Unit 11 – The NTFS File System

Unit 11.1 – Introduction

According to some sources the “NT” in “Windows-NT” stood for “**new technologies**”. With Windows NT came **NTFS**, the **new technologies file system**. It represented a great departure from its predecessors, with many advantages and new features. Here are some:

- per file ownership and access permissions
- multiple names for each directory or file
- journaling (no need for a file system check if not shut down gracefully)
- alternate data streams (more than one set of clusters for a given file name)
- gobs of metadata
- ability to store small files without allocating any clusters
- data runs/extents (saves lots of space for contiguous files)
- directory B-trees (for very efficient searching)
- support for sparse files (relatively few clusters actually stored)

With the new features came new complexity. Compared to FAT32, NTFS is very complicated. But, with the proper tools it is manageable. Microsoft has not yet published all the details of NTFS, so much of this information is speculative. Brian Carrier's book *File System Forensic Analysis* (ISBN: 0-321-26817-2) is an excellent source.

Unit 11.2 – Common Features of File Systems

Before we get into the details of the NTFS file system let's remind ourselves of some of the features all file systems have.

All file systems have...

1. A group of blocks, usually 4 kB (4096 bytes) in size (but other sizes are possible). These blocks are used to store new directories and files as needed.

2. A scheme to keep track of which block are used and which are unused.

3. An initial sector that describes where to find the root directory and how big the various tables that keep track of things are.

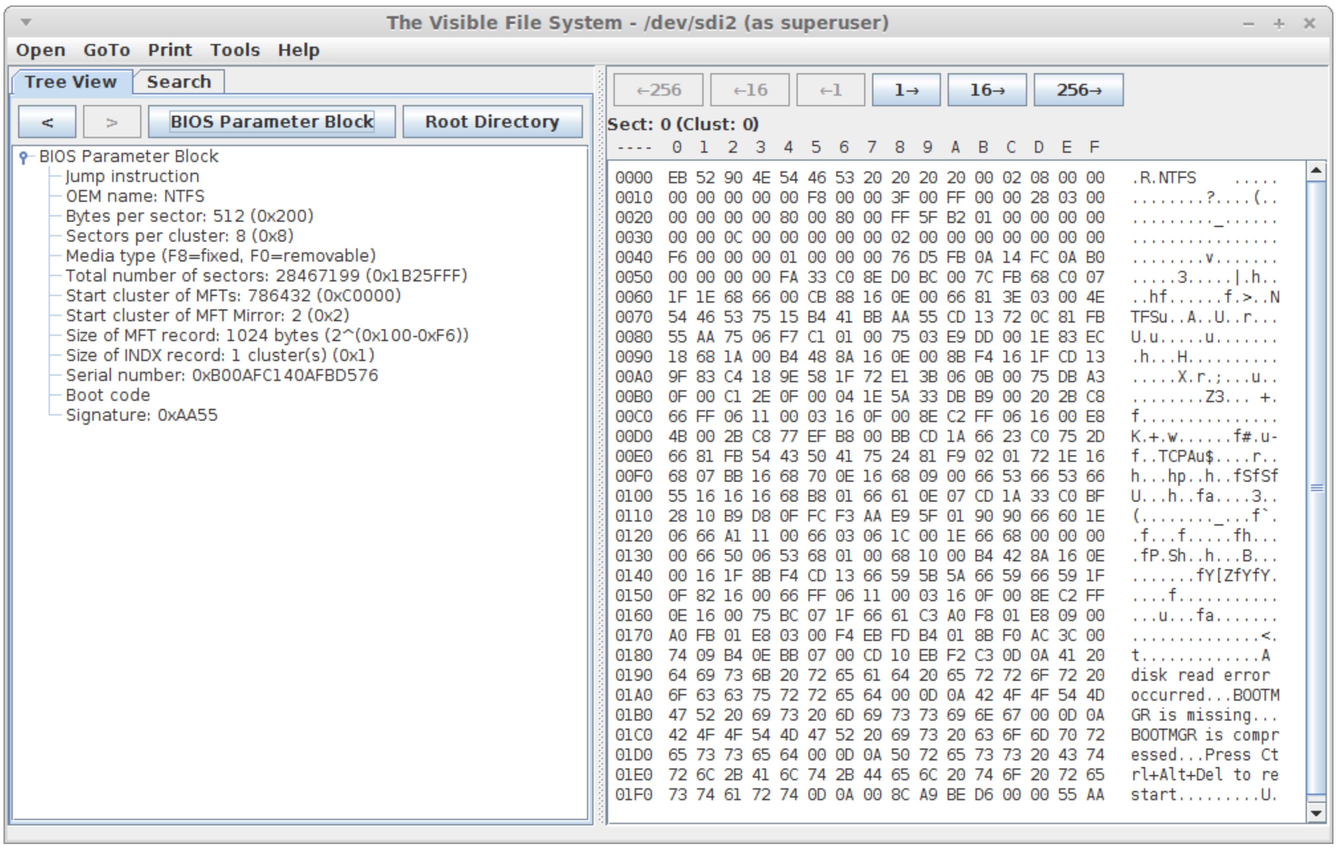
4. A directory structure that starts with the root directory. The root directory can store files and subdirectories within it.

5. A way to record which blocks belong to a particular file or directory.

Unit 11.3 – The BIOS Parameter Block (BPB)

The first sector of an NTFS partition contains the Bios Parameter Block (BPB). It gives information such as the number of sectors per cluster, the start of the first set of MFT entries, the size of MFT entries and INDX clusters as well as a serial number for the volume. It may also contain bootloader code.

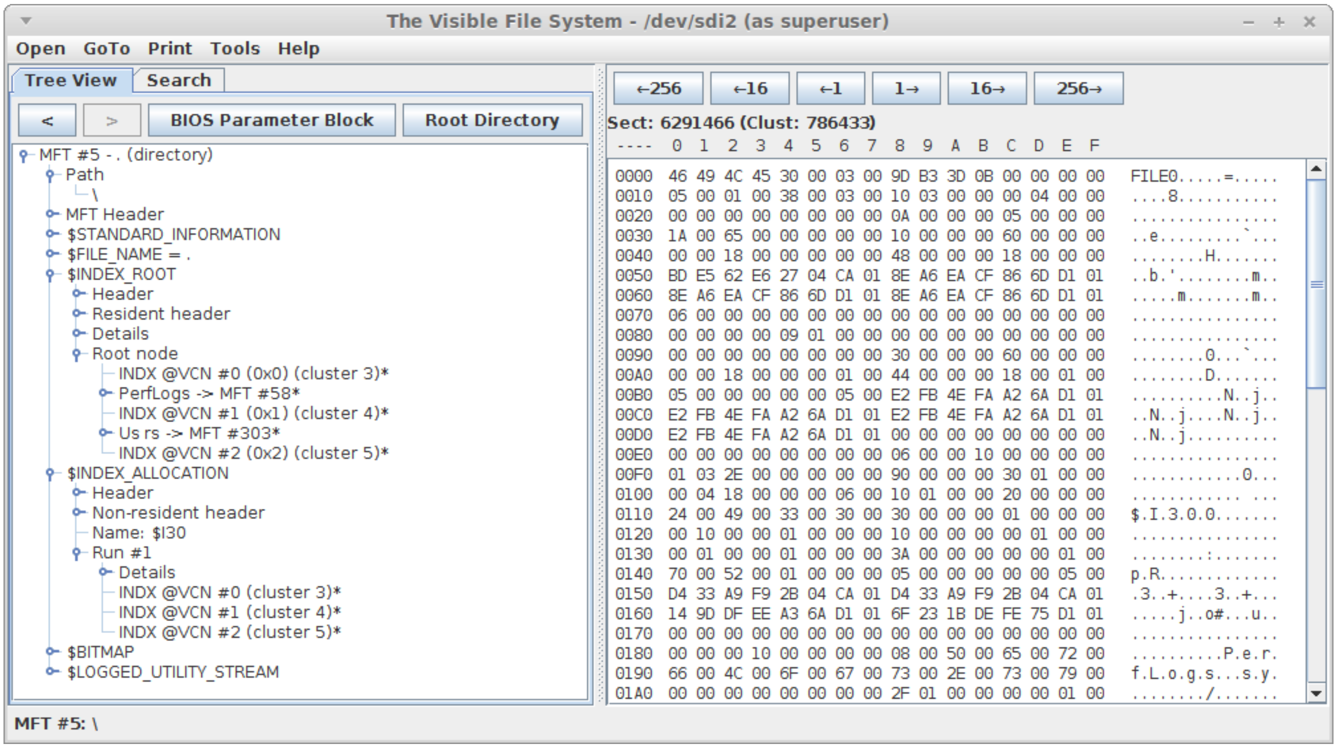
Below is a screenshot of the BPB for an NTFS partition.



Unit 11.4 – Master File Table (MFT) Entries

The master file table is made up of MFT entries throughout the partition (not necessarily all in one place). Every directory and file will have one or more MFT entries pointing to it. The MFT entry(s) contains the cluster allocation for the directory or file (which clusters contain the data for that directory or file) and lots of metadata.

The root directory is at MFT #5. Below is a screenshot from the VisibleFS program. It shows the MFT entry for the root directory.



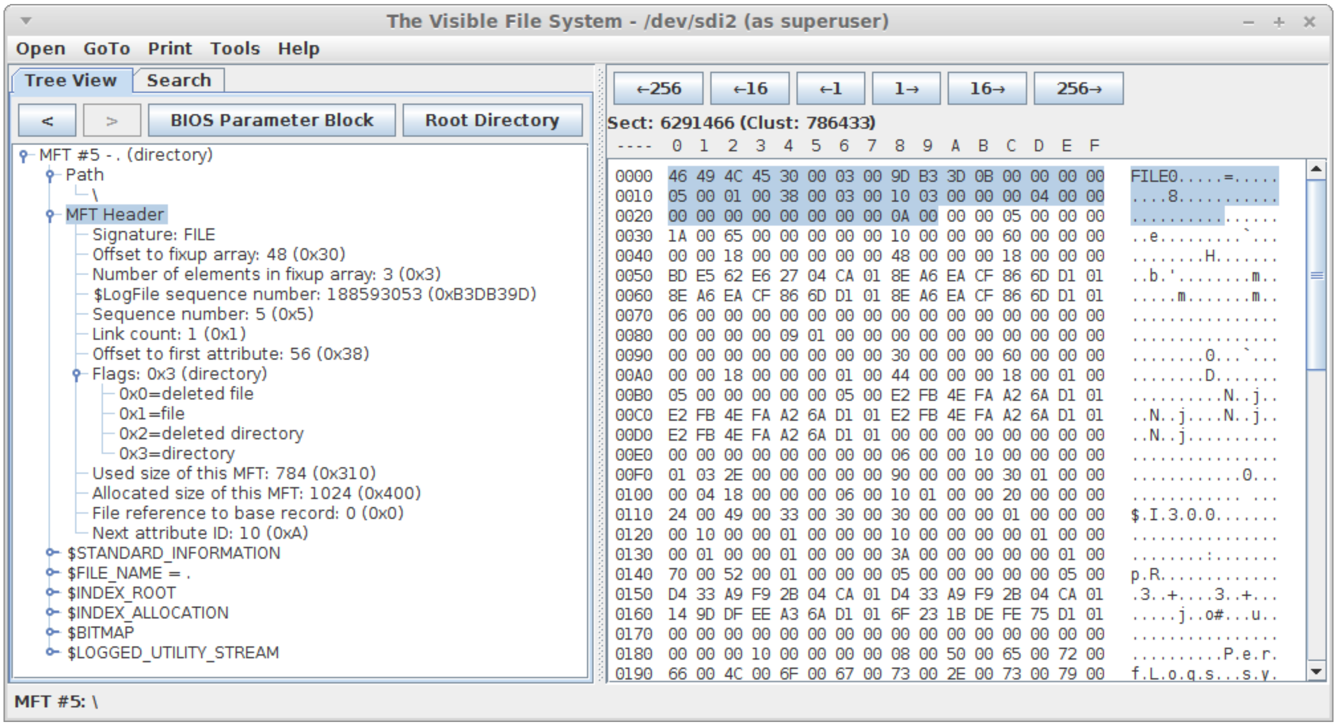
Notice that it is organized into "**attributes**" that start with a $. **$STANDARD_INFORMATION** and **$FILE_NAME** are always present, but the others depend on what the MFT entry points to. Often there is more than one $FILE_NAME attribute, one for the DOS name and one for the long filename, and for hard links (more on this later).

Also note the **$INDEX_ALLOCATION** attribute. It lists the clusters that belong to the root directory. They are stored in the form of a "run" that starts with cluster 3 and goes for a total of 3 clusters. If a directory is fragmented then it will have more than 1 run.

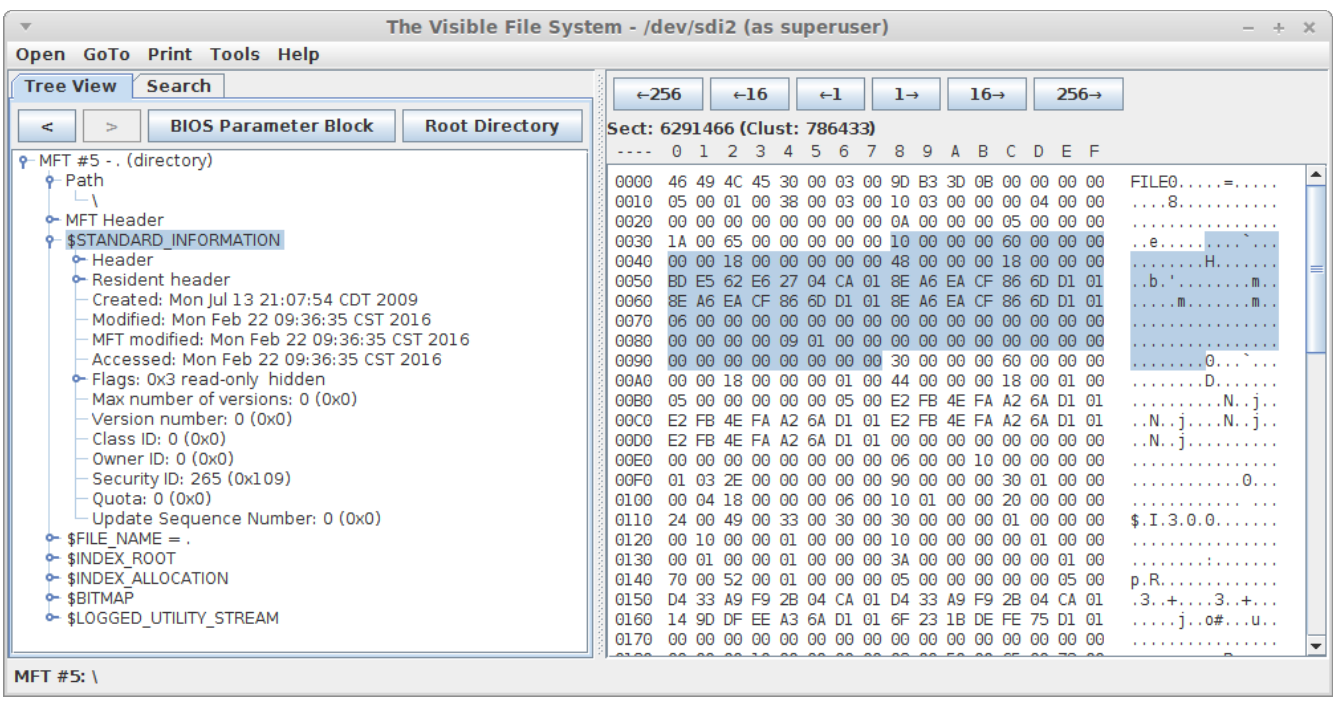
The **$INDEX_ROOT** attribute gives the root node of a B-tree that describes the directories and files in the root directory. We will talk about B-trees later in this unit.

Let's look at the header and various attributes individually. We will skip a lot of the details at this point as they are not relevant to understanding how NTFS works. Some of the details are offsets and sizes of things so that the OS knows where one attribute ends and the next one starts.

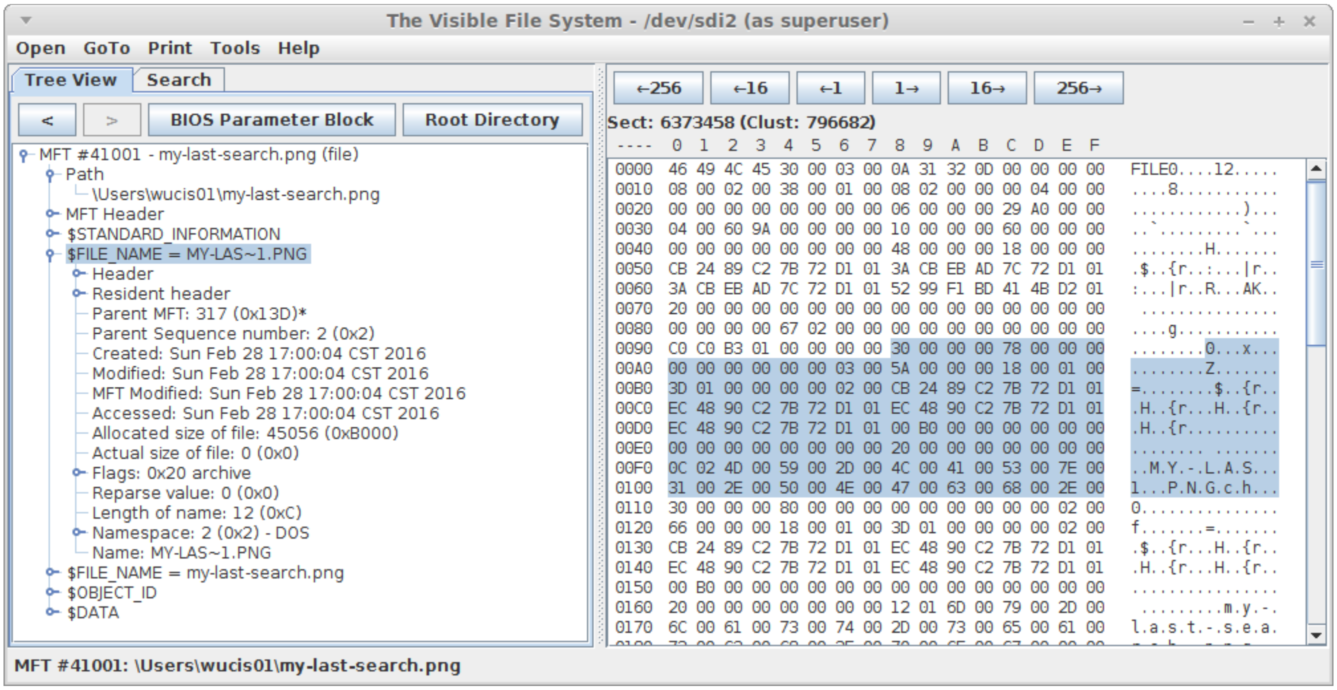
Below is a screenshot showing the MTF header. Note that all MFT entries start with the signature "FILE". It may be that MFT entries that are not yet allocated are blank. Note that the flags in the header tell you that this is a directory. Deleted files or directories are evident by looking at this flag.



Next is a screenshot of the $STANDARD_INFORMATION attribute. Note the four timestamps. Next is a flag that tells you that this is marked as a read-only, hidden file (this seems like a mistake since it is the root directory, mistakes like this are common in NTFS).

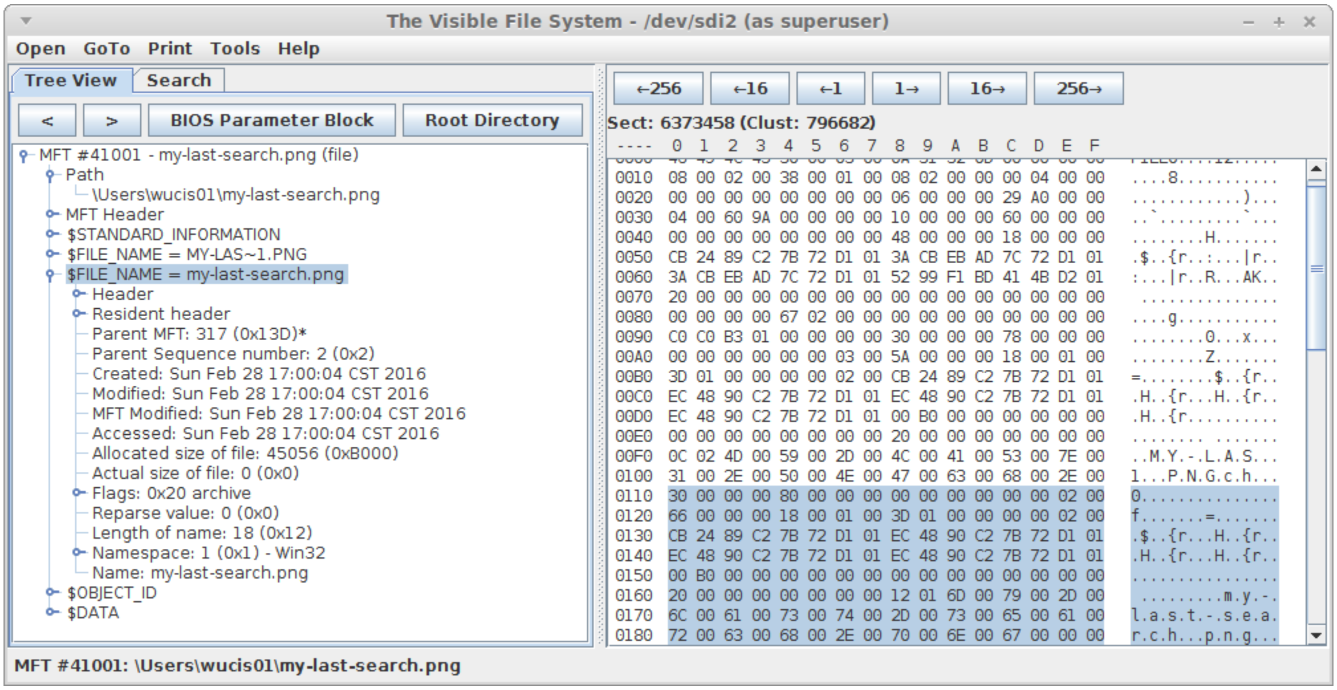


The next screenshot shows the first $FILE_NAME attribute for the file \Users\wucis01\my-last-search.png.

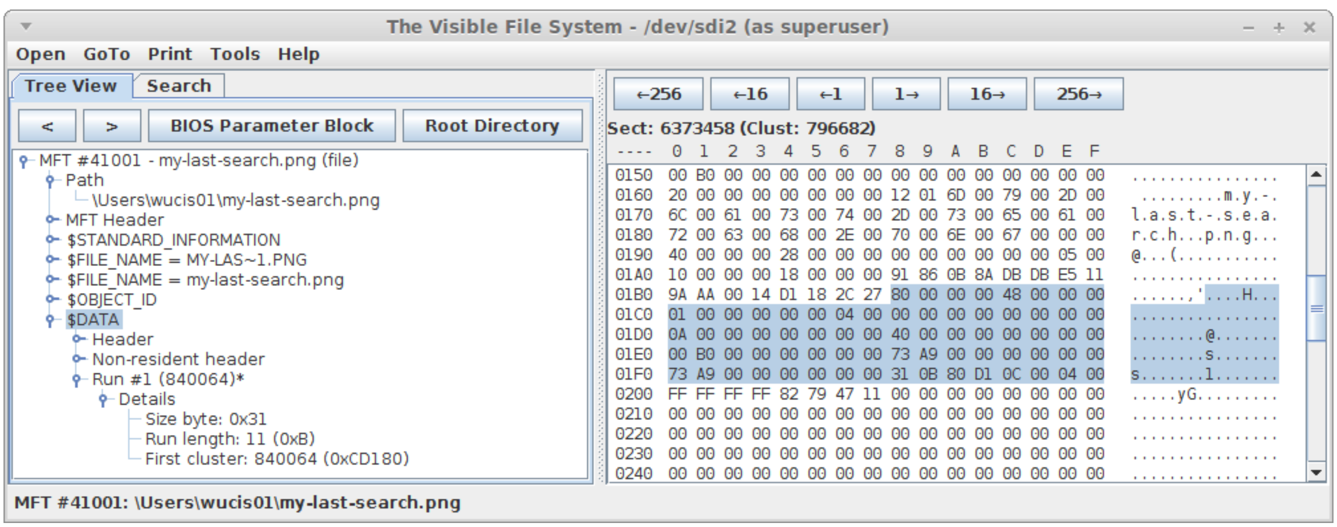


Note the reference to the parent MFT entry which is MFT#317, which is \Users\wucis01. Notice there are four more timestamps after that. For some reason the actual size of this file is listed as 0 bytes (this is not correct but the correct number is given in the directory entry that points to this MFT, more on this later). The allocated size is 45056 bytes (since allocation is done in whole clusters only, in this case 11 clusters). Finally, the DOS name of the file is given in UTF-16 little-endian. Note that the namespace is 2, which indicates it is a DOS name.

Directly below is the second $FILE_NAME attribute. Below is a screenshot of that one.



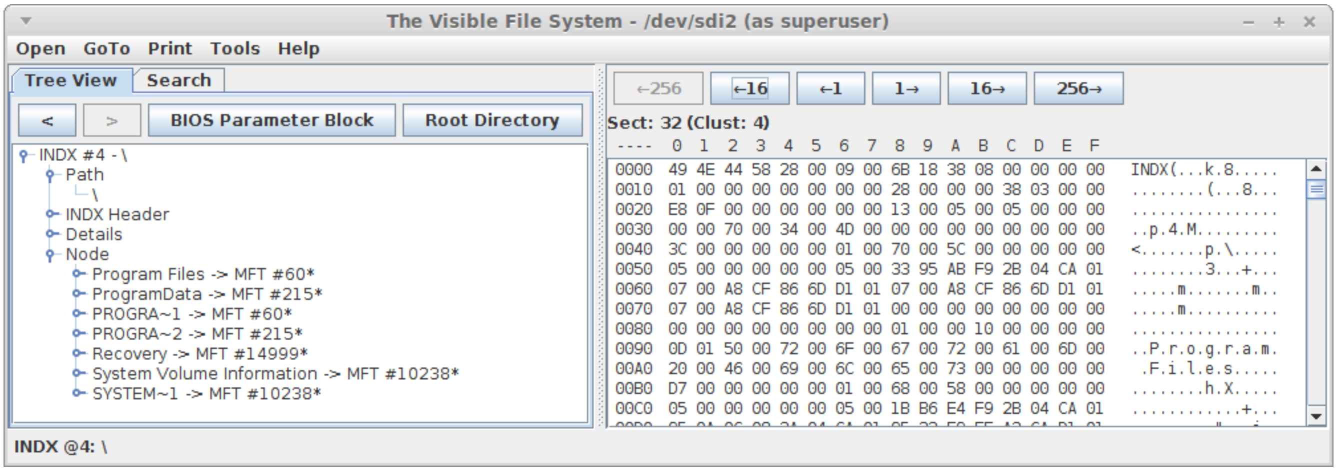
This one gives the long file name in UTF-16 little-endian. Note that it has the same parent MFT, timestamps and other properties except namespace. The next screenshot shows the data runs of this file.



In this case the file has one data run. The runs are given in a cryptic form. The actual data run appears at offset 0x1F8 as the bytes “310B80D10C”. The first byte gives the size of the other items. It says that there is 1 byte in the run length and a 3 bytes in the starting cluster. The run length is “0B” (0x0B=11 in decimal) and the starting cluster is “80D10C”. Putting it in little-endian form 0x0CD180 = 840064 in decimal. If the file was fragmented there would be more than one of these strings of bytes, one for each run.

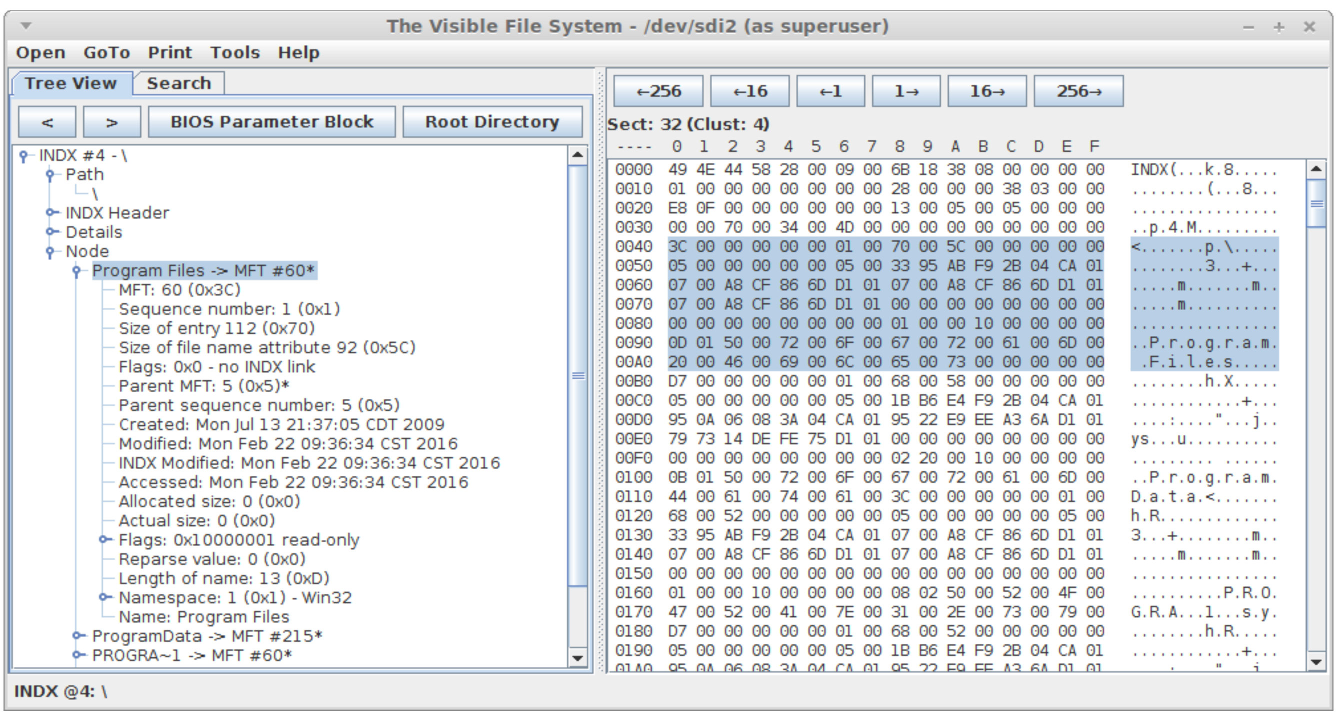
Unit 11.5 – INDX Clusters

When an MFT entry points to a directory, it may also point to one or more INDX clusters. These contain the listing of what is contained within this directory. Below is a screenshot of an INDX cluster.



Note the signature "INDX" at the start of the cluster. Also note that the listing is in alphabetical order. This is only a partial listing though. The other parts are in another part of a B-tree (more on that later).

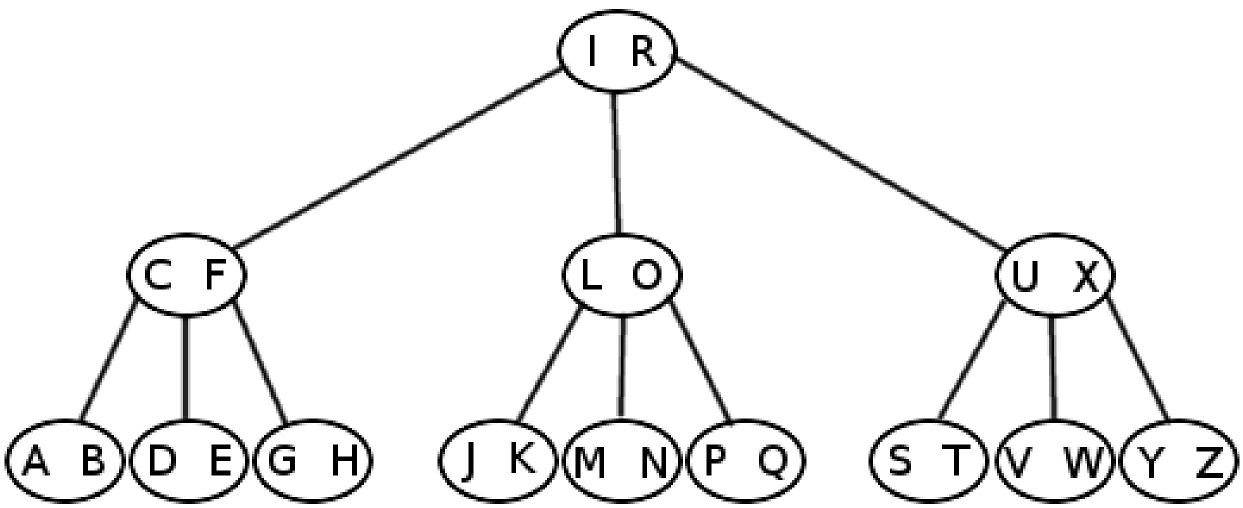
The following screenshot shows the first directory entry expanded.



Note that the first field points to the MFT for this item (#60). There are four timestamps and a size (in this case zero since it a directory). Finally there is the name. Directory entries can actually appear inside of MFT entries as well. We will see that in the next section.

Unit 11.6 – Directory B-trees

The B-tree is a data structure that allows for fast searching. It grows in such a way that it always has the same number of nodes on any path from root (called a self-balancing tree). The following is an illustration of a B-tree using single letters as keys for simplicity.

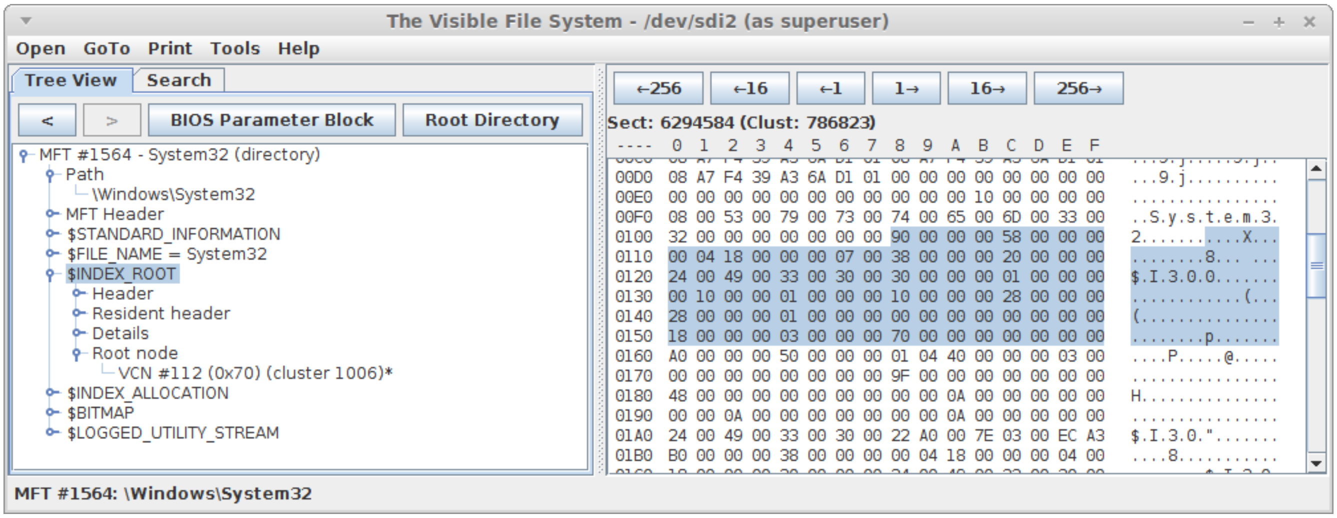


Let's imagine that we're searching for “E”. You start at the root node and compare E with I and R. Since E is before I you go left. You now compare E with C and F. Since E comes between C and F you go straight down. You find E in that node. Any node can be located in this way with a minimum of comparisons. In this B-tree the OS needs to search at most three nodes to find any of the 26 items.

The B-trees in NTFS can have several dozen entries per node instead of just two.

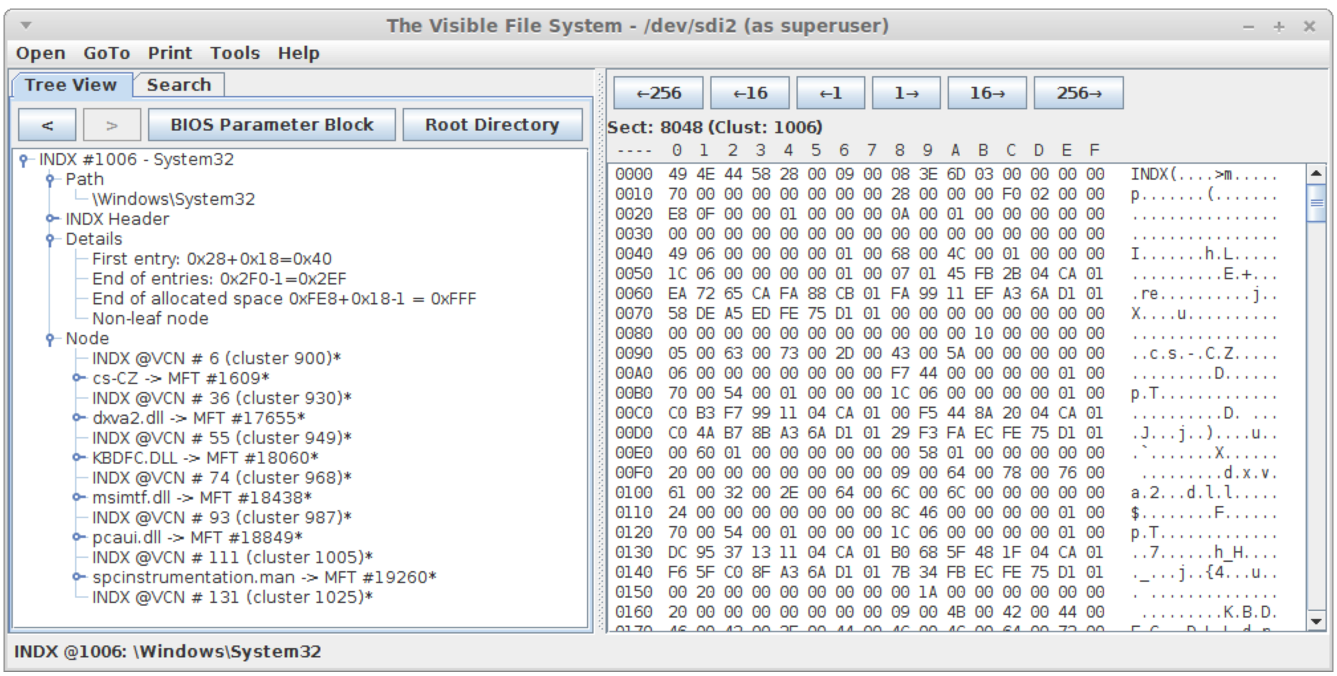
Nodes with no children are called “**leaf nodes**”. The other nodes are called “**nonleaf nodes**”.

Let's follow an NTFS B-tree from the root node to a leaf node. Below is the MFT entry for the \Windows\System32 directory.



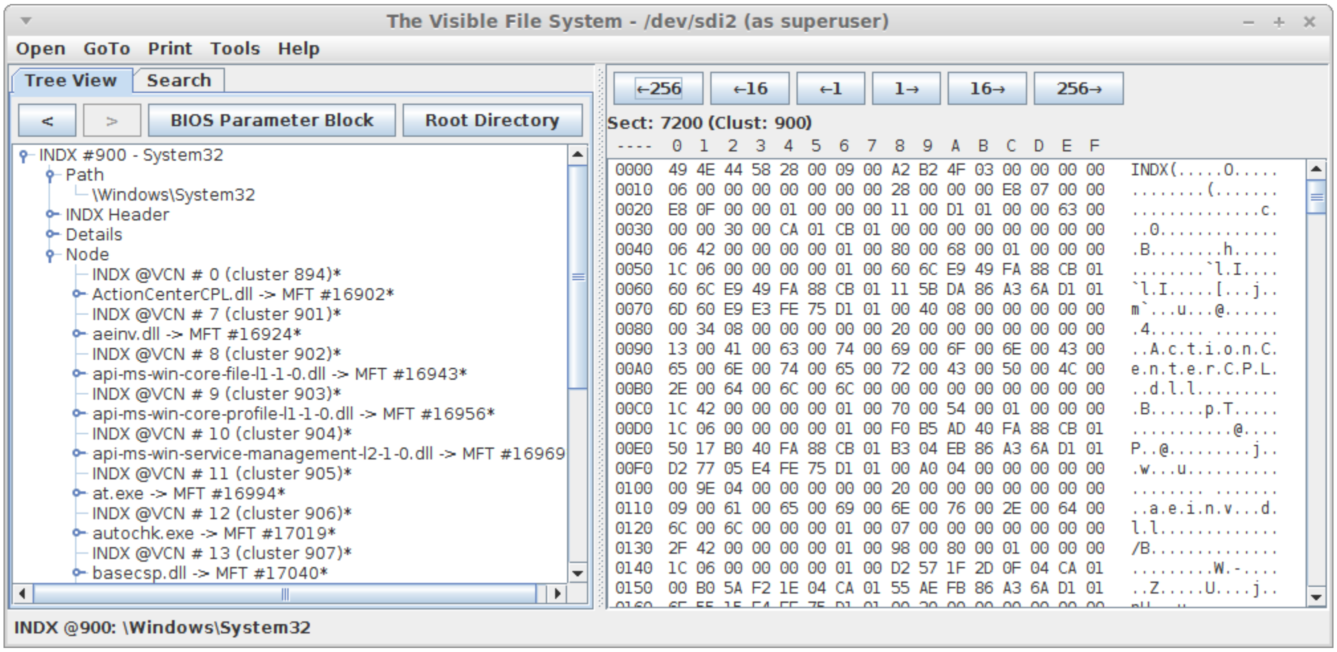
Note that the $INDEX_ROOT attribute only has a reference to **VCN** 112 in it. VCN means **Virtual Cluster Number**. That VCN maps to **logical (physical) cluster** number 1006 in the partition. Sometimes the $INDEX_ROOT attribute will also have directory entries as well. This one just happens to have only a link to an INDX cluster.

Next we follow this link to the INDX cluster at 1006. The screenshot is below.



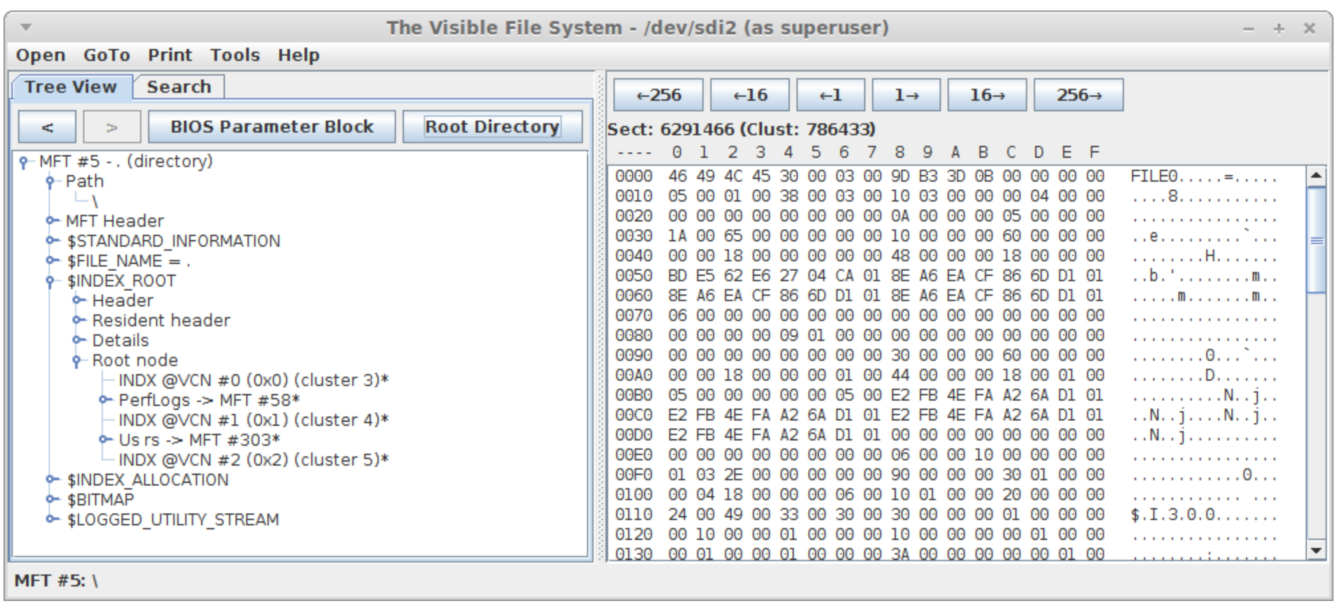
This is a non-leaf node (as indicated in the Details section). Note the links before and after each directory entry. This means that this node contains the entries for “cs-CZ”, “dxva2.dll”, etc. as well as links to other INDX clusters which are the children in the B-tree before, after and between each entry.

Any item with a name before “cs-CZ” will be in the subtree at the node in INDX cluster 900. Any item with a name between “cs-CZ” and “dxva2.dll” will be in the subtree at the node in INDX cluster 930. The INDX cluster 900 is shown below.



Note that this is also a non-leaf node. The links go to child B-tree nodes before, after and between each directory entry.

Here's a screenshot where the root of the B-tree is stored in the $INDEX_ROOT attribute of the MFT entry.



In this case there are two items in the root B-tree node and three links to INDX clusters (before, between and after).

Note that there is always one more link than there are items. When there are no items there is one link. When there is one item there are two links. When there are two items there will be three links, etc.

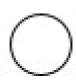
B-trees are used because in previous file systems when directories had a lot of items it just took too long to look through them one-after-another (compare with FAT32 LBA). In your next project you will practice searching through a B-tree.

Unit 11.7 – Building a B-Tree from Scratch

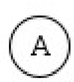
B-Trees are in a class of data structures known as self-balancing trees. They grow in an interesting way, from the bottom upward. I think it is helpful to show how B-Trees are constructed, but I don't expect you to be able to do this on an exam. This is just for those of you that are interested in this kind of thing. If you're a CIS major you will see this again in CM307.

B-Tree nodes have a limit on how many things can fit in them. In NTFS the limit is the size of the $INDEX_ROOT in the MFT and the size of the INDX clusters. In this simple example, I'm going to allow each node to hold two things, but not three. I'm going to add the letters A-K in that order.

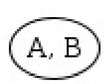
We start with an empty B-Tree:



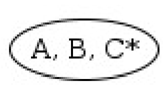
After adding A we get this:



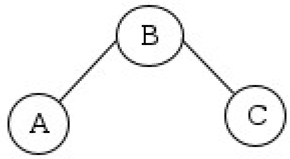
Then we add B:



Next we add C (the * indicates that the node needs to split because it has too many items):

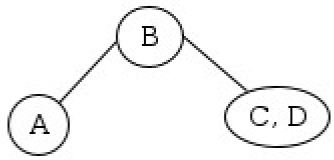


The split operation takes the middle node up to a new, parent node and splits the existing node into two parts:

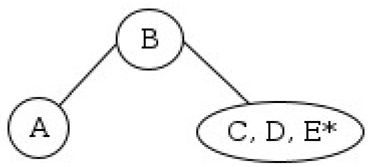


Note the ordering. The search starts at the root node and you go left if the search item < B and right if the search item is > B.

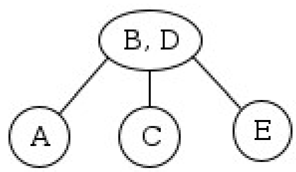
We always add at the lowest level. When we add D we start at root and compare D to B. Since D > B we go right and add there since it is a leaf node (the lowest level).



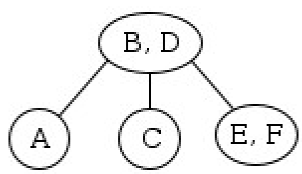
When we add E we force another split:



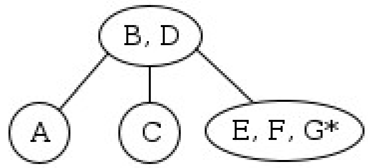
Split causes the D to move up (the middle element):



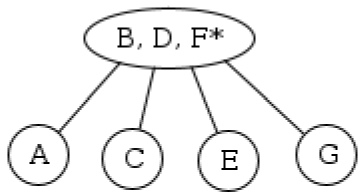
Note that if there are N items in a node, there are N+1 children. Next we add F:



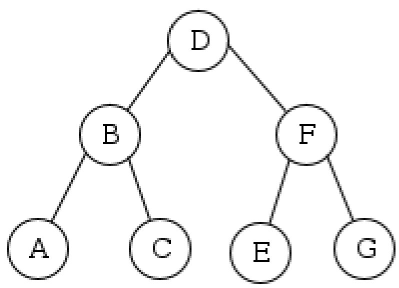
And then G, which causes another split:



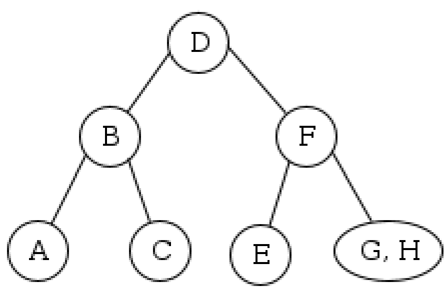
which causes the F to move up:



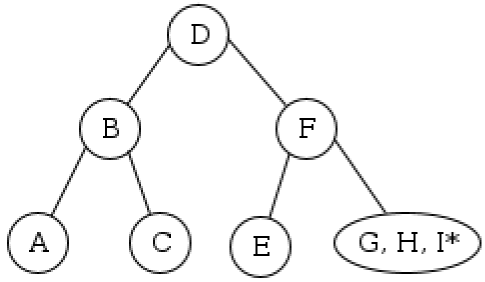
which then causes the D to move up:



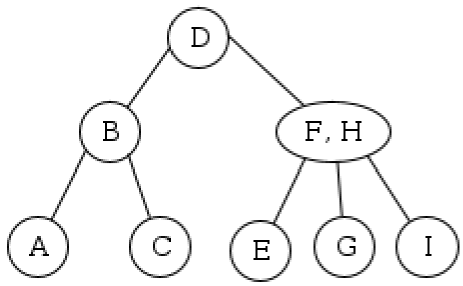
Let’s continue by adding H:



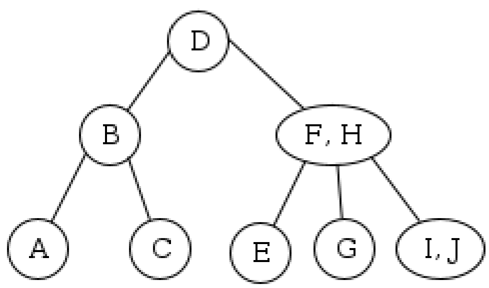
Then add I:



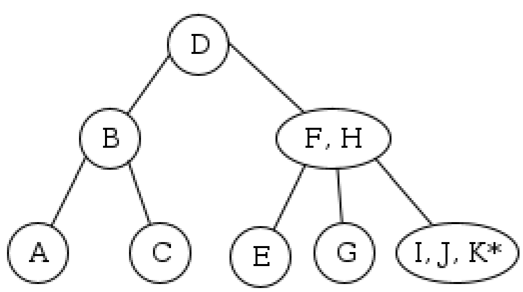
which causes a split where H moves up:



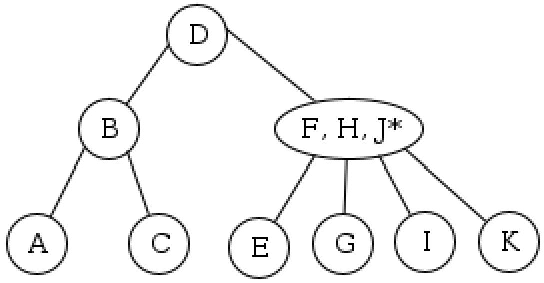
Adding J we get here:



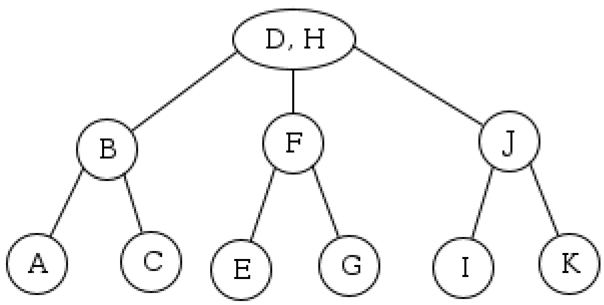
Then adding K we get here:



which causes J to move up:



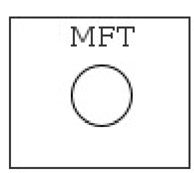
which then causes the H to move up:



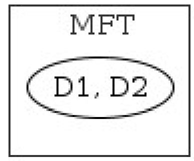
Note that the tree is always balanced.

Unit 11.8 – B-Trees and the MFT

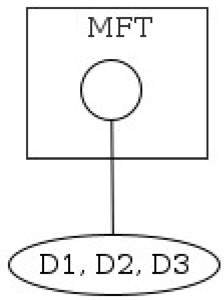
The situation in NTFS is a little more complicated than what was just presented since the root node may be in the MFT or in an INDX block. Here's how the B-Tree grows in NTFS. We start with an empty root node inside the MFT:



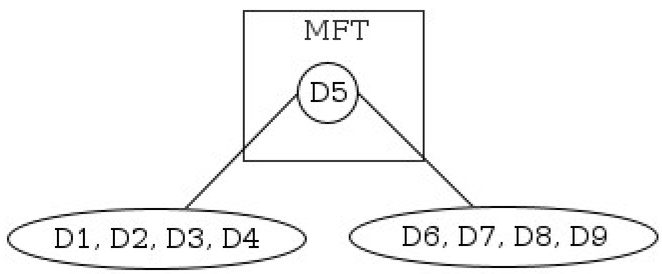
Now, let's add directories D1 and D2 with the assumption that they fit inside the MFT:



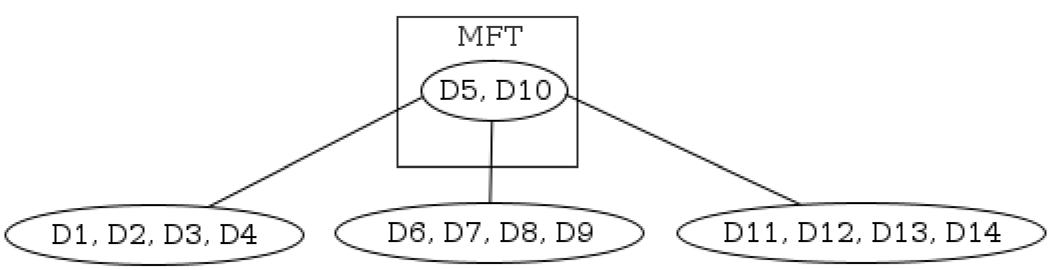
Let's say that adding D3 makes it too big to fit in the MFT. (In real life you can have as many as 10 or so directory entries in an MFT.) In this case an INDX cluster is allocated and all the entries are moved there. The MFT just contains a pointer to the INDX cluster.



After adding many more directories to the INDX cluster it will eventually split into two INDX clusters. The middle item will be moved up into the MFT:

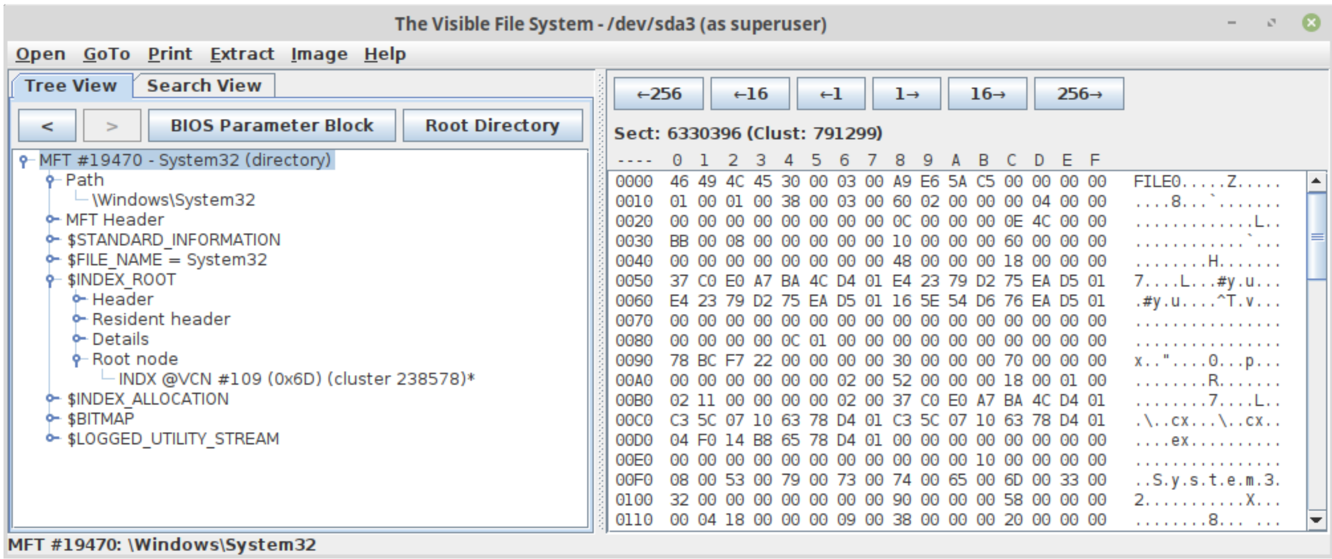


Still later, the rightmost INDX cluster will split again moving its middle element into the MFT:



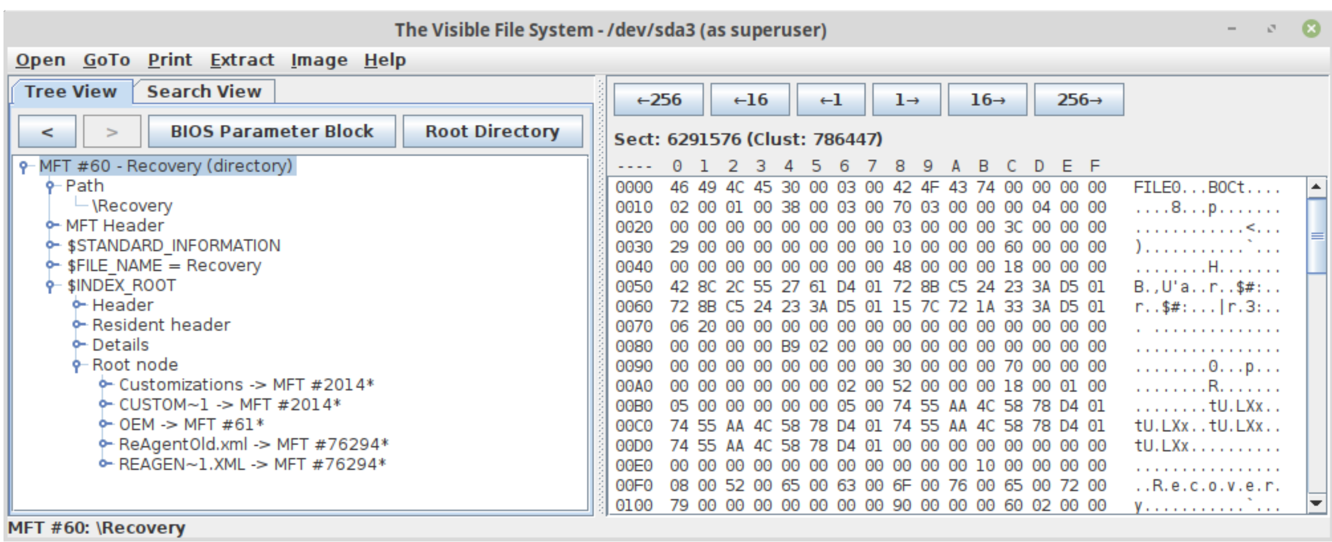
Eventually the MFT will fill up and it will create a new INDX cluster, move its contents into there and store a pointer to it.

Now, let's look at some examples in NTFS. Here's the MFT for C:\Windows\System32:



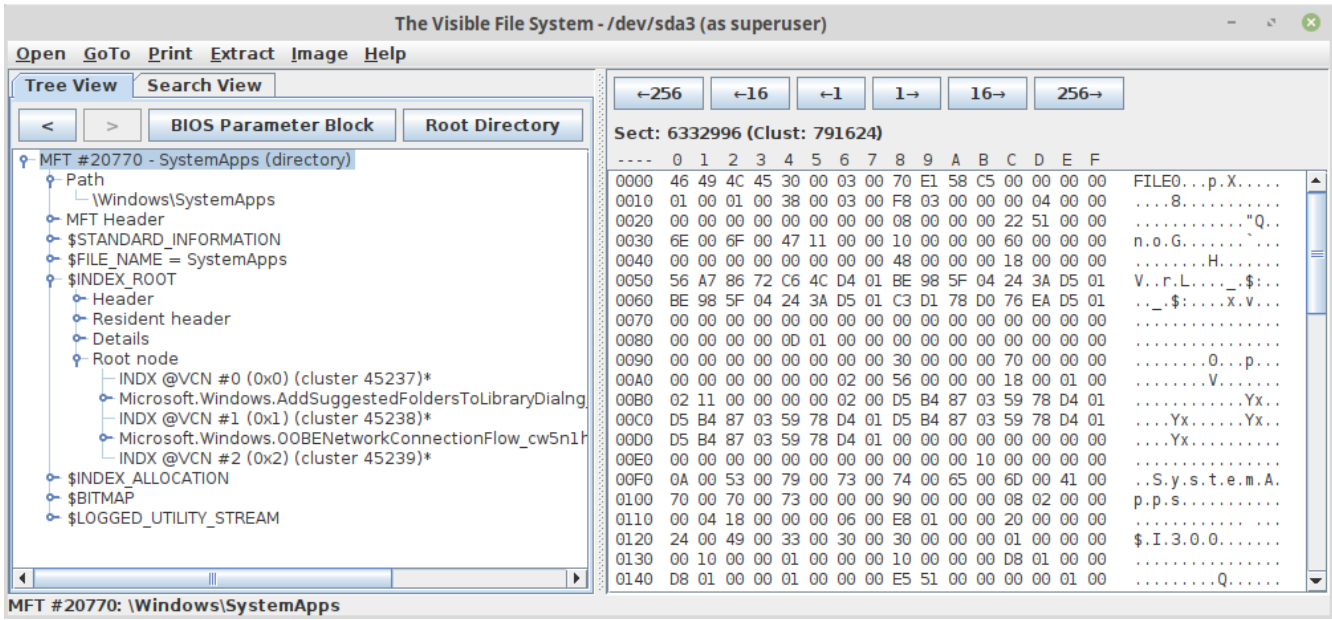
Notice that the MFT contains only a reference to an INDX cluster at 238578.

Here is a screenshot of the directory C:\Recovery:

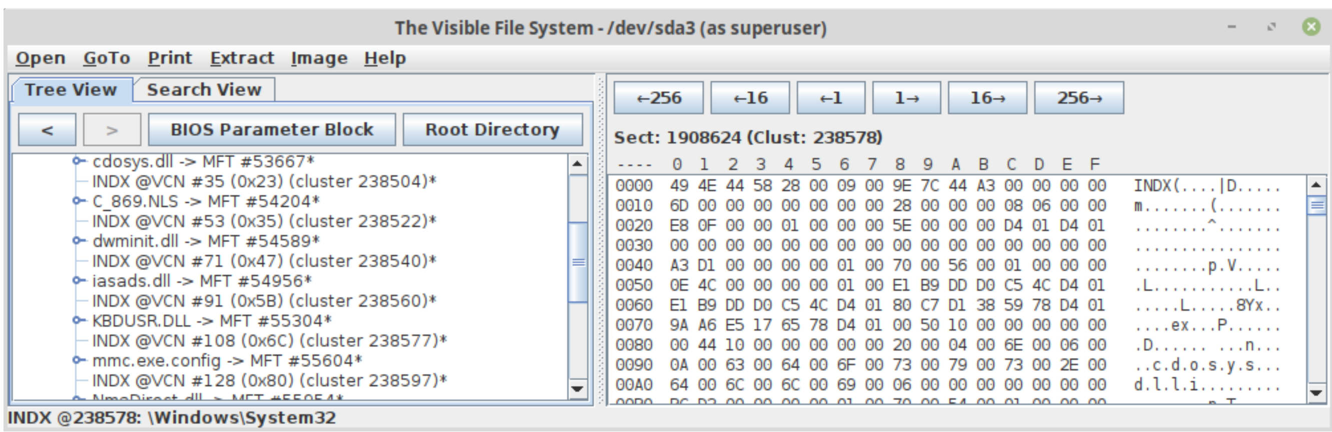


In this case the entire B-Tree node is inside the MFT. There are no pointers to child INDX clusters.

Here's a screenshot of the directory C:\Windows\SystemApps. The root node in the MFT has two items and pointers to three children.



Here is a screenshot of one of the non-leaf INDX clusters for C:\Windows\System32.



If you were searching for a name that was between dwminit.dll and iasads.dll then you would go to INDX 238540 to continue your search.

Unit 11.9 – Efficiency of B-Trees

The efficiency of the search in B-Tree is dependent on the “fanout” of the nodes. Fanout is a measure of how many children each B-Tree node has. I looked at the nodes at the second level of the B-Tree for C:\Windows\System32 and most of the nodes had 17 children. So, for simplicity, let’s make a model where all nodes have 17 children. That means that each node has 16 items in it. Let’s also assume, as in the System32 directory, that the B-Tree has 3 levels.

The first level has 16 items. It has 17 children.

The second level has 17 nodes, each with 16 items in each node. That makes 17*16 = 272 items in the second level.

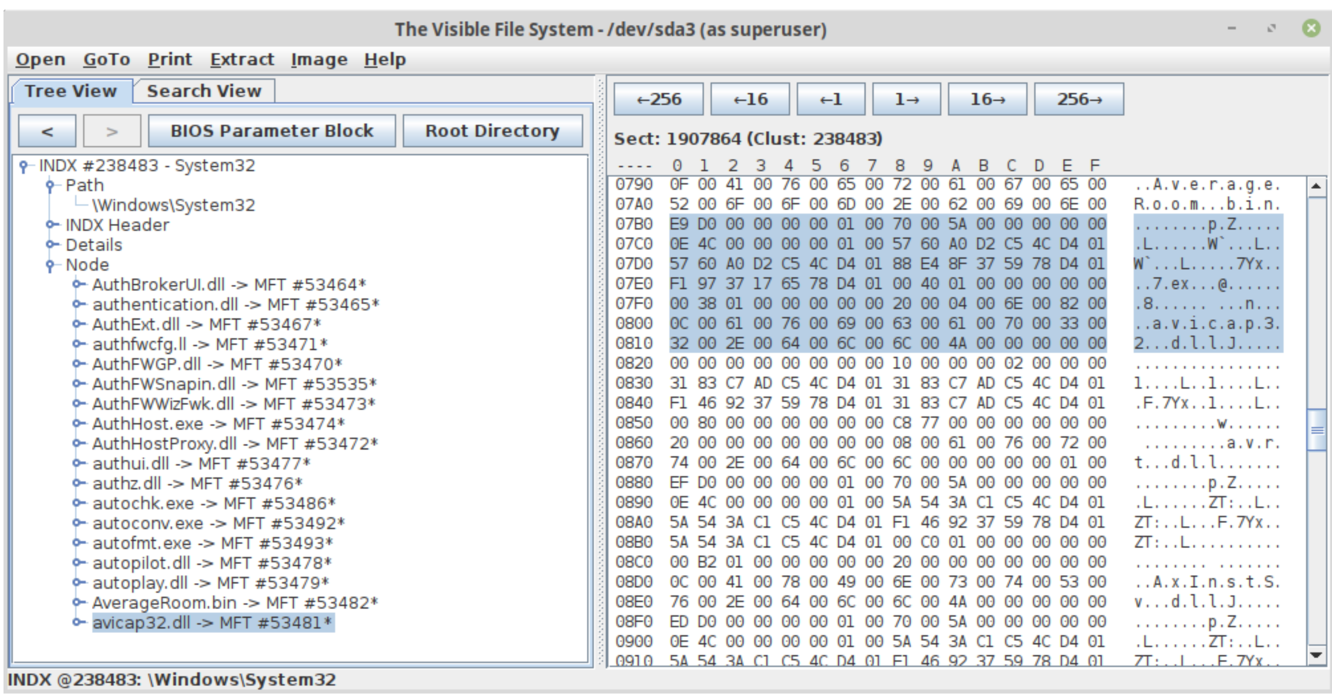
The third level has 17*17 nodes, each with 16 items in each node. That makes 4624 items in the third level.

Adding these up we get 16 + 272 + 4624 = 4912 items in the tree. To search the entire tree we need to look through at most three nodes. That means that we really only actually need to look at 3*16 = 48 items. That's about 1% of the total number of nodes in the tree. If the tree had four levels the percentage would be even smaller.

Unit 11.10 – Forensic Data in Split Nodes

When a node splits half of the data is copied to a new node. In the existing node, the pointer to the end of the items is moved over, but the data is not zeroed out after the pointer. Therefore the existing node contains a snapshot of what was there before the split. That data will remain there until the space is used by data added after the split.

Here's a screenshot of the VisibleFS program showing the abandoned data after a node split:



On the left and right the last directory entry is highlighted. After that you can see the file names "avrt.dll" and "AxInstSv.dll".

It’s a longshot, but there may be some forensic value in these abandoned directory entries. A file that was deleted long ago may still have a directory entry in the slack space of an INDX cluster. It can provide timestamps and an MFT number. If you’re lucky that MFT may simply be marked as deleted but still contain ownership information and even cluster allocation.

Unit 11.11 – The Big Picture

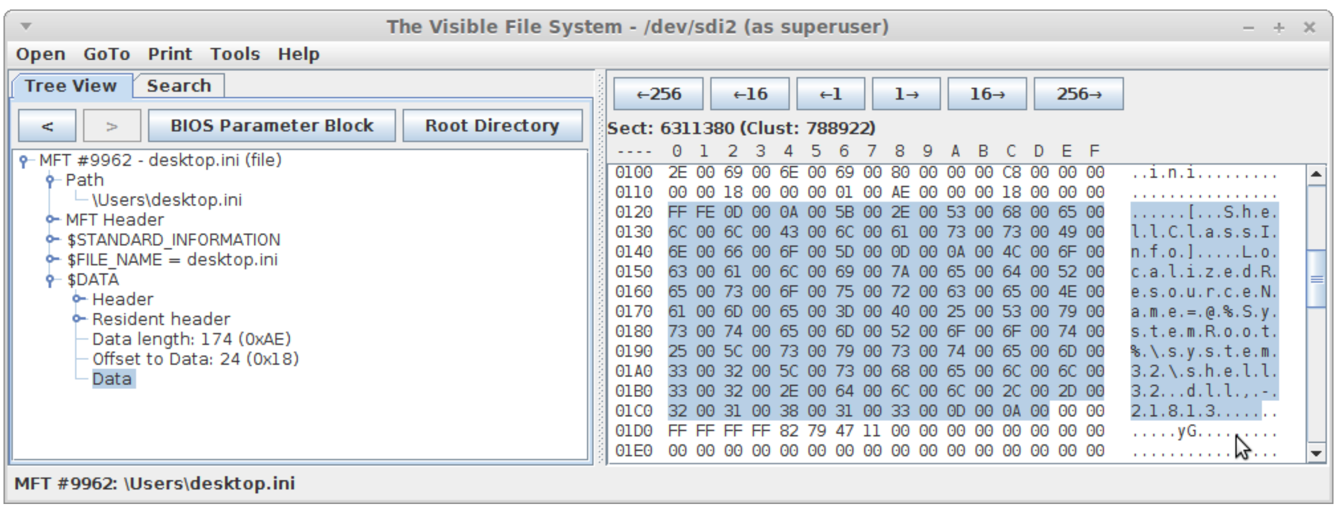
There is much more to NTFS than what we have seen here. There is ownership, access permissions, journaling, reparse points, etc. But what we have seen is the basics of navigating through the file system. It goes like this:

Given a path, you start in the root directory (MFT#5). The $INDEX_ROOT attribute may contain the entire B-tree in a single node, or may point to one or more INDX clusters where the remainder of the B-tree is stored. In any case, you traverse the B-tree looking for the next item in the given path.

Eventually you end up at the MFT entry for the file you are looking for. The MFT entry has the allocation for the file in the $DATA attribute. That's it...

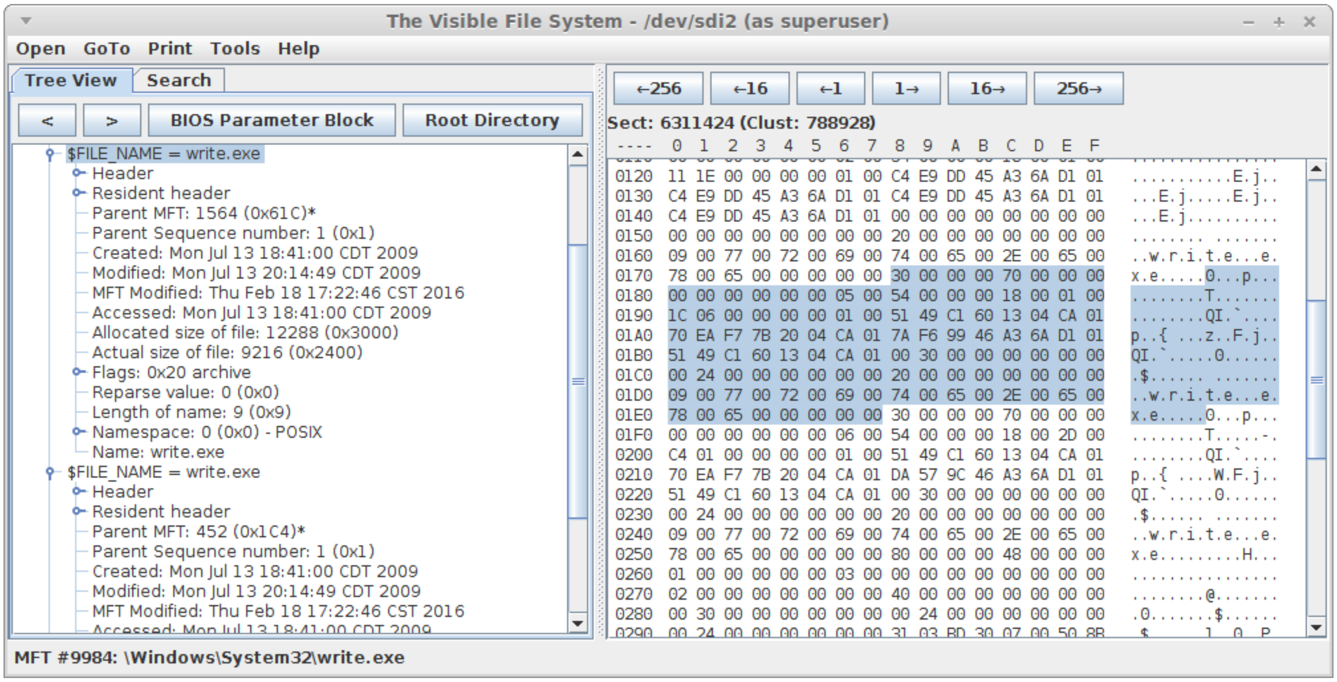
...except that there are many variations on that theme. Here's one.

For small files the file itself is sometimes stored entirely inside the MFT entry itself. The screenshot below shows that case.



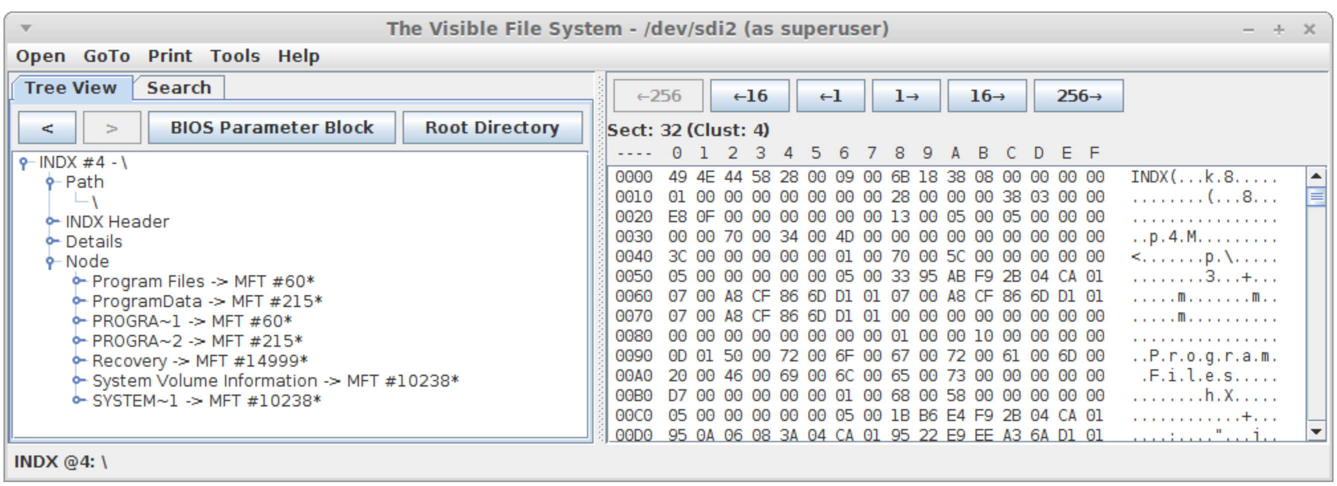
Note the file starts with “FFFE”. That's the byte-order-mark (BOM) for UTF-16LE.

Another variation is hard links to files. A given MFT entry may appear in multiple directory entries in MFT's and INDX clusters. In this case they have extra $FILE_NAME attributes, each one with a different parent MFT. Here's a screenshot of that case.



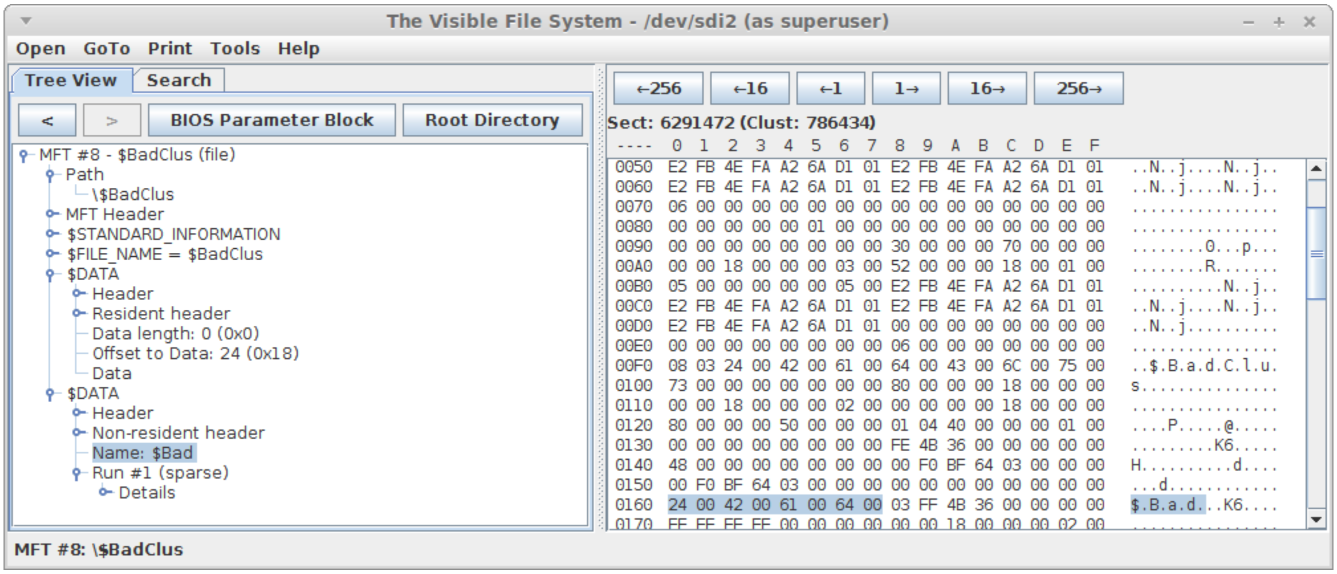
Note that the first $FILE_NAME attribute points to MFT entry #1564 as its parent and the second points to MFT entry #452 as its parent. The path to MFT entry #1564 is "\Windows\System32" and the path to MFT entry #452 is "\Windows". This is known as a "**hard link**" in the file system. This file actually has two other $FILE_NAME attributes that point to other MFT entries as well (the names are long and messy, probably for internal system use).

Another thing to notice at this point is that sometimes the same item appears multiple times in a directory listing. This is usually to accommodate the DOS names and long file names. Here is a screenshot showing this:



Note that both "Program Files" and "PROGRA~1" both point to MFT entry #60. Same for "Program Data" and "PROGRA~2". This may have forensic value in that there are four more timestamps to look at. If a suspect used a program to alter timestamps it is possible that it didn't alter all of them.

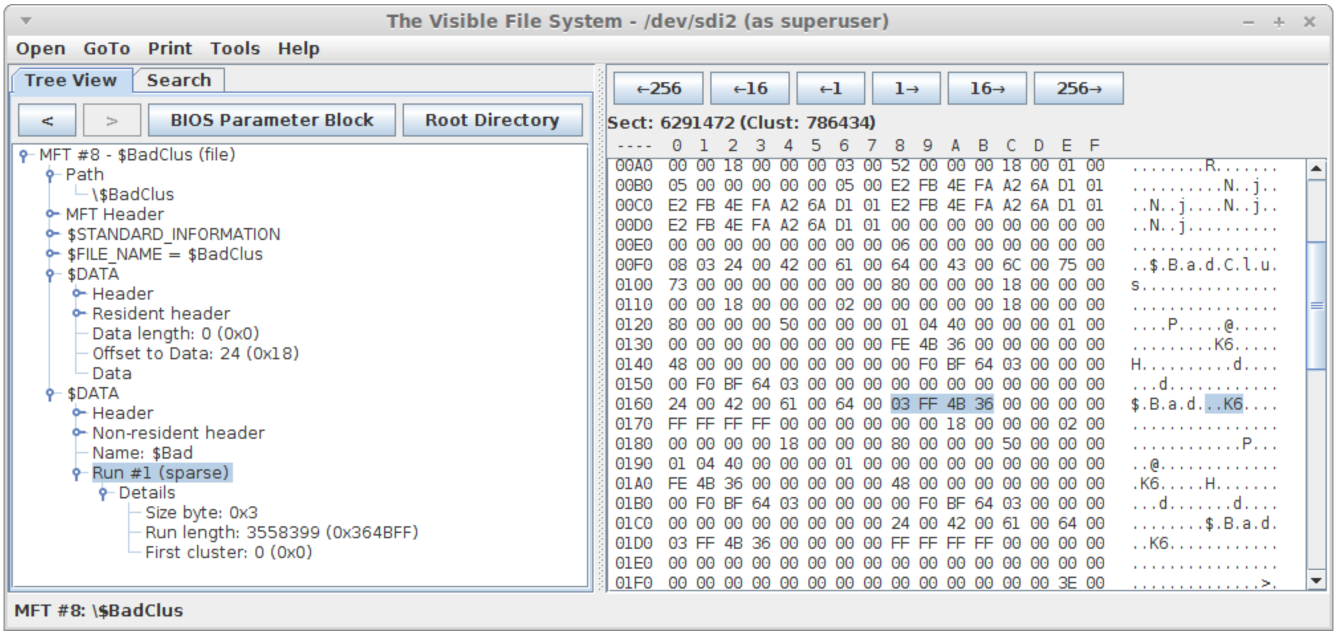
Another variation is **alternate data streams**. A given file can actually have more than one allocation of data. The first one is the default and the others are given names. Below is a screenshot of an MFT entry with an alternate data stream (mutiple $DATA attributes).



Note that the first $DATA attribute doesn't have a name and the second has the name “$Bad”. This MFT entry is used to keep track of bad clusters on a drive.

The same MFT also has a **sparse data run** (see screenshot on next page). Note the the run string is “04 FF 0A AF 01”. The “04” tells you that there are 4 bytes in the run length and 0 bytes in the starting cluster. Sparse files have zeros in entire clusters and these clusters are marked with placeholders like this and are not stored on the drive.

Also note that the first $DATA attribute has no clusters allocated. This means that it is an empty file.


The Visible File System - /dev/sdi2 (as superuser)
Open GoTo Print Tools Help
Tree View Search
BIOS Parameter Block
Root Directory
MFT #8 - $BadClus (file)
Path
\$BadClus
MFT Header
$STANDARD_INFORMATION
$FILE_NAME = $BadClus
$DATA
Header
Resident header
Data length: 0 (0x0)
Offset to Data: 24 (0x18)
Data
$DATA
Header
Non-resident header
Name: $Bad
Run #1 (sparse)
Details
Size byte: 0x3
Run length: 3558399 (0x364BFF)
First cluster: 0 (0x0)
Sect: 6291472 (Clust: 786434)
MFT #8: \$BadClus