Inter-Office Memorandum

DRAFT - DRAFT - DRAFT - DRAFT

To	Pilot group	Date	October 17, 1977
From	Paul McJones, Dave Redell	Location	Palo Alto
Subject	Volume File Map implementation	Organization	SDD/SD

XEROX Archive: Pilot

Filed on: [Ifs]<McJones>VolFilMapImpl.memo

The per-volume, self-contained mapping of locally resident files into intra-volume storage addresses is discussed. After a brief motivation, several possible data structures are presented, with our current choices indicated.

Introduction

A *file* is the basic information storage object in Pilot [PilotFS]. It has a name, called a *fileID*, and a state, which is a possibly *immutable* sequence of zero or more 512-byte pages. A file must physically reside on some one *volume* (except that copies of immutable files may exist on several volumes). A system element running Pilot will have zero or more directly attached volumes, and may also be connected to a Xerox Wire, allowing access to files on volumes attached to file server system elements. (Pilot requires at least one attached volume or Xerox Wire connection.) Thus algorithms and data structures must exist to find the *network location* and volume of files specified as operands of Pilot operations.

Since fileIDs are intended to be unique across all OIS system elements, one might require that there exist a mapping giving the "address" (network location, volume, and perhaps volume page number) of every fileID. We reject this as unnecessary, undesirable, and probably unimplementable. Instead, every system element will be able to find "local" files, and will solicit the help of a "clearinghouse" to find members of some set of remote files. Here we are concerned with the management of local files.

It has been proposed that each system element maintain a single System File Inventory which would map fileIDs into (volume, volume page number)'s for all locally attached volumes [PilotC&F]. We feel instead that the data structure supporting this mapping should be distributed across the volumes involved. Thus each volume contains a *volume file map,* whose primary purpose is to map fileID into volume page number (physical storage address) for each file residing on the volume. Certainly to deserve the classification dismountable a volume should have a self-contained map. Even when a system element has several nondismountable volumes we prefer to maintain separate, self-contained mappings. This simplifies the hardware reconfigurations which belie the classification "nondismountable".

Assumptions

The set of physical devices which have been proposed for Pilot file storage span a wide range of capacities, from 10^5 bytes (floppy) to 10^8 bytes (Trident). This fact, together with the range of uses to which the associated system element will be put (including single user workstation, file server, RIS-ROS buffer) makes it hard to set down definite design parameters such as maximum number of files, average/maximum time to access a file, etc.

Nevertheless, we must try. In the case of the rumored 7 megabyte Shugart work station disk, a population of around 500 files might be expected (one check found a perhaps typical Alto nonprogrammer disk with 2 megabytes in use to contain 150 files). A floppy diskette might be expected to have substantially fewer files, especially if IBM formatting is used: OS/6 diskettes are limited to 32 files. In the other direction, it would be desirable in a file server application with a large disk to allow almost as many files as there are disk pages (i.e. hundreds of thousands).

It is not necessarily advisable to use the same map structure for the whole range of disks. Especially at the bottom end, floppies become candidates for a simpler, less capacious volume file map to the extent they are used in an essentially serial "super mag card" mode (as they were originally designed to be used).

[Available real memory?]

[Access time?]

Some possible representations

It is thus necessary to design an associative structure capable of handling hundreds to hundreds of thousands of records, where a record has a *key* of about 8 bytes (fileID) and a *value* of a few to a few tens of bytes (address and some other attributes of the file). The two standard techniques are B-trees and hash tables.

We list the main advantages and disadvantages of each.

<u>B-tree</u>

grows smoothly over wide range (unlike hash table, which requires preallocated contiguous range of disk pages and rehashing when table gets too full)

supports several hundred files with single disk access per lookup

can enumerate contents sorted by fileID (useful for a "scavenger" which compares with directory, or for bulk update of clearinghouse database: can find all fileIDs with "foreign" volumeID part)

guaranteed upper bound on access time (whereas none with hash table)

<u>Hash table</u>

average number of disk accesses per lookup can be kept less than say 1.1 (see [Knuth, page 535])

requires less real memory than B-tree with resident root (provided disk address of hash bucket is <u>computable</u> from key)

Weighing these various issues, we have decided to use a B-tree.

A look at the B-tree approach

The B-tree will consist of several levels of *B-tree pages* and a level of *file descriptor pages.* A B-tree page contains an alternating sequence of keys and pointers, beginning and ending with pointers. Here a key is a fileID, while a pointer will probably be a volume page number. Since the files on a given volume are likely to be clustered in the very large fileID space, it seems plausible that *front compression* of the keys will allow a useful reduction in the overall B-tree height (by decreasing the number of leaf pages to be pointed to). Storing each key as the trailing bytes that differ from the previous key, together with a 1-byte count of the number of such bytes, results in the following statistics on key length (which include the count byte):

2 bytes	min
9	max
3-4	avg

[Note that to decode a key with this compression technique requires the scanning of all the preceding keys. This doesn't seem unreasonable.]

To calculate the number of entries per page, note that the first pointer will have a full key, the last pointer will have no key, and the other pointers will have a compressed key. Using 2 bytes for a pointer, we can handle volumes with up to 33 megabytes. Thus the number of entries (actually pointers) per page is as follows:

43 ptrs/page	min (6 + 41x6 + 1 = 253)
126	max (6 + 124x2 + 1 = 255)
85	avg (6 + 83x3 + 1 = 256)

[These number assume each (key, pointer) pair starts on a word boundary; using separate arrays for length, key, and pointer would improve packing.]

File descriptors

The leaves of the B-tree actually contain the file descriptions. The makeup of a file descriptor will be something like:

8 bytes	fileID
8	dates of last read, write
2	size
2	immutable, temporary (?), map depth (3 bits)
24	extended attributes (2 bytes each)
20	page group descriptors (see below) (4 bytes each)
64	total

Thus 8 file descriptors fit in a leaf page.

[We should examine whether or not it would be worthwhile to go to a larger pagesize (e.g. 512 words) to reduce the number of pointers in the higher levels of tree.]

[Are 12 words of extended attributes enough? Usually enough?]

File page map

There must be a way to find the i^{th} page of a file. We expect some files (e.g. code, RIS-ROS buffers) to be allocated totally contiguously, and hope most files will be allocated in a few *extents*, or runs of contiguous pages. Therefore we propose to describe a file with a sequence of *page group descriptors*, each of which describes an extent. The number of page group descriptors needed for a given file depends on the fragmentation of the volume; it lies in the interval [1..file page count]. A page group descriptor consists of:

2 bytes	file page number
2	volume page number

(The size of a page group is implicit in the start of the next page group, or, for the last page group, the size of the file.) It seems reasonable that most files will require a half-dozen or fewer page groups, so we plan to incorporate about that many in the file descriptor. For files with more page groups, we will use a B-tree. Each internal or leaf page contains 128 descriptors, so:

1 level (root only) gives 5 groups
2 level (root + leaves) gives 640 groups
3 levels (root + 1 + leaves) gives 81,920 groups (> #pages on disk)

Capacity of the volume file map

Suppose we have a B-tree with no more than 5 pointers in the root, 1 level of interior B-tree pages, and file descriptor pages at the next level. Using the "average" numbers leads to a total of 3400 file descriptors. The reason for limiting the size of the root is so it can be permanently resident in real memory, thus making the cost of a random fileID lookup 2 disk accesses.

With a larger resident root of say 100 words, a two level tree could provide single disk reference access to 264 file descriptors on average. The problem with this is growth: if a 265th file is added, a third level must be added to the tree and most of the hundred words of root are wasted.

[Are 5 page group descriptors enough sufficiently often?]

Note: when keys are added to a B-tree in sorted order, as will mostly be the case with fileIDs, all but the rightmost page at any level is only half full. This could be improved by biasing the splitting algorithm.

Caching

If, as we expect to be the case, there is locality of reference in the accesses to the volume file map, some time can be saved by maintaining a cache of recently referenced key-value pairs. The B-tree approach, with its best-case 2 disk accesses, is an immediate candidate for caching, although hashing too would probably profit. With either approach it is important to cache individual file descriptors rather than whole pages of the map (B-tree, hash table) unless for example frequent sequential access to many files in fileID order are expected (and the B-tree approach is taken).

Remaining issues

How shall space on a volume be allocated to files?

To what extent can/should the volume file map (and necessary allocation structures) be made to look like a file?

References

[Knuth] *The Art of Computer Programming, Volume 3; Searching and Sorting* by D. E. Knuth

[PilotC&F] *Pilot: The OIS Control Program; Concepts and Facilities,* August 1976

[PilotFS] *Pilot: The OIS Control Program; Functional Specification,* September 1977