先進計算機構成論 13

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

前回の内容

1. キャッシュ

補足：ブロッキング（タイリング）

```
for (int ii = 0; ii < SIZE; ii += BLOCK_II) {
  for (int jj = 0; jj < SIZE; jj += BLOCK_JJ) {
    for (int kk = 0; kk < SIZE; kk += BLOCK_KK) {
      for (int i = ii; i < ii + BLOCK_II; i++) {
        for (int j = jj; j < jj + BLOCK_JJ; j++) {
          for (int k = kk; k < kk + BLOCK_KK; k++) {
            a[k][j] += b[k][i] * c[i][j];
```

- ループをブロックに分割する
  - ◇ i, j, k がそれぞれ 0 ～ SIZE-1 をとるときに，最終的に全ての組み合わせが計算されていればよい
- キャッシュ・フレンドリーなアクセスへ
  - ◇ 六重ループも自由に順番が入れ替え可能
  - ◇ 各方向のブロックのサイズをよく考える

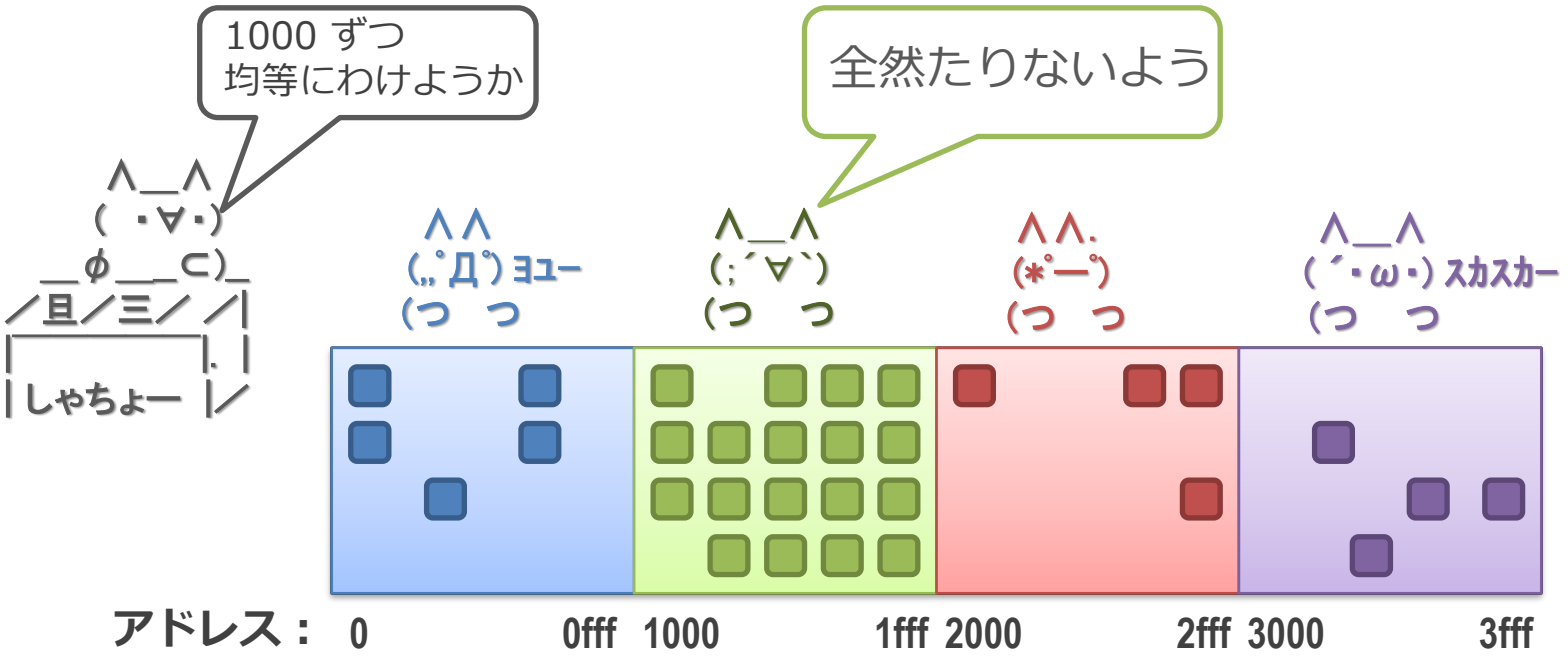
今日の内容

1. 保護機構
    1. **仮想メモリ**
    2. 特権モード
2. 脆弱性とアタック
    1. バッファ・オーバーフロー
    2. Return Oriented Programming
    3. マイクロアーキテクチャ面の脆弱性

仮想メモリのモチベーション

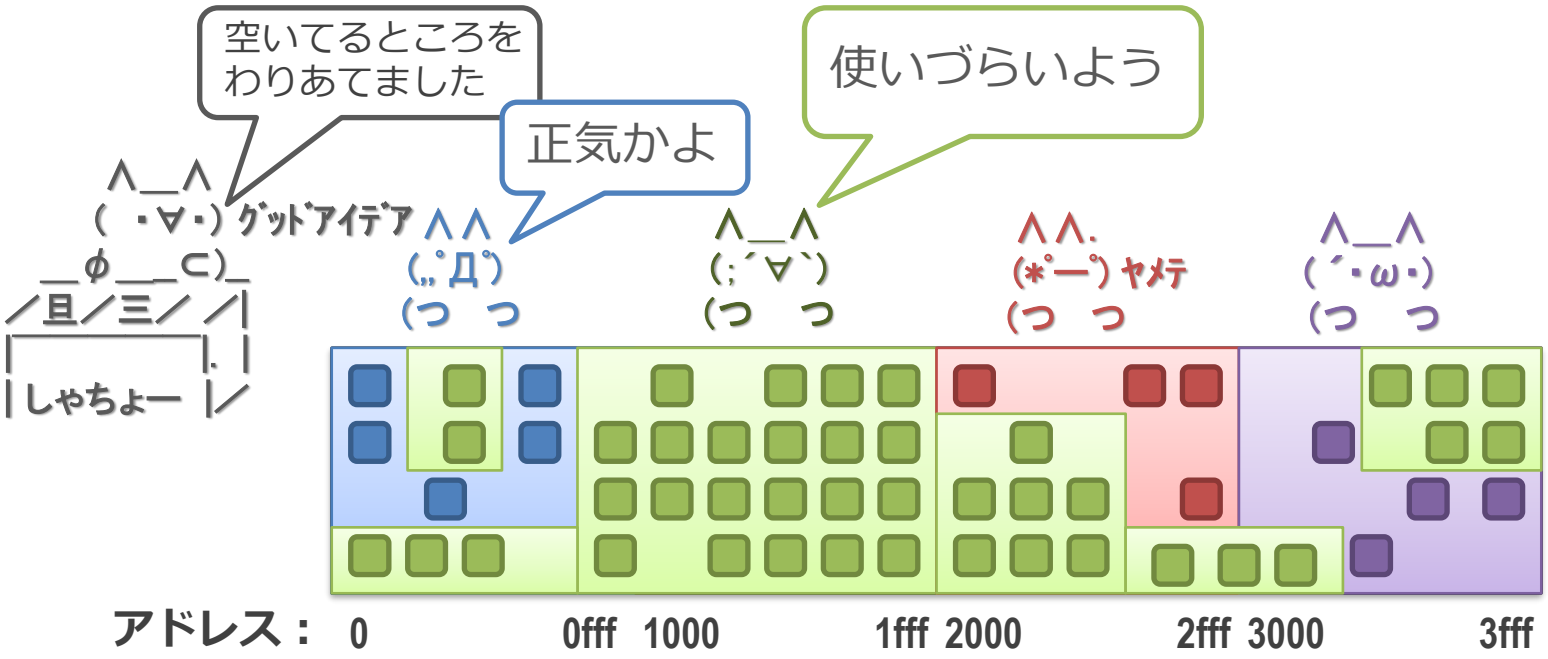
- 前提：複数のプログラムを同時に動かす場合を考える
  - ◇ メモリは複数のプログラムで共有される
- 問題：どうやって共有するか？
  1. どうやって領域の割り当てを行う？
  2. どうやって各人の領域を保護する？

1. どうやって領域の割り当てを行う？



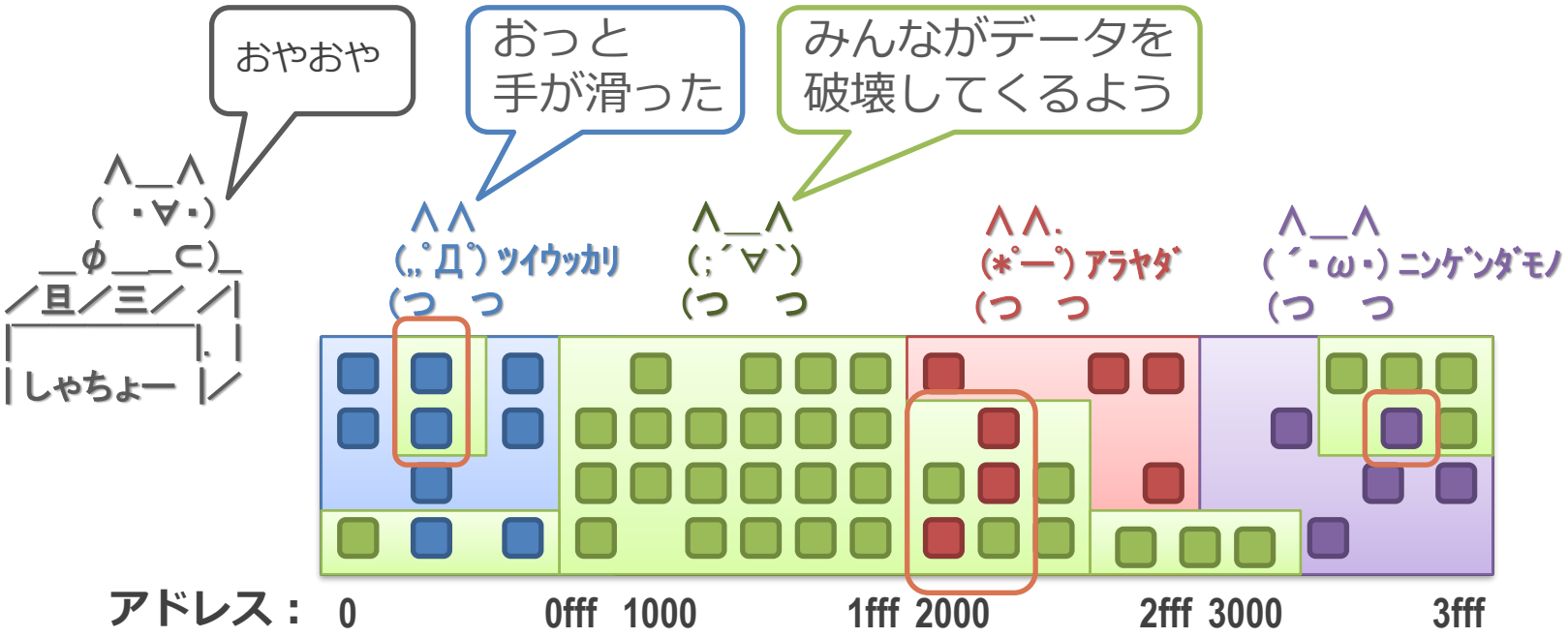
- 単純には均等に分ければ良い
  - ◇ しかし，プログラムごとに必要なメモリの量は違うのが普通

1. どうやって領域の割り当てを行う？



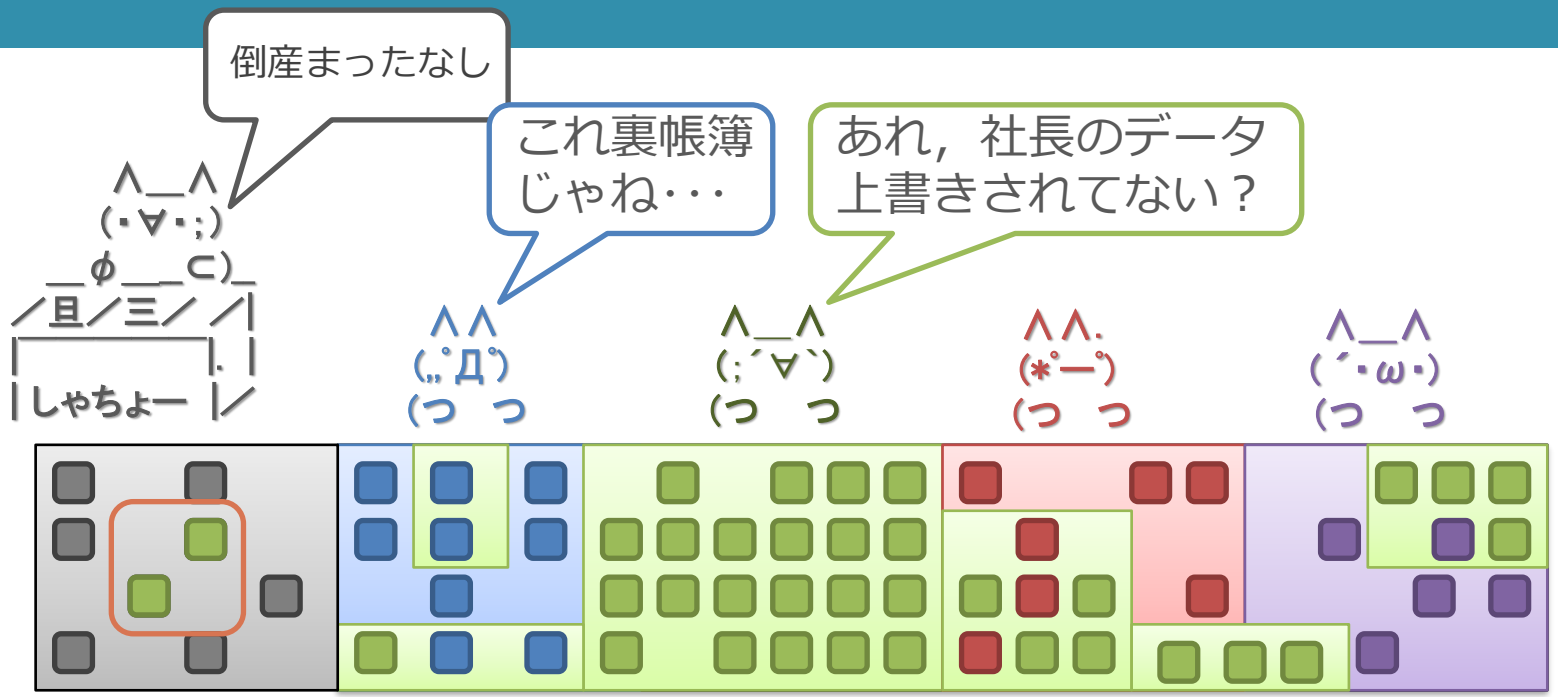
- たくさん使う人に都度割り当てると，メモリ空間が細切れになってとても使いにくい

2. どうやって各人の領域を保護する？



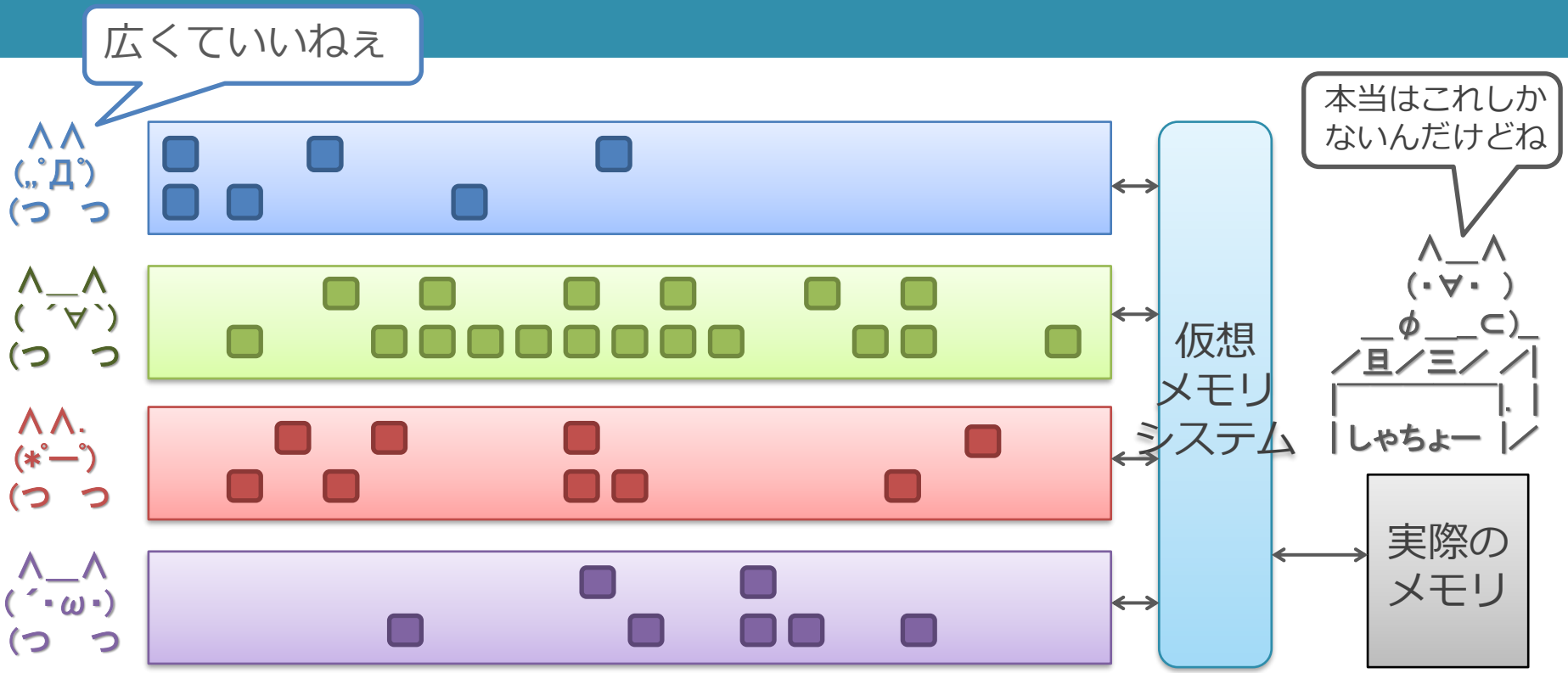
- 他の人のプログラムの領域をバグで誤って上書きしてしまうことも

2. どうやって各人の領域を保護する？



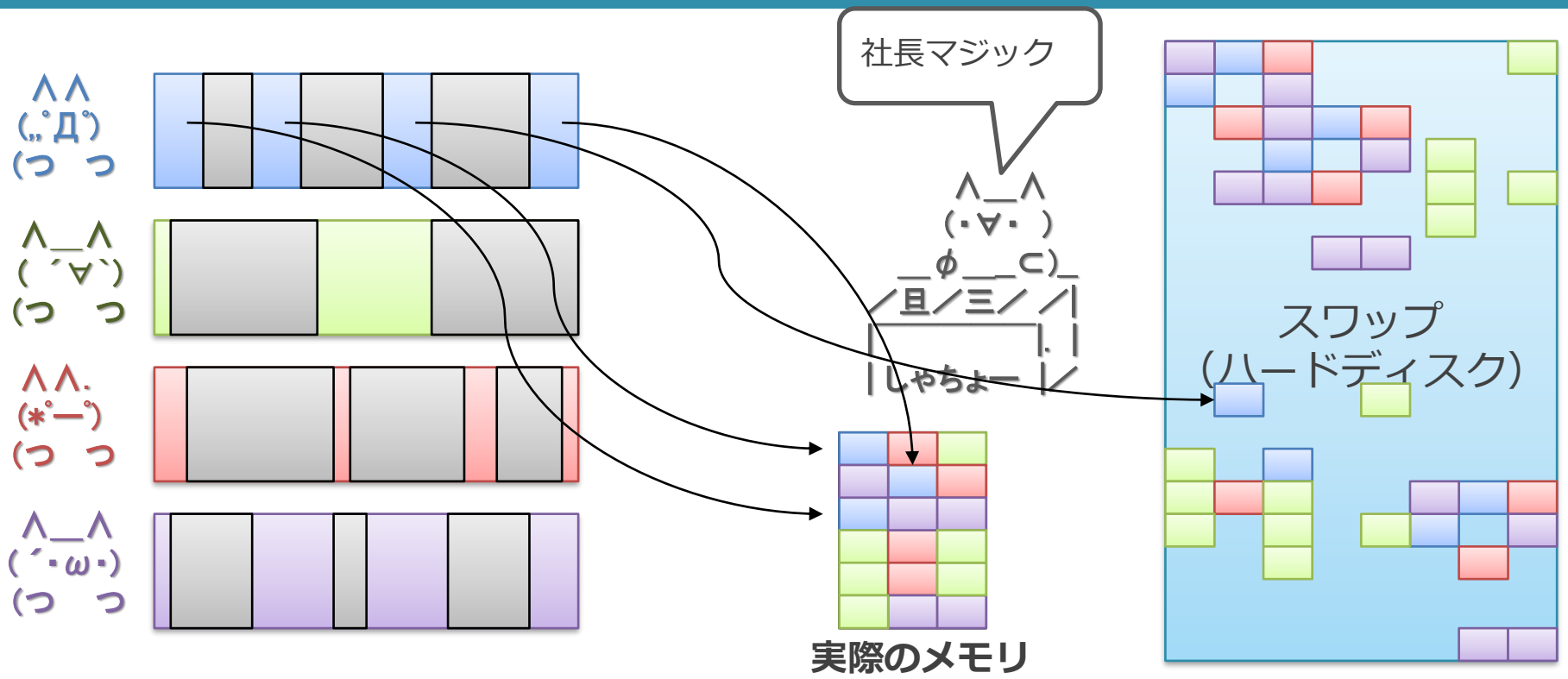
- 特に OS の管理領域がバグで誤って破壊されると OS ごと落ちる
  - ◇ 管理領域をユーザーに勝手に見られるのもまずい

仮想メモリ



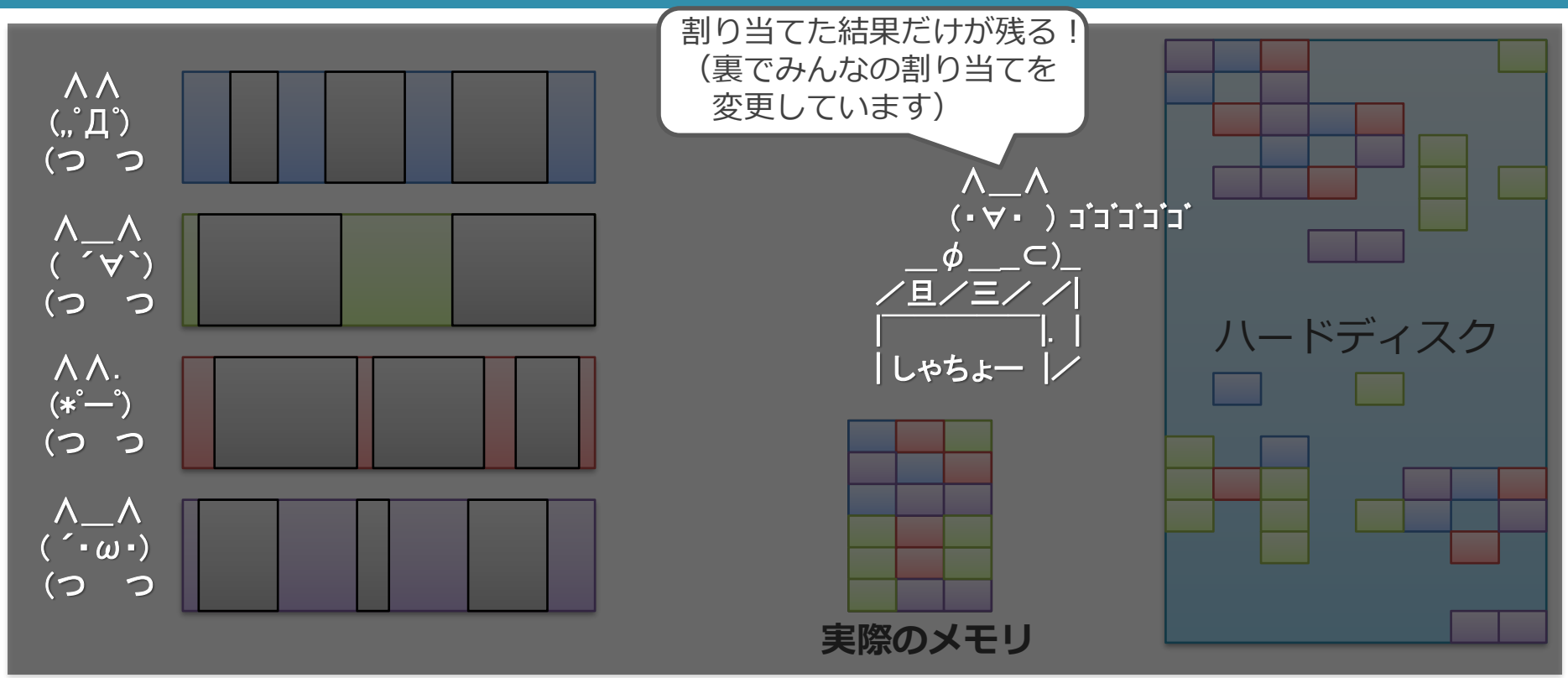
- プログラムごとに専用の大きなメモリが用意されているように「見せかける」技術
  - ◇ プログラムからは「自分専用の」メモリがあるかのように見える
  - ◇ アドレスで指定できる場所はすべて自分の空間
  - ◇ 他人の空間は読み書きできない

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大きなハードディスクにマップされる
    - □ これを「スワップ領域」という
    - □ 実際の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

マップの更新は透過的に行われる



- 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
  - ◇ 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，通常認識できない
- プログラムの実行を止めて裏で再割当てを行う
  - ◇ 実際には全員を止める訳ではなく，関係者だけ止める

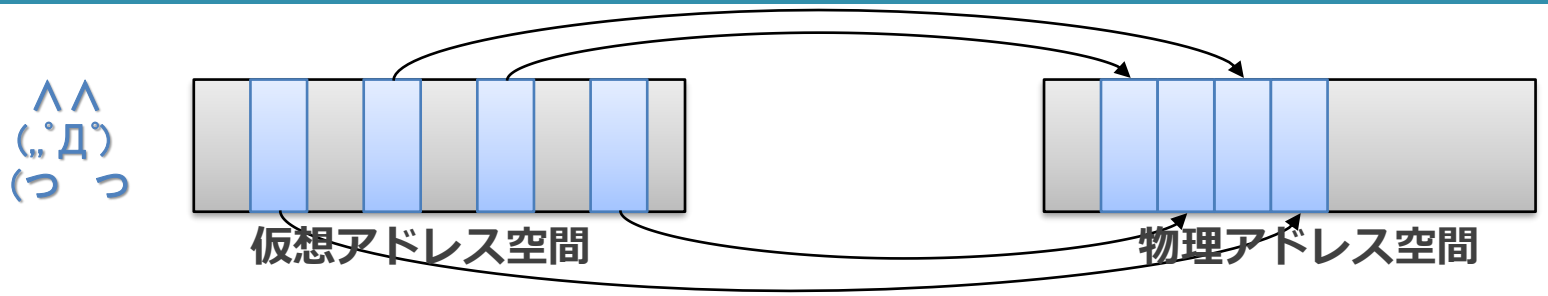
仮想メモリの基本のまとめ

- 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

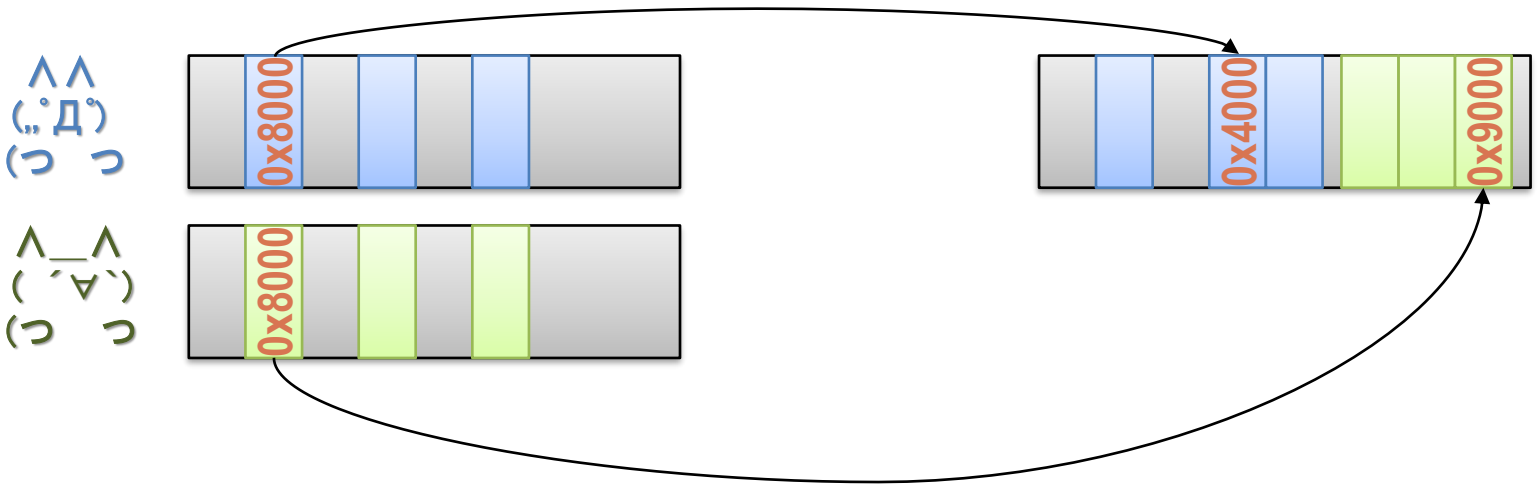
1. 仮想メモリの詳細
    1. 仮想アドレスと物理アドレス
    2. ページ・テーブル
    3. TLB
    4. キャッシュ・アクセスとの関係

仮想アドレスと物理アドレス



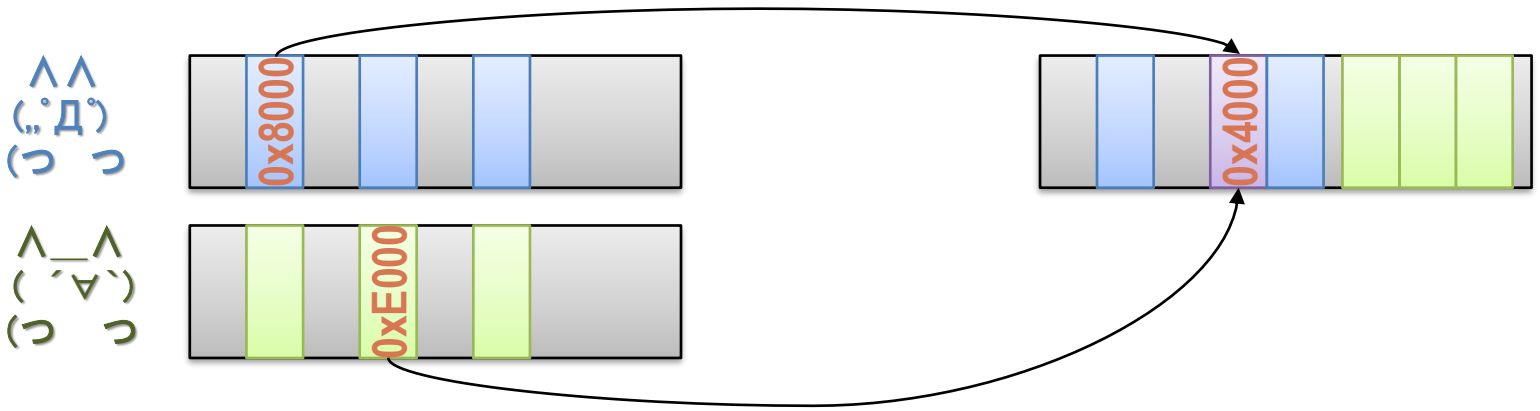
- 仮想アドレス（論理アドレスともいう）
  - ◇ プログラムから見えるアドレス
  - ◇ C 言語などでポインタに入っているのはこれ
- 物理アドレス
  - ◇ 物理的なメイン・メモリのアドレス
  - ◇ プログラマからは見えない
- メモリ・アクセス時は，毎回仮想アドレスから物理アドレスに変換してアクセスする

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
- 正確にはプロセスごと

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

変換の実装

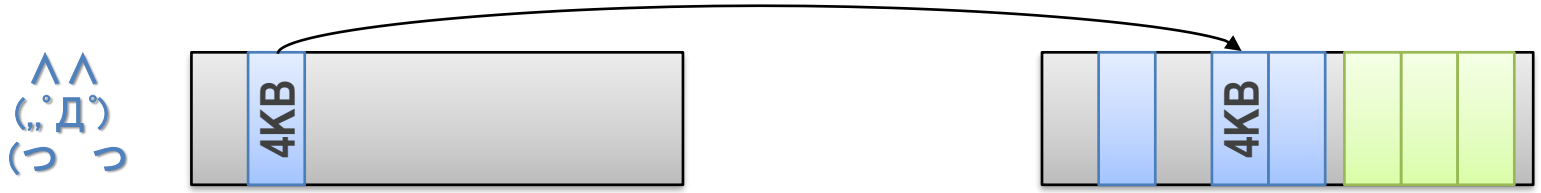
- 単純な実装
  - ◇ 仮想アドレス → 物理アドレス の変換表を用意すれば良い
- コスト：アドレスが 32 bit だとして，1 byte 単位で表を作ると…
  - ◇ データ 1 byte ごとに，その表には 32 bit = 4 byte のアドレスを記録することになる
  - ◇ 変換表に実データの 4 倍の容量がいる

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
    4. キャッシュ・アクセスとの関係

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で決まっている
- 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）



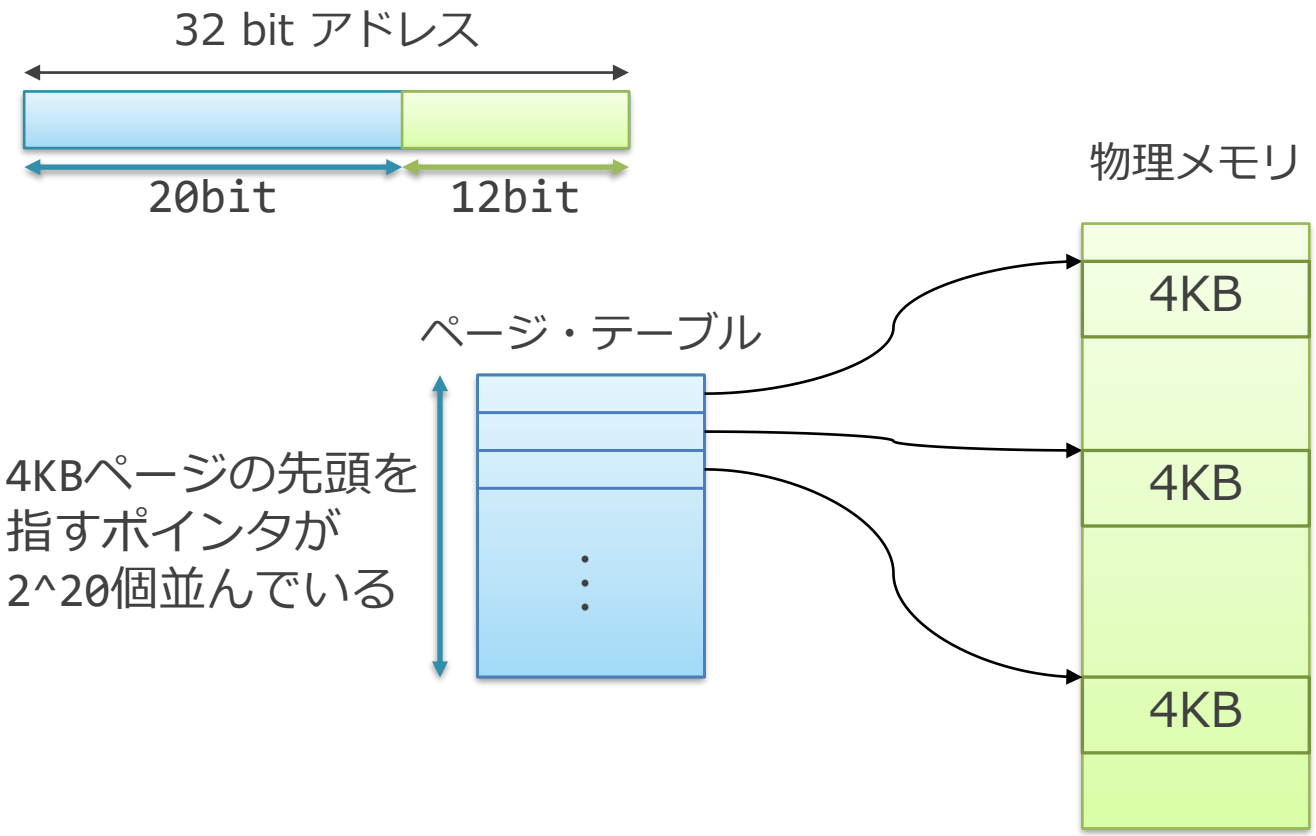
ページ・テーブルの管理

- 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・テーブルの更新はソフト（OS）が行う
  - 全部ハードでやると大変
  - OS ごとにどのようにマップしたいかも異なるし，柔軟に対応したい

多段ページ・テーブル

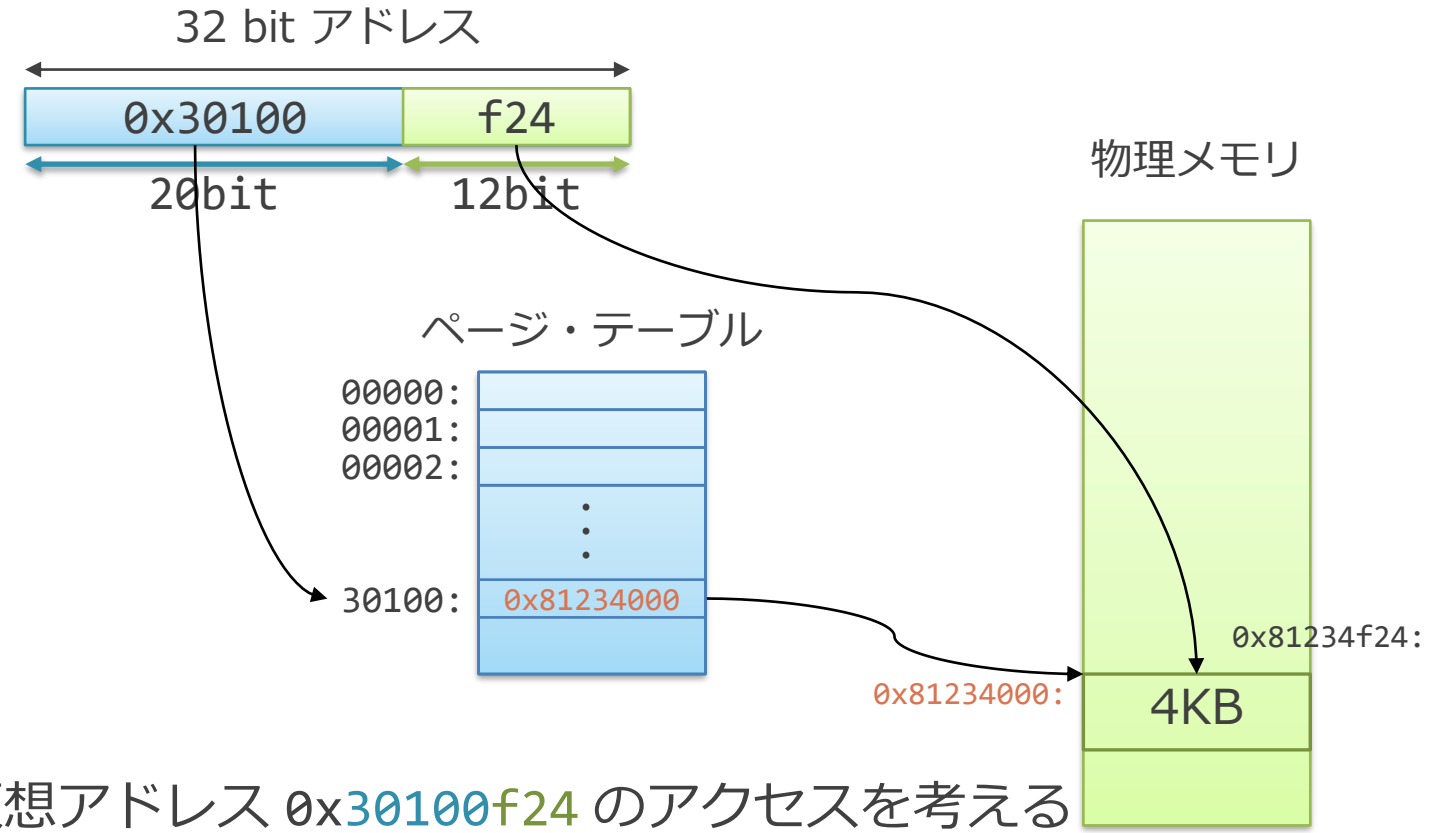
- ページ単位で管理したとしても，なおページ・テーブルは大きい
  - 64 bit のアドレス空間で，ページ・サイズを 4KB とした場合，
  - （アドレスの個数）/（ページ・サイズ）*（アドレスのサイズ） = （2^64）/ 4KB * 64bit = 16EB / 4KB * 8B = 32PB
    - たとえ 1B しかメモリを使わないプログラムでも 32PB が必要に
- 多段ページ・テーブルと呼ぶ構造で効率良く保持する
  - プログラムで使うメモリ容量に比例した程度の容量でページ・テーブルを作る方法

単段ページ・テーブル



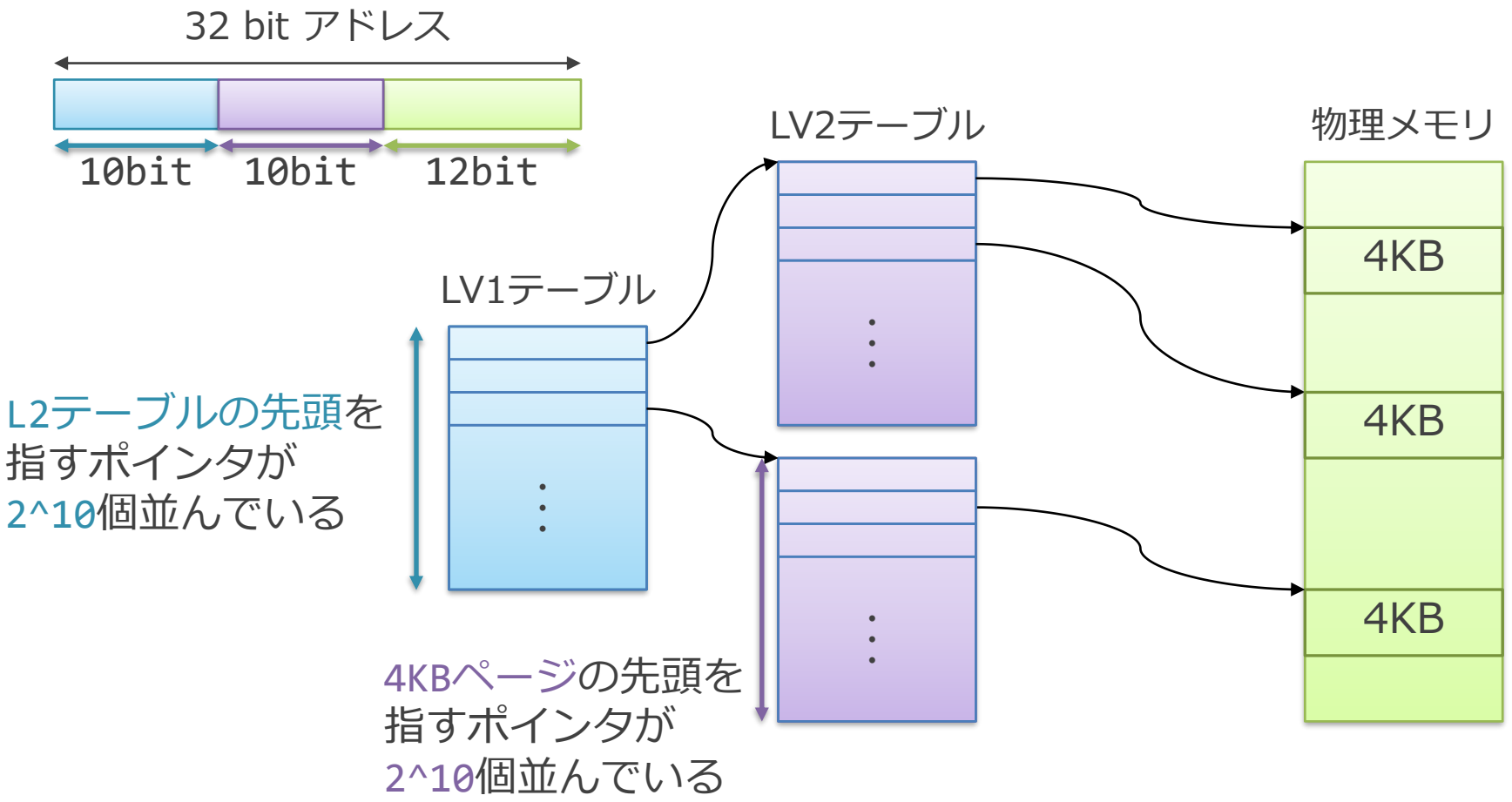
- まず単段のページ・テーブルを考える
  - ◇ アドレス・サイズが 32 bit, ページ・サイズ 2^12=4 KB を仮定

単段ページ・テーブルの動作



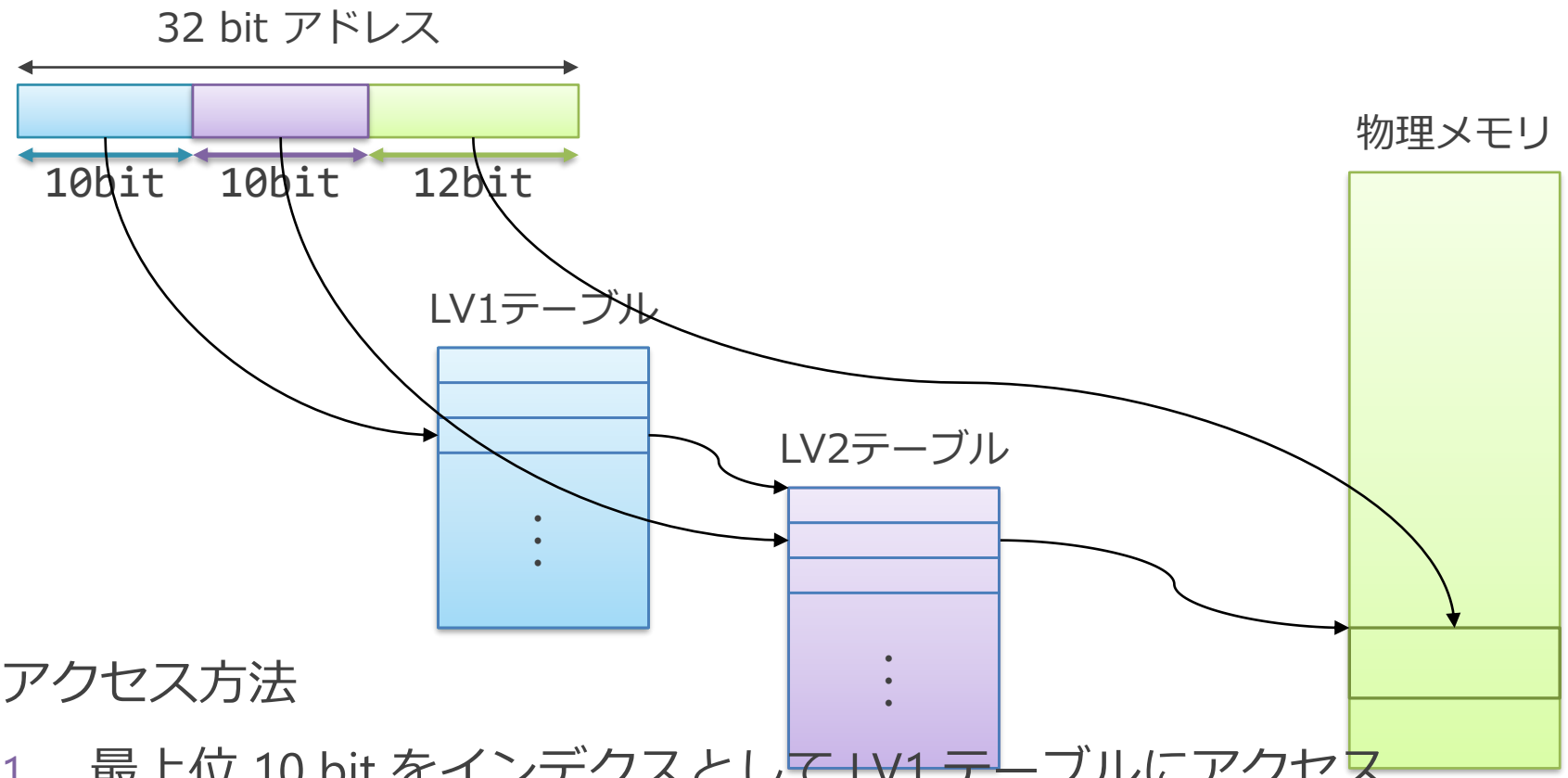
- 仮想アドレス 0x30100f24 のアクセスを考える
  - 上位 20 bit である 30100 を取り出し，それをインデクスとしてページ・テーブルにアクセス
  - 割り当てられたページの先頭の物理アドレス 0x81234000 を得る
  - 下位 12bit である f24 と結合して 0x81234f24 にアクセス

2段ページ・テーブル



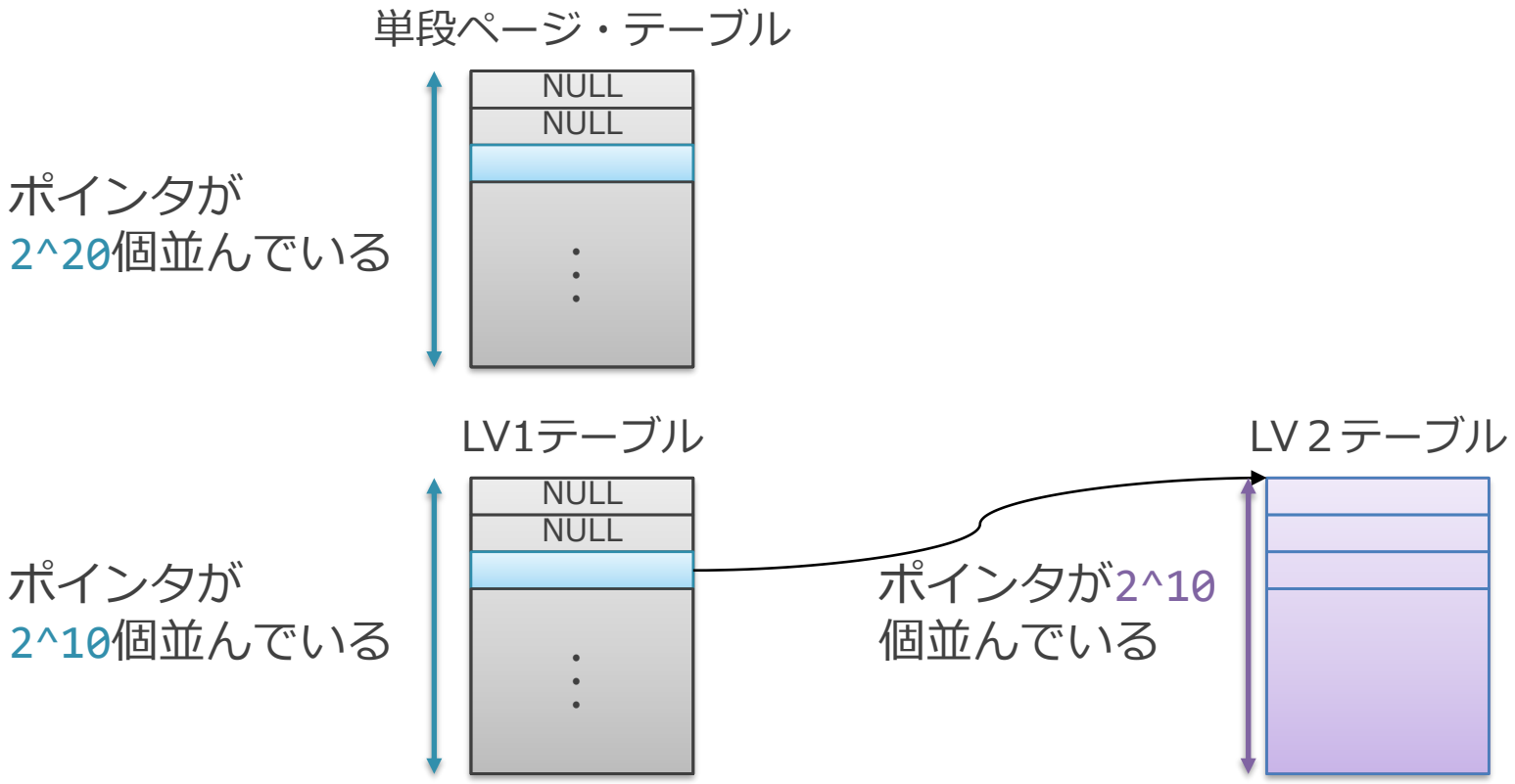
- 複数段のテーブルを経て物理メモリにアクセス
  - ◇ LV1テーブル → LV2テーブル → 物理メモリ

2段ページ・テーブルのアクセス



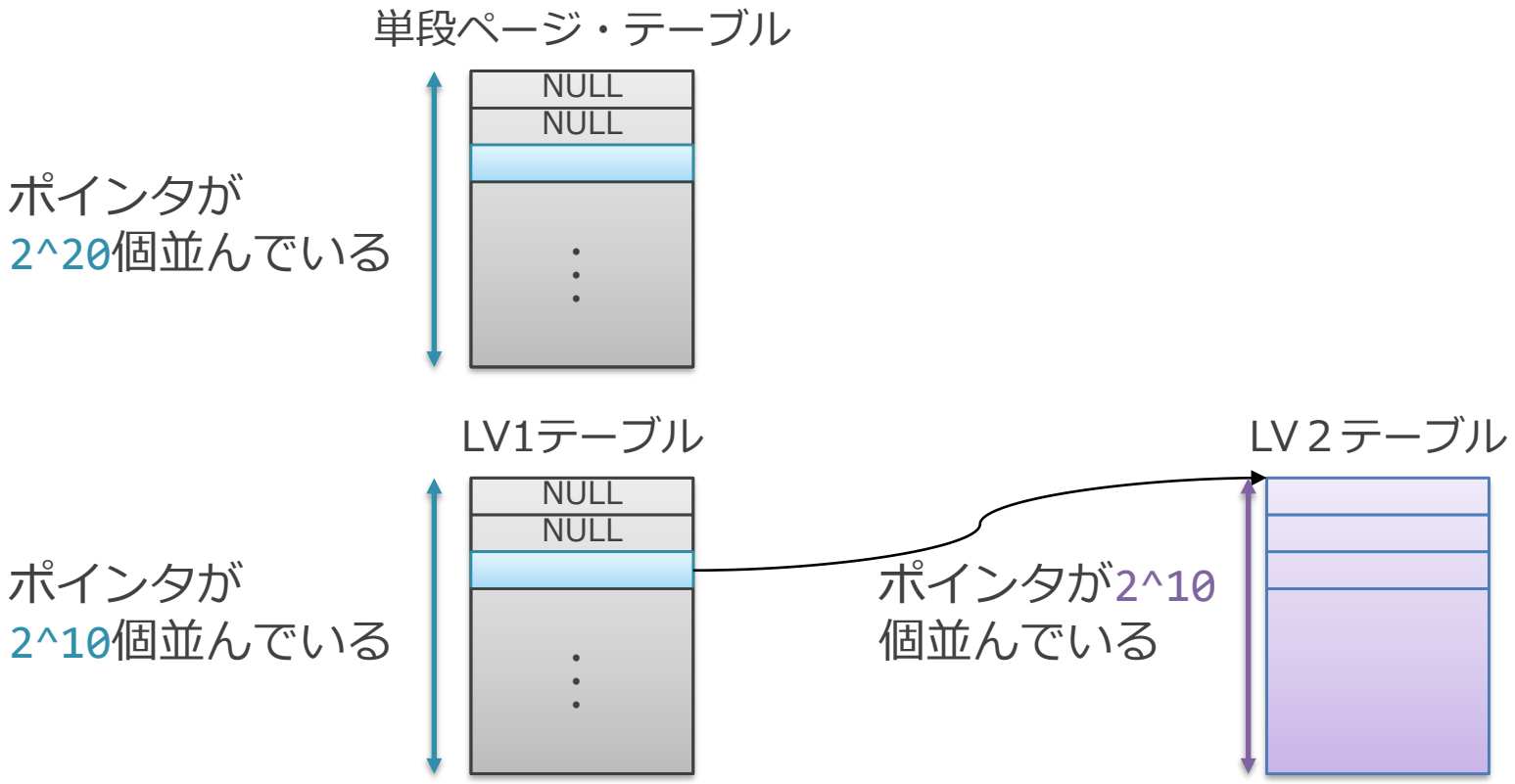
- アクセス方法
  1. 最上位 10 bit をインデクスとして LV1 テーブルにアクセス
  2. 次の 10bit をインデクスとして，1. で得られた LV2 テーブルの先頭アドレスにアクセス
  3. 最下位 12 bit をインデクスとして得られた物理メモリ上の 4KB 領域にアクセス

2段ページの利点：必要な容量が少ない



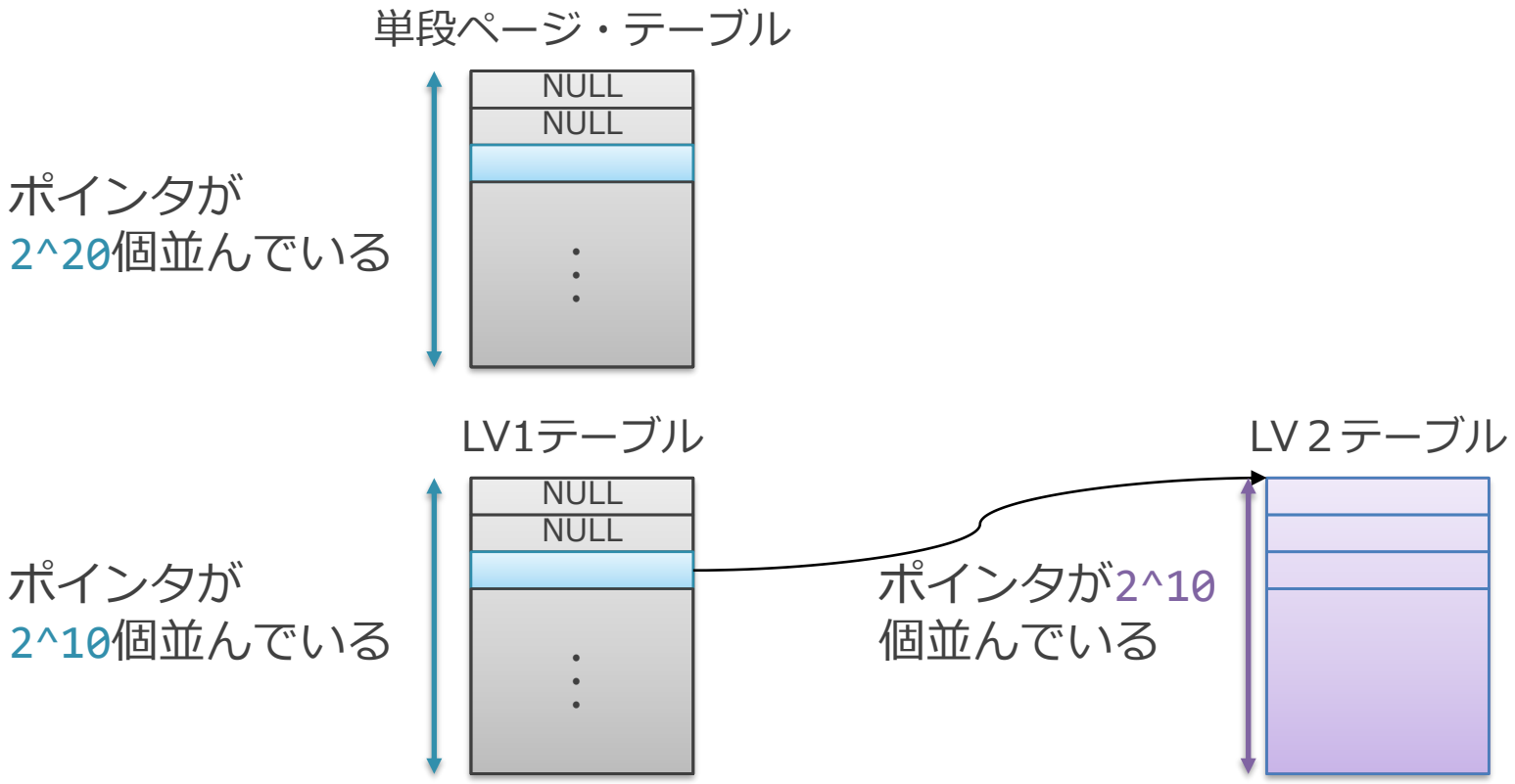
- 確保された領域に対応するエントリのみ，有効なポインタが入る
  - ◇ 未確保の領域は無効なポインタが入る

2段ページの利点：必要な容量が少ない



- 単段と 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=1M 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・テーブル自体も物理メモリ上に存在
  - LV1テーブルの先頭のアドレスを CPU の特別なレジスタに設定
    - そこが LV1 テーブルだと CPU に認識される
    - 中身は OS が書き込む
  - プロセスを切り替える際は，このレジスタを再設定する
- 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

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**
    4. キャッシュ・アクセスとの関係

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
    4. **キャッシュ・アクセスとの関係**

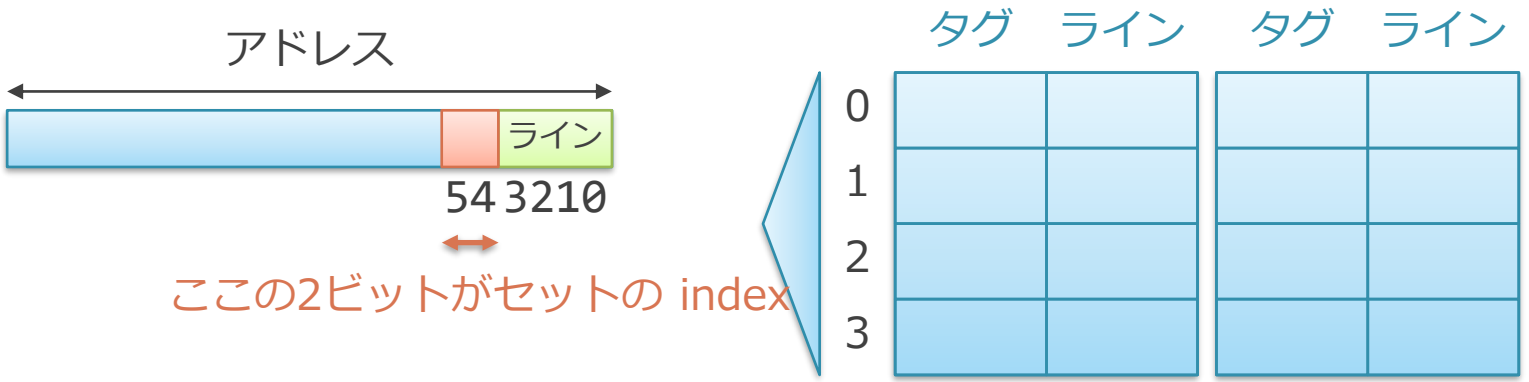
L1 キャッシュとの関係

- キャッシュは通常は物理アドレスでアクセスされる
  - ◇ 複数の仮想アドレスが同一の物理アドレスを指していることがあるから
  - ◇ 論理アドレスでキャッシュを作ると，プログラムを切り替えた時に中身を捨てなければいけないというのもある
- ロード時のレイテンシが伸びる
  - ◇ アドレス計算 → TLB アクセス → L1キャッシュ

仮想インデクス物理タグ方式
Virtually-indexed physically-tagged: VIPT

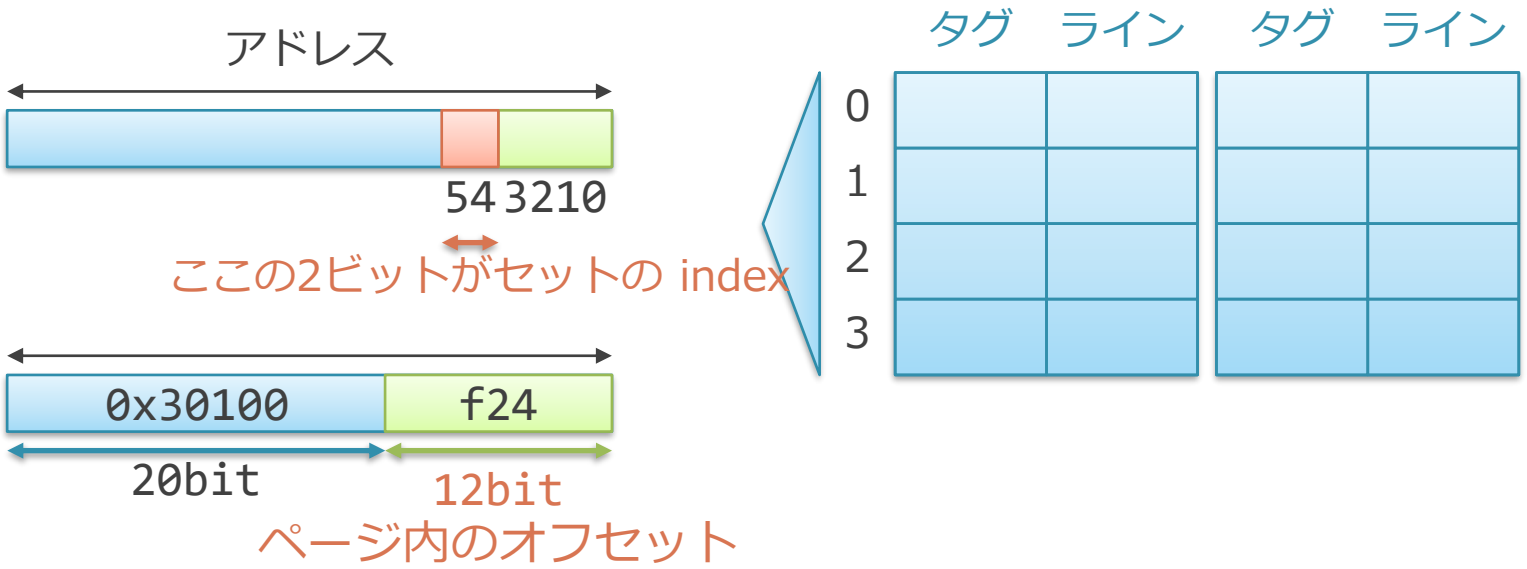
- TLB アクセスを介さずに L1 キャッシュへのアクセスを開始する方法
  - ◇ L1 キャッシュのインデクスを仮想アドレスでひいて，
  - ◇ 出てきたタグを TLB から得られた物理アドレスで比較する

アドレスとセットの対応（復習）



- セットのインデクスは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

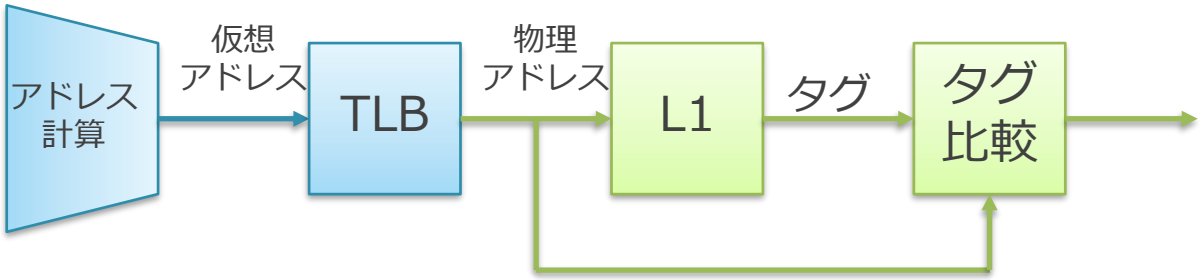
ページ内オフセットとセットの index



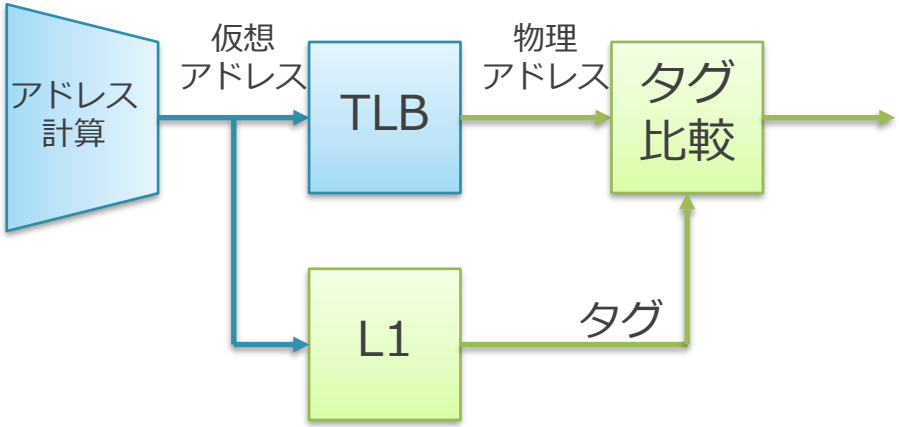
- 4KB ページの場合，ページ内のアドレス（下位12bit）は仮想でも物理でも同じ
  - ◇ セットの位置を決定している部分はここに含まれている
  - ◇ つまり，仮想アドレスを使ってセットの index を決定しても結果は同じ

仮想インデクス物理タグ方式

直列に変換を
行う場合

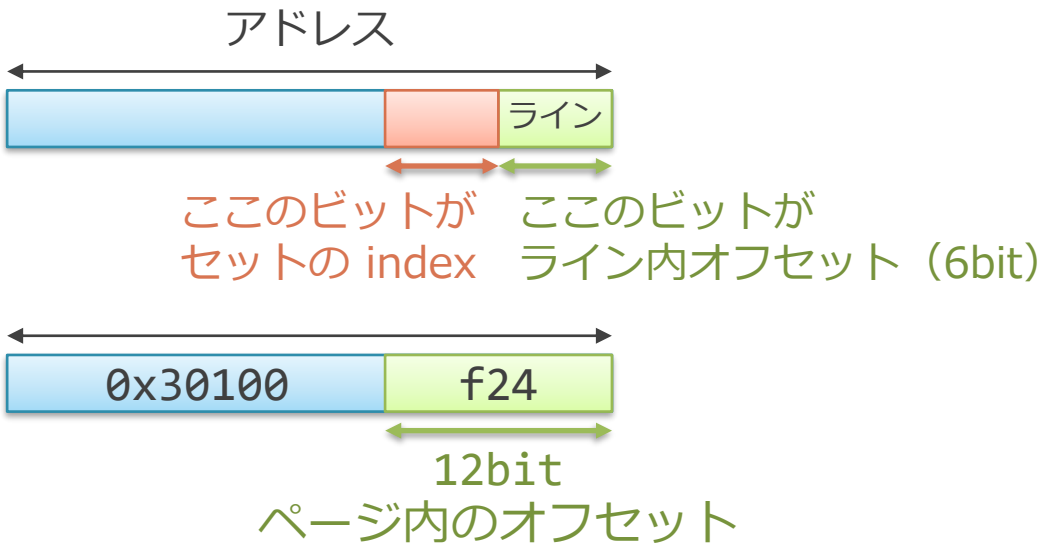


仮想インデクス
物理タグ方式



- TLB アクセスを介さずに L1 キャッシュへのアクセスを開始

仮想インデクス物理タグ方式の問題



- セット個数の最大数が大きく取れない
  - ◇ 4KB ページの場合，ページ内オフセットは 12bit
  - ◇ 64B ラインの場合，そこに 2^6=64 より 6bit とられる
  - ◇ つまり 12-6=6 bit 分 = 64 セットしか使えない

仮想インデクス物理タグ方式の問題

- way 数を増やす方向でしか L1 キャッシュ容量を増やせない
  - 12-6=6 bit 分 = 64 セットしか使えない
    - たとえば 64セット × 64Bライン × 8way = 32KB
- L1 キャッシュが 32KB 程度となっているのは，このことが大きい
  - L2 以降のキャッシュは変換後の物理アドレスでアクセスするので問題とならない
- これ以上 way 数を増やすのはなかなかつらい
  - 比較器の個数が増えるので遅延が問題になってくる
  - それでも最近は 48KB～64KB の構成がでてきた

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
- 仮想インデクス物理タグ方式
  - ◇ 物理アドレスへの変換を行わずに L1 キャッシュへアクセスする方法

今日の内容

1. 保護機構
    1. 仮想メモリ
    2. **特権モード**
2. 脆弱性とアタック
    1. バッファ・オーバーフロー
    2. Return Oriented Programming
    3. マイクロアーキテクチャ面の脆弱性

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動作モード
  - ユーザー・モード
    - 通常のプログラムが動くモード
    - ページ・テーブルの操作や外部デバイスへの操作は制限されている
  - 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 を介すため，無茶はできない

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
    - ジャンプ命令というよりは，割り込みに近い

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

（レジスタ番号とか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 に要求番号を設定
    - 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)
...
```

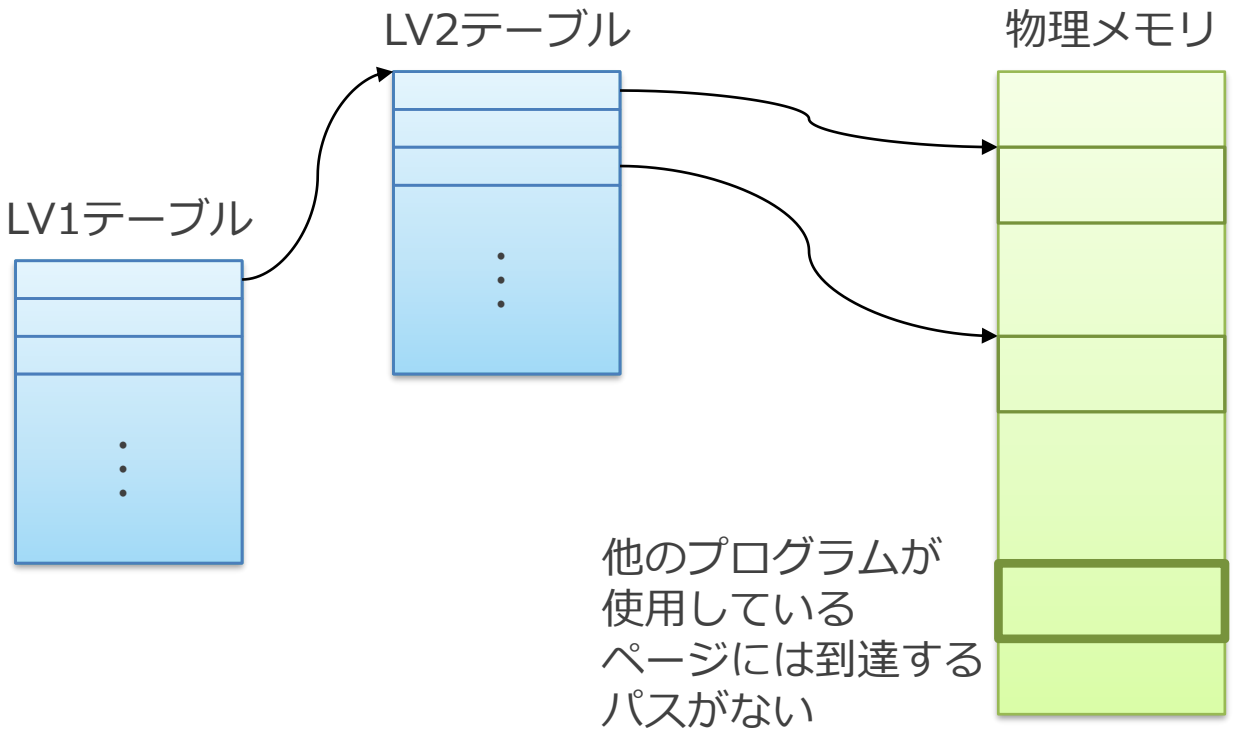
```
li x17 ← 1034  // システム・コール rename の要求番号を設定
ld x10 ← (...) // リネーム対象のファイル名が入っている文字列へのポインタをロード
ld x11 ← (...) // リネーム後のファイル名が入っている文字列へのポインタをロード
ecall          // システム・コール呼び出し．返り値は x10 に入る
```

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で強制的に行われる
  - ◇ 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

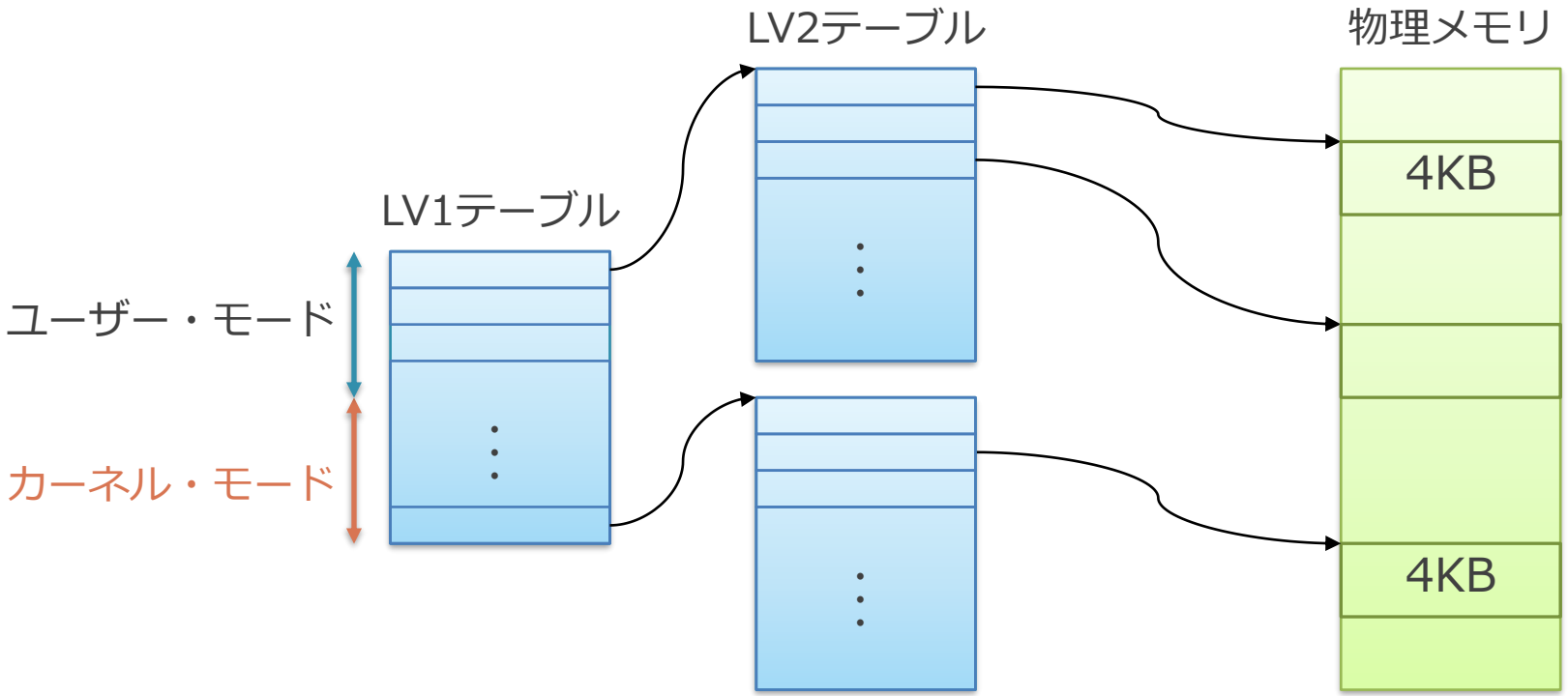


ページごとの保護

- 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も違反すると例外が起きる

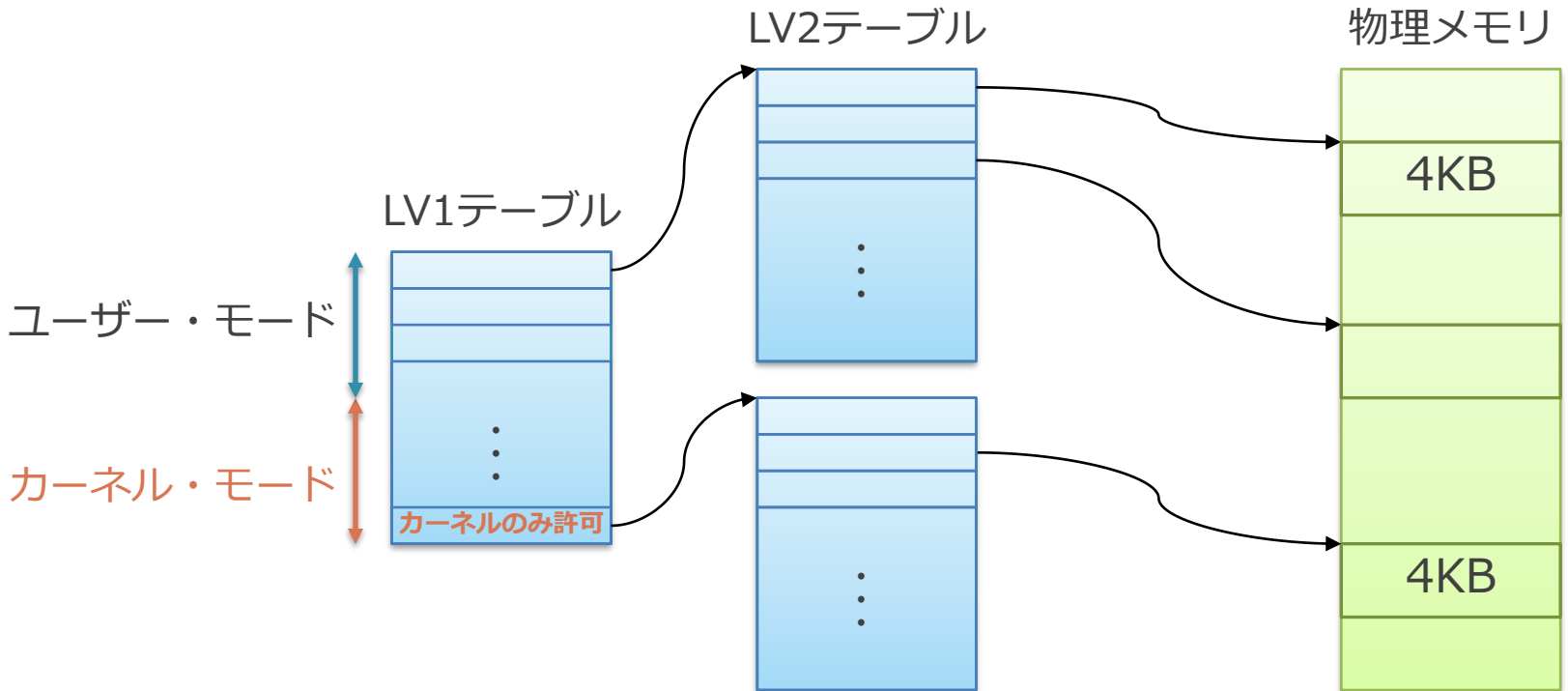
ページごとの保護を利用した仮想アドレスの共有による最適化

- 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

今日の内容

1. 保護機構
    1. 仮想メモリ
    2. 特権モード

2. **脆弱性とアタック**
    1. バッファ・オーバーフロー
    2. Return Oriented Programming
    3. マイクロアーキテクチャ面の脆弱性

脆弱性

- 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
  - ◇ 仮想メモリや特権モードの働きによる
- ところが，ソフトウェアに問題があるとこれを突破できてしまう
  - ◇ そのようなセキュリティ上の問題のことを脆弱性という
- いくつかの代表的な脆弱性やそれを利用した攻撃を紹介
  1. バッファ・オーバーフロー
  2. Return Oriented Programming
  3. マイクロアーキテクチャ面の脆弱性

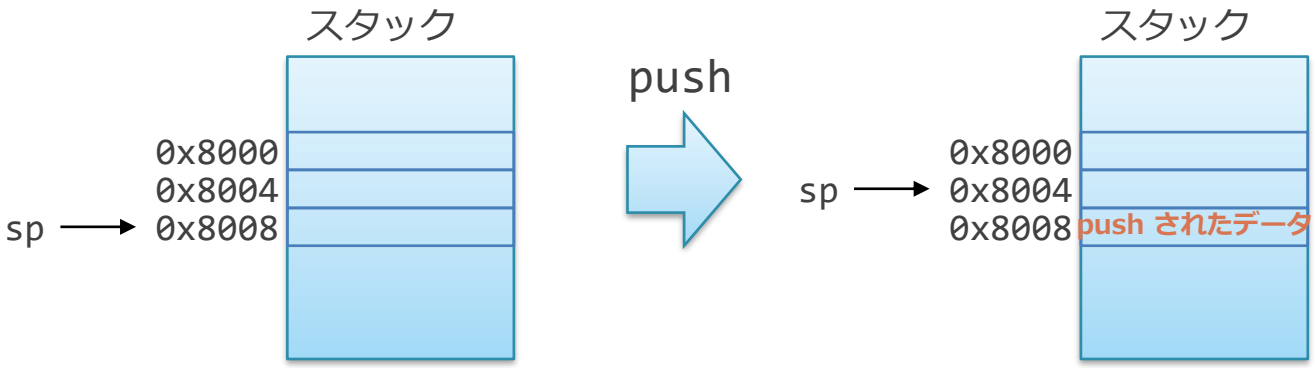
今日の内容

1. 脆弱性とアタック
    1. **バッファ・オーバーフロー**
    2. Return Oriented Programming
    3. マイクロアーキテクチャ面の脆弱性

バッファ・オーバーフロー脆弱性

- 配列の境界値チェックの忘れなどで生じる
  - 典型的なもの：
    - ```
      // src を size バイト分だけ buf にコピー
      // buf の容量を超えてコピーが行われる可能性がある
      void func(uint8_t* src, size_t size){
          uint8_t buf[8];
          memcpy(buf, src, size);
      }
      ```
- もし上記の src と size を外部から与えることが出来れば，最悪コンピュータを乗っ取ることができる
  - ネットワークごしでなくても，画像ファイルを送って読ませるとかでも良い
      ```

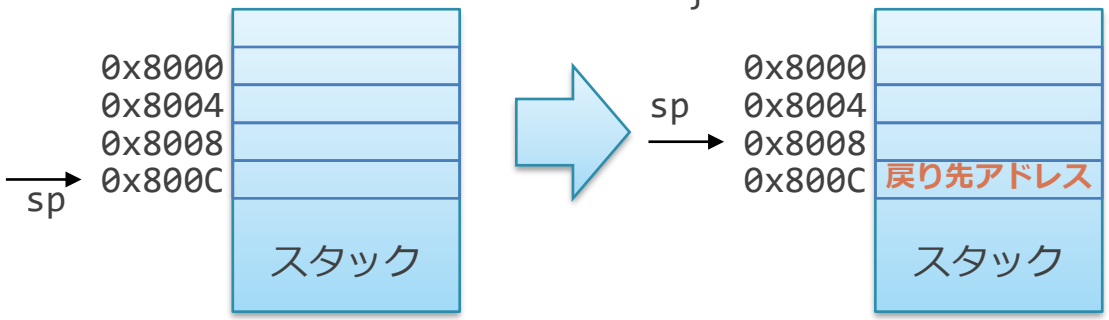
背景：C 言語の関数呼び出しの仕組み



- 関数呼び出しはメモリ上のスタック構造を使って実現
  - ◇ 特定のレジスタ sp（RISC-V では x2）がスタックのトップをさしている
  - ◇ データを push するときは sp を減少（図上で上に移動）
  - ◇ pop するときは sp を増加（図上で下に移動）

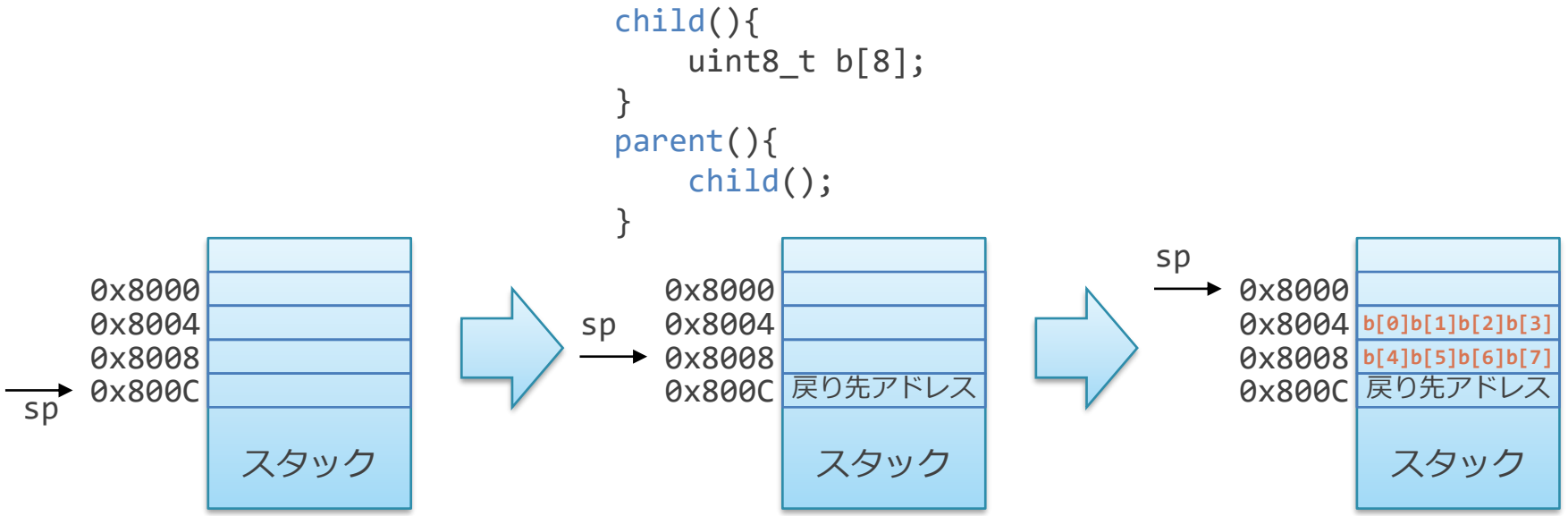
背景：C 言語の関数呼び出しの仕組み

```
child(){
    uint8_t b[8];
}
parent(){
    child();
}
```



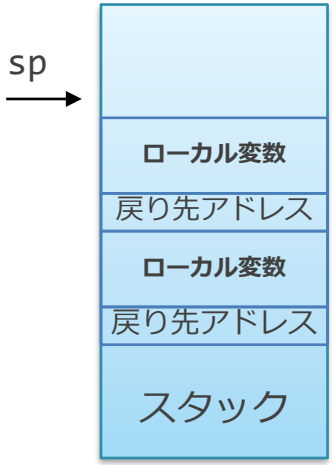
- parent() から child() を呼び出す場合
  1. 後で戻ってこれるよう，parent 内の戻り先を push
  2. child 先頭にジャンプ

背景：C 言語の関数呼び出しの仕組み



- parent() から child() を呼び出す場合（続き）
  3. スタック上にローカル変数の領域を確保 = sp を減らす
     - ローカル変数はその時の sp からの相対オフセットでアクセス
       - b[4] なら上の場合，sp+8=0x8000+8=0x8008
     - これにより関数呼び出しごとに領域が確保される

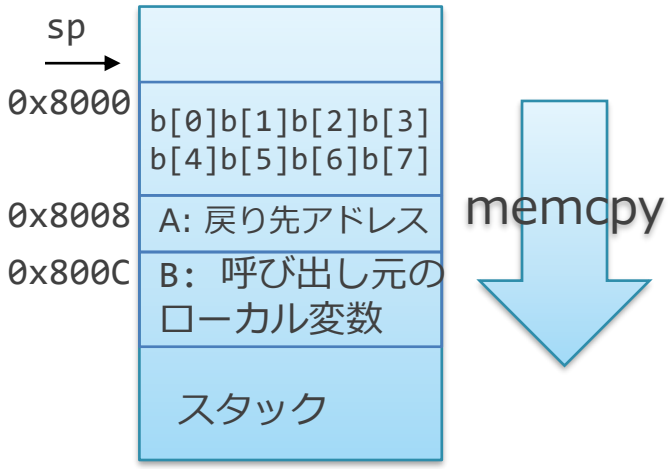
背景：C 言語の関数呼び出しの仕組み



- 関数内で関数を呼び出すと，どんどん上方向（アドレスが小さい方向に）に伸びていく
  - ◇ 各関数呼び出しの戻り先アドレスとローカル変数がサンドイッチされた構造になる
- （全体に，実際にはもうちょっとややこしいが簡略化している

バッファ・オーバーフロー

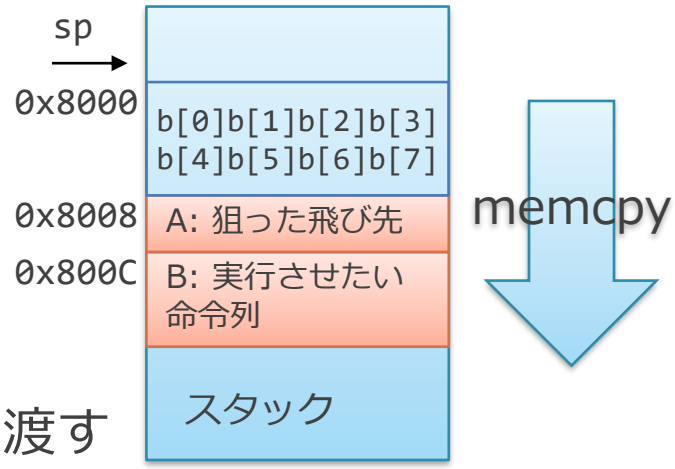
```
func(uint8_t* src, size_t size){
    uint8_t b[8];
    memcpy(b, src, size);
}
```



- size に 8 より大きな値を渡した場合，b の領域を超えて src の内容により下記が上書きされる
  - ◇ A: 戻り先アドレス
  - ◇ B: 呼び出し元のローカル変数

バッファ・オーバーフローによる任意コード実行

```
func(uint8_t* src, size_t size){
    uint8_t b[8];
    memcpy(b, src, size);
}
```



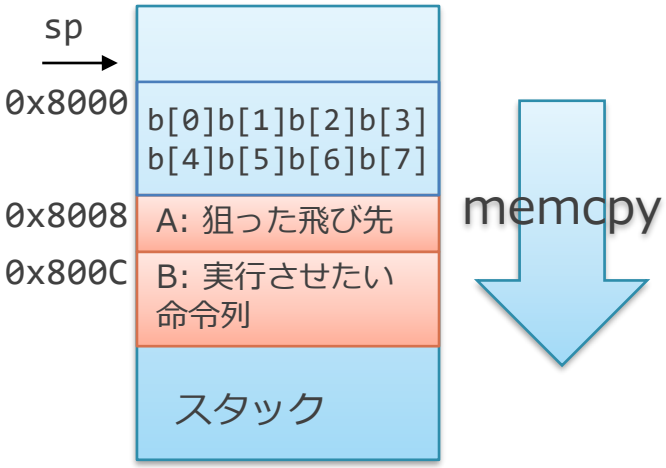
- 下記のように上書きされるよう src を渡す
  - ◇ A: 自分が飛ばせたい先のアドレス
    - □ ここでは B: の先頭 = 0x8008
    - □ 関数から抜けるときの return でここに飛ぶようになる
  - ◇ B: 自分が実行させたい命令列
    - □ ここから外部プログラムを起動するシステムコールを呼ぶ
    - □ bash などのシェルをコマンド付きで起動すればもうなんでもできる

バッファ・オーバーフローによる任意コード実行

- 「自分が飛ばせたい先のアドレス」は決め打ちなのか？
  - これは決め打ちになる
  - 対策されていないプログラムの場合，sp の初期値は固定なので予想できる
- 特権モードの「権限の昇格」にも使える
  - カーネル内でオーバーフローをおこせればカーネル・モードで任意のコードが実行できる
  - スマホやゲーム機ではユーザーはカーネル・モードになれないが，それを突破するのなんかにもよく使われる

バッファ・オーバーフロー脆弱性への対策

- ページごとの権限に「実行して良いか」を追加
  - スタック領域は実行不能に設定する
    - 通常はスタック上のデータを実行することはない
  - x86 では NX (no-execute) bit と呼ばれる
- 下記の場合，B に飛んだ瞬間に実行権限がページにないため落ちる



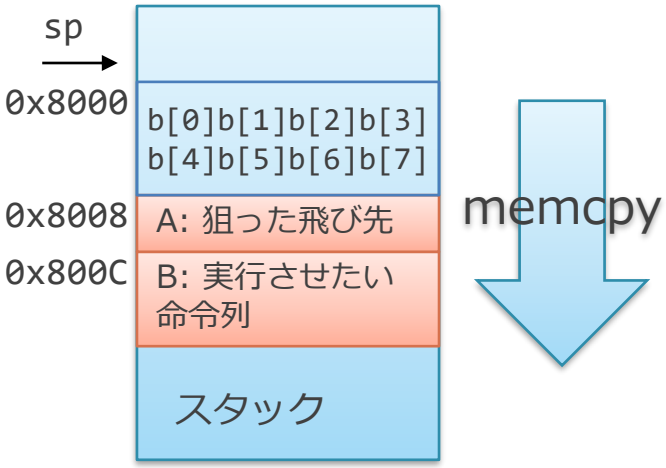
今日の内容

1. 脆弱性とアタック
    1. バッファ・オーバーフロー
    2. **Return Oriented Programming**
    3. マイクロアーキテクチャ面の脆弱性

ページに実行権限を設定した場合の動作

- 下記の場合，B に飛んだ瞬間に実行権限がページにないため落ちる
  - ◇ A や B の上書き自体は防いでいない
  - ◇ 狙った飛び先に飛ばすことだけなら出来る

- return-to-libc 攻撃
  - ◇ C 言語の標準関数等を狙って呼ぶことは可能
  - ◇ 引数を自由に設定できないので，大したことはできない



Return Oriented Programming (ROP)

- 関数の末尾をつなぎ合わせて任意の動作を行う方法
- プログラム内の様々な関数の末尾をみる
  - RISC-V に ret/pop はないが,
    それに相当する操作をしているものとする
- 使える部品（ガジェットと呼ぶ）が転がっている
  - 「スタックから値を取り出して x4 に入れる」
  - 「スタックから値を取り出して x6 に入れる」
  - 「x6 をアドレスとして読んで x7 に入れる」

```
// 関数Aの末尾
pop x4
...
ret

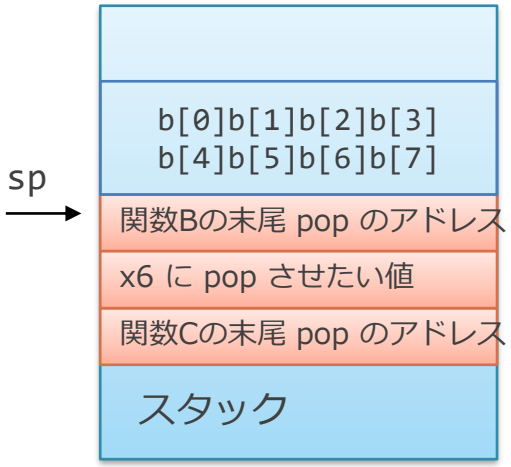
// 関数Bの末尾
pop x6
...
ret

// 関数Cの末尾
ld x7 ← (x6)
...
ret
```

Return Oriented Programming (ROP)

- ガジェットのアドレスと入力をオーバーフローで書きこむ
  - ◇ そして繰り返し return によりガジェットを実行させていく

- 下記の場合,
  1. オーバーフローが起きた関数の ret により sp が指す関数 B 末尾の pop x6 に飛ぶ
  2. pop x6 を実行して x6 にオーバーフローで書かれた値を読む
  3. ret で関数 C 末尾の `ld x7←(x6)` に飛ぶ




```
// 関数Aの末尾
pop x4
...
ret

// 関数Bの末尾
pop x6
...
ret

// 関数Cの末尾
ld x7 ← (x6)
...
ret
```

Return Oriented Programming (ROP)

- 同様の原理でループや分岐も実現可能・・・らしい
- プログラムを解析しガジェットを集めて任意の動作を可能にするコンパイラ？が開発されている

ROP（だけじゃないけど）に有効な対策

- Address Space Layout Randomization (ASLR)
  - ◇ スタックやプログラムのコードが配置される位置を起動ごとに毎回ランダムに変化させる
  - ◇ ROP やその他の攻撃はガジェットのアドレスがわかっていないと適用できない
  - ◇ 既にこれは Windows や Linux で広く使われている

ROP（だけじゃないけど）に有効な対策

- バッファ・オーバーフローによる攻撃の起点は，外部から命令のアドレスを書き込むこと
- 対策：アドレスに署名をつける（Pointer Authentication）
  - 仮想アドレスは 64 bit 全ては使っておらず 40 bit ぐらい
    - ページ・テーブルを小さくするため
  - あいている上位ビット部分にポインタのハッシュ値をいれておく
    - ハッシュ関数が未知なら外部から正しいものは書き込めない
  - 最近の ARM に導入された
  - RISC-V にも拡張が提案されている

今日の内容

1. 脆弱性とアタック
    1. バッファ・オーバーフロー
    2. Return Oriented Programming
    3. **マイクロアーキテクチャ面の脆弱性**

マイクロアーキテクチャ面の脆弱性

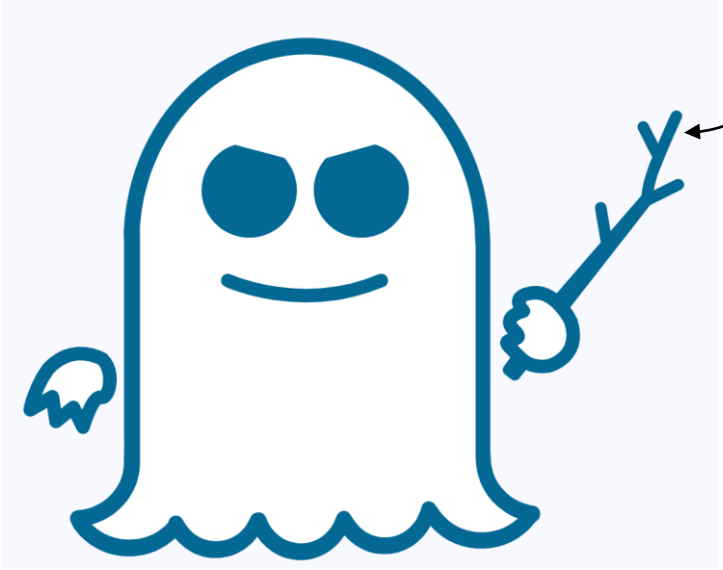
1. スレッド間で共有されたユニットを利用したもの
   - ◇ 例：分岐予測器を利用したもの
     - □ 共有された分岐予測器を観測して別のスレッドの分岐方向を盗み見る
     - □ 1bit ずつ結果が定まる RSA 暗号などは鍵や平文がもれる可能性がある

2. 投機実行による副作用を利用したもの
   1. Spectre/Meltdown

- ■ 時間がないので，2. についてざっと紹介

Spectre

画像は Spectre の公式マスコット（https://meltdownattack.com/ より）



- 2018 年に発表された CPU の投機実行に関わる脆弱性
  - この講義で話した色々な内容が関わる
  - 分岐予測，out-of-order 実行，セットアソシアティブ・キャッシュ，特権モード

Spectre

- いくつか異なるやりかたがある
  - Variant 1,2,3,4
  - Meltdown
- ここでは Variant 1 と 4 を紹介
  - １つわかれば他も大体わかると思う

Spectre Variant 1 の概要

- 分岐予測による投機実行の副作用を観測
  - ◇ 投機実行は通常の if 文などによるチェックに関係なく予測に従って行われる
  - ◇ 投機実行によりキャッシュの内もが置き換わる
  - ◇ その置き換え結果を観測して，タイミングから間接的に目的の値を得る

分岐予測

プログラム

```
if a > 0:
    a=*p
else:
    a=*q
```

- 動作
  - 「if a > 0」の結果を予測して，命令を取り込む
    - 前回はこっちに行ったので，次もこっちに違いないとかで予測
  - あとから予測が正しいか確認する
  - これまでの説明と違って，*p と *q のメモリを読んでいる

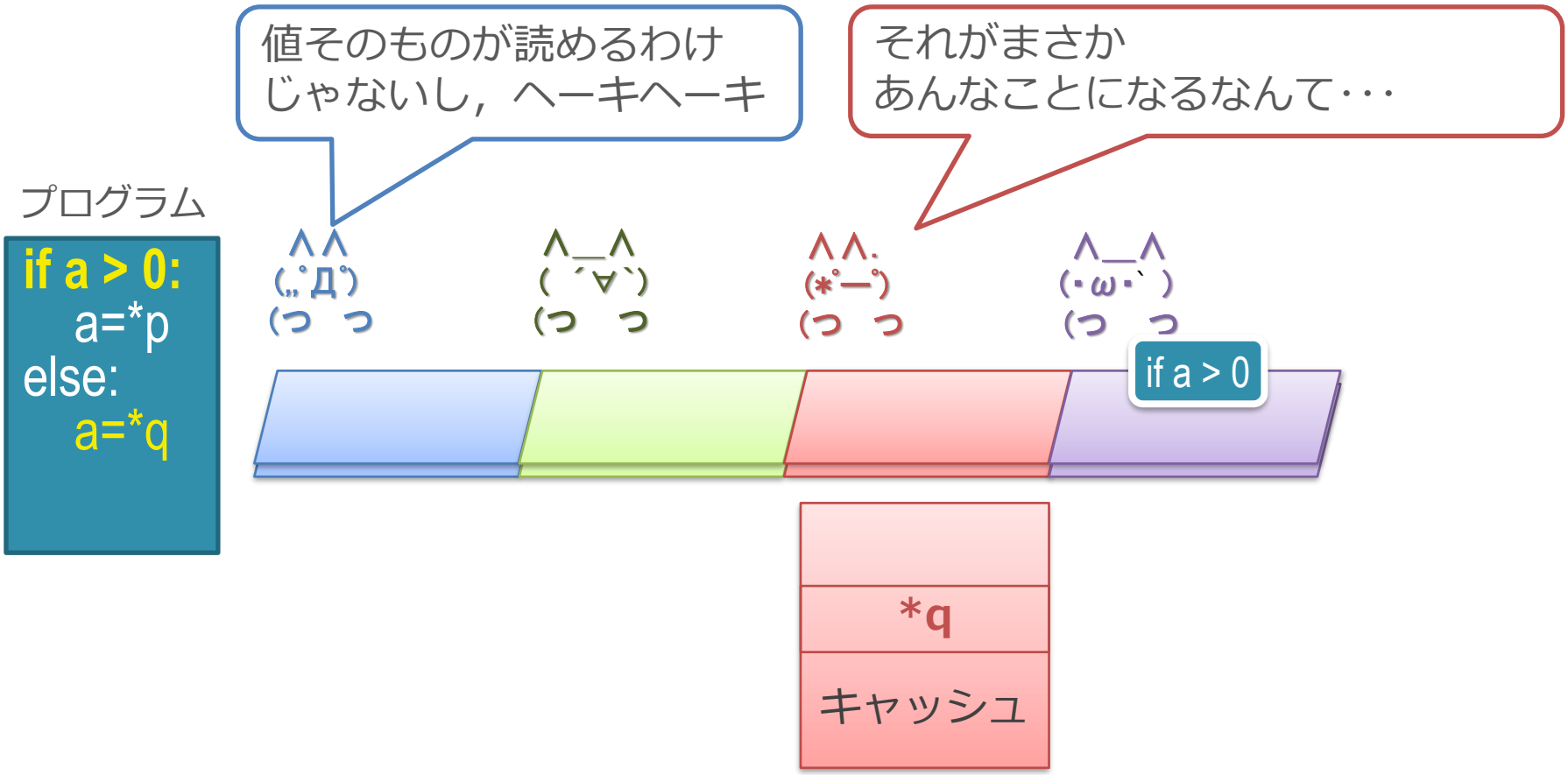
分岐予測ペナルティ

プログラム

```
if a > 0:
    a=*p
else:
    a=*q
```

- 予測が間違っていた場合，以降の処理を取り消してやり直す

投機実行の副作用が残っている



- a=*q そのものの命令は取り消された･･･
  - ◇ しかし，キャッシュには *q の値が既に読み込まれ残っている
  - ◇ これを観測して投機実行中に行われた演算の結果を得る

前提

```
if (x < array1_size)
    y = array2[array1[x] * 64];
```

- 前提：
  - 上記のコードがシステム・コール内にあったとする
  - ユーザー・モードからは x をシステム・コールの引数として渡す
    - x は if で範囲チェックされているので，通常であれば array1 の有効なエントリしか読むことはできない
- 攻撃：
  - 上記を利用してカーネル・モード内の任意アドレスの値をユーザー・モードから読む

攻撃の手順

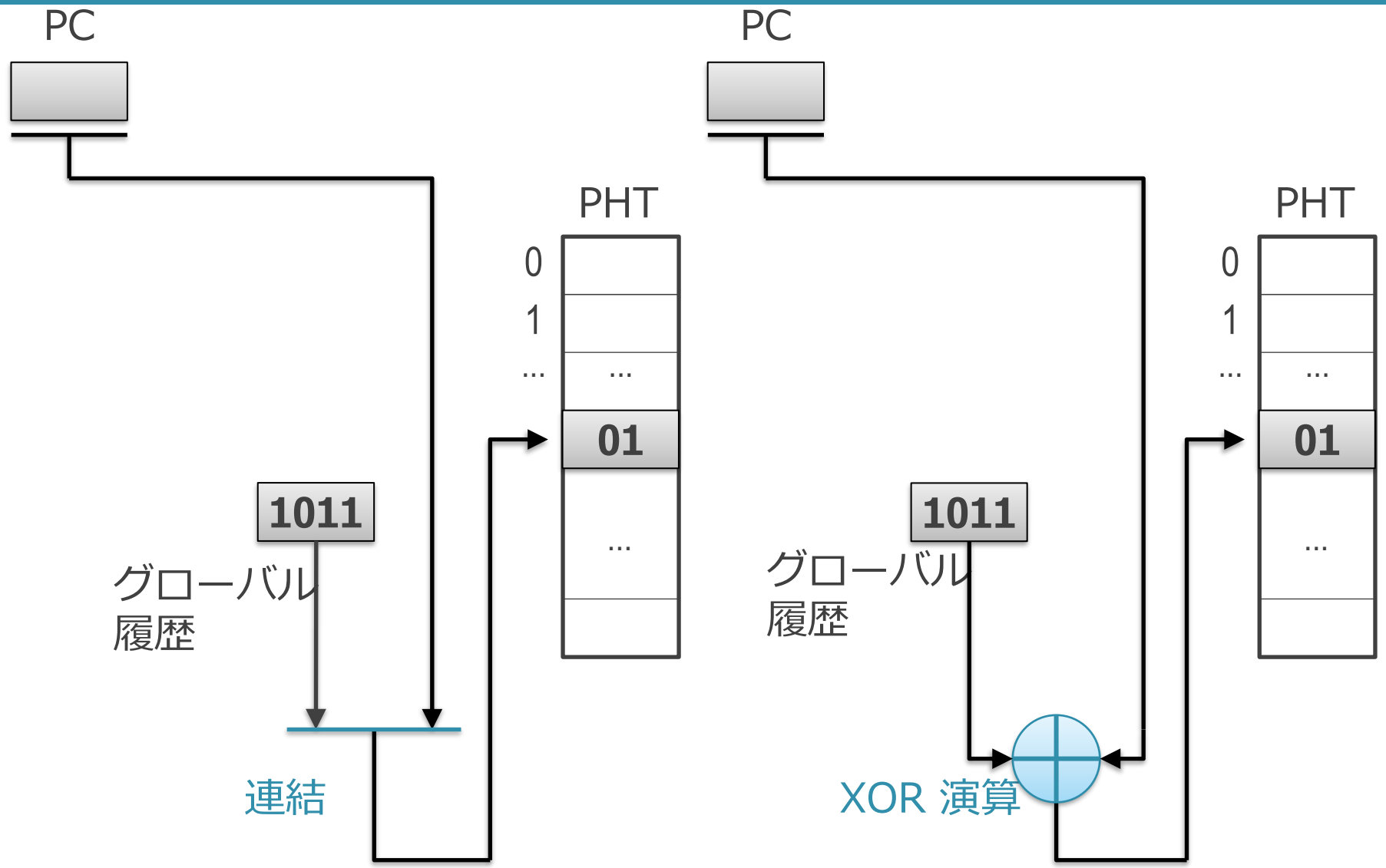
- 手順
  1. 分岐予測器の事前学習
  2. キャッシュの埋め尽くし
  3. 攻撃対象コードの実行
  4. キャッシュの観測による値の取得

分岐予測器の事前学習

```
if (x < array1_size)
    y = array2[array1[x] * 64];
```

- 上記の２行目が投機実行されるよう分岐予測器を事前に学習する
  - ◇ 分岐予測器は PC の一部とグローバル履歴を使う
  - ◇ カーネル・モード内の上記分岐で予測に使う予測器のカウンタを事前に望みの値にしておけば良い

g-share 予測器



- ビットを単純に連結するかわりに，XOR 演算して結合

分岐予測器の事前学習

- `if (x < array1_size)`
  `y = array2[array1[x] * 64]; // 分岐成立時に実行とする`

- 色んな方法が考えられる
  - ◇ たとえば下記のような全て if が成立する文を敷き詰めて実行
  - ◇ やがて PHT 内の全カウンタが 3 になり，どこを予測しても成立になる

```
if (0 < 1){}
if (0 < 1){}
if (0 < 1){}
if (0 < 1){}
if (0 < 1){}
if (0 < 1){}
…
```

攻撃の手順

- 手順
  1. 分岐予測器の事前学習
  2. **キャッシュの埋め尽くし**
  3. 攻撃対象コードの実行
  4. キャッシュの観測による値の取得

キャッシュの埋め尽くし

1. 事前にキャッシュを用意したデータで埋めておく
2. 攻撃対象のコードを実行
3. 再度 1. のデータにアクセスしレイテンシを測定
   - ◇ ミスが発生してレイテンシが長いラインがあった場合，2. 内でのアクセスで追い出されたことがわかる

- 上記の方法は Prime & Probe とも呼ばれる

攻撃の手順

- 手順
  1. 分岐予測器の事前学習
  2. キャッシュの埋め尽くし
  3. **攻撃対象コードの実行**
  4. キャッシュの観測による値の取得

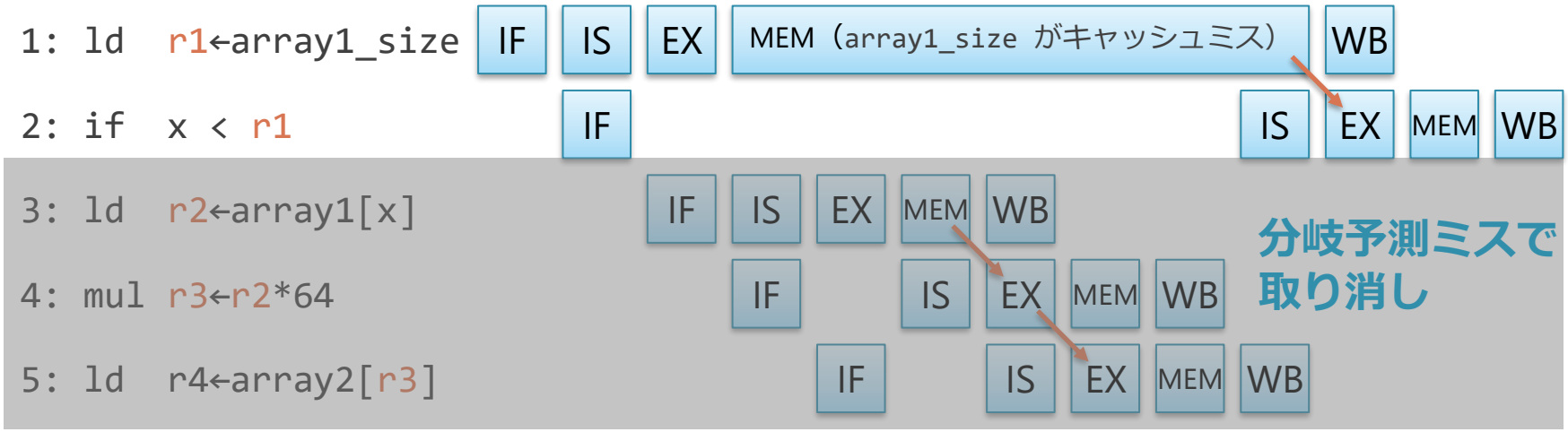
攻撃対象コードの実行

- `if (x < array1_size)`
  `y = array2[array1[x] * 64];`

- x に読みたい場所のオフセットを設定して呼び出す
  - ◇ `array1[x]` では `*(array1 + x)` のように 2 つの値の合計をアドレスとしてロードする
  - ◇ x を調整すれば任意のアドレスを作れる

攻撃対象コードの実行

```
if (x < array1_size)
    y = array2[array1[x] * 64];
```



- これまでの下準備により，
  1. 分岐予測器が成立と予測して３行目を投機的にフェッチ
  2. array1_size がキャッシュにないため，依存する分岐命令の実行が遅れる
  3. ３行目以降が out-of-order 実行で先に実行される
     - x は範囲外だが，投機的に実行されキャッシュが汚される
  4. 分岐命令が実行され WB ステージで命令が取り消される

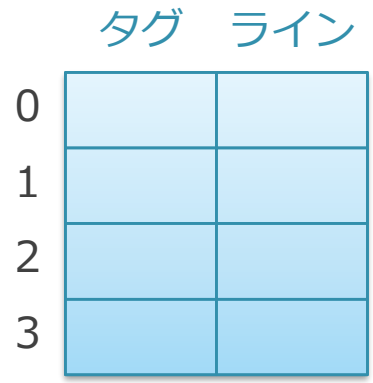
攻撃の手順

- 手順
  1. 分岐予測器の事前学習
  2. キャッシュの埋め尽くし
  3. 攻撃対象コードの実行
  4. **キャッシュの観測による値の取得**

キャッシュの観測による値の取得

- `if (x < array1_size)`
  `y = array2[array1[x] * 64];`

- 動作
  1. `array1` がバイト単位の配列の場合，`array1[x]` が取りうる候補は `0-255` の `256` 通り
  2. `array2` へのアクセスにより，`array1[x]`の値に応じた `256` 箇所のどこかがアクセスされる
  3. 64倍されているので，値ごとにセットアソシアティブ・キャッシュの異なるセットにアクセス
     - たとえば `array1[x]=1` で `array2` の先頭が `0x8000` であれば，右のセットの１番のラインが置き換えられる
  4. ユーザーに戻って来たあと，キャッシュにアクセスしてレイテンシを測る
     - セットの１番へのアクセスのレイテンシが長ければ，読み出した値が１であったとわかる



そんな都合の良いコードはあるのか？

- *64 ではなくとも，この形の間接アクセスはそれなりにあるようだ
  - ◇ `if (x < array1_size)`
    `y = array2[array1[x] * 64];`
- Variant 2 では BTB を事前学習する
  - ◇ 間接分岐の飛び先を自分で用意したコードのアドレスにしておく
  - ◇ いくらでも自由にコードが用意できる

Spectre Variant 4

- Variant 4 ではメモリ投機による投機実行を利用する
- メモリ投機：
  - 先行するストアに依存しないと予測したロードを先に投機実行する
  - これにより不正に読みたい場所を投機的に読み出す
  - その後は Variant 1 と同じ

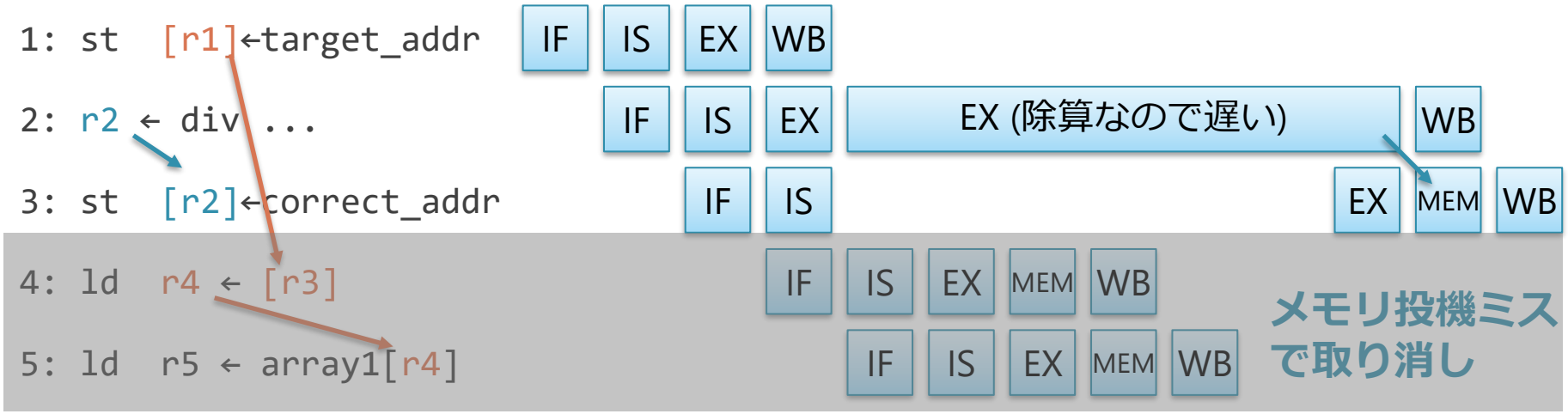
Spectre Variant 4

- ```
  // Variant 1 の攻撃対象コード
  // 分岐予測により投機的に array[x] を読ませる
  if (x < array1_size)
      y = array2[array1[x] * 64];
  ```

- ```
  // Variant 4 の攻撃対象コード
  // メモリ投機により投機的に array[x] を読ませる
  // a, b, c は同じ場所を指すポインタ
  *a = X // array1[x] として不正に読みたいインデックス
  b = なにかとても時間がかかる処理
  *b = Y // array1[x] としてただしく読めるインデックス
  x = *c;
  y = array2[array1[x] * 64];
  ```

攻撃対象コードの実行の様子

r1,r2,r3 は同じ値（アドレス）が入っているとする



- 命令 1 が *r1 に書き込んだ target_addr を
  命令 4 が投機的に *r3 から読み出し，array1 の読み出しに使用
  - ◇ 命令 4 は本当は 命令 3 に依存しているが，依存しないと予測
  - ◇ この領域は本来は読めないので，本来は例外で落とされる
- r2 も r1/r3 と同じアドレスを指していることが後からわかるので，その際に命令 4 は取り消される
  - ◇ 例外は起きない

Spectre のやばいところ

- カーネル・モードの値が読めるだけではない
- ブラウザ上で動いている javascript から，ブラウザ本体のメモリが読める可能性がある
  - ◇ パスワードやクレカ番号も読めるかも

Spectre 対策：ソフト編

- ブラウザ：
  - ◇ キャッシュのヒット・ミスがわからなくなるぐらいに時間計測機能の精度を落とす（ノイズを載せる）
- OS：カーネルとユーザーで仮想アドレス空間を完全に分離
  - ◇ システム・コールごとに切り替えが起きるので性能低下
  - ◇ 最近の Windows アップデートでこれで 1 割とかぐらい性能が落ちたという話も
- 範囲チェックの後にさらにガードを設ける
  - ◇
```
if (x < array1_size)
    x %= array1_size; // 投機実行されても x は範囲内に
```
- 「ここを超えて投機実行はしない」というフェンス命令があるのでそれを要所にいれる
  - ◇ 投機実行が阻害されるのでめっさ性能が落ちる

Spectre 対策：ハード編

- 基本的にまだみんな研究段階
  - 方向性１：
    - 投機状態ではキャッシュを置き換えないように頑張る
    - アクセスの開始がおくれて結構性能が下がることが多い
    - 考え落とした穴が多分どんどん見つかる気がする
    - InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy, MICRO 2018
    - MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State, ISCA 2020
  - 方向性２：
    - キャッシュのセットのインデクスをランダマイズする
    - アドレスとセットの対応がプログラムやモードごとに変化
    - 値とセットの対応がとれなくなる
    - CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping, MICRO 2018

まとめ

1. 保護機構
    1. 仮想メモリ
    2. 特権モード
2. 脆弱性とアタック
    1. バッファ・オーバーフロー
    2. Return Oriented Programming
    3. マイクロアーキテクチャ面の脆弱性

レポート課題

- 第１２回の講義資料を参照
- 締め切り：
  - 今年の秋に卒業予定の人：8/6（火曜）まで
  - そうでない人：8/13（火曜）まで
- 提出方法：
  - 「先進計算機構成論レポート <学籍番号>」 のタイトルで，以下までメールに添付して提出
    - shioya@ci.i.u-tokyo.ac.jp

感想

- 本日の講義でよくわかったところ，わからなかったところ，質問，感想などを書いてください
  - ◇ LMS の出席を設定するので，そこにお願いします
  - ◇ パスワード: spectre
- 意見や内容へのリクエストもあったら書いてください
- 今日は別に書いても書かなくても成績に影響しません

感想とか質問とか

- CPUの複数のコア間で共有されているデータは同期する必要がありますが、この場合、CPUがより大きなキャッシュまたはより多層のキャッシュを使用することを制限することがありますか。

感想とか質問とか

- メモリアクセス時間が容量の平方根ぐらいに比例する話がとても腑に落ちて、自分が他人に説明する時にも使えるとなと思いました

感想とか質問とか

- CPUやアクセラレータに搭載するメモリは、どのようなことが研究のトレンドなのでしょうか？

感想とか質問とか

- 最近のプログラムでは, hard wareの都合で生じる現象は大体隠蔽されているようなイメージを持っていたので意外だった. このようなプログラマが書いたプログラムについて, キャッシュミスを減らすようなソフトウェア的サポートにはどのようなものがあるのかが気になった.

感想とか質問とか

- セットアソシアティブ方式のキャッシュにおいて、例えば行列積計算における2つの行列など、頻繁にアクセスされる複数のデータが運悪く同じセットに割り当てられてしまうことによる性能低下は起こりうるのでしょうか。また、それを避ける仕組みなどは存在するのでしょうか。

感想とか質問とか

- Zen5のアーキテクチャでは、キャッシュが増えているが、キャッシュアクセスするためのアクセス数も増えた。そこで、キャッシュにアクセスするサイクル数とキャッシュの大きさ間のバランスに気になる。