THE ENTITY OPERATING SYSTEM: A DESIGN FOR THE IMPLEMENTATION OF VEOS

Daniel Pezely, Michael Almquist, with William Bricken
February 1991

It is important to note that much of the system is borrowed from Lisp systems design and Unix systems design since both have proven to be successful and the elements of each which are used suffice for our implementation design.

Our system is written in C for portability, but the code structure is very much like C++ or Objective-C. The comments in the code even refer to structures as Classes and each derived class, even if nothing more than a typedef, has a complete set of macros with a consistent naming convention which correspond to the derived class. (The reason for such typedef classes is to replace only one or two routines of the associated base-class library, and the function names should be kept similar in such a way as to avoid client-programmer confusion; this is described in detail below.)

Goals

We need to introduce a platform which resides on top of a computer system's native operating system which provides a system-independent layer which is completely user-extensible. Total extensibility usually causes problems and such designs are known as `second systems'. However, our approach should be successful with this goal because the ability for user modifications comes from design features inherently and was not planned for in the original design.

The way to provide for complete user-extensibility is to be consistent with what the user has access to and what the system accesses as part of its normal operational routines. By this, I mean all internal system variables are kept in the user data space but internal pointers are maintained to bypass the user access channels. This allows the user access to all the system internal variables thus free to modify the values. By having a tight control over when the internal variables are accessed and how the user may modify the values, the sanity of the system may be maintained.

Grouples and Lisp S-expressions

The root class Grouple_Class which is the head of a chain of Grouple_Term nodes which form lists of lists and/or symbols. This combination of Grouple_Class and Grouple_Term forms the notion of a grouple. The grouple is an abstracted version of s-expressions in Lisp systems.

In Lisp, there are basic categories of s-expressions: lists, functions, and

symbols (or strings of atomic characters). With grouples, the designation of the type of s-expression is specified in another grouple which is the parent to the particular s-expression. Think of it as a list with specific keywords used to specify the type of a sub-list, and that sub-list is a Lisp s-expression.

The member fields within Grouple_Class are for bit-flags (for a shared memory environment), the head and tail of the list of Grouple_Terms, and a way to reference the next node in the Grouple_Class list.

The member fields within Grouple_Term are, first a union-structure continuing the various C syntactic forms for strings, pointers to the beginning of other grouples to build nested grouples, and function pointers; and the other Grouple_Term member fields are for specifying which union-structure form was used, specifying the array sizes for such things as strings, and finally the reference to the next Grouple_Term in the chain within a single grouple.

==========

The C data structures

```
typedef struct _Grouple_Class         Grouple_Class;
typedef struct _Grouple_Term          Grouple_Term;
typedef  unsigned long                Grouple_Symbol;
typedef enum _Grouple_FormTypes       Grouple_FormTypes;

struct _Grouple_Class  {
     unsigned long               flags;    /* Locked,Modified,etc */
     Grouple_Term *              head;     /* Actual grouple */
     Grouple_Term *              tail;     /* End of grouple */
     Grouple_Class *             next;
};

enum _Grouple_FormTypes  {
     UNKNOWN_FORM,  SYMBOL_FORM,  TERM_FORM,  FN_FORM
};

struct _Grouple_Term {
     union _Grouple_Form  {
     Grouple_Symbol *  symbol;             /* Name or value */
     Grouple_Class *   list;               /* Nested grouple */
     int          (*fn)();                 /* Function pointer */
     }                  form;              /* Form of this term */
     Grouple_FormTypes  type;              /* Which Form to use */
     unsigned           size;              /* For arrays... */
     Grouple_Term *     next;
};
```

==========

Strings defined in this design are not restricted to the standard ASCII characters (8 bits) but are permitted to be any unsigned long integer (32 bits) with no restriction of tokens for terminating the string, as C strings use value zero.

The array-size field of Grouple_Terms specify the number of elements within the string, and similarly, arrays of nested Grouple_Terms and arrays of functions are permitted by definition.

Although the two C data structures reference each other, the corresponding library routines are kept separate and clean from unnecessary cross-connections. The only routines which need to access both structures directly are the New() and Delete() routines for allocation/construction and deallocation/destruction, respectively.

Common Lisp and Parsing

Since the structure of Lisp is provided with this design, a full implementation of Common Lisp will also be provided with the kernel. However, the Lisp system will be external to the kernel and may be thought of as a kernel service or system daemon when thinking in terms of message-passing operating systems. This externalization of the Lisp system allows users to use whichever implementation of Lisp they chose. At present, XLisp will be the pre-configured system since it is easily accessible and comes in C source code form.

A Common Lisp reader will be provide by our system for receiving input messages. The reader will only construct a grouple to contain a valid message, and the parser will try to resolve the new message.

The parse tables will consist of first a table of constants, then tables for built-in system routines and tables for user-provided routines. The next phase of the parser will be to pass any yet unresolved messages on to the Lisp system. Successfully resolved messages from any phase of the parser will be matched against an empty grouple. Empty matches will be discarded, and non-empty matches will be substituted back into the DataSpace. Substitutions may be multiple grouples by listing the head of each grouple in a keyword-named list (grouple) which the substitution mechanism of the parser recognizes.

If the DataSpace is shared between multiple parsers or if multiple Lisp routines need access to the same grouple, negotiations must be made so as to re-substitute the original message back into the DataSpace for the additional routines to use. This may be done by creating macro-like functions for naive routines and placing such macros in a specific user space which gets checked prior to all other user spaces.

A garbage collection routine will be used to clean up the various spaces by checking time-stamps and other criteria set by the user of the system.

The DataSpace and Sub-spaces

The DataSpace is the collection of all the allocated grouples. Every grouple created is immediately appended to the DataSpace so that just as a disk file may be accessed sequentially, so too can the DataSpace. Within the DataSpace are various sub-spaces which provide segmentation and reduce the complexity of searching the entire DataSpace.

These sub-spaces consist of the WorkSpace for all input/output messages and temporary grouples, the GroupleFreeList and TermFreeList for holding unused grouples and grouple terms, the SysLibSpace for organizing the list of all system library routines available to the user, the UsrLibSpace for organizing the list of all user-defined routines, the SysConfigSpace for enumerating the various actual and virtual resources available to the system and the user, and also, each function created by the user is held in its own space which is known by the UsrLibSpace to provide for Lisp Packages. Other spaces may exist within the DataSpace, but the ones mentioned are the primary ones. The names used will be maintained as system keywords for future versions of the system.

Within the C source code, a space is defined as a pointer to Grouple_Class. However, the pointer does not reference the grouple containing the name of the data list but references the actual list which contains the data. As mentioned above, the names and types and other related information within a Lisp system are held in one grouple and one element of that grouple references a second grouple which contains the actual data. It is this second grouple which the C pointers reference.

The DataSpace is not a shared memory environment yet. This will eventually be done, but existing database system will be used for that. The DataSpace resides in main memory only, specifically in only the heap of the kernel process.

Main() and a Replaceable Main Loop

The Main.c file is approximately 120 lines, half of which are block comments describing in English the source code block which follows it. The Main function does three things: initialize the DataSpace to construct all of the various sub-spaces mentioned above, configure the kernel for command-line parameters and config files, and then enters an infinite loop which indirectly calls the read-parse loop.

Main indirectly calls the read-parse loop in the literal sense. That is, a

function pointer is used --- which is of course accessible via the DataSpace --- so that a command called via the parser may replace the read-parse routine by another routine. If, however, the replacement function is null, the default read-parse loop is restored. The specific details are described the block comments of Main.c.

The Result

With such a system that permits user access to all kernel variables and a replaceable main event-loop, a solid structure is provided which may be changed as the user's needs predict. Such as system is not meant for high-performance but is meant for research, development, and ideally to be used as a tool to gain insight and understanding which will yield flexible high-performance systems in the future.

(Security has not been designed into this system since it is meant to be for research and development and not for a production environment. Such features will be included in future generations of the design.)