The system data structure contention problem and a novel software solution for shared memory, floating control parallel systems*

by JERRY P. PLACE and ALAN A. GOERNER
University of Missouri
Kansas City, Missouri

ABSTRACT

Of the many varieties of multiprocessor architecture which have been proposed in the last five to eight years, shared memory, floating control multiprocessor systems are in many ways the most elegant. Shared memory, floating control architectures are distinguished by the structural attribute that each processor is an equal partner in the management of system resources even to the point of sharing a common copy of the operating system code and its data structures. A central feature and issue of such systems is the provision of dynamic processor load balancing. The most natural technique, however, for assuring dynamic processor load balancing by dispatching processes from a single shared queue is, we show, highly inefficient. We make the argument, validated by simulation studies, that the contention for the process dispatching queue, as they are commonly implemented, becomes so great so quickly as to severely limit the size and utility of such architectures. Examining the requirements for a solution to this problem, we derive a new, highly concurrent process dispatching queue structure with constant time average performance for all operations.

*This research was supported by grant K-5-27116 from United Telecommunications, Inc., Westwood, Kansas.
AN INTRODUCTION TO PARALLEL SYSTEMS

Parallel configurations of computing machines are being developed today across a wide range of architectures. It has been recognized that parallel configurations of computing machines can provide an enormous amount of computing power for the solution of problems.

Parallel architectures may be characterized as loosely-coupled or tightly-coupled (see Figure 1). Loosely-coupled systems generally exhibit a low level of sharing. Each system has its own memory, peripherals and copy of the operating system. Tightly-coupled systems exhibit a high degree of sharing. Typically, tightly-coupled systems share memory, peripherals and the operating system.

Control in tightly-coupled systems fits into one of three categories: master/slave, separate supervisor, and floating control. The master/slave mode assigns one of the processors in the configuration as the master. The operating system kernel routines always execute on the master. The master is responsible for dispatching work to the slave processors. The separate supervisor method of control maintains a copy of the operating system kernel for each processor in the system. Other system entities such as tables and common routines must be accessible through shared memory or a shared file system. The floating control method maintains one copy of the operating system and each processor executes that one copy. Thus, each processor is responsible for satisfying its own requests for service. Floating control systems are the most difficult to design and implement. The single copy of the operating system must be reentrant and constructs to ensure determinism must be embedded in the operating system.

The basic architecture of the parallel system and the mechanism of control largely determine the granularity of the system. Granularity is a measure of the degree of cooperation possible among processors in the system. Granularity may be expressed as very coarse, coarse, medium, fine, and very fine. Very coarse granularity implies parallelism at the independent task system level. Coarse granularity implies parallelism at the program level within a task system. Medium granularity implies parallelism at the procedure level within a specific program. Fine granularity implies parallelism among instructions within a procedure, and very fine granularity implies parallelism within a single instruction. Units of work in the system can be considered as shown in Figure 2, with independent task systems at the top. Task systems are composed of programs; programs are composed of procedures; procedures are composed of instructions, and instructions can be divided into the functional actions that occur in the hardware of the system.

LOAD BALANCING AND THE SYSTEM DISPATCHING QUEUE

A major problem in the successful employment of parallel architectures, especially shared memory floating control architectures, is balancing the load among the processors in the system. The easiest way to do this is to have a single queue of processes ready to execute; a single dispatching queue. Figure 3 depicts a configuration of processors about a shared memory with a single dispatching queue. Each program in the system is divided into its component procedures. All procedures com-
peting for resources in the configuration are represented by procedure control blocks in the dispatching queue. The procedure control blocks indicate the status of the procedure, either dispatchable or not, and the dispatching priority of the procedure. The scheduling mechanism behaves like a multi-level feedback queue which allows procedures to migrate from one priority level to another, depending upon their original point of entry. Since processors share all system resources including the operating system, processors are considered equivalent, thus any procedure may execute on any processor, regardless of the procedures that are executing on other processors. Thus, it is possible that several procedures from the same program will be executing in parallel on different processors in the system. As a procedure executes, events occur that cause the status of the procedure to change, perhaps it issues an I/O request, or it consumes system resources beyond a specified limit. In order to update the procedure control block to reflect the new status of the procedure, the processor must have exclusive access to the dispatching queue to ensure determinism in the system.

For example, a general purpose, moderately parallel system will support a number of concurrently active procedures; a degree of multiprogramming of several hundred is not unreasonable. The dispatching queue must support changing of priorities and setting procedure status such as EXECUTING, BLOCKED(SWAPPED-IN), BLOCKED (SWAPPED-OUT), and READY. When a procedure is first selected for execution, its control block must be marked appropriately so that it will not be selected by another processor. When the procedure is blocked for I/O or completes, the processor must again change the status in the procedure control block. Assume a single dispatching queue with several hundred procedures of varying priorities. Each processor must implement the search algorithm to find the highest priority procedure. Once the highest priority procedure is found, its status must be changed from READY to EXECUTING. Each processor must have exclusive access to the queue of procedure control blocks so that processors do not interfere with one another in the modification of the control blocks as part of the dispatching process. Processors are synchronized by requiring them to own the lock that guards the dispatching queue before they are allowed access to the queue. Thus, there must be mutually exclusive access to the dispatching queue to ensure that processor 1 does not select procedure b for execution when procedure b is in the process of getting its status changed by processor 2.

The dispatcher is enacted by each processor executing the single copy of the dispatching code that is part of the shared operating system in the shared memory. In essence, floating

---

```
PROGRAM Prog1;
PROCEDURE Proc1;
  ...
END Proc1;

PROCEDURE Proc3;
BEGIN
  x:=SQRT(y);
  a:=(b*c)/d;
END Proc3;
END Prog1;

PROGRAM Progn;
END Progn;
```

**Figure 2—Granularity**

**Figure 3—System dispatching queue**
control means that each processor must find its own job to execute next, independently of all the other processors. A processor finds its next unit of work by searching the system dispatching queue, and at a low level of entries (under 1,000), existing implementation methods do not appear to have much impact on performance. Since the dispatching queue may be used by one processor at a time, contention for its use can become a major system bottleneck.

CONTENTION IMPACT ON PROCESSOR OVERHEAD

The contention problem is graphically produced in Figure 4. Three simulation models of moderately parallel systems were created. The models evaluated degrees of parallelism from one to thirty processors. Three cases were studied, a single dispatching queue, a special dispatching processor and three equivalent dispatching queues. Each case assumed an average procedure execution time of 440 $\mu$s between context switches. The dispatching queue size was set at two hundred process control blocks and 200 ns was used as the memory access time, with a setup time of 50 $\mu$s per access. Thus, all processors required $50 + U(10,100)$ $\mu$s to update the dispatching queue and find its next procedure to execute. The simulations measured the time that processors had to wait while other processors had exclusive access to the queue (i.e., processor wait induced by contention for access to the dispatching queue).

The two alternatives to the single dispatching queue were chosen because of their practicality and because they did not materially increase the load balancing problem. The special dispatching processor searched the dispatching queue while other processors were executing user procedures. When a general purpose processor required a context switch, it was the function of the special dispatching processor to have a procedure ready for immediate switching. The dispatching function overlapped the execution of user procedures.

The three-queue case divided the dispatching queue into three separate but equal dispatching queues. If queue #1 was locked by processor b when processor a needed a context switch, processor a attempted to obtain the lock for queue #2. If that queue was locked by processor c, processor a attempted to obtain the lock of queue #3. If that queue was also locked, processor a was forced into a wait state for any one of the queues. Thus, the amount of time required to find a context-switch candidate by any processor was reduced by approximately one-third. More dispatching queues would further reduce the access time and processor contention, however, load balancing, the placement of procedures on queues, would become a significant problem.

Processor wait-time overhead for the three cases is summarized in Figure 4. The single dispatching queue presents a formidable impediment to moderate parallelism. At twenty processors, 80% of each processor's time is spent waiting for the dispatching queue lock. At a configuration greater than six processors, wait-time overhead exceeds 30% per processor in the configuration. The special dispatching processor is an improvement for configurations up to six processors. After six, however, the special dispatching processor behaves the same as a single queue. Multiple dispatching queues perform better. However, after twenty-two processors, contention for

![Figure 4: Data structure contention overhead](From the collection of the Computer History Museum (www.computerhistory.org))
the queues again induces significant wait time on every processor in the system. More dispatching queues would improve performance, but, as the number of queues increased, load balancing among the processors would also become a significant problem. The solution to this problem is to be found in a more sophisticated data structure supporting the dispatching process rather than multiple dispatching queues.

**DISPATCHER—O(1) SOLUTION**

Our solution to the problem of contention for the process dispatching queue is embodied in a module called Dispatcher. Dispatcher is unusual in two respects; in the analysis and design methodology which gave rise to it as well as in its data structure. That is, although novel in its structure, Dispatcher is not an ad hoc solution. A specific goal of the solution phase of this investigation was that the specification and implementation of Dispatcher should, as much as possible, be derived and stated from the set of requirements drawn from the preceding analysis.

The insights of these simulation studies give rise to five such requirements, ordered from most to least important. First, a complete package of services for process dispatching must be considered, designed, and implemented, helping to ensure that one operation is not optimized at the expense of others. Second, the Dispatch operation must execute in as nearly constant time as possible. Achieving this goal means, at minimum, that processor contention will no longer be a binary function of both the number of processes and processors, and will depend only on the latter parameter; thereby reducing significantly, we conjecture, the complexity of contention. Third, for all operations, but especially for Dispatch, the Dispatcher data structure and access protocol must minimize concurrent access blocking as much as possible while ensuring database consistency. That is, within the limits of consistency, the Dispatcher data structure and protocol must: (1) reduce the likelihood of collision and, thereby, reduce blocking, and (2) reduce the duration of blocking. And finally, fourth, the Dispatcher protocol must be live (i.e., free from deadlock), and, fifth, all other operations must execute in as nearly constant time as possible.

In accord with our orientation mentioned above, the first requirement is satisfied by developing a formal specification for a process dispatching queue. The technique used is an experimental one based on the notion of an inheritance hierarchy among abstract data types. In this regard, the technique is a descendant of the similar notions in object oriented programming. Although a full description of the development of the Dispatcher specification is the topic of a sequel to this paper, the derivation of Process_Dispatch_Queue, the immediate ancestor of Dispatcher in the hierarchy, exhibits most of the interesting points (see Figure 5).

The most surprising point being that process dispatching queues are not queues, nor are they priority queues. Upon examination, process dispatching queues are seen to be a hybrid data structure combining features of priority queues and simple databases. For example, a Dequeue operation seldom actually removes a process from the data structure but, rather, Updates its status to “Executing.” This dual heritage is represented in the “is” clause of the derivation. This clause implies that Process_Dispatch_Queue multiply inherits all the operators, exceptions, and axioms of both Priority_Queue and Database_V2.

Beyond this initial insight, the derivation also clarifies two other subtleties of this heritage. First, the bridge axioms show exactly how the priority queue function and database function of Process_Dispatch_Queue are related (i.e., that an enqueue is equivalent to a store of a process with status “Ready” and that a dequeue is not a delete but an update of the process’s status to “Executing.” Second, the exception conditions for Process_Dispatch_Queue are exactly the union of the exceptions for priority queues and for databases.

The end result of the derivation process is the automatic generation of a package definition for Dispatcher and a set of guidelines for the implementation of the package body. In the case of Dispatcher, the most important guideline, the one which gives the Dispatcher data structure its unique flavor (see Figure 6), can be rendered “Implement multiple inheritance as a multilinked structure.” The rationale for this approach is simply the congruence of the facts that Dispatcher logically has two independent access mechanisms, the priority queue and the database, and that multilinked structures, by definition, have two orthogonal access paths. With this useful hint, the requirements on a solution point, more or less directly, to the nature of each access mechanism.

For priority queue access, Dispatcher uses a data structure similar to the process state queue of Digital Equipment Corporation’s VMS operating system.7 This structure is a variation of Henrikson’s event set implementation which has recently been shown to have, with splay trees, better aggregate performance than any other priority queue structure in the literature. The Dispatcher implementation, like its VMS predecessor, uses a bit vector in the maintenance of the “Top” field, thereby delivering O(1) performance for the dequeue operation Dispatch and realizing the second requirement.

The fifth requirement’s goal of O(1) performance for all other operations is guaranteed by the use of an open hash table for database access. All operations of Dispatcher except Dispatch (i.e., Create, Fetch, Block, Unblock, Terminate, Delete, ChgData, and ChgPriority) access the data structure through the open hash table. Dispatch alone enters through the priority queue mechanism, which provides access only to the subset of process control blocks with status equal “Ready.”

However, the time complexity of the corresponding operations is only one reason for choosing the Henrikson and open hash table structures. Equally important is the high degree of partitioning these structures provide as a basis for satisfying the third requirement. As noted above, one technique of reducing concurrent access blocking is to reduce the likelihood of collision. The disjointness of the buckets of the open hash table and of the subqueues of the Henrikson priority queue support a much finer degree of locking than is possible in the numerous tree and list implementations which are traditional for priority queues and process dispatching queues.

The second objective of the third requirement (i.e., to minimize blocking intervals), necessitates, then, the development of a locking protocol (see Figure 7) to manage the fine grain
type Process_Dispatch_Queue

imports

  type Proc_ID is scalar; constant Null_ID is Proc_ID;
type Proc_Data is scalar; constant Null_Data is Proc_Data;
type Proc_Priority is scalar; constant Null_Priority is Proc_Priority;

is

  Priority_Queue (Proc_Control_Block, Proc_Priority, Null_PCB)
and
  Database_V2 (Proc_Control_Block, Proc_ID, Update_Field, Update_Value,
         Null_PCB);

where

  constant Null_PCB is structure {Null_ID, Null_Data, Null_Priority, Terminated};

  types
    Proc_Status is scalar {Ready, Executing, Blocked, Terminated};
    Proc_Control_Block is structure {Proc_ID, Proc_Data, Proc_Priority,
                                      Proc_Status};
    Update_Field is scalar {Proc_Data, Proc_Priority, Proc_Status};
    Update_Value is Proc_Data » Proc_Priority » Proc_Status;

  operators
    Create_DQ is Create_Q and Create_DB;
    Enqueue has Proc_Control_Block unfolded;
    Store has Proc_Control_Block unfolded, Proc_ID redundant;
    Replace deleted;
    ForAllItems deleted;

  axioms

    {PDQ : Process_Dispatching_Queue;
     ID : Proc_ID; D : Proc_Data; P : Proc_Priority; S : Proc_Status}

    Enqueue (PDQ, ID, D, P, S) = Store (PDQ, ID, D, P, Ready);
    Dequeue (PDQ) = Update (PDQ, Front (PDQ).Proc_ID, Proc_Status,
                      Executing);

  exceptions

    No_Ready_Procs renames Queue_Empty;
    ID_Not_Found renames Key_Not_Found;
    ID_Already_Exists renames Key_Already_Exists;

end Process_Dispatch_Queue.

Figure 5—Modula-2 implementation of dispatcher structure

of Dispatcher's concurrency control. This protocol has three levels of locking: bucket locks, queue locks, and node locks. Each of these locks secures a different part of the node structure. When a bucket is unlocked, all bucket links ("bNext" fields) are guaranteed to be intact and safe to traverse. Similarly, queue locks manage the integrity of the queue links "qPrev" and "qNext," and node locks manage the "id," "data," "status," and "priority" fields. The locking discipline will lock and hold a bucket only as long as necessary to traverse the bucket and lock the node. When the node is successfully seized and the bucket is released, the Dispatcher algorithms guarantee that no further changes will be made to the "bNext" fields thus allowing another processor to enter the bucket and traverse safely past the locked node as needed.

This "lock at the last moment and unlock at the first opportunity" philosophy runs counter to the prevailing theory of database concurrency control. For general reasons of database consistency and deadlock prevention, the canonical approach to database concurrency control is to lock all resources at once and to release them at once. Dispatcher, as a special
which Dispatcher implements, thereby satisfying the fourth requirement of the network's topology, which ensures deadlock freedom. The formal proof of liveness is developed by using the net invariant techniques of Petri net theory for colored Petri nets. However, a more intuitive argument can be made based on the notion of a resource ordering.

We note that the classic deadlock situation, and the one most applicable to Dispatcher, is resource waiting. That is, deadlock occurs when processor 1 has seized resource 1, processor 2 has seized resource 2, processor 1 tries to seize resource 2, and processor 2 tries to seize resource 1. While neither processor will back off and release the resource it has, neither one can progress. This elementary, but lethal, form of deadlock arises simply because the two processors requested their resources in the opposite order. The simple resolution stipulate that any routine that violates this ordering will defer all resources be requested according to that ordering, and to

Figure 6—Dispatching data structure

purpose system data structure, satisfies a more stringent set of conditions, though, which justify the use of its looser and more efficient philosophy.

Still, the demonstration of the liveness (i.e., freedom from deadlock) of the Dispatcher protocol is a significant aspect of its development. The formal proof of liveness is developed by using the net invariant techniques of Petri net theory for colored Petri nets. However, a more intuitive argument can be made based on the notion of a resource ordering.

We note that the classic deadlock situation, and the one most applicable to Dispatcher, is resource waiting. That is, deadlock occurs when processor 1 has seized resource 1, processor 2 has seized resource 2, processor 1 tries to seize resource 2, and processor 2 tries to seize resource 1. While neither processor will back off and release the resource it has, neither one can progress. This elementary, but lethal, form of deadlock arises simply because the two processors requested their resources in the opposite order. The simple resolution which Dispatcher implements, thereby satisfying the fourth requirement, is to define an ordering on resources, insist that all resources be requested according to that ordering, and to stipulate that any routine that violates this ordering will defer to the others and back off.

This observation permits us to conclude that Dispatcher is live in eight of its nine operators because they all enter via the open hash table and respect the ordering bucket → node → queue. Only Dispatch, which enters via the priority queue mechanism, violates this ordering by seizing a queue first and then attempting to seize a node. For this reason, Dispatch, and only it, implements a simple backoff subprotocol.

It is, finally, important to note that this implementation does compromise, in a small way, the second requirement's goal of $O(1)$ performance for Dispatch. The actual complexity of Dispatch is $\Theta(1)$ with the interesting caveat that its performance improves as the number of nodes in the structure increases. This latter effect arises because the probability of a back off equals the probability of a collision on the one node which is at the head of the priority queue. The probability of this collision is proportional to $p/n$ where $p$ is the number of processors and $n$ is the number of nodes, a probability that decreases as $n$ increases.

CONCLUSION

Clearly, tightly coupled, shared memory, floating control parallel processors can bring, in an elegant and incrementally extendable way, significant processing power to bear on the solution of user problems. However, we have demonstrated that neither the traditional control mechanisms and data structures nor their coarsely parallelized counterparts are effective and efficient approaches for the operating systems of this new class of computer architecture. For these new parallel processors to realize their promise in a general purpose environment, these control and data structures must be rethought and not merely translated. Failing this, we show that the contention for the essential system data structures is sufficient to fully negate the power and promise of these systems.

After a careful analysis of the requirements for a solution to the system data structure contention problem and after a thorough formal derivation of the specification for process dispatching queues, we found that there were, in fact, aspects of process dispatching queues which have gone unnoticed. These points indicated that a natural solution to the contention problem would possess three key attributes. These are: (1) a multilinked structure with orthogonal priority queue and database access mechanisms, (2) a highly differentiated structure for each access mechanism partitioning the nodes into many disjoint subsets, and (3) a fine-grained, multilevel concurrency control mechanism. Dispatcher is a process dispatching queue implementation possessing these attributes and providing $O(1)$ performance for all operations.

REFERENCES

Figure 7—Dispatcher protocol