Operating Systems: The Problems Of Performance and Reliability

B. Randell ***University of Newcastle upon Tyne, Computing Laboratory***

The problems of achieving satisfactory levels of system performance and reliability are amongst the most difficult that operating system designers and implementors have to face. This is particularly the case with generic operating systems, i.e., systems intended for use in many different versions, in a wide variety of different environments. The present paper attempts to explore the reasons for these difficulties, and to discuss the interplay between performance and reliability, and, in particular, the problems of achieving high reliability in the presence of hardware failures and software errors.

1. INTRODUCTION

The task of preparing a survey paper on operating systems is daunting, to say the least. A mere catalogue of the numerous existing operating systems, and of research efforts in operating system design, would be inadequate. Moreover the subject, despite an almost frantic rate of development, is still, at least in this author's opinion, in far too disorganised and immature a state for a worthwhile analytical survey and classification to be feasible. There is not even any general agreement as to the meaning of the term "operating system". Paraphrasing Barron [3], who was in fact discussing assemblers:

> *"What is an operating system? Like many other things in computing it is difficult to define precisely. though an experienced programmer will recognise one when he sees it."*

However, the definition given by Creech [4] is adequate for most purposes:

> *"An operating system can be defined to be that part of a computer system which attempts to so allocate and co-ordinate the resources of the system to achieve the optimum performance of that system. The resources involved include processors, peripheral I/O devices, operating system facilities, memory and time. The task is further complicated by the fact the operating system itself must use these resources".*

This definition is better appreciated when one realises the extent of the spectrum of systems that it covers - from say, the LAP6 operating system [27] which contains less than 5000 instructions, and took less than two man years to develop, to IBM's OS/360, which contains several million lines of code, and has taken several man-millennia of effort to develop. (Incidentally, for those of you who are not familiar with LAP6, it is worth mentioning that this system provides a filling system and facilities for program preparation and assembly, and on-line editing, all on a LINC computer with 2048 12-bit words of memory.)

Rather than attempt a general survey. therefore, the present paper concentrates on just two aspects of operating systems, aspects which however are among the most important to the users of such systems, namely performance and reliability. The decision to concentrate on just these two aspects of system behaviour, and to ignore such other important aspects as functional capability, arises from the nature of the problems relating to system performance and reliability that face the operating system designer. Virtually every decision taken by the designers and implementors of an operating system has the potential of having a significant (and in the present state of our knowledge, often unforseeable) effect on the overall performance and reliability of the system. Furthermore, there are systems such as OS/360, which might be described as generic rather than specific in nature, being designed for a whole range of environments and machine configurations. In the case of such systems, the designers and implementors have only an indirect (though by no means small) influence on the levels of performance and reliability that will be achieved in a given installation. Even the criteria by which performance and reliability are judged will vary from installation to installation. Thus many of the problems of achieving acceptable performance and reliability at a particular installation will have to be tackled by the staff of that installation. For these reasons, and others, the problems of performance and reliability of operating systems are particularly difficult ones, and their discussion seems appropriate on an occasion such as the present one.

There is one somewhat tricky problem involved in discussing these two particular aspects of operating systems, and that is their psychological connotations. The topic of system performance can have overtones of mindless preoccupation with easily measurable (though not so easily evaluatable) attributes such as response time and storage utilization; the topic of system reliability can arouse the somewhat puritanical reaction that anything other than absolute reliability (particularly as far as software is concerned) is unacceptable. One of my aims in this talk is to convey my own, somewhat different views of these two topics.

2. SYSTEM PERFORMANCE AND RELIABILITY

The implication of the term "system performance" is that it is a measure of the rate at which a system is capable of doing useful work. In the same vein, "system reliability" can be regarded as a measure of the trustworthiness of the results produced by a system.

Quite crude characterizations of computing system performance (e.g. CPU utilization) and reliability can be adequate for comparing two different computing systems if their functional capabilities are identical or near-identical. However when two computing systems have very different functional capabilities it is in general very difficult to find means of characterising performance and reliability which will facilitate meaningful comparisons of the two systems. Similarly, two different installations can have very different opinions as to the relative importance of the various aspects of a system's performance and reliability. Thus it is hardly surprising that there are no generally accepted standards for measuring performance and reliability, and it is not the intention here to propose any.

Performance and reliability are both "commodities" which are of value to users, and whose "production" will involve the incurrence of costs. Enough different computing systems have been produced and installed that one can attempt to quantify the relationship that holds between system performance (however crudely this might be measured) and cost - Grosch's "law" that performance is proportional to the square of cost, is a well-known example. Such a relationship may tell us more about a certain manufacturer's pricing policies than the realities of his development and manufacturing costs. However it does indicate that there is at least a certain level of understanding amongst customers of the need to assess their performance requirements, and of how the performance that they obtain from a system is, or should be, related to the amount that they paid for it. The situation with regard to system reliability is very different. A really naive user will not realise just how unreliable both the hardware and the software of the computing system that the manufacturer delivers to him might be. Most users will be hard put to quantify the value that they place on obtaining a certain level of reliability, leave alone have any idea how best to allocate the money that they wish to spend in order to obtain this level of reliability.

The specific role of the operating system in all this is rather interesting. It is perhaps not too cynical or misleading to regard the task of the operating system as that of enabling an installation to achieve the inherent performance capabilities, and surpass the inherent reliability capabilities, of the basic hardware. Of course it is not unknown for the amount of resources used by the operating system itself (CPU time, storage, etc.) to be so great as to cause one to question whether it is in fact making a positive contribution to the capacity of the computing system to produce useful work. Similarly, an operating system may contain so many errors that these errors become the dominant factor in the overall reliability of the computing system, rather than hardware failures, whether or not these failures are dealt with adequately by the operating system.

So far no mention has been made of any interactions between performance and reliability, but obviously these exist. Reliability is at least in part bought at the expense of performance - precautionary measures such as attempts to detect the occurrence of errors, multiple recording of data, etc., all use up resources and can impact performance. Conversely, the lack of reliability at some point within a system, can sometimes be dealt with by the system, using fall-back and recovery techniques, so that the problem manifests itself to a user of the system simply as reduced performance. It is when an error goes undetected that the system will produce untrustworthy results, or perhaps no results at all (which can of course also happen even if the error is detected, if the system is not capable of coping with the situation).

Clearly, operating system designers have to be aware of these interactions between performance and reliability, and must attempt rational trade-off decisions. These decisions are very difficult for two reasons. Firstly, in our present state of knowledge, it is often difficult to predict what impact a particular feature, which is intended, say, to improve computing system reliability, will have on either reliability or performance. Secondly, it is very difficult, even when these impacts can be predicted, to judge whether the feature is worthwhile, in view of the lack, discussed earlier, of accepted norms for relating reliability to cost. This second point is more serious when one is designing a generic operating system, intended for use in various versions in many different environments, rather than a special operating system, for a specific environment, for which an assessment of the relative value of performance and reliability can be obtained. Of course the sad reality is that both the performance and reliability achieved by the early versions of most operating systems are far from adequate, and many iterations are usually needed before adequate levels are obtained.

Let me now turn from this attempt to discuss system performance and reliability in general terms, to discuss a particular problem in operating system design which illustrates the confusion which surrounds these two topics.

3. THE DEADLOCK PROBLEM

The problem of deadlock has been achieving ever greater attention during the last few years, and much worthwhile research has been done. However there has been a tendency to regard the problem as solely one of program correctness. and hence system reliability. In fact this problem is a very good example of the interaction between reliability and performance.

Deadlocks arise when two or more processes are allowed to proceed to the point where each reaches a situation where it is waiting for some action by one of the other processes. The standard simple example involves two processes, one having obtained resource A and requested resource B, the other having obtained resource B and requested resource A. Potential sources of deadlock problems are process communication facilities, and shared resources such as storage, I/O devices and operating system services. When viewed as an abstract problem in the theory of operating system design, the usual assumption is that the essence of the problem is to ensure that deadlocks will not occur under any circumstance. With this in mind various authors, sometimes using differing assumptions as to the amount of information that will be available to the system about the future behaviour of processes, have produced various algorithms for the scheduling of processes and the allocation of resources to processes.

Now one can in fact always avoid deadlocks by disallowing any parallel activity, but for performance reasons this is unlikely to be practicable. This fact makes it clear that deadlock avoidance strategies must be assessed not only by the extent to which they succeed in their goal of avoiding all deadlocks, but also by the extent to which they allow multiple activities to proceed in parallel. In fact in many cases it is feasible to provide restart facilities which enable a deadlock situation to be resolved by the rather brutal technique of abandoning one or more processes, and later restarting them. As Needham and Hartley [21] have pointed out, it is then appropriate to regard the task of the operating system designer, with respect to the deadlock problem as being that of finding a suitable trade-off between such factors as cost and effectiveness of a scheduling and allocation algorithm, the frequency with which it fails to avoid deadlocks, and costs of restarting after a deadlock. It would be, to say the very least, aesthetically pleasing if one could satisfy oneself that the best trade-off, in terms of system performance was achieved by algorithms which guaranteed the avoidance of deadlocks, but there is no reason to suppose that this is always or even often the case.

It should however be admitted that there are varying qualities of restart, the ideal restart being one whose occurrence is not noticeable to the users of the system, or at least which does not require any overt action on the part of the users. Where restart facilities fall badly below this ideal it is all too easy to justify an inadequate solution to the deadlock problem by not taking the costs to the users of restarts into account in the trade-off decisions. This is certainly the case where deadlocks can be directly caused by simple (accidental or wilful) actions by a user, as is the case in OS/360 (see Holt [15).

4. SYSTEM PERFORMANCE PROBLEMS

Let us, for the moment, leave aside the problems of reliability, and concentrate on the problems of achieving acceptable performance from an operating system. As mentioned earlier, every design decision, indeed every instruction written by an implementor, is potentially the source of considerable influence on system performance. When designing a system we can have a set of preconceived ideas as to which aspects of the design, and which parts of the coding, will be most critical with respect to performance. However these intuitions can be very wrong. The reasons for this can range all the way from insufficient understanding of underlying principles, to silly coding mistakes.

For example, during recent years much work has been done on so-called "virtual memory" systems, either of the paging type, such as Atlas, or of the segmenting type, such as the B5000. Considerable effort has gone into the design and study of "replacement algorithms", i.e. algorithms for choosing which information to remove from working storage when space is needed in order to bring further information into working storage from backing storage. However it is now becoming clear that the question of which replacement algorithm is used is comparatively unimportant. Much more important, from the point of view of performance, is the problem of avoiding thrashing, the situation in which the system spends virtually all of its efforts transferring information to and fro between working storage and backing storage. The usual cause of thrashing is that too many programs have been allowed to compete for CPU time and hence working storage, so that programs are excessively "space-squeezed" and continually need access to information which is not in working storage.

In certain circumstances it may be regarded as acceptable for one of the duties of the operators to be that of looking out for the symptoms of thrashing, and when necessary instructing the operating system to desist from trying to run one or more currently active jobs. This is the case with, for example, the MCP operating system on the B5000 and its successors [23], and the THE system on the X8 [7] . However in general it is not acceptable to wait until thrashing has become so pronounced that it is eventually noticed by the operator; instead, strategies for avoiding thrashing by controlling the level of multiprogramming of working storage will be included in the operating system, perhaps as an extension to the basic replacement algorithm [24] . These strategies, which can often be simple to the point of naiveté, can have a far greater effect on performance than the basic replacement strategy.

At the other end of the spectrum, horror stories about the massive effects of conceptually trivial coding errors are legion. One particular one that I remember is, it so happens, also concerned with a trivial memory system. Many experiments had been conducted, and many incremental improvements had been made to the storage management strategies. In fact, by far the biggest single performance improvement was due to the eventual (and accidental) discovery of a trivial mistake in the coding of the terminal communication routine.

All this is of course a clear indication of our willingness to design and implement systems of a level of complexity which challenges, and often defeats, our ability to comprehend them. Unfortunately, the analysis of even very simple algorithms, so as to determine what their performance will be, can be extremely laborious, even when gross simplifying assumptions are made as to the statistical properties of the input data. (Such analyses can however produce quite unexpected and illuminating results, despite the simplicity of the algorithms - see Knuth [19]. Luckily, as Knuth also shows, almost equally valuable information can be obtained much more easily by experiment, using trace routines, and routines which record the frequency of execution of the various statements making up an algorithm. What is surprising is that such simple tools are not more commonly used to assist in program development.

Basically similar, but more extensive monitoring facilities, either hardware or software, are now coming into common use for "tuning" operating systems [13]. As I am sure you know, several companies now offer a service which involves spending a day or so monitoring the behaviour of an installation's operating system and standard application programs, and then making recommendations for modifications in order to improve system performance. The quite spectacular improvements which are almost always made are more an indication of the lamentable state of the original system, and of the lack of understanding of the system by the installation staff, than of any great conceptual sophistication in the tools and techniques that these companies use. Clearly this tuning process is a very worthwhile (though not necessarily as intellectually satisfying as it is demanding) method of making improvements in a complex operating system. In fact, I must admit that I think of the task as involving a kind of pathology, being concerned with trying to analyse the obscure causes of unpleasant symptoms in diseased organisms.

This type of tuning is an "after the fact" method of improving the quality of an operating system. It therefore in no sense replaces the activity, which one would like to think was a standard component of any system design and implementation project, of analysing and monitoring first the design, later the partially implemented system, and where necessary, of causing re-design and re-implementation to be undertaken. Ideally one would expect that the designer of an operating system component would be working from a detailed specification, not only of what the component was supposed to do but also of the estimated resources (CPU time, storage space, channel time, etc.) that it was expected to need. Also available to him would be similar estimates relating to those other components with which his component would have to interact. As the system implementation proceeded, the estimates would be checked against what was achieved in practice. It would thus become clear where the redesign of a component was necessary, either because its design had been based on premises that had turned out to be untrue, or in a further attempt to make it conform more closely to the original estimates of its resource utilization.

All this may seem rather utopian at the moment. For example, in some operating systems one is forced to assume that the designers of system components which use disk access routines were as ignorant of the time taken by these routines, as the designers of the disk access routines were of the frequency with which these routines would be used. In many cases, simple back-of-the-envelope type calculations would be sufficient to expose gross disparities in the system design, but comparatively few software designers and implementors are trained or motivated to work this way. This is, I feel, indicative of how far we are from having an occupation which truly merits the title of "software engineering".

However, it is one thing to find out that a partially implemented system should be changed, and quite another to carry out the proposed changes. Comments about the need for structure and modularity, and advocating the use of (decent) high level languages, are all clearly appropriate at this point, even if they do sound like mere motherhood [22]. Perhaps as important, whenever the number of designers and implementors warrants it, are automated or semi-automated techniques for policing and coordinating their work. A simple example would be facilities for maintaining up to date, and accurate, lists of which system modules use, or modify, which common data structures. (Hopkins [16] has given an all too graphic account of what has happened in OS/360 due to a vast number of inadequately coordinated attempts to improve the performance of individual modules and groups of modules.)

Let me conclude this discussion on system design problems by returning briefly to my earlier point about the dangers of over-ambitious design goals. It is very noticeable that some of the more successful operating systems, from a practical point of view, are those whose designers have had a clear idea of the intended environment, and have resisted the temptation to attempt a giant leap in all directions at once, so to speak, by implementing the most sophisticated and general system that they could envisage. Just two examples of this are the Cambridge Multi-Access System [28] and APL/360 19] . In the Cambridge System, it was decided that the main uses of the terminals would be for file editing and job submission, and that these facilities could be, if carefully designed, provided quite economically. On the other hand, the more general ability to interact from a terminal with any user-supplied program, which it was felt would place a heavy load on system resources, was provided only as a special mode of use, called "expensive mode", the use of which was very carefully rationed. The designers of the APL/360 system were very conscious of the need to avoid excessive information transfer between working storage and backing storage, which was a flailing-arm disk. Such information transfers could be caused by switching between users, and by user commands to load and save "workspaces" containing programs and data. Switching between users was performed under the control of the system, and was calculated to be manageable - the worry was the load and save commands. By the simple expedient of allowing such commands only from a terminal, not from within APL programs, the maximum rate at which such commands can be given has been severely limited. It would be difficult to ascertain how much this decision has contributed to APL/360's undoubted efficiency, but certainly a potentially difficult bottleneck has thus been completely avoided.

5. SYSTEM RELIABILITY

As users have become (sometimes unintentionally) more dependent on their computer systems - often far more dependent than the quality of either the hardware or the software would justify - the subject of system reliability has become ever more important. However, as discussed earlier, it is no use obtaining reliability unless it is matched by adequate performance. A result that is "guaranteed" correct, produced after all need for it has passed, may well be less valuable than a timely result which has some (hopefully small, and known) probability of being incorrect.

Needless to say, a complex system will not in general be designed to produce a single result, but rather a whole set of results, to each of which a different reliability requirement might be attached. (For example, in a system which maintains a large inventory file, inserting an incorrect value into the file may be regarded as much worse than occasionally failing to answer, or answering incorrectly, requests for information from the file). In such circumstances it is only sensible to try to design the system in such a way that its more commonly occurring faults at least do not affect the more crucial of the results that the system is producing, even though they might affect the overall reliability (and performance) - the terms "graceful degradation" and "fail-soft" are the currently fashionable ones for characterising computing systems that are designed in this way. In fact a look at one of the most obviously successful projects that involved obtaining ultra-high reliability from a complex software system, the Project Apollo Ground System [2], is most instructive. One of the most striking features is the care which has been taken, in the design of the environment which surrounds the system, to minimise the extent to which reliance is put on the correct and continuous functioning of the system.

A typical dictionary definition of system is "a whole composed of components in orderly arrangement according to some scheme or plan". From this definition (and resisting the temptation to question whether the phrase "orderly arrangement" is fully applicable to complex computing systems) one is led to the view that the task of achieving reliability of a system can be split up into:

(i) that of making sure that the components of which the system is constructed are reliable;

(ii) that of coping with the consequences of any failures to achieve (i) completely.

Let us take as the level of components of interest to us in a computing system such major hardware modules as processors, memories, channels, I/O devices. etc., and the major software modules which make up the operating system, and consider what reliability one might reasonably expect from such components.

6. HARDWARE RELIABILITY

Much progress has been made in achieving ultrahigh reliability from hardware modules which are essentially electronic such as processors and memories; Darton [5], for example, has reported on a small demonstration computer which is to all intents and purposes absolutely reliable. (Any single failure can be detected, the offending circuit board identified, and replaced, all without interrupting the system - the time to replace a board is infinitesimal compared to the mean time between failures.) However, as I am sure you all know, much of this progress has yet to be reflected in the average present-day computing installation.

The situation is worse with electromechanical devices, where the levels of redundancy needed to achieve comparable reliability are much higher. Thus the "hidden" ninth spindle on an IBM 2314 disk drive, kept as a spare, although of value in increasing the probability of there being eight spindles in working order, does not prevent loss of data caused by a head crash. To do this would require duplication of the entire set of eight spindles, and that all data be automatically recorded in duplicate. It is unlikely that this would be regarded as the most effective way of utilizing the eight extra spindles.

In summary therefore, there seems little chance that the operating system designer will be able to avoid taking at least some of the responsibility for coping with the consequences of hardware failures. It is, however, I think reasonable to expect improved hardware facilities for reporting and identifying modules that are in error, for system reconfiguration, etc., such as in the Burroughs D825 [l] and the IBM 9020 [17], to become more widespread.

7. SOFTWARE RELIABILITY

A common view of software reliability is that it is achieved solely by ensuring that the software is correct, i.e. is free of bugs. Software bugs are seen as the equivalent of design errors in hardware, with there being no equivalent to the failure that can occur after all the design errors have been removed, such as are caused by component ageing. this view is somewhat simplistic - for a start. the distinction between hardware design errors and later hardware failures can be somewhat arbitrary. However what is more to the point is that in today's cruel world it is rarely possible to wait until all the bugs have been removed from a complex software system, before it is used to provide service. (For that matter, it is not uncommon for blatant hardware design errors to be found many years after installation of a complex computing system). Indeed there are many who would deny the possibility of a large software system ever reaching a bug-free state. Certainly the current statistical evidence is on their side - it was recently stated that each release of OS/360, which is (one hopes) an extreme case, has on average over one thousand distinct errors reported in it.

It is worth examining what we mean by the term "correctness". Needless to say, the results produced by a system can only be "correct" with respect to some criterion. One would like to assume that such a criterion would be part of the detailed specification that was used to guide the design and implementation of a system. However, for other than very simple systems, such specifications are unlikely to be accurate or complete. Rather, they are often little more than an initial bargaining offer, subject to renegotiation as the system implementation proceeds and the designers and their customers start getting detailed feedback. Naturally, to a user, the fact that a system is correct with respect to some inadequate or obsolete specification will be irrelevant - to him it will be, in essence, incorrect.

It is against this background that the current research on the topic of program correctness should be assessed. Much of this work derives from that of Floyd [10], who proposed the use of automated theorem proving techniques to check the consistency of a program with programmer-supplied formal assertions about the relationships which should hold amongst the values of the variables at various stages during the execution of a program. This has in fact been done by King [18], but king's work, impressive though it is, makes it clear that, at the present state of development, even quite simple programs can tax the abilities of automated theorem proving techniques. Of direct importance to the problems of system reliability is the work of Dijkstra and his colleagues on the T.H.E. system [7] who took as their goal the task of satisfying themselves, a priori, as to the "correctness" of their design for a multiprogramming system. The degree to which they achieved their goal is indeed remarkable - however the techniques of system structuring that they developed are, I believe, of great importance in themselves irrespective of whether they are used for facilitating the construction of correctness "proofs".

All of this is not intended to downgrade the importance of efforts to ensure that bugs are located and removed from software, or of research efforts aimed at improving our ability to specify accurately the intended behaviour of software, and to construct correct software and at providing rigorous proofs of software correctness. Rather, the point is that for the foreseeable future complex computing systems must, I believe, contain effective provisions for coping with software bugs, as well as hardware failures, if such systems are to achieve really high reliability.

8. COPING WITH UNRELIABLE COMPONENTS

One obvious distinction can be made between the problems of coping with unreliable hardware and unreliable software. This is that one would expect estimates of the probability of the occurrence of the various kinds of hardware failures, based on experimental trials of prototype hardware, to be available. In contrast, predictions as to what software errors will be made must be predictions of the frailty of human, rather than of hardware. In fact the situation with regard to coping with software errors is somewhat paradoxical: in order to know exactly what precautions to take, one would like to know what errors are likely to be made. However, if one really knew this, one would take extra care in the preparation of the relevant parts of the program, in order to avoid making the errors! In practice this distinction between hardware and software is less important than one might imagine for various reasons:

(i) in any cases, one has to try and cope with an error situation without knowing whether its underlying cause is a program bug or a hardware failure - indeed in some cases one may never find out:

(ii) many of the precautions that one takes because of possible software errors are quite general and not dependent on the specific type, or the location, of the error:

(iii) it is not always wise to rely too heavily on the accuracy of the hardware failure rate estimates. The features that are built into an operating system in an effort to cope with error situations can de divided into:

(i) preparations for the possibility of errors:

(ii) error detection facilities:

(iii) error recovery facilities.

The first category included techniques such as multiple recording of important information e.g. file directories, the preparation of fall-back and restart facilities such as dumps, audit trails etc. (see for example Fraser [11] and the provision and use of protection mechanisms. This latter topic is receiving much attention at the moment but is I am sure still at a very early stage of development. One approach involving the idea of "capabilities" is due to Dennis and van Horn[6] and has been developed by Lampson[20] and by Yngve and Fabry a description of whose work has been given by Wilkes [28]. The idea is that a given process (which might be part of the activity of the operating system, or arise from the execution of user's program) should have at any given moment a list of "capabilities" associated with it which indicate and delineate what the process is permitted to do. The intention is that each process be given the minimum set of capabilities that it needs in order to perform its function. If any errors are encountered the capability mechanism limits their possible consequences and increases their chances of being detected. The topic of protection mechanisms is closely related to that of addressing structures - if a process cannot obtain the address of an object, even accidentally it cannot harm the object. My own view is that this relationship has yet to be fully exploited and that future protection mechanisms may well owe as much to work on addressing structures such as that of the B6500 (see Hauck and Dent [14]) as to the work on the capability concept.

In general an operating system in addition to containing its own error detection mechanisms should be capable of dealing with reports it receives of errors that have been detected (but which cannot be dealt with) within its components and those that have shall we say escaped the vigilance of the system and have been detected outside the system perhaps by the operators. (In fact, this classification can be applied more finely at every discernible level in the system.) Its own error detection mechanisms will ideally only be needed for errors within the system itself - in practice they might regrettably have to be used for attempts at detecting errors that occur inside components even hardware components such as processors and memory. However the aim should be that all components have a reliable mechanism for error detection if not error correction.

All error detection is based on the provision of redundant information whose consistency can be checked. The idea of hardware and data redundancy is well-known, but program redundancy is more novel. Clearly program redundancy is something quite different from having multiple identical copies of a program - rather it involves redundancy in the specification of the intended process. (In fact Floyd's work on program correctness proofs described earlier uses just such redundancy but the consistency checks are applied before, rather than during execution.) Examples of program redundancy some involving redundancy already implicit in the data others involving the deliberate introduction of explicitly redundant data include:

(i) positive checking - at a multi-way branch where the path to be taken depends on the value of a variable each path is taken only as the result of a positive check leaving an extra error path to be taken if none of these checks apply:

(ii) sum checks - a typical example is to maintain a sum check on a table, adjusted with each change to a table entry and checked at appropriate intervals:

(iii) bi-directional links - even where a uni-directional linked list would suffice bi-directional links are used and checks are made that an item which points at another item is itself pointed at by that item:

(iv) dog tags - a set of unique names are generated and one is attached for example to each pay of information. A process which accesses a page will do so by using its address. However the process will also have a copy of the dog tag which will be checked against the dog tag kept with the page wherever it is stored.

This list is clearly not exhaustive - techniques such as these form part of the folklore (but not with few exception such as Watson[26] the literature) of operating system design and are probably reinvented almost daily. In fact the idea of dog tags way used Eckert [8] and the idea of using assertions for manually checking programs can be found in the writing of both von Neumann [12] and Turing [25]!

The sorts of actions which I consider as part of error recovery include determination of the extent of the damage reporting of the error and to whatever extent possible the repairing of the damage so that the system can continue to provide service. Determination of the extent of the damage can be explicit from a knowledge of where the error occurred (this of course is where the protection mechanism is exceedingly useful providing that it itself is not involved in the error) or explicit by tentative exploration during which facilities are exercised and consistency checks are made on data. Error repair usually involves such acts as file recovery the re-establishment of system data structures etc. and in the case of identifiable hardware failure perhaps retrying the action which caused failure or if necessary reconfiguring the system to isolate the failed component. In the case of software error it will often be the case that all one can do is to make sure that those services which are not affected by the error are resumed with as little delay as possible.

Perhaps it is appropriate to conclude this topic by noting that these problems of error recovery are amongst the most tricky (particularly when one tries as one should to allow for further errors occurring during the recovery process itself) and the most important of the whole design. Indeed one might suggest that error recovery should be amongst the first problems that are treated during the system design process rather than as so often happens one of the last.

9. CONCLUSIONS

I have attempted to give you my own personal perspective on two problem areas in operating system design. These particular problems interest me because they are what I think of as "system" problem as opposed to "component" problems. As such their main enemy is complexity - the complexity that we are all so willing to build into our systems. If a system is simple enough then performance and reliability are unlikely to be too much of a problem. Complex highly complex systems have been created and have been made to work but at a cost of what a decade ago would have been unbelievable amounts of programming time and effort. However there will be no easy solutions to the problems of performance and reliability unless and until we have learnt how to reduce and mater this complexity.

ACKNOWLEDGEMENTS

The preparation of this paper have greatly aided by the discussions that I have had with many friends and colleagues in particular Jim Eve, Bob Floyd, Jim Horning, Butler Lampson, Hugh Lauer and Bill Lynch

REFERENCES

[1] J.P. Anderson, S.A. Hoffman. J. Shifman and R.J. Williams, D825 - a multiple-computer system for command control, in *AFIPS Conf. Proc.*. Vol.22, 1962 FJCC (Spartan Books, Washington D.C. 1962) 86-96

[2] J. Aron, Apollo programming support, in *Software engineering techniques*, Eds. J.N. Buxton and B. Randell, NATO Science Committee. Brussels (1970) 43-48

[3] D.W. Barron, *Assemblers and loaders* (MacDonald, London, 1969)

[4] B.A. Creech. Implementation of operating systems. *IEEE International Convention Digest* (March 1970)

[5] K.S. Darton. The Dependable process computer. *Electrical Review* 186.6 (6 Feb. 1970) 207-209,

[6] J.B. Dennis and E.C. Van Horn. Programming semantics for multiprogrammed computations. *Comm. ACM* 9.3 (1966) 143-155

[7] E.W. Dijkstra. The structure of the 'THE' - multiprogramming system *Comm. ACM* 11.5 (1968) 341-346

[8] J.P. Eckert. Reliability and checking. *Theory and techniques for design of electronic digital computers*. Ed. C.C. Chambers, Moore School of Electrical Engineering. Univ of Pennsylvania. Philadelphia. Pa. (19480 35 1-35.16.

[9] A.D.Falkoff and K.E. Iverson. The APL 360 terminal system. *Interactive systems for experimental applied mathematics*. E. K. Klerer and J. Reinfields (Academic Press. New York. 1968) 22-37

[10] R.W. Floyd Assigning meanings to programs. in *Proc. Symposia in Applied Mathematics*. Vol. 19 American Mathematical Society (1967). 19-32

[11] A.G. Fraser, Integrity of a mass storage filing system. *Comp. J.*12. 1 (1969) 1-5

[12] H.H. Goldstien and J. von Neumann. *Planning and coding problems for an electronic computing instrument*. Part 2. Vol. 1 Institute for Advanced Study, Princeton, J.J. (1947) (Reprinted in: John von Neumann: Collected Works. Vol. 5. Ed. A.A. Taub (Pergamon. Oxford, 1963) 80-151

[13] C.C. Gotlieb and G.H. MacEwen System evaluation tools, in: *Software engineering techniques* Eds. J.N. Buxton and B. Randell. NATO Science Committee. Brussels (1970) 93-99

[14] E.A. Hauck and B.A. Dent, Burroughs' B6500/B7500 stack mechanism, in: *AFIPS Conf. Proc.*. Vol. 32 1968 SJCC (Thompson, Washington, D.C. 1968) 245-251

[15] R.C. Holt, *On deadlock in computer systems*, PhD Thesis. Cornell Univ. Ithaca, N.Y. (Jan. 1971)

[16] M.E. Hopkins. Computer aided software design, in *Software engineering techniques*. Eds. J.N. Buxton and B. Randell. NATO Science Committed. Brussels (1970) 99-101

[17] J. Keeley et al. An application-oriented multiprocessing system. *IBM System J*.6.2 (1967) 78-132

[18] J.C. King *A Program verifier*. PhD Thesis Carnegie-Mellon Univ. Pittsburg. Pa (Sept. 1969)

[19] D.E. Knuth. The analysis of algorithms in: *The teaching of programming at university level*. Ed. B. Shaw. Computing Laboratory, The University of Newcastle upon Tyne (1971) 49-62

[20] B.W. Lampson. Dynamic protecti9n structures, in: *AFIPS Conf. Proc*. Vol. 35 1969 FJCC (AFPIS Press, Montvale, N.J. 1969) 27-28

[21] R.M. Needham and D.F. Hartley. Theory and practice in operating system design, in: *Proc. 2nd ACM Symp. on operating system principles*, Princeton Univ., Princeton, N.J. (Oct. 20-22 1969) 8-12

[22] P.G. Neumann. The role of motherhood in the pop art of system programming, in: Proc. *2nd ACM Symp. on operating system principles*, Princeton Univ. Princeton, J.J. (Oct. 20-22 1969) 14-18

[23] C. Oliphant. Operating system for the B5000. *Datamation* 10.5 (1964) 42-45

[24] A.J. Shils. *The load leveller*. Report RC 2233 IBM Research Center. Yorktown Heights, N.J. (Oct. 7 1968)

[25] A. Thuring. Checking a large routine. *Report on a conf. on high speed automatic calculating machines*. Univ. Mathematical Laboratory. Cambridge (June 22-25 1949) 67-68

[26] R.W. Watson, *Timesharing system design concepts* (McGraw-Hill, New York, N.Y. 1970)

[27] M.A. Wilkes, Conversational access to a 2048-word machine. *Comm. ACM* 13.7 (1970) 407-414.

[28] M.V. Wilkes. *Time-sharing computer systems* (MacDonald, London, 1968)