DataSpace Entity Specifications

The MOSES Project:

Meta Operating System And Entity Shell

Daniel J. Pezely

9 April 1991

---

1 Introduction

The DataSpace of the MOSES Project is a shared, virtual memory which appears central and an exclusive resource to each of its multiple users. Users will ultimately be client programmers whose routines will be accessed by higher-level users. This is the description and specification overview of the DataSpace and its operational routines.

2 Design and Use

The DataSpace contains all data structures for the kernel, as well as all allocated storage for the kernel and some of its daemons and applications. This storage facility (location, functionality, and access methodology) is referred as the DataSpace Entity.

The DataSpace provides a virtual, centralized, and shared memory for applications and also shares that memory with the kernel. Therefore, all ker-

nel variables are potentially accessible to users and applications. The unity which the DataSpace provides, makes accessing distributed memories transparent to its users. The distributed memories may be shared, but each user can have the illusion of ownership. Allowing for distribution, the DataSpace can store run-time specific data, such as function pointers, in RAM and all other data on a database server.

The DataSpace is an associative memory tuple space similar to the Linda [1] shared memory space, and the Linda commands may be inplemented using the DataSpace operations.

In Linda, the information list construct is a tuple; however, our design allows for nesting and the actual address of the tuple may be used. Such a *tuple* violates its definition, so we refer to information lists as *grouples*.

The operations on the DataSpace are defined in the following modules:

- Symbol module – operations on generic, atomic data-strings
- Term module – operations on elements of grouples
- Grouple module – operations on whole grouples
- Globals module – operations on kernel variables

where each module has these base operations:

1. *New* – attempts memory allocation
2. *Delete* – removes one reference to data and deallocates unreferenced data
3. *Copy* – duplicates structure and content of data
4. *Evaluate* – executes specified task
5. *Select* – finds a match in the DataSpace from a pattern
6. *Substitute* – replaces one instance of data with another

Nesting is allowed via addressing the DataSpace internally. If the actual address is used for access, the addressed grouple is checked against the search pattern to avoid use of stale addresses. This feature is provided for user-alternative access methods such as hashing and trees.

The DataSpace is linear; however, to implement organizational structures such as trees, sublists, or hash tables, the indices needed should be implemented within the DataSpace. That is, since indices are references which are just data, use the DataSpace to store that data. Since the code to implement such structures would have to be linked in with the kernel to execute efficiently, the indices could be accessed without the complexity of searching the DataSpace via the DataSpace-Globals Module.

2.1 Storing Data

2.2 Storing Functions

2.3 Use of Genuine Dynamic Linking

3 Data Structures

The two data structures described in the following subsections provide a head and body relationship similar to disk file headers and their corresponding chained data blocks. As files relate to a file system, the following data structures will be maintained within the DataSpace. File system and networked file system design features should be part of the data structures to provide the user with a shared memory.

The two structures are `GroupleHeadClass` and `GroupleTermClass`, and referred to as heads and terms, respectfully. *Grouple* in this context collectively refers to a head and its corresponding chain of terms. And the DataSpace is the list of all allocated grouples linked together.

As is commonly the case with handles, references to the head will always remain intact while the second reference, to the terms, may be altered; thus overall data integrity is maintained.

For more complex grouple storage and access methods then the linear order of the DataSpace, indices may be constructed and maintained using grouples. That is, by creating a grouple which is ordered via a tree pattern or hash function, accessing the DataSpace by first accessing a known grouple could reduce searching complexity. Such tools should be implemented as complete objects which use the functions prototyped in this document and not need to modify these data structures or add functions to the the DataSpace Object. Such objects should be the *only* other features added to the

kernel; all other objects should be built into external kernel services.

3.1 Type Definitions

```
typedef struct _GroupleHeadClass  GroupleHeadClass;
typedef struct _GroupleTermClass  GroupleTermClass;
typedef unsigned long             GroupleSymbolClass;
typedef GroupleHeadClass    * ( * GroupleFnPtrClass)(GroupleHeadClass * );

typedef enum _GroupleFormTypes    GroupleFormTypes;

enum _GroupleFormTypes {
    UNKNOWN_FORM = 0,
    SYMBOL_FORM,
    SUBLIST_FORM,
    FN_FORM
};
```

Of the two structures, `GroupleTermClass` is referenced inside `GroupleHeadClass`. The `Symbol` should occupy the full word size of the host CPU general registers and only effects the choice for optimal allocation and transfer sizes. The order of enumeration constants shall be maintained to accept run-time and dynamic linking consistency.

The `VariableClass` definition is equivalent to the `GroupleHeadClass` *typedef*. This definition duplication is intended to give client programmers a simplified type for use with variables to be held in the DataSpace.

Grouple symbols are generic for holding characters larger then just one byte. To maintain consistency, the network byte-order (most significant octet first) shall be used for communication and possibly storage.

The `GroupleFnPtrClass` type-definition is a function pointer which references a function taking a `GroupleHeadClass` reference parameter and returns the same type as its parameter.

3.2 GroupleHeadClass

```
struct _GroupleHeadClass  {
    unsigned long                   flags;
```

```
    unsigned long                   readers;
    unsigned long                   links;
    GroupleTermClass               * head;
    GroupleTermClass               * tail;
    GroupleHeadClass               * next;
};
```

Three status fields are used in this structure: `flags`, `readers`, and `links` representing, respectfully, modification and data sharing flags, the number of entities reading this grouple, and the number of grouples referencing this grouple. These fields are *unsigned* integers and resemble fields in a file-system *inode*.

The `head` and `tail` member fields respectfully reference the start and end of the chain of terms associated with this grouple and both shall be either null or non-null.

The `next` field of the grouple structure may or may not be used for organizing the allocated grouples. If the field is not used, chains of the root level lists can be efficiently allocated and managed by using what the BSD people call *superblocks* [2]. This is discussed further in the **DataSpace Management** section below.

All `GroupleHeadClass` instances shall be linked into the DataSpace when allocated, thus allowing traversal of all allocated grouples.

3.3 GroupleTermClass

```
struct _GroupleTermClass {
    unsigned                        flags;
    GroupleFormTypes                type;
    union _GroupleForm {
        GroupleSymbolClass * symbol;
        GroupleHeadClass   * sublist;
        GroupleFnPtrClass    fn;
    }                               form;
    unsigned                        size;
    GroupleTermClass              * next;
};
```

A `GroupleTermClass` instance shall store and maintain only one form a symbol, a pointer to a nested grouple, or a function pointer via a *union* structure and corresponding *enum* field, `type`. These members are, respectfully, `symbol`, `sublist`, and `fn` within the `form` *union* and `type` *enum*.

Since the `type` field is an enumeration with only three values, only two bits are required to store the value. The extra bits may be used as a `flags` field, which would be helpful for parallel evaluation which operates upon a multiple terms concurrently.

The `fn` field is a pointer to a function which takes a `GroupleHeadClass` pointer parameter and returns the same type. The C language syntax can become messy if typecasting is necessary, so a *typedef* is provided for the function pointer: `GroupleFnPtrClass`.

The `size` member field is for maintaining array lengths of any of the three `form`s. For example, `size` would refer to the number of characters in the `symbol` field and not to the number of bytes of that field. A zero value in `size` shall mean an array is not used in this node, and a non-zero value shall mean that an array is used. Arrays can be allocated by typecasting the pointers, since a pointer to a pointer is still a pointer. The `next` field points to the next term in the chain of this grouple.

All member fields are pointers with the exception of `type` which is an *enum* and `size` which is an *unsigned* integer.

4 DataSpace Management

The order and structure of the DataSpace has only one requirement: some users must be able to traverse the entire DataSpace from beginning to end without regard for any subspace boundaries.

This traversal ability requirement may be done by ordering all `GroupleHeadClass` structures in a list indexed by a tree or a table. There is a `next` field in the grouple head data structure give above for the simplest implementation using a singlely linked list. However, since the grouple structure serves much the same purposes as a file system *inode*, the *superblock* methodology might be used. [2]

As is done with many large database systems today, multiple storages and access mechanisms are used. Although only one access method is used at once, as the data grows, different approaches become more efficient. The

desired degree of efficiency is ultimately up to the implementor and development requirements.

It is important to note that at this level, we are only accessing grouples, without any knowledge or consideration of what is stored within those grouples. When we need to have high-performance access to the grouples selectively, we then need an intelligent parser which is the next level higher in the abstract model.

5 Parsing the DataSpace

Any parser which operates on the DataSpace does not directly access any grouple. Instead, the parser will decide what it wants and tell that to one of the selection routines. The selection routines, then, do the actual access of the DataSpace.

6 Global Variables

All global variables within the kernel should be stored within the DataSpace. This provides users access to otherwise internal data and is modifiable at run-time.

The only extra overhead of using variables in this way is the indirection of pointers to the data. The functions calls used to access the data should be *inline* routines to handle the known templates, thus have no overhead of parameter passing at run-time.

All data structures which might need to be created for the I/O routines shall be linked into the DataSpace by storing the structure address as a symbol and storing the first word of the structure as the first element of a symbol array. The Grouple I/O Object, for example, maintains active file descriptor tables in this way.

6.1 Syntax

All global variables used by the kernel (and thus all the objects and modules within the kernel) shall be:

1. stored in the form of *grouples*

2. using standard notational conventions, the structure corresponding to the Global-Variable template will be:

   `<variable-name> [<sublist of properties> ... ] <value>`

   The value of the variable shall always be the last element in the grouple, even if the value is a reference to another grouple

3. stored in the DataSpace

6.2 Location

All global values used by the kernel, the kernel programming objects, and their associated modules, shall be stored in the DataSpace. This gives the user access to all otherwise hidden variables and state values. Global pointers will be maintained to reference specific elements of the DataSpace for run-time efficiency, so restrictions must be placed on such grouples.

6.3 Access

The kernel routines shall access all global variable values via a pointer to a `GroupleHeadClass` structure (which will always be linked into the DataSpace) and reference the `tail` member field to reach the actual value. Actual operations are implementation dependent.

References

[1] Carriero, N., Gelernter, D., "Applications Experience with Linda," *Symposium on Principles and Practice of Parallel Programming, Proceedings of the ACM/SIGPLAN*, Volume 23, Issue 9, September 1988, pp. 173-187.

[2] Leffler, S.J., McKusick, M.K., Karels, M.J., Quaterman, J.S. *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley Publishing Company, New York, NY, 1989, pp. 13, 203-208, 281-310.