OPERATING SYSTEMS IN A CHANGING WORLD[1]

by

Maurice Wilkes

Olivetti Research, Cambridge, England

The principles by which processes are queued and managed, laid down at the early SOSP meetings, still remain valid. Since then new insights have led to important advances. For example, insights into the user interface have led to the development of windows and menus. At a different level, authentication—the process by which the system can become sure that a person demanding resources is the person he or she claims to be—has become well understood. Authentication is the necessary basis on which resource allocation and billing, as well as security can based.

I shall begin with a survey of some of the past lessons that have been learned and then go on to discuss present research directions.

Process Management

Much early effort was devoted to understanding cooperating processes and the design of synchronization primitives. This was successful and the days are now long over when an operating system would stop every few hours, or every few days, because of some arcane synchronizing error that was hard to pinpoint. While there is always scope for research, synchronization is no longer a principal preoccupation of operating systems specialists.

In the 1960s and 1970s, a popular area of research was the analysis of operating systems in terms of queueing theory. Each resource—files, memory, processor, etc—had a queue associated with it. This was in days of limited I/O bandwidths and minimal resources generally, and it was natural to view the passage of tasks through an operating system as being like the movement of cars in Manhattan on a busy day, with queues at every intersection.

In my view, queueing theory failed to produce any results of particular value.

[1]Text of an invited address delivered at SOSP14, December 1993.

This was partly because of its inadequacy as a mathematical discipline. In order to get results it was usually necessary to assume an exponential probability distribution; this was particularly unfortunate when analyzing a time-sharing system, since tasks generated by users tend to fall into two groups, short and long, giving rise to a bimodal distribution. Nowadays, we have more resources and more bandwidth and we like to think of tasks moving through a system like cars moving along a freeway.

An important development of recent years is the concept of threads. Threads are a development of the long-standing concept of lightweight processes, that is, processes which can be rapidly created and destroyed without the overheads associated with regular processes. Lightweight processes are important to system designers since they make it possible to use processes freely in the structuring of a system. Since threads run in the same address space as the process that spawns them, they can communicate with low overheads.

Interprocess communication may be effected in a time-sharing system either by passing messages or by making use of shared memory. Lauer and Needham showed that the two methods are essentially equivalent, and that the choice between them is largely a matter of convenience. Message passing may also be used to provide communication between processes running in separate computers, for example, between a user process running in a workstation and one running in a file server. Appreciation of this fact led to vigorous discussion on the subject of remote procedure calls. This discussion—and a parallel discussion of other communication primitives—drew attention to the ease with which bandwidth can be lost by inefficient implementation.

File and Memory Management

Early time-sharing systems had very limited high speed memories, and much effort was perforce devoted to arriving at a satisfactory paging policy. A particular problem was the avoidance of thrashing, a situation in which one process would load pages only to have them overwritten by another process before it could make any effective use of them. The working set model developed by Denning proved fruitful in understanding this phenomenon and in guiding the development of satisfactory paging algorithms. As high speed memories got larger, paging problems became less acute, and there was less need for fine-tuning of paging algorithms. However, since that time, high

speed memories have increased in speed by a very large factor, but disc latency has remained much the same. The result is that the solutions arrived at earlier have failed to scale. I shall return to this subject when I come to discuss current research directions.

An innovation introduced in MULTICS at MIT was the system of attaching files to the virtual memory instead of reading from and writing to them in the conventional manner; such files are said to be mapped on to the virtual memory. This creates a different way of life for the programmer and its merit has never been clearly established.

It is sometimes claimed that attachment of files reduces copying overheads, but it is hard to see that there is much to choose between the two systems in this respect. It is true that the contents of an attached file can be modified in place, without it being necessary to copy the whole file. However, if there is a requirement for the original form of the file to be preserved—for example, for restarting purposes—then a copy of it must be made before it is attached.

The problem is not removed by implementing a system—for example, copy-on-write—in which changes to a file are accumulated in temporary storage and not written back into the file until a late stage, since the changes do eventually come to be written back, and a programmer who wishes to preserve the original contents of a file is still under the necessity of making an advance copy.

A similar problem occurs in database systems in which fixed length records are updated in place. Here the problem is usually met by making the committing of changes an operation that is explicitly called for by the user of the database.

My impression is that there is now little interest now in providing file attachment as a feature for users of time-sharing systems. It is, in any case, not appropriate when a file server is being used.

Operating systems brought with them the need for memory protection to prevent user programs from interfering with the system and with each other. In response to this need, hardware designers provided two modes of operation, namely, user mode and privileged mode. It was widely felt that something more was needed and the MULTICS system pioneered rings or levels of pro-

tection. As a process moved through the rings in an inward direction it acquired access to more segments of memory. This lead was followed by various vendors who provided rings of protection as a feature of their hardware.

However, it eventually became clear that the hierarchical protection that rings provided did not closely match the requirements of the system programmer and gave little or no improvement on the simple system of having two modes only. Rings of protection lent themselves to efficient implementation in hardware, but there was little else to be said for them.

The attractiveness of fine-grained protection remained, even after it was seen that rings of protection did not provide the answer, and led to much work being done on capability systems in which the capabilities were bit patterns recognized by the hardware. This again proved a blind alley and all modern processors use the simple system of two modes, user mode and privileged mode.

Hardware support for capabilities was essential if the resulting system was to run fast enough to have any chance of being acceptable in practice. I could not, therefore, see any point in work which was done on capability systems implemented entirely in software. I am not here referring to systems in which capabilities in the form of sparse bit patterns are passed from one computer to another across a network.

UNIX and its History

UNIX has had a complex and strange history. Originating as a system for a PDP-7 with a small address space and used in a research organization, it was developed to run on large minicomputers in a networked environment. It finally emerged as a full scale competitor to the major proprietary systems, with the advantage—unique among major systems—of being machine independent. UNIX appealed mostly to users in science and engineering and had, by the mid 1980s, attained a significant presence in that sector.

The early personal workstations were designed by minicomputer engineers who were familiar with UNIX. It was natural, therefore, that when they required an operating system, they should look favorably on UNIX. Indeed, they had little option, since there was no other machine independent operating system obviously available. The availability of UNIX, as an operating

system independent of the processor instruction set, was a major factor in enabling workstations with RISC instruction sets to be accepted by the market. UNIX has stood up well to the challenge, and at the present time it is in universal use for personal workstations.

Unfortunately, there were two competing brands of UNIX and not all personal workstations had the same brand. This situation has become worse, not better, as time has gone on. This is because some users were not happy with either of the brands originally available, and some companies did not find it easy to live with the proprietary problems that they both entailed. In consequence, a number of attempts were made to produce a version of UNIX that would be free of proprietary constraints and would achieve dominance, either because it was manifestly superior to any existing variant or because it had overwhelmingly strong industrial backing, None of these attempts have proved successful. All the systems that have resulted have their adherents, with the result that we now have more brands than ever before. If it turns out, in the next few years, that UNIX has run its course, it will be largely because it has been smothered by its variants.

Microsoft have recently introduced a new operating system known as NT, which is obviously targeted at the UNIX market. Like UNIX it is a general purpose time-sharing system that can be run on a workstation, but it would also be entirely at home in the role of providing a departmental time-sharing system on a large VAX or other minicomputer. It implements threads and in one version—Windows NT—it provides a modern interface to the user. Windows NT pays special attention to user authentication and security, initially to the level of Class C2. The user who creates an object generally becomes its owner and can specify by means of an access control list how much use other users may make of it. There is a comprehensive system of resource accounting which enables each user to be assigned quotas for memory usage, number of active objects, and processor time. These are all features appropriate to a time-sharing system serving a large and diverse population of users. It is obviously the belief of the designers of NT that such systems will continue to play a major role in the computing world. As a competitor to UNIX a strong appeal of NT will be that it is a new system entirely free from variants, and that moreover it has every prospect of remaining so.

Operating Systems for Personal Workstations

It appears from what I have just said that present thinking—and practice—is that the appropriate operating system for a high end personal workstation differs in no respect from the traditional operating system developed in the very different environment of large computers and time-sharing. I can see arguments for believing that this is a natural and proper state of affairs. A single user workstation needs multitasking both so that the user may run background tasks and so that the system may accept information coming in through a network. It can be argued that virtually all of the process handling facilities, provided in a general purpose time-sharing system, are needed for these purposes and that virtually nothing could be saved by taking account of the fact that there is a single user only. Similarly, some protection of files must be provided in order to protect the user against himself or against accident and it can be argued, though here I think less convincingly, that a fully certified security system provides what is needed without either getting in the user's way or adding to the overheads of the system. I shall return to this subject later.

A Range of Operating Systems

The powerful personal workstations that I have been considering constitute only a small part of the total range of desk top computers. There are also PCs, laptops, and even smaller computers; these exist in much larger numbers than do large workstations. Most of the low end computers—all of those that are IBM compatible—run MSDOS while the high end run UNIX. Since from the hardware point of view the computers form a continuous range this makes an awkward division.

Historically this division is due to the fact that workstations were developed by minicomputer engineers, who as I have explained, turned naturally to UNIX as a system known to them. In any case they could hardly have adopted MSDOS from the PC world, since it would have lacked features essential to the high end user—for example file protection and multithreading—and these users would have been infuriated by the restriction to eight letter file names. On the other hand it was this very simplicity and lack of frills that constituted the real strength of MSDOS in its own proper sphere.

In a column I wrote for the Communications of the ACM I remarked that it would be too much to expect that a single operating system could be found that would be suitable both for powerful workstations and also for laptops, and even smaller computers. I did suggest, however, that as a long term aim we might hope for a range of operating systems that were, in some sense, compatible or at least friendly to one another. The family relationship between the various operating systems would be close enough to permit the migration up and down the range of files and programs.

There does not appear to be any insuperable difficulty in providing for the migration of files. The full specification of a filing system might include file protection and the retention of a full updating history. In a lower level system these features might not be provided, but that would not prevent the files themselves from being copied; it would simply mean that some attributes would be lost. Similarly, when a file was being copied in an upward direction, default values for some attributes might have to be supplied.

Easy movement of programs in the downward direction would make life easier for application programmers. They would be able to develop applications on a workstation equipped with a full operating system and then package them to run on smaller computers.

In this connection, I might remark that system programmers need more and more to visualize the requirements of application programmers. The idea, originating at Xerox PARC, that they would do good work provided that they were users of the systems they created was a fruitful one in its day, but something more is now required.

File Servers and Other Servers

Along with workstations and ethernets came the requirement for servers of various kinds—file servers, print servers, network interface servers, and perhaps compute servers. An easy way to make a server is to take a workstation running UNIX and equip it with the necessary software. However, a full operating system is unnecessary. This along with the need to make servers highly secure and to maximize throughput—or rather to secure a good compromise between throughput and response time—has led to the design of servers becoming a specialized subject. This is particularly noticeable in the case of file servers; here effective and reliable error recovery is of the highest

importance and so is file security. Performance is optimized by making use of sophisticated software cacheing systems.

File security, and indeed security generally, presents great problems to the designer of a traditional time-sharing system serving many users drawn from diverse, and possibly competing, organizations. In a file server it is relatively straight forward. This is because users cannot run arbitrary programs in a file server in the way they can on a general purpose time-sharing system. All they can do is to invoke one of a small set of services offered by the file server.

I do not question that a secure time-sharing system can be designed. I would even agree that it could be implemented in such a way that that the probability of another user or a penetrator obtaining unauthorized access to a user's files would be negligibly small, even taking account of major system malfunction. However, I believe that such a system would necessarily be implemented on safety-first principles, and would run intolerably slowly. The optimizations necessary to make the system acceptable would carry with them many risks of introducing security loop holes, and a detailed certification of the implementation would be necessary. However eminent the authority ultimately responsible for the certification, I would be extremely reluctant to entrust any sensitive information to such a time-sharing system. I am, however, aware that I am here expressing a personal view and that there are others do not feel the same way.

However, I believe that the above problem is likely to lose its importance, since we may soon expect to have, instead of large single computers, groups of workstations and servers linked by a LAN and connected to the outside world through one or more gateways. In my view, the unit of security should then be the group as a whole. The responsibility for security against espionage or sabotage would rest with the designer of the software for the gateways, rather than with the designers of the operating systems for the individual workstations.

RESEARCH DIRECTIONS

Structure of an Operating System

In earlier times designers were often attracted by a layered model, often associating the layers of the model with varying degrees of memory protection. Such models if rigidly adhered to tended to lead to systems that ran very slowly. I remember one operating system constructed in this way that was so slow that an extra year of development time was needed to make it acceptably fast.

A widely expressed ideal is that an operating system should consist of a small kernel surrounded by routines implementing the various services that the operating system provides. The intention is that the kernel should contain all the code that needs to be certified as safe to run in privileged mode and that the peripheral routines should run in user mode. When such a system is implemented, it is found that frequent passage in and out of the kernel is necessary and that the overheads involved in doing this make the system run slowly. There is thus an irresistible pressure to put more and more in the kernel. I am driven reluctantly to the conclusion that the small kernel approach to the structuring of an operating system is misguided. I would like to be proved wrong on this.

Recent developments in programming languages have led to modular or object-oriented models. These free programming languages from the limitations of hierarchical scope rules, and as such I regard them as representing a major advance. It has long been recognized that an operating system is not a monolithic program, but is made up of routines for scheduling, handling interrupts, paging, etc. It would lend itself well to modular or object-oriented design and I am aware that such designs are being worked on. I would like to see public identification of the modules and accepted definitions of the interfaces between them.

A modular or object-oriented operating system would be capable of having plugged into it modules implementing a variety of different policies. It would form a valuable tool for the type of experimental research that I shall mention in the section on software performance.

Paging

I remarked above that high speed memories have increased in speed by a very large factor, but that disc latency has remained much the same. We can no longer rely on disc transfers being overlapped with useful computation. The result is that users of top end workstations are now demanding enough high speed memory to hold their long term working set, so that disc transfers will be reduced in number. This may be a workable solution at the top end, but will hardly solve the problem for users of more modest equipment. We need to look again at policies other than demand paging, even perhaps roll-in/roll-out policies that take advantage of the high serial transfer rates that can be sustained when blocks of contiguous words are transferred

The problem is to be seen against the background of probable future developments in memory systems. Up until now, memories have been growing larger but not faster. We may expect to see some emphasis put on the development of faster memory systems for use in top end workstations. Available approaches are via: 1. chips optimized for speed rather than capacity; 2. chips designed for streaming; 3. the exploitation, for cacheing purposes, of the internal line registers to be found in memory chips. The last two approaches involve software as well as hardware.

Conventional data caches support data locality in one dimension only. For example, if a matrix is stored by rows, then elements in the same row or in adjacent rows may find their way into the cache, but elements that are in adjacent columns will not do so. In high energy physics a numerical algorithm is as likely to require data words separated by a largish increment in the address space (referred to as a stride) as to require data words close together in the address space. Innovators are beginning to turn their thoughts to new forms of cache (if that word can properly be used to describe them) which support locality in terms of strides of any length from one upwards. Any solution to this problem is likely to come from the collaboration of computer architects, compiler writers, and operating system specialists.

Focus on Software Performance

In the parallel field of processor design, recent years have seen a great emphasis placed on the quantitative evaluation of processor performance. This has been done by running bench marks on processors or on simulated versions

of them. Simulation has been important because it has enabled comparative trials to be made of competing architectures—and of rival features in the same architecture—without its being necessary to implement the processors in silicon. The result has been to develop architectures of very great efficiency and to eliminate features which were ineffective, or which overlapped other features without providing any significant advantage.

There has been no move to subject operating systems to a similar penetrating analysis. To do so is the greatest challenge facing operating systems specialists at the present time. It is not an easy challenge; software differs from hardware in very great degree, and it does not follow that, because quantitative methods have been successful in processor design, they will be equally successful in operating system design.

If they do nothing else, measurements make people stop and think. For example, last May, a former colleague who now works for a company that is engaged in doing data base type work in C++, told me of an experiment in compiling the common libraries and client side of their system on a SPARCstation 10, and also on a 66 MHz 486 running Windows 3.1. In the result the SPARCstation took a little less than 10 minutes and the 486 took 6 minutes. The cost ratio of the two systems was ten to one.

I found this an arresting result. Obviously one should not draw far reaching conclusions from a single test of this kind. However, following up such a result, and performing further experiments under carefully controlled conditions, cannot fail to lead to a better understanding of what is going on.

There is, I know, work being done on the quantitative evaluation of system performance. I hope that more and more research workers will turn to this subject, and I hope that some of them will take the RISC movement in processor architecture as a model.

Single User Operating Systems

In spite of what I said above, I am far from happy with the assertion that the ideal operating system for a personal workstation differs in no way from an operating system designed to provide a time-sharing service on a minicomputer. I feel that that the use of a full scale multiuser time-sharing system for a personal workstation must be a case of overkill.

One can approach the problem by asking what are the minimum facilities needed to make the user of a personal workstation happy. The philosophy behind this approach is that, if you carry around facilities and features that you do not really need, you incur an unnecessary cost.

In a research environment users are likely to place great stress on the quality of the interactive service they receive; at the same time, they will want to be able to run one or more background tasks. They will also need network facilities for mail and file transfers. In research laboratories, many users like to be able to use other people's workstations at night or when their owners are away, and thus secure extra computing power. In some environments this is undoubtedly an important requirement. The question is whether it could be provided more simply than by running a multiuser time-sharing system, such as UNIX, on all workstations.

Personal users are likely to be very interested in that form of security better called protection or integrity. This will enable them to protect their files and running programs from accidental damage, whether it arises from their own actions or from those of other users. They are less likely to be interested in secrecy. In fact, many scientific users, especially in universities, see no need for secrecy and would be apprehensive that a system which emphasised it would prevent them from doing many things that they want to do.

The Oberon system, distributed by Niklaus Wirth from ETH Zurich, comprises a language and an operating system, both designed on the lean principles for which Wirth is well known. Those who feel that there may be a future for lean operating systems would do well to examine Oberon.

Program Optimization by an Operating System

There are limits to what can be done, either by a compiler or by a human being, to optimize a program without running it. Major optimization, especially of a program running on parallel hardware, is usually achieved by acting on observations made of the behavior of the program as it is run repeatedly. At present, this must be done by a human being.

There are various simple ways by which an operating system, or rather an operating system and a compiler working together, could use information from earlier runs to optimize a program automatically. For example, it might

cause a record to be kept of the number of times the various branches in the program were taken, and then modify the source code so as to minimize this number. This could be regarded as a form of branch prediction based on experience with that particular program, rather than on general program statistics. Similarly, an operating system might keep track of the way that paging occurred and modify the code so that appropriate prefetching took place and so that frequently activated code was retained in memory. Clearly there are issues here affecting the language designer, the operating system designer, and perhaps the hardware designer also.

I would like to feel that the above approach might lead to improved methods for automatically optimizing programs that run on parallel hardware. In my view, significant progress with this problem will only be made if ways can be found of making use of experience accumulated during the program's entire running history.

Acknowledgement

I am grateful to many colleagues with whom I have had discussions while preparing this address; in particular, I would like to thank: J. R. Bacon, R. M. Needham, C. P. Thacker, and N. Wirth.