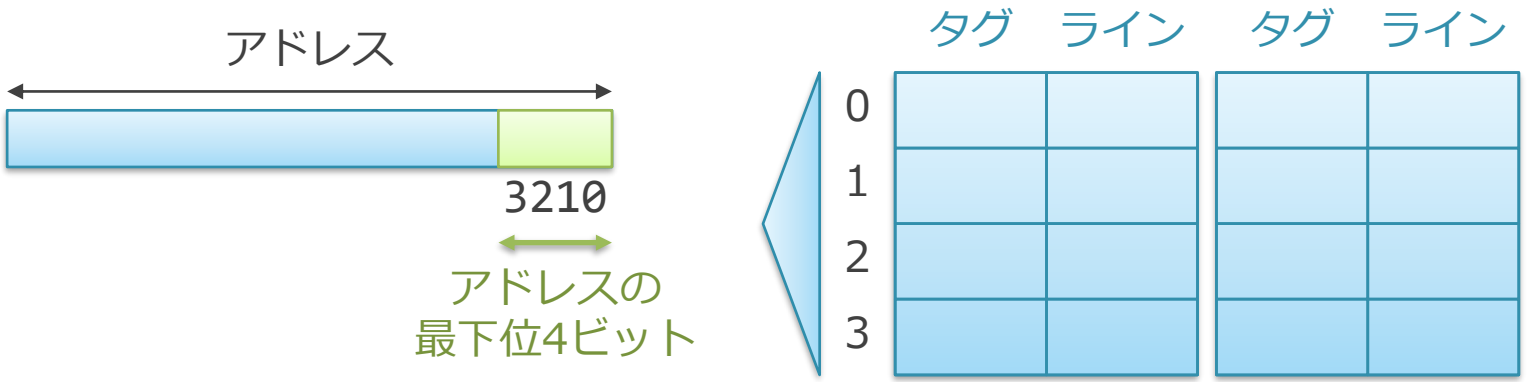
コンピュータ アーキテクチャ Ⅰ　第１１回

塩谷 亮太 (shioya@ci.i.u-tokyo.ac.jp)
東京大学大学院情報理工学系研究科 創造情報学専攻

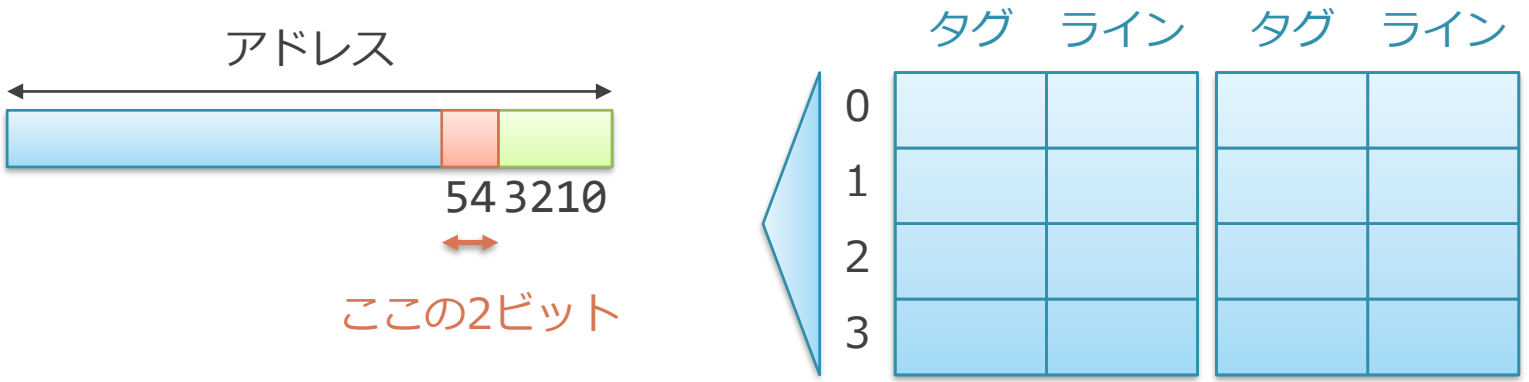
課題の解説

アドレスとラインの対応



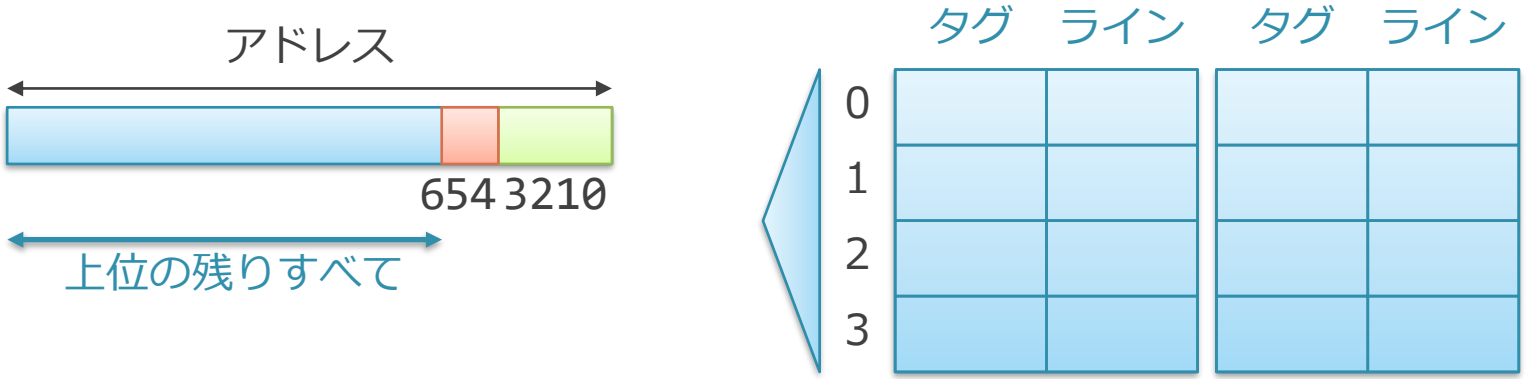
- アドレスは1バイト単位でメモリの位置を表すものとする
- 最下位ビット０～３（計４ビット）
  - 最下位部分がライン内の位置に対応
    - 空間局所性を利用するために連続した１６バイトが１ラインに
  - 4ビットなのは，ラインサイズが16バイトだから
    - 2 ^ 4 = 16
    - （ラインサイズは必ず２の累乗になる

アドレスとセットの対応



- ライン部分の上位にあるビット４～5 （計2ビット）
  - この部分を使って，どのセットにアクセスするか決める
  - ２ビットなのは，セット数が４だから
    - ２＾２＝４
  - セット数も必ず２の累乗になる
- アドレスのこの部分はなるべくばらけた方がよい
  - 同じセットにアクセスがいかず，競合がおきにくくなる

アドレスとタグの対応



- 残りの上位のビットがタグとなる
- タグにはセット（赤）やライン（緑）の部分は入れないでよい
  - あるセットにアクセスするアドレスは，赤部分は常に一定だから
    - セット 1 にアクセスする場合，赤部分は絶対 01
  - 緑部分はラインの中の位置を表すので，関係ない

課題 １０

- アドレスの幅が 16 bit，ラインサイズ8B，４エントリのキャッシュについて考える

- 連想度を以下の様に変えた場合に，
  - 1（ダイレクトマップ）
  - 2
  - 4（フルアソシアティブ）

- 以下のようなアドレスによる 1B のアクセスがあった場合を考える

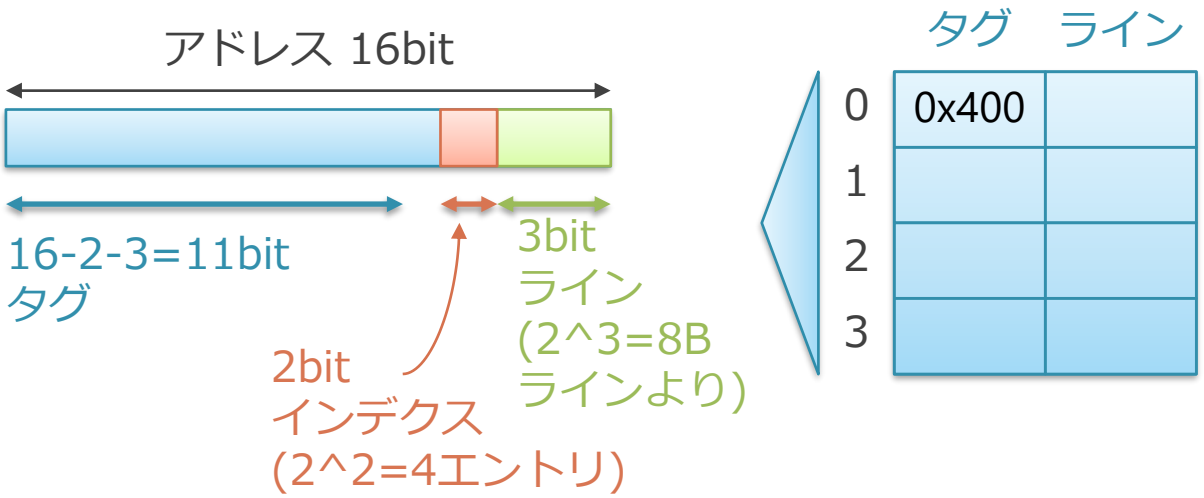
1. `0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003`
2. `0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000`
3. `0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007`

課題 １０

- (1) 上記それぞれの場合で，アクセスが全て終わった後のキャッシュの状態（タグの中身）を示せ
  - ４エントリのタグにそれぞれ何が残っているかを，
    連想度3パターン×アクセス系列3パタン= 9 パターン分答える
- (2) 上記それぞれの場合のヒット率を計算せよ
- (3) 各アクセスにおけるヒット時に，それが空間的局所性と時間的局所性のいずれによるのかを分類して答えよ
- 多少多いかもですが，
  - 途中までしか出来なくても良いです
  - 試験までには１回解いておくと良いです
  - 実は (1) がちゃんとできれば (2) と (3) はおまけみたいなものです

課題 １０

- 考え方の方針：
  1. アドレスの系列を全部２進数のビット列に直す
  2. ２進数のビット列を下の図の割り当てごとに分類する
     - 上から順に，タグ部，インデックス部，ライン部の３つに分類される
     - このインデックス部により，キャッシュのどこにアクセスされるかが決まる
       - =テーブルの何行目のエントリにいくかが決まる
  3. タグ部を対応するエントリに上書きしていく
  4. 対応するエントリに同じタグがある=hit，ない=miss
     - 全く同じアドレスにアクセスした事がある hit=時間局所性
     - 全く同じアドレスにはアクセスしていないが hit=空間局所性

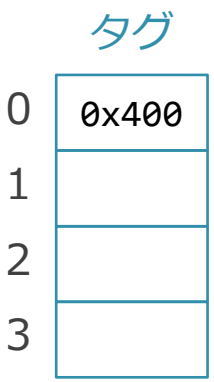


課題 １０ ダイレクトマップの場合 の系列１

1. 0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003

タグ（黒）とインデックス（オレンジ）を決定するビットが全部同じ = 同じライン

1. 0x8000 = 0b1000 0000 0000 0000 miss
2. 0x8001 = 0b1000 0000 0000 0001 hit 空間的局所性
3. 0x8002 = 0b1000 0000 0000 0010 hit 空間的局所性
4. 0x8003 = 0b1000 0000 0000 0011 hit 空間的局所性
5. 0x8000 = 0b1000 0000 0000 0000 hit 時間的局所性 or 空間的局所性
6. 0x8001 = 0b1000 0000 0000 0001 hit 時間的局所性 or 空間的局所性
7. 0x8002 = 0b1000 0000 0000 0010 hit 時間的局所性 or 空間的局所性
8. 0x8003 = 0b1000 0000 0000 0011 hit 時間的局所性 or 空間的局所性
   0b100 0000 0000=0x400



- (1) 最初のアクセスで左のようにタグの 0x400 が書き込まれた後はずっとそのまま
- (2) ヒット率：7/8=0.875
- (3) 上の通り

課題 １０ ダイレクトマップの場合 の系列２

1. 0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000
   インデックス（オレンジ）を決定するビットが全部同じ = 同じエントリへ上書き
   最後にアクセスされた 0xb000 の分が残る
   1. 0x8000 = 0b1000 0000 0000 0000 miss
   2. 0x9000 = 0b1001 0000 0000 0000 miss
   3. 0xA000 = 0b1010 0000 0000 0000 miss
   4. 0xB000 = 0b1011 0000 0000 0000 miss
   5. 0x8000 = 0b1000 0000 0000 0000 miss
   6. 0x9000 = 0b1001 0000 0000 0000 miss
   7. 0xA000 = 0b1010 0000 0000 0000 miss
   8. 0xB000 = 0b1011 0000 0000 0000 miss
      0b101 1000 0000=0x580

タグ

0	0x580
1	
2	
3	

- (1) 最初のアクセスで左のようにタグの 0x580 が書き込まれた後はずっとそのまま
- (2) ヒット率：0
- (3) 全部ミスなので局所性がない

課題 １０ ダイレクトマップの場合 の系列３

1. 0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007

1. 0x8000 = 0b1000 0000 0000 0000 miss
2. 0x9001 = 0b1001 0000 0000 0001 miss
3. 0x8002 = 0b1000 0000 0000 0010 miss
4. 0x9003 = 0b1001 0000 0000 0011 miss
5. 0x9004 = 0b1001 0000 0000 0100 hit　空間局所性
6. 0xA005 = 0b1010 0000 0000 0101 miss
7. 0x9006 = 0b1001 0000 0000 0110 miss
8. 0x8007 = 0b1000 0000 0000 0111 miss
   0b100 0000 0000=0x400

タグ

0	0x400
1	
2	
3	

- (1) 0番エントリのタグが 0x400,0x480,x400,0x480,0x500,x480,0x400 の順で更新される
- (2) ヒット率：1/8=0.125
- (3) 上の通り

課題 １０
2-way セットアソシアティブの場合 の系列１

1. 0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003
   インデックスを決定する部分が1bitだけになる

   1. 0x8000 = 0b1000 0000 0000 0000 miss
   2. 0x8001 = 0b1000 0000 0000 0001 hit 空間的局所性
   3. 0x8002 = 0b1000 0000 0000 0010 hit 空間的局所性
   4. 0x8003 = 0b1000 0000 0000 0011 hit 空間的局所性
   5. 0x8000 = 0b1000 0000 0000 0000 hit 時間的局所性 or 空間的局所性
   6. 0x8001 = 0b1000 0000 0000 0001 hit 時間的局所性 or 空間的局所性
   7. 0x8002 = 0b1000 0000 0000 0010 hit 時間的局所性 or 空間的局所性
   8. 0x8003 = 0b1000 0000 0000 0011 hit 時間的局所性 or 空間的局所性
      0b1000 0000 0000=0x800

タグ

0	0x800	
1		

- (1) 最初のアクセスで左のようにタグの 0x800 が書き込まれた後はずっとそのまま
- (2) ヒット率：7/8=0.875
- (3) 上の通り

課題 １０
2-way セットアソシアティブの場合 の系列２

1. 0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000

   インデックスを決定するビットが全部同じ = 同じエントリに入る
   同一セット内の２ラインでは長時間アクセスされていない方が上書きされる

   最後にアクセスされた 0xb000 の分が残る

   1. 0x8000 = 0b1000 0000 0000 0000 miss　(0b1000 0000 0000=0x800
   2. 0x9000 = 0b1001 0000 0000 0000 miss　(0b1001 0000 0000=0x900
   3. 0xA000 = 0b1010 0000 0000 0000 miss　(0b1010 0000 0000=0xA00
   4. 0xB000 = 0b1011 0000 0000 0000 miss　(0b1011 0000 0000=0xB00
   5. 0x8000 = 0b1000 0000 0000 0000 miss　(0b1000 0000 0000=0x800
   6. 0x9000 = 0b1001 0000 0000 0000 miss　(0b1001 0000 0000=0x900
   7. 0xA000 = 0b1010 0000 0000 0000 miss　(0b1010 0000 0000=0xA00
   8. 0xB000 = 0b1011 0000 0000 0000 miss　(0b1011 0000 0000=0xB00

タグ

0	0xA00	0xB00
1		

- (1) セット０に
  0x800,0x900,0xA00,0xB00,0x800,0x900,0xA00,0xB00
  が書き込まれ，左のようになる
- (2) ヒット率：0
- (3) 全部ミスなので局所性がない

課題 １０
2-way セットアソシアティブの場合 の系列３

1. 0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007

1. 0x8000 = 0b1000 0000 0000 0000 miss (0b1000 0000 0000=0x800
2. 0x9001 = 0b1001 0000 0000 0001 miss (0b1001 0000 0000=0x900
3. 0x8002 = 0b1000 0000 0000 0010 hit (0b1000 0000 0000=0x800 空間
4. 0x9003 = 0b1001 0000 0000 0011 hit (0b1001 0000 0000=0x900 空間
5. 0x9004 = 0b1001 0000 0000 0100 hit (0b1001 0000 0000=0x900 空間
6. 0xA005 = 0b1010 0000 0000 0101 miss (0b1010 0000 0000=0xA00
7. 0x9006 = 0b1001 0000 0000 0110 hit (0b1001 0000 0000=0x900 空間
8. 0x8007 = 0b1000 0000 0000 0111 miss (0b1000 0000 0000=0x800
   0b100 0000 0000=0x400

タグ

0	0x800	0x900
1		

- (1) 0番エントリのタグが更新される
  その時一番アクセスされていないものが上書きされる
- (2) ヒット率：4/8=0.5
- (3) 上の通り

課題 １０
フルアソシアティブの場合 の系列１

1. 0x8000, 0x8001, 0x8002, 0x8003, 0x8000, 0x8001, 0x8002, 0x8003
   インデックスを決定する部分がなくなる

   1. 0x8000 = 0b1000 0000 0000 0000 miss
   2. 0x8001 = 0b1000 0000 0000 0001 hit 空間的局所性
   3. 0x8002 = 0b1000 0000 0000 0010 hit 空間的局所性
   4. 0x8003 = 0b1000 0000 0000 0011 hit 空間的局所性
   5. 0x8000 = 0b1000 0000 0000 0000 hit 時間的局所性 or 空間的局所性
   6. 0x8001 = 0b1000 0000 0000 0001 hit 時間的局所性 or 空間的局所性
   7. 0x8002 = 0b1000 0000 0000 0010 hit 時間的局所性 or 空間的局所性
   8. 0x8003 = 0b1000 0000 0000 0011 hit 時間的局所性 or 空間的局所性
      0b1 0000 0000 0000=0x1000

タグ

0x1000			

- (1) 最初のアクセスで左のようにタグの 0x1000 が書き込まれた後はずっとそのまま
- (2) ヒット率：7/8=0.875
- (3) 上の通り

課題 １０
フルアソシアティブの場合 の系列２

1. 0x8000, 0x9000, 0xA000, 0xB000, 0x8000, 0x9000, 0xA000, 0xB000
   インデックスを決定するビットが全部同じ = 同じエントリに入る
   最後にアクセスされた 0xb000 の分が残る
   1. 0x8000 = 0b1000 0000 0000 0000 miss　(0b1 0000 0000 0000=0x1000
   2. 0x9000 = 0b1001 0000 0000 0000 miss　(0(b1 0010 0000 0000=0x1200
   3. 0xA000 = 0b1010 0000 0000 0000 miss　(0b1 0100 0000 0000=0x1400
   4. 0xB000 = 0b1011 0000 0000 0000 miss　(0b1 0110 0000 0000=0x1600
   5. 0x8000 = 0b1000 0000 0000 0000 hit　(0b1 0000 0000 0000=0x1000 時間
   6. 0x9000 = 0b1001 0000 0000 0000 hit　(0b1 0010 0000 0000=0x1200 時間
   7. 0xA000 = 0b1010 0000 0000 0000 hit　(0b1 0100 0000 0000=0x1400 時間
   8. 0xB000 = 0b1011 0000 0000 0000 hit　(0b1 0110 0000 0000=0x1600 時間

タグ

0x1000	0x1200	0x1400	0x1600

- (1) 左のようになる
- (2) ヒット率：4/8=0.5
- (3) 上の通り

課題 １０
フルアソシアティブの場合 の系列３

1. 0x8000, 0x9001, 0x8002, 0x9003, 0x9004, 0xA005, 0x9006, 0x8007

    1. 0x8000 = 0b1000 0000 0000 0000 miss 0b1 0000 0000 0000=0x1000
    2. 0x9001 = 0b1001 0000 0000 0001 miss 0b1 0010 0000 0000=0x1200
    3. 0x8002 = 0b1000 0000 0000 0010 hit 0b1 0000 0000 0000=0x1000 空間
    4. 0x9003 = 0b1001 0000 0000 0011 hit 0b1 0010 0000 0000=0x1200 空間
    5. 0x9004 = 0b1001 0000 0000 0100 hit 0b1 0010 0000 0000=0x1200 空間
    6. 0xA005 = 0b1010 0000 0000 0101 miss 0b1 0100 0000 0000=0x1400
    7. 0x9006 = 0b1001 0000 0000 0110 hit 0b1 0010 0000 0000=0x1200 空間
    8. 0x8007 = 0b1000 0000 0000 0111 hit 0b1 0000 0000 0000=0x1000 空間

タグ

0x1000	0x1200	0x1400	

- (1) 左の通り
- (2) ヒット率：5/8=0.625
- (3) 上の通り

ヒット率のまとめ

	系列 1	系列 2	系列 3
ダイレクトマップ	0.875	0	0.125
セットアソシアティブ	0.875	0	0.5
フルアソシアティブ	0.875	0.5	0.625

- 基本的には連想度を上げると（フルアソに近づくと）ヒット率があがる
  - 実は常に上がるわけではなく，逆に下がるパターンもある

仮想メモリと特権モード

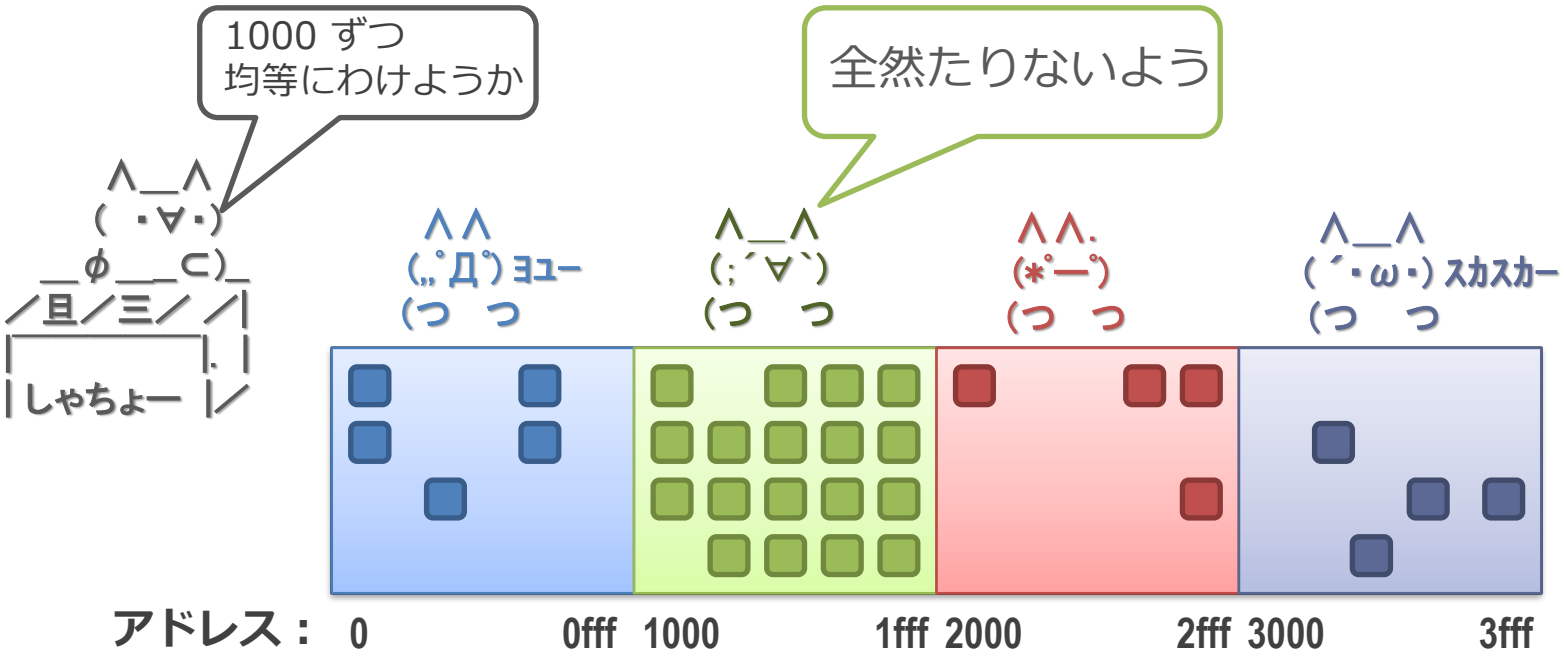
今日の内容

1. **仮想メモリ**
    1. モチベーションと基本
    2. 詳細
2. 特権モード

仮想メモリのモチベーション

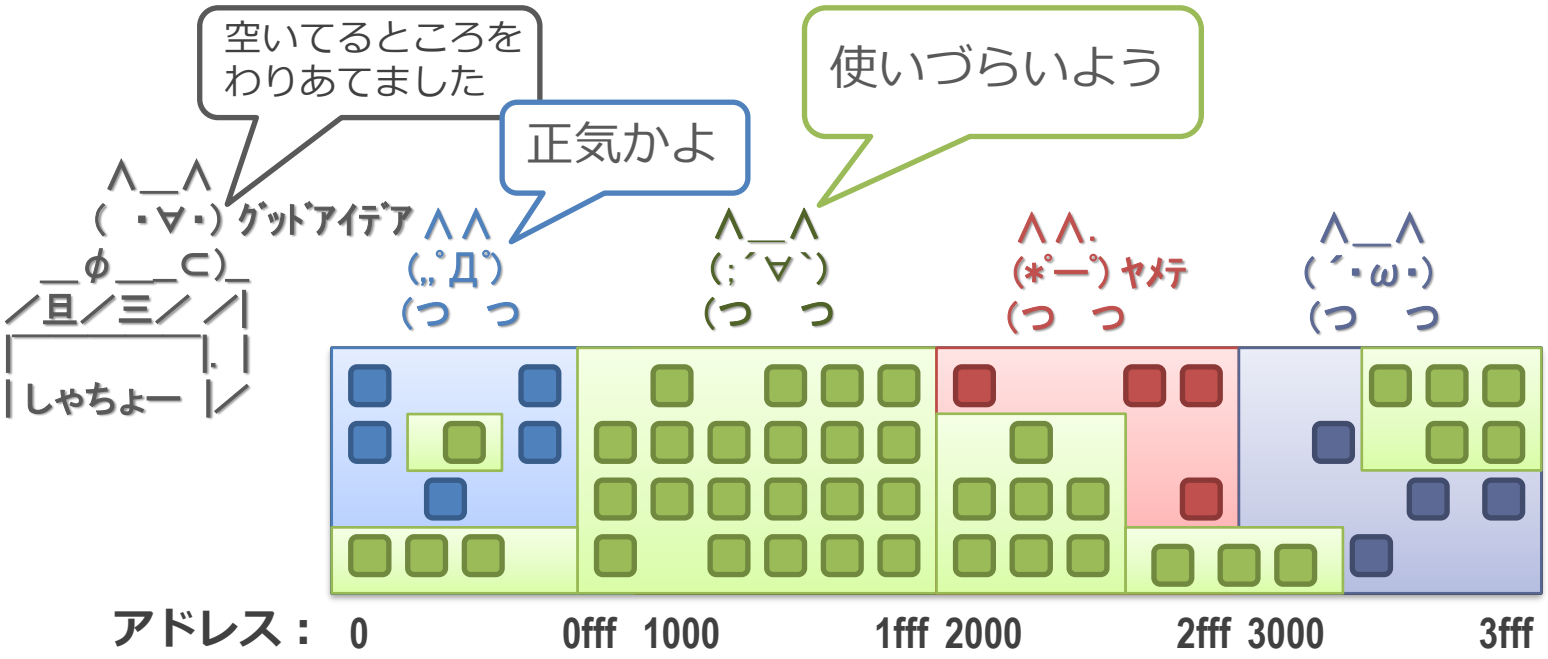
- 前提：複数のプログラムを１つの CPU 上で同時に動かすことを考える
  - 上の図では４つのプログラムが動くとする
  - メモリは複数のプログラムで共有される
- 問題：どうやって共有するか？
  1. どうやって領域の割り当てを行う？
  2. どうやって各人の領域を保護する？

1. どうやって領域の割り当てを行う？



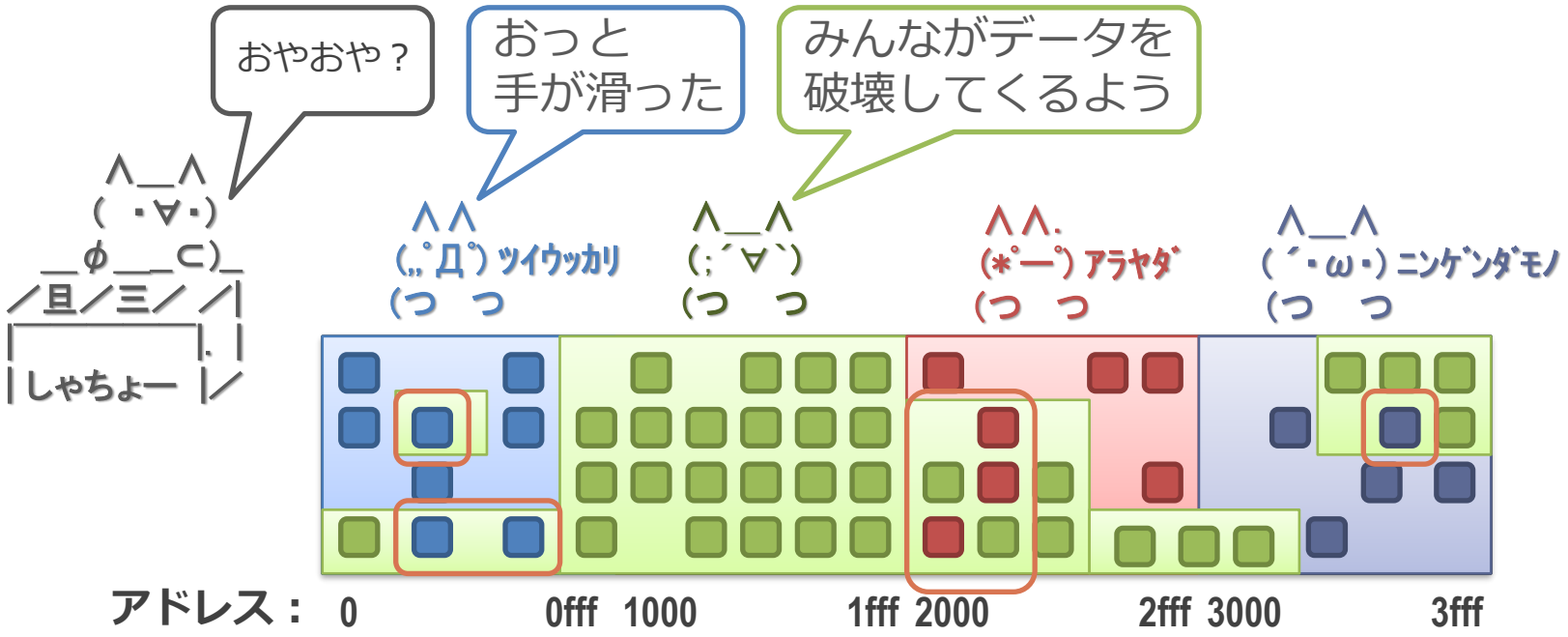
- 単純には均等に分ければ良い
  - しかし，プログラムごとに必要なメモリの量は違うのが普通

1. どうやって領域の割り当てを行う？



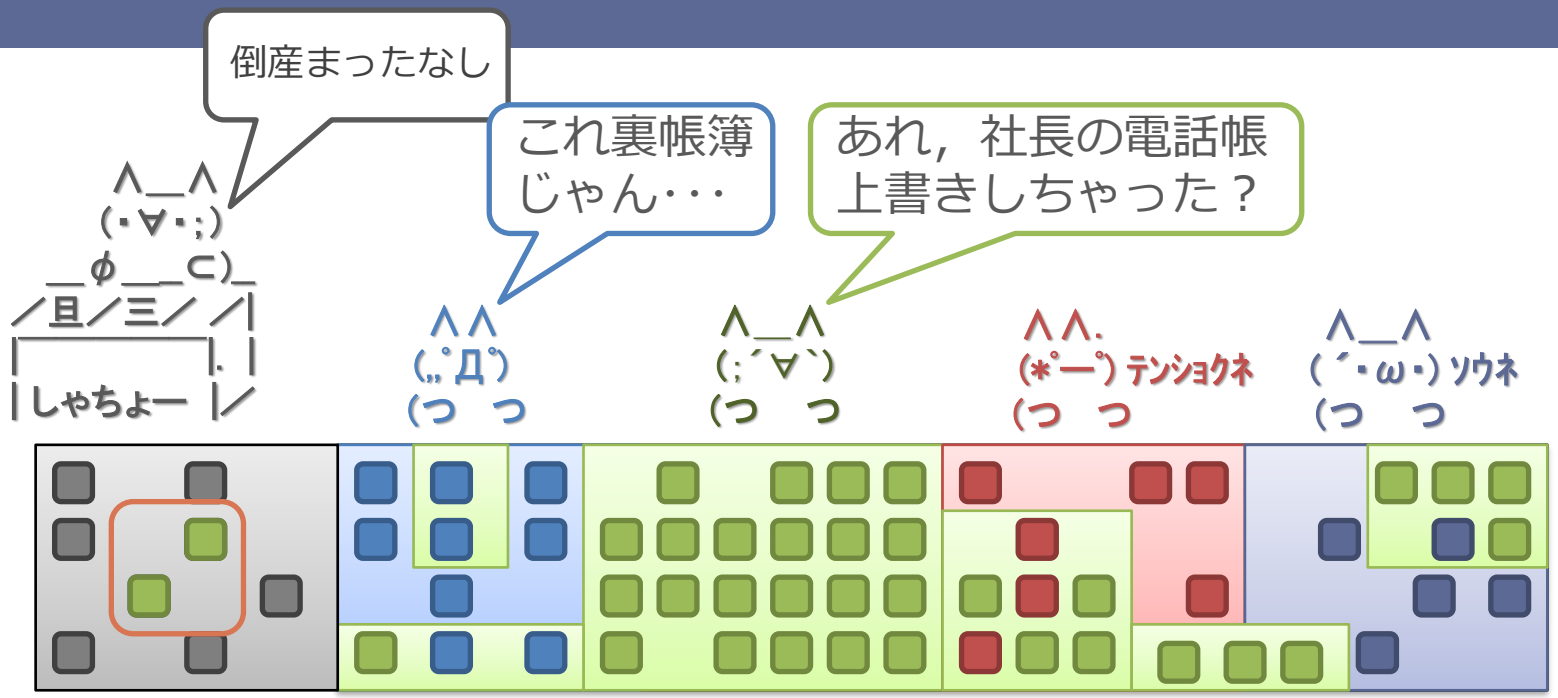
- たくさんメモリを使う人に都度割り当てると，メモリ空間が細切れになってとても使いにくい
  - 青の人のメモリ：0x400-0x7ff, 0xc00-0xfff
  - 緑の人のメモリ：0x000-0x3ff, 0x800-0xbff, 0x1000-0x2000 …

2. どうやって各人の領域を保護する？



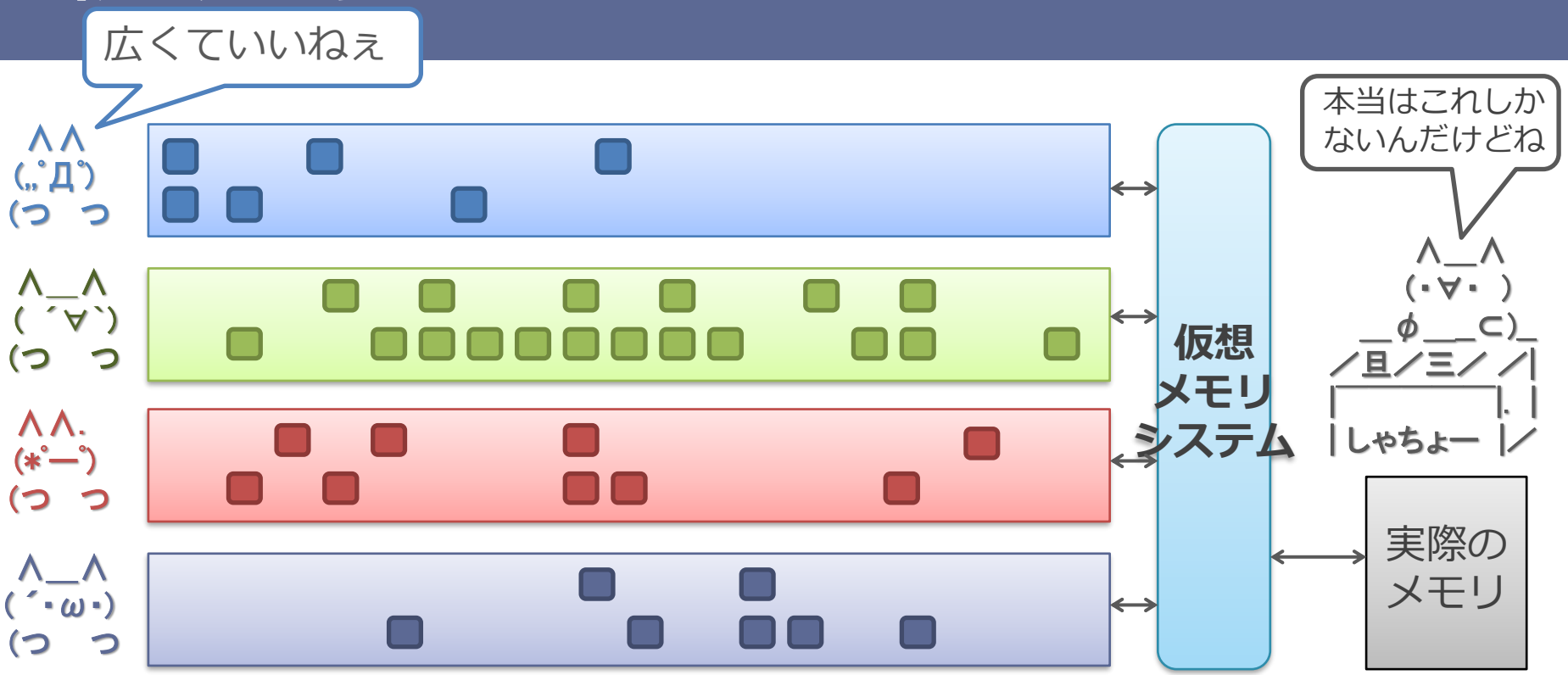
- 他の人のプログラムの領域を誤って上書きしてしまうことも
  - 例：バグで配列の最大サイズを超えて書き込むと，他のプログラムのメモリが破壊される
  - 上の図だと，緑の人の領域を青，赤，紫のデータが誤って上書き

2. どうやって各人の領域を保護する？



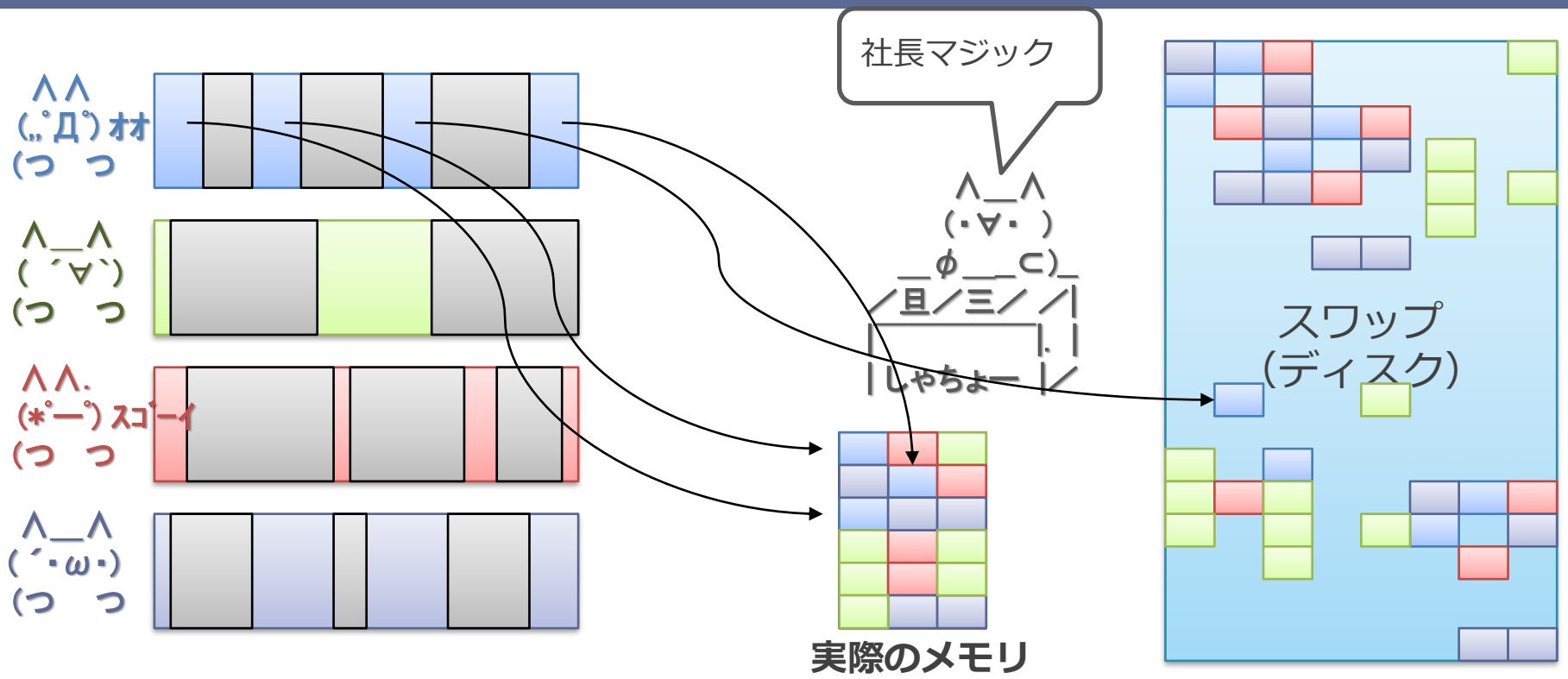
- 特に OS の管理領域がバグで破壊されると OS ごと落ちかねない
- OS の管理している領域をユーザーに勝手に見られるのもまずい

仮想メモリ



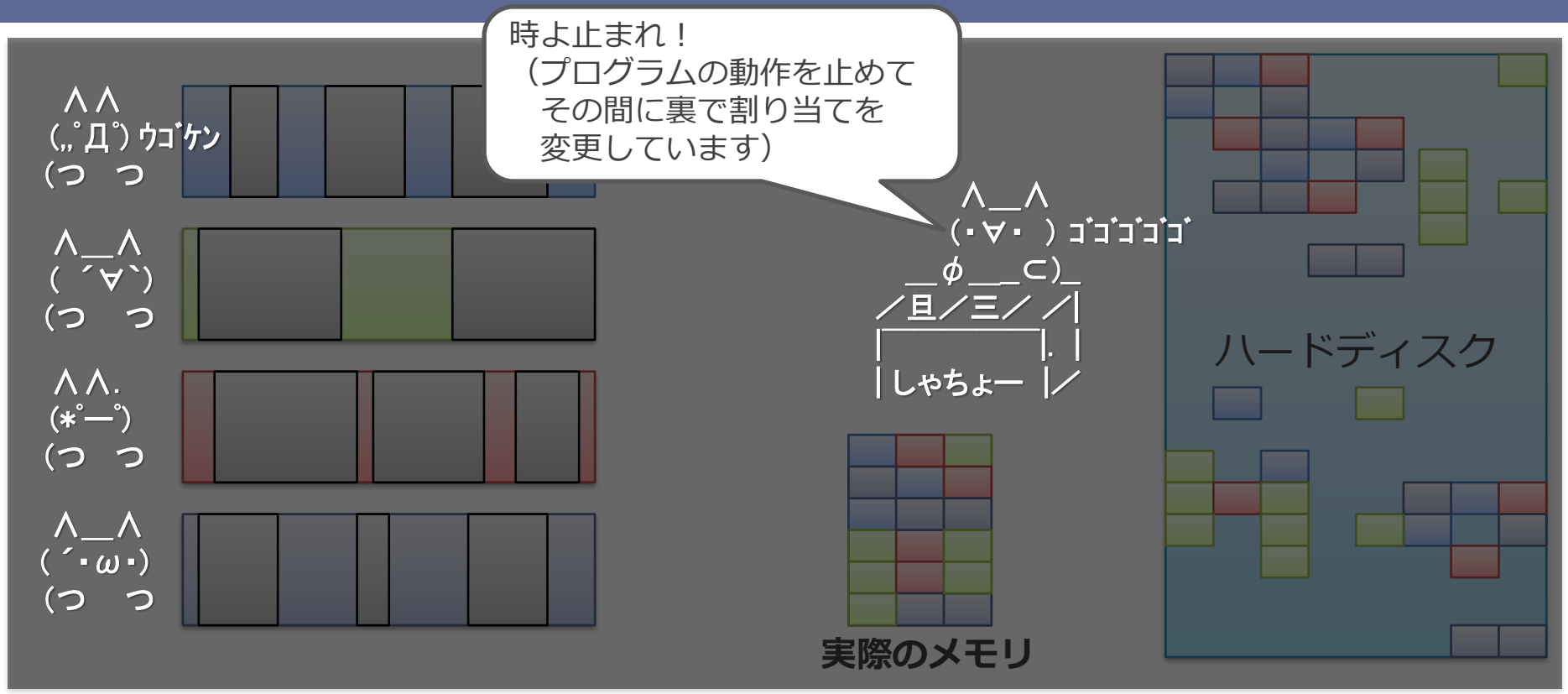
- プログラムごとに専用の大きなメモリが用意されているように「見せかける」技術
  - プログラムからは「自分専用の」メモリがあるかのように見える
  - アドレスで指定できる場所はすべて自分の空間
  - 他人の空間は読み書きできない

メモリのマップ



- 各人の仮想的なメモリの使っている部分が細切れに実際のメモリにマップされる
  - 足りない場合はより容量が大きな（そして遅い）ディスクにマップされる
    - これを「スワップ領域」という
    - ある意味，メモリがスワップのキャッシュになっている
  - これにより，効率的に実際のメモリを共有

マップの更新はユーザーからは透過的に行われる
（更新されていることを感知できない）



- これらの管理は OS によって，プログラムからは透過的に行われる
  - 透過的=プログラムの実行を止めて OS が裏で再割当てを行う
  - プログラマはこれらのことを意識しないで良い
  - というか，裏で動いているこれらの管理の動作は通常認識できない

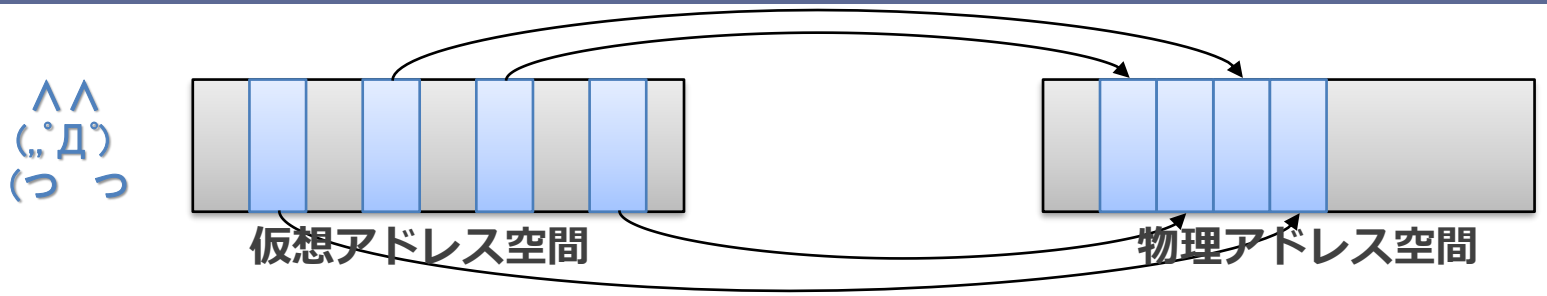
仮想メモリの基本のまとめ

- モチベーション：複数のプログラムでメモリをどうやって共有するか
  1. どうやって領域の割り当てを行う？
  2. どうやって各人の領域を保護する？
- 仮想メモリ：プログラムごとに専用の大きなメモリが用意されているように「見せかける」技術
  - プログラムからは「自分専用の」メモリがあるかのように見える

仮想メモリの詳細

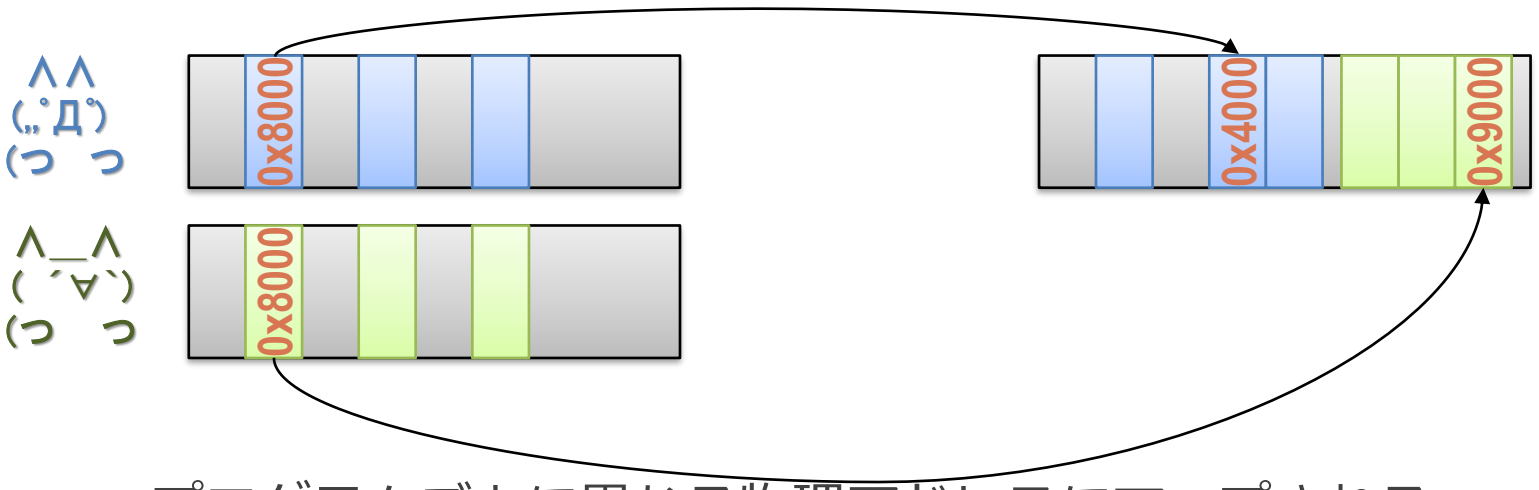
1. 仮想メモリの詳細
    1. 仮想アドレスと物理アドレス
    2. ページ・テーブル
    3. TLB

仮想アドレスと物理アドレス



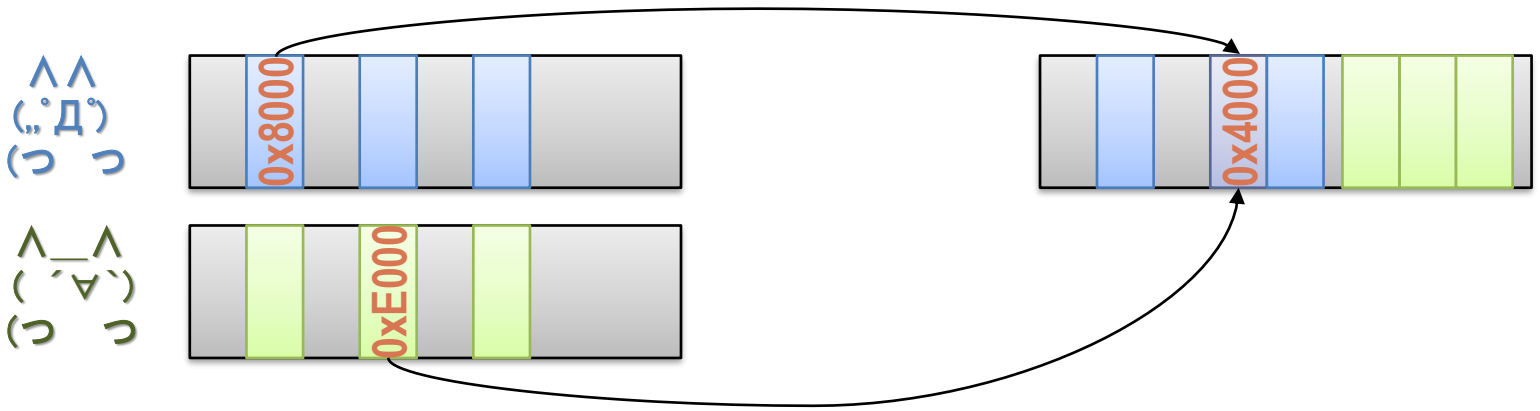
- 仮想アドレス（論理アドレスともいう）
  - プログラムから見えるアドレス
  - C 言語などでポインタに入っているアドレスの数字はこれ
- 物理アドレス
  - 物理的なメイン・メモリのアドレス
  - プログラムからは見えない（=どういう数字なのかはわからない）
- メモリ・アクセス時は，毎回仮想アドレスから物理アドレスにCPU が裏で変換してアクセスする

同じ仮想アドレスが指す物理アドレスはプログラムごとに異なる



- プログラムごとに異なる物理アドレスにマップされる
  - 上の例では，青の人のアドレス 0x8000 と緑の人のアドレス 0x8000 はそれぞれ異なるアドレスに変換される
- 正確にはプログラムごとではなくプロセスごと
  - 同じプログラムを複数立ち上げた場合，それぞれに専用の仮想アドレスの空間が提供される

逆に違う仮想アドレスから
同じ物理アドレスを共有することもできる



- 同一の物理アドレスを異なるプログラムの仮想アドレスから指す事もできる
  - プログラム間でデータのやり取りをするときなんかに使う
  - OS はこのあたりの機能をフル活用して作られている
  - （来学期の講義で詳しくやるはずです

仮想メモリの詳細

1. 仮想メモリの詳細
    1. 仮想アドレスと物理アドレス
    2. **ページ・テーブル**
    3. TLB

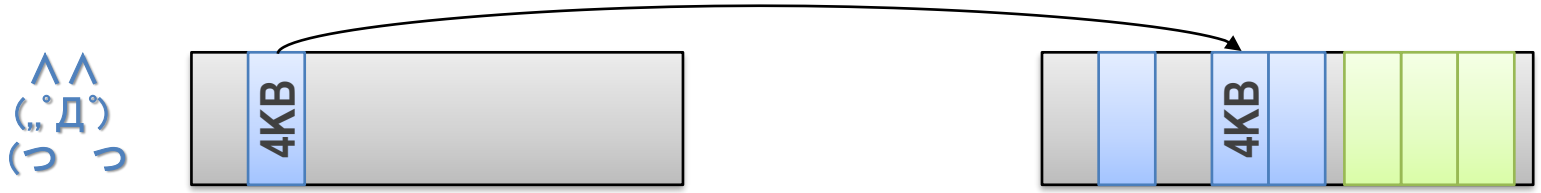
変換の実装

- 単純な実装
  - 仮想アドレス → 物理アドレス の変換表を用意すれば良い
- コスト：アドレスが 32 bit だとして，1 byte 単位で表を作ると…
  - データ 1 byte ごとに，その表には 32 bit = 4 byte のアドレスを記録することになる
  - 実データの 4 倍の容量が変換表に必要になってしまう

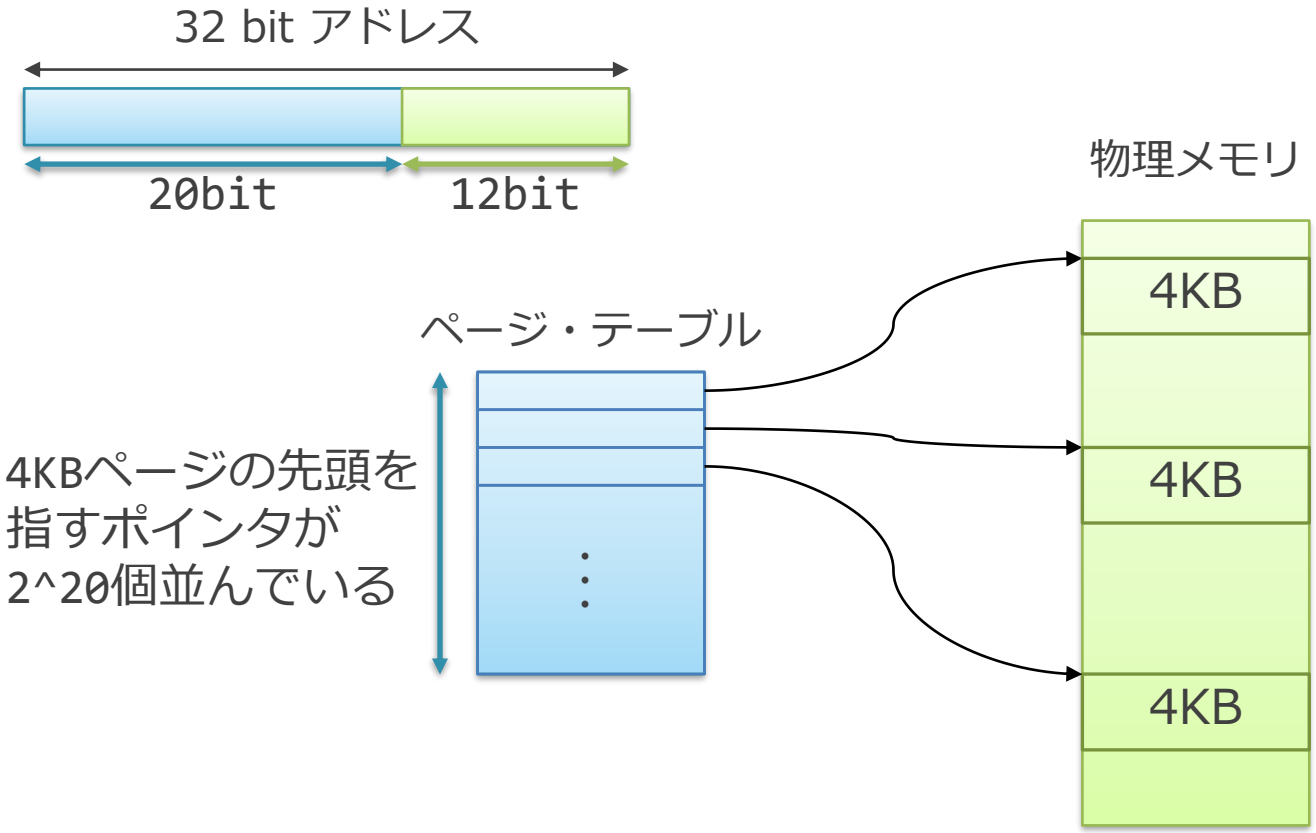
ページ単位での管理

- 変換表に必要な容量を減らすため，通常は「ページ」という単位でまとめて管理される
  - ページ単位の変換表を「ページ・テーブル」と呼ぶ
  - ページのサイズは 4KB から 数 MB ぐらい
    - 命令セットごとに仕様として決まっている

- 例：仮想アドレス上の連続した 4KB の領域（ページ）を，物理アドレス上の 4KB にマップ
  - 1 byte ごとに物理アドレスを覚える必要があったのが，4KB ごとでよくなる（4096 分の 1 の容量で済む）

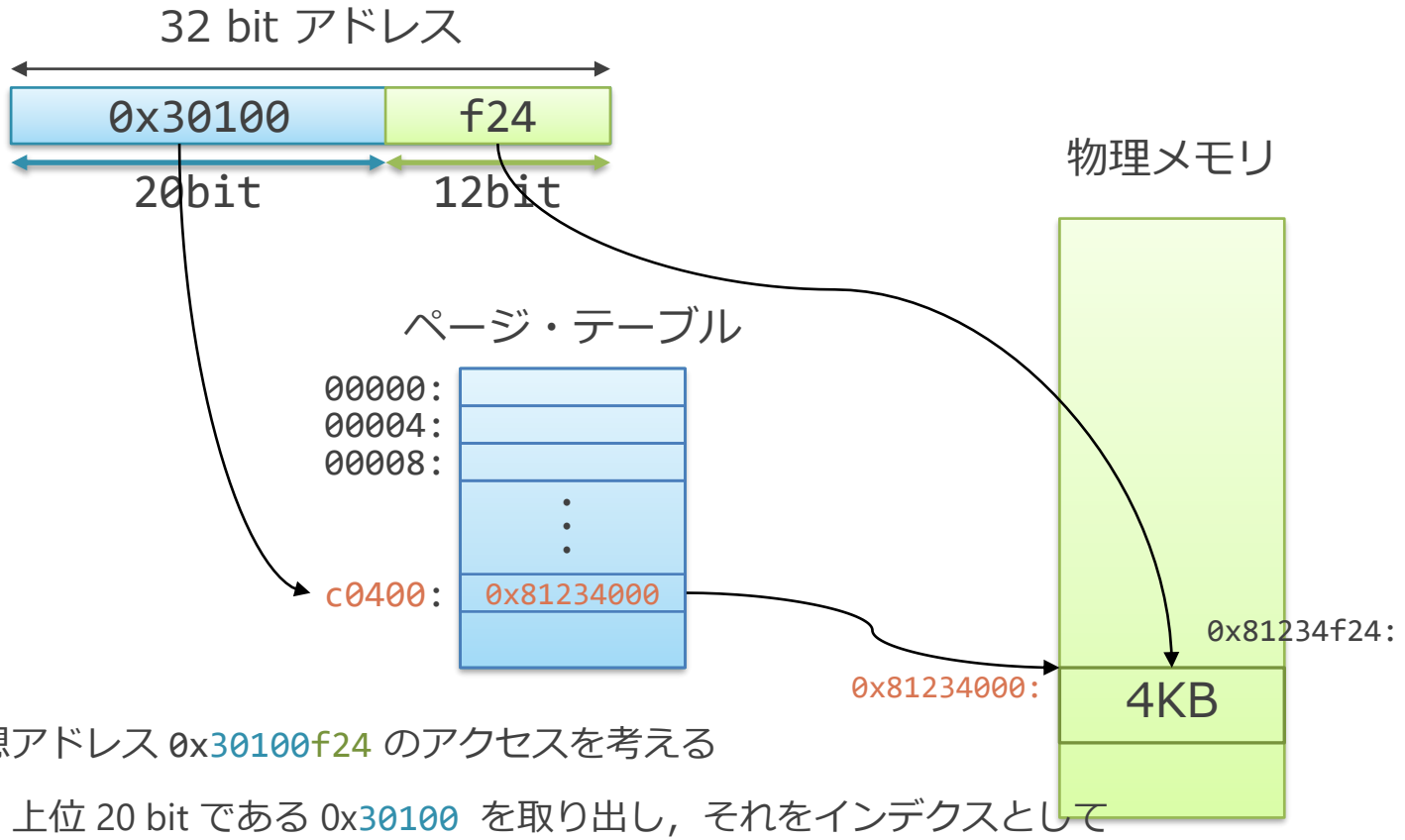


4KB ページを使った単段ページ・テーブルの例



- まず単段のページ・テーブルを考える
  - アドレス・サイズが 32 bit, ページ・サイズ 2^12=4 KB を仮定

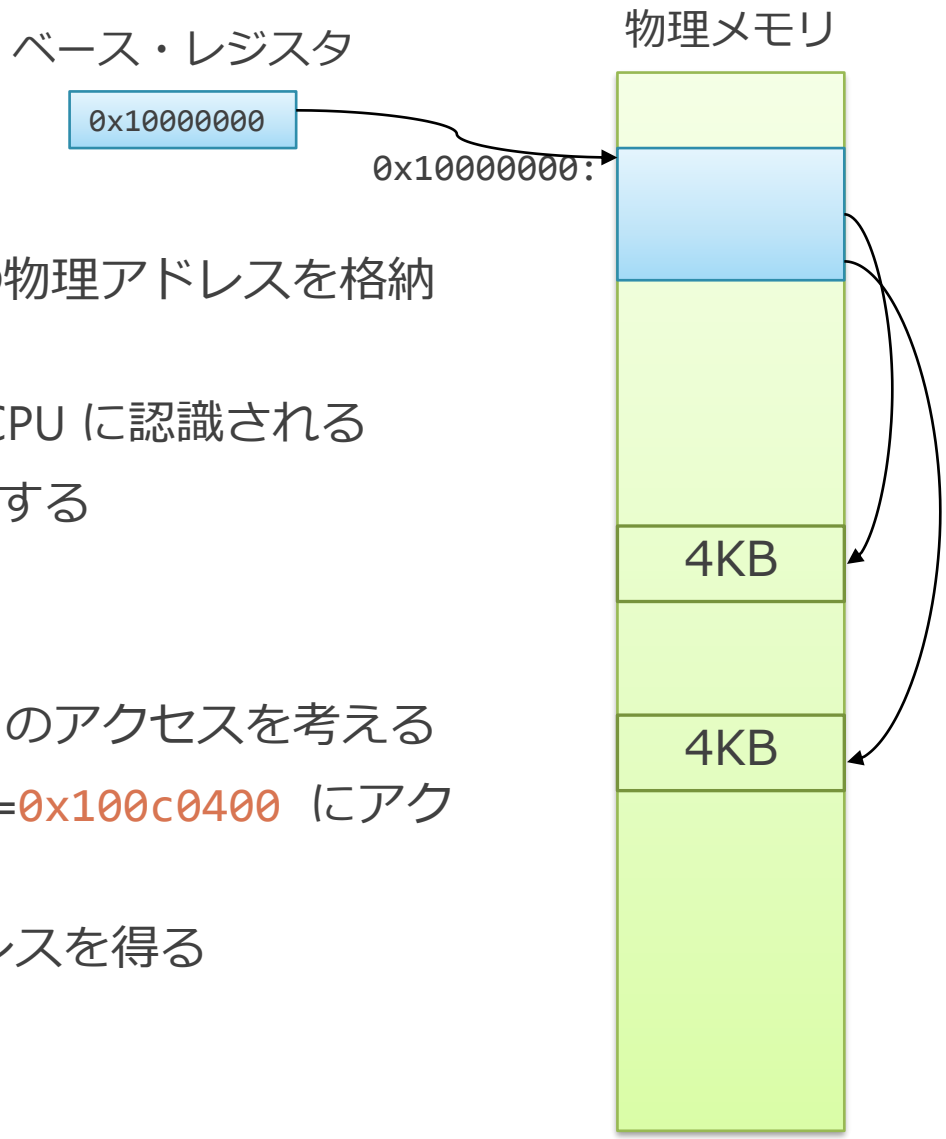
単段ページ・テーブルの動作



- 仮想アドレス 0x30100f24 のアクセスを考える
  - 上位 20 bit である 0x30100 を取り出し，それをインデクスとしてページ・テーブルにアクセス
    - 物理メモリへのポインタはここでは 4B 単位なので，
    - 0x30100 × 4B = 0xc0400 にアクセス
  - マップされているページの先頭の物理アドレス 0x81234000 を得る
    - 0x81234000 は OS がこの論理アドレスに割り当てたページのアドレス
  - 下位 12bit である f24 と結合して 0x81234f24 にアクセス

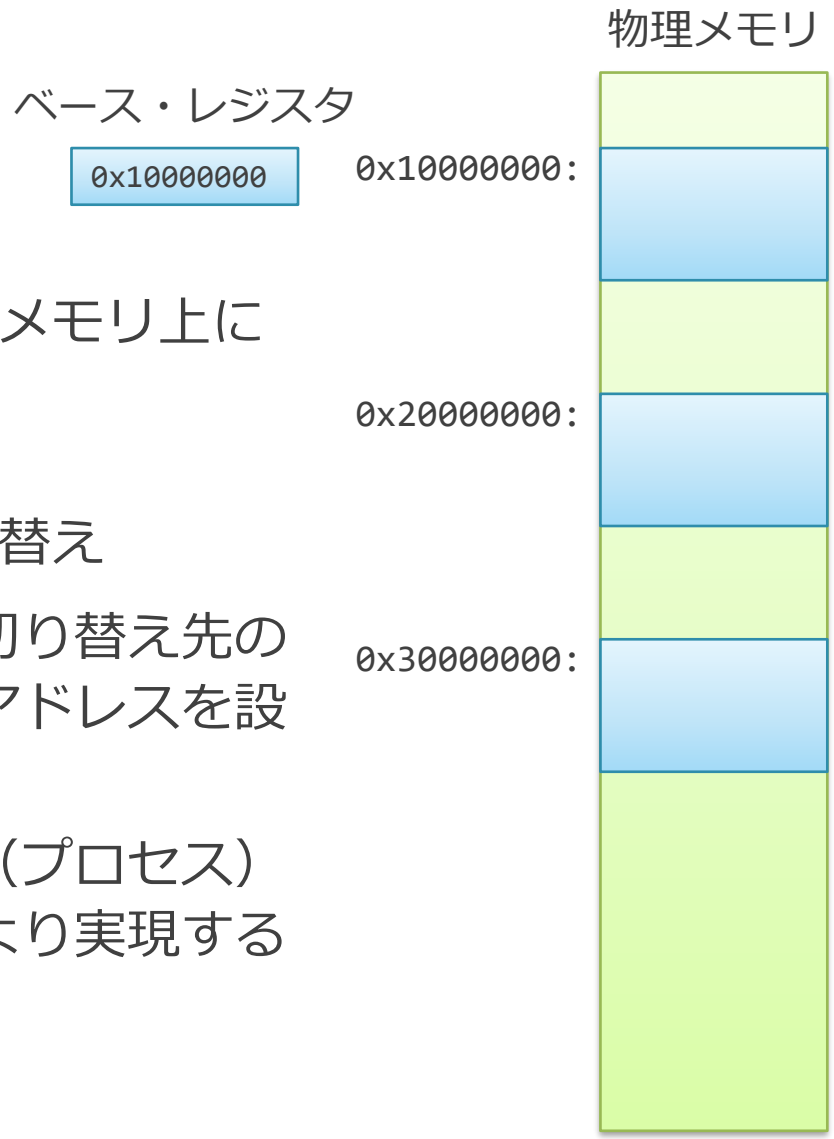
実際にはページ・テーブルも物理メモリ上に取られる

- ベース・レジスタ
  - ページ・テーブルの先頭の物理アドレスを格納する特別なレジスタ
  - そこが LV1 テーブルだと CPU に認識される
  - このレジスタは OS が設定する
- アドレス変換の手順
  - 仮想アドレス `0x30100f24` のアクセスを考える
  - `0x10000000+0x30100×4B=0x100c0400` にアクセス
  - 対応する 4KB の物理アドレスを得る
  - あとは前ページと同じ



プロセス切り替えはベース・レジスタの中身を入れ替えで実現する

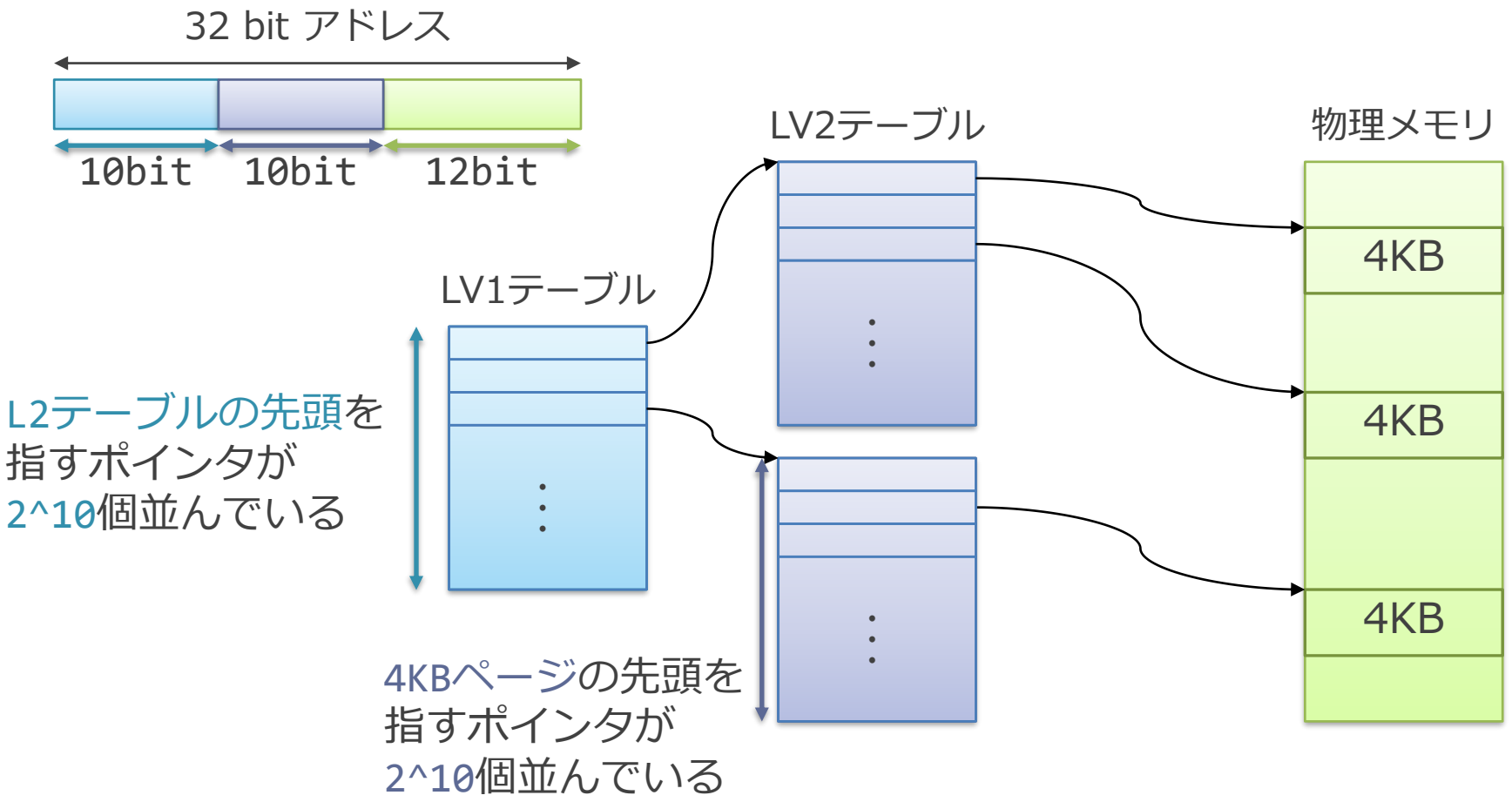
- ページ・テーブルは物理メモリ上に複数存在できる
- ページ・テーブルの切り替え
  - ベース・レジスタに切り替え先のページ・テーブルのアドレスを設定する
  - 実行するプログラム（プロセス）の切り替えはこれにより実現する



多段ページ・テーブル

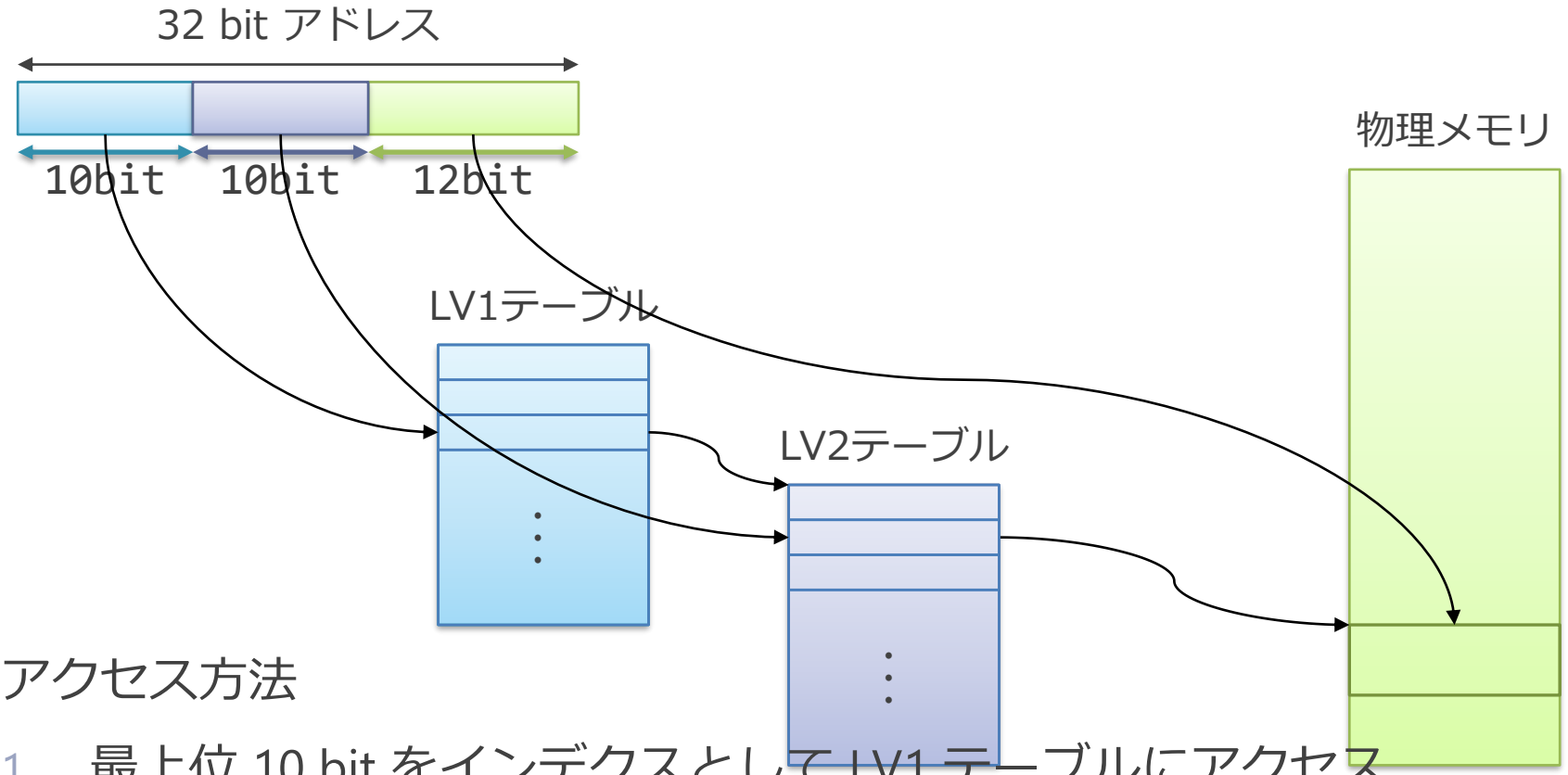
- ページ単位で管理したとしても，なおページ・テーブルは大きい
  - 64 bit のアドレス空間で，ページ・サイズを 4KB とした場合，
  - （アドレスの個数）/（ページ・サイズ）*（アドレスのサイズ） = （2^64）/ 4KB * 64bit = 16EB / 4KB * 8B = 32PB
    - たとえ 1B しかメモリを使わないプログラムでも 32PB が必要に
    - 32 bit のアドレス空間なら大分ましだが，それでも 4MB が必要
- 多段ページ・テーブルと呼ぶ構造で効率良く保持する
  - プログラムで使うメモリ容量に比例した程度の容量でページ・テーブルを作る方法

2段ページ・テーブルの例



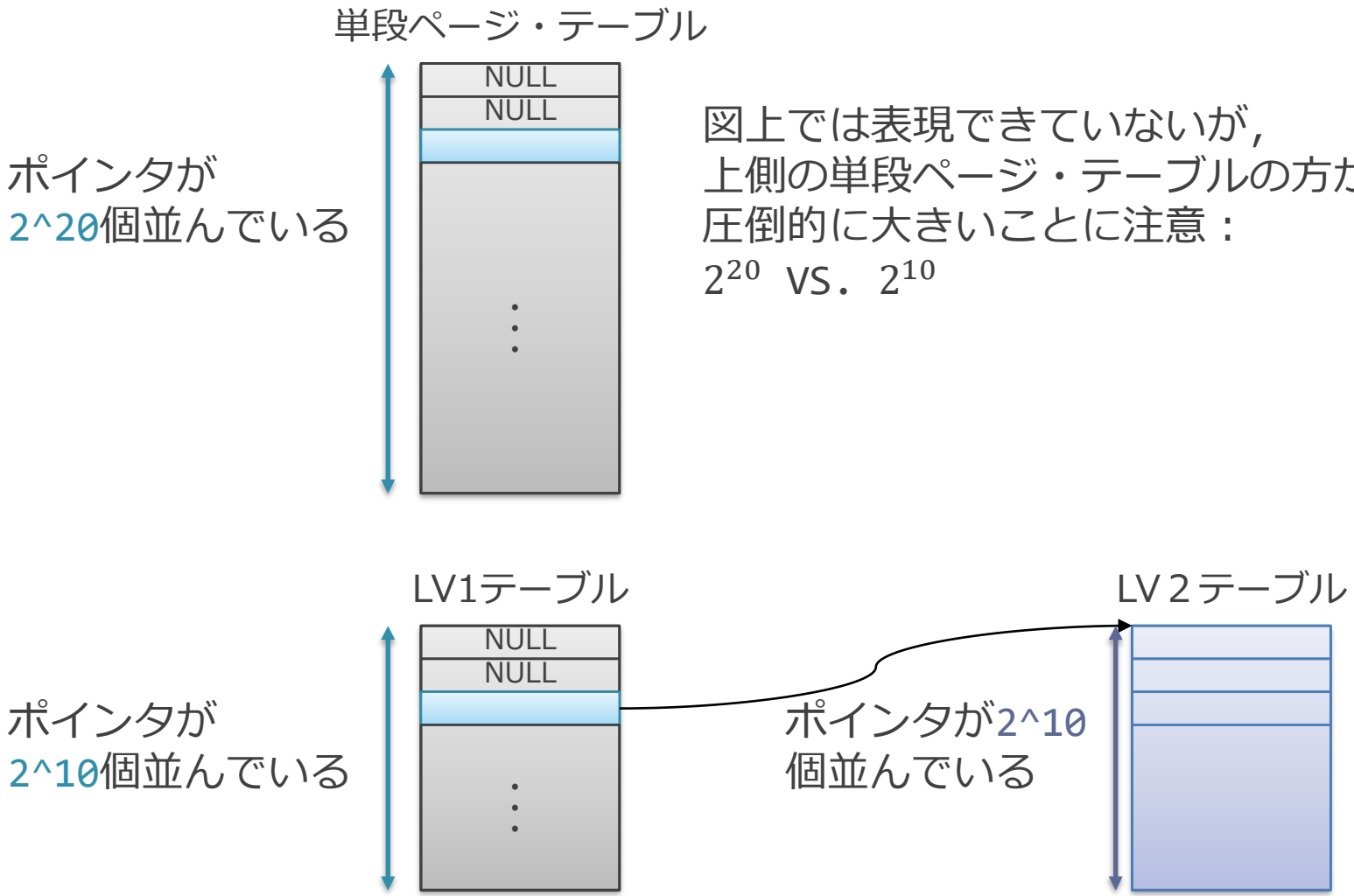
- 複数段のテーブルを経て物理メモリにアクセス
  - LV1テーブル → LV2テーブル → 物理メモリ

2段ページ・テーブルのアクセス



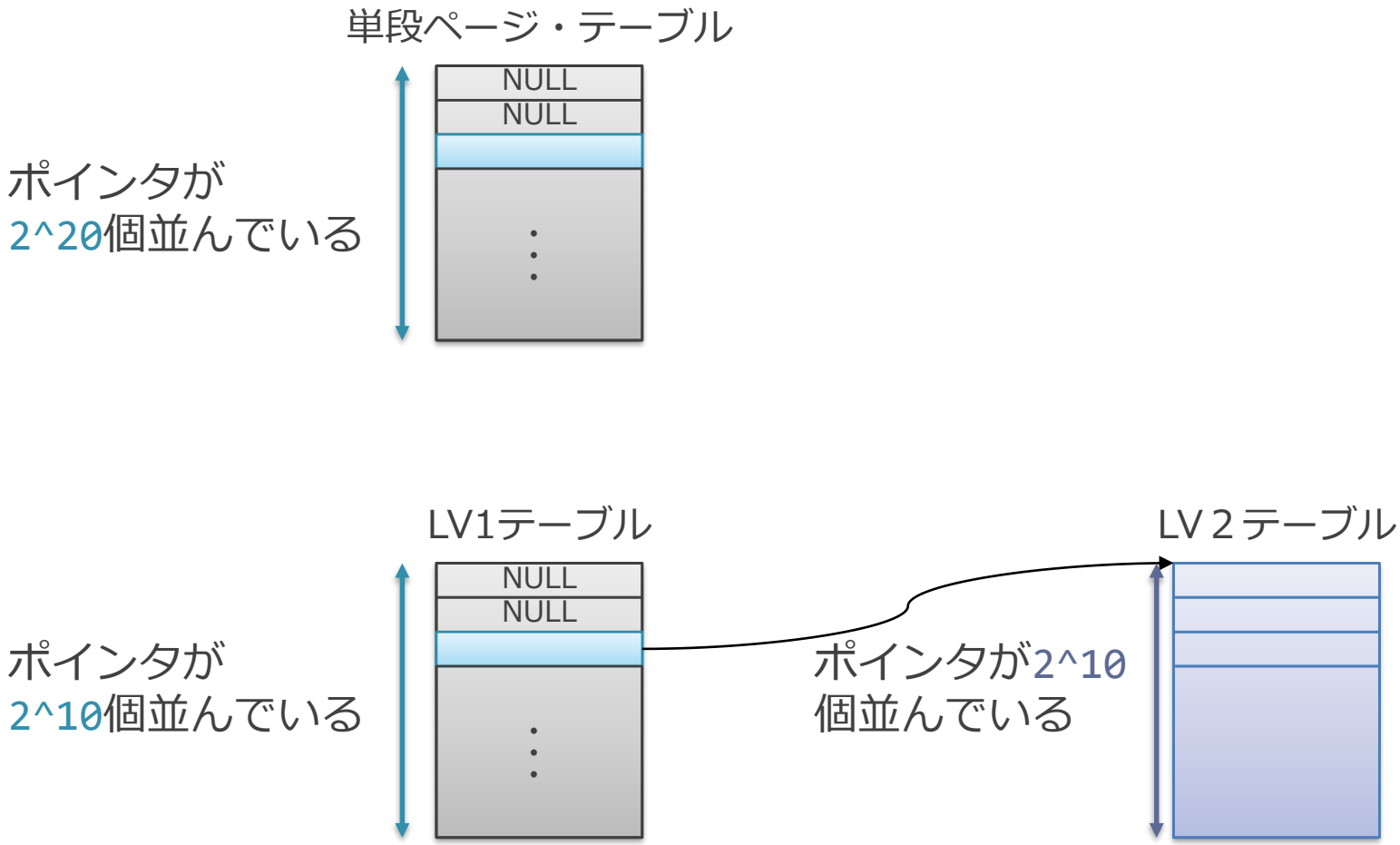
- アクセス方法
  1. 最上位 10 bit をインデクスとして LV1 テーブルにアクセス
  2. 次の 10bit をインデクスとして，1. で得られた LV2 テーブルの先頭アドレスにアクセス
  3. 最下位 12 bit をインデクスとして得られた物理メモリ上の 4KB 領域にアクセス

２段ページの利点：必要な容量が少ない



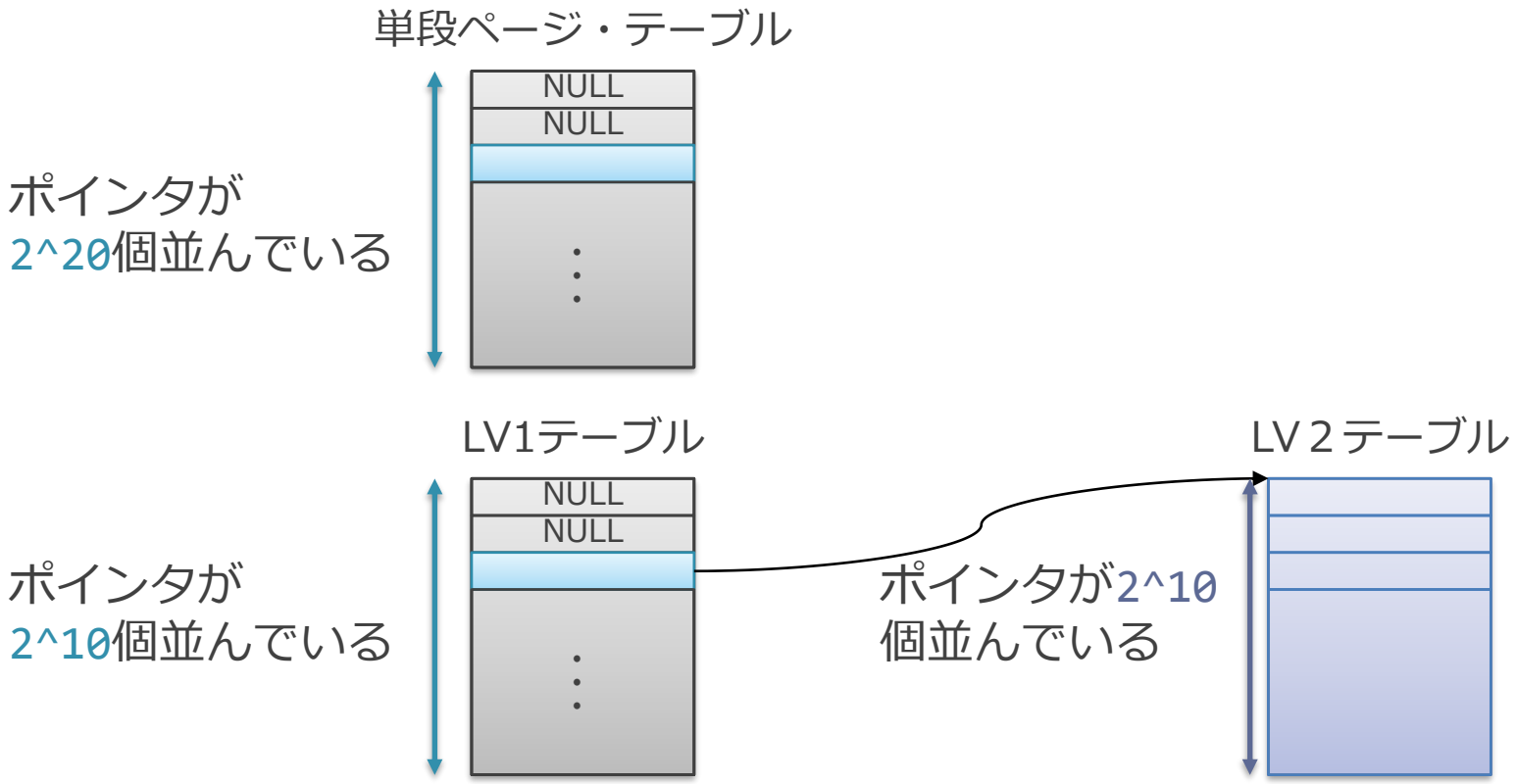
- 確保された領域に対応するエントリのみ，有効なポインタが入る
  - 未確保の領域は無効なポインタが入る

２段ページの利点：必要な容量が少ない



- 単段のテーブルと多段の LV1 のテーブルは固定長にせざるを得ない
  - どこに有効なポインタが入っているかわからない
  - LV2 テーブルはその領域が確保された場合のみ存在

2段ページの利点：必要な容量が少ない



- 4KB のメモリを確保したときにページテーブルに必要な容量：
  - 単段：ポインタが 2^20=1024K 個
  - 2段：ポインタが 2^10+2^10=2K 個

ページ・テーブルの詳細

- 実際には容量効率を重視してもっと多段になっている
  - x86_64 だと4段
- ページ・テーブルをたぐって仮想アドレスから物理アドレスを得る操作を「ページ・テーブル・ウォーク（Page Table Walk）」と呼ぶ

ページ・テーブルの管理

- ページ・テーブルを使ったアドレス変換は基本的に CPU が行う
  - このため，ページ・サイズなどのテーブルの構造は CPU ごとに仕様で決まっている
  - 昔は変換の一部分をソフトで行うものもあったが，今は大概ハードで完結して行う
    - ソフトが介在する場合，オーバーヘッドが非常に大きい
- ページ・テーブルの中身の更新はソフト（OS）で行う
  - 全部ハードでやるとあまりに大変
  - 更新は変換よりも頻度がかなり低い
  - OS ごとにどのようにマップしたいかも異なるし，柔軟性をもたせたい

MMU: Memory Management Unit

- MMU：
  - これまでに説明した仮想メモリの仕組みを実現するハードウェアのこと
  - ページ・テーブルのアクセスによる変換などを行う

仮想メモリの詳細

1. 仮想メモリの詳細
    1. 仮想アドレスと物理アドレス
    2. ページ・テーブル
    3. **TLB**

ページ・テーブルの速度面のオーバーヘッド

- 多段テーブル
  - x86_64 の場合，４段のページテーブルになっている
    - より容量効率を重視
  - これだとメモリ・アクセス毎に追加で４回のアクセスが発生
    - 毎回こんなことしてたら遅くなりすぎてとても耐えられない

TLB: Translation Lookaside Buffer

- ページ・テーブルのキャッシュ
  - ウォークの結果得られた物理アドレスをキャッシュ
  - 役割は完全にキャッシュだけど，歴史的経緯でバッファと呼ぶ
- 仮想アドレスの上位アドレスでアクセス
  - ヒットすると，対応するページの物理アドレスが一発で得られる
  - ミス時は通常のウォークを行って物理アドレス

TLB: Translation Lookaside Buffer

- 64 エントリぐらい用意されるのが典型的
  - 高速性を優先してエントリ数は非常に少なくなっている
    - ロードやストアの実行の度にアクセスされるから
  - カバーできる範囲が 64 x 4KB=256KB と意外とせまい
- プログラムの実行が切り替わる度にフラッシュされる
  - 仮想アドレスはプログラム間で同じ値が使われる
  - 最近はプロセス識別子というものが導入されて，フラッシュを避けていることもある
    - 「仮想アドレス+プロセス識別子」でアクセスする

仮想メモリのまとめ

- 仮想メモリ：
  - プログラムごとに専用の大きなメモリが用意されているように「見せかける」技術
- ページ・テーブル
  - 仮想アドレスから物理アドレスへの変換表
  - TLB はページ・テーブルのキャッシュ

今日の内容

1. 仮想メモリ
2. **特権モード**

モチベーション

- 仮想メモリ
  - プログラムごとに専用の大きなメモリが用意されているように「見せかける」技術
- ページ・テーブルの操作は誰が行うのか？
  - 各プログラムが勝手に好き勝手に操作できてはまずい
  - 他のプログラムのメモリへのアクセスが自由にできてしまう

特権モード

- CPU 内に用意されている特権モード
  - ユーザー・モード
    - 通常のプログラムが動くモード
    - ページ・テーブルの操作は制限されている
    - グラフィックやディスクなどの外部デバイスへの操作も制限されている
  - カーネル・モード
    - OS が動作するモード
    - ページ・テーブルの操作などはこのモードの時しか行えない

システム・コール

- 特権が必要な操作を行う場合，OS に要求をなげて実行してもらう
  - カーネル・モードで動く OS に依頼する
  - メモリ確保やファイル読み書き，ページ・テーブルの操作など
- これらの操作は必ず OS を介す
  - 特定のプログラムによりコンピュータ全体の動作を破壊することはできないようにしている
  - たとえば，「あるプログラムからコンピュータ上の全てのメモリにゼロを書き込む」とかはできない

システム・コール

- 特権が必要な操作を受け付ける関数をシステム・コールと呼ぶ
  - 呼び出しのために，モード遷移を伴う特殊な関数呼び出し命令が用意されている
  - あらかじめ OS で設定された固定のアドレスに強制ジャンプする
    - ユーザー・モードから任意の場所に飛べるわけではない

システム・コール

- RISC-V の場合：
  - ユーザー・モード から ecall 命令を実行するとカーネル・モードに
  - カーネル・モード から eret 命令を実行するとユーザー・モードに

RISC-V 64bit Linux の場合

（具体的なレジスタ番号とかは違うかもしれないですが，間違ってたらスイマセン）

- RISC-V 64bit Linux の場合の例
  - ファイルをリネームするシステム・コール rename() の呼び出し
  - 手順：
    - レジスタ x17 にシステム・コールの識別番号を設定
      - RISC-V Linux の rename の場合は 1034
    - x10~x13 に引数を設定
    - ecall を実行
  - 注：OS ごとにレジスタの使い方等のルールは自由なので，みんな違う

```
// https://github.com/riscv/riscv-linux/blob/riscv-next/include/uapi/asm-generic/unistd.h より
#define __NR_rename 1034
__SYSCALL(__NR_rename, sys_rename)
#define __NR_readlink 1035
__SYSCALL(__NR_readlink, sys_readlink)
...
```

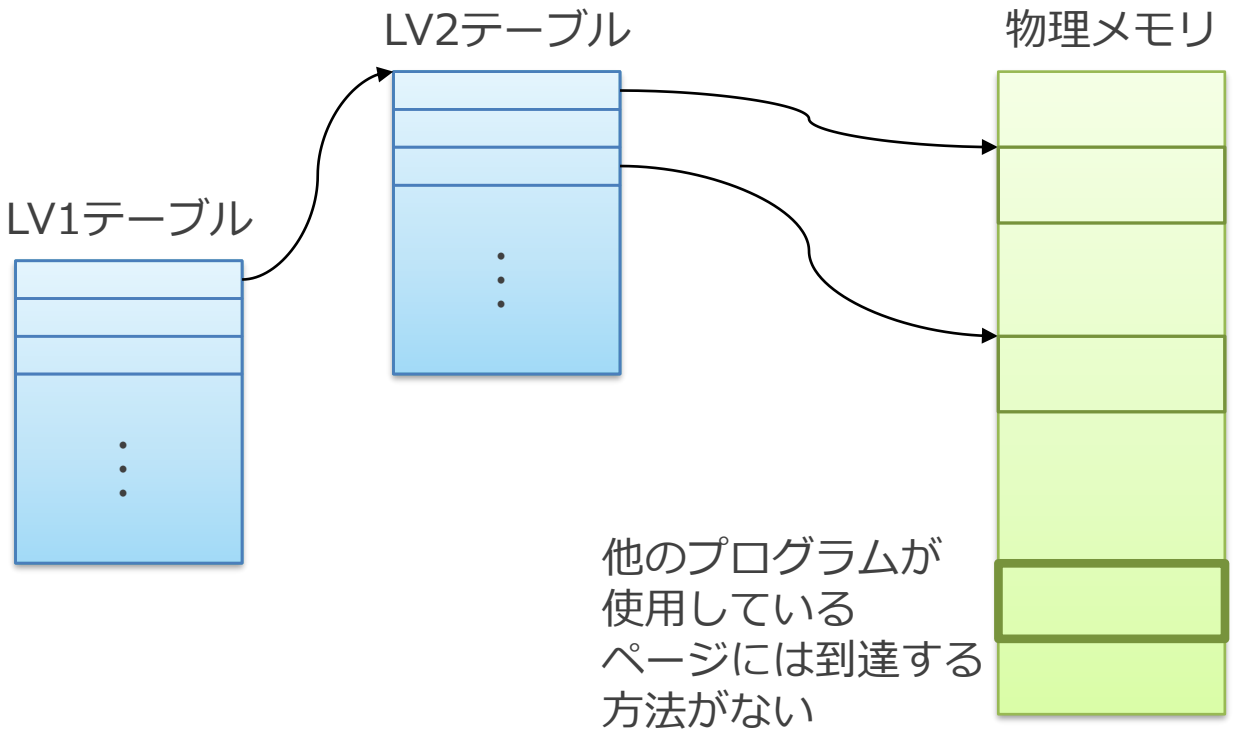
```
li x17 ← 1034  // システム・コール rename の要求番号を設定
ld x10 ← (...) // リネーム対象のファイル名が入っている文字列へのポインタをロード
ld x11 ← (...) // リネーム後のファイル名が入っている文字列へのポインタをロード
ecall          // システム・コール呼び出し．返り値は x10 に入る
```

システム・コールによるメモリの確保

- Linux では通常はシステム・コール mmap によってメモリを確保
  - malloc とかを呼ぶと，その奥底では mmap が呼ばれている
- mmap には確保したいメモリのサイズを渡す
  - mmap 内で要求サイズ分だけ仮想アドレスが使えるように，ページ・テーブルを更新
  - プログラムは返ってきた仮想アドレスを使用する
  - ページ・テーブルを直接操作することは通常はない

仮想メモリによる保護

- ユーザー・モードからは，他のプログラムが持つメモリは読めない
  - アドレス変換は自動かつ強制的に行われる
  - このため自分に用意されたページ・テーブルから指されていないものは参照しようがない
  - カーネル・モードはページ・テーブル自体を自由に切り替えられるので任意のメモリにアクセスできる

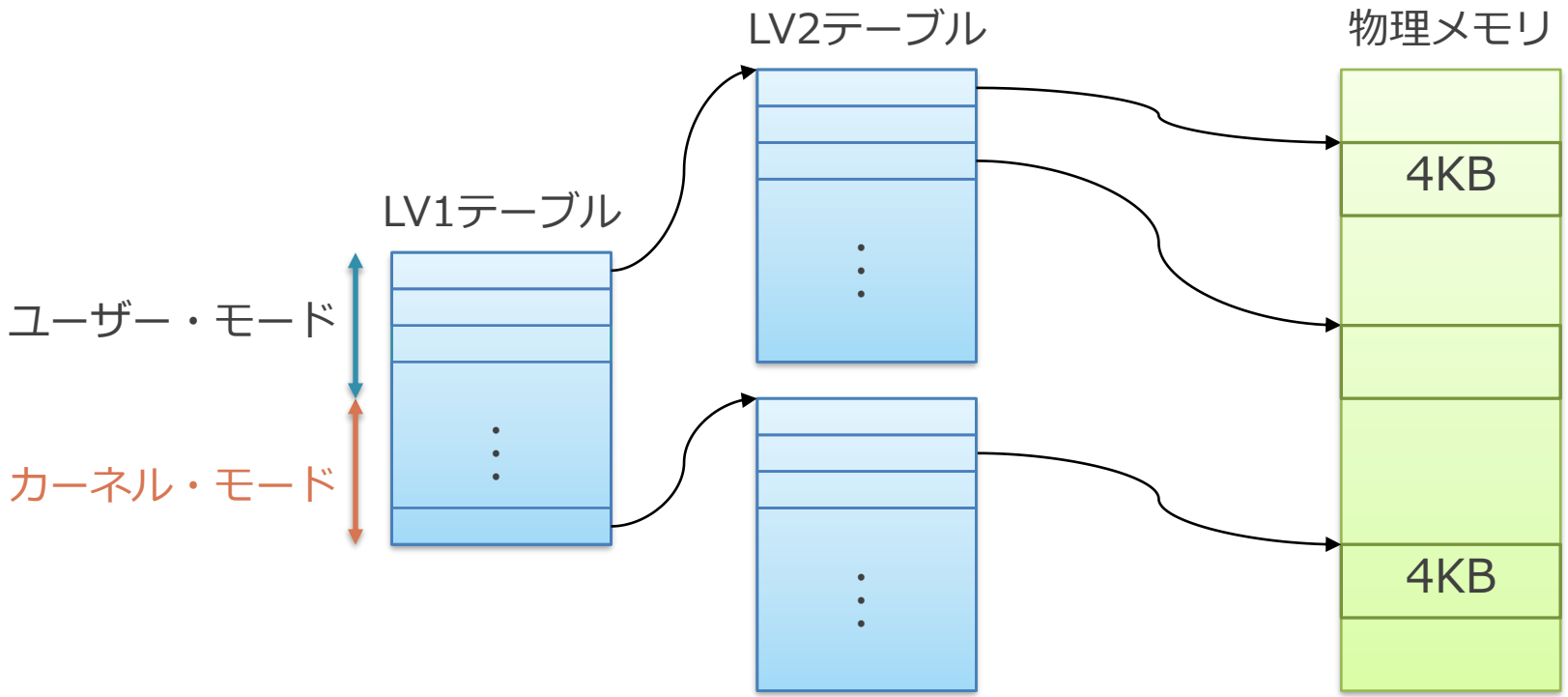


ページごとの保護

- ページごとにさらに，モードごとの権限を設定できる
  - ページ・テーブル内にポインタと一緒に格納
- 「カーネル・モードでは読めるがユーザー・モードでは読めない」のような属性が設定できる
  - これらのチェックに違反すると OS にプログラムの実行を止められる
  - 「Access Violation」や「Segmentation Fault」でプログラムが停止するのは，この機能による

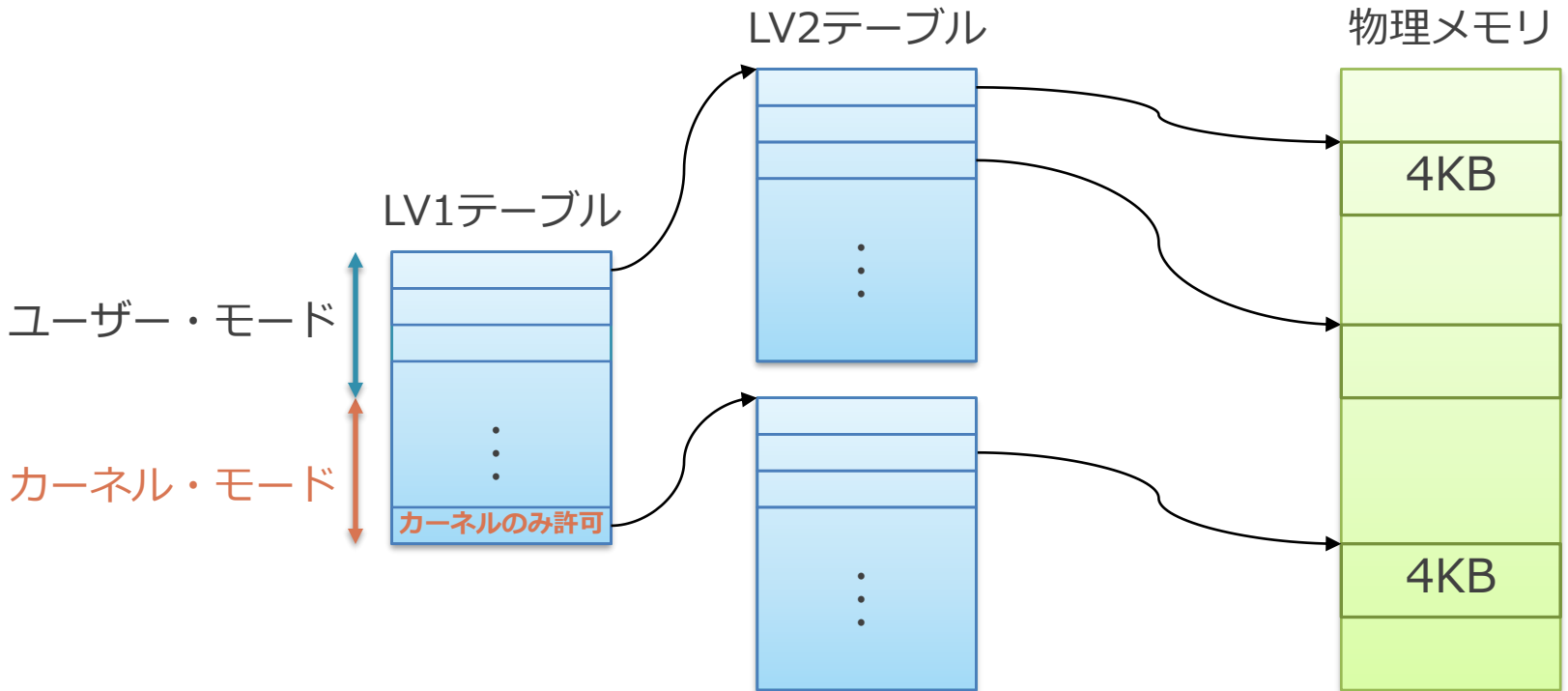
ページごとの保護を利用した仮想アドレスの共有による最適化

- ユーザー・モードと OS は仮想アドレスを共有することが多い
  - たとえば，全てのプログラムの仮想アドレス空間の後ろ半分は OS が使用など
  - 利点：システム・コール呼び出し時にページ・テーブルを OS 用仮想アドレスに切り替えなくてよくなる



ページごとの保護を利用した仮想アドレスの共有による最適化

- ページごとの権限を利用して保護
  - ユーザー・モードからでも OS の物理メモリにページ・テーブルを介して到達可能
  - カーネル領域はユーザーから読むと落ちるよう設定するので安全
  - エントリに「カーネルのみ許可」と権限が設定される



仮想メモリと特権モードによる保護のまとめ

- CPU には操作できる権限が設定されたモードがある
  - ユーザー・モード
  - カーネル・モード
- ユーザー・モードではメモリなどを変更する操作は自由には行えない
  - カーネル・モードで動作する OS に依頼して行う
  - 当然他人のファイルやメモリへアクセスしようとすれば落とされる
- 他のプログラムや OS 領域のメモリを読むことは基本的にできない
  - プログラムごとに独立した仮想アドレス空間を提供
  - ページごとのアクセス権限の設定

まとめ

1. 仮想メモリ
    1. モチベーションと基本
    2. 詳細
        1. 仮想アドレスと物理アドレス
        2. ページ・テーブル
        3. TLB
2. 特権モード
    1. システム・コール
    2. メモリ保護

課題 １１

- 以下のような状況の仮想メモリについて考える：
  - 仮想アドレス空間と物理アドレス空間は共に 32bit である
  - 単段ページ・テーブルを使用
  - ページ・サイズは 64KB である
  - ベース・レジスタには物理アドレス 0x20000000 が設定されている
  - 仮想アドレス 0x10000000 と 0xfea50000 から始まるページには，それぞれ物理アドレス 0x30000000 と 0x30010000 から始まるページが割り当てられているものとする
  - TLB は存在しない

課題 １１

- (1) 仮想アドレス 0x10002000 に格納されている値を読み出す際にアクセスされる物理アドレスをすべてあげよ
- (2) 仮想アドレス 0xfea51fff に格納されている値を読み出す際にアクセスされる物理アドレスをすべてあげよ
- (3) 仮想アドレス 0x10000000 と 0xfea50000 から始まる 2 つのページ*のみ*が確保されている場合を想定する．この時に使用される物理メモリの容量の合計（ページ・テーブルとページそのもの）を求めよ
- (4) (3)と同様の条件で，２段ページ・テーブルを使用した場合に使用される物理メモリの容量の合計を求めよ．この時 LV1 と LV2 に使用されるアドレスのビット幅は等しいものとする．

提出方法

- 以下を提出：
  1. 課題１１：
     - 提出は Moodle の「課題１１」のところからお願いします
     - 紙に書いた場合は写真を撮ってアップロードしてください
  2. 感想や質問：
     - 「感想や質問」のところに投稿してください
     - わからない場所がある場合，具体的に書いてもらえると良いです

- 提出締め切り
  - Moodle に設定した締め切りまで

- 注意：
  - 課題の出来は，ある程度努力したあとがあれば良しです
    - 必ずしも正解していなくても良いです

期末試験について

- 7/30 にいつもの講義の時間にここで実施
- A4 １枚手書きのみの持ち込み可
  - 裏表双方へ書き込み可
  - 印刷したものを一部貼り付けるなどはダメ
  - 必ず全部完全に手書きにしてください
- 基本的に課題で出した部分を中心に出題する予定
  - 練習問題と課題を中心に勉強しておいてください

来週 7/23 について

- 課題の解説と質問に答える回になります
  - 必ずしも出席しないで良いです
  - この日は新しく課題は出さないです

質問とか感想

質問とか感想

- 行列積のところで、ijkの順番を入れ替えていい、というところがよく分からなかったです。

```
// a[] に b[]*c[] を足しているが，足し算はどんな順番で
// やっても結果は変わらない（交換法則）
// i,j,k の 0 から SIZE-1 までの全ての組み合わせが計算
// されていれば良い
for (int k = 0; k < SIZE; k++) {
    for (int j = 0; j < SIZE; j++) {
        for (int i = 0; i < SIZE; i++) {
            a[k][j] += b[k][i] * c[i][j];
        }
    }
}
```

質問とか感想

- ダイレクトマップが実際に使われることは少ないとのことで、授業を聞いている時はなぜ少ないのかよく分からなかったのですが、課題をやってみてこれは少ないだろうな、と実感できました。

質問とか感想

- Missしたときにメモリにアクセスというのがわかりませんでした。データからタグを探してくるということですか？

質問とか感想

- 行列になる理由は二次元配列になるからでしたっけ…？ちょっと理解が追いついていないです。もう一度解説していただけると助かります。

行列の２次元配列による表現

```
uint32_t A[2][2];
```

$$\begin{bmatrix} A[0][0], A[0][1] \\ A[1][0], A[1][1] \end{bmatrix}$$

- A[y][x] の場合：
  - 1次元目（x）：何列目か
    - x が増えると参照位置が右に移動
  - 2次元目（y）：何行目か
    - y が増えると参照位置が下に移動

質問とか感想

- 行列の考えが使われていることに驚いた。今まで習ってきた行列は２次元のものだったが３次元のものもあったりするのかどうか気になった。

質問とか感想

- 連想度が２の時、２セットになると思うのですが、１セット目に入れるのか２セット目に入れるのかどこで判断すればいいのかわかりませんでした。
  - 連想度が２の場合，１つのセットに２ラインが入ります
    - セット数は連想度とは無関係です
  - 長い時間アクセスされてなかった方が上書きされます

質問とか感想

- 課題10(2)で、どういう時にhitになるのかがよくわかりませんでした。タグ部分が一致している時hitになるという認識でしたが、例題スライドp35では
  - 1 0x8010 miss
  - 2 0x8010 hit (時間的局所性)
  - 6 0x8010 miss
- となっており、1,2で2のみhitなのは1と2が連続しているのが関係していると考えているのですが、6はなぜmissになっるのか、よくわかっていません。
  - 1. でキャッシュに乗せられたラインは 4. で上書きされていてキャッシュから消えているからです

質問とか感想

- アドレスとラインの対応のスライドで
  - 「4ビットなのは，ラインサイズが16バイトだから2 ^ 4 = 16 **(ラインサイズは必ず２の累乗になる)」**
    とありましたが、この部分が理解できませんでした。演習ではアドレスは16進数で表されていましたが、このスライドでは2進数で表されているとして説明しているということなのでしょうか。

- アドレスとセットの対応でセット数も必ず２の累乗になるのはなぜかもう一度解説を聞きたいです。

  - ２進数の一定の桁数でライン中の位置を表すので，
    そのサイズは２の累乗になるということです
  - たとえば３桁を割り当てる場合，2^3 = 8

質問とか感想

- 課題はスライドNo35あたりを参考に取り組みましたが、すべてセット0に書き込まれて競合が起きまくっていて心配です。ダイレクトマップ方式では特に台無しな感じになってしまっており、この方式の問題点を実感しました。

質問とか感想

- 今回の課題最初は何やればいいか全くわからなかったけれど、練習問題を見て、スライドを復習しながら解いたら、授業の内容も理解でき、問題も解けたので良かったです。

質問とか感想

- 「アクセス時の動作の例」を見て一瞬IEEE形式がよぎりましたが全然違いました。

質問とか感想

- 課題９の解説をもう一回お願いしたいです

質問とか感想

- ヒットがよくわからない。