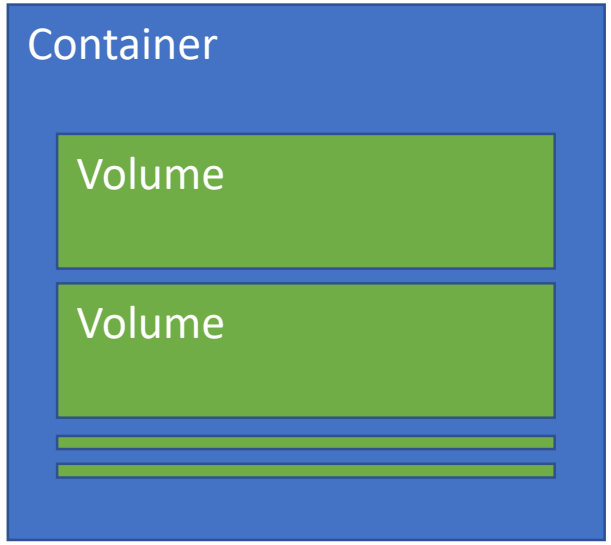
APFS 101

ContainerとVolume

- APFSのパーティションはContainerと呼ばれ、内部に複数のVolumeを定義できる。
- 各VolumeはContainer内の空き領域を共有する。



メタデータとobject

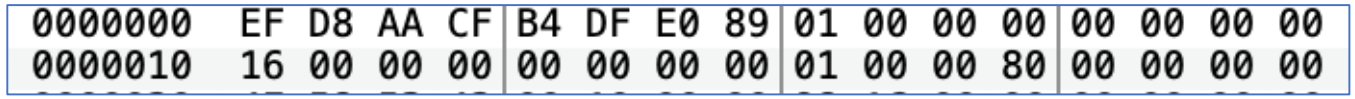
- APFSではメタデータ（file名、タイムスタンプ、パスやデータへの参照など）はobjectと呼ばれる。3種類のobjectが定義されており、共通ヘッダ（次頁参照）のo_typeフィールドのObject Type Flagsの値で識別することができる。
  - Physical object (OBJ_PHYSICAL)
    - ストレージ上に保持されるobject。更新時は新たな複製が作られ、その複製には異なるobject-id(o_oid) が付与される。o_oid＝物理アドレス（container superblockの先頭からの offset）となる。
  - Ephemeral object (OBJ_EPHEMERAL)
    - メモリに保持されるobject。同時にストレージ上にも存在し、更新されるたびに上書きされる。
  - Virtual object (OBJ_VIRTUAL)
    - ストレージ上に保持されるobject。更新時は新たな複製が作られ、同じo_oidが付与される。Virtual objectの物理アドレスはObject map（OBJECT_TYPE_OMAP）から参照する（後述）。

objectヘッダ(checksum, oid, xid)

- objectは32バイトの共通のヘッダ（obj_phys）を持っている。
- Transaction id(o_xid)を使ってobjectの世代管理を行う。同じo_oidを持つobjectが複数存在する場合、xidの値が最も大きいobjectが最新の状態を示す。
- 共通ヘッダ先頭にはobjectのFletcher-64 checksum（o_cksum）が保持されており、この検証に失敗した場合は同じo_oidで古いo_xidを持つobjectが利用される。
- Fletcher-64 checksumの計算については本資料P.29に記載している。
- type/sub_typeとobject type flagの定義については資料[1]のP.13およびP.18参照。

```
struct obj_phys {
        uint8_t         o_cksum[MAX_CKSUM_SIZE];
        oid_t           o_oid;
        xid_t           o_xid;
        uint32_t        o_type;
        uint32_t        o_subtype;
};
typedef struct obj_phys obj_phys_t;
```

- sample.rawの先頭にあるobjectのヘッダをバイナリエディタで開いた例



o_chksum:	\xEF\xD8\xAA\xCF\xB4\xDF\xE0\x89
o_oid:	0x0000000000000001
o_xid:	0x0000000000000016
o_type:	0x80000001
o_subtype:	0x00000000

メタデータの参照例

Container内のVolumeを参照する

Container内のVolumeを参照する (1)

- Volume Superblock（apfs_superblock_t）のmapを探す(1)
  - Container冒頭のContainer Superblock（nx_superblock_t）のnx_omap_oidフィールドから参照されている omap_phys_t（Containerのobject map）をたどる。Containerのobject mapのom_tree_oidフィールドの参照先に、OMAPのBTREE（btree_node_phys_t ）が配置されている。
  - sample.rawではブロック0x17ECにomap_phys_tが、0x17EDにbtree_node_phys_t があることがわかる。
  - nx_superblock_tの定義は資料[1]のP.25、omap_phys_tの定義はP.41参照。

Object Type: NX_SUPERBLOCK

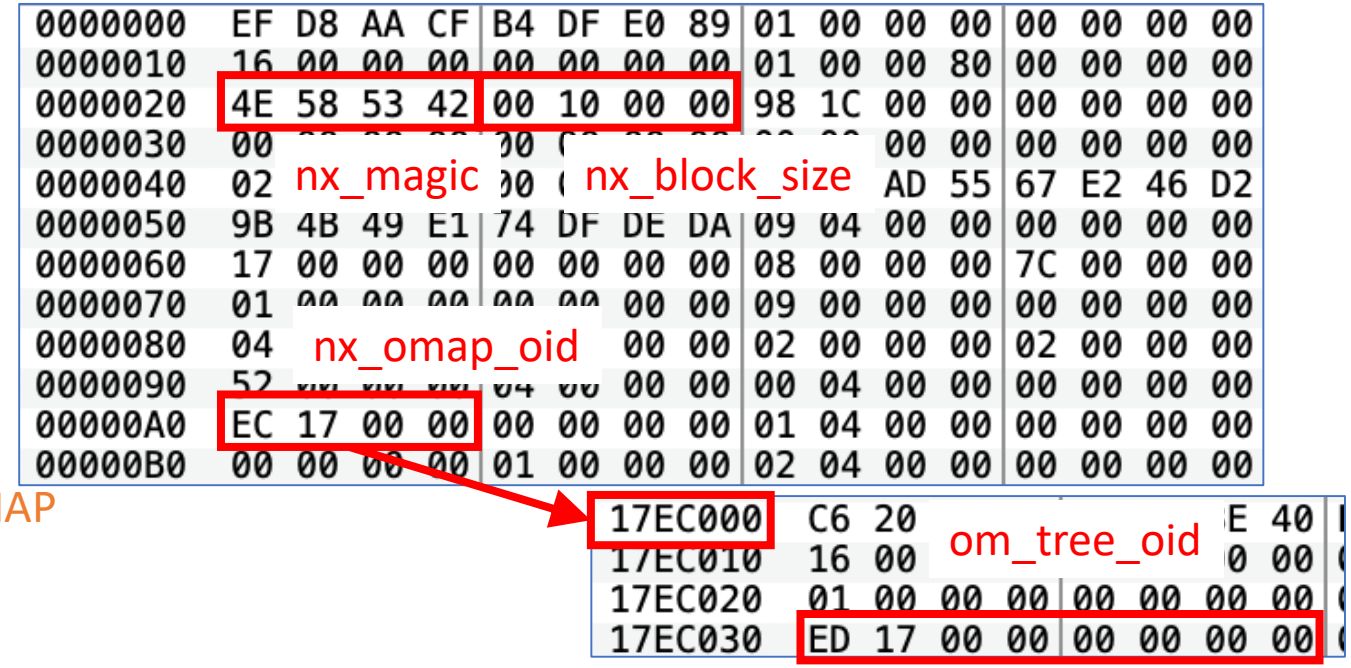
nx_superblock_t
...
0x20 u32 nx_magic
0x24 u32 nx_block_size
...
0xA0 u64 nx_omap_oid
...

Object Type: OMAP

omap_phys_t
...
0x30 u64 om_tree_oid
...

Object Type: BTREE
Object Sub Type: OMAP

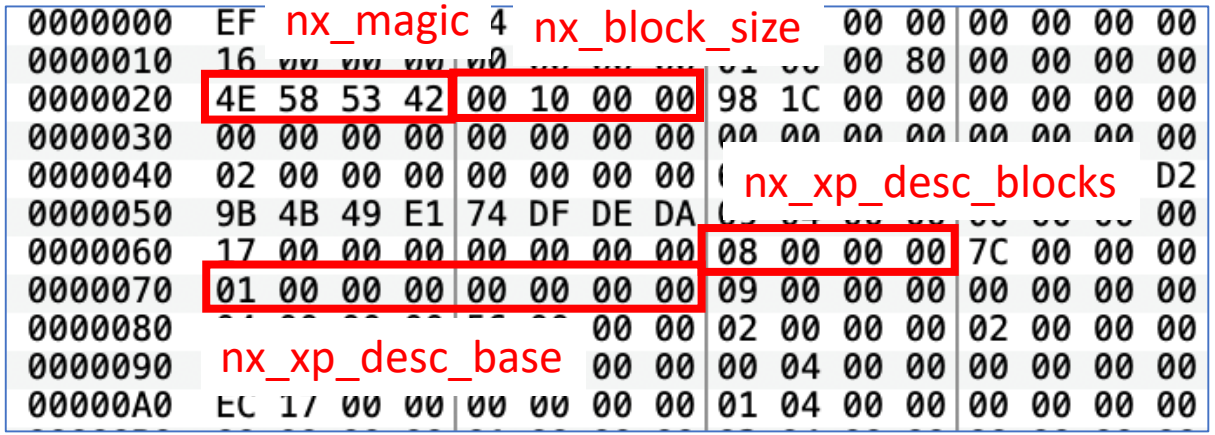
btree_node_phys_t



Container Superblockのバックアップを探す (1)

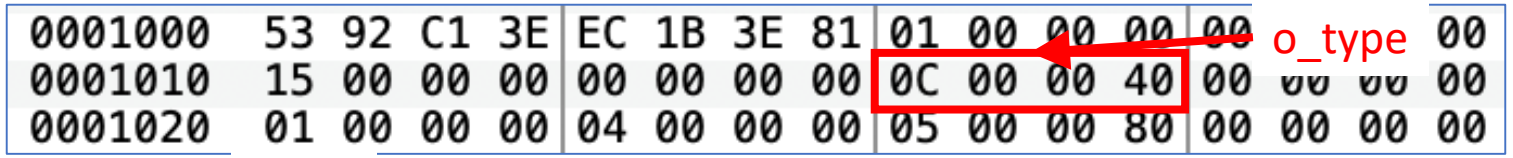
- 前頁ではContainer冒頭のContainer Superblockを参照したが、APFSでは最新を含む複数の世代のContainer Superblockの複製がcheckpoint descriptor areaに保存されているので、必要に応じてそれらを探し、利用することができる。
- checkpoint descriptor areaを探すためには、Container先頭のContainer Superblockのnx_xp_desc_baseフィールドと nx_xp_desc_blocks フィールドを参照する。
- sample.rawではcheckpoint descriptor areaがブロック1(nx_xp_desc_base)から始まり、全部で8(nx_xp_desc_blocks)ブロック存在することがわかる。このブロック1というのはブロックアドレスのことで、ここではblock sizeが0x1000なので、物理アドレス（Container先頭からのオフセット）0x1000を示す。

```
nx_superblock_t
...
0x20 u32 nx_magic
0x24 u32 nx_block_size
...
0x68 u32 nx_xp_desc_blocks
...
0x70 u64 nx_xp_desc_base
...
```

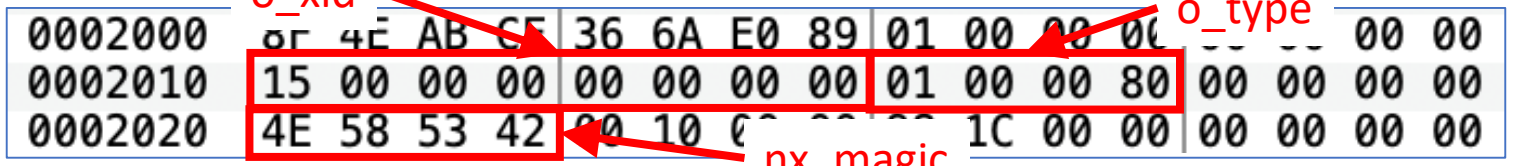


Container Superblockのバックアップを探す (2)

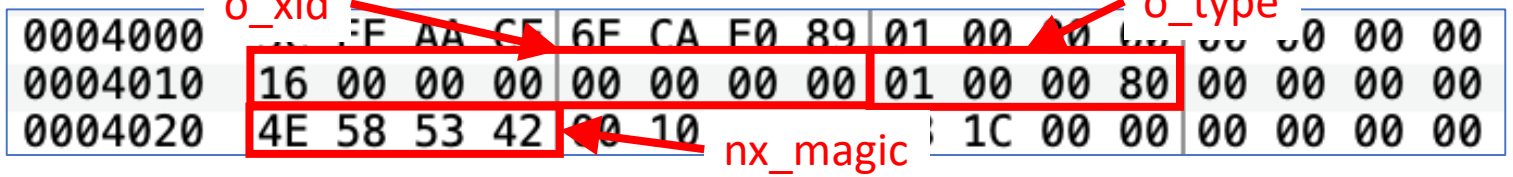
- checkpoint descriptor areaにはcheckpoint_map_phys_tまたはnx_superblock_tが保存される。ここにあるnx_superblock_tが最新および過去のContainer Superblockの複製。このnx_superblock_tのうち、最大のtransaction id(o_xid)を持つnx_superblock_tが最新のContainer Superblockの複製。
- sample.rawでは0x1000から8ブロックがcheckpoint descriptor areaなので、8ブロックそれぞれのヘッダを調べ、nx_superblock_tかどうか確認する（nx_magicまたはOBJECT_TYPEを確認）。
- sample.rawでは、4つのnx_superblock_tが見つかる（0x2000, 0x4000, 0x6000, 0x8000）。このうちo_xidの値が一番大きいのは0x4000のnx_superblock_tなので、これが最新のContainer Superblockだとわかる。これはContainer先頭のnx_superblock_tのコピーでもある。



(Type: OBJECT_TYPE_CHECKPOINT_MAP)



nx_superblock_t、xid: 0x15



nx_superblock_t、xid: 0x16

Container Superblockのバックアップを探す (3)

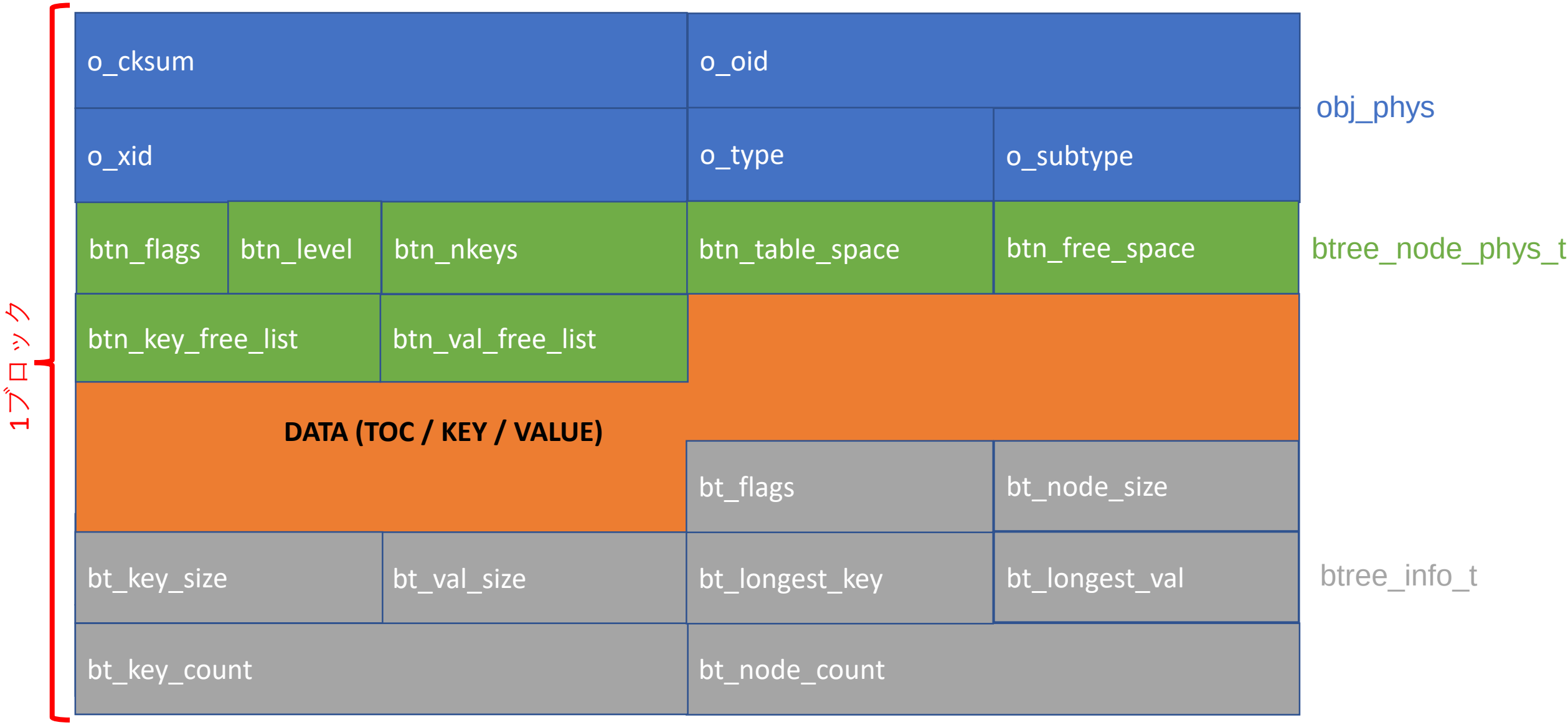
- 本資料では、Container Superblockの「バックアップを探す手段」としてcheckpoint descriptor areaの探索を紹介した。しかし、資料[1]には、Container先頭のContainer Superblockではなく、checkpoint descriptor areaにある最新世代のContainer Superblockを利用するよう記載されている（Container先頭のContainer Superblockは最新世代のものとは保障されていない）。
- ただし、Appleの実装（macOSのdisk utility）や、代表的なAPFSマウント用ツールであるapfs-fuseでは、Container先頭に配置されているContainer Superblockに矛盾などがない場合、そのContainer先頭のContainer Superblockが最新のContainer Superblockとして使用される。
  - 例えば、Container先頭Container Superblockの一部（nx_xp_desc_base、nx_xp_desc_blocks以外のフィールド）が破損している場合、 checkpoint descriptor areaにある最新のnx_superblock_tが正常であっても対象のイメージをマウントできなくなる。
- 現時点ではAPFSパーティションマウント時の動作を再現するにはContainer先頭のContainer Superblockを利用するのが適していると考えられる。

Container内のVolumeを参照する (2)

- Volume Superblock（apfs_superblock_t）のmapを探す(2)
  - BTREE（btree_node_phys_t）のB-Tree構造にはContainer内に存在するVolumeを示すVolume Superblock（apfs_superblock_t）が列挙されている。
  - まず、B-Treeを解析する必要がある。
  - B-Treeについては資料[1]のP.107、 btree_node_phys_tについてはP.108に記載されているが、次頁以降で簡単に解説する。

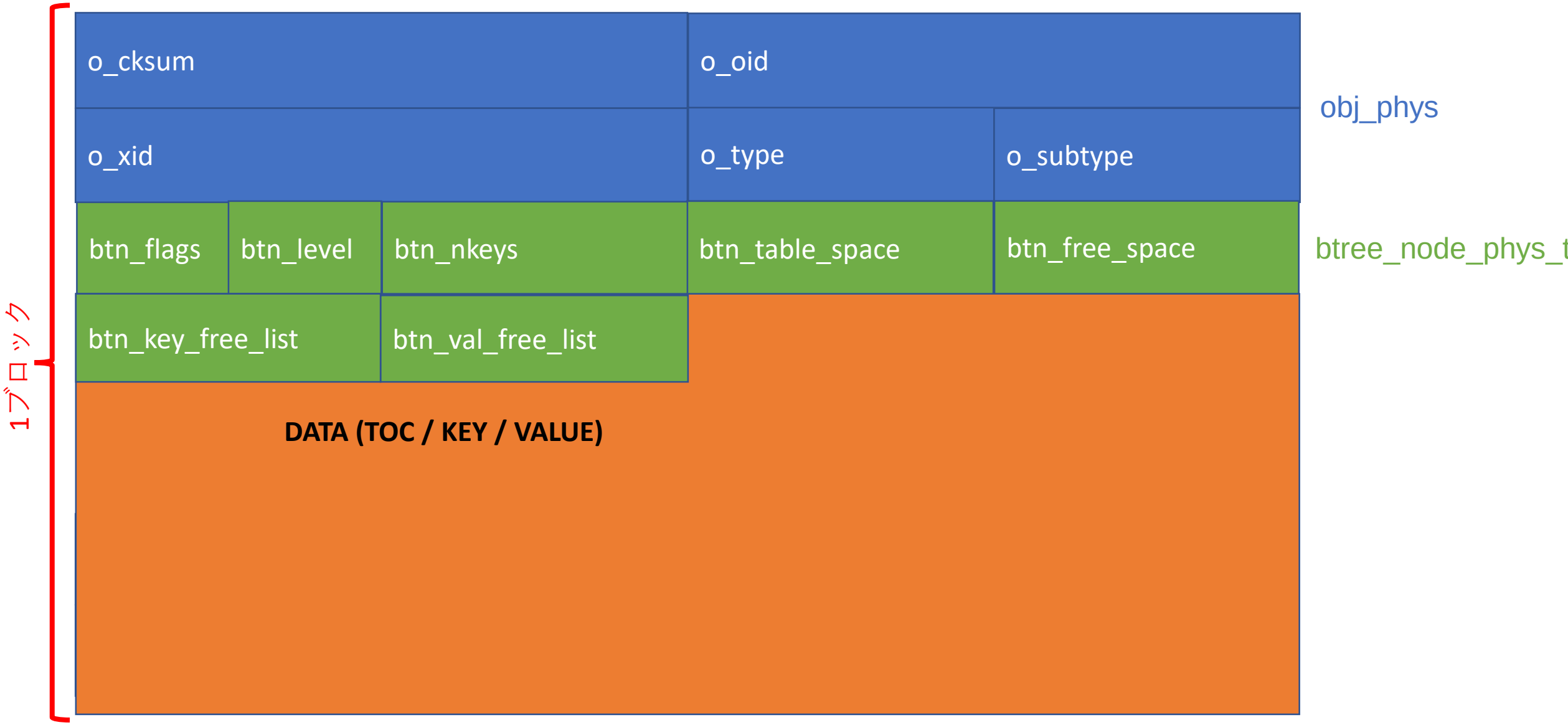
B-Treeの構造 (1) ROOT NODE

btn_flagsにBTNODE_ROOT(0x0001)
が設定されている



B-Treeの構造 (2) LEAF NODE

btn_flagsにBTNODE_ROOT(0x0001)
が設定されていない



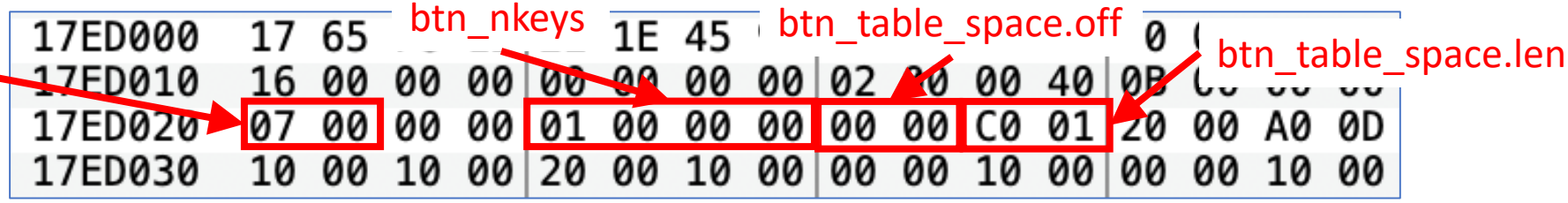
B-Treeの構造 (3) TOC/KEY/VALUE (1)

- btree_node_phys_tのbtn_data[]はTOC（Table of Contents）、KEY、VALUEという3種類のデータを含んでいる。
- TOCはbtn_data[]の先頭から始まる。KEYはTOCの末尾から始まる。
- 各KEY/VALUEの配置はTOCに保持されている（後述）。TOCでは各KEYはKEYフィールド先頭からのオフセット、各VALUEはVALUEフィールド末尾からオフセットで表現される。
- VALUEはbtn_data[]の末尾に配置される。 btree_node_phys_tのflagsにBTNODE_ROOTが設定されている場合は、btree_node_phys_tで始まるblockの末尾−0x28バイトがbtn_dataの末尾となる。BTNODE_ROOTが設定されていない場合は、 btree_node_phys_tで始まるblockの末尾がbtn_dataの末尾となる。

B-Treeの構造 (4) TOC/KEY/VALUE (2)

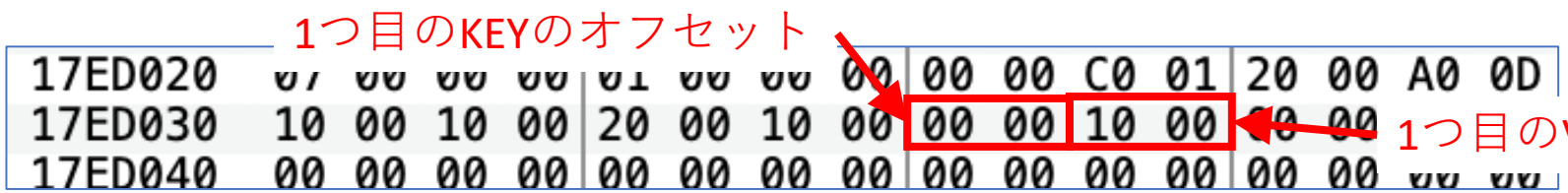
- TOC、KEY、VALUEフィールドの配置はbtn_table_spaceの値から計算する。
- 図のsample.rawの例では以下のようになる。
  - TOCのオフセット(btn_table_space.off)： 0x0、TOCの長さ(btn_table_space.len)：0x1C0
  - TOCはbtree_node_physの末尾+上記オフセットから始まるので、物理アドレスは以下のように計算できる。
    - ブロック先頭＋sizeof(btree_node_phys) + btn_table_space.off = 0x17ED000 + 0x38 + 0x0 = 0x17ED038
  - btn_nkyesが1なので、KEYの数は1つ（＝VALUEも1つ）。KEYフィールドはTOCの末尾から始まるので、物理アドレスは以下の様になる。
    - TOCの先頭＋TOCの長さ(btn_table_space.len) = 0x17E038 +0x1C0 = 0x17ED1F8
  - VALUEフィールドbtn_data[]の末尾から（先頭に向かって）始まるので、フィールド先頭は以下の様になる。
    - ブロック先頭 + ブロック長 – sizeof(btree_info_t) = 0x17ED000 + 0x1000 – 0x28 = 0x17EDFD8
  - なお、B-TreeがBTNODE_ROOTでない場合、VALUEフォールドはブロック末尾（ブロック先頭 + ブロック長）から始まる。

btn_flags:
- BTNODE_ROOT(1)
- BTNODE_LEAF(2)
- BTNODE_FIXED_KV_SIZE(4)



B-Treeの構造 (5) TOC/KEY/VALUE (3)

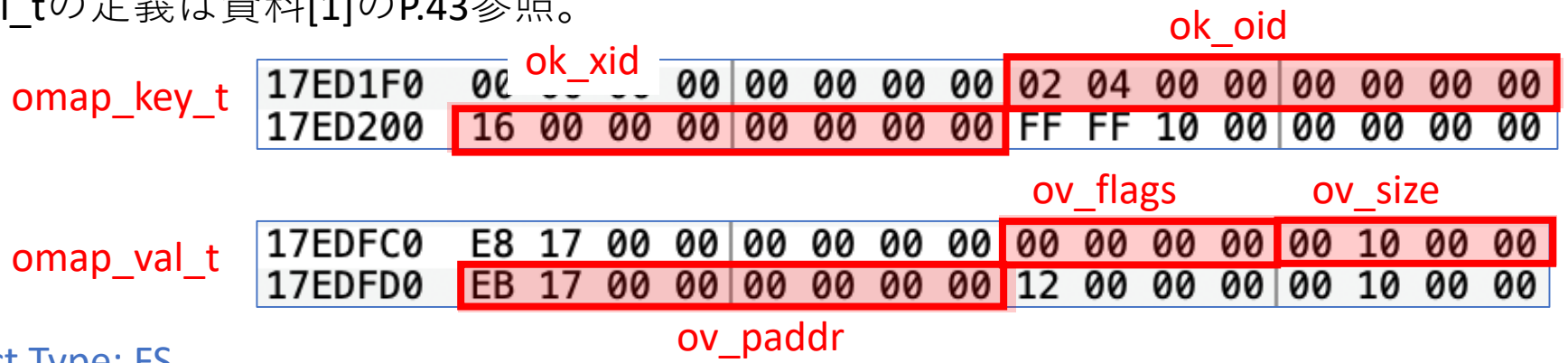
- btree_node_phys_tのflagsにBTNODE_FIXED_KV_SIZEが設定されている場合、TOCはKEY、VALUEのオフセットの組み合わせ（KEYのオフセット(u16)、VALUEのオフセット(u16)）を保持する。このとき、KEY、VALUEとも長さは0x10バイトに固定される。
- BTNODE_FIXED_KV_SIZEが設定されていない場合、TOCはKEY、VALUEのオフセットと長さを保持する（KEYのオフセット(u16)、KEYの長さ(u16)、 VALUEのオフセット(u16)、VALUEの長さ(u16)）。
- 図のsample.rawの0x17ED000のbtree_node_phys_tではBTNODE_FIXED_KV_SIZEが設定されているので以下のようになる。

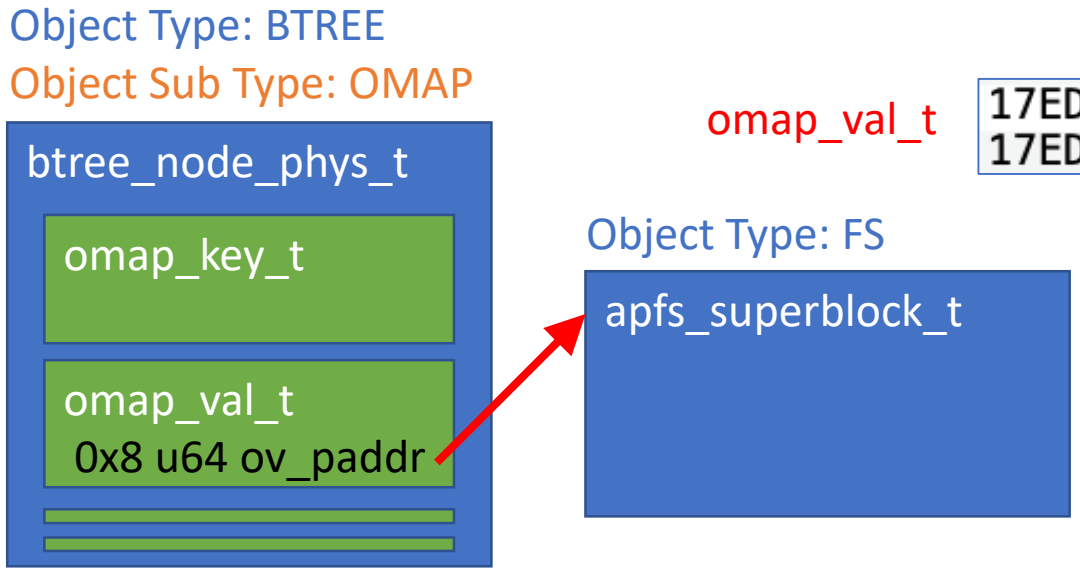


  - 1番目のKEYはKEYフィールドの先頭から始まる。長さは0x10。 0x17ED1F8 ～
  - 1番目のVALUEはVALUEの末尾−0x10バイトから始まる。長さは0x10。 0x17EDFC8 ～
  - KEYのオフセットはKEYフィールドの先頭からのオフセット、VALUEのオフセットはVALUEフィールドの末尾からのオフセット。複数KEYがある場合、例えば2番目のKEYの方が、1番目のKEYよりKEYフィールドの先頭側に配置されている場合などもある。

Container内のVolumeを参照する (3)

- Volume Superblock（apfs_superblock_t）のmapを探す(3)
  - sample.rawにはvolumeが1つしかないので、OMAPのBTREEにはomap_key_tとomap_val_tの組み合わせが1つ保持されている。
  - omap_key_tとomap_val_tの定義は資料[1]のP.43参照。



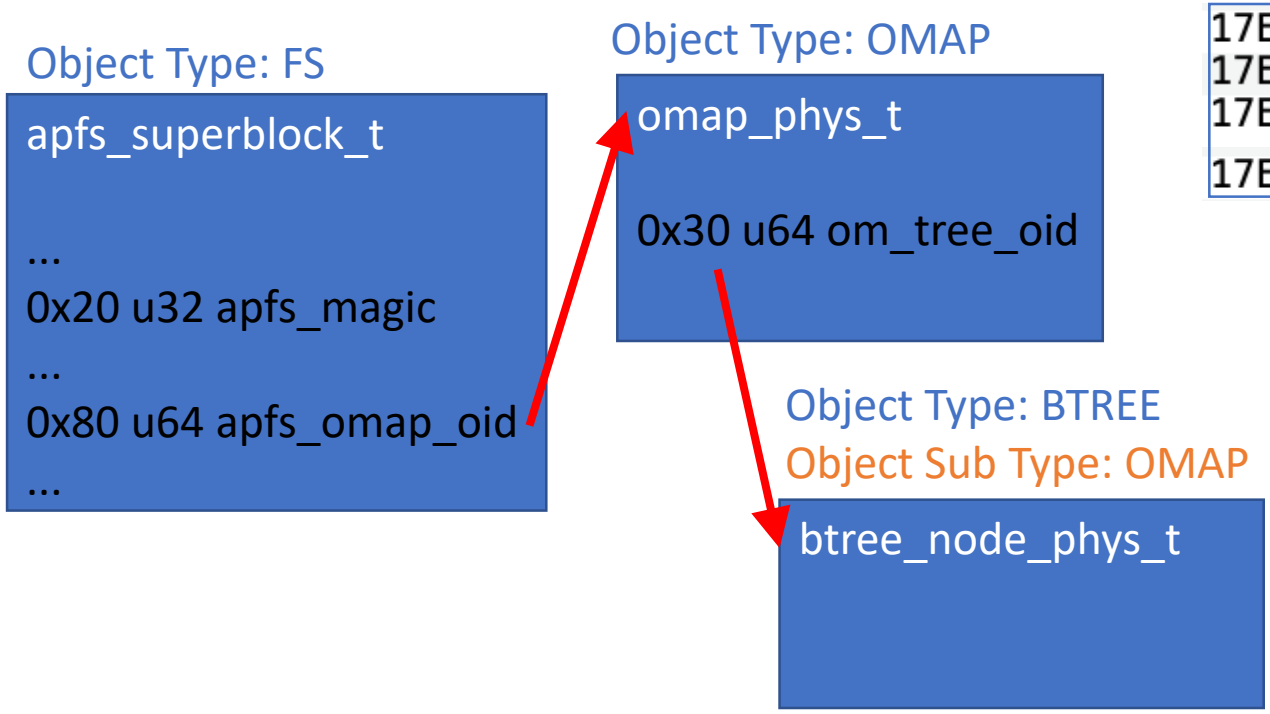


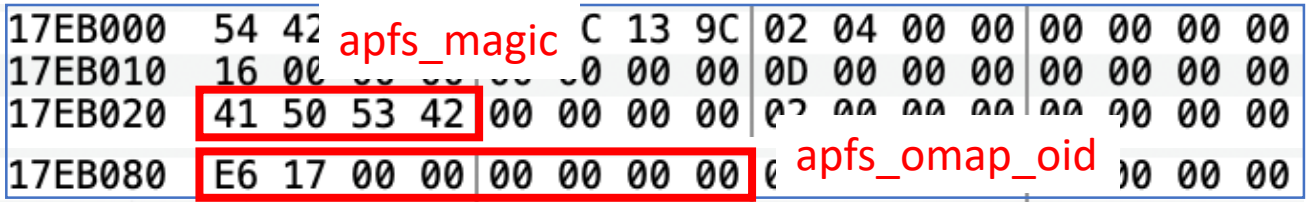
つまり、対象のVolume Superblockは
- object id: 0x402
- transaction id: 0x16
- size: 0x1000
- physical address block: 0x17EB -> 0x17EB000

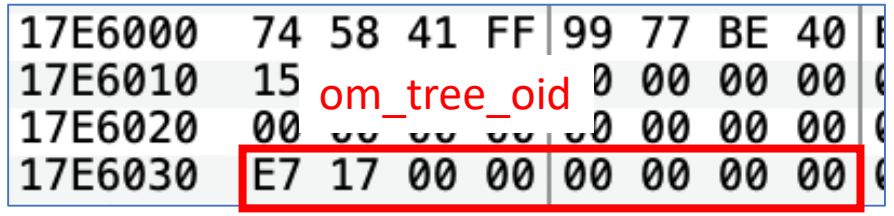
Volume内のfileを参照する

Volume内のfileを参照する (1)

- Container内のVolumeを探した時と同様に、Volume Superblock(apfs_superblock_t) のapfs_omap_oid フィールドからOMAP（omap_phys_t）をたどってB-Tree（subtype:OMAP）を探す。
- Volume Superblockの定義は資料[1]のP.48参照。



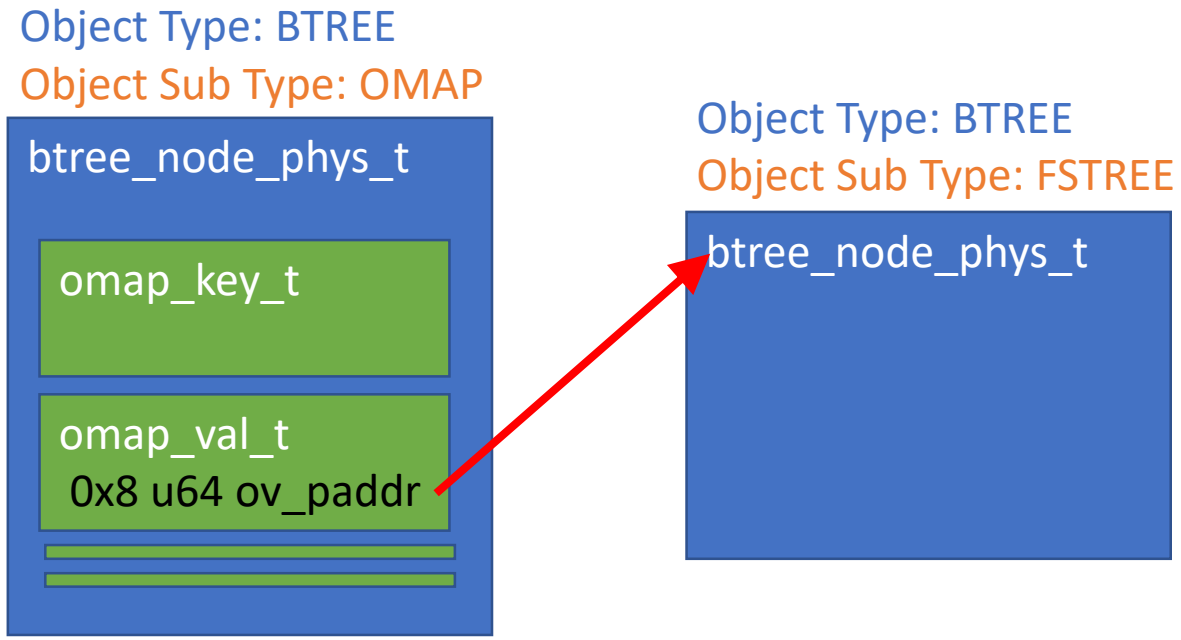




sample.rawの例では、ブロック0x17E7（＝0x17E7000バイト）にB-Tree（subtype:OMAP）が配置されていることがわかる。

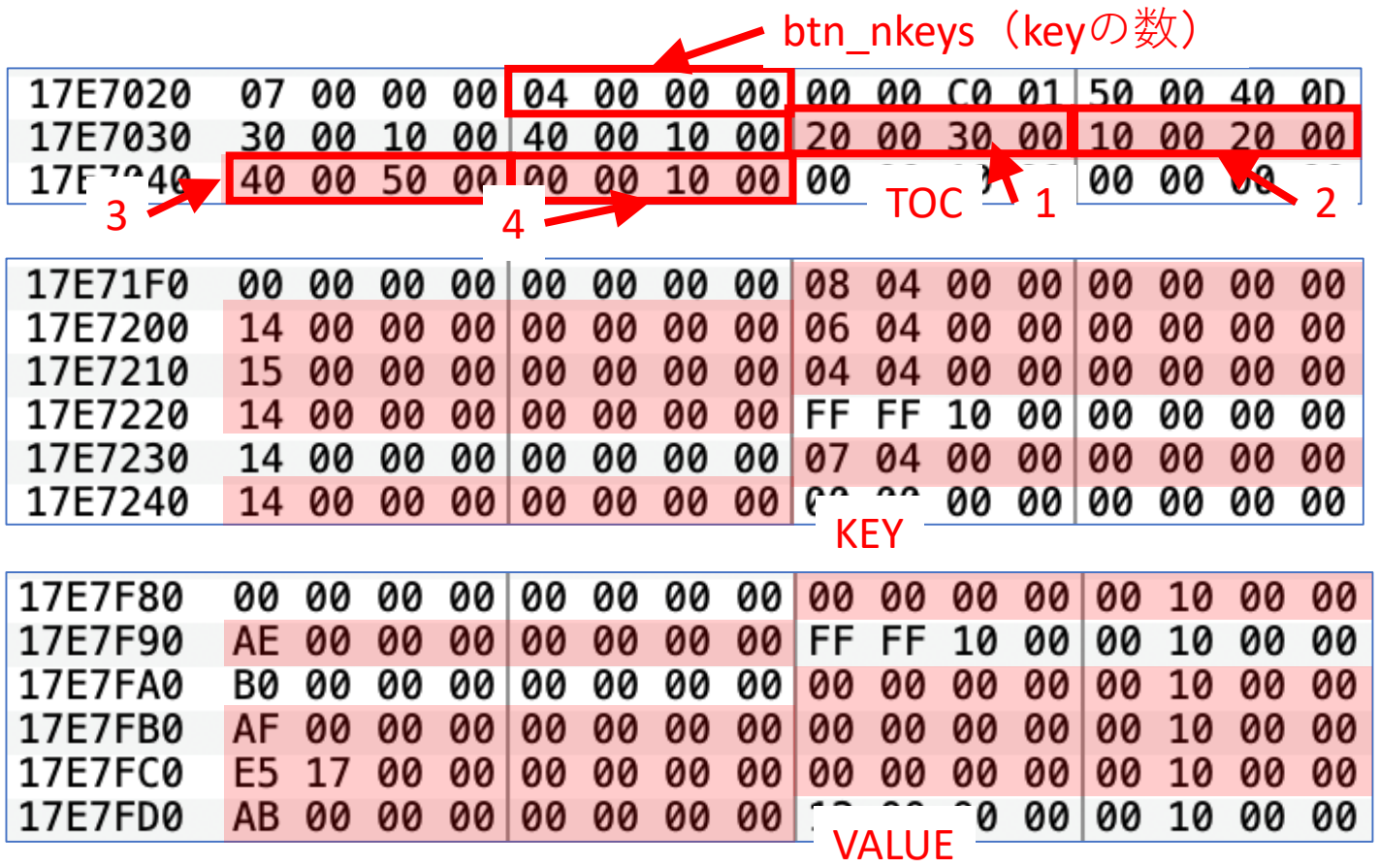
Volume内のfileを参照する (2)

- B-Tree（subtype:OMAP）にはfileやdirectory情報のB-Tree（subtype: FSTREE）への参照が保持されている。
- Volumeに複数のFSTREEが存在する場合は複数のomap_key_t/omap_key_val_tの組み合わせが保持されている。



Volume内のfileを参照する (3)

- sample.rawのケースでは、4つのFSTREEが見つかる。



1番目のKEY(omap_key_t):
- ok_oid: 0x404
- ok_xid: 0x14

1番目のVALUE(omap_val_t):
- ov_flags: 0x0
- ov_size: 0x1000
- ov_paddr: 0xAF

2番目のKEY(omap_key_t):
- ok_oid: 0x406
- ok_xid: 0x15

2番目のVALUE(omap_val_t):
- ov_flags: 0x0
- ov_size: 0x1000
- ov_paddr: 0x17E5

3番目のKEY(omap_key_t):
ok_oid: 0x407
ok_xid: 0x14

3番目のVALUE(omap_val_t):
ov_flags: 0x0
ov_size: 0x1000
ov_paddr: 0xAE

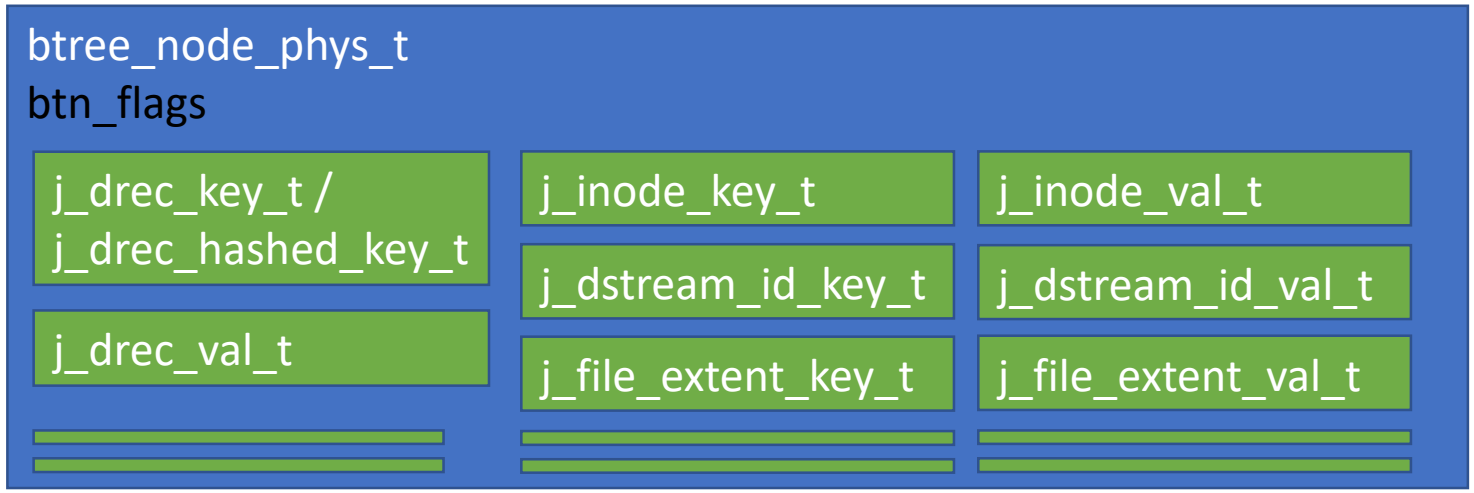
4番目のKEY(omap_key_t):
ok_oid: 0x408
ok_xid: 0x14

4番目のVALUE(omap_val_t):
ov_flags: 0x0
ov_size: 0x1000
ov_paddr: 0xAB

File System Object (1)

- FSTREE（btree_node_phys_t）には、fileやdirectoryに関する情報がkey/valueの形で保持されている。
- ボリューム内のfileやdirectoryの情報を得るためには、あらかじめすべてのFSTREEを解析してB-Treeに含まれる情報を保持しておく必要がある。

Object Type: BTREE
Object Sub Type: FSTREE **(root node / leaf node)**

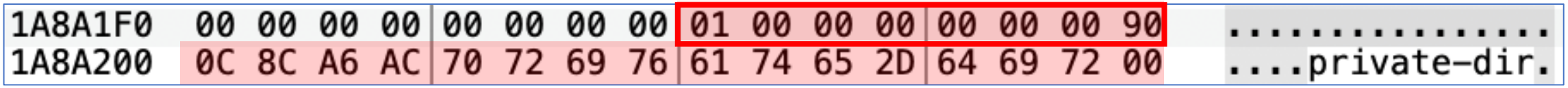


File System Object (2)

- KEYはj_key_tヘッダを持っており、そのなかのOBJ_TYPEの値により、key/valueの種類が識別できる。
- j_key_tの定義は資料[1]のP.64、OBJ_TYPE（j_obj_types）の定義はP.75参照。

File System Object (3)

- 例として、 sample.rawの3つ目のFSTREE（paddr: 0xAEのbtree_node_phys_tから始まるB-Tree）のKEYを取り上げる。まず、このFSTREEを解析してKEYのオフセットを特定する必要がある。本資料P.14から解説しているB-Tree解析と同じ手順で解析できる。ただし、このFSTREEはBTNODE_FIXED_KV_SIZEが定義されていないため可変長なので、TOCの読み方に注意する必要がある。また、BTNODE_ROOTが定義されていないためLEAF NODEなので、 btree_info_tがない（ブロック末尾がbtn_data[]の末尾となる）。
- sample.rawの3つ目のFSTREE（paddr: 0xAEのbtree_node_phys_tから始まるB-Tree）の1番目のKEYは、j_key_tに0x9000000000000001が格納されているので、TYPE->9 (APFS_TYPE_DIR_REC)、ID->1 となる。



- TYPEから、このKEYはj_drec_key_tまたはj_drec_hashed_key_tであることがわかる。なお、これはj_key_t末尾から4バイトのhashが格納されているのでj_drec_hashed_key_t。
- j_drec_hashed_key/j_drec_valueは対応するdirectory配下のfileの情報（IDやdirectoryへの追加日時など）を格納している。ここではid: 1のdirectory（最上位のdirectory。"root"というdirectoryとは異なるので注意）配下の"private-dir"というdirectoryに関する情報が含まれている。

File System Object (4)

- APFSは以下のようなdirectory構成をとる。

  /（最上位のdirectory）

  ├ private-dir

  ├ root

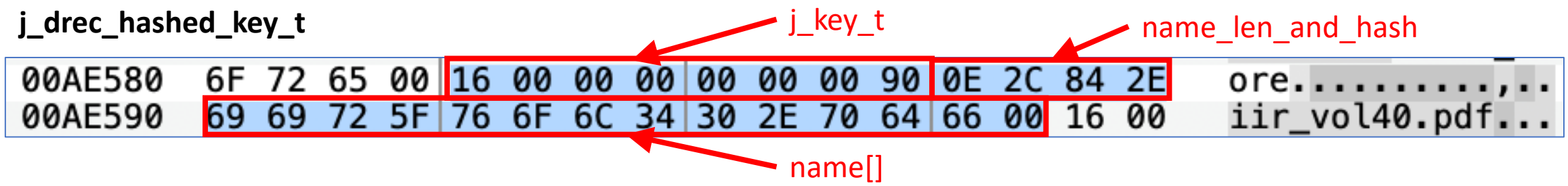
  　　├（ユーザが配置するdirectoryやfile）

- 資料[1]では上記の"root"がマウントポイントとされており、実際にmacOSのDiskUtilityでマウントすると、"root"がマウントポイントとなる。ただし、apfs-fuseでマウントした場合は、最上位のdirectory（上記の"/"）がマウントポイントとなるので注意が必要。

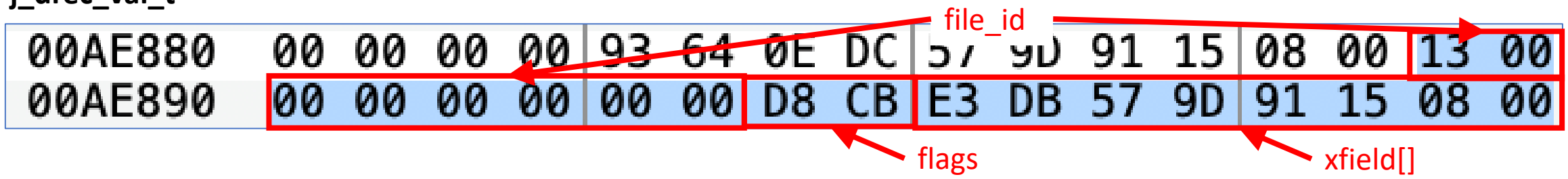
Volume内のfileを参照する (4)

- 例えばsample.raw内にある"iir_vol40.pdf"という名称のfileは、ID:16のdirectory配下にあり、idが13であることがわかる。
- j_drec_hashded_key_t、j_drec_val_tの定義は資料[1]のP.70、P.71参照。

j_drec_hashed_key_t



j_drec_val_t

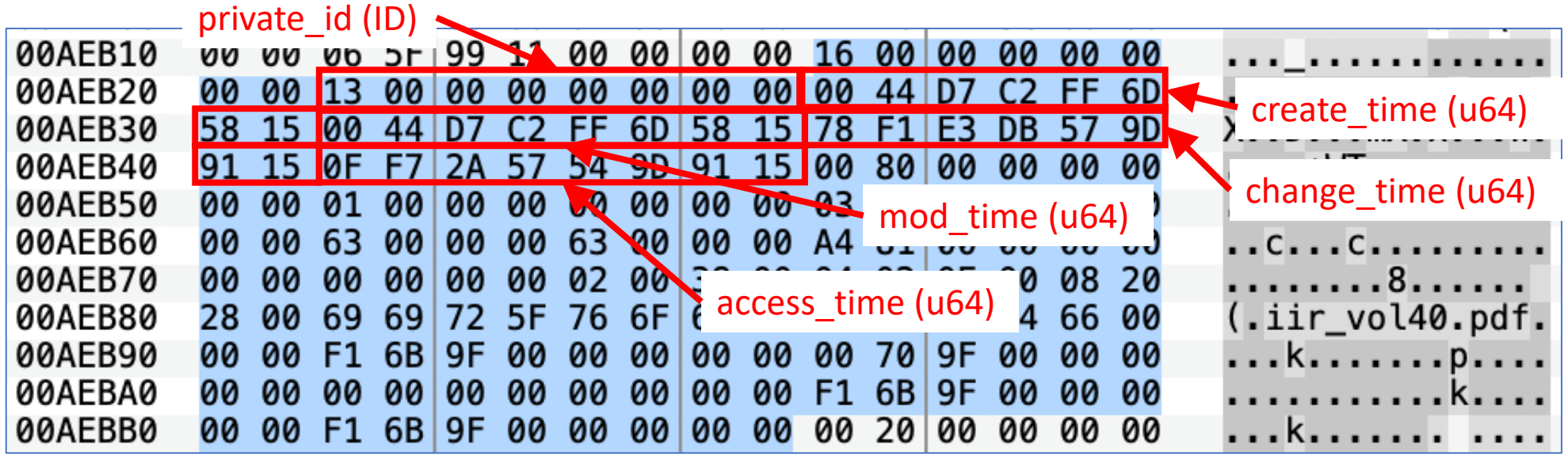


Volume内のfileを参照する (5)

- 例えばsample.raw内にある"iir_vol40.pdf"という名称のfile（ID:13）に関するタイムスタンプなどの情報はID:13のj_inode_val_tに格納されている。
- j_inode_key_t、j_inode_val_tの定義は資料[1]のP.65、P.66参照。

j_inode_key_t

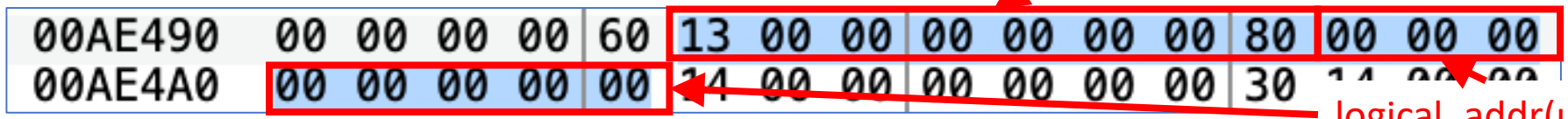
j_inode_val_t



Volume内のfileを参照する (6)

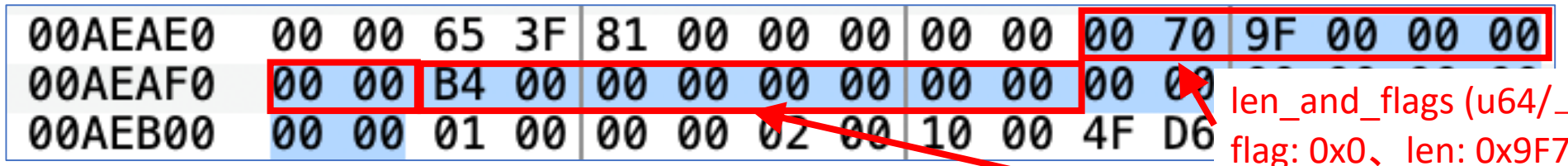
- 例えばsample.raw内にある"iir_vol40.pdf"という名称のfile（ID:13）のコンテンツデータへの参照はID:13のj_file_extent_val_tに格納されている。
- j_file_extent_key_t、j_file_extent_val_tの定義は資料[1]のP.91、P.92参照。

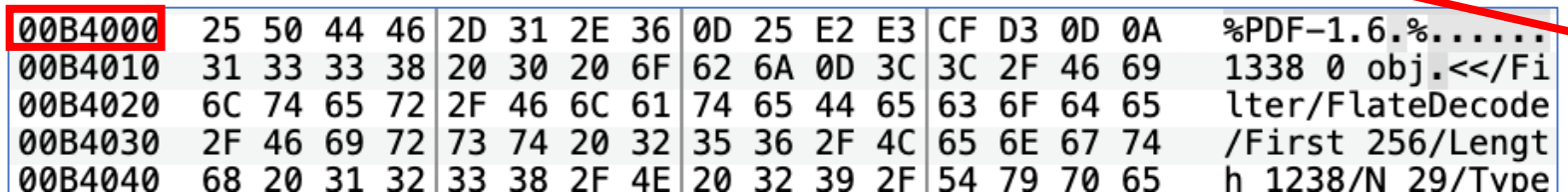
j_file_extent_key_t



ファイルがフラグメント化していると、1つのファイルに対してフラグメントの数だけj_file_extentが作成される。logical_addrは、そのj_file_extent（＝フラグメント）の、ファイル中でのオフセットを示す。

j_file_extent_val_t





Fletcher-64 checksumの計算

Fletcher-64 checksumの計算

- objectを変更した場合、object先頭のFletcher-64(o-cksum)を再計算する必要がある（checksumが検証できないobjectは利用されない）。
- 下記URLからダウンロードできるfletcherNbit.pyを使って計算することができる。
  - https://github.com/njaladan/hashpy
- 例えばsample.rawの0x2000から始まるblockのchecksumは左下のサンプルコード(f64_calc.py)を使う。

```
import fletcherNbit

target_image="path/to/sample.raw"
target_object=0x2000
object_length=0x1000

f = open(target_image, "rb")
f.seek(target_object+8)
data_for_checksum = f.read(object_length-8)
checksum = fletcherNbit.Fletcher64()
checksum.update(data_for_checksum)
print(checksum.cb_hexdigest())
```

f64_calc.py

左のサンプルコードを実行すると、以下の様な結果が得られる。この結果の上位4バイト、下位4バイトをそれぞれリトルエンディアンで結合すると、APFSで使われているFletcher-64 Checksumが得られる。

```
> python f64_calc.py
0xcfab4e8f89e06a36
```

```
0002000  8F 4E AB CF 36 6A E0 89
```

apfs-fuse

apfs-fuseのコンパイル (1)

- APFSマウント用ツール。以下のURLからダウンロードしてコンパイルして使う。
  - https://github.com/sgan81/apfs-fuse
- 以下で、SIFT (https://digital-forensics.sans.org/community/downloads) 環境でのコンパイル手順を紹介する。
  - aptコマンドで依存ライブラリをインストールする（gitに記載されているものと一部異なるので注意）。

```
> sudo apt update
> sudo apt install fuse libfuse-dev bzip2 libbz2-dev cmake git libattr1-dev cmake-curses-gui
```

  - Gitからダウンロードしてsubmoduleを更新する。

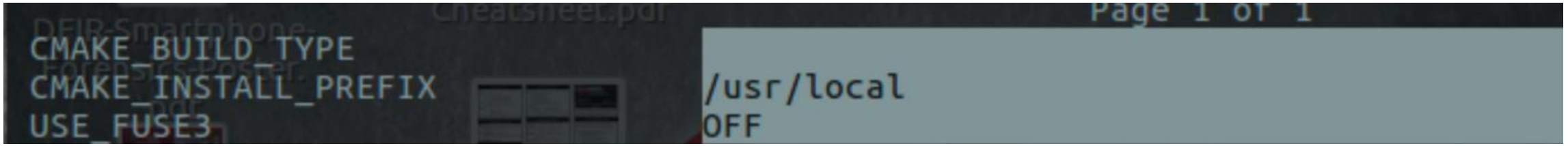
```
> git clone https://github.com/sgan81/apfs-fuse.git
> cd apfs-fuse
> git submodule init
> git submodule update
```

apfs-fuseのコンパイル (2)

- Buildフォルダを作り、buildする。

```
> mkdir build
> cd build
> cmake ..
> ccmake .　＃<- FUSE3をOFFにする
> make
```

- なお、ccmakeでは以下のような画面になる。矢印キーでUSE_FUSE3の行に移動し、ENTERキーを押下してONをOFFに変更する。その後cキー、gキーの順に押下するとconfigが更新されてccmakeが終了する。



apfs-fuseの使い方

- sample.rawイメージは以下のコマンドでマウントできる。

```
# apfs-fuse -s 0 <image file> <mount dir>
```

- アンマウント時は次のコマンドを使う。

```
# fusermount -u <mount dir>
```

apfs-dump

- apfs-fuseに含まれるapfs-dumpコマンドを使うと、イメージに含まれるすべてのobjectをdumpすることができる。

```
> apfs-dump <image file> <output.txt>
```

- apfsイメージの解析には極めて便利。

```
[paddr]          | cksum             | oid               | xid               | type     | subtype  | descript
-----------------+-------------------+-------------------+-------------------+----------+----------+---------
0000000000000000 | EFD8AACFB4DFE089 | 0000000000000001 | 0000000000000016 | 80000001 | 00000000 | Containe

0020 u32 magic            : 4253584E
0024 u32 block_size       : 00001000
0028 u64 block_count      : 0000000000001C98
0030 u64 features         : 0000000000000000  []
0038 u64 ro_compat_feat's : 0000000000000000  []
0040 u64 incompat_feat's  : 0000000000000002  [NX_INCOMPAT_VERSION2]
```

関連資料

[1] Apple File System Reference

- https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf
- 構造体の定義だけでなく、以下の各章の冒頭の説明は有用。
  - P.9 (Objects)
  - P.24 (Container)
  - P.41 (Object Maps)
  - P.64 (File-System Objects)
  - P.107 (B-Trees)

[2] APFS Internals

- https://objectivebythesea.com/v1/talks/OBTS_v1_Levin.pdf
- OS X Internalsの著者がAPFSの概要をまとめた資料。ポイントが簡潔にまとめられている。

[3] Decoding the APFS file system

- https://cyberforensicator.com/wp-content/uploads/2017/11/DIIN_698_Revisedproof.1-min-ilovepdf-compressed.pdf
- 資料[2]の次に読み易い解説。ただしApple公式資料の公開前に書かれたため、用語が異なっている。以下に一部の対応表を示す。

当該論文	公式
Volume Checkpoint Superblock	OBJECT_TYPE_FS
Volume Block	OBJECT_TYPE_BTREE - OBJECT_TYPE_OMAP
Table	OBJECT_TYPE_BTREE
Node ID#	Transaction id
B-Tree Object Map	OBJECT_TYPE_BTREE - OBJECT_TYPE_OMAP
B-Tree Root Node	OBJECT_TYPE_BTREE - OBJECT_TYPE_FSTREE (root node)
B-Tree Leaf Node	OBJECT_TYPE_BTREE - OBJECT_TYPE_FSTREE (leaf node)

その他

- [4] Apple File System (APFS)
  - https://github.com/libyal/libfsapfs/blob/master/documentation/Apple%20File%20System%20(APFS).asciidoc
  - 簡潔で構造体の定義が追い易い。
- [5] APFS filesystem format
  - https://blog.cugu.eu/post/apfs/
  - B-Treeの構造が比較的わかりやすく図示されている。
- [6] njaladan/hashpy
  - https://github.com/njaladan/hashpy
  - 各種hashライブラリのpython実装。Fletcherも含まれている。
- [7] Mounting an APFS image in Linux
  - http://az4n6.blogspot.com/2018/01/mounting-apfs-image-in-linux.html
  - apfs-fuseの使い方ガイド。