Multiprocessing made easy

by RONALD J. PRICE
Perkin-Elmer Data Systems Group
Tinton Falls, New Jersey

INTRODUCTION

The concurrent execution of processes on multiple machines has appealed to system architects for quite some time.* In fact, it is not such a new concept as often thought. According to P. Enslow, "The first operational 'true' multiprocessor was delivered in 1962."†

Increased throughput is usually the major objective of multiprocessing, but there are other advantages over interleaved, multi-programming of a uniprocessor that oftentimes take on greater importance.

For example:

- Resource sharing
- Responsiveness to external events
- Expandability in a stepwise manner
- System modularity
- Survivability

There are a number of reasons why multiprocessing is not as popular as perhaps it could be, as Enslow aptly points out in Reference 2. One big problem is software. Program development is difficult because the behavior of concurrent programs is time-dependent and seldom reproducible with a given set of input data. Add parallelism with increased possibilities for time-dependent interactions and deadlock situations, and we have a most complex problem. The lack of software for the number of available multiprocessor systems\(^1\) attests to this fact.

Help is on the scene, though!

A new design concept that combines shared data structures and the operations permitted on them in a single syntactical construct called a "monitor" has been suggested in the literature by C. A. R. Hoare\(^3\) and Per Brinch Hansen\(^4\) as an effective tool for building concurrent system programs. Brinch Hansen based his Concurrent Pascal language on this principle. He extended the sequential program properties of the Pascal language with concurrent programming tools called processes and monitors.\(^5\) This paper is based on the constructs in Concurrent Pascal, although the research works of E. W. Dijkstra\(^*\) and N. Wirth\(^7\) are also relevant.

Evidence exists that the monitor concept, particularly as implemented in Concurrent Pascal, is a powerful tool. Brinch Hansen recorded improvement in programmer productivity while building a complete operating system with his language.\(^8\) The utility of the language has also been tested for many diverse applications.*

From personal experience, the author has found that the mere act of decomposing a concurrent problem into logical processes and monitor modules, irrespective of how they might be implemented, is an excellent design methodology. Furthermore, once system facilities are provided to manage process and monitor programs, the systems programmer need no longer suffer through analyses of time-dependent race conditions and interprocess communication protocols. Performance need not be sacrificed either. In fact, the author discovered that performance improved, probably because of a better overall system design.

The monitor concept is a relatively new idea and some weaknesses can be identified. Several researchers seek additional language constructs and operators (for examples see References 9 through 14). This is to say nothing of the age-old question whether compilers can produce optimum time and space code, nor the difficulties of implementing language support for multiprocessing. On the other hand, the system programmer need not wait for the ultimate language. He can reap the benefits of this new software technology for current problems even if he has to use assembly language. All he has to do is build a kernel facility, which manages process and monitor programs, and adhere to the programming conventions promulgated by these experimental languages.

The next section presents a brief explanation of the monitor concept and how it relates to multiprocessing applications. This is followed by a section on implementation techniques for supporting multiprocessors. The last section illustrates an example application and its solution. By employing two assembly language solutions, one with monitors and one without, the example evaluates the fundamental utility of the monitor concept for solving concurrent/multiprocessing problems; i.e., biases of a compiler for generating code and for enforcing programming conventions are not involved. The example also illustrates the value of the monitor as a systems structured programming tool by solving the concurrent problem in stages.

---

* Tightly coupled, multi-port shared memory systems are assumed in this paper, although much of the presentation equally applies to other types of multi-processor systems.
MONITOR CONCEPT

The reader should consult the references for an in-depth explanation of monitors and how they are employed in concurrent programs. Briefly though, monitors consist of shared data structures combined with a well-defined set of procedures permitted to operate on the shared data. In this context, processes are sequential programs which synchronize their operations and get access to shared data structures by calling monitor procedures in a manner much like subroutines. A run-time kernel executive enforces mutual exclusion on access to a monitor from competing processes.

The simple example in Figure 1 illustrates how processes can asynchronously exchange messages through monitors. Input and output operations are overlapped concurrently with job processing. The INPUT process reads a message from the IN_TERMINAL and pushes it on a buffer stack for the JOB process in the IN_BUFF monitor; a similar operation takes place on output. The monitors can be represented by ordinary program code except they are passive modules and do not execute until called. They contain the procedures (for example, PUSH and GET) that operate on the shared buffer stacks. They also contain the counters and flags needed to control the operation.

Inter-process synchronization is achieved through the use of a delay-queue construct by which a monitor procedure can delay its calling process or continue another process waiting in a delay-queue. For example, in Figure 1 the JOB process knows nothing of the situation of the INPUT and OUTPUT processes. If it tries to GET a message when none is available, it is delayed by IN_BUFF until continued when one is supplied by the INPUT process. Likewise, if the JOB process tries to PUT an output message when no buffer is available for the data, it is delayed by OUT_BUFF until the OUTPUT process FETCHes a previous message and frees a buffer.

Other applications for monitors include classical reader-writer problems, process synchronization on events, resource sharing, priority scheduling, and so on. The concept is valid because processes, containing their own private data structures, cannot directly intercommunicate, change control (shared) data, or access a shared resource; to do so they must call a monitor, and the kernel permits a monitor to service only one process at a time.

The technique of describing a system in terms of monitors and sequential processes can be used to represent a multiprocessor program. Any process is a candidate to be executed in parallel with others on separate processors. The three processes in Figure 1 could be executed literally simultaneously on different processors with no more difficulty (functionally) than if interleaved concurrently on one processor. Their combined execution becomes serial only when interacting through the same monitor.

In a multiprocessor configuration, monitor code can be executed out of shared memory or be replicated in each processor's private local memory. That is, if necessary, an instance of a given monitor can be represented on multiple machines. The separate representations can even be implemented differently (for example, microcoded) in different processors provided they execute the procedures as required by their respective processes. Monitor code can be duplicated in different processors because the run-time kernel permits only one version to execute at any given time. However, only one copy of a monitor's variable data structures can exist and it must be located in shared memory to be accessible by the different processors.

Each physical processor contains a run-time kernel. Collectively, these kernels create a virtual machine which supports an integral distributed processing system. In fact, control over processes for scheduling and dispatching is shared among the processors and consequently the virtual machine can be considered truly distributed. 35

For example, the collection of the processes and monitors in Figure 1 represents a concurrent system program that accomplishes an application function (albeit rather trivial in this case). It consists of disjoint modules which run in harmony as one program where parts of it may execute in parallel. No single processor controls the execution of the others. With adequate kernel support, the application program can be executed on multiple machines or on a single machine without change to any of its code.

In short, the use of monitors in this manner reduces the software implementation effort for multiprocessor systems to the scope of single-processor designs. Or from another viewpoint, greater degrees of performance can be achieved from one basic design simply by adding processors.
IMPLEMENTATION NOTES

A run-time kernel is needed to manage concurrent programs according to the principles of Concurrent Pascal. The kernel is responsible for waiting and dispatching processes (sometimes called short-term scheduling), and essentially is a miniature executive. In this case, it also recognizes monitor program modules and provides exclusive access to their respective shared data structures. Additionally, the kernel manages delay-queues and implements the delay and continue operators.

Given the kernel, the systems programmer builds an application program by coding appropriate process and monitor modules that solve his problem. These program modules are then linked together and loaded into the computer with the kernel. These steps can be automated with a compiler. In fact, the existence of the kernel is transparent to a program written in Concurrent Pascal or equivalent language. (Concurrent Pascal also provides strong type checking and a notion of access rights, both of which are outside the scope of this paper.)

Space does not permit documenting a complete kernel, but Figure 2 is a basic flowchart of four key kernel routines. Illustrated are mechanisms for entering and exiting a monitor and for implementing the delay and continue operators. These routines are executed in behalf of the using program when requesting the services of the kernel. The DELAY and CONTINUE routines in Figure 2 adhere to the conventions of Concurrent Pascal; note that CONTINUE implies EXIT as well. Alternative kernel designs are possible, as deemed appropriate for given installation objectives and requirements.

Support for multiprocessors is reflected in Figure 2 in several ways. In particular, the kernels (one in each machine) share data structures located in shared memory to control their operations. They need a mechanism for resolving contention when more than one kernel tries to access a common kernel data structure. This can happen when selecting a process to dispatch and when attempting to grant access to a monitor for a process. A hardware read-modify-write function (Test and Set instruction) is employed to lock a data structure before a kernel accesses it. When a kernel finds a structure busy, it “spins” on the lock, testing it until it becomes free. Locking is depicted in Figure 2 by boxes with titles. The kernel need not necessarily treat a busy lock as a busy wait situation, but it is not likely to have anything else to do except spin.

Processor time consumed while spinning on a lock is data contention overhead and will degrade total system performance if the frequency of accessing a given structure is high among the processors. Probability of contention can be minimized by dedicating a separate lock for each monitor and for each process dispatch chain in the system. Although the lock intervals can become a bottleneck when average process execution time between kernel calls approaches the execution time of the kernel routines, this is generally not the case because the lock intervals are relatively short. For example, in Figure 2 the lock interval in the ENTER MON-ITOR routine is not for the duration of the monitor procedure, but rather only for the short interval the kernel takes to interrogate the monitor’s busy flag and to chain the process to the monitor if the monitor is busy with another process. Upon exiting a process from a monitor, the kernel interrogates the monitor’s busy chain and makes a waiting process (if any) ready for executing the monitor.

A process can be either dedicated or shared. The code for a shared process is located in shared memory and can be executed by any processor in the system; a dedicated process is located in a processor’s private local memory. A shared process has an advantage over a dedicated process in that it can be dispatched by any processor in the system, whereas a dedicated process might be in a ready state but unavailable to idle processors in the system. While there are a number of reasons for having shared processes, they might execute slower than dedicated processes because of memory access contention and particular hardware constraints of a given installation. Processes might also have to be dedicated if they frequently access resources (peripherals, clocks, sensor equipment, etc.) that are accessible only on a given processor.

Process dispatching is distributed (except for the control structures in shared memory) because any processor’s kernel can set processes ready for execution on other processors in the system. As illustrated in Figure 2, this can happen when a process waiting on a monitor busy chain is activated and when a process waiting in a delay-queue is continued. No single kernel is responsible, nor need be interrupted, for deciding the order of putting processes in a ready state. When a kernel selects a process for dispatching it selects the highest priority process that it is capable of executing (processes inside a monitor have priority over processes not in monitor mode).

Figure 2 reflects some design choices that would not be required for a single processor system. For example, when continuing a process waiting in a delay-queue, the kernel might not be able to switch immediately to the continued process because it might be dedicated to another processor. The kernel routine falls through Dispatch instead. In fact, Dispatch is exercised whenever the kernel has the opportunity in order to insure fair priority treatment of processes in monitor mode. Otherwise, ready processes inside monitors might be blocking resources needed by other processes and therefore idling processors.

An inter-processor interrupt facility enables a kernel in one processor to signal the kernel in another processor when setting one of its processes ready for execution. All idle processors can be alerted when readying a shared process. The interrupt need not convey any information except to cause the recipient kernel to cycle through its Dispatch routine. If processes are not to be preempted, which has merits germane to the philosophy of the monitor concept, the interprocessor interrupt need be generated only when the destination processor is idle (or when the readied process is in monitor mode, if so desired to allow monitors to preempt processes). The interrupt facility is not an absolute requirement, however. When in idle mode with no
Figure 2—Kernel procedures to support monitors
work to perform, the kernel could continually cycle through its dispatch chains instead of going into a wait state (or perhaps periodically on a clock interval).

EXAMPLE SYSTEM

An example problem is presented to illustrate how monitors can be employed in building non-trivial concurrent programs. The utility of the monitor concept is evaluated by comparing the results with another solution of the same problem that did not employ monitors. Performance measurements are also given of various test runs when applying the monitor solution, as a multiprocessing program, to a dual processor configuration.

The following example application was defined as a benchmark model for investigating kernel designs and for evaluating relevant performance aspects. The benchmark is intended to reflect much of the character of a highly interactive transaction processing system, but actually it is a hypothetical application wherein data formats and processing functions are kept to a minimum (simulated) in order not to obscure the inherent interactions in the application problem being solved.

The benchmark application was implemented twice on an existing multi-tasking operating system in assembly language. Conventional operating system features for interprocess communications and synchronization were employed in the first implementation. This was done before any monitor capability was even designed. The second case employed monitors. A kernel facility employing the techniques described in the previous section was developed to run under the environment of an operating system task.

The application can be briefly described as terminal operators being able to exchange messages and to start concurrent job processes which execute disc I/O and computing functions according to the operator's request. The job programs send their results to the terminal specified in the originator's message. Job operations and terminal message exchanges run concurrently. Terminal I/O is overlapped so that an operator can input a message while the previous message is being processed. Output messages preempt input; job output takes precedence over terminal messages; job requests are serviced first-in-first-out. Each message delivered to a terminal contains a header identifying the source unless the previous output was from the same source.

Only the monitor approach to the benchmark problem is described here. Documenting the solution employing conventional techniques would serve no useful purpose. The monitor solution is itself somewhat arbitrary; there are probably many other ways of solving the benchmark problem in terms of monitors and processes. The approach taken here attempts to balance performance with conservation of memory. The benchmark program is built up in stages to simplify testing. Five terminals and two job processes are supported.

The benchmark program is described in pictorial form in Figures 3 through 5 with circles representing processes and boxes representing monitors. Program modules are identified by a global name and a type name in parentheses. Program modules of the same type have identical code; their data structures are replicated during program integration to build separate instances of the same module type.

The first stage consists of a single terminal subsystem as illustrated in Figure 3. The INPUT process continually inputs messages and passes them on to the message EXCHANGE monitor. The OUTPUT process continually gets messages from the EXCHANGE monitor when available and outputs them to the terminal. The terminal resource CONTROL monitor is responsible for granting exclusive access to the terminal and for maintaining its read/write/idle mode. The CONTROL monitor preempts a pending read I/O by executing halt I/O if the OUTPUT process requests access while the terminal is in read mode. The purpose of the READ monitor is to insure, by being a higher priority program module than the OUTPUT process, that the read I/O will be physically posted before the OUTPUT process can attempt to preempt it and cause contention problems with halt I/O. The CONTROL monitor also writes the prompt character while granting read access to avoid potential contention with halt I/O.

The next stage is to add the JOB processes as illustrated...
in Figure 4. The aspects of controlling the JOB processes is much the same as for the OUTPUT process. However, with the addition of the JOB processes, the CONTROL monitor must maintain the identification of the last writer in order to indicate whether or not a header message is required on a new output request. It must also provide a delay queue for the JOB processes separate from the OUTPUT process in order to give priority to job output versus terminal output. Separate EXCHANGE monitors could have been provided for the JOB and OUTPUT processes, but a combined monitor approach was selected to conserve memory. Code in the EXCHANGE monitor is kept to a minimum to avoid contention losses in multiprocessor configurations.

Adding terminals to Figure 4 impacts the buffer requirements of the EXCHANGE monitor. It must provide an exchange mechanism for each source-destination pair, times two to allow overlapped I/O. This number in terms of buffers is not actually required by the application (in fact two times the number of terminals is adequate), so a buffer pool is employed in EXCHANGE to dispense message buffers, two per INPUT at most. With the addition of Pop and Push procedure calls in EXCHANGE for obtaining and releasing buffers, the full benchmark program is illustrated in Figure 5 (for five terminals and two job processes).

Although this third stage appears to be complex, it is actually a trivial extension. All complex logic (of CONTROL and EXCHANGE in particular) was verified in the previous two stages (CONTROL and EXCHANGE were initially written with all the features needed in the final stage).

The total application consists of:

- 5 input processes of type IN
- 5 output processes of type OUT
- 2 job processes of type JOB
- 5 read monitors of type RD
- 5 control monitors of type CTRL
- 1 exchange monitor of type MGR
- 5 logical terminals of type TERM
- 2 logical devices of type DISC
- 27 delay queues (not described)

Clearly the system is highly modular and structured.

Comparisons between the two implementation philosophies for a single processor configuration are summarized in Table I. Development effort for the kernel is not included in Table I, as it is a generalized facility taking the place of the tools otherwise provided by the conventional operating system.

Although the design effort was the same, coding and testing of the monitor solution took less time which resulted in nearly a doubling of productivity. Documentation, acceptance testing, and other activities normally encountered in a production environment are not included in these figures; however, the overall improvement should still be significant considering the understandability, modularity, and structured design of the monitor approach. While developing the monitor solution to the benchmark application, we experienced no aggravation in resolving race conditions and in defining correct inter-process communication protocols as occurred in the conventional case. The translation from design to code was also smoother, and the design itself was much more lucid. For example, when running tests to measure minimum buffering requirements we were unable to explain the conventional-system results, but the monitor-system results were predictable.

<table>
<thead>
<tr>
<th>TABLE I—Benchmark Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Development Time (work-days)</strong></td>
</tr>
<tr>
<td><strong>Code</strong></td>
</tr>
<tr>
<td><strong>Test</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
<tr>
<td><strong>Memory Size (1,000 bytes)</strong></td>
</tr>
<tr>
<td>Sharable code</td>
</tr>
<tr>
<td>Total including data (monitor system includes over ½K for multiprocessing)</td>
</tr>
<tr>
<td><strong>Performance Tests (messages per unit time)</strong></td>
</tr>
<tr>
<td>User transactions with minimum job processing</td>
</tr>
<tr>
<td>Execute-bound exchanges</td>
</tr>
<tr>
<td>I/O-bound exchanges</td>
</tr>
</tbody>
</table>
The two solutions required approximately the same amount of memory. It should be noted that the kernel for the monitor solution was designed to support multiprocessing while the conventional-system was intended to run only on a single processor. The monitor solution therefore incurred unnecessary overhead when applied to a single processor.

The single processor monitor-system ran considerably faster than the conventional-system. The execute-bound test simulated maximum loading for terminal to terminal message exchanges. The I/O-bound test was performed with a single high-speed "terminal" (magnetic tape) sending to a low speed terminal. The better performance of the monitor solution, even with its resident multiprocessing overhead, is attributed to a better design and to more efficient interprocess communications with monitors versus existing operating system features.

The conventional-system was not designed to support multiprocessing because it seemed to be a very laborious and unproductive task. The real test of the monitor approach came about when we were able to bring up two different multiprocessor versions—a dual processor and a four-processor configuration with shared JOB processes—of the identical application program in only the short time needed to build load modules for each machine. After debugging the application on a single processor, no errors were uncovered during any of the multiprocessor tests on real hardware. Indeed, the processes which ran interleaved concurrently on a single processor ran in parallel on multiple processors without change to any of the application code.

Performance comparisons between single and dual processor configurations are given in Table II. In these tests the processors were the same model, and the data structures stored in shared memory for the dual processors were located in the same physical memory module for the single processor tests; furthermore, code was not executed out of shared memory. Consequently, memory access times were the same for the two configurations.

In the first test case, messages were exchanged between "null" terminals which did not involve real I/O nor much processing. The result is that the extra processor helped little (33 percent) because the application exhibited little
TABLE II—Multiprocessing Comparisons

<table>
<thead>
<tr>
<th></th>
<th>Single Processor</th>
<th>Dual Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal-to-terminal exchanges with “null” devices</td>
<td>383</td>
<td>510</td>
</tr>
<tr>
<td>User transactions with job processing (per processor)</td>
<td>182</td>
<td>181</td>
</tr>
<tr>
<td>Concurrent execution of job transactions</td>
<td>182</td>
<td>357</td>
</tr>
</tbody>
</table>

parallelism in this test run and because data contention (spinning on locked data structures) was a bottleneck due to the very short execution periods of the application routines.

The second test represents a more realistic situation. The rate of processing user transactions was measured on the single processor. In the dual processor case, one processor was kept busy executing job computations while the second processor processed the same user transactions as in the single processor case. The dual processor was able to process essentially the same number of messages as the single processor (99 percent) along with heavy computing loads.

The third test was derived from a different, but similar, application. It allowed the compute functions of job processing to be executed in parallel. The dual processor was able to process 1.96 times as many messages per unit time as the single processor.

CONCLUSIONS

We conclude that the monitor concept is indeed an effective tool for building concurrent system programs. Furthermore, the concept can be applied as a useful design methodology, irrespective of the implementation mechanisms. Highly modular and structured systems will result with minimal opportunities for obscure time-dependent bugs and deadlock situations.

We applied the monitor concept to multiprocessing and demonstrated its particular utility for such applications. System programs intended to run on multiprocessor configurations can be reduced to the scope of single processor designs by employing monitors. We also illustrated techniques for implementing a run-time kernel to support multiprocessing. With such a kernel, integrated distributed processing programs can be built that accommodate different multiple processor configurations without change to the application code.

ACKNOWLEDGMENT

Messrs. G. Anderson, J. Goettelmann, T. Kibler, and D. Schmaltz are thanked for their helpful contributions and review.

REFERENCES