

ZFSの性能測定とチューニング

佐藤 広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2016/11/29

自己紹介

- ▶ 名前：佐藤 広生
- ▶ FreeBSD src + ports + doc committer (2000-)
- ▶ FreeBSD コアチームメンバ、リリースエンジニア(2006-)
- ▶ FreeBSD Foundation 理事(2008-)
- ▶ その他の*BSD/オープンソース関連の活動いろいろ
- ▶ 東京工業大学助教(2009-)

お話しすること

- ▶ ZFSを使ってみた/ているけれど...
 - ▶ キャッシュ容量はどう決めれば良い？
 - ▶ ハードウェアの性能限界が出せていない気がするけれど、性能ってどう調べれば良いの？
 - ▶ なんかUFSの方が性能的にマシだったような...
- ▶ ZFSの構造と性能の調べ方を知しましょう
- ▶ FreeBSDに限らず使えます
(ただしOracle Solarisを除く)

お話しすること

- ▶ システム管理者が知っておくべき
 - ▶ ZFSの構造
 - ▶ DTraceを使ったボトルネック分析
 - ▶ チューニングパラメータ

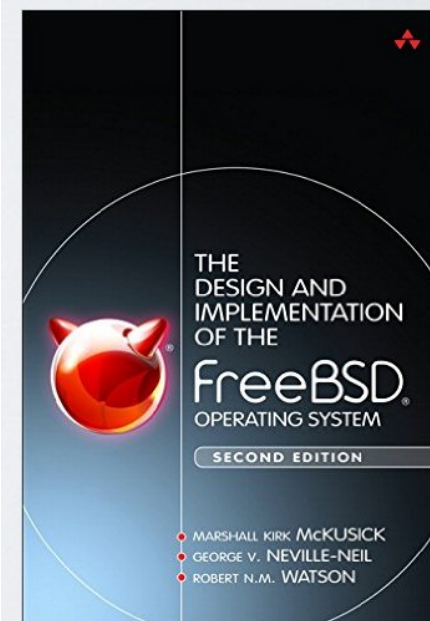
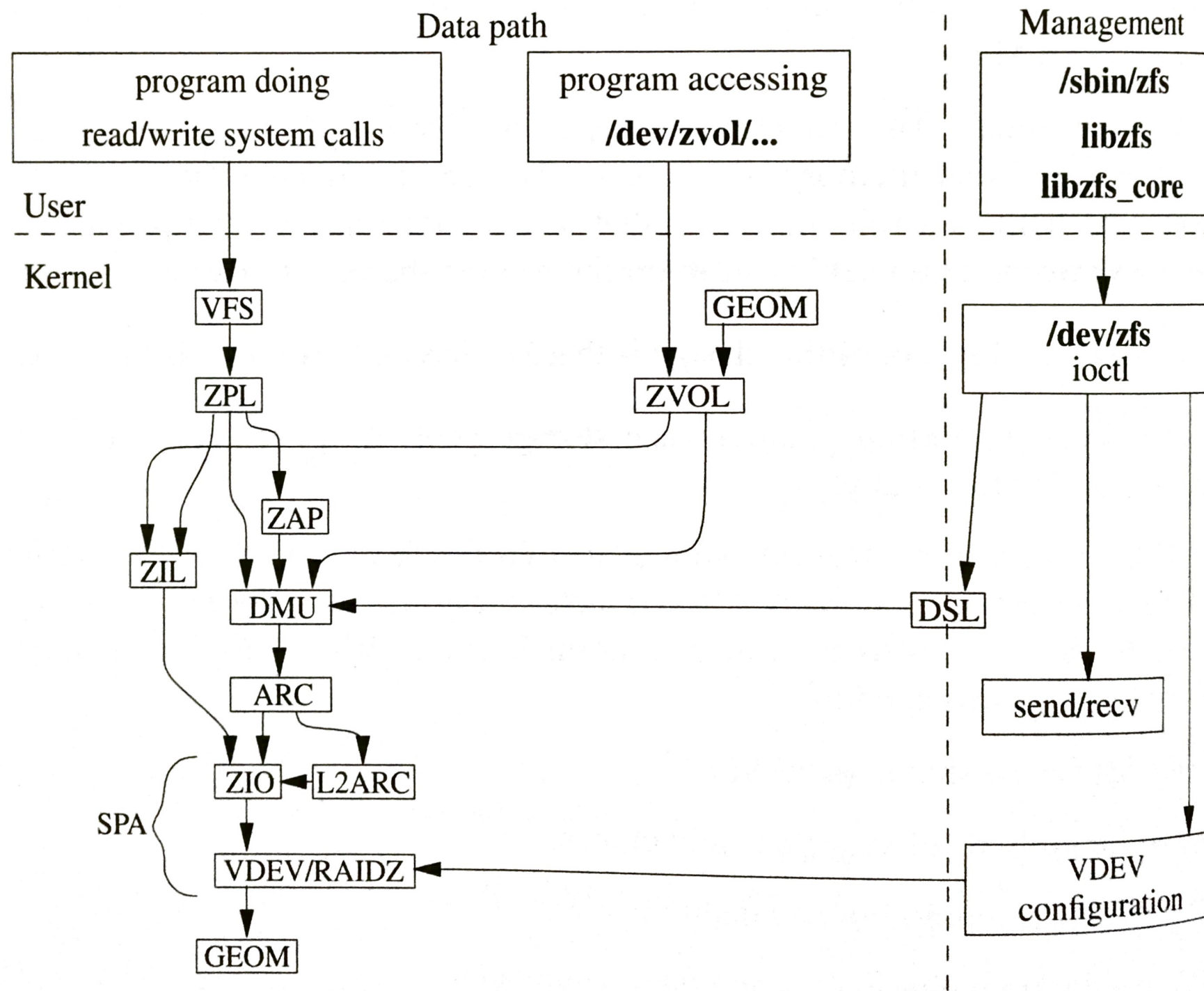
...を解説します。

- ▶ DTraceを使うので、チュートリアル資料を
スライドの後ろの方に付けてあります

ZFSの構造

ZFSの構造

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.



ZFSの構造

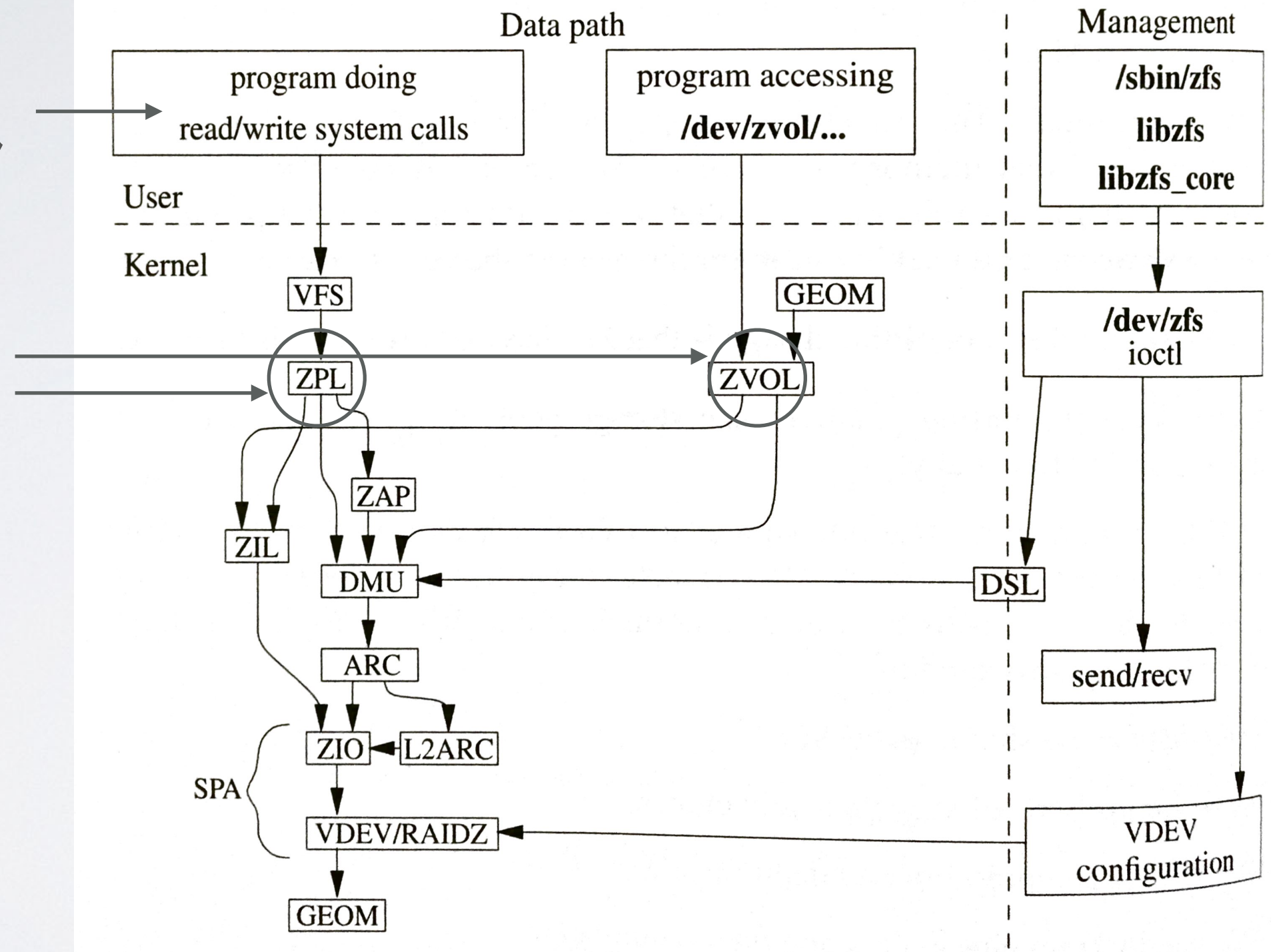
- ▶ 理解しよう：「ZFSのデータ処理は3層構造」
 - ▶ 1層目：POSIX API (znode)
 - ▶ 2層目：DMU (dnode)
 - ▶ 3層目：SPA (ブロックポインタ)
- ▶ 参考：UFSはvnode, inode, ブロックポインタ
- ▶ 「dnodeが複数のデバイスにまたがる点」が異なる

ZFSの構造

ユーザランド
アプリケーション

ZFSの入り口
(2つある)

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.

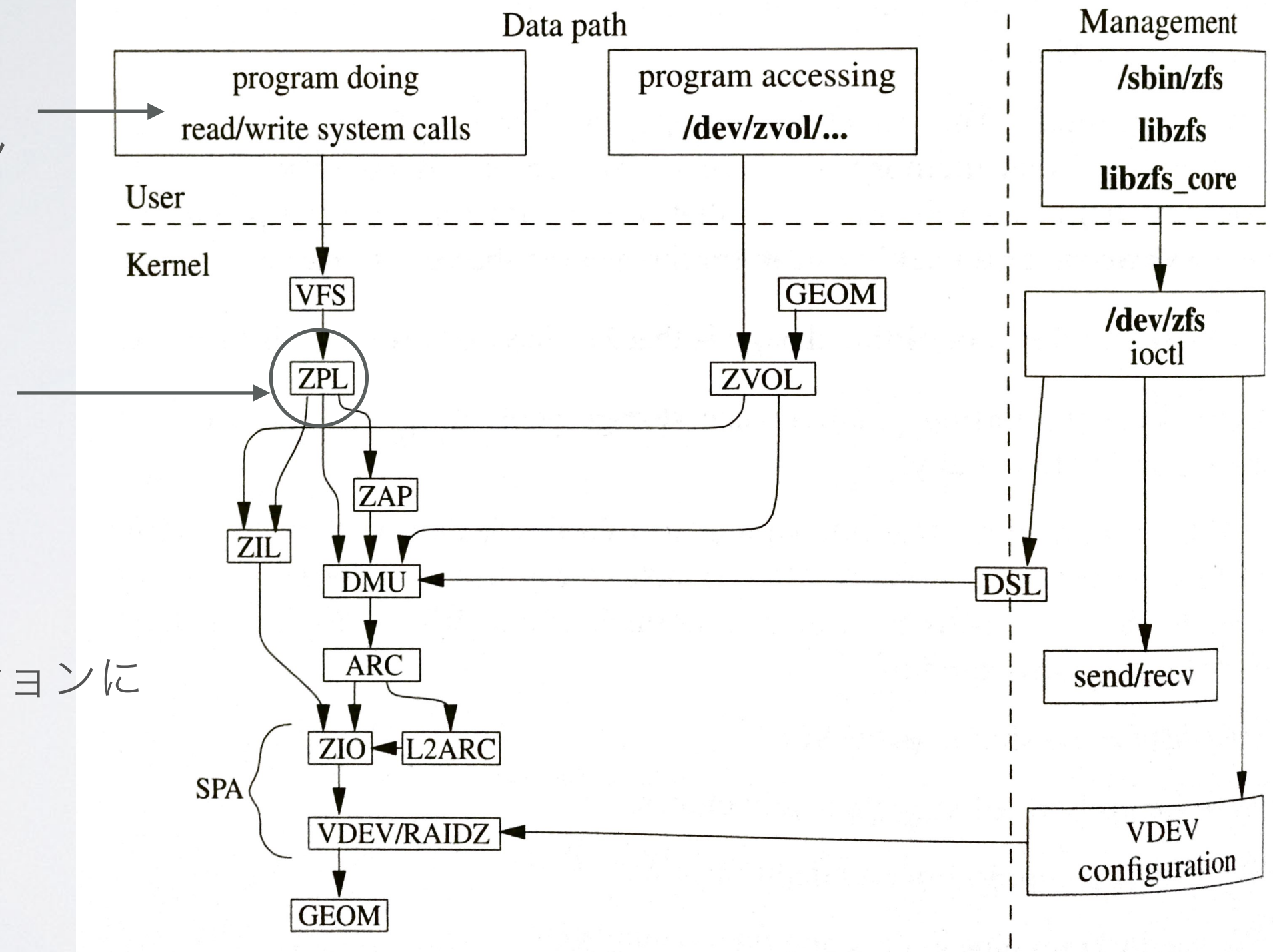


ZFSの構造

ユーザランド
アプリケーション

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.

ZPL
POSIXの定義する
「ファイル」や
「ディレクトリ」の
構造をアプリケーションに
見せる処理をする。



ZFSの構造

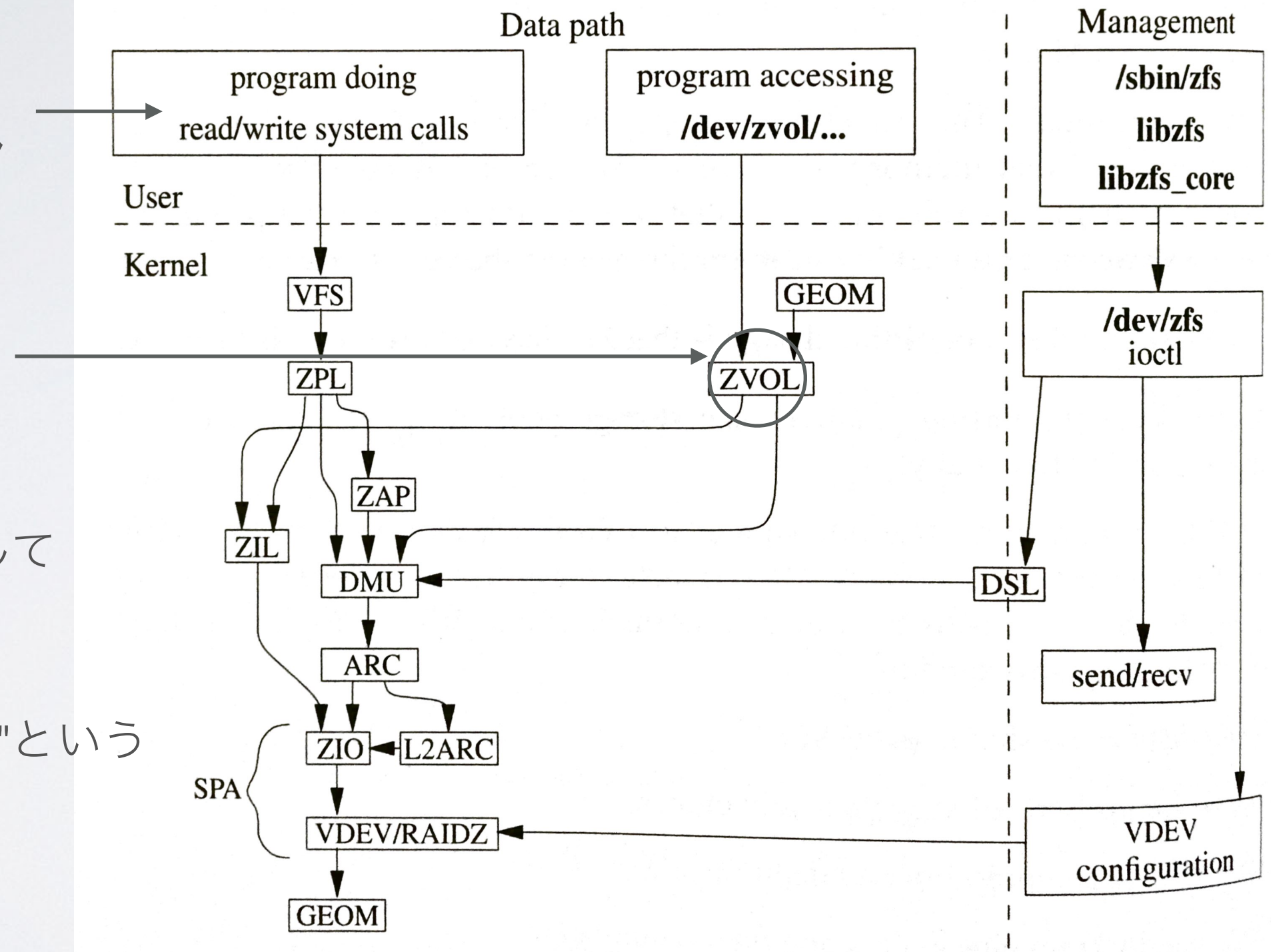
ユーザランド
アプリケーション

ZVOL

/dev/da0 のように
ブロックデバイスとして
見える姿を処理する。

実際には"/dev/zvol/xxx"という
名前で見える。

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.

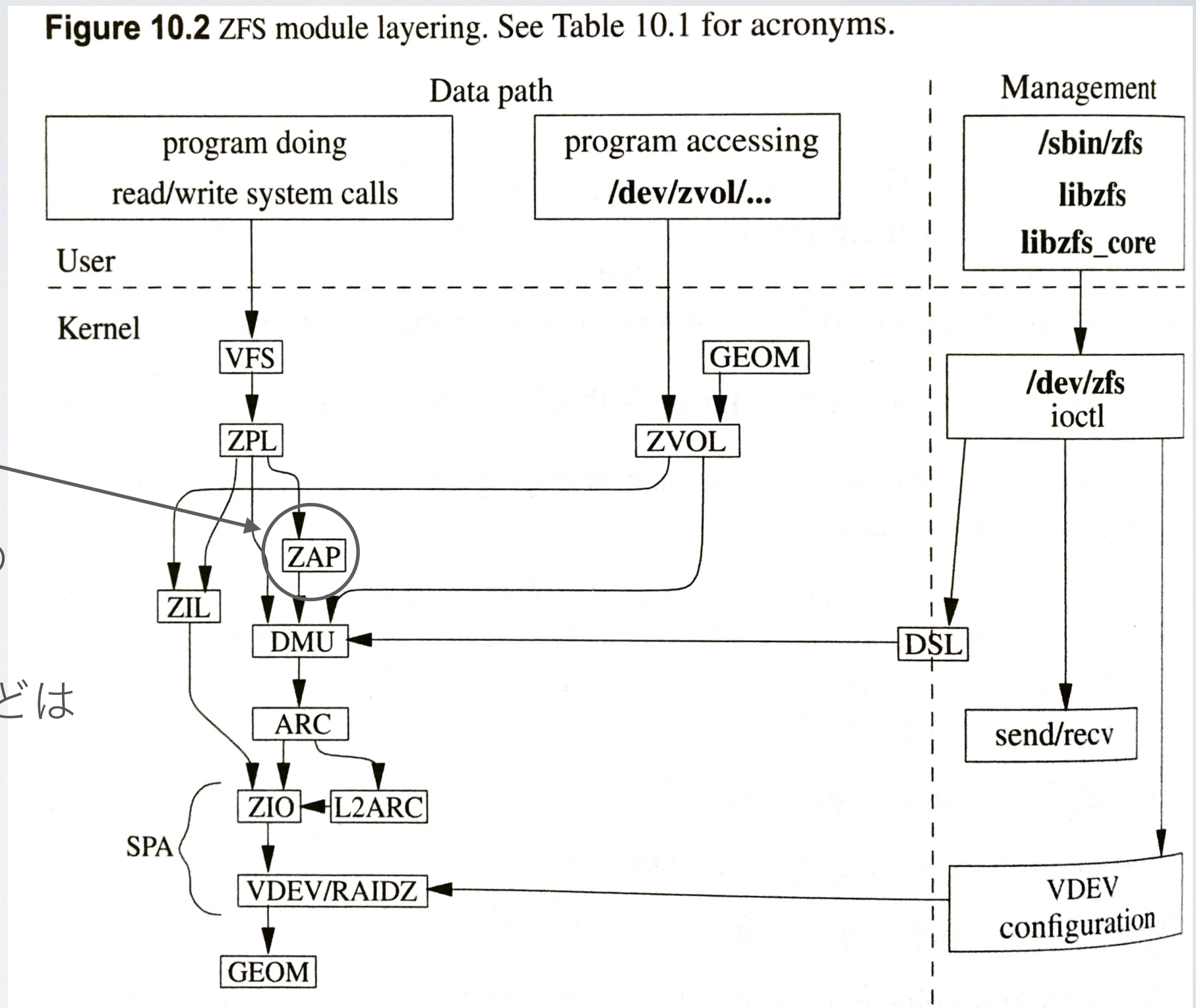


ZFSの構造

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.

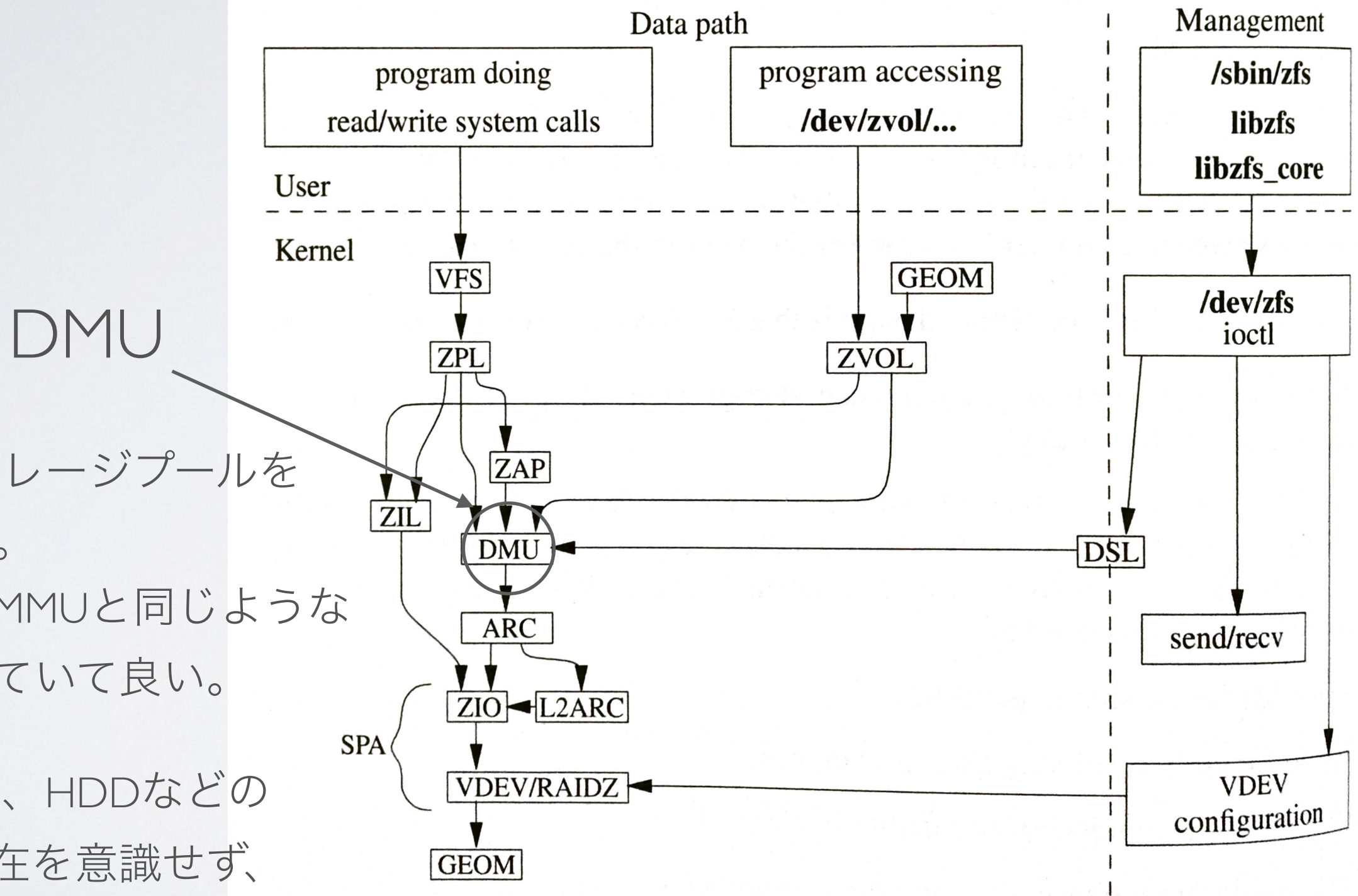
ZAP メタデータ処理

ZAPは、key-valueペアの
処理を担うところ。
ディレクトリや属性などは
ここで処理される。



ZFSの構造

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.



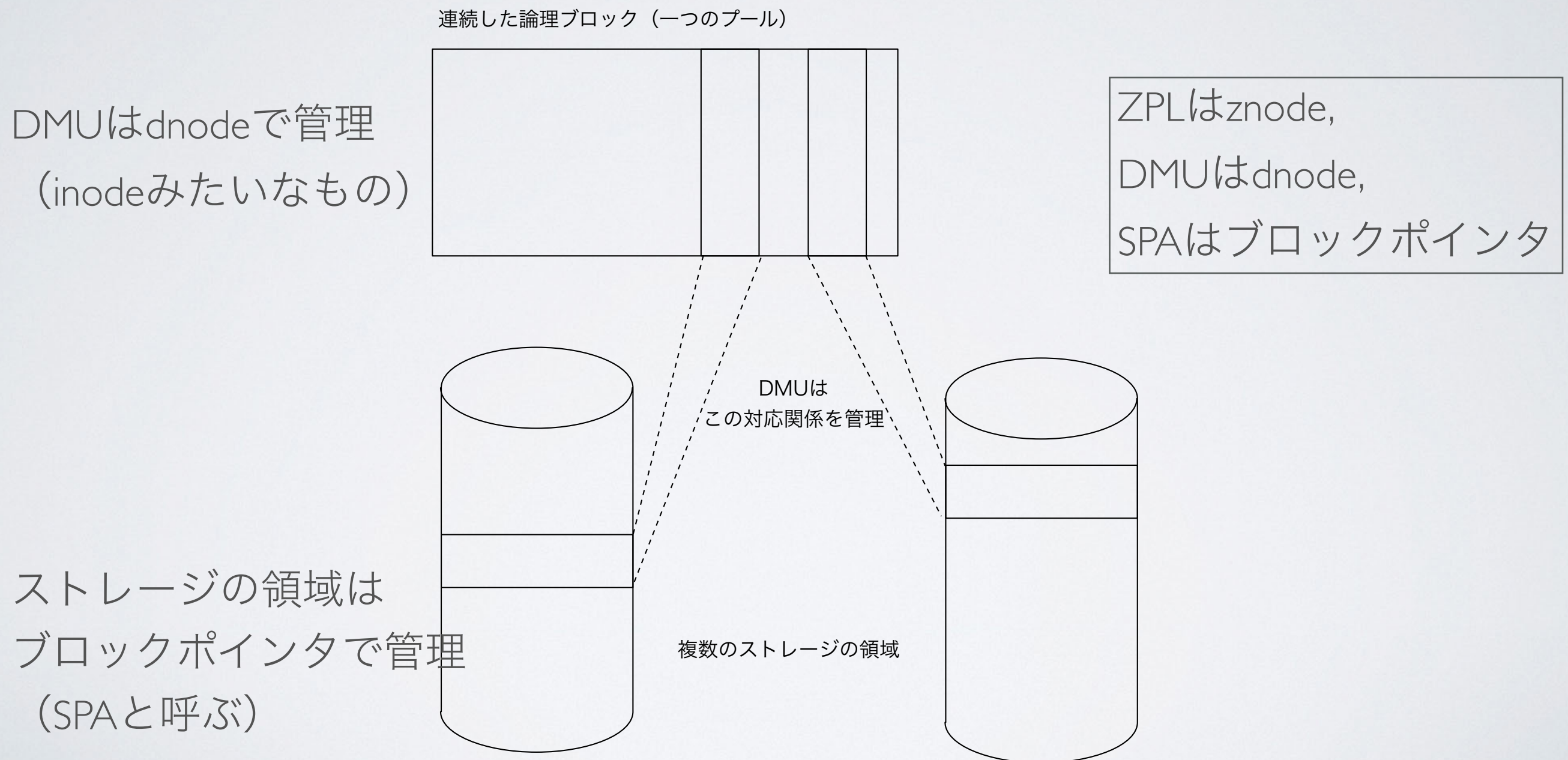
DMU

DMUは、ストレージプールを管理する部分。
仮想メモリのMMUと同じような発想だと思っている良い。

DMUの上側は、HDDなどのデバイスの存在を意識せず、データはプールの論理ブロック番号で管理される。

ZFSの構造

- ▶ DMUが管理している領域のイメージ
= 複数の領域を集約して、上位層に対して連続領域に見せる

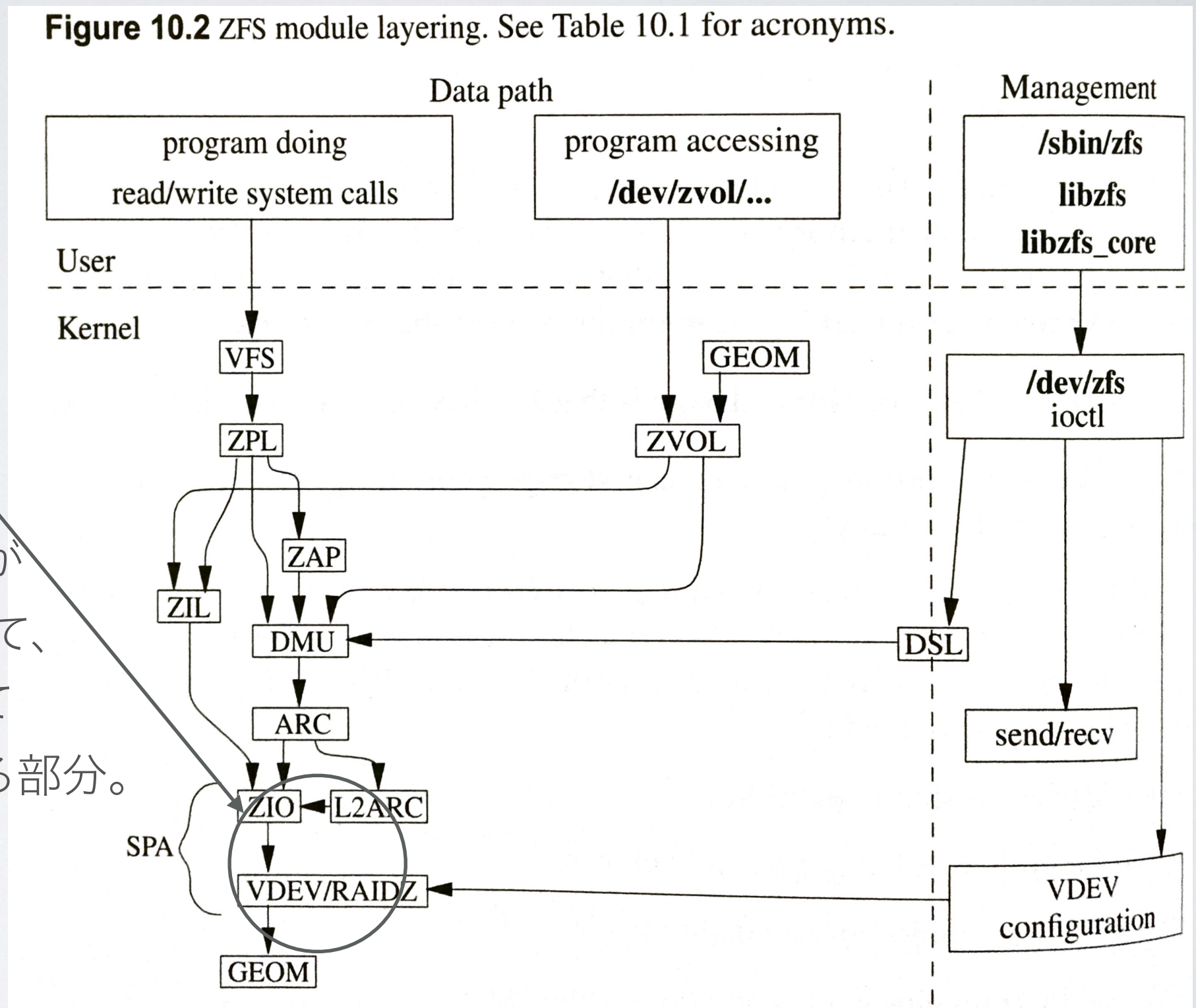


ZFSの構造

Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.

SPA

SPAは、DMUで読み書きが必要になった領域に対して、それを disk I/O に翻訳して I/O リクエストを発行する部分。



ZFSの動作の特徴

- ▶ アプリケーションからのファイルアクセスは、DMUを通過してディスクのI/O要求に翻訳される
- ▶ DMUは、「連続する論理ブロック (=zpool)」をメモリ空間に割り当てる。
☆アプリケーションがzpool領域を操作 = メモリの操作
- ▶ メモリ空間への操作は、SPAによってdisk I/O に変換
(キャッシュにデータがあればI/Oなしでデータを送る)
- ▶ 仮想メモリっぽい動作 (でかいmmap()のようなもの)

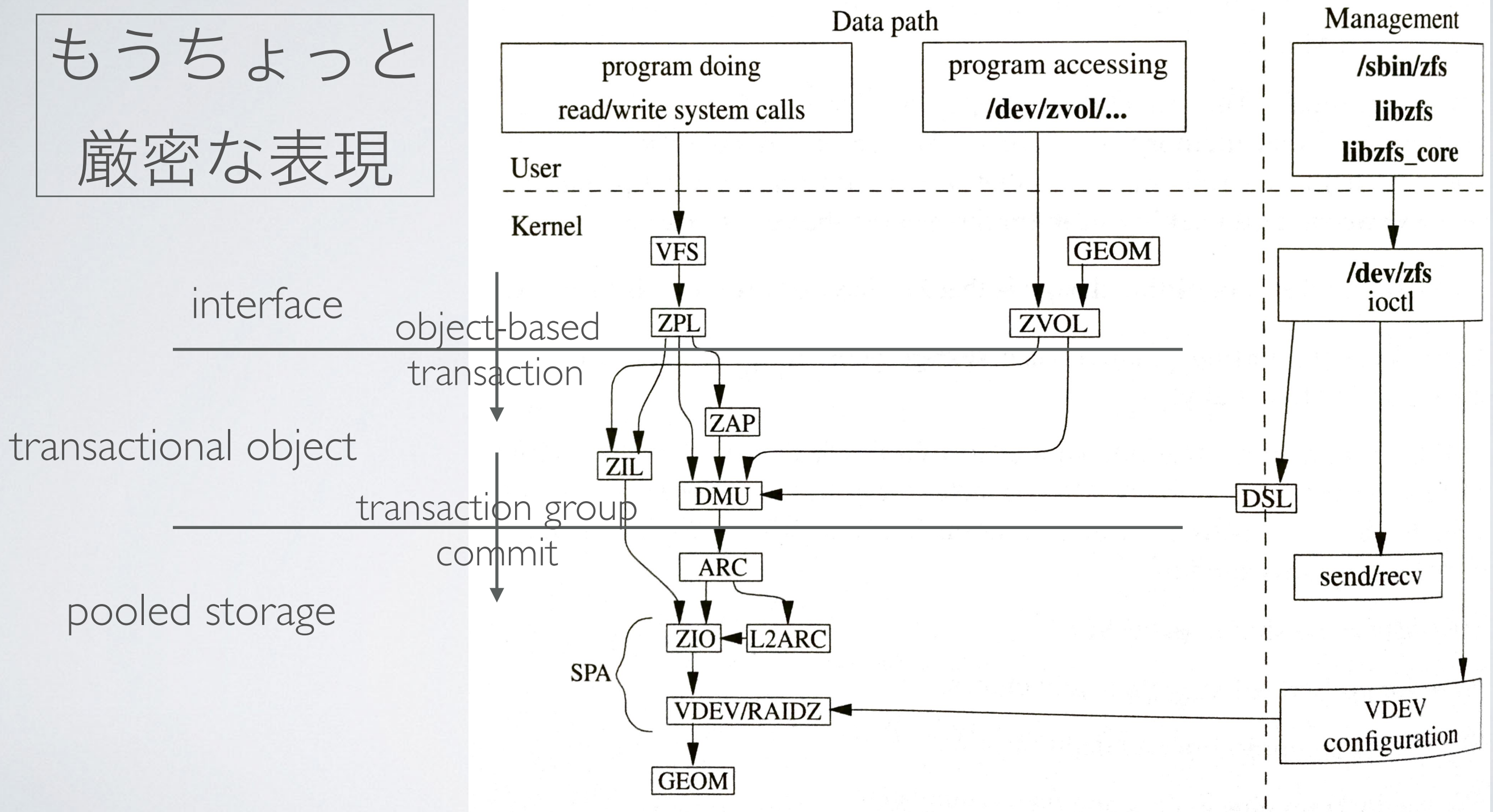
ZFSの構造

- ▶ 理解しよう：「ZFSのデータ処理は3層構造」
 - ▶ 1層目：POSIX API (znode)
 - ▶ 2層目：DMU (dnode)
 - ▶ 3層目：SPA (ブロックポインタ)
- ▶ アプリケーションからのアクセスが
ディスク入出力に反映されるまでの間、
データはDMU・SPAという2つの層を流れる

ZFSの構造

もうちょっと
厳密な表現

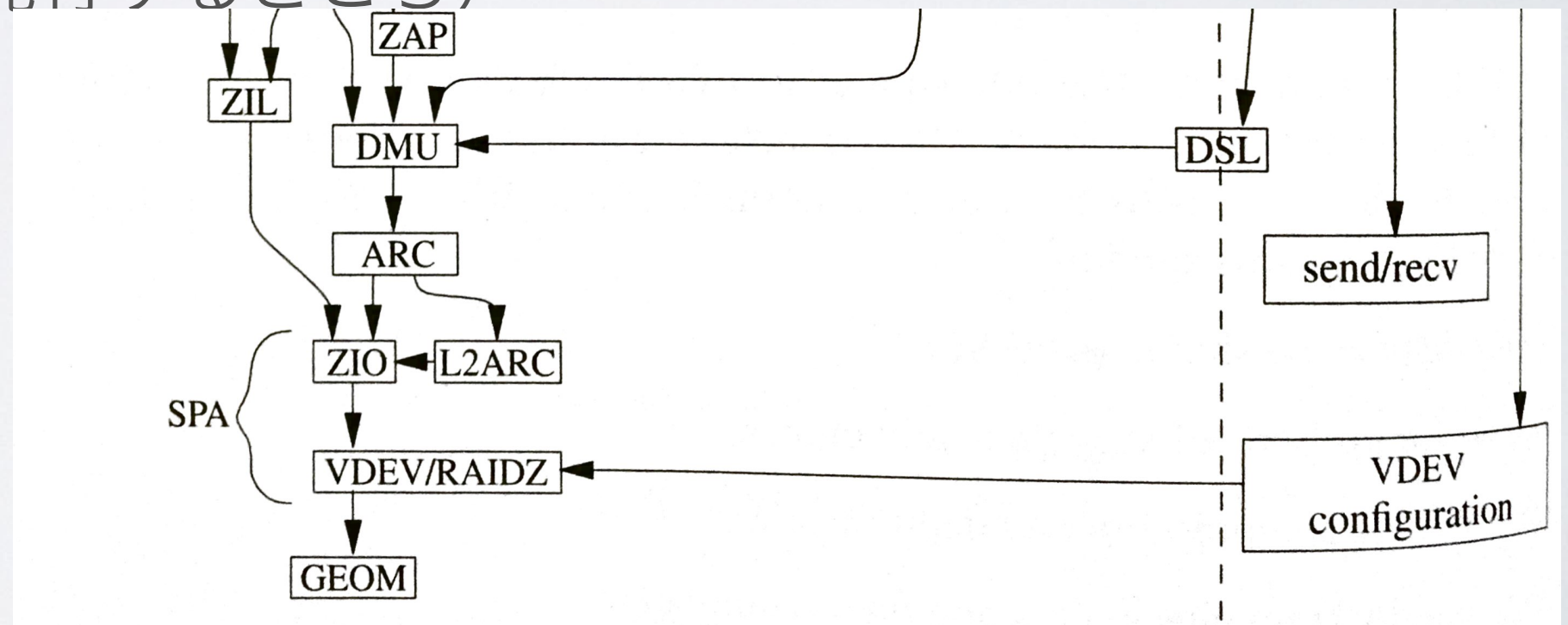
Figure 10.2 ZFS module layering. See Table 10.1 for acronyms.



ZFSの性能決定要因

性能を決めるのはどこか？

- ▶ SPAの効率が性能に大きく影響するため、その動作を知ることが重要！
- ▶ DMUより下位の要素：
 - ARC (キャッシュ)
 - ZIO (I/Oを発行するところ)



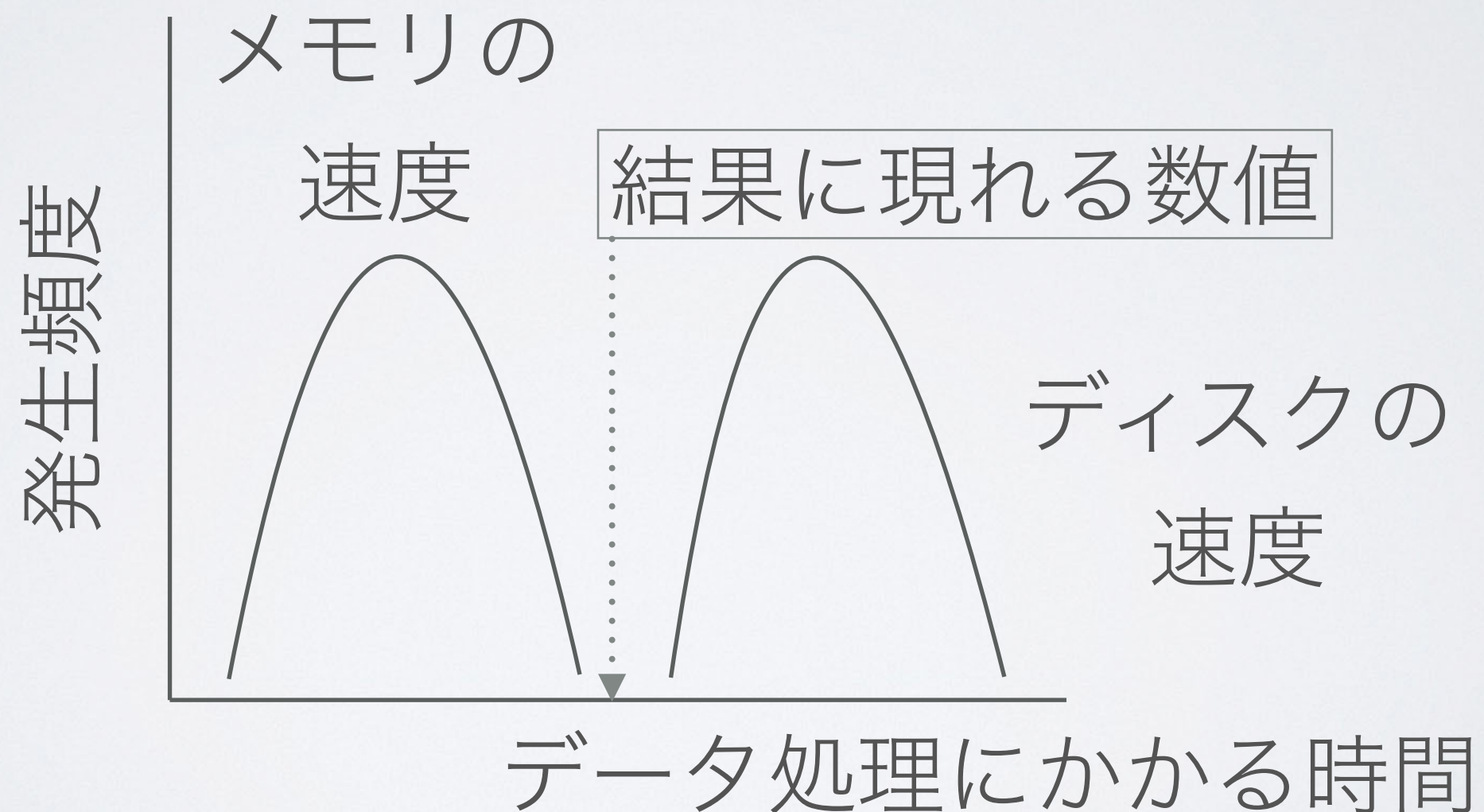
よくある落とし穴

- ▶ iozoneやbonnieなどのマイクロベンチマークツールで測定なぜダメなのか？
 - キャッシュヒットとそうでない場合に性能差が大きい



よくある落とし穴

- ▶ iozoneやbonnieなどのマイクロベンチマークツール
なぜダメなのか？
→ キャッシュヒットとそうでない場合に性能差が大きい



よくある落とし穴

- ▶ iozoneやbonnieなどのマイクロベンチマークツール
なぜダメなのか？
- ▶ 「自分の使いたい負荷パターンで
キャッシュがヒットしているかどうか」が支配的
- ▶ ヒット率を上げるにはどうしたら良いかを考える
- ▶ キャッシュミスを意図的に発生させた場合の
性能の平均値を見ても意味はない
(キャッシュヒットのチューニング結果が現れない)

よくある落とし穴

- ▶ iozoneやbonnieなどのマイクロベンチマークツール
なぜダメなのか？
- ▶ あらゆる負荷パターンで万能のチューニング法は存在しない
- ▶ 必ず自分で測定しよう

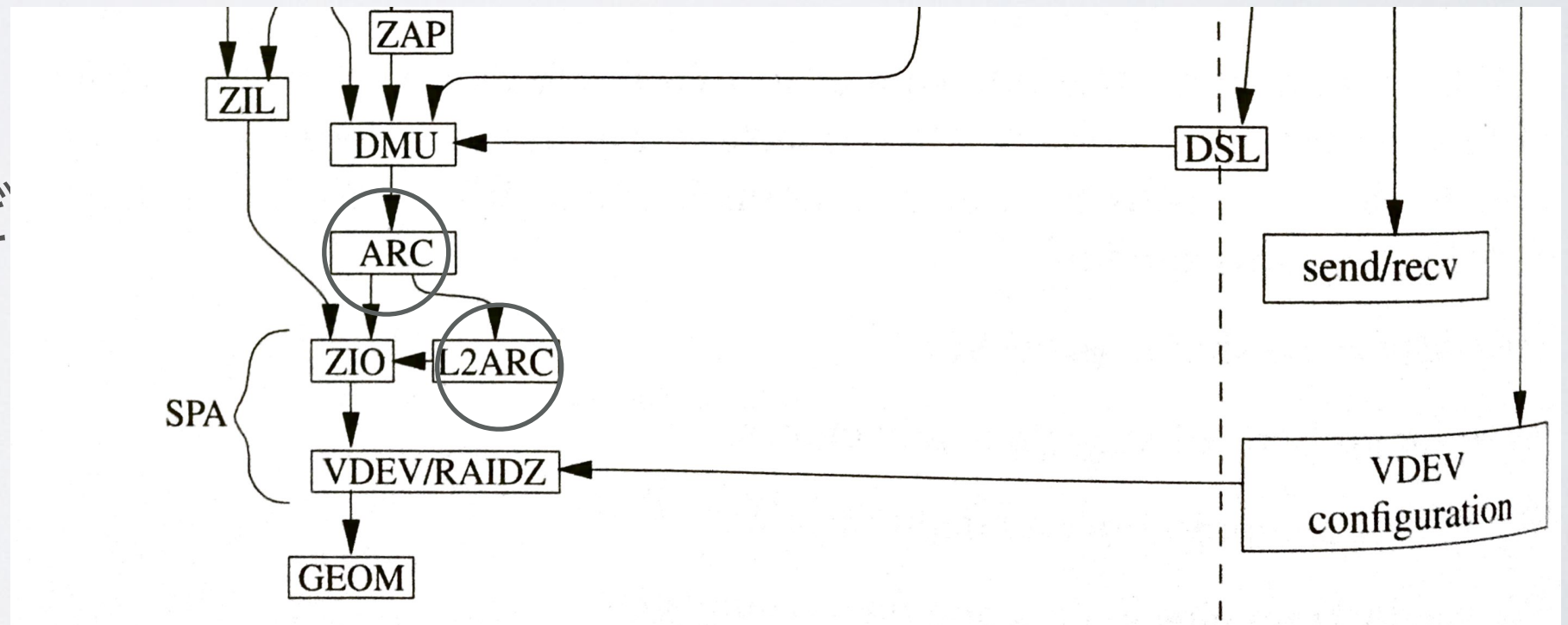
ZFSの読み出し性能

ZFSのI/O：読み出し

- ▶ ZFSはUFSと比較して、データブロックを読み出すまでにブロックポインタをたどる回数が多い
(uberblock -> objset -> objset -> ... -> data)
- ▶ 同一の処理に対して、より多くのIOPSが必要ということ！
- ▶ ほとんどの場合、キャッシュによってその差は見えなくなる
- ▶ メタデータのキャッシュヒット率が低下すると、UFSと比較してかなり遅くなってしまうことがある

ZFSのI/O：読み出し

- ▶ キャッシュはどこにある？ = DMU, ARC, L2ARC
- ▶ DMUのキャッシュはdnodeレベル、ARC, L2ARCはブロックポインタレベルで保持する
- ▶ ARCはメモリ、L2ARCはHDD, SSDなどの記憶装置



ZFSのI/O：読み出し

- ▶ ヒット率を調べるには？
- ▶ sysutils/zfs-stats をインストールして zfs-stats(8) を使いましょう

```
hrs@pool % zfs-stats -E
```

```
-----  
ZFS Subsystem Report                               Fri Jun 24 15:31:52 2016  
-----
```

```
ARC Efficiency:                                     9.16b  
  Cache Hit Ratio:                                87.49%  8.01b  
  Cache Miss Ratio:                               12.51%  1.15b  
  Actual Hit Ratio:                               86.94%  7.97b  
  
  Data Demand Efficiency:                         79.96%  98.35m  
  Data Prefetch Efficiency:                       35.42%  35.41m  
  
CACHE HITS BY CACHE LIST:  
  Most Recently Used:                             0.93%   74.29m  
  Most Frequently Used:                           98.45%   7.89b  
  Most Recently Used Ghost:                       0.61%   48.86m  
  Most Frequently Used Ghost:                     0.23%   18.37m  
  
CACHE HITS BY DATA TYPE:  
  Demand Data:                                    0.98%   78.64m  
  Prefetch Data:                                   0.16%   12.54m  
  Demand Metadata:                                95.33%   7.64b  
  Prefetch Metadata:                              3.53%  283.22m  
  
CACHE MISSES BY DATA TYPE:  
  Demand Data:                                    1.72%   19.71m  
  Prefetch Data:                                   2.00%   22.87m  
  Demand Metadata:                                84.43%  967.76m  
  Prefetch Metadata:                              11.86%  135.91m  
  
-----
```

ZFSのI/O：読み出し

- ▶ ヒット率を調べるには？
- ▶ sysutils/zfs-stats をインストールして zfs-stats(8) を使いましょう

```
hrs@pool % zfs-stats -E
```

```
-----  
ZFS Subsystem Report
```

```
Fri Jun 24 15:31:52 2016  
-----
```

```
ARC Efficiency:                                9.16b  
Cache Hit Ratio:                             87.49% 8.01b  
Cache Miss Ratio:                             12.51% 1.15b  
Actual Hit Ratio:                             86.94% 7.97b  
  
Data Demand Efficiency:                       79.96% 98.35m  
Data Prefetch Efficiency:                     35.42% 35.41m  
  
CACHE HITS BY CACHE LIST:  
Most Recently Used:                           0.93% 74.29m  
Most Frequently Used:                         98.45% 7.89b  
Most Recently Used Ghost:                     0.61% 48.86m  
Most Frequently Used Ghost:                   0.23% 18.37m  
  
CACHE HITS BY DATA TYPE:  
Demand Data:                                  0.98% 78.64m  
Prefetch Data:                                0.16% 12.54m  
Demand Metadata:                             95.33% 7.64b  
Prefetch Metadata:                           3.53% 283.22m  
  
CACHE MISSES BY DATA TYPE:  
Demand Data:                                  1.72% 19.71m  
Prefetch Data:                                2.00% 22.87m  
Demand Metadata:                             84.43% 967.76m  
Prefetch Metadata:                           11.86% 135.91m  
-----
```

追加スライド

Cache Hit Ratio と
Demand Metadata の積が
80% を超えるかどうかが目安

ZFSのI/O：読み出し

修正

- ▶ "zfs-mon -a" で表示されるARC Demand Metadata が 定常的に 90% を切っているとちょっと厳しい
- ▶ キャッシュに使うメモリが足りないか、メタデータに割り当てている量が足りない
- ▶ **調整 1：**
ZFSプロパティのprimarycache, secondarycache
- ▶ デフォルトは all になっている。metadataに設定すると、メタデータだけをキャッシュするようになる。
- ▶ secondarycache は L2ARCの設定。
- ▶ 注：primarycacheをmetadataにすると、L2ARCはallにしてもmetadataだけになる

ZFSのI/O：読み出し

- ▶ Demand Metadata が 90% を切っているとちょっと厳しい
- ▶ キャッシュに使うメモリが足りないか、メタデータに割り当てている量が足りない
- ▶ **調整 2：**
sysctl の `vfs.zfs.arc_meta_limit`
- ▶ ARCのうち、メタデータに使う量（バイト単位）
- ▶ デフォルトは全物理メモリの 1/4 に設定される
(`arc_max`は(全物理メモリ- 1GB))
- ▶ `vfs.zfs.arc_meta_limit="9G"` のように指定できる

ARCに関する注意点

ARCパラメータ

▶ 知っておこう（１）：

Q：ARC使用量を設定しているのに制限が効かないことがある

A：ZFSのARC使用量はフィードバック制御を使って制限している

▶ `vfs.zfs.arc_max`: ARCの最大値。

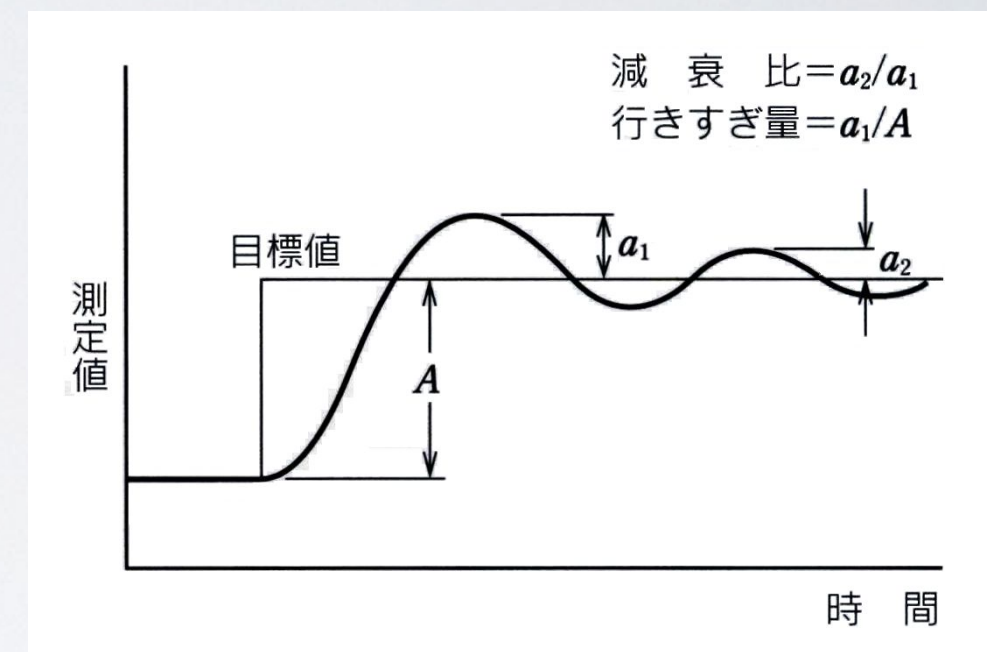
デフォルトは（全物理メモリ量-1GB）。

▶ `vfs.zfs.arc_min`: ARCの最小値。

デフォルトは全物理メモリ量の1/32。

▶ 設定してもハードリミットにはならない！

→ 一時的に超えても構わない値にすること



ARCパラメータ

▶ 知っておこう (2) :

Q : L2ARCを単純にデカくすればヒット率は改善するんじゃないの?

A : L2ARCを増やすと、耐えられないくらいメモリ消費量が増える

▶ キャッシュ管理のためにメモリが必要

▶ ARCのみ = 176B/8kB (平均が8kBブロックのケース)

▶ L2ARC = 240B/8kB (平均が8kBブロックのケース)

つまりL2ARCの容量 x 2.92%のメモリを消費！

ARCパラメータ

▶ 知っておこう（２）：

Q：L2ARCを単純にデカくすればヒット率は改善するんじゃないの？

A：L2ARCを増やすと、耐えられないくらいメモリ消費量が増える

つまりL2ARCの容量 $\times$ 2.92%のメモリを消費！

- ▶ ARCはメモリだから小さいが、L2ARCは大容量SSDでも組める
- ▶ 例：L2ARC = 1.6TB SSD、RAM = 60GB にしたシステム
(Amazon EC2など)
→ キャッシュ管理データは $1600\text{GB} \times 2.92\% = \mathbf{46\text{GB}}$ ！
- ▶ メインメモリの8割近くが持っていられる

ARCパラメータ

▶ 知っておこう（２）：

Q：L2ARCを単純にデカくすればヒット率は改善するんじゃないの？

A：L2ARCを増やすと、耐えられないくらいメモリ消費量が増える

つまりL2ARCの容量 × 2.92%のメモリを消費！

- ▶ <朗報> 10.3, 11.0 以降は 1.59% の消費に減りました。
- ▶ ヒット率を見ながらARC量を決める
- ▶ L2ARCは大きければ大きいほどヒットは増えるが、メモリ消費量を犠牲にするので、物理メモリ量の決定にも影響する
- ▶ ARCも2.15%ほどメモリを使うので、8Gで172MBくらい食います

ARCパラメータ

- ▶ 想像だけでチューニングしないこと！
 - 必ず測定をする
- ▶ 適当に値をいじったり、誰かの値を理解せずコピーしない
 - 自動チューニングはそれなりに優秀です
- ▶ 何事も増やせば良いというものではない
 - ▶ アプリケーションに使うメモリ
 - ▶ ARCに使うメモリ
 - ▶ ARCの管理に消費されるメモリ (ARC量の2%)
 - ▶ L2ARCの管理に消費されるメモリ (L2ARC量の3%)

ARCパラメータ

- ▶ 何事も増やせば良いというものではない
 - ▶ アプリケーションに使うメモリ
 - ▶ ARCに使うメモリ
 - ▶ ARCの管理に消費されるメモリ (ARC量の2%)
 - ▶ L2ARCの管理に消費されるメモリ (L2ARC量の3%)
- ▶ **例：8GB物理メモリ、アプリケーションに3GBとする**
 - ▶ 残りは5GB
 - ▶ 1GBをキャッシュ管理に確保するとL2ARCは最大33GB
 - ▶ 4GBをARCにすると管理用メモリは80MB



ARCパラメータ

- ▶ 何事も増やせば良いというものではない
 - ▶ アプリケーションに使うメモリ
 - ▶ ARCに使うメモリ
 - ▶ ARCの管理に消費されるメモリ (ARC量の2%)
 - ▶ L2ARCの管理に消費されるメモリ (L2ARC量の3%)
- ▶ **例：8GB物理メモリ、アプリケーションに3GBとする**
 - ▶ デフォルト = ARCが最大7GB。大丈夫？
 - ▶ アプリケーション用メモリが優先される
 - ▶ L2ARC管理領域を確保しないと効率が下がる



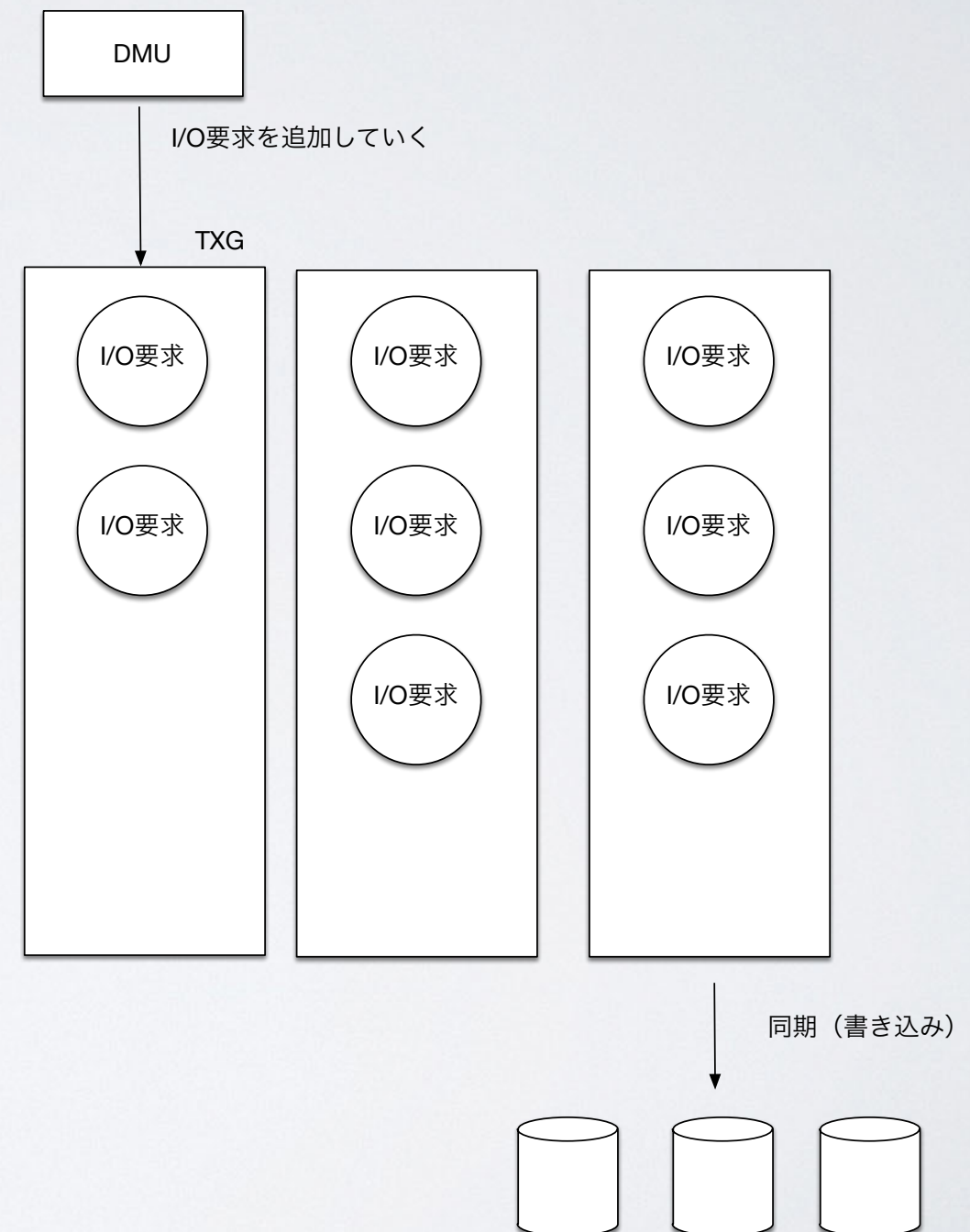
ZFSの書き込み性能

ZFSのI/O：書き込み

- ▶ ZFSは、特に書き込みの性能にチューニングが必要！

SPAの処理構造

- ▶ DMUで変更があった部分 (dirty data) は、I/O要求として蓄積されていく
- ▶ この単位をTXGと呼ぶ
- ▶ TXGは最大3つ。
Syncing TXGというTXGにある要求が処理される



ZFSのI/O：書き込み

▶ SPAの同期

▶ TXGは、

「処理データが一定量に達する」か、
「一定時間が経過する」かのいずれかで
ストレージへの同期を行う

▶ 一定量とは、全物理メモリ量の1/10である。

▶ 一定時間とは、5 秒である。

▶ 例えば16GBのRAMならどうなる？

▶ 5秒間に平均320MB/sの書き込みが生じる！

▶ HDDは間に合うか？ 自分の用途でこの速度は十分か？

ZFSのI/O：書き込み

- ▶ 同期が間に合わないとうなる？
 - ▶ 重要：同期が終わるまで、ZFSの読み出しI/Oはブロックされてしまう
 - ▶ 典型的な症状：
 - ▶ 書き込み負荷が高くなると、数秒おきにカクンカクンとシステムの応答が極端に遅くなったり、止まって見える
 - ▶ スムーズにデータが流れないので書き込み速度はHDDの本来の速度限界よりもSPAの処理で律速になってしまう
- ▶ 大きなメモリ、遅いストレージの組み合わせが要注意

ZFSのI/O：書き込み

- ▶ チューニングするには
 - ▶ 想像でやらない：同期が間に合っているのかを調べよう
- ▶ 性能の測定
 - ▶ DTraceを活用！大変便利です。
(スライド末尾にチュートリアルをつけました)
- ▶ まず、自分の使いたい負荷をかけた状態でディスクI/Oを測定

```
# dtrace -s diskio.d -c "sleep 30"
```

<https://people.allbsd.org/~hrs/FreeBSD/diskio.d>

ZFSのI/O：書き込み

```
# dtrace -s diskio.d -c "sleep 30"
```

write

value	----- Distribution -----	count
16		0
32		31
64	@@@@	505
128	@@@@@@@@@@@@@@@@@@@@@@@@@@@@	2531
256	@@@@@	588
512	@@	223
1024	@	57
2048		46
4096	@	71
8192		22
16384		34
32768	@	69
65536		27
131072		0

read

value	----- Distribution -----	count
16		0
32		2
64	@@@@@@@@@@@@@@@@	4404
128	@@@@@@@@@@@@@@@@@@	5937
256		131
512		33
1024		116
2048	@	226
4096	@	510
8192	@@@@	1546
16384	@@@@	1418
32768	@	323
65536		44
131072		7
262144		0

	avg latency	stddev	iops	throughput
write	1974us	9683us	140/s	1632k/s
read	5043us	10832us	489/s	1394k/s

ZFSのI/O：書き込み

	avg latency	stddev	iops	throughput
write	1974us	9683us	140/s	1632k/s
read	5043us	10832us	489/s	1394k/s

- ▶ この結果を確認し、チューニング前後でIOPSとスループットが増加するかどうかを調べること

ZFSのI/O：書き込み

- ▶ 次に、Syncing TXGの処理が間に合っているのかどうか確認

```
# dtrace -s delayed-io.d -c "sleep 30"  
dtrace: script 'delayed-io.d' matched 1 probe  
dtrace: pid 58771 has exited
```

no delay

12151

注：no delay だけなら問題無し

<https://people.allbsd.org/~hrs/FreeBSD/delayed-io.d>

ZFSのI/O：書き込み

- ▶ delayed が出ているようなら、間に合っていないので latencyを測定する。

```
# dtrace -s delay-mintime.d -c "sleep 30"
```

latencyのヒストグラムが出ます。

<https://people.allbsd.org/~hrs/FreeBSD/delay-mintime.d>

ZFSのI/O：書き込み

```
# dtrace -s zfs-txg.d a
```

CPU	ID	FUNCTION:NAME
8	57917	none:txg-synced 64MB of 2454MB synced in 0.22 seconds
11	57917	none:txg-synced 702MB of 2454MB synced in 0.20 seconds
7	57917	none:txg-synced 444MB of 2454MB synced in 0.17 seconds
11	57917	none:txg-synced 442MB of 2454MB synced in 0.16 seconds
4	57917	none:txg-synced 384MB of 2454MB synced in 0.18 seconds
1	57917	none:txg-synced 478MB of 2454MB synced in 0.47 seconds
2	57917	none:txg-synced 1320MB of 2454MB synced in 0.75 seconds
1	57917	none:txg-synced 1672MB of 2454MB synced in 0.62 seconds
8	57917	none:txg-synced 1502MB of 2454MB synced in 0.26 seconds
0	57917	none:txg-synced 500MB of 2454MB synced in 0.19 seconds
0	57917	none:txg-synced 500MB of 2454MB synced in 0.22 seconds
3	57917	none:txg-synced 501MB of 2454MB synced in 0.24 seconds
8	57917	none:txg-synced 615MB of 2454MB synced in 0.34 seconds
8	57917	none:txg-synced 886MB of 2454MB synced in 0.28 seconds
6	57917	none:txg-synced 3MB of 2454MB synced in 0.21 seconds

<https://people.allbsd.org/~hrs/FreeBSD/zfs-txg.d>

ZFSのI/O：書き込み

pool名を引数に

TXGが保持できる最大サイズ

TXGのI/Oのサイズ

同期にかかった時間

dtrace -s zfs-txg.d a

CPU	ID	FUNCTION:NAME
8	57917	none:txg-synced 64MB of 2454MB synced in 0.22 seconds
11	57917	none:txg-synced 702MB of 2454MB synced in 0.20 seconds
7	57917	none:txg-synced 444MB of 2454MB synced in 0.17 seconds
11	57917	none:txg-synced 442MB of 2454MB synced in 0.16 seconds
4	57917	none:txg-synced 384MB of 2454MB synced in 0.18 seconds
1	57917	none:txg-synced 478MB of 2454MB synced in 0.47 seconds
2	57917	none:txg-synced 1320MB of 2454MB synced in 0.75 seconds
1	57917	none:txg-synced 1672MB of 2454MB synced in 0.62 seconds
8	57917	none:txg-synced 1502MB of 2454MB synced in 0.26 seconds
0	57917	none:txg-synced 500MB of 2454MB synced in 0.19 seconds
0	57917	none:txg-synced 500MB of 2454MB synced in 0.22 seconds
3	57917	none:txg-synced 501MB of 2454MB synced in 0.24 seconds
8	57917	none:txg-synced 615MB of 2454MB synced in 0.34 seconds
8	57917	none:txg-synced 886MB of 2454MB synced in 0.28 seconds
6	57917	none:txg-synced 3MB of 2454MB synced in 0.21 seconds

<https://people.allbsd.org/~hrs/FreeBSD/zfs-txg.d>

ZFSのI/O：書き込み

TXGが保持できる最大サイズ

TXGのI/Oのサイズ

同期にかかった時間

```
# dtrace -s zfs-txg.d a
CPU      ID
  8    57917
```

FUNCTION:NAME
none:txg-synced

64MB of 2454MB synced in 0.22 seconds

書き込みI/Oがあれば、少なくとも5秒で1回の同期が発生する。
一回の同期が5秒以上かかっている = 間に合っていない。

<https://people.allbsd.org/~hrs/FreeBSD/zfs-txg.d>

ZFSのI/O：書き込み

- ▶ チューニングはどうやったら良いの？
 - ▶ 1) TXGのサイズ調整
 - ▶ 2) TXGの同期I/O発行量
 - ▶ 3) TXGのタイムアウト

ZFSのI/O：書き込み

▶ 1) TXGのサイズ調整

▶ 同期までに溜め込むI/Oの量は、次のsysctlで決まっている

`vfs.zfs.dirty_data_max: 2573481984`

溜め込む量（バイト）。

`vfs.zfs.dirty_data_max_percent: 10`

`data_max`をメモリ量から起動時に自動計算する時の%値。

`vfs.zfs.dirty_data_max_max: 4294967296`

`data_max_percent`で計算された値の上限。

`vfs.zfs.dirty_data_sync: 67108864`

TXG sync を強制的にスタートするサイズ: デフォルト64MB

▶ これらを増減させると、一回の同期量が変わる。

ZFSのI/O：書き込み

▶ 2) TXGの同期I/O発行量

▶ 同期用のI/Oを、HDDやSSDにいくつ並行で送るかという数

▶ sysctl とデフォルトは次の通り

vfs.zfs.vdev.trim_max_active: 64
vfs.zfs.vdev.trim_min_active: 1
vfs.zfs.vdev.scrub_max_active: 2
vfs.zfs.vdev.scrub_min_active: 1
vfs.zfs.vdev.async_write_max_active: 10
vfs.zfs.vdev.async_write_min_active: 1
vfs.zfs.vdev.async_read_max_active: 3
vfs.zfs.vdev.async_read_min_active: 1
vfs.zfs.vdev.sync_write_max_active: 10
vfs.zfs.vdev.sync_write_min_active: 10
vfs.zfs.vdev.sync_read_max_active: 10
vfs.zfs.vdev.sync_read_min_active: 10
vfs.zfs.vdev.max_active: 1000

非同期書き込み=10

同期書き込み=10

非同期読み出し=3

同期読み出し=10

scrub = 2

合計：35（注意：VDEV単位）

ZFSのI/O：書き込み

- ▶ 2) TXGの同期I/O発行量
- ▶ デバイスのキュー長を調べて、余裕があるなら増やす

```
# camcontrol identify /dev/da0 | grep Q
```

```
pass0: 600.000MB/s transfers, Command Queueing Enabled
Tagged Command Queuing (TCQ)      no          no
Native Command Queuing (NCQ)      yes          32 tags
NCQ Queue Management              no
NCQ Streaming                     no
Receive & Send FPDMA Queued       no
```

- ▶ 増やした時にTXG同期が
間に合ってるかどうか入念にチェックすること。
- ▶ デフォルトの値はSSDなど、IOPSが大きくとれるもの
に対しては少ないので、増やした方が良い。

ZFSのI/O：書き込み

▶ 3) TXGのタイムアウト

- ▶ 5秒のタイムアウトを短くすることができる

`vfs.zfs.txg.timeout`

- ▶ 短くするよりも、自分の負荷のデータ量で制限する方が良い

- ▶ 5秒より短いtimeoutは、書き込み負荷が小さい時の処理オーバヘッドが増加したり、フラグメンテーション増加の要因になる

- ▶ そもそも書き込み負荷が小さい用途なら、増やすのも一つの方法

ZFSのチューニングまとめ

▶ 読み込み：

キャッシュヒットがキーポイント。

足りなければ増やすしかないが、特にメタデータのキャッシュヒット率が低いと全体的な性能が落ちる

▶ 書き込み：

TXGの同期量がキーポイント。

自分の処理したい負荷、自分のマシンのメモリ、ストレージの能力に対して適正かどうかを測定して判断

▶ まだまだありますが、今回はこのへんで...

(もっと聞きたいという人は要望を表明ください)

おしまい

- ▶ 質問はありますか？



DTrace入門（おまけ）

佐藤 広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2016/11/29

DTraceとは？

- ▶ 名前は聞いたことがあるけどよく知らない...
- ▶ システムやアプリケーションプログラムの内部動作を稼働中に調べるためのツール
 - ▶ 動作を調べる
 - ▶ ベンチマークを取る
 - ▶ などなど。 **「稼働中に」** **「対象を変更せずに」** がポイント
- ▶ 2005年3月にSolarisに追加され、現在はOpenSolaris(illumos), Mac OSX, FreeBSD, NetBSD, Linuxなどで使える（カーネルが対応している必要あり）

DTraceとは？

- ▶ Unix系OSでシステムの動作を調べるコマンドはたくさんある
- ▶ **Solaris:** sar(1), vmstat(1M), mpstat(1M), iostat(1M), netstat(1M), kstat(1M), prstat(1M),...
- ▶ **Mac OSX:** sar(1), vm_stat(1), top(1),...
- ▶ **FreeBSD:** systat(1), vmstat(8), iostat(8), netstat(8), sockstat(1), procstat(1), top(1),...
- ▶ DTrace は、これらと何か違うものなのか？

DTraceの特徴

- ▶ 個々のツールは統計情報の取得にCPU時間を多く使う
→ top(1)はとても重いユーティリティです。
- ▶ ktraceやtrussのようなツールはプロセス単位でしか情報が取れない
- ▶ システムコールなどのカーネルの動作は調べられるけれど、ユーザランドプログラムの動作を調べる方法は少ない
- ▶ 知りたい情報以外もたくさん出てきてしまう

- ▶ 例：今あるプロセスが read(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい！

準備

- ▶ FreeBSDの動作を調べるには
- ▶ FreeBSD 10 系ならば GENERIC カーネルで使えます。
- ▶ 9は
 - options KDTRACE_HOOKS
 - options DDB_CTF
 - options KDTRACE_FRAME
 - makeoptions DEBUG="-g"
 - makeoptions WITH_CTF=1を入れてカーネルを作る必要があります。
(ハンドブック参照のこと)

準備

- ▶ FreeBSD以外のユーザランドプログラムにも DTraceを使いたい場合には
- ▶ /etc/make.confに
STRIP=
CFLAGS+=-fno-omit-frame-pointer
WITH_CTF=1
を付けてください

準備

- ▶ DTraceカーネルモジュールを読む

```
# kldload dtraceall
```

- ▶ dtraceコマンドを実行する

```
# dtrace -l
```

- ▶ 覚えておこう

- ▶ 基本的にdtraceコマンドだけを使う
- ▶ root権限が必要
- ▶ カーネルモジュールは、読み込まれていなければ
dtraceコマンドを実行する時点で自動で読み込まれる

準備

```
# dtrace -l
```

ID	PROVIDER	MODULE	FUNCTION NAME
1	dtrace	BEGIN	
2	dtrace	END	
3	dtrace	ERROR	
4	fbt	kernel	camstatusentrycomp entry
5	fbt	kernel	camstatusentrycomp return
6	fbt	kernel	cam_compat_handle_0x17 entry
7	fbt	kernel	cam_compat_handle_0x17 return
:			
:			
:			

▶ -l を指定して実行すると、何かたくさん行が出てくる

使ってみよう

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

使ってみよう

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```

使ってみよう

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -n 'syscall::read:entry /execname != "dtrace"/ { @reads[execname, arg2] =  
count(); }' -c "sleep 10"  
dtrace: description 'syscall::read:entry ' matched 2 probes  
dtrace: pid 98661 has exited
```

ftpd	71	1
ftpd	128	1
ftpd	260	1
rsync	262144	1
ftpd	41448	2
ftpd	1048600	2
sshd	16384	2
inetd	260	3
inetd	32768	9
ftpd	32768	14
spegla	4096	16
spegla	16384	18
cvsupd	8192	203
cvsupd	4096	1249
spegla	1	1653

使ってみよう

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -n 'syscall::read:entry /execname != "dtrace"/ { @reads[execname, arg2] = count(); }'  
-c "sleep 10"  
dtrace: description 'syscall::read:entry ' matched 2 probes  
dtrace: pid 98661 has exited
```

ftpd	71	1	回数
ftpd	128	1	
ftpd	260	1	
rsync	262144	1	
ftpd	41448	2	
ftpd	1048600	2	
sshd	16384	2	
inetd	260	3	
inetd	32768	9	
ftpd	32768	14	
spegla	4096	16	
spegla	16384	18	
cvsupd	8192	203	
cvsupd	4096	1249	
spegla	1	1653	

何をやったのか？

- ▶ dtrace コマンドは、D言語（注意：Walter Bright のD言語ではなく DTrace専用の簡易言語）で書かれたスクリプトを実行する
- ▶ 「-n "スクリプト本体"」、もしくは「-s スクリプトファイル名」
- ▶ 「-c "コマンド"」は、指定したコマンドが終了したら dtrace を止めるという指定

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```


何をやったのか？

- ▶ D言語
 - ▶ AWKにととても良く似た文法
 - ▶ 「プローブ /述語/ { 手続き }」 という宣言を並べるだけ！
- ▶ プローブ：調べたいポイント。「-l」 で出てきたリストがそれ。
 - ▶ `syscall::read:entry = read(2)` が呼ばれた時点

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```

何をやったのか？

- ▶ D言語
 - ▶ AWKにととても良く似た文法
 - ▶ 「プローブ /述語/ { 手続き }」 という宣言を並べるだけ！
- ▶ 述語：条件の絞り込み（省略しても良い）
 - ▶ `/execname != "dtrace"/`
 - 実行ファイル名が `dtrace` でないもの

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```


何をやったのか？

- ▶ D言語
 - ▶ AWKにととても良く似た文法
 - ▶ 「プローブ /述語/ { 手続き }」 という宣言を並べるだけ！
- ▶ 手続き：何をやるか
 - ▶ `execname` と `arg2` をキーにした連想配列に回数をセット

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{  
    @reads[execname, arg2] = count();  
}
```


詳しく読んでみる

プローブはread(2)の入り口

```
syscall::read:entry /execname != "dtrace"/  
{  
    @reads[execname, arg2] = count();  
}
```

- ▶ プローブを知るには？
 - ▶ dtrace -l で一覧が表示される
 - ▶ 「プロバイダ：モジュール：関数：プローブ名」
 - ▶ システムコールなら、syscallプロバイダにある
 - ▶ 省略や「*」が使える (syscall::read*: とするとreadvも対象)

詳しく読んでみる

述語

```
syscall::read:entry /execname != "dtrace"/  
{  
    @reads[execname, arg2] = count();  
}
```

- ▶ 述語で使える書き方
 - ▶ AWKと同じ（regexではない）。組み込み変数ができる。

DTraceの組み込み変数

名前	意味
arg0, arg1, ... arg9	引数。int64_t。定義されていなければ0になる
args[]	引数。型は定義されたもの。
cpu	実行しているCPUのID
curpsinfo	現在のプロセスの情報 (struct psinfo_t)
errno	最後のシステムコールのerrno
execname	現在のプロセス名
pid, ppid, uid, tid	PID, PPID, UID, スレッドID
timestamp	時刻

DTraceの組み込み変数

名前	意味
▶	ここで出てくる構造体の定義は、 <code>/usr/lib/dtrace/</code> にある。
▶	カーネルのヘッダファイルにある構造体もちろん使用可能
<pre>typedef struct psinfo { int pr_nlwps; /* number of threads */ pid_t pr_pid; /* unique process id */ pid_t pr_ppid; /* process id of parent */ pid_t pr_pgid; /* pid of process group leader */ pid_t pr_sid; /* session id */ uid_t pr_uid; /* real user id */ uid_t pr_euid; /* effective user id */ gid_t pr_gid; /* real group id */ gid_t pr_egid; /* effective group id */ uintptr_t pr_addr; /* address of process */ string pr_psargs; /* process arguments */ u_int pr_arglen; /* process argument length */ u_int pr_jailid; /* jail id */ } psinfo_t;</pre>	

詳しく読んでみる

述語

```
syscall::read:entry /execname != "dtrace"/  
{  
    @reads[execname, arg2] = count();  
}
```

▶ 述語で使える書き方

- ▶ AWKと同じ（regexではない）。組み込み変数ができる。
- ▶ プロセス名が dtrace と一致しないものの全部、という意味。

詳しく読んでみる

述語

```
syscall::read:entry
/execname != "dtrace" && uid == 1000/
{
    @reads[execname, arg2] = count();
}
```

▶ 述語で使える書き方

- ▶ AWKと同じ（regexではない）。組み込み変数ができる。
- ▶ プロセス名が dtrace と一致しないものの全部、という意味。
- ▶ こうすると、さらに UID が 1000 に一致するものに限定

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{ 手続き  
    @reads[execname, arg2] = count();  
}
```

- ▶ 手続きで使える書き方
 - ▶ これもAWKにととても似ている
 - ▶ 変数は宣言せずに使える。C言語風のキャスト文法にも対応
 - ▶ 宣言もできる。char, short(int16_t), int(int32_t), long long(int64_t), float, double, long doubleが使える。

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{ 手続き  
    @reads[execname, arg2] = count();  
}
```

▶ read(2) のarg2って何だ？

```
% man 2 read
```

```
ssize_t  
read(int fd, void *buf, size_t nbytes);
```

↓ここ

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{ 手続き  
    @reads[execname, arg2] = count();  
}
```

- ▶ 手続きで使える書き方
 - ▶ 文字列型として string がある。
 - ▶ []をつけると配列型になり、連想配列になる。配列のキーは","で区切って複数指定可能。
 - ▶ 構造体も定義可能。/usr/lib/dtrace/*.d の内容がデフォルト

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{ 手続き  
    @reads[execname, arg2] = count();  
}
```

▶ 手続きで使える書き方

- ▶ スレッドローカル変数として、`self->x = 1` という書き方ができる。0 を代入するか、スレッドがなくなると解放される。
- ▶ 手続きローカル変数として、`this->x = 1` という書き方ができる。

詳しく読んでみる

```
syscall::read:entry /execname != "dtrace"/  
{ 手続き  
    @reads[execname, arg2] = count();  
}
```

- ▶ 集約変数 @ と集約関数
 - ▶ カウンタ用の特殊文法。
 - ▶ @a = count(); とすると、呼ばれる度に a++ される。
 - ▶ @a[execname] = count(); とすると、プロセス名単位に
- ▶ a++ と同じだが、SMP環境での挙動がポイント

詳しく読んでみる

- ▶ 集約変数 @ と集約関数
 - ▶ 集約変数の代入は集約関数しかできない
 - ▶ `count()` : カウンタの値を1増やす
 - ▶ `sum(a)` : カウンタの値をa増やす
 - ▶ `avg(a)` : カウンタの平均値を計算する
 - ▶ `min(a), max(a)` : カウンタに最小値 or 最大値を保持する
 - ▶ `quantize(a)` : 2の冪乗の単位でヒストグラムを生成する
 - ▶ `lquantize(a, b, c, d)` : 線形なヒストグラムを生成する

もっと使ってみよう

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```

もっと使ってみよう

- ▶ 例：**rsync**のプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べて**ヒストグラム**にしたい

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname == "rsync" /  
{ @reads = quantize(arg2); }'
```


もっと使ってみよう

- ▶ 例：rsyncのプロセスが10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べてヒストグラムにしたい

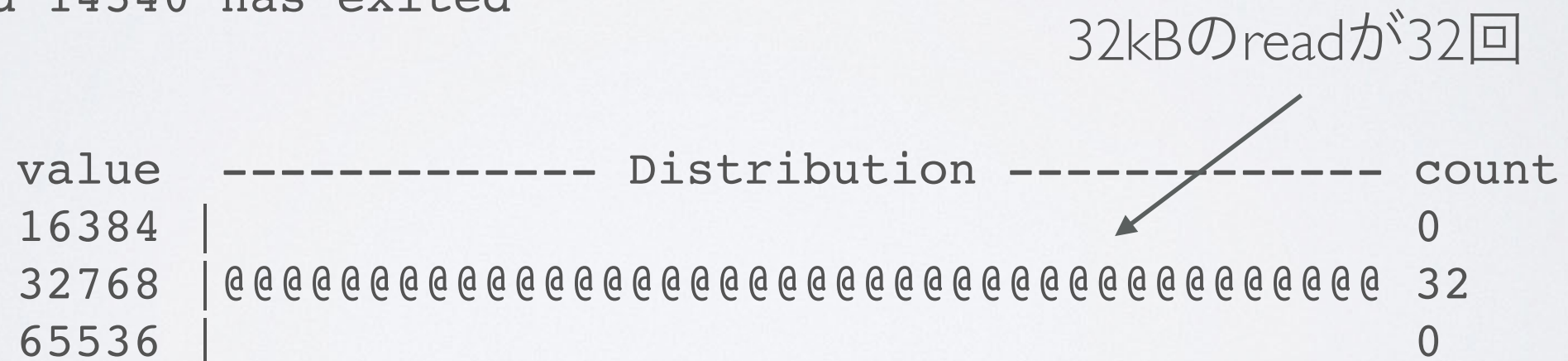
```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname == "rsync"/ { @reads = quantize(arg2); }'  
dtrace: description 'syscall::read:entry' matched 2 probes  
dtrace: pid 14340 has exited
```

value	----- Distribution -----	count
16384		0
32768	@@@@	32
65536		0

もっと使ってみよう

- ▶ 例：rsyncのプロセスが10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べてヒストグラムにしたい

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname == "rsync"/ { @reads = quantize(arg2); }'  
dtrace: description 'syscall::read:entry' matched 2 probes  
dtrace: pid 14340 has exited
```



quantize() は回数と大きさのヒストグラムを生成する

もっと使ってみよう

- ▶ スクリプトにして実行しよう

例：5秒の間に、プロセス名と、そのプロセスが開いたパス名のリストを出すスクリプト

```
#!/usr/sbin/dtrace -s
profile:::tick-5s
{
    exit(0);
}
syscall::open:entry
{
    @opens[execname, copyinstr(arg0)] = count();
}
```

もっと使ってみよう

- ▶ スクリプトにして実行しよう

例：5秒の間に、プロセス名と、そのプロセスが開いたパス名のリストを出すスクリプト

```
#!/usr/sbin/dtrace -s
profile:::tick-5s
{
    exit(0);
}
syscall::open:entry
{
    @opens[execname, copyinstr(arg0)] = count();
}
```

shebang行を書いて
"opens.d" という名前で保存

もっと使ってみよう

- ▶ スクリプトにして実行しよう

例：5秒の間に、プロセス名と、そのプロセスが開いたパス名のリストを出すスクリプト

```
#!/usr/sbin/dtrace -s
profile:::tick-5s
{
    exit(0);
}
syscall::open:entry
{
    @opens[execname, copyinstr(arg0)] = count();
}
```

profile:::tick-Ns というプローブは、
N秒後に一回マッチするプローブ

exit()関数は、dtraceを終了させる

もっと使ってみよう

- ▶ スクリプトにして実行しよう

例：5秒の間に、プロセス名と、そのプロセスが開いたパス名のリストを出すスクリプト

```
# ./opens.d
dtrace: script './opens.d' matched 3 probes
CPU      ID          FUNCTION:NAME
  0    55680          :tick-5s

ftpd      /etc/group          8
ftpd      /etc/pwd.db         8
ftpd      /usr/share/zoneinfo/UTC  8
ftpd      /usr/share/zoneinfo/posixrules  8
ftpd      .                    24
#
```


もっと使ってみよう

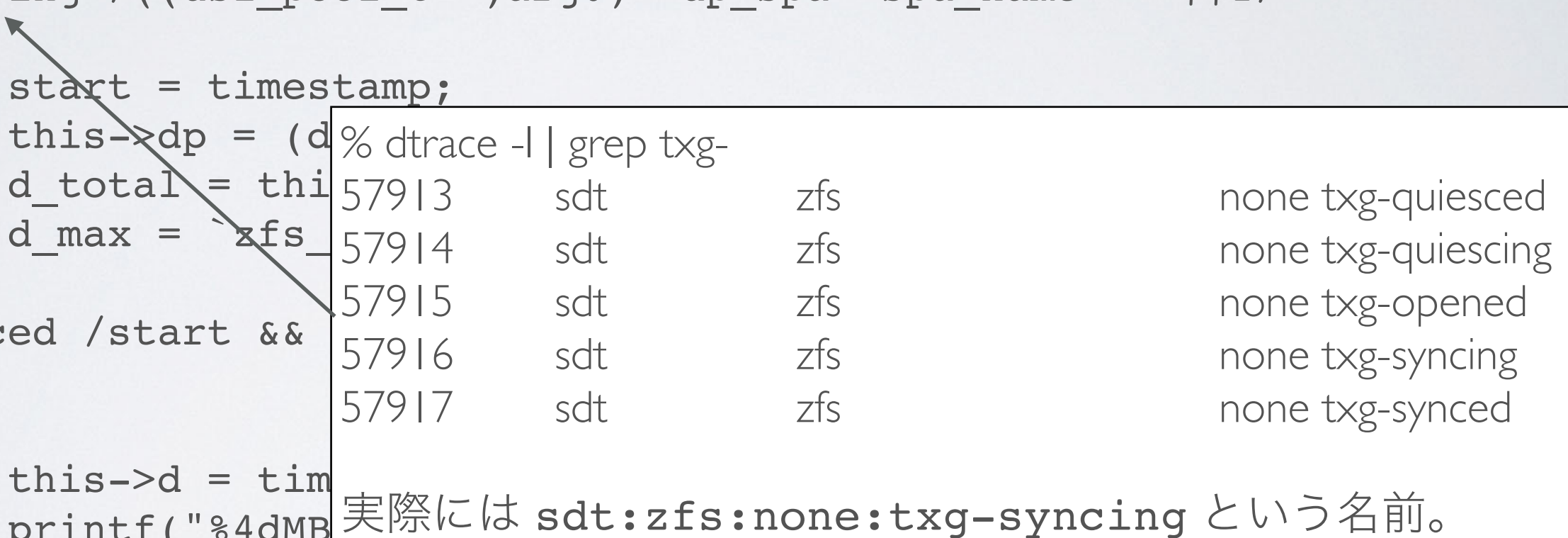
▶ 前回のZFSの性能チューニングで使ったスクリプト

```
/*
 * Measure ZFS transaction group statistics
 */
txg-syncing /((dsl_pool_t *)arg0)->dp_spa->spa_name == $$1/
{
    start = timestamp;
    this->dp = (dsl_pool_t *)arg0;
    d_total = this->dp->dp_dirty_total;
    d_max = `zfs_dirty_data_max`;
}
txg-synced /start && ((dsl_pool_t *)arg0)->dp_spa->spa_name
== $$1/
{
    this->d = timestamp - start;
    printf("%4dMB of %4dMB synced in %d.%02d seconds",
    d_total / 1024 / 1024,
    d_max / 1024 / 1024, this->d / 1000000000,
    this->d / 10000000 % 100);
}
```

もっと使ってみよう

▶ 前回のZFSの性能チューニングで使ったスクリプト

```
/*
 * Measure ZFS transaction group statistics
 */
txg-syncing /((dsl_pool_t *)arg0)->dp_spa->spa_name == $$1/
{
    start = timestamp;
    this->dp = (d
    d_total = thi
    d_max = `zfs_
}
txg-synced /start &&
== $$1/
{
    this->d = tim
    printf("%4dMB
    d_total / 1024 / 1024,
    d_max / 1024 / 1024, this->d / 1000000000,
    this->d / 10000000 % 100);
}
```



57913	sdt	zfs	none txg-quietced
57914	sdt	zfs	none txg-quietcing
57915	sdt	zfs	none txg-opened
57916	sdt	zfs	none txg-syncing
57917	sdt	zfs	none txg-synced

実際には sdt:zfs:none:txg-syncing という名前。

もっと使ってみよう

▶ 前回のZFSの性能チューニングで使ったスクリプト

```
/*
 * Measure ZFS transaction group statistics
 */
txg-syncing /((dsl_pool_t *)arg0)->dp_spa->spa_name == $$1/
{
    start = timestamp;
    this->dp = (dsl_pool_t *)arg0;
    d_total = this->dp->dp_dirty_total;
    d_max = `zfs_`$1
}
```

\$\$1 は、dtraceコマンドの引数の1番目（ここではzpool名）

このへんはカーネルのソースを読んで、

どのデータにアクセスしているのか調べないとわかりません....

a->spa_name

```
    this->d = timestamp - start;
    printf("%4dMB of %4dMB synced in %d.%02d seconds",
        d_total / 1024 / 1024,
        d_max / 1024 / 1024, this->d / 1000000000,
        this->d / 10000000 % 100);
```

```
}
```

もっと使ってみよう

- ▶ とてもとても複雑な例

```
# less /usr/sbin/dtruss
```

- ▶ truss(1)をD言語で書いたもの。そのまま実行できます。

```
# dtruss df -h
SYSCALL(args)                = return
mmap(0x0, 0x8000, 0x3)         = 6422528 0
issetugid(0x0, 0x0, 0x0)     = 0 0
lstat("/etc\0", 0x7FFFFFFFD3E8, 0x0) = 0 0

...
```


ここまでのまとめ

- ▶ DTraceとは、カーネルやユーザランドプログラムの挙動を調べるための機構です。
- ▶ DTraceを使うには、カーネルが対応する必要があります。
FreeBSD, Mac OS Xなどは、標準インストールで使えます。
- ▶ プローブ（データ収集点）とそれに対する処理をD言語で書き、イベントドリブンで動作します。
- ▶ D言語はAWKの文法によく似ています。

ここまでのまとめ

- ▶ スクリプトを書くにはソースを読む能力が必要ですが、すでにたくさんのスクリプトが公開されています。
性能測定用スクリプトなどは、詳しい内部構造を知らなくても使えます。
- ▶ プローブの種類は、OSによって異なります。
他のOS用のスクリプトを持ってくる時には要注意。
- ▶ 既存のコードを変更せず、性能与える影響を最小限に抑えるように設計されているため、デバッグなどにも便利です
(開発者はprintfデバッグの代わりに使えて重宝します)

次の話題

- ▶ DTraceで性能測定
- ▶ ユーザランドプログラムで使うには

復習：使い方

- ▶ 前準備は、前の資料の「準備」を参照のこと

- ▶ 例：今走っているプロセスが10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -c "sleep 10" -n '  
syscall::read:entry /execname != "dtrace"/  
{ @reads[execname, arg2] = count(); }'
```


復習：使い方

- ▶ 前準備は、前の資料の「準備」を参照のこと

- ▶ 例：今走っているプロセスが10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

プローブ

述語

```
syscall::read:entry /execname != "dtrace"/  
{  
    @reads[execname, arg2] = count();  
}
```

アクション

復習：使い方

- ▶ 例：今走っているプロセスが 10秒間でread(2)のシステムコールを何回発行し、どれくらいのデータ長のデータを読んだのか調べたい。

```
# dtrace -n 'syscall::read:entry /execname != "dtrace"/ { @reads[execname, arg2] = count(); }'  
-c "sleep 10"  
dtrace: description 'syscall::read:entry ' matched 2 probes  
dtrace: pid 98661 has exited
```

ftpd	71	1	回数
ftpd	128	1	
ftpd	260	1	
rsync	262144	1	
ftpd	41448	2	
ftpd	1048600	2	
sshd	16384	2	
inetd	260	3	
inetd	32768	9	
ftpd	32768	14	
spegla	4096	16	
spegla	16384	18	
cvsupd	8192	203	
cvsupd	4096	1249	
spegla	1	1653	

性能測定

- ▶ I/Oの測定：ioの発生原因とデータ長を出す ioプロバイダ

```
profile:::tick-5s 5秒で終わり
{
    exit(0);
}
io:::start /args[0] != NULL/ IOのデータ長
{
    @[pid, execname] = quantize(args[0]->bio_bcount);
}
```

```
20976 sync
value ----- Distribution ----- count
 1024 | 0
 2048 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
 4096 | 0
 8192 | 0
16384 | @@@@@@@@@@@@@@@@@@ 1
32768 | 0
```

性能測定

- ▶ I/Oの測定：特定のプロセスのioの発生原因とデータ長を出す

```
profile:::tick-5s
```

```
{  
    exit(0);
```

`$target` は「-c コマンド」もしくは -p PIDで指定

```
}  
io:::start /pid == $target && args[0] != NULL/  
{
```

```
    @[ustack()] = count();  
}
```

`ustack()`は、ユーザランドのスタックトレースを返す

性能測定

- ▶ I/Oの測定：特定のプロセスのioの発生原因とデータ長を出す

```
# dtrace -s iopid.d -c 'cp -R /var/log /tmp'
dtrace: script 'iopid.d' matched 2 probes
dtrace: pid 21004 has exited
      libc.so.7`__sys_getdirentries+0x7
      0x3
      0xc985c031
      1
      libc.so.7`mkdir+0x7
      cp`0x804c6d8
      0x585
      1
      libc.so.7`__sys_openat+0x7
      0xffffffff9c
      0x9090c35d
      1
      libc.so.7`_write+0x7
      0x4
      0x44f8e0f
      177
```

性能測定

- ▶ I/Oの測定：I/Oのレイテンシをデバイス単位で出す

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
io:::start
```

#pragma D option quiet は「-q」と同じ

```
{
    start_time[arg0] = timestamp;
}
```

timestampは現在時刻。

io-startで代入

```
io:::done /this->start = start_time[arg0]/
{
```

io-doneで代入

```
    this->delta = (timestamp - this->start) / 1000;
    @a[args[1]->device_name, args[1]->unit_number] =
        quantize(this->delta);
    start_time[arg0] = 0;
}
```

デバイス名(da0)など

```
profile:::tick-10s
{
    exit(0);
}
```


性能測定

- ▶ I/Oの測定：I/Oのレイテンシをデバイス単位で出す

da0

value	----- Distribution -----	count
0		0
1		1
2		0
4		2
8		2
16		1
32		13
64	@	78
128	@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@	3971
256	@@@@@@@@@	1125
512	@@@	490
1024	@@	232
2048		65
4096		11
8192		0

性能測定

tcpプロバイダ

▶ ネットワーク :

80/tcpに到着するパケット数をクライアントIP別に集計

tcp::receiveはTCP接続を受け取った時点

```
tcp::receive /args[4]->tcp_dport == 80/  
{  
    @[args[2]->ip_daddr] = count();  
}
```

```
# dtrace -s httpc.d -c "sleep 10"  
dtrace: script 'httpc.d' matched 1 probe  
dtrace: pid 21183 has exited
```

```
127.0.0.1  
::1
```

```
12  
12
```


性能測定

▶ ネットワーク：TCP接続にかかる時間を測定

```
tcp::connect-request  
{
```

cs_cidは接続ID

```
    start[args[1]->cs_cid] = timestamp;
```

```
}  
tcp::connect-established /start[args[1]->cs_cid]/  
{
```

```
    @["Latency (us)", args[2]->ip_daddr] =  
        quantize(timestamp - start[args[1]->cs_cid]);  
    start[args[1]->cs_cid] = 0;
```

```
}  
# dtrace -s tcpl latency.d -c "nc -z www.yahoo.com 80" ncを使って接続
```

```
dtrace: script 'tcpl latency.d' matched 2 probes
```

```
Connection to www.yahoo.com 80 port [tcp/http] succeeded!
```

```
dtrace: pid 21350 has exited
```

Latency (us)		172.16.87.129
value	Distribution	count
131072		0
262144	@@@@	1
524288		0

性能測定

▶ ネットワーク：TCPステートマシンの遷移動作を見る

```
#pragma D option quiet
#pragma D option switchrate=10
```

```
dtrace:::BEGIN 最初に一回
{
    printf(" %12s %3s %12s %-20s %-20s\n",
           "PID", "CPU", "DELTA(us)", "OLD", "NEW");
    last = timestamp;
}
```

```
tcp:::state-change
{
    this->elapsed = (timestamp - last) / 1000;
    printf(" %12d %3d %12d %-20s -> %-20s\n",
           pid, cpu, this->elapsed,
           tcp_state_string[args[5]->tcps_state],
           tcp_state_string[args[3]->tcps_state]);
    last = timestamp;
}
```

tcps_stateはステートを保持している

性能測定

- ▶ ネットワーク：TCPステートマシンの遷移動作を見る

wgetを使って接続：TCP接続が2本出てる！

```
# dtrace -s tcpstate.d -c "wget -o /dev/null http://www.google.com"
```

PID	CPU	DELTA(us)	OLD	NEW
21473	1	350651	state-closed	-> state-syn-sent
12	1	103086	state-syn-sent	-> state-established
21473	1	64244	state-closed	-> state-syn-sent
12	1	102376	state-syn-sent	-> state-established
21473	1	78939	state-established	-> state-fin-wait-1
12	1	187	state-fin-wait-1	-> state-fin-wait-2
12	1	1544	state-fin-wait-2	-> state-time-wait
21473	1	16546	state-established	-> state-fin-wait-1
12	1	729	state-fin-wait-1	-> state-fin-wait-2
12	1	16	state-fin-wait-2	-> state-time-wait

PID=12ってなんだ？

ユーザランドDTrace

- ▶ PIDプロバイダを使ってユーザランドも調べられる
- ▶ そもそもDTraceの動作は、カーネル・ユーザランドの区別を特別扱いしない
- ▶ ユーザランド関数のプローブ：

`pid$target:procname:probefunc:entry`

ユーザランドDTrace

- ▶ dtrace -l をしてみる

```
# dtrace -l -P pid\${target} -c "/bin/ls" | wc -l  
230383
```

- ▶ 「pid\\${target}:共有ライブラリ名:関数名:プローブ名」

```
# dtrace -l -P pid\${target} -c "/bin/ls"  
.....  
56421    pid21553    libutil.so.9    forkpty return  
.....  
269515   pid21553    libc.so.7       poll entry  
.....
```

ユーザランドDTrace

- ▶ /bin/echoを実行して、libc の関数を呼んでるところを見る

```
# dtrace -n 'pid$target:libc*::entry { @[probefunc] = count(); }' \  
-c "/bin/echo"
```

```
dtrace: description 'pid$target:libc*::entry ' matched 2996 probes
```

```
dtrace: pid 21579 has exited
```

__malloc	1
__sys_exit	1
__sys_writev	1
_exit	1
_writev	1
atexit	1
exit	1
malloc	1
memset	1
mmap	1
writev	1
__cxa_finalize	2

「pid\$target:**libc***::entry」を見ている

ユーザランドDTrace

- ▶ 自分で作ったプログラムならどうなの？

```
#include <stdio.h>

void
myfunc(int i)
{
    printf("...%d\n", i);
    return;
}

int
main(void)
{
    printf("hello, world\n");
    myfunc(100);

    return(0);
}
```

ユーザランドDTrace

- ▶ 自分で作ったプログラムならどうなの？

```
# cc hello.c
# dtrace -q -n 'pid$target:a.out:myfunc:entry \
    { printf("entry = %d\n",arg0); }' \
    -c ./a.out

hello, world
...100
entry = 100
#
```

「pid\$target:実行ファイル名:関数名:entry」 になるところに注意

```
# dtrace -l -m pid$target:a.out -c ./a.out
53695    pid21805          a.out          _start entry
53696    pid21805          a.out          myfunc return
53697    pid21805          a.out          myfunc entry
53698    pid21805          a.out          myfunc 0
53699    pid21805          a.out          myfunc 1
....
```

「-m プロバイダ:モジュール」 で表示を制限できる

ユーザランドDTrace

- ▶ MySQLやPostgreSQLは、DTraceに対応している
 - `mysql$target:::filesort-start` や `mysql$target:::filesort-done` が定義されていて、データベースの検索の性能分析や動作をトレースできる

```
mysql$target:::filesort-start
{
    self->ts = timestamp;
    printf("Sort start: %s", copyinstr(arg0));
}
mysql$target:::filesort-done
{
    printf("Sort done: %d ms / Result: %s",
        (timestamp - self->ts) / 1000000,
        copyinstr(arg0));
}
```

ユーザランドDTrace

- ▶ DTraceに対応してるという意味は？
 - 独自のプロバイダを持っているということ
(SDT: Statically Defined Tracingというタイプのプローブ)
- ▶ 関数単位であれば、対応していなくても使える
- ▶ USDTの作り方：

```
% cat provider.d
provider database {
    probe query__start(char *);
    probe query__done(char *);
};
```

定義

```
% dtrace -h -s provider.d
```

provider.hの生成

```
DATABASE_QUERY_START("hoge") -> database$target:::query-start
DATABASE_QUERY_DONE("fuga") -> database$target:::query-done
```

対応関係

ユーザランドDTrace

```
#include <stdio.h>
```

```
#include <sys/sdt.h>  
#include "provider.h"
```

provider.h と sys/sdt.h をinclude

```
int  
main(void)
```

```
{
```

```
    /* Give us time to start DTrace */  
    sleep(5);
```

定義したものを使う

```
    DATABASE_QUERY_START("SELECT * FROM apples");
```

```
    /* simulate a long query */
```

```
    sleep(1);
```

```
    DATABASE_QUERY_DONE("TOO MANY APPLES");
```

```
    return (0);
```

```
}
```

注意：libelf をリンクすること

まとめ

- ▶ 一見複雑そうに見える測定でも、数行のスクリプトでいけるだけの記述力
- ▶ デバッグ、ユニットテスト、動作の検証、性能測定など、いろいろな用途に重宝するはず
- ▶ 触ってみましょう！