DEAPLAN-a Design and Implementation Language for Operating Systems

Tatsuya Hayashi*

1. INTRODUCTION

In this paper a design and implementation language, DEAPLAN, is described. This language is designed so that we may describe not only each program (element) but also the upper parts of the hierarchical structure in operating systems which can never be described in conventional programming languages.

DEAPLAN is also designed to be suitable for use in structured top-down design of the whole operating systems.1)~4) It requires no prior condition concerning the structure of operating systems. We may say that DEAPLAN is a kind of the extensible language5) and also one of the very high level languages.6)

In the following, we will first formalize the logical structure of operating systems in order to clarify the major features that seem to be necessary in such design languages. Then we will discuss the characteristics of DEAPLAN which is followed by the example description.

2. LOGICAL STRUCTURE OF OPERATING SYSTEMS

In this paper we consider Space, Datum and Module to be the essential entities that constitute operating systems. A module is an entity that executes given logical function, having one of the various complexity levels. Procedure in programming language is one of the modules and the whole operating system itself is also considered a module with the highest level of complexity.

A module generally has its internal structure and consists of internal data and constituent modules. Operating systems thus form the hierarchical structures. On the other hand, since spaces, data and modules exist in operating systems as mentioned above, the following three relations arises among them: (1) mapping (or allocation) relation: space ~ data/module, (2) access relation: data ~ module, (3) call relation: module ~ module.

This paper first appeared in Japanese in Joho—Shori (Journal of the Information Processing Society of Japan), Vol. 14, No. 9 (1973), pp. 652~660.

* Software Planning Department, Fujitsu Limited, Japan

The logical function of module M is described by the sequence of commands where a command means a calling (or activation) instruction to either constituent module or external module of M. Each format of the command is defined in the module which is called by that command.

An entry is formed of entry name and none or certain number of formal parameters. A command is obtained from the corresponding entry by replacing formal parameters with actual ones. Every command activates the execution of corresponding logical function. The logical function of any module which is called by the corresponding command may be further divided into some small functions. Consequently the logical function of module M would be finally represented using undividable basic modules. We consider the modules that correspond to statements (if ~, do ~, etc.), operations (~ + ~, ~ * ~, etc.) or procedure/function calls (ABS (~), etc.) in conventional programming languages to be suitable for such basic modules.

In this paper statements, operations and procedure/function calls are regarded as the commands to the modules whose types are ST, OP and PROC respectively. In the case of ST and OP actual parameters are passed by call by name and in the other cases by call by value.

3. CHARACTERISTICS OF LANGUAGE

In this section we will explain the basic characteristics of DEAPLAN.

3.1 Descriptive Format

A descriptive unit is a module in DEAPLAN. In that case each constituent module except statement, internal procedure and operator is represented in a different descriptive unit from its parent's.

Now the descriptive format (called external module declaration) is shown using eight kinds of declarations as follows: MODULE module name, module type (parent module name) ; entry declarations; data declarations; module declarations; logic declaration; space declarations; map declarations; access declarations; call declarations, END module name.

Module declarations effectively correspond to so called blocks in programming languages concerning the scope of names and the declaration of the constituent module is considered to be an immediate descendant block of its parent's declaration. But the scope of declaration of the module which lies in the upper layer of operating systems is generally so wide that access and call declarations seem to be necessary in order to clarify and limit the scope of individual entities.

3.2 Kinds of Modules

It may be said that types of module already established as standards are restricted to those which appear in programming languages and correspond to statements, procedures and operators. We call these types ST, PROC and OP respectively and also make them as standards in DEAPLAN. Besides that the system design language must allow arbitrary high level types of module to be defined and introduced.

ST module corresponds to text macro and BEGIN block in programming languages. We have no macro and no BEGIN block in DEAPLAN from the standpoint of structured top-down design.

In DEAPLAN there are almost the same basic modules of ST, PROC and OP as in PL/I or BLISS7). DEAPLAN is an extensible language since designers can arbitrarily define new modules of these standard types.

3.3 Kinds of Data

Data types in programming languages such as in PL/I are also necessary in design languages. Besides that ability to define any data or data type whose internal structure is undefined is absolutely necessary since in early stage it is neither easy nor appropriate to decide the internal structure. The internal structure which will be defined in later stage may be contained in the declaration of the constituent module that actually needs it, as follows: DATA [TYPE] EXT REFINED data (data type) name ORG; ELM

3.4 Time Span of Data and Modules

Generally modules (and spaces) do not always exist in the system continuously from the beginning to the end of its activation period as well as data. For the reason of efficiency or logical constraint they are dynamically created and deleted. Hence it is desirable that each time span of data or modules might be specified in design languages. This is done in DEAPLAN such that:

$$\text{TSPAN}\left\{\begin{array}{l}\text{STATIC}\\ \text{AUTOMATIC}\\ \text{CONTROLLED}\quad [(\text{module name})]\end{array}\right\}$$

DEAPLAN representation of simple model operating system MOS is given in Fig. 1.

```
MODULE TYPE SYSTEM % WHOLE OPERATING SYSTEM %;
MODULE MOS SYSTEM;
  ENTRY EOS;
  DATA JOB-STREAM ORG; TSPAN CONTROLLED (IN-CTRL);
  DATA SYSIN ORG; TSPAN CONTROLLED (IN-CTRL, EXEC-CTRL);
  DATA (INPUT-QUEUE, OUTPUT-QUEUE) ORG; TSPAN STATIC;
  DATA SYSOUT ORG; TSPAN CONTROLLED (EXEC-CTRL, OUT-CTRL);
  DATA LIST-STREAM ORG; TSPAN CONTROLLED (OUT CTRL);
  DATA (SYSCATLG, SYSACCT, SYSSVCLIB, SYSEXECLIB, SYSJCLLIB) ORG; TSPAN STATIC;
  DATA CSCB ORG; TSPAN CONTROLLED (CONSOLE-COMMUNICATION, MASTER-CTRL, COMMAND-
       PROCESSOR);
  DATA CSCB-QUEUE ORG; ELM 1 (FIRST-CSCB, LAST-CSCB) PTR (CSCB); TSPAN CONTROLLED
       (MASTER-CTRL);
  DATA USER-FILE ORG; TSPAN CONTROLLED (EXEC-CTRL);

  MODULE TYPE NON-TASK % SUPERVISOR %;
  MODULE TYPE TASK;
  MODULE TASK-CTRL NON-TASK; TSPAN STATIC;
         ENTRY ATTACH, DETACH, LINK, XCTL, RETURN, WAIT, POST, PGM-INTERRUPT;
         END TASK-CTRL;
  MODULE IO-CTRL NON-TASK; TSPAN STATIC;
         ENTRY EXCP, IO-INTERRUPT; END IO-CTRL;
  MODULE MASTER-CTRL TASK MAIN ; TSPAN STATIC;
         ENTRY EMC; END MASTER-CTRL;
  MODULE CONSOLE-COMMUNICATION TASK;
         TSPAN CONTROLLED (MASTER-CTRL);
         ENTRY ECC; END CONSOLE-COMMUNICATION;
  MODULE COMMAND-PROCESSOR TASK;
         TSPAN CONTROLLED (MASTER-CTRL);
         ENTRY ECP; END COMMAND-PROCESSOR;
  MODULE IN-CTRL TASK;
         TSPAN CONTROLLED (COMMAND-PROCESSOR);
         ENTRY EIC; END IN-CTRL;
  MODULE EXEC-CTRL TASK;
         TSPAN CONTROLLED (COMMAND-PROCESSOR);
         ENTRY ECC; END EXEC-CTRL;
  MODULE OUT-CTRL TASK;
         TSPAN CONTROLLED (COMMAND-PROCESSOR);
         ENTRY EOC; END OUT-CTRL;
  MODULE JOB-PROCESSOR TASK; TSPAN CONTROLLED (EXEC-CTRL);
         ENTRY EJP; END JOB-PROCESSOR;


  SPACE TYPE CR % CARD %, DA % DIRECT ACCESS STORAGE %, LP % LINE PRINTER SHEET %;
  SPACE BASE DECKS CR, SHEETS LP;
  SPACE BASE SYSTEM-RESIDENCE-VOLUME DA (1 VOL, (1 VOL = 200 CYL, 1 CYL = 10 TRK,
             1 TRK  3000 BYTE)),
             SYSTEM-WORK-VOLUMES DA (2 VOL),
             USER-VOLUMES DA (10 VOL);
  SPACE BASE HCM MEM (1024 KB, (1 KB = 1024 BYTE, 1 BYTE = 8 BIT));
             ELM 1 RESIDENT-AREA        192 KB,
                 1 SYSTEM-WORK-AREA      64 KB,
                 1 TRANSIENT-AREA       768 KB;
  MAP JOB-STREAM ON DECKS, LIST-STREAM ON SHEETS;
  MAP (SYSIN, SYSOUT) ON SYSTEM-WORK-VOLUMES,
      (INPUT-QUEUE, OUTPUT-QUEUE) ON SYSTEM-WORK-VOLUMES BDRY 1 CYL;
  MAP (SYSCATLG, SYSACCT, SYSSVCLIB, SYSEXECLIB, SYSJCLLIB) ON SYSTEM-RESIDENCE-
      VOLUME;
      USER-FILE ON USER-VOLUMES;
  MAP (CSCB, CSCB-QUEUE) ON SYSTEM-WORK-AREA;
  MAP (TASK-CTRL, IO-CTRL, MASTER-CTRL, CONSOLE-COMMUNICATION, COMMAND-PROCESSOR) ON
      RESIDENT-AREA BDRY 8 BYTE;
  MAP (IN-CTRL, EXEC-CTRL, OUT-CTRL, JOB-PROCESSOR) ON TRANSIENT-AREA BDRY 4 BYTE;
  ACCESS IN-CTRL (USE (JOB-STREAM, SYSJCLLIB), SET SYSIN), JOB-PROCESSOR (USE SYSIN,
         UPDATE USER-FILE, SET SYSOUT), OUT-CTRL (USE SYSOUT, SET LIST-STREAM);
  ACCESS (IN-CTRL, EXEC-CTRL) UPDATE INPUT-QUEUE, (EXEC-CTRL, OUT-CTRL) UPDATE
         OUTPUT-QUEUE;
  ACCESS EXEC-CTRL UPDATE (SYSCATLG, SYSACCT), TASK-CTRL USE (SYSSVCLIB, SYSEXECLIB);
  ACCESS (MASTER-CTRL, CONSOLE-COMMUNICATION) UPDATE (CSCB-QUEUE, CSCB), COMMAND-
         PROCESSOR USE CSCB;

  CALL FROM MASTER-CTRL TO (CONSOLE-COMMUNICATION, COMMAND-PROCESSOR),
       FROM COMMAND-PROCESSOR TO (IN-CTRL, EXEC-CTRL, OUT-CTRL),
       FROM EXEC-CTRL TO JOB-PROCESSOR;
  CALL TO (TASK-CTRL, IO-CTRL) FROM (MASTER-CTRL, CONSOLE-COMMUNICATION, COMMAND-
       PROCESSOR, IN-CTRL, EXEC-CTRL, OUT-CTRL, JOB-PROCESSOR);


END MOS;
MODULE CONSOLE-COMMUNICATION TASK (MOS);
  ENTRY ECC;
  DATA EXT REFINED CSCB ORG;
           ELM 1 NXTCSCB     PTR (CSCB),
               1 STATUS      BIT(8),
                 2 PENDING   BIT(1),
                 2 *         BIT(7),
               1 COMMAND     ORG,
                 2 CODE      UBIN(8),
                 2 OPERAND-LENGTH BIN(7),
                 2 OPERAND-IMAGE  CHAR(*);
           TSPAN CONTROLLED (COMMAND-SCHEDULER, COMMAND-PROCESSOR, MASTER-CTRL);
  DATA EXT CSCB-QUEUE ORG;
       ELM 1 FIRST-CSCB PTR (CSCB),
           1 LAST-CSCB  PTR (CSCB);
       TSPAN CONTROLLED (MASTER-CTRL);
  MODULE TYPE PGM % LOAD MODULE %;
  MODULE CONSOLE-WAIT PGM; TSPAN CONTROLLED (MASTER-CTRL); ENTRY ECW EQU ECC;
         END CONSOLE-WAIT;
  MODULE CONSOLE-IO PGM; TSPAN CONTROLLED (CONSOLE-WAIT); ENTRY ECI; END CONSOLE-IO;
  MODULE COMMAND-SCHEDULER PGM; TSPAN CONTROLLED (CONSOLE-IO); ENTRY ECS;
         END COMMAND-SCHEDULER;

  ACCESS EXT COMMAND-SCHEDULER (SET CSCB, UPDATE CSCB-QUEUE);

  CALL EXT FROM MASTER-CTRL TO CONSOLE-WAIT;
  CALL FROM CONSOLE-WAIT TO CONSOLE-IO,
       FROM CONSOLE-IO TO COMMAND-SCHEDULER;
END CONSOLE-COMMUNICATION;
```

Fig. 1 An example description by DEAPLAN

4. CONCLUSION

In this paper we have discussed about the design and implementation language for operating systems and have presented DEAPLAN. This language requires no prior condition concerning the structure of operating systems and has the following facilities: (1) ability to define and use any module and/or module type whose logical level is higher than that of a Procedure appearing in conventional programming languages, (2) ability to define and use any module whose internal structure is undefined, (3) ability to define and use any data and/or data type whose internal structure is undefined, (4) ability to provide appropriate means for highly efficient structured top-down design, (5) ability to describe storage allocation and to clarify and/or restrict interface among modules and data. Using such design languages, it is hoped that we can take a step toward the computer aided design of operating systems.

ACKNOWLEDGEMENT

The author wishes to express his appreciation to Prof. K. Inoue for valuable discussions.

REFERENCES

1) Dijkstra E. W.: Notes on structured programming, EWD 249, Technical U. Eindhoven, The Netherlands (1969).

2) Dijkstra E. W.: A constructive approach to the problem of program correctness, BIT 8, pp. 174 ~ 186 (1968).

3) Wirth N.: Program Development by Stepwise Refinement, CACM, vol. 14, No. 4, pp. 221 ~ 227 (1971).

4) Snowdon P. A.: PEARL: An Interactive System for the Preparation and Validation of Structured Programs, ACM SIGPLAN Notices, vol. 7, No. 3, pp. 9 ~ 26 (1972).

5) Proc. of the international symposium on extensible languages, SIGPLAN Notices, vol. 6, No. 12 (1971).

6) Proc. of the international symposium on very high level languages, SIGPLAN Notices, vol. 9, No. 4 (1974).

7) Wulf W. A., Russel D. B. & Haberman A. N.: BLISS: A Language for Systems Programming, CACM vol. 14, No. 12, pp. 780 ~ 790 (1971).