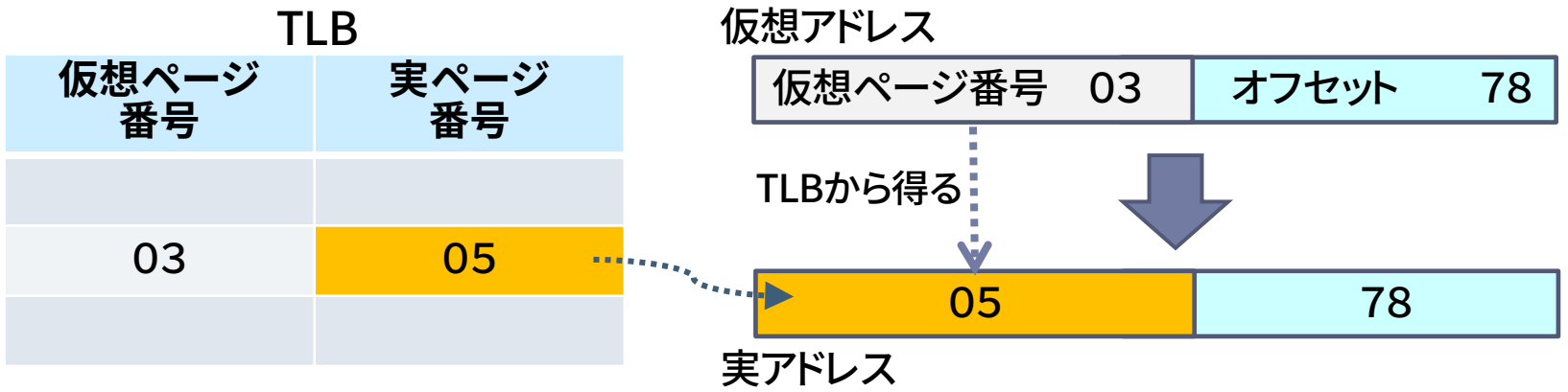
メモリ管理の補足です。

オペレーティングシステム
（2024年 第６回の補足）

メモリ管理の補足

アドレス変換の高速化

- アドレス変換キャッシュ
  - ページテーブルを介したアドレス変換では、1回のページアクセスに複数回のメモリアクセスが必要 … 実行が遅くなる
  - アドレス変換の高速化のため、よく使われる仮想ページに高速にアクセスできるよう、最近使われた仮想ページ番号とそれに対応する実ページ番号をキャッシュしておくハードウェア機構（連想メモリ）を持っている
    （TLB：トランスレーションルックアサイドバッファ）



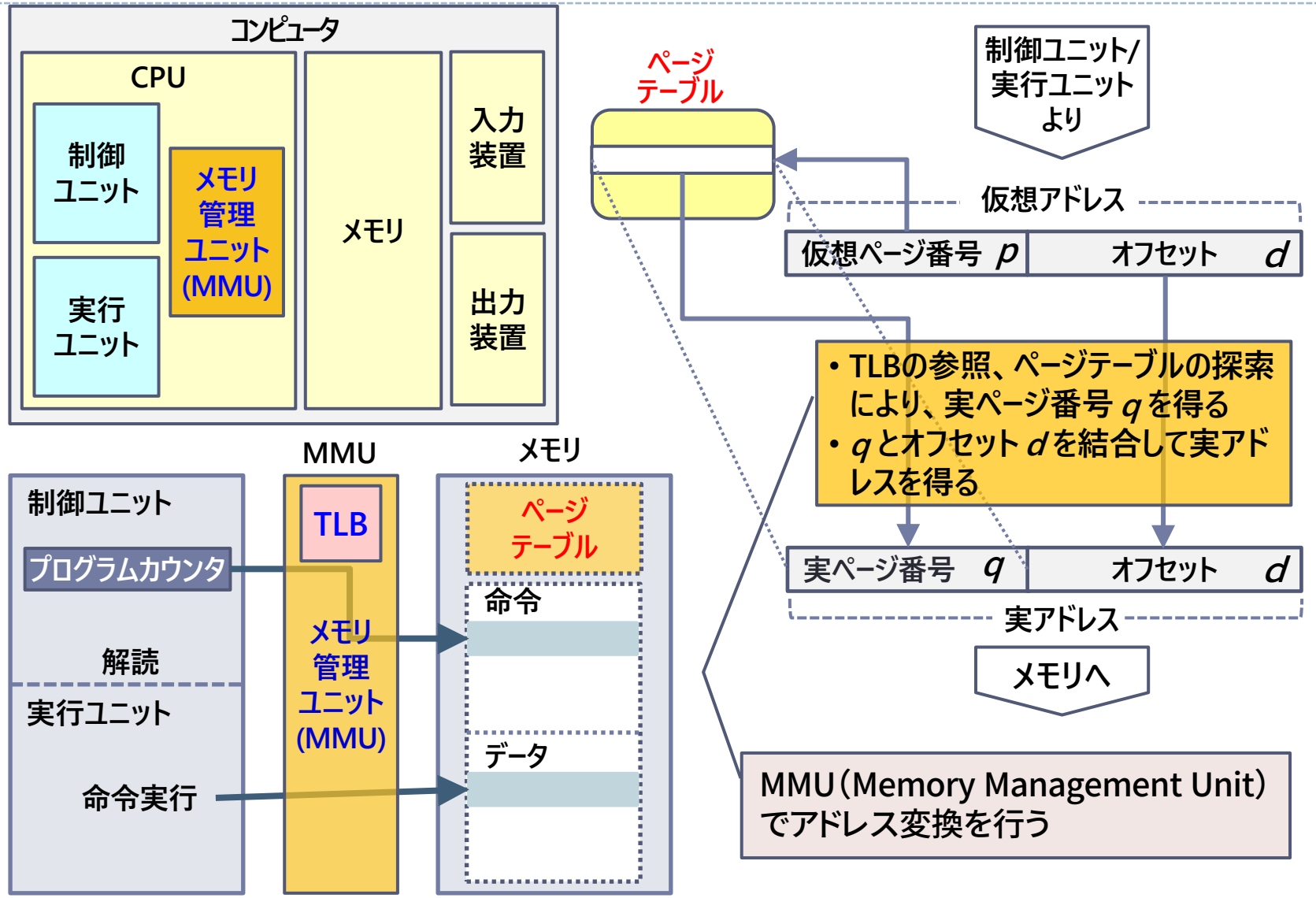
  - TLBではすべての対応情報を保持できない
    メモリ内のページテーブルにすべての対応情報を保持し、TLBでその一部をキャッシュする

高速に実アドレスを得るため、仮想ページ番号はまず、専用のハードウェア機構（アドレス変換キャッシュ）を用いて高速に変換され実ページ番号が得られます。
このアドレス変換キャッシュは、TLB（Translation Lookaside Buffer）と呼ばれます。

TLBは高速ですが、小容量（数十～数百個程度）なので対応するエントリがない場合、メモリ内のページテーブルが引かれて、実ページ番号が得られ、これとオフセットを加え（連結し）て実アドレスを得ます。

「キャッシュ」とは、一度利用したデータ（直近に読込んだものや使用頻度が高いもの）を高速に再利用するために、より高速な記憶装置に一時的に記憶しておくことです。
また、その際に使われる高速な記憶装置や、一時的に記憶されたデータそのものを表すこともあります。

アドレス変換(DAT)についての補足



CPUにはメモリ管理ユニット(MMU)があり、ここでアドレス変換(DAT)が行われることを、第3回でお話ししました。

制御ユニット/実行ユニットから出てくるプログラムのアドレス(仮想アドレス)は、MMUで変換されて実アドレスとなってメモリを参照します。この際、MMUの中でTLBとページテーブルを参照して実アドレスを生成しています。

図の右半分にあるように、仮想アドレス $<p, d>$ はまず TLBを参照して p に対応するエントリがTLBにあれば、実ページ番号 q を得て、実アドレス $<q, d>$ を得ます。TLBに p に対するエントリが存在しなければ、ページテーブルを引いて q を得ます。(さらに、TLBのエントリを更新する操作があります)

二次記憶の管理に関する補足

- バッキングストア
  ページの内容を格納するために、二次記憶装置上に確保される領域（スワップ領域、ページング領域）
  - Linuxの場合、専用のパーティションやスワップ用のファイルなどが使用される
- メモリマップトファイル
  プログラムやデータを格納しているファイルそのものをバッキングストアとして利用する方式
  - ファイルの内容を仮想アドレス空間に取り込んで、通常の仮想記憶領域と同じようにアクセスできる（メモリアクセスとしてプログラミングできる）
  - ページングの機構を使用するのでバッファリングが不要

仮想記憶を実現するのに、二次記憶装置上にプログラムやデータの実体を格納するための領域をスワップ領域やページング領域（総称してバッキングストア）と呼びます。

プログラムやデータを格納しているファイルそのものを仮想記憶のバッキングストアとして利用する方式も考えられ、これによりファイルの内容を仮想アドレス空間に取り込んで、通常の仮想記憶領域と同じようにアクセスできます。
このようなファイルのアクセス方法をメモリマップトファイルと呼びます。ファイルには先頭から順にアドレスをつけることができ、このアドレスを仮想アドレスに対応付けることで、ファイルの内容を仮想アドレス空間に取り込むことができます。

セグメンテーション

「プログラムへのメモリ割り付けが連続的な領域でなければならない」という制約を回避する手法として、前回

- メモリのページ化（ページング）
  - メモリ領域を一定サイズの小さなブロック（ページ）に区切る

方式を挙げた

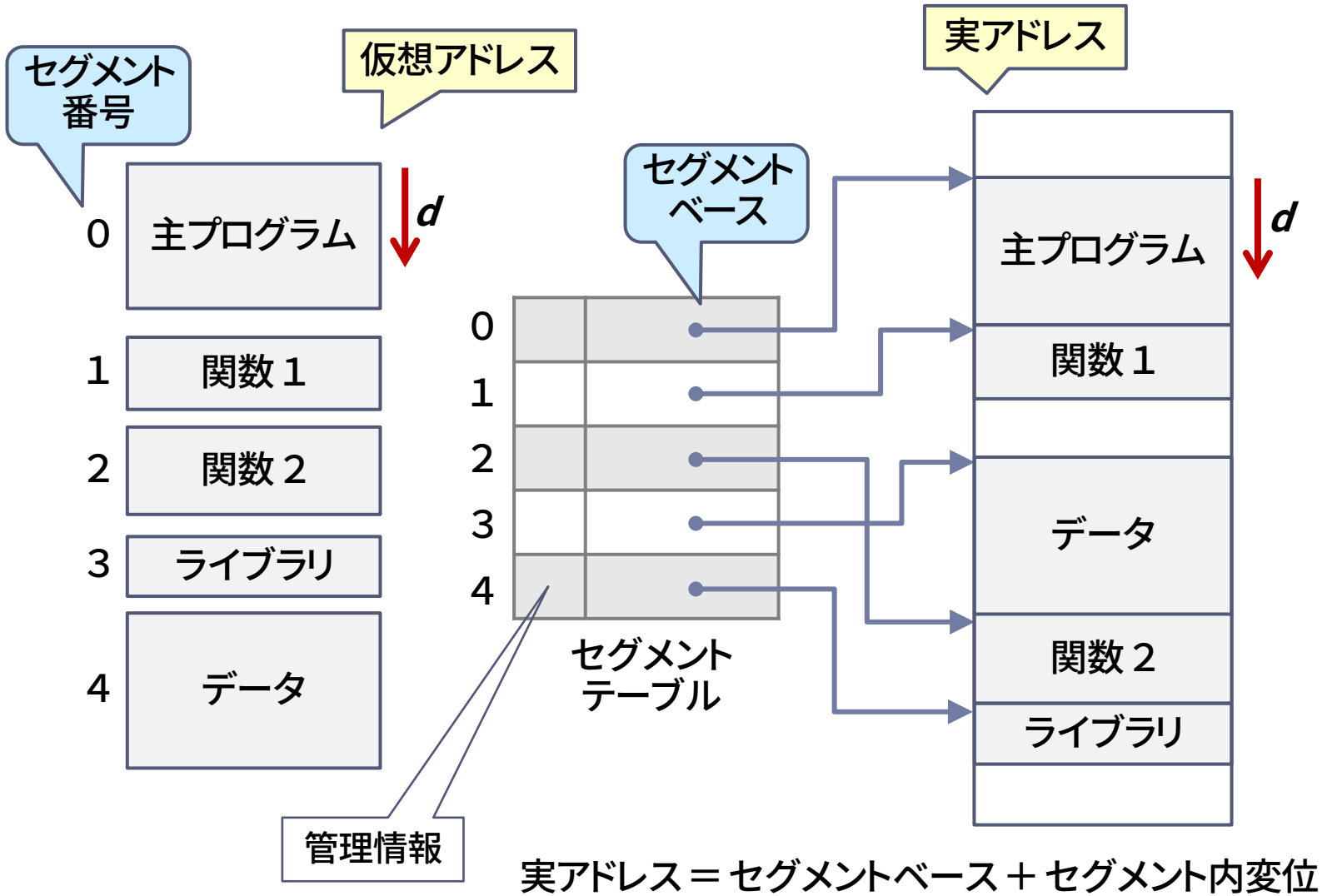
別の手法として、

- セグメント化（セグメンテーション）

空き領域が出ない、すなわち可変長の区画（セグメント）を複数用意するようにする方式がある

- プログラム作成者が、プログラムの内部構造に基づいて定義する「セグメント」を単位とする
- オーバーレイにおいて、モジュールの配置をユーザではなくOS（とハードウェア支援）で行うもの

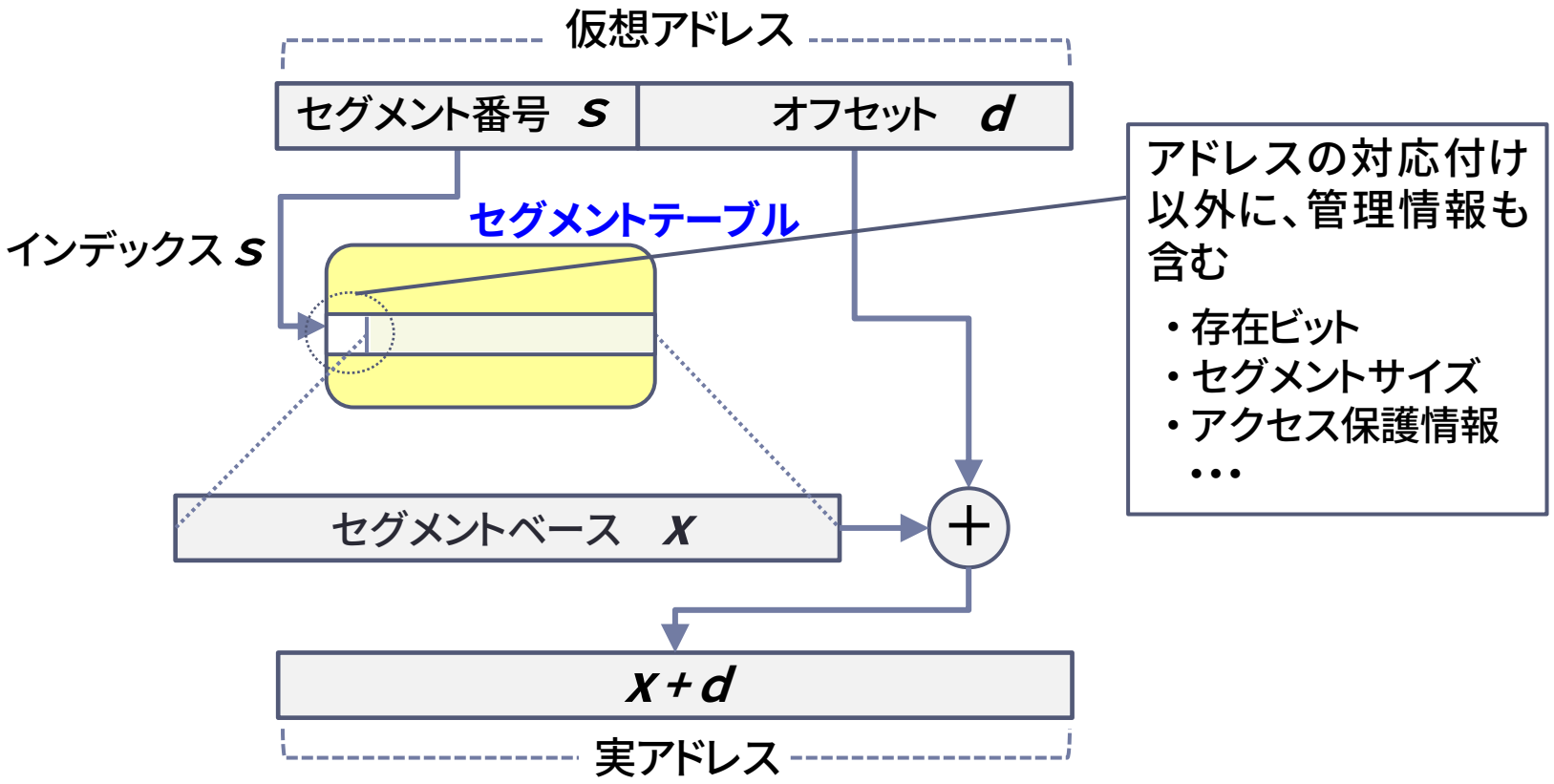
セグメンテーションでは、プログラムのモジュールやライブラリ、データなど論理的な構造に基づいて定義されるセグメントを割当ての単位とします。
第6回資料、p.15-16のオーバーレイにおいて、モジュールの配置をOSとハードウェアで行うものと考えられます。



仮想アドレス空間のセグメントに対して、それを識別するセグメント番号が付けられ、仮想アドレスは、セグメント番号とセグメント内の変位の2つの要素で構成されます。（ページングでのページ番号とページ内変位と対比させてください）

仮想アドレスから実アドレスへの変換は、セグメント番号でセグメントテーブルを引き、対応するセグメントの開始アドレスにセグメント内変位を加算することで行います。

- セグメンテーションにおけるアドレス変換
  - 仮想メモリから実メモリ空間への対応づけはセグメントテーブルと呼ばれる構造で実現



セグメント番号がインデクスとなってセグメントテーブルからセグメントのベースアドレスが得られ、これにオフセットdを加算することで実アドレスが得られます。ページングの時と違って、加算が入っています。

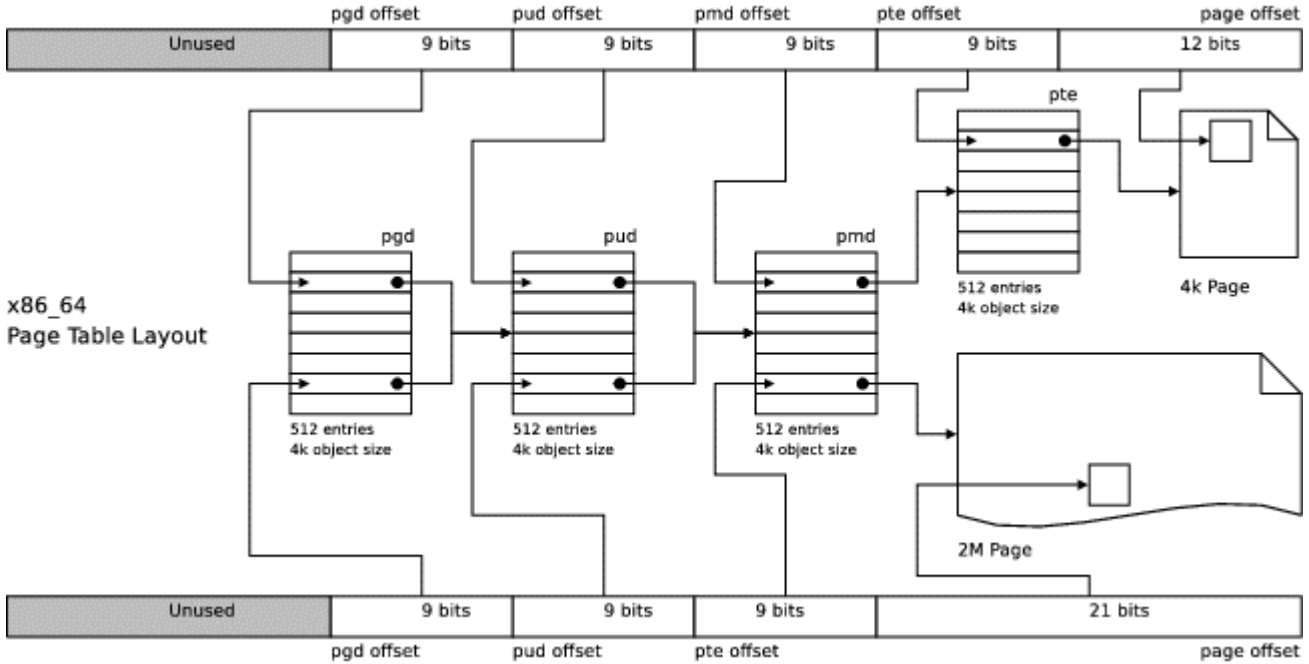
セグメントに対するアクセス手順の概要は以下です。ページングの処理概要と対比してください。
ハードウェア機構により、セグメント番号でセグメントテーブルが引かれます。アクセス保護情報によってアクセスが許されるか調べられます(許されない場合、例外が発生)

アクセス可の場合、存在ビットを調べます。そのセグメントがメモリに存在すれば、セグメントベースとオフセットを加えて実アドレスが得られます。このとき、オフセットとセグメントサイズを比べてオフセットが大きい場合はエラーとして割込みが発生します。

存在ビットによって、メモリに存在しないことが分かれば、割込み(セグメントフォールト)が発生します。セグメントフォールトが発生すると、二次記憶からセグメントが読み込まれて、実行を再開します。

大容量(64ビット級)アドレス空間でのページテーブル

- x86_64では仮想アドレスが48ビットで指定できる … 256TB空間
  - 4KBページで2段のページテーブルとすると、ページ番号部分は36ビット
    → 仮に、これを18ビットずつ2つの部分に分けると、1段のページテーブルが2^{18}エントリとなり現実的なサイズではなくなる
- そこで、4段のページテーブルを用いてページングを行う(x86_64の場合)
  - 12ビットオフセット(4KBページ)と9ビット(512エントリ)×4段のページテーブル



PageTableStructure - linux-mm.org Wiki より。図の下半分は、ページサイズが2MBのときのもの

現在、Windows PCなどでは64ビットアーキテクチャ(x86_64など)が使用され、32ビットよりはるかに大きなメモリ空間が利用できます。
このため、OSでは4段など多段のページテーブルを用いて管理します。

2段のページテーブルでは、18ビット分のエントリをもつテーブル1個のサイズは2MBです(アドレスが48ビットなので、1エントリのサイズも8バイトとなります)。
x86_64のアーキテクチャとしても4段のページテーブルを実現するハードウェアとなっています。

かなり以前のLinuxでは、前回スライドp.14の3段のモデルでした。バージョン2.6.11以降のLinuxでは4段のページテーブルを用いています。(IA-32の2段に合わせる場合は、中間のテーブル2段分をスキップする形になります)

さらに、最近のx86_64では、57ビットで仮想アドレスを指定でき、128PB空間となります。このため、さらに9ビット分のテーブルを加えた5段のアーキテクチャとし、OSもそれを利用できるようにしています。