

# オペレーティングシステム

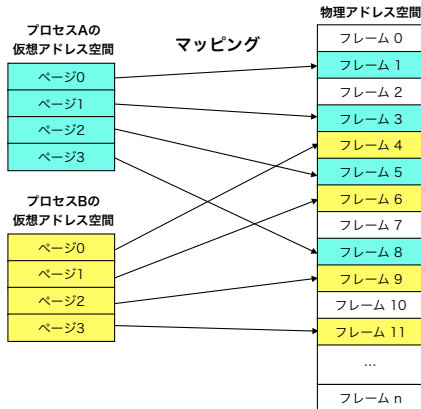
## 第 12 章 ページング

<https://github.com/tctsigemura/OSTextBook>

ページングは以下のようなメモリ管理方式である.

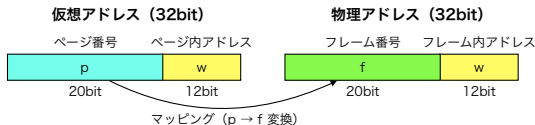
- メモリより広い仮想アドレス空間を使用できる.
- 外部フラグメンテーションを生じない.
- メモリコンパクションが不要である.
- Windows, macOS, Linux 等, 現代の OS のほとんどが採用している.
- 用語
  - **ページ**: 仮想アドレス空間をページに分割する.
  - **フレーム**: 物理アドレス空間をページに分割する.

## 基本概念



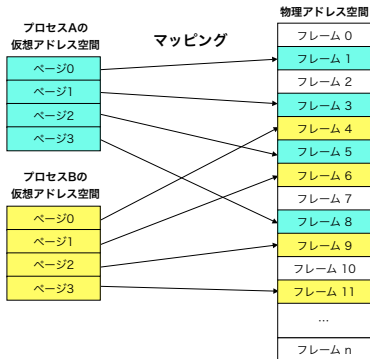
- ページをフレームにマッピングする.
- ページサイズとフレームサイズは同じ.
- どのフレームも任意プロセスの任意ページにマッピングできる.

# ページとフレーム



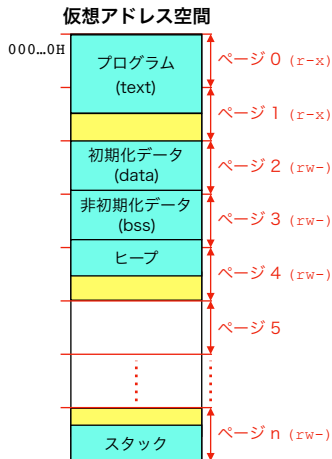
- 仮想アドレスの上位ビットがページ番号 (p)
- 仮想アドレスの下位ビットがページ内アドレス (w)
- ページサイズは2の累乗にする.
- ページ内アドレスがwビットならページサイズは $2^w$  バイト
- 物理アドレスの上位ビットがフレーム番号 (f)
- 物理アドレスの下位ビットがフレーム内アドレス (w)
- ページ内アドレスとフレーム内アドレスは同じ (w)
- p を f に変換することでページをフレームにマッピングする.
- p と f のビット数は異なっても良い.

# マッピング関数



- $p \rightarrow f$  変換関数をマッピング関数と呼ぶ.
- **ページテーブル** (表) を用いて実装する.
- MMU がページテーブルを参照してマッピングする.
- プロセス毎にマッピング関数は異なる.
- ディスパッチャがMMU を操作しマッピングを入れ換える.

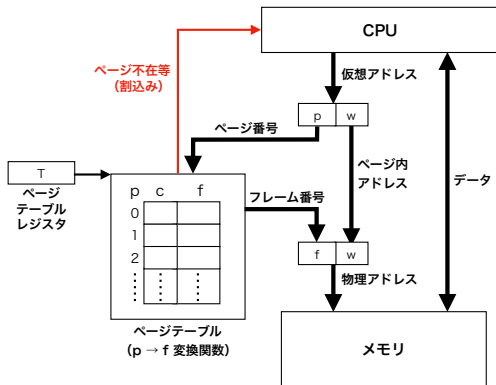
## フラグメンテーション



- **外部フラグメンテーション**は解決した。
  - どのフレームでも任意のプロセスの任意のページにマッピングできる。
  - メモリコンパクションも不要になった。
- **内部フラグメンテーション**が発生する。  
ページ毎にメモリ保護モードを設定する。
  - プログラム領域は **r-x** にする。
  - データ領域は **rw-** にする。
  - **あな**部分にはフレームを割当てない。  
(sparse address spaces)
  - スタック領域も **rw-** にする。

領域サイズはページの倍数ではない。  
フラグメント部分のアクセスは不正だが  
検知できない。

# ページング機構の概要



- ページテーブルの一つの**エントリ**をページ番号 (p) で選択する。
- 選択したエントリに格納されているフレーム番号 (f) を取り出す。
- フレーム番号 (f) とページ内アドレス (w) を結合し物理アドレスにする。

# ページテーブルエントリ

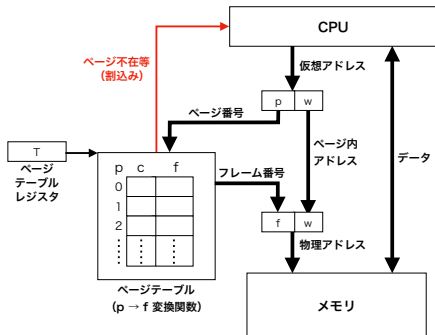
- *f*フィールド：フレーム番号
- *c*フィールド：制御

名称	ビット数	意 味
V (Valid)	1	フレームが割り付けられている。
R (Reference)	1	ページの内容が参照された。
D (Dirty)	1	ページの内容が変更された。
RWX (Read/Write/eXecute)	3	許されるアクセス方法。

- V=0 ならページ不在割込み
- R はページの使用頻度の測定等を使用
- D=0 ならスワップアウト不要
- RWX によりメモリ保護 (メモリ保護割込み)

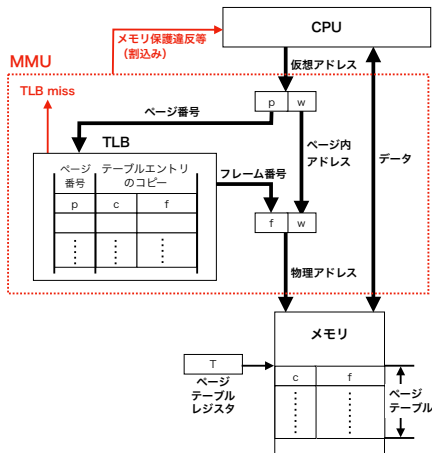


## ページテーブル



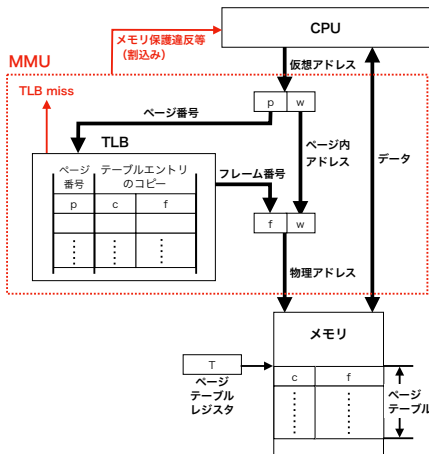
- ページテーブルはかなり大きな表である。
- ページ番号が20ビットなら  $2^{20} = 1\text{Mi}$  エントリ
- エントリのサイズが4バイトと仮定すると全体で4MiB
- プロセス毎に必要なのでディスパッチの度にロードするのも大変
- **ページテーブルレジスタ**にアドレスを記録しメモリ上に置く

# TLB (Translation Look-aside Buffer)



- メモリ上にあるページテーブルにアクセスするには時間がかかる。
- 変換結果 (p と f の対応) を TLB にキャッシュする。
- TLB (数十から数千エントリ) は高速な連想メモリ。

# TLB (Translation Look-aside Buffer)



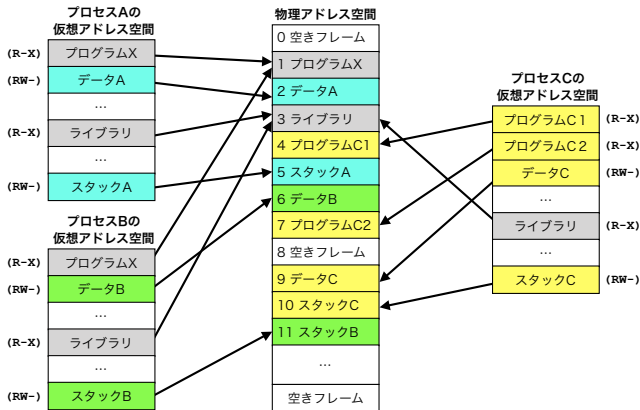
- 1) ページ番号 (p) で TLB を検索しエントリを選択する. (*TLB miss*)
- 2) RWX をチェックする. (メモリ保護例外)
- 3) フレーム番号 (f) を出力する.

- *TLB miss* のときページテーブルを検索すること.
- ハードウェアで自動的に行う場合  
ページテーブルレジスタからページテーブルの位置を知る.  
ハードウェアを用いることで高速化
- ソフトウェアで行う場合  
TLB miss 割込みを発生し OS に切替える.  
ハードウェアが単純 => チップ面積に余裕 =>  
TLB のエントリ数を増やす => TLB miss の頻度を低くする.
- ページテーブルのエントリ  $V=0$  の場合  
ページ不在割込みを発生
- ページテーブルのエントリ  $V=1$  の場合
  1. TLB のエントリの一つをページテーブルに書き戻す.
  2. TLB の空いたエントリにページテーブルからロード

# TLB エントリのクリア

- プロセススイッチのとき
- ページテーブルに変更があったとき
- TLB の内容は変更前のページテーブルのエントリなのでクリアする必要がある.
- TLB のクリアは**大きなペナルティ**を伴うので避けたい.
- TLB のエントリがプロセス番号を含む方式
- TLB のエントリを個別にクリアできる方式

## フレームの共用



- プロセスが変更しないページ (R-X) は共用できる.
- ページテーブルの操作により実現
- ライブラリは**位置独立コード**でなければならない.

# 位置独立コード

## 位置独立コードのイメージ

```
CALL  200,PC    // 200 番地先にあるサブルーチン実行
JMP   100,PC    // 100 番地先にジャンプする
LD     G0,4,FP   // ローカル変数はスタック上
ST     G0,40,G11 // グローバル変数はレジスタ相対で参照
```

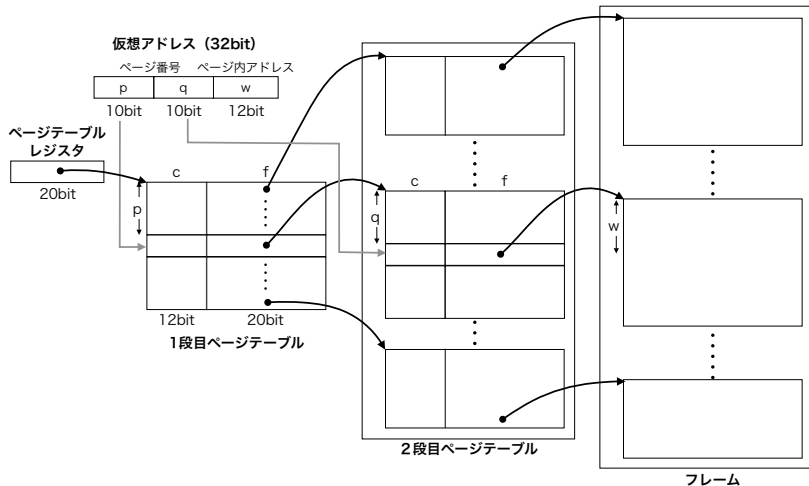
- ライブラリはマッピングされる仮想アドレスが変化する.
- どのアドレスにマッピングされても大丈夫なプログラム  
=> **位置独立コード**
- PC 相対で JMP や CALL を行う.
- データはレジスタをベースにアクセスする.

# ページテーブルの編成方法

- ページテーブルは大きくなりすぎる. (32 ビット CPU の例)
  - 仮想アドレス 32 ビット, ページサイズ 4KiB, エントリ 4B の例
$$2^{32} \div 4KiB = 2^{32} \div 2^{12} = 2^{20} = 1Mi \text{ エントリ}$$
$$1Mi \text{ エントリ} \times 4B = 4MiB$$
  - 32 ビット CPU の普及が始まった当時の PC は,  
メモリを 4MiB ~ 16MiB しか搭載していなかった.
- ページテーブルは大きくなりすぎる. (64 ビット CPU の例)
  - 仮想アドレス 48 ビット, ページサイズ 4KiB, エントリ 8B の例
$$2^{48} \div 4KiB = 2^{48} \div 2^{12} = 2^{36} = 64Gi \text{ エントリ}$$
$$64Gi \text{ エントリ} \times 8B = 512GiB$$
  - 現代の 64 ビット PC のメモリは, 4GiB ~ 16GiB 程度?
- ページテーブルを小さくする工夫が必要!!

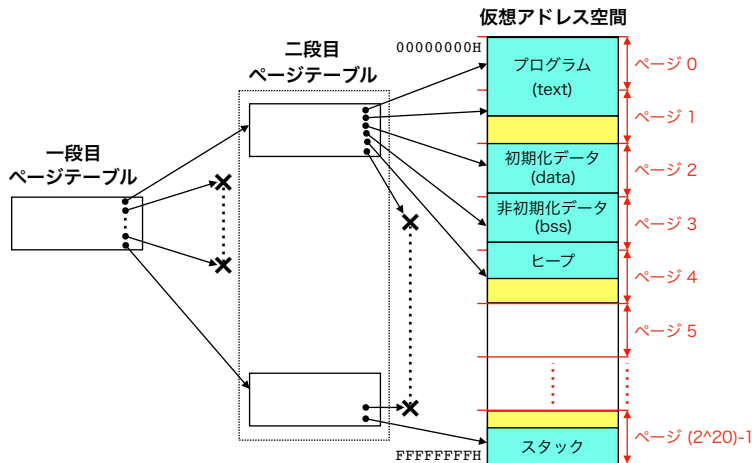


## 二段のページテーブル (IA-32 の例)



- 一段目のページテーブルサイズ 4KiB = フレームサイズ
- 二段目のページテーブルの区画サイズ 4KiB = フレームサイズ

# ページテーブルフレームの節約

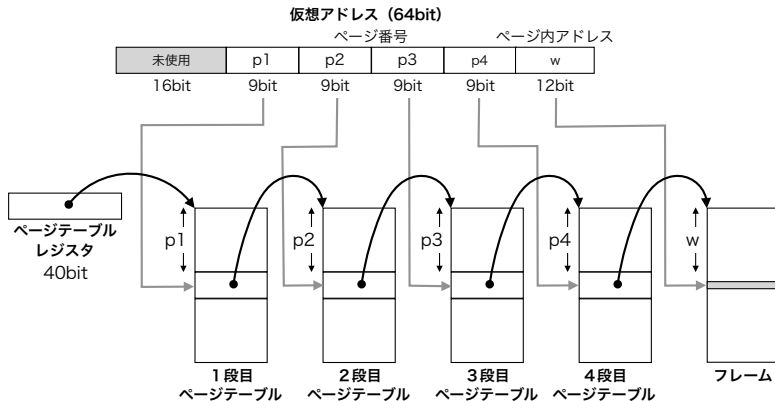


- 仮想アドレス空間の**あな**部分の二段目を省略
- 一段目のページテーブルエントリの V=0 にする.
- 従来 1,025 フレーム => 3 フレーム

## 64 ビット仮想アドレス空間 (x86-64 の例)

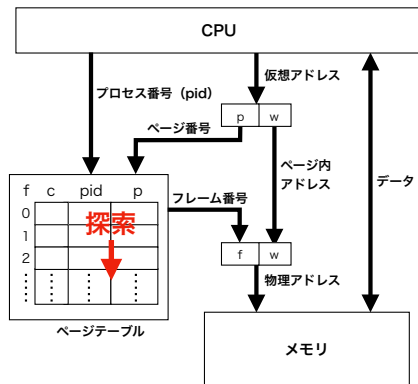
- 実質 48 ビット仮想アドレス => 256TiB (十分大きい)
- 仮に二段のページテーブルならページテーブルの区画は 18 ビット (p), 18 ビット (q), 12 ビット (w) と仮定  
エントリサイズ = 8B と仮定  
 $2^{18} \times 8B = 2MiB$
- プロセスあたり最低でも 3 区画必要  
 $2MiB \times 3 = 6MiB$
- 400 個のプロセスがあったとすると  
 $6MiB \times 400 = 2.4GiB$  (8GiB の 30%)
- 二段のページテーブルでは区画が大きくなりすぎる.
- 区画を小さくするために段数を多くする.

# 四段のページテーブル (x86-64 の例)



- ページサイズ (フレームサイズ) は 4KiB
- ページテーブルの区画は  $2^9 \times 8B = 4KiB$
- ページテーブルは最低 7 フレーム (28KiB)
- 400 プロセスでも約 11MiB で済む (8GiB の 0.14%)

# 逆引きページテーブル



- テーブルでフレーム番号とページ番号の立場が逆転（逆引き）
- システム全体でページテーブル一つ（プロセス毎ではない）
- どの仮想アドレス空間のエントリか識別するための pid あり

# 逆引きページテーブル

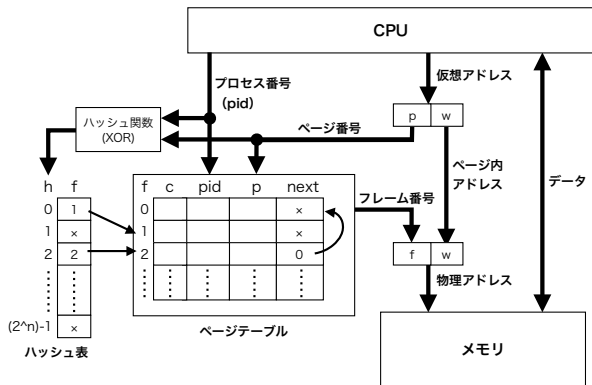
## ページテーブルのサイズ

- 8GiB のメモリを 4KiB のページで分割する場合のエントリ数
$$8GiB \div 4KiB = 2^{33} \div 2^{12} = 2Mi \text{ エントリ}$$
- 1 エントリ 8 バイト仮定すると
$$2Mi \times 8B = 12MiB$$
- システム全体で 12MiB で済む (8GiB の 0.2%)

## Page Table Walk

- CPU は仮想アドレスの他に pid (プロセス番号を出力)
- ページテーブルを pid と p (ページ番号) で探索する
- 線形探索などを用いると遅くて実用にならない

# 逆引きページテーブル (IBM 801 の例)



- ハッシュ表を用いて探索を高速化
- ハッシュ表の大きさは二の累乗 ( $0 \sim 2^n - 1$ )
- ページテーブルは next を使用してチェーンを作る

# 逆引きページテーブル (IBM 801 の例)

## ハッシュ表を用いた *Page Table Walk*

- pid と p を用いてハッシュ関数を計算する ( $h \leftarrow f(pid, p)$ )  
(ハッシュ関数は pid と p の XOR。。。速度優先)
- ハッシュ表の第 h エントリを見る
  - 空 (図では ×) ならページ不在 (**割込み!**)
  - 空でなければページテーブルのインデクス (f)
- ページテーブルの第 f エントリの内容を見る
  - pid , p が一致 → この時の f をフレーム番号にする (**完了!**)
  - pid , p が一致しない → next を見る
    - 空 (図では ×) ならページ不在 (**割込み!**)
    - 空でなければ  
ページテーブルのインデクス (f) を更新して再度トライ

TLB : 不可欠!



# 練習問題

- (1) 一回のメモリアクセス時間に 1ns, page table walk に 2ns かかるとする, TLB のヒット率が 90 パーセントの時の平均メモリアクセス時間を計算しなさい.
- (2) 図 12.7 において,  $p = 1$  の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい.
- (3) 図 12.7 において,  $p = 1$ ,  $q = 1$  の仮想アドレスの範囲を 8 桁の 16 進数で答えなさい.
- (4) 逆引きページテーブルを用いる場合, TLB に格納すべき最低限の情報範囲を考察しなさい.
- (5) 図 12.11 に,  $pid = 3$ ,  $p = 2$  のページがフレーム 1 にマッピングされるようなページテーブルの状態を書き込みなさい.
- (6) 逆引きページテーブルを用いるシステムで, プロセス間でページの共有が可能か考察しなさい.