

メモリホットプラグ — 設計と実装

Date: 2003-10-10

岩本 俊弘, VA Linux Systems Japan

TOC

- 発表の流れ
- メモリホットプラグの実現可能性
- メモリホットプラグの効果
- カーネルによるメモリ管理
- VM (仮想記憶)
- LinuxのVM -- ページキャッシュ
- LinuxのVM -- デマンドロード(1)
- LinuxのVM -- デマンドロード(2)
- LinuxのVM -- スワッピング
- LinuxのVM -- スワップアウト
- LinuxのVM -- 内部データ構造
- ゾーン -- ページの管理単位
- ザーンの追加と削除
- メモリ削除の課題
- swapoutアプローチ(1) -- 回収可能なページ
- swapoutアプローチ(2) -- なぜhighmem?
- swapoutアプローチ(3) -- alloc_page禁止
- swapoutアプローチ(4) -- kswapdの閾値
- swapoutアプローチ(5) -- LRUリスト
- swapoutアプローチ(6) -- 強制解放
- swapout アプローチの限界 (1) -- kswapd の動作
- swapout アプローチの限界 (2) -- page count
- swapout アプローチ -- まとめ
- remap アプローチ (1) -- 概要
- remap アプローチ (2) -- 動作の流れ
- remap アプローチ (3) -- ページの read 処理
- remap アプローチ (4) -- ページの read 処理
- remap アプローチ (5) -- まとめ
- それでもうまく処理できないページ
- HugeTLB ページ
- 課題
- 質問は?

発表の流れ

- メモリホットプラグって何? 何のため?
- カーネルにおけるメモリ管理
 - 仮想記憶サブシステム
 - メモリ管理データ (zone, page)
- ページ強制解放, 移動
- 課題

メモリホットプラグの実現可能性

システム稼動中のメモリ追加、メモリ切り離しは可能

- **ハードウェアのサポートは必須**
- **OSのサポート**
 - **メモリ領域の動的変更への追従**
`alloc_page`
 - **削除予定のメモリ領域上に配置されているデータの退避**

メモリホットプラグの効果

- 可用性の向上

- ・ システム負荷に合わせたメモリの動的増設、切り離しが可能
- ・ システム稼動中のメモリの交換が可能



基幹システムへの適用領域拡大

ほかには...

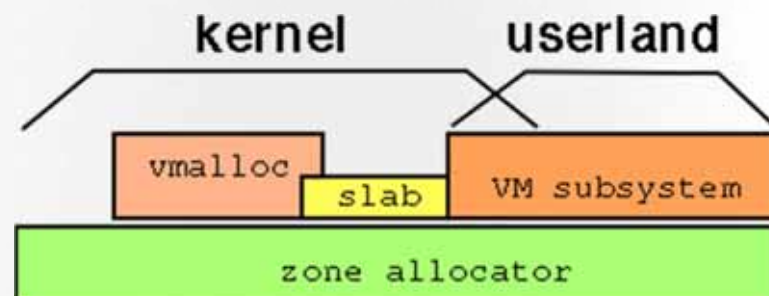
- process migration (NUMA用)
- 省電力(組み込み用途)
 - ・ ページを DRAM チップ間で最適配置 (研究レベル?)
- マルチプロセッサ SoC

カーネルによるメモリ管理

大きく分けて:

1. カーネル内部データ
slab allocator, vmalloc
2. プロセスメモリ
3. ファイルキャッシュ

2. と 3. はほとんど同じ扱い => VM subsystem



VM (仮想記憶)

- VMが提供する機能

- プロセス毎に独立なアドレス空間
- プロセス間の物理メモリの共有
- ページキャッシュ
- スワップ
- デマンドロード

- ハードウェアサポート (MMU)

ページ単位(PAGE_SIZE (4kB, 8kB, ...) の粒度)

で以下の機能:

- アドレス変換

page table, PTE (page table entry)

movl &edx, (&eax)

||

0x40001234



0xabcd234

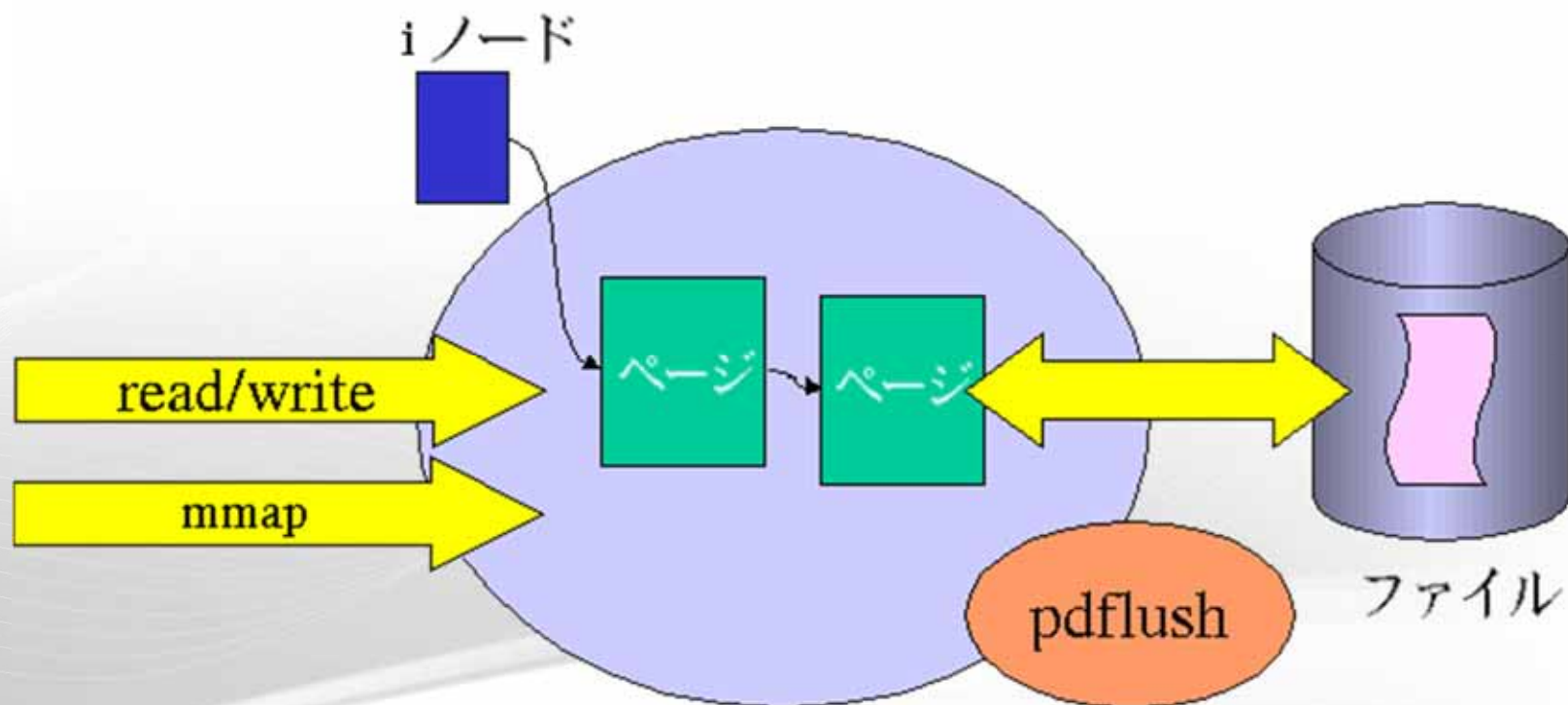
- ページ保護

読み書き禁止, 書きこみ禁止, 実行禁止など

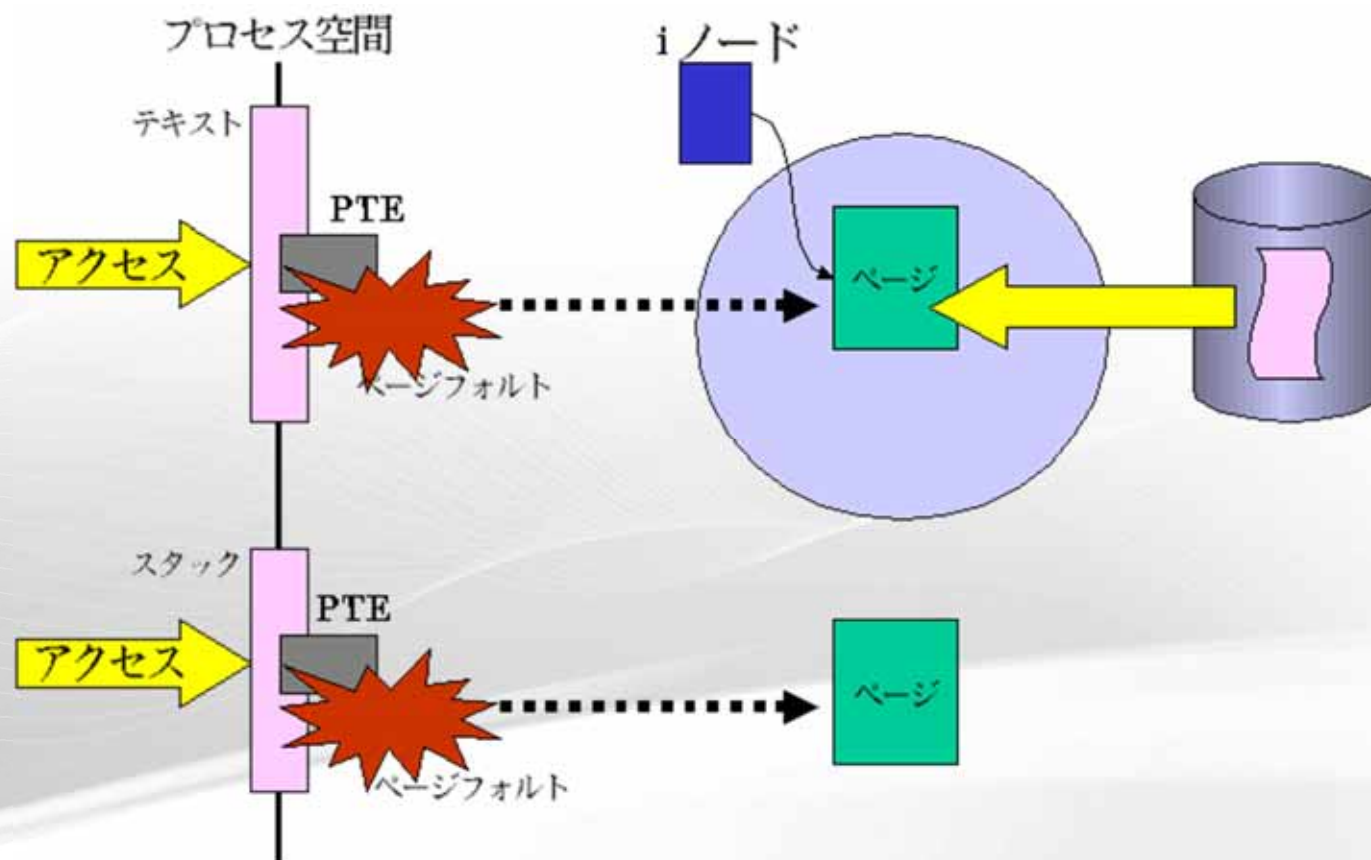
- Fault

保護違反が発生するとカーネルに処理が移行

LinuxのVM -- ページキャッシング

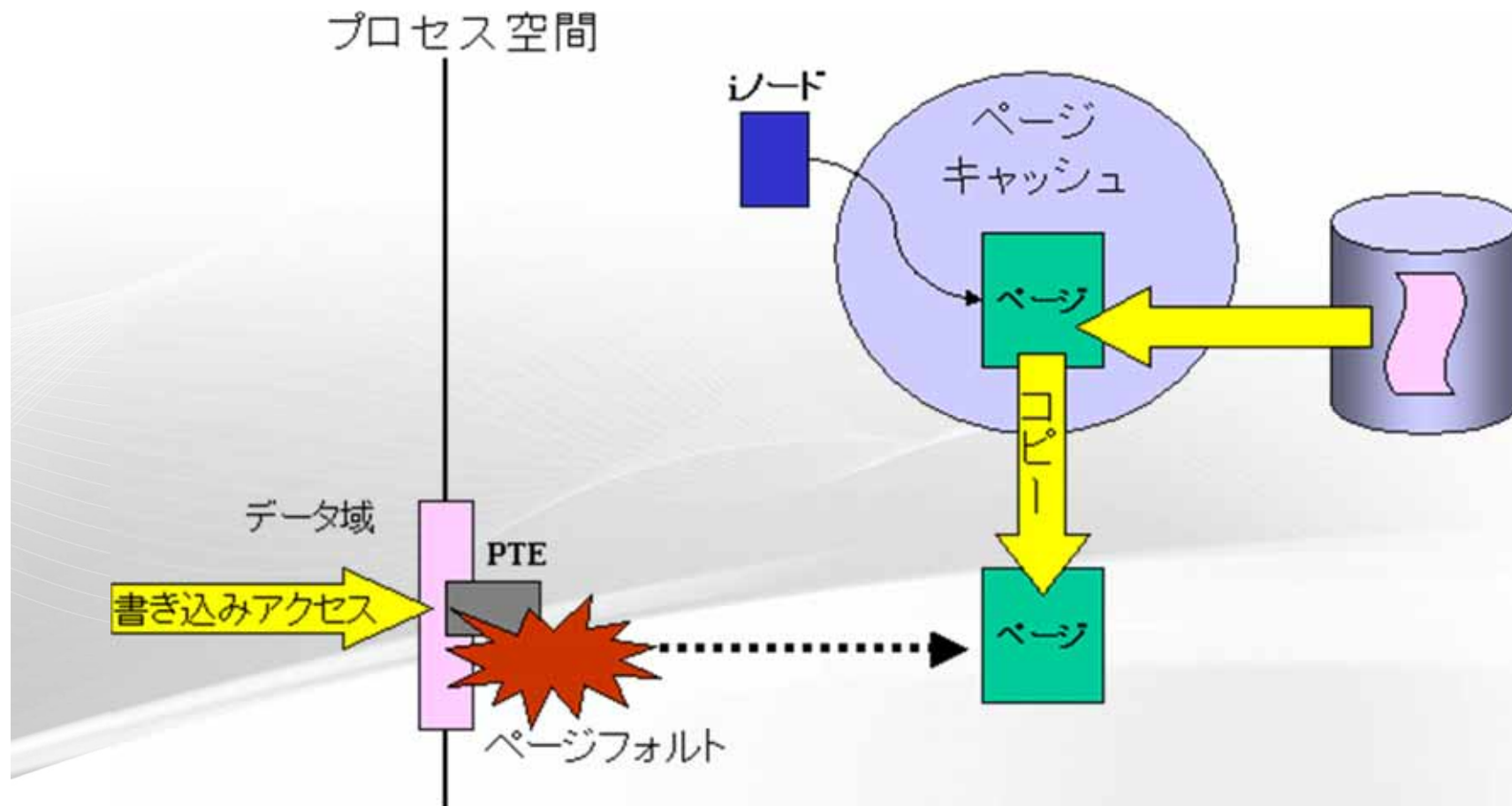


LinuxのVM -- デマンドロード(1)

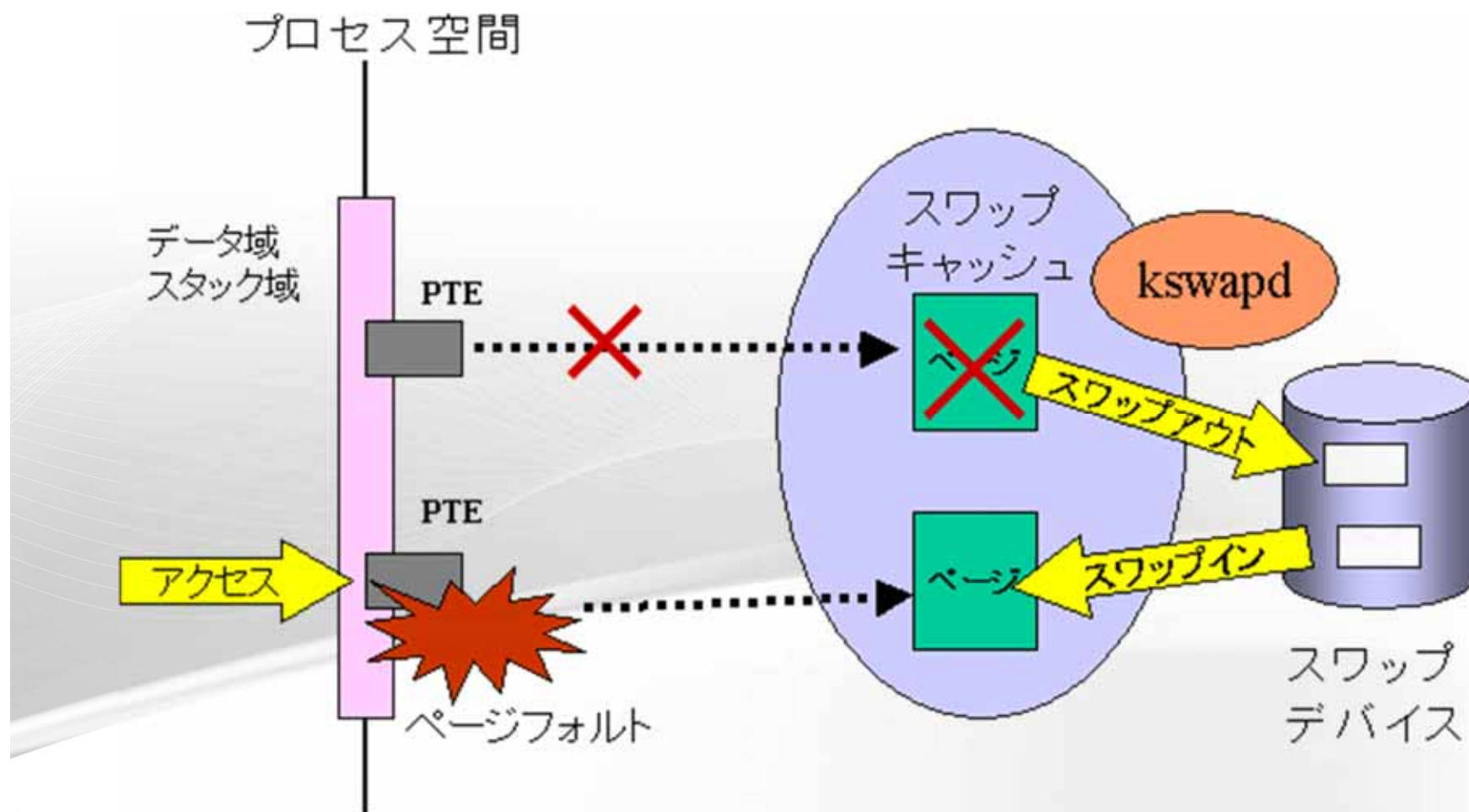


- find_vma()
- do_file_page()
- expand_stack()

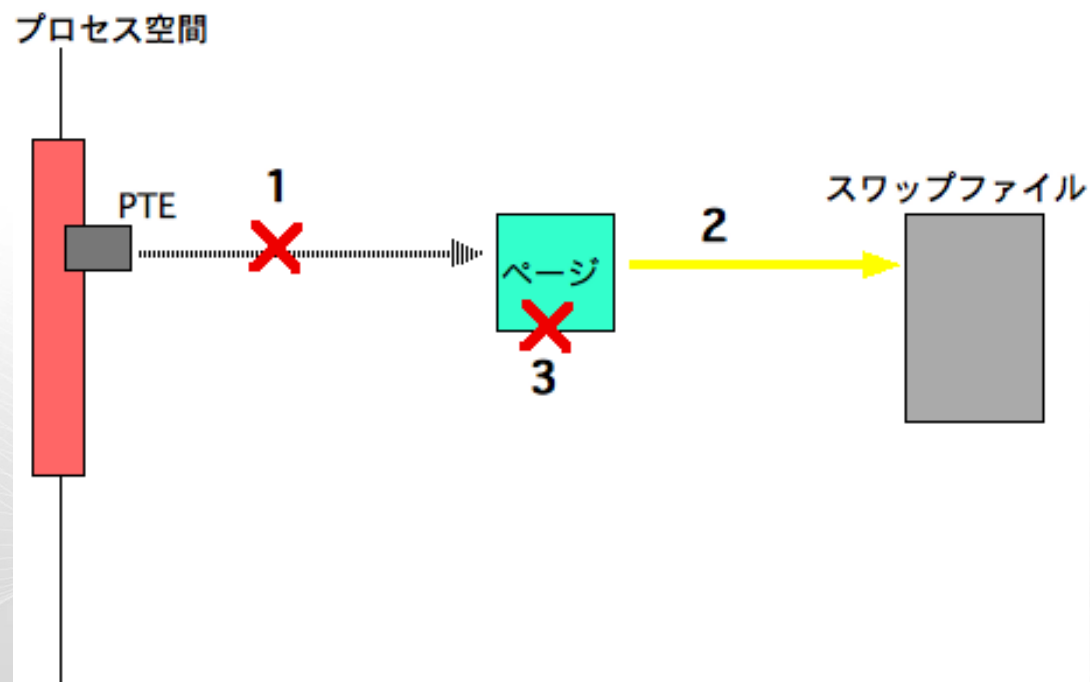
LinuxのVM -- デマンドロード(2)



LinuxのVM -- スワッピング



LinuxのVM -- スワップアウト



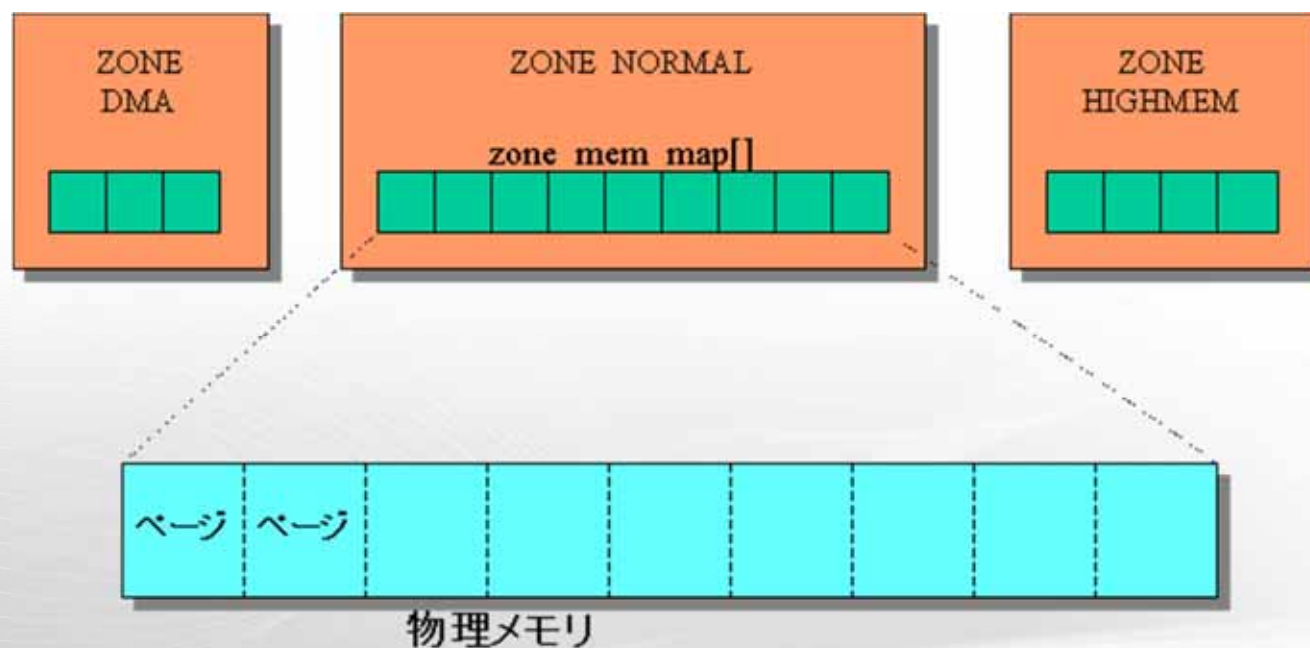
1. PTEを無効化
swap-in時のためにスワップエントリをPTEに埋めこむ
2. ページの内容をファイルに書き出す
3. ページを解放

ページアウトもほとんど同じ処理

LinuxのVM -- 内部データ構造

- radix tree
struct address_space とファイルオフセットから対応するページキャッシュを求めるためのもの
- mapping->a_ops->writepage
 - swap_writepage
 - shmem_writepage
 - ext2_writepage
 - ...

ゾーン -- ページの管理単位



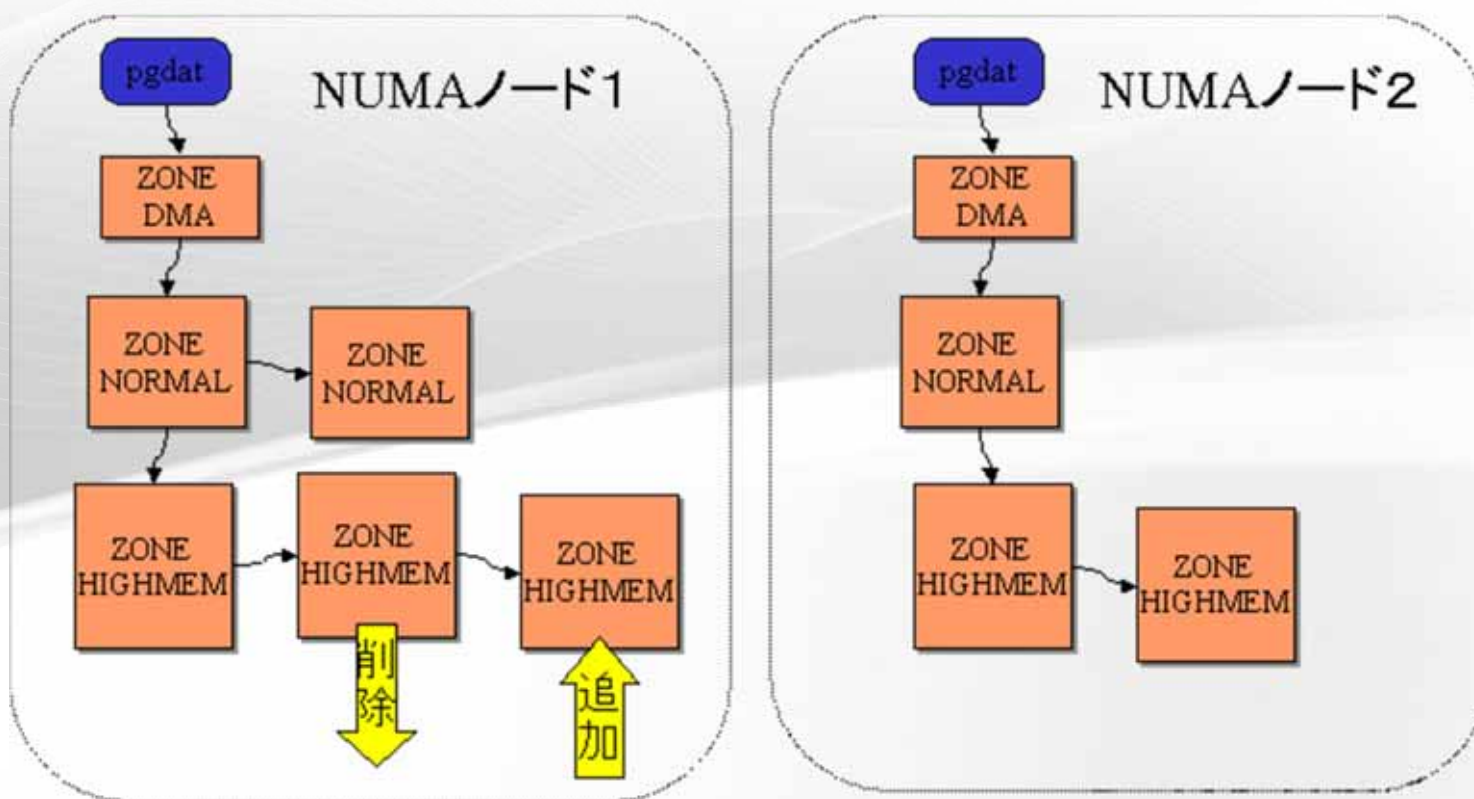
zone**構造体**: LRUリスト

- zone_mem_map[]
- ...

zone**の属性** (dma, normal, highmem)

ゾーンの追加と削除

- 例えばこんな感じにする
- NUMA, DISCONTIGMEMに注意
- zoneの参照は頻繁に起きるので注意



メモリ削除の課題

メモリ削除 その上のデータの退避

1. カーネルテキスト、カーネルデータ
2. ページキャッシュ
3. プロセス用メモリ

1 はあきらめる。(hardwareまかせ)

退避先:

- ディスク
- (他のゾーンの)メモリ

swapoutアプローチ(1) -- 回収可能なページ

kswapdを使ってページを解放する

- **スワップアウト処理で回収可能なページ**
 - ・ ページキャッシュとプロセスメモリ
highmemゾーンは基本的に解放可能なものだけなので
対象を highmem に限定する
- **対象はhighmemゾーン**
 - ・ kswapdの対象外のページはnormalゾーンへ
 - vmalloc域用ページ
 - ページテーブル

swapoutアプローチ(2) -- なぜhighmem?

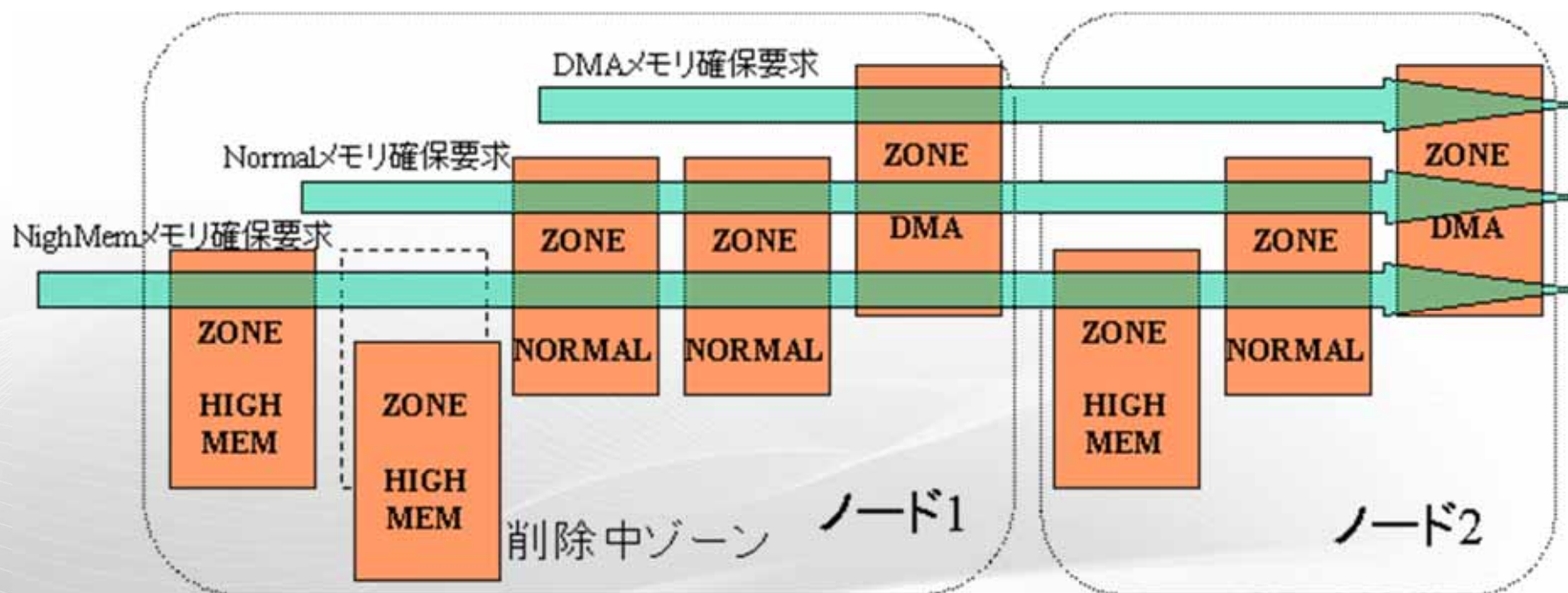
highmemは本来 32bit CPU 用

- dma, normal はストレートマップ
- 常にmapされているわけではない
kmap, kunmap
- なので page 構造体のポインタ経由でしか参照されない



強制解放可能なゾーンという意味をもたせる

swapoutアプローチ(3) -- alloc_page禁止



解放処理を始める前に対象ゾーンからの
alloc_page を禁止

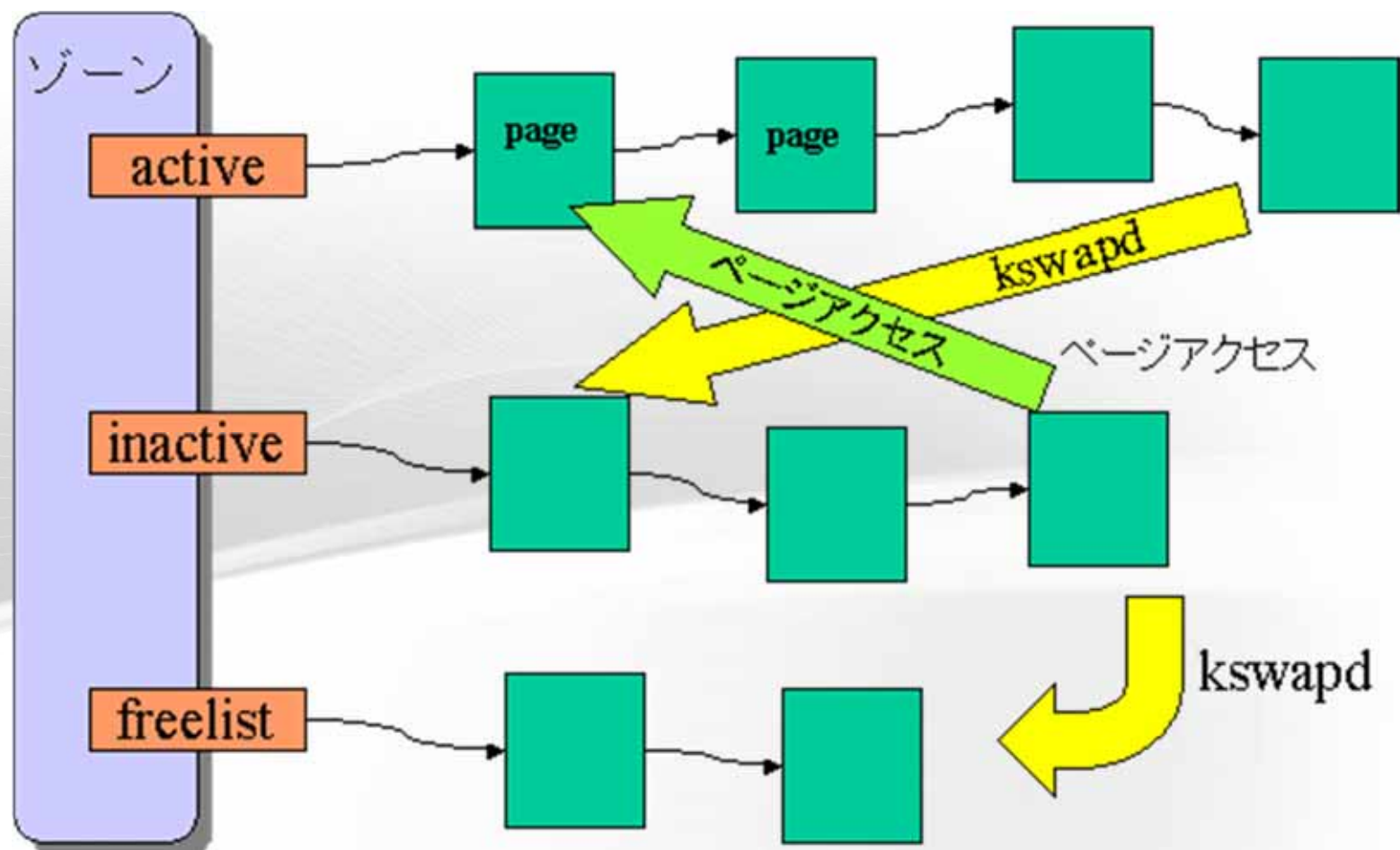
swapoutアプローチ(4) -- kswapdの閾値

kswapd は普通はそんなに動かない (重いから)
解放対象ゾーンの閾値を変更



swapoutアプローチ(5) -- LRUリスト

おさらい



swapoutアプローチ(6) -- 強制解放

参照頻度の高いページは解放しづらい

- ページ参照フラグ(PG_referenced)
- PTE中のアクセスビット^(†)

kswapd はこれらを見て最近使われたページを判断して、
解放処理 を中断



解放対象ゾーン上のページでは無視する

(†) MMUが自動的に立てる (あるいは fault ハンドラで処理)

swapout アプローチの限界 (1) -- kswapd の動作

kswapd の動作:

- shrink_listが解放されるべきページのリストを引数として呼ばれる
- 各ページに対して:
 1. dirty なら dirty bit をクリアして writepage を発行 (以下の処理は実行されない) (writepageがない場合ここで常に失敗)
 2. 結びつけられた buffer を解放
 3. page cache から除いて処理が完了

swapout アプローチは上の処理を繰り返す



- writepageと次の処理にタイムラグがある
- writeはブロックしない

swapout アプローチの限界 (2) -- page count

参照カウンタでページ解放を管理 (マクロ page_count)

- get_page
- put_page
if (! --page->count) release_page;
- **使っている範囲を** get_page ... put_page **で囲む**
- **別の名前(...):** page_cache_get, page_cache_release

例:

- PTE **に** map **するとき**
- read, write **システムコール内**
- (write**中は** writeback **フラグなので別**)

swapout アプローチ -- まとめ

限界はあるが、「だいたい」動く

問題となるページ:

- 頻繁にアクセスされるもの
- writepageがないもの(backing storeを持たないもの)
- ...



アクセスをブロックして memcpyする

- ディスクを経由しないので速い
- writepageがなくても大丈夫

remap アプローチ (1) -- 概要

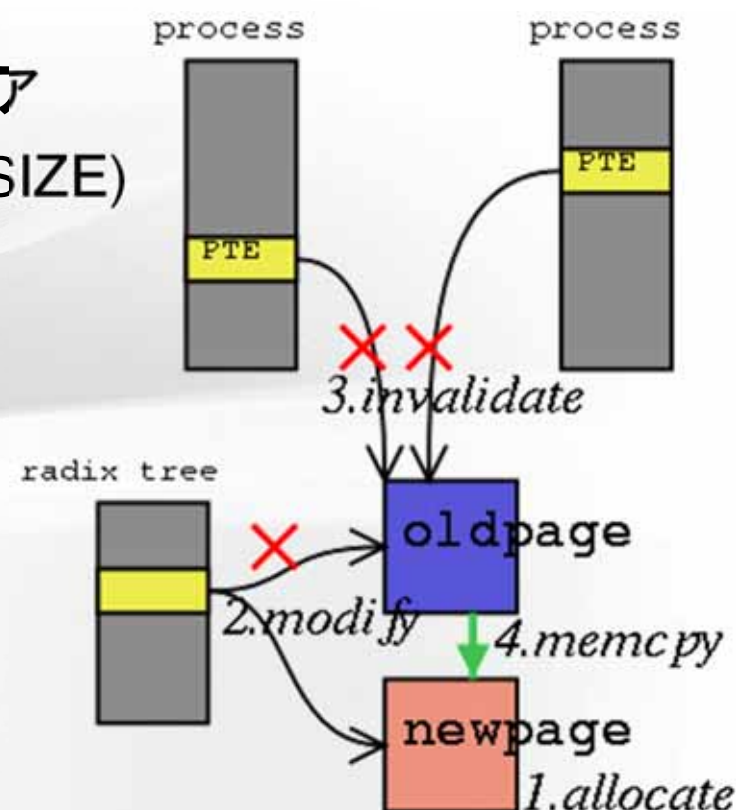
別のゾーンからページ(newpage)を確保して、
削除するページ (oldpage)をコピーしてすげかえる。

以下の状況が必要:

1. 誰も oldpage にアクセスしていない
2. (コピー前に) newpage がアクセスされない
 - 1.はユーザランド側は自明。カーネル側は page の参照数。
 - 2.は newpage をアクセスしようとするするとブロックするようにする。

remap アプローチ (2) -- 動作の流れ

1. newpageを確保. ブロックする状態
2. radix tree に 交換元のページ(oldpage) **のかわりに** newpage を登録
3. oldpage を参照する PTE すべてをクリア
4. memcpy(newpage, oldpage, PAGE_SIZE)
5. newpageのロックを解除



remap アプローチ (3) -- ページの read 処理

フォルトハンドラ起動 (page-in, swap-in):

- PTEが無効なので fault発生
- ディスクからの読みこみ先のページを確保
- readコマンド発行
完了ハンドラ(end_buffer_async_read, mpage_end_io_read) で
uptodate ビットを立てページのロックを外す
- 読み込み完了を待つ
(ページのロックが得られるまでsleep)
- プロセスが動作再開

```
handle_pte_fault
do_file_page
    (filemap_)populate
    filemap_getpage
    find_get_page
    if (page == NULL)
    page_cache_read
    if (! PageUptodate()){
        lock_page
        readpage
        wait_on_page_locked
    }
```

remap アプローチ (4) -- ページの read 処理

- read システムコール
やってることはほとんど同じ

```
do_generic_mapping_read
    find_get_page
    if (PageUptodate())
return;
    lock_page
    readpage
    wait_on_page_locked
```

- write では find_lock_page するのでそこでブロック



**newpage は !PageUptodate && PageLocked
にしておけばよい。これは「読み込み中」**

remap アプローチ (5) -- まとめ

- ページキャッシュについては以上の通り
- プロセスメモリの場合:
 - スワップを管理する radix tree に登録
 - 必要なら) add_to_swap
 - PTE を無効にする際 swap entry アドレスを書く
- sysfsのsymlinkのページは動いた

それでもうまく処理できないページ

- unmapしてはいけないもの
mlock, aioイベントテーブル
- faultが実装されていないもの
hugeTLBページ
- count が 0 にならないもの
NFS中ページ

HugeTLB ページ

- 代替領域の確保が大変
- いまある実装はおかしい
- remapを使って領域を実行時に確保？

課題

- 周りの人を巻きこむ
2.7 に向けて. main tree にフィードバック
- remap, swapout の併用のさせかた
- 動かさないページ対策
 - HugeTLB page
 - やればできる
 - 変なfilesystem
 - NFS とか NFS とかジャーナリングファイルシステム とか
 - 行儀のわるいデバイスドライバ
- normal ゾーンの memory pressure 低減
HIGHPTTE

質問は？

Powered by:

- SlideML (<http://www.slideml.org/>)
- mozilla (<http://www.mozilla.org/>)