South African
Computer
Journal
Number 2
May 1990

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 2
Mei 1990

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Subscriptions

	Annual	Single copy
Southern Africa:	R32-00	R8-00
Elsewhere:	$32-00	$8-00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Editorial

Information Systems Research: A Teleological Approach?

The request to write this editorial came at a very opportune time, coinciding as it did with an intense examination of the development of the field of information systems and an analysis of the progress of IS research. I have therefore used this opportunity to focus my thoughts and outline some of my conclusions. By doing so I don't pretend to answer any questions, merely perhaps to stimulate thought amongst those SACJ readers involved in IS research.

The last fifteen years has seen a tremendous growth in the study of information systems. During this period a number of journals devoted to IS research appeared such as *MIS Quarterly, The Journal of MIS, Information and Management* and *Data Base.* There are now many research-based activities: the International Conference on Information Systems; the annual IS doctoral dissertation colloquium; and various awards for IS research contributions. Hundreds of universities world-wide have formed information systems departments with (reasonably) standard curricula.

Yet with all this, what has *really* been achieved from a research viewpoint? Are we any closer to understanding the true nature of information systems? Is there a general unified theory of information systems? Is there even an accepted, unique body of IS knowledge? The answer to all of these must surely be no.

We have, I believe, achieved precious little. Yes, we do understand something of IS development approaches. We understand a little more now than we used to about how users interact with systems. But to get back to the first question, do we really understand what information systems *are* and how they work? No. Which begs the question: Why not?

There are, again I believe, a number of reasons, but the foremost must be that the majority of people in the IS research community either reside in the business schools of the USA or are drawn from other disciplines. These people, it would appear, are researching for research's sake; to publish in order to secure tenure or develop a research track record, not to further the body of knowledge of the subject. There seems an almost frantic zeal to generate and test hypotheses, trying to adopt and pursue what is seen to be a "scientific approach". But there is very little focus - there can't be, or the answers to my questions earlier would be yes rather than no!

Let me hasten to add that there is nothing unique about these IS researchers. "Publish or perish" is still very much alive and well! But also they are really not all that different from other social scientists. As Nagel [3] observed:

> "... in no area of social enquiry has a body of general laws been established, comparable with outstanding theories in the natural sciences in scope of explanatory power or in capacity to yield precise and reliable predictions ..."

Why should this be the case? Is it because the great intellects gravitate to the natural sciences and the social sciences pick up the second best who are incapable of generating these general laws? I hope not! The answer may well be that we have become locked into a particular research approach which is inappropriate to developing a body of social science, and more particularly, IS knowledge. Maybe we should be learning from our own source discipline (systems theory) and be developing a real research approach which complements our field of study.

To explore this further let me go back to the roots of information systems. What is an information system? Do we really have an accepted definition? Probably the most widely referenced is that provided by Davis and Olson [2]:

> "an integrated, user-machine system for providing information to support operations, management and decision-making functions in an organization. The system utilizes computer hardware and software; manual procedures; models for analysis, planning, control and decision making; and a database".

Note how this emphasizes the man-machine interrelationship and underscores computers as a core component when they are not even necessarily a part of the information system. The worst aspect is that it does little to describe what a system is, and this may well be one of the causes of our research dilemma. Again, if we draw on systems theory then a more appropriate definition might well be: "a hierarchical set of procedures utilizing information to monitor and control organizational performance". Note that this definition fits with general systems theory that *all systems* have four basic foundations: cybernetics, hierarchy, control and information [1].

An additional aspect not apparently recognised by IS researchers is that the information system, just like any other system, biological or otherwise, suffers from the problem first identified by our own Jan Christiaan Smuts [4]: that of holism. Simply put, this says that the whole is greater than the sum of the parts. This means that information systems, unlike science, cannot be reduced to simple isolated fields of enquiry and then analyzed or tested using hypotheses and laboratory experiments from which elaborate generalizations may be inferred. They have levels of complexity with new factors emerging at each level. The problem with most of the current research is that it starts out with a reductionist approach and then focuses on the highest (or lowest) level. Thus the majority of the topics have as their target the interaction between user and computer or the management or application of technology. There is very little research that is taking place at fundamental level, that of developing a general theory of information sytems. This is the teleological approach, searching for the natural laws and developing the theory based on deduction and logical development. Until we can advance *that* area of knowledge and, from a basis of these fundamental laws, develop a hierarchy of hypotheses that can *then* be tested, we will have little focus to our IS research. It will remain a fragmented, uncohesive smattering of the work of individuals who are merely grasping at tenure. There are few people who would today argue against the inclusion of information systems as a field of study at a university or as a fruitful research area. But until such time as we focus on the foundation theory, it will remain unstructured and immature.

References

[1] P Checkland, [1984], *Systems Thinking, Systems Practice*, John Wiley and Sons, New York.

[2] Davis and Olson, [1985], *Management Information Systems: Conceptual Foundations, Structure and Development*, 2nd Ed: MacGraw-Hill, New York.

[3] E Nagel, [1962], *The Structure of Science*, Routledge and Hegan Paul, London.

[4] J C Smuts, [1929], *Holism and Evolution*, MacMillan, London.

Prof Peter Lay
Assistant Editor: Information Systems

This SACJ issue is sponsored by
Department of Computer Science
University of Cape Town

Implementing UNIX on the INMOS Transputer

P J McCullagh and G de V Smit
Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

The transputer is examined as the target for a UNIX *port and specific problems encountered during implementation of a* UNIX *kernel in a transputer environment are discussed. Some solutions to the problems are suggested and their relative efficiencies discussed.*
Keywords: *Multiprocessor, UNIX, distributed processing, transputer.*
Computing Review Categories: *C.1.2, D.4.7*

Presented at Vth SA Computer Symposium

1 Introduction

The success of UNIX[1] twenty years after its initial conception is proof that it was not just a good idea at the time. In fact, UNIX is becoming widely recognized as an industry standard for operating systems.

The transputer, unlike UNIX is a relatively new innovation [7,11]. The transputer is a concurrent processing machine based on mathematical principles[2] and is increasingly being used to solve a wide range of problems of a parallel nature.

Some interest has been shown in combining these two aspects of computing [12]. This paper examines the issues involved in implementing UNIX on transputers. Due to the parallel nature of the transputer and the ease with which multiple processors are connected to form a network, it is natural to think in terms of a distributed UNIX system on transputers. In fact, it will become evident that certain problems involved in implementing UNIX on transputers can only be solved, at least at this point in time, by using more than one transputer. Nevertheless the main focus of this paper is on the general problem of the (in)compatibility of UNIX and transputers and not so much the problem of creating a distributed UNIX.

The problems and solutions discussed in this paper are based on work done as part of a project undertaken by the Laboratory for Advanced Computing in the Department of Computer Science at the University of Cape Town. The project involved the creation of a distributed version of UNIX (based on Tanenbaum's MINIX [10]) on a network of transputers. The prototype system, known as DISTRIX, is illustrated in figure 1. The satellite model [1] is used to distribute user processes across a network of transputers, while system processes execute on two central processors. More detail about the DISTRIX project and the method used to create a distributed UNIX can be found in [9].

The next section presents a brief description of the transputer processor. Section 3 discusses the general issues involved in getting UNIX to run on one or more transputers. Section 4 describes some particular problems encountered in the implementation of UNIX system calls on the transputer and possible solutions to these problems.

2 The INMOS Transputer

The transputer is a fast[3] micro-processor produced by INMOS to meet today's demands for highly parallel processing. It was designed essentially as a 'stand-alone' processor requiring no support chips to function correctly and effectively [11].

The transputer has a micro-coded scheduler which controls the running of an arbitrary number of concurrent processes. Machine code level instructions are provided to enable the creation and termination of processes. Processes are typically given a 1 millisecond timeslice and context switches require only 1 microsecond. This is considerably faster than typical micro-processors currently running UNIX.

Information passing between and synchronization of processes is done by means of a simple form of message passing implemented in the transputer microcode. The concept of a *message* (consisting of a length and a string of bytes) and a *channel* along which a message is passed from one process to another is supported by the transputer hardware.

A transputer communicates with the outside world via four hardware *links*, each of which can transfer data at a rate of 20 Mbits per second. Conceptually there is no difference between communication across a hardware link and internal message passing along a channel; the machine code instruction is the same.

[1] UNIX is a registered trademark of AT&T Bell Laboratories.
[2] Communicating Sequential Processes (CSP) by Hoare [6].
[3] The transputer is rated at 10 Mips by INMOS.

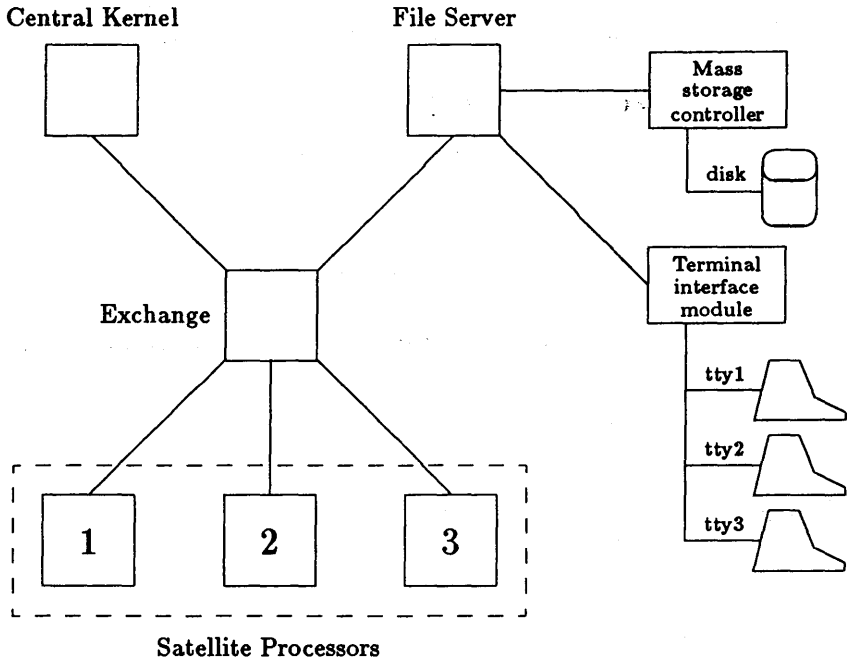

Figure 1: Current DISTRIX configuration.

In this way networks of transputers can be built by connecting the hardware links as required.

The transputer does not have a bus as do conventional micro-processors, but *link adapters* are available to connect the transputer to the popular busses, thus giving the transputer access to other peripherals. In addition to this, more and more peripherals are being produced which connect directly to the transputer link, allowing faster and more efficient access.

The transputer is a true 32 bit machine. It has an instruction pointer, a workspace pointer, an operand register and three general purpose registers, A, B and C, which operate as a 3 word stack. This stack is used to hold the arguments and result of an instruction and to pass parameters[4] and return values of a function call. Besides absolute addressing, memory can be addressed relative to the workspace pointer. This makes the workspace pointer the most natural choice for the stack pointer in a *C* program[5] which enables easy access to local variables and parameters.

The workspace pointer is also of special use to the transputer scheduler. The process queues of ready-to-run processes are maintained as linked lists of workspace pointers. The scheduler has registers which point to the front and back of these queues, thus enabling it to add a process to a queue as the process becomes ready to run and to determine the next process to be run.

[4]Of course, only 3 parameters can be passed in this manner and the rest must be placed on a stack maintained by the program in memory.

[5]*C* is the system programming language of UNIX.

3 Hardware Support for Operating System Functions

Let it be said at the outset that the transputer was *not* designed with an operating system like UNIX in mind. In fact, it may be argued that the transputer was not designed to run an operating system *at all*. Nevertheless, the question may be asked: how suitable is the transputer for running UNIX? In the following sections some general issues in this regard are discussed.

Memory Protection

Since UNIX is a multi-user operating system, users should be able to rely on the operating system for protection from the other users of the system. This includes both protection of data in secondary storage and that of executing programs in main memory. The problem of protection of data in secondary storage is discussed in the following section. With regard to protection of user data in main memory, some of the work can be done by the operating system software, but to ensure complete safety, some hardware support is needed.

The transputer provides no such support for an operating system. In fact, the transputer has one address space (as opposed to multiple virtual address spaces supported by most processors running UNIX). This permits any process to read or write any part of memory. As a result, malicious users cannot be prevented from disrupting the execution of other processes in the system. Perhaps even worse, a program that is under development (or incorrect for some other reason) can quite easily and unintentionally bring the

entire system and all its processes to a standstill.

While it is not possible to protect every process from all others, it is possible to protect a group of processes from other groups by executing each group on a different processor. In this case the fact that each transputer has its own memory serves as protection. In the UNIX environment one might, for example, partition processes according to ownership: all processes created by/for a particular user execute on one processor (one of the satellites in figure 1).

This does not solve the more severe problem of protecting the operating system. Every processor, no matter how small its function, must contain some kernel code to control its operation and integrate it into the rest of the system. This code is imminently vulnerable, and corruption would cause that processor to become useless and possibly disrupt other parts of the system. Corrupted processors must be detected and a method found to terminate former activity on that processor gracefully. The processor should then be rebooted without affecting the running of the system as a whole. An example of such a recovery system may be found in Amoeba [8]. Amoeba provides a *boot service* which performs the recovery function. Any server in the system may register with the boot server. The boot server periodically polls registered servers and if there is no reply within a specific time the boot server assumes the server has failed and initiates the activation of a new server.

Peripheral Protection

On the transputer all I/O is done via the links. Since the links are easily accessible to all processes, all processes can directly access the peripherals connected to those links. On the transputer there are no privileged instructions or execution levels which allows restricted access to peripherals by the operating system.

If it is known how the data is stored and how to activate a particular peripheral, there is nothing to prevent a user from, for example, accessing any part of the file system. The only way to prevent this is to connect peripherals only to 'privileged' transputers which do not run user programs. To ensure complete safety, links to such transputers (from other less privileged parts of the system) must be carefully monitored.

Virtual Memory

UNIX usually relies on hardware to be able to give each process in the system an unlimited address space in the form of virtual memory. Only part of the program's address space will occupy physical memory at any given time, while the rest resides on disk and is swapped in when required.

To support this, the hardware must provide an address translation mechanism and inform the operating system when a page fault occurs. The process concerned is then suspended while the operating system fetches the page from disk.

The transputer has no virtual memory capabilities. Instead all processes must be located somewhere in the physical address space at all times. In addition, very real limits are placed on the size of the code, stack and data segments of a program. Though off-chip solutions to these problems are being sought ([2]), they will never be solved adequately for a multi-process system due to the fact that transputer instructions are not restartable. If instructions were restartable, a process that causes a page fault could be descheduled and another process run while the page is fetched. However, since instructions are not restartable, the instruction that caused the fault must be stalled[6] while the page is being fetched by another processor. Since the transputer is not able to deschedule a process in the middle of executing an instruction, no other ready-to-run processes can become active on the same processor as the stalled process. Therefore, a page fault on a processor causes all processes to be suspended while the page fetch is done.

Until INMOS adds some virtual memory support hardware to the transputer, UNIX users will have to be content with solving their memory requirement problems by the addition of physical memory. The transputer has a 4 gigabyte physical address space which should be adequate for applications in the foreseeable future.

Interrupts

The transputer does not support interrupts as do conventional processors. Nevertheless, the transputer is very competent at real-time processing because, instead of an interrupt handling routine which is activated on the occurrence of an event, a dedicated process can be provided to wait for the occurrence of the event. Waiting is always done on a channel, so all events in the transputer are simulated by a channel input.

The solution of having dedicated event handling processes is not acceptable for UNIX environments, since this changes the interrupt model for application programs. Therefore some method of implementing interrupts must be found in order to run UNIX on the transputer. One possible solution is discussed in Section 4.

Process Scheduling

The transputer incorporates machine level instructions for the creation and termination of processes. Queues of ready-to-run processes are maintained by a micro-coded scheduler. The scheduler runs processes at one of two priority levels. Each priority level has its own process queue. Processes on the queues are scheduled on a round-robin basis. A high priority

[6] A stalled instruction is one that has been halted at some point in the execution of that instruction's micro-code. This can be achieved using off-chip hardware.

process will always be selected to run before a low priority process. In addition, high priority processes can never be pre-empted and once running, will only be descheduled on input from or output to a channel (or link). On the other hand, low priority processes can be pre-empted when their timeslice expires or by a high priority process becoming ready to run.

This implies that if implementors plan to use the built-in scheduler of the transputer to control processes, kernel code must run at high priority level, and user processes will have to be set at low priority. As a result all user processes run at the same priority level, which is not 'true' UNIX. At least two problems arise should one wish to circumvent the transputer scheduler. Firstly, this would negate one of the greatest advantages of the transputer, namely, the provision of fast, efficient process scheduling. Secondly, implementing such a solution is extremely difficult, if not impossible, considering the lack of interrupts and, more importantly, the relationship between the process queues and some complex instructions such as input, output and block moves.

On the other hand, having only one priority level for user processes on the transputer is not as bad as it would be on many other processors due to the fast switching time and small timeslice. For example, the problem of CPU bound jobs degrading the response of interactive programs (e.g. an editor) is minimal because, even if there were 50 CPU bound jobs scheduled before the CPU bound job, the delay would still only be 50 milliseconds. Considering the speed of the transputer, the interactive program would then be able to execute approximately 10000 instructions before it is descheduled and thus have time to respond to a user's request.

In addition, if a separate processor is given to each user as in the solution described in Section 3, users can only blame themselves for slow response times.

4 Particular Problems in Supporting UNIX System Calls

Implementing UNIX, to a large extent, reduces to implementing the system calls. A number of UNIX system calls give particular problems on the transputer due to the transputer's unusual architecture. In the following sections these problems are described and possible solutions examined. Only version 7 system calls [3] have been considered.

The KILL System Call

The KILL system call causes what is known as a software interrupt to occur in a selected process. As mentioned before, the transputer does not support the concept of an interrupt so some rather inelegant fiddling has to be done in order to implement this system call on the transputer.

The first problem is to identify and isolate the process that is being signaled. Processes in UNIX are uniquely identified by their process identifiers whereas transputer processes are uniquely identified by their workspace pointer and priority level.

The workspace pointer is usually used by a program as a stack pointer and therefore the workspace pointer can take on a range of values. However, the operating system can derive the range of values from the process identifier provided by the KILL system call since a table of processes is maintained by the kernel containing, amongst other information, a memory map of each process in the system.

Armed with this range of values and knowing that the process is low priority, the kernel must then search for the particular workspace pointer of the process. This can be found in one of three places depending on the state of the particular process.

1. If the process is descheduled but ready to run, it will be found on the transputer low priority queue.
2. If the process is doing I/O, the workspace pointer will be found at a memory location associated with the channel.
3. If the process was running when the kernel began executing, it will be found stored as the interrupted low priority process at a fixed place in memory, ready to resume execution as soon as no high priority processes are ready to run.

Unfortunately, if the workspace pointer is found in place 3, the process is not *suitable* for signaling as it is possible that the process is in the middle of a complex instruction such as a block move. So in this case, the process must be allowed to continue executing until its workspace pointer is found in either place 1 or 2. If the process to be signaled is compute bound and the only low priority process ready to run, a dummy transputer process must be introduced to force the process to be descheduled and placed on the process queue. Fortunately this is simple to do on the transputer.

Once the kernel has the workspace pointer, this value can be used to reference memory where the current instruction pointer of the process can be found (stored by the transputer scheduler). The value of the instruction pointer can then be changed to the appropriate interrupt handling routine.

The workspace pointer must also be altered to reflect a procedure call. This can be done either by the kernel or by the process itself just before it executes the interrupt handler. The latter option is preferable from an implementation point of view as the former may require altering the transputer process queue. However, a danger with this solution is that if signals arrive in quick succession (i.e. a subsequent software interrupt occurs before the stack pointer is adjusted) previous interrupts will be lost.

The FORK System Call

By far the greatest problem is presented by the FORK system call. The problem with FORK stems from the fact that the transputer memory is organized into one linear address space whereas some form of segmentation is required.

When a process executes the FORK system call the kernel makes an identical copy of the process (known as the child) and causes it to begin execution as a new process, with its own code and data areas, at the point of exit from the FORK call. The only difference between the execution of the 'parent' and the child is that the FORK call returns a different value to each of them.

When the FORK is executed on the transputer, the child process begins with every absolute address stored by the program (naturally) identical to that of the parent. Since the transputer has only one address space, these values all refer to data and code belonging to the parent—an *unacceptable* situation.

To solve this problem it helps to examine the solution used in conventional UNIX systems. Almost every one of these systems uses a hardware supported solution in which every address used by a program is not absolute, but is offset by a base or segment register. The segment registers are then changed when the processor switches context, thus ensuring that all addresses refer to the memory of the process being run.

In brief, the solution is to add another level of indirection to every absolute memory reference. On the transputer this has to be done by software and should be done as efficiently as possible.

One method is to designate some position in a program (e.g. the start of the code section) as the base. Before an address is stored (be it a pointer or a procedure return address) the value of the base is subtracted. Just before the address is used (to access memory or return from a procedure) the base is added. Program-counter relative addressing, in the form of the transputer's *load pointer to instruction* instruction, is used to find the base value at runtime.

Another method is to designate an available register as the base pointer. The operating system then ensures that this register is set before a process begins its time quantum. The value in this register is either subtracted or added to every address depending on whether it is being stored or used to access memory. Although the second method suggested is more efficient, it is impossible on the transputer as the only available register is the workspace pointer and memory can only be referenced at constant offsets from the workspace pointer.

The additional code to add and subtract the base value must be placed in the program by the compiler. It cannot be done by an assembler because it is a matter of semantics as to whether a value is an address or not. As a result, programmers working in assembler will have to be made aware of the potential problems and the solutions if they want their program to perform the FORK system call correctly.

The NICE System Call

It is not surprising that the NICE system call, which adjusts the execution priority of a process, has little or no meaning in a system that has only one priority level for all the user processes.

On the other hand, in a complex system of a number of transputers it may be possible to give some meaning to the priority of processes by the number of processes that may run concurrently with a particular process on the same processor. For example, a top priority process may require exclusive use of a processor while a low priority process may share a processor with any number of other processes .

The PTRACE System Call

This system call enables a parent process to control the execution of a child process and to examine and change the child's core image. Its primary use is the implementation of breakpoint debugging which unfortunately cannot be done well on a transputer. In fact, as will be seen, the only way to implement single stepping is by emulating instructions.

There are two aspects of the PTRACE system call that cause problems on the transputer. Firstly, a process that is being traced must be suspended indefinitely on receipt of a signal. In order to do this the same procedure as for a signal must be followed to the point where the workspace pointer is found in either place 1 or 2 as described in Section 4. The process must then either be removed from the transputer's internal scheduling queue or, if it is waiting on input, the reply message to the process must be stored.

The second problem is to suspend the process as soon as possible after the execution of at least one instruction. There are two main difficulties involved with this:

1. How is it determined that a process has executed an instruction?
2. How soon after that instruction can the process be suspended again?

These problems arise from the fact that the processes are scheduled and descheduled by a micro-coded scheduler and there is no way of circumventing this[7]. Therefore, to determine that a process has executed an instruction (question 1) is a complex task involving a combination of monitoring the process queue and the process currently under execution. Although a preempting high priority process can determine which low priority process is currently running, there is, unfortunately, no guarantee that a particular process will

[7]Processes will not run on the transputer unless they are scheduled by the on-chip scheduler.

be 'caught' during its execution. The smaller the intervals between preemptions the greater the chance, but doing this greatly increases the overhead on the transputer as a whole.

In answer to question 2, a process can only be suspended again after the micro-code scheduler itself has descheduled the process. In the worst case the process may have executed about 10000 instructions by the time it is descheduled.

Probably a far better solution would be to use a simulator that interprets transputer machine code. When the child process issues the system call to indicate it is ready to be traced by its parent, the operating system stops the process and starts an interpreter.

Except for speed it should not be possible to tell the difference between a program that is interpreted and one that is not. In addition, the operating system would have full control over the execution of the program and be able to execute instructions one at a time.

5 Conclusion

As mentioned in the introduction, many of the solutions suggested in this paper have been used in an implementation of UNIX on transputers called DISTRIX. Although DISTRIX is a minimal implementation of UNIX (only version 7 system calls are implemented), it is unique in that the functionality of UNIX has not been compromised. Most operating systems for the transputer, for example HELIOS [5], Trollius [4] and Meikos by Parsys, incorporate a strong level of UNIX compatibility [12]. However, they fall short of 'true' UNIX for various reasons. The most common failure is the lack of a conventional FORK system call. They also tend to avoid some of the other problems outlined in section 3. In pointing out these problem areas this paper has shown why the general tendency is to avoid implementing pure UNIX.

This paper has also indicated how the problems may be overcome and the implementation of DISTRIX has proved that UNIX can run on the transputer. However, it is necessary to introduce some inefficiencies. For example, it was found that the extra code added by the compiler to implement the FORK system call, caused programs to be approximately 40% bigger and execute about 30% slower. The overhead is not prohibitive and does not negate the original advantage of the transputer, namely its speed. Some of this overhead may also be reduced by changing the compiler more extensively.

Certain problems such as virtual memory support and kernel protection cannot be solved adequately, although many problems are solved by allocating one processor to each user.

These disadvantages must be seen as a trade-off against high power for low cost. The intended purpose of the system will further influence the trade-off. If, for example, the system is intended mainly for software development, the lack of adequate protection could be a serious deficiency. If the system is to be used for applications such as the correlation of statistical data, the lack of virtual memory may be prohibitive. However, should the system be used mainly for running well-debugged application software without excessive memory requirements, the above mentioned problems may be of little significance.

Nevertheless, the transputer has potential as a basis for large distributed operating system due to its high speed and ease with which a distributed environment is created. The implementation of the DISTRIX operating system has shown that UNIX can run on a network of transputers. Although there were difficulties, the work has been successful and if INMOS add more hardware support for operating systems to the transputer, DISTRIX can be updated to take advantage of the new features.

References

[1] M J Bach, [1986], *The design of the UNIX operating system*, Prentice-Hall.

[2] P J Bakkes, R Pina, and J J du Plessis, [1988], Transputer Enhancement: Virtual Memory Management for the Transputer, *Parallel Processing '88, Part I*, Johannesburg, October 1988.

[3] Bell Laboratories, [1979], *UNIX time-sharing system—programmer's manual*, CBS College Publishing.

[4] Cornell Theory Center, [1988], *Facts about the Trollius Operating System*, Cornell University.

[5] N H Garnett, [1987], HELIOS - An Operating System for the Transputer. *7th OCCAM User Group Int. Workshop on Parallel Programming of Transputer based machines*, September 14-16 1987.

[6] C A R Hoare, [1978], Communicating Sequential Processes. *Communications of the ACM*, **2**(8), 666–677.

[7] Inmos Limited, [1984], *IMS T414 Transputer Reference Manual*, Bristol.

[8] S J Mullender and A S Tanenbaum, [1986], The design of a capability-based distributed operating system, *Computer Journal*, **29**(4), 287–300.

[9] G de V Smit, P K Hoffman and P J McCullagh, [1989], DISTRIX: A Multi-Processor UNIX Workbench, *Technical Report CS-89-04-00*, Department of Computer Science, University of Cape Town, South Africa.

[10] A S Tanenbaum, [1987], *Operating systems—Design and implementation*, Prentice-Hall, 1987.

[11] P Walker, [1985], The Transputer, *Byte*, **10**(5), 219.

[12] P H Welch, [1989], Transputers, UNIX, and future support environments, *Workshop of the Occam User's Group UNIX and Operating Systems SIGs*, Canterbury. 21 February 1989.

NOTES FOR CONTRIBUTORS

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted **in triplicate** to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin. Distinguish clearly between such cases as:
  - upper and lower case letters;
  - the letter O and zero;
  - the letter I and the number one; and
  - the letter K and kappa.
- References should be listed at the end of the text **in alphabetic order** of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:
  [1] E Ashcroft and Z Manna, [1972], The translation of 'GOTO' programs to 'WHILE' programs, *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255.
  [2] C Bohm and G Jacopini, [1966], Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM*, **9**, 366-371.
  [3] S Ginsburg, [1966], *Mathematical theory of context free languages,* McGraw Hill, New York.

Manuscripts *accepted* for publication should comply with the above guidelines, and may provided in one of the following three formats:

- in a **typed form** (i.e. suitable for scanning);
- as an **ASCII file** on diskette; or
- in **camera-ready** format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies.

Charges

A charge per final page, scaled to reflect scanning, typesetting and reproduction costs, will be levied on papers accepted for publication. The costs per final page are as follows:

Typed format:	R80-00
ASCII format:	R60-00
Camera-ready format :	R20-00

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 2, May 1990
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 2, Mei 1990
ISSN 1015-7999

Contents