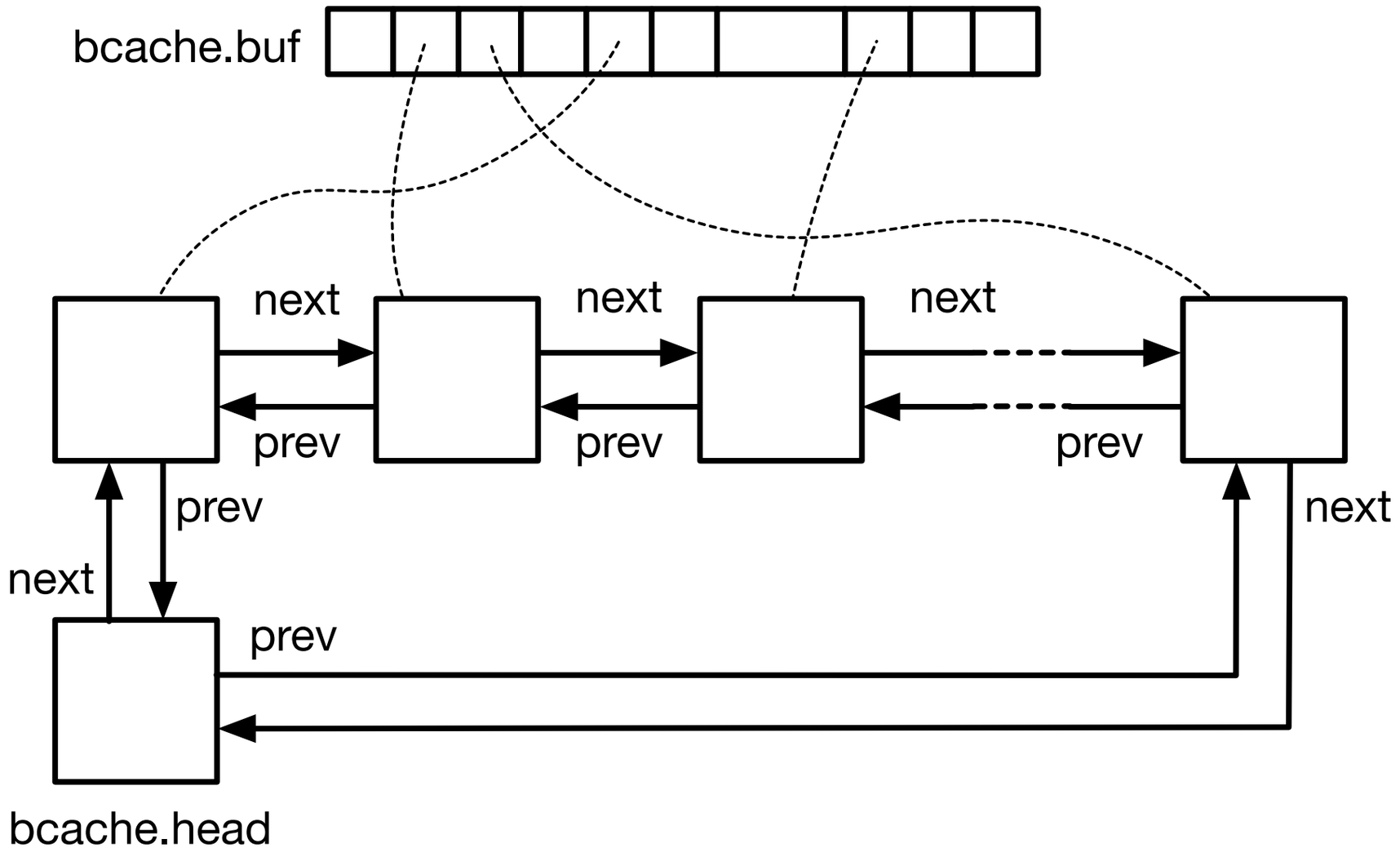
bcache

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



バッファのフラグ

- buf構造体のフィールド
  - refcount
    - バッファの参照数
      - 0の場合はバッファの構造体は使用されていない
  - valid
    - ディスクの内容が読み込まれているか否か

バッファキャッシュ関連の関数

- struct buf *bget(uint dev, uint blockno);
  - 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.
- 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
- 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

バッファキャッシュの使用例(1)

```
int
readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n) {
  ...
  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    if (either_copyout(user_dst, dst, bp->data + (off % BSIZE), m) == -1) {
      brelse(bp);
      break;
    }
    brelse(bp);
  }
  return n;
}
```

- readi(fs.c)
  - 読み出したいブロックのキャッシュをbreadで確保し，その内容を読んだ後はbrelseで解放する

バッファキャッシュの使用例(2)

```
int
writei(struct inode *ip, int user_src, uint64 src, uint off, uint n) {
  ...
  for(tot=0; tot<n; tot+=m, off+=m, src+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    if (either_copyin(bp->data + (off % BSIZE), user_src, src, m) == -1) {
      brelse(bp);
      break;
    }
    log_write(bp);
    brelse(bp);
  }
  ...
  return n;
}
```

- writei(fs.c)
  - readiと同様だが，バッファに書き込んだ後はlog_write（後述）を呼び出している（bwriteを呼び出していないことに注意）

スリープロック(sleep lock)

- 条件を満たすまでプロセス（カーネルスレッド）をSLEEPING状態で待たせる
  - スピンロックと異なりCPU時間を消費しない
  - xv6ではファイルシステム関連のシステムコールで用いられている

```
struct sleeplock {
  // ロックされているか否かのフラグ
  uint locked;
  // この構造体をアクセスするためのスピンロック
  struct spinlock lk;

  // 以下はデバッグ用情報
  char *name;
  int pid;
}
```

スリープロックAPI

```
void acquiresleep(struct sleeplock *lk) {
  acquire(&lk->lk);
  // wakeupがlkに関してSLEEPINGになっているプロセスを全て起こそうと
  // するため, sleepから戻ったのちにwhile文を用いて再チェックする.
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}

void releasesleep(struct sleeplock *lk) {
  acqruire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

sleep

```
void sleep(void *chan, struct spinlock *lk) {
  struct proc *p = myproc();
  // lkによるスピンロックをプロセス構造体のもので置き換える
  if (lk != &p->lock) {
    acquire(&p->lock);
    release(lk);
  }

  // chanで待つように設定してプロセスの状態をSLEEPINGにする
  p->chan = chan;
  p->state = SLEEPING;
  // スケジューラに制御を移す
  sched();
  // wakeupで起こされた
  p->chan = 0;

  // 元のlkによるスピンロックに戻す
  if (lk != &p->lock) {
    release(&p->lock);
    acquire(lk);
  }
}
```

wakeup

```
void wakeup(void *chan) {
  struct proc *p;

  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    // chanについて待っているプロセスを起こす
    if (p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
    }
    release(&p->lock);
  }
}
```

- chanに関して待っている（SLEEPINGである）プロセスを全て起こす（RUNNABLEにする）
  - chanは任意のデータ

sched

```
void sched() {
  int intena;
  struct proc *p = myproc();
  ...
  intena = mycpu()->intena;

  // スケジューラに制御を移す
  swtch(&p->context, &mycpu()->scheduler);

  mycpu()->intena = intena;
}
```

scheduler

```
void scheduler() {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;) {
    intr_on();
    // プロセステーブルをみて, RUNNABLEのプロセスを実行
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        // RUNNINGにしたプロセスに制御を移す
        swtch(&c->scheduler, &p->context);
        // sched() によりここに戻る.
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

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

- `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(...);
  バッファb1を変更
  log_write(b1);
  ...
  b2 = bread(...);
  バッファb2を変更
  log_write(b2);
  ...
  ...
  bn = bread(...);
  バッファbnを変更
  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

fs.c

```
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

log.c

```
void log_write(struct buf *b) {
  ...
  acquire(&log.lock);
  for (i = 0; i < log.lh.n; i++)
    if (log.lh.block[i] == b->blockno) break;
  log.lh.block[i] = b->blockno;
  if (i == log.lh.n) {
    bpin(b);
    log.lh.n++;
  }
  release(&log.lock);
}
```

- ログヘッダ(log.lh)のブロックリストにブロック番号を記録するだけ.

write_log

log.c

```
void write_log() {
  int tail;
  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *to = bread(log.dev, log.start + tail + 1);
    struct buf *from = break(log.dev, log.lh.block[tail]);
    memmove(to->data, from->data, BSIZE);
    bwrite(to);
    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

log.c

```
void install_trans() {
  int tail;
  for (tail = 0; tail < log.lh.n; tail++) {
    struct buf *lbuf = bread(log.dev, log.start + tail + 1);
    struct buf *dbuf = break(log.dev, log.lh.block[tail]);
    memmove(dbuf->data, lbuf->data, BSIZE);
    bwrite(dbuf);
    bunpin(dbuf);
    brelse(lbuf);
    brelse(dbuf);
  }
}
```

- 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.

コミット

```
void commit() {
    if (log.lh.n > 0) {
        write_log();     // 変更されたキャッシュをログに記録
        write_head();    // ログヘッダをログに記録
        install_trans(); // ログに記録された変更をディスクに反映
        log.lh.n = 0;
        write_head();    // ログを消去
    }
}
```

- 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.

まとめ

- ファイルシステム(2)
  - xv6のファイルシステム