A language for distributed processing

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

INTRODUCTION

The main question being addressed here is, what is a good way to program a multiple processor system (whether tightly or loosely coupled) to accomplish an integral distributed processing application? Writing concurrent programs for a uniprocessor is tough enough, but writing programs which interact and operate simultaneously in parallel can be a most difficult and frustrating experience. Opportunities abound for operational failures due to race conditions, for time-dependent bugs and for deadlock situations.

Help is on the scene, though, in the form of new concurrent languages as typified by Concurrent Pascal. The new software technology embodied by these languages can be applied to multiple processor problems as a methodology regardless of the implementation mechanisms. Nevertheless, the utility of having an effective language is beyond question, even if only as a design tool.

A key feature of Concurrent Pascal is the monitor construct that protects critical data regions shared among cooperating sequential processes. With a mutual exclusion mechanism, only a single process is permitted to access the critical region at any given time. This notion was first suggested by Dijkstra, formalized by Hoare, and implemented by Brinch Hansen in Concurrent Pascal. Monitors, or an equivalent construct or capability, have since been incorporated in many other languages.

Although different linguistic variations are possible, Concurrent Pascal was selected as a base for implementing distributed processing programs because of its track record and extensive documentation. The language has proved to be a powerful and effective tool in practice for building structured concurrent programs. Brinch Hansen recorded improvement in programmer productivity while building a complete operating system with his language, and the utility of the language has been tested for many diverse applications.

There has been some criticism of the language, however. For one thing, it depends on a run-time kernel facility that is invariant and built with a different language. For another, critical system design decisions have been assumed by the language. Researchers are also actively pursuing improved language constructs, most notably the manager concept, which ultimately may lead to simpler and even more reliable concurrent programming concepts.

The purpose of this report is to propose two fundamental modifications to Concurrent Pascal that not only will alleviate many of the above concerns, but more importantly, will extend the language's applicability to distributed system environments.

In many respects, the proposed changes are adaptations of principles incorporated in Wirth's real-time language Modula. As presented in the next two sections, they would enable the kernel and system control operators (i.e., the lowest levels of an operating system) to be written in the language itself and would enable partitions of a global, distributed multiprocessor program to be mapped to physical processors, but yet represented as an integral program.

The last section of the paper summarizes the proposed concepts and applies them as a methodology for constructing systems—from kernels, across processor boundaries, and up through application programs. As such, the extended language is a systems description language in that it can be employed to describe the algorithmic behavior of a multiple processor system (not to be confused with a hardware description language that prescribes physical circuits). It offers the systems designer a tool for:

- Synthesis
- Documentation
- Modeling
- Simulation
- Verification

and implementation if used directly as an implementation language.

Although the emphasis of this report is on distributed processing, the proposed extensions increase the power of the language for solving complex operating system problems irrespective of the multiprocessing issues. For example, the following problem areas are difficult under Concurrent Pascal as defined, but are quite amenable with the modified language:

- Data communications
- Process creation
- On-line system generation
- Dynamic software restructuring

The main intent of this paper is to justify and to explain...
the benefits of the proposal, not to specify the language nor to suggest a method of implementation. Semantic details and the mechanics of integrating the new constructs within the language need further study and exposure to actual practice.

The level of presentation assumes the reader is familiar with Concurrent Pascal, but the definition of a few items might be useful. A program that can be described by the language is called a concurrent program. It consists of system (or program) components defined as process types and monitor types (and class types not mentioned here); redefinition of the monitor type and the definition of a task component as a partition of a concurrent program are described. A Concurrent Pascal program includes a programmable initial process that directs the initialization of the components in the program. The interpretation of concurrency is the execution of multiple processes overlapped in time, either by multiplexing periods of execution on a single machine or by simultaneous execution on multiple machines. When important, the latter connotation of true concurrency (i.e., parallelism) will be explicitly denoted in context; multiprocessing implies parallelism, for example.

CONCURRENT PASCAL WITH A PROGRAMMABLE KERNEL

As represented by Figure 1, Concurrent Pascal is based on a virtual machine kernel that implements process switching, mutual exclusion on access to monitors, and the various control operators (DELAY, CONTINUE, etc.). The definition of the virtual machine interface can be a problem for system builders interested in different kernel features and/or in multiple machine operations. The problem is that the virtual machine has been abstracted away from the systems programmer to the point of existing literally in another world as defined by its unique language (typically assembly). Moreover, the line between the real and virtual machine might not be optimum for a given application. There are simply too many variables, parameters, factors and extenuating circumstances to consider in general.

In some situations, the programmer of the concurrent program would like to have an influence on the design of one or more of the virtual machine modules, sometimes even to interact with the internal machine dynamically. A prime example of this is programming interrupt service routines. Interrupt handling (typically for I/O processing) is related more to an application than to central general purpose kernel routines; this is clearly so in dedicated systems.

The handling of interrupts has historically caused untold grief and frustration for system programmers. The interrupt is an indeterminate and irreproducible happening. Contemporary systems researchers recommend against using it as a synchronization mechanism and avoid preemption in general. The notion of an interrupt does not even exist in Concurrent Pascal. Instead, synchronizing primitives are provided (DELAY and CONTINUE) that allow system programs to be designed with so-called cooperating sequential processes.

Unfortunately, the processes embodied by most peripheral devices on even modern computers cannot be considered cooperative. Modula was designed to handle them. But even if we stopped using the interrupt as a synchronizing mechanism, we still need it as a signal with which to measure time and to build real-time functions.

So although we might want to hide the interrupt in some abstract way, we still have to deal with it. Today this is generally accomplished through the kernel. However, not only is the interrupt hidden by the kernel, it is also typically unaccessible to the high-level software in a direct manner. Brinch Hansen and Hoare point out that scheduling cannot rely solely on built-in abstractions and that high-level software should be in control of response times at the lowest level. Indeed, the interrupt is the simplest form of low-level scheduling for machines that can switch an instruction stream automatically upon recognizing an external signal. (Some machines provide multiple priority states where an interrupt level may be interrupted by yet another level, but for purposes of discussion, a single level is assumed here.)

The ability to dispatch programmable service routines in rapid response to external signals and to manage them in a disciplined manner could be afforded to Concurrent Pascal by extending the language with a new construct that allows procedures to be called with interrupts disabled. To allow controlled sharing of the uninterruptable procedures and their data structures, they could be treated much like the ordinary "virtual-time" (i.e., interruptable) monitors. This new construct could then take the form of another system...
type in the language—a "real-time" monitor (for want of a better name). Generally speaking, the idea being presented here is to incorporate the real-time principles of Modula within the framework of Concurrent Pascal. Actually, we need not add a new system type to the language, but only have to redefine the monitor to include statements that execute in real-time.

The use of "real-time" monitors for interrupt handling is illustrated in Figure 2. Different delay (wait-on signal) and continue (send-signal) operators would be needed that are consistent with the real-time environment. With appropriate entry and exit mechanisms, processes could communicate directly with interrupt service routines without going through pre-defined intermediary kernel routines; even interrupt service handlers could directly intercommunicate.

The "real-time" monitor construct would have far more application than just for programming interrupt handlers. For example, when multiprogramming a single machine,
mutual exclusion on access to a monitor is assured simply by having interrupts disabled. In fact, there can be no busy queueing of processes on a “real-time” monitor. Consequently, they could be used in certain situations as a more efficient substitute for the ordinary “virtual-time” monitors in Concurrent Pascal.

Moreover, since the procedures in “real-time” monitors represent indivisible operations to their using program components, they can be employed to implement Concurrent Pascal’s “virtual-time” monitors with the language itself. That is, in Concurrent Pascal a process does not directly call a monitor procedure. The call is actually intercepted by a kernel routine to perform mutual exclusion and busy queueing if necessary. This kernel intervention is installed by the compiler in a transparent manner to the programmer. Under the proposed language, this kernel routine would be programmed explicitly and not automatically installed by the compiler (except possibly by default as an implementation-dependent feature).

The various monitor operators and, for that matter, any kernel-like function the systems builder needs would also be programmed in a direct manner. Even conditional critical regions with different scheduling algorithms (guarded regions) can be implemented with this “real-time” construct. In other words, the “real-time” monitor is a means for implementing explicit kernel routines, although the compiler could still support standard implicit kernel calls in a transparent manner.

Figure 3 is an extension of Figure 2 with kernel modules illustrated. An important aspect of this viewpoint is that the full power of the language can be brought to bear on the construction of the lower-level software when it is included as an integral part of the entire system. Such capability is important for embedded systems, process control environments, and data communications applications.

Kernel-like functions could be “hidden” through levels of abstraction, but this would be up to the systems builder and not a condition of the language. In fact, no run-time program, nor a pre-defined kernel definition, is required to support the proposed language.

The kernel can be treated as a concurrent program in its own right, and Figure 3 also illustrates this point. The Genesis process interacts with external processes in peripheral equipment through real-time monitors. It also performs system initialization and takes on the role of the initial process of a concurrent program as per Concurrent Pascal, including in this case the explicit creation of the high-level abstracted user processes. The Kernel Services real-time monitor in this example provides the standard Enter, Exit, Delay, Continue, etc., procedures and a Dispatch procedure for multiplexing processes. The kernel might control private devices as illustrated, but interrupt handling for the high-level software would also be supported (typically with considerable hardware assist) for dispatching processes in real-time monitors in response to interrupt signals.

The Genesis process selects high-level processes to execute with the Dispatch procedure and executes them much like a subroutine with interrupts enabled. So the high-level abstracted processes are in reality still the Genesis process in disguise. When the Genesis process recognizes an interrupt signal (presumably with hardware assist), it enters Kernel Services (with interrupts disabled) and takes appropriate action. In the event this action results in activating a waiting process, the Genesis process can decide whether to preempt (reschedule) the current running process or to schedule the waiting process. Typically, the action in response to an interrupt signal would be to dispatch the recipient process immediately in its real-time monitor which in turn would initiate scheduling actions as required.

Many of these low-level functions could be implemented in hardware or firmware. Nevertheless, they can be accurately represented and programmed with the “real-time” monitor construct.

Incorporating the real-time feature does not make the proposed language machine-dependent. From a language point of view, the new proposed construct simply represents the sequential state of the machine. However, escape mechanisms would have to be provided in the compiler for programming machine-dependent features in the low-level software modules; or provide machine-dependent statements as an adjunct to the high-level machine-independent language.

A MULTITASKING CONCURRENT PASCAL

The representation of a kernel as a concurrent program becomes more important when we consider a multiple processor system. Figure 4 is an example expansion on Figure 3 to illustrate kernels for a three-processor system; the surrounding higher-level software is not illustrated. The Inter-Kernel Communication (IKC) monitors are real-time monitors designed for exchanging information between kernels.

As should be evident from the previous discussion, the Genesis process together with the support modules in each kernel’s partition is actually a sequential program running on a sequential machine; parallelism is just an illusion to the higher-levels of software. Even in Saxena’s verification of the monitor concept, he had to represent the idle state of multiple physical processors with an idle process for each processor, the equivalent of the Genesis process. Consequently, in order to represent true concurrency (i.e., parallelism) we need a mechanism for representing the multiple processors, or at least the actions of their kernels.

Even if we were to assume the prior existence of a collection of cooperating kernels on multiple machines that form a virtual multiple instruction, multiple data path machine on which we somehow apply the high-level concurrent program, we still could not take full advantage of the parallel machine with Concurrent Pascal as defined. Loading and initialization of the program, for example, must take place sequentially, either on a single processor or in sequential phases on multiple processors because the initial process of the concurrent program is really the Genesis process of a single kernel.

So we need a way of dividing the global concurrent program into logical partitions that can be delegated to separate processors for initiation and execution. Indeed, we have no viable alternative but to divide the program into physical
Figure 3—Multiple levels of a concurrent system program.
Figure 4—Multiprocessing kernels.
partitions if it is going to be run on a loosely-coupled configuration that does not share memory.

The partitioning mechanism proposed here is to extend Concurrent Pascal with a task block structure somewhat analogous to *module* in Modula. The task construct, however, defines a concurrent system component that contains a collection of processes and monitors. It specifically includes an initial process for initializing the task. And tasks cannot be nested. A multiple task program represents parallelism in that different tasks, via their initial processes can be dispatched and executed simultaneously by separate processors. In other words, a multiprocessing program can include multiple initial processes which represent abstracted extensions of multiple kernels.

Each kernel in Figure 4 would be represented by a separate task, and each would be dedicated to a specific processor. The higher-level software could be implemented as extensions of each kernel or as separate tasks. As one or more tasks on a tightly-coupled system, the high-level modules need not be dedicated to specific machines and could be dispatched by any of the three kernels.

Each task is, in essence, an independent concurrent program and can be compiled into a separate load module. Tasks are linked at run-time to form a global system.

The correctness of the system can be tested with an integral compilation where the tasks interact through monitors at the interface of the task boundaries. The compilation of any given task, however, need only include its predecessor tasks in the system and not any task outside its view of the system.

Regardless of the issue of being able to express parallelism in the language, the task construct is a tool for partitioning a multiprocessing system program. Access rights as implemented in Concurrent Pascal will assure a structured design.

We can divide a concurrent program into sections by taking advantage of the isolation property of monitors. That is, processes intercommunicate and synchronize their operations through monitors, and consequently, they need not know anything about each other—even their existence. For example, in Figure 5 the User_B process need not know of the presence of the User_A process when calling the Buff_2 monitor, nor for that matter, even if multiple job processes interface the Buff_2 monitor. Therefore, we can safely cut the program between monitors and processes as illustrated. The trick is to keep the access right arrows pointing in the same direction across the task boundary. (Whether task initialization is performed by a separate initial process or one of the application processes in each task is not relevant to this example.)

Note that the system structure and hierarchical order of the program components, as required by Concurrent Pascal, is preserved if we define and initiate Task A before Task B, even if one physical processor dispatches Task A and another processor dispatches Task B. This would not be the case, however, if the Job_3 process were included in the Task A partition because then each task would have access rights to each other in a cycle.

Sometimes the initial layout of a concurrent program does not lend itself to partitioning. For example, if we tried to apply the tasks in Figure 6 to two different machines, the multiprocessing program could easily crash when started (even if one task is initiated before the other) because the design does not guarantee that the monitors will be initialized before being called. But then the program might not crash; the problem is a time-dependent race condition.

Start-up is only part of the problem. We also need orderly ways of stopping a multiprocessing program, and more importantly, mechanisms for detecting error situations across processor boundaries and recovering from them. This is what partitioning is about.

Figure 7 shows how we can take advantage of the insertion property of monitors to resolve this task layout problem. Here, a message exchange monitor and server process are inserted in the User_A process access path. This gets the arrows pointing in the same direction across the task bound-
A correct way to partition a multiprocessing program is to group logically-related processes and monitors into separate tasks in such a manner that their access rights point in the same direction across the task boundaries and by arranging the tasks in a hierarchy such that tasks which access other tasks are ranked below their predecessors, as in Figure 8. This ranking assures an orderly initialization (and termination) and eliminates race conditions and deadlock situations that otherwise might occur with a cyclic control structure.

In some arrangements, tasks, such as Task C in Figure 8, can be literally removed and brought back on-line without disturbing the rest of the system. The status of Task A does have to be known to Tasks B and C, however. In essence, the kernel tasks (not illustrated) and Task A form a virtual machine for Tasks B and C. This capability allows a system program to be generated and restructured dynamically.

DISTRIBUTED PASCAL

A key feature of this proposal for implementing distributed programs is the ability to describe interface monitors between processors. The characteristics of a given interface can be programmed with the 'real-time' monitor construct. Parallel kernels can then be described with the task block structure where the legality of their interface monitors is tested with an integral compilation. Higher-level tasks are built on top of the kernel tasks.

Interface monitors can be implemented in shared memory employing "thick-wire" communication techniques or in shared "thin-wire" I/O facilities. Mutual exclusion between machines is achieved by mutual cooperation in adhering to a protocol.

In the thick-wire case, permission to access the data structures is achieved by locking the monitor with a read-modify-write operation (e.g., Test and Set instruction) and then the data structures are manipulated in place. The logic for manipulating the data (i.e., the monitor's program code) can also be located along with the data if the hardware configuration allows code to be executed out of shared memory, or otherwise the logic can be replicated in the private memory of each processor.1,20

In the thin-wire case, data are physically copied from one location to another. Although Concurrent Pascal's monitors cannot be directly supported across a thin-wire boundary, an abstracted user's environment illustrated by Figure 9a could be supported by an underlying message communications system as depicted by Figure 9b. This software message system is conceptually the same thing implemented in hardware to support shared memory; however, the flexibility of a thick-wire emulation in software has to be highly constrained because of the limited bandwidth and long response times of the communication facilities.

A good case can be made for adopting a standard thin-wire communication technique for multiple processor systems which is adaptable to networks as well as to tightly-coupled architectures.15,22,30 The overhead normally associated with a message-based system can be ameliorated by implementing message exchange facilities in hardware.16,28

Special languages have been proposed for message systems,1,21 but Concurrent Pascal is a very suitable language...
for expressing networked systems, including the communications protocol. Moreover, the language offers the flexibility of general monitor designs where appropriate in addition to any built-in message exchange monitors of the communications system.

In any case, Concurrent Pascal as proposed to be modified is open-ended in the sense that both communication approaches can be accommodated. For example, if an operating system built with the language establishes a message-based inter-process communications protocol for conventional system use, the underlying implementation can still be based on thick-wire techniques where appropriate and more efficient.

Figure 10 depicts a multiple-task, multiple-processor system employing both thick-wire and thin-wire communications. The kernels dedicated to the processors in each tightly-coupled dual processor complex interface through multiple real-time monitors in shared memory, whereas the two complexes interface through a single real-time monitor over a communications channel. Kernels are represented by tasks and form the lower levels of the system. Higher-level tasks in the global system intercommunicate through monitors in a hierarchical fashion, as well. It is important to note that the different levels do not necessarily imply physical levels; that is, virtual kernels that emulate process switching, interrupts, etc., on top of real kernels is not a requirement to support high-level concurrent programs.

The point being made here is that this total system can be described with a single program (although part of it might be implemented in hardware). The program consists of a set of cohesive routines (program components) that implement the behavior of the global system. Indeed, the whole system operates as a harmonious confederation of cooperating sequential processes, some of which may run in parallel.

Even if the language is used only as a modeling tool, it can help us to design reliable systems by applying computer programming technology to their construction. This is so because Concurrent Pascal is based on proven software engineering techniques.

When building the "THE" multiprogramming system, Dijkstra suggested employing hierarchical levels of abstraction as a methodology for dealing with the complexity of operating systems; that is, modules are built on top of others with well defined interfaces and interactions. This technique, actually a formal method of structured programming, is an invaluable aid for proving program correctness and is an inherent capability of Concurrent Pascal.

The axiomatic definition of Pascal and the treatment of critical regions (e.g., monitors of Concurrent Pascal) and other research efforts have led to many proofs of program correctness relevant to concurrent programming (e.g., References 8, 14, 17, 26). These principles are now being applied in attempts to discover simpler, more flexible and more reliable techniques for constructing monitors and by enlisting the aid of the compiler itself.

The fact that formal constructs can lead to provably-correct programs may sound academic in reality. However, they actually do in practice lead to rapid program synthesis and to program correctness by inspection. Testing becomes much more systematic and takes on more of a verification role than a debugging operation. Modification and maintenance are also assisted.

By maintaining the consanguinity of Concurrent Pascal as proposed, we can apply these formal constructs to the construction of distributed systems. The language serves as a synthesis aid by enabling the system designer to decompose a system in terms of task components which in turn are decomposed into logically-related processes and monitor components; this can be illustrated in diagrammatical form. Moreover, the language allows a system to be designed in incremental stages, and it can be used to simulate and to evaluate different implementation strategies. In particular, the system designer can describe proposed solutions as models that accurately represent the physical environment and that can be demonstrated to run correctly. The language can also serve as a vehicle for documentation and testing. Finally, it becomes a piece-part of the end product where it is employed as an implementation language.
APPLICATION TASKS

KERNEL TASK

LOCAL DEVICES

OS TASK

NETWORK

APPLICATION TASKS

KERNEL TASK

LOCAL DEVICES

NETWORK

Figure 10—Distributed processing system.
CONCLUSION

Changes to the language Concurrent Pascal are proposed that enable it to be used to:

1. Describe the algorithmic behavior of the physical system.
2. Express the physical parallelism of a distributed multiprogramming system.

As such, the new language acquires the connotation of Distributed Pascal.

ACKNOWLEDGMENT

Mr. Gary Anderson and Mr. Tom Kibler are thanked for their helpful contributions and review.

REFERENCES
