FreeBSD 勉強会

ストレージの管理: GEOM, UFS, ZFS

後編

佐藤　広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2012/8/24

講師紹介

佐藤 広生 <hrs@FreeBSD.org>

- *BSD関連のプロジェクトで10年くらい色々やってます
- カーネル開発・ユーザランド開発・文書翻訳・サーバ提供
  などなど
- FreeBSD コアチームメンバ(2006 年から4期目)、
  リリースエンジニア
  (commit 比率は src/ports/doc で 1:1:1くらい)
- AsiaBSDCon 主宰
- 技術的なご相談や講演・執筆依頼は hrs@allbsd.org まで

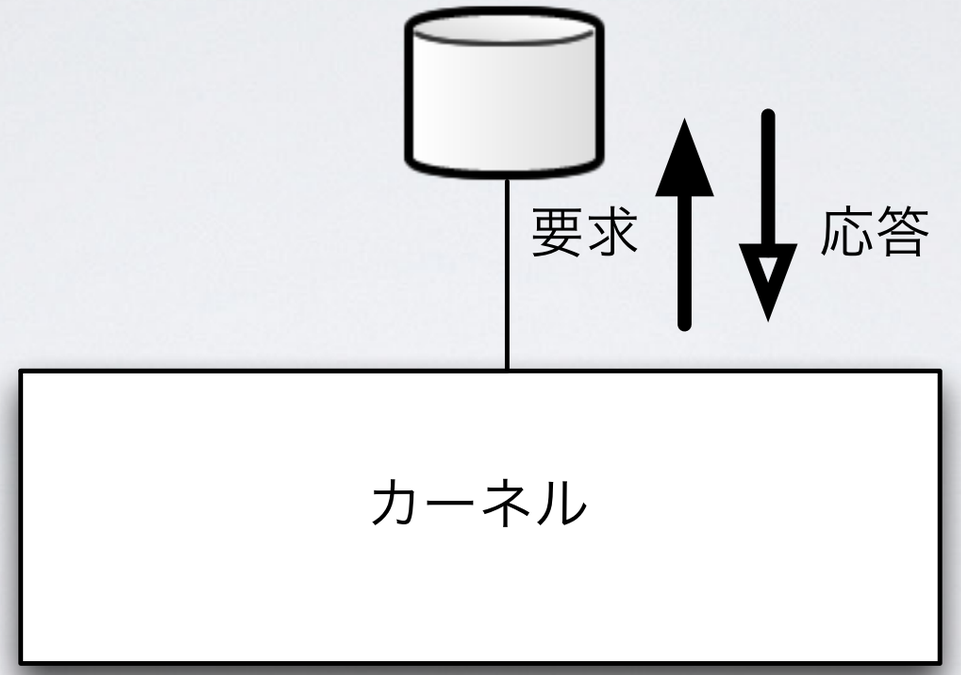
お話すること

- **GEOMとUFS（続き）**
  - 性能を決める要因と性能指標
  - GEOM 種類別活用シナリオ
  - HASTを使ったストレージの冗長化
- **ZFS**
  - 構造とコンセプト
  - 技術詳細と利点/欠点
  - 使い方と構築事例

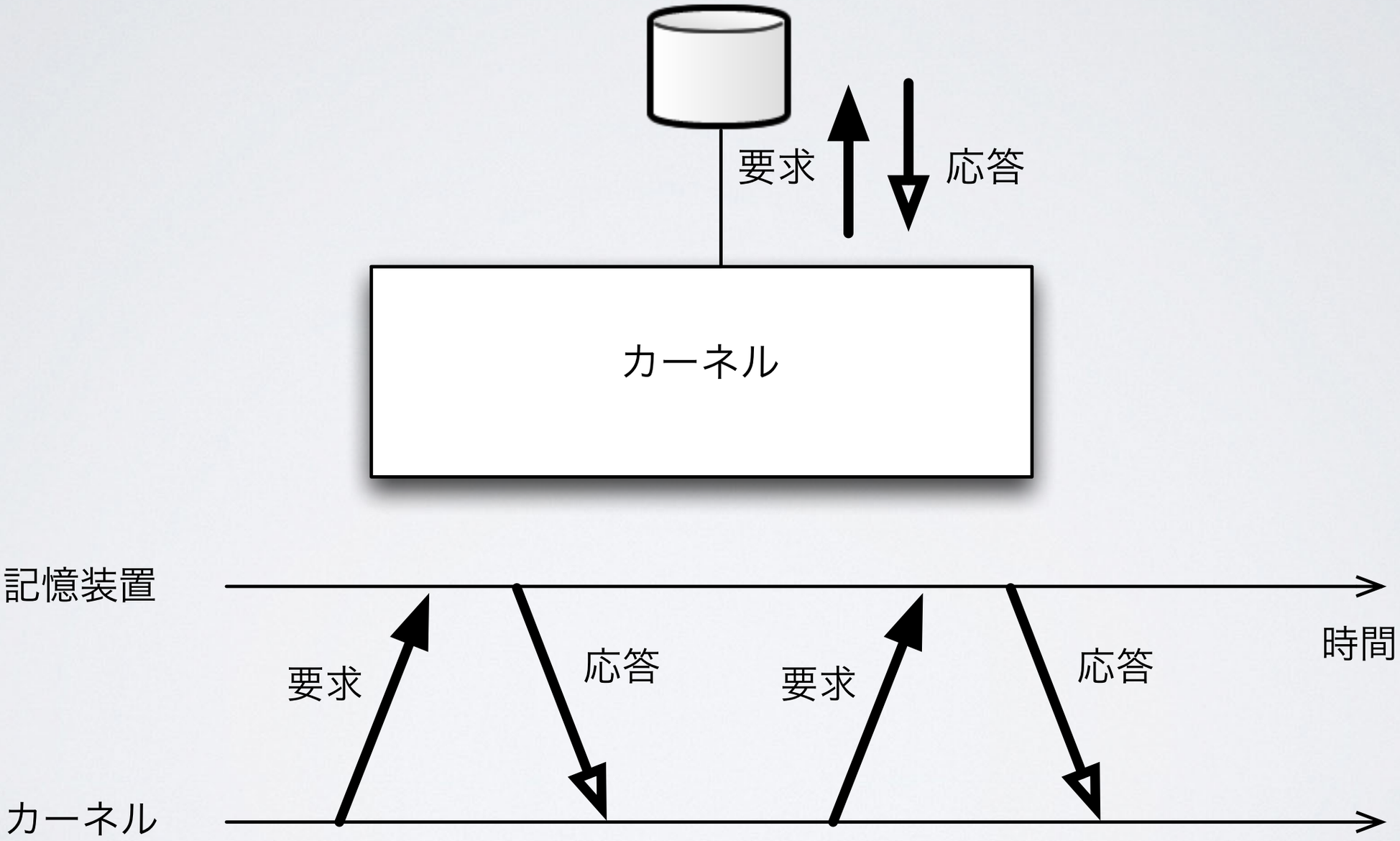
お話すること

- **GEOMとUFS（続き）**
  - 性能を決める要因と性能指標
  - GEOM 種類別活用シナリオ
  - HASTを使ったストレージの冗長化
- **ZFS**
  - 構造とコンセプト
  - 技術詳細と利点/欠点
  - 使い方と構築事例
    ...をやろうとしたのですが、分量が大きいので
    今回はライブデモにします

記憶装置の性能



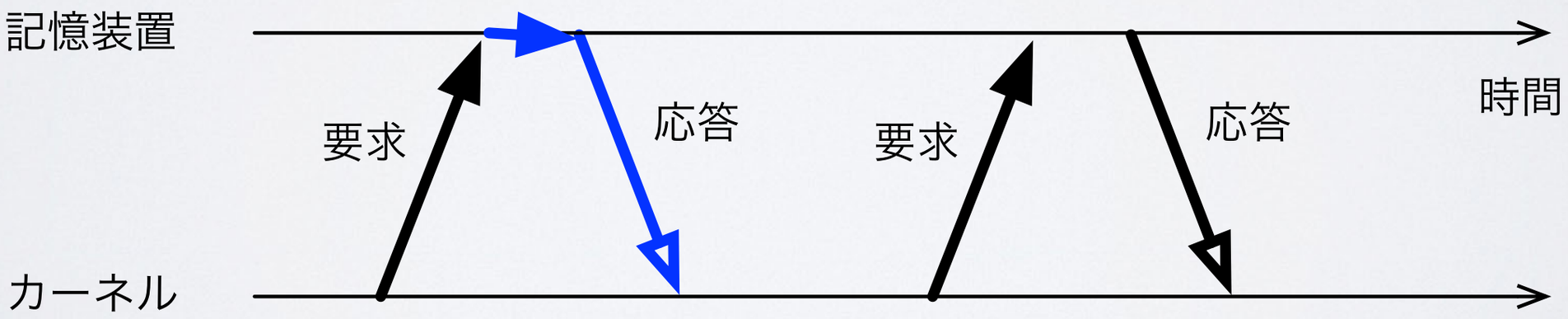
記憶装置の性能



記憶装置の性能

- **読み出し要求**
  - 応答＝記憶装置はデータを用意してカーネルに送り返す
  - かかる時間＝データ待ち時間＋データ転送時間

- **読み出し要求**
  - データの転送がカーネルから発生（逆になる）



記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - USB 2.0 / 8GB の同型のメモリ

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - USB 2.0 / 8GB の同型のメモリ
  - 「USB 3 対応で 60MB/s で使えるぜ」と書いてある

```
da2 at umass-sim0 bus 0 scbus4 target 0 lun 0
da2: <BUFFALO USB Flash Disk 1.00> Removable Direct Access SCSI-6 device
da2: 40.000MB/s transfers
da2: 7368MB (15089664 512 byte sectors: 255H 63S/T 939C)

da3 at umass-sim1 bus 1 scbus5 target 0 lun 0
da3: <BUFFALO USB Flash Disk 1.00> Removable Direct Access SCSI-6 device
da3: 40.000MB/s transfers
da3: 7368MB (15089664 512 byte sectors: 255H 63S/T 939C)
```

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を**カーネル内部で**転送する速度

```
# dd if=/dev/zero of=/dev/null bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 0.009452 secs (8874857381 bytes/sec)
```

- bs=1024x1024: 1MB単位
- count=80: 80 回
- 結果は 8.3 GiB/s（注：マシンによって違う）

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を **da2-カーネル-da3** と転送

```
# dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 9.370823 secs (8951837 bytes/sec)
```

- bs=1024x1024: 1MB単位
- count=80: 80 回
- 結果は 8.5 MiB/s （約1/1000）

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を **da2 からカーネル**に転送

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80
80+0 records in
80+0 records out
??
```

- 「読み出すだけ」にしてみた
- 結果は？

  1) 1-10 MiB/s　2) 10-20 MiB/s
  3) 20-30 MiB/s　4) 30-40 MiB/s

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を **da2 からカーネル**に転送

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 3.399664 secs (24674815 bytes/sec)
```

- 「読み出すだけ」にしてみた
- 結果は 23.5 MiB/s
- 「読む→カーネル→書く」の後ろの部分がないので速いぞ

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を**カーネルから da3** に転送

```
# dd if=/dev/zero of=/dev/da3 bs=1024x1024 count=80
80+0 records in
80+0 records out
??
```

- じゃあ「書き込むだけ」にしてみたら？
- 結果は？

  1) 1-10 MiB/s　2) 10-20 MiB/s
  3) 20-30 MiB/s　4) 30-40 MiB/s

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を**カーネルから da3** に転送

```
# dd if=/dev/zero of=/dev/da3 bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 6.468696 secs (12968004 bytes/sec)
```

- じゃあ「書き込むだけ」にしてみたら？
- 結果は 12.4 MiB/s

記憶装置の性能

- **実験：USB メモリのデータをコピーする時間は？**
  - 80GB のデータ（値は0）を**カーネルから da3** に転送

```
# dd if=/dev/zero of=/dev/da3 bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 6.463656 secs (12978001 bytes/sec)
```

read:23.5MiB/s
write:12.5MiB/s
の組み合わせが 8MiB/s になった

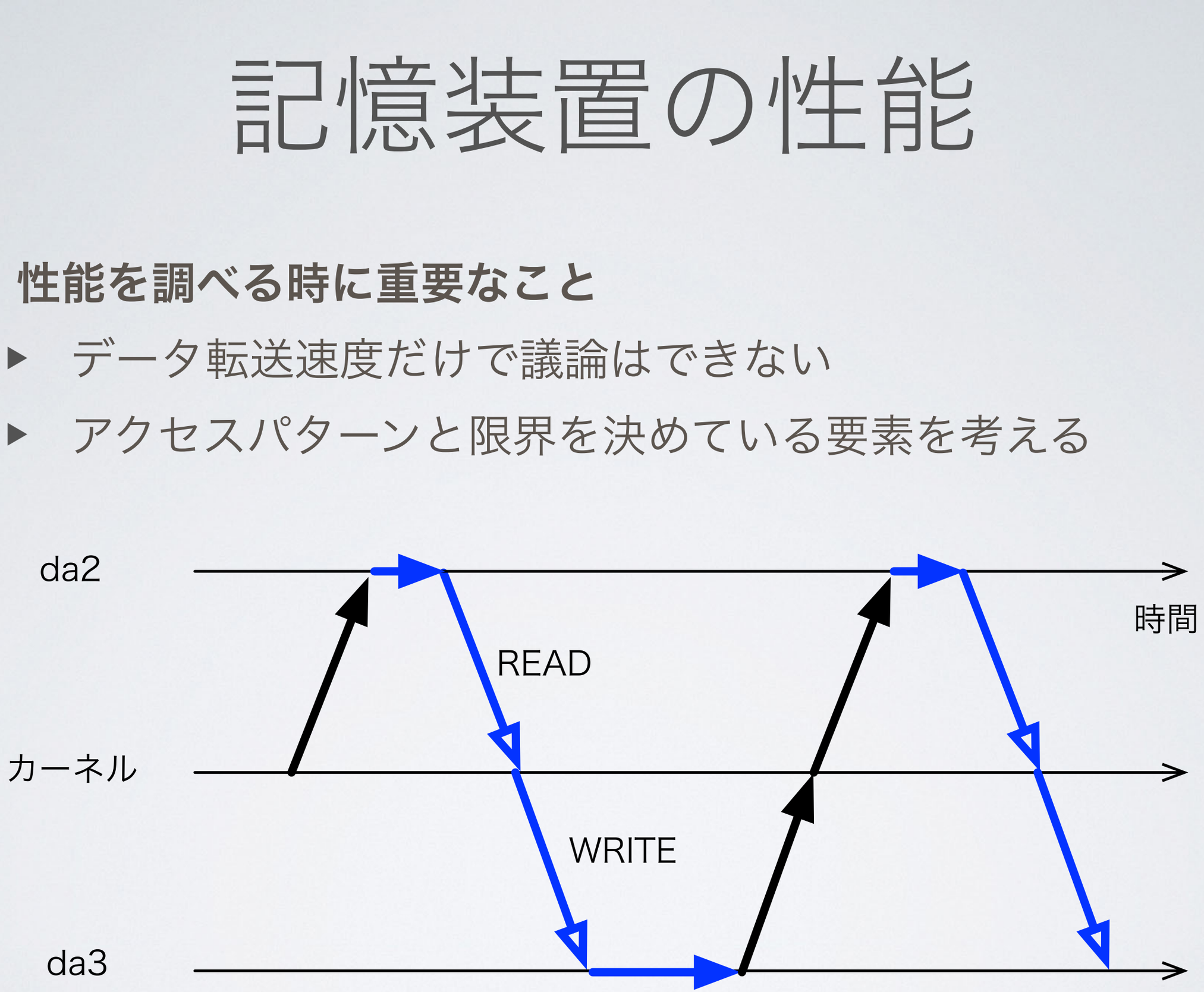
- じゃあ「書き込むだけ」にしてみたら？
- 結果は 12.4 MiB/s

記憶装置の性能

- **性能を調べる時に重要なこと**
  - データ転送速度だけで議論はできない
  - アクセスパターンと限界を決めている要素を考える

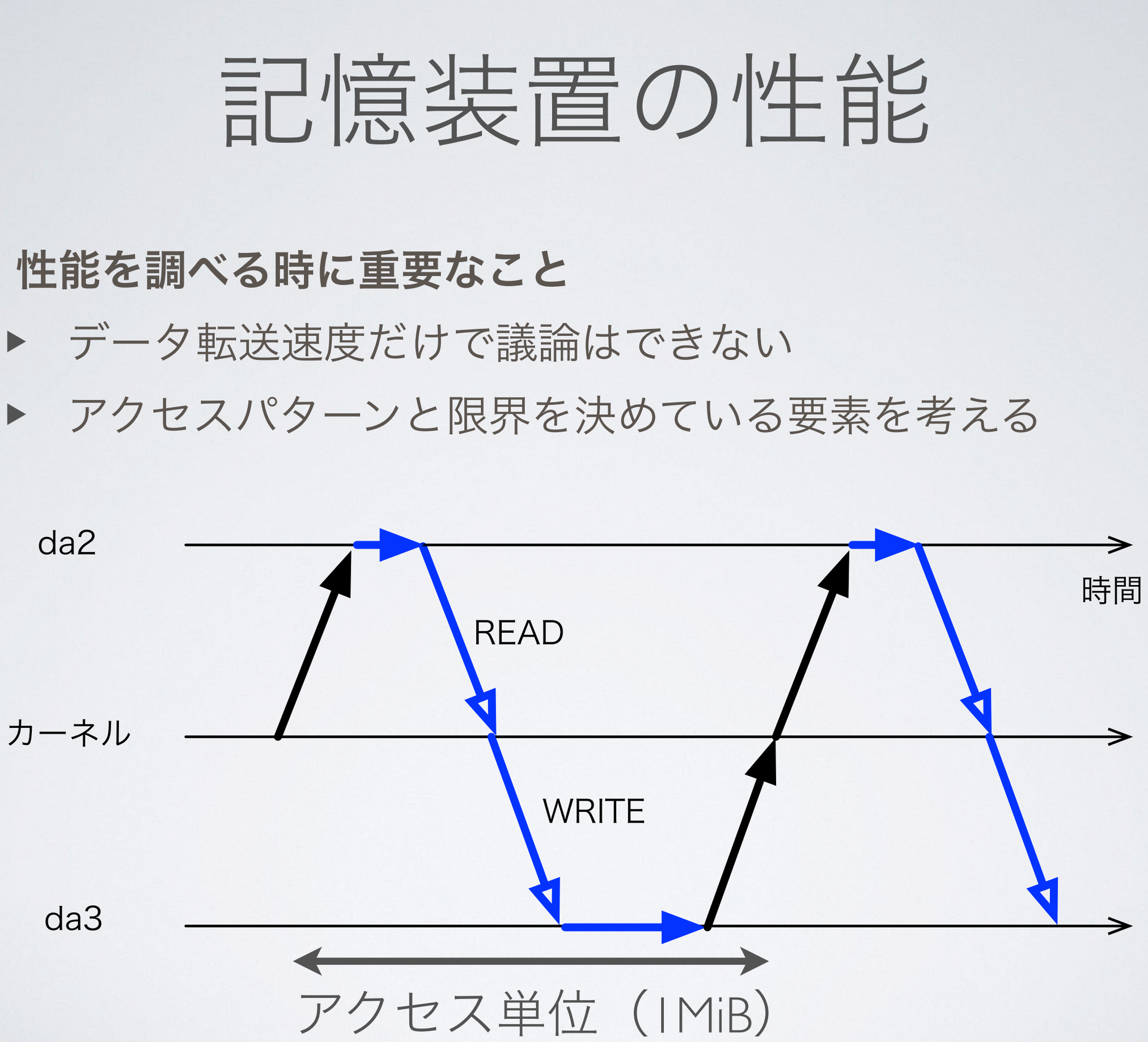
記憶装置の性能

- **性能を調べる時に重要なこと**
  - データ転送速度だけで議論はできない
  - アクセスパターンと限界を決めている要素を考える



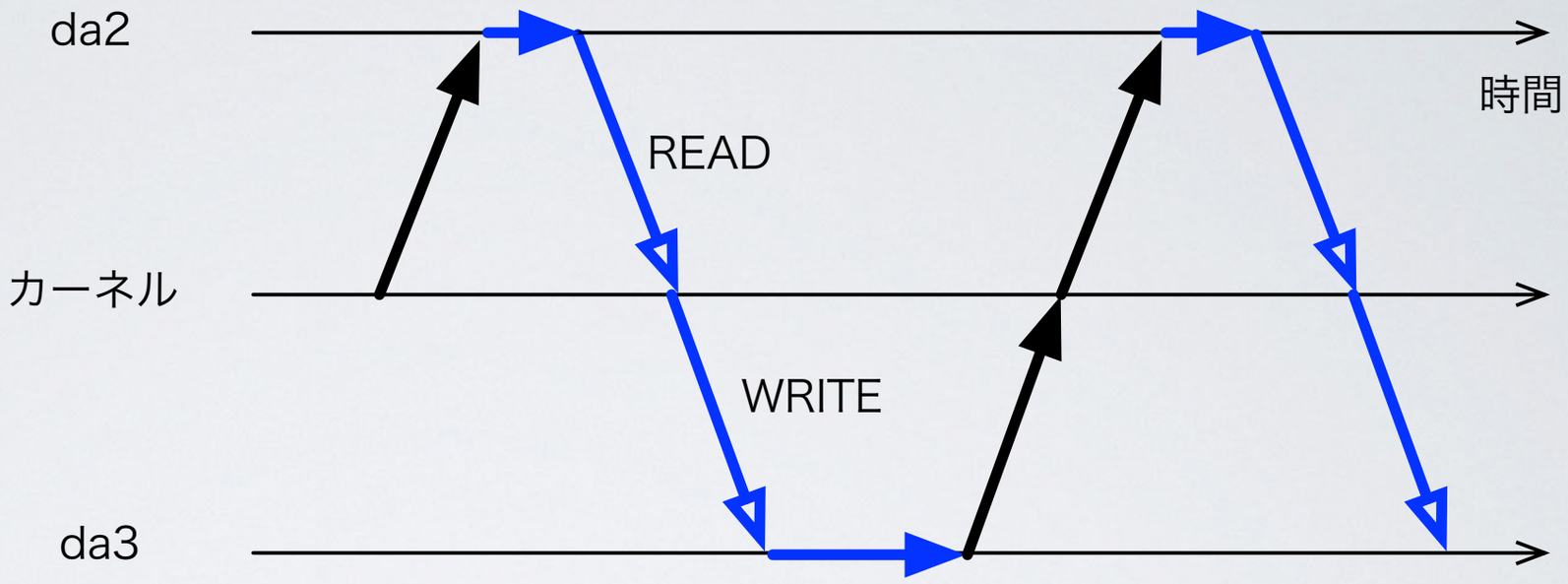
記憶装置の性能

- **性能を調べる時に重要なこと**
  - データ転送速度だけで議論はできない
  - アクセスパターンと限界を決めている要素を考える

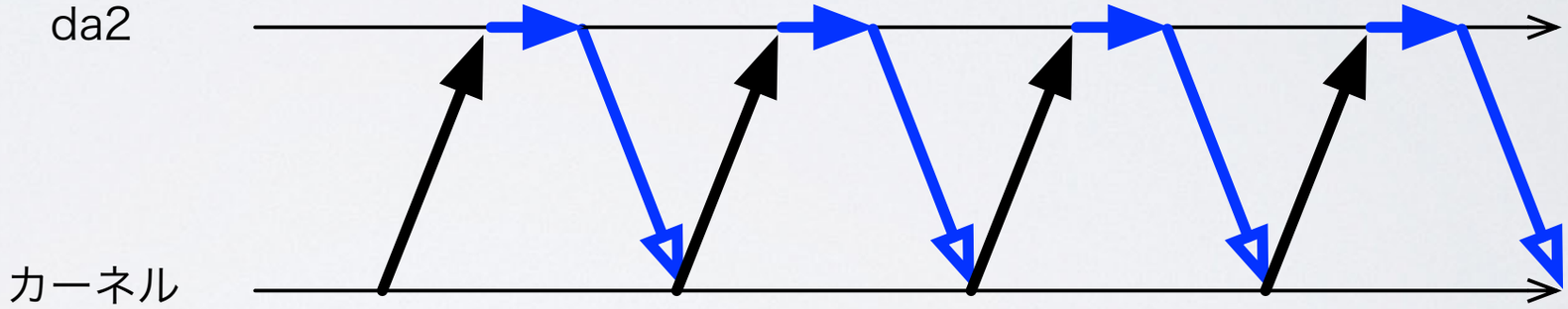


記憶装置の性能

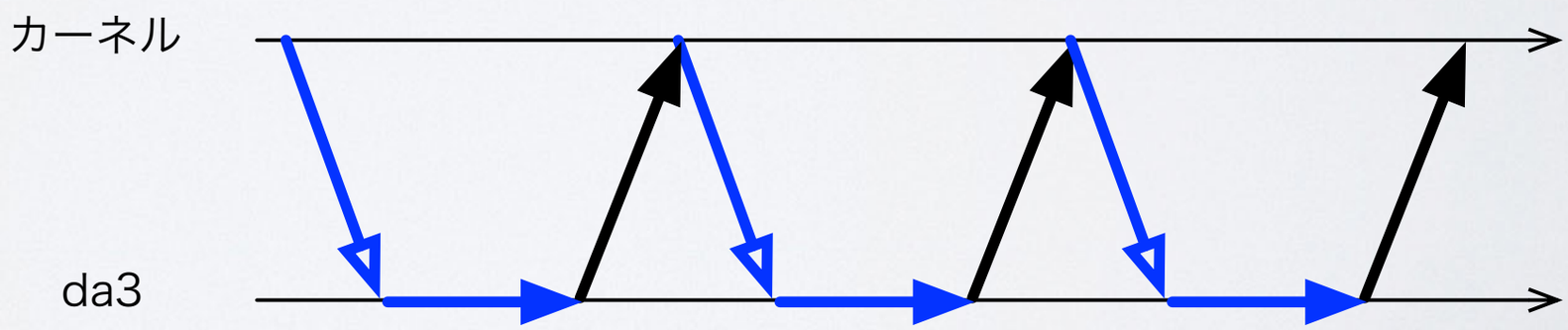
読み書き



読み出し

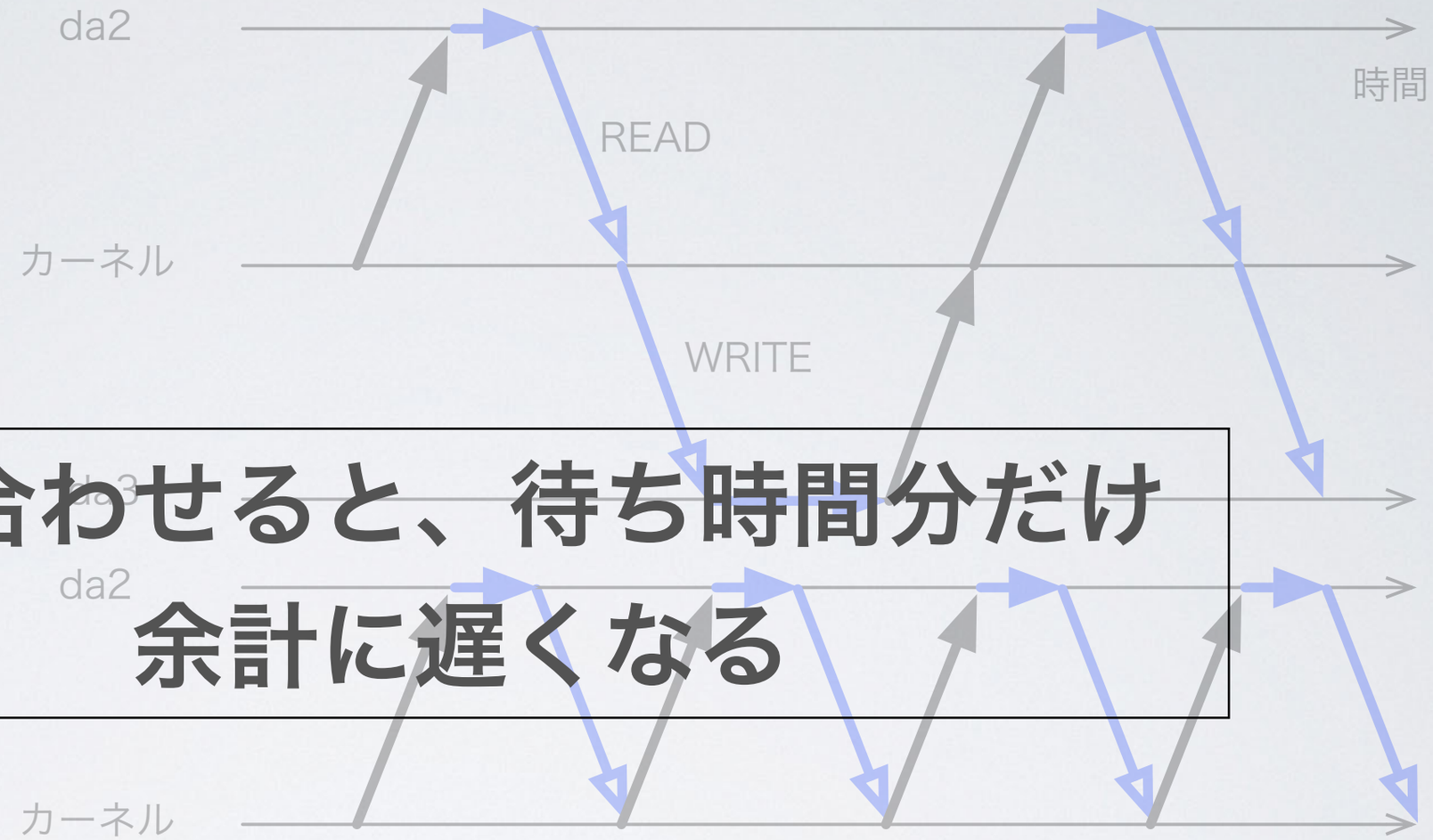


書き込み



記憶装置の性能

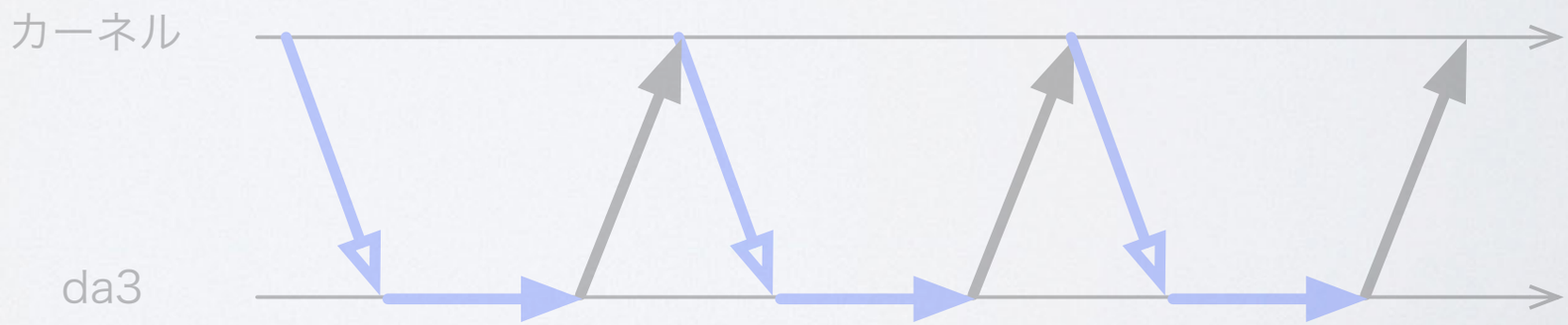
読み書き



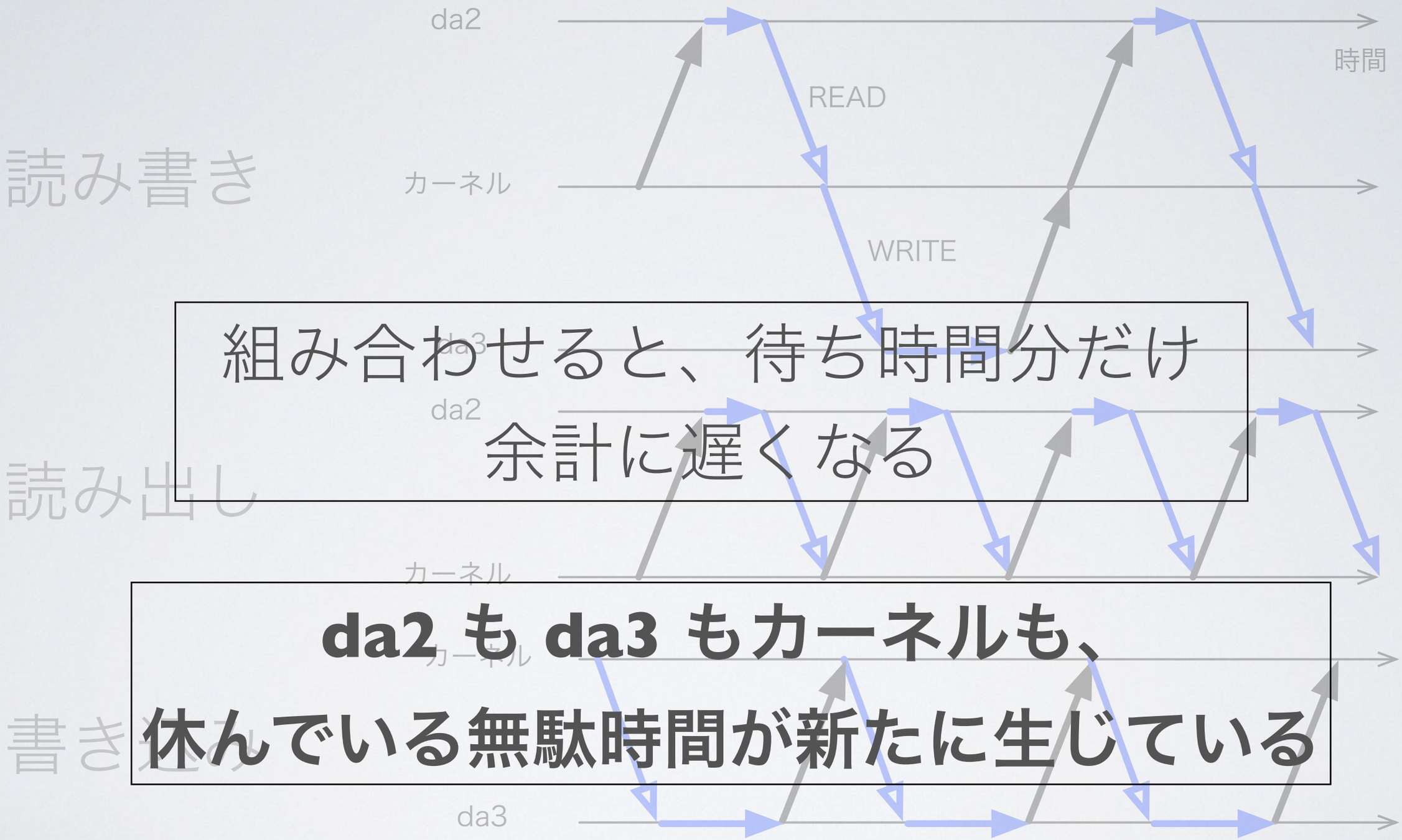
組み合わせると、待ち時間分だけ余計に遅くなる

読み出し

書き込み



記憶装置の性能



組み合わせると、待ち時間分だけ
余計に遅くなる

da2 も da3 もカーネルも、
休んでいる無駄時間が新たに生じている

記憶装置の性能

- **このパターンでの限界性能**
  - 書き込みが一番遅い（12.5MiB/s)
  - 逆に考えれば、そこまでは速度を上げられるはず

記憶装置の性能

- **このパターンでの限界性能**
  - 書き込みが一番遅い（12.5MiB/s)
  - 逆に考えれば、そこまでは速度を上げられるはず

```
# /usr/bin/time -l ¥
  sh -c 'dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 &
  dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 skip=40 seek=40 &
  wait'
40+0 records in
40+0 records out
41943040 bytes transferred in 6.984241 secs (6005383 bytes/sec)
40+0 records in
40+0 records out
41943040 bytes transferred in 7.046337 secs (5952460 bytes/sec)
        7.24 real         0.00 user         0.20 sys
```

- 基本的なアイディア
  - 同時に 2 プロセスから読み書きをして、隙間時間をなくす
  - 12MiB/s 前後まで向上

記憶装置の性能

- このパターンでの限界性能
  - 書き込みが一番遅い（12.5MiB/s)
  - 逆に考えれば、そこまでは速度を上げられるはず

```
# /usr/bin/time -l ¥
  sh -c 'dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 &
  dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 skip=40 seek=40 &
  wait'
40+0 records in
40+0 records out
41943040 bytes transferred in 6.984241 secs (6005383 bytes/sec)
40+0 records in
40+0 records out
41943040 bytes transferred in 7.046337 secs (5952460 bytes/sec)
        7.24 real         0.00 user         0.20 sys
```

こんなの考えるの正直めんどい

- 基本的なアイディア
  - 同時に 2 プロセスから読み書きをして、隙間時間をなくす
  - 12MiB/s 前後まで向上

記憶装置の性能

- このパターンでの限界性能
  - 書き込みが一番遅い（12.5MiB/s)
  - 逆に考えれば、そこまでは速度を上げられるはず

```
# /usr/bin/time -l ¥
  sh -c 'dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 &
  dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=40 skip=40 &
  wait'
40+0 records in
40+0 records out
41943040 bytes transferred in 6.984241 secs (6005383 bytes/sec)
40+0 records in
40+0 records out
41943040 bytes transferred in 7.046337 secs (5952460 bytes/sec)
        7.24 real         0.00 user         0.20 sys
```

実際に使っている状態で情報を調べる

- 基本的なアイディア
  - 同時に 2 プロセスから読み書きをして、隙間時間をなくす
  - 12MiB/s 前後まで向上

記憶装置の性能

- **性能を調べる時に重要なこと**
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80 > /dev/null &
# iostat -x da2 -w 1 | grep ^da2
da2         0.0   0.0      0.0      0.0    0    0.0   0
da2         0.0   0.0      0.0      0.0    0    0.0   0
da2       269.6   0.0  17253.2      0.0    2    3.2  60
da2       452.5   0.0  28960.8      0.0    1    3.2  99
da2       434.6   0.0  27816.0      0.0    2    3.4 100
da2       110.8   0.0   7090.1      0.0    0    3.4  25
da2         0.0   0.0      0.0      0.0    0    0.0   0
da2         0.0   0.0      0.0      0.0    0    0.0   0
```

- iostat = i/o の統計出力コマンド

記憶装置の性能

- **性能を調べる時に重要なこと**
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80 > /dev/null &
# iostat -x da2 -w 1 | grep ^da2
da2         0.0   0.0      0.0      0.0    0    0.0    0
da2         0.0   0.0      0.0      0.0    0    0.0    0
da2       269.6   0.0  17253.2      0.0    2    3.2   60
da2       452.5   0.0  28960.8      0.0    1    3.2   99
da2       434.6   0.0  27816.0      0.0    2    3.4  100
da2       110.8   0.0   7090.1      0.0    0    3.4   25
da2         0.0   0.0      0.0      0.0    0    0.0    0
da2         0.0   0.0      0.0      0.0    0    0.0    0
```

read/sec　　rKiB/sec　　qlen　%busy

write/sec　wKiB/sec　svc_t

記憶装置の性能

- **性能を調べる時に重要なこと**
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80 > /dev/null &
# iostat -x da2 -w 1 | grep ^da2
                        extended device statistics
device     r/s   w/s    kr/s    kw/s wait svc_t  %b
da2        0.0   0.0     0.0     0.0    0   0.0   0
da2        0.0   0.0     0.0     0.0    0   0.0   0
da2      269.6   0.0 17253.2     0.0    2   3.2  60
da2      452.5   0.0 28960.8     0.0    1   3.2  99
da2      434.6   0.0 27816.0     0.0    2   3.4 100
da2      110.8   0.0  7090.1     0.0    0   3.4  25
da2        0.0   0.0     0.0     0.0    0   0.0   0
da2        0.0   0.0     0.0     0.0    0   0.0   0
```

IOPS = 読み書きの回数/秒

rate = データの転送量/秒

記憶装置の性能

- **性能を調べる時に重要なこと**
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価

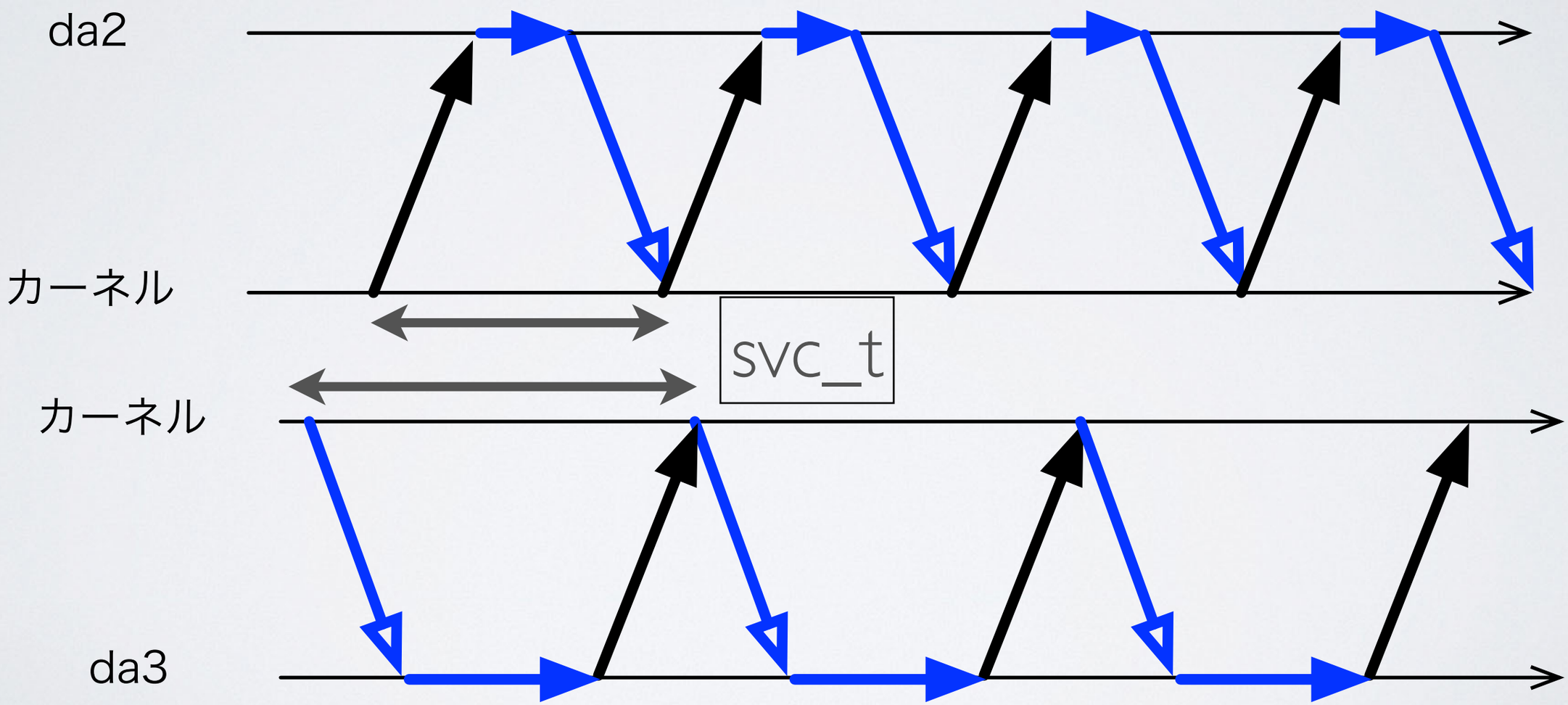
```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80 > /dev/null &
# iostat -x da2 -w 1 | grep ^da2
                        extended device statistics
device     r/s   w/s    kr/s    kw/s  qlen  svc_t  %b
da2        0.0   0.0     0.0     0.0     0    0.0   0
da2        0.0   0.0     0.0     0.0     0    0.0   0
da2      269.6   0.0 17253.2     0.0     2    3.2  60
da2      452.5   0.0 28960.8     0.0     1    3.2  99
da2      434.6   0.0 27816.0     0.0     2    3.4 100
da2      110.8   0.0  7090.1     0.0     0    3.4  25
da2        0.0   0.0     0.0     0.0     0    0.0   0
da2        0.0   0.0     0.0     0.0     0    0.0   0
```

read/sec rKiB/sec qlen %busy
write/sec wKiB/sec svc_t

svc_t = カーネル待ち時間

記憶装置の性能

- **性能を調べる時に重要なこと**
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価



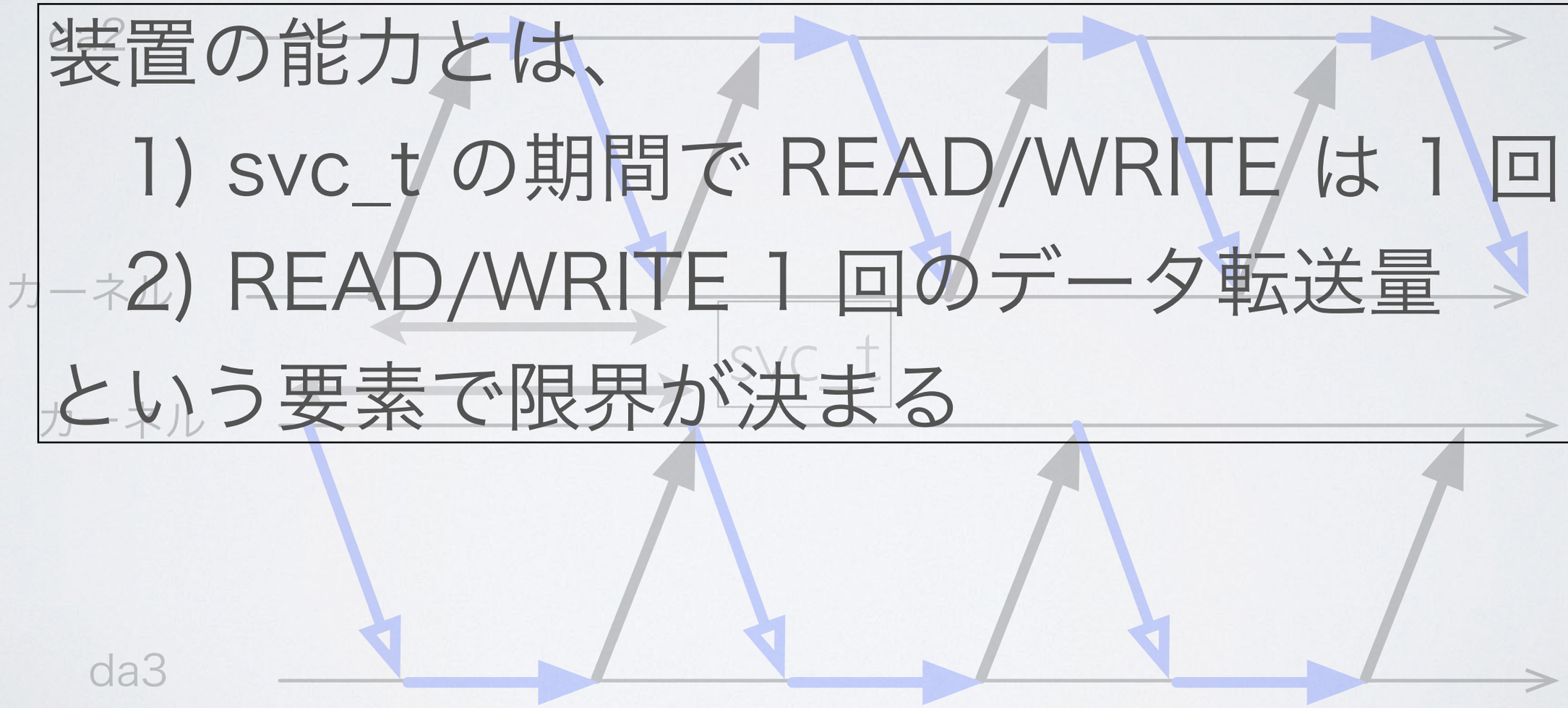
記憶装置の性能

- 性能を調べる時に重要なこと
  - 「データ転送速度」「IOPS（処理遅延）」に分けて評価

装置の能力とは、

1) svc_t の期間で READ/WRITE は 1 回
2) READ/WRITE 1 回のデータ転送量

という要素で限界が決まる



記憶装置の性能

- **さて、こいつの性能は？**

```
# dd if=/dev/da2 of=/dev/null bs=1024x1024 count=80 > /dev/null &
# iostat -x da2 -w 1 | grep ^da2
da2         0.0   0.0      0.0      0.0    0   0.0   0
da2         0.0   0.0      0.0      0.0    0   0.0   0
da2       269.6   0.0  17253.2      0.0    2   3.2  60
da2       452.5   0.0  28960.8      0.0    1   3.2  99
da2       434.6   0.0  27816.0      0.0    2   3.4 100
da2       110.8   0.0   7090.1      0.0    0   3.4  25
da2         0.0   0.0      0.0      0.0    0   0.0   0
da2         0.0   0.0      0.0      0.0    0   0.0   0
```

- 読み出しのsvc_t は 3.2（単位はms）
- 毎秒440回くらい読み出し処理ができてる = 2.3 ms / 回
- 452.5 回 で 28960.8 KBi 転送してる = 64 KiB/回

記憶装置の性能

- **さて、こいつの性能は？**

```
# dd if=/dev/zero of=/dev/da3 bs=1024x1024 count=80 > /dev/null &
# iostat -x da3 -w 1 | grep ^da3
da3         0.0   0.0      0.0      0.0    0    0.0    0
da3         0.0 111.0      0.0   7100.9    2    7.4   56
da3         0.0 196.1      0.0  12550.9    2    7.6  100
da3         0.0 195.1      0.0  12484.3    1    7.6   99
da3         0.0 189.0      0.0  12093.7    0    7.8  100
da3         0.0 191.2      0.0  12239.6    1    7.7   99
da3         0.0 196.0      0.0  12545.2    1    7.6   99
```

- 読み出しのsvc_t は 7.6（単位はms）
- 毎秒195回くらい読み出し処理ができてる = 2.3 ms / 回
- 196.1 回 で 12550.9 KBi 転送してる = 64 KiB/回

記憶装置の性能

- **さて、こいつの性能は？**
  - IOPS: 読み 440, 書き 195（64KiB アクセス時）
- べたコピーだと「読む→書く」だから
- 一回あたり 1/440 + 1/195 秒かかる。
- 1秒で 135.1 回。64KiB/回だと **64 x 135.1 = 8.4 MiB/s**

記憶装置の性能

- **さて、こいつの性能は？**
  - IOPS: 読み 440, 書き 195（64KiB アクセス時）
- べたコピーだと「読む→書く」だから
- 一回あたり 1/440 + 1/195 秒かかる。
- 1秒で 135.1 回。64KiB/回だと **64 x 135.1 = 8.4 MiB/s**

```
# dd if=/dev/da2 of=/dev/da3 bs=1024x1024 count=80
80+0 records in
80+0 records out
83886080 bytes transferred in 9.370823 secs (8951837 bytes/sec)
```

- 結果は 8.5 MiB/s （約1/1000）

記憶装置の性能

- **svc_t って何に使うの？**
  - カーネルが出せる1秒あたりの r/w 要求数の限界値は、この逆数に支配される
  - これよりも多く突っ込もうとすると、待たされる

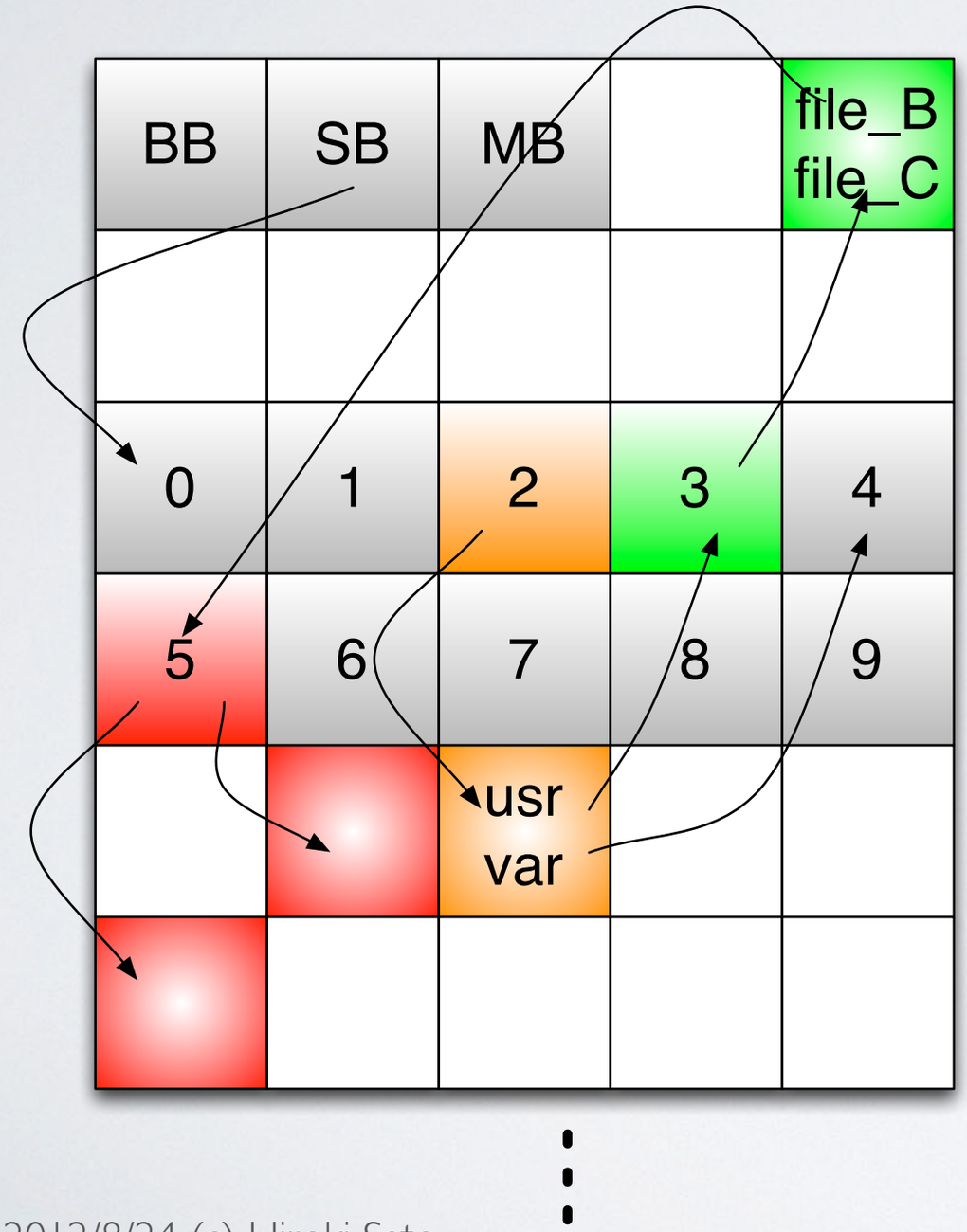
- svc_t の小さい記憶装置は高価
  - 回転数の速いHDD
  - SSD
  - HDDの制御チップの性能でももちろん変わる

記憶装置の性能

- **このパラメータは転送データの単位でも変わる**
  - **64KiB アクセス**
    - IOPS: 読み 440, 書き 195
    - svc_t: 読み 2.3, 書き 5.1
  - **1KiB アクセス**
    - IOPS: 読み 1270, 書き 1150
    - svc_t: 読み 0.8, 書き 1.0
- 記憶装置のアクセス = 固定時間＋データ量依存時間

ファイルシステムへの影響

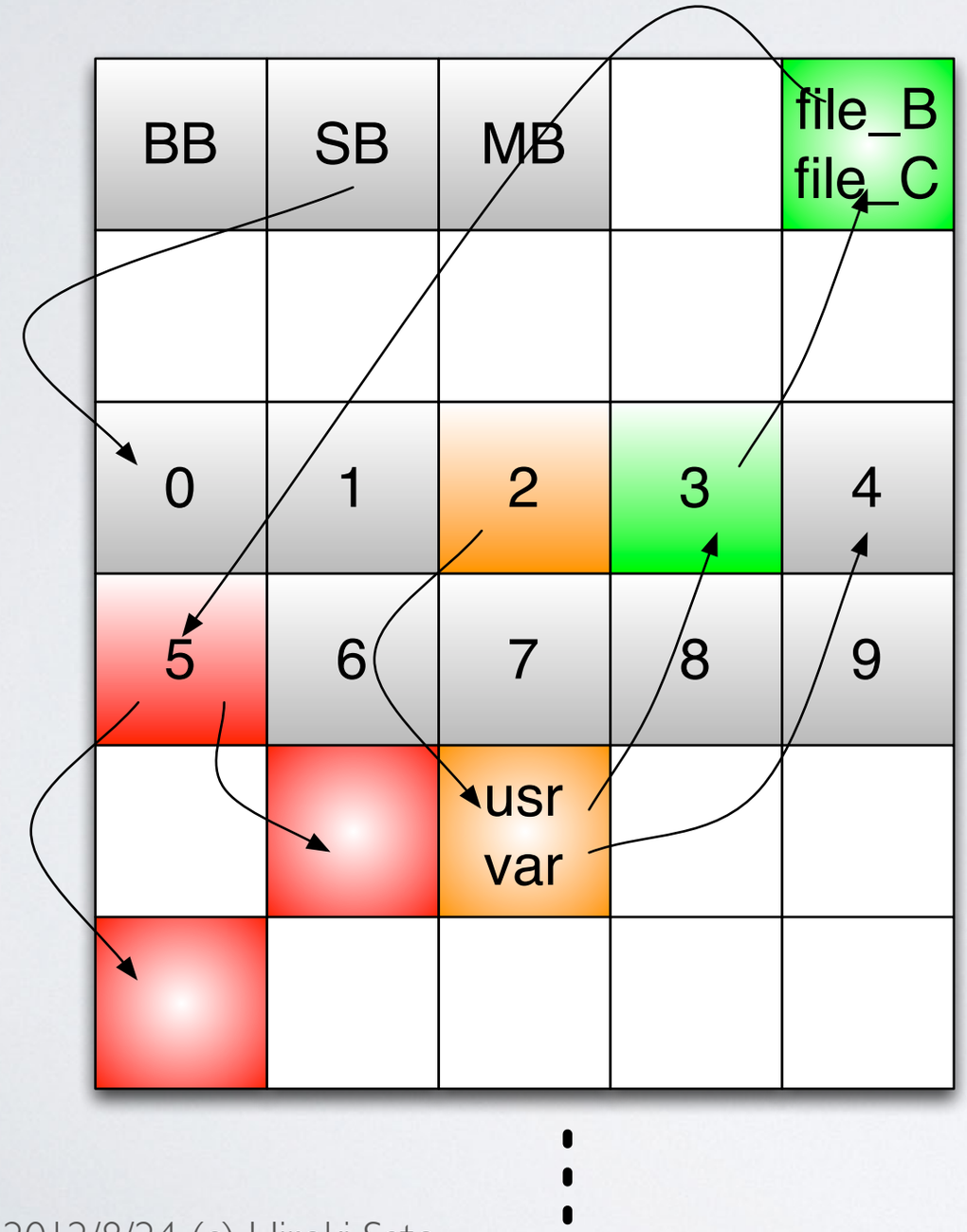
- **UFS の速度**



- いつも SB から inode へのリンクを数珠つなぎにアクセス
- アクセスが inode 付近に集中する

ファイルシステムへの影響

- **UFS の速度**



- いつも SB から inode へのリンクを数珠つなぎにアクセス
- アクセスが inode 付近に集中する
- 1万ファイルが置いてあるディレ

ファイルシステムへの影響

- **inode の操作が、全体のアクセス時間を制限しやすい**
  「メタデータ」へのアクセスと表現することが多い

- 例：1万ファイル全部の inode を読むアクセス
  - 1000 IOPS のデバイスで単純に 10 秒
    （実際にはキャッシュがかかるのでもっと速い）
- FreeBSD が備えている機能
  - UFS_DIRHASH （inode にある情報のキャッシュ）
    - vfs.ufs.dirhash_maxmem: 2097152
  - SoftUpdates （inode への書き込みを減らす）

記憶装置の性能

- **まとめ**
  - 自分が使いたいアクセスパターンが発生している時の、**記憶装置の svc_t と IOPS** を調べよう
  - **カーネルが1 回あたりに読み書きする単位**を調べよう
    - BSD系OSは基本的に 64KiB を使う

- **%b が 100 になっていなければ、記憶装置の限界まで使い切っていないので、高速化の余地はある**

記憶装置の性能

- **まとめ**
  - 自分が使いたいアクセスパターンが発生している時の、**記憶装置の svc_t と IOPS** を調べよう
  - **カーネルが1 回あたりに読み書きする単位**を調べよう
    - BSD系OSは基本的に 64KiB を使う
  - **%b が 100 になっていなければ、記憶装置の限界まで使い切っていないので、高速化の余地はある**
  - iostat はデバイス単位だが、同じ情報を GEOM 単位で出せる gstat というものもある

GEOM の活用例

- すでに /dev/ada0 がある時のシステムディスクのミラー
- パーティション分割とGPT
- RAID0, RAID1, RAID3 の構成と性能差
- HAST

gmirror の設定

- 手順
  - /dev/ada1 を使って /dev/mirror/gm0 を生成する
  - /dev/ada0 の内容を /dev/mirror/gm0 へコピー
  - /dev/mirror/gm0 を起動ディスクにして再起動
  - /dev/ada0 を /dev/mirror/gm0 に追加
  - ada0 と ada1 のデータが同期するまで待つ

注意：

FreeBSD ハンドブックの手順は間違っているので、参考にしないこと！

gmirror の設定

```
# kldload geom_mirror

# gmirror label -b round-robin gm0 /dev/ada1
GEOM_MIRROR: Device mirror/gm0 launched (1/1)

# gpart backup ada0 > /tmp/ada0.txt
# gpart backup ada0s1 > /tmp/ada0s1.txt
# gpart restore mirror/gm0 < /tmp/ada0.txt
# gpart restore mirror/gm0s1 < /tmp/ada0s1.txt

# newfs -U /dev/mirror/gm0s1a
# newfs -U /dev/mirror/gm0s1d
# newfs -U /dev/mirror/gm0s1e

# mount /dev/mirror/gm0s1a /mnt
# dump aL0Cbf 32 64 - / | (cd /mnt && restore -rf -)
# mount /dev/mirror/gm0s1d /mnt/var
# mount /dev/mirror/gm0s1e /mnt/usr
# dump aL0Cbf 32 64 - /var | (cd /mnt/var && restore -rf -)
# dump aL0Cbf 32 64 - /usr | (cd /mnt/usr && restore -rf -)
```

gmirror モジュールのロード

ada1 から mirror/gm0 を作成

ada0 のラベルを mirror/gm0 にコピー

ada0 と同じファイルシステムを mirror/gm0 に作成

ada0 のデータをコピー

gmirror の設定

```
# cat /dev/ada0.txt
MBR 4
1 freebsd   63 20971440 [active]
# cat /dev/ada0s1.txt
BSD 8
1 freebsd-ufs         0  2097152
2 freebsd-swap  2097152   524288
4 freebsd-ufs   2621440  8388608
5 freebsd-ufs  11010048  8388608

# gpart list ada0 | grep last
last: 20971518

# gpart list mirror/gm0 | grep last
last: 20971517
```

MBRパーティションの開始セクタと総セクタ数

BSDパーティションの開始セクタと総セクタ数

ada0の総セクタ数

mirror/gm0 の総セクタ数

注意：ada0 よりも gm0 は 512B だけ小さいので、
必要ならばパーティション情報を書き換える

gmirror の設定

- 起動ディスクを変更して、かつ
  geom_mirror モジュールを自動的に読み込むようにする。

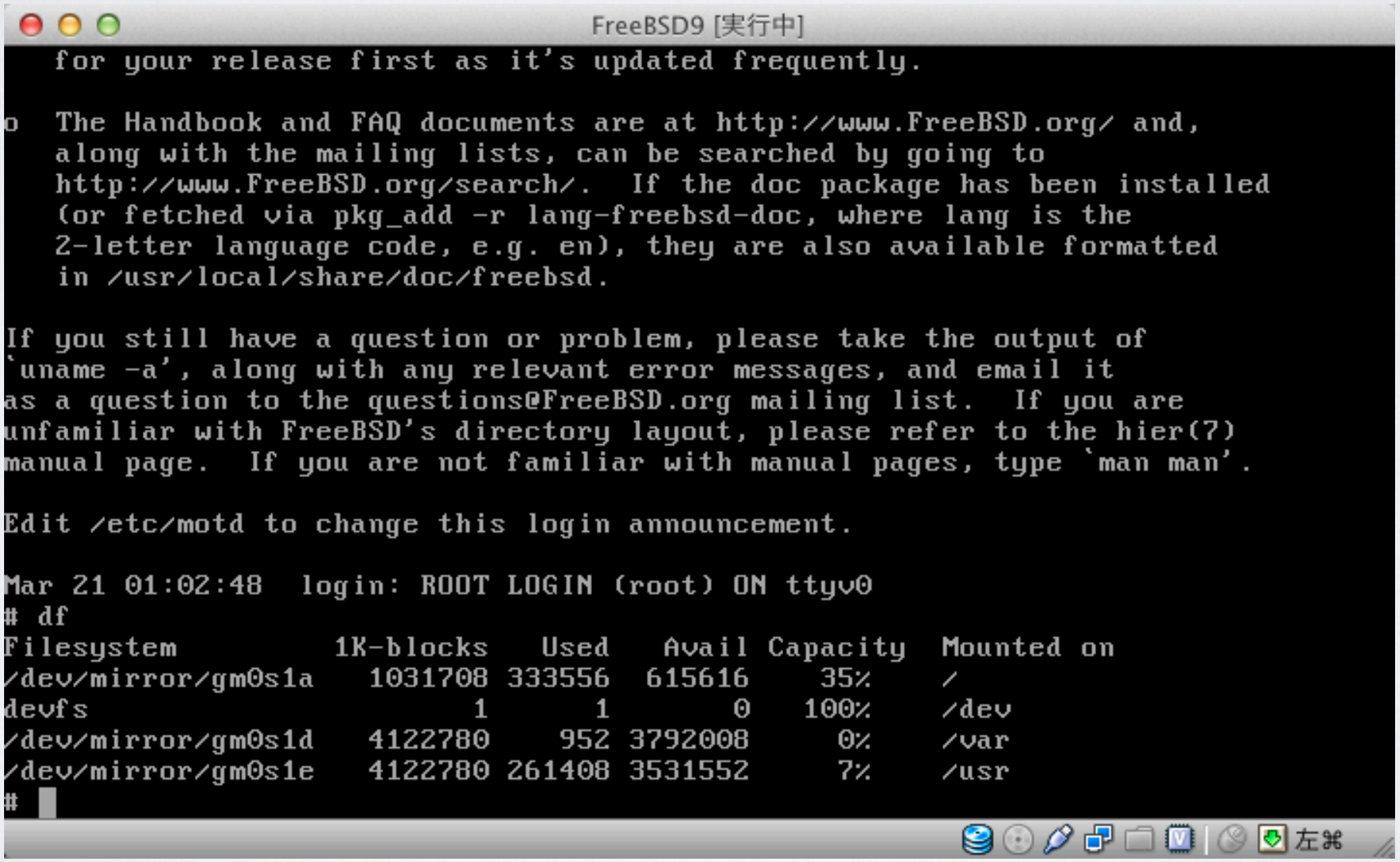
```
# sed -e 's,/dev/ada0,/dev/mirror/gm0,' < /etc/fstab > /tmp/fstab.new
# cp /tmp/fstab.new /mnt/etc/fstab
# echo "geom_mirror_load=YES" > /boot/loader.conf
# cp /boot/loader.conf /mnt/boot/loader.conf

# cat /etc/fstab
/dev/mirror/gm0a1b none swap sw 0 0
/dev/mirror/gm0a1a / ufs rw 1 1
/dev/mirror/gm0a1d /var ufs rw 2 2
/dev/mirror/gm0a1e /usr ufs rw 2 2
# cat /mnt/etc/fstab
# cat /boot/loader.conf
geom_mirror_load=YES
# cat /mnt/boot/loader.conf

# reboot
```

gmirror の設定

- 再起動後は mirror/gm0 しか使っていない
  （実体は ada1）



gmirror の設定

- mirror/gm0 に ada0 を追加

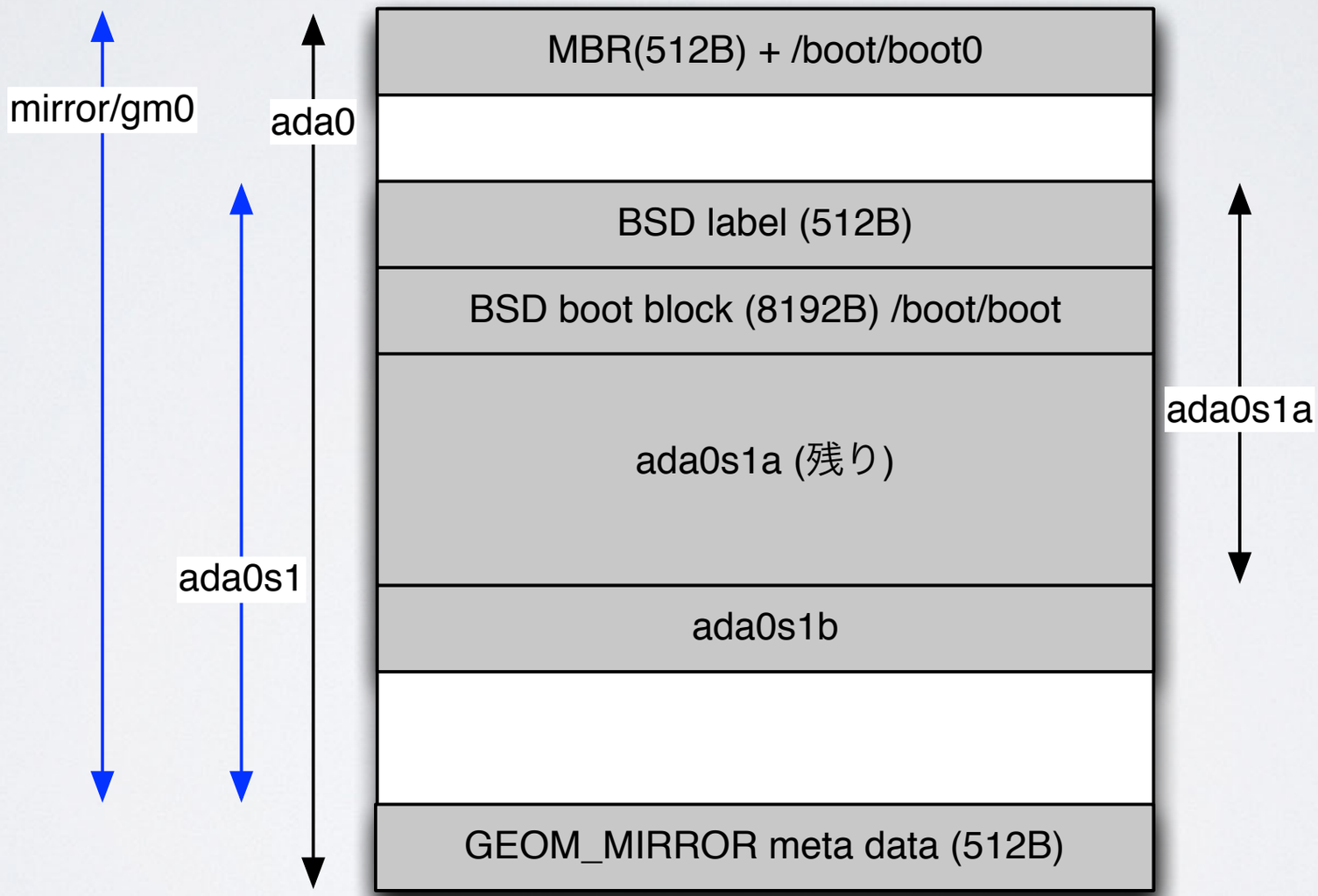
```
# gmirror insert gm0 /dev/ada0
GEOM_MIRROR: Device gm0: rebuilding provider ada0
# gmirror status
      Name    Status  Components
mirror/gm0  DEGRADED  ada1 (ACTIVE)
                      ada0 (SYNCHIRONIZING, 64%)
#
GEOM_MIRROR: Device gm0: rebuilding provider ada0 finished.
# gmirror status
      Name    Status  Components
mirror/gm0  COMPLETE  ada1 (ACTIVE)
                      ada0 (ACTIVE)
#
```

gmirror の設定

- これで ada0 と ada1 がミラーされた状態に
  - ディスク(例えばada1)が壊れたら、それを取り外して新しいのに交換。
  - gmirror forget gm0 で ada1 の設定が消える
  - gmirror insert gm0 ada1 で、gm0 に ada1を再度追加
- BIOS は /dev/ada0 から起動することに注意。
  - gm0 が見えるのはカーネルをロードした後だけ！
  - さきほどの例の再起動は、ada0 で起動して、起動後に mirror/gm0 に移っている
  - ada0 が完璧に壊れると起動できなくなる！

gmirror の設定

- これで ada0 と ada1 がミラーされた状態に



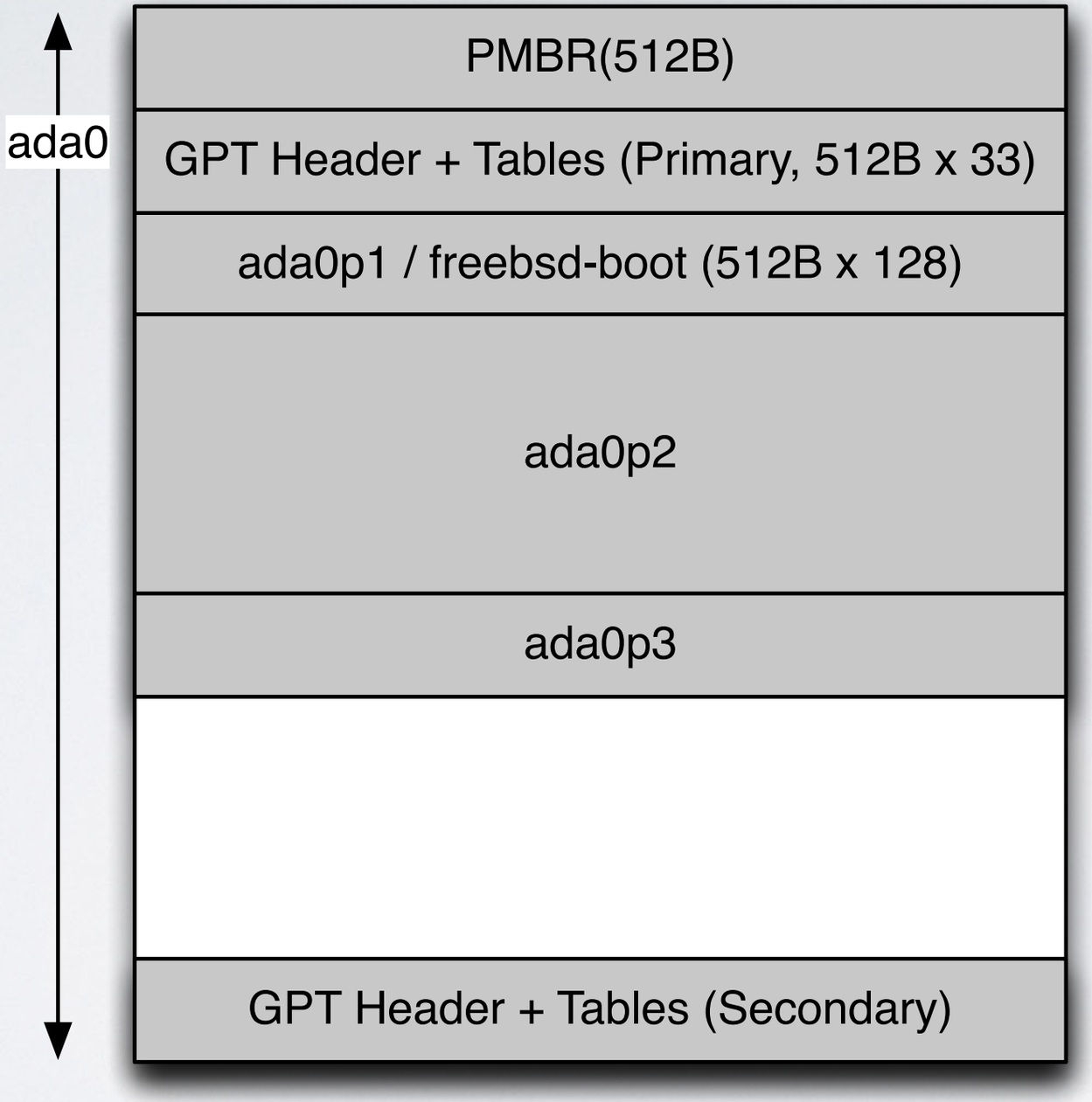
ストレージ構成の戦略

- パーティションを分けた方が良いのか?
  - サーバ用途なら、下記をおすすめ
    / = 3GB
    swap = メインメモリと同じ大きさ
    /var = 30GB
    /usr = 20GB
    /tmp = 1GB のメモリディスク
    残り = サービスに必要な容量
  - 分けないと、ディスクが壊れた時の被害が大きくなる
  - 分けないと、バックアップを取るのが面倒になる
  - デスクトップ用途なら “/” 一個でも良いが...

ストレージ構成の戦略

- GPTにした方が良いのか?
  - MBR は 2TB を超えるパーティションが作れない
  - これでGPTになる:
    - gpart create -s gpt ada0
    - gpart add -t freebsd-boot -s 128 ada0
    - gpart bootcode -b /boot/pmbr -p /boot/gptboot ada0
    - デバイスノード名は ada0p1, ada0p2, ... のようになる
- 9.0R からは GPT がデフォルトに

ストレージ構成の戦略

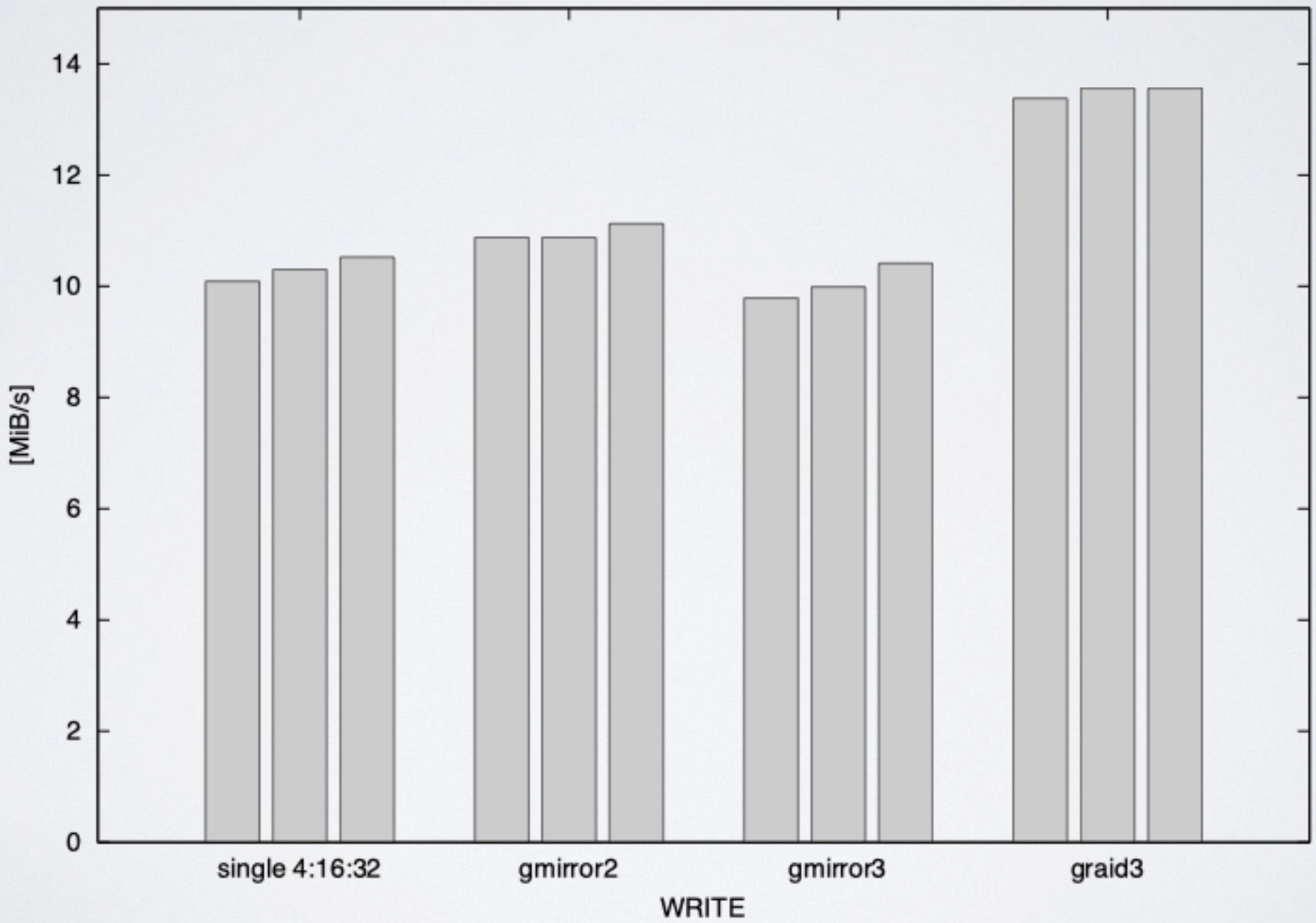

ada0
PMBR(512B)
GPT Header + Tables (Primary, 512B x 33)
ada0p1 / freebsd-boot (512B x 128)
ada0p2
ada0p3
GPT Header + Tables (Secondary)

RAID0, 1, 3 の構成と性能差

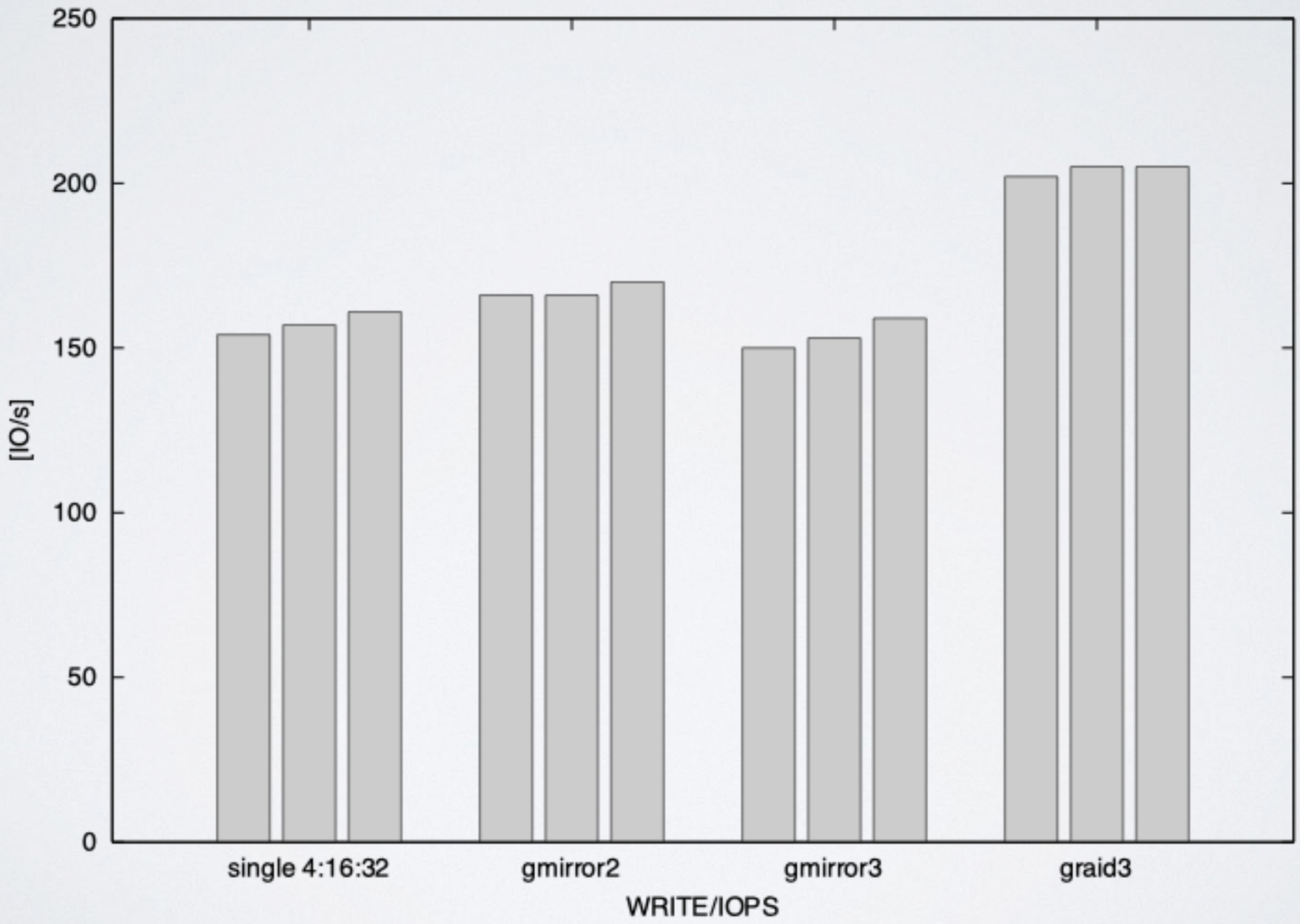
- かなり低速な HDD 3 台での比較
- 15,000回の W, R+W 混合アクセスを発生
- 隙間時間を空けないように、4, 16, 32 プロセスで読み書き
- アクセスサイズは 64KiB よりも小さい一様分布
- IOPS とデータ転送量を調査

- （7 年くらい前の、かなーり古いデータではあります）

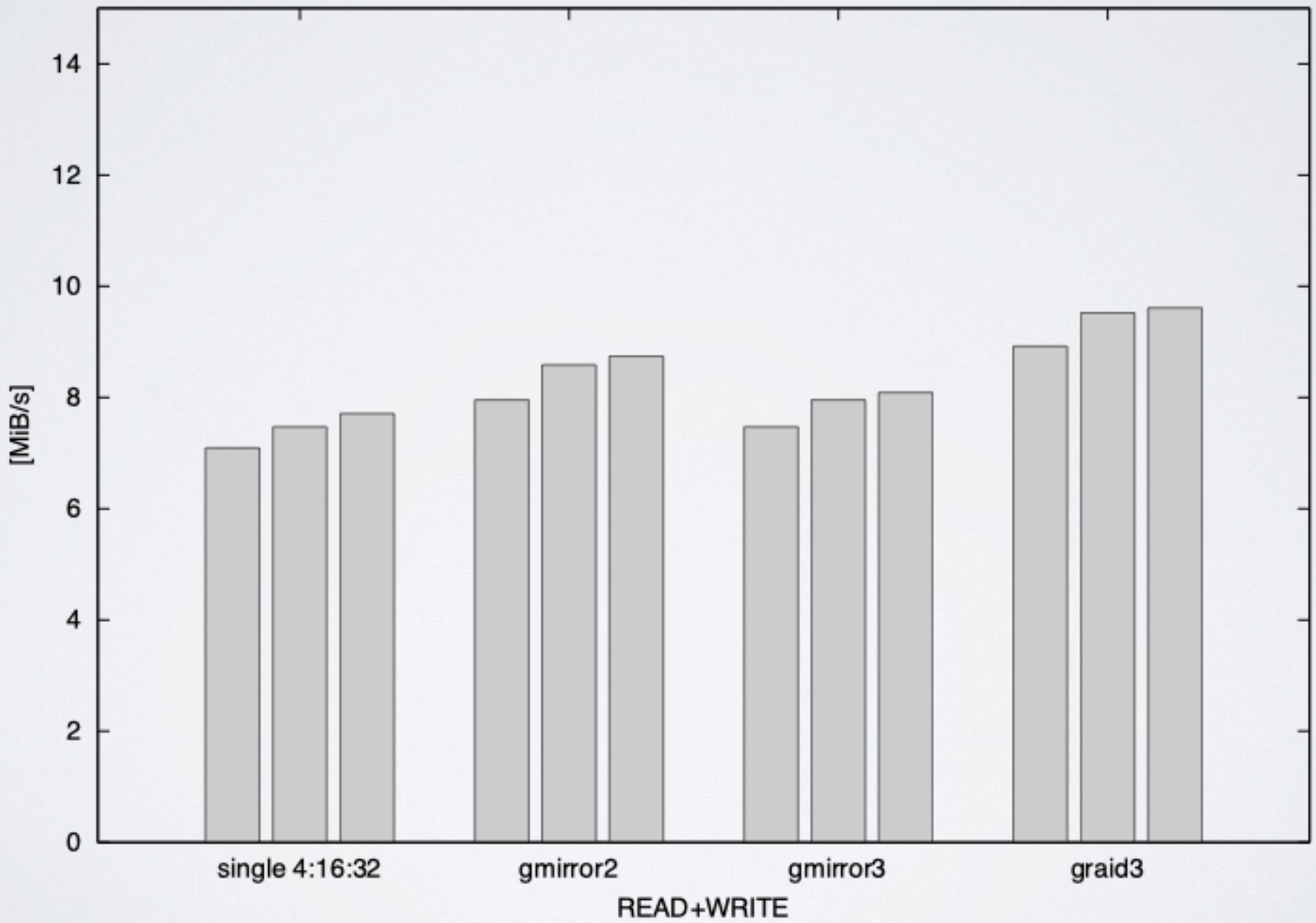
RAID0, 1, 3 の構成と性能差



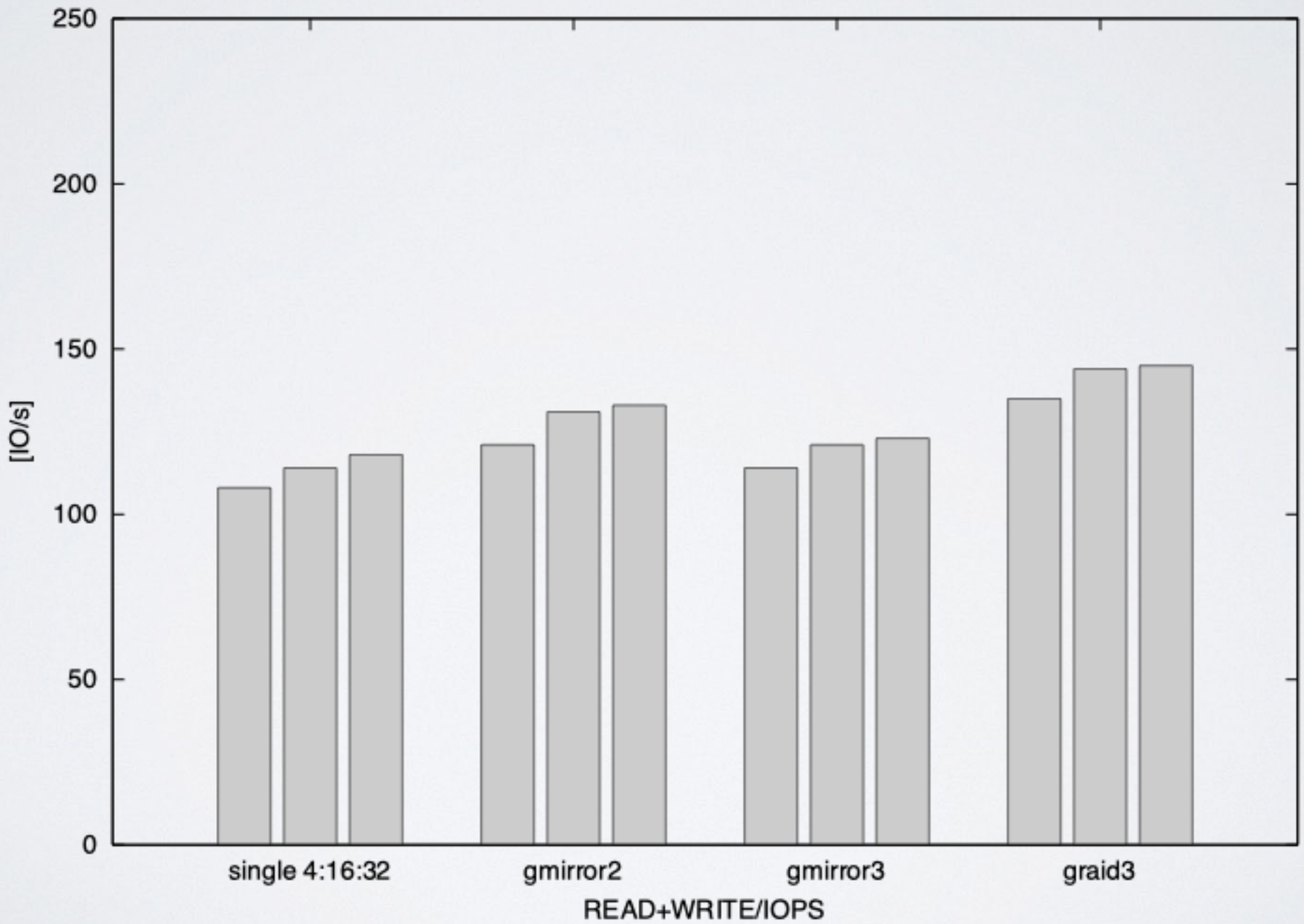
RAID0, 1, 3 の構成と性能差



RAID0, 1, 3 の構成と性能差



RAID0, 1, 3 の構成と性能差



HAST

- **GEOM ベースでつくられた高可用性ブロックストレージ**
  - gmirror をネットワーク越しに行うもの
  - VRRP や CARP と組み合わせると、
    片方が壊れても動き続けるストレージサーバが作れる

HAST

- **GEOM ベースでつくられた高可用性ブロックストレージ**
  - gmirror をネットワーク越しに行うもの
  - VRRP や CARP と組み合わせると、
    片方が壊れても動き続けるストレージサーバが作れる
  - /etc/hast.conf に、次のように設定できる

```
resource sampleres {
        on machine-a {
                local /dev/ada1
                remote 192.168.2.1
        }
        on machine-b {
                local /dev/ada1
                remote 192.168.1.1
        }
}
```

HAST

- **GEOM ベースでつくられた高可用性ブロックストレージ**
  - gmirror をネットワーク越しに行うもの
  - VRRP や CARP と組み合わせると、
    片方が壊れても動き続けるストレージサーバが作れる
  - あとは hastctl コマンドで制御(hastd_enable="YES")

```
machine-a# hastctl create sampleres
machine-a# /etc/rc.d/hastd start
machine-a# hastctl role primary sampleres
```

```
machine-b# hastctl create sampleres
machine-b# /etc/rc.d/hastd start
machine-b# hastctl role secondary sampleres
```

HAST

- **GEOM ベースでつくられた高可用性ブロックストレージ**
  - gmirror をネットワーク越しに行うもの
  - VRRP や CARP と組み合わせると、
    片方が壊れても動き続けるストレージサーバが作れる
  - あとは hastctl コマンドで制御
  - 設定できたら/dev/hast/sampleres が生成されるので、
    そこにデータを入れる

ZFS

- **近年開発されたFSのひとつ**
  - 従来:
    デバイス→ボリュームマネージャ→ファイルシステム
  - ZFS：
    ZPOOL→DMU→ZFS
    - たくさんの記憶装置が存在することを前提に、PB 級のスケーラビリティに優れた構造を達成することが主目的

ZFS

- **近年開発されたFSのひとつ**
  - 従来:
    デバイス→ボリュームマネージャ→ファイルシステム
  - ZFS：
    ZPOOL→DMU→ZFS
    - たくさんの記憶装置が存在することを前提に、PB 級のスケーラビリティに優れた構造を達成することが主目的
    - もちろん今までのFSの欠点は潰しているが、小さい規模では嬉しくない部分も多い。適材適所。
    - ZFSは堅牢だとかスケールするだとかいうセールストークは5割引くらいで読みましょう

ZFS

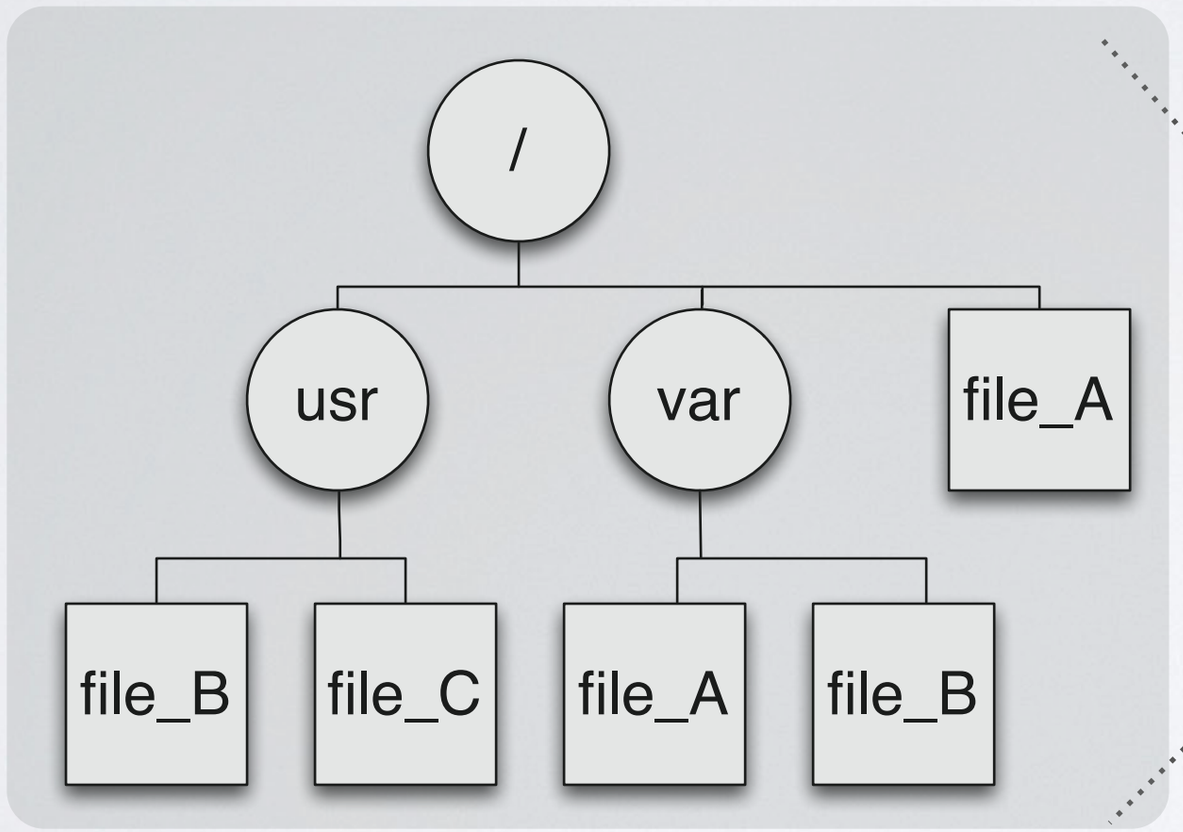
- **コンセプト**
  - **ZPOOL**: ブロックデバイスの集まりに名前を付けたもの
    - 例）/dev/ada0 + /dev/ada1 で 1 個の ZPOOL を構成

ZFS

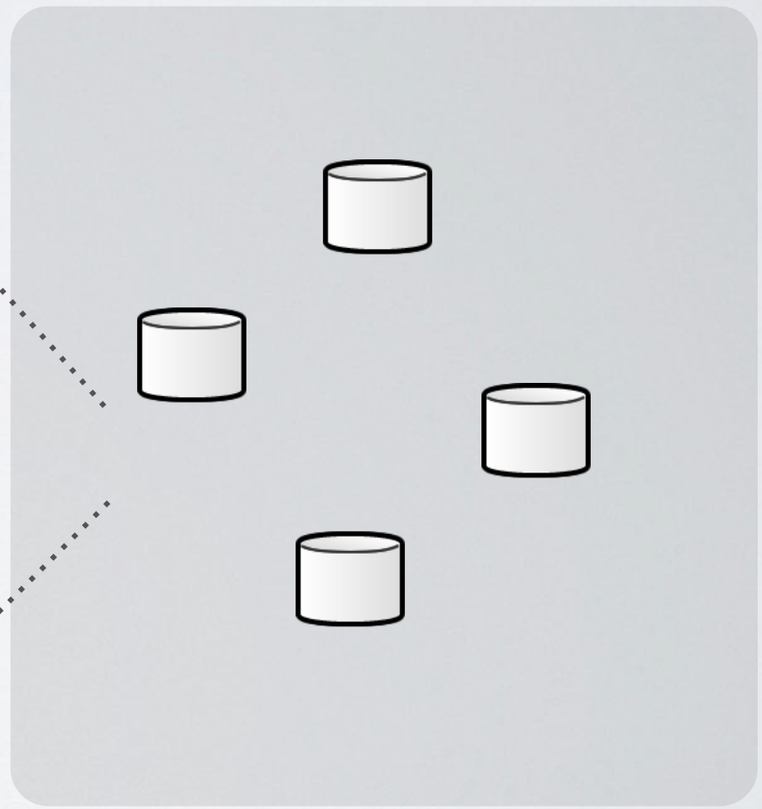
- **コンセプト**
  - **ZPOOL**: ブロックデバイスの集まりに名前を付けたもの
    - 例）/dev/ada0 + /dev/ada1 で 1 個の ZPOOL を構成
    - 構成の方法にバリエーションがある
      - /dev/ada0 と /dev/ada1 をミラーしつつ構成
      - /dev/ada{0,1,2,3,4} で RAID5 を組んで構成
      - どれも、GEOM の時と同じように「1 個」に見える

ZFS

- **コンセプト**
  - **ZFSデータセット**: ZPOOLの上に構成できる名前空間
    - ファイルシステムなど



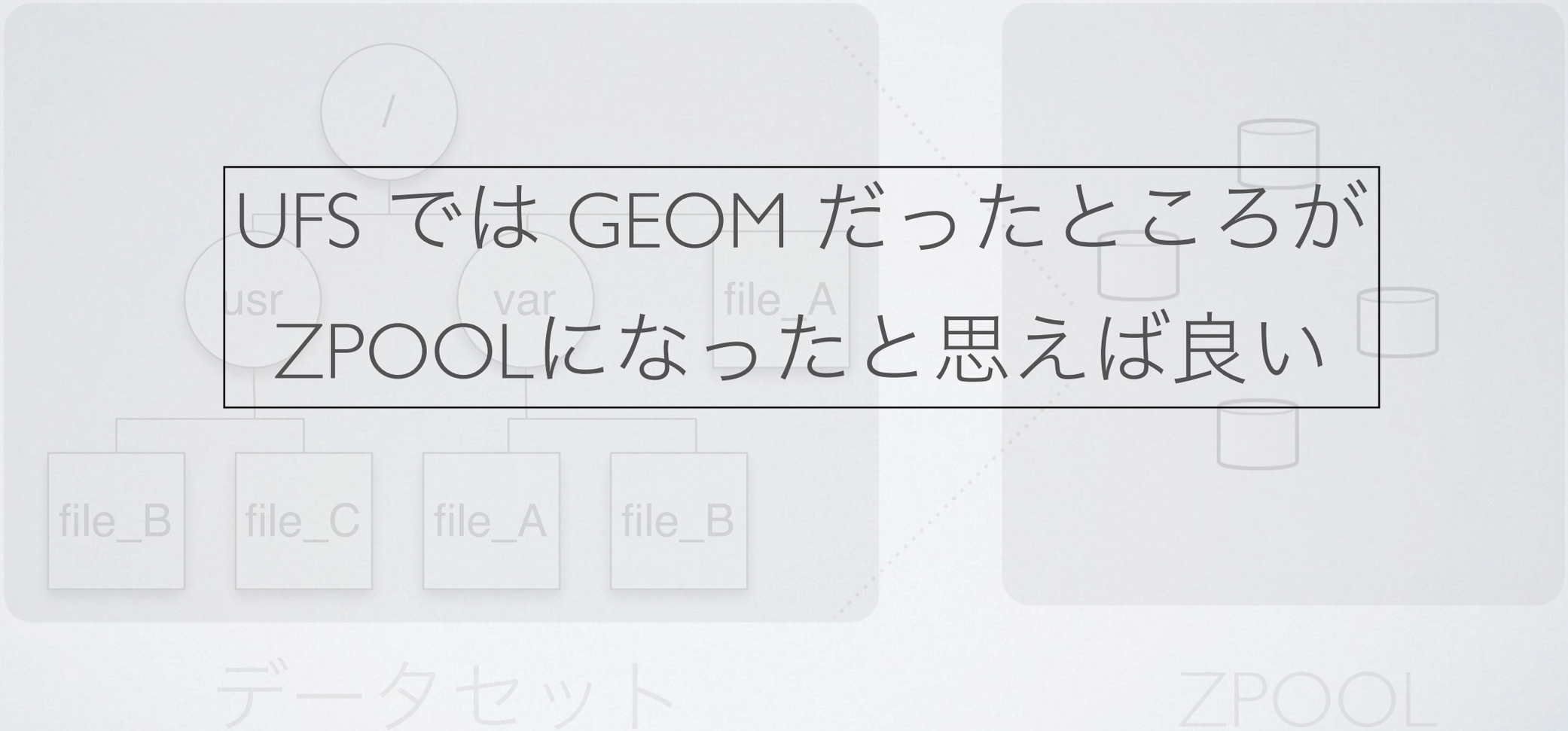
データセット



ZPOOL

ZFS

- **コンセプト**
  - **ZFSデータセット**: ZPOOLの上に構成できる名前空間
    - ファイルシステムなど



UFS では GEOM だったところが
ZPOOLになったと思えば良い

ZFS

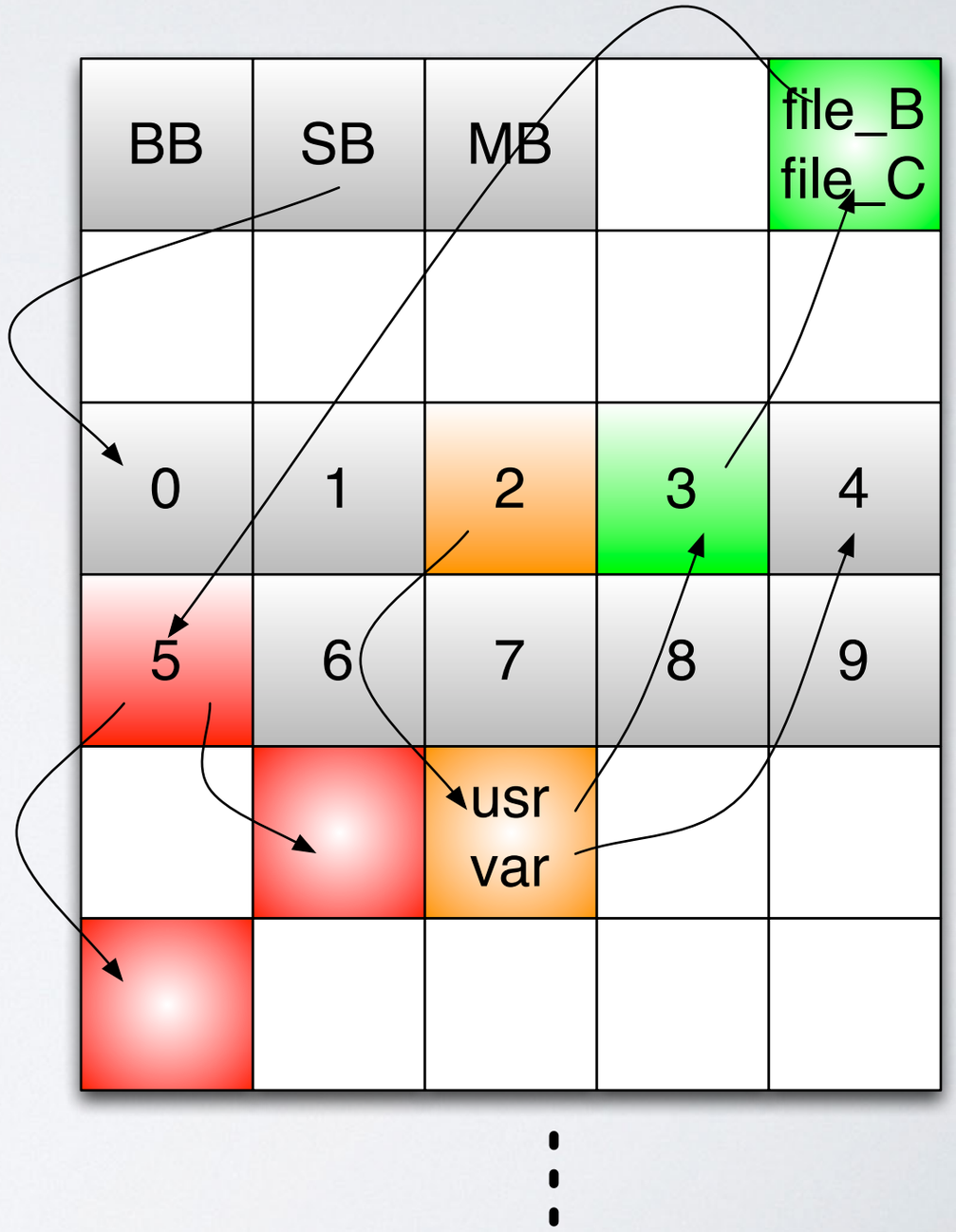
- **コンセプト**
  - ZPOOLで記憶装置を管理
  - ZFSデータセット1個＝ファイルシステム1個（相当）

ZFS

- **コンセプト**
  - ZPOOLで記憶装置を管理
  - ZFSデータセット1個＝ファイルシステム1個（相当）
- **dnode:**
  ZPOOL 上にZFSデータセットを構成するために
  作られる、UFS の inode に似たリンク用の情報単位
- **DMU(Data Management Unit)**:
  dnode の操作を担当するカーネルの部分

ZFSの構造

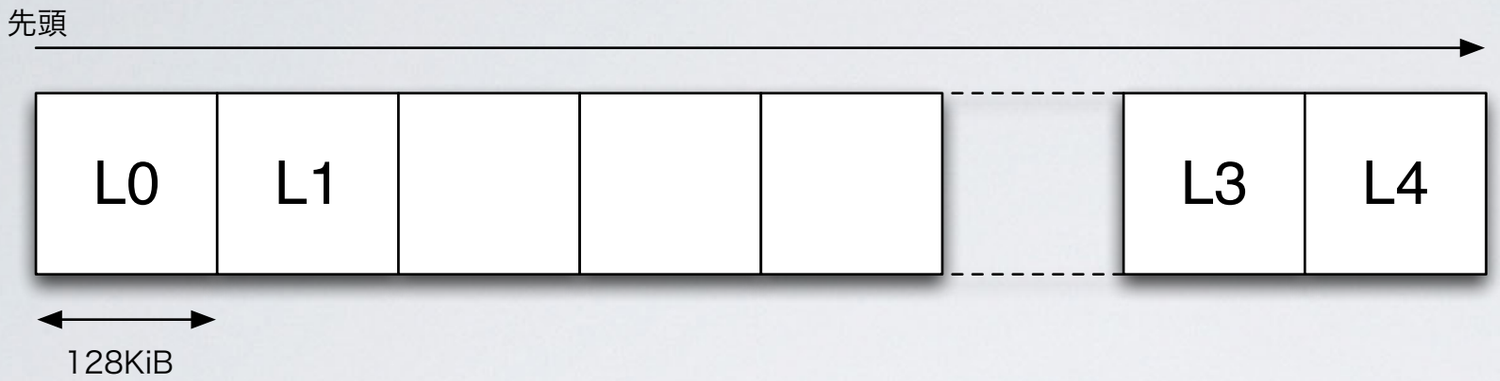
- **UFS の復習**
  - 最初はスーパーブロック。ここには inode の在処がある
  - inode はディレクトリ・ファイルの情報と、データブロックの場所を記録しているところ
  - inode は配列状に並んでいる。ルートディレクトリの場所だけ、予め決まっている
  - あとはとにかくリンクを辿る



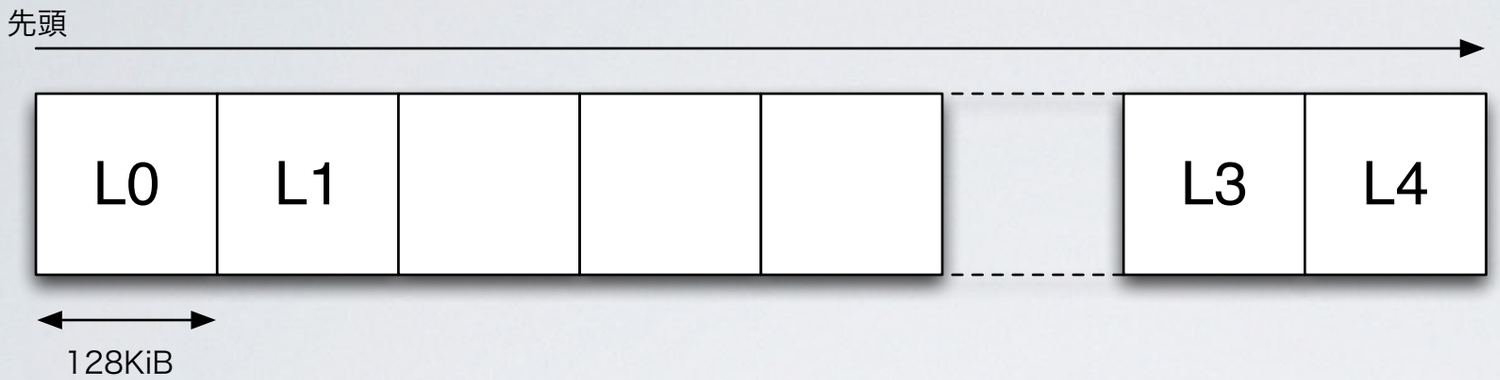
ZFS

- **コンセプト**
  - ZPOOLで記憶装置を管理
  - ZFSデータセット1個＝ファイルシステム1個
    = dnode の集合体（厳密にはちょっと違うけれど...）
  - dnodeには種類がある
    - ZAPオブジェクト（name=value を格納する）
    - DSLデータセットオブジェクト（ファイルシステムを指す）
    - DSLディレクトリオブジェクト（ファイルシステムの集合体を指す）
    - inode がデータブロックにディレクトリ情報を入れていたのと同じく、
      dnode も役割を分けていろいろな機能を持たせている

ZFSの構造



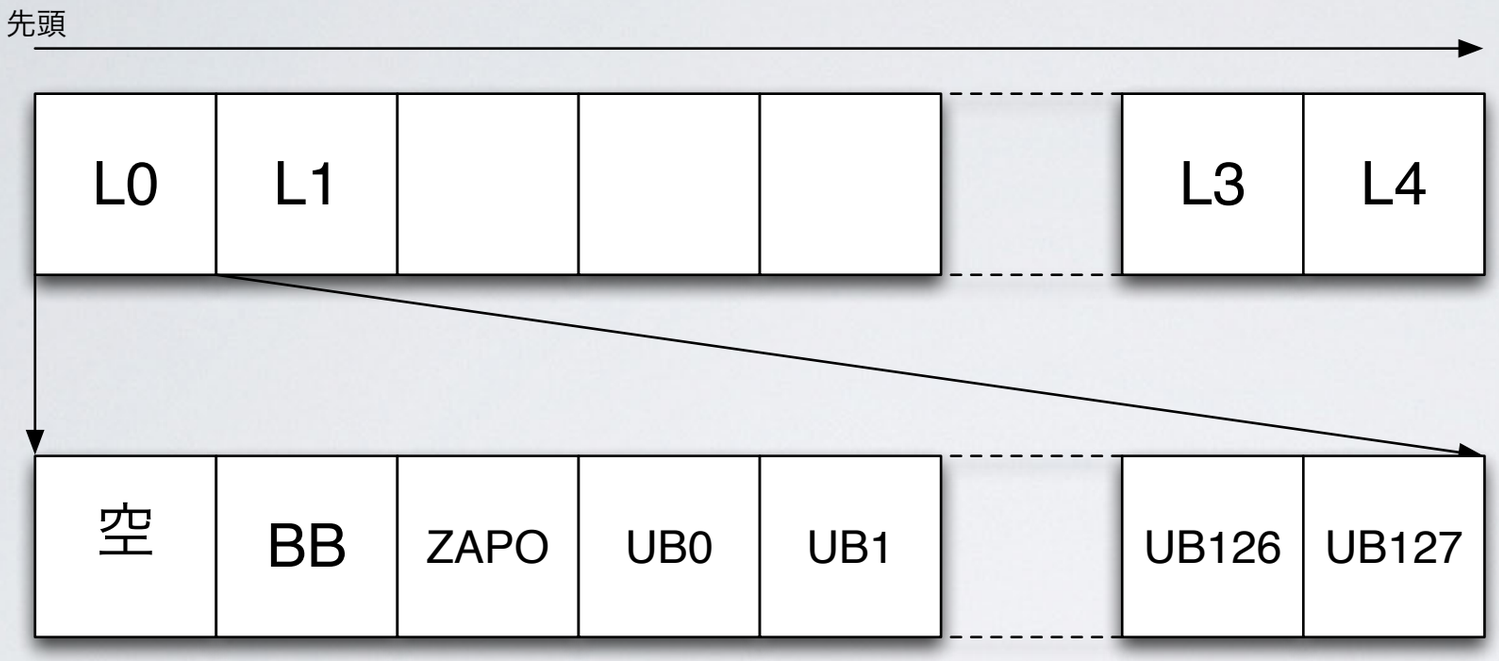
ZFSの構造



VDEVラベル（128KiB, 4個のコピー）

→ UFS の スーパブロックに相当

ZFSの構造



ZFSの構造

ラベルの中身は、

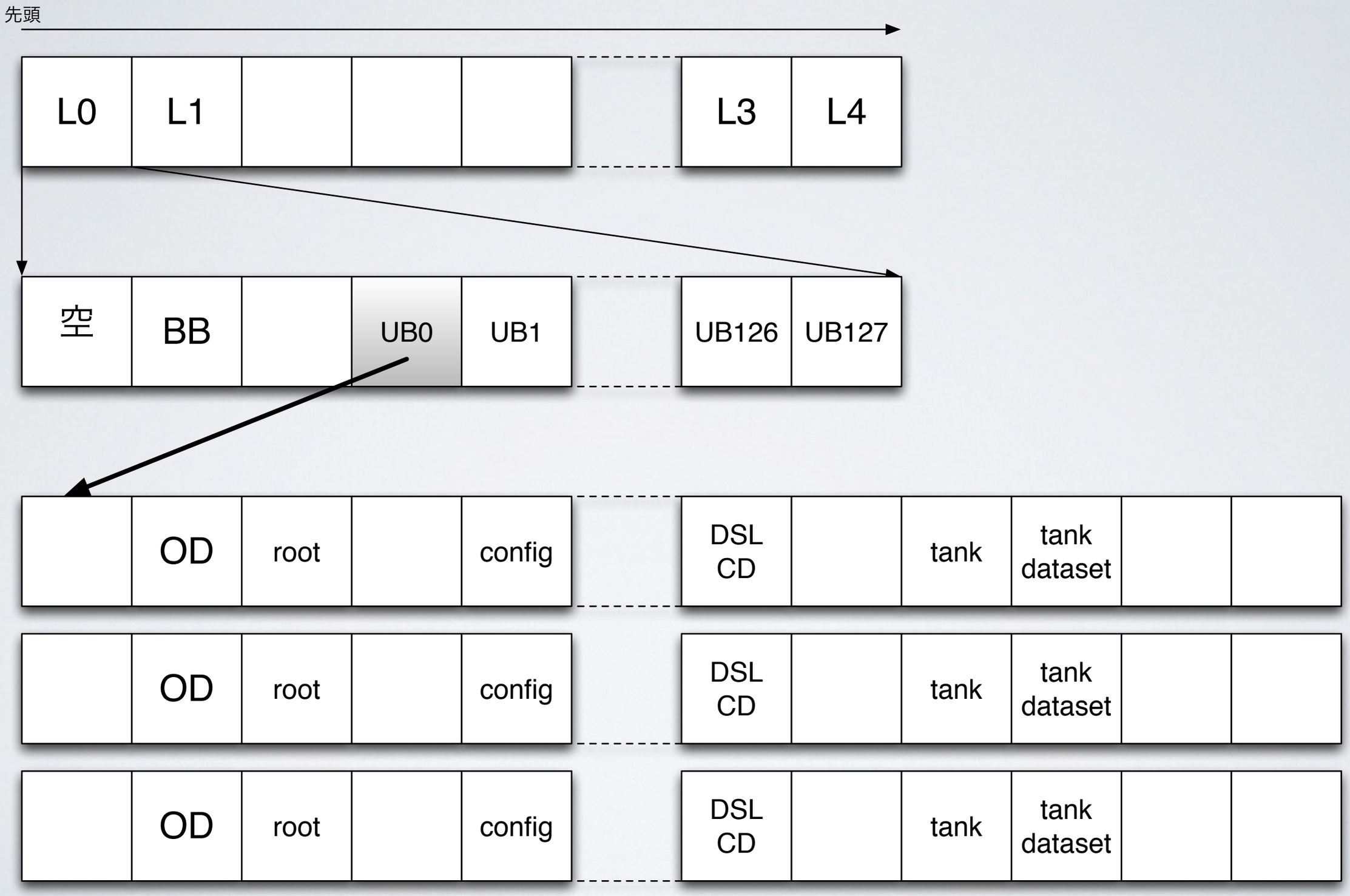
①起動ブロック

②ZAPオブジェクト

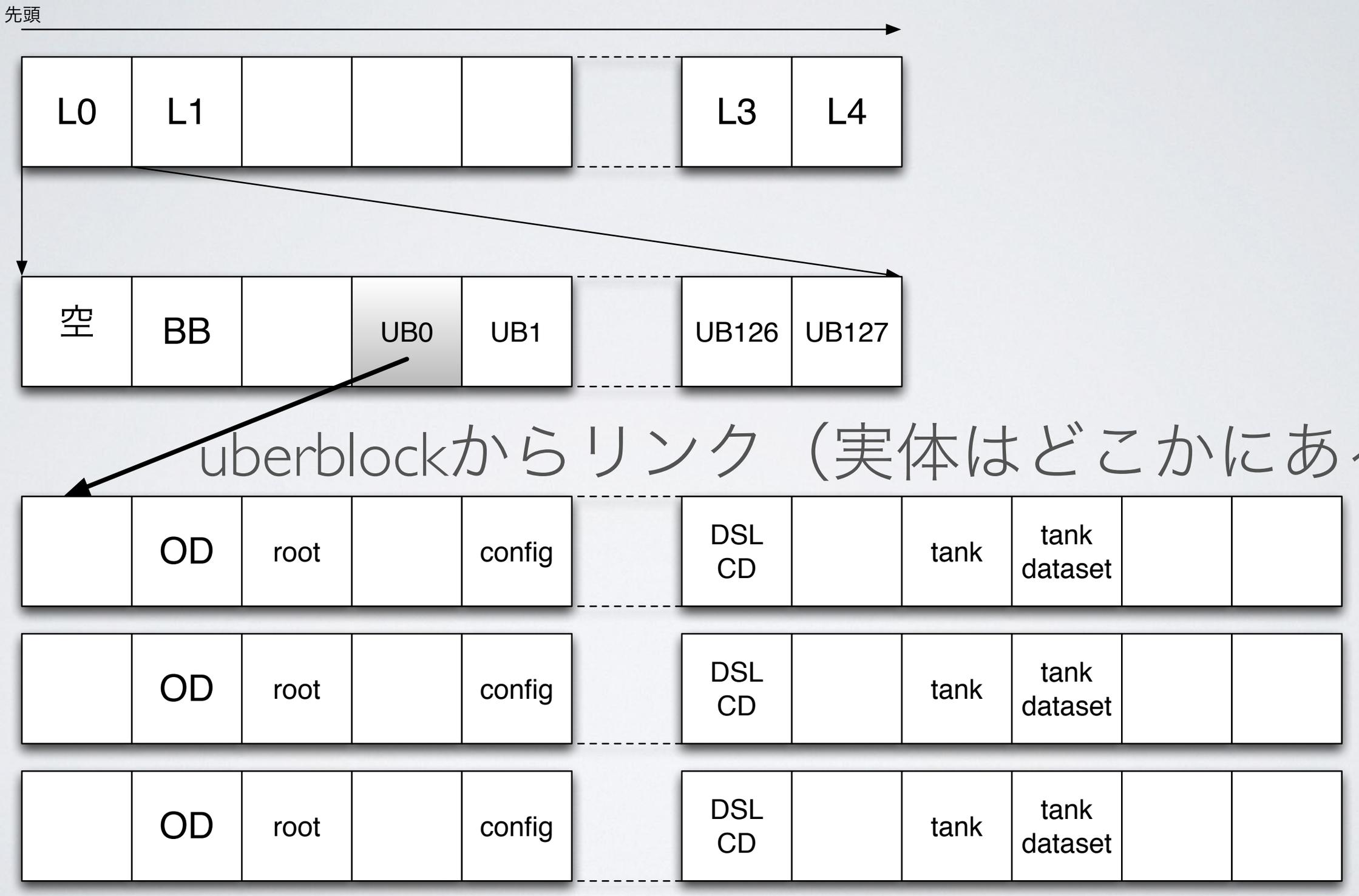
　(name=value を保存できる領域)

③uberblockの配列

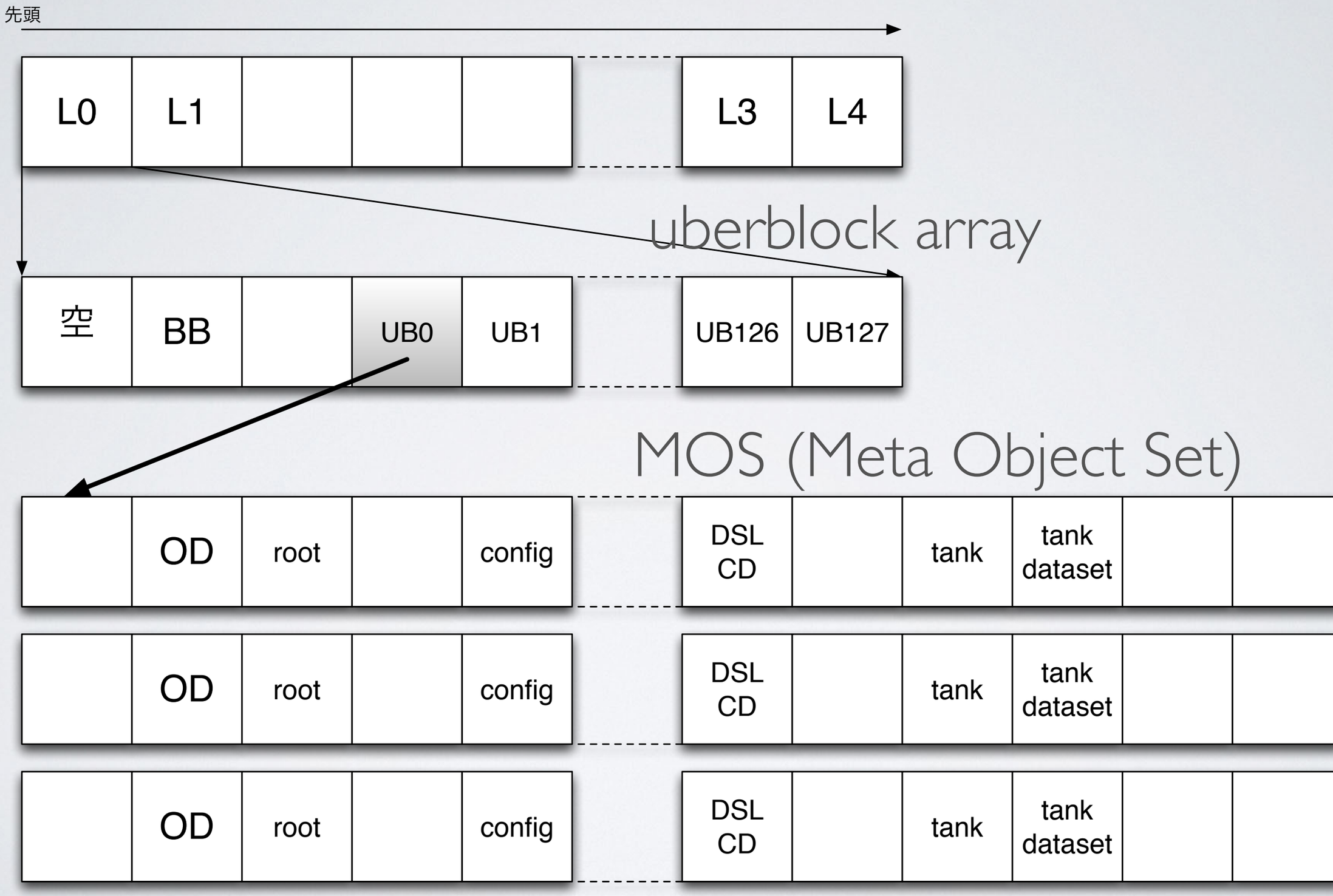
ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset

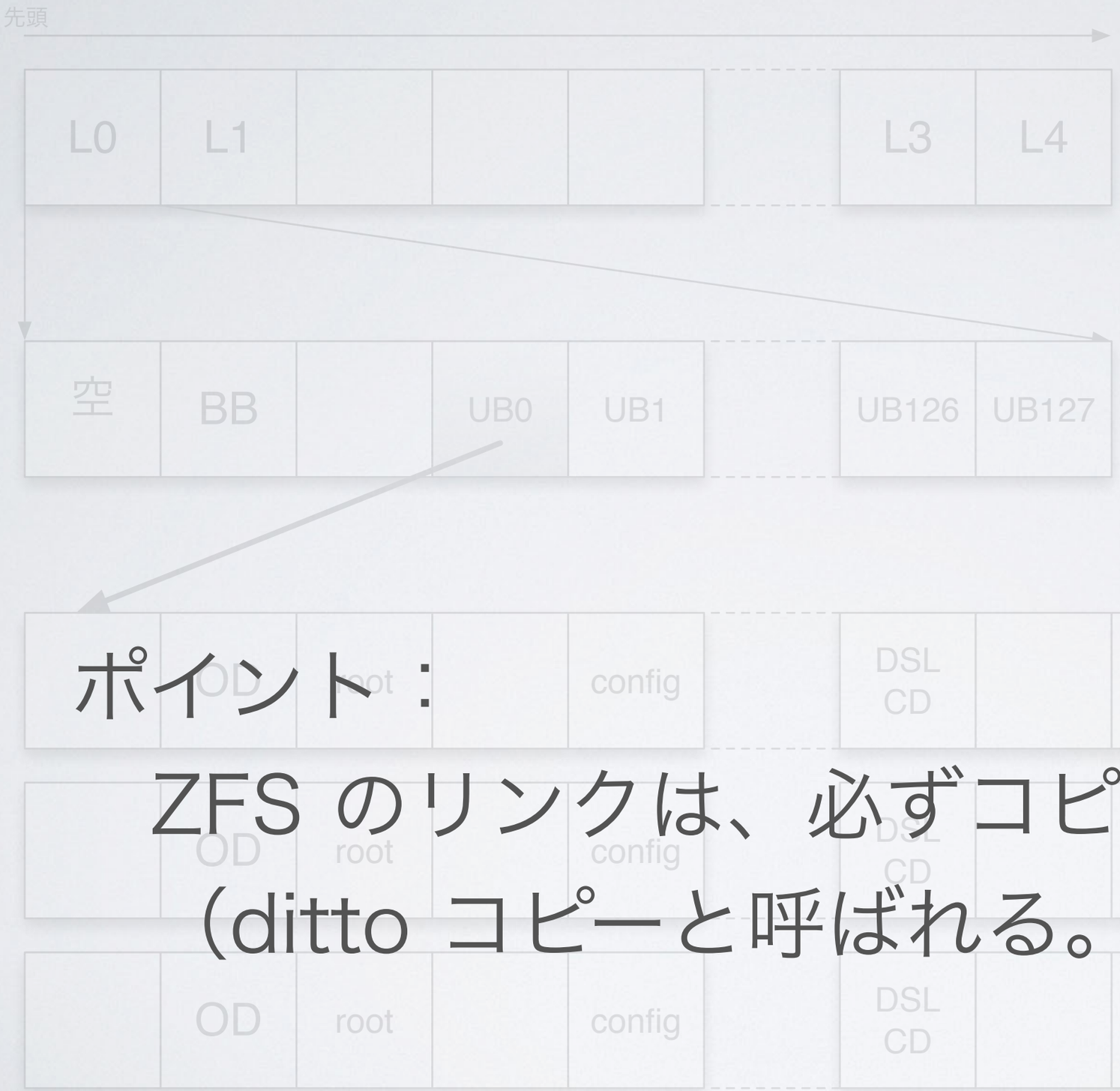
ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
uberblockからリンク（実体はどこかにある）
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset
OD
root
config
DSL
CD
tank
tank
dataset

ZFSの構造


先頭
L0
L1
L3
L4
uberblock array
空
BB
UB0
UB1
UB126
UB127
MOS (Meta Object Set)
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset

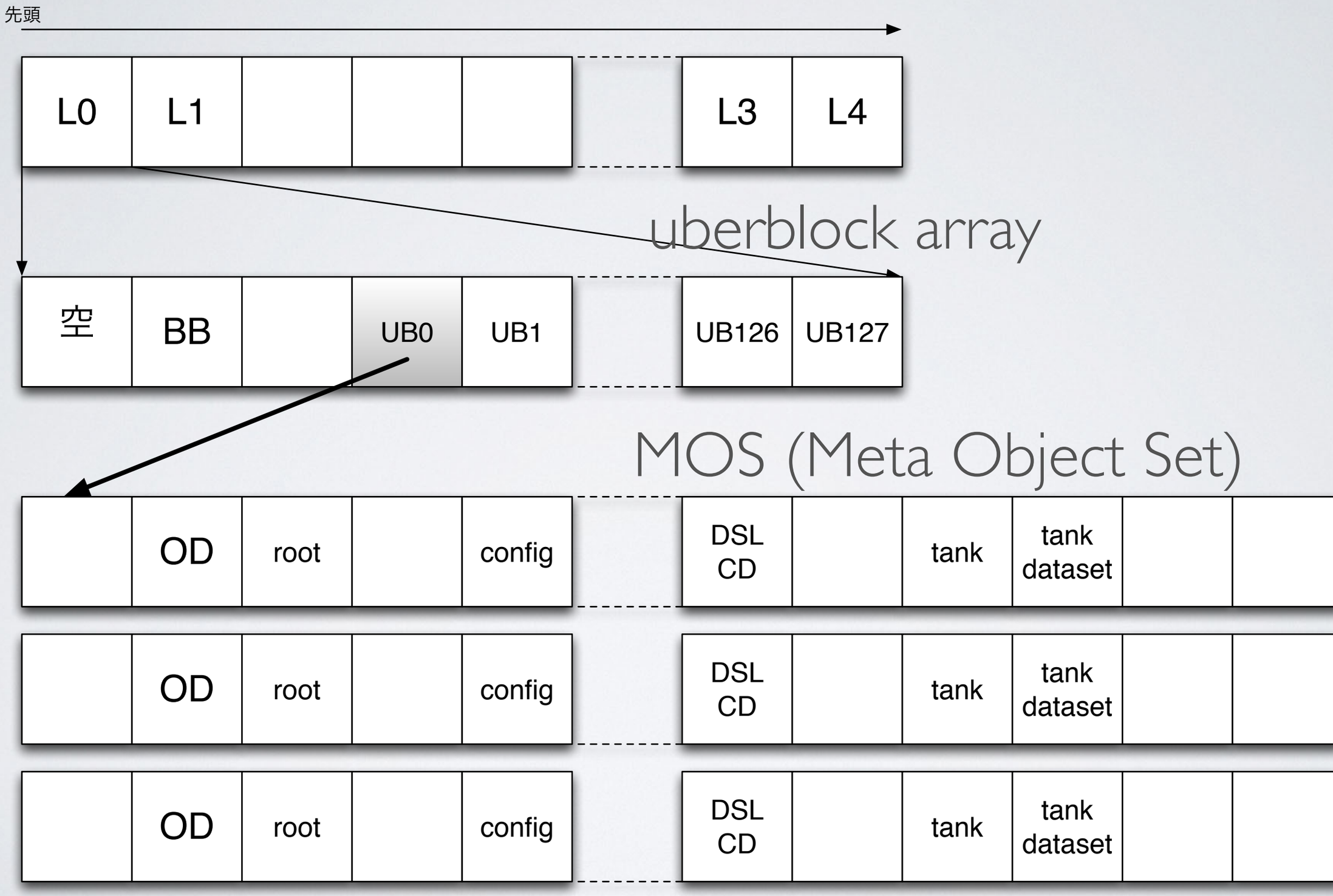
ZFSの構造



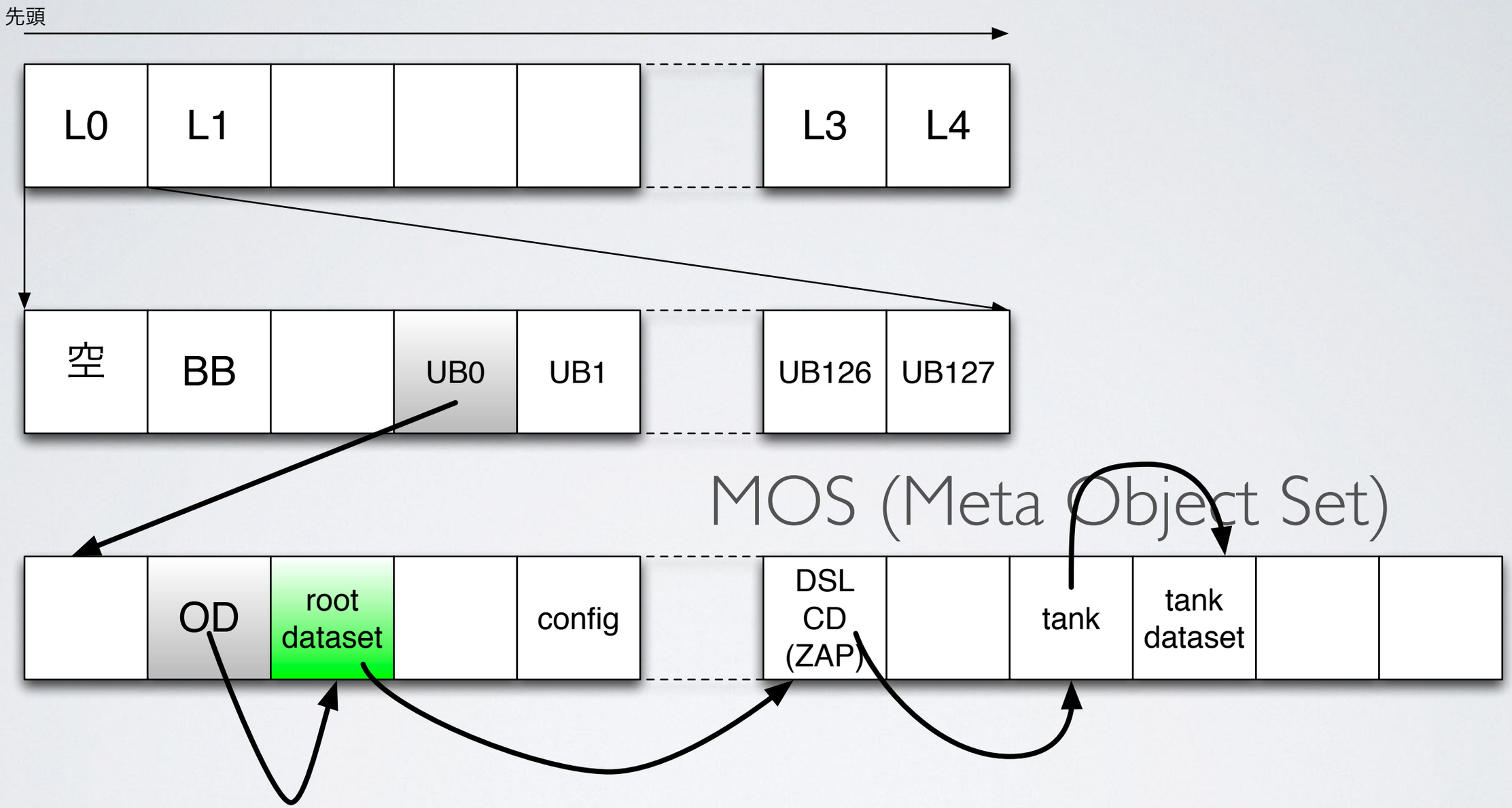
ポイント：

ZFS のリンクは、必ずコピーが作成される
（ditto コピーと呼ばれる。最大3個）

ZFSの構造


先頭
L0
L1
L3
L4
uberblock array
空
BB
UB0
UB1
UB126
UB127
MOS (Meta Object Set)
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset
OD
root
config
DSL CD
tank
tank dataset

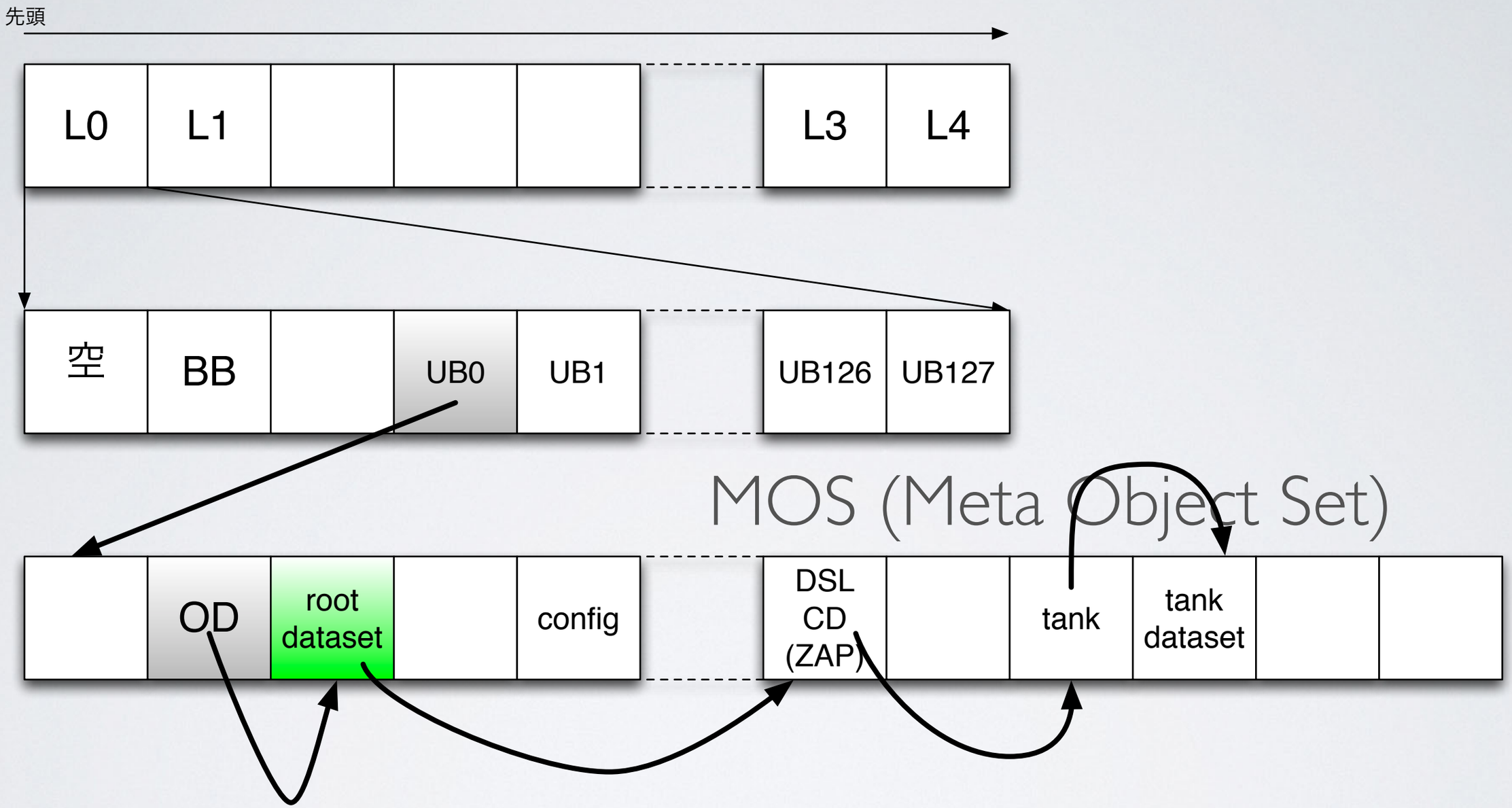
ZFSの構造



MOSの実体は、dnode が並んでいる配列構造

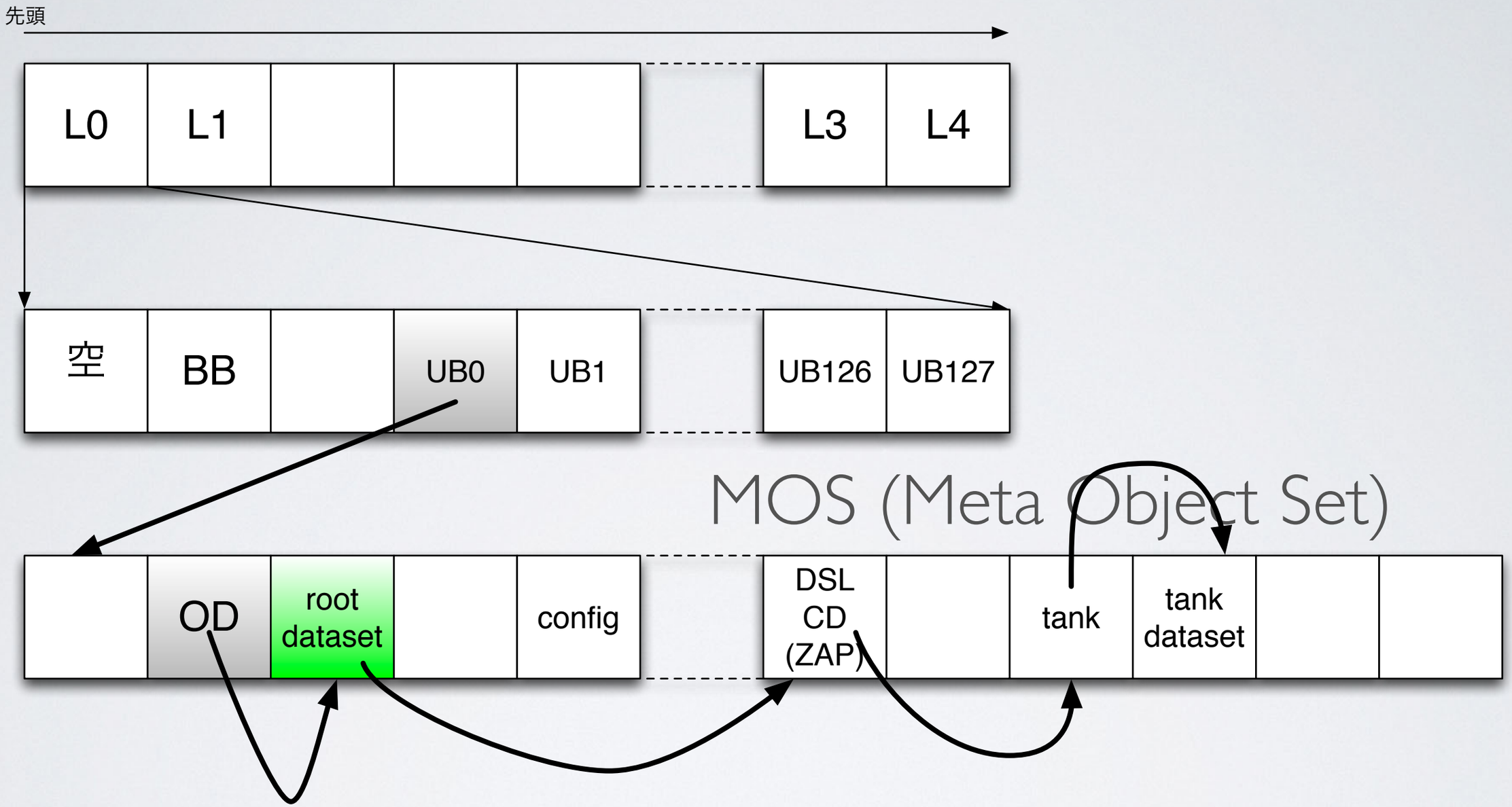
（＝UFSで言うinode領域と同じ）

ZFSの構造



ただし、MOSにはZPOOLに含まれるファイルではなく、
ZFSデータセット（＝ファイルシステム）の情報が格納される

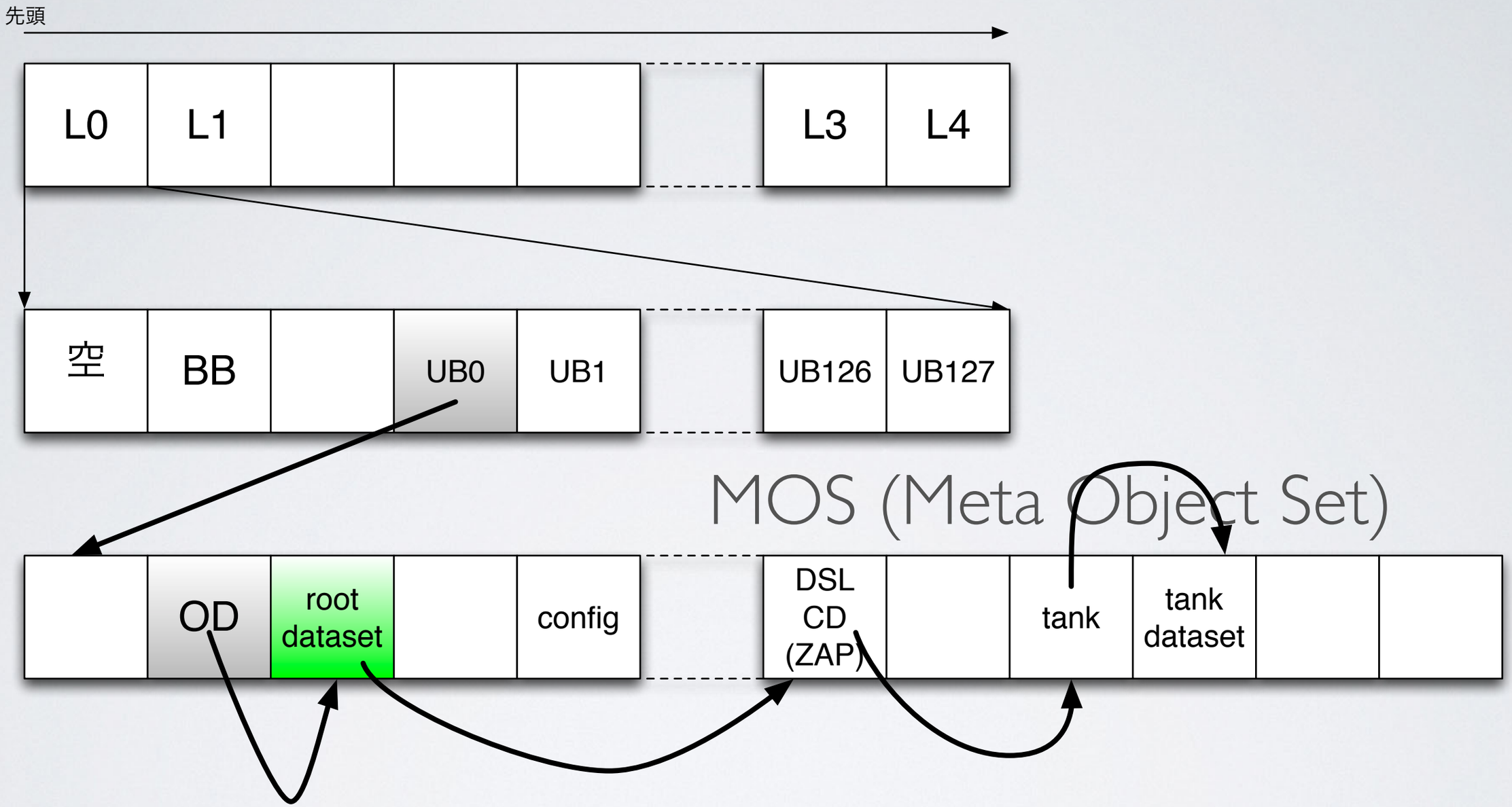
ZFSの構造



ルートデータセット：ZPOOLに必ずあるファイルシステム

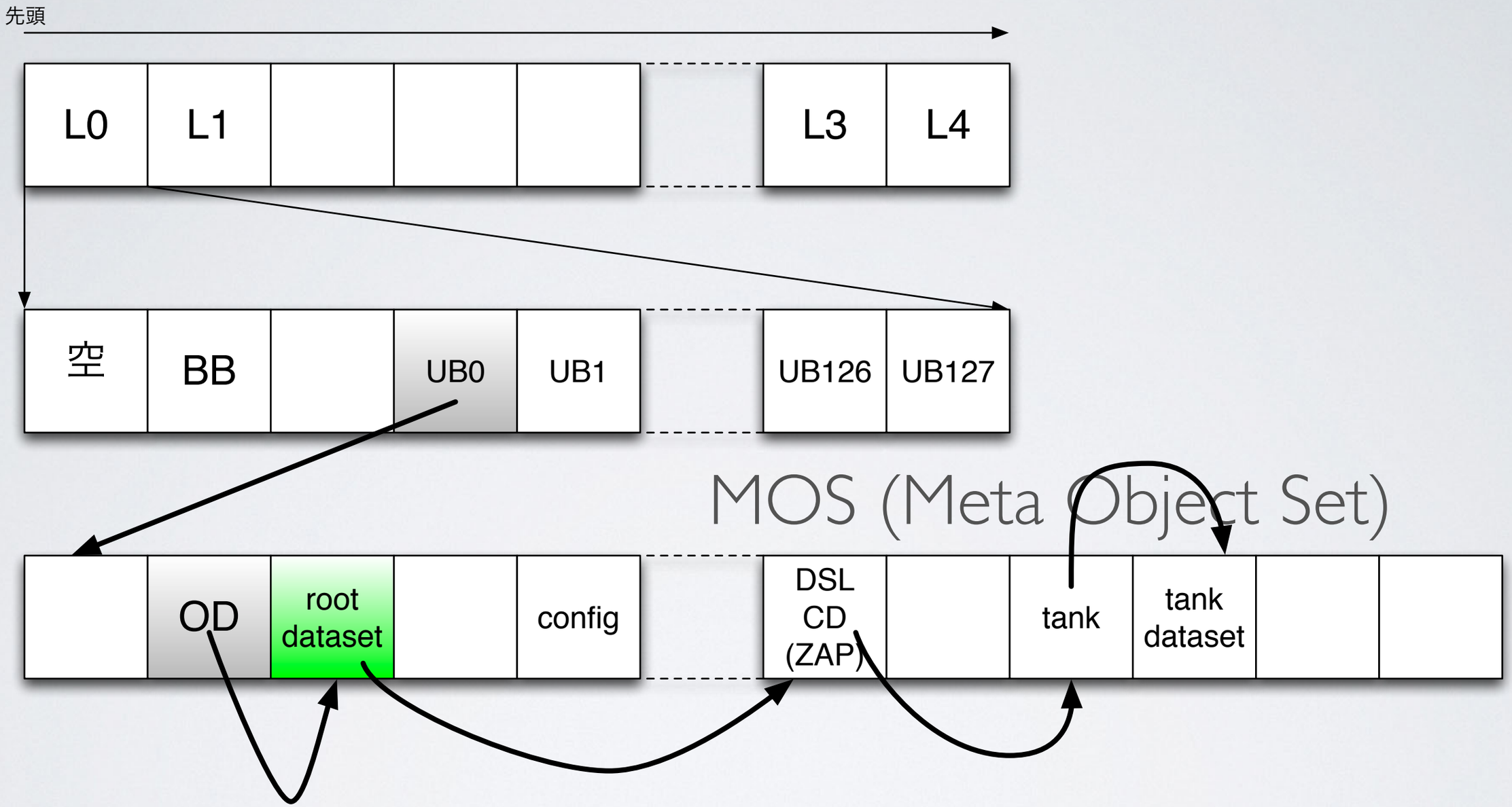
（ちなみにdnode 番号は必ず２）

ZFSの構造



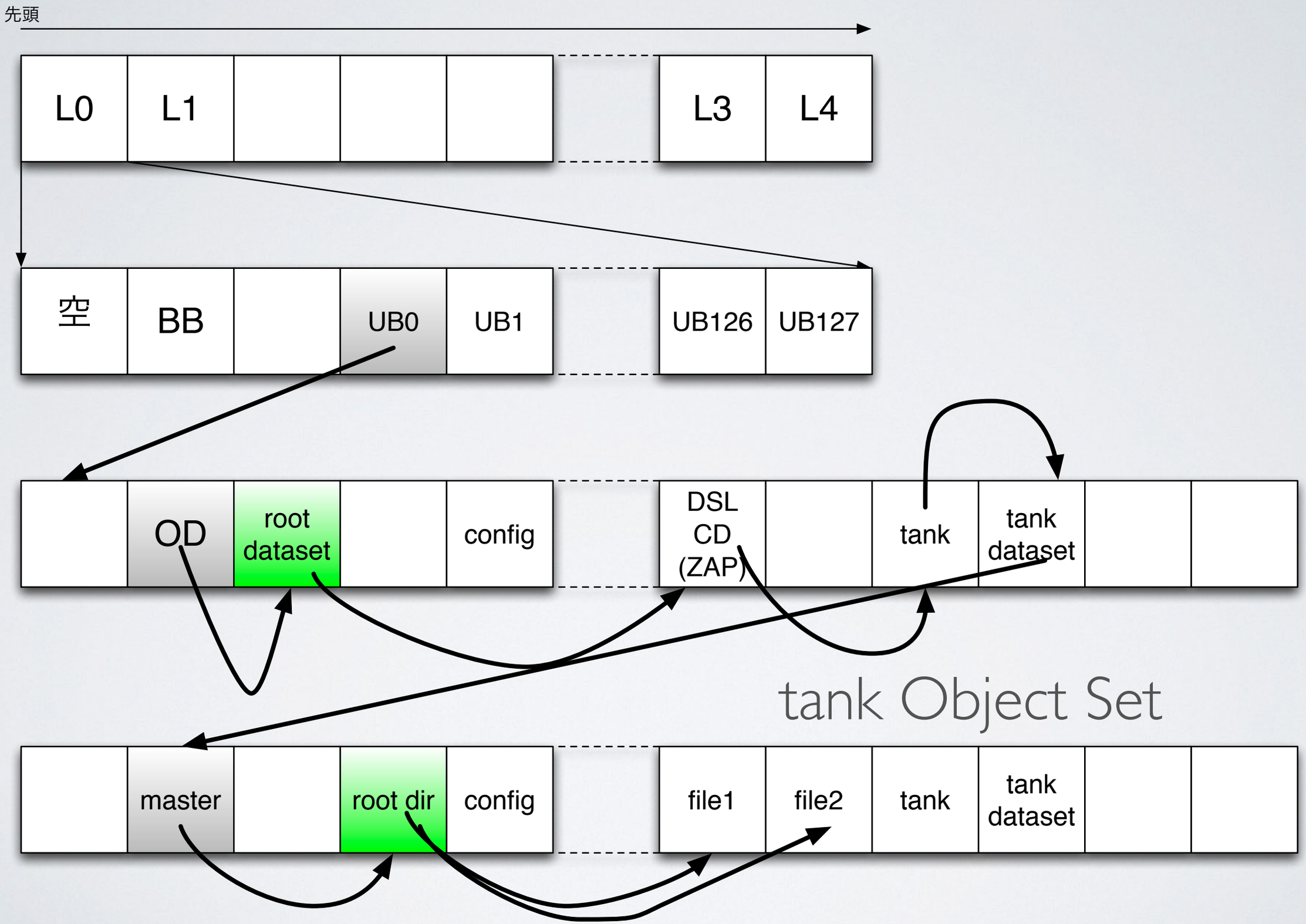
tank という名前のデータセットが登録されている

ZFSの構造

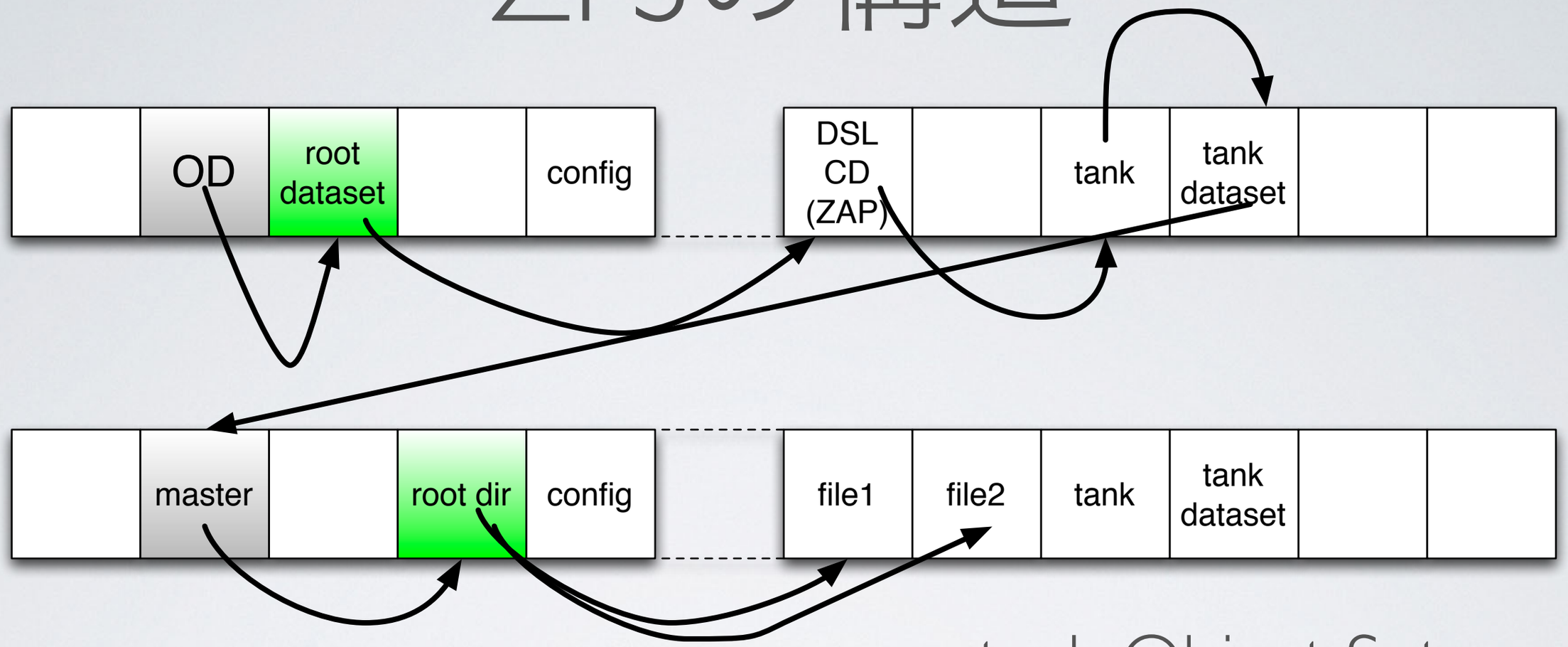


tank dataset にデータセットの中身の情報が。

ZFSの構造


先頭
L0
L1
L3
L4
空
BB
UB0
UB1
UB126
UB127
OD
root dataset
config
DSL CD (ZAP)
tank
tank dataset
tank Object Set
master
root dir
config
file1
file2
tank
tank dataset

ZFSの構造



tank Object Set

tank dataset は、もうひとつのdnode 配列を指している。

このdnode配列は、UFSのinodeとほぼ同じ機能

= tank dataset がスーパーノード

= object set 配列が inode (ルートディレクトリはdnode=3)

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

- **データセット**＝ファイルシステム（だけではないが）相当
- **uberblock** =データセットのスーパブロック
- **データセットオブジェクト**
  ＝ファイルシステムのスーパブロック

ZFSの構造

- **つまり**
  - ファイルシステムを inode っぽい構造で管理
  - ファイルとディレクトリを、inode っぽい構造で管理

という2段構えになっている

（簡単のため中間ブロックをいくつか飛ばして説明しています）

- **データセット**＝ファイルシステム（だけではないが）相当
- **uberblock** =データセットのスーパブロック
- **データセットオブジェクト**
  ＝ファイルシステムのスーパブロック
- 発想的には、今までのファイルシステムの技術を
  素直に拡張した形。**複数の記憶装置にまたがって構成可能**

ZFSのデータ処理

- **ブロックの処理**
  - ブロックへのリンクは、最大3重コピー
  - ブロックのチェックサムを記録できる
  - ブロックが壊れていたら、正常なコピーを使う

ZFSのデータ処理

- **ブロックの処理**
  - ブロックへのリンクは、最大3重コピー
  - ブロックのチェックサムを記録できる
  - ブロックが壊れていたら、正常なコピーを使う
- **書き込みは copy-on-write + トランザクション**
  - 書き換えないでコピーする
  - リンクは下から書き換える（SoftUpdates と同じ）
  - 一番上は uberblock をずらす
  - I/O をトランザクショングループ(txg)に分けてコミット

ZFSのデータ処理

- **特徴的なのは**
  - ブロック単位でデータの破壊を検証できる
  - ほぼ必ずコピーがある
  - SoftUpdates のような仕組みが入っている
  - 書き込みが個々のリンクやブロックではなく、
    かなり大きいまとまり(txg)で発生する

ZFSのデータ処理

- **ZFS が苦手な処理**
  - UFS以上にリンクを辿りまくる
    ＝dnode アクセスが性能に敏感なのは同じ。
      複数台のディスクに分散させることでIOPSを抑える

ZFSのデータ処理

- **ZFS が苦手な処理**
  - UFS以上にリンクを辿りまくる
    ＝dnode アクセスが性能に敏感なのは同じ。
    複数台のディスクに分散させることでIOPSを抑える
  - COW は、予想外の負荷上昇を発生させる
    - 全部上書きする場合も、全部読んでコピーしてから！
    - 10GB のファイルの 1 バイトを変更するとどうなる？
    - zvol に dd で 1MiB 単位で連続データを
      書き込んだ後、1KiB 単位で連続データを書き込むと?
    - txg は、読む→書くのサイクルを分散させる工夫でもある

Tips

- amd64 で使いましょう
- vfs.zfs.vdev.cache.size は “0” に
- vfs.zfs.vdev.cache.bshift は、小さいI/O をまとめる設定
  デフォルトは 64KiB (16)
- メタデータは vfs.zfs.arc_meta_limit を設定して
  キャッシュを稼ぐ
  - おおよその計算法：データ量／(8KiB * 128B)
  - 1TB で 16GB くらい

ZFSの構築事例

- 業務で使ってるのは出しにくいので、
  allbsd.org のストレージサーバを生け贄に
  - Supermicro X8DTL (X5650 6 Core x 2 SMT)
  - 2TB SATA HDD 13 台, SAS2004 (mps(4)) 経由
  - ほぼ NFS のみでサービス
  - 負荷は FTP のようなシーケンシャルと
    CVSup のような fsync ばりばりなものが入り乱れ

おしまい

- 使い方とトラブルシュートの話は
  どこかでやりたい

- 質問はありますか？