Implementation of an Unrestricted File Organization for Micro-PL/CS

James Archer, Jr.

TR79-367

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by a grant from the NSF MCS77-08198.

Implementation of an Unrestricted File Organization for Micro-PL/CS

James Archer, Jr.
Department of Computer Science
Cornell University

Micro-PL/CS is a version of PL/CS developed for a single-user, interactive environment. A file system extension makes PL/CS self-sufficient for standalone file processing and secondary storage management. The basis of the file system extension is the Unrestricted File Organization which provides a free mixture of sequential, indexed and random file operations. The structure, operation, and system-interface procedures of the UFO are presented and explained. The Micro-PL/CS file extension implementation is then sketched in terms of the UFO primitives.

PL/CS is a disciplined instructional subset of PL/I originally implemented as a batch error-correcting compiler based on PL/CS [5,4,6]. Micro-PL/CS is a new system developed to be used in a single-user, interactive environment. The basic system configuration consists of a processor, a high-speed CRT terminal, and random-access secondary storage. Micro-PL/CS is implemented as a syntax-directed synthesizer which actively assists the user in the structured development of correct programs [10]. Interactive development facilities allow freely intermingled program development, execution, and "immediate mode" operations.

The file system extension (FSE) augments the facilities of Micro-PL/CS in both instructional and personal computing environments while making the system self-sufficient. The FSE presupposes a very flexible underlying file organization. The purpose of this paper is to describe the Unrestricted File Organization (UFO), its motivation and implementation. The Micro-PL/CS FSE implementation is outlined in terms of the UFO primitives to display its adequecy to the task.

System Overview

Each user has an independent file system -- there is no provision for sharing or communication between users or for the processing of "compatible" media for interchange with other systems. A file system is a collection of files.

1) A file is a named sequence of records. The file-name is an arbitrary-length string of printable characters.

2) A record is a sequence of items. Each record is numbered by its sequential position within its file. The number of a record changes whenever a record is added to or deleted from a lower-numbered position in the file. Optionally, each record can be named by a "key-value". The key-value is a character string.

3) Items are distinct objects in "LIST" format.

The basis for the file system extension to Micro-PL/CS is the Unrestricted File Organization (UFO) underlying it. UFO supports direct-access (by key-value or

record number) and sequential file operations. The following characteristics of UFO files are noteworthy:

1) Some records are keyed, others are not.

2) Records with keys are in key-value order, following the standard rules for character string comparison.

3) Record numbers are sequential.

4) The number of items in a record can vary from record to record; items can be changed or added.

5) Records may be inserted at/deleted from/modified at any position in the file.

Consider the following section of a typical file selected to illustrate the basic characteristics detailed above:

Key	Number	Contents
ace	15	item 1▯ item 2▯ item 3▮
	16	item 1▯ item 2▮
lob	17	item 1▮
serve	18	▮
smash	19	item 1▯ ...▯ item n▮

If a record were written with a key-value of "forehand", it would be inserted as record 17, causing the record with the key "lob" to be numbered 18, etc. Unkeyed records are considered to be subsidiary to the immediately preceding keyed record (if any exists) for the purpose of record placement.

Implementation Criteria

Before describing the UFO implementation, it is necessary to explain what is and is not expected of it. Primarily, UFO is a file system, comparable to VSAM; it is not a database system. VSAM is a flexible, large-scale production file organization whose design has been documented [11,12,9]; UFO is a highly flexible, small-scale, interactive file organization. VSAM is as flexible as possible without unduly sacrificing efficiency; UFO is as efficient as possible without sacrificing flexibility. The two systems bear some resemblance in overall structure and operation. The remainder of this section is intended to explicate the problems which UFO solves and the criteria for choosing among possible methodologies.

The UFO file structure should be quite flexible. UFO files can contain any mixture of keyed and unkeyed records. Unkeyed records are assumed to have the key-value of the preceding keyed record. Records are accessible by record number. An index is required to perform either keyed or random access. The access time for a record should not depend heavily upon the density of keys in the file. The index, including record numbers, must be easily maintainable with respect to item/record

insertion/deletion. It is considered reasonable, however, that records are a useful concept for file access; random access to one thousand single-item records will be faster than random access to the one thousand items stored in a single record.

UFO should be the sole organization for a small system. Although no explicit implementation restrictions force files to remain small, the files of realistic interest will be small. Typical device capacities are in the hundreds of thousands of characters; files can be expected to be generally smaller. A structure to support many hundred-record files well and a few thousand-record files adequately is different from a structure which must be efficient for million-record (or larger) files. On the other hand, since UFO is the only file organization, it cannot be predicated on the assumption that the user will choose it only if he needs its power. Most user files will, in all probability, be used in the simplest sequential manner imaginable, requiring none of the flexibility built into the UFO. UFO must be able to handle simple sequential access patterns efficiently. The assumed environment includes sharply limited secondary storage. The file organization should be able to adopt structures which require additional machine processing to reduce secondary storage requirements. Further, it is assumed that little or no "room for expansion" will be left for update operations. Extensively updated files can be expected to be less time and space efficient than sequentially created ones. Sequential record overflow is not considered adequate.

UFO file operations should be isolated and incremental. The basic unit of data transfer is an item, rather than an entire record (as would be more standard for file systems). Although access support structures must be made to accomodate finding records, each record is not written as a whole. In particular, this means that the system must work well for each of the atomic operations without hampering overall efficiency. The user is able to perform any of the basic file operations in "immediate mode", a particularly isolated form of file access. Ideally, the cumulative effect of a well-defined series of operations to a particular locality of a file should closely resemble what could be expected given foreknowledge of the operations to be performed. This principle does not extend to files created in a completely random order.

Basic File Structure

The UFO assumes an underlying fixed-length sector secondary storage medium. The actual sector length and total device capacity determine the sizes of pointers and other fields, but do not affect the overall design philosophy. Typical sector sizes for currently available devices range between 128 and 512 characters; file capacities of 64K sectors are within reason. The file system expects to be able to read, write, and allocate sectors. It is not conceptually important whether sectors are logically addressed within files or physically for the entire device. A logical number would presuppose a more sophisticated device manager. Whichever addressing method is used, the file structure is prepared to take advantage of consecutively addressed sectors; consecutive sectors are not required, however. Minimization of sector accesses for common operations is a system design goal.

A physical file consists of two logical components: an index and a data area. The data area is maintained in logical sequential order; it is possible to read the data portion of the file sequentially without reference to the index. The index is a B*-tree modified to keep track of record numbers efficiently and handle variable-length keys [8]. Keys are stored entirely in the index. Index blocks correspond directly to sectors; data area records are independent of sector boundaries. The

ability to add or delete items (effectively changing the lengths of existing records), coupled with the implicitly inherited key structure make a "static directory" (as proposed in [7]) undesirable for the current system. Fortunately, the system requirements do not introduce the problems of concurrency and secondary indices which such directories help to alleviate.

Data Area

The data area consists of a series of items separated into records. Conceptually, each of these items is simply a varying character string. Consistent with the "print" representation of a list of items, we will associate item separation with a tab character (shown above as ▯) and record separation with a newline character (shown above as ▮). The actual characters may not be tab and newline, these are logical associations. These two characters are examples of "distinguished" characters, characters which are used to indicate a special function. In order that these characters can occur in data items, an escape sequence is used whereby the escape character (whatever is chosen) causes its immediate successor to be taken literally (including the escape character). This escape process is entirely within the file system and need not concern the user, to whom it is transparent. Efficient character representation dictates that the escape and all other distinguished characters occur only rarely in normal usage. Consider the following section of a file data area:

ab▯cde▮fghi▯▯j▮▮▯▮

There are six items and four records. Wherever it is unambiguous, the tab marking the termination of the last item is dropped. The last record, a single null item, is the only situation in which the abbreviation would be ambiguous. The next to the last record is null, i.e. it contains no items.

Items (and consequently records) are immune to sector boundaries. Items overlap consecutive sectors without any indication; consecutive sectors are assumed to be logically adjacent. Chaining of logically consecutive, but physically separated, sectors is accomplished by a pointer at the end of the sector. The mechanism links sectors; it does not effect the termination of items. The pointer indicates the number of the next logical sector. All data is stored within a sector "left-justified"; any free space in the sector is represented by trailing null characters. Insertion within a sector is accomplished by moving data within the sector (presuming the insertion fits). To minimize the upward (into the index) impact of such shifts, all records starting within a block must be under the same index sector.

Item Representation

The representation depicted so far has assumed that items would be stored as character strings. The chief attraction of the character representation is its universality; not all items are character strings, but they can all be converted to character strings. Unfortunately, the PL/I-produced string representation for numeric data is very inefficient. Further, the precise conversion process must be maintained in order to avoid conversion anomalies caused by writing and reading items with different variable types. Since secondary storage is precious and there will be a considerable number of numeric items in files, a more economical storage method is needed. Two general types of compaction were considered:

1) code individual items in an internal (binary) form, or
2) adopt a general character-compression coding.

The internal format coding would be able to compress numeric and BIT items; character items are already in internal form. The general form of such a coding would be a distinguished character, indicating the type and length of the value, followed by the fixed-length value itself. In the case of BIT items, the value is actually coded in the distinguished character (PL/CS only allows BIT(1)). No trailing item separator is needed for these values. The coded representations must preserve the types of the items, but need not exactly match the internal representation. For instance, many small numeric values could be stored in a single byte rather than at full length. This method accomplishes significant space savings at only moderately increased processing complexity; it is somewhat machine-dependent, but transferability of files in internal format is not a system objective.

More general coding schemes, in the manner of Huffman coding, offer the possibility of space savings for all items, but the possible savings are materially less compelling and much more expensive to acheive. Further, considering the absence of any reasonable model of character frequencies, the likelihood that both upper/lower case text will abound, and the burgeoning number of distinguished characters considered in the foregoing, an enforced, information-theoretic coding scheme based on erroneous assumptions might actually increase the total space requirement. For similar reasons, a CCITT (often called Baudot) coding scheme was also rejected. A scheme for compressing strings of characters was adopted. The processing cost is not high, and the use of the method can be reserved for those cases where space is actually saved. The method is tuned to make blank compression particularly effective, though repetitions of more than four of any other character can be compressed.

Index Area

The file index is a modified B*-tree, each node of which occupies a sector. Three types of information are stored in the index: keys, record numbers, and pointers (to either another index level or to data). The sectors of the file are structured into a strict hierarchy by the index; all of the data records starting in a sector appear in the same index sector. Note that this can cause problems when key lengths approach sector lengths and records are short; this is not considered a common file configuration. Not all data records are indexed. Keyed records are always indexed. Every data sector in which a new record starts has an index entry corresponding to that first record, keyed or not.

All keys are stored in the index as variable-length character strings with explicit terminators. Keys are compacted using a high-order abbreviation technique. Keys which share a common prefix with the previous key begin with distinguished character(s) indicating the length of this common prefix. We will represent the distinguished character for an n-character match as "n". Consider the following sequence of keys, shown in both forms with item separators.

cat▯catastrophe▯catastrophic▯catatonic▯
cat▯3astrophe▯9ic▯4tonic▯

The distinguished character Q is used to indicate a complete key match. Remember

Under "normal" conditions, assuming some locality and consistency of access, the number of sector operations should be small.

Three file management routines are provided:

Create: find a file; creating new if necessary
Access: find an existing file
Destroy: remove a file from the directory

In addition, the directory is accessible as a read-only file containing a record for each file. The directory is readable in the same form as any user file; file-names form the set of keys of the directory record; the index information for the file are the items of the records (its all done with mirrors).

Actual file manipulation is carried out using the following routines. All routines have an FD parameter which is not shown. The routines are intended as a system, not a direct user, interface. As a result, only the primitives required are supported; some "reasonable" logical operations require more than one primitive call. Basic communication is controlled by record number and item number. Items are treated generically for simplicity, even though the coding algorithms discussed above would require type information. Parameter abbreviations are : c (count), d (data item), i (item number), k (key), and r (record number). All parameters are assumed to be passed by reference and are updated as required.

CreateRecord(r, c): Create c new records immediately after the one numbered r; r is updated accordingly.

CreateKey(k, r): Creates a new, keyed record at the appropriate position in the file; returns the position of the record in r. If the key already exists, the new record is not created, but r is set.

LocateKey(k, r): Finds the position in the file corresponding to where the key k would go; returns the position in r; updates k if the passed value is not actually present.

ReadKey(r, k): Returns the key associated with record r in k.

ReadItem(r, i, d): Reads the next item, starting at item i of record r into d; for the item actually read, returns the record in r and one past the number of the item in i.

WriteItem(r, i, d): Writes the value d at item i of record r. If the item exists, it is replaced. Otherwise, the item is written at the end of record r; i is updated to one past the number of the item actually written.

FindItem(r, i): Checks for the existence of item i. If item i does not exist, i is set to the last item that does exist; a non-positive value for i assures that the last item of the record will be found. If record r does not exist, r is set to the last record in the file and i is appropriately updated.

TruncateRecord(r, i): Truncate/expand record r so that it contains exactly i items.

DeleteRecord(r, c): Delete c records starting with record r. None of the records need exist; r is unchanged; c is updated to reflect the number of records actually deleted.

Summary of the Micro-PL/CS File System Extension

Micro-PL/CS file processing statements are GET, PUT and DELETE; there is a set of built-in functions/pseudo-variables to control file operations. A complete summary of the syntax and semantics are given in [1]. Brief descriptions of each the statements, built-in functions and pseudo-variables is given below. Note that assignment to the COUNT, KEY, and REC pseudo-variables can change the file. The functions/pseudo-variables for a particular file are tightly coupled: assigning to any of the pseudo-variables can change the values of any of the other functions for that file. Four functions/pseudo-variables are provided to perform record positioning in the file: REC, REMAIN, KEY, and FIND.

REC(file-name) (function and pseudo-variable)

Returns the current record number (from 1 to the size of the file).

Assignment to the pseudo-variable causes the current record pointer to be set to the numbered record. Values may be assigned which arbitrarily extend the file.

REMAIN(file-name) (function)

Returns the number of records until end-of-file is encountered on the file. A 0 value indicates that the current record is the last record in the file.

KEY(file-name) (function and pseudo-variable)

Returns the character string value of the key for the current record in the file, file-name. If the current record is unkeyed, KEY returns the key of the most closely preceding keyed record, or LOW if no preceding keyed record exists.

Assignment to the pseudo-variable sets the current record pointer to the record with the assigned key-value. Assignment of a key-value that did not previously exist causes a null record with that key to be created.

FIND(file-name, key-value) (function)

Sets the current record pointer at the first record whose key is greater than or equal to key-value and returns the key of the character-string value of the key. If no record exists with key-value greater than or equal to key-value, HIGH is returned and the current file pointer is set to "end of file".

In order to easily process variable numbers of items, two function/pseudo-variables, COUNT and ITEM, are provided.

COUNT(file-name) (function and pseudo-variable)

Returns the number of items in the current record for the file-name.

Assignment to the pseudo-variable changes the number of items in the current record, either truncating or extending the record with null items.

ITEM(file-name) (function and pseudo-variable)

Returns the current item number in the current record of the file. The first value for any record is 1.

Assignment to the pseudo-variable moves the current item pointer to the specified position within the record.

Independent of input form, PL/CS always prints out program statements in a canonical order; that order is reflected by the order of the optional phrases in the prototype statements.

```
PUT [FILE(file-name)] [SKIP[(n)]] ¦LIST(list)      ¦ ;
                                  ¦EDIT(list)(list)¦
```

1. If the SKIP phrase is omitted, items are written starting at the end of the current record, i.e. at REC(file-name), COUNT(file-name)+1.

2. If the SKIP phrase is specified:

   a. If the SKIP count, n, is positive, n records will be written, numbered REC(file-name)+1 ... REC(file-name)+n. Any items transmitted will be to REC(file-name)+n. As a result, the preceding n-1 records will be null.

   b. If the SKIP count is zero, output begins with ITEM(file-name), updating exisiting fields or extending the record as needed.

3. Both the LIST and EDIT phrases can be omitted. This construction can be used to create null records or to truncate or extend the current record.

```
GET   [FILE(file-name)]   [NEXT]   |LIST(list)       | ;
                                   |EDIT(list)(list)|
```

1. If NEXT is present, REC(file-name) is incremented and ITEM(file-name) is set to 1 prior to data transmission.

2. LIST format items are read starting with REC(file-name), ITEM(file-name). Record boundaries are ignored. Null records are ignored when searching for the next item of the data list.

3. EDIT format input is limited to the current item; ITEM(file-name) is incremented by 1.

```
DELETE FILE(file-name) [RECORD[(n)]];
```

1. If RECORD phrase is present, deletes the number of records specified (one is the default) starting with REC(file-name). REC(file-name) remains unchanged, but refers to the first record following those deleted. The number of records to be deleted must be positive.

2. If RECORD phrase is absent, this statement deletes the entire file.

FSE Implementation

The following are examples of the kind of operations required to implement the FSE. To simplify the code, all cases assume that the file referenced has been previously accessed and that statically discernible errors have been detected. The variables, recordnumber, itemnumber, filesize, and key are used as if they were specific to the file being processed. The FD parameter has been dropped from each of the UFO calls.

```
GET FILE(filename) [NEXT] [LIST(data items)];
-----------------------------------------

        IF "NEXT phrase is present"
            THEN DO;
                recordnumber = recordnumber + 1;
                itemnumber = 1;
                END;
```

```
        DO "for all data items, d";
            CALL ReadItem(recordnumber, itemnumber);
            IF recordnumber > filesize
                THEN ERROR;
            END;
```

PUT FILE(filename) [SKIP(n)] [LIST(data items)];

```
        IF "SKIP(n) phrase is present"
            THEN IF (n > 0)
                THEN DO;
                    /** create n records; adjust item and file size */
                        CALL CreateRecord(recordnumber, n);
                        itemnumber = 1;
                        filesize = filesize + n;
                    END;

                ELSE; /* SKIP(0); no action required */

            ELSE DO;
                /** no SKIP phrase, find the end of the record */
                    itemnumber = -1;
                    CALL FindItem(recordnumber, itemnumber);
                END;

        DO "for all data items, d";
            CALL WriteItem(recordnumber, itemnumber, d);
            END;
```

DELETE(filename) RECORD(n);

```
        CALL DeleteRecord(recordnumber, n);
        filesize = filesize - n;
```

COUNT(filename) built-in function

```
        c = -1;
        CALL FindItem(recordnumber, c);
        RETURN(c);
```

COUNT(filename) = n;

```
        CALL TruncateRecord(recordnumber, n);
```

```
FIND(filename, k)                    built-in function
-----------------

        key = k;
        CALL LocateKey(key, recordnumber);
        RETURN(key);


ITEM(filename)                       built-in function
--------------

        RETURN(itemnumber);


ITEM(filename) = n;
------------------

        r = recordnumber;
        tempn = n;
        CALL FindItem(r, tempn);
        IF ((r = recordnumber) & (tempn = n))
            THEN itemnumber = n;
            ELSE ERROR;


KEY(filename)                        built-in function
-------------

         CALL ReadKey(recordnumber, key);
         RETURN(key);

REC(filename) = n;
------------------

        IF (n > filesize)
            THEN DO;
                CALL CreateRecord(filesize, n-filesize);
                filesize = n;
                END;
        recordnumber = n;


REC(filename)                        built-in function
-------------

        RETURN(recordnumber);


REMAIN(filename)                     built-in function
----------------

        RETURN(filesize-n);
```

Summary

The implementation of an Unrestricted File Organization for Micro-PL/CS has been outlined. The system requirements and design criteria have been presented and explained in the context of the Micro-PL/CS File System Extension, which is more fully presented in [1]. The significant characteristics of a storage structure for efficient index and data implementation have been presented, and a set of algorithms sketched for their maintenance. The access method procedures were described and used to sketch the actual implementation of the FSE.

References

1. Archer, J. E. Jr., "A File System Extension to Micro-PL/CS", Technical Report 79-366, Department of Computer Science, Cornell University.

2. Bayer, R. and E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1 (1972), pp 173-189.

3. Brown, M. R., and R. E. Tarjan, "A Representation for Linear Lists with Movable Fingers", Proceedings of the Tenth Annual Symposium on Theory of Computing, May 1978, pp 19-29.

4. Conway, R. W., A Primer on Disciplined Programming, Winthrop Publishing Co., 1978.

5. Conway, R. W. and R. L. Constable, "PL/CS -- A Disciplined Subset of PL/I", Technical Report 76-293, Department of Computer Science, Cornell University.

6. Conway, R. W. and D. Gries, Introduction to Programming, 3rd ed. Winthrop Publishing Co., 1979.

7. Held, G. and M. Stonebraker, "B-trees Re-examined", CACM 21, 2 (1978), pp 139-143.

8. Knuth, D. E., Sorting and Searching, Addison Wesley, 1973, pp 471-479.

9. Marayuma, K. and S. E. Smith, "Analysis of Design Alternatives for Virtual Memory Indexes", CACM 20, 4 (1977), pp 245-254.

10. Teitlebaum, R. T., "The Cornell Program Synthesizer, a Micro-processor Implementation of PL/CS", Technical Report 79-370, Department of Computer Science, Cornell University.

11. Wagner, R. E., "Indexing Design Considerations", IBM Systems Journal, 12, 4 (1973).

12. Wiederhold, G., Database Design, McGraw-Hill, 1977, pp 182-191.