UNIXシステムプログラミング

第2回 バッファキャッシュ (1)

2019年10月4日

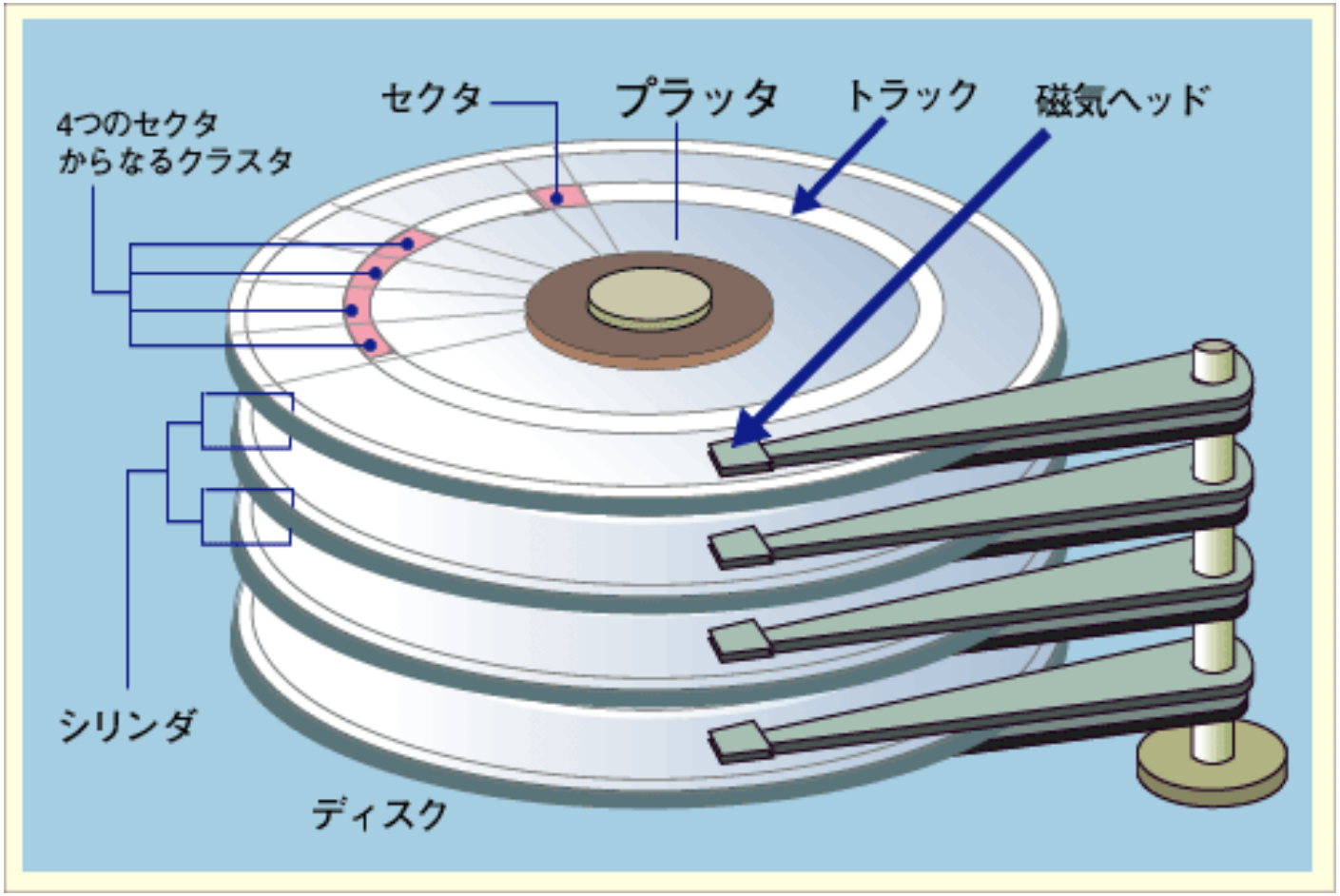
情報工学科

寺岡文男

課題2 : バッファキャッシュ管理

- 課題のねらい
  - 自己参照型の構造体ポインタ
  - 双方向リストの扱い
  - ビット操作
  - 文字列処理
- 課題の概要
  - HDD (Hard Disk Drive)アクセスは非常に時間がかかる (ms単位)
  - HDDアクセスの抑制 → システム全体の効率向上
  - 一度HDDから読み込んだデータをカーネル内部にキャッシュ
  - カーネル内部のバッファキャッシュ管理方法を模擬するプログラムを作成する

HDDの構造

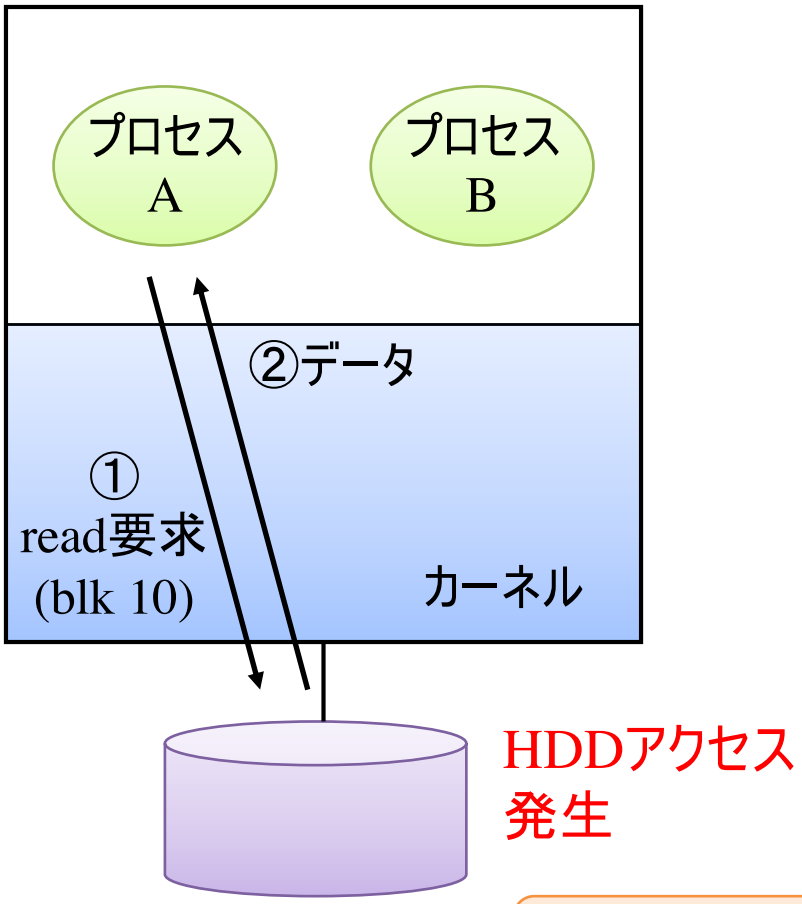


- セクタ (e.g., 512 bytes) 単位でデータの読み出し/書き込みを行う.

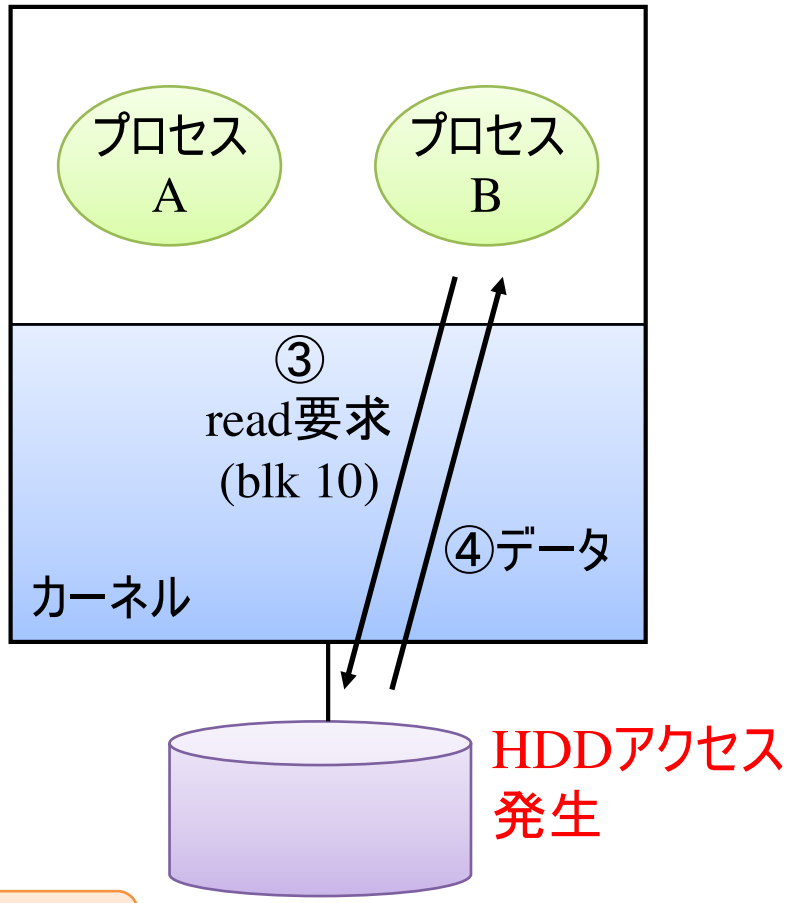
- アクセス時間 = 位置決め時間(シーク) + 回転待ち時間(サーチ) + データ転送時間

動作例

(1)プロセスAがblk-10をread

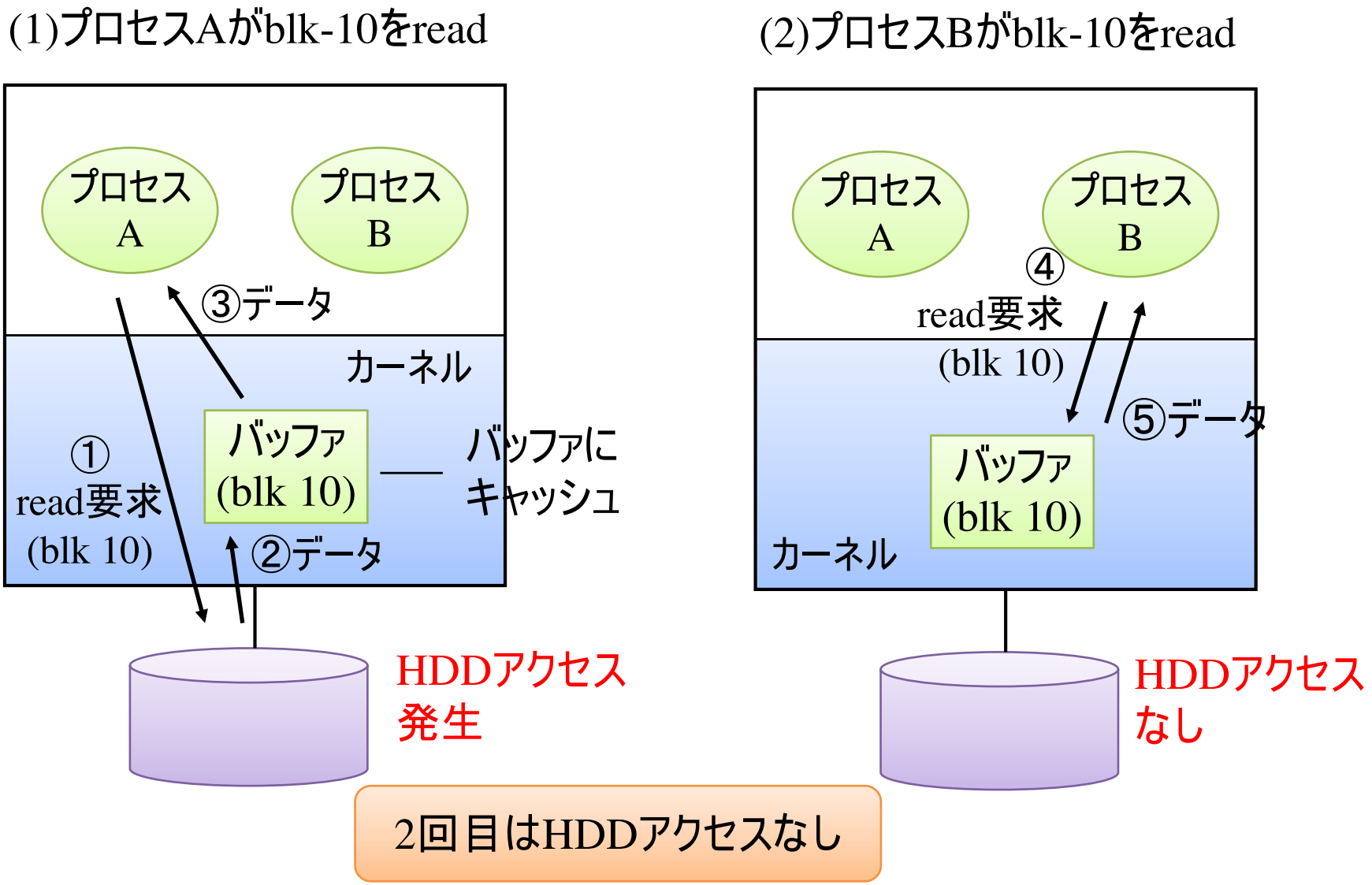


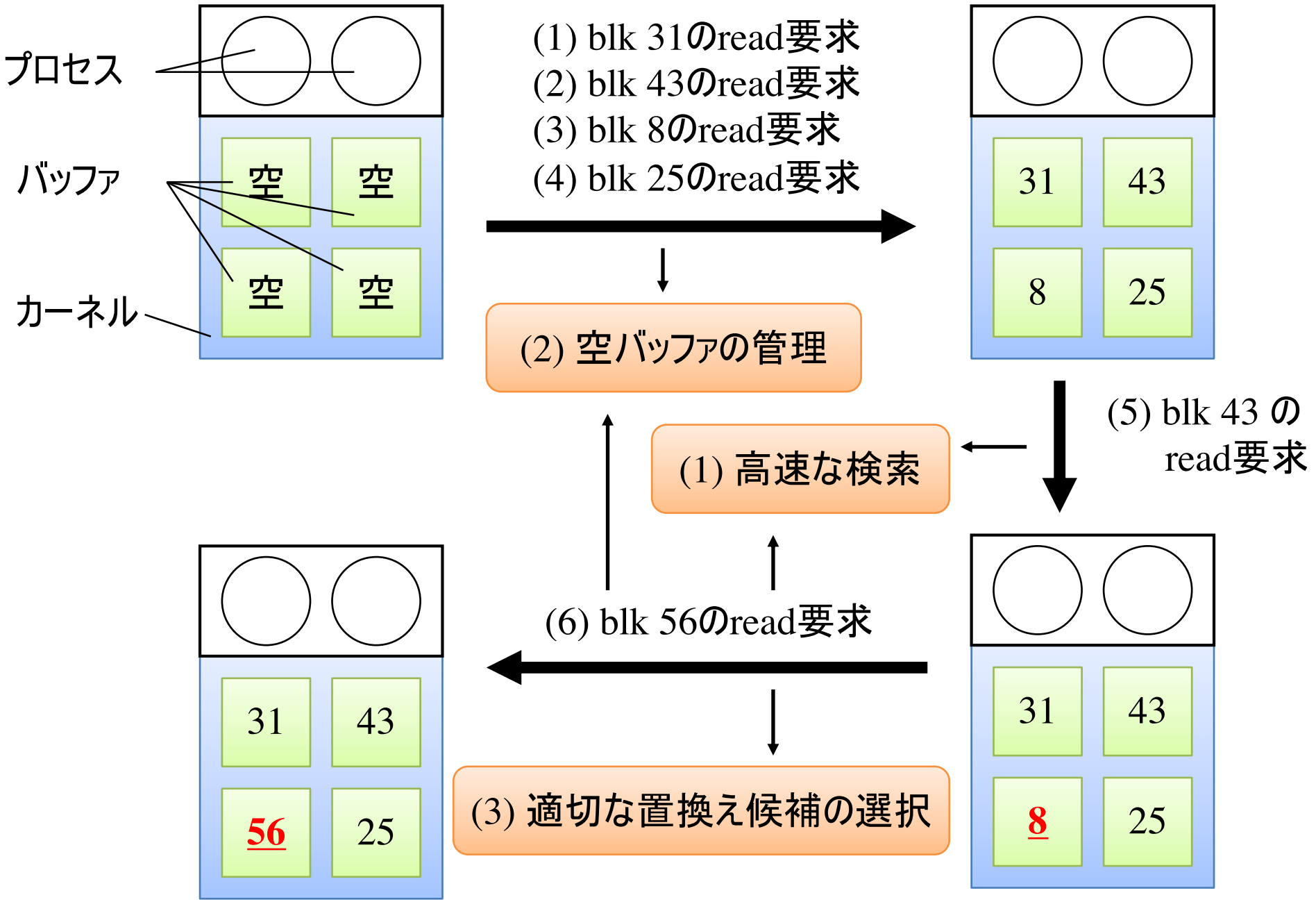
(2)プロセスBがblk-10をread



毎回HDDにアクセス

動作例：キャッシュを導入

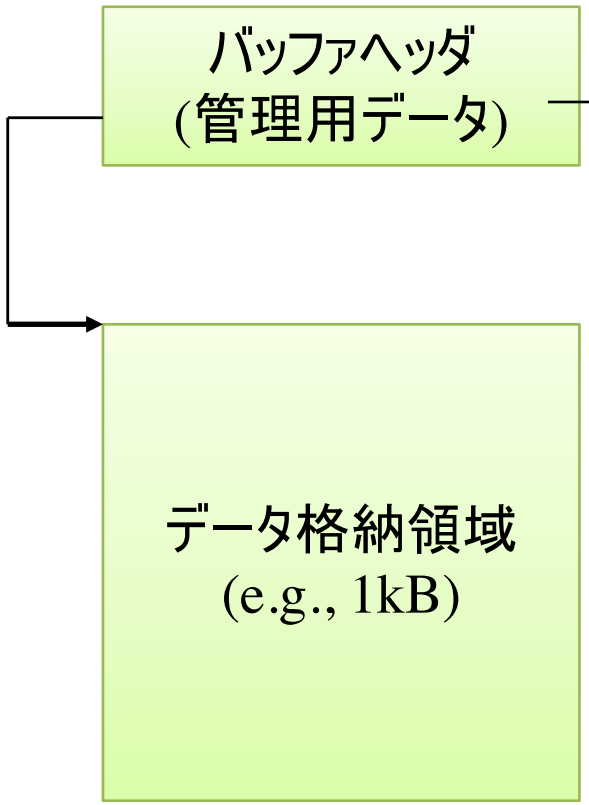



プロセス
バッファ
カーネル
空
空
空
空
(1) blk 31のread要求
(2) blk 43のread要求
(3) blk 8のread要求
(4) blk 25のread要求
31
43
8
25
(2) 空バッファの管理
(5) blk 43 の
read要求
(1) 高速な検索
(6) blk 56のread要求
31
43
56
25
(3) 適切な置換え候補の選択
31
43
8
25

データ構造＋アルゴリズム
＝プログラム

- 問題を解くために適したデータ構造を定義する
- データ構造間の関連を定義する
- データ構造を操作するアルゴリズムを考案する

バッファヘッダ




```
struct buf_header {
  int blkno;   /* 論理ブロック番号 */
  char *cache_data;
               /* データ領域への
                  ポインタ */
};
```

- HDDは固定長のブロック(論理ブロック)の集合
  (簡単のため論理ブロックサイズは1 kBとする)
- 1つの論理ブロックはたかだか1つのバッファで
  キャッシュされる.

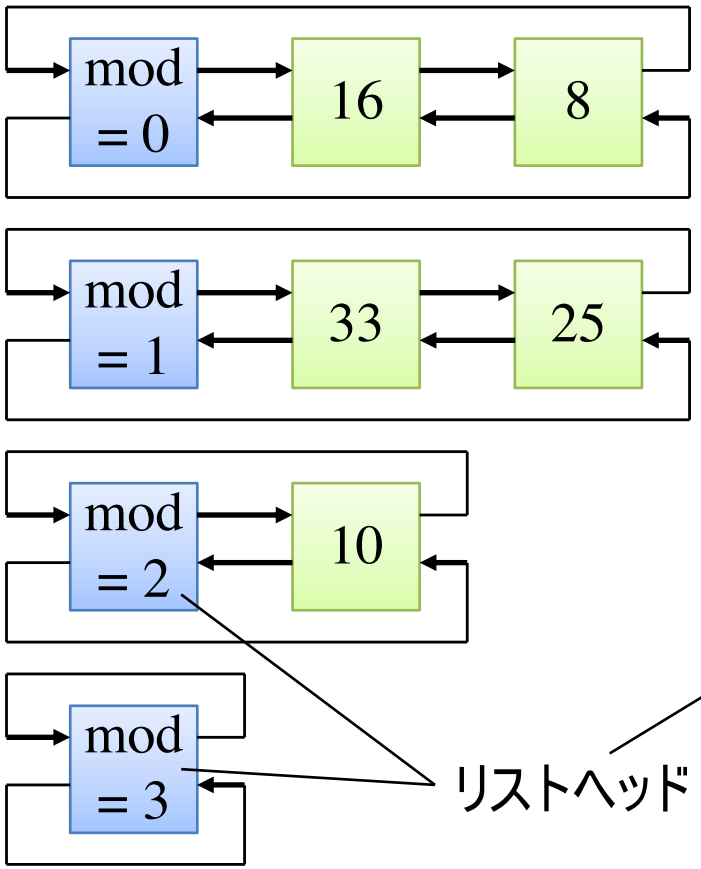
buf_headerに順次必要なメンバを加えていく

効率のよい検索 : ハッシュ関数

- 検索 : 指定されたブロックがキャッシュされているか?
- すべてのバッファヘッダの “blkno” を調べるのでは効率が悪い
- ハッシュ関数を利用
  - 一方向関数
  - 任意長の入力 → 固定長の出力
  - 簡略化のため「4の剰余」というハッシュ関数を想定
  - すべてのブロック番号が, 0～3の4つの値(ハッシュ値)にマッピングされる
  - 同じハッシュ値をもつものをリストで管理

効率のよい検索：双方向リスト

- 後述するように，リストの途中にあるメンバを削除する場合がある → 双方向リストが便利




```
struct buf_header {
  int blkno;
  struct buf_header *hash_fp;
  struct buf_header *hash_bp;
  char *cache_data;
};

#define NHASH  4
struct buf_header hash_head[NHASH];
```

hash_fp, hash_bp: 自己参照ポインタ

リストヘッド：つながる要素と同じ型を持つ

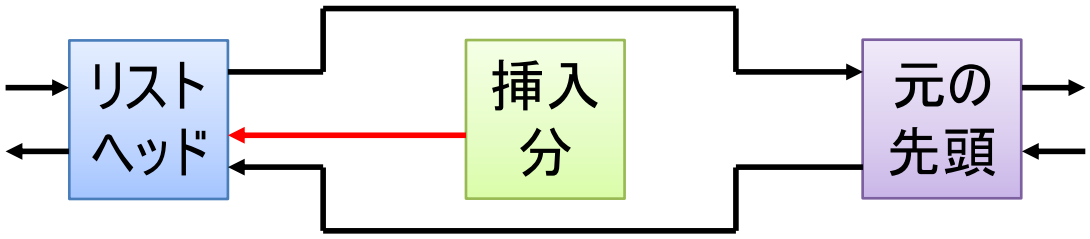
双方向リストの検索

```
01: struct buf_header *
02: search_hash(int blkno)
03: {
04:     int h;
05:     struct buf_header *p;
06:
07:     h = hash(blkno);
08:     for (p = hash_head[h].hash_fp;
                                  p != &hash_head[h];
                                      p = p->hash_fp)
09:         if (p->blkno == blkno)
10:             return p;
11:     return NULL;
12: }
```

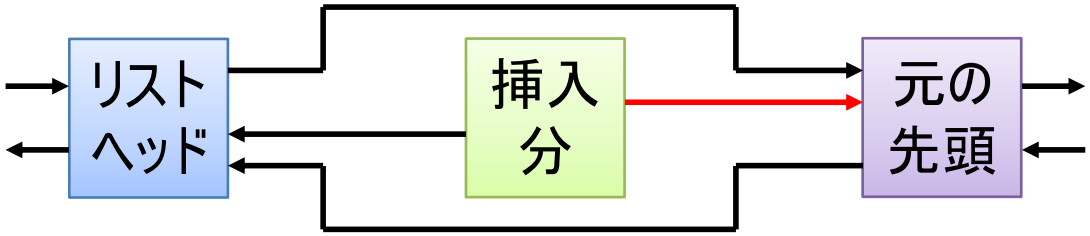
このfor文を理解するように

双方向リストへの挿入 (先頭)

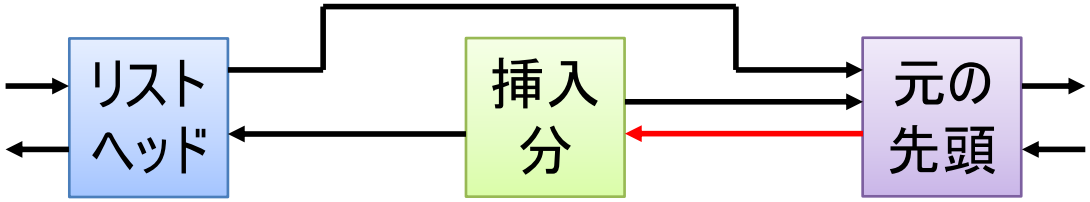
1:



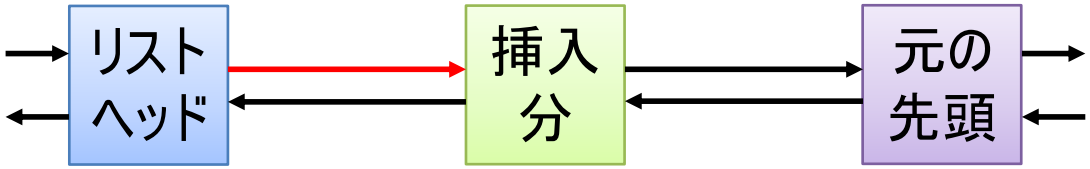
2:



3:



4:




```
void insert_head(
    struct buf_header *h,
    struct buf_header *p)
```

をプログラミングしてみよう

リストの末尾に挿入する

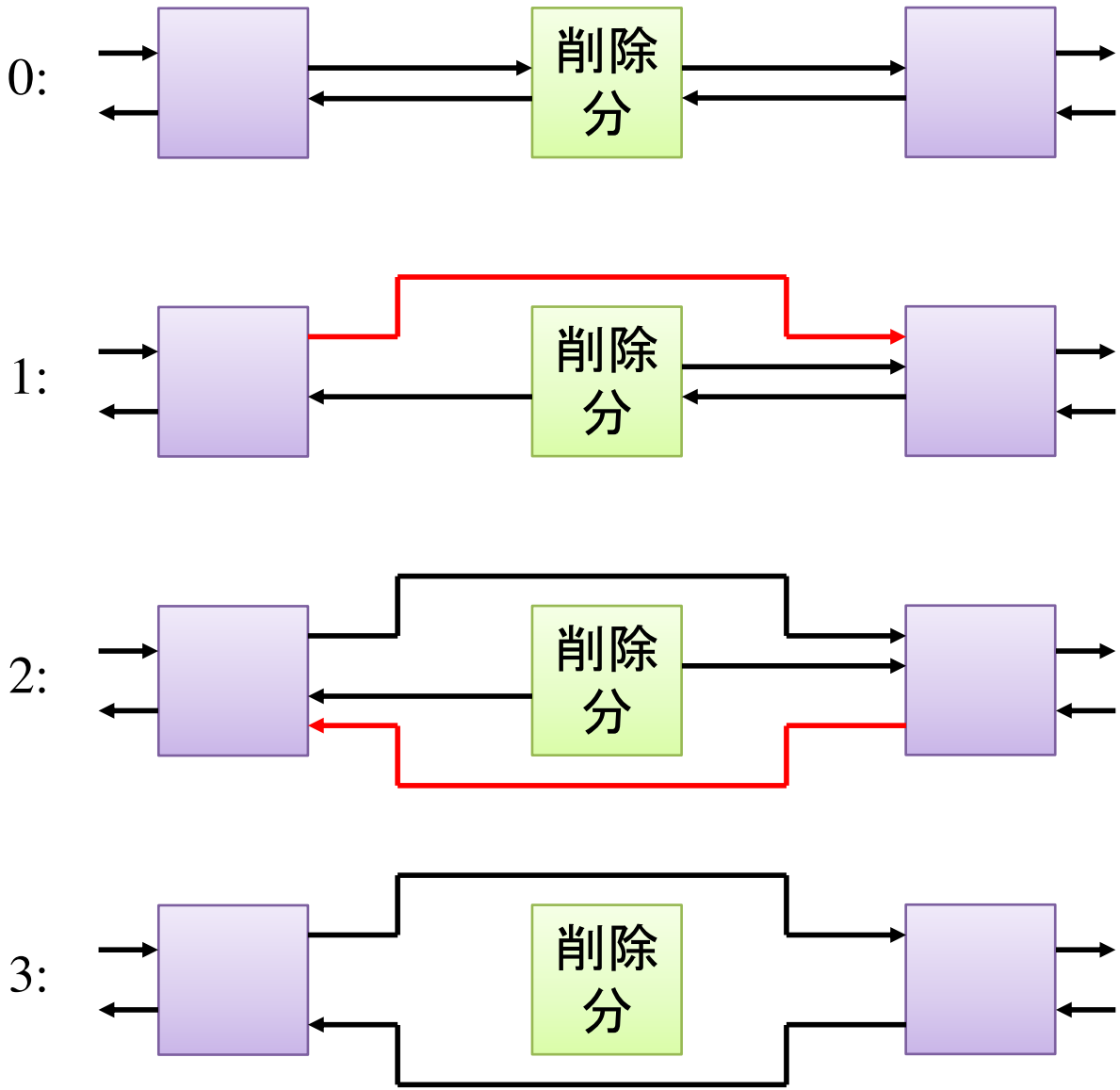
```
void insert_tail(
    struct buf_header *h,
    struct buf_header *p)
```

もプログラミングしてみよう

(2章の練習問題1, 2, 3)

- リストが空のときでも正しく動作することを確認すること

双方向リストからの削除



void remove_from_hash(
struct buf_header *p)

をプログラミングしてみよう
(2章の練習問題4)

バッファの状態

- バッファはさまざまな状態を持つ
- ロックされている (STAT_LOCKED)
  - あるプロセスがこのバッファを占有している．他のプロセスはアクセスできない．
- 有効なデータを保持している (STAT_VALID)
  - データ領域に有効なデータを保持している．
- 遅延書込み (STAT_DWR)
  - データ領域は書換えられており，必要に応じてHDDに書き戻す必要がある．
- カーネルがread/writeをしている (STAT_KRDWR)
  - カーネルがデータ領域に対してreadまたはwriteを行っている．
- 他のプロセスがwaitしている (STAT_WAITED)
  - 他のプロセスがこのバッファがフリーになるのを待っている．
- バッファのデータは古い (STAT_OLD)
  - フリーリストの先頭に戻すため．

バッファの状態

```
struct buf_header {
    int blkno;                          /* 論理ブロック番号 */
    struct buf_header *hash_fp;         /* ハッシュの順方向ポインタ */
    struct buf_header *hash_bp;         /* ハッシュの逆方向ポインタ */
    unsigned int stat;                  /* バッファの状態 */
    char *cache_data;                   /* データ領域へのポインタ */
};

#define STAT_LOCKED   0x00000001 /* ロックされている */
#define STAT_VALID    0x00000002 /* 有効なデータを保持 */
#define STAT_DWR      0x00000004 /* 遅延書込み */
#define STAT_KRDWR    0x00000008 /* カーネルがread/write */
#define STAT_WAITED   0x00000010 /* 待っているプロセスあり */
#define STAT_OLD      0x00000020 /* バッファのデータは古い */
```

バッファの状態を保持するため、メンバ “stat” に対応するビットをセットする

ビット操作

- たとえば，あるバッファがロックされたら “stat” の STAT_LOCKEDビットのみをセットしたい.
- 同様に，あるバッファのロックが解除されたら “stat” の STAT_LOCKEDビットのみをリセットしたい.
  - 対象とするビット以外を変えてはならない.
- STAT_LOCKEDのみをセットする場合
  `p->stat |= STAT_LOCKED;`
- STAT_LOCKEDのみをリセットする場合
  `p->stat &= ~STAT_LOCKED;`
  - “p” はバッファヘッダを指すものとする.
- 2章の練習問題5, 6を解いてみよう.

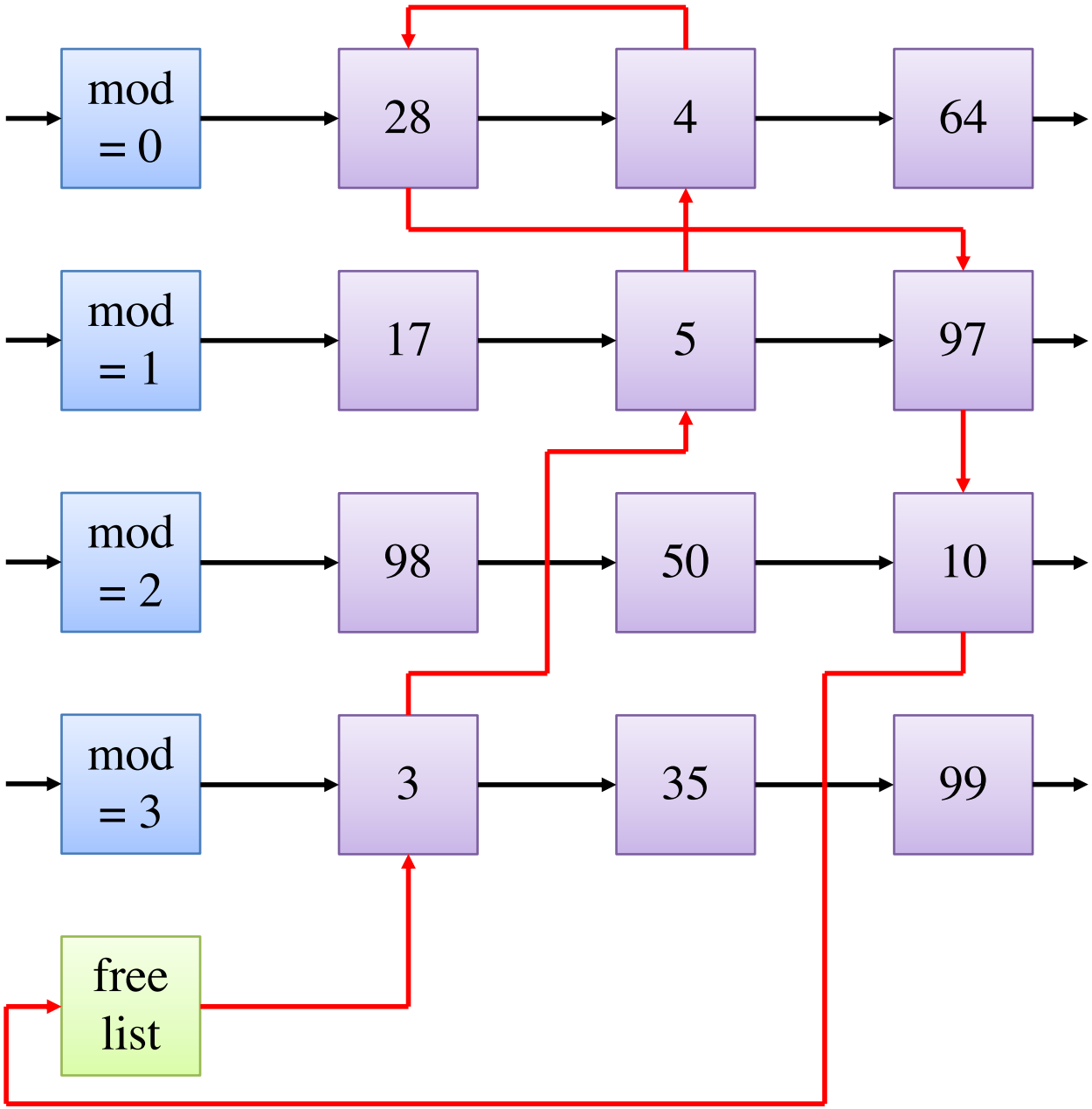
フリーリスト

- フリーなバッファ (ロックされていないバッファ)の管理のため，フリーリストを導入する.
  - ハッシュリストとは別の双方向リスト

```
struct buf_header {
    int blkno;                          /* 論理ブロック番号 */
    struct buf_header *hash_fp;         /* ハッシュの順方向ポインタ */
    struct buf_header *hash_bp;         /* ハッシュの逆方向ポインタ */
    unsigned int stat;                  /* バッファの状態 */
    struct buf_header *free_fp;         /* フリーリスト順方向ポインタ */
    struct buf_header *free_bp;         /* フリーリスト逆方向ポインタ */
    char *cache_data;                   /* データ領域へのポインタ */
};

struct buf_header freelist; /* フリーリストのリストヘッド */
```

ハッシュリストとフリーリスト

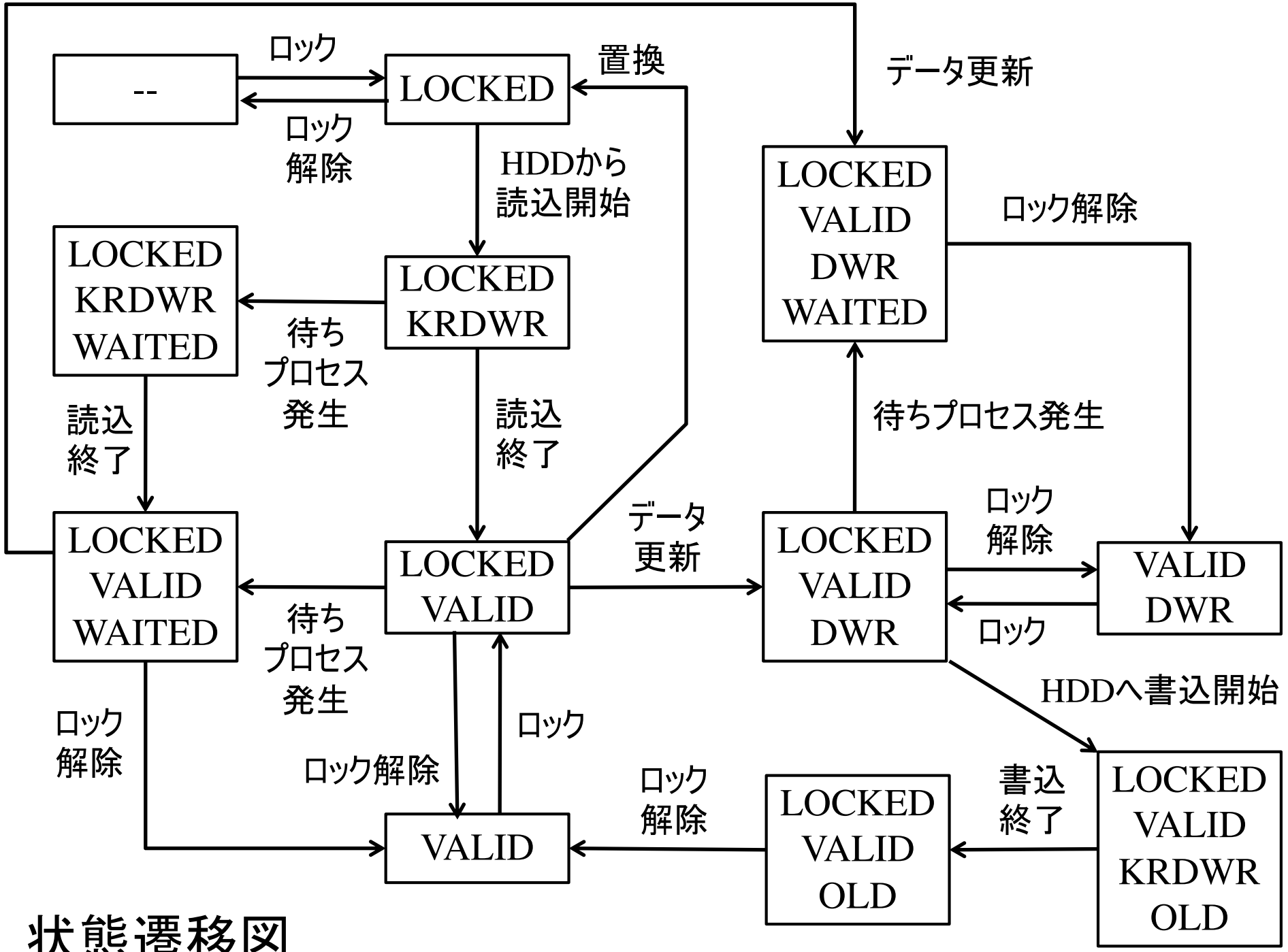


- 逆方向ポインタは省略
- 各バッファは論理ブロックのデータをキャッシュしている
- 3, 5, 4, 28, 97, 10はロックされていない(フリー)
- ハッシュリストとフリーリストは別のポインタを利用している
- フリーなバッファは別の論理ブロック用に置換えることができる

置換方法

- すべてのバッファが使用中のとき，あらたな論理ブロックのデータをキャッシュする場合，どうするか?
  → フリーなバッファを選びデータを置き換える.
- 頻繁に使用されるバッファを置き換えるとシステムの効率が悪くなる.

- LRU (Least Recently Used)
  - 直近において使用頻度の低い順序に並べておき，先頭のものを置き換える.
  - ロックされていたバッファがフリーになった場合は，フリーリストの最後に挿入する. → LRUの順序になる.


--
ロック
LOCKED
ロック
解除
置換
データ更新
HDDから
読込開始
LOCKED
VALID
DWR
WAITED
ロック解除
LOCKED
KRDWR
WAITED
LOCKED
KRDWR
待ち
プロセス
発生
読込
終了
読込
終了
待ちプロセス発生
LOCKED
VALID
WAITED
LOCKED
VALID
データ
更新
LOCKED
VALID
DWR
ロック
解除
VALID
DWR
ロック
待ち
プロセス
発生
HDDへ書込開始
ロック
解除
ロック解除
ロック
ロック
解除
書込
終了
LOCKED
VALID
KRDWR
OLD
VALID
LOCKED
VALID
OLD

状態遷移図