An Implementation of the Operating System (SOS) Realizing the Cooperating Process in ALGOL or FORTRAN

Norihisa Doi*

1. INTRODUCTION

Computer education of universities is in a steady progress in Japan. In Keio University not only the introductory course of programming but also many other lectures on the information science are available for every student since the Institude of Information Science was established five years ago.

One of those courses is of the operating system. In recent years the operating system has become bigger and bigger and more and more complicated. It is very difficult to teach the essence of big and complicated operating system in a short period. We therefore think such a system is necessary that allows students to grasp the general concept of the operating system (even partially) and to realize a cooperating process to a certain extent using high level languages. This is the standpoint from which we developed a model of operating system which realizes the purposes in the environment of ALGOL and FORTRAN.

Here we describe the operating system SOS (The Structured Operating System). It is designed basically on the nucleus extension approach[1),2)], but it also involves an attempt to separate and clarify the functions which have conventionally been mingled together indifferently, in other words to structurize the system.

2. STRUCTURE OF NUCLEUS

Nucleus is a level of abstraction to abstract the processor.[3)] The structure of the nucleus is shown in Fig. 1 along with main resources and primitives. It shows a multi-layer structure with several levels which are marked off with the horizontal lines. Each level consists of a set of related functions. Some of the functions and resources belonging to each level can be referred to from higher levels but not from lower levels.

level	respon-sibility	abstraction	resources	primitives
0	process	process	stateword current-process	storestateword setstartpoint executeprocess
1	domain	basic system of process	process table noofprocess	createprocess
				sibling, relatives ascendant, daughter
				deleteprocess
				readdescendant readsibling
			domain	createobject
				inherent, grant
				checkdomain
2	scheduling	quasi virtual processor	timer	timer, hold
			readylist noofready-process noofactive-process	inreadylist outreadylist
				schedule
				suspend/release
				wakeup/block
				start/stop
3	synchronous operation	cooperating system		succ
				semaphore semaphorearray
				fork
				signal/wait
				await/cause
				chunkp/chunkv

Fig. 1 The structure of the nucleus

This paper first appeared in Japanese in Joho-Shori (Journal of the Information Processing Society of Japan), Vol. 16, No. 1 (1975), pp. 7~14.

* Keio Institute of Information Science, Keio University.

There are three kinds of level classification as described below. The first one classifies levels (levels 0 to 3) according to responsibility. It supports the abstraction of processor and gives the logical essentials of the levels of the abstraction of processor. In other words the processor is abstracted by means of responsibility classified into levels. To carry out each responsibility, corresponding resources, particularly a data base, are needed. In this sense this classification of levels is an abstraction corresponding to resources and the responsibilities are the results of the abstraction. The second classification consists of functional levels with resources, which constitute the levels of abstraction of responsibilities, divided finely. This classification clarifies the process of abstraction up to the preceding stage and enables stepwise design. The third classification is purely functional.

Primitives and ordinary operations are organized by a pile-up method which piles up operations of low primitivities on the basis of high primitivity operations. Primitives are the operations which are used in levels higher than the level to which the operations belong. The primitivity is the degree of abstraction of a primitive. For example, fork which is a primitive to initiate an asynchronous operation consists of storestateword, createprocess, setstartpoint and start (Fig. 2[8]). Principal primitives and primitives used to realize the principal ones are shown in Fig. 3.

The development and improvement are very easy since operations of lower primitivities are realized organically, resources are made to belong to each level exclusively, and multilayer structure with many levels are employed.

```
type  P=1‥maximum number of process;
      D=‥1maximum number of capabilities;
procedure fork
  (var i: P; pname: alpha; startpoint: addres; p: integer;
  d: array [D] of integer);
begin
  storestateword;
  i:=createprocess (pname, p);
  setstartpoint (startpoint, process[i]. stateword);
  start (i, d);
end
```

Fig. 2 primitive *fork*

primitives	primitives used to construct the primitive
schedule	executeprocess, setcurrentprocess, hold
suspend	storestateword, modifysuspended, ascendant, daughter, sibling, modifyactiveprocess, schedule
release	modifysuspended, ascendant, daughter, sibling, modifyactiveprocess
wakeup	setstate, setwws, ascendant, daughter, sibling, inreadylist
block	storestateword, setstate, setwws, daughter, outreadylist, schedule
start	storestateword, setstate, daughter, inreadylist, schedule
stop	storestateword, setstate, ascendant, outreadylist, schedule
fork	storestateword, setstartpoint, createprocess, start
signal	succ, release
wait	storestateword, suspend, succ

Fig. 3 The state of the pile of primitives

3. SCHEDULING

A system of processes are formed under the nucleus described in Chapter 2. A process of the system can stay in one of five states: dead, dormant, ready, blocked and running. Transition between the states and associated premitives are shown in Fig. 4. The states of ready and blocked are further divided into the states of suspended and unsuspended. Transition between these states are illustrated exactly in relation to the operations

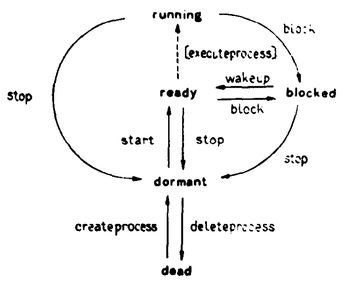

Fig. 4 The states of processes and the primitives that cause the transitions between them

and state variables of the system in Fig. 5. Actually the running state is considered as a part of the ready state. In synchronous operations other than wakeup/block, therefore, transitions occur between the states of ready unsuspended and ready suspended. Short-term scheduling made in the nucleus selects a process of the highest priority out of the processes under the ready unsuspended state and allots the processor to the process. This is a way of coexistence for all synchronous operations.

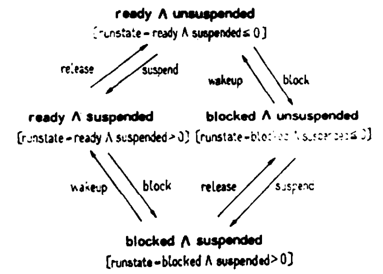
ready ∧ unsuspended
[runstate = ready ∧ suspended ≤ 0]
release
suspend
wakeup
block
ready ∧ suspended
[runstate = ready ∧ suspended > 0]
blocked ∧ unsuspended
[runstate = blocked ∧ suspended ≤ 0]
wakeup
block
release
suspend
blocked ∧ suspended
[runstate = blocked ∧ suspended > 0]

Fig. 5 The relation between ready/blocked, suspended/unsuspended, and the primitives.

SOS which realizes pseudo-parallel processing in the language processor is processed as a process in the whole system. If no process is registered in the readylist or processes entered in the readylist are all under the ready suspended state, the system enters into an endless loop and SOS does not work as an operating system any more. To prevent such a catastrophe, time and a process can be assigned so that the processor be allotted forcibly to the assigned process after the designated time has elapsed. (Strictly speaking, SOS abandons the processor voluntarily for the designated span of time.)

4. PROTECTION

At present SOS protects processes, reference to procedures (including primitives defined in the nucleus) and address locations, applying the concept domain.[5]

The rights of control and communication between processes are managed with a list which holds the hierarchical relations of processes. The domain of process involves the reference to procedures. Potential **capability** of process on reference can be defined with primitive inherent. Procedures defined in user programs can be defined as the object of capability with createobject. The right of ownership belongs to the defined process. A process which has the right of ownership of an object can give the right of use of the object to another process (grant). A parent process can transfer the capability which is assured by its right to children processes (as the parameters of start) at the beginning of the execution of the children processes. Procedures defined with createobject can check the right of use using checkdomain.

Concerning the reference of addressed locations, the definition of potential capability of procedure uses the character of language.

5. PROCESSING OF STATEWORDS

When using primitives which may transfer control to other process, such as start, stop and wait, it is necessary to save the statewords[4] which are effective at the time of referring the primitives. For this purpose, a procedure which saves main registers is made with the assembler. At the time of collection, the entry point of the procedure of these primitives is replaced with the instruction which calls the saving procedure and the instruction at the entry point of the primitives is inserted in the entry point of the saving procedure. This makes it possible to switch processes.

6. APPLICATION

SOS does not allow essentially for the same code to run simultaneously as different processes. Such a necessity, if arises, can be generally met by preparing the codes as many as the processes.

As an example of the application of SOS, parts of the simulation program (FORTRAN) of the Five Dining Phylosophers[6] are shown in Figs. 6 and 7.

The algorithm for philosopher 1 is given in Fig. 6.[7] MODIFY is a procedure to modify the number of forks available for neighboring philosophers. TT and TTT are the times they start and end eating spaghetti. Fig. 7 shows a part of the scheduler (a process) which controls the philosopher process which has been generated and initiated, using a time table. The scheduler and philosopher process are synchronized with semaphore MASTER and array semaphore SEM. Fig. 8 shows a part of the results of the execution. The figures enclosed with asterisks indicate the philosophers who have started eating, and two figures which correspond to TT and TTT respectively follow.

```
C
C
C      KENJA-1
C
C
   10 CONTINUE
      T  = RANDU(IV)*10.
      IF( T  .EQ. 0 ) GO TO 10
      CALL SIGNAL(MASTER)
      P = P1
      CALL WAIT(SEM(1,P1))
      CALL WAIT(FSEM)
   12 IF( FORKS(1) .EQ. 2 ) GO TO 11
       CALL AWAIT(EVENT,FSEM)
       GO TO 12
   11  CONTINUE
       CALL MODIFY(1,-1)
       CALL SIGNAL(FSEM)
C
   13 CONTINUE
      T = RANDU(IV)*10.
      IF( T .EQ. 0 ) GO TO 13
      CALL TIMER(TT)
      TTT = TT+T*100
      WRITE(6,100) TT,TTT
  100 FORMAT(/' * 1 *',I15,I15/)
      CALL SIGNAL(MASTER)
      P = P1
      CALL WAIT(SEM(1,P1))
C
      CALL WAIT(FSEM)
      CALL MODIFY(1,1)
      CALL CAUSE(EVENT,FSEM)
      CALL SIGNAL(FSEM)
C
      GO TO 10
C
C
```

Fig. 6 The algorithm for the phylosopher 1.

```
C
C
C
      CALL FORK(P1,'KENJA1',$10,3,D)
      CALL FORK(P2,'KENJA2',$20,3,D)
      CALL FORK(P3,'KENJA3',$30,3,D)
      CALL FORK(P4,'KENJA4',$40,3,D)
      CALL FORK(P5,'KENJA5',$50,3,D)
C
C
C     SCHEDULER
C
C
      DO 7 K = 1,10
       TIMETL(K) = 2**34
       MARK(K) = .FALSE.
    7  CONTINUE
    5 CONTINUE
      CALL WAIT(MASTER)
C
      IF( .NOT. ENFLAG ) GO TO 1002
       CALL SIGNAL(MASTER)
       ENFLAG = .FALSE.
       GO TO 1000
 1002 CONTINUE
C
      CALL TIMER(TT)
      TIMETL(P) = TT+T*100
      MARK(P) = .TRUE.
C
      SW = .FALSE.
    6 CONTINUE
      J = 3
    4 CONTINUE
      IF( J .GT. 7 ) GO TO 1
       IF( .NOT. MARK(J) ) GO TO 2
        IF( TT .LT. TIMETL(J) ) GO TO 3
         MARK(J) = .FALSE.
         CALL SIGNAL(SEM(1,J))
         SW = .TRUE.
    3    CONTINUE
    2   CONTINUE
        J = J+1
        GO TO 4
    1  CONTINUE
       IF( (READYP .EQ. 1) .AND. (.NOT. SW) ) GO TO 1000
        ENTIME = 2**34
        DO 1001 I = 3,7
        IF( .NOT. MARK(I) ) GO TO 1001
         IF( ENTIME .LE. TIMETL(I) ) GO TO 1001
          ENTIME = TIMETL(I)
 1001     CONTINUE
          CALL TIMER(TT)
          ENTIME = ENTIME-TT
          GO TO 5
 1000   CONTINUE
        CALL TIMER(TT)
        SW = .FALSE.
        GO TO 6
C
```

Fig. 7 The scheduler for the phylosophers' system.

```
                * 2 *      62979668        62979968

                                            * 5 *      62979779        62980479

                           * 3 *      62979995        62980195

                * 2 *      62980216        62980516

                                      * 4 *      62980507        62981307

 * 1 *      62980541        62980641

                * 2 *      62980745        62981645
```

Fig. 8 The result of the simulation of the dining phylosophers' system.

7. PROBLEMS ON THE INCORPORATION OF THE FUNCTIONS OF OPERATING SYSTEM

The functions of operating system should be generally involved in the programming language itself as its functions or elements like reserved words. In our attempt, however, the incorporation of OS functions is restricted because a high level language is used to make our system without modifying the language processor. Although it is necessary to realize as natural one as possible, it is impossible to make completely the same one on the processing of queues and so on. Two important problems must be considered for it. The first one is the form of reference. If ALGOL or FORTRAN is used, however no means are available other than the procedure (or,subprogram) to implement function. The second problem is how to realize the functions, which influences the easiness of use. When realizing the functions of OS on a programming language, we cannot help indicating concretely what the operating system should tacitly perform originally. We simplified the method of implementation and made effort to incorporate the functions in a form as natural as possible. But in some primitives, potential functions are seen explicitly.

8. PROBLEMS ON REALIZING OS WITH ALGOL/FORTRAN

The scheduler is the greatest problem on realizing an operating system using ALGOL/FORTRAN. SOS may stop functioning depending on the states of processes of SOS because SOS itself is regarded as a process as described before. It is necessary to devise a means of escapement (see 3).

In the case of ALGOL, switching of processes involves problems. One is due to the OS functions formed into procedures. When a primitive which may cause to switch from a process to another is used, the scheduler may be called. In such a case, control is transferred from the scheduler to the process through executeprocess. Because ALGOL procedures are designed to allow recursive call, stack grows more and more, which may require considerable area. In the second, simple saving and restoration of registers which follow the switching of processes may not satisfactory, because, for example UNIVAC ALGOL (SIMULA) allocates the pointers dynamically at the execution time even to data areas which are most non-local declared in the outermost loop.

The block structure (and own) is convenient for the definition of potential capability of procedure in relation to the reference of addressed locations. But no means are available which inhibit the reference of data which are non-local for the block. Concerning this point, the FORTRAN version avoids the problem partially by using labelled common blocks.

9. CONCLUSION

We have described a method to realize the cooperating process using existing languages, ALGOL and FORTRAN, which do not allow parallel processing, and the operating system needed for the purpose. The operating system enables to carry out a demonstration of the latest principles, though more or less unnaturally. It is also very useful as a teaching material.

On the design of the operating system, we attempted an approach to structurize the system. This method makes it easy to understand, design, develop and maintain the system, although it involves disadvantage on overhead.

The memory areas needed for SOS (FORTRAN version) are given in Fig. 9 for reference.

	No. of FORTRAN statements*	Instruction code (words)	Data (words)
level 0		227	
level 1	149	477	187
level 2	158	494	170
level 3	119	531	142
dat base			1,904
miscellaneous**		3,702	2,156
total	426	5,431	6,715

(a) The summary of the SOS

	No. of FORTRAN statements*	Instruction code (words)	Date (words)
semaphore	10	30	12
signal	18	68	27
wait	19	71	25
await	9	56	14
cause	13	73	18

(b) The size of the typical primitives.

* FORTRAN program part (including debugging aids)
** FORTRAN libraries

Fig. 9

Acknowledgement

The author owes very much to Mr. Yoshio Ohno and Mr. Kiichi Yamamoto of Keio Institute of Information Science and Mr. Mitsukuni Hasegawa of Nippon Univac Kaisha, Ltd. on carrying out our plan. The author also wishes to express his deep appreciation to Prof. Yoshio Hayashi.

Reference

1) Brinch Hansen, P. : "The Nucleus of a Multiprogramming system," Comm. ACM, Vol. 13, No. 4, pp. 238-241, 250 (1970).

2) Takahashi, H., Kamada, H. : An Approach to the Design of Operating System, Information Processing, Vol. 11, No. 1, pp. 20-31 (1970).

3) Dijkstra, E. W. : "The Structure of the "THE"-Multiprogramming System," Comm. ACM, Vol. 11, No. 5, pp. 341-346 (1968).

4) Lampson, B. W. : "A Scheduling Philosophy for Multiprocessing System," Comm. ACM, Vol. 11, No. 5, pp. 347-360 (1968).

5) Lampson, B. W. : "Dynamic Protection Structures," Proc. of AFIPS FJCC, pp. 27-38 (1969).

6) Dijkstra, E. W. : "Hierarchical Ordering of Sequential Processes," Acta Informatica, Vol. 1, No. 2, pp. 115-138 (1971).

7) Doi, H. : Solution to nano-pico school, Bit, Vol. 6, No. 4, pp. 115-138 (1971).

8) Wirth, N. : "The Programming Language Pascal," Acta Informatica, Vol. 1, pp. 35-63 (1971).