

Inside the High Performance File System

Part 5: FNODEs, ALSECs and B+trees

Written by Dan Bridges

Introduction

This article originally appeared in the August 1996 issue of Significant Bits, the monthly magazine of the Brisbug PC User Group Inc.

Last month you saw how DIRENTs (directory entries) are stored in 4-sector structures known as DIRBLKs. These blocks have limited space available for entries. Due to the variable length of filenames (1-254 characters), the maximum number of entries depends on the average filename length. If the average name length is in the 10-13 character range, a DIRBLK can hold up to 44 entries.

When there are more files in a directory than can fit in a single DIRBLK, other DIRBLKs will be used and the connection between these blocks forms a structure known as a B-tree. Since there can be many elements (entries) in a node (DIRBLK), a HPFS B-tree has a quick "fan-out" and a low height (number of levels), ensuring fast entry location.

This time, we'll take a long look at how a file's contents are logically stored under HPFS. To the best of my knowledge, this topic has not been well-covered in the scanty information available about HPFS. You will find it helpful to contrast the following file-sector allocation methods with last month's directory entry concepts.

Fragging a File

Since HPFS is inherently fragmentation-resistant, we have to twist its arm a little to produce fragmented files. The method I came up with first fills up an empty partition with a number of files created in an ascending name sequence. The next step deletes every second file. Finally, I create a file that is approximately one-half the partition's size. This file then has nowhere to go except into all the discontinuous regions previously occupied by the deleted file entries.

This process takes some time with a large partition (100 MB) so I suggest you use a very small partition (1 MB). At first glance, you may think that if we fill up a 1 MB partition with say 100 files, then delete File1, File3, ... File99, and then create a 512K file, we will end up with a file with exactly 50 extents (fragments). This is not so, since each individual file occupies a FNODE sector as well as the sectors for the file itself, whereas a single fragmented file still has only 1 FNODE. So there is slightly more space

available in each gap for an extent than there was for a file, and a 512K file will find more than 512K of space available and ends up occupying fewer gaps than expected and we end up with a smaller number of extents than was specified. For example, in the 50-gap, 1 MB partition scenario we end up with 45 extents. There are also variations produced by things like the centrally located DIRBAND, the separate Root DIRBLK and multiple Databands to "fragment" the available freespace for very large files. So the number of gaps produced by deleting alternate files is only an rough approximation of the number of extents that will be produced.

Figure 1 shows the MakeExtents.cmd REXX program. You specify the number of gaps you want to produce. For example, to originally produce 100 files on N:, delete half of them and leave 50 gaps, you would issue the command "MakeExtents N: 50".

```
/* Produces a large, fragmented file */
PARSE ARG numOfExts
CALL RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
CALL SysLoadFuncs /* Load REXXUTIL.DLL external funcs */
CALL SysCls
EXIT /* Safety line. Delete this when you've adjusted the
      drive to suit your system. Formats the drive. */
'echo y | format n: /1 /fs:hpfs'
SAY
CALL SysMkDir 'n:\test' /* REXX MD. Faster than OS/2 MD */
currentDir = Directory() /* Store current drive/directory */
CALL Directory 'n:\test' /* Change to test dirve/directory*/
/* Determine free space */
PARSE VALUE SysDriveInfo('n:') WITH . free .

/* Determine size of each sequential file */
fileSize = (free - (numOfExts*2*512)) % (numOfExts*2)
secsInFile = fileSize % 512
sectorFill = Copies('x',512) /* 512 bytes of 'x' char */
Fill_20K = Copies(sectorFill,40) /* 20,480 bytes of 'x' */

/* Create string of the required length */
CALL MakeFile secsInFile

DO i = 1 TO numOfExts*2 /* Produce the file sequence */
  CALL CreateFile /* Fixed-length filenames: File00001 */
END i

DO i = 1 TO numOfExts*2 BY 2 /* Delete alternate files */
  CALL SysFileDelete 'n:\test\file' || Right("0000" || i,5)
END i
```

```
PARSE VALUE SysDriveInfo('n:') WITH . free .

fragmentedFileSecs = ((free-512) % 512)-1
CALL MakeFile fragmentedFileSecs

i='FRAGG'          /* Fragmented filename: FileFRAGG */
CALL CreateFile    /* Create "FileFRAGG" */
CALL Directory currentDir /* Return to original location */

EXIT              /*****/

MakeFile: PROCEDURE EXPOSE file sectorFill fill_20K
ARG secs
file = ''
/* If final file is over 20K, speed up creation a little */
IF secs>40 THEN
    file = Copies(fill_20K, secs%40)

file = file||Copies(sectorFill, secs//40)
RETURN file

CreateFile:
CALL Charout 'n:\test\file' ||Right("0000" ||i,5),file,1
CALL Stream 'n:\test\file' ||Right("0000" ||i,5),'C','CLOSE'
RETURN
```

Figure 1: The MakeExtents.cmd program produces a fragmented file. When set up correctly, this program will wipe a partition.

FNODEs, ALSECs, ALLEAFs and ALNODEs

Every file and directory on a HPFS partition has an associated FNODE, usually situated in the sector just before the file's first sector. The role of an FNODE is quite specific: to map the location of the file's extents (fragments) and any associated components, namely EAs (Extended Attributes - up to 64K of ancillary information) and ACLs (Access Control Lists - to do with LAN Manager).

FNODEs and ALSECs (to be discussed shortly) contain a list of either ALLEAF or ALNODE entries. See Figure 2. An ALLEAF entry contains three dwords: logical sector offset (where the start of this run of sectors is within the total number of sectors in the file - the logical start sector is 0); run size in

sectors; physical LSN (where the run starts in the partition). An ALLEAF entry is at the end of the B+tree. An ALNODE entry is an intermediate component in that it does not contain any extent information. Rather, it points to an ALSEC, and in turn the ALSEC can contain a list of either ALLEAFs (the end of the line) or ALNODEs (another descendant level in the B+tree).

Offset hex	(dec)	Data	Size bytes	Comment
Header				
00h	(1)	Signature	4	0xF7E40AAE
04h	(5)	Seq. Read History	4	Not implemented.
08h	(9)	Fast Read History	4	Not Implemented.
0Ch	(13)	Name Length	1	0-254.
0Dh	(14)	Name	15	Last 15 chars. (Full name in DIRBLK.)
1Ch	(29)	Container Dir LSN	4	FNODE of Dir that contains this one.
20h	(33)	ACL Ext. Run Size	4	Secs in external ACL, if present.
24h	(37)	ACL LSN	4	Location of external ACL run.
28h	(41)	ACL Int. Size	2	Bytes in internal (inside FNODE) ACL.
2Ah	(43)	ACL ALSEC Flag	1	>0 if ACL LSN points to an ALSEC.
2Bh	(44)	History Bits Count	1	Not implemented.
2Ch	(45)	EA Ext. Run Size	4	
30h	(49)	EA LSN	4	
34h	(53)	EA Int. Size	2	
36h	(55)	EA ALSEC Flag	1	>0 if EA LSN points to an ALSEC.
37h	(56)	Dir Flag	1	Bit0 = 1 if dir FNODE, else file FNODE.
38h	(57)	B+Tree Info Flag	1	0x20 (5) Parent is an FNODE, else ALSEC. 0x80 (7) ALNODEs
follow, else ALLEAFs.				
39h	(58)	Padding	3	Reestablish 32-bit alignment.
3Ch	(61)	Free Entries	1	Number of free array entries.
3Dh	(62)	Used Entries	1	Number of used array entries.

3Eh	(63)	Free Ent. Offset	2	Offset to next free
entry in array.				
If ALLEAFs (Maximum of 8 in an FNODE)				
Extent #0				
40h	(65)	Logical LSN	4	Sec offset of this
extent within file.				
The first extent has an				
offset of 0.				
44h	(69)	Run Size	4	Number of sectors in
this extent.				
48h	(73)	Physical LSN	4	File: LSN of extent
start.				
Dir: This B-tree's				
topmost DIRBLK LSN.				
...				
Extent #7				
94h	(149)	Logical LSN	4	
98h	(153)	Run Size	4	
9Ch	(157)	Physical LSN	4	
If ALNODEs (Maximum of 12 in an FNODE)				
Extent #0				
40h	(65)	End Sector Count	4	Running total of secs
mapped by this				
alnode. 1-based. If EOF				
is within this				
alnode then field				
contains 0xFFFFFFFF.				
44h	(69)	Physical LSN	4	File: LSN of ALSEC.
Dir: This B-tree's				
topmost DIRBLK LSN.				
...				
Extent #11				
98h	(153)	End Sector Count	4	
9Ch	(157)	Physical LSN	4	
Tail				
A0h	(161)	Valid File Length	4	Should be the same as
File Size in DIRENT.				
A4h	(165)	"Needed" EAs Count	4	If any, EAs vital to

the file's wellbeing.

A8h	(169)	User ID	16	Not used.
B8h	(185)	ACL/EA Offset	2	Offset in FNODE to
		first ACL, if present,		otherwise offset to
		where EAs would be		stored, if internalised.
BAh	(187)	Spare	10	Unused.
C4h	(197)	ACL/EA Storage	316	Only 145 bytes appear
		available for EAs.		

Figure 2: Layout of an FNODE. This component can contain either an array of ALNODE or ALLEAF entries.

Returning to the B-tree structure of DIRBLKs, you will remember that both intermediate and leaf components contain DIRENT data. So you may find the entry you're looking for in a node. This is not the case with a B+tree. Since an ALNODE can only point to an ALSEC, you must always proceed to the bottom of the tree, to a leaf, to retrieve extent information.

An ALNODE entry only contains two dwords: a running total indicating the logical sector offset of the last sector in the ALSEC (i.e. how far we are through the file - this starts from 1); the physical LSN of where to find the ALSEC. The advantage of the smaller entry size of an ALNODE compared to an ALLEAF is that, in the same space, there can be more of them.

An FNODE contains other data. One important piece of information is the last 15 characters of the filename. This comes in handy when we need to undelete. The last 316 bytes of the sector is also set aside for internal ACL/EAs (stored completely within the FNODE). In the Graham Utilities manual it is stated that up to 316 bytes of EAs can be stored within the FNODE but my experiments with OS/2 Warp v3 show that only up to 145 bytes of EAs can be internalised. Refer to Part 6 for further information.

Figure 3 shows the structure of an ALSEC. You will notice that there is much more space in the sector devoted to ALNODE/ALSEC entries than is available in an FNODE sector (480 bytes compared to 96 bytes). This leads to the following maximum number of entries:

	ALLEAF	ANODE
FNODE	8	12
ALSEC	40	60

Offset	Data	Size	Comment
hex (dec)		bytes	
Header			
00h (1)	Signature	4	0x37E40AAE

04h	(5)	This block's LSN other blks nearby.	4	Helps when placing
08h	(9)	Parent's LSN or another ALSEC.	4	Points to either FNODE
0Ch	(13)	Btree Flag FNODE, else ALSEC.	1	0x20 (5) Parent is an 0x80 (7) ALNODEs
follows, else ALLEAFs.				
0Dh	(14)	Padding alignment.	3	Reestablish dword
10h	(17)	Free Entries entries.	1	Number of free array
11h	(18)	Used Entries entries.	1	Number of used array
12h	(19)	Free Ent. Offset entry.	2	Offset to first free

If ALLEAFs (Maximum of 40 in an ALSEC)

Extent #0

14h	(21)	Logical LSN extent within file.	4	Sec offset of this Zero-based.
18h	(25)	Run Size	4	Secs in this extent.
1Ch	(29)	Physical LSN start.	4	File: LSN of extent Dir: This B-tree's
topmost DIRBLK LSN.				
...				

Extent #39

1E8h	(489)	Logical LSN	4
1ECh	(493)	Run Size	4
1F0h	(497)	Physical LSN	4

If ALNODEs (Maximum of 60 in an ALSEC)

Extent #0

14h	(21)	End Sector Count mapped by this is within this	4	Running total of secs alnode. 1-based. If EOF alnode then field
-----	------	--	---	---

```

contains 0xFFFF.
18h      (25)   Physical LSN          4      File: LSN of ALSEC.
                                                Dir: This B-tree's
topmost DIRBLK LSN.
...

Extent #59
1ECh     (493)  End Sector Count          4
1F0h     (497)  Physical LSN          4

Tail
1F4h     (501)  Padding              12      Unused.

```

Figure 3: The layout of an ALSEC. This component can contain either an array of ALNODE or ALLEAF entries.

Some Examples

The main program this month, ShowExtents.cmd (to be discussed later), needs to know the LSN of the FNODE or ALSEC that you want to start with. It would be possible to design a version that accepted the full pathname of a file but it would be a larger program. For the purpose of comprehending these structures, the requirement of having to specify a LSN is acceptable. To determine the file's FNODE location use last month's ShowBtree.cmd. Figure 4 shows ShowBtree's output on a 1 MB partition after "MakeExtents 7" was issued. From the information reported in Figure 4 we will first examine the TEST directory's FNODE. Figure 5 shows the result of issuing "ShowExtents N: 1033". Since there is no information in the allocation array area of a directory FNODE (the 128 byte region commencing at decimal offset 65), ShowExtents is designed to terminate early in such a situation.

```

Root Directory:
1016-1019 Next Byte Free: 125 Topmost DirBlk
This directory's FNODE: 1032 (\ [level 1]) 1016->1032
*****
SD      21 #00: ..          FNODE:1032
D       57 #01: test       FNODE:1033
E       93 #02:

36-39 Next Byte Free: 409 Topmost DirBlk
This directory's FNODE: 1033 (test [level 1]) 36->1033
*****
SD      21 #00: ..          FNODE:1033
        57 #01: file00002   FNODE:432
        97 #02: file00004   FNODE:664

```

```
137 #03: file00006      FNODE:896
177 #04: file00008      FNODE:1154
217 #05: file00010      FNODE:1386
257 #06: file00012      FNODE:1618
297 #07: file00014      FNODE:1850
337 #08: fileFRAGG      FNODE:316
E      377 #09:
```

Figure 4: Last month's program, ShowBtree.cmd, shows the LSN of FileFRAGG's FNODE.

```
FNODE STRUCTURE
LSN:                1033
Signature:          F7E40AAE
Name Length:        4
Name:               test
Container Dir LSN:  1032
EA Ext. Run Size:   0
EA LSN:             0
EA Int. Size:       0
EA ALSEC Flag:      0
Dir Flag:           Directory FNODE
Topmost DIRBLK LSN: 36
```

Figure 5: ShowExtents' output when displaying the contents of a directory FNODE.

Next, we'll look at an FNODE with a full complement of 8 ALLEAF entries. On my system, this is produced when "MakeExtents 7" is issued. See Figure 6. The next free entry in the array of ALLEAF entries is at offset 104 dec. Since the start point for this offset is counted from 65 dec, this means that the next entry would start at 169 dec. This is actually past the end of the available entry area, at the beginning of the tail region. This is another indication that the array is full. (The main indication is the "0" value in the Free Entries field.)

```
FNODE STRUCTURE
LSN:                316
Signature:          F7E40AAE
Name Length:        9
Name:               fileFRAGG
Container Dir LSN:  1033
EA Ext. Run Size:   0
EA LSN:             0
EA Int. Size:       0
EA ALSEC Flag:      0
Dir Flag:           File FNODE
```

```
B+tree Info Flag:  ALLEAFs follow
Free Entries:      0
Used Entries:     8
Next Free Offset: 104
Valid data size:  420352
"Needed" EAs:     0
EA/ACL Int. Off:  0
```

ALLEAF INFORMATION

```
Extent #0: 115 sectors starting at LSN 317 (file sec offset:0)
Extent #1: 116 sectors starting at LSN 548 (file sec off:115)
Extent #2: 116 sectors starting at LSN 780 (file sec off:231)
Extent #3: 116 sectors starting at LSN 1038 (file sec off:347)
Extent #4: 116 sectors starting at LSN 1270 (file sec off:463)
Extent #5: 116 sectors starting at LSN 1502 (file sec off:579)
Extent #6: 116 sectors starting at LSN 1734 (file sec off:695)
Extent #7: 10 sectors starting at LSN 1966 (file sec off:811)
```

Figure 6: A FNODE with a full ALLEAF array.

If we need to map any more extents we must switch from a FNODE (with ALLEAFs) structure to FNODE (with ALNODEs) -> ALSEC (with ALLEAFs). Figure 7 shows the mapping of a 10-extent file ("MakeExtents 8"). The B+tree Info Flag tells us that the FNODE contains an array of ALNODEs. There is only one entry in this array. The End Sector Count value is not shown here but, in this example, you could easily check it out using Part 2's SEC.cmd ("SEC N: 316") and then look at the four bytes at offset 40h (in the case of a single entry in the array). Since this is the sole entry, you will find FFFFFFFFh (appears to be the array End-of-Entries indicator) at this location.

FNODE STRUCTURE

```
LSN:                316
Signature:          F7E40AAE
Name Length:       9
Name:              fileFRAGG
Container Dir LSN: 1033
EA Ext. Run Size:  0
EA LSN:            0
EA Int. Size:     0
EA ALSEC Flag:    0
Dir Flag:          File FNODE
B+tree Info Flag:  ALNODEs follow
Free Entries:      11
Used Entries:     1
Next Free Offset: 16
```

```
Valid data size:    418304
"Needed" EAs:      0
EA/ACL Int. Off:   0

FNODE Entry #0
ALSEC STRUCTURE
Signature:         37E40AAE
This LSN:         933
Parent's LSN:     316
B+tree Info Flag: Parent was an FNODE; ALLEAFs follow
Free Entries:     30
Used Entries:     10
Next Free Offset: 128
```

ALLEAF INFORMATION

```
Extent #0: 101 sectors starting at LSN 317 (file sec off:0)
Extent #1: 102 sectors starting at LSN 520 (file sec off:101)
Extent #2: 102 sectors starting at LSN 724 (file sec off:203)
Extent #3: 102 sectors starting at LSN 1158 (file sec off:305)
Extent #4: 102 sectors starting at LSN 1362 (file sec off:407)
Extent #5: 102 sectors starting at LSN 1566 (file sec off:509)
Extent #6: 102 sectors starting at LSN 1770 (file sec off:611)
Extent #7: 42 sectors starting at LSN 1974 (file sec off:713)
Extent #8: 5 sectors starting at LSN 928 (file sec off:755)
Extent #9: 57 sectors starting at LSN 934 (file sec off:760)
```

Figure 7: A 10-extent file is mapped in a 1-level B+tree with a single ALSEC.

The next section in the display in Figure 7, labelled "FNODE Entry #0" shows us that the sole ALNODE entry points to LSN 933. Here we are seeing this ALSEC's layout. The B+tree Info Flag informs us that this ALSEC contains ALLEAF entries i.e. the actual mapping of the extents. Notice that we have 10 ALLEAF entries in the allocation array. Remember that an ALSEC has much more space available for array entries than an FNODE has, in that it can store up to 40 ALLEAF entries. You can verify this by adding the ALSEC's Free Entries and the Used Entries values together.

When you try and map more than 40 extents you will exceed the capacity of a sole ALSEC. What happens in this case is that more ALNODE entries are created in the FNODE, each pointing to an ALSEC. Figure 8 shows a 42-extent layout (produced with a parameter of "45").

```
FNODE STRUCTURE
LSN:              316
Signature:        F7E40AAE
Name Length:     9
```

Name: fileFRAGG
Container Dir LSN: 1033
EA Ext. Run Size: 0
EA LSN: 0
EA Int. Size: 0
EA ALSEC Flag: 0
Dir Flag: File FNODE
B+tree Info Flag: ALNODEs follow
Free Entries: 10
Used Entries: 2
Next Free Offset: 24
Valid data size: 393192
"Needed" EAs: 0
EA/ACL Int. Off: 0

FNODE Entry #0
ALSEC STRUCTURE

Signature: 37E40AAE
This LSN: 588
Parent's LSN: 316
B+tree Info Flag: Parent was an FNODE; ALLEAFs follow
Free Entries: 0
Used Entries: 40
Next Free Offset: 232

ALLEAF INFORMATION

Extent #0: 16 sectors starting at LSN 317 (file sec off:0)
...
Extent #39: 17 sectors starting at LSN 1668 (file sec off:720)

FNODE Entry #1
ALSEC STRUCTURE

Signature: 37E40AAE
This LSN: 996
Parent's LSN: 316
B+tree Info Flag: Parent was an FNODE; ALLEAFs follow
Free Entries: 38
Used Entries: 2
Next Free Offset: 32

ALLEAF INFORMATION

Extent #40: 17 sectors starting at LSN 1702 (file sec off:737)
Extent #41: 14 sectors starting at LSN 1736 (file sec off:754)

Figure 8: 42 extents require a 1-level B+tree with 2 ALNODE entries in the FNODE pointing to 2 ALSECs.

There is space in an FNODE for 12 ALNODE entries. If each of these points to a full ALSEC (with ALLEAFs) i.e. 40-entries each, this two-level structure can accommodate 480 extents (parameter "564").

Let's see what happens when we exceed this value. Figure 9 shows a 482-extent layout ("565"). Interesting things have occurred. We now have a 2-level B+tree structure. The FNODE ALNODE array has been adjusted to contain a sole entry. This in turn points to an ALSEC that has 13 ALNODE entries. Each of these ALNODE points to another ALSEC which contains ALLEAF entries. 12 of the ALSECs (with ALLEAFs) are full i.e. 12*40 while the 13th ALSEC (with ALLEAFs) only maps 2 extents.

FNODE STRUCTURE

```
LSN:                1000
Signature:          F7E40AAE
Name Length:       9
Name:              fileFRAGG
Container Dir LSN: 1033
EA Ext. Run Size:  0
EA LSN:            0
EA Int. Size:      0
EA ALSEC Flag:     0
Dir Flag:          File FNODE
B+tree Info Flag:  ALNODEs follow
Free Entries:      11
Used Entries:      1
Next Free Offset:  16
Valid data size:   524264
"Needed" EAs:      0
EA/ACL Int. Off:   0
```

FNODE Entry #0

ALSEC STRUCTURE

```
Signature:          37E40AAE
This LSN:           1333
Parent's LSN:       1000
B+tree Info Flag:   Parent was an FNODE; ALNODEs follow
Free Entries:       47
Used Entries:       13
Next Free Offset:   112
```

ALNODE INFORMATION

ALSEC Entry #0 situated at LSN 328 (file sec count:582)

ALSEC STRUCTURE

Signature: 37E40AAE

This LSN: 328

Parent's LSN: 1333

B+tree Info Flag: ALLEAFs follow

Free Entries: 0

Used Entries: 40

Next Free Offset: 232 ALLEAF INFORMATION Extent #0-#39

ALNODE INFORMATION

ALSEC Entry #1 situated at LSN 394 (file sec count:622)

ALSEC STRUCTURE 394 (40) ALLEAF INFORMATION Extent #40-#79

ALNODE INFORMATION

ALSEC Entry #2 situated at LSN 476 (file sec count:662)

ALSEC STRUCTURE 476 (40) ALLEAF INFORMATION Extent #80-#119

ALNODE INFORMATION

ALSEC Entry #3 situated at LSN 558 (file sec count:702)

ALSEC STRUCTURE 558 (40) ALLEAF INFORMATION Extent #120-#159

ALNODE INFORMATION

ALSEC Entry #4 situated at LSN 640 (file sec count:742)

ALSEC STRUCTURE 640 (40) ALLEAF INFORMATION Extent #160-#199

ALNODE INFORMATION

ALSEC Entry #5 situated at LSN 722 (file sec count:782)

ALSEC STRUCTURE 722 (40) ALLEAF INFORMATION Extent #200-#239

ALNODE INFORMATION

ALSEC Entry #6 situated at LSN 804 (file sec count:822)

ALSEC STRUCTURE 804 (40) ALLEAF INFORMATION Extent #240-#279

ALNODE INFORMATION

ALSEC Entry #7 situated at LSN 886 (file sec count:862)

ALSEC STRUCTURE 886 (40) ALLEAF INFORMATION Extent #280-#319

ALNODE INFORMATION

ALSEC Entry #8 situated at LSN 968 (file sec count:902)

ALSEC STRUCTURE 968 (40) ALLEAF INFORMATION Extent #320-#359

ALNODE INFORMATION

```
ALSEC Entry #9 situated at LSN 1085 (file sec count:942)
ALSEC STRUCTURE 1085 (40) ALLEAF INFORMATION Extent #360-#399
```

ALNODE INFORMATION

```
ALSEC Entry #10 situated at LSN 1167 (file sec count:982)
ALSEC STRUCTURE 1167 (40) ALLEAF INFORMATION Extent #400-#439
```

ALNODE INFORMATION

```
ALSEC Entry #11 situated at LSN 1249 (file sec count:1022)
ALSEC STRUCTURE 1249 (40) ALLEAF INFORMATION Extent #440-#479
```

ALNODE INFORMATION

```
ALSEC Entry #12 situated at LSN 1331 (file sec count:At end)
ALSEC STRUCTURE 1331 (2) ALLEAF INFORMATION Extent #480-#481
```

Figure 9: 482 extents are mapped by a 2-level B+tree with 1 ALNODE entry in the FNODE pointing to 1 ALSEC, which in turn points to 13 ALSECs.

If you look at FNODE Entry #0's Used & Free Entries values you can verify that, in an ALSEC, there can be a maximum of 60 ALNODEs. It would take $60 \times 40 = 2,400$ extents to fill this level up again. Going past this would require the presence of a second FNODE entry. Since we can have up to 12 ALNODE entries in an FNODE, this means we could map $12 \times 60 \times 40 = 28,800$ extents before the need to insert another intermediary ALSEC level would arise.

On a 100 MB partition I produced a 3-level 44,413 extent structure ("44500"). To put this discussion on B+tree fan-out in perspective, it should be remembered that, in the fragmentation analysis performed in Part 3 on 20,800 files in 5 partitions, there were only 14 files with more than 8 extents (i.e. requiring an ALSEC) and the largest number of extents reported was 30.

The ShowExtents Program

Figure 10 presents the ShowExtents.cmd REXX program. You will need to get SECTOR.DLL. The program first determines if the LSN you've specified belongs to an FNODE or ALSEC. (You can bypass the FNODE and commence the examination from an ALSEC.) Once it has determined this, the next most important consideration is: does the allocation array consist of ALLEAFs or ALNODEs? If it contains ALLEAFs we've reached the end of the tree and need only show the extents. If we are looking at an array of ALNODEs we need to recurse down each ALNODE entry, loading the ALSEC pointed to by the entry and then see whether it contains either ALLEAFs or ALNODEs. And so on...

```
/*Shows the layout of FNODE and ALSECs. Requires SECTOR.DLL*/
PARSE UPPER ARG drive lsn
/* There must be at least two parms supplied */
```

```
IF drive = '' | lsn = '' THEN CALL HELP
/* Register external functions */
CALL RxFuncAdd 'QDrive','sector','QDrive'
CALL RxFuncAdd 'ReadSect','sector','ReadSect'
alleafEntryCount = 0
anodeEntryCount = 0
SAY
CALL MainRoutine
EXIT  /*****EXECUTION ENDS HERE*****/
```

MainRoutine:

```
PROCEDURE EXPOSE drive lsn alleafEntryCount anodeEntryCount
usedEntries = 0
sectorString = ReadSect(drive,lsn) /* Read in required sec */
IF FourBytes2Hex(1) = 'F7E40AAE' THEN
  /* Is an FNODE */
  DO
    alSecIndicator = ''
    CALL DisplayFnode
  END
ELSE
  /* Not an FNODE */
  DO
    IF FourBytes2Hex(1) = '37E40AAE' THEN
      /* Is an ALSEC */
      DO
        alSecIndicator = 'Y'
        CALL DisplayALSEC
      END
    ELSE
      /* Neither an FNODE or an ALSEC */
      DO
        SAY 'LSN' lsn 'is not an FNODE or ALSEC'
        EXIT
      END
    END
  END
END
RETURN
```

DisplayFnode:

```
SAY 'FNODE STRUCTURE'
SAY 'LSN:    ' lsn
```

```
SAY 'Signature:      ' FourBytes2Hex(1)
SAY 'Name Length:   ' Bytes2Dec(13,1)
SAY 'Name:         ' Substr(sectorString,14,Bytes2Dec(13,1))
SAY 'Container Dir LSN:' Bytes2Dec(29,4)
SAY 'EA Ext. Run Size:' Bytes2Dec(45,4)
SAY 'EA LSN:       ' Bytes2Dec(49,4)
SAY 'EA Int. Size:  ' Bytes2Dec(53,2)
SAY 'EA ALSEC Flag: ' Bytes2Dec(55,1)
IF Bitand(Byte2Char(56),'1'x) = '1'x THEN
    dirFlag = 'Directory FNODE'
ELSE
    dirFlag = 'File FNODE'

SAY 'Dir Flag:      ' dirFlag
IF dirFlag = 'Directory FNODE' THEN
    SAY 'Topmost DIRBLK LSN:' || Bytes2Dec(73,4)
ELSE
    DO
        /* Is a file, so determine extents */
        CALL DetermineBtreeInfo 57
        SAY 'B+tree Info Flag: ' btreeInfo
        SAY 'Free Entries:     ' Bytes2Dec(61,1)
        usedEntries = Bytes2Dec(62,1)
        SAY 'Used Entries:     ' usedEntries
        SAY 'Next Free Offset:  ' Bytes2Dec(63,2)
        SAY 'Valid data size:   ' Bytes2Dec(161,4)
        SAY '"Needed" EAs:       ' Bytes2Dec(165,4)
        SAY 'EA/ACL Int. Off:   ' Bytes2Dec(169,4)
        CALL ShowALLEAF_or_ANODE
    END
RETURN

FourBytes2Hex: /* Given offset, return Dword */
ARG startPos
rearranged = Reverse(Substr(sectorString,startPos,4))
RETURN C2X(rearranged)
```

```
Bytes2Dec:
ARG startPos,numOfChars
temp = Substr(sectorString,startPos,numOfChars)
IF C2X(temp) = 'FFFFFFFF' THEN
    RETURN 'At the end'
```

```
ELSE
    RETURN Format(C2D(Reverse(temp)),,0)
```

```
Byte2Char:
ARG startPos
RETURN Substr(sectorString,startPos,1)
```

```
DetermineBtreeInfo:
ARG btreeByteOffset
IF Bitand(Byte2Char(btreeByteOffset),'20'x) = '20'x THEN
    btreeInfo = 'Parent was an FNODE; '
ELSE
    btreeInfo = ''

IF Bitand(Byte2Char(btreeByteOffset),'80'x) = '80'x THEN
    DO
        btreeInfo = btreeInfo||'ALNODEs follow'
        alNodeIndicator = 'Y'
    END
ELSE
    DO
        btreeInfo = btreeInfo||'ALLEAFs follow'
        alNodeIndicator = 'N'
    END
RETURN
```

```
DisplayALSEC:
SAY 'ALSEC STRUCTURE'
alSecIndicator = 'Y'
SAY 'Signature:    ' FourBytes2Hex(1)
lsn = Bytes2Dec(5,4)
SAY 'This LSN:    ' lsn
SAY "Parent's LSN:    " Bytes2Dec(9,4)
CALL DetermineBtreeInfo 13
SAY 'B+tree Info Flag: ' btreeInfo
SAY 'Free Entries:    ' Bytes2Dec(17,1)
usedEntries = Bytes2Dec(18,1)
SAY 'Used Entries:    ' usedEntries
SAY 'Next Free Offset: ' Bytes2Dec(19,1)
CALL ShowALLEAF_or_ANODE
```

RETURN

```
ShowALLEAF_or_ANODE: PROCEDURE EXPOSE drive lsn sectorString,  
    usedEntries alleafEntryCount anodeEntryCount entrySize,  
    alsecIndicator alnodeIndicator
```

```
IF alsecIndicator = 'Y' THEN
```

```
    entryOffset = 21
```

```
ELSE
```

```
    entryOffset = 65
```

```
IF alnodeIndicator \= 'Y' THEN
```

```
    /* Is an ALLEAF */
```

```
    DO
```

```
    SAY
```

```
    IF usedEntries = 0 THEN
```

```
        DO
```

```
        SAY 'Zero-length file'
```

```
        EXIT
```

```
        END
```

```
    SAY 'ALLEAF INFORMATION'
```

```
    entrySize = 12
```

```
    DO entry = alleafEntryCount TO alleafEntryCount+usedEntries-1
```

```
        fileSecOffset = Bytes2Dec(entryOffset,4)
```

```
        runSize = Bytes2Dec(entryOffset+4,4)
```

```
        physicalLSN = Bytes2Dec(entryOffset+8,4)
```

```
        SAY 'Extent #' || entry || ':' runSize 'sectors starting
```

```
            at LSN' physicalLSN '(file sec offset:' || fileSecOffset
```

```
||')'
```

```
                /* Wrapped long line */
```

```
            entryOffset = entryOffset+entrySize
```

```
        END entry
```

```
    alleafEntryCount = entry
```

```
    END
```

```
ELSE
```

```
    DO
```

```
    /* Is either an ALNODE in an ALSEC or in an FNODE */
```

```
    entrySize = 8
```

```
    IF alSecIndicator \= 'Y' THEN
```

```
        /* In an FNODE */
```

```
        DO entry = anodeEntryCount TO anodeEntryCount+usedEntries-1
```

```
        lsn = Bytes2Dec(entryOffset+4,4)
        SAY
        SAY 'FNODE Entry #' || entry
        CALL MainRoutine
        entryOffset = entryOffset+entrySize
    END entry
ELSE
    DO
    /* In an ALSEC */
    listStart = 65
    sectorString = ReadSect(drive,lsn)
    DO entry = anodeEntryCount TO anodeEntryCount+usedEntries-1
        SAY
        SAY 'ALNODE INFORMATION'
        fileSecOffset = Bytes2Dec(entryOffset,4)
        lsn = Bytes2Dec(entryOffset+4,4)
        SAY 'ALSEC Entry #' || entry 'situated at LSN'
            lsn '(file sec count:' || fileSecOffset || ')'
            /* Wrapped long line */
        CALL MainRoutine
        anodeEntryCount = entry
        entryOffset = entryOffset+entrySize
    END entry
    END
END
RETURN
```

```
Help:
SAY 'ShowExtents shows the extents mapped by a FNODE or ALSEC'
SAY 'structure.'
SAY
SAY ' Usage: ShowExtents drive LSN_of_a_FNODE/ALSEC '
SAY ' Example: ShowExtents C: 316'
EXIT
```

Figure 10: The ShowExtents.cmd program.

Counting Extents

It is handy to be able to report just the number of extents in a file. HPFS-EXT, in the Graham Utilities, can do this. It take a filename. It is available in the demo version of the GU's, "GULITE.xxx".

The freeware FST (currently FST03F.xxx) does just about everything. You can specify either a filename ("FST INFO N: \TEST\FILEFRAGG" - note the space after the drive letter) or a LSN ("FST INFO N: 1000"). It will include the height of the B+tree and the total number of extents at the end of its display. Unfortunately, it displays a lot of other info, and sometimes you're only interesting in just the number of levels and extents.

I cut down ShowExtents.cmd to produce CountExtents.cmd The design was not amenable to showing the height but it was a straightforward matter to show just the number of extents. I've not bothered to present it here since most readers will probably prefer to specify the filename. (The FNODE LSN keeps changing as you increase the number of extents so this makes it more difficult to use CountExtents.)

Conclusion

In this installment we have seen how a file's sectors are mapped by FNODEs and ALSECs. These file system components can contain either an array of ALNODE or ALLEAF entries. By following through to the ALLEAFs we can examine the mapping of extents.

We have also seen how a B+tree is different from a B-tree. In a DIRBLK B-tree, DIRENT information can be found in a node entry. But in an ALSEC B+tree, extent information is not stored in node entries, only in the leaves. The filling of nodes in an ALSEC B+tree is also much more efficient than the utilisation of nodal space in a DIRENT's B-Tree.

When the next installment is presented we'll look at Extended Attributes. While not specifically a HPFS topic, they are well integrated into the file system and will fit well into this series.