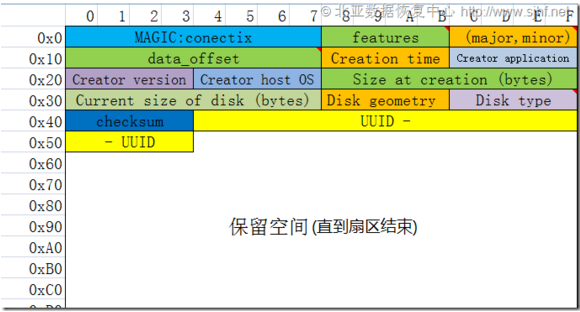
# Virtual Hard Disk Image Types

## Fixed Hard Disk Image

### VHD布局



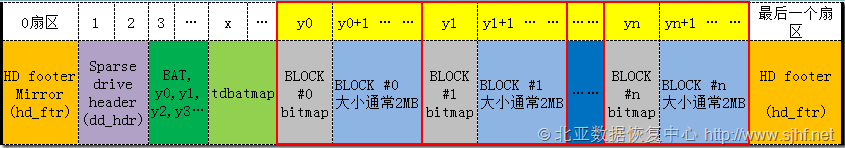
### HD footer(hd\_ftr)



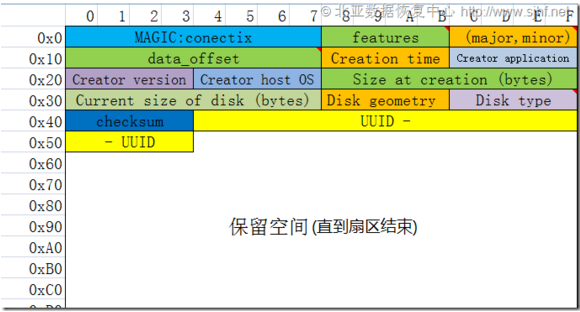
关于hd\_ftr中各个域的说明，请参考”Virtual Hard Disk Format Spec\_10\_18\_06.doc”。

## Dynamic Hard Disk Image

### VHD布局

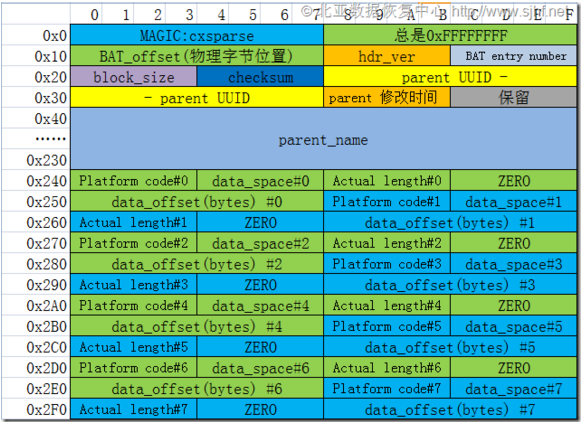


### HD footer(hd\_ftr)



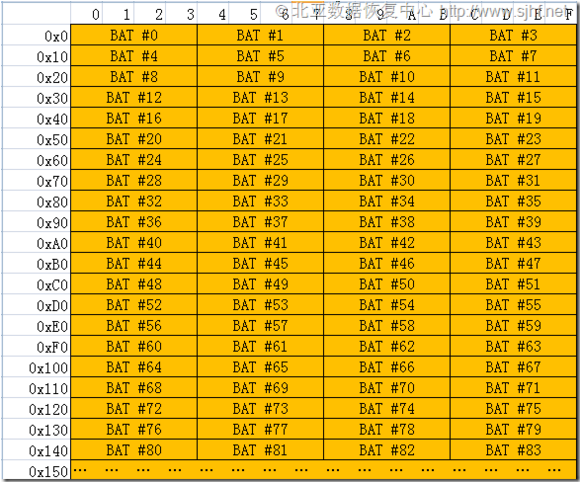
关于hd\_ftr中各个域的说明，请参考”Virtual Hard Disk Format Spec\_10\_18\_06.doc”。

### HD header(dd\_hdr)



关于dd\_hdr中各个域的说明，请参考”Virtual Hard Disk Format Spec\_10\_18\_06.doc”。

### BAT(Block allocation table)

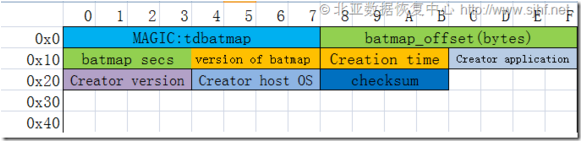


关于BAT的说明，请参考”Virtual Hard Disk Format Spec\_10\_18\_06.doc”。

在创建vhd的时候，会根据dd\_hdr::max\_bat\_size计算BAT区域有效空间大小并按照sector向上对齐计算BAT区域实际占用空间大小，最后将BAT区域有效空间中每一个BAT entry初始化为0xFFFFFFFF。

每一个有效的BAT entry都记录了虚拟磁盘中的某个Block（大小为VHD\_BLOCK\_SIZE，默认为2M）在VHD文件中的起始sector位置，如BAT #53中记录的值为0x12345678，则表示虚拟磁盘中第53个Block的内容存放于VHD文件中以0x12345678个sector为起始地址的Block中(实际上每个Block大于2M， 因为Block中包括bitmap和Data两部分，2M是Data部分的大小，bitmap区域的大小未计算在内，BAT entry中记录的值实际上是bitmap所在的sector)。

### Tdbatmap(包括batmap header + batmap)



**Batmap header结构如下：**

struct dd\_batmap\_hdr {

char cookie[8]; /\* should contain "tdbatmap" \*/

uint64\_t batmap\_offset; /\* byte offset to batmap \*/

uint32\_t batmap\_size; /\* batmap size in sectors \*/

uint32\_t batmap\_version; /\* version of batmap \*/

uint32\_t checksum; /\* batmap checksum -- 1's complement of batmap \*/

char marker; /\* generic marker field \*/

};

**dd\_batmap\_hdr::batmap\_offset**

记录tdbatmap中batmap区域的偏移，计算方式为：

dd\_batmap\_hdr的偏移 + 向上对齐到sector的dd\_batmap\_hdr区域大小。

**dd\_batmap\_hdr::batmap\_size**

记录tdbatmap中batmap区域的大小，计算方式为：

dd\_hdr::max\_bat\_size向上对齐到Byte，然后再向上对齐到sector。

**batmap**

batmap区域所占用的空间大小记录在dd\_batmap\_hdr::batmap\_size中，从batmap区域的第一个bit到第dd\_hdr::max\_bat\_size - 1个bit位中的每一个bit位，如果为1，则表明相应的block全满，否则表明相应的block中只有部分填充或者完全未被填充。

### BLOCK #n bitmap

通过dd\_hdr::block\_size可以计算出一个block包含多少个sector（按照规定，一定是2的n次方个），bitmap中就需要这么多个bit位去表示该block中的所有sector，但是bitmap区域所占用的空间是需要向上对齐到sector的。

### BLOCK #n

BLOCK #n的data区域(不包括bitmap部分)的起始sector一定是页对齐的。

## Differencing Hard Disk Image

差异磁盘是建立在一个固定或动态磁盘上的快照。其本意是差异磁盘中仅存储自创建备份点以来的所有改动。本质上，差异磁盘就是个动态磁盘。

# Metadata Manipulation

## Metadata layout

| **Dynamic Disk header fields** |
| --- |
| Copy of hard disk footer (512 bytes) |
| Dynamic Disk Header (1024 bytes) |
| BAT (Block Allocation table)  1. Array of absolute sector offsets into the file (uint32\_t).  2. Rounded up to a sector boundary.  3. Unused entries are marked as 0xFFFFFFFF.  4. max entries in dd\_hdr::max\_bat\_size. |
| Tdbatmap header  1. Fixed size: Rounded up (sizeof(vhd\_batmap\_header\_t)) to sector boundary.  2. sizeof(vhd\_batmap\_header\_t) is less than a sector. |
| Tdbatmap map  1. Rounded up ((dd\_hdr::max\_bat\_size + 7) >> 3) to sector boundary. |
| Parent locators  1. For differencing hard disk image only, no space occupied for other hard disk image types.  2. At most 8 parent locators. |
| Data Block 1  1. Indicated by vhd\_state::first\_db.  2. Data region must start from page boundary. |
| Data Block 2  1. Include both bitmap region and data region.  2. Bitmap region must be rounded up to sector boundary.  3. Data region must start from page boundary. |
| … |
| Data Block n  1. Include both bitmap region and data region.  2. Bitmap region must be rounded up to sector boundary.  3. Data region must start from page boundary. |
| Hard Disk Footer (512 bytes) |

## BAT

**BAT内存结构初始化**

相关数据结构：

vhd\_state::vhd\_bat\_state::bat [type: vhd\_bat\_t，对应于VHD文件中的BAT]

vhd\_state::vhd\_bat\_state::bat::bat [type: uint32\_t\*，uint32\_t的数组，数组中每一个元素记录某个block在VHD文件中的偏移]

vhd\_state::vhd\_bat\_state::bat\_buf [type: char\*，一个sector大小的内存区域，专门用于更新BAT]

宏：

#define bat\_entry(s, blk) ((s)->bat.bat.bat[(blk)])

该宏用于快速获取指定block @blk在BAT内存区域的地址。

相关调用链：

\_vhd\_open -> \_\_vhd\_open -> vhd\_initialize\_dynamic\_disk -> vhd\_initialize\_bat -> vhd\_read\_bat [初始化BAT（从VHD文件中读取BAT内容以初始化BAT内存结构），vhd\_state::vhd\_bat\_state::bat\_buf ]

**BAT内存结构更新**

首先在VHD文件中更新BAT（schedule\_bat\_write），当收到BAT更新响应的时候，更新BAT内存结构（finish\_bat\_write：bat\_entry(s, s->bat.pbw\_blk) = s->bat.pbw\_offset）。

**BAT在VHD文件中更新**

通过schedule\_bat\_write进行，生成请求并调用aio\_write。

**Batmap内存结构初始化**

相关数据结构：

vhd\_state::vhd\_bat\_state::bat\_map [type: vhd\_bat\_map\_t，对应于VHD文件中的batmap]

相关调用链：

\_vhd\_open -> \_\_vhd\_open -> vhd\_initialize\_dynamic\_disk -> vhd\_initialize\_bat -> vhd\_read\_batmap [batmap内存结构，从VHD文件中读取batmap内容以初始化batmap内存结构]

**Batmap内存结构更新**

在finish\_bitmap\_transaction中进行，如果更新bitmap之后检测到相应block的bitmap全为1，则设置该block的batmap为1.

**Batmap在VHD文件中更新**

Batmap在数据写过程中只会更新内存数据结构，并不会更新VHD文件中内容，而是在调用\_vhd\_close的时候调用vhd\_write\_batmap更新之。

## Bitmap cache

**Bitmap cache初始化**

相关数据结构：

vhd\_state::bitmap[VHD\_CACHE\_SIZE] [type: struct vhd\_bitmap\*，bitmap cache中已存在的bitmap]

vhd\_state::bitmap\_free[VHD\_CACHE\_SIZE] [type: struct vhd\_bitmap\*， bitmap cache中尚未使用的bitmap]

vhd\_state::bitmap\_list[VHD\_CACHE\_SIZE] [type: struct vhd\_bitmap，vhd\_state::bitmap和vhd\_state::bitmap\_free中记录的bitmap指针一定是vhd\_state::bitmap\_list中的元素的地址]

相关调用链：

\_vhd\_open -> \_\_vhd\_open -> vhd\_initialize\_dynamic\_disk -> vhd\_initialize\_bitmap\_cache [初始化时候，vhd\_state::bitmap\_free[#] = &vhd\_state::bitmap\_list[#]，vhd\_state::bitmap[#] = NULL]

**Bitmap cache更新**

schedule\_bitmap\_read -> alloc\_vhd\_bitmap + aio\_read + install\_bitmap [读取VHD文件上bitmap内容以更新bitmap cache，首先从bitmap cache中申请一个bitmap，然后调用aio\_read读取其内容，最后将其添加到bitmap cache中去]

alloc\_vhd\_bitmap [从bitmap cache中分配bitmap，首先检查vhd\_state::bitmap\_free中是否有空闲的bitmap，如果有，直接分配之，否则需要从vhd\_state::bitmap中剔除一个旧的bitmap，并分配该剔除的bitmap以满足当前block的需求]

Install\_bitmap [在vhd\_state::bitmap寻找一个空闲位置，并将该分配到的bitmap添加进去]

**Bitmap cache剔除**

模拟的LRU机制。

## Bitmap

**Bitmap内存结构初始化**

相关数据结构：

vhd\_bitmap::shadow [type: char\*，关于vhd\_bitmap::map的副本，在更新VHD文件中的bitmap之前，需要先更新该副本内容，并将该副本内容更新的VHD文件中，如果更新VHD文件失败，则会从vhd\_bitmap::map中拷贝内容覆盖vhd\_bitmap::shadow]

vhd\_bitmap::map [type: char\*，在更新bitmap到VHD文件成功之后，才将vhd\_bitmap::shadow中的最新内容拷贝到vhd\_bitmap::map中，见finish\_bitmap\_write]

Bitmap（确切的说，应该是关于某个block的bitmap）要么不在内存中，要么在bitmap cache中，bitmap cache中至多维护VHD\_CACHE\_SIZE = 32个bitmap。在读取关于某个block的数据，但是该block的bitmap不在bitmap cache中时，就会通过schedule\_bitmap\_read从VHD文件中读取关于该block的bitmap，并添加到bitmap cache中，从VHD文件中读取到的bitmap内容存放在vhd\_bitmap::map中，在成功读取之后需要将vhd\_bitmap::map中的内容拷贝到vhd\_bitmap::shadow中（见finish\_bitmap\_read）。

**Bitmap内存结构更新**

只有写操作才会涉及bitmap更新，更新按照如下步骤进行：

1. 更新vhd\_bitmap::shadow；
2. 将vhd\_bitmap::shadow中的内容写入VHD文件；
3. 如果更新VHD文件成功，则将vhd\_bitmap::shadow的内容拷贝到vhd\_bitmap::map中，如果更新VHD文件失败，则将vhd\_bitmap::map中的内容拷贝到vhd\_bitmap::shadow

**Bitmap在VHD文件中更新**

1. Bitmap更新到VHD文件是通过finish\_data\_write -> finish\_data\_transaction -> schedule\_bitmap\_write进行的，将vhd\_bitmap::shadow中的内容写入VHD文件。

## Footer

**Footer内存结构初始化**

相关结构：

vhd\_context\_t::footer [type: vhd\_footer\_t]

相关调用链：

\_\_vhd\_open -> vhd\_open -> vhd\_read\_footer [从VHD文件中读取footer内容]

**Footer在VHD文件中更新**

相关调用链：

allocate\_block -> write/pwrite/pwrite64 -> \_libvhd\_io\_pwrite -> \_libvhd\_io\_write\_bytes ->

vhd\_io\_write\_bytes -> \_\_vhd\_io\_dynamic\_write\_bytes ->

\_\_vhd\_io\_dynamic\_write\_bytes\_aligned -> vhd\_write\_footer [在分配新的block的时候，会重新写入footer中的内容到VHD文件的末尾，写入之前无需从VHD文件中读取，因为内存中已经有footer相关的数据了]

\_vhd\_close -> vhd\_write\_footer [关闭vhd时需要更新footer到VHD文件中]

# Read/Write Operation

## Write

**vhd\_queue\_write**

static void

vhd\_queue\_write(td\_driver\_t \*driver, td\_request\_t treq)

{

struct vhd\_state \*s = (struct vhd\_state \*)driver->data;

/\*循环处理，直到所有的涉及的sectors都处理了（当然，中途也可能退出）\*/

while (treq.secs) {

int err;

uint8\_t flags;

td\_request\_t clone;

err = 0;

flags = 0;

clone = treq;

/\*读取bitmap cache（包括BAT，batmap和bitmap），并根据返回值做相应处理，

\* 见read\_bitmap\_cache

\*/

switch (read\_bitmap\_cache(s, clone.sec, VHD\_OP\_DATA\_WRITE)) {

case -EINVAL:

err = -EINVAL;

goto fail;

case VHD\_BM\_BAT\_LOCKED:

err = -EBUSY;

goto fail;

/\*clone.sec所在的block尚未分配，需要先分配，然后再写入数据\*/

case VHD\_BM\_BAT\_CLEAR:

/\*因为要分配block，所以本次写还需要更新BAT和bitmap\*/

flags = (VHD\_FLAG\_REQ\_UPDATE\_BAT |

VHD\_FLAG\_REQ\_UPDATE\_BITMAP);

/\*至多写从clone.sec开始的clone.secs个sector\*/

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

/\*见schedule\_data\_write\*/

err = schedule\_data\_write(s, clone, flags);

if (err)

goto fail;

break;

/\*clone.sec所在的block已经分配，但是clone.sec所在的sector尚未写过数据\*/

case VHD\_BM\_BIT\_CLEAR:

/\*因为clone.sec所在的sector尚未写入过数据，所以需要更新bitmap\*/

flags = VHD\_FLAG\_REQ\_UPDATE\_BITMAP;

/\*在clone.sec所在的bitmap中查找从clone.sec开始的至多clone.secs个连续

\* 的都为0的bit位，这些连续的bit位都为0的sectors可以一块写入

\*/

clone.secs = read\_bitmap\_cache\_span(s, clone.sec, clone.secs, 0);

/\*见schedule\_data\_write\*/

err = schedule\_data\_write(s, clone, flags);

if (err)

goto fail;

break;

/\*clone.sec所在的block已经分配，且clone.sec所在的sector已经写过数据\*/

case VHD\_BM\_BIT\_SET:

/\*在clone.sec所在的bitmap中查找从clone.sec开始的至多clone.secs个连续

\* 的都为1的bit位，这些连续的bit位都为1的sectors可以一块写入

\*/

clone.secs = read\_bitmap\_cache\_span(s, clone.sec, clone.secs, 1);

/\*见schedule\_data\_write\*/

err = schedule\_data\_write(s, clone, 0);

if (err)

goto fail;

break;

/\* clone.sec所在的block的bitmap不再bitmap cache中，需要首先从vhd文件中

\* 读取bitmap并添加到bitmap cache中，然后再执行IO

\*/

case VHD\_BM\_NOT\_CACHED:

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

/\*首先从vhd文件中读取bitmap并添加到bitmap cache中，见

\* schedule\_bitmap\_read

\*/

err = schedule\_bitmap\_read(s, clone.sec / s->spb);

if (err)

goto fail;

/\*将该写请求添加到clone.sec所在的bitmap的waiting队列中，该队列中的请

\* 求只在bitmap读取成功之后才会被处理（调度）

\*

\* 见\_\_vhd\_queue\_request

\*/

err = \_\_vhd\_queue\_request(s, VHD\_OP\_DATA\_WRITE, clone);

if (err)

goto fail;

break;

case VHD\_BM\_READ\_PENDING:

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

/\*将该写请求添加到clone.sec所在的bitmap的waiting队列中，该队列中的请

\* 求只在bitmap读取成功之后才会被处理（调度）\*/

err = \_\_vhd\_queue\_request(s, VHD\_OP\_DATA\_WRITE, clone);

if (err)

goto fail;

break;

default:

ASSERT(0);

break;

}

/\*根据本次处理的sector的个数（保存在clone.secs中），调整下一个即将处理的

\* sector（treq.sec）、剩余的等待处理的sectors个数（treq.secs）以及数据指针

\*（treq.buf）

\*/

treq.sec += clone.secs;

treq.secs -= clone.secs;

treq.buf += vhd\_sectors\_to\_bytes(clone.secs);

continue;

fail:

clone.secs = treq.secs;

td\_complete\_request(clone, err);

break;

}

}

**read\_bitmap\_cache**

static int

read\_bitmap\_cache(struct vhd\_state \*s, uint64\_t sector, uint8\_t op)

{

uint32\_t blk, sec;

struct vhd\_bitmap \*bm;

/\*对于fixed类型的VHD，每一个block都是存在的\*/

/\* in fixed disks, every block is present \*/

if (s->vhd.footer.type == HD\_TYPE\_FIXED)

return VHD\_BM\_BIT\_SET;

/\*计算@sector所在的block及其block内部偏移\*/

/\* the extent the logical sector falls in \*/

blk = sector / s->spb;

/\* offset within the extent the logical sector is located \*/

sec = sector % s->spb;

/\*block数目不能超过s->vhd.header.max\_bat\_size\*/

if (blk > s->vhd.header.max\_bat\_size) {

DPRINTF("ERROR: sec %"PRIu64" out of range, op = %d\n",

sector, op);

return -EINVAL;

}

/\* BAT区域中该block所在的记录为DD\_BLK\_UNUSED，表示它并未被分配，

\* 如果是写请求，且block和当前正处于pending write状态的block不相同，且当前

\* BAT处于locked状态，则返回VHD\_BM\_BAT\_LOCKED表明当前正在处理该处于pending

\* write状态的block，需要等待，否则返回VHD\_BM\_BAT\_CLEAR表明需要为当前的写

\* 请求分配新的block

\*

\* 关于BAT区域的设置请参考“BAT更新”

\*/

if (bat\_entry(s, blk) == DD\_BLK\_UNUSED) {

if (op == VHD\_OP\_DATA\_WRITE &&

s->bat.pbw\_blk != blk && bat\_locked(s))

return VHD\_BM\_BAT\_LOCKED;

return VHD\_BM\_BAT\_CLEAR;

}

/\*至此，BAT区域中该block所在的记录肯定不是DD\_BLK\_UNUSED，则检查batmap，

\* 看该block所在的bit位是否为1，如果是，则表明该block中所有sector都已填充数

\* 据，无需再去bitmap区域检查@sector所在的bit位是否为1了，返回

\* VHD\_BM\_BIT\_SET，表明该sector中有有效数据

\*

\* 关于batmap区域的设置请参考“batmap更新”

\*/

if (test\_batmap(s, blk)) {

DBG(TLOG\_DBG, "batmap set for 0x%04x\n", blk);

return VHD\_BM\_BIT\_SET;

}

/\*检查bitmap cache中是否有关于该block的bitmap，没有则返回

\* VHD\_BM\_NOT\_CACHED，表明需要去读取关于该block的bitmap

\*

\* 关于bitmap cache的操作请参考“bitmap cache更新”

\*/

bm = get\_bitmap(s, blk);

if (!bm)

return VHD\_BM\_NOT\_CACHED;

/\*bitmap cache中有关于该block的bitmap，则在bitmap lru中调整该block的bitmap\*/

/\* bump lru count \*/

touch\_bitmap(s, bm);

/\*如果该block的bitmap设置了VHD\_FLAG\_BM\_READ\_PENDING，则暂时不能读取之，

\* 而VHD\_FLAG\_BM\_READ\_PENDING只在调度读取bitmap的过程中被设置（调度到

\* bitmap读请求的时候设置，读取成功之后清理）

\*/

if (test\_vhd\_flag(bm->status, VHD\_FLAG\_BM\_READ\_PENDING))

return VHD\_BM\_READ\_PENDING;

/\*读取该block的bitmap，查看指定sector所在的bit位是1还是0，是1则返回

\* VHD\_BM\_BIT\_SET ， 否则返回VHD\_BM\_BIT\_CLEAR

\*/

return ((vhd\_bitmap\_test(&s->vhd, bm->map, sec)) ?

VHD\_BM\_BIT\_SET : VHD\_BM\_BIT\_CLEAR);

}

**schedule\_data\_write**

/\*schedule\_data\_write不只涉及数据更新，也涉及元数据的更新，是否更新元数据，以及更\* 新哪些元数据，是通过@flags来控制的

\*/

static int

schedule\_data\_write(struct vhd\_state \*s, td\_request\_t treq, vhd\_flag\_t flags)

{

int err;

uint64\_t offset;

uint32\_t blk = 0, sec = 0;

struct vhd\_bitmap \*bm = NULL;

struct vhd\_request \*req;

/\*对于fixed类型的vhd，无需更新元数据部分，直接写数据\*/

if (s->vhd.footer.type == HD\_TYPE\_FIXED) {

offset = vhd\_sectors\_to\_bytes(treq.sec);

goto make\_request;

}

/\*计算treq.sec所在的block及其在block内部偏移\*/

blk = treq.sec / s->spb;

sec = treq.sec % s->spb;

/\*获取计算得到的block @blk所在的起始sector\*/

offset = bat\_entry(s, blk);

/\*需要新分配block\*/

if (test\_vhd\_flag(flags, VHD\_FLAG\_REQ\_UPDATE\_BAT)) {

/\*通常，VHD\_FLAG\_OPEN\_PREALLOCATE都会被设置\*/

if (test\_vhd\_flag(s->flags, VHD\_FLAG\_OPEN\_PREALLOCATE))

/\*分配block，见allocate\_block\*/

err = allocate\_block(s, blk);

else

err = update\_bat(s, blk);

if (err)

return err;

/\*分配到的block的起始sector\*/

offset = s->bat.pbw\_offset;

}

/\*即将写入的数据所在的绝对偏移\*/

offset += s->bm\_secs + sec;

offset = vhd\_sectors\_to\_bytes(offset);

make\_request:

/\*从vhd\_request cache中分配vhd\_request\*/

req = alloc\_vhd\_request(s);

if (!req)

return -EBUSY;

req->treq = treq;

req->flags = flags;

req->op = VHD\_OP\_DATA\_WRITE;

req->next = NULL;

/\*需要更新bitmap\*/

if (test\_vhd\_flag(flags, VHD\_FLAG\_REQ\_UPDATE\_BITMAP)) {

/\*如果本次写新分配了block，则bitmap会在allocate block中设置，其它情况也一

\* 定能确保bitmap在bitmap cache中，所以get\_bitmap一定能从bitmap cache中

\* 获取bitmap

\*/

bm = get\_bitmap(s, blk);

ASSERT(bm && bitmap\_valid(bm));

/\*锁定该bitmap，直到bitmap更新成功之后才能解除锁定\*/

lock\_bitmap(bm);

/\*如果与该bitmap相关的transaction已经关闭，则将本次数据写请求添加到bitmap

\* 的queue中，该队列中保存的都是那些等待在下一个transaction中执行的请求，

\* 并且为该请求设置VHD\_FLAG\_REQ\_QUEUED标识

\*/

if (bm->tx.closed) {

/\*当数据读写结束的时候会设置transaction关闭，或者在元数据读写失败的情

\* 况下也会设置transaction关闭

\*/

add\_to\_tail(&bm->queue, req);

set\_vhd\_flag(req->flags, VHD\_FLAG\_REQ\_QUEUED);

} else

/\*否则直接添加到该bitmap的当前transaction中，参考“bitmap更新”\*/

add\_to\_transaction(&bm->tx, req);

} else if (sec == 0 && /\* first sector inside data block \*/

s->vhd.footer.type != HD\_TYPE\_FIXED &&

bat\_entry(s, blk) != s->first\_db &&

test\_batmap(s, blk))

/\*否则sec所在的bit位一定已经设置为1，如果写的是除第1个block外的其它block

\* 的第1个sector，且该block全部填满，则（在padded bitmap区域中）写入冗

\* 余bitmap，从schedule\_redundant\_bm\_write的注释来看，是为了性能的考虑，

\* 见schedule\_redundant\_bm\_write，

\*/

schedule\_redundant\_bm\_write(s, blk);

/\*异步写\*/

aio\_write(s, req, offset);

DBG(TLOG\_DBG, "%s: lsec: 0x%08"PRIx64", blk: 0x%04x, sec: 0x%04x, "

"nr\_secs: 0x%04x, offset: 0x%08"PRIx64", flags: 0x%08x\n",

s->vhd.file, treq.sec, blk, sec, treq.secs, offset, req->flags);

return 0;

}

**allocate\_block**

static int

allocate\_block(struct vhd\_state \*s, uint32\_t blk)

{

int err, gap;

uint64\_t offset, size;

struct vhd\_bitmap \*bm;

ssize\_t count;

uint64\_t next\_db;

ASSERT(bat\_entry(s, blk) == DD\_BLK\_UNUSED);

if (bat\_locked(s)) {

ASSERT(s->bat.pbw\_blk == blk);

if (s->bat.req.error)

return -EBUSY;

return 0;

}

/\*s->next\_db中记录的是下一个尚未分配的空间的首地址（以sector为单位）\*/

gap = 0;

offset = vhd\_sectors\_to\_bytes(s->next\_db);

next\_db = s->next\_db;

/\*block的数据区域（不包括bitmap区域）必须是页对齐的\*/

/\* data region of segment should begin on page boundary \*/

if ((next\_db + s->bm\_secs) % s->spp) {

gap = (s->spp - ((next\_db + s->bm\_secs) % s->spp));

next\_db += gap;

}

if (next\_db > UINT\_MAX)

return -ENOSPC;

s->next\_db = next\_db;

s->bat.pbw\_blk = blk;

s->bat.pbw\_offset = s->next\_db;

DBG(TLOG\_DBG, "blk: 0x%04x, pbwo: 0x%08"PRIx64"\n",

blk, s->bat.pbw\_offset);

if (lseek(s->vhd.fd, offset, SEEK\_SET) == (off\_t)-1) {

ERR(s, -errno, "lseek failed\n");

return -errno;

}

/\*将从offset开始的长度为(s->spb + s->bm\_secs + gap)的区域填充上0，这里是同步写\*/

size = vhd\_sectors\_to\_bytes(s->spb + s->bm\_secs + gap);

/\*调用链为write -> \_libvhd\_io\_pwrite -> \_libvhd\_io\_write\_bytes -> vhd\_io\_write\_bytes ->

\* \_\_vhd\_io\_dynamic\_write\_bytes -> \_\_vhd\_io\_dynamic\_write\_bytes\_aligned，参考

\* \_\_vhd\_io\_dynamic\_write\_bytes\_aligned

\*/

count = write(s->vhd.fd, vhd\_zeros(size), size);

if (count != size) {

err = count < 0 ? -errno : -ENOSPC;

ERR(s, -errno,

"write failed (%zd, offset %"PRIu64")\n", count, offset);

return err;

}

/\*如果bitmap cache中没有关于该block的bitmap，则分配一个新的bitmap并添加到

\* bitmap cache中

\*/

/\* empty bitmap could already be in

\* cache if earlier bat update failed \*/

bm = get\_bitmap(s, blk);

if (!bm) {

/\* install empty bitmap in cache \*/

err = alloc\_vhd\_bitmap(s, &bm, blk);

if (err)

return err;

install\_bitmap(s, bm);

}

/\*现在的BAT和block所在的bitmap都是不能读取的，但是bitmap现在不能更新，必

\* 须在写成功之后才能去更新之

\*/

lock\_bat(s);

lock\_bitmap(bm);

/\*因为新分配了block，所以要更新BAT\*/

schedule\_bat\_write(s);

add\_to\_transaction(&bm->tx, &s->bat.req);

return 0;

}

static int

\_\_vhd\_io\_dynamic\_write\_bytes\_aligned(vhd\_context\_t \*ctx,

char \*buf, size\_t size, uint64\_t off)

{

char \*map;

int i, err, ret;

uint64\_t blk\_off, blk\_size, blk\_start;

uint32\_t blk, bytes, first\_sec, last\_sec;

if (off & (VHD\_SECTOR\_SIZE - 1) || size & (VHD\_SECTOR\_SIZE - 1))

return -EINVAL;

/\*确保BAT存在于内存中\*/

err = vhd\_get\_bat(ctx);

if (err)

return err;

/\*确保batmap存在于内存中\*/

if (vhd\_has\_batmap(ctx)) {

err = vhd\_get\_batmap(ctx);

if (err)

return err;

}

map = NULL;

blk\_size = vhd\_sectors\_to\_bytes(ctx->spb);

do {

/\*计算当前要更新的block、偏移和大小\*/

blk = off / blk\_size;

blk\_off = off % blk\_size;

bytes = MIN(blk\_size - blk\_off, size);

first\_sec = blk\_off >> VHD\_SECTOR\_SHIFT;

last\_sec = secs\_round\_up\_no\_zero(blk\_off + bytes);

blk\_start = ctx->bat.bat[blk];

if (blk\_start == DD\_BLK\_UNUSED) {

err = \_\_vhd\_io\_allocate\_block(ctx, blk);

if (err)

goto fail;

blk\_start = ctx->bat.bat[blk];

}

/\*block中数据区的起始地址\*/

blk\_start = vhd\_sectors\_to\_bytes(blk\_start + ctx->bm\_secs);

err = vhd\_pwrite(ctx, buf, bytes, blk\_start + blk\_off);

if (err)

goto fail;

if (vhd\_has\_batmap(ctx) &&

vhd\_batmap\_test(ctx, &ctx->batmap, blk))

goto next;

/\*准备bitmap，用于更新bitmap到VHD文件中\*/

err = vhd\_read\_bitmap(ctx, blk, &map);

if (err) {

map = NULL;

goto fail;

}

/\*将起于first\_sec，止于last\_sec的所有sector都在bitmap中置位\*/

for (i = first\_sec; i < last\_sec; i++)

vhd\_bitmap\_set(ctx, map, i);

/\*将bitmap写入到VHD文件中\*/

err = vhd\_write\_bitmap(ctx, blk, map);

if (err)

goto fail;

if (vhd\_has\_batmap(ctx)) {

for (i = 0; i < ctx->spb; i++)

/\*bitmap中至少有一个bit位为0，则无需设置batmap，否则如果bitmap

\* 中所有bit位全为1，则设置batmap为1

\*/

if (!vhd\_bitmap\_test(ctx, map, i)) {

free(map);

map = NULL;

goto next;

}

/\*设置batmap为1，并写入到VHD文件中\*/

vhd\_batmap\_set(ctx, &ctx->batmap, blk);

err = vhd\_write\_batmap(ctx, &ctx->batmap);

if (err)

goto fail;

}

free(map);

map = NULL;

/\*处理剩余部分\*/

next:

size -= bytes;

off += bytes;

buf += bytes;

} while (size);

err = 0;

out:

/\*最后更新footer\*/

ret = vhd\_write\_footer(ctx, &ctx->footer);

return (err ? err : ret);

fail:

free(map);

goto out;

}

int

vhd\_write\_footer(vhd\_context\_t \*ctx, vhd\_footer\_t \*footer)

{

int err;

off64\_t off;

/\*找到文件的末尾\*/

if (ctx->is\_block)

err = vhd\_footer\_offset\_at\_eof(ctx, &off);

else

err = vhd\_end\_of\_data(ctx, &off);

if (err)

return err;

/\*在文件末尾写入footer\*/

err = vhd\_write\_footer\_at(ctx, footer, off);

if (err)

return err;

/\*如果不是块设备文件，则调用ftruncate设置文件大小\*/

if (!ctx->is\_block) {

err = ftruncate(ctx->fd, off + sizeof(vhd\_footer\_t));

if (err)

return -errno;

}

if (!vhd\_type\_dynamic(ctx))

return 0;

/\*对于dynamic类型的VHD还要在头部更新footer\*/

return vhd\_write\_footer\_at(ctx, footer, 0);

}

static void

schedule\_redundant\_bm\_write(struct vhd\_state \*s, uint32\_t blk)

{

uint64\_t offset;

struct vhd\_request \*req;

ASSERT(s->vhd.footer.type != HD\_TYPE\_FIXED);

ASSERT(test\_batmap(s, blk));

/\*分配请求，设置要写的内容，要写的数据的起始偏移，要写的数据的大小等，其中涉

\* 及到s->padbm\_buf，s->padbm\_size两个变量，看看他们是在哪里被设置的（参考

\* vhd\_initialize\_dynamic\_disk）：

\*

\* s->padbm\_size实际上是s->bm\_secs向上对齐到页后的大小，代码如下：

\* s->padbm\_size = (s->bm\_secs / getpagesize()) \* getpagesize();

\* if (s->bm\_secs % getpagesize())

\* s->padbm\_size += getpagesize();

\*

\* s->padbm\_buf 是大小为 s->padbm\_size的内存，其布局及初始状态为：

\* |0000 ... ... ... ... 0000|1111 ... ... ... ... 1111|

\* |-----> padding <--------|-----> bitmap <--------|

\*

\* err = posix\_memalign(&buf, 512, s->padbm\_size);

\* s->padbm\_buf = buf;

\* bm\_size = s->bm\_secs << VHD\_SECTOR\_SHIFT;

\* memset(s->padbm\_buf, 0, s->padbm\_size - bm\_size);

\* memset(s->padbm\_buf + (s->padbm\_size - bm\_size), ~0, bm\_size);

\* s->debug\_skipped\_redundant\_writes = 0;

\*/

req = alloc\_vhd\_request(s);

if (!req)

return;

req->treq.buf = s->padbm\_buf;

/\*bat\_entry中记录的关于该block的bitmap起始地址（而不是数据区域的始地址）\*/

offset = bat\_entry(s, blk);

ASSERT(offset != DD\_BLK\_UNUSED);

offset <<= VHD\_SECTOR\_SHIFT;

offset -= s->padbm\_size - (s->bm\_secs << VHD\_SECTOR\_SHIFT);

req->op = VHD\_OP\_REDUNDANT\_BM\_WRITE;

req->treq.sec = blk \* s->spb;

req->treq.secs = s->padbm\_size >> VHD\_SECTOR\_SHIFT;

req->next = NULL;

DBG(TLOG\_DBG, "blk: %u, writing redundant bitmap at %" PRIu64 "\n",

blk, offset);

/\*将req->treq.secs个sector写入从offset开始的地方，要写的数据内容存放在

\* req->treq.buf中

\*/

aio\_write(s, req, offset);

}

static int

schedule\_bat\_write(struct vhd\_state \*s)

{

int i;

uint32\_t blk;

char \*buf;

uint64\_t offset;

struct vhd\_request \*req;

ASSERT(bat\_locked(s));

/\*s->bat.bat\_buf是一个sector大小的内存，专用于更新BAT（初始化过程参考：\_vhd\_open

\* -> \_\_vhd\_open -> vhd\_initialize\_dynamic\_disk -> vhd\_initialize\_bat）-> vhd\_read\_bat +

\* vhd\_read\_batmap

\*

\* 注意：更新BAT并不会去更新batmap，batmap在\_vhd\_open的时候从vhd文件中读

\* 取内容初始化内存结构的batmap，然后在close之前所有关于batmap的操作都是在

\* 内存中进行的，直到显示调用\_vhd\_close的时候才会将之刷入到vhd文件中

\*/

req = &s->bat.req;

buf = s->bat.bat\_buf;

blk = s->bat.pbw\_blk;

init\_vhd\_request(s, req);

/\*每次要更新某个block的BAT，虽然要更新的只是4个字节，但是会将这4个字节所

\* 在的sector的内容一起更新下去（因为磁盘操作的最小单位是sector），然后将该block

\* 所对应的那4个字节设置为该block的起始地址

\*/

memcpy(buf, &bat\_entry(s, blk - (blk % 128)), 512);

((uint32\_t \*)buf)[blk % 128] = s->bat.pbw\_offset;

/\*以大端模式编码\*/

for (i = 0; i < 128; i++)

BE32\_OUT(&((uint32\_t \*)buf)[i]);

/\*设置写偏移\*/

offset = s->vhd.header.table\_offset + (blk - (blk % 128)) \* 4;

req->treq.secs = 1;

req->treq.buf = buf;

req->op = VHD\_OP\_BAT\_WRITE;

req->next = NULL;

/\*异步写\*/

aio\_write(s, req, offset);

set\_vhd\_flag(s->bat.status, VHD\_FLAG\_BAT\_WRITE\_STARTED);

DBG(TLOG\_DBG, "blk: 0x%04x, pbwo: 0x%08"PRIx64", "

"table\_offset: 0x%08"PRIx64"\n", blk, s->bat.pbw\_offset, offset);

return 0;

}

## Read

static void

vhd\_queue\_read(td\_driver\_t \*driver, td\_request\_t treq)

{

struct vhd\_state \*s = (struct vhd\_state \*)driver->data;

DBG(TLOG\_DBG, "%s: lsec: 0x%08"PRIx64", secs: 0x%04x (seg: %d)\n",

s->vhd.file, treq.sec, treq.secs, treq.sidx);

/\*循环处理，直到所有的涉及的sectors都处理了（当然，中途也可能退出）\*/

while (treq.secs) {

int err;

td\_request\_t clone;

err = 0;

clone = treq;

/\*对于每一个即将处理的sector来说，都是首先读取bitmap cache，根据返回值做

\* 相应的处理

\*/

switch (read\_bitmap\_cache(s, clone.sec, VHD\_OP\_DATA\_READ)) {

case -EINVAL:

err = -EINVAL;

goto fail;

/\*当前要读取的block在BAT区域中记录的是DD\_BLK\_UNUSED，表示当前VHD文

\* 件中没有关于该block的数据，需要去parent VHD文件中读取（只有differencing

\* 类型的VHD文件有parent，对于其它类型的VHD文件则直接填充0返回即可）

\*/

case VHD\_BM\_BAT\_CLEAR:

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

/\*重定向到parent VHD中读取，如果不是differencing 类型的VHD文件，则直

\* 接填充0返回（这也是由td\_forward\_request完成的）

\*/

td\_forward\_request(clone);

break;

/\*当前要读取的sector所在的bit位为0，则也需要重定向到parent VHD文件中读

\* 取（只有differencing类型的VHD文件有parent，对于其它类型的VHD文件则直

\* 接填充0返回即可）

\*/

case VHD\_BM\_BIT\_CLEAR:

clone.secs = read\_bitmap\_cache\_span(s, clone.sec, clone.secs, 0);

/\*重定向到parent VHD中读取，如果不是differencing 类型的VHD文件，则直

\* 接填充0返回（这也是由td\_forward\_request完成的）

\*/

td\_forward\_request(clone);

break;

case VHD\_BM\_BIT\_SET:

/\*找到从clone.sec开始，最大长度为clone.secs的连续的都为1的bit，这些

\* bit所代表的sector是可以一起读取的

\*/

clone.secs = read\_bitmap\_cache\_span(s, clone.sec, clone.secs, 1);

/\*准备读请求并异步读取\*/

err = schedule\_data\_read(s, clone, 0);

if (err)

goto fail;

break;

/\*bitmap cache中没有关于当前读取的block相关的bitmap，需要先读取其bitmap

\* 并添加到bitmap cache中

\*/

case VHD\_BM\_NOT\_CACHED:

err = schedule\_bitmap\_read(s, clone.sec / s->spb);

if (err)

goto fail;

/\*在关于bitmap的读请求返回之前，关于数据的读请求必须排队等待\*/

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

err = \_\_vhd\_queue\_request(s, VHD\_OP\_DATA\_READ, clone);

if (err)

goto fail;

break;

/\*已经发送了读取bitmap的请求，但是请求尚未完成，当前的关于数据的读请求

\* 必须排队等待

\*/

case VHD\_BM\_READ\_PENDING:

clone.secs = MIN(clone.secs, s->spb - (clone.sec % s->spb));

err = \_\_vhd\_queue\_request(s, VHD\_OP\_DATA\_READ, clone);

if (err)

goto fail;

break;

case VHD\_BM\_BAT\_LOCKED:

default:

ASSERT(0);

break;

}

/\*根据前面处理的连续的sector的数目更新treq，为后续读做准备\*/

treq.sec += clone.secs;

treq.secs -= clone.secs;

treq.buf += vhd\_sectors\_to\_bytes(clone.secs);

continue;

fail:

clone.secs = treq.secs;

td\_complete\_request(clone, err);

break;

}

}

## Create

int

vhd\_create(const char \*name, uint64\_t bytes, int type, uint64\_t mbytes,

vhd\_flag\_creat\_t flags)

{

return \_\_vhd\_create(name, NULL, bytes, type, mbytes, flags);

}

static int

\_\_vhd\_create(const char \*name, const char \*parent, uint64\_t bytes, int type,

uint64\_t mbytes, vhd\_flag\_creat\_t flags)

{

int err;

off64\_t off;

vhd\_context\_t ctx;

uint64\_t size, psize, blks;

switch (type) {

case HD\_TYPE\_DIFF:

if (!parent)

return -EINVAL;

case HD\_TYPE\_FIXED:

case HD\_TYPE\_DYNAMIC:

break;

default:

return -EINVAL;

}

if (strnlen(name, VHD\_MAX\_NAME\_LEN - 1) == VHD\_MAX\_NAME\_LEN - 1)

return -ENAMETOOLONG;

if (bytes && mbytes && mbytes < bytes)

return -EINVAL;

memset(&ctx, 0, sizeof(vhd\_context\_t));

psize = 0;

blks = (bytes + VHD\_BLOCK\_SIZE - 1) >> VHD\_BLOCK\_SHIFT;

/\* If mbytes is provided (virtual-size-for-metadata-preallocation),

\* create the VHD of size mbytes, which will create the BAT & the

\* batmap of the appropriate size. Once the BAT & batmap are

\* initialized, reset the virtual size to the requested one.

\*/

if (mbytes)

blks = (mbytes + VHD\_BLOCK\_SIZE - 1) >> VHD\_BLOCK\_SHIFT;

size = blks << VHD\_BLOCK\_SHIFT;

ctx.fd = open(name, O\_WRONLY | O\_CREAT |

O\_TRUNC | O\_LARGEFILE | O\_DIRECT, 0644);

if (ctx.fd == -1)

return -errno;

ctx.file = strdup(name);

if (!ctx.file) {

err = -ENOMEM;

goto out;

}

/\*检查该VHD是否是块设备\*/

err = vhd\_test\_file\_fixed(ctx.file, &ctx.is\_block);

if (err)

goto out;

/\*初始化footer\*/

vhd\_initialize\_footer(&ctx, type, size);

if (type == HD\_TYPE\_FIXED) {

err = vhd\_initialize\_fixed\_disk(&ctx);

if (err)

goto out;

} else {

int raw = vhd\_flag\_test(flags, VHD\_FLAG\_CREAT\_PARENT\_RAW);

/\*初始化header\*/

err = vhd\_initialize\_header(&ctx, parent, size, raw, &psize);

if (err)

goto out;

/\*初始化batmap并写入VHD文件\*/

err = vhd\_create\_batmap(&ctx);

if (err)

goto out;

/\*初始化BAT并写入VHD文件\*/

err = vhd\_create\_bat(&ctx);

if (err)

goto out;

if (type == HD\_TYPE\_DIFF) {

/\*更新parent locator到VHD文件中\*/

err = vhd\_write\_parent\_locators(&ctx, parent);

if (err)

goto out;

}

}

if (mbytes) {

/\* set the virtual size to the requested size \*/

if (bytes) {

blks = (bytes + VHD\_BLOCK\_SIZE - 1) >> VHD\_BLOCK\_SHIFT;

size = blks << VHD\_BLOCK\_SHIFT;

}

else {

size = psize;

}

ctx.footer.orig\_size = size;

err = vhd\_set\_virt\_size\_no\_write(&ctx, size);

if (err)

goto out;

}

if (type != HD\_TYPE\_FIXED) {

/\*写footer在VHD文件的头部\*/

err = vhd\_write\_footer\_at(&ctx, &ctx.footer, 0);

if (err)

goto out;

/\*写header部分到VHD文件\*/

err = vhd\_write\_header\_at(&ctx, &ctx.header, VHD\_SECTOR\_SIZE);

if (err)

goto out;

}

err = vhd\_seek(&ctx, 0, SEEK\_END);

if (err)

goto out;

off = vhd\_position(&ctx);

if (off == (off64\_t)-1) {

err = -errno;

goto out;

}

if (ctx.is\_block)

off -= sizeof(vhd\_footer\_t);

/\*写footer到VHD文件的尾部\*/

err = vhd\_write\_footer\_at(&ctx, &ctx.footer, off);

if (err)

goto out;

err = 0;

out:

vhd\_close(&ctx);

if (err && !ctx.is\_block)

unlink(name);

return err;

}

# 参考资料

storage layout系列之VHD结构详解：<http://www.educity.cn/os/1148317.html>

Virtual Hard Disk Format Spec\_10\_18\_16.doc