Simulation of a Hybrid Machine—Another Pedagogical Aid for Programming Operating Systems in a High Level Language

TSUNETOSHI HAYASHI*

There are some inherent difficulties in programming operating systems (OSs) in a high level programming language. This is chiefly because an OS program requires close interaction with the hardware on which it runs, while a high level programming language tends to hide the machine structure. In this paper, a model and implementation of a hybrid machine which enables programs to cope with such difficulties is proposed. The model can handle both OSs written in a high level language and job programs in a binary machine code, and it can also provide a good environment for OS programming. An implementation has been used in an educational programming laboratory course in operating systems. Some drawbacks of the model and measures to address such drawbacks are discussed in detail.

1. Introduction

As programming methodology and software engineering were developing, it became apparent that systems programs such as compilers and operating systems (OS) as well as user applications should be coded in a high level programming language. A high level programming language can help to shorten the time between coding and certification of the program.

It seems, however, current implementations of high level languages and system supports such as programming environment, debug aid, and dump utility are not sufficient for this purpose, especially in OS programming.

On the other hand, writing a compiler in a high level programming language is much easier than writing an OS in such a language. For, the former can be done in a good programming environment such as TSS, unless it handles object binary machine code. In addition, the compiler writing methodology seems almost unified in comparison with that of OS writing. The difficulties of OS programming in a high level programming language are caused by lack of these properties.

A hybrid (virtual) machine model which will remedy these difficulties and provide a good programming environment for OS programming in a high level language will be proposed in this paper. The readers, however, should bear in mind that it is the concept being proposed which is important and not the particular details of how it is implemented.

In the following, considerations on the difficulties and OS programming environment are given as preliminaries. Outline of the hybrid machine will be described in section 2. Section 3 gives outline of an example OS. Section 4 deals with an implementation using simulation of the hybrid model, some drawbacks of the model, and their solution.

1.1 Difficulties of OS Programming in a High Level Language

An OS program must have close interaction with the machine on which it runs. But a program written in a high level language cannot have such interaction, since the language tends to hide the machine interface from a programmer unless some special facilities are provided with the language. The low level functions such as context switching, process scheduling, and interrupt handling are hidden from a user, and are dealt with by the run-time routines, or by the run-time environment OS, as parallel programs, monitors, exceptions, synchronization primitives, and pseudo processes. A high level language is based on much more abstract terms than the hardware for the sake of portability and reliability.

However the low level functions of the multiprogramming mechanism and interrupt handler are difficult to write in a high level language. This is the most desirable part to be written in such a language because its function is intricate and ill-defined, and its program is hard to certify.

Other parts of an OS outside of the multiprogramming mechanism layer may be written in a high level language rather easily. There are many examples: Multics, Solo [4], and Unix [5].

1.2 Good OS Programming Environment and Support

We should consider the criteria necessary for good programming environment and support for OS programming:

(1) An OS program should have to access several machine features such as interrupt handling, and context switching explicitly and directly (without intervention of a run-time routine or environment OS). This is required for implementing the multiprogramming mechanism, and evaluating traffic control or synchronizing primitive algorithms.

*Educational Center for Information Processing, Kobe University of Commerce, 4-3-3 Seiryodai, Tarumi, Kobe, Hyogo 655.

(2) Programs must be able to implement basic facilities constituting an OS: monitor functions, processes, and command interpreters.

(3) Subtle details of the hardware should not have to be exploited by OS programs: for example, an OS should be started without initial program loading or binary bootstrap operation.

(4) Diversification of OS concepts should not be restricted.

The hybrid machine model proposed fulfills these criteria well, and helps one to concentrate his efforts to develop algorithms constituting an OS. The model was first conceived for a laboratory course in systems programming, and an implement has been used for developing a small OS [1]. The model can be, however, of use for experimenting with OS algorithms at the kernel level and evaluating their performance and behavior, as well as for educational purposes.

In the following, we call the OS under scrutiny the proto type OS, and the environment in which it runs the environment OS.

2. Outline of Hybrid Machine Model

The hybrid machine model is uniquely and exclusively designed to meet above criteria: that algorithms consituting an OS as a whole should be written in a high level language, and that OS programs can have direct access to machine functions such as interrupts and input/output handling.

The basic principles of the model are:

(1) The machine has, at least, two program status: supervisor and user. The status is kept in a simulated program status word (PSW). The machine must be provided with the definition of the PSW.

(2) In the supervisor status, the machine inhibits interrupts and runs OS programs written in a high level language.

(3) In the user status, the machine permits interrupts and runs a program in binary machine code which is stored in (simulated) main memory.

(4) The machine is provided with several hardware status registers, PSWs, and main memory. These registers hold and indicate previous, current, and next status of the machine. They can be referred as data structures to by OS programs written in a high level language.

(5) The machine is provided with several instructions requisite for OS programming (probably the least set of these instructions may be *idle*, *start I/O*, *supervisor call*, and *test and set*). OS programs in a high level language invoke these instructions as either library procedures or primitive operations, depending on the implementation of the language used. The declarations for the registers and library may be included in the program prologue.

The language can be of any algorithmic type such as Algol 60, Pascal, and PL/I, or of typeless language such as BCPL and Bliss. The machine code can be of any real machine or of imaginary machine depending on the implementation of the model. We should make the point that the programs in different status run on completely different (virtual) machines, if OS programs in a high level language might ever be compiled into the same machine code. At first glance the model seems peculiar but it is natural and convenient for OS programming and development.

An implementation of the hybrid machine model is described in the following example. This one is very small in scale and is slightly modified from the model given in [1[for explanation. However the same principle can be applied for much larger models.

The configuration of the model is shown in Fig. 1. It has a standard input device, output device, and a console device such as a card reader, line printer, or typewriter. This is the minimum configuration of the model. If necessary, some direct access devices may be added to the configuration.

The machine also has an operator call interrupt button in its imaginary console for convenience of operation. As for the interrupt, much will be mentioned later.

Programs can refer to the status of the machine through the data declarations given in Fig. 2. It also shows library procedure declarations, and as mentioned above, they can be regarded as hardware registers, PSWs, and privileged instructions provided with the

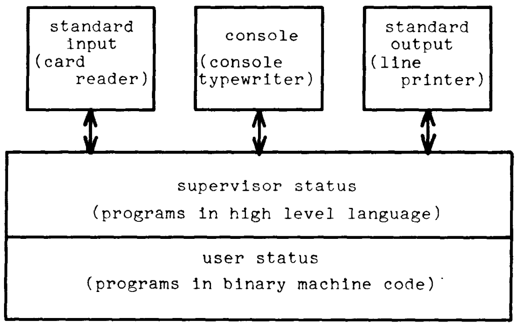
standard input (card reader)
console (console typewriter)
standard output (line printer)
supervisor status (programs in high level language)
user status (programs in binary machine code)

Fig. 1. Configuration of a model machine.

```
/* Hardware registers accessible in supervisor status */
const bits=··· ··· ···;
      max memory=··· ··· ···;
type PSW=record
           status: (supervisor, user);
           program counter: 0···max memory;
           program environment: ··· ··· ···;
           ··· ··· ···
      end;
      interrupts=(operator call, PCI, card end,
                  printer end, typewriter end);
      word=array [bits] of Boolean;
var old PSW, new PSW, current PSW: PSW;
      ISR: array [interrupts] of Boolean;
      main memory: array [0···max memory] of word;
procedure start io (device: ···, var data: ···)···;
procedure idle ··· ··· ···;
procedure supervisor call (··· ··· ···)··· ··· ···;
```

Fig. 2 Hardware registers in Pascal-like language.

machine.

In this model, interrupt events are indicated by the interrupt status register (ISR). To be more precise, some parts of the functions implemented simulate such interrupt events and set the register. The interrupt sources are program controlled interrupt (PCI), operator call interrupt, and device end interrupt signals. More interrupt sources such as program failure may be easily added if necessary.

The PCI is invoked either when a program in the supervisor status issues a supervisor call instruction, or when a program in the user status executes a program controlled interrupt machine code. The operator call interrupt is invoked when the operator call button is pushed by an imaginary operator, and the device end signals are invoked when the corresponding devices terminate their operation.

When such an interrupt event takes place, the model takes following actions.

—The machine changes to the supervisor status if it were in the user status;

—It saves the current PSW in the old PSW;

—It invokes an interrupt handling procedure which has been supplied as a part of the prototype OS;

—At the exit of this procedure, it restores the current PSW from the new PSW, and continues programs in the mode indicated in the (restored) PSW.

This interrupt handling procedure with a specific name is written as the nucleus of the prototype OS. The name of the procedure is, by convention, defined in the prologue declarations of the language, unless indirect procedure call mechanism, or procedure variable is provided with the language. The procedure can realize the multiprogramming mechanism by switching the PSWs. Note that an indicator is reset when a program assigns a new value to it.

For the vectored interrupt architecture, not just one interrupt handling procedure but as many procedures as interrupt sources may be supplied and invoked when a corresponding interrupt event occurs. In this case, registers such as ISR holding interrupt source information are not necessarily referred to by an interrupt handling procedure. The information needs be maintained only in the program which implements the machine model. This architecture seems more feasible because of its simplicity.

Library procedures for OS programming, or privileged instructions in the supervisor status, consist of the following set, but more may be added if necessary.

~*idle*—The *idle* procedure stops continuous execution of programs unless there is at least an interrupt event remaining. If an interrupt event is invoked while the execution is stopped then the procedure resumes normal execution of programs at the next statement to the call to the *idle* procedure.

~*start I/O*—The *start I/O* procedure starts an input/output operation of a device designated by the parameter. The machine continues execution of programs in parallel with such operations.

~*supervisor call*—The *supervisor call* procedure is for passing control to the OS nucleus from programs in the supervisor status. This is necessary for implementing the wait (block) function.

~*test and set*—The *test and set* procedure is necessary for implementing multiple processor systems.

The hybrid structure of the model is adopted to enable OS programs to be written in a high level language, and to avoid machine handling, such as booting an OS from a console, testing programs step by step, patching memories from the console, etc. by using, for example, interactive debugging facilities provided with the language. Such handling is very cumbersome and time-consuming, tends to fall into the program-debug cycle, and does not improve the procedures of writing correct programs from the beginning.

3. Notes on the OS Implementation

Naturally we must consider the question concerned with the model: what OSs can we write and how?

The following small scale example OS shows answers to this question. This example is, in principle, the same as the one which appeared in [1] for educational purpose. A large scale OS can be written within the same framework.

3.1 Example OS Specification

The example OS is very small and concise but retains the essential structure of existing OSs. The specifications are:

(1) It handles a single job stream externally, but has multiprogramming capability internally.

(2) It consists of the nucleus, several processes, and several monitor functions (monitor macro instructions).

(3) It has a simple job control facility (job control language) and a set of operator commands, but no spooling or resource allocation facility.

(4) Each process corresponds to an interrupt source and to job processing.

(5) Monitor functions which may be invoked by processes are based on the interaction among the processes.

(6) The nucleus establishes multiprogramming and transfers the control between monitor functions, as well as handles interrupt events.

Each process handles a corresponding interrupt source and is usually waiting for interrupt events to occur. A process is dedicated for job processing and deals with the job control facility.

The monitor functions support coordination and communication between processes. They can be of any type of communicating primitives such as the P/V operator, and are invoked by processes through the PCI indirectly.

3.2 Example OS Implementation

The example OS can be implemented on the machine model in the following way:

(1) The interrupt handling procedure (or procedure entries) functions as the nucleus. For each interrupt event, the handler awakes the corresponding process; for the PCI, it enables the called monitor function to run in the environment of the calling process. Then it schedules the next process to run, and dispatches it by restoring a PSW which has been saved on the entry to the handler.

(2) A process is provided for each task: input/output devices (for each device); operator interrupt; and job stream processing. Processes for I/O devices usually wait for an I/O request and device-end interrupt. It issues an input/output command against the former, and returns an end-of-operation message to the requesting process against the latter. A process for operator interrupt accepts and processes operator commands. A process for job processing is awakened by an operator command and interprets the job control language.

(3) Monitor functions provide a means to exchange messages and signal another process. They are invoked by processes through the PCI, and run in the environment of a calling process. This is made possible by maintaining the monitor function activation information with each calling process.

4. Implementation of the Hybrid Machine Model

The hybrid machine model can be implemented in many ways by interfacing (mapping) between the hardware, and the software. Probably a machine with these high level specifications may be built with little difficulty by using recent LSI technology. In the following, ways of implementation using only software techniques, and exploiting existing computer systems are considered.

Some of the feasible implementations may be
(1) by using the firmware;
(2) by using the virtual machine concept; and
(3) by using an existing programming environment through simulation.

In (1) above, resources provided with the hardware can be fully exploited, and programs run as fast as in a conventional machine. The run-time environment of the high level language, however, must be also supported by the firmware, and this fixes the language for OS programming, and causes inflexibility of the system.

In case (2), the run-time environment is supported by software emulation, and this method seems fairly flexible. The programs in the binary machine code run as fast as in a conventional machine. Programs in a high level language might run a little slower because of traps required for the emulation. In both cases (1) and (2), the link from prototype OS to run-time environment is through a specially defined trap instruction.

In case (3), the run-time environment can be the same as that of ordinary, non-OS programs. The link to run-time environment from prototype OS is through a conventional subroutine call mechanism. The binary machine code must be fully simulated by software for the sake of the linkage to prototype OS programs unless an adequate exception handling mechanism is provided and supported by the language.

In the following, this method of implementation is discussed.

4.1 Support System

The support system implements the hybrid machine model through simulation. Figure 3 shows the outline of the support system and its relationship with an OS. The principal control flow shown in the figure is as follows:

—When a real or simulated interrupt occurs, it invokes the OS nucleus (the interrupt handling procedure);
and at the end of the nucleus:
—If the current PSW is set to user status, it simulates binary machine code stored in the main memory according to the PSW;
and:
—If the current PSW is set to supervisor status, it dispatches the OS program according to the PSW.

The support system is written beforehand as a set of library programs, and later when a prototype OS is completed, the support system is linked together with the OS programs. When the load module is started by the environment OS, the support system takes control and runs the prototype OS on the (virtual) hybrid machine. No initial program load or machine handling is necessary. The test of the prototype OS can be done in an adequate programming environment, and if necessary, the support system can implement useful

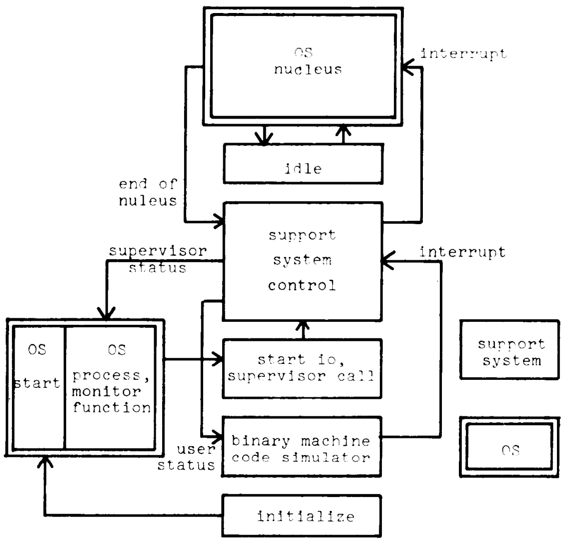

Fig. 3 Outline of support system.

facilities such as program trace, snap shot, and memory dump in interactive mode.

Some discussion on implementing interrupt events are in order.

(i) Operator call interrupt: an attention signal of the OS of the programming environment, or a message entered at the terminal can be used for simulating an operator call interrupt event. In the former case, the support system must declare a program entry to the environment OS for accepting attention signals. In the latter, the idle instruction and machine code simulator should be designed to accept a message input at any moment.

(ii) Device end interrupt: real data transmission timing or simulated timing may be used for this event. In the former, the support system must have a provision for acknowledging an end of transmission. In the latter, the system has to do the timing while simulating machine code and idling, or otherwise a timer provided for the environment OS can be used.

(iii) Program controlled interrupt: implementing this is straight forward and causes almost no problem. Note that this interrupt is the main entry to the support system from the prototype OS.

4.2 Considerations on Some Difficulties

There are a few difficulties or shortcomings in this model. Some of them are (i) a language issue; and (ii) parameter passing to monitor functions from job and prototype OS programs.

(i) Language issue

The multiprogramming of a prototype OS might violate the block structure control of that OS programs, or at least, procedure call nest chains. Multiprogramming is established through indirectly called coroutines, if the prototype OS and the support system are to be regarded as an integrated program. The nucleus determines a routine to be dispatched.

If a process calls a monitor function by invoking supervisor call primitives, control will be passed to the function, and will be returned again to the next point of the invocation after process switching has occurred several times. The activation information at the invocation might be lost if programs in different block levels were dispatched while another process is running. The PSW must keep this information in order to cope with this problem.

In a contemporary programming language, the activation information is kept in the run-time stack. The language must be designed to have some provision to save this information conveniently in a PSW, or a whole stack must be kept in a PSW, and this should be avoided. The run-time stack should be implemented using a linked chain or a pointer so that a PSW needs to keep just a chain or a pointer. The run-time stack can grow like a tree with each branch corresponding to a process.

Another solution: the processes and monitor functions of a prototype OS are all written in the same block, and supervisor call primitive passes the label attached to the next statement to the invocation as a parameter. A PSW holds the label as a current value of the program counter. Shortcomings are that the OS programs cannot be adquately structured, no procedure is available in a process or in a monitor function, and many labels and "gotos" are distributed throughout the program.

(ii) Parameter passing

Parameters cannot be passed to a monitor function from binary job programs or from processes in the usual way as the link between a function and a program is indirect, and is quite different from a conventional procedure call. Parameters must be reduced to pointers referring to them, and held in activation information with the PSW of a calling program.

In addition, there is the difficulty that user job programs and OS programs run on quite different virtual machines (in a sense that a programming language defines a virtual machine). Therefore parameters given to a monitor function from user job programs and OS processes are also quite different: binary coded data and abstract type data.

A monitor function must determine the type of parameters by examining the program status of calling programs. This overhead may be a little annoying, but the method is workable.

There can be another method, in which the support system automatically converts the parameters from source (calling) program type to destination (called) program type. Shortcomings are that the data type of parameters must be defined by the support system and not by the OS writer, and that the life-time of parameters must be well-known to the support system. In the multiprogramming environment, this may be difficult.

5. Conclusion

So far a hybrid machine model for OS programming in a high level programming language is described. It illustrates that OS programs including the nucleus can be written in such a language in an adequate programming environment. This model seems to be a good compromise between the current hardware implementation and the programming system. It will be of much help in exploiting the OS development methodology.

In a sense, this model lies just in the middle of Solo/concurrent-Pascal and Unix/C approaches. In concurrent-Pascal, the language implies the multiprogramming mechanism, and an OS writer is unable to deal with it directly. In this particular case, user jobs are restricted to just one language (sequential-Pascal).

In the Unix/C approach, there is no such restriction, but since the virtual machine implied by C is as low as a real macine, the level of programming, especially in data abstracting, also seems as low as that of an assembler. Programs might also suffer from peculiarities of the hardware such as interrupt architecture at the cost of programming flexibility.

The hybrid machine model proposed in this paper removes these restrictions. A language for the model will be considered for development. Perhaps it will be of a typeless language with chain-linked run-time stack. This language can resolve parameter type matching, since user job programs and prototype OS programs run on similar, but not quite the same virtual machines.

References

1. Shimasaki, M., Hayashi, T., Kitazawa, S., Furutani, S., Watanabe, M. and Watanabe, K. A Laboratory Course on Programming in Department of Information Science, *Trans. of IPSJ*, 21, 2 (March 1980) 83–90, (in Japanese).
2. Hayashi, T. An Educational Project in Operating Systems Programming Laboratory Course, *Journ. of Kobe Univ. of Commerce*, 32, 1, (August 1980), 40–55.
3. Donovan, J. J. and Madnick, S. E. Software Projects, McGraw-Hill (1977).
4. Hansen, P. Brinch The Architecture of Concurrent Programs, Prentice-Hall (1977).
5. Ritchie, D. M. and Thompson, K. The Unix Time-Sharing System, *Comm. of the ACM*, **17**, 7 (July 1974), 365–375.

(Received January 12, 1981)