Information Processing in Japan Vol. 10, 1970

A Conception of Operating System Construction

HIDETOSI TAKAHASI* AND HISAO KAMEDA*

1. *Introduction*

Operating systems play an important role in any use of computers, say, batch processing, real time or time-shared use. In general, however, operating systems have so much sophisticated functions that constructing and maintaining them requires a considerable amount of labor. In addition, their external specifications also vary according to the actual environment, and this leads to additional complication. To cope with this complexity problem, a more systematic approach is needed.

The crucial problem here lies not in the design techniques for individual operating systems but rather in finding universal principles of operating systems. Such principles should not bring about undue complication or inefficiency of operating systems constructed on them. Any truly significant principle should not drastically deviate from the traditional ones and should receive empirical support.

The principle of the following generalized scheme of operating systems, which is believed to be a solution to the above mentioned problem is related to the existence of a general basic program upon which any kind of operating system can be built. Modern operating systems are usually constructed around a basic program called master control program (MCP), which handles all kind of interrupts and peripheral device controls, and thus enable "multiprogramming". We will develop a generalized but simplified construction of operating systems based thereupon. We constructed an experimental time-sharing system (upon HITAC 5020) for testing these principles.

2. *The analysis of existing concepts and principles*

Interrupt handling: The interrupt can be regarded as a kind of transfer of control between separate program modules. Another such example is the subroutine jump. In either of these transfers, the current contents of the program counter, general registers, which in all represent the machine conditions at the instant of the transfer and hence called "stateword", are saved immediately after the transfer occurs, and they are restored when the former program continues its execution as if no transfer had ever occurred in the midst of its com-

This paper first appeared in Japanese in Joho-Shori (the Journal of the Information Processing Society of Japan), Vol. 11, No. 1 (1970), pp. 20-31.

* Faculty of Science, the University of Tokyo.

putation.

In spite of this similarity between interrupts and subroutine jumps, the logical aspects of these two types of transfers are quite different. In subroutine jumps, it is customary that control is returned to the calling module just after the execution of the subroutine is over, and it is essential that the subroutine has gone through its computation before the calling routine resumes control, which results of necessity in a kind of nested arrangement. In the case of interrupts, on the other hand, the interrupted routine does not need to wait until the routine pertinent to the interrupt has done with its computation, but can resume control at any time. The often used nested arrangement to store the statewords of the interrupted routines into a "push-down-stack" is, therefore, irrelevant to interrupt handling, and it is more preferable to make the system handle the stateword of each module separately. Thus, each module will be treated as an independent entity by the MCP, and this independence between the interrupted module and the interrupting routine can be described clearly in terms of virtual machine, task [6], process [5], sequential process [1], etc. At any rate, a more careful analysis of interrupts is needed for discerning the principle of master control programs.

Logical connectivity in various transfers of control: Interrupts can be divided into the following categories; external interrupts (caused by the interval timer, peripheral devices, etc.), traps (caused by privileged instruction violation, memory protection violation, etc.), and supervisor calls. The transfers in external interrupts have no logical meaning, since they are not caused by the interrupted routines. In the case of traps, the activity of the interrupted programs should be suspended for the purpose of the inspection of errors, since the 'interrupted' program has brought the error. Thus, traps are more akin to subroutine jumps than to external interrupts. Supervisor calls and subroutine jumps differ from each other in that, in the former case, the routine issuing the request can often run in parallel with the requested routine, whereas, in the latter case, the routine issuing the request should wait until the pertinent service routine has carried out the requested task.

Thus, transfers among modules can be ordered in the following way, according to the degree of the logical connection (plus that of the necessity of synchronization) among modules.

1. subroutine jumps.
2. traps.
3. supervisor calls.
4. external interrupts.

As external interrupts, traps and supervisor calls are handled by similar hardware mechanisms, it seems natural to arrange things in such a way that the MCP handle all of these three kinds of interrupts and to free individul function modules from handling of these transfers. Thus, we can simplify the

whole structure of any operating system by committing interrupt logic to the care of the MCP and releasing other parts of the operating system from handling of transfers of control which have little logical significance.

Interprocess control communication for external interrupts and supervisor calls: While the transfer of control caused by an external interrupt implies nothing logical, it is possible to regard it as a request from an external device to the pertinent service routine. This request is interpreted by the MCP as a "wake-up" signal which makes pertinent routine (process) ready for execution [5]. Similarly, a number of supervisor calls may be interpreted as "wake-up" signals which make the requested service routine (process) or the pertinent I/O device ready for execution. When a service routine has done with the requested work and needs only to wait for other wake-up signals, the routine enters the "blocked" state by issuing a speial supervisor call named "block" to the MCP. Notice that "wake-up" and "block" are conceptually identical with POST and WAIT calls of IBM OS/360 respectively [6]. In this fashion, external interrupts and supervisor calls can be uniformly handled.

3. *Our proposed scheme*

Experiences with operating systems hitherto constructed seem to show that the general basic program (MCP) should fill the following newly noticed requirements;

I. to supply facilities enough for man-computer communication,

II. to permit easy modification and extension of system programs; i.e., to secure the independence among mutually independent parts and to prevent mutual interference among them,

III. to draw a gradual distinction in capability between system programs and user programs in place of a merely dichotomous distinction, i.e., to handle both in the unified fashion for the sake of simplicity and generality.

I. comes from the following; i) User programs often want to make use of external interrupt facilities provided by the hardware. ii) They will also want to put the trap operations by protection violation, etc. under their control and to construct an online debugging program (like DDT) protected from the interferences by the programs being debugged. Thus, each user program wants to be provided with its own monitor for handling these interrupts and traps. Hence the layered monitor operation is desirable.

These requirements will be fulfilled with the following principles or schemes.

System hierarchy: In our scheme both system programs and user programs consist of several processes (modules). Thus, the requirement III should be applied to the relation among processes. The notion of graded organization or layered monitor operation is expressed as follows. A process (say, Y) can supervise another process (say, Z) while the process Y is supervised still other process (say, X). In this

configration X has more capability than Y, and Y has more capability than Z.

Such an organization may be realized by the introduction of total ordering among processes. That is equivalent to the introduction of the concept of 'level' among programs. In this scheme, processes of the same level have equal power. Any process can control the processes of lower levels. No process can control the processes of higher levels. This concept of 'level' seems to underlie the concept of protection ring of Multics [2]. However, for the purpose of securing the independence among mutually independent processes, partial ordering of processes is more appropriate than total ordering. In addition, it should be so arranged that the system processes which are more frequently modified or improved than other processes in the system programs be given less capability so as to prevent errors in them from intermingling with the operation of processes which are not completely debugged either (requirement II). To fulfill all these requirements we will introduce the concept of tree-like system hierarchy, which is realized under the control of the MCP. Each process of the system hierarchy corresponds to one node of the tree (see Fig. 1). The capability (especially, the controllable memory space) of a process in the tree includes the capabilities of the inferior processes.

Trap handling: Traps cause transfers of control to the supervisor in usual systems. In our system, traps in a process lead to waking-up of the process immediately superior to it in the system hierarchy. Since a process and all the processes inferior to it form a group which is led by the process, traps in it should be regarded as errors in the whole group. So, for the purpose of inspection of the errors, the activity of the whole group should be frozen. As it should be possible to continue the activity of the whole group after the inspection is done, the state [waked-up/blocked] of each process should be reserved while the process is frozen. Thus, each process should have [frozen/unfrozen] state distinction in addition to [waked-up/blocked]. The state [frozen/unfrozen] should be stored recursively in a nested arrangement. Freezing and unfreez-

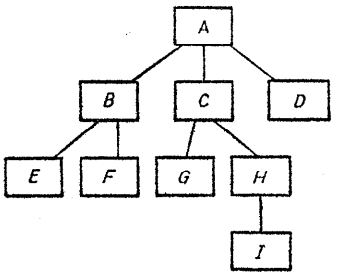

Fig. 1. An example of the tree-like system hierarchies. Each box represents a module (process). Alphabetical characters denote the names of modules.

ing may be added as new basic operations on processes in interprocess communication, since they will be useful also for other purpose, say, debugging.

In Fig. 1, processes C, G, H and I will be frozen when a trap occurs in C or when process A freezes C. If H and I are already frozen by C or by a trap in process H, only C and G will be unfrozen, and H and I will still be in the frozen state, at the time when A issues a request to unfreeze C, and so forth.

The 'suspend' and 'release' operations of Lampson [3] are similar to 'freeze' and 'unfreeze' respectively except that the former operate on a single process and are not related to the tree-like system hierarcy.

System hierarchies and memory protection: As described above, a process can control the memory space controllable to any processes inferior to it in the system hierarchy, but not vice versa. In Fig. 2, the controllable space of A

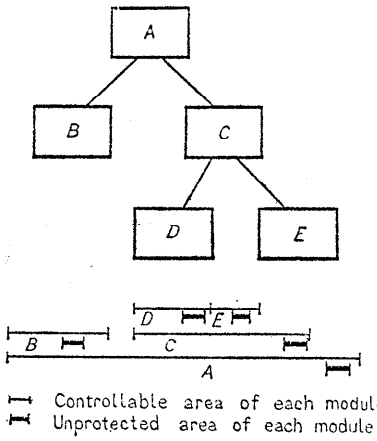

Fig. 2. The relation between the system hierarchy and memory protection.

includes the controllable space of B and that of C, and that of C includes that of D and that of E. However, it is desirable that A should be able to prevent itself from interfering with the space controllable to B or C accidentally and erroneously. Processes do not always need to keep the unprotected space as wide as the controllable space. Therefore, it is preferable to keep the unprotected space as small as possible and to change it dynamically within the bounds of the controllable space, in the course of processing. Thus, 'unprotected memory space' should be understood as a concept distinct from the concept of 'controllable space', which is included in the capability of each process.

4. *Explanations of actual constructions*

In this section, the actual constructions which incorporate the above ideas are explained, along with our implementation. The hardware of the next characters is supposed.

i. It distinguishes the two modes [master/slave].

ii. It allows us to specify the memory protection boundaries very finely and to change the whole aspect of protection very easily.

iii. It may not be equipped with segmentation or paging mechanisms.

The specification of the master control program: Each process can issue requests for the following basic operations on another process or itself, which are processed by the MCP.

- wake up (n): to wake up the process indicated by 'n'. If the process is already waked up, the wake-up-waiting switch of it is set. One wake-up-waiting switch is provided for each process.
- block: to halt itself and give control to another process. If the wake-up-waiting switch is on, it is reset and the process continues its activity without halting.
- change protect (a): to change the memory protection bounds to 'a' within its controllable space bounds.
- change stateword (n, a): to change the stateword of the immediate inferior process specified by 'n' to 'a'. The demanded capability should not exceed that of the process which has issued the request.
- freeze (n): to freeze the activity of the whole group whose leader process is 'n'.
- unfreeze (n): to unfreeze the activity of the whole group whose leader process is 'n'.
- create (n, m): to create a process immediately inferior to it. An identification number is given as 'n' to the new process by the MCP. The maximum number of processes which the new process can create further is demanded through 'm'.
- delete (n): to delete the whole group led by the process 'n'.
- change priority: to change the priority of itself.

For the sake of simplicity and preservation of independence among processes, we have the following restrictions. 'Wakeup' operates only on the immediate superior and inferior processes. 'Change stateword', 'freeze', 'unfreeze', and 'delete' operate only on the immediate inferior processes. This restriction prevents erroneous waking-up of external devices (say, in Fig. 1, E) by impertinent processes (say, G or H), etc., while retaining enough power for the system.

Interrupts are handled in the following way. External interrupts cause a transfer to the special service routine in the master mode which wakes up the pertinent process and sets the flag for each interrupt. The trap handling routine freezes all processes in the group whose leader is the trapped process and wakes up the immediate superior process.

The inner structure of the master control program: Our MCP consists of a procedure part (which is a pure procedure) and a data part. The data part

consists of several data blocks of uniform size, and a single block for system data. One data block is attached to each process, and it contains information which tells the control state [waked-up/blocked and frozen/unfrozen], the state-word, the linked lists which show the structure of the system hierarchy, etc.

The procedure part contains routines operating on these data blocks and I/O control programs. External interrupts, traps, and basic operation on processes issued by the user programs result in execution of one of these routines. The MCP always chooses the process of the highest priority among the waked-up and unfrozen processes for next execution.

Mailboxes in interprocess communication: One 'mailbox' flag should be provided for one-way communication between each pair of processes in order that the waked-up process can know which process has waked it up. A process which wakes up another can only set the flag, while the process which is waked up can only reset it, so that critical racing is prevented. For two-way communication, two mailboxes are needed for each pair.

5. *An example — a time-sharing operating system*

In this section, a TSS constructed upon the above MCP is described.

The structure and specification of a time-sharing operating system: The system hierarchy of our TSS is shown in Fig. 3. The whole system consists of a resident part common to all users and a transient part pertinent to each user which

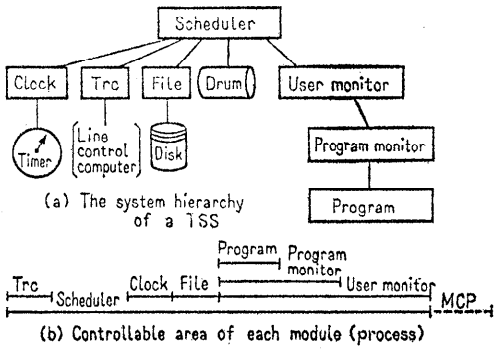

Fig. 3. The structure of an experimental time-scharing operating system.

is to be rolled in and out. The group of processes led by the user monitor process forms a functional unit which forms the transient part, and each user can have his own hierarchy of processes tailored to his specific needs. The user monitor process includes a command language interpreter. The functions of some other processes are as follows:

• clock.... is responsible for handling the interval timer and is consulted by the scheduler for time slicing. It is waked up by the hardware timer, and if

that means the end of a time quantum alloted to the running user, it wakes up the scheduler, etc.

• trc .... is responsible for communication with the line control computer. Being waked up by the line control computer, it sets a flag corresponding to the teletype device.

• scheduler .... handles mainly the rolling in and out of the user programs, and acts as an interface between the system processes and the user programs. It gives each user a virtual operating system and a pseudo-processor which consists of a variable number of processes and is furnished with a private console typewriter, clock, and file (see Fig. 4),

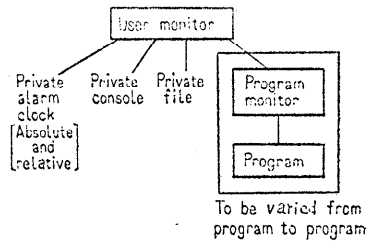

Fig. 4. The structure of a virtual operating system given to each user.

Since the space controllable to the user monitor process of each user includes that of all his processes, swapping is required only for the space of the user monitor process. To suppress the activities of the processes of the user being swapped, the freeze operation is found effective. When this process is waked up by the clock process and that means the end of the time quantum of the running user, it scans the flag for all terminal devices, lists the demanded users in the ready-user queue, selects the next user, freezes the running user, wakes up the drum for swapping, and unfreezes the next user.

When this process is waked up by the user monitor process, the request from the user is answered, i.e., the pertinent processes are waked up for these services by the scheduler.

The characteristics of our TSS: This section shows how the TSS makes use of the features of the MCP.

In our TSS, adding the process group of a user to the ready-user queue is something different from making a process ready in the MCP, but the scheduler takes care of the following correspondence between them.

The state of a user is	if his processes are
(B) blocked, (out of the ready-user queue)	all in the blocked or frozen state.
(R) ready, (in the ready-user queue)	at least one of them in the waked-up and unfrozen state.

Thus, at the stage of the scheduler, various events which lead to adding of a user to the queue, as well as various events which lead to the deletion of a user from the queue, are treated in a uniform fashion, and the distinction of such states as input wait, output wait, waiting command, working, dormant[4], etc. is irrelevant at this stage. Thus our scheme is more flexible, for example, in that it allows parallel I/O operations.

If a user intends to make use of the interruption feature (requirement I of Sec. 3) in his own program, he may arrange his program hierarchy as in Fig. 3. Normally, only the program process is in the waked-up state and the remaining two processes are in the blocked state. When the input key is pressed while the system is not in the input mode, the system wakes up the user monitor process, which inspects the code, finds out that it is not the quit code and wakes up the interface process, which controls the activity of the program process according to the code. This interface process is suitable also for running online debugging programs.

Requirement II is fulfilled in the following way. Since the clock, trc, and user monitor processes have no common controllable space (see Fig. 3), they cannot interfere with one another. The restrictions on basic operations on processes (Sec. 4) prevent many malfunctions in lower processes from spreading out. Since the scheduler process protects its own unused space, it can prevent itself from interfering with the inferior processes erroneously, etc.

The clock and trc processes are similar to user programs in that they cannot destroy the scheduler and are controlled by it, whereas user programs are not distinguished from system programs in that both have the same set of MCP calls and trap functions. Thus, the MCP has realized the graded organization of the operating system.

The system description language: In order to improve the flexibility of the system construction, the system programs, except the MCP, were written in FORTRAN IV, whenever possible. This resulted in a drastic reduction in the programming and debugging task with little overhead by practically eliminating all clerical errors. The MCP occupied about 1K words, and the resident TSS monitor occupied about 2K words.

6. *Conclusion*

Interrupts (external interrupts, supervisor calls, and traps) are hardware functions which are indispensable to any operating system, and an MCP that makes use of these hardware functions to bring universal and useful software entities or concepts may form the core of any kind of operating systems. The tree-like system hierarchy introduced makes the distinction between system programs and user programs a relative one. New basic operations on processes (freeze,

and unfreeze) are proposed and examined. A generalized MCP is sketched, and an experimental TSS is roughly described and scrutinized. The actual experiment presented here provides qualitative justification of our scheme, although quantitative evaluation of the system is still left open.

The authors express their gratitude to Professor Eiiti Wada, the TSS group of the Central Research Laboratory of Hitachi, Ltd., and the members of the Computer Center of The University of Tokyo for their cooperation and valuable discussions.

References

[1] Dijkstra, E. W., The Structure of "THE" Multiprogramming System. *CACM* 11, 5 (1968), 341-346.

[2] Graham, R. M., Protection in an Information Processing Utility. *CACM* 11, 5 (1968), 365-370.

[3] Lampson, B. W., A Scheduling Philosophy for Multiprocessing Systems. *CACM* 11, 5 (1968), 347-360.

[4] Saltzer, J. H., CTSS Technical Notes. *MAC-TR-16*, MIT (1965).

[5] Saltzer, J. H., Traffic Control in a Multiplexed Computer System. *MAC-TR-30*, MIT (1966).

[6] Witt, B. J., The functional structure of OS/360, Part II Job and task management. *IBM Syst. J.* 5, 1 (1966), 12-29.