

ファイルシステム

田浦健次郎

目次

ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

mmap の実装, 主記憶と 2 次記憶の統合

ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

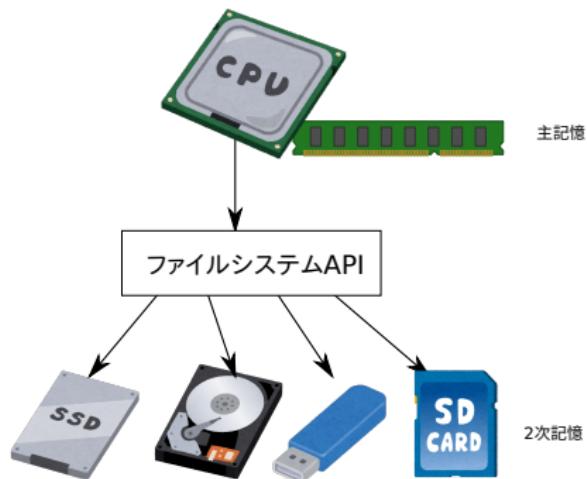
キヤツシユ

先読み

mmap の実装、主記憶と 2 次記憶の統合

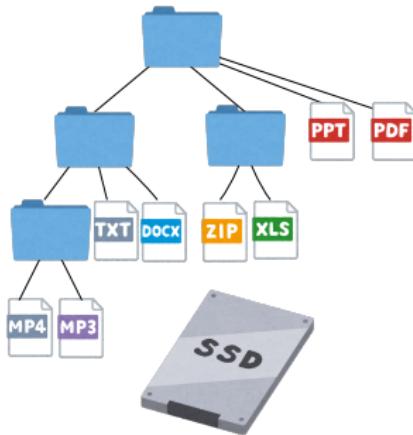
ファイルシステムとは

- ▶ 第一義的には、2次記憶装置に対してOSが提供している抽象化
- ▶ 2次記憶装置
 - ▶ ハードディスク
 - ▶ SSD
 - ▶ USBメモリ
 - ▶ SD card
 - ▶ etc.
- ▶ 色々な2次記憶装置を簡便な、共通のインターフェースで読めるようにする



ファイルシステムが提供する基本的な抽象化

- ▶ ファイル
 - ▶ ≈自由に伸縮できるバイト配列
- ▶ ディレクトリ, フォルダ(階層構造)
 - ▶ 他のファイルやディレクトリを含むことが出来る「入れ物」
- ▶ 名前空間
 - ▶ ファイル, ディレクトリに名前を付けられる



ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

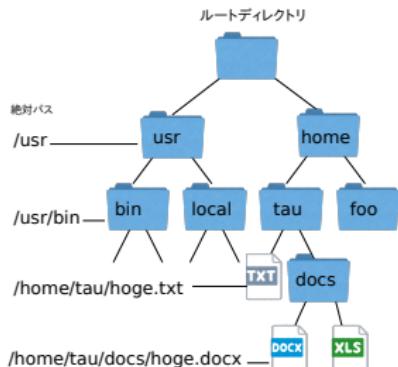
mmap の実装, 主記憶と 2 次記憶の統合

open, close

- ▶ `int fd = open(path, flags[, mode]);`
 - ▶ `path` で指定されるファイルを開く, 場合により, 新たに作る
 - ▶ 成功すれば, 後に `read/write` などに渡せる, ファイルディスクリプタ `fd` を返す (実体はただの整数)
 - ▶ `flags`
 - ▶ `O_RDONLY`, `O_WRONLY`, `O_RDWR` : 読むだけ, 書くだけ, 両方
 - ▶ `O_CREAT` : 存在しなければ作る
 - ▶ `O_EXCL` : 存在していたらエラー
 - ▶ `O_TRUNC` : 存在していたら一旦空 (0 バイト) にする
 - ▶ etc.
- ▶ ファイルディスクリプタ (`open` されたファイル) に紐付いている情報
 - ▶ ファイルオフセット : 次の読み書きが行われる場所
 - ▶ `open` 直後は 0

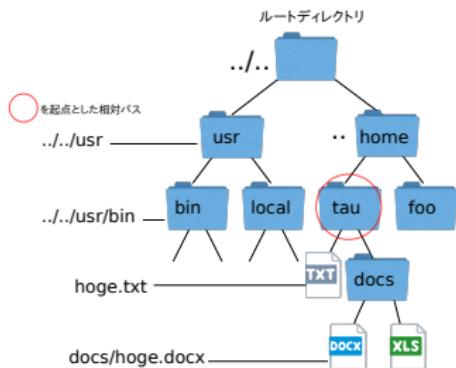
ファイルシステムの構成とパス(path)について

- ▶ ファイルシステム全体は、一つの大きな木構造
- ▶ 木構造の
 - ▶ リーフが(通常の) **ファイル**
 - ▶ リーフでない内部ノードは、**ディレクトリ**(フォルダとも言う)
- ▶ 任意のファイル・ディレクトリは木構造の根(**ルートディレクトリ**という)からそれに至る道(**絶対パス**)を指定することで一意に特定可能
 - ▶ 絶対パスの例 `/usr/bin/python`: ルートディレクトリ内の `usr` 内の `bin` 内の `python`



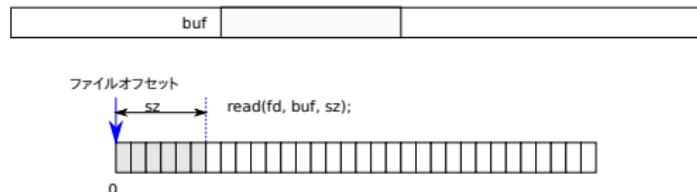
相対パスとカレントディレクトリ

- ▶ 任意のディレクトリを起点としたファイル・ディレクトリへのパスを相対パスという
- ▶ 特に親ディレクトリを .. というパスで指定する
- ▶ これで、任意の起点から任意のファイル・ディレクトリを指定可能
- ▶ 各プロセスは「現在のディレクトリ」(カレントディレクトリ)という属性があり、相対パスは、通常カレントディレクトリからの相対パス
- ▶ カレントディレクトリの変更: chdir システムコール、シェルの cd コマンド



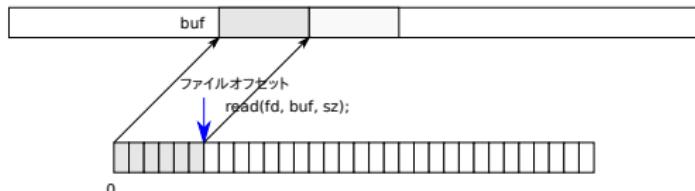
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進



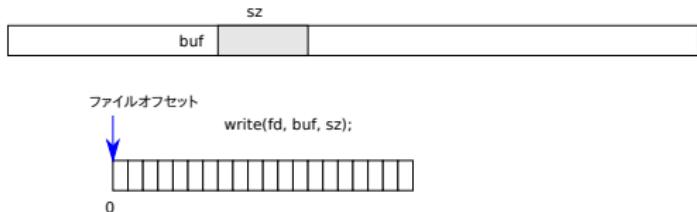
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進



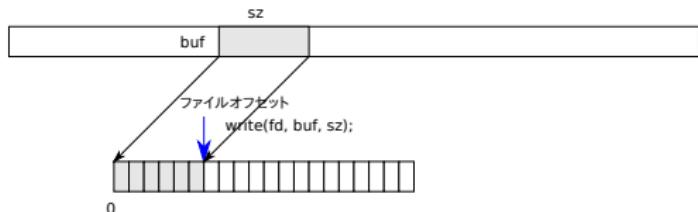
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進
- ▶ `ssize_t w = write(fd, buf, sz);`
 - ▶ buf から始まる領域から, 最大 sz バイトを, ファイルオフセットに書き込む
 - ▶ 場合によりファイルが伸長する
 - ▶ 実際に書き込まれたバイト数を返す
 - ▶ ファイルオフセットが w 前進



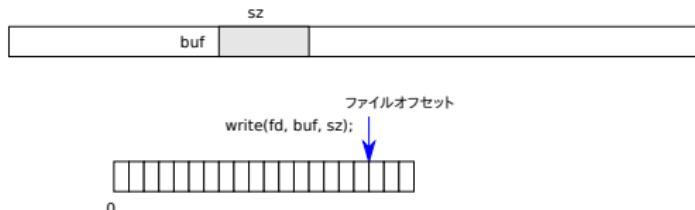
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進
- ▶ `ssize_t w = write(fd, buf, sz);`
 - ▶ buf から始まる領域から, 最大 sz バイトを, ファイルオフセットに書き込む
 - ▶ 場合によりファイルが伸長する
 - ▶ 実際に書き込まれたバイト数を返す
 - ▶ ファイルオフセットが w 前進



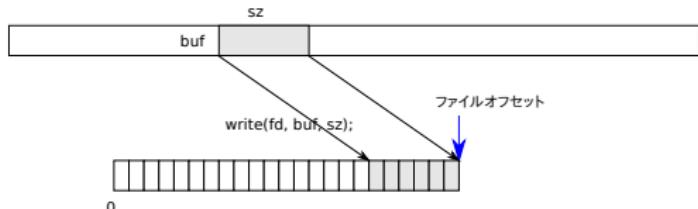
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進
- ▶ `ssize_t w = write(fd, buf, sz);`
 - ▶ buf から始まる領域から, 最大 sz バイトを, ファイルオフセットに書き込む
 - ▶ 場合によりファイルが伸長する
 - ▶ 実際に書き込まれたバイト数を返す
 - ▶ ファイルオフセットが w 前進



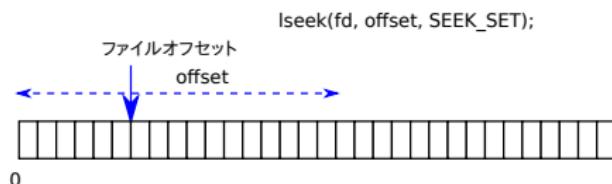
read, write

- ▶ `ssize_t r = read(fd, buf, sz);`
 - ▶ ファイルオフセットから最大 sz バイト, buf から始まる領域に読み込む
 - ▶ 実際に読み込まれたバイト数を返す
 - ▶ ファイルオフセットが r 前進
- ▶ `ssize_t w = write(fd, buf, sz);`
 - ▶ buf から始まる領域から, 最大 sz バイトを, ファイルオフセットに書き込む
 - ▶ 場合によりファイルが伸長する
 - ▶ 実際に書き込まれたバイト数を返す
 - ▶ ファイルオフセットが w 前進



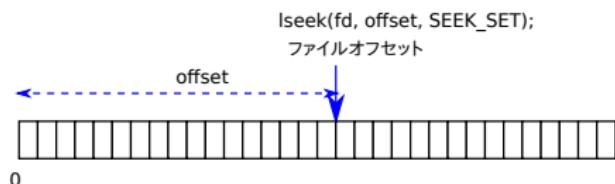
lseek

- ▶ `off_t o = lseek(fd, offset, whence);`
 - ▶ ファイルオフセット(次に read/write が作用する場所)を *whence* に応じた *offset* にする
 - ▶ これで、ファイルの一部だけを読み書きできる
 - ▶ SEEK_SET : ファイル先頭から
 - ▶ SEE_CUR : 現在位置から
 - ▶ SEEK_END : ファイルの終わりから



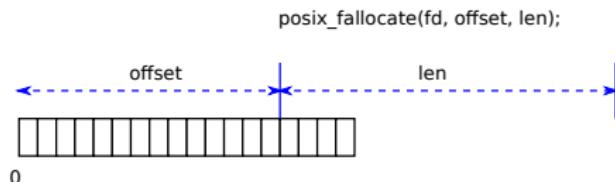
lseek

- ▶ `off_t o = lseek(fd, offset, whence);`
 - ▶ ファイルオフセット (次に read/write が作用する場所) を *whence* に応じた *offset* にする
 - ▶ これで、ファイルの一部だけを読み書きできる
 - ▶ SEEK_SET : ファイル先頭から
 - ▶ SEE_CUR : 現在位置から
 - ▶ SEEK_END : ファイルの終わりから



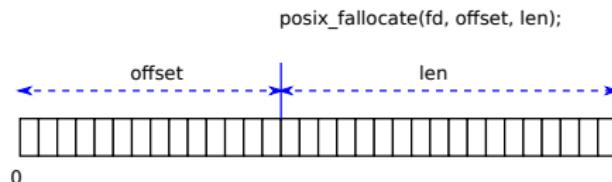
fallocate, ftruncate

- ▶ `int e = posix_fallocate(fd, offset, len);`
- ▶ `int e = fallocate(fd, mode, offset, len);`
 - ▶ ファイルの $[offset, offset + len)$ バイトの範囲を、必要ならばファイルを伸長し、0にする
 - ▶ `fallocate` は Linux 固有 (man -s 2 ftruncate 参照)



fallocate, ftruncate

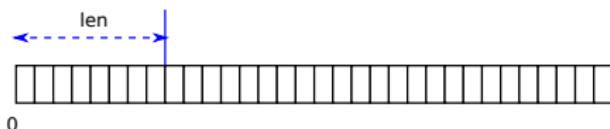
- ▶ `int e = posix_fallocate(fd, offset, len);`
- ▶ `int e = fallocate(fd, mode, offset, len);`
 - ▶ ファイルの $[offset, offset + len)$ バイトの範囲を、必要ならばファイルを伸長し、0にする
 - ▶ `fallocate` は Linux 固有 (man -s 2 ftruncate 参照)



fallocate, ftruncate

- ▶ `int e = posix_fallocate(fd, offset, len);`
- ▶ `int e = fallocate(fd, mode, offset, len);`
 - ▶ ファイルの $[offset, offset + len)$ バイトの範囲を、必要ならばファイルを伸長し、0にする
 - ▶ `fallocate` は Linux 固有 (man -s 2 ftruncate 参照)
- ▶ `int e = ftruncate(fd, off_t len);`
 - ▶ ファイルの大きさを `len` バイトに、短縮 (伸長できる OS もある) する

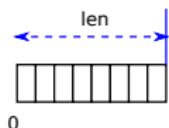
`ftruncate(fd, len);`



fallocate, ftruncate

- ▶ `int e = posix_fallocate(fd, offset, len);`
- ▶ `int e = fallocate(fd, mode, offset, len);`
 - ▶ ファイルの $[offset, offset + len)$ バイトの範囲を、必要ならばファイルを伸長し、0にする
 - ▶ `fallocate` は Linux 固有 (man -s 2 ftruncate 参照)
- ▶ `int e = ftruncate(fd, off_t len);`
 - ▶ ファイルの大きさを `len` バイトに、短縮 (伸長できる OS もある) する

`ftruncate(fd, len);`



close, unlink

- ▶ `int e = close(fd);`
 - ▶ open したファイルを閉じる (*fd* が無効になり, 以降の open における再利用の対象になる)
- ▶ `int e = unlink(path);`
 - ▶ ≈ ファイルを消す

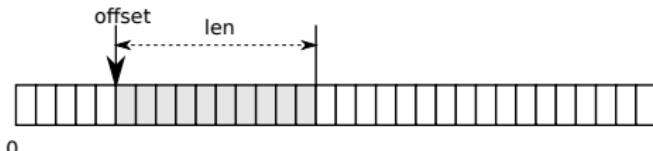
mmap

- ▶ 有用かつ奥深い API
- ▶ 色々な役割を兼ねる
 1. ファイルを「メモリ上にあるかのように」読み書きする
 2. メモリを割り当てる (sbrk の役割を兼ねる)
 3. 割り当てるアドレスを指定する (疎なアドレス空間)
 4. 読み書き保護を設定する

mmap API 基本

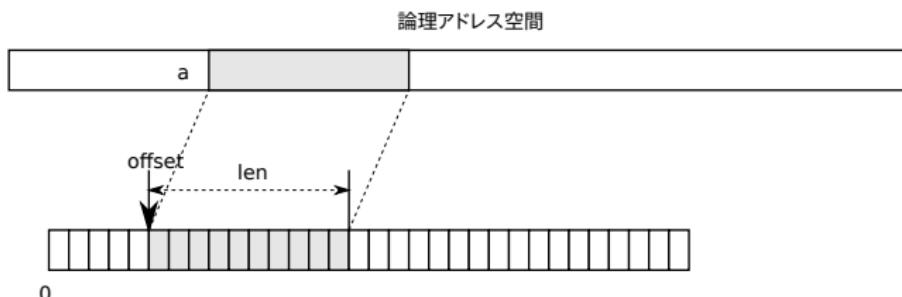
- ▶ `void * a = mmap(a0, len, prot, flags, fd, o);`
- ▶ 基本: fd に対応するファイルの o バイト目から len バイトが、返り値のアドレス a に「対応付けられる（マップされる）」
- ▶ $\iff a[i]$ がファイルの $(o + i)$ バイト目に対応
- ▶ 返り値 (a) これまで使われていなかったアドレス
- ▶ すなわち、メモリが新たに割り当てられる効果も持つ

論理アドレス空間



mmap API 基本

- ▶ `void * a = mmap(a0, len, prot, flags, fd, o);`
- ▶ 基本: fd に対応するファイルの o バイト目から len バイトが、返り値のアドレス a に「対応付けられる（マップされる）」
- ▶ $\iff a[i]$ がファイルの $(o + i)$ バイト目に対応
- ▶ 返り値 (a) これまで使われていなかったアドレス
- ▶ すなわち、メモリが新たに割り当てられる効果も持つ

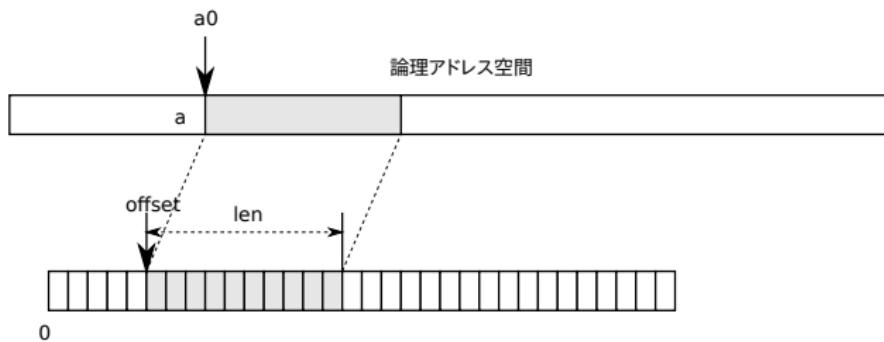


mmap API オプション

- ▶ `void * a = mmap(a0, len, prot, flags, fd, o);`
- ▶ *prot* : 割り当てられた領域の保護属性
 - ▶ PROT_EXEC : (命令として) 実行可能にする
 - ▶ PROT_READ : 読み出し可能にする
 - ▶ PROT_WRITE : 書き込み可能にする
- ▶ *flags* : MAP_SHARED または MAP_PRIVATE を指定; 複数プロセスが同じ領域を mmap した場合,
 - ▶ MAP_SHARED *a[i]* への書き込みは,
 1. ファイルへ反映される
 2. それらのプロセス間で共有される
 - ▶ MAP_PRIVATE *a[i]* への書き込みは,
 1. ファイルへ反映されない
 2. それらのプロセス間でも共有されない
 - ▶ その他のフラグは後述

mmap API アドレスの指定

- ▶ a_0
 - ▶ $a_0 (\neq 0) \rightarrow$ アドレス a_0 を割り当てるよう指定 (a_0 が空いていない場合はエラー)
 - ▶ $a_0 = 0 \rightarrow$ OS が、空いているところを割り当てる



mmapでファイル読み出し

```
1 int fd = open(filename, O_RDONLY);
2 struct stat sb[1];
3 fstat(fd, sb);
4 long sz = sb->st_size;
5 char * a = mmap(0, sz, PROT_READ, MAP_PRIVATE, fd, 0);
/* filename の i バイト目が a[i] で読み出せる */
6 for (long i = 0; i < sz; i++) {
7     putchar(a[i]);
8 }
9 }
```

- ▶ 注: エラーチェックは省略(以下同様)
- ▶ エラーチェックのやり方(`man err`参照)

```
1 if (fstat(fd, sb) == -1) err(1, "fstat");
```

mmapでファイル書き出し

```
1 int fd = open(filename, O_RDWR|O_TRUNC|O_CREAT, 0777);
2 posix_fallocate(fd, 0, sz); /* sz バイトにする */
3 char * a = mmap(0, sz, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
4 /* a[i]に書き込むと{\it filename}のi バイト目に書き込むことになる */
5 for (long i = 0; i < sz; i++) {
6     a[i] = i % 128;
7 }
8 munmap(a, sz);
9 close(fd);
```

mmapでメモリ割り当て

- ▶ sz バイト割り当てる時

```
1 char * a = mmap(0, sz, PROT_READ|PROT_WRITE,  
2                      MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
```

- ▶ 効果は

```
1 char * a = sbrk(sz);
```

と似ているが

1. 保護属性を指定可能 (PROT_xxx)
2. 個別に解放 (munmap) 可能
3. 割り当てるアドレスを第一引数 (mmap(a_0 , ...)) で指定可能

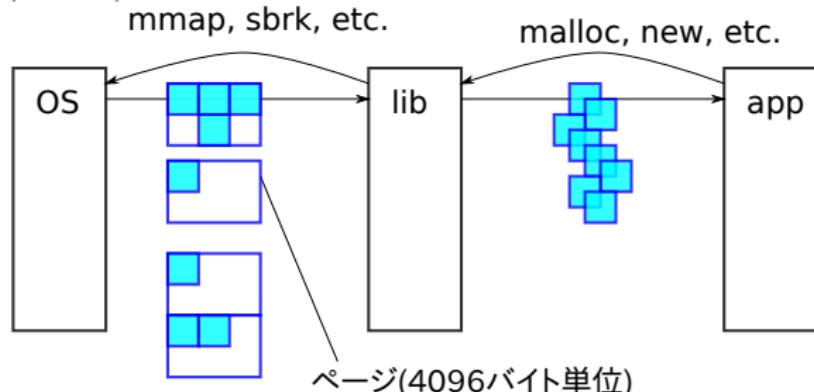
などの利点がある

munmap, mremap

- ▶ `int munmap(a, l) : a から l バイト (ページ単位) 開放`
- ▶ `a' = mremap(a, l, l', flags, a') : a から l バイト割り当てられていた領域を, l' バイトに拡張`

補足: OSのメモリ割当 (mmap, sbrk) vs. malloc, new

- ▶ 比喩:
 - ▶ OS ≈ お菓子工場
 - ▶ 言語処理系・ライブラリ ≈ お菓子屋
 - ▶ アプリ ≈ お菓子を買う子供
- ▶ mmap, sbrk, etc. : ページ (4096 バイトなど) 単位での割当・開放
- ▶ malloc, new, etc. : より細かい単位での割当・開放



ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

mmap の実装、主記憶と 2 次記憶の統合

ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

mmap の実装、主記憶と 2 次記憶の統合

キャッシュ

- ▶ 一度読んだファイル(の一部分)をメモリ(キャッシュ)上に保持
- ▶ 一度書いたファイルもメモリ上に一定期間保持
- ▶ ファイル読み込み時、キャッシュ上にあるデータは2次記憶とのIOを行わずに返す

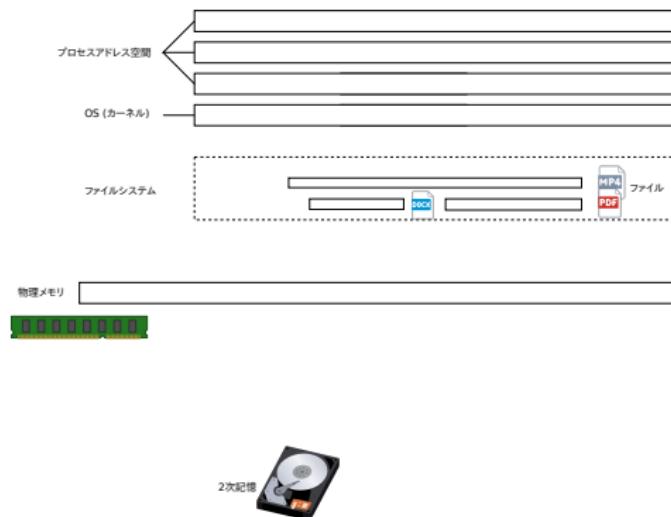
キヤッシュが存在する理由

- ▶ 2次記憶の性能 \leq 主記憶の性能
- ▶ 性能の目安

	遅延	最大転送速度
主記憶	10^{-7}	$10^{10} \sim 10^{11}$ bytes/sec
SSD	10^{-5}	$10^8 \sim 10^9$ bytes/sec
HDD	10^{-2}	$10^7 \sim 10^8$ bytes/sec

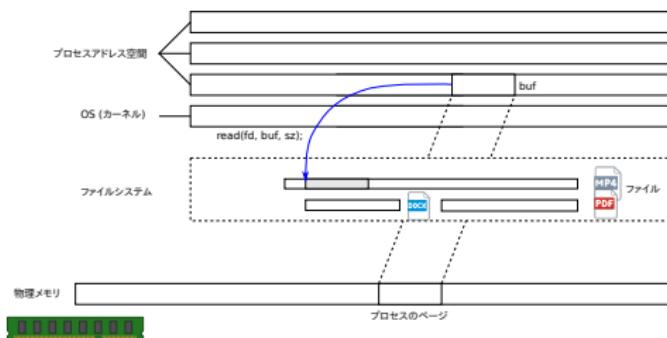
- ▶ 注: 最大転送速度は構成に大きく依存する
 - ▶ HDD も多数並べる (RAID 構成) ことで転送速度は稼げる
 - ▶ 主記憶も同様で、複数のモジュールを束ねて性能を出すのが普通

キヤツシユの動き



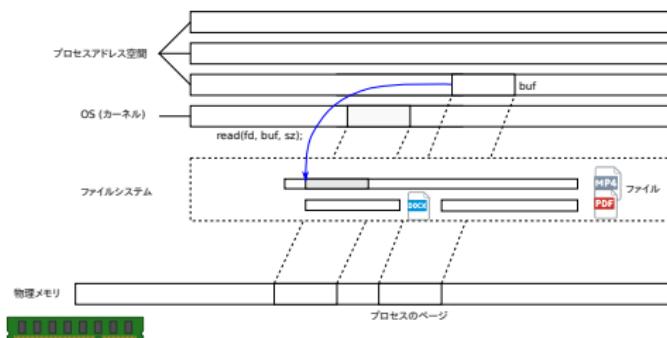
キヤツシユの動き

1. read 要求された部分がキヤツシユ上にない →



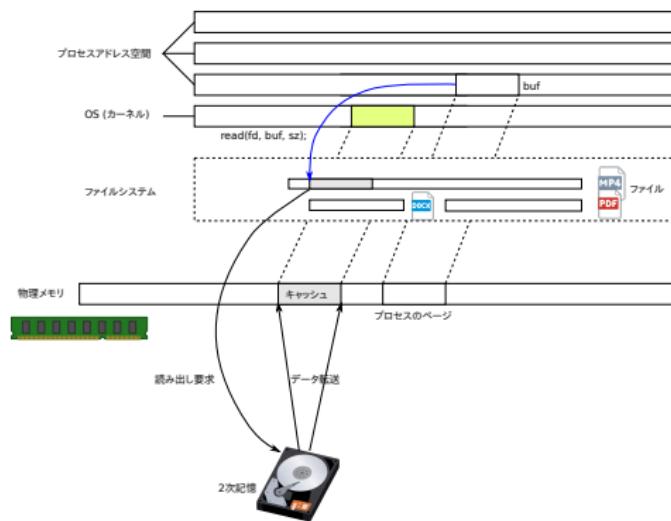
キヤツシユの動き

1. read 要求された部分がキヤツシユ上にない →
 - 1.1 カーネル内にキヤツシユのための領域を割り当てる



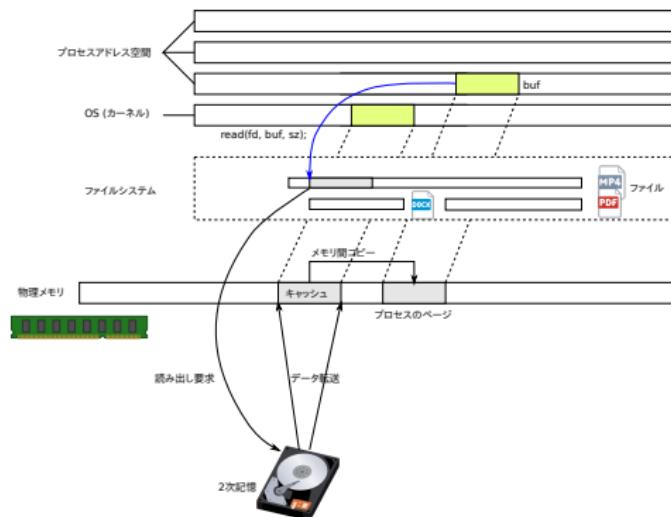
キャッシュの動き

1. read 要求された部分がキャッシュ上にない →
 - 1.1 カーネル内にキャッシュのための領域を割り当てる
 - 1.2 2次記憶からキャッシュ上に読み込むとともに



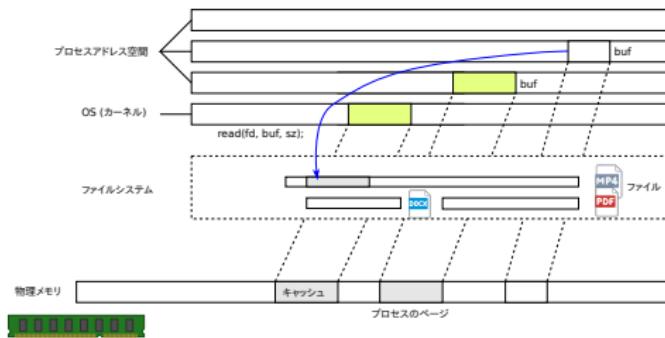
キャッシュの動き

1. read 要求された部分がキャッシュ上にない →
 - 1.1 カーネル内にキャッシュのための領域を割り当てる
 - 1.2 2次記憶からキャッシュ上に読み込むとともに
 - 1.3 プロセス指定のメモリにもコピー



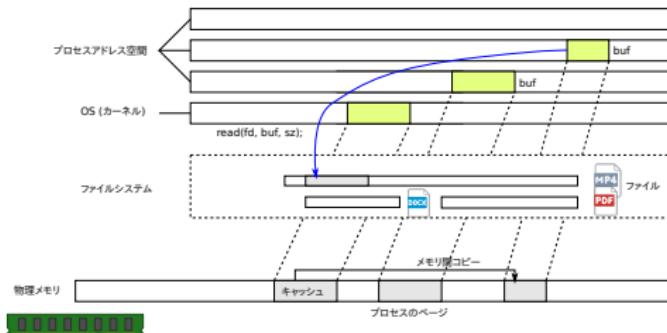
キャッシュの動き

1. read 要求された部分がキャッシュ上にない →
 - 1.1 カーネル内にキャッシュのための領域を割り当てる
 - 1.2 2次記憶からキャッシュ上に読み込むとともに
 - 1.3 プロセス指定のメモリにもコピー
2. read 要求された部分がキャッシュ上にある →



キャッシュの動き

1. read 要求された部分がキャッシュ上にない →
 - 1.1 カーネル内にキャッシュのための領域を割り当てる
 - 1.2 2次記憶からキャッシュ上に読み込むとともに
 - 1.3 プロセス指定のメモリにもコピー
2. read 要求された部分がキャッシュ上にある →
 - 2.1 キャッシュからプロセス指定のメモリにコピー (高速)



ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

mmap の実装、主記憶と 2 次記憶の統合

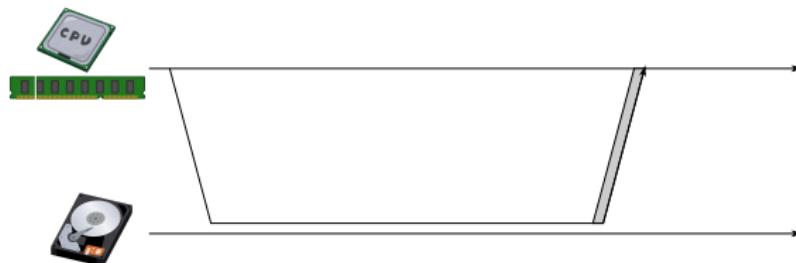
先読みとは

- ▶ プロセスが要求していない部分を先に(キャッシュ内に)読んでおくこと
- ▶ 実際問題としては、ファイルの読み出しが連続した部分に何回か行われた時点で発動

先読みの理由

- ▶ 2次記憶の遅延が大
- ▶ 性能の目安再掲

	遅延	最大転送速度
主記憶	10^{-7} sec	$10^{10} \sim 10^{11}$ bytes/sec
SSD	10^{-5} sec	$10^8 \sim 10^9$ bytes/sec
HDD	10^{-2} sec	$10^7 \sim 10^8$ bytes/sec

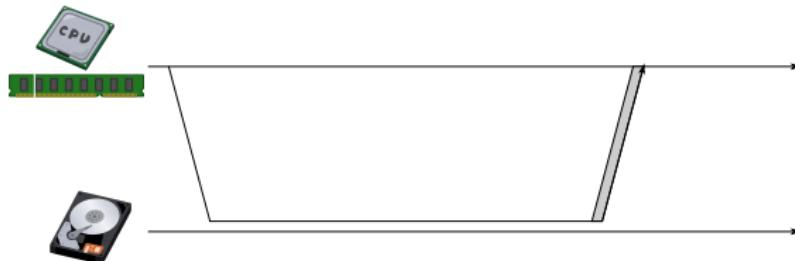


先読みの理由

- ▶ 2次記憶の遅延が大
- ▶ 性能の目安再掲

	遅延	最大転送速度
主記憶	10^{-7} sec	$10^{10} \sim 10^{11}$ bytes/sec
SSD	10^{-5} sec	$10^8 \sim 10^9$ bytes/sec
HDD	10^{-2} sec	$10^7 \sim 10^8$ bytes/sec

- ▶ 小さなデータも「遅延」はほとんど同じ

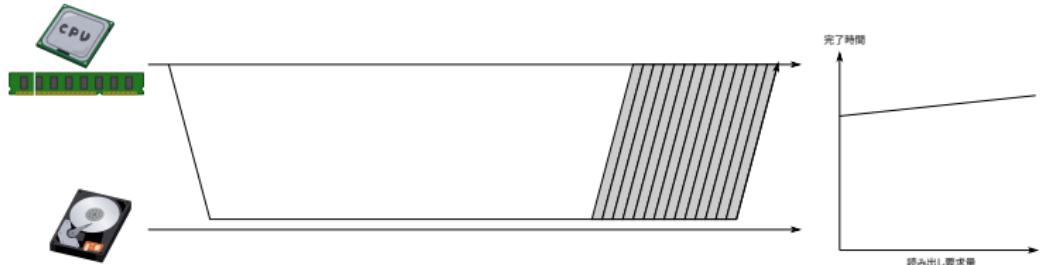


先読みの理由

- ▶ 2次記憶の遅延が大
- ▶ 性能の目安再掲

	遅延	最大転送速度
主記憶	10^{-7} sec	$10^{10} \sim 10^{11}$ bytes/sec
SSD	10^{-5} sec	$10^8 \sim 10^9$ bytes/sec
HDD	10^{-2} sec	$10^7 \sim 10^8$ bytes/sec

- ▶ 小さなデータも「遅延」はほとんど同じ



先読みの理由

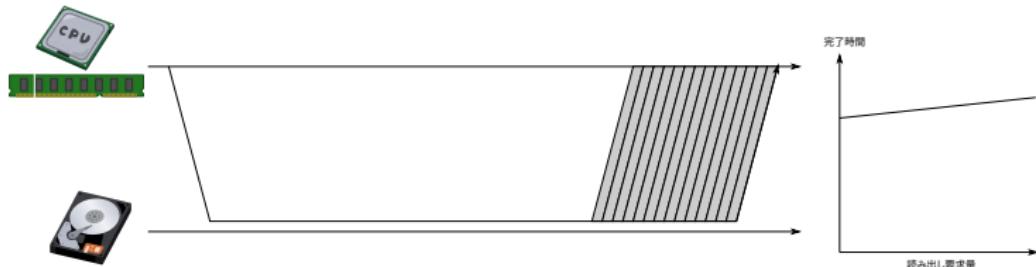
- ▶ 2次記憶の遅延が大
- ▶ 性能の目安再掲

	遅延	最大転送速度
主記憶	10^{-7} sec	$10^{10} \sim 10^{11}$ bytes/sec
SSD	10^{-5} sec	$10^8 \sim 10^9$ bytes/sec
HDD	10^{-2} sec	$10^7 \sim 10^8$ bytes/sec

- ▶ 小さなデータも「遅延」はほとんど同じ
- ▶ e.g., 10^{-2} sec の遅延で 100MB/sec の転送速度は

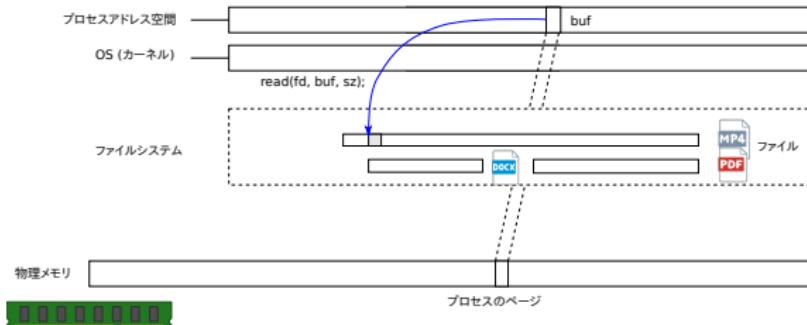
$$100\text{MB/sec} \times 10^{-2} = 1\text{MB/sec}$$

以上の読み出しの場合のみ達成



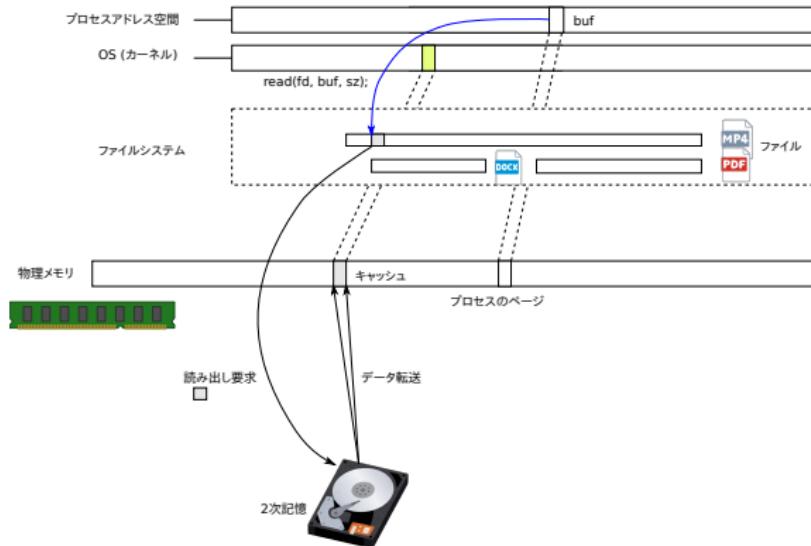
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



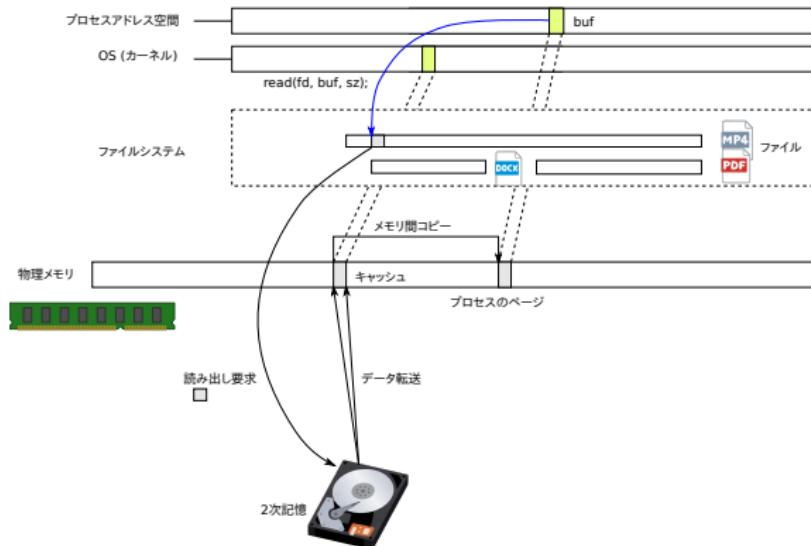
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



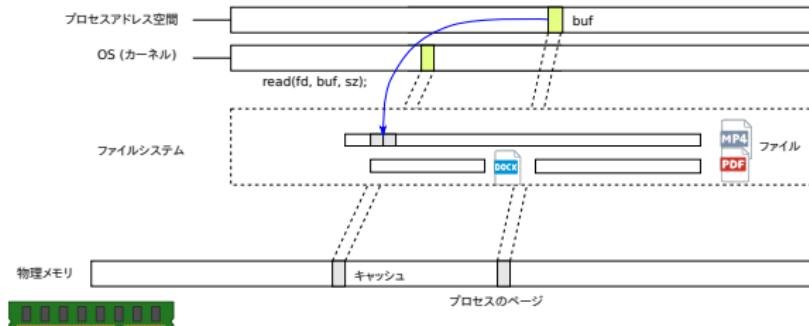
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



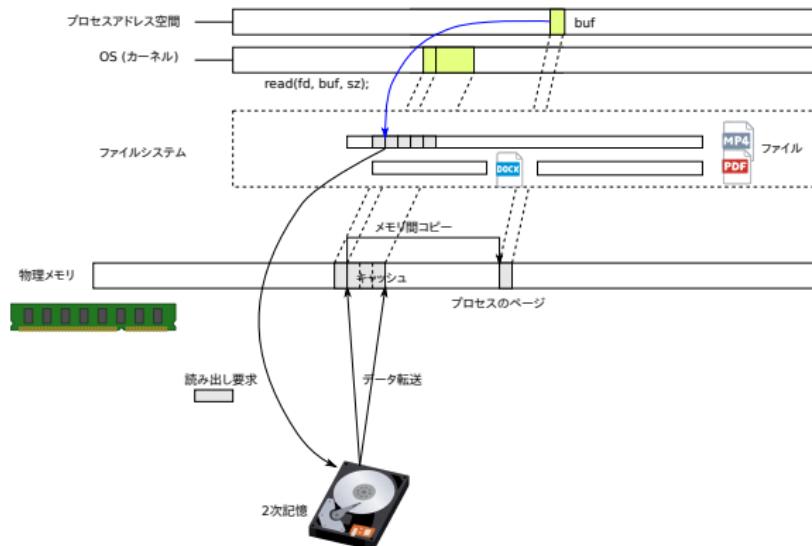
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



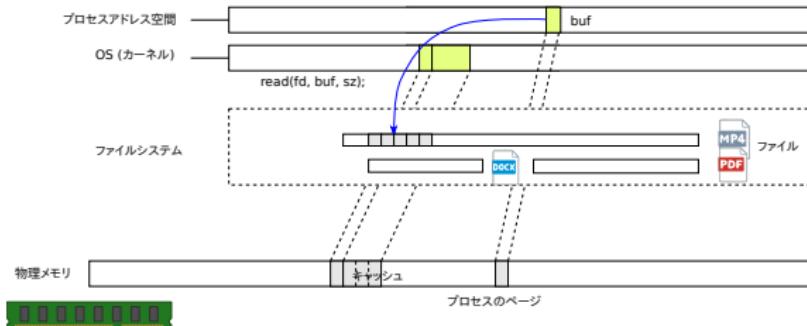
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



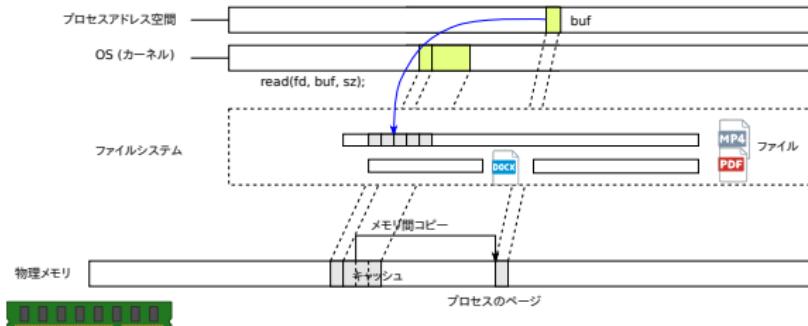
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



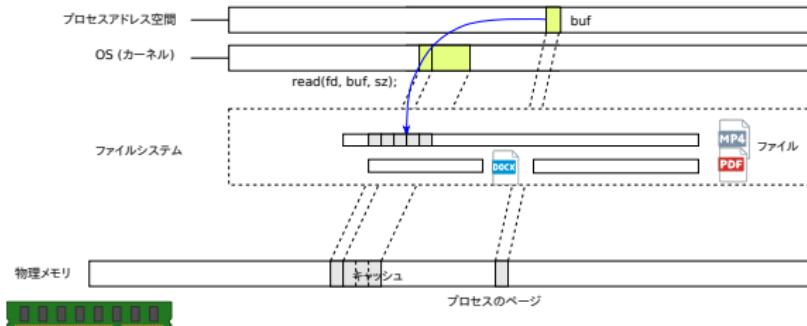
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



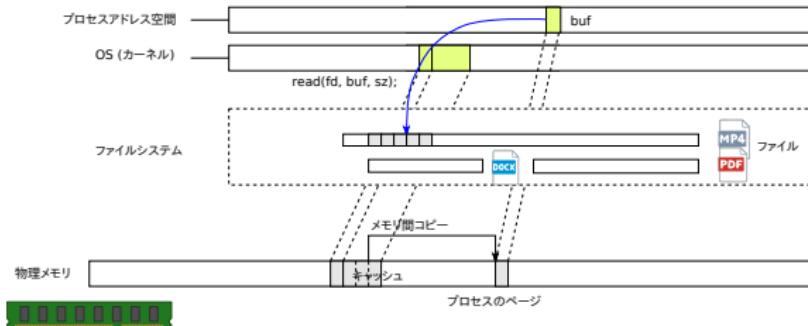
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



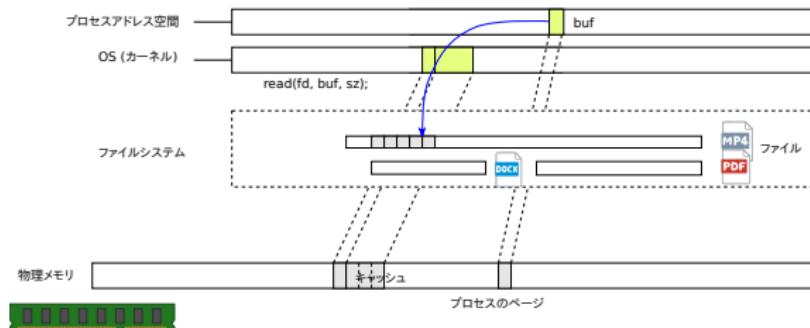
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



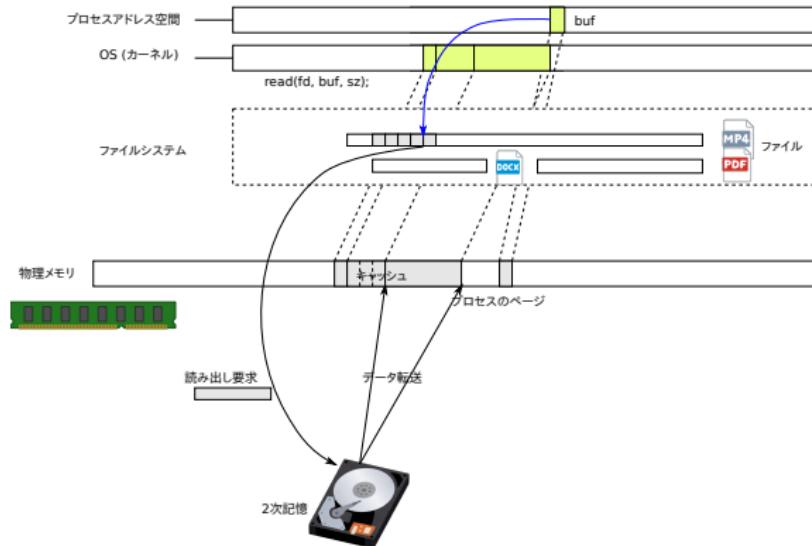
先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



先読みの動作

- ▶ 始めはプロセスが (read で) 要求したサイズをそのまま 2 次記憶装置に要求
- ▶ 逐次的な読み出し (連続したオフセット) を検出すると、大きな読み出しを 2 次記憶装置に発行



ファイルシステムの役割

ファイルシステムの API

ファイルシステム高速化のための OS の機構

キヤツシユ

先読み

mmap の実装、主記憶と 2 次記憶の統合

復習: mmap の効果

▶ 動作

```
1 char * a = mmap(0, sz, proto, flags, fd, offs);
```

で、アドレス範囲 $[a, a + sz)$ が、ファイルの範囲 $[offs, offs + sz)$ に「対応（マップ）」される

▶ つまり

1. $a[i]$ を読むとファイルの $(offs + i)$ バイト目が読み出せる
2. $flags = \text{MAP_SHARED} | \dots$ ならば
 - 2.1 $a[i]$ に書き込むと、ファイルの $(offs + i)$ バイト目に書き込める
 - 2.2 複数のプロセスが同じファイルの同じ場所をマップした場合、それらが書いた結果も共有される（[プロセス間共有メモリ](#)）

mmap の実装 (...はこうちやないよ)

- ▶ 1. だけであれば、以下と「意味的には」同じこと（説明の簡潔さのため read が sz バイト未満でリターンした場合の処理は省略）

```
1 char * a = mmap(0, sz, proto, flags, fd, offs);
```

≈

```
1 char * a = malloc(sz);
2 read(fd, a, sz);
```

- ▶ 2. (MAP_SHARED の動作) はそれでは済まない
- ▶ 1. を含め、mmap は OS の仮想記憶管理の仕組みの拡張として以下のように（エレガントに）実装されている

mmap の実装 概要

▶ mmap呼び出し時

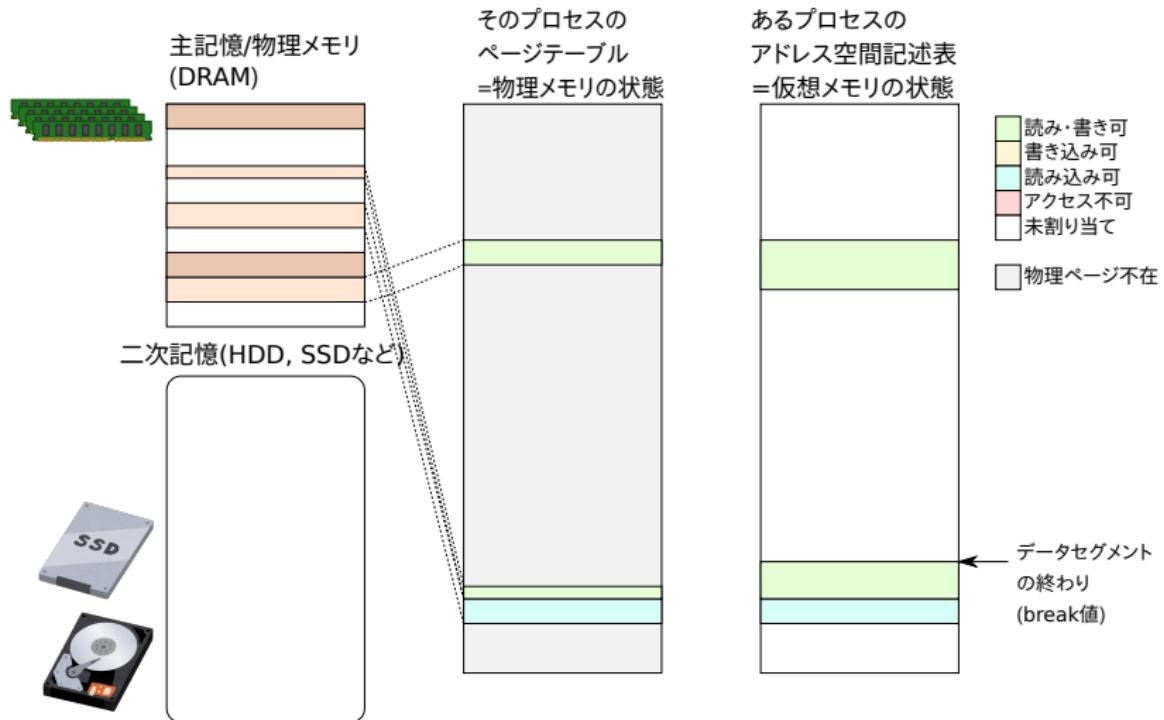
```
1 char * a = mmap(0, sz, proto, flags, fd, offs);
```

1. $[a, a + sz)$ を論理的に割当てる (アドレス空間記述表に記述)
2. ファイルとの対応関係も記録しておく
3. ページテーブル上では $[a, a + sz)$ に対応するページは「不在」としておく (通常の要求時ページングと同じ)

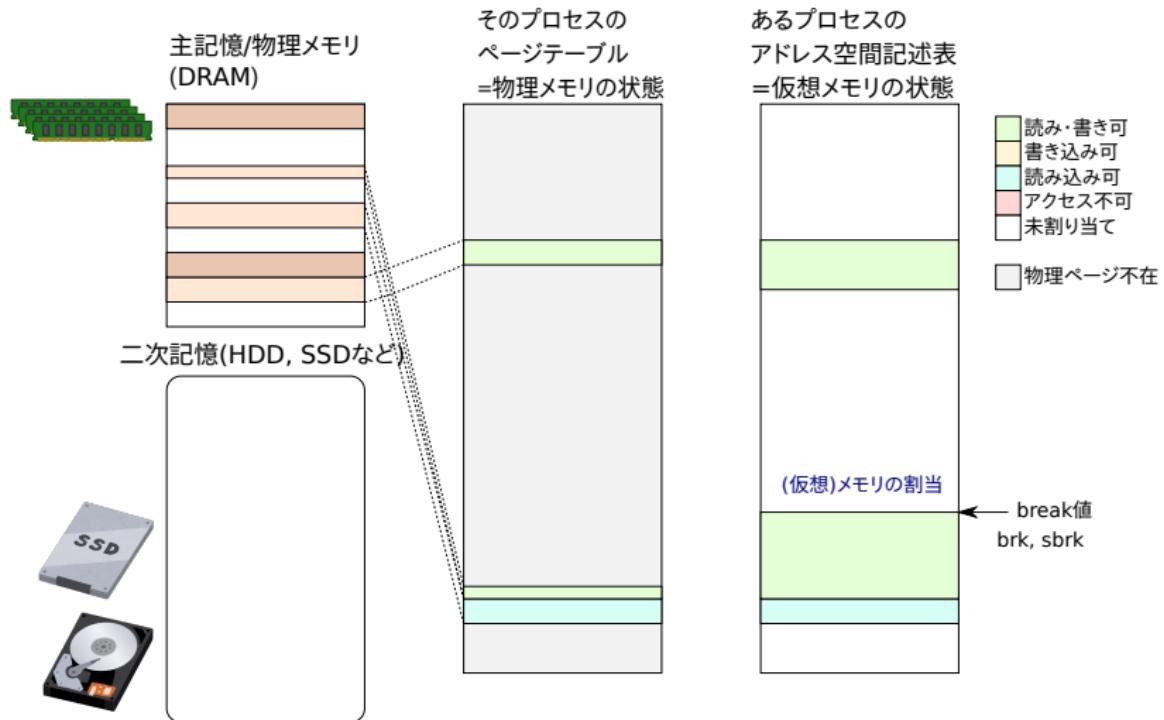
▶ ページ fault 発生時

- ▶ アドレス空間記述表を見て, mmap でファイルに対応した領域であれば、ファイルの対応する領域を読み込む

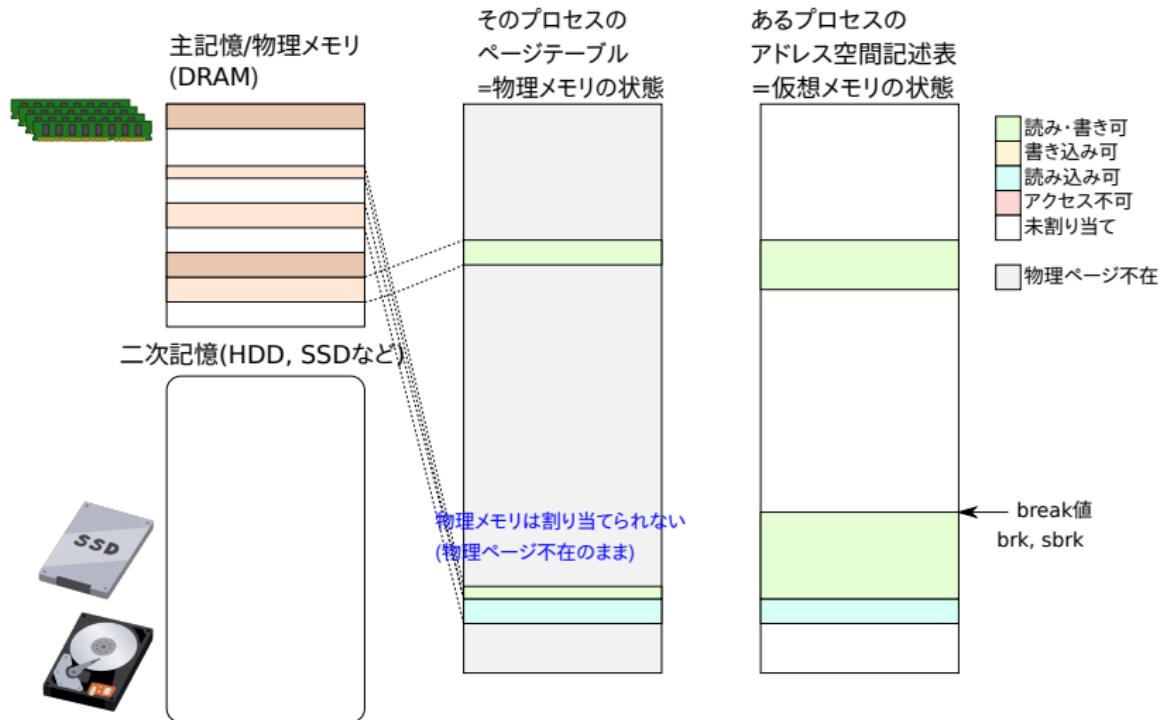
復習: mmap以前のメモリ管理動作図



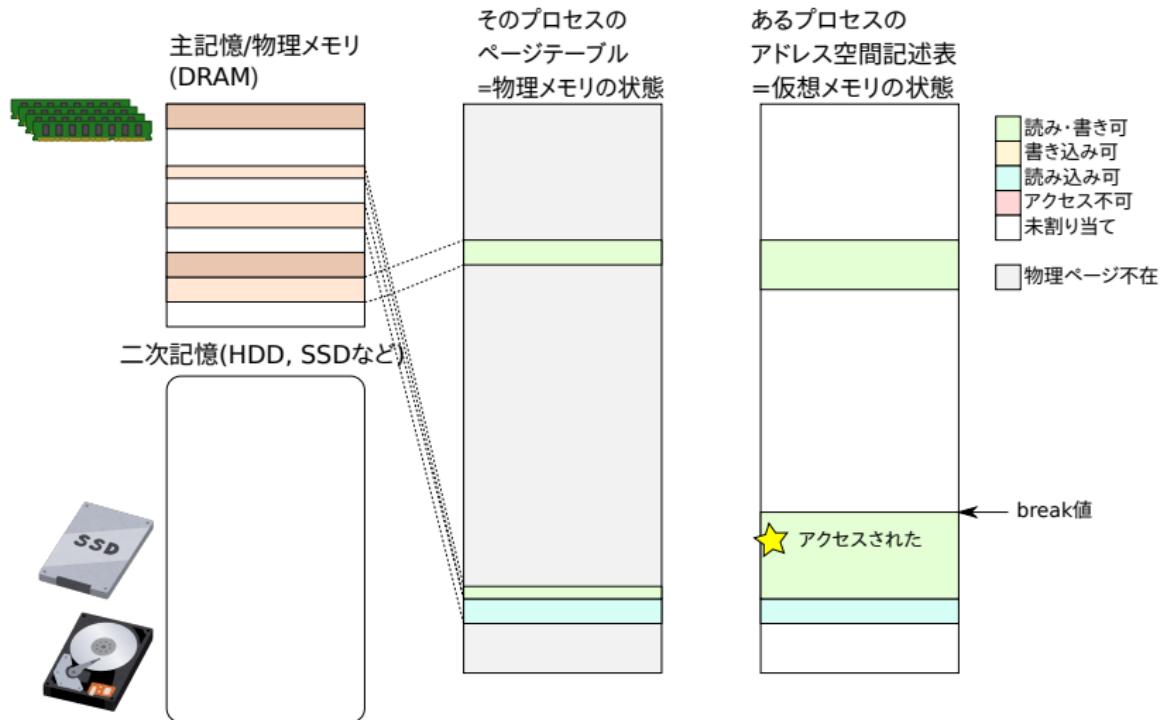
復習: mmap以前のメモリ管理動作図



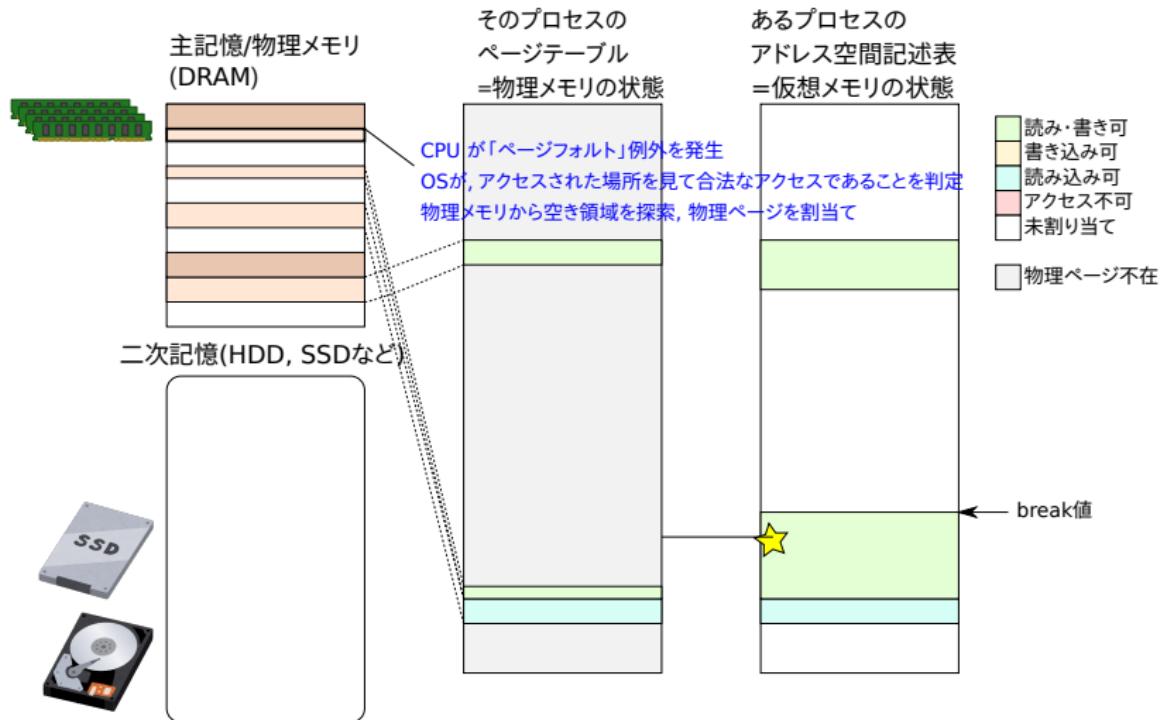
復習: mmap以前のメモリ管理動作図



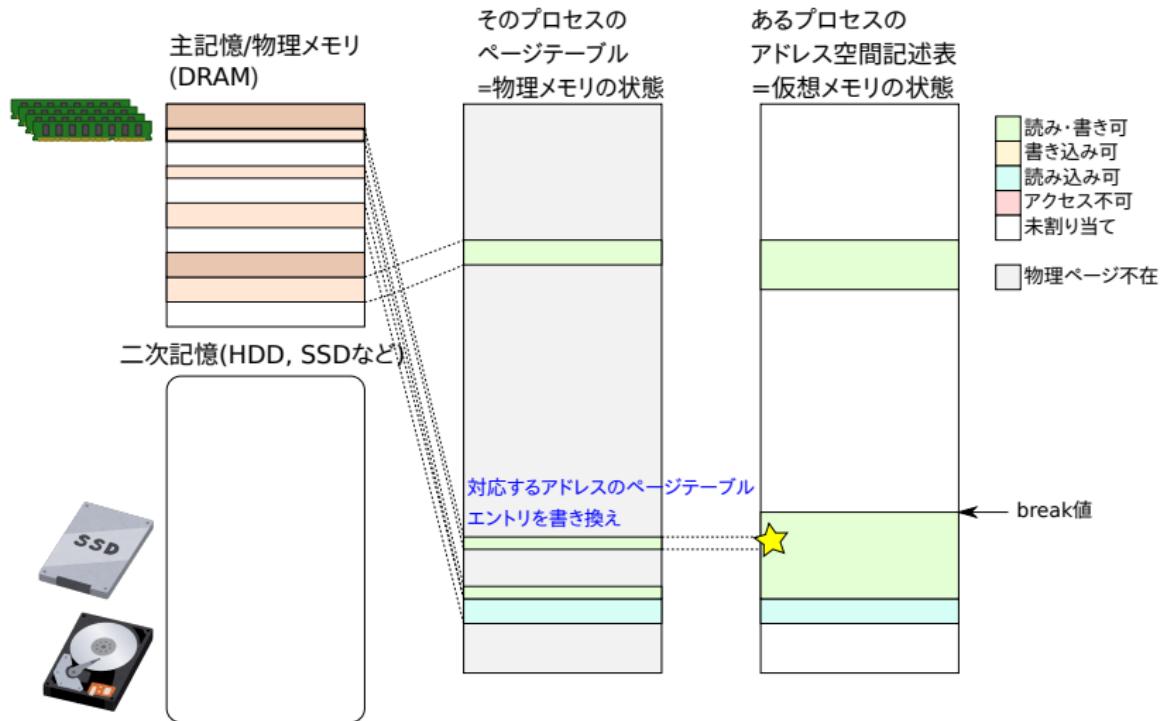
復習: mmap以前のメモリ管理動作図



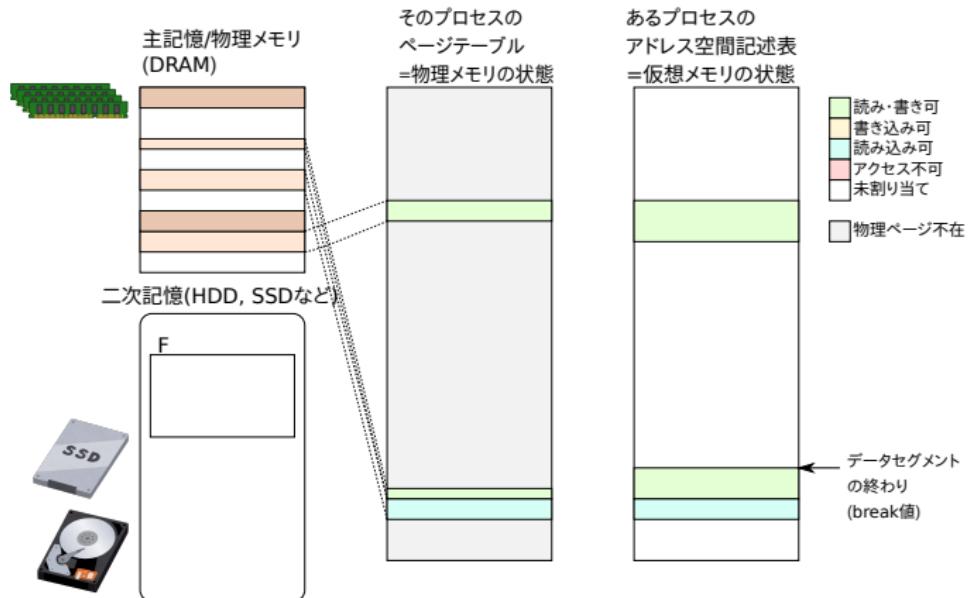
復習: mmap以前のメモリ管理動作図



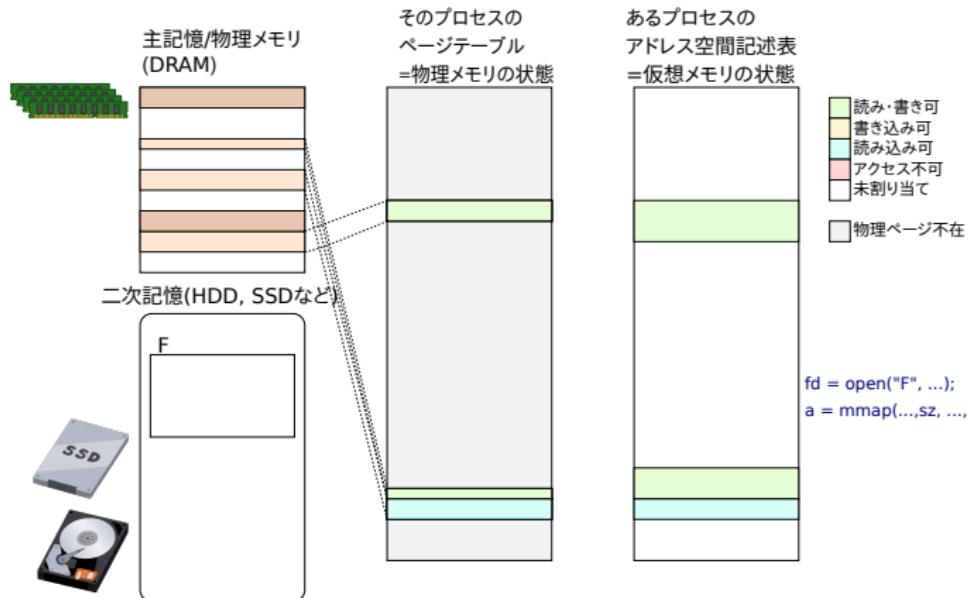
復習: mmap以前のメモリ管理動作図



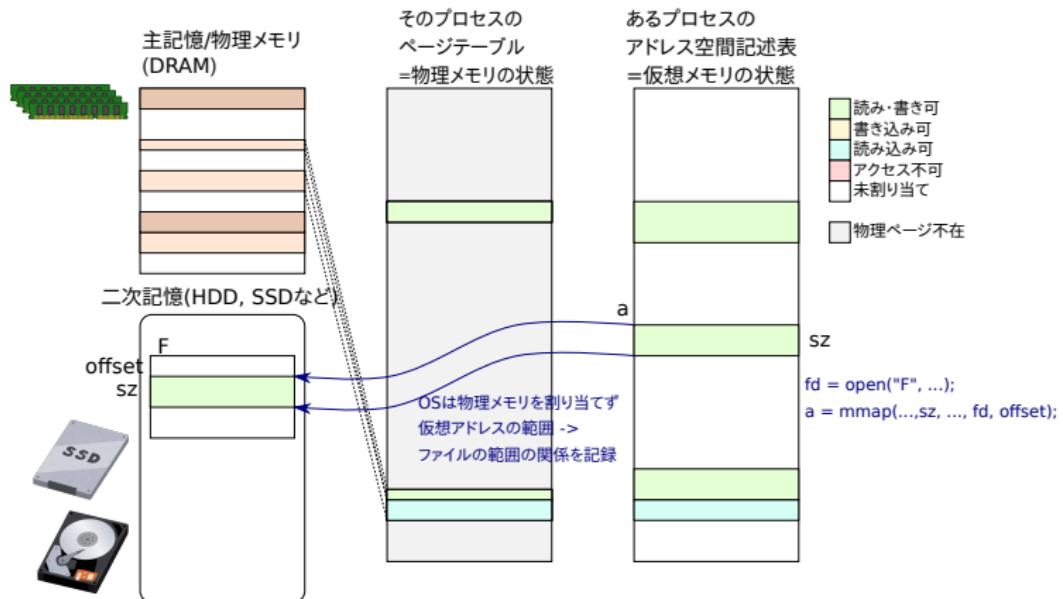
mmap動作図



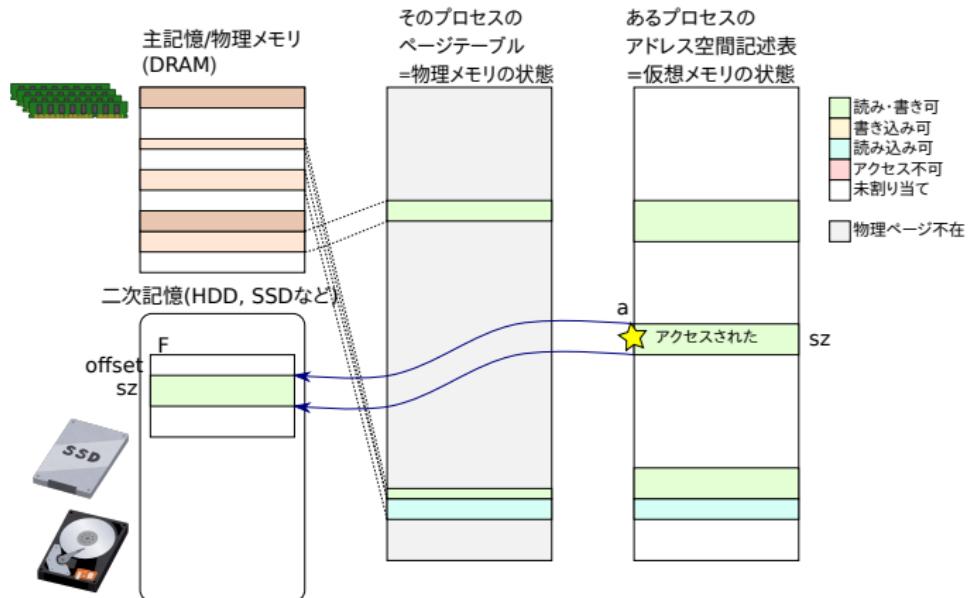
mmap動作図



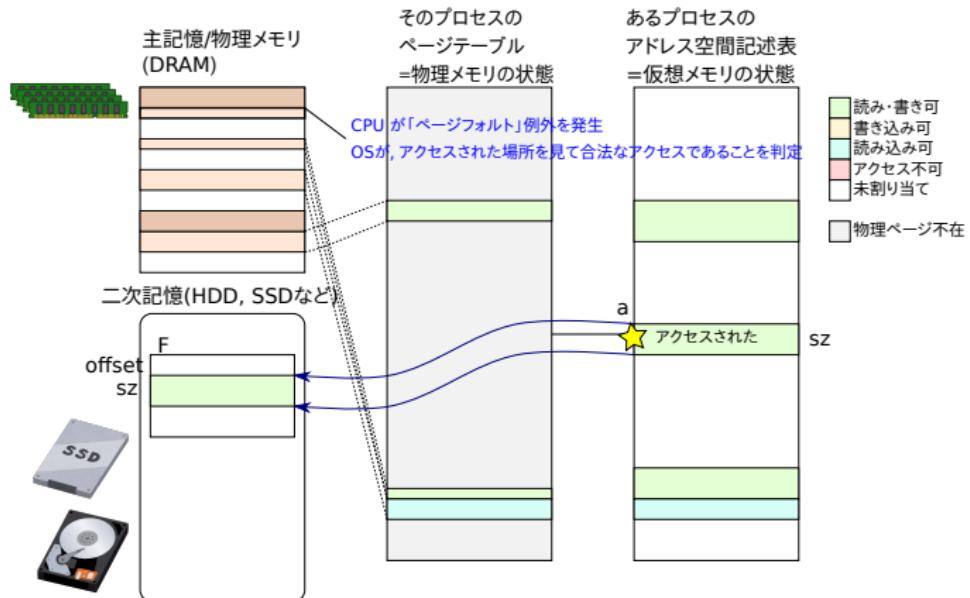
mmap動作図



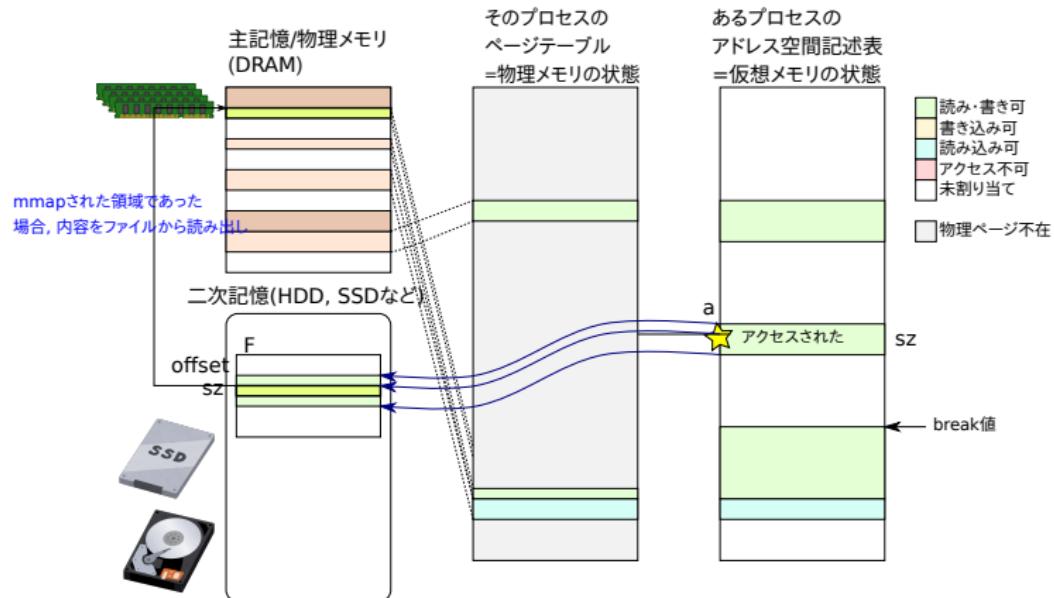
mmap動作図



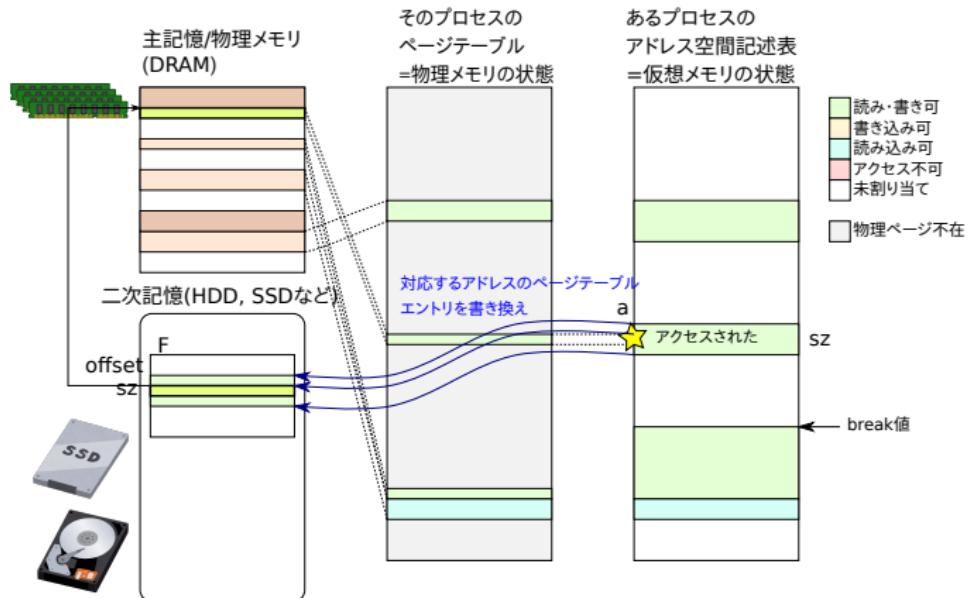
mmap動作図



mmap動作図



mmap動作図



要するに

- ▶ mmap がなくても OS は以下のことをやっていた
 - ▶ プロセスに割り当てた領域でも、実際に物理ページが対応しているとは限らない
 - ▶ 物理ページが対応していないページへのアクセス → ページフォルト →
 - ▶ 物理メモリを割り当てる
 - ▶ 中身は、初めてであれば 0 で埋める、そうでなければ 2 次記憶（ページング領域、スワップ領域）から読み込む
- ▶ mmap は、ページング領域として任意のファイルを指定できるようにしたに過ぎない



OS内のページフォルト処理 (mmapなし)

```
1 handle_page_fault(void * a, access_type rw) {  
2     そのプロセスのアドレス空間記述表を見る;  
3     if (aへのアクセス rw は非合法) segmentation fault 発生;  
4     p = 空いている物理ページ;  
5  
6  
7     if (aへのアクセスは初めて) {  
8         p を 0 で埋める;  
9     } else {  
10        aの中身を ページング領域から読み込み, p をそれで埋める;  
11    }  
12    仮想アドレスa → p を対応(ページテーブル書き換え);  
13 }
```



OS内のページフォルト処理 (mmapあり)

```
1 handle_page_fault(void * a, access_type rw) {  
2     そのプロセスのアドレス空間記述表を見る;  
3     if (aへのアクセス rw は非合法) segmentation fault 発生;  
4     p = 空いている物理ページ;  
5     if (a は mmap でファイルと関連付けられた領域) {  
6         a の中身を 関連付けられたファイル から読み込み, p をそれで埋める;  
7     } else if (aへのアクセスは初めて) {  
8         p を 0 で埋める;  
9     } else {  
10        a の中身を ページング領域 から読み込み, p をそれで埋める;  
11    }  
12    仮想アドレスa → p を対応(ページテーブル書き換え);  
13 }
```



共有マッピングとプライベートマッピング

▶ 復習

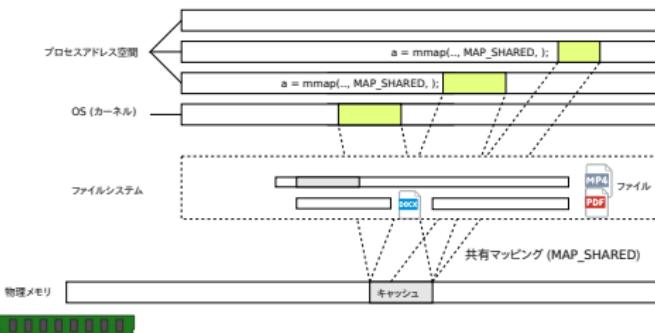
```
1 char * a = mmap(0, sz, proto, flags, fd, offs);
```

- ▶ 共有マッピング (*flags* = MAP_SHARED | ...)
 - ▶ 書き込みがファイルに反映される
 - ▶ 複数プロセス間でも書き込みが共有される
- ▶ プライベートマッピング (*flags* = MAP_PRIVATE | ...)
 - ▶ 書き込みはファイルに反映されない
 - ▶ 複数プロセス間でも書き込みは共有されない
- ▶ 違いは, 物理メモリをどう使うかの違いで, もたらされる

mmap の物理メモリ利用

▶ MAP_SHARED

- ▶ 同じ場所(ファイル + オフセット)に対しては、同じ物理アドレス(物理ページ)を用いる
- ▶ 実はキャッシュと同じ物理ページ



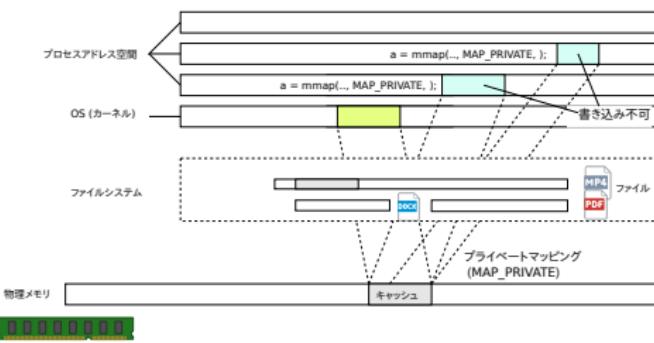
mmap の物理メモリ利用

▶ MAP_SHARED

- ▶ 同じ場所(ファイル + オフセット)に対しては、同じ物理アドレス(物理ページ)を用いる
- ▶ 実はキャッシュと同じ物理ページ

▶ MAP_PRIVATE

- ▶ (一般には) 同じ場所に対しても、異なる物理アドレス(ページ)を用いる必要がある
- ▶ しかし、「書き込みが実際に起きるまでは」同じ物理ページを用いる(コピーオンライト copy-on-write)



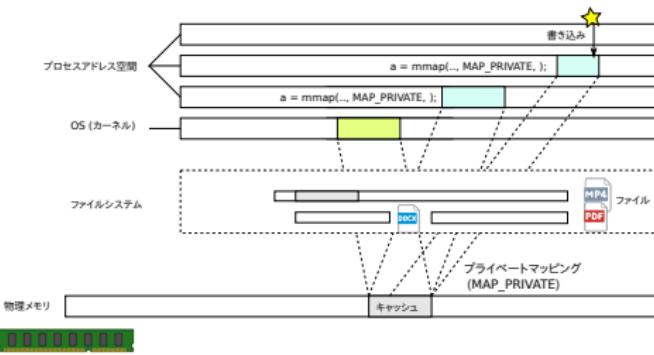
mmap の物理メモリ利用

▶ MAP_SHARED

- ▶ 同じ場所(ファイル + オフセット)に対しては、同じ物理アドレス(物理ページ)を用いる
- ▶ 実はキャッシュと同じ物理ページ

▶ MAP_PRIVATE

- ▶ (一般には) 同じ場所に対しても、異なる物理アドレス(ページ)を用いる必要がある
- ▶ しかし、「書き込みが実際に起きるまでは」同じ物理ページを用いる(コピーオンライト copy-on-write)



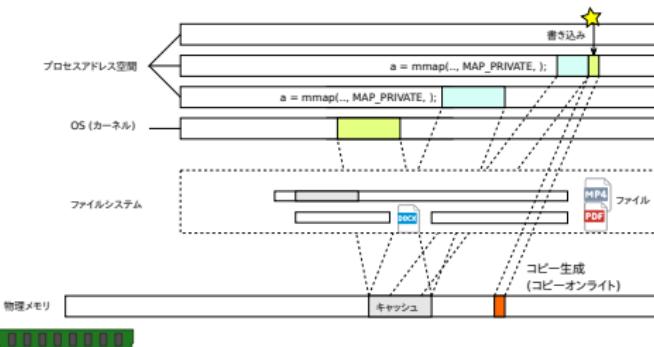
mmap の物理メモリ利用

▶ MAP_SHARED

- ▶ 同じ場所(ファイル + オフセット)に対しては、同じ物理アドレス(物理ページ)を用いる
- ▶ 実はキャッシュと同じ物理ページ

▶ MAP_PRIVATE

- ▶ (一般には) 同じ場所に対しても、異なる物理アドレス(ページ)を用いる必要がある
- ▶ しかし、「書き込みが実際に起きるまでは」同じ物理ページを用いる(コピーオンライト copy-on-write)



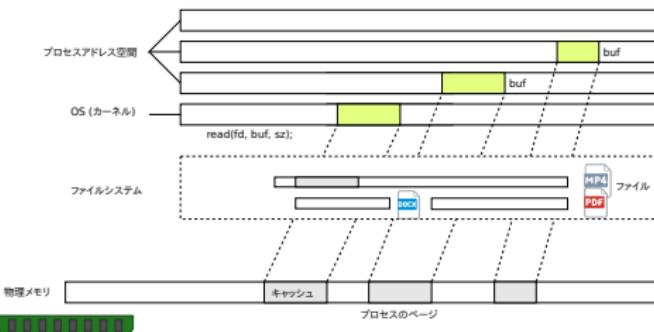
mmap の物理メモリ利用

▶ MAP_SHARED

- ▶ 同じ場所(ファイル + オフセット)に対しては、同じ物理アドレス(物理ページ)を用いる
- ▶ 実はキャッシュと同じ物理ページ

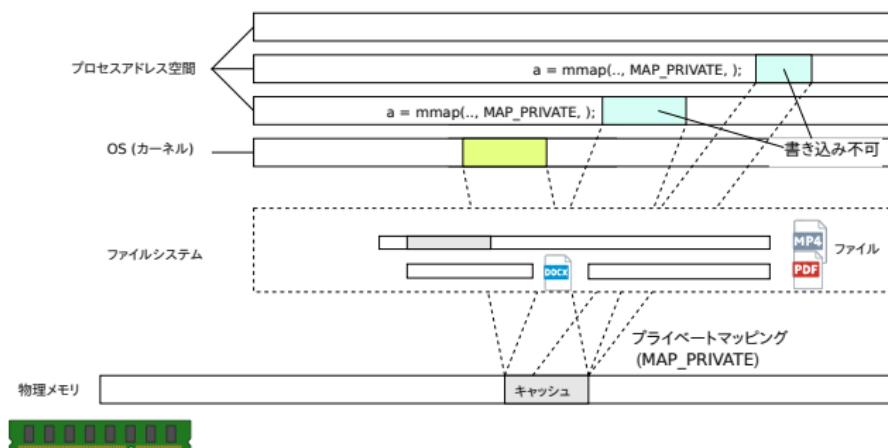
▶ MAP_PRIVATE

- ▶ (一般には) 同じ場所に対しても、異なる物理アドレス(ページ)を用いる必要がある
 - ▶ しかし、「書き込みが実際に起きるまでは」同じ物理ページを用いる([コピーオンライト copy-on-write](#))
- ▶ 復習: read(各プロセスが異なる物理ページを用いる)



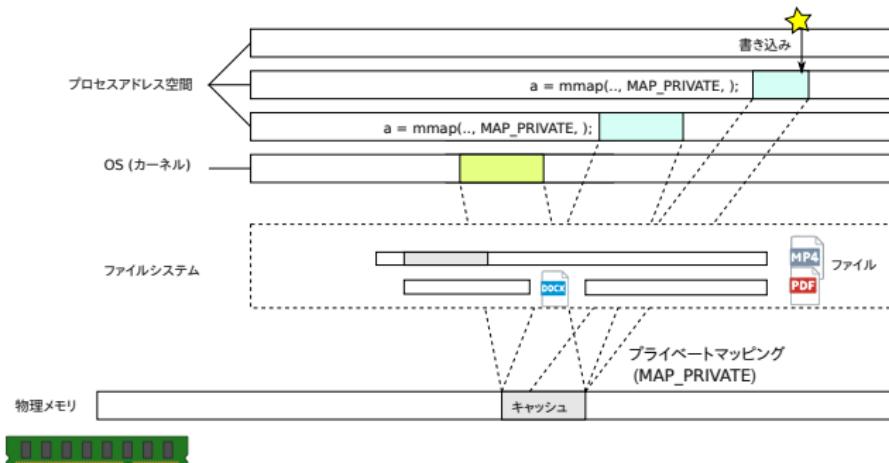
kopioオンライン

- ▶ 「変更されるまでは物理メモリを共有する」方式の呼称
- ▶ 物理ページを共有しつつ、MMUの機能で「書き込み不可」属性をつけておく



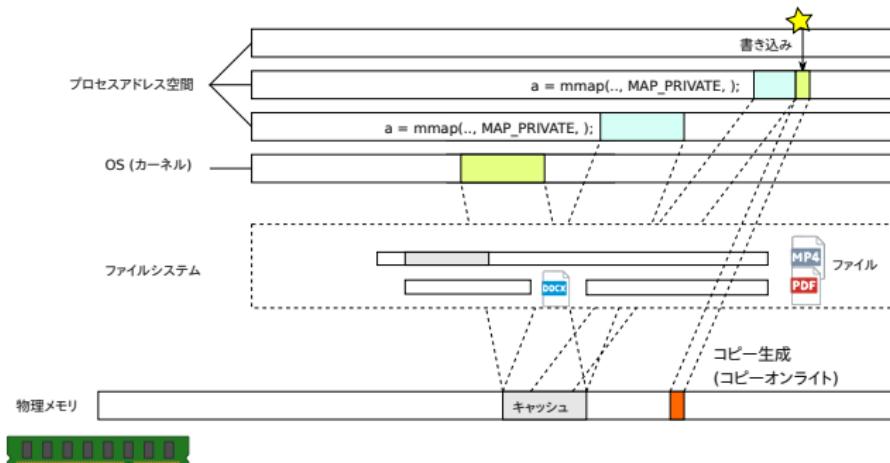
kopirionaito

- ▶ 「変更されるまでは物理メモリを共有する」方式の呼称
- ▶ 物理ページを共有しつつ、MMUの機能で「書き込み不可」属性をつけておく
- ▶ 書き込み時に保護例外が発生 → OSが対応する物理メモリをコピーし、「書き込まれたページ → 新しい物理ページ」にマッピングを変更



kopirionaito

- ▶ 「変更されるまでは物理メモリを共有する」方式の呼称
- ▶ 物理ページを共有しつつ、MMUの機能で「書き込み不可」属性をつけておく
- ▶ 書き込み時に保護例外が発生 → OSが対応する物理メモリをコピーし、「書き込まれたページ → 新しい物理ページ」にマッピングを変更



mmap が効果的な場面 (1)

- ▶ 大きなファイルのごく一部を飛び飛びにアクセスする
 1. lseek + read (または pread) で明示的に一部だけをアクセスするのは面倒
 2. といってファイルの全てをメモリに読み込むのは無駄
⇒ mmap でファイル全体をマップする
- ▶ 典型例: 大きなデータの索引 (index)
 - ▶ 2 分探索のための整列された配列
 - ▶ 探索木 (2 分探索木, B 木等)
 - ▶ ハッシュ表

mmap が効果的な場面 (2)

- ▶ 多く (P 個) のプロセスが同じファイルをアクセスする
 - ▶ mmap (共有マッピング) → 同一領域に対する物理ページは共有
 - ▶ mmap (プライベートマッピング) → 書き込まれていない領域に対する物理ページは共有
 - ▶ read → 読み込んだプロセスごとに異なる物理ページ

mmap が効果的な場面 (3)

- ▶ プログラム(命令)の読み込み
- ▶ 特に、多くのプログラムが用いる大きなライブラリ(GUIなど)
- ▶ (1), (2) の性質を併せ持つ

プログラム起動の高速化

- ▶ Unix におけるプログラム起動 ≈
 1. fork (アドレス空間の複製)
 2. execv (プログラムの実行)
 - 2.1 指定された実行可能ファイルを読み込み
 - 2.2 共有ライブラリを読み込み
- ▶ 高速化
 - ▶ fork 時に, 物理メモリを親プロセス, 子プロセスで共有 (コピー・オン・ライト)
 - ▶ すぐに execv されれば, 子プロセスのマッピングは解除
 - ▶ 共有ライブラリを mmap で読み込み