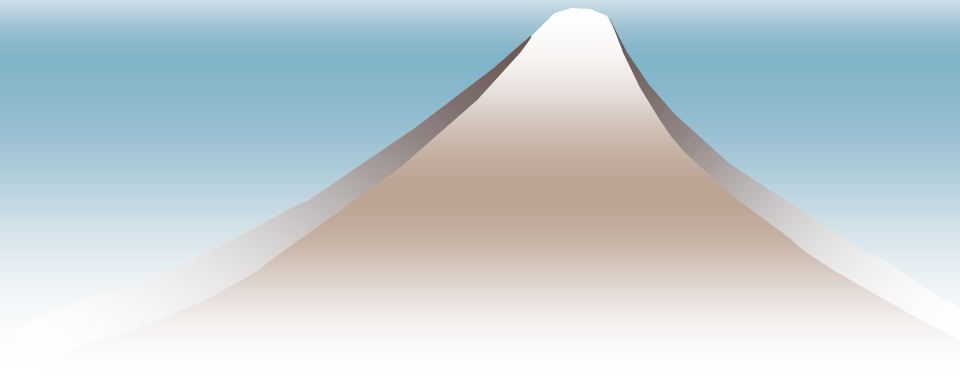
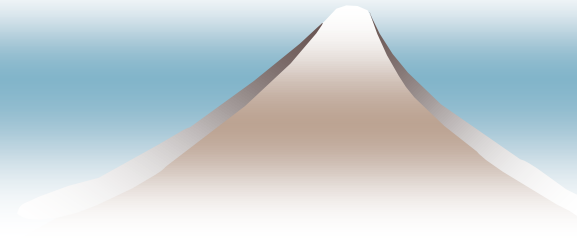


メモリ管理(1)



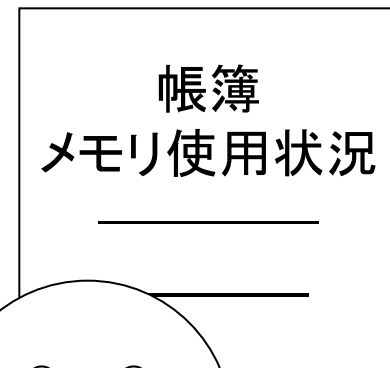
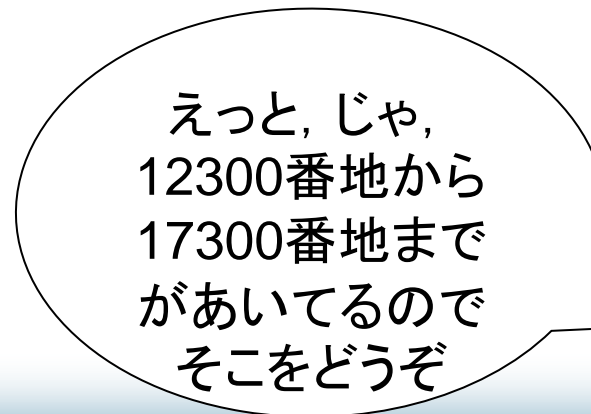
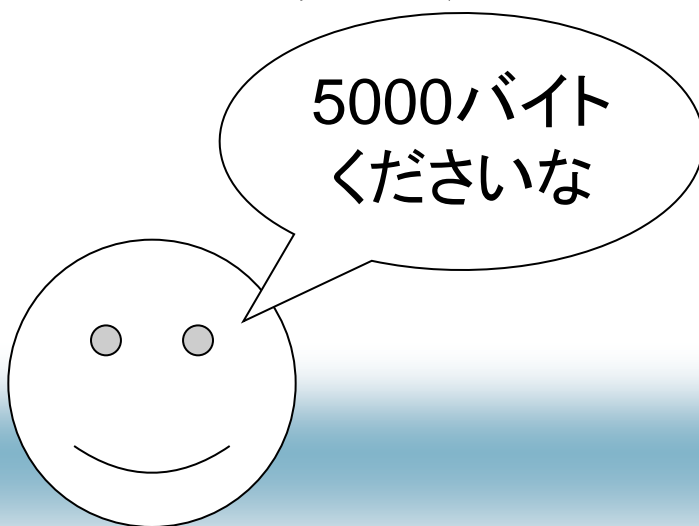
メモリー—思い出そう

- ◆ プログラムの実行のために、ありとあらゆるものがメモリーに格納されなくてはならなかったことを
 - グローバル変数, 配列
 - 局所変数・配列(スタック)
 - 実行中に確保される領域(malloc, new)
 - プログラムのコード



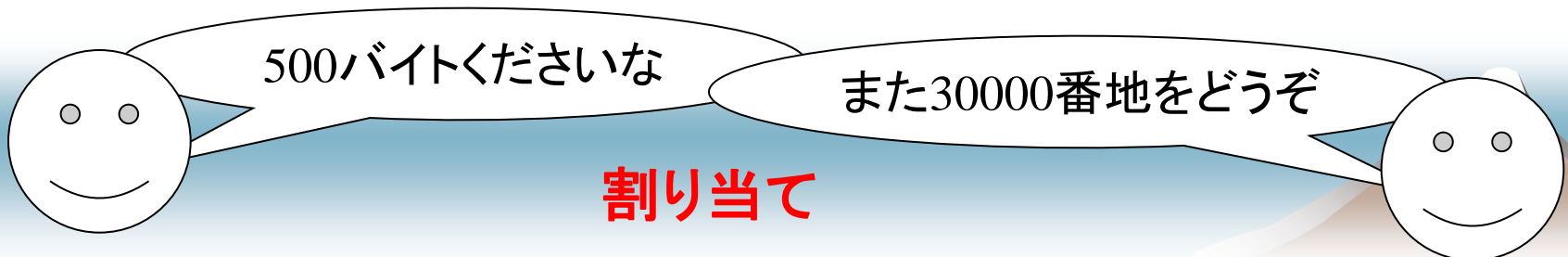
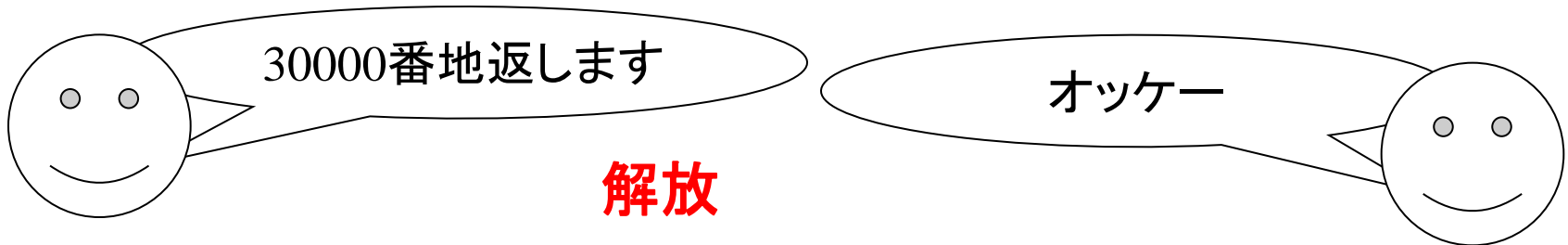
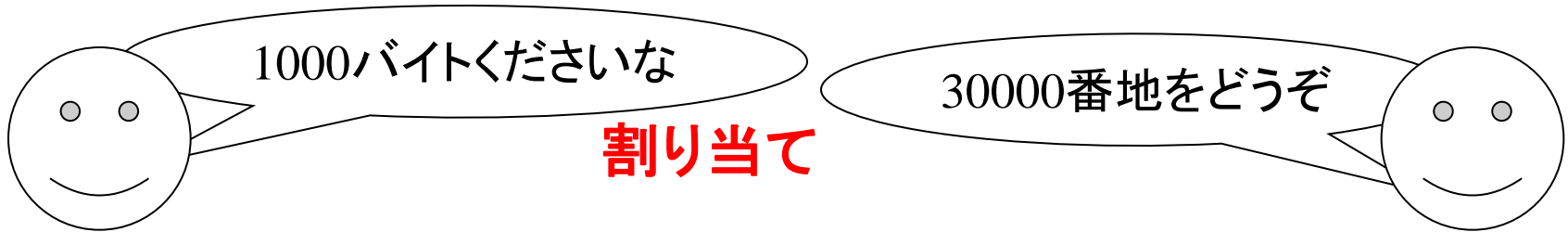
メモリの「管理」とは

- ◆「誰が」、メモリの「どの部分を」、「今」、使ってよいかを記憶しておき、
- ◆「メモリ割り当て要求」にこたえることができるようにすること



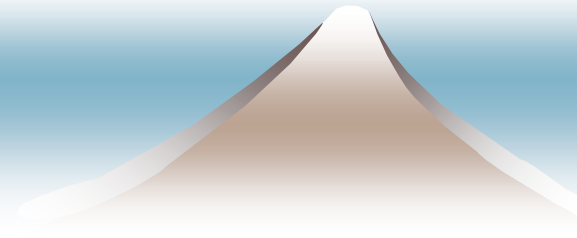
あらゆるメモリ管理に共通の概念

◆ 割り当て(allocation)と解放(deallocation)



OSのメモリ管理API

- ◆ Unix : brk, sbrk, mmap, etc.
- ◆ Win32 : VirtualAlloc, VirtualFree, MapViewOfFile, etc.
- ◆ 詳しくは後述する

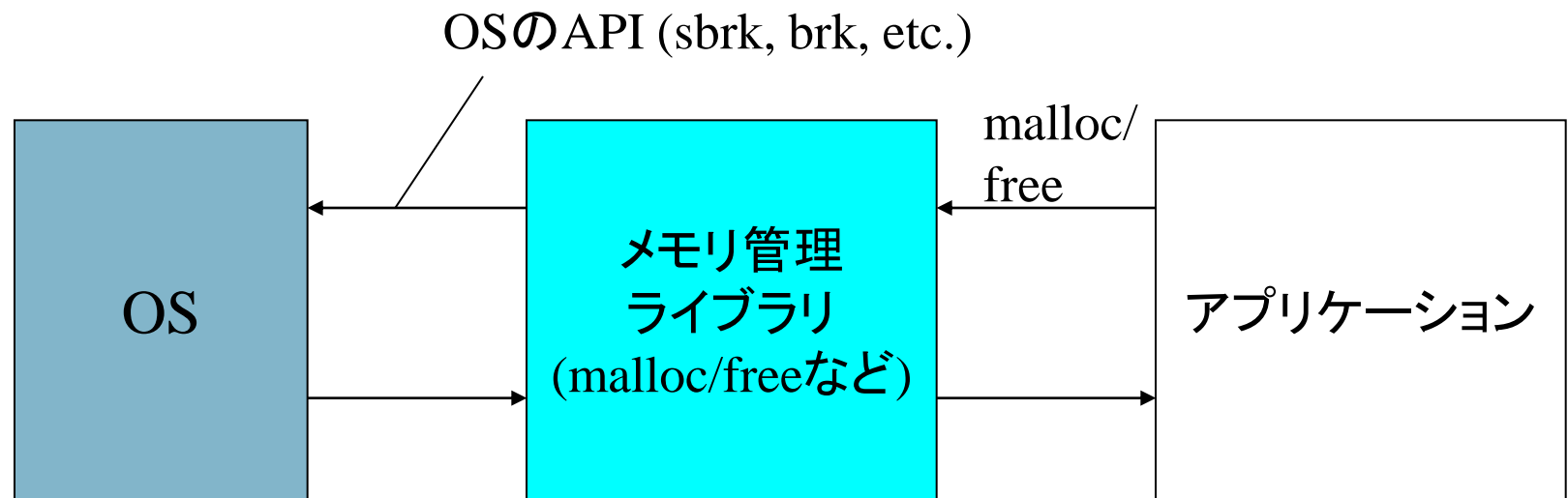


普段良く使っているメモリ割り当てプリミティブ・APIの実例

- ◆ C
 - グローバル変数, スタック, malloc/free, strdup, etc.
- ◆ C++
 - グローバル変数, スタック, new/delete, STLの諸操作, ...
- ◆ Java, C#
 - new, Stringの連結などの諸操作
 - Garbage Collection
- ◆ Python
 - リスト, 辞書, オブジェクト生成, 文字列連結などの諸操作
 - Garbage collection
- ◆ Perl, シェルスクリプト, Visual Basic, ...

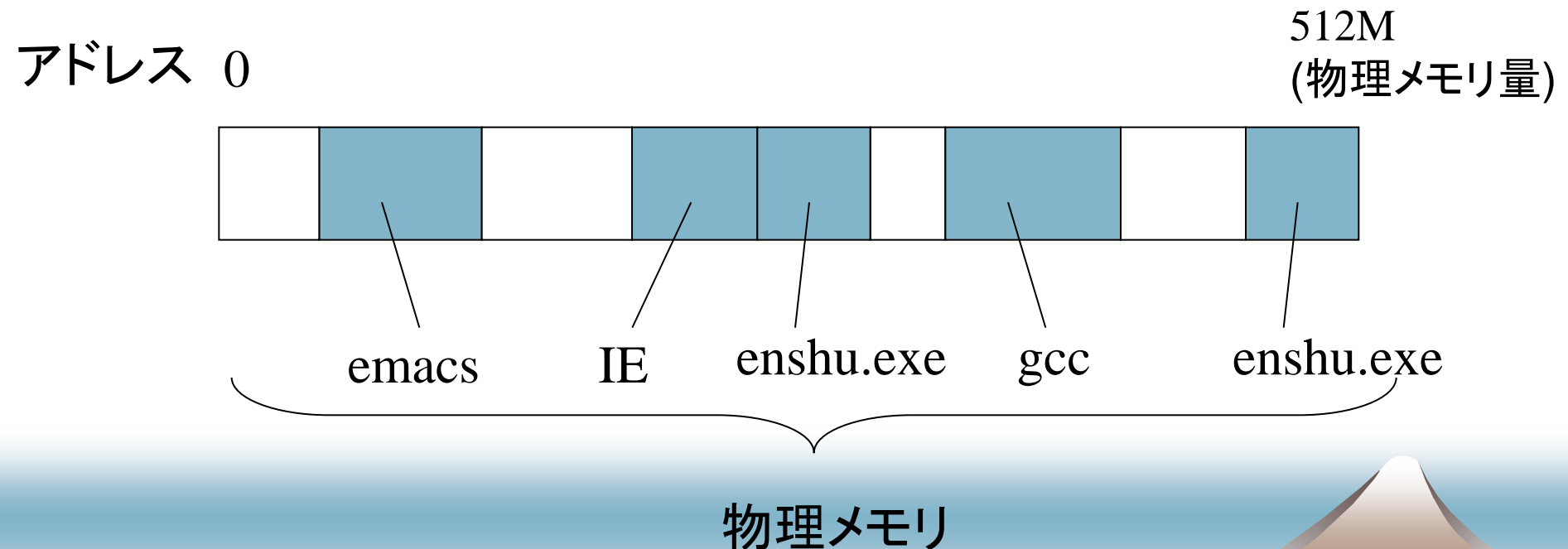
注: OSのAPIとプログラミング言語のAPIの関係

- ◆ malloc/freeなどはOSとアプリケーションプログラムの仲介屋(問屋・小売店・客)



そもそもメモリ管理は難しいか？

- ◆ 空き領域をきちんと記録しておいて，メモリ割り当て・解放要求に答えればよい？



OSがない状態での安直なメモリ 管理—問題点

◆ 危険な相互作用

- 他のプロセスが利用しているメモリを, 他のプロセスが読み書きできてしまう
- 割り当てられていないメモリでも読み書きできてしまう

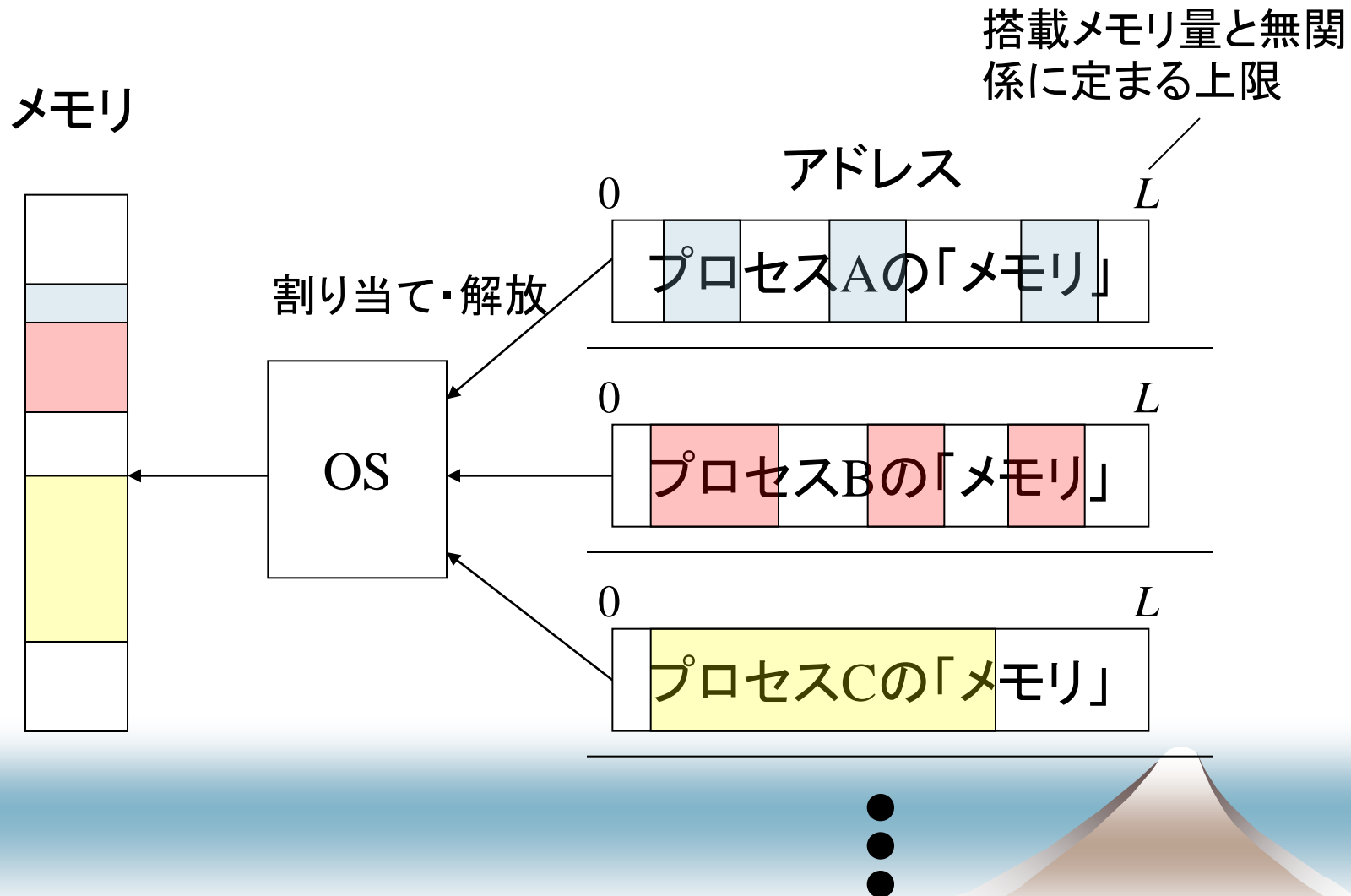
◆ メモリ量の制限

- 合計の「割り当て中」メモリ量 \leq 物理メモリ量

◆ アドレスの被再現性

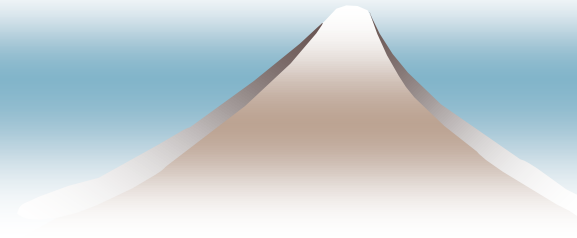
- 並行して実行しているプロセスの有無などで, 割り当てられるアドレスが異なる

OSが提供すべき「強い」メモリ管理



仮想記憶(Virtual Memory)

- ◆ 今時のOSが必ず提供する重要機構
 - あたかも
 - 「各プロセスが」
 - 「他のプロセスと分離された」
 - 「物理メモリ量に依存しない量の(たくさんの)」
- メモリを持っているかのような錯覚を与える



仮想記憶

◆ ~~危険な相互作用~~

- 各プロセスのメモリが「分離」している
- 割り当てられていないメモリへのアクセスを防ぐ

◆ ~~メモリ量の制限~~

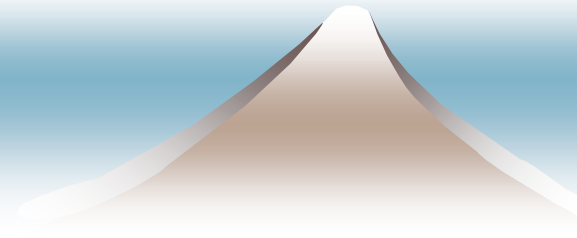
- 割り当て可能なメモリ量が物理メモリ量に依存しない(もちろん無制限ではない)

◆ ~~アドレスの被再現性~~

- 割り当てられるアドレスは他のプロセスの有無によらない

以降の話

- ◆ 以下ではこれらの機能がどのように実現されているのかを見る
 - ハードウェア(CPU)とOSの組み合わせ
- ◆ 重要用語・概念
- ◆ CPUの機能
- ◆ OSの提供するメモリ管理API (次週)



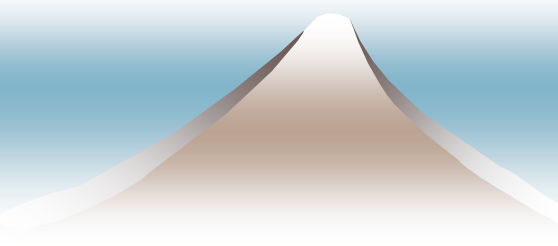
仮想記憶の仕組み概要

◆ アドレス変換

- プログラムが用いるアドレス(論理アドレス)と, メモリハードウェアが用いるアドレス(物理アドレス)は同一ではない

◆ ページング

- すべての「割り当て済み」領域に, 常に物理的なメモリ領域が割り当てられているわけではない
- いったん確保された物理メモリも他で必要になったらディスクに追い出される



重要概念

—論理アドレスと物理アドレス

◆ 論理アドレス

- プログラムが理解(メモリアクセス時に指定)するアドレス

◆ 物理アドレス

- メモリハードウェアが理解するアドレス



論理アドレス

- ◆ プログラムは論理アドレスを用いてメモリをアクセスする

- `main() { p = 10000; printf(“%d¥n”, (*p)); }`

- ◆ 複数の論理アドレス空間が存在する

- プロセス \leftrightarrow 論理アドレス空間

論理アドレス10000
をアクセス

論理アドレス空間

- ◆ 複数の論理アドレス空間は分離している
 - プロセスAの10000番地とプロセスBの10000番地は「別の場所」
- ◆ 各論理アドレス空間の大きさは、物理メモリ量に**よらない**(ポインタのbit数とOSの設計で決まる)

0 $2^{32} - 1$

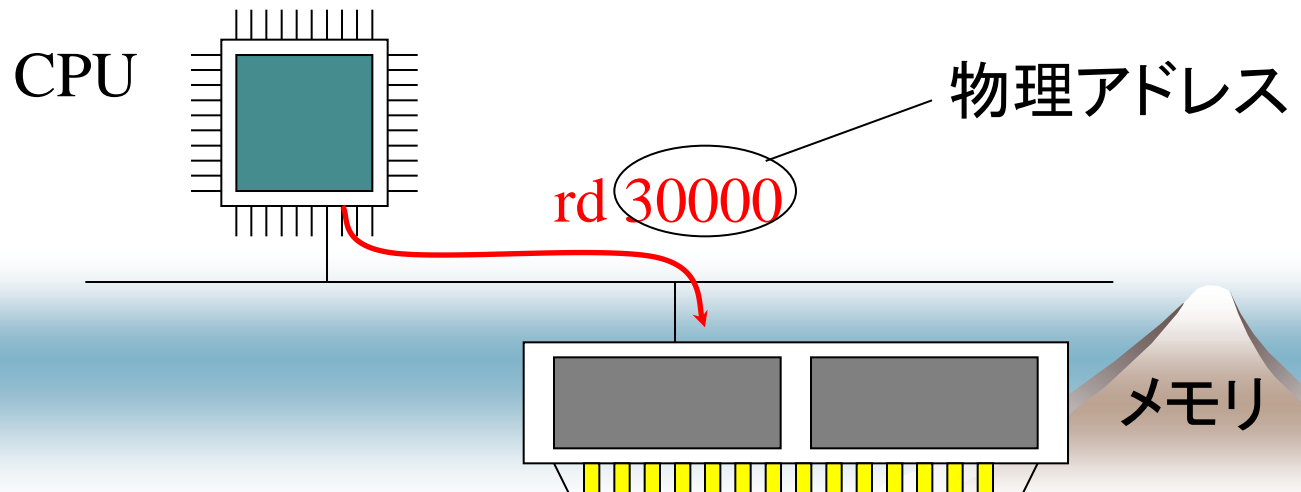
論理アドレス空間 A

論理アドレス空間 B

論理アドレス空間 C

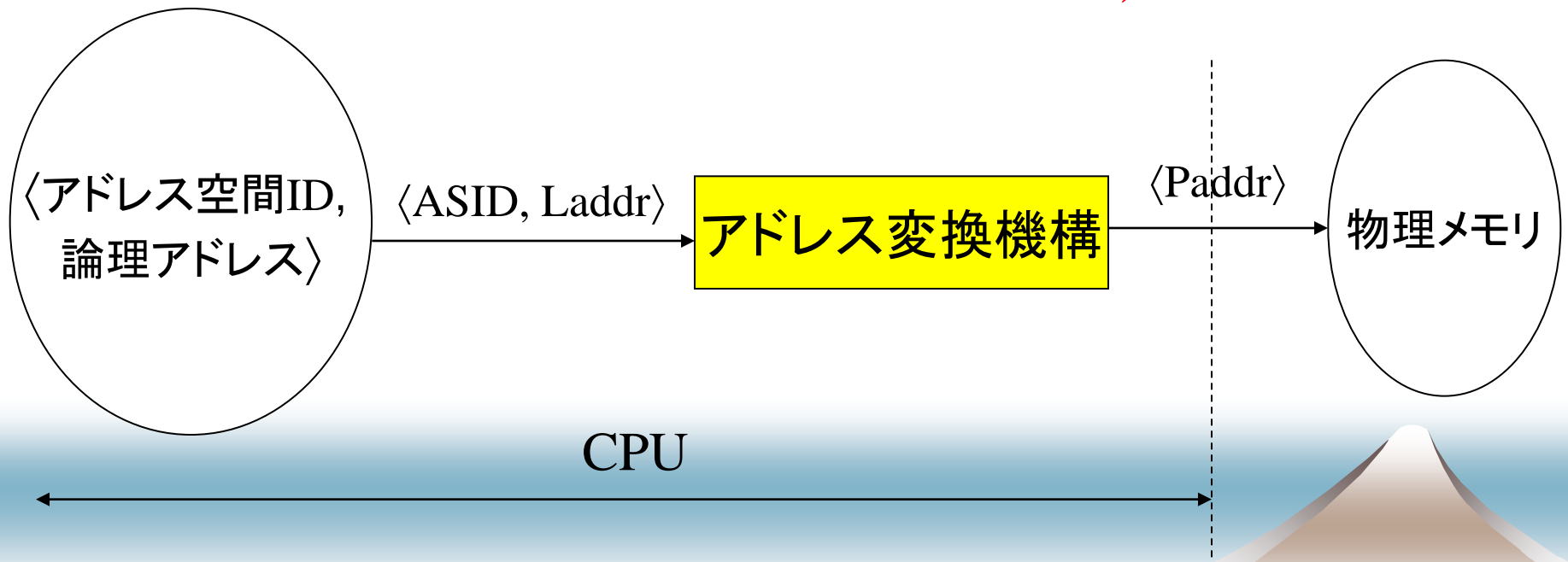
物理アドレス

- ◆ メモリハードウェアが理解するアドレスは**物理アドレス**
- ◆ 可能な物理アドレスは0 ... 搭載メモリ量 - 1
- ◆ (当然)各アドレスは計算機にひとつだけ存在する



これらすべての帰結(1)

- ◆ 〈アドレス空間, 論理アドレス〉の組を, 対応する「物理アドレス」に変換する仕組み(アドレス変換; Address Translation)が必要

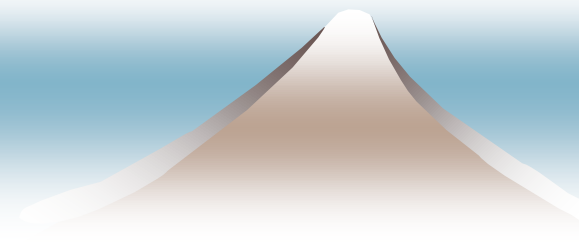


これらすべての帰結(2)

- ◆「割り当て中のメモリ」すべてに物理メモリを確保することは不可能
- ◆ ディスクを補助記憶域としてうまくやりくり(ページング; paging)する必要がある

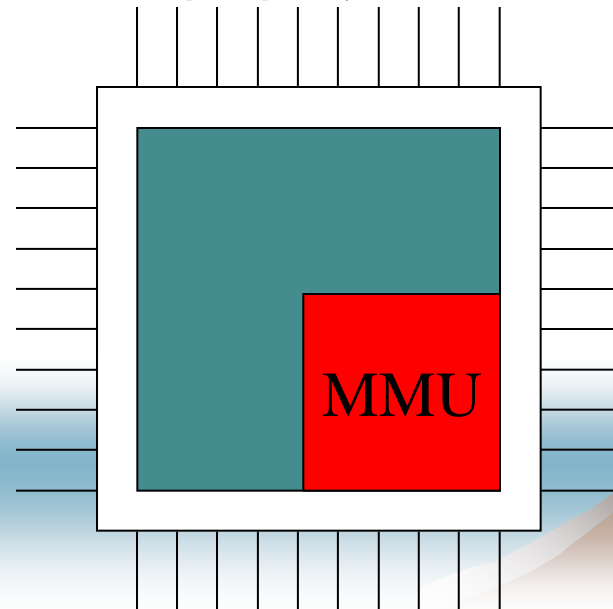


メモリ管理のためにCPUが備える機構



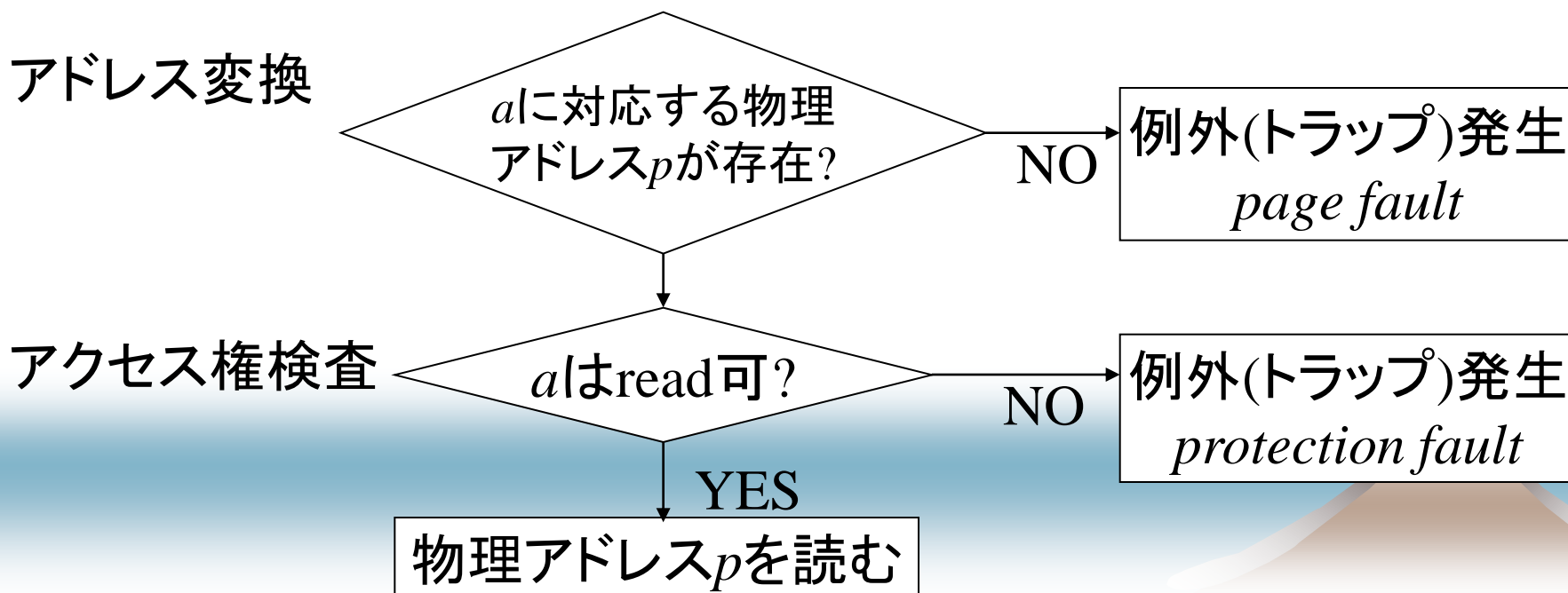
メモリ管理ユニット(Memory Management Unit; MMU)

- ◆ 役割: すべてのメモリアクセス命令実行に介在して, 以下を行う
 - アクセス権の検査
 - アドレスが物理メモリ上に存在するかの検査
 - アドレス変換
 - その他



メモリアクセス命令時のCPUの動作(概要)

- ◆ アクセス権検査; アドレス変換; アクセス
- ◆ 例: $\text{load } [r_1], r_2$ (storeもほぼ同じ)
 - レジスタ r_1 の中身がアドレス a だったとする



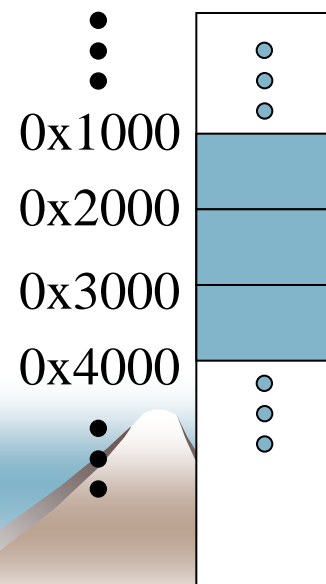
ページ

- ◆ CPU (MMU)は各アドレスに対応する
 - 保護属性(read/write可 etc.)
 - 物理アドレスなどを記憶する必要がある
 - ◆ 保護属性, アドレス変換をすべてのアドレスに個別に保持するのは不可能
- ⇒「ページ」(対応付けの単位)の概念

ページ

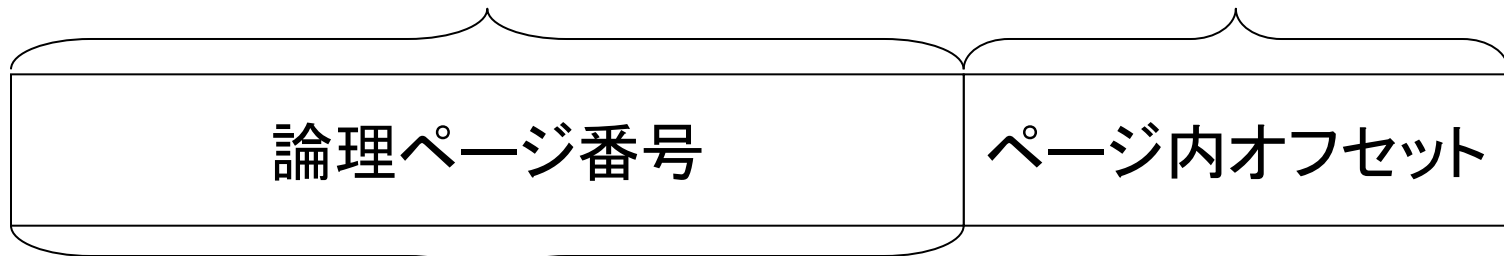
- ◆ ハードウェアが，保護属性，アドレス変換，などを保持する単位
- ◆ 典型的には4096バイト，8192バイトなどの連続したアドレスの範囲

~~〈ASID, 論理アドレス〉 → 属性や物理アドレス~~
〈ASID, 論理ページ番号〉 → 属性や物理ページ番号



アドレスとページ

- ◆ 例: 32 bit論理アドレス. 4096バイトページ
20 bit 12 bit



- ◆ 〈アドレス空間ID, 論理ページ番号〉をキーとして対応付けを保持する

物理ページ番号	保護属性	その他の属性	0
Mapping不在 (対応する物理ページなし)			1

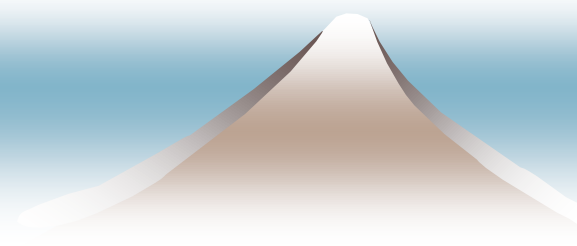
重要な属性

◆ 保護属性

- 読み出し可(readable)
- 書き込み可(writable)
- ユーザモードでアクセス可

◆ その他の属性

- 参照(reference) bit (read時にset)
- 汚れ(dirty) bit (write時にset)



Mapping不在

- ◆ 対応する物理ページが現在, 存在しないことを示す
- ◆ 二つの場合がある
 - 論理的に割り当てられていない
 - そこをアクセスするのは実際, 違反
 - 論理的には割り当て中. だが,
 - OSが「たった今は」物理メモリを割り当てていない
- ◆ CPUは両者を区別しない(OSが区別する)

対応付けの実際

◆ ページテーブル

- メモリ上に置かれた表
- 〈論理アドレス空間, 論理ページ番号〉をキーとして〈物理ページ番号, 属性〉を値とする表

◆ TLB (Translation Lookaside Buffer)

- CPU内にある, 通常100エントリ程度の連想記憶(通常fully associative)
- 役割: ページテーブルのキャッシュ

工夫のない対応付けに必要な ページテーブルの大きさ

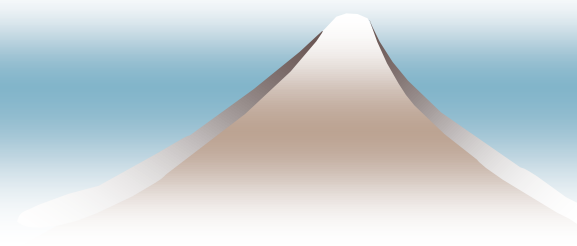
◆ 仮定

- 32 bit論理アドレス. 20 bitページ番号
- 1エントリ32 bit (物理ページ番号+属性)


◆ $2^{20} \times 4 \times n = 4n \text{ MB}$

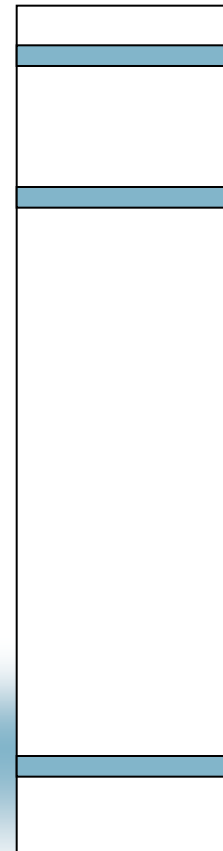
- n : 論理アドレス空間の数 \approx 同時に存在するプロセス数

◆ まだ大きすぎる(もう一工夫)

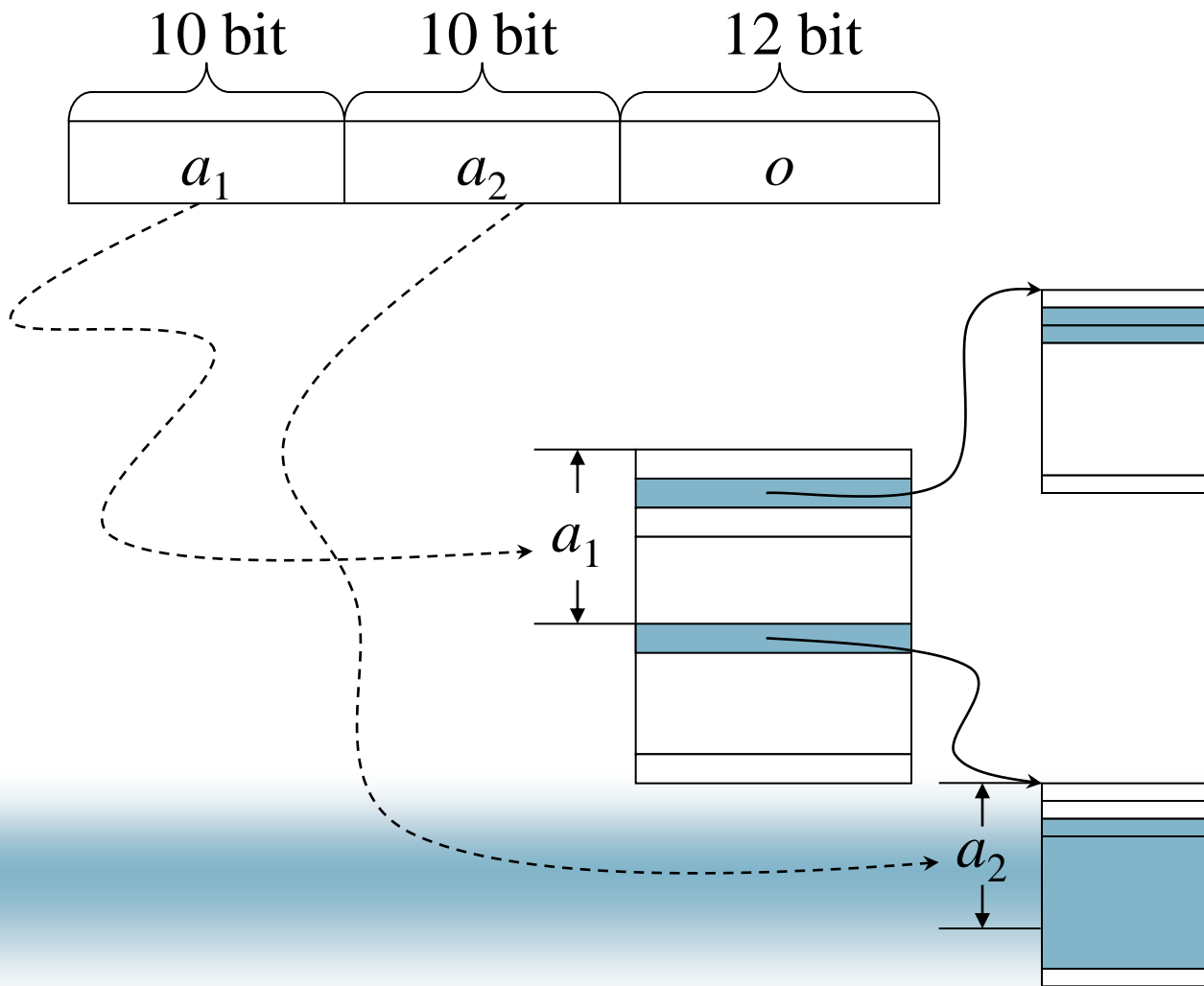


ページテーブルの構造

- ◆「ほとんど空」の論理アドレス空間を小さく表現する
 - ◆多段のページテーブル
- 

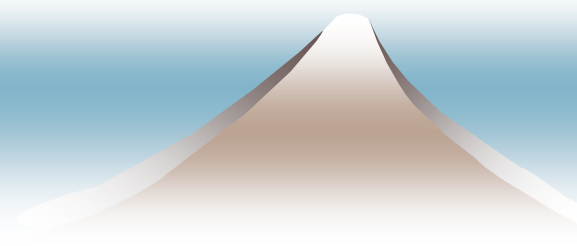


多段のページテーブル



64bitアドレス?

- ◆ Madhusudhan Tallurl, Mark D. Hill, Yousef A. Khalidi. *A New Page Table for 64-bit Address Spaces*. SOSPP 1995



メモリアクセス時のCPUの動作: まとめ (read)

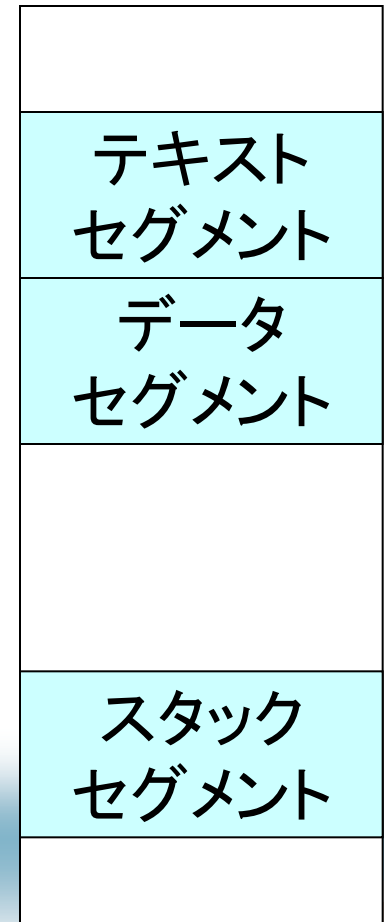
```
◆ read(a) {  
    p,attr = lookup_TLB(a) ;  
    if (!found) p,attr = lookup_page_table(a) ;  
    if (!found) raise page fault;  
    if (!attr.readable) raise protection fault;  
    if (!attr.user && CPU_mode == user)  
        raise protection fault;  
    read p; /* in cache or memory */  
    set reference bit for a;  
}
```

Writeの場合

```
◆ write(a, v) {  
    p, attr = lookup_TLB(a) ;  
    if (!found) p, attr = lookup_page_table(a) ;  
    if (!found) raise page fault ;  
    if (!attr.writable) raise protection fault ;  
    if (!attr.user && CPU_mode == user)  
        raise protection fault ;  
    write v to p ; /* in cache or memory */  
    set reference/dirty bit for a ;  
}
```

余談：セグメンテーション ページング以前の仮想記憶

- ◆ セグメント：
 - (ページよりも大きな)連続したアドレスの範囲
 - 必要に応じて伸ばせる
- ◆ 各論理アドレス空間で割り当て中のメモリは, 少数(数個)のセグメントとする
- ◆ 必要に応じてセグメントを丸ごと移動, ディスクに退避



そういえばセグメンテーション フォルトって何だっけ

- ◆ セグメントを越えたアクセス
- ◆ 今日的には, protection fault, access violation

