

UNIXシステムプログラミング

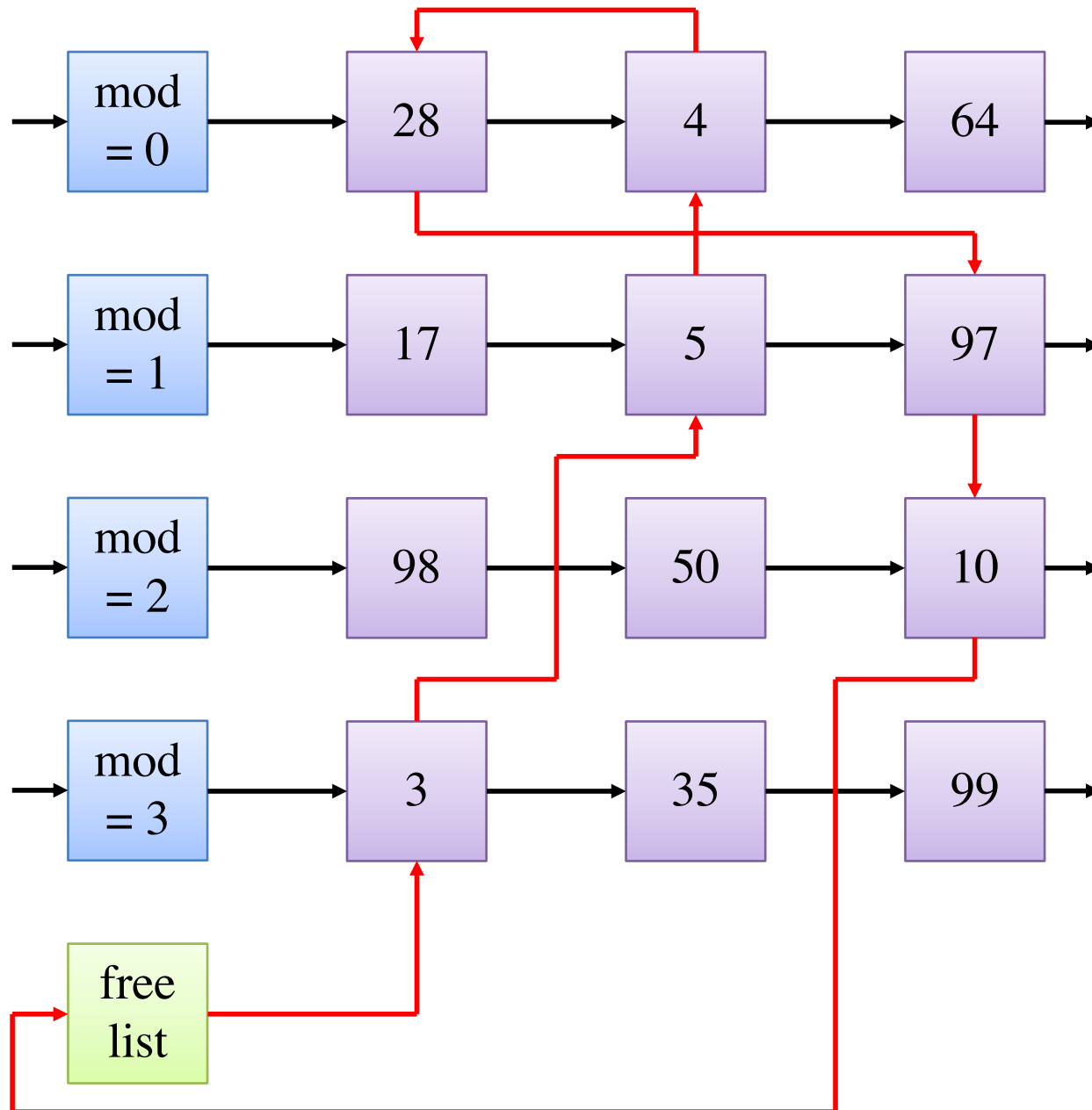
第3回 バッファキャッシュ (2)

2019年10月11日

情報工学科

寺岡文男

ハッシュリストとフリーリスト (再掲)



- 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

バッファ管理アルゴリズム (1)

- 論理ブロック番号を入力の引数としてとり、ロックされたバッファヘッダへのポインタを返す関数

`struct buf_header *getblk(int blkno)` を考える.

- `getblk()`の動作には5つの場合(シナリオ)がある
 - シナリオ1: 最も単純なシナリオ
 - シナリオ2: バッファの置換が発生
 - シナリオ3: HDDへの書戻しが発生
 - シナリオ4: 呼び出したプロセスはスリープする
 - シナリオ5: バッファヘッダはロック状態 → スリープ
- 次ページの擬似コード参照

getblk()の擬似コード

```
struct buf_header *
getblk(int blkno)
{
    while (buffer not found) {
        if (blkno in hash queue) {
            if (buffer locked) {
                /* scenario 5 */
                sleep(event buffer becomes
                           free);
                continue;
            }
            /* scenario 1 */
            make buffer locked;
            remove buffer from free list;
            return pointer to buffer;
        } else {
            if (not buffer on free list) {
                /* scenario 4 */
                sleep(event any buffer
                           becomes free);
                continue;
            }
        }
    }
}
```

```
        remove buffer from
                           free list;
        if (buffer marked for
                delayed write)
            /* scenario 3 */
            asynchronous write
                           buffer to disk;
            continue;
        }
        /* scenario 2 */
        remove buffer from
                           old hash queue;
        put buffer onto new
                           hash queue;
        return pointer to buffer;
    } /* end of "else"
  } /* end of while()
}
```

バッファ管理アルゴリズム (1)

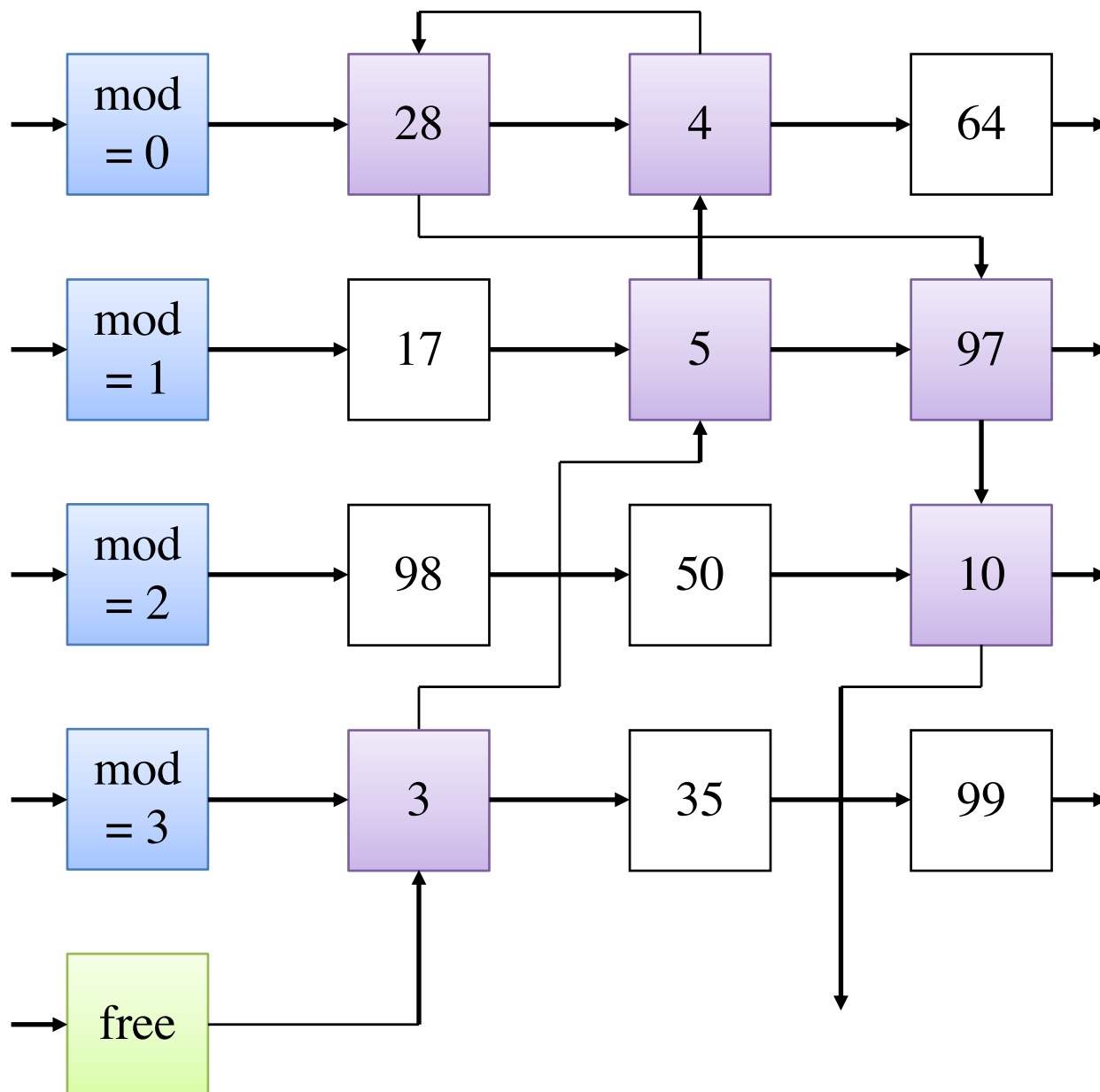
- シナリオ1: 最も単純なシナリオ

- 検索するバッファヘッダがハッシュリストにあり, かつフリーである.



- 見つかったバッファヘッダをフリーリストから削除. ロック状態にしてそのポインタを返す.

シナリオ1: getblk(10): 初期状態



- 各バッファの状態を考えてみよう

- LOCKED
- VALID
- DWR
- KRDWR
- WAITED
- OLD

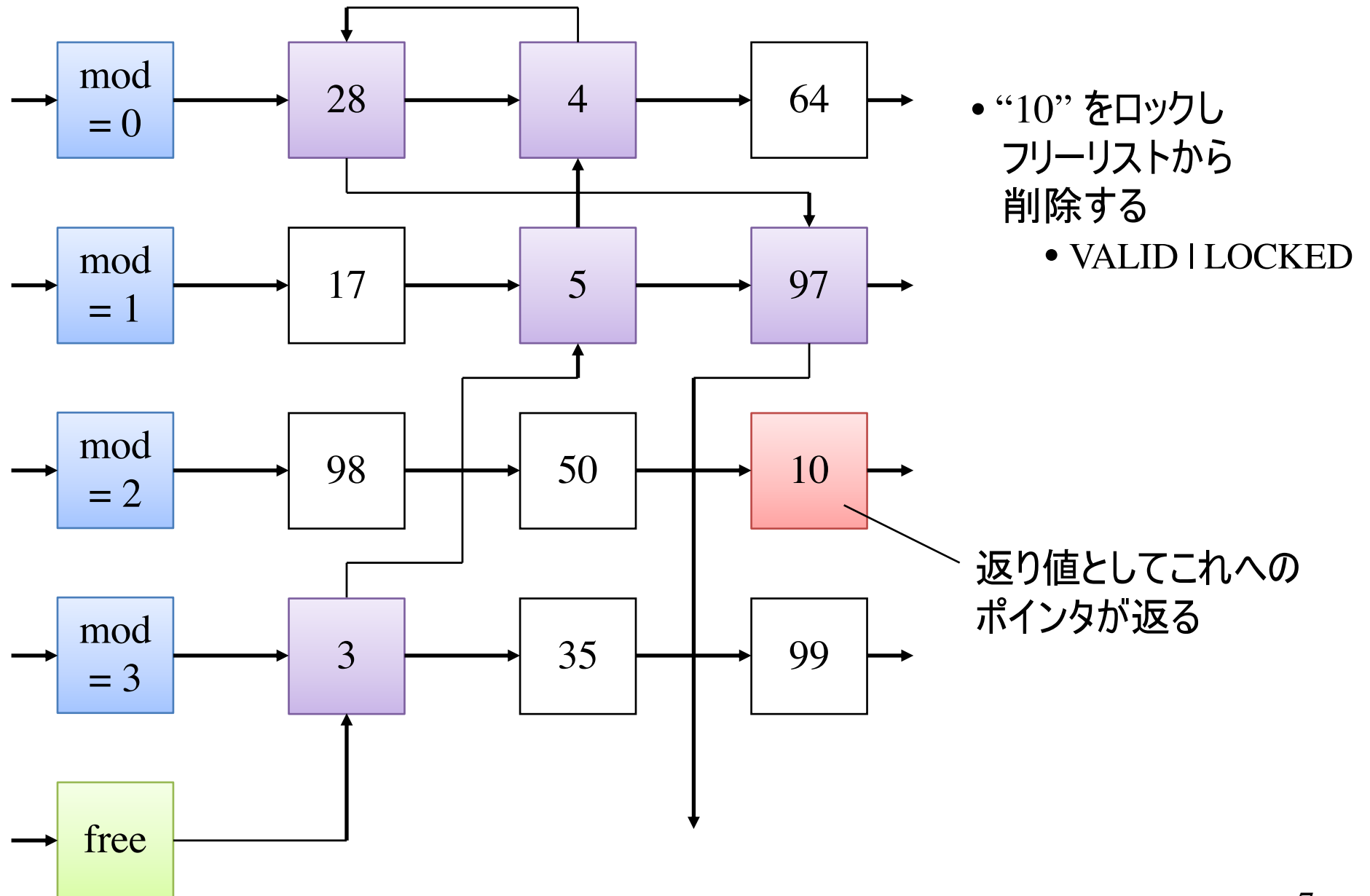


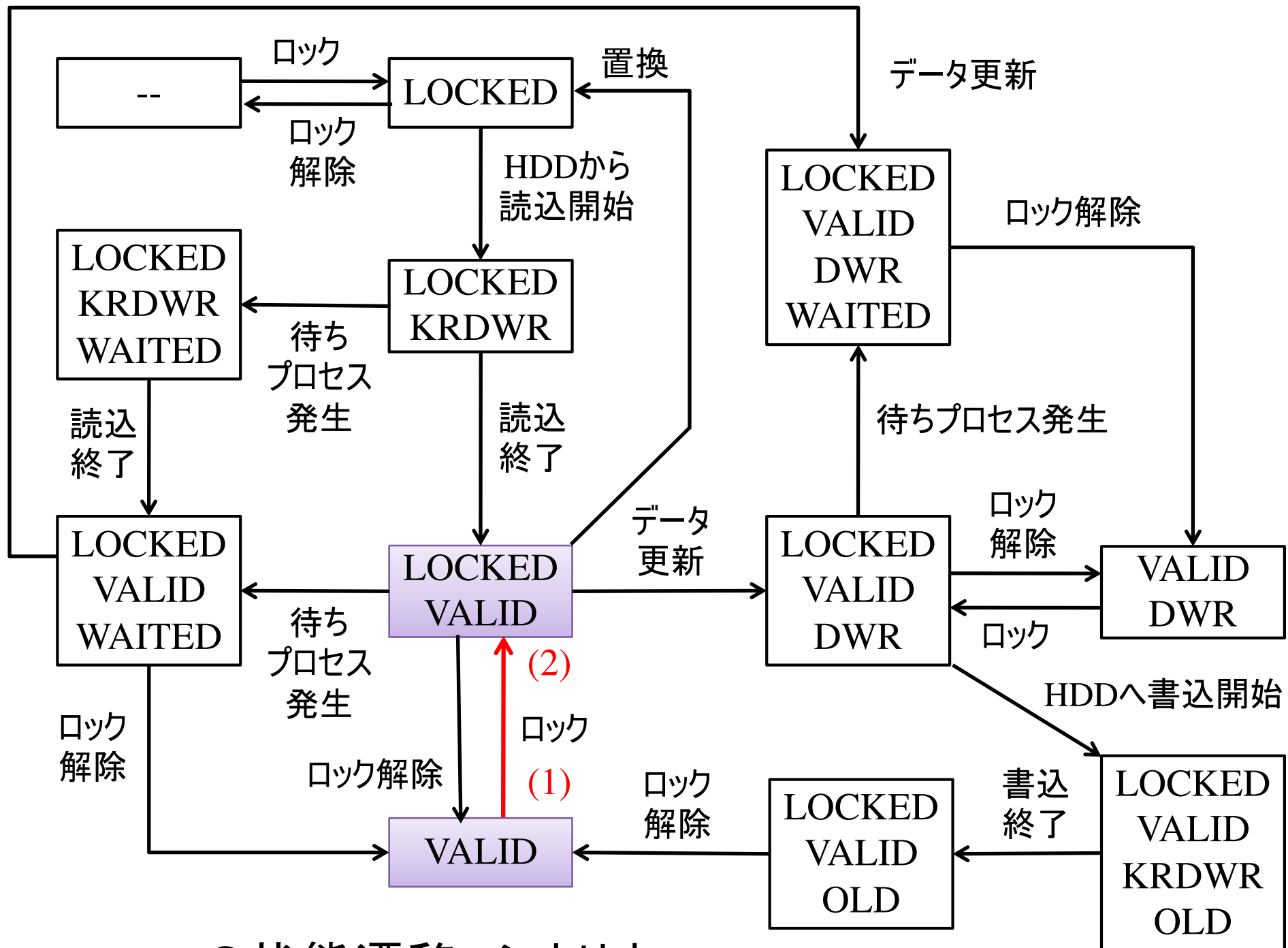
- VALID

11

- VALID | LOCKED

シナリオ1: getblk(10): 実行後





bufhead-10の状態遷移 (シナリオ1)

バッファの解放

- `getblk()`は要求したプロセスにロックされたバッファヘッダへのポインタを返す.
 - 要求したプロセスはこのバッファを利用 (read and/or write)
 - 他のプロセスはこのバッファを利用できない
- プロセスは利用終了後 `brelease()`でバッファを解放する.
 1. フリーバッファを待っているプロセスをwakeup
 2. 解放するバッファを待っているプロセスをwakeup
 3. 解放するバッファヘッダをフリーリストに挿入
 4. バッファヘッダをunlocked 状態にする
 - 相互排除が必要 (詳しくはOSの講義で)

brelease()の擬似コード

シナリオ4に対応

```
void brelease(struct buf_header *buffer)
{
    wakeup all procs; event, waiting for any buffer to become free;
    wakeup all procs; event, waiting for this buffer to become free;
    raise processor execution level to block interrupts;
    if (buffer contents valid and buffer not old)
        enqueue buffer at end of free list;
    else
        enqueue buffer at beginning of free list;
    lower processor execution level to allow interrupts;

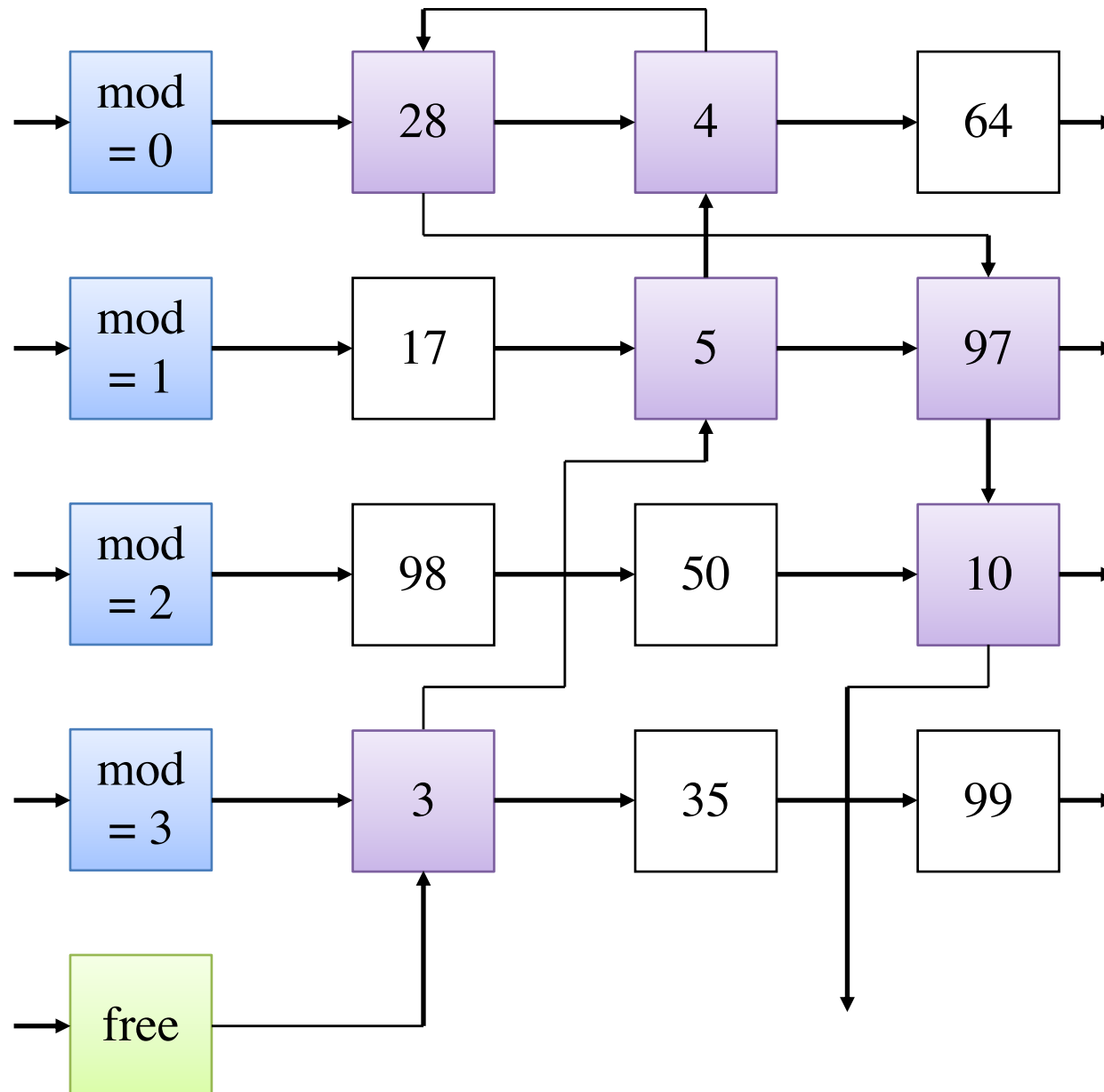
    unlock(buffer);
}
```

シナリオ5に対応

シナリオ3に対応

- “raise processor execution level” から “lower processor execution level” までは
相互排除の区間 (= **critical section**)
 - あるプロセスがこの区間を実行中に別のプロセスがこの区間を実行しない
ようにする

シナリオ1のあと, brelse(10)を実行



- 逆方向ポインタは省略

- “10” はフリーリストの末尾に挿入 (→ LRUを実現)



- VALID



- VALID | LOCKED

バッファ管理アルゴリズム (2)

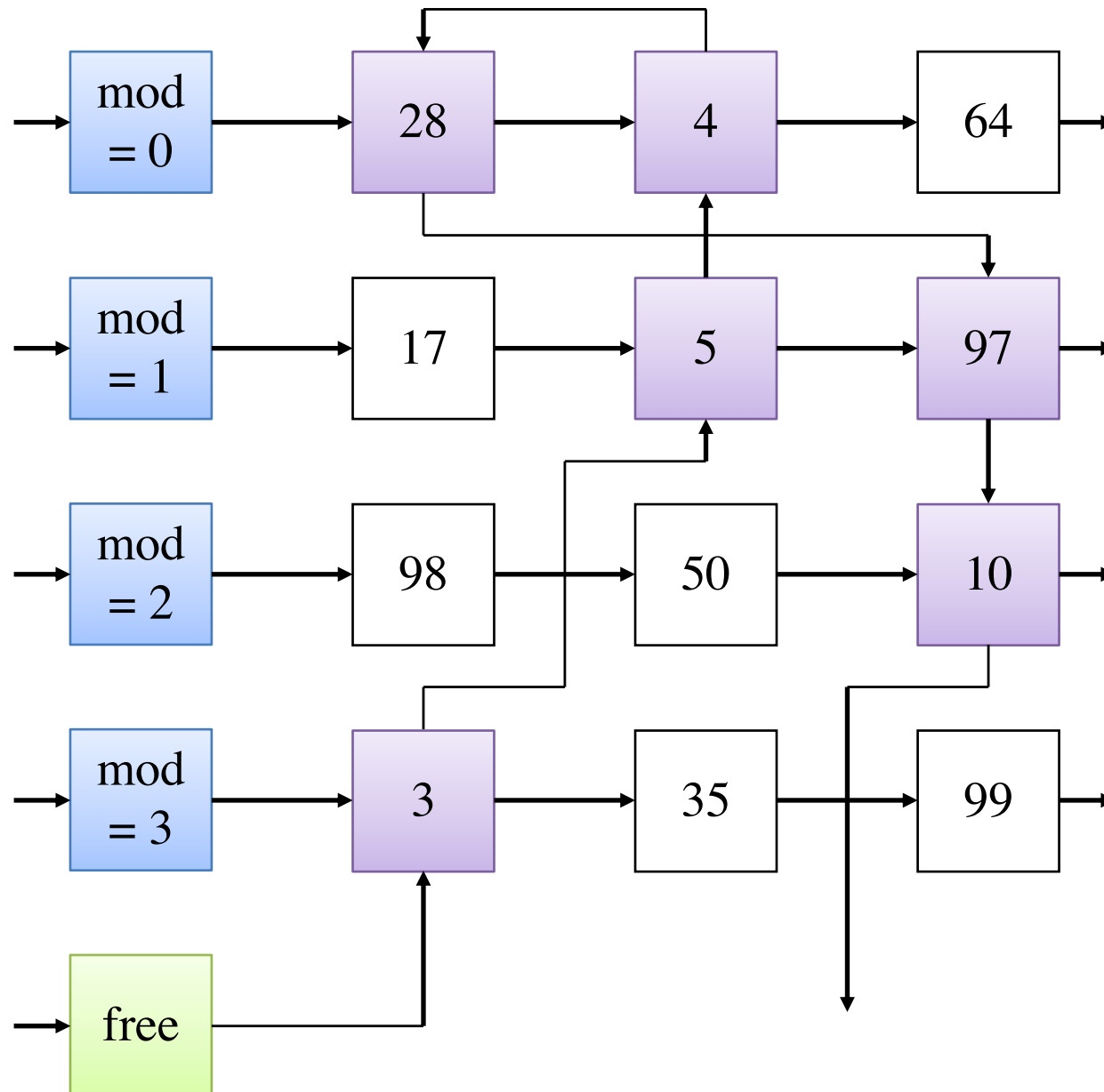
- シナリオ2: バッファの置換が発生

- 検索するバッファヘッダがハッシュリストにない.

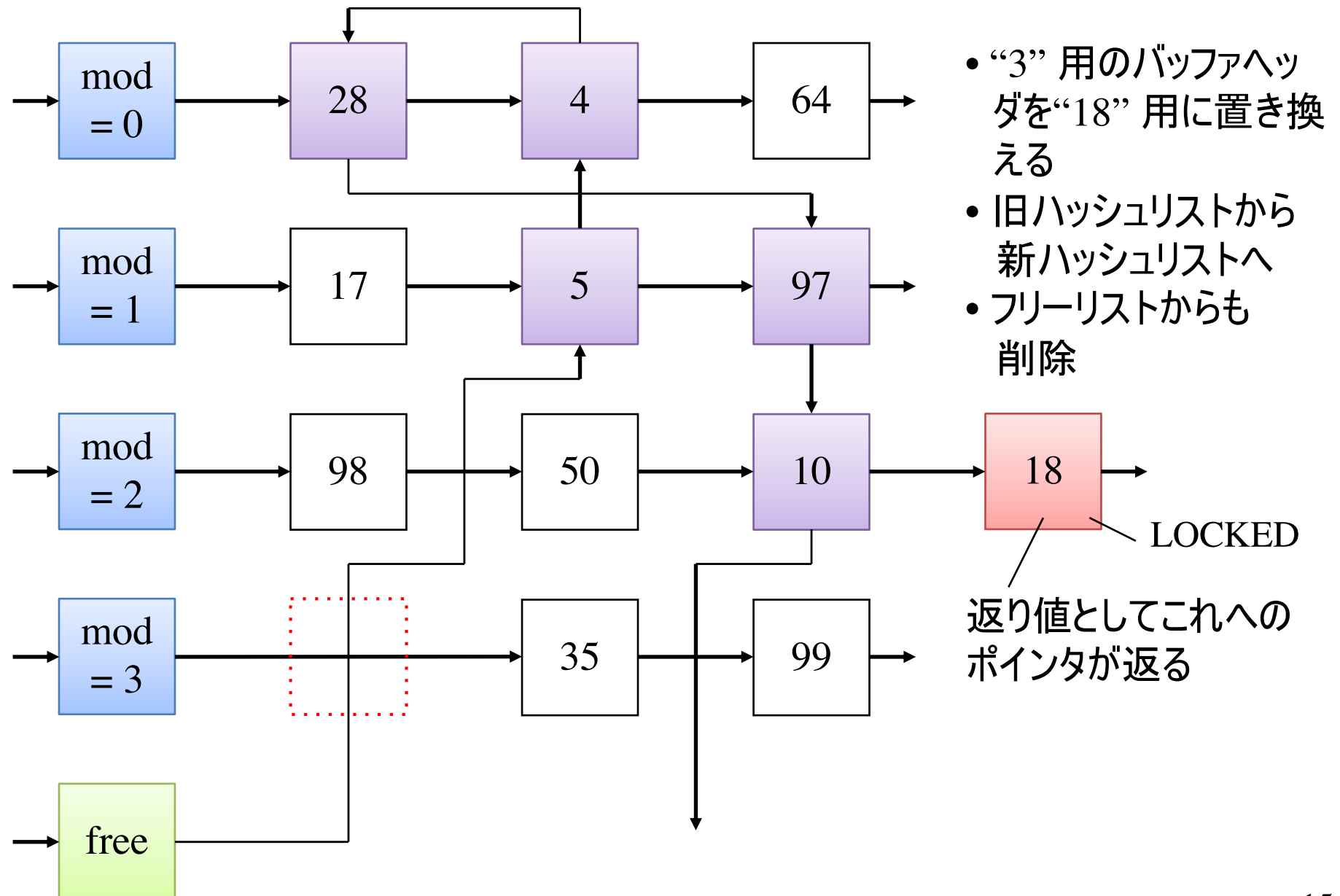


- フリーリストの先頭のバッファヘッダの内容を置き換える.
 - このバッファヘッダをフリーリストから削除してロックし、新しいハッシュリストへ.
 - このバッファヘッダのポインタを返す.

シナリオ2: getblk(18): 初期状態

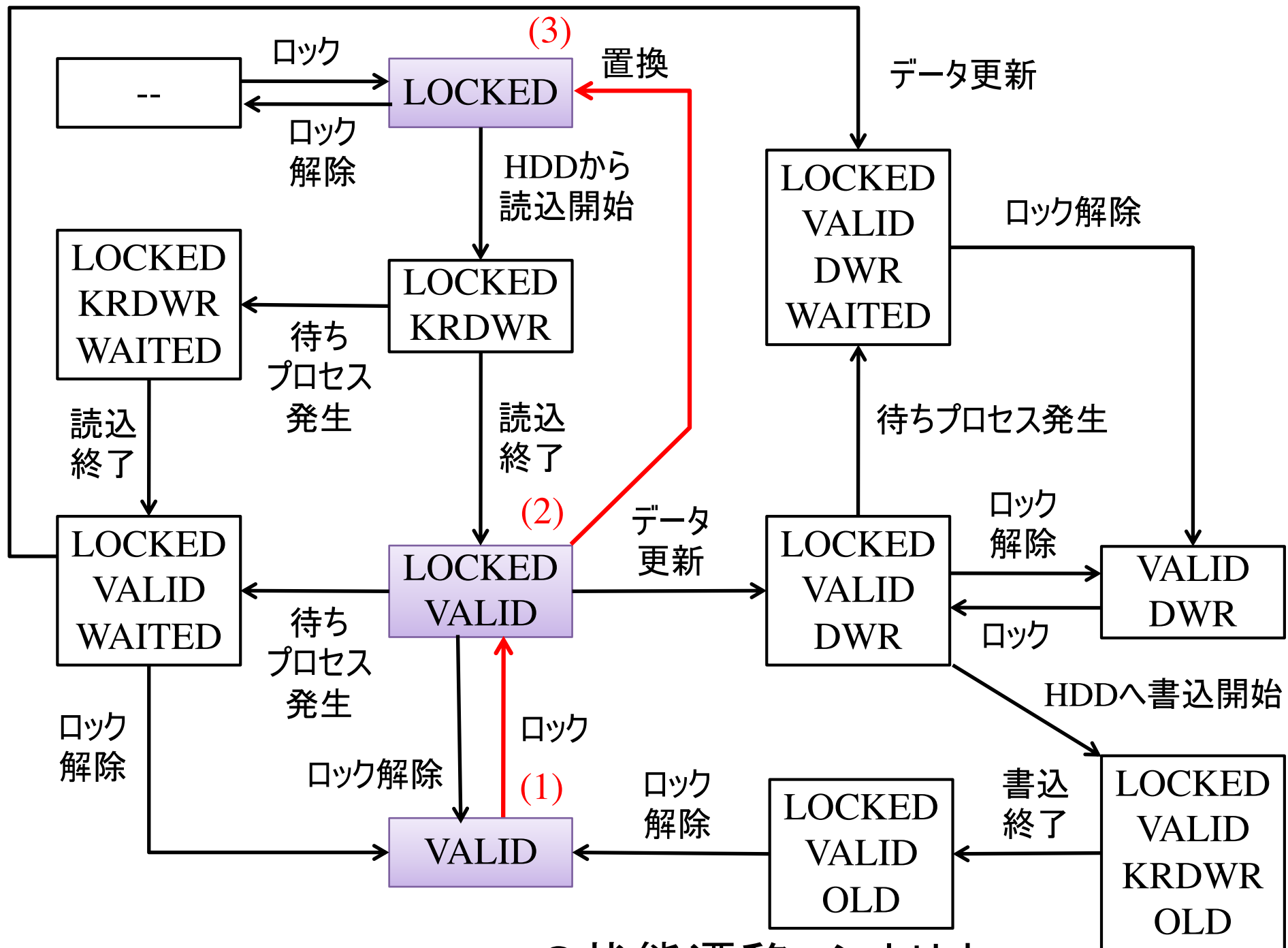


シナリオ2: getblk(18): 実行後

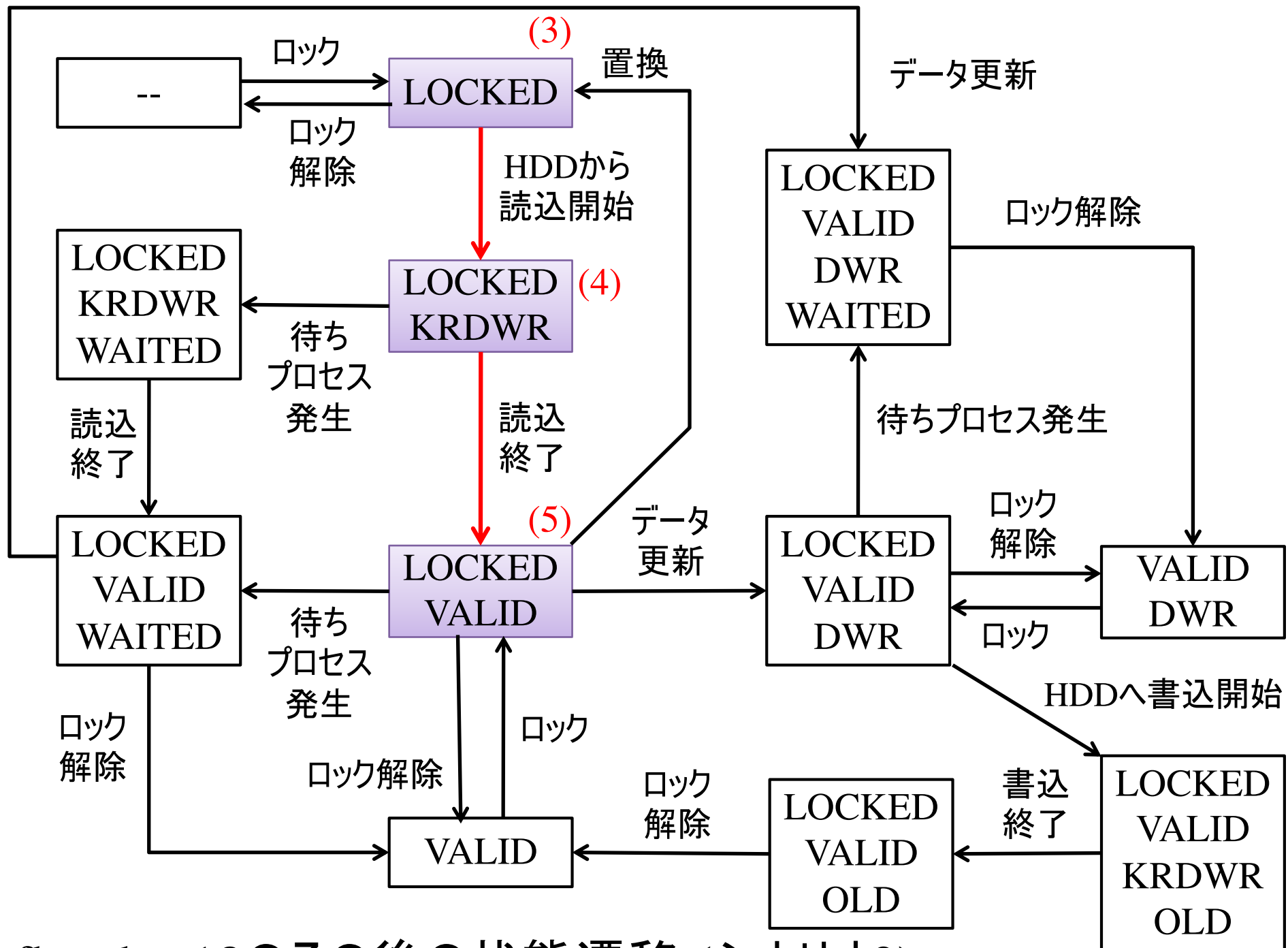


bufheader-18のその後

- もとはblk 3のデータをキャッシュ (bufheader-3)
- blk 18のデータをキャッシュするために使い回し(bufheader-18)
 - データ格納領域にはまだblk 3のデータがある
 - VALID ではなくなる
- OSはblk 18のデータ読み込みを開始する
 - KRDWR になる
- やがてデータ読み込みが完了
 - KRDWR ではなくなる
 - VALID になる

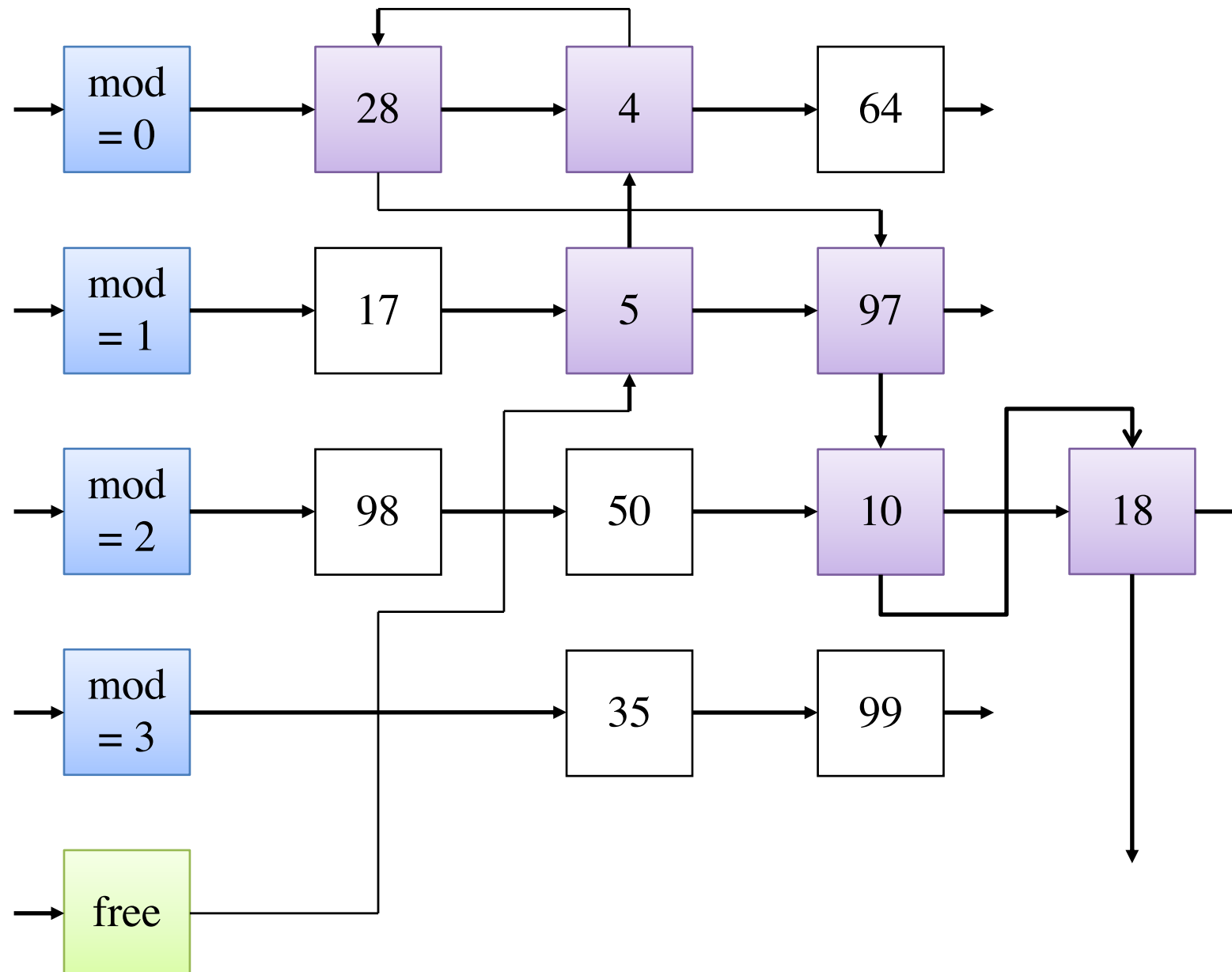


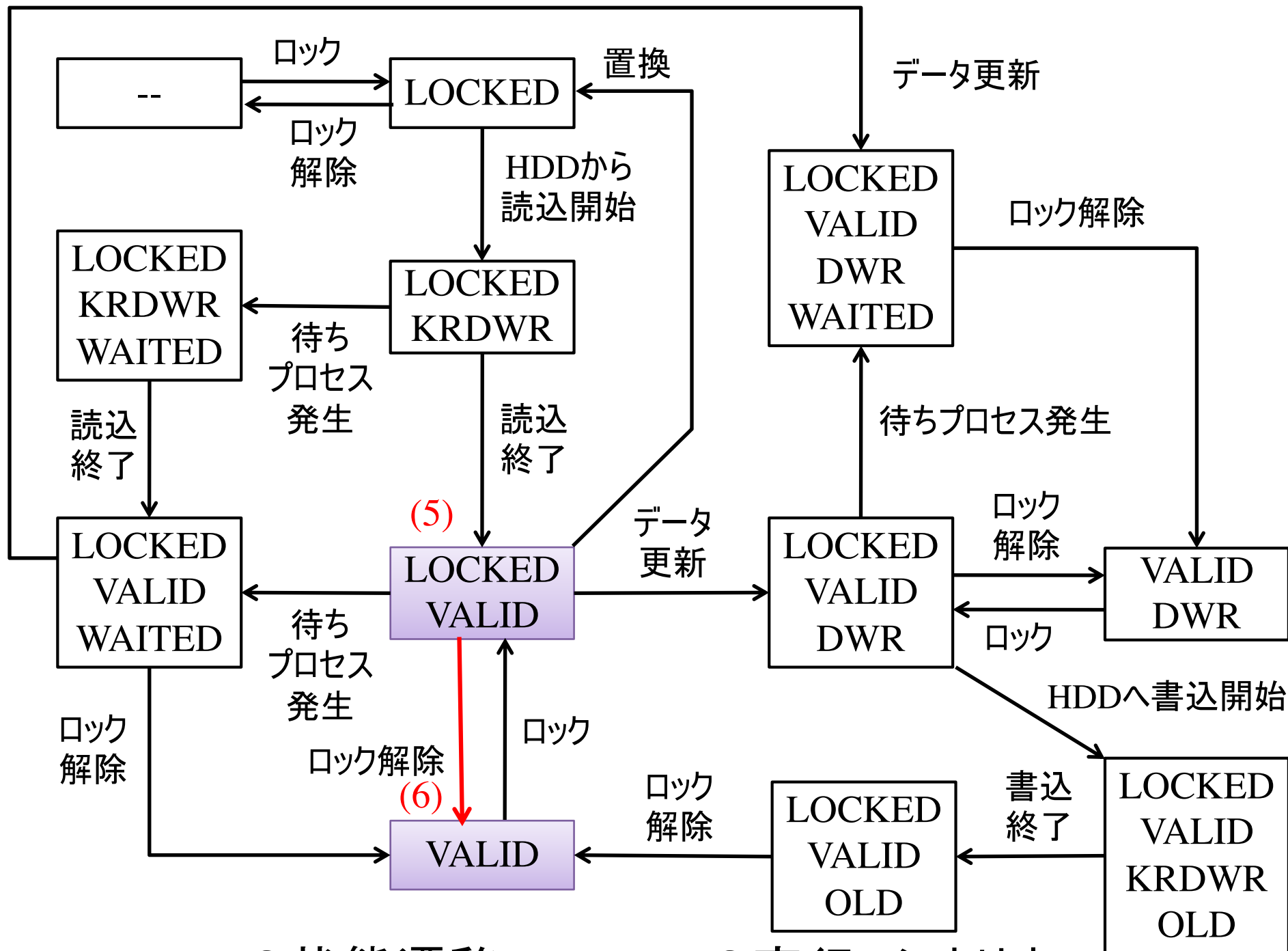
bufheader-3(bufheader-18)の状態遷移 (シナリオ2)



bufheader-18のその後の状態遷移 (シナリオ2)



シナリオ2のあと, brelse(18)を実行



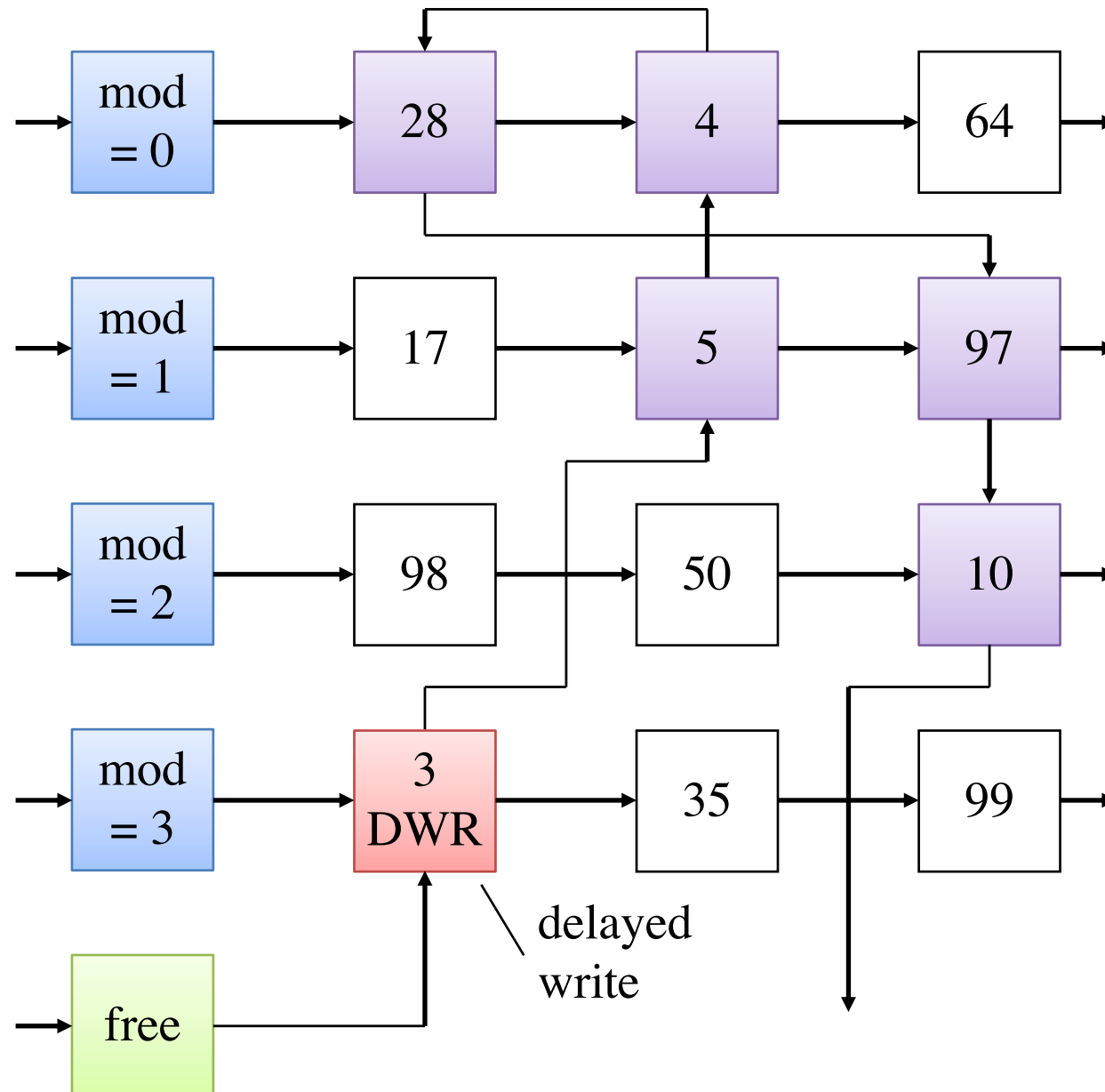


bufheader-18の状態遷移: brelse()の実行 (シナリオ2)

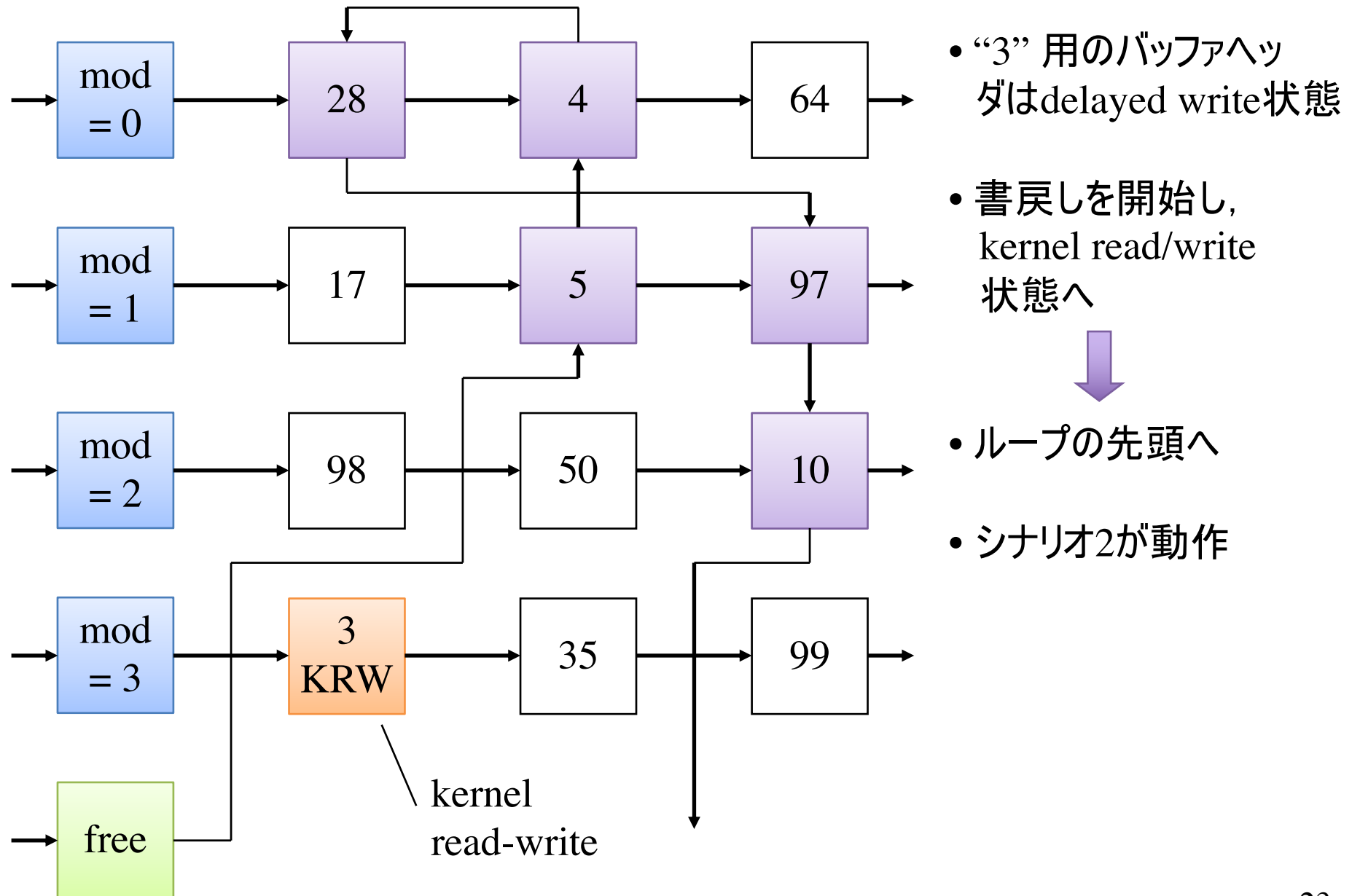
バッファ管理アルゴリズム (3)

- シナリオ3: HDDへの書戻しが発生
 - 検索するバッファヘッダがハッシュリストになく、フリーリストの先頭に遅延書込み(STAT_DWR)ビットがセットされている.
 - STAT_DWR: データが書き換えられ、HDDへの書き戻しを待っている状態
- 
- HDDへの書戻しを起動し、フリーリストの次のエンTRIESを調べる.
- 
- その後、シナリオ2が実行される.

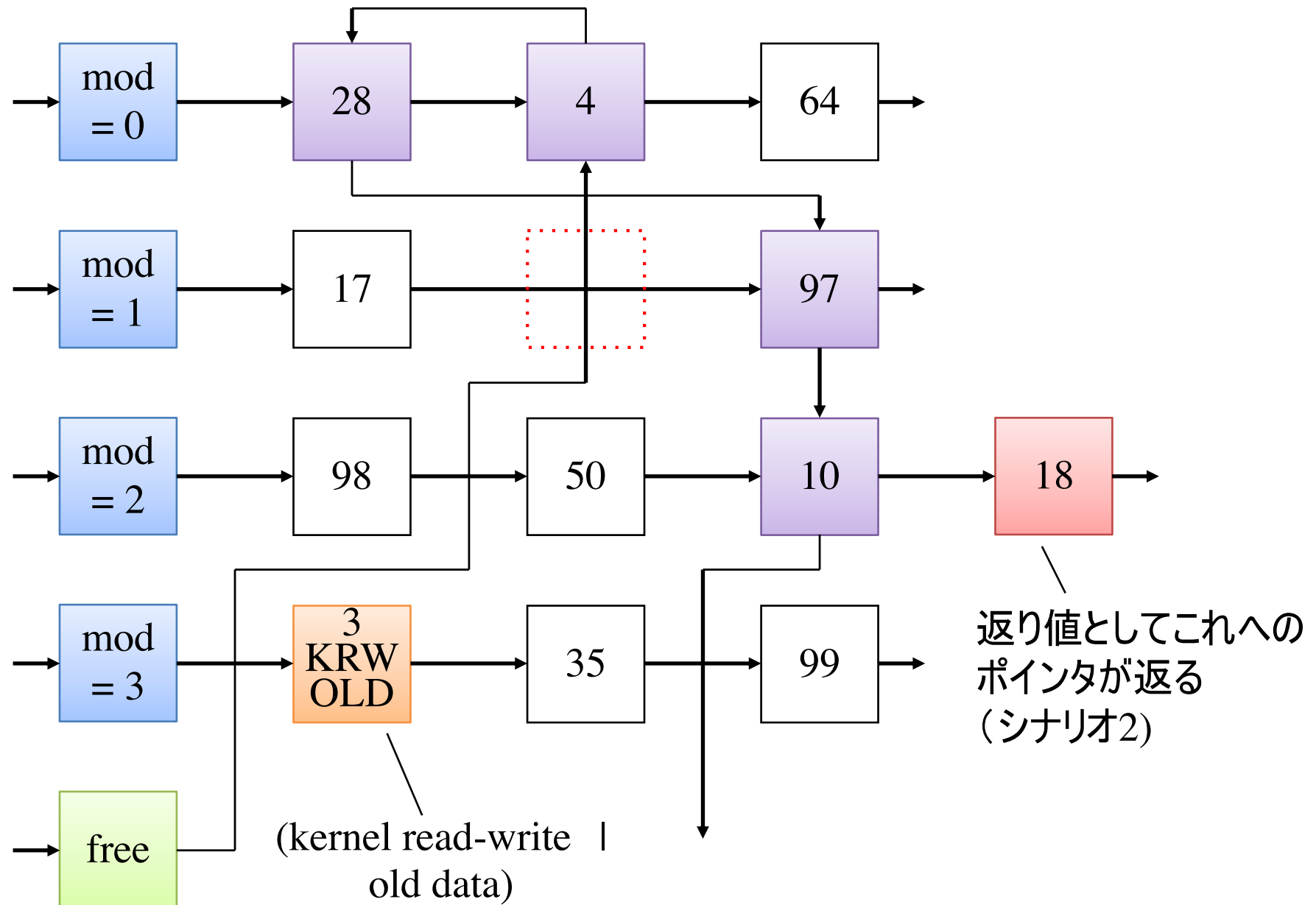
シナリオ3: getblk(18): 初期状態

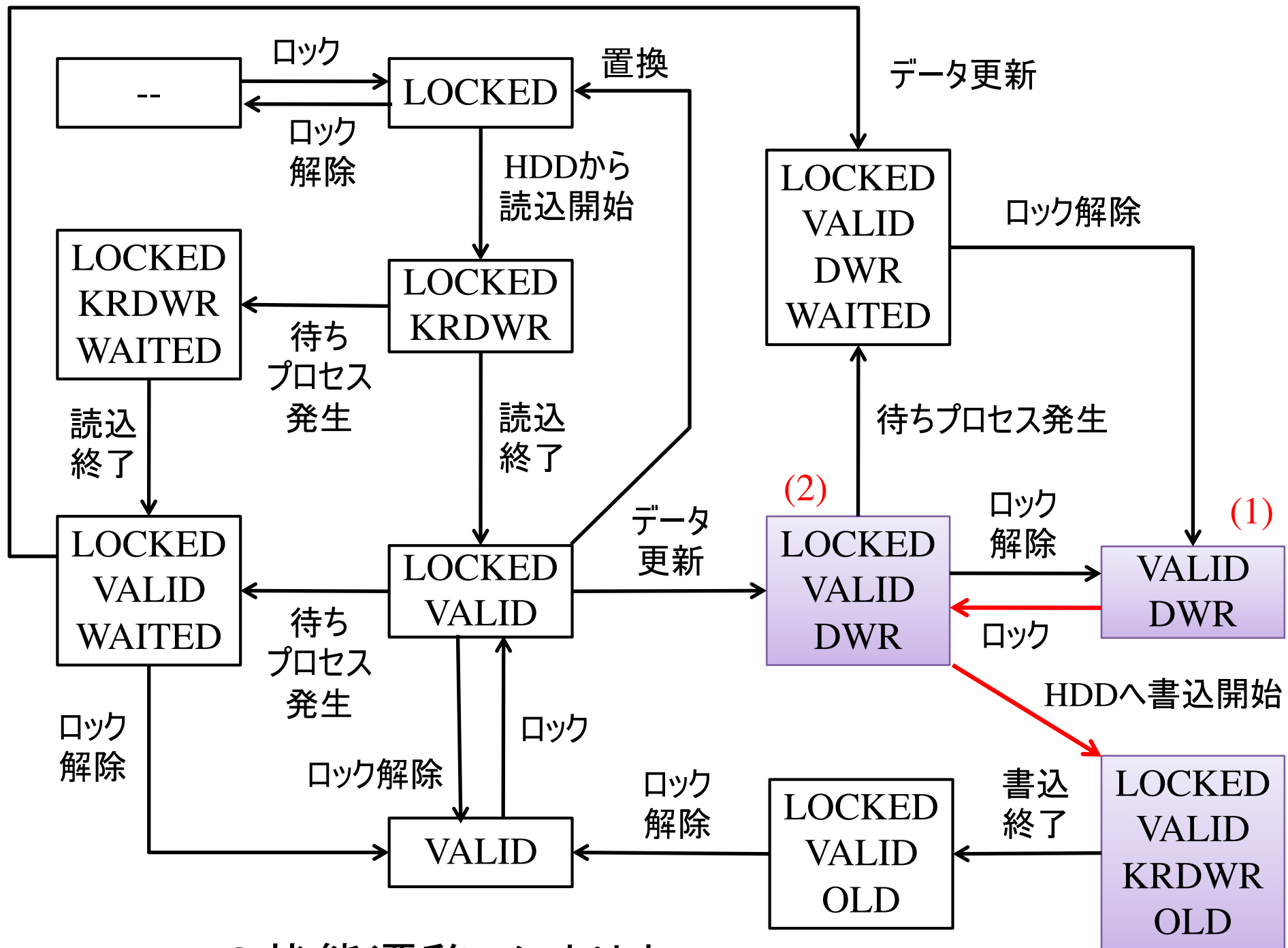


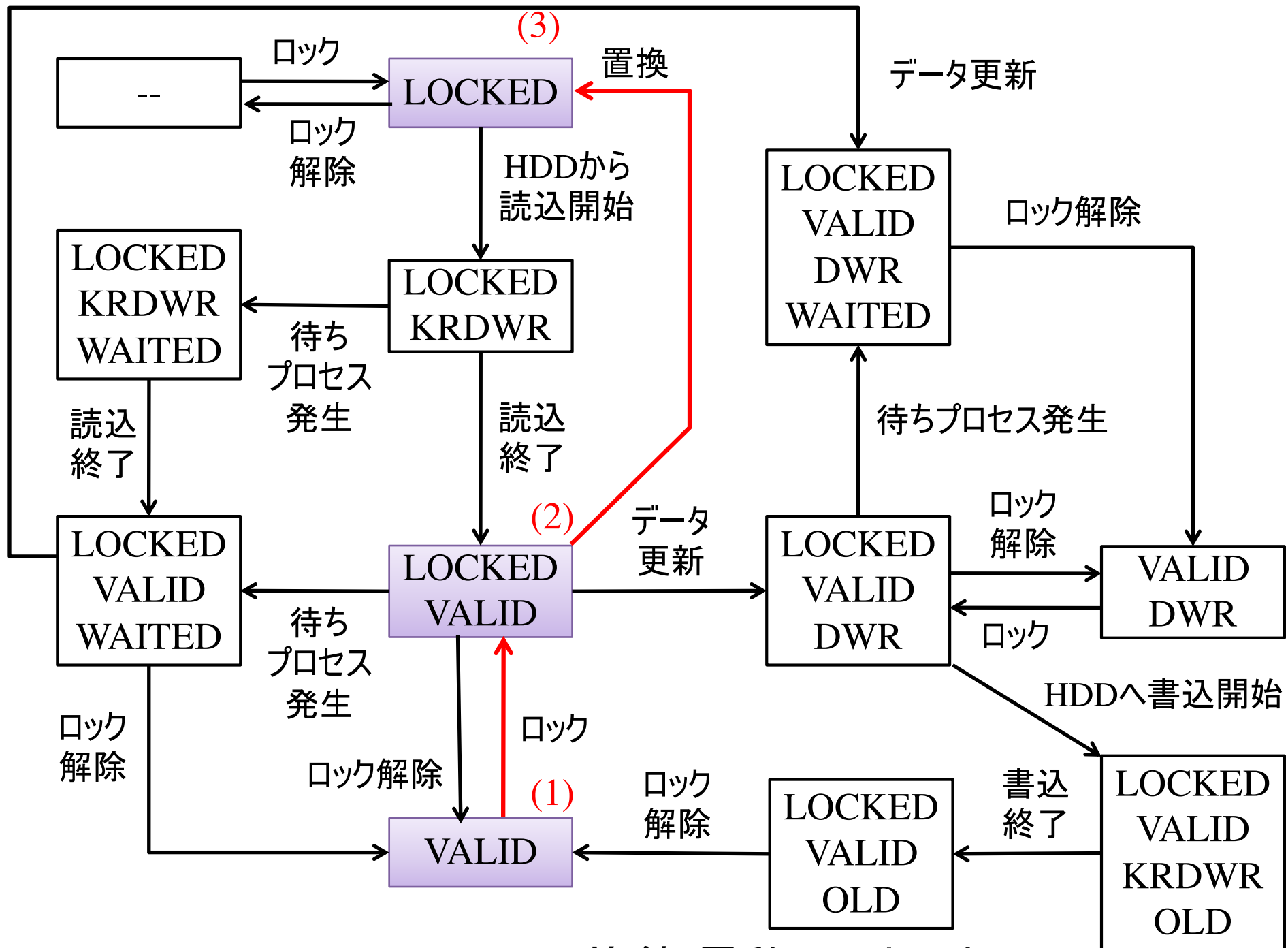
シナリオ3: getblk(18): 中間状態1



シナリオ3: getblk(18): 中間状態2

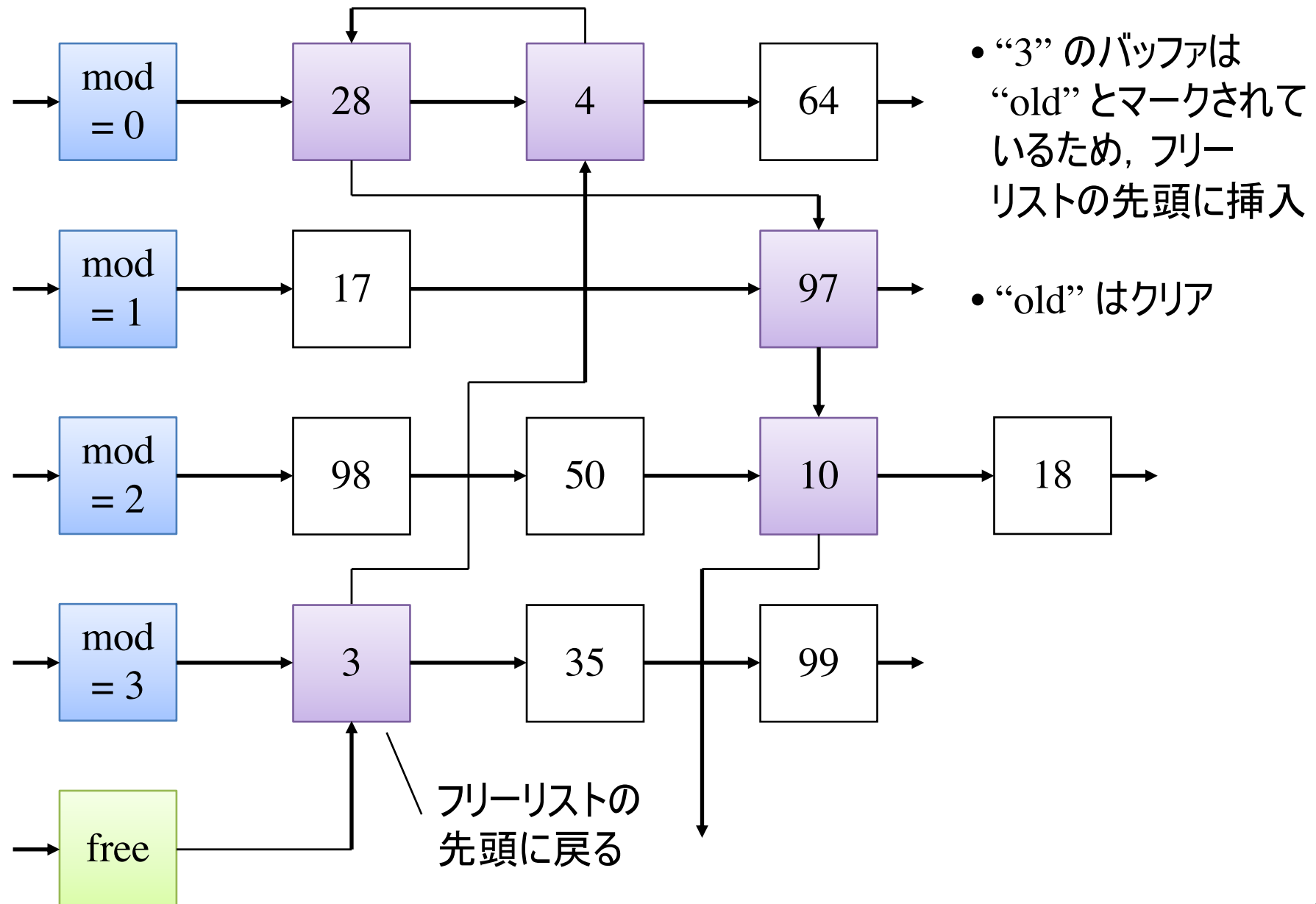






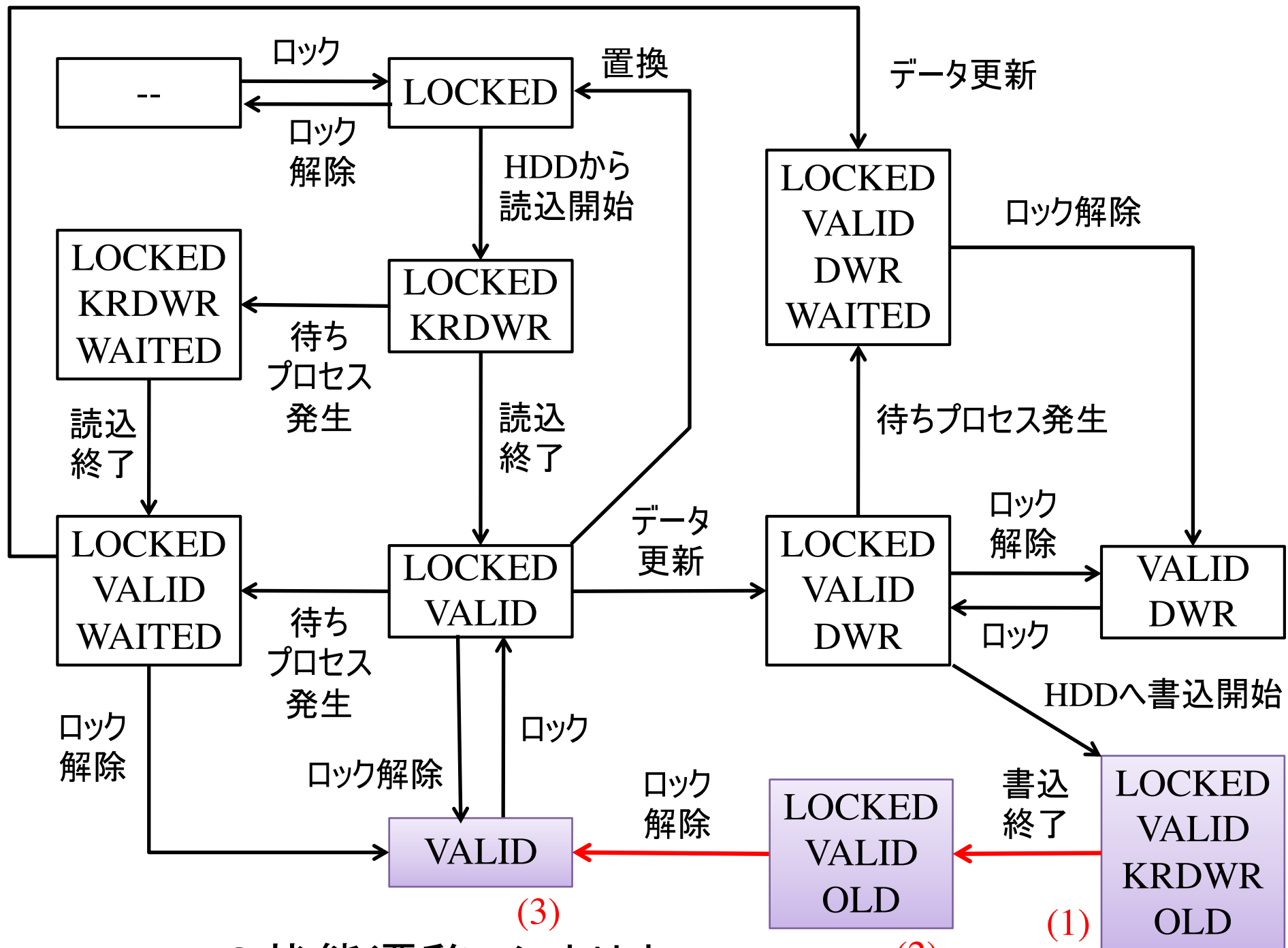
bufheader-5(bufheader-18)の状態遷移 (シナリオ3)

シナリオ3: getblk(18): 最終状態



bufheader-3のその後

- もとはdelayed write状態
 - プロセスがデータを書き換えたがHDDに書き戻していない
- OSはデータをHDDに書き戻す動作を開始
 - LOCKED になる → フリーリストから外れる
 - KRDWR になる
 - OLD もセットする → フリーリストの先頭に挿入するため
- HDDへの書き戻し終了
 - KRDWR ではなくなる
- フリーリストの先頭に挿入される
 - LRUの順序を守るため
 - OLD と LOCKED がリセットされる

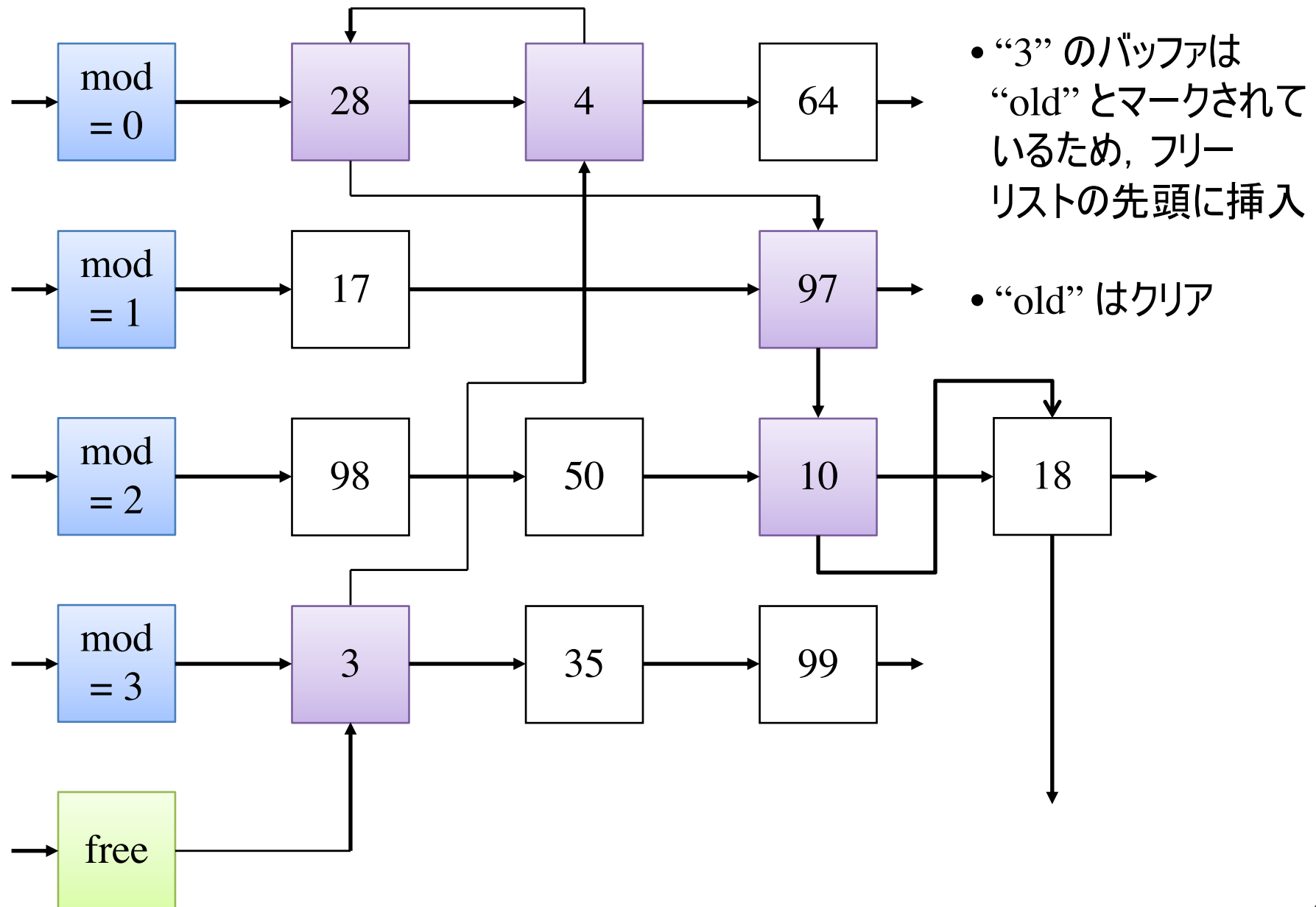


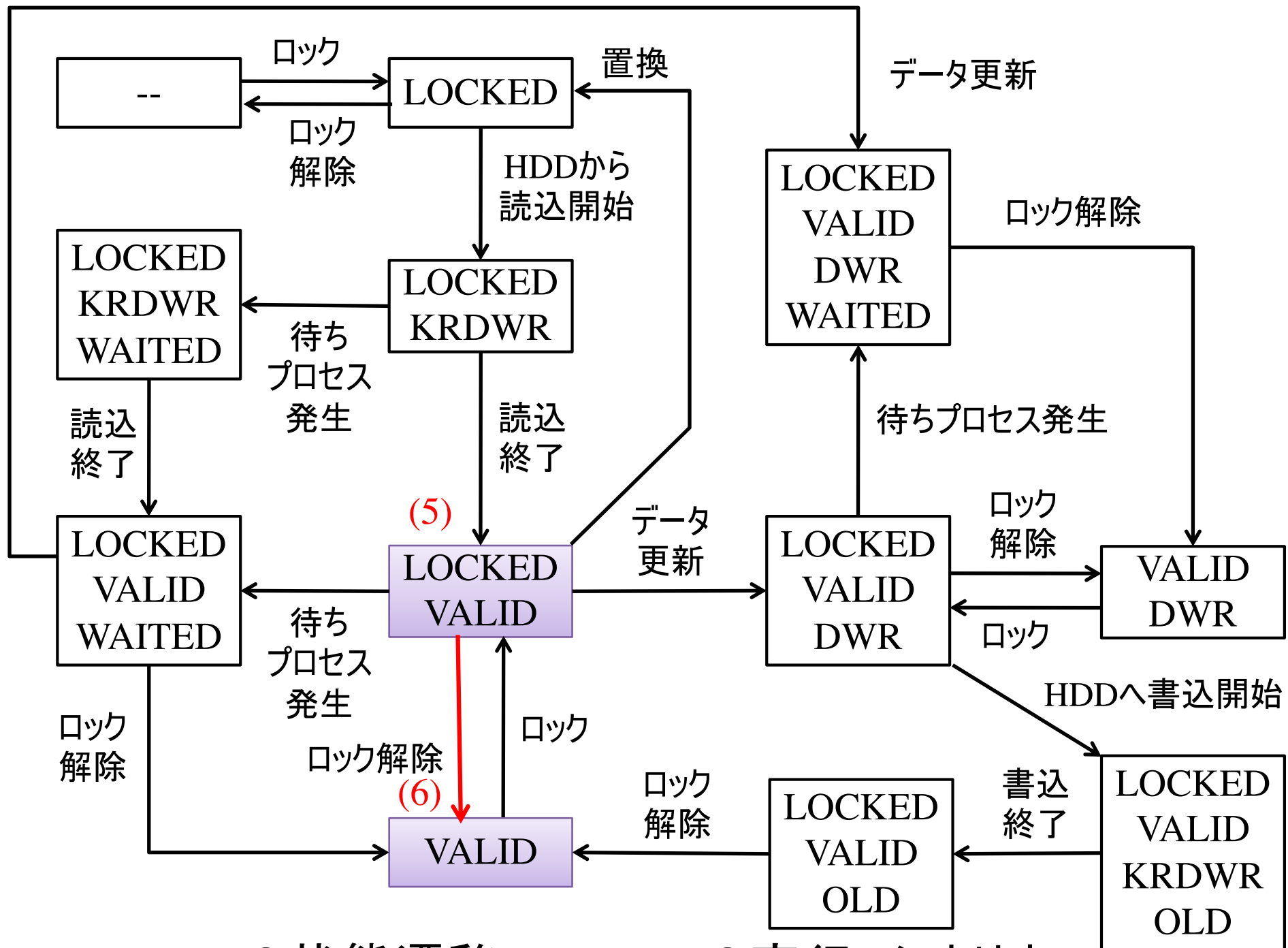
bufheader-3の状態遷移 (シナリオ3): 2

bufheader-18のその後

- もとはblk 5のデータをキャッシュ (bufheader-5)
- blk 18のデータをキャッシュするために使い回し(bufheader-18)
 - データ格納領域にはまだblk 5のデータがある
 - VALID ではなくなる
- OSはblk 18のデータ読み込みを開始する
 - KRDWR になる
- やがてデータ読み込みが完了
 - KRDWR ではなくなる
 - VALID になる


シナリオ3のあと, brelse(18)を実行



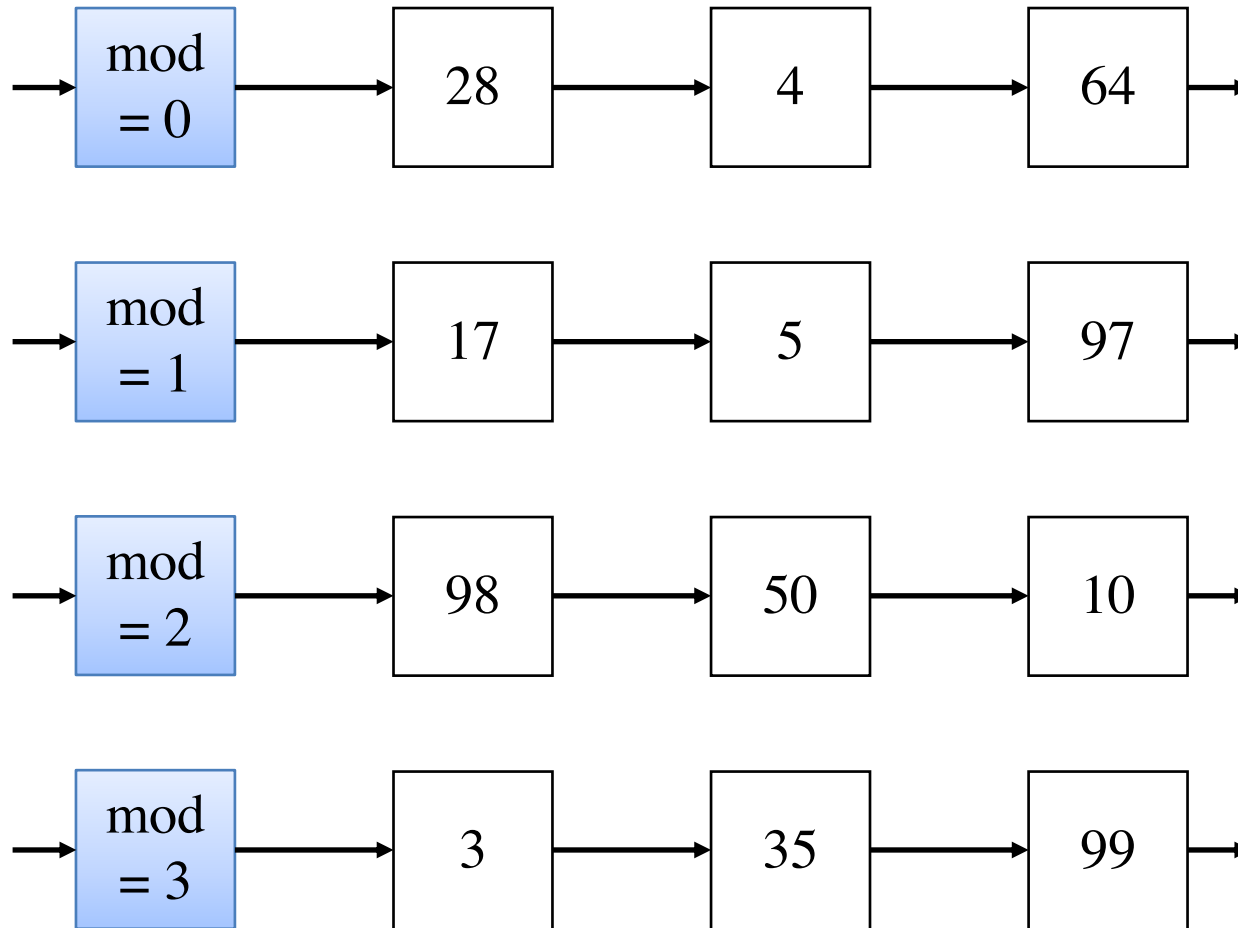


bufheader-18の状態遷移: brelse()の実行 (シナリオ3)

バッファ管理アルゴリズム (4)

- シナリオ4: 呼び出したプロセスはスリープする
 - 検索するバッファヘッダはハッシュリストになく, フリーリストも空である.
- 
- バッファがフリーされるまでsleepする.
 - sleepに関してはOSの講義参照
 - wakeupしたらループの最初に戻る. (Q: なぜか?)


シナリオ4: getblk(18)



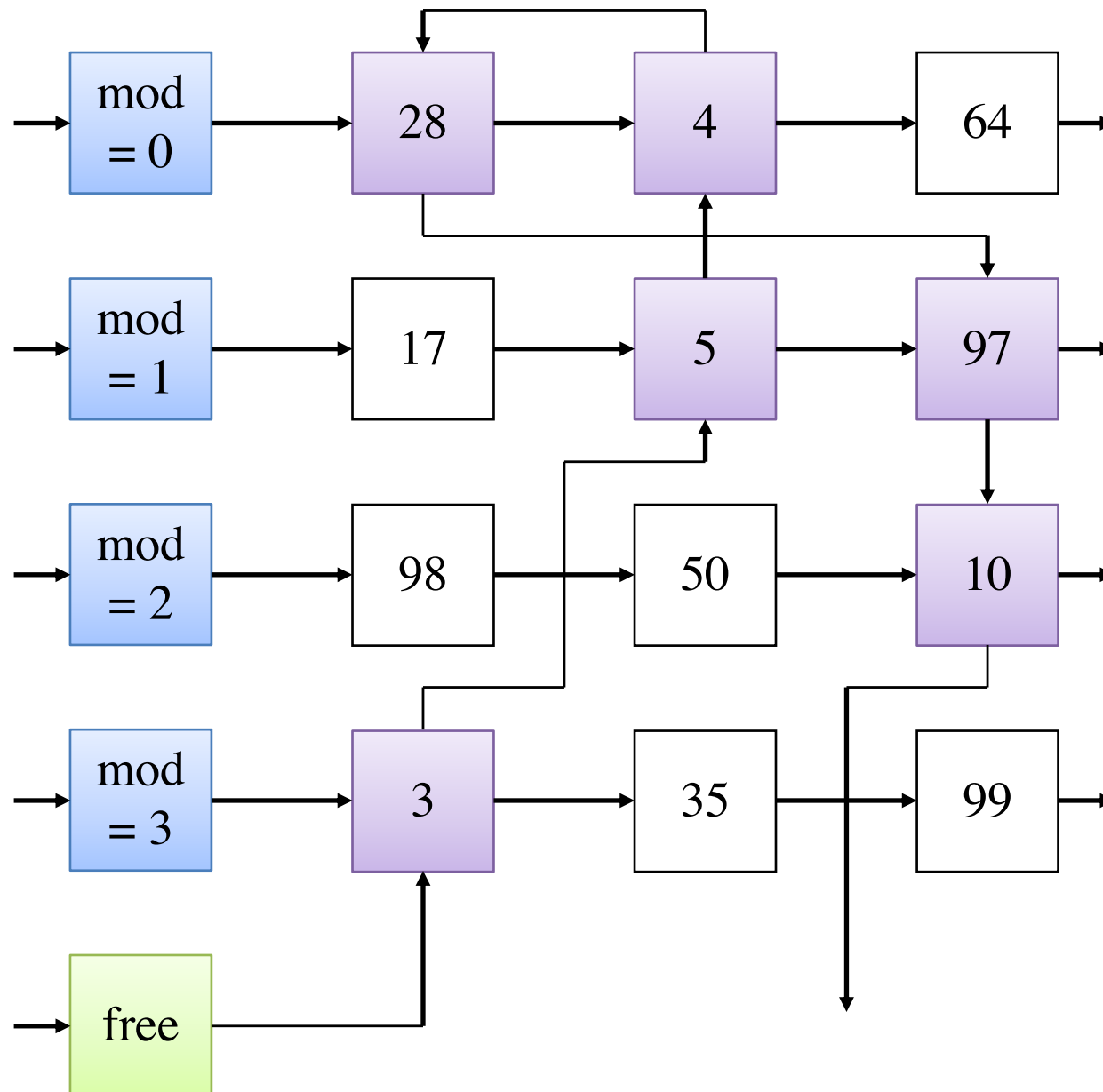
- フリーリストは空
- 呼び出したプロセスはバッファがフリーになるのを待つsleep
- wakeupしたらループの先頭へ

free

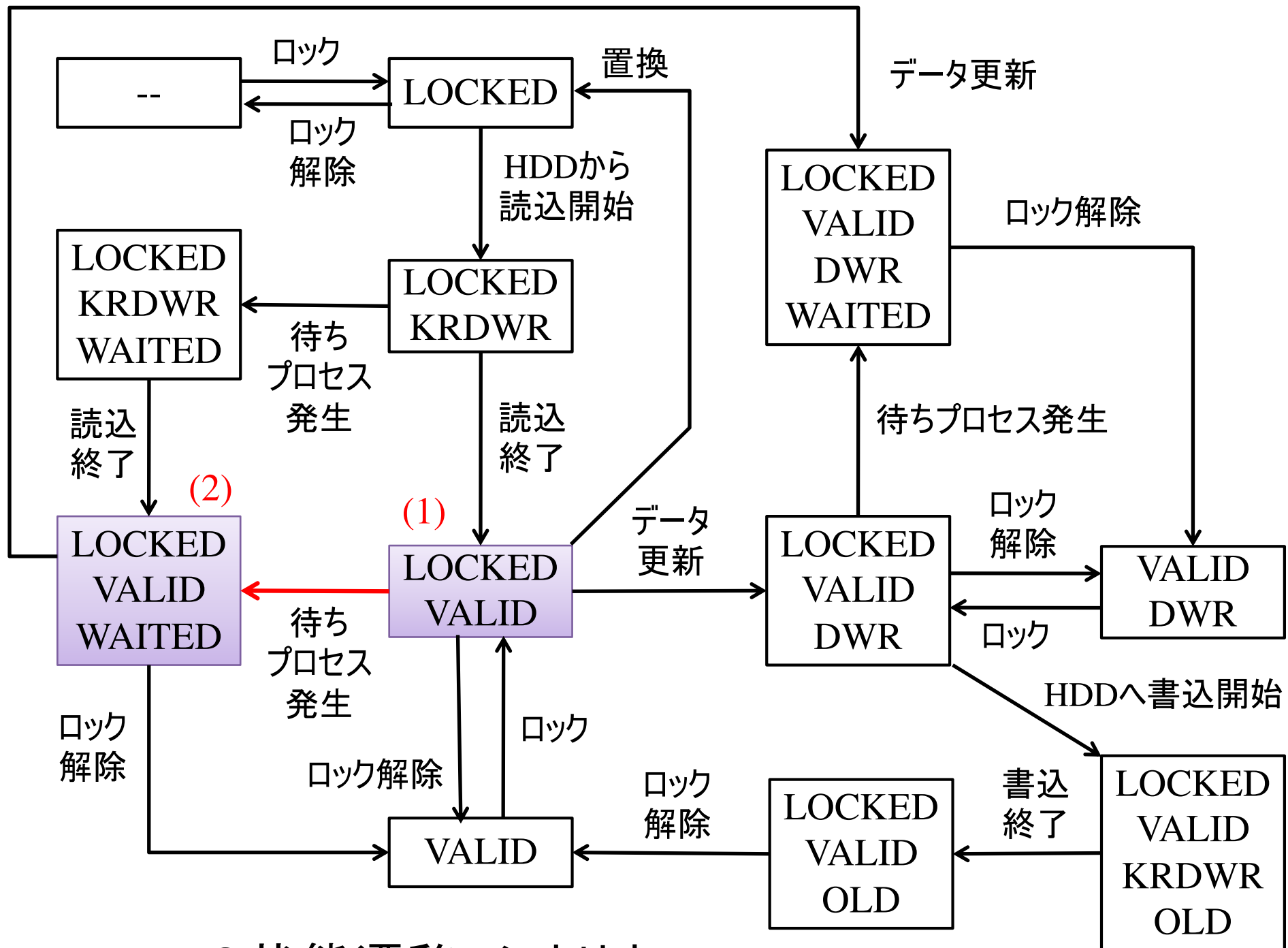
バッファ管理アルゴリズム (5)

- シナリオ5: バッファヘッダはロック状態 → スリープ
 - 検索するバッファヘッダはハッシュリストにあるが、ロックされている.
- 
- そのバッファヘッダがフリーになるまでスリープ.
 - wakeupしたらループの最初に戻る. (Q: なぜか?)

シナリオ5: getblk(64)



- “64” はロック状態
- 呼び出したプロセスは “64” がフリーになるのを待つsleep
- wakeupしたらループの先頭へ



bufhead-64の状態遷移 (シナリオ5)

課題2: ステップ1

- `getblk()`, `brelse()`およびこれらで使われている関数をC言語で記述しなさい.
 - 挿入, 削除の関数については, 先週作成済みのはず
 - `sleep`, `wakeup`, `raise cpu execution level`, etc. の部分に関してはテキストの pp.36参照.
 - “`gcc -c getblk.c`” を実行し, 文法エラーをチェックする.
- ステップ2のプログラミングは, 次週の文字列処理の講義を参考にするとよい.

課題2: ステップ2

- `getblk()`, `brelease()`の動作をインタラクティブに確認するための `main()` および必要な関数をC言語で記述しなさい。
 - 詳細はテキストのpp.36以降を参照
- 提出方法
 - 締切: 2019年11月7日(木) 20:00 JST
 - “`main.c`” の先頭にコメント行として学籍番号と氏名を明記
 - 1つのディレクトリに必要なすべてのファイルを保存
 - 例: ディレクトリ名を “`ex-2`” とする
 - 余分なファイルは消すように. (`*.o`ファイルなど)
 - “`bufcache`” という実行形ファイルを生成するMakefileを作成
 - `ex-2`ディレクトリの1つ上のディレクトリで
“`tar czf ex-2.tgz ex-2`” を実行
 - `keio.jp` に `ex-2.tgz` ファイルをアップロード

注意事項(1)

- コマンド入力のエラーを認識できるか.
- 各シナリオ実行時の表示は正しいか.
- シナリオ1の実行後, p.7の状態になるか.
- その後, brelseを実行するとp.11の状態になるか.
- シナリオ2の実行後, p.15の状態になるか.
- その後p.18の状態遷移を手動で実行し, brelseを実行するとp.20の状態になるか.
- シナリオ3の実行後, p.24の状態になるか.
- その後p.29の(2)までの状態遷移を手動で実行し, brelseを実行するとp.27の状態になるか.
- シナリオ4の実行後のbrelseが正しく動作するか.
- シナリオ5の実行後, LOCKED状態であったバッファをbrelseコマンドで解放したとき正しい動作をするか.

余談

- **Q:** 提出された数十人分のファイルをどのようにしてコンパイルするか?

– **A:** shellスクリプトの利用

提出されたファイルを 01.tgz, .. とする. 展開すると 01 というディレクトリが作成され, その下にファイルが格納されるはず.

```
% ls
01.tgz 02.tgz 03.tgz ...
% for i in *.tgz
> echo $i
> tar xzf $i
> cd ${i%.*}
> make >& err
> cd ..
> done
%
```

- ・すべての *.tgz ファイルについてループ
- ・どのファイル进行处理しているかの表示
- ・*.tgzファイルを展開
- ・展開したディレクトリに移動
- ・コンパイルし, エラーを“err”に格納
- ・1つ上のディレクトリに移動
- ・forループの終了 → 実行開始