LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Supporting State-Sensitive Computation in a Dataflow System

Computation Structures Group Memo 294
March 1989

Paul S. Barth
Rishiyur S. Nikhil

This report was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. Paul Barth was supported by a fellowship from Schlumberger Technology Corporation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Managing Shared Resources in a Parallel System

Paul S. Barth
Rishiyur S. Nikhil

Massachusetts Institute of Technology
Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA

barth@juicy-juice.lcs.mit.edu (617)-253-3219
nikhil@juicy-juice.lcs.mit.edu (617)-253-0237

Abstract

One well-known construct for managing shared resources in a parallel system is Hoare's monitors[4], which encapsulate the shared data, operations on it, and synchronization between operations. This paper describes *managers*, a version of monitors for the declarative language Id[8]. Like monitors, managers provide encapsulation; however, managers make two key improvements critical to parallel execution. First, operations on a shared resource have internal concurrency. This concurrency allows difficult locking issues encountered in monitors, such as nested monitors, recursive monitors, and the precise semantics of `wait` and `signal`, to be resolved programmatically without losing abstraction. Second, the implementation uses low overhead, non-busy-waiting locks for mutual exclusion. This low overhead increases the availability of shared resources, and encourages composite manager structures which reduce bottlenecks. The construct is described in detail, and our experience with applications using managers is described. This experience indicates that managers are effective programming construct for systems with multiple, dynamic threads of control and efficient context switching.

1 Introduction

Managing shared resources in a highly parallel system is difficult. One well-known construct for encapsulating operations on shared resources is Hoare's monitors[4], which encapsulate the shared data, operations on it, and synchronization between operations. This encapsulation allows invariants to be established between operations on the shared data, which are useful in correctness proofs.

This paper describes *managers*, a version of monitors for the declarative language Id[8]. Id allows fine-grained, parallel execution and non-strict function evaluation. Like monitors, managers provide encapsulation; however, managers make two key improvements critical to parallel execution.

First, operations on a shared resource have internal concurrency. Specifically, two parallel execution threads are used to update the resource and return the result to the caller. This separation allows

[†]This report was supported in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. Paul Barth was supported by a fellowship from Schlumberger Technology Corporation.

difficult locking issues encountered in monitors, such as nested monitors, recursive monitors, and the precise semantics of `wait` and `signal`, to be resolved programmatically without losing abstraction. Although managers are implemented in Id, this technique is effective in other languages supporting multiple, dynamic threads of control.

Second, the implementation uses non-busy-waiting locks for mutual exclusion. On a dataflow machine such as Monsoon[9], process suspension and synchronization take just a few machine instructions, giving manager operations efficiency on par with primitive operations such as test-and-set. This low overhead increases the availability of shared resources, and encourages composite manager structures which reduce bottlenecks. Other implementations with low-cost synchronization can also benefit from this technique.

The next section describes the manager construct and gives several examples, ranging from a simple memory manager to graph algorithms. Section 3 describes the implementation of the manager construct on a dataflow machine. In Section 4 we describe our experiences using managers in several application. Section 5 contains concluding remarks.

2 Managers

A manager is a high-level interface to a single *state variable*, which is shared by an arbitrary number of processes. The state variable may be a simple value or a data structure, such as a list, array, or tuple. Managers define functions for creating, accessing, and modifying state variables. Processes request a resource by calling these functions; the manager synchronizes incoming requests to ensure exclusive access to the state variable. Processing a request updates the state variable and returns a result to the caller. Figure 1 illustrates this view of managers.

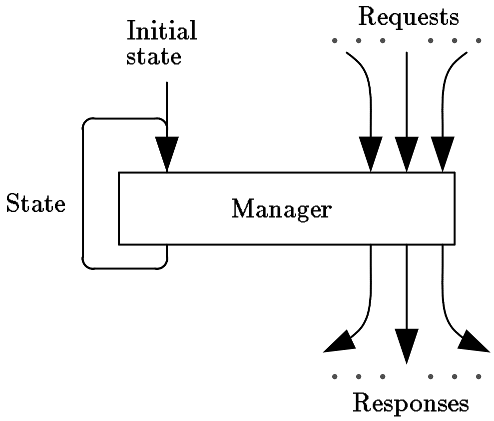

Figure 1: System View of a Manager

2.1 Encapsulating Operations on Shared Resources

Managers encapsulate the state variable like abstract data types. Specifically, each manager defines an abstract type for the state variable, with a set of query and update operations (called *request* *handlers*), and a constructor for creating instances of the type. State variables can be accessed only through this interface. In addition to defining the interface functions, the abstraction hides the synchronization and updates on the state variable, so that request handlers can be written functionally. The program interface to managers is similar to abstract types: manager instances are created (with some initial state) and passed to functions who may make requests on the instance. Manager requests return a result, just like other function calls. The synchronization and state update is entirely hidden from the caller.

As an example, consider a simple memory manager. A list of free memory blocks is created from a section of memory. An allocate request pops the first block off the free list; if the list is empty the response "no" is returned. Deallocate requests push a memory block onto the free list and receive an acknowledgement. Query requests return the number of blocks on the free list. The manager for this example is shown in Figure 2.[1]

```
type response = block | no ;
type ack      = ok;

MANAGER free_list = create_free_list block
 typeof alloc    = free_list -> response;
 typeof dealloc  = free_list -> block -> ack;
 typeof query    = free_list -> num
 rep (list block)
 { def create_free_list mem = break_up_mem mem;
   def_handler alloc   nil      = nil,no
        |      alloc   blk:rest = rest,blk;
   def_handler dealloc lst blk  = (blk:lst),ok;
   def_handler query   lst      = lst,(length lst);
 } ;
```

Figure 2: A Simple Memory Manager

As with abstract types, the manager construct has two components: the *interface* definition (the first four lines), and the *representation* (everything following `rep`). The interface gives the type signature of the constructor and the operators. The constructor `create_free_list` takes a block and creates an instance of the type `free_list`. The operators `alloc`, `dealloc`, and `query`, take a free list instance (and any other arguments) and return a result. Note that the interface definition hides the access and update of the contents of the free list.

The representation gives the type of state variable, and defines the constructor and operators. Here the type of a free list is a list of memory blocks. The constructor computes the initial state of the instance; in this case, it breaks the section of memory provided into a list of blocks. In the representation, functions that access or change the state variable are defined with the `def_handler` construct, which provides mutual exclusion on the state. (These are analagous to monitor entry procedures.) Handlers are written as functions from the old state (and other arguments) to a *pair* of values: the new state and the result. Note that the type signature of a function defined with `def_handler` hides the state and its update. For example, the `dealloc` function takes a free list and block as arguments, and returns an acknowledgement, as declared in the interface. However, the expression that computes the result is given the state of the free list (a list of free blocks) and returns the new state (adding the block to the list) and the acknowledgement (`Ok`). Thus,

[1]Some syntactic notes: The infix colon (`:`) is the cons operator; comma (`,`) is the infix tupling operator. A multi-line function (*e.g.,* `alloc`) uses pattern-matching on the structure of its argument in lieu of conditionals to select the result expression. Note that this manager relies on a more primitive manager for the cons operation.

`def_handler` provides an important abstraction: handlers are *defined* as state transformations that assume exclusive access; but handlers are *used* as functions without reference to the state. This distinction only applies to a handler's first argument; other arguments (even those referring to managers) pass in unchanged. Interface functions may be handlers, as in this example, or not; conversely handlers may be exported or strictly local. The manager body may also contain local values or function definitions.

2.2 Concurrency and Request Handlers

The `def_handler` construct has an important impact on parallelism. Each handler computes a pair of values, the new state and the result. Corresponding to each of these values is a thread of computation; to the extent that these threads are independent,[2] they may be executed in parallel. These threads asynchronously update the state (thus unlocking it) and return the result to the caller. This asynchrony may increase availability of the resource, or lower the latency in responding to a request. For example, a `query` request can return the state immediately, making it available while it is computing the length. Alternatively, the acknowledgement of a `dealloc` request may be returned before the state is updated.

Beyond efficiency, independent threads within a request handler can be stopped and resumed independently, allowing managers to schedule and reorder requests, as described in the next section. This generalizes to support recursive and cyclic structures, beyond the capabilities of monitors.

2.3 Scheduling with Managers

Scheduling reorders manager requests, deferring some while allowing later ones to proceed. This requires a facility for suspending and resuming a thread, which we call a *mailbox*. Associated with every request handler is synchronization variable referred to with the keyword `MAILBOX`. The mailbox is a first-class value that may be passed to functions and stored in data strucutres. The keyword `LATER` suspends an execution thread, awaiting a `send` operation on the mailbox. This sends a value to the mailbox, which resumes the suspended thread.

For example, consider a slight variation on the free list. In this scenario, allocate requests to an empty list are deferred until memory is deallocated. Deallocate forwards its block to a deferred allocate, if any. The manager in Figure 3 implements this example.

The interface to this memory manager is identical to the previous one. However, the representation is significantly different. The state variable contains two items: the list of free blocks, and a list of deferred allocate requests. The `alloc` request handler shows how they are deferred. If there is a block on the free list, it is returned as usual. If not, the state is updated by adding `alloc`'s mailbox to list of deferred allocates, while leaving the free list empty. The response part of the result is `LATER`, indicating that the response will be determined later by a `send` operation on the mailbox.

Deallocate requests proceed as before when there are no deferred allocates. Otherwise, the block is sent to the first mailbox on the list, which allows the suspended alloc request to return its result. The new state has the mailbox removed from the deferred allocate list, and an acknowledgement is returned.

[2]In Id, threads are defined by data dependencies. Two threads with no data dependencies are independent, and may execute in parallel.

```
MANAGER patient_free_list = create_patient_free_list block
 typeof alloc    = patient_free_list -> block;
 typeof dealloc  = patient_free_list -> block -> ack ;
 typeof query    = patient_free_list -> num;
 rep ((list block),(list (MAILBOX block)))
  {
    % the state is pair: (free'list,list of deferred allocs)
    def create_patient_free_list block = (break_up_mem block),nil;

    def_handler alloc (nil        ,allocs)   = (nil,MAILBOX:allocs),LATER
        |       alloc ((block:lst),nil   )   = (lst,nil)           ,block;

    def_handler dealloc (lst,nil)      block = ((block:lst),nil),ok
        |       dealloc (nil,M:allocs) block = {send M block
                                                In
                                                  (nil,allocs),ok};

    def_handler query   (lst,allocs)         = (lst,allocs),(length lst)
  }
```

Figure 3: The Patient Memory Manager

2.4 Mailboxes, Signal, and Wait

Although mailboxes are very powerful, they open the door to a new class of programming errors. Mailboxes could be sent multiple values or lost, or a request handler could release its mailbox *and* return a result. Type-checking may uncover some of these errors at compile-time, but most can only be detected at run-time, if ever. Further, this use of mailboxes makes `alloc` and `dealloc` responsible for scheduling, as well as memory management, which is clearly not modular.

One way to reduce these errors is to encapsulate scheduling operations in a manager. For example, consider the `wait` and `signal` monitor operations. These allow a condition on the state to be established before a handler is executed. If the condition is false, `wait` suspends the result of the request and releases the state, unchanged. When the state is modified, the `signal` operation resumes the handler which returns its result.

Figure 4 shows how simple versions of `wait` and `signal` can be encapsulated in a manager for the patient free list. Calling `wait` simply enqueues the mailbox on the queue of waiting processes and suspends its result with `LATER`. `Signal` takes a new free list as an argument: if there are no waiting allocs, it simply returns it for the new state. Otherwise, it sends the head of the free list to the first mailbox, pops the mailbox, and returns the remainder as the new state.

Although this version of a waiting queue is specific to the problem at hand, more general versions can be constructed similarly. This illustrates an important point: managers allow queuing to be incorporated in a modular, programmatic way. Because waiting queues are managers, different algorithms can employ different paradigms. These include FIFO (first-in, first-out), priority, and predicate-based scheduling. It can be extended to also include a "notify" paradigm[6], where the state is returned immediately and the signal is requeues the request.

Note that type of composition and modularity cannot be achieved with monitors; wait and signal operations implicitly modify the resource lock. This has several ill-effects, as amply described in the literature[3, 6, 7]. Simple monitor nesting, as illustrated above, traditionally presents problems when the nested monitor suspends. Managers allow issues like this to be resolved explicitly and programmatically, as appropriate to the algorithm.

```
manager waiting_q = create_waiting_q num
  typeof wait   = waiting_q -> block;
  typeof signal = waiting_q -> (list block) -> (list block)
  rep (list (MAILBOX block))
  { def create_waiting_q n = nil;
    def_handler wait q = (mailbox:q),later;
    def_handler signal nil   lst         = nil,lst
       |        signal (m:q) (block:lst) = {send m block;
                                            In
                                              q,lst};
   };
MANAGER patient_free_list = create_patient_free_list block
 typeof alloc    = patient_free_list -> response;
 typeof dealloc  = patient_free_list -> block -> ack ;
 typeof query    = patient_free_list -> n;
 rep ((list block),waiting_q)
  {     def create_patient_free_list block = (break_up_mem block),nil;

    def_handler alloc (nil,      q)    = (nil,q),wait q
        |       alloc ((block:lst),q) = (lst,q),block;

    def_handler dealloc (lst,q) block = ((signal q (block:lst)),q),ack;

    def_handler query   (lst,q)  = (lst,q),(length lst)
  };
```

Figure 4: A Wait and Signal Manager

2.5 Composition

Individual managers can support a wide range of programs. However, many applications may require several managers working in concert. The example above illustrates a simple nested manager structure. Since managers as first-class objects, networks of managers can be defined using standard data structures such as lists and arrays. Furthermore, managers can form arbitrary graph structures by storing manager references in the state variable. These structures can be made dynamic by operations that alter the references during execution. This section gives examples of these types of composition.

2.5.1 The Buddy System

Consider a buddy system memory manager, structured as follows. Let there be a vector v of free lists, where element i is a manager for memory blocks of size 2^i. Allocate and deallocate requests are directed to the appropriate manager. Allocation removes the head of the list and returns it; however, if the list is empty, an allocation request is issued to the manager of the next larger size block. This recurs until a block is found or the end of the vector is reached. When the block is returned, it is split: half is returned as the result, and half is placed on free list. Deallocate requests coalesce blocks in a similar, recursive fashion, forwarding successively larger blocks until no buddy is found. The free list manager (without boundary conditions) is given in Figure 5.

Note that although both `alloc` and `dealloc` are recursive, they have different locking behavior. In `alloc`, both the state and result depend on the result of the recursive call. Therefore, free list i is locked until the recursive call to free list $i+1$ returns.[3] `dealloc`, on the other hand, releases

[3]This is appropriate, since another allocate request would simply duplicate the work currently outstanding.

```
manager buddy_list = make_buddy_list block
  typeof alloc   = buddy_list -> num -> block;
  typeof dealloc = buddy_list -> block -> num -> ack
  rep (list block)
     {
      def make_buddy_list mem = break_up_mem mem;
      def_handler alloc block:mem_q i = mem_q,block
        |         alloc nil       i  =
                      {big_block = alloc v[i+1] i;
                       top,bottom = split big_block i;
                       In
                       (top:mem_q),bottom};
      def_handler dealloc mem_q block i =
          if (free_buddy? block) then
            {new_q     = remove_buddy block mem_q;
             big_block = (coalesce block i);  % join memory and deallocate
             In
               new_q,(dealloc v[i+1] big_block (i+1))}
          else (block:mem_q),ok
       };
```

Figure 5: The Buddy System

its state immediately. The recursive call only delays the result, an acknowledgement indicating the deallocate has completed. Monitors only support the first type of locking scheme; a monitor implementation of the above would delay the state until coalescing completes.[4]

2.5.2 Manager Graphs

Trees Managers have been structured as trees to implement union-find[1] and parallel priority queue[5] algorithms. In manager trees, the state contains pointers to other managers. These pointers are traversed and modified by manager operations, allowing the structure of the trees to dynamically change. In the union-find algorithm, this enables path compression, a technique that drastically reduces the cost of set membership (find) operations. In the parallel priority queue, tree balancing is done by recursive calls on the manager links. Therefore, both of these algorithms demonstrate good amortized time complexity. In addition, the latency of individual operations is low, because the state is available before the recursion has terminated. Trees also improve availability of the resource, since locking is localized to the subtree being modified.

Cyclic Graphs As a final example, consider a simple, recursive algorithm on a directed graph. The problem is to count the number of distinct nodes reachable from a node. One natural solution is as follows: at each node, leave a mark and recursively count all neighbors of the node. Add one to the sum of the counts returned. Whenever a mark is reached, return zero.

```
manager node = make_node (list node)
  typeof count = node -> num
  rep (bool,(list node))
  {make_node neighbors = (false,neighbors);
   def_handler count (true,neighbors)  = (true,neighbors),0
      |       count (false,neighbors) = (false,neighbors),
```

[4] Alternatively, the recursive call could be moved outside the entry procedure, but this results in fragmented control flow.

```
                                        1+(sum (map count neighbors))
    };
```

This simple manager cannot be written as a monitor without potential deadlock, since the lock is held until the result is returned. A monitor solution requires the recursion to be moved to an external procedure, and reduces the monitor to a test-and-set operation. In the process, abstraction and modularity are destroyed.

2.6 Programming Paradigms

These examples illustrate the variety of concurrent programming paradigms managers support. The separate, parallel computation threads within a manager operation can be controlled to support a broad class of algorithms. In the simplest case, both threads return immediately. In this case, managers provide mutual exclusion, while parallelism improves efficiency and availability. Including mailboxes that suspend one of the threads[5] presents two additional cases:

1. **Suspended Result:** requests can be reordered. Examples include the patient free list, and any type of scheduler.
2. **Suspended State:** resource locked when result returned. Presumably the result contains a mailbox so the state thread can be resumed. This allows mutual exclusion over a set of resources, but in general breaks the abstraction.

Composite managers allow recursive structures with distributed locks. Recursive calls on these structures can delay one or both of the threads, resulting in the following three cases:

1. **Recursive State, Immediate Result:** the result is returned immediately while the structure is recursively updated. Note that the new state will be returned immediately by the recursive call, so the resource will only be locked for one recursive call, even if the call chain is deep. This implements a technique called *lock-coupling*, where a resource lock is not released until the next is acquired.
2. **Immediate State, Recursive Result:** Here the resource is available while the result is being recursively computed. Examples are the `dealloc` handler in the buddy system, and the cyclic graph manager.
3. **Recursive State and Result:** Resource and result are unavailable until the end of the recursive call. Unlike the first case, resources are locked all along the call chain. This is useful for excluding entry into the chain until the recursion finishes, as in the buddy system example. Note that this is the only form of recursion supported by monitors.

2.7 Summary

Like monitors, managers provide encapsulation. The state variable and its associated operations are defined in a single construct. The external interface to a manager is clearly defined, and request handlers can be written as as state transition functions. Further, the state variable is completely hidden from the interface.

[5] Suspending both threads results in deadlock.

Managers are more expressive than monitors. Mailboxes allow requests to be deferred and reordered with the addition of simple queueing operations. Managers can be composed into networks of asynchronous processes in an abstract and modular way. Recursive operations on such structures can explicitly control resource availability and deadlock avoidance.

3 Implementation

The implementation of managers needs to be very efficient because we use them for frequently accessed, critical resources (for example, the memory manager in Id's own runtime system). There are two key issues:

- ensuring mutual exclusion, i.e., two requests to the same manager must acquire the manager state serially, and
- the implementation of `MAILBOXes`, `LATER`, `SENDs`, etc., for scheduling managers.

It is clear that we need some notion of a *locked cell* for mutual exclusion. Further, to avoid busy-waiting, we need some way of queuing waiting processes on locked cells and awakening a waiting process when the cell is unlocked. Also, because of fine-grained concurrency, these operations must be very efficient. All this is standard; what is novel is the way these things blend in naturally with existing dataflow architectures [2]. The key features of a dataflow architecture are these:

- A dataflow processor is capable of supporting thousands of threads simultaneously, each of which is described completely by a *tag* (or *continuation*). The processor can switch between threads on every instruction.
- Memory reads are done using split-phase transactions. Each read is accompanied by a tag describing the continuation that expects the datum. Meanwhile, the processor executes other threads. The memory responds with a datum *and the tag*, so that when it arrives, the processor knows exactly what to do with it.
- In the memory, every location has extra "presence" bits designating it as *empty* or *full*. When reads arrive at an empty location, the memory controller queues the accompanying tags at that location. When the location is written later (making it full), the controller also dispatches the datum to all the tags that were waiting there for it. The space for the deferred lists is managed locally by the memory controller. This behavior is referred to as *I-structure* semantics.

With this basis, a small extension is sufficient to implement managers. Instead of just reads and writes, the memory controller also accepts `read_and_lock` and `write_and_unlock` requests. As in an ordinary read, a `read_and_lock` request is accompanied by a tag, and if the cell is empty, the tag is queued there. However, if the cell is full, instead of just returning the datum with the tag, the cell is also redesignated as empty. If an empty cell has no deferred readers, a `write_and_unlock` simply writes the datum there and marks it full. If there are deferred readers, however, it leaves the cell empty; instead, it dequeues one of the waiting tags and sends it the datum.

Figure 6 summarizes lock operations on I-structure memory units, showing the contents of three cells: the first is unlocked, the second is locked with no waiting tags, and the third is locked with

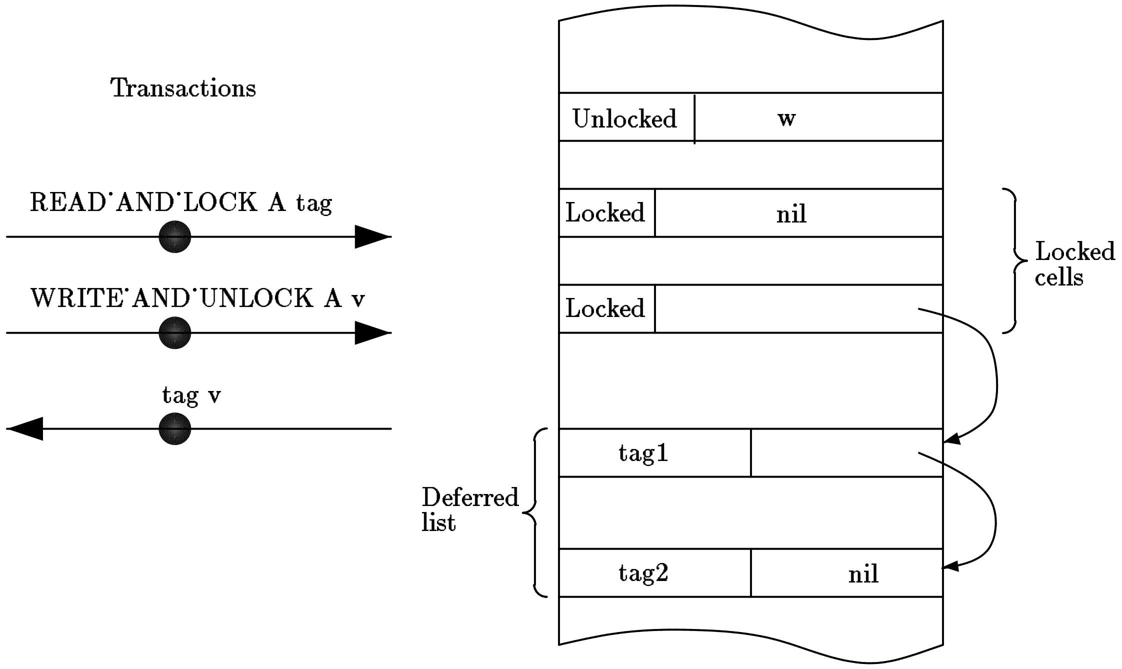

Figure 6: I-structure memory

a deferred list. Note that there is no busy-waiting at all. The thread that requires the result of the `read_and_lock` travels to remote memory, waits there in a deferred queue, and comes back to the processor to resume executing— there is no polling. Meanwhile, the processor executes other threads.

Implementing Managers and Simple Request Handlers

A manager is implemented as a single locked cell containing the object that represents its state. Scalar objects may be stored directly while non-scalar objects may be represented by pointers. Consider the simple memory manager, repeated here in outline:

```
MANAGER free_list = create_free_list block
 ...
 { def create_free_list mem = break_up_mem mem;
   def_handler dealloc lst blk  = (blk:lst),ok;
   ...
 } ;
```

The constructor simply allocates a locked cell containing the initial state:

```
create_free_list mem = allocate_locked_cell (break_up_mem mem);
```

The `dealloc` handler is desugared into:

```
def dealloc cell blk = {  lst = read_and_lock cell ;
                          new_state,result = blk:lst, ok ;
                          write_and_unlock cell new_state ;
                        In
                          result } ;
```

The first argument is the manager, i.e., the locked cell. We extract the free list from the cell, thereby locking it; we compute the new state (`blk:lst`), and place it back in the state cell, thereby unlocking it. Independently, we compute the result (`ok`) and return it. If two threads invoke handlers on the same manager at the same time, only one of them succeeds at the `read_and_lock`; the other one gets deferred. The winner computes the new state and, when replacing it in the cell using `write_and_unlock`, the new state is passed on to the second thread. Thus, we achieve the goal of mutual exclusion and serialization on the state variable.

With this implementation, the body of `dealloc` is very efficient, amounting to as few as three or four instructions. The only remaining overhead is function call overhead, which is often eliminated by the inline substitution optimization.

Implementing Scheduling Request Handlers

`MAILBOXes` are implemented using cells with standard I-structure semantics. A mailbox is an empty cell. The `LATER` construct is simply a read against this cell, which therefore gets deferred as explained earlier. The `SEND` construct is simply a write to the cell, which therefore frees the deferred thread waiting there. Thus, the cost of a mailbox is one storage cell. A mailbox is passed around by passing a pointer to this cell.

4 Discussion

4.1 Experience with Managers

We have used managers in a variety of Id application and systems programs. Our experience shows three things: encapsulation is critical to correct, modular programs; separate state and result threads do increase parallelism and resource availability; and a low-overhead implementation encourages the use of many, local managers in composite structures.

In an implicitly parallel language like Id, execution order is often hard to predict or control. Manager encapsulation and implicit mutual exclusion greatly simplify control issues without unnecessarily restricting parallelism. For example, when the buddy system was initially implemented with locks, the early release of a lock caused coelescing to intermittently fail. When recast as managers, the recursive call structure ensured exclusion when the buddy was checked, eliminating the error.

Of course, encapsulation does not eliminate deadlock. An initial version of the union-find algorithm for an image analysis program deadlocked when attempting to union two elements already in the same set. This was remedied by locking elements in a canonical order. Such considerations will always be present in parallel, shared resource programs.

Modularity was improved by distinguishing between interface operators and request handlers as suggested in [6]. The union operation cited above tested its elements before locking one by calling a request handler; therefore, union was an interface operator but *not* a request handler. The parallel priority queue demonstrated the converse: the tree balancing function was a completely local handler, called by the interface operators but not visible at the interface. A corallary of this distinction is that request handlers can be made short, by moving state insensitive computations and checks outside the handler. This increases resource availability, as suggested by [3].

Experience with the buddy system, parallel priority queue, and image analysis programs indicates that parallel state and result threads do increase resource availability. Although all of these perform several hundred instructions per operation to reorganize the structure, they release their resources before completing execution. The parallel priority queue has the highest availability, releasing after a single function call (a few dozen instructions). The buddy system has good amortized availability, since two memory blocks are retrieved whenever a queue runs empty. When there are many, distinct sets, union-find is also highly available. However, in the presence of a few, large sets, union-find suffers from contention at the roots.

Parallel state and result threads allowed scheduling managers to be more modular. In a priority printer scheduler, small files are printed before large files, even if the large file arrives first. The manager for this includes a separate scheduling manager, so the policy can be altered independently from the rest of the program.

Finally, low overhead is central to successful use of managers. Since managers are implemented as lock operations, there is no temptation to use locks directly to improve performance. Further, low overhead encourages composite manager structures, as experienced in the three examples cited above. The image analysis program was originally a highly sequential algorithm using one centralized data structure. When recast as a distributed, union-find structure, much more parallelism was exposed.

5 Conclusion

Managers are an effective construct for managing shared resources in a parallel system. Like monitors, managers encapsulate operations on a shared resource. Managers extend the monitor paradigm to allow the succinct expression of concurrent state-sensitive computations, yet provide control and flexibility in their use. They have been used for a variety of applications, and provide a foundation for correct, efficient parallel algorithms on shared resources.

Acknowledgements: The authors are indebted to the members of the Computation Structures Group at M.I.T. for their technical comments and suggestions. Arvind has been instrumental in the inception and development of these concepts. Richard Soley and Ken Steele developed locked storage cells, which are instrumental to efficient implementation. Numerous discussions with Ken Traub, Steve Heller, Jonathan Young, Jamey Hicks, Steven Brobst, Greg Papadopoulos, Arthur Altman, and David Culler significantly improved these ideas.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Tree Structures for the UNION-FIND Problem*, pages 139–145. Addison Wesley Publishing Company, Reading, Massachusetts, 1974.

[2] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 1989 (to appear). An earlier version appeared in *Proceedings of the PARLE Conference, Eindhoven, The Netherlands*, Springer-Verlag LNCS Volume 259, June 15-19, 1987.

[3] R. C. B. Cooper and K. G. Hamilton. Preserving abstraction in concurrent programming. *IEEE Transactions on Software Engineering*, 14(2):258–263, February 1988.

[4] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 10(10):549–557, October 1974.

[5] D. W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, January 1989.

[6] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[7] A. M. Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5–7, July 1977.

[8] R. S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[9] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, August 1988.