組込みエンジニアのためのLinux入門
仮想メモリ編

2007.2.22

株式会社アプリックス
小林哲之

このスライドの対象とする方

- 今までずっと組込み機器のプロジェクトに携わってきて最近はOSにLinuxを使っている方々

このスライドの目的

- Linuxの仮想メモリの仕組みを理解し現在のプロジェクトに役立てる。
  - 仕組みを知らなくてもプログラムは動くが、性能を引き出すためには仕組みの理解が重要。

まず基本の概念から

- 仮想～、論理～ virtual, logical
  - 仮想アドレス、論理デバイス、論理セクタ、仮想マシン
  - あたかも ... のように扱う
- 実～、物理～ real, physical
  - 実アドレス、物理デバイス、物理セクタ
  - そのもの、そのまんま

仮想化：あたかも・・・ 実は

- あたかも巨大のようだが、実は少ない。
- あたかも平らのようだが、実は凸凹。
- あたかもたくさんのようだが、実はひとつ。
- あたかも占有しているようだが、実は共有。

仮想化は複雑さや個々に依存することを隠蔽するマジック。
マジックなので種も仕掛けもある。
＝ 実と仮想の対応付け（マッピング）
あたかもそう見えるように変換している。

仮想化の代償

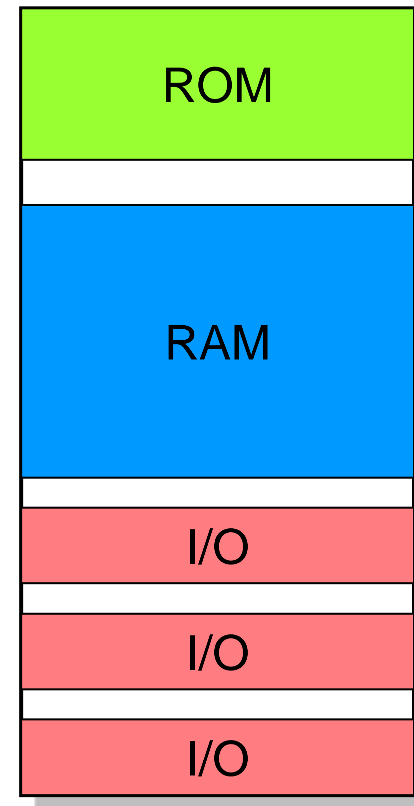
- マイナス面よりもプラス面が勝っているから仮想化を行うわけだが、常にプラスとは限らない。

物理メモリと仮想メモリ

- いままでの大抵の組込み機器プロジェクトでは物理メモリしか扱うことがなかった。
- 最近は組み込みシステムの規模の増大化に伴ってLinuxやWindow CEなどPC向けOSの流れを持つOSを使用することが多くなってきた。これらのOSは仮想メモリシステムを備えている。
- このスライドではLinuxについて説明する。

物理メモリ

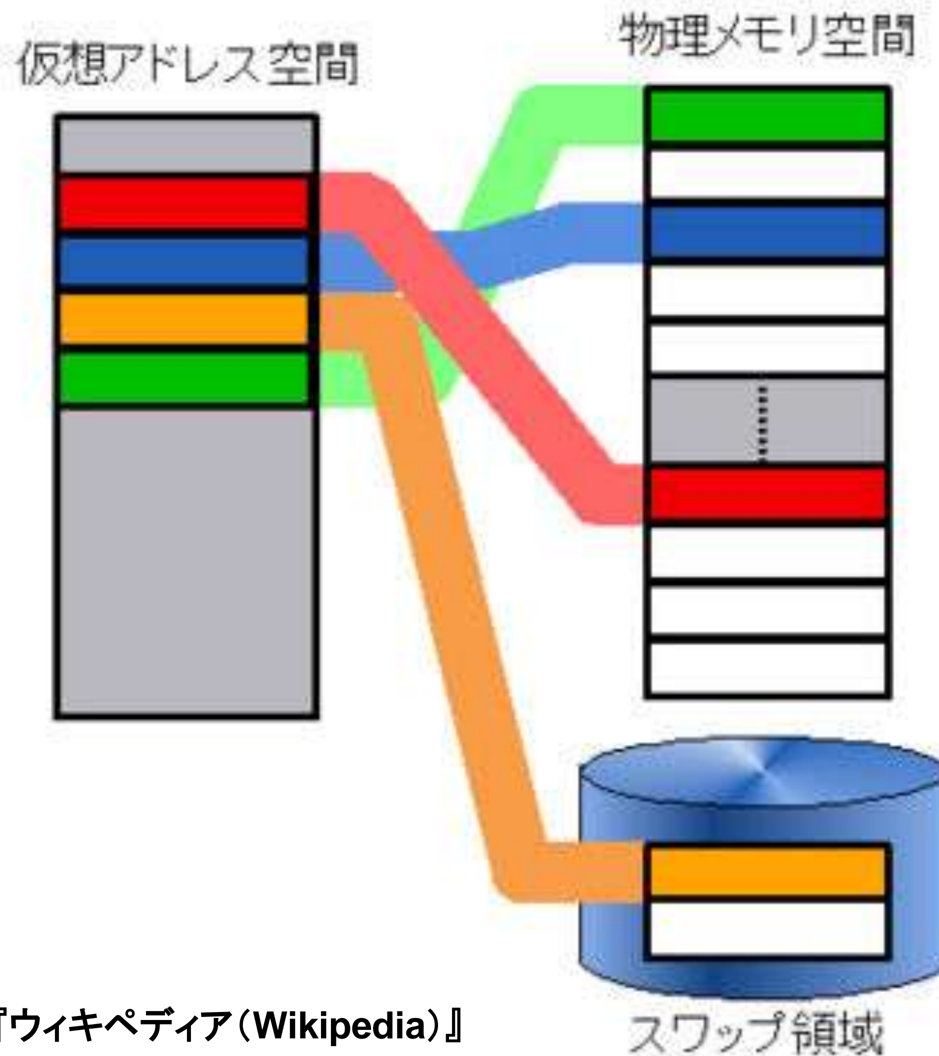
- 単一のメモリ空間
- 機器ごとにROM,RAM,I/Oの実装アドレスが異なるのでそれを意識してプログラムする



仮想メモリ

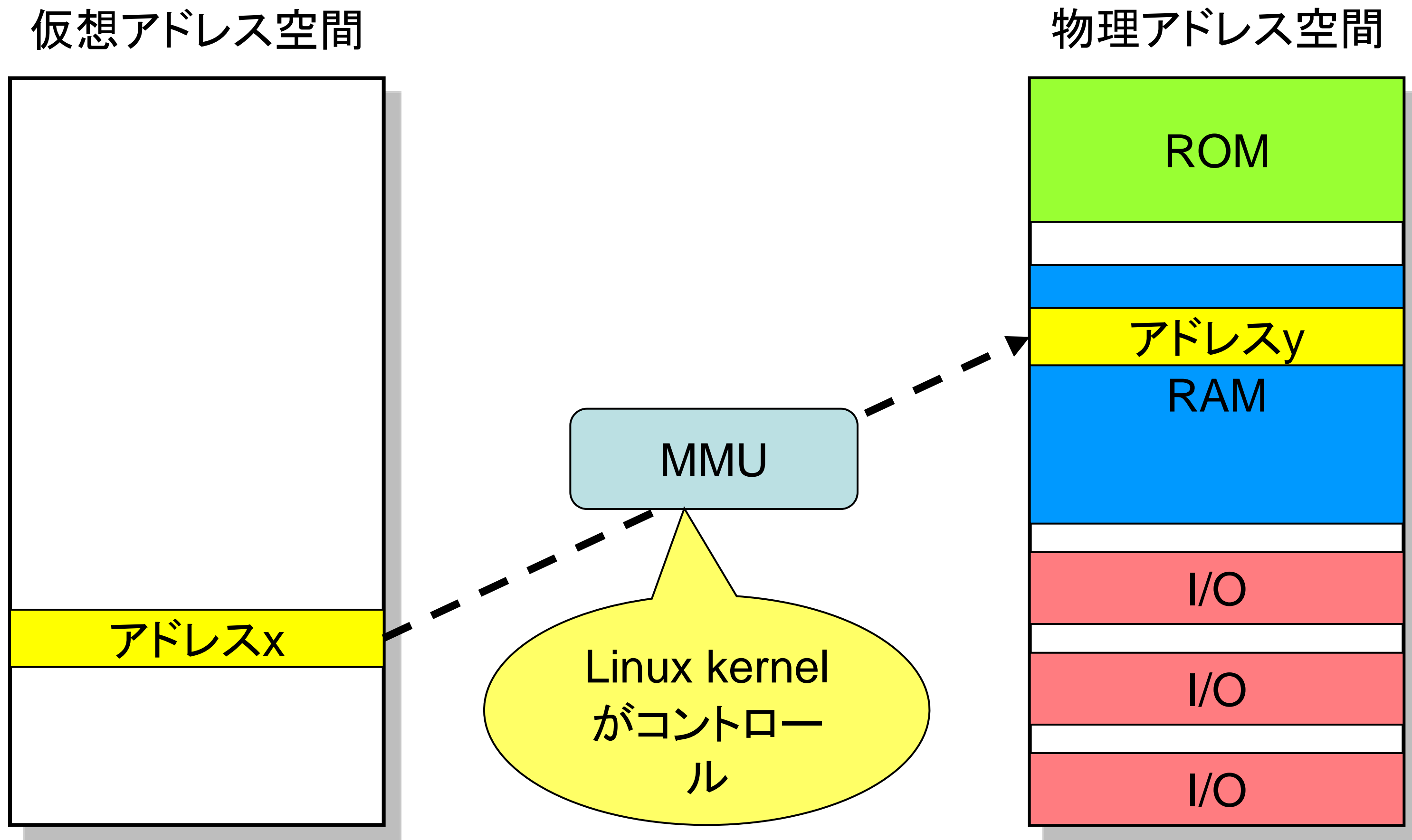
- 利点
  - ユーザープログラムは実際のメモリマップ（実装アドレス、実装サイズ）に依存しなくなる。
  - 不連続な物理メモリの断片を連続する仮想メモリとして利用できる。
  - メモリ保護：バグによって無関係の部分のメモリが破壊されることを防止できる。
- 新しい概念の導入
  - アドレス変換
  - 多重メモリ空間
  - デマンドページング

仮想メモリの概念図

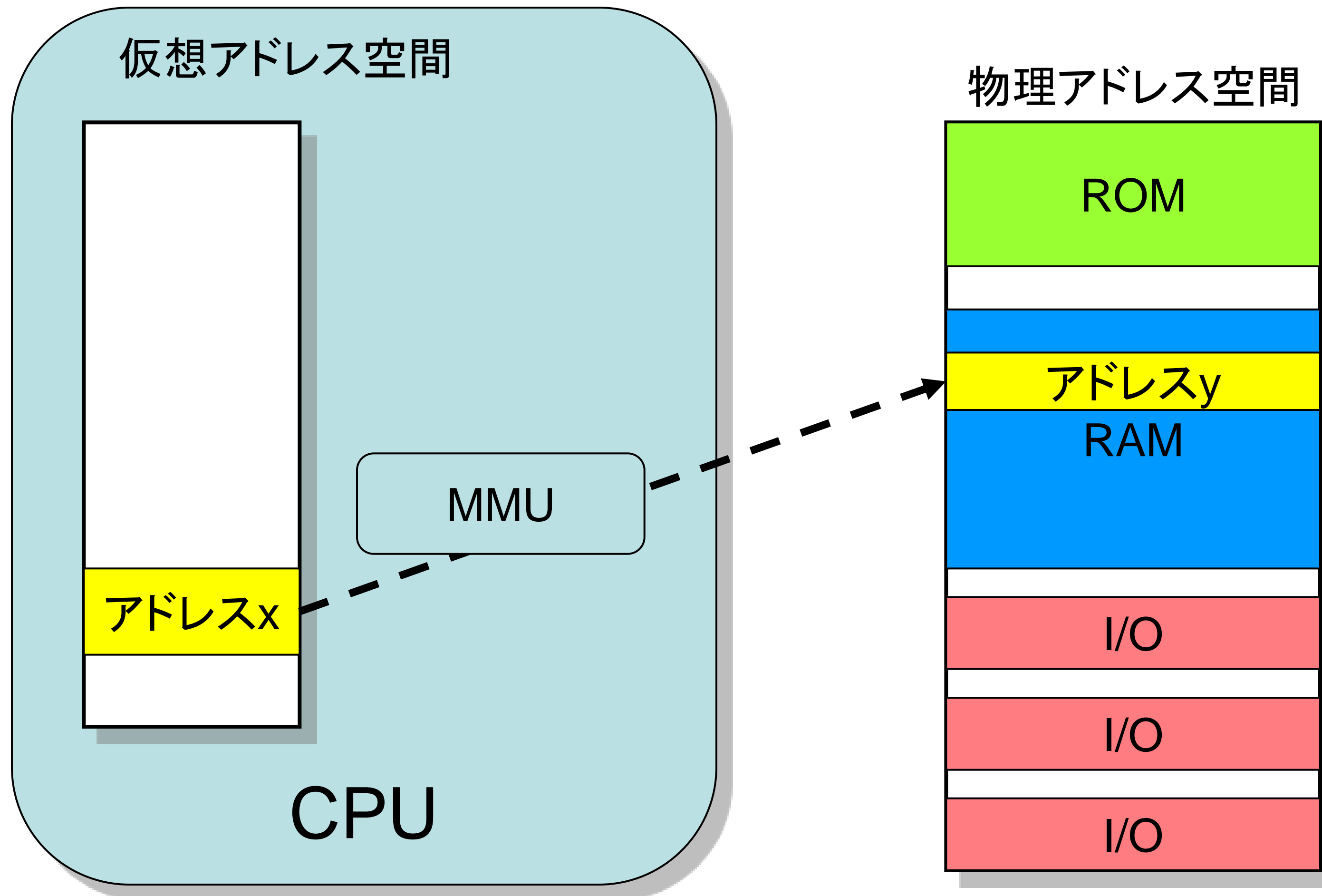


出典: フリー百科事典『ウィキペディア（Wikipedia）』

アドレス変換

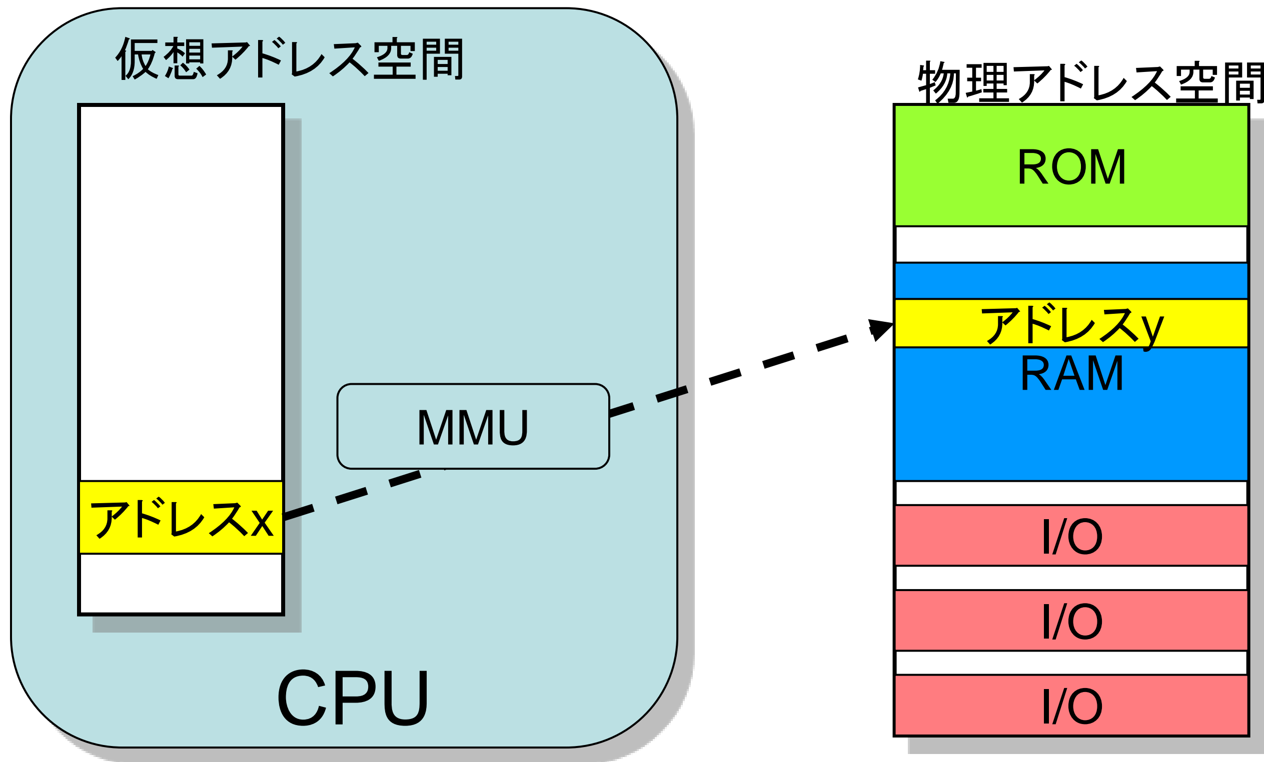


仮想メモリはCPUの中だけ



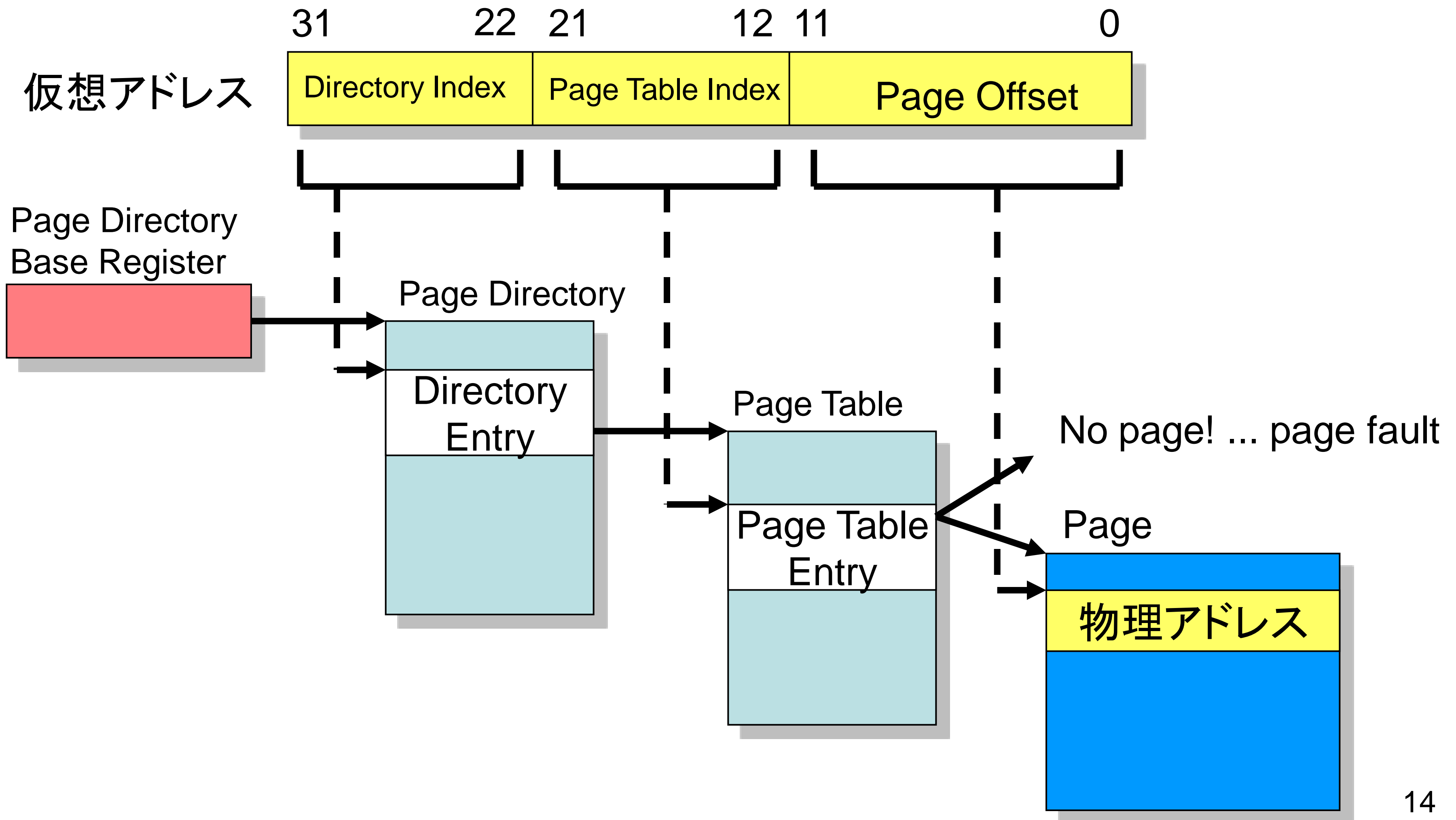
CPUから外にでてくるアドレスバスにのるのは物理アドレスだけ。
ロジアナでは仮想アドレスは観測できない。

ユーザープログラムは仮想アドレスだけ



物理アドレスを扱うのはカーネルモードだけ。
つまりカーネル本体とデバイスドライバ。

MMUでのアドレス変換

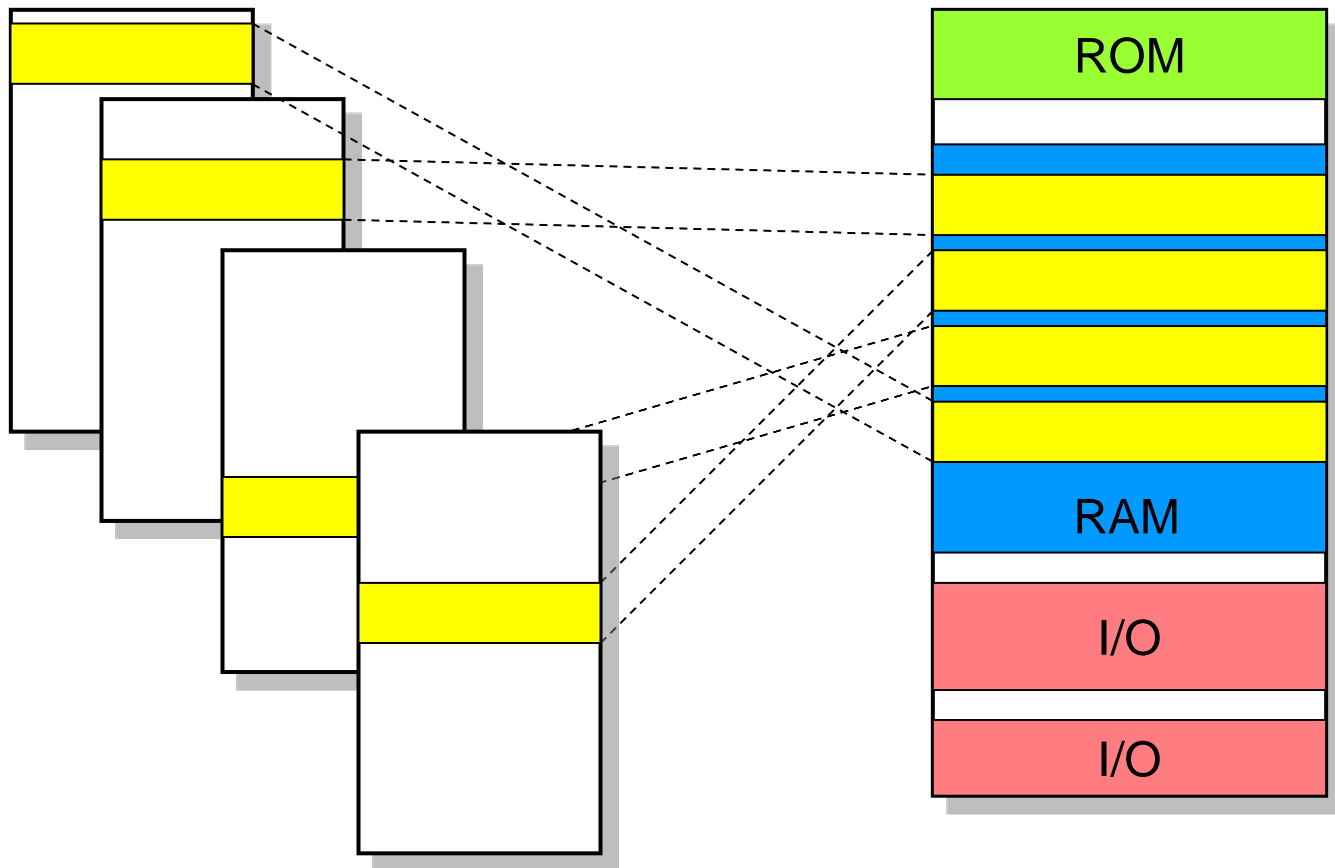


TLB

仮想アドレスページ	物理アドレスページ
…	

- Translation Lookaside Buffers
- 仮想アドレスをkeyとして物理アドレスを得るハッシュテーブルのようなもの
- 大抵のアドレス変換はTLBにヒットするので実際にPage Directory, Page Tableをアクセスせずに済む。

多重メモリ空間



デマンドページング

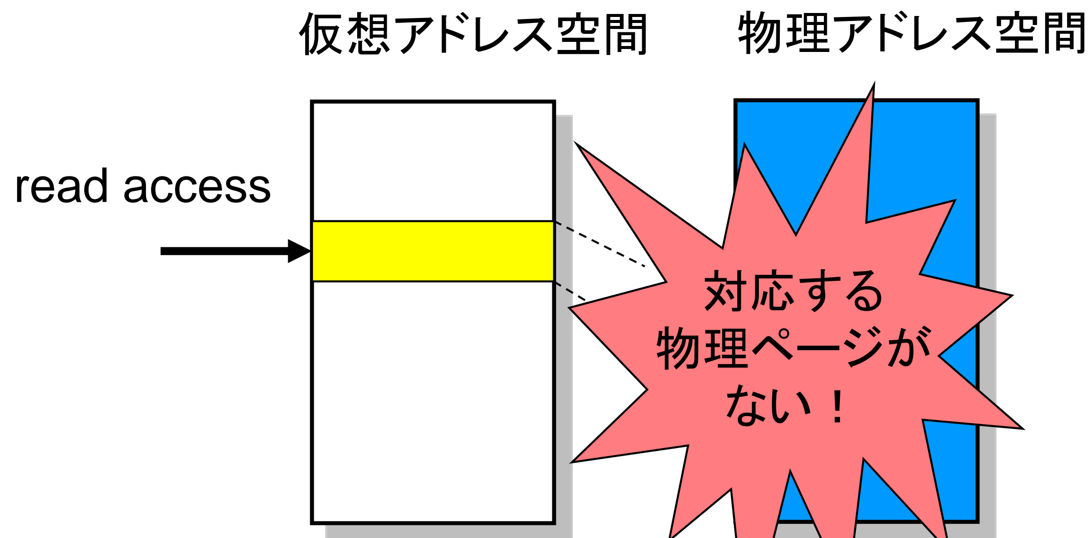
- ページ単位でマッピングされる
  - ページのサイズはたいていは4Kbytes
- 2段階で行われる
  1. 仮想メモリの割り当て(mmap)
     管理台帳に登録するだけ
  2. 実際にアクセスがあったときに初めて
     そのページに物理メモリが割り当てられる

アクセスのないページには物理メモリが割り当てられないので

仮想メモリサイズ >= 実際に必要な物理メモリサイズ

デマンドページングの動作例

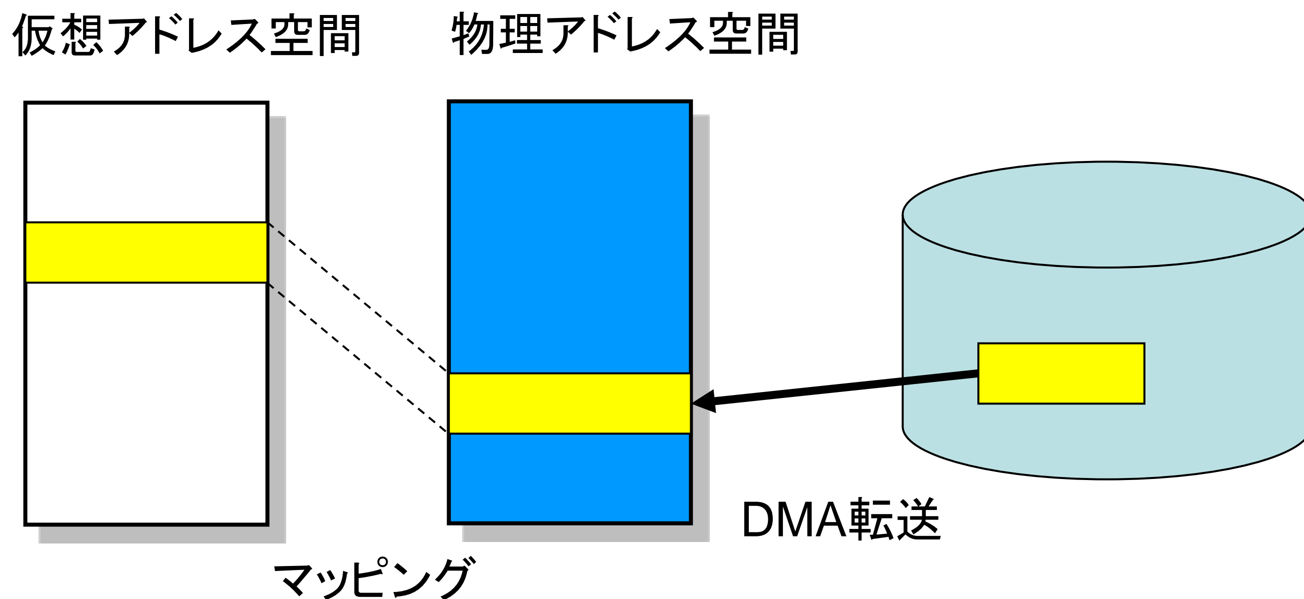
(1)



ページフォールト発生
カーネルモードへ

デマンドページングの動作例（続）

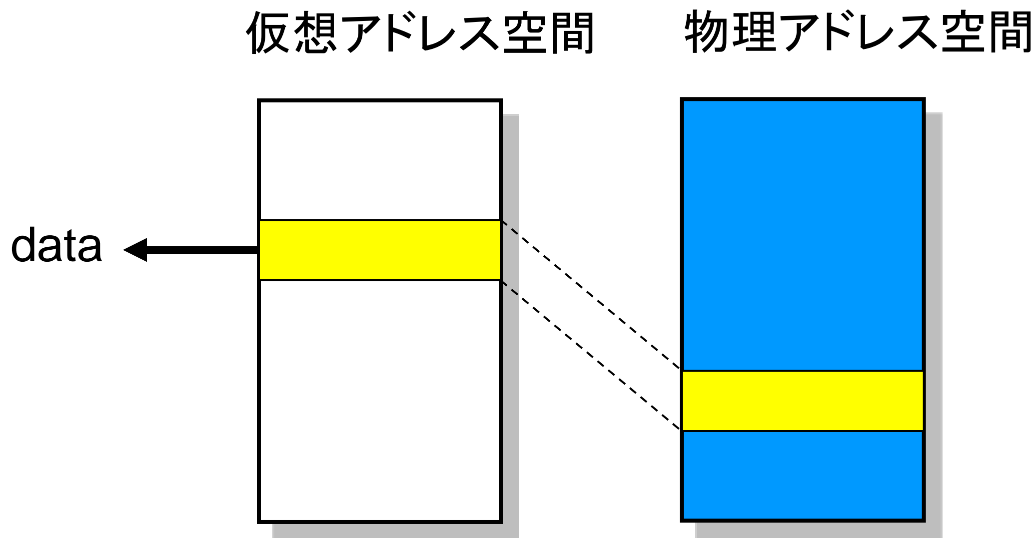
(2)



カーネルがデータをロードして物理アドレスをマッピングする。

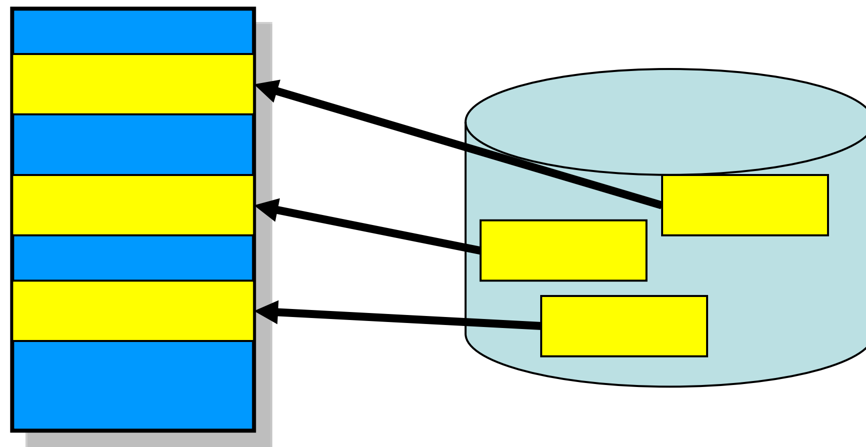
デマンドページングの動作例（続）

(3)



ユーザーモードに復帰。
ユーザープログラムからは何事もなかったように
データが読める。... でも実際に時間はかかっている。

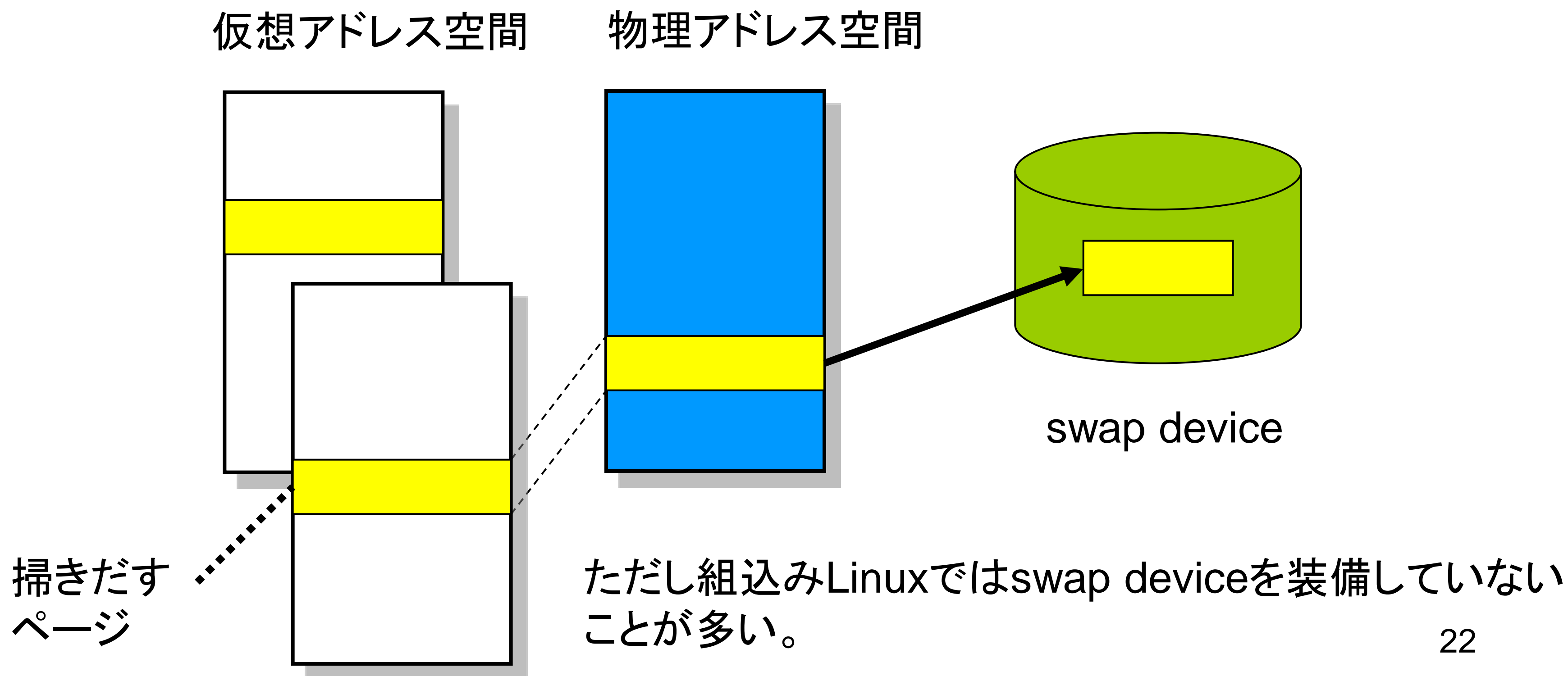
ページキャッシュ



ディスクからの読んだ内容はメモリに余裕がある限り保持しておく。
シーケンシャルにアクセスされる場合が多いので数ページ分をまとめて先読みを行う。
そのため(2)では毎回ディスクアクセスが発生するとは限らない。

ページアウト

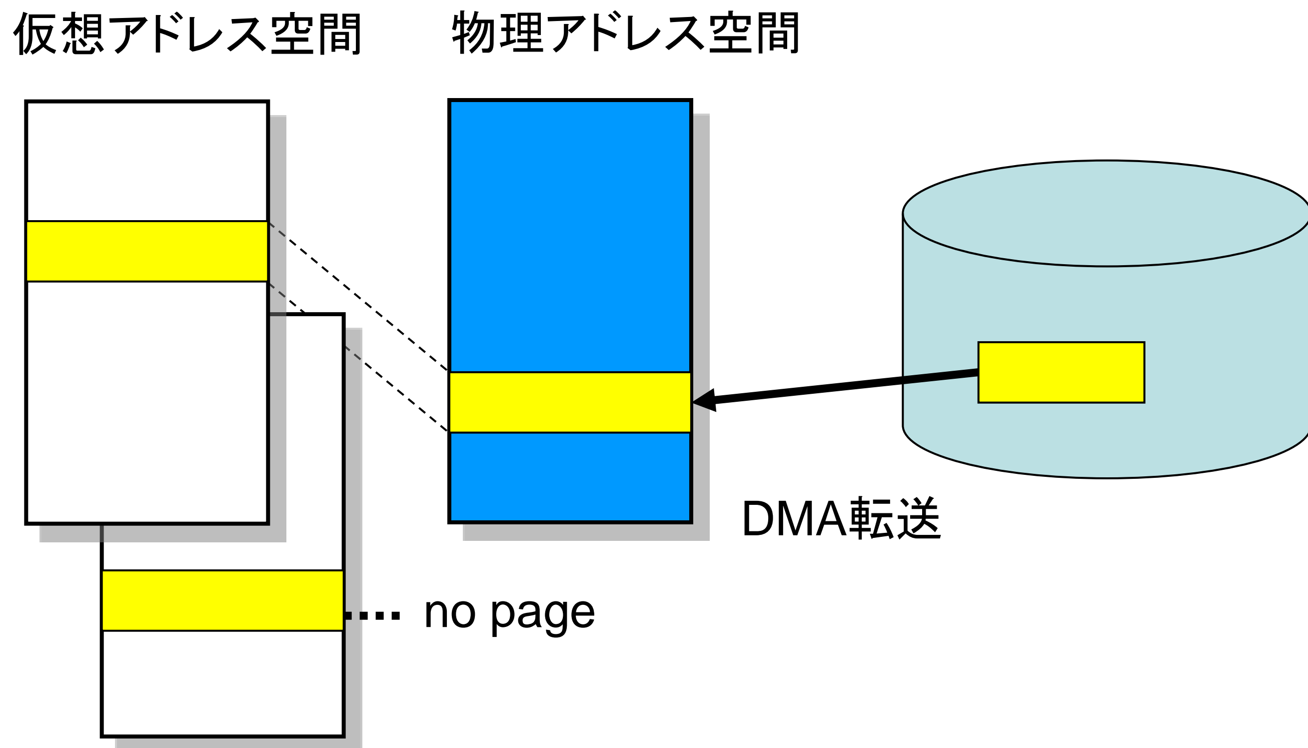
(2)で物理メモリの空きがなかった場合、使用頻度の低いと思われるページを解放する。そのページの内容が変更されていなければそのまま破棄。
変更されていればスワップデバイスに掃きだす。



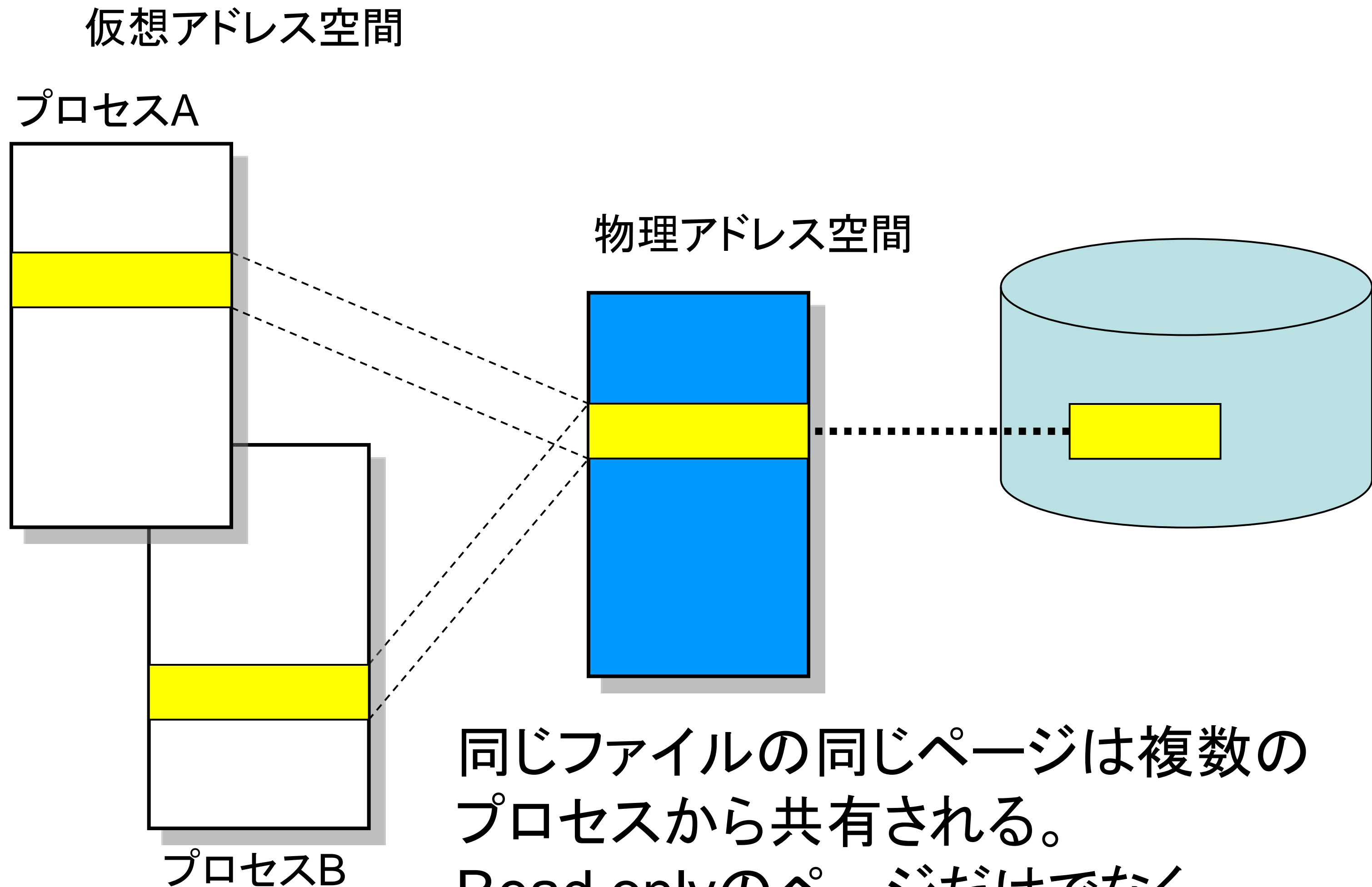
ただし組込みLinuxではswap deviceを装備していないことが多い。

ページアウト（続）

解放した分のページを使って要求されたページの割り当てを行う。
このような「お手玉」をすることで実際に搭載されている物理メモリサイズよりも大きなサイズの仮想メモリを扱うことができる。



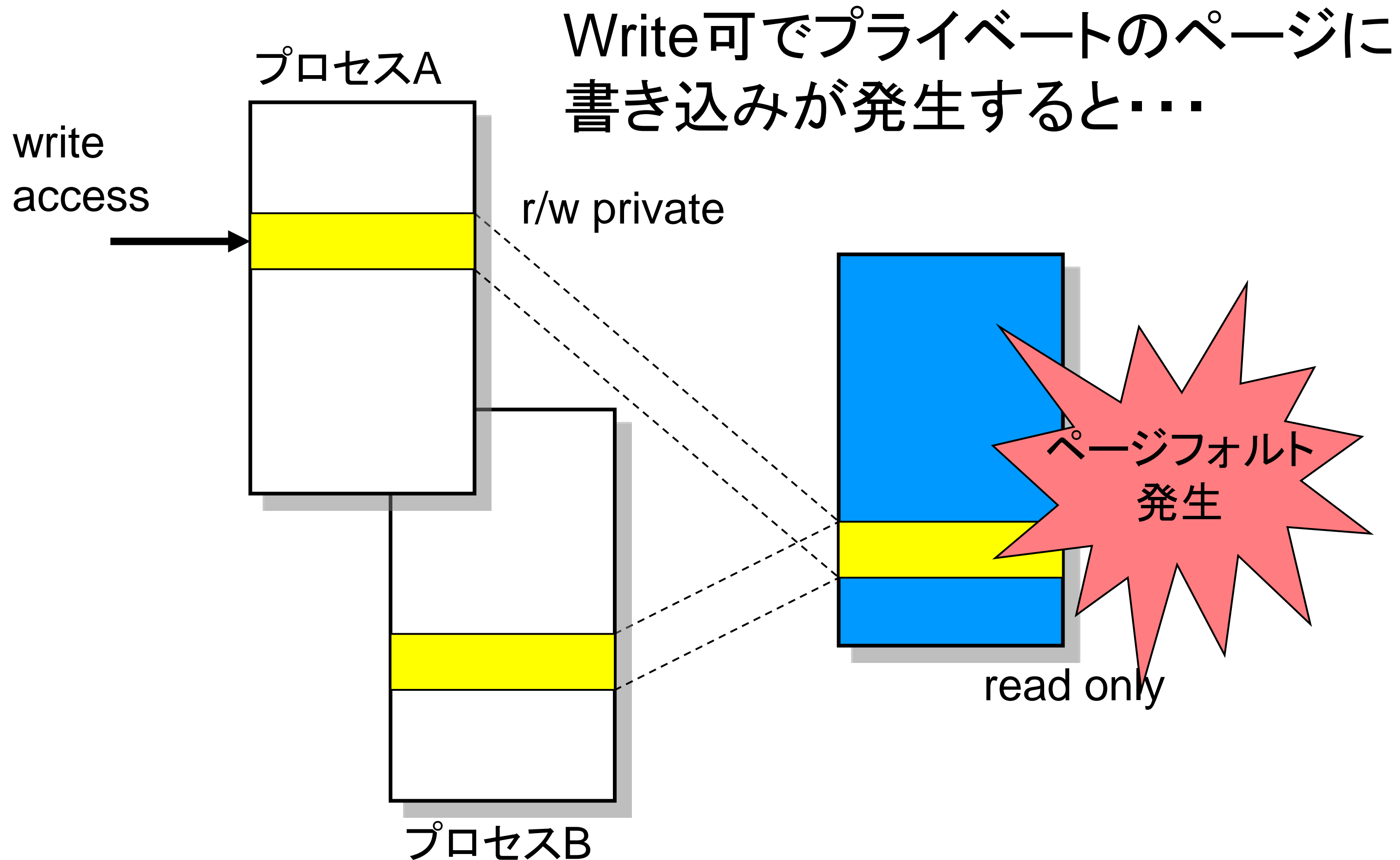
ページの共有



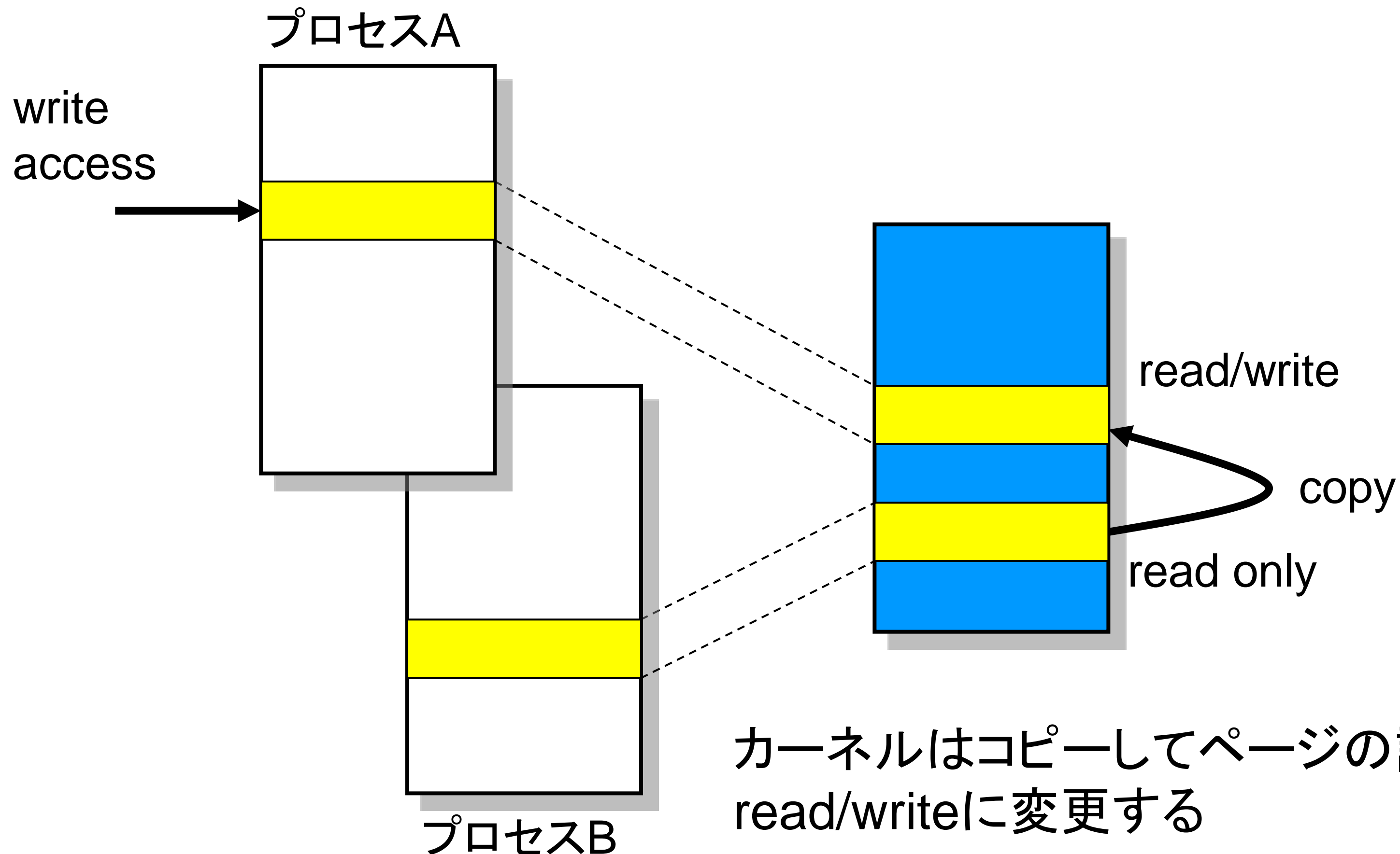
同じファイルの同じページは複数のプロセスから共有される。
Read onlyのページだけでなくwrite可のページも。

コピーオンライト

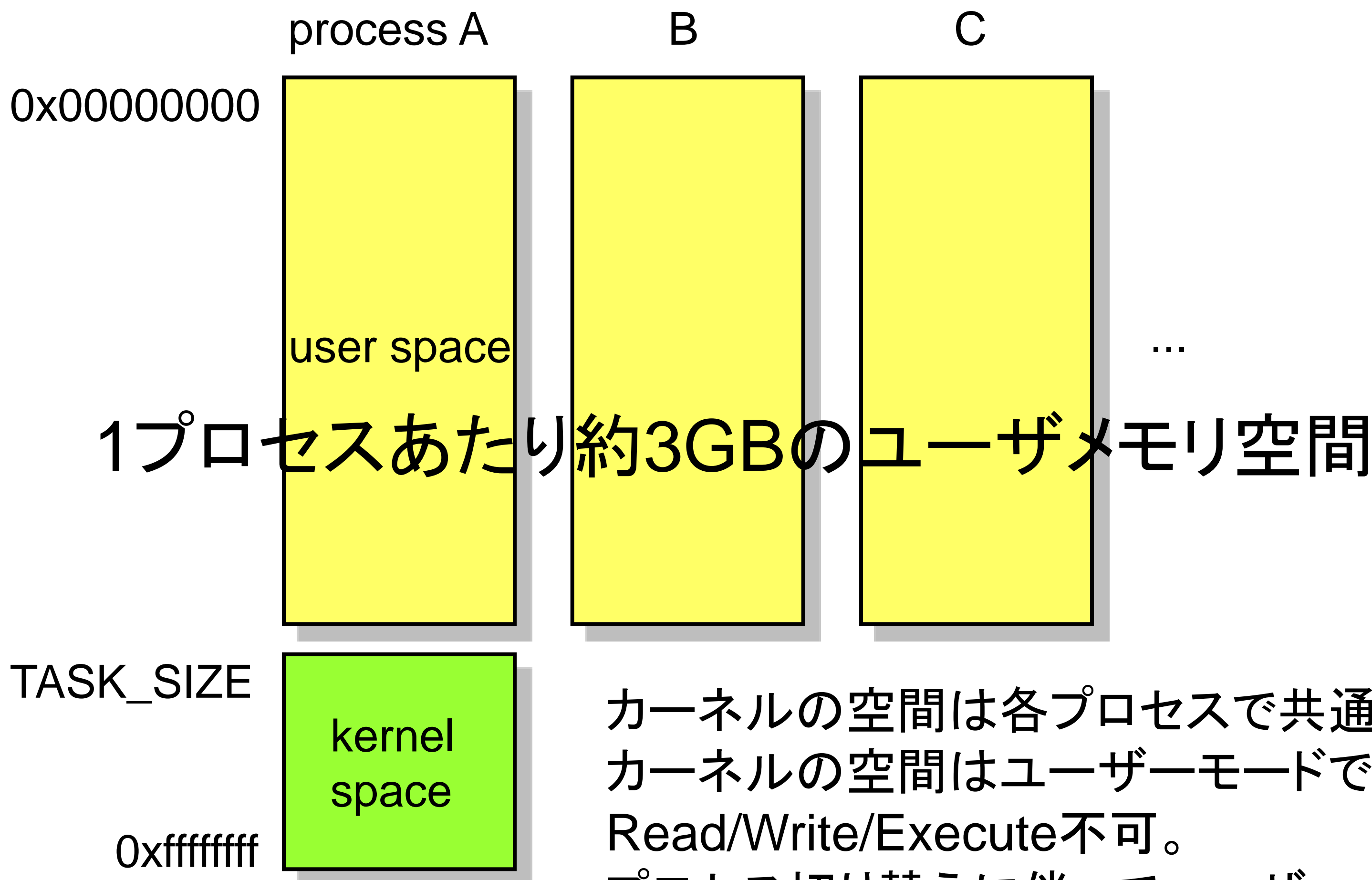
Write可でプライベートのページに書き込みが発生すると・・・



コピーオンライト（続）



プロセスのメモリ空間



カーネルの空間は各プロセスで共通。
カーネルの空間はユーザーモードでは
Read/Write/Execute不可。
プロセス切り替えに伴ってユーザーメモリ
空間が切り替わる。

TASK_SIZEはi386では0xc0000000
ARMでは 0xbf000000

ユーザプロセスのメモリ空間の実例

cat /proc/<PROCESS_ID>/maps

```
00101000-0011a000 r-xp 00000000 fd:00 15172739   /lib/ld-2.4.so
0011a000-0011b000 r-xp 00018000 fd:00 15172739   /lib/ld-2.4.so
0011b000-0011c000 rwxp 00019000 fd:00 15172739   /lib/ld-2.4.so
0011e000-0024a000 r-xp 00000000 fd:00 15172740   /lib/libc-2.4.so
0024a000-0024d000 r-xp 0012b000 fd:00 15172740   /lib/libc-2.4.so
0024d000-0024e000 rwxp 0012e000 fd:00 15172740   /lib/libc-2.4.so
0024e000-00251000 rwxp 0024e000 00:00 0
08048000-08049000 r-xp 00000000 fd:00 11666681   /home/koba/lab/loop/a.out
08049000-0804a000 rw-p 00000000 fd:00 11666681   /home/koba/lab/loop/a.out
b7fef000-b7ff1000 rw-p b7fef000 00:00 0
b7fff000-b8000000 rw-p b7fff000 00:00 0
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
```

アドレス範囲

file offset

inode

file name

r: read
w: write
x: execute
s: shared
p: private (copy on write)

device
major:minor

ユーザプロセスのメモリ空間の実例 (詳細)

cat /proc/<PROCESS_ID>/smaps

```
      ....
0011e000-0024a000 r-xp 00000000 fd:00 15172740   /lib/libc-2.4.so
Size:                 1200 kB
Rss:                   136 kB
Shared_Clean:          136 kB
Shared_Dirty:            0 kB
Private_Clean:           0 kB
Private_Dirty:           0 kB
0024a000-0024d000 r-xp 0012b000 fd:00 15172740   /lib/libc-2.4.so
Size:                   12 kB
Rss:                     8 kB
Shared_Clean:            0 kB
Shared_Dirty:            0 kB
Private_Clean:           0 kB
Private_Dirty:           8 kB
0024d000-0024e000 rwxp 0012e000 fd:00 15172740   /lib/libc-2.4.so
Size:                    4 kB
Rss:                     4 kB
Shared_Clean:            0 kB
Shared_Dirty:            0 kB
Private_Clean:           0 kB
Private_Dirty:           4 kB
      ....
```

RSS = 物理メモリサイズ

システムコールmmap

```
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags,
        int fd, off_t offset);

int munmap(void *start, sizt_t length);
```

- ファイルやデバイスをメモリにマップ/アンマップする
- 引数 *prot*
  - PROT_NONE または PROT_EXEC, PROT_READ, PROT_WRITEのOR演算
- 引数 *flags*
  - MAP_FIXED, MAP_SHARED, MAP_PRIVATE, MAP_ANONYMOUS, ...

mmapのtips

- MAP_FIXEDを指定しなければカーネルが空いているページをさがしてくれる。
- MAP_FIXEDを指定したときに既存のページと重なっていたら、そのページは内部的にmunmapされる。
  - なのでこのオプションは通常は使用しない。
- ファイルのオフセットはページサイズの整数倍でなければならない。
- mmapとmunmapのアドレス、サイズは一致していなくてもよい。

mmapの使い方(1)

- 巨大サイズのmallocの代用
  - コンパクションなどデータのコピーが発生しない。
  - malloc/freeと違ってmunmapするときのaddr, sizeはmmapで確保したときと異なっていてもよい。
    - まとめて1回のmmapで確保して少しづつ分割してmunmapで返却するのもあり。
  - glibcのmallocの実装ではある一定以上のサイズのmallocはmmapを呼び出す。
    - DEFAULT_MMAP_THRESHOLD = (128*1024)

mmapの使い方(2)

- 高速なファイルアクセス
  - read, writeのシステムコールでは内部で物理ページにバッファリングしている。そこからユーザの指定した配列にコピーしている。
  - mmapを使うことで直接ページにアクセスできるようになるのでデータのコピーを減らすことができる。
  - Java1.4の java.nio.MappedByteBuffer

mmapの使い方(3)

- プロセス間の共有メモリ
  - 複数のプロセスから同じファイルをR/W可sharedでマッピングする。
  - IPCの共有メモリのシステムコール(shmget, shmat,..)は内部で同様のことを行っている。

mmapの使い方(4)

- 物理メモリ、I/Oポートのアクセス
  - デバイスファイル /dev/mem をマッピングすることでユーザーモードで物理メモリ空間をread/writeすることが可能。
  - /dev/memをアクセスするにはrootの権限が必要。

まとめ

- 仮想メモリの使用量と物理メモリの使用量は異なる。実際に問題になるのは物理メモリの使用量。
- 仮想メモリのオーバーヘッドはいつ発生するのかを意識する。
  - TLBミス
  - ページフォルト
- システムコールmmapの活用。

参考文献

- Linux kernel ソース
  http://www.kernel.org/
- GNU C ライブラリのソース
  http://www.gnu.org/software/libc/
- “詳解LINUXカーネル 第2版”
  オライリージャパン
- “Linuxカーネル2.6解読室”
  SoftBank Creative
- Linux manコマンド
- その他たくさんのWEB検索結果

おまけ：最近の話題

- CELFのBootTimeResourcesより
  - KernelXIP
  - ApplicationXIP
  - (DataReadInPlace)
- CELFのMemoryManagementResoucesより
  - Huge/large/superpages
  - Page cache compression