The University of Warwick

COMPUTER CENTRE REPORT

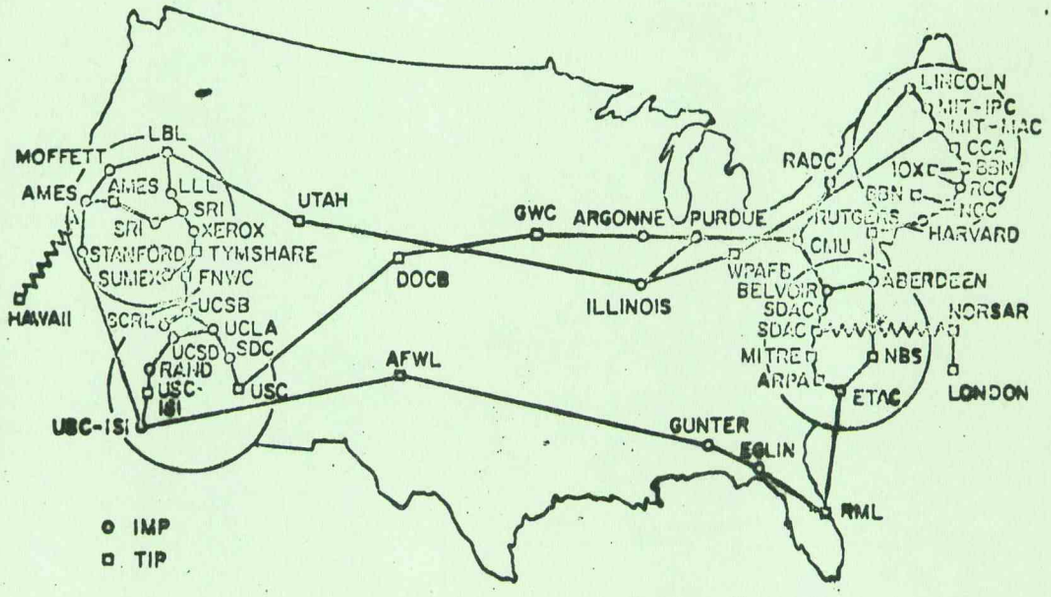
No.10

CURRENT RESEARCH IN OPERATING SYSTEMS AND COMPUTER NETWORKS

(A Personal View)

by

COLIN WHITBY-STREVENS



ARPA Network Nodes

31 March 1975

(From Bolt Beranek and Newman, Inc., Report #3063

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
Warwickshire.

JULY 1975

CURRENT RESEARCH IN OPERATING SYSTEMS AND COMPUTER NETWORKS

(A Personal View)

COLIN WHITBY-STREVENS

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
Warwickshire.

JULY 1975

CONTENTS

CONTENTS *PAGE*

Note

This report presents my personal view of the activities of many people in many places. In a number of cases the work is in an early stage, and in others my summaries are based on partial information. To this is added a human element. It is thus likely that this report contains bias and errors of fact. I hope that these do not outweigh its usefulness, and I apologise for any errors of commission or omission. Equally the opinions quoted and the conversations reported belong to the named individuals and should not be taken to imply the policy of any organisation or institution.

Acknowledgement

The preparation of this report has received financial support from the Science Research Council and the William Waldorf Astor Foundation and I wish to thank both organisations accordingly. I have also received indirect support from the University of Warwick and the University of Connecticut. In particular I would like to thank Prof. T.L. Booth and the faculty at UCONN for making us so welcome. I devote a separate page to thanking the many people who have given me of their time in discussion and advice. Finally I wish to thank my wife and daughter for their patient support, tolerance and encouragement during this three-month period.

C. Whitby-Strevens
Mt. Haystack
Vermont
U.S.A.

July 1975

Personal Discussions

At various stages in the preparation of this report I have had personal discussions with the following:-

C.G. Bell,	Digital Equipment Corporation
T. Bredt,	Stanford University
P. Brinch Hansen,	California Institute of Technology
J. Burchfiel,	Bolt, Beranek and Newman
J.N. Buxton,	University of Warwick
V.G. Cerf,	Stanford University
D. Clark,	Massachusetts Institute of Technology
B. Croxon,	Digital Equipment Corporation
R. Eckhouse,	Digital Equipment Corporation
R.S. Fabry,	University of California, Berkeley
R.J. Feiertag,	Massachusetts Institute of Technology
A.G. Fraser,	Bell Laboratories
R. Greenblatt,	Massachusetts Institute of Technology
M. Hammer,	Massachusetts Institute of Technology
C.A.R. Hoare,	Queen's University, Belfast
M.R. Kraley,	Bolt, Beranek and Newman
T. Knight,	Massachusetts Institute of Technology
Barbara Liskov,	Massachusetts Institute of Technology
W.C. Lynch,	Case Western Reserve University
A. Osborne,	Osborne and Associates, Berkeley, California
C.V. Ramamoorthy,	University of California, Berkeley
D.M.R. Park,	University of Warwick
D. Richie,	Bell Laboratories
R. Schantz,	Bolt, Beranek and Newman
A.C. Shaw	University of Washington
D. Siewiorek,	Carnegie-Mellon University
M. Spier,	Digital Equipment Corporation
R. Thomas,	Bolt, Beranek and Newman
H. Weber,	Massachusetts Institute of Technology
S. Wecker,	Digital Equipment Corporation
W. Wulf.	Carnegie-Mellon University.

In addition I held useful conversations with research students at several of the above universities. I wish to thank all of these for giving me their time. In addition many of the above offered hospitality and helped me with arrangements when I visited them and I am accordingly grateful.

Introduction

I have spent much of the last six years contributing towards our local operating system research effort [39]. This project is now all but complete and so I find myself looking back over it, and also looking forward to new horizons. I have also looked outward (something it is all too easy not to do) during the past three months to see where others have taken the art whilst we were developing the Warwick system, and this report is the result.

This report thus contains descriptions of completed and ongoing work in areas that to me are interesting and/or relevant. During the time that I spent compiling this report, a number of ideas crystallised concerning future research possibilities at Warwick. Hence "relevant work" means work that is clearly relevant to the research proposal which is a companion to this report. "Interesting work" is work that relates to the completed Warwick project, or work that is not in a sufficient stage of development to have produced ideas and results that could be used in our new project. Clearly that is a situation that may change at any moment.

Various categories of work are not included:-

1) This report was prepared whilst I was on sabbatical leave from Warwick and based at the University of Connecticut. I have therefore concentrated on reporting activities in the U.S.A.

2) A significant omission is a description of the current state of database technology. I am not even an amateur in this field, and so consider myself inadequate to include it in this survey. I also feel that it can be treated largely as a separate area. However there are several items in this report in which current developments place constraints on database design. I look forward to reading a lucid account of the database art when the many current contentions resolve themselves.

3) I have not tried to be exhaustive. Plainly there are many people all hammering away at their own systems. Much work tends to be of the form "I took this little particular problem and solved it this way", and while this is obviously necessary to develop the start of the art, I have chosen to include those projects of perhaps wider significance, or direct relevance. In addition there are probably numerous activities that I quite simply have not come across.

4) I have visited a number of centres, and held valuable conversations with various people. These personal contacts tend to dominate, and often the work reported is ongoing (and possibly not yet published). I would have liked to have visited other places, met other people and discussed their work, but various technical (non-intrinsic) reasons prevented this. The net result is the current unbalanced opus.

The information contained in this report comes in two classes:-

The first is the product of visits to various centres, and attendance at the Reliable Software Conference in Los Angeles. I have included summaries of the many conversations that I have held, as it is not just the research that is important, but the view the researcher has of his work and of the world, and the context in which he works. I have been particularly struck by the quality of the graduate students I have met. Frequently they have shown the greatest understanding of current problems and the greatest originality in deriving solutions. A not insignificant part of this report is concerned with work that was/is being carried out as part of a Ph.D. programme. Much of the work reported here is of a tentative nature and may well change shape radically by the time it takes on a final form. I have also quoted many opinions, which are probably even more tentative and liable to change.

The second source is the product of quite a wide background reading exercise. Some ideas came to fruition shortly after we embarked on the Warwick project, and are clearly so relevant to a new project that I have included appropriate summaries. In addition several survey papers have appeared and are thus of value to anyone interested in the ideas reported here.

I started out with interests in operating system structures, complete systems (not just highly optimised scheduling (e.g. paging) algorithms) and methodology. I have discovered a technological revolution which is plainly going to have a powerful impact on the way we use computers, a greater understanding of reliability, and a wide interest in computer networks. Closely related to this is the development of distributed systems, and it is in this area that I propose further research work at Warwick.

The report commences with a large general section on operating systems, a section on protection and security, and a section on methodology. We see the acceptance of the "kernel" idea (e.g. [109, 118]), the separation of mechanism and policy, and a questioning of Dijkstra-style hierarchies. I feel that perhaps the greatest advances have been made in the area of methodology, and nearly all these advances are concerned with data abstraction. We have known how to use "control abstractions" for some time, but are only just coming to grips with data abstraction. We are also coming to understand the problems of reliability, its relationship with correctness, and in particular that the two are not equivalent. Correctness is a useful, though not crucial, component of overall reliability. Reliability has to be assessed within an operational context. Computer networks are now moving from the speculative stage into the implementation stage. It is plain that any future development of operating system software should not ignore the rôle of networking. The purpose of networking is not primarily to share processing resources, but to share information. Everybody is building networks, but as yet nobody really knows how - we lack any formal, or "high level", framework in which to assess networking issues. The report provides a basic pragmatic background to an understanding of networks, and refers to some of the most significant work under way. One important issue is the transparency of the network

(is the user aware of the different types of computers and systems in the network). A totally transparent system becomes a "distributed system". Such systems offer economic advantages, and significant improvements in reliability. The report concludes with a brief mention of the impact of L.S.I. and the potential for "personal computers" interconnected in a network. Such systems remove the need to write c.p.u. and memory multiplexing operating systems. Such systems (using concepts such as "virtual memory") will become interesting museum pieces in the history of computing.

The report takes the form of collected items under the seven major headings. Firstly I would point out that many of the items span several headings, and so this format is not well defined. Secondly I would have liked to have taken each issue and gathered together all appropriate comments and contributions, presenting an orthogonal version of this report, but pressure of time does not allow.

My main purpose in writing this report is to reflect on work proceeding in parallel with the Warwick operating system project, and to gather together material to form a base for a new project. However, I hope that the report, imperfect as it is, will have a wider usefulness. I would emphasise that all responsibility for mistakes and misrepresentation is mine alone and I apologise to any who are consequently offended . Logistics have prevented the various people mentioned from checking the descriptions herein, though I have tried to capture the direction and spirit of their work.

Operating Systems - Structure and General Discussions

This section is a mixed bag of reports and discussions. Since Dijkstra proposed semaphones and Brinch Hansen described a system of interprocess message communication [11] a number of real systems have been built, and the experience is reflected here.

There seems to be a trend away from the strict Dijkstra-style hierarchies [26]. One important trend is towards the separation of mechanism and policy (see the Hydra system). It is interesting to note (elsewhere) that the Multics experience is that mechanism can be viewed hierarchically, while policy cannot. In any case we should not try to let one structure reflect the communication relationships,protection relationships, synchronisation relationships and resource allocation relationships in a system. It seems that we need different structures for most of these.

Relevant papers and books not explicitly mentioned are [48, 50, 51, 59, 115]. Another important trend is towards the "data separation" of processes. All recent proposals and discussions take Brinch Hansen's line that separate processes communicate via messages wherever possible, and that where it is desirable to share a data structure then this should be via a formalised "monitor" structure. Many proposals use separate processes instead of monitors, and thus all communication is carried out via a message system. The practicality of this approach allows the construction of distributed systems (using several computers connected only via communications lines).

From the point of view of the design of the algorithms contained within the individual processes, several systems we have seen are constructed along the lines that it does not matter if a process is activated in error - it checks on the work it has to do. Reliability is further enhanced by activating all processes from time to time, thus preventing seizure if an "activation" command is lost. Another technique used increasingly is that of persistence - particularly in networking systems.

Survey of Design Goals for Operating Systems

This 3-part paper by Abernathy et al [2] reports the results of a literature search on the subject "design goals for operating systems". In addition, it reviews design goals for specific operating systems and develops a general set of operating system design goals. Special purpose design goals and conflicts among design goals are also discussed.

Abernathy and Co. conclude :-

a) a significant treatment of operating system design goals and objectives does not exist!

b) a list of design goals can be developed, e.g.

I General Design Goals

A Primary Goals

1. Maximise system efficiency
2. Maximise system generality
3. Minimise system operating errors

B Secondary Goals

4. Maximise system transparence
5. Maximise data security

II Marketing Design Goals

6. Maximise customer satisfaction
7. Maximise appeal to potential customers

III Specific Design Goals for On-line Systems

8. Process on-line data
9. Optimise communication control

IV Specific Design Goals for Multi-access Systems

10. Maximise remote-access
11. Minimise response time

V Specific Design Goals for Control systems

12. Provide ability to maintain and/or regenerate the system on-line

VI Miscellaneous Design Goals

13. Minimise Development Effort

c) These goals are mutually conflicting. The operating system designer must establish relative priority for each conflicting goal.

d) Most operating systems will probably have goals 1, 2, 3 as listed in (b) above. However other goals (not listed in (b)) may be considered.

They recommend that priority weighted goals for each operating system project should be properly documented and published. They also recommend a deeper level of research be undertaken on the subject of design goals for operating systems. They also note the lack of study of operating systems design parameters, trade-off analysis techniques, and languages for the design and implementation of operating systems.

Knott's Process Management and Intercommunication Primitives

Gary Knott has brought together a number of current ideas in operating systems design in the form of a proposal for process management and intercommunication primitives [62]. He attempts to attain a realistic set of concepts, and considers carefully the rôle of peripheral devices, interrupts, and memory sharing and protection. The process management primitives are

CREATE
SUSPEND
RUN
CONTINUE
FREEZE
UNFREEZE
DESTROY
STATE

Freeze and unfreeze are used to enforce synchronisation on possibly unco-operative processes (e.g. a program under observation by a debugging system).

Process communication is based on an elaboration of Brinch Hansen's message system [11]. Each process has a set of input and output message ports. An input port is able to accept only a finite total of messages. One feature of the system is that it is interrupt-like. When a message arrives at a port, control is transferred to a predetermined location. The notion of the interrupt pervades the message system. I don't approve.

Knott discusses problems of broadcasting, removing messages after sending, dealing with unreceived messages, non-existent processes etc.. He has a system for enabling and disabling input ports. The list of process communication primitives is

RECEIVE (this sets up a port)
SEND
DISABLE
ENABLE
UNSEND
UNRECEIVE

Knott discusses his proposal at length in the light of various situations that arise in operating systems.

HYDRA

A team, led by Bill Wulf, at Carnegie-Mellon University Pittsburg have developed an interesting system called HYDRA [118]. This system runs on a multiprocessor PDP-11 configuration operating with a potentially vast shared memory. Details of this shared access mechanism and the processor-processor communications are plainly influenced by the base hardware chosen. However the overall design philosophy of HYDRA is machine independent and of significance as it crystallises a number of concepts proposed elsewhere, as well as possessing some original features. Within a couple of years of it first starting to tick, HYDRA seems likely to earn a place in the classical history of operating systems.

The overall goal of the system is to provide a set of facilities for the convenient, flexible, efficient and reliable construction of co-existing operating systems. HYDRA thus serves as the host for "exploration of the space of user-visible operating environments". In fact HYDRA is the operating system kernel or nucleus. The design of HYDRA is based on the following principles:-

1. Multiprocessor environment. Envisaged applications for HYDRA implied that multiprocessor operation would be needed. Part of the HYDRA project is the investigation of the potentials and problems of multiprocessor systems.

2. Separation of mechanism and policy. The kernel provides mechanisms for controlling the system resources, but scheduling algorithms etc. belong to the operating environments that HYDRA supports and have no place in the kernel.

3. Integration of the design with the implementation methodology. The methodology is a hybrid of Dijkstra-style structured programming and Parnas-style modularisation.

4. Rejection of strict hierarchical structures. Wulf points out that there is no reason to suppose that the same (hierarchical) structure should exist for control as for resource allocation, or protection, etc..

5. Uniform protection structure. All resources are protected using a variation on the capability concept. HYDRA supports (and ensures the integrity of) a capability mechanism, but is flexible enough to allow external interpretation of the protection conditions.

6. Reliability. HYDRA should not only be correct, but should be able to detect and recover from errors that do exist (both hardware and software).

The mechanisms provided in HYDRA center on the abstract resource. Many "types" of resource are available, and new types can be defined dynamically. A second key concept is that of an execution domain, and finally the protection mechanism that controls access to resources from within execution domains.

An object is the realisation of a resource. Each object consists of

(i) a unique name

(ii) a type

(iii) a representation.

The representation is subdivided into

(iii)(a) data

(iii)(b) capabilities.

A capability consists of a reference to an object together with a collection of access rights to that object. Capabilities are unforgeable, being manipulated only by the kernel.

HYDRA has three types of objects to handle programs. The first is a procedure - corresponding roughly to the conventional concept of some code + data. The capability part of a procedure object contains capabilities for other objects that the procedure might wish to access, together with capability templates - containing only the type of object, not a specific instance. This is filled in when the procedure is called, and allows a calling procedure to pass an object as a parameter for the called procedure access.

The second object for program manipulation is called the LNS (Local Name Space), which is the same as a procedure, save that it represents the procedure under execution with all the templates filled in etc.. The LNS is created when the procedure is invoked and is deleted when the procedure exits. The LNS forms in effect an execution domain. A change of procedure implies a change of execution domain, and so incurs more kernel overhead than in a conventional language.

The third object for program manipulation is the process. This corresponds closely to the now well accepted concept and HYDRA provides the usual repertoire of message-buffering and semaphore operations. One way of viewing a process is to regard it as a sequence of changes of LNS's.

At any instant in time we see that HYDRA is administering a number of processes, each of which is executing in a separate LNS, and thus is able to access the objects described by the capabilities of the LNS within the rights specified by the respective capabilities.

The kernel performs all checking of rights. In fact the rights of a capability fall into two sets - those that are common to all types of objects, and those that are dependent upon the particular type of the object the capability refers to. Each "right" in a capability has the privilege of being able to call any procedure that is a member of the class of procedures associated with the right. For rights that are independent of object type, then the kernel defines the classes (and hence the procedures that can be called, and so finally the semantics of the right), but for rights that are type-dependent then the class is defined by the creator of the type (a type is an object of type TYPE!)

One interesting type-independent right is the "path" right, and provides an extension to a LNS. The LNS (an object) has capabilities to access a number of other objects (of various types). If one such capability includes the "path" access rights, then the LNS is also allowed to access the capabilities of the object referenced by this capability. Hence the LNS can access the objects accessible to an object it can reference (provided a "path" right exists). This extends to an indefinite "depth", though a "path" right must exist at each level of indirection.

The topic of access rights is completed by the CALL and RETURN primitives provided by the kernel. The CALL primitive provides the mechanism for entering a procedure, and deals with the necessary change of LNS. The new LNS is made up of the procedure being called together with capabilities passed as parameters by the calling procedure. This only takes place however after a certain amount of checking, which revolves around the template capabilities (which may be thought of as formal parameters) and the actual capability parameters of the call. Firstly there is conventional type-checking. Secondly there is a check that each actual capability parameter contains certain access rights. These are specified as part of the formal parameter specification. Provided both these checks are validated, then the actual capability inserted in the LNS contains access rights which are also specified in the parameter specification. This provides an adequate level of access checking, but has the unusual advantage of being able to expand access rights across a change of LNS. On reverting to the original LNS (on procedure exit - RETURN) the original set of access rights are re-instated.

We have described at length the protection mechanism of the HYDRA system, and its relation to the other concepts. We would re-emphasise that this is just a mechanism. It is designed so that independent operating environments can co-exist in isolation, but within one such environment any desired security policy can be implemented. "High level" data sharing systems can be implemented with any degree of security. For example, the protection mechanism can support a security system for mutually suspicious processes. One concept that has been consciously avoided in HYDRA is the concept of ownership. This is seen as a "high level" concept which may or may not be present in a security system. It is seen as having no place in the base facilities provided by the kernel.

We see the HYDRA proposal as an interesting and valuable contribution to the understanding of operating system structure. Mechanism and policy are clearly demarked, and the mechanisms proposed, while not necessarily "the most primitive adequate set", clearly are practical and capable of supporting a wide range of operating systems, each with its own scheduling policies, security systems etc. - i.e. each with its own user environment. The principle that the kernel should not seek to impose an overall system structure (e.g. hierarchical) seems equally to be a part of the commonsense clear thinking that characterises this system.

Discussions with Wulf and research students involved in Hydra

Methodology

All programming has been carried out in BLISS10. The byword is "Modular decomposition" [80], which allows design, coding and debugging to be carried out in parallel. Programmer productivity has been measured at 26 instructions of machine code per day overall. Certain compromises have to be made at times for the sake of efficiency, but they are few and far between. The project has proceeded through 3 to 4 iterations (not counting support tools), and the HYDRA kernel has been running for about 2 years.

One result of the methodology used is that iterations between programmers (and the modules they write) is minimised, and as a consequence productivity has remained linear, despite the iterations, as the system has grown. (One might expect a tailing off of productivity as the system size increases). Wulf has developed a structured programming language called ALPHARD [117]. He is interested in the symbolic manipulation of computer descriptions, e.g. the automatic generation of manuals. He sees the issues in hardware design as being similar to but much harder than those of software design. In this ALPHARD has become a machine description language. Wulf would like to build a "true" compiler compiler, with a formal code generation section.

General

Wulf believes that a general problem in industry is that too much intellectual effort is spent in forcing a program into the mould of (unsuitable) facilities provided by an operating system. In HYDRA only the kernel is really fixed. It is easy to build a new subsystem (e.g. filing system). This property is the direct result of the policy/mechanism separation philosophy. In fact the kernel provides (i) a clean protected image (ii) parameterised policies. The kernel contains a system called the kernel multiprogramming system (KMPS) which operates according to parameters set by a policy module (external to the kernel).

Each sub-operating system contains a policy module whose job it is to share the resources allocated to the sub-operating system between the tasks it controls. Each policy system is given a time guarantee and a space guarantee. Each process has its own space and time requirements, and the policy module determines the appropriate percentage of these resources that the process should receive in the long term. Short term scheduling (timesharing) is performed by the kernel in such a way as to reflect the expressed desires of the various policy modules.

Still to be developed are improvements to the guarantee system, so that more resource may be given than has been guaranteed on the understanding that it may be taken back in order to prevent deadlock, and a system of aliases for capabilities.

There are several noteworthy features of the hardware architecture. Perhaps the neatest is that the swapping disc has the property that 1 page (8K bytes) fits one revolution exactly. All the counters in the logic are "modulo 8K bytes" and a happy result is that a page transfer can start the "next" disc segment, and continue for one revolution. Hence there are no latency delays associated with this disc. In fact the quickest way to zero a page of core memory is to do a disc transfer.

The hardware architecture is based on a large memory accessed via a crossbar switch from all processors. Each processor can generate an 18 bit address, which is divided into a 5 bit page field and 13 bit displacement field. This is fed into a relocation box on the same unibus (also on the unibus are local store and local peripherals). The relocation box generates a 25 bit address (12 + 13) to the crossbar switch and hence to the shared memory. Each processor thus has a relocation box (which occupies 3 boards, including parity generation and checking). Unfortunately the relocation boxes have given trouble (someone blamed hardware design problems) and the system has a m.t.b.f. of 1½ hours.

As a result the kernel is still not 100% reliable, and other fairly important general purpose software (such as a decent command program) is yet to be developed.

The system also employs a separate processor-processor chain. This is used to allow one processor to stop all others, so that data structures can be checked etc.. It is also used to send inter-processor interrupts, which allow, amongst other things, ready access to all peripherals from any processor.

The two largest problems faced by Wulf and his team are stated as :-

(i) The use of a commercial machine. Much effort has been spent in getting round a multitude of odd little limitations, such as the restricted addressing field, the halt instruction etc..

(ii) The use of a crossbar switch. This is a prime target for reliability problems as all the system depends on it. Also it involves some delay, and if two processors access the same memory module there is an additional 1.5 microsecond delay. However, these problems are not considered crucial as there are better techniques now developed elsewhere which overcome them.

As to the future, much effort is required to make the system more user oriented. So far the main effort at the user level has been concerned with the speech analysis software. The topic of most significant research interest is to study the decomposition of computing tasks in order to take advantage of parallel instruction sequences.

The Hierarchical Problems of Bredt and Saxena

Bredt and Saxena, of Stanford Digital Systems Lab, have been looking at the limits of applying Dijkstra-type hierarchies to operating system structures [10]. In particular they wish to design a system in which the number of processes is so large that it is undesirable to retain all of the process state information in main memory at all times. Equally each process may have access to a large main memory.

The Dijkstra hierarchical structure is inadequate as the process management level is higher than the memory management level and hence cannot use the virtual memory resources for the process bases. If it cannot "see" all the process static then it cannot operate, as there are no higher levels to call upon. Exchanging these two levels provides a feasible but unattractive solution - as it is impossible to overlap the operations of the memory manager with anything else.

The solution proposed by Bredt and Saxena [93] is a double two level system :-

Level 1) Simple Scheduler for a fixed number of concurrent processes.

Level 2) Simple memory manager providing a virtual address space for a fixed number of processes.

Level 3) General Scheduler for an indefinite number of processes.

Level 4) General memory manager for an indefinite number of processes.

They discuss the design of each of the four levels in detail and provide a set of procedures written in extended PASCAL using monitors as suggested by Hoare [52].

Comments on UNIX by Richie

The UNIX system is well described in the paper by Richie and Thompson [88]. The system has a number of similarities to the Warwick Modular One system and is therefore of interest.

There are currently (June 1975) some 50 UNIX installations. They are mainly in the academic world, but some U.S. defense establishments are also interested. The system was re-written in 1973 in the language C (a grandchild of BCPL [87]). Only the interrupt handlers are written in assembly language.

As much as possible has been written at the user level as utilities, e.g. the login program, editors etc.. The main mechanism for program creation is a sub-process spawning operation. An interesting feature is that the synchronisation and scheduling system is similar to that of the Warwick system, in that it does not use a multiplicity of queues, but relies instead on process states.

The main development effort in UNIX was in defining a good user interface. The system is now stable - the only work is maintenance. Like all good systems it is issued complete with documentation and character versions. The next issue is expected to need two discs as there will be more utilities.

The file system is of interesting design. It is structured on a random-access basis within individual files (as against the more usual bias towards sequential access). This allows "random-access" editing to be carried out efficiently. Another feature is that, from the technical point of view, there is no need for a read-write lock - the system will maintain consistency as far as the operations performed will allow. Of course in certain circumstances a read-write lock is desirable. The same mechanism is used for a form of disc buffering called the pipeline. A process (e.g. an early compiler phase) can output information to a pipeline which another process (e.g. next compiler phase) can read. The system synchronises the processes so that the reading process will wait if necessary for the writing process, but otherwise there are no restrictions placed on the use of the pipeline, e.g. the two processes could be run sequentially if desired.

The main use of UNIX at Bell Labs is for text processing and typesetting. UNIX recurs elsewhere in this report in the context of Fraser's "Spider" network.

(Footnote: Part of a demonstration of UNIX given to me included an interactive speech synthesiser which earned high marks in a nearly correct pronunciation of my surname!)

Discussions with W.C. Lynch

Case Western have developed an operating system based on a Univac 1108 (196K x 36 bit words) [71]. This is run by CHI on a commercial basis, with 25% of capacity reserved for the University.

Lynch has carried out a number of monitoring experiments [72], and discovered that many aspects of the system are c.p.u. limited. As an example he discovered that some 8000 P or V synchronisation operations were being performed per second. This alone takes about 5% of the computer time. Modules such as the data-management system are c.p.u. limited. Simply plugging in more c.p.u. power is not the immediate answer as the administrative overhead is increased. Instead, work is about to start at Case Western to see if a well structured distributed processor system can help. Lynch believes that there is a danger of buying problems in hardware, and feels that interest might arise in "high-level" languages for microprocessors.

Lynch has used a methodology similar to Brinch Hansen's precedence graphs [13]. However Lynch adds numbers to the arrows to see how many activations might be made along a certain path. This is an important factor as there is a danger that a well structured system with good concepts might incur too much run-time overhead. Lynch thus is sympathetic with Hoare's approach that as much checking as possible be performed once and for all at compile time. This implies unfortunately that there is a static number of processes in the central parts of the system.

On the subject of system tuning, Lynch believes that it is perfectly possible to devise algorithms that will work well in a variety of environments. The difficulty is in determining the characteristics of these environments, and in finding enough of them. In this context Lynch is working on an in-depth study of some half-dozen operating systems together with how they work and how well they work.

Discussions with Richard Eckhouse

Eckhouse is the project leader of the operating systems research and development group of Digital Equipment Corporation. He is leading a project generally concerned with multi-processing systems. DEC have been experimenting with designs ranging from networked systems to full multi-processor systems. The approach has been a strictly practical one using PDP-11 computers.

The early experiments used a network with a conventional message exchange system between processes in the network. Messages were "transmitted" very efficiently using memory sharing if the two processes happened to reside in the same computer. Otherwise messages were transmitted over a communications network.

The later systems have attempted to overcome the disadvantages of a network system by connecting the unibusses of the systems together using a "bus-window" system. This is in fact an address mapping bus-to-bus link. Attempts to access store in certain address ranges on one bus are serviced by the link and transferred to the other bus after performing an appropriate address transformation. Modifications have been made to the hardware of the PDP-11 to allow sharing of I/O "space" using the bus-window.

The PDP-11 is not designed as a multi-processor and as a result experiments to place 2 PDP-11's on the same bus have met with difficulty. In particular the arbitration is built into the PDP-11's processor hardware. A better design would put the arbitration in the bus together with an automatic queueing system of operations. The current "bus-window" system is seen as the midpoint between a communications link network and full sharing on a single bus. Eckhouse believes that this system could be very reliable and have high availability.

My visit interrupted Eckhouse's preparation of a paper for the forthcoming operating systems conference at Austin, Texas.

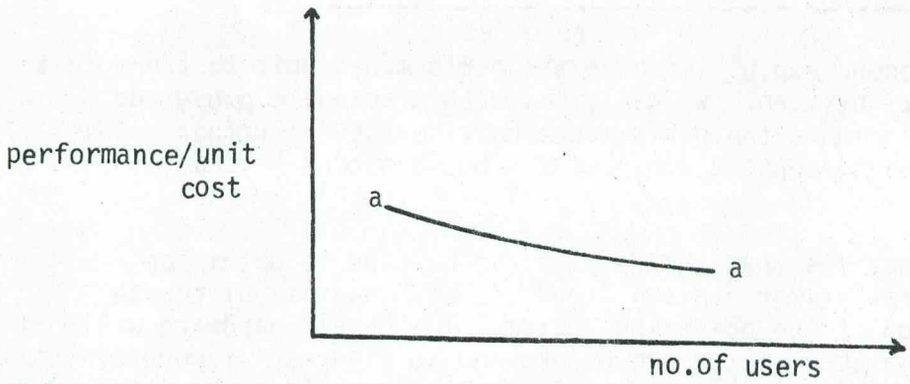
Looking to the future Eckhouse sees a more generalised form of architecture with the coexistence of various philosophies within the same system. University research could tackle the problem of determining the right structures to support such architecture. (Perhaps something better than the Von Newman principles should be developed).

An essential principle that should be applied in a networking environment is that the user should not know that there is a network there at all. We should concentrate at looking at the user's point of view. (So far research has concentrated on protocols and message switching strategies for optimum performance). The technical details of economic solutions to the user's computing problems should be totally transparent.

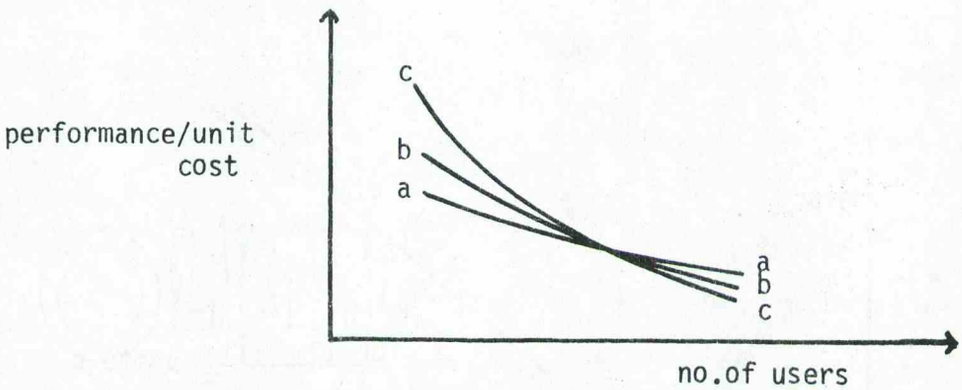
Eckhouse sounds two warning bells in discussing the economics of computing. One is that while the costs of memories and c.p.u.'s are being reduced at a rapid rate, communication costs are only decreasing slowly. In seeking economic solutions we should rely on communications less and less. The second warning is that Large Scale Integration does not reduce costs to near zero. We still need a lot of small scale and medium scale integrated circuitry to support it. By way of example, a typical PDP-11 computer needs five boards of logic, whereas the equivalent LSI/11 (with the processor and store implemented in LSI) still needs three boards to be viable. A reduction certainly, but the "computer on a chip" attitude is misleading.

Discussions with C. Gordon Bell

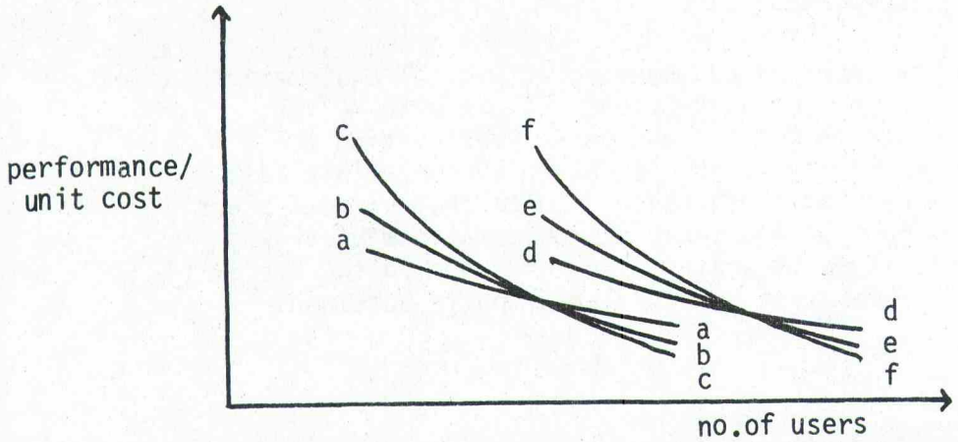
Bell is Vice President of Engineering at Digital Equipment Corporation and Professor of Computer Science at Carnegie Mellon University. Amongst other things, he has been investigating the balance of processors vs memory for DEC's PDP-10 system. This has taken the form of feed-back of performance results from various users with various configurations. To display his observations he draws a graph of "performance/unit cost" against "no. of users on the system". With a limited amount of memory a fairly flat graph is obtained:-



This may be interpreted as saying that the total usable computation delivered by the system decreases slightly as the number of users increases. This seems reasonable as the overhead involved in user administration will increase, but this is a relatively small amount (in a good system). The system, at least for a small number of users, is memory bound. If we add more memory (line bb and even more memory, line cc) then we obtain the following result:-



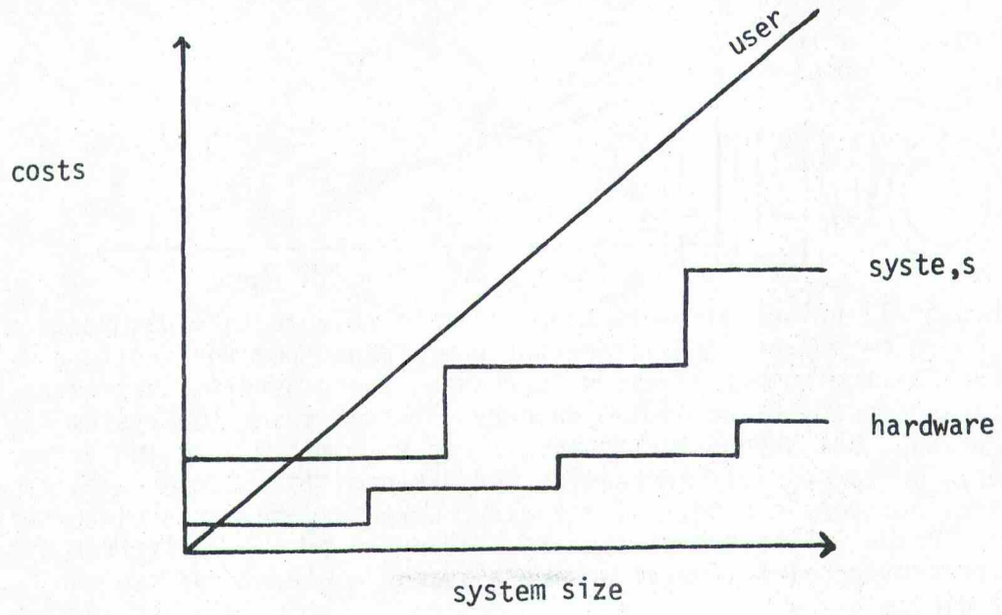
With a small number of users the system is able to take advantage of the extra memory and deliver better performance per unit cost. However as the number of users increases, these benefits decrease until a "crossover" point is reached. At that point the system is "balanced", but beyond that point is cpp.u. limited. Adding extra memory, while possibly increasing performance, is not seen as a cost-effective exercise - it actually decreases the performance/unit cost. Actually the system appears already to be c.p.u. limited at the crossover point, for if we put a second c.p.u. in we see the following:-



Introducing a second c.p.u. improves the performance/unit cost and this is plainly the right step. We can also, with a second c.p.u., add more memory to increase the performance/unit cost even further. This allows the system to support more users - but a second "crossover" is observed.

Bell hypothesises that the results can be used to obtain an optimal configuration for a given "load". The "load" is of course partly a function of the operating system. The PDP-10 hardware will only support a maximum of two processors and so figures for more are not available, but Bell believes that the pattern would be repeated.

The cost computation is a crucial factor in the above exercise. Initially only hardware was included. Secondly to this was added the central support costs (systems support programmers etc.). However Bell realised that a user with a poorly responding terminal will be incurring the cost of his time - even if the system is performing at 100% efficiency! So he included the user's costs. A graph of these three costs against system size look as follows:-



Thus in all but the smallest systems, the users costs far outweigh all other considerations. So slow response from a large time-sharing service is expensive - and economies of scale are probably false. As a corollary it is more economical to buy smaller "personal" systems cutting down on the user costs, wherever the application will allow.

In the general context of operating systems, Bell's opinion is that modern day systems are sufficiently modular to be able to take advantage of parallel processing capabilities. This applies to other categories of system software too, however the more general problem of how to derive parallel processing solutions in general applications areas has still to be solved. Bell looks to the Universities to tackle this problem area and develop suitable methodologies.

Discussions with P. Brinch Hansen

Brinch Hansen's main interests are in methodology, for large software systems and in the theory of abstract data types [12,13,14,15]. He uses access graphs as a formalism in the design process of a system. These do not imply any particular implementation, but form a basis for the process and monitor concepts of Concurrent Pascal [16]. Both of these concepts include access rights. Access rights do not imply data flows.

Brinch Hansen is engaged in designing and implementing operating systems as part of the Concurrent Pascal project. He hopes to continue this for the next 12-24 months and then to write a "Cook-book" for operating systems - as against a book full of "grandiose principles" [13]. He then wants to change fields - possibly into developing methodology for distributed systems of computers. As he sees it, the current problem is that so far we have built large systems on expensive computers with megabyte wide communications (using cross-bar core access systems or similar) in which the assumptions are that critical sections won't take much time. Hence we can apply well developed Dijkstra-based methodology. To come up to date we should use distributed cheap computer systems with byte-wide communication where "memory-sharing" does not exist. Equally suitable methodology does not exist! So this is the initial task - to work on methodology for distributed systems.

One reason for greater cost on shared processor systems (e.g. Hydra) is that the speed of the cross-bar switch is comparable to the memory speed - and hence there is a serious degradation especially when the load is light.

When programming in Concurrent Pascal, Brinch Hansen still pretends that he is using a system of classes in the way he used to in Simula!

Discussions with R.S. Fabry

Fabry's Interests are threefold:-

(i) Protection using capabilities (esp problems of error recovery).

(ii) Paging systems (developing better schemes than the working set model).

(iii) Scheduling problems.

Capabilities were originally developed as an addressing mechanism. Now they are seen as a protection mechanism [29]. Fabry is working (with Redell at M.I.T.) on the problem of being able to "recover" a capability after passing it on, especially in error situations. Several systems (e.g. Plessey's in the U.K.) have been developed using capabilities. However their use is still at system level, not at language level. Fabry is working to narrow the gap, i.e. so that facilities similar to those provided in languages can be provided in systems, and so that protection structures can be represented in suitable languages. An interesting problem is the confinement problem - preventing a called routine from passing on the capability to a third routine (see also Lampson [64]).

Fabry's view of protection at compile time is that it is valuable economically, and a useful aid to system development, provided that it does not restrict the run-time flexibility in any way (e.g. the dynamic creation of processes). It is not clear that this is so in e.g. Concurrent Pascal. A multi-process version of Pascal was developed at Berkeley for the purposes of teaching concurrent programming in an undergraduate O.S. course. This was done by a simple assembly code addition to provide a mechanism of duplicate stacks to support the various processes.

The Rocquencourt Operating Systems Conference, and Lampson's Open Operating System

The most recent operating systems conference was held at Rocquencourt in April 1974. The proceedings [46] contain some 16 papers including topics such as synchronisation, levels of abstraction, simulation, queueing theory and modelling, as well as studies of particular systems. Among the latter we note an interesting paper by Lampson on an "open" operating system for a single user machine [65] - based on OS6 [105]. This is seen as a step away from the "all or nothing" approach of the current day "closed" systems, giving the user the option to use a facility unaltered, to alter it to suit his desires, or to replace or ignore it altogether. Lampson claims that this type of flexibility can be provided in a multi-user system - without jeopardising robustness. A subsidiary aim of an operating systems project could well be to include this sort of flexibility.

II

Protection and Security

Much work is going on in this area, and is well documented. I include a brief paraphrase on Lampson's Classic paper, and also of Saltzer's survey. I do not consider this an "activity area" from my point of view, though it is clearly relevant. Nevertheless I describe two interesting activities that I came across. In addition I would point interested readers to the "Workshop on Protection in Operating Systems" held at IRIA, Rocquencourt in August 1974. Other relevant papers are [8, 28, 55, 82, 83, 89, 94, 109]. Many of these emphasise the "kernel" approach to operating system structure - where the small central core of the system provides secure mechanisms (such as the manipulation of non-forgeable capabilities). The Multics activity described here is a fairly "practical" project. Those who relish more formal approaches will find enjoyment within the references given above. In the context of a distributed system project, several people have pointed out that security is enhanced if the information is physically (and geographically) distributed.

Lampson on Protection

In his paper "Protection" [63], Butler Lampson considers a number of models of protection systems with the object of classifying the various ad hoc mechanisms that have evolved. The simplest and most general system contains a number of distinct processes which are able to communicate only through a message system. Lampson points out that each process corresponds to a separate protection domain. In particular the problem of "mutual suspicion" is easily catered for. The integrity of the protection model depends only on the "system" inserting the name of the calling process at the head of the message. The model bears a close resemblance to the system proposed by Brinch Hansen [12]. Lampson notes two flaws in the model:-

a) it is impossible to retain control over a runaway process, since there is no way to force a process to do anything. However such faulty processes will do no more harm than waste resources;

b) an elaborate system of conventions is required to get processes to cooperate. The first problem is that a process has to have the names of other cooperating processes built into it (in order to protect itself from processes it does not wish to talk to). The second is that cooperating processes must agree upon the interpretation of messages.

He also concludes that the model is too bare for convenient use. He develops a model based on an access matrix of domains and objects. In general an access matrix is large and sparse. A realistic method of overcoming this problem is to use capabilities. A capability is an "object/permitted access by a given domain" pair. A table of such capabilities is associated with each domain.

Saltzer's Survey on Information Protection

Saltzer has produced an excellent survey paper on "Information Protection" [92]. He categorises the ongoing research work into nine areas:-

System Penetration Exercises- directed at current (supposedly secure) systems;

User interface studies - to rationalise the often unacceptably complex protection environment;

Proofs of correctness of protection mechanisms - extending assertion-proving techniques to cover constructions encountered in operating systems;

Mathematical models of protection kernels - what it means to protect information. Such models might suggest assertions which would then be input to proofs of correctness;

Protection mechanisms - architectural structures for controlling access to change protection specifications, for user-defined protected subsystems, and capability systems. Security in data communication networks - user authentication, encryption, etc.;

Data base facilities - including protection mechanisms in data management systems;

Authentication mechanisms;

Department of Defense operational problems.

He then gives brief summaries of work being conducted (at the rate of 3 man years/year or more) at 20 research centers.

Saltzer's own work on Multics is described in [91].

Multics Certification

R.J. Feiertag is working on a project to produce a "certifiable" version of the Multics system [76]. The motivation comes from the Department of Defense which has some high security applications. The aim is to produce a system in which it is possible to have confidence that it is correct and secure. The Multics team are not trying to prove correctness formally.

The method of approach is to seek ways of restructuring Multics to make it simpler and smaller. The aim is to derive a kernel which is the only part that has to be examined for security constraints, and which is small enough to understand. Various mechanisms have already been removed from the supervisor, e.g. the dynamic linking mechanism, and the name management system. Currently the Multics team is working on removing the segment hierarchy (leaving a simple 1-level system).

The questions still being asked are "what should be left in the kernel?" and "how should it be structured?".

As implied above, the Multics team are interested in security, not just correctness. Firstly it is necessary to define security, and to define the items to be protected. The certification exercise seems to exist on three levels:-

1) to show that the defined operations work correctly (together with the correctness of the associated security policy);

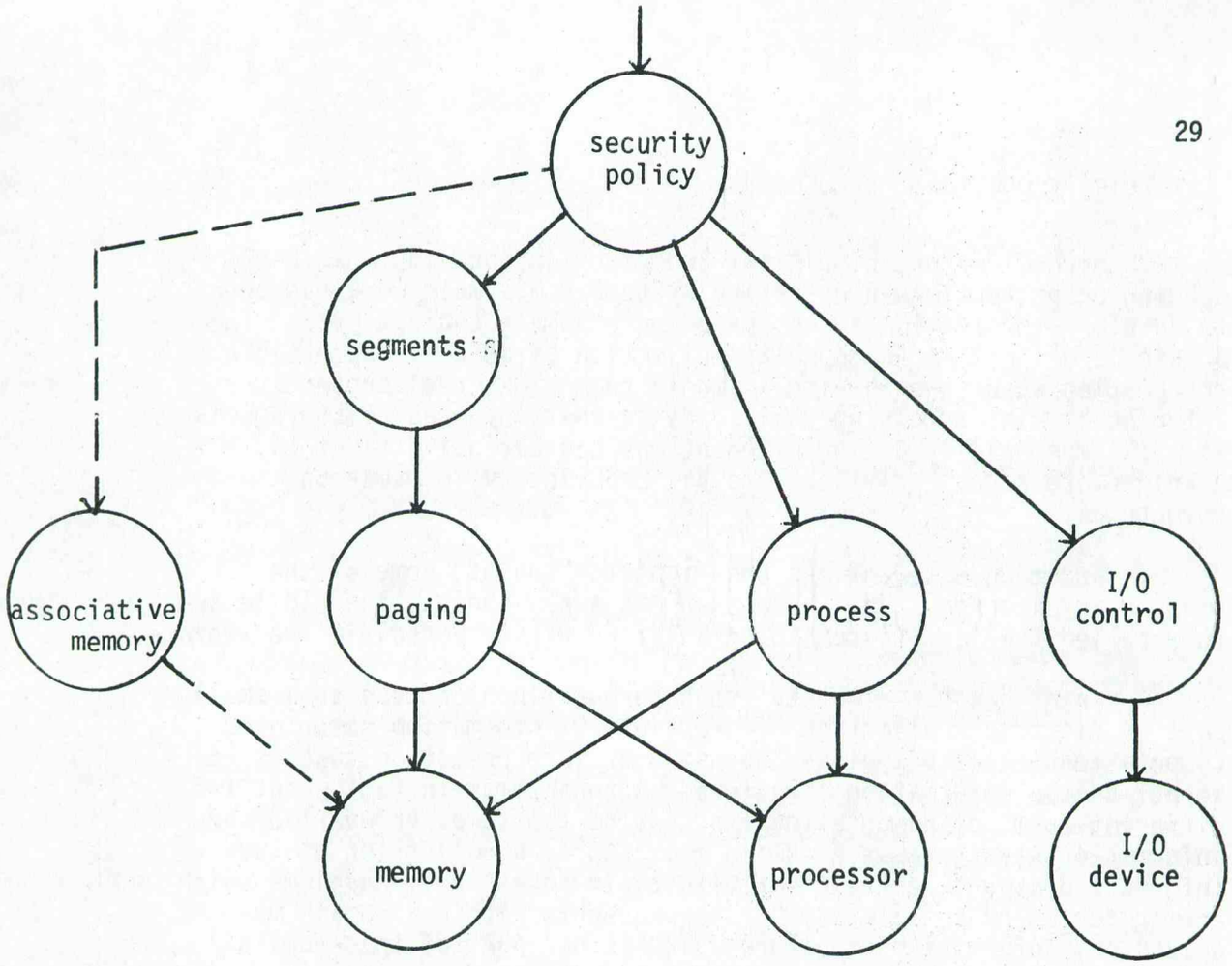
2) to show that services are always available and that there is a fair share of resources (this covers topics of policy, management and scheduling);

3) to show that the confinement problem is satisfactorily resolved.

Feiertag is concentrating on the first of these levels, which is probably the easiest. As yet there are insufficient concepts to enable (2) to be carried out, let alone (3).

As an aside we note that the two lower levels seem to reflect the principle of a policy/mechanism separation that we have observed elsewhere. Feiertag is interested in demonstrating that the mechanisms are correct.

To do this he has produced an abstraction of "regions", which contain all the primitive functions needed. These are structued as follows:-



Thus he observes that the mechanisms fall into a hierarchy. All accesses to any mechanism pass through a common security policy, and some mechanisms rely on others for their implementation. For reasons of efficiency, memory access is often short circuited through an associated memory, which has just enough information about the security policy, segments and paging mechanisms in order to function. This provides an ugly, but controllable complication.

Feiertag has been using CLU [68] as an implementation language. Each region above is implemented as a CLU cluster. While this has allowed him to produce an implementation in about 500 lines of code, there are some disadvantages. The language tends to hide some problems, for example it assumes the existence of stacks etc.. There is a problem of matching the facilities in the language with those available in the environment. Barbara Liskov is aware of these limitations, and feels slightly sad as otherwise operating systems are typical of the applications areas for CLU.

Spier's Domain

Michael Spier, of Digital Equipment Corporation, has been working on protection in operating systems. His main work has been to develop a protection system based on a domain concept [97]. The domain is defined as "a capsular collection of mutually accessible information structures having a single common external protective interface". The accessing philosophy is that any area of storage is at all times held by a single tenant who has exclusive right to grant/revoke access privileges to his proprietary information structures.

In addition a tenant has the right to "sublet" some of the proprietary storage. This area then has a new tenant, the old tenant (now called the lessor) retaining only the right to reclaim the storage.

A tenant has the right to grant computational access to a small part of his total collection of proprietary information structures. Computational access implies the usage or interpretation that is to be put on the information structure. A tenant may in fact grant two differing forms of computational access to the same, or overlapping information structures. A domain consists of a collection of such information structures together with their accessing mechanisms which are protected by a "fire wall". No procedure within a domain may access any information structure which is not part of that domain.

Information sharing is carried out by arranging that several different domains have computational access to a common information structure.

Tenants and domains are conceptually separate. However, by using a tenant name we can create a domain consisting of all the information structures pertaining to the tenant. This duality makes the access mechanisms much simpler. Note however that while tenants may be considered as domains, the reverse is not true.

Spier considers the hardware implications of the system. Some computational operations are readily checked by hardware, other more elaborate operations need software checking. A processor is restricted by hardware to the structures comprised in its current domain of execution. Calls across domains are handled by hardware operations and use a gate mechanism - a gate is an inter-domain entry point. Spier also sees the need for a temporary "argument domain" which is carried by a process on its inter-domain journey and which complements the access range of the current domain of execution.

To demonstrate the practicality of these ideas Spier has implemented an operating system with the kernel/domain architecture on a PDP-11/45 [100].

More recently Spier has developed a model [102] which can be used to guarantee the deterministic runtime behaviour of the realisation of a correct protection mechanism. The aim is to help explain the relation between abstract access control and protection mechanisms, and their execution environment.

Another of Spier's recent activities has been to devise a message exchange system for a real-time environment containing some ugly constraints and a variable number of processes [99].

Methodology

Taking a wide view of this title, I review an excellent low-brow book on software project management. Narrowing down, structured programming [23] is a byword and the contribution by Parnas seems to be reverberating through many halls. Closely related is the work on data abstraction, here I report on the work of Barbara Liskov and Stephen Gillies, which includes a survey of other people's work in the same area. Also related is the work of Per Brinch Hansen and Tony Hoare, embedded in Niclaus Wirth's language PASCAL [114]. The treatment of data seems to be one of the major advancing areas of the 1970's. Plainly I could have broadened out into the topic of data-bases etc. [24], but both my knowledge in this area is slim, and even a superficial investigation of current work in data-bases would require a program larger than this current research project. I complete this section with a brief look at two new methodologies as applied to operating sytems. For a long look at the way we tackle large software, see Spier [98]. He also has some comments on modularity [101]. Aspects of methodology also arise in the discussions in the Reliability section of this report. One common denominator to almost all contributions on methodology is the deprecation of the flowchart!

The Mythical Man-Month

At the Los Angeles Software Reliability conference, Fred Brooks introduced his book under this title [17]. The book takes the form of a collection of essays written as the result of an analysis of the "OS/360 experience" - Brooks was manager of the project during its critical design phase. "The book is a belated answer to Tom Watson's probing questions as to why programming is hard to manage".

Brooks vividly describes large scale system programming as a tar-pit, "and many great and powerful beasts have thrashed in it". He analyses the myth that men and months are interchangeable, and states that to add more men to an ailing project is like dousing a fire with gasoline. Brooks' law states: "Adding manpower to a late software project makes it later".

He discusses the ideal of having a system designed by as few minds as possible, and quotes Harlan Mills' idea of organising the system's team like a surgical team: "...instead of each member cutting away on the problem, one does the cutting (the surgeon/chief programmer) and the others give him every support...". Brooks contends that conceptual integrity is *the* most important consideration in system design. In this context he draws the distinction between architecture and implementation.

Brooks then discusses the problems of system description and communication within the system team, and makes some valuable recommendations. The remainder of the book contains many valuable quotations of other people's experience and illustrations of problems that can arise. He gives many recommendations as to techniques that should be employed to overcome the problems of managing a systems software team.

The book is eminently readable and is to be recommended.

Parnas' Technique of Modular Decomposition

One of the most significant papers on programming methodology, which is directly relevant to operating systems, has been written by D.L. Parnas [80]. This is on the topic of modular decomposition. Parnas points out that the expected benefits of modular programming are (1) managerial - separate groups can work largely independently on separate modules; (2) flexibility - changes can be made to an individual module without changing other modules; (3) comprehensibility - modules can be studied independently, and better understanding should mean better design.

Parnas asserts that it is a serious mistake to decompose a system into modules on the basis of a flowchart. This will inevitably result in various modules sharing elaborate data-structures and instantly counteract all the expected benefits.

Instead Parnas proposes the concept of "information hiding". Each module is characterised by its knowledge of a design decision which it hides from all other modules. Its definition and interface are chosen to reveal as little as possible about its inner workings. Modules no longer correspond to steps in the processing.

One begins with a list of difficult design decisions, or design decisions that are likely to change. Each module is then designed to hide such a decision from the other modules in the system. Parnas gives the following more specific examples of criteria to be used in decomposing a system into modules:-

(1) A data structure and its internal linkings, accessing procedures and modifying procedures are all part of a single module.

(2) The sequence of instructions necessary to call a given routine and the routine itself are part of the same module (applies particularly to assembly code).

(3) The formats of control blocks used in queues in operating systems and similar programs must be hidden within a "control block module".

(4) Character codes, alphabetic orderings and similar data should be hidden in a module.

(5) The sequence in which certain items will be processed should (as far as practical) be hidden within a single module.

With this approach Parnas points out that module interfaces are more abstract then in the conventional "flow-chart" approach. They consist primarily of function names and the numbers and types of parameters. This means that independent development of modules can begin much earlier than would be possible otherwise.

Parnas is well aware that this approach may engender implementation inefficiencies if care is not taken. It is likely that the assumption that a module is one or more subroutines will have to be abandoned. Instead subroutines and programs are assembled collections of code from various modules.

Parnas compares this form of modular decomposition with the hierarchical structuring techniques advocated by Dijkstra and others, and concludes that both are valuable methodologies but are, in fact, independent. In particular it is possible to modularise in the old-fashioned way and produce a hierarchical structure, but still retain the flaw that important design decisions are represented in the module interfaces.

A final significant advantage of Parnas' technique, mentioned almost in passing in the paper, is that it can result in considerable carryover of work from one project into another.

Liskov and Gillies on Data Abstraction Specification Techniques

An area of methodology which shows promise of aiding the design and construction of large scale software systems is that of formal data abstraction techniques. Barbara Liskov and Stephen Gillies presented a survey paper at the Software Reliability Conference [69] and we summarise the main contents here.

Data abstractions have been used in programming languages for a comparatively long time. Until recently programming languages provide a fixed set of data abstractions (e.g. integers, arrays). If the programmer wanted to use other data-types (e.g. lists), then he had to map them onto those provided in the implementation language. More recently programming languages have been devised which allow, to a greater or lesser extent, the user to define his own data-types and the operations that can be performed on them. This development has led to the study of data abstractions and of formalisms for defining them and the associated operations. With suitable formalisms the task of correctness proofs of programs can be tackled.

A data abstraction is defined to be a group of related functions or operations that act upon a particular class of objects, with the constraint that the behaviour of the objects can be observed only by applications of the operations. We see that this concept is strongly related to that of "information hiding" of Parnas [80]. The implementation of a data abstraction is provided by a multi-procedure module.

The paper by Liskov and Gillies is concerned with the various formal specification techniques that have been developed, with particular (though not exclusive) reference to data abstractions. The advantages of formal specifications over informal specifications (or no specifications at all) are given as :

(1) a proof process is available, and moreover computers can be used to aid the proof process, e.g. by checking the steps of a program proof, or by generating test cases to "cover" all possibilities;

(2) communication of the concepts is improved - as there is less room for "intuitive" comprehension (which can lead to errors). Ambiguities are less likely, and are more obvious;

(3) a concise and well-understood design language is available as a byproduct.

A wide spectrum of specification methods are under development. The major distinction between the various methods is the approach to defining the semantics of the data abstraction. Syntax is a lesser, and better understood problem. The specification techniques fall into two categories, those that specify semantics by forming a model of the abstraction, and those that define the objects of the abstraction implicitly using descriptions of the operations.

In the first category we find systems that use a model of predefined objects (e.g. sets or graphs) and those which use models of arbitrary, but well understood, objects. Systems in this category tend to suggest an implementation for the abstraction. This is an advantage if the particular abstraction on hand fits well, as specifications will be fairly easy to construct and comprehend, and reasonably minimal. However, we have the obvious limitation that the range of application is relatively narrow.

In the second category we find systems that are inherently free of implementation details or bias. A technique developed by Parnas divides the operations into V (value returning) and O (operate to produce a change in state). The specifications take the form of indicating the effect of each O operation on the result of all the V operations. One problem is that in practice the O operations are not independent, and that the effect of an O operation may well depend upon the history of the previously applied O operations. The separation of O and V also has the effects of making the specifications difficult to read and fussy to change (a new V operation may mean tampering with many O operations).

Data descriptions may be determined by an axiomatic approach - expressing the properties possessed by the objects and operations upon them as axioms. This approach certainly is compact and widely applicable but somewhat lacks comprehensibility and constructability. It is not always obvious that the specifications correspond to the concepts involved. An improvement is to be found in the similar algebraic definition technique. This uses the "presentation" technique and as a result reduces the number of axioms required.

Liskov and Gillies discuss these techniques in greater detail and conclude that all the current techniques have deficiencies and that they may have to be used in combination on a large programming project. In addition the techniques apply to the individual modules of the system (where in fact each module corresponds to a data abstraction). The techniques are not particularly suitable for dealing with a system as a whole. They conclude by indicating areas of research which need to be pursued to develop specification techniques as part of programming methodology, particularly with respect to the use of abstract data-types. The paper includes a bibliography referencing the techniques mentioned here.

CLU

Barbara Liskov has developed a language called CLU [68]. The objectives are to bring together current developments in methodology, in particularly modularity and structured programming. The latter is essentially the recognition of abstractions. Two kinds of abstraction may be represented in programs - abstract operations and abstract data-types. The former are naturally represented by procedures. The criteria developed by Liskov for the handling of the latter are:

1. A data-type definition must include definitions of all operations applicable to objects of that type.

2. A user of an abstract data-type need not know how objects of the type are represented in storage.

3. A user of an abstract data-type may manipulate the objects only through the type's operations.

CLU is described as a structured programming language. It is modular - with two kinds of modules: procedures (supporting abstract operations) and clusters (supporting abstract data-types). The CLU system maintains a database containing a description unit for each module. This gives enough information about a module to enable compilation of other modules which use the abstractions implemented by the module. In fact each new module submitted to the system is accompanied by an association list identifying the particular description units to be used for the various abstractions within the module. The compiler uses the association lists and description units to type-check all module interfaces completely at compile time.

Type conversion may occur across a module interface. An abstract data-type will be implemented by a set of operations defined within a module. Part of the definition describes the representation of the abstract data-type, in terms of other already understood data-types. Within the module it is the representation that matters, whereas outside the module the representation is unknown and irrelevant. So CLU provides features to turn an abstract data-type into its representation on entry to a module, and the reverse on exit.

CLU makes a clear distinction between variables and objects. The latter exist independently of individual programs in a CLU universe (like an Algol 68 heap). CLU variables exist only in programs and form a convenient way for programs to reference objects. There are two methods of parameter passing in CLU. The first, call by copy, is to make a copy of each of the objects referred to by the actual parameters, and to make the formal parameters refer to the copies. This mechanism is also used for the conventional assignment operation. The second mechanism, call by sharing, sets the formal parameters to refer to the same objects as the actual parameters. This allows the only permitted side-effect in CLU, and is the principle mechanism used by the routines within a module implementing a data abstraction.

CLU also supports a rather more general form of abstraction mechanism, called a type-generator. This allows definitions of whole classes of types. (An example is the array, defining a structuring and access mechanism that is independent of the type of object in the array).

A CLU compiler (written naturally in CLU) has been implemented at MIT and the language is being used by several groups within project MAC.

Barbara Liskov's earlier work on the Venus Operating System [67] is also worthy of note as a practical system organised along Dijkstra's hierarchical principles.

Per Brinch Hansen on Concurrent Pascal

The first objective of Concurrent Pascal [16] (or concurrent Simula!) is to do for operating systems design what Pascal [14] has done for high level sequential programming. There is a very strong reduction in economic costs associated with a language like Pascal (let alone advantages of elegance, simplicity etc.). The second objective is to do it before talking about it! Since the conception of the ideas of Concurrent Pascal (esp monitors), three years have been spent in writing three compilers and the kernel of an operating system.

The language concerns itself with access rights. A sequential program may be associated with a data structure in such a way as to be the only program that can access the data structure. A process may access the data structure by calling this program. To make access rights completely explicit each process has a list of access rights which lists all the components that can be accessed by the given process. A data structure may be shared by two processes by giving both processes the access rights to call the program associated with the data structure. The program is called using send and receive messages. Integrity is ensured by the use of the language structure called a monitor which allows compile-time verification that the data structure is accessed only through proper calls of the associated procedure.

It is desirable to be able to decompose a monitor into other monitors, or perhaps call other processes. In a manner analagous to the above, Concurrent Pascal allows access rights to processes and monitors to be associated with a monitor. This gives an explicit mechanism for describing a hierarchy of access rights.

Monitors and Processes are rather similar objects, both have access rights, and private data structures. The difference between them is the way they are scheduled for execution. A process goes on for ever in Concurrent Pascal, whereas a monitor is a collection of procedures waiting to be called by processes. It is perhaps more illuminating to view them as new kinds of data-types - for they define data structures and the legal operations that can be performed on them. This is suitable for compile-time checking in the traditional manner of type-checking.

By defining processes and monitors as data-types we can allocate different instances of the same kind of data-type, thus creating many similar processes, for example, which are algorithmic copies of one another.

Concurrent Pascal allows one to decompose an operating system into a hierarchy of processes and monitors - which represent the "virtual" resources available at the various levels. This is "programming in the large". The word hierarchy is perhaps not completely suitable, as the access rules of a large program may well turn out not to be tree-structured, but be graph-like. This is partially the explanation why block structured languages have not been very convenient for large programming tasks. The access structures do not conform. Of course a program is a linear structure, but there just is no convenient way to place the partial ordering on a graph necessary to obtain such a linear structure.

The algorithms of the operating system will be published in the IEEE Transactions on Software Engineering in July. The single user operating system consists of some 24 modules (on 20 sheets of line-printer paper) - (one may define a structured program as a collection of programs each less than one sheet of lineprinter paper long). Two errors were found during the debugging phase, and since then no problems have arisen.

Problems with Concurrent Pascal:

1) No provision for dealing with interrupts;
2) No built-in facility for talking about store management and demand paging;
3) No provision for getting rid of processes - there is a fixed configuration of monitors and processes;
4) The operating system assumes that all user programs will be compiled by a correct Pascal Compiler!

In effect Concurrent Pascal is a programming language for writing core-resident real-time operating systems for mini-computers. It is particularly suitable for "one-off" operating systems and for teaching purposes.

Hoare on Monitors

C.A.R. Hoare has developed Brinch-Hansen's concept of a monitor as a method of structuring an operating system [52]. The overall aim is to provide a high-level programming structure that provides all the facilities necessary to construct resource schedules. Such a scheduler is written as a "monitor".

A monitor contains local administrative data together with procedures and functions which are called by programs wishing to acquire and release resources. The data variables of a monitor cannot be accessed by outside procedures (other than by calling a procedure of the monitor). As a simple extension it is possible to declare a class of monitors, all with the same algorithms, but with different identifiers and their own separate instances of their local variables.

It is assumed that the monitors will exist within a parallel processing environment. In general any procedure of a monitor may be called by any process. However monitors have the crucial property (guaranteed by the "system") that only one process at a time can be executing the procedures of a given monitor. This enables a degree of integrity to be ensured with respect to the monitor's local variables.

There are many circumstances when a resource cannot be allocated immediately and the requesting process has to wait. An appropriate data structure (called somewhat misleadingly a "condition variable") is provided. This represents a (possibly empty) set of processes which are waiting for a "condition" (e.g. the resource being available). This data structure is local to the monitor. The monitor can apply the operation "wait" to this data structure. This results in (i) the calling process being placed in a "suspended" state and being added to the data structure, and (ii) the monitor being made available for entry by other processes. The inverse operation is "signal", also applied to the data structure. If the data structure is empty then this has no effect. Otherwise one of the processes is extracted from the data structure and continues execution in the monitor at the instruction following the "wait" that caused it to be placed in the data structure. Only when this "freed" process exits from the monitor (or calls another "wait") will the process calling "signal" be allowed to proceed.

An important aspect of Hoare's work is that he gives a scheme for proving that a particular scheduling algorithm treats its local data properly, despite the synchronisation and associated mid-procedure process changing described above. It is possible to verify an absence of the potential interference effects that frequently occur when a data structure is shared between processes. However the dreaded "deadly-embrace" has to be taken care of by more pragmatic methods.

Hoare concludes his paper with a discussion on the merits of disjoint resource schedulers, and suggests the following principles to avoid some of the foreseen problems:-

1. Never seek to make an optimal decision; simply avoid persistently pessimismal decisions.

2. Do not seek to create a virtual environment which is better than the hardware capacity.

3. Use preemptive techniques where possible.

4. Use "grain of time" methods (see Dijkstra).

5. Keep a low mean and variance on waiting times.

6. Avoid fixed priorities and indefinite overtaking.

7. Avoid effects such as thrashing in overload conditions.

8. Make rules for correct and sensible use of monitor calls. Necessary checking should be performed by an algorithm (particularly in hardware) local to each process - a "user envelope".

Systems Design and Documentation using Path Descriptions

Whilst attending the Software Reliability Conference in Los Angeles, I was introduced by Per Brinch Hansen to Alan Shaw of the University of Washington. Shaw has recently produced a textbook on operating systems [95].

Shaw has developed a technique, not unlike Brinch Hansen's precedence graphs [13], for describing sequential and concurrent systems. We quote from the abstract of a preliminary paper [96]:

"Systems are described in a top-down fashion by sets of rules or productions giving the possible execution-time control paths through the various components. Sequential and concurrent paths are expressed in an extended regular expression language which permits the specification of both a fixed and a variable degree of concurrency, critical sections and cyclic activities".

The notation extends the language of regular expressions with a simple concurrency operator, and an operator denoting a variable degree of concurrency. A third operator denotes cyclic processes. The preliminary paper gives a description of a simple spooled multi-programming system in this extended regular expression language, together with informal pictorial representations of each statement as a graph.

Bredt's Syntax-directed Operating System Design

Thomas Bredt, of Stanford University Digital Systems Lab, has developed some ideas on developing appropriate methodologies for the specification of the messages that processes in an operating system send to each other [9].

We have noted a trend towards structuring a system along the lines of a set of processes communicating using a message system and away from memory sharing. Bredt has developed a syntax-directed methodology for specifying the formats that any particular message may take. A process may thus be thought of as a high-level language interpreter - accepting messages in its own little high-level language.

With this system operating system design (or architecture) takes the form of determining the particular processes and designing their external interfaces, in terms of a high-level language grammar. The functionality of each process can perhaps be linked to the syntax specification but otherwise, in the current state of the art, is informal. The implementation of the architecture is thus a matter of writing the appropriate high-level interpreters.

The main advantage of this approach is that it brings the well understood techniques associated with high-level languages to bear upon the problems of operating system design. It simply makes the design job easier if the message formats associated with a process are specified formally, a suitable language environment can place compile-time checks on all generated messages that they are in the correct format, and the command or message interpreters are straightforward to implement.

Bredt illustrates his ideas with the description of a simple batch processing spooling system using syntax diagrams to describe the command (message) format of each process, and indirectly the underlying semantics.

Reliability

As part of the research program, I attended the International Conference on Reliable Software in Los Angeles in April. The major recurring theme of the conference is that reliability extends far beyond correctness. I present notes here on four of the sessions at the conference (other references to the conference are made elsewhere in this report). Wulf's tutorial gives a good impression of the problems of reliability. The panel discussion and Vyssotski's presentation are both valuable, interesting, and neither are formally available in printed form in the proceedings. Randell's contribution is included because (a) in my prejudiced opinion it was one of the best and best received at the conference; (b) I learned more about work on my own back doorstep 5000 miles away, and I need a reminder to my conscience.

Wulf on Reliability

Wulf gave the following tutorial paper [119] at the Software Reliability conference:-

Reliability is a probalistic notion (observed by the end user), it is the probability that the system will be functioning correctly after various periods of time. It does not a priori relate to what the cause of a failure may be. Even a perfectly correct program may be unreliable. There is a tendency to treat reliability and correctness as equal - this assumes (falsely) that software can be no more reliable than the underlying hardware. This also presumes that a hardware malfunction means simply that a restart is necessary. In reality the damage can be spread much wider.

The criteria for what constitutes reliability can vary widely from application to application. Reliability is not an absolute concept. A "soft" definition of reliability is adequate in many circumstances - where minor malfunctions can be tolerated and absorbed, so long as there is not a total breakdown. In computer hardware, the notion of coverage is much more important than component reliability. Coverage is the probability that, given a failure does occur, you recover from it. Increasing coverage by relatively small amounts increases the mission time of the system significantly more than by increasing component reliability by large amounts.

We hence concentrate on this notion of coverage. In building the Hydra multiprocessor system it would be nice to claim an advantage of reliability over single processor systems. However one malfunctioning processor can generate bad information which other processors subsequently use (believing it to be correct).

The approach to increasing coverage depends in a critical way on the construction methodology used in the system. The technique used in Hydra is similar to Parnas' Modular decomposition technique, and also there exists a strong parallelism with Barbara Liskov's Data Abstraction techniques. All information about the representation of some abstraction is localised to the code of the module which realises the abstraction. The module defines a set of actions that can be performed on the abstract idea. The only way of performing operations on the abstract data type is by invoking the functions of the module. Modules hide information about the representation of the abstract data type. In Hydra we require that every module be responsible for the reliability of the appropriate abstract data type representation.

The same criteria for achieving reliability are applied to hardware modules as to software modules.

An important property for a module is that the information it possesses is bounded both from above and from below, i.e. module A may use module B. Module A knows how it uses the data abstraction presented by module B, even though it is ignorant of the internal representation of this data abstraction. Its knowledge is bounded from below. Similarly B's information is bounded from above. B does not know anything about the "higher" abstractions implemented by other modules using its facilities.

The reliability responsibilities of each module in Hydra are

(i) to detect (almost) any error (within some limit of the cost involved). This does imply some form of redundancy;

(ii) to limit the damage caused from any error that might occur;

(iii) to attempt to make the error transparent to modules "above" it;

(iv) otherwise to place the data structure into a consistent state and to give up control.

Reliability involves cost, at all stages including the design stage.

Absolute reliability is impossible - all we can hope to do is to decrease the cost of failure.

In response to a question, Wulf said that one of the problems is that there are a number of techniques available, or being investigated, but no adequate formal scheme for studying the problem. The available techniques are more a part of computing folklore than the science.

Software Reliability Conference - Panel Discussion Notes

"How do you get people to produce Reliable Software"

Members:

Dr. A. Ershov	Academy of Science, U.S.S.R.
Capt. Grace Hopper	U.S. Navy, U.S.A.
Dr. H. Mills	IBM-FSD, U.S.A.
Prof. B. Randell	University of Newcastle, U.K.
Dr. W. Royce	Space Applications Corp. U.S.A.
Prof. W.W. Turski	Poland
Prof. G. Weinberg	SUNY - Binghampton, U.S.A.

Ershov: The capabilities needed (in extreme proportions) for a software programmer are:-

Insight;

Logic & combinatoric abilities;

Patience;

Discipline;

An ability to preplan his own activity.

The problem lies in combining these abilities in a balanced manner. This usually takes about 8 years after leaving university (c.f. Fred Brooks' comment that it is only in his third system that a programmer shows his true ability) - too long. The way to reduce this period is to combine teaching & training, style & essence (as is carried out in medical and musical shools). This involves a 7 year training period:

3 years - full course of fundamental maths, and the use of abstractions, informal computer science courses (with intensive practice);

2 years - a "marriage" of maths and programming, mathematical courses in theory of computation, software engineering. A series of 2 person mini-projects;

2 years - participation in a real project, but under careful leadership of a teacher.

Such frameworks hardly exist at the moment. However, there is no other way to reduce the length of time it takes for a systems programmer to reach the stage of maturity where he can produce reliable software. It involves combining hard core mathematics, product oriented engineering with a professional perfection.

Grace Hopper:

In the context of reliable software, her concern is with people - not only the systems analysts and programmers, but those who define the problem and those who receive the end result. If you cannot define the problem then you cannot write reliable software. Many of today's difficulties arise out of poor communications. We often forget that our software interfaces with people. We often fail to treat programmers as people - they are not told why they are doing something. This allows them to make sensible secondary decisions. There are also major problems in managing people. Any company involved in producing software should have a dual ladder of promotion so that technical people can rise as high as professional people - but without being overwhelmed with administrative responsibilities. We have to satisfy two needs of programmers:- (a) to be part of an active group; (b) to attain individual accomplishment. This is possible if all members of a group take part in the initial design phase. Furthermore, a natural leader of the group will appear. Programming tools should be well standardised and well validated - so that they can be relied upon. We don't use the computer to help our programmers as much as we could.

Software does have to interface to people, who are unpredictable. It is thus very important to provide for the correction and maintenance of software in the area of data processing.

Mills:

The computer explosion over the last 25 years has occurred because there is great human need and great social value in what the software industry is providing. This social value will force the development of reliable software: the world cannot afford not to have reliable software. Human beings have a number of latent capabilities that are not discovered until technology uncovers them. We learn totally new capabilities over a period of time as a society. Programmers will come to write error free code in the future simply because everyone else is doing it. We underestimate the size and complexities of the software problems we face, and need in-depth education and to take software development seriously. So programmers as a society will come to write better more reliable code, and this is already happening.

Randell:

The approach of an individual can be characterised by whether he believes that Murphy's law will be repealed. A priori techniques for convincing oneself in correctness do not cover all of the problem. When we automate a task we reduce the error rate and increase the size of the disasterous-type errors. We need to admit that things will go wrong, and to provide some means of coping with these. One great battle is against complexity. We must add extra facilities, but not increase complexity. We should inculcate into programmers a disgust for complexity and a pride in simplicity.

Royce:

The most difficult problem in engineering today is getting mistakes out of software. The magnitude of this difficulty must be appreciated before the problems can be solved. Three root causes are identified:

1) we have to allocate enough resources to getting mistakes out of software; this means convincing customers;

2) programmers have weaknesses and frailties; this has to be faced, we have to employ the right kind of programmers;

3) we need a good requirements analysis - we tend to jump right into a problem.

Turski:

"I do not understand what this panel is about". Reliable software does not exist. Reliability rests not in the thing itself but its mode of operation and the thing itself. Correctness is a branch of mathematics, itself an art, and thus should be taught with poetry and music as ingredients necessary to the intellectual development of information processing. We are not in a very young profession, we are in the oldest profession. Information processing was the thing that made us into human beings. The real problem is how to make automated information processing reliable. One way of getting someone to produce a reliable anything is to get him to use the thing. The software producer has to place himself in the customer's shoes to think in the terms that the customer is using.; Otherwise there is no way to gauge the infinite complexity of the poor customer's problems. It is no use us telling the customer how difficult it is to write software - he won't understand. If he did he would not come to us. Our software has to work well in his environment - which is beyond our power of definition. Information processing is part of the natural world in which definitions play no significant rôle.

Weinberg:

If you communicate what it is you want then the programmer can succeed in producing the desired program. Much of the failure of reliability comes down to an initial failure of communication. Knowing why you are doing something is much more important than knowing how to do it. Henry Ford's Fundamental Feedback principle as applied to the prevention of river pollution: "a factory's input must be downstream of their output". This is an example of communicating what is desired, without implying how to achieve it. Actions speak louder than words. A project manager must take action to produce the right environment for the programmer. He has to decide what to measure, and this will affect how people work. (A sure sign of impending doom is when the manager decides to

measure the number of lines of code per day). Programming is an engineering discipline, and as such is the art of making compromises between conflicting goals. The secret of successful engineering is to get the customer to relax his specifications. A big step towards obtaining reliable software is to find out what the customer really wants. (This may involve delivering something which is not what he wants). A danger is to overdraw the initial specification.

We have to be careful to avoid "ritualism" in programming. It is important to understand fully why certain techniques are used. Otherwise they will be used in a self defeating manner. It's no use telling a programmer to use structured programming if he does not understand why (or understand the purpose of structured programming).

Vyssotski - Recent Experience at Bell Labs.

Software reliability extends beyond correctness. This talk is concerned primarily with software for telephone switching systems (e.g. ESS), but the principles extend to many other types of software. There are dedicated real-time systems (24 hrs/day 365 days/year for 40 years). Software reliability is not well defined, but systems unavailability is! Average system downtime on ESS is 15 minutes per year for all reasons - hardware, software, power and operational failures (over 500 systems). This contrasts with up to 200 hrs per year for a conventional commercial machine. The main reason for this contrast is that ESS has had high system availability as a major design objective from the outset.

It seems common sense, but it is often ignored, that the way to achieve high availability is to have a high availability objective, against which results may be measured. Here is a chronicle of problems, taken from a wide variety of systems, that prevented an availability objective from being met:-

1) On successive reads of a block on disc, the transfer failed, the system had no provision for continuing in this circumstance.

2) Secondary pointers on disc were mysteriously corrupted. It took 48 hours to rebuild the database as no utility software could deal with the situation.

3) An operator ignored repeated requests for an alternative output device after the main one had failed. The output queue filled up and the system stopped.

4) A manual restart procedure was confusing. An operator loaded an incorrect tape which was accepted through inadequate checks and a database was destroyed.

5) A c.p.u. error caused a wrong arithmetic result. The result was not checked for reasonableness.

6) A software patch for a previous problem was incorrect. When inserted it brought the system down and restart took an hour.

7) Power down detection software was inadequate to tell which unit was faulty.

8) A switch on a circuit card was incorrectly set by the maintenance people - this was not detected by the test software, but it brought the system down.

9) After a memory failure a spare module was tried. It was bad, but the software had no facilities for testing spare unused memory modules.

10) A recovery procedure had no facility for advancing past a bad spate of records on a backup tape. The entire group of transactions had to be inserted (slowly & painfully) by hand.

There is a natural human tendency to fail to plan for problems of hardware failure. Budgets must be allocated to this sort of problem area, and it must be someone's responsibility to think about it, even though there are no neat solutions. Someone must have the responsibility of making sure that the system attains the high availability objectives. When you start recording all outages for any reason it becomes apparent that it is possible to design software to give high availability, even on ailing hardware. The software has to continually evolve to react more rationally or more efficiently to problems that one would not ordinarily think of as software problems.

Part of the problem of reliability is that the market-place does not care enough to measure it, and demand it. The hardware manufacturers simply supply what is demanded. High reliability (hardware and software) costs money. At the moment it seems that customers are just not willing to spend the money.

An approach particularly well suited to some of the big software packages is to start with the catastrophe that you wish to avoid, and then to analyse the design for all the ways this could happen. If some of the ways have an unpleasant high probability associated with them, then you change the design.

System Structure for Software Fault-Tolerance

One of the more interesting presentations at the Software Reliability conference in Los Angeles was that by Brian Randell of Newcastle University describing a software fault-tolerance discipline [85].

Randell points out that to provide fault-tolerance, it is necessary to provide redundancy, and this has to take the form of alternative designs etc.. The method is based on the "recovery block" which incorporates a general method of calling any alternative procedures etc. when the main-line algorithm fails, and a general error detection/spare component structuring scheme. Normally the alternate procedure will try to have the same effect as the main algorithm, however it may only be able to offer an acceptable, but less desirable effect, which will also be associated with a warning that this has had to be done.

One problem that arises is that any damage done by attempting the main algorithm has to be repaired. Randell has attempted to do this by a mechanism involving resetting the system state using a recursive cache. This stores the old versions of global variables as modified by the main algorithm as and when these modifications occur. Various implementations of such a recursive cache (which is partly a hardware mechanism) are under investigation.

This only solves part of the problem. The algorithm will be part of a process and may well have interacted with other processes. The effect of these communications also may have to be "undone". This problem has been tackled by devising a scheme of conversations and placing constraints on the relationship of recovery blocks and conversations. Randell concludes that "if it is considered important that a complex system be provided with extensive error recovery facilities whose dependability can be the subject of plausible a priori arguments, then the system structure will have to conform to comparatively restrictive rules".

Networks

The topic of computer networks (and I discover computer-communication networks) is currently very popular. It is also plainly very relevant to today's computing needs and tomorrow's computing systems, yet I found myself largely ignorant of issues, techniques, and possibilities. A large part of this section is thus given over to summarising my recently acquired knowledge in this area. For local systems a ring topology seems to be popular, and Fraser's own contributions, and survey, are valuable. Finally I look at a network of a slightly different sort, which is really a new form of computer hardware architecture.

A surprise to me was to discover that the main motivation for networking is "data based", not the sharing of processing resources. Much of current database research should be considered in the light of networking requirements. We note some work in this field in passing in the next section. The light treatment of this topic here has already been mentioned, nevertheless the importance of developments in database technology cannot be overemphasised.

Networks - Principles and Practice

The idea of connecting computers together has existed for some time. A number of computer networks have been built for specific purposes (e.g. airline reservations, defence purposes) and have proved their operational value. By contrast, recent developments have been concentrating on the development of "public" or resource sharing networks, and on the development of appropriate communications techniques.

The value of computer networks

"In a society which is increasingly dependent on computers for its business functions, and in which information is becoming recognised as its most valuable resource, the importance of information distribution via data networks is becoming paramount". Ruth Davis in her keynote speech to COMPCON 73 chose "Computing Networks : a powerful national force" as her title [25]. In this address she asserts the following "truths":-

(1) Computer Networks are essential for all those real time geographically dispersed control activities vital to our individual and national well-being.

(2) Computer Networks are the only practical means available for the sharing of expensive information resources, computing resources, and information handling equipment.

(3) Computer Networks are the only practical means of providing equality of access to, and an equality of quality in, public services independent of geographic location.

(4) Minicomputers are becoming personal computing resources and minicomputers linked to computer networks provide each person with his own individual information center.

(5) Maxicomputers available through computer networks are perhaps the only economically justifiable means for the large scientific calculations essential to the advancement of much needed basic research and engineering.

(6) Manufacturing without minicomputers in the production process will be unheard of.

(7) Centralised management, in a real-time sense, of geographically dispensed organisations is impossible without computer networks.

These may be seen as "external" advantages of computer networks, i.e. the relationship of the network with its environment. Stelmach [103] points out some advantages of networks as an information processing architecture: by segmenting an application, networks of small computers are more economic, and can provide more processing power than could possibly be built into one large central computer; a network is more flexible, it can expand easily; minor alterations can be handled conveniently; software development is simplified; fault tolerance is improved (a system can run in a slightly degraded mode without one faulty computer).

Farber [36] points out advantages of security if a data-base is distributed across a network. From the military aspect even the physical destruction of the data-base is difficult(!) and "penetration" of the system has to be performed repeatedly at many distinct locations. In fact the PRIME system at U.C. Berkeley was developed with the specific aim of investigating such security problems.

For British readers, the concept of anything "Value Added" is viewed with suspicion! This is the term applied to a network (a V.A.N.) which uses the common carriers (telephone lines etc.) together with additional equipment to provide a general "public" networking service. The additional equipment deals with message routing, terminal interfacing, computer interfacing, error detection and correction etc.. If a user is connected to a VAN, either through his terminal or through his computer, he then has the potential to access any other computer or terminal also connected to the VAN. This is an example of the flexibility and resource availability that computer networks can bring.

Network Architecture

There are two main types of computer networks

(i) Centralised systems;

(ii) Distributed systems.

A centralised system connects remote terminals to a central computer system. Such systems are common - almost any timesharing system may be quoted as an example. However, we exclude them from this discussion. We also exclude from this section multiprocessor systems, where the c.p.u.'s all access a common memory.

A distributed system possesses several distinct computers that communicate with each other, and possibly with users, over a communication network. Even when we discuss computer networks as restricted to distributed systems, we find the need to separate two subtypes

(ii)(a) computer-communications networks;

(ii)(b) computer networks.

Note the ambiguous terminology (a computer network is a subclass of the class of distributed computer systems, which is a subclass of computer networks!)

The difference between these two types is the view the user has of the system. If he is using a computer-communications network then he sees a number of different computers providing different resources and services. He establishes a connection through the system to the computer he desires to use. Having done this he is no longer aware of the existence of the network - he uses the distant resource just like a local machine.

The user has to learn the procedures for using the distant machine, which will probably be quite different from his own machine and from most others in the network. We note in passing Martin Richard's proposed research into portable operating systems, which is motivated by using various systems in the ARPA net which differ from one another only in the trivial details of how they are used.

On the other hand, if a user is using a computer network then he views it as just a large computing system. The existence of the network is transparent at all times. He does not have to master a variety of control procedures, and indeed the identity and location of the resources he uses need not be revealed to him. They might also vary from time to time for the identical job as the load on the network varies. The network takes on the responsibility of allocating resources.

Of course the existing networks do not fall completely cleanly into these two categories - a computer-communications network may, in limited circumstances, exhibit properties of a computer network, and a computer network may not succeed completely in its aims of hiding the existence of the network from the user.

Network Topology

The ultimate in network topology is to connect every computer to every other computer in the network. This gives for n computers $(n \times (n-1)/2$ interconnections. This is both uneconomic and unnecessary. There are various ways of reducing the number of interconnections. One is by using a "star" configuration. One computer is selected as the centre of the star and all others connected to it. Communication between any two planets is via the sun. This may be generalised by constructing several subnetworks, each a star, and then connecting the suns together (as a super star!). Another philosophy is to form a ring, with each computer connected to exactly two others. This is a popular technique for small localised networks (i.e. geographically within a mile) and we summarise Fraser's survey of rings as a separate section in this report.

Of course networks are often like Topsy and just grow, without any detailed plan. Here the philosophy may be to connect each computer to up to three others that are geographically convenient. In this way the network can sprawl at will, and it only means establishing one or two links to add a new machine into the network.

The various topologies give rise to a number of technical problems. The fully interconnected network is uneconomic and wasteful. The star configuration falls flat on its face if the central sun computer fails. In most configurations messages are passed between two specific computers only via a number of other intermediate computers. In general therefore each machine in the network has to deal with messages that are just "passing through". If it is already using an output connection which is needed by such a message, then either it has to store this message until it has finished its own output, or it has to "reject" this message. Both solutions have their disadvantages. In some configurations there is a choice of routes between any two machines. This helps reliability enormously, and also can help to "even out" local transmission peaks, but introduces the problems of choosing routes, and more subtely, that of "overtaking". If one computer is transmitting a series of messages to another computer, then different routes may be chosen for successive messages and a later message may be sent along a "faster" route than one earlier message and so arrive ahead of it. Another problem with any network is that of detecting and dealing with errors.

Network Components

So far we have just been considering a number of computers connected together in a network. Unfortunately both the hardware and the software of many computer systems just have not been designed with networking in mind. This is not really to be condemned, simply accepted as a fact of life. To a large extent networks are still experimental and it is not possible to take the necessary decisions with any degree of confidence. Many networks interconnect a wide variety of machines, from various manufacturers, and so there are many differences to be catered for even in a computer-communications architecture.

As a result the networking function for each computer in a network is carried out by a network interfacing computer - usually a small minicomputer. The terminology is to talk about a "node" in a network. This consists of the main computer and the interfacing processor. In some networks (e.g. ARPA) there may be two or three computers at a node. However this is considered still as one node as there is only one interfacing computer which multiplexes access to the network between the several main computers.

The interfacing processor can take care of the various problems mentioned in the previous section. In particular it can perform the message store & forward and routing functions without bothering the main computer at the node. The usual name given to the main computer at a node is the rather misleading word "host". In ARPA terminology the

interfacing computer is called an IMP. ARPA also permits the connection of a cluster of terminals directly onto the network (i.e. not through a host computer). The necessary interface computer in this case is called a TIP (terminal interface computer). This allows conventional time-sharing access to any suitable computer on the network.

Network Protocols

For one computer to send a message to another computer across a network, more has to be done than simply pump the bit-train down an appropriate wire. As we noted above there are many side problems, such as message switching, errors etc. - in fact enough to warrant the use of special network connection processors. The job of these processors is to deal with all these side issues. However this implies that rather more information is transmitted between the interfacing processors (IMPs) across a network than just the appropriate message. Firstly we need source and destination information, then error checking information, then sequencing information (to check for lost duplicated or mis-ordered messages). We also need other messages to cope with correction from errors, e.g. request to retransmit.

All this adds up to a "protocol" - a system of standard message formats together with a set of rules for their use. The types of messages described in the last paragraph form the basis for an IMP to IMP protocol. All the IMPs on a network will supply messages in the standard formats as defined by the protocol, and will obey the standard rules about which messages can be transmitted that the protocol lays down. This they do regardless of the particular host machine that each one interfaces to.

This last sentence implies the existence of other sets of protocols - IMP to HOST protocols. One job that the IMP has to perform is to accept messages in the format that the particular HOST provides, and to "translate" to the standard IMP to IMP protocol. The choice of IMP to HOST protocols is largely dictated by the particular host machine, and thus is regarded as a technical rather than intrinsic problem. It does however influence the choice of IMP to IMP protocols, the latter have to be selected to be within the bounds of possibility implied by the former.

A third level of protocol is that of HOST to HOST protocol. In most networks considerable adaption of the host machine software has had to be carried out to enable sensible use of the network. In point of fact we may find several "types" of communication that a job can distinguish within a computer, e.g. terminal I/O, file I/O, communication with other jobs. In a networking environment we may well find several jobs within a machine using various of these via the network to other computers. This variety gives rise to the need for a high level HOST to HOST protocol, so that a host can differentiate between the various incoming messages and deal with them accordingly. As well as source and destination node information (handled by IMP) we need source and destination software object information.

Software Considerations

As pointed out earlier, most computer systems are not designed with networking in mind. For example a common situation is that all terminals will be interfaced through a hardware multiplexor, and this assumption pervades deeper into the system than it ought. Suddenly a set of terminals is available through the network interfaced by the HOST to HOST protocol software. A measure of the quality of the software design probably may be obtained by the ease of interfacing this new source of user terminals to the system.

In fact most systems require the development of a network control package (NCP). In theory this implements the HOST to HOST protocol, but the practical rammifications often extend far deeper into the system.

Two philosophies have emerged over the role of the network itself in the communications functions it supports. We have noted that within a host computer we may have several jobs (processes or tasks) all communicating over the network with similar jobs on other host computers. The two network philosophies supporting this are

(i) the datagram system. Each message from each job has destination information incorporated (as part of the protocol). Each IMP along the route uses this destination information to decide the next IMP to send the message to (other information such as loading availability may also be considered). Ultimately the message arrives at its destination;

(ii) the virtual circuit system. Each IMP supplies a system of logical input and output ports. Communication takes place between jobs in three phases. The first is the establishment of a virtual circuit through a succession of IMPs connecting the hose node to the destination node. This virtual circuit is then used for as long as the two jobs are communicating. The final phase is the disconnection of the virtual circuit.

The datagram system is relatively less demanding on the IMPs but relies upon sophisticated HOST to HOST protocol (e.g. this protocol has to deal with error situations).

The virtual circuit system throws much more responsibility upon the IMPs and requires a sophisticated IMP to IMP protocol. However, the routing is only performed once for a series of communications.

Packet Switching

This technique used almost universally in one form or another has been developed to alleviate some of the problems that arise in a network. Messages tend to vary considerably in size (from a single character to a complete book) and IMPs have a problem of finding buffer space, possibly for complete messages. With long messages the probability of an error occurring is high and it takes a long time to correct. Long messages can also monopolise physical circuits for unacceptable lengths of time.

The solution to these problems is to break down all messages into one or more fixed length packets, which are transmitted as separate units. More protocols are needed, but this is only an extension of facilities already provided to meet the basic needs. Packet switching is used in both the datagram and virtual circuit philosophies described above. Only two packet switching networks existed three years ago, but now such networks are under development in many major industrialised countries.

Networking Research and Development

We devote separate sections to Fraser's "Spider" system, his survey of other ring networks, and Farber's distributed computer system. The most well known network is the ARPAnet. This has about 50 nodes, mainly in America (there are two in Europe), each node existing in a university or defense-associated research organisation. Besides simply supplying a tremendous wealth of resources at each node, the ARPAnet has fostered much collaboration and interaction between geographically separated like-minded people. Much research is being pursued at various establishments on the individual systems connected to ARPA, on protocols, and on the problems emerging through use of the network. About a dozen other major networks are now under development. Various of these are described in the proceedings of the various network conferences (e.g. [1]), and much is collected together in the technical notes of the International Network Working Group (IFIP TC6.1), available from Vinton Cerf at Stanford Digital Systems Lab.

A valuable source of further information is the recently revised "Computer Networks : a tutorial" [3] which contains a collection of 34 reprints of leading papers, together with articles on various aspects of networking.

For further reading see [21, 27, 74, 75, 81, 86, 104, 110, 113, 116].

Discussions with Vinton Cerf

Cerf, of the Digital Systems Lab, Stanford University, is interested in a variety of aspects of networking [18, 19]. He is also the secretary of IFIP TC6.1 International Network Working Group, and edits the INWG notes. Recent contributions give a good idea of some of the "current" issues in the study of networking [18, 60, 73, 84, 108].

Cerf is particularly interested in the problems of communication and resource allocation. He assumes a network environment in which there is no central resource allocation mechanism. Instead a resource is interrogated and either a system of demand sequencing is used, or the resource maintains a queue of demands. This can lead to problems of queue overflow. Several resources may be demanded simultaneously, and problems of deadlock can arise. One possible solution to the latter is to impose a well ordering structure on the conventions for asking for resources. Another possibility is to ensure that resources are always available by duplicating them, maybe using a "virtual resource" system rather than by supplying many copies of a physical resource. It is possible that such virtual resources can be managed each by a small processor. The problems of queueing (and deadlock) are not totally alleviated.

Cerf has been investigating some of the problems of failure and deadlock that occur in the ARPAnet, and evaluating the alternatives of deadlock prevention as against deadlock detection and recovery. He observes that the flow control system in the ARPAnet works satisfactorily until a failure occurs. He believes that communication failure detection is essential in a network and suggests that messages be transmitted, and nodes interrogated, simply to prove that they are still "alive".

A.G. Fraser on Spider and Other Networks

Sandy Fraser has developed a ring network at Bell Labs described fully in a technical report "Spider - A Data Communications Experiment" [43]. A short description of the network is given in the abstract:

"Spider is an experimental packet switched data communications system that provides 64 full-duplex asynchronous channels to each connected terminal (= host computer). The maximum data-rate per channel is 500 K bits/sec. Error control by packet retransmission is automatic and transparent to users. Terminals are connected to a central switching computer by carrier transmission loops operating at 1.544 mb/s, each of which serves several terminals. The interface between the transmission loop and a terminal contains a stored program microcomputer. Cooperation between microcomputers and the central switching computer effects traffic control, error control, and various other functions related to network operation and maintenance".

The current system contains just one loop with the switching computer (TEMPO I), four PDP-11/45 computers, two Honeywell 516 computers, two DDP 224 computers, and one each of Honeywell 6070, PDP-8 and PDP-11/20. In fact many of these are connected in turn to other items of digital equipment.

Spider has been running since 1972 and recent work has shifted away from the design and construction of the network itself to developing user services to support other research activities at Bell Labs. A major example of this has been to construct a free-standing file store (extracted in fact from UNIX [88]) and connect it to the network. This is available as a service to any user of the network.

Fraser gives the following two criteria in the choice of a "common user data service function" :-

(a) Technological independence of the host computer from the workings of the network;

(b) Communication availability between any two computers connected to the network.

Both of these imply the need for a relatively sophisticated interface between the "terminal" (i.e. host computer) and the network. Fraser's approach is to connect the terminals to a ring using terminal interface units (TIU's) - small interfacing microcomputers. However there are many functions which he centralises into a switching computer which "controls" the ring, rather than place them in the TIU's.

The ring is used in different ways by the various computers connected to it. The filestore has already been mentioned. Two computers use this for conventional back-up, and access it on a file-by-file basis. Two other machines - dedicated to laboratory experiments - access it on a block-within-file basis. To help with program development for these dedicated machines, the UNIX system (available on yet more computers) is used during "unavailable" periods for editing and program preparation. The user then leaves his programs in the filestore ready to load when he next gains access to the dedicated machine. Two other "dedicated" machines provide the user interface of UNIX, but lack peripherals and a UNIX kernel! In place of both is a small amount of software that transmits all calls on the UNIX system to a full UNIX system elsewhere! The ring system with its filestore also provides a convenient buffering service. Finally Fraser tells of the time where one of the PDP-11 machines was delivered sans discs. A small alteration to a UNIX system diverted all disc transfer requests to the filestore, where a suitable amount of disc was made available. The system ran a full swapping version of UNIX at about a quarter of the normal speed.

Fraser has been investigating the problems of reliability of networks. In the Spider network, the TIU's are designed to "short circuit" on failure, so as not to break the ring. This is carried out by a Line Access Module (LAM) which leaves the TIU in a state where it can still receive packets, but has no control over what is transmitted. Unfortunately the relay switching in the LAM causes problems when system reconfiguration takes place. Fraser finds that the terminal /TIU interface is a source of trouble. The transition between different electrical and physical environments seems to be a potent source of error.

At the more general level Fraser considers that it is only worth while using a remote machine if it is far better and much more reliable than a local one. The user has only the network interface, and if the distant machine is fallable, or ignores him from time to time, then he will find its use a frustrating experience [41].

Fraser is reflecting on the analysis of and construction of networks at the national level. Packet switching is a necessary feature, however he considers that mini (or micro) computers are not suitable for interfacing to a network as they are too slow, and hardware/logic solutions will be necessary. This still leaves the problem of protocols, for which ad hoc solutions are found due to the lack of formalism for its consideration.

In the development of the Spider system, Fraser has worked on the problems of supporting microprocessors - such as FLY (SPIDER's TIU). He has developed a timesharing environment for FLY, which enables operations such as Compile & go, halt, single-shot & interrogate from the terminal. To perform these operations on a particular FLY, it is necessary merely to plug it into a socket on the environment computer.

Fraser is also working on an alternative to the systems such as the DEC UNIBUS system with the objective of being more reliable. The system will not be daisy-chained, and probably not as fast, but will transmit over greater distances.

On the topic of current issues in networks, Fraser considers that we are now at the stage with protocols which corresponds to the stage that languages had reached in 1955. Much work is needed, particularly in the development of appropriate formalisms. As yet nobody knows the answers. Packet switching is an important technique. Packets should be small (about 8 characters together with a condensed destination). Error detection should be external to the switching mechanism, and should be end-to-end, not switch-to-switch. Retransmission usually does not cost very much. If a route seems unreliable then it is usually better to find another route. An interesting problem is automatic error detection and reconfiguration, which should be transparent where possible to the user. Ultra-high speeds are not necessary.

Loops for Data Communications

Sandy Fraser at Bell Labs has written a survey with this title [42] surveying the various ring systems that have been developed. He points out that, with "bursty" demands on a transmission system [38], demand sharing of transmission lines promises minimum delay for given cost, and that this technique is easily achieved on a digital transmission loop. The demand sharing techniques allocates all of the bandwidth to a given terminal for small portions of time. The loop system solves the problems of contention for service, buffer storage and distributing control information, three problems that arise in a demand sharing system.

Fraser surveys 6 loop systems, including the original experiments by Newhall and Farmer [38], Fraser's own Spider loop and Farber's DCS loop [30]. He discusses the problems involved in loop reconfiguration (for reliability). He concludes that the problem is not simple, and requires the services of a loop supervisor, equipped with automatic diagnosis and repair facilities. One of the major difficulties is to set the error detection thresholds at just the "right" level. Fraser then discusses the problems of blocking (when one terminal floods the network and so prevents any others from using it); traffic flow analysis and loop queueing delay; distribution of service (e.g. the effect when one node produces or consumes much more than the other nodes, as in a filestore); switching throughput; and costs and pragmatics.

Fraser concludes that several users can share a transmission loop without undue discomfort or interference, and that the technique is well suited to demand sharing. It is not difficult to protect a loop from most types of terminal malfunction. In a large heterogeneous network as much control as possible should be centralised, to handle functions such as network call and network maintenance. This management and maintenance function unfortunately adds to the initial establishment costs of a network.

Siewiorek's Computer Modules

Dan Siewiorek (with S. Fuller) is establishing a project at Carnegie-Mellon with the overall aim of building big systems with a speedup of design and construction. He hopes to achieve this aim by developing an architectural technique based on computer modules as building blocks [44]. This approach has become feasible as LSI has permitted the development of microcomputers. Siewiorek envisages systems that may contain several, 100 or possibly up to 1000 microcomputers.

The idea of a computer module has developed from C. Gordon Bell's register transfer modules (RTM's) [6] [54]. (See also Bell's "bible" [5]). These were seen several years ago as providing an easy way of building computers. (This work is worthy of study, but is outside the scope of this report). A computer module may be thought of as on the scale between an RTM and a processor of the C.mmp system.

The basic philosophy of Siewionek's approach is to put processors next to memory. This forms a processor/memory pair - a computer module. The system is then built up of computer modules suitably interconnected. A system would possess sufficient computer modules for each one to be assigned a particular function within the system. The main topic for research is the technique of connecting computer modules together.

Siewiorek envisages the possibility of using the network at the memory reference level. This instantly rules out techniques developed for networks such as ARPA which are slow, with the overheads associated with a fixed formal message system. Techniques developed by Farber at U.C. Irvine for the Distributed Computer System, using a high speed loop, are still too slow for Siewiorek. If one considers multiprocessors then the Prime system at U.C. Berkeley has a limited crossbar which is superseded by the C.mmp system. However in this system the correct functioning of the switch is crucial.

The outline of Siewiorek's architectural plan is as follows. Each computer module consists of a PDP LSI-11 processor and local memory, together with an interface denoted by K,q. The K,q interface performs a naming function and connects the computer module to a bus. Amongst other things the K,q interface directs memory references that cannot be satisfied by the local memory onto the bus. Up to eight computer modules may be connected to a single bus to form a "cluster". The bus is an "intelligent bus" and has an associated address mapping computer (an Intel 3000) connected to it. This takes the 16 bit LSI-11 addresses and translates them into 28 bit system wide addresses. This operation is called the K-map. The busses are interconnected by special interfaces called Kc modules. The K-map operation takes 5 microseconds if there is no competition for the bus. The "intelligence" is necessary as it enables a form of message switching to avoid deadlocks. Incoming bus cycles have to be allowed whilst external references are being attempted.

Siewiorek is planning to keep the degree of multiprogramming within each computer module at a maximum level of 2. He is building a kernel system which will include a message transaction system.

This project is at an early stage, but it is interesting to see Siewiorek's budget. His initial system will have 10 LSI-11 computers, costing about $35,000. The total cost at this stage will be about $100,000. The second increment will have 100 processors and will contain 1½ MK of memory!

VI

Distributed Systems

Under this heading I report on a number of systems which have one architectural feature in common: they have a number of computers, communicating using a message system, and the distributed nature of the system is transparent to the user. Firstly I describe the DCS at the University of California, Irvine. This is the closest system to that proposed for the Warwick software project, and so it is presented with (comparatively) more detail than normal. A number of similar developments are described, including one aimed at giving the ARPAnet the above-mentioned characteristics. Finally some initial ideas originating at Stanford on the subject of distributed databases is presented. The question of "to distribute or not to distribute" is an open one. We know more about how to organise a central database, but reliability is decreased, and communications overhead increased. At the current state of the art, one of the several particular distributed schemes may suffice, but we have no overall methodology or formalism for considering general access and storage techniques for distributed databases.

The Distributed Computer System

A team at the University of California, Irvine, led by David J. Farber, has been developing the D.C.S. for about four years. The system is well documented [30-37, 47, 53, 73, 90].

Farber considers a "distributed system" as being composed of a number of minis connected together by a communications system and viewed as a single distributed machine rather than as a network of independent minicomputers. In considering the design aims for such a system, Farber lists the following "technology pulls" towards distributed processing:-

1) changes in cost curves for processors

2) low cost per bit very large stores

3) low cost small stores.

In addition a number of applications require distributed databases and specialised processing needs. Farber also lists a number of economic motivations:-

1) modular growth

2) system reliability

3) availability of communications

4) low vulnerability of the system

5) incremental modernisation

6) dynamic restructuring.

He notes current issues of maintainability, operability and cost effectiveness, and he believes that D.C.S. will provide appropriate solutions. Future issues will be efficient preplanning of information flow, fault detection, hardware and software vitality, efficient distribution of information and structured programming technology applied to distributed processing. He concludes that distributed processing offers:-

1) modular growth

2) fail soft behaviour

3) incremental modernisation

4) dynamic restructuring

5) decreased design development time

6) ease of system validation.

The DCS developed at U.C. Irvine consists of 3 Lockheed SUE computers and 2 Varian 620 computers connected to a unidirectional ring. Each Varian contains a disc unit. Various other peripherals, including 6 terminals, are connected via the various computers. Each computer is connected to the ring via a ring interface. The ring operates at 2.3 Mbaud. The ring control is distributed between the ring interfaces and a system of variable length messages is used.

An interesting feature of the system is that the communication system is process oriented, not processor oriented. Thus each message sent around the ring has a process (or possibly a set of processes) as destination. This is independent of the location of the particular process. Each ring interface has a sixteen name associative memory containing the names of the processes currently located on the attached computer for fast recognition of relevant messages. Thus hardware, software and control are all distributed between the various components of the network. The system is both redundant and modular, and does not depend on any one active component. It is thus relatively immune to total failure, and exhibits fail-soft behaviour.

The communications system may be considered as a series of message slots. To transmit a message the ring interface waits for an empty slot, and then places it on the ring. The message is passed from ring interface to ring interface around the ring until it returns to the originator, where it is checked against the original and removed from the ring. This scheme allows messages to be broadcast to all processes, or classes of processes - a valuable feature. If a message is mutilated or lost then it is retransmitted. A sequencing system is used to allow receiving processes to deal with the multiple copies that this system might generate. For reliability a duplicate pair of cables is used, and a distributed mechanism is employed for removing circulating messages (the identity of the originator is corrupted so nobody recognises them to remove them from the ring) and a software mechanism can amputate a failing component.

Each processor functions under its own operating system kernel. The system is process oriented and the kernel makes the network transparent to the individual process. Processes communicate by sending messages to each other, not by sharing memory. Within each processor the kernel simulates a local ring. If a destination process is not on this local ring (because the process exists in another computer) then the kernel directs the message onto the hardware ring. Thus a process does not need to know where another process physically resides in order to communicate with it.

The structure imposed on the process system follows the classical Dijkstra style hierarchy. Level 0 contains process scheduling software while level 1 implements interprocess communication. Each process has a set of input channels, with each of which can be associated some other processor class of processes, and one output channel. A process can "wait" for a message to appear on one or several input channels. Level 2 contains routines to check the ring

and other processors for malfunctions. A testing process broadcasts a message round the ring which should meet with a response from all other machines. It may decide, after further examination, that a particular machine is ailing, in which case it transmits a special closedown message to the appropriate ring interface. A ring interface will disconnect itself from the system if it receives two such messages from distinct machines within a short space of time. Enhancements to this scheme provide for automatic reloading of the system at a particular node, and informing other processes in the system of the action being taken.

Level 3 software forms a distributed resource management scheme. This is based on an economic model. When a resource is requested, a message is broadcast round the system asking for "bids". Several machines may offer the service, but at differing rates. A certain amount of control is needed for such a system and all contracts are recorded and approved by a notary process. The system is carefully designed so that the controlling files are not critical and easily reconstructed, thus the system will continue to run if those files are lost.

The goals of reliability and fail-soft operation have led to the development of a distributed file system. Each file retains its fully qualified global name so that a file can be uniquely identified and located regardless of the availability of other components of the file system. The system does have a structure of directories and catalogues, but has the unusual feature that each such structure can only be accessed through a corresponding process. At the "top" level each processor has a copy of a central component process. This process provides the name of a catalogue process which serves the fileowner. All the central component processes are identical. The catalogue (which will service several owners) provides a translation of the owner and filename to the volume (the physical storage medium) of the file. The particular volume process performs any necessary acces checks, and creates a file process associated with the file. Access to the file is via this particular process. The volume and file processes operate on the physical machine containing the storage medium. If this is moved, then so are these processes. All traffic (addressed to the process) is re-routed automatically. A volume process is easily created when the physical storage medium is loaded. Failure of a catalogue process slows down the system until the process is re-established, but it can be operated in an incomplete state. From the user's point of view it is his responsibility to take any further precautions he considers necessary by creating backup copies, archived copies etc. etc..

This mechanism allows any desired security policy to be implemented and any desired access modes. The access process could be a sophisticated database language interpreter in relevant circumstances. Farber also points out the advantages of security in a distributed system - each volume could have its own security policy which would have to be penetrated, and even total physical destruction of a geographically dispersed file system is difficult.

Jensen's Mini-Multicomputer

E. Douglas Jensen, of Honeywell Inc, Minneapolis, is developing a distributed computer system aimed at real-time control applications [56]. He assumes that enough is known about the application to be able to partition the processing requirements into small self-contained functions which are allocated to the processing elements effectively on a one-to-one basis. Functions which always execute at different times may be allocated to the same processor, but those which may require activation at unpredictable times use a dedicated processor which is idle when the function is not active.

The motivations for a distributed system are to achieve this functional modularity and to distribute processing power to geographically distinct but related centres of need. The machine is organised on a system of global communication busses. Processors communicate with each other and with I/O devices using these busses. The busses each carry 16 communication lines (plus some for control). A system may contain up to four busses, though each module (i.e. processor or I/O device) connects to all busses. Control of the bus is distributed across the standard interfaces that are used. The standard interface uses an associative memory of up to 256 four character names to identify the processing elements connected to it (albeit software processes, or I/O devices). Thus destinations are identified symbolically.

The system permits many of the functions normally associated with interrupts to be carried out within the message system.

Distributed Computation Research at Bolt Beranek and Newman Inc.

The goals of this research program are [106]:

1) To identify and understand fundamental problems of computing in a distributed, multicomputer environment;

2) To determine the impact of communication networks on computer systems;

3) To develop techniques which enable convenient and effective use of the resources distributed throughout a computer network.

The approach has been pragmatic - to use the existing ARPAnet and TENEX system as a base for developing software to meet goals 3, and thus achieve goals 1 & 2.

The main practical achievement has been the development of RSEXEC [107] - the combining of a number of TENEX's as one system distributed across those installations supporting TENEX on the ARPAnet. A major component of RSEXEC is the distributed file system which spans host boundaries. The user can access files without having to remember where they are. However, he can specify one or several hosts for storage of a file, thus keeping multiple copies. R. Thomas has been working on the problem of maintaining consistency of such a distributed filing system, especially if several people are updating shared information. An example of this use is the access control system [20] - which can run in any convenient RSEXEC system for any TIP. The database of users and passwords has to be replicated for reliability, and may be updated from any of a number of points (e.g. when a user chooses a new password).

In theory the user simply logs in via the local TIP, and requests TENEX service. He is connected to an arbitrary machine. The current state of development is that the filing system is distributed, but on logging in the user queries the state of the various TENEX systems available and selects one. Thereafter, however, the network is transparent. This system (the access by TIP to RSEXEC to provide the user with information) demonstrates the feasibility of having small hosts (TIPS) share the resources of larger hosts to provide the user with an extended range of services in a network transparent manner.

An interesting example of other network software that has been developed is the program CREEPER which demonstrates the possibility of a program "creeping" from host to host whilst in mid-execution. This principle has been extended in the dynamic multi-host simulation system called McROSS, which includes consistent application oriented primitives to support the construction and execution of multi-computer programs. Experience with McRoss led to the development of the Multi-TELNET system which allows the user to control conveniently a number of jobs on different computers from a single terminal.

The research report discusses the concepts, techniques and issues that this work has covered, and identifies areas for further research. These are

1) Distributed databases - the development of a coherent methodology for their design, implementation and management.

2) The use of "persistence" as technique for achieving reliable task completion in a distributed environment.

3) Hot Switchover.

4) Efficient job configuration.

A discussion of the work on duplicate databases carried out at BBN [58], together with a comparison with other techniques of solving similar problems, is presented by H.C. Forsdick, a graduate student in the EECS department at M.I.T. [40].

Richard Schantz has been involved in a number of aspects of interprocess communication across networks [57] formerly at S.U.N.Y., now at B.B.N. In particular he is co-author of a survey paper on this topic [4].

Discussion with M.F. Kraley on the Pluribus (B.B.N.)

The motivation for the development of the Pluribus [49, 77, 78] arises from speed problems (current and anticipated) in the IMP processors of the ARPAnet. The aim was to develop a machine with ten times the speed of the current Honeywell H316.

A conventional minicomputer is used, the Lockheed SUE computer. This machine is built around the bus principle, but unlike, say, the PDP-11, the bus arbitration and control is separated from the processor. Thus the number of processors that may be connected to a bus ranges from zero upwards.

A limiting feature of "bus" architectures is ultimately the bus bandwidth. The solution devised by Ornstein is to extend the architecture to have several busses, interconnected by bus couplers. The bus coupler performs address translation and parity checking. The SUE bus has a 20 bit wide address field and a 16 bit wide data field. There is thus plenty of addressing space to allow implementation of this architecture.

The system supports three types of bus. All busses have their own power supply and bus arbitration card.

(a) Memory bus. This supports 16 K (could be more) of "shared" memory.

(b) I/O bus. This connects the system to the outside world.

(c) Processor bus. This supports two processors and 8 K of local memory.

In general a system will have a number of busses of each type. The philosophy of bus interconnection is to place a bus coupler between all pairs of busses of differing type. For an IMP the configuration chosen is 2 memory busses, 2 I/O busses and 7 processor busses, giving 32 bus couplers. In fact this size is typical rather than definitive, as some IMPs may well be larger than others.

A heavily used "resource" is program code. The most heavily used code is placed into the local memory to a processor pair, and thus only about 1 in 4 accesses to memory use a bus coupler. The system has high reliability as a major goal (occasional faults can be tolerated, total system failure cannot). For this multiple copies of programs and data have to be isolated, so that no single failure can take out all copies. There is no fixed assignment of processors to tasks, nor I/O devices to c.p.u's.

In a real-time system there is a problem of interrupt servicing. The simple, though inelegant, solution is to break down all functions (including I/O servicing routines) into small tasks of typically 100 machine instructions. With each task is associated a number, and the system supports a hardware implemented task queue, called a PID (Pseudo Interrupt Device). Both peripherals and (software) tasks can arrange to queue tasks in the PID. When a processor comes to the end of a task it interrogates the PID for a "suggestion" as to the next

task to be tackled. Conceptually there is one PID for the whole machine, in fact for reliability several PIDs are used.

The philosophy of reliability is interesting. The responsibility for reliability moves into the software. Each task is coded in such a way that it can be called when there is no work for it to do without harmful effects. This means that if a task is called erroneously, no damage is done. In fact all tasks are called periodically, just to keep them going. This philosophy means that no damage is done if the "wrong" task is called - the correct one will be called ultimately. Indeed if a complete PID should fail (and the associated queue lost) and a switchover is made to a reserve,then the system will recover gracefully. The problem of containment is also catered for. If an "active failure" is detected, then one or more bus couplers can be disconnected using an "amputation switch".

Currently all coding and task decomposition is performed by hand at the level of assembly code. A prototype pluribus system is operational and is being used as an IMP on the ARPAnet.

The National Software Works

One of the criticisms that is levelled at the ARPAnet project is that it has not succeeded in its goal of resource sharing, amongst its members. The main use made is for file transfer and other forms of message sending.

A possible explanation is that there is little "network transparency" from the user's point of view. He has to maintain an account, files, and an operational skill for each remote system he wishes to use.

The NSW project has been established to tackle the problems of heterogeneracy in the context of reducing software development costs. In his paper [22] Crocker points out that in 1972 the American Air Force spent more than $1,000,000,000 on software, yet software development is a labour intensive industry with little automation. Currently we find that a large number of software development tools have been built, though few are in widespread use, for a variety of technical reasons. The aim of NSW is to provide users with access to software development tools on whichever machine the tools happen to be.

NSW is in effect a network wide operating system, which gathers up the resources available and presents a common interface to the user. The user thus sees just one operational interface, has one set of files and one account number with NSW. The network is made as transparent as possible. It is the job of NSW to make sure that the right files are in the right place at the right time, and NSW will have its own account with each system providing resources.

The structure of NSW is tri-partite. Firstly there are the (widely differing) "Tool Bearing Host" machines. Each software development tool is available on the TBH that is most appropriate. TBH systems are simply the ordinary systems with the necessary extra software to support the NSW. Such systems are under development at Bolt Beranek & Newman, Cambridge, Mass, University of California, Los Angeles, and the Massachusetts Institute of Technology. The second part of the structure is composed of terminal supporting Front End systems. This is the user's end of the system. The Front End development is taking place at Stanford Research Institute. These two parts are connected (logically) by the "Works Manager" - the central controlling software. Actually it is planned that, for reliability purposes, the Works Manager be a distributed system. The Works Manager's job is mainly administrative - to provide the user with a common interface to all resources. However, where many instances of a resource may be found, the Works Manager will also provide a suitable scheduling function.

The NSW is still very much in the design stages, though a limited demonstration is anticipated soon. It is necessary to impose a further structure over and above the ARPAnet HOST-HOST protocols, but it is still a matter for debate exactly how the purpose of the system is divided between Front End, Works Manager and TBH.

The current plan is for the Front End to be rather more sophisticated than a TIP. In particular each "tool" will have associated with it a grammar describing the syntax of the operations that the tool will accept. The Front End will maintain a copy of the grammar for the tool currently in use, and translate the user's requests from the common interface language into the language the tool (or TBH) will accept. A standard command language is under development. (However it is possible that the user can use his own favourite command language - provided that a translator is written that will accept the system grammars!)

Rich Schantz at Bolt Beranek & Newman is developing a TBH for TENEX (the BBN PDP/10 timesharing system). Amongst the problems that he is investigating is the tricky one of intercepting all file accesses to see if they are of local interest only or should be passed back to the Works Manager. It is difficult to do this cleanly correctly and "transparently" in all circumstances.

Crocker in his paper mentions other issues as yet unresolved, such as linespeed (is ARPA good enough?), local/remote computing balance, tool documentation and support issues etc.. This project should make ARPA more amenable and as a result allay the criticisms mentioned above. However, ultimate success will depend on how much it is used, not simply on whether it fulfils its objectives.

By way of a footnote, Dave Clark at M.I.T. believes that the NSW is not a well conceived project, in that it starts with a well defined set of resources (program development tools) and tries to "share" these resources amongst the various Airforce programming teams. A better approach, Clark believes, to this problem would be to select a complete system and deliver it to the Airforce. At the technical level the NSW is based on a "procedure call" type of protocol, and lacks interprocess communication. Clark is of the opinion that resource sharing is based on data, not processing capability. The problems that ought to be tackled are those of shared data management and multiple copy files.

Network Operating Systems at Digital Equipment Corporation

Stuart Wecker has been investigating the design problems of a multi-function, multi-processor, minicomputer operating system [111, 112]. Earlier work in the same area has been carried out by Metcalfe [74]. We quote from the abstract of Wecker's second paper [112]:-

"From an intuitive feeling, it seems that connecting many inexpensive minis together should result in a system where the total capability of the hardware equals or exceeds that of many larger single processor systems while still being very much cheaper. The problem thus becomes one of how to connect the systems together, how to structure the operating system(s) to operate with the hardware structure, and how to take advantage of the total capability of such a system".

Wecker examines the two approaches to multiprocessor systems - close-coupled (memory sharing) and loose-coupled (over a network) in the light of the goals of reliability, performance and sharability, and deduces that a loosely-coupled design is to be preferred. He then argues (as if he needed to!) for the software multi-processor approach, where each process thinks it is running as its own processor.

He creates a system in which processes are independent with respect to physical location and parameters of execution. They exchange messages via the system nucleus which is responsible for the physical routing. Processes synchronise their actions via a set of events, which are 1-bit markers belonging to a process, accessible to other processes in the system. Processes thus execute in a sequential uninterruptable voluntary wait mode. Each process has a data channel where all messages for it are queued.

The system is effective but simple, and simple are its protection mechanisms. Each message has the identity of the author process inserted into it by the system, so each service process can check on the processes trying to use the service. The remaining problem concerns events. Without any protection it would be possible for a malicious process to mark any events of any processes it chooses. To protect against this a double scheme of marking rights, and a special marking process, is used. If a process wishes to mark an event in another process for which it does not possess an inherent marking right (established at process creation time), then it sends a message to the marking process which performs the necessary security checks and carries out the actual marking.

A prototype of the system has been developed for a PDP-11/20 and work is in hand in developing the system on a physical network.

Distributed Databases at Stanford

Whilst I was at Stanford, two research students, C. Widdoes and L. Trabb-Pardo, described some initial ideas on catalogue structures in a distributed data base. Their aims are to derive a system in which locating and moving objects is fast and reliable.

Two main principles underly their thinking:-

1) a separation of catalogue information from the node containing the object;

2) duplication of catalogue information for efficiency.

Their system faces squarely the situation, implied by these two principles, that catalogue information may be out of date or otherwise incomplete and erroneous. The system is general in that it is applicable to finding any sort of object in a distributed system, e.g. processes or users, as well as files.

Conceptually we may envisage a master directing L which defines the mapping of each object ℓ to its location (host). We assume that all names are globally unique. This directory is partitioned into a number of smaller directories L_i such that $\bigcup_i L_i = L$. This could be done, for example, by defining a hash function

$$g : g(\ell) = i \iff \ell \in L_i$$

More precisely L and L_i are sets of names; we can define a function H of names to locations. We define a catalogue for each L_i called

$$C_i = \{< \ell, H(\ell) > \mid \ell \in L_i\},$$

this defines the function H for all $\ell \in L_i$.

Ideally one might wish to put a copy of all the C_i catalogues at each host. This would then mean that the function H could be determined instantly for any given ℓ. In practice this is both waste of resources and leads to problems of ensuring consistency. Indeed the network would have to work very hard to keep all the C_i up to date.

Instead, Widdoes and Trabb-Pardo propose to keep just a few copies of each C_i at certain selected nodes. Each copy is not necessarily up to date and may not contain details of all $\ell \in L_i$. So we define "copies" of C_i called C_{ik} such that

$$\bigcup_k C_{ik} = C_i \qquad (o \leq k \leq n_i)$$

In addition we create a "buddy list" B_i which defines the nodes at which we choose to store the C_{ik} :

$$B_i = < h_{io}, \ldots, h_{ik}, \ldots, h_{in} >$$

h_{ik} gives the location of C_{ik} .

We may also choose to move the locations of the C_{ik}, so each host maintains a perhaps out of date copy, called the Catalogue Distribution Map:

$$CDM_{\ell}(i) = < h_i, h_2, \ldots, h_n >$$

Thus each host can determine, for a name, ℓ, the class it belongs to by applying $g(\ell)$. If it does not contain one of the copies of C_i, or if it cannot find a reference to ℓ in the copy, then it uses its $CDM(g(\ell))$ to determine the other hosts it should interrogate who might know about ℓ.

This work is in its initial stages, but Widdoes and Trabb-Pardo are able to demonstrate protocols for ensuring fail-safe consistency for the file operations create, locate, withdraw, and more. The redundancy provides speed as well as reliability.

VII

Technology, and Personal Computers

The development of our art/science/technology is motivated by a mixture of economic and altruistic considerations. The subject is technology driven, in that economic motivation has led to the continual development of new hardware technologies which we must learn to use, and use properly and well. Often, by the time that the lesson has been learned, technology has moved on, and we have to develop a new set of techniques. L.S.I. is the latest technological baby, and it bears the promise of near-zero cost non-mechanical hardware when it reaches maturity. This reduction in costs implies that we can consider giving the individual user as much memory and processing power as he wants - for his own use. No longer will it be necessary to write large resource sharing operating systems. The problem that LSI does not solve is that of information sharing, and hence we see the development of personal computers connected to a network [7]. A commercial system of this form is the Linolex system, using a small micro-programmed "business" computer which also acts as an intelligent terminal [66]. IBM are developing "something" that will sell at about $5,000 and will include a floppy disc (further details not forthcoming). Digital Equipment have recently produced a system called the Classic($8,000 including 2 floppy discs), which is based on the PDP/8. Although aimed at the educational market (not financially the most flush), this is a form of personal computer. Two other "personal computer" systems which include a networking facility, are described in this section.

The impact of LSI

Large Scale Integration is the name given to the technology that allows a large number of circuit elements to be contained within a single chip. Currently available are some 60 different "c.p.u. chips" and a wide variety of memories - of up to about 4K bits. So currently L.S.I. can support a small microprocessor or a small amount of integrated circuit memory on a single chip.

The best available microprocessors are comparable with small minicomputer c.p.u's vis à vis functional capabilities and speed. Most currently available microprocessors are somewhat slower and less versatile.

The best available L.S.I. memories are considerably faster than equivalent core memories, but most are volatile.

The impact that L.S.I. promises is due primarily to the low cost mass-production techniques that can be applied. L.S.I. holds out the promise of a "$5 computer", provided that enough can be sold. This machine is likely to have rather different external characteristics than today's machines, due to its application area, which will probably be in domestic appliances, motor cars etc..

However it seems likely that the hardware costs of the non-mechanical parts of several purpose computers will continue to fall. The development costs of L.S.I. chips is falling. Currently it is estimated that a c.p.u. chip costs $100,000 to design. With near-zero manufacturing costs this means that a production run of 1000 chips would cost $100 each! If these costings are correct, even to within an order of magnitude, then we could well see an explosion in the number of different designs of computer available - and could well find that many designs are oriented towards new application areas (e.g. washing machines, carburettors). There are already over 50 different c.p.u. chips available. This figure could well reach 1000 within another two years. Adam Osborne [79] defines three types of microcomputers based on L.S.I. c.p.u. chips. The first is simply the unsupported chip itself. To use such a chip takes a considerable amount of extra logic supporting memory, I/O interfaces, control sequencing etc., as well as power supplies. In some cases multichip cp.u's are available, of greater flexibility than a single c.p.u. chip. The second is the computer card. This surrounds the c.p.u. chip with some logic to provide control sequencing and a"clean" memory/peripherals interface. Basically the computer card is much easier to use. Sometimes the computer card will contain a small amount of memory. The third form is a complete computer system.

We make two observations. The first is that a c.p.u. chip is a long way from being a complete computer system. While the c.p.u. chip may only cost $100, the computer system may cost $2000 (without peripherals). The second is that there are a range of applications for each level of microcomputer, but it is only the third complete system level that will impact upon current computer markets.

Some of the microcomputers are being designed to mimic current minicomputers. This obviously has a great saving on software development. However their relative cheapness (and the price will continue to fall) implies that we will seek to use them in new ways. One example is that the "personal computer" idea is viable. Another is that we may see a significant increase in the amount of "intelligence" in an intelligent terminal.

We also note that the cost of computer memory is ever decreasing. Even core RAMs are available at very low costs.

As yet the control and interfacing functions have not been taken over by L.S.I., but it is likely that the remaining technical problems will be solved soon. Osborne points out that microcomputers have evolved through three distinct "generations" inside two years. The $5 computer on a chip is probably not so far off. Hardware and software costs will soon be totally incomparable, with the intangible costing infinitely more than the tangible.

ALTO

A small "personal" computer has been developed, at the Xerox Palo Alto Research Center. This is called ALTO, and is connected into a network based on the ETHER (ALOHA) principle. At Xerox Parc the "ether" is a co-axial cable. The communication principle, developed at Hawaii University, that any machine may transmit into the ether at any time that the net appears silent. If two machines transmit at the same instant then the messages are garbled and lost, but this is catered for by using persistence.

The machine developed by Xerox is similar to the Nova (Data General), with up to 64K of semiconductor memory. The display is a T.V. monitor driven by a main memory video buffer. The display is high resolution (16 x 20 dot character), and the video buffer can take up to 30K. The machine also contains an exchangeable disc drive (0.5M bytes).

Xerox have built some 30 of these machines and connected them in a network. One of the network design philosophies is that the distributed nature of the system should not be hidden (see article in Datamation May 1975).

The estimated cost of an Alto is about $10,000.

The LISP Machine

This is under development in the M.I.T. A.I. Laboratory by Richard Greenblatt and Tom Knight [45, 61]. In the context of this report it is interesting as it is designed as a personal computer ("runs at the same speed at 3 p.m. as 3 a.m.") that will be connected to a network for purposes of sharing and backup. The goals of the system are mainly LISP oriented, but of more general interest are:-

1. Capability to run large programs.
2. Single user stand alone system (connected to a network).
3. Non-prohibitive cost ($70K).

15. Display oriented console interaction available.
16. High reliability and redundancy.
17. Provision to maintain "continuity" over a long period of time.

General features of interest are

(a) that the network is planned as a packet-switched ETHER (ALOHA)-type network. This has no structure to the medium used for transmission, and relies on persistence to work properly. At M.I.T. it is planned to use optical couplers as a form of isolation from the network.

(b) The file system is central, not distributed. The reasons are that backup is difficult, and sharing even more so. The local discs on a personal computer are for temporary storage only.

(c) The display is seen as a crucial part of the system. The technique is to use an ordinary t.v. monitor which is driven from a "video-buffer" in main memory. This technique is already used in the local "Incompatible Timesharing System" (rumoured to be insecure by design). The video buffer is 16K of 16 bit words and forms a bit-pattern of information to be displayed. This thus uses software for character generation and provides a general graphics facility.

(d) The entire philosophy of the system depends upon the cost of memory continuing to fall. At the time of my visit, Greenblatt and Knight had established a 32K x 18 bit core memory (350 ns) at $2350 as "this week's best buy". They point out the necessity of at least parity checking, and preferably error correction with integrated circuit memory.

Bibliography

1. -
Proceedings of the 1975 Symposium on Computer Networks : Trends and Applications.
IEEE Computer Society (June 1975).

2. ABERNATHY, D.M., MANCINO, J.S., PEARSON, C.R. & SWIGER, D.C.
Survey of Design Goals for Operating Systems.
ACM SIGOPS Vol.7 Nos.3,4, Vol.8 No.1 (1973-1974).

3. ABRAMS, M., BLANC, R.P. & COTTON, I.W. (Eds)
Computer Networks : Text and References for a Tutorial.
(Revised Edition).

4. AKKOYUNLU, E., BERNSTEIN, A. & SCHANTZ, R.
Interprocess Communication Facilities for Network Operating Systems.
Computer (June 1974).

5. BELL, C.G. & NEWELL, A.
Computer Structures : Readings and Examples.
McGraw-Hill, New York, 1971.

6. BELL, C.G., GRASON, J. & NEWELL, A.
Designing Computers and Digital Systems using PDP16 Register Transfer Modules.
Digital Press, Digital Equipment Corporation, Maynard, Mass. U.S.A. (1972).

7. BERNSTEIN, G.B.
Forecasting the Ideal Keyboard Terminal.
Proc Compcon 74 Fall (Sept. 1974).

8. BRANSTAD, M.A. & BRANSTAD, D.K.
Computer Security Applications Utilizing Minis.
Proc Compcon 74 Fall (Sept. 1974).

9. BREDT, T.H.
Command Language Processing in Formally Described Operating Systems.
Prepared for IFIP Working Conf. on Command Languages
Lund, Sweden, Aug. 1974.

10. BREDT, T.H. & SAXENA, A.R.
Hierarchical Design Methods for Operating Systems.
Proc Compcon 74 Fall (Sept. 1974).

11. BRINCH HANSEN, P.
The Nucleus of a Multiprogramming System.
Comm. of ACM 13, 4(1970) pp. 238-242.

12. BRINCH HANSEN, P.
Structured Multiprogramming.
Comm. of ACM 15, 7(1972) pp. 574-578.

13. BRINCH HANSEN, P.
Operating System Principles.
Prentice Hall (1973).

14. BRINCH HANSEN, P. & DEAMY, P.
A Structured Operating System.
Californian Inst. of Technology, Pasadena,
Calif. (May 1974).

15. BRINCH HANSEN, P.
A Programming Methodology for Operating System Design.
Proc IFIP 74 Congress (Aug. 1974).

16. BRINCH HANSEN, P.
The Purpose of Concurrent Pascal.
Proc International Conference on Reliable Software
(April, 1975).

17. BROOKS, F.P.
The Mythical Man-Month.
Addison-Wesley Publishing Co.Inc. (1975).

18. CERF, V.
An Assessment of Arpanet Protocols.
INWG No. 70 (July 1974).

19. CERF, V., DALAL, & SUNSHINE
Specification of Internet Transmission Control Program,
Revision (Dec. 1974).

20. COSELL, B., JOHNSON, P., MALMAN, J., SCHANTZ, R.,
SUSSMAN, J., THOMAS, R. & WALDEN, D.
A Non-trivial Example of Computer Resource Sharing.
Submitted to 5th Symposium on Operating Systems
Principles, Austin, Texas (Nov. 1975).

21. COTTON, I.W.
Computer Networks : Capabilities & Limitations
Structural Mechanics Computer Programs,
University Press of Virginia, Charlottesville (1974).

22. CROCKER, S.D.
The National Software Works : A New Method for Providing Software Development Tools using the Arpanet.
Consiglio Nazionale delle Ricerche INSTITUTO DI ELABORAZIONE DELLA INFORMAZIONE Meeting on 20 Years of Computer Science, Pisa (June 1975).

23. DAHL, O.J., DIJKSTRA, E.W. & HOARE, C.A.R.
Structured Programming.
Academic Press 1972.

24. DATE, C.J.
An Introduction to Databases.
Addison Wesley Publishing Corpn.

25. DAVIS, Ruth M.
Computing Networks : A Powerful National Force.
Keynote Speech, Compcon 73 (Feb. 1973).

26. DIJKSTRA, E.
The Structure of the "THE" - Multi-Programming System.
Symposium on Operating System Principles, Gatlinburg, Tenn. (Oct. 1967).

27. ELOVITZ, H.S. & HEITMEYER, C.L.
What is a Computer Network?
NAT. Telecommunications Conference (1974).

28. ENGLAND, D.M.
Capability Concept, Mechanism and Structure in System 250.
Proc. Intern. Workshop on Protection in Operating Systems.
IRIA-LABORIA, Rocquencourt, France, Aug. 1974, pp. 63-82.

29. FABRY, R.S.
Capability-Based Addressing.
Comm. of the ACM, Vol.17 No. 7, July 1974, pp.403-412.

30. FARBER, D.J. & LARSON, K.C.
The Structure of a Distributed Computer System - The Communications System.
Proceedings Symposium on Computer-Communications Networks and Teletraffic, Microwave Research Institute of Polytechnic Institute of Brooklyn pp. 21-27 (April 1972).

31. FARBER, D.J. & LARSON, K.C.
The Structure of a Distributed Computing System - Software.
Symposium on Computer Communications Networks and Teletraffic, Polytechnic Inst. of Brooklyn, (April 1972).

32. FARBER, D.J. & HEINRICH, F.R.
The Structure of a Distributed Computer System - The File System.
Proceedings International Conference on Computer Communications pp.364-370 (Oct. 1972).

33. FARBER, D.J., FELDMAN, J., HEINRICH, F.R., HOPWOOD, M.D., LARSON, K.C., LOOMIS, D.C. & ROWE, L.A.
The Distributed Computing System.
Proceedings of the Seventh Annual IEEE Computer Society International Conference pp. 31-34 (Feb. 1973).

34. FARBER, D.J.
An Overview of Distributed Processing Aims.
Proc. Eighth Annual IEEE Computer Society International Conference (Feb. 1974).

35. FARBER, D.J.
Software Considerations in Distributed Architecture.
Computer, Vol.7, No.3, pp. 31-35 (March 1974).

36. FARBER, D.J.
Distributed Data Bases - An Exploration
(Technical Report). Caine, Farber & Gordon, Inc. Pasadena, California (1974).

37. FARBER, D.J.
The Status of the Distributed Computer System.
Information and Computer Science Dept.
University of California, Irvine (1975).

38. FARMER, W.D. & NEWHALL, E.E.
An Experimental Distributed Switching System to Handle Bursty Computer Traffic.
Proc. ACM Symposium 'Problems in the Optimization of Data Communications Systems'. Pine Mtn. Georgia Oct. 1969.

39. FOOTITT, R.M. & WHITBY-STREVENS, C.
The University of Warwick Modular One Operating System.
Software Practice & Experience Vol.4 (Oct. 1974).

40. FORSDICK, H.C.
A comparison of Two Schemes that Control Multiple Updating of Databases.
Paper submitted to Area Exam Committee : Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (May 1975).

41. FRASER, A.G.
On the Interface between Computers and Data Communications Systems.
Comm. ACM Vol.15 (7) pp. 566-573 (July 1972).

42. FRASER, A.G.
Loops for Data Communications.
Computing Science Technical Report No. 24
Bell Laboratories (Dec. 1974).

43. FRASER, A.G.
SPIDER - A Data Communications Experiment.
Computing Science Technical Report No. 23
Bell Laboratories (Dec. 1974).

44. FULLER, S.H., SIEWOREK, D.P. & SWAN, R.J.
Computer Modules : An Architecture for Large Digital Modules.
Depts. of Comp. Sci. and Electrical Engineering
Carnegie-Mellon Univ. (Oct. 1973) pp. 231-238.

45. GREENBLATT, R.
The LISP Machine.
M.I.T. A.I. Lab. Working Paper 79 (Nov. 1974).

46. GOOS, G. & HARTMANIS,J. (Ed)
Operating Systems - Proceedings of an International Symposium held at Rocquencourt. April 1974.
(Lecture Notes in Computer Science No. 16),
Springer-Verlag, Berlin.

47. GORD, E.P. & HOPWOOD, M.D.
Nonhierarchical Process Structure in a Decentralised Computing Environment.
Technical Report No. 32, Department of Information and Computer Science, University of California, Irvine (June 1973).

48. HABERMANN, A.N.
Synchronization of Communicating Processes.
CACM Vol.15, No. 3, p.171 (1972).

49. HEART, F.E., ORNSTEIN, S.M.,
CROWTHER, W.R. & BARKER, W.B.
A New Minicomputer/Multiprocessor for the ARPA Network.
Proc AFIPS Vol.42 (1973).

50. HOARE, C.A.R.
Theory of Parallel Programming.
Operating System Techniques, Academic Press (1972).

51. HOARE, C.A.R.
A Structured Paging System.
Computer Journal 16, 3(Aug. 1973), pp.209-215.

52. HOARE, C.A.R.
Monitors : An Operating System Structuring Concept.
CACM 17, 10(Oct. 1974), pp.549-557.

53. HOPWOOD, M.D., LOOMIS, D.C. & ROWE, L.A.
The Design of a Distributed Computing System.
Technical Report No. 25, Department of Information and Computer Science, University of California, Irvine (June 1973).

54. HUEN, W.H. & SIEWIOREK, D.P.
Intermodule Protocol for Register Transfer Level Modules : Representation and Analytic Tools.
Proc 2nd Annual Symposium on Computer Architecture (Jan. 1975).

55. IAZEOLLA, G.G.
A Set Theoretic Model of Protection.
Proc Intern. Workshop on Protection in Operating Systems, IRIA-Laboria, Rocquencourt, France, (Aug. 1974), pp.107-120.

56. JENSEN, E.D.
A Distributed Function Computer for Real-time Control.
Proc 2nd Annual Symposium on Computer Architecture (Jan. 1975).

57. JOHNSON, P.R., SCHANTZ, R.E. & THOMAS, R.H.
Interprocess Communication to support Distributed Computing.
Submitted to SIGCOMM-SIGOPS Interface meeting on Interprocess Communications (March 1975).

58. JOHNSON, P.R. & THOMAS, R.H.
The Maintenance of Duplicate Databases.
Bolt, Beranek and Newman Inc., Cambridge, Massachusetts.

59. KATZAN, H.Jr.
Operating Systems : A Pragmatic Approach.
Van Nostrand Reinhold, New York, 1973.

60. KLEINROCK, NAYLOR & OPDERBECK.
A Study of Line Overhead in the ARPANET.
INWG No. 71.

61. KNIGHT, T.
CONS.
M.I.T. A.I. Lab. Working Paper 80
(19:48, 6 Nov. 1974).

62. KNOTT, G.D.
A Proposal for Certain Process Management and Intercommunication Primitives.
OSR Vol.8 No.4 & Vol.9 No.1 (1974-1975).

63. LAMPSON, B.W.
Protection.
Proceedings of 5th Annual Princeton Conference on Information Sciences and Systems. (March 1971).

64. LAMPSON, B.W.
A Note on the Confinement Problem.
Comm. of the ACM Vol.16 No.10 (Oct. 1973),pp.613-615.

65. LAMPSON, B.W.
An Open Operating System for a Single-User Machine.
Proc Intern. Symposium on Operating Systems held at Rocquencourt, France (April 1974), p.208
(Springer-Verlag).

66. LINCOLN, A.J.
A Processing Terminal.
Proc COMPCON Fall 74 (Sept. 1974).

67. LISKOV, B.H.
The Design of the Venus Operating System.
CACM 3(March 1972), pp. 144-149.

68. LISKOV, B.H.
A Note on CLU.
Computation Structures Group Memo 112 (Nov. 1974).

69. LISKOV, B.H. & ZILLES, S.N.
Specification Techniques for Data Abstractions.
IEEE Transactions on Software Engineering
Vol.1, No. 1, p.7 (March 1975).

70. LOOMIS, D.C.
Ring Communication Protocols.
Technical Report 26, Department of Information and
Computer Science, University of California,
Irvine (Jan. 1973).

71. LYNCH, W.C.
An Operating System Designed for a Computer Utility Environment.
Operating Systems Techniques, edited by
C.A.R. Hoare & R. Perrot, Academic Press, London (1972).

72. LYNCH, W.C., LANGNER, J.W. & SCHWARTZ, M.S.
Reliability Experience with CHI/OS.
Proc Intern. Conference on Reliable Software, (April 1975).

73. McKENZIE
Internetwork Host-to-Host Protocol.
INWG No. 74 (1974).

74. METCALFE, R.
Strategies for Inter-Processor Communication in a Distributed Computing System.
Symposium on Computer Communications and
Teletraffic, Polytechnic Institute of Brooklyn (April 1972).

75. METCALFE, R.
Strategies for Operating Systems in Computer Networks.
Proc of the ACM Conference (Aug. 1972).

76. ORGANICK, E.I.
The Multics System : An Examination of its Structure.
Cambridge, Mass. M.I.T. Press, 1972.

77. ORNSTEIN, S.M., CROWTHER, W.R., KRALEY, M.F., BRESSLER, R.D., MICHEL, A. & HEART, F.E.
Pluribus - A Reliable Multiprocessor.
AFIPS - Conf. Proc. Vol.44 (1975).

78. ORNSTEIN, S.M.
Pluribus Document 1 : Overview.
Report No. 2999, Bolt, Beranek & Newman Inc.
Cambridge, Mass. (May 1975).

79. OSBORNE, A. & Associates.
The Value of Micropower.
General Automation, Inc. California (1974).

80. PARNAS, D.L.
On the Criteria to be used in Decomposing Systems into Modules.
CACM 15, 12 (Dec. 1972), pp.1053-1058.

81. PIERCE, J.R.
Network for Block Switching of Data.
BSTJ 51, 6 (July-Aug. 1972).

82. POPEK, G.
Protection Structures.
Computer, Vol.7, No.6 (June 1974).

83. POPEK, G.J. & KLINE, C.S.
A Verifiable Protection System.
Proc. Intern. Conf. on Reliable Software (April 1975).

84. POUZIN, L.
The Cyclades Network - Present State and Development Trends.
INWG No.75 (Nov. 1974).

85. RANDELL, B.
System Structure for Software Fault Tolerance.
Proc. Intern. Conf. on Reliable Software (April 1975).

86. REAMES, C.C. & LIU, M.T.
A Loop Network for Simultaneous Transmission of Variable-length Messages.
Proc 2nd Annual Symposium on Computer Architecture (Jan. 1975).

87. RICHARDS, M.
BCPL : A Tool for Compiler Writing and System Programming.
Proc AFIPS Conf. 35 (1969 SJCC).

88. RITCHIE, D.M. & THOMPSON, K.
The UNIX Time-sharing System.
CACM Vol.17, No.7 (July 1974).

89. ROBINSON, L., LEVITT, K.N., NEWMANN, P.G. & SAXENA, A.R.
On Attaining Reliable Software for a Secure Operating System.
Proc Intern. Conf. on Reliable Software (April 1975).

90. ROWE, LA., HOPWOOD, M.D. & FARBER, D.J.
Software Methods for Achieving Fail-soft Behaviour in the Distributed Computing System.
1973 IEEE Symposium on Computer Software Reliability p.7-11 (April 1973).

91. SALTZER, J.H.
Protection and the Control of Information Sharing in Multics.
CACM Vol.17, No.7 (July 1974).

92. SALTZER, J.H.
Ongoing Research and Development on Information Protection.
ACM-SIGOPS Vol.8, No. 3 (July 1974).

93. SAXENA, A.R. & BREDT, T.H.
A Structured Specification of a Hierarchical Operating System.
Record of 1975 Intern. Conf. on Reliable Software, Los Angeles (April 21-23, 1975).

94. SCHROEDER, M.D. & SALTZER, J.H.
A Hardware Architecture for Implementing Protection Rings.
Comm ACM 157-170 (1972).

95. SHAW, A.C.
The Logical Design of Operating Systems.
Prentice Hall Inc. (1974).

96. SHAW, A.C.
Systems Design and Documentation using Path Descriptions.
Department of Computer Science, Univ. of Washington, Seattle, Washington 98195 (1975).

97. SPIER, M.J.
A Model Implementation for Protective Domains.
Int. J. of Comp. & Inf. Sci., Plenum Press, New York. (June 1973).

98. SPIER, M.J.
A Critical look at the State of our Science.
Operating Systems Review ACM SIGOPS 8.2(1974) pp.9-15.

99. SPIER, M.J., HILL, R.L., STEIN, T.J. & BRICKLIN, D.
The Typeset-10 Message Exchange Facility : A Case Study in Systemic Design.
1974 Sagamore Conf. on Parallel Processing, Univ. of Syracuse, Aug. 20-23 1974. Springer-Verlag, New York, "Lecture Notes in Computer Science" (July 1974).

100. SPIER, M.J., HASTINGS, T.N. & CUTLER, D.N.
A Storage Mapping Technique for the Implementation of Protective Domains.
Software - Practice and Experience, Vol.4 (1974), pp. 215-230.

101. SPIER, M.J.
A Pragmatic Proposal for the Improvement of Program Modularity and Reliability.
Int. J. of Comp. & Inf. Sci., Plenum Press, New York (Oct. 1974).

102. SPIER, M.J.
The Binding Function Model for Access Control Mechanisms.
Digital Equipment Corp., Mass. (April 1975).

103. STELMACH, E.V.
Introduction to Minicomputer Networks.
Digital Equipment Corp., Maynard, Mass, U.S.A.(1974).

104. STEWARD, E.H.
A Loop Transmission System.
IBM Research Triangle Park, Nth. Carolina (1970).

105. STOY, J.E. & STRACHEY, C.
OS6 - An Experimental Operating System for a small Computer.
Computer Journal Vol.15, Nos. 2 and 3.

106. SUTHERLAND, W.R. & THOMAS, R.H.
Natural Communication with Computers Final Report - Volume III - Distributed Computation Research at BBN Oct.1970 to Dec.1970.
BBN Report 2976, Bolt, Beranek & Newman Inc. Cambridge, Mass.

107. THOMAS, R.H.
A Resource Sharing Executive for the ARPAnet.
Proc. National Computer Conference (1973).

108. WALDEN, D.
Network Design Issues.
INWG No. 64.

109. WALTER,K.G., SCHAEN,S.I., OGDEN,W.F., ROUNDS,W.C., SHUMWAY,D.G., SCHAEFFER,D.D., BIBA,K.J., BRADSHAW,F.T., AMES,S.R. & GILLIGAN,J.M.
Structured Specification of a Security Kernel.
Proc. Intern. Conf. on Reliable Software (April 1975).

110. WANN, D.F. & ELLIS R.A.
Conjoined Computer Systems : An Architecture for Laboratory Data Processing and Instrument Control.
Proc.2nd Annual Symposium on Computer Architecture (Jan. 1975).

111. WECKER, S.
A Building Block Approach to Multi-Function Multiple Processor Operating Systems.
AIAA Computer Network Systems Conference, Huntsville, Alabama (April 16-18, 1973).

112. WECKER, S.
Investigations of Multi-Processor Mini-Computer Systems.
Digital Equipment Corp., Mass. (Aug. 1973).

113. WELLER, D.R.
A Loop Communication System for I/O to a small Multi-user Computer.
Proc IEEE Int. Comp. Soc. Conf. (Sept. 1971)

114. WIRTH, N.
The Programming Language PASCAL.
Acta Informatica 1, 1 (1971), pp.35-63.

115. WITHINGTON, F.G.
Fourth Generation Computer Systems.
Proc COMPCON 74 Fall (Sept. 1974).

116. WOOD, D.C.
A Survey of the Capabilities of 8 Packet Switching Networks.
Proc 1975 Symposium on Computer Networks: Trends & Applications (June 1975).

117. WULF, W.A.
The Programming Language ALPHARD.
Comp. Sci. Dept., Carnegie-Mellon Univ., Pittsburgh, Pa. (1974).

118. WULF, W.A., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C.,& POLLACK, F.
HYDRA : The Kernel of a Multiprocessor Operating System.
CACM Vol.17, No.6 (June 1974).

119. WULF, W.A.
Reliable Hardware-Software Architecture.
Proc. Intern. Conf. on Reliable Software (April 1975).