Augmented File System

The Augmented File System (AFS) is an major evolution of the Macintosh File Manager (FM). It consists of the FM programmatic interfaces with mostly new implementations "behind" them - in particular in the areas of disk space management and the directory.

The immediate motivation for AFS is to provide support for very large disk storage volumes (on the order of 100Mbytes) and the large number of individual files (on the order of 10,000 files) that will be found on such volumes. Since the development of AFS is a significant effort, it should address future expansion to handle even larger volumes - 500Mbytes and 50,000 files. (Note that volume here does not have to be a single disk drive. What logically appears to be a volume could, via software, be constructed out of several physical volumes.)

The first use of AFS will be in the AppleBus File Server. In the future AFS might be the integral file system for Big Mac (or even used as an External File System on 512k Macs).

Motivation

The FM has several problems in dealing with large volumes:
1. The FM has an allocation map with 12-bit entries for each allocation block. If each allocation block is 2 disk blocks (1024 bytes), then such an allocation map for a 100Mbyte volume would be over 200,000 bytes or 400 disk blocks.
2. The FM keeps the entire allocation block map in memory. Even if this information were "paged" through memory, its 400 block size would be time consuming to process.
3. The FM has a fixed number of blocks on disk for the file directory. To provide safely for 10,000 files about 1400 disk blocks would have to be allocated at volume initialization, regardless of whether they were ever used.
4. The FM's file directory is a simple list (non-hierarchical) and unsorted. Users can't cope with 10,000 files in this fashion. Dynamically sorting that number of files or inferring some structure from file names would be prohibitive. The Finder also needs hierarchical structuring to deal with large numbers of files, replacing its current in-memory, dynamically-created structures.
5. The FM makes little provision for caching of disk structures in memory. Caching will be necessary for at least the directory information on very large volumes.

Approach

While there are several problems with the current File mManager, its design is not gratuitously ignored in designing the AFS. Since the File Server is based on the Macintosh OS and the AFS is a candidate for use on Macintosh, the design of AFS is similar to and upward compatable from the current FM where possible.

1. The interfaces to the FM should be usable to address files stored in an AFS volume.
2. Additional interfaces will be provided to allow more sophisticated enumeration of files through the hierarchical directory of AFS.
3. For the File Server, most of the FM code will be replaced to handle different space allocation and directory structures. Only higher-level code in the FM which maps client calls into lower-level operations will be retained.
4. If/when AFS is used inside a Macintosh, it will have to exist as an External File System, assuming the continued existince of the current Macintosh FM.

Overview of AFS Volume Format

(This discussion ignores block sparing. It is assumed that block sparing is performed in the device driver for the rigid disk and that the AFS sees the volume as a sequence of n logical disk blocks numbered (0 . . n-1). Block sparing is not discussed further.)

The volume contains 5 categories of information (see Figure AFS-1):
1. Startup information (boot tracks) used during system startup for bootstrapping.
2. Information describing the volume (VI).
3. Space allocation information (SAI).
4. A hierarchical directory (DIR).
5. The remainder of the volume - disk blocks allocated to (resource or data forks of) files or available for new files or file extensions.

Category 1 information is not discussed in detail in this document.

Details of AFS Volume Format

File numbers - File numbers are 32-bit numbers. Each new file created on the volume gets a file number one greater than that of the file created just before it. The first file created gets file number 5 (see below for 1-4). File number 0 is used for certain available blocks (see below).

Logical Block Format - There are n logical blocks on a volume numbered (0 . .

n-1). Each logical block is 524 bytes and has exactly the same format as blocks on standard FM volumes - 12 bytes of tags (same subfields as in the FM - file *, resource/data flag, sequence * in file, modified date) and 512 bytes of data.

File Body Format - As a consequence of the above two facts, the format of the body of files on an AFS volumes is similar to those on an FM volume. The body consists of the blocks on the volume that comprise the data and resource forks plus a header page(s) which precedes the pages in the data fork.

The header page is used in conjunction with the directory entry for the file. The directory entry contains file attributes which must be quickly obtainable (name, dates, etc.) plus a pointer to the file's body. The header page contains file attributes whose access can take longer (eg., are only needed when a file is opened/closed) as well as redundant information for scavenging the directory.

The header page(s) is described more completely in the section on the Directory.

Startup Information - Blocks (0 .. 14) are startup information - bootstrap code and bad block tables. They "belong" to the boot ROM and disk device driver. These blocks are file number 1. Flag information for these blocks: file number=1; fork type=file attributes=0; mod date=date last bad block added for bad block page, bootstrap install date for other pages (that are used).

Volume Information (VI) - The Volume Information Block (VIB) is stored in pages 15 and n-1 of the volume. It is a superset of the FM Volume Information (see figure AFS-2). The backup VIB on page n-1 us updated when the volume is unmounted. These blocks are tagged as file number 2; fork type=file attributes=0; mod date=date of last update.

1. Very large volumes are supported by extending the fields that count allocation blocks to 32 bits. (With allocation units of 2Kbytes, a WORD would only suffice for up to 128Mbyte volumes.)
2. A fixed block is allocated for the Directory (this simplifies allocation of space and allows internal Directory node pointers to be 16 bits). It can be expanded by moving what follows it. An initial allocation of .5% of the volume size will suffice for almost all situations. Several other fields are present for the Directory. They are described in that section.
3. A field has been added at the end to indicate the presence of additional expansion volumes. The first version of the AFS will not implement this capability. Expansion volumes are basically just additional allocation blocks added to the end of the space on the primary volume. Expansion volumes don't contain directory or space allocation information.

Scavenging - Volume information can be recreated or verified by scanning the SAI and DIR areas of the disk.

Space Allocation Information (SAI) - There are four blocks statically allocated to record free space on the volume. The blocks holding SAI are file number 3. Flag information for this block: file number=3; fork type=file attributes=0; mod date=date of last update of each page. The SAI describes the availability of space across the entire disk - space consumed by system startup information, volume information, SAI itself, and the Directory is marked allocated at volume initialization.

In this design, the free space on a volume is described by (allocation block #, # blocks) pairs called e***xtents***. Each extent describes a contiguous range of free allocation blocks. (Unused extent entries contain 0, 0). For a newly initialized volume (up to 128Mbytes) there is one extent in the SAI area. The SAI area is shown in figure AFS-3. The statically allocated SAI space can hold up to 408 extents. If the free space on the volume becomes further fragmented, then additional 4-block tables are allocated from the pool of available space as shown in AFS-3.

This part of the design described disk data structures, not algorithms. There are many possible strategies on keeping sorted (all or just the first 408) extents (probably by allocation block #), keeping unused extent entries at the end, etc. Also, the extents describing the largest areas might be kept in the first 408 entries. When space is freed, the implementation should check whether it is adjacent to already free space before blindly adding another extent. Exactly how to manage the SAI is left to the implementation and tuning.

A SQUISH utility is described below. Essentially its goal is to reduce the fragmentation of free space (like Crunch in UCSD Pascal) and to reduce the fragmentation within each file. The existence of this utility running in the background on the File Server or being explicitly invoked will tend to reduce the number of extents in SAI to one.

Scavenging - The SAI information can be verified/reconstructed by subtracting system space (SSI, VI, SAI, DIR) and enumerating the Directory (see below) subtracting all the space allocated to files.

Directory - The directory is used to map textual file names into information about the file and its location on the disk. Since the directory should be able to store 50,000 entries, the linear, unsorted approach of the FM is not adequate.

The AFS directory:
1. implements a sorted, hierarchical name structure so that the large number of files can be effectively presented to the end user;
2. includes sufficient information such that the Finder can operate without scanning the entire directory and building the file tree in memory (as it does now with the FM).

Name Hierarchy - The Directory supports a hierarchical name space of the form:
root>X>Y> . . . >File
Throughout this discussion, the root is implicit and pathnames for a file are stated in the form:
X>Y> . . . >File
The strings delimited by ">" are the *elements* of the name and can be up to 32 characters in length. All elements of the pathname except the last one are naming directories (same as containers or folders). (NOTE that from here on **D**irectory refers to the Directory as a whole, while **d**irectory refers to a container or sub-directory within the Directory's hierarchical name space.) The last element names an actual file (eg, the MacWrite object file or a document). The pathname of a file contains 0 or more directory elements.

The hierarchy of the Directory is described by 3 types of record structures stored within the Directory - *file records*, *directory records* and *thread records*. For every file (leaf node) in the Directory there is one file record. For every directory there is a directory record and a thread record. The format of these records is shown in figure AFS-4. As described later, these records are stored in the Directory's disk data structure (a B*tree, the reason for the null pointer in all 3 records) sorted by *key*, where a key defined as shown in AFS-4.

Every directory has a 16-bit ID. The root's ID is 0. As each new directory is created it is given the next greater ID number. A significant value of this shorthand name for a directory is that it can be renamed by replacing just the thread and directory records for that directory. The (presumably more numerous than 2) records for the contents of that directory are unaffected.

An example hierarchy of directories and files that might be on a volume is shown in figure AFS-5. Below the diagram are all the record entries (type shown first, uninteresting components not shown) that would appear inside the Directory for such a volume in the order they would appear - sorted by key. To find the entry for Source>Text>Memo is performed by looking up <0>Source yielding <9>; looking up <9>Text yielding <12>; finally looking up <12>Memo yielding that files attributes and location information.

Enumeration of Source>* is performed by looking up <0>Source yielding <9>; then looking up <9>null and enumerating entries until key.pID ≠ <9>.

The directory and file records facilitate lookups - proceeding left to right through pathnames or proceeding down the tree. The thread record allows proceeding up the tree. For example, given the key <12>Memo, it is not possible to easily generate its pathname Source>Text>Memo without thread records. But with thread records proceed as follows: lookup <12>null yielding <9> and Text; lookup <9>null yielding <0> and Source; lookup <0>null yielding <-1> and null ==> done.

[HMMM - I think we can simplify things by asserting that the root always has ID=0 and not even have a thread entry for it. Any holes in that logic?]

Internal Structure of the Directory - The Directory's file, directory and thread records on stored and kept sorted in a B*tree. (Described in Knuth, Vol. 3, pp. 471-479 under "Multiway Trees" attached.) A B*tree is a multiway or m-ary tree that is perfectly balanced - all leaf nodes are at the same level (depth) in the tree.

AFS implements this structure on disk in the form of 2048 bytes *nodes*. These nodes are all in the DIR section of the disk shown in figure AFS-1. The first node in the DIR area is the root of the tree. It points to the nodes which are its children, etc. The unused nodes in the DIR area are linked together on a free list. When the free list is exhausted, the DIR area is grown by moving the file(s) that follow it.

Each node conceptually consists of n pointers and n-1 keys (P0, K1,P1, . . . Kj, Pj). For ease of implementation, the AFS Directory nodes contain an additional (redundant) key - Kj+1. The detailed structure of a node is shown in figure AFS-6.

All nodes in the Directory's B*tree are in this format. All non-leaf nodes are index nodes and contain only index records. All leaf nodes contain only leaf records - the file records, directory records and thread records described above.

When the Directory is edited (insert/delete of file, directory, thread records) a node may become full or empty. As described in Knuth, the implementation will keep nodes horizontally balanced. That is, if a record is inserted and it "belongs" in a full node, the adjacent sibling nodes are examined for free space and records are rotated if possible before a node is split and another node is inserted in the tree.

Details of File Records and File Structure - Figure AFS-7 shows the detailed contents of a file record in the Directory as well as the file's data and resource forks. File attributes are stored in the Directory and in the leader page of the data fork. The attributes in the Directory are the "popular" ones that must be quickly obtainable, the ones in the leader page are slower to obtain. For example, the ones in the file record of the Directory are readily available during enumeration - these are the ones required by the Finder, for example, in displaying a window. The attributes in the leader page are those required at file open/close or launch time.

The leader page is put on the data fork (requiring a data fork for all files) because there are expected to be relatively few files with no data fork (ie, more MacWrite files (data and resource) or Multiplan files (data only) than the actual MacWrite code file or Multiplan code file (resource only).

The first of the extents in the file record describes the (start of) the data fork since it always exists. The remaining 89 extents describe the remainder of the data and resource forks in order, with extents for the two forks intermixed and discriminated by the high-order bit. This guarantees that a file can grow to 180Kbytes even under worst-case conditions of fragmentation.

Flags - The Finder flags are as described in the FM documentation. "scav. hole" means that during a scavenge not all pages of the file could be found and pages containing all 0 were inserted.

NOTE: It was asserted earlier that the SAI information could be reconstructed by enumerating the Directory. This is true for entries where "extents overflow" is false. When fragmentation has caused the number of extents to exceed 6, the header page must also be read.

"User" attributes - When used in the File Server, the AFS may have clients that use foreign file systems (MS/DOS, Unix) are have additional attributes (author, last read date, etc.). This design does not explicitly allow for them for the following reasons:
* if the foreign file system is Mac compatible (or a Mac application with additional attribute requirements), then there is more than enough functionality in the resource fork to store this information (by using the user attributes field or defining new resource types);
* if the foreign system is unaware of Mac conventions, then it undoubtedly will not have resource forks at all. In using the AFS through our filing protocols, it can merely read "resource fork" as "unlimited additional attributes fork" and store any additional attributes that way.
[Anyone buy that argument?]

Scavenging - The name and extent information in file records can be recreated from the leader page and file tags respectively. The other information is presumed expendable (true for Finder-type?). The name and extent information in the leader page can be recreated from the file record and file tags respectively. The other information is presumed expendable (true for Finder-creator and password?). Other pages in the data fork and all of the resource fork is not recreatable.

The directory and thread records can be used to recreate one another. This presumes that the two entries for any given directory will be "far apart" (in different nodes of the B*tree) and not both be lost.

The index nodes of the B*tree can be recreated from the leaf nodes.

Interesting events in the life of an AFS

Volume Initialization - All blocks on the volume are written with 0 as the file number in the tags. The volume, space allocation and directory information is set to initial values. Startup information is handled outside AFS although the tag information is set correctly.

File Create/Delete/Allocate/Extend - The Volume Information is updated as appropriate. The SAI is updated as is extent information in the directory (Extend/Allocate). The Directory gains/loses an entry on Create/Delete. The above changes need not be flushed for every event, depending on the particular implementation (however, if writes are deferred, the implementation must be able to restore consistency after abnormal shutdown).

On Allocate/Extend all blocks added to the file are written with correct tag information (is this necessary?).

Delete - The tags of the first pages of both the data and resource forks are set to file number=0. This is necessary for scavenging so that deleted files don't reappear.

Squish

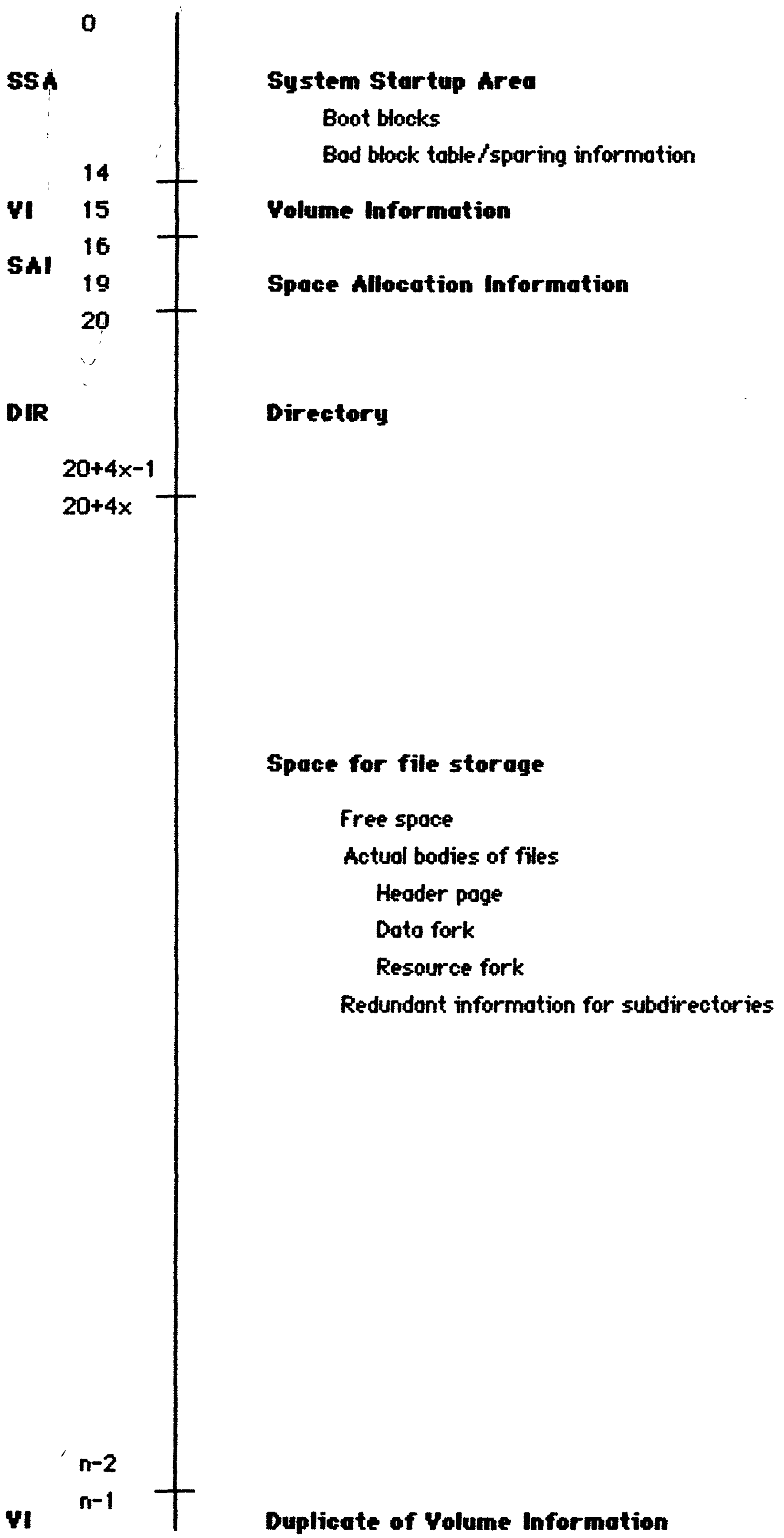
The goal of the Squish algorithm is that all files have a single extent for each fork (be contiguous) and that all free allocation blocks be contiguous.

Squish has two distinct modes - the first enumerates the Directory doing a best effort job making each file contiguous, the second moves files as far toward the "front" of the volume as possible making all free space contiguous. If the free space on the volume exceeds the size of the largest file, successive applications of Squish will achieve perfection - no fragmentation. Smarter versions of Squish can achieve perfection with any non-zero amount of free space.

Note that making free space contiguous can be time consuming - it is driven either by a traversal of the SAI information requiring lookups in the Directory based on file number or by repeated traversals of the Directory looking for extents preceded by free space. It is possible that applying these two phases during the same enumeration of the Directory would help. It would also be nice for Squish to be suspendable so that it could be a background task. More design ergs needed here.

ISSUES

1. Does it make sense to duplicate the VI?
2. Should the space for SAI be a function of volume size as with the Directory?
3. Is statically allocating space for the Directory (say .75% of the volume) OK?
4. Do we need better facilities for more "user" attributes?
5. Is 2048 the right allocation unit size?
6. Do we need to think further now about expansion volumes?
7. Is the scavenging plan for directory/thread records OK?
8. Need to worry about file number overflow? Directory ID overflow?
9. OK to only store file number in tags?



System Startup Area

Boot blocks
Bad block table/sparing information

Volume Information

Space Allocation Information

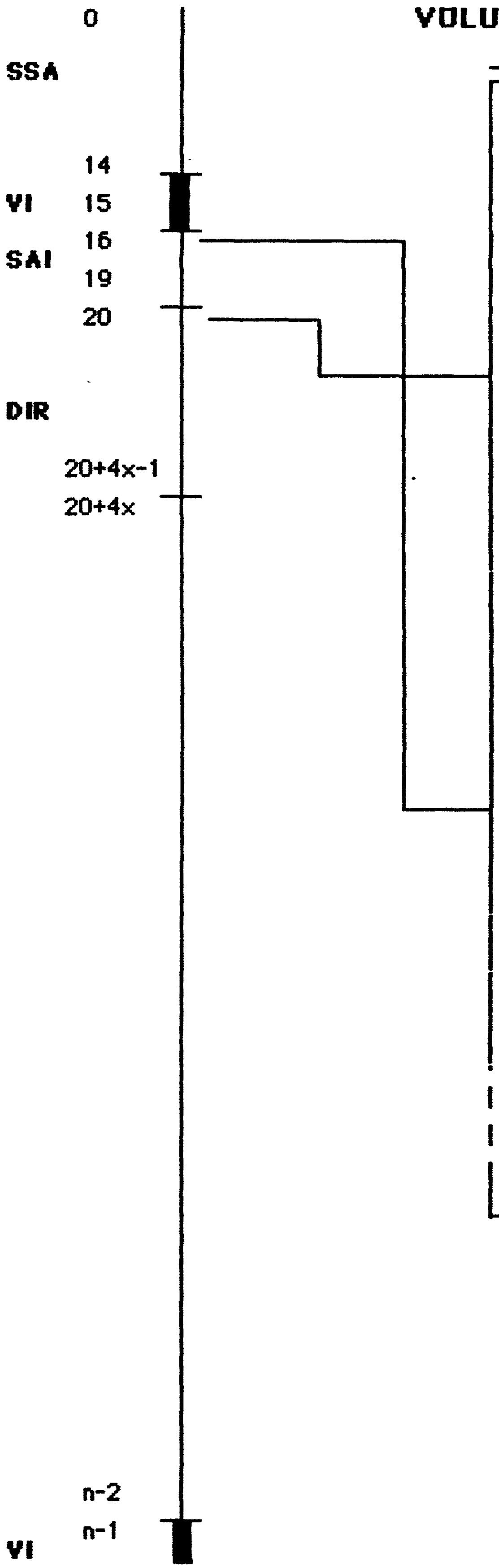
Directory

Space for file storage

- Free space
- Actual bodies of files
  - Header page
  - Data fork
  - Resource fork
- Redundant information for subdirectories

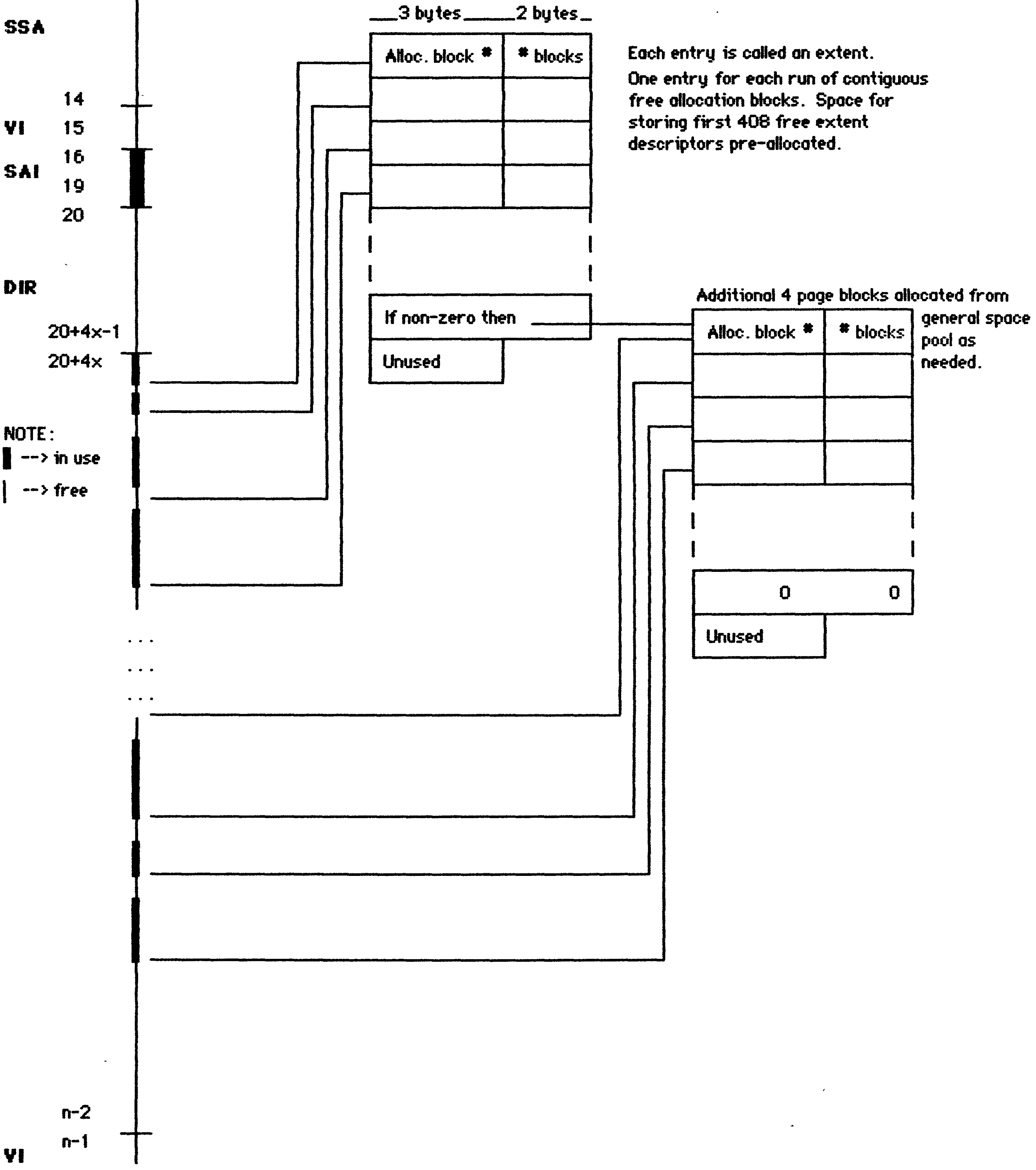
Duplicate of Volume Information

VOLUME INFORMATION



2 bytes	
drSigWord	AFS signature always $DOD1
drCrDate	volume initialization date
. . .	
drLsBkUp	date of last backup
. . .	
drAtrb	b0=1 --> volume open (scav.); b7=1 --> vol. locked by HW;
drNmFls	number of files on vol. b15=1 --> vol. locked by SW
drDirSt	starting block number of Directory (also root node)
drBlLen	number of blocks in Directory area (4x)
NEW	block number of first free Directory node (node = 4 blocks)
NEW	next directory ID to use
NEW	pointer to first node of directory backup info
. . .	
drNmAlBlks	# allocation blocks on volume (n/4)
. . .	
drAlBlkSiz	allocation block size in bytes (2048)
. . .	
drClpSiz	minimum # bytes to allocate (AFS value TBD)
. . .	
drAlBlSt	block # of start of SAI info
NEW	number of blocks in primary SAI info
drNxtFNum	file number for next file to be created
. . .	
drFreeBks	number of allocation blocks free on volume
. . .	
drVn	volume name
28 bytes	
NEW	count of associated expansion volumes - always 0 for now - for future expansion - would be followed by volume name and signature for each expansion volume

SPACE ALLOCATION INFORMATION



Each entry is called an extent.

One entry for each run of contiguous free allocation blocks. Space for storing first 408 free extent descriptors pre-allocated.

Additional 4 page blocks allocated from general space pool as needed.

KEY

LEN	ParentID	Name	00
0	1 2	3	35

DIRECTORY RECORD

Key (Name=directory name)
. . .
. . .
Type=directory
ID of directory
Date created
. . .
[redacted]

44
~~48~~ bytes

THREAD RECORD

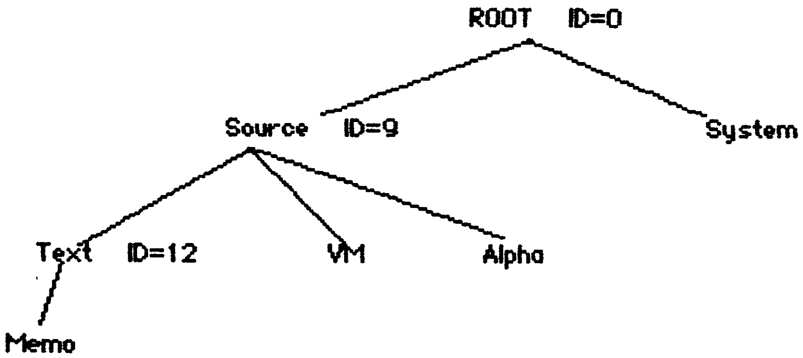
Key (Name= null name)
. . .
. . .
Type=thread
parent ID
myName
. . .
. . .
[redacted]

68
~~72~~ bytes

FILE RECORD

Key (name=file name)
. . .
. . . ← Type=File
File attributes
. . .
. . .
Extent entries describing location of data and resource forks and extended attributes
[redacted]

See subsequent diagram for details



Thread
Key=<0>null
pID=-1
myN=null

Dir
Key=<0>Source
id=9

File
Key=<0>System

Thread
Key=<9>null
pID=0
myN=Source

File
Key=<9>Alpha

Dir
Key=<9>Text
~~pID=9~~
ID=12

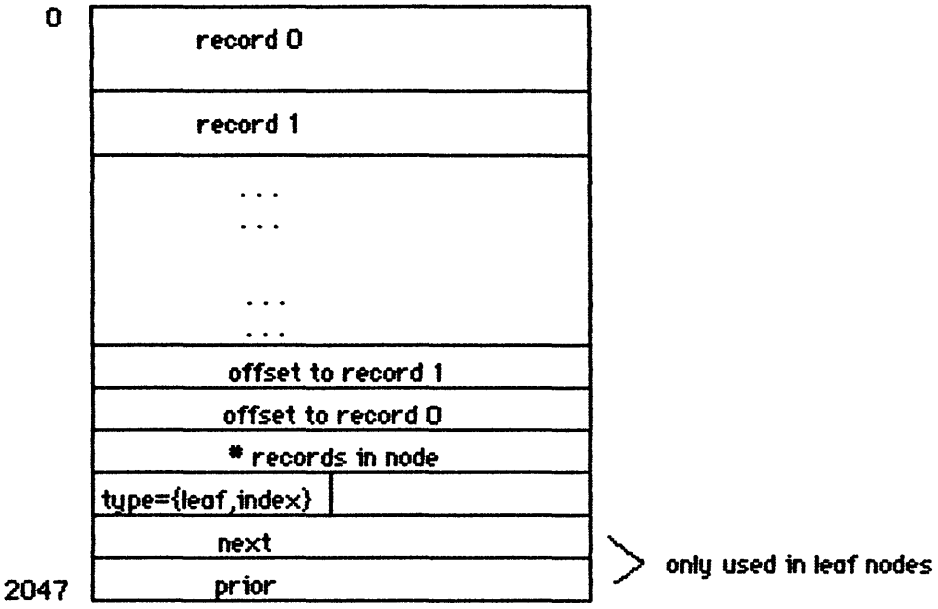
File
Key=<9>VM

Thread
Key=<12>null
pID=9
myN=Text

File
Key=<12>Memo

FIGURE AFS-5

A Node in the B*tree used to store Directory records



disk page

Format for all records in Index Nodes

16-bit pointer to a node	36-byte key

A typical B*tree

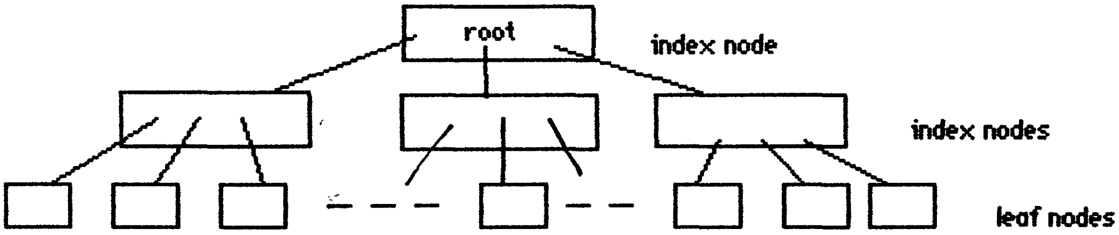
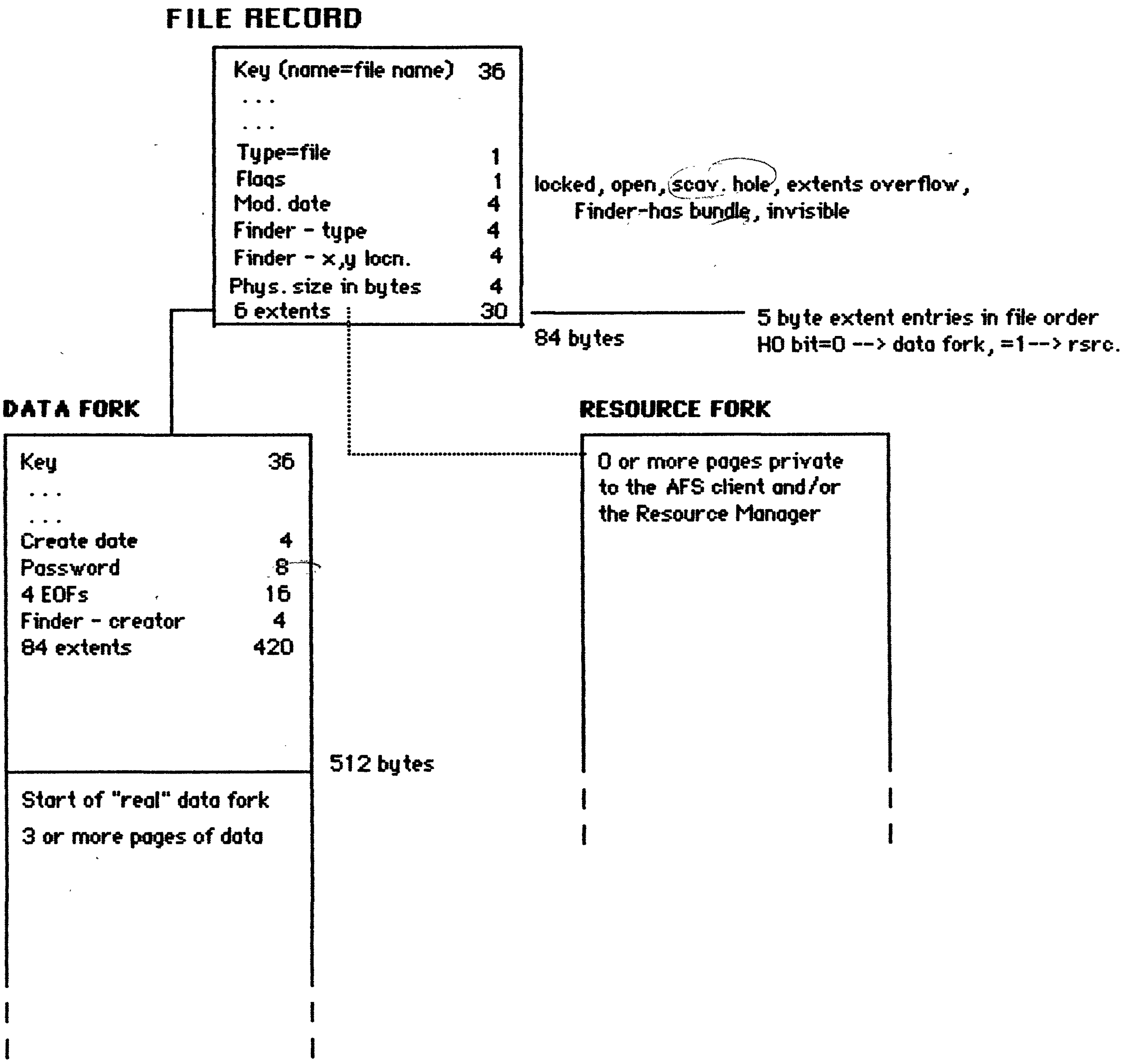


Figure AFS-6



FILE RECORD

Key (name=file name)	36
. . .	
. . .	
Type=file	1
Flags	1
Mod. date	4
Finder - type	4
Finder - x,y locn.	4
Phys. size in bytes	4
6 extents	30

locked, open, scav. hole, extents overflow,
Finder-has bundle, invisible

84 bytes

5 byte extent entries in file order
HO bit=0 --> data fork, =1--> rsrc.

DATA FORK

Key	36
. . .	
. . .	
Create date	4
Password	8
4 EOFs	16
Finder - creator	4
84 extents	420

512 bytes

Start of "real" data fork

3 or more pages of data

RESOURCE FORK

0 or more pages private to the AFS client and/or the Resource Manager

Figure AFS-7