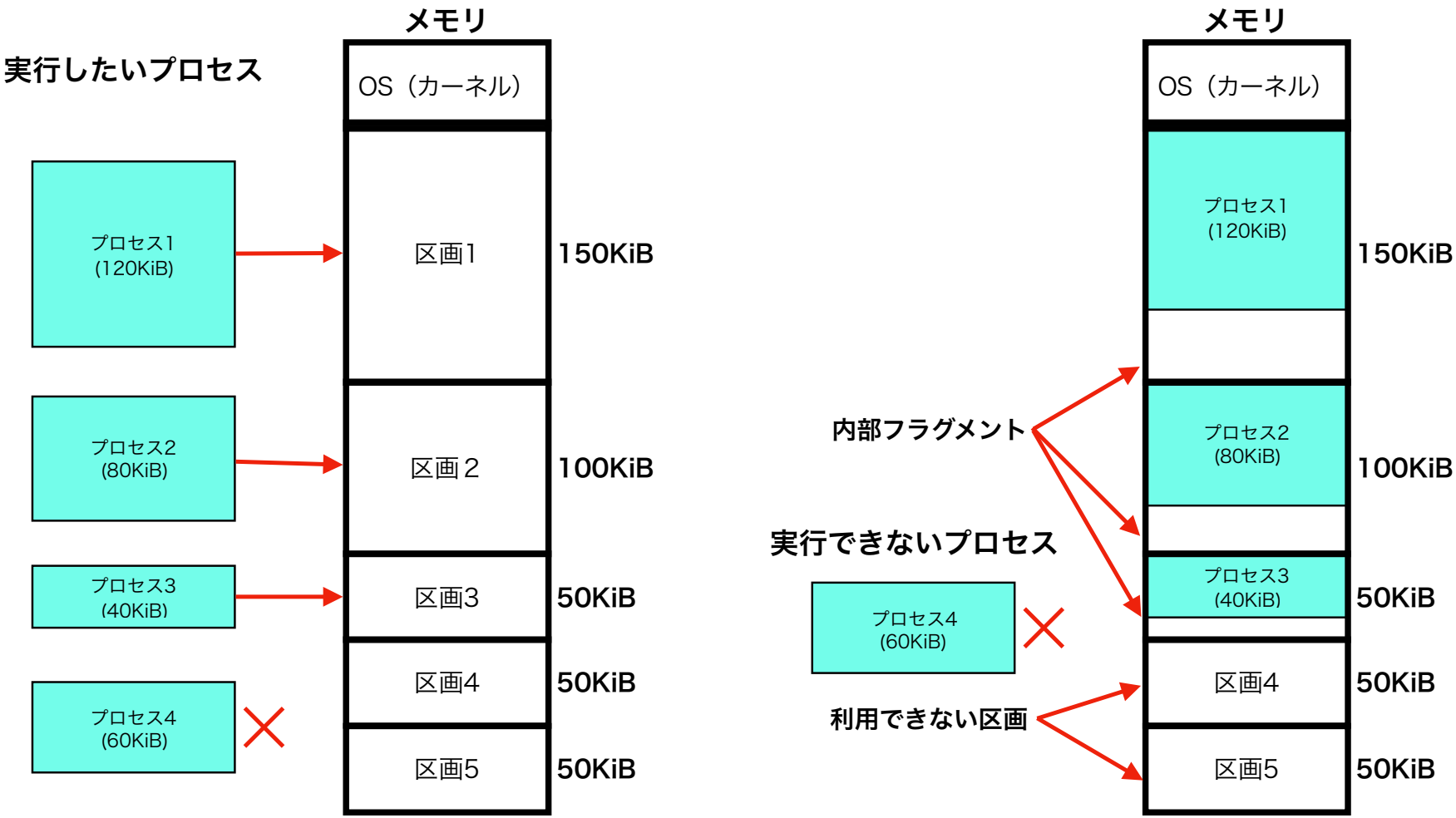
オペレーティングシステム
第９章 メモリ割付け方式

https://github.com/tctsigemura/OSTextBook

固定区画方式



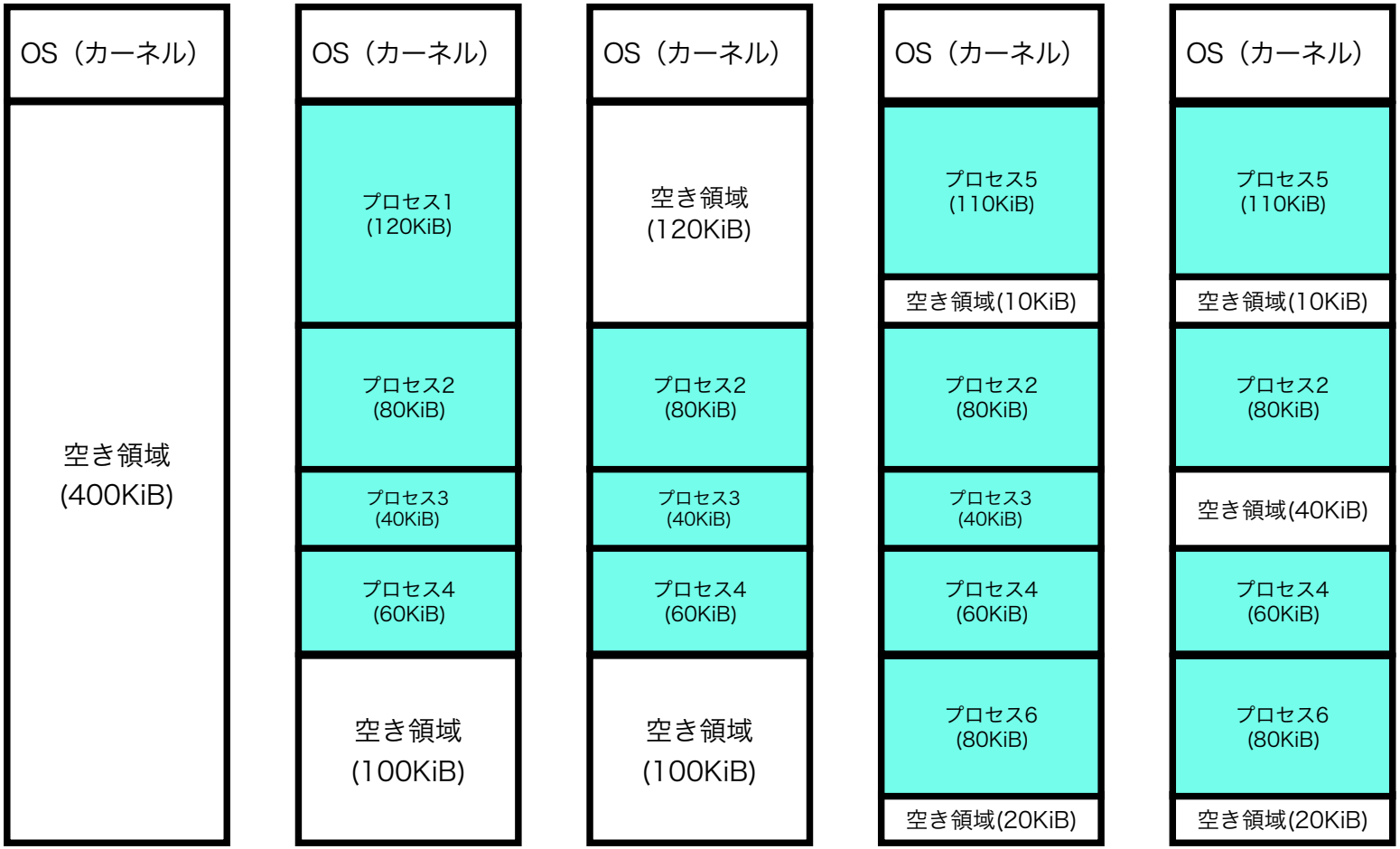
(a) 区画を選択しプロセスをロード　　(b) プロセスを実行

予めメモリを大小数種類の区画に分割しておく.

固定区画方式の特徴

1. 空き領域の管理が容易である.
2. 領域内部に無駄な領域（**内部フラグメント**）が生じる.
3. 小さな領域が複数空いていても大きなプロセスは実行できない.
4. 実行可能なプロセスのサイズに強い制約がある.
   （図の例では，151KiB のプロセスは実行できない.）
5. 同時に実行できるプロセスの数に制約がある.
   （図の例では，同時に五つ以上のプロセスは実行できない.）

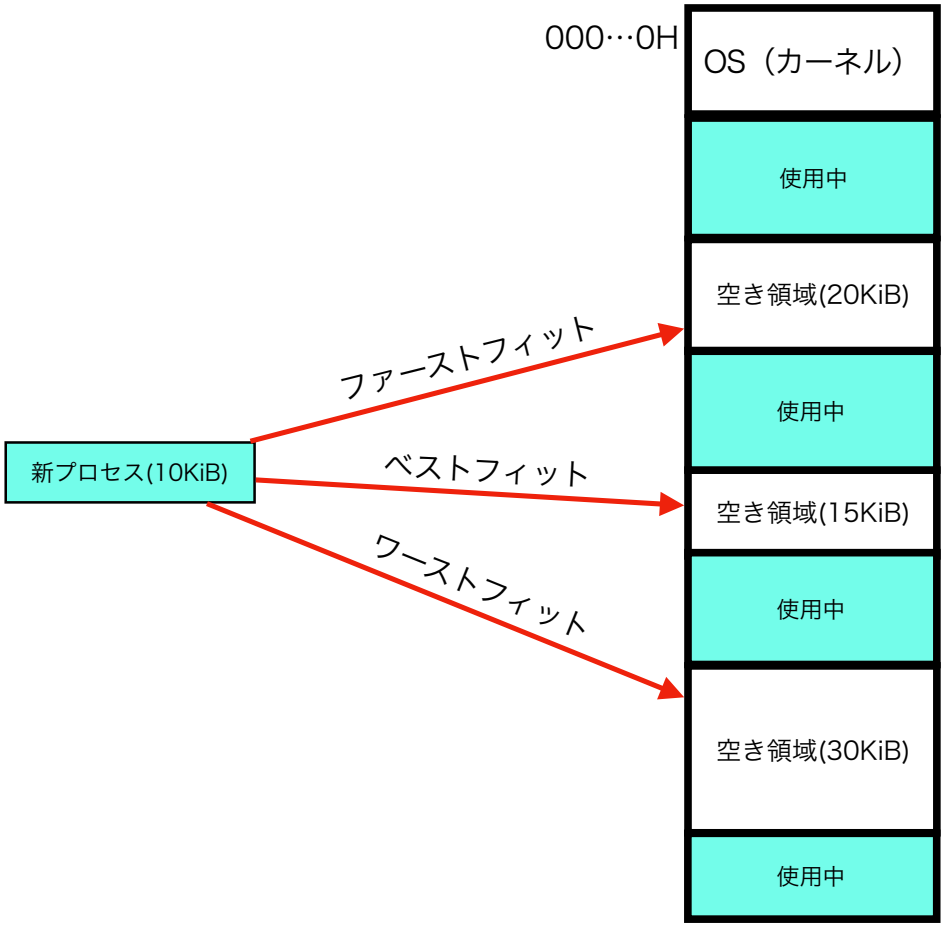
可変区画方式



(a) 初期状態　(b) 実行開始　(c) P1終了　(d) P5，P6開始　(e) P3終了

必要に応じて空き領域から区画を作る．
外部フラグメントが生じる．

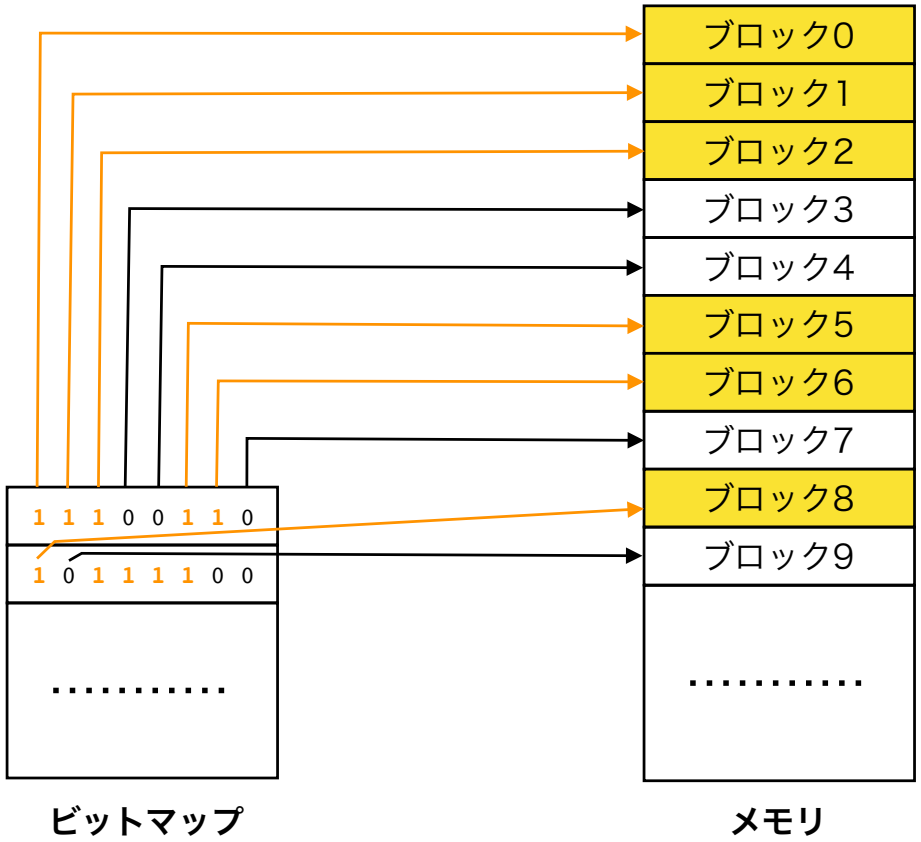
可変区画方式の領域選択方式



- ファーストフィット（first-fit）方式：アドレス順にさがす.
- ベストフィット（best-fit）方式：最小の領域を選択する.
- ワーストフィット（worst-fit）方式：最大の領域を選択する.

空き領域の管理方式（ビットマップ方式）

どこに利用可能な空き領域があるかビットマップで管理する．



- メモリを一定の大きさのブロックに分割する．
- ビットマップの 1 ビットが 1 ブロックに対応する．

ビットマップの大きさ

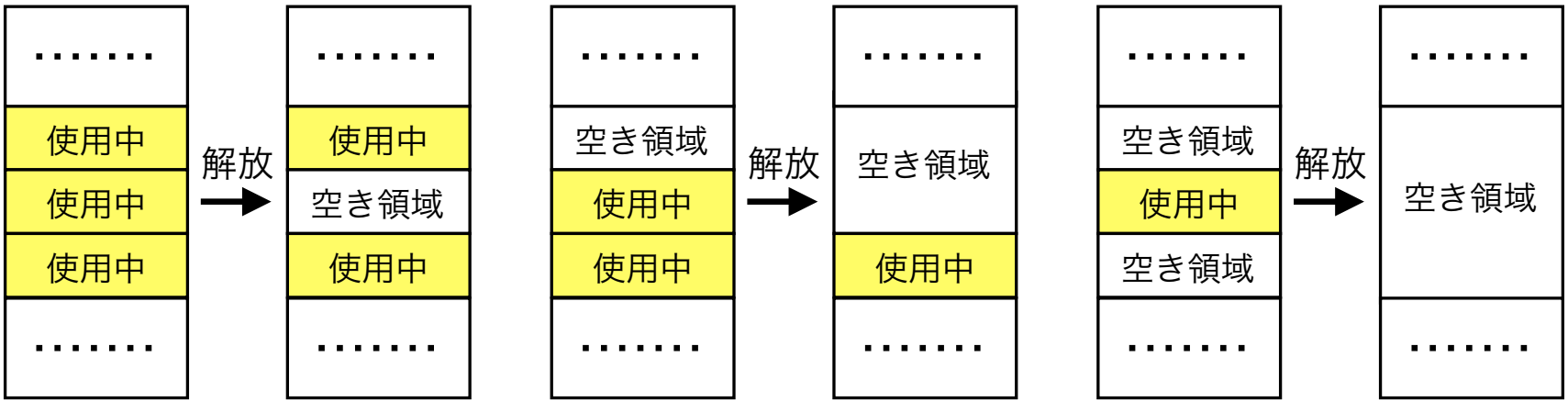
ビットマップの大きさを計算してみる.

- メモリサイズ：8GiB
- ブロックサイズ：4KiB
- ブロック数：$8GiB \div 4KiB = (8 \times 2^{30}) \div (4 \times 2^{10}) = 2 \times 2^{20} = 2Mi$
- ビットマップのサイズ：$(2 \times 2^{20}) \div 8 = 2^{18} = 256KiB$

無視できるほど小さくはない.
ビットマップを小さくするにはブロックサイズを大きくすればよい.
内部フラグメントが大きくなる.

空き領域の管理方式（リスト方式）

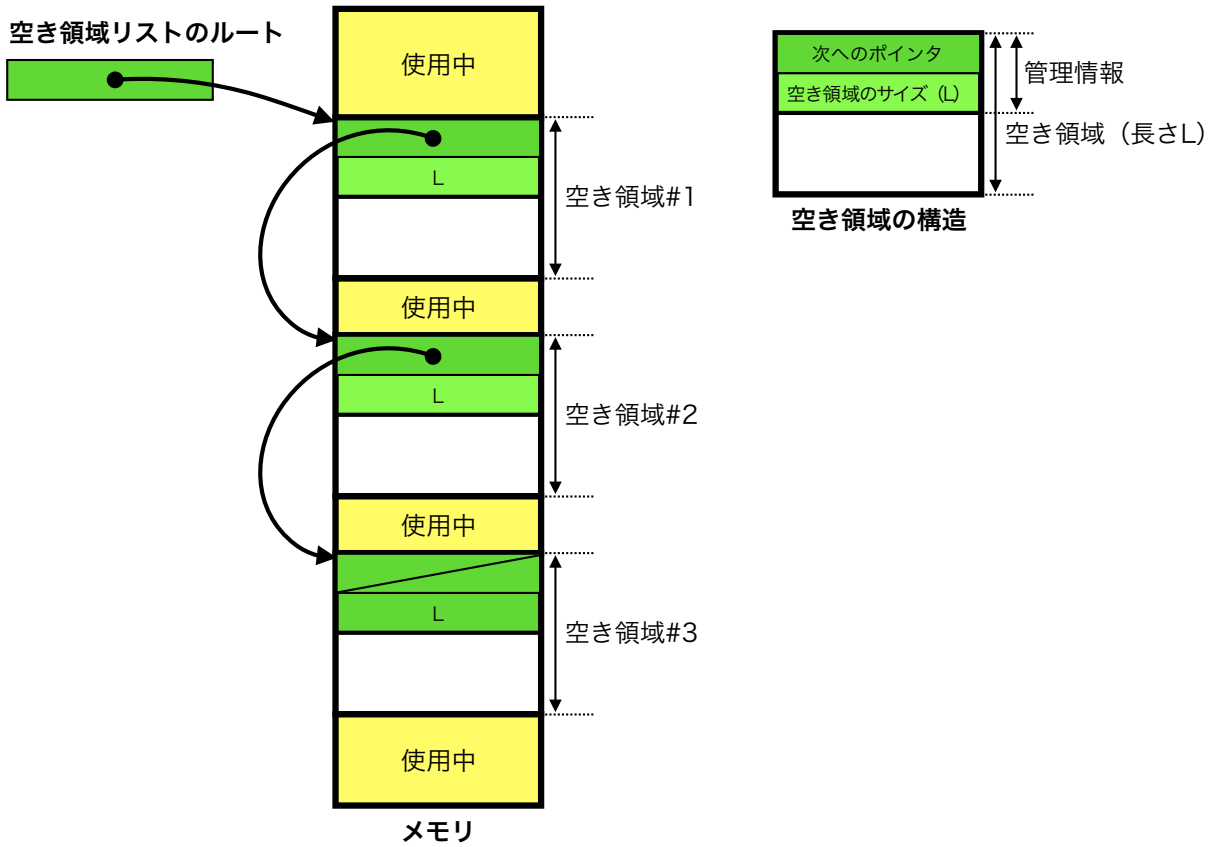
空き領域をリストにして管理する.
様々なサイズの空き領域が混在しても良い.



(a) 前後が使用中の場合　(b) 片側が空き領域の場合　(c) 両側が空き領域の場合

使用中の領域が解放されると隣接する空き領域と合体させる.

空き領域リストのデータ構造



- 空き領域の一部を管理データの格納に使用する.
- アドレス順のリストにして管理する.
- ファーストフィットの探索に都合が良い.
- 隣接領域との合体にも都合が良い.

メモリ管理の実装例

TacOS のメモリ管理プログラムを実装例とする.

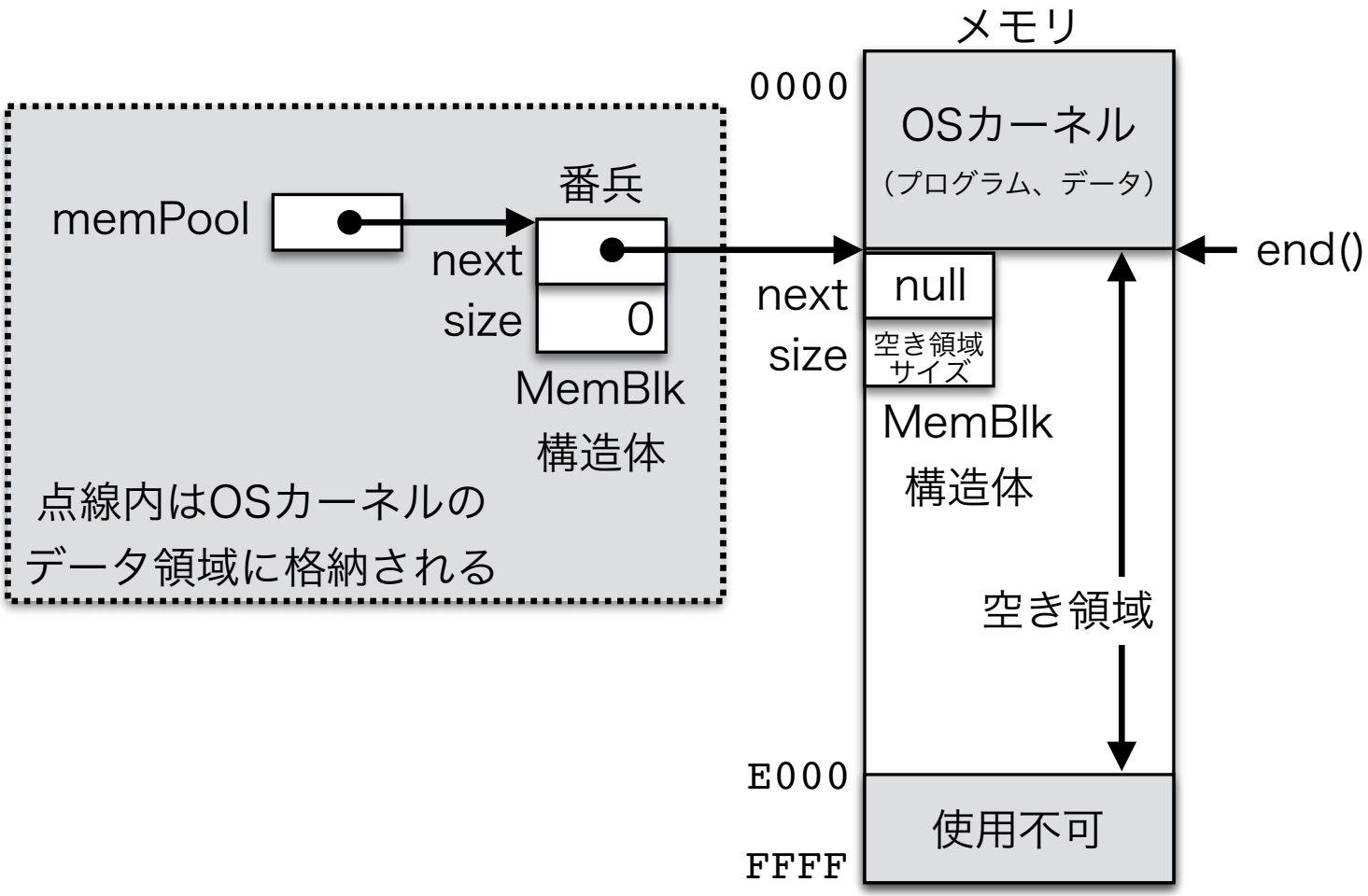
- 可変区画方式
- ファーストフィット
- OS がユーザプロセス領域の割当てに使用
- プロセスがヒープ領域を管理するプログラムも同じアルゴリズム

データ構造の初期化

```
1  #define MBSIZE sizeof(MemBlk)                    // MemBlk のバイト数
2  #define MAGIC  (memPool)                         // 番兵のアドレスを使用する
3
4  // 空き領域はリストにして管理される
5  struct MemBlk {                                  // 空き領域管理用の構造体
6    MemBlk next;                                   // 次の空き領域アドレス
7    int    size;                                   // 空き領域サイズ
8  };
9
10 //-------------------------------------------------------------------------
11 // 初期化ルーチン
12 //-------------------------------------------------------------------------
13 // メモリ管理の初期化
14 MemBlk memPool = {null, 0};                      // 空き領域リストの番兵
15 public int _end();                               // カーネルの BBS 領域の最後
16
17 void mmInit() {                                  // プログラム起動前の初期化
18   memPool.next = _ItoA(addrof(_end));            // 空き領域
19   memPool.next.size = 0xe000 - addrof(_end);     // 空きメモリサイズ
20   memPool.next.next = null;
21 }
```

- memInit() はカーネル起動時に一度だけ実行される.

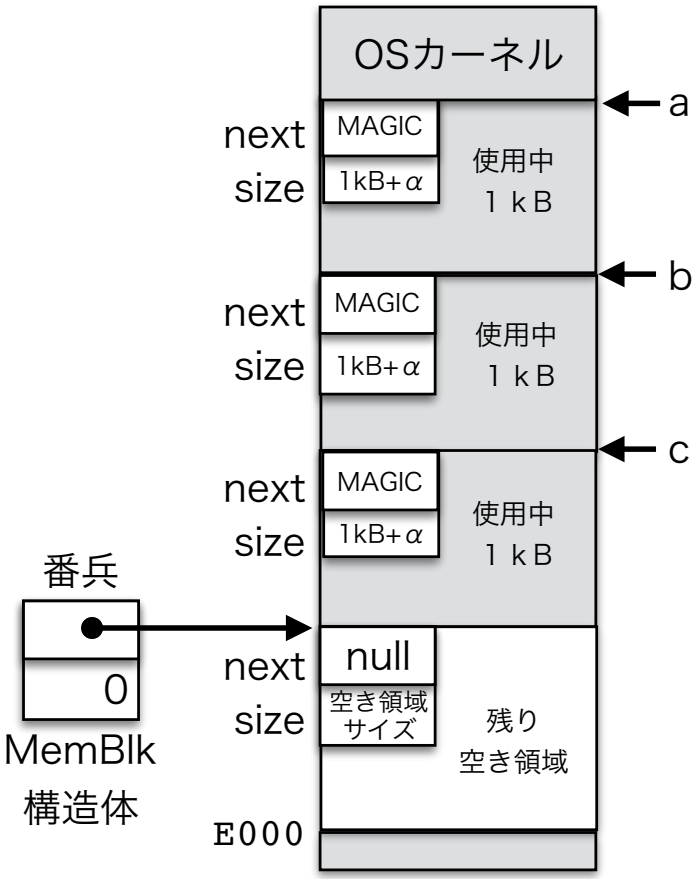
初期化直後のデータ構造



- `_end()` はカーネルサイズにより決まる.
- EOOO より後ろはビデオメモリや IPL ROM がある.

メモリの割付け

右のプログラムで a，b，c を割付けたときのデータ構造



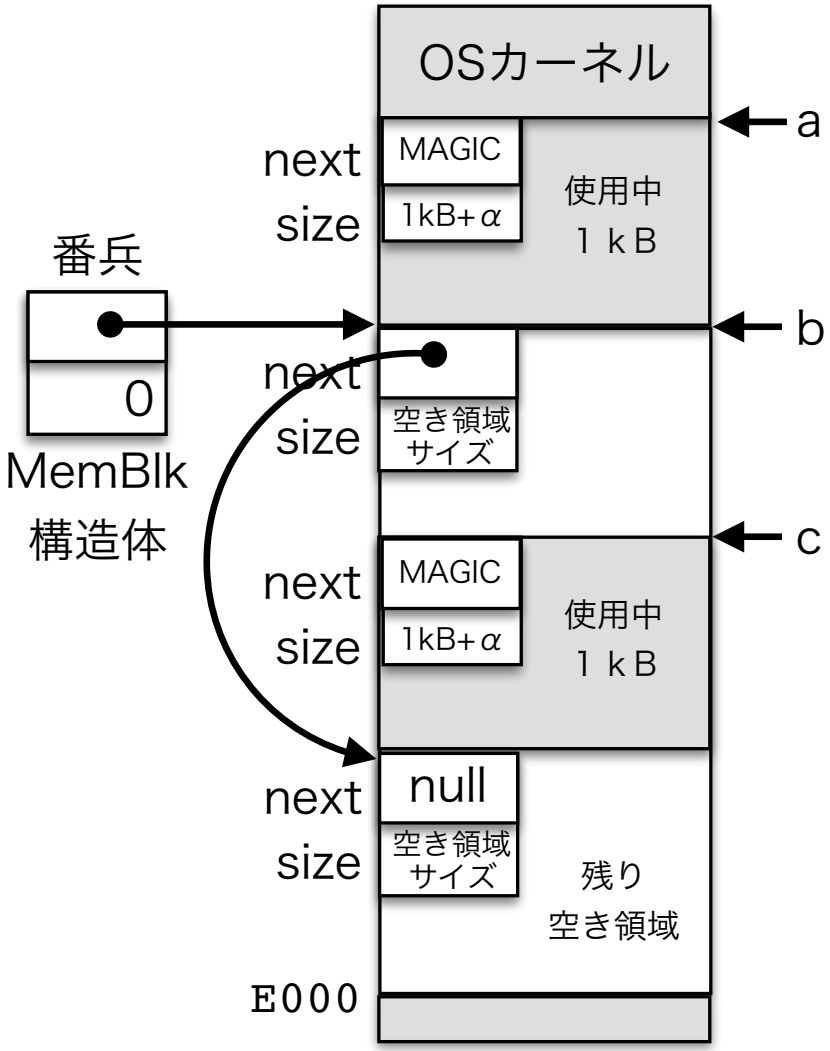

```
a = mmAlloc( 1024 );  // 1KiB の領域を割り付ける
b = mmAlloc( 1024 );  // 1KiB の領域を割り付ける
c = mmAlloc( 1024 );  // 1KiB の領域を割り付ける
```

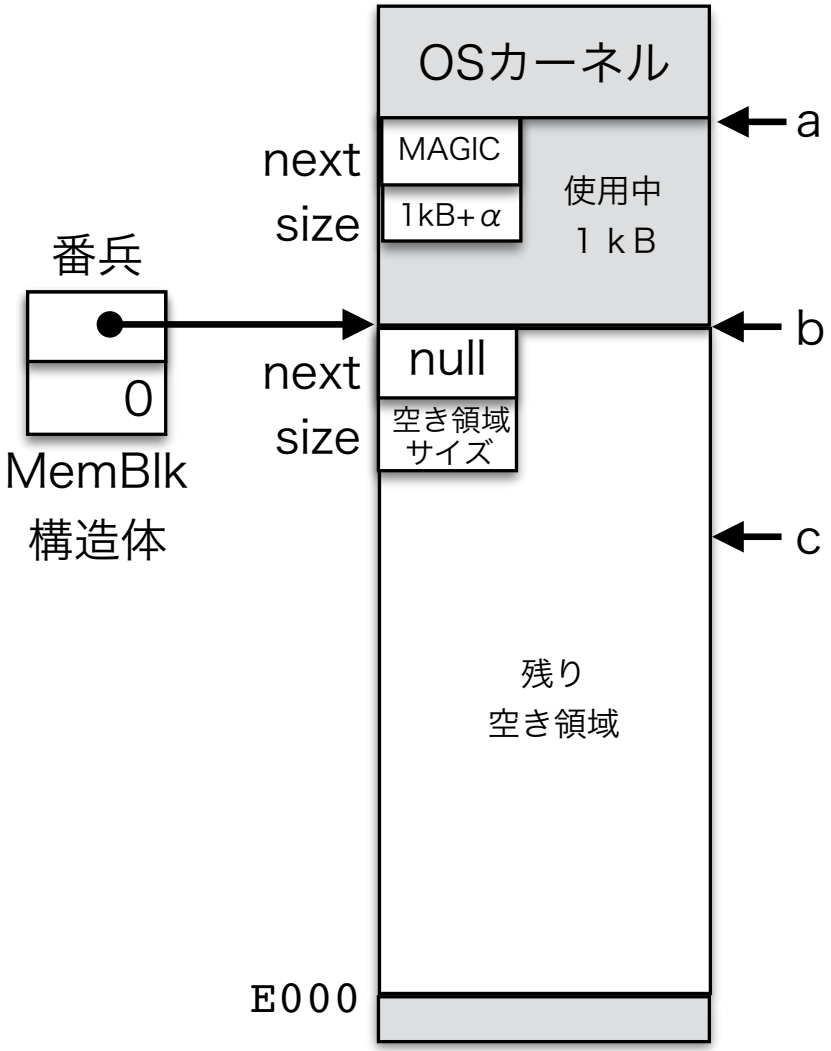
メモリの割付けプログラム

```
1  // メモリを割り付ける
2  int mmAlloc(int siz) {                          // メモリ割り当て
3    int s = (siz + MBSIZE + 1) & ~1;              // 制御データ分大きい偶数に
4    MemBlk p = memPool;                           // 直前の領域
5    MemBlk m = p.next;                            // 対象となる領域
6
7    while (_uCmp(m.size,s)<0) {                   // 領域が小さい間
8      p = m;                                      // リストを手繰る
9      m = m.next;
10     if (m==null) return 0;                      // メモリが不足する場合は
11   }                                             // エラーを表す null ポインタ
12
13   if (_uCmp(m.size ,s+MBSIZE+2)<=0) {           // 分割する価値がない領域サイズ
14     if (memPool.next==m && m.next==null)        // リストの長さがゼロにならない
15       return 0;                                 //   ようにする
16     p.next = m.next;                            // リストから外す
17   } else {                                      // 領域を分割する価値がある
18     MemBlk n = _addrAdd(m, s);                  // 残り領域
19     n.next = m.next;
20     n.size = m.size - s;
21     p.next = n;
22     m.size = s;
23   }
24   m.next = MAGIC;                               // マジックナンバー格納
25   return _AtoI(_addrAdd(m, MBSIZE));            // 管理領域を除いて返す
26 }
```

領域の解放

b，c を開放したときのデータ構造





メモリの解放プログラム（前半）

```
1  // メモリを解放する
2  int mmFree(void[] mem) {                          // 領域解放
3    MemBlk q  = _addrAdd(mem, -MBSIZE);             // 解放する領域
4    MemBlk p = memPool;                             // 直前の空き領域
5    MemBlk m = p.next;                              // 直後の空き領域
6
7    if (q.next!=MAGIC)
8      badaddr();                                    // 領域マジックナンバー確認
9
10   while (_aCmp(m, q)<0) {                         // 解放する領域の位置を探る
11     p = m;
12     m = m.next;
13     if (m==null) break;
14   }
15
16   void[] ql = _addrAdd(q, q.size);                // 解放する領域の最後
```

- 領域の本当の先頭アドレスを計算する．
- MAGIC を確認する．
- 空き領域リストを辿り，挿入位置を決める．

メモリの解放プログラム（後半）

```
17    void[] pl = _addrAdd(p, p.size);                    // 直前の領域の最後
18
19    if (_aCmp(q,pl)<0 || m!=null&&_aCmp(m,ql)<0)        // 未割り当て領域では？
20      badaddr();
21
22    if (pl==q) {                                        // 直前の領域に隣接している
23      p.size = p.size + q.size;
24      if (ql==m) {                                      // 直後の領域とも隣接してる
25        p.size = p.size + m.size;
26        p.next = m.next;
27      }
28    } else if (ql==m) {                                 // 直後の領域に隣接している
29      q.size = q.size + m.size;
30      q.next = m.next;
31      p.next = q;
32    } else {
33      p.next = q;
34      q.next = m;
35    }
36    return 0;
37  }
```

練習問題

可変区画方式で管理される 100KiB の空き領域がある時，次の順序で領域の割付け解放を行った．ファーストフィット方式を用いた場合とベストフィット方式を用いた場合について，実行後のメモリマップを図示しなさい．

1. 30KiB の領域を割付け
2. 40KiB の領域を割付け
3. 20KiB の領域を割付け
4. 先程割付けた 40KiB の領域を解放
5. 10KiB の領域を割付け