NUE/TAO/ELISのOS的側面

竹内郁雄　奥乃博　大里延康
(日本電信電話公社 武蔵野電気通信研究所)

1. INTRODUCTION

NUE (New Unified Environment) is a total programming environment which supports AI research and AI software development. NUE is:

(1) Based on list processing language TAO implemented on Lisp machine ELIS [1]
(2) Used mainly (co-operating) researchers and professional programmers
(3) Highly interactive via advanced terminals
(4) Suitable for large program development.

NUE, as expected from its name ("nue" in Japanese means a legendary Japanese chimera with monkey's head, tiger's limbs, racoon's body, serpent tail and golden mountain thrush's voice, or such incredible mixture of somethings), is so designed that it incorporates everything useful for AI programming. We think that the tool for AI research is necessarily monstrous because AI itself is intrinsically monstrous. Thus, the kernel language TAO is a multiple paradigm programming language which combines Lisp, Prolog, Smalltalk and even Fortran on the basis of S-expression of Lisp. TAO can meet various requirements that arise in practical AI programming. (The "monstrous" here is not the word for aesthetics of programming language design and its implementation, it represents the richness of the language functionality.)

The name TAO is originated from the ancient Chinese philosophy "Taoism", and the language design itself is strongly influenced by the philosophy. For example, the words of the first chapter of Tao Te Ching,

"The Tao that can be Taoed is not the true Tao"

implies the vital evolving power of the TAO; TAO does not stay at a fixed, stuffy, bureaucratic position; TAO will change and augment its functional power being motivated by user's needs and implementor's new ideas. The programming environment NUE also inherits this basic philosophy because the border of environment and kernel language is blurred anyway in such a conversational programming system. In this sense, NUE is more vague than TAO for the present since TAO becomes concrete only recently. There is only dream of NUE now. There is not any solid design of NUE. NUE will emerge and become clearer as time passes since the current status of TAO/ELIS is sufficient enough to experiment novel ideas on programming environment without much effort. The various ideas will be fused into some ideal solution through "artificial" and "natural" selection.

This report will describe mainly the OS facets of the language TAO, but also show some ideas of the first stage NUE.

2. MULTIPROGRAMMING

Multiprogramming facilities are embedded in TAO and their crucial part is implemented by firmware (about 1 Ksteps). Typical process switching time is less than 20 or 30 microseconds.

Each process is an object in the Smalltalk sense, called udo (acronym of User Defined Object). Process udo's structure is shown in Figure 1. Process creation and maintenance is programmed partly in object oriented style of TAO. Semaphore and mailbox described below are also realized as a udo.

FIGURE 1. Structure of process udo.

```
{udo}28457process, an object of class process (version 0),
has instance variable values:
 !oblist:                    {vector}28451(oblist . 10)
 !status:                    #10 ... bit table of process status
 !whostate:                  running ... simplified mnemonic of process status
 !priority:                  2
 !wait-for-what:             (4 5) ... semaphore, mailbox, etc. used also as
                                 return value slot
 !resumer:                   nil ... used when process is used as coroutine
 sys!prestk-memblk-list:     nil ... list of swapped stack contents block
 sys!sp:                     {stkpt}32509 ... preserved stack pointer
 sys!sbr:                    #7002 ... preserved stack boundary check register
 sys!bottom-stack-block#:    #17 ... stack block at bottom
 !name:                      gonbe ... process name
 !interprocess-closure:      nil ... shared variables with other processes
 !initial-function:          top ... process's first function [top-level loop]
 !initial-argument-list:     ({udo}27952fundamental-stream ... [top's arguments]
                              {udo}27982fundamental-stream )
 !quantum:                   5 ... [this process will run at most 5 * 20ms]
 !login:                     {udo}28440login ... login that creates this process
 !interrupt-fn-args:         nil ... form that will be evaluated on interrupt
                                 (set when interrupt actually occurs)
```

Process's waiting state is one of the following:

(1) I/O wait: Wait for I/O ready signal interrupt from FEP.
(2) Semaphore wait: Semaphore is the counting type.
Semaphore is a udo with two instance variables:
sys!semaphore-process-queue
sys!semaphore-value
(3) Mail wait: Arbitrary S-expression may be posted in Mailbox.
Mailbox is a udo with two instance variables:
sys!mail-process-queue
sys!mail-queue
(4) General wait: Wait until a TAO form evaluates to non-nil value. The evaluation can be performed either every 20 millisecond, or every 1 second, or every 1 minute.
(5) Timeout wait: Wait until a certain amount of time elapses. Timeout wait can be overlapped with one of the above waits.

ELIS has 32K word high-speed stack memory which is segmented into 16 stack blocks of 2K words. Each process is associated to a number of consequent stack blocks dynamically. That is, the number of consequent stack blocks allocated for a process gets larger when the process consumes much stack memory and reduces when it needs fewer stack memory. An internal table controls which process occupies each stack block. If many processes run concurrently, stack block occupation conflict may occur. In that case, the process which is currently sleeping or waiting will release the conflicted block and save the stack contents into main memory. When the process gets to run, the saved contents are restored. This swapping, however, brings in only less than 2 millisecond overhead per one stack block even in the worst case where both swap-out and swap-in take place simultaneously.

Since stack memory of ELIS has a ring structure, one process could utilize full 32K word stack memory regardless to the stack bottom address. However, TAO does not exploit the ring structure because of some implementation convenience. (One of the 16 stack blocks is utilized for system working area which must accessed faster than areas in main memory, for example.) This may cause a problem that a process whose stack bottom is near to the system reserved stack block has a limited stack capacity much less than the maximum available stack capacity. However, this problem can be alleviated by the stack contents relocation because the contents are almost self relocatable.

Anyway, the stack block allocation for concurrent processes is not a trivial problem. We don't know the best solution yet, but we expect that we can experiment a variety of allocation algorithms by using TAO, not by rewriting microcode, because TAO has abilities to clobber any word and any stack address of ELIS.

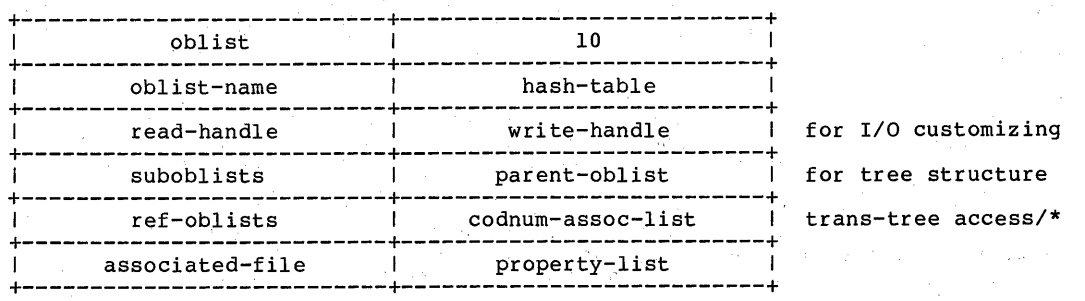
Common variables shared by two or more concurrent processes are packed into a Lisp function closure (See interprocess-closure in Figure 1). Function closure is a data structure which holds pairs of variable names and their values. Since variable binding information itself can be treated as a kind of concrete data type, variable sharing can be realized quite naturally in such Lisp type language TAO which has no block structure like Algol.

3. OBLIST GRAPH --- MULTIPLE NAME SPACE

Lisp's identifiers (atom) are unique in the sense that two identifiers with the same print name are identical. The fact that identifier itself is an individual data type and identifier is unique is one of the most important features of Lisp which mark off Lisp from other conventional programming languages. However, same identifiers often conflict each other when many programmers are involved in developing a large scale application software in Lisp. Moreover, important (globally accessed) identifiers and unimportant ones are all at the same level, which makes it harder to maintain the program modularity. TAO, as ZetaLisp [2] did, allows that more than one identifiers with the same print name co-exist in the same Lisp environment. However, TAO goes further than ZetaLisp in that TAO allows multiple users to share some common identifiers and not to share other identifiers intentionally according to the level of their co-operation.

Traditionally, Lisp controls the identifier uniqueness by a device called oblist or obarray which is essentially a hashed name table. To realize multiple name space as described above, more than one oblists (or obarrays) should be arranged in some structured manner. TAO arranges oblists in a tree structure whose root is named "univ". Direct successors (suboblists) of univ is: bas (oblist for basic TAO), sys (oblist for system programmer's TAO), and key (oblist for keywords which are very commonly accessed from all oblists). This is a static structure determined at the oblist creation time. Moreover, each oblist can dynamically span arbitrary access path from it (called ref-oblists), thereby any kind of trans-tree short-cut access to other oblists is possible. Figure 2 shows the oblist structure.

FIGURE 2. Oblist structure (realized by a vector, not a proper udo)



Each identifier in an oblist is either external or internal; external identifier can be accessed from other oblists without explicit designation of oblist name, but internal one cannot.

An oblist, called current oblist, is associated with each process. When an identifier is input (or "intern"ed), the following search rule is applied:

(1) If the identifier is of the form A,
   (a) search the current oblist
   (b) search ref-oblists (a list of ref-oblists) one by one from the top
   (c) search parent-oblist chain from near to far
   (d) if not found, create A in the current oblist

(2) If the identifier is of the form A!B!...!Y!Z
   (a) if the user has not the privilege, error.
   (b) search an oblist which embraces A. If found, examine the oblist name. If it equals to A, restart searching for B!C!...!Z from the oblist. If it does not equal to A, examine the names of the suboblists whether one of them equals to A. If such one is found, restart searching for B!C!...!Z from the suboblist.
   (c) if one of oblist-name is not found, create Z in the current oblist
   (d) if just the Z is not found, create it in A!B!C!...!Y oblist

(3) If the identifier is of the form !A
   (a) search key oblist for A
   (b) if not found and the user has the privilege, create A in key oblist

As can be seen above, a certain kind of access to other oblist is forbidden if the user has not appropriate privilege. There still remains some issues among us on how to deal with the modification of identifier's property (function definition, property, global value) from users in other oblists. (Controlling them completely by user's privileges may burden some primitive Lisp operations with overhead.)

The following functions are used to control the identifier scope.

(export id1 id2 ... idN) makes id1, id2, ... idN external.
(intern-local* id1 id2 ... idN) makes id1, id2, ... idN local to the current oblist. That is, even if idj in another oblist are accessible from here, intern-local* creates new idj with the same print name in the current oblist. (ZetaLisp calls this feature "shadowing".)
(import id1 id2 ... idN) deletes id1, id2, ... idN from the current oblist. This cancels the effect of intern-local*.

4. MULTIPLE USERS

As opposed to the current trend that single language dedicated machine is developed as a super-PERSONAL machine, the first step of NUE is to develop a simple multiple user environment. We think that most super-personal machines are too work-station oriented, and hence much design effort and much CPU power are devoted to the cosmetology to make the system's face beautiful. We aimed at the powerful engine for AI at first, that is, a clever machine rather than smile selling machine, which can be achieved at the second step on a clever machine.

We assume that most users who log in co-operate to certain extent to achieve a common goal. Hence, protection between users are loose for default. They can access to and modify common data as they want. We expect that the co-operating users are sitting in one room so that they can freely talk to each other on their tasks. The number of active concurrent users is expected to be 4 or 5 at maximum to guarantee the response time comparable to TSS Lisp on a high-speed large scale computer. This usage of an AI machine would be suitable at least for next few years from the viewpoint of cost performance and rapid build-up of common tools.

Many people say us at a first glance that TAO/ELIS's fatal error is the lack of bitmap display and mouse. We don't ignore them at all. But considering our daily programming activity, we scarcely need them. We prefer to occupy two or more dumb but fast terminals rather than occupy a single, expensive bitmap display for the present, even after experiencing most of famous super-personal computers such as LMI Lambda, Symbolics 3600 and Xerox 1100SIP. (Our impression on these systems is that their man-machine interface is excellent but too slow.) We consider that bitmap display interface will be more improved in the future and it will be an independent ready-made technology that can be easily attached to AI programming engines such as TAO/ELIS. Hence, we decided to give priority to develop a simple multiple user environment based on dumb terminal of at least 9600 baud rate.

An ideal man-machine interface for personal use will be developed soon after we complete the basic development of the first stage NUE. The technology we obtain by making the first step will be inherited directly to the real super-personal system.

5. CONCLUDING REMARKS

The language TAO runs only interpretedly for the present. After making the very basic multiple user environment, we focus on the compiler construction by which TAO will speed up by a factor of 3 to 5.

Then, the real-time garbage collection (GC) will be implemented. If ELIS has only 4 or 8 megabyte memory, the current batch type garbage collection makes the user feel almost no timing delay since it takes only 1 or 2 seconds. The algorithm of the real-time garbage collection is quite simple. It is a variation of batch type mark-and-collect garbage collection. If any data type's free storage becomes short, the GC process wakes up and runs concurrently with other processes. GC process chases other process's stack modification at its maximum speed; if a process runs in GC marking mode, GC will mark the whole stack area for the process again from the beginning. After some time, almost all data are already marked and hence GC process's chasing will be much faster. Eventually, GC will catch up all the process. If it is expected to take much time to catch up, GC process's priority and quantum will be changed upward. In GC marking mode, only rplaca (replace car) and rplacd (replace cdr) slow down quite slightly in order to check whether marked list is rplaca(d)ed by unmarked data. No other list processing suffers from the overhead brought in by real-time garbage collection. The collection phase is easier than marking phase; it collects free storage only by need. Further detail is omitted here.

The next and very important one is network support. But we cannot say anything about that for the present.

ACKNOWLEDGMENT

We gratefully thanks to Mr. Y. Hibino who developed the base machine ELIS and now promotes NUE project. We also thanks his valuable suggestions.

REFERENCES

[1] H.G. Okuno, I. Takeuchi, N. Osato, Y. Hibino, and K. Watanabe: TAO - A Fast Interpreter-Centered Lisp System on Lisp Machine ELIS, Conference Record of 1984 ACM Symposium on Lisp and Functional Programming, Austin, Aug. 1984.
[2] D. Weinreb, D. Moon, and R. Stallman: LISP Machine Manual, MIT, Jan. 1983.