FreeBSD 勉強会

ストレージの管理: GEOM, UFS, ZFS

佐藤　広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2012/7/20

FreeBSD 勉強会

ストレージの管理: GEOM, UFS, ZFS

佐藤 広生 <hrs@FreeBSD.org>

東京工業大学/ FreeBSD Project

2012/7/20

講師紹介

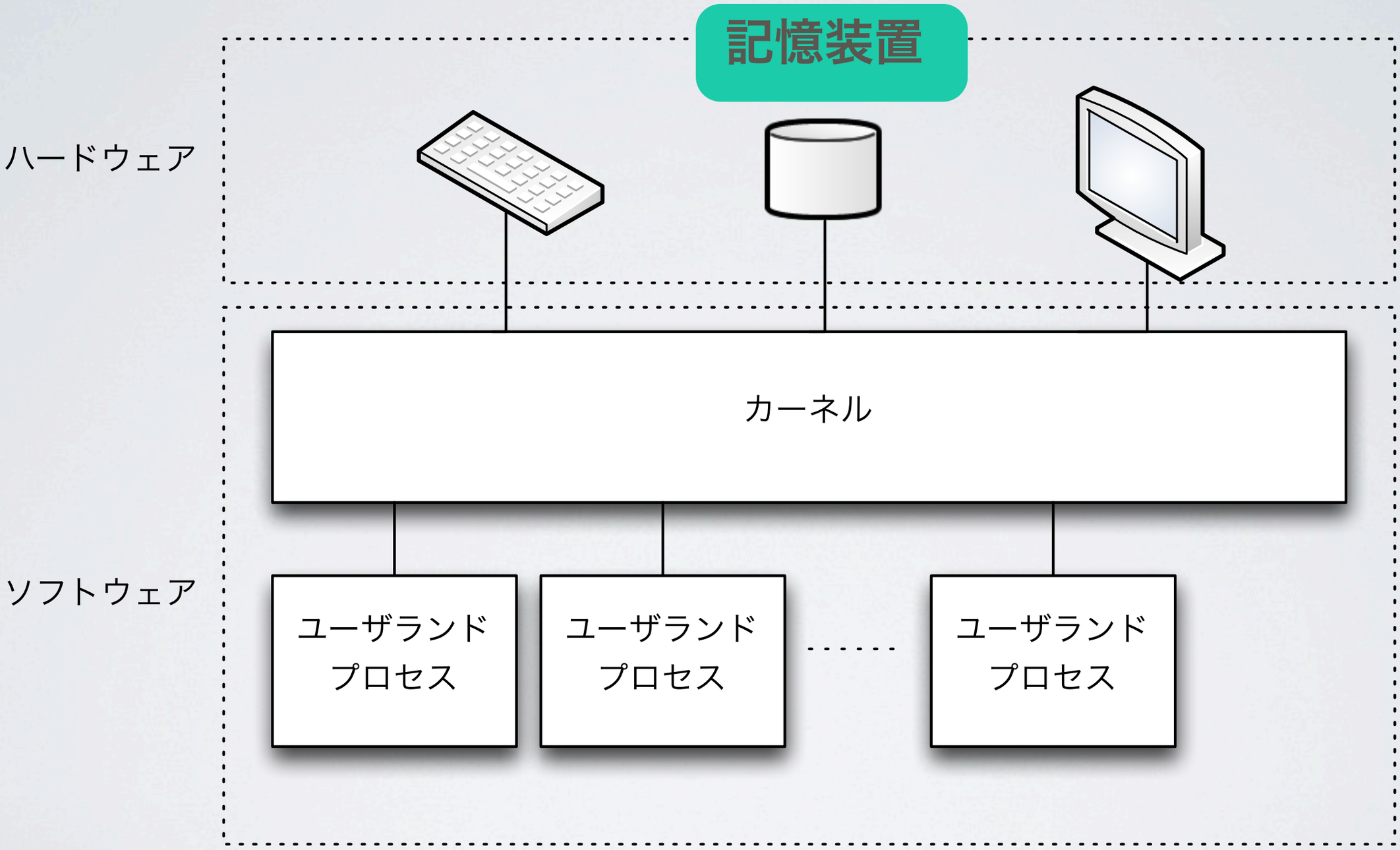
佐藤 広生 <hrs@FreeBSD.org>

- *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

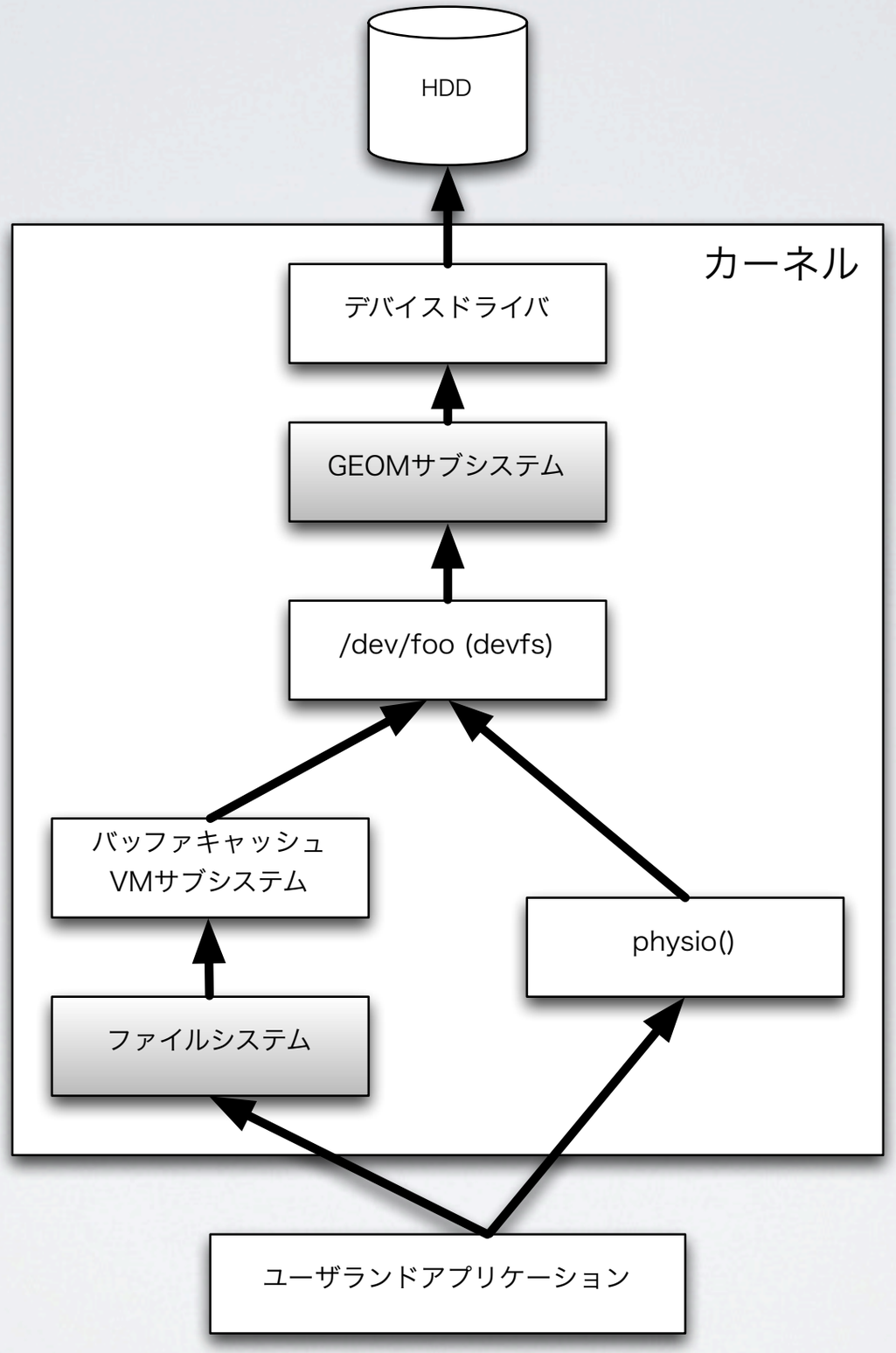
お話すること

- **ストレージ管理**
  - まずは基礎知識
- **GEOMフレームワーク**
  - 構造とコンセプト
  - 使い方
- **ファイルシステム**
  - 原理と技術的詳細を知ろう
  - UFS の構造
  - ZFS の構造（次回）
- **GEOM, UFS, ZFSの実際の運用と具体例（次回）**

復習：UNIX系OSの記憶装置



復習：UNIX系OSの記憶装置



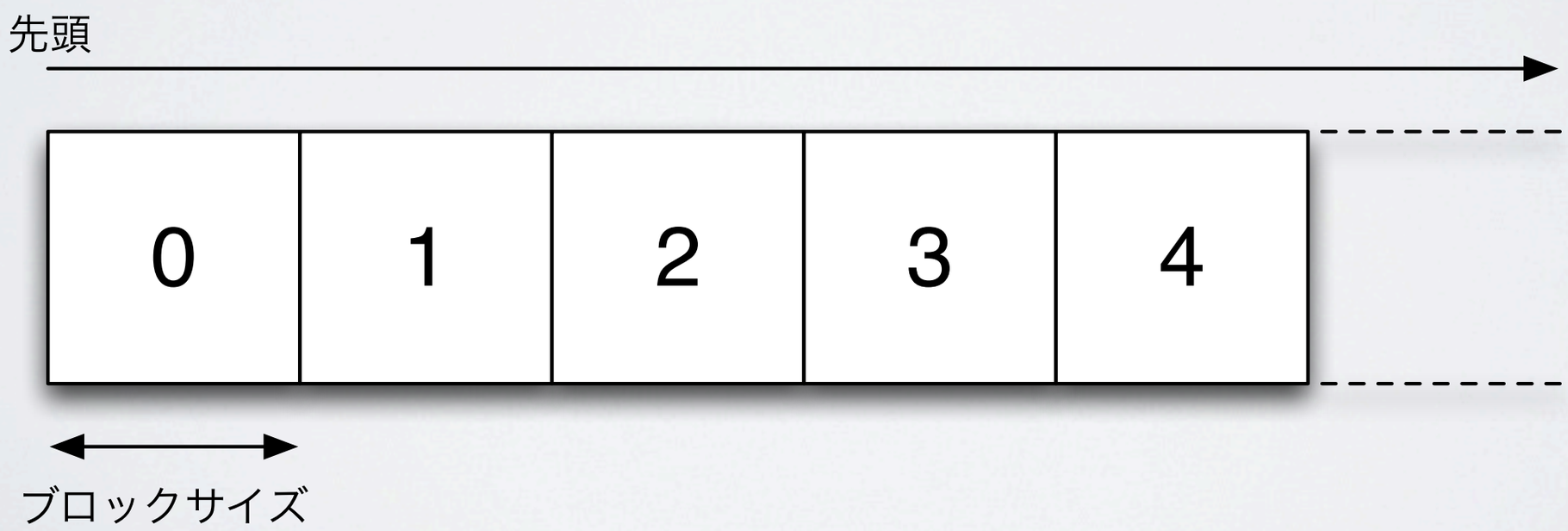
復習：UNIX系OSの記憶装置

- **記憶装置はどう見える？**
  - UNIX系OSでは、資源は基本的に「ファイル」
  - /dev/ada0 (SATA, SAS HDD)
  - /dev/da0 (SCSI HDD, USB mass storage class device)
  - 特殊ファイル（デバイスノード）

```
# ls -al /dev/ada*
crw-r-----  1 root  operator    0,  75 Jul 19 23:46 /dev/ada0
crw-r-----  1 root  operator    0,  77 Jul 19 23:46 /dev/ada1
crw-r-----  1 root  operator    0,  79 Jul 19 23:46 /dev/ada1s1
crw-r-----  1 root  operator    0,  81 Jul 19 23:46 /dev/ada1s1a
crw-r-----  1 root  operator    0,  83 Jul 19 23:46 /dev/ada1s1b
crw-r-----  1 root  operator    0,  85 Jul 19 23:46 /dev/ada1s1d
crw-r-----  1 root  operator    0,  87 Jul 19 23:46 /dev/ada1s1e
crw-r-----  1 root  operator    0,  89 Jul 19 23:46 /dev/ada1s1f
```

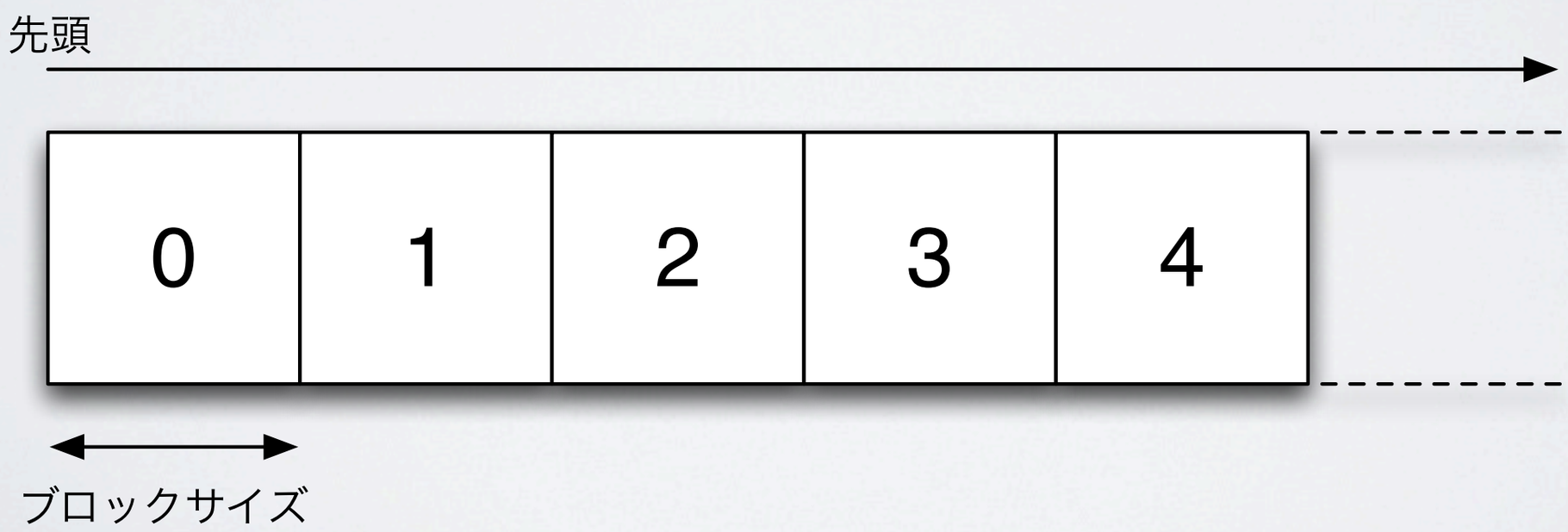
復習：UNIX系OSの記憶装置

- **記憶装置のデバイスノード**
  - 「ブロック」という記録単位が連続している構造（1次元配列）
  - ブロックにはアドレスがある（0 から連番）
  - ブロックにはサイズがある（典型的には 512 バイト）



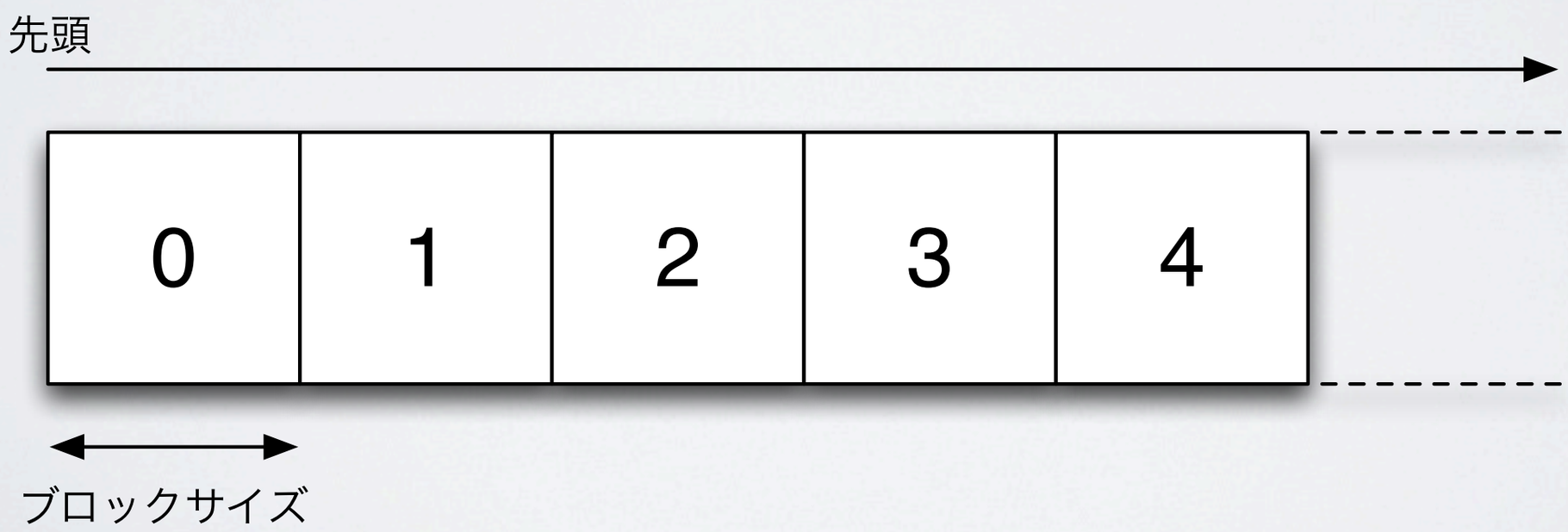
復習：UNIX系OSの記憶装置

- **記憶装置のデバイスノード**
  - I/O 操作は「write」か「read」
  - カーネルの中では struct bio と struct buf で管理



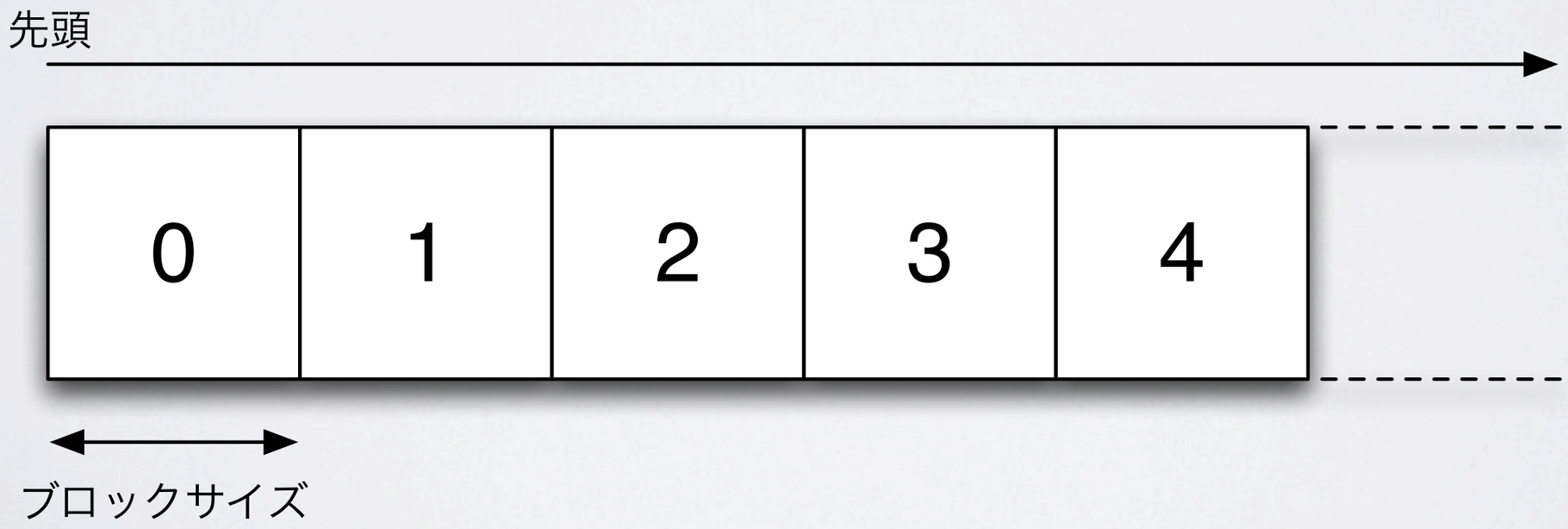
復習：UNIX系OSの記憶装置

- **記憶装置のデバイスノード**
  - 「cat /dev/ada0」ってやるとどうなる？
  - 先頭のブロックから順番に、記録されている内容が表示される



復習：UNIX系OSの記憶装置

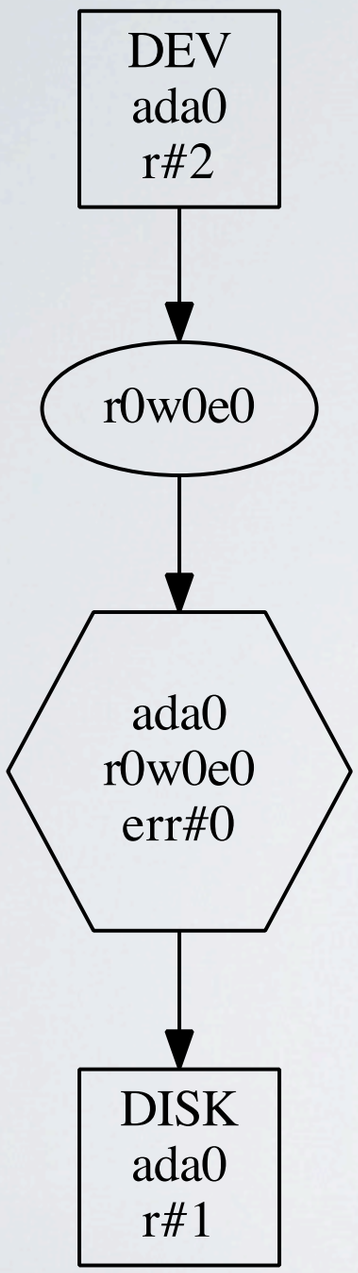
- **「キャラクタデバイス」と「ブロックデバイス」**
  - カーネルがキャッシュを提供するかどうかの区別
  - もうFreeBSDには「ブロックデバイス」がないので、この区別は忘れましょう。
  - /dev/rwd0 と /dev/wd0



GEOM とは？

- **記憶装置を管理するフレームワーク**
  - FreeBSD 5.0 より追加（DARPA 支援, 2002年）
  - 記憶装置の抽象化をより柔軟に
- まずは実体を見てみましょう

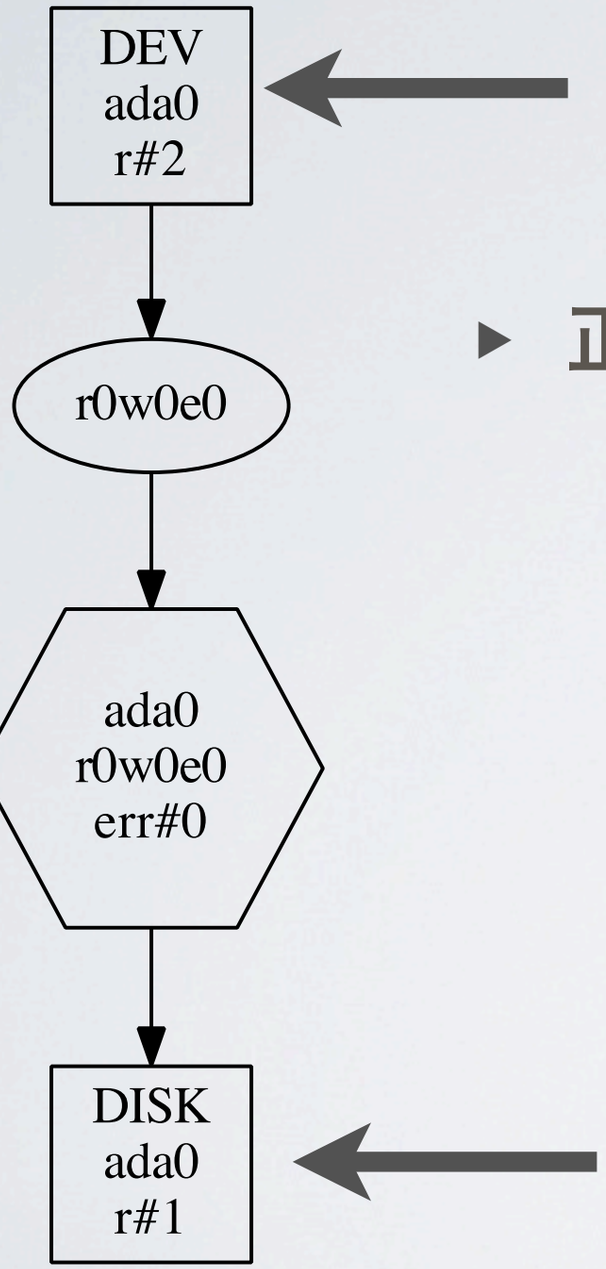
GEOM とは？




```
# ls -al /dev/ada*
crw-r-----  1 root  operator    0,  75 Jul 19 23:46 /dev/ada0
crw-r-----  1 root  operator    0,  77 Jul 19 23:46 /dev/ada1
crw-r-----  1 root  operator    0,  79 Jul 19 23:46 /dev/ada1s1
crw-r-----  1 root  operator    0,  81 Jul 19 23:46 /dev/ada1s1a
crw-r-----  1 root  operator    0,  83 Jul 19 23:46 /dev/ada1s1b
crw-r-----  1 root  operator    0,  85 Jul 19 23:46 /dev/ada1s1d
crw-r-----  1 root  operator    0,  87 Jul 19 23:46 /dev/ada1s1e
crw-r-----  1 root  operator    0,  89 Jul 19 23:46 /dev/ada1s1f
```

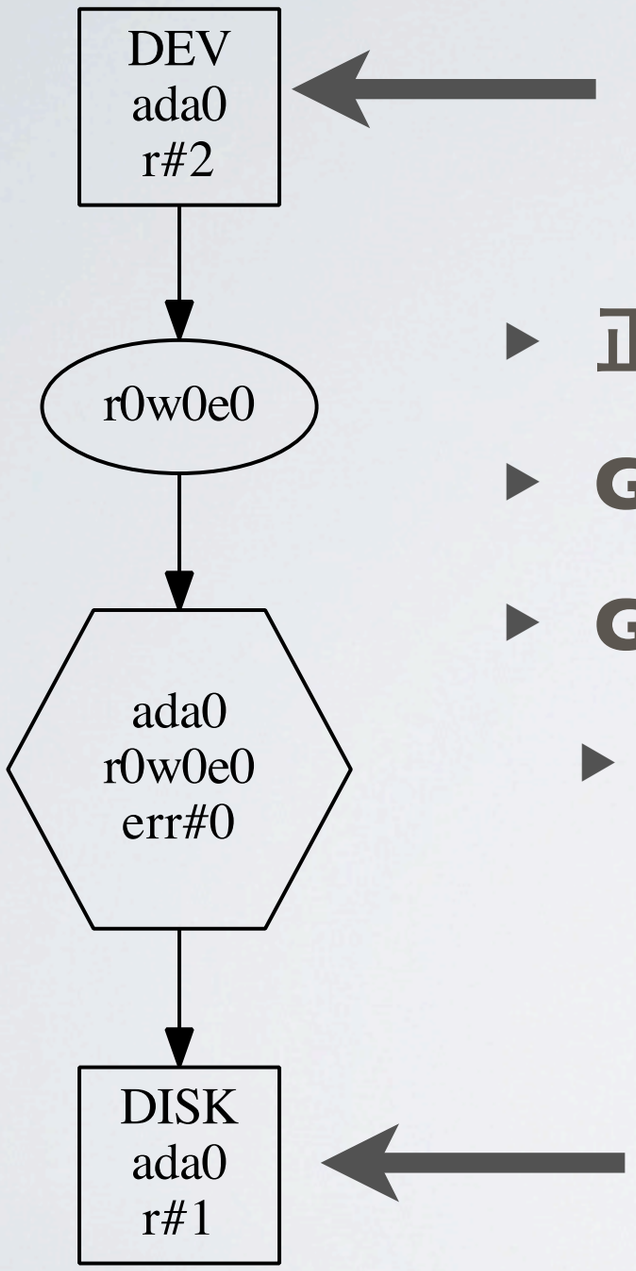
- /dev/ada0 は、左のような構造で認識されている

GEOM とは？



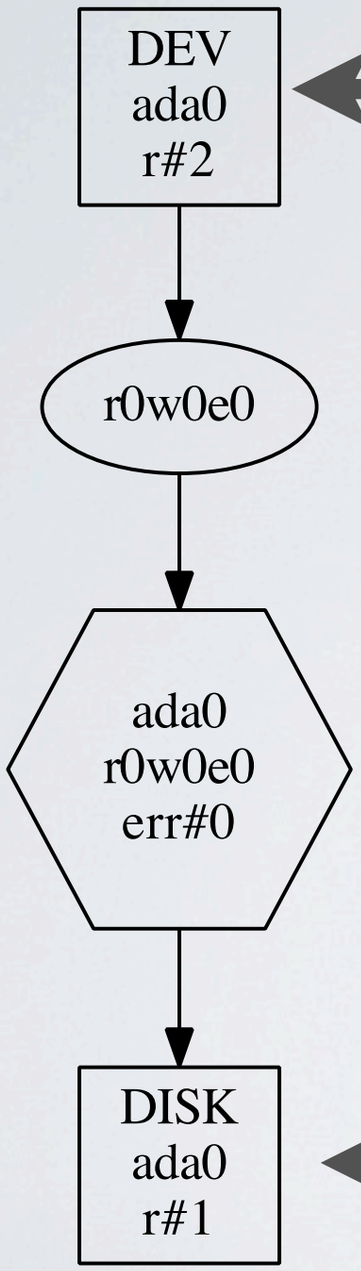
- 正方形が **GEOM** と呼ばれる部分

GEOM とは？



- **正方形が GEOM と呼ばれる部分**
- **GEOM には種類がある**
- **GEOMクラス**
  - データ入出力処理のこと（いろいろ種類あり）

GEOM とは？

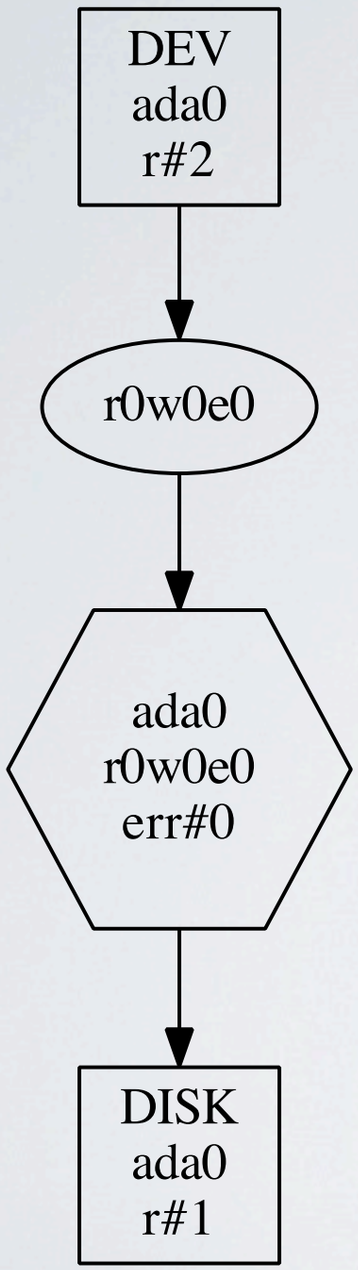


DEV class（デバイスノード担当）

- **正方形が GEOM と呼ばれる部分**
- **GEOM には種類がある**
  - **GEOMクラス**
    - データ入出力処理のこと（いろいろ種類あり）

DISK class（記憶装置とのやりとり担当）

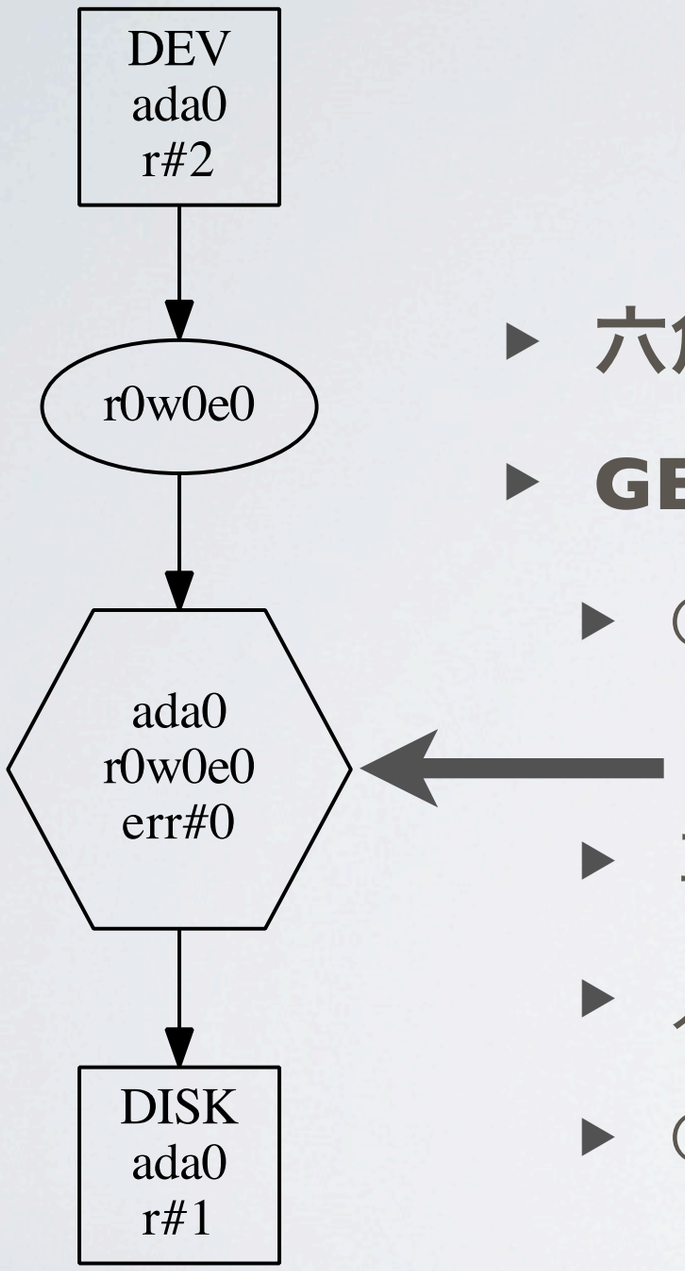
GEOM とは？



- **六角形はGEOMプロバイダ**

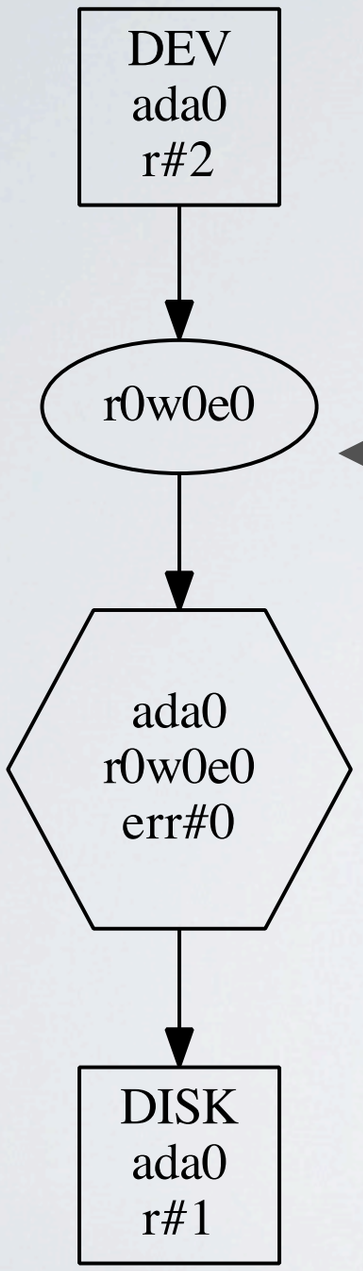
DISK class が持つ provider

GEOM とは？



- **六角形はGEOMプロバイダ**
- **GEOMプロバイダ**
  - GEOM が提供する接続点
  - コンシューマ（後述）と接続可能
  - 入出力処理をしたければここにつなげ！
  - GEOM クラスは 0 個以上のプロバイダを持つ

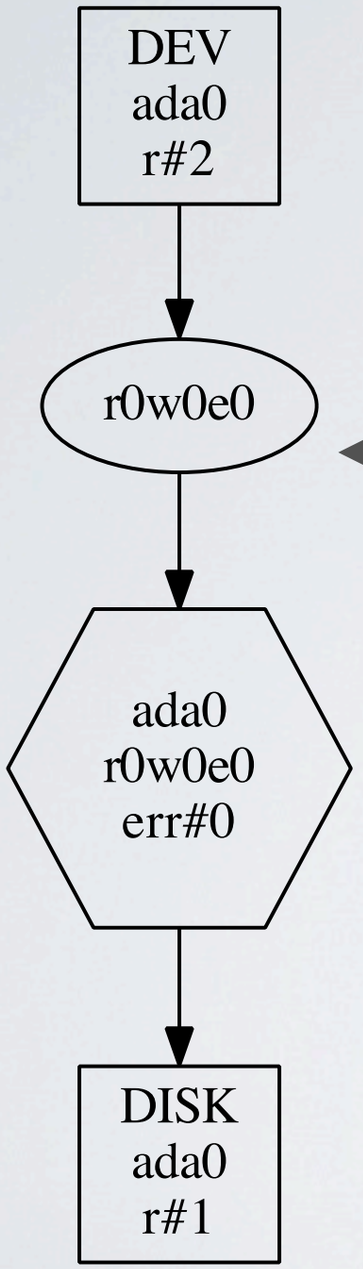
GEOM とは？



- **楕円形はGEOMコンシューマ**

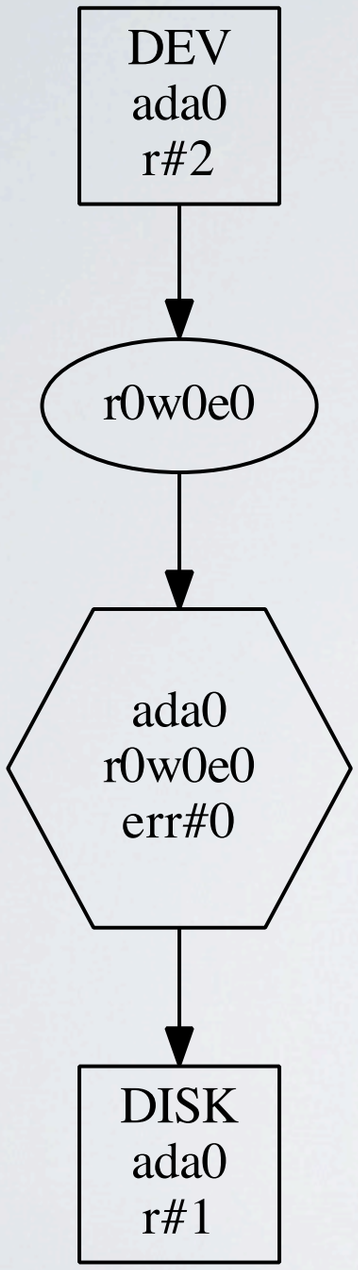
DEV class が持つ consumer

GEOM とは？



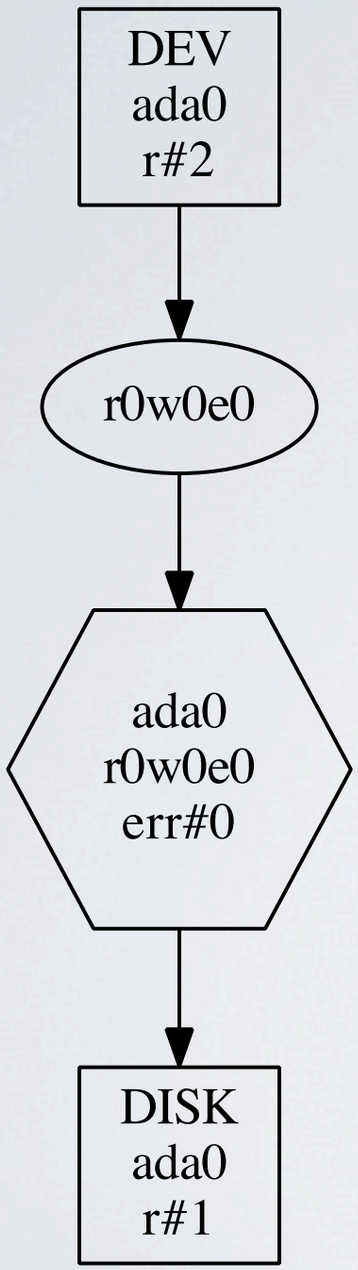
- **楕円形はGEOMコンシューマ**
- **GEOMコンシューマ**
  - GEOM が提供する接続点
  - プロバイダと接続可能
  - GEOM クラスは 0 個以上のコンシューマを持つ

GEOM とは？



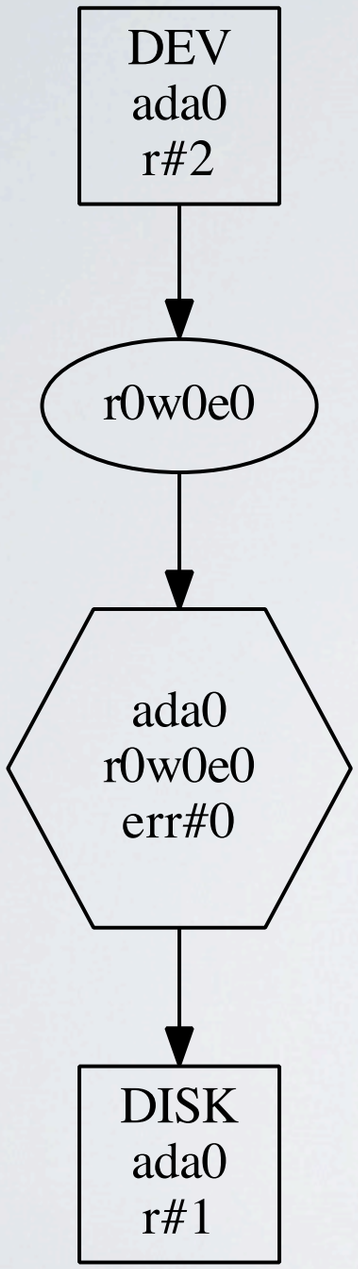
- **この接続はどういう意味？**
  - 「SATA HDD が /dev/ada0 を提供する」という意味
- SATA HDD とのデータの入出力処理: DISK
- /dev/ada0 の生成とデータの入出力処理: DEV
- 2 種類のデータ処理を行う GEOM を連結

GEOM とは？



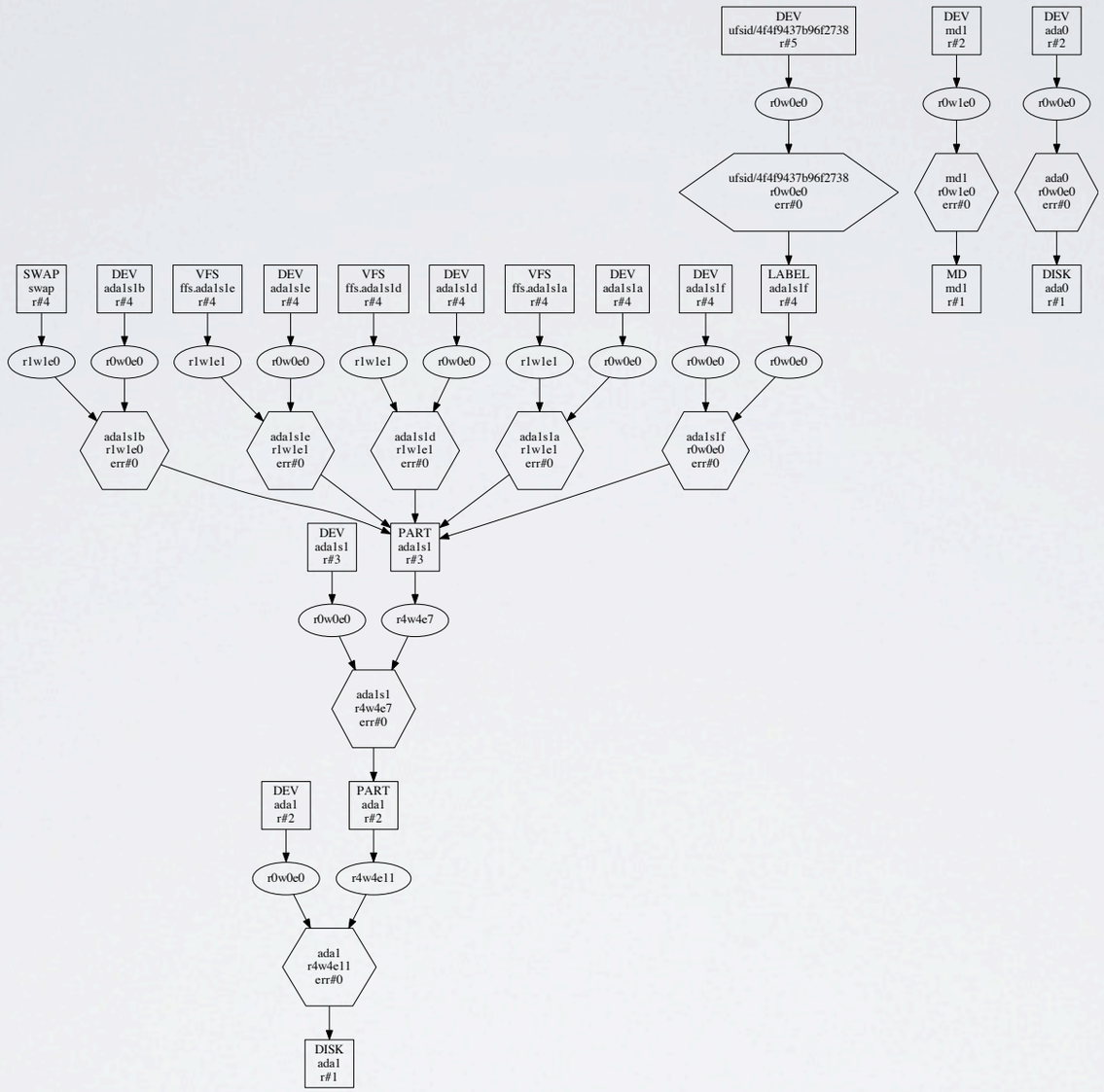
- **この接続は、誰がいつつくるの？**
  - GEOMクラスによって決まっている
  - DISKクラスのGEOM
    - 物理ディスクが認識された時に自動生成され、DEVクラスのGEOMを生成して連結する

GEOM とは？



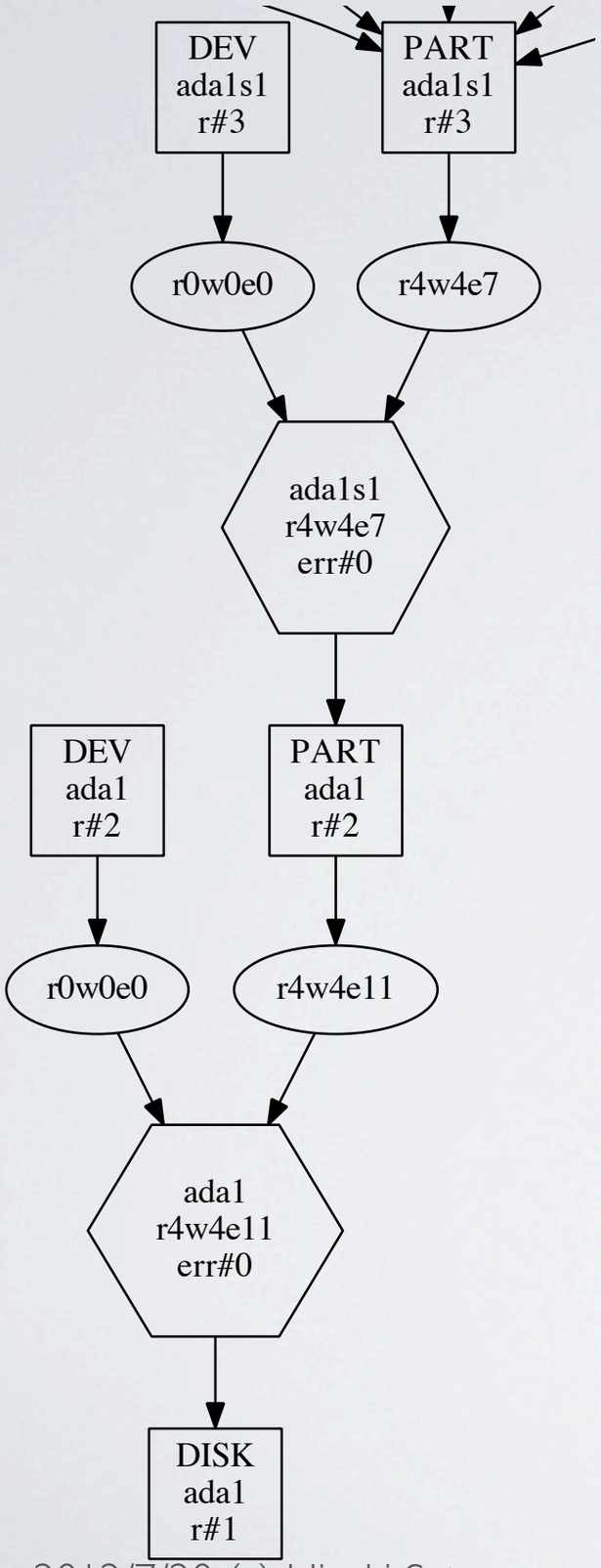
- **覚えよう**
  - FreeBSD では、すべてのストレージデバイスがGEOMの管理下に置かれている。
  - 見た目が伝統的なUNIX系OSと変わらないように自動的に処理されているので、意識しなくても大丈夫。

GEOM とは？



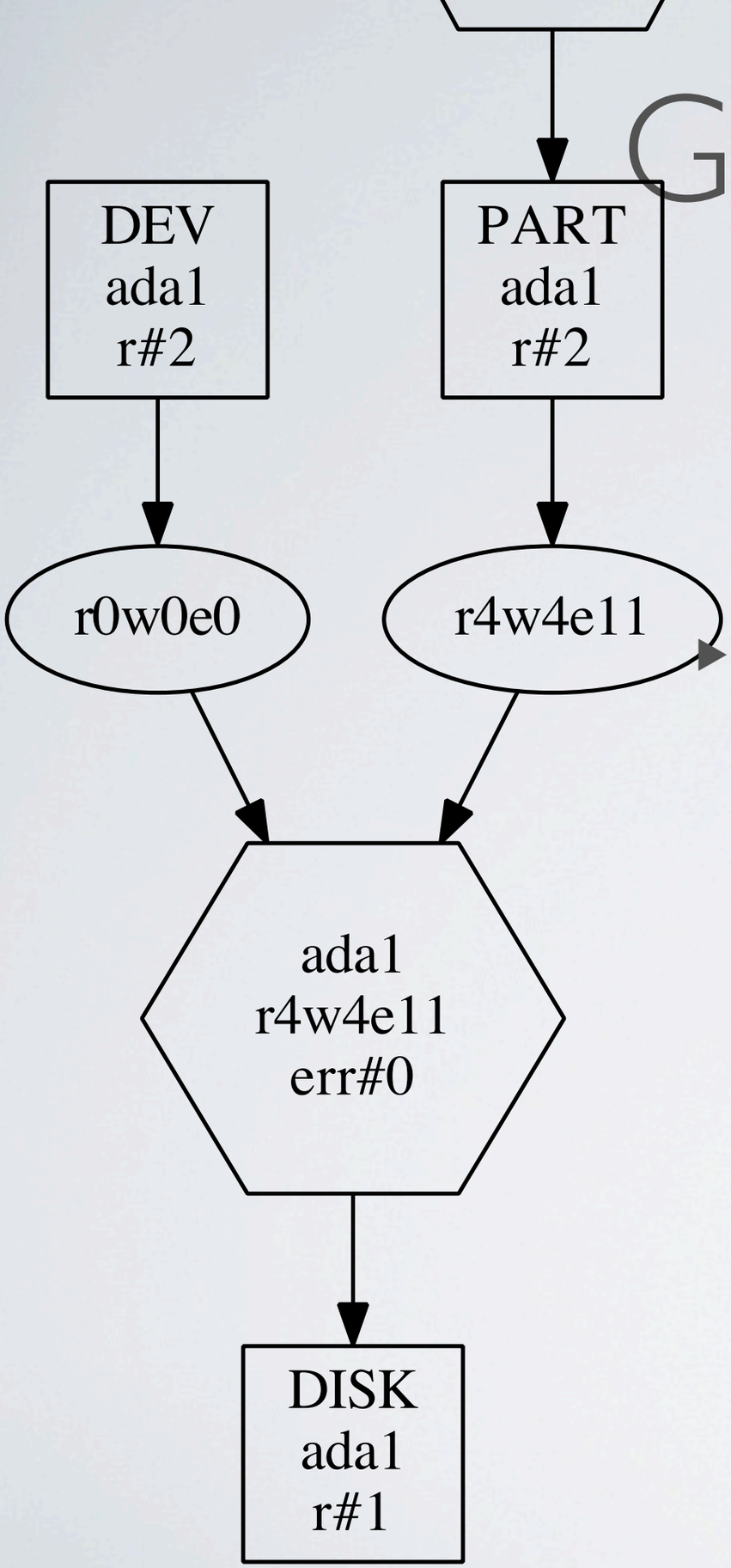
ada0 と ada1 と md0 があるマシンの
GEOM構造の一例

GEOM とは？



- **パーティションもGEOM**
- ada1 の DISK プロバイダから、
  PART クラスを経由して /dev/ada1s1 が生成

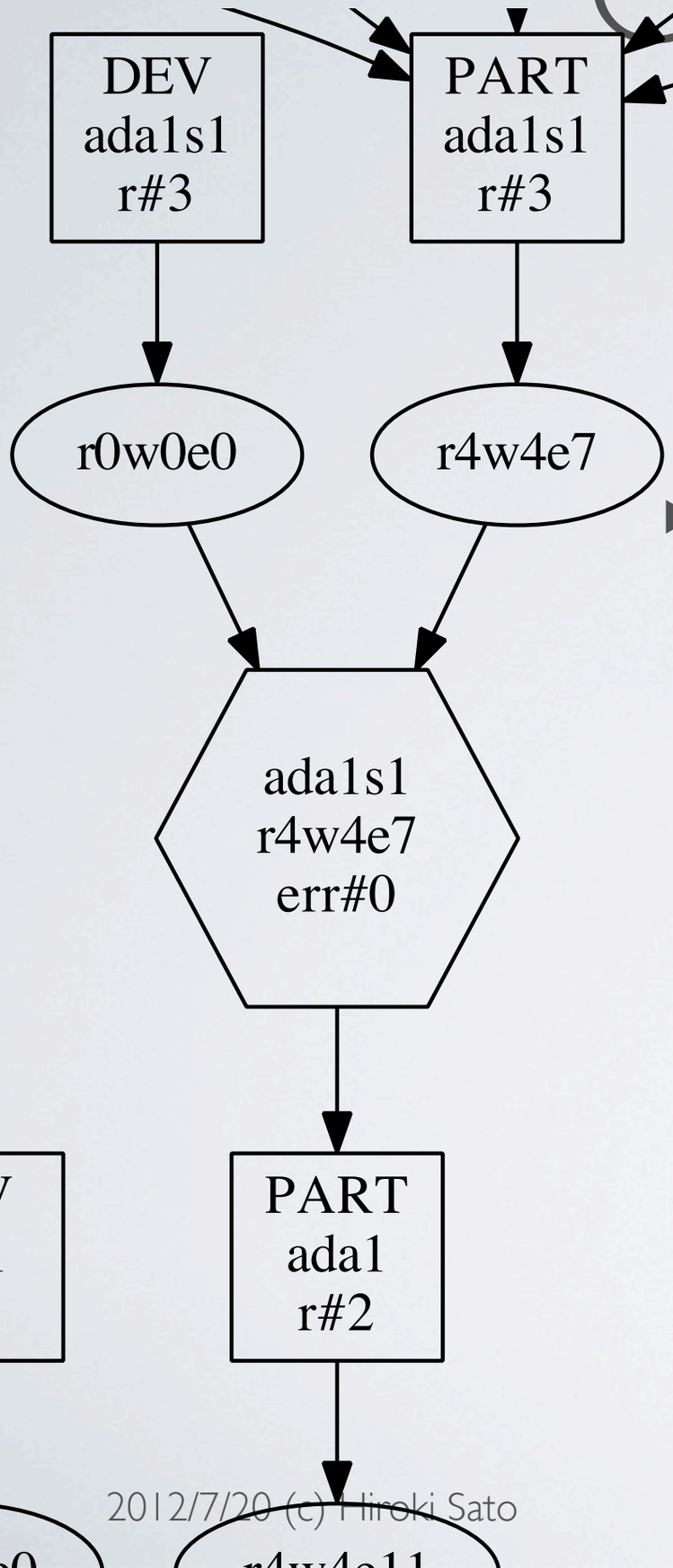
GEOM とは？



パーティションもGEOM

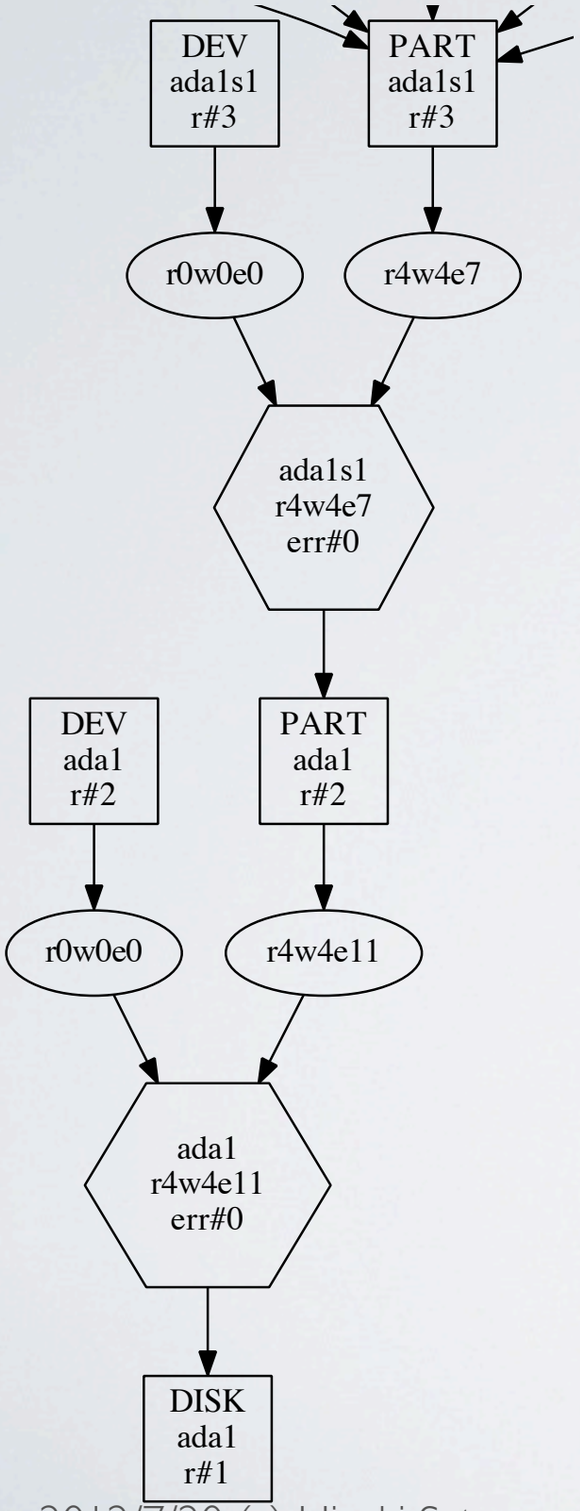
- ada1 の DISK プロバイダから、
  PART クラスを経由して /dev/ada1s1 が生成

GEOM とは？



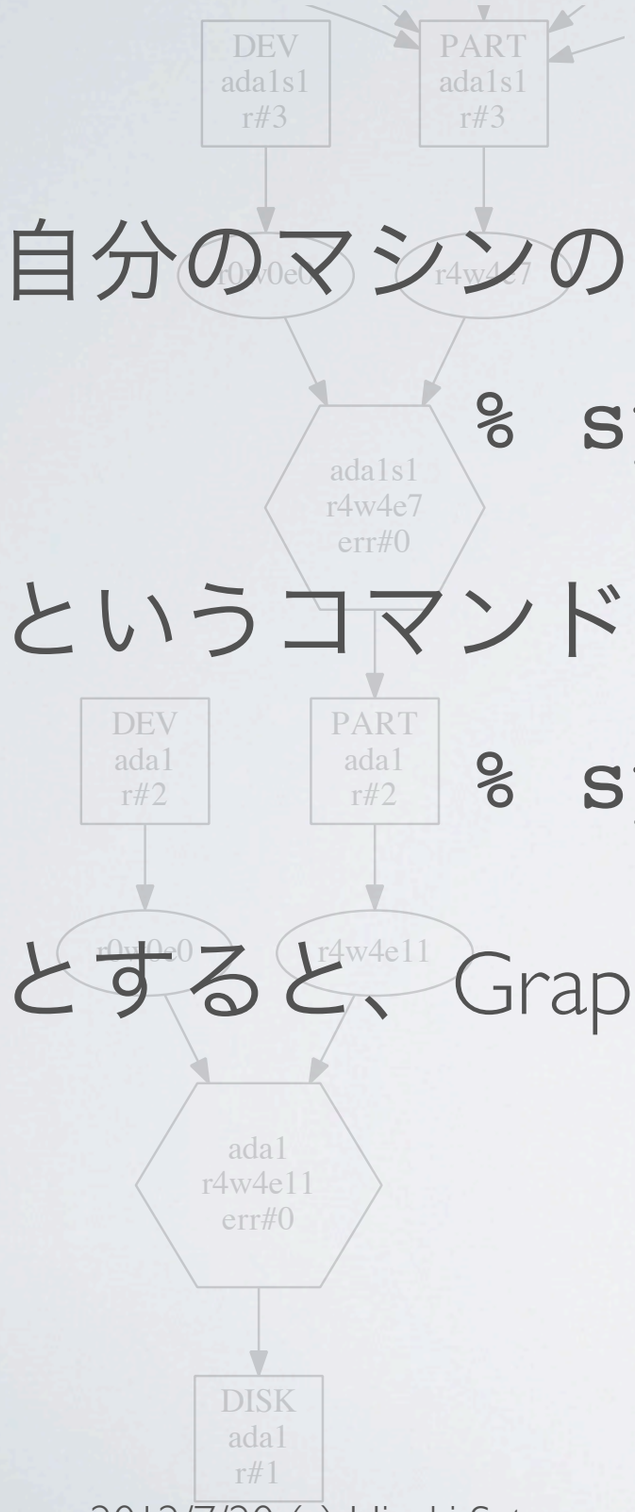
- パーティションもGEOM
- ada1 の DISK プロバイダから、
  PART クラスを経由して /dev/ada1s1 が生成

GEOM とは？



- **パーティションもGEOM**
  - ada1 の DISK プロバイダから、
    PART クラスを経由して /dev/ada1s1 が生成

GEOM とは？



自分のマシンのGEOM構造は、

```
% sysctl kern.geom.confxml
```

というコマンドで表示できます(XML形式)

```
% sysctl kern.geom.confdot
```

とすると、Graphviz の dot ファイルでも出してくれます。

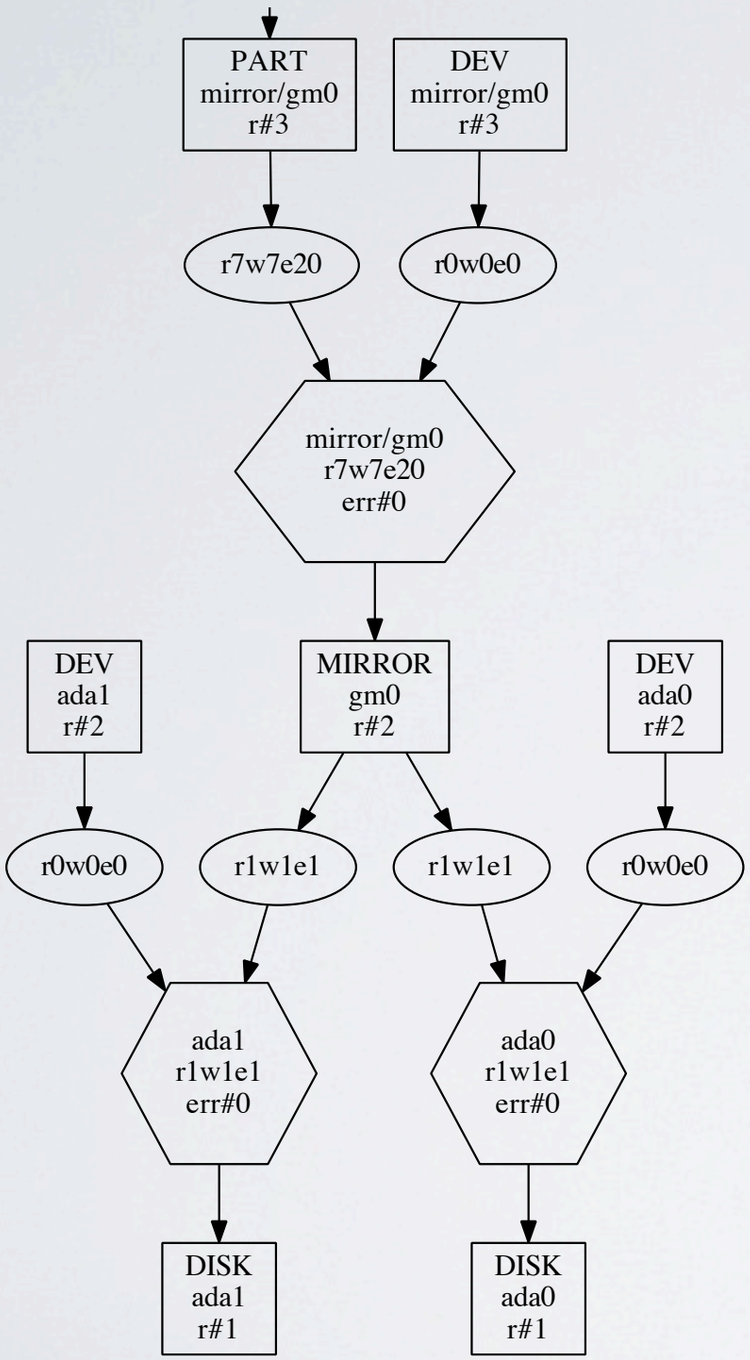
GEOM の応用

- **GEOMを意識的に使う**
  - いろいろなGEOMクラス＝いろいろなデータ処理
    - パーティション分離　(PARTクラス)
      - BSDラベル、GPT、Linux LVM、VTOC8、...
      - JBOD（CONCATクラス）
      - RAID0（STRIPEクラス）
      - RAID1（MIRRORクラス）
      - データ暗号化（ELIクラス、BDEクラス）
      - などなど
- イメージ：自分のやりたいデータ処理を積み重ねる！

MIRRORクラス

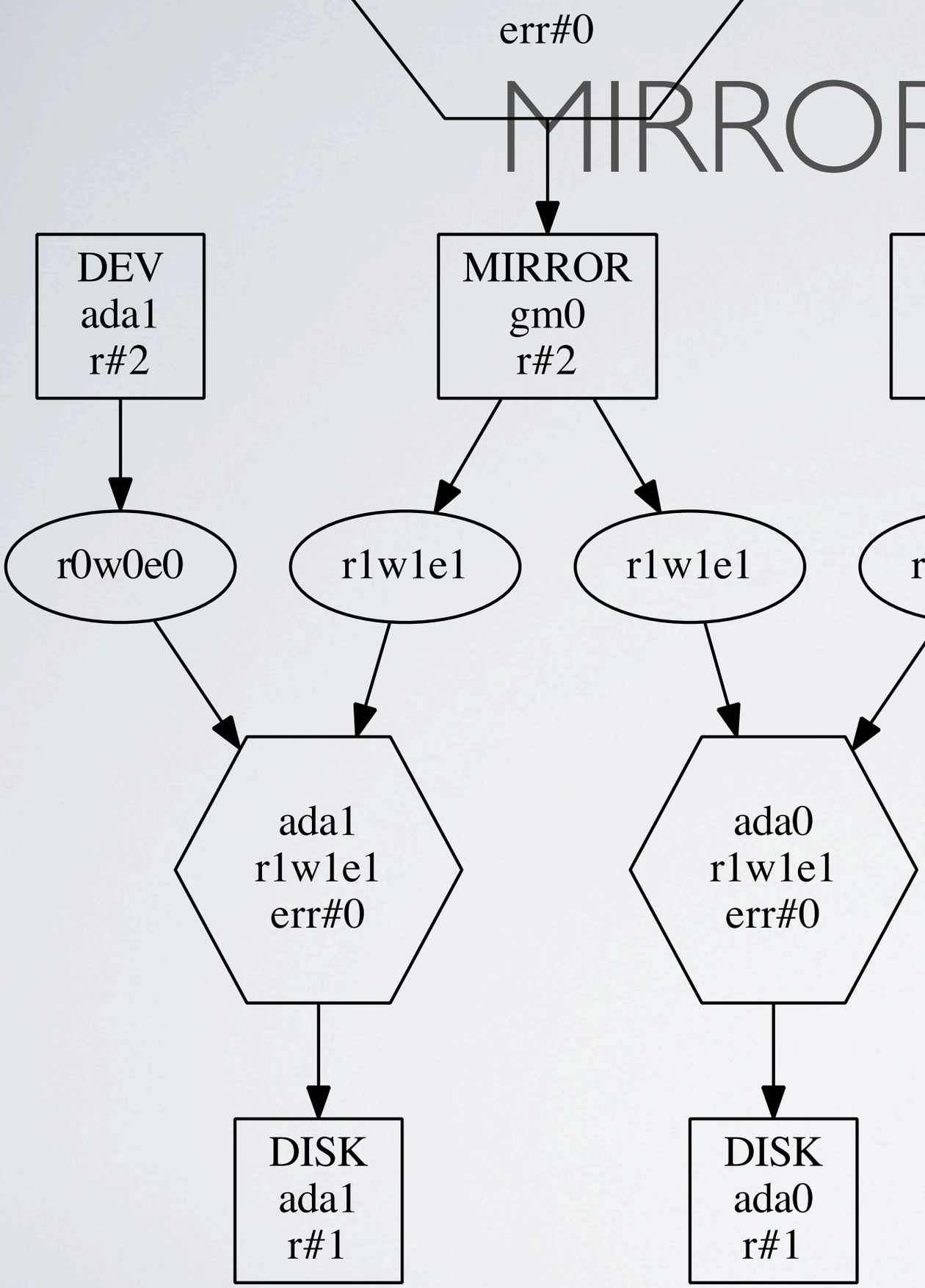
- **GEOMを使ってRAID1（ミラーリング）してみよう**
  - 2 台のHDD(ada0, ada1)があるマシン
  - ada0 と ada1 を MIRRORクラスに接続する

MIRRORクラス



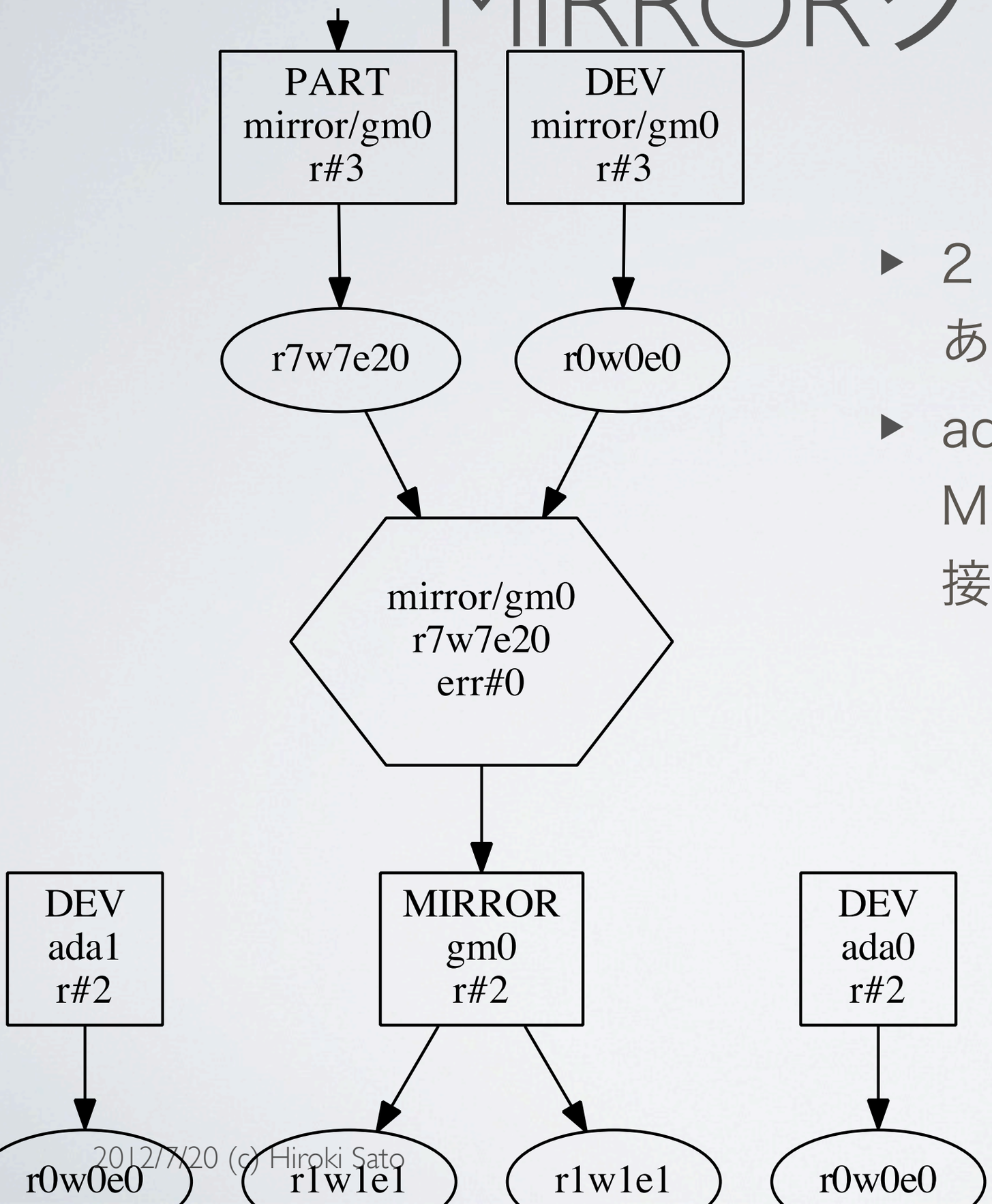
- 2 台のHDD(ada0, ada1)があるマシン
- ada0 と ada1 をMIRRORクラスのGEOMに接続する

MIRRORクラス



- 2 台のHDD(ada0, ada1)があるマシン
- ada0 と ada1 をMIRRORクラスのGEOMに接続する

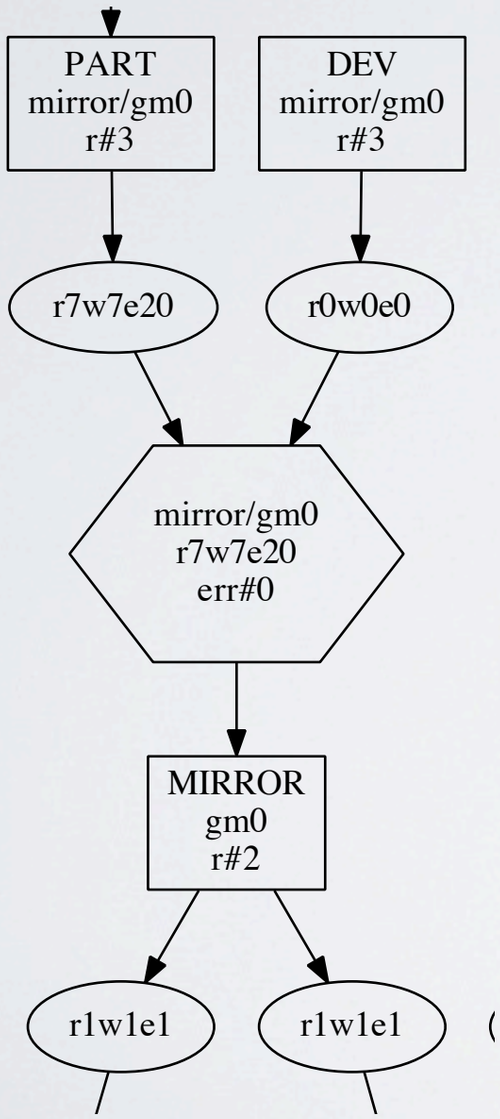
MIRRORクラス



- 2 台のHDD(ada0, ada1)があるマシン
- ada0 と ada1 をMIRRORクラスのGEOMに接続する

MIRRORクラス

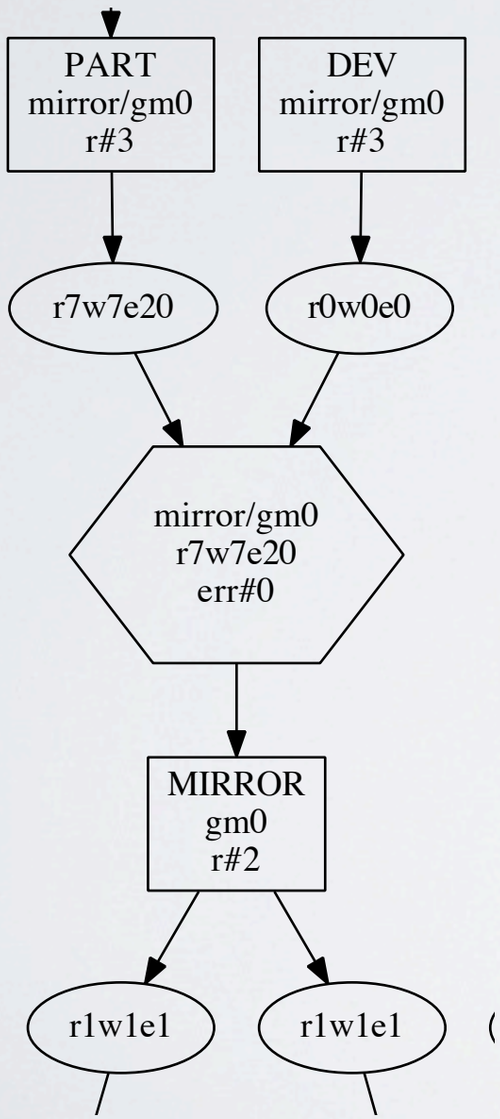
```
# ls -al /dev/mirror
crw-r-----  1 root  operator    0, 100 Jul  4 02:11 /dev/mirror/gm0
crw-r-----  1 root  operator    0, 101 Jul  4 02:11 /dev/mirror/gm0s1
```



- MIRRORクラスのGEOMは、2 個以上のGEOMコンシューマを持っている。
- 3 台の HDD（= DEVクラス）をつなげば、全部ミラーリングされる
- GEOMプロバイダなら何でも接続可能！
  - USBメモリとHDD
  - パーティション1個と別のディスク全体
  - などなど

MIRORクラス

```
# ls -al /dev/mirror
crw-r-----  1 root  operator    0, 100 Jul  4 02:11 /dev/mirror/gm0
crw-r-----  1 root  operator    0, 101 Jul  4 02:11 /dev/mirror/gm0s1
```



- **この接続は、誰がいつつくるの？**
  - カーネルが記憶装置の最終ブロックを読んで、その設定を使って自動でつくる
- この動作は tasting と呼ばれ、他のGEOMクラスでも使われている
- 設定は gmirror(8)コマンドで書き込める

やってみよう

```
# gmirror load
# gmirror label gm0 ada0 ada1
GEOM_MIRROR: Device mirror/gm0 launched (2/2).
# gmirror status
      Name    Status  Components
mirror/gm0  DEGRADED  ada1 (ACTIVE)
                      ada0 (SYNCHIRONIZING, 64%)
# ls -al /dev/mirror
total 0
crw-r-----  1 root  operator    0, 120 Jul 20 03:01 gm0
# gmirror status
      Name    Status  Components
mirror/gm0   COMPLETE  ada1 (ACTIVE)
                       ada0 (ACTIVE)
```

やってみよう

- gmirror load でカーネルモジュールをロード
- gmirror label で設定データ書き込み(gm0という名前)
- カーネルが自動的にMIRRORクラスのGEOMを生成、/dev/mirror/gm0 が出現
- MIRRORクラスは自動的にミラー処理開始

  gmirror status で状況確認ができる。
- あとは /dev/mirror/gm0 がふつうのHDDと同じだと思ってよし。
- 標準カーネルにはMIRRORクラスが入っていないので、

  /boot/loader.confにgeom_mirror_load="YES" を追加しておくこと。

やってみよう

- ディスク(例えばada1)が壊れたら、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
- gmirror label は既存のMIRRORクラスのGEOMには適用できない
- gmirror stop gm0 とすると、MIRRORクラスが停止する
- gmirror clear ada0 とすると、ada0 に書き込まれた設定が消える

GEOM の使い方

- **基本的に自動で動く!**
  - 設定データ（GEOMメタデータ）を書き込んでおくと、tasting によって適切なGEOMが自動生成される
  - 処理に応じたデバイスノードが生える、というイメージ
    - /dev/mirror/gm0
    - /dev/ada0
    - /dev/ada1
    - gm0 からアクセスすると、ada0 と ada1 に分配される
- **自由に組み合わせが可能！**

GEOM の使い方

- **GEOMメタデータの書き込み**
  - 専用コマンド
    - gmirror (MIRRORクラス)
    - gstripe (STRIPEクラス)
    - などなど
  - FreeBSD 以外で書き込まれる可能性があるもの
    - RAIDクラス (一部のBIOSがサポートしているRAID機能)
    - パーティション情報
    - などなど
- 実際の運用事例については次回に紹介します

GEOM の特徴まとめ

- **記憶装置と、データ処理の抽象化**
  - 処理の結果は新しいデバイスノードとして見える
    - /dev/ada0 と /dev/ada1 のミラー = /dev/mirror/gm0
    - /dev/ada0 と /dev/ada1 の連結 = /dev/concat/gc0
    - /dev/ada0 を暗号化した場合の平文アクセス = /dev/ada0.eli
  - 階層構造をとることが可能
- **設定方法はGEOMクラスによって異なる**
  - 何もせずに自動的に働くもの、設定を読んで働くもの、コマンドを実行して初めて働くもの、等々
- **FreeBSD のストレージ管理の基礎**

ファイルシステム

- GEOMが提供するもの＝記憶装置に対応するデバイスノード
- 一次元のデータブロック配列なので、そのままでは使いにくい
- どうする？
  - UNIX系OSでは、ファイルシステムに対応づけを行って使うことが多い

UNIX系OSのファイルシステム

- **ファイルシステムとは**
  - データへのアクセス手順を提供
  - データ格納領域を指定するための名前空間を提供
  - いわゆる「名前＝内容」のペアが保存できる空間
- 必ずしも記憶装置が関係しているわけではない

UNIX系OSのファイルシステム

- **ファイルシステムとは**
  - データへのアクセス手順を提供
  - データ格納領域を指定するための名前空間を提供
  - いわゆる「名前＝内容」のペアが保存できる空間
  - 必ずしも記憶装置が関係しているわけではない
- **UNIX系OSでは**
  - 単一ルートを持つ木構造の名前空間を使う
    - いわゆる “/” ディレクトリが必ず1個ある
  - アクセスはシステムコール(open や write)で行う

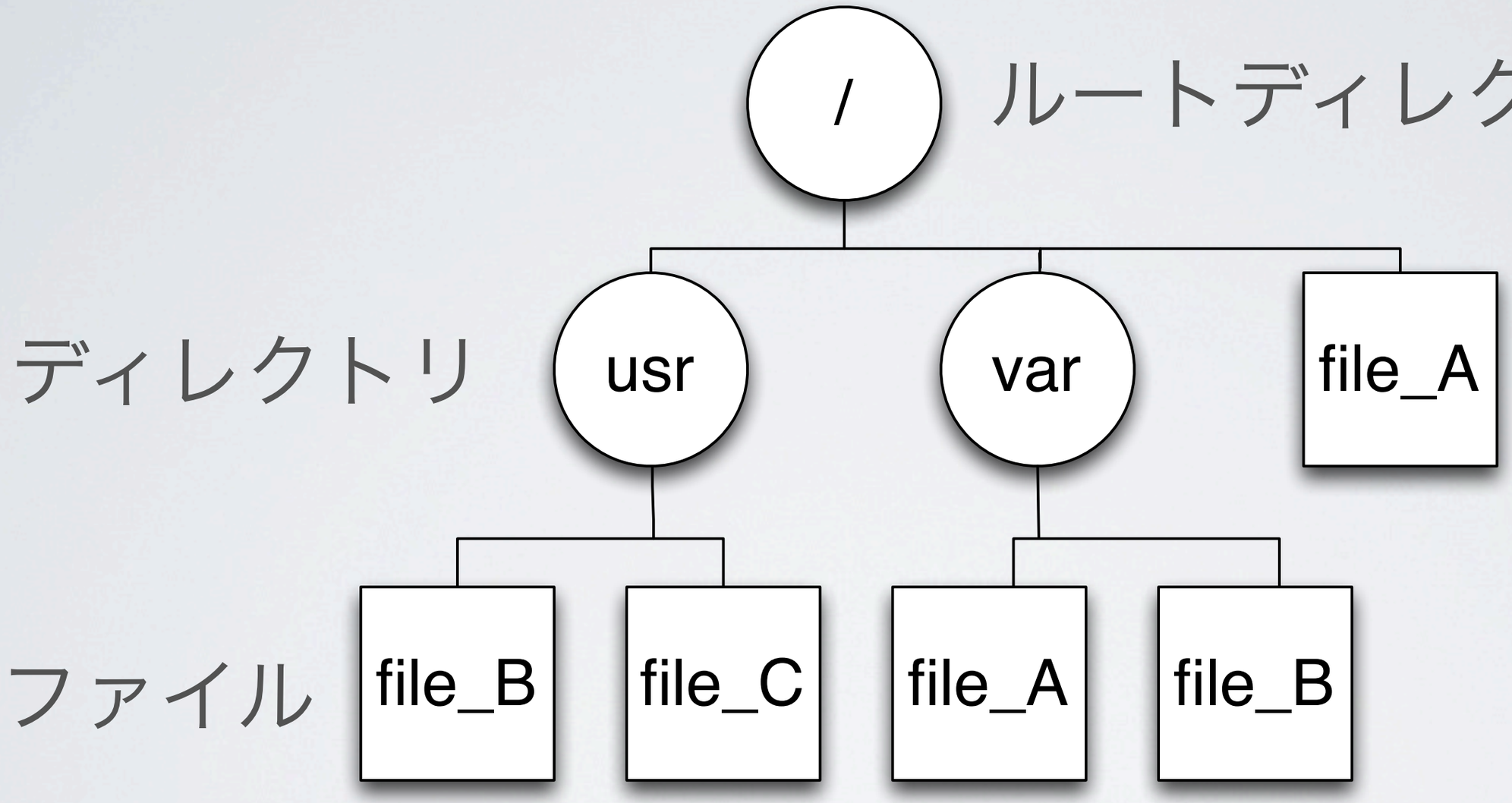
UNIX系OSのファイルシステム

- **カーネルが起動する時、ファイルシステムは必要か？**
  - なくてもカーネルは仕事ができる
  - ユーザランドプログラムはほぼ何もできない
    - UNIX系OSの資源はファイル単位で管理されているから

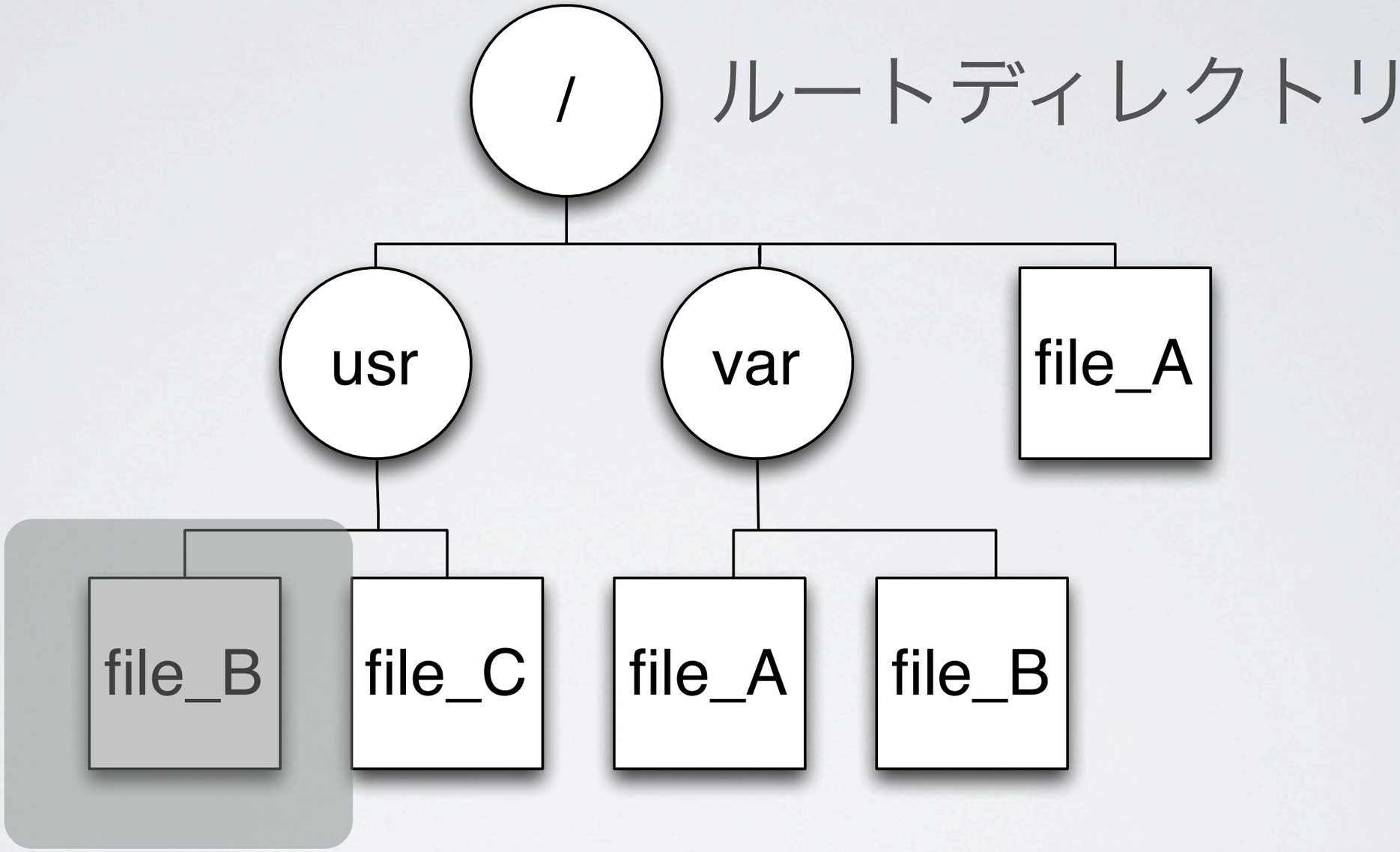
UNIX系OSのファイルシステム

- **カーネルが起動する時、ファイルシステムは必要か？**
  - なくてもカーネルは仕事ができる
  - ユーザランドプログラムはほぼ何もできない
    - UNIX系OSの資源はファイル単位で管理されているから
- **起動の最終段階で、”/” を用意する**
  - ファイルシステムの構造を記憶装置に書き込んでおき、
    カーネルが起動できたら、その構造を”/”に対応づける
  - 記憶装置にあるファイルシステム構造を、
    カーネルが管理するファイルシステムに対応づける操作を
    「マウント」と呼ぶ

ファイルシステムの構造

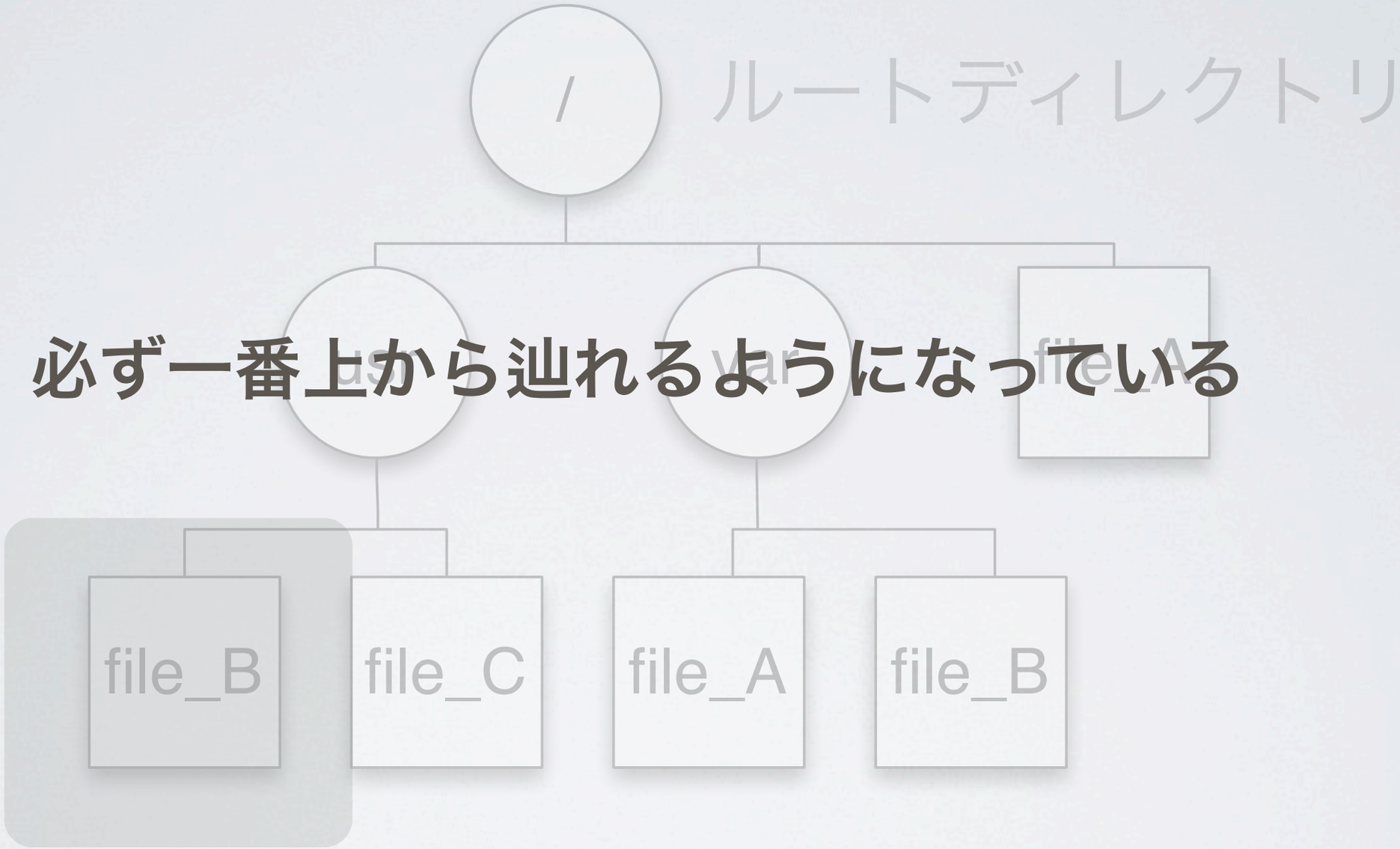


ファイルシステムの構造



`/usr/file_B` (名前)

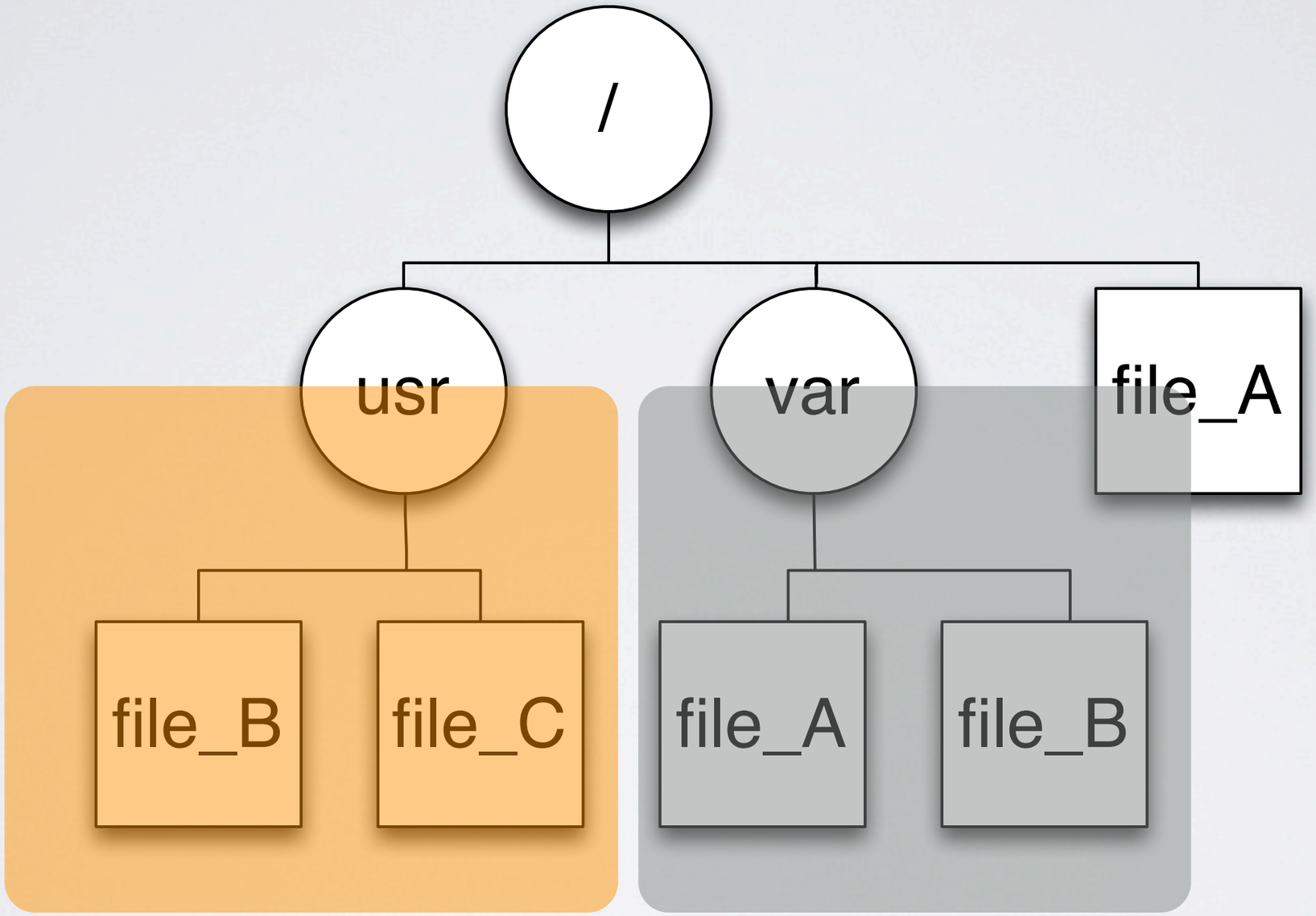
ファイルシステムの構造



必ず一番上から辿れるようになっている

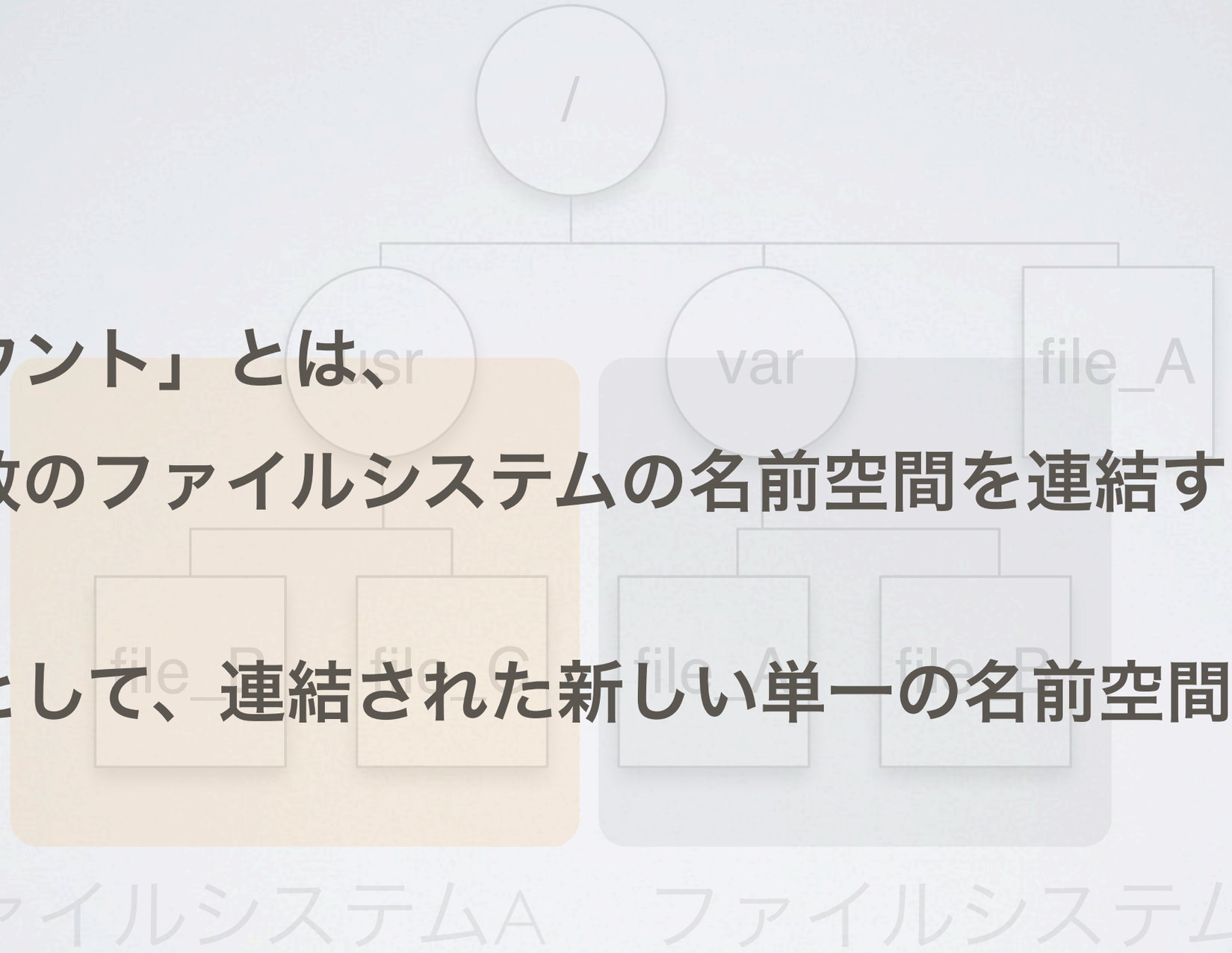
/usr/file_B

ファイルシステムの構造



ファイルシステムA　ファイルシステムB

ファイルシステムの構造

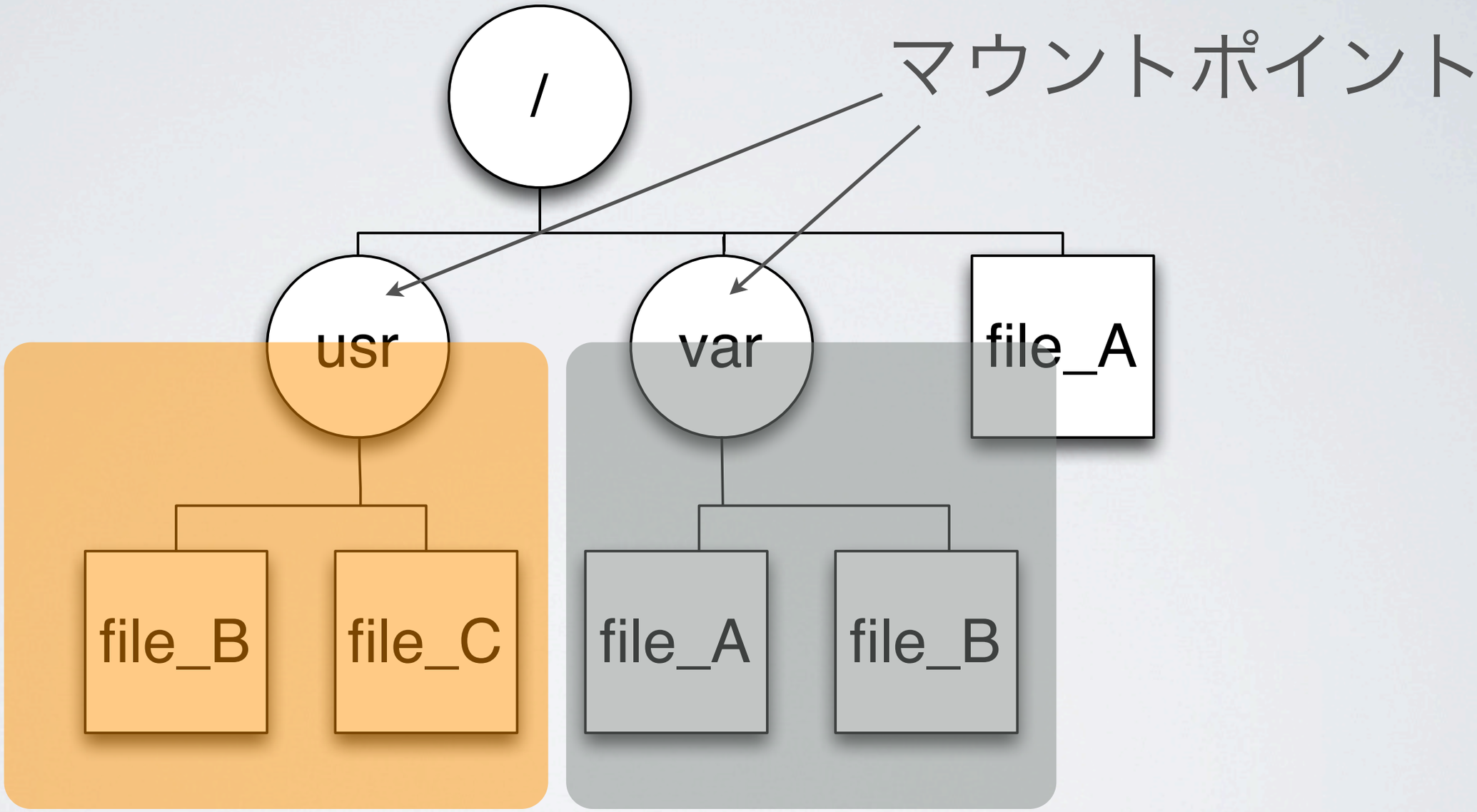


「マウント」とは、

複数のファイルシステムの名前空間を連結すること

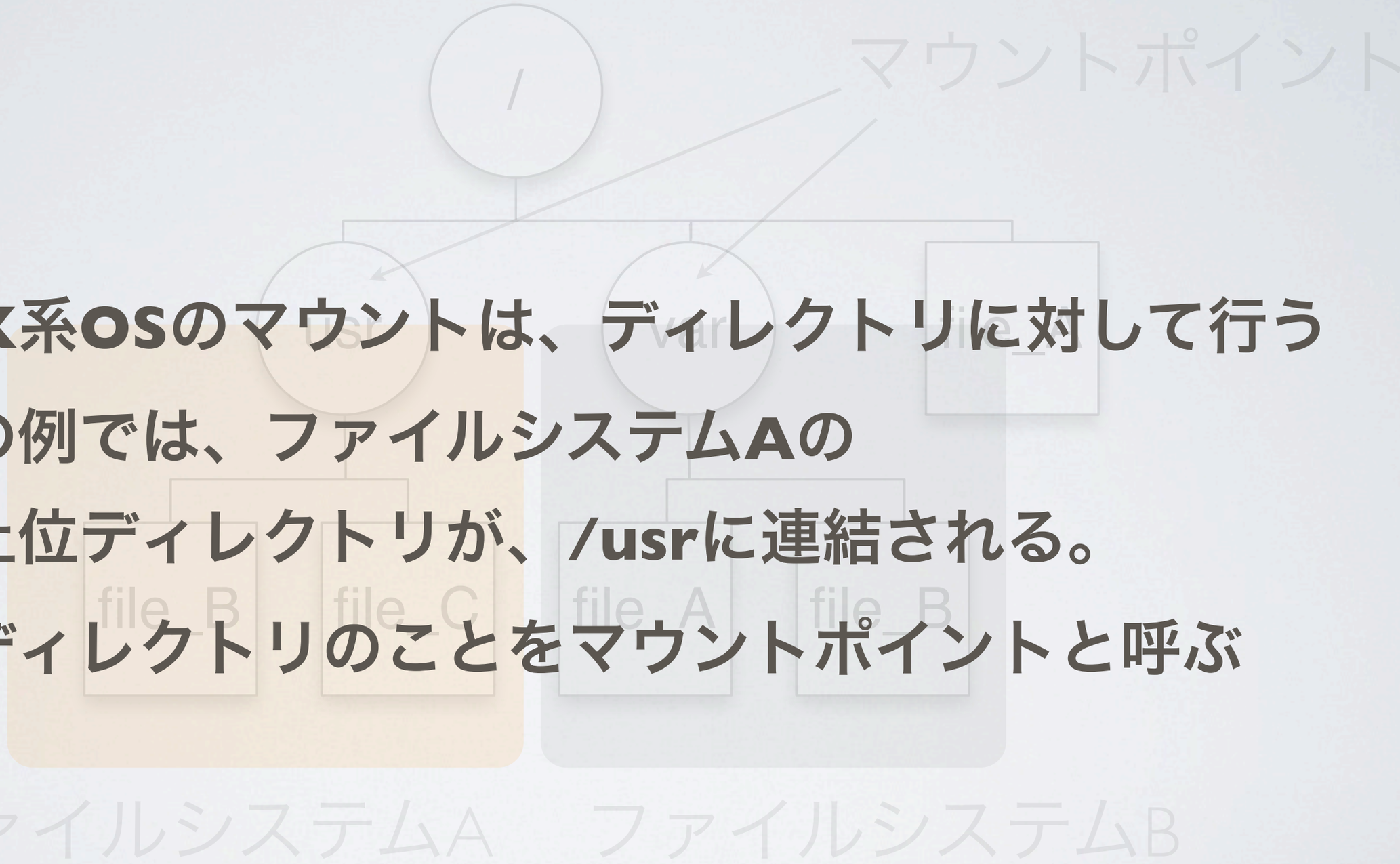
結果として、連結された新しい単一の名前空間ができる

ファイルシステムの構造



ファイルシステムA　ファイルシステムB

ファイルシステムの構造



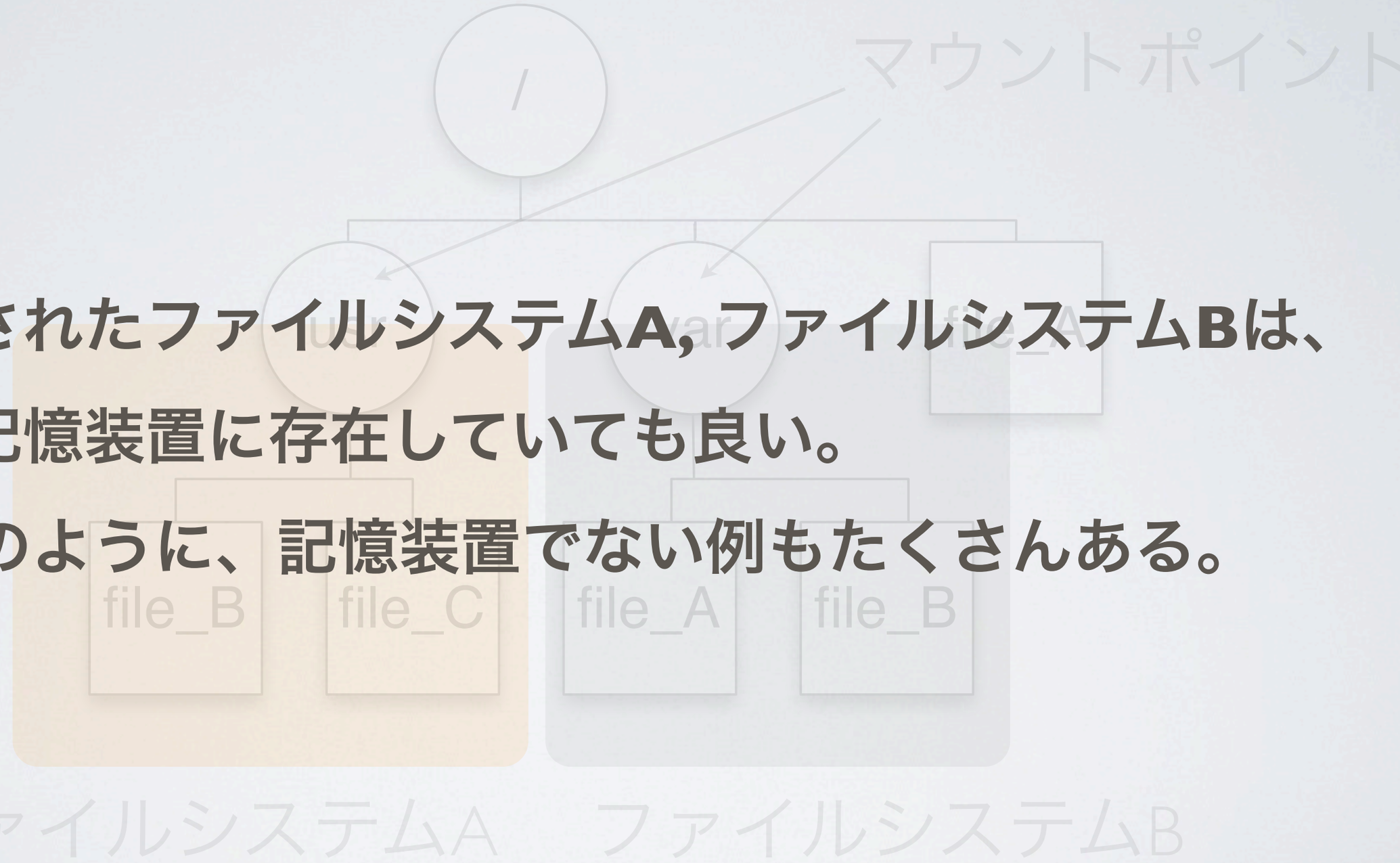
UNIX系OSのマウントは、ディレクトリに対して行う

この例では、ファイルシステムAの

最上位ディレクトリが、/usrに連結される。

このディレクトリのことをマウントポイントと呼ぶ

ファイルシステムの構造



連結されたファイルシステムA,ファイルシステムBは、どの記憶装置に存在していても良い。

NFSのように、記憶装置でない例もたくさんある。

ためしてみよう

- ファイルシステムをつくってマウントする
  - `mdconfig -a -t swap -u 0 -s 10m`
  - `newfs /dev/md0`
  - `mount /dev/md0 /mnt`

```
# mdconfig -a -t swap -u 0 -s 10m
# newfs /dev/md0
/dev/md0: 10.0MB (20480 sectors) block size 32768, fragment size 4096
    using 4 cylinder groups of 2.53MB, 81 blks, 256 inodes.
super-block backups (for fsck -b #) at:
 192, 5376, 10560, 15744
# mount /dev/md0 /mnt
# mount
    :
    :
/dev/md0 on /mnt (ufs, local)
```

ファイルシステムの構造

- **記憶装置上のファイルシステムの構造とは？**
  - newfs がつくり、mount で連結されるもの
- 「名前＝内容」のペア構造を保存したい
- 保存する場所は一次元のデータブロック配列
- いろいろな書き込み方がある
  ＝いろいろなファイルシステムが存在

考えてみよう

- 名前＝内容が記録できる構造を想像してみよう

考えてみよう

- 名前＝内容が記録できる構造

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

⋮

/dev/ada0 の構造

考えてみよう

- 名前＝内容が記録できる構造

F	F	F	F	F
F	F	F	F	F
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

ファイル名を保存する

ブロックを予め決める

考えてみよう

- 名前＝内容が記録できる構造

a=10	F	F	F	F
F	F	F	F	F
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

ファイル名:“a”

内容: アドレス10

考えてみよう

- 名前＝内容が記録できる構造

大きいファイルは

複数のブロックを使って記録

考えてみよう

- **この方式の欠点**
  - ファイルの削除と追加を繰り返すと、内容に使っていたブロックの隙間がたくさんできてしまう。
  - ディレクトリ構造がない。「ファイル＝内容」だけ。

UFSの構造

- **UNIXが採用したデータ構造**

先頭

出発点は同じ

UFSの構造

- **UNIXが採用したデータ構造**

0	1	2	3	4
5	6	7	8	9
I	I	I	I	I
I	I	I	I	I
20	21	22	23	24
25	26	27	28	29

⋮

領域の中間点付近に、
管理情報用の専用領域を確保
（inode と呼ぶ）

UFSの構造

- **UNIXが採用したデータ構造**

BB	SB I=10	MB	3	4
5	6	7	8	9
I	I	I	I	I
I	I	I	I	I
20	21	22	23	24
25	26	27	28	29

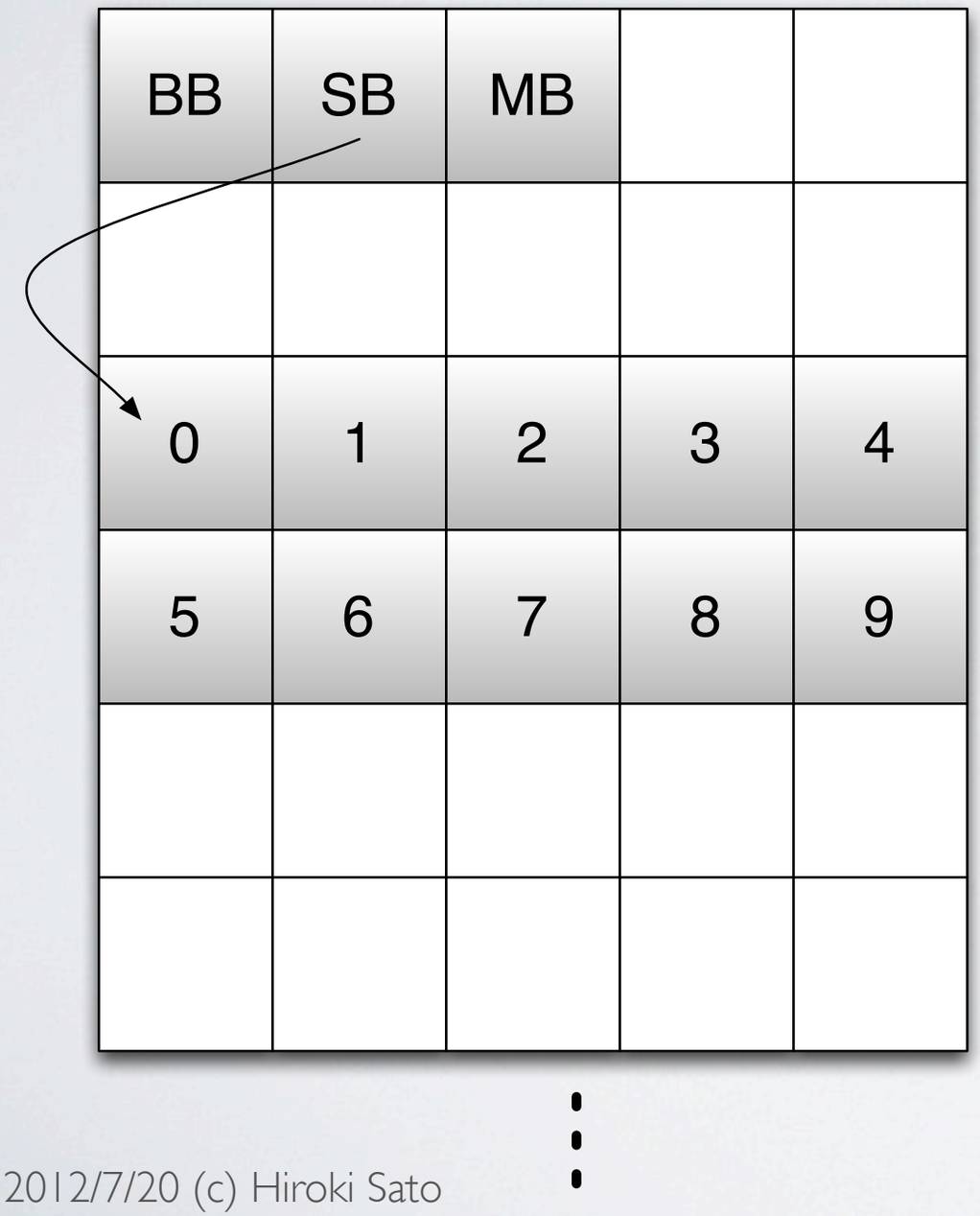
⋮

inode の場所は全体容量によって
変わってしまうので、
アドレスを記録する専用の場所も確保
（super block と呼ぶ）

BB = boot block
SB = super block
MB = metadata block

UFSの構造

- **UNIXが採用したデータ構造**

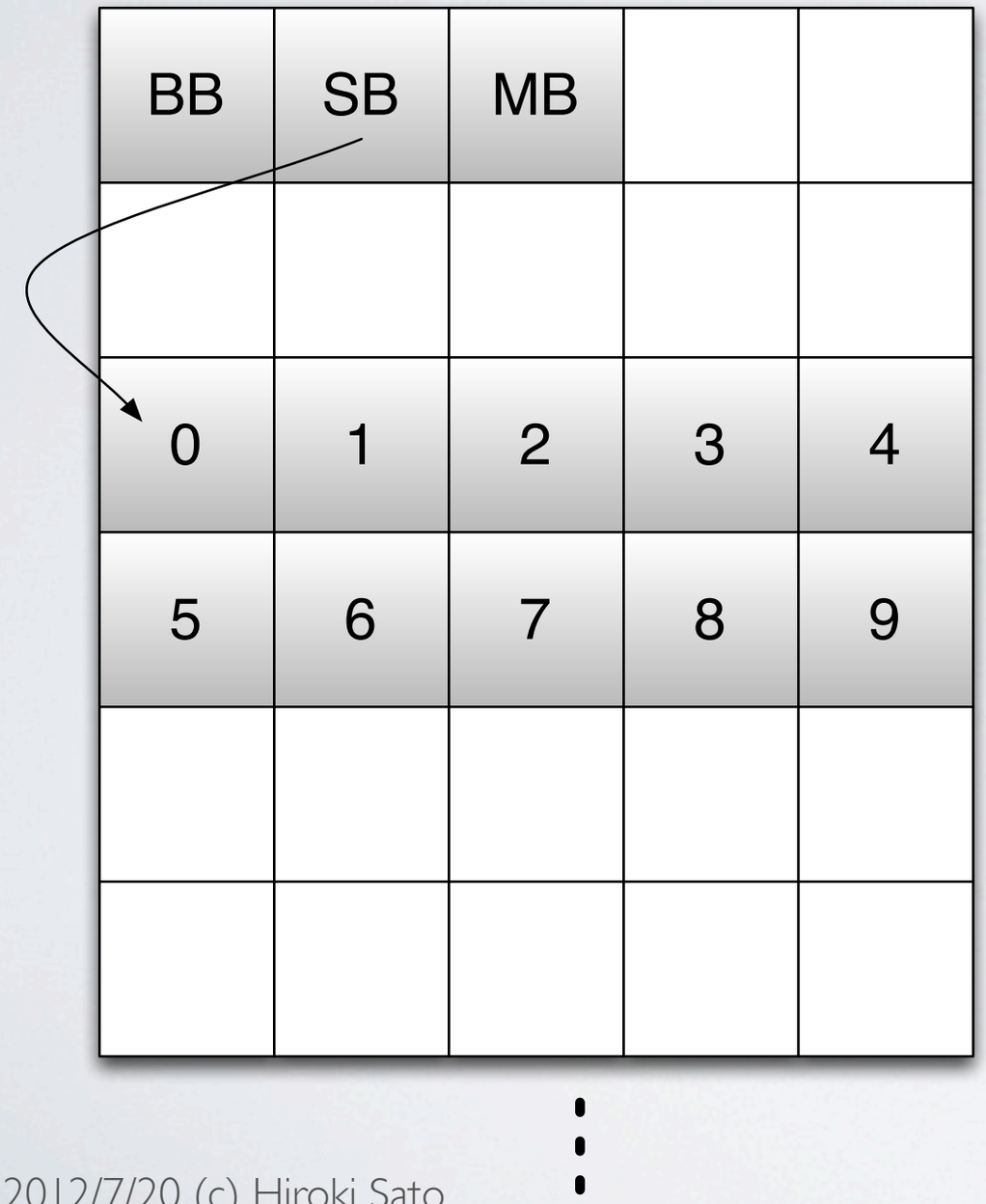


inode には番号をふっておく。

（まぎらわしいので他の番号は省略）

UFSの構造

- **UNIXが採用したデータ構造**

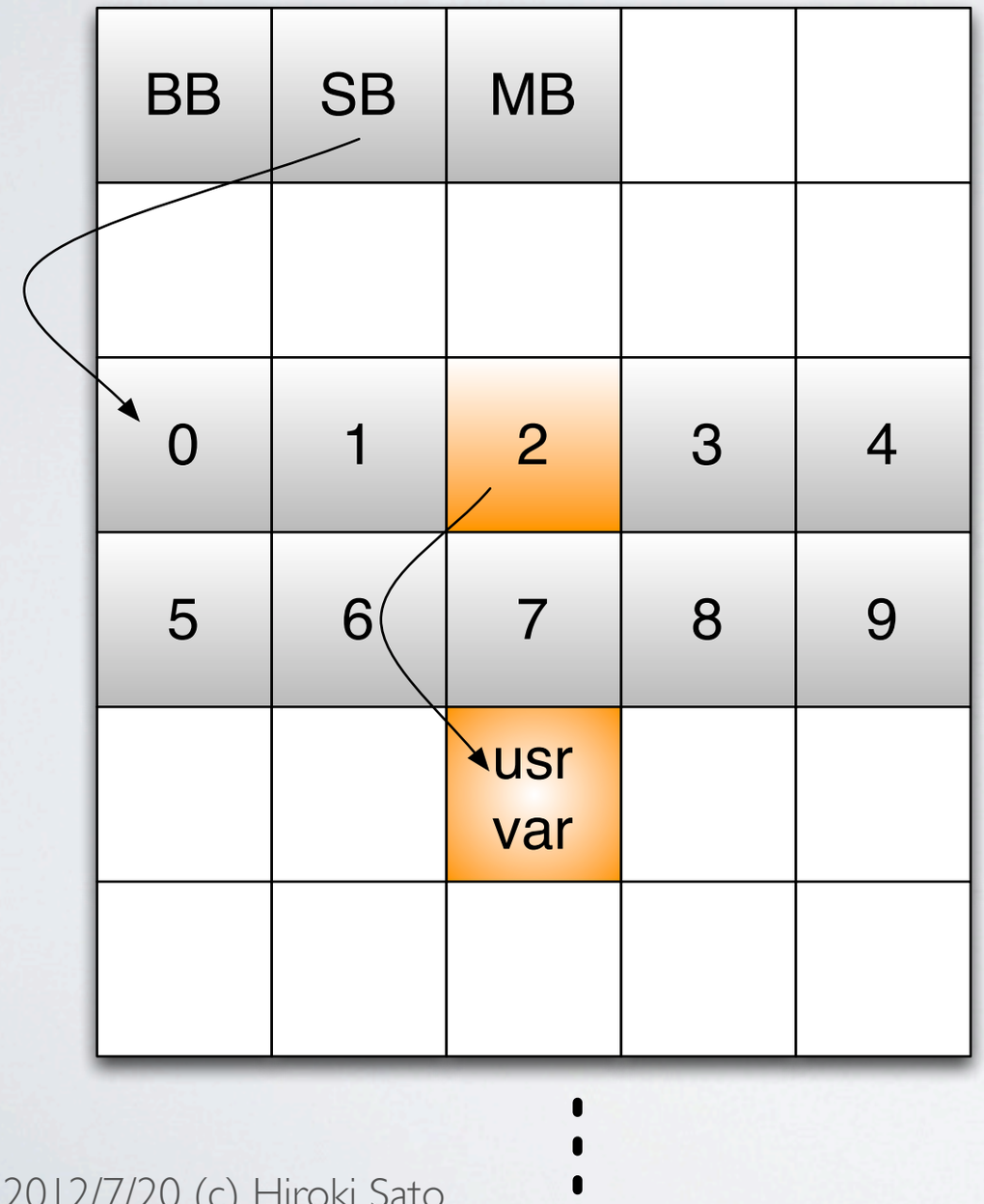


inode に記録されているもの

- ファイル名
- ファイル内容の形式
- 内容が記録されている場所
- 所有者などの補助情報

UFSの構造

- **UNIXが採用したデータ構造**



2 番目の inode は、常に
ルートディレクトリの
ファイル一覧が格納された
特殊ファイルを指している

UFSの構造

- **UNIXが採用したデータ構造**

特殊ファイルの中身

usr = inode 3 番

var = inode 4 番

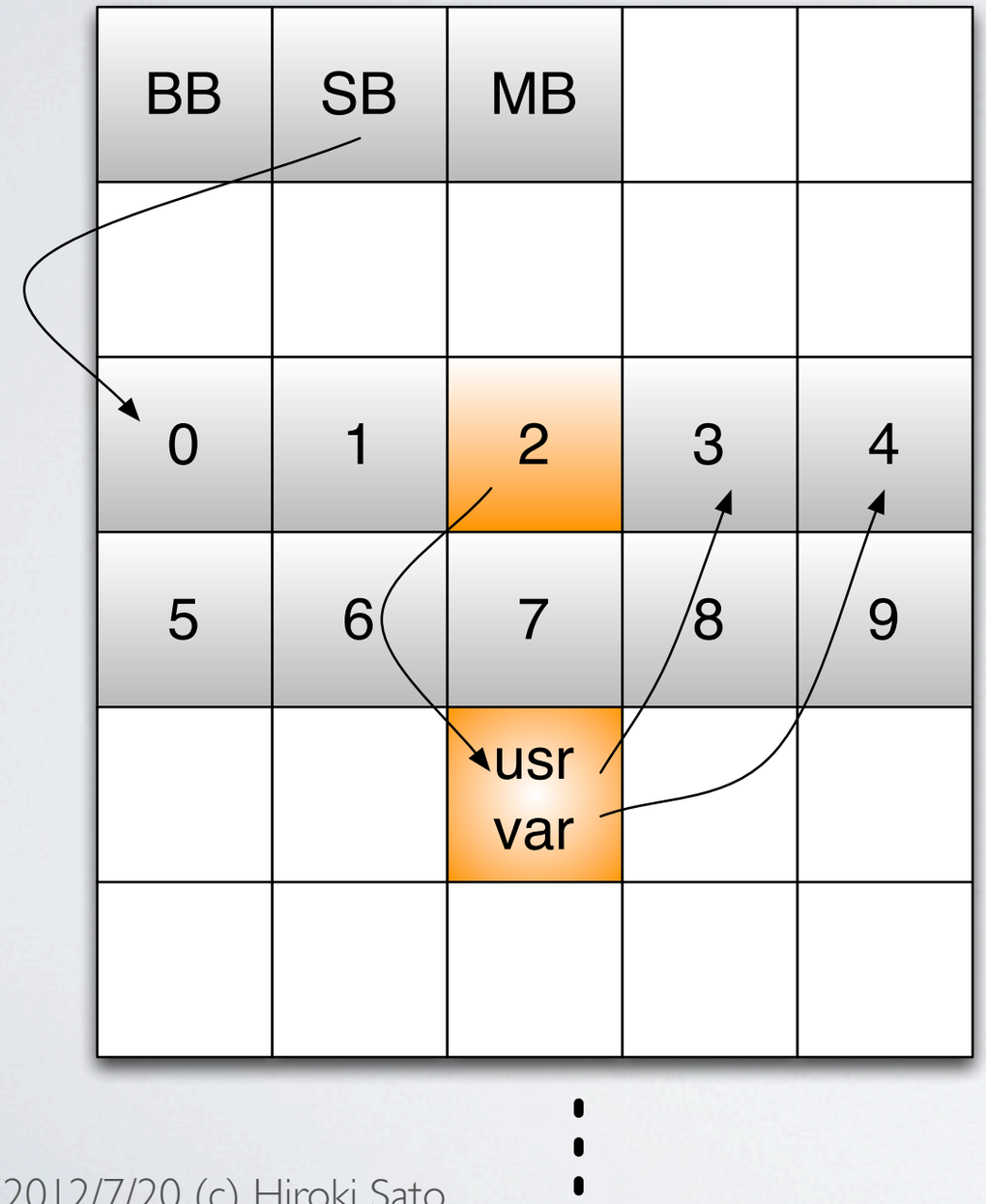
⋮

⋮

ディレクトリの正体はこれ。

UFSの構造

- **UNIXが採用したデータ構造**



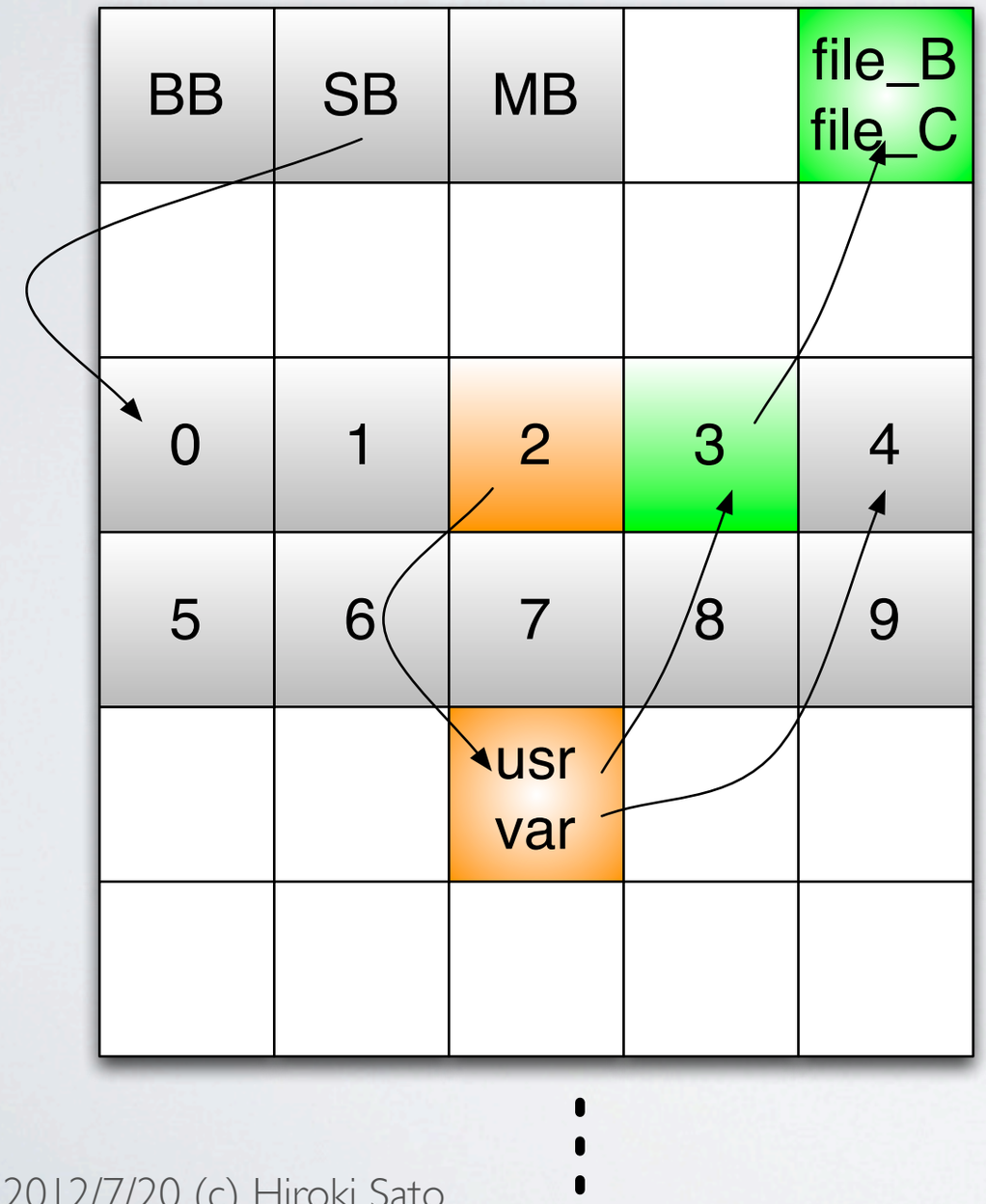
特殊ファイルの中身

usr = inode 3 番

var = inode 4 番

UFSの構造

- **UNIXが採用したデータ構造**



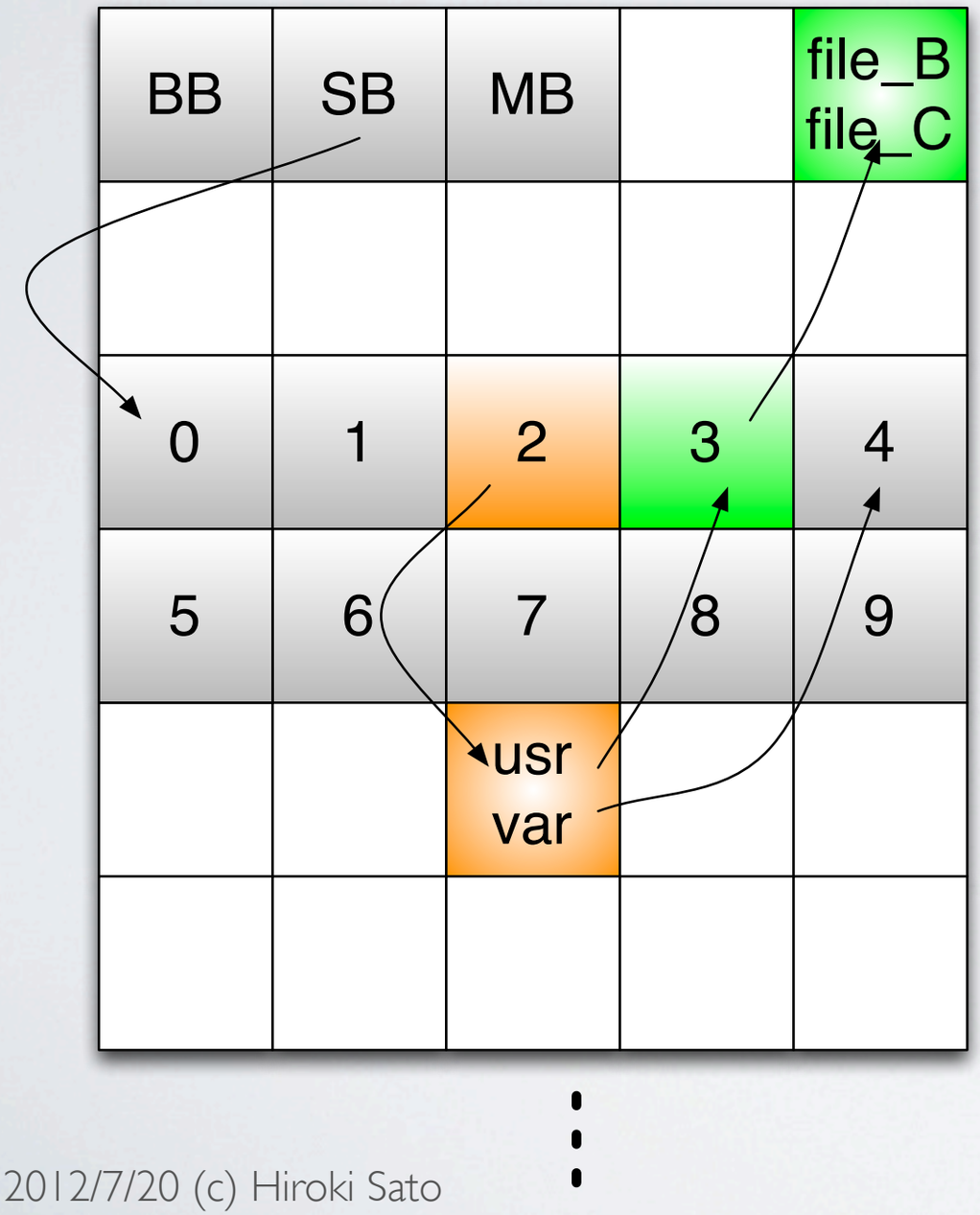
inode 3 番には、

「/usr のディレクトリ情報が

入っているアドレス」が書かれている。

UFSの構造

- **UNIXが採用したデータ構造**



ディレクトリ情報には、

file_B = inode 5 番

file_C = inode 6 番

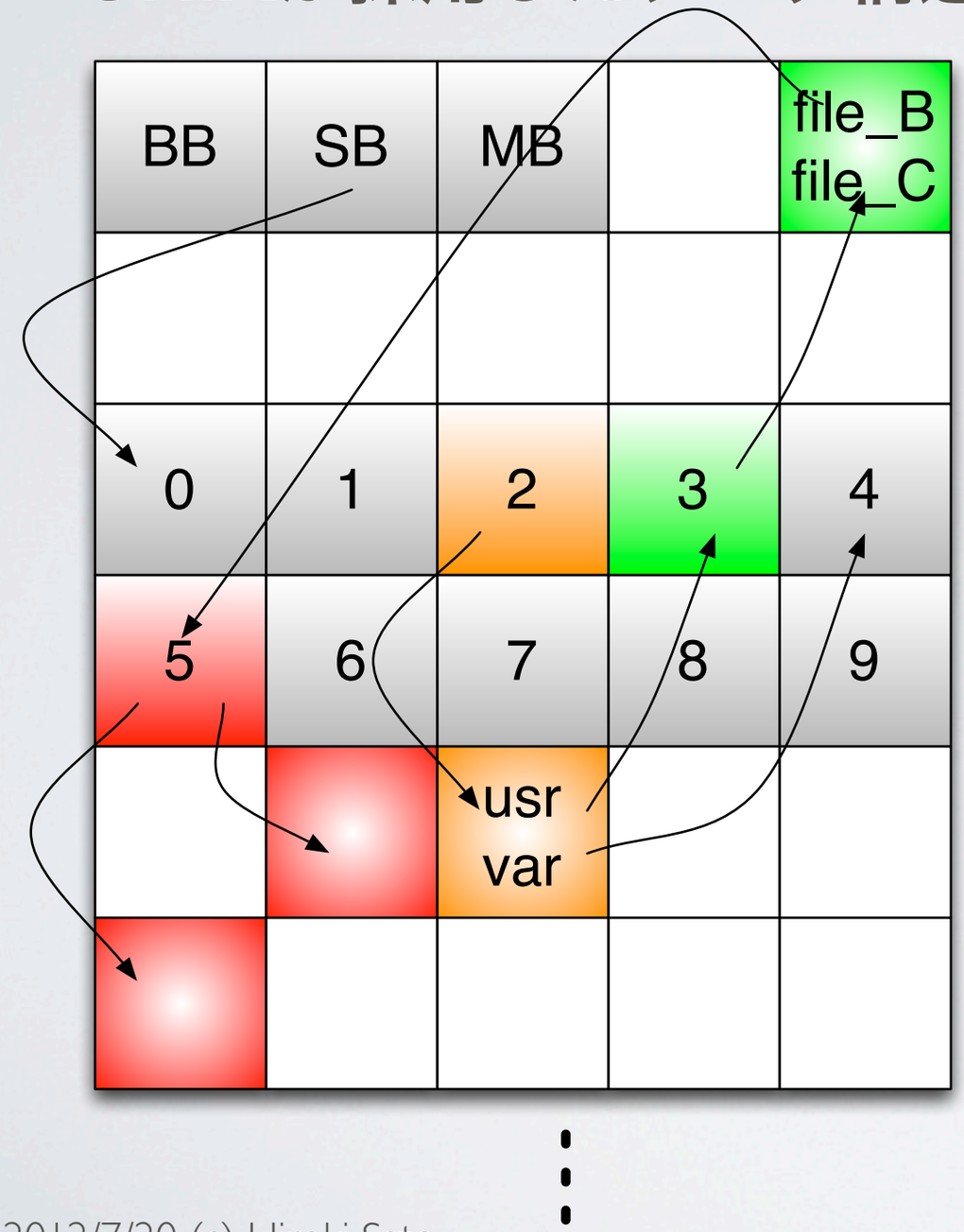
のように、

またファイル名と

アドレスのペアが並んでいる

UFSの構造

- **UNIXが採用したデータ構造**



ディレクトリ情報には、

file_B = inode 5 番

file_C = inode 6 番

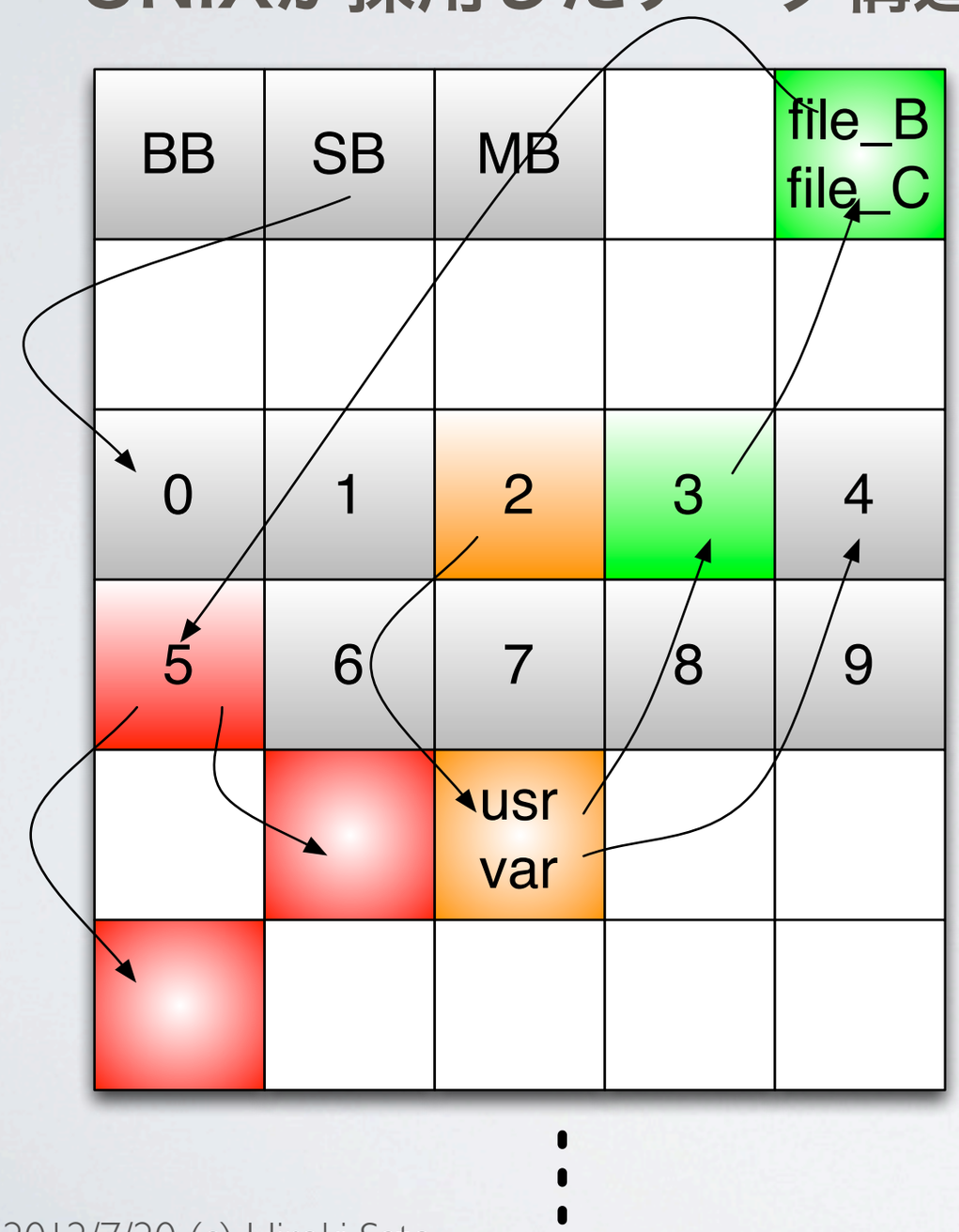
のように、

またファイル名と

アドレスのペアが並んでいる

UFSの構造

- **UNIXが採用したデータ構造**

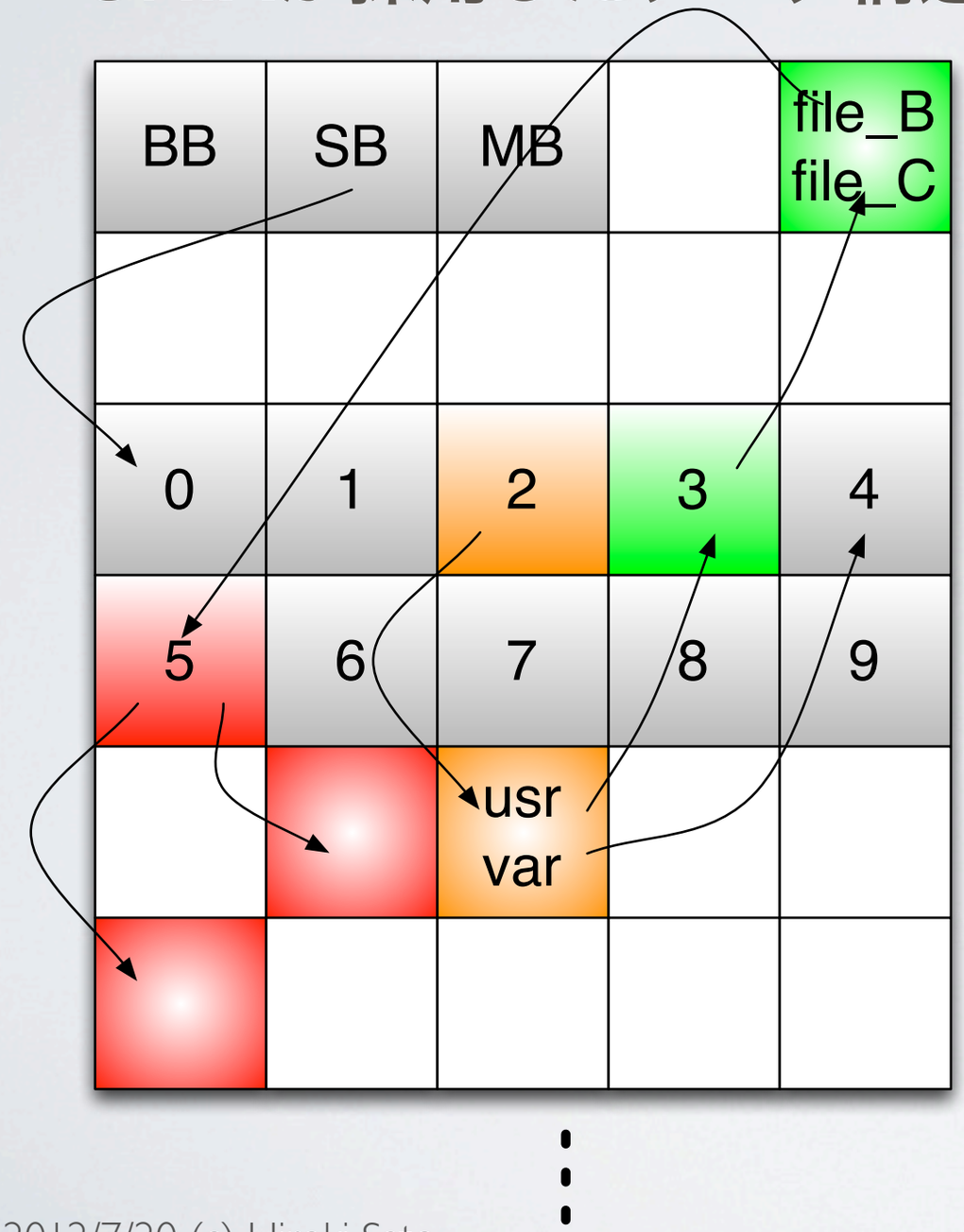


inode 5 番には、file_B の中身がどこにあるのかが書かれている。

ファイルが大きいと、複数のブロックのアドレスが書かれる。

UFSの構造

- **UNIXが採用したデータ構造**



ルートディレクトリから出発して、

/usr/file_B　にたどり着き、

その中身がどこに格納されているか、

まで分かった。

UFSの構造

- **本当のところは、もうちょっと複雑です。誤解なきよう。**
- 間接参照ブロックの構造や、ブロック内部のフラグメント、BSDが改良を加えたシリンダグループなど、説明に入っていないものはたくさんあります。

UFSの構造

- **つまり newfs コマンドのお仕事とは**
  - super block と inode を書き込む
  - inode 2 番にルートディレクトリをつくる
- 本当かどうか確かめてみましょう

UFSの構造

```
# mdconfig -a -t swap -u 0 -s 10m
# newfs /dev/md0
/dev/md0: 10.0MB (20480 sectors) block size 32768, fragment size 4096
    using 4 cylinder groups of 2.53MB, 81 blks, 256 inodes.
super-block backups (for fsck -b #) at:
 192, 5376, 10560, 15744
# mount /dev/md0 /mnt
# echo "hello" > /mnt/a
# umount /mnt
```

- /dev/md0 にファイルシステム構造をつくる
- /mnt にマウント
- /mnt/a というファイルをつくる（内容はhelloという文字列）

UFSの構造

```
# fsdb /dev/md0
Editing file system `/dev/md0'
Last Mounted on /mnt
current inode: directory
I=2 MODE=40755 SIZE=512
    BTIME=Jul 20 04:22:05 2012 [0 nsec]
    MTIME=Jul 20 05:31:44 2012 [0 nsec]
    CTIME=Jul 20 05:31:44 2012 [0 nsec]
    ATIME=Jul 20 04:22:05 2012 [0 nsec]
OWNER=root GRP=wheel LINKCNT=3 FLAGS=0 BLKCNT=8 GEN=79c8ce8f
fsdb (inum: 2)> ls
slot 0 off 0 ino 2 reclen 12: directory, `.'
slot 1 off 12 ino 2 reclen 12: directory, `..'
slot 2 off 24 ino 3 reclen 16: directory, `.snap'
slot 3 off 40 ino 4 reclen 472: regular, `a'
fsdb (inum: 2)> lookup a
component `a': current inode: regular file
I=4 MODE=100644 SIZE=6
    BTIME=Jul 20 05:31:44 2012 [0 nsec]
    MTIME=Jul 20 05:35:11 2012 [0 nsec]
    CTIME=Jul 20 05:35:11 2012 [0 nsec]
    ATIME=Jul 20 05:31:44 2012 [0 nsec]
OWNER=root GRP=wheel LINKCNT=1 FLAGS=0 BLKCNT=8 GEN=ffffffff379fdc1
fsdb (inum: 4)> blocks
Blocks for inode 4:
Direct blocks:
57 (1 frag)
```

UFSの構造

- inode 番号 2 にはルートディレクトリがあった。
- ルートディレクトリには、4個のファイル名があった。
- ファイル「a」は、inode番号 4 割り当てられていた。
- inode 番号 4 が指しているデータブロックは 57 番だった。
- /mnt/a は、57番目のブロックに”hello”が書かれているはず
  - 場合によってこの番号は変わります

壊れたらどうする？

- **「マウント」という操作をすれば、ファイルシステムの一部として記憶装置が使えることは分かった**
  - 名前空間の連結中は、いつアクセスされるか分からない
  - ファイルシステムの一部はメモリに格納されており、定期的に記憶装置に書き込まれる
  - メモリと記憶装置のデータが一致していない瞬間がたくさんある
- **いきなり電源が切れたりすると、すごく困る**

壊れたらどうする？

- **壊れ方のパターン**
  - スーパーブロックが壊れる＝ルートディレクトリが見えない
    ほぼすべてのデータへのアクセスができなくなる
    inode の場所を覚えていれば何とかなるが....
  - inode が壊れる＝その先の参照ができなくなる
    愉快な現象がたくさん
  - どれも書き込み途中の中途半端な状態が問題になる

壊れたらどうする？

- **fsck というツール**
  - UFS には、ファイルシステムをチェックするツールがある
  - fsck を実行すると、スーパーブロックから順番に
    inode を読み込み、すべての参照関係が矛盾なく
    存在するかどうかを調べてくれる。
  - 情報がおかしければ、可能な限り修復してくれる。
- ファイルシステムに異常があると、マウントができない

壊れたらどうする？

- **管理者が注意しなければならないこと**
  - fsck の行う操作のひとつに、「未参照ブロックの開放」がある。
  - inode が参照していないのに、使用中とマークされている
    ブロックを未使用に戻す操作
- これをやらないと、使ってないのに容量が減ってしまう

壊れたらどうする？

- **管理者が注意しなければならないこと**
  - fsck の行う操作のひとつに、「未参照ブロックの開放」がある。
  - inode が参照していないのに、使用中とマークされているブロックを未使用に戻す操作
- これをやらないと、使ってないのに容量が減ってしまう
- **この操作はすごくメモリを消費する**
  - 全部の参照関係をメモリに記憶し、それと比較しつつ進むため
  - 容量 1GB に対して 0.5MB のメモリが必要だと考えましょう
  - 1TB のファイルシステムがあるなら、500MB のメモリがないといざという時に悲しい結果になりかねません

壊れないようにする工夫

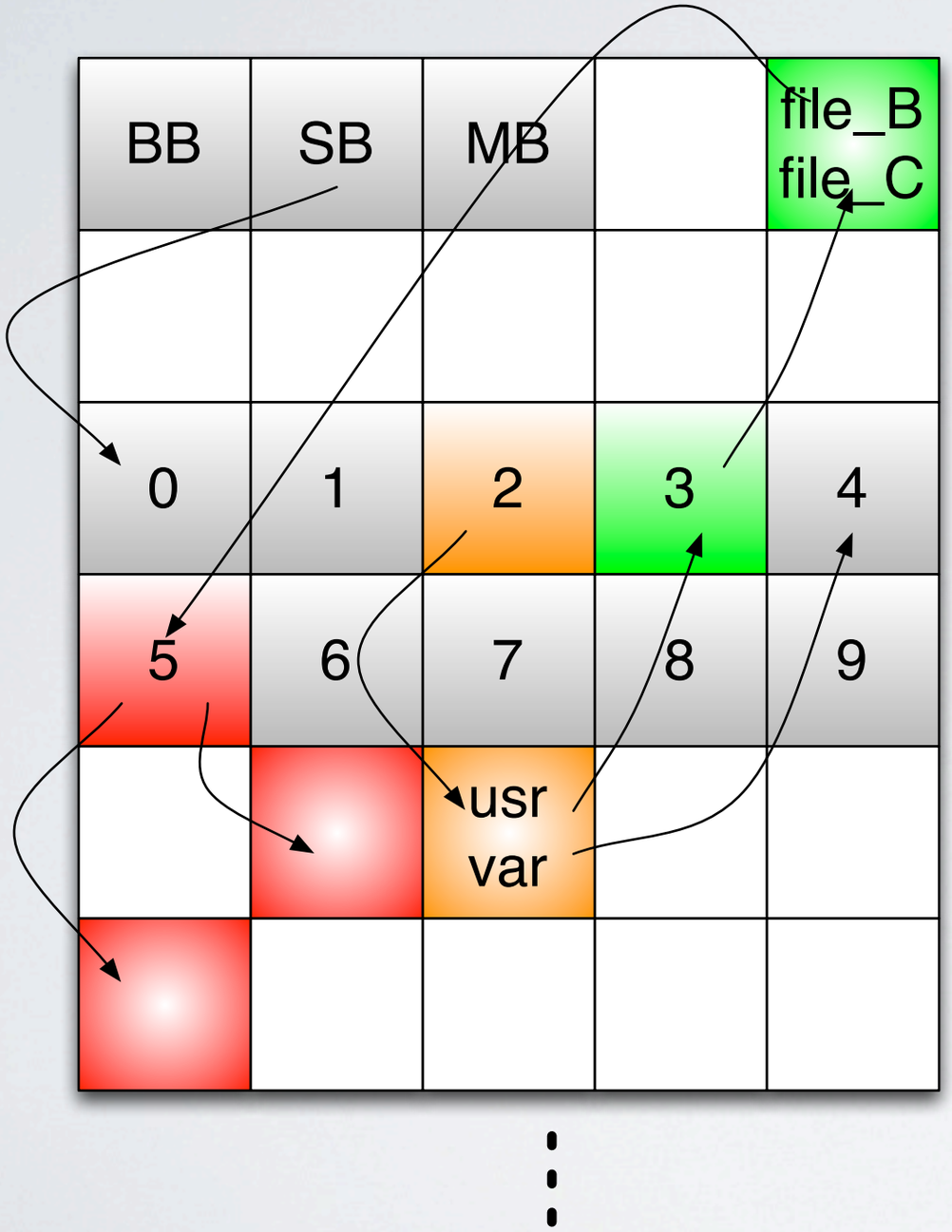
- **UFS の欠点を克服しつつ、壊れにくくする改良**
  - UFS は inode へのアクセス効率が性能に大きく効く
  - inode を非同期書き込みにすると速くなるが、信頼性が心配
  - 何とかならないか＝softupdates というテクニックを開発

壊れないようにする工夫

- **UFS の欠点を克服しつつ、壊れにくくする改良**
  - UFS は inode へのアクセス効率が性能に大きく効く
  - inode を非同期書き込みにすると速くなるが、信頼性が心配
  - 何とかならないか＝softupdates というテクニックを開発

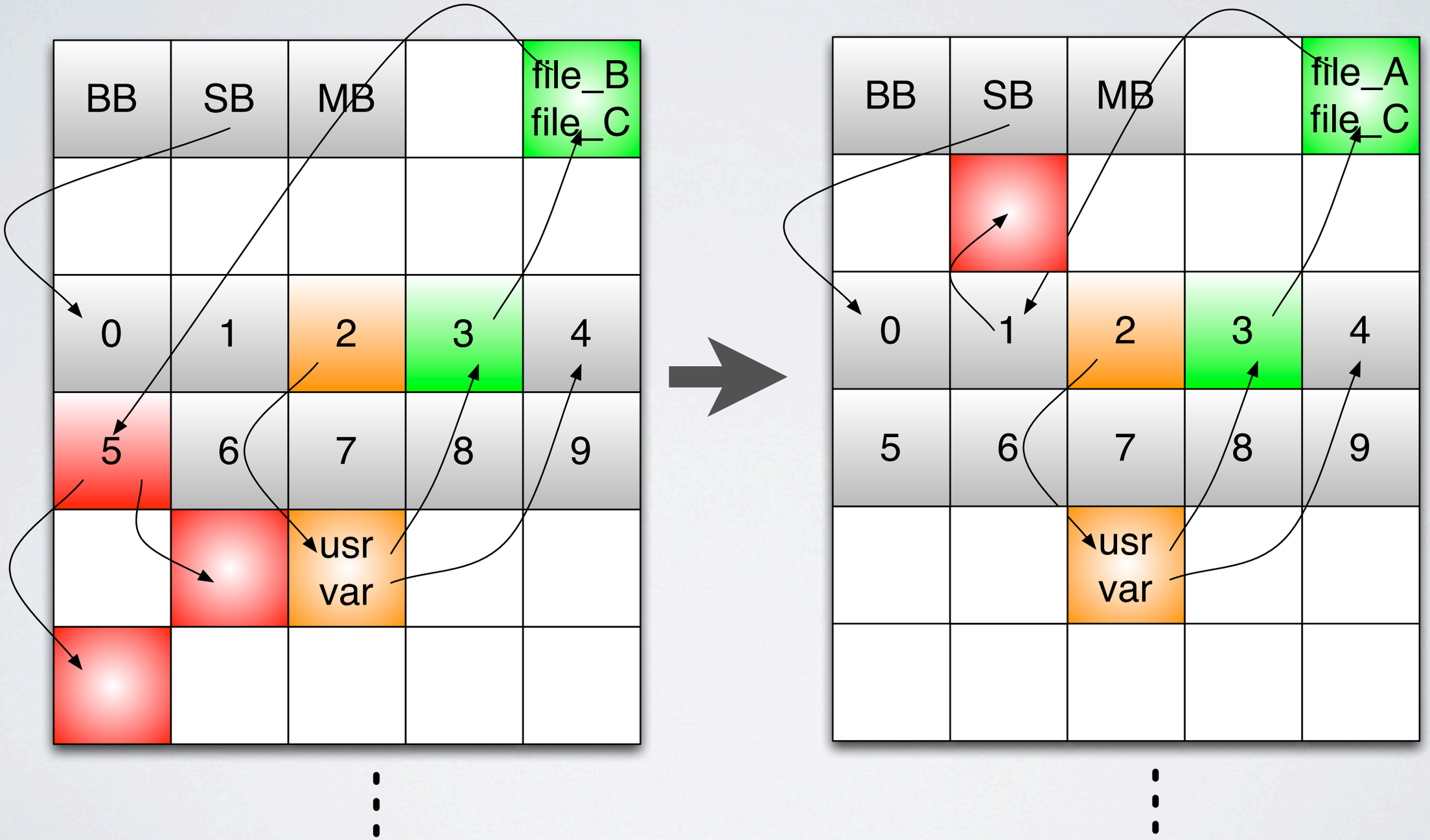
- **SoftUpdates**
  - 記憶装置に書き込む順番を厳しく規定することで、
    壊れた瞬間が存在しないように工夫したもの
  - メタデータの書き込みが非同期になる
    （ライトバックキャッシュがかかる）

SoftUpdates の原理

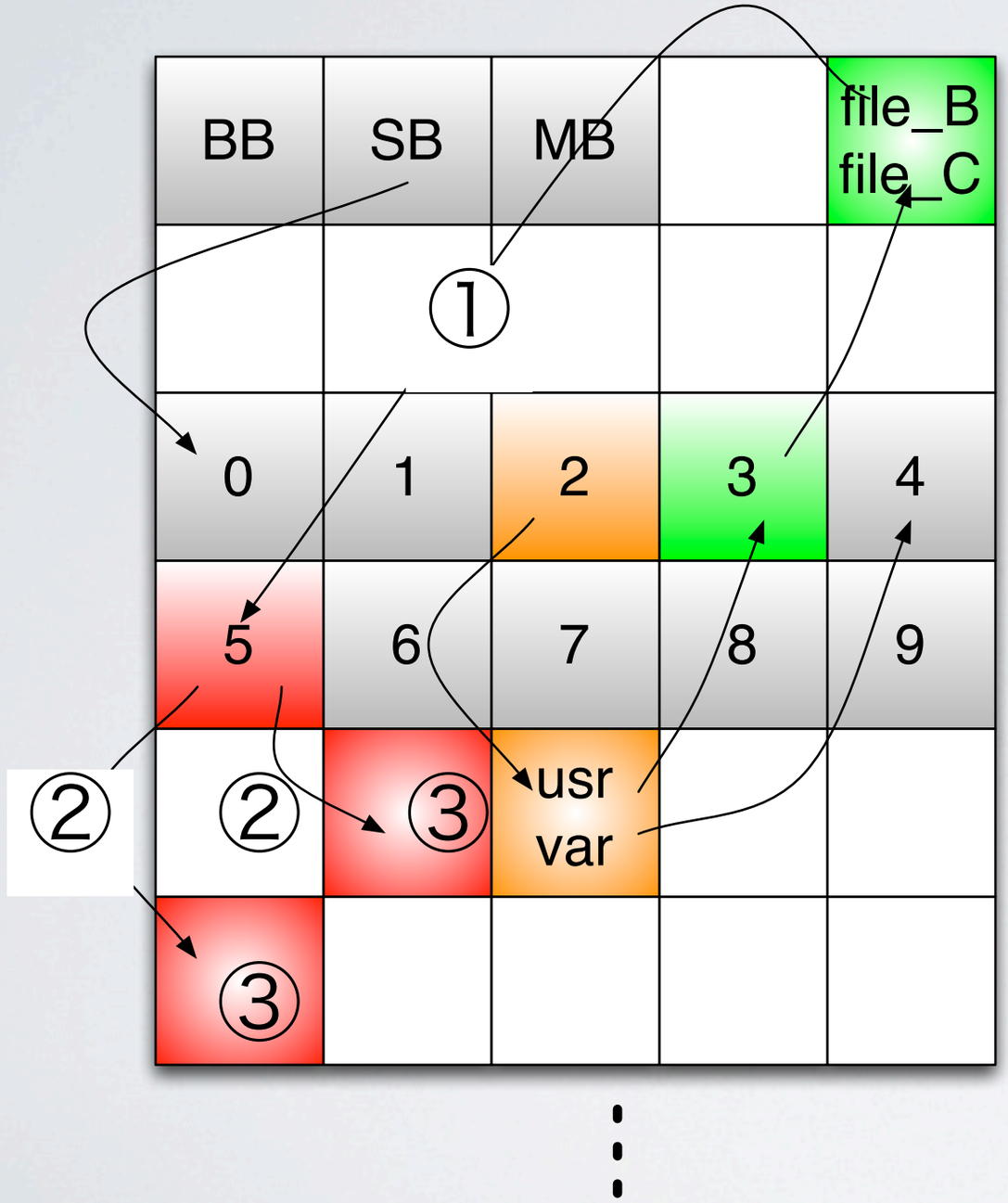


- **初期状態**
  - /usr/file_B
  - /usr/file_C
- ここで
  - /usr/file_A を作成
  - /usr/file_B を削除
- することを考える

SoftUpdates の原理



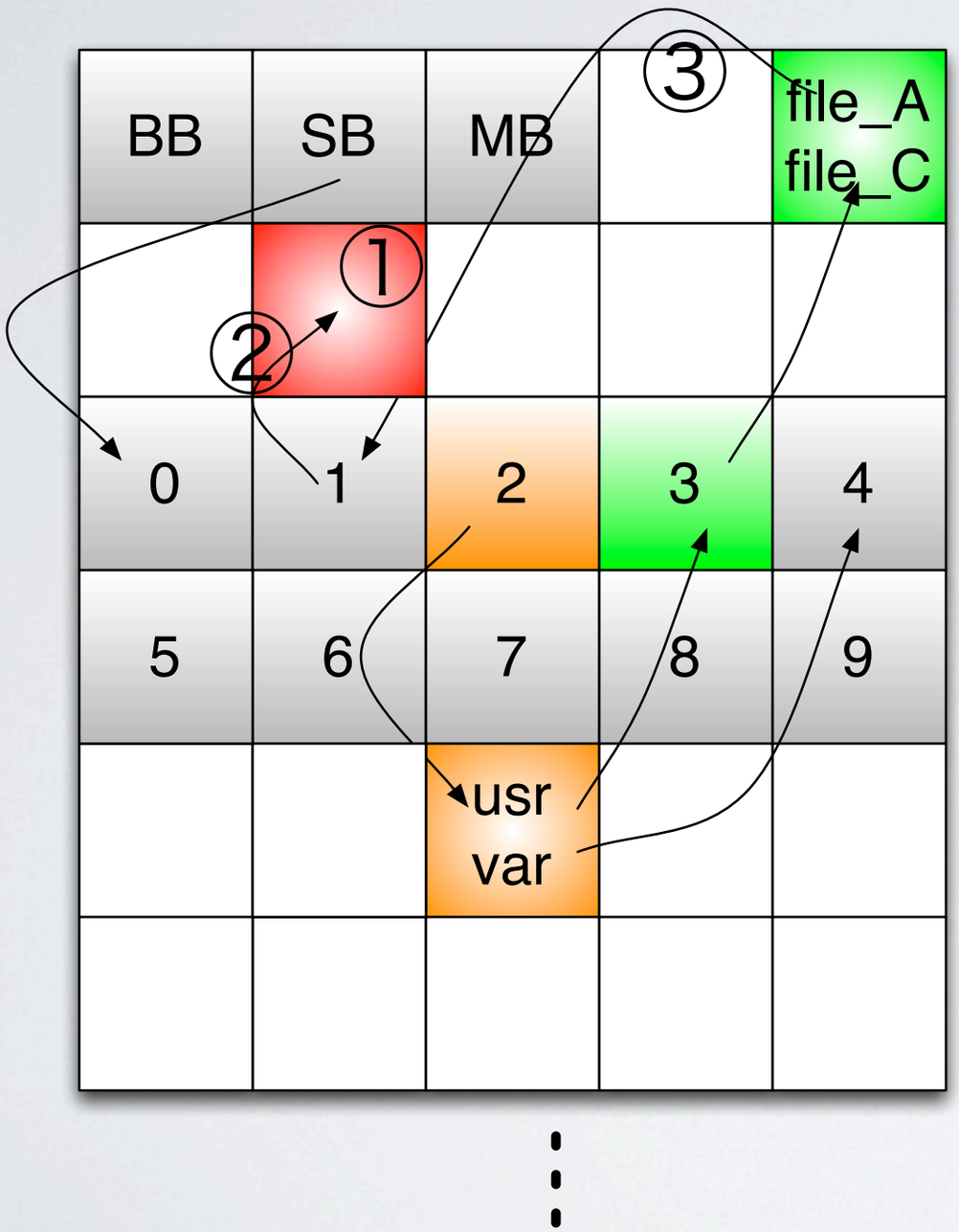
SoftUpdates の原理



- **file_B の削除手順**
  - ①エントリを消す
  - ② inode から消す
  - ③ データブロックを消す

- どの段階で電源が切れても、
  データに矛盾は発生しない

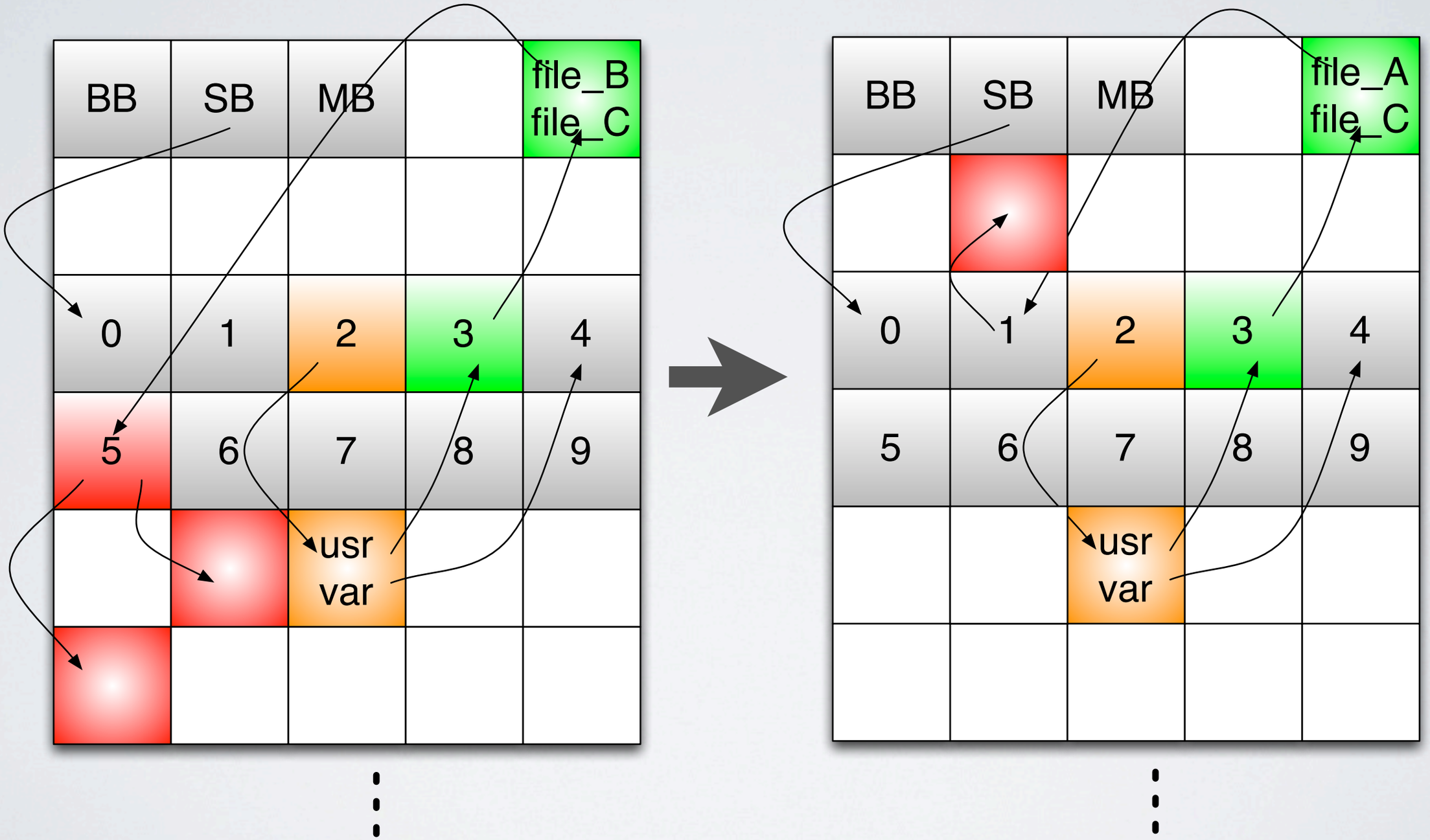
SoftUpdates の原理



- **file_A の追加手順**
  - ①データブロック準備
  - ② inode をつくる
  - ③ エントリをつくる

- どの段階で電源が切れても、
  データに矛盾は発生しない

SoftUpdates の原理



SoftUpdates の原理

- **追加と削除について、両方とも並行に処理できる**
  - 順番さえ守れば、どんなタイミングで処理しても良い
  - メモリ上で操作されたものを、まとめて記憶装置に書き戻す
- **SoftUpdates が有効な場合、ファイルシステムが損傷する可能性は低くなる。**
  - 電源が切れて fsck が走ったときは、未使用ブロックの回収のみが本当に必要な工程となる
  - 前述のように、メモリと時間をすごく消費する

SoftUpdates の原理

- **9.0 からは、SoftUpdates にジャーナリングを追加**
  - 先ほどの更新手順を「事前に記録してから」実行
  - 電源が切れたら、その記録を読んで再生する
  - fsck の時間がほぼなくなった
  - 安全にはなったが、電源断の時に戻る幅は大きくなっている

- ただし、UFS snapshot にバグが発見されており、
  dump -L などが使えない状態になっている。
  現在対応中。

SoftUpdates の原理

- 有効化と有効かどうかの確認

```
# tunefs -n enable /dev/md0
tunefs: soft updates set
# mount /dev/md0 /mnt
# mount
/dev/md0 on /mnt (ufs, local, soft-updates)
# umount /mnt
# tunefs -j enable /dev/md0
tunefs: Journal file fragmented.
tunefs: soft updates journaling set
# mount /dev/md0 /mnt
# mount
/dev/md5 on /mnt (ufs, local, journaled soft-updates)
```

まとめ

- **UNIX系OSのストレージとは**
  - ブロック構造を持った記憶領域の一次元配列
  - /dev にデバイスノードとして見える
- **GEOMとは**
  - FreeBSD が記憶装置を管理するために導入したフレームワーク
  - 記憶装置間、記憶装置とカーネル間で行われるデータ処理を部品化し、自由に組み合わせられるようにした
  - RAID や暗号化など、できることは多彩
- **ファイルシステム**
  - ユーザランドから見える記憶装置の姿のひとつ
  - 実体は記憶装置に置かれることが多い。
  - UFS の構造を簡単に知っておこう

まとめ

次回は、このあたりの知識を前提として

- ZFS の原理
- GEOM, UFS, ZFS の実際の操作や運用事例、性能分析、トラブル対応の方法

を扱う予定です。

おしまい

- 質問はありますか？