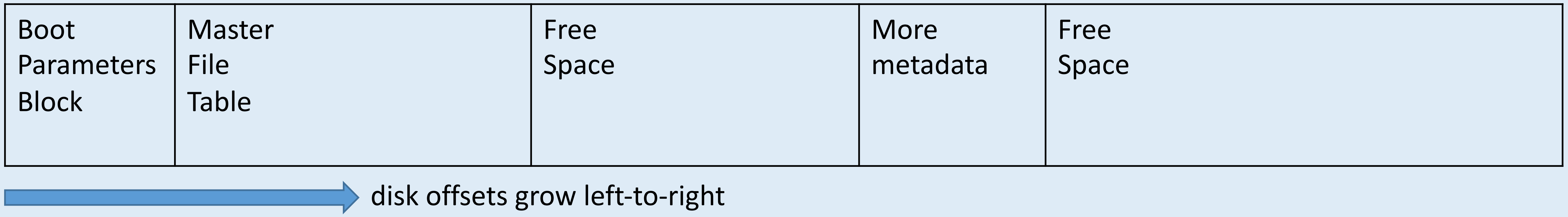
The basics of file systems

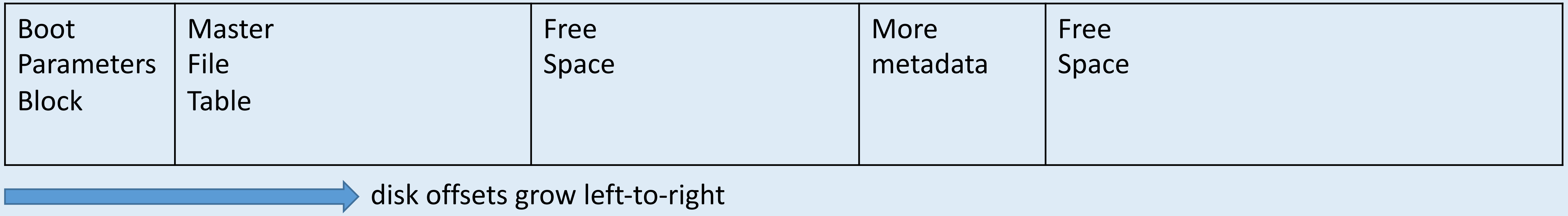
Neapolis University Pafos

JET BRAINS

The layout of NTFS in general



The layout of NTFS in general



Boot Parameters Block is similar to the superblock of ext2. It contains some basic info about the FS and a pointer to the MFT.

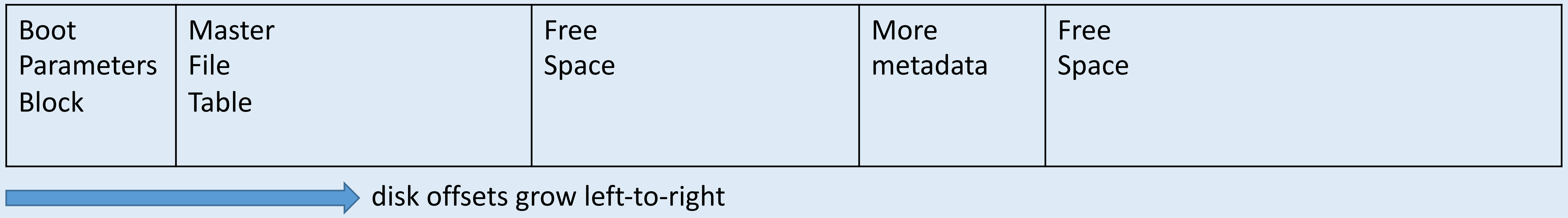
MFT is the “inode table”. Every MFT entry describes one file*.

“More metadata” is a disk area that stores

- MFT Mirror (the copy of first 4 MFT entry),
- The journal.

** Sometimes a file may take multiple entries in the MFT (we will see it later).*

The layout of NTFS in general



Boot Parameters Block is similar to the superblock of ext2. It contains some basic info about the FS and a pointer to the MFT.

MFT is the “inode table”. Every MFT entry describes one file*.

“More metadata” is a disk area that stores

- MFT Mirror (the copy of first 4 MFT entry),
- The journal.

Is it possible to grow the MFT if we grow a partition that contains NTFS?

Where is the block bitmap?

** Sometimes a file may take multiple entries in the MFT (we will see it later).*

Boot Parameters Block (src/linux/fs/ntfs3/ntfs.h)

```
struct NTFS_BOOT {
    u8 jump_code[3];
    u8 system_id[8]; // “NTFS    "

    u8 bytes_per_sector[2];

    u8 sectors_per_clusters;
    u8 unused1[7];
    u8 media_type;
    u8 unused2[2];
    __le16 sct_per_track;
    __le16 heads;
    __le32 hidden_sectors;
    u8 unused3[4];
    u8 bios_drive_num;
    u8 unused4;
    u8 signature_ex;
    u8 unused5;
    __le64 sectors_per_volume;
    __le64 mft_clst;
    __le64 mft2_clst;
    s8 mft_record_size;
    u8 unused6[3];
    s8 indx_record_size;
    u8 unused7[3];
    __le64 serial_num;
    __le32 check_sum;

    u8 boot_code[0x200 - 0x50 - 2 - 4];
    u8 boot_magic[2];
};
```

BPB contains:

- Magic numbers that confirm that a partition really contains a NTFS-formatted file system,

Boot Parameters Block (src/linux/fs/ntfs3/ntfs.h)

```
struct NTFS_BOOT {
    u8 jump_code[3];
    u8 system_id[8]; // “NTFS    "

    u8 bytes_per_sector[2];

    u8 sectors_per_clusters;
    u8 unused1[7];
    u8 media_type;
    u8 unused2[2];
    __le16 sct_per_track;
    __le16 heads;
    __le32 hidden_sectors;
    u8 unused3[4];
    u8 bios_drive_num;
    u8 unused4;
    u8 signature_ex;
    u8 unused5;
    __le64 sectors_per_volume;
    __le64 mft_clst;
    __le64 mft2_clst;
    s8 mft_record_size;
    u8 unused6[3];
    s8 indx_record_size;
    u8 unused7[3];
    __le64 serial_num;
    __le32 check_sum;

    u8 boot_code[0x200 - 0x50 - 2 - 4];
    u8 boot_magic[2];
};
```

BPB contains:

- Magic numbers that confirm that a partition really contains a NTFS-formatted file system,
- Information about the size of the most important data structures in this file system,

Boot Parameters Block (src/linux/fs/ntfs3/ntfs.h)

```
struct NTFS_BOOT {
    u8 jump_code[3];
    u8 system_id[8]; // “NTFS    "

    u8 bytes_per_sector[2];

    u8 sectors_per_clusters;
    u8 unused1[7];
    u8 media_type;
    u8 unused2[2];
    __le16 sct_per_track;
    __le16 heads;
    __le32 hidden_sectors;
    u8 unused3[4];
    u8 bios_drive_num;
    u8 unused4;
    u8 signature_ex;
    u8 unused5;
    __le64 sectors_per_volume;
    __le64 mft_clst;
    __le64 mft2_clst;
    s8 mft_record_size;
    u8 unused6[3];
    s8 indx_record_size;
    u8 unused7[3];
    __le64 serial_num;
    __le32 check_sum;

    u8 boot_code[0x200 - 0x50 - 2 - 4];
    u8 boot_magic[2];
};
```

BPB contains:

- Magic numbers that confirm that a partition really contains a NTFS-formatted file system,
- Information about the size of the most important data structures in this file system,
- Pointers to the MFT and MFT Mirror.

Boot Parameters Block (src/linux/fs/ntfs3/ntfs.h)

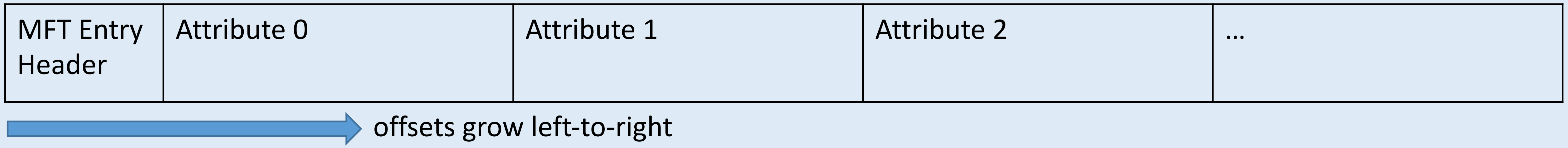
```
struct NTFS_BOOT {
    u8 jump_code[3];
    u8 system_id[8]; // “NTFS    "

    u8 bytes_per_sector[2];

    u8 sectors_per_clusters;
    u8 unused1[7];
    u8 media_type;
    u8 unused2[2];
    __le16 sct_per_track;
    __le16 heads;
    __le32 hidden_sectors;
    u8 unused3[4];
    u8 bios_drive_num;
    u8 unused4;
    u8 signature_ex;
    u8 unused5;
    __le64 sectors_per_volume;
    __le64 mft_clst;
    __le64 mft2_clst;
    s8 mft_record_size;
    u8 unused6[3];
    s8 indx_record_size;
    u8 unused7[3];
    __le64 serial_num;
    __le32 check_sum;

    u8 boot_code[0x200 - 0x50 - 2 - 4];
    u8 boot_magic[2];
};
```

How does the OS know

- the version of NTFS,
- the size of the MFT

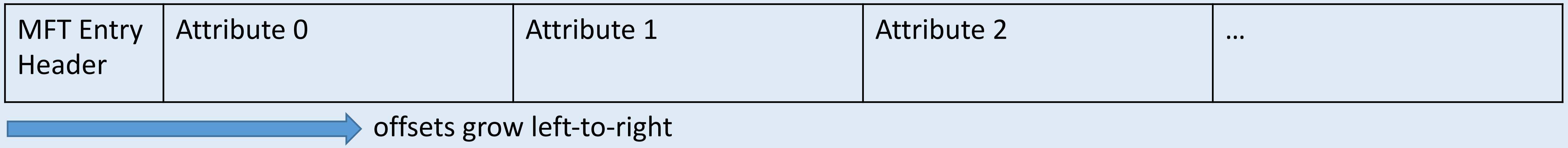
?

The structure of MFT Entries



A file in NTFS can be viewed as a record in a table of a database. File attributes can be viewed as table columns.

The structure of MFT Entries



A file in NTFS can be viewed as a record in a table of a database. File attributes can be viewed as table columns.

A regular file has the following attributes:

- $STANDARD_INFORMATION (creation and modification time, flags like “system”, “compressed”, etc.)
- $FILE_NAME,
- $SECURITY_DESCRIPTOR,
- $DATA.

Атрибуты файлов (src/linux/fs/ntfs3/ntfs.h)

ATTR header	Attribute name	Resident attribute value or a runlist

```
struct ATTRIB {
    enum ATTR_TYPE type;
    __le32 size;
    u8 non_res;
    u8 name_len;
    __le16 name_off;
    __le16 flags;
    __le16 id;

    union {
        struct ATTR_RESIDENT res;
        struct ATTR_NONRESIDENT nres;
    };
};
```

```
struct ATTR_RESIDENT {
    __le32 data_size;
    __le16 data_off;
    u8 flags;
    u8 res;
};
```

Short values are stored directly in MFT entries.

$STANDARD_INFORMATION and file names are always resident.

```
struct ATTR_NONRESIDENT {
    __le64 svcn;
    __le64 evcn;
    __le16 run_off;
    u8 c_unit;
    u8 res1[5];
    __le64 alloc_size;
    __le64 data_size;
    __le64 valid_size;
    __le64 total_size;
};
```

Runlist is a list of extents that store the attribute value.

Runlists

Runlist maps VCNs, Virtual Cluster Numbers (cluster numbers within an attribute value), to LCNs, Logical Cluster Number (cluster numbers on the disk). Compare this to logical and physical offsets of extents in ext4.

Runlist of an uncompressed file is a list of variable-length records that have the following format (size of elements are shown in units of 4 bits):

F	L	length	delta LCN	F	L	length	delta LCN	…
1	1	2*L	2*F	1	1	2*L	2*F	

The VCN of runlist[i] is calculated implicitly. Adjacent items of a runlist describe adjacent extents of an attribute value.

The LCN of runlist[i] is calculated as the sum of the last LCN in runlist[i-1] and the delta LCN from runlist[i].

Runlists

Runlist maps VCNs, Virtual Cluster Numbers (cluster numbers within an attribute value), to LCNs, Logical Cluster Number (cluster numbers on the disk). Compare this to logical and physical offsets of extents in ext4.

Runlist of an uncompressed file is a list of variable-length records that have the following format (size of elements are shown in units of 4 bits):

F	L	length	delta LCN	F	L	length	delta LCN	…
1	1	2*L	2*F	1	1	2*L	2*F	

The VCN of runlist[i] is calculated implicitly. Adjacent items of a runlist describe adjacent extents of an attribute value.

The LCN of runlist[i] is calculated as the sum of the last LCN in runlist[i-1] and the delta LCN from runlist[i].

A range of VCNs may be mapped to no LCNs. This way NTFS represents sparse files.

	F, L	Length	Delta LCN	
runlist[0]	4, 1	128	2^31 – 123	The extent begins in the cluster 2^31 - 123
runlist[1]	0, 1	64	(empty)	This is a hole
runlist[2]	2, 1	128	2^15	The extent begins in the cluster 2^31 - 123 + 2^15

Атрибуты файлов (src/linux/fs/ntfs3/ntfs.h)

ATTR header	Attribute name	Resident attribute value or a runlist

```
struct ATTRIB {
    enum ATTR_TYPE type;
    __le32 size;
    u8 non_res;
    u8 name_len;
    __le16 name_off;
    __le16 flags;
    __le16 id;

    union {
        struct ATTR_RESIDENT res;
        struct ATTR_NONRESIDENT nres;
    };
};
```

```
struct ATTR_RESIDENT {
    __le32 data_size;
    __le16 data_off;
    u8 flags;
    u8 res;
};
```

```
struct ATTR_NONRESIDENT {
    __le64 svcn;
    __le64 evcn;
    __le16 run_off;
    u8 c_unit;
    u8 res1[5];
    __le64 alloc_size;
    __le64 data_size;
    __le64 valid_size;
    __le64 total_size;
};
```

If an attribute value is very big and fragmented, it needs many entries in the runlist. How do we handle long runlists that can’t fit into an MFT entry?

Attributes of regular files

- $STANDARD_INFORMATION,
- $FILE_NAME,
- $OBJECT_ID,
- $SECURITY_DESCRITPOR,
- $DATA,
- $EA_INFORMATION,
- $EA.

Attributes of regular files

- **$STANDARD_INFORMATION,**
- $FILE_NAME,
- $OBJECT_ID,
- $SECURITY_DESCRITPOR,
- $DATA,
- $EA_INFORMATION,
- $EA.

```
struct ATTR_STD_INFO5 {
    __le64 cr_time;          // 0x00: File creation file.
    __le64 m_time;           // 0x08: File modification time.
    __le64 c_time;           // 0x10: Last time any attribute was modified.
    __le64 a_time;           // 0x18: File last access time.
    enum FILE_ATTRIBUTE fa; // 0x20: Standard DOS attributes & more.
    __le32 max_ver_num;      // 0x24: Maximum Number of Versions.
    __le32 ver_num;          // 0x28: Version Number.
    __le32 class_id;

    // Win2k and later

    __le32 owner_id;         // 0x30: Owner Id of the user owning the file.
    __le32 security_id;      // 0x34: The Security Id is a key in the
                             // $SII Index and $SDS.
    __le64 quota_charge;     // 0x38:
    __le64 usn;              // 0x40: Last Update Sequence Number of the
                             // file. This is a direct index into the file
                             // $UsnJrnl. If zero, the USN Journal is
                             // disabled.
};
```

Attributes of regular files

- $STANDARD_INFORMATION,
- **$FILE_NAME,**
- $OBJECT_ID,
- $SECURITY_DESCRITPOR,
- $DATA,
- $EA_INFORMATION,
- $EA.

```
struct ATTR_FILE_NAME {
    struct MFT_REF home;     // 0x00: MFT record for directory.
    struct NTFS_DUP_INFO dup;// 0x08:
    u8 name_len;             // 0x40: File name length in words.
    u8 type;                 // 0x41: File name type.
    __le16 name[];           // 0x42: File name.
};
```

Attributes of regular files

- $STANDARD_INFORMATION,
- **$FILE_NAME,**
- $OBJECT_ID,
- $SECURITY_DESCRITPOR,
- $DATA,
- $EA_INFORMATION,
- $EA.

```
struct ATTR_FILE_NAME {
    struct MFT_REF home;    // 0x00: MFT record for directory.
    struct NTFS_DUP_INFO dup;// 0x08:
    u8 name_len;            // 0x40: File name length in words.
    u8 type;                // 0x41: File name type.
    __le16 name[];          // 0x42: File name.
};
```

The file name type selects a namespace where the name is visible:

- FILE_NAME_POSIX,
- FILE_NAME_UNICODE,
- FILE_NAME_DOS.

Files in NTFS have at least 2 names. These are a UNICODE name for Win32 and a DOS name. The DOS name is contracted to 8.3 format (8 bytes for the name, 3 bytes for the extension).

Files may have more names if they are hardlinked to.

Attributes of regular files

- $STANDARD_INFORMATION,
- $FILE_NAME,
- **$OBJECT_ID,**
- $SECURITY_DESCRITPOR,
- $DATA,
- $EA_INFORMATION,
- $EA.

```
struct OBJECT_ID {
    struct GUID ObjId;
    struct GUID BirthVolumeId;
    struct GUID BirthObjectId;
    struct GUID DomainId;
};
```

Unique file IDs were added for the UI. Windows Explorer has file shortcuts. Essentially, these are symlinks. When the target of a shortcut is moved, the shortcuts becomes dangling and can no longer be used. Windows 2000 reimplemented shortcuts to point to unique file IDs instead of their paths.

Attributes of regular files

- $STANDARD_INFORMATION,
- $FILE_NAME,
- $OBJECT_ID,
- $SECURITY_DESCRITPOR,
- **$DATA,**
- $EA_INFORMATION,
- $EA.

This attribute holds the content of the file.

There is an important difference from POSIX file systems. There may be multiple attributes $DATA in a file. The unnamed one holds the file content in the POSIX sense. Named attributes are typically used as extended attributes.

Directories in NTFS

A directory in NTFS is a B-tree which indexes files on their fixed attribute (e.g., Win32 name, DOS name, unique ID, etc.).

A directory is a file that NTFS that has the following attributes:

- $STANDARD_INFORMATION,
- $FILE_NAME,
- $SECURITY_DESCRIPTOR,
- **$INDEX_ROOT,**
- **$INDEX_ALLOCATION,**
- **$BITMAP.**

These 3 attributes are NTFS's way to store a B-tree.

If a directory indexes files by their `$FILE_NAME`, then it works like a directory in ext2.

Files with many attributes

Suppose an attribute value is comprised of many extents e.g. the file data is very fragmented or has many holes. Their runlist may be too long to fit into one MFT Entry.

Files with many attributes

Suppose an attribute value is comprised of many extents e.g. the file data is very fragmented or has many holes. Their runlist may be too long to fit into one MFT Entry.

1. A file may be described by multiple MFT Entries.

A file may have an attribute $ATTRIBUTE_LIST (possibly non-resident!). The value of this attribute is an array that contains the following entries:

```
struct ATTR_LIST_ENTRY {
    enum ATTR_TYPE type;    // 0x00: The type of attribute.
    __le16 size;            // 0x04: The size of this record.
    u8 name_len;            // 0x06: The length of attribute name.
    u8 name_off;            // 0x07: The offset to attribute name.
    __le64 vcn;             // 0x08: Starting VCN of this attribute.
    struct MFT_REF ref;     // 0x10: MFT record number with attribute.
    __le16 id;              // 0x18: struct ATTRIB ID.
    __le16 name[];
};
```

Elements in this array list attributes of a file and point to MFT Entries that contain those attributes.

Files with many attributes

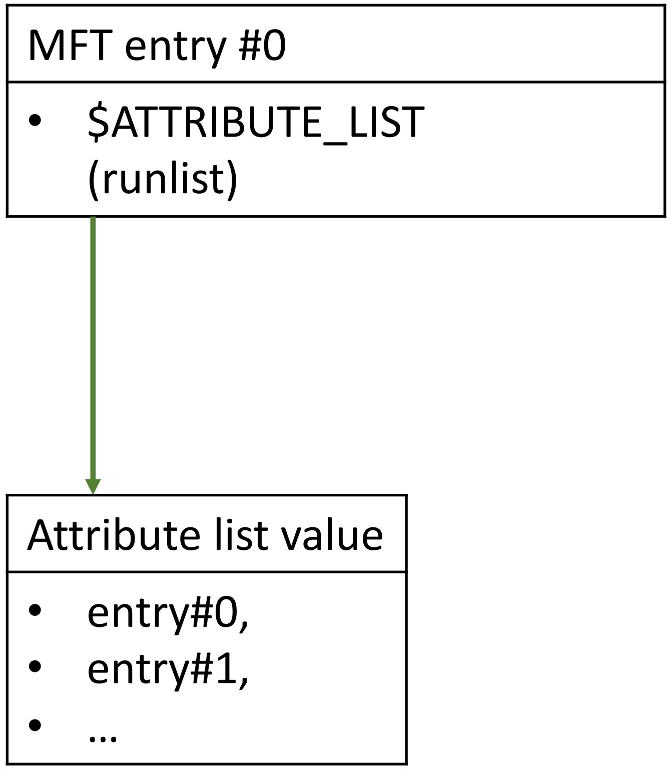
How do we implement attributes that have many extents (they may be highly fragmented or have holes)? Their runlists may be too big to fit into one MFT Entry.

1. A file may be described by multiple MFT Entries.
2. A file may have repeated attributes. In this case the values of repeated attributes are concatenated to turn them into one attribute.

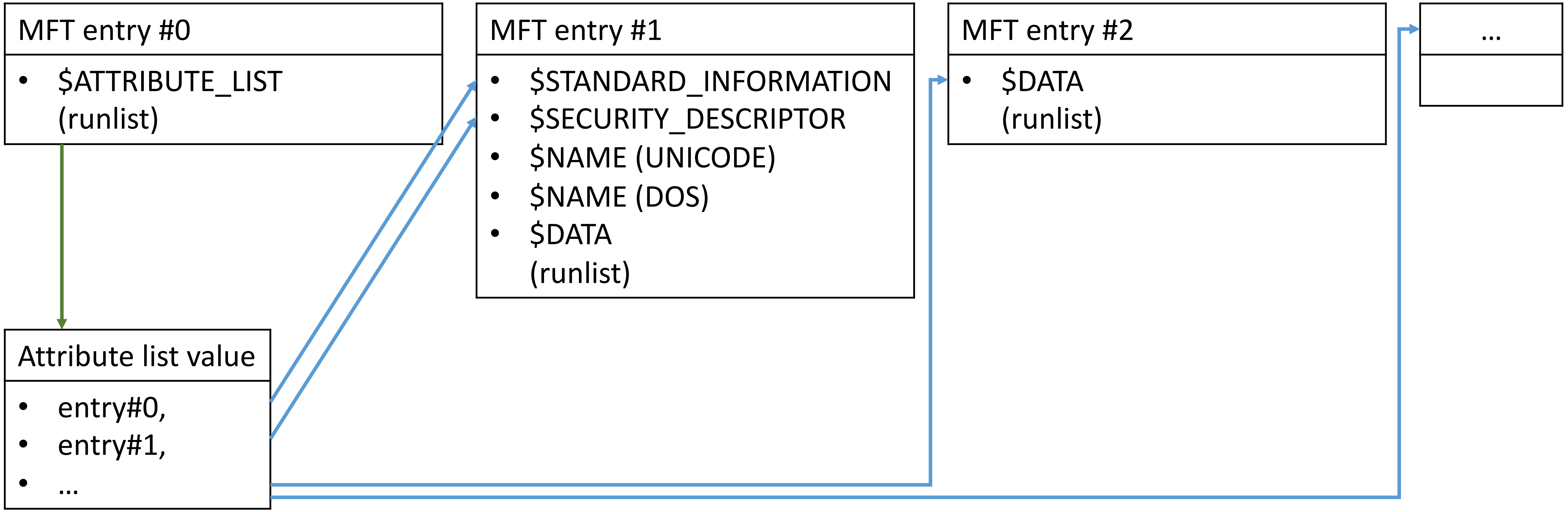
```
struct ATTR_NONRESIDENT {
    __le64 svcn;         // 0x10: Starting VCN of this segment.
    __le64 evcn;         // 0x18: End VCN of this segment.
    __le16 run_off;
    u8 c_unit;
    u8 res1[5];
    __le64 alloc_size;
    __le64 data_size;
    __le64 valid_size;
    __le64 total_size;
};
```

The runlist of a non-resident attribute counts VCNs from SVCN (Start VCN) of the attribute. This way NTFS can join runlists of repeated attributes.

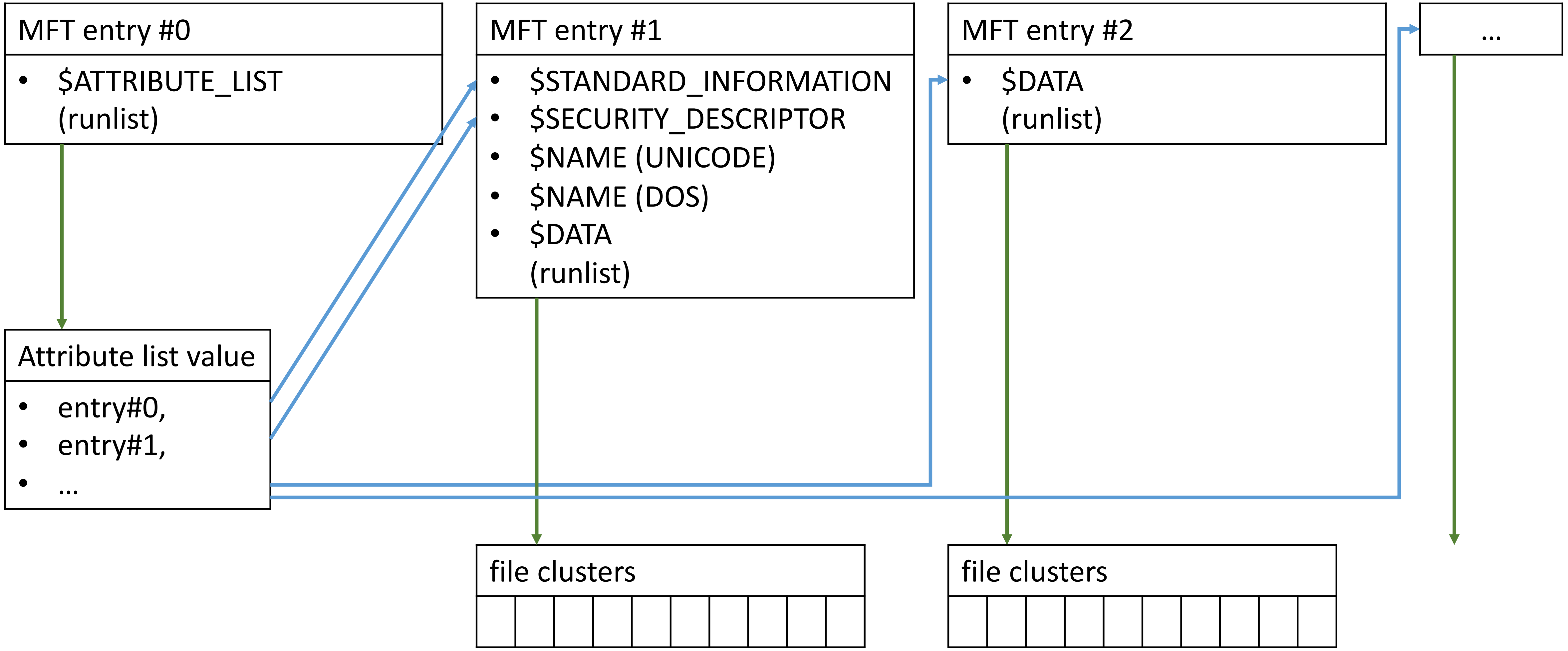
Files with many attributes



Files with many attributes



Files with many attributes



Files in NTFS: differences from POSIX

1. Multiple namespace for space names.
2. Names in the Unicode namespace are **case-preserving**. This means the names are not changed when files are created or listed, but when searching for files, names are compared in a case-insensitive manner.
3. Directories may index files on arbitrary attributes, not necessarily their names.
4. Files may have multiple data streams (compare this to ”file data” + “extended attributes” in ext4):
   - OpenFile(“file.txt”) opens the unnamed $DATA,
   - OpenFile(“file.txt:alt”) opens $DATA named “alt”.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.`
- `$Bitmap,`
- `$Boot,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- `$UpCase,`
- `$Extend,`
- `$ObjId,`
- `$Reparse,`
- `$UsnJournal.`

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- **`$MFT`**,
- `$MFTMirr`,
- `$LogFile`,
- `$Volume`,
- `$AttrDef`,
- `.`
- `$Bitmap`,
- `$Boot`,
- `$BadClus`,
- `$Quota`,
- `$Secure`,
- `$UpCase`,
- `$Extend`,
- `$ObjId`,
- `$Reparse`,
- `$UsnJournal`.

Describes an area of the volume that is occupied by MFT itself. This way it possible to

- Increase the size of the MFT when the FS grows.
- Decrease the size of the MFT if there is not enough space for user files.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- $MFT,
- $MFTMirr,
- $LogFile,
- **$Volume**,
- $AttrDef,
- .
- $Bitmap,
- $Boot,
- $BadClus,
- $Quota,
- $Secure,
- $UpCase,
- $Extend,
- $ObjId,
- $Reparse,
- $UsnJournal.

This file has attributes $VOLUME_NAME and $VOLUME_INFORMATION. They hold the name of the NTFS volume and the NTFS version infromation.

```
struct VOLUME_INFO {
    __le64 res1;    // 0x00
    u8 major_ver;   // 0x08: NTFS major version number
    u8 minor_ver;   // 0x09: NTFS minor version number
    __le16 flags;   // 0x0A: Volume flags, see VOLUME_FLAG_XXX
};

#define VOLUME_FLAG_DIRTY           cpu_to_le16(0x0001)
#define VOLUME_FLAG_RESIZE_LOG_FILE cpu_to_le16(0x0002)
...
```

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

This is the root directory of the NTFS volume.

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.           <---`
- `$Bitmap,`
- `$Boot,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- `$UpCase,`
- `$Extend,`
- `$ObjId,`
- `$Reparse,`
- `$UsnJournal.`

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT`,
- `$MFTMirr`,
- `$LogFile`,
- `$Volume`,
- `$AttrDef`,
- `.`
- **`$Bitmap`**,
- `$Boot`,
- `$BadClus`,
- `$Quota`,
- `$Secure`,
- `$UpCase`,
- `$Extend`,
- `$ObjId`,
- `$Reparse`,
- `$UsnJournal`.

This file stores the bitmask of used and free clusters. Cf. the block bitmap in ext2.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.`
- `$Bitmap,`
- **`$Boot`**`,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- `$UpCase,`
- `$Extend,`
- `$ObjId,`
- `$Reparse,`
- `$UsnJournal.`

This file has $DATA that points to a cluster with the BPB.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.`
- `$Bitmap,`
- `$Boot,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- **`$UpCase,`**
- `$Extend,`
- `$ObjId,`
- `$Reparse,`
- `$UsnJournal.`

Recall that NTFS is a **case-preserving** file system. When looking files up, name comparisons must be case-insensitive.

This is a huge problem because the rule for transforming upper-case letters to lower-case letters does not only depend on the language and the country. Different versions of Unicode have different case folding rules. This is why NTFS is forced to store the case folding rules that were in effect when a volume was created. The rules are stored in this file.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.`
- `$Bitmap,`
- `$Boot,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- `$UpCase,`
- `$Extend,`
- **`$ObjId,`**
- `$Reparse,`
- `$UsnJournal.`

This is a directory that indexes files by their object ID.

System files of NTFS

The first 16 files in a NTFS volume are system files that store the metadata of NTFS:

- `$MFT,`
- `$MFTMirr,`
- `$LogFile,`
- `$Volume,`
- `$AttrDef,`
- `.`
- `$Bitmap,`
- `$Boot,`
- `$BadClus,`
- `$Quota,`
- `$Secure,`
- `$UpCase,`
- `$Extend,`
- `$ObjId,`
- **`$Reparse`**`,`
- `$UsnJournal.`

This files stores the list of mount points (reparse points in the Windows parlance). Unlike Linux and BSD, mount points are persisted on the disk.

There are multiple types of reparse points:

- Junction point (similar to mounts or bind-mounts, >= win2k),
- Symbolic links (>= vista),
- OneDrive.

Extra reading

- https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf
- /src/linux/fs/ntfs3/*

To do at home

Learn the API of ntfs-3g and write a program that

- Walks a NTFS volume and produces a list of all files and directories,
- For each file finds clusters that contain its data and constructs a reverse map that associates disk clusters to files that use them.

1. Create a NTFS volume,
2. Export it over iSCSI with `tgtd`,
3. Mount this volume as read-only from a Windows VM,
4. Read some files in this volume in the Windows VM,
5. Record the iSCSI session with `tcpdump`,
6. Use the reverse map from the previous exercise to deduce the list of files accessed by the VM.