A NEW APPROACH TO CONSTRUCTION OF COMPUTER SYSTEMS

Tatsuya Hayashi
Fujitsu Limited, Computer Science Laboratory
Kawasaki
JAPAN

In this paper we propose a new method in which original form of the whole system (described in a design and implementation language called DEAPLAN) is directly implemented as it is without either modification or transformation. In other words a kind of high level language machine is considered in the more throughgoing way. Our hardware apparently has neither CPU nor storage device and only consists of a large number of quantum processing units (QPUs) except channels and peripherals. Therefore, extremely speaking, the main contemporary concepts such as virtual space, reenterability and multiplexing of CPU as well as compiling and linkage editing are all disappeared in our system. The identity of implemented version of the system with the original form seems to give a more fundamental solution to the problem of the rapid growth of operating systems compared with the mere structured programming and so on.

1. INTRODUCTION

In the present decade, the people such as N. A. Chomsky (Linguistics), R. Jakobson (Phonology), C. Levi = Strausse (Cultural Anthropology), Bourbaki (Modern Mathematics) and J. Piaget (Psychology) have obtained excellent results based on the Structuralism. The Structuralism has various aspects. But we may say that its fundamental standpoint exists where both simple positivism and inductionism should be excluded and asserts that using the provisional models, we could recognize the deeper realities which lie behind our various cultural activities and usually are out of the range of our consciousness. Then, the structured programming proposed by E. W. Dijkstra [1] is also considered to belong in the same category.

We also have taken the same approach when we have developed a design and implementation language (DEAPLAN) [20] . DEAPLAN, requiring no prior condition concerning the structure of operating systems, enables us to describe the whole hierarchical logical structure of any operating system.

In this paper, we present a new architecture of computer systems based on the principle of Structuralism. In other words, considered is a new method by which the system is directly implemented from the original representation in DEAPLAN without any omission or transformation.

Each of current operating systems (MULTICS [2] , OS/VS II [3] , etc.) is thought of essentially having some hierarchical structure in its original form. But when actually implemented, the system is transformed into the one which, as a rule, consists of only the lower parts (procedure, statement, operation, etc.) of the original hierarchical structure which are described in conventional programming languages. The upper layers (subsystem, job, task, load module, etc.) of the original form that cannot be described in conventional programming languages are indirectly represented, if any, in such a manner as using control blocks. Furthermore, even if the lower parts of the original form are considered, it may be said that they have never been implemented as they stood, unless they have been preprocessed or compiled into the other form. This comes from the fact that the conventional hardware or machine language is not suitable to accept the hierarchical structure in original form of the system.

In this paper, we consider the direct implementation of the system where its original form represented in DEAPLAN is retained invariably. In the different point of view, we also consider the high level language machine in the more throughgoing way than the designers of the conventional machines, [4] ~ [17] , have done, since we do not transform the original representation into any intermediate one and we can implement not only each program but also the whole operating system.

From this standpoint, the hardware of our system becomes quite different from what are conventional including the current high level language machines. In our system, except channels and peripherals, the hardware only consists of a large number of quantum processing units (QPUs) which might be regarded as microminiaturized versions of the current microcomputers with writable control storage (WCS) as shown in Fig. 1. There is apparently neither CPU nor storage device which may be seen in the current computer system. Therefore, the main contemporary concepts, such as virtual storage, reenterability and multiplexing of CPU as well as compiling and linkage editing are all disappeared in our system.

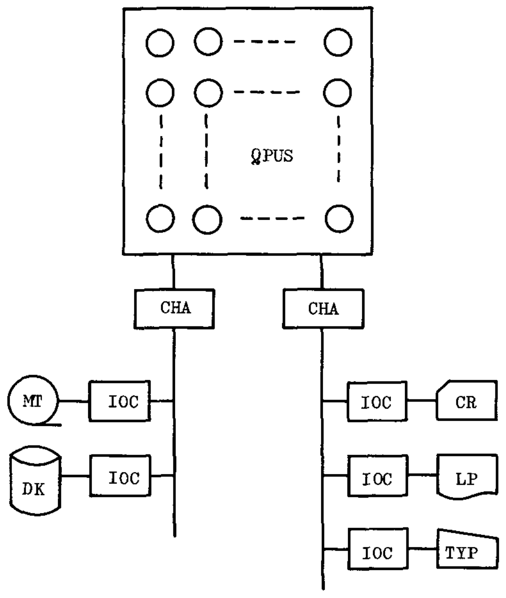


Fig. 1 Hardware System

In the following, we will first introduce the characteristics of DEAPLAN in section 2 and next the hardware requirements in section 3. Then, we will present our method in section 4 and finally in section 5 we will refer to the significance of our system.

2. BRIEF SKETCH OF DEAPLAN

To be easily understandable we will briefly sketch the characteristics of DEAPLAN by contrasting it with conventional programming languages such as PL/I.

At first the descriptive unit in PL/I is an external procedure which usually exists in the lowest layer of hierarchical systems. In the upper layer of operating systems there exist higher level modules than procedures such as load-modules, tasks, jobs, subsystems or entire system itself (in DEAPLAN an entity having acess to data is always called module). Such higher level modules can be also described as independent units of description in DEAPLAN.

Since it is usual that kinds or types of module are different among various operating systems, DEAPLAN provides no standard higher level type of module and instead allows one to define and introduce new types freely.

There are only three standard types: PROC (procedure/function), ST (statement) and OP (operator). As is already seen procedures/functions, statements and operators are also regarded as modules in DEAPLAN. These modules are introduced (when necessary) within their parent modules according to custom of programming languages like PL/I, although it may be a view that each of these modules should be described separately.

Second, the unit of description contains declarations of entries, parameters, data, constituent modules and execution logic (sequence of executable statements) as well as in PL/I. But in the declaration of constituent module one can define not only procedure module but also new OP or ST module and new data types may be introduced in the data declaration. DEAPLAN, therefore, is an extensible language [18] and also a very high level language [19].

In the case of higher level modules or external procedures access and call declarations (which may be considered declarations of the scope or the capability [21], [22]) are included besides those mentioned above in the unit of description. In the access declaration access relations between constituent modules and (inner, external) data are specified, while in the call declaration call relations among constituent/external modules being specified.

In the case of lower level modules which appear in programming languages it may be reasonable that the scope of data and modules is based on the block structure. But in the case of higher level modules the block becomes too wide to be practical and such two relational declarations seem to be necessary on the standpoint of security. W. Wulf even says that global variables in(block structured) programming languages are considered harmful.

There are two other declarations in DEAPLAN: space and map declarations. Since they are applicable only to the conventional computer system with storage devices, further explanation is omitted, although strictly speaking the map declaration may be used in the different manner (cf. 4).

The third characteristic of DEAPLAN is that when inner data or constituent modules are declared, design decisions concerning their inner structure may be defered. This enables us to do structured top-down designing.

Since, as already mentioned, ST primitives (modules) can be freely introduced, we are able to do it much efficiently. Further we should like to add that DEAPLAN has no BEGIN blocks, because new ST primitive fulfill the function of BEGIN block.

The fourth characteristic is that the concept of time span (storage attribute in PL/I) is applied not only to data but also to modules. When the whole system is considered this concept seems to be necessary, although it has no value in the case of the lower module as its constituent modules are always together with it.

Finally we will explain the fifth characteristic. It is considered in DEAPLAN that the logic declaration consists of a sequence of commands which activate specific (constituent, external) modules respectively.

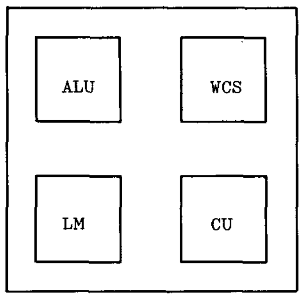
A command is a combination of the entry name of corresponding module and actual parameters. Although we have previously said that statements and operations in programming languages are kinds of modules, exactly speaking statements and operations are commands that activate corresponding ST and OP modules respectively. In our system the concept of command is the most fundamental. The logic declaration may be omitted in the case of higher level modules and is regarded as consisting of only one command which activates the main constituent module.

3. HARDWARE REQUIREMENTS

In this section we will discuss hardware requirements for our system. The hardware consists of only one kind of components called quantum processing unit (QPU). There are sufficiently large number of QPUs in the system. As is explained later, a set of QPUs enables us to represent the whole operating system as it is.

3.1 Quantum Processing Unit

A QPU may be regarded as the microminiaturized version of a current microcomputer and consists of arithmetic logical unit (ALU), local memory (LM), writable control storage (WCS) and communication unit (CU) as shown in Fig. 2. The actual function of each QPU is determined by the program stored in WCS (called quantum program). A QPU is assigned a unit (device) number and has communication with each other using CU.



ALU	:	Arithmetic Logical Unit
LM	:	Local Memory
WCS	:	Writable Control Storage
CU	:	Communication Unit

Fig. 2 Quantum Processing Unit

The DEAPLAN representation of system is distributively allocated on the large number of QPUs. The set of QPUs may, therefore, be considered a new sort of device which is unified and reorganized from the current storage units and CPUs.

3.2 Quantum Instruction

A quantum program which determines the specified function of QPU is formed by quantum instructions. The set of quantum instructions is common to all QPUs. The main instructions are shown below.

(1) Inter QPU communication
- a. activate/inform termination
- b. halt/inform halt
- c. enter interrupt directory/delete interrupt directory/inform interruption
- d. request module call
- e. request data access
- f. lock/unlock
- g. transfer data
- h. request creation(deletion)of data or module
- i. enter directory/delete directory/search directory
- j. get QPU/free QPU

(2) Arithmetic and control
- a. arithmetic operations
- b. logical operations
- c. compare/branch operations

In this section we will omit the further explanation of quantum instructions. These are discussed in the next section as occasion arises.

4. SYSTEM CONSTRUCTION METHOD

In order to understand easily we will explain our method using a relatively simple example. In Fig. 3(a) and (b) a program PROGX and its main external procedure P1 are described in DEAPLAN respectively. These modules are implemented without any transformation as shown in Fig. 4. In Fig. 4 a QPU is indicated by a circle and its unit number is placed on the shoulder of the circle. By the way since the DEAPLAN representation of module is implemented as it is, extremely speaking, neither linkage editor nor compiler is necessary in our system except a loader.

```
module PROGX PROGRAM; tspan controlled(USER-JOB);
       entry PROGXE;
       data (X, Y) bin(31); tspan static;
       data Z org 1 V char (8),
                  1 W bin(31); tspan static;
module P1 proc main; tspan static;
       entry main P1E;
       end P1;
module P2 proc; tspan static;
       entry P2E;
       end P2;
module P3 proc; tspan static;
       entry P3E;
       end P3;
access P1 (read (X, Y), write Z.W),
       P2 write X,
       P3 write Y;
call from P1 to P2, from P2 to P3;
end PROGX;
```

Fig. 3(a) Original Form of Program PROGX (in DEAPLAN)

```
module P1 proc (PROGX) main; tspan static;
       entry main P1E;
       data (A, B, C) char (8); tspan automatic;
       data ext (X,Y) bin (31);
       data ext Z org 1 V char (8);
                      1 W bin (31);
module Q1 proc;
       entry Q1E;
          .
          .
          .
       end Q1;
module Q2 proc;
       entry Q2E;
          .
          .
          .
       end Q2;
logic
          .
          .
          .
       P2E;
          .
          .
          .
       if A=B then Z.W=X else Z.W=X+Y;
          .
          .
          .
access ext P1 (read (X, Y), write Z.W);
call ext from P1 to P2 entry P2E;
end P1;
```

Fig. 3(b) Original Form of external procedure P1 (in DEAPLAN)

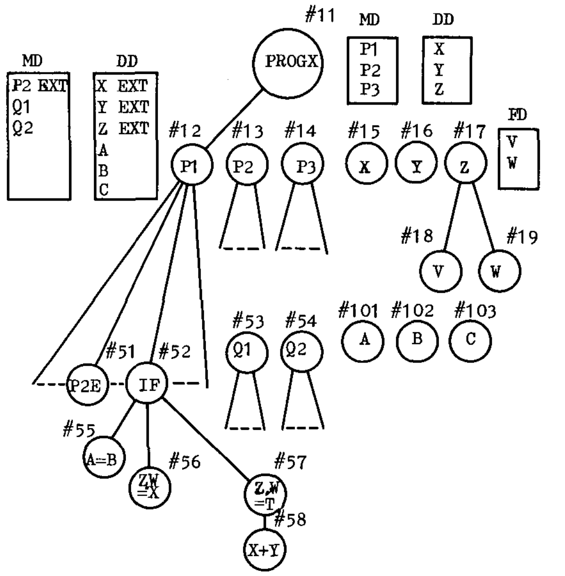


Fig. 4 Implementation of PROGX and P1

4.1 Module Creation

Each module is allocated on the different QPU without regard to its complexity level. In Fig. 4 program module PROGX and its constituent modules (external procedures) P1, P2 and P3 are allocated on QPU #11, #12, #13 and #14 respectively. Inner procedures Q1 and Q2 of P1 are on QPU #53 and #54 respectively.

Next according to the LOGIC declaration of P1, ST module IF, OP module = (compare), = (assign) and + (add) are allocated on QPU #55 ~ #58 respectively. Although we have regarded = (assign) as an OP module, it may be contained in the set of ST modules.

We note here that the object placed on QPU #51 is not a module but a command to external procedure P2. It is, therefore, not necessary originally since P1 should directly activate P2, but is left as is in Fig. 4 so that we might easily follow the flow of control.

Now in the case of higher level modules module directory (MD) and data directory (DD) are generated in LM of the QPU on which that module is allocated. Each MD entry contains a constituent or relevant external module name, type, time span, entry, QPU number,ID number, call attribute and so on.

ID number is provided so that we may uniquely identify each module or datum throughout the lifetime of whole system. It will be appropriate, for example, to make each created time of module or datum such an ID number. Each DD entry contains an inner or relevant external data name, type, time span, QPU number, ID number, access attribute, etc.

4.2 Data Creation

Data are also allocated on different QPUs in the similar manner as modules. The structured data or array such as Z is placed on more than one QPUs retaining its hierarchy and a field directory (FD) is generated in every LM of the upper level QPUs. Each FD entry has field name, type, QPU number, ID number, etc.

Since the time span of data A, B and C is AUTOMATIC, they are created when procedure Pl is activated and deleted as soon as Pl terminates its action. Accordingly QPU numbers and ID numbers assigned to data A, B and C may be different in every allocation.

4.3 System Behavior

We will explain the behavior of our system using the example of Fig. 4. When PROGX (QPU #11) is called it gives an activation instruction to Pl (QPU #12) at once and waits for the termination signal from Pl. The reason why PROGX could directly communicate with Pl thus from the first time is that the loader previously sets Pl's QPU number and ID number in PROGX. The lines in Fig. 4 means that such linkages are given at the beginning by the loader. The more information the loader is available, the more linkages it can staticly give at the allocation time. Otherwise linkages are dynamically done at the execution time.

Now when Pl is activated by PROGX it examines the directories, finds data A, B and C to be AUTOMATIC, allocates them on different QPUs and stores the QPU numbers and their ID numbers into DD of Pl. Then Pl gives an activation instruction to the QPU on which the module corresponding the first command in the LOGIC declaration is placed. When Pl receives the termination signal it activates the next QPU on which the module corresponding the second command is placed and waits for the signal.

Pl repeats this action until finally it receives the termination signal which is sent by the QPU on which the module corresponding the last command is placed. Pl then deletes data A, B and C and sends the termination signal to PROGX which in turn informs its caller that the whole operation is completed.

Next we will consider the case in which Pl activates P2E (QPU #51). Although P2E wants to call P2 (QPU #13), it does not know where P2 is because there is no static link between P2E and P2. P2E therefore gives an request call instruction to its parent Pl. After Pl recognizes validity of the request by checking the call attributes in MD, it then gives an request call instruction to its parent PROGX which in turn checks its validity using MD and sends Pl the QPU number and ID number of P2.

Then, Pl records the information in its own MD and transfer the QPU number and ID number of P2 to requester P2E, thus completing the dynamic linkage between P2E and P2. In addition, we may say that our module directory or data directory is a kind of capability list [21] , [22] .

Now assume that Pl activates IF (QPU #52). IF immediately initiates = (QPU #55) which in turn gives a request access instruction concerning data A and B to its parent IF. =(QPU #55) finally gets the QPU numbers and ID numbers of A and B after following the similar progress as mentioned above in the case of request call instruction.

When = (QPU #55) gives transfer data instructions to A (QPU #101) and B (QPU #102), each of them first checks ID number attached to each instruction. Generally ID number of data or module is kept in the QPU on which the data or module is placed. Therefore if ID number is always added to communication instructions the receiving QPUs may be able to detect the erroneous or illegal access.

Returning to the subject, when = (QPU #55) receives both values from A and B it compares them and inform the parent IF of the result. If the result is "l" B IF initiates = (QPU #56) otherwise = (QPU #57). Actions of the modules such as = (QPU #56), = (QPU #57) and + (QPU #58) may be also presumed. Here operand T of = (QPU #57) means the value which + (QPU #58) returns.

As already mentioned, in our system every linkage between QPUs is always examined its validity by the loader or higher level QPUs whether it is static or dynamic. Besides communication errors between QPUs are prevented by the use of ID numbers. Thus the security of data access and module activation is improved in our system.

On the other hand, there is the possibility of the lowering of access efficiency in the case of subscripted variables, elements of structured data or AUTOMATIC inner data. To avoid this situation one may specify the data allocation using MAP declaration in such a manner that these data are allocated on the continuously numbered QPUs. Then it will be possible to locate necessary QPUs based on a representative QPU number without any intervention of antecedent QPUs. In this case, however, it is inevitable that data security goes down because the same ID number must be given to each member of the data group.

5. SIGNIFICANCE OF OUR SYSTEM

At present it seems difficult either technically or economically to construct the system based on our method described in this paper. But we may expect in near future that the progress of LSI, hardware and software technology enables us to implement the system based on our method.

The significance of our system then is as follows:

(1) Sharp raise of marginal developing size of OS

At present the structured programming or the use of higher level systems programming languages is proposed in order to deal with operating systems which show a tendency to grow larger scale. It is unquestionable that these tools can relax the serious situation to a certain extent. But they do not seem good enough to solve the problem fundamentally.

To the contrary, our method enables the system to be implemented without any transformation of its (so to speak) three-dimensional logical structure, and then the whole system becomes transparent to designers. Therefore the upper limit of the system size may be drastically raised in our method.

(2) Ease of growth or extension

As mentioned above, since the whole system is constructed in a three-dimensional and transparent manner, it is easy to modify or extend the system in order to adapt it to the change in the situation.

(3) Improvement of system security

The system is divided into very small parts and is distributed on the large number of QPUs. Thus each data or module on the QPU becomes a unit of protection.

Moreover the implemented hierarchical structure keeps in itself the execution locus of the system. These feature seems to improve the system security.

(4) Improvement of man-machine interface

As is evident from (1), sharp raise of the marginal size of OS makes a room for introduction of more intelligent features into the system so that we may further push the man-machine interface to the human side.

(5) Contribution to the relevant area

If the QPU would have any relation with the neuron our system might contribute in some degree to the relevant area such as bionics, nervous-physiology, pattern recognition and artificial intelligence.

6. CONCLUSION

We have formerly developed a system design language DEAPLAN based on the structuralism. In this paper we have proposed a method in which the DEAPLAN representation of any system has been directly implemented as it is without either modification or transformation. Extremely speaking, therefore, neither compiler nor linkage editor is necessary in our system except a kind of loader.

Our hardware consists of a large number of quantum processing units (QPUs) which are microminiaturized versions of the current microprocessors with WCS. Using the QPU network we can give the implemented system the same structure as in the original DEAPLAN representation which is never given by the conventional computer hardware.

The identify of implemented version of the system with the original form in DEAPLAN seems to give a more fundamental solution to the problem of the rapid growth of operating systems compared with the structured programming or the use of higher level systems programming languages.

In addition, our method using only one kind of element (QPU) similar to the neuron seems to contribute in some degree to the relevant area such as bionics, nervous-physiology, pattern recognition, and artificial intelligence.

ACKNOLWEDGEMENT

The author wishes to express his appreciation to Prof. K. Inoue for his useful advice given when he has been in this company.

REFERENCES

[1] Dijkstra, E. W.: "Notes on Structured Programming", Technological U. Eindhoven, The Netherlands, August, 1969

[2] Organick, E. I.: "The Multics System: An Examination of Its Structure", MIT Press, 1972

[3] Scherr, A. L. et al.: "Functional Structure of IBM Virtual Storage Operating Systems", IBM Syst. J. Vol. 12, No. 4, pp 368 ~ 411, 1973

[4] Rice, R. & Smith, W. R.: "SYMBOL--A Major Departure from Classic Software Dominated Computing System", Proc. SJCC Vol. 38, pp 575 ~ 587, 1971

[5] Smith, W. R. et al: "SYMBOL--A Large Experimental System Exploring Major Hardware Replacement of Software", Proc. SJCC Vol. 38, pp 601 ~ 616, 1971

[6] Bashkow, T. R., Sasson, A. & Kronfeld, A.: "System Design of a FORTRAN Machine", IEEE Trans. on Electronic Computers, Vol. EC-16, No. 4, pp 485 ~ 499, 1967

[7] Anderson, J. P.: "A Computer for Direct Execution of Algorithmic Languages" FJCC, pp 184 ~ 193, 1961

[8] Mullery, A. P., et al.: "ADAM--A Proglem-Oriented Symbol Processor" SJCC, pp 367 ~ 380, 1963

(9] Melbourne, A. J., et al: "A Small Computer for the Direct Processing of FORTRAN Statements", Computer Journal, Vol.8, No.1, pp 24 ~ 27, 1965

[10] Weber, H.: "A Microprogrammed Implementation of EULER on the IBM System/360 Model 30", CACM Vol. 10, No. 9, pp 549 ~ 558, 1967

[11] Sugimoto, M.: "PL/I Reducer and Direct Processor", 24th National Conf. of ACM, pp 519 ~ 538, 1969

[12] Thurber, K. J., et al.: "Systems Design for a Cellular APL Computer", IEEE Trans. on Computers Vol. C-19, No. 4, pp 291 ~ 300, 1970

[13] Zaks, R. et al.: "A Firmware APL Time-Sharing System", SJCC, pp 179 ~ 190, 1971

[14] Hassit, A., et al.: "Implementation of a High Level Language Machine", 4th Annual Workshop on Microprogramming, 1971

[15] Shapilo, M. O.: "A SNOBOL Machine: A High-Level Language Processor in a Conventional Hardware Framework"

[16] Wilner W. T.: "Design of Burroughs B 1700", FJCC pp 489 ~ 497, 1972

[17] Wilner, W. T.: "Burroughs B 1700 Memory Utilization", FJCC, pp 579 ~ 589, 1972

[18] Proc. of the international symposium on extensible languages, SIGPLAN Notices Vol. 6, No. 12, 1971

[19] Proc. of the international symposium on very high level languages, SIGPLAN Notices Vol. 9, No. 4, 1974

[20] Hayashi, T.: "DEAPLAN--A Design and Implementation Languages for Operating Systems", Journal of the Information Processing Society of Jpana. Vol. 14, 1974

[21] Dennis, J. B. & Van Horn, E. C.: "Programming Semantics for Multiprogrammed Computations", CACM Vol. 9, No. 3, pp 143 ~ 155, 1966

[22] Wulf, W. et al.: "HYDRA: The Kernel of a Multiprocessor Operating System", CACM, Vol. 17, No. 6, pp 337 ~ 345, 1974