Computer Systems
G. Bell Editor

# Synchronizing Processors with Memory-Content-Generated Interrupts

J. Carver Hill
Lawrence Livermore Laboratory
University of California

**Implementations of the "Lock-Unlock" method of synchronizing processors in a multiprocessor system usually require uninterruptable, memory-pause type instructions. An interlock scheme called read-interlock, which does not require memory-pause instructions, has been developed for a dual DEC PDP-10 system with real-time requirements. The read-interlock method does require a special "read-interlock" instruction in the repertoire of the processors and a special "read-interlock" cycle in the repertoire of the memory modules.**

**When a processor examines a "lock" (a memory location) with a read-interlock instruction, it will be interrupted if the lock was already set; examining a lock immediately sets it if it was not already set (this event sequence is a read-interlock cycle). Writing into a lock clears it.**

**Having the processor interrupted upon encountering a set lock instead of branching is advantageous if the branch would have resulted in an effective interrupt.**

**Key Words and Phrases: interrupts, supervisors, monitors, debugging, parallel processing, associative memories, microprogramming**

**CR Categories: 4.32, 6.29**

This work was performed under the auspices of the U.S. Atomic Energy Commission. Author's address: Technical Information Department, Lawrence Livermore Laboratory, University of California, P.O. Box 808, Livermore, CA 94550.

In multiprocessor systems it is necessary for the processors to have some unambiguous means of communication so that their use of nonsharable resources (including impure program segments) can be interlocked and their cooperative execution of jobs can be coordinated. As a simplistic example, suppose there is a pointer kept in memory which contains the starting address of the next job to be run. If two processors simultaneously attempt to perform a read-then-update of this pointer, it is conceivable that both will read the same pointer value and, as a result, attempt to execute the same job. Confusion will be a certain result.

Common methods of interlocking processors in a multiprocessing environment are usually variations on the Lock-Unlock method [2]. In that method, the processor's basic instruction set must include LOCK, a memory-pause type of instruction that will test some specified lock (read an addressed bit or word in memory). This instruction will set the lock (i.e. will write in a one in that bit or word) if it was previously unset (i.e. at zero), but will branch (possibly to itself) without changing the lock if it was already set. A simple CLEAR or WRITE instruction can be used as UNLOCK, if only the process that sets a lock is permitted to clear it.

The crucial feature of the scheme is that, in addition to being uninterruptable (a sufficient condition for interlocking processes in a single-processor system), LOCK must be a memory-pause type of instruction. That is, after transmitting the content of the lock to a processor, the memory module containing the lock must not begin any cycles for other processors until the original processor has determined and transmitted the new lock state.

These pause-type memory cycles are longer than ordinary memory cycles by at least the two-way propagation delay between the processor and the memory module plus any multiplexing delay that is overlapped for ordinary cycles. Sometimes this prolongation of memory cycles is of no consequence, but frequently it has a deleterious effect upon system operation by increasing the latency of memory-access channels [3]. At best, it causes only a higher incidence of transfer-timing errors; at worst, it precipitates system thrashing [5].

Recently Zelkowitz published a hardware schema for a powerful interrupt structure that, among numerous other potential uses, provides a straightforward method of interlocking processors [6]. The key feature of the schema is the addition of an associative memory to each processor that will monitor all main-memory references, and interrupt the processor if the contents of particular referenced locations meet certain criteria.

In solving the processor-interlock problem for the Nevada Automated Diagnostics System,[1] subject to the constraint that prolonged or pause-type memory cycles

[1] The Nevada Automated Diagnostics System is a multiprocessor (DEC PDP-10's), multiuser time-sharing system for remote control and monitoring of experimental apparatus, real-time data acquisition, and problem-time data analysis and display.

could not be employed, an interrupt scheme called *read-interlock* was implemented. It is, in a sense, the lower limit of Zelkowitz's idea. In this case, the associative memory consists of only one cell, which contains a fixed pattern called the interlock pattern. The key principle, of having a processor interrupted or not interrupted, depending upon the content of a referenced memory location, is preserved.

The read-interlock scheme requires the addition of a *read-interlock instruction* to the instruction set of the processors and a *read-interlock cycle* to the cycle repertoire of the memory modules. The read-interlock instruction differs from an ordinary read instruction (which must also be a member of the instruction set) only in that the processor asserts a *read-interlock line* as well as the read-request line of the memory bus. Thus no modification of the processor's instruction-execution logic is required.

When the read-interlock line is asserted, the addressed memory module fetches the content of the addressed word and transmits it to the initiating processor as in an ordinary read cycle. But, instead of restoring the *just-read* data during the *write* portion of the cycle, as in an ordinary read cycle, the module immediately loads the accessed location with the interlock pattern (which is available as a hardwired pattern in every memory module). This sequence of memory-module events is a read-interlock cycle.

When the processor receives the interlock pattern in response to a read-interlock instruction, it is interrupted. (Data fetched in response to an ordinary read instruction that is coincidentally identical to the interlock pattern does not precipitate an interrupt.) The routine entered in response to this interrupt can be an executive-level routine not bound by address and instruction restrictions imposed upon lower-level processes. Consequently, it may make any response deemed suitable by the system designer. An absurdly simple but sometimes useful response is to return control of the processor to the interrupted process at the location of the read-interlock instruction (calculable from the contents of PC). This piece of pure procedure is functionally equivalent to a LOCK instruction that branches to itself if the lock is already set.

When a processor does not receive the interlock pattern in response to a read-interlock instruction, the running process will not be interrupted (for that reason), and it may leisurely examine the lock location and subsequently update the lock with new information. If any other processor or process examines the lock in the interim (with a read-interlock instruction), it will obtain the interlock pattern and be interrupted. It follows that sophisticated locking procedures, such as the dot-product method [4], present no special problem. Certainly a great deal can be done with the read-interlock feature even without adding embellishments such as a free field within the interlock pattern for microprogramming.

The principal difference between the read-interlock scheme and the TEST-SET scheme implemented in the 260 series is an interrupt (as opposed to a branch) response to the lock-set condition. While branching alone is faster than interrupting, the end result of the branch (if taken) is likely to be an interrupt (a call to the operating system to suspend the running job and go to the next job in the queue). In that event, evoking the interrupt directly from the lock-set condition gets the change-of-context bookkeeping off to a headstart.

However, it is expected that in the vast majority of lock-examinations the processor will find the lock clear and proceed with no delay of any kind.

Hopefully, the apparent utility of even so rudimentary an implementation of a memory-content-generated interrupt scheme as read-interlock will encourage other implementations and experiments. In their book [1], Bell and Newell call this area of intercommunication between processes and processors the least understood dimension of the computer space. Perhaps that is another way of saying that the most powerful innovations yet to be made in system architecture must be in this area.

Received October 1971; revised August 1972

**References**

1. Bell, C.G., and Newell, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1961, p. 83.
2. Dennis, J.B., and Van Horn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9,* 3 (Mar. 1966), 143–155.
3. Hill, J.C. Cycle-allocation disciplines for multi-access memory systems. In Proc. IEEE Internt. Conf. on Syst. Networks and Computers (Jan. 1971), 688–692.
4. Shoshani, A., and Bernstein, A.J. Synchronization in a parallel-accessed data base. *Comm. ACM 12,* 11 (Nov. 1969), 604–607.
5. Staudhammer, J., Combs, C.A., and Wilkinson, G. Analysis of computer peripheral interference. Proc. ACM 22nd Nat. Conf., 1967 ACM, New York, pp. 97–101.
6. Zelkowitz, Marvin. Interrupt driven programming. *Comm. ACM 14,* 6 (June 1971), 417–418.