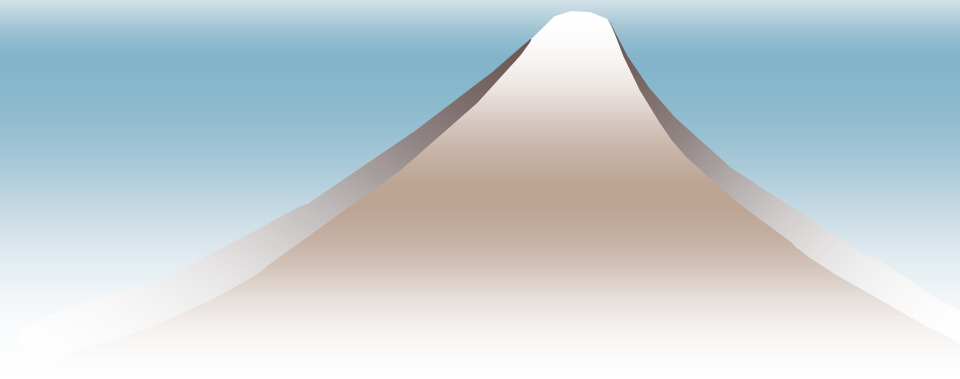


# イベント通知機構・メモリ保護API

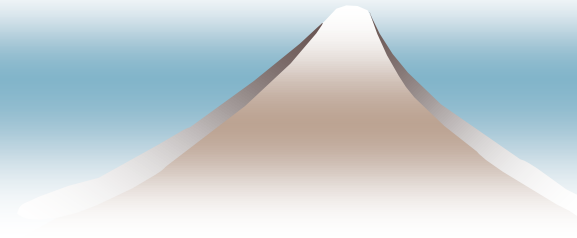


# 仮想記憶機構

- ◆ 毎メモリアクセスに介在し,
  - アドレス変換
  - 条件により例外発生(ページフォルト, 保護違反)
- ◆ 元々の目的
  - プロセス間の保護(メモリの分離)
  - 仮想記憶 > 物理記憶
  - demand paging

# 仮想記憶機構の「応用」

- ◆ 備わっている「機構」の，もともとの目的をちょっと逸脱した利用



# OS内部で用いられた応用(1)

## ◆ メモリマップドファイル

- 大きなファイルの効率的なランダムアクセス
- 読み込み専用メモリのプログラム間での共有(メモリ節約)
  - プログラムテキスト(ライブラリ)

## ◆ プロセス間共有メモリ

- 異なる(論理)ページを同一の物理ページにマッピングする(API: メモリマップドファイルのMAP\_SHARED)

# OS内部で用いられた応用(2)

## ◆ Copy-on-write

- プライベートマッピングであっても、実際に書き込まれるまでは物理ページを共有(メモリの節約)
- 書き込まれたらコピーを生成
  - 書き込みの検出はページテーブル, TLB内の保護属性を「書き込み不可」にすることで行う
- 一番の用途: Unixのfork

# 共通アイデア

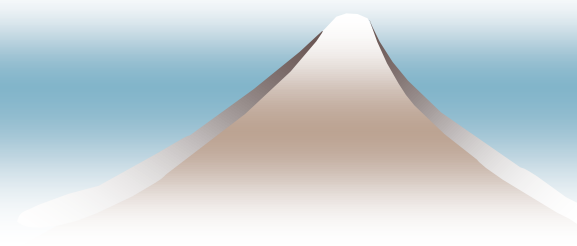
- ◆ ページテーブルに「仕掛け」をしておく
  - 書き込み不可, マッピング不在, 物理メモリ共有etc.
- ◆ 「重要な」メモリアクセスをOSがCPU例外で検出
  - 保護違反またはページフォルト
- ◆ 例外時の動作によって様々な処理(copy-on-write, ページング, etc.)を実現

# さらなるアイデア: ユーザレベル仮想記憶API

- ◆ メモリ(ページ)の保護属性をアプリケーションが操作できるようにする
  - 読み出し可・不可
  - 書き込み可・不可
  - 出る単 readonly (読み込み可・書き込み不可)
- ◆ 「保護違反」をユーザレベルに通知
  - 単にプログラムを終了させるのではない処理が可能
  - Segmentation Faultは実はその一例

# 以降の概要

- ◆ (準備となる話題): イベントの通知API
  - Unix : シグナル
  - Windows : 構造化例外処理
- ◆ 保護違反通知の高度な応用例
- ◆ Andrew W. Appel and Kai Li. “*Virtual Memory Primitives for User Programs*”



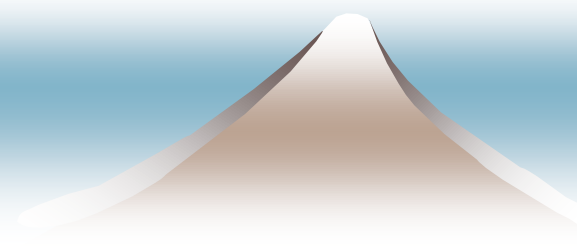


# イベント通知API

## ◆ 基本概念

- スレッド実行中におこる例外的な事象またはスレッドの外でおこる事象を(スレッドが明示的に監視することなく)通知する
  - cf. CPUに対する割り込み
- いわば「スレッドに対する割り込み」

## ◆ API

- Unix : シグナル
  - Windows : 構造化例外処理
- 

# Unixシグナル: 基本概念

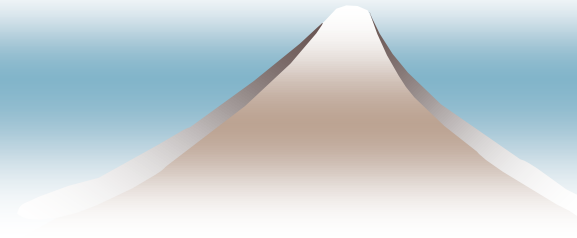
- ◆ スレッドが,
  - 受け取りたいシグナルと,
  - それを受け取ったときの処理(action; シグナルハンドラ)

をOSに登録

- ◆ シグナルの配達
    - 事象発生時にOSが配達
    - 他のプロセスが明示的にシグナルを配達
- ⇒ 対応するハンドラが(突然)実行される

# 例1

- ◆ シグナル : SIGSEGV (Segmentation Fault)
- ◆ いつ発生するか?
  - プロセスが「保護違反」(違法なメモリアクセス)を起こしたとき
  - 注: プロセスが毎メモリアクセスごとに「違法チェック」をやっているわけではない

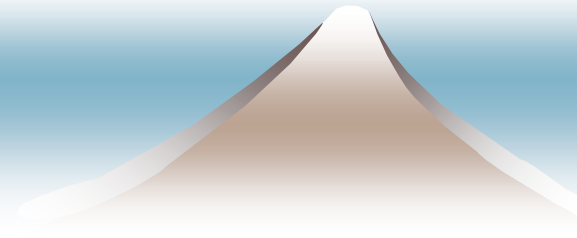


## 例2

- ◆ シグナル : SIGALRM
- ◆ いつ発生するか?
  - alarmシステムコールによって指定した時間が経過したとき
  - 注 : プロセスが1命令(または数命令)ごとに「現在時刻チェック」をやっているわけではない

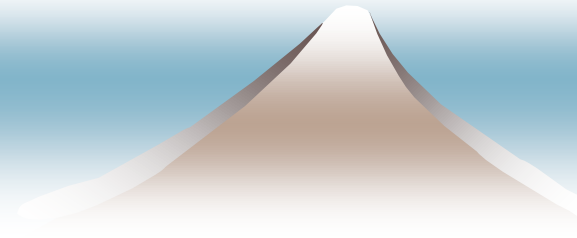
## 例3

- ◆ シグナル : SIGINT
- ◆ いつ発生するか?
  - (通常)端末に向かってCtrl-Cをtypeしたとき



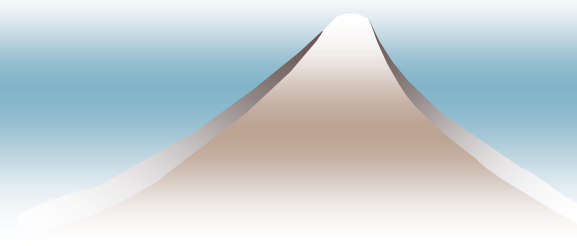
# killシステムコールとkillコマンド

- ◆ 明示的にシグナルを送るAPI
- ◆ `kill(pid, sig); /* システムコール */`
- ◆ `kill -sig pid # コマンド`
  - プロセスpidにシグナルsigを発生させる
  - 良く使う “`kill -9 pid`” はpidにシグナルSIGKILLを発生させている



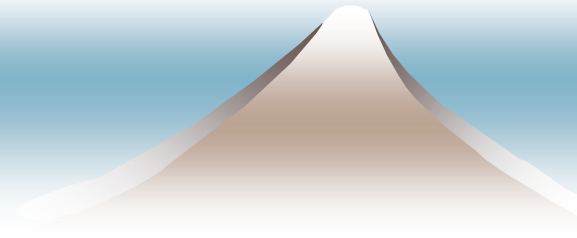
# その他のシグナル(抜粋)

- ◆ SIGILL /\* 不正命令の実行 \*/
- ◆ SIGBUS /\* バスエラー \*/
- ◆ SIGUSR1, SIGUSR2 /\* ユーザ定義シグナル \*/
- ◆ SIGKILL /\* プロセスの消滅 \*/



# シグナルハンドラの登録

- ◆ `int sigaction(signum, &act, &oldact);`
  - シグナル`signum`発生時の動作を`act`で指定
  - これまでの動作を`oldact`に格納

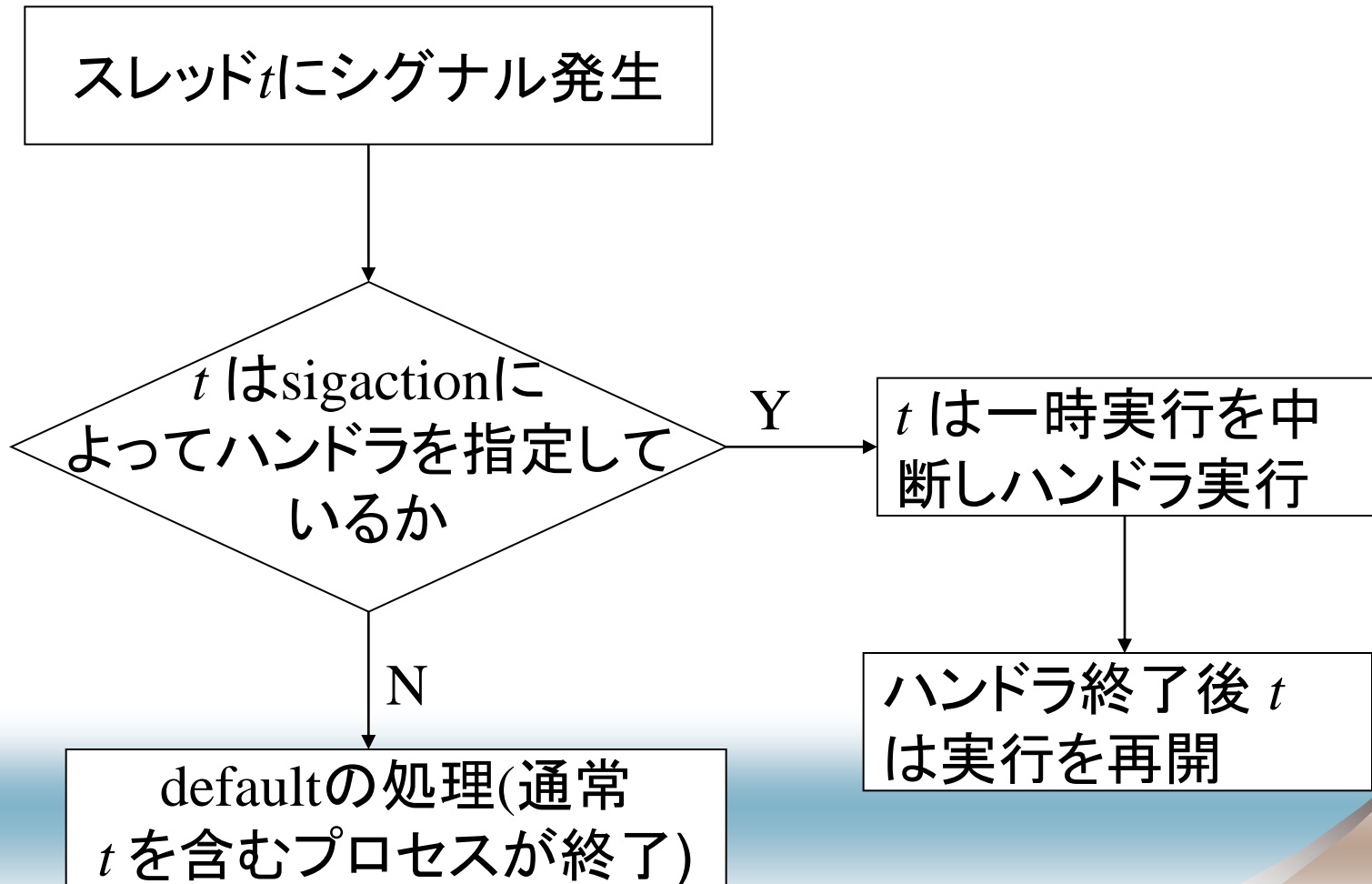




# シグナル使用テンプレート

- ◆ `void h(signum) {  
    シグナル発生時の処理; ... }`
- ◆ `struct sigaction act;  
act.sa_handler = h;  
...  
/* シグナルハンドラ登録:  
   SIGINT発生時にhが実行される */  
int sigaction(SIGINT, &act, &oldact);`

# シグナル発生時の流れ



# Windows構造化例外処理

- ◆ Visual C++に組み込まれた例外処理の構文で「例外」と「対応する処理」(例外ハンドラ)を指定
- ◆ 文法:  

```
__try {  
    本体;  
} __except (例外フィルタ) {  
    例外ハンドラ;  
}
```

# 構造化例外処理

...
__try
__try
__try

◆ `__try { S; } __except (F) { H; }`

- 注:  $S$ 内にまた\_\_try が現れることもある

◆ 意味:  $S$ を実行

- 途中に例外が発生しなければそのまま上の文全体の実行が終了
- 例外が発生したら, ...
  - (最も最近突入した\_\_tryブロックに対応する) $F$ を実行. その値によってその後の実行方法が決まる

# 構造化例外処理

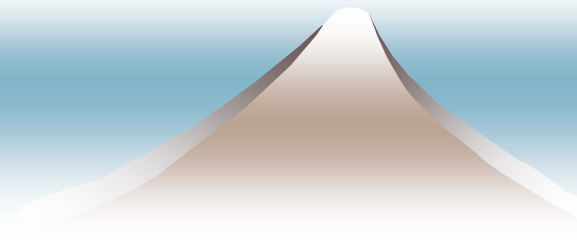
- ◆ フィルタ( $F$ )の評価結果により,
  - EXCEPTION\_CONTINUE\_EXECUTION
    - 例外発生地点から実行再開
  - EXCEPTION\_EXECUTE\_HANDLER
    - $E$ に対応する例外ハンドラ( $H$ )を実行
  - EXCEPTION\_CONTINUE\_SEARCH
    - ひとつ前に突入した\_\_tryに対応するフィルタを実行(なければ終了)

# 例外の種類

- ◆ STATUS\_ACCESS\_VIOLATION: 保護違反
- ◆ STATUS\_FLOATING\_DIVIDE\_BY\_ZERO : 0除算
- ◆ STATUS\_ILLEGAL\_INSTRUCTION: 未定義命令
- ◆ STATUS\_PRIVILEGED\_INSTRUCTION: 特権命令の不正な実行
- ◆ ...
  - 多くがUnixのシグナルに対応

# シグナル・例外がなぜ有用か？

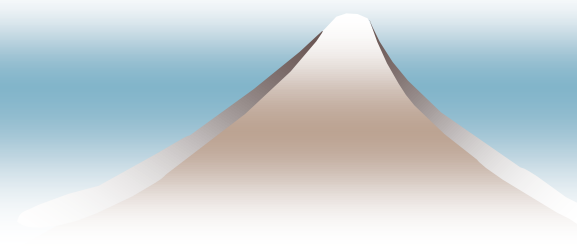
- ◆ 毎回検査していたのでは遅すぎる処理の代わりに、ハードウェアによる検査+「例外」の処理で代用する
  - 割り算 $a/b$ のたびに $b \neq 0$ を検査する
  - メモリアクセス $*p$ のたびに $p$ がある領域をさしていないかどうかを検査する
  - ...



# 保護違反(SEGV)の捕捉

- ◆ 例外処理の中でも特に,
  - メモリ保護API (mprotect, mmap, VirtualAlloc, VirtualProtect, MapViewOfFile)によるメモリ領域の保護属性の設定
  - シグナル・構造化例外処理による, それらの領域へのアクセスの捕捉

には多数の応用が発明されてきた  
「仮想記憶APIの応用」



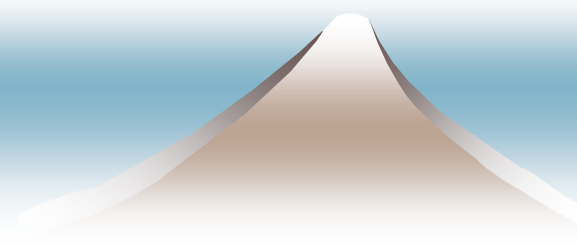


# 多くの応用に共通の基本アイデア

- ◆ ある領域をREAD (WRITE)不可に設定
- ◆ 通常通りプログラムを実行
- ◆ 途中で保護違反が発生したら, その場所をREAD (WRITE)可に設定+実行を継続
- ◆ ポイント
  - 実行結果は通常と変わらない
  - 「実行中どこ(どのページ)にアクセスしたか」の情報が得られる

# これまでに発明された応用

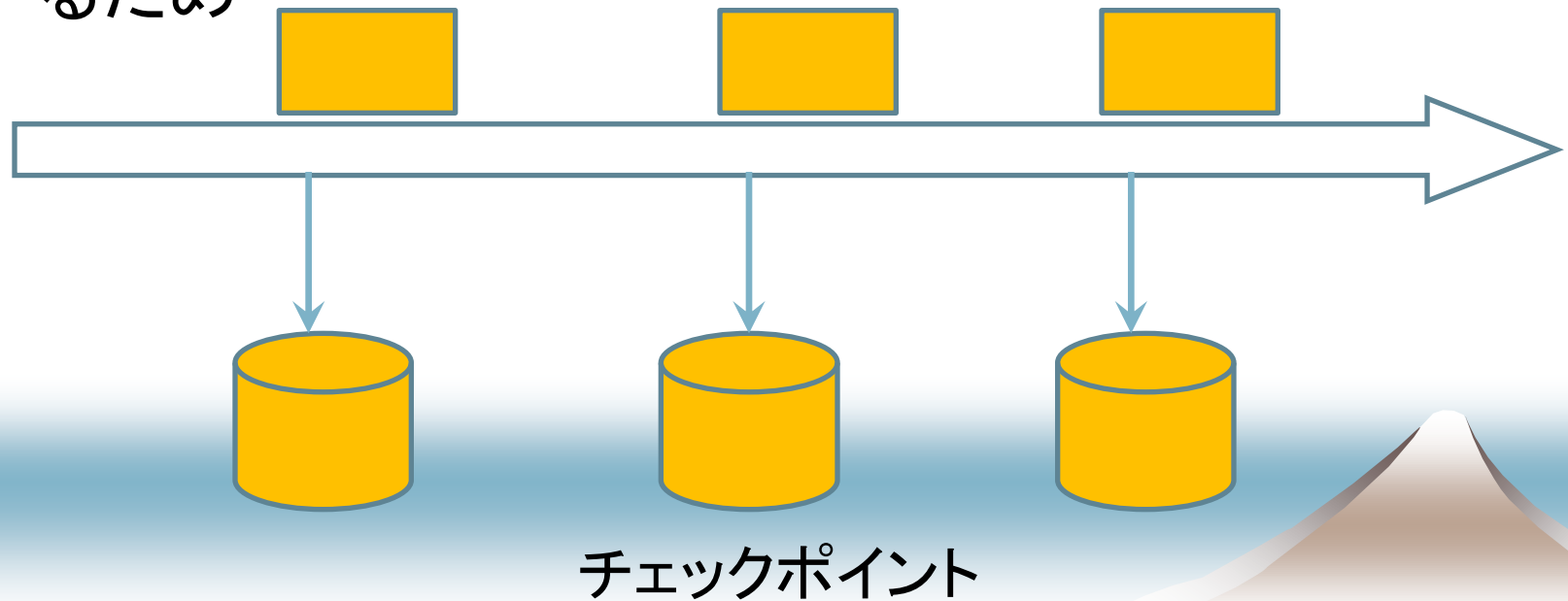
- ◆ 圧縮つきメモリマップドファイル
- ◆ 並行チェックポインティング
- ◆ トランザクションつきメモリマップドファイル(永続オブジェクト)
- ◆ Incremental Garbage Collection
- ◆ ネットワークページング
- ◆ 仮想共有メモリ(Shared Virtual Memory)
- ◆ ...
- ◆ 興味のある人は,
  - Andrew W. Appel and Kai Li. “*Virtual Memory Primitives for User Programs*”



# 応用例1: 差分チェックポイントティング

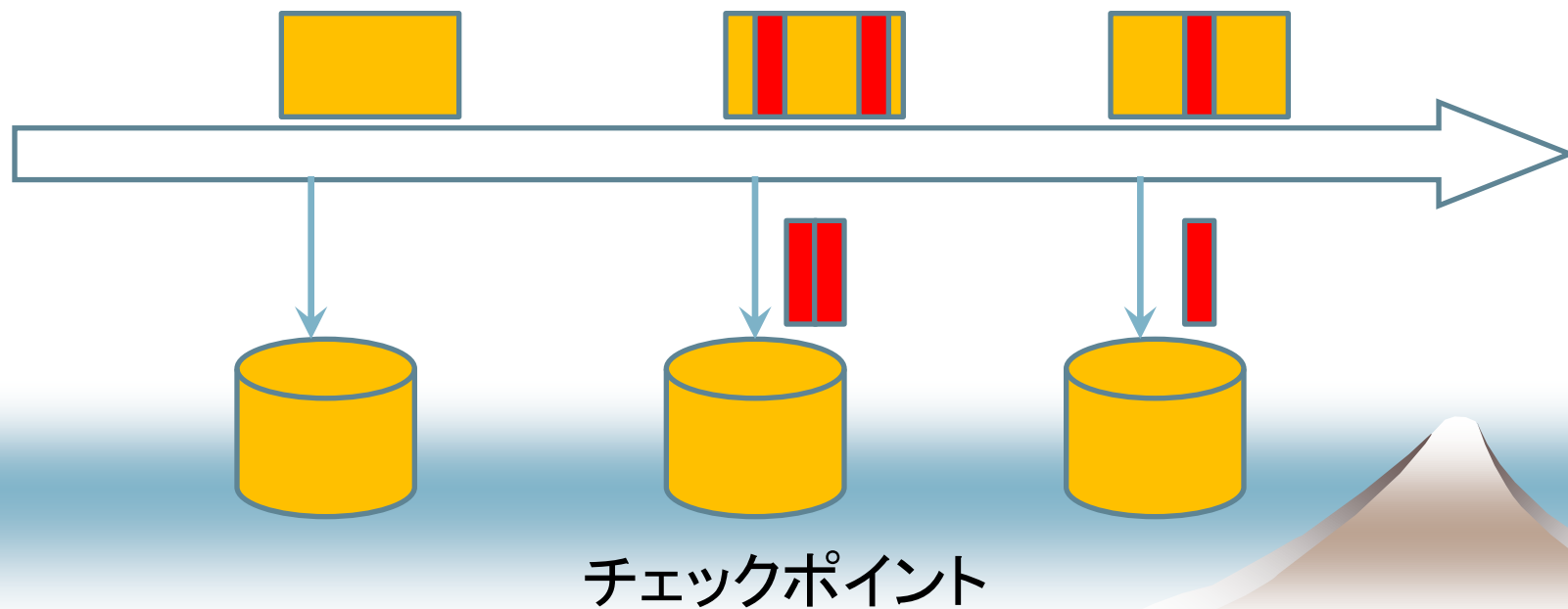
## ◆ チェックポイントティング:

- (長い)計算の途中でデータをファイルに書き出す
- おもな目的: 将来のクラッシュ時に途中から再開するため



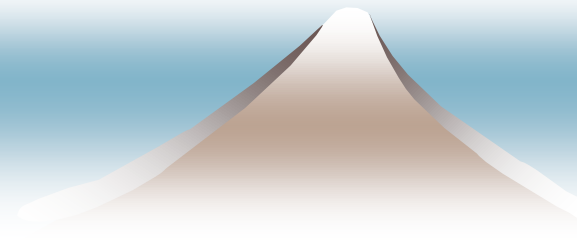
# 「差分」チェックポイントティング

- ◆ チェックポイント保存時に「前回との差分」だけを保存（高速・停止時間が短い）



# 差分チェックポイントティング：方法

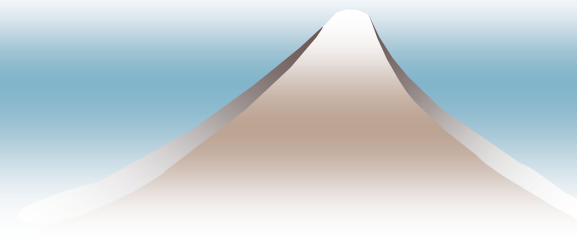
- ◆ チェックポイント保存後、全ページを“read-only”に設定
- ◆ Write fault時に書き込まれたページを記録  
→ “dirty pages”
- ◆ 次回チェックポイント時には、“dirty pages”だけを保存



# 応用例2:

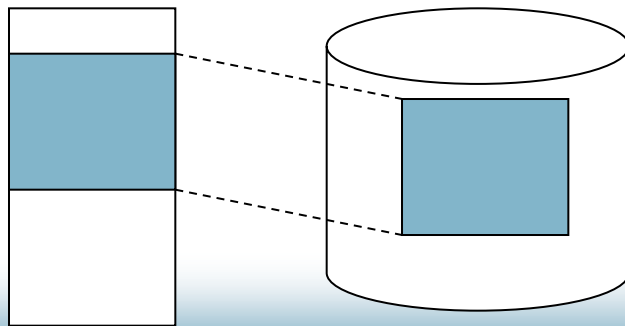
## 圧縮つきメモリマップドファイル

- ◆ 通常のメモリマップドファイル:
  - あるページに初めてアクセスした時OSがファイルの内容をメモリに(そのまま)コピー
- ◆ 圧縮つきメモリマップドファイル:
  - ファイルの内容が圧縮して格納されている
  - 初めてアクセスした時, メモリの内容を「解凍して」コピー

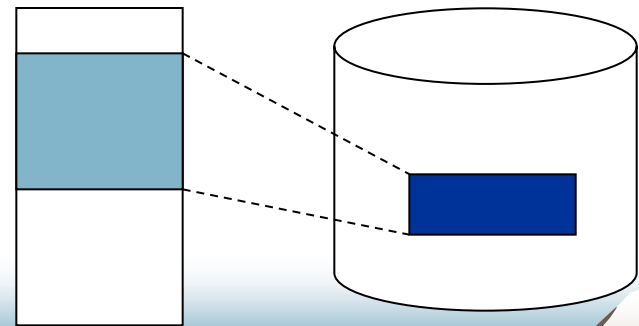


# 通常 vs 圧縮つき

- ◆ 注: OSがメモリマップドファイルの拡張としてサポートしてもよいのだが, 今はそれが無いという前提で, メモリ保護APIを利用して「ユーザが」実現することを考える



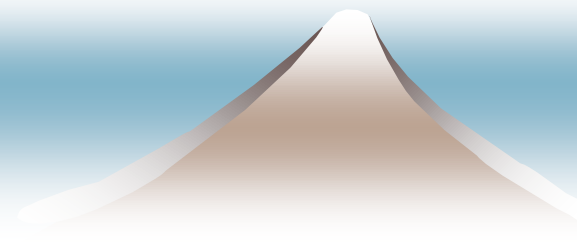
通常のメモリマップドファイル



圧縮つきメモリマップドファイル

# 圧縮つきメモリマップドファイル 実現概要

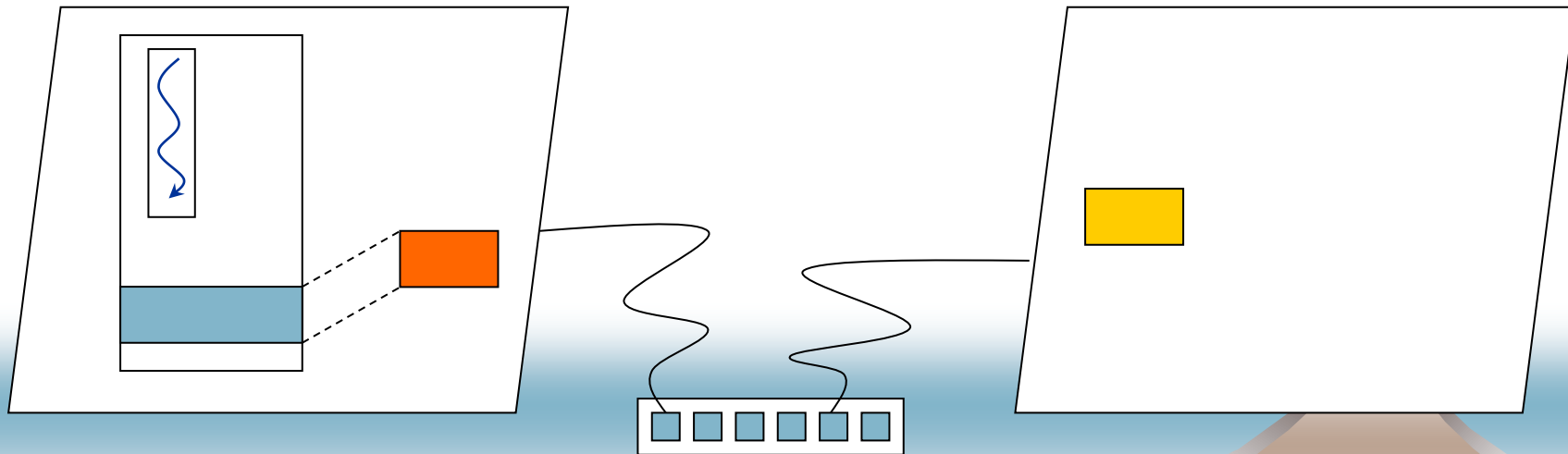
- ◆ ファイルは断片(例: 16KB)ごとに圧縮して格納
- ◆ 将来内容が展開される領域を「アクセス不可」に設定
- ◆ 保護違反発生時に、対応する断片を解凍して、読み込み(コピー or マップ)





# 応用例3: ネットワークページング

- ◆ ディスクの代わりに、「ほかの計算機のメモリ」をページング領域(退避場所)として使う
- ◆ ディスクの速度 < ネットワークの速度 のときに有効



通常のネットワーク(ソケットAPI利用)

# ネットワークページング 基本アイデア

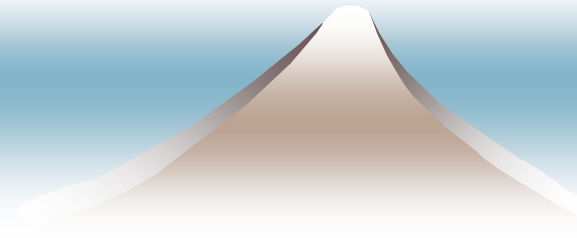
- ◆ OSのページングをユーザレベルで「真似」
- ◆ OSのページング
  - 空のページをアクセス→ ページフォルト→ ディスクからページイン
- ◆ ユーザレベルのページング
  - 読み書き禁止のページをアクセス→保護違反 (segmentation fault) →ネットワーク経由でページイン

# ネットワークページング 基本アイデア

- ◆ OSのページングをユーザレベルで模倣
- ◆ OSのページング
  - 物理メモリにないページをアクセス
  - ページフォルト
  - 2次記憶からページイン
  - アプリに戻る
- ◆ ユーザレベルのページング
  - 物理メモリにないページをアクセス
  - アクセス違反(segmentation fault)
  - (シグナルハンドラ) 他のマシンからページイン

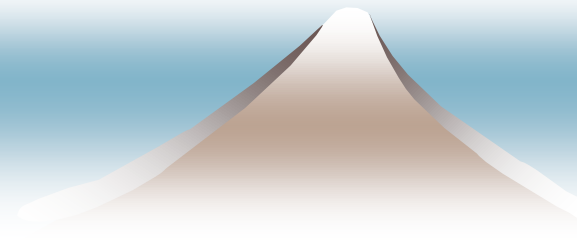
# ネットワークページング 実際

- ◆ 一定数(< 物理ページ数)のページ以外はすべて「アクセス不可」に設定
- ◆ Segmentation faultのハンドラ
  - アクセス違反を起こした対象ページを取得
  - ネットワーク経由でページを取得。代わりにどれかのページを追い出す



## 応用例4: 仮想共有メモリ

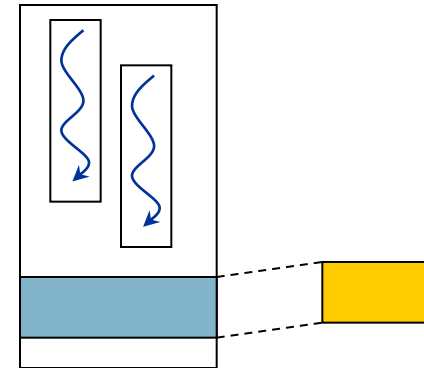
- ◆ 物理的にはメモリを共有していない, 複数の計算機間での「擬似的な」メモリの共有
  - 誰かが書き込んだ結果が自動的に他の人に反映



# 基本の復習: 共有メモリ

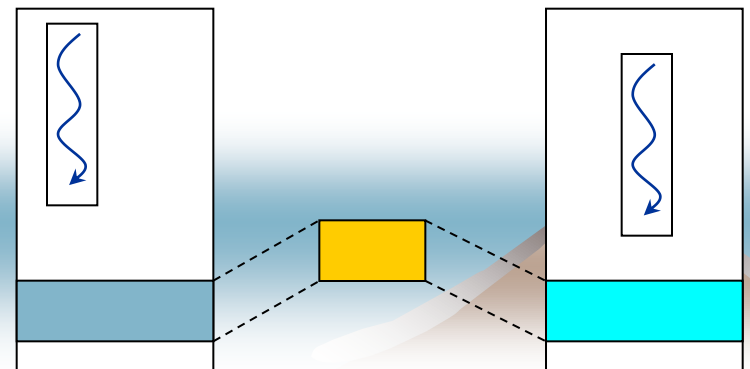
## ◆ 1プロセス内の複数スレッド

- 物理メモリ共有, アドレス空間共有



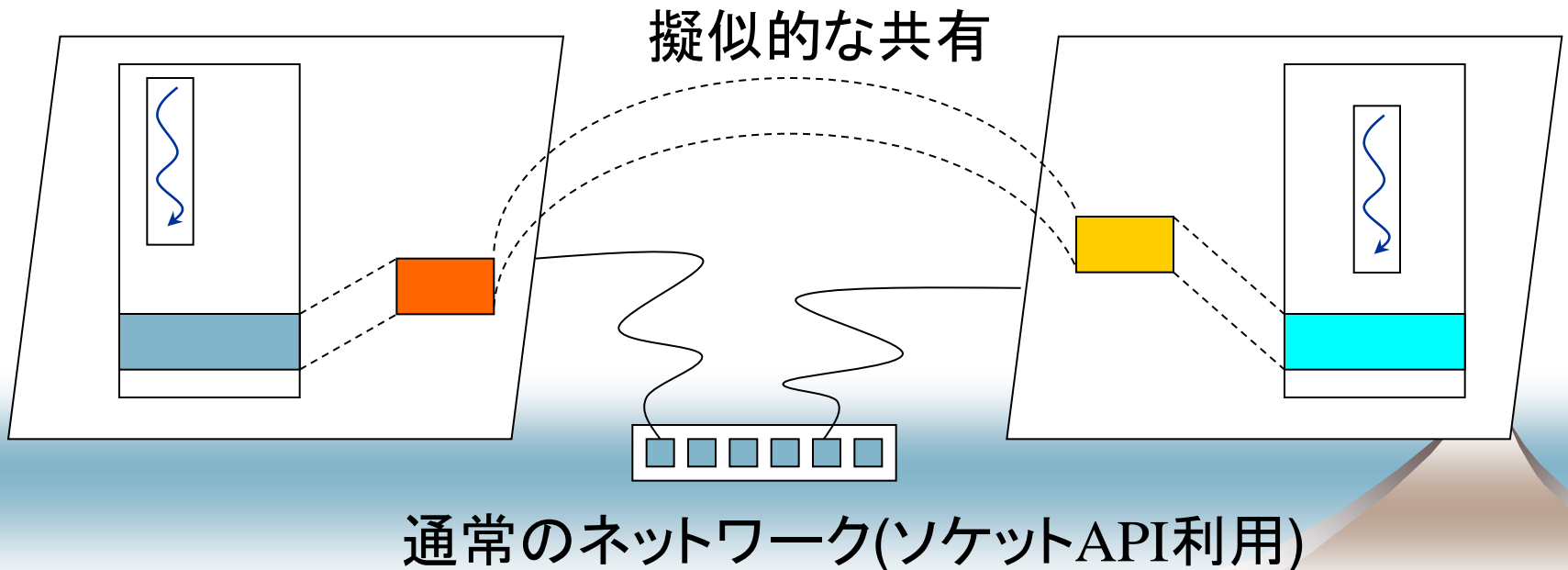
## ◆ プロセス間共有メモリ

- 物理メモリ共有, アドレス空間は分離
- 明示的なAPI (メモリマップドファイル + MAP\_SHARED)によって, ことなる論理ページを同一の物理ページにマッピング



# 仮想共有メモリ

- ◆ 異なる (物理メモリを共有していない) 計算機間で「あたかも」メモリを共有しているかのような錯覚を与える技術

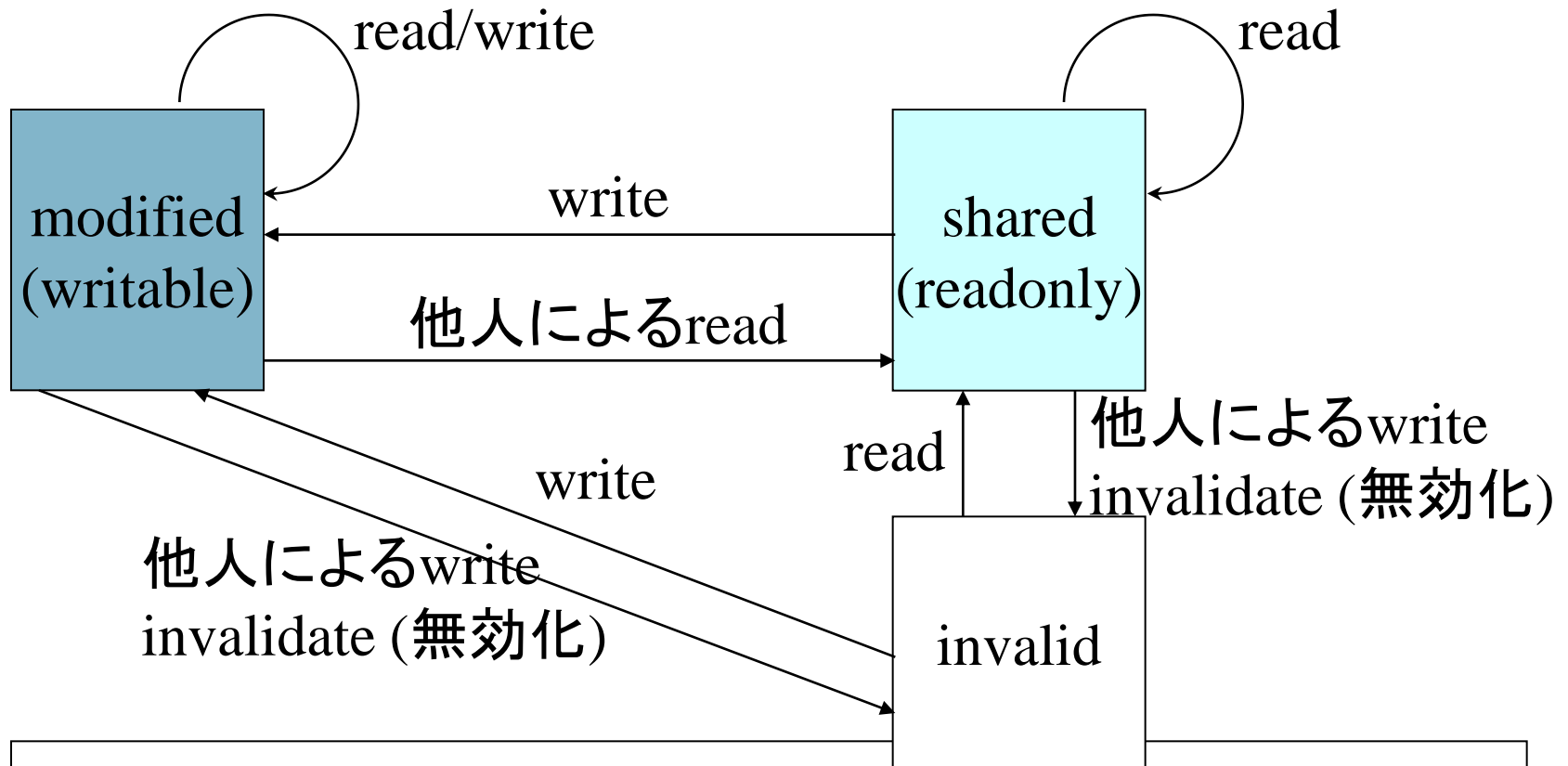


# そもそも共有メモリとは

- ◆ 通信の一形態
- ◆ 「メモリへの書き込み」結果を(明示的な通信プリミティブの呼び出し無しに)自動的に他のプロセス・スレッド・計算機に反映する
- ◆ 仮想共有メモリ・分散共有メモリ:
  - (最も単純には)すべてのメモリ書き込みを捕捉できれば実現可能 ⇒メモリ保護機能を使う



# 各ページの状態とその状態間遷移



◆ 不変条件: 各ページについて,

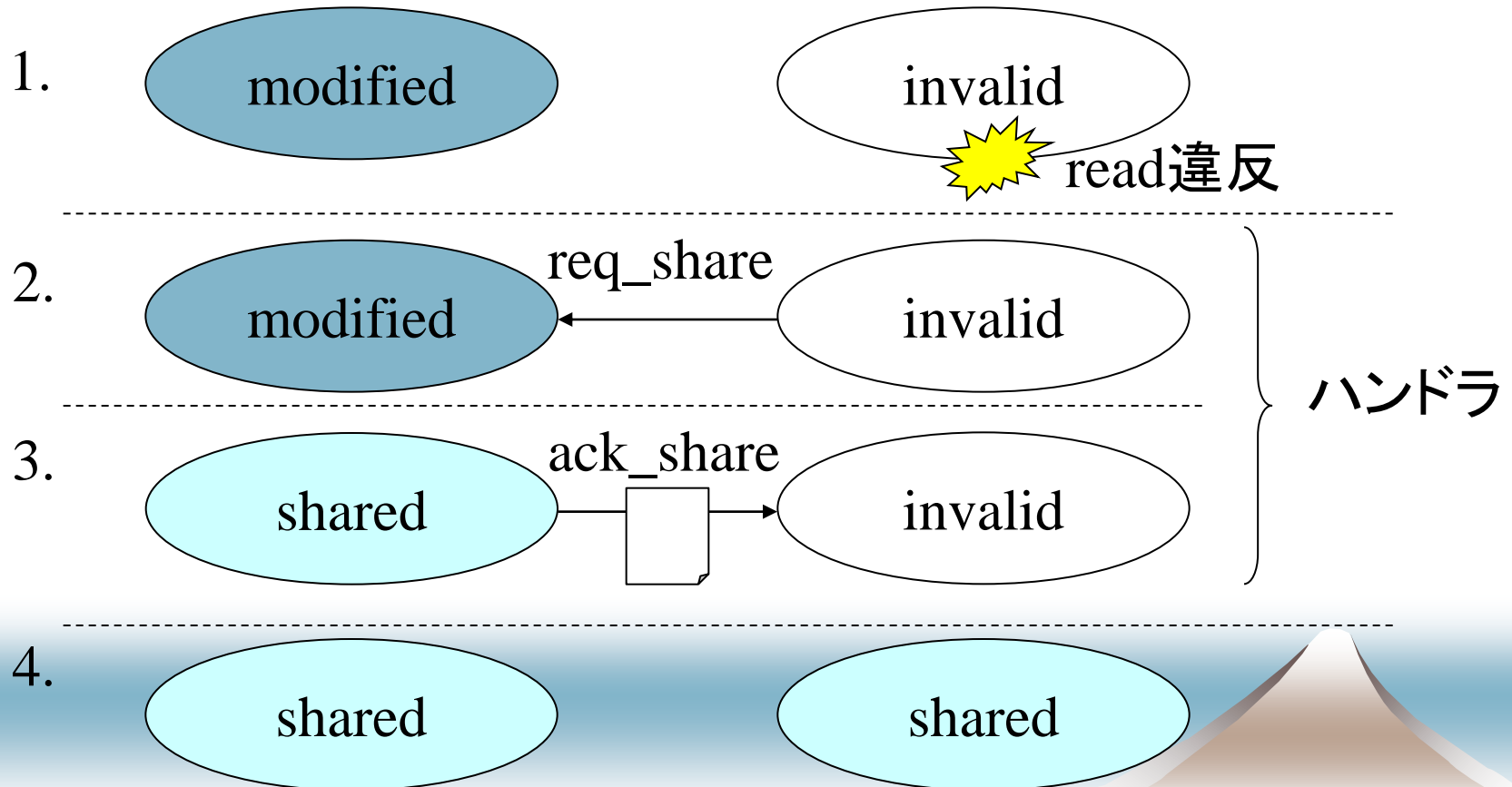
“1 modified +  $(n - 1)$  invalid”または“0 modified”

# ページの状態

- ◆ Invalid : ページはアクセス不可
  - 他のプロセス(1つ)が同一ページをmodifiedで保持しているかもしれない
- ◆ Shared : ページはread可, write不可
  - 他のプロセス(任意個)が同一ページをsharedで保持しているかもしれない
- ◆ Modified : ページはread/write可
  - 他のプロセスは同ページを保持していない(全員invalid)

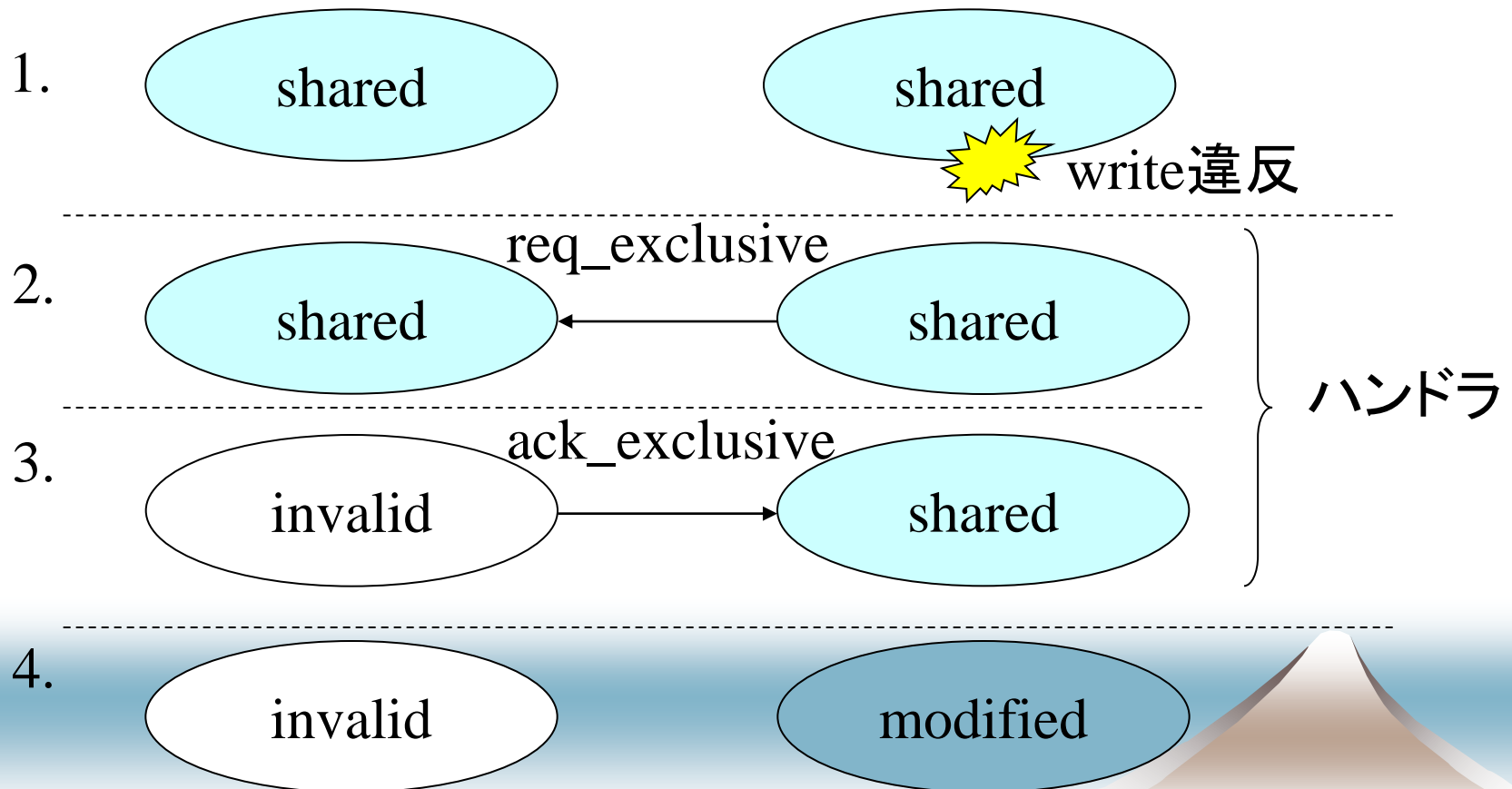
# Read違反

- ◆ invalidなページを読もうとしたときに発生



# Write違反

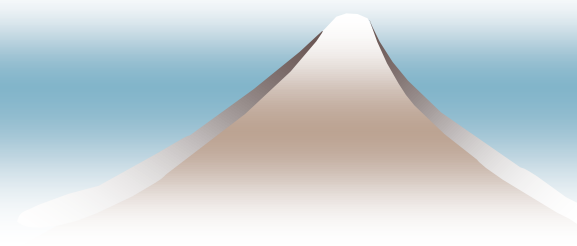
- ◆ invalid/sharedなページを読もうとしたときに発生



# 応用例5:

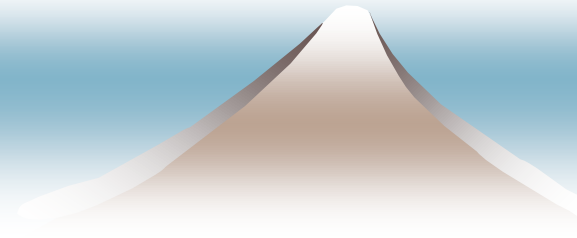
## Incremental Garbage Collection

- ◆ 目的: Garbage Collection (GC; 自動メモリ管理)の「停止時間」を短くする
- ◆ 以降の話
  - GCの基本
  - GCの停止時間
  - Incremental GCの原理
    - write barrier



# GCとは?

- ◆ malloc/free (new/delete)に代わる自動メモリ管理の一種
- ◆ 「今後もう使われない領域」を自動的に検出して解放(free)



# 自動メモリ管理(ごみ集め; Garbage Collection)

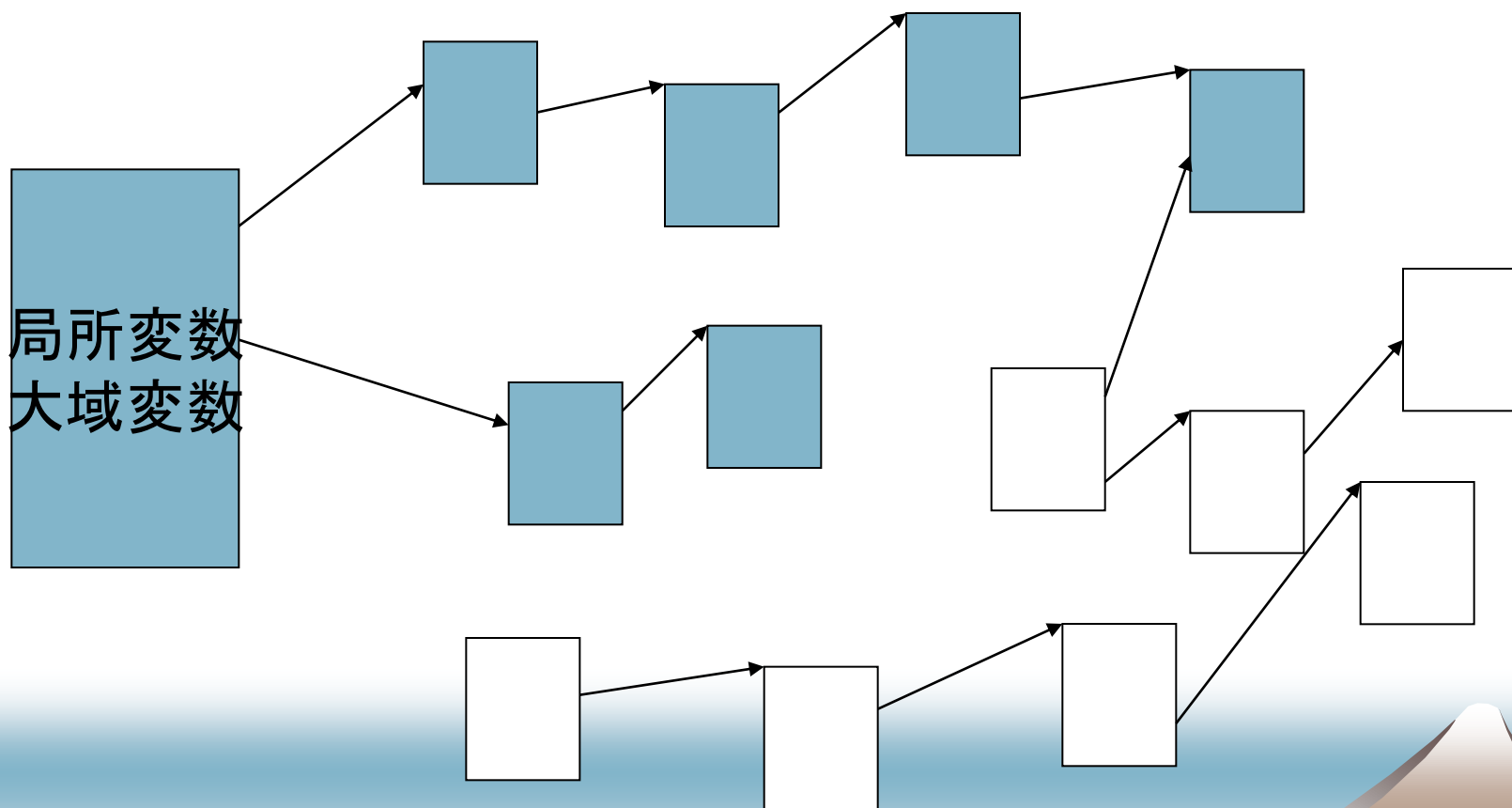
- ◆ 「もう使われない領域」を自動的に発見して解放
- ◆ プログラムの安全性は格段に向上する
  - 最近のほとんどの言語に備わる機能
  - C/C++用にはライブラリ(Boehm GC)がある
- ◆ GC用語
  - (プログラム実行中のある時点で)「**生きている領域**」  
= その時点以降使われる(アクセスされる)領域
  - 反対語: 「死んでいる領域」

# 生きている・死んでいる領域を見つけるのにGCが行っている近似

- ◆ 現在局所・大域変数に入っているアドレス(が指すメモリ領域)は「生きている」
  - 注: アドレスが指す「メモリ領域」≡ そのアドレスを含む, 一回のメモリ割り当てで割り当てられた領域
- ◆ ある「使われる」メモリ領域中に入っているアドレス(が指すメモリ領域)は「生きている」
  - 配列の要素, 構造体のフィールドなど
- ◆ 「今後使われる」∈「生きている」



要するに局所・大域変数から「到達可能」なものが「生きている」



# GCの基本原理

- ◆ 割り当て (e.g., `malloc(sz);`)
  - 空き領域から $sz$ バイト分の連続した空き領域( $a$ )を発見
  - $[a, a + sz)$  を使用中と記録 ( $a$  をキーとして探索すると $sz$ が分かるよう, 何らかのデータ構造に記録しておく)
- ◆ このとき, 空き領域が足りなくなったら(基準は様々)GCを起動
- ◆ `GC_MALLOC()` {  
    if (GCLした方がよい) { `GC();` }  
    空き領域を見つけてreturn;  
}

# マーク

- ◆ 生きているものすべてに「印」をつける
  - 印の場所: オブジェクトの先頭に専用の領域を作っておく, ハッシュ表などを作る, などがある
- ◆ 要するにグラフ探索の要領

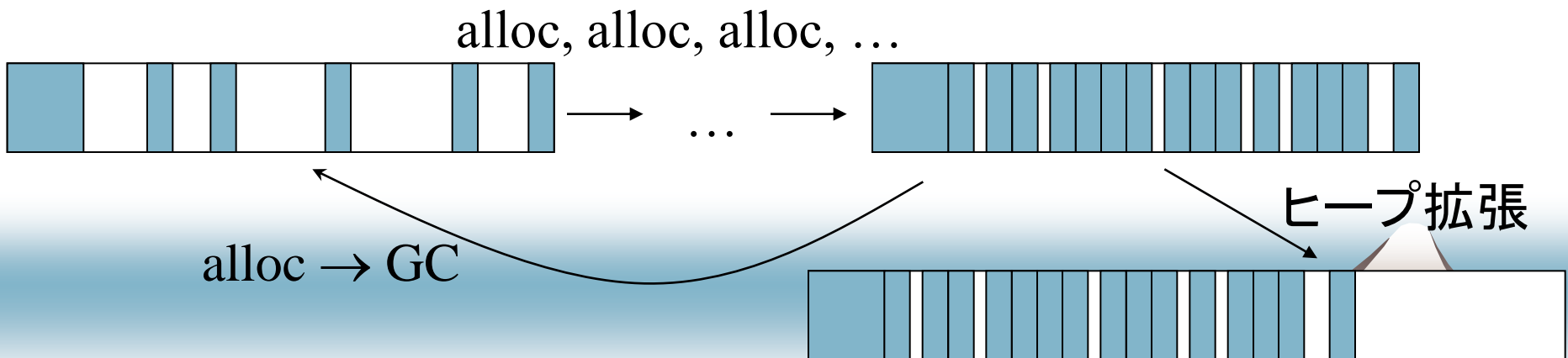


# 基本的なGCアルゴリズム (mark-and-sweep GC)

- ◆ GC() {  
    mark\_phase();  
    sweep\_phase(); }
- ◆ mark\_phase() {  
    root (局所変数, 大域変数)から, ポインタの鎖  
    をたどってたどれるオブジェクトをすべて見つけ  
    る(mark) }
- ◆ sweep\_phase() {  
    markされていないオブジェクトを全部解放 }
- ◆ 注:「オブジェクト」= 1回のメモリ割り当てで割り  
    当てられたメモリ領域

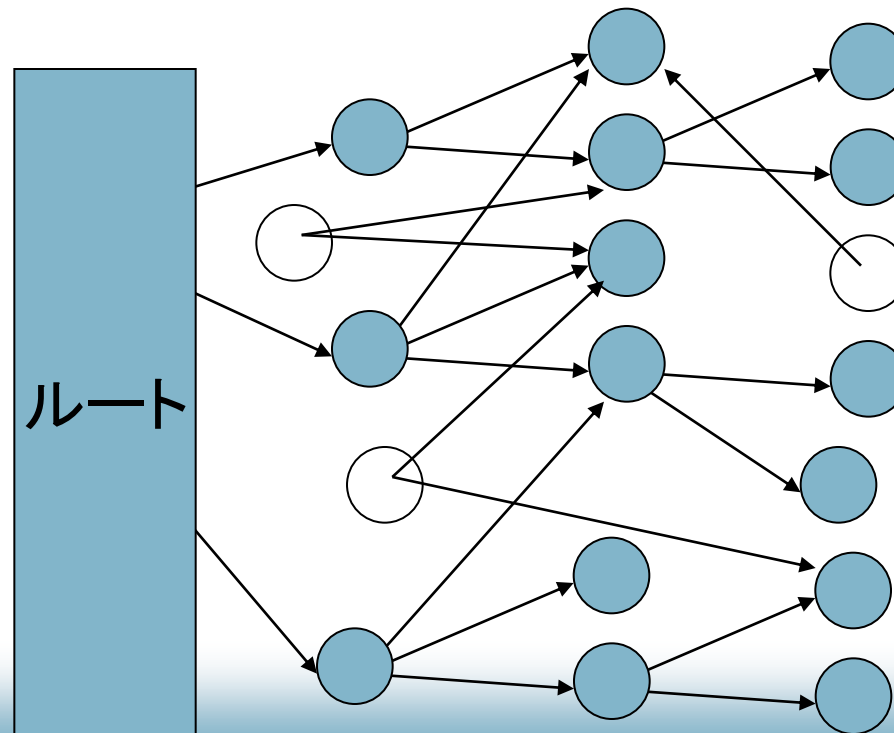
# GCはいつ起動されるのか？

- ◆ メモリ割り当て要求時に, 「自分が管理しているヒープ領域」が満杯になったら...
  - 選択肢1: GCを起動
  - 選択肢2: OSからメモリを獲得(ヒープ拡張)



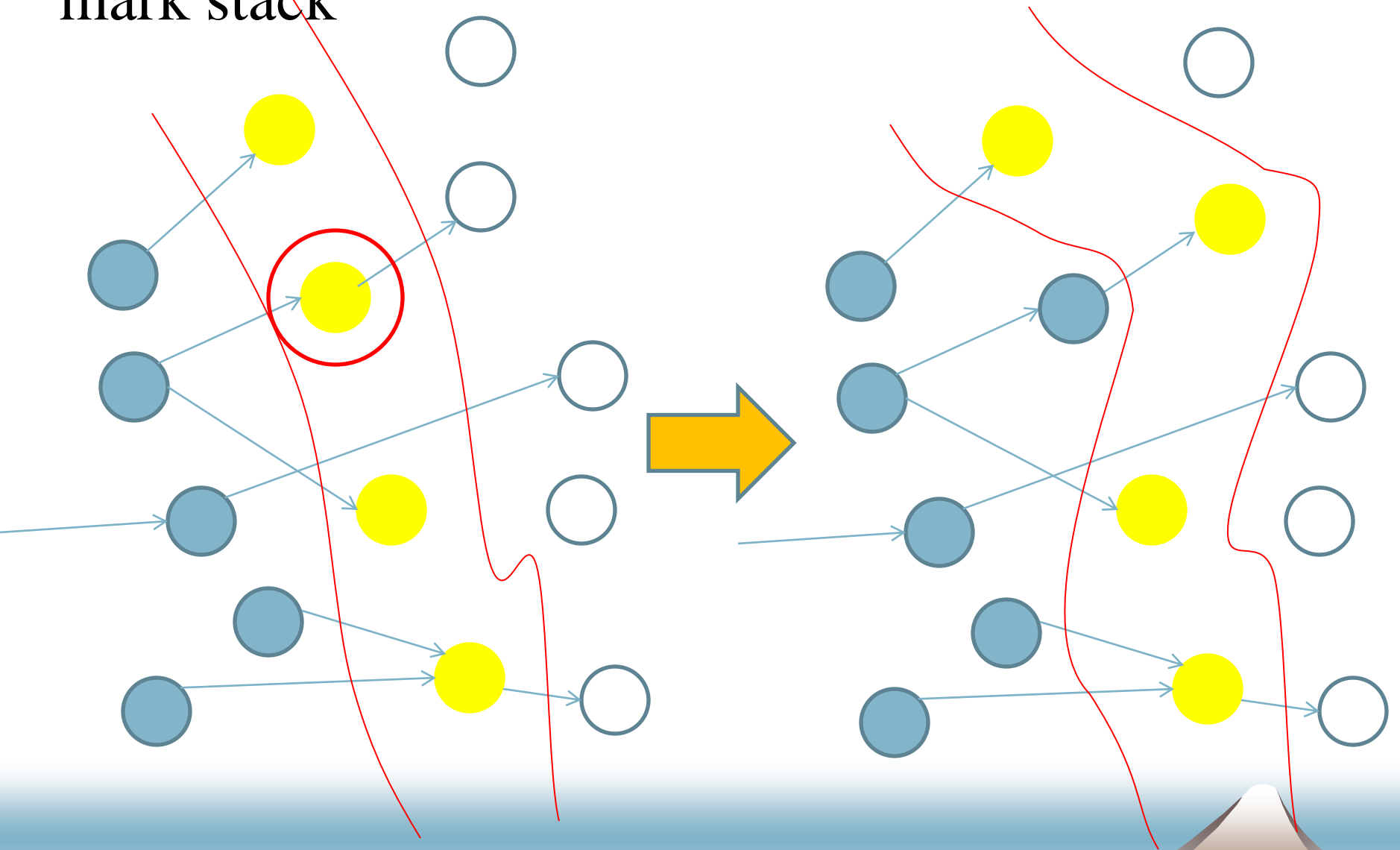
# markフェーズ

## ◆ 実質的にはグラフの探索



- ◆ `mark_phase()` {  
    `mark_stack = empty_stack();`  
    for all pointer `p` in `ROOT` { `mark(p);` }  
    while (`!empty(mark_stack)`) {  
        `p = pop(mark_stack);`  
        for all pointer `q` in object pointed by `p` {  
            `mark(q);`  
        }  
    }  
}
- ◆ `mark(p)` {  
    if (`already_marked(p)`) return;  
    set a flag indicating `p` is already marked;  
    `push(p, mark_stack);`  
}

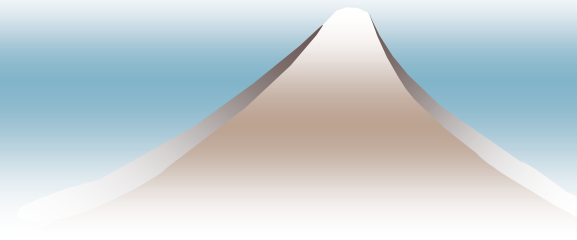
mark stack





# 1回のmarkフェーズにかかる時間

- ◆ 実行時間はほぼ、「markされたオブジェクトの量(バイト数)」に比例
- ◆ 大きなデータを用いるプログラムは一回のmarkフェーズ(つまりGC)にかかる時間が長くなる
- ◆ 通常のGCアルゴリズムではその間ユーザプログラムが停止している
- ◆ ⇒ つまりメモリ割り当てに、時々非常に時間がかかる



# Incremental GC

- ◆ markフェーズを「少しずつ」行う
  - 例: 1度のメモリ割り当て毎に一定量(e.g., 50KB)markする

通常



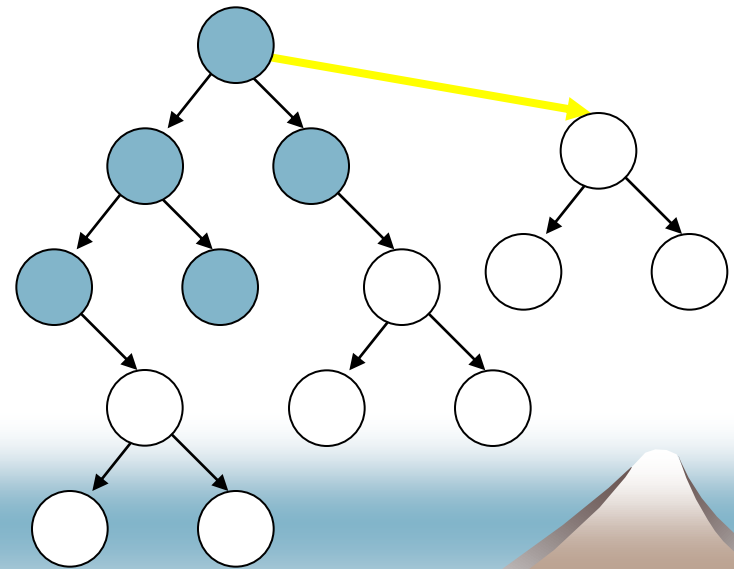
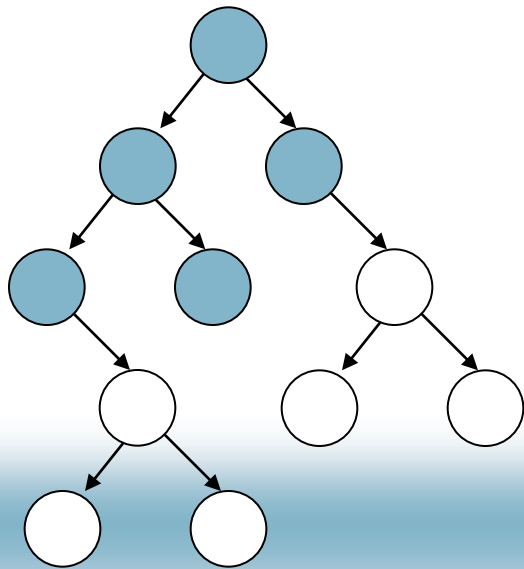
incremental



少しmark = 少しオブジェクトグラフを探索

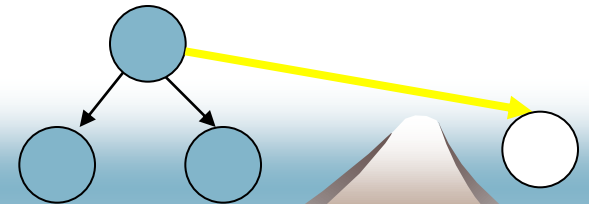
# 難しさ

- ◆ (GCの側から見ると) markとmarkの間にオブジェクトグラフが変化している



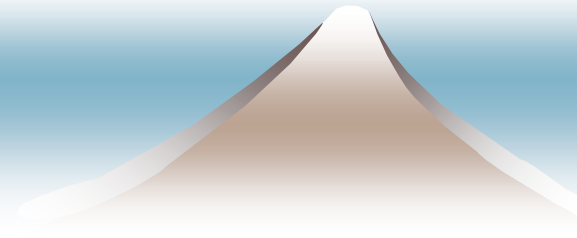
# 問題と解決方法

- ◆ 問題: すでにGCが探索済みであるようなオブジェクトへ, 他のオブジェクトへのポインタが新たに書き込まれた場合
- ◆ 解決方法 (write barrier):
  - 「そのような書き込み」を見つける
  - 書き込まれた探索済みオブジェクトを再び「未探索」とみなす



# Write Barrier

- ◆ 方法1: コンパイラ(言語処理系)にポインタの書き込み時に実行される特別なコードを挿入させる
- ◆ 方法2: 仮想記憶APIを用いて, (read-onlyに設定)書き込みを検出・記録する



# 仮想記憶APIによるwrite barrier を用いたincremental GC

- ◆ 新しいmark phase開始時:
  - 全ページを read-only に設定
- ◆ 各 allocation 時:
  - Mark phase中であれば一定量のオブジェクトをmark
- ◆ 書き込み検出時:
  - 書き込まれたページを dirty page 集合に追加
- ◆ Mark phase終了時
  - Dirty page 集合から再びmark