The structure of the MU5 operating system

D. Morris, G. D. Detlefsen*, G. R. Frank, and T. J. Sweeney

Department of Computer Science, University of Manchester, Manchester M13 9PL

This paper describes the structure of the Operating System for a multicomputer complex which is being constructed at the University of Manchester. At present the complex consists of a modified ICL 1905E and the MU5 machine. The Operating System has a highly modular structure consisting of a small kernel coded for the real machine and a number of separate programs each of which has its own virtual machine.

(Received October 1971)

The MU5 system currently being constructed at the University of Manchester enables a number of different computers to communicate via a common interface unit. At present the complex consists of a new machine, MU5, designed within the University (Kilburn, Morris, Rohl, and Sumner, 1968) and a modified ICL 1905E. This paper describes the structure of the Operating System for the complex. An earlier description (Morris and Detlefsen, 1969) related to the prototype system developed on the 1900 machine.

Any operating system which protects itself and other user programs from the currently executing program does so by restricting each program's access to the actual hardware. In this sense such systems might be said to provide a *virtual machine* for each user progıam. In many systems (e.g. the ICL 1900 George Operating System) this virtual machine is only trivially different from the real machine. Other systems aim to pıovide a virtual machine which offers significant software advantages over the real machine.

In early systems like Atlas (Kilburn, Howarth, Payne, and Sumner, 1961) each virtual machine was independent and unaware of the existence of others. That is, all programs running in user mode were independent. The entire operating system was a privileged program coded for the real machine. If any part of this system contained an error it could cause the whole system to 'crash'.

On MU5, and other systems which have similar properties (Kerr, Bernstein, Detlefsen, and Johnston, 1969, and Vyssotsky, Corbato, and Graham, 1965), only a small kernel of the operating system, which creates the virtual machine feature, is coded for the real machine. The rest of the system, in the MU5 case, is made up of separate programs (called *supervisors*) each of which has its own virtual machine. In general there is a supervisor controlling each autonomous system activity and a supervisor to assist each class of user activity. **Fig. 1** illustrates a possible configuration. The design is open ended and new supervisors can be added at any time. Since they are subject to the protection system applied to user jobs they cannot interfere with the rest of the system. Each activity running in its own virtual machine is termed a *process*.

Unlike user jobs in an Atlas type of system the supervisors of MU5 are not independent. They require facilities to communicate with each other. Also they require to create and control other virtual machines containing user jobs. Thus the system kernel (the Supervisor Supervisor) must create virtual machines which comprise:

1. A virtual store.
2. A virtual input/output system.
3. An instruction set which satisfies the requirements of supervisors as well as user jobs.

The virtual store

This section is restricted to a factual summary of the MU5 segmented storage organisation. It is not feasible to relate it to the many other descriptions of storage organisations which have appeared in the literature. The system derives mainly from the informal conventions for exploiting a large virtual memory which evolved with the Atlas system. We believe that the notion (described below) of common segments and the mechanism for passing segments from one process to another are novel.

The size of an address in MU5 depends on the size of the object it addresses. An 8-bit byte address contains 32 bits. This address is regarded by the store management part of the Supervisor Supervisor as the concatenation of a segment number and a position within a segment as follows

SEGMENT NUMBER	POSITION IN SEGMENT
<——14——>	<——18——>

Thus the virtual store of each process is segmented and contains 16K segments each of 256K bytes. A similar virtual store is provided in the 1900 machine in the complex except it has only 64 segments of 64K 24-bit words.

In general, segments will be assigned to the logically distinct parts of a program as, for example in **Fig. 2.** If a logical area is bigger than a segment it may occupy several consecutive segments. Address modification across segment boundaries is permitted.

The main features of this segmented store are the following:

1. Associated with each segment are some access control bits which determine the sort of access that can be made. The possible access states are: obey only, obey and read, read

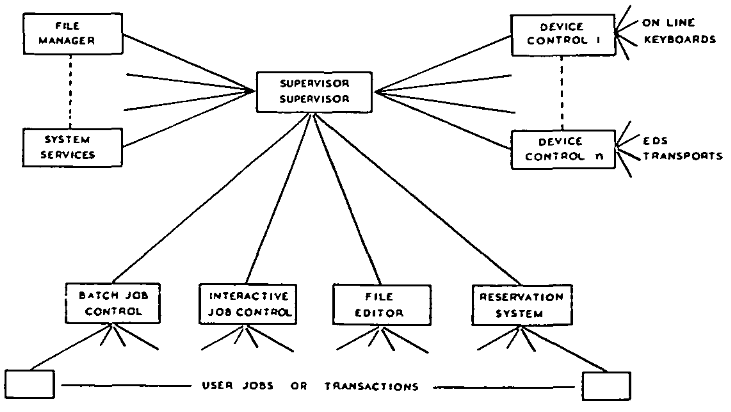


Fig. 1

*Now at the General Electric Research and Development Center, Schenectady, New York, USA.

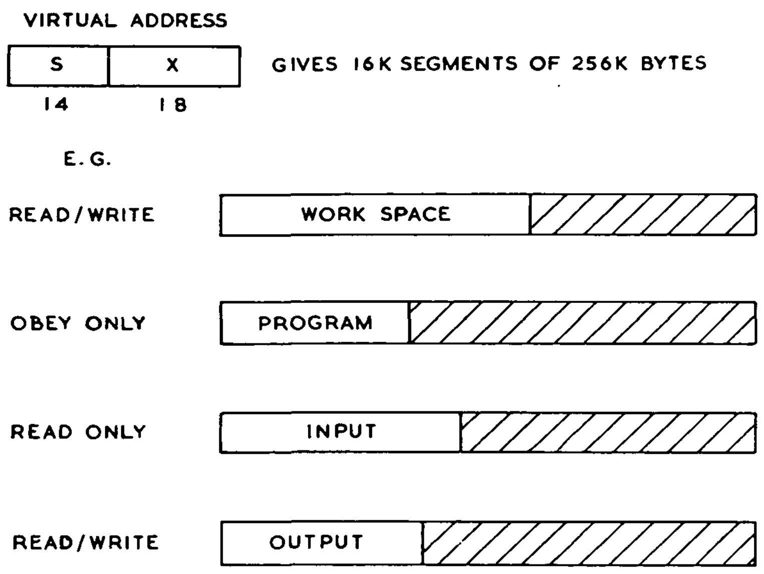


Fig. 2

only and read and write. There is a further bit which determines whether the user may alter the access state.

2. Segments can be shared. That is, the same segment can exist in several virtual stores. For example, it may be a compiled program which is being used simultaneously by several users. Alternatively, it could be a piece of data in the case where several programs are collaborating on the same task. This sharing may arise either as a result of several processes opening the same file, or as a result of a process's sending one of its segments as a message to another process (see next Section). The originator of the file or segment will control the access state allotted to the others.
3. The segments forming the upper half of each virtual store are *common* to all virtual machines. They are write protected from the user and contain the Supervisor Supervisor, its working space, compilers and library procedures.
4. Bulk I/O is achieved by transferring segments in and out of the virtual stores concerned.
5. Segments of code, I/O or data can be filed. The action of the corresponding open file command links a file segment into a virtual store. Very large files have to be subdivided into several segments.

The segmented store is implemented by paging. Each segment may have its own page size. A fuller description of the store management scheme will be given elsewhere.

Input/Output

Input output propagates through the machine complex via a message switching system. This allows any process to send a message to any other process. A message consists of a short header (about 100 characters) and optionally a segment of virtual store. There are some terminal processes in the system, called device controllers, which have the privilege of communicating with the real input/output devices. A user can send input to any process in the system via the device controller attached to his input station. These device controllers recognise a common command language which enables a user to name the process to which the input is to be sent, and state whether it is to be buffered into a segment or sent line by line, and so on. Similarly, output is achieved by the processes sending messages to the output device controllers. The controllers attached to interactive terminals deal with both input and output and also provide interlocks.

Fig. 3 illustrates the message switching system. Each process (or virtual machine) has a unique name and messages can be sent to it by any other process which knows this name. Into each virtual machine are 16 channels which can be addressed by the transmitting process. The process running in each virtual machine can exert some control over the 16 input channels.

There are three principal states for each channel. First it can be *closed* which means all messages directed to it will be returned to the sender. Second it can be *open*, which means that messages from any source are permitted. The third state is *dedicated* in which case it is open to one specified process and closed to the rest.

Associated with the message channels is a software interrupt system. The arrival of a message on any channel for which the interrupt inhibit has not been set will cause an interrupt. This means a procedure call is forced to an address preset by the user. The contents of all registers will be preserved so that the program can be resumed when the interrupt has been processed. Alternatively, interrupts can be inhibited and the process can poll its input channels at will. In both cases the messages are queued outside the virtual machine and a specific command must be issued to read each one into the virtual store. A process which is unable to continue until its input arrives may suspend itself awaiting the arrival of a message. To each message the system appends the name of the sender; therefore anonymous messages cannot be sent.

A normal user does not need to use the message system directly. There is a library of input/output and other housekeeping routines available in the common part of the virtual store.

As well as forming the basis of the input/output system the message system satisfies two other needs. It allows programs to communicate in order to synchronise or to assist each other. Also it is a general mechanism by means of which a segment may be transferred from one virtual store to another. This transfer does not involve copying the contents of the segment; only a pointer to the page table for the segment is actually transferred.

The message systems of all the computers in the MU5 complex will be linked. Thus the virtual machine network depicted by Fig. 3 is distributed across these machines. The device controllers will run in the machine to which the devices are connected. An optimum distribution of the remaining modules will be determined empirically.

The instruction set of the virtual machine

Each compiler makes available to the user its own instruction set. The basic instruction set of the virtual machine consists of the instructions which compilers may generate. In MU5 this Compiler Target Language (CTL) (Capon, Morris, Rohl, and Wilson, 1971) has been defined at as high a level as possible. It contains a formal procedure calling mechanism; thus the effective instruction set can be arbitrarily extended by adding CTL library procedures. This CTL library includes the organisational commands implemented by the Supervisor Supervisor for the following tasks:

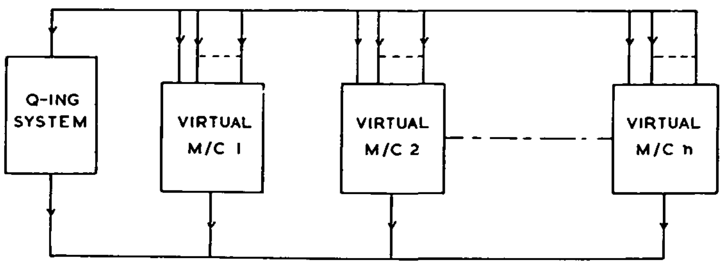


Fig. 3

1. Creating and controlling processes.
2. Creating and deleting segments.
3. Opening and closing files.
4. Sending and receiving messages.

Entry to these procedures sets a digit in the hardware which gives them special privilege. There is an extra system protection bit included with the access control bits described earlier. Access to segments which have this bit set is only possible from privileged procedures. This means that system information such as file directory can be held in the user's virtual store. Also system information relevant to more than one process such as the message queues can be held in the common virtual store. To a large extent, therefore, the 'supervisor calls' can run inside a user's virtual machine in order to perform tasks on his behalf. The commands associated with the more involved operations such as file management use the message system to call upon the services of slave processes whose responsibility it is to co-ordinate these tasks.

Some implementation points

1. *The locked-in part of the Supervisor Supervisor*
Only a small part of the Supervisor Supervisor is forced to run in locked down store in the real machine. This is concerned mainly with store control and CPU scheduling. A provisional description of the former has been given in Morris and Detlefsen (1969).

The CPU scheduling algorithm cannot have a knowledge of the relative priorities of the supervisors built into it because new supervisors can be added dynamically. Similarly, although a supervisor may know the relative priorities of jobs under its control it will not know the priorities of those under the control of other supervisors. This problem is resolved by relating priority to cost.

There are 16 priorities from which a supervisor may choose when it requests that a user virtual machine be activated. At the bottom level the rate of charge for CPU time is almost negligible. In the present system the charge thereafter increases linearly with priority. This means that long compute jobs are prohibitively expensive to run except at the bottom few priority levels so that they usually wait until the system is otherwise idle. Short development jobs are usually assigned the higher priorities.

The scheduling algorithm selects processes for running primarily according to priority. It perturbates this order for two reasons only. First, in order to maintain good system efficiency jobs are classified (e.g. long and short) and a suitable mix of jobs is multiprogrammed. Thus a low priority job of one type may be run together with a higher priority job of a different type. Second, some restriction is applied to the amount of CPU time which can be utilised in one continuous burst at the higher priorities. When a job reaches this limit it is halted and moved to the end of the queue of waiting jobs at its priority level. This time limit is included as a guard against the possibility of an interactive process's monopolising the CPU during a long operation. It is expected that interactive jobs will normally become halted awaiting I/O (i.e. messages) before the time limit is reached. When they are freed, as a result of receiving a message, they are placed at the end of the list of processes waiting for CPU time at the relevant priority level.

2. *Levels of protection*
Intentionally the system provides only one protected level inside each virtual machine. This is always occupied by the Supervisor Supervisor. Subsystems which require protection are intended to be run in separate virtual machines and be activated by use of the message system. In contrast a number of other systems provide multiple levels of protection in which subsystems may occupy intermediate levels (Graham, 1968).

3. *System deadlocks*
A system of this kind deadlocks when every process is awaiting either the action of some other process or a resource currently in use by some other process. In a virtual machine system, if an infinity of virtual resources can be created the problem does not arise. We are concerned only with total system deadlocks and exclude partial deadlocks due to logical errors in collaborating groups of processes. Unfortunately the Supervisor Supervisor has to map the virtual resources into the real hardware, and this is where deadlocks can arise. Foı example, any list maintained in locked down store will be limited in size.

The theoretical solutions to this problem have been found too restrictive. A common restriction is that the processes are independent (see for example Haberman, 1969). In any case a near-deadlock situation may be almost as undesirable as deadlock because of the degeneration of system performance which develops. The system under discussion relies on empirical tuning, i.e. adjustment of parameters, in order to avoid deadlock situations. For example, if the amount of unused real store falls below a certain level the device controllers will refuse to accept further input.

The user's interface with the system

Supervisors provide the means whereby users of the system initiate and control jobs. The main task of such a supervisor is to create and activate a new virtual machine for each user job. Once started, the facilities which are provided by the library of housekeeping procedures, within the virtual machine, usually enable the job to run to completion without further assistance from its supervisor. However, some supervisors provide special facilities which a process may access by sending its supervisor a message.

There are several different types of supervisor in the system corresponding to different types of jobs. The simplest handles jobs with a single input stream and no requirements for special facilities. Another deals with the larger batch-processing type of job, providing facilities for multiple input documents, special scheduling requirements and jobs which depend on the prior execution of other jobs. A third supervisor handles the interactive type of job; in this case the user is given a greater degree of control over the execution of the job. In addition there are more specialised supervisors including an information retrieval system, an online desk calculator system and a prototype of an airline reservation system.

Conclusions

A prototype version of the Supervisor Supervisor has been implemented for the 1905E and has now been in use for about 18 months. The implementation of this system took approximately 6 months (and three people). In the time since its introduction the number of supervisors and other facilities has increased steadily and it has been used to develop much of the software for MU5. It is too early at this stage to draw conclusions about the performance of a system structured in this way, but experience with the prototype system indicates that there is no serious loss of efficiency.

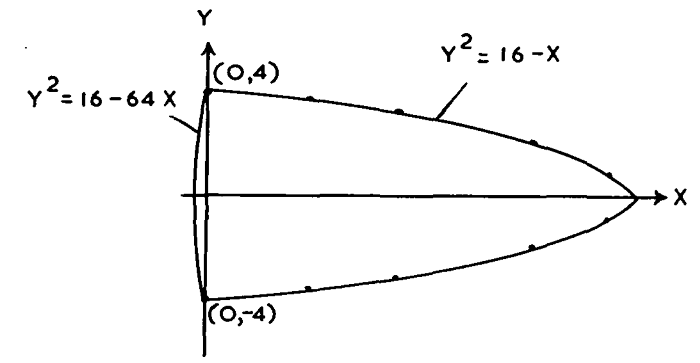
During the next 12 months a new edition of the 1905 system, which will interface with MU5, is being developed. The Supervisor Supervisor for MU5 itself is also being developed in this period.

References

Capon, P. C., Morris, D., Rohl, J. S., and Wilson, I. R. (1971). The MU5 Compiler Target Language and Autocode, *The Computer Journal*, Vol. 15, No. 2, pp. 109-112.

Graham, R. M. (1968). Protection in an Information Processing Utility, *Communications of the ACM*, Vol. 11, pp. 365-369.
Haberman, A. N. (1969). Prevention of System Deadlocks, *Communications of the ACM*, Vol. 12, pp. 373-385.
Kerr, R. H., Bernstein, A. J., Detlefsen, G. D., and Johnston, J. B. (1969). Overview of the R & DC Operating System, Report No. 69-C-355, General Electric Research and Development Center, Schenectady, New York.
Kilburn, T., Howarth, D. J., Payne, R. B., and Sumner, F. H. (1961). The Atlas Supervisor, Proceedings of the Eastern Joint Computer Conference, Washington, D.C, pp. 279-294.
Kilburn, T., Morris, D., Rohl, J. S., and Sumner, F. H. (1968). A System Design Proposal, Proceedings of the IFIP Congress, 1968, North-Holland Publishing Company.
Morris, D., and Detlefsen, G. D. (1969). An Implementation of a Segmented Virtual Store, IEE Conference on Computer Science and Technology, Manchester.
Morris, D., and Detlefsen, G. D. (1970). A Virtual Processor for Real Time Operation, *Software Engineering*, Vol. 1, pp. 17-28, Academic Press.
Vyssotsky, V. A., Corbato, F. J., and Graham, R. M. (1965). Structure of the MULTICS Supervisor, Proceedings of the AFIPS Fall Joint Computer Conference, pp. 203-212, Spartan Books.

Continued from page 112



61 DISTINCT, STABLE, LOCALLY MAXIMUM 65-GONS EXIST, WITH FROM 5 TO 65 VERTICES ON Y = 16 − X.

Fig. 3

$y^2 = 16 - x$, the larger the area of the 65-gon. Each of the 61 locally maximum 65-gons are stable, in the sense that if small enough perturbations are made of each of their points (even simultaneously), then the iterative process of adjusting the middle points of various triplets of vertices must converge back to the initial configuration. The sharp angles at the intersections $(0, \pm 4)$ prove to be impassable barriers to the migrations of vertices of N-gons for $N \leqslant 65$ (see Fig. 3).

Any point of a circle may be the vertex of a regular inscribed polygon. The circle may be projected onto any ellipse, so that the regular inscribed polygon is projected onto a locally maximum polygon of the ellipse (note that the projection preserves tangency and parallelism). It is probably characteristic of the ellipse (and circle) that any of its points may be used as a vertex of a locally maximum polygon of any order.

Yours faithfully,
K. A. Brons

1928 Cardinal Lake Drive
Cherry Hill
New Jersey 08034
USA
6 December 1971

To the Editor
The Computer Journal

Sir

Calculation of a double-length square root from double-length number using single precision techniques

I write to comment on the letter by D. W. Honey (this *Journal*, Vol. 14, Nov. 1971, p. 443) where he describes a method which he attributes to his colleague, Mr. J. Grabau. The method given is, however, quite well known, being Newton's method with rearrangement of terms to exhibit the correction to be made at any stage. The usual form of Newton's method for finding $\sqrt{a}$ is

$$x_{i+1} = \tfrac{1}{2}\left(x_i + \frac{a}{x_i}\right)$$

which can be rewritten as

$$x_{i+1} = x_i + \frac{a - x_i^2}{2x_i}$$

to show the correction. Mr. Honey's (or Mr. Grabau's) technique is therefore seen to be equivalent to one more step of the Newton process after the single-length result has been obtained.

However, *it is necessary to take care when this method is being used* in fixed-point arithmetic, as overflow could result if the 'wrong' single-length square root is taken. It is not enough to take the unrounded (rounded down) value, because this leads to a value x satisfying

$$0 \leqslant a - x^2 \leqslant 2x\,,$$

and this can obviously give overflow. No such difficulty can arise if we take the rounded value, because this satisfies

$$-x \leqslant a - x^2 \leqslant x\,,$$

with a correction of at most $\frac{1}{2}$ unit, although it may be of either sign. In Mr. Honey's example, therefore, he *should* have used 14 as his initial guess at $\sqrt{192}$, which would have led to 13·86 as the better approximation, instead of 13·88. Since $(13{\cdot}86)^2 = 192{\cdot}0996$ this gives an error which is about $\frac{1}{7}$ of that quoted. Indeed, it is not difficult to show that the maximum relative error using the rounded single-length approximation will be about $\frac{1}{4}$ of the error that could arise from using the unrounded version. Choosing the rounded approximation is thus noticeably more accurate for the same amount of work.

Yours faithfully,
P. A. Samet

Computer Centre
University College London
19 Gordon Street
London WC1
14 December 1971

Mr. Honey replies:

I am obliged to Professor P. A. Samet for his letter commenting on 'Grabau's Method' for obtaining a double precision result using single precision techniques.

I think that I may have misled the readers by the lack of emphasis on the single precision. Professor Samet is quite correct in his observation that Newton's method is involved, although I had not appreciated this fact at the time. My main concern was that single precision techniques are used throughout the process and is something often overlooked by software designers with non-mathematical background.

I am also obliged for Professor Samet's further comment re 'overflow' (again sometimes overlooked), and his development of my worked example in decimal in which a rounded value is taken in preference to an unrounded value—a technique I shall remember in future.

I am sure that several readers will have gained benefit from our minor correspondence—which is its basic purpose. Thank you, Professor Samet, once again.