To: Distribution

From: M. D. MacLaren

Date: 02/11/76

Subject: The Implementation of Indexed Files in Multics

This note summarizes the technical properties of indexed files in the Multics storage system as implemented in the I/O module vfile_. It also mentions some possible improvements under consideration for future implementation. For detailed information on the storage system, the I/O system, and vfile_, see the publications:

Multics Programmers' Manual, Reference Guide, Honeywell Order Number AG91, and
Multics Programmers' Manual, Subroutines, Honeywell Order Number AG93.

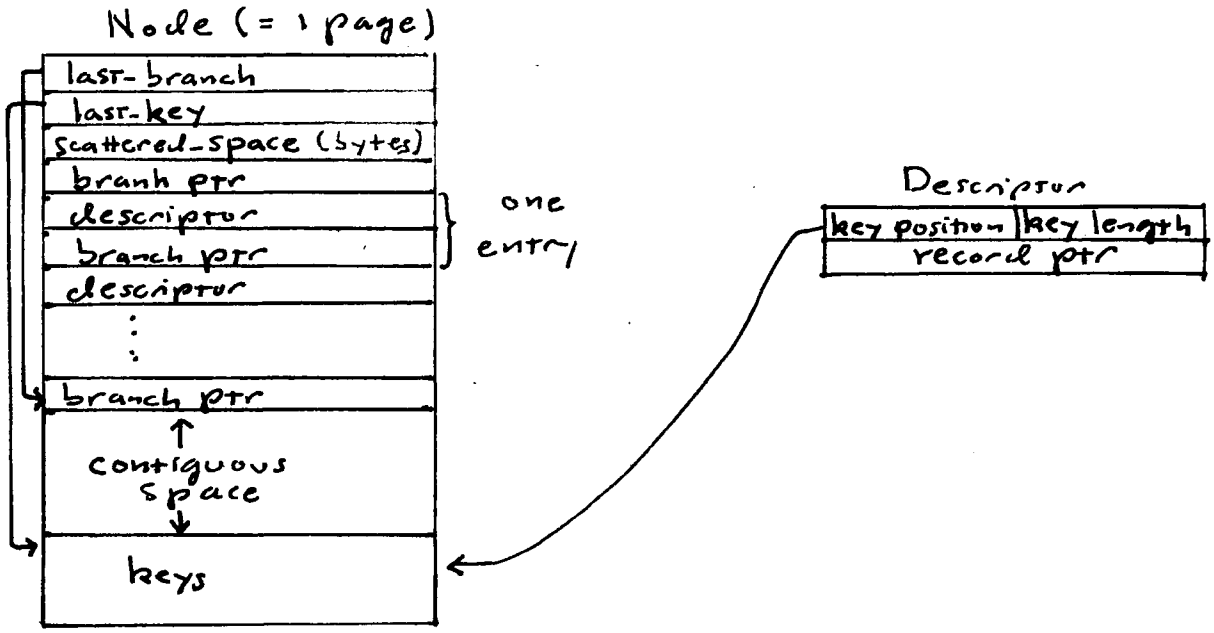
An indexed file is kept as a multi-segment file with one or more index segments and one or more distinct record segments. Each segment may contain up to 256bk 36-bit words. Because the file is in the virtual memory, the implementation of vfile_ does not involve explicit I/O requests (I/O is done by the system's page control). However, all use of vfile_ is through a device independent I/O interface with operations such as seek_key (locates a record), read_record, delete_record, etc.

Space for records is managed dynamically as records are written, rewritten, and deleted. A chained list of free blocks is kept, and allocations is by first fit with a roving pointer. Merging of adjacent free blocks is done with boundary tags (Knuth, vol. 1, p 442, Algorithm C). The space overhead is one word per allocated record. The minimum size of an allocated block is currently fixed at eight words. The end of a segment is treated specially so that the last non-zero word of the segment immediately follows the last allocated record.

To date we have no evidence that searching for a free block is a performance problem for anyone (This is, of course, very dependent on the particular application.) However as a result of some simulation studies, we are considering using a separate free list for each range of block sizes [2**m, 2**(m+1)-1], each list with its own roving pointer. This scheme is now used for PL/I areas in Multics and for general system dynamic allocation. Given a request for a block of size b, the first list searched is

the one such that b is in [2**m, 2**(m+1)-1].

The index is kept as a B-tree. (R. Bayer and E. McCreight, Acta Informatica 1, pp. 173-189, 1972; and Knuth, vol. 3, Sec 6.2.4). Each node occupies one page (1024 words). Keys are variable length 8≤length≤256 characters (9-bit). For consistency with the PL/I (and Multics) rules for character string comparison, trailing blanks are ignored. The layout of a node is as follows:



Descriptors are two words, each key is a string of from 8 to 256 9-bit bytes, other items are one word each. The record and branch pointers are actually number pairs (component segment number in file, offset in segment). The key position and key length (each a half-word) locate the key string within the node. The variables last_branch and last_key together define the free space shown in the figure. The variable scattered_space gives the scattered free space available in the keys section (resulting from deletion of entries). The programs for insertion and deletion of an entry (branch, descriptor, and key) are roughly as follows:

Deletion

add size of entry to total available space
If total available space > 1/2 page, use underflow procedure
Else compact the array of branches/descriptors, setting last_branch = last_branch -1, and add size of key to scattered space.

Insertion

If size of entry <= contiguous free space, do a simple insertion
Else if size of entry <= total available space, compact the keys section and then do a simple insertion
Else use the overflow procedure.

The overflow procedure splits the node only if neither the left or right brother node has sufficient space available to correct the overflow by shifting some entries to the brother. The number of entries shifted is chosen to make the space used in each node as close to equal as possible. The underflow procedure is the usual one for B-trees. The node is balanced (by shifting entries) with its right brother if it has a right brother; otherwise the left brother is used.

Variable length keys introduce an effect not present with fixed length keys. Shifting entries between brother nodes (which also involves one entry in the parent) may cause the parent to overflow or underflow. Thus, it is possible for an underflow to cause the parent node to overflow! Fortunately, this does not further complicate the program.

The I/O system distinguishes a special case of "keyed sequential output" for file creation or extension. For vfile_, this means the records are output in key order, i.e. are always appended to the file. In this mode it does not shift entries on overflow of nodes. Instead it splits with only one entry in the right half. This means that nodes on the right edge may contain only one entry, but all other nodes are proper and are almost full. Writing a file in the normal mode but with records in key order also results in very full nodes but takes much longer because of repeated shifts to balance the same pair of nodes.

It may happen that, while vfile_ is modifying a file, its execution is interrupted and not resumed (e.g. the system may crash). This may leave the file in a state where new operations on the file cannot be performed, e.g. a node may have been split but the new entry not yet made in its parent node. The program vfile_ has been coded so that the next time the file is used, the interrupted operation is automatically completed.

Actually, the whole program for the operation is reexecuted but with a flag set to indicate that it is reexecuting. This causes certain nonrepeatable blocks of code to be skipped until the point of interrupt is reached. A counter is used to determine that point. This method seems fairly easy to use (given a well structured program) and the overhead is moderate; in our case, less than 10% cpu time and less that 1-1/3 pages of space in the file. One of the pages is used for compacting a node by copying, which is the fastest way to do it in Multics.

The following are under consideration for future works:

1. Multiple keys. A popular feature for keyed files. The keys could be stored with the records as well as in the index so that all index entries could be located from any one entry.

2. Duplicate keys. Another popular feature. This would allow the creation of several index entries (i.e. records) with the same key. A seek would find the first one, subsequent ones would be obtained sequentially.

3. Individual record locks. For synchronizing parallel update operations on a file.

4. Omitting the branch pointers in leaf nodes. To save space and hence fit more entries in a given size index. (All branch pointers in a leaf are null.)

5. Leaving the record pointers out of non-leaf nodes. (at the cost of extra information in the leafs, where the record pointers would be placed). The aim of this is to permit more entries in non-leaf nodes, hence, a smaller tree height.

6. Special case treatment of zero length and/or very short records and of short fixed length keys.

7. A special form of the file in which branch pointers are implicit (the locations of all sons are predetermined). Also, the record ptrs might be in a separate array with their locations computed in analogy with the node locations. This appears to be the most compact imaginable form of B-tree. Apparently, it can be implemented so as to gracefully decay into the normal form as random insertions and deletions are made.