TECHNICAL MEMORANDUM

73

A simple memory manager

J.K. Gibson

Operations Department

July 1983

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen

1. INTRODUCTION

Computer programmes such as numerical weather prediction models often require large areas of main memory. Indeed, many such programmes cannot execute without the aid of an input/output system, with large areas of memory being continually refreshed from data held in on-line scratch space or backing store. To use the available memory efficiently, it is often necessary to re-use specific areas of memory for different purposes, over-writing one set of variables by another. Such practises are potentially dangerous, and must be organised with care. Memory maps must be produced, consulted, and updated as modifications are introduced.

Memory references to large, shared, areas of data become even more complex if a language of the FORTRAN type is used. Often it is necessary to resort to addressing displacements within a single, globally accessible array. Although displacement variables can be given meaningful names, the dimensional structure and direct relevance of well chosen array names and declarations are lost.

The ideas contained in the following paper were conceived in an attempt to provide a software solution to the memory management problem. They have been successfully implemented into the ECMWF operational weather prediction model using Cray Fortran, and could be adapted to any computer language which allows the use of based variables.

2. BASIC AIMS

The basic aims of a simple memory manager of the type under discussion are:

a) to enable the user to ALLOCATE arrays for use as WORK SPACE or LONG TERM STORAGE.

b) to enable LONG TERM STORAGE arrays to be LOCATED, and used.

c) to enable WORK SPACE or LONG TERM STORAGE arrays to be returned when no longer required.

LONG TERM STORAGE is defined as array space required to be accessed by more than one routine. WORK SPACE is defined as array space required only within a single routine.

3. MECHANISM FOR ADDRESSING STORAGE

The memory manager is designed to supply pointers to based variables. In consequence, it is only suitable for use in association with languages which support such features (e.g. PL1, Cray Fortran, etc.) The concept of based variables is gradually becoming accepted as an important feature of high level computer languages, and will probably be included in the next ANSI Fortran standard. Simply stated, it enables the user to define a POINTER variable in association with an array or structure. The assignment of a valid memory address to the POINTER variable then defines the location of the corresponding array or structure in memory. Thus, in Cray Fortran,

```
DIMENSION  A(1000)
POINTER  (IB, B(1000))
IB=LOC(A)
```

has the effect of making array B equivalent to array A.

4. INTERFACE TO THE USER

Various implementations of the ideas contained in this paper will provide varying interfaces to the user. A sufficient interface, for the purpose of illustrating the following sections, would consist of five routines:

```
INILOC (KLENGTH)
ALLOC  (KPOINT, KSPACE, KNAME, KCODE)
ALLOCB (KPOINT, KSPACE, KNAME, KCODE)
LOCATE (KPOINT, KSPACE, KNAME, KCODE)
UNLOC  (KNAME, KCODE)
```

where

KLENGTH is the number of words of memory to be managed.

KPOINT is used to return a POINTER value to the user.

KSPACE is the array size in words.

KNAME is the array name (LONG TERM STORAGE) or the name of the user routine (WORK SPACE)

KCODE is a code number in the range 1 to 98 (LONG TERM STORAGE) or 99 (WORK SPACE).

INILOC is used to initialise the memory manager, and is called once only. ALLOC is used to allocate array space. Arrays are identified by the memory manager by means of a name, KNAME, and a code, KCODE. It is thus possible to have several arrays with the same name, provided a different code is associated with each. This device has been found particularly useful where different spatial representations of a variable are required to be stored simultaneously. Thus, for instance, a code of 3 could be assigned to all variables in grid point space. A single name could be used for a variable stored in both grid point and spectral space, the memory manager being capable of distinguishing the required array space according to the supplied value of the code. If work space is required, ALLOC is called using the name of the calling routine as KNAME, and a code of 99.

ALLOCB is used to allocate array space within the managed memory area in such a way that repeated calls to ALLOCB will result in a set of arrays occupying a contiguous area of memory.

LOCATE is used to locate an array previously allocated by ALLOC or ALLOCB. Only LONG TERM arrays may be located. WORK SPACE is only allocated, as it is only addressable from the allocating routine.

UNLOC releases previously allocated space. Released space may be re-allocated by subsequent calls to ALLOC or ALLOCB. LONG TERM arrays are released singly, whereas a single call to UNLOC releases all areas of WORK SPACE corresponding to KNAME.

5. MANAGEMENT METHOD

5.1 Management tables

Management information is maintained in stack-like tabular form. For each allocated area, table entries record:

a) the name (KNAME)

b) the address (KPOINT)

c) the code (KCODE) (long term only)

d) the length (KSPACE)

Two sets of tables are maintained - one for LONG TERM space, the other for WORK SPACE. A stack-pointer is associated with each table, indicating the number of valid entries in each table at any time.

5.2 Allocation strategy - Work Space

Work space is always allocated from the next area of available space to that of the last table entry. No attempt is made to re-use released areas corresponding to entries within the table. As routines requiring work space are rarely deeply nested, this simple strategy is both sufficient and efficient.

5.3 Allocation strategy - Long Term Space

Efficient use of memory dictates that long term storage must re-use released areas without resulting in memory fragmentation. A simple but effective strategy is:

a) re-use areas only if they are exactly the correct size.

b) do not combine adjacent released areas into larger areas.

c) if no suitable area can be re-used (i.e. no exact fit) then add a new table entry and allocate from the residual area of available space.

The key to the success of this strategy lies within the repetitive nature of most numerical computations, and the resolution dependence of many array lengths. In consequence, the sizes of array spaces requested are far from random, and often similar.

5.4 Release strategy - Work Space and Long Term Space

When space is released, the "name" entries in the tables are replaced by blanks. If the space corresponding to the last table entry is released, the stack-pointer is decremented and the appropriate length added to the available space.

6. ALGORITHMS

6.1 General

The following algorithms were devised specifically for a Cray-1 computer with vector capabilities and a 64 bit word length. Characters are stored 8 per word, using 8 bits per character (ASCII). Names are assumed to be left justified, up to 7 characters in length. Despite these machine dependent features, some of the following algorithms are suited to other configurations.

6.2 Matching a single table entry

A specialised routine, MATCH, was written in Cray assembler language (CAL) to examine table entries 64 at a time using the vector facilities of the Cray-1 thus:

a) store value to be matched in a scaler register.

b) load 64 values from table into a vector register.

c) store vector difference in vector register.

d) set mask to ones for zero vector elements.

e) update counter and table address.

f) return to b) is mask is zero.

g) count leading zeros in non-zero mask and update counter.

h) return value of counter (zero if no match found).

This provides a means of locating the entry in a table matching a given value. The subscript of the first matching entry is returned.

6.3 Matching two table entries simultaneously

The routine MATCH described above was extended to match two table entries to two given values simultaneously. The resulting routine, MATCH2, uses the following algorithm.

a) store value to be matched in Table 1 in scaler register 1 (S1)
b) store value to be matched in Table 2 in scaler register 2 (S2)
c) load 64 values from Table 1 into vector register 1 (V1)
d) obtain vector differnce S1-V1 in vector register 2 (V2)
e) set mask to ones for V2 zero elements (VM0
f) update counter, Table 1 address, and Table 2 address
g) store VM in scaler register 3 (S3)
h) return to c) if S3 is zero
i) load 64 corresponding values from Table 2 into vector register 3 (V3)
j) obtain vector difference S2-V3 in vector register 4 (V4)
k) set mask to ones for V4 zero elements (VM)
l) store VM in scaler register 4 (S4)
m) return to c) if S4 is zero
n) obtain logical product of S3 and S4 in scaler register 5 (S5)
o) return to c) if S5 is zero
p) count leading zeros in S5 and update counter
q) return value of counter

This requires the second table to be consulted only for segments where matching values have been found in Table 1. Where no matching values exist in Table 1, a zero value is returned; in such cases Table 2 is not examined, and the only additional cost is that incurred in updating Table 2's address in f) above.

6.4 Hashing names

Even when vector matching routines are available, searching tables can be expensive. In consequence it was considered desirable to provide a hashing technique based on 7 character names. The algorithm used is:

a) obtain exclusive OR (XOR) of KNAME with KNAME left shifted 12

b) XOR a) with KNAME left shifted 24

c) XOR b) with KNAME left shifted 36

d) XOR c) with KNAME left shifted 48

e) return d) right shifted 56 with zero fill.

The result is an 8 bit value formed by exclusively OR-ing

a) characters 1, 4, and 7, and

b) the lowest 4 bits of character 2 plus the highest 4 bits of character 3, and

c) the lowest 4 bits of character 5 plus the highest 4 bits of character 6.

Using this technique, 90% of the names used in ECMWF's operational numerical forecast are hashed to unique values in the range 0 to 255.

6.5 Allocation of long term space

A hash table of length 256 is initiated with all values equal to 1. The algorithm for allocation of long term space is then:

a) hash KNAME, and extract value stored at this entry of hash table (IPOS)

b) check entries in management tables at position IPOS against KNAME and KCODE. If correct entry located, return KPOINT from tables to user.

c) otherwise use 6.3 above to find table entries matching both KNAME and KCODE. If matched, return KPOINT from tables to user, and update hash table.

d) if no existing entry can be found, use 6.3 above to find table entries with blank name and matching length (KSPACE). If matched, return KPOINT from tables to user, and update hash table.

e) if no returned space of correct length can be found allocate space from the residual area, adding a new table entry to the top of the table stack, and up-dating the hash table.

6.6 Location of long term space

Previously allocated space is located as follows:

a) hash KNAME and extract hash table entry (IPOS)

b) check entry IPOS in management tables against KNAME and KCODE. If correct, return KPOINT to user.

c) otherwise use 6.3 above to match both KNAME and KODE. Return the matched entry for KPOINT to the user.

6.7 Release of long term space

Previously allocated space is released as follows:

a) hash KNAME and extract hash table entry, IPOS

b) check entry IPOS management tables. If not correct, match KNAME and KCODE using 6.3 above.

c) change KNAME in table entry to blank.

d) if IPOS is at the top of the table stack, collapse the stack, returning space to the residual area, until a non-blank entry appears at the top of the stack of names.

6.8 Memory distribution

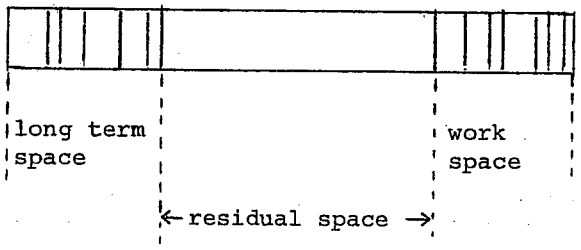

Fig.1 Memory distribution

Fig.1 illustrates one method of organising the memory space to be managed. Long term space is allocated from low order memory, work space from high order memory, with a moveable partition of residual space separating the two. This provides the flexibility of allowing all space not allocated for long term storage to be used as work space, and vice versa.

7. CONCLUDING REMARKS

Memory management software using the above ideas has been found to be both flexible and efficient. A version used by ECMWF's numerical weather prediction model uses less than 4% of the CPU time of the code; of this, 1.2% are traceback overheads to allow comprehensive error facilities to provide useful information when errors are detected.

Memory management software does not relieve the user of the task of planning the use of memory space in a sensible manner. It does provide a means of using space efficiently, and allows the user to use sensible array names and structures. The capability of allocating several areas of storage in a contiguous block facilitates the buffering of data for input/output processes. Flexibility is enhanced because extra variables can be introduced into the code without disturbing a rigidly structured memory configuration.

Good diagnostic and trace facilities are important. It has been found that the provision of switchable trace facilities has been especially useful in programme development. The ability to call a trace routine to print a map of the managed space in critical areas assists the user to plan, and often points to ways in which problems may be overcome.

A final, but important feature worthy of comment is the usefulness of based variables as a means of reducing programming errors. In an environment where all POINTER variables are set as a result of calls to memory management routines the liklihood of using non assigned data, or over-writing previously assigned data incorrectly, is reduced. If a POINTER variable has not been set before reference is made to a based variable, a fatal error usually results. Over-writing in a managed system takes the form of allocating space which must first have been released. The resulting increased confidence in the integrity of the code is a considerable benefit.