A Unified Local, Global, and Routine Design For Mumps Micro-Computer System

Frank M. Brown

Department of Computer Sciences

The University of Texas at Austin

Austin, Texas

TR 146 May 1980

A UNIFIED LOCAL, GLOBAL, AND ROUTINE DESIGN FOR MUMPS MICRO-COMPUTER SYSTEM

Frank M. Brown
Department of Computer Science
University of Texas at Austin
Austin, Texas 78712

Abstract

This paper describes the basic design of Comp Consultants Standard Mumps system for Tano Corporation's Outpost 11 6800 based micro-computer running under the Flex 2.0 floppy disc operating system with 64k of bytes of random access memory. The Mumps system consists of an executive which includes I/O device handlers an interpreter for Standard Mumps and a floppy disc storage system for global variables based on balanced trees with key compression by indexes and with 32k of random access memory buffers to make up for the slow floopy disc access times.

Comp Consultants Mumps involves some significant departures from previous Mumps implementations. In particular local variables and routine lines are stored in the same manner as global variables. For example the storage key for a routine consists of three parts: The routine name, a label, and a line offset number. Representing local variables and routines as global variables not only increases the available buffer space by eliminating special code to handle local variable storage, routine storage, and program editing; but also increases the utilization of the buffers by keeping in memory what is most often accessed regardless whether it is a local variable, a global variable, or program code. This unified representation of local variables and routines as global variables is described in detail in this paper.

1. Introduction

Our goal was to develop a complete single user stand alone Mumps system for a small Motorola 6800/6809 based micro computer . An example of such a computer is Tano Corporation's Outpost 11 which consists of 64 kilo bytes of Random access memory and two 190 Kilobyte floopy disk drives. The problem of developing a Mumps system for such a machine centered around the very slow approximately 1/3 second access times for disk blocks. Since this time is far too long for accessing a global variable stored in a disk block in a reasonable Mumps system, it is clear that not only would some sort of scheme for temporarily storing variables into Random access memory buffers be needed but also a large number of such buffers would be needed so that the most often used disk blocks would already be in memory when they were to be accessed. Fortunately microcomputers such as the Outpost 11 have relatively large memories for their size most of which is not needed for any particular purpose, and thus a large portion of this memory, 32 Kilobytes in an Outpost 11, can be used as memory buffers for the floopy discs. Thus a solution to the slow floopy disc access time is to use a large number of Random access memory buffers to temporarily store as many as possible of those global variables which are most often accessed.

To achieve this goal we decided to use the "least-recently-used-buffer-replacement scheme" described in [1]. This scheme is essentially to keep a list of all available buffers in memory ordered so that the most recently used buffers are at the start of the list and the least recently used buffers are at the end. Thus when a variable is accessed, and if it is in the same buffer as some other variable which was accessed, the current buffer will usually be found at the beginning of the list. On the other hand, if the variable was not in a buffer the whole list will be examined and the last buffer in the list will be moved to the front of the list, and the disk block containing that variable will be loaded into that buffer.

2. Global Variables

Having decided on a buffering scheme for global variables we now considered the problem of actually representing those variables within disk blocks. We decided to represent Global variables as B-trees in order to give them a logrithmic retrieval and updating time. In this scheme each global variable is represented as a sequence of bytes consisting of six parts.

1. A compression count of 1 byte
2. A Key
3. A Key end mark of one byte
4. An allocation Key of one byte equal to zero
5. A data byte count of one byte
6. The data

The Compression Count is a count of the number of Global indexes that a variable has in common with the alphabetically previous Global, not the number of characters. The reason for this is to facilitate the execution of the Next operation by always being able to return a pointer to an entire index rather than having to physically create it. The Keys are allowed to be any string of ASCII characters. They coalate in the sequence negative numbers including decimals followed by positive numbers including decimals followed by all other strings.

The allocation count will be used in a more significant manner later. The data byte count specifies the number of following characters in the string which is the value of this global variable. The fact that the allocation count is zero indicates that when this variable is set to have a smaller string as its value all the extra space in the value field is collected for use by other variables.

As many global variables as possible are fit into one disk block. When that block becomes filled the data in it is split evenly between two blocks.

3. Local Variables

Having decided on how to handle global variables we then considered the problem of representing local variables. We realized that unlike earlier Mumps multiuser time sharing systems where there are significant differences in local and global variables, that these differences had essentially vanished in our system. For example the fact that global variables are global to all users where as local variables are distinct for each user is superfulous in a single user system. Furthermore the distinction between global variables being stored on a disk and local variables being stored in memory is essentially superfulous in view of the solution of storing as many as possible of the most often accessed global variables into memory buffers. For these reasons we decided to treat local variables as if they were global variables. Thus local variables are stored on the disc and temporarily brought into memory buffers by the same mechanism which accesses global variables since they are more often accessed than globals they will be in the memory buffers most of the time.

Local variables are represented in the same manner as global variables except that they have a non-zero allocation count. This allocation count is used to give each local variable a fixed partition size so that when it is set to value which is a smaller string, the left over space will not always be collected. The decision on when space should be collected is under user control based on two systems parameters 1. The Allocation parameter
and 2. The Threshold parameter.

The threshhold parameter determines when space should be collected and the allocation parameter determines how much space to collect if the allocation count (AC) minus the data byte count (BC) is greater than the threshold parameter (TP)

$$AC - BC > TP$$

then the number of bytes collected is the allocation count minus the sum of the new byte count and the allocation parameter (AP)

$$AC - (BC + AP)$$

By setting the threshold to a reasonably large value e.g. 20 and the allocation count to a small value e.g. 5 the user should be able to avoid much data movement that would otherwise be necessary when local variables were set to new values without using up too much memory.

4. Routines

Having decided how to represent both global variables and local variables we finally considered the problem of representing routines. In earlier multi-user-time sharing systems a user routine along with his local variables was restricted to fit within a certain partition size in memory, usually 4 Kilobytes. When a new routine was accessed it was loaded from the disk into the partition thus destroying the old routine. Thus if the new routine later returned control to the old routine it had to be reloaded from the disk into the partition. This of course was very inefficient if inside a loop a routine merely called another routine, which later returned control to it, because each pass thru the loop would involve up to eight Kilobytes of data being loaded from the disk which would take up to 4/5 of a second in data transfer time alone. Thus again, what is needed is a large number of memory buffers in which to store the most often used routines, so that most calls to routines could avoid any disk accesses.

In order to maximize the use of memory buffers, we decided to use the same buffering system for both variables and routines as this would allow whatever was most often accessed, be it a variable, or a routine to be in the memory buffers. For example, routines will now be brought into a memory buffer when they are accessed and deleted only when that memory buffer is needed to load some other routine or variable. Thus in effect we have designed a virtual memory system for Mumps routines and variables.

The fact that we have a virtual memory system for mumps routines instead of a small set partition size leads to one further problem: Namely, that there is no longer any reason for routines to be of any particular size. Thus for example if the routines are large they may contain many line labels, and in executing a GOTO instruction the mumps interpreter is faced with the problem of preparing an inefficient linear search thru all line labels in order to find the correct one. Note that even in MIIS where the address of the next line label is stored with the previous line label, a linear search is involved.

On the other hand if routines are small there will be many routines and then the Mumps interpreter is forced to preform a similar inefficient linear search over routine names. Clearly, what is needed is a more efficient representation of Mumps programs involving better search techniques, such as the efficient tree representations used with Mumps variables which involve logarithimic time search. Can we think of a better representation? Our answer is yes, and our solution is simply to represent Mumps Routines as global variables. That is, a line of Mumps code is represented as a global variable whose name is the Routine name followed by the preceding label name followed by a number representing a line offset from the preceding label, and whose data consists of the rest of that line. Programs are executed by jumping to the appropriate label and executing lines in order until control is transferred to another line. For example, the following program fragment of a routine R which prints the numbers 1 to 100

```
        SET I=1
START   IF  I=101; GOTO END
        DO  FOO
        SET I=I+1
        GOTO START
END     KILL
        QUIT
FOO     WRITE I
        QUIT
```

is represented as global variables as follows:

```
R("END",1)    =  "KILL"
R("END",2)    =  "QUIT"
R("END",3)    =  "*GOTO FOO,1"

R("FOO",1)    =  "WRITE I"
R("FOO",2)    =  "QUIT"
R("FOO",3)    =  "ERROR-NO SUCH LINE"

R("INIT",1)   =  "SET I=1"
R("INIT",2)   =  "*GOTO START,1"

R("START",1)  =  "IF I=101; GOTO END"
R("START",2)  =  "DO FOO"
R("START",3)  =  "SET I=I+1"
R("START",4)  =  "GOTO START,1"
R("START",5)  =  "*GOTO END,1"
```

To execute this program an access to the first line of the entry point is made which returns "SET I=1". This line is executed and the next line from the entry point is executed which is a jump: *GOTO START,1. Thus the line START,1 is accessed. This line is an IF test which is false for I=1 so the next line is accessed which is a "DO FOO" statement. This line stacks the next line START,3 for later use and FOO,1 is accessed. The WRITE statement in FOO,1 is executed and the next FOO line is accessed resulting in a "QUIT". The QUIT causes the last stacked key, namely START,3, to be accessed. Thus, I is incremented to 2, and START,4 is then accessed. Since START,4 is a GOTO START,1, START,1 is again accessed with I=2. The system continues around this loop in this manner until I=101, at which time the IF statement in line START,1 will eval to true causing the GOTO END statement on that line to be executed.

5. Conclusion

We have designed virtual memory system for Mumps on a small micro computer. This system is based on a unified representation for global variables, local variables, and Routines, namely that everything is represented as a global variable in an m-way branching B-tree. The advantages of this representation are numerous and include the facts that

1. The Mumps system has a virtual memory allowing programs of up to 190 Kilobytes.
2. Whatever is most often accessed be it a global variable, a local variable or a line of Mumps code will be in the memory buffers thus minimizing the number of physical access to disk blocks.
3. No special code is needed to handle local variables thus allowing this memory to be used for buffers.
4. No special code is needed to move routines into memory as they are needed thus allowing the memory space saved to be used for buffers.
5. No special code is needed to write primitives for a Mumps Program editor because the necessary primitives for manipulating a program are already available as the normal global variable B-tree operations.
6. GOTO instructions in Programs with many large routines can be executed efficiently by logarithmic search instead of linear search as in previous Mumps systems.

Besides all the concrete advantages, we believe there is one further more abstract advantage of this Mumps design: namely that if Mumps programmers make full use of its facilities they will be forced to write more esthetically pleasing structured programs consisting of many small functions implemented by DO jumps rather than useing one large linear routine with many GOTO jumps as this will allow the little DO subroutines to be loaded and deleted by the buffering system.

References

1. Kuuth, Donald E. The Art of Computer Programming Volume 31 Sorting and Searching, pages 473-478, Addison-Wesley Publishing Company 1973.