第2章　バッファキャッシュ

2.1　課題の目的

第 2 の課題として，カーネル内部で行っているハードディスク (HDD) 入出力効率化のためのバッファキャッシュの管理方法を模擬するプログラムを作成する．本課題を通して以下の技法を学ぶ．

- 自己参照型の構造体ポインタ
- 双方向リスト構造の操作
- ビット操作
- 文字列処理 (第 3 章参照)

2.2　HDD の入出力

2.2.1　HDD の構造

図 2.1-(a) に示すようにハードディスク (HDD) は複数枚のプラッタ (円盤) から構成されており，その両面にデータを記録するための磁性体が塗布されている．プラッタの両面ごとに情報の記録や読み出しを行うヘッドが設けられている．また図 2.1-(b) に示すようにそれぞれのプラッタは同心円状の多数のトラックから構成されている．トラックは複数のセクタから構成される．また，すべてのプラッタにおける同一のトラック番号をもつトラックの集合をシリンダと呼ぶ．ディスクの入出力はセクタ単位で行われる．セクタサイズはブロックサイズとも呼ばれ，一般的には 512 バイトである．オペレーティングシステムカーネルは複数のセクタを 1 つの論理ブロックとして扱い，論理ブロック単位で入出力を行う．一般的なオペレーティングシステムでは，入出力の効率化およびディスクスペースの効率的な利用のために数種類の論理ブロックサイズを使い分けている．論理ブロックサイズは一般的には 1k バイト〜8k バイト程度である．以下，簡略化のためにセクタサイズ (ブロックサイズ) と論理ブロックサイズは等しいとし，サイズは 1k バイトとする．

2.2.2　キャッシングによる HDD 入出力の効率化

オペレーティングシステムカーネルは論理ブロック単位で HDD の入出力を行う．HDD の入出力には通常ミリ秒単位の時間がかかる．これは CPU の観点からはとてつもなく長

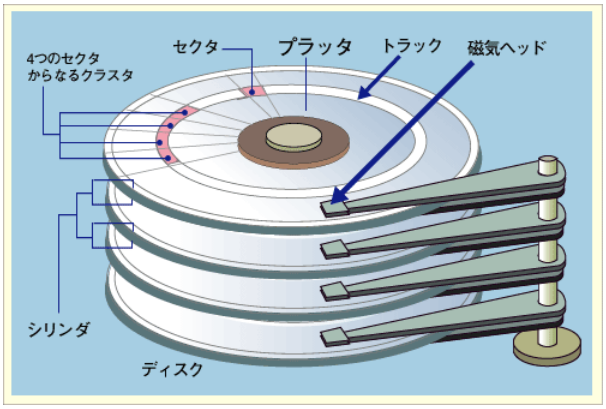

図 2.1: HDD の構造

い時間である．たとえば 4GHz のクロックスピードをもつ CPU において平均 4 クロックで 1 命令を実行するとすると，1 秒間に 10 億命令の実行が可能である．1 ミリ秒では 100 万命令の実行が可能である．したがって，効率の面からはなるべく HDD の入出力を減らすことが重要となる．

HDD 入出力を減らすための手法の 1 つがキャッシングである．カーネルに HDD のある論理ブロックの読み込み要求があり，HDD からデータを読み込んだとする．カーネルはそのデータをカーネル内に一時的に保持しておき，次に同じ論理ブロックに対する読み込みが発生した場合には HDD から読み込むのではなく，一時的に保持しているデータを返す．このように，一時的にデータを保持しておくことをキャッシングとか，データをキャッシュするという．

HDD への書込みの場合も同様にキャッシュが利用される．カーネルに対し，ある論理ブロックへの書込み要求があり，その論理ブロックのデータがカーネル内にキャッシュされていたとする．このような場合はキャッシュされているデータを書換えるだけで，HDD への書込みは行わない．その後，書換えられたキャッシュデータは定期的に HDD に書き込まれる．あるいは，そのバッファ領域が別の論理ブロックのキャッシュのために再利用する際にも HDD への書込みが発生する (第 2.4.3 節参照)．

2.3 バッファ管理の基礎

2.3.1 必要要件

カーネル内に用意できるキャッシュ用のバッファ領域は有限であるため，HDD から読み込んだすべてのデータをキャッシュとして保持することはできない．したがって要求のあった論理ブロックがキャッシュされているかは，バッファ領域を調べてみないと分からない．また，たとえば 100 ブロック分のバッファ領域がある場合，100 個の論理ブロックをキャッシュすることができるが，101 個目をキャッシュする際には，すでにキャッシュしている 100 個の論理ブロックのデータから 1 つを選んで新しくキャッシュする論理ブロックのデータと置き換える必要がある．以上のことから，効率的なバッファ管理には以下の 2 つの操作を高速に行う必要がある．

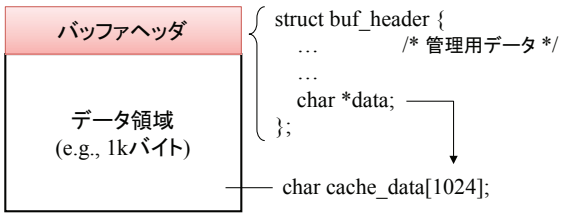

図 2.2: バッファの構造

- 検索：目的とする論理ブロックのデータがキャッシュされているかを確認.
- 置換：キャッシュされているデータを置き換える際，どのデータを追い出すべきかを決定.

2.3.2 バッファの構造

バッファ領域を管理するため，論理ブロック 1 つ分のデータを保持する領域ごとに管理用データとして図 2.2 のような構造体 (`struct buf_header`) を定義する．データを保持する領域と管理用データの領域は連続している必要はない．実は，このような場合はデータ領域と管理用データ領域を分けて配置した方が都合がよい．理由は各自考えてみよう．

管理用データ領域を今後はバッファヘッダと呼ぶ．データ領域とバッファヘッダを分けて配置するため，`struct buf_header` のメンバとしてデータ領域へのポインタとして `char *data` を定義する．以降，前節で述べた検索と置換の操作に必要となる `struct buf_header` のメンバを検討していく．

2.3.3 双方向リストによる検索の効率化

まず，あるバッファ領域がどの論理ブロックのデータをキャッシュしているかを示すため，`struct buf_header` に論理ブロック番号を示すメンバである `int blkno` を加える．

次に検索方法を考えてみよう．指定された論理ブロック番号のデータがキャッシュされているかを確認するもっとも単純な方法は，すべてのバッファヘッダの `blkno` を順々に調べていくことである．しかしこれでは効率が悪い．そこでハッシュリストによる検索方法を採用することにする．

ハッシュ関数 $y = f(x)$ を考える．ハッシュ関数とは一方向性関数であり，任意長の入力 x から固定長の値 y を出力する．入力 x から出力 y を求めることはできるが，出力 y から入力 x を求めることはできない．ここでは単純化のため，ハッシュ関数として $y = f(blkno) = blkno \ mod \ 4$ (4 の剰余) を採用する．すると，すべての論理ブロック番号 (`int blkno`) は 0〜3 という値 (ハッシュ値) にマッピングされる．

次に，同一のハッシュ値をもつバッファ領域をリスト構造で管理する．ここでは図 2.3 に示すような双方向リストを用いる．双方向リストとは，各要素が順方向と逆方向の 2 つのポインタで結合されているリスト構造である．第 2.4.1 節で示すように，バッファ管理においてはリスト内の任意の要素をリストから削除することがある．双方向リストは任意

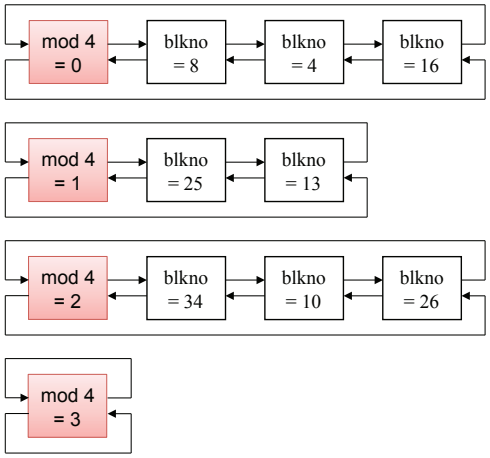

図 2.3: 双方向リストによるバッファの管理

```
01: #define NHASH    4
02:
03: struct buf_header hash_head[NHASH];
```

図 2.4: リストヘッドの宣言

の要素をリストから削除する場合に便利な構造である．なぜ便利かは各自で考えてみるとよい．

図 2.3 の 4 つの双方向リストはそれぞれハッシュ値が 0〜3 の論理ブロックのためのものである．双方向リストの先頭 (最も左) はリストヘッドと呼ばれる管理用のデータであり，リストに連結されるデータ構造と同一の型である struct buf_header 型をとる．なぜリストヘッドはリストに連結されるデータと同じ型をとるのがよいのかは，各自で考えてみるとよい．この例ではハッシュ値は 0〜3 の 4 種類であるので，リストヘッドは図 2.4 に示すように 4 つの要素を持つ struct hash_list の配列として宣言すればよい．

双方向リストにおいて，リストのための変数は struct buf_header 型の領域を指しているので struct buf_header 型のポインタということになり，これを struct buf_header 型のメンバとして含むことになる．すなわち構造体のメンバとして自分自身へのポインタを含むことになる．このようなポインタを自己参照型のポインタと呼ぶ．双方向リストのための自己参照型ポインタを含む struct buf_header の宣言は図 2.5 のようになる．

2.3.4 検索方法

論理ブロック番号を入力し，これに対応するバッファが存在するかを調べる関数 struct buf_header * hash_search(int blkno) は図 2.6 のようになる．まず，この関数は見つかったバッファヘッダへのポインタを返すので，01 行目で関数の型として “struct buf_header *” を宣言する．02 行目から 06 行目までは説明の必要はないであろう．07 行目では入力

```
01: struct buf_header {
02:     int blkno;                    /* 論理ブロック番号 */
03:     struct buf_header *hash_fp; /* ハッシュの順方向ポインタ */
04:     struct buf_header *hash_bp; /* ハッシュの逆方向ポインタ */
05:     . . .
06:     . . .
07:     . . .
08:     char *cache_data;             /* キャッシュデータ領域へのポインタ */
09: };
```

図 2.5: struct buf_header の定義 (1)

```
01: struct buf_header *
02: hash_search(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: }
```

図 2.6: struct buf_header *hash_search()

された論理ブロック番号のハッシュ値を計算し，h という変数に代入しておく．なお，int hash(int blkno) という関数は別のところで定義されているとする．

08 行目の for 文が実質的な検索である．ここをよく理解して欲しい．まず，hash_head[h] は入力された論理ブロックのバッファヘッダがリンクされるリストヘッドである．そしてこのリストヘッドの順方向ポインタが指すバッファヘッダから検索が開始される．検索が開始されるバッファヘッダへのポインタを変数 p に代入しておく (p = hash_head[h].hash_fp)．for 文の継続条件は，リンクをたどった結果がリストヘッドに達しない間である (p != &hash_head[h])．そして for 文を 1 回実行するごとに順方向ポインタをたどっていく (p = p->hash_fp)．09 行目では，現在調べているバッファヘッダの論理ブロック番号と入力された論理ブロック番号が等しいかを調べている．もし等しければ 10 行目で見つかったバッファヘッダへのポインタを返す．入力された論理ブロックがキャッシュされていない場合は for 文が終了するので，11 行目で NULL ポインタを返す．

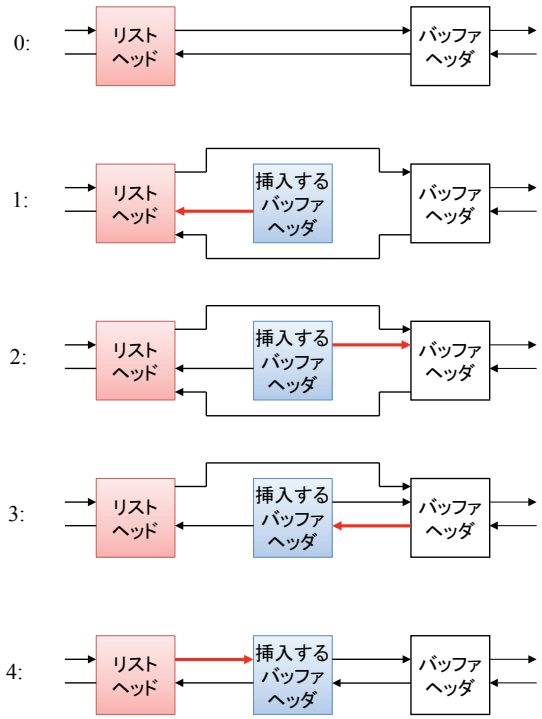

図 2.7: 双方向リストの先頭への挿入

図 2.3 の “mod 4 = 3” のようにリストが空の場合，この for 文は本文を実行することなく終了することを確かめて欲しい．

2.3.5 リストへの挿入

カーネルに HDD 読み込みの要求があり，要求された論理ブロックがキャッシュされていない場合は HDD から新たに論理ブロックを読み込み，これをキャッシュする．すなわち新しく読み込まれたデータを保持するデータ領域のためのバッファヘッダを双方向リストに挿入することになる．ここではバッファヘッダを双方向リストの先頭あるいは末尾に挿入する操作について考えてみる．

双方向リストの先頭に挿入する場合の操作を図 2.7 に示す．第 0 段階は初期状態であり，第 4 段階が終了状態である．各段階において変化したポインタを赤線で表している．第 1 段階においては，挿入するバッファヘッダの逆方向ポインタをリストヘッドに設定している．第 2 段階においては，挿入するバッファヘッダの順方向ポインタを元々先頭にあったバッファヘッダに設定している．第 3 段階では，元々先頭にあったバッファヘッダの逆方向ポインタを挿入するバッファヘッダに設定している．最後に第 4 段階では，リストヘッドの順方向ポインタを挿入するバッファヘッダに設定している．

この操作を行う関数 void insert_head(struct buf_header *h, struct buf_header *p) を C 言語で記述することは練習問題とするので，各自考えて欲しい．ここで引数 h はリストヘッドへのポインタであり，引数 p は挿入するバッファヘッダである．その際，双方向リストが空の場合でも正しく動作することを確認して欲しい．さらに双方向リストの末尾

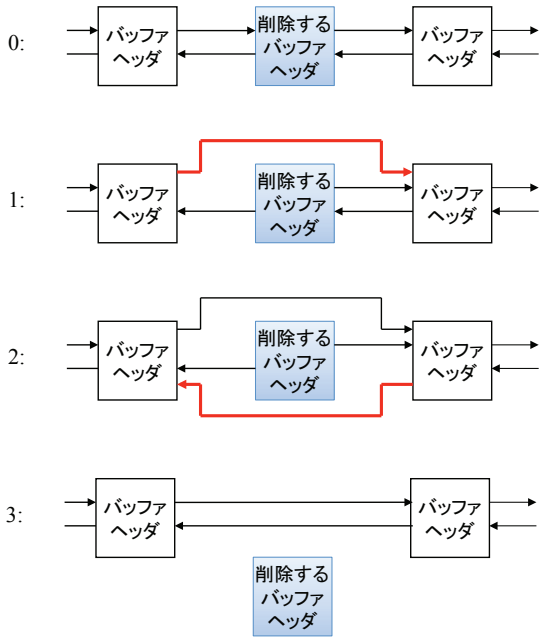

図 2.8: 双方向リストからの削除

に挿入する `void insert_bottom(struct buf_header *h, struct buf_header *p)` も記述すること．

2.3.6 リストからの削除

次に，双方向リストに接続されているバッファヘッダを双方向リストから削除する操作について考えてみる．削除の操作を図 2.8 に示す．

第 0 段階は初期状態であり，第 3 段階が終了状態である．各段階において変化したポインタを赤線で表している．第 1 段階では，削除するバッファヘッダの後方にあるバッファヘッダの順方向ポインタを，削除するバッファヘッダの前方にあるバッファヘッダに設定している．第 2 段階では，削除するバッファヘッダの前方にあるバッファヘッダの逆方向ポインタを，削除するバッファヘッダの後方にあるバッファヘッダに設定している．最後に第 3 段階では，削除するバッファヘッダの順方向ポインタおよび逆方向ポインタに `NULL` ポインタを設定している．

この操作を行う関数 `void remove_hash(struct buf_header *p)` を C 言語で記述することも練習問題とする．引数 `p` は削除すべきバッファヘッダへのポインタである．その際，削除するバッファヘッダが双方向リストの最後の要素である場合，削除後のリストヘッドの状態が正しくなることを確認して欲しい．

2.3.7 バッファの状態

カーネル内部には多数のバッファ領域が用意されている．バッファのデータ領域はさまざまな状態を持つ．たとえば，あるバッファ領域は論理ブロックの有効なデータをキャッ

シュしていたり，またあるバッファ領域は現在未使用である場合もある．また，あるプロセスが使用しているバッファ領域は他のプロセスに横取りされないようにロックしなければならない．また，あるバッファ領域に HDD からデータを読み込んでいる場合，それが終了するまでバッファ領域のデータは使用できない．

この例では，バッファ領域は以下のような状態を持つこととする．

- ロックされている (STAT_LOCKED)：あるプロセスがこのバッファ領域を使用している．他のプロセスはこのバッファを使用することはできない．
- 有効なデータを保持している (STAT_VALID)：このバッファはある論理ブロックの有効なデータをキャッシュしている．
- 遅延書込み (STAT_DWR)：このバッファ領域のデータ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)：現在，このバッファ領域のデータはカーネルが read または write を行っている．このキャッシュデータを利用するには，カーネルの read/write の終了を待つ必要がある．
- 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 (STAT_WAITED)：このバッファ領域はロックされており，他のプロセスがこのバッファ領域がフリーになるのを待っている．
- このバッファが保持しているデータは “古く” なっている (STAT_OLD): このバッファが保持しているデータには書込みが行われたが，遅延書込みのため，まだ HDD に書き戻されていない．そのためデータは “古く” なっている．このバッファが解放されたときにはフリーリストの先頭に戻す必要がある (2.3.9 節および 2.5 節参照)．

バッファ領域は複数の状態を同時に持つことがある．たとえば，あるバッファ領域はロックされており，有効なデータを保持していることがある．またあるバッファ領域は，ロックされており，有効なデータを保持しており，さらに他のプロセスがフリーになるのを待っているかもしれない．

このような場合，プログラム上では状態を保持する変数 (たとえば unsigned int 型) を用意し，その変数中の各ビットを各状態に対応させる．int 型が 32 ビットであるマシンの場合，unsigned int 型の変数では 32 種類の状態を表すことができる．そこでこの例題では図 2.5 で定義したバッファヘッダに状態を保持するための変数を加える (図 2.9 の 05 行目)．そして各状態を図 2.10 のように定義する．

図 2.11 にバッファの状態遷移図を示す．図において四角で示したのが 1 つの状態である．図 2.11 から分かるように，バッファは図 2.10 に示した複数の状態を同時にとることがあることに注意して欲しい．状態遷移の詳細は 2.4.1〜2.4.5 節で述べる．

2.3.8 ビット操作

バッファの状態が変わった場合，そのバッファヘッダのメンバである unsigned int stat の対応するビットを操作する必要がある．たとえば，あるフリーなバッファがあるプロセスに

```
01: struct buf_header {
02:     int blkno;                    /* ハッシュの論理ブロック番号 */
03:     struct buf_header *hash_fp; /* ハッシュの順方向ポインタ */
04:     struct buf_header *hash_bp; /* 逆方向ポインタ */
05:     unsigned int stat;            /* バッファの状態 */
06:     . . .
07:     . . .
08:     char *cache_data;             /* キャッシュデータ領域へのポインタ */
09: };
```

図 2.9: struct buf_header の定義 (2)

```
01: #define STAT_LOCKED     0x00000001  /* ロックされている */
02: #define STAT_VALID      0x00000002  /* 有効なデータを保持 */
03: #define STAT_DWR        0x00000004  /* 遅延書込み */
04: #define STAT_KRDWR      0x00000008  /* カーネルが read/write */
05: #define STAT_WAITED     0x00000010  /* 待っているプロセスがある */
06: #define STAT_OLD        0x00000020  /* 保持しているデータは古い */
```

図 2.10: バッファの状態の定義

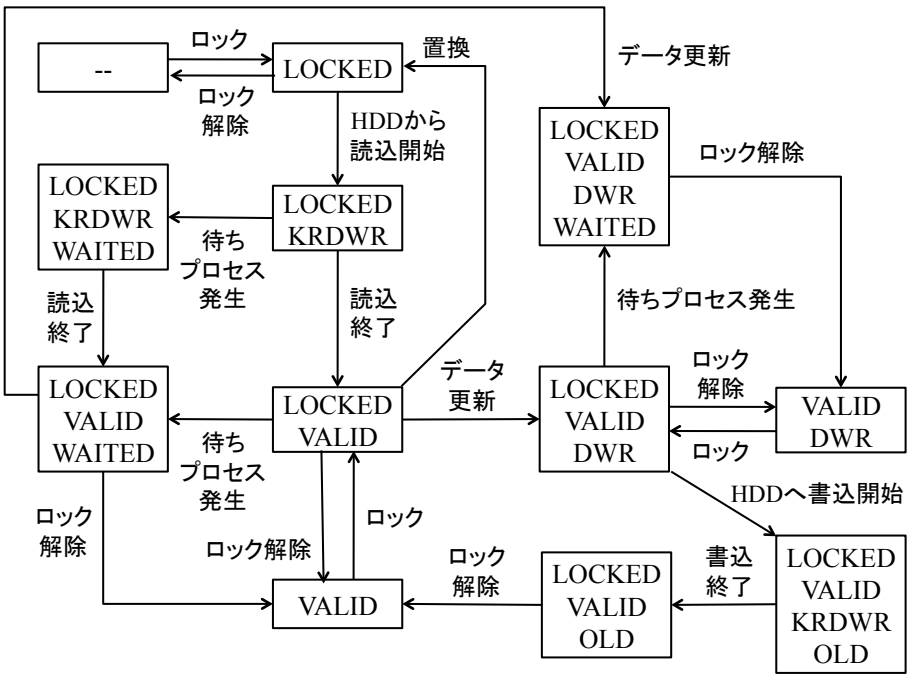

図 2.11: バッファの状態遷移図

```
01: struct buf_header *p;
02:
03: p = ...;                        /* p が struct buf_header 型の領域を
                                       指すようにする */
04: p->stat |= STAT_LOCKED;         /* STAT_LOCKED ビットをセット */
05: p->stat &= ~STAT_LOCKED;        /* STAT_LOCKED ビットをリセット */
```

図 2.12: ビットのセット・リセット

よってロックされたとすると，対応するバッファヘッダのメンバである stat の STAT_LOCKED ビットをセットしなければならない．すなわち，ポインタ struct buf_header *p が対応するバッファヘッダを指しているとすると，p->stat の STAT_LOCKED ビット以外は変化させずに STAT_LOCKED ビットのみをセットする必要がある．同様に，ロックされていたキャッシュがフリーになった場合は p->stat の STAT_LOCKED ビット以外は変化させずに STAT_LOCKED ビットのみをリセットする必要がある．特定のビットのセット・リセットは図 2.12 に示すように行う．

図 2.12 の 04 行目では p->stat の STAT_LOCKED ビットのみをセットし，他のビットは変化させていない．なぜそうなるのかを確認すること．なお，“|” はビットごとの論理和を表す二項演算子であることを思い出すように．同様に，05 行目では p->stat の STAT_LOCKED ビットのみをリセットし，他のビットは変化させていない．なぜそうなるのかを確認すること．なお，“&” はビットごとの論理積を表す二項演算子であり，“~” はビットごとの反転を表す単項演算子であることを思い出すように．

2.3.9 フリーリスト

次に，ロックされていない (フリーな) バッファ領域の管理法について考える．フリーなバッファとはメンバ stat に STAT_LOCKED ビットがセットされていないもののことである．たとえば，カーネルにある論理ブロックの読み込み要求があり，バッファを検索したところ該当する論理ブロックはキャッシュされていないことがわかったとする．このような場合にはフリーなバッファ領域を確保し，そのデータ領域に HDD からデータを読み込むことになる．このような場合にフリーなバッファ領域の検索が発生するが，そのために各バッファヘッダのメンバ stat を順々に調べていくのでは効率が悪い．

そこでフリーなバッファの管理にも双方向リストを用いる．フリーなバッファを管理するための双方向リスト (以降，フリーリストと呼ぶ) に用いる順方向ポインタと逆方向ポインタを struct buf_header の定義に加える (図 2.13 の 06，07 行目参照)．これが本課題における struct buf_header の最終形である．また，ハッシュリストと同様にフリーリストにもリストヘッドである struct buf_header free_head を定義する．

たとえば，12 個のバッファが論理ブロック 3, 4, 5, 10, 17, 28, 35, 50, 64, 97, 98, 99 のキャッシュとして使用されているとすると，これらは図 2.14 に示すようにハッシュリストで管理される．このとき，論理ブロック 3, 4, 5, 10, 28, 97 のバッファがフリーである (ロッ

```
01: struct buf_header {
02:     int blkno;                    /* 論理ブロック番号 */
03:     struct buf_header *hash_fp;   /* ハッシュの順方向ポインタ */
04:     struct buf_header *hash_bp;   /* ハッシュの逆方向ポインタ */
05:     unsigned int stat;            /* バッファの状態 */
06:     sturct buf_header *free_fp;   /* フリーリストの順方向ポインタ */
07:     struct buf_header *free_bp;   /* フリーリストの逆方向ポインタ */
08:     char *cache_data;             /* キャッシュデータ領域へのポインタ */
09: };
```

図 2.13: `struct buf_header` の定義 (3)

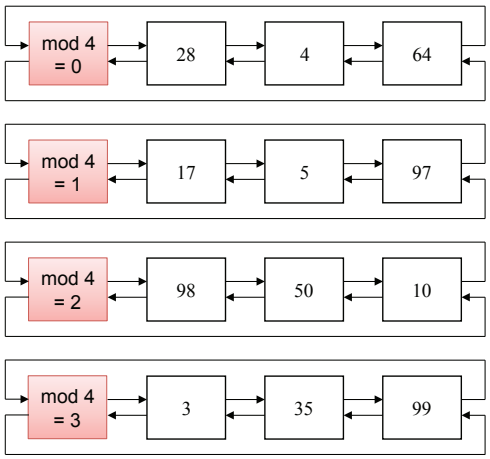

図 2.14: ハッシュリスト上のバッファ

クされていない) とすると，これらのバッファはフリーリスト上にもあることになる．その結果，この 12 個のバッファは図 2.15 に示すように管理される．この図において，たとえば論理ブロック 3 のバッファはハッシュリストにつながっていると同時に，フリーであるためフリーリストにもつながっている．一方，論理ブロック 35 のバッファはフリーリストにつながっていないため，現在ロックされている (フリーでない) ことを示している．

2.3.10 置換方法

すべてのバッファ領域が使用中のとき，新たな論理ブロックをキャッシュする必要がある場合，現在使用中のバッファを新たな論理ブロックのキャッシュとして置き換える必要がある．このとき，どのバッファを選ぶかが性能に大きな影響を与える．頻繁に利用されるデータを保持しているバッファを置き換えてしまうと，次にこのデータの読み込み要求があったときには HDD から読み込んでこなければならず，時間がかかってしまう．したがって，なるべく利用頻度の低いバッファを置き換えるのが有効である．

このような目的に使われるのが**LRU (Least Recently Used)** という手法である．LRU

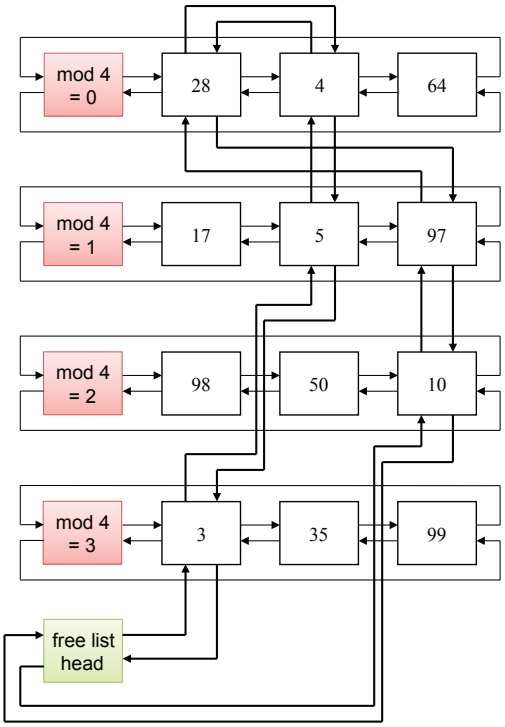

図 2.15: ハッシュリストとフリーリスト

では，利用された時刻によって先頭が最も古くなるように対象となるアイテムを並べておく．ここで扱っているバッファに適用すると，アクセスされた時刻によって先頭が最も古くなるようにフリーなバッファをフリーリストにつなぎかえる．すなわち，あるバッファがアクセスされ，その後フリーになったときにはフリーリストの最後にこのバッファをつなげればよい．このようにすると，自然にフリーリストにはアクセスされた時刻が古い順にバッファがつながれることになる．そしてバッファの置き換えが必要になったときにはフリーリストの先頭のバッファを選択する．このようにすることによって LRU が実現される．

2.4 バッファ管理アルゴリズム: getblk()

次に，論理ブロック番号を指定して，その論理ブロック用のバッファを得るアルゴリズム getblk() について見てゆく．getblk() の擬似コードを図 2.16 に示す．getblk() は入力として論理ブロック番号をとり，出力としてはロックされた struct buf_header へのポインタを返す．

getblk() の動作には 5 つの場合 (シナリオ) がある．以下，それぞれのついて説明する．なお，シナリオ 1, 2, 5 の初期状態は図 2.15 に示したとおりとする．

2.4.1 シナリオ 1

これは最も単純なシナリオである．要求された論理ブロックのバッファがハッシュリスト上に存在し，かつフリーな場合である．たとえば，図 2.15 において論理ブロック 10 が

```
01: struct buf_header *  /* 出力: ロックされた buf_header へのポインタ */
02: getblk(int blkno)    /* 入力: 論理ブロック番号 */
03: {
04:     while (buffer not found) {
05:         if (blkno in hash queue) {
06:             if (buffer locked) {
                    /* シナリオ 5 */
07:                 sleep(event buffer becomes free);
08:                 continue;
09:             }
                /* シナリオ 1 */
10:             make buffer locked;
11:             remove buffer from free list;
12:             return pointer to buffer;
13:         } else {
14:             if (not buffer on free list) {
                    /* シナリオ 4 */
16:                 sleep(event any buffer becomes free);
17:                 continue;
18:             }
19:             remove buffer from free list;
20:             if (buffer marked for delayed write) {
                    /* シナリオ 3*/
21:                 asynchronous write buffer to disk;
22:                 continue;
23:             }
                /* シナリオ 2 */
24:             remove buffer from old hash queue;
25:             put buffer onto new hash queue;
26:             return pointer to buffer;
27:         }
28:     }
29: }
```

図 2.16: バッファ割り当てアルゴリズムの擬似コード

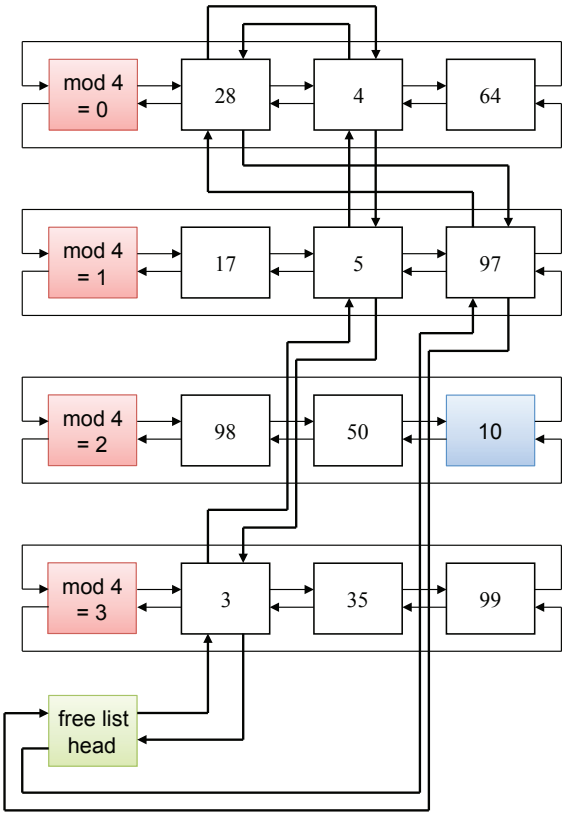

図 2.17: シナリオ 1 の動作後 (getblk(10))

要求された場合である．論理ブロック 10 のバッファは “mod 4 = 2” のリスト上に見つかり，さらにフリーリスト上にもある．そこでこのバッファをロック状態にし，フリーリストから削除する．getblk(10) の実行後は図 2.17 に示したようになる．

また，論理ブロック 10 を保持するバッファ (これを “buf-10” と呼ぶこととする) の状態遷移を図 2.18 に示す．最初，buf-10 の状態は “VALID” である (1)．次に buf-10 はフリーリストから外されてロックされるため，“LOCKED | VALID” 状態となる (2)．

2.4.2 シナリオ 2

このシナリオではバッファの置換が発生する．要求された論理ブロックのバッファがハッシュリスト上に存在しない場合である．たとえば，図 2.15 において論理ブロック 18 が要求された場合である．論理ブロック 18 用のバッファは “mod 4 = 2” のハッシュリスト上に存在するはずだが，実際には存在しない．そこでフリーリストの先頭にあるバッファを要求された論理ブロック用に使用することになる．この例では論理ブロック 3 用に使われていたバッファを論理ブロック 18 用に置き換えることになる．getblk(18) の実行後は図 2.19 に示したようになる．

図 2.20 に buf-3(途中で buf-18 となる) の除隊遷移を示す．最初，buf-3 は “VALID” 状態であるが (1)，ロックされるため “LOCKED | VALID” 状態となる (2)．このバッファは論理ブロック 3 のデータをキャッシュしているが，次に論理ブロック 18 のために使われること

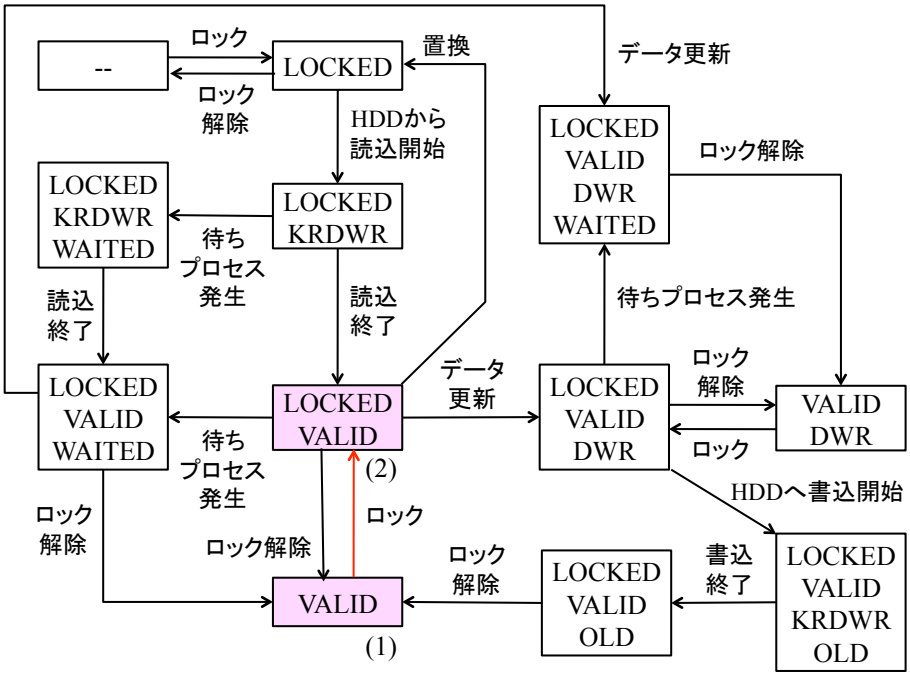

図 2.18: シナリオ 1: buf-10 の状態遷移

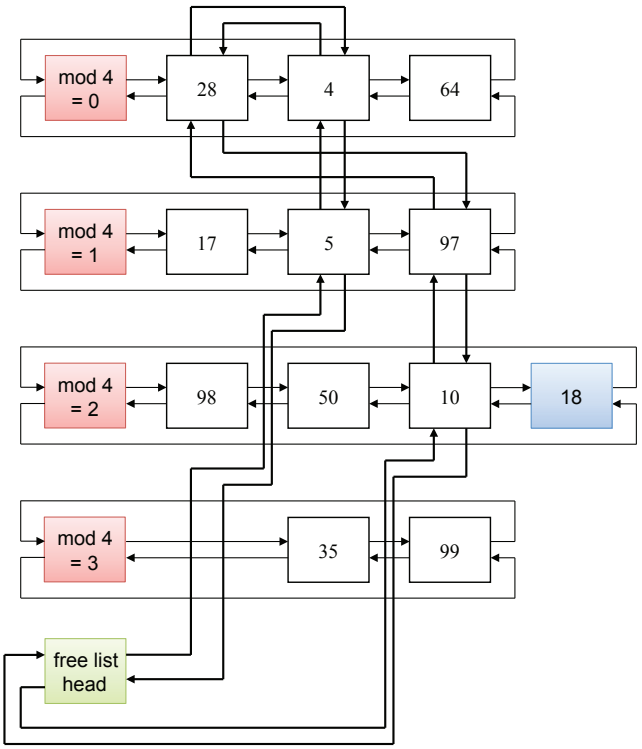

図 2.19: シナリオ 2 の動作後 (`getblk(18)`)

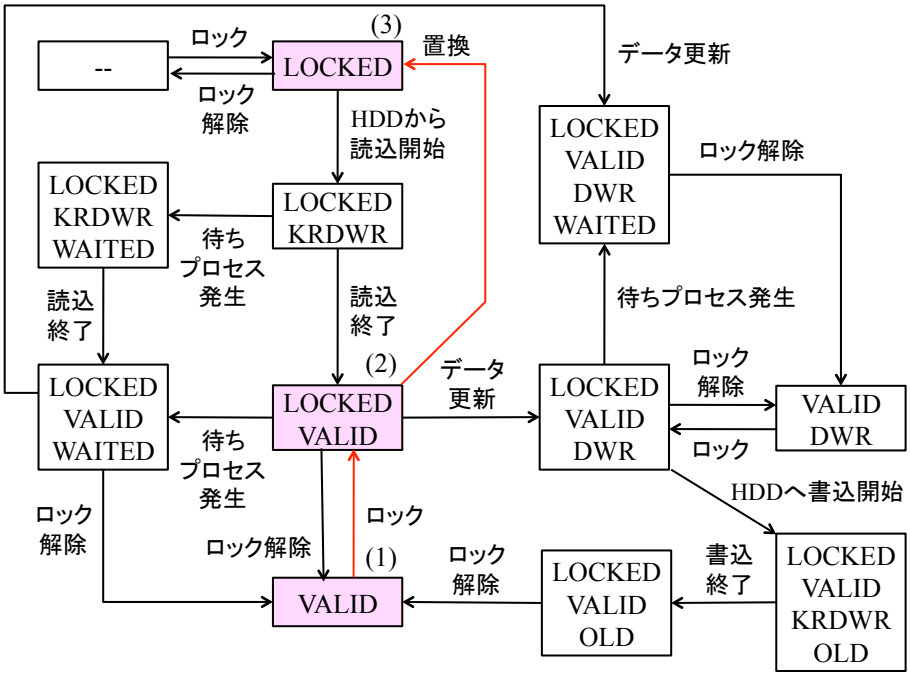

図 2.20: シナリオ 2: buf-3(buf-18) の状態遷移

になるので，このバッファは有効なデータを保持していない状態となる．そこで “LOCKED” 状態に遷移する (3).

このシナリオによって得られたバッファ(buf-18) には有効なデータがキャッシュされていないため，通常はこの後に HDD からの読み込みが発生する．その際の buf-18 の状態遷移を図 2.21 に示す．図 2.20 の続きで，buf-18 は “LOCKED” 状態にある (3)．次にカーネルはこのバッファに HDD からデータを読み込むため，“KRDWR” ビットをセットし，“LOCKED | KRDWR” 状態となる (4)．やがてデータの読み込みが終了すると，“LOCKED | VALID” 状態となる (5).

2.4.3 シナリオ 3

このシナリオではキャッシュされていたデータの HDD への書き戻しが発生する．ユーザからの HDD への書込み要求の際にはただちに HDD への書込みは行われず，キャッシュされているデータのみが書換えられる．このようにして HDD へのアクセスを減らし，システムの性能向上を図っている．このようにデータが書換えられたバッファは STAT_DWR (delayed write) という状態になる．delayed write という状態のバッファが他の論理ブロックのために置き換えられる際には，データを HDD に書き戻さなければならない．

シナリオ 3 の例としては，シナリオ 2 と同様に getblk(18) が実行され，要求されたバッファがハッシュリスト上になく，フリーリストの先頭のバッファを置き換えようとするが，このバッファが delayed write という状態であったという場合である．この場合，delayed write 状態のバッファについては HDD へのデータの書込みが開始される．HDD への書込

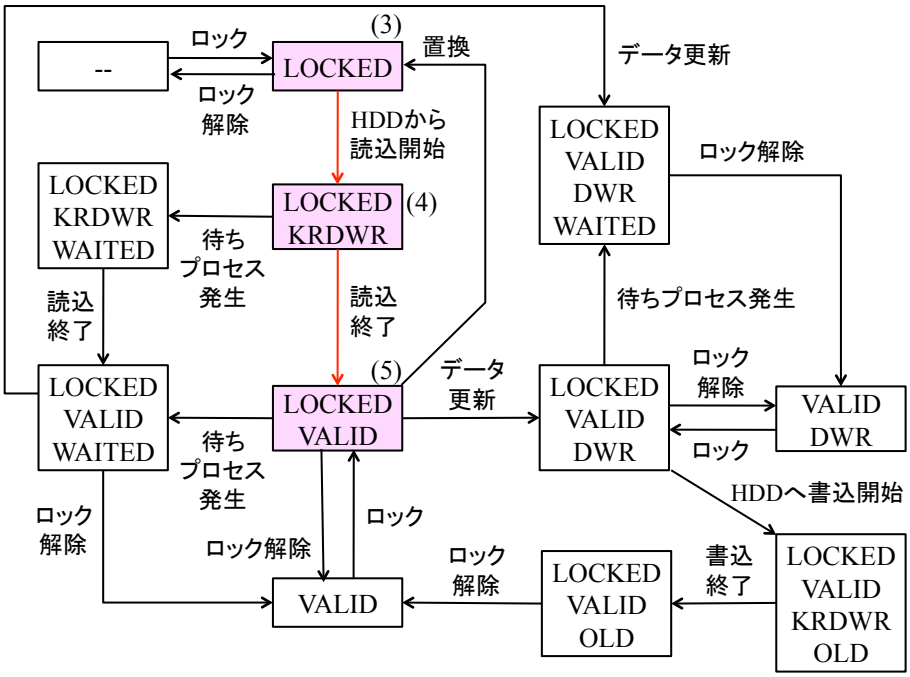

図 2.21: シナリオ 2: buf-18 のその後の状態遷移

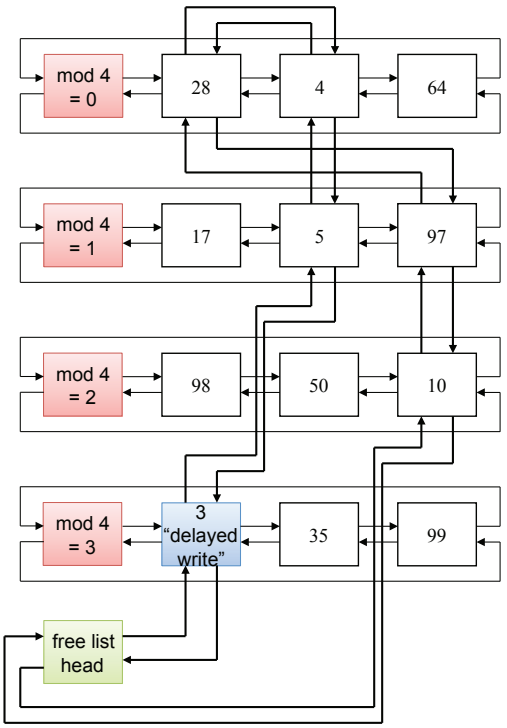

図 2.22: シナリオ 3 の初期状態 (`getblk(18)`)

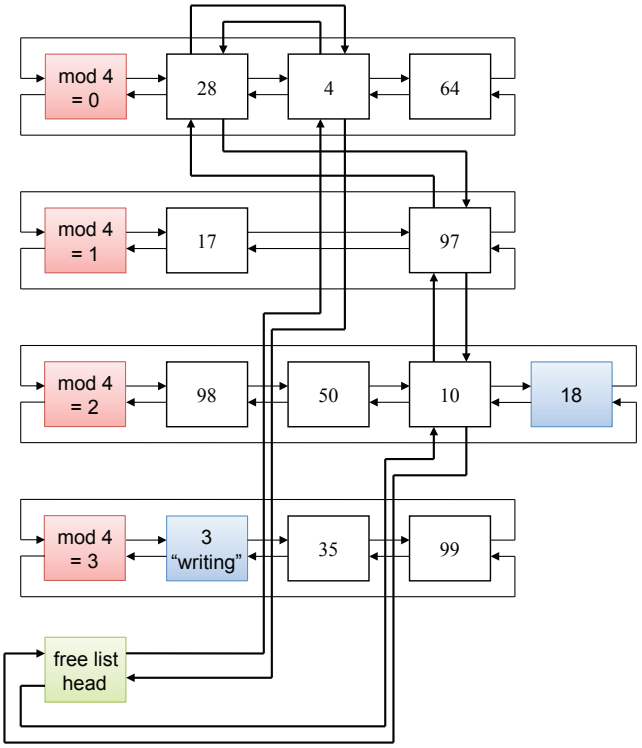

図 2.23: シナリオ 3 の中間状態 (`getblk(18)`)

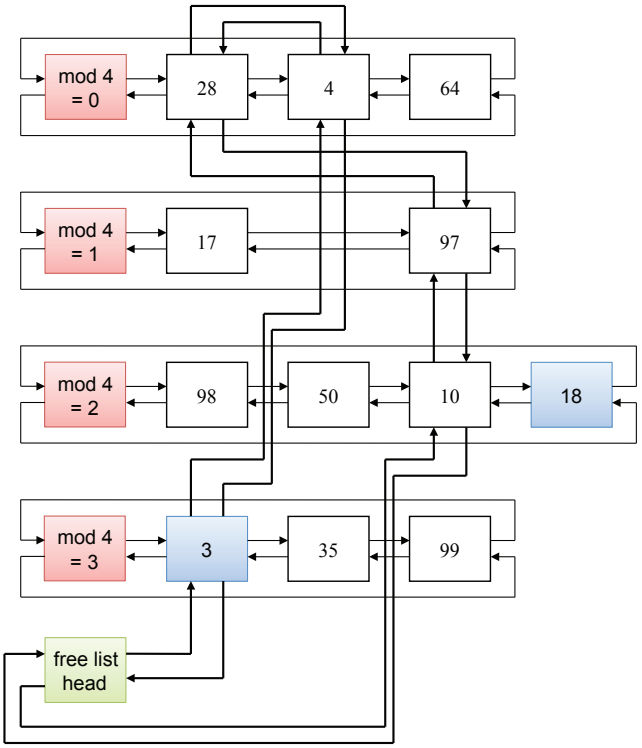

図 2.24: シナリオ 3 の動作後 (`getblk(18)`)

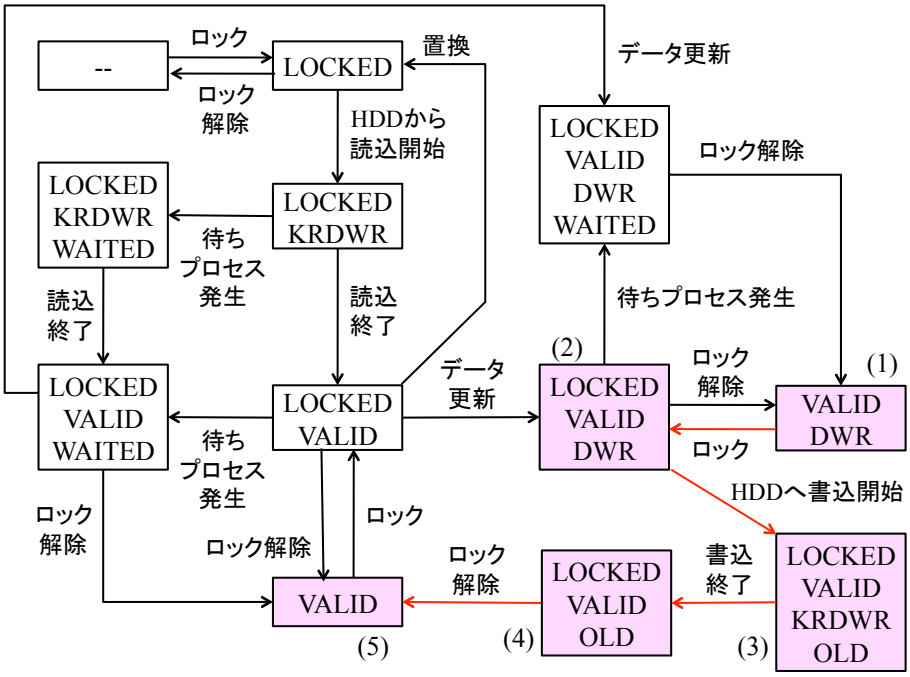

図 2.25: シナリオ 3: buf-3 の状態遷移

みは時間がかかるので，終了を待つことなく `while` ループの先頭に戻る．この例ではフリーリストの先頭にある論理ブロック 3 用のバッファが delayed write という状態であったため，このバッファの HDD への書き戻しを開始して `while` ループの先頭に戻る．その後はシナリオ 2 にしたがって論理ブロック 5 用のバッファが置き換えられて論理ブロック 18 用のバッファとなる．この例における初期状態を図 2.22 に示す．図 2.23 は中間状態である．論理ブロック 5 用のバッファが論理ブロック 18 用に置き換えられ，論理ブロック 3 用のバッファのデータは HDD に書き込まれている途中である．最終状態を図 2.24 に示す．論理ブロック 3 用のバッファはデータの HDD への書込みが終わり，再度フリーリストの先頭につながれている．

buf-3 の状態遷移を図 2.25 に示す．最初，buf-3 は有効なデータをキャッシュしており，データが書き換えられているがまだ HDD に書き戻されていないため “`VALID | DWR`” 状態となっている (1)．buf-3 を buf-18 として使用するにはキャッシュしているデータを HDD への書き戻す必要がある．そこでまずバッファをロックし (2)，次に書き戻し作業中を表すために “`LOCKED | VALID | DWR | OLD`” 状態となる (3)．“`OLD`” はこのバッファをフリーリストに挿入する際，先頭に加えることを示すための状態である．やがて書き戻し作業が終了し (4)，最終的にはフリーリストの先頭に挿入されて “`VALID`” 状態となる．

一方，論理ブロック 5 のためのバッファ (buf-5) の状態遷移はシナリオ 2 の図 2.20 や図 2.21 と同様である．

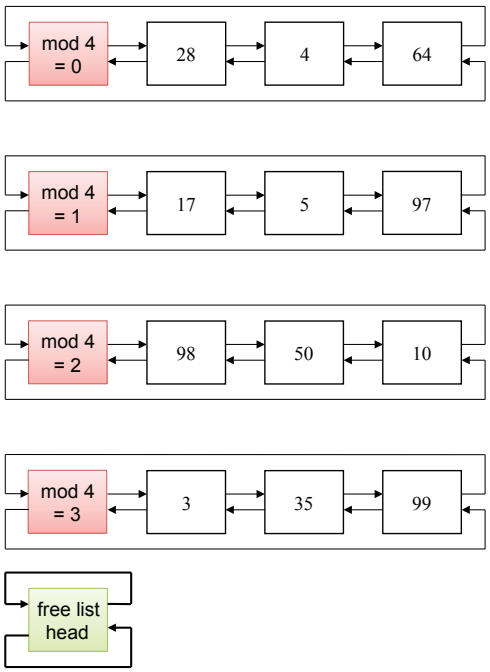

図 2.26: シナリオ 4 (getblk(18))

2.4.4 シナリオ 4

このシナリオでは，要求されたバッファがハッシュリスト上になく，かつフリーリストが空のときである．この場合，この getblk() を実行したプロセスは空バッファが生じるまでスリープする．図 2.26 のような状態で getblk(18) が実行されると，このプロセスは空バッファを待ってスリープする．

2.4.5 シナリオ 5

このシナリオはシナリオ 1 と同様に要求されたバッファはハッシュリスト上にあるが，シナリオ 1 と違ってロックされているという場合である．たとえば，図 2.15 において getblk(64) を実行した場合である．論理ブロック 64 用のバッファはハッシュリスト上にあるが，このバッファはロックされているためフリーリスト上にない．getblk(64) を呼び出したプロセスは，このバッファがフリーになるのを待ってスリープする．

以降はシステムプログラミングというよりオペレーティングシステムに関することだが，シナリオ 5 において sleep() の終了後に continue を実行して while ループの先頭に戻っていることに注意して欲しい．sleep() が終了したということは，目的のバッファがフリーになったということなので，そのバッファを getblk() の返り値としてもよいのではないかと思うだろう．しかしそれではまずいのである．これ以上の説明はオペレーティングシステムの範囲になるのでここでは割愛するが，各自考えてみると良い．

図 2.27 に buf-64 の状態遷移を示す．最初，buf-64 は “LOCKED | VALID” 状態にある (1)．次にこのバッファのロックの解放を待つプロセスが現れるので，“LOCKED | VALID

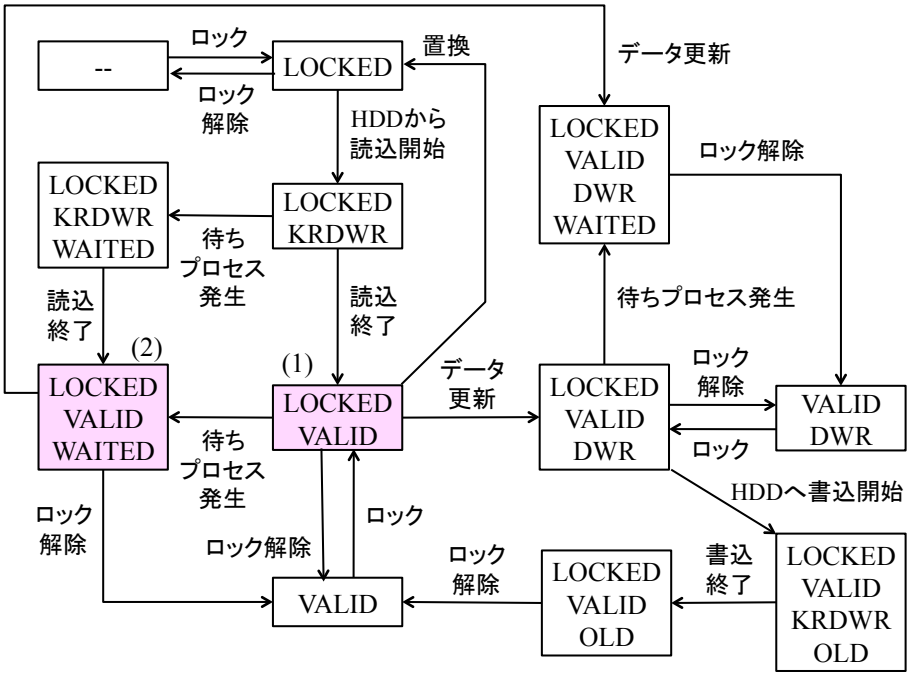

図 2.27: シナリオ 5: buf-64 の状態遷移

| WAITED" 状態となる.

2.5 バッファの解放

カーネルは getblk() でロックされたバッファを得ると，たとえば HDD からバッファにデータを読み込んだり，バッファにデータを書込み，さらに HDD にデータを書込むかもしれない．このような場合，カーネルはバッファをロックしたままにしておき，他のプロセスがこのバッファにアクセスしないようにする．カーネルはバッファの利用が終わると，brelse() を呼び出してバッファを解放する (図 2.28 参照).

brelse() では，まず sleep しているプロセスを wakeup する．03 行目の wakeup は，シナリオ 4 で sleep したプロセスを wakeup することに相当する．また 04 行目の wakeup は，シナリオ 5 で sleep したプロセスを wakeup することに相当する．05 行目と 10 行目の説明は後回しにする．06 行目でバッファの状態を判断し，その結果によってフリーリストの先頭に挿入するか，末尾に挿入するかを決めている．なお，シナリオ 3 のように delayed write とマークされたバッファの内容がディクスに書き戻されたあとには old というマークが付けられ，このバッファはフリーリストの先頭に挿入される.

05 行目と 10 行目の説明は，OS の講義で際どい領域 (**critical region**) や相互排除 (**mutual exclusion**) などを習わないと理解が困難である．06 行目から 09 行目ではフリーリストを操作する．あるプロセスがこの区間を実行しているうちに HDD からの割り込みが発生し，割り込み処理ルーチンの中から brelse() が呼び出されてフリーリストを操作しようとすると，リスト構造が破壊されてしまうことがある．そこでフリーリストの操作が

```
01: void brelse(struct buf_header *buffer)  /* buffer is locked */
02: {
03:     wakeup all procs; event, waiting for any buffer to become free;
04:     wakeup all procs; event, waiting for this buffer to become free;
05:     raise processor execution level to block interrupts;
06:     if (buffer contents valid and buffer not old)
07:         enqueue buffer at end of free list;
08:     else
09:         enqueue buffer at beginning of free list;
10:     lower processor execution level to allow interrupts;
11:
12:     unlock(buffer);
13: }
```

図 2.28: バッファ解放アルゴリズムの擬似コード

完了するまでの間は割り込みが発生しないようにする必要がある．そこで 05 行目で CPU の実行レベルを上げて割り込みを禁止している．そして 10 行目で CPU の実行レベルをもとのレベルまで下げ，割り込みを許している．このような操作を相互排除と呼ぶ．

図 2.16 で示した擬似コード内でもハッシュリストやフリーリストを操作している．本来このような区間は際どい領域であるので相互排除が必要であるが，煩雑になるので記述を省略している．

課題2 ーステップ1

図2.16に示した擬似コードによるgetblk()をC言語で記述しなさい．getblk()内で使用する関数もすべて記述しなさい．なお，sleep()については以下のようにコメントとして記述すればよい．

```
01:         // sleep();
02:         printf("Process goes to sleep\n");
03:         return NULL;
```

次に，図2.28に示した擬似コードによるbrelseをC言語で記述しなさい．なお，1つ目のwakeup()と2つ目のwakeup()については以下のようにコメントとして記述すればよい．ただし，2つ目のwakeup()については，このwakeup()を実行する条件をきちんと判断しなさい．

```
1つ目のwakeup()
01:         // wakeup();
02:         printf("Wakeup processes waiting for any buffer\n");

2つ目のwakeup()
01:         // wakeup();
02:         printf("Wakeup processes waiting for buffer of blkno %d\n",
                                                                blkno);
```

また，raise/lower processor execution levelについては以下のようにコメントとして記述すればよい．

```
01:         // raise_cpu_level();
02:         // lower_cpu_level();
```

課題2 ーステップ2

ステップ1で作成したgetblk()の動作をインタラクティブに確かめるためのmain()および必要な関数をC言語で作成しなさい．バッファの数は12個とする．またハッシュ関数は4の剰余とする．makefileを作成し，makeコマンドでコンパイルを実行できるようにしなさい．実行形のファイル名はbufcacheとする．

bufcacheを実行すると“$ ”というプロンプトを表示し，コマンド待ちの状態になるようにしなさい．コマンドの種類は以下のとおりとする．ここに挙げるコマンドに加えて独自のコマンドを作ってもよい．文字列操作に関しては第3章を参照するとよい．

- **help**
  ヘルプを表示する．

- **init**
  ハッシュリストやフリーリスト初期化し，図 2.15 の状態にする．

- **buf** [n ...]
  引数が無い場合はすべてのバッファの状態を表示する．引数が指定された場合は，バッファの番号 (バッファに関連付けられている論理ブロック番号ではない) が n のものの状態を表示する．

- **hash** [n ...]
  引数が無い場合はすべてのハッシュリストを表示する．引数が指定された場合は，ハッシュ値が n のハッシュリストを表示する．

- **free**
  フリーリストを表示する．

- **getblk** n
  論理ブロック番号 n を引数として `getblk(n)` を実行する．

- **brelse** n
  論理ブロック番号 n に対応するバッファヘッダへのポインタ bp を引数として `brelse(bp)` を実行する．

- **set** n $stat$ [$stat$ ...]
  論理ブロック番号 n のバッファについて状態 $stat$ をセットする．

- **reset** n $stat$ [$stat$ ...]
  論理ブロック番号 n のバッファについて状態 $stat$ をリセットする．

- **quit**
  このソフトウェアを終了する．

各コマンドの実行結果は以下のようにすること．

help

各コマンドのシンタックスと機能を英語で表示すること．

init

ハッシュリストやフリーリスト初期化し，図 2.15 の状態にする．バッファの状態 (`stat`) は以下のことを反映するように設定すること．この他，バッファがフリーリスト上にあるかどうかも考慮すること．

- すべてのバッファには有効なデータがキャッシュされている．

- すべてのバッファのデータには書き込みは行われていない．

- すべてのバッファのデータに対してカーネルは read/write は実行していない.
- すべてのバッファに対して，このバッファがフリーになるのを待っているプロセスは存在しない.

buf [n ...]

バッファの状態を以下のように表示する．なお，この例は初期化状態で buf コマンドを引数無しで実行したときの表示例である．一番左の数字 (0〜11) はバッファの番号である．次の数字はそのバッファに関連付けられている論理ブロック番号である．その次はバッファの状態をビットごとに表しており，左 (上位ビット) から STAT_OLD(“O”)，STAT_WAITED(“W”)，STAT_KRDWR(“K”)，STAT_DWR(“D”)，STAT_VALID(“V”)，STAT_LOCKED(“L”) とする．そのビットがセットされているときは対応するアルファベットを表示し，セットされていないときは “-” を表示する.

```
[ 0: 28 ----V-]
[ 1:  4 ----V-]
[ 2: 64 ----VL]
[ 3: 17 ----VL]
[ 4:  5 ----V-]
[ 5: 97 ----V-]
[ 6: 98 ----VL]
[ 7: 50 ----VL]
[ 8: 10 ----V-]
[ 9:  3 ----V-]
[10: 35 ----VL]
[11: 99 ----VL]
```

hash [n ...]

ハッシュリストを以下のように表示する．なお，これは初期化後に hash コマンドを引数無しで実行した表示例である．各行の左端の数字 (0〜3) はハッシュ値である.

```
0: [ 0: 28 ----V-] [ 1:  4 ----V-] [ 2: 64 ----VL]
1: [ 3: 17 ----VL] [ 4:  5 ----V-] [ 5: 97 ----V-]
2: [ 6: 98 ----VL] [ 7: 50 ----VL] [ 8: 10 ----V-]
3: [ 9:  3 ----V-] [10: 35 ----VL] [11: 99 ----VL]
```

free

フリーリストを以下のように表示する．なお，これは初期化後に free コマンドを実行した表示例である (紙面の都合で 2 行になっているが，実際には 1 行).

```
[ 9:  3 ----V-] [ 4:  5 ----V-] [ 1:  4 ----V-] [ 0: 28 ----V-]
[ 5: 97 ----V-] [ 8: 10 ----V-]
```

getblk n

論理ブロック番号 n を引数として取り，`getblk(n)` を実行する．`getblk()` の実行に際しては，どのシナリオ (1〜5) が実行されたかを表示すること．

brelse n

論理ブロック番号 n を引数として取り，指定された論理ブロック番号 n に対応したバッファヘッダへポインタ bp を引数として `brelse(bp)` を実行する．`brelse()` の実行に際しては，指定されたバッファがフリーリストのどこに挿入されたかを表示すること．

set n *stat* [*stat* **...**]

論理ブロック番号 n と 1 つ以上の状態 *stat* を引数としてとり，指定された論理ブロック番号 n に対応したバッファに状態 *stat* を設定する．*stat* は以下のようにアルファベット 1 文字で表すとする．

- L: STAT_LOCKED (locked)
- V: STAT_VALID (contain valid data)
- D: STAT_DWR (delayed write)
- K: STAT_KRDWR (kernel read/write)
- W: STAT_WAITED (process is waiting)
- O: STAT_OLD (data is old)

reset n *stat* [*stat* **...**]

論理ブロック番号 n と 1 つ以上の状態 *stat* を引数として取り，論理ブロック番号 n に対応したバッファの状態 *stat* をリセットする．*stat* の表し方は set コマンドと同様とする．

注意事項

プログラムの実行確認の際には以下の点に注意するとよい.

- コマンド入力のエラーを認識できるか.
- 各シナリオの動作後，バッファの状態は正しいか.
- scenario 3 を実行した際，scenario 3 と scenario 2 が実行されているか.
- scenario 3 の実行後，最初 DWR 状態であったバッファの状態は正しいか.
- scenario 3 の実行後，最初 DWR 状態であったバッファの状態を set コマンドあるいは reset コマンドで HDD への書き戻しが終わった状態にし，その後 brelse コマンドでこのバッファを解放したとき，正しい動作をするか.
- scenario 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.

練習問題

1. 第2.3.5節(図2.7)で解説した関数`void insert_head(struct buf_header *h, struct buf_header *p)`をC言語で記述しなさい.

2. 上記に関連し,`void insert_tail(struct buf_header *h, struct buf_header *p)`をC言語で記述しなさい.

3. 上記2問の関数を1つにまとめた`void insert_hash(struct buf_header *h, struct buf_header *p, int where)`をC言語で記述しなさい.なお,`int where`で先頭か末尾かを指定することとする.

4. 第2.3.6節(図2.8)で解説した関数`void remove_hash(struct buf_header *p)`をC言語で記述しなさい.

5. 第2.3.8節(図2.12)を参考にして,`p->stat`の`STAT_LOCKED`ビットと`STAT_VALID`ビットの両方を一度にセットする操作をC言語で記述しなさい.

6. 第2.3.8節(図2.12)を参考にして,`p->stat`の`STAT_LOCKED`ビットと`STAT_VALID`ビットの両方を一度にリセットする操作をC言語で記述しなさい.