オペレーティングシステム

第5章 仮想記憶システム

情報工学科（担当：阿部，鷹合）

資料作成日：September 21, 2022

テキスト：古市栄治「ｵﾍﾟﾚｰﾃｨﾝｸﾞｼｽﾃﾑ入門」，日本理工出版会，2018
（資料はテキストに沿って作成し，直接引用した箇所には📄を付記した）

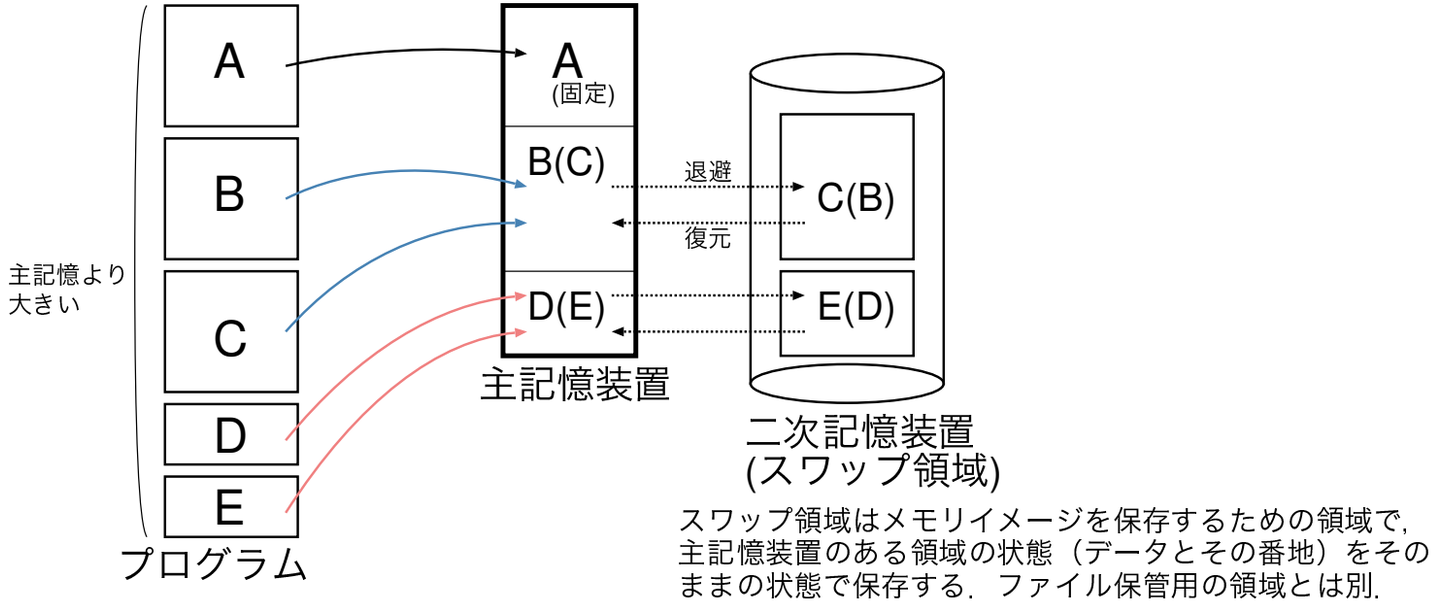
KIT 金沢工業大学

目次

5.1 主記憶の管理

❶ オーバレイ構造

- 仮想記憶が登場するまで使われていた方式
- プログラムをセグメント単位に分割し，複数のセグメントで主記憶の同じ領域を共有する構造
- 複数のセグメントで主記憶を共用することで，プログラム全体で使用する主記憶を節約する
- セグメント切り替え時に二次記憶装置が使われる（退避と復元）



スワップ領域はメモリイメージを保存するための領域で，主記憶装置のある領域の状態（データとその番地）をそのままの状態で保存する．ファイル保管用の領域とは別．

❷ 複数のプログラムの割付け

マルチプログラミングを実現するためには，複数のプログラムを同時に主記憶上に配置することが必要．

(1) 固定区画割付け

主記憶を予め大きさを固定した区画に分割しておき，ジョブを最適な区画に割り当てて実行させるという最も単純な方法📄

問題点 区画内で使われない領域ができる

問題点 最大区画より大きなジョブは実行できない

問題点 同時に実行可能なジョブは区画の数まで

(2) 可変区画割付け

主記憶を予め区画に分けることをせず，ジョブの実行開始時に必要な大きさの領域を切り出してジョブに割り当てる方式．📄

問題点 実行するジョブが増えてくるに従って，主記憶上に小さな空き領域が多数できる（**フラグメンテーション**）．

問題点 各ジョブの配置場所を移動させることで大きな空き領域を確保できる（**コンパクション**）が，その間は全プログラムが休止．

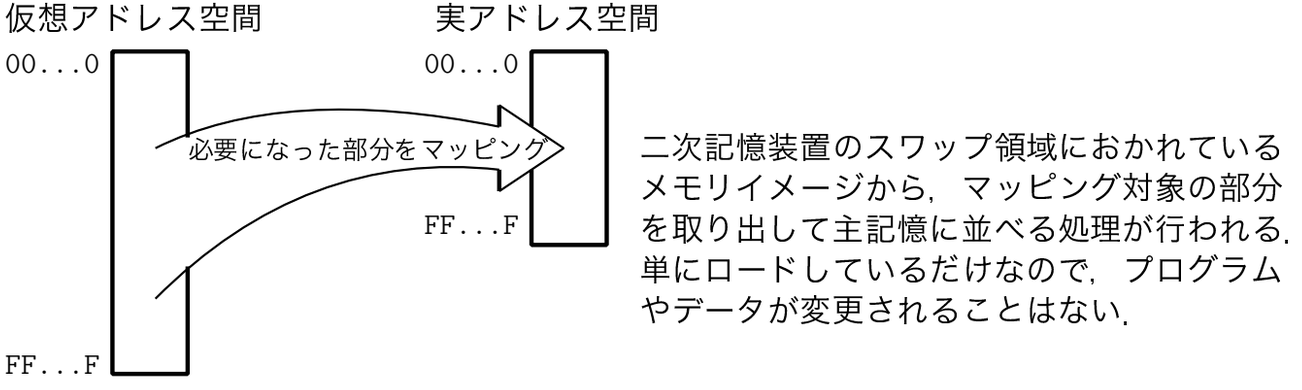
5.2 仮想記憶の仕組み

仮想記憶は**主記憶装置の実際の容量を気にせずにプログラムを動かす**ための仕組みで，IBM System/370で実用化された(1970).

❶ 仮想記憶の概念

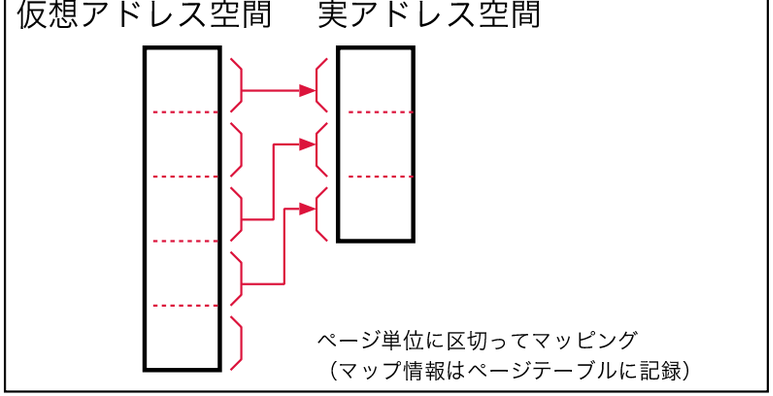
- 広大なアドレス空間（**仮想アドレス空間**）をプログラムに提供する.
  - コンパイラ（正確にはリンカ）は，仮想アドレス空間を想定してプログラムをレイアウト（機械語命令やデータの配置を決める）.
  - CPUも仮想記憶装置上のプログラムを逐次実行していく（ｽﾗｲﾄﾞ6）.
  - 利用者・開発者は次の2点を考える必要はない.
    ①仮想記憶の方式（ページングやセグメンテーション）
    ②主記憶装置のアドレス空間（**実アドレス空間**）の大きさ
- 仮想記憶以前は主記憶装置を気にしながらプログラムを開発・実行
  - **オーバレイ**は大きなプログラムを分割して，小さなアドレス空間に収めるためのテクニック
  - **固定区画割付け**は，区画サイズより大きなプログラムは動かせない

❷プログラム実行のメカニズム　プログラム実行時には，参照したい**「仮想アドレス空間**[1)]**の部分空間」を「実アドレス空間の部分空間」にマッピング**する処理が行われる．

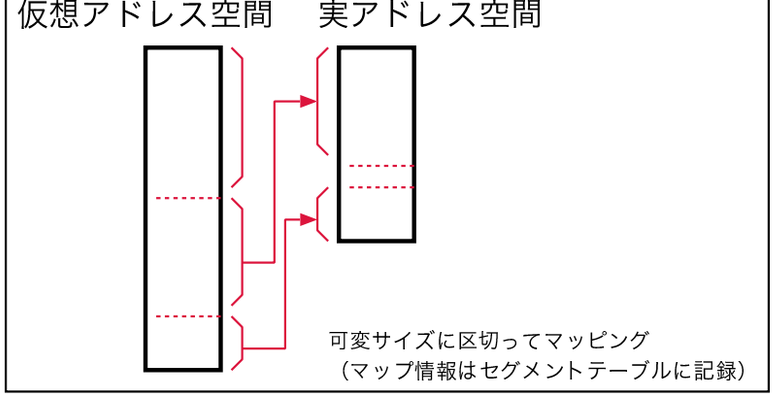


二次記憶装置のスワップ領域におかれているメモリイメージから，マッピング対象の部分を取り出して主記憶に並べる処理が行われる．単にロードしているだけなので，プログラムやデータが変更されることはない．

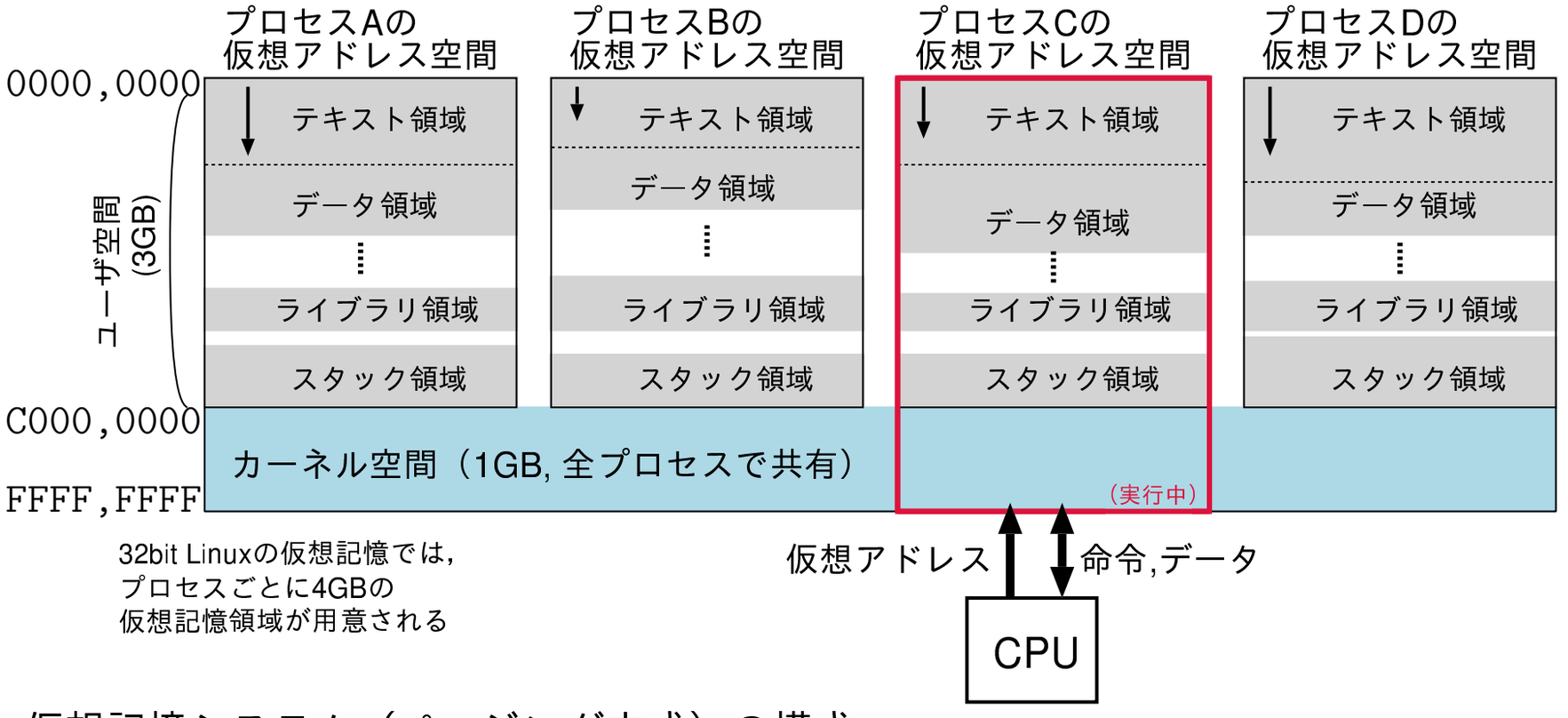
ページング方式の場合（主流）



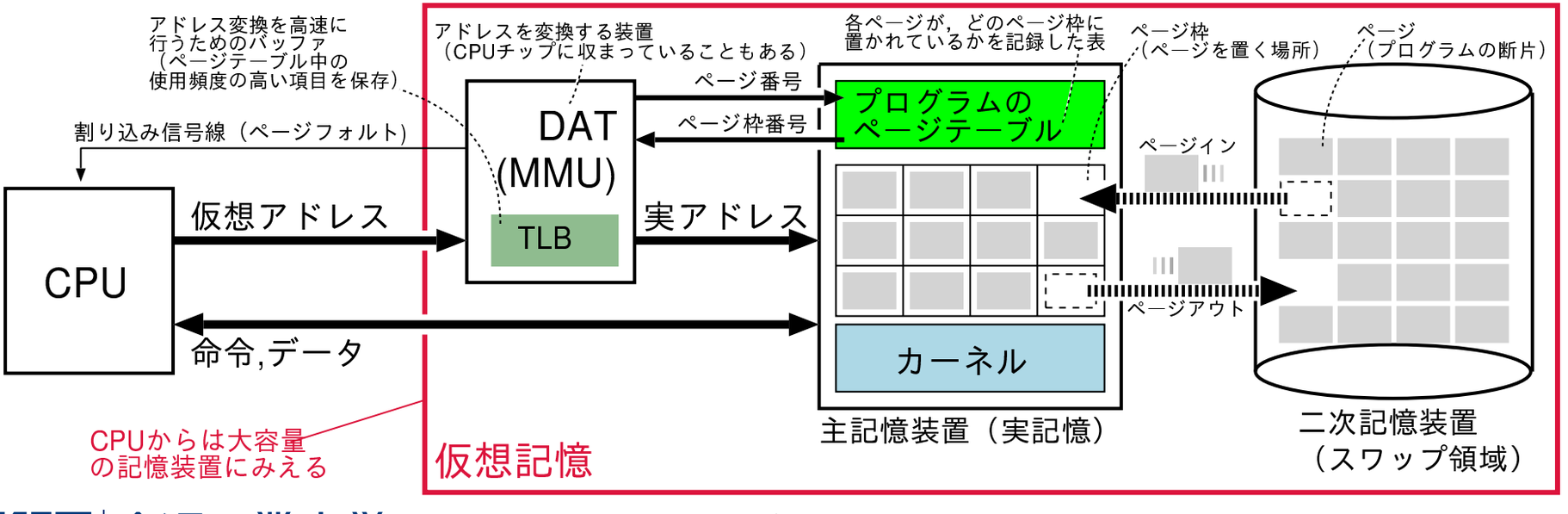
セグメンテーション方式の場合



教科書では「スワップ領域」を「仮想記憶」(Virtual Storage)と呼んでいるが，この科目では「スワップ領域」だけでなく「アドレス変換機構」等を含めた広範な部分を**「仮想記憶」**(Virtual Memory)と呼ぶことにする．



ー仮想記憶システム（ページング方式）の構成ー



1) **ページング**　プログラムのアドレス空間を**ページ**という固定サイズに区切り，ページを主記憶上の**ページ枠**(フレーム)にロードして使用する方式．

2) **ページイン**　スワップ領域のページを主記憶上に読み出すこと．

3) **ページアウト**　主記憶上のページ枠をあける為に，**ページ置き換えｱﾙｺﾞﾘｽﾞﾑ**によって選択されたページをスワップ領域に追い出すこと．

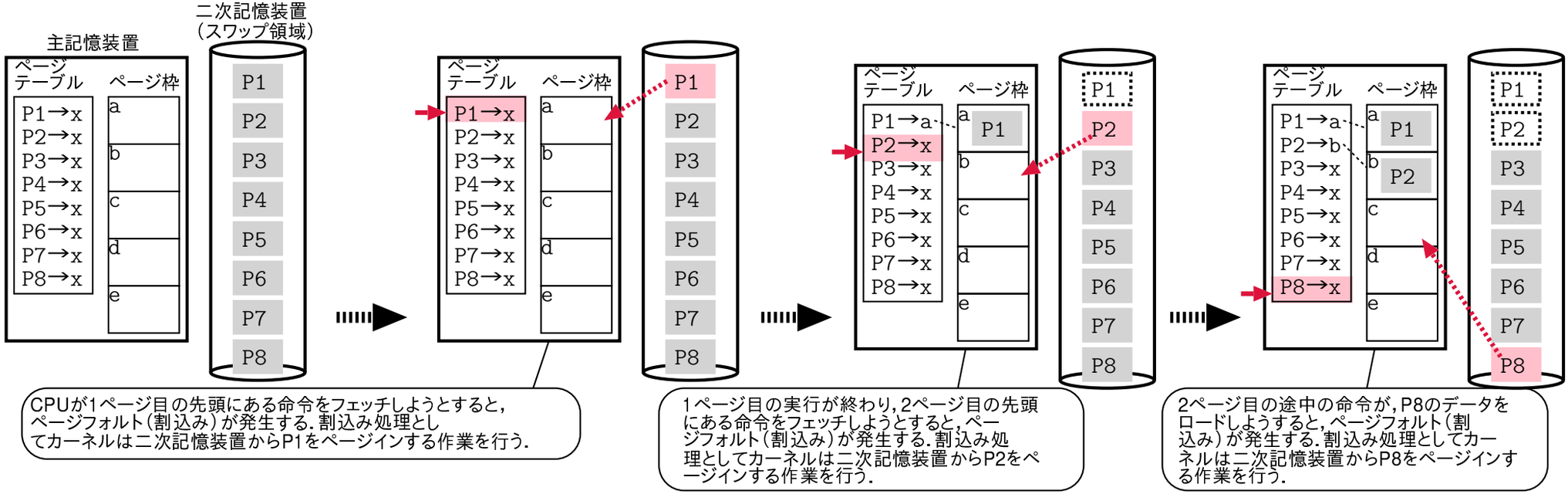
4) **ページフォルト（割込みの一つ）**　CPUが参照しようとしたページが主記憶上に見つからないこと．ページフォルトが起きると，カーネルは（必要に応じてページアウトしてから），ページインを行う．ページイン完了後はページフォルトを引き起こした命令から再開する．

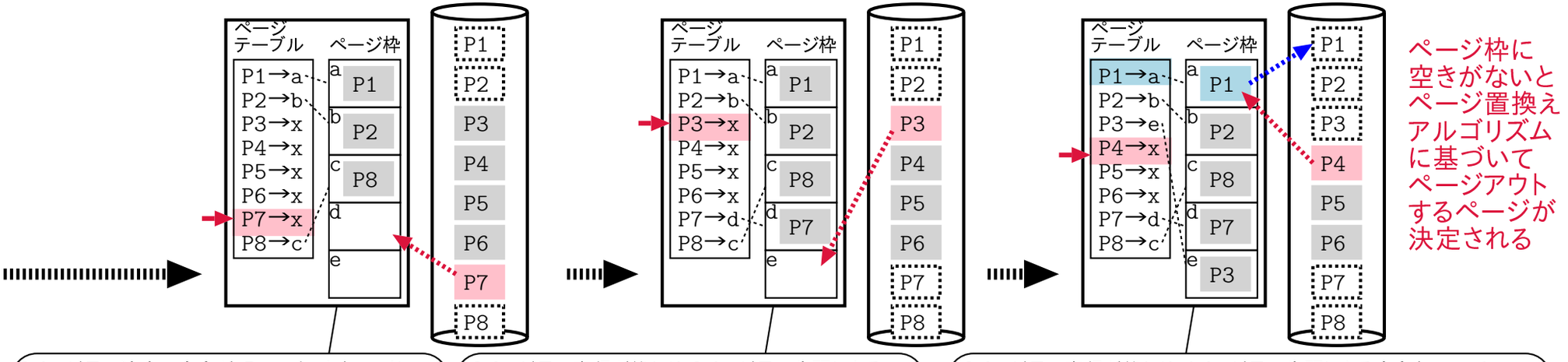
5) **ページテーブル**　各ページが，どのページ枠に置かれているかを記録した表．かなり大きな表となるため，主記憶装置上に置かれる．カーネルによるページインやページアウトに伴い更新される．

6) **スラッシング**　ページアウトとページインが多発し，実行が進みにくい状態に陥ること．

7) **DAT**　ﾍﾟｰｼﾞﾃｰﾌﾞﾙを参照し，仮想ｱﾄﾞﾚｽを実ｱﾄﾞﾚｽに変換する装置．

ページングの様子（テキストp117）





※ 実際にはP1の内容に変更がなかった場合はページアウトは不要．

演習①：システムの主記憶の使用状況を確認しよう．

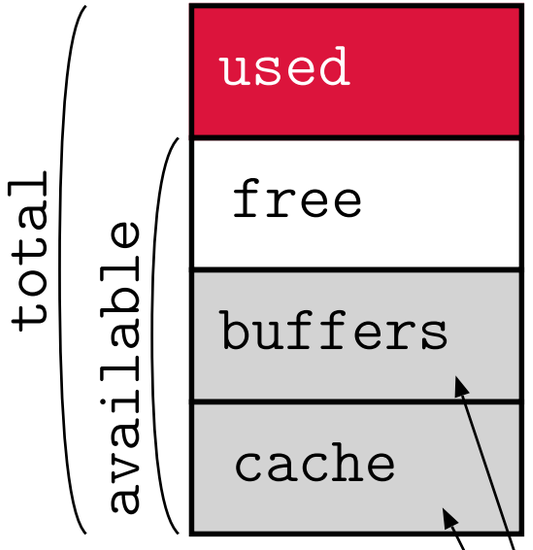
【端末1】

```
$ free -mw ↵   （次のスライドも参考に）
              合　計      使用中      未使用  仮想ﾌｧｲﾙｼｽﾃﾑ/共有ﾒﾓﾘ  ﾃﾞｨｽｸﾌﾞﾛｯｸなど  ﾍﾟｰｼﾞｷｬｯｼｭなど  新プロセスで利用可
              total       used       free       shared      buffers       cache   available
Mem:           1987        534        247            5          157        1048        1251
Swap:           947          0        947

$ ls -lh /swapfile ↵
-rw------- 1 root root 948M  7月 23 11:21 /swapfile

$ watch -n 0.1 "free -mw" ↵
```

（メモリ消費の多いfirefoxブラウザを立ち上げたり，find / などを実行してみる）

- "Mem"はPCに取り付けられている主記憶装置の容量．また，"Swap"はﾒﾓﾘ不足時に使う，ﾃﾞｨｽｸ上に確保されたﾒﾓﾘｲﾒｰｼﾞの退避場所のこと．
- "buffers"や"cache"は，ディスクとメモリの間でデータ転送があったときに自動的に消費される領域
- ディスクから取り出したファイルに再びアクセスするときは，ディスクではなく主記憶上のページキャッシュにアクセスすれば済むので，ディスクアクセス回数を削減できる．
- "buffers"や"cache"は専用のコマンドを使うと開放でき，"free"が増やせる．

freeコマンドの出力の見方



使用中の領域
("shared"も"used"の一部)

空き量域

ディスクドライブなどのブロック単位で
読み書きする装置用の入出力バッファとして使われる領域

ページ単位に区切られたファイルの断片(ページキャシュ),
メタ情報(inodeなど)が置かれる領域

プロセスから,ディスクやファイルにアクセスすることを要求された場合,
このメモリ領域を読み書きすれば済むので実際にはディスクアクセスを省略できる.
(また,メモリ不足時にはここを開放して,空き領域を増やすこともできる※1,2)

※1...それでも足りなければ,ディスク上のスワップ領域(またはスワップファイル)が使われる.
※2...コマンドで開放を指示することもできる.

【端末2】
```
$ sync ↵  ﾃﾞｨｽｸに書き出して無い情報があれば書き出す
$ sudo sh -c "echo 3 > /proc/sys/vm/drop_caches" ↵  ﾊﾞｯﾌｧ/ｷｬｯｼｭのｸﾘｱ
```

演習②：プロセスのページフォルトの回数を観察しよう．

【端末1】

```
$ watch -n 0.1 "ps -o pid,maj_flt,min_flt,vsz,rss,start,cmd -C xeyes"
```

PID	MAJFL (Major Page Fault)	MINFL (Minor Page Fault)	VSZ (Virtual Set Size)	RSS (Resident Set Size)	STARTED	CMD
2000	30	358	17092	5324	10:15:09	xeyes
2001	0	374	17092	5376	10:15:15	xeyes
2002	0	373	17092	5376	10:15:20	xeyes
2003	24	357	17092	5316	10:15:25	xeyes

----- 補足 -----

MAJFL：ディスクアクセスが必要なページフォルトの回数

MINFL：ディスクアクセスが不要なページフォルトの回数

（ﾍﾟｰｼﾞｷｬｼｭ上にﾍﾟｰｼﾞがあるので，ﾍﾟｰｼﾞﾃｰﾌﾞﾙの更新のみで済む）

VSZ：仮想記憶装置におけるメモリ使用量（KB）

RSS：実記憶装置（主記憶装置）における消費量（KB）

【端末2】

```
$ sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"   ﾊﾞｯﾌｧ/ｷｬｯｼｭのｸﾘｱ
$ xeyes &
$ xeyes &
$ xeyes &
$ sudo sh -c "echo 3 > /proc/sys/vm/drop_caches"   ﾊﾞｯﾌｧ/ｷｬｯｼｭのｸﾘｱ
$ xeyes &
```

ステップアップ

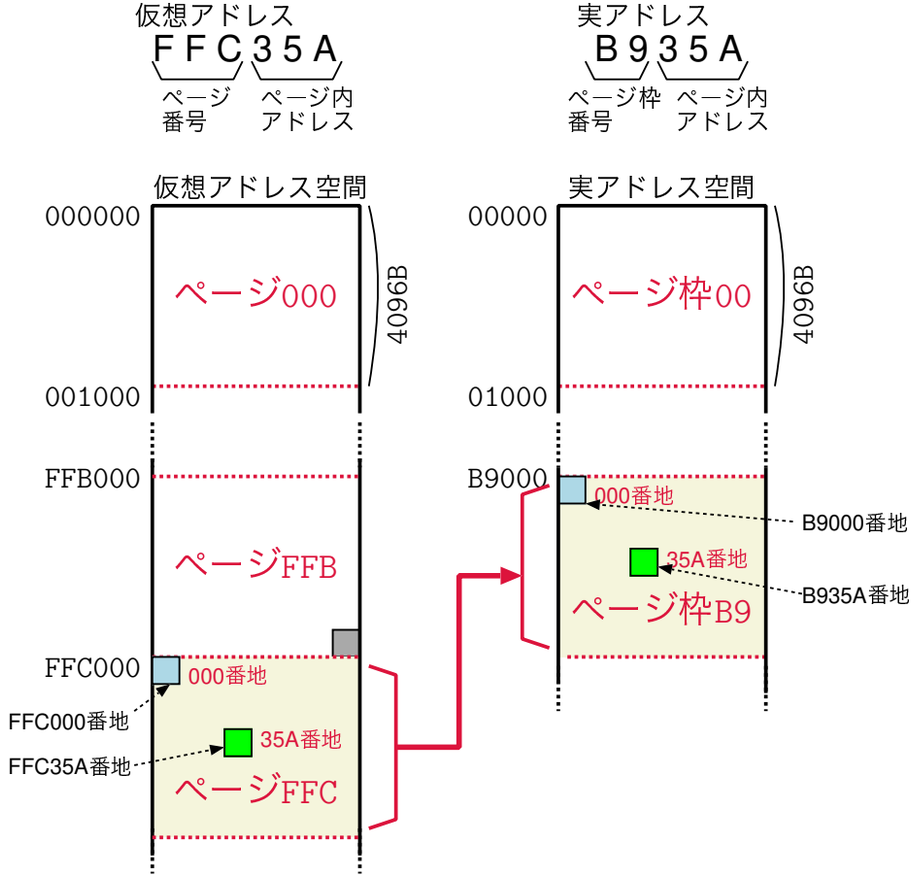
1. 各自Linuxで何らかの操作を行い，それに伴うメモリの使用状況の変化を`free`コマンドでモニタリングしなさい．

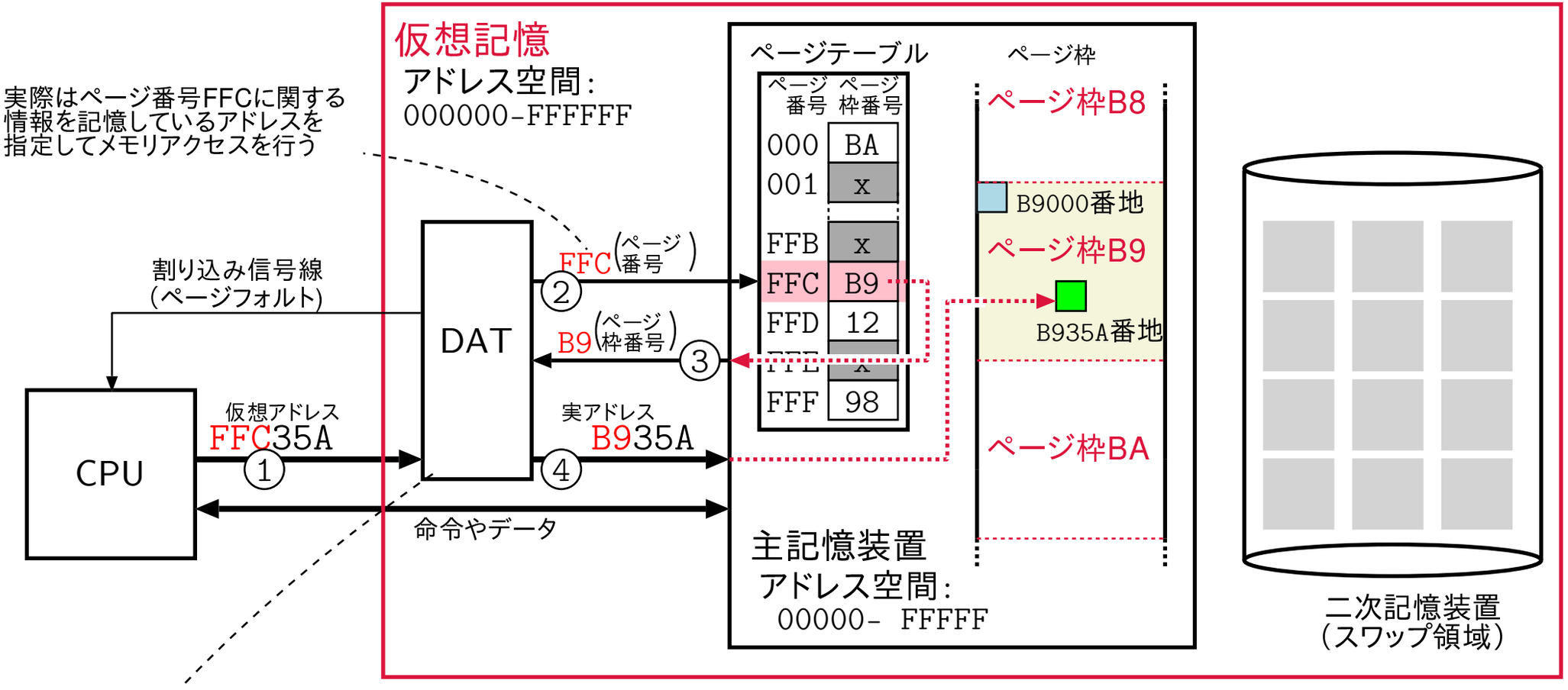
2. 演習②を行って得られた結果について考察してみなさい．

5.3 アドレス変換

❶ ❷ 仮想アドレス，実アドレス

アドレスの上位部分を**ページ番号（ページ枠番号）**として，下位の部分を**ページ内アドレス**として使う．ページテーブルにはページ番号とページ枠番号の対応付けを保存する．





実際のDATは，過去にアクセスしたページの情報をTLB(Translation Look-aside Buffer)という，並列に検索できる構造をとるキャシュメモリ(連想メモリ)に格納しているので，主記憶のページテーブルにアクセスする回数はずっと少なくなる

CPUが実行する命令の並びの例

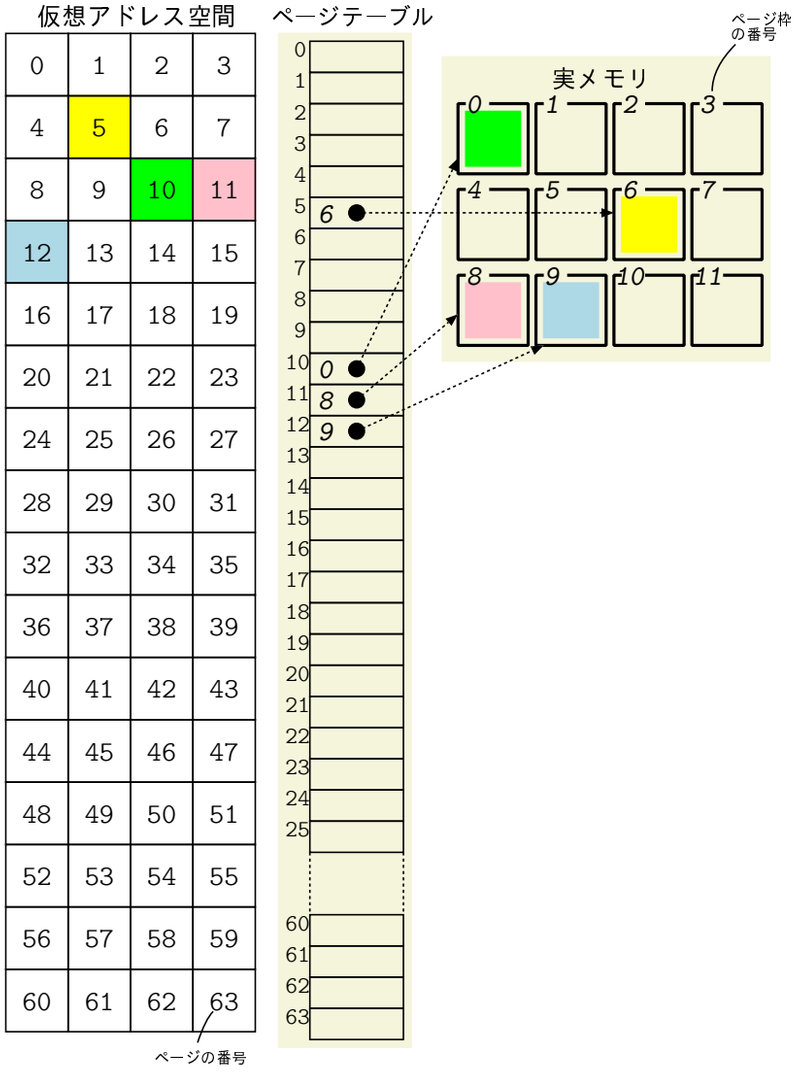
```
LDB R1,0xFFC35A    ; R1←[0xFFC35A]
STB R1,0xFFC000    ; R1→[0xFFC000]
STB R1,0xFFBFFF    ; R1→[0xFFBFFF]
```

仮想アドレスが比較的近い場所を参照していても，
ページが異なるのであれば，ページフォルトになることも当然ある

❸ アドレス変換の実際（マルチレベルページング）

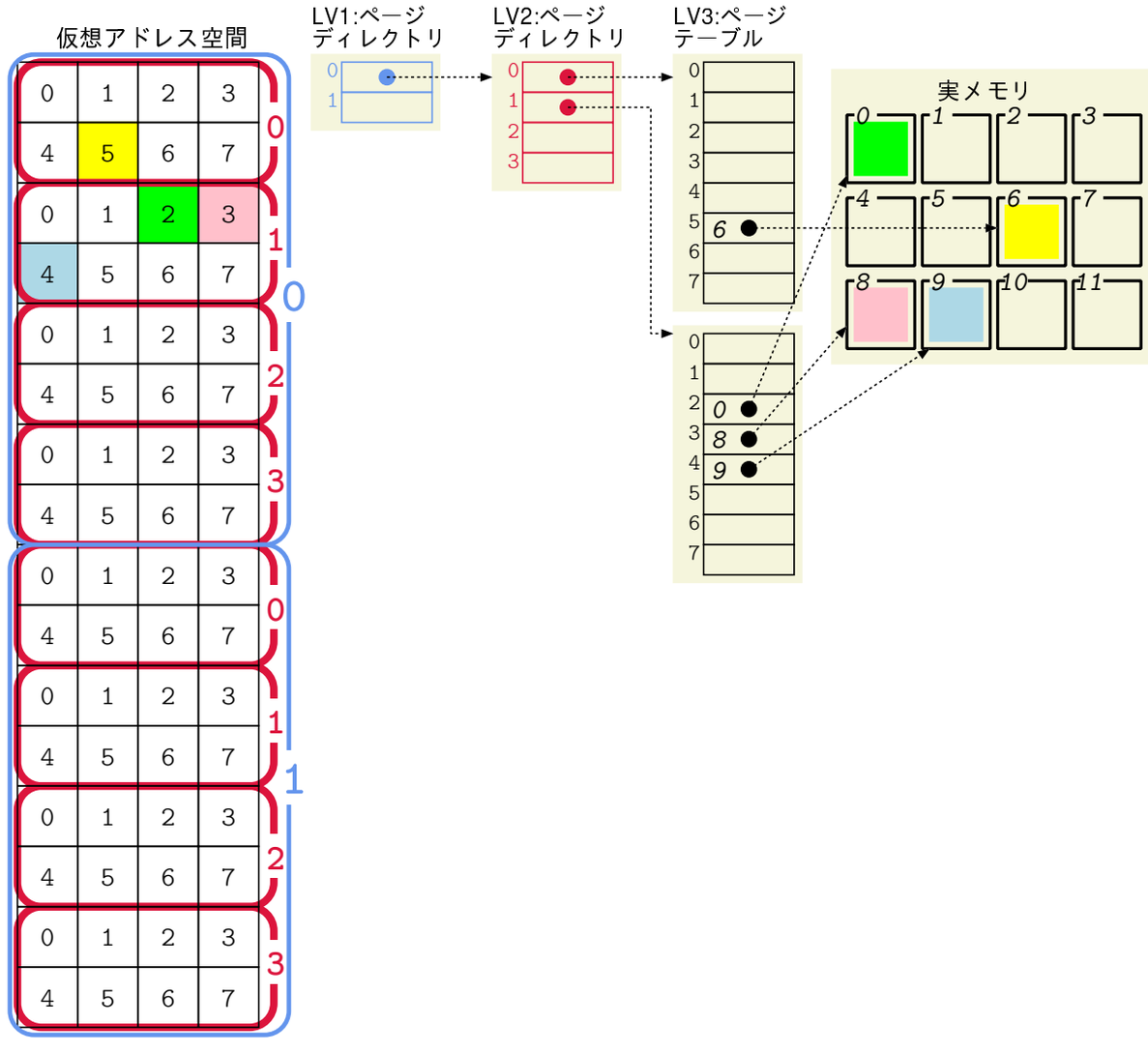
①巨大なページテーブルで主記憶領域を圧迫したくない，②仮想アドレス空間には使われない領域がある（未使用ページ多数），という点を踏まえて**マルチレベルページング**と呼ばれる方法が取られている．

- アドレス表現に32ビットが使われるとすると仮想アドレス空間は4GBになる．ページサイズを4KBとすると100万ページになる．
- 仮想アドレス空間を例えば**4MB単位（1024ページ分）の区画に分け，その区画内のみを管理するページテーブルを用意**したとすると，1024個のページテーブルを用意すれば4GB全体をカバーできる．全体のページ数としてはやはり100万ページであることにはかわらないが，実際には**「未使用領域の区画を管理するページテーブル」は作る必要がなくなる**ので，その分主記憶領域を節約できる．
- この方式では複数のページテーブルを管理する表（**ページディレクトリ**）が主記憶上に必要．



単レベルページング

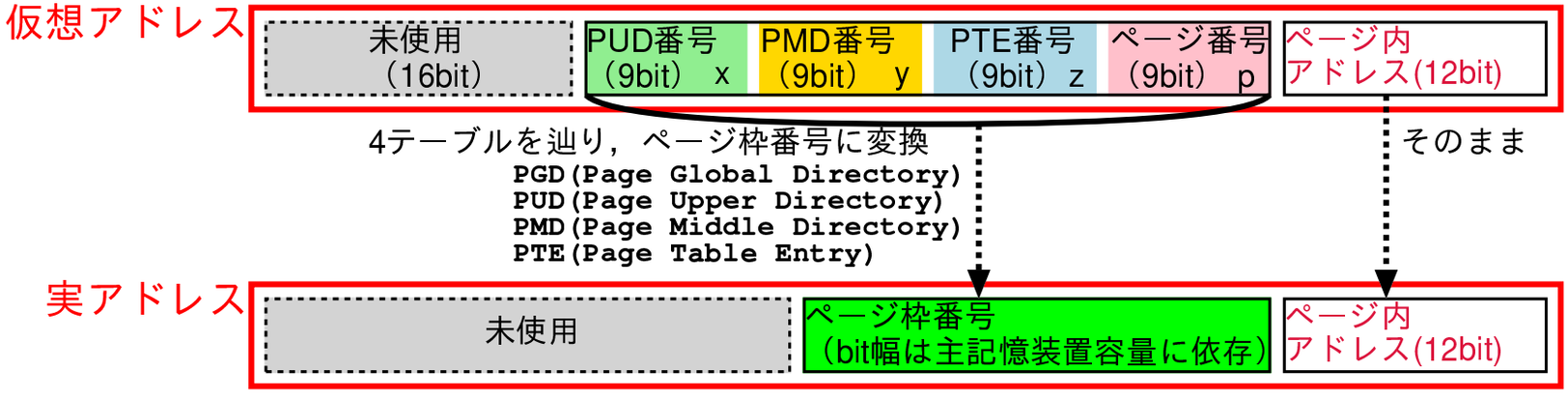
一つのページテーブルで管理
（仮想アドレス空間が巨大だとテーブルも巨大になる）

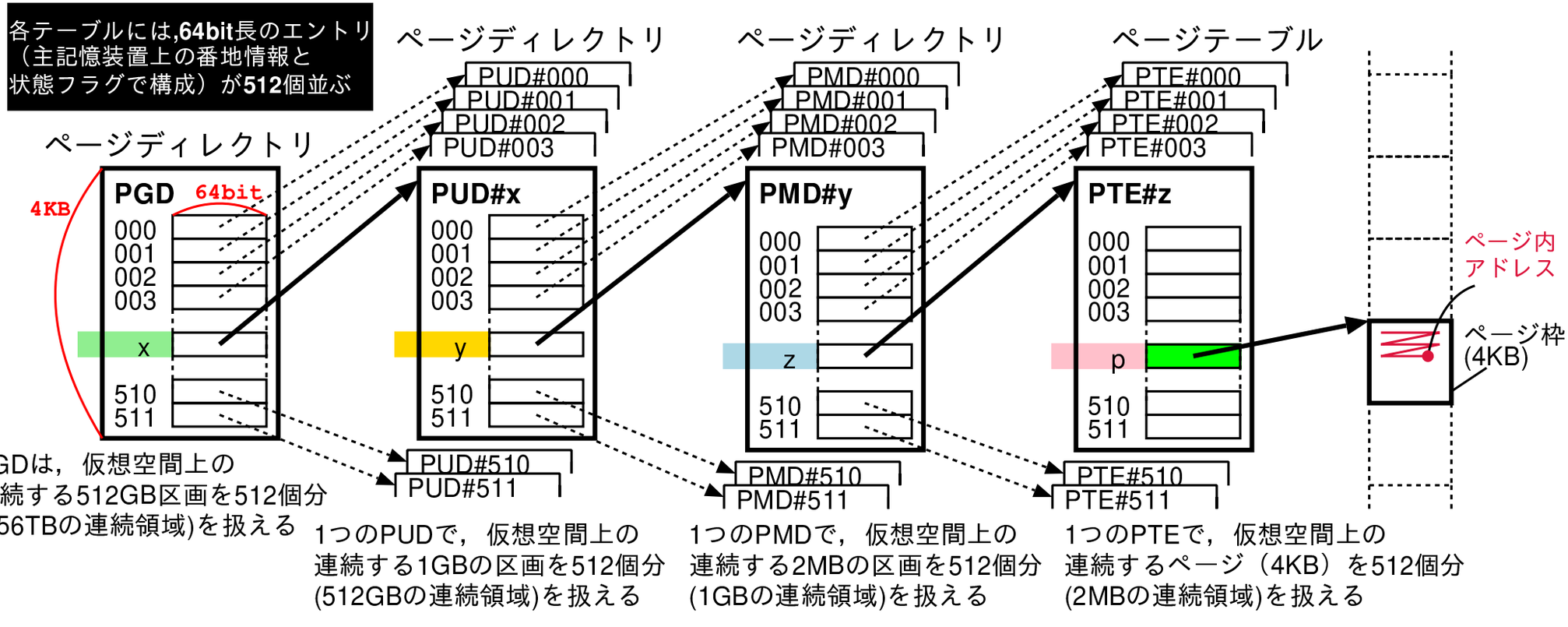


3レベルページング

仮想アドレス空間を領域に分割し，領域ごとにページテーブルを設ける
(使用されない仮想領域用のテーブルは不要．必要になったら追加．)

参考 64bit Linux(x86_64)における4レベルページングの概要





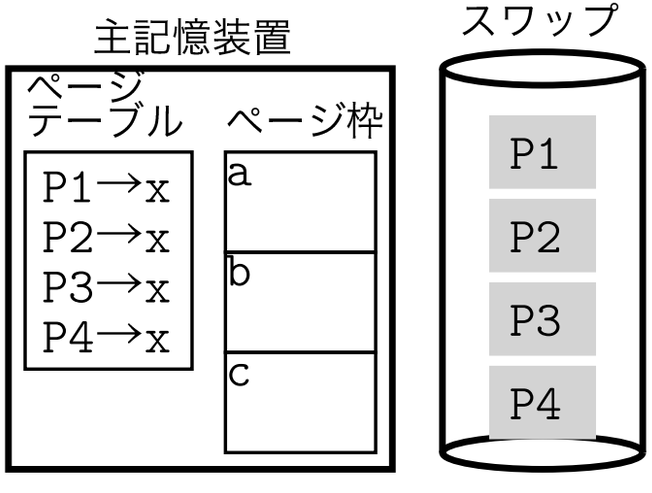
5.5 ページ置換えの技法

ページアウトを行う際，二次記憶に追い出すページをうまく選択することで，以後のページフォルトの発生を少なくすることができる．

❶ **FIFO**　主記憶上に一番古くから存在するページを二次記憶に追い出す

❷ **LRU**　最も長い間参照されていないページを二次記憶に追い出す

❸ **参照ビット法**　LRUを近似的に実現する



初期状態

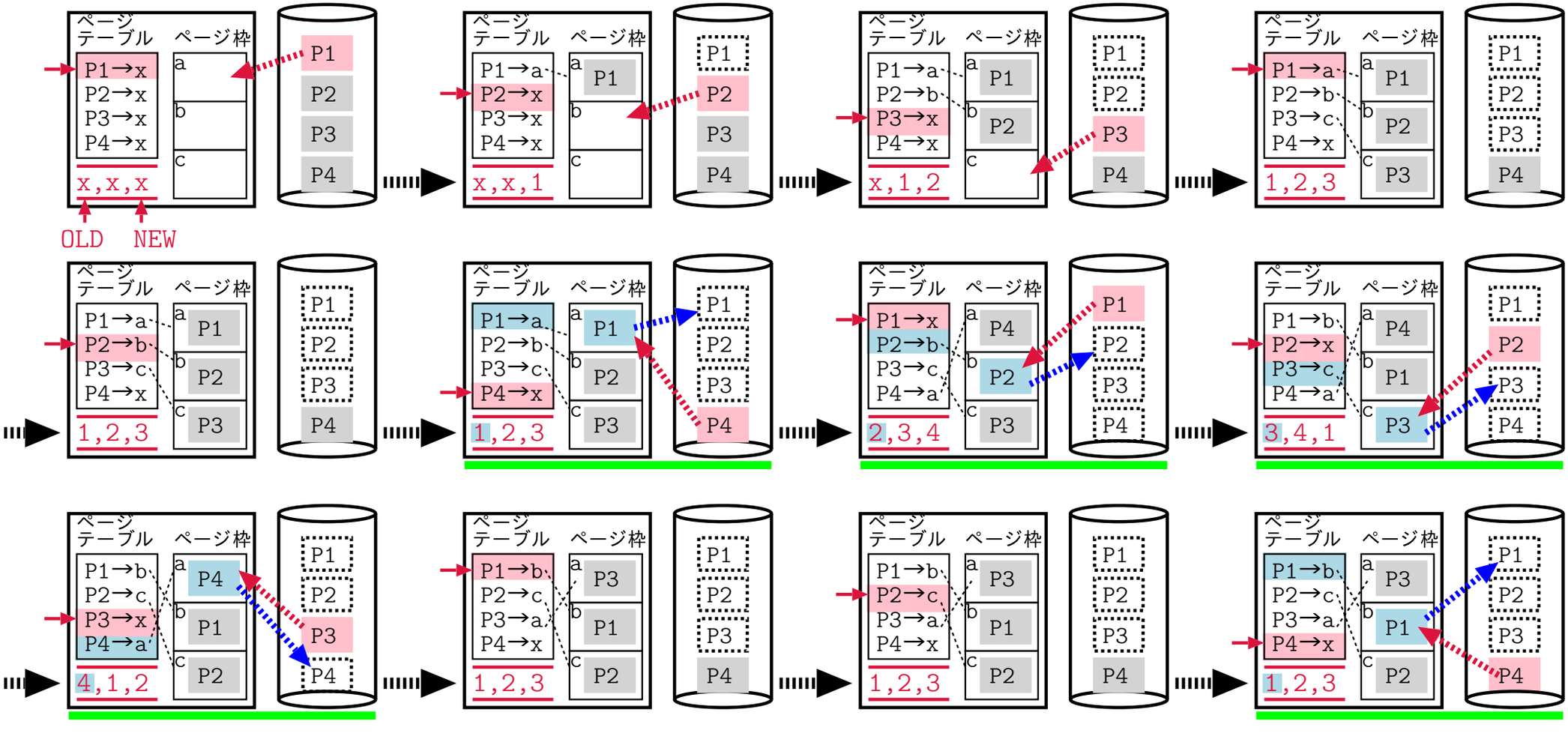
ページ参照順

1,2,3,1,2,4,1,2,3,1,2,4

として，FIFOとLRUでどう変わるかを確かめよう

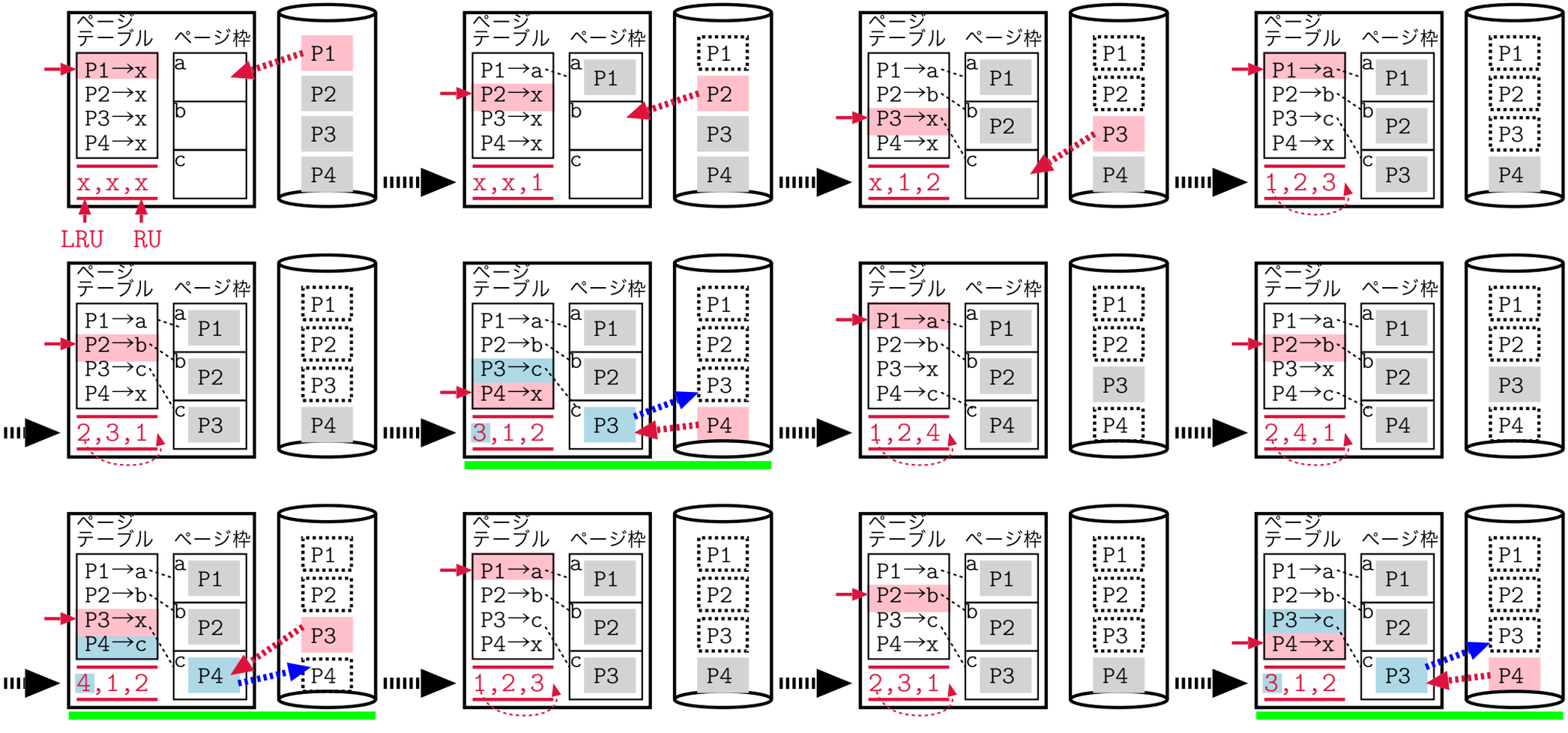
ページング(FIFOアルゴリズム)

ページ参照順
1,2,3,1,2,4,1,2,3,1,2,4



ページング(LRUアルゴリズム)

ページ参照順
1,2,3,1,2,4,1,2,3,1,2,4



演習準備：はじめにASLR[2)]をシステム全体で無効化して，xeyesを2つ立ち上げる．プロセス番号も調べておく．

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ xeyes & ↵
$ xeyes & ↵
$ pidof xeyes ↵
4443 4442
```

演習③：プロセスのページフォルトの回数，仮想メモリ消費サイズ(VSZ)，実メモリ消費サイズ(RSS)を観察しよう．

```
$ watch -n 0.1 "ps -p 4443,4442 -o pid,maj_flt,min_flt,vsz,rss,cmd" ↵

  PID  MAJFL  MINFL    VSZ   RSS CMD
 4442      3    351  18408  5316 xeyes
 4443      0    354  18408  5348 xeyes
```

[2)] ASLR: プロセスごとにメモリレイアウトをランダムにする機能．最近のLinuxではプロセスが外部から攻撃されにくくするために，この機能が有効化されている．

演習④：各プロセスの仮想アドレス空間を観察しよう．

pmapは */proc/プロセスID/maps* の情報をわかりやすく表示するコマンド

```
$ pmap -x 4442 ↵   (-xの代わりに-Xを指定すると更に詳細が表示される)
Address           Kbytes     RSS   Dirty Mode  Mapping
000055555554000       8       8       0 r---- xeyes データ領域(ReadOnly)
000055555556000      12      12       0 r-x-- xeyes 命令領域
000055555559000       4       4       0 r---- xeyes
00005555555a000       8       8       8 r---- xeyes
00005555555c000       4       4       4 rw--- xeyes
00005555555d000     584     304     304 rw---   [ anon ]
(略)
00007ffff6e94000    7580     380       0 r---- locale-archive
(略)
00007ffff76ae000    1504    1144       0 r-x-- libc-2.31.so Cライブラリ
(略)
00007ffff7d75000    2048       0       0 ----- libXt.so.6.0.0
(略)
00007fffffde000     132      24      24 rw---   [ stack ]
(略)
---------------- ------- ------- -------
total kB           18412    5420     636
                  (↑VSZ) (↑RSS)
```

もう一方のxeyesに対しても確認せよ（ASLRが無効なので↑と同じ番地になる）

演習⑤：プロセスごとに，ページテーブルで消費しているメモリサイズを調べよう．

```
$ cat /proc/4442/status | grep Vm ↵
VmPeak:    18556 kB    ← 仮想メモリの消費量（ピーク値）
VmSize:    18556 kB    ← 仮想メモリの消費量
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      5176 kB    ← 実メモリの消費量（ピーク値）
VmRSS:      5176 kB    ← 実メモリの消費量
VmData:      768 kB    ← データ領域の消費量
VmStk:       132 kB    ← スタック領域の消費量
VmExe:        20 kB    ← テキスト領域の消費量
VmLib:      3692 kB    ← 共有ライブラリサイズ
VmPTE:        52 kB    ← このプロセスのPTEの合計サイズ
VmSwap:        0 kB
（/proc以下のファイルの詳細説明は man procfs で調べられる）
```

【参考】PTEは64bit Linux(x86_64)の最下位レベルのページテーブルのこと．1つのPTEは4KBの大きさ（「8バイト長のエントリ」××512個で構成）を持ち，2MB（=4KB×512ページ）の仮想空間をカバー．52KB分のPTEだと，13個(=52KB/4KB)のPTEを使用して，26MB (=13×2MB)の仮想空間領域をカバーしていることになる．

演習⑥：システム全体でページテーブルでどれだけメモリを消費しているかを調べよう．

```
$ watch -n 0.1 "cat /proc/meminfo | grep PageTables"

PageTables:    25696 kB        ←全プロセスのPTEの合計サイズ
```

firefoxなどを立ち上げたり，終了させたりして，ページテーブルで消費しているメモリの変化を見てみるとよい．

演習⑦：test.cを使って，1つのプロセスが実行を完了するまでにCPUを使った時間(ユーザモード，カーネルモード)，コンテクストスイッチの回数，ページフォルトの回数を見てみよう．

```
$ gcc test.c ↵
$ /usr/bin/time -v ./a.out ↵

Command terminated by signal 2
Command being timed: "./a.out"
User time (seconds): 1.77      ← CPUをユーザモードで使用した時間
System time (seconds): 2.87    ← CPUをカーネルモードで使用した時間
Percent of CPU this job got: 93%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:05.00 ← 経過時間
(略)
Maximum resident set size (kbytes): 1248 ← RAMの最大消費量
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0        ← メジャーページフォルト数
Minor (reclaiming a frame) page faults: 53 ← マイナーページフォルト数
Voluntary context switches: 561 ← 自発的にCPUの実行権を手放した回数
Involuntary context switches: 0 ← 強制的にCPUの実行権を取られた回数
(略)
Exit status: 0
```

List 1: test.c

```
 1 #include<stdio.h>
 2 #include<unistd.h>
 3 int main()
 4 {
 5    int cnt = 0;
 6    alarm(5);   // 約5秒後にALRMシグナルを受け取るようセット
 7                //（↑ デフォルトではプログラムを停止する）
 8
 9    while (1) {
10       cnt++;
11       getpid();
12    }
13    return 0;
14 }
```

ステップアップ

1. test2.cについて説明してみなさい．

2. test2.cを動かし，次のコマンドでメモリの変化をモニタリングしてみなさい（演習⑤を参考）．
   `watch -n 0.1 "cat /proc/プロセスID/status | grep Vm"`

3. ↑でモニタした結果について（特に，VmPTEとVmRSSの変化）考察してみよ．

List 2: test2.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#define N 1024*1024*200 // 200M

int main()
{
    char *p = NULL;
    int n;
    while(1) {
        p = malloc(N); // OSにメモリ確保を要求
        if(p == NULL) {
            printf("Out of memory!!\n");
            return 0;
        }
        for(n = 0; n < N; n++) {
            if(n % 1024 == 0) {
                usleep(1);
                printf("%20dk\r", n / 1024);
                fflush(stdout);
            }
            p[n] = 'X'; // 確保した領域にアクセス
        }
```

```
24          sleep(2);
25          free(p); // メモリ解放
26          sleep(2);
27      }
28      return 0;
29  }
```

参考 malloc()を呼び出してメモリ領域を動的に確保しても，直ちに実メモリが確保されるわけではない（**確保されたのは仮想メモリ上の領域でしかない**）．その仮想メモリ領域にアクセスを試みたときに，**マイナーページフォルト**（ディスクアクセスを必要としないページフォルト）が起き，そこではじめて実メモリ上にメモリが確保される（＝ページが実メモリ上にマッピングされ，アクセス可能になる）．