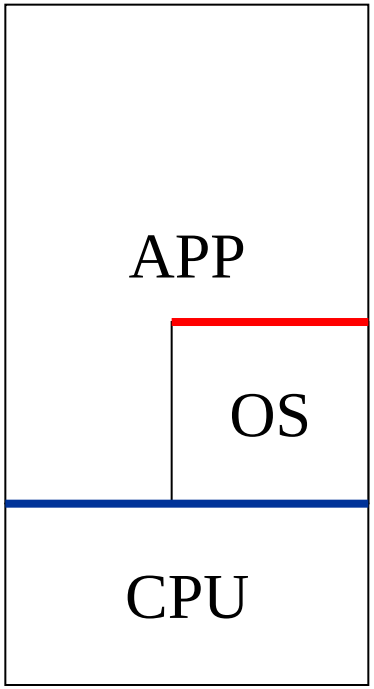
メモリ管理(2)

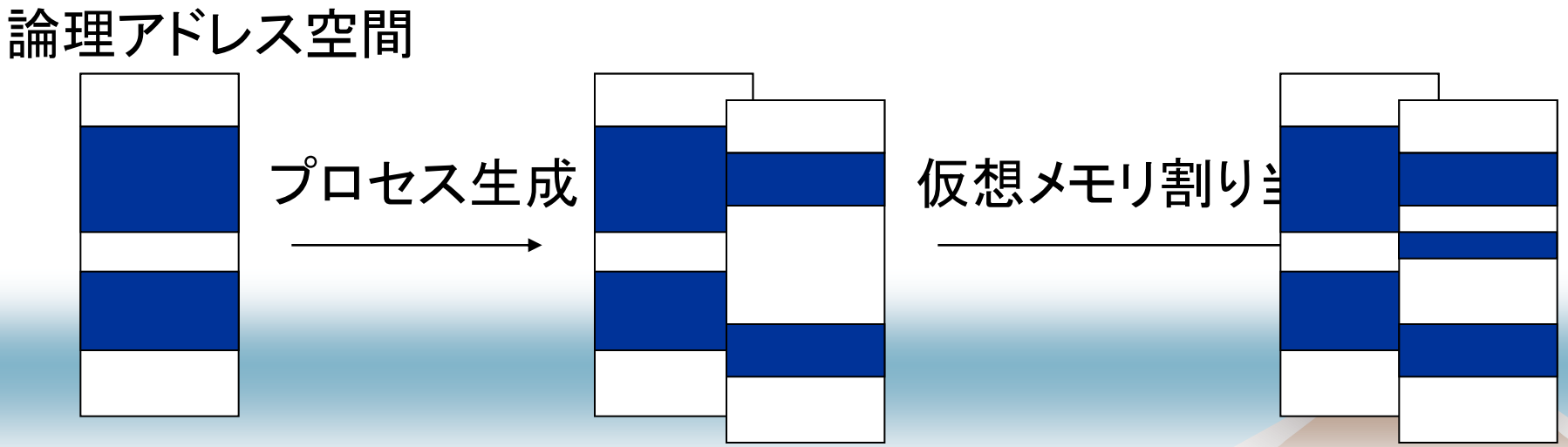
思い出そ～～う

- 物理アドレスと論理アドレス
- 論理アドレス空間
- アドレス変換
- メモリ管理ユニット (MMU)
- ページ
- ページテーブル，TLB
- 保護違反，ページフォルト
- ページング



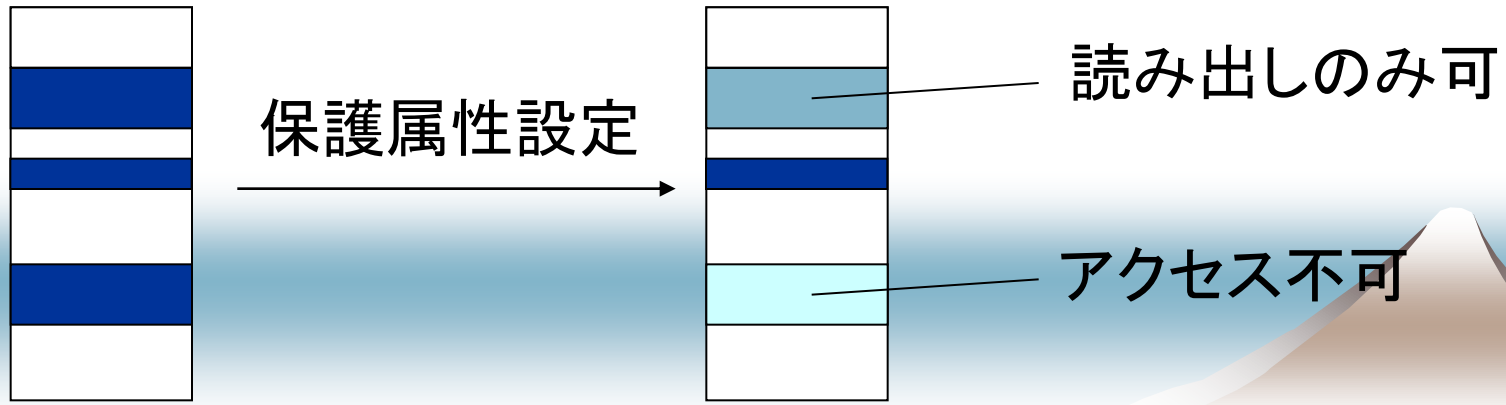
OSが提供するメモリ関連API (1)

1. 論理アドレス空間生成=プロセスの生成
   - プロセスの作成(プログラムの起動)
2. 論理的なメモリ(仮想メモリ)割り当て・解放
   - メモリを(論理的に)割り当て, 解放する



OSが提供するメモリ関連API (2)

3. 保護属性の設定
   - 割り当て中のメモリ(ページ)のread/write属性(可/不可)を設定する
4. 通常のread/write
   - 妥当なアクセスを成功させ，そうでないものを禁止する



メモリAPI実例: Windows

- プロセス生成
  - CreateProcess
- 仮想メモリ割り当て・解放
  - VirtualAlloc, VirtualFree
- 保護属性の設定
  - VirtualProtect

メモリAPI実例: Unix系

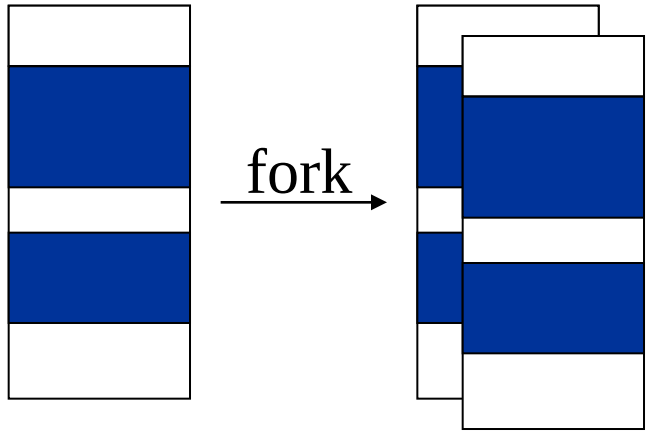
- プロセス生成
  - fork, exec
- 仮想メモリ割り当て・解放
  - brk, sbrk, mmap
- 保護属性の設定
  - mprotect

注

- もちろん実際には，ほとんどの場合メモリの割り当てにはmalloc/free, new/deleteなど，それぞれの言語のライブラリ・文法を使う
  - 使いやすくOS非依存
- malloc/free, new/deleteは内部でsbrk etc.を(時々)呼び出す

forkとexec

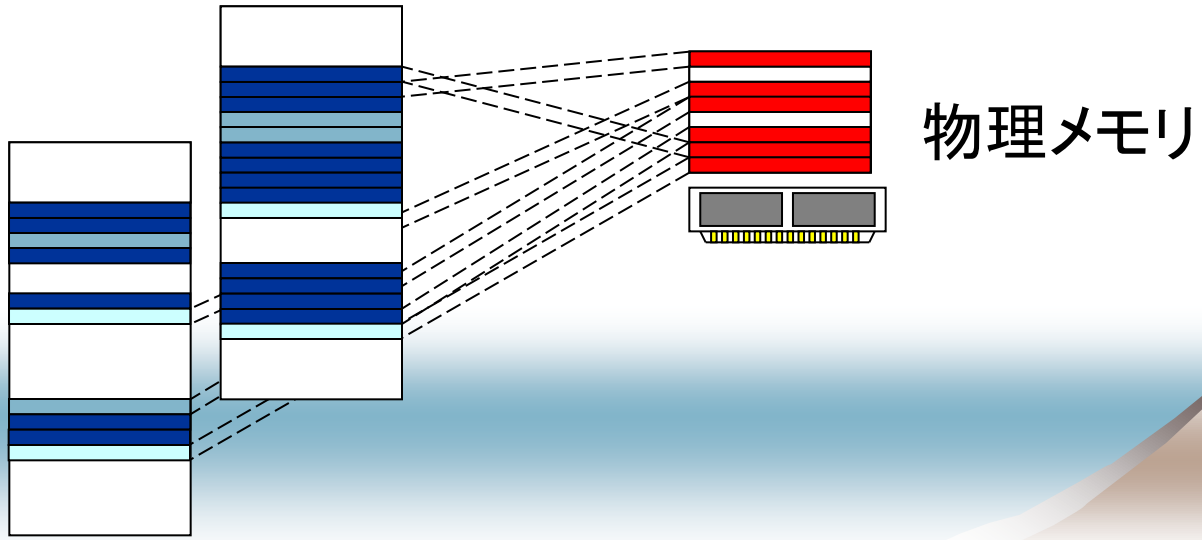
- fork : プロセス(アドレス空間)の丸ごとコピー
  - p = fork();
    if (p == 0) {
      /* 子プロセス */
    } else {
      /* 親プロセス */
    }



- 注: fork後，両者はメモリを共有しているのではなく, 同一内容のコピーを持つ

メモリ管理APIの動作
注目点(1)

- 物理メモリ512MBでも，1GBの仮想メモリ割り当てが成功する
- その時点で物理メモリが確保されているわけではない(≈小切手)



メモリ管理APIの動作
注目点(2)

- 物理メモリに存在しないページにアクセスすると，CPUが「ページフォルト」を発生させる
- 特に，割り当て後初めてのアクセスには必ずページフォルトが発生する
- OSはそれを「こっそり」処理して一見何事もなかったかのようにアプリケーションを再開
  - Demand Paging：小切手の換金

メモリ管理APIの動作 注目点(3)

- 実際に「時間がかかる処理」はメモリ割り当てそのものではなく，そこへのアクセス時に(時折発生する)ページフォルト
- 同じメモリアクセスでも
  - ページフォルトが発生するか否か
  - ページフォルトが発生する場合でも，ディスクからのページの取り出しが発生するか否か

でコスト(時間)が全く違う

ページフォルトを計測してみよう

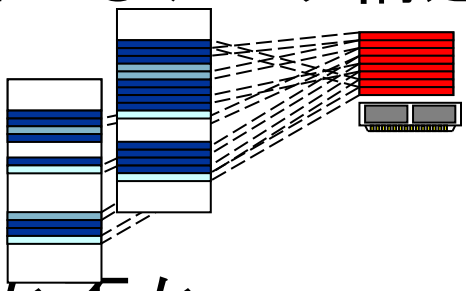
- キーワード
  - Demand paging
    - 2種類のページフォルト(メジャー, マイナー)
  - スラッシング(thrashing)

Quiz

- calloc vs. malloc

メモリ管理のためのデータ構造

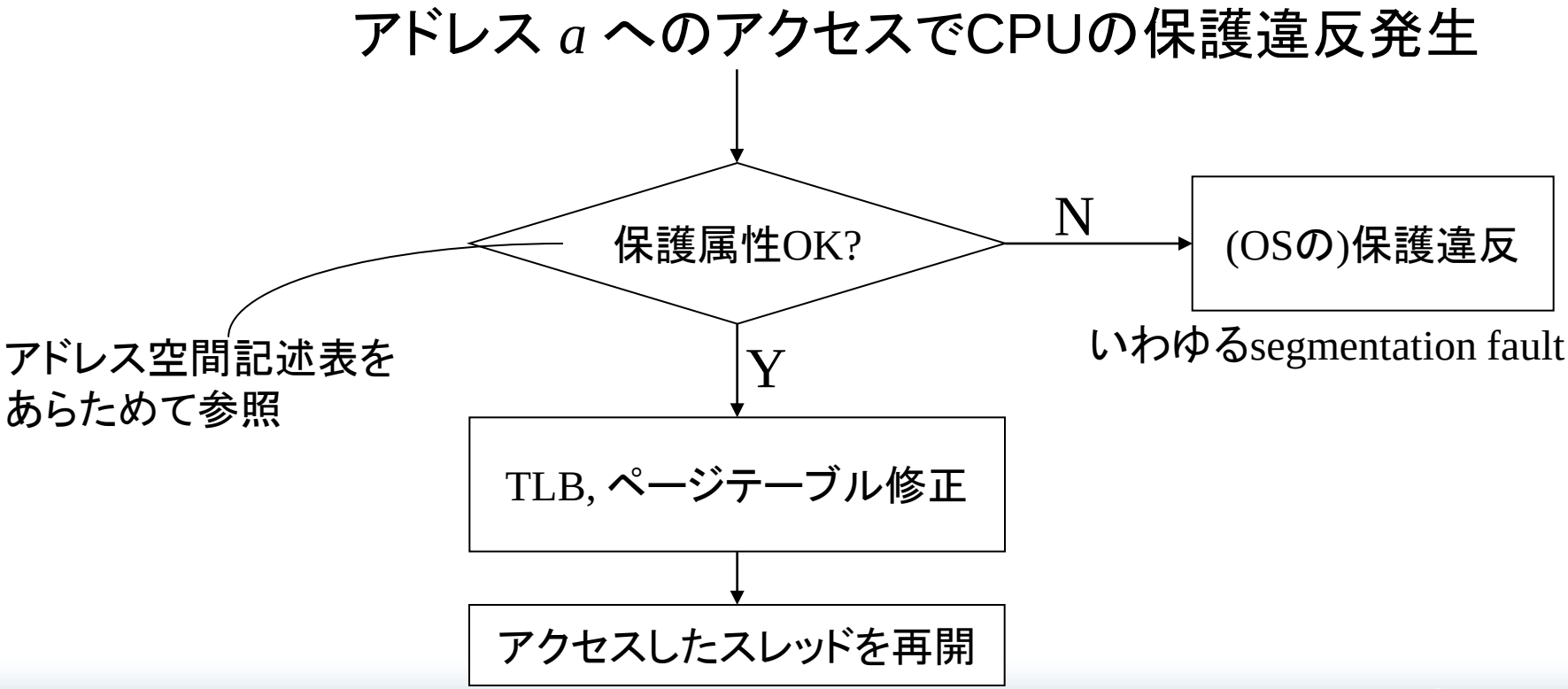
- アドレス空間記述表
  - 割り当て中の仮想メモリ領域(論理ページの集合)，保護属性などを記述した，OSが定めるデータ構造
- ページテーブル&TLB
  - 各論理ページに対し，
    - そのページが物理メモリにあるか否か
    - ある場合，その物理ページ番号と保護属性などを記述した表．CPUが毎メモリアクセス時に参照



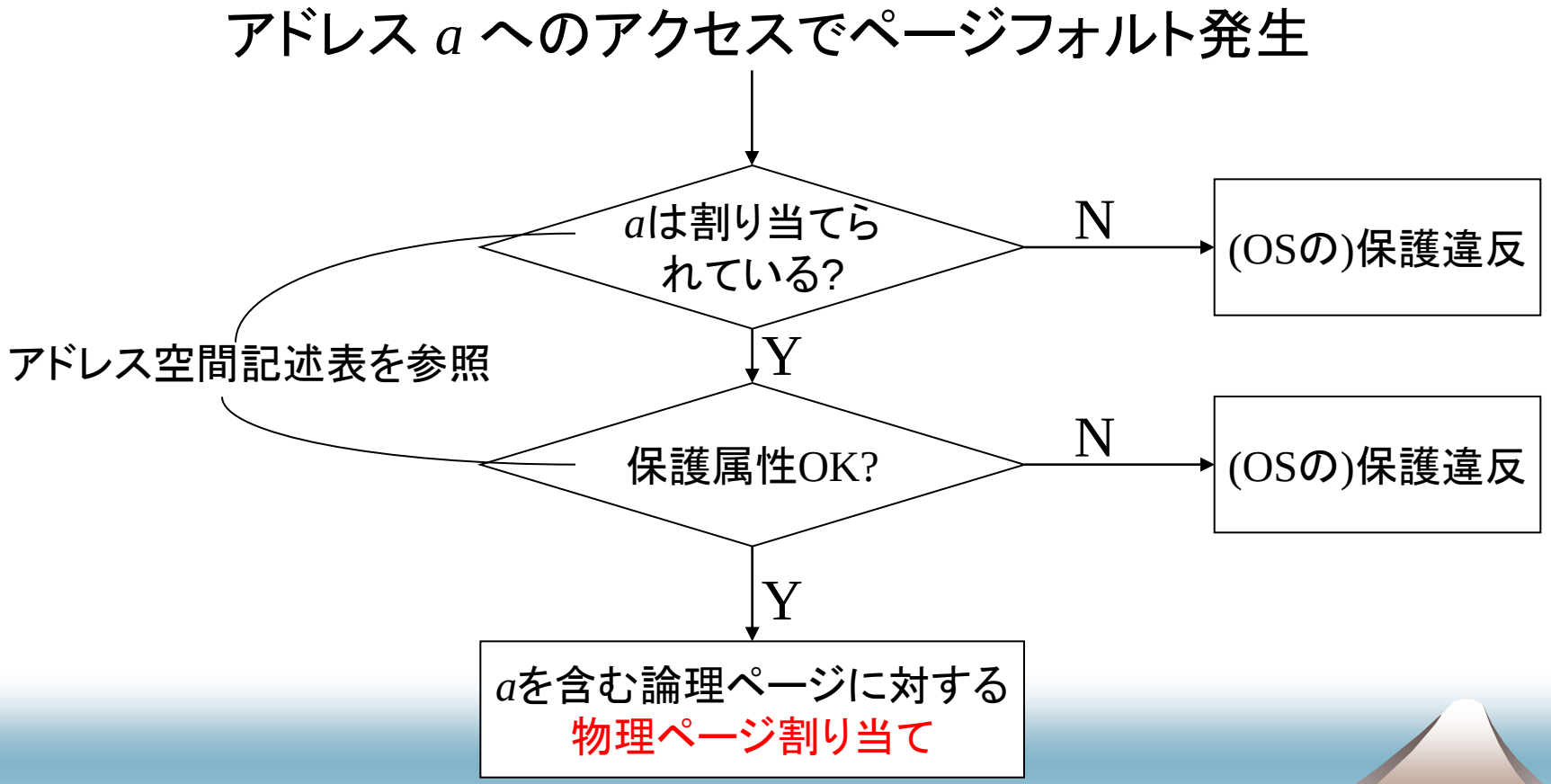
各APIの動作の実際

- 仮想メモリ割り当て・解放
  - アドレス空間記述表に記録
  - (解放の場合)ページテーブル, TLBのエントリ削除
- 保護属性の設定
  - アドレス空間記述表に記録
  - (禁止の場合) ページテーブル, TLBのエントリ修正
- ポイント (不変条件):
  - ページテーブル&TLBで許されるアクセス
    ⊂ アドレス空間記述表で許されるアクセス

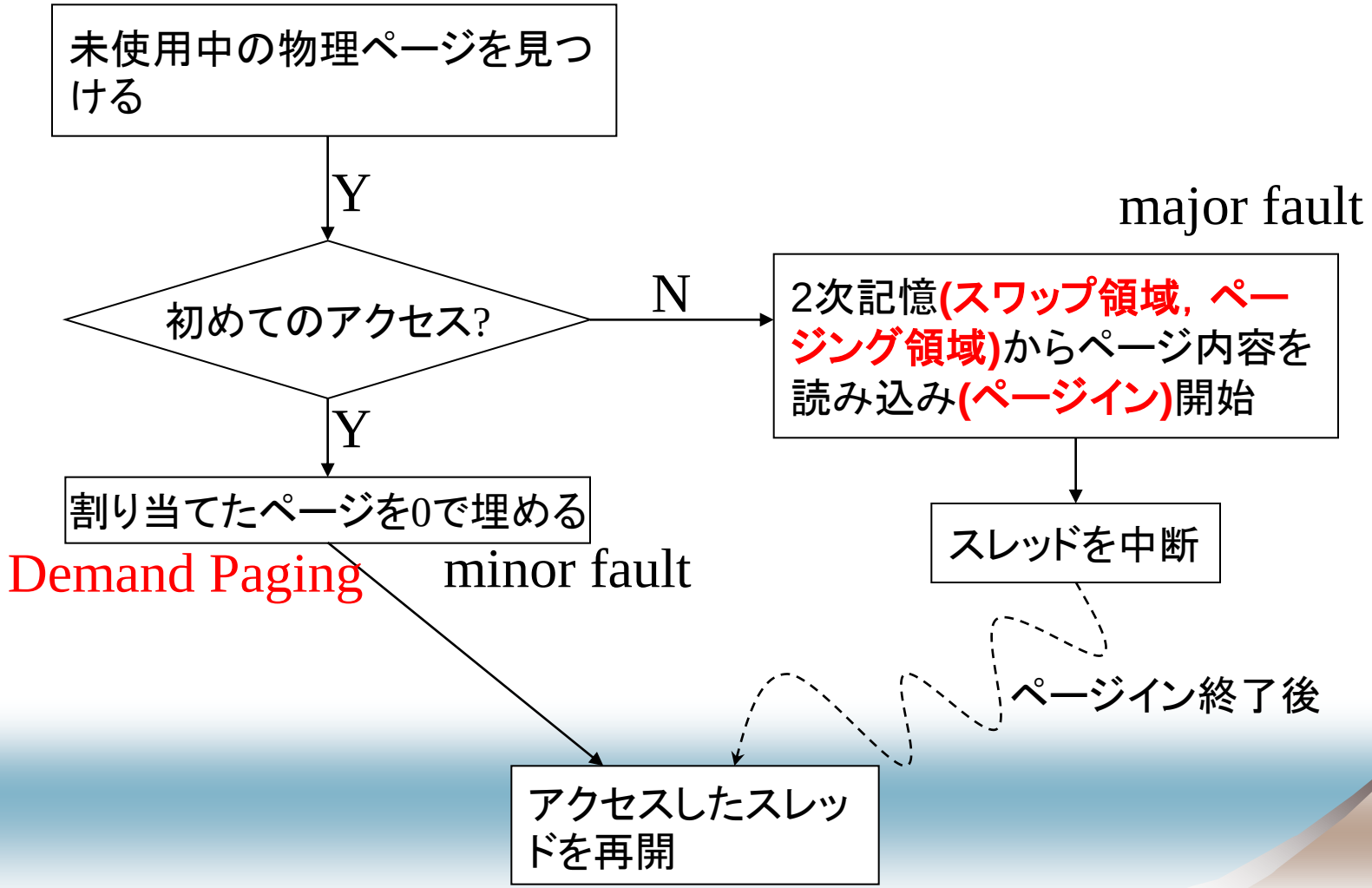
CPU例外(トラップ)発生時の処理 —保護違反



CPU例外(トラップ)発生時の処理 ―ページフォルト



物理ページ割り当て


未使用中の物理ページを見つける
Y
初めてのアクセス?
N
major fault
2次記憶(スワップ領域，ページング領域)からページ内容を読み込み(ページイン)開始
Y
割り当てたページを0で埋める
スレッドを中断
Demand Paging
minor fault
ページイン終了後
アクセスしたスレッドを再開

ページャ，スワッパ

- 空き物理ページが少なくなると起動される
- 「適切な」ページを2次記憶に移動(ページアウト)
- 選択の基準:
  - ページイン・ページアウトのコスト最小化
  - ページ置換アルゴリズム(後述)

(OSの)保護違反発生時の処理

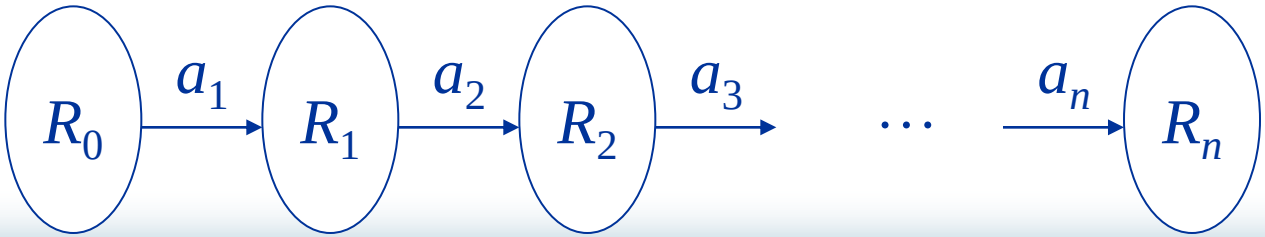
- 通常は「プログラムの終了」
  - Segmentation Fault (Unix)
  - “深刻なエラー…「送信する・しない」” (Windows XP)
- 実は，プログラムで処理可能な例外が発生
  - Unix : シグナル
  - Windows : 構造化例外処理
  - 来週以降, その応用とともに後述
- 例外が処理されない場合，プログラム終了

ページ置換アルゴリズム

- どれかのページを物理メモリから除去する必要があるとき，どのページを除去すべきか?
- 目標：ページイン・アウトのコスト最小
  - 置換「数」を少なくする
  - 一置換あたりのコストを少なくする
    - ページインしてから更新されていない(ディスクへ書き出す必要がない)ページを優先的に置換する

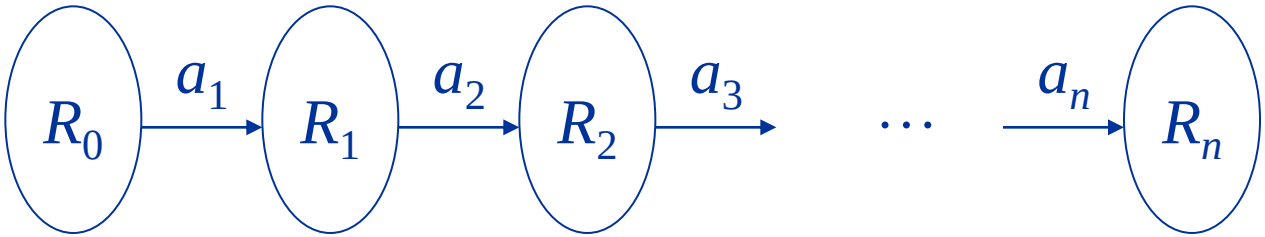
若干理想化された問題のモデル(1)

- 問題: R_0 (初期常駐ページ集合)と未来のアクセス系列$\boldsymbol{a} = a_1, a_2, \ldots, a_n$が与えられる.
- アルゴリズム$A(R_0, \boldsymbol{a})$は，各アクセス後の常駐ページ集合$R_1, R_2, \ldots, R_n$を定める(R_i : アクセスa_i直後の常駐ページ集合)



若干理想化された問題のモデル(2)

- 目的: ページ置換数$\sum| R_i - R_{i+1}|$ の最小化
- 制約: $a_i \in R_i$, $| R_i | = M$ (物理ページ数)



条件 $a_i \in R_i$
ページ置換数 $= \sum| R_i - R_{i+1} |$

注: 行っている理想化

- 物理メモリは常に満杯
- どのページも置換するコストは変わらない
  ⇒ 置換「数」最小化だけが目標

重要性の確認

- 極端な例:
  - 物理ページ数n
  - アクセス系列 $\overbrace{1,2,\ldots,n,n+1}, \overbrace{1,2,\ldots,n,n+1}, \ldots$
- 最低のアルゴリズム:
  - 全アクセスでページフォルト
- 最高のアルゴリズム:
  - n アクセスに一度のページフォルト

未来のアクセス系列が既知であれば，常に最適なアルゴリズムが存在する

- アルゴリズム:
  “ページ置換時に，現在物理メモリにあるページの中で，次にアクセスされる順番が最後のものを除去する”
- チャレンジ課題: これが「最適」であることを証明せよ

でも未来のアクセスは未知だから…

- 物理ページ数n, アクセスページ数$> n$ ならば，どんなアルゴリズムも，最悪の場合，全アクセスでページ置換が必要(当然)
- どんなアルゴリズムも，近い将来起こりそうなアクセスを予想する経験則に基づく

手がかり:
アクセスの時間的局所性
(temporal locality)

- 多くのプログラムで,「最近アクセスした場所を，またアクセスする」
  - はるか昔にアクセスしただけのページよりも，最近アクセスしたページのほうが，次に先にアクセスされる可能性が高い
  - 例: スタック

⇒ LRU (Least Recently Used)置換の考え方

LRU置換

- Least Recently Used ○○:
  - 最後に使われたのが，もっとも遠い過去であるような○○
- 例
  - 現在の常駐ページ = { 2, 3, 6, 7, 8 }
  - 最近のアクセス: 8710979876827 32

現在の常駐ページ中
Least Recently UsedなPage

LRU置換アルゴリズム

- ページ置換時にLRUページを置換する
- 次に使われるのももっとも遠いのではないかと期待する!
  - 実際どのくらい一般的に本当かは定かではないが
- ページングに限らず様々な場面で登場する考え方

正確なLRU置換を実装するのは困難

- 候補1:
  - ハードウェアで全物理ページの最終アクセス時刻(カウンタ)を記録
  - ページ置換時に全物理ページのカウンタ比較
- 候補2:
  - ハードウェアで最終アクセス時刻の順番にリストを作っておく

LRUの近似
(Not Recently Used : NRU)

- 要するに「最近使われたもの」(recently used)とそうでないもの(not recently used)をだいたい区別できればよい
- ハードウェアに簡単に実装できる機構: reference/dirty bits (R/D bits)
  - ページテーブル中にあり，1ページにつき2 bit
    - reference bit : read時にset
    - dirty bit : write時にset
  - ソフトウェア (OS)によってclear

NRUの一例

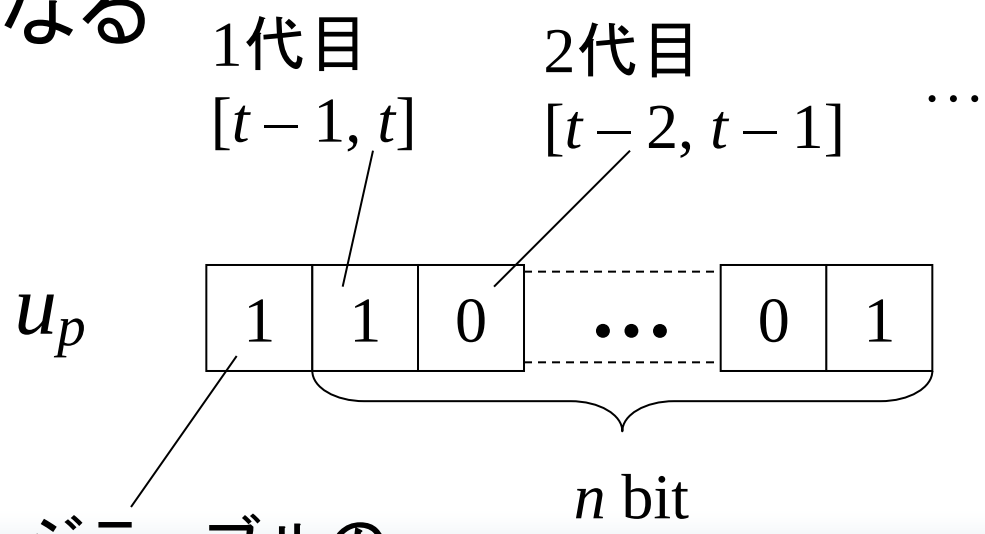
- OSが1秒おきに
  - for each page p in physical memory {
    $u_p = R_p \mid D_p$; /* R_p : reference bit
    D_p : dirty bit */
    $R_p = D_p = 0$; /* clear reference/dirty bits */
    }
- ページ置換時
  - page out p s.t. $u_p = R_p = D_p = 0$ if any;
  - otherwise page out p s.t. $R_p = D_p = 0$ if any;

追加reference bit方式(1)

- 前方式の一般化：各ページにつき，ページテーブル(R/D bits) + n 代のコピーを保持
- 最後にR/D bitsをclearした時刻をtとして，
  - 1代目：時刻 $t-1$から t までに使われたら1
  - 2代目：時刻 $t-2$から $t-1$までに使われたら1
  - …

追加reference bit方式(2)

- 各ページにつき，概念的には以下のような最近n秒のaccess履歴を管理していることになる



追加reference bit方式(3)

- OSが1秒おきに
  - for each page p in physical memory {
    $u_p = (u_p >> 1) + ((R_p \mid D_p) << (n - 1))$;
    $R_p = D_p = 0$;
    }
- ページ置換時
  - page out p s.t. $(((R_p \mid D_p) << n) + u_p)$ is minimum

その他の考慮事項

- 一括read/write
  - 一回のdisk accessで近隣の複数ページをまとめて読み込む
  - 逐次アクセスに対して効果的