

メモリ管理(仮想記憶)

田浦健次郎

目次

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

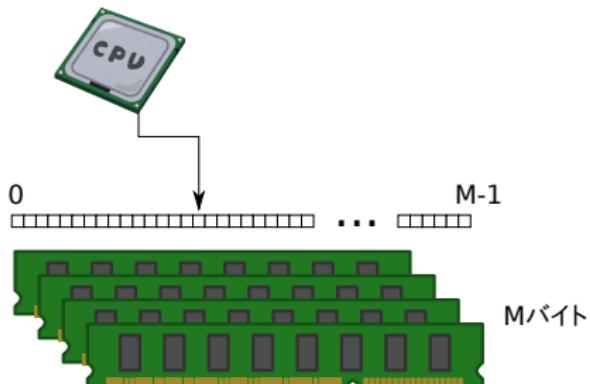
資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

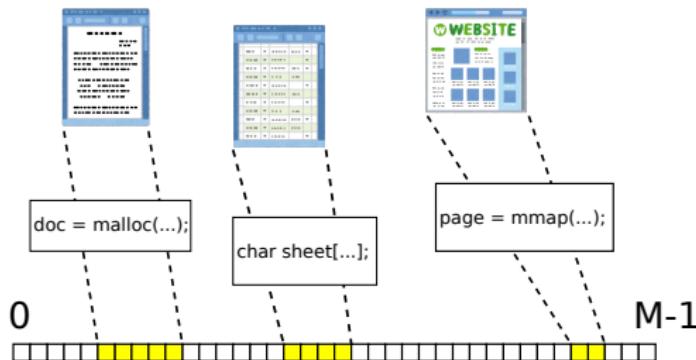
メモリ

- ▶ コンピュータには主記憶（メインメモリ、しばしば単にメモリ）が搭載されている
 - ▶ 典型的な大きさ: 4GB (ノートPC) ~ 256GB (サーバ)
 - ▶ $1\text{GB} = 2^{30}\text{B} = 1073741824\text{B}$
 - ▶ 1B (1バイト) = 8 bit
- ▶ 1Bごとに番地(アドレス)という通し番号がついている
 - ▶ 例: 8GBのメモリ $0 \sim 2^{33} - 1 = 8589934591$
- ▶ CPUはアドレスを指定してメモリを読み書きする (load命令, store命令)



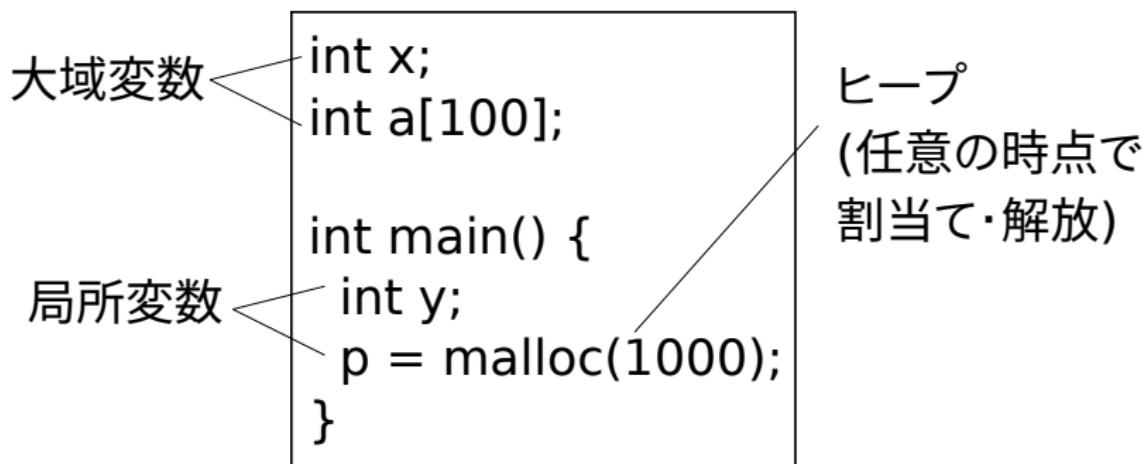
プログラム言語とメモリ

- ▶ プログラムが使うデータはほぼ全てメモリの中にある
 - ▶ 編集中の文書, スプレッドシート, ...
 - ▶ 開いているウェブページ, ...
- ▶ プログラム内では「変数」「配列」などに格納される



プログラム言語とメモリ

- ▶ C/C++でメモリを使う手段
 - ▶ 大域変数・配列
 - ▶ 局所変数・配列
 - ▶ ヒープ



プログラムとアドレス

- ▶ 変数・配列 ≈ アドレス (実際は単なる整数) に名前を付けたもの
- ▶ C 言語のポインタ = アドレス (← ポインタがよくわからないという人へ)
- ▶ ポインタ変数はアドレスを格納する変数
- ▶ C 言語では「変数」「配列」に対応するアドレスをあからさまに (整数として) 見ることが可能

```
1 printf("%ld %ld %ld %ld\n", &x, a, &y, p);
```

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

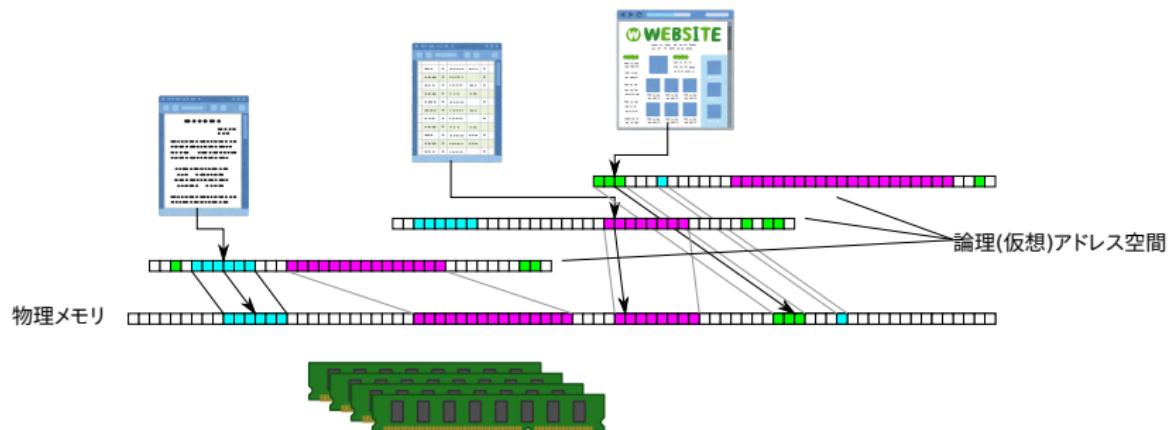
資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

論理アドレスと物理アドレス

- ▶ プロセスごとに論理(仮想)アドレス空間
- ▶ プロセスが扱うアドレスはそのプロセスの論理アドレス空間内のアドレス: 論理(仮想)アドレス
- ▶ 物理メモリを実際にアクセスするときのアドレス: 物理アドレス



論理アドレス空間

1. 1 プロセスに 1 論理アドレス空間
2. アドレスの範囲は物理メモリの量によらない
 - ▶ ソフト (OS) の設計で決まる
 - ▶ 例: Linux x86-64 (Intel の 64 bit CPU) の場合: $0 \sim 2^{48} - 1$
 - ▶ 注: 同じ論理アドレスが全プロセスにある
 - ▶ プロセス P の 10000 番地 \neq プロセス Q の 10000 番地
(対応する物理アドレスが異なる)
3. 論理アドレスから物理アドレスへの変換 (アドレス変換) を CPU が行う
4. 対応する物理アドレスがない場合もある

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

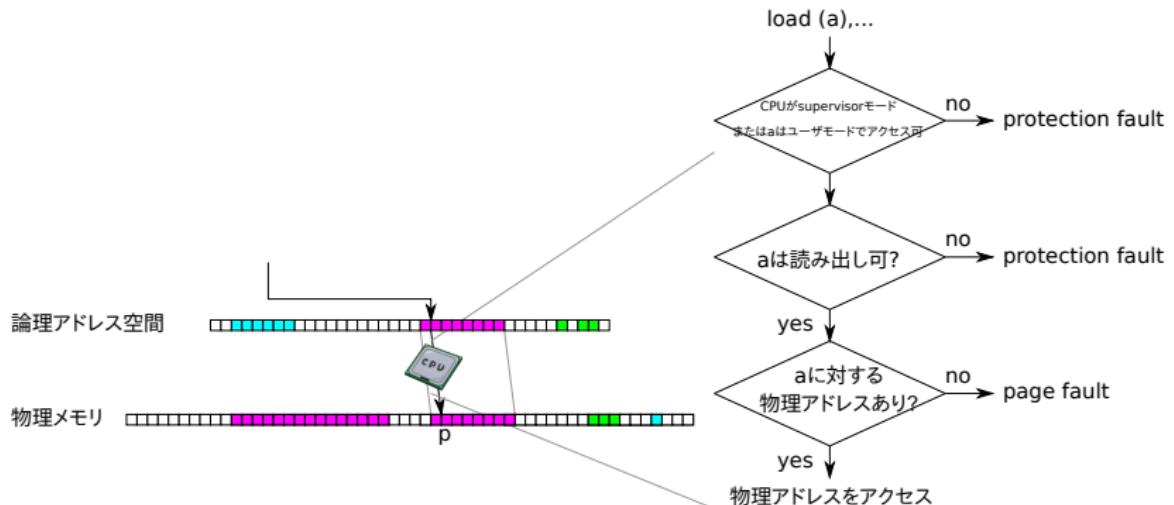
資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

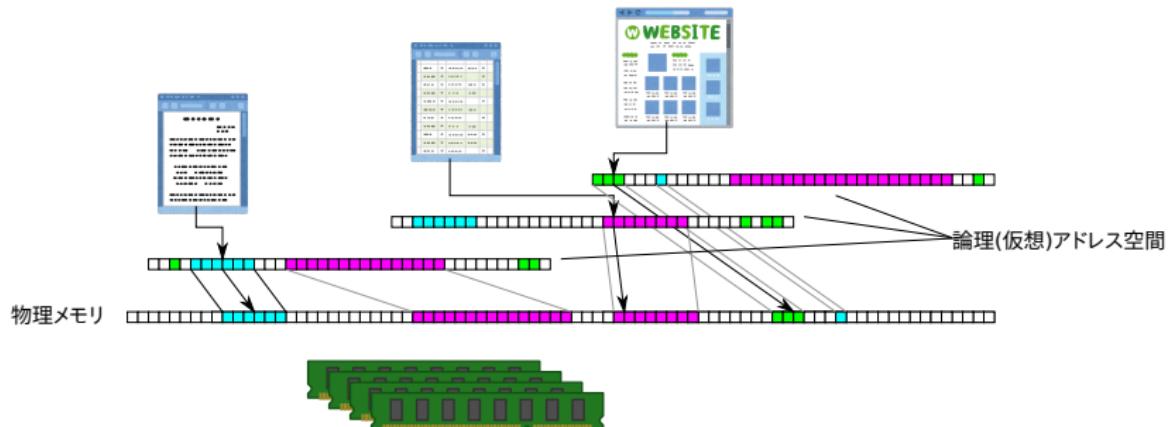
メモリ管理ユニット

- ▶ Memory Management Unit (MMU)
- ▶ CPU 内のハードウェアで全メモリアクセスに介在
- ▶ (論理アドレス空間, 論理アドレス) に対し以下を行う
 1. アクセス権(読み・書き・実行権限)の検査
 2. 対応する物理アドレスが存在するかの検査
 3. 存在していればそのアドレスをアクセス



MMUによって可能になること

1. プロセス間でメモリの分離(保護)
 - ▶ あるプロセスが他のプロセスのデータを見たり壊したり、決してできないようになる
2. カーネル(OS)データの保護
3. 物理メモリ量を越えたメモリ割り当て
4. 要求時ページング(**on demand paging**): 物理メモリを確保せずに、メモリ割り当てを(高速に)行える



メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

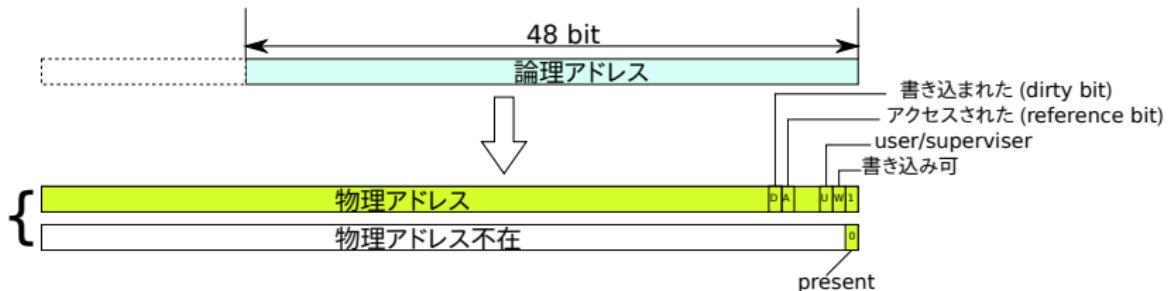
資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

アドレス変換の要件

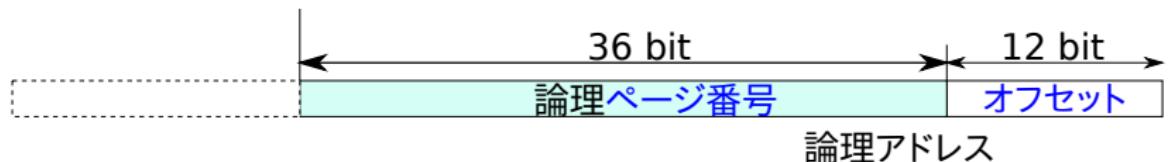
- ▶ 実現すべきもの = 論理アドレス空間ごとに一つの
論理アドレス → (アクセス許可, 物理アドレス) の写像
- ▶ 例: Intel 64 bit の論理アドレス 48 bit



- ▶ 文字通りすべてのアドレス(1バイトごと)に、対応する情報(物理アドレスのサイズ + 数bit ≈ 8バイト)を持たせるのは非現実的
- ▶ $2^{48} \times 8 \times \text{プロセス数} = 2\text{PB} \times \text{プロセス数}$
- ▶ ⇒ ページ

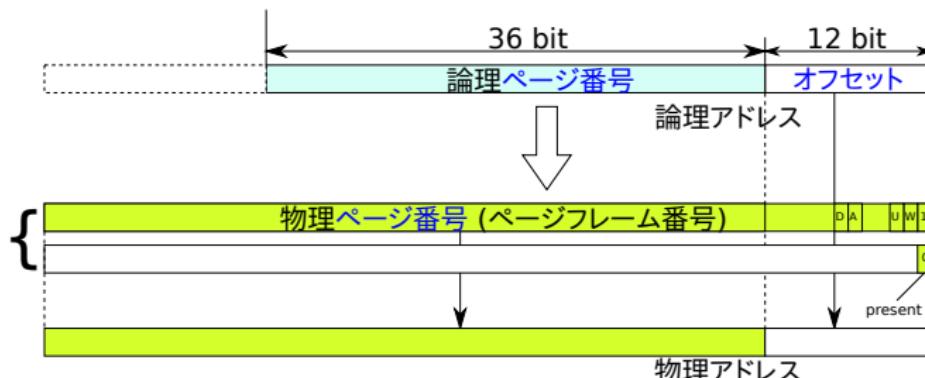
ページ

- ▶ ページ：最下位 A bit 以外が共通の, 2^A 個のアドレス（から成る領域）
 - ▶ その A bit を (ページ内) オフセットと呼ぶ
 - ▶ それ以外の部分：論理ページ番号と呼ぶ
 - ▶ ページは連続する 2^A バイト
- ▶ 典型: $A = 12$ (4KB) または 13 (8KB)



ページ

- ▶ アクセス許可情報, 物理アドレス (のオフセット以外の部分; **物理ページ番号** または **ページフレーム番号**) は, ページごとにひとつだけ持たせる. i.e.,
 - ▶ 1 ページ内の全アドレスのアクセス許可は共通
 - ▶ 1 ページ内の全アドレスは連続したアドレスに変換される (オフセット部分は変換されない)
- ▶ 必要なものは, アドレス空間に一つの, 論理ページ番号 → (アクセス許可, 物理ページ番号) の写像になった



ページテーブル

- ▶ この写像

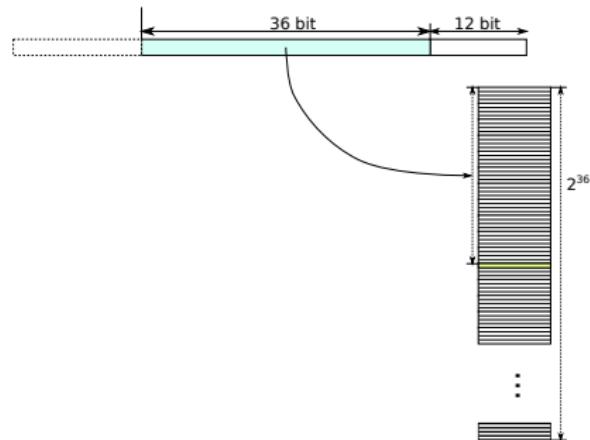
論理ページ番号 → (アクセス許可, 物理ページ番号)

を実現するデータ構造を一般に, ページテーブルと呼ぶ

- ▶ 実際の構造は?

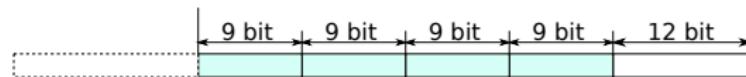
- ▶ 論理ページ番号を index とする
ただの配列 (フラットな配列)
だとすると, $2^{36} = 64G$ 要素; 1
要素 (物理ページ番号 + アクセ
ス許可) 8 バイト ⇒ $64 G \times 8B$
 $= 512GB$ (依然として非現
実的)

⇒ 多段ページテーブル



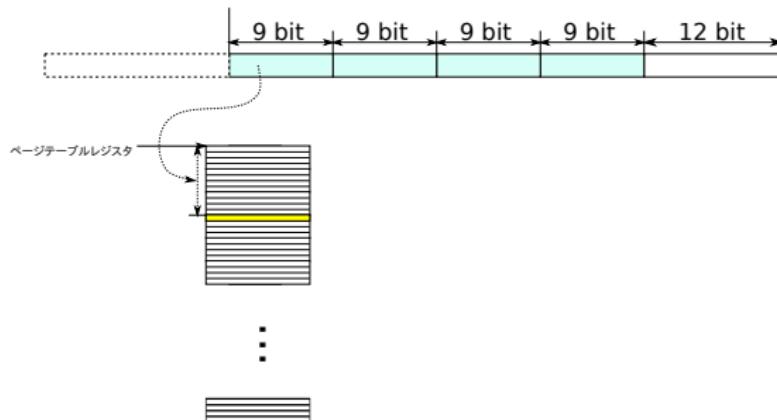
多段ページテーブル

- ▶ 論理ページ番号を複数の、小さい index に分割 (e.g., 36 bit = 9 bit × 4)
- ▶ ページテーブルは深さ ≤ 4 の木構造



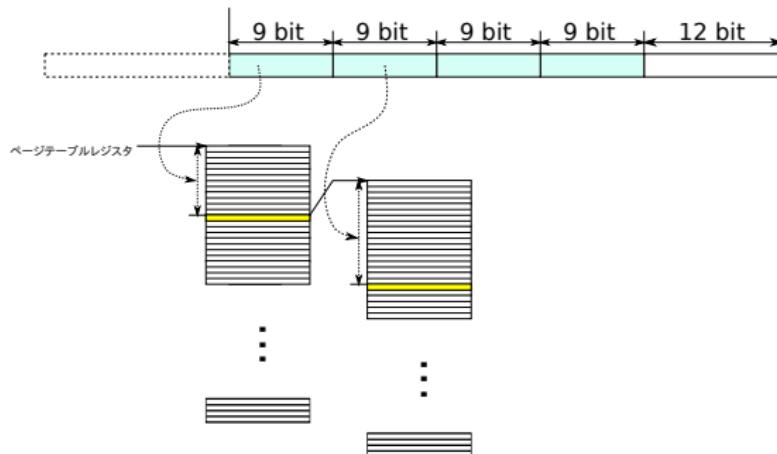
多段ページテーブル

- ▶ 論理ページ番号を複数の、小さい index に分割 (e.g., 36 bit = 9 bit × 4)
- ▶ ページテーブルは深さ ≤ 4 の木構造



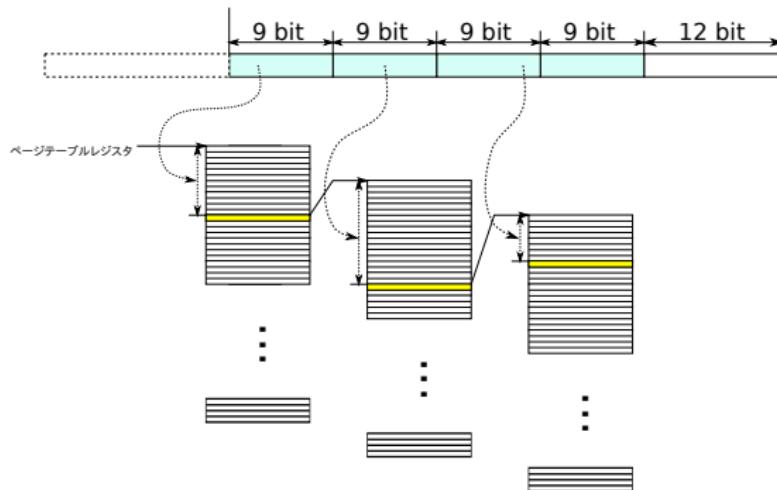
多段ページテーブル

- ▶ 論理ページ番号を複数の、小さい index に分割 (e.g., 36 bit = 9 bit × 4)
- ▶ ページテーブルは深さ ≤ 4 の木構造



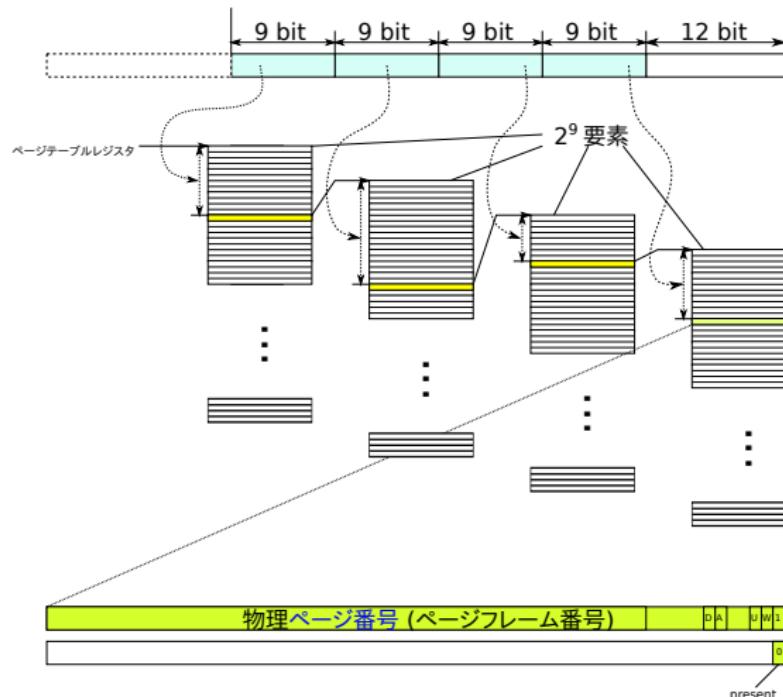
多段ページテーブル

- ▶ 論理ページ番号を複数の、小さい index に分割 (e.g., 36 bit = 9 bit × 4)
- ▶ ページテーブルは深さ ≤ 4 の木構造



多段ページテーブル

- ▶ 論理ページ番号を複数の、小さい index に分割 (e.g., 36 bit = 9 bit × 4)
- ▶ ページテーブルは深さ ≤ 4 の木構造



多段のページテーブルに必要なメモリサイズ

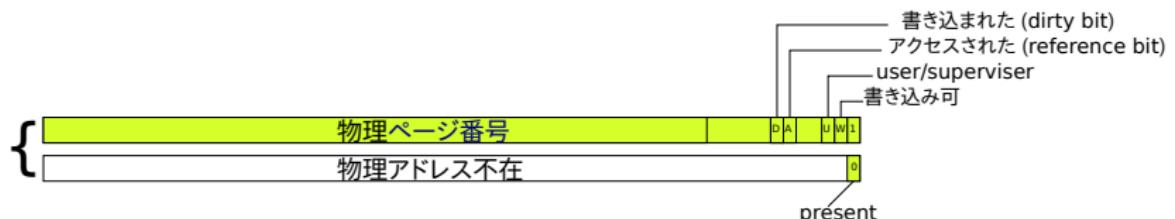
- ▶ 仮定: 36 bit 論理ページ番号 $\rightarrow 9 \text{ bit} \times 4$
- ▶ 一つの表 : 2^9 要素
- ▶ 必要な表の数 : $1 + 2^9 + 2^{18} + 2^{27} \approx 2^{27}$
- ▶ メモリサイズ : $8 \text{ バイト/要素} \times 2^9 \text{要素} \times 2^{27} = 512\text{GB}$
(あれ?)
- ▶ 結局最悪の場合, 1 ページにつき 1 つの物理ページが必要と考えれば当然...

多段のページテーブルに必要なメモリサイズ

- ▶ ほとんどのプロセスは、論理アドレス空間のごく一部しか、使って（割り当てて）いない
- ▶ そのような場合のメモリサイズが問題
- ▶ ただの配列 → 依然として最悪の場合と同じだけ必要
- ▶ 多段ページテーブル → ほとんどのアドレスの、下位の表は不要

ページごとの情報(ページテーブルエントリ)

- ▶ P : 物理ページが割り当てられているか?
- ▶ いる ($P = 1$) 場合:
 - ▶ W : 書き込み可能ならば 1
 - ▶ U : ユーザモードでアクセス可ならば 1
 - ▶ A : アクセスされたときに 1 が set される
 - ▶ D : 書き込まれたときに 1 が set される
 - ▶ PFN : 物理ページ番号 (ページフレーム番号)



Translation Lookaside Buffer (TLB)

- ▶ 1つの論理アドレスをアクセスする準備—アドレス変換のために、4回のメモリアクセスが必要(避けたい)
⇒ **Translation Lookaside Buffer (TLB)**
- ▶ (論理アドレス空間, 論理ページ番号) → (アクセス許可, 物理ページ番号) の写像の一部を保持するプロセッサ内のキャッシュ
- ▶ ≈ 1024 要素程度

その他 細かめの話

▶ Large pages, huge pages

- ▶ ページ内のオフセットを何 bit とするか (= どこまでを論理ページ番号 (アドレス変換のキー) とするか) で 1 ページのサイズが変わる
- ▶ 多段のページテーブルでは、何段目までを論理ページ番号に使うかで、異なるページサイズを同居可能
 - ▶ $36 + 12 \Rightarrow$ (通常の) 4KB ページ
 - ▶ $27 + 21 \Rightarrow$ 2MB ページ
 - ▶ $18 + 30 \Rightarrow$ 1GB ページ
- ▶ 57 bit 論理アドレス (5 段のページテーブル) Intel の最新 (Ice Lake) のアーキテクチャで導入
 - ▶ 48 bit \Rightarrow 256 TB 論理アドレス空間
 - ▶ 57 bit \Rightarrow 128 PB 論理アドレス空間

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

Unix メモリ割り当て API (1)

- ▶ `int e = brk(l);`
 - ▶ データ領域の終わりのアドレスを l にする
 - ▶ $\Rightarrow x < l$ のアドレス x が「利用可能」になる (つまりメモリが割り当てられる)
- ▶ `void * a = sbrk(sz);`
 - ▶ データ領域の終わりのアドレスを sz バイト増やし, 増やす前のアドレスを返す
 - ▶ $\Rightarrow x \in [a, a + sz)$ のアドレスが利用可能になる (つまりメモリが割り当てられる)
 - ▶ `sbrk(0)` は「データセグメントの終わりのアドレス」を返す

余談: なぜ変な名前(brk, sbrk)?

- ▶ 古の Unix でのアドレス空間:
割当て領域を少数の連続領域
(セグメント) に限定
 - ▶ コード(テキスト)セグメント
 - ▶ スタックセグメント
 - ▶ データセグメント
- ▶ データセグメントの終わりのア
ドレスを **break** 値と呼んで
いた
- ▶ 現代の OS: MMU によりページ
単位で任意に割当て・非割当領
域を管理可能



Unix メモリ割り当て API (2)

- ▶ `mmap`, `mremap`, `munmap` : より柔軟なメモリ割り当て, 開放, ファイルのマッピング(後述)
- ▶ `int e = mprotect(a, sz, prot);`
 - ▶ メモリ領域のアクセス許可を設定(書き込み, 読み込み不可に)できる

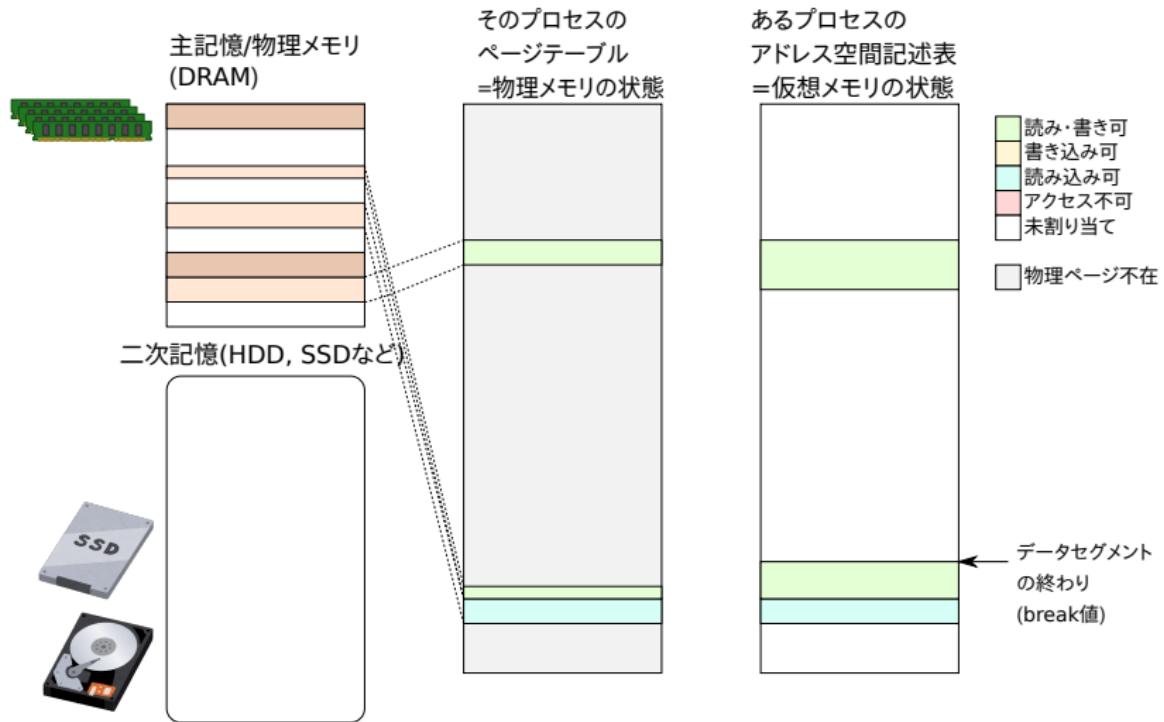
メモリ割り当て = 「仮想」メモリの割当 ≠ 「物理」メモリの割当

- ▶ brk, sbrk, mmap などでメモリを割り当てるものの効果
 - ▶ = 所定の論理アドレスの範囲が「アクセス可能になる」
 - ▶ = 「アクセス時に Segmentation Fault が起きなくなる」
- ▶ 実際の動作:
 1. OS が管理するデータ ([アドレス空間記述表](#), Linux `mm_struct`) に、割当て済みであることを記述する (だけ)
 2. 物理メモリはすぐには割り当てられない

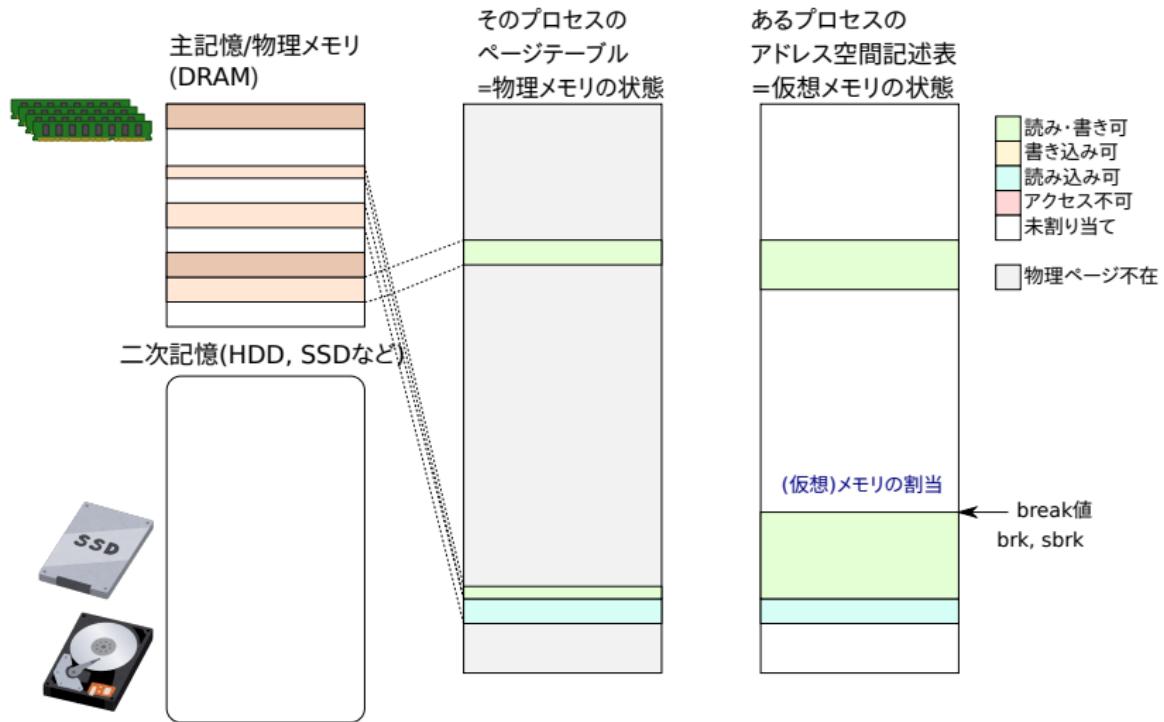
メモリ割り当て = 「仮想」メモリの割当 ≠ 「物理」メモリの割当

- ▶ 実際にアクセスが起きた際に
 1. CPU がページ fault 例外を発生
 2. OS の例外処理 (ページ fault 処理) が発動
 - ▶ Unix: マイナーフault と呼び後述のメジャーフault と区別
 3. OS がアドレス空間記述表を見て、そのアクセスが実際には合法であることが確認されると、物理メモリが割り当てられる (**要求時ページング**)

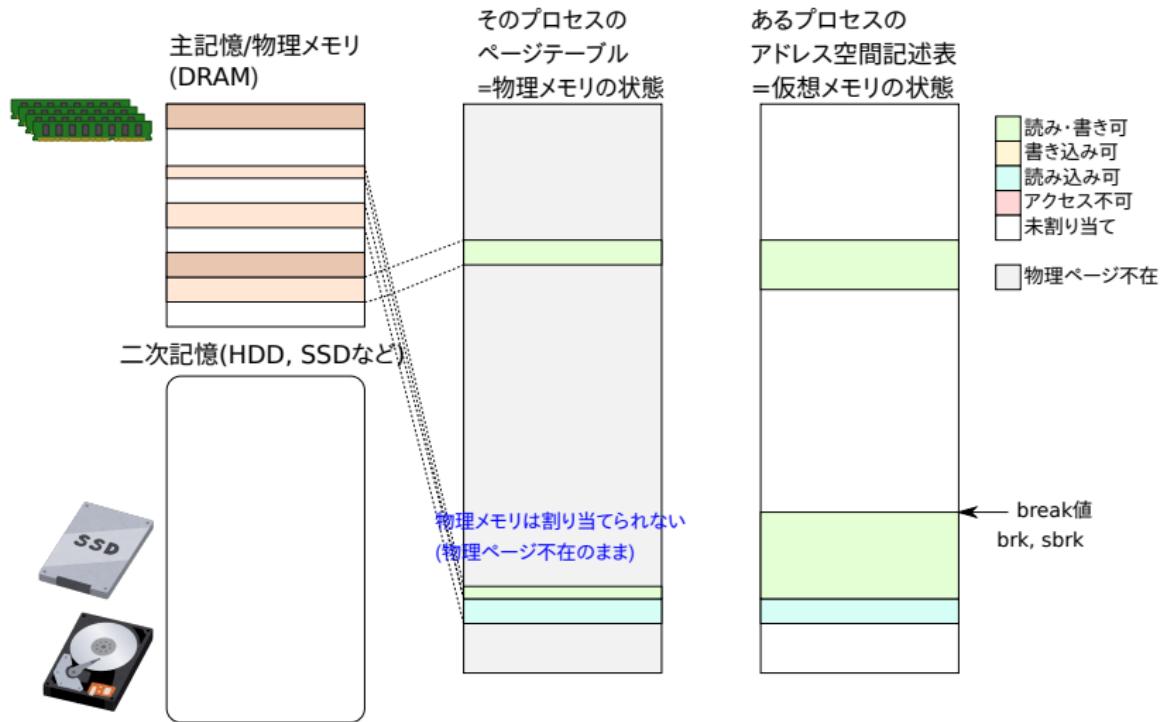
メモリ管理動作図(割当て～要求時ページング)



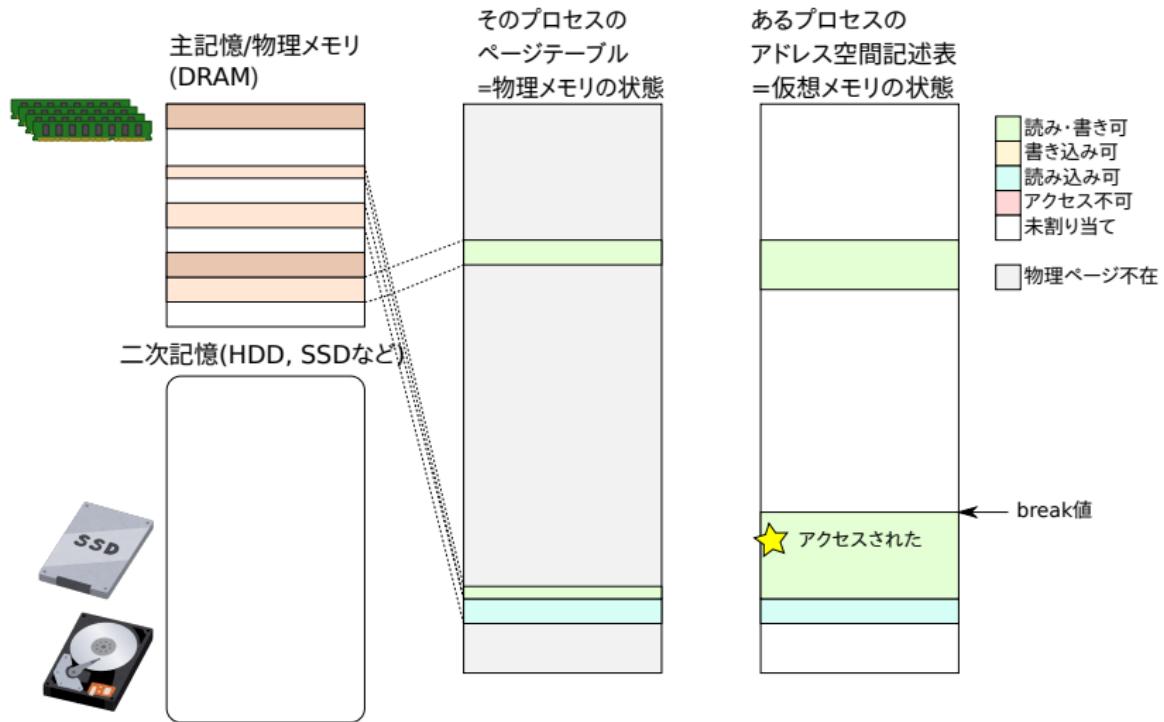
メモリ管理動作図(割当て～要求時ページング)



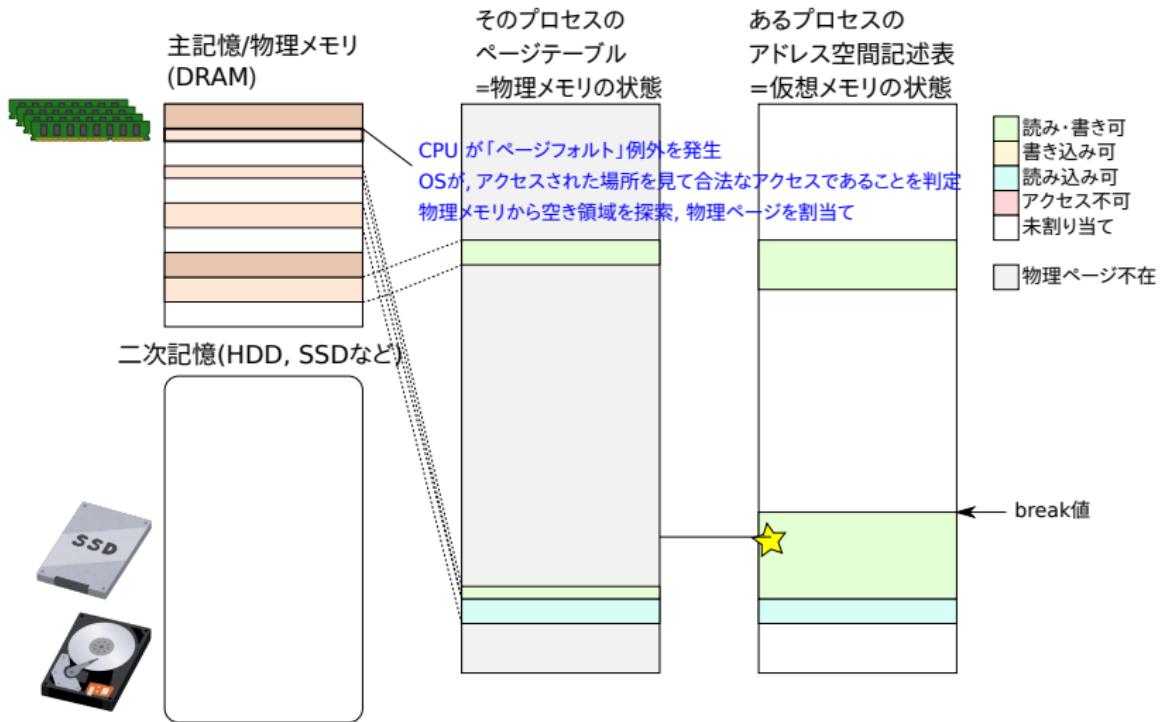
メモリ管理動作図(割当て～要求時ページング)



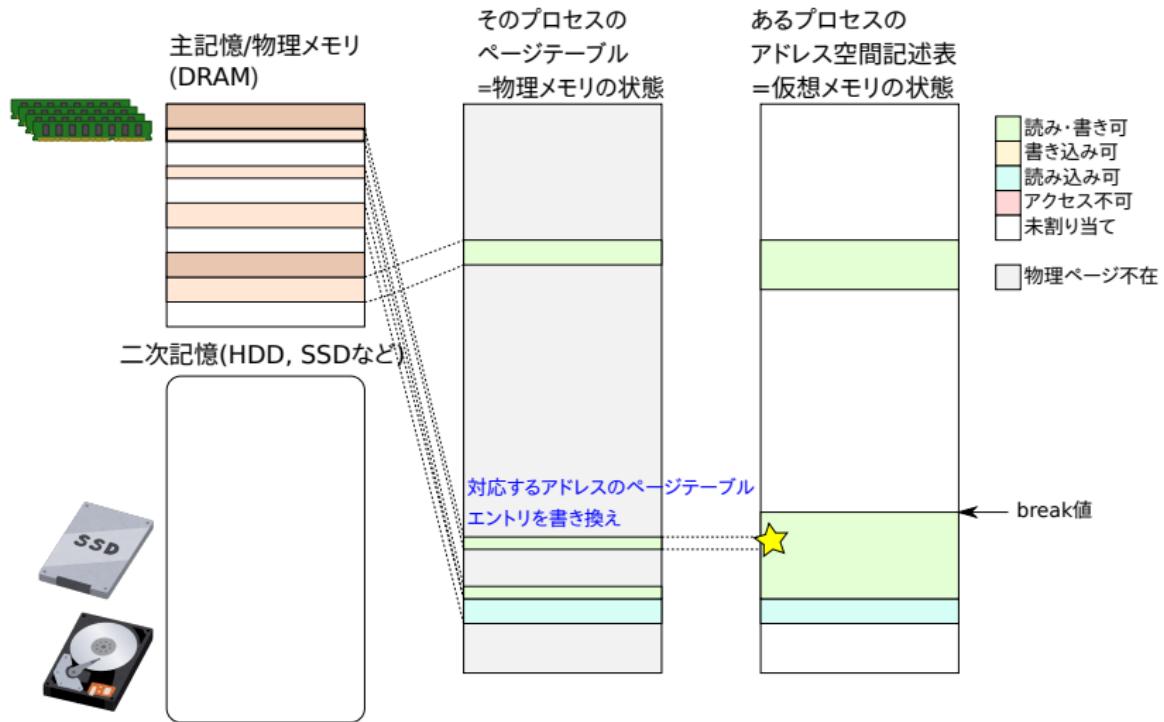
メモリ管理動作図(割当て～要求時ページング)



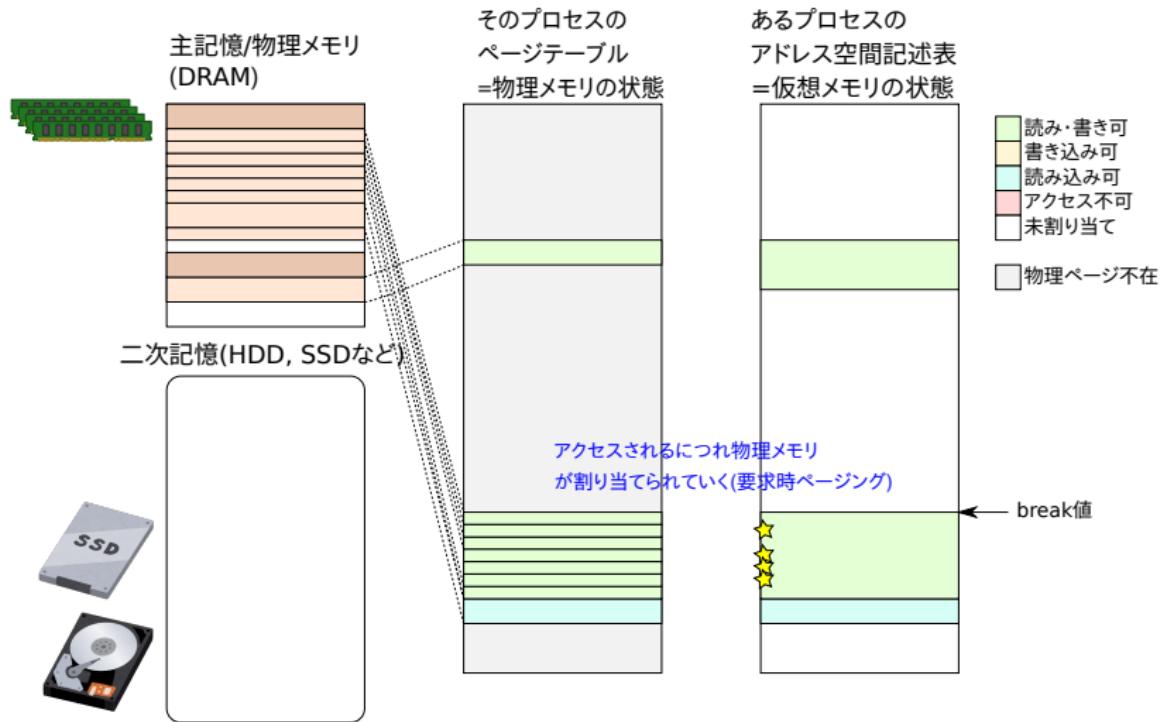
メモリ管理動作図(割当て～要求時ページング)



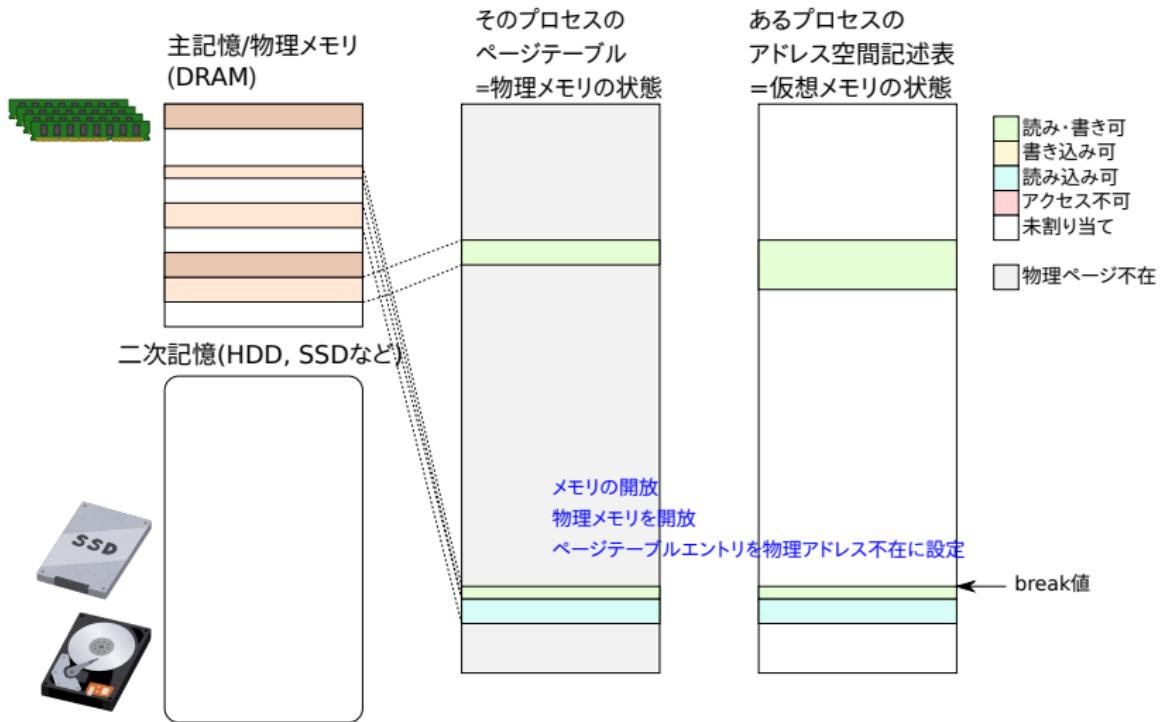
メモリ管理動作図(割当て～要求時ページング)



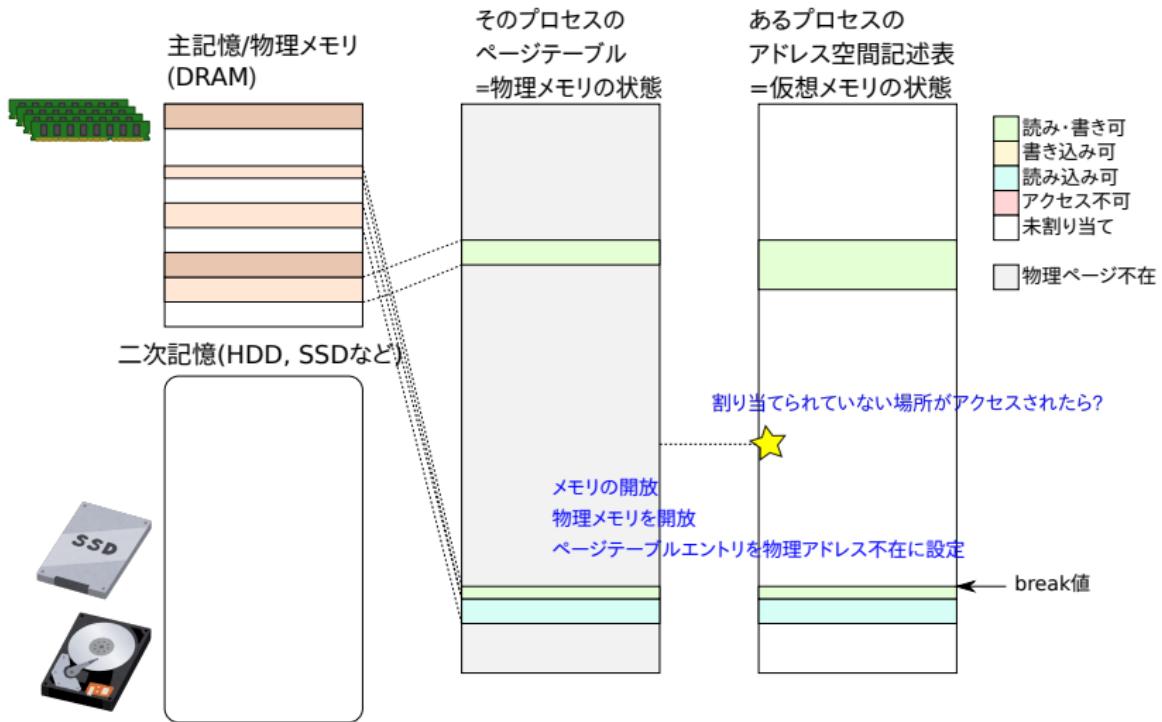
メモリ管理動作図(割当て～要求時ページング)



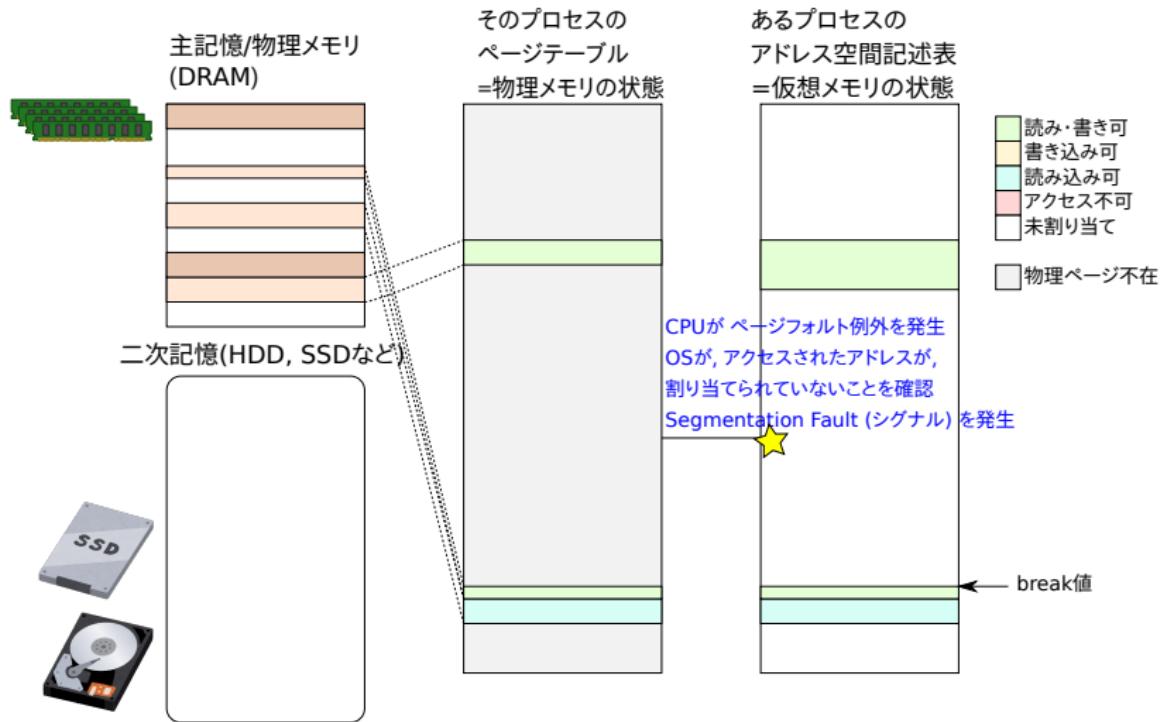
メモリ管理動作図(割当て～要求時ページング)



メモリ管理動作図(割当て～要求時ページング)



メモリ管理動作図(割当て～要求時ページング)



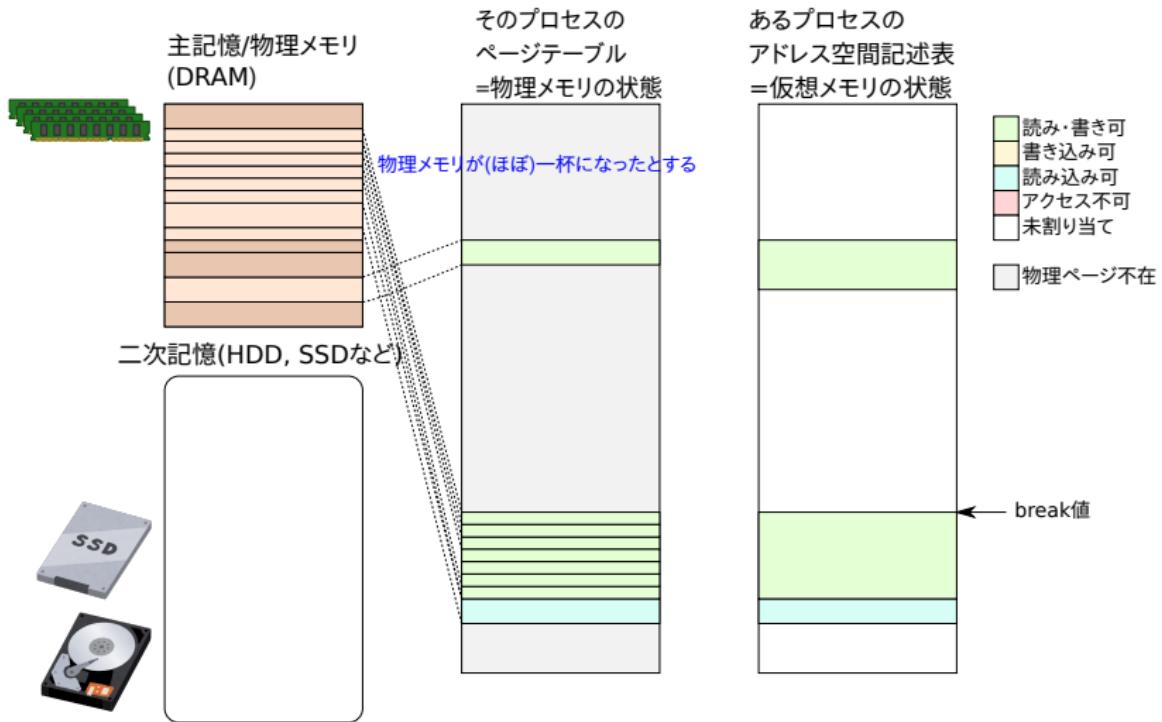
ページ(スワップ)アウト・イン

- ▶ 実際にアクセスが起きたページに物理メモリを割当 (要求時ページング)
- ▶ ⇒ 当然, 物理メモリが足りなく (団囲する) ことがあり得る
- ▶ OSは物理メモリが団囲すると, 一部の内容を2次記憶に退避して, 解放する (**ページアウト, スワップアウト**)
- ▶ 退避されたページの, ページテーブルエントリは「物理ページ不在」とし, 次にアクセスする際にページ fault が起きるようにしておく
- ▶ ページアウトされた領域が再びアクセスされたら, ページ fault 处理で, 2次記憶に退避された内容を読み込む (Unix: **メジャーフault**)

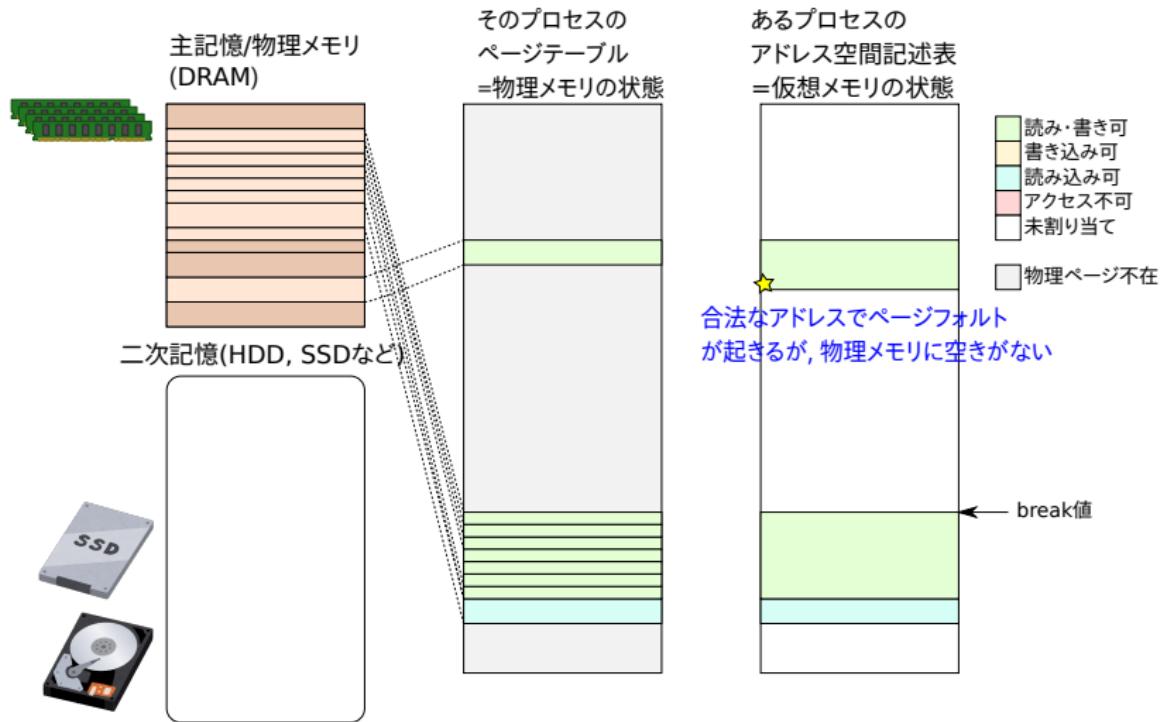
マイナーフォルト vs. メジャーフォルト

- ▶ マイナーフォルト: 割当てられたページへの初めてのアクセス時におこるページフォルト
 - ▶ 要求時ページングをしている限り必然的に(1ページにつき1度は)起こる
 - ▶ 実際の処理は物理メモリの探索 + ゼロ初期化
- ▶ メジャーフォルト: ページアウトされたページに対するアクセス時におこるページフォルト
 - ▶ ページの内容を2次記憶から読み込む処理を伴う
 - ▶ マイナーフォルトより時間がかかる

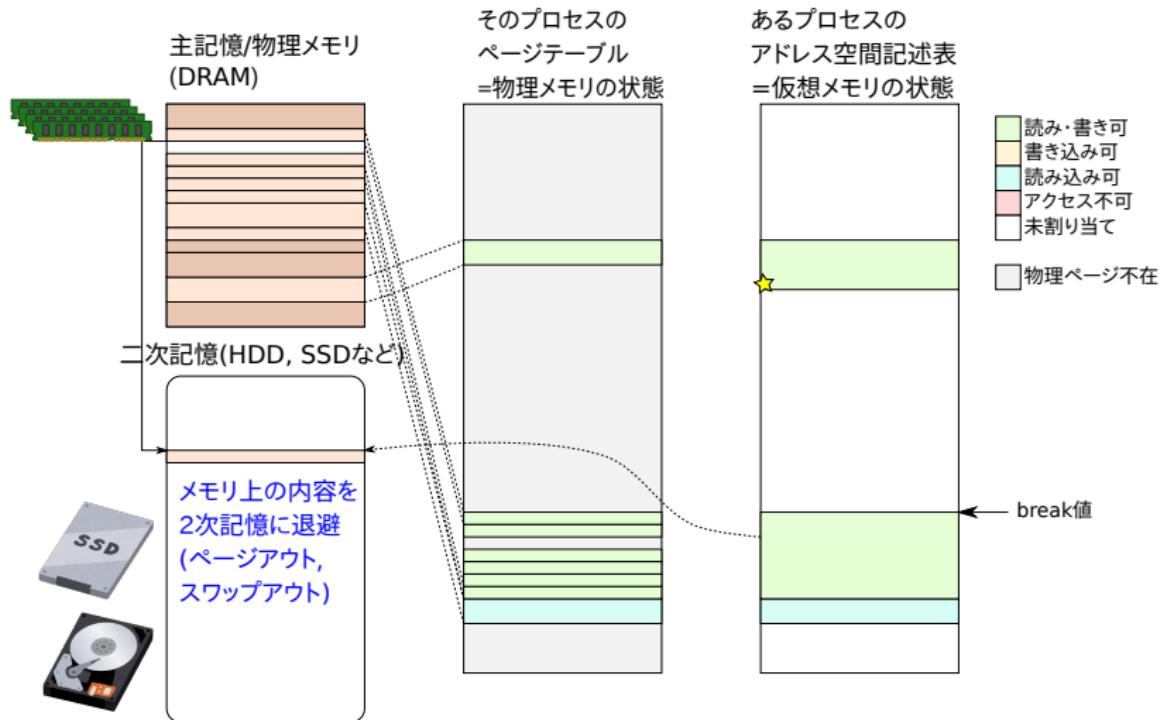
メモリ管理動作図(ページアウト・ページイン)



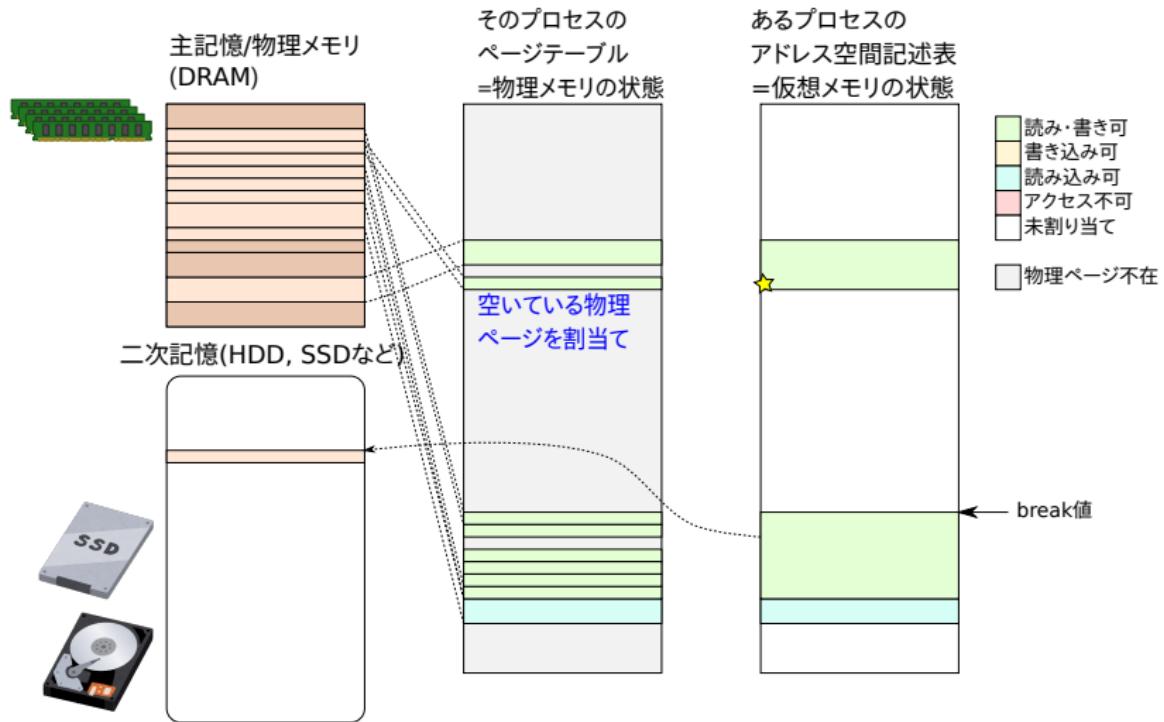
メモリ管理動作図(ページアウト・ページイン)



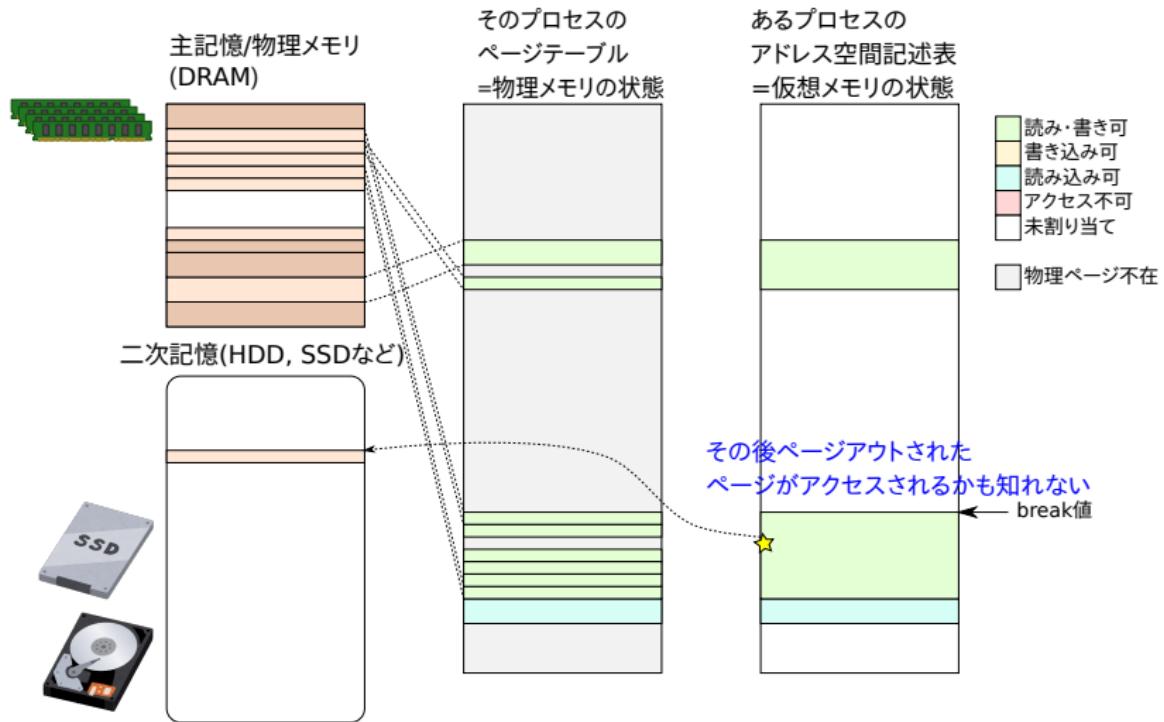
メモリ管理動作図(ページアウト・ページイン)



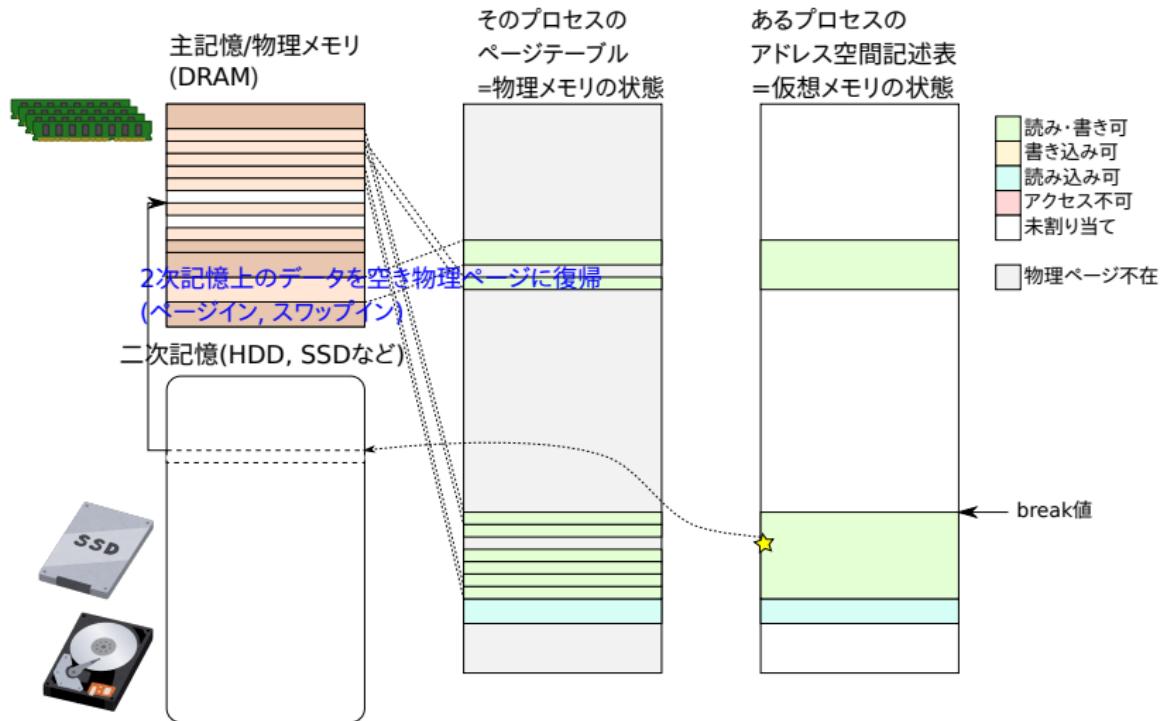
メモリ管理動作図(ページアウト・ページイン)



メモリ管理動作図(ページアウト・ページイン)



メモリ管理動作図(ページアウト・ページイン)



メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OSのメモリ割り当て API

資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

資源使用量を知る API

- ▶ `int e = getrusage(RUSAGE_SELF, ru);`
 - ▶呼び出したプロセスの種々の資源使用量, イベント数を `ru` に返す
 - ▶マイナーフォルト, メジャーフォルト, 最大物理メモリ 使用量 (RSS), CPU 仕様時間, etc.
- ▶ `int e = getrusage(RUSAGE_CHILDREN, ru);`
 - ▶上記同様. ただし子プロセスに対して

資源使用量を制限する API

- ▶ `int e = setrlimit(RLIMIT_res, rl);`
 - ▶呼び出したプロセスの資源 `res` の使用量を制限
 - ▶仮想メモリ (`AS`), CPU 使用時間 (`CPU`), etc.
 - ▶物理メモリ (`RSS`) は効き目がない模様 (man page: “*This limit has effect only in Linux 2.4.x, x < 30, and there affects only calls to madvise(2) specifying MADV_WILLNEED*”)
- ▶ `int e = prlimit(pid, RLIMIT_res, rl);`
 - ▶ `getrlimit`, `setrlimit` の機能をひとつに統合
 - ▶(権限があれば) 制限するプロセスを指定可能
- ▶ cgroups (後述)

物理メモリの割当状況を知る API

- ▶ `int e = mincore(addr, len, vec);`
 - ▶ $[addr, addr + len)$ 内の各論理ページが物理メモリ上にある・ないを `vec` から始まる (char の) 配列に格納する
 - ▶ i ページ目が物理メモリにある $\iff \text{vec}[i] == 1$
- ▶ 本来アプリケーションが知る必要がない情報ではあるが、OS の挙動を知るために有用

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

cgroups

- ▶ Linux で一部のプロセス (とその子孫) に対する資源割当量を制御する仕組み
 - ▶ コンテナ (Docker など) の基盤
- ▶ 詳細 → `man cgroups`
- ▶ version v1 と v2 がある
- ▶ 演習環境では cgroups v2 を設定している
- ▶ 以下は自分のマシンで cgroups v2 を使ってみたい人のための参考情報

cgroups v2 を手動で設定

1. 適当な空ディレクトリを作成

```
1 $ mkdir ~/tmp/cg2
```

2. cgroup2 ファイルシステムをマウント

```
1 $ sudo mount -t cgroup2 none ~/tmp/cg2
```

- ▶ 起動時に, /sys/fs/cgroup 下にすでに設定がなされている可能性あり

```
1 $ mount | grep cgroup
2 cgroup2 on /sys/fs/cgroup type cgroup2 (rw,nosuid,nodev,noexec,
    relatime)
```

以下が起動時になされている

```
1 sudo mount -t cgroup2 /sys/fs/cgroup
```

起動時に cgroups v2 を有効にする

1. /etc/default/grub で GRUB_CMDLINE_LINUX_DEFAULT を設定

```
1 GRUB_CMDLINE_LINUX_DEFAULT="... systemd.unified_cgroup_hierarchy=1"
```

2. 上記設定をカーネル起動パラメータに反映

```
1 $ sudo update-grub
```

3. 再起動

- ▶ 以下のファイルが見えれば (*) 有効

```
1 $ ls /sys/fs/cgroup  
2 ... cgroup.procs ... cgroup.subtree_control ...
```

- ▶ 以下はこうなっている前提で説明

cgroups v2 の基本操作

1. トップディレクトリ (`/sys/fs/cgroup`) 下に好きなディレクトリ階層構造を作る

```
1 $ sudo mkdir /sys/fs/cgroup/R  
2 $ sudo mkdir /sys/fs/cgroup/R/A  
3 $ sudo mkdir /sys/fs/cgroup/R/B  
4 $ ...
```

▶ ディレクトリ ⇄ プロセスのグループ

2. プロセスのグループに対し資源量 (物理メモリ量など) の制限をかける

```
1 # R 下のグループの memory を制御  
2 $ echo +memory | sudo tee /sys/fs/cgroup/R/cgroup.subtree_control  
3 # A に属するプロセスグループの合計物理メモリ使用量を 256MB にする  
4 $ echo 256M | sudo tee /sys/fs/cgroup/R/A/memory.high
```

3. プロセスをグループに所属させる

```
1 $ echo pid | sudo tee /sys/fs/cgroup/R/A/cgroup.procs
```

演習用 Jupyter 環境 cg_mem_limit の正体

▶ コマンドの仕様

```
1 cg_mem_limit コマンド
```

でコマンドの物理メモリ使用量を 256MB に制限 (それを越えたらページアウト)

▶ 仕組み

1. fork
2. 子プロセスの ID を

/sys/fs/cgroup/taulec/user/cgroup.procs に書く

3. 子プロセスはそれを持ってから exec

▶ 注: 事前設定

```
1 $ echo +memory | sudo tee /sys/fs/cgroup/taulec/cgroup.  
subtree_control
```

- ## ▶ 全 user に対し

```
1 $ mkdir /sys/fs/cgroup/taulec/user  
2 $ echo 256M | sudo tee /sys/fs/cgroup/taulec/user/memory.high
```

スワップ領域

- ▶ memory.high を設定する(物理メモリ量を制限する)場合、**スワップ領域**の有効化が必要
- ▶ すでに設定されているか? → free, swapon コマンド

```
1 $ free
2      total        used         free        shared   buff/cache available
3 Mem:   32513016  5507528  20926012  124652   6079476   26408444
4 Swap:  2097148      6144  2091004
5 $ swapon
6 NAME      TYPE SIZE USED PRIO
7 /swapfile file  2G    6M    -2
```

スワップ領域の有効化

- 大きさに合わせたファイルをつくる (2048MB の例)

```
1 $ sudo dd if=/dev/zero of=/swapfile bs=$((1024 * 1024)) count=2048
```

- スワップ領域用のファイルとして初期化

```
1 $ sudo chmod 600 /swapfile  
2 $ sudo mkswap /swapfile
```

- 有効化する

```
1 $ sudo swapon /swapfile
```

設定を永続化したければ以下を/etc/fstab に記述

```
1 /swapfile swap swap defaults 0 0
```

メモリの復習

論理アドレス空間

メモリ管理ユニット

アドレス変換の実際

OS のメモリ割り当て API

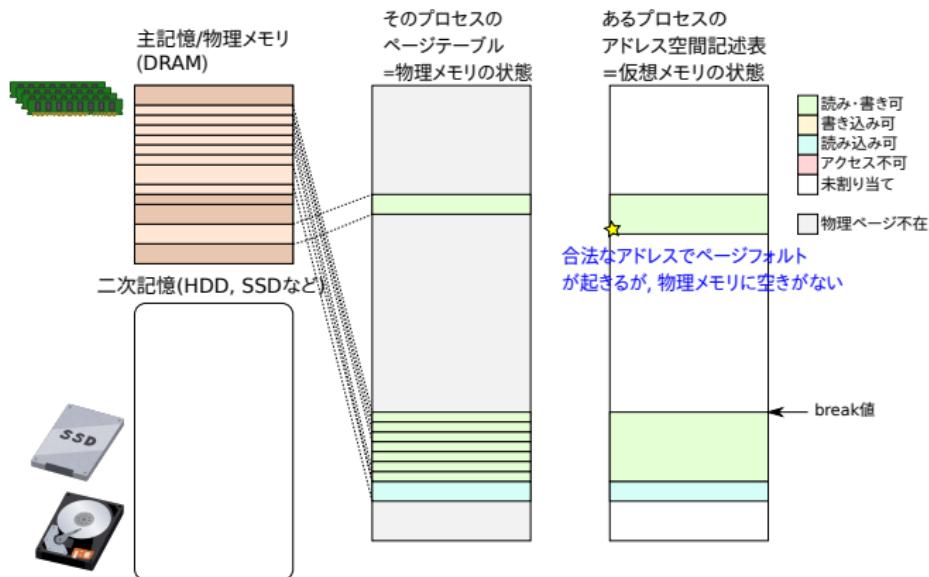
資源使用量を知る・制限する API

cgroups

ページ置換アルゴリズム

ページ置換アルゴリズム

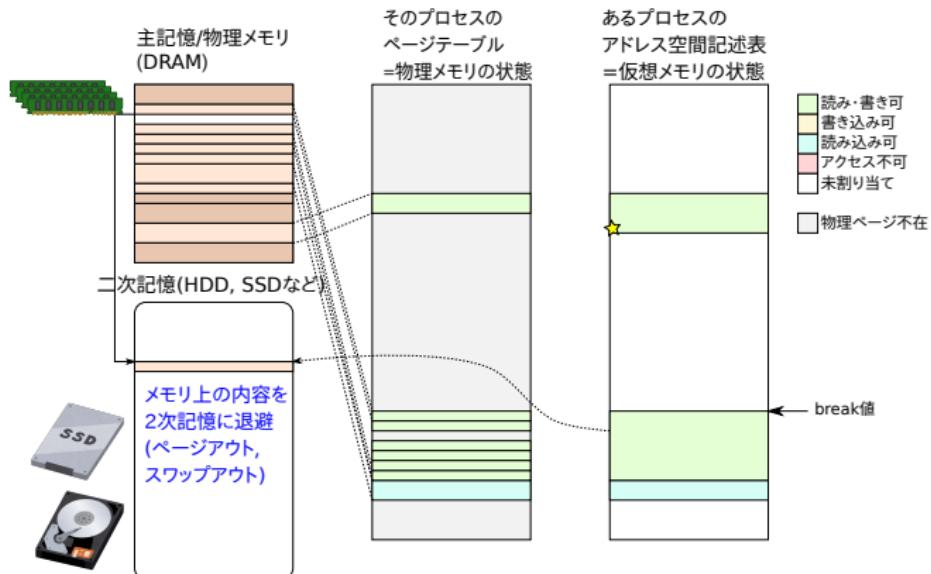
- ▶ **問題:** 物理メモリ上のページを 2 次記憶に退避 (ページアウト) する際, どのページを退避させるか?



- ▶ **目標:** ページの置換 (退避と復帰) のコストの最小化
- ▶ → ページ置換アルゴリズム

ページ置換アルゴリズム

- ▶ **問題:** 物理メモリ上のページを 2 次記憶に退避 (ページアウト) する際, どのページを退避させるか?

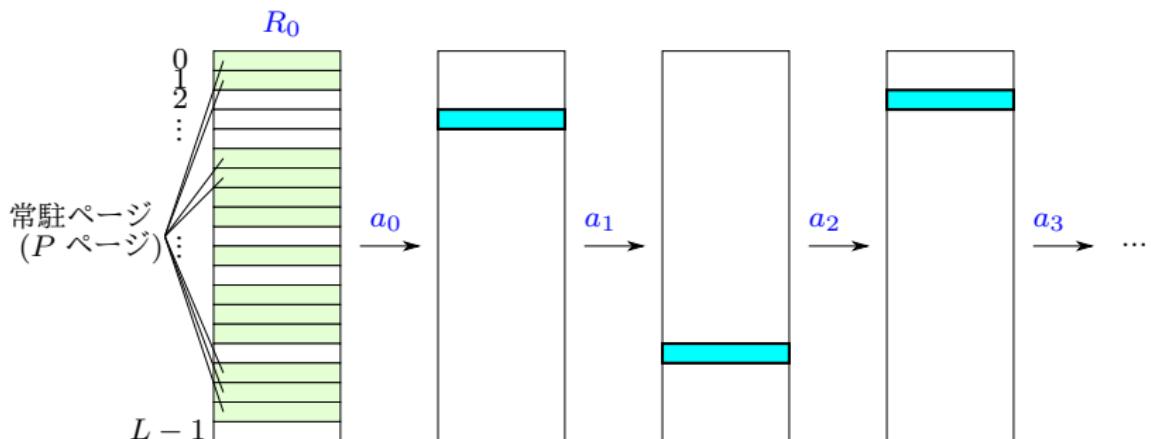


- ▶ **目標:** ページの置換 (退避と復帰) のコストの最小化
- ▶ → ページ置換アルゴリズム

ページ置換の問題のモデル化 (入力)

▶ 入力

- ▶ L : アクセスされるページ数 (ページ番号 $0, \dots, L - 1$)
- ▶ P : 物理メモリサイズ (ページ数)
- ▶ R_0 : 初期常駐ページ集合 ($|R_0| = P$)
- ▶ $a = a_0, a_1, a_2, \dots, a_{n-1}$: アクセスされるページの系列



ページ置換の問題のモデル化 (出力)

▶ 出力

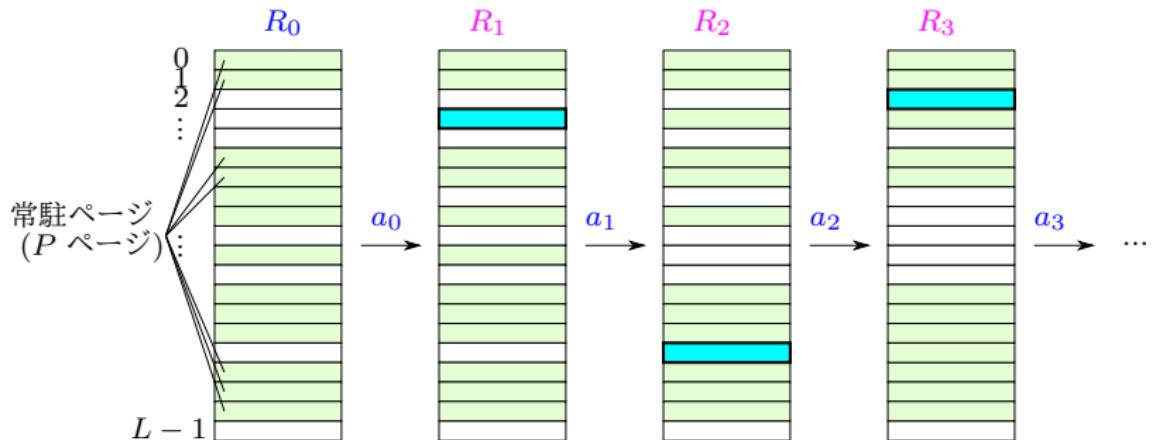
- ▶ 各アクセス後の常駐ページ集合の系列

$$\mathbf{R} = R_1, R_2, \dots, R_n$$

- ▶ ただし以下を条件とする

- ▶ $|R_i| = P$

- ▶ $a_i \in R_{i+1}$ (解釈: a_i アクセス直後は a_i が物理メモリ上にある)

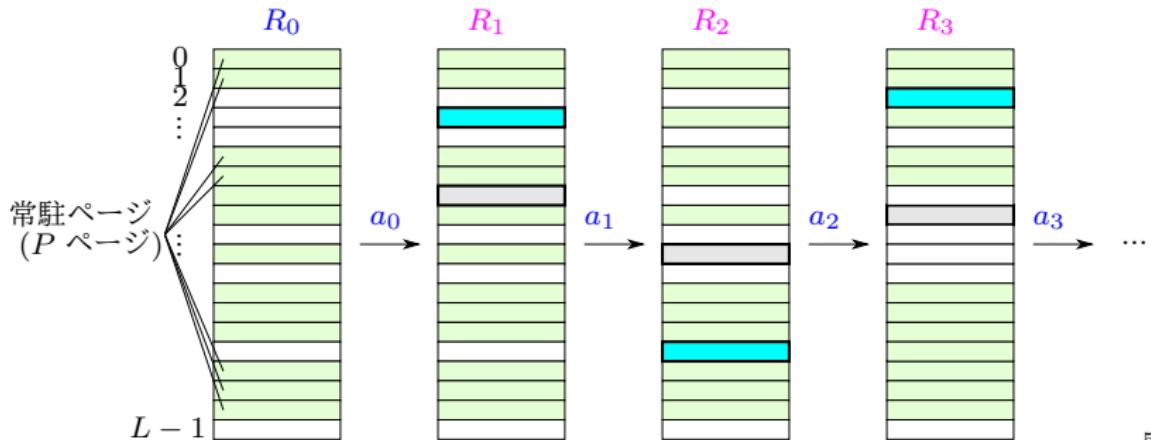


ページ置換の問題のモデル化 (評価尺度)

- ▶ 評価尺度 = ページ置換数

$$\text{REP}(\mathbf{R}) \equiv \sum_{i=0,1,\dots,n-1} |R_i \setminus R_{i+1}|$$

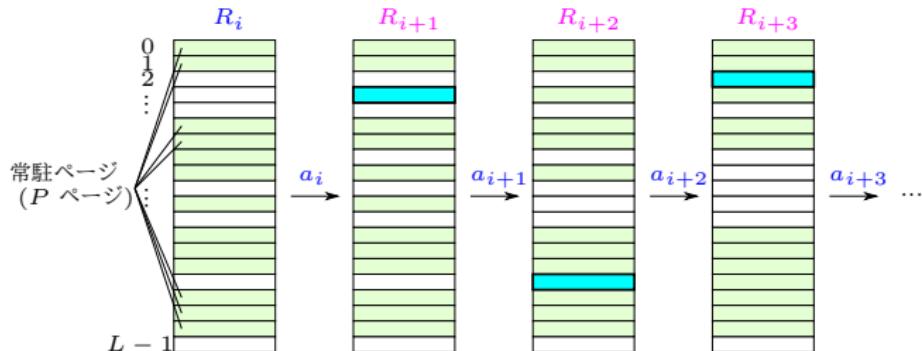
- ▶ $(R_i \setminus R_{i+1})$ は集合の差分 (ページアウト数)
- ▶ $|R_i| = P$ の条件下ではページアウト数 = ページイン数
- ▶ (アクセスされていないページをインすることになると)
すると) = ページフォルト数



2つの問題設定

- ▶ オフライン問題: アルゴリズム (A) は、将来のアクセス系列を全て入力とする (知っている)

$$A_{\text{OFF}}(R_i, [a_0, a_1, \dots, a_{i-1}], [a_i, \dots, a_{n-1}]) = R_{i+1}$$



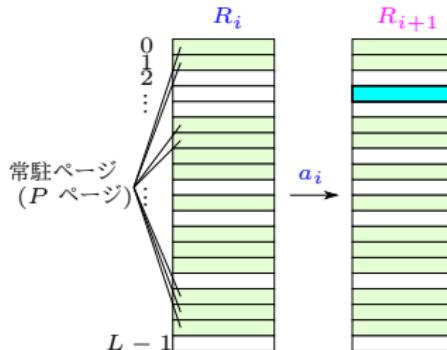
2つの問題設定

- ▶ オフライン問題: アルゴリズム (A) は, 将来のアクセス系列を全て入力とする (知っている)

$$A_{\text{OFF}}(R_i, [a_0, a_1, \dots, a_{i-1}], [a_i, \dots, a_{n-1}]) = R_{i+1}$$

- ▶ オンライン問題: アルゴリズム (A) は, 将來のアクセスはわからない

$$A_{\text{ON}}(R_i, [a_0, a_1, \dots, a_{i-1}], [a_i]) = R_{i+1}$$



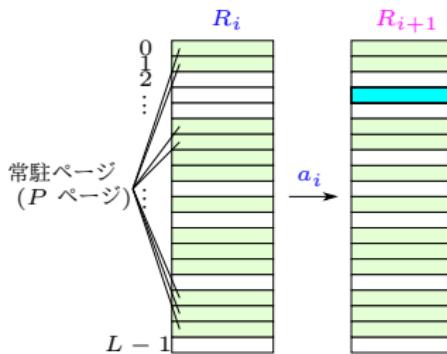
オフライン vs オンライン

▶ オフライン問題

- ▶ 将来のアクセスがわかっているという仮定は非現実的
- ▶ 最適なアルゴリズムが存在し、現実にあわない仮定であっても設計の指針にはなる

▶ オンライン問題

- ▶ 問題設定は現実的
- ▶ どんなアルゴリズムでも最悪ケースは同じ

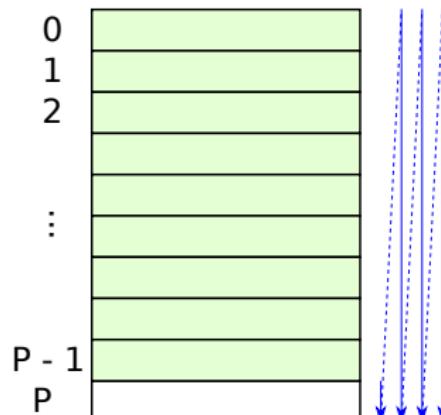


重要性を確認する問題

- ▶ 初期常駐ページ $R_0 = \{0, 1, \dots, P - 1\}$
- ▶ アクセス系列

$$\mathbf{a} = P, 0, 1, \dots, P, 0, 1, \dots, P, 0, 1, \dots, P, \dots$$

- ▶ (物理メモリ量 + 1 ページ) を何度もスキャン
- ▶ ≈ 物理メモリを上回る配列の全てを何度も同じ順番でアクセス



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

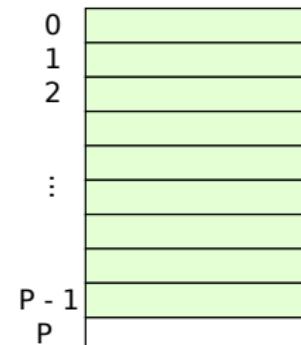
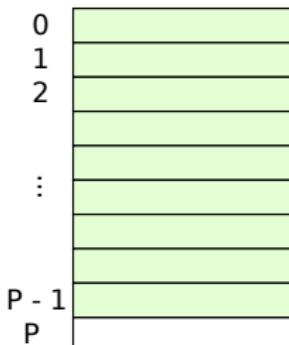
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

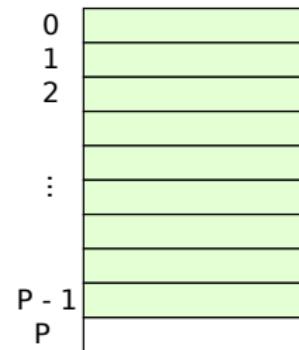
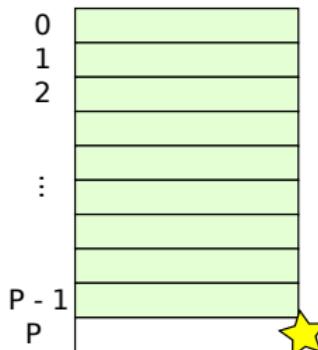
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

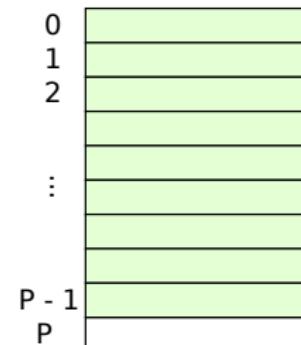
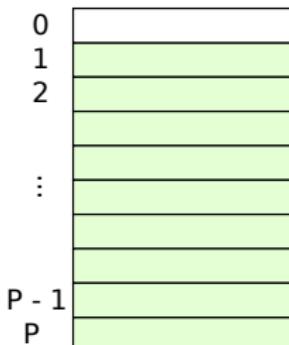
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

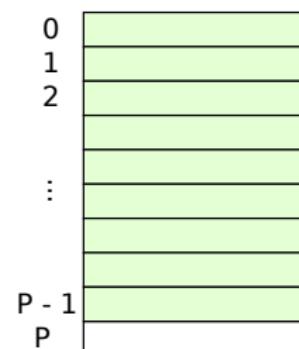
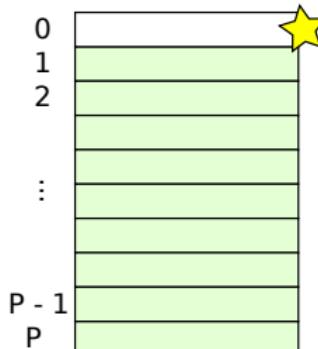
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

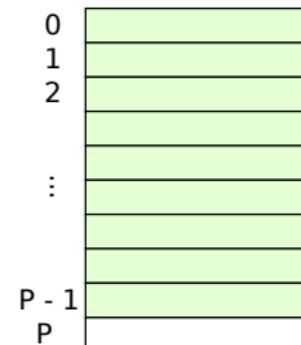
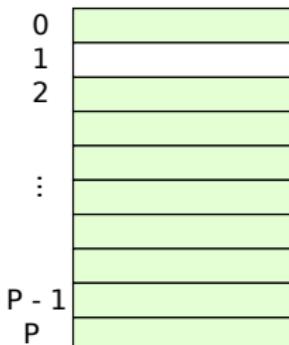
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

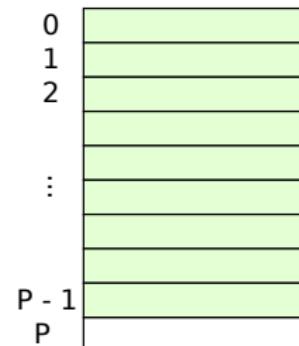
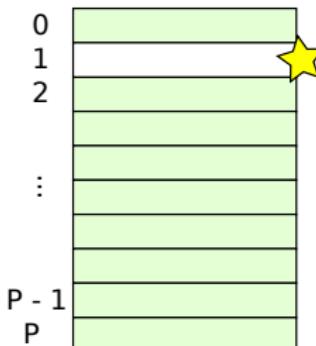
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

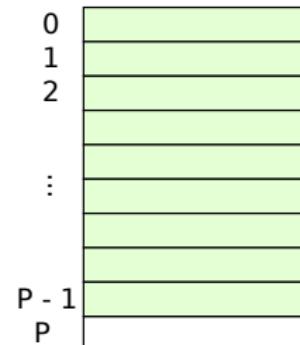
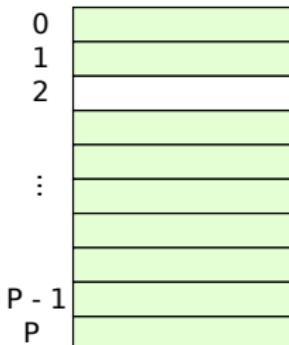
$$(q + 1) \mod (P + 1)$$

をページアウト

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

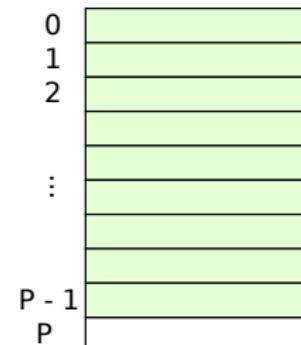
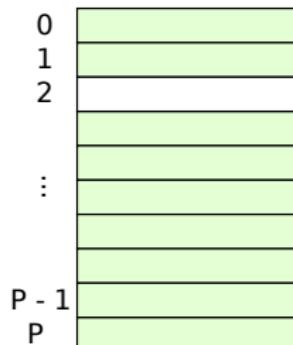
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

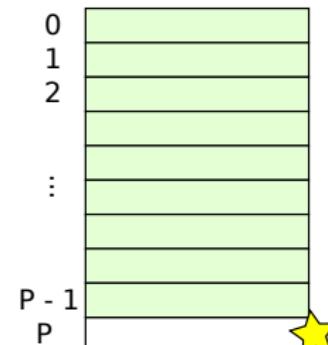
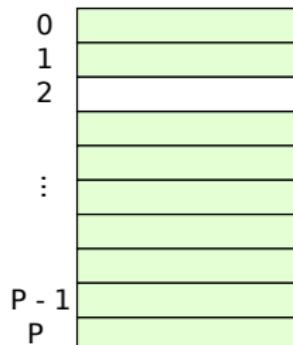
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

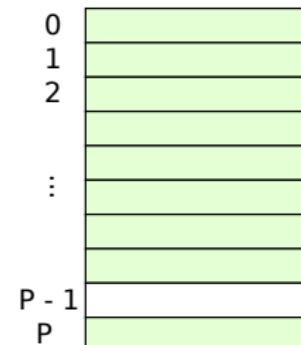
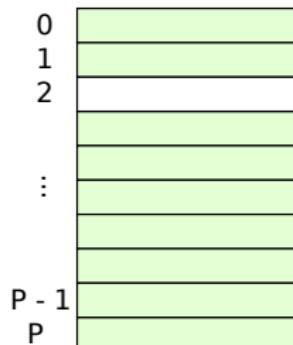
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

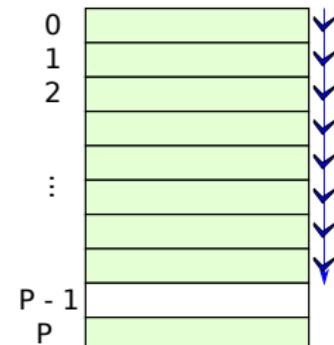
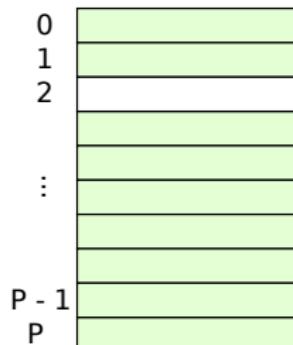
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

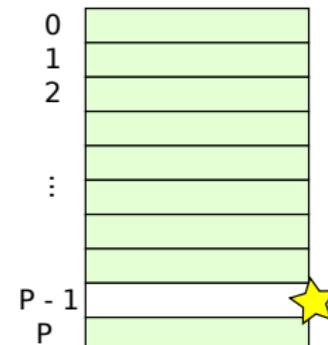
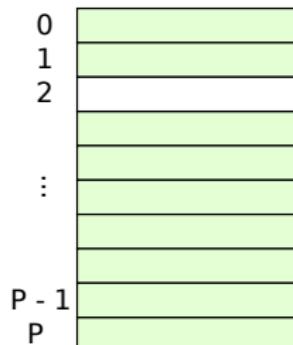
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

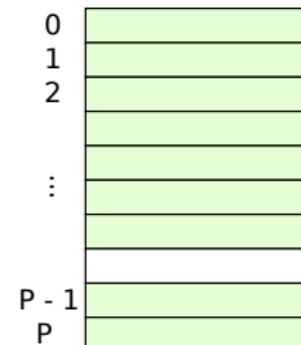
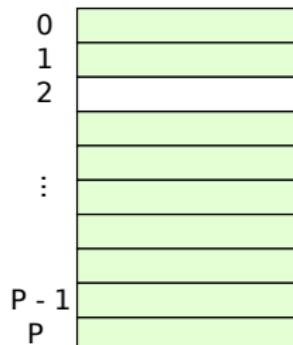
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

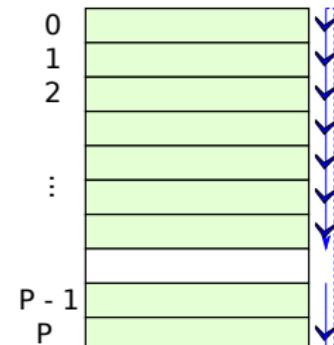
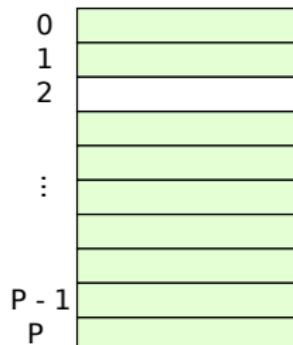
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

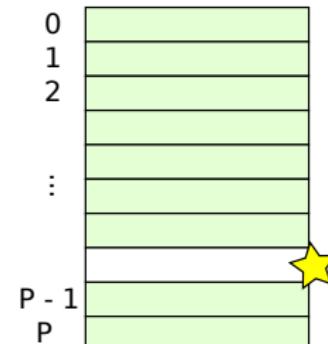
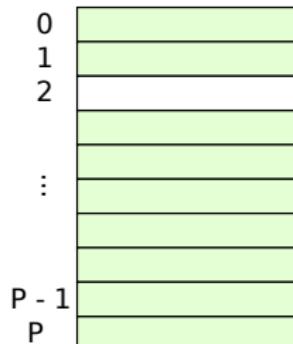
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

▶ 最悪

$$(q + 1) \mod (P + 1)$$

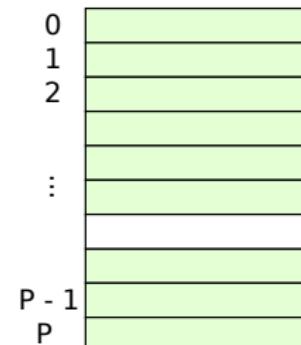
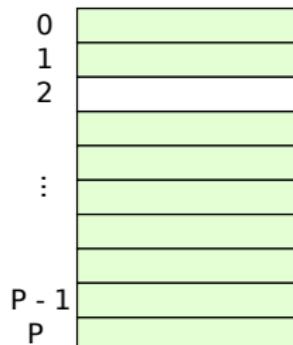
をページアウト

▶ ページFault率 1

▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト



最悪 vs. 最適

ページ q をアクセスしてページアウトが必要になったとき,

- ▶ 最悪

$$(q + 1) \mod (P + 1)$$

をページアウト

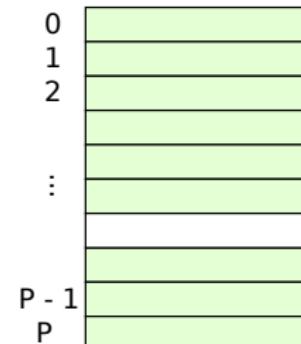
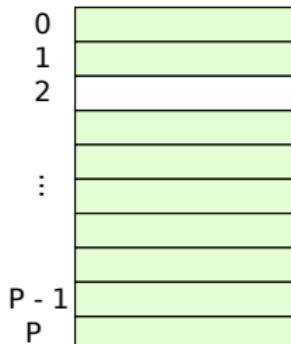
- ▶ ページFault率 1

- ▶ 最適(と思われる)アルゴリズム

$$(q - 1) \mod (P + 1)$$

をページアウト

- ▶ ページFault率 $1/P$



実践的な注

- ▶ ページ置換が重要とは言え、実践的には、ページ置換は頻繁に起きてはならない
- ▶ 起きているのならメモリが足りない
- ▶ ページ置換が頻繁に起きる状態をスラッシングといい、メモリが足りない（またはアプリがメモリを使いすぎている）徴候

オフライン問題に対する最適アルゴリズム

- ▶ 最適アルゴリズム (Bélády の OPT): 状態 R_i で a_i がアクセスされたとき,
 - ▶ $a_i \in R_i \Rightarrow$ 何もしない $\iff R_{i+1} = R_i$ (当然)
 - ▶ $a_i \notin R_i \Rightarrow$ 常駐ページのうち, 次にアクセスされるまでの時間が最も長いページを置換する (次におこるページ置換を極力先延ばしにする)
- ▶ 例

$$\begin{aligned} & A_{\text{OPT}}(\{1, 3, 7, 9\}, *, [2, 6, \underline{1}, 6, \underline{9}, \underline{7}, 4, 1, 5, \underline{3}, 3, 5 \dots]) \\ &= \{1, 3, 7, 9\} \setminus \{3\} \cup \{2\} \\ &= \{1, 2, 7, 9\} \end{aligned}$$

- ▶ 前掲「重要性を確認する問題」に適用してみよ

証明のヒント

- ▶ 最適アルゴリズム A_{OPT} の系列

$$\mathbf{R} = R_0 \xrightarrow{a_0} R_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} R_n$$

任意のアルゴリズムの系列

$$\mathbf{S} = S_0 \xrightarrow{a_0} S_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} S_n$$

に対し以下を n (系列の長さ) に関する帰納法で

$$\text{REP}(\mathbf{R}) \leq \text{REP}(\mathbf{S}) + |S_0 - R_0| \quad (\dagger)$$

- ▶ これが言えれば $S_0 = R_0$ とすることで証明が完了

オンライン問題に対してはどんなアルゴリズムの最悪ケースも同じ

- ▶ 明らかに、どんなアルゴリズムも、 $P < L$ である限り、物理メモリ上にないページが直後にアクセスされることを避けられない
- ▶ 何をやっても「最悪のケース（ページフォルト率 1）」は避けられない

実際のアルゴリズム

- ▶ オフラインの最適アルゴリズムは無理, オンラインの最適アルゴリズムは考えるだけ無駄
- ▶ どうするか?
 - ▶ 「次に使われるのがいつか (すぐか, 当分先か)」の予想に基づく
 - ▶ それに基づき, すぐ使われる (可能性が高そうな) ものを残す
- ▶ どう予想するか?
 - ▶ 経験則: 最近使われたものはまたすぐ使われる
- ▶ ⇒ アルゴリズム例
 1. LRU
 2. FIFO
 3. LRU の近似 (セカンドチャンスまたはクロック)
 4. エイジング

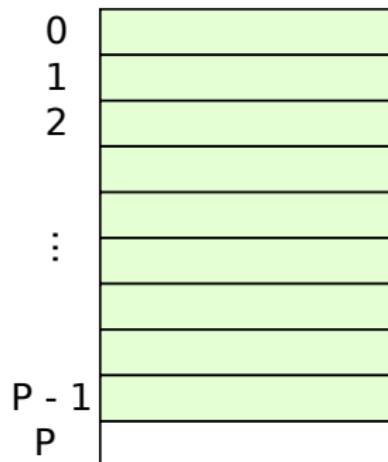
LRU 置換

- ▶ ページアウト時に, Least Recently Used (過去にアクセスされた時点が最も遠い) なページを置換する
 - ▶ しばらく使われていないページは今後も当分使われない
 - ▶ ≈ 最近使われたページは近い将来またすぐ使われる
- ▶ 根拠となるプログラムの性質: アクセスの局所性
 - ▶ 空間的局所性: あるアドレスをアクセスした直後, 近い(例: 同じページの別の)アドレスをアクセスする傾向
 - ▶ 時間的局所性: あるアドレスをアクセスしてからあまり間をおかずに, 同じアドレスをアクセスする傾向
- ▶ LRU 置換は, 未来のアクセスが過去の反転であるとして, A_{OPT} を適用しているのと同じ

$$\begin{aligned} & A_{\text{LRU}}(R, [a_0, \dots, a_{i-1}], [a_i]) \\ = & A_{\text{OPT}}(R, [a_0, \dots, a_{i-2}, a_{i-1}], [a_i, a_{i-1}, a_{i-2}, \dots]) \end{aligned}$$

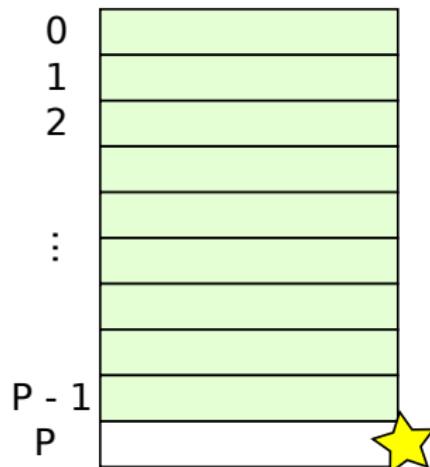
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



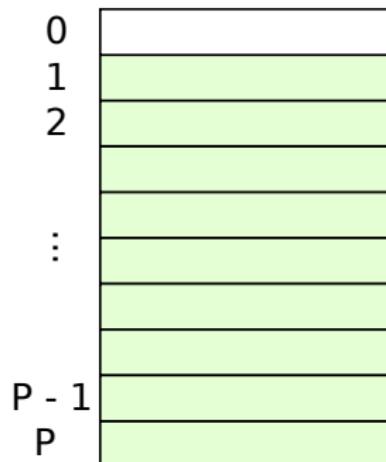
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



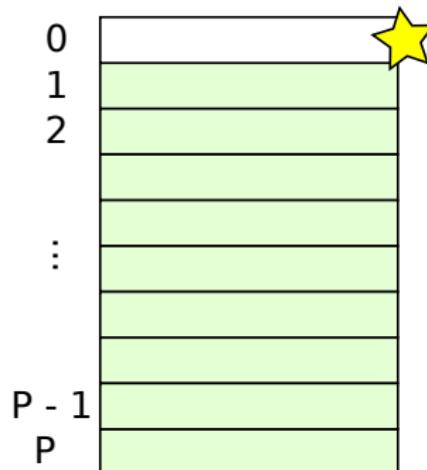
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



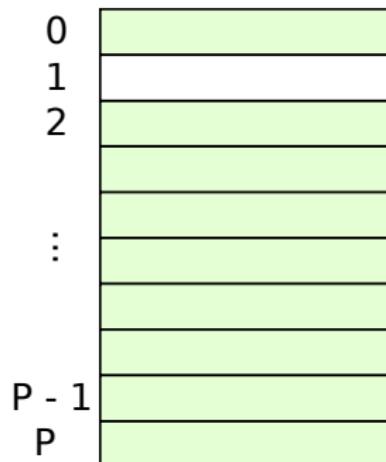
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



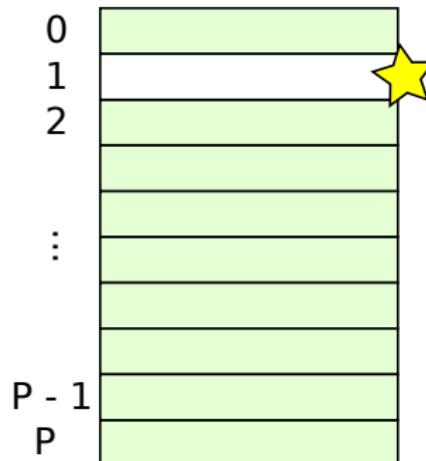
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



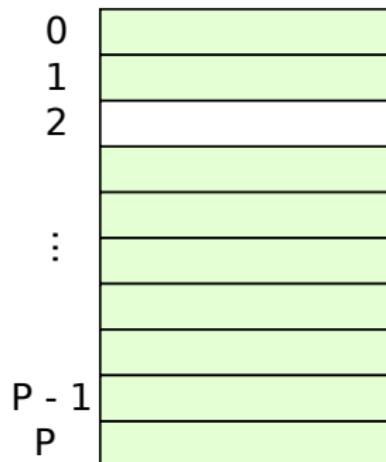
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



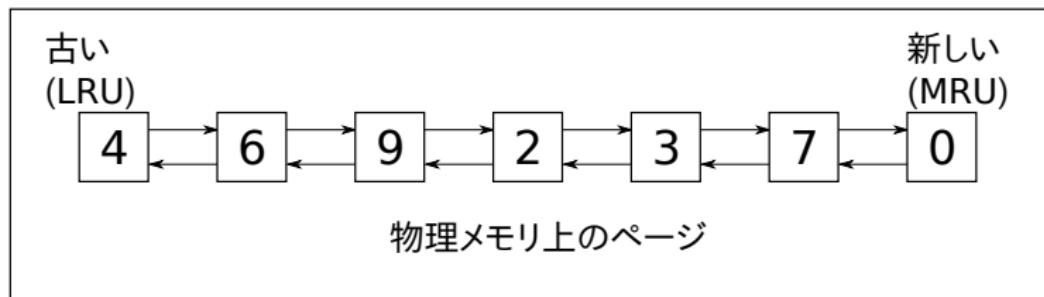
若干の脱線

- ▶ LRU 置換は多くの現実的なアクセスパターンで良い性能を示すが、「配列のスキヤンの繰り返し」というよくあるパターンで、「配列のサイズ > 物理メモリ量」となった途端に最悪の性能を示す
- ▶ 前掲「重要性を確認する問題」で、LRU 置換が最悪の性能（ページフォルト率 1）になることを確認せよ



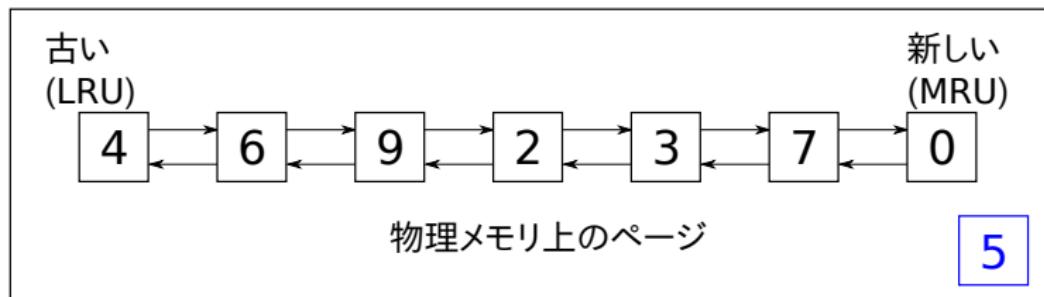
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



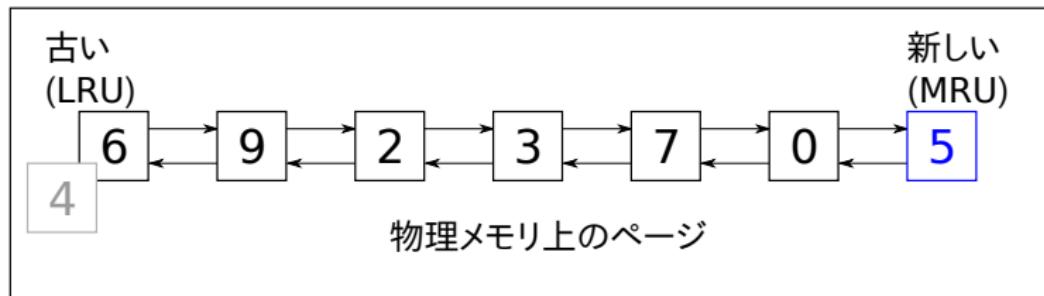
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



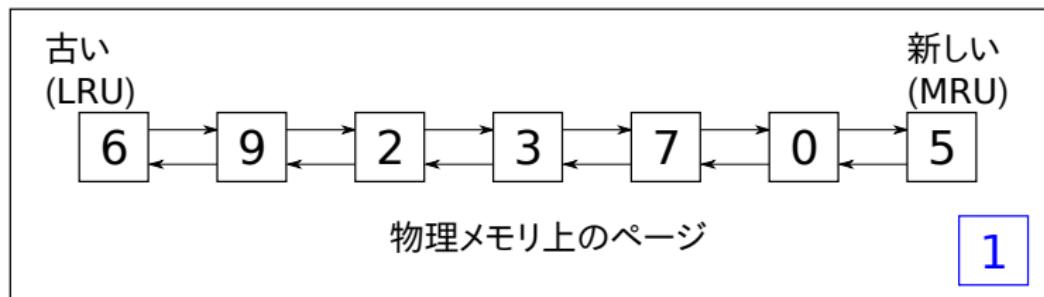
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



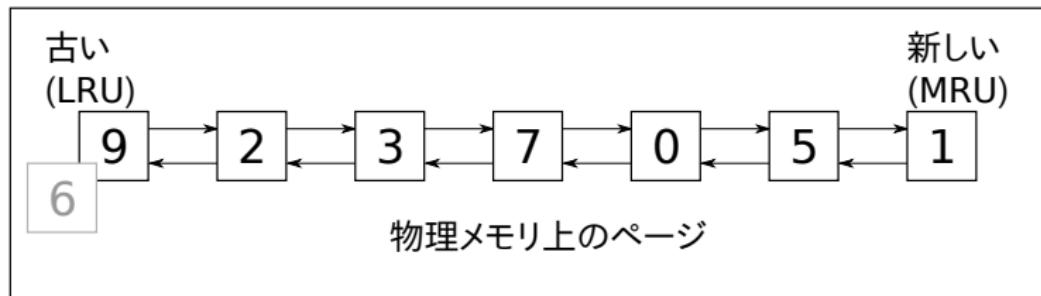
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



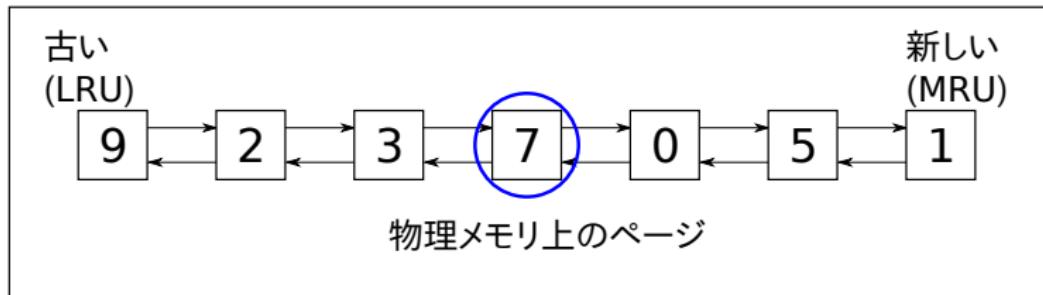
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



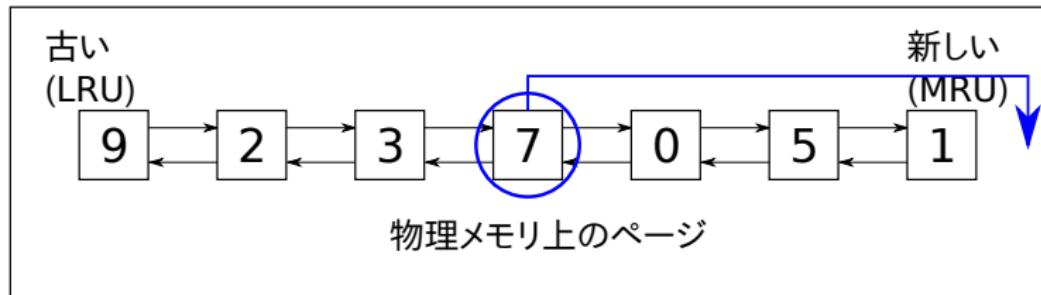
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



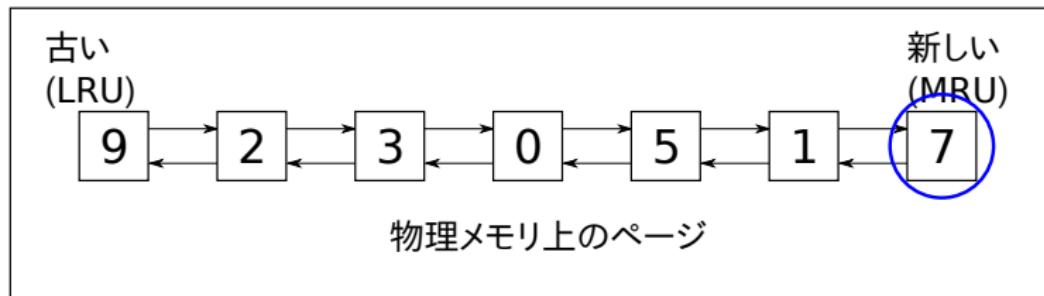
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



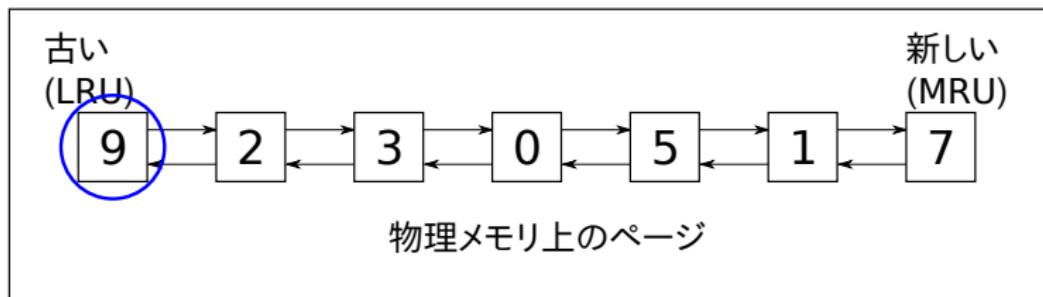
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



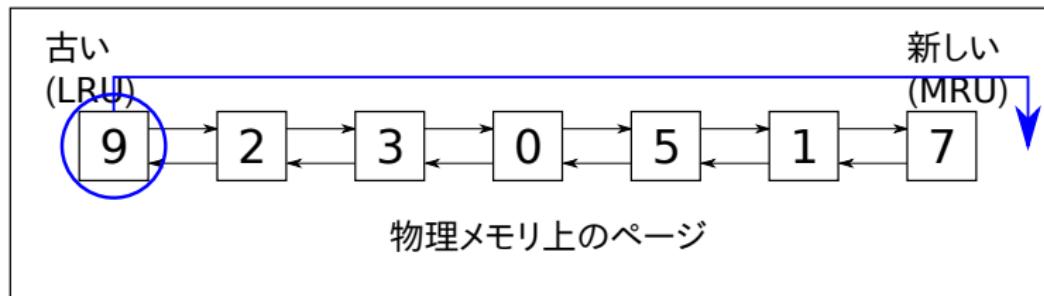
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



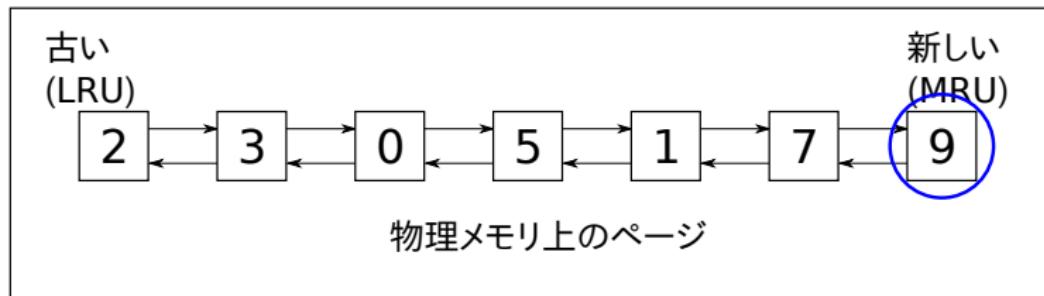
LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動



LRU実装に必要なデータ構造

- ▶ 物理メモリ上の P ページを「LRU 順 (先頭が Least, 末尾が Most Recently Used)」に保持 (LRU リスト)
 - ▶ 物理メモリ上にないページへのアクセス (ページフォルト発生) 時
 - ▶ リスト先頭 (図の左端) のページを除去
 - ▶ アクセスされたページは末尾へ挿入
 - ▶ 物理メモリ上有るページへのアクセス時
 - ▶ そのページをリストの末尾 (図の右端) へ移動

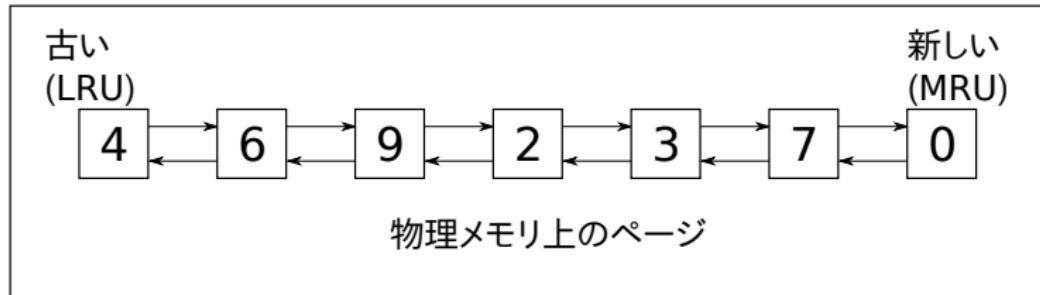


LRU 実装の困難さ

- ▶ 物理メモリ上にあるページへのアクセスの度にデータ構造を変更する必要がある
 - ▶ 例外は発生しないので、ソフトウェアで介入は困難
 - ▶ 特別なハードウェアの仕組みが必要（メモリアクセスのたびに介入）
- ▶ ⇒ LRU 置換の近似 (Not Recently Used (NRU) 置換)
- ▶ ≈ 「最近使われたかどうか」を大雑把に把握する

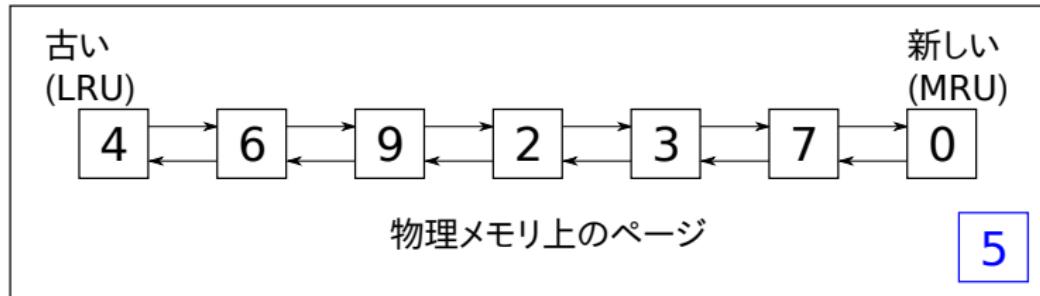
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



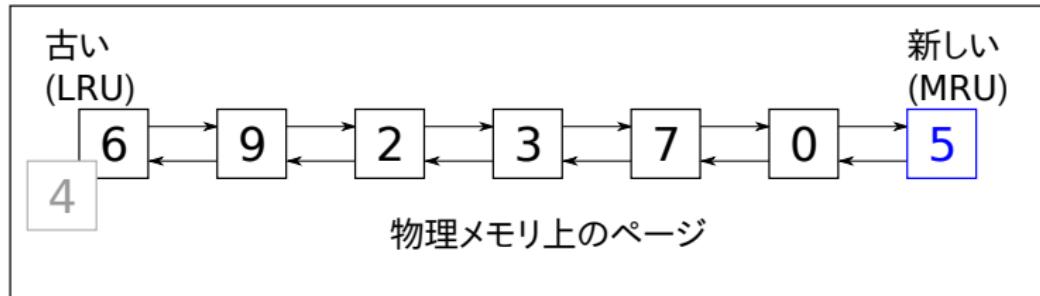
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



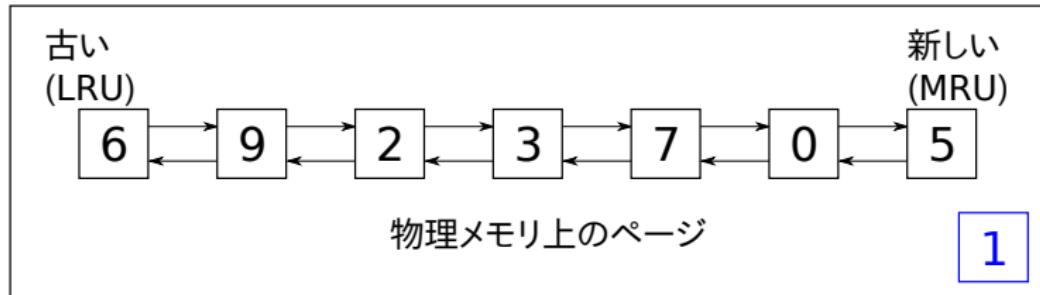
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



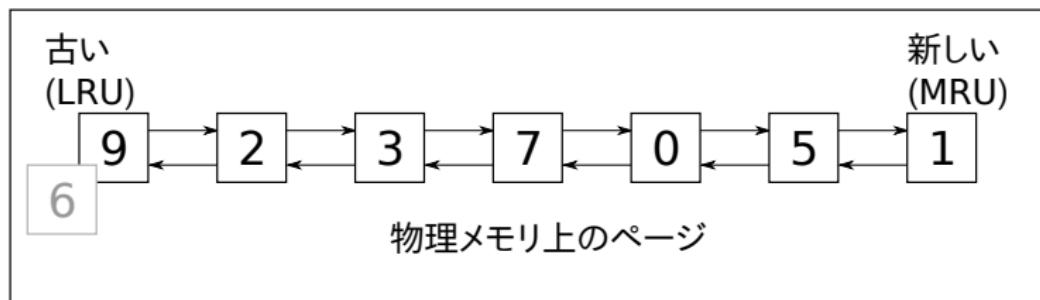
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



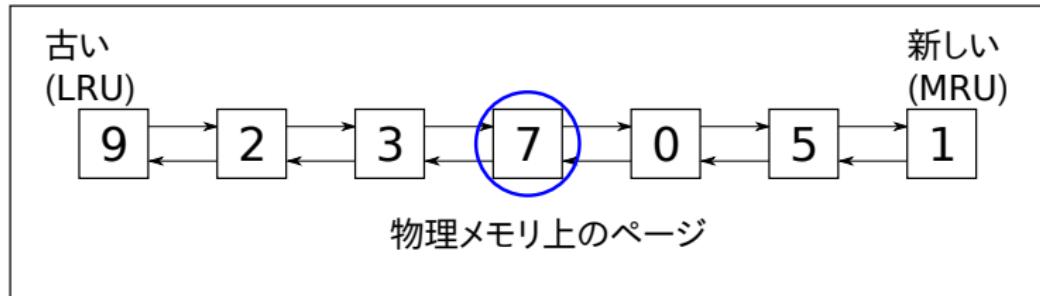
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



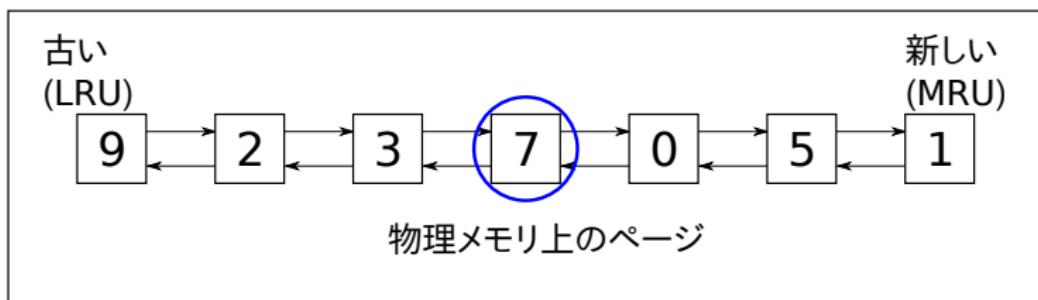
FIFO

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



FIFO

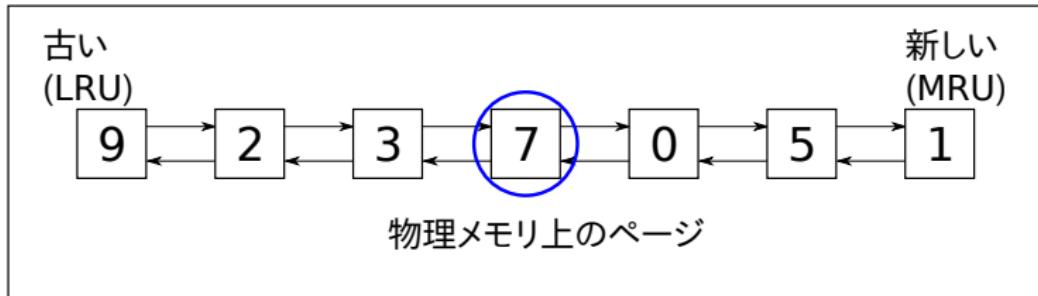
- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



- ▶ 問題点: ページインしてから使われたものとそうでないものを区別できない

FIFO

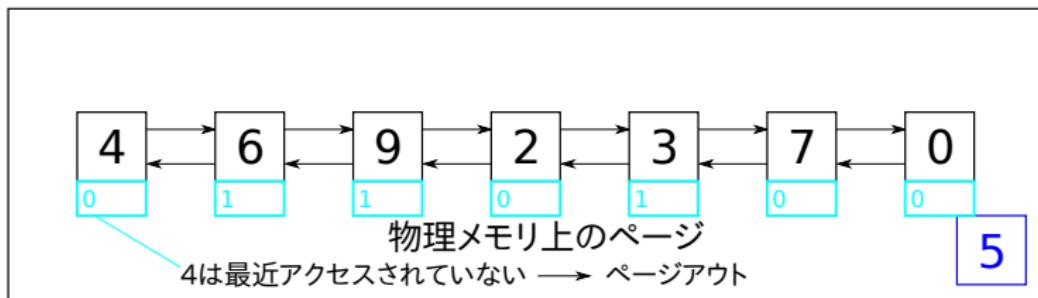
- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時
 - ▶ LRU と同じ
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ 何もしない



- ▶ 問題点: ページインしてから使われたものとそうでないものを区別できない
- ▶ ⇒ セカンドチャンスまたはクロックアルゴリズム

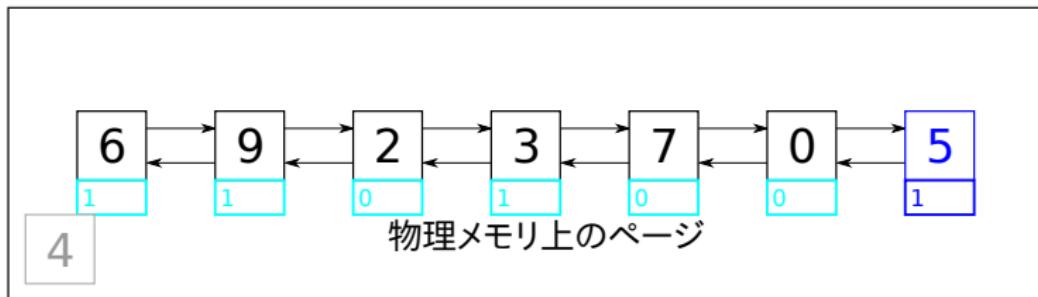
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



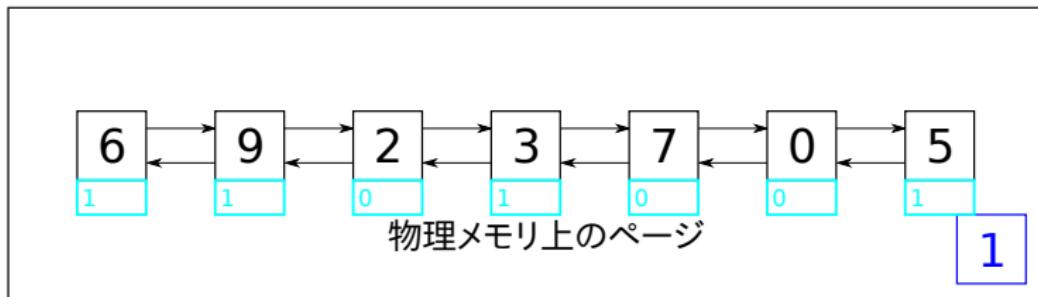
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



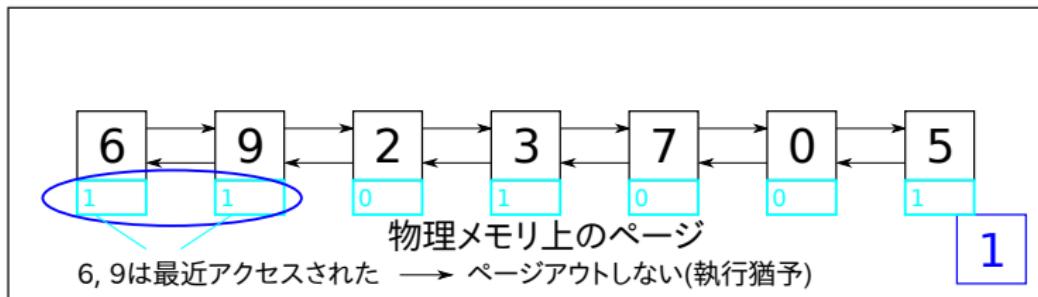
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



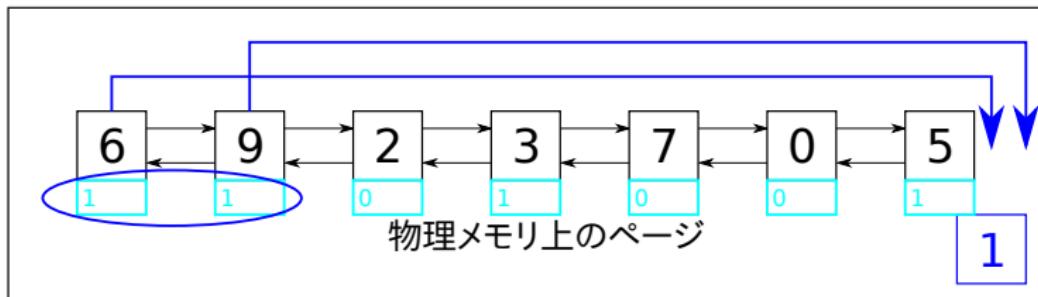
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



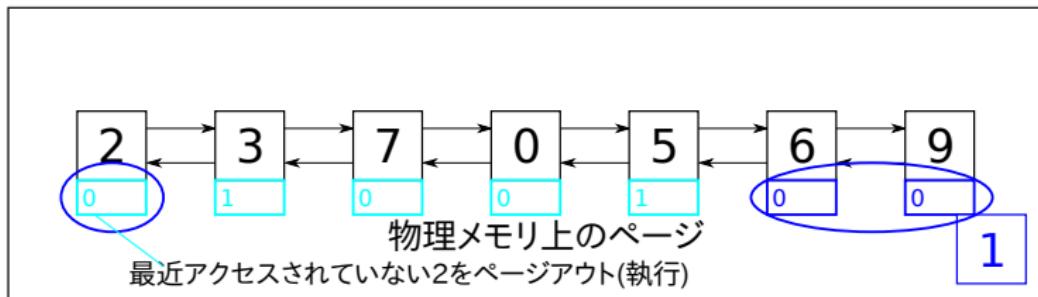
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば、0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



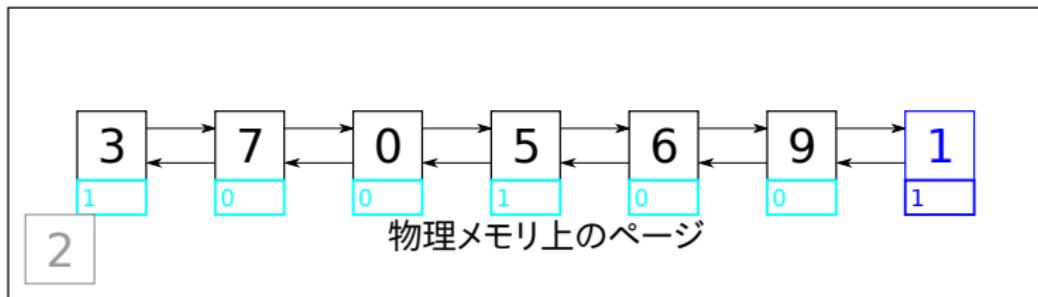
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



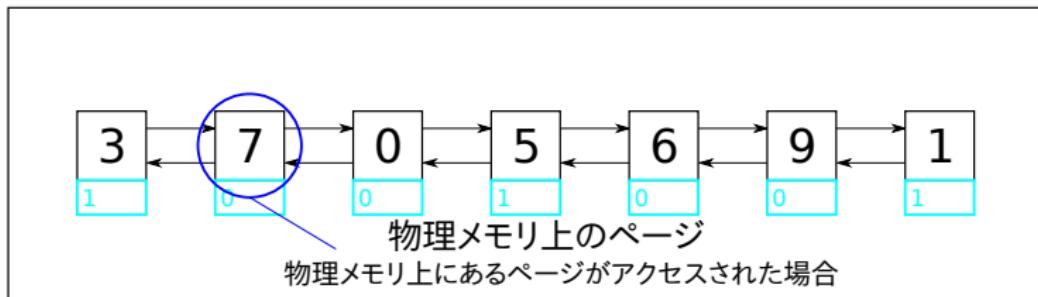
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



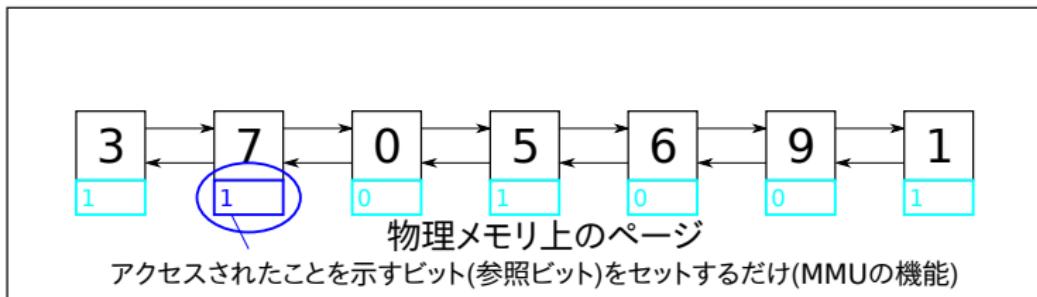
セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)



セカンドチャンス

- ▶ 物理メモリ上の P ページを「ページインされた順」に保持 (FIFO)
- ▶ 物理メモリ上にあるページへのアクセス時
 - ▶ (MMU が) そのページの参照ビットを 1 にする
- ▶ 物理メモリ上にないページへのアクセス (ページ fault 発生) 時; リスト先頭のページの参照ビットが
 - ▶ = 0 ならばページアウト
 - ▶ = 1 ならば, 0 にしてリスト末尾へ再挿入 (復活のチャンス, 執行猶予)

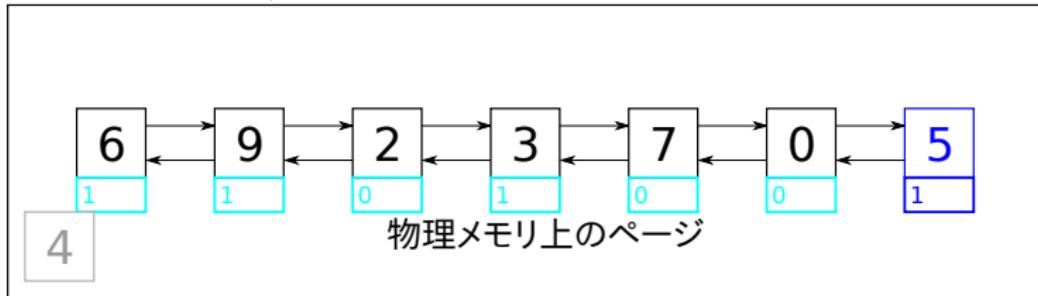


クロックアルゴリズム

- ▶ 動作はセカンドチャンスと同じ
- ▶ 二重リンクリストの代わりに, 循環バッファ(配列)を使う(途中から要素を削除する事はないのでそれで十分)
- ▶ 詳細は省略(データ構造の練習問題)

セカンドチャンスの性質

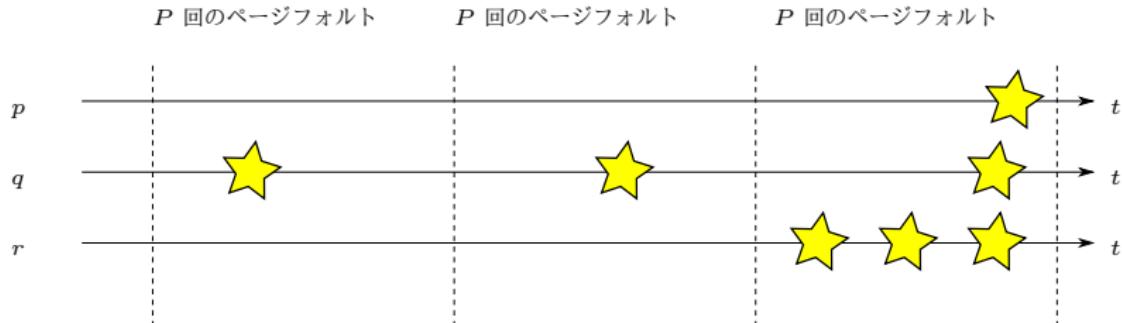
- ▶ 参照ビットは、「最近アクセスがあったか否か」の指標



- ▶ LRUリストの先頭から i 番目(左端が0番目)のページの参照ビットが1 $\iff (P - i)$ 回前(直前が1回前)のページフォルト時またはそれ以降にアクセスされている (†)
 - ▶ 注: (†)はページフォルト時に「物理メモリ上の全ページの参照ビットが1」というケースを考えるとは少し不正確だが、本質的ではないので深入りしない

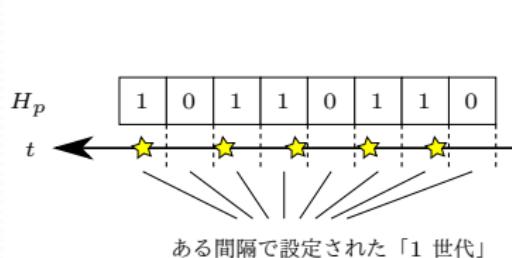
セカンドチャンスの限界

- ▶ P 回のページフォルトより短い粒度や、それ以前のアクセス履歴は用いていない
- ▶ 例えば以下のようなアクセス履歴を持つページ (p, q, r) を区別できない



エイジング

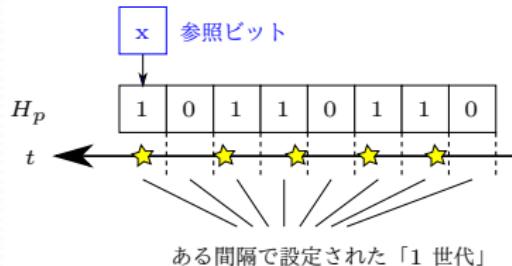
- ▶ 物理メモリ上の各ページ p に、一定間隔の区間ごとの「アクセスあり・なし」を、 N 世代分記録したカウンタ値 H_p を持たせる



- ▶ 「1 世代」 = 一定の実時間かページ置換回数

エイジング

- ▶ 物理メモリ上の各ページ p に、一定間隔の区間ごとの「アクセスあり・なし」を、 N 世代分記録したカウンタ値 H_p を持たせる



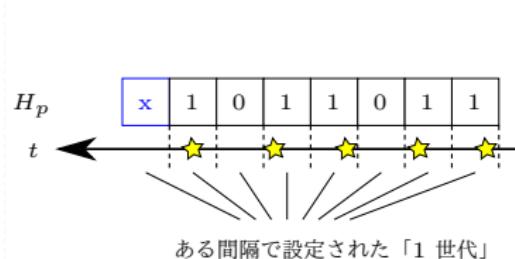
- ▶ 「1世代」 = 一定の実時間かページ置換回数
- ▶ 世代が変わる時, H_p を decay させ, 直近のアクセスある・なしを反映

$$\begin{cases} H_p &= (H_p \gg 1) + (R_p \ll (N - 1)) \\ R_p &= 0 \end{cases}$$

- ▶ R_p : p の参照ビット

エイジング

- ▶ 物理メモリ上の各ページ p に、一定間隔の区間ごとの「アクセスあり・なし」を、 N 世代分記録したカウンタ値 H_p を持たせる



- ▶ 「1世代」 = 一定の実時間かページ置換回数
- ▶ 世代が変わる時、 H_p を decay させ、直近のアクセスある・なしを反映

$$\begin{cases} H_p &= (H_p \gg 1) + (R_p \ll (N - 1)) \\ R_p &= 0 \end{cases}$$

- ▶ R_p : p の参照ビット
- ▶ ページ置換時: カウンタ値 H_p が最小のページを退避

現実の実装における考慮

現実の実装ではこれまでの考察に加え以下を考慮する

- ▶ ページアウトは物理メモリが完全に満杯する前に(大部分埋まったところで), ページ�オルトと非同期に始める
- ▶ ページアウト対象を選ぶ際は, ページが最近アクセスされているかだけではなく, 2次記憶への書き込みが必要かも考慮する
 - ▶ あるページが初めてページアウトされるとき ⇒ 書き込み必要
 - ▶ あるページがページインしてから変更されているとき ⇒ 書き込み必要
 - ▶ あるページがページインしてから変更されていないとき ⇒ 書き込み不要(単に物理メモリを開放すれば良い)
- ▶ ページアウトは1ページ毎ではなく, 連続した数十ページをまとめて行った方が, IO性能の効率がよい

ページングを制御する API

- ▶ `int e = posix_madvise(addr, length, adv);`
- ▶ `int e = madvise(addr, length, adv);`
- ▶ 両者は似ているが後者は Linux 特有であり、機能が多い
- ▶ *adv* に $[addr, addr+length)$ の範囲のアクセスパターンを教える
 - ▶ (POSIX_)MADV_NORMAL
 - ▶ (POSIX_)MADV_SEQUENTIAL : 逐次的にアクセスする
 - ▶ (POSIX_)MADV_RANDOM : ランダムにアクセスする
 - ▶ (POSIX_)MADV_WILLNEED : 近い将来必要
 - ▶ POSIX_MADV_DONTNEED : 当分不要
 - ▶ MADV_DONTNEED : 不要 (値が失われても良い)
 - ▶ MADV_COLD : ページアウト候補対象に
 - ▶ MADV_PAGEOUT : 即座にページアウト

madvise の利用法

- ▶ OS が必要とするのはどれをページアウトすべきか
- ▶ 特に MADV_PAGEOUT は直接それをアプリケーションから OS に教えることを可能にする
- ▶ アプリケーションが今後のアクセスパターンを知つていれば, A_{OPT} に従つて, ページアウト対象を決めれば良い
- ▶ それ以外の *adv* の効果は不明