University of Edinburgh

Department of Computer Science

An Experiment in Doing it Again, *But Very Well This Time*

by

N. H. Shelness, D. J. Rees,
P.D. Stephens and J.K. Yarwood

INTERNAL REPORT CSR-18-77

James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh
EH9 3JZ

December, 1977

AN EXPERIMENT IN DOING IT AGAIN, BUT VERY WELL THIS TIME

N.H. Shelness and D.J. Rees

Department of Computer Science, Edinburgh University

P.D. Stephens and J.K. Yarwood

Edinburgh Regional Computing Centre

abstract: This paper describes a rationale for re-implementing an operating system, a particular operating systems re-implementation project, the tools being used by the project and the structural changes being made to EMAS, the particular operating system.

AN EXPERIMENT IN DOING IT AGAIN, BUT VERY WELL THIS TIME

> "Our problem is that we never do the same thing again. We get a lot of experience on our first simple system, and when it comes to doing the same thing again with better designed hardware, with all the tools we know we need, we try and produce something which is ten times more complicated and fall into exactly the same trap. We do not stabilize on something nice and simple and say "let's do it again, but do it very well this time."
>
> David Howarth [H072]

INTRODUCTION

In the summer of 1976 a small group of teaching staff in the Edinburgh department of computer science decided, given the effect that they felt the ICL 2900 computer range was likely to have on the British computing scene, to examine and evaluate the system. The basis for this excercise, in addition to written information at various levels of detail, was an ICL 2970 computer that the department agreed to house in its research laboratories on behalf of the university's computing centre in exchange for one hours hands on access a day during the prime shift. There were three aspects of the 2900 range that this group were particularly interested in studying.

1. The appropriateness of the order code as a target language for the compilation of one or more high level languages.

2. The appropriateness of the architecture as a vehicle for a modern virtual memory operating system.

3. The appropriateness of the both the order code and the architecture as a basis for a range of hardware implementations.

The group split informally into three overlapping sub-groups each of which dedicated itself to studying one of the three aspects. It was out of the second of these sub-groups that the project described in this paper grew.

The members of this second sub-group had a fairly clear idea of the sort of functions that a modern virtual memory operating system should provide, as well as the level of performance it should achieve based on almost ten years experience with these sorts of systems. This experience had been gained primarily in the design, development, use and modification of the Edinburgh Multi-Access System, but was tempered also by the experience of others aquired both from the open literature and more directly in face to face discussion. Even with this experience, or perhaps because of it, the sub-group felt that the evaluation should consist of a study of the effect that the architecture had on a real operating system and not a study of the architecture in isolation. Three operating systems were identified as candidates for this evaluation.

1. The manufacturers operating system (VME/B).

2. The Manchester University Software System (MUSS 2900).

3. A re-implementation of EMAS for the ICL 2900.

The first candidate was rejected almost immediately, largly because it was not self supporting, and also because its size and its performance made it difficult to detect any beneficial effects that the architecture might have had upon it.

The second candidate appeared initially most attractive. Versions of MUSS existed while a re-implementation of EMAS was merely an idea. It appeared to be well structured and efficient, and the use of MUSS seemed also to form a basis for collaborative research involving both the Edinburgh and the Manchester departments of computer science, but after much effort both in Edinburgh and Manchester it too was rejected as a candidate. There were two reasons for its rejection. The first was timing. At the time the Edinburgh sub-group needed it, the 2900 version of MUSS was still being implemented. The second was that the Edinburgh group felt uncomfortable with the software technology used in constructing MUSS, and with the impact this had on our ability to either modify or extend it.

Thus with some sadness at the failure of the MUSS approach, the task of re-implementing EMAS for the ICL 2900 was started. At this point the university computing centre expressed an interest in seeing EMAS re-implemented for the ICL 2900 as an alternative to the manufacturer's operating system. Without making any commitment to use a finished product, they were willing to commit staff to the project and a full scale re-implementation excercise was launched which had as its goal the development of a full production system and not merely an architectural evaluation.

RE-IMPLEMENTATION

By re-implementation, we mean re-programming and altering the structure of an existing system where necessary, so as to better implement the function of that system. We distinguish re-implementation from two other activities. These are the transportation of an existing system to new hardware, and the design of a new system. If we transport a system, we leave its function and its structure unaltered. Re-implementation is not merely a combination of transportation and re-design, but is a unique activity in its own right. It is doing it again, but very well this time.

We felt that it was necessary to alter the structure of the existing 4/75 EMAS system for three reasons:-

1. The existing implementation of EMAS on the ICL 4/75 is essentially a prototype. This is despite the fact that the system had existed as a product for over seven years. As with almost all prototypes, it had many rough edges, and parts of questionable quality. It should be possible, and in other branches of engineering it certainly is possible, to reconstruct these parts without altering the function of the system as a whole [WI76].

2. The function of the system had evolved over the last six years in ways that were not conceived of in the original design. This is not a unique phenomenon. As has been documented by Belady and Lehman [BE76], the function of many large programs evolves over time. This results eventually in a disastrous clash between the program's function and its structure. Belady and Lehman have indicated that they believe that this phenomenon is endemic to all large programs. Two examples of such change of function in EMAS are:-

a. The transition of the system from being the centre of a computing utility to a node on a network.

b. The transition of the system from possessing its own private file system, to sharing file systems with other EMAS systems [SH75].

3. Improvements in technology allow for simplifications in system structure. To take one example: in the existing implementation of EMAS for the ICL 4/75, drums are allocated on a page by page basis, and operate both as a read and a write cache for pages held on disk. There were two reasons for doing this. The first was that drums on the System 4/75 were relatively small (2 megabytes). The second was that the disks were very slow when compared with the drums (a factor of 20). Neither of these factors exist for ICL 2900 computing systems. The drums are larger (6 megabytes) and the disks are faster (only a factor of 4 times slower than the drums). Given these improvements in technology, we can allocate the drums in bigger chunks and dispense with the use of the drums as a write cache for the disks. We can use the same unit of allocation on the drums that we use on the disks, which though less efficient in the use of drum space considerably simplifies drum management. We can also eliminate the fiendish problem of inconsistency between the contents of the disks and the contents of the drums. This in turn eliminates the complex code needed to move pages asynchronously from the drums to the disks, at what we believe is the now acceptable cost of having to update the copy of a page held on disk each time we update the copy held on the drum.

THE PROJECT

The EMAS re-implementation project has been informal in organization, small in size, and relatively short in duration. It commenced in October, 1976 with the intent of demonstrating a system by June 1977, and generating a first release by April 1978.

The project has had ten members. A six man core consisting of four full time workers and two part time workers has implemented most of the system. This group have been augmented by a second group of about four, who were available to make specific contributions in an area of expertise (for example device handlers). The contribution of this second group has been very limited in time, to the extent that there have never been more than one or two of them active at the same time.

Why has it been possible to handle a project of this scale with such a small team and in such a relatively short period of time? We have been able to identify three factors. The first was that all the tools required by the project already existed, were effective, and could be trusted. The second and perhaps the most important was that all the individuals involved possessed a consistent and complete model of that which they were implementing. The third was the quality of the staff involved, who have on average considerably more than ten years system construction experience and a host of large projects to their credit.

GOALS

What did we hope to achieve beyond the construction of an EMAS system for the ICL 2900? This question has a simple answer. We wished to show the viability of re-implementation as an approach to operating systems

construction. Historically, the construction of general purpose operating systems has been an extremely expensive and often unsuccessful exercise. We believe that several of the causes of high cost and low success are the inverse of those previously introduced as a basis for our intended success.

It is incredible that academics as well as manufacturers continue to implement production operating systems in new and often barely implemented languages. The implementation of MULTICS in PL/1, and 4/75 EMAS in IMP are two historical examples. The implementation of the SUE operating systems in the SUE language, and ICL's SUPERVISOR B in S3 are two more recent examples. Having ourselves once faced the problems of writing an operating system on the shifting sands of an initial language implementation, we did not wish to do so again, and believe it should be avoided at all costs. We have also observed that a major problem in operating systems implementation, even with relatively small teams, is that team members do not share a common model of the system, individual parts of which they are implementing. One of the ways that this problem may be overcome is by first building prototype model systems, which can then be developed into products through re-implementation.

TOOLS

A brief section describing the tools used in the project seems necessary. In keeping with the philosophy of the project, there are no new tools being developed, but in contrast with many other operating system projects, the tools have existed from the first day. There are three factors that have made this possible. Firstly, we possess the existing 4/75 EMAS system as a development base. Secondly, we had the advantage of two years software development on and for ICL 2900 computer systems, by several members of the project prior to the begining of the project, which resulted not only in insight into the machine, but in the prior development and extensive checkout of compilers, diagnostic facilities and performance measurement tools. Thirdly we had the advantage of knowing which diagnostic facilities and performance measurement tools were likely to be useful and which were not. The advantages that accrue from these insights should not be under-estimated. There is an unrivalled opportunity in an operating system for the generation of a massive amount of information which by its very scale becomes almost useless. As an aside, it has been our experience as teachers of undergraduates, that no matter how good students are at developing small to medium scale programs, they almost always fail in their first attempt at a large program, because they fail to build in the correct diagnostics and performance measuring tools. No amount of prior warning will convince all but the very best of the need.

The system is being implemented in the same dialect of the Edinburgh Implementation Language (IMP) [ST74] as was the initial EMAS system. This is being done despite the existence of more modern and consistent dialects of IMP [RO77] and the existence of languages such as CONCURRENT PASCAL [HA75] and MODULA [WI76]. There are three reasons for this choice. Firstly, we wish to transport some programs from the existing 4/75 system. Secondly, we understood the language, both how to use it and how to compile it. Thirdly and perhaps most importantly it already existed, not only the language and a robust compiler, but source code formatters, trace packages and post-mortem diagnostic procedures as well.

It is perhaps best to consider diagnostics in two classes, those that are produced dynamically while the system is running and those that are produced statically by a system post mortem procedure after a system failure.

In both cases language based diagnostics can be produced. Language based diagnostics and checks have always existed in IMP. For example code can be inserted by the compiler to check on the validity of array indices or to check that variables have been written to before they are read from, for casting run time error messages in source language terms and for outputing the names and values of all variables on the dynamic stack either at the time of error or on demand. For reasons that we do not understand, it was believed, at the time EMAS was originally implemented, that such checks and output would be either impossible or too expensive in an operating system. This is not the case, and therefore these facilities have been included in the re-implemented system, though of course they can be disabled by re-compilation in production systems to save time and space.

In addition to language based diagnostics, the running system can also produce traces. The most fundamental type of trace is a module trace. Module tracing is extremely easy in any message-based system such as EMAS. All messages go through a single central mechanism, and therefore the cost in code, if not in time, of tracing messages is small. As the flow of messages through the system is high, a certain selectivity is needed. Therefore one is able to nominate the subset of messages to be traced by specifying either their source or destination addresses. Module tracing can be enabled and disabled either from the operator's console or by any supervisory module or process. Nearly all of this was in the original version of EMAS and proved useful. In addition any module can output its own trace information using standard language I/O procedures. In this way other forms of trace are produced, such as page fault traces or channel use traces.

During early system development we were, in the case of a system failure, able produce a small post-mortem analysis without having to resort to a dump. This was achieved by using the restart facility provided by the hardware, to enter a post-mortem diagnostic procedure. Using activation records, and information provided when the system was built, this procedure was able to provide a limited printout containing language based diagnostics, register values, re-assembled machine orders from around the point of failure and annotated tables in a format which matched their structure rather than the width of the print line. In those rare instances when this was not enough, or the post-mortem procedure failed, we had to resort to the tedium of a full dump. We had the experience of de-bugging the first EMAS system using uninterpreted dumps as the main means of fault analysis. It was an unnecessarily slow and tedious process.

As the system became more robust it was felt that a two stage post-mortem procedure was more appropriate, both because there was more information needed which was increasing the size of the resident procedure, but also because greater selectivity was needed in chosing what to analyse. The first stage which is entered via the restart facility dumps the store to tape from which it can be analysed interactively at a later stage. While this is more appropriate in a production system, we would continue to recomend the inclusion of an immediate post-mortem analysis package to suport the early stages of system development.

As the development of the system progressed the emphasis moved from that of error detection and correction to that of performance measurement and improvement. There were three tools available to the project for performance measurement and one that we wished we had. The three tools were a simple module execution profiling facility, an event tracing facility and a remote terminal emulator [AD77] built into the communications network. The tool we did not posses was a language based execution profiling facility.

The module execution profile is generated in the resident supervisor and output on demand. The module dispatcher maintains three counts for each module. These are a count of the number of times the module is entered, a count of the total number of instructions executed in the module and a count of the total amount of time spent in the module. This profile is usefull for determining where time is being spent and possibly being wasted in the system.

The event tracing facility produces an event trace in the same format as that produced by 4/75 EMAS, so that the same utility programs can be used to manipulate it and to build a detailed analysis of resource usage and contention.

The remote terminal emulator is perhaps the most important performance measurement tool we posses both because it allows us to measure the performance of the system as seen by a user at a terminal, and because it provides us with repeatable loads. As the remote terminal emulator sits on the network, an emulated terminal and emulated user are indistinguishable from a real terminal and real user. In addition, as performance data is gathered by the emulator, its use introduces no measurement overheads into the system. The avilability of the remote terminal emulator has made it possible to study the effects of a software change quantitatively rather than qualitatively.

EMAS

We do not propose in this paper to give a detailed introduction to the philosophy, design or initial implementation of EMAS. An overview can be gained from [WH73] and information about various specific aspects of the system from [SH74, RE75, MI75, WI75, AD75, SH75]. It is nonetheless useful to note the various fundamental features that give the system its current appearance:-

1. Interactive working.
   The system is normally accessed from an interactive terminal, which it uses as its major source of control information. Initially this was a teletype directly linked to the system. Currently it may be one of a number of devices attached via a geographically distributed network.

2. Multiple virtual memories.
   Each user is allocated a 16 megabyte virtual address space.

3. Mapped files.
   Files are not accessed via a procedural interface (read block, write block, etc.), but by being associated with a range of virtual addresses and accessed as if they were memory.

4. Controlled sharing of information.
   Initially this was simultaneous file sharing in all modes of access between users of the same machine. More recently a limited form of simultaneous file sharing (read only files) has been introduced between users of a number of interconnected but disjoint systems [SH 75].

5. Transparent memory hierarchy.
   The system operates a four level memory hierarchy (tape, disk, drum, core), but the user is only aware of files and virtual addresses.

6. Minimal user constraints.

The system attempts to restrain the user as little as possible in his or her use of files, languages, virtual addresses etc. There is a set of facilities provided by a standard subsystem, but the user may ignore them and easily provide his or her if he or she so wishes.

7. High user throuput under all loads.
The system cannot allocate more resource than it possesses, but it attempts to allocate as much as possible to the users at all times. There is no thrashing due to use of local scheduling policies [SH74], and higher level bottlenecks have been eliminated where ever possible. For example the file index is multi-threaded and accessed in parallel by each virtual process.

8. Repeatability and enforced fairness.
The resources used by a job are dependent only upon its requirements, and not on other demands on the system. Therefore if a job is run again it uses the same resources. A user should get a fair share of the system determined by its own resource requirements and the number of users of the system. The greater the resources required the lower the priority.

STRUCTURAL CHANGES.

There are four structural changes being made to the system as part of the re-implementation exercise. They are:-

1. A decomposition of device control into three separate functional modules: device configuration, channel scheduling and the translation of logical transfer requests into physical device commands for each device type. The first two used to be included as sub-functions of a device handler whose major function was the third.

2. A re-modularization of virtual memory control from being partitioned primarily by function and secondarily by virtual process to being partitioned primarily by virtual process and secondarily by function.

3. Altering the connection and control of communications front-end processors and spooled I/O devices so that communications and spooled data transfers take place directly to and from paged virtual memory rather than from dedicated real memory buffers.

4. Giving each process three message addresses rather than one. One address being used for initial requests to a process, a second address for replies to a process and a third address for signalling asynchronous events to a process.

I/O CONTROL

The need for the decomposition of device control functions became obvious on EMAS as the complexity of device interconnection increased. This complexity is considerably greater in ICL 2900 computer systems, for there is greater scope for dynamically re-binding peripheral device to particular controllers and device addresses.

In the 4/75 system, a device handler would on initialization determine the configuration of devices for which it was responsible. It would build a data structure, descriptive of this configuration, for its own use in device and channel scheduling. In the re-implemented system both the configuration

function and the channel scheduling function have been factored out of the device handler, leaving it to perform the conversion of multiple logical I/O commands for each logical device into sequential physical device command programs for each real device. (In the case of ICL 2900 drums, the hardware accepts logical I/O commands directly, and therefore no translation is required.) Figure 1 is a schematic of the old and the new structures.

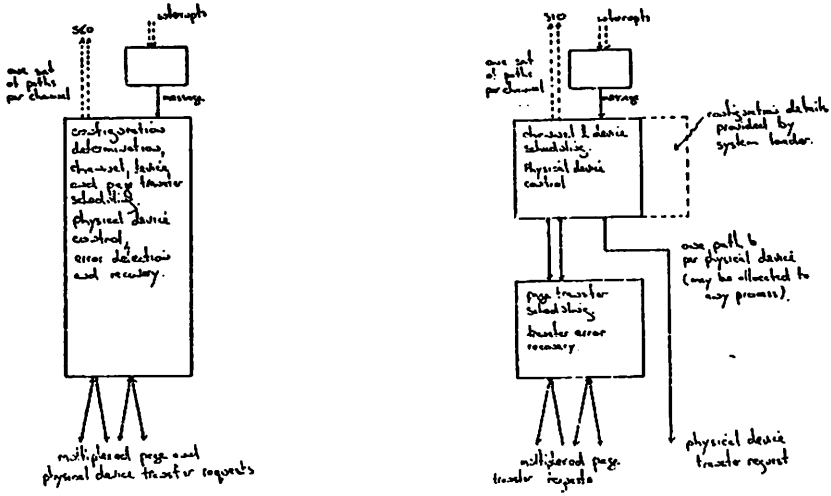


figure 1

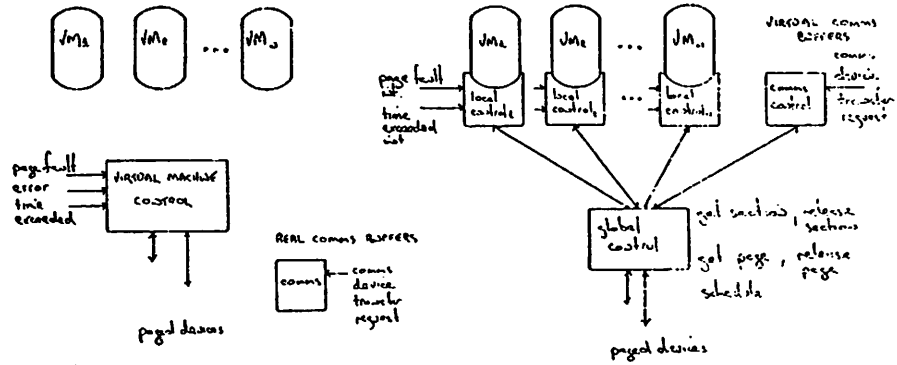
In the re-implemented system, device configuration information appears in a read only segment which can be considered as part of the abstract machine on which the system runs. This is achieved by loading the system in two stages. Initially a small cut-down supervisor is loaded, which builds a data structure corresponding to the configuration. To do this it employs an inordinately complicated sequence of operations to determine the model and number of processors, memory boxes, memory modules, I/O controllers, the devices attached to them and all their possible interconnections. In addition to building this data structure, the cut down supervisor responds to a small subset of operator commands, (ie. to perform device transfers). One of these commands causes the full system to be loaded from a nominated file, passed the read only segment describing the configuration, and then entered.

VIRTUAL MEMORY CONTROL

The re-implementation of virtual memory control (paging) is perhaps the most fundamental change being made between the existing 4/75 and the re-implemented 2900 system.

In EMAS, unlike other fully paged systems, page replacement is not a global function performed over all available page frames in the system, but a function performed separately, for each virtual process in the multi-programming set, over a set of page frames allocated to each virtual process upon its entry to the multi-programming set. This functional organization is not manifest in the structure of the 4/75 system, where page replacement and other virtual memory control functions are performed by modules in the resident supervisor. In the re-implemented system, there is

instead a module, called a local controller associated with each virtual process in the system. Each local controller is only active while the virtual process with which it is associated is in the multi-programming set and is paged out when the virtual process is removed from the multi-programming set. Figure 2 is a schematic of the old and the new structures.



4/75 EMAS 2900 EMAS

figure 2

In the old structure each module was serially re-entrant and non-switchable. It was invoked by a message, and had to multiplex operations on behalf of many virtual processes.

In the new structure, each module is also serially re-entrant, though the code may be shared by a number of local controllers running in parallel. Each local controller is entered via a trap from the associated virtual process (a page fault is a synchronous event and not an interrupt) or upon a return from a call made by a local controller upon the global controller.

The global controller is effectively a monitor, or series of monitors, that handles page movement between the various levels of the storage hierarchy, and the allocation of resources to a local controller for a period of process time. The global controller does not enforce this allocation, and depends upon the honesty of the local controllers in staying within their allocations. Local controllers must therefore be trusted programs and cannot be provided by a user at will.

NON-VIRTUAL USER I/O

The handling of non-virtual I/O primarily with interactive terminals but also with non-paged magnetic tapes, had always been a problem in EMAS. Spooled I/O, to line printers and from card readers etc., is not a problem as the user operates to a file interface. He nominates a file to be output, and accepts a file of input. The problem is to give the appearance of mapped I/O for interactive traffic.

Two approaches were used in the 4/75 EMAS system. In the first, DIRECTOR copied data between a user's virtual memory and dedicated buffers in locked down real store. The transfer was performed in response to commands issued by the user's process and external I/O activity. This approach carried with it a high overhead, as a number of DIRECTOR pages were required to perform

this operation, and if the data transfer was larger than the buffer size the process would be paged in and out of store merely to refill the buffer rather than to procede. This had the effect non only of incurring paging activity, but also of altering the identity of the user's working set.

This approach was modified with the advent of network operation, so that DIRECTOR transfered data between virtual memory and inter-process messages, which are sent to or received from the communications front-end processor. As these messages are small all interactive traffic has to be fragmented into 18 character packets. As only limited number of packets (10) may be sent ahead without acknowledgement, similar unneccessary paging activity is introduced for transfers of over 180 characters length.

In the re-implemented system we make an attempt to overcome this problem, and to integrate the handling of RJE traffic from the SPOOLing manager and inter-machine traffic within the same scheme. The SPOOLing manager currently multiplexes large blocks into the flow of inter-process messages to and from the front-end.

In the new scheme a special virtual memory is maintained by a special local controller called the "communications controller". This virtual memory is not accessed by a virtual processor, as is a user's virtual memory, but by communications front-end processors and spooled peripherals to perform data transfers. The virtual memory controlled by the communications controller contains files which in turn contain input or output buffers. Pages of these files are transfered into primary memory by the communications controller in response to peripheral transfer requests, in the same way that a local controller transfers a page to primary memory upon the occurance of a page fault. A page brought to primary memory by the communications controller needs only remain in primary memory until the transfer which operates as a burst completes, though there is a delay involved before the communications controller releases the page. This allows a page to be reclaimed while still in primary memory, so that either the next burst transfer can procede directly or a process sharing the file can claim the page before it is removed from memory.

In the case of interactive traffic these communications buffer files (one for and one for output) will also be connected into a user's virtual memory. The user will read input from the input buffer file and will write output to the output buffer file. Synchronization being performed by director calls which in turn cause inter-process messages to be sent to a front end processor via the communication controller. In the case of spooled I/O, the files to be input ot output will themeselves serve as communications buffer files. They will not be connected into another virtual memory at the same time.

The advantages of this approach are that no pages need to be permanently tied down in main store and that the unit of transfer between the front-end and the main-frame can be appropriate to that logical interface, while the unit of transfer handled by the system remains a page. As all page sharing is handled by the global controller, a user's local controller will normally request the relevant input page before the communications controller releases it. In the case of output, only one buffer page is paged in if more than one burst transfer to the front-end is required and not the user and director processes as has previously been the case

THREE ADDRESSES PER PROCESS

The change to giving each process, both system processes and virtual processes, three message addresses is a simple one that perhaps does not require its own section, but the simplification that derives from this trivial change argues for its inclusion.

In the 4/75 system, a process is invoked by a message addressed to it. If a process can receive a number of different messages,the body of the process will take the form of a case statement (Figure 3). As long as the process does not itself need to interact with another process this poses no problems. If it does, it will receive a reply multiplexed into the incoming message stream. As a system-process must relinquish control before another system process can be invoked, a single activity punctuated by a request on another process and the reception of a reply must be implemented by two arms of the case statement. If the process requires to maintain the context of the first arm while awaiting the reply, it must either queue all other messages received before the reply, or possess a means of saving multiple contexts. Neither approach is particularly satisfactory. By giving each process three addresses, we allow it to receive synchronous requests at one address, replies at a second and asynchronous events at a third. The system also provides a means of inhibiting and enabling a message stream by destination address. Using this facility a process may inhibit the arrival of requests while awaiting a reply, thereby using a simple global mechanism to eliminate the need for message queueing within the process or the handling of multiple contexts. Figure 4 is a schematic of the new structure.

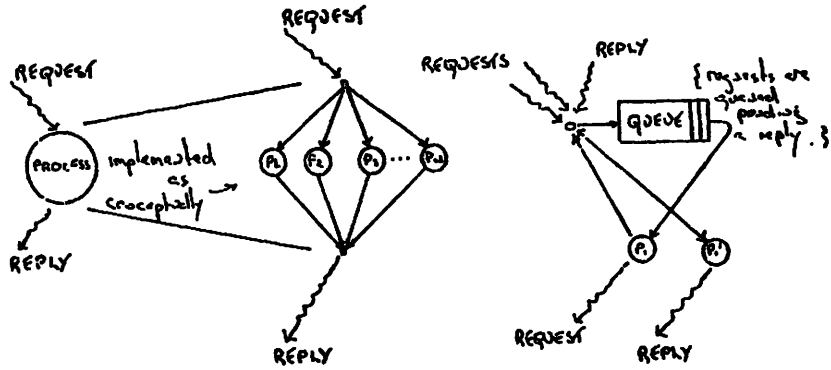


figure 3

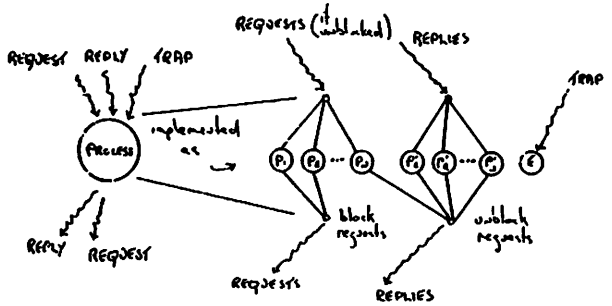


figure 4

CONCLUSION

We have described an operating systems re-implementation project, a rationale for re-implementing operating systems, and the structural changes that are being made to a particular system. At the time of writing (April 1978) the project is on schedule. All major functions have been implemented, and attention is now being paid to the final stages of system performance evaluation and tuning.

REFERENCES

[AD75] Adams, J.C. & Millard, G.E.,
'Performance Measurement on the Edinburgh Multi-Access System',
Proceedings (ICS 1975), Gelenbe & Potier eds., North-Holland

[AD77] Adams, J.C., Currie, W.S. & Gilmore, B.A.C.,
'The Structure and Uses of the Edinburgh Remote Terminal Emulator',
Edinburgh Computer Science Department Report (CSR-12-77)

[BE76] Belady, L.A. and Lehman, M.M.,
'A Model of Large Program Development',
IBM Systems Journal, Vol 15 No 3, 1976

[HA75] Hansen, P.B.,
'The Programming Language Concurrent Pascal',
IEEE Transactions on Software Engineering, Vol 1 No 2, Sept 1975

[HO72] Hoare, C.A.R. and Perrot, R.H., eds.,
Operating Systems Techniques,
Academic Press, 1972.

[MI75] Millard, G.E., Rees, D.J. & Whitfield, H.,
'The Standard EMAS Subsystem',
Computer Journal, Vol 18 No 3, 1975

[RE75] Rees, D.J.,
'The EMAS Director',
Computer Journal, Vol 18 No 2, 1975

[RO77] Robertson, P.S.,
'The IMP-77 Language'
Edinburgh Computer Science Department Report (CSR-19-77)

[SH74] Shelness, N.H., Stephens, P.D. & Whitfield, H.,
'The Edinburgh Multi-Access System,
Scheduling and Allocation Procedures in the Resident Supervisor',
RAIRO (Informatique & Computer Science), B3, Sept. 1975

[SH75] Shelness, N.H. & Yarwood, J.K.,
'An Operational Method for Achieving Dynamic Sharing of Files
in a Distributed Interactive Computing Utility',
IIASA (WP-76-9), 1976

[ST74] Stephens, P.D.,
'The IMP Language and Compiler',
Computer Journal, Vol 17 No 3, 1974

[WH73] Whitfield, H. and Wight, A.S.,
'The Edinburgh Multi-Access System',
Computer Journal, Vol 18 No 4, 1973

[WI75] Wight, A.S.,
'The EMAS Archiving Program',
Computer Journal, Vol 18 No 2, 1975

[WI76] Wilkes, M.V.,
'Software Engineering & Structured Programming',
IEEE Transactions on Software Engineering, Vol 1 No 4, 1976

[WI73] Wilkes, M.V.,
'The Dynamics of Paging',
Computer Journal, Vol 16 No 1, 1973

[WI76] Wirth, N.,
'MODULA, A Language for Modular Multi-Programming',
ETHZ, technical report 18, March 1976.