The structure of the Reiser file system

by Florian Buchholz

The Reiser file system was created by Hans Reiser. The design objectives were to increase performance over the ext2 file system, offer a space efficient file system, and to improve handling of large directories compared to existing file systems. Reiserfs uses balanced trees to store files and directories and it also offers journaling.

This document describes the on-disk structure of the Reiser file system version 3.6. This document does not describe how the file system tree is balanced, how the journaling is performed, or how files and directories are managed within an implementation of the file system.

Blocks

The reiserfs partition is divided into blocks of a fixed size. The blocks are numbered sequentially starting with block 0. There is a maximum number of 2^32 possible blocks in one partition.

The partition starts with the first 64k unused to leave enough room for partition labels or boot loaders. After that follows the superblock. The superblock contains important information about the partition such as the block size and the block numbers of the root and journal nodes. The superblock block number differs depending on the block size, but always starts at byte 65536 of the partition. The default block size for reiserfs under Linux is 4096 bytes. This makes the superblock block number 16. There is only one instance of the superblock for the entire partition.

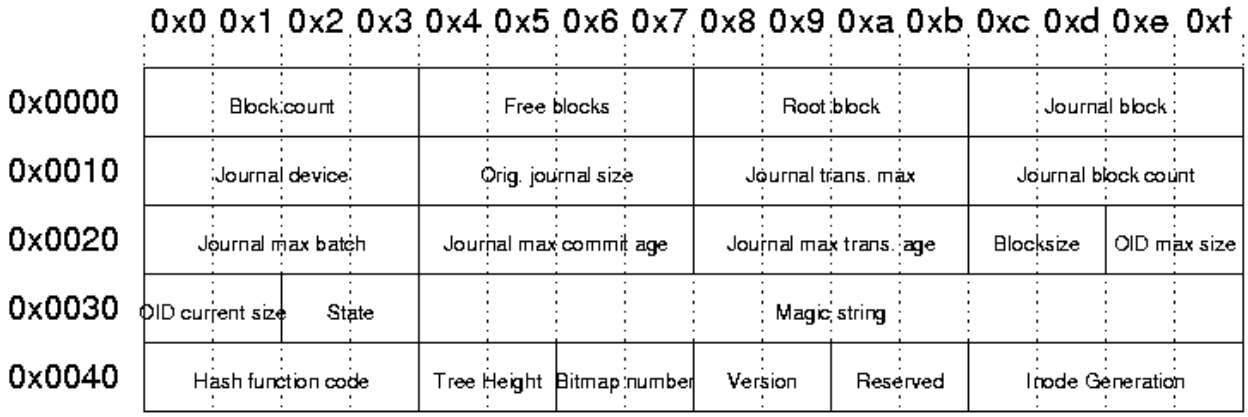
Directly following the superblock is a block containing a bitmap of free blocks. The number of blocks mapped in the bitmap depends directly on the block size. If a bitmap can map k blocks, then every k-th block will be a new bitmap block.

Block size	**4,096**	**512**	**1,024**	**8,192**
#blocks in bitmap	32,768	4,096	8,192	65,536
superblock	16	128	64	8
1st bitmap	17	129	65	9
2nd bitmap	32,768	4,096	8,192	65,536
3rd bitmap	65,536	8,192	16,384	131,072
4th bitmap	98,304	12,288	24,576	196,608

...
(assuming that the partition is large enough to have 2nd, 3rd, 4th bitmap)

Following the first bitmap block should be the journal, but the information in the superblock is the authorative source for that information.

The Superblock



Name	Size	Description
Block count	4	The number of blocks in the partition
Free blocks	4	The number of free blocks in the partition
Root block	4	The block number of the block containing the root node
Journal block	4	The block number of the block containing the first journal node
Journal device	4	Journal device number (not sure what for)
Orig. journal size	4	Original journal size. Needed when using partition on systems with different default journal sizes.
Journal trans. max	4	The maximum number of blocks in a transaction
Journal magic	4	A random magic number
Journal max batch	4	The maximum number of blocks in a transaction
Journal max commit age	4	Time in seconds of how old an asynchronous commit can be
Journal max trans. age	4	Time in seconds of how old a transaction can be
Blocksize	2	The size in bytes of a block
OID max size	2	The maximum size of the object id array
OID current size	2	The current size of the object id array
State	2	State of the partition: valid (1) or error (2)
Magic string	12	The reiserfs magic string, should be "ReIsEr2Fs"
Hash function code	4	The hash function that is being used to sort names in a directory
Tree Height	2	The current height of the disk tree
Bitmap number	2	The amount of bitmap blocks needed to address each block of the file system
Version	2	The reiserfs version number
Reserved	2	
Inode Generation	4	Number of the current inode generation.

The inode generation number is a counter that denotes the current generation of inodes. The counter is increased every time the tree gets re-balanced.

Example:

The following is the start of the superblock of a 256MB reiserfs partition on an Intel based system:

```
00000000 66 00 01 00 93 18 00 00 82 40 00 00 12 00 00 00  f........@......
00000010 00 00 00 00 00 20 00 00 00 04 00 00 ac 34 11 57  ..... ......¬4.W
00000020 84 03 00 00 1e 00 00 00 00 00 00 00 00 10 cc 03  ..............Ì.
00000030 08 00 02 00 52 65 49 73 45 72 32 46 73 00 00 00  ....ReIsEr2Fs...
00000040 03 00 00 00 04 00 03 00 02 00 00 00 dc 52 00 00  ............ÜR..
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	66	00	01	00	93	18	00	00	82	40	00	00	12	00	00	00
0x0010	00	00	00	00	00	20	00	00	00	04	00	00	ac	34	11	57
0x0020	84	03	00	00	1e	00	00	00	00	00	00	00	00	10	cc	03
0x0030	08	00	02	00	52	65	49	73	45	72	32	46	73	00	00	00
0x0040	03	00	00	00	04	00	03	00	02	00	00	00	dc	52	00	00

Block count: 65638
Free blocks: 6291
Root block: 16514
Journal block: 18
Journal device: 0
Original journal size: 8192
Journal trans. max: 1024
Journal magic: 1460745388
Journal max. batch: 900
Journal max. commit age: 30
Journal max. trans. age: 0
Blocksize: 4096
OID max. size: 972
OID current size: 8
State: 2 (error)
Magic String: ReIsEr2Fs
Hash function code: 3
Tree height: 4
Bitmap number: 3
Version: 2
Inode generation: 21212

Bitmap blocks

The bitmap blocks are simple bitmaps, where every bit stands for a block number. One bitmap block can address (8 * block size) number of blocks. Byte 0 of the bitmap maps to the first eight blocks, the second byte to the next eight, and so on. Within a byte, the low order bits map to the the lower number blocks. Bit 0 maps to the first block, bit 1 to the second, etc. A set bit indicates that the block is in use, a zero bit that the block is free.

Example:

```
00000400 ff ff f7 ff 7f 00 00 00 00 00 00 00 00 80 cb bd  ÿÿ÷ÿ..........Ë½
```

These 16 bytes of bitmap block 0 describe block numbers 8192 to 8319.

Blocks 8192-8210:	used
Block 8211:	free (f7 is 11110111 binary)
Blocks 8212-8230:	used
Blocks 8231-8302:	free
Blocks 8303-8305:	used
Block 8306:	free
Block 8307:	used
Blocks 8308-8309:	free
Blocks 8310-8312:	used
Block 8313:	free
Blocks 8314-8317:	used
Block 8318:	free
Block 8319:	used

Had the above entry been from a bitmap block other than bitmap block 0, then (bitmap block # * block size * 8) needs to be added for the proper block number. By bitmap block # we understand the ordinal number (0 for the 1st, 1 for the second, ...) not the block number of the bitmap block.

Given a block number b, one can determine its status as follows:

b div (8 * block size) : bitmap block # (integer division)

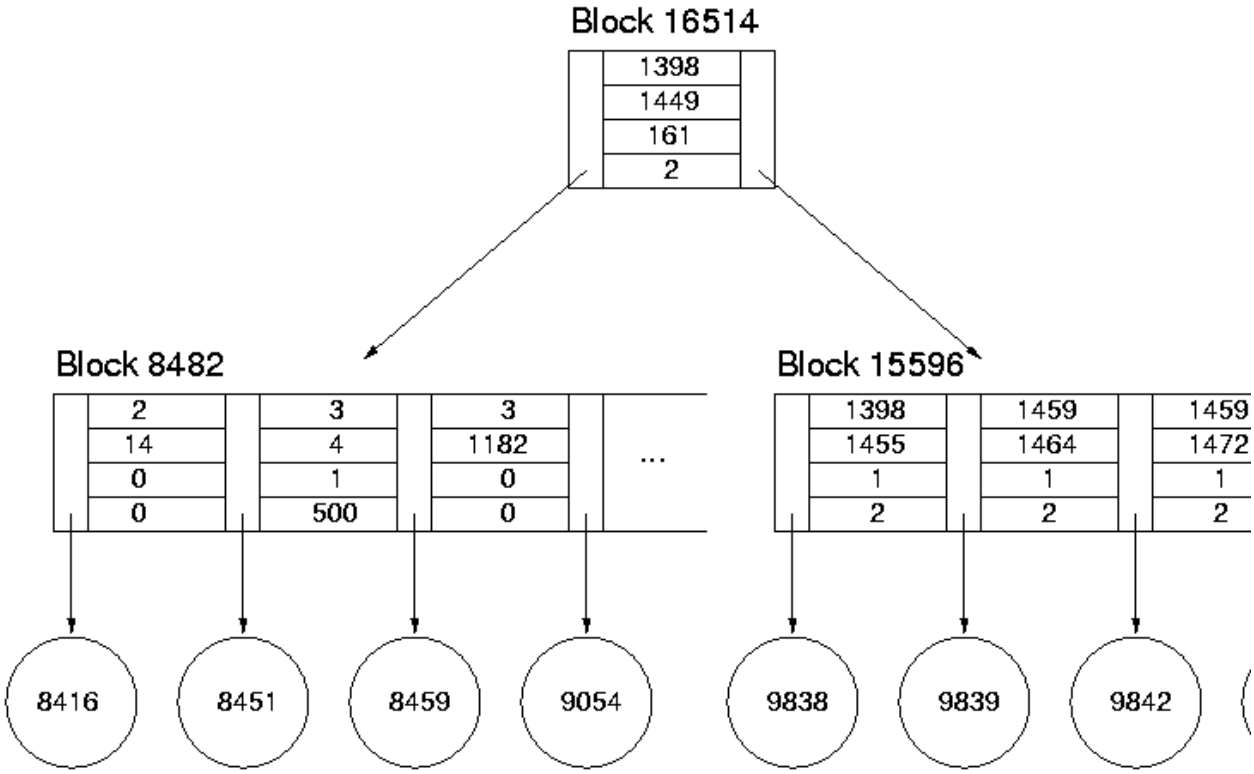
Let r = b mod (8* block size), then

r div 8: byte within bitmap block, and
r mod 8: bit within byte

The File System Tree

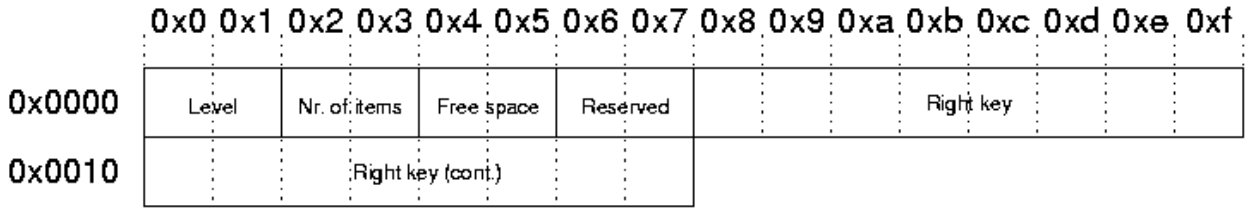
The Reiser file system is made up of a balanced tree (B+ or S+ tree as it is called in the reiserfs documentation). The tree is composed of internal nodes and leaf nodes. Each node is a disk block. Each object (called an item) in reiserfs (file, directory, or stat item) is assigned a unique key, which can be compared to an inode node number in other file systems. The internal nodes are mainly composed of keys and pointers to their child nodes. There is always one more pointer than there are keys. P0 points to the objects that have keys smaller than K0, P1 to those K0<=obj

For our example partition, part of the S+ tree looks like this (think of the key as a large 128-bit number for now):



Block headers

Each disk block that belongs to an internal or leaf node starts with a block header. Only unformatted blocks don't have a block header. A block header is always 24 bytes long and contains the following information:



Name	Size	Description
Level	2	level of the block in the tree
Nr. of items	2	number of items in the block
Free space	2	free space left in the block
Reserved	2	
Right key	16	right delimiting key for the block

The right delimiting key was originally used for leaf nodes but is now only kept for compatibility.

Example:

The following is the block header of block 8416, the leftmost leaf node in the tree.

```
00000000 01 00 06 00 e4 04 00 00 00 00 00 00 00 00 00 00  ....ä...........
00000010 00 00 00 00 00 00 00 00
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	01	00	06	00	e4	04	00	00	00	00	00	00	00	00	00	00
0x0010	00	00	00	00	00	00	00	00								

Level: 1
Items: 6
Free space: 1252 bytes

Keys

Keys are used in the Reiser file system to uniquely identify items, but also to locate them in the tree and achieve local groupings of items that belong together. A key consists of four objects: the directory id, the object id, the offset within the object, and a type. Note that the actual object identifier is only one part of the key. The directory id is present so that files that belong into the same directory are grouped together and for the most part are located in the same subtree(s). The offset is present because an indirect item can at most contain (blocksize-48)/4 pointers to unformatted blocks (see indirect items below). For a block size of 4096 bytes this would result in a maximum file size of 4048KB. To be able to handle larger files, multiple keys are used to reference the file. All fields of the key are the same, except for the offset, which denotes the offset in bytes of the file, which a particular key references. I do not know why the type of an object is part of the actual key.

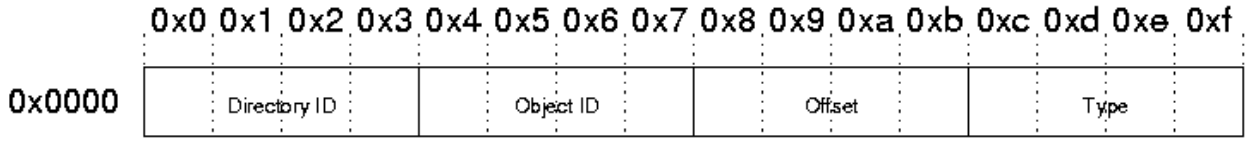
In reiserfs up until version 3.5 the offset and the type fields were both 4 byte values. This meant, that the maximum file size was limited to roughly 2^32 bytes, or 4GB (2^32 bytes plus the data of one more indirect item plus the tail, actually). To increase the maximum file size in the file system, in version 3.6, the offset field was increased to 60 bits, and the type field shrunk to 4 bits. This now allows for a theoretical maximum file size of 2^60 bytes, but since there can be only 2^32 blocks with a maximum of 2^16 bytes per block, the file system itself only supports 2^48 bytes.

In order not to be incompatible to older versions of the file system, there are now to different versions of keys around, which can be very confusing as the key itself doesn't carry a version number. To make up for this, the formerly reserved last 16 bits of the item header now contain a version number, so if necessary, the key's version number can be obtained from there. This makes it fairly straightforward for keys contained in leaf nodes, but if one really wanted to determine the version of a key inside an internal node, one would have to follow the tree down to the leaf, first. The code in the reiserfs library actually uses this ugly hack to determine the key format:

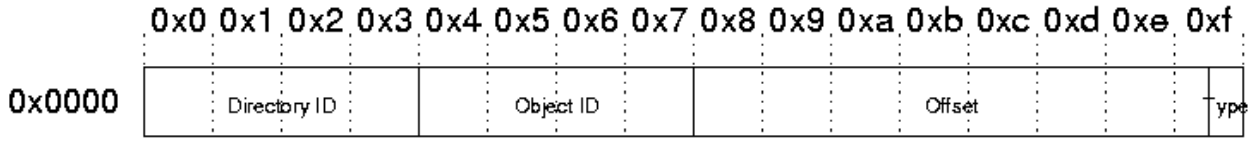
```
static inline int is_key_format_1 (int type) {
    return ( (type == 0 || type == 15) ? 1 : 0);
}

/* old keys (on i386) have k_offset_v2.k_type == 15 (direct and
   indirect) or == 0 (dir items and stat data) */

/* */
int key_format (const struct key * key)
{
```

```
    int type;

    type = get_key_type_v2 (key);

    if (is_key_format_1 (type))
        return KEY_FORMAT_1;

    return KEY_FORMAT_2;
}
```

This actually implies that stat items will always be assumed to have KEY_FORMAT_1, because they, also, have a type of zero in version 2.



Name	Size	Description
Directory ID	4	the identifier of the directory where the object is located
Object ID	4	the actual identifier of the object ("inode number")
Offset	4	the offset in bytes that this key references
Type	4	the type of item. Possible values are: Stat: 0 Indirect: 0xfffffffe Direct: 0xffffffff Directory: 500 Any: 555



Name	Size	Description
Directory ID	4	the identifier of the directory where the object is located
Object ID	4	the actual identifier of the object ("inode number")
Offset	60 bits	the offset in bytes that this key references
Type	4 bits	the type of item. Possible values are: Stat: 0 Indirect: 1 Direct: 2 Directory: 3 Any: 15

Only stat items have an offset of 0. Files (direct and indirect items) and directories always start with an offset of 1 so that they are sorted behind the stat item in the leaf nodes. For directory items the "offset" field contains the hash value and generation number of the leftmost directory header of the directory item (see below), not the offset in bytes.

Examples:

The following shows the first two keys of the internal node that is contained in block 8482. The first one is of version 2, the second of version 1.

```
00000000 02 00 00 00 0e 00 00 00 00 00 00 00 00 00 00 00  ................
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	0	0

Directory id: 2
Object id: 14
Offset: 0
Type: Stat item (0)

```
00000000 03 00 00 00 04 00 00 00 01 00 00 00 f4 01 00 00  ............ô...
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	03	00	00	00	04	00	00	00	01	00	00	00	f4	01	00	00

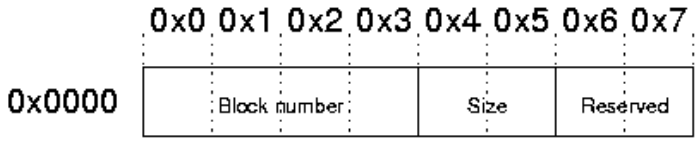
Directory id: 3
Object id: 4
Offset: 1
Type: Directory item (500)

Two keys are compared by comparing their directory ids first, and if those are equal, by comparing the object ids, and so on for offset and type. The fact that the Linux reiserfs code generates a warning when the type fields need to be compared for keys stored in memory indicates that the type field does not matter from a structural point of view. The only time the field needs to be compared seems to be during "tail conversion", where a direct item is changed into an indirect one.

Internal nodes

An internal node block consists of the block header, keys, and pointers to child nodes. Other than the figure of the S+-tree above, the internal nodes have all the keys first, which are sorted by the key values. Then following the last key comes the pointers, starting with the pointer to the subtree containing all the keys smaller to the first key.

The level in the block header should always be larger than 1 for internal nodes. The number of items in the block header denotes the number of keys in the node, not the combined number of keys and pointers. There is always one more pointer than there are keys. The following figure describes the layout of the pointer structure:



Given a key n (whose position in the block is 24 + n * 16 bytes) and a total number of k keys in the block, the left pointer that corresponds to key n can be found at byte 24 + k * 16 + n * 8. The free space starts at byte blocksize - free space, where free space is the value from the block header.

Example:

```
00000000 02 00 a0 00 e0 00 00 00 00 00 00 00 00 00 00 00  .. .à..........
00000010 00 00 00 00 00 00 00 00 02 00 00 00 0e 00 00 00  ................
00000020 00 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00  ................
00000030 01 00 00 00 f4 01 00 00 03 00 00 00 9e 04 00 00  ....ô...........
00000040 00 00 00 00 00 00 00 00 04 00 00 00 05 00 00 00  ................
...
00000a10 01 10 00 00 00 00 00 20 e0 20 00 00 04 0b b4 cc  ....... à ....´Ì
00000a20 03 21 00 00 94 0d 54 c5 0b 21 00 00 e0 0f 2f c5  .!....TÅ.!..à./Å
00000a30 5e 23 00 00 b4 0f f4 ff 60 23 00 00 38 07 a9 ff  ^#..´.ôÿ`#..8.©ÿ
...
```

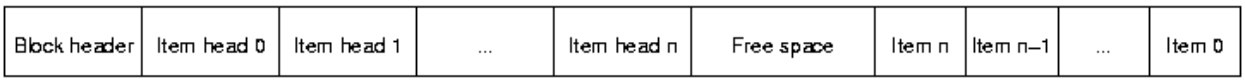
Level: 2
Nr. items: 160
Free space: 224 bytes

Key 0: {2, 14, 0, 0}
Key 1: {3, 4, 1, 500}
Key 2: {3, 1182, 0, 0}
...
Ptr 0: {8416, 2820}
Ptr 1: {8451, 3479}
Ptr 2: {8459, 4064}
Ptr 3: {9054, 4020}
...

This example shows parts of block 8482, which is also depicted in the diagram describing the S+-tree above. Key 0 starts at byte 24 (0x18), and since there are 160 items in the block, Ptr 0 starts at byte 2584 (0xa18). Note that the reserved parts of the pointers actually contain junk data. The free space starts at byte 3872 (0xf20) and it may also contain junk data.

Leaf nodes

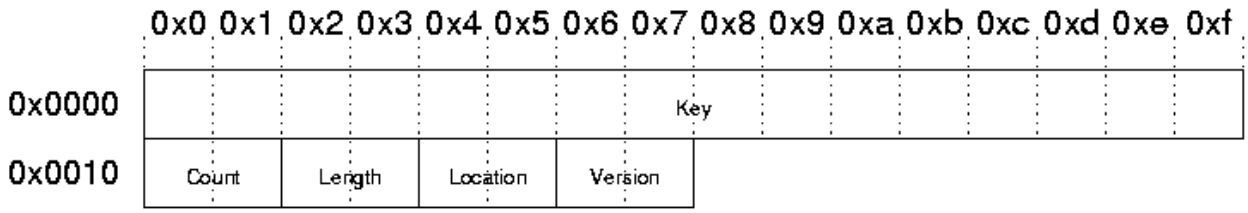
Leaf nodes are found at the lowest level of the S+-tree. Except for indirect items all the data is contained within the leaf nodes. Leaf nodes are made up of the block header, item headers, and items:



Note that the free space in the block is located between the last item header and item, and that items are in reverse order. This way, new item headers and items can simply be added without having to rearrange existing items. New headers go after the last header, and new items before the first on-disk item. Also note that items are of variable length.

Item Headers

The item header describes the item it refers to. It contains the key for the item as well as the item's location and size within the leaf node. The type of the item is determined by its key.



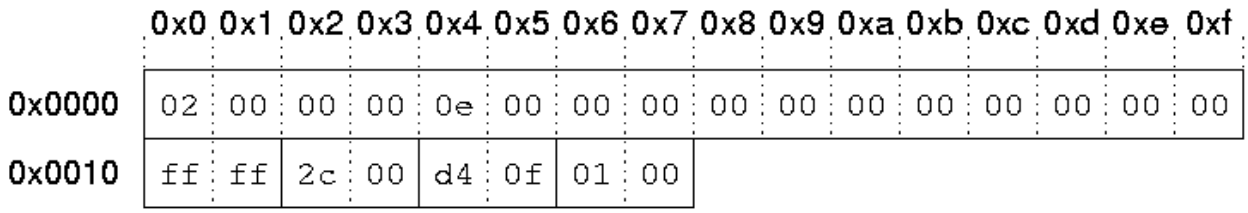
Name	Size	Description
Key	16	The key that belongs to the item
Count	2	The free space in the last unformatted node for an indirect item if this is an indirect item 0xffff for stat and direct items the number of directory entries for a directory item
Length	2	total size of the item
Location	2	offset to the item body within the block
Version	2	0 for all old items (keys), 1 for new ones

Note that the comments in the structure definition indicate that new items have a version of 2. However, the KEY_FORMAT_3_6 constant is defined as 1 and this is used to set the version.

Example:

The following is the item header for the stat item described by key {2, 14, 0, 0}, which was used earlier as an example of type 2 (version 3.6). It shows that the version is indeed the new version, even though the heuristic above would indicate an old key.

```
00000000 02 00 00 00 0e 00 00 00 00 00 00 00 00 00 00 00  ................
00000010 ff ff 2c 00 d4 0f 01 00                          ÿÿ,.Ô...
```



Key: {2, 14, 0, 0}
Count: 0xffff
Length: 44 bytes
Location: byte 4052
Version: 1 (3.6)

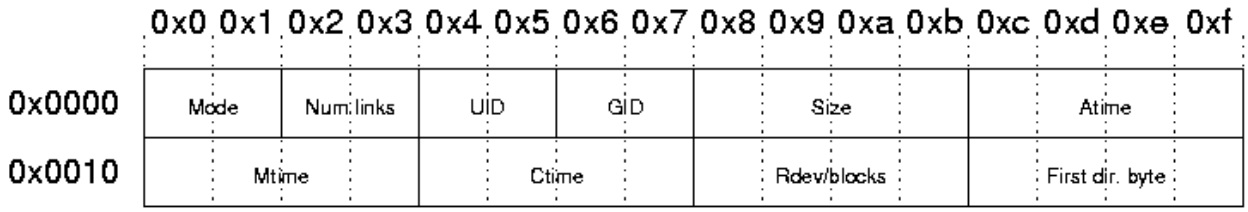
Items

Items finally contain actual data. There are four types of items: stat items, directory items, direct items, and indirect items. Files are made up of one or more direct or indirect item, depending on the file's size. Every file and directory is preceded by a stat item.

Stat Items

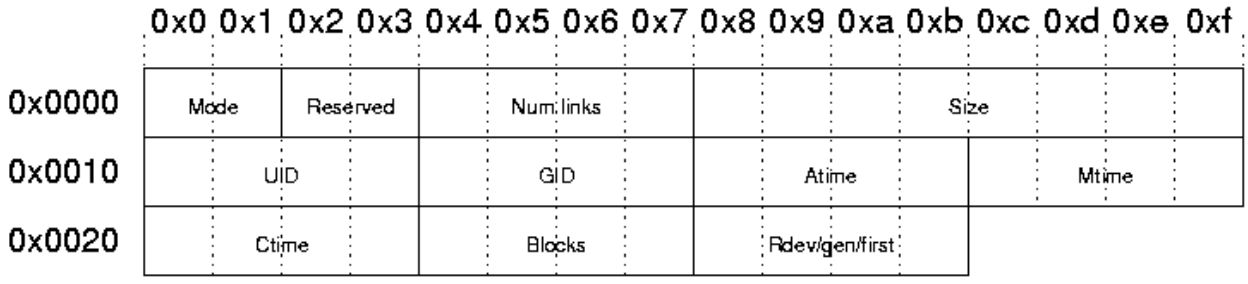
Stat items contain the meta-data for files and directories. Keys belonging to stat items always have an offset and type of 0, so that the stat item key always comes first before the other one(s) belonging to the same "inode number". Due to the same reason that there are two versions of keys, there are also two versions of stat items, as the size field was increased from 32 bits to 64 bits. For some reason, the fields for number of hard links, user id, and group id also were increased from 16 bits to 32 bits, each and other fields were introduced. Thus a stat item of version 3.5 is 32 bytes in size, whereas one of version 3.6 has 44 bytes.

The structure of a stat item of version 1:



Name	Size	Description
Mode	2	file type and permissions
Num links	2	number of hard links
UID	2	user id
GID	2	group id
Size	4	file size in bytes
Atime	4	time of last access
Mtime	4	time of last modification
Ctime	4	time stat data was last changed
Rdev/blocks	4	Device number / number of blocks file uses
First dir. byte	4	first byte of file which is stored in a direct item if it equals 1 it is a symlink if it equals 0xffffffff there is no direct item.

The structure of a stat item of version 2:



Name	Size	Description
Mode	2	file type and permissions
Reserved	2	
Num links	4	number of hard links

Name	Size	Description
Size	8	file size in bytes
UID	4	user id
GID	4	group id
Atime	4	time of last access
Mtime	4	time of last modification
Ctime	4	time stat data was last changed
Blocks	4	number of blocks file uses
Rdev/gen/first	4	Device number/ File's generation/ first byte of file which is stored in a direct item if it equals 1 it is a symlink if it equals 0xffffffff there is no direct item.

The file mode field identifies the type of the file as well as the permissions. The low 9 bits (3 octals) contain the permissions for world, group, and user, the next 3 bits (from lower to higher) are the sticky bit, the set GID bit, and the set UID bit. The high 4 bits contain the file type. On a Linux system, possible values for the file type are (as defined in stat.h):

Constant Name	16-bit Mask	4-bit value	Description
S_IFSOCK	0xc000	12	socket
S_IFLNK	0xa000	10	symbolic link
S_IFREG	0x8000	8	regular file
S_IFBLK	0x6000	6	block device
S_IFDIR	0x4000	4	directory
S_IFCHR	0x2000	2	character device
S_IFIFO	0x1000	1	fifo

Other operating systems might have additional file types. Only regular files and directories have other items associated with the stat item. In all the other cases the stat item makes up the entire file.

The "rdev" field applies to special files that are not regular files (S_IFREG), directories (S_IFDIR), or links (S_IFLNK). In those cases, the field holds the device number (or socket number) belonging to the file. The "generation" field applies to the other cases and denotes the inode generation number for the file/directory/link (see above for superblock inode generation field' description). The "first" field doesn't seem to be used in version 2 anymore.

Example:

The following example shows the stat item denoted by key {2, 14, 0, 0} from the item header example above:

```
00000000 ff 43 05 00 03 00 00 00 50 00 00 00 00 00 00 00  ÿC......P.......
00000010 00 00 00 00 00 00 00 00 2d 1c 17 3f 34 94 ff 3e  ........-..?4.ÿ>
00000020 34 94 ff 3e 01 00 00 00 00 00 00 00              4.ÿ>........
```

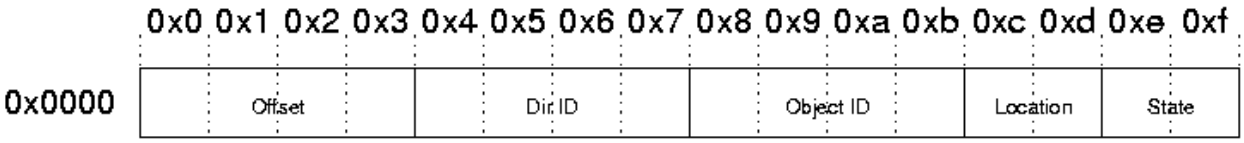
	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	ff	43	05	00	03	00	00	00	50	00	00	00	00	00	00	00
0x0010	00	00	00	00	00	00	00	00	2d	1c	17	3f	34	94	ff	3e
0x0020	34	94	ff	3e	01	00	00	00	00	00	00	00				

Mode: 0x43ff -- type: directory, sticky bit set, 777 permissions
Reserved: 5
Num. links: 3
Size: 80 bytes
UID: 0
GID: 0
Atime: Thu Jul 17 16:59:09 2003
Mtime: Sun Jun 29 20:36:52 2003
Ctime: Sun Jun 29 20:36:52 2003
Blocks: 1
First: 0

Directory Items

Directory items describe a directory. If there are too many entries in a directory to be contained in one directory item, it will span across several directory items, using the offset value of the key. Directory items are made up of directory headers and file names. Just like leaf nodes, the free space (if there is any) is located in the middle of the item. The structure of a directory item is as follows:

Directory headers contain an offset, the first two parts of the referenced item's key (directory id and object id), the location of the name within the block, and a status field.



Name	Size	Description
Offset	4	Hash value and generation number
Dir ID	4	object id of the referenced item's parent directory
Object ID	4	object id of the referenced item
Location	2	offset of name within the item
State	2	bit 0 indicates that item contains stat data (not used) bit 2 whether entry is visible (bit set) or hidden

The file names are simple zero-terminated ASCII strings. File name entries seem to be 8-byte aligned,

but the information in the directory headers should be the authorative source for the start of the name (and implicitly the end by looking at the previous header entry). The "offset" field is aptly misnamed as it contains a hash value of the file name. Bits 7 through 30 of the field contains the actual hash value and bits 0 through 6 a generation number in case two file names within a directory hash to the same value. Bit 31 seems to be unused. The hash value is used to actually search for file and directory names in reiserfs, and the directory items are sorted by the offset value. Three different hash functions are possible: keyed tea hash, rupasov hash, and r5 hash. The purpose of the hash function is to create different values for different strings with as little collisions as possible. In the Linux implementation of reiserfs, the r5 hash seems to be the default.

Example:

The following example is an entire directory item, that belongs to the stat item example from the previous section:

```
00000000 01 00 00 00 02 00 00 00 0e 00 00 00 48 00 04 00  ............H...
00000010 02 00 00 00 01 00 00 00 02 00 00 00 40 00 04 00  ............@...
00000020 00 6d 6f 73 0e 00 00 00 60 00 00 00 30 00 04 00  .mos....`...0...
00000030 76 69 2e 72 65 63 6f 76 65 72 00 00 00 00 00 00  vi.recover......
00000040 2e 2e 00 00 00 00 00 00 2e 00 00 00 00 00 00 00  ................
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	01	00	00	00	02	00	00	00	0e	00	00	00	48	00	04	00
0x0010	02	00	00	00	01	00	00	00	02	00	00	00	40	00	04	00
0x0020	00	6d	6f	73	0e	00	00	00	60	00	00	00	30	00	04	00
0x0030	76	69	2e	72	65	63	6f	76	65	72	00	00	00	00	00	00
0x0040	2e	2e	00	00	00	00	00	00	2e	00	00	00	00	00	00	00

Header 0: {hash 0, gen. 1, 2, 14, byte 0x48, 4 (bit 2 set: visible)}
Header 1: {hash 0, gen. 2, 1, 2, byte 0x40, 4 (bit 2 set: visible)}
Header 2: {hash 15130330, gen. 0, 14, 96, byte 0x30, 4 (bit 2 set: visible)}
Name 2: "vi.recover"
Name 1: ".."
Name 0: "."

As one can see, the directory referenced by key {2, 14, 0, 0} consists of 3 entries, which in return have the following keys (all these keys will lead to the stat item for the directory first):

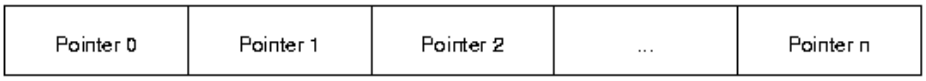
```
.          {2, 14, 0, 0}
..         {1, 2, 0, 0}
vi.recover {14, 96, 0, 0}
```

Direct Items

Direct items contain the entire file body of small files or the tail of a file. For small files, all the necessary other information can be found in the item header and the corresponding stat item for the file. For the tail of a file, the key for the direct item is the last one for the file.

Indirect Items

In direct items contain pointers to unformatted blocks that belong to a file. Each pointer is 4 bytes long and contains the block number of the unformatted block. An indirect item that takes up an entire leaf node can at most contain (blocksize-48) / 4 pointers (the 48 bytes are for the block and item headers). In a partition with 4096 bytes block size, a single indirect item can at most reference 4145152 bytes (4048 KB: 1012 pointers to 4K blocks). Larger files are composed of multiple indirect items, using the offset value in the key, plus a possible tail.



The Journal

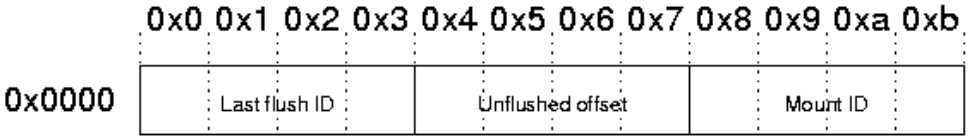
The journal in reiserfs is a continuous set of disk blocks and it describes transactions made to the file system. Each time the file system is modified in any way, instead of performing the changes directly in the file system, the transactions that belong together (those that need to be atomic so that the file system is in a consistent state) are written into the journal first. At a later point the transactions in the journal will be flushed and, if everything was successful, marked as such.

The journal is of fixed size in the file system. In the 2.4.x Linux implementation the journal size is fixed at 8192 blocks plus one block for the journal header. The journal itself consists of variable-length transactions and a journal header. The journal starts with the list of transactions and the journal header is at the end of the journal. A transaction spans at least three disk blocks and the journal header is exactly one block. The journal is a circular buffer, meaning that once the last block of the journal is reached, it wraps around and uses the first block again.

It can often be read that reiserfs only records the file system meta data in its journal. This is not entirely correct. It is true, that purpose of the journaling is to ensure the integrity of the meta data. However, reiserfs journals entire disk blocks as they have to appear in the file system after the journal transaction is committed. Since directories, stat data and small files are stored directly in the leaf nodes of the tree, some amount of data is also contained in the journal and could be used to reconstruct earlier versions of a file or directory.

Journal Header

The journal header is a single block which describes where the first unflushed transaction can be found in the journal. The journal header is the last block of the journal. In our example the journal's first transaction starts at block 18 and there are 8192 journal blocks. Therefore. the journal header is at block 8210. There are only 12 bytes of information in the journal header. The rest of the block is undefined.



Name	Size	Description
Last flush ID	4	The transaction ID of the last fully flushed transaction

Name	Size	Description
Unflushed offset	4	The offset (in blocks) of the next transaction in the journal
Mount ID	4	The mount ID of the flushed transaction

The transaction pointed to by the offset must have a higher transaction ID or a higher mount ID than the flushed transaction in order to be considered an unflushed transaction. If this is not the case, all transactions are considered flushed and the block pointed to by the offset is used to start recording new journal transactions.

Example:

```
00000000 e2 74 02 00 24 1c 00 00 1d 01 00 00 12 00 00 00  ât..$...........
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb
0x0000	e2	74	02	00	24	1c	00	00	1d	01	00	00

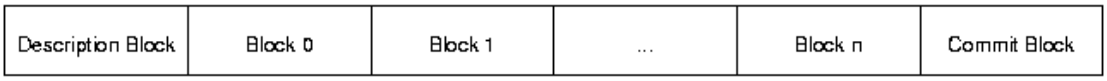
Last flush ID: 160994
Unflushed offset: 7204 blocks
Mount ID: 285

In this example, the first unflushed transaction can be found at block 7222 (since the journal starts at block 18). However, the block found there does not contain a transaction description (see below) and therefore there aren't any unflushed transactions for the partition.

Transactions

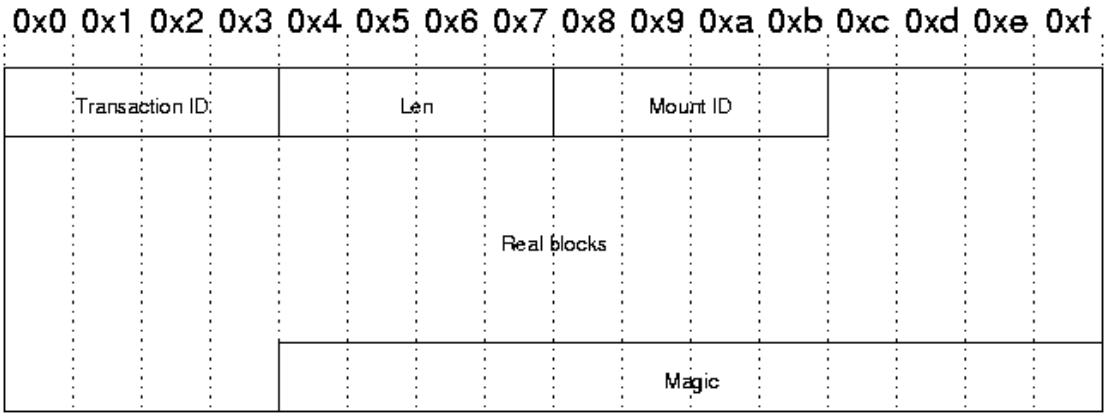
Transactions describe changes in the file system. Instead of directly modifying blocks in the file system tree, instead the new or changed blocks are first written into the journal and mapped to their real location in the file system.

A transaction consists of a transaction description block, a list of blocks, and a commit block at the end. All those blocks are contiguous within the journal.



Description block

The description block contains the transaction and mount IDs, the number of blocks in the transaction, a magic number, and the first possible half of mappings.



Name	Size	Description
Transaction ID	4	The transaction ID
Len	4	Length (in blocks) of the transaction
Mount ID	4	Mount ID of the transaction
Real blocks	Block size - 24	Mapping for blocks in transaction
Magic	12	Magic number. Should be "ReIsErLB"

The "Real blocks" field is theoretically dependant on the block size. The first 12 bytes of the block have the IDs and the length, and the last 12 bytes contain the magic string. Everything in between is used for the block mapping. However, in the Linux 2.4.x implementation, the struct for a description block defines

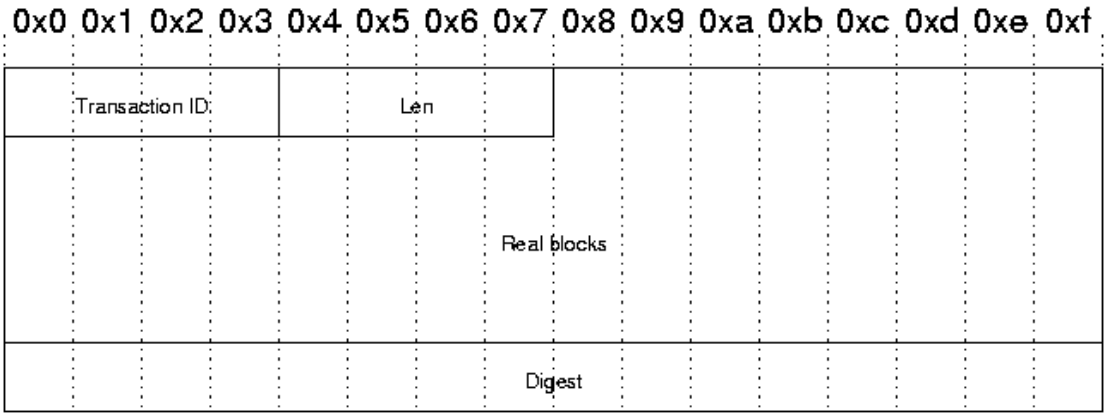
```
__u32 j_realblock[JOURNAL_TRANS_HALF];
```

where JOURNAL_TRANS_HALF is a constant set to 1018. This means that the blocksize has to be 4096 for journaling to work with reiserfs under Linux!

The actual block mapping is done as follows: The "Real blocks" field is seen as an array that contains for each block in the transaction the actual block number of the block in the file system. If we number every four bytes in the field as r0 through rn, then block 0 of the transaction is how block number r0 needs to look like after flushing the journal. Block 1 of the transaction is block r1, and so on. If the "Real blocks" field of the description block is not large enough, the field in the commit block is used in addition. This limits the maximum number of blocks in one transaction to 2*(blocksize-24)/4. (2036 for a block size of 4K), but the actual limit is set in the superblock.

Commit block

The commit block terminates a transaction. It contains a copy of the transaction ID and the transaction length. There is also a 16 byte field reserved for a digest value at the end of the block, but this is not used currently.



Name	Size	Description
Transaction ID	4	The transaction ID
Len	4	Length (in blocks) of the transaction
Real blocks	Block size - 24	Mapping for blocks in transaction
Digest	16	Digest of all blocks in transaction. Not used.

Example:

The following example describes an old transaction in our example partition. The transaction starts in block 7243 (the description block), spans 4 data blocks (7244-7247) and has its commit block at block number 7248. Only the description block is shown, as the other blocks are not relevant for the example.

```
00000000 1b 6e 02 00 04 00 00 00 1b 01 00 00 90 22 00 00  .n...........".. 
00000010 07 f7 00 00 aa 22 00 00 10 00 00 00 00 00 00 00  .÷..ª".......... 
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................ 
...
00000ff0 00 00 00 00 52 65 49 73 45 72 4c 42 00 00 00 00  ....ReIsErLB.... 
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0000	1b	6e	02	00	04	00	00	00	1b	01	00	00	90	22	00	00
0x0010	07	f7	00	00	aa	22	00	00	10	00	00	00	00	00	00	00
0x0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
...								...								
0x00ff0	00	00	00	00	52	65	49	73	45	72	4c	42	00	00	00	00

Transaction ID: 159259
Length: 4 blocks
Mount ID: 283
Real blocks[0]: 8848
Real blocks[1]: 63239

Real blocks[2]: 8874
Real blocks[3]: 16
Magic: ReIsErLB

This transaction therefore describes the following mapping: when the transaction is committed/flushed, block 7244 is written to block 8848, block 7245 to block 63239, block 7246 to block 8874, and block 7247 to block 16 (the superblock).

Navigating reiserfs

In addition to the file system tree itself, in order to access files, one needs to navigate through the directory tree, as well. The root directory of a Reiser file system always has the key {1, 2, 0, 0}. The keys for subsequent directories and files within the directory hierarchy can then be found in the headers of the directory items. Since the keys in reiserfs are sorted by parent directory ID first, items that are in the same directory are grouped together in the file system tree. This allows for searching for keys locally instead of always having to go through the root node of the file system.

A key {a, b, 0, 0} will always yield the stat item of the directory or file, and subsequent items will follow immediately after that in the file system tree. The stat item contains the size of the actual item in bytes. With this information and using the size information of the individual item headers, the keys for other parts of the directory/file can be constructed and the parts located. In many cases, the items will be arranged consecutively on the disk, anyway.

The following three examples will show three different types of files: a very small file consisting only of a stat item and a tail, a larger file that actually has an indirect item, and finally a very large file that spans over multiple indirect items. We again use the example partition from above, which is an image of a partition mounted as "/var" in a SuSe Linux 8.0 system.

Example 1: small file

The first example is that of a small file that contains only of a stat item and one direct item. The file is "/var/log/y2start.log-initial". The root directory ("/var") has key {1,2,0,0}, which by navigating the file system tree can be found in block 8416. There we can find that the "log" directory has key {2,13,0,0}. This directory is also contained in block 8416. The file "y2start.log-initial" has key {13, 1633, 0, 0}. By inspecting block 8482, we find that this key is contained in the leaf node block number 24224. The item headers for the keys {13, 1633, 0, 0} and {13, 1633, 1, 2} are as follows:

```
00000090 0d 00 00 00 61 06 00 00 00 00 00 00 00 00 00 00  ....a...........
000000a0 ff ff 2c 00 a4 0b 01 00 0d 00 00 00 61 06 00 00  ÿÿ,.¤.......a...
000000b0 01 00 00 00 00 00 00 20 ff ff f0 00 b4 0a 01 00  ....... ÿÿð.´...
```

Key: {13, 1633, 0, 0}
Count: 0xffff
Length: 44 bytes
Location: byte 2980 (0xba4)
Version: 1 (new)

Key: {13, 1633, 1, 2}
Count: 0xffff
Length: 240 bytes
Location: byte 2740 (0xab4)
Version: 1 (new)

At byte 2740 (0xab4) in the block, we find the direct item for the file followed by the stat item at byte 2980 (0xba4):

```
00000ab0              65 6e 76 0a 65 63 68 6f 20 59 32 44      env.echo Y2D
00000ac0 45 42 55 47 20 28 29 0a 6d 65 6d 69 6e 66 6f 20  EBUG ().meminfo
00000ad0 31 20 3d 20 4d 65 6d 3a 20 31 30 33 33 34 35 36  1 = Mem: 1033456
```

```
00000ae0 20 38 35 39 37 36 20 39 34 37 34 38 30 20 30 20   85976 947480 0
00000af0 36 34 32 34 20 35 37 31 37 32 0a 69 53 65 72 69  6424 57172.iSeri
00000b00 65 73 3d 31 0a 68 76 63 5f 63 6f 6e 73 6f 6c 65  es=1.hvc_console
00000b10 3d 31 0a 58 31 31 69 3d 0a 4d 65 6d 54 6f 74 61  =1.X11i=.MemTota
00000b20 6c 3d 31 30 33 33 34 35 36 0a 66 62 64 65 76 5f  l=1033456.fbdev_
00000b30 6f 6b 3d 31 0a 75 70 64 61 74 65 3d 0a 58 56 65  ok=1.update=.XVe
00000b40 72 73 69 6f 6e 3d 34 0a 58 53 65 72 76 65 72 3d  rsion=4.XServer=
00000b50 66 62 64 65 76 0a 78 73 72 76 3d 58 46 72 65 65  fbdev.xsrv=XFree
00000b60 38 36 0a 73 63 72 65 65 6e 3d 66 62 64 65 76 0a  86.screen=fbdev.
00000b70 6d 65 6d 69 6e 66 6f 20 32 20 3d 20 4d 65 6d 3a  meminfo 2 = Mem:
00000b80 20 31 30 33 33 34 35 36 20 39 32 34 30 34 20 39   1033456 92404 9
00000b90 34 31 30 35 32 20 30 20 38 32 33 32 20 36 30 35  41052 0 8232 605
00000ba0 31 36 0a 00 a4 81 05 00 01 00 00 00 ef 00 00 00  16..¤.......ï...
00000bb0 00 00 00 00 00 00 00 00 00 00 00 00 25 15 3e 3d  ............%.>=
00000bc0 25 15 3e 3d 25 15 3e 3d 08 00 00 00 d5 02 00 00  %.>=%.>=....Õ...
```

Mode: S_IFREG (regular file), -rw-r--r--
Num. links: 1
Size: 239
UID: 0
GID: 0
A/M/Ctimes: 07/23/2002 21:47:01
Blocks: 8
Gen: 725

Note that the stat item contains the correct size for the file, 239 bytes. This means that byte 2979 (0xba3) of the block does not belong to the file anymore.

Example 2: file with indirect item

The file "/var/log/SaX.log" is 7121 bytes long. It therefor cannot fit as a direct item and needs to be split either into two unformatted blocks or one unformatted block and a tail. In this case, the file will take up two unformatted blocks described by one indirect item. The key for the file is {13, 1490, 0, 0} and examining block 8482 we find out that it is contained in leaf node block number 27444.

```
00000040                          0d 00 00 00 d2 05 00 00          ....Ò...
00000050 00 00 00 00 00 00 00 00 ff ff 2c 00 a4 0b 01 00  ........ÿÿ,.¤...
00000060 0d 00 00 00 d2 05 00 00 01 00 00 00 00 00 00 10  ....Ò...........
00000070 00 00 08 00 9c 0b 01 00                          ........
```

Key: {13, 1490, 0, 0}
Count: 0xffff
Length: 44 bytes
Location: byte 2980 (0xba4)
Version: 1 (new)

Key: {13, 1490, 1, 1}
Count: 0
Length: 8 bytes
Location: byte 2972 (0xb9c)
Version: 1 (new)

```
00000b90                                        12 52 00 00              .R..
00000ba0 13 52 00 00 a4 81 05 00 01 00 00 00 d1 1b 00 00  .R..¤.......Ñ...
00000bb0 00 00 00 00 00 00 00 00 00 00 00 00 3f aa 4a 3d  ............?ªJ=
00000bc0 bd aa 4a 3d bd aa 4a 3d 10 00 00 00 54 05 00 00  ½ªJ=½ªJ=....T...
```

Mode: S_IFREG (regular file), -rw-r--r--

Num. links: 1
Size: 7121
UID: 0
GID: 0
C time: Fri Aug 2 10:50:23 2002
M/Atimes: Fri Aug 2 10:52:29 2002
Blocks: 10
Gen: 1364
Block 1: 21010
Block 2: 21011

The file is thus made up of the contents of blocks 21010 and 21011. Block 21010 contains a full 4096 bytes of data, whereas block 21011 contains only 3025 bytes. For some reason, though the item header for the indirect item (see above) doesn't contain a count of 1071 bytes as one would have expected.

Example 3: a large file

The file "/var/lib/rpm/fileindex.rpm" is a file of over 11 MB in size. A single indirect item can not describe the file, as there isn't enough space in a block for such a large indirect item. The file has the key {4, 7, 0, 0}, which can be found in block 16822. This block, however, contains only the stat item for the file. The indirect items for the file span over three more blocks: Key {4, 7, 1, 1} is in block 13286, key {4, 7, 4145153, 1} in block 20171, and key {4, 7, 8290305, 1} in block 20987. Block 13286 contains one single indirect item:

```
00000010                          04 00 00 00 07 00 00 00          ........
00000020 01 00 00 00 00 00 00 10 00 00 d0 0f 30 00 01 00  ..........Ð.0...
```

Key: {4, 7, 1, 1}
Count: 0
Length: 4048 bytes
Location: byte 48 (0x30)
Version: 1 (new)

What follows are 1012 pointers to unformatted blocks. Block 20171 has the same structure. Block 20987 also holds just one indirect item, but uses only 3320 bytes for 830 pointers. Note how the offset for the next key derives directly from offset of the previous key and the number of pointers in the previous indirect item:

1 + (1012 pointers * 4096 bytes blocksize) = 4145153
4145153 + (1012 pointers * 4096 bytes blocksize) = 8290305

Back to forensics page

Last modified: Sun Aug 17 18:19:30 EST 2003