EFFICIENTLY MULTI-PROGRAMMING A SINGLE PROCESSOR

B. H. Venter
Department of Computer Science, University of Port Elizabeth
P O Box 1600, Port Elizabeth, 6000 South Africa
e-mail: csabhv@upe.ac.za

SUMMARY

A mechanism for efficiently multi-programming a single processor is explained in detail, with specific references to an implementation on the Motorola 680XX-architecture. Design alternatives are discussed and the chosen design is justified in terms of the assumptions made and requirements imposed. Potential pitfalls are highlighted and problems with the 680XX-architecture are pointed out. The multi-programming mechanism presented in the paper differs significantly from older multi-programming mechanisms and supports a novel scheduling technique for real-time processes.

Keywords operating system, process, real-time, scheduler, Motorola 680XX, computer architecture

INTRODUCTION

Efficiently programming a single processor to "simultaneously" execute several processes at once appears to be a relatively simple problem. However, if one constructs a multi-programming mechanism without the benefit of experience or a successful example, there are plenty of opportunities to make mistakes. The result may be long hours of searching for subtle non-reproducible bugs and an eventual "patched up" mechanism with several undesirable properties, such as expensive context switches, long periods with interrupts masked, and so on.

A number of operating system textbooks (for example[1, 2, 3, 4]) describe multi-programming mechanisms in a fair amount of detail. However, these expositions skim over *important* details. Furthermore, the multi-programming mechanisms they describe tend to be inefficient and largely based on out-dated assumptions and requirements. No detailed description of a modern, general-purpose multi-programming mechanism that also supports real-time processes, has so far been published.

This article is an attempt to explain and justify an efficient, general-purpose multi-programming mechanism based on modern assumptions, in enough detail to enable the reader to construct a similar mechanism with confidence. The article makes specific references to the Motorola 680XX-architecture[5] and offers some criticisms of the architecture. However, the discussion is relevant to other architectures as well.

ASSUMPTIONS AND REQUIREMENTS

This paper is based on work performed as part of a larger effort[6, 7] to implement a distributed operating system for a multi-processor system. In this system each processor executes solely from a private memory. A process is created in the private memory of a particular processor and is executed by that processor only. In other words, processes do not migrate and the only scheduling decision global to all processors is deciding in which processor a new process must be created. The latter aspect of scheduling will not be considered explicitly in this paper.

The implementations of some other multi-processor operating systems have taken the approach that a given processor should execute at most one process at a time, thus avoiding the need for a multi-programming mechanism. We felt, however, that this would be too restrictive and that it should be possible to run more processes in the system than there are physical processors to run them.

Nevertheless, a major aim of the system was to achieve "main-frame" performance by using as many microprocessors as may be necessary – even if most of the processors are idle most of the time. The multi-programming mechanism described in this paper is therefore based on the assumption that an individual processor will be running tens of processes rather than thousands of processes at any given time. It is further assumed that an individual processor will not usually be overloaded. These *assumptions* are in marked contrast with the assumptions of many of the older multi-programming mechanisms, and has led to a much simpler mechanism.

The *operating system* design *requires* that each process must be able to specify a desired "minimum rate of execution". An implementation must see to it that each process progresses no slower than at its minimum rate or, if the processor is overloaded, at as close to its minimum rate as is possible.

Over and above allowing a minimum rate of execution to be prescribed to the scheduler, the design requires the implementation to distinguish among two classes of processes: real-time and ordinary. The distinction is simply that no ordinary process may execute while a real-time process is ready to execute. This allows the designer of a real-time system to ignore any "non real-time" processes when calculating whether a given hardware configuration has enough capacity to guarantee the required minimum rates of execution (and hence maximum response times).

The "minimum rate of execution" is defined as follows: Let R and C be the numbers on an absolute scale representing execution rates, where R represents the desired minimum rate of execution of a process and C the maximum rate of execution possible on the processor executing the process. The process should then receive at least R/C of the total processor capacity for as long as it remains ready to execute.

For example, if an implementation chooses MIPS as an absolute scale on which to measure execution rates, and if C = 5 for some processor, and R = 2 for some process running on the processor, then it means that the processor has to expend at least 40% of its 5 MIPS capacity on the process in order to let the process progress at its required rate of 2 MIPS. The remainder of the processor's capacity can be used to concurrently execute other processes.

Since all processes must progress "simultaneously", it is necessary to put an upper limit on how long a ready process can be denied the processor. The *operating system* design *requires* that a process should receive at least R/C of the available processor capacity every 100 milliseconds, while it remains ready to execute. In other words, after the process becomes ready, the processor should spend at least (R/C)100 milliseconds of the next 100 milliseconds executing the process. The same applies to the next 100 milliseconds. And so on, for as long as the process remains ready.

The choice of 100 milliseconds is somewhat arbitrary and can be changed without invalidating the "rate of execution" idea. Basically, a 100 milliseconds was chosen because it was felt that executing every ready process every 100 milliseconds would result in individual slices that are large enough to do useful work in, while allowing interactive processes at least 10 processor slices per second. (Real time processes are allowed to override the scheduler when response times of much less than 100 milliseconds must be guaranteed. See the discussion of the routine `Execute_next`.)

Another *assumption* that sets this multi-programming mechanism apart from many older ones, is that virtual memory is no longer necessary since real memories are now larger than virtual memories used to be. Not supporting virtual memory has significantly simplified the entire operating system implementation effort, including the multi-programming mechanism.

The multi-programming mechanism is simplified because it can assume that processes are in memory all the time and that there is no need to cater for page faults. Furthermore, there is no need to take paging behaviour into account when making scheduling decisions.

PROCESS SCHEDULING

Some implementations of multi-programming provide two levels of process scheduling. The first level (macro-scheduling) is used to select a small subset of processes, and the processor is switched only among the processes in the subset using the second level (micro-scheduling). This technique obviously reduces the number of context switches as well as the number of times the more expensive macro-scheduling decisions must be made. It is also useful in conjunction with managing virtual memory working sets.

However, the technique basically assumes that there will always be more ready processes than the processor can handle. That is, it assumes that the processor will seldom be idle. Given the contrary assumptions and requirements stated above, a single level of scheduling is more appropriate.

The requirements stated above virtually dictate that two ready queues must be maintained and the lower priority queue served only when the higher priority queue is empty. Furthermore, an executing process must be preempted if a higher priority process becomes ready. The only decision thus left to the scheduling mechanism is the choice of the next process to execute, given a set of ready processes of the same priority.

The simplest mechanism that readily presents itself, is to organize each ready queue as a first-in-first-out (FIFO) queue, giving each process a static time slice of $(\mathrm{R}_P/\mathrm{C})100$ milliseconds before switching to the next process. (R_P denotes the minimum rate of execution, specified by process P. Thus, time slices differ from process to process, but a particular process always receives the same slice.)

Clearly, if the processor is not overloaded, each process will progress at least as fast as its minimum rate, provided there is no interference from higher priority processes, and only minimal interference from interrupt servicing. Of course, if the processor is overloaded, a process may not receive the processor within the 100 millisecond period. However, the delay increases in proportion to the degree of overloading. And, if averaged over several 100 millisecond periods, the rate of execution also decreases in proportion to the degree of overloading. The simple FIFO mechanism thus keeps the rate of execution "close" to the desired rate – as is required. Moreover, in the overloaded case, the number of context switches are as few as can be made while still satisfying the requirements.

However, if the processor is underloaded, a process may receive the processor, exceed its time slice and lose the processor, and then receive the processor again, all within a single 100 millisecond period. That is, more context switches will be performed than is absolutely necessary. Clearly, the number of context switches can be minimized by dynamically increasing the time slice of the executing process in proportion to the degree of underloading.

However, increasing the time slices in proportion to processor underloading is difficult to implement. The processor loading changes every time a process blocks or becomes ready, and consequently the time slice must be recalculated with each such event. Since these events are likely to be frequent, and the recalculations relatively expensive, the advantage of dynamically increasing time slices must be closely examined.

First of all it must be noted that if the processor is truly underloaded in the sense that the processor is often idle, it doesn't matter much whether some of the idle time is utilized instead to perform extra context switches. Furthermore, if most context switches are caused by the executing process blocking, rather than exceeding its time slice, the number of context switches cannot be much reduced by any policy. And, as already noted, if the processor is seldom idle, as well as overloaded (rate-wise), the fixed time slice mechanism results in minimal context switches.

It thus follows that the only situation where increasing time slices will save a significant number of context switches, is the one where the processor is "underloaded" in the sense that all processes can progress faster than their minimum rates of progress, but "overloaded" in the sense that processes usually exceed their time slices and the processor is seldom idle. Given the assumptions stated above, this does not amount to sufficient justification for varying process time slices as the processor's load varies.

SCHEDULING THE SCHEDULER

The multi-programming mechanism presents the following four routines as its interface to the rest of the operating system:

```
Make_ready (pcb)
Execute_next (pcb)
Handle_slice_expired
Block
```

The first three routines are usually called by interrupt handlers, and the last is called by the handler of a TRAP instruction executed by the current process. The effect of the routines will be described in the next section and is illustrated by figure 1.

For a variety of reasons, it is complicated to write these routines such that they are safely interruptible by each other. On the other hand, it is highly undesirable to mask interrupts for the duration of each routine, since a routine may execute several hundred instructions.

Consequently, calling `Make_ready` from an interrupt handler does not cause the real routine to be immediately executed. Instead, a "stub" routine is executed which merely adds a request to execute the real `Make_ready` to an "action queue". The same applies to the other routines.

The action queue is serviced by the "scheduler routine dispatcher" (SRD) which executes the requests one by one – by calling the real scheduler routines. This way, the scheduler routines can be executed with interrupts enabled, but without the possibility of a (real) routine being invoked while another is still executing.

It is of course necessary that the operations of adding and removing action requests must be "atomic". This can be done either by appropriately using the 68020 compare-and-swap instructions, or by masking interrupts for the short duration of these operations.

Rather than require all interrupt handlers to branch to the SRD when they return control, interrupt handlers are allowed to return control in the normal fashion (using an RTE instruction). The scheduler stub routines arrange that the last RTE instruction (which would have resumed the interrupted process) branches to the SRD. This way, an interrupt handler only "pays" for multi-programming if it actually uses a scheduler routine. For example, an interrupt handler servicing a serial input port, has to "pay" for scheduling only when a carriage return is pressed and the process waiting for an input line from the port has to be unblocked. Since this handler only needs to save and restore the one or two registers its needs (rather than *all* the registers), the savings can be considerable.

The first action taken by the SRD is to save all registers in the process control block (PCB) of the current process. Next, the SRD removes a request from the action queue and calls the corresponding scheduler routine. This continues until the queue is empty. Note that scheduler routines have the free use of most registers.

Once the queue is empty, the SRD restores the saved registers from the current PCB and resumes the execution of the current process. It follows that a context switch is accomplished simply by changing the current PCB.

Any entries added to the action queue *after* a decision has been made to exit the SRD won't be served until the SRD is restarted. Arranging for the SRD to restart as soon as it completes its exit, is rather tricky. Consequently, the SRD masks interrupts before checking whether to exit, and keeps them masked throughout the exit sequence (should a decision be made to exit). The following sequence is thus executed atomically at the end of the SRD:

```
IF action queue is empty THEN
  restore the registers of the current process
  resume the execution of the current process
ENDIF
```

As already mentioned, the scheduler routine stubs that are called from interrupt/TRAP handlers arrange that the RTE instruction that would have resumed the interrupted process, instead branches to the start of the SRD. It is of course possible for the SRD to be interrupted, and for a scheduler routine stub to be executed by such an interrupt. In such a case, the stub must simply allow the interrupt to return normally, rather than arranging for it to branch to the start of the SRD.

AN OVERVIEW OF THE MECHANISM

The multi-programming mechanism maintains three queues of ready processes. The ready queues are served in order of priority, with the highest priority queue reserved for real-time processes and the lowest priority queue reserved for a dummy "idle" process that is always ready. In addition, a pointer to the PCB of the currently executing process is maintained, as well as two global flags ("blocked" and "context switch requested").

The effect of each of the routines making up the multi-programming mechanism interface, can be summarized as follows:

`Make_ready (pcb)`
: The PCB is added to the end of the appropriate ready queue. If the process has a higher priority than the current process, a context switch is requested.

`Execute_next (pcb)`
: The PCB is added to the front of the real-time queue and a context switch is requested. (This routine provides interrupt handlers for real-time devices with a way to get blocked "high

priority" processes executing very soon after the interrupt was generated.)

`Handle_slice_expired`
: The PCB of the current process is added to the end of the appropriate queue and a context switch is requested.

`Block`
: A context switch is requested and the "blocked" flag is set to true. The PCB of the current process is presumed to be inserted into the appropriate data structure by the caller.

A point to note is that when a process becomes the "current process", its PCB is removed from its ready queue. And because a process can only block if it is the current process, it is never in a ready queue when it is the target of `Block`.

Of course, should the current process lose the processor without blocking, the PCB must be added to the end of the appropriate ready queue. This must be done carefully, since the current process may lose the processor more than once, as well as block, during a single run of the SRD. For example, while blocking itself a process expires its slice and is thus interrupted by the timer, and while this interrupt is being processed, an interrupt is received from an input device, with the result that a higher priority process is unblocked and the running process is preempted.

The simplest way to add the current process back into a ready queue at most once per run of the SRD, is to do it inside the uninterruptible termination sequence. Hence the SRD termination sequence must be extended as follows (making use of the global flags):

```
IF action queue is empty THEN
  IF context switch requested THEN
    IF not blocked THEN
      add current PCB to end of appropriate ready queue
    ENDIF
    choose a new current process
  ENDIF
  restore the registers of the current process
  resume the execution of the current process
ENDIF
```

In the implementation on which this paper is based, this requires at most 19 extra instructions to be executed in uninterruptible mode, bringing the maximum path length in this mode to 64 instructions. Should this be unacceptable, these instructions can be removed and the interruptable routines extended to test if previous routines have already placed the PCB back into a ready queue. It is also possible to somewhat reduce the remaining 45 instructions by resorting to coding tricks that make the code less readable. However, this was not felt to be necessary.

Some of the reasons for the rather high instruction count are given in the following sections.

TIME-OUTS

What happens to a process between blocking and unblocking is not strictly a concern of the multi-programming mechanism. However, in the system under discussion, processes mostly block subject to time-outs. That is, a process unblocks when the event it is waiting for occurs, or when an associated time-interval expires.

Thus, blocking on a time-out happens almost as frequently as context switches. Furthermore, the time-out mechanism interacts closely with the multi-programming mechanism, and also deals with hardware timers and interrupts. Consequently, it is convenient to treat the time-out mechanism as part of the multi-programming mechanism.

Thus, the multi-programming mechanism is extended with the following two calls:

```
Block_with_time_out (time_out)
Handle_time_out
```

Additionally, `Make_ready` and `Execute_next` are extended to appropriately update the time-out data structure when a process becomes ready before its time-out expires. That is, the rest of the operating system can always make processes ready by using these calls, regardless of whether the processes have pending time-outs.

The following high level description of the time-out mechanism suffices to specify what is required of the mechanism:

Let $C_1, C_2, ..., C_n$ be a set of (non-negative) integer counters and associate a counter with each process in the system. Initially, all counters are set to infinity.

Let $E_1, E_2, ..., E_m$ be a sequence of events, where each event causes either:

A) All counters to be decremented by one.

B) A given counter to be set to a given value.

Let furthermore, the following conventions apply:

1. When a process blocks, pending a time-out, its counter is set to the duration of the time-out.

2. When a process unblocks, before it times out, its counter is set to infinity.

3. When a counter reaches zero, it is set to infinity and its associated process is unblocked on a time-out.

Events of type A correspond to "timer ticks" and events of type B correspond to processes being added to or being removed from the list of processes waiting on time-outs. It thus follows that the above description describes exactly what the time-out mechanism must accomplish.

Clearly, the repeated action of decrementing all the counters will represent the bulk of the cost of a mechanism implemented exactly as outlined above. This cost can be reduced by various techniques, as described below. However, the minimal cost of the mechanism will increase as the length of the time-out list increases. This unsatisfactory state of affairs can only be rectified by hardware means. That is, by providing as many timers as there are processes.

The above description of the time-out mechanism fails to exploit three properties which allow cost reduction. The first such property is that it is not necessary to decrement counters with infinite values. The second is that a counter with a non-infinite value can only be decremented or set to infinity (because a process cannot block while it is blocked). The third property is that the sequence of events can be expected to have long runs of events of type A (timer ticks).

The first property can be exploited by maintaining an actual list of the PCBs of processes waiting on time-outs, and decrementing only their counters. However, care must be taken that the cost of maintaining such a list does not exceed the cost of the redundant decrements (assuming that the number of counters will have a small upper bound). The second property makes it easier to maintain the list, since it ensures that a PCB will never be added to the list if it is already in it.

The third property can be exploited by collapsing runs of type A. That is, by decrementing the counters with the run length at the end of a run, rather than by one for every event. The principal problem with this technique is that counters may reach zero during a run. That is, the tests for zero cannot be reduced to the same extent as the decrementing of the counters.

The problem can be solved by refining the technique to end a run as soon as the run length equals the minimum counter value. If, as can be expected, time-outs (counters reaching zero) occur rarely, few runs will actually be broken up.

The refinement introduces the need to find the minimum counter value. This can be done by using the loop which decrements the counters to also find the resulting minimum counter value. Thus, the minimum value at the end of a run can be found at a modest cost. Moreover, it is straightforward to update this minimum for each event of type B, making it possible to have the minimum available at the start of the next run of events of type A.

Collapsing runs should save effort when most events of type A come in runs of about three or more. And since a type A event corresponds to a timer tick every millisecond, it is very likely that this will be the case.

It should be clear by now that a hardware timer can be used to process type A events. That is, when a process is added to or removed from the time-out list (type B events), the following must be done:

```
add:
  stop the hardware timer
  subtract the elapsed time from all counters and ready all
    processes with counters <= 0
  add the PCB
  start the hardware timer with the new minimum

remove:
  remove the PCB
  stop the hardware timer
  subtract the elapsed time from all counters and ready all
    processes with counters <= 0
  start the hardware timer with the new minimum
```

If the hardware timer counts down to zero, its interrupt is processed similarly, omitting the add/remove step.

A REFINEMENT TO REDUCE THE COST OF A CONTEXT SWITCH

In the target system there are eight 80-bit floating point registers that must be saved and restored as 96-bits each. Doing so for every context switch is obviously expensive, especially as many processes will never make use of the floating point registers. It thus makes sense to avoid saving and restoring the floating point registers when possible.

Clearly, a simple flag in the PCB can be used to distinguish "floating point" processes from "ordinary" processes. The floating point registers can then be saved/restored only when the registers of a floating point process is saved/restored.

However, if there is only one active floating point process, there is no need to save or restore the floating point registers at all. Given the fact that the implementation targets a multi-processor, as well as the fact that floating point processes can be statically distributed among these processors, it can be quite likely for a processor to have but one active floating point process at a time. It thus makes sense to formulate the mechanism so that such situations can be exploited, especially since it can be done at the cost of a few instructions per context switch.

The floating point refinement basically works like this: Keep a pointer to the PCB of the process "owning" the floating point registers in a scheduler variable. When no real floating point process exists, the dummy "idle" process is designated the owner.

When saving the registers of the current process, do not save the floating point registers. Instead, modify the SRD to end as follows:

```
IF action queue is empty THEN
  IF context switch requested THEN
    IF not blocked THEN
      add current PCB to end of appropriate ready queue
    ENDIF
    choose a new current process
  ENDIF
  IF the current process uses floating point registers
  AND it is not the owner of the floating point registers THEN
    save the floating point registers in the PCB of the owner
    make the new current process the new owner
    restore the floating point registers from the PCB
```

```
    of the new owner
  ENDIF
  restore the other registers of the current process
  resume the execution of the current process
ENDIF
```

TRAPS, PROBLEMS, AND CRITICISMS

The easiest trap to fall into is to implement a fancy scheduling algorithm. Even if the algorithm is particularly efficiently implemented, it is likely to execute hundreds rather than tens of instructions every time a scheduling decision is made. In practice, making a scheduling decision means deciding where in the ready queue a process must be inserted when it becomes ready or exceeds its slice. This means that scheduling decisions are made as often as context switches. And when context switches are made every few milliseconds (how else in an interactive or real-time environment?), the cost of a fancy scheduler cannot be justified.

Another trap is the manipulation of scheduler data structures from interrupt handlers. For example, letting the handler for the "time slice expired" interrupt enter the executing process into the ready queue. Or letting the handler for the "time-out" interrupt update the time-out list. Even relatively simple operations on data structures require multiple instructions and leave the data structure dangerously inconsistent if interrupted. Continually masking and unmasking interrupts as critical sequences of instructions are entered and exited, is error prone and inefficient. Moreover, such an approach may inadvertently result in interrupts being masked for longer than is tolerable in a real-time environment.

The 680XX-architecture also provides numerous problems. For instance, it is easy enough to say in pseudo code "arrange that, when the last TRAP/interrupt handler returns, it is not the current process that is resumed, but the SRD".

The obvious way to do this is to copy the saved program counter (PC) and status register (SR) of the last stacked return frame to the PCB of the current process, and then to modify the return frame so that the corresponding RTE instruction branches to the SRD instead of back to the current process.

However, locating the PC and SR of the last return frame is a problem. The 680XX-architecture provides a separate supervisor stack, so one knows the address of the last word of the last frame. Unfortunately, there are different frame sizes and the offset of the PC+SR from the last word is different for different frame sizes. To compound the problem, an interrupt handler can be interrupted before even executing a single instruction. Thus it is not possible for an interrupt handler to reliably record its frame type in a suitable location.

Fortunately, the problem can be overcome by assuming all frames are of the same type, and relying on the handlers of interrupts with strange frames to inspect for, and fix up, incorrect "arrangements" before returning control (such handlers are rarely executed). Nevertheless, one would rather not have the problem in the first place.

Another simple pseudo code statement with a complicated realization on the 680XX-architecture, is "save all registers" meaning save the current processor context. One would like to do so as efficiently as possible, preferably with a single instruction. However, the MOVEM instruction is quite inadequate. First of all, context swapping takes place in supervisor mode, but involves the user mode stack pointer which is not visible to the MOVEM instruction in supervisor mode. The result is two extra instructions for each save/restore.

Moreover, in order to save the registers directly into the current PCB, rather than saving them on the stack and then copying the stacked copies to the PCB, a pointer to the PCB location must be kept in a register. However, this cannot be done because the registers have not been saved and hence cannot be changed. The result is three extra instructions.

Additionally, separate instructions are needed to save floating point registers, switch the memory map, invalidate the instruction cache, and so on. One can understand how this came about, but a dedicated "current PCB register" and appropriate context save/restore instructions now seem overdue, especially in the light of the introduction by the 68020 of a Master Stack Pointer which is of no use in the multi-programming mechanism described here.

Another problem on the 680XX-architecture is performing atomic insertions and removals of linked list elements. The 68020 purports to support this through its compare-and-swap instructions. However, instruction sequences using these instructions are difficult to construct and understand, and involve an unpredictable number of iterations through a loop. Dedicated linked list instructions would be more efficient as well as easier to use.

It is significant to note that, as far as one can tell from the description[8], the 680XX-like "TRON" architecture designed by the University of Tokyo, addresses both the above problems (as well as fixing some other 680XX quirks).

A further problem, not with the 680XX-processor architecture, but one that is present in many micro-computer systems, is the unsuitability of hardware timers for use by multi-programming mechanisms.

The target hardware of the implementation on which this paper is based, provides each processor with two Zilog Z8536 CIO Counter/Timer and Parallel I/O units. These timers are unsuitable for several reasons:

1. CIO registers must be accessed 8 bits at a time.
2. Accessing CIO registers incurs several wait-states.
3. CIO registers must be accessed in two steps: first write the register number in the control register, then read/write the control register.
4. An interrupt generated by the CIO must be cleared by writing to a CIO register.

Points 3 and 4 combine to form yet another problem: The handler of an interrupt from the CIO must write to a CIO register in order to clear the interrupt. Hence, accessing CIO registers must be done with interrupts masked.

Together, these factors conspire to make the operation of starting and stopping a timer needlessly expensive and a major factor in the time taken by context switches.

A more ideal micro-computer design would provide an array of millisecond resolution timers – as many as the maximum number of processes that can be simultaneously executed (256 seems a reasonable number). Each of these timers should have a uniquely addressable register associated with it. Starting a timer should be as simple as writing the time period into its register. Stopping a timer and determining the time left, should be as simple as reading its register. Preferably, the value returned should not be the time left, but the time elapsed.

If a timer counts down to zero, it should interrupt and automatically enter the stopped state. Naturally, the interrupt request logic should be cleared by the processor hardware interrupt service logic, and the identity of the interrupting timer should be obtainable by means of a single read instruction.

The availability of such an array of timers will significantly simplify and speed up the implementation of the multi-programming/time-out mechanism described in this paper. Naturally, it would be too expensive to implement such an array using existing "off the shelf" chips. However, it should be possible to create a single custom VLSI chip providing such an array.

CONCLUSION

The multi-programming mechanism presented in this article, is as simple as I can devise for the given requirements and I am correspondingly confident that it is correct. It is also more efficient than any other mechanism that I have come across.

Unfortunately, the simplicity of the high level description of the mechanism cannot be matched by an equally simple implementation on the given hardware.

References

[1] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall Inc., 1986

[2] R. W. Turner, *Operating Systems Design and Implementations*, Macmillan Publishing Company, 1986

[3] D. Comer, *Operating Systems Design The XINU Approach*, Prentice-Hall, Inc., 1984

[4] S. E. Madnick and J. J. Donovan, *Operating Systems*, McGraw-Hill Inc., 1974

[5] Motorola Corporation (Ed.), *MC68020 32-Bit Microprocessor User's Manual*, 2nd edn., Prentice-Hall Inc., 1985

[6] B. H. Venter, 'A detailed look at operating system processes', *Quaestiones Informaticae*, **6**(1), 2-7, May 1988

[7] B. H. Venter, *The design of a new general-purpose operating system*, Ph.D. Thesis, University of Port Elizabeth, South Africa, 1987

[8] K. Sakamura, 'Architecture of the TRON VLSI CPU', *IEEE Micro*, 17-31, April 1987