ファイルシステムAPIと
メモリマップドファイル

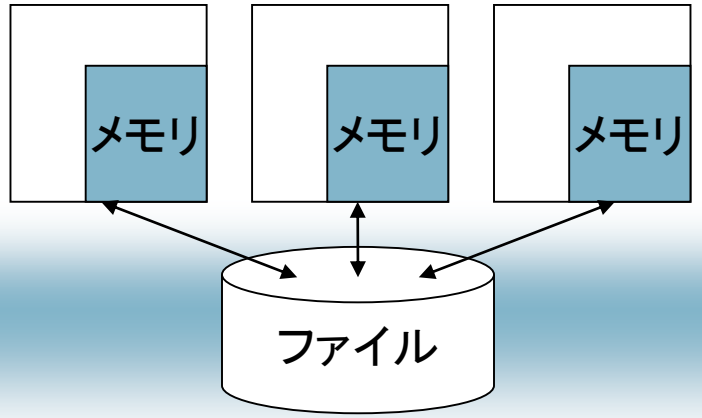
ファイルシステムの役割(1)

- 様々な種類の2次記憶装置へ,
  - 簡便で
  - 効率的で
  - 安全で
  - 統一的な(装置によらない)

読み書き手段を提供する

ファイルシステムの役割(2)

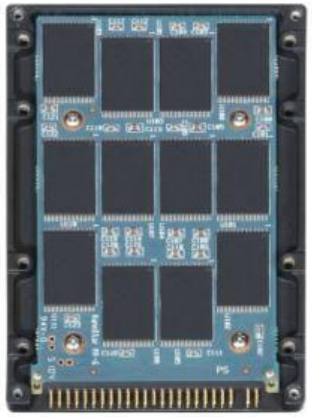
- 電源を切っても失われない情報の(ほとんど唯一の)格納場所
  - メモリの内容は電源を切ると失われる
- プロセス間で情報を共有する自然な場所
  - プロセス間でメモリは分離されていた



OSがない状態での2次記憶

- ハードディスク(HDD), Solid State Drive (SSD), USB メモリ, etc.
  - 固定サイズの「ブロック」の集合(典型: 512B, 1KB)
  - ブロックのアドレス: (シリンダ, トラック, セクタ)または全ブロックの通し番号(LBA)
  - 読み書きのインタフェース
    - I/Oコマンド(特権命令)発行
    - 終了通知の割り込み

HDD



SSD

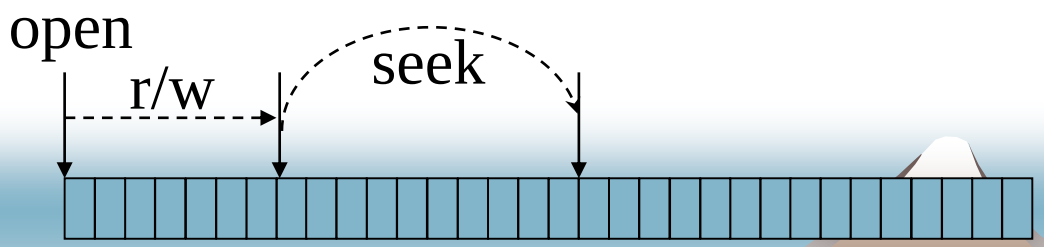
http://blog.dubspot.com/dj-producer-tools-ssds-solid-state-drives-wave-of-the-future-should-you-get-one/

OSが提供する抽象化: ファイル

- ファイル名(パス名)
  - ややこしいアドレスではなく自由な文字列
- 各ファイルをバイト配列として簡便に読み書き
  - ⟨ファイル名, オフセット⟩ → 記憶場所
  - キャッシュを用いた効率的アクセス
- ファイルの作成，伸長
  - 空き領域確保
- ファイルの所有者，読み書き権限
  - 単一のデバイスを安全に共有・分離

API：基本概念

- 開く(open)
  - 権限の検査，以後の読み書き準備
- 逐次的な読み書き(read/write)
- 位置あわせ(seek; 頭出し)
- メモリマップドファイル(後述)
- 閉じる(close)



Unix API

- int *fd* = open(*path*, *access*);
- int *m* = read(*fd*, *buf*, *n*);
- int *m* = write(*fd*, *buf*, *n*);
- off_t *o'* = lseek(*fd*, *o*, *from*);
- int *err* = close(*fd*);

Windows API

- HANDLE *h* = CreateFile(*path*, *access*, …);
- BOOL *ok* = ReadFile(*h*, *buf*, *n*, &*m*, …);
- BOOL *ok* = WriteFile(*h*, *buf*, *n*, &*m*, …);
- DWORD *o*’ = SetFilePointer(*h*, *o*1, &*o*2, *from*);
- BOOL *ok* = CloseHandle(*h*);

C言語の標準ライブラリAPI (1)

- FILE * *fp* = fopen(*path*, *mode*);
- size_t *sz*’ = fread(*buf*, *sz*, *n*, *fp*);
- size_t *sz*’ = fwrite(*buf*, *sz*, *n*, *fp*);
- int fseek(*fp*, *o*, *from*);
- int *c* = fgetc(*fp*);
- int *c*’ = fputc(*c*, *fp*);

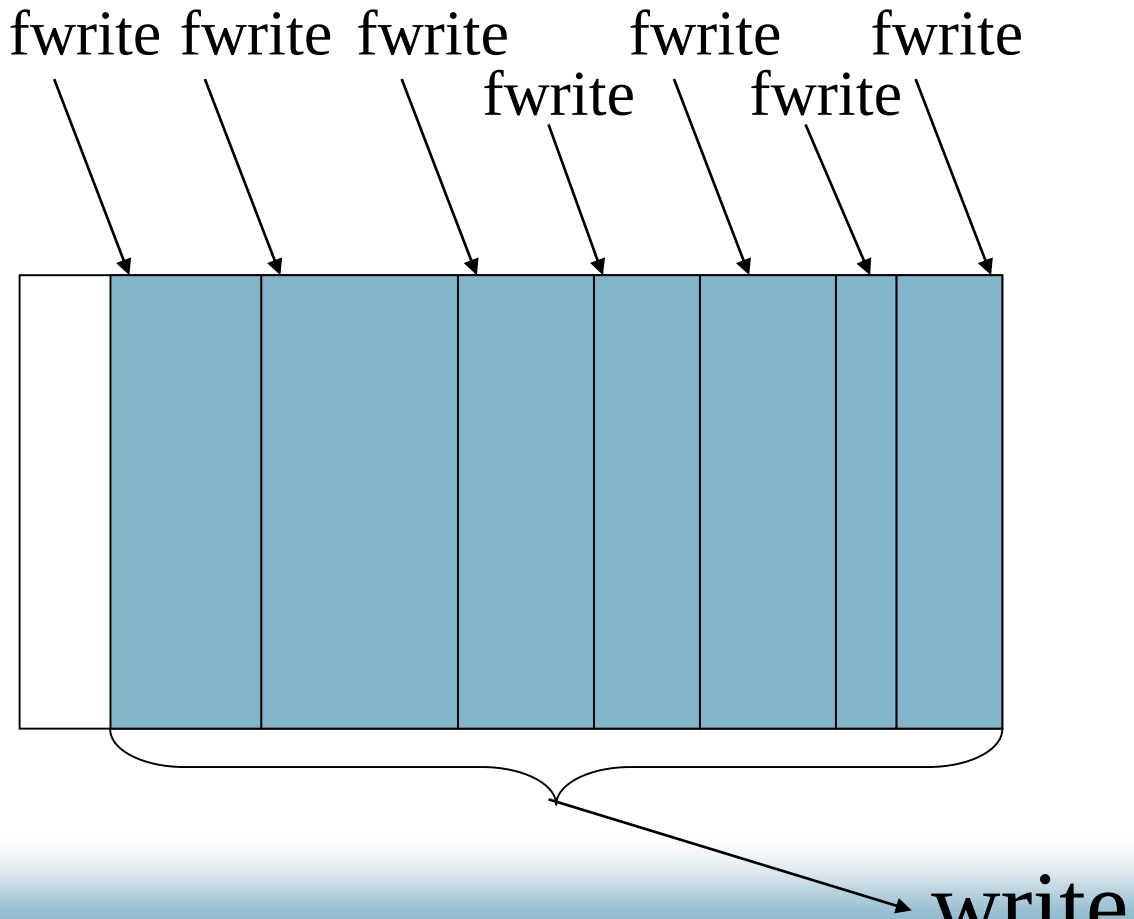
C言語の標準ライブラリAPI (2)

- char * *s*’ = fgets(*s*, *sz*, *fp*);
- int *ok* = fputs(*s*, *fp*);
- fscanf(*fp*, “%d”, &*x*); /* 例 */
- fprintf(*fp*, “%d”, *x*); /* 例 */
- int *err* = fclose(*fp*);

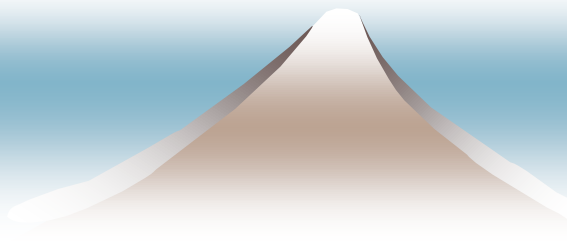
open/read/write系とfopen/fread/fwrite系の関係

- open/read/write系: システムコール
- fopen/fread/fwrite系: (結局はopen etc.を呼び出す)ライブラリ
  - 違い1: 書式付入出力(fprintf, fscanf)など高機能なIOのサポート
  - 違い2: バッファリングを行う
    - 複数回のfwriteをメモリ上に保持して一度のwriteシステムコールで書き込む
    - 一度のreadシステムコールで大量に読み込んで以降の複数回のfreadに答える

バッファリング



ファイルシステムの実装と性能



ファイルシステム実装の概要

- アドレス変換
  - 論理的な位置⟨ファイル名, ファイル内オフセット⟩→ ディスク内の位置(⟨シリンダ，ヘッド，セクタ⟩またはLBA)へ変換
- 読み出し
  - アドレス変換+ブロックの読み出し
- 書き込み
  - 必要に応じて空きブロックを割り当てる
  - アドレス変換+ブロックへの書き込み


Traditional hard disk drive


Solid state hard drive

基本事項

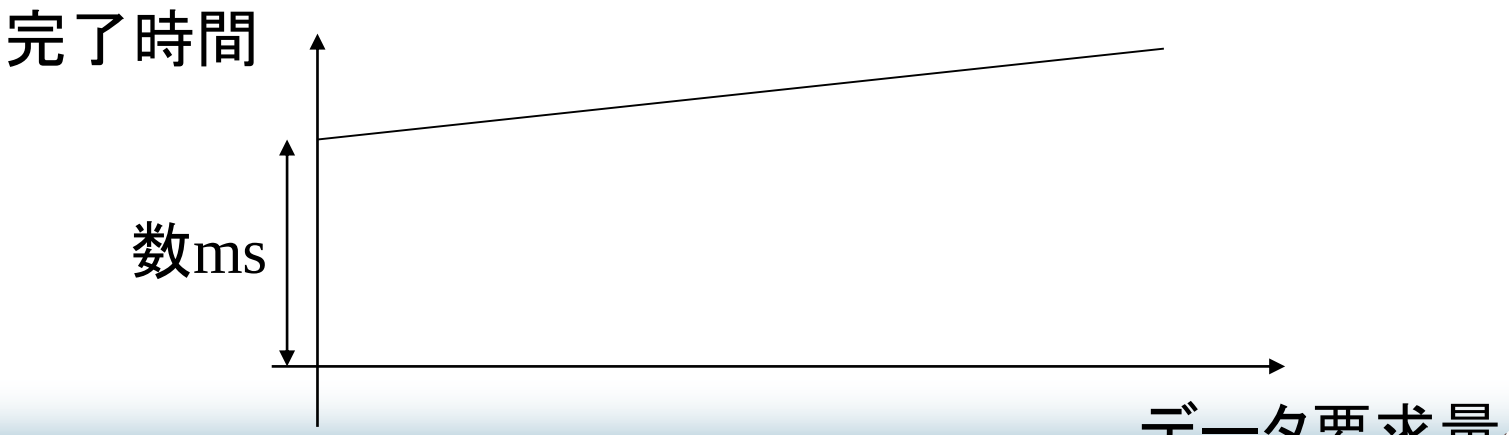
- 2次記憶のランダムアクセスは遅い
- 典型的な「遅延」(最小単位の読み書き時間)

	読み出し	書き込み
HDD	$O(10^{-2}\ s)$	$O(10^{-2}\ s)$
SSD	$O(10^{-5}\ s)$	$O(10^{-2}\ s)$
主記憶	$O(10^{-7}\ s)$	$O(10^{-7}\ s)$

- HDDが遅い理由 回転
- SSDの書き込みが遅い理由
  - 書き込みのverify, 書き込みの単位が大きい

ディスクのアクセス時間

- メモリに比べると圧倒的に遅い
- 一定オーバーヘッド(HDDの位置あわせ，回転待ちなど)が大部分を占める



OSが備えている，ディスクの遅さへの対処

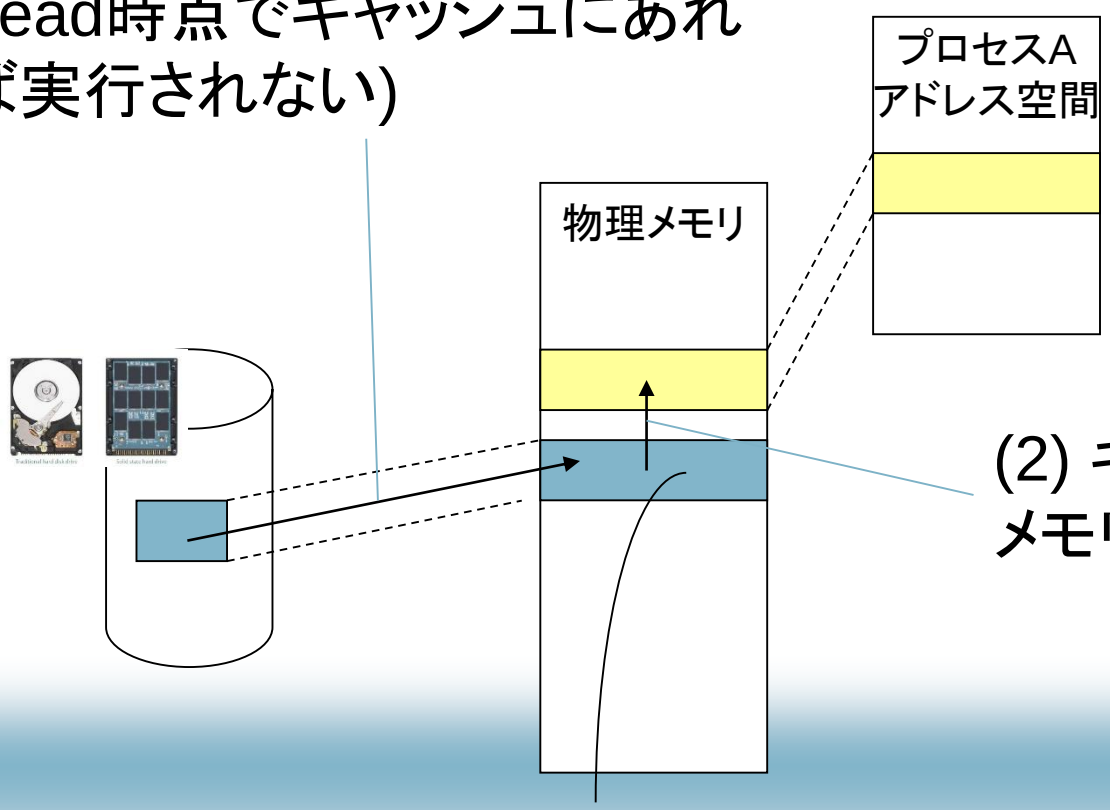
- キャッシュ・遅延書き込み
- 先読み(prefetch)
- ディスクスケジューリング(HDD)
- 空き領域管理・連続割り当て

キャッシュ

- 一度アクセスされたファイル断片をメモリに保存
- 効果
  - 複数回同じ断片を読み書きする場合, (うまくいけば)2回目以降のディスクI/Oが不要
  - 書き込みをメモリ上にいったん蓄え, 大きな単位で書き込む
- メモリが (アプリケーションによるメモリ割り当て要求，別のファイル読み書きによって) あふれた時に捨てられる(LRU)

readの動作

(1) 2次記憶 →キャッシュ(I/O)
(read時点でキャッシュにあれば実行されない)



(2) キャッシュ→ プロセスメモリへのメモリ間転送

先読み

- 近い将来アクセスが予想される部分を事前に読み込んでおく
  - 1度のヘッドの位置あわせ + 回転待ちでたくさんのデータを読む
- 近い将来のアクセスなんてわかるのか?
  - 実際には先頭からの順次アクセス(sequential scan)に対して発動されるのが典型

ファイルキャッシュとプリフェッチの効果測定

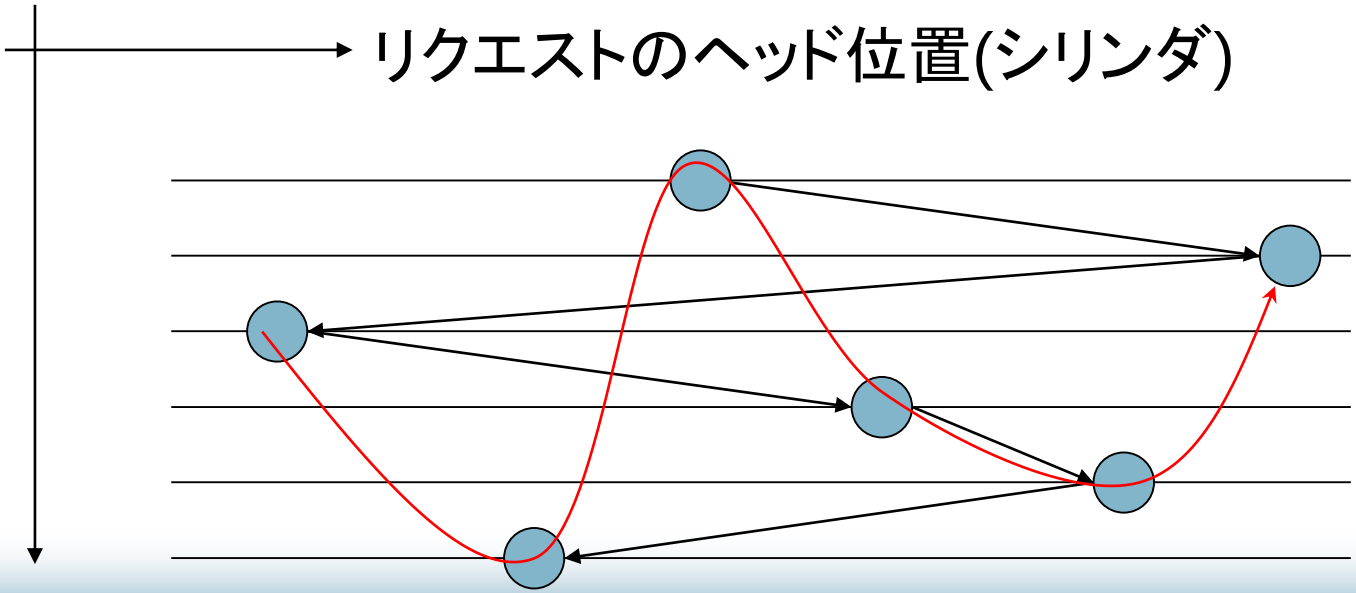
- 適当な大きさのファイル作成
- キャッシュからデータを追い出す
- 同じファイルを複数回読み出して，「時間 vs 読み出し量」測定
  - 1回目と2回目の違い(キャッシュ)
  - 逐次読み出し vs ランダム読み出し
  - 一度に行う読み出し量の違いによる変化

連続した領域への割り当て

- 一度に読み出すのに都合の良いブロック(例: 同じシリンダ(円周)内の全ブロック)にファイルの連続した領域を割り当てる
  - cf. いわゆる「デフラグツール」
- 先読みの効果を大きくする

HDDのIOスケジューリング

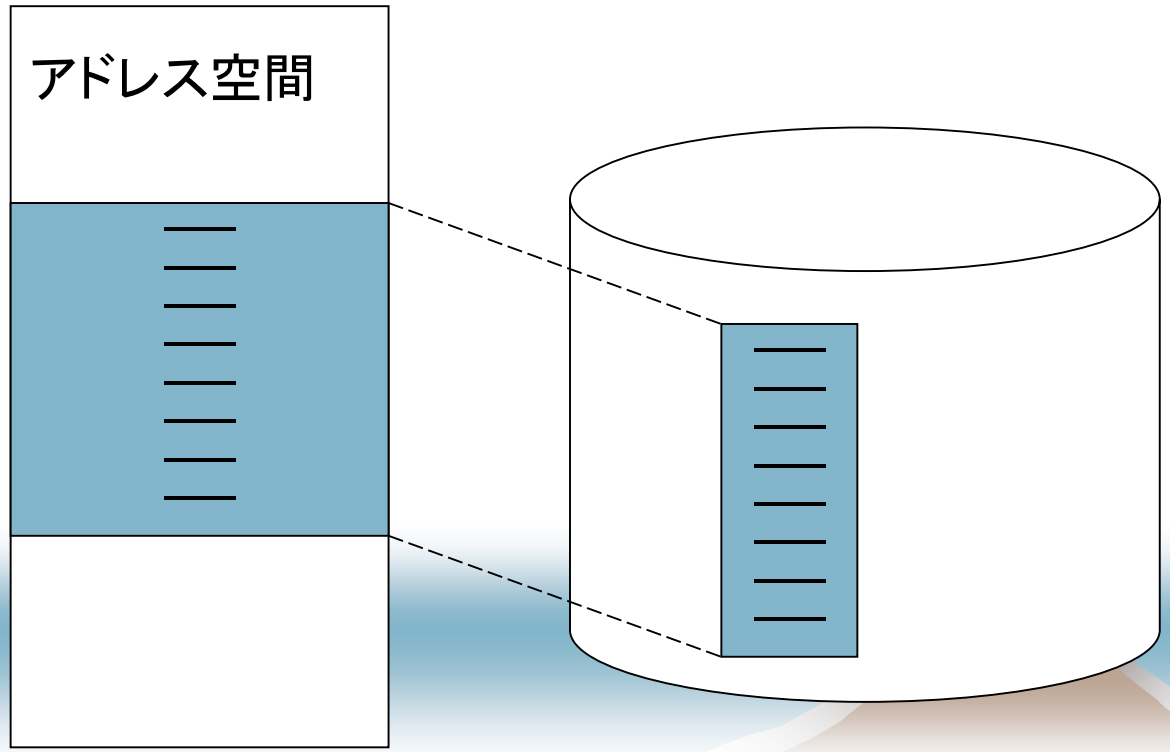
- アクセスすべきブロックを並び替えて，少ないヘッドの動きで一度に読む



メモリマップドファイル
ファイルシステムと仮想記憶の連携

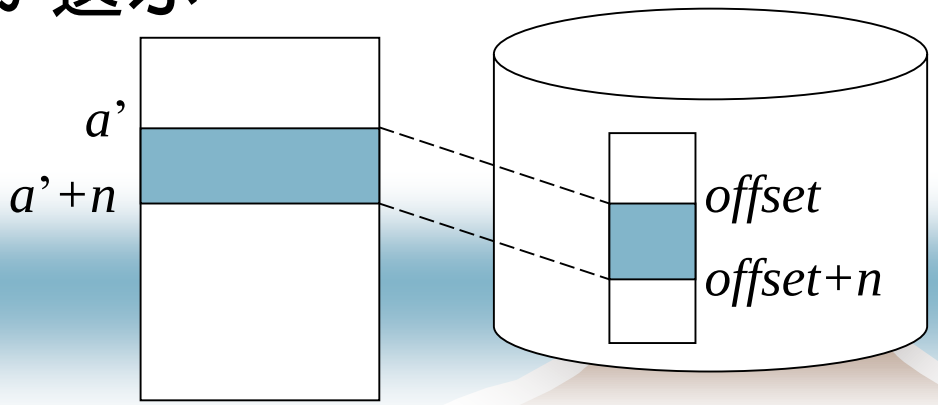
メモリマップドファイル

- ファイルを明示的なread/writeではなく「あたかもメモリの様に」読み書きするAPI



メモリマップドファイル: Unix API

- *fd* = open(*file*, *access*);
  a' = mmap(*a*, *n*, *prot*, *share*, *fd*, *offset*);
  - 意味: "*file*の*offset*バイトから始まる*n*バイトを, アドレス[*a'*, *a'* + *n*)で*access*可能にする"
    - $a \neq 0 \Rightarrow a' = a$ (空いていれば)
    - $a = 0 \Rightarrow a'$はOSが選ぶ



プライベート/共有マッピング

- パラメータ*share*
  - 複数のプロセスが同じファイルをmmapした場合の挙動を指定
  - *share* = MAP_PRIVATE
    - プロセスごとに別のコピーを見る
    - 書き込み結果はファイルに反映されず，プロセス間でも共有されない
  - *share* = MAP_SHARED
    - 書き込み結果はプロセス間で共有され，ファイルにも反映される

メモリマップドファイル: Windows API

- *h* = CreateFile(*file*, *access*, …);
  m = CreateFileMapping(*h*, …);
  a’ = MapViewOfFileEx(*m*, *prot*,
  *offset*1, *offset*2, *n*, *a*);
- *prot* = FILE_MAP_COPYでMAP_PRIVATEと似た効果を持つ

メモリマップドファイルの用途(1)

- ファイルの読み書き
  - あたかもmmapがファイルの中身すべてをメモリに読んできているかのように動作する
  - 書き込みが適宜ファイルに反映される(MAP_SHARED)

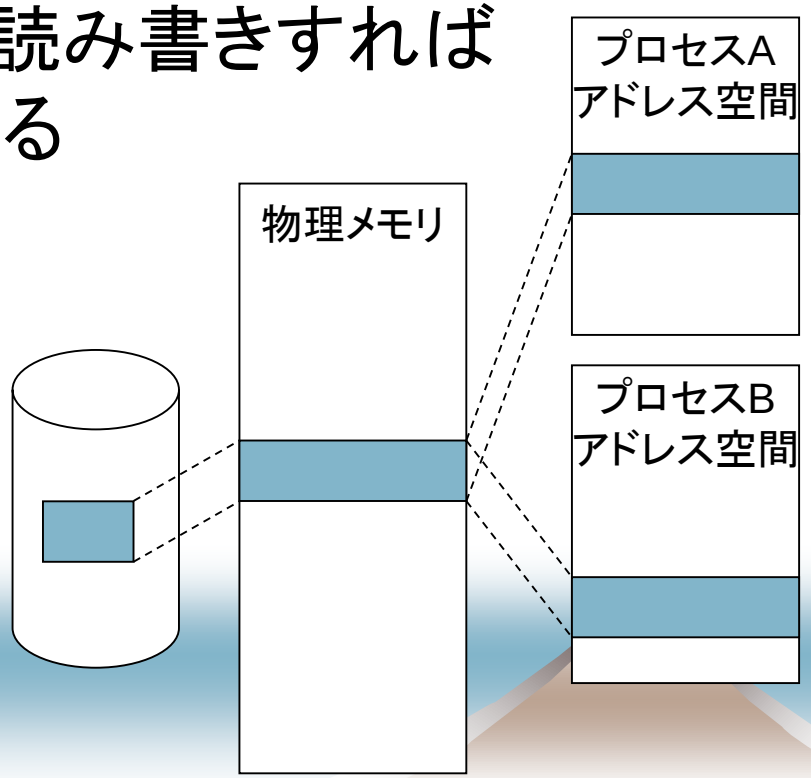
  ⇒ 特に, ファイルへの「ランダム」アクセスを行う簡便な手段

メモリマップドファイルの用途(2)

- メモリの割り当て
  - sbrk (Unix)やVirtualAlloc (Win32)に代わるメモリ割り当て手段
    - Unix : fd = –1, flagsにMAP_ANONYMOUSを渡す, または特別なファイル/dev/zeroをMAP_PRIVATEでmmapすると，特定のファイルに結びついていないメモリ領域を得る
    - Win32 : INVALID_HANDLE_VALUEをCreateFileMappingに渡すと同様の効果

メモリマップドファイルの用途(3)

- プロセス間共有メモリ
  - 同じファイルの同じ部分を複数のプロセスがMAP_SHAREDで読み書きすれば更新が互いに反映される



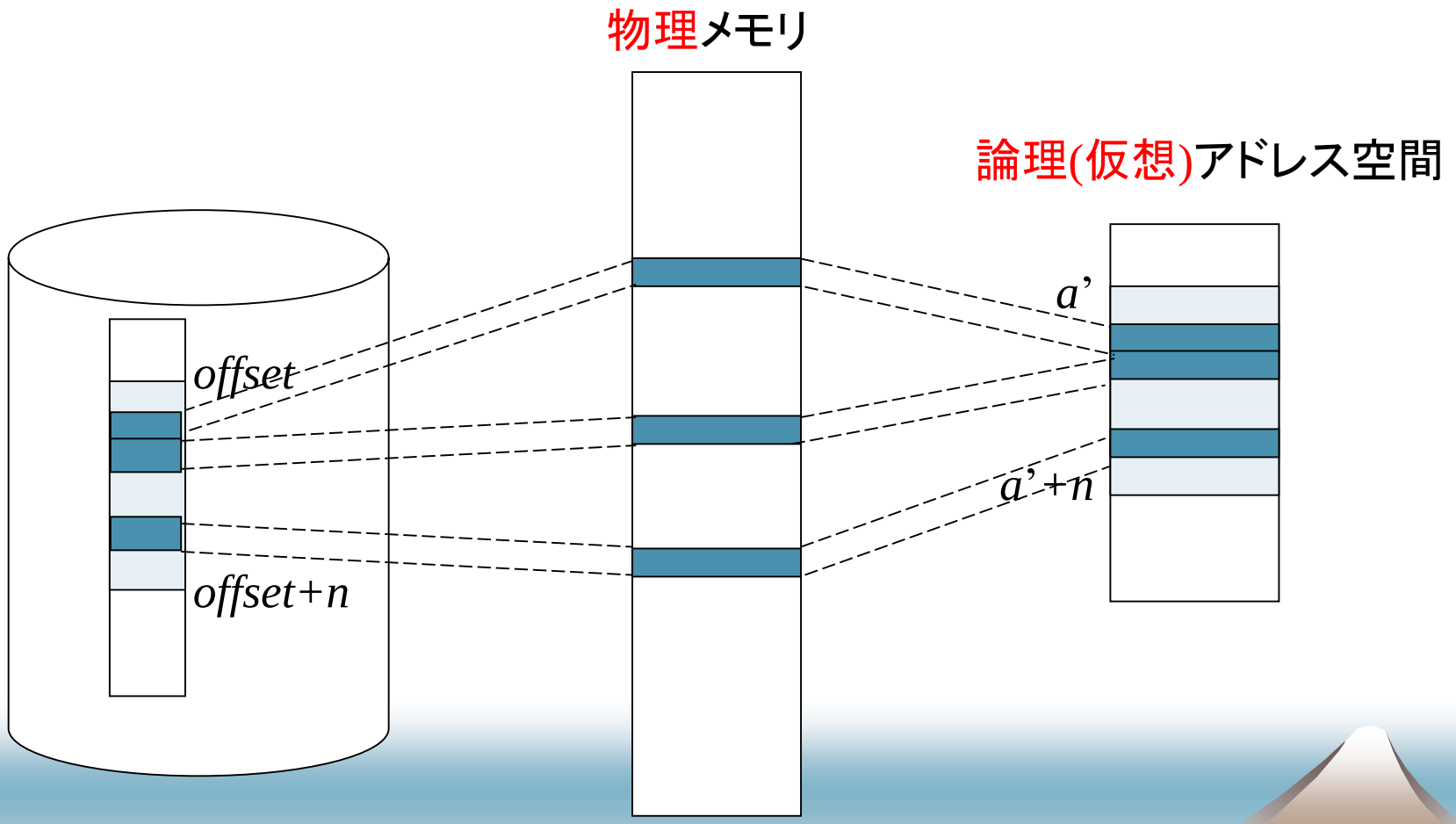
メモリマップドファイルの仕組み(1)

- mmap/MapViewOfFile etc.の実行時にファイルの中身をすべて読むわけではない
- あるページが初めてアクセスされた際に，ページフォルトが発生 ⇒ OSが対応するファイルから内容を読み込む
- ページへの書き込み ⇒ 適当なタイミングで元のファイルに反映

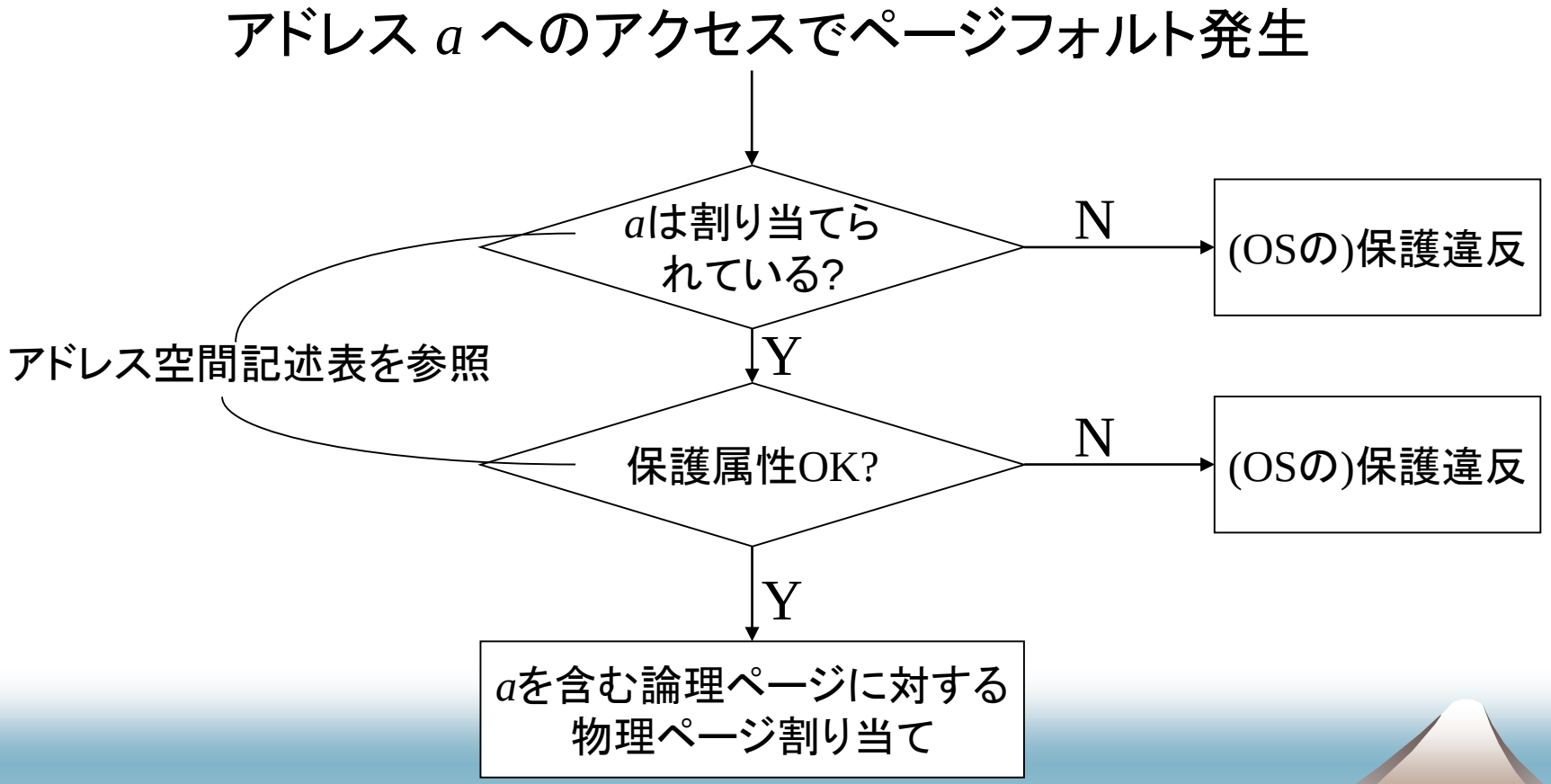
メモリマップドファイルの仕組み(2)

- OSにとっては，メモリ管理(仮想記憶)機構の自然な延長
  - メモリの退避場所としてページング/スワップ領域の変わりに通常のファイルを使うだけ

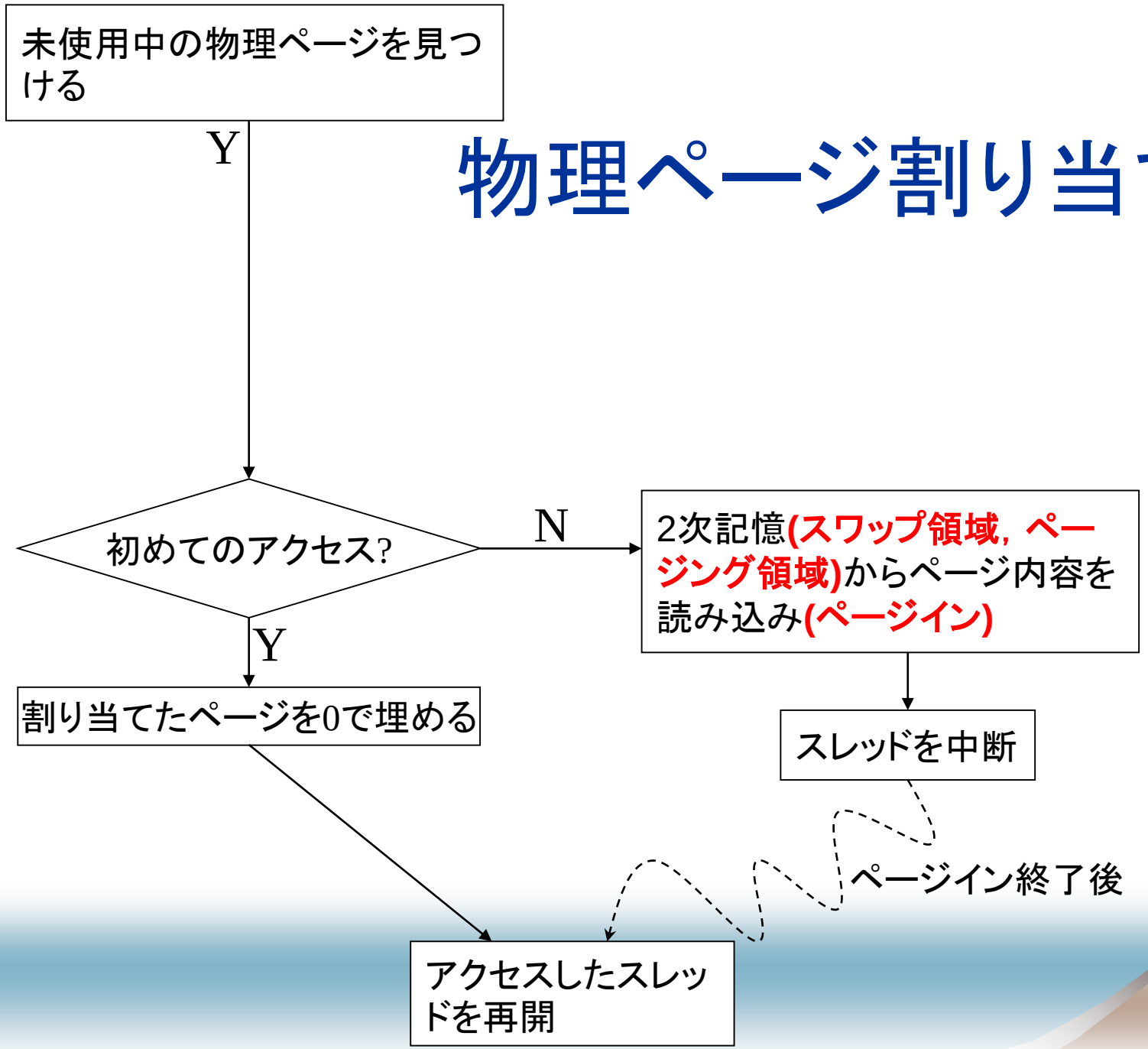
mmap動作図



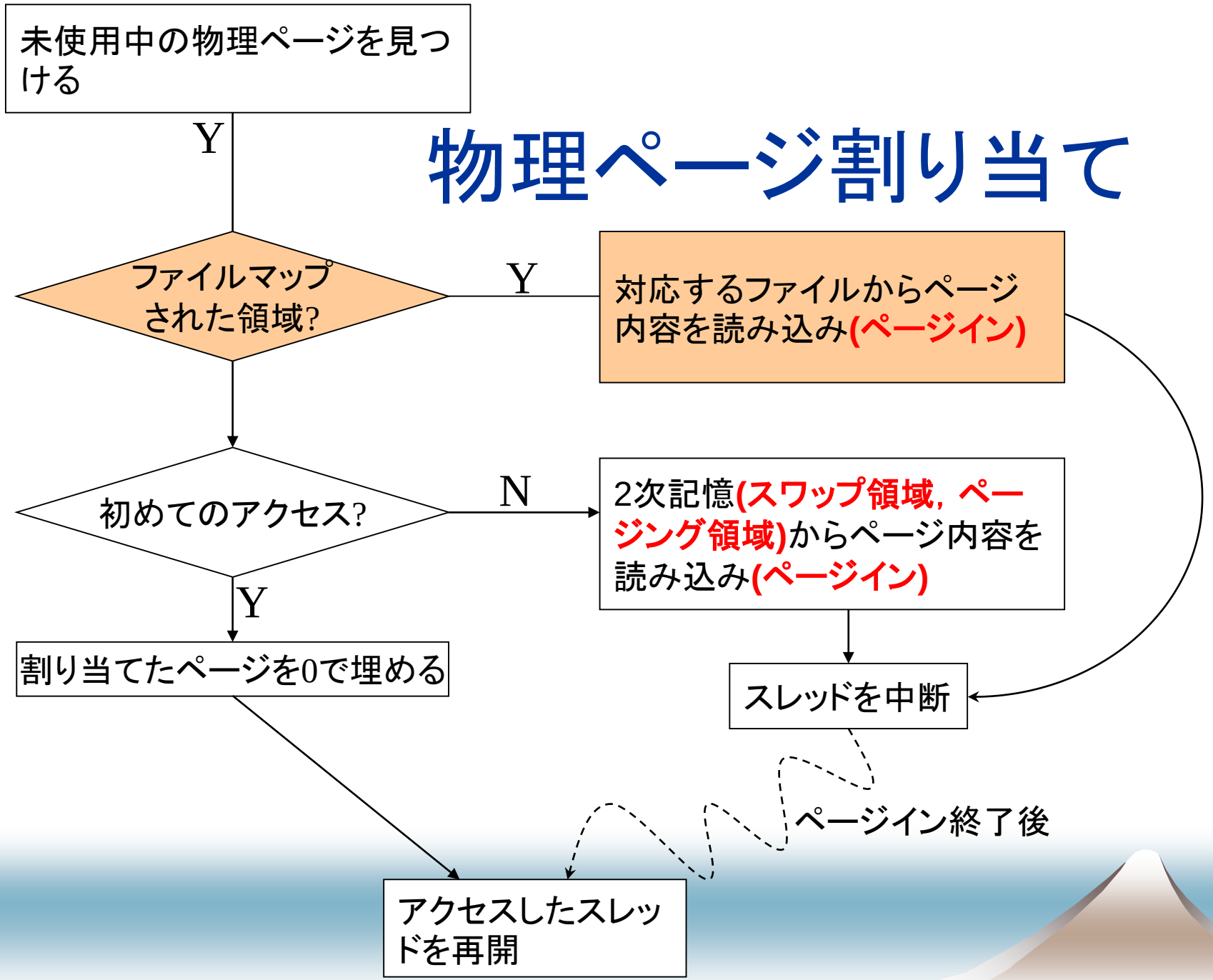
メモリマップドファイルの仕組み(2) ―ページフォルト処理(復習)



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
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)
Y
割り当てたページを0で埋める
スレッドを中断
ページイン終了後
アクセスしたスレッドを再開

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
ファイルマップされた領域?
Y
対応するファイルからページ内容を読み込み(ページイン)
初めてのアクセス?
N
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)
Y
割り当てたページを0で埋める
スレッドを中断
ページイン終了後
アクセスしたスレッドを再開

mmapシステムコール内の動作

- アドレス空間記述表へ，新たにmmapされた領域を記録する(だけ)
  - 後のpage fault時に実際のIOを発動する

mmapとreadの性能挙動観察

- 大きなファイルの全内容を次の二通りの方法で大きな配列に読み込む
  - mallocしてその領域にread
  - mmap

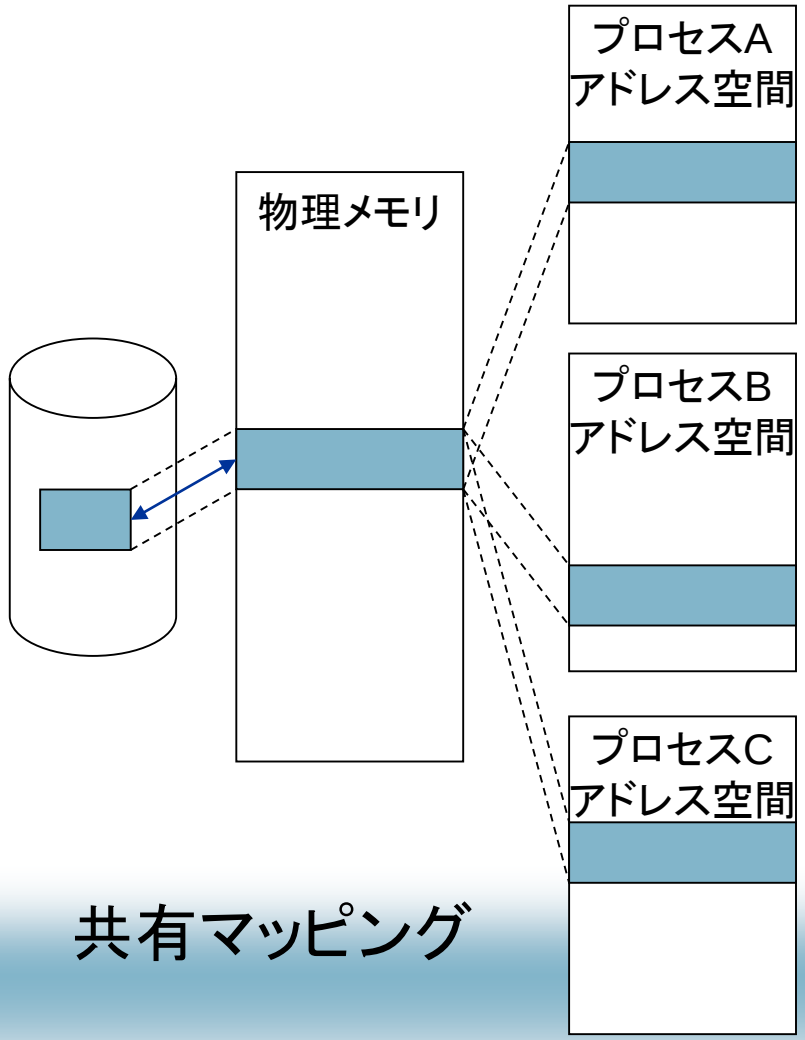
メモリマップドファイルが有効な場面 (1)

- 大きなファイルの一部だけをランダムアクセスする場合
  - すべてをメモリマップするだけで後はメモリの読み書きと同じようにアクセスできる
    - 実際のファイルへのアクセスは個々のページを初めて触るまで行われない(cf. readの場合)
  - 実はプログラムコード(特にライブラリ)はメモリマップドファイルを利用して共有されている

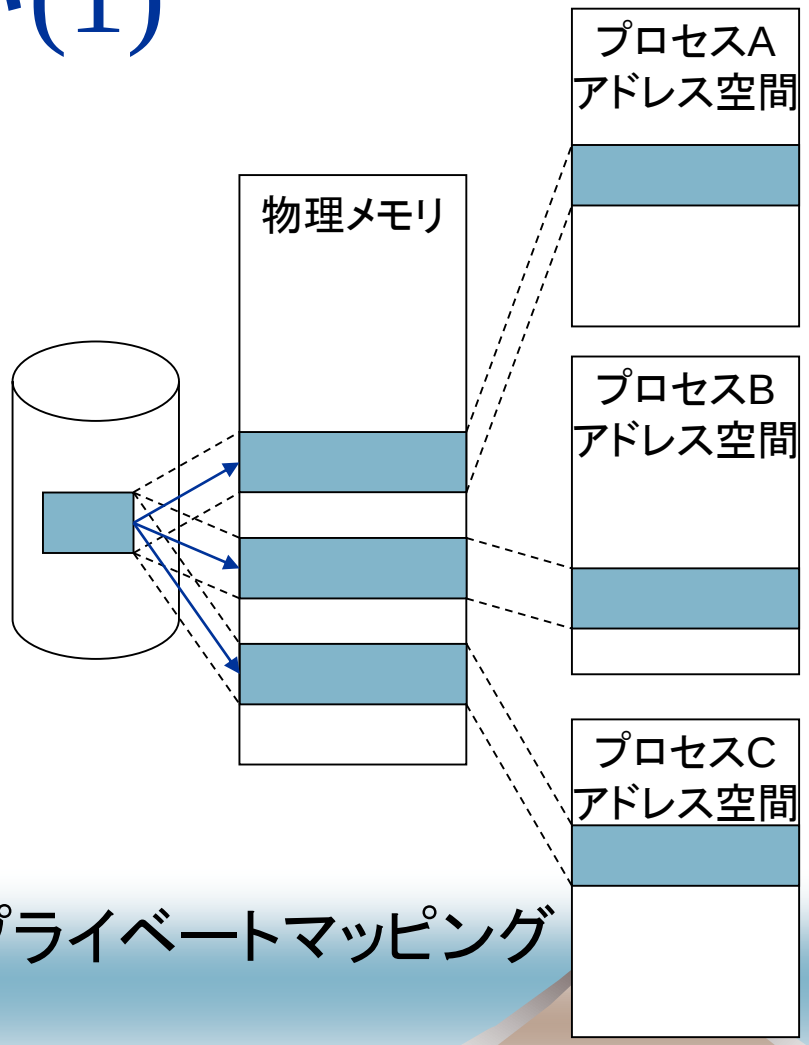
まめ知識

- strace：Linuxでプロセスが発しているシステムコールの列を表示
- strace ⟨コマンド名⟩

プライベート/共有マッピングの違い(1)



共有マッピング



プライベートマッピング

プライベート/共有マッピングの違い(2)

- プライベート(Unix mmapのMAP_PRIVATE) :
  - (基本的には)マッピングの数だけ物理メモリを消費
- 共有(Unix mmapのMAP_SHARED) :
  - すべてのマッピングで物理メモリを共有

⇒ (意味の違いを度外視すれば)共有のほうが物理メモリの利用効率が良い

OS内部のマッピングの最適化

- 読み出し専用マッピング
- Copy-on-writeマッピング
- 考え方: 可能な限りマッピング(物理メモリ)を共有する

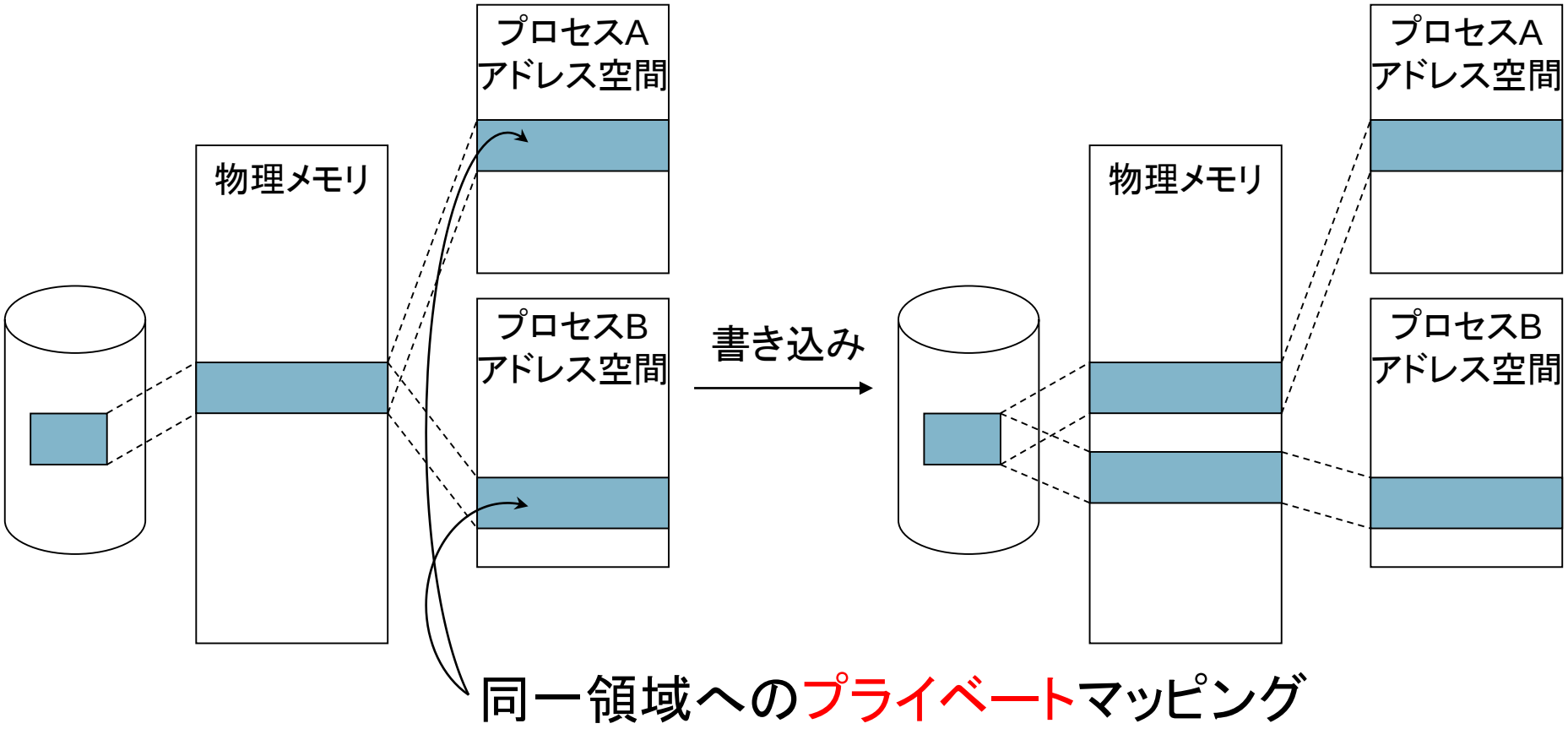
読み出し専用マッピング

- 当然ながら常に(プライベートマッピングであっても)物理メモリを共有できる
- 典型的使用場面
  - プログラム開始時にプログラムテキストを読み出すために使われている

Copy-on-writeマッピング(1)

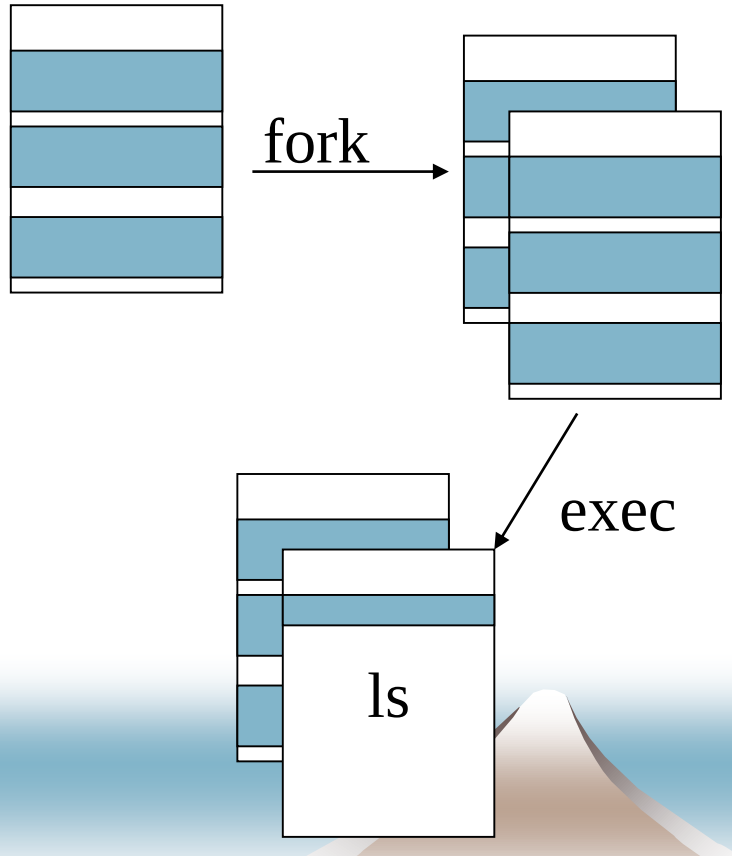
- 書き込み可でマップされた領域も，実際に書き込まれるまでは物理メモリを共有しておく
  - (ページテーブル，TLBの)保護属性を「書き込み不可」にしておく
  - 最初の書き込み発生時にCPU保護例外が発生．この時点でOSが新しい物理ページを割り当て，コピーを作る

Copy-on-writeマッピング (2)



Copy-on-writeの別の応用
高速fork (1)

- fork : アドレス空間のコピー
- pid = fork();
  if (pid == 0) { /* child */
    …;
    execve(“/bin/ls”, …);
  } else { /* parent */
    …;
  }



Copy-on-writeの別の応用: 高速fork (2)

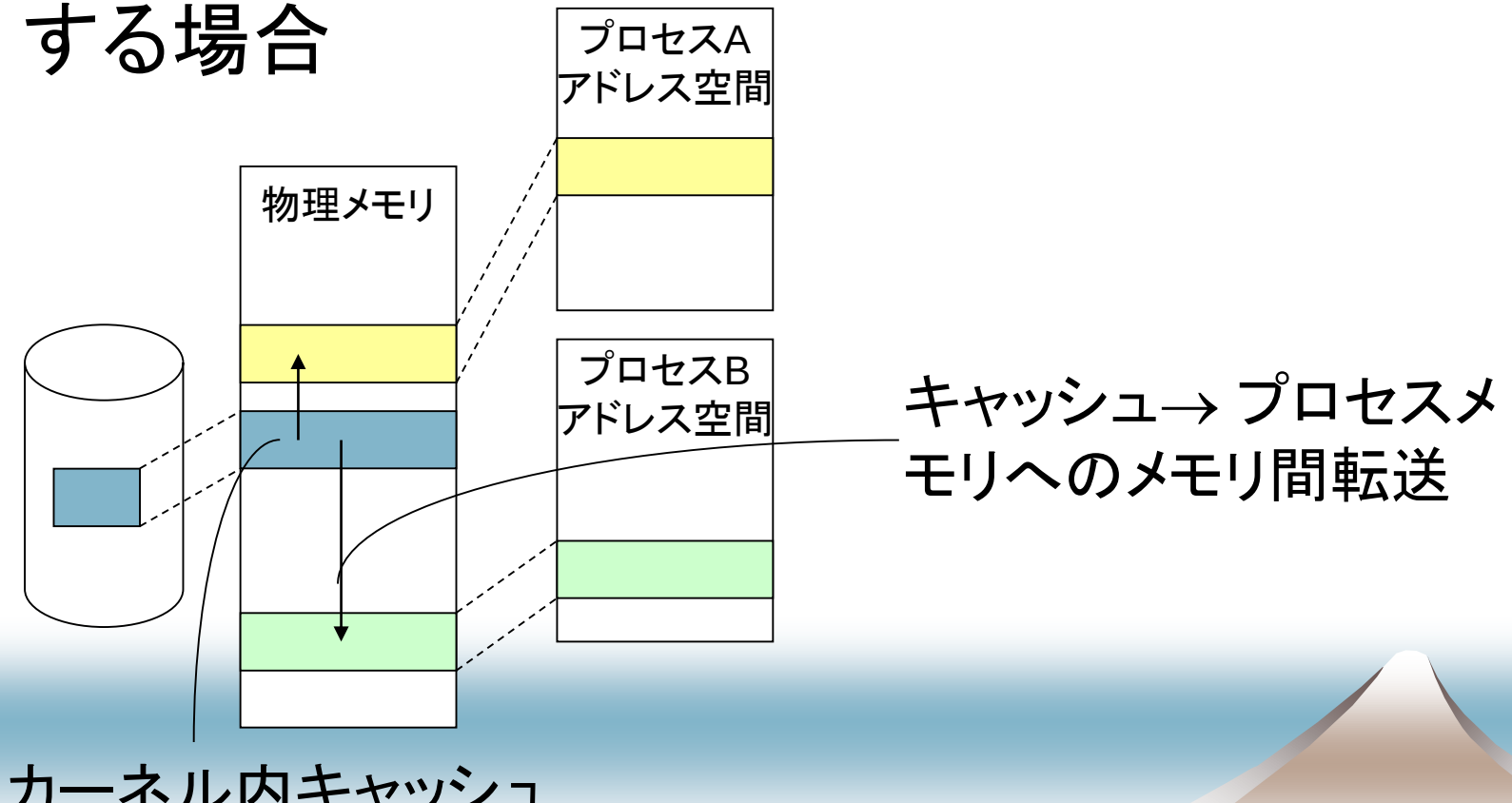
- 子プロセス生成=ページテーブル+アドレス空間記述表のコピー(≠物理メモリのコピー)
  - 生成直後は物理メモリを親子で共有
  - ただし「書き込み不可」に設定しておく
- 書き込まれたページのみ，書き込まれた時点でコピーを生成していく
- 子プロセスがやがてexecveを実行すると，子プロセスのマッピングは除去される

メモリマップドファイルが有効な場面 (2)

- 多数のプロセスが大きなファイルをアクセスする場合
  - 共有マッピング：常に物理ページが共有される
  - プライベートマッピング：書き込まれるまで物理ページが共有される
  - cf. readの場合:
    - 読み込みに使うバッファがプロセス数分
    - ファイルの読み込みに使うキャッシュ

read vs. mmap (1)

- 二つのプロセスA, Bが同じファイルをreadする場合



read vs. mmap (2)

- 同じ状況でmmapが使われた場合